

Sistemi Operativi

Programma che controlla l'esecuzione dei programmi applicativi. Realizza un'interfaccia tra l'hardware ed il software applicativo.

Kernel: porzione del S.O. residente in memoria centrale. Contiene le funzioni più utilizzate;

Storicamente, si passa dall'esecuzione seriale dei programmi (schede perforate), ai processi batch (introduzione dei monitor, sw che controlla i programmi in esecuzione). Si introducono anche nuove funzionalità (Memory Protection impedisce l'alterazione della zona di memoria allocata al monitor, Timers impedisce che un job monopolizzi il sistema)

I/O uniprogrammato: il processore attende che si svolgano le operazioni di I/O prima di procedere nell'esecuzione

I/O multiprogrammato: il processore switcha tra diversi job quando è in attesa di I/O (massimizza il tempo processore utile)

Time sharing: si utilizza l' I/O multiprogrammato per gestire più job interattivi assegnando uno slot temporale ad ognuno per l'esecuzione delle istruzioni (minimizza tempi di risposta).

PROCESSO: un'istanza in esecuzione di un programma. Unità di attività caratterizzata da un singolo thread sequenziale di esecuzione, uno stato corrente e un set di risorse associate.

MEMORY MANAGEMENT: Problematiche:isolamento dei processi, allocazione e management automatici, programmazione modulare, protezione e controllo degli accessi, persistenza] (**Soluzioni:** memoria virtuale, filesystem, paging, information flow control, access control, scheduling)

La **Struttura del sistema** è vista come un livello gerarchico, in cui ogni livello si occupa di un determinato aspetto, offrendo funzionalità al livello superiore utilizzando quelle di più basso livello.

SO Moderni: architettura microkernel (kernel con funzioni essenziali: **IPC, Address space, basic scheduling**) multithreaded (processo suddiviso in più thread eseguibili simultaneamente) symmetric multiprocessing (più processori in parallelo) object oriented (kernel modulare) distributed (astrae eventuale distribuzione delle risorse fisiche)

Processi

Include Program Counter, stack, data section. Processi **batch** (non interattivi), **interattivi** (con utenti) e **real-time** (i più critici, dialogano con hw soggetto a vincoli temporali).

DISPATCHER: programma che assegna il processore ai processi. Impedisce che un processo monopolizzi il processore. La prima implementazione è fatta con un modello a 2 stati.

- Stato **Running:** istruzioni in esecuzione
- Stato **Not Running:** in attesa di assegnamento a un processore

Problema: non si può usare una politica FIFO perchè il processo estratto secondo questa logica potrebbe essere bloccato.

Si passa ad un modello più dettagliato, a 5 stati:

- **New:** il processo è stato appena generato
- **Ready:** il processo è pronto per essere assegnato ad un processore
- **Running:** il processo è in esecuzione
- **Blocked:** il processo è bloccato in attesa di un evento
- **Exit:** il processo termina l'esecuzione

Implementazione: **single blocked queue** o **multiple blocked queues**

Il modello viene raffinato e diventa a 7 stati, aggiungendone altri due: il processore infatti è più veloce dell' I/O quindi tutti i processi potrebbero essere in attesa di I/O. Questi processi vengono swappati su disco per liberare memoria (vanno in suspend).

- **ready suspend:** il processo è su disco ma non sta attendendo eventi
- **blocked suspend:** il processo è su disco e attende l'arrivo di un evento

N.B. La sospensione può essere causata dallo swap, decisa dal sistema operativo o dall'utente. Possono esserci processi temporizzati (si sospendono quando non sono attivi) oppure processi child possono essere sospesi dal parent.

Struttura dell'immagine dei processi in memoria virtuale:

1. PCB: (Process Control Block) Struttura dati usata dal sistema operativo per controllare i processi.
2. System stack: ogni processo ha associato uno o più stacks (LIFO) per le chiamate a procedura
3. User Space: il programma da eseguire e la struttura dati che il programma può manipolare
4. Shared Space: spazio condiviso tra processi

PCB[1] L'identificazione di un processo può avvenire tramite ID unico, Parent ID oppure user ID

PCB[2] Informazione di stato: PSW (Program Status Word), registri user, process stack pointers

PCB[3] Informazione di controllo: scheduling e state info, data structuring, IPC flags, signals, msg, privileges, memory management (virtual memory), resource ownership

Il **kernel** gestisce il management dei processi: creazione, modifica, terminazione, scheduling, switching, synchronization, supporto IPC, management dei PCB

Lo switching tra processi avviene salvando lo stato del processo corrente nel suo **PCB** e caricando lo stato dal PCB del processo da attivare (interrupt, traps, supervisor calls).

Come viene trattato il kernel?

1. **Non process kernel:** il kernel è un'entità privilegiata e viene eseguito al di fuori di ogni processo. Solo i programmi utenti rientrano nel concetto di processo
2. **Esecuzione con gli user-process:** il sw del so opera contestualmente agli user programs. Il processo è eseguito in modalità privilegiata quando appartiene al sistema operativo. I processi possono eseguire sia user programs, sia system programs.
3. **Process-Based S.O.:** le maggiori funzionalità del kernel sono poste in processi separati. Utile in contesti di processori dedicati.

Inter Process Communcation

In un dato istante, in un sistema sono presenti più processi: essi potrebbero interferire tra di loro. Oppure potrebbero cooperare per il raggiungimento di un task comune (bisogna coordinare). Bisogna coordinare l'accesso alle risorse condivise.

Separando le attività in più processi indipendenti si può guadagnare in efficienza.

Contesti di concorrenza:

1. **Multiple applications:** CPU swicha tra più processi
2. **Applicazioni strutturate:** un'applicazione come set di processi concorrenti
3. **Struttura OS:** il sistema operativo è un set di processi o di thread.

La concorrenza introduce problemi: sharing di risorse globali, management di allocazione delle risorse, difficoltà a localizzare gli errori.

Concetti fondamentali per il sistema operativo:

- **Tenere traccia dei processi attivi**
- **Allocazione / Deallocazione delle risorse**
- **Proteggere i dati e le risorse**
- **Risultato dei processi indipendente dalla velocità di esecuzione**

Livelli di interazione tra processi:

- **non sono informati dell' di altri processi** (sono indipendenti - competizione sulle risorse)
- **indirettamente informati:** (i processi condividono alcune risorse)
- **direttamente informati:** (i processi lavorano insieme - cooperazione)

Competition on resources: Mutua esclusione, sezioni critiche, deadlock, starvation

Cooperation by sharing: come sopra, più problematica della consistenza dei dati

Cooperation by communication: possibili deadlock e starvation

Requisiti:

Sezione critica: segmento di codice in cui un processo utilizza risorse condivise.

