Alessandro Monticelli

Sistemi Operativi

A.A. 2022/23

Sommario

Premesse 8
Licenza e disclaimer 9
Il presente 9
Introduzione ai Sistemi Operativi 10
Hardware di un Computer 10
Supporto ai Processi da parte del livello ISA 12
Registri disponibili ai Programmi nei processori Intel IA-32 12
Supporto ai modi di esecuzione della CPU per i processi 13
Supporto al Sistema da parte del livello ISA 15
Processi 18
Chiamata ad Interrupt da Programma 18
Chiamata ad Interrupt da Programma – Cambio di Contesto 19
Il meccanismo delle System Calls 22
Instruction Set Architecture (ISA) e Sistema Operativo 22
Interrupt e Eccezioni 22
Eccezioni 22
Interrupt 23
Sistemi operativi Multitasking 24
Sistemi Operativi “Interrupt Driven” 25
Sistema Operativo – Scheduler 25
Passaggio di stato dei processi 25
Domande riassuntive 27
Richiami di Linguaggio C 31
Moduli e variabili 32
Variabili Globali e Specificatore extern 32
Protezione degli accessi esterni al modulo 33
Variabili globali e Specificatore static 33
Protezione dagli accessi esterni alla funzione 34
GCC e Makefile 36
GCC – Opzioni del preprocessore 36
Feature test macros 36
Makefile 40

Concetti di dipendenza e di albero delle dipendenze 40
Condizione di rigenerazione di un target 41
Struttura del Makefile: Make Rules 42
Sintassi del Makefile 42
Ordine delle regole e costruzione dell'albero delle dipendenze 43
Esecuzione delle command list 44
Esecuzione del make 44
Best practices per la costruzione di Makefile 45
Target fittizi (.PHONY) 46
Variabili nei Makefile 47
Conclusioni 48
Domande riassuntive 49
Librerie 51
Librerie statiche e condivise 51
Thread e POSIX Thread 52
I Thread 52
Contesto di esecuzione di un Thread 52
Operazioni atomiche: definizione 53
Operazioni atomiche 54
Operazioni compare-exchange o compare-and-swap 54
System Call non rientranti 55
Thread Safe Call 55
POSIX Thread 56
POSIX Thread APIs 56
Compilazione e linking 57
API per creazione ed esecuzione di pthread 57
Pthread identifiers 59
Passaggi di argomenti a pthread 59
Terminazione e risultato di un pthread 60
Attesa della terminazione di un pthread 61
Mutua esclusione e sincronizzazione 63
Race condition 63
Mutex Variables 63

Operazioni con Mutex ... 64
Creazione e distruzione di variabili Mutex ... 64
API per mutua esclusione ... 64
Condition Variables ... 65
API per Condition Variables ... 68
Creazione e distruzione di Condition Variables ... 68
Attesa ... 68
Abilitazione al risveglio ... 68
Domande riassuntive ... 70
Gestione della memoria ... 74
Introduzione ... 74
Binding Address ... 74
Indirizzi logici e indirizzi fisici ... 75
Esempi di MMU ... 75
Registro di rilocazione ... 75
Registro di rilocazione e limite ... 76
Allocazione di memoria ... 76
Definizioni ... 76
Allocazione a partizioni fisse ... 77
Gestione della memoria ... 77
Frammentazione interna ... 77
Allocazione a partizioni dinamiche ... 77
Frammentazione esterna ... 77
Compattazione ... 78
Selezione del blocco libero ... 79
First Fit ... 79
Best Fit ... 79
Worst Fit ... 79
Strutture dati ... 79
Paginazione ... 80
Dimensione delle pagine ... 81
MMU per la paginazione ... 82
Implementazione della page table ... 82

Segmentazione 82
Confronto 83
Supporto hardware per segmentazione 84
Segmentazione e condivisione 84
Segmentazione e frammentazione 85
Segmentazione e paginazione 85
Memoria virtuale 85
Pager/Swapper 87
Algoritmi di sostituzione o rimpiazzo 89
On demand puro 90
Sostituzione delle pagine 91
Dirty Bit 91
Domande riassuntive 92
File system 95
Concetti principali 95
File 95
Attributi 95
Directory 96
File in uso 96
Struttura dei file 97
Metodi di accesso 97
Accesso Sequenziale 97
Accesso diretto 98
Accesso Indicizzato 98
Strutture del File System 99
File Control Block 99
Allocazione dei blocchi 99
Allocazione Contigua 100
Allocazione concatenata 101
Clustering 101
Allocazione indicizzata 102
Architetture per la virtualizzazione di sistemi 103
Virtualizzazione o Emulazione 103

Docker ... 106
Installare applicazioni ... 107
Salvare i cambiamenti ... 107
Esecuzione in background ... 107
Kubernetes ... 108
Applicazioni per Kubernetes ... 108
Cluster Kubernetes ... 109
Cluster Kubernetes con nodi virtuali ... 110
Kubernetes as a Service ... 111
Container as a Service ... 112
Serverless o Function as a Service (FaaS) ... 113
Interfaccia a utente caratteri (Incompleto) ... 114
Nozioni sull'uso del terminale ... 114
Interpretazione dei comandi bash: Expansions ... 114
Utenti e Gruppi ... 117
Permessi di file e directory ... 118
Visualizzazione permessi di file e directory ... 120
Permessi speciali ... 120
Subshell ... 121
Variabili ... 122
Visibilità delle variabili ... 124
Variabili vuote o non esistenti ... 125
Riferimenti indiretti a variabili ... 125
History expansion ... 125
Comando set ... 126
Avvio della shell bash ... 127
Parametri a riga di comando ... 128
Separatore di comandi e delimitatore di argomenti ... 129
Quoting di singoli caratteri ... 129
Brace Expansion – generazione di stringhe ... 129
Annidamento della brace expansion ... 131
Brace expansion con Sequence expression ... 131
Tilde expansion ... 131

Wildcards 133
Pathname substiution 133
Comandi ed eseguibili utili 134
Parametri a riga di comando passati al programma 137
Valutazione Aritmetica di espressioni tra interi 138
Exit Status 139
Prova d’esame 24/01/2023 142
Crediti 145

Premesse

Questo è un riassunto del corso di Sistemi Operativi del CdL di Ingegneria e Scienze Informatiche (Università di Bologna) tenuto dal prof. Vittorio Ghini nell'A.A. 2022/23

Licenza e disclaimer

Quest'opera è stata rilasciata con licenza Creative Commons Attribuzione - Non commerciale - Condividi allo stesso modo 3.0 Unported. Per leggere una copia della licenza visita il sito web http://creativecommons.org/licenses/by-nc-sa/3.0/ o spedisci una lettera a Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Il presente documento è frutto unicamente della mia attività di studio autonomo ed è distribuito così com'è, l'autore declina ogni responsabilità derivante da un uso improprio del materiale presente in questo documento.

Tutti i diritti sono riservati. In caso di mancata attribuzione di crediti o diritti d'autore vi chiedo gentilmente di contattarmi all'indirizzo mail: alessandr.monticell4@studio.unibo.it in modo da poter rimediare.

Introduzione ai Sistemi Operativi

Hardware di un Computer

I componenti principali di un computer sono:

- CPU: esegue istruzioni, effettua calcoli, controlla input/output, coordina spostamento Dati.
- BUS: trasferisce i dati tra la memoria e gli altri dispositivi.
- Memoria RAM (volatile): mantiene programmi (sequenza di istruzioni) quando devono essere eseguiti, e i dati che i programmi usano, ma solo finché il computer è acceso.
- Memoria ROM (permanente): contiene il BIOS, le istruzioni per fornire i servizi base, usate dalla CPU anche nel momento dell'accensione del computer (il BOOT).
- Dischi: mantengono enormi quantità di programmi e dati, anche dopo lo spegnimento del computer.
- Sono inoltre presenti alcune periferiche di I/O (Input/Output).

Il sistema operativo mette a disposizione diversi servizi:

- Protezione della CPU – sfrutta la capacità della CPU di operare in due diverse modalità:
    - Modalità kernel: Permette di utilizzare tutte le istruzioni ISA, tutti gli indirizzi di memoria, tutti i registri, tutte le periferiche.
    - Modalità utente: Limita le istruzioni eseguibili, l'accesso alla memoria, ai registri e alle periferiche.
- Gestione della Memoria:

- Indirizzamento, segmentazione, protezione, memoria virtuale, paging

- Gestione I/O:
  - Periferiche (tastiera, video, rete, dischi), Storage (filesystem)
- System Calls, Librerie di sistema, utilities:
  - Servizi messi a disposizione in diverse modalità. Le syscall sono disponibili mediante chiamate a interrupt (effettuabili in assembly).

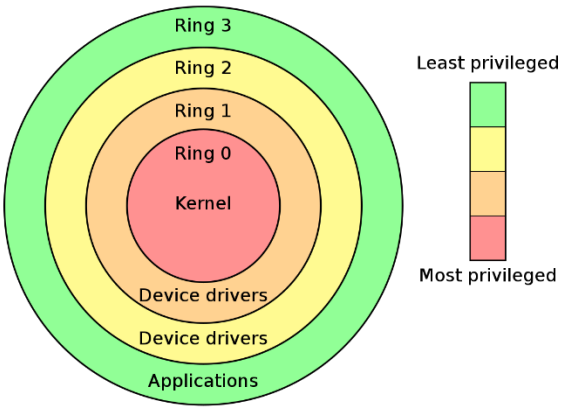


Figura 1: Instruction Set Architecture of IA-32 – Privilege levels (Rings)

Organizzazione della Memoria

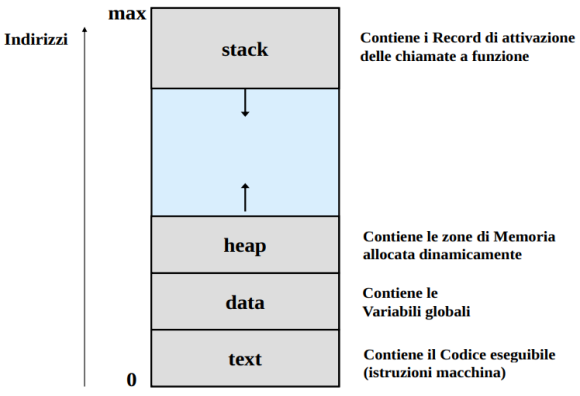


Figura 2: Organizzazione della Memoria usata dalla parte utente dei Processi

Supporto ai Processi da parte del livello ISA

I processori moderni hanno caratteristiche comuni che permettono di fornire ai livelli superiori un'interfaccia di programmazione denominata livello ISA (Instruction Level Architecture) variabile da processore a processore ma avente alcune analogie:

Modalità di protezione (modo kernel, modo utente) e istruzioni per lo switch tra i due modi.

ALU, unità aritmetico-logica e istruzioni per comandarla.

Registri general-purpose e istruzioni per scambio dati tra registri della CPU, memoria e periferiche.

Registri specializzati (Il cui nome varia a seconda del processore):

- Extended Instruction Pointer (EIP) serve a formare l'indirizzo in memoria della prossima istruzione da eseguire (detto Program Counter o PC). Contiene l'offset della prossima istruzione da eseguire, rispetto all'inizio del segmento di codice.
- Program Status (PS) detto anche Registro di stato (Status Register)
- Stack Segment Register (SS). Punta all'inizio dello stack.
- Code Segment Register (CS). Punta all'inizio della sezione text, il codice eseguibile.
- Data Segment Register (DS). Punta all'inizio della sezione data, al cui interno sono contenute le variabili locali.

Registri disponibili ai Programmi nei processori Intel IA-32

I registri sono memorie ad alta velocità inserite all'interno della CPU.

In un'architettura Intel a 32 bit (in questo esempio IA-32) sono presenti:

- Otto general-purpose registers a 32bit.

- Sei segment registers a 16bit
- Flag del Processor Status (EFLAGS) e Instruction Pointer (EIP)

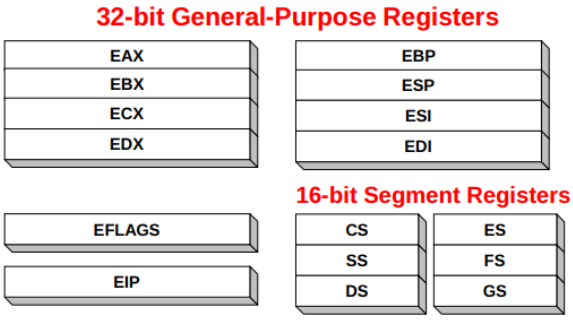


Figura 3: Registri di un processore Intel IA-32

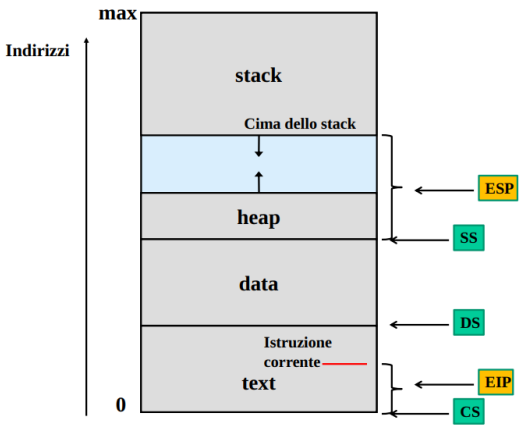


Figura 4: Uso dei registri di segmento e offset nella parte utente dei Processi

Supporto ai modi di esecuzione della CPU per i processi

Quando un processo esegue una chiamata ad interrupt, la CPU entra nella modalità di esecuzione kernel e deve iniziare uno, o più, stack "di sistema" separato rispetto allo stack utilizzato per le normali chiamate a funzione.

Ciò permette al kernel di separare e proteggere i record di attivazione utilizzati nelle chiamate ad interrupt.

Ciascun processo, perciò, mantiene uno stack per ciascun privilegio della CPU.

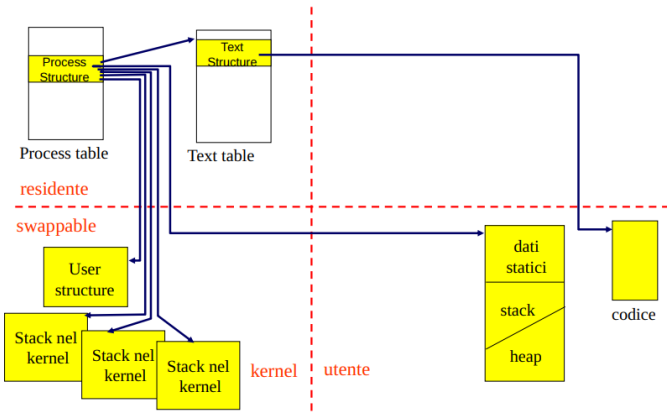


Figura 5: Immagine in memoria dei Processi

In esecuzione su un computer possono solitamente trovarsi:

- Processi (i programmi lanciati dagli utenti)
- Alcuni gestori degli interrupt:
  - Per eventi asincroni originati dalle periferiche (scadenza timer, arrivo caratteri da tastiera, arrivo messaggio dal DMA (Direct Memory Access), da usb, da rete)
  - Per eventi sincroni originati dai processi (invocazioni di System Call mediante chiamate esplicite all'istruzione che esegue interrupt software).
- Alcuni gestori delle eccezioni:
  - Per eventi sincroni generati dalle istruzioni eseguite dai processi.

I processi, gli interrupt e le eccezioni eseguono ciascuno in un proprio contesto denominato Task. I Task mantengono le informazioni sui segmenti di memoria utilizzati e sul valore concorrente dei registri principali.

Lo scheduler del sistema operativo determina il tempo di esecuzione per ciascun Task.

Salvare e ripristinare lo stato di un task consente di intervallare l'esecuzione di vari task, in modo che essi occupino la CPU ad intervalli (interleaving).

Supporto al Sistema da parte del livello ISA

I processori moderni offrono una API (livello ISA, Instruction Level Architecture) che fornisce supporto ai processi e all'implementazione del sistema operativo.

Il livello ISA offre infatti anche diversi registri dedicati al sistema operativo e numerose istruzioni macchina che consentono al sistema operativo di effettuare l'interleaving concedendo l'uso della CPU per piccoli intervalli di tempo ai processi che hanno bisogno di occupare la CPU, salvando e cambiando il contesto di esecuzione ogni volta che cambia il processo in esecuzione; di eseguire in un contesto separato e protetto le System Calls chiamate dal processo.

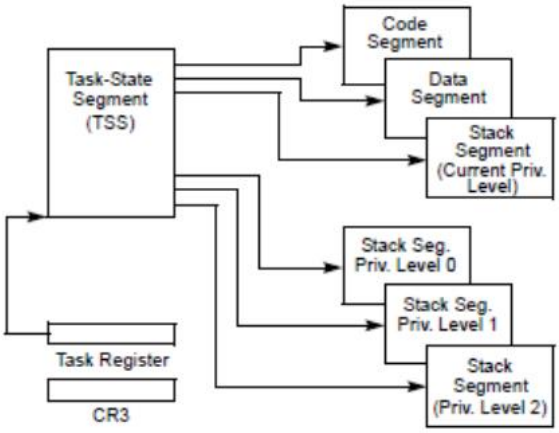


Figura 6: Struttura di un Task

31	15	0	
I/O Map Base Address	Reserved	T	100
Reserved	LDT Segment Selector		96
Reserved	GS		92
Reserved	FS		88
Reserved	DS		84
Reserved	SS		80
Reserved	CS		76
Reserved	ES		72
EDI			68
ESI			64
EBP			60
ESP			56
EBX			52
EDX			48
ECX			44
EAX			40
EFLAGS			36
EIP			32
CR3 (PDBR)			28
Reserved	SS2		24
ESP2			20
Reserved	SS1		16
ESP1			12
Reserved	SS0		8
ESP0			4
Reserved	Previous Task Link		0

Reserved bits. Set to 0.

Figura 7: Task-state segment

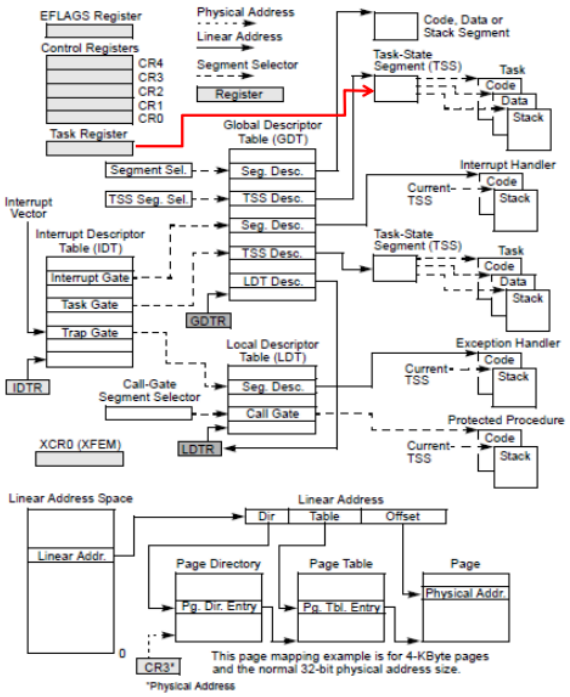


Figura 8: Registri a livello di sistema (E alcune strutture dati per la gestione dei task)

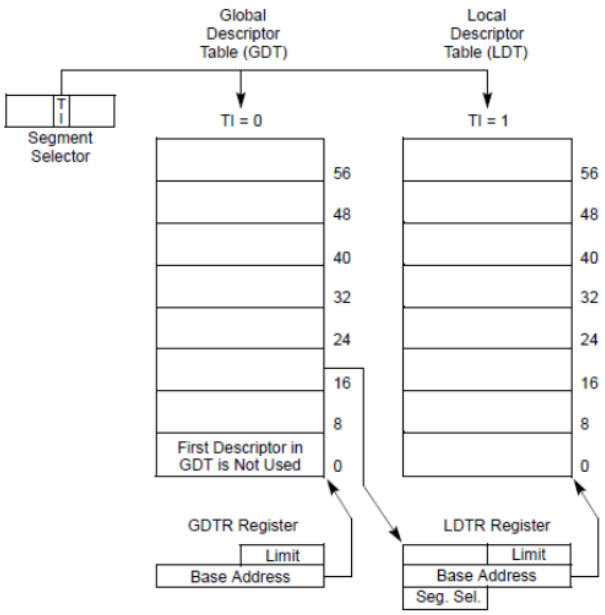


Figura 9: Descriptor Tables

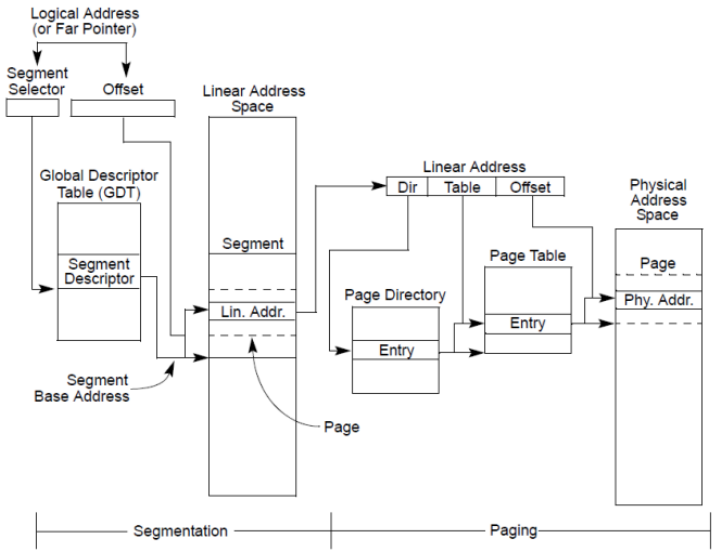


Figura 10: Segmentazione e Paginazione (IA-32)

(cc IA-32 Volume 3A System Programming Guide.pdf by Intel)

Processi

Un processo è l'istanza in memoria di un programma eseguibile.

Quando un programma viene lanciato, il loader (che è parte del sistema operativo):

- Crea i segmenti di memoria per il processo, popolando la tabella con le informazioni sui diversi segmenti.
- Carica in memoria il codice eseguibile del programma.
- Carica in memoria la sezione Data.
- Crea lo stack utente per l'esecuzione delle funzioni del programma.
- Crea gli stack di sistema per l'esecuzione delle System Calls che verranno invocate dal programma utente.
- Copia eventuali argomenti passati alla riga di comando, mettendoli a disposizione del main.
- Ordina alla CPU di eseguire la prima istruzione eseguibile (il main del programma).

Chiamata ad Interrupt da Programma

I programmi chiamano Interrupt per ottenere servizi dal sistema operativo o dal BIOS. Per chiamare un interrupt da programma, si usa l'istruzione assembly: `INT n` dove `n` è un numero intero maggiore o uguale a zero, usato come indice per accedere all'`n`-esima posizione del vettore degli Interrupt.

Il vettore degli Interrupt è posizionato a partire dall'indirizzo 0 in memoria, l'`n`-esimo elemento del vettore contiene diverse informazioni, tra cui l'indirizzo (CS:IP) della prima istruzione eseguibile della routine di gestione dell'interrupt di indice n.

Le routine sono fornite da BIOS e Sistema Operativo.

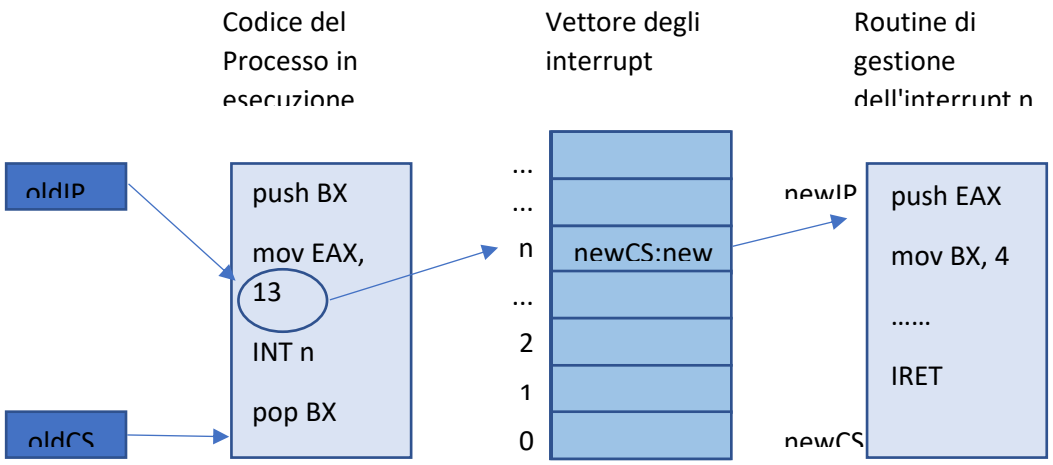


Figura 11: Gestione degli interrupt

Chiamata ad Interrupt da Programma – Cambio di Contesto

Se durante l'esecuzione il codice (Figura 12) di un processo sta utilizzando lo stack utente, quando la CPU esegue l'istruzione INT n:

- L'Instruction Pointer viene fatto puntare all'istruzione successiva ad INT.
- Il processore passa in modalità kernel con il livello di privilegio da utilizzare.
- Il processore salva i valori attuali dei registri SS, SP, CS, IP nello stack di sistema del processo che ha chiamato l'interrupt, cioè lo stack dedicato all'esecuzione in modalità kernel per quel processo.
- Il processore carica i registri SS e SP con i valori dello stack di sistema per quel livello di privilegio.
- Il processore carica i registri SP e IP con l'indirizzo della routine di gestione dell'interrupt trovato nell'n-esima posizione del vettore di interrupt.
- A questo punto (Figura 13) viene eseguita la routine dell'Interrupt, nello stack di sistema.

La routine dell'Interrupt termina con l'istruzione IRET (Interrupt RETurn) (Figura 14) che:

- Carica i registri IP, CS, SP e SS con i valori salvati sullo stack di sistema.
- Fa tornare il processore in modalità utente.
- L'esecuzione ricomincia (Figura 15) dall'istruzione del programma successiva all'istruzione INT n utilizzando lo stack utente del processo.

