La Bibbia di Sistemi operativi

Mario Petruccelli
Università degli studi di Milano

A.A. 2018/2019

Sommario

1 Virtualization

1.1 Introduzione

Processi Un processo, informalmente, è un programma in esecuzione. Un programma a sua volta, è una sequenza finita di istruzioni scritte in un linguaggio comprensibile all'esecutore (CPU). L'esecuzione di un programma da parte del processore è:

- **Fetch** Prelievo istruzione dalla memoria.
- **Decode** Decodifica dell'istruzione.
- **Execute** Esecuzione dell'istruzione.

1.1.1 Virtualizzazione

La virtualizzazione consiste nel prendere una risorsa fisica e trasformarla in una più generale, potente e facile da adoperare forma virtuale di se stessa.

Virtualizzazione della CPU L'illusione consiste nel far credere che il sistema abbia un elevato numero di cpu virtuali. Avere più CPU permeterrebbe a più programmi di essere eseguiti in **parallelo** nonostante il processore fisico effettivo sia uno solo. Se due processi vogliono essere eseguiti entrambi ad un certo tempo, oppure vogliono accedere alla stessa periferica, quale dei due ha la priorità? La risposta viene data con l'introduzione delle politiche di priorità (**politiche di scheduling**).

Virtualizzazione della memoria Consiste nel fabbricare l'illusione che ogni processo abbia il proprio spazio di indirizzi virtuali privato (**address space**) al quale accede e sarà il sistema operativo ad occuparsi di mappare nella memoria fisica.

Con la virtualizzazione è fondamentale riuscire a distinguere i processi in esecuzione. Per fare ciò viene associato un **PID** (process id) ad ogni job. il PID è un numero univoco.

1.1.2 Concorrenza

Si riferisce a tutta quelle serie di problemi che sorgono, e che vanno risolti, quando all'interno dello stesso programma più entità lavorano in parallelo. Le entità in questione si chiamano **threads**.

1.1.3 Persistenza

La persistenza è legata alla memorizzazione dei dati all'interno della memoria. La non volatilità delle memorie ha introdotto la possibilità di memorizzare dati in modo persistente. Il software nel sistema operativo che generalmente gestisce i dischi è chiamato **file system**.

1.1.4 Protezione ad anelli

Un modello di potezione implementato dal sistema operativo è quello ad anelli. Ci sono 5 livelli e 3 anelli differenti. A ciascun anello corrisponde un relativo livello di sicurezza.

- **Level 1** *Hardware level* qui vengono eseguiti, ad esempio, i device drivers visto che essi richiedono accesso diretto all'hardware dei dispositivi (microcontroller).
- **Level 2** *Firmware level* Il firmware sta in cima al livello elettronico. Contiene in software necessario dal dispositivo hardware e dal microcontroller.
- **Level 3: ring 0** *Kernel level* Questo è il livello dove opera il kernel, dopo la fase di bootload siamo qui.
- **Level 4: ring 1 e 2** *Device drivers* I device drivers passano atraverso il kernel per accedere all'hardware.
- **Level 5: ring 3** *Application level* Qui è dove viene eseguito normalemente il codice utente.

1.2 Processi

Sistema multiprogrammato Sistema nel quale è possibile eseguire più programmi contemporaneamente, idea alla base della virtualizzazione.

1.2.1 Multiprogrammazione

Time sharing prevede che il tempo di CPU sia equamente diviso fra i programmi in memoria.

Real time sharing La politica di scheduling è differente. Alcuni processi vanno serviti prima di altri.

1.2.2 Virtualizzazione della CPU

L'illusione consiste nel rendere indipendenti il numero di processi dal numero di processori. Si vuole disaccoppiare le entità logiche (*processi*), dalle entità fisiche (*processori*), in modo tale che ad ogni processo venga assegnato un processore logico mappato su processore fisico.

I concetti fondamentali alla base della virtualizzazione sono:

- **Time sharing** Meccanismo mediante il quale il tempo di CPU viene diviso equamente fra i processi.
- **Context switch** Meccanismo che consente di interrompere l'esecuzione di un processo in corso sulla CPU fisica e assegnare quest'ultima ad un nuovo processo.

1.2.3 Processi

Un processo è un programma in esecuzione, il sistema operativo deve fornire alcune interfacce (**APIs**) per la gestione dei processi che permettano di fare:

- **Create** Creazione di un nuovo processo.
- **Destroy** Eliminazione forzata di un processo. Molti processi termineranno per conto loro, ma l'utente potrebbe voler eliminare processi non ancora terminati.
- **Wait** Mette in attesa un processo.
- **Miscellaneous control** Sospensione di un processo per farlo ripartire dopo un certo tempo.
- **Status** Interfacce che restituiscono lo stato e altre informazioni di un processo.

Creazione di un processo La prima cosa che deve fare il sistema operativo per eseguire un programma è caricare il suo codice ed eventuali dati statici da disco a memoria, nell'address space del processo.

- **Allocazione dello stack** Un po' di memoria deve essere creata per lo stack del programma (*variabili locali, parametri delle funzioni e indirizzi di ritorno*).
- **Allocazione dello heap** Un po' di memoria deve essere creata per lo heap del programma (*dati allocati dinamicamente*).
- **Inizializzazione I/O** Standard input, output ed error.
- **Salto ed esecuzione** Salto all'entry point ed esecuzione. (*main*)

Stato di un processo

- **Running** È in esecuzione sul processore.
- **Ready** In attesa di essere eseguito dal processore.
- **Blocked** In stato di block, il processo sta esegundo qualche operazione (*es: I/O*).

Strutture dati Il sistema operativo deve tenere traccia delle informazioni fondamentali di un processo per poter ripristinare l'esecuzione di un processo interrotto. Esse sono:

- Porzioni di memoria coinvolte.
- Valori dei registri di CPU usati dal processo.
- Stato dei dispositivi di I/O usati dal processo.

Questi dati sono organizzati in strutture chiamate **Process Control Block (PCB)**, salvate in un per-process **kernel stack**, il quale risiede nel kernel space.

1.2.4 Process API

La creazione di un processo avviene tramite la `fork()`, la quale genera un processo identico a quello in esecuzione. Tale processo prende il nome di padre, quello generato viene chiamato figlio. L'esecuzione del processo figlio parte dall'istruzione successiva alla `fork()`. La `fork()` ritorna al figlio 0, al padre il **PID** del figlio e **-1** in caso di errore. Il processo figlio avrà il **proprio** address space, registri, PC, ecc...

La `wait()` è una funzione che forza il padre ad aspettare che il processo figlio termini la propria esecuzione. Senza, l'output potrebbe essere **non-deterministico** e potrebbero crearsi processi orfani o zombie. Esiste anche la `waitpid` che viene usata se si ha a che fare più di un figlio.

Per eliminare un processo esiste la funzione `kill()`. Solo il padre può distruggere il figlio. Ciò può portare alla creazione di processi **zombie** (processi terminati la cui **PCB** è ancora in memoria).

La `exec()` serve per generare un processo che fa qualcosa di diverso da quello padre. `exec(nome_programma, arg)` prende il nome di un eseguibile e alcuni argomenti, carica il codice e i dati statici di quell'eseguibile, sovrascrivendo il code segment corrente all'interno del PCB del figlio. Heap, stack e altre parti di memoria vengono re-inizializzate. Rimane la relazione padre-figlio.

1.3 Context Switch

1.3.1 Shell

Come mai `fork()` ed `exec()` sono due system call separate? Per rispondere introduciamo la **shell**.

La shell è un programma del sistema operativo `Unix` il cui compito è riconoscere ed eseguire altri programmi; si può dire che essa sia il genitore di tutti i processi che vengono mandati in esecuzione. Nello specifico, essa esegue una `fork()`, cambia il file descriptor se richiesto, ed infine invoca la `exec()`. Poi si mette in attesa che il programma abbia terminato prima di tornare in attesa di istruzioni. Esistono due tipi di shell, grafica (*terminale*) e interattiva (*aprire programmi col mouse*).

La separazione di `fork()` ed `exec()` è dovuta alla presenza della shell, con la quale possiamo andare ad effettuare alcune modifiche dopo la `fork()` e prima dell'`exec()`, come ad esempio la sostituzione del file descriptor.

```
$> wc file.c > n.txt
```

La shell esegue la `fork()` per poter mandare in esecuzione il programma `wc`. Prima di sostituire il codice del padre all'interno del PCB del figlio, sostituisce il file descriptor relativo allo standard output con n.txt. Successivamente esegue l'`exec()` producendo l'output desiderato all'interno di n.txt. Queste manipolazioni non sarebbero possibili se `fork()` ed `exec()` fossero un'unica system call perchè non si avrebbe accesso al PCB del figlio prima dell'exec().

1.3.2 Direct execution

Il concetto di direct execution è semplice: il programma viene eseguito direttamente sulla CPU fisica.

Quando il sistema operativo desidera iniziare l'esecuzione di un programma, viene fatto quanto segue:

- Crea una entry nella lista dei processi.
- Alloca la memoria per il programma.
- Carica il programma in memoria.
- Imposta lo stack con `argc/argv`.
- Pulisce i registri.
- Esegue la chiamata a `main()`.
  Si ha un salto dalla zona kernel al `main`. Il processo a questo punto deve:
- Eseguire il codice del `main()`.
- Ritornare dal main a fine esecuzione.
  Dal processo si torna alla zona kernel. Il sistema operativo infine:
- Rimuove la entry dalla lista dei processi.

Tuttavia la direct execution solleva alcune problematiche:

- Il sistema operativo non può assicurarsi che un programma in esecuzione non faccia qualcosa che non dovrebbe fare.
- Il sistema operativo non può fermare un processo in esecuzione.

Il primo problema si risolve con l'introduzione dello **user mode**. Il codice che viene eseguito in questa modalità di elaborazione è limitato in termini di istruzioni eseguibili. Nasce quindi anche la **kernel mode**, modalità in cui opera il sistema operativo e che consente di eseguire tutte le istruzioni privilegiate.

Per permettere ad un processo di eseguire istruzioni privilegiate vengono introdotte delle **system call**. Per eseguirle, un programa deve eseguire un'istruzione **trap** (*interrupt via software*). Questa istruzione salta nel kernel, aumenta i privilegi a kernel mode, esegue le operazioni privilegiate e ritorna al processo scalando i privilegi tramite un'istruzione **return-from-trap**. Durante questo procedimento bisogna assicurarsi di salvare i registri del chiamante. Per sapere dove la trap deve saltare, il kernel imposta una **trap table** al boot time. Non è il processo utente a specificare l'indirizzo dei **trap handlers** perchè potrebbe saltare ovunque nel sistema.
Per specificare la system call, generalmente viene assegnato un **system-call-number** che solitamente viene inserito in un registro appropriato.

