**SISTEMI OPERATIVI**

**Capitolo 1-4**

Un sistema operativo deve soddisfare tre requisiti per garantire efficacia durante l’elaborazione:

* *uso efficiente*: assicurare l’uso efficiente delle risorse del pc;
* *convenienza per l’utente*: fornire metodi convenienti per usare il pc;
* *assenza di interferenze*: prevenire interferenze nelle attività dei suoi utenti.

Il SO soddisfa questi 3 requisiti eseguendo 3 funzioni primarie durante la sua esecuzione:

* gestione dei programmi;
* gestione delle risorse;
* sicurezza e protezione;

***Overhead***: costituisce l’impiego di risorse di CPU e memoria.

***Idle***: utilizzo delle risorse non efficiente (un programma che non sfrutta le risorse assegnategli).

Immagine che contiene testo, schermata, diagramma, Carattere

Descrizione generata automaticamente

Schema dello scheduling.

***Prelazione***: fase decisionale, in cui il SO sottrae la CPU al programma al termine del periodo di tempo specificato, con successivo reinserimento nella waiting list per la CPU.

Una buona politica di scheduling si ha quando ogni programma subisce un piccolo ritardo prima di avere accesso all’utilizzo della CPU. Quando questo non avviene e si prelaziona troppo tardi, l’overhead diminuisce ma i tempi di attesa per altri programmi aumenterebbero con conseguenti ritardi.

***Kernel***: è l’insieme di routine che costituiscono il nucleo del SO.

* Implementa le funzioni di controllo;
* Fornisce un insieme di servizi per i programmi utente;
* Si trova in memoria durante il funzionamento di un SO.

Un SO alterna l’esecuzione di diversi programmi utente da parte della CPU. La fine di un’operazione di I/O o la richiesta di una risorsa da parte di un programma, causa la generazione di un interrupt (software) nel sistema.

***Interrupt***: evento che dirotta la CPU verso l’esecuzione di codice kernel.

La CPU può operare in due modalità:

* ***Modalità kernel***: può eseguire tutte le istruzioni del pc, il kernel opera con la CPU in questa modalità in modo da controllare le operazioni del pc.
* ***Modalità utente***: non può eseguire quelle istruzioni che, se usate in modo improprio, possono potenzialmente interferire con altri programmi utente.

Immagine che contiene testo, diagramma, schermata, Piano

Descrizione generata automaticamente

Componenti della CPU visibili ai P.U. e al SO:

* ***registri GPR***: mantengono dati, indirizzi, valori, o stack pointer durante l’esecuzione di un programma.
* ***Registri PSW***: contengono l’informazione necessaria a controllare il funzionamento della CPU.

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

Informazioni che descrivono lo stato della CPU

***PC***: contiene l’indirizzo della prossima istruzione da eseguire.

***CC***: indica alcune caratteristiche del risultato di un’istruzione aritmetica. (< =, = 0, > 0)

***M***: indica il bit di modalità in cui è impostata la CPU (0 kernel o 1 utente).

***MPI***: contiene informazioni relative alla protezione della memoria del processo in esecuzione. Suddiviso in due campi, registro base e registro limite.

***IM***: indica quali interrupt sono abilitati e quali sono mascherati. Per interrupt mascherati si intende che solo appunto quelli mascherati sono consentiti in un dato momento. (ad esempio solo quelli > = intero m)

***IC***: descrive la condizione o l’evento che ha generato quell’interrupt.

Immagine che contiene testo, schermata, cerchio, Carattere

Descrizione generata automaticamente***Processo***: programma in esecuzione – unità di misura del lavoro di elaborazione.

***Commutazione di processo (context switch)***: fase in cui il SO prelaziona un processo per schedularne un altro.

***Thread***: programma in esecuzione che viene però eseguito nell’ambito di un processo, ovvero utilizza il codice, i dati e le risorse di un processo.

***Deadlock***: situazione in cui i processi attendono le azioni di altri processi per un tempo infinito.

Situazione di deadlock che coinvolge tre processi.

*Immagine che contiene testo, schermata, diagramma, Piano

Descrizione generata automaticamente***Memoria virtuale**

Consiste nella possibilità di memorizzare una quantità di dati maggiore della memoria fisica presente nel sistema. È possibile farlo memorizzando sul disco il codice e i dati di un processo, caricando in memoria solo alcune parti.

Per fare ciò il SO utilizza il modello di allocazione non contigua della memoria.

Durane l’esecuzione di un processo, la CPU passa l’indirizzo logico di ogni istruzione o dato utilizzato alla MMU, che determina gli indirizzi fisici in memoria dove sono memorizzati il codice e i dati.

Avviene quella che è chiamata traduzione dell’indirizzo (logico 🡪 fisico).

Se non sono presenti in memoria, la MMU genera un interrupt, e il SO carica la porzione di codice o dati mancanti, e viene ripristinata l’esecuzione del processo.

