-SISTEMI OPERATIVI:

1 Sistema operativo(S/O): è un insieme di utilità progettate per gestire le risorse fisiche e logiche di un elaboratore fornendo all'utente un'astrazione più semplice e potente, costituita da una macchina virtuale, un ambiente in cui la memoria è virtualizzata ed è possibile eseguire applicazioni senza particolari conoscenze. Il sistema operativo risiede o nella ROM(esempio nei macchinari industriali) o in una memoria secondaria per poi venire caricato in RAM, completamente o in parte, all'avvio della macchina.

2 Programmi e processi
Si dice programma un codice che esegue una o più istruzioni; un programma in esecuzione è detto, invece, processo. In un sistema operativo i processi (utente e di sistema) avanzano in modo concorrente(non simultaneamente) secondo diverse politiche di ordinamento, garanti della fairness, decise nello scheduling.

2.1 Risorse Una risorsa è un qualsiasi elemento fisico o logico necessario alla creazione, esecuzione ed avanzamento di processi e può essere:
- Durevole o consumabile;
- Ad accesso atomico o divisibile;
- Ad accesso individuale o molteplice;

Tipologie di risorse:
- Risorse CPU: è la risorsa più importante perché indispensabile per l'avanzamento.
- Risorse di memoria: come la lettura(accesso multiplo di più processi), la scrittura(individuale) e la virtualizzazione della RAM per renderla più grande e prerilasciabile grazie alla creazione di registri virtuali che si riferiscono a quelli fisici.
- Risorse I/O: sono riutilizzabili e non prerilasciabili per via della loro funzione(esempio stampante) e della loro lentezza.

Le risorse del tipo I/O sono gestite da un programma specifico, il BIOS.

2.2 Stati di avanzamento dei processi.

un processo disattivato viene caricato in memoria tramite una chiamata di sistema che crea una struttura dati detta Process Control Block(PCB), entra in stato di pronto, va inesecuzionein base alla priorità attribuitagli, dove può terminare, essere prerilasciato(o sospeso)per dare spazio ad un altro processo, o messo inattesa. Stati di avanzamento

2.3 Gestione dei processi.
La gestione dei processi è affidata al nucleo del sistema operativo, il kernel il quale:
- Grazie al dispacer esegue Il componente che gestisce gli scambi tra i processi in esecuzione è il dispat-cher.

La scelta dei processi da eseguire e l'ordine e affidata allo scheduler.
I processi in stato di pronto sono accodati in una struttura detta lista dei pronti(ready list), spesso del tipo First-Come-First-Served(FCFS).I processi possono essere CPU-bound, attività dalla durata molto lunga, o I/O-bound, comprendenti attività di breve durata sulla CPU ed altre di I/O molto lunghe. Quest'ultimi sono

penalizzati dalla tecnica FCFS, motivo per cui vengono messi a disposizione di ogni processo uguali quanti di tempo (round-robin) e vengono istituite code diverse in base alla priorità o alla categoria dei vari processi.

Sincronizzazione dei processi:

meccanismi per la gestione delle risorse condivise da più processi, per esempio un variabile che più processi devono modificare, il problema arriva quando c'è un ordine da rispettare affinché il tutto vada come programmato ed esca il risultato sperato.

La race condition: è quando il risultato dipende molto dalla corretta sincronizzazione tra i processi nel loro accesso e modifica della risorsa, la **sezione critica** è la parte del codice dove se trova l'accesso alla risorsa condivisa tra processi concorrenti.

Il modo più semplice per risolvere questo problema è l'uso di un **LOCK** (lucchetto) detto anche **MUTEX** non permette a nessun'altro processo di accedere alla risorsa mentre è in uso dal processo che ha messo il lock.

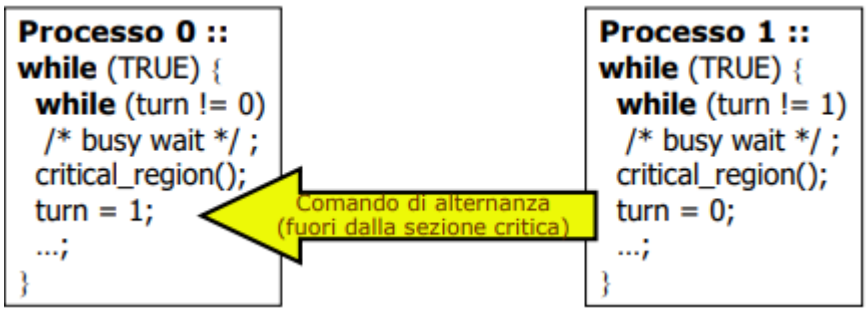
```
/* i processi A e B devono accedere
** a X ma prima devono verificarne
*/ lo stato di libero
if (lock == 0) { // variabile "lucchetto"
   /* X è già in uso:
   */ occorre ritentare con while do
}
else {
// X è libera, allora va bloccata ...
   lock = 0;
   ... // uso della risorsa
// ... e nuovamente liberata dopo l'uso
   lock = 1;
}
```

Questo metodo non è efficace perché la risorsa può essere rilasciata prima dopo il lock e prima di modificare la risorsa. Questo causa ancora race condition in più busy wait visto che l'altro processo rimane in attesa che la risorsa venga rilasciata e quindi occupa il processore per niente.

Una soluzione per la sincronizzazione tra processi deve soddisfare 4 punti per essere ammissibile:

- Garantire accesso esclusivo.
- Garantire attesa finita(no busy wait infinito).
- Non fare assunzioni sull'ambiente di esecuzione.
- Non subire condizionamenti dai processi esterni alla sezione critica.

Soluzioni esotiche #1



Uso di variabile condivisa tra i due processi per sincronizzare l'accesso alla memoria cambio alternato della variabile turn in 0 e 1.