- Solo un processo in un istante può eseguire la sua sezione critica (mutual exclusion).
- Un processo che si ferma nella sua sezione non critica non deve interferire con gli altri
- Se la sezione critica non è occupata da altri processi, devo avere accesso immediato (progress)
- Non si fanno assunzioni riguardo alla velocità del processo e al numero di processi concorrenti.
- Un processo rimane nella sezione critica per un tempo finito. Inoltre chi richiede di accedere alla sezione critica, deve ottenerla in tempo finito (bounded waiting)

Algoritmo BUSY WAITING:

How do we implement a mutual exclusion policy? No matter what policy we implement, a process that is about to enter its critical section must first check to see if any other processes are in their critical sections. As we shall see, there are, potentially, a number of ways to enforce mutual exclusion. Ideally, we would like to implement a policy that works in the most efficient way possible.

What about using a lock variable, which must be tested by each process before it enters its critical section? If another process is already in its critical section, the lock is set to 1, and the process currently using the processor is not permitted to enter its critical section. If the value of the lock variable is 0, then the process enters its critical section, and it sets the lock to 1. The problem with this potential solution is that the operation that reads the value of the lock variable, the operation that compares that value to 0, and the operation that sets the lock, are three different atomic actions. With this solution, it is possible that one process might test the lock variable, see that it is open, but before it can set the lock, another process is scheduled, runs, sets the lock and enters its critical section. When the original process returns, it too will enter its critical section, violating the policy of mutual exclusion.

The only problem with the lock variable solution is that the action of testing the variable and the action of setting the variable **are executed as separate instructions.** If these operations could be combined into one indivisible step, this could be a workable solution. These steps can be combined, with a little help from hardware, into what is known as a **TSL** or **TEST and SET LOCK** instruction. A call to the **TSL** instruction copies the value of the lock variable and sets it to a nonzero (locked) value, all in one step. While the value of the lock variable is being tested, no other process can enter its critical section, because the lock is set. Let us look at an example of the **TSL** in use with two operations, `enter_region` and `leave_region`:

`enter_region:` (executed when a process wants to enter its critical section)

```
    tsl register, lock    // copy lock to register and set lock to 1
    cmp register, 0      // see if lock variable was set
    jnz enter_region    // if lock was set, loop
    ret                 // enter critical section
```

`leave_region:` (executed when a process wants to leave its critical section)

```
mov lock, 0          // store a 0 in lock variable
ret                 // done
```

You can see from the example above that if a process is interrupted after executing the TSL, there is no danger that another process might enter its critical section. This is so because the lock variable is set to a non-zero value while the original process is testing it. Another advantage of the TSL instruction is that it works on machines with many processors as well.

Look closely at the code for the "enter_region" example, and you will see that if a process tests a variable which is locked, it will continue to test the variable again and again until it can enter its critical section. In other words, as long as a process is denied access to its critical section, it will stay in a tight loop and wait until it can proceed, all the while wasting processor time. This continuous testing of the lock variable is called **busy waiting**. Mutual exclusion policies that require busy waiting waste valuable processor time, and in some cases can lead to situations where a process will test the lock variable forever, a very undesirable occurrence (**STARVATION** e **DEADLOCK**)

C'è anche la soluzione di **Peterson** (turn e flag[i])

Supporto Hardware x BUSY WAITING:

- Disabilitazione degli interrupt (disabilitati quando è richiesta la mutua esclusione - perdo efficienza in esecuzione! Limitazioni nell'esecuzione interleaved di programmi). Pericolo di blocco del sistema (loop infiniti, waiting 4 res). Soluzione praticabile su sistemi uniprocessore.
- Operazioni macchina speciali : test & set

SLEEP & WAKEUP

As we have seen, busy waiting can be wasteful. Processes waiting to enter their critical sections waste processor time checking to see if they can proceed. A better solution to the mutual exclusion problem, which can be implemented with the addition of some new primitives, would be to block processes when they are denied access to their critical sections. Two primitives, **Sleep** and **Wakeup**, are often used to implement blocking in mutual exclusion.

How do Sleep and Wakeup Work?

Essentially, when a process is not permitted to access its critical section, it uses a system call known as Sleep, which causes that process to block. The process will not be scheduled to run again, until another process uses the Wakeup system call. In most cases, Wakeup is called by a process when it leaves its critical section if any other processes have blocked.

In the next few sections we will discuss three different implementations of mutual exclusion policies which do *not* require busy waiting. The first, **semaphores**, is a solution proposed by E.W. Dijkstra in 1965. Dijkstra's semaphores closely model the Sleep and Wakeup principle. Secondly, we have **monitors**, which are a solution to the mutual exclusion problem implemented in certain programming languages. Lastly we will discuss **message passing**, which has the advantage of being able to work in distributed operating systems.

As we have seen, sleep and wakeup is a solution to the problem of race conditions which does not require busy waiting. One must be careful in implementing a sleep

and wakeup policy, since race conditions can arise in certain circumstances. Consider the following implementation of sleep and wakeup.

```
var occupied;          // 1 if critical section is occupied, 0 if not.
var blocked;          // counts the number of blocked processes

Enter_Region:        // process enters its critical section
{
  IF (occupied){      // if critical section occupied
    THEN blocked = blocked + 1; // increment blocked counter
    sleep();          // go to "sleep", or block
  }
  ELSE occupied = 1; // if can enter critical section,
                    // increment counter
}

...
...                // critical section
...

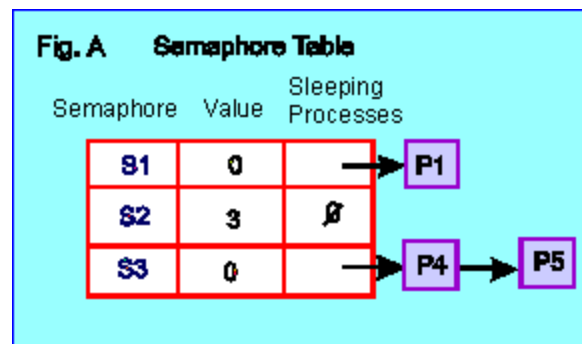
Exit_Region:         // process exits its critical section
{
  occupied = 0;
  IF (blocked){
    THEN wakeup(process); // if another process is sleeping,
                          // wake the process up.
    blocked = blocked - 1; // decrement blocked counter
  }
}
```

In the example above, a process wanting to enter its critical section must first check to see if any other processes are currently in the critical section. If so, the process calls the Sleep() system call, which causes it to block. Upon leaving its critical section, a process must check to see if any other processes have been put to sleep, waiting to enter their critical section. If a process is waiting, the process now exiting its critical section must wakeup the sleeping process.