1.3.3 Switch tra processi

Cooperative approach Soluzione via software che consiste nel programmare il processo in modo che, dopo un certo numero di secondi di utilizzo della CPU, il comando torni al sistema operativo. Il problema è che se vengono creati loop infiniti nel programma, la CPU non verrebbe mai condivisa.

Time interrupt Soluzione via hardware che consiste nel creare una nuova componente che genera un segnale elettrico (**time interrupt**) dopo un certo lasso di tempo. Ci sarà quindi un orologio interno che invierà un segnale al piedino del microprocessore. L'hardware deve inoltre fermare l'esecuzione del processo corrente, salvarne lo stato per dare il controllo allo **scheduler**, che nel caso decidesse di cambiare processo, farà eseguire al sistema operativo codice a basso livello che prende il nome di **context switch**.

Context switch Ciò che deve fare il sistema operativo è salvare alcuni valori dei registri per il processo in corso di esecuzione (*nel kernel stack*) e ripristinarne altri per il processo scelto. Viene eseguita una return-from-trap per mandare in esecuzione il processo scelto.
Interrupt, system call ed eccezioni sono eventi che inducono il mode switch.

1.4 Scheduling policy

Dati n processi, a quale assegno il processore? La scelta è fatta dallo **scheduler**, un modulo del sistema operativo che implementa una politica decisionale.

CPU burst è l'intervallo di tempo in cui viene usata intensamente la CPU.

I/O burst è l'intervallo di tempo in cui viene usato intensamente I/O.

CPU bound processi con CPU burst lunghi, ad esempio compilatori, simulatori, calcolo del tempo, ecc...

I/O Bound processi con I/O burst lunghi, ciò comporta maggiore interattività con l'utente.

Stato di IDLE è lo stato in cui è una risorsa accesa e funzionante ma non utilizzata.

Un processo in esecuzione si trova o in CPU burst o in I/O burst. Lo scheduler, per essere efficiente, deve ottimizzare l'uso delle risorse in modo tale che, se la CPU è occupata con l'esecuzione di un processo, i dispositivi di I/O lo sono con un altro e viceversa. L'ottimizzazione della CPU viene dunque portata mediante lo scheduler. Per valutare la bontà di un algoritmo di scheduling si devono introdurre delle metriche di valutazione.

$$T_{turnaround} = T_{termine} - T_{arrivo}$$

$$T_{response} = T_{first-exec} - T_{arrivo}$$

$$T_{wait} = T_{turnaround} - T_{job}$$

1.4.1 Algoritmo FIFO

L'algoritmo FIFO (*First In First Out*) mette in esecuzione il primo processo arrivato. Il problema a cui può portare questo algoritmo è l'**effetto convoglio**, ovvero quando un certo numero di piccoli consumatori di una risorsa vengono messi in coda dietro un enorme consumatore.

1.4.2 Algoritmo SJF

L'algoritmo SJF (*Shortest Job First*) mette in esecuzione il processo con CPU burst minore. In questo modo si evita l'effetto convoglio, ma solo se i processi arrivano allo stesso istante.

1.4.3 Algoritmo STCF

L'algoritmo STCF (*Shortest Time to Completion First*) ogni volta che arriva un processo, lo compara il processo in esecuzione e lascia il processore a quello che ha CPU burst minore.

SJF e STCF funzionano molto male per quanto riguarda il tempo di risposta (spesso possono indurre anche al verificarsi della starvation). Inoltre non conoscono a priori il CPU burst di un processo, perciò sono solo algoritmi teorici.

1.4.4 Round Robin

L'algoritmo **Round Robin** assegna un **quanto di tempo** ad ogni processo. Viene inizializzato un timer che, una volta arrivato a zero, forza un context switch. Il quanto di tempo va scelto bene, altrimenti si hanno troppi context switch se è troppo piccolo, o degenera in FIFO se è troppo grande.

1.5 Multilevel feedback scheduler

Il problema che **MLFQ** (*MultiLevel Feedback Queue*) cerca di risolvere è:

- Ottimizzare il $T_{turnaround}$.
- Aumentare l'interattività utente/sistema, minimizzando il $T_{response}$.

L'approccio che si usa consiste nell'avere un certo numero di **code** distinte, ognuna assegnata ad un diverso **livello di priorità**. MLFQ sfrutta i diversi livelli di priorità per decidere quale processo eseguire: viene scelto quello all'interno della coda di priorità maggiore. Se ci sono più processi all'interno di una certa coda, viene usato **RR**.

- 1. If priority(A) > priority(B), A runs (B doesn't).
- 2. If priority(A) = priority(B), A & B run in RR.
- 3. Quando un processo entra nel sistema, viene posizionato nella coda di priorità massima.
- 4a. Se un processo utilizza tutto il lasso di tempo a disposizione durante l'esecuzione, la sua priorità viene ridotta.
- 4b. Se un processo libera la CPU prima di terminare il lasso di tempo a disposizione, il livello di priorità rimane invariato.
- 5. **Priority boost** Dopo un certo periodo di tempo, tutti i processi vengono spostati nella coda di priorità più alta. (*Evita la starvation dei long running jobs e il monopolio della CPU se qualche processo la rilascia poco prima del lasso di tempo.*)

1.5.1 Better accounting

La scelta del tempo è cruciale, se settato troppo grande, i long running jobs potrebbero ancora andare in starvation, se impostato troppo piccolo, i processi interattivi potrebbero non avere una porzione adeguata dela CPU. Per evitare che possa essere raggirato l'algoritmo di scheduling, lo scheduler tiene tracia di quanto tempo ha consumato un processo in un certo livello di **MLFQ**. Le regole 4a e 4b diventano:

- 4. Una volta che un processo ha usato il tempo a disposizione in un certo livello (indipendentemente da quante volte ha rilasciato la CPU), la sua priorità viene ridotta.

1.6 Address space

Un programma per essere eseguito deve risiedere in memoria. Essa può essere usata implicitamente (**stack:** `int x`) o esplicitamente (**heap:** `int *x = malloc(sizeof(int))`). Lo stack è gestito autonomamente, lo heap è gestito dal programmatore attraverso opportune funzioni (`malloc, realloc, free, ...`), per cui non si conosce a priori la dimensione.

1.6.1 Memory API

- `malloc()` Riceve in input un argomento di tipo `size_t` (numero di bytes), se ha successo restituisce un puntatore all'inizio della zona allocata nello **heap**, se fallisce restituisce NULL.
- `free()` Riceve in input un puntatore, la grandezza della regione da liberare viene tenuta nella libreria `memoryallocation`.

La system call per la gestione diretta della memoria è `int brk(void *addr)`

1.6.2 Memory errors

- **Dimenticarsi di allocare la memoria.** È da patchare il fatto che si possa indurre un `segfault` in modo tale da poter accedere al core dump della memoria e vedere dati sensibili (**attacchi core-dump**). È necessario eliminare questi dati dopo il loro utilizzo.
- **Non allocare abbastanza memoria.** Può portare a vulnerabilità come il **buffer overflow**.
- **Dimenticarsi di inizializzare memoria allocata.** Potrebbero esserci valori come 0 o valori random.
- **Dimenticarsi di liberare la memoria.** Il **memory leak** può portare ad un esaurimento della memoria disponibile.
- **Liberare la memoria prima di aver finito di usarla.** Questo errore è chiamato **dangling pointer**, può causare un crash o la sovrascrittura di memoria valida.

- **Liberare la memoria più di una volta.** Problema noto come **double free**, il risultato è indefinito, la libreria `memory-allocation` potrebbe confondersi e fare cose strane. I crash sono la cosa più comune.
- **Chiamata di `free()` incorretta.** La funzione si aspetta un puntatore prodotto in precedenza da una `malloc()`. Quando viene passato alla `free` un valore diverso, possono succedere cose brutte e pericolose.

1.6.3 Virtualizzazione della memoria

Con l'avvento della **multiprogrammazione** la memoria diviene una risorsa condivisa, bisogna iniziare a far fronte a tutte le problematiche che ciò comporta.

- **Protezione** un processo non può invadere lo spazio di un altro.
- **Interattività** Ci devono essere molti processi in esecuzione.

Il meccanismo di astrazione che si vuole implementare prende il nome di **address space**, esso è il punto di vista di un processo sulla memoria del sistema, ovvero l'astrazione che il sistema operativo gli fornisce.

Gli obiettivi della virtualizzazione della memoria sono riassunti come segue:

- **Trasparenza.** Il programmatore scrive il codice indipendentemente dalla grandezza della memoria.
- **Efficienza.** Il meccanismo di virtualizzazione non deve avere overhead troppo elevato.
- **Protezione.** Bisogna proteggere i processi da altri processi, dal sistema operativo, e viceversa.

1.6.4 Mapping

Il **mapping** consiste nel trovare una corrispondenza fra indirizzo logico e indirizzo fisico. Nei sistemi **monoprogrammati** ciò era facile poichè ogni programma veniva mappato a partire dall'indirizzo `64KB` fino alla fine. Il compilatore assegnava ai programmi indirizzi costanti. Nel caso della **multiprogrammazione** invece, il compilatore assegna indirizzi preliminarli al programma, i quali vengono successivamente rilocati.

1.6.5 Base e Bound

Questa tecnica di mapping utilizza due registri, **base** e **bound**. Assunzioni:

- Il programma viene caricato in locazioni contigue di memoria. (Un programma da `32KB` verrà caricato in `32KB` locazioni adiacenti)
- L'indirizzo logico è sempre minore dell'indirizzo fisico.

Mediante la rilocazione siamo in grado di calcolare l'indirizzo fisico come segue:

```
indirizzo fisico = indirizzo logico + Base
```

Base è un registro contenente il punto di partenza (indirizzo fisico) del programma. **Bound** è il registro limite. Se un processo prova a saltare in zone di un altro processo viene generato un errore di segmentazione.

1.6.6 MMU

MMU sta per **Memory Managment Unit** ed è una componente hardware per la rilocazione degli indirizzi. L'input è un indirizzo logico prodotto dalla CPU, l'output è l'indirizzo fisico. Generalmente questa traduzione viene fatta a runtime. Prima di eseguire l'istruzione a cui sto puntando, l'indirizzo logico viene tradotto in indirizzo fisico (**rilocazione dinamica**).
Ore che abbiamo la rilocazione dinamica, il sistema operativo deve fare le seguenti cose per implementare la memoria virtuale:

- Quando un nuovo processo viene creato, il sistema operativo dovrà cercare in una struttura dati (spesso chiamata **free list**) spazio libero per il nuovo address space e marcarlo come in uso.
- Quando un processo termina, deve riabilitare tutta la memoria allocata per il processo all'interno della free list e pulire ogni struttura dati associata ad esso.
- Quando avviene un context switch deve salvare nel PCB i registri base e bound e ripristinare quelli del nuovo processo.
- Quando un processo viene fermato è possibile muovere un address space da una locazione di memoria a un'altra. Basta deschedularlo, copiare l'address space dalla locazione corrente a quella nuova e infine aggiornare il registro **base**.