PROBLEMI: presenza di busy wait(perché così entrambi i processi sono in esecuzione e quindi tolgono potenza e tempo alla CPU senza motivo), Condizionamento esterno della sezione critica, Rischio di race condition sulla variabile condivisa stessa.

Soluzione esotica #2 G. L Peterson:

```
IN(int i) :: {
int j = (1 - i); // l'altro
flag[i] = TRUE;
turn = i;
while (flag[j] && turn == i)
 { // attesa attiva  };
OUT(int i) :: {
flag[i] = FALSE; }
```

```
Processo (int i) ::

while (TRUE) {
  IN(i);
  // sezione critica
  OUT(i);
  // altre attività
}
```

Benché migliore della precedente è applicabile solo a copie di processi quindi non adatta ai processori multicore odierni. Ognuno dei processi vede se stesso come *processo i* e l'altro come *processo j*, i sarà uguale solo a 0 o 1 e mettendo *j=1-i* allora j è uguale a 0 quando i è 1 e viceversa. Ogni processo lascerà passare l'altro quando vedrà *flag[j]=1 e turn come se stesso,* per un processo, al momento del controllo corrisponderà a 0 e per l'altro a 1. La funzione **IN()** mette in attesa il processo *i* finché il processo *j* non viene rilasciato.

"*anche l'altro processo ha richiesto l'accesso e io sono arrivato per secondo visto che turn ha il mio identificativo (ho quindi sovrascritto l'identificativo dell'altro*".

Su *flag*[] non vi può essere scrittura simultanea come invece c'è su *turn* e la condizione in uscita è esppressa in modo da evitare race condition.(ESEMPIO SU SLIDE sincronizzazioneprocezzi.pdf Pag. 13-14).

TECNICHE ATERNATIVE E/O COMPELEMENTARI DI SINCRNIZZAZIONE:

1- Disabilitazione delle interruzioni:
   a. Risolve il problema prevenendo il prerilascio della risorsa quando è in uso da un processo.
   b. Inaccettabile in sistemi che prevedono molte interruzioni.
   c. Gli interrupt non devono essere lasciati all'utente.
   d. Non funzionale con 2 processori.
2- Test-and-set-Lock(supporto **hardware**):
   a. Cambiare automaticamente il valore della variabile lock se e libera.

```
enter_region

!! regione critica: la zona di programma che delimita
!! l'accesso e l'uso di una variabile condivisa

leave_region
```

```
enter_region:
                    !! Copia il valore di LOCK in R1
 TSL R1, LOCK       !! e pone LOCK = 1 (bloccato)
                    !! inoltre, blocca memory bus

 CMP R1, 0          !! verifica se LOCK era 0 tramite R1
 JNE enter_region   !! attesa attiva se R1==0
 RET                !! altrimenti ritorna al chiamante
                    !! con possesso della regione critica

leave_region:
 MOV LOCK, 0        !! scrive 0 in LOCK (accesso libero)
 RET                !! ritorno al chiamante
```

Tutte queste soluzioni(Peterson e TSL) hanno in comune il difetto della **busy wait** e della **inversion priority** , dati due processi H(alta priorità) e L(bassa priorità) supponiamo che H rilasci il processore per eseguire I/O e che venga fatto mentre L è in sezione critica, allora H rimarrà bloccato in busy wait perché L non ha modo di concludere la sezione critica senza processore.

Problema produttore-consumatore: due casi principali.

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item();                /* generate next item */
          if (count == N) sleep();              /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep();              /* if buffer is empty, got to sleep */
          item = remove_item();                 /* take item out of buffer */
          count = count - 1;                    /* decrement count of items in buffer */
          if (count == N - 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

- Inizia il *consumatore*, legge `count == 0` ma viene prerilasciato prima di eseguire l'istruzione `sleep()`; a quel punto, viene eseguito il *produttore*, che incrementa `count` (portandolo ad `1`) ma non sveglia il *consumatore*, poiché non addormentato, allora, quando il produttore verrà prerilasciato, il *consumatore* si addormenterà e la condizione `count == 1` non sarà mai vera, poiché il *produttore* continuerà ad incrementare `count` fino a `count == N`, quando si addormenterà, causando un deadlock;
- Analogamente, inizia il *produttore* e produce fino a `count == N` ma viene prerilasciato prima di eseguire l'istruzione `sleep()`; allora, il *consumatore* decrementerà `count` fino ad `N-1` e non sveglierà il *produttore*, poiché non addormentato, e consumerà tutti gli elementi del buffer per poi addormentarsi, causando un deadlock.

1- **soluzione sleep & wakeup:**
   a. le wakeup() svegliano le sleep() in attesa, le wakeup non sono memorizzate e se non vengono usate vanno perse. Questa soluzione però ha un problema sulla variabile *count*, se per esempio il buffer è vuoto e il consumer deve mettersi in sleep, però se lo scheduler decide di fermare il consumer,prima che si metta in sleep e passare al producer il quale mette count=1 ed emette una wakeup che però non viene ascoltata da nessuno è persa visto che non c'è nessuno sleep in ascolto e quindi la wakeup va persa; quando lo scheduler torna su consumer che non verrà mai svegliato perché la sua wakeup è stata persa.
2- **Soluzione mediante semaforo:**
   a. Uso di una variabile di controllo atomica(accesso indiviso) detta semaforo. Per questo questa soluzione di poggia su una macchina virtuale che fornisce un accesso indiviso.
      i. Ci sono due tipi di semafori, **binario(**detto **Mutex** permette l'accesso alla regione critica ad un solo processo alla vota e **contatore** che invece permette l'accesso simultaneo a più processi.
   b. P() oppure down() è la funzione che decrementa il contatore se non è già a 0 altrimenti mette in coda il processo. Serve per l'accesso alla risorsa.
   c. V() oppuere up() incrementa il contatore di 1 e chiede al dispatcher di mettere in pronto il processo successivo. Serve per il rilascio della risorsa.