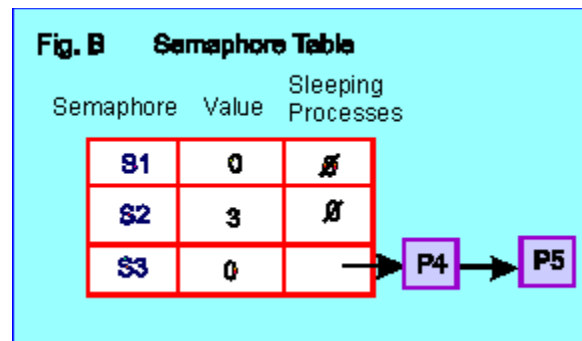
Now consider what would happen if two processes, A and B, were using this method for mutual exclusion. Process A enters its critical section, and before it leaves the critical section, process B is scheduled. Now process B attempts to enter its critical section. It sees that process A is already there, so it increments the "blocked" variable in preparation to go to sleep. Just before it can call the Sleep primitive, process A is scheduled again. Process A exits its critical section. Upon exiting, it sees that the "blocked" variable is now 1, so it calls Wakeup on process B. But process B is not yet asleep. The Wakeup is lost, so when process B is scheduled again, it will call Sleep, and block forever.

Semaphores allow a Sleep and Wakeup mutual exclusion policy to be implemented without the risk of losing a Wakeup. Basically, a semaphore is a new type of variable. Semaphores can have a value of 0 (meaning no Wakeups are saved), or a positive integer value, indicating the number of sleeping processes. Two different operations can be performed on a semaphore, **DOWN** and **UP**, corresponding to Sleep and Wakeup.

Let us look at an example of semaphores. In figure A, there are three semaphores: S1 has a value of zero and process P1 is *blocked* on S1; S2 has a value of three, meaning three more processes may execute a down operation on S2 without being put to sleep; S3 has a value of zero, and has two sleeping processes, P4 and P5, blocked on it.



Now let us look at what happens when a process executes an UP on S1. In figure B, process P1 is activated by an UP operation. It executes a DOWN on S1 and enters its critical section. The state of the semaphores after these actions has now changed.



Here is an example of how semaphores are used to enforce mutual exclusion:

```
semaphore mutex;      // variable's value is restricted to 0 or 1.

DOWN(mutex);        // execute a DOWN on mutex before entering
...                 // critical section
...

UP(mutex);          // execute an UP on mutex when leaving
                   // critical section
```

The value of a semaphore need not always be 0 or 1. Semaphores can have a value greater than one. The original value of the semaphore dictates the total number of processes that can enter a specific region at a time. For instance, a semaphore with an original value of 4 would allow four processes to enter the section of code they protect. We will see in later examples how this is useful.

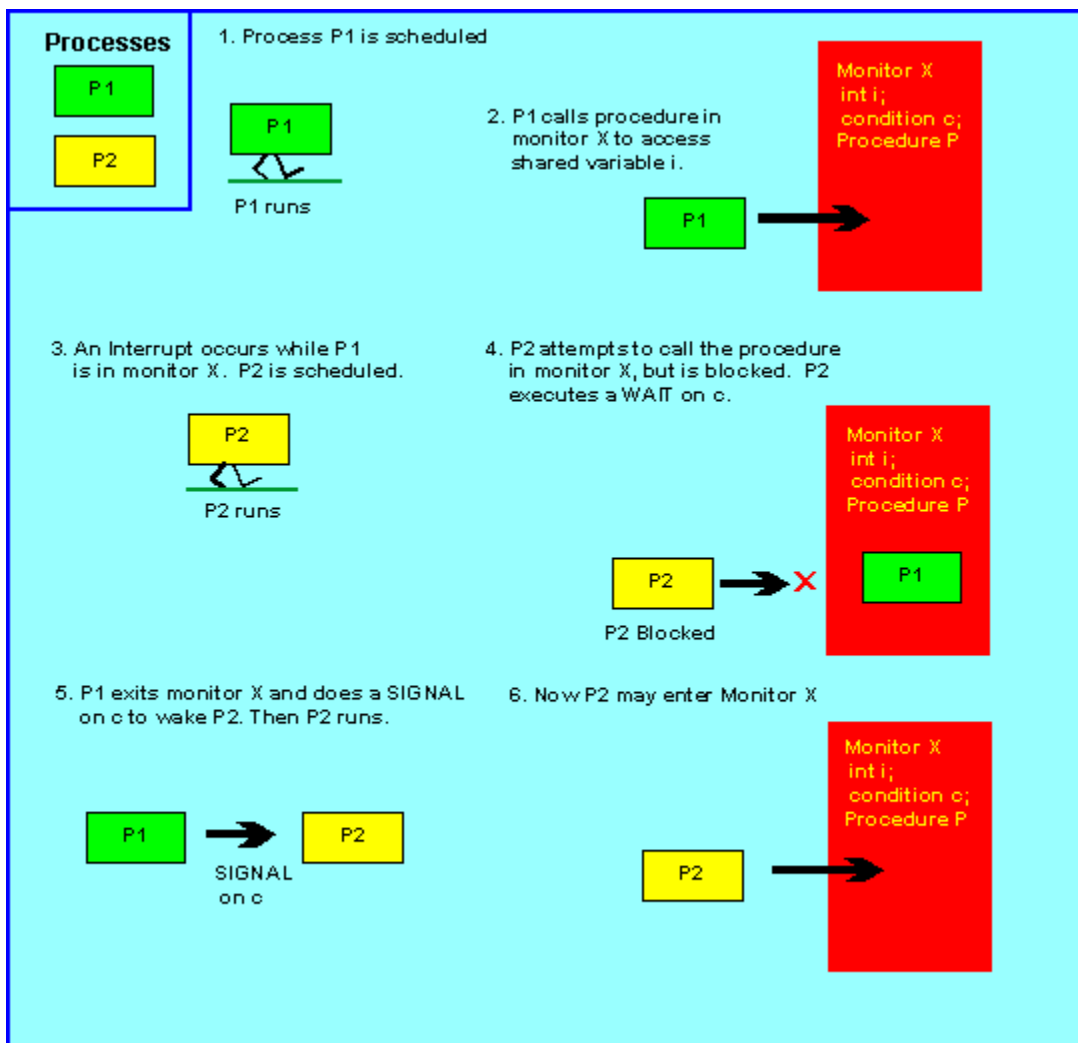
The problem with the semaphores described in the previous page is that they require the programmer to use low-level system calls. It is easy to put these system calls in the wrong order, resulting in a deadlock. A higher level solution would make implementing mutual exclusion and synchronization a little easier. Monitors do just that. They are a high level construct found in some programming languages which contain variables, procedures, and data structures. Monitors closely resemble the protected classes and modules which are found in languages like C++ and Ada, in that they only allow access to the variables they contain through the functions

they contain. Only one process may execute a function within a monitor at any one time. This is what allows monitors to enforce mutual exclusion.