Il sistema operativo deve fornire degli **exception handler**. Per esempio, se un processo prova ad accedere a memoria al di fuori del suo **bound**, la CPU deve sollevare un'eccezione.

1.7 Segmentazione

1.7.1 Binding

Durante il processo di rilocazione vengono cambiati tutti gli indirizzi del programma per evitare che vadano fuori dallo spazio di indirizzamento previsto. Il **binding** è l'operazione che viene fatta per modificare gli indirizzi. Può essere:

- **Early binding.** Rilocazione degli indirizzi fatta a **compile time**. Il compilatore deve conoscere la posizione di partenza del programma in memoria, ma funziona solo quando il compilatore genera direttamente il codice assoluto (*sistemi embedded, monoprogrammati, ...*).
- **Delayed binding.** La rilocazione degli indirizzi viene fatta durante il trasferimento del programma da disco a memoria (*operazione svolta dal sistema operativo prima dell'introduzione dell'MMU*).
- **Late binding.** La rilocazione degli indirizzi viene fatta immediatamente prima di eseguire l'istruzione corrente, quindi a **runtime**. Per implementare questa tecnica serva l'MMU.

1.7.2 Segmentazione

Con la tecnica base e bound, c'è dello spazio potenzialmente non utilizzato tra lo stack e lo heap. L'idea alla base della **segmentazione** è quella di dividere il programma in **segmenti** che possono essere caricati in porzioni di memoria differenti siccome ad ognuno di essi è associata una coppia base-bound. I segmenti sono inseriti in modo indipendente all'interno della memoria fisica, in questo modo siamo in grado di evitare gli sprechi. Questo risparmio di memoria, tuttavia, complica notevolmente l'MMU, la quale deve gestire più segmenti presenti all'interno della memoria (ogni processo ha tre segmenti). Il meccanismo funziona come segue:

- **Input:** indirizzo logico B (prodotto dal compilatore).

- Individua il segmento s di appartenenza dell'indirizzo B.
- Calcola l'offset k sottraendo all'indirizzo virtuale l'indirizzo di partenza (logico) del segmento ($k = B$ - indirizzo iniziale di s)
- Viene calcolato l'indirizzo fisico sommando k e il base register (Indirizzo fisico = $Base(s) + k$)

Se un processo cerca di produrre un indirizzo illegale, l'hardware rileverà che l'indirizzo è out of bounds, trap nel sistema operativo, il quale terminerà il processo (**segmentation fault**).
L'hardware per conoscere il segmento e l'offset taglia l'address space in segmenti basati sui primi bit dell'indirizzo virtuale (**approccio esplicito**). Nell'**approccio implicito** invece l'hardware determina il segmento in base a come è formato l'indirizzo. Se, ad esempio, l'indirizzo è stato generato dal program counter, appartiene al code segment; se è dello stack o del base pointer, deve appartenere al segmento stack. Ogni altro indirizzo viene interpretato come parte del segmento heap.

1.7.3 Stack

Siccome lo stack cresce al contrario, invece dei soli valori base e bound, l'hardware ha bisogno di sapere in quale direzione cresce il segmento (un bit settato a 1 se il segmento cresce positivamento, 0 negativamente). Il controllo del bound register viene fatto in valore assoluto.

1.7.4 Permessi

Code sharing. Per risparmiare memoria, a volte è utile condividere certi segmenti tra gli address spaces. Per supportare la condivisione abbiamo bisogno di **protection bits** da parte dell'hardware. Vengono aggiunti solamente pochi bit per segmento, a indicare quando un programma può leggerne, scriverne o eseguirne il codice contenuto.

1.7.5 Coarse grained and fine grained

Gli esempi visti fin'ora utilizzavano la tecnica **coarse grained** (poche fette relativamente grandi). Alcuni dei primi sistemi erano più flessibili e permettevano che gli address spaces consistessero in un gran numero di piccoli segmenti, questo concetto era espresso come segmentazione **fine grained**.

Ciò richiede un ulteriore supporto hardware, una **segment table** all'interno della memoria.

1.7.6 Frammentazione

La segmentazione solleva un numero di nuove problematiche:

- Cosa dovrebbe fare il sistema operativo a fronte di un context switch? I segment registers devono essere salvati e ripristinati.
- Come viene gestito lo spazio libero in memoria fisica? Quando un nuovo address space viene creato, il sistema operativo deve essere in grado di trovare lo spazio in memoria fisica per i suoi segmenti.

Il problema generale è che la memoria fisica consuma velocemente piccoli spazi liberi, rendendo difficile l'allocazione di nuovi segmenti o la crescita di quelli già esistenti. Questo problema è noto come **frammentazione esterna**. Si può risolvere con la **deframmentazione**, compattando la memoria fisica e riarrangiando i segmenti esistenti, copiando i dati dei segmenti in una regione contigua di memoria e cambiando il valore dei loro segment registers. Questa operazione è piuttosto complessa e dispendiosa oltre che bloccante. Un approccio più semplice è quello di usare un algoritmo per la gestione della **free-list** che tenta di mantenere un elevato spazio disponibile contiguo in memoria. Purtroppo però la frammentazione esisterà sempre a prescindere da quanto buono sia l'algoritmo per minimizzarla.

1.8 Paginazione

La **paginazione** nasce per gestire in modo ottimale lo spazio libero in memoria e l'address space di un programma. Consiste nel tagliare gli spazi in fette di una certa dimensione. Anzichè dividere l'address space di un processo in segmenti, esso viene diviso in unità di dimensione fissata, ognuna delle quali è chiamata pagina.
Vediamo la memoria fisica come un array di slots di dimensione fissata, chiamati **page frames**. Ogni frame può contenere una singola pagina di memoria virtuale. Ciò porta ad alcuni vantaggi:

- **Flessibilità.** Il sistema sarà in grado di supportare l'astrazione dell'address space efficacemente, a prescindere da come un processo ne fa uso. Non

vogliamo, ad esempio, dover fare assunzioni riguardo la direzione di crescita dello heap e dello stack e come vengono usati.

- **Semplicità** della gestione dello spazio libero. Per esempio, supponiamo che il sistema operativo desideri inserire il nostro addess space da `64B` in memoria fisica. Siccome i programmi sono divisi in pagine di dimensione fissata, il problema della segmentazione viene ridotto di molto visto che, siccome il sistema operativo tiene traccia della free list, gli basta semplicemente prendere il primo frame disponibile e assegnarlo a una pagina.

Per memorizzare dove ogni pagina virtuale dell'address space è posizionata in memoria fisica, il sistema operativo tiene una struttura dati per ciascuno processo nota come **page table**. Il ruolo principale della page table è di memorizzare, per ogni pagina virtuale dell'address space, il corrispondente frame fisico.

1.8.1 Address translation

Per tradurre l'indirizzo virtuale generato da un processo, dobbiamo per prima cosa dividerlo in **Virtual Page Number (VPN)** e **offset**. Siccome si conosce la dimensione di ciascuna pagina, si può dividere l'indirizzo virtuale in:

- **VPN:** Bit più significativi che fanno da indice per accedere alla page table del processo per trovare il frame fisico corrispondente (**PFN**).
- **Offset:** Bit che servono per indirizzare la grandezza di una pagina.

A questo punto si traduce l'indirizzo virtuale in fisico sostituendo il **Physical Frame Number (PFN)** al VPN.

1.8.2 Page tables

Le page tables possono essere terribilmente grandi. Per esempio, immaginiamo un address space da `32 bit` con pagine da `4KB`. L'indirizzo virtuale sarà diviso in `20 bit` di VPN e `12 bit` di offset. `20 bit` di VPN implicano 2^{20} possibili traduzioni per ogni processo. Assumendo di aver bisogno di `4B` per **page table entry (PTE)** per mantenere la traduzione fisica più ogni altra informazione utile otteniamo `4MB` di memoria necessari per ogni page table.

Con 100 processi in esecuzione, questo significa che il sistema operativo avrà bisogno di **400MB** di memoria.

Cosa contiene una page table? La page table è sempicemente una struttura dati usata per mappare gli indirizzi virtuali in indirizzi fisici. La forma più semplice è chiamata **page table lineare** che è semplicemente un array. Il sistema operativo indicizza l'array con il VPN e consulta la PTE a quell'indice per trovare il PFN desiderato.
Ogni PTE contiene diversi bit:

- **Valid bit.** Indica quando una particolare traduzione è valida. Per esempio, quando un programma inizia l'esecuzione, avrà code e heap a un'estremità del suo spazio di indirizzamento e lo stack dall'altra. Tutto lo spazio non utilizzato in mezzo sarà marcato come invalido e se il processo tenterà di accedervi, verrà generata una trap al sistema operativo che lo terminerà. È cruciale per supportare un address space sparso.
- **Protection bits.** Indicano quando una pagina può essere letta, scritta o eseguita. Accedere a una pagina in modo non consentito da questi bit genererà una trap nel sistema operativo, il quale terminerà il processo.
- **Present bit.** Indica se la pagina in questione è in memoria fisica o su disco. Consente al sistema operativo di swappare le pagine liberando la memoria fisica.
- **Dirty bit.** Indica se la pagina è stata modificata da quando risiede in memoria.
- **Reference bit.** Viene usato per tenere traccia se una pagina è stata acceduta da quando risiede in memoria.

1.8.3 Quanto è lenta la paginazione?

Per ogni riferimento a memoria (sia per prelevare un'istruzione che per un load o store esplicito), la paginazione ne necessita uno aggiuntivo per prelevare la traduzione dalla page table. I riferimenti a memoria aggiuntivi sono costosi e in questo caso rallenteranno il processo di un fattore pari a due o più.

1.9 Translation Lookaside Buffer

Siccome le informazioni di mappatura risiedono generalmente in memoria fisica, la paginazione richiede un accesso aggiuntivo per ogni indirizzo virtuale generato dal programma. L'obbiettivo è snellire la tecnica introdotta, cercando di **diminuire il numero di accessi a memoria fisica** (alla page table). Viene aggiunta alla MMU una cache hardware delle traduzioni virtual-to-physical più popolari chiamata **translation lookaside buffer o TLB**. Per ogni indirizzo virtuale, l'hardware controlla per prima cosa il TLB per vedere se la traduzione desiderata è presente al suo interno.