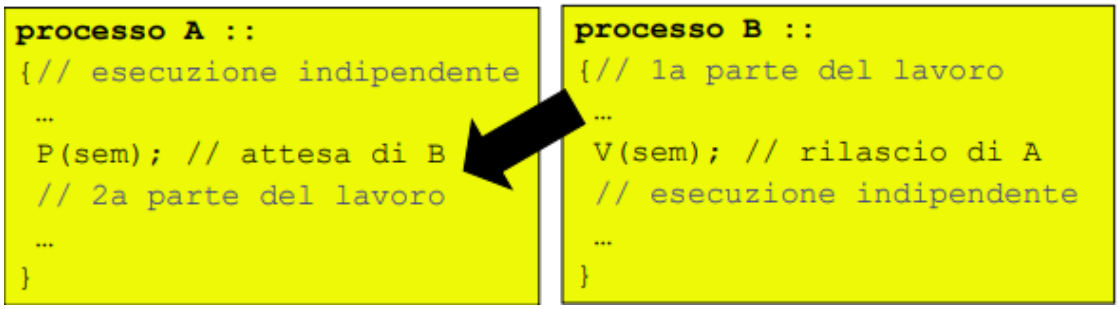
```
Processo Pi ::
{ // avanzamento
  P(sem);
  /* uso di risorsa
     condivisa R */
  V(sem);
  // avanzamento
}
```

P(sem) viene invocata per richiedere accesso a una risorsa condivisa R

- Quale R tra tutte?

V(sem) viene invocata per rilasciare la risorsa

- I semafori possono essere usati anche per coordinare l'esecuzione di processi che si influenzano tra di loro; per esempio far continuare un processo solo se viene prima eseguita una presente in un altro processo .



- Il semaforo **Mutex** è composto da un campo intero(0 o 1) e da una coda di PCB dei processi in attesa che vengono poi gestiti dallo scheduler al suo richiamo.

```
void P(struct sem){
  if (sem.valore = 1)
     sem.valore = 0; // busy
  else {
     suspend(self, sem.coda);
     schedule();}
}
void V(struct sem){
  sem.valore = 1 ; // free
  if not_empty(sem.coda){
     ready(get(sem.coda));
     schedule();}
}
```

Funzionamento di V() e P() in semaforo mutex.

- Il semaforo contatore ha la stessa struttura del mutex ma usa una logica diversa per il campo valore che inizializzato con la capacità massima della risorsa.

```
void P(struct sem){
  sem.valore -- ;
  if (sem.valore < 0){
     suspend(self, sem.coda);
     schedule();}
}
void V(struct sem){
  sem.valore ++ ;
  if (sem.valore <= 0){
     ready(get(sem.coda));
     schedule();}
}
```

Funzionamento di V() e P() in semaforo contatore.

Monitor:

l'uso dei semafori però è difficile e rischioso e in caso di errata implementazione si possono causare situazioni di deadlock o perfino di *race condition*(quello per cui sono stati creati), per questo è sconsigliato lasciare all'utente l'implementazione di strutture così delicate. Esempio di uso errato di semafori:

```
#define N …                /* posizioni del contenitore   */
typedef int semaforo;      /* P decrementa, V incrementa,
                              il valore 0 blocca la P     */
semaforo mutex = 1;
semaforo non-pieno = N;
semaforo non-vuoto = 0;

void produttore(){
   int prod;
   while(1){
      prod = produci();
      P(&non-pieno);
      P(&mutex);
      inserisci(prod);
      V(&mutex);
      V(&non-vuoto);
   }
}

void consumatore(){
   int prod;
   while(1){
      P(&non-vuoto);
      P(&mutex);
      prod = preleva();
      V(&mutex);
      V(&non-pieno);
      consuma(prod);
   }
}
```

```
#define N …                /* posizioni del contenitore   */
typedef int semaforo;      /* P decrementa, V incrementa,
                              il valore 0 blocca la P     */
semaforo mutex = 1;
semaforo non-pieno = N;
semaforo non-vuoto = 0;

void produttore(){
   int prod;
   while(1){
      prod = produci();
      P(&mutex);
      P(&non-pieno);
      inserisci(prod);
      V(&mutex);
      V(&non-vuoto);
   }
}

void consumatore(){
   int prod;
   while(1){
      P(&non-vuoto);
      P(&mutex);
      prod = preleva();
      V(&mutex);
      V(&non-pieno);
      consuma(prod);
   }
}
```

In questo caso il mettere prima P(mutex), che da accesso esclusivo al contenitore, può causare problemi perché il produttore() potrebbe essere prerilasciato e non permettere al consumatore di verificare se ci sono risorse nel contatore. I linguaggi ad alto livello offrono strutture esplicite di controllo delle regioni critiche, il monitor definisce una regione critica e il compilatore mette il codice necessario per il controllo degli accessi.

I **monitor** sono aggregati ti sottoprogrammi, variabili e strutture dati, solo i sotto programmi possono accedere alle variabili e solo un processo alla volta può essere attivo dentro il processo. Questo lavoro viene eseguito dal compilatore durante l'esecuzione.

La mutua esclusione però non basta per garantire la sincronizzazione dei processi, vengono usate delle condition varibles, *Wait() // forza l'attesa del chiamante* e *Signal() // risveglia il processo in attesa* , il segnale di risveglio non ha memoria quindi va perso se non usato.