How do monitors handle blocking? Wait & Signal

What happens when a process calls a function inside a monitor that is already occupied by another process? Monitors use condition variables and two operations called WAIT and SIGNAL. When a process calls a function in a monitor that is already occupied, it performs a WAIT operation on some condition variable. This will cause the process to block, until the other process, which currently occupies the monitor, leaves the monitor and performs a SIGNAL on the condition variable.

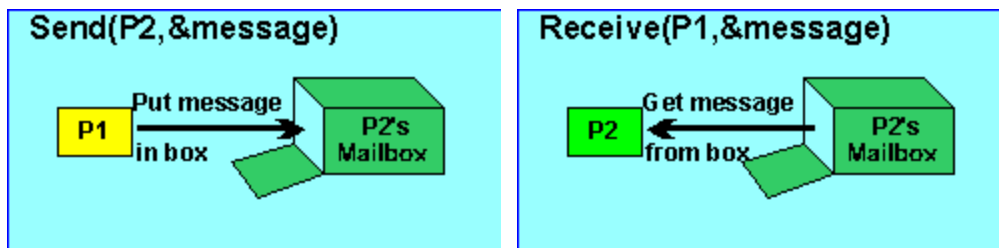
Look at the following example, in which two processes interact with one monitor.



So far we have discussed two different types of interprocess communication: semaphores and monitors. These methods work well on computers with shared memory, but they are not effective on distributed systems. Semaphores and monitors provide no machine to machine communication capability. The last method we will discuss, **Message passing**, solves this problem. Two primitives, SEND and RECEIVE are used in the message passing scheme. The SEND primitive sends a message to a destination process while the RECEIVE primitive receives a message from a specified source process. Message Passing works on distributed systems because these messages

can be sent from machine to machine through a network.

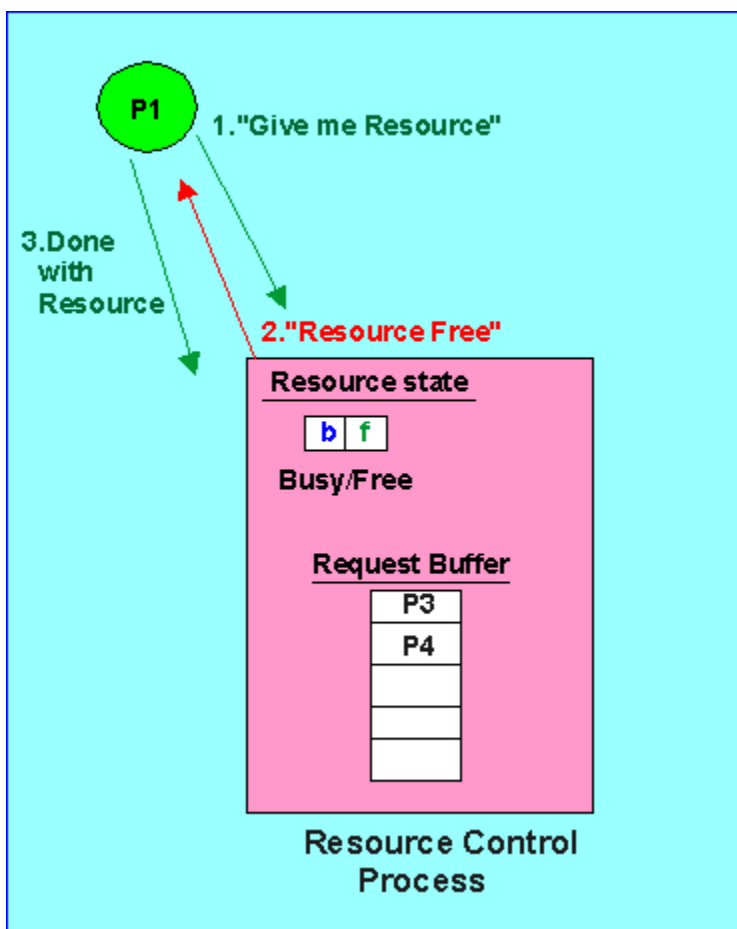
Typically, each process has a mailbox; a buffer which receives all the messages which are sent to that process. The destination of the SEND and RECEIVE system calls is a process' mailbox, not the process itself.



The pictures above illustrate the **SEND** and **RECEIVE** primitives used in message passing. **SEND** takes two arguments, the destination mailbox and the message. **RECEIVE** also takes two arguments, the ID of the sender, and the message.

How is Mutual Exclusion implemented with Message Passing?

Several methods exist for implementing mutual exclusion with message passing. One method uses processes to regulate access to shared resources. A resource control process has a variable which records the status of a resource (*free or busy*), and a buffer which contains request messages from other processes.



This pictures shows an interaction between a process that wants to access a shared

resource, and a resource control process. Three messages are sent here. Message (1) is a request for access to the resource. Process P1 then blocks, waiting for a reply from the resource control process. Message (2) is the reply sent by the control process. This message wakes up P1, and P1 now uses the resource. Message (3) informs the resource control process that P1 is done using the resource.

The code for this interaction looks like this:

P1:

```
send(control_process, "Give me resource");
receive(control_process, Message); // block while waiting for a reply

    {
    ...Critical Section...
    }

send(control_process, "Done with resource");
```

Scheduling (uniprocessore)

Sostanzialmente il tempo di macchina si divide in **CPU Bursts** (esecuzione) e **I/O Bursts** (attesa per I/O). Lo scheduler decide a chi assegnare la CPU (scheduling preemptive, scheduling non-preemptive). Bisogna garantire quanto più possibile il troughtput di processi assegnati al processore, cercando di assegnare le risorse correttamente in base alle richieste e alle priorità e minimizzando il tempo di esecuzione medio dei processi (tempo per svolgere l'operazione), il tempo di risposta e il tempo di attesa nella coda.