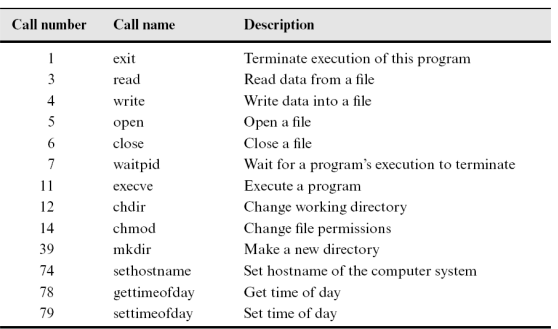
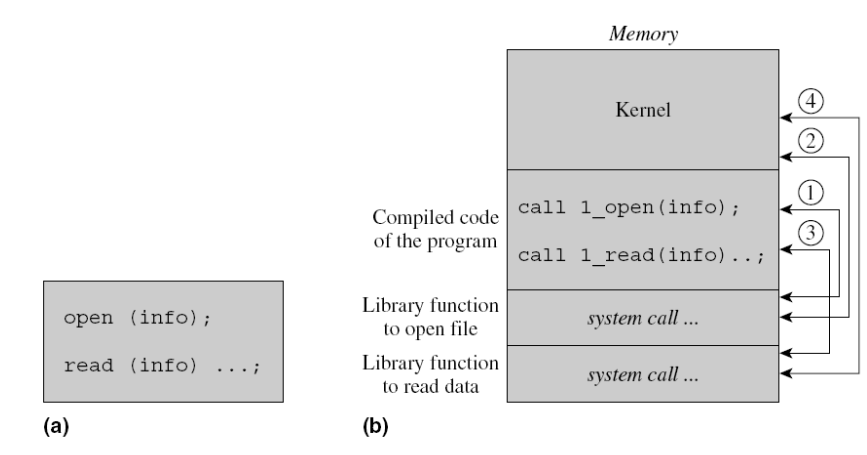
***Memoria gerarchica***: poiché le memorie veloci sono costose, la soluzione è quella di avere diverse unità di memoria con differenti velocità.

La CPU utilizza solo la memoria più veloce per accedere ai dati. Se non sono presenti, vengono copiati dalla memoria più lenta in quella più veloce, e resteranno lì finché non verrà rimosso per fare spazio ad altri dati.

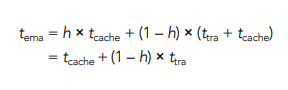
**EAT** (tempo effettivo di accesso alla memoria)

System call

Esempio di chiamata di sistema

**Per ogni dato/istruzione richiesto durante l’esecuzione di un programma, la CPU effettua una ricerca nella cache confrontando gli indirizzi dei byte richiesti con gli indirizzi dei byte nei blocchi di memoria presenti in cache.

Quando questi sono presenti in memoria si verifica un hit. In caso contrario un miss.

 *formula*

**tra**: trasferimento di un blocco dalla memoria alla cache.

**(1-h)**: probabilità che non si verifichi un hit

***Hit ratio***: è il rapporto tra hit e miss nella cache. Più è alto, più sarà efficiente la cache, in quanto avremo un numero ridotto di accessi alla memoria principale.

**La memoria**

La differenza sostanziale tra la memoria e la cache è che la gestione della memoria e il trasferimento dei blocchi tra mem e disco sono effettuati dal SW, mentre nella cache dall’HW.

Per il resto il funzionamento della memoria centrale è analogo a quello della cache, i blocchi di byte che vengono spostati tra il disco e la mem centrale qui vengono chiamate pagine.

***System call***: è una richiesta che un programma fa al kernel attraverso un interrupt SW.

***Throughput***: il numero di job, programmi, processi o sottorichieste completati da un sistema in una unità di tempo.

***Tempo di turnaround***: tempo di completamento di un job o di un processo.

***Tempo di risposta***: tempo trascorso dalla sottomissione di una sottorichiesta da parte di un utente fino al momento in cui il processo risponde ad essa.

**Sistemi di elaborazione batch**

Immagine che contiene testo, ricevuta, Carattere, linea

Descrizione generata automaticamente

Si utilizzavano le schede perforate per l’input. Un gruppo di schede equivaleva ad un job e i suoi dati.

Per ovviare a questo problema (perdita di tempo di CPU, in quanto venivano caricate da una persona manualmente), fu introdotta l’elaborazione batch, ovvero una sequenza di job identificati da speciali schede che ne indicavano l’inizio e la fine.

Il batching kernel veniva inizializzato da una persona, e cominciava l’elaborazione. Terminato un job iniziava il successivo, e così via. Quindi l’operatore interveniva solo all’inizio e alla fine del batch.

**Sistemi multiprogrammati**

**Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente**

Erano in grado di mantenere in memoria molti programmi utente.

Le caratteristiche principali che permettevano la multiprogrammazione erano il supporto del DMA, che gestiva autonomamente le richieste e le operazioni di I/O.

La protezione della memoria, per evitare interferenze di altri programmi (dati, istruzioni ed operazioni di I/O).

Infine, l’implementazione ed esecuzione di istruzioni privilegiate che possono essere eseguite solo in modalità kernel (passaggio dalla modalità utente, dedicata all’esecuzione di programmi utente).