ESEMIO DI USO DI MONITOR:

```
monitor PC
 condition non-vuoto, non-pieno;
 integer contenuto := 0;
 procedure inserisci(prod : integer);
 begin
  if contenuto = N then wait(non-pieno);
  <inserisci nel contenitore>;
  contenuto := contenuto + 1;
  if contenuto = 1 then signal(non-vuoto);
 end;
 function preleva : integer;
 begin
  if contenuto = 0 then wait(non-vuoto);
  preleva := <preleva dal contenitore>;
  contenuto :=  contenuto - 1;
  if contenuto = N-1 then signal(non-pieno);
 end;
end monitor;
```

```
procedure Produttore;
begin
 while true do begin
  prod := produci;
  PC.inserisci(prod);
 end;
end;
```

```
procedure Consumatore;
begin
 while true do begin
  prod := PC.preleva;
  consuma(prod);
 end;
end;
```

- Con l'uso di Wait() e Signal() i due processi, produttore e consumatore, si alternano la risorsa; in questo caso un buffer dove andranno a scrivere e leggere alternandosi. Wait e Signal sono gestiti in mutua esclusione quindi non c'è race condition.

MESSAGE PASSING:

I monitor non possono essere utilizzati per sincronizzare lo cambio di informazioni tra elaboratori perché non c'è modo per usare lo stesso monitor su macchine diverse e quest'ultime utilizzano memorie distinte dove memorizzare le variabili.

Il message passing permette grazie a due funzioni: *send(destination, &mag) e recive(source, &mag)* di scambiare informazioni in modo sincronizzato tra elaboratori distinti. Con l'utilizzo di un buffer condiviso per mettere i dati.

ESEMPIO PRODUTTORE CONSUMATORE CON MESSAGE PASSING:

```
#define N 100                                   /* number of slots in the buffer */

void producer(void)
{
     int item;
     message m;                                 /* message buffer */

     while (TRUE) {
          item = produce_item( );               /* generate something to put in buffer */
          receive(consumer, &m);                /* wait for an empty to arrive */
          build_message(&m, item);              /* construct a message to send */
          send(consumer, &m);                   /* send item to consumer */
     }
}
```

```
void consumer(void)
{
     int item, i;
     message m;

     for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
     while (TRUE) {
          receive(producer, &m);          /* get message containing item */
          item = extract_item(&m);        /* extract item from message */
          send(producer, &m);             /* send back empty reply */
          consume_item(item);             /* do something with the item */
     }
}
```

BARRIERE:

Un altro modo per sincronizzare gruppi di processi che devono cooperare tra di loro per un risultato finale è l'uso di barriere. Raggiunto un punto di esecuzione il processo viene bloccato finché non ci arrivano anche tutti gli altri processi del gruppo. Questa tecnica si può applicare sia ad ambiente locale che distribuito(elaboratori distinti) ed evita la necessita di scambio di messaggi.

PROBLEMI DI SINCRONIZZAZIONE:

Un modo per verificare l'efficacia di un meccanismo di sincronizzazione è la sua applicazione a problemi particolari:

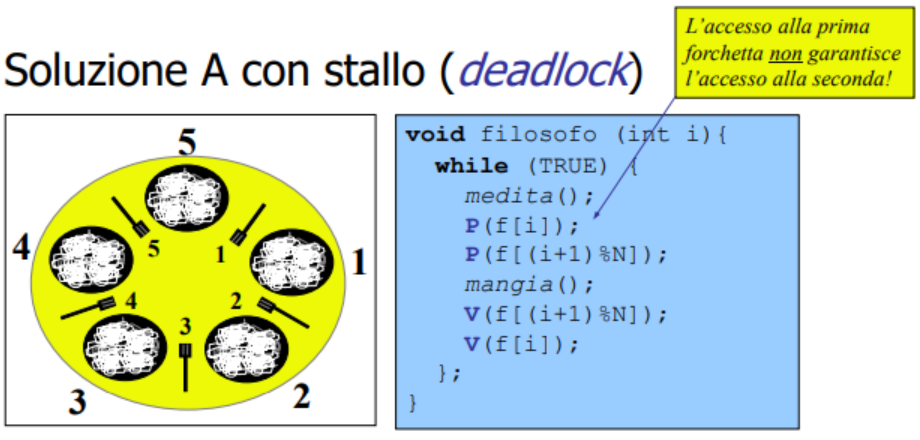
- Filosofi a cena: accesso esclusivo a risorse limitate.
- Lettori e scrittori: accesso concorrente a basi di dati.
- Barbiere che dorme: prevenzione di race condition.
- Produttore consumatore.

Le situazioni di rischio da evitare sono.

- Deadlock: stallo con blocco del programma.
- Starvation: stallo senza blocco, esp. Un solo processo non accede ad una risorsa.
- Race condition: esecuzione in ordine errato quindi da risultati diversi dalle aspettative.

Filosofi a cena:

dei filosofi sono seduti intorno ad un tavolo rotondo ogni filosofo ha una forchetta alla sua destra e condivide quella di sinistra, o vice versa. Ogni filosofo dopo avere pensato mangia e per mangiare prendere due forchette, ciò implica che quelli alla sua destra e sinistra non possono mangiare in contemporanea perché non hanno due posate.