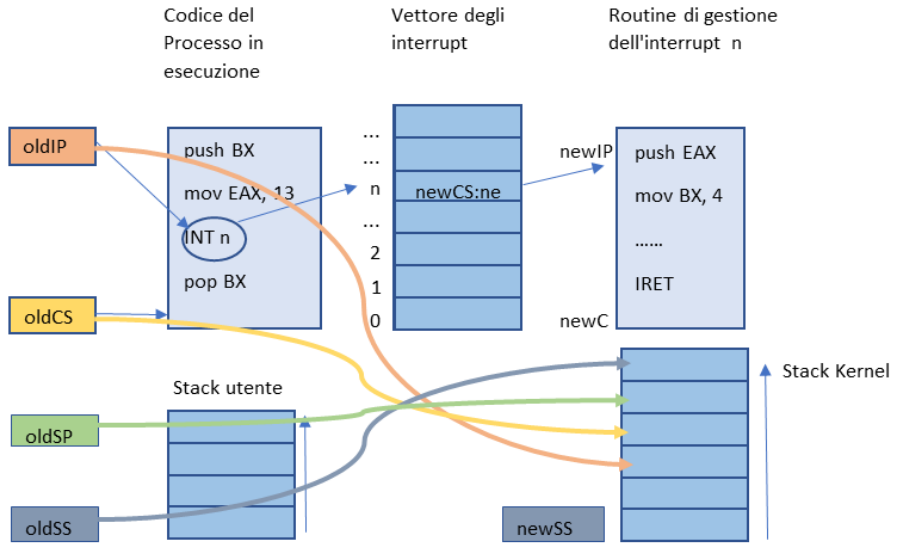


Figura 12: Prima chiamata a Interrupt

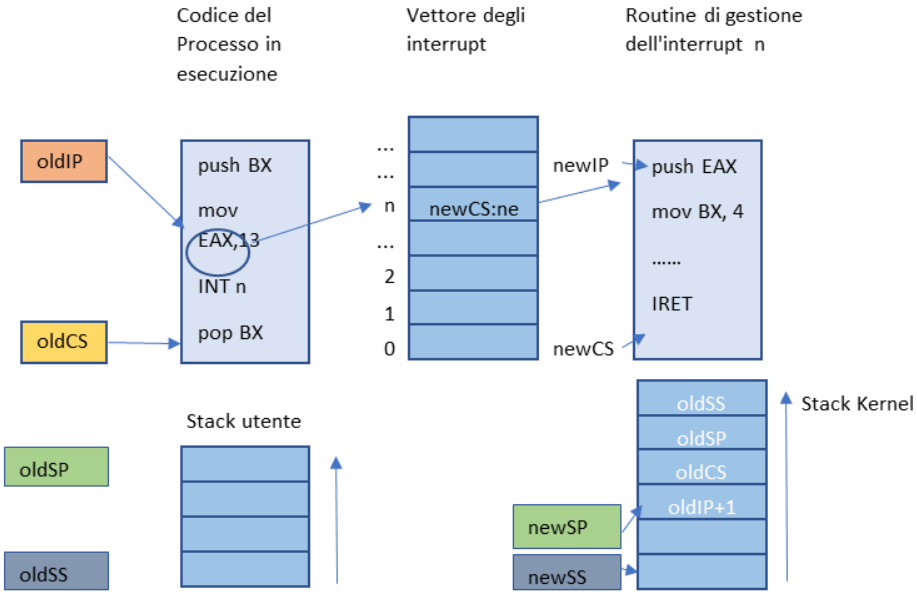


Figura 13: Dopo l'esecuzione di INT n

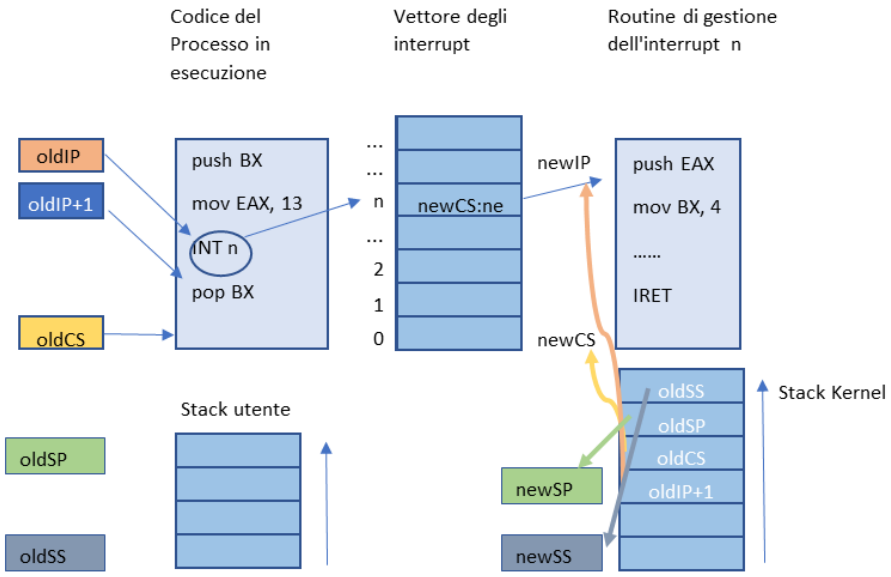


Figura 14: Prima dell'esecuzione di IRET

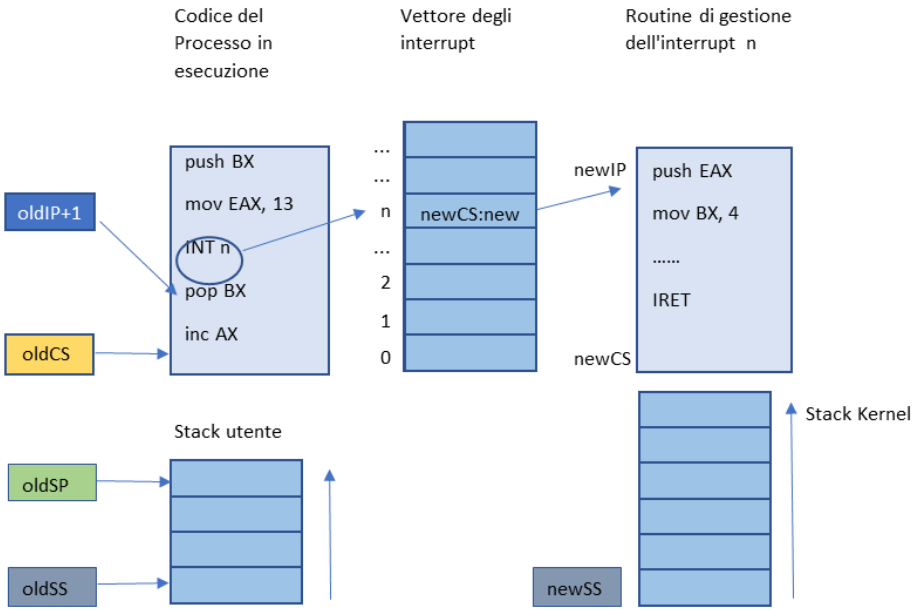


Figura 15: Dopo l'esecuzione di IRET

Il meccanismo delle System Calls

Un programma utente, durante l'esecuzione può invocare una System Call. La System Call è implementata nella routine di gestione di un certo interrupt n; quando essa viene invocata eseguendo una istruzione INT n, con un certo n intero, il programma passa i parametri all'interrupt mettendoli in registri (EAX e altri). A questo punto avviene la chiamata ad interrupt e la CPU passa dalla modalità utente alla modalità kernel. Il contesto viene salvato e la routine di gestione dell'interrupt (un puntatore a funzione nel vettore degli Interrupt) viene eseguita sullo stack del kernel. A questo punto la CPU ritorna in modalità utente e il contesto salvato in precedenza viene ripristinato.

Instruction Set Architecture (ISA) e Sistema Operativo

Interrupt e Eccezioni

Eccezioni (Exceptions) e Interruzioni (Interrupts) sono entrambe gestite mediante routine rintracciabili a partire dal Vettore degli Interrupt.

Eccezioni

Le eccezioni sono sincrone rispetto alle istruzioni eseguite dalla CPU, ovvero sono provocate involontariamente da un'istruzione macchina eseguita dalla CPU che provoca un errore o un caso particolare da gestire. I principali tipi di eccezioni sono:

- TRAP: L'istruzione che ha causato il TRAP viene momentaneamente interrotta, ma alla fine della gestione della TRAP viene portata a buon fine.
- FAULT: L'istruzione che ha causato il fault viene interrotta, e viene eseguita la routine di gestione. In un successivo momento l'istruzione verrà rieseguita da principio.

Esempi di FAULT exceptions: Page Fault, la pagina di memoria che contiene l'indirizzo di memoria a cui stiamo cercando di accedere è salvata nello swap su disco. La pagina deve quindi essere caricata in memoria. Prima o poi l'istruzione verrà nuovamente eseguita.

- ABORT: Causata da un errore irrimediabile, l'istruzione viene abortita e il processo viene killato.

Esempi di ABORT exceptions: Divisione per 0, Segmentation Fault (Il nome può trarre in inganno, tuttavia questo errore si verifica quando un processo tenta di accedere ad una porzione di memoria senza che abbia i permessi per farlo).

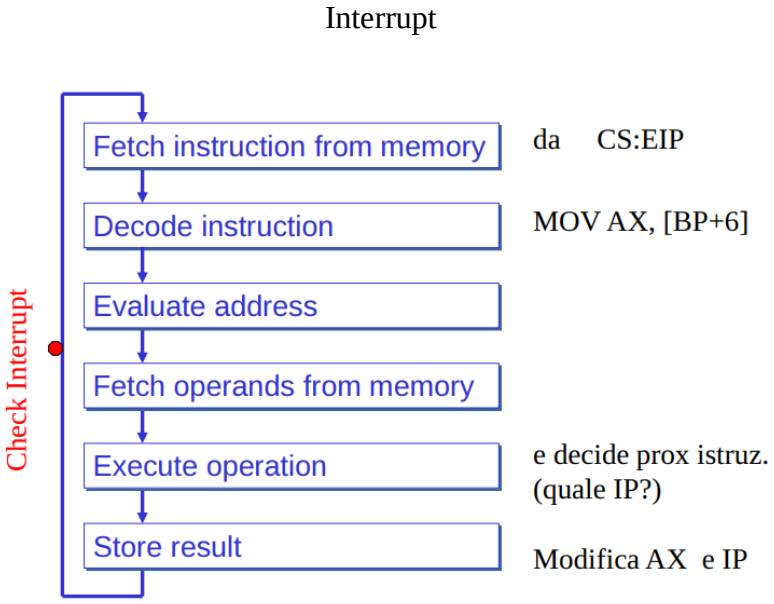


Figura 16: ISA – Instruction Processing – Ciclo di Neumann

Gli Interrupt possono essere sia sincroni che asincroni, a seconda che essi siano software o hardware. In particolare, gli interrupt software sono esplicitamente chiamati dalla CPU (mediante l'istruzione INT n). Gli interrupt hardware sono, invece, scatenati da una periferica che, mediante il BUS, avvisa la CPU che è accaduto un evento da gestire.

Gli Interrupt sono gestiti in base alla loro priorità. Gli interrupt che possono essere gestiti con poca urgenza (come l'immissione di un carattere da tastiera) sono detti mascherabili, quelli che devono essere gestiti immediatamente (ad esempio errori gravi durante la comunicazione tramite BUS) sono detti non mascherabili.

Durante l'esecuzione di una routine di gestione degli Interrupt, la CPU imposta un bit che disabilita la gestione degli Interrupt mascherabili.

Sistemi operativi Multitasking

Nei moderni sistemi operativi sono presenti in memoria diversi processi, thread, gestori di interrupt e di eccezioni (Multitasking), i quali cercano di eseguire contemporaneamente contendendosi l'uso della CPU (Time sharing).

Per alternare l'uso della CPU da parte dei programmi, il Sistema Operativo implementa uno scheduler che opera in modo da ridurre lo spreco di risorse al minimo. Per fare ciò l'esecuzione della CPU viene suddivisa in un determinato numero di quanti temporali (detti time slice). Allo scadere di un quanto temporale, il processo (o il thread) corrente viene interrotto e l'esecuzione passa ad un altro processo o thread.

Quando un processo (o un thread) richiede un'operazione di I/O, la CPU viene assegnata ad un altro processo, mentre le periferiche attendono il completamento delle operazioni di I/O, una volta terminate le operazioni da parte delle periferiche, il processo che l'aveva richiesta si rimette a disposizione dello scheduler, attendendo il suo turno per l'utilizzo della CPU.

I context switch (Il passaggio da un processo a un altro) avvengono così frequentemente che i programmi sembrano essere eseguiti contemporaneamente.

Sistemi Operativi “Interrupt Driven”

Gran parte delle funzionalità del kernel viene eseguita all’interno di una qualche routine di gestione degli interrupt, ossia al verificarsi di un evento che ha sollevato un interrupt o un’eccezione. Per questo motivo i Sistemi Operativi moderni sono detti “Interrupt Driven”.

Uno dei più importanti eventi che sollevano un interrupt è proprio lo scadere del time slice concesso a un processo dallo scheduler. Allo scadere del quanto di tempo, il timer di sistema genera un interrupt che risveglia lo scheduler. In seguito, lo scheduler decide quale altro processo (o thread) dovrà essere eseguito e passa il controllo a tale processo.

Sistema Operativo – Scheduler

Passaggio di stato dei processi

- Ammissione (new -> ready)

  Una volta creato e collocato in memoria dal loader, il processo è inserito nella coda dei processi ready.

- Dispatch (ready -> running)

  Il processo viene scelto dallo scheduler come il prossimo ad eseguire ed è messo in esecuzione

- Time interrupt (running -> ready)

  Il processo termina il suo quanto di tempo e torna in coda nella coda dei processi ready

- Request (running -> waiting)

  Il processo effettua una chiamata a sistema bloccante e viene inserito nell’opportuna coda di attesa.

- Event (waiting -> ready)

  L’attesa termina quando si verifica un evento e il processo torna nella coda dei ready

- Termination (running -> terminated)

  Il processo esegue l’ultima istruzione e termina.

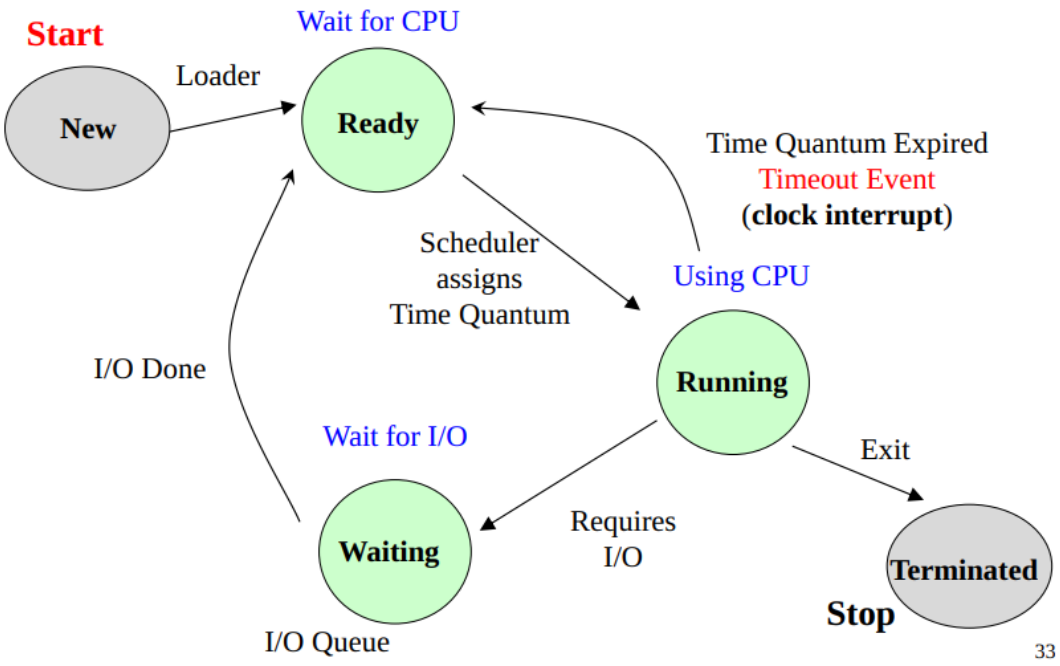


Figura 17: Ciclo di vita dei processi (context switch)

Domande riassuntive

- Descrivere l'organizzazione della memoria usata dalla parte utente dei processi.

  La memoria usata dalla parte utente dei processi è suddivisa in diverse aree:

  1. Stack, porzione di memoria dedicata alle chiamate a funzione e per lo storage di variabili locali
  2. Heap, area di memoria usata per la gestione della memoria dinamica (allocazione di memoria in più rispetto a quella richiesta inizialmente dal processo avviene nello heap)
  3. Codice, l'area di memoria che contiene il codice eseguibile
  4. Data, l'area di memoria che contiene le variabili globali e statiche del programma.

- Che cos'è un Task?

  Un Task è il contesto di esecuzione di un processo o di un thread. I Task mantengono le informazioni sui segmenti di memoria utilizzati e sul valore corrente dei registri principali.

- Un Task può avere più stack a sua disposizione? Perché?

  Si, in vari casi, ad esempio la creazione di diversi thread all'interno di un processo, ognuno con un proprio stack. Inoltre, ogni processo ha a disposizione uno stack per ogni privilege ring della CPU. Infine, alcuni sistemi mettono a disposizione diversi stack per la gestione di Interrupt e Eccezioni.

- Qual è la differenza tra un programma ed un processo?

  Un programma è una sequenza di istruzioni che possono essere eseguite dal calcolatore, un processo è un'istanza di un programma in memoria, con il proprio spazio di indirizzamento, i propri registri e il proprio stack.

- Quanti stack ha a disposizione un processo?

  Uno per ogni privilege ring della CPU.

- Cos'è una System Call?

Una System Call è un'interfaccia fornita dal sistema operativo che permette a un programma di richiedere un servizio del sistema operativo (Ad esempio l'accesso ad un file, o alle periferiche di I/O).

- Descrivi il meccanismo delle System Calls.

  Un programma utente, durante l'esecuzione può invocare una System Call. La System Call è implementata nella routine di gestione di un certo interrupt n; quando essa viene invocata eseguendo una istruzione INT n, con un certo n intero, il programma passa i parametri all'interrupt mettendoli in registri (EAX e altri). A questo punto avviene la chiamata ad interrupt e la CPU passa dalla modalità utente alla modalità kernel. Il contesto viene salvato e le routine di gestione degli interrupt (che altro non sono che puntatori a funzione nel vettore degli Interrupt) sono eseguite sullo stack del kernel. A questo punto la CPU ritorna in modalità utente e il contesto salvato in precedenza viene ripristinato.

- Cosa sono Interrupt ed Eccezioni?

  Un Interrupt è un segnale che un dispositivo (hardware o software) invia alla CPU per interrompere l'esecuzione corrente e richiedere l'esecuzione di una routine specifica come, ad esempio, la lettura di un carattere da tastiera, la stampa su schermo eccetera.

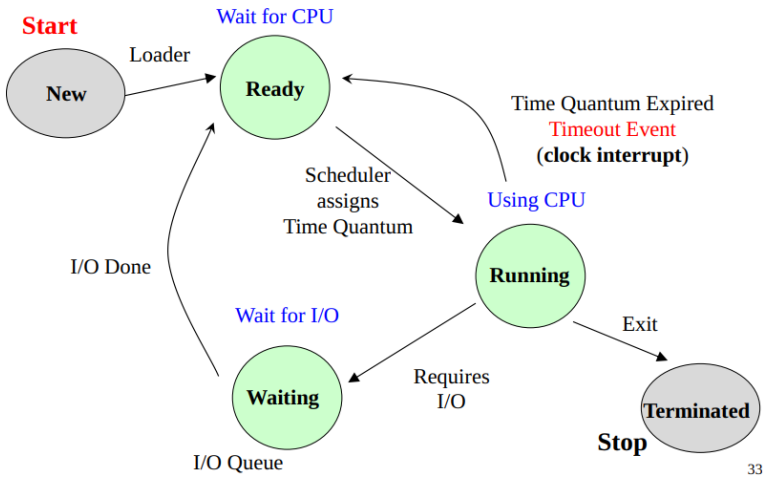
  Un'eccezione invece è il risultato di un'operazione ambigua o pericolosa, sono quindi provocate involontariamente da un'istruzione macchina eseguita dalla CPU che causa un errore o un caso particolare da gestire. Possono essere ti 3 tipi: TRAP, FAULT, ABORT. Nel caso di eccezioni di tipo TRAP l'istruzione viene momentaneamente sospesa, ma alla fine della gestione del trap viene portata a termine. Nel caso di eccezioni FAULT l'istruzione viene interrotta e viene eseguita la routine di gestione. Successivamente l'istruzione verrà rieseguita dall'inizio. Nel caso di errore grave (ad es. Divisione per 0 o segmentation fault) viene sollevata un'eccezione di tipo ABORT, l'errore è irrimediabile quindi l'istruzione viene terminata e il processo killato.

- Spiegare la differenza tra Interrupt sincroni e Interrupt asincroni.

Gli interrupt ASINCRONI (interrupt hardware) vengono scatenati da una periferica che, mediante un BUS, avvisa la CPU che è necessaria la gestione di un evento. Gli interrupt SINCRONI (interrupt software) sono esplicitamente chiamati dalla CPU con l'istruzione INT n

- Disegnare il diagramma di stato dei processi, dal punto di vista dello scheduler.

  START: New -> (loader) -> READY [wait for CPU] -> (Scheduler assigns Time slice) -> RUNNING [using CPU] => if requires I/O -> WAITING [I/O queue] -> I/O Done -> goto READY => else -> EXIT => (TERMINATED).



- Cosa è la tabella dei processi?

  È una struttura dati tenuta in memoria dal kernel che contiene tutti i processi del sistema.

- Cosa sono i livelli di privilegio (rings) della CPU?

  I privilege rings della CPU sono dei meccanismi di protezione offerti dal sistema operativo, che permettono di isolare le risorse del sistema, limitando l'accesso alle parti più sensibili. In un'architettura IA-32 ci sono 4 anelli di protezione, numerati da 0 a 3, dove 0 rappresenta l'anello più privilegiato (è il livello di privilegio del sistema operativo e del kernel). In genere le applicazioni utente lavorano nel ring 3 mentre i servizi di sistema nei ring 1 e 2.

- Qual è il compito del loader?

Il compito del loader è: creare in memoria i segmenti per il processo, popolando la tabella con le informazioni sui diversi segmenti, caricare il codice eseguibile del programma, caricare la sezione dati, creare lo stack utente e gli stack di sistema (per le funzioni del programma e per le System Calls rispettivamente), copiare eventuali argomenti passati a riga di comando e ordinare alla CPU di eseguire la prima istruzione eseguibile corrispondente al main del programma.

- Indicare la differenza tra FAULT e ABORT e dare un esempio per ciascuna delle due.

  Nel caso di eccezioni FAULT l'istruzione viene interrotta e viene eseguita la routine di gestione. Successivamente l'istruzione verrà rieseguita dall'inizio. Nel caso di errore grave (ad es. Divisione per 0 o segmentation fault) viene sollevata un'eccezione di tipo ABORT, l'errore è irrimediabile quindi l'istruzione viene terminata e il processo killato.

- Spiegare in che situazione un processo si trova in stato di waiting.

  Un processo entra in stato di waiting quando effettua una System Call bloccante (ad esempio l'accesso alle periferiche di I/O), e viene inserito nell'opportuna coda di attesa. Lo stato di waiting termina quando si verifica un evento e il processo torna nella coda dei ready.

Richiami di Linguaggio C

Modello di compilazione

Gli step essenziali della compilazione per un programma in C sono rappresentati dallo schema seguente:

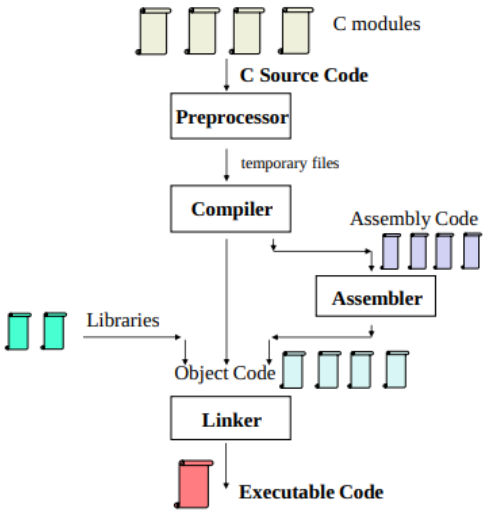


Figura 18: Step della compilazione di un programma C

La compilazione di un programma in C si effettua tramite l'utilizzo di programmi detti compilatori (Come lo GNU C Compiler, meglio conosciuto come gcc o il Microsoft C Compiler).

Questi compilatori svolgono 3 funzioni principali: preprocessing, compilazione e linking.

Per ragioni di mera praticità prenderemo in esempio solo il gcc (esiste un porting del gcc per sistemi windows chiamato MinGW-w64).

Per quanto riguarda il gcc, le 3 operazioni di preprocessing, compilazione e linking vengono eseguite da tre programmi differenti (il preprocessor, il compiler e il linker), i quali sono coordinati da un programma apposito, comunemente chiamato compilatore. Esiste una utility

(chiamata make) per eseguire le varie fasi per i soli file che sono stati modificati dopo l'ultima compilazione, diminuendo così il lavoro del compilatore.

In particolare, le utility che eseguono le fasi di preprocessing, compilazione e linking sono rispettivamente: cpp (The C Preprocessor) , gcc, ld (The GNU Linker).

Moduli e variabili

Variabili Globali e Specificatore extern

Le variabili globali sono quelle variabili dichiarate al di fuori da tutte le funzioni, in una qualsiasi posizione del file. Una variabile globale è accessibile da tutte le funzioni implementate dopo la dichiarazione della variabile.

Di default, una variabile globale NomeVar è visibile da tutti i moduli in cui esiste una dichiarazione extern di NomeVar, ossia: extern tipo NomeVar;

Una dichiarazione di questo tipo segnala al compilatore che nel modulo in cui essa è presente, la variabile NomeVar non esiste ma esiste in qualche altro modulo, e il modulo contenente la dichiarazione extern è autorizzato ad utilizzare la variabile. Il compilatore non dovrà quindi preoccuparsi di cercare la variabile all'interno del file perché assume che la sua dichiarazione esista da qualche parte. Sarà successivamente il linker a cercare in tutti i moduli fino a trovare la dichiarazione di NomeVar come variabile globale.

La variabile viene allocata fisicamente in memoria solo nel modulo in cui compare la dichiarazione senza extern (che può, a questo punto, essere unica tra i moduli) e precisamente nel punto esatto in cui compare la dichiarazione. Nei moduli con la dichiarazione extern resta solo un riferimento per il linker.

Protezione degli accessi esterni al modulo

Variabili globali e Specificatore static

Se vogliamo che una variabile (NomeVar), collocata in un determinato file, non sia accessibile a nessun altro modulo è necessario modificare la sua dichiarazione in quel modulo, facendola precedere dalla keyword static: static tipo NomeVar;

In questo modo quella variabile è accessibile solo alle funzioni del modulo in cui si trova.

Per maggiore chiarezza vediamo un esempio. Supponiamo che un programma sia formato da due moduli: var.c, main.c.

main.c contiene il main del programma ed alcune funzioni, tra cui la funzione f che accetta come parametro formale un intero e lo stampa.

var.c contiene due variabili intere, una globale (A) ed una globale ma statica (C), quindi visibile solo all'interno del modulo var.c

N.B.: Non esiste alcuna dichiarazione della variabile B nei moduli.

```
/*File var.c*/

int A=1;
static int C;

/*File main.c*/
#include <stdio.h>
extern int A;
extern int C;
void f(int var) {   printf("var=%d\n", var); }    /*stampa un
intero*/

void main(void) {
     f(A);     /*Corretto*/
     f(B);     /*Errore: 'B': undeclared identifier*/
     f(C);     /*Errore: unresolved external symbol_C*/
}
```

Il modulo main contiene due errori, con l'istruzione `f(C)` il modulo main tenta di accedere alla variabile C, la quale è però protetta dallo specificatore static all'interno del modulo var.