Una delle necessità di questo SO era mantenere un numero sufficiente di programmi in memoria, facendo si che la CPU e i dispositivi di I/O abbiano lavoro sufficiente da effettuare.

Questo numero è definito come grado di multiprogrammazione.

Per fare ciò, il SO adotta due tecniche di multiprogrammazione:

* *Appropriato mix di programmi*: il kernel seleziona un mix di programmi CPU-bound e I/O-bound.
* *Scheduling a priorità con prelazione*: ad ogni programma è assegnata una priorità. La CPU verrà assegnata sempre a quello a priorità più alta (nella queue). Un programma con priorità più bassa verrà prelazionato per uno a priorità più alta.

**Sistemi time-sharing**

Progettati per fornire una risposta veloce, il cui risultato viene ottenuto condividendo il tempo di CPU tra i processi in modo tale che ogni processo al quale è stata fatta una sottorichiesta ottenga un tempo di CPU senza attendere troppo.

Il nome della tecnica utilizzata in questi sistemi è chiamato **scheduling round-robin con time slicing**.

Il funzionamento si basa sul mantenimento di una coda di scheduling dei processi che aspettano di poter utilizzare la CPU. Viene schedulato il processo in testa alla coda, quando:

* termina la sua esecuzione (completa tutte le operazioni che doveva fare)
* effettua una richiesta di I/O
* termina il time slice, ovvero il quanto di tempo specificato.

**Time slice**: la più grande porzione di tempo di CPU che ogni processo può utilizzare quando viene schedulato per essere eseguito dalla CPU.

Quindi, quando un processo supera il time slice, viene prelazionato e riportato alla fine della coda.

**Sistemi operativi real-time**

Definito come un programma che risponde alle attività in un sistema esterno entro un intervallo di tempo massimo determinato dal sistema esterno.

Si verifica un errore nel sistema esterno se l’applicazione impiega troppo tempo per rispondere ad un attività, nello specifico:

* il tempo di risposta non deve essere maggiore del requisito di risposta (il più grande valore di tempo per il quale il sistema può funzionare correttamente)
* l’evento/azione deve essere completata entro un certo periodo di tempo (deadline)

La differenza tra requisito di risposta e deadline è che la prima si riferisce al tempo limite entro il quale bisogna che la richiesta venga gestita, mentre la seconda indica entro quanto dev’essere completata.

**Sistemi operativi distribuiti**

Composto da diversi computer singoli connessi attraverso una rete, ciascuno dei quali potrebbe a loro volta essere un pc, un sistema multiprocessore o un cluster (gruppo di computer).

Tra i benefici principali c’è la condivisione delle risorse, che richiedono in ogni caso tecniche speciali per accedervi.

**Sistemi operativi moderni**

Contiene elementi di tutti gli ambienti di elaborazione classici, ed usa tecniche diverse per applicazioni diverse.

**Capitolo 5**

**Processo**

È un programma in esecuzione che utilizza le risorse a esso allocate.

Comprende sei componenti:

* ***ID***: è l’identificativo univoco assegnato dal SO.
* ***Codice***: è il codice del programma.
* ***Dati***: sono i dati usati durante l’esecuzione del programma.
* ***Stack***: contiene i parametri delle funzioni e delle procedure chiamate durante l’esecuzione del programma, compresi gli indirizzi di ritorno.
* ***Risorse***: insieme delle risorse allocate dal SO.
* ***Stato della CPU***: composto dal PSW e dai registri GPR della CPU.

Il SO non è a conoscenza della natura di un programma, per questo motivo non è in grado di stabilire se l’esecuzione di una funzione/procedura è sua volta un processo o meno.

Ciò viene stabilito dal programma stesso, che si occupa del resto dell’esecuzione (il SO si occupa solo di gestire le system call richiamate), e in questo modo le funzioni di un programma possono essere processi separati o essere parte di un singolo processo.

Quando il kernel avvia l’esecuzione di un programma genera un processo. Questo può a sua volta generare un processo figlio (e così via), dando vita ad un albero di processi la cui radice è il processo primario.

**Implementazione dei processi**

Le funzioni fondamentali del kernel per il controllo dei processi sono:

***Salvataggio del contesto***: consente di salvare lo stato della CPU e le informazioni riguardo le risorse del processo che è stato interrotto durante l’esecuzione.

***Gestione dell’evento***: l’ECB (event control block) analizza la condizione che ha portato all’interrupt, o all’eventuale richiesta da parte di un processo che ha portato a una system call per effettuare una determinata operazione.

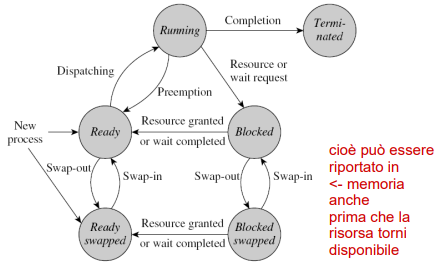
***Scheduling***: seleziona il prossimo processo a cui assegnare la CPU.