Ogni forchetta modellata come un semaforo binario

Soluzione B con stallo (*starvation*)

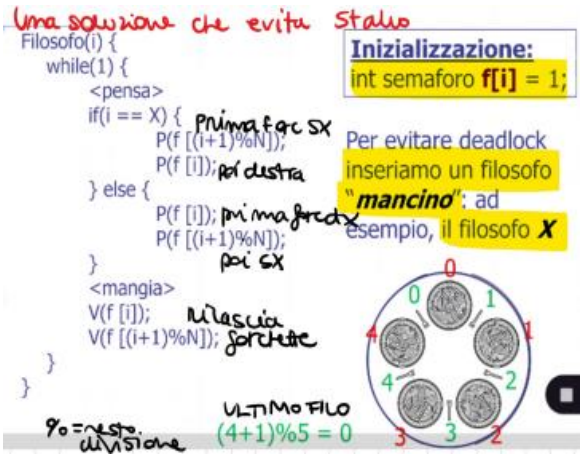
Errato: f contiene semafori che sono strutture dati

```
void filosofo (int i){
while (TRUE) {
  OK = FALSE;
  medita();
  while (!OK) {
   P(f[i]);
   if (!f[(i+1)%N]) {
    V(f[i]);
     sleep(T);}
   else {
    P(f[(i+1)%N]);
    OK = TRUE;
   };
  mangia();
  V(f[(i+1)%N]);
  V(f[i]);
}
```

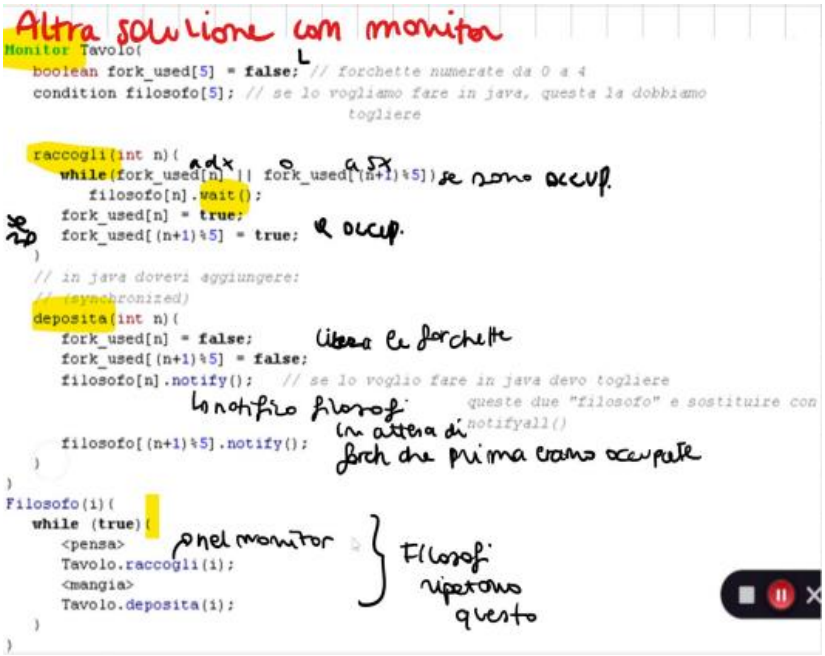
Un'attesa a durata costante difficilmente genera una situazione differente

Queste soluzioni possono essere migliorate in diversi modi per eliminare i problemi che hanno.

1.Utilizzare in soluzione A un semaforo a mutua esclusione per incapsulare gli accessi a entrambe le forchette (Funzionamento garantito).



2.In soluzione B, ciascun processo potrebbe attendere un tempo casuale invece che fisso • (Funzionamento non garantito).



3.Algoritmi sofisticati, con maggiore informazione sullo stato di progresso del vicino e maggior coordinamento delle attività (Funzionamento garantito).

STALLO:

Cause:

- Accesso esclusivo a risorse condivisa: risorsa bloccata da un processo che la usa.
- Accumulo risorse: i processi accumulano risorse senza rilasciarle.
- Inibizione di prerilascio: la decisione di rilascio della risorsa è lasciata al processo.
- Condizione di attesa circolare: attesa di rilascio di una risorsa da parte di un processo che ne attende un'altra.

Soluzioni:

- Prevenzione: impedire le cause elencate.
- Riconoscimento e recupero: riconoscere e sbloccare uno stallo con una procedura di recupero.
- Indifferenza: non prendere provvedimenti quando si verifica.

Bisogna impedire il verificarsi di almeno una delle condizioni necessarie e sufficienti:

- Accesso esclusivo alla risorsa: alcune risorse non lo permettono.
- Accumulo di risorse: però alcuni processi ne richiedono l'uso simultaneo.
- Inibizione del prerilascio: però alcune risorse non permettono di farlo.
- Attesa circolare: difficile da rilevare e sciogliere.

Prevenzione dello stallo:

- Prevenzione sulle richieste di accesso
    - A tempo di esecuzione: si verifica se ogni richiesta possa provocare stallo, molto onerosa e non ha uno schema fisso di risoluzione.
    - Prima dell'esecuzione: ordinamento delle risorse da usare prima dell'esecuzione.

Riconoscimento dello stallo:

- A tempo di esecuzione: lavoro molto oneroso, blocca di continuo il sistema per analizzare lo stato dei processi e verificare quelli successivi. Lo sblocco di uno stallo comporta la terminazione di un processo in attesa per liberare le risorse necessarie.
- Staticamente: si può arrivare prima dell'esecuzione a capire che lo stallo non è risolvibile.

RISOLUZIONE DI ALCUNI PROBLEMI:

-Problema lettori scrittori: un database deve essere letto e scritto, più lettoti possono leggere in contemporanea mentre gli scrittori solo quando non c'è nessuno a leggere o scrivere.