Rispetto al modello a stati si suddivide lo scheduling tra **short** (ready, running, blocked), **medium** (suspend) e **long term scheduling** (new, exit). L'I/O scheduling invece decide a che processo assegnare le risorse di I/O

- **Long term:** decide quali processi ammettere nel sistema
- **Medium term:** fa parte della funzione di swapping, decide dove mettere in memoria i processi
- **Short term:** decide come gestire la coda dei processi per ammetterli in run (Dispatcher). Il dispatcher è quello più utilizzato in un periodo di tempo, viene invocato ad ogni evento (I/O, clock, interrupt...). Lo scheduler può essere user o system oriented, oppure orientato alla performance oppure no.

Ad ogni processo generalmente è associata una **priorità**: lo scheduling è basato su diverse code che gestiscono questo aspetto (bisogna fare attenzione alla Starvation dei processi a bassa priorità, si decide dinamicamente la priorità anche in base all'età del processo)

- Dispatching Preemptive: un processo in run rimane tale finchè non richiede I/O

oppure termina

- Dispatching non Preemptive: lo stato del processo può essere modificato dal SO

Politiche di scheduling (gestione coda):

1. **FCFS**: i processi sono inviati al processore nell'ordine di arrivo. Il tempo di attesa ha una forte variabilità, non si distingue tra “processi piccoli” e “processi grandi” (che potrebbero essere eseguiti subito prima di quelli più onerosi)
2. **SJF (shortest job first)**: ad ogni processo si associa la grandezza del CPU burst che richiede per essere eseguito. Viene data priorità ai processi con burst minore. In questo modo i processi “veloci” vengono eseguiti subito (preemptive). Modifica non preemptive: **SRTF (shortest remaining time first)**: se arriva in CPU un processo con burst time < tempo rimanente del processo in run, si blocca quello in run e si invia in run quello piccolo. Ottimizza il tempo di risposta.
3. **SPN (shortest process next)**: simile a SJF, ma si usa una stima del tempo di esecuzione del processo. Tuttavia è difficile predire le stime di esecuzione per processi lunghi (se la stima è sbagliata, il processo va in abort: pericolo di starvation x processi lunghi)
4. **SRT (shortest remaining time)**: versione preemptive del SPN
5. **HRRN (highest response ratio next)**: si calcola un punteggio per ogni processo (ratio): viene calcolata la frazione ((tempo stimato d'esecuzione + il tempo di attesa)/t.attesa)
In questo modo si tiene conto anche da quanto un processo si trova nella coda (viene inviato in ready il processo con il ratio più alto)
6. **Feedback**: si penalizzano i processi che sono stati in run più a lungo. Non si può conoscere il tempo rimanente per il completamento di un processo.
7. **Priority**: entra in run il processo a priorità più alta. Se un processo “invecchia” (aging) devo alzare la sua priorità per evitare starvation.
8. **Round-Robin**: si suddivide un periodo di esecuzione in molti quanti temporali prefissati. Ad ogni processo viene associato un quanto entro il quale eseguire le sue operazioni (time sharing)

FAIR SHARE SCHEDULING: le applicazioni utente sono un set di processi (threads). L'utente vuole mantenere una certa performance per la sua applicazione.

Le stime vengono calcolate con una media esponenziale sul tempo di burst corrente. Si possono avere varie politiche più dettagliate combinando in vari modi diverse code a priorità con diverse politiche.

Combinando in vari modi più code con diverse politiche e utilizzando la priorità si ottengono risultati più raffinati:

- **multilevel queues**: creo code diverse per processi batch e processi utente, con diverse politiche locali. Ogni coda è servita sequenzialmente (starvation!) oppure si assegna un **time slice** per ogni coda (es. 80% foreground 20% background)

Deadlock

Un set di processi è detto in **deadlock** se ognuno di essi sta aspettando un evento

che solamente un altro processo appartenente al set può causare (Condizioni necessarie: **mutua esclusione, no preemption, hold & wait, circular wait**)

Come si gestisce:

- **Ignorare il problema:** applicabile se DL è poco frequente e non è mission-critical.
- **Detection & Recovery:** costruisco il grafo delle risorse e lo mantengo aggiornato. Per ispezione posso stabilire se sono presenti dei deadlock (cicli nel grafo).
 1. Recovery by preemption: se un processo sta causando un deadlock, gli vengono tolte le risorse che vengono assegnate al richiedente.
 2. Recovery by rollback: checkpoints dello stato dei processi + immagine dati. Se ho deadlock, libero la risorsa e faccio rollback del processo coinvolto.
 3. Recovery through killing processes: killo i processi che causano i deadlock (quello che ha svolto meno lavoro, quelli che causano più deadlocks, quelli ristartabili senza problemi)
- **Prevention:** si cerca di assicurare che almeno una delle condizioni necessarie per il deadlock non sia mai soddisfatta.
 - Mutua esclusione: risorse condivise, spoolers (non è applicabile in tutti i casi)
 - Hold & Wait: P chiede le risorse prima dell'esecuzione, se no viene sospeso (difficile sapere in anticipo le risorse, problemi di "overbooking")
 - No preemption: applicabile raramente
 - Circular waiting: numero le risorse. Ogni processo può detenere una sola risorsa alla volta. Così garantisco aciclicità del grafo delle risorse. (usare 1 sola risorsa è restrittivo!)

In alternativa si adotta un meccanismo **2 Phase Locking** simile a quello applicato nei DBMS.

- **Avoidance:** concetto di **Safe State** (non ci sono deadlock ed esiste un modo per soddisfare le richieste sulle risorse) e **Unsafe State** (i processi possono andare avanti ma non posso garantire che le richieste sulle risorse verranno soddisfatte). Il sistema permette l'allocazione di risorse solo nel Safe State. (Algoritmo di Banker)

Starvation

Certi processi, seppur non implicati in un deadlock, non vengono mai serviti. Problema tipico dei sistemi a priorità (non capita quindi in sistemi che adottano politiche FCFS o Round-Robin).

Deadlock in sistemi distribuiti

Anche in un contesto distribuito è possibile incorrere in situazioni di deadlock:

1. **Ignore:** sempre possibile
2. **Detection & Recover:** largamente usato
3. **Avoidance:** mai usato, troppo difficile predirre l'uso delle risorse (overhead troppo ampi, bisogna distribuire il sistema di check su ogni nodo rendendolo mutualmente esclusivo)

4. **Prevention:** applicabile specialmente in sistemi transazionali (centralizzato, gerarchico, distribuito)

Threads

Si suddivide un processo in 2 componenti: **risorse allocate** e **contesto d'esecuzione**. Un processo deve avere almeno un thread, i thread di un processo condividono le risorse di quel processo.