***Dispatching***: consente l’accesso alle risorse per il processo selezionato e salva lo stato della CPU per iniziare o proseguire l’esecuzione.

**Stati di un processo e transizione degli stati**

Un processo può avere diversi stati, ready, running, blocked e terminated.

Si ha una transizione di stato quando appunto un processo cambia il proprio stato passando ad esempio dallo stato di ready a quello di running.

***swapping***: consente di rimuovere temporaneamente un processo dalla memoria centrale.

Il kernel identifica un processo in due parti:

* Una costituita dal codice, dai dati, dallo stack. (contesto del processo)
* L’altra contenente le informazioni riguardanti l’esecuzione del programma, ovvero lo stato del processo, lo stato della CPU, il puntatore allo stack e altre informazioni. (PCB)

**Contesto del processo**

È formato da:

* Spazio di indirizzamento: codice dati e stack del processo.
* Informazioni di allocazione della memoria: informazioni relativa all’area di memoria allocata al processo.
* Stato delle attività di elaborazione del file: informazioni relative ai file usati.
* Informazioni di relazione con altri processi: ad esempio l’ID del processo genitore.
* Informazioni relative alle risorse: riguardante le risorse allocate al processo.
* Informazioni varie: relative al funzionamento di un processo.

Ogni processo ha uno spazio di indirizzamento diverso.

**PCB**

* **ID del processo**: id univoco assegnato dal SO.
* **ID genitore e dei figli**
* **Priorità**: valore numerico, assegnata al momento della creazione. Può cambiare dinamicamente.
* **Stato del processo**: stato corrente.
* **PSW**: program status word, è un immagine del PSW eseguita l’ultima volta che il processo è stato interrotto o prelazionato.
* **GPR**: contenuto dei registri general purpose salvati l’ultima volta che il processo è stato bloccato o prelazionato (definizione in alto).
* **Informazione sugli eventi**: esempio di un processo blocked, in questo campo ci sarà l’informazione relativa all’evento per il quale il processo è in attesa.
* **Informazione sui segnali**
* **Puntatore al PCB**

**ECB**

Contiene tre campi:

* Descrizione dell’evento
* ID del processo
* Puntatore all’ECB.

L’ECB permette di velocizzare la ricerca del processo a cui fa riferimento l’evento che si è verificato.

Il kernel crea al momento opportuno un ECB e inserisce al suo interno le informazioni relative all’evento e al processo. Quando si verifica un evento, il kernel controlla la lista degli ECB per trovare un ECB con la descrizione corrispondente, e il relativo processo collegato.

**Interazioni tra i processi**

I processi interagiscono tra di loro per diversi motivi:

* **condivisione dei dati**: per evitare inconsistenza di dati
* **scambio di messaggi**: i processi si scambiano informazioni inviando messaggi.
* **Sincronizzazione**: coordinamento delle attività
* **Segnali**: utilizzati per comunicare ad un processo l’occorrenza di una situazione di eccezione.

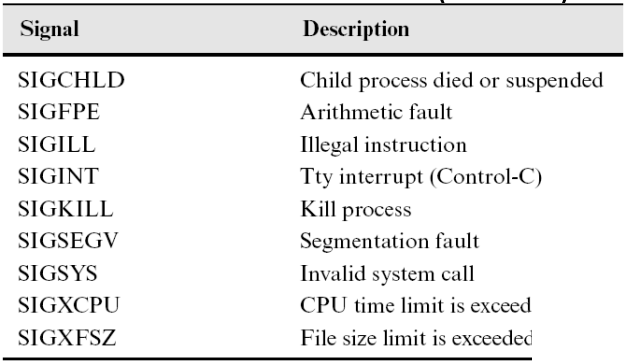
*Premessa: segnali e interrupt non sono la stessa cosa.*

**Segnali**

I segnali permettono la comunicazione tra i processi.

Ci sono varie system call associate ai segnali:

* **register handler**: chiamata di sistema, che un processo effettua per utilizzare i segnali, con la quale viene gestita una funzione non di sistema per uno specifico segnale. (signal handler)
* **signal handler**: routine da eseguire alla ricezione di uno specifico segnale. (altrimenti il kernel esegue un default handler)
* **kill**: invia un segnale ad un altro processo



Segnali in Unix

**Thread**

Esecuzione di un programma che usa le risorse di un processo.

Vengono implementati per ovviare al problema dell’overhead generato dalla commutazione tra i processi.

L’overhead di questo tipo ha due componenti:

* **overhead relativo all’esecuzione**: salvataggio dello stato della CPU del processo in esecuzione + caricamento dello stato della CPU del nuovo processo. (inevitabile)
* **overhead dovuto all’uso delle risorse**: anche il contesto del processo deve essere commutato.

Un thread viene creato da un processo mediante una system call.