Risoluzione con semafori:

```
typedef int semaphore;              /* use your imagination */
semaphore mutex = 1;                /* controls access to 'rc' */
semaphore db = 1;                   /* controls access to the database */
int rc = 0;                         /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {                  /* repeat forever */
        down(&mutex);               /* get exclusive access to 'rc' */
        rc = rc + 1;                /* one reader more now */
        if (rc == 1) down(&db);     /* if this is the first reader ... */
        up(&mutex);                 /* release exclusive access to 'rc' */
        read_data_base( );          /* access the data */
        down(&mutex);               /* get exclusive access to 'rc' */
        rc = rc - 1;                /* one reader fewer now */
        if (rc == 0) up(&db);       /* if this is the last reader ... */
        up(&mutex);                 /* release exclusive access to 'rc' */
        use_data_read( );           /* noncritical region */
    }
}

void writer(void)
{
    while (TRUE) {                  /* repeat forever */
        think_up_data( );           /* noncritical region */
        down(&db);                  /* get exclusive access */
        write_data_base( );         /* update the data */
        up(&db);                    /* release exclusive access */
    }
}
```

Con il semaforo mutex gestisco la mutua esclusione sulle variabili condivise

Con il semaforo db gestisco l'alternanza lettori-scrittori sul database; solo il primo lettore di una sequenza deve passare attraverso questo semaforo

29

-Problema del barbiere sonnolento: un barbiere dorme se non ci sono clienti, sleep(), il primo cliente sveglia il barbiere che si mette a lavoro i clienti successivi si siedono su N sedie, se sono tutte occupate vanno via.

Risoluzione con semafori:

```
#define CHAIRS 5                    /* # chairs for waiting customers */

typedef int semaphore;              /* use your imagination */

semaphore customers = 0;            /* # of customers waiting for service */
semaphore barbers = 0;              /* # of barbers waiting for customers */
semaphore mutex = 1;                /* for mutual exclusion */
int waiting = 0;                    /* customers are waiting (not being cut) */

void barber(void)
{
    while (TRUE) {
        down(&customers);           /* go to sleep if # of customers is 0 */
        down(&mutex);               /* acquire access to 'waiting' */
        waiting = waiting - 1;      /* decrement count of waiting customers */
        up(&barbers);               /* one barber is now ready to cut hair */
        up(&mutex);                 /* release 'waiting' */
        cut_hair( );                /* cut hair (outside critical region) */
    }
}


void customer(void)
{
    down(&mutex);                   /* enter critical region */
    if (waiting < CHAIRS) {         /* if there are no free chairs, leave */
        waiting = waiting + 1;      /* increment count of waiting customers */
        up(&customers);             /* wake up barber if necessary */
        up(&mutex);                 /* release access to 'waiting' */
        down(&barbers);             /* go to sleep if # of free barbers is 0 */
        get_haircut( );             /* be seated and be serviced */
    } else {
        up(&mutex);                 /* shop is full; do not wait */
    }
}
```

ORDINAMENTO DEI PROCESSI:

Nella coda dei pronti troviamo dei puntatori che riferiscono alle caselle di una tabella con i PCB dei processi in coda, ogni processo ha un proprio descrittore detto PCB che specifica le seguenti caratteristiche:

- Identificatore del processo.
- Contesto di esecuzione del processo: (tutte le info per ripristinare lo tato di un processo dopo una sospensione).
- Stato di avanzamento del processo: (puntatore ad una lista dei processi nello stesso stato).
- Priorità : precedenza rispetto agli altri processi, prima e durante l'esecuzione.
- Diritti di accesso alle risorse e ad alcuni previlegi.
- Discendenza famigliare: collegamento con PCB di eventuali genitori o figli.
- Puntatore alle risorse assegnategli.

Il PCB relaziona il processo alla sua macchina virtuale.

Lo scheduling dei processi:

Le decisioni di scheduling sono molto importanti affinché non si verifichino errori; quando si crea un processo(se è padre o figlio), alla sua terminazione(sapere chi deve andare al suo posto), quando si blocca(sapere se è un semaforo, l'attesa di una risorsa o di un I/O) e quando c'è un interrupt I/O.

Ci sono diversi modi per decidere come alternare i processi in esecuzinoe:

- Scambio cooperativo: sta al processo in esecuzione la decisione di lasciare o meno la risorsa, non molto conveniente perché il processo non ha un quadro generale.
- Scambio a prerilascio: il processo in esecuzione viene rimpiazzato da uno con maggiore importanza(sistemi real time) o quando finisce il suo quanto di tempo(sistemi interattivi). Questo metodo necessita di clock e tramite un meccanismo esterno che conta il temo e di un software che identifica l'interruzione e la notifica allo scheduler se necessario.

Lo **scheduler** è il componente che decide l'ordinamento dei processi, è progettato prima dei processi che deve gestire e usa metriche diverse a seconda del sistema che deve governare (Batch: no preemption, Interattivi: preemption, Real time: no preemption).

Bisogna quindi rendere il suo funzionamento parametrico, configurando opportunatamente i suoi attributi, rispetto a quelli specifici assegnati ai processi per poterlo utilizzare con applicazioni diverse.

Il **dispatcher** è il componente del attua le scelte di ordinamento dei processi, ovvero esegue gli ordini dello scheduler. Attua il **context switch,** salva il contesto del processo che esce, installa quello in entrata e gli da il controllo della CPU per essere eseguito.

Politiche e meccanismi di ordinamento:

l'applicazione influenza le politiche di ordinamento con i valori considerati dai meccanismi del nucleo(scheduler) per decidere l'ordinamento dei processi e l'attribuzione delle risorse ad essi.

L'efficienza delle tecniche si misura con:

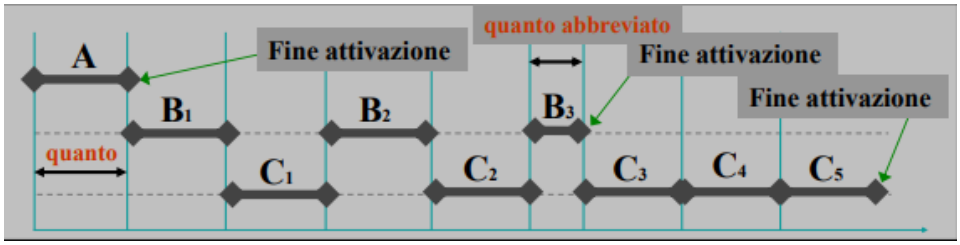
- Percentuale di utilizzo utile della CPU, ovvero che devono usarla per lo più i processi da eseguire che il nucleo(dispatchere e scheduler).
- Throughput: numero di processi avviati all'esecuzione per unità di tempo (produttività).