Ad ogni thread si associano informazioni su PC, stack, insieme dei registri, thread figli e stato.

Vantaggi derivanti dall'uso di threads:

- **Maggior efficienza** di gestione e risorse
- **Condivisione di memoria**, non usano sempre il kernel per l'I/O
- **Cambi di contesto più veloci**
- **Implementazioni efficienti** di importanti architetture
- **Supporto multiprocessore** più semplice.

Architetture a thread (user threads oppure kernel threads utilizzando la libreria **thread package**):

- **Team Model:** si partiziona un'applicazione individuando alcune attività da separare e creando un thread per ogni attività: più reattività agli inputs e maggiore modularità
- **Dispatcher:** simile al Team Model. **Thread Dispatcher** che riceve le richieste di servizio e le invia ai thread che le eseguono (replicazione di attività)
- **Pipeline:** si partizionano le attività su base temporale. Si gestisce una catena di algoritmi: l'uscita di un algoritmo rappresenta l'ingresso per un altro algoritmo. I thread possono procedere anche parallelamente (nel caso uno si ponga in attesa di I/O)

User thread package: si implementa uno strato runtime che si posiziona sopra il kernel (Vantaggi: permette di aggiungere supporto ai thread a SO che non li prevedono, possibilità di gestire le politiche di scheduling in modo customizzato Svantaggi: architettura monolitica, un blocco di un thread può bloccare tutto il processo, difficile supporto multiprocessore)

Kernel thread package: il supporto ai threads è implementato direttamente nel kernel. Lo scheduling è gestito associando quanti di tempo ad ogni thread (I/O blocked thread non blocca l'esecuzione degli altri thread) (Il supporto si effettua con chiamate a sistema, quindi ho un overhead rispetto alla soluzione user package)

Relazioni Thread-Processi:

{One|Many} Thread <--->{One |Many}Process

Meccanismi di comunicazione e sincronizzazione tra thread:

Si utilizzano più o meno le stesse idee già sviluppate per i processi: semafori,

mutex, condition variables, monitors, message passing.

- **Semaforo:** variabile intera inizializzata a valore non negativo.
 1. **Wait():** decremento il valore. Se è negativo, blocco il chiamante
 2. **Signal():** incremento il valore. Se resta non positivo, si risveglia un processo bloccato.Necessaria politica per scegliere quale processo risvegliare su un semaforo tra quelli a lui allocati.
- **Mutex:** essenzialmente semaforo binario (**init, lock, trylock, unlock**)
- **Condition variable:** variabili con coda d'attesa associata. In genere si usano assieme ai mutex. (**init,wait, signal, broadcast:** risveglia tutti i processi nella coda)
- **Monitor e message passing:** esattamente come per i processi.

POSIX Threads (esempio di implementazione dell'architettura a thread)

Tread package dello standard posix. Specifica tipi e funzioni per i thread, nonché mutex e condition variables. Principali attributi di un thread:

- **detachstate:** un thread può essere detached(quando termina viene cancellato) o joinable (viene cancellato solo se è argomento di una pthread_join).
- **inheritsched:**determina se il thread eredita la politica di scheduling del thread padre
- **schedpolicy:** la politica di scheduling associata al thread
- **stacksize:** dimensione minima per lo stack del thread

Inter Process Communication (IPC)

La comunicazione tra processi può avvenire in vari modi: Message passing, Signals, Pipes and FIFOs, Tracing & Debugging, Sockets, RPC. Tutti i modi sono duali: quello che posso fare con uno, lo posso fare anche con l'altro

Message Passing: modalità generica di comunicazione tra processi, disponibile sia in ambiente locale che distribuito. Si implementa con le primitive **send & receive**. Le **procedure** possono essere non bloccanti (comunicazione asincrona tramite system buffers) oppure bloccanti (comunicazione sincrona). L'**indirizzamento dei messaggi** può avvenire direttamente (source/dest id) oppure indirettamente (shared mailbox, coda di messaggi). La **mailbox** può essere privata oppure condivisa (**porta: una mailbox in sharing tra più clients e un server**). *Le mailboxes sono detenute dal processo/thread che ne ha richiesto l'apertura al SO. I messaggi contengono varie informazioni (tipo,source/dest id, control info)*

Pipes: buffer condiviso e limitato in politica FIFO, che mette in comunicazione due processi. Basato sul modello producer/consumer. La mutua esclusione è imposta

dall'OS. Le chiamate sono bloccanti (produttore si blocca se non c'è spazio nel buffer, consumatore si blocca se non c'è input nel buffer). **Named pipes (FIFOs)** sono pipes con un riferimento nel filesystem.

Shared Memory: un blocco di memoria virtuale è condiviso da più processi. I processi attaccano una regione di memoria condivisa al suo spazio di indirizzamento virtuale con `shmget`. IPC più veloce in UNIX. Bisogna provvedere alla mutua esclusione.

Segnali: sono messaggi di un singolo bit. Modalità di notifica asincrona di un evento. Gli interrupts sono segnali. Vengono settati nella `process table` del processo. Il segnale è trattato non appena il processo entra in user mode. Se non viene specificata una routine, in genere ad ogni segnale è associato un comportamento di default. In UNIX ad ogni segnale si associa una funzione da compiere quando viene catchato quel segnale. Si possono disabilitare e riabilitare gli interrupts (ad esempio durante una `fork`). Per lanciare segnali si può usare la primitiva `alarm` o `kill`.

Semafori: sono una generalizzazione dei semafori contatori (con più funzionalità). Si include

- Il valore **S** del semaforo
- il numero di processi che attendono l'aumento di **S**
- il numero di processi che attendono **S=0**

Ci sono code di processi bloccate su un semaforo. In UNIX: Primitiva `semget` per creare un array di semafori. Primitiva `semop` per compiere atomicamente un'operazione su ogni semaforo. L'operazione da compiere è diversa a seconda del valore di `sem_op`:

1. `sem_op > 0`: `S++` e i processi in attesa di un certo valore di `S` sono risvegliati
2. `sem_op < 0`:
 - `|sem_op|>S`: processo corrente bloccato sull'evento
 - `|sem_op|<=S`: `S=S-|sem_op|`
then if `S=0` risveglia processi che attendono `S=0`
3. `sem_op = 0`: `S=0` non faccio niente. `S!=0`, blocco il processo corrente sull'evento `S=0`

Sockets: meccanismi di comunicazione connection-oriented. Sono gli end-point di una comunicazione bidirezionale tra processi. Famiglie di indirizzi UNIX, INET, PROPRIETARI.