In questo caso l'errore non è segnalato dal compilatore, in quanto C ha una dichiarazione di tipo extern all'interno del modulo main. Sarà quindi il linker che, non trovando alcuna dichiarazione di una variabile C globale e accedibile da moduli esterni all'interno del modulo var (o di altri eventuali moduli del progetto), segnalerà tale errore.

Il secondo errore è causato dall'espressione `f(B)`, con la quale il modulo main tenta di accedere alla variabile B, la quale non è definita all'interno dello stesso modulo o in altri, di conseguenza in questo caso è il compilatore a segnalare l'errore. Il modulo main contiene due errori, con l'istruzione `f(C)` il modulo main tenta di accedere alla variabile C, la quale è però protetta dallo specificatore static all'interno del modulo var.

In questo caso l'errore non è segnalato dal compilatore, in quanto C ha una dichiarazione di tipo extern all'interno del modulo main. Sarà quindi il linker che, non trovando alcuna dichiarazione di una variabile C globale e accedibile da moduli esterni all'interno del modulo var (o di altri eventuali moduli del progetto), segnalerà tale errore.

Il secondo errore è causato dall'espressione `f(B)`, con la quale il modulo main tenta di accedere alla variabile B, la quale non è definita all'interno dello stesso modulo o in altri, di conseguenza in questo caso è il compilatore a segnalare l'errore.

Protezione dagli accessi esterni alla funzione

Ci sono alcuni casi in cui è utile che alcune variabili mantengano un valore tra una chiamata ad una funzione e la successiva. Lo specificatore static, applicato ad una variabile locale ordina al

compilatore di collocare la variabile non nello stack all'atto della chiamata alla funzione, ma in una locazione di memoria permanente (per tutta la durata dell'esecuzione), come fosse una variabile globale. A differenza di queste ultime però, le variabili locali statiche saranno visibili solo all'interno del blocco (delimitato da due parentesi graffe) all'interno del quale è stata dichiarata.

L'effetto è che tale variabile static:

1. Viene inizializzata una sola volta, ossia la prima volta che la funzione viene chiamata.
2. Mantiene il suo valore anche dopo che il controllo è fuori dalla funzione.

Un banale ma esplicativo esempio dell'utilizzo delle variabili locali statiche è contare il numero di volte in cui una funzione viene eseguita.

```
/* file contaf.c */
#include <stdio.h>
void f(void) {
      static int contatore=0; /*viene inizializzato una volta*/
      contatore = contatore + 1;
      printf("contatore =%d\n", contatore);
}
void main() /*per vedere cosa succede in f*/
{
      int i;
      for( i=0; i<100; i++ )
      f();
}
```

GCC e Makefile

GCC – Opzioni del preprocessore

-DNOMESIMBOLO definisce il simbolo NOMESIMBOLO senza specificarne un valore. Il simbolo apparirà es

istente a partire dall'inizio del file che si sta compilando.

-DNOMESIMBOLO=VALORE definisce il simbolo NOMESIMBOLO assegnandogli il valore VALORE che è una stringa qualunque. Il simbolo apparirà esistente a partire dall'inizio del file che si sta compilando.

Feature test macros

Un esempio particolare di utilizzo della define è rappresentato dal modo in cui si stabilisce quale standard utilizzare per compilare condizionatamente alcune parti di file di inclusione.

Si consideri ad esempio il file di inclusione standard <time.h> in cui è contenuto il prototipo della funzione nanosleep(). Tale funzione è definita nello standard POSIX.1b, specificato nel 1993 (IEEE Standard 1003.1b-1993) quindi nel file di inclusione il prototipo della funzione nanosleep() è protetto da una direttiva al preprocessore che controlla l'esistenza del simbolo _POSIX_C_SOURCE e controlla che quel simbolo abbia un valore uguale o superiore a 199309L.

```
#if defined _POSIX_C_SOURCE && _POSIX_C_SOURCE >= 199309L
          extern int nanosleep(const struct timespec
     *__requested_time, struct timespec *__remaining);
#endif
```

Per abilitare la compilazione, comprendendo quel prototipo, occorre definire quel simbolo _POSIX_C_SOURCE, assegnandogli un valore maggiore o uguale a 199309L, PRIMA dell'inclusione del file time.h.

Un modo per farlo è inserire la dichiarazione direttamente nel codice del mio modulo prima del punto in cui si include il file header.

```
/*file miomodulo.c*/
#define _POSIX_C_SOURCE 199309L
#include<time.h>
…
```

Un altro modo per abilitare la compilazione di quel prototipo è dichiarare quel simbolo passandolo come opzione al compilatore gcc quando lo si lancia per compilare, ovvero usando:

```
gcc –c -D_POSIX_C_SOURCE=199309L miomodulo.c
```

Analogamente, per utilizzare la funzione strerror_r() occorre includere il file di inclusione <string.h> ma, poiché quella funzione è definita nelle specifiche IEEE Std 1003.1-2001 (cioè POSIX 200112L) occorre anche definire il simbolo –D_POSIX_SOURCE=200112L

Similmente, per abilitare le sole versioni POSIX dello standard originale (IEEE Std 1003.1) si può usare il simbolo –D_POSIX_C_SOURCE=1.

- -D'NOME_MACRO=IMPLEMENTAZIONE_MACRO' definisce la macro. I due singoli apici servono ad impedire che eventuali parentesi tonde o punti e virgole nella macro siano interpretati rispettivamente come raggruppamenti di comandi e fine del comando, sono quindi NECESSARI.

  Esempio: -D'SALUTA(stringa)=printf("%s\n", stringa); fflush(stdout)'

- -UNOMESIMBOLO elimina la definizione del simbolo NOMESIMBOLO, è equivalente alla direttiva del preprocessore #undef NOMESIMBOLO, ma ha effetto sin dall'inizio dei file che si stanno compilando.
- -I/path/to/headers specifica il percorso relativo o assoluto per raggiungere la directory in cui sono contenuti i file di intestazione dell'utente. L'opzione può essere utilizzata più volte per specificare più di un percorso. Ad esempio: –I/home/studente/include -I../directory
- -E ordina di effettuare la sola fase di preprocessing, mandando sullo stdout il codice sorgente C generato dal preprocessore.
- -S ordina di tradurre il file sorgente C tramutandolo nel corrispondente sorgente assembly.
- -masm=dialect usato assieme al flag –S ordina di generare il sorgente asm secondo il dialetto specificato, cioè Intel (-masm=intel) oppure AT&T (-masm=att), quest'ultima è l'opzione di default.
- -c ordina di compilare il modulo specificato e di non procedere al linking.
- -o nomefile ordina di mettere il risultato della fase realizzata in un file di nome "nomefile". Può essere usato per specificare sia il nome di un modulo oggetto generato nella fase di compilazione che il nome dell'eseguibile generato in fase di linking.
- -ansi ordina di compilare secondo lo standard ansi, cioè c90.
- --std=STANDARD specifica lo standard da usare nella compilazione (ansi, c89, c90, gnu99 eccetera.).

- -Wpedantic quando si compila secondo lo std ansi ordina di avvisare quando si individuano istruzioni che rappresentano delle estensioni comunemente usate nel linguaggio C, che però rendono il codice poco leggibile e/o potenzialmente pericoloso.
- -Wall avvisa quando individua istruzioni non sbagliate ma potenzialmente pericolose o poco leggibili. Ciò non significa però abilitare tutti i warning.
- -Werror ordina al compilatore di trattare ogni warning come fosse un errore, interrompendo la compilazione
- -L/path/to/libraries specifica il percorso relativo o assoluto per raggiungere la directory in cui sono presenti le librerie fornite dall'utente. L'opzione può essere usata più volte per specificare più di un percorso.
- -lnome_libreria specifica di utilizzare la libreria indicata dal nome "ristretto" nome_libreria. Ad esempio, la libreria matematica libm.a viene specificata con –lm mentre una eventuale libreria libnome_libreria.a viene specificata con –lnome_libreria.

  (Nelle ultime versioni del gcc le istruzioni –lnome_libreria devono essere collocate obbligatoriamente alla fine della riga di comando, dopo l'elenco dei moduli da compilare).

- -Wl,-soname,nome_libreria (Virgole obbligatorie), da usare nel caso in cui si stia creando una libreria dinamica, specifica il nome nome_libreria da assegnare alla libreria dinamica che si sta creando.
- -Wl,rpath,/path/to/RunTimeLibDir (Virgole obbligatorie) utilizzato quando si sta creando un eseguibile che deve usare una libreria dinamica collocata fuori dalla directory predefinita. In questo modo si indica la directory RunTimeLibDir.

Makefile

Il file dependency system di Unix nasce per automatizzare il corretto aggiornamento di più files che hanno delle dipendenze. Viene solitamente utilizzato per automatizzare le operazioni di compilazione e linking di progetti scritti in linguaggio C, ma può essere utilizzato anche per altri scopi.

Concetti di dipendenza e di albero delle dipendenze

Supponiamo di aver scritto un programma costituito da due moduli C, implementati in due file main.c e funzioni.c, di avere un altro file funzioni.h incluso in main.c, e di avere un ulteriore file di intestazioni strutture.h incluso sia in funzioni.c che in main.c.

Per compilare il progetto occorre compilare i due moduli main.c e funzioni.c, generando così' i due moduli oggetto main.o e funzioni.o. Fatto questo occorre eseguire il linking dei due moduli main.o e funzioni.o per generare l'eseguibile main.exe

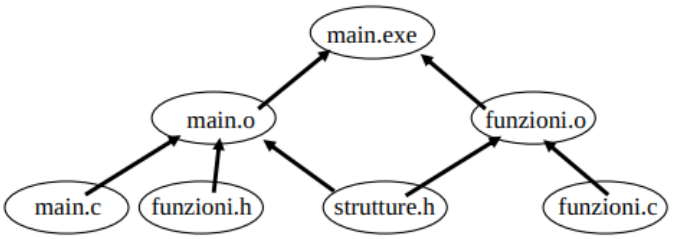


Figura 19: Albero delle dipendenze dell'esempio

Da questo esempio è possibile trarre il concetto di dipendenza di un file da un altro. Un file target1 dipende da un altro file dipendenza1 quando il contenuto del file target1 dipende dal contenuto del file dipendenza1. Una modifica del file dipendenza1 causa la necessità di effettuare delle operazioni per rigenerare il contenuto del file target1. Uno stesso file target può dipendere da diversi altri file.

Nelle dipendenze di un target vanno considerato anche i file di inclusione inclusi, con una direttiva del preprocessore, nei moduli C. Nell'esempio citato in precedenza, dato che il file funzioni.h è incluso nel file main.c, una modifica del file funzioni.h viene vista dal compilatore come una modifica al file main.c, è quindi necessario ricompilare anche il file main.c.

In sostanza, un modulo oggetto dipende sia dal file C che implementa quel modulo che dai file di intestazione inclusi nel codice sorgente.

La rappresentazione grafica di tutte le dipendenze diventa un albero detto dependency tree. Si noti che se il file target1 è a sua volta dipendenza del file target2, allora la modifica del file dipendenza1 causa la necessità di rigenerare anche il contenuto del file target2, utilizzando il contenuto del file target1. In pratica, a causa dell'insieme delle dipendenze è necessario rigenerare tutti i target che dipendono, sia direttamente che indirettamente, da un file dipendenza. Quindi la modifica di un file dipendenza causa la necessità di rigenerare l'intero ramo che si origina dal file dipendenza modificato.

Condizione di rigenerazione di un target

È importante sottolineare che la necessità di rigenerare un target esiste solo se è vera una delle seguenti condizioni:

1. Il file target non esiste, bisogna quindi generarlo ex-novo.
2. Il file target esiste, ma la data dell'ultima compilazione è precedente a quella dell'ultima modifica dei file dipendenza; quindi, il modulo oggetto di quel file non corrisponde più al contenuto di uno o più file dipendenza.

Si noti che la rigenerazione di un target rende necessaria la rigenerazione in cascata di tutti gli altri target dipendenti dal file rigenerato.

Struttura del Makefile: Make Rules

Il file Makefile di un progetto è un file di testo che contiene un insieme di regole (dette make rules) le quali:

- Descrivono la struttura dell'albero delle dipendenze.
- Descrivono le regole di generazione dei vari target di ogni dipendenza del progetto.

Ciascuna make rule:

- Indica uno o più target a cui la regola si riferisce (target list).
- Elenca le dipendenze da cui quel target dipende (dependency list).
- Elenca le operazioni che devono essere svolte (command list) quando una delle dipendenze è più recente del target (o se il target non esiste).

Solitamente l'operazione descritta genera il target, ma ciò non è sempre necessario. Esistono target fittizi, inseriti per svolgere sempre determinate operazioni.

Le make rules sono così formate:

```
target_list:   dependency_list
TAB            command1
TAB            …
TAB            commandN
RIGA_VUOTA
```

Sintassi del Makefile

I Makefile seguono particolari regole sintattiche per ragioni di compatibilità storica.

- Il simbolo # indica l'inizio di una riga commentata.
- Ciascuna dependency list e ciascun comando termina dove termina la riga.
- In ciascuna regola, le dipendenze sono separate tra loro da almeno uno spazio bianco o tabulazione.

- Ogni comando della command list inizia con una tabulazione.
- Ciascuna regola deve essere separata dalla successiva con una riga vuota.
- Il Makefile deve terminare con una andata a capo (\n).

Il Makefile dell'esempio nella Figura 19 sarebbe così costruito:

```
all:      main.exe
#riga vuota
main.exe:      main.o funzioni.o
gcc –o main.exe main.o funzioni.o
main.o:        main.c funzioni.h strutture.h
gcc –c –ansi –Wpedantic main.c
#riga vuota
funzioni.o:    funzioni.c strutture.h
gcc –c –ansi –Wpedantic funzioni.c
#riga vuota
clean:
-rm main.exe *.o
#NEWLINE
```

Ordine delle regole e costruzione dell'albero delle dipendenze

L'ordine delle regole scritte nel Makefile è molto rigido.

- Il make costruisce l'albero delle dipendenze a partire dalla prima regola leggendo il file delle regole.
- Il target trovato nella prima regola costituisce la radice dell'albero delle dipendenze.
- Ciascuna dipendenza nella dependency list della prima regola viene appesa come nodo figlio alla radice dell'albero
- Per ciascuna dipendenza (nodo figlio e/o foglia) appena aggiunta si cercano nel Makefile le regole che hanno questa dipendenza come proprio target.
- Si continua utilizzando le foglie come candidate target, cercando regole che hanno come target le candidate e, se le si trova, aggiungendo come foglie le dipendenze delle candidate target.

- Ci si ferma quando nelle foglie ci sono solo delle dipendenze che non compaiono come target in nessuna regola.

Esecuzione delle command list

L'albero delle dipendenze viene visitato partendo dai livelli più bassi. Per ciascuno nodo N si legge la data di ultima modifica del file N e la si confronta con la data di ultima modifica del file presente nel nodo padre P. Se il file del nodo padre non esiste o ha una data di ultima modifica più vecchia del nodo figlio, allora viene eseguita la lista di comandi della regola che ha il padre come target.

Esecuzione del make

La creazione dell'albero delle dipendenze e l'esecuzione delle liste di comandi viene effettuata eseguendo il comando make, passandogli come parametro l'opzione -f seguita dal nome del file che contiene le regole da seguire

```
make –f nomeMakefile
```

Si può omettere l'opzione -f, il comando make cercherà di default un file denominato Makefile.

Durante l'esecuzione, il make stampa sullo stdout i comandi delle liste mentre li esegue.

Aggiungendo l'opzione -n permette di stampare i comandi che il make avrebbe eseguito, senza eseguirli.

Posso ordinare di eseguire il make specificando esplicitamente il target che voglio sia usato come root del dependency tree. In questo modo verrà eventualmente rigenerato solo il ramo dell'albero che ha il target specificato come radice.

```
make funzioni.o
```

Best practices per la costruzione di Makefile

La prima regola stabilisce qual è il target radice nell'albero delle dipendenze.

Spesso si definisce per prima una regola che come target specifica "all", come dipendenze specifica il (o i) target reali ed ha una lista di comandi vuota. Ciò serve ad indicare quali sono i target da ottenere.

```
all: main.exe
```

Nel file delle regole, è bene esplicitare quali sono i target fittizi elencandoli come dipendenze di un target predefinito denominato .PHONY. Per i target elencati come dipendenza di .PHONY il make non cerca l'esistenza di un file e non cerca di applicare al target le regole implicite.

Esempio:

```
.PHONY:    clean
clean:
           rm *.exe *.o *~
```

Il make termina con un errore quando uno dei comandi di una lista dei comandi viene eseguito e restituisce un errore. Se di un comando non interessa il risultato, ma si vuole impedire che il make termini anche se quel comando termina con un errore, è possibile mettere un - davanti al comando. Il make interpreta quel - come l'ordine di non considerare un eventuale errore restituito dal comando.

Esempio

```
.PHONY:    clean
clean:
           -rm *.exe *.o *~
```

Target fittizi (.PHONY)

È possibile specificare dei target che non sono file e che hanno come scopo solo l'esecuzione di una sequenza di comandi. Questi target non specificano alcuna dipendenza e non possono comparire come prima regole, in modo da essere eseguiti solo in caso di passaggio come argomento al comando make. Dato che il target fittizio non è un file, il target fittizio provoca l'esecuzione del programma in ogni caso. Un target fittizio molto comune nei Makefile è:

```
clean:
        rm *.exe *.o *~
```

Questa regola serve per eliminare tutti i file generati dalla compilazione (eseguibili e moduli oggetto) e tutti gli eventuali file temporanei degli editor. Dato che nel Makefile non c'è una regola che abbia come target clean (non esiste alcuna regola che crea un file denominato "clean"), il comando "`rm *.exe *.o *~`" verrà eseguito ogni volta che si invocherà:

```
make clean
```

Controindicazioni sull'uso dei target fittizi

L'uso di target fittizi potrebbe compromettere il corretto funzionamento del make. Bisogna prestare particolare attenzione a due problemi:

- Mascheramento da parte di file esistenti: Se per sbaglio nella directory viene creato un file chiamato con lo stesso nome del target fittizio allora, poiché la regola fittizia non ha dipendenze e il file c'è già, quel file non ha necessità di essere aggiornato e quindi la lista dei comandi non verrà mai eseguita.
- Spreco di tempo: In caso di regole implicite (che qui non vengono trattate) il make spreca tempo, perché cerca di risolvere il target fittizio applicandogli le regole implicite.

Variabili nei Makefile

È possibile definire delle variabili all'interno dei Makefile. In un Makefile una variabile è un simbolo definito in un Makefile che rappresenta una stringa di testo (valore della variabile). Si ricorda che il Makefile non è uno script, per tanto è possibile assegnare alle variabili solo valori costanti, i quali non possono quindi essere risultati di altri comandi. Inoltre, è impossibile, in generale, eseguire comandi al di fuori delle command list inserite nelle make rules.

La dichiarazione delle variabili va inserita all'inizio del Makefile:

```
PERCORSO=/usr/local/lib
```

Esiste un caso particolare in cui definire una variabile modifica il funzionamento del make.

I comandi specificati nelle liste di comandi del Makefile sono eseguiti in una shell sh, non in una bash.

La shell sh è simile ma non uguale alla bash e alcune differenze potrebbero causare problemi.

In particolare: in sh non esistono le espressioni condizionali con le doppie parentesi quadre, ma solo quelle con le semplici parentesi quadre. Inoltre, nel comando builtin echo della sh non esiste l'opzione -e.

Per eseguire i comandi in una bash devo settare la variabile:

```
SHELL=/bin/bash
```

Conclusioni

Il Makefile per l'esempio descritto all'inizio della sezione è:

```
CFLAGS=-ansi –Wpedantic
TARGET=main.exe
OBJECTS=main.o funzioni.o
all: ${TARGET}

main.exe: main.o funzioni.o
          gcc -o ${TARGET} main.o funzioni.o

main.o:    main.c funzioni.h strutture.h
          gcc -c ${CFLAGS} main.c
funzioni.o:    funzioni.c strutture.h
          gcc -c ${CFLAGS} funzioni.c

.PHONY:   clean

clean:
          -rm ${TARGET} ${OBJECTS} *~ core
```

Domande riassuntive

- A cosa serve l'opzione -I del gcc?

  -I/path/to/headers specifica il percorso relativo o assoluto per raggiungere la directory in cui sono contenuti i file di intestazione dell'utente. L'opzione può essere utilizzata più volte per specificare più di un percorso.

  Ad esempio: -I/home/studente/include -I../directory

- A cosa serve l'opzione -L del gcc?

  -L/path/to/libraries specifica il percorso relativo o assoluto per raggiungere la directory in cui sono presenti le librerie fornite dall'utente. L'opzione può essere usata più volte per specificare più di un percorso.

- A cosa serve l'opzione -l del gcc?

  -lnome_libreria specifica di utilizzare la libreria indicata dal nome "ristretto" nome_libreria. Ad esempio, la libreria matematica libm.a viene specificata con -lm mentre una eventuale libreria libnome_libreria.a viene specificata con –lnome_librerira.

- In linguaggio C, se un programma è costituito da più moduli, è possibile che una stessa funzione (cioè una funzione con stesso nome, stessi tipi di argomenti e stesso tipo di dato restituito) sia implementata in due diversi moduli senza che questo provochi un errore nella generazione del programma eseguibile? Perché?

  No, una funzione in C non può essere implementata in modo identico in due diversi moduli. Ciò causerebbe un errore a link-time e rappresenta un'ambiguità nel codice (non si sa quale delle due funzioni dovrebbe essere eseguita). Tuttavia se una delle due implementazioni fosse di tipo "static" la visibilità della funzione resterebbe circoscritta al modulo in cui essa è dichiarata, rendendo possibile avere due funzioni con stesso nome, parametri e tipo di dato restituito in due moduli diversi.

- La seguente istruzione si trova nella funzione main di un modulo scritto in C.

  ```
  ris = PRODOTTO ( 3 - 4 , SOMMA ( 2 , 7 ) );
  ```

  Precedentemente, nello stesso file, le due macro sono state così definite:

  ```
  #define PRODOTTO( X, Y ) ( (X)*(Y) )

  #define SOMMA( X , Y ) ( (X)+(Y) )
  ```

  Se quel modulo viene processato dal preprocessore del compilatore gcc, in che modo viene tradotta quella istruzione dal preprocessore?

  ```
  ris = ( (3 - 4) * ( (2)+(7) ) ); => ris = -9;
  ```

- Supponiamo di inserire le seguenti due istruzioni, in un modulo scritto in C, in una zona fuori da tutte le funzioni del modulo;

```
#define NUM 10

int vet [ 10 ];
```

Se quel modulo viene processato dal preprocessore del compilatore gcc, in che modo viene tradotta quella istruzione dal preprocessore?

L'istruzione "#define NUM 10" verrà utilizzata per sostituire ogni occorrenza di NUM con "10" all'interno del modulo. Nel caso di "int vet[10]" la macro NUM non compare, di conseguenza non verrà modificata dal preprocessore.

Librerie

Le librerie sono file che contengono le implementazioni in codice macchina di alcune funzioni.

Possono implementare sia funzioni di sistema che definite dall'utente.

In genere le librerie sono mantenute sul disco in directories predefinite, in file il cui nome inizia per "lib". L'estensione del file varia a seconda che la libreria sia statica (.a) o condivisa (.so).

Il nome ufficiale della libreria è rappresentato dal nome, escluso il simbolo "lib". (Il nome ufficiale è quello da usare nell'opzione del linker –InomeLib).

Librerie statiche e condivise

Le librerie statiche sono accorpate ai moduli a link-time (la fase della compilazione in cui si genera l'eseguibile collegando librerie e moduli oggetto).

Il linker deve sapere dove cercarle sul disco per copiarne il contenuto e inserirlo nell'eseguibile, il quale avrà a disposizione le funzioni implementate in libreria sin dall'inizio del run-time.

Le librerie condivise (dinamiche) non sono accorpate ai moduli né inserite nell'eseguibile a link-time. Per ciascuna chiamata a una funzione della libreria il linker inserisce al volo la porzione di codice macchina (stub) e la posizione a run-time della libreria nell'eseguibile rendendola disponibile quando necessario.

Il linker avrà bisogno di sapere sia la posizione della libreria a link-time (deve verificare se essa fornisce effettivamente la funzione chiamata e copiare dalla libreria il solo frammento che effettua il caricamento), che la posizione a run-time della libreria sul disco, in quanto deve inserire questa posizione nel codice insieme allo stub.

Per utilizzare un percorso diverso da quello di default si possono utilizzare le opzioni del linker descritte nel capitolo precedente.

Thread e POSIX Thread

I Thread

Un thread è un singolo flusso di istruzioni, all'interno di un processo, che lo scheduler può far eseguire separatamente e concorrentemente con il resto del processo. Per poter eseguire in parallelo con il resto del processo, un thread deve possedere un proprio contesto per realizzare il proprio flusso di controllo. Ogni processo ha il proprio PID, Program Counter, Stato dei registri, Stack, Codice, Dati, File Descriptors, Entità IPC (Inter Process Comunication), Azioni dei segnali.

L'astrazione dei thread vuole consentire di eseguire procedure in parallelo. Ciascuna procedura da eseguire in parallelo sarà quindi un thread.

Contesto di esecuzione di un Thread

Tutti i thread di un processo condividono le risorse del processo (variabili globali e CPU) ed eseguono nello stesso spazio utente. Ciascun thread può usare tutte le variabili globali del processo e condivide la tabella dei file descriptors. Ciascun thread potrà avere propri dati locali, e sicuramente avrà un proprio stack, un proprio program counter, un proprio stato dei registri ed una propria variabile errno. Dati globali ed entità rappresentano lo stato di esecuzione del thread.

Vantaggi e Svantaggi nell'uso di Thread

I vantaggi nell'uso dei thread sono diversi, si ha una condivisione degli oggetti semplificata grazie alla visibilità dei dati globali. È più semplice la gestione degli eventi asincroni (I/O), i thread condividono lo stesso spazio di indirizzamento, di conseguenza la comunicazione tra thread è più

semplice della comunicazione tra processi. Infine, il context switch è molto veloce, dato che buona parte dell'ambiente viene mantenuta nel passaggio da un thread a un altro.

Nonostante ciò, ci sono diversi svantaggi da tenere in considerazione nella progettazione di applicazioni multithreading. Ci sono delle situazioni in cui è necessario gestire la concorrenza nell'accedere a risorse condivise da parte di diversi thread contemporaneamente. Bisogna quindi gestire la mutua esclusione dei dati condivisi per evitare che i thread vi accedano nel momento sbagliato (Ad esempio quando un thread che vuole leggere il contenuto di una variabile vi accede mentre un altro thread sta effettuando delle operazioni in scrittura su tale variabile).

La concorrenza viene gestita sia da chi scrive il programma che dal sistema operativo, che implementa le funzioni di libreria e le System Calls utilizzate "contemporaneamente" da più thread.

Le funzioni delle librerie di sistema e le System Call devono essere rientranti (thread safe).

Operazioni atomiche: definizione

Un'operazione è atomica se è indivisibile, quindi se nessun'altra operazione che usa i dati condivisi dall'operazione atomica può cominciare prima che l'operazione atomica sia terminata; quindi, non può esistere interleaving tra le diverse operazioni. Il risultato di tale operazione è sempre lo stesso se le condizioni di partenza sono uguali.