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN);
if (Success == True){ //TLB HIT
  if (CanAccess(TlbEntry.ProtectBits == True){
    Offset = VirtualAddress & OFFSET_MASK;
    PhysAddr = (TlbEntry.PFN << SHIFT) | Offset;
    Register = AccessMemory(PhysAddr);
  }
  else
    RaiseException(PROTECTION_FAULT);
}
else{              //TLB MISS
  PTEAddr = PTBR + (VPN * sizeof(PTE));
  PTE = AccessMemory(PTEAddr);
  if(PTE.Valid == False)
    RaiseException(SEGMENTATION_FAULT);
  else if (CanAccess(PTE.ProtectBits) == False)
    RaiseException(PROTECTION_FAULT);
    else{
      TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits);
      RetryInstruction();
    }
}
```

L'algoritmo che l'hardware segue funziona in questo modo:

- Estrae il VPN dall'indirizzo virtuale.
- Controlla se il TLB contiene la traduzione per il VPN. Se così fosse, abbiamo un **TLB hit**, la traduzione è cioè contenuta in cache.
- Se la CPU non trova la traduzione nella TLB abbiamo un **TLB miss**. L'hardware accede alla page table per trovare la traduzione e, as-

sumendo che l'indirizzo virtuale generato dal processo sia valido e accessibile, aggiorna il contenuto del TLB con la nuova entry. Queste operazioni sono parecchio costose.

- Una volta che il TLB è aggiornato, l'hardware riprova l'istruzione, ottenendo un TLB hit.

1.9.1 Performance e località

Il TLB migliora le performance grazie al **principio di località**. Esso si divide in:

- **Spaziale.** Se la CPU sta eseguendo un'istruzione presente in memoria, vuol dire che con molta probabilità le prossime istruzioni da eseguire si troveranno fisicamente nelle vicinanze di quella in corso.
- **Temporale.** Se accedo all'istruzione 100 al tempo t_0, con molta probabilità acederò nuovamente ad essa negli istanti di tempo successivi.

1.9.2 TLB miss

Chi gestisce un TLB miss? Ci sono due possibili risposte:

- **Hardware.** L'HW deve sapere la posizione delle page tables in memoria (attraverso il page table register), oltre al loro formato esatto. In presenza di un miss, l'HW deve accedere alla page table, trovare la PTE corretta, estrarre la traduzione desiderata, aggiornare il TLB con la pagina contenente l'indirizzo fisico ricercato e riprovare l'istruzione.
- **Software (S.O).** Al verificarsi di un TLB miss, l'hardware solleva un eccezione per mettere in pausa il flusso corrente di istruzioni, aumenta i privilegi a livello kernel e salta a un trap handler. Questo trap handler è codice scritto all'interno del sistema operativo, il cui scopo è la gestione esplicita dei TLB misses. Il codice cercherà la traduzione nella page table, userà "speciali" istruzioni privilegiate per aggiornare il TLB e, infinie, eseguirà la **return-from-trap**. A questo punto, l'hardware riproverà l'istruzione (TLB hit).

TLB return from trap In questo caso, quando si torna da una TLB miss-handling trap, l'hardware deve ripristinare l'esecuzione dall'istruzione che aveva causato la trap nel sistema operativo.

Quando il TLB miss-handler è in esecuzione, il sistema operativo deve essere molto attento a non causare una catena infinita di TLB misses. Se ho un miss, viene generata un'eccezione. Bisogna fare un context switch per permettere al S.O. di gestire l'evento. Per mandarlo n esecuzione bisogna mettere l'indirizzo del TLB miss-handler nel PC. Questo indirizzo tuttavia, come tutti gli altri, viene passato all'MMU. Quest'ultima lo cerca nel TLB, ottenenedo un miss. Parte quindi un loop. La soluzione che viene adottata per risolvere questo problema consiste nel tenere il miss handler all'interno del TLB.

1.9.3 TLB - contenuto

Una address-translaion cache tipica potrebbe avere 32, 64 o 128 entries ed essere ciò che viene chiamato **fully associative**. Ciò significa che una traduzione potrebbe essere ovunque nel TLB e l'hardware dovrà cercare in parallelo fino a trovare la traduzione desiderata. Una entry del TLB ha il seguento aspetto: `VPN | PFN | other bits`.
Tra gli other bits generalmente ci sono il **valid bit**, i **protection bits**, ...

1.9.4 TLB - Context Switch

Il TLB contiene traduzioni virtual to physical che sono valide per il processo in esecuzione ma prive di significato per gli altri. Bisogna assicurarsi che quando cambiamo processo, il processo che sta per essere eseguito non usi le traduzioni di quello precedente. Un approccio semplice ma inefficace è fare un **flush** (impostando tutti i valid bit a 0) del TLB a fronte di un context switch. Ogni volta che un processo verrà eseguito, incapperà in TLB misses. Per ridurre questo overhead, alcuni sistemi aggiungono un supporto hardware per abilitare la condivisione del TLB attraberso context switcher. In particolare alcuni sistemi hardware forniscono un campo **Address Space Identifier** (ASID) nel TLB.

1.10 Multi Level Page Tables

Le page tables sono grandi e consumano troppa memoria.

1.10.1 Bigger pages

Una possibile soluzione è quella di fare pagine più grandi. Il problema è che questo comporta a sprechi di spazio all'interno delle pagine stesse (**frammentazione interna**). La memoria si riempie subito di pagine contenenti parecchio spazio vuoto.

1.10.2 Paginazione e segmentazione

Assumiamo di avere un address space nel quale la porzione usata da stack e heap è piccola. Per esempio, usiamo uno spazio di indirizzamento da `16KB` con pagine da `1KB`. La page table relativa a questo address space sarà quindi:

PFN	valid	prot	present	dirty
10	1	r-x	1	0
-	0	_	-	-
-	0	_	-	-
-	0	_	-	-
23	1	rw-	1	1
-	0	_	-	-
-	0	_	-	-
-	0	_	-	-
-	0	_	-	-
-	0	_	-	-
-	0	_	-	-
-	0	_	-	-
-	0	_	-	-
-	0	_	-	-
28	1	rw-	1	1
4	1	rw-	1	1

Come possiamo osservare dalla figura, la maggior parte della page table non è utilizzata. Invece di avere una singola page table per l'intero address space del processo, perchè non averne una per segmento logico? Con la segmentazione avevamo un registro *base* che ci diceva dove ogni segmento risiedeva in memoria fisica e un registro *bound* che ne esprimeva la grandezza. Nel nostro approccio ibrido, abbiamo queste strutture nell'MMU; qui, non usiamo il *base* per puntare al segmento stesso, ma teniamo l'indirizzo fisico della page table di quel segmento. Il registro *bound* è usato per indicare la fine della

page table relativa a un segmento.
Nell'hardware, assumiamo che ci siano tre paia di *base/bound*: una per code, heap e stack. In un context switch, questi registri devono essere cambiati per riflettere la locazione delle page tables del nuovo processo in esecuzione. In un TLB miss, l'hardware usa i bits del segmento per determinare quale coppia *base/bound* usare. L'hardware quindi prende il *base* corretto e lo combina col VPN come segue, per formare l'indirizzo della PTE:

```
SN = (VirtualAddress & SEG_MASK) >> SN_SHIFT
VPN = (VirtualAddress & VPN_MASK) >> VPN_SHIFT
AddressOfPTE = Base[SN] + (VPN * sizeof(PTE))
```

La differenza critica, sta nella presenza di un registro *bound* per segmento. Ogni *bound* contiene il valore della massima pagna valida nel segmento. Se il code segment sta usando le sue prime tre pagine (0, 1 e 2), la page table del segmento avrà solamente tre entries allocate e il bound register sarà impostato a 3. In questa maniera, il nostro approccio ibrido realizza un risparmio di memoria significativo rispetto alla classica page table lineare.

Purtroppo questo approccio si porta dietro con sè tutti i problemi della segmentazione, e se avessimo ad esempio heap molto grandi e sparsi in memoria, finiremmo con l'avere ancora una volta sprechi per via dele page table. Seconda cosa, il manifestarsi ancora una volta della frammentazione esterna.

1.10.3 Multi Level Page Tables

Un altro approccio potrebbe essere trasformare una page table lineare in qualcosa di sime a un albero.

- Per prima cosa, viene tagliata la page table in unità page-sized.
- Se una pagina di una PTE è invalida, non viene allocata.
- Per tenere traccia se una pagina delle page table è valida (e se valida, dove risiede in memoria), usiamo una nuova struttura chiamata **page directory**.

Ciò che fa la **multi-level table** è far scomparire parti della page table lineare e tenere traccia di quali pagine sono allocate.

La page directory, consiste in una serie di **page directory entries (PDE)**, le quali hanno un valid bit e un numero di page frame (**PFN** -

in questo caso rappresenta l'indirizzo di memoria dove è situata una page table). Il bit di valità è un po' diverso, se la PDE è valida significa che almeno una delle pagine della page table a cui la entry punta (via PFN) è valida.

Vantaggi:

- La multi-level table alloca spazio **solamente per la page table in proporzione all'ammontare di address space in uso** (*se c'è tanto spazio in mezzo tra due pagine utilizzate, vengono comunque allocate solo due pagine*).
- Se implementata correttamente, **ogni porzione della page table entra ordinatamente in una pagina**, rendendo più facile la gestione della memoria; il sistema operativo prende semplicemente la prossima pagina libera quando ha bisogno di allocare o far crescere una page table.

Abbiamo aggiunto l'**indirezione**, che ci permette di posizionare pagine della page table ovunque vogliamo in memoria fisica. Questa tecnica ha un costo: in un TLB miss saranno necessari due caricamenti da memoria per prelevare la traduzione corretta dalla page table (una per la page directory e una per la PTE stessa, *trade off time-space*). Nel caso medio (TLB hit), le performance sono **identiche** alla page table lineare. Un altro aspetto negativo è la **complessità**, che sia l'hardware o il sistema operativo a gestire la consultazione delle page tables.

1.10.4 Più di due pagine

Bisogna evitare che la page directory diventi troppo grande, altrimenti l'obiettivo di fare in modo che ogni pezzo della multi-level page table entri in una pagina svanisce. Quando si ha a che fare con pagine piuttosto piccole che lasciano parecchi bit di VPN, è preferibile splittare la page directory stessa in più pagine, aggiungendo un'altra page directory sopra ad essa.

1.11 Page Fault e Swap

Per supportare address spaces di grandi dimensioni (per permettere ai processi di non preoccuparsi se c'è abbastanza spazio in memoria), il sistema operativo avrà bisogno di posizionare altrove le pagine che non sono largamente richieste.

1.11.1 Swap space

La prima cosa da fare è riservare un po' di spazio su disco per muovere le pagine avanti e indietro. Nei sistemi operativi, ci riferiamo a questa locazione come **swap space**. La sua dimensione è importante, in quanto determina il numero massimo di pagine di memoria che possono essere usate da un sistema ad un dato istante di tempo.

1.11.2 Present bit

Quando l'hardware guarda nella PTE, potrebbe scoprire che la pagina non è presente in memoria fisica. Il modo in cui l'hardware (o il sistema operativo in caso di software-managed TLB) determina ciò è attraverso un nuovo **present bit** in ogni PTE. Se il present bit è a 0, la pagina è da qualche parte su disco. A fronte di un **page fault**, viene invocato il sistema operativo, il quale manda in esecuzione un **page-fault handler**.

1.11.3 Page Fault

Se una pagina non è presente, il sistema operativo viene messo al comando per gestire il page fault sia nei sistemi hardware-managed TLB che nei software-managed TLB. Il sistema operativo può usare i bits della PTE relativi al PFN come indirizzo su disco. Quando l'I/O del disco è completato, il sistema operativo aggiorna la page table per marchiare la pagina come presente e aggiorna il campo PFN della PTE e riprova l'istruzione. Questo nuovo tentativo potrebbe generare un TLB miss (è possibile aggiornare anche il TLB a seguito di un page fault per evitare questo scenario). Mentre viene fatto I/O il sistema operativo sarà libero di eseguire altri processi in ready.

1.11.4 Memoria piena

Il sistema operativo potrebbe voler prima swappare una o più pagine su disco per fare spazio a quelle nuove in procinto di caricare. Il processo di scegliere una pagina da sostituire è noto come **page replacement policy**. Spostare la pagina sbagliata può avere dei costi elevati in termini di performance (*si può causare una velocità disk-like, 10.000 o 100.000 volte più lento*).

A fronte di un page fault, il sistema operativo deve trovare il frame fisico per far risiedere la pagina, e se tale frame non c'è, bisogna aspettare che

l'algoritmo di replacement venga eseguito e liberi delle pagine dalla memoria rendendole disponibili per l'utilizzo.

1.11.5 Replacements

Piuttosto che aspettare che si riempa la memoria, il sistema operativo tiene un piccolo ammontare di memoria libera. In molti sistemi, vengono utilizzati un **high watermark** (HW) e un **low watermark** (LW) per facilitare la decisione di quando iniziare a sfrattare le pagine. Quando il sistema operativo nota che ci sono meno di LW pagine disponibili, un thread (**swap deamon**) in background rensponsabile della liberazione della memoria viene eseguito. Il thread sfratta le pagine fino a quando non ce ne sono HW disponibili.

1.12 Replacement policies

Decidere quale pagina (o pagine) sfrattare è incapsulato all'interno della **politica di replacement** del sistema operativo.

1.12.1 Cache management

È possibile vedere il nostro obiettivo come la massimizzazione del numero di cache hits. Conoscere il numero di cache hits e misses ci permette di calcolare l'**average memory access time** (AMAT).

$$AMAT = T_M + (P_{MISS} * T_D)$$

Dove T_M rappresenta il costo di accesso a memoria, T_D il costo di accesso a disco, e P_{MISS} la percentuale di miss (da 0.0 a 1.0).

1.12.2 Optimal replacement policy

La politica ottimale di replacement conduce al minor numero di misses in generale. Se dobbiamo sfrattare delle pagine, perchè non selezionare quelle che verranno usate più avanti nel tempo? È un approccio semplice ma difficile da implementare.

1.12.3 FIFO policy

FIFO (*first in first out*) ha un buon punto di forza: è semplice da implementare. Purtroppo non è in grado di determinare l'importanza dei blocchi,

se una pagina viene acceduta parecchie volte, FIFO deciderà comunque di sfrattarla.

1.12.4 Random policy

L'algoritmo random, che sceglie una pagina casuale da sostituire, ha proprietà simili al FIFO, è semplice da implementare ma non sceglie intelligentemente i blocchi da sfrattare.

1.12.5 LFU and LRU

Per evitare di sfrattare pagine importanti sfruttiamo il **principio di località**. Se un processo accede una pagina di recente, è molto probabile che quest'ultima verrà acceduta nuovamente nel futuro prossimo. Un tipo di informazione "storica" che potrebbe essere usata in una politica di page replacement è la **frequenza**. La politica **Least Frequently Used** (LFU) sostituisce le pagine usate meno di frequente. Simile è LRU **Least Recently Used** che sostituisce la pagina usata meno di recente.
Esistono anche una classe di algoritmi opposti, Most Frequently Used MFU e Most Recently used MRU.

1.12.6 LRU approssimato

Dato che scansionare tutti i tempi per trovare la pagina least recently used è molto costoso, possiamo usare un'approssimazione. L'idea richiede supporto hardware, nella forma di **use bit**. Questo bit è contenuto in ogni pagina del sistema e ogni volta che una di esse viene riferita (letta o scritta), lo use bit è settato dall'hardware a 1. L'hardware non pulisce mai il bit, è il sistema operativo che ha il compito di settarlo a 0. Ci sono molti modi ma il **clock algorithm** è un approccio molto semplice e funzionale. Immaginiamo tutte le pagine del sistema arrangiate in una lista circolare. Una **clock hand** punta a una pagina (non importa quale). Quando deve essere fatta una sostituzione, il sistema operativo controlla se la pagina puntata P ha lo use bit a 1 o a 0. Se a 1 non è un buon candidato per la sostituzione, il bit viene settato a 0 e la clock hand passa alla prossima pagina. L'algoritmo continua fino a quando non trova una pagina con use bit a 0. Se la pagina è **dirty**, deve essere riscritta su disco prima di essere sfrattata, quindi molti sistemi preferisco pulire pagine **clean**.

1.12.7 Trashing

Cosa dovrebbe fare il sistema operativo quando la memoria è semplicemente sovraccaricata e la richiesta di memoria dell'insieme dei processi in esecuzione eccede la memoria fisica disponibile? In questi casi il sistema è in costante paginazione (**trashing**).

- **Admission control.** Dato un insieme di processi, un sistema può decidere di non eseguirne un sottoinsieme.
- **Out of memory killer.** Quando la memoria è sovraccarica, questo demone sceglie un processo che sta usando intensamente la memoria e lo termina, riducendo piano piano l'utilizzo della risorsa. (*Linux*)

2 Concurrency

2.1 Threads e locks

Un **thread** è un sottoinsieme delle istruzioni di un processo, che può essere eseguito in maniera concorrente con altre parti di esso. L'obiettivo dei threads è quello di rendere più veloce l'esecuzione di un processo. Un programma multi-thread ha più punti di esecuzione (molteplici PCs, da ognuno dei quali vengono prelevate ed eseguite istruzioni). Possono essere visti come processi separati che **condividono lo stesso address space**. Ogni thread ha il proprio insieme privato di registri. Se ci sono due threads in esecuzione su un singolo processore, per switchare da T1 a T2 deve avvenire un context switch. Invece che il PCB avremo bisogno di un **Thread Control Blocks** (TCBs) per memorizzare lo stato di ogni thread di un processo. La differenza principale tra thread switch e context switch è che nel primo caso l'address space rimane lo stesso.
Un altra grande differenza tra threads e processi riguarda lo stack. In un processo multi-thread, ogni thread è indipendente e potrebbe chiamare varie routines. Invece di un singolo stack nell'address space ce ne sarà uno per thread (**thread-local storage**).

Utilizzare i thread abilita la sovrapposizione dell'I/O con altre attività all'interno di un singolo programma.

2.1.1 Thread creation

Vogliamo creare un programma che generi due threads. Ogni threads eseguirà la funzione `mythread()` con argomenti diversi (stringa A o B). Una volta che un thread viene creato, potrebbe venire eseguito subito (dipende dallo scheduler) o essere messo in stato di ready.

```
1  #include <stdio.h>
2  #include <assert.h>
3  #include <pthread.h>
4
5  void *mythread(void *arg) {
6    printf("%s\n", (char *) arg);
7    return NULL;
8  }
9
10
```

```
11 int main(int argc, char *argv[]) {
12   pthread_t p1, p2;
13   int rc;
14   printf("main: begin\n");
15   rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc==0);
16   rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc==0);
17   // join waits for the threads to finish
18   rc = pthread_join(p1, NULL); assert(rc==0);
19   rc = pthread_join(p2, NULL); assert(rc==0);
20   printf("main: end\n");
21   return 0;
22 }
```

Il thread principale chiama `pthread_join()` che aspetta il completamento di un particolare thread. L'ordine in cui viene eseguito il programma, dipende solo dallo scheduler.

2.1.2 Dati condivisi

La **race condition** consiste nell'avere più threads che concorrono all'uso della stessa risorsa. Il risultato della computazione **non è deterministico** in quanto dipende esclusivamente dalle decisioni dello scheduler e da quanto siamo fortunati con i timer interrupts. Una **sezione critica** è un pezzo di codice che accede alle variabili condivise e non deve essere eseguita simultaneamente da più thread. Abbiamo bisogno della **mutua esclusione**, la quale garantisce che, se un thread è in esecuzione in sezione critica, agli altri verrà proibilito l'accesso.

2.1.3 Atomicità

Un modo per risolvere il problema potrebbe essere quello di avere istruzioni più potenti che, in un singolo passo, facciano esattamente ciò di cui abbiamo bisogno. In questo caso è l'hardware a garantire l'atomicità, tramite delle **synchronization primitives**.

2.1.4 Thread creation

```
1 #include <pthread.h>
2 int pthread_create( pthread_t *thread, const pthread_attr_t *
    attr, void * (*start_routine) (void*), void * arg );
```

- `pthread_t *thread` è un puntatore a una struttura di tipo `pthread_t` che useremo per interagire con i thread.
- `attr` viene usato per specificare ogni tipo di attributo che il thread potrebbe avere. (*Grandezza stack, informazioni riguardanti priorità di scheduling, ...*)
- Il terzo argomento è un **puntatore a funzione** e consiste nel nome della funzione che vogliamo far eseguire al thread creato.
- `arg` è l'argomento che deve essere passato alla funzione che il thread deve eseguire.

I puntatori sono `void` perchè permettono di passare ogni tipo di argomento facendo semplicemente un cast.

2.1.5 Thread completion

Se vogliamo aspettare il completamento di un thread, dobbiamo chiamare la routine `pthread_join()`