**TCB**

Analogo al PCB, avente:

* thread ID, priorità e stato
* stato della CPU (GPR+PSW)
* puntatore al PCB genitore
* puntatore al TCB (liste)

Tra i vantaggi dei thread rispetto ai processi, oltre ad un overhead inferiore, abbiamo:

* Minore overhead dovuto alla creazione e alla commutazione.
* Comunicazione più efficiente: spazio di indirizzamento condiviso (genitore), quindi zero overhead per le chiamate di sistema.
* Progettazione semplificata

Per la gestione dei segnali, se un processo crea più thread, può affidarne il compito in modo statico (o al primo o all’ultimo) o in modo dinamico (quello con priorità più alta).

Standard POSIX in C è pthread.

**Thread di livello kernel, utente e ibridi**

**Kernel**, gestiti appunto dal kernel, hanno il vantaggio di essere come i processi, ma con meno informazioni di stato (conveniente per i programmatori), lo svantaggio è nella commutazione in quanto essendo gestiti come processi, se un thread interrotto e quello selezionato dal kernel appartengono allo stesso processo si genera comunque overhead. C’è parallelismo.

**Utente**, sono gestiti dalla libreria dei thread, e quindi la commutazione è veloce non essendo coinvolto il kernel. Non c’è ne concorrenza ne parallelismo. Il vantaggio principale è quello di avere una commutazione meno onerosa rispetto ai T. di livello kernel, in quanto la sincronizzazione e la schedulazione sono implementate dalla libreria. Gli svantaggi sono due, ovvero che il kernel non distingue se è un processo o un thread (un thread bloccato blocca tutti i thread di quel processo), e al più un thread alla volta può essere operativo.

**Ibridi**, implementano sia i T. di livello kernel che di livello utente, e implementano un metodo per associare le due tipologie di T.

**Sincronizzazione dei processi**

Quando i processi (o thread, è uguale) concorrono per l’accesso ai dati che condividono, si parla di sincronizzazione, necessaria al fine di non avere dati inconsistenti. Per fare ciò vengono definite quelle che sono chiamate “sezioni critiche”, ovvero porzioni di codice eseguite in maniera mutuamente esclusiva.

I processi interagiscono tra di loro per scambiare dati, messaggi etc.. mediante:

* **read\_set**: insieme di dati letti dal processo Pi e/o segnali ricevuti da Pi
* **write\_set**: insieme di dati modificati dal processo Pi e/o segnali inviati da Pi

**Processi interagenti**: due processi, Pi e Pj, sono interagenti se il write\_set di uno dei due processi si sovrappone al write\_set o al read\_set dell’altro, ovvero:

(read\_seti ∪ write\_seti) ∩ (read\_setj ∪ write\_setj)

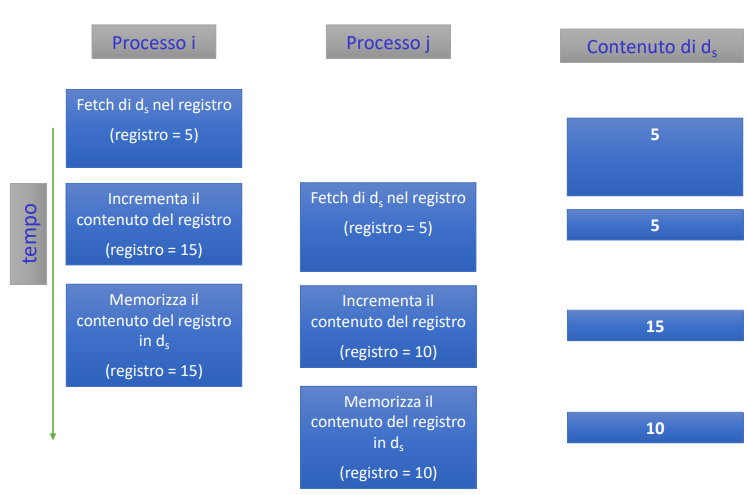
I processi che non interagiscono sono chiamati **processi indipendenti**.

**Race condition**

Quando l’accesso ai dati non condiviso non è coordinato, avviene race condition.