Nel caso in cui fosse necessario interrompere un'operazione atomica, bisognerà in seguito farla ripartire da capo, il risultato dell'operazione atomica potrebbe essere differente rispetto a quella interrotta se le condizioni di partenza sono diverse da quelle dell'operazione interrotta.

Le istruzioni C ad alto livello, in generale, non sono atomiche, neanche le operazioni di assegnamento di una costante ad una variabile le quali risultano atomiche solo se la dimensione della variabile è minore dell'ampiezza del bus dati.

Il discorso è analogo per l'assembly, infatti il linguaggio mette a disposizione delle strutture sintattiche simili a delle macro, come ad esempio l'istruzione call che invoca una funzione e salva alcuni dati nello stack, operazioni che in linguaggio macchina non sono eseguite atomicamente. Tuttavia, a parte le macro, esistono altre operazioni elementari in asm che non sono atomiche (ad esempio incrementare di 1 il valore di un byte in memoria), mentre quelle di copia di un dato (da registro a memoria o viceversa) sono generalmente atomiche se l'ampiezza del bus dati è maggiore o uguale alla dimensione dei registri.

Operazioni atomiche

Operazioni compare-exchange o compare-and-swap

I processori, a livello ISA, mettono a disposizione un set di istruzioni macchina basiche e atomiche per consentire l'implementazione di flussi di esecuzione atomici più complessi che si basano sul valore di una variabile "di protezione". Queste istruzioni variano in base alla CPU utilizzata. Nelle architetture Intel x86 (IA-32) E x64 (EM64T, Intel 64) esiste un'operazione cmpxchg ed un prefisso lock che modifica il comportamento di questa istruzione, rendendolo atomico. In altre famiglie di processori questo tipo di operazione si chiama compare-and-swap.

L'istruzione lock cmpxchg ha come parametri una variabile intera in memoria e il nome di un registro in cui deve essere già presente il valore da assegnare alla variabile. L'argomento implicito è il registro EAX in cui deve essere posto il valore da confrontare col valore della variabile.

L'istruzione lock cmpxchg preleva il valore della variabile puntata dall'indirizzo. lo confronta col valore nel registro EAX, se i due valori sono uguali assegna alla variabile il nuovo valore. In pratica, atomicamente, se la variabile ha il valore cercato (inserito nel registro EAX) allora le viene assegnato il nuovo valore, altrimenti mantiene il valore che aveva.

Riassumendo, le istruzioni simili a compare-exchange servono a realizzare in modo atomico la verifica del valore di una variabile globale che indica se la risorsa è occupata. Se la variabile vale 0, viene settata a 1. Confronto e assegnamento vengono eseguiti insieme e atomicamente.

System Call non rientranti

I thread di un processo usano il sistema operativo tramite system call che sfruttano dati e tabelle di sistema dedicate. Le System Call dovrebbero essere costruite in modo da poter essere utilizzate da più thread contemporaneamente. Alcune System Call, tuttavia, non sono pensate per tali scopi (ad esempio la funzione char *inet_ntoa()).

Thread Safe Call

Le implementazioni dei thread per soddisfare gli standard POSIX offrono delle funzioni thread safe (rientranti o reentrant), le quali non causano problemi in lettura o scrittura di strutture dati interne al sistema operativo e condivise dai thread di uno stesso processo. In particolare, tutte le funzioni dello standard ANSI C e le funzioni degli standard POSIX.1 sono thread safe, ad eccezione di alcune: strerror, inet_ntoa, ctime, getlogin, rand, readdir, strtok, ttyname, gethostXXX, getprotoXXX, getservXXX.

Per ovviare al problema delle funzioni di libreria non rientranti, sono state rese disponibili altre funzioni che invece sono implementate in modo rientrante. Solitamente, il nome delle funzioni rientranti finisce con _r. Ad esempio, la funzione char *strerror(int errnum); fa cose diverse ma

restituisce il risultato nello stesso modo della inet_ntoa, infatti non è rientrante. Esiste una versione rientrante di tale funzione, denominata strerror_r, disponibile se il simbolo _POSIX_C_SOURCE è definito maggiore o uguale di 200112L.

POSIX Thread

I thread sono stati standardizzati a partire dalla versione dello standard IEEE POSIX 1003.1c (1995), che specifica la API dei thread. I thread POSIX sono noti come pthread.

Le API per pthread distinguono le funzioni in 3 gruppi:

1. Thread Management: funzioni per creare, eliminare, attendere la fine dei pthread.
2. Mutexes: funzioni per supportare un tipo di sincronizzazione semplice chiamata mutex (MUTual EXclusion). Comprende funzioni per creare ed eliminare la struttura per mutua esclusione di una risorsa, acquisire e rilasciare tale risorsa.
3. Condition variables: funzioni a supporto di una sincronizzazione più complessa dipendente dal valore di variabili, secondo i modi definiti dal programmatore. Comprende funzioni per creare ed eliminare la struttura per la sincronizzazione, per attendere e segnalare le modifiche delle variabili.

POSIX Thread APIs

Lo standard POSIX ha definito una convenzione sui nomi delle funzioni. Gli identificatori della libreria dei pthread iniziano col prefisso “pthread_”.

pthread_	indica la gestione dei thread in generale.
pthread_attr_	funzioni per la gestione proprietà dei thread.
pthread_mutex_	gestione della mutua esclusione.

pthread_mutexattr_ proprietà delle strutture per mutex.

pthread_cond_ gestione delle condition variables.

pthread_condattr_ proprietà delle condition variables.

pthread_key_ dati speciali dei thread.

Compilazione e linking

Di programmi che usano pthread

Il file pthread.h contiene le definizioni dei pthread. Normalmente si può compilare secondo il dialetto –ansi. Per aderire allo standard POSIX nella sua prima versione originaria IEEE STD 1003.1 occorre inserire il simbolo –D_POSIX_C_SOURCE=1. Il dialetto –ansi non comprende però alcune funzioni messe a disposizione recentemente dalle librerie pthread, in particolare alcune API relative ai semafori per distinguere letture e scritture su variabili. Quindi compilando con il dialetto –ansi le parti di pthread.h relative a tali funzioni vengono escluse dal compilatore. Quindi, se fosse necessario utilizzare tali funzioni è bene compilare secondo un dialetto più moderno, ad esempio secondo lo standard gnu99.

In un programma che usa i pthread è bene definire, all'inizio di ciascun file, il simbolo _THREAD_SAFE. Ciò consente al preprocessore di selezionare, se esistono, le implementazioni rientranti delle funzioni di libreria.

Durante la fase di linking, usando il gcc, è necessario aggiungere il flag –lpthread per usare la libreria dei pthread libpthread.so.

API per creazione ed esecuzione di pthread

- ```
  int pthread_create(pthread_t *thread, pthread_attr_t *attr,
                     void *(*start_routine)(void *), void *arg);
  ```
  ```

  - Crea un thread e lo rende eseguibile, cioè lo mette a disposizione dello scheduler che lo farà partire prima o poi.
  - Il primo parametro è un puntatore ad un thread id in cui verrà scritto il thread id del thread appena creato.
  - Il secondo parametro seleziona caratteristiche particolari, può essere NULL per il comportamento di default.
  - Il terzo parametro start_routine è l'indirizzo della procedura da far eseguire al thread. Deve avere come unico argomento un puntatore e restituisce un puntatore a void (void *).
  - Il quarto parametro è un puntatore che viene passato come argomento a start_routine. Tale puntatore punta ad una area di memoria allocata dinamicamente. Può essere NULL se la start_routine non richiede argomenti.
  - Restituisce un intero che vale 0 se la creazione ha avuto successo, un codice di errore (diverso da 0) altrimenti.

- `void pthread_exit(void *retval);`
  - Termina l'esecuzione del thread da cui viene chiamata, immagazzina l'indirizzo retval, restituendolo ad un altro thread che attende la sua fine.
  - Il sistema libera le risorse allocate dal thread.
  - Il caso del main è particolare, se il main termina (chiama return) prima che i thread a lui legati si esauriscano e non chiama la pthread_exit, allora tutti i thread sono terminati. Se il main chiama pthread_exit allora u thread possono continuare a vivere fino alla terminazione.

Pthread identifiers

Il tipo di dato pthread_t è il tipo di dato che contiene l'identificatore univoco di un pthread.

Due funzioni utili a maneggiare tale dato sono:

- `pthread_t pthread_self(void);`
    - Restituisce l'id del thread che la chiama.
- `int pthread_equal(pthread_1 thread1,  pthread_t thread2);`
    - Restituisce 1 se i due pthread id sono uguali.

Passaggi di argomenti a pthread

La funzione pthread_create richiede un puntatore per il passaggio dei parametri al pthread che sta creando. Nel momento in cui il pthread inizia l'esecuzione ha a disposizione questo parametro.

Il chiamante di pthread_create deve allocare dinamicamente una struttura dati in cui collocare i parametri da passare al thread. Il pthread creato userà quest'area di memoria e la deallocherà in seguito.

Formalmente, la funzione che implementa il pthread prende come argomento un puntatore a void (void *).

Nel seguente esempio vengono creati 10 pthread, passando a ciascuno un valore intero. Ogni pthread legge tale valore e lo mette in una propria variabile indice.

```
/*main*/

#define NUM_THREADS 10
pthread_t tid;
int, rc;
int*p;
for(t = 0; t < NUM_THREADS; ++t) {
*p = t;
rc = pthread_create(&tid, NULL, func, (void*)p);

/*pthread*/
void *func(void *arg) {
     int indice;
     /* uso arg come puntatore a int */
     indice = *((int*)arg);
     /* rilascio la memoria*/
     free(arg);
     printf("ricevuto %i\n", indice);
     pthread_exit(null);
}
```

Terminazione e risultato di un pthread

Quando un pthread termina, potrebbe essere utile far conoscere ad un altro pthread il risultato del suo "lavoro". Occorre quindi un meccanismo tramite il quale un pthread può restituire un risultato (un'area di memoria).

La funzione pthread_exit permette ad un pthread di stabilire quale area di memoria deve essere restituita come risultato al termine dell'esecuzione. Formalmente, ogni pthread restituisce un puntatore a void (void *). Da questo deriva che la funzione pthread_exit prevede come argomento un puntatore di quel tipo.

Il risultato restituito viene mantenuto nello stack del pthread anche dopo la terminazione del pthread stesso, e fino a che un altro pthread non richiede quel risultato tramite la chiamata a pthread_join.

La memoria occupata dallo stack del pthread terminato viene rilasciata solo dopo la join con un thread che ne abbia richiesto il risultato oppure, ovviamente, alla terminazione del processo.

Bisogna considerare comunque che, nel caso in cui un processo sia attivo a lungo e, nel frattempo, crea diversi pthread senza rilasciare la memoria dello stack, essi riempiranno la memoria causando la terminazione del processo.

È possibile configurare i pthread al momento della propria creazione per consentire che il loro stack venga liberato immediatamente dopo la terminazione senza attendere la pthread_join (questo tipo di thread è detto detached thread). Così facendo si perde la possibilità di recuperare il risultato dei pthread.

Attesa della terminazione di un pthread

È possibile far attendere la terminazione di un pthread ad un altro pthread dello stesso processo. Occorre conoscere l'identificatore del pthread di cui si attende la terminazione. La funzione pthread_join vuole come argomento l'indirizzo di un puntatore a void (void *), in cui verrà collocato l'indirizzo restituito dal pthread il cui identificatore è passato come primo argomento.

Nello scheletro di esempio che segue, il main crea 10 pthread, poi aspetta la terminazione dei 10 pthread, da ciascuno riceve un puntatore a intero e lo usa per stampare il valore intero lì contenuto, infine rilascia la memoria allocata dei pthread.

```
/*Codice main*/

#define NUM_THREADS 10
pthread_t tid[NUM_THREADS];
int t, rc; int *p;
/*creo i pthread*/
for(t = 0; t < NUM_THREAD; ++t) {
     int ris;
     rc = pthread_join(tid[t], (void **)&p);
     ris = *p;
     printf("pthread restituisce %i\n", ris);
     free(p);
}
/*codice pthread*/
void *func(void *arg) {
     int *ptr;
     int indice = *((int*)arg);
     free(arg);
     /* qui il pthread svolge alcune operazioni */
     ptr = (int *) malloc(sizeof(int));
     /* il pthread vuole passare un valore intero, ad esempio
     100+indice */
     *ptr = 100+indice;
     pthread_exit((void *)ptr);
}
```

Mutua esclusione e sincronizzazione

Race condition

Poniamoci nel caso in cui due thread debbano decrementare il voler di una variabile globale “data”, ma solo se questa è maggiore di zero. Inizialmente data vale 1.

A seconda del tempo di esecuzione dei due thread, la variabile data assume valori diversi. In questo caso si parla di race condition.

Più formalmente, due (o più) thread sono in race condition quando tentano di accedere a delle risorse condivise contemporaneamente, causando comportamenti errati o insapettati in quanto sono assenti meccanismi di controllo degli accessi alla variabile.

Mutex Variables

Mutex è l’abbreviazione di mutua esclusione. Una variabile mutex (formalmente di tipo pthread_mutex_t) serve a regolare l’accesso a dei dati che non possono essere acceduti contemporaneamente da più thread.

Ogni thread, prima di accedere a tali dati, deve effettuare un’operazione di pthread_mutex_lock su una stessa variabile mutex. L’operazione detta “lock” di una variabile mutex blocca l’accesso alla variabile condivisa da parte degli altri thread.

Se più thread eseguono la lock su una stessa variabile mutex, solo uno la termina e prosegue con l’esecuzione, gli altri rimangono bloccati. In questo modo il processo che continua può accedere ai dati che sono ora protetti dalla mutua esclusione. Una volta terminate le operazioni sulla variabile condivisa, il pthread effettua un’operazione di pthread_mutex_unlock, la quale libera la variabile mutex. Ora un altro thread, rimasto in attesa di ricevere la mutex, potrà terminare la lock ed accedere ai dati condivisi.

La sequenza d'uso tipica di una mutex è:

- Creare ed inizializzare la mutex.
- Diversi thread tentano l'accesso alla mutex con pthread_mutex_lock.
- Un solo thread termina la lock e prende possesso della mutex, lasciando gli altri in attesa.
- Il thread che detiene la mutex accede ai dati.
- Una volta finito la mutex viene rilasciata con pthread_mutex_unlock.
- Un altro thread termina la lock e prende possesso della mutex.
- Al termine del programma (o quando non è più necessaria) la mutex viene distrutta.

Operazioni con Mutex

Creazione e distruzione di variabili Mutex

Le variabili Mutex devono essere create usando il tipo di dato pthread_mutex_t e devono essere inizializzate prima di poter essere utilizzate.

Si può fare in due modi:

- Staticamente: `pthread_mutex_t myMutex=PTHREAD_MUTEX_INITIALIZER;`
- Dinamicamente:
```
pthread_mutex_t mymutex;

pthread_mutex_init(&mymutex, &attribute);
```

Finito l'uso delle variabili mutex è bene rilasciarle. Per farlo si utilizza la funzione `pthread_mutex_destroy(&mymutex);`

API per mutua esclusione

- `int pthread_mutex_lock(pthread_mutex_t *mutex);`

  Blocca la variabile mutex passata come argomento.

Se la variabile mutex è sbloccata, diventa bloccata e di proprietà del thread chiamante, la funzione pthread_mutex_lock termina immediatamente e restituisce il controllo al chiamante, con risultato 0. In caso di problemi viene restituito un codice di errore. Bisogna prestare particolare attenzione all'utilizzo della lock, in quanto se il thread chiamante possiede già la mutex, esso si blocca per sempre.

- `int pthread_mutex_unlock(pthread_mutex_t *mutex);`

  Sblocca la variabile mutex passata come argomento.

  La funzione deve essere chiamata su una mutex già bloccata e di proprietà del thread chiamante. In tal caso la funzione pthread_mutex_unlock sblocca la variabile mutex e termina restituendo 0. In caso di problemi viene restituito un codice di errore. Se la unlock viene chiamata da un thread che non possiede la mutex si genera un comportamento impredicibile, dato che la mutex viene sbloccata anche se di proprietà di un thread diverso.

- `int pthread_mutex_trylock(pthread_mutex_t *mutex);`

Questa funzione è come la lock(), tuttavia se la mutex è già in possesso di un altro thread restituisce immediatamente il controllo al chiamante con risultato EBUSY, altrimenti restituisce 0. Questa funzione porta a creare codice che causa busy waiting.

Condition Variables

Le Mutex consentono di effettuare sincronizzazioni tra pthread nel momento in cui ciascun pthread tenta l'accesso in mutua esclusione. L'accesso viene consentito senza poter valutare a priori il valore delle variabili condivise.

Con l'uso delle condition variables si risolve questa mancanza, esse consentono di effettuare la sincronizzazione mentre questi detengono già la mutua esclusione, bloccandoli o facendoli continuare in base a condizioni definite dal programmatore e dipendenti dal valore dei dati condivisi. Senza le condition variables si causerebbe busy waiting durante la sincronizzazione in quanto sarebbe necessario effettuare un loop continuo in cui prima si blocca la mutex, poi si valuta il contenuto delle variabili, se questo rispetta le condizioni richieste allora il pthread esegue le operazioni e poi rilascia la mutua esclusione; se invece la condizione non è rispettata il pthread rilascia la mutex e ricontrolla il valore, in continuazione, causando appunto busy waiting.

Le condition variables vanno utilizzate insieme ad una mutex per consentire di accedere in mutua esclusione alle variabili condivise.

Ci sono tre API per sincronizzare i pthread con le condition variables:

- pthread_cond_wait: blocca un thread fino a che un altro non lo risveglia e può prendere la mutex.
- pthread_cond_signal: sveglia un altro thread bloccato sulla wait, ma l'altro thread deve prima ottenere la mutex per ripartire.
- pthread_cond_broadcast: sveglia tutti i thread bloccati sulla wait, ma ciascuno di questi deve prima ottenere la mutex per ripartire.

Nell'esempio si vede come utilizzare le condition variables correttamente e senza causa busy waiting. Una gallina produce un uovo impiegando del tempo ed una volpe tenta di mangiarle, se mangia un uovo allora attende di digerirlo prima di mangiarne un altro.

La variabile condivisa in questo caso è uova.

```
int uova = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
void *gallina_produce_uova(void *arg) {
     while(1) {
          /* gallina produce uova impiegando tempo ...*/
          /* poi gallina depone uovo e fa coccode' */
          pthread_mutex_lock(&mutex);
          uova ++;
          pthread_cond_signal(&cond); /* coccode' avvisa volpe
          che c'è uovo */
          pthread_mutex_unlock(&mutex);
          }
     }
     void *volpe_mangia_uova(void *arg) {
          while (1) {
               pthread_mutex_lock(&mutex);
               while( uova==0 ) {
                    /* attendi che venga prodotto uovo, attendi
               signal */
                    pthread_cond_wait(&cond, &mutex);
               } /* quando arrivo qui c'è almeno un uovo */
               mangia(); /* volpe mangia uovo impiegando tempo
               ...*/
               uova --;
               pthread_mutex_unlock(&mutex);
               burp(); /* volpe digerisce impiegando tempo */
          }
     }
     int main(void) {
          pthread_t tid;
          pthread_create(&tid, NULL, gallina_produce_uova,
          NULL);
          pthread_create(&tid, NULL, volpe_mangia_uova, NULL);
          pthread_exit(NULL);
     }
```

API per Condition Variables

Creazione e distruzione di Condition Variables

Le condition variables devono essere create usando il tipo di dato pthread_cond_t e devono essere inizializzate prima di poter essere usate.

Si può fare in due modi:

- Staticamente: `pthread_cond_t mycond=PTHREAD_COND_INITIALIZER;`
- Dinamicamente:
```
pthread_cond_t mycond;
pthread_cond_init(&mycond, &attribute);
```

È bene rilasciare le condition variables dopo aver finito di usarle.

Attesa

```
1. pthread_cond_wait(pthread_cond_t *cond,
        pthread_mutex_t *mutex);
```

È eseguita da un thread quando vuole bloccarsi ed aspettare che una condizione sia vera.

Prima di chiamare la wait il pthread deve ottenere il possesso della mutex specificata. Quando la wait blocca il pthread, automaticamente la wait rilascia la mutex e si mette in attesa di essere abilitato al risveglio da una signal chiamata da un altro thread.

La wait continua e termina solo al verificarsi di due condizioni:

- Un altro pthread l'ha risvegliata con una chiamata alla signal.
- La wait ha riottenuto la mutex.

Quando la wait termina, il pthread detiene la mutex specificata nell'argomento.

Abilitazione al risveglio

- `pthread_cond_signal(pthread_cond_t *cond);`

Controlla se c’è qualche thread in attesa con una wait sulla condition variable specificata. Se esiste allora lo abilita a svegliarsi, ma il thread chiamato deve poter ottenere la mutex prima di procedere. Dopo aver abilitato il thread la funzione termina. Se non esiste un thread in attesa da abilitare, la funzione semplicemente termina.

Non è specificato quale sia l’ordine con cui le signal abilitano i thread in attesa su una wait.

Le chiamate a signal e broadcast devono essere effettuate da un thread che detiene la mutex specificata nelle wait per quella condition variable. In tal modo si protegge l’accesso alla condition variable che è essa stessa una variabile condivisa.

In pratica, prima di chiamare la signal o la broadcast occorre prima chiamare la lock.

Per abilitare tutti i thread in wait per una condition variable si può fare una iterazione di signal oppure utilizzare la broadcast.

Domande riassuntive

- Che cosa hanno in comune i POSIX thread di uno stesso processo?

  I thread di uno stesso processo condividono le risorse (tempo di CPU) e i dati (variabili globali) del processo. Inoltre, condividono la stessa tabella dei file descriptors del processo.

- Se un processo possiede diversi POSIX thread, e quel processo genera un processo figlio, il processo figlio possiede dei thread?

  No, i processi figli non ereditano i thread del processo padre.

- Se un processo possiede diversi POSIX thread, e quel processo genera un processo figlio, i thread del processo figlio "vedono" le variabili globali del processo padre oppure no?

  Si, ma solo se le variabili NON sono static o NON sono allocate sullo stack, quindi vanno allocate nello spazio di indirizzamento del processo o bisogna usare esplicitamente una memoria condivisa.

- Se un primo thread esegue una chiamata a funzione di libreria del C, e questa produce un errore e scrive il codice d'errore nella variabile errno, un altro thread può controllare il contenuto della variabile errno per conoscere il tipo di errore capitato nel primo thread?

  Si, usando la funzione strerror_r per recuperare il codice di errore in maniera rientrante.

- Definire la Liveness.

  La liveness indica il corretto funzionamento anche in presenza di condizioni di errore, in pratica il processo può continuare a vivere e a raggiungere l'obbiettivo senza entrare in stallo.

- Definire il Busy Waiting.

  Il busy waiting consiste in un loop infinito in cui un thread controlla continuamente una data condizione finché essa non viene rispettata. Ciò causa ovviamente spreco di risorse in quanto il thread continua a chiedere l'accesso ad una risorsa non disponibile finché essa non lo diventa.

- Definire il Deadlock.

Due o più thread (o processi) sono in deadlock se sono bloccati permanentemente a causa di una richiesta comune di risorse, ogni thread (o processo) attende che una risorsa sia liberata da un altro processo, nessuno però riesce ad ottenerla in quanto tutti gli altri sono bloccati in attesa della stessa risorsa.

- Definire la Starvation.

La starvation si verifica quando un thread non riesce a ricevere le risorse di cui ha bisogno per completare il proprio lavoro (Ad esempio se non ha mai priorità sufficiente per accedere ai dati in presenza di altri thread con priorità più alta che sono in race condition su una risorsa).

- Descrivere concisamente qualche tipo di strumento, messo a disposizione dallo standard POSIX, per sincronizzare tra loro dei processi.

I principali strumenti di sincronizzazione tra thread POSIX sono le mutex e le condition variables. Le mutex (abbreviativo di mutua esclusione) permettono la gestione dell'accesso a risorse condivise (variabili o strutture globali), garantendo l'ownership delle risorse a un thread per volta durante le sezioni critiche (in genere lettura/scrittura su delle variabili da parte di più thread). Le condition variables permettono invece di far comunicare tra loro i thread e sincronizzare l'accesso alle risorse. Una condition variable è associata ad una state variable la quale è utilizzata come condizione di controllo, quando la condizione non è soddisfatta il thread che vuole accedere alla risorsa si mette in stato di wait, fino a quando il thread che sta accedendo alle risorse non lo avvisa svegliandolo, a questo punto se il thread che si trovava in wait riesce ad ottenere la mutex sulla variabile, allora può procedere, altrimenti viene messo nella ready queue e attende di ottenere la mutex.

- In linguaggio ANSI C, siano dichiarate le seguenti variabili, e siano queste variabili debitamente inizializzate:

```
pthread_mutex_t mutex;
pthread_cond_var cond;
```

considerare poi la seguente porzione di codice, in cui condizione sia una espressione da valutare:

```
while ( condizione ) {
     pthread_cond_wait( &cond, &mutex );
}
fai_qualcosa();
```

La porzione di codice, sopra riportata, effettua busy-waiting?

No, la funzione pthread_cond_wait() serve proprio per evitare il busy waiting, il thread non effettua polling continuo su una condizione ma attende di essere svegliato dal thread che sta occupando le risorse attraverso una signal o una broadcast.

- Spiegare cosa si intende con Race Condition.

Due (o più) thread sono in race condition quando tentano di accedere a delle risorse condivise contemporaneamente, causando comportamenti errati o insapettati (spesso si ha perdita di informazioni o alcune operazioni non vengono eseguite in quanto lo scheduler del sistema operativo non sincronizza efficacemente i thread in race condition) in quanto sono assenti meccanismi di controllo degli accessi alla variabile.

- Spiegare cosa si intende quando si dice che una sequenza di istruzioni viene eseguita in maniera "atomica".

Una sequenza di istruzioni è atomica (indivisibile) se durante la sua esecuzione nessun'altra istruzione può interromperla e iniziare ad eseguire. Se si ha necessità di interrompere un'istruzione atomica è necessario eseguirla da capo. Le istruzioni atomiche sono deterministiche, a parità di condizioni iniziali il risultato finale è sempre lo stesso.

- Che cos'è un Thread?

Un thread è un flusso di esecuzione all'interno di un processo che lo scheduler può far eseguire parallelamente o concorrentemente al resto del processo. Un thread condivide col processo la tabella dei file descriptors e le variabili globali, tuttavia, un thread ha un suo contesto di esecuzione, quindi un suo PID, un suo PC, uno stack e una sua variabile errno.

- Cosa vuol dire che una funzione è thread safe?

Una funzione è thread safe (o rientrante, o reentrant) se non causa problemi in lettura/scrittura su delle risorse da parte di più thread.

- Che cosa si intende con Pthread?

Un pthread è un'interfaccia per la programmazione concorrente POSIX compliant, ossia che aderisce allo standard POSIX (Per la precisione IEEE POSIX 1003.1

(1990)). Lo standard mette a disposizione varie funzioni per la creazione, la sincronizzazione e la gestione dei thread.

- A cosa serve la funzione pthread_join?

  La funzione pthread_join serve ad attendere e ricevere il risultato del thread passato come argomento, il chiamante si blocca finché il thread passato non termina; quindi, può essere usata per sincronizzare due thread di cui uno deve partire necessariamente dopo la terminazione dell'altro.

- Che cos'è un Pthread detached?

  Un pthread è detached quando il suo stack viene deallocato subito dopo la sua terminazione, così facendo si risparmia memoria; tuttavia, non sarà possibile conoscere il valore di ritorno tramite la pthread_join.

- Definire cosa fa la funzione cmpxchg con e senza il prefisso lock.

  Cmpxchg (compare-and-exchange) è un'istruzione fornita dai processori intel che permette di confrontare il valore di una variabile con un valore in un registro (EAX), se questi sono uguali la variabile viene aggiornata con un nuovo valore passato come argomento esplicito (insieme, ovviamente, alla variabile da confrontare con EAX).

  Se viene usata con il prefisso lock, allora confronto e assegnamento sono effettuati atomicamente, altrimenti no.

- Differenza tra multithreading e multi-processing?

  Il multithreading si riferisce all'esecuzione di più thread all'interno di un unico processo, mentre con multiprocessing si intende l'esecuzione simultanea di più processi su un sistema con più core o processori.

Gestione della memoria

Introduzione

Binding Address

Con il termine binding address (rilocazione) si indica l'associazione di indirizzi di memoria fisica ai dati e alle istruzioni di un programma.

Il binding avviene:

1. Durante la compilazione: Gli indirizzi vengono calcolati al momento della compilazione e resteranno gli stessi ad ogni esecuzione. Il codice generato è detto "assoluto".
    a. Vantaggi:
        i. Non richiede hardware apposito
        ii. Semplice
        iii. Molto veloce
    b. Svantaggi:
        i. Non funziona con multiprogramming
2. Durante il caricamento del programma in memoria: Il codice generato dal compilatore non contiene indirizzi assoluti ma relativi rispetto all'Instruction Pointer o ad un indirizzo di base, detto inizio del segmento.

Il loader si occupa di aggiornare tutti i riferimenti agli indirizzi coerentemente al punto iniziale di caricamento.

a. Vantaggi:
    i. Funziona con multiprogramming.
    ii. Non richiede hardware apposito.

b. Svantaggi:

i. Richiede una traduzione degli indirizzi da parte del loader, e quindi formati particolari dei file eseguibili.

3. Durante l'esecuzione del programma: L'individuazione dell'indirizzo di memoria effettivo viene eseguita durante l'esecuzione dalla MMU (Memory Management Unit), un componente hardware dedicato.

Indirizzi logici e indirizzi fisici

Ogni processo è associato ad uno spazio di indirizzamento logico. Gli indirizzi usati in un processo sono indirizzi logici, ovvero riferimenti a questo spazio di indirizzamento.

Ad ogni indirizzo logico corrisponde un indirizzo fisico. La MMU opera come una funzione di traduzione da indirizzi logici a indirizzi fisici.

Esempi di MMU

Registro di rilocazione

Se il valore del registro di rilocazione è R, uno spazio logico {0...MAX} viene tradotto in uno spazio fisico {R...MAX+R}. Ad esempio, nei processori intel 8086 esistono 4 registri base per il calcolo degli indirizzi: CS, DS, SS, ES.

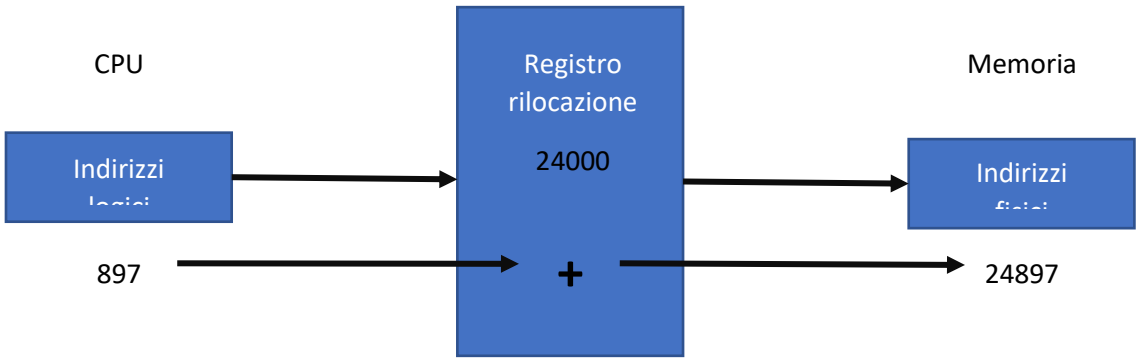


Figura 20: Registro di Rilocazione per calcolo indirizzo fisico

Registro di rilocazione e limite

Il registro viene impiegato per implementare meccanismi di protezione della memoria

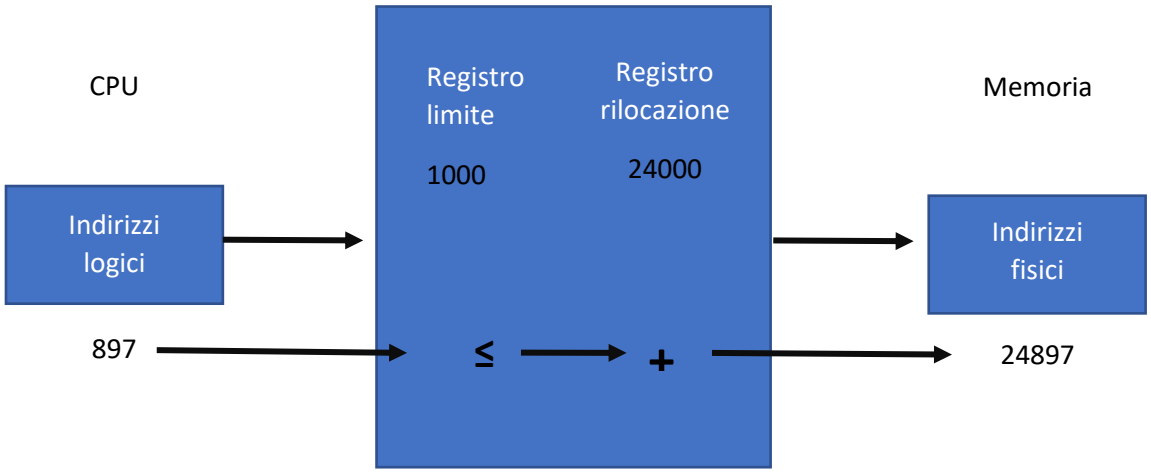


Figura 21: Protezione della memoria tramite Registro Limite

Allocazione di memoria

È una delle funzioni principali della MMU. Consiste nel reperire ed assegnare uno spazio di memoria fisica ad un programma che viene attivato, oppure per soddisfare delle richieste effettuate dai programmi durante la loro esecuzione.

Definizioni

Allocazione contigua: Tutto lo spazio assegnato ad un programma è formato da celle consecutive.

Allocazione non contigua: Lo spazio assegnato ad un programma può essere diviso in varie aree di memoria.

Allocazione statica: Un programma deve mantenere la propria area di memoria dal caricamento alla terminazione. Non si può rilocare il programma durante l'esecuzione.

Allocazione dinamica: Un programma può essere rilocato durante l'esecuzione.

Allocazione a partizioni fisse

La memoria disponibile (quella non occupata dal Sistema Operativo) viene suddivisa in partizioni. Questo tipo di allocazione è statica e contigua, risulta molto semplice ma si ha un notevole spreco di memoria; il grado di parallelismo è limitato dal numero di partizioni.

Gestione della memoria

È possibile realizzare una coda di programmi da eseguire per ogni partizione o una coda unica per tutte le partizioni.

Frammentazione interna

Se un processo occupa una dimensione inferiore a quella della partizione che lo contiene, lo spazio non utilizzato viene sprecato. La presenza di spazio inutilizzato all'interno di una partizione è detta frammentazione interna.

Allocazione a partizioni dinamiche

La memoria disponibile viene assegnata ai processi che ne fanno richieste. Nella memoria possono esserci zone inutilizzate per effetto della terminazione dei processi o perché i processi non utilizzano tutta l'area disponibile. Questo tipo di allocazione è statica e contigua.

Frammentazione esterna

Dopo alcune allocazioni e deallocazioni di memoria dovute ad esecuzioni e terminazioni di processi lo spazio di memoria libero appare suddiviso in piccole aree.

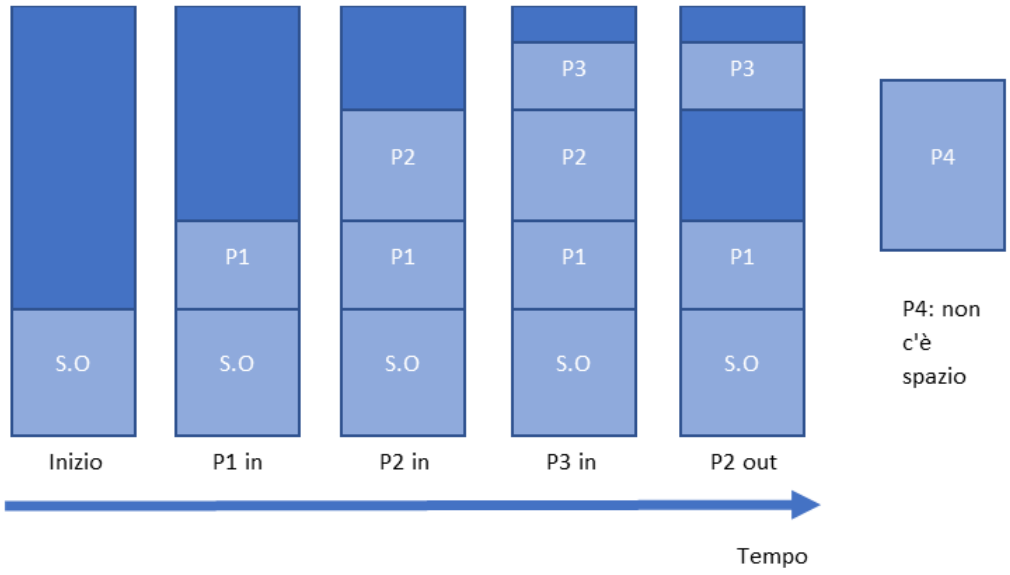


Figura 22: Frammentazione esterna con partizionamento dinamico

Compattazione

Se è possibile rilocare i programmi durante l'esecuzione, allora è possibile procedere alla compattazione della memoria. Ciò consiste nello spostare tutti i programmi in modo da unificare le aree di memoria inutilizzate.

Questa operazione è molto dispendiosa (Occorre copiare fisicamente grandi quantità dati in memoria), inoltre non può essere utilizzata in sistemi interattivi, i processi devono essere fermi durante la compattazione.

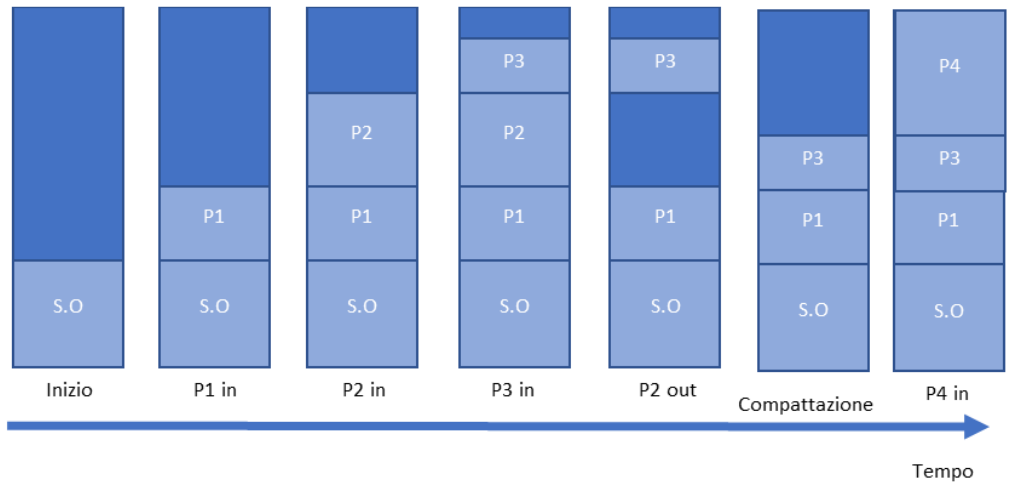


Figura 23: Meccanismo di Compattazione

Selezione del blocco libero

First Fit: Scorre la lista dei blocchi liberi, partendo dall'inizio della lista, fino a quando non trova il primo segmento vuoto di dimensioni sufficienti da contenere il processo.

Next Fit: Come first fit, ma invece di ripartire dall'inizio della lista parte dal punto in cui si era fermato all'ultima allocazione. Più lento di First Fit.

Best Fit: Seleziona il più piccolo blocco libero capace di contenere il processo. Più lento di First Fit e causa più frammentazione.

Worst Fit: Seleziona il più grande fra i blocchi presenti in memoria. È difficile allocare processi di grandi dimensioni.

Strutture dati

È possibile ottimizzare il costo di allocazione mantenendo una lista separata per i soli blocchi liberi e, eventualmente, ordinando tale lista per dimensione. Per mantenere queste informazioni si può

usare la tabella dei processi per i blocchi occupati, mentre i blocchi liberi possono contenere le informazioni necessarie dentro sé stessi.

Paginazione

I meccanismi visti (partizionamento fisso o dinamico) non sono efficienti nell'uso della memoria. I sistemi moderni utilizzano il meccanismo della paginazione, grazie alla quale si riduce il fenomeno della frammentazione interna ed elimina quella esterna.

Lo spazio di indirizzamento logico è diviso in blocchi di dimensione fissa chiamati pagine. La memoria fisica viene suddivisa in un insieme di blocchi della stessa dimensione delle pagine chiamati frame.

Quando un processo viene allocato in memoria, vengono reperiti ovunque in memoria un numero sufficiente di frame per contenere le pagine del processo.

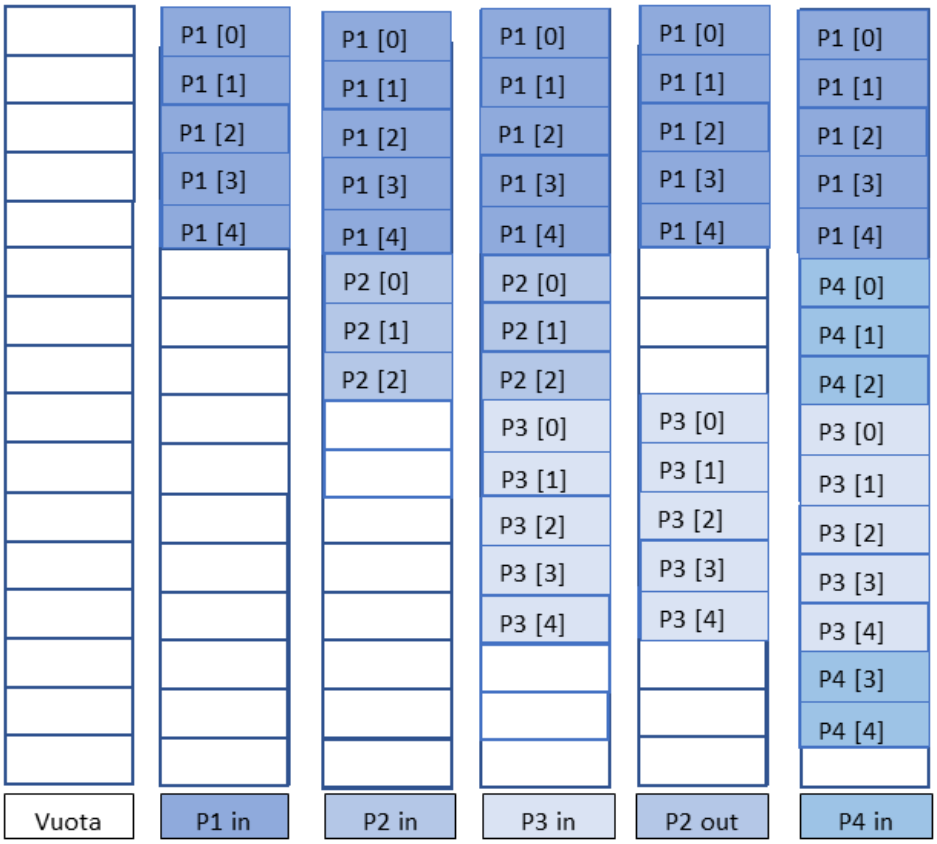


Figura 24: Paginazione

Dimensione delle pagine

La dimensione deve essere una potenza di due per semplificare la trasformazione da indirizzi logici a fisici. La scelta della dimensione deriva da un trade-off. Pagine troppo piccole causano una crescita in dimensioni della tabella. Pagine troppo grandi causano una frammentazione interna più importante. Dimensioni tipiche delle pagine sono 1KB, 2KB, 4KB, 4MB.

MMU per la paginazione

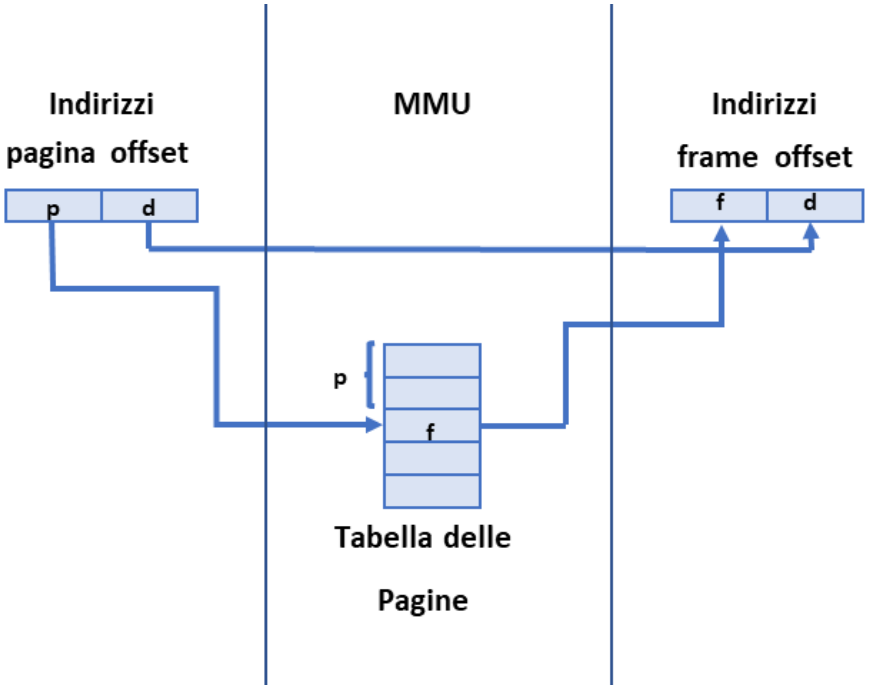


Figura 25: MMU per paginazione

Implementazione della page table

La tabella delle pagine si può caricare in dei registri dedicati, ma sarebbe un'operazione troppo costosa (In un processore a 32bit con dimensione di pagina di 4K ci sarebbero 1.048.576 pagine nella page table).

Un'alternativa meno costosa sarebbe caricarla in memoria, tuttavia il numero di accessi verrebbe raddoppiato; ad ogni riferimento, bisognerebbe prima accedere alla page table, poi al dato.

La soluzione è implementare una memoria cache per la tabella delle pagine, chiamata Translation Lookaside Buffer (TLB).

Segmentazione

In un sistema con segmentazione la memoria associata ad un programma è suddivisa in aree differenti dal punto di vista funzionale.

Ad esempio:

- Aree text: Contengono il codice eseguibile, sono (normalmente) read-only, possono essere condivise tra più processi.
- Aree dati: Possono essere condivise o no.
- Area stack: Read/Write, non può essere assolutamente condivisa.

In un sistema basato su segmentazione, uno spazio di indirizzamento logico è dato da un insieme di segmenti. Un segmento è un'area di memoria (logicamente continua) contenente elementi tra loro affini. Ogni segmento è caratterizzato da un nome (normalmente un indice) e da una lunghezza. Ogni riferimento di memoria è dato da una coppia {nome, offset}.

Spetta al programmatore o al compilatore la suddivisione in segmenti del programma.

Confronto

Paginazione	Segmentazione
La divisione in pagine è automatica	La divisione in segmenti spetta al programmatore
Le pagine hanno dimensione fissa	I segmenti hanno dimensione variabile
Le pagine possono contenere informazioni disomogenee	Un segmento contiene informazioni omogenee
Una pagina ha un indirizzo	Un segmento ha un nome
La dimensione tipica della pagina è 1-4KB	La dimensione tipica di un segmento è 64KB - 1MB

Supporto hardware per segmentazione

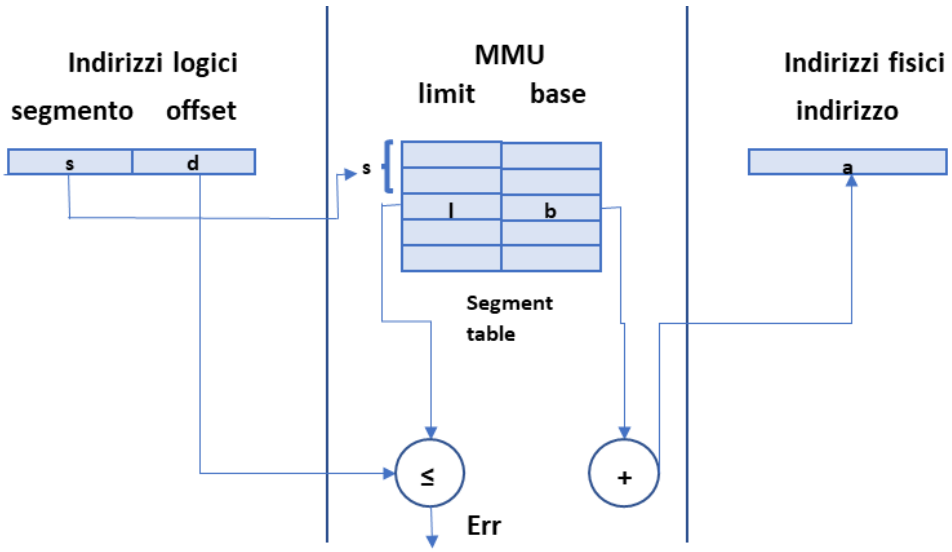


Figura 26: MMU per segmentazione

Segmentazione e condivisione

La segmentazione consente la condivisione di codice e dati.

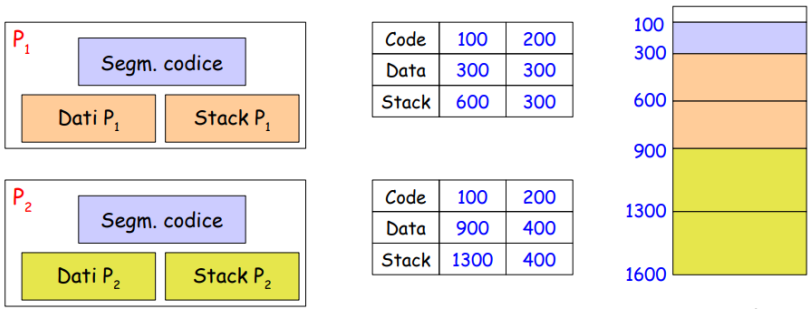


Figura 27: Esempio segmentazione e condivisione

© 2002-2005 Renzo Davoli, Alberto Montresor

Segmentazione e frammentazione

Allocare segmenti di dimensione variabile è del tutto equivalente al problema di allocare in modo contiguo la memoria dei processi.

È possibile utilizzare tecniche di allocazione dinamica o la compattazione. Tuttavia, così si ritorna ai problemi precedenti.

Segmentazione e paginazione

È possibile utilizzare il metodo della paginazione combinato alla segmentazione. Ogni segmento viene suddiviso in pagine che vengono allocate in frame per la paginazione.

Si ottengono così sia i benefici della segmentazione (condivisione, protezione) che quelli della paginazione (eliminazione della frammentazione esterna).

Memoria virtuale

È la tecnica che permette l'esecuzione di processi che non sono completamente in memoria. Permette di eseguire in concorrenza processi che nel loro complesso hanno necessità di memoria superiore di quella disponibile. La memoria virtuale può diminuire le prestazioni del sistema se implementata nel modo sbagliato.

I requisiti di un'architettura di Von Neumann richiedono che le istruzioni da eseguire e i dati su cui operano devono essere in memoria.

Tuttavia, ciò non significa che tutto lo spazio di indirizzamento logico sia in memoria. I processi non utilizzano tutto il loro spazio di indirizzamento contemporaneamente.

Ogni processo ha accesso ad uno spazio di indirizzamento virtuale che può essere più grande di quello fisico. Gli indirizzi virtuali possono essere mappati su indirizzi fisici della memoria principale, oppure, possono essere mappati su una memoria secondaria (disco).

In caso di accesso ad indirizzi virtuali mappati in memoria secondaria, i dati associati vengono trasferiti in memoria principale, se la memoria è piena si sposta in memoria secondaria i dati contenuti in memoria principale che sono considerati meno utili.

Nella memoria virtuale si utilizza la paginazione a richiesta (demand paging), in cui si fa uso della tecnica della paginazione, ammettendo però che alcune pagine possano trovarsi in memoria secondarie. Nella tabella delle pagine si usa un bit che indica se la page è in memoria centrale o no.

Quando un processo tenta di accedere ad una pagina che non si trova in memoria, il processore genera un'eccezione di tipo FAULT (page fault), il pager (un componente del sistema operativo) si occupa di caricare la pagina mancante in memoria, aggiornando la tabella delle pagine.

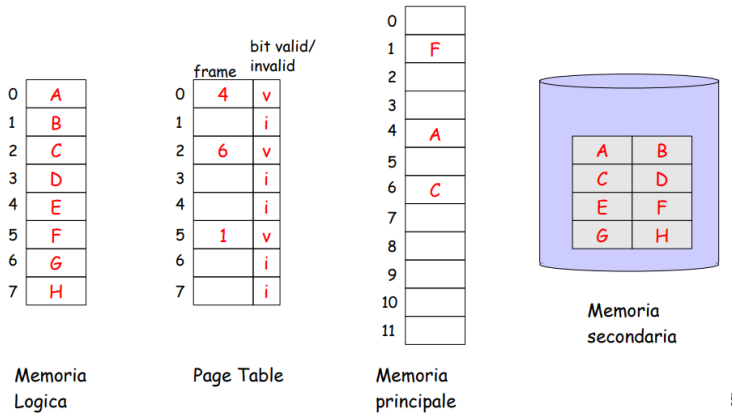


Figura 28: Esempio Memoria Virtuale

© 2002-2005 Renzo Davoli, Alberto Montresor

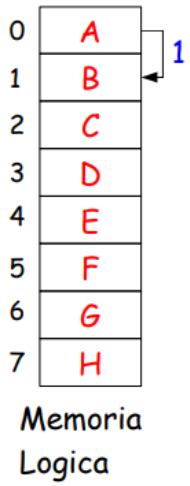
Pager/Swapper

Con il termine swap si intende l'azione di copiare l'intera area di memoria usata da un processo dalla memoria secondaria a quella principale (swap-in) e viceversa (swap-out).

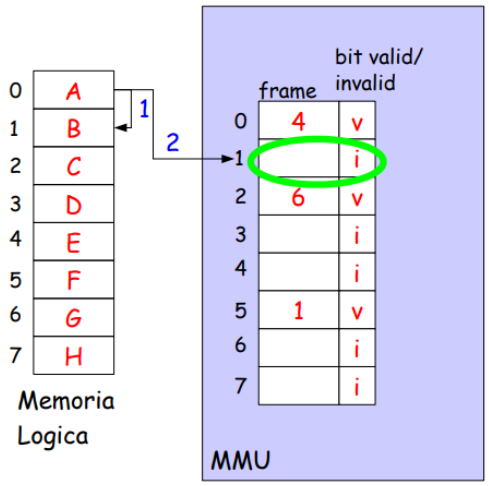
Questa tecnica si utilizzava in passato quando demand paging non esisteva.

La paginazione su richiesta può essere vista come una tecnica di lazy swap con la quale viene caricato in memoria solo ciò che serve. Quindi alcuni sistemi operativi indicano il pager con il nome di swapper, che è però una terminologia obsoleta. Tuttavia, ancora oggi, si utilizza il termine swap-area per indicare l'area del disco utilizzata per ospitare le pagine in memoria secondaria.

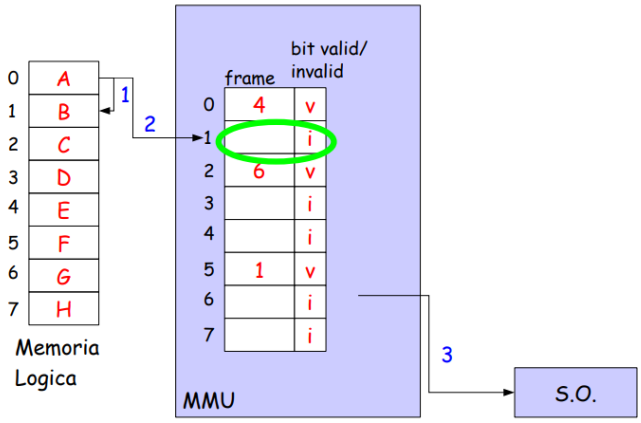
- Supponiamo che un'istruzione macchine del codice in pagina 0 faccia riferimento alla pagina 1:



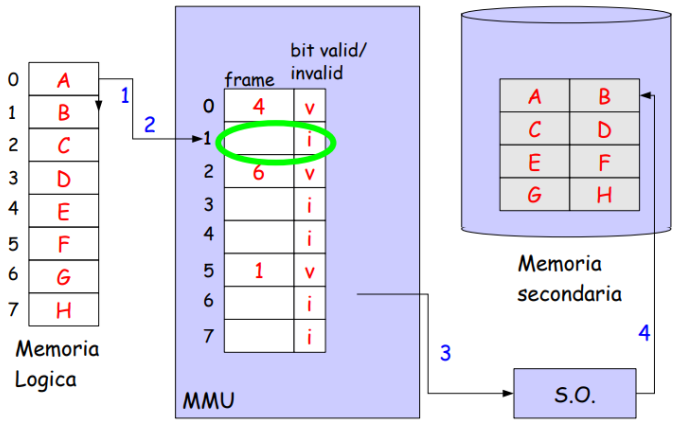
- La MMU scopre che la pagina 1 non è in memoria principale



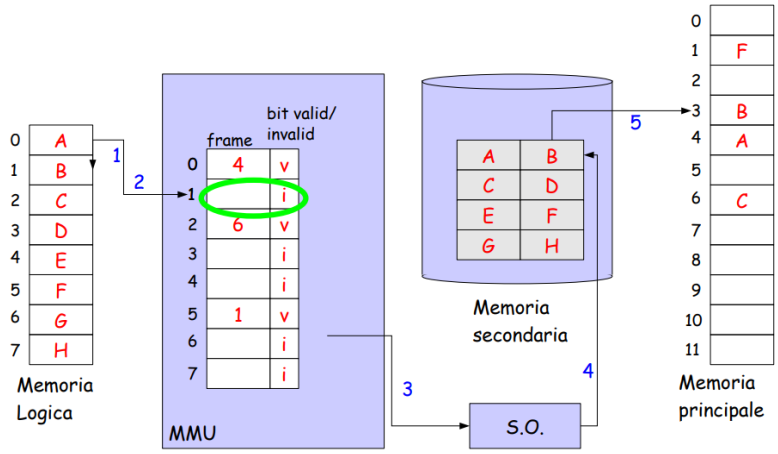
- Viene generata un'eccezione di tipo FAULT "Page Fault", catturata dal S.O.



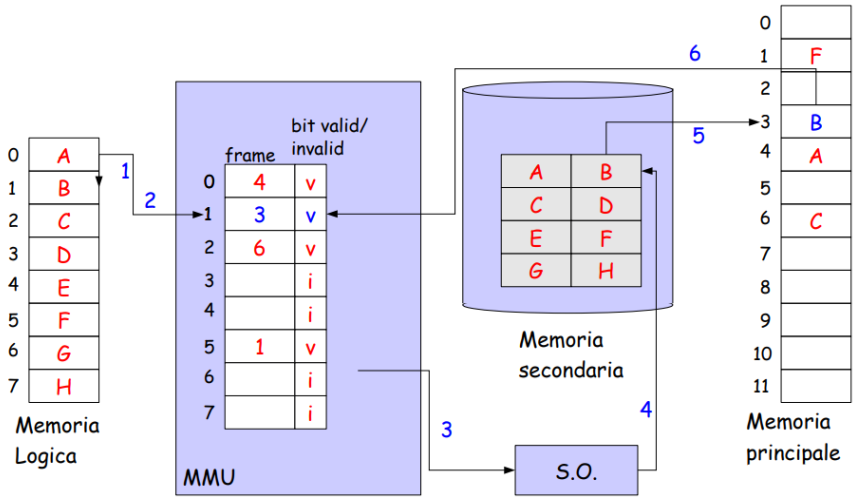
- Il S.O. cerca in memoria secondaria la pagina



- Il S.O. carica in memoria principale il contenuto della pagina



- Il S.O. aggiorna la page table e riavvia l'esecuzione dell'istruzione



© 2002-2005 Renzo Davoli, Alberto Montresor

- In mancanza di frame liberi è necessario liberarne uno, si sceglie la pagina meno utile.

Algoritmi di sostituzione o rimpiazzo

Demand paging:

1. Individua la pagina in memoria secondaria (disco);

2. Individua un frame libero;
3. Se non esiste un frame libero:
    a. Richiama algoritmo di rimpiazzo;
    b. Aggiorna la tabella delle pagine (Invalida la pagina da rimuovere);
    c. Se la pagina invalidata è stata cambiata, scrive la pagina sul disco;
    d. Aggiorna la tabella dei frame (Ora c'è un frame libero);
4. Aggiorna la tabella dei frame (Frame occupato);
5. Legge la pagina da disco (Quella che ha provocato il fault);
6. Aggiorna la tabella delle pagine;
7. Riattiva il processo.

On demand puro

È possibile avviare l'esecuzione di un processo senza pagine in memoria:

Quando il sistema operativo carica nel Program Counter l'indirizzo della prima istruzione, genera un primo page fault.

Una volta portata la prima pagina in memoria, le altre vengono caricate quando sono indirizzate perché contengono dati o codice.

In un sistema con memoria virtuale basata su demand paging è importante mantenere un numero di page fault basso, altrimenti il tempo di accesso effettivo aumenta moltissimo, rallentando visibilmente l'esecuzione dei processi. È fondamentale scegliere bene il meccanismo di sostituzione delle pagine.

Sostituzione delle pagine

Se nessun frame è libero è possibile liberarne uno scaricando il suo contenuto dalla memoria principale alla backing store. La sostituzione delle pagine (una successione di swap-out della vecchia pagina e uno swap-in di quella nuova) deve essere registrata nella tabella delle pagine.

Dirty Bit

Lo scaricamento si può evitare se la pagina non è stata modificata, per verificare tale condizione si utilizza un bit aggiuntivo detto dirty bit (o bit di modifica), gestito via hardware: Quando la pagina è caricata vale 0, quando è scritta vale 1. Quando la pagina è scaricata viene eseguita la scrittura su disco solo se il dirty bit vale 1.