```
#include <pthread.h>
int pthread_join( pthread_t thread, void **value_ptr );
```

- `thread` è usato per specificare quale thread stiamo aspettando.
- `**value_ptr` è un puntatore al valore di ritorno.

2.2 Locks

I lock vengono usati per introdurre la **mutua esclusione**, permettendo quindi di eseguire atomicamente la sezione critica. Per usare un lock, basta aggiungere il codice necessario attorno alla sezione critica come segue:

```
lock_t mutex; //lock allocato globalmente
...
lock(&mutex);
x = x + 1;    //sezione critica
unlock(&mutex);
```

Un lock è una variabile, e come tale va **dichiarata e inizializzata**. Un lock, ad un certo istante di tempo, può trovarsi in due stati: **disponibile** o **acquisito**. Il funzionamento è questo:

- viene chiamata la routime `lock()` per acquisire il lock.
- Se disponibile, il thread chiamante riceve il lock e può entrare in sezione critica. Se acquisito, il thread chiamante rimarrà bloccato nella routine `lock()` fino a quando il thread in sezione critica non termina e invoca la `unlock()`.
- Una volta acquisito il lock, un thread può operare in sezione critica.

Il nome della libreria `POSIX` per un lock è **mutex**. È possibile proteggere la sezione critica con un unico grande lock (**coarse-grained**) ma è possibile usare svariati lock (**fine-grained**). Mediante il lock, il programmatore guadagna un po' di **controllo sullo scheduler**. I locks però devono avere le seguenti proprietà:

- **Correctnes.** Garantire la mutua esclusione.
- **Fairness.** Evitare che i thread vadano in starvation. Tuttu devono poter accedere alla sezione critica prima o poi.
- **Performance.** L'overhead di time dovuto all'introduzione dei lock non deve minare le performance.

Per progettare un lock funzionante, abbiamo bisogno di aiuto da parte dell'hardware e dal sistema operativo.

2.2.1 Controlling interrupts

Se gli interrupt vengono disabilitati prima di entrare in sezione critica e riabilitati dopo, abbiamo implementato un rudimentale lock.

```
1 void lock() { DisableInterrupts(); }
2 void unlock() { EnableInterrupts(); }
```

Vantaggi

- In intel `CLI` e `STI` sono già presenti nell'ISA del processore, non dobbiamo quindi apportare modifiche dal punto di vista hardware.
- Soluzione molto semplice.

Svantaggi

- È necessario avere fiducia nei threads (`CLI` e `STI` sono istruzioni privilegiate). Un loop infinito in sezione critica potrebbe causare una catastrofe.
- Non funziona bene in presenza di più processori, visto che non siamo in grado di disabilitare gli interrupts su tutte le CPU.
- Disabilitare gli interrupts per periodi di tempo troppo estesi può portare alla perdita di alcuni di essi. Ad esempio, la CPU può perdersi il fatto che il disco ha comunicato di aver terminato la read request.
- Inefficienza.

2.2.2 Load e Store

```
1  typedef struct __lock_t { int flag; } lock_t;
2
3  void init(lock_t *mutex) {
4    // 0 -> lock is available, 1 -> held
5    mutex->flag = 0;
6  }
7
8  void lock(lock_t *mutex) {
9    while (mutex->flag == 1) // TEST the flag
10     ; // spin-wait (do nothing)
11   mutex->flag = 1; // now SET it!
12 }
13
14 void unlock(lock_t *mutex) {
15   mutex->flag = 0;
16 }
```

Viene usata una flag per indicare se un thread è in possesso del lock. Il primo thread che entrerà in sezione critica chiamerà la routine `lock()`, la quale controllerà il valore di flag e la setterà a 1 nel caso in cui essa sia uguale a 0 (a indicare che il thread è ora in possesso del lock) o farà spin-lock in caso contrario. Una volta finito la sezione critica, il thread chiama la `unlock()` e pulisce flag (rilasciando il lock). Se un altro thread chiamasse la `lock()` mentre il primo è in sezione critica, esso farà **spin-wait** nel ciclo while fino a quando non verrà chiamata la `unlock()`. Ci sono però due problemi:

- **Correctness.** È possibile che entrambi i thread settino flag a 1 ed entrino in sezione critica. Se `T1` vede il lock a 0 e prima di settarlo si ha un interrput, sia `T2` che `T1` setteranno il flag a 1 ed entreranno in sezione critica.
- **Performance.** Questo ciclio è chiamato ciclio di busy waiting o **spin-lock**. Il thread è in uno stato di attesa che mantiene occupato il processore, vengono quindi sprecati cicli di CPU

2.2.3 Test and Set

Viene implementato il supporto hardware con l'istruzione `TestAndSet`:

```
int TestAndSet( int *old_ptr, int new) {
  int old = *old_ptr; //prelevo vecchio valore di ptr
  olt_ptr = new;      //inserisco "new" in old_ptr
  return old;         //restituisco il vecchio valore
}
```

Restituisce il vecchio valore puntato da `ptr` e lo aggiorna a `new`. La chiave di questo meccanismo è che questa sequenza di operazioni viene eseguita **atomicamente**.

```
typedef struct __lock_t {
  int flag;
} lock_t;

void init(lock_t *lock) {
  // 0 indicates that lock is available, 1 that it is held
  lock->flag = 0;
}

void lock(lock_t *lock) {
  while (TestAndSet(&lock->flag, 1) == 1)
    ; // spin-wait (do nothing)
}

void unlock(lock_t *lock) {
  lock->flag = 0;
}
```

Con questo tipo di lock, siamo sicuri che un singolo thread potrà acquisire il lock ed entrare in sezione critica. Per funzionare correttamente su un singolo processore, ha bisogno di un **preemptive scheduler** (situazione per cui un processo viene temporaneamente interrotto e portato fuori dalla CPU), ad

esempio che interromperà un thread attraverso un timer. Senza, non ha senso siccome un thread in spinning non potrà mai rinunciare al lock.

2.2.4 Algoritmo di Peterson

L'idea è di garantire che due thread non entrino mai in sezione critica allo stesso tempo.

```
1  int flag[2];
2  int turn;
3
4  void init() {
5    flag[0] = flag[1] = 0;  // 1->thread wants to grab lock
6    turn = 0;               // whose turn? (thread 0 or 1?)
7  }
8
9  void lock() {
10   flag[self] = 1;    // self: thread ID of caller
11   turn = 1 - self;   // make it other threads turn
12   while ((flag[1-self] == 1) && (turn == 1 - self))
13     ;                // spin-wait
14 }
15
16 void unlock() {
17   flag[self] = 0;    // simply undo your intent
18 }
```

Se una cella di `flag` viene settata a 1 indica che il corrispondente thread desidera entrare in sezione critica. `turn` indica il turno del thread per entrare in sezione critica.

2.2.5 Spin locks

Vantaggi

- Semplicità (poche righe di codice).
- **Correctness.** Fornisce la mutua esclusione correttamente.

Svantaggi

- **Fairness** Non siamo in grado di garantire che ogni thread entrerà in sezione critica.

- **Performance** In una CPU monoprocessore lo spin lock è molto costoso. Quando abbiamo N threads a contendersi il lock, nel caso peggiore verranno sprecato $N - 1$ fette di tempo. In altri casi funziona ragionevolmente (*se il numero di threads è più o meno uguale al numero dei processori*).

2.2.6 Soluzioni

Dare la precedenza ad altri thread invece che fare spinlock.

```
1  void init() {
2    flag = 0;
3  }
4
5  void lock() {
6    while (TestAndSet(&flag, 1) == 1)
7      yield();  //rilascia la cpu
8  }
9
10 void unlock() {
11   flag = 0;
12 }
```

La primitiva `yield()` è una semplice **system call** che muove il chiamante dallo stato di running a quello di ready (*deschedula il thread*). Con due thread funziona bene ma con tanti che si contendono la sezione critica no. Se un thread acquisisce la CPU e viene interrotto prima di chiamare la `unlock()`, tutti gli altri chiameranno la `lock()`, troveranno il lock occupato e chiameranno la `yield()`. Oltre al problema della **starvation** abbiamo anche problemi di **performance**.

Sleeping instead of spinning Lo scheduler viene lasciato troppo al caso. `Solaris` mette a disposizione due routines:

- `park()` per mettere un thread chiamante in stato di sleep.
- `unpark(threadID)` per svegliare un particolare thread.

Queste due routines possono essere usate per costruire un lock che mette il chiamante a dormire se il lock è già acquisito e lo sveglia quando è disponibile. Per evitare la starvation si usa una **coda** per controllare chi è il prossimo a prendere il lock. Inoltre, viene usata una variabile `guard` per fare spin-lock

attorno a `flag` e manipolare la coda in uso dal lock. Un thread potrebbe comunque essere interrotto durante l'acquisizione o il rilascio del lock, causando gli altri thread a fare spin-wait ancora una volta. Tuttavia, il tempo speso a fare spinning è limitato (solo poche istruzioni all'interno del codice di `lock` e `unlock`.

```
typedef struct __lock_t {
  int flag;
  int guard;
  queue_t *q;
} lock_t;

void lock_init(lock_t *m) {
  m->flag = 0;
  m->guard = 0;
  queue_init(m->q);
}

void lock(lock_t *m) {
  while (TestAndSet(&m->guard, 1) == 1)
    ; //acquire guard lock by spinning
  if (m->flag == 0) {
    m->flag = 1; // lock is acquired
    m->guard = 0;
  } else {
    queue_add(m->q, gettid());
    m->guard = 0;
    park();
  }
}

void unlock(lock_t *m) {
  while (TestAndSet(&m->guard, 1) == 1)
    ; //acquire guard lock by spinning
  if (queue_empty(m->q))
    m->flag = 0; // let go of lock; no one wants it
  else
    unpark(queue_remove(m->q)); // hold lock (for next thread!)
  m->guard = 0;
}
```

Se `guard` fosse dopo `park()` tutti i thread successivi avrebbero trovato `guard` a 1, andando in spin-lock. conseguentemente, lo scheduler sceglie i threads in modo da garantire che non avvenga nessun deadlock. `flag` non viene settata nuovamente a 0 quando un altro thread viene svegliato perchè

passiamo il lock direttamente dal thread che lo rilascia al prossimo che lo acquisisce.

È Possibile che un thread sul punto di fare `park()` venga switchato al thread in possesso del lock e che quest'ultimo lo rilasci. Ciò potrebbe portare allo sleep permanente del primo thread (**waiting race**). Si può risolvere con una terza chiamata a `setpark()` (introdotto da `Solaris`) che serve per indicare che un thread è in procito di fare `park()`. Se quindi dovesse avvenire un thread switch e un altro thread chiamasse `unpark()` prima che `park()` sia effettivamente chiamata, la successiva `park()` ritorna immediatamente invece di dormire.