- Tempo di attesa: tempo che deve aspettare il processo in coda di pronto.
- Turn-around: tempo di completamento del processo.
- Tempo di risposta: il tempo che il processo aspetta prima di essere avviato.

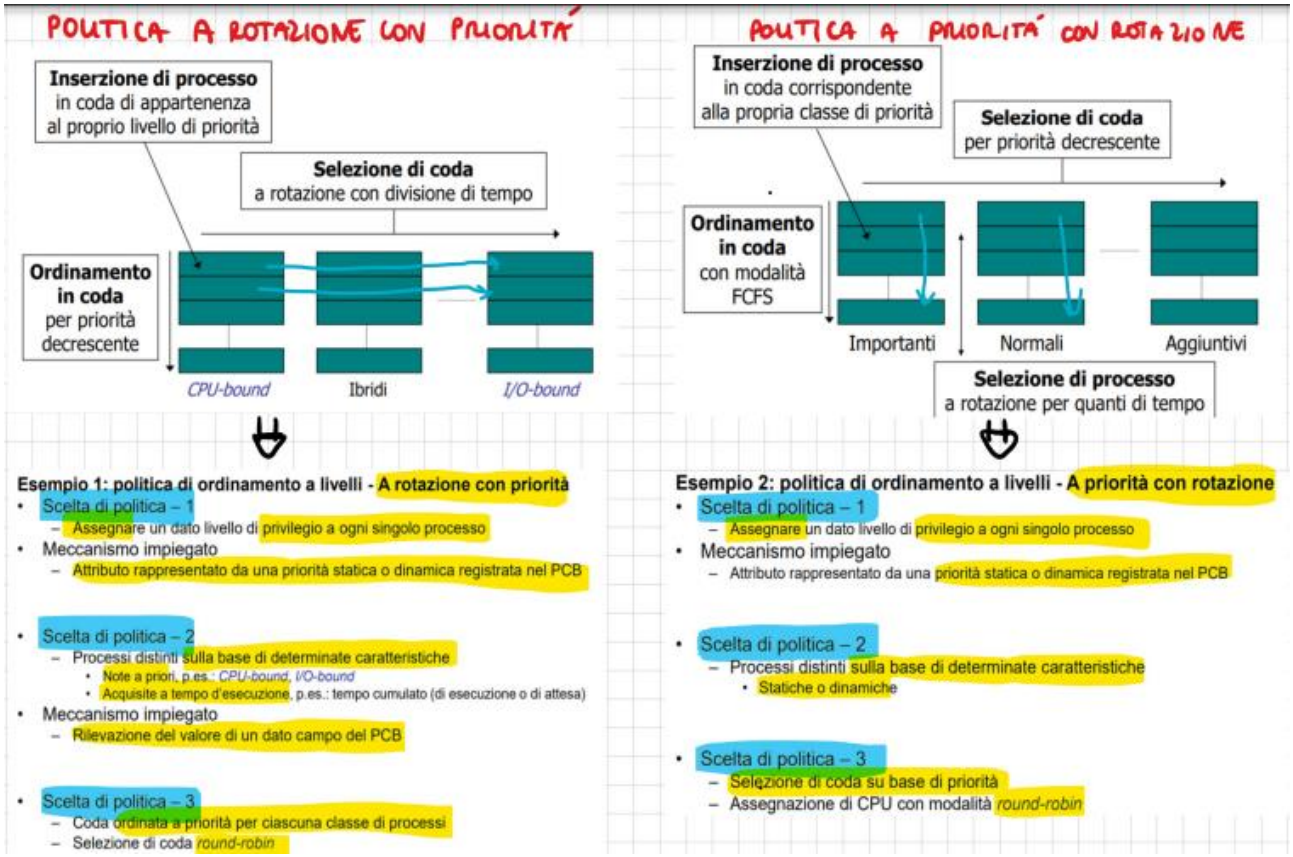
La **garanzia** di completamento dell'esecuzione di un processo dipende dalla politica di scambio usata. Quella che si cerca di rispettare di più è la *Fairness* (esp: lo scambio cooperativo non la garantisce).

I processi in stato di pronto sono messi in *ready list* (lista dei pronti); il modo più semplice per gestirla è la FCFS(first come first served) il primo che entra viene servito per primo, molto facile da realizzare ma in caso di processi lenti rallenta anche quelli dopo.

Una sua miglioria **round-robin** si ottiene con l'aggiunta di un quanto di tempo in cui il processo può lavorare e finito il quale deve cedere la risorse a quello successivo e si alternano finché non finiscono.



Con l'aggiunta anche di priorità si ottiene **la politica a rotazione con priorità e la politica a priorità con rotazione** dove i processi vengono divisi in gruppi a seconda delle priorità e selezione in round-robin con la divisione di tempo.



Politiche e meccanismi:

I **meccanismi** sono le sezioni del S/O che mettono in pratica le scelte di ordinamento e di gestione dei processi e risiedono nel nucleo; le **politiche** sono, invece, determinate fuori dal nucleo, nello spazio applicazioni.

Le caratteristiche che tutte le politiche devono rispettare sono:

- **Equità**: o *fairness*, ovvero la corretta distribuzione delle opportunità di esecuzione.
- **Coerenza**: o *enforcement*, applicazione della politica scelta nello stesso modo a tutti.
- **Bilanciamento**: utilizzo di tutte le risorse disponibile.