Domande riassuntive

- Cos'è e a cosa serve il DMA?

  Il DMA (Direct Memory Access) è una tecnologia che consente a un dispositivo di accedere direttamente alla memoria del computer senza dover passare attraverso il processore.

  Ciò significa che il dispositivo può trasferire dati direttamente dalla memoria alla periferica. Ciò rende il trasferimento dei dati più efficiente e diminuisce il carico sulla CPU.

- Spiegare che cosa è la rilocazione statica.

  Con rilocazione statica si intende l'allocazione di processi in memoria che non possono essere spostati una volta avviati. In pratica i processi allocati staticamente devono mantenere il proprio spazio di memoria dall'avvio dell'esecuzione alla terminazione.

- Una stessa struttura dati ha la stessa dimensione in tutti i processori? Perché?

  No, dipende dall'architettura della CPU (64 o 32 bit), la dimensione potrebbe variare anche in base al sistema operativo (dipende dalla strategia di allocazione usata).

- Spiegare come dovrebbe funzionare l' "algoritmo ottimale" di sostituzione delle pagine.

  L'algoritmo ottimale di sostituzione delle pagine dovrebbe mettere in memoria una pagina subito prima che il sistema ci acceda, sostituendo la pagina che dovrebbe essere utilizzata più tardi. In pratica l'algoritmo dovrebbe "predire il futuro" conoscendo quando ogni singola pagina verrà utilizzata.

- Spiegare come funziona l'algoritmo "Least Recently Used (LRU)" di sostituzione delle

  pagine.

  L'idea di LRU è che se una pagina non è utilizzata di recente, è meno probabile che venga utilizzata di recente. Si usa una struttura dati per mantenere traccia delle pagine in memoria, ogni volta che una pagina viene usata, viene spostata in cima alla lista, in modo che la pagina in fondo sia quella che è stata usata meno di recente. Quando la memoria è piena la pagina infondo alla lista viene sostituita.

- Che cos'è il thrashing?

Il thrashing si verifica quando il sistema operativo è costantemente impegnato a sostituire le pagine in memoria, senza che si verifichi un aumento di prestazioni. Ciò si verifica ad esempio quando la memoria non è sufficiente per supportare l'esecuzione dei processi in corso. In questo caso la CPU viene impegnata per la sostituzione delle pagine, riducendo significativamente le prestazioni generali. Il thrashing può essere evitato aumentando la memoria o modificando l'algoritmo di sostituzione utilizzato oppure usando il paging della memoria.

- Nell'ambito della paginazione per la memoria virtuale, cosa si intende con "allocazione dei frame ai processi".

L'allocazione dei frame ai processi si riferisce al processo di assegnazione di un certo numero di frame di memoria fisica per ogni processo in esecuzione sul sistema. Quando viene avviato un processo il sistema operativo assegna un certo numero di frame di memoria per contenere le pagine del processo. Questi frame vengono usati per contenere le pagine del processo in memoria. Il numero di frame assegnati al processo varia dalle dimensioni del processo e dalla quantità di memoria disponibile sul sistema. L'allocazione dei frame ai processi permette di isolare i processi, garantendo che un processo non possa accedere alle pagine di un altro e garantendo che un errore in un processo non causi problemi agli altri.

- Spiegare a cosa serve il procedimento di sostituzione delle pagine nell'ambito della gestione della memoria virtuale.

Il procedimento di sostituzione delle pagine serve per gestire la quantità di memoria disponibile sul sistema. La memoria virtuale consente ai processi di usare più memoria di quella fisicamente presente. Tuttavia, poiché la quantità di memoria fisica è limitata, è necessaria una gestione efficiente. Quando una nuova pagina è richiesta e non c'è abbastanza memoria disponibile, il sistema seleziona una pagina in memoria e la scrive sul disco (swap-out), inserendo in memoria la pagina richiesta (swap-in).

- In che contesto si parla di "dirty bit" e a che cosa serve?

Il dirty bit è un flag utilizzato nella gestione della memoria virtuale, che indica se una pagina è stata modificata o meno. Se il bit è a 0 la pagina non è stata modificata, può quindi semplicemente essere rimossa dalla memoria, altrimenti se è a 1 la pagina è stata modificata e quindi è necessario prima copiarla sul disco per poterla rimuovere dalla memoria.

- Spiegare cosa avviene allo scatenarsi di un page fault.

Un page fault è un’eccezione di tipo FAULT che si verifica quando il processore tenta di accedere a una pagina di memoria che non è attualmente caricata in memoria. Quando avviene un page fault il pager carica la pagina dal disco nella memoria fisica (swap-in) e la CPU riprende l’esecuzione

File system

Concetti principali

Per la maggior parte degli utenti il file system è l'aspetto più visibile del Sistema Operativo. Esso rappresenta un'astrazione del modo in cui i dati sono allocati e organizzati su un dispositivo di memoria di massa.

Attraverso il file system, il S.O. offre una visione logica uniforme della memorizzazione delle informazioni sui diversi supporti.

L'elemento base nella gestione a livello logico della memoria di massa è il file.

File

Un file è un insieme di informazioni correlate e registrate nella memoria secondaria con un nome.

Il file è la più piccola unità di memoria secondaria assegnabile all'utente che può scrivere sulla memoria secondaria solo registrando un file.

Il file può avere una sua struttura interna, a seconda del formato o del tipo. Per esempio, un file eseguibile è una sequenza di sezioni di codice che possono essere caricate ed eseguite.

Attributi

Gli attributi principali di un file sono:

1. Nome: È l'unica informazione mantenuta in un formato simbolico leggibile da una persona.
2. Tipo: Usato nei sistemi che supportano tipi di file diversi.
3. Locazione: Puntatore al dispositivo e alla locazione di quel file sul dispositivo.
4. Dimensione.

5. Protezione: Informazioni di controllo che specificano chi può leggere, scrivere o eseguire il file.

6. Ora, data e identificazione dell'utente: Informazioni usate per funzioni di protezione dei dati e di monitoring. Possono essere riferite alla creazione, all'ultima data di utilizzo o all'ultima lettura eseguita.

Directory

Il file system può essere molto ampio (contiene potenzialmente migliaia di files). È tipicamente organizzato in una struttura gerarchica, la quale mantiene gli elementi terminali all'interno di contenitori detti directory.

File in uso

Il sistema mantiene in memoria l'elenco dei file in uso in una tabella apposita (Tabella dei file aperti). Quando viene iniziata un'operazione sul file, il file viene inserito nella tabella in modo che ogni successiva operazione non comporti fasi di ricerca.

Informazioni sui file aperti

A ciascun file aperto sono associate diverse informazioni:

- Un puntatore alla posizione corrente del file: per il supporto a read e write. Se diversi processi operano sul file, esistono diversi puntatori alla posizione corrente.

- Contatore delle aperture: Se più processi possono aprire il file, il S.O. tiene conto delle aperture avvenute. Quando sono tutte chiuse, rimuove il file dalla tabella dei file aperti.

- Posizione su disco dei file: Quando il file è aperto viene memorizzata la sua posizione sul dispositivo per evitare di doverlo cercare nuovamente ad ogni operazione.

Struttura dei file

Tipicamente il Sistema Operativo non si occupa di gestire la struttura interna associata ai file, ma lascia che siano le applicazioni a definirla e utilizzarla.

Per il Sistema i files sono sostanzialmente insiemi ordinati di byte, i formati dei vari file sono gestiti a livello applicazione.

In questo modo si riduce fisicamente la dimensione e la complessità del codice del Sistema Operativo, e si ha una maggiore flessibilità nella definizione di nuovi formati di files.

Tutti i Sistemi Operativi supportano il formato eseguibile, per poterli caricare in memoria ed eseguire.

Metodi di accesso

Il Sistema Operativo offre diversi metodi di accesso ai file:

- Sequenziale: Le informazioni del file vengono elaborate in ordine (es: nastri).
- Diretto: Il file è costituito da un insieme di blocchi, ordinati, a cui si accede direttamente (es: dischi).
- Indicizzato: Al file è associato un file indice in modo da velocizzarne la ricerca (es: database).

Accesso Sequenziale

Le operazioni possibili con un accesso di tipo sequenziale sono:

- Read: legge la porzione del file successiva e avanza il puntatore di posizione.
- Write: scrive i nuovi dati in coda al file e avanza l'EoF.

- Reset: Riporta il puntatore di posizione all'inizio del file.

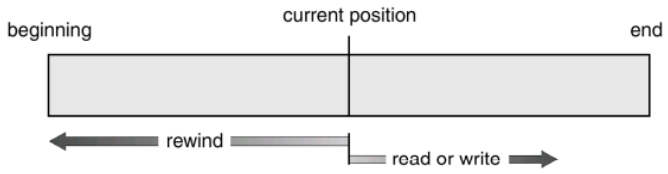


Figura 29: Accesso sequenziale

Accesso diretto

I dispositivi ad accesso casuale (non sequenziale) utilizzano un modello di accesso diretto (o relativo). All'interno di tali dispositivi i file sono costituiti da record logici di lunghezza fissa (blocchi), i quali vengono considerati come un insieme ordinato numerato, accessibili arbitrariamente.

Il numero di blocco è solitamente relativo (Ogni file ha come blocco di inizio il blocco 0).

Le operazioni consentite sono:

- Lettura: read(n), legge il blocco con posizione relativa n.
- Scrittura: write(n), scrive il blocco con posizione relativa n.
- Riposizionamento: seek(n), si posizione sul blocco con posizione relativa n.

Accesso Indicizzato

Ad ogni file viene associato un indice, il quale viene registrato su un file, questa tecnica permette di ricercare file in maniera molto veloce. L'indice contiene una o più chiavi di ricerca alle quali sono associati i puntatori ai diversi blocchi del file.

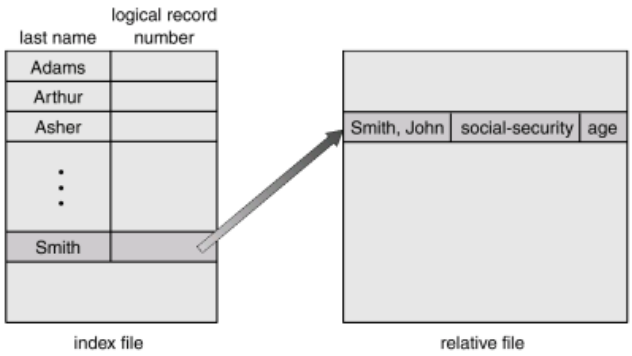


Figura 30: Accesso indicizzato

Strutture del File System

File Control Block

Per il sistema operativo i files sono memorizzati in un opportuno descrittore, detto File Control Block, il quale contiene:

- Nome del file
- Proprietario
- Dimensioni
- Informazioni sui permessi
- Posizioni dei blocchi

Il descrittore occupa un blocco su disco, ed è chiamato inode nei sistemi Unix.

Allocazione dei blocchi

Un elemento che incide fortemente sulle prestazioni è il metodo di allocazione dei blocchi utilizzato. Esso specifica come i blocchi del disco sono allocati ai diversi file.

Ci sono tre tecniche di allocazione dei blocchi:

- Allocazione Contigua (Contiguous Allocation)
- Allocazione Concatenata (Linked Allocation)
- Allocazione Indicizzata (Indexed Allocation)

Allocazione Contigua

Nell'allocazione contigua i file occupano insiemi di blocchi contigui sul disco. È un tipo di allocazione molto semplice. Questo sistema di allocazione soffre di frammentazione esterna (risolta con una routine di deframmentazione), similmente alla gestione della memoria le strategie di allocazione maggiormente utilizzate sono worst fit e best fit. Bisogna trovare una porzione di disco sufficientemente grande da contenere tutto il file.

Nell allocazione contigua:

- Gli indirizzi del disco definiscono un ordinamento lineare dei blocchi.
- L'accesso sequenziale al blocco b+1 in seguito all'accesso al blocco b non richiede lo spostamento della testina.
- L'accesso diretto è fatto calcolando la posizione assoluta in base a quella relativa.
- Il file è definito dalla posizione iniziale e dalla lunghezza

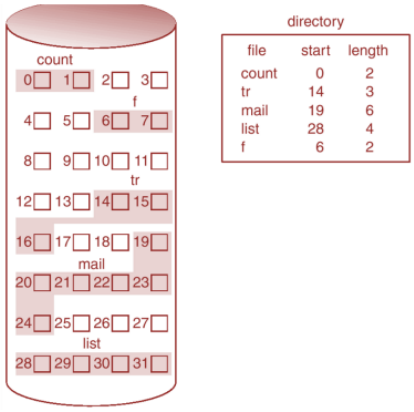


Figura 31: Allocazione Contigua

Allocazione concatenata

Per risolvere i problemi di frammentazione dell'allocazione contigua è possibile ricorrere all'allocazione concatenata.

Ogni file è costituito da una lista concatenata di blocchi del disco, i quai possono essere distribuiti ovunque. La directory contiene un puntatore al primo blocco del file.

Nonostante con questa tecnica si eliminino i fenomeni della frammentazione esterna e interna (i blocchi sono allocati singolarmente e non è necessario pre-allocare il file), questa tecnica di allocazione risulta poco efficiente nell'accesso diretto (bisogna, infatti, scorrere l'intero file fino a trovare la posizione cercata), inoltre ogni blocco contiene un puntatore al successivo, causando un'occupazione di spazio decisamente maggiore dell'allocazione contigua.

Clustering

Per risolvere il problema dell'occupazione dello spazio solitamente i blocchi non si allocano singolarmente ma a blocchi (cluster). Ciò riporta comunque il problema della frammentazione interna (una parte di un blocco potrebbe rimanere inutilizzata).

Allocazione indicizzata

La maggior parte dei problemi riscontrati dall'allocazione contigua e da quella concatenata si può risolvere inserendo tutti i puntatori ai blocchi in una tabella detta index block.

Ogni file ha un proprio index block, in cui l'-esimo elemento (dell'index block) punta all'i-esimo elemento del file.

In questo modo è possibile effettuare accessi diretti senza frammentazione; tuttavia, si ha ancora spreco di spazio, in quanto è necessario allocare della memoria per mantenere l'index block.

Esistono tre tecniche di allocazione indicizzata:

- Schema concatenato: L'index block occupa esattamente un blocco. Se non è sufficiente l'ultimo puntatore punta ad un altro index block.
- Indice multilivello: L'index block di primo livello punta ad altri blocchi indice di secondo livello e così via.
- Schema combinato: Viene usato negli inode di alcune versioni di Unix, una parte dei puntatori del blocco indice puntano direttamente ai blocchi del file (blocchi diretti), mentre altri puntano ad indici multilivello (blocchi indiretti singoli, doppi o tripli). Quindi se il file è piccolo vengono impiegati solo blocchi diretti, al crescere delle dimensioni aumenteranno anche le indicizzazioni multilivello coinvolte.

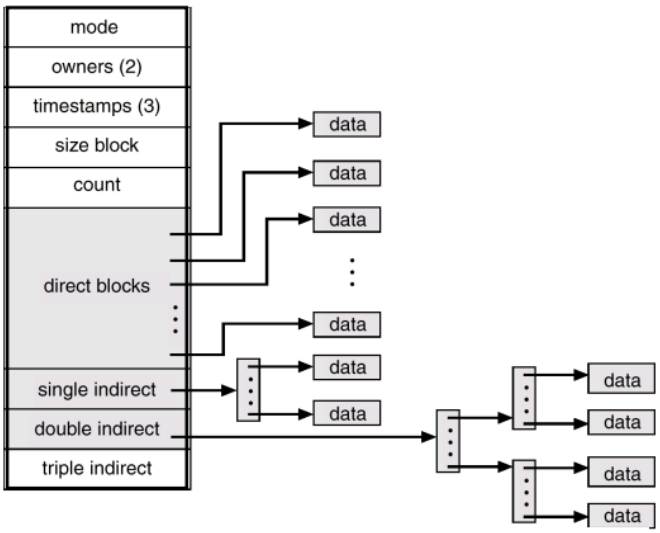


Figura 32: inode, Schema Combinato

Architetture per la virtualizzazione di sistemi

Virtualizzazione o Emulazione

Tramite la virtualizzazione è possibile eseguire uno o più sistemi operativi (e le relative applicazioni) su un unico PC, in un ambiente protetto e monitorato che prende il nome di Virtual Machine.

Il sistema su cui viene eseguita la macchina virtuale è detto host, la macchina virtuale è detta guest.

Il codice della macchina virtuale viene eseguito direttamente dall'host; nonostante ciò, il sistema guest non vede questo layer di astrazione e rileva la macchina virtuale come se fosse eseguito su una macchina fisica.

Il codice della macchina virtuale deve quindi essere codice macchina eseguibile dalla macchina reale sottostante. (Per fare un esempio, non si può virtualizzare un sistema operativo che dovrebbe girare su x86 a 64 bit su un processore ARM).

L’emulazione di un processore invece avviene tramite un programma di controllo, ogni istruzione che il guest esegue viene tradotta in una sequenza di istruzioni della macchina host. Il processo di emulazione risulta più lento rispetto alla virtualizzazione, in quanto è necessario tradurre ogni singola istruzione dal formato del sistema ospite a quello ospitante.

Virtualizzazione del livello Hardware o del livello Software

Parlando di virtualizzazione è necessario fare una prima distinzione a seconda delle risorse virtuali che si vogliono presentare all’utente. Bisogna distinguere le macchine virtuali dalle partizioni isolate (dette anche container).

Nel caso delle VM (virtualizzazione a livello Hardware) viene offerta all’utente un’interfaccia su cui installare il sistema operativo, una CPU virtuale (dello stesso tipo di quella fisica) e altre risorse Hardware virtuali.

I container (virtualizzazione a livello del Sistema Operativo), invece, presentano una partizione del sistema operativo corrente, su cui installare ed eseguire applicazioni che restano isolate nella partizione, pur accedendo ai servizi di uno stesso Sistema Operativo.

Confronto tra virtualizzazione Hardware o del S.O.

La virtualizzazione a livello del Sistema Operativo è composta da un unico kernel (quello del sistema ospitante) e istanze isolate di user-spaces (un altro sinonimo di container), le quali possono essere avviate e spente indipendentemente tra loro. Ogni container ha un proprio filesystem e proprie interfacce di rete. Viene garantito l’isolamento del filesystem, dell’IPC (Inter-Process Comunication) e della rete. Inoltre, viene fornito un sistema di gestione delle risorse (CPU, memoria, rete e I/O).

Nella virtualizzazione hardware i sistemi operativi eseguono in maniera concorrente sullo stesso hardware e possono essere eterogenei. L'hypervisor (Il software che permette la virtualizzazione) fa da multiplexer per l'accesso alle risorse hardware e garantisce l'isolamento (e la protezione) tra le due macchine.

I vantaggi di utilizzare dei container sono diversi. L'overhead (il delta del consumo di risorse dovuta al layer di astrazione) per il context-switch e per gli accessi in memoria è molto basso. Tuttavia, il prezzo da pagare è che non si possono ospitare sistemi operativi diversi e l'isolamento non può essere totale.

I vantaggi della virtualizzazione a livello hardware sono speculari rispetto a quelli dei container (aumenta l'overhead a causa del layer di astrazione in più rappresentato dall'hypervisor, ma si ha un isolamento migliore e può girare qualsiasi sistema operativo compatibile con l'hardware della CPU fisica).

Perché usare i Container

Perché creano uno spazio utente isolato, con proprie interfacce di rete, librerie e files, isolando un'applicazione dal resto del sistema operativo.

Si possono far girare più container sullo stesso kernel, risparmiando spazio su disco e condividendone i servizi di base.

Una volta costruito e configurato un container per una specifica applicazione, posso replicare quel container anche su un'altra VM.

Principali Container

- Docker: Basato su Linux che fornisce un supporto a livello kernel chiamato LXC (Linux Container).
- Hyper-V: sviluppato da Microsoft, fornisce delle unità di isolamento chiamate container, che in realtà sono macchine virtuali).
- Container di Windows Server.

Docker

I container Docker sono eseguiti a partire da immagini di container che possono essere scaricate dal repository Docker Hub. È possibile scaricare un'immagine Docker ed eseguirla localmente- Dopo aver scaricato l'immagine, la runtime machine di Docker crea il container partendo dall'immagine, una cui copia verrà salvata su un registro locale.

È possibile eseguire dei container interattivi. Usando i flags –i –t consente di interagire con una shell bash all'interno del container connessa mediante stdin/stdout/stderr alla shell in cui lanciamo il container in foreground (In questo esempio useremo l'immagine di Ubuntu disponibile per Docker).