Tipi di sockets: **stream** (over TCP), **datagram** (over UDP), **raw** (accesso diretto ai protocolli)

STREAM client: `socket->connect->read/write->close`

STREAM server: `socket->bind->listen->accept->read/write ->close`

DATAGRAM client/server: `socket->bind->sendto/rcvfrom->close`

Memory mapped files: si mappa un file nella memoria

Algoritmi di elezione

Gli algoritmi di elezione sono usati in un contesto distribuito in cui più processi interagiscono tra di loro tramite un coordinatore. Se cade il coordinatore o cadono dei nodi, possono esserci dei problemi. In questo caso quando un processo "ritiene" che possa essere caduto un nodo, lancia una procedura di "elezione" per capire se il coordinatore è ancora funzionante, ed eleggere un nuovo coordinatore se quello precedente non lo è.

Si assume che ogni processo abbia la conoscenza degli ID degli altri processi (conosca la topologia della "rete" di processi). Inoltre si assume che i processi non siano informati dello stato degli altri processi. L'obiettivo degli algoritmi è "trovare accordo" tra i processi per decidere chi è il nuovo coordinatore.

Bully: L'ordinamento gerarchico è indotto dagli ID unici assegnati nella topologia ad ogni processo.

Quando un processo P ritiene che il coordinatore sia caduto, invia un messaggio ELECTION a tutti gli altri nodi con ID più alto. Se nessuno risponde, P è il nuovo coordinatore. Altrimenti se qualcuno risponde (P con ID più alti che ricevono un messaggio di elezione da un altro P più "basso" risponde), P "perde l'elezione".

Alla fine dell'algoritmo sarà rimasto un solo processo con ID massimo, che sarà coordinatore e informerà gli altri processi di questa decisione con un messaggio.

Un processo che si risveglia o che riparte come prima cosa deve lanciare un'ELEZIONE.

Ring: I processi sono fisicamente e/o logicamente ordinati. Ognuno di essi conosce qual è il suo vicino nella struttura ad anello.

Quando un processo P sospetta un coordinator fault, invia un ELECTION al suo successore nell'anello. Ogni membro dell'anello riceve il messaggio (che è propagato), e aggiunge il suo ID. Quando il P che ha lanciato l'ELECTION riceve il messaggio dal suo predecessore, allora lo trasforma in un messaggio COORDINATOR e lo ritrasmette, per informare gli altri su chi sia il coordinatore (ID più alto) e su che nodi siano ancora attivi.

Se il prossimo nodo nell'anello è morto, il messaggio si invia al successore del successore :)

Mutua esclusione distribuita (DME)

Ho una rete di n processi. Ogni processo risiede su un P_i processore. Ogni processo ha una sezione critica.

Si richiede che quando P_i sta eseguendo nella sua sezione critica, nessun altro P_j stia eseguendo nella sua sezione critica. I requisiti della DME sono gli stessi della ME in un contesto distribuito. Ci sono vari approcci alla soluzione del problema:

1. **Centralizzato:** un processo si fa carico di gestire i permessi d'accesso alla sezione critica per tutti gli altri. I processi mandano request, quando ricevono un reply dal coordinatore possono eseguire la sezione critica. Al termine notificano il manager con un release.

- Svantaggi: collo di bottiglia per le prestazioni. Inoltre se ho un crash del processo coordinatore è un problema.
- Vantaggi: assicura ME, ordinamento delle richieste, no starvation, basso traffico di rete (solo 3 messaggi per comunicare)

• **Distribuito:** ogni nodo ha visione parziale del resto della rete e deve decidere in base a quello. Ordinamento di eventi basato sul timestamping -tipo Lamport- (xchè funzioni processo Pi lo deve inviare a tutti i processi diversi da se stesso). Eventi con stesso timestamp sono ordinati in base al PID. Per garantire ME e assenza di Deadlock, tutti i processi devono essere informati di cosa avviene nel resto del sistema.

1. Pi vuole entrare in CS. Invia messaggio **request(Pi,TS)** a tutti gli altri processi.
2. Pj (per ogni j≠i) che riceve una request può rispondere con **reply** (subito o dopo).
3. Quando Pi riceve tutti i reply, allora entra in CS
4. Quando finisce di eseguire CS, invia **reply** per le richieste giunte nel frattempo dagli altri e alle quali non aveva ancora risposto

Come Pj decide quando inviare il reply?

1. Se Pj sta eseguendo CS, lo invierà alla fine dell'esecuzione di CS
2. Se Pj non vuole eseguire CS, risponde subito
3. Se Pj sta attendendo un reply da qualcun altro (ovvero vuole eseguire CS ma non ha ancora iniziato perchè mancano dei reply), confronta il TS della sua richiesta con il TS della richiesta esterna: se il suo è più alto, risponde con reply (Pi ha chiesto prima), altrimenti non risponde.

Vantaggi: assicuro assenza di deadlock e di starvation ("schedulazione" basata sul TS), oltre alla mutua esclusione.

Svantaggi: processi devono conoscere tipologia della rete (che può variare dinamicamente, complesso il refresh della topologia). Se uno dei processi cade, il sistema è compromesso (monitoraggio continuo dello stato dei processi). I processi che non sono in sezione critica devono interrompersi frequentemente per mandare il **reply**

Il protocollo è adatto in topologie di dimensioni ridotte con pochi processi stabili.

• **Token Ring:** esiste una rete logica associata in cui ogni processo ha una posizione e conosce il suo successore. Il token è un entità detenuta da un solo processo nel ring. Solo il processo che ha il token può entrare nella sua sezione critica senza fare richieste. Quando esce da CS, passa il token ad un altro processo. Quando un processo vuole entrare in CS, fa richieste agli altri per avere il token e attende finchè non glielo danno.

- Vantaggi: ME, No starvation. Worst Case: processo attende che tutti gli altri n-1 processi eseguano 1 volta CS.
- Svantaggi: difficile ritrovare/rigenerare token persi. Difficile rilevare un crash di un processo

Allocazione distribuita dei processori

Ho un set di processori su cui devo allocare un set di processi nel modo più efficiente. **Allocazione** può essere non migrante (processo allocato a processore X rimane lì finché non termina) oppure migrante (processo può essere riallocato in esecuzione ad un altro processore).