A queste si aggiungono altre caratteristiche da favoreggiare a seconda dal sistema usato:

- Tempo di risposta: tempo trascorso dall'entrata in coda fino all'inizio dell'esecuzione.
- Tempo di attesa: tempo passato in coda.
- Tempo di esecuzione: durata effettiva del lavoro.
- Tempo di turn-around: tempo di attesa +tempo di esecuzione.

Classifica dei meccanismi:

- Sistemi a lotti (*batch*): utilizza un ordinamento predeterminato con lavori poco urgenti e di lunga esecuzione ciascuno, senza prerilascio perché l'obbiettivo della politica è impiegare meno tempo per singolo lavoro(turn-around).
- **Sistemi interattivi**: gradi attività in contemporanea, prerilascio essenziale per fornire un tempo di risposta più corto e dare una sensazione di velocità all'utente.
- **Sistemi a tempo reale**: lavori brevi e urgenti con prerilascio solo se necessario, l'obiettivo è rispettare le scadenze dei processi (*deadline).*

Ogni sistema ha applica delle politiche diverse:

- **Sistemi a lotti**:
  - **FCFS**: *first come first served*, senza priorità o prerilascio ma basso utilizzo delle risorse.
  - **SJF**: *shortest job first,* richiede di sapere il tempo di esecuzione dei processi e li esegue dal più corto, senza prerilascio ma non è equa verso chi arriva per primo.
  - **SRTN**: *shortest remaining time next,* utilizza il prerilascio e esegue il processo che necessita di meno tempo ancora per terminare, se arriva un processo più corto di quello in esecuzione lo si esegue per primo.

  In generale parliamo di lavori quando operiamo senza prerilascio e di processi quando operiamo con prerilascio.

- **Sistemi interattivi:**
  - **OQ:** ordinamento a quanti, utilizzo di un quanto di tempo in *round-robin* con prerilascio ma senza priorità.
  - **OQP:** ordinamento a quanti di priorità, utilizzo dei quanti in *round-robin* ma con l'aggiunta di priorità(quanti diversi a seconda della priorità).
  - **GP:** con garanzia per processo, prerilascio con promessa di una data quantità di tempo di esecuzione(1/n-processi) ed esegue prima il lavoro più penalizzato dalla garanzia con una verifica periodica o a eventi.
  - **SG:** senza garanzia, utilizzo di prerilascio e priorità, ogni processo riceve un dei numeri casuali a seconda della priorità(priorità più alta=più numeri), poi viene fatta un estrazione per decide chi prende la risorsa, ovvio che (più numeri = più probabilità di vittoria.
  - **GU:** con garanzia per l'utente, uguale a GP ma riferita agli utenti, processore di più processi.

- **Tempo reale (real time):** nei sistemi real time il risultato deve rispettare una scadenza, oltre alla quale il suo risultato vale di meno, niente o anche negativamente. Per questo lo scheduling deve offrire garanzia di completamento e quindi analizzare staticamente ovvero predicibile, il caso peggiore è quando tutti i processi sono pronti per eseguire all'istante iniziale.
    - **Modello semplice:(cyclyng executive)** insieme fissato di processi periodici on caratteristiche note, ogni processo e diviso in una sequenza ordinata di cui si consce la durata. L'ordinamento è come una sequenza di chiamate a procedure fino al loro completamento. Un ciclo maggiore (major cycle) racchiude l'invocazione a N cicli minori (minor cycle) di durata specifica e suddiviso in N processi che rientrano nella durata del ciclo.

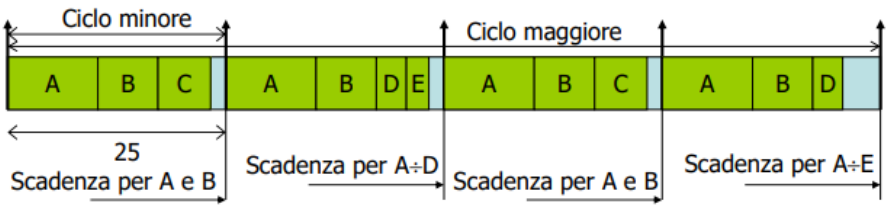
Processo	Periodo T	Durata C
A	25	10
B	25	8
C	50	5
D	50	4
E	100	2

Conviene che i periodi siano armonici!

$U = \Sigma_i (C_i / T_i) = 46/50 = 0.92$

Ciclo maggiore di durata 100 → MCM di tutti i periodi

Ciclo minore di durata 25 → periodo più breve



Se U < 1 possiamo eseguire tutti i processi.

    - **Ordinamento a priorità fissa scadenza uguale al periodo (D=T):** processi periodici indipendenti e noti gestiti con priorità e prerilascio se necessario. Assegnazione di priorità secondo il periodo(rate monotonic), priorità maggiore per periodo più breve, per controllare se è possibile si effettua un test, è sufficiente ma non necessario, con la seguente formula:

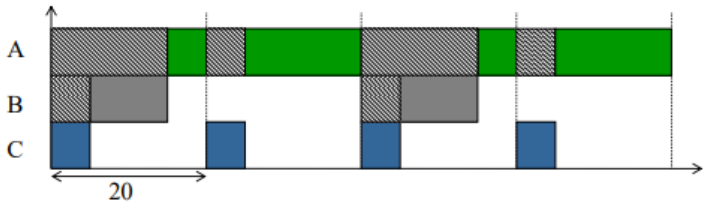
$$U = \sum_{i:=1}^{n} \left(\frac{C_i}{T_i} \right) \leq f(n) = n(2^{1/n} - 1)$$

n è il numero di processi indipendenti.