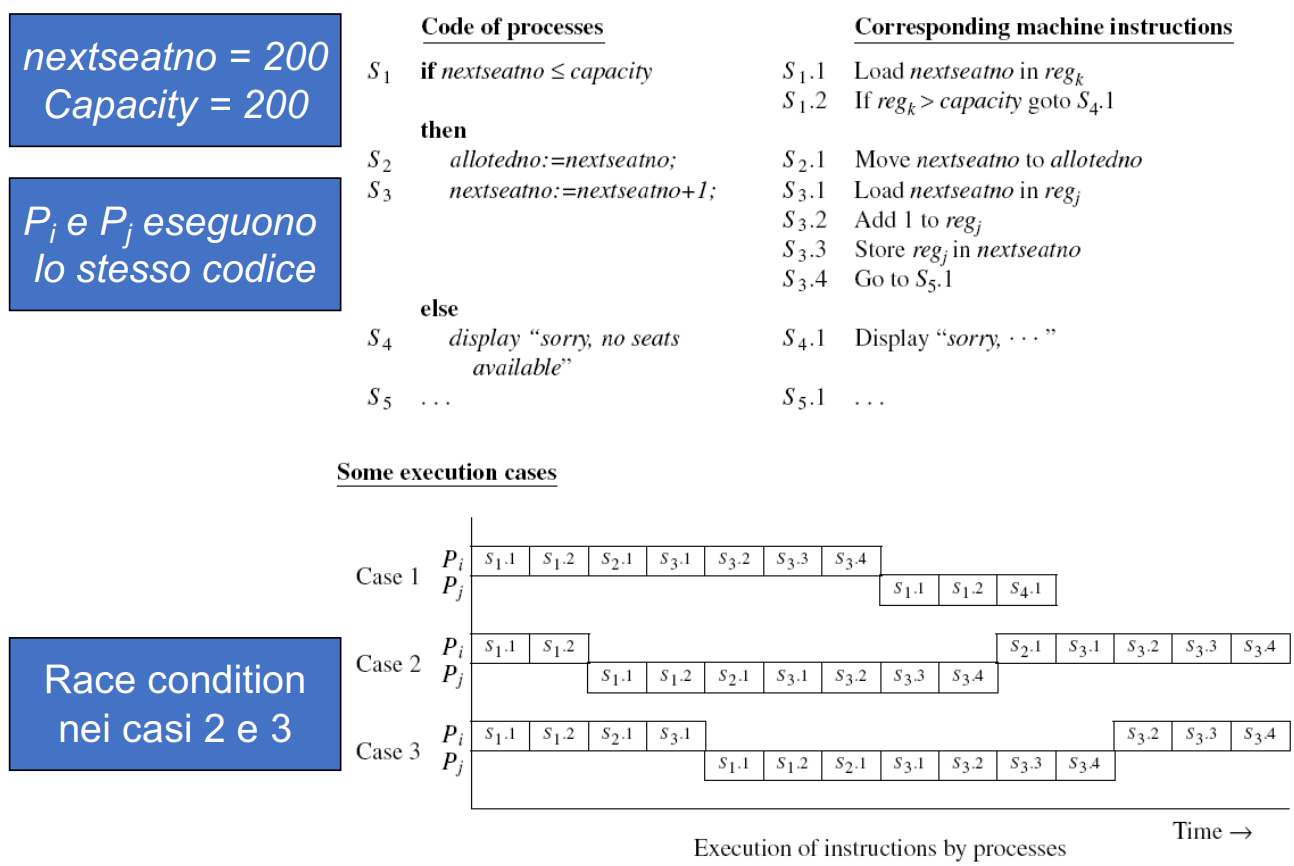
Operazione Ai: ds = ds + 10;

Operazione Aj: ds = ds + 5;



**Una condizione secondo la quale il valore di un dato condiviso ds, ottenuto come risultato dall’esecuzione delle operazioni ai e aj su ds nell’ambito di processi interagenti, può essere differente sia da**

fi(fj (ds)) che da fj(fi(ds))



nextseatno all’inizio è 200, quindi c’è un posto disponibile.

L’esempio si basa sul fatto che se il Processo Pi entra nell’if e viene prelazionato prima di effettuare l’istruzione S2.1, avviene race condition, in quanto dopo che Pj avrà prenotato un posto e Pi verrà ripristinato, nextseatno sarà uguale a 201, ma Pi continuerà l’esecuzione essendo già nell’if, e questo non può accadere. Per evitare ciò si ricorre alla mutua esclusione.

Per verificare se in un applicazione, l’esecuzione dei processi causa race condition, bisogna che vengano soddisfatte le **condizioni di Bernstein**:

( read\_seti ∩ write\_setj ) != ∅

( write\_seti ∩ read\_setj ) != ∅

( write\_seti ∩ write\_setj ) != ∅

Quando si hanno più processi:

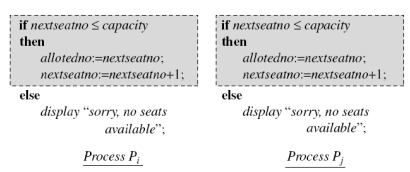
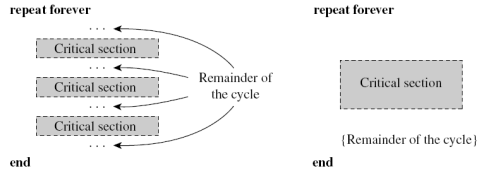
* si determina se la coppia Pi Pj richiede sincronizzazione per l’accesso ai dati
* garantire la mutua esclusione

Per fare ciò si definisce ***update\_set***, ovvero l’insieme di dati aggiornati dal processo Pi (letti modificati e riscritti da Pi).

Come prima, anche in questo caso avviene race condition se

*( update\_seti ∩ update\_setj ) != ∅*

**Sezione critica**

Per implementare la mutua esclusione si ricorre all’utilizzo di sezioni critiche.

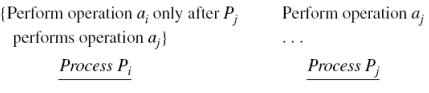
**Proprietà essenziali di un CS**

* **mutua esclusione**: in ogni momento, al più un processo può entrare in sezione critica.
* **Progresso**: quando non c’è nessun processo in sezione critica, e uno di questi vorrebbe entrarci, deve essere garantito l’accesso.
* **Attesa limitata**: dopo che un processo Pi ha indicato di voler accedere alla CS, il numero di volte che gli altri processi possono ottenere l’accesso a una CS prima di Pi è limitato da un numero finito.