Obiettivi:

1. Massimizzare CPU execution
2. Minimizzare tempo medio di risposta
3. Minimizzazione rateo di risposta (tempo di esecuzione processo su una macchina diviso tempo di esecuzione su macchina benchmark dedicata a quel processo)
4. Bilanciare il carico
5. Migliorare comunicazione (trasferire sulla stessa macchina i processi che dialogano frequentemente)
6. Affidabilità (processi persistenti trasferiti su altre macchine in caso di shutdown della loro)
7. Specializzazione (un processo con richieste HW particolari può essere trasferito su un sistema adatto ai suoi scopi)

Tradeoffs dell'algoritmo:

Centralizzato VS Distribuito, Deterministico VS NonDet. ,Ottimo vs SubOttimo, sender-init VS reciever-init (riguardo alla locazione), **process-init VS OS-init** (chi inizia l'algor.)

L'idea base è di trasferire solamente le informazioni indispensabili, distruggendo il processo sul sistema source (spostando il PCB e aggiornando i links) e creandolo sul sistema destination.

Classificazione degli algoritmi:

1. **Basati su grafi deterministici:** richiede un'ottima conoscenza della topologia del sistema, che è rappresentato come un grafo con pesi sugli archi. Minimizzare il traffico. Trovare clusters con certe proprietà
2. **Centralizzati:** non richiede conoscenza topologia, è euristico. Coordinatore mantiene una tabella sull'utilizzo delle risorse, aggiornata da messaggi. Alla creazione di un processo, la workstation lo può spostare accumulando penalità (UP) nella tabella di utilizzo. Se non ci sono richieste, il suo valore di penalità nella tabella decresce (DOWN). Un valore alto identifica una ws "avida" di risorse, valori negativi identificano una ws a cui servono risorse. Quando un processore è libero, viene accettata la richiesta pendente che ha associato lo score più basso. Soluzione buona ma poco scalabile.
3. **Gerarchici:** si cerca di rimuovere il collo di bottiglia dato dalla centralizzazione. Ottengo un'organizzazione ad albero, con più livelli di managers. Un manager che riceve una richiesta e non ha risorse, la passa al livello più alto. Bisogna bilanciare bene quanti processori associare ad ogni manager (rischio di avere capacità sprecata oppure di sovraccaricare i nodi superiori)
4. **Euristici distribuiti:** un nuovo processo nato sulla macchina X manda un messaggio di probing a una macchina "a caso" e chiede se può spostarlo lì. Se la macchina è caricata sotto un certo limite risponde sì, altrimenti risponde no. X

ritenta la stessa cosa con altre macchine. Dopo N tentativi falliti, il processo rimane su X.

5. Bidding: le WS diventano "actors": compratori, venditori, prezzo basato sulla domanda/offerta. Il prezzo di un processore dipende dalle sue caratteristiche. I processi devono "comprare" il CPU time (ogni processo lancia un processo figlio che si preoccupa di controllare "il mercato" e sceglie le soluzioni più economiche, più veloci ... varie politiche!)

Strategie di migrazione: bisogna spostare PCB su macchina destinazione, aggiornare i links al processo che viene sostituito sulla macchina sorgente. Come gestire file aperti? E lo spazio di indirizzamento?

Address Space: strategie **eager all** (cancello e sposto tutto anche quello che non mi serve), **precopied** (il processo continua a eseguire su ws source finché non ho trasferito lo spazio di indirizz. -> devo poi aggiornare le pagine modificate nel frattempo), **eager dirty** (trasferisco solo porzione in memoria principale. Se servono altri blocchi, li invio on-demand--> remote memory paging.) **copy by reference** (si spostano solo le pagine per referenza) **flushing** (si tolgono le pagine dalla memoria principale flushandole su disco)

Criterio di scelta: trasferisco tutto se lo spostamento è definitivo, uso le politiche che lasciano roba sulla macchina source se il trasferimento è solo temporaneo.

Starter Utility: responsabilità di migrazione (e di long term scheduling e memory allocation) è adibita alla starter utility associata a ogni macchina. La migrazione ha luogo quando le 2 starter unit sono d'accordo. I sistemi possono anche decidere di rimandare indietro processi quando fregano troppe risorse e non c'è più disponibilità (**eviction**)

Sistemi distribuiti per architetture parallele

Sistemi multiprocessore: un sistema con più processori

Sistema multicomputer: più sistemi uniprocessore in cooperazione

Sistema multiprocessore a memoria condivisa: lettura/scrittura delle stesse locazioni di memoria. A parte aspetti di scheduling e sincronizzazione, sono simili a sistemi uniprocessore. I programmi vedono lo stesso spazio di indirizzamento virtuale

- **Architettura UMA/NUMA:** accesso uniforme/non uniforme a memoria
 - **UMA bus condiviso**
 - **UMA crossbar switch**
 - **UMA switch-network multistadio**
 - **NUMA:** indispensabili per collegare più di 100 processori.
- **Sistemi Operativi**
 - **OS specifico per ogni macchina**
 - **Master-Slave**

- **SMP**: symmetric multiprocessor system
- Sincronizzazione: non basta disabilitare gli interrupt, non basta istruzione TSL (test & set lock deve agire anche sul BUS)
- Scheduling:
 - **Time sharing**: singola tabella processi x tutto il sistema. Ogni processore libero esegue il prox processo pronto nella tabella (in base a qualche politica)
 - **Gang Scheduling**: eseguire in contemporanea i processi correlati. I membri di una gang (set di processi correlati) sono eseguiti in timesharing simultaneamente da più processori. I membri di una gang hanno quindi i time slices uguali.
 - **Space Sharing**: gruppo di **k** processi correlati ha un set di **k** processori assegnati.

Sistema multicomputer a scambio di messaggio: difficili da programmare

Sistema distribuito: sistemi multicomputer collegati in WAN

Sistemi Multicomputer: introdotti perchè al crescere delle CPU i Multiprocessore diventano costosi ed ingestibili. Multicomputer: sono normali calcolatori collegati in rete. Problema critico diventa realizzare efficientemente la rete, ma i tempi in gioco sono più grandi e quindi il problema è meno stringente.

- Topologia della rete: star, ring, toro, doppio toro, ipercubo, grid ...
- Strategie di switching: packet, circuit, wormhole routing
- Interfacce di rete: influiscono notevolmente sul SO. In genere dotate di RAM per garantire un flusso continuo di dati (eventualmente anche controller DMA e CPU)