```
docker run -it --name myubuntu ubuntu
```

L'opzione --name myubuntu assegna il nome myubuntu al container. Se non specificato il nome del container è assegnato casualmente.

Il prompt dovrebbe ora cambiare in qualcosa del tipo: root@d9b100f2f636:/#. Ciò ci indica che stiamo ora lavorando dentro al container.

La sequenza alfanumerica dopo la "@" è l'ID del container.

A questo punto si può eseguire qualsiasi comando all'interno del container myubuntu. Ogni cambiamento del filesystem del container resta isolato dal resto. Per terminare il container bisogna digitare exit. La shell termina e con essa termina il container.

Se fai partire nuovamente il container dall'immagine ubuntu, i file creati precedentemente creato non c'è più nel nuovo container.

Installare applicazioni

Per installare un'applicazione nel container myubuntu è sufficiente utilizzare i comandi di ubuntu per l'installazione dei pacchetti.

Salvare i cambiamenti

Dopo aver eseguito un container interattivamente è possibile creare un'immagine locale, salvando così i cambiamenti fatti (L'immagine conterrà quindi le applicazioni installate precedentemente e tutti gli eventuali file creati durante l'uso del container).

```
Docker commit –it –name myContainer /path/to/image
```

Esecuzione in background

Il comando per l'esecuzione in background di un container esponendo una porta all'esterno è:

```
docker run -it -d --rm -p 80:8888 --name myContainer
path/to/image node index.js
```

Segue una breve spiegazione dei parametri usati:

- -d serve ad eseguire il container in background.
- --rm elimina il container quando termina.
- -p 80:8888 connette la porta 80 dell'host sulla porta 8888 del container.

- node è il comando da lanciare non appena viene creato.
- index.js è l'argomento passato al comando node.

Kubernetes

Kubernetes permette di fornire automaticamente scalabilità e resilienza ad applicazioni basate su container docker in esecuzione su cluster di host fisici o virtuali.

Applicazioni per Kubernetes

Le applicazioni per Kubernetes sono costituite da più pods. Ciascun pod è un insieme di container che cooperano e comunicano tra loro come se si trovassero in un unico host fisico e svolgono un certo servizio. Ad esempio, un pod può realizzare un servizio web, mentre un altro può eseguire dei calcoli o salvare dati in un database.

Un pod può essere replicato in più istanze se serve eseguire le stesse operazioni in parallelo, distribuendo il carico. Le diverse istanze dei pods possono eseguire su uno stesso host oppure su host diversi per decentralizzare il carico.

Ciascun pod può esporre delle porte di protocollo per ricevere comunicazioni da container di altri pods.

In tal modo, uno dei pod di tipo A, che espone il servizio web, riceve le richieste degli utenti web esterni, poi chiede ad uno dei pod di tipo B di effettuare un calcolo, ricevuto il risultato lo restituisce all'utente web esterno.

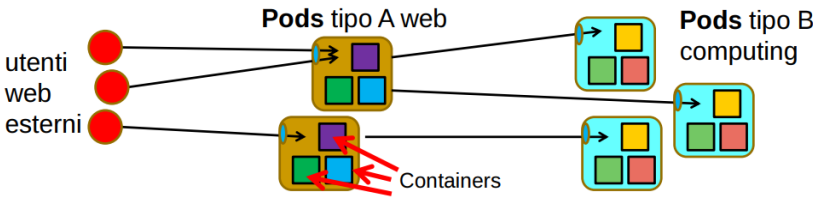


Figura 33: Struttura di un'applicazione Kubernetes

Cluster Kubernetes

Kubernetes organizza degli host (nodi del cluster) fisici o virtuali per poter eseguire e replicare i pods sui nodi, distribuendo così il carico. L'applicazione decide quanti pods usare e Kubernetes decide automaticamente su quali nodi allocare i pods.

Uno degli host svolge il ruolo di control plane (controllore del cluster), monitorando gli altri host e decidendo automaticamente in quali nodi eseguire i pods. Il control plane è il punto di accesso al cluster dall'esterno. Gli altri host (worker nodes) eseguono i pods delle applicazioni.

Gli utenti web esterni vedono solo il nodo controllore, ed inviano a lui le richieste. Il controller instrada le richieste ricevute ai pods dei worker nodes, che le soddisfano.

Se un pod va in crash, Kubernetes ne crea immediatamente una nuova istanza.

Scalabilità

Si può configurare Kubernetes affinché, all'aumentare del carico dell'applicazione, effettui 3 tipi di operazioni:

- Autoscaling verticale: Aumenta le risorse a disposizione di un pod.
- Autoscaling orizzontale: Aumenta il numero delle repliche dei pods attivi
- Cluster autoscaling: Aumenti il numero dei nodi che compongono il cluster redistribuendo i pods.

Ciò consente di scalare modulando il numero di pods/nodi in base alla quantità di carico del cluster. In questo modo è possibile mantenere l'applicazione in funzione anche se un host si guasta, in quanto ne rimangono altri a disposizione (Il sistema è decentralizzato).

La possibilità di realizzare cluster autoscaling dipende dal tipo di host usato. Per host fisici occorre un sistema di accensione degli host fisici guidabile via software (Es. Wake on LAN), mentre per gli host virtuali occorre un supporto, da parte del cloud pubblico o privato che ci fornisce le macchine virtuali, che ci dia la possibilità di creare e configurare nuove VM in automatico.

Cluster Kubernetes con nodi virtuali

È possibile creare un cluster kubernetes formato da host che siano macchine virtuali. Queste VM possono essere create manualmente su host fisici oppure possono essere create via software su cloud pubblici o privati.

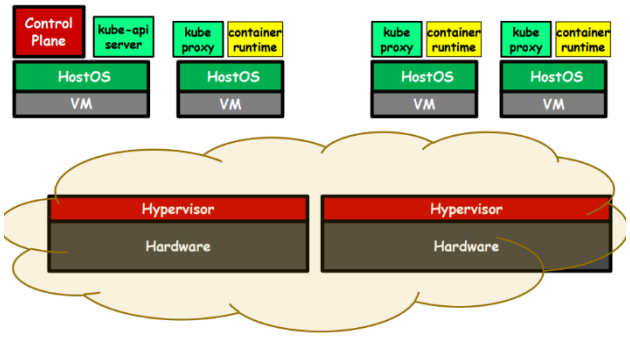


Figura 34: Nodi virtuali su cluster Kubernetes

Esistono cloud di diversi provider, i principali sono Google Cloud, Azure, AWS. È possibile costruire un cloud privato, partendo da macchine fisiche in una propria sala macchine, gestendo le proprie macchine con un insieme di software open source basati su Ubuntu (Openstack).

I sistemi cloud forniscono API mediante le quali possiamo ordinare, via software, la creazione e la configurazione di VM e di reti virtuali tra di esse. Le API sono, purtroppo, differenti in base

al provider. Esistono, tuttavia, alcuni software (come Terraform) i quali offrono un layer di astrazione per i cloud sottostanti, offrendo delle API utilizzabili su tutti i sistemi cloud per creare e configurare le VM.

In tal modo si può creare un cluster kubernetes formato da host virtuali ed applicare il cluster autoscaling configurando kubernetes affinché usi Terraform per creare nuove VM quando necessario.

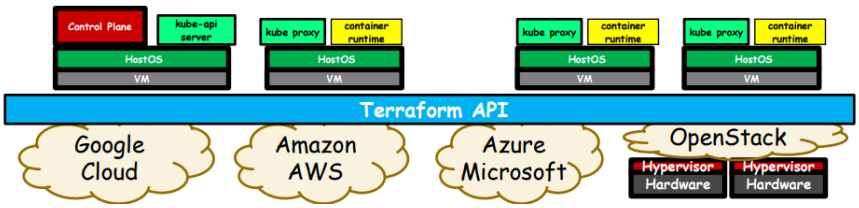


Figura 35: Layer Terraform per la traduzione delle API dei provider

Kubernetes as a Service

Alcuni sistemi cloud forniscono un servizio Kubernetes senza che sia necessario creare esplicitamente un cluster.

Il cluster viene creato ed installato dal provider su richiesta via software, a chi produce l'applicazione vengono fornite delle API attraverso cui chiedere di inizializzare il cluster Kubernetes ed effettuare il deployment dell'applicazione.

In tal modo si possono costruire applicazioni scalabili e resilienti in maniera più semplice, pagando al provider il costo del cluster.

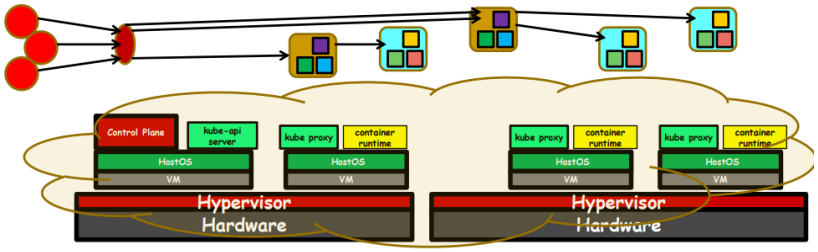


Figura 36: Cluster Kubernetes fornito dal provider

Container as a Service

Alcuni sistemi cloud offrono la possibilità di far eseguire un container, la cui immagine è fornita dallo sviluppatore dell'applicazione, in molteplici istanze su un cluster Kubernetes-as-a-service costruito dal provider.

Il container viene incapsulato in un pod predisposto dal provider ed eseguito nel cluster. In tal modo lo sviluppatore non si deve occupare di gestire il cluster ma solo di costruire un container da far eseguire. Questo approccio è fornito da: Google Container Engine (GKE), Amazon EC2 Container Service (ECS), Azure Container Service (ACS), IONOS Container Cluster.

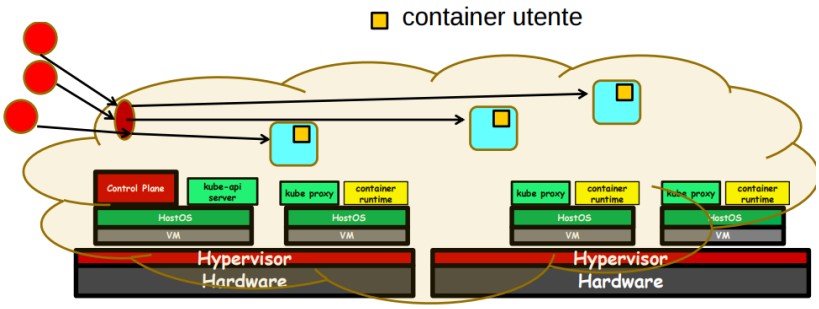


Figura 37: Kubernetes-as-a-service

Serverless o Function as a Service (FaaS)

Alcuni provider cloud astraggono ulteriormente, offrendo un servizio che consente di implementare delle funzioni che vengono eseguite, anche in più istanze contemporaneamente, senza definire né container né VM, e nemmeno un cluster Kubernetes (almeno apparentemente).

Quello che accade è che il provider cloud incorpora la funzione definita dall'utente in un container e la esegue su un cluster Kubernetes replicandolo man mano che aumenta il carico di richieste da fornire.

Per l'utente è facile creare queste funzioni ed il problema della creazione del contesto di esecuzione (container e pod) è risolto dal provider cloud.

Il provider offre alcune funzioni di libreria, lo sviluppatore implementa diverse altre funzioni, ciascuna delle quali può utilizzare, chiamandole, altre funzioni dello sviluppatore o di libreria, in modo da poter implementare anche funzionalità complesse.

Troviamo questo approccio nei principali provider cloud: AWS Lambda, Azure Functions e Cloud Functions (Google Cloud).

Interfaccia a utente caratteri (Incompleto)

Verrà saltata l'introduzione sul funzionamento del sistema operativo e del filesystem (già trattati nei capitoli precedenti), sulle librerie e le chiamate di sistema e sullo standard POSIX e C. Ritengo che le slide del prof. siano sufficientemente riassuntive e gli argomenti siano abbastanza banali da poter assumere che non serva prendere appunti su queste parti del programma.

Tuttavia, se qualcuno vorrà prendersi la briga di farlo non sarò io ad impedirlo.

Nozioni sull'uso del terminale

Il terminale era, una volta, un dispositivo formato da schermo e tastiera. Ad oggi i terminali sono integrati ne PC, il suo schermo viene emulato in una finestra dell'ambiente grafico ed è perciò detto terminale virtuale o emulatore di terminale.

L'interfaccia a caratteri di comando (CLI) permette di impartire ordini al SO sottoforma di sequenze di caratteri alfa-numerici.

L'interprete dei comandi (Shell) è un programma eseguibile che si occupa di un singolo terminale. Attende i caratteri digitati sulla tastiera, una volta ricevuti li interpreta ed esegue gli ordini ricevuti per poi ritornare in ascolto.

In questo corso si userà la shell Bash, una delle più usate, ma comunque ne esistono altre con comandi, regole e funzionalità più o meno diverse tra loro.

Prima di procedere apriamo una breve parentesi sulla struttura del filesystem di Linux.

Mentre su Windows le partizioni del disco sono viste come logicamente separate, sui sistemi UNIX (Linux, Mac OS) le partizioni sono viste come collegate tra loro. Esiste la partizione principale chiamata "/", questa partizione contiene, come le altre partizioni, files e directories.

Interpretazione dei comandi bash: Expansions

La shell legge ciascuna riga di comando e la reinterpreta, eseguendo alcune operazioni.

Alcune di queste servono a capire dove finisce un comando e dove inizia il successivo, altre ad individuare le parole ed identificare quelle riservate del linguaggio, identificare la presenza di costrutti linguistici (for, while, if...). Altre sostituiscono (espandono) alcune parti della riga con altre, interpretando caratteri speciali.

Le espansioni principali (in ordine di effettuazione) sono:

- History expansion `!123`
- Brace expansion `a{damn,czk,bubu}e`
- Tilde expansion `~/nomedirectory`
- Parameter and variable expansion `$1 $? $! ${var}`
- Arithmetic expansion `$(( ))`
- Command substitution `` ` `  $( ) ``
- Word splitting
- Pathname expansion
- Quote removal

Variabili e Variable expansion

La shell dei comandi permette l'uso di variabili (simboli dotati di un nome e di un valore modificabile). Il nome è una sequenza di caratteri anche numerici (il nome è case sensitive)

Il valore è una sequenza di caratteri, compresi numeri e simboli di punteggiatura.

Le variabili di una shell possono essere stabilite e modificate sia dal S.O. che dall'utente.
Alcune variabili (d'ambiente) vengono impostate dal sistema appena viene iniziata l'esecuzione

della shell (es. `PATH`). Altre invece possono essere create ex-novo dall'utente in qualunque momento dall'esecuzione della shell.

Le variabili possono essere usate quando si digitano degli ordini per la shell. La shell riconosce i nomi delle variabili e cambia il contenuto del comando sostituendo al nome della variabile il suo valore. La shell distingue il nome della variabile in un comando grazie ai caratteri ${}. Quindi affinché avvenga la sostituzione in un comando la forma da usare è ${NOME_VARIABILE} (Le parentesi graffe sono opzionali se il nome della variabile in un comando è preceduta e seguita da spazi.

Segue un esempio sull'uso delle variabili in una shell bash.

`VAR=CIAO` Definizione di una variabile di nome VAR e valore CIAO.

`echo ${VAR}` Stampa a video della variabile VAR, sul terminale si vedrà CIAO.

`echo ${VAR}X` Stampa a video della variabile VAR seguita dal carattere X, si vedrà CIAOX.

`echo $VAR` Stampa a video della variabile VAR, si vedrà CIAO.

`echo $VARX` La bash non trova nessuna variabile di nome VARX, non verrà stampato nulla. Non è definita una variabile con quel nome.

`echo $VAR X` La bash interpreta correttamente questo comando in quando VAR è preceduta e seguita da spazi, si vedrà CIAO X.

Bisogna sottolineare che non sono consentiti spazi quando si assegna il valore a una variabile prima e dopo il segno "=". (VAR = CIAO, VAR= CIAO, VAR =CIAO sono assegnamenti errati).

La variabile PATH

Esiste una variabile d'ambiente fondamentale, chiamata PATH. Essa viene impostata dal S.O. all'avvio della shell. È possibile modificare tale variabile.

Il contenuto della variabile PATH è una sequenza di percorsi assoluti (intervallati dal carattere ":" due punti) delle directory che contengono gli eseguibili. In un sistema Linux possiamo aspettarci che di default il contenuto della variabile PATH sia simile a:

```
/usr/local/sbin:/usr/local/bin:/usr/bin
```

Questa PATH definisce i percorsi che portano alle directory
/usr/local/sbin
/usr/local/bin
/usr/bin
Quando ordino alla shell di eseguire un file binario, scrivendone solo il nome (senza indicare il percorso per raggiungere quel file), allora la shell cerca quel file all'interno delle directories indicate nella variabile PATH, nell'ordine con cui i percorsi sono elencati.

Utenti e Gruppi

Nei sistemi Unix/Linux sono presenti le astrazioni di utente (user) e gruppo di utenti (group).

Un utente può corrispondere ad una persona umana, o essere una rappresentazione di un'entità usata per indicare chi esegue un servizio di sistema (ad esempio lo user mysql che esegue il servizio del database MySQL).

Un utente è associato ad una stringa (il suo username) e da un identificatore numerico (userID), entrambi devono essere univoci nel sistema.

Un gruppo è associato ad una stringa (il groupname) e da un identificatore numerico (groupID), anch'essi univoci. Ogni utente appartiene ad uno o più gruppi.

Ciascun file (e directory) del filesystem appartiene ad un utente (owner), che normalmente è l'utente che ha creato quel file, ed è associato ad un gruppo.

Un utente può tentare di accedere ad un file (in lettura o scrittura) o eseguire un binario anche se non ne è il proprietario.

Il proprietario di un file stabilisce quali utenti o gruppi di utenti possono accedere ad un file, configurandone i permessi di accesso.

Permessi di file e directory

Al momento della creazione di un file, il sistema operativo assegna l'utente che lo ha creato come proprietario. Il proprietario può cambiare a sua volta il proprietario del file usando il comando:

```
chown nuovoProprietario myFile
```

Ciascun file mantiene diversi diritti di lettura, scrittura ed esecuzione assegnati al proprietario, al gruppo del proprietario e a tutti gli altri.

Ad ogni tipo di permesso sono associati un valore numerico e una lettera:

- Lettura: valore 4, simbolo r.
    - Avere i permessi in lettura di un file vuol dire poterlo aprire ed estrarne il contenuto ma non modificarlo. Su una directory si ha accesso all'elenco dei file e delle directories al suo interno.
- Scrittura: valore 2, simbolo w.
    - Avere i permessi in scrittura su un file vuol dire poterlo modificare (si potrebbe assegnare solo il permesso di scrittura, consentendo la modifica senza poter leggere il file, sconsigliato). Su una directory si ha la possibilità di creare, eliminare e

modificare il nome del contenuto (ciò non implica l'accesso alla directory, non ci posso "entrare").

- Esecuzione: valore 1, simbolo x.
    - Avere i permessi di esecuzione su un file vuol dire poterlo eseguire. Su una directory significa potervi accedere e vedere le proprietà dei file all'interno.

I comandi per cambiare proprietario, gruppo e permessi sono: `chown, chgrp, chmod.`

Se un utente (effective userID) vuole accedere ad un file, si applicano i seguenti criteri di utilizzo basasti sui permessi di quel file:

Se l'effective userID è l'owner del file, si applicano le User permissions.

Se l'effective groupID corrisponde al groupID del file, si applicano le Group permissions.

In tutti gli altri casi si applicano i permessi generali (Others).

user			group			others		
R	W	X	R	W	X	R	W	X
4	2	1	4	2	1	4	2	1

Figura 38: Permessi di file e directory

Solo il proprietario del file può cambiare proprietario, gruppo e permessi del proprio file. È possibile fare ciò in maniera sintetica usando un formato numerico:

Si sommano i valori associati ai permessi di Read (4), Write (2) ed Exec (1) per ottenere un unico valore che indicherà quali sono i permessi per l'owner, il group e gli altri utenti. Ad esempio, supponiamo esista un file con nome "miofile.txt" nella directory corrente, e che si vogliano assegnare tutti i permessi all'owner, i permessi in lettura e scrittura per il group, i permessi di

sola lettura per gli altri. Per l'owner avrò il valore 7 (4+2+1), per il group 6 (4+2) per gli altri 4. Il comando completo sarà:

```
chmod 764 ./miofile.txt
```

Visualizzazione permessi di file e directory

Per visualizzare tutte le informazioni (e quindi anche i permessi) di tutti i file in una directory si può utilizzare il comando `ls` con i flag `-l` (long listing) e `-a` (all files).

```
ls -la
```

L'output dovrebbe essere simile a questo:

```
total 48K
drwxr-xr-x 1 aleemont aleemont  262 Jan 23 16:24 .
drwx------ 1 aleemont aleemont  622 Jan 23 16:20 ..
-rw------- 1 aleemont aleemont  782 Jan 23 16:21 .bash_history
-rw-r--r-- 1 aleemont aleemont  141 Jan 23 16:21 .bashrc
drwx------ 1 aleemont aleemont 1.8K Jan 23 16:22 .config
drwxr-xr-x 1 aleemont aleemont   20 Jan 23 16:22 Desktop
drwxr-xr-x 1 aleemont aleemont   34 Jan 23 16:22 Documents
drwxr-xr-x 1 aleemont aleemont 1.9K Jan 23 16:22 Downloads
-rw-r--r-- 1 root     root        0 Jan 23 16:24 foo.txt
-rwxr-xr-- 1 root     root        0 Jan 23 16:24 main.c
drwxr-xr-x 1 aleemont aleemont   88 Jan 23 16:22 Pictures
drwxr-xr-x 1 aleemont aleemont    0 Jan 23 16:22 Public
drwxr-xr-x 1 aleemont aleemont    0 Jan 23 16:22 Templates
drwxr-xr-x 1 aleemont aleemont    0 Jan 23 16:22 Videos
```

Permessi speciali

Eseguendo il comando ls –lah /usr/bin/passwd vedremo i permessi del file "passwd" nella directory /usr/bin. Si tratta dell'eseguibile che consente di modificare la propria password.

```
ls -lah /usr/bin/passwd
```

Output:

```
-rwsr-xr-x 1 root root 51K Jan  5 11:33 /usr/bin/passwd
```

Si noti la “s” nella sezione delle user permissions, sostituisce la x di Execution. Dice che quando quell’eseguibile viene eseguito da un utente che ha i permessi per farlo, il processo creato dall’eseguibile esegue con i permessi di chi lo ha lanciato, ma anche con i permessi del proprietario dell’eseguibile (root, nell’esempio).

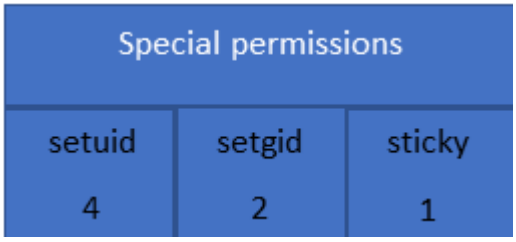


Figura 39: Permessi speciali

La “s” rappresenta:

- “setuid” nelle user permissions, si setta col comando `chmod 4***`. In esecuzione, il processo associato ottiene anche i diritti dell’owner (tipicamente root).
- “setgid” nelle group permissions, si setta col comando `chmod 2***`. Il comportamento è simile al setuid, ma si applica a tutto il group.

Sticky bit (rappresentato da t), si setta col comando `chmod 1***`. Si utilizza per le directories (i file vengono ignorati). I file all’interno della directory possono essere rinominati o cancellati solo dal proprietario del file, della directory o da root (ovviamente).

Subshell

Una subshell è una shell (detta figlia) create da un’altra shell (padre).

Una subshell viene creata quando:

- Si eseguono dei comandi raggruppati.
- Si esegue uno script.
- Si esegue un processo in background.

Ogni shell ha una propria working directory (la directory corrente) e delle proprie variabili.

Ogni subshell eredita dalla shell padre una copia delle variabili d'ambiente ma non le variabili locali.

Quando una shell deve eseguire uno script effettua queste operazioni:

- Legge la prima riga dello script in cui è indicato quale interprete deve eseguire i comandi (ad esempio #!/bin/bash, #!/bin/zsh).
- Crea una subshell.
- Il nome dello script viene passato come argomento (flag –c) alla nuova subshell.
- La subshell esegue lo script.
- Alla fine dell'esecuzione la subshell termina e restituisce il controllo alla shell padre, ritornando un valore intero che indica il risultato delle operazioni.

Variabili

Ogni shell supporta due tipi di variabili:

- Locali: Non ereditate dalle subshell create da una shell padre. Vengono utilizzate per calcoli e computazioni all'interno di script.

- D’ambiente: Ereditate dalle subshell attraverso la creazione di una copia. La modifica di una variabile d’ambiente in una subshell non comporta la modifica delle variabili d’ambiente della shell padre. Vengono solitamente utilizzate per la comunicazione fra parent e child shell. Alcuni esempi sono: `$HOME`,`$PATH`,`$USER`,`%SHELL`,`$TERM`

Il comando per visualizzare le variabili d’ambiente è `env`.

Quando viene dichiarata una nuova variabile con la sintassi descritta prima si crea una variabile locale.

Per creare una variabile d’ambiente si usa il comando export.

Ad esempio, se si volesse creare una variabile d’ambiente di nome `ENV_VAR` e valore `100` si dovrebbe usare il comando:

```
export ENV_VAR=100
```

Si può rendere una variabile d’ambiente anche una variabile già dichiarata:

```
ENV_VAR=100
#Operazioni varie
export ENV_VAR
```

Esecuzione di script nella stessa shell

In alcuni casi potrebbe tornare utile eseguire uno script senza che esso venga eseguito in una shell figlia. Per fare ciò esiste il comando `source`.