Il progresso e l’attesa limitata insieme prevengono la **starvation**.

**Starvation**: l’incapacità perpetua da parte di un processo di ottenere le risorse di cui necessita per essere eseguito.

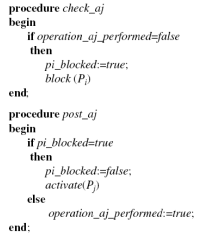
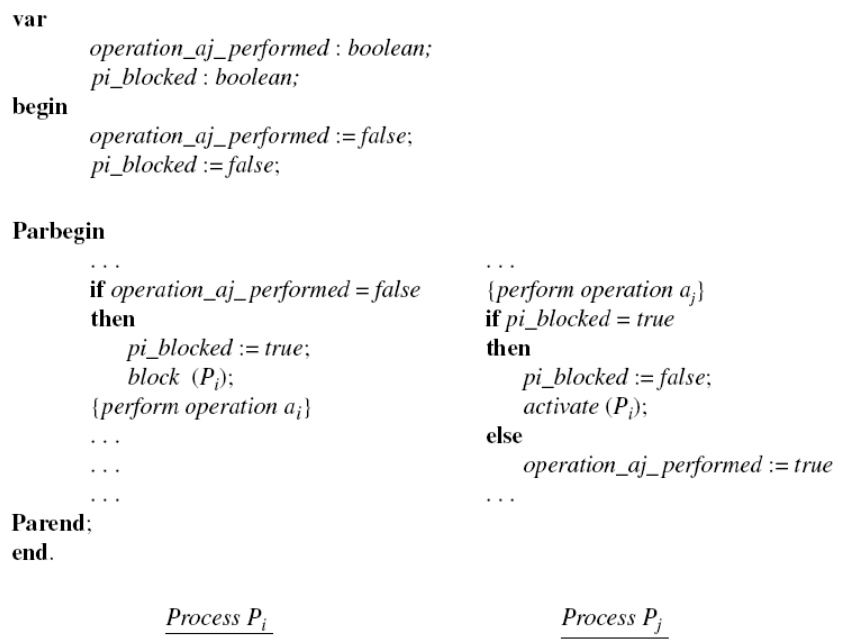
**Sincronizzazione di controllo**



Per garantire che l’esecuzione delle operazioni avvenga nell’ordine desiderato nella comunicazione tra due processi, bisogna effettuare il controllo della sincronizzazione.

**Operazione indivisibile**: detta anche operazione atomica, garantisce l’esecuzione di una sequenza di azioni senza essere prelazionati. Non può essere eseguita concorrentemente né con se stessa ne con altre operazioni.

Due operazioni indivisibili sono check\_aj e post\_aj.



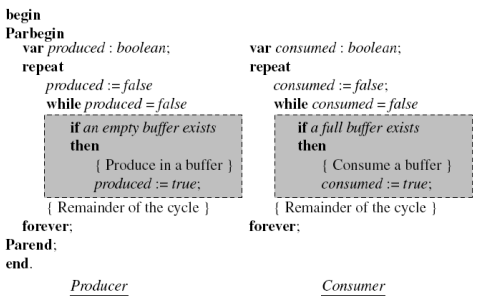
versione naive

**Produttori consumatori con buffer limitati**

Si dispone di un numero non definito di produttori e di consumatori, ed un insieme finito di buffer.

Un buffer si dice “pieno” quando un produttore produce,

Immagine che contiene testo, schermata, Carattere, algebra

Descrizione generata automaticamente“vuoto” quando un consumatore estrae un elemento dal buffer (all’inizio il buffer è vuoto).

Versione efficiente mediante l’utilizzo di check e post

Versione non efficiente

Ci sono vari problemi in questa soluzione, tra i quali la mancata segnalazione tra prod e cons della produzione/consumazione (check e post).

**Lettori e scrittori**

La soluzione deve rispettare vari requisiti:

* molti lettori possono leggere concorrentemente
* Nessuno può leggere mentre uno scrittore sta scrivendo
* Solo uno scrittore alla volta può scrivere
* (opzionale) Un lettore ha priorità non prelazionable sugli scrittori

**Immagine che contiene testo, schermata, Carattere, documento

Descrizione generata automaticamente**

**I filosofi a cena**

Ogni processo rappresenta un filosofo, e:

* ciascuno deve poter mangiare (prendendo una forchetta per volta) quando ha fame
* nessuno dovrebbe morire di inedia

Evitare deadlock e livelock.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamenteSchema di un filosofo Pi.

In questa soluzione c’è sia il rischio di deadlock che di race condition, salvo la condizione in cui se la forchetta di destra non è disponibile, rilascia la forchetta di sinistra e riprova più tardi (livelock).