```
queue_add(m->q, gettid());
setpark();
m->guard = 0;
```

2.3 Condition Variables

Ci sono molti casi in cui un thread desidera controllare quando una condizione è vera prima di continuare la propria esecuzione. Per esempio un thread genitore potrebbe voler attendere il completamento del figlio prima di continuare (`join()`).

```
volatile int done = 0;

void *child(void *arg) {
  printf("child\n");
  done = 1;
  return NULL;
}

int main(int argc, char *argv[]) {
  printf("parent: begin\n");
  pthread_t c;
  Pthread_create(&c, NULL, child, NULL); // create child
  while (done == 0)
    ; // spin
  printf("parent: end\n");
  return 0;
}
```

Usare variabili condivise funziona ma è molto inefficiente, siccome il genitore spreca tempo di CPU a fare spin. Una **condition variable** è una coda esplicita in cui i threads possono mettersi quando la condizione di esecuzione

non è quella desiderata. Quando lo stato cambia, il thread (uno o più) viene svegliato e può quindi riprendere la propria esecuzione. Una condition variable ha associate due operazioni:

- `wait()` Mette in sleep un thread.
- `signal()` Viene usata quando un thread ha cambiato qualcosa nel programma e vuole quindi svegliarne uno in sleep che aspettava il verificarsi di quella condizione.

La `pthread_cond_wait( pthread_cond_t *c, pthread_mutex_t *m)` prende anche un mutex come parametro. Si assume che questo mutex sia **locked** quando la `wait()` viene invocata. La responsabilità della wait è di liberare il lock e mettere il thread chiamante in stato di sleep (atomicamente). Quando un thread viene svegliato, deve acquisire nuovamente il lock prima di ritornare dalla `wait()` (per prevenire la race condition). Una soluzione al problema del `join()`:

```
1  int done = 0;
2  pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
3  pthread_cond_t c = PTHREAD_COND_INITIALIZER;
4
5  void thr_exit() {
6    Pthread_mutex_lock(&m);
7    done = 1;
8    Pthread_cond_signal(&c);
9    Pthread_mutex_unlock(&m);
10 }
11
12 void *child(void *arg) {
13   printf("child\n");
14   thr_exit();
15   return NULL;
16 }
17
18 void thr_join() {
19   Pthread_mutex_lock(&m);
20   while (done == 0)
21     Pthread_cond_wait(&c, &m);
22   Pthread_mutex_unlock(&m);
23 }
24
25 int main(int argc, char *argv[]) {
26   printf("parent: begin\n");
```

```
27   pthread_t p;
28   Pthread_create(&p, NULL, child, NULL);
29   thr_join();
30   printf("parent: end\n");
31   return 0;
32 }
```

2.3.1 Approcci sbagliati

Senza la variabile done se il figlio viene eseguito immediatamente e chiama la `thr_exit()`, essa chiamerà la `signal()`, ma non ci sono thread in sleep sulla condizione. Quando il genitore verrà eseguito chiamerà la `wait()` e rimarrà bloccato, nessun thread lo sveglierà mai.

Senza il lock se il genitore chiama `thr_join()` e controlla il valore di done, vedrà che è a zero e si metterà in sleep. Prima di chiamare la wait viene interrotto e viene eseguito il figlio. Quest'ultimo cambia `done` a 1 e chiama `signal()` ma non c'è nessun thread in attesa del verificarsi della condizione. Quando il genitore viene nuovamente eseguito, andrà a dormire per sempre.

Vanno usati sempre i **cicli while** per i controlli.

2.3.2 Produttore e consumatore

I thread produttori generano i dati e li inseriscono in un buffer, i consumatori prendono questi dati dal buffer e li consumano in un certo modo. Il buffer è una risorsa condivisa ed è necessario sincronizzare l'acesso ad essa per evitare la race condition.

```
1  int buffer;
2  int count = 0; // initially, empty
3
4  void put(int value) {
5    assert(count == 0);
6    count = 1;
7    buffer = value;
8  }
9
10 int get() {
11   assert(count == 1);
12   count = 0;
```

```
13   return buffer;
14 }
```

- `put()`, assumendo che il buffer sia vuoto, inserisce semplicemente un valore in esso e lo marca come "pieno" settando la variabile `count`.
- `get()` fa l'opposto, settando il buffer a vuoto (`count = 0`) e restituisce il valore prelevato.

```
1  cond_t cond;
2  mutex_t mutex;
3
4  void *producer(void *arg) {
5    int i;
6    for (i = 0; i < loops; i++) {
7      Pthread_mutex_lock(&mutex);
8      if (count == 1)      //questo verra' cambiato con un while
9        Pthread_cond_wait(&cond, &mutex);
10     put(i);
11     Pthread_cond_signal(&cond);
12     Pthread_mutex_unlock(&mutex);
13   }
14 }
15
16 void *consumer(void *arg) {
17   int i;
18   for (i = 0; i < loops; i++) {
19     Pthread_mutex_lock(&mutex);
20     if (count == 0)      //questo verra' cambiato con un while
21       Pthread_cond_wait(&cond, &mutex);
22     int tmp = get();
23     Pthread_cond_signal(&cond);
24     Pthread_mutex_unlock(&mutex);
25     printf("%d\n", tmp);
26   }
27 }
```

Con un singolo produttore e un singolo consumatore, il codice sopra funziona, ma se abbiamo ad esempio due consumatori, la soluzione due problemi critici:

- Supponiamo che ci siano due consumatori e un produttore:
  - C_1 acquisisce il lock e controlla che ogni buffer sia pronto per la consumazione ma non ce ne sono, chiama la `wait`.

  - P_1 viene eseguito, acquisisce il lock e controlla che tutti buffer siano pieni, ma non lo sono. Va avanti riempendo il buffer. Invoca la `signal()` per informare che il buffer è stato riempito.
  - C_1 si muove nella coda di ready dallo stato di sleeping sulla condition variable (*non viene ancora eseguito*).
  - P_1 continua fino a quando non realizza che il buffer è pieno, e poi va in sleep.
  - C_2 viene eseguito e consuma il valore nel buffer (salta la `wait` perchè il buffer è pieno).
  - C_1 viene ora eseguito, prima di ritornare dalla `wait()`, acquisisce nuovamente il lock, chiama la `get()` ma **non ci sono buffer da consumare**.

    Il problema è questo: dopo che il produttore sveglia C_1, ma prima che esso venga eseguito, lo stato del buffer è cambiato (per colpa di C_2). Segnalare un thread lo sveglia solamente dicendogli che lo stato è cambiato (in questo caso che un valore è stato messo nel buffer), ma non ci sono garanzie che quando il thread svegliato venga eseguito lo stato sia lo stesso (**Mesa semantics**).

    Soluzione: cambiare l'**if** in **while** in modo che se C_1 viene svegliato ricontrolla immediatamente lo stato della variabile condivisa. Se il buffer è vuoto, tornerà semplicemente a dormire.

- C'è una sola condition variable.
  - Vengono eseguiti prima C_1 e C_2 e vanno in sleep.
  - Il produttore mette un valore nel buffer e sveglia uno dei consumatori, diciamo C_1.
  - Il produttore torna indietro e prova a inserire più dati nel buffer, siccome è pieno il produttore chiamerà `wait()` e andrà a dormire.
  - C_1 è pronto per essere eseguito e **due threads stanno dormendo sulla condizione** (C_2 e P_1).
  - C_1 Si sveglia ritornando dalla `wait()`, controlla nuovamente la condizione e trova che il buffer è pieno. Consuma il valore e segnala la condizione, svegliando un thread in sleeping. **Quale dei due sveglia?** Se svegliasse il consumatore, troverà il buffer vuoto

e si metterà in sleep. Il produttore viene lasciato a dormire. Tutti e tre i threads sono in stato di sleeping. Un **consumatore non dovrebbe poter svegliare altri consumatori**, ma solo produttori (e viceversa).

La soluzione è usare due condition variables per segnalare correttamente quale tipo di thread andrebbe svegliato.

```
1  int buffer[MAX];
2  int fill = 0;
3  int use = 0;
4  int count = 0;
5
6  void put(int value) {
7    buffer[fill] = value;
8    fill = (fill + 1) % MAX;
9    count++;
10 }
11
12 int get() {
13   int tmp = buffer[use];
14   use = (use + 1) % MAX;
15   count--;
16   return tmp;
17 }
18
19 cond_t empty, fill;
20 mutex_t mutex;
21
22 void *producer(void *arg) {
23   int i;
24   for (i = 0; i < loops; i++) {
25     Pthread_mutex_lock(&mutex);
26     while (count == MAX)
27       Pthread_cond_wait(&empty, &mutex);
28     put(i);
29     Pthread_cond_signal(&fill);
30     Pthread_mutex_unlock(&mutex);
31   }
32 }
33
34 void *consumer(void *arg) {
35   int i;
36   for (i = 0; i < loops; i++) {
```

```
37      Pthread_mutex_lock(&mutex);
38      while (count == 0)
39        Pthread_cond_wait(&fill, &mutex);
40      int tmp = get();
41      Pthread_cond_signal(&empty);
42      Pthread_mutex_unlock(&mutex);
43      printf("%d\n", tmp);
44    }
45  }
```

I thread produttori aspettano sulla condizione `empty` e segnalano `fill`, per i consumatori l'inverso. Inoltre per abilitare più concorrenza ed efficienza si aggiungono più slot al buffer, in modo tale che più valori possano essere prodotti o consumati prima di andare in sleep. Un produttore dorme solo se tutti i buffer sono pieni, un consumatore dorme solo se tutti i buffer sono vuoti.

2.4 Semafori

Un semaforo è un oggetto con un valore di tipo integer utilizzabili sia come locks che come condition variables.
Il valore iniziale di un semaforo ne determina il comportamento.