```
source nomescript
```

Alternativa più breve: `.` `Nomescript`

Eseguire uno script nella shell stessa è utile, ad esempio, per modificare tramite uno script le variabili d'ambiente.

È importante sottolineare che la prima riga dello script (che contiene il percorso assoluto all'interprete di quello script) viene ignorata se si lancia uno script col comando source. Quindi bisogna essere assolutamente sicuri che la shell in cui lanciamo lo script chiami l'interprete corretto per quello script. Uno script scritto per una shell diversa da bash (come zsh, fish o sh) non può, in generale, essere interpretato correttamente da bash.

Visibilità delle variabili

È possibile definire delle variabili d'ambiente in una subshell di un eseguibile che si sta per fare eseguire, senza che esse vengano ereditate da quelli successivi, semplicemente scrivendo la dichiarazione prima del nome del comando.

In pratica:

```
var="valore" comando      #comando vede e può usare var
echo ${var}               #echo non vede var, non stampa nulla
```

Bisogna ricordare che se esiste una variabile d'ambiente con lo stesso nome di quella che vogliamo sia vista dal comando, questo vedrà solo la nuova variabile e non quella globale. La variabile d'ambiente ritorna visibile dopo l'esecuzione.

Esempio:

```
export var="var d'ambiente"

var="nuova var comando              #comando vede "nuova var"

echo ${var}                         #echo vede "var d'ambiente"
```

Variabili vuote o non esistenti

Bisogna distinguere tra una variabile vuota ed una che non esiste.

Una variabile è vuota se è stata dichiarata assegnandole una stringa vuota come valore (`var=""` è vuota).

Una variabile non esiste se non è mai stata dichiarata.

È possibile eliminare una variabile usando il comando `unset`

```
unset nomevariabile
```

Dopo averla eliminata, quella variabile non esiste.

Riferimenti indiretti a variabili

Supponiamo esista una variabile "varA" con un valore qualsiasi ed una variabile il cui valore è il nome della prima variabile. Voglio usare il valore della prima variabile sfruttando solo il nome della seconda variabile (Che avrà quindi come valore "varA"). Si dice che la seconda variabile è un riferimento indiretto a varA.

Si può accedere al valore di varA usando l'operatore ${!nomeRifIndiretto}

Ad esempio:

```
varA=VALUE
varB=varA
echo ${!varB}
```

L'output sarà: `VALUE`

History expansion

La funzionalità history memorizza i comandi lanciati dalla shell e permette di visualizzarli e rilanciarli. La storia dei comandi viene mantenuta (in un file chiamato .bash_history, di solito situato nella home directory dell'utente) anche dopo la chiusura della shell.

Il comando built-in history visualizza un elenco numerato (dal comando lanciato prima a quello più recente) dei comandi precedentemente lanciati dalla shell.

L'output del comando history è simile a questo:

```
43  mkdir build                 #più                          vecchio
44                                      cd                    build/
45                                       cmake                     ..
46  make
47  make                        #più recente
```

Lanciando il comando `!NUMERO` eseguo il comando che nella history è in posizione "NUMERO". Ad esempio, lanciando `!43` il comando eseguito sarà `mkdir build`.

Comando set

Il comando built-in set svolge diversi compiti:

- Se lanciato senza nessun parametro visualizza tutte le variabili della shell, sia quelle locali che quelle d'ambiente. Inoltra visualizza le funzioni implementate nella shell.
- Se lanciato con dei parametri serve per settare o resettare un'opzione di comportamento della shell in cui viene lanciato.

Esempio:

`set +o history` disabilita la memorizzazione di ulteriori comandi nel file .bash_history. I comandi lanciati prima della disabilitazione restano salvati nel file di history.

`set -o history` abilita la memorizzazione dei comandi, appendendoli al file di history.

`set -a` rende (immediatamente) tutte le variabili create o modificate nella shell variabili d'ambiente.

`set +a` rende tutte le variabili create o modificate nella shell variabili locali (è il comportamento di default).

Avvio della shell bash

La shell bash si comporta in maniera diversa a seconda degli argomenti passati nel momento in cui ne viene lanciata l'esecuzione. I modi di esecuzione della bash sono 3:

- Shell non interattiva
    - Shell figlia che esegue script.
    - Lanciata con argomento –c /path/to/script.
    - La shell non interattiva non esegue i comandi contenuti in nessuno dei file (di sistema o dell'utente) che customizzano il comportamento della shell interattiva.
- Shell interattiva:
    - Shell lanciata senza argomenti (è quella di default).
    - Quando lanciata esegue (se esiste) il solo file ".bashrc" collocato nella home directory dell'utente.
- Shell interattiva di login:
    - Shell interattiva che richiede username e password prima di poter essere usata. Lanciata con argomento –l (--login).
    - Quando lanciata esegue (se esistono) i seguenti files:

1. /etc/profile.
2. Il primo trovato tra ".bash_profile", ".bash_login" e ".profile" collocati nella home directory dell'utente.
3. ".bashrc" collocato nella home directory dell'utente.

- Nel momento in cui termina, la shell di login esegue il file .bash_logout collocato nella home dell'utente (se il file esiste).

Parametri a riga di comando

I parametri (o argomenti) di un programma sono un insieme ordinato di caratteri, separati da spazi bianchi, che vengono passati al programma eseguito in una shell nel momento iniziale in cui il programma viene lanciato.

Quando da una shell si vuole chiamare un comando con degli argomenti è quindi necessario scriverli accanto al nome del comando prima di lanciarlo.

Esempio:

```
chmod u+x /home/aleemont/miofile.exe
```

Il primo simbolo della riga di comando (`chmod`) è il nome dell'eseguibile (potrebbe essere interessante sapere che viene visto come l'argomento di indice 0). Il secondo simbolo (`u+x`) è l'argomento di indice 1, il terzo (`/home/aleemont/miofile.exe`) è l'argomento di indice 2. ,Nell'esempio il numero di argomenti passati è 2.

Una volta cominciata l'esecuzione del programma questo ha modo di conoscere il numero degli argomenti passati e di ottenere tali argomenti utilizzandoli.

Separatore di comandi e delimitatore di argomenti

In bash è possibile lanciare diversi comandi in successione su un'unica riga. Al termine di un comando verrà eseguito il successivo. Per concatenare dei comandi in un'unica riga si usa il separatore ";". Esso stabilisce dove finisce un comando e dove inizia il successivo.

Se si avesse necessità di stampare stringhe che contengono il separatore dei comandi sarebbe sufficiente racchiudere la stringa tra doppi apici (echo "cp ; sudo" stamperà cp ; sudo a schermo, mentre echo cp; sudo stamperà cp ed eseguirà il comando sudo). I doppi apici sono i delimitatori di argomenti, tutto ciò che è racchiuso all'interno dei doppi apici è visto dalla shell come un unico argomento.

Quoting di singoli caratteri

Prendendo ancora in considerazione il comando `echo "cp ; sudo"` è bene sapere che è possibile disabilitare l'interpretazione di singoli caratteri speciali utilizzando il carattere di escape "\". Quindi il comando `echo cp \; sudo` stamperà correttamente a video la stringa "cp ; sudo".

Brace Expansion – generazione di stringhe

Quando viene passata una riga di comando da eseguire alla bash, essa tenta di capire se nella riga sono presenti coppie di parentesi graffe che rappresentano un ordine di generare stringhe secondo alcune regole.

- Un ordine di brace expansion è una stringa di testo
    - Racchiusa tra separatori (spazi, tab, new lines).
    - Al cui interno non compaiano una coppia di graffe non precedute da un $.

  - Al cui interno non ci siano spazi o tab.

- Questa stringa che ordina una brace expansion è formata da tre parti, di cui la prima (preambolo) e l'ultima(postscritto) possono non essere presenti.
- La parte di mezzo è racchiusa all'interno di parentesi graffe.
- Dentro le graffe sono presenti una o più stringhe separate da virgole.
- Ciascuna stringa rappresenta una possibile scelta di stringhe che possono essere aggiunte al preambolo e seguite dal postscritto per formare delle nuove stringhe di testo.

Ad esempio, se la riga di comando è fatta così:

```
echo p{er,ier,olp,al}o
```

Il comando viene espanso in questo modo:

```
echo pero piero polpo palo
```
stampa `pero piero polpo palo`

Il preambolo e il postscritto sono opzionali (`echo {c,i,a,o}` va bene e stampa "`c i a o`").

Non possono esserci spazi bianchi o tab nella brace expansion. Se sono presenti spazi le parentesi non vengono espanse e si ottiene in output la stringa passata ad echo senza alcuna formattazione.

```
echo a{bb,cc, dd}ee
```
stampa `a{bb,cc, dd}ee`

Per generare stringhe con spazi bianchi è necessario proteggere i singoli spazi con un backslash

```
echo a{bb,cc,d\ \ d}ee
```
stampa `abbe accee ad  dee`

Non si può proteggere tutto l'ordine di brace expansion con doppi apici, in quanto essi disabilitano l'interpretazione, tuttavia, si possono proteggere le singole stringhe della brace expansion.

`echo "a{bb,cc,d d}ee"` stampa `a{bb,cc,d d}ee`

Mentre

`echo a{bb,cc,"d d"}ee` stampa `abbee accee ad dee`

Annidamento della brace expansion

È possibile inserire degli ordini di brace expansion all'interno di altri.

Ad esempio, il comando

```
echo /usr/{ucb/{ex,edit},lib({bin,sbin}}
```

Stampa `/usr/ucb/ex /usr/ucb/edit /usr/lib/bin /usr/lib/sbin`

Si possono inserire variabili nella brace expansion

```
A=bin
B=log
C=boot
echo ${A}{${B}${C},${C},${A}${B}}a
```

stampa binlogboota binboota binbinloga

Brace expansion con Sequence expression

Esiste una forma di brace expansion che permette la generazione di stringhe elencando un intervallo di caratteri. L'intervallo si indica con {INIZIO..FINE} (i due estremi collegati da 2 punti).

```
echo a{b..g}h
```

Stampa `abh ach adh aeh afh agh`

Tilde expansion

La tilde expansion riguarda 5 casi essenziali:

1. Un carattere tilde isolato.
2. Un carattere tilde seguito da slash non quotato.
3. Un carattere tilde seguito da slash non quotato seguito da altri caratteri.

- In questi casi la tilde viene sostituita dal percorso assoluto della home directory dell'utente che sta eseguendo la riga di comando (effective user).
    - Ad esempio, supponendo che l'effective user si chiami "aleemont":
        - `cd ~` cambia la directory corrente in /home/aleemont
        - `echo ~/` visualizza /home/aleemont
        - `echo ~/dir` visualizza /home/aleemont/dir

4. Una parola che inizia con la tilde seguita da un nome utente.
5. Una parola che inizia con la tilde seguita da un nome utente, seguita da slash non quotato a cui possono seguire altri caratteri.

- In questi casi la tilde più nome utente viene sostituita dal percorso assoluto della home directory dell'utente specificato.
    - Ad esempio, supponendo che esista l'utente "peter":
        - `echo ~root` visualizza /home/peter
        - `echo ~peter/dir` visualizza /home/peter/dir
    - Se lo username specificato non esiste l'espansione non avviene

- `echo ~username_che_non_esiste`

  visualizza ~username_che_non_esiste

Wildcards

Pathname substiution

I metacaratteri '*' e '?' vengono interpretati dalla shell, che cerca di sostituirli con una sequenza di caratteri per ottenere i nomi di files nel filesystem.

- '*' può essere sostituito da una qualunque sequenza di caratteri, anche vuota.
- '?' può essere sostituito da un singolo carattere qualsiasi (non vuoto).
- [char1char2char3...] può essere sostiuito da un solo carattere tra quelli specificati.

Alcuni esempi:

- `ls /home/aleemont/*.c` visualizza il nome di tutti i file nella directory /home/aleemont che terminano con ".c".
  (Ad esempio, main.c, test.c, .c, a.c, min.c, mpin,c).
- `ls /home/aleemont/main*` visualizza il nome di tutti i file nella directory /home/aleemont che iniziano con "main".
  (Ad esempio, main.c, main.o, main.exe).
- `ls /home/aleemont/m?in.c` visualizza il nome di tutti i file nella directory /home/aleemont che hanno iniziano con "m",
  seguita da un carattere (qualsiasi, non vuoto) e

terminano con “in.c”.

(Ad esempio, main.c, mpin.c).

Per quanto riguarda l’espansione delle parentesi quadre “[ ]” invece:

- `[abk]` Viene sostituito da un carattere tra a, b, k.
- `[1-7]` Viene sostituito da un carattere tra 1,2,3,4,5,6 o 7.
- `[c-f]` Viene sostituito da un carattere tra c, d o f.
- `[[:digit:]]` Viene sostituito da una cifra.
- `[[:upper:]]` Viene sostituito da un carattere maiuscolo.
- `[[:lower:]]` Viene sostituito da un carattere minuscolo.

Notare che le parentesi quadre selezionano uno solo tra i caratteri elencati al loro interno.

Altri esempi: Supponiamo che in una directory ci siano i file aB, a1B, a2B, akB, akmB, akmtB.

- `ls a[[:digit:]]B` Visualizza a1B a2B.
- `ls a[[:lower:]][[:lower:]][[:lower:]]B` Visualizza akmtB.
- `ls a[[:lower:]][[:lower]]B` (N.B: [[]][[]]) Visualizza akmB.
- `ls a[[:lower:][:lower:]]B` (N.B: [[] []]) Visualizza akB.

Comandi ed eseguibili utili

pwd	Mostra la working directory corrente.
cd /path/to/dir	Cambia la working directory

mkdir /path/to/newdir	Crea una nuova directory nel path specificato
rmdir /path/to/emptydir	Elimina la directory indicata se è vuota
ls –lah /path/to/dirOrFile	Stampa informazioni sui file nel percorso
rm /path/to/file	Elimina il file specificato
echo string(s)	Visualizza su stdout le stringhe specificate
cat /path/to/file	Visualizza su stdout il contenuto del file
env	Visualizza le variabili ed il loro valore
which command_name	Visualizza il percorso all'eseguibile (se in PATH)
mv /old/path/to/file /new/path/to/file	Sposta il file specificato in una nuova posizione
ps aux	Stampa informazioni sui processi in esecuzione
du /path/to/dir	Visualizza l'occupazione del disco
kill –9 pid	Elimina il processo avente il pid specificato
killall process_name	Elimina tutti i processi process_name
bg	Ripristina un job fermato e messo in background
fg	Riporta un job in foreground
df	Mostra lo spazio libero nei filesystsems montati
touch /path/to/file	Crea il file specificato se non esiste, altrimenti ne aggiorna la data.

more /path/to/file	Mostra il file specificato un poco alla volta
head /path/to/file	Mostra le prime 10 righe del file specificato
tail /path/to/file	Mostra le ultime 10 righe del file specificato
man command_name	È il manuale, mostra le informazioni sul comando specificato
find	Cerca dei files nel filesystem
grep	Cerca tra le righe di file quelle che contengono il pattern specificato
read variable_name	Legge input da stdin e lo inserisce nella variabile specificata
wc	Conta il numero di parole o di caratteri in un file
true	Restituisce exit status 0
false	Restituisce exit status 1

Parametri a riga di comando passati al programma

Esistono variabili d'ambiente che contengono gli argomenti passati ad uno script.

- `$#` Il numero di argomenti passati allo script.
- `$0` Nome del processo in esecuzione.
- `$1, $2, $3...` Argomento 1, 2, 3, … rispettivamente.
- `$*` Tutti gli argomenti concatenati e separati da spazio.
- `$@` Come `$*` ma se quotato gli argomenti sono quotati separatamente.

```
$* = $@ = $1 $2 $3 … $n.
"$*" = "$1 $2 $3 … $n".
"$@" = "$1" "$2" "$3" … "$n".
```

`$@` risulta particolarmente utile quando uno script deve eseguire un comando passandogli tutti i parametri ricevuti.

I parametri non possono essere modificati. Il programma vede gli argomenti dopo l'eventuale sostituzione dei caratteri "*" e "?".

Ad esempio, supponiamo di avere uno script "esempio.sh" con il seguente contenuto:

```
#!/bin/bash
Echo "Sono stati passati $# argomenti"
Echo "Gli argomenti sono: $*"
```

Lanciando lo script con:

```
./esempio.sh arg1 arg2
```

L’output sarà:

```
Sono stati passati 2 argomenti
Gli argomenti sono:  arg1 arg2
```

Supponiamo ora che nella directory corrente ci esistano i file x.c y.c z.c:

```
./esempio.sh *.c
```

Stamperà in output:

```
Sono stati passati 3 argomenti
Gli argomenti sono:  x.c y.c z.c
```

Valutazione Aritmetica di espressioni tra interi

La bash permette di valutare una stringa come se fosse un’espressione matematica costituita da operazioni aritmetiche tra soli numeri interi.

L’operatore `(())` racchiude tutta una riga di comando (non si può racchiudere una parte di una riga con questo operatore) che deve essere un’espressione (più eventuale assegnamento).

Viene eseguita la riga di comando racchiusa tra le 2 coppie di parentesi tonde valutando aritmeticamente gli operandi (Ad esempio, `((NUM=3+2))` assegna 5 alla variabile NUM).

L’operatore `$(())` racchiude una parte di una riga di comando (non si può racchiudere un’intera riga con questo operatore), che deve essere un’espressione (più eventuale assegnamento).

La bash:

Valuta aritmeticamente l’espressione racchiusa da `$(())`.

Nella riga di comando sostituisce il risultato calcolato all’espressione `$((…))`.

Esegue la riga di comando modificata.

Ad esempio eseguendo il comando:

```
echo ciao$((3+2))
```

La bash valuta prima l'espressione `$((3+2))` calcolandone il risultato (5). Nella riga di comando viene sostituita l'espressione: `echo ciao5`. Infine, viene eseguito il comando, il cui output sarà "ciao5".

Le valutazioni aritmetiche possono contenere:

- Operatori aritmetici (+, -, *, /, %).
- Assegnamenti
- Parentesi tonde per accorpare operazioni e modificare precedenze.

Se usate in una valutazione aritmetica, le parentesi tonde non creano una nuova bash.

Exit Status

Valore restituito da un programma al chiamante

Ogni programma o comando restituisce un valore numerico compreso tra 0 e 255 per indicare la presenza di errori di esecuzione o meno. Un risultato 0 indica che non ci sono stati errori, un risultato diverso da zero indica, chiaramente, l'opposto.

Tale risultato non viene visualizzato sullo schermo ma viene passato alla shell che ha chiamato l'esecuzione del programma. In tal modo il chiamante può controllare in modo automatizzato la funzionalità dei programmi.

Per restituire un valore in uno script bash si utilizza il comando `exit`

(ad es. `exit 9` terminerà il programma restituendo 9 come risultato).

Per catturare il risultato di un programma si usa la variabile d'ambiente "`$?`", la quale viene modificata ogni volta che un programma o un comando termina, inserendo al suo interno il risultato del programma appena terminato.

Exit status di espressioni valutate aritmeticamente

La valutazione aritmetica effettuata tramite gli operatori `$(( ))` o `(( ))` restituisce un valore di exit status che sarà:

- Diverso da zero se è accaduto un errore. Il valore restituito indica il tipo di errore.
    - `(( ls 5 ))` `$?==1` (Non valutabile aritmeticamente).
- Uguale a zero se la valutazione aritmetica restituisce true.
    - `(( 5>=2 ))` `$?==0`.
- Diverso da zero se la valutazione aritmetica restituisce false.
    - `(( 5<=2 ))` `$?==1`.
- Uguale a zero se la valutazione aritmetica fornisce un risultato intero diverso da zero.
    - `(( 5 ))` `$?==0`.
    - `(( VAR=5+3 ))` `$?==0`.
- Diverso da zero se la valutazione aritmetica fornisce un risultato intero uguale a zero.
    - `(( 0 ))` `$?==1`.
    - `(( VAR=6-2*3 ))` `$?==1`.

- Se l'assegnamento è interno all'espressione (non eseguito per ultimo), il risultato dell'espressione è quello del confronto eseguito per ultimo.
    - `(( (VAR=6-2*3) != 1 ))` `$VAR==0, $?==0` (la condizione è vera)

Prova d’esame 24/01/2023

Tempo limite: 50 minuti

1. Descrivere le differenze tra una libreria statica ed una libreria caricata dinamicamente.

   Una libreria statica viene accorpata fisicamente al codice sorgente dal linker a link-time. Le funzioni in essa definite saranno disponibili da subito durante il run-time.

   Per le librerie caricate dinamicamente invece il linker accorpa la porzione di codice macchina della funzione chiamata, rendendola disponibile solo al momento della chiamata.

2. Perché le system calls vengono messe in esecuzione mediante un interrupt (istruzione int in assembly 8088) invece che eseguirle, come si farebbe con una normale funzione implementata in assembly, chiamandola con l'istruzione call?

   Le syscall sono API usate dai processi per richiedere servizi al sistema operativo. Il meccanismo degli interrupt viene utilizzato in quanto esso risulta essere molto efficiente, inoltre per ragioni di sicurezza, il trigger di un Interrupt permette di isolare il contesto di esecuzione di una syscall dal resto del sistema; infatti, le syscall hanno spesso bisogno di un livello di privilegio molto alto per poter eseguire determinate operazioni ed è fondamentale garantire che le altre parti del processo utente non abbiano accesso a zone sensibili del sistema.

3. Definire precisamente cos'è una eccezione di tipo fault ed indicarne un esempio

   Un’eccezione di tipo FAULT è un errore che interrompe il normale flusso di esecuzione. Quando viene sollevata un’eccezione di tipo FAULT l’esecuzione viene interrotta e ripartirà da capo una volta gestita l’eccezione. Un esempio di eccezione di tipo FAULT è il “page-fault”, che si verifica quando la pagina di memoria contenente l’indirizzo che si sta tentando di accedere non è caricata in memoria centrale ma sullo swap del disco, sarà quindi necessario caricare la pagina in memoria e ricominciare l’esecuzione del programma che si stava eseguendo.

4. Nella virtualizzazione del livello di sistema operativo, quale kernel esegue le system call invocate all'interno di un container ?

   Nella virtualizzazione del livello di sistema operativo il container esegue syscalls direttamente al Kernel di sistema operativo host (Il sistema su cui si esegue il container), a differenza della virtualizzazione a livello hardware dove il sistema operativo guest effettua chiamate a sistema al Kernel del sistema operativo guest (che vede un layer di astrazione aggiuntivo, che consiste in hardware “virtuale” messo a disposizione dall’hypervisor), che verranno tradotte dall’hypervisor in chiamate a sistema comprensibili al Kernel host.

5. Nei moderni sistemi operativi che usano sia la segmentazione sia la paginazione sfruttando il supporto di un processore moderno (come quelli a partire dall'architettura IA-32), quando specifico l'indirizzo di una istruzione mediante la coppia <RegistroDiSegmento:Offset>, cosa contiene esattamente il RegistroDiSegmento ? L'indirizzo fisico di inizio del segmento? L'indirizzo virtuale di inizio del segmento? Altro (in questo ultimo caso specificate voi esattamente che cosa è "altro") ?

   Il RegistroDiSegmento conterrà l'indirizzo virtuale di inizio del segmento, in quanto l'indirizzo fisico verrà calcolato dalla MMU.

6. Nel contesto del filesystem, descrivere concisamente come è organizzata l'allocazione ai diversi file dei blocchi del disco secondo il metodo di allocazione indicizzata combinata. Indicarne eventuali vantaggi e svantaggi.

   Lo schema di allocazione combinata unisce lo schema ad indicizzazione concatenata a quello ad indicizzazione multilivello.

   Il primo consiste nel creare un index block con riferimenti diretti ai blocchi del file, l'index block occupa un solo blocco, e se esso non fosse sufficiente allora l'ultimo puntatore dell'index block punterà ad un altro index block.

   Nell'indicizzazione multilivello invece l'index block di primo livello contiene riferimenti ad index blocks di secondo livello e così via. Nello schema combinato si usano entrambe le tecniche, impiegando in parte riferimenti diretti (blocchi diretti) e in parte riferimenti ad altri index block. Negli inode dei sistemi Unix per file di piccole dimensioni si usano solamente blocchi diretti, al crescere delle dimensioni del file aumenteranno anche le indicizzazioni multilivello coinvolte.

7. Nel contesto della programmazione concorrente in ambito POSIX, quale è il significato dell'errore EINTR che viene inserito nella variabile errno da alcune funzioni di libreria standard per linguaggio C ? Motivare la risposta.

   L'errore EINTR viene inserito nella variabile errno quando una routine di gestione di una syscall viene interrotta da un segnale. Ciò accade se un segnale viene inviato al processore durante l'esecuzione di una routine di gestione della syscall, interrompendone il flusso di istruzioni. Ad esempio se durante una read() il processo (in attesa di ricevere la disponibilità dei dati nel file descriptor) riceve un segnale, allora la read() termina resistuendo –1 e settando errno a EINTR.

8. Supponiamo che V1 e V2 siano due variabili intere. Considerare le tre seguenti porzioni di codice 1, 2 e 3. Come vedete vengono tutte eseguite detenendo la mutua esclusione sulla variabile globale mutex. Ipotizziamo che vi siano altri thread che modificano spesso quelle variabili V1 e V2, previa acquisizione e successivo rilascio della mutua esclusione su quella stessa mutex. Ipotizziamo infine che ci sia un altro thread che molto spesso lancia la funzione pthread_cond_broadcast(&cond) abilitando i processi a proseguire uscendo dalla wait sulla condition variable cond. Nei tre casi di codice proposto, quando il thread raggiunge la label A, le variabili V1 e V2 hanno entrambe valore 1? Motivare la risposta.

codice 1.

```
pthread_mutex_lock(&mutex);

while( V1 != 1 || V2 != 1 ) pthread_cond_wait(&cond,
&mutex);

A: printf(" V1= %i V2= %i \n", V1, V2);

pthread_mutex_unlock(&mutex);
```

codice 2.

```
pthread_mutex_lock(&mutex);

while( V1 != 1 && V2 != 1 ) pthread_cond_wait(&cond,
&mutex);

A: printf(" V1= %i V2= %i \n", V1, V2);

pthread_mutex_unlock(&mutex);
```

codice 3.

```
pthread_mutex_lock(&mutex);

while( V1 != 1 )

pthread_cond_wait(&cond, &mutex);

while( V2 != 1 ) pthread_cond_wait(&cond, &mutex);

A: printf(" V1= %i V2= %i \n", V1, V2);

pthread_mutex_unlock(&mutex);
```

Risposta:

Codice 1: Non è detto, se V1==1 allora l'OR è vero, quindi procede a prescindere dal valore di V2, inoltre il valore delle variabili potrebbe essere modificato durante la wait da altri thread.

Codice 2: In questo caso si ha la certezza, la stampa verrà eseguita se e solo se V1 ==1 e V2==2.

Codice 3: Non è detto, infatti è possibile che anche se V1 == 1, durante la wait essa potrebbe essere modificata, idem per V2.

Crediti

Questo documento è frutto della rielaborazione delle slides del prof. Vittorio Ghini per il corso di Sistemi Operativi A.A. 2022/23, e del documento “Sistemi Operativi, Modulo 6: Gestione della memoria”, A.A. 2009-10 (prof. Renzo Davoli, prof. Alberto Montressor).

Si ringraziano i prof. V. Ghini, R. Davoli e A. Montressor per aver messo a disposizione il materiale.

Nessuna violazione dei diritti d’autore è intenzionale, tutti i diritti sono riservati ai rispettivi proprietari e/o autori del materiale.