Una soluzione migliore è la seguente:

Immagine che contiene testo, schermata, Carattere, documento

Descrizione generata automaticamenteIn questa soluzione però il loop causa una condizione di attesa attiva.

**Approcci algoritmici per le sezioni critiche**

Abbiamo algoritmi a due processi, e ad n-processi.

**Soluzioni per 2 processi**

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

Questa soluzione per 2 processi viola la condizione del progresso.

**Immagine che contiene testo, schermata, Carattere, diagramma

Descrizione generata automaticamente**

In questo caso viene violata la mutua eslusione e c’è possibilità di avere deadlock.

**Algoritmo di Dekker**

Combina le soluzioni di entrambi, e viene utilizzata una variabile “turn” per decidere quale dei due processi P1 e P2 deve entrare in CS.

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

**Algoritmo di Peterson**

Usa un array booleano, di flag (ciascuno per ogni processo), che equivalgono alle variabili di stato utilizzate nell’algoritmo di Dekker.

Quando un processo vuole entrare in CS imposta il flag a true e successivamente a false quando esce.

Turn evita livelock.

Immagine che contiene testo, schermata, Carattere, documento

Descrizione generata automaticamente

**Soluzioni per n-processi**

A differenza di prima, qui ogni processo controlla lo stato di altri n-1 processi.

**Algoritmo del panettiere**

Ogni processo prende un numero, e quello con il numero più piccolo viene servito (entra in CS).

Vengono utilizzati due array:

* choosing [0..n-1 ], dove choosing [i] indica se Pi è impegnato nella scelta
* number [0..n-1], dove number [i] contiene il numero scelto da Pi.

(number[i] = 0 significa che Pi non ha scelto il numero)

Quindi il processo con la coppia (number[i],i) viene servito prima:

Immagine che contiene testo, Carattere, schermata, Blu elettrico

Descrizione generata automaticamente

L'array "choosing" è una parte fondamentale di questo algoritmo e viene utilizzato per stabilire un ordine di priorità tra i thread in modo che uno di essi possa acquisire la risorsa condivisa in base a tale ordine.

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

**Semafori**

E’ una variabile intera condivisa con valori non negativi, soggetta a due operazioni:

* inizializzazione
* operazioni indivisibili wait e signal

Immagine che contiene testo, ricevuta, Carattere, schermata

Descrizione generata automaticamente

S := S – 1 🡪 sta a significare che il thread ha avuto accesso alla risorsa e quindi deve segnalare di aver utilizzato parte delle risorse.

Immagine che contiene testo, schermata, Carattere, schermo

Descrizione generata automaticamenteImplementazione di un semaforo - no valori negativi

La mutua esclusione può essere implementata utilizzando un semaforo inizializzato a 1.

⬇️

Var sem : semaphore := 1

Begin

wait(sem) wait(sem)

{ sezione critica } { sezione critica }

signal(sem) signal(sem)

{ parte restante del ciclo } { parte restante del ciclo }

End

Pi Pj

E’ possibile utilizzarli anche per le segnalazioni tra i processi

Immagine che contiene testo, schermata, Carattere, ricevuta

Descrizione generata automaticamente

In questo caso non può esserci race condition in quanto la wait e la signal sono operazioni indivisibili.

Esistono poi quelli che sono chiamati semafori binari, ossia una variante dei semafori (contatori), la cui differenza sta nel fatto che possono assumere solo due valori, 0 ed 1, l’implementazione resta la medesima.

**Produttori consumatori con semafori – buffer singolo**

**Immagine che contiene testo, ricevuta, Carattere, schermata

Descrizione generata automaticamente**Si evita attesa attiva poiché i semafori sono utilizzati per controllare se i buffer sono pieni o vuoti.

**Produttori consumatori con semafori – n buffer**

**Immagine che contiene testo, Carattere, schermata, ricevuta

Descrizione generata automaticamente**

**Lettori-Scrittori con semafori**

**Immagine che contiene testo, schermata, Carattere, ricevuta

Descrizione generata automaticamenterunread**: numero di lettori in lettura

**totread**: numero di lettori che intendono leggere o in lettura

stessa cosa per **runwrite** e **totwrite**.

**Esercitazione** (26 ottobre)

Identificazione del problema

* Problema classico (prod cons, lett scritt etc..)
* Problemi di sincronizzazione vari (prob di serializzazione, rendezvous, multiplex, barriera)

Problema barbiere addormentato

* Un barbiere
* N sedie clienti
* 1 posto disponibile

Problema: se non ci sono clienti il barbiere è in attesa.

Quando arriva un cliente, viene risvegliato da un cliente, e il b. lavora.

Se arriva un cliente mentre il b. è occupato, il cliente attende su una delle N sedie.

Se tutte le sedie sono occupate, il cliente và via.

Schema simile rendezvous.

Abbiamo due thread, 4 semafori (per i due rendezvous), e 1 altro semaforo per i clienti che arrivano.

Dichiarazione e inizializzazione:

sem\_CS = 1;

cliente\_disponibile, barbiere\_pronto semaphore = 0;

cliente\_completato, barbiere completato semaphore =0;