```
1 #include <semaphore.h>
2 sem_t s;
3 sem_init(&s, 0, 1);
```

Il secondo valore di `sem_init()` viene settato a 0 e indica che il semaforo è condiviso tra thread dello stesso processo. Il terzo argomento è il valore con cui lo si inizializza. Per interagire con esso vengono introdotte due routines:

- `sem_wait(sem_t *s)` decrementa il valore del semaforo `s` di uno e aspetta se il valore del semaforo è negativo.
- `sem_post(sem_t *s)` incrementa il valore del semaforo `s` di uno, se ci sono uno o più threads in attesa ne sveglia uno.

Il valore del semaforo, quando negativo, è uguale al numero di thread in attesa.

2.4.1 Semafori binari: locks

Per implementare i lock, il codice sarà:

```
sem_t m;
sem_init(&m, 0, 1); //inizializza il semaforo a 1
sem_wait(&m);
//sezione critica
sem_post(&m);
```

Il valore di partenza del semaforo `m` a 1 è critico per la realizzazione del lock. I semafori **binari** possono trovarsi solamente in due stati: acquisito o disponibile.

2.4.2 Semafori per ordinare

I semafori possono essere usati anche come condition variables, esempio:

```
sem_t s;

void *child(void *arg) {
  printf("child\n");
  sem_post(&s); // signal here: child is done
  return NULL;
}

int main(int argc, char *argv[]) {
  sem_init(&s, 0, 0); //Inizializzo a 0 per forza
  printf("parent: begin\n");
  pthread_t c;
  Pthread_create(c, NULL, child, NULL);
  sem_wait(&s); // wait here for child
  printf("parent: end\n");
  return 0;
}
```

In questo modo ci aspettiamo di avere come risultato: `parent: begin, child, parent: end`

2.4.3 Semafori come produttore e consumatore

Si usano due semafori, `empty` e `full`, che indicano quando una entry del buffer è stata svuotata o riempita rispettivamente.

```
int buffer[MAX];
int fill = 0;
int use = 0;

void put(int value) {
```

```
6   buffer[fill] = value;
7   fill = (fill + 1) % MAX;
8 }
9
10 int get() {
11   int tmp = buffer[use];
12   use = (use + 1) % MAX;
13   return tmp;
14 }
```

Produttori e consumatori:

```
1 sem_t empty;
2 sem_t full;
3
4 void *producer(void *arg) {
5   int i;
6   for (i = 0; i < loops; i++) {
7     sem_wait(&empty);
8     put(i);
9     sem_post(&full);
10  }
11 }
12
13 void *consumer(void *arg) {
14  int i, tmp = 0;
15  while (tmp != -1) {
16    sem_wait(&full);
17    tmp = get();
18    sem_post(&empty);
19    printf("%d\n", tmp);
20  }
21 }
22
23 int main(int argc, char *argv[]) {
24 // ...
25 sem_init(&empty, 0, MAX); //MAX buffers are empty to begin with
26 sem_init(&full, 0, 0); // ... and 0 are full
27 // ...
28 }
```

Il produttore aspetta che il buffer diventi vuoto per poterlo riempire di dati e il consumatore attende che il buffer sia pieno prima di consumare i dati. Con un singolo consumatore e un produttore funziona bene, con `MAX = 10` abbiamo un problema di **race condition**:

- Abbiamo due produttori in procinto di chiamare la `put()`

- P_1 viene eseguito prima e inizia a riempire il buffer. Prima che esso incrementi `fill` a 1, viene interrotto.
- Il produttore P_2 inizia ad essere eseguito e anche lui, inserisce i suoi dati nello stesso punto del buffer sovrascrivendo quelli vecchi.

Manca la **mutua esclusione**, riempire il buffer e incrementare il contatore è una porzione di codice critica. Basta aggiungere i semafori binari come locks. Inizialmente può sembrare una buona idea metterli intorno a `wait` e `post`, ma questo potrebbe portare a un **deadlock**.

- Il consumatore viene eseguito e acquisisce mutex.
- Chiama la `sem_wait(&full)`. Possiede ancora il lock.
- Viene eseguito un produttore che, se fosse possibile, riempirebbe il buffer di dati e sveglierebbe il consumatore. Chiama `sem_wait(&mutex)`. Il lock è già in possesso del consumatore e il produttore è bloccato ad aspettare. Il consumatore ha il lock ed è in waiting, produttore non ha il lock, ed è in waiting.

Si sposta quindi il mutex intorno solo alla `get()` e alla `put()` e si ottiene il seguente codice:

```
1  sem_t empty;
2  sem_t full;
3  sem_t mutex;
4
5  void *producer(void *arg) {
6    int i;
7    for (i = 0; i < loops; i++) {
8      sem_wait(&empty);
9      sem_wait(&mutex); //(MOVED MUTEX HERE...)
10     put(i);
11     sem_post(&mutex); //(... AND HERE)
12     sem_post(&full);
13   }
14 }
15
16 void *consumer(void *arg) {
17   int i;
18   for (i = 0; i < loops; i++) {
19     sem_wait(&full);
20     sem_wait(&mutex); //(MOVED MUTEX HERE...)
```

```
21      int tmp = get();
22      sem_post(&mutex); //(... AND HERE)
23      sem_post(&empty);
24      printf("%d\n", tmp);
25    }
26  }
27
28  int main(int argc, char *argv[]){
29    // ...
30    sem_init(&empty, 0, MAX);//MAX buffers are empty to begin with
31    sem_init(&full, 0, 0);   // ... and 0 are full
32    sem_init(&mutex, 0, 1);  // mutex=1 because it is a lock
33    // ...
34  }
```

2.4.4 Implementazione dei semafori

```
1   typedef struct __Zem_t {
2     int value;
3     pthread_cond_t cond;
4     pthread_mutex_t lock;
5   } Zem_t;
6
7   // only one thread can call this
8   void Zem_init(Zem_t *s, int value) {
9     s->value = value;
10    Cond_init(&s->cond);
11    Mutex_init(&s->lock);
12  }
13
14  void Zem_wait(Zem_t *s) {
15    Mutex_lock(&s->lock);
16    while (s->value <= 0)
17      Cond_wait(&s->cond, &s->lock);
18    s->value--;
19    Mutex_unlock(&s->lock);
20  }
21
22  void Zem_post(Zem_t *s) {
23    Mutex_lock(&s->lock);
24    s->value++;
25    Cond_signal(&s->cond);
26    Mutex_unlock(&s->lock);
27  }
```

2.5 Problemi relativi alla concorrenza

2.5.1 Non-deadlock bugs

Atomicity-violation bug consiste nel non mettere dei **locks** attorno ad una variabile condivisa.

Order-violation bug consiste nel non assegnare un ordine a due thread che accedono alla stessa variabile. Ad esempio, un thread potrebbe dare per scontato che una variabile sia già inizializzata ed utilizzarla, ma se è ancora a `NULL` questo causerà un crash. Per risolvere questo bug, basta usare le **condition variables**.

2.5.2 Deadlock bugs

I deadlock avvengono quando un thread T_1 è in possesso del lock L_1 e in attesa di un altro L_2; il thread T_2 che possiede il lock L_2 è in attesa che L_1 venga rilasciato.

```
1 Thread 1:                    Thread 2:
2 pthread_mutex_lock(L1);      pthread_mutex_lock(L2);
3 pthread_mutex_lock(L2);      pthread_mutex_lock(L1);
```

Le ragioni per cui i deadlock avvengono possono essere:

- Troppo codice
- Dipendenze complesse tra i componenti.
- L'incapsulamento del codice.

Affinchè avvenga un deadlock devono essere valide quattro condizioni:

- **Mutua esclusione** I threads richiedono il controllo escluso delle risorse che acquisiscono.
- **Hold-and-wait** I thread tengono le risorse allocate da essi (ad esempio i locks che hanno già acquisito) finchè aspettano risorse addizionali (ad esempio i lock che vogliono acquisire).
- **No preemption** Le risorse (ad esempio i locks) non possono essere rimosse forzatamente dai threads che le stanno tenendo.

- **Circular wait** Esistono catene circolari di threads.

Se una qualunque di queste condizioni non è soddisfatta, un deadlock non può avvenire.

2.5.3 Prevenzione dei deadlocks

- **Circular wait** La miglior tecnica di prevenzione è fornire un acquisizione del lock ordinata. Per esempio, possiamo prevenire il deadlock acquisendo sempre L_1 prima di L_2. Questo ordine consente di evitare wait cicliche e quindi deadlock.

- **Hold-and-wait** Per evitare il deadlock dovuto a questa condizione basta acquisire tutti i locks in una volta atomicamente, in modo da evitare che non ci siano thread switch prematuri nel mezzo dell'acquisizione del lock. La soluzione però è problematica: richiede di sapere esattamente quali locks devono essere posseduti e di acquisirli tutti in una volta. Tutti i locks devono essere acquisiti in una volta anche se non è necessario possederli tutti.

- **No preemption** Per non dare tutti i locks come acquisiti fino a quando la `unlock()` non viene invocata, si può usare una routine come `pthread_mutex_trylock()` che prende il lock (se disponibile) e ritorna **success** o un codice di errore se il lock è già posseduto. Questo per evitare l'acquisizione multipla di locks, perchè potremmo aspettarne uno mentre siamo in possesso di un altro. Sorge un nuovo problema, la **livelock**.

```
1 top:
2 pthread_mutex_lock(L1);
3 if(pthread_mutex_trylock(L2) != 0) {
4   pthread_mutex_unlock(L1);
5   goto top;
6 }
```

  È possibile che due threads tentino ripetutamente questa sequenza e falliscano nell'acquisire il lock. Si può risolvere con un delay random prima di fare il loop back.

- **Mutua esclusione** Si possono utilizzare itruzioni hardware per evitare di usare i locks, in questo modo non sarebbe possibile che si verifichino deadlocks (livelock può comunque accadere).

2.5.4 Algoritmo del banchiere

Si potrebbe non eseguire concorrentemente threads che potrebbero causare deadlock. Un approccio che utilizza questa tattica è l'algoritmo banchiere, che permette di gestire istanze multiple di una risorsa.

Sua n il numero di processi e m il numero di tipi di risorse, abbiamo che:

- `disponibili[j] = k` se ci sono k istanze disponibile del tipo di risorsa R_j. Vettore lungo m.
- `massimo[i,j] = k` se il processo P_i può richiedere al più k istanze del tipo di risorsa R_j. Matrice $n \times m$.
- `assegnate[i,j] = k` se a P_i sono attualmente allocate k istanze di R_j. Matrice $n \times m$.
- `necessità[i,j] = k` se P_i può richiedere k ulteriori istanze di R_j per completare il proprio task.

`necessità[i,j] = massimo[i,j] - assegnate[i,j]`

```
lavoro[m]
fine[n]
lavoro = disponibili

for i = 0 to n:
  fine[i] = false

back:
if si trova i tale che fine[i] = false & richiesta[i] <= lavoro:
  lavoro = lavoro + assegnate[i]
  fine[i] = true
  goto back

for i = 0 to n:
  if fine[i] = false:
    il processo i e' in deadlock

il sistema e' in stato sicuro.
```

Se i deadlock sono rari, viene fatto un reboot e basta. Periodicamente un deadlock detector viene eseguito e costruisce un grafico delle risorse e lo controlla per individuare i cicli. Se accade un deadlock, il sistema ha bisogno di essere fatto ripartire.

3 Persistence

4 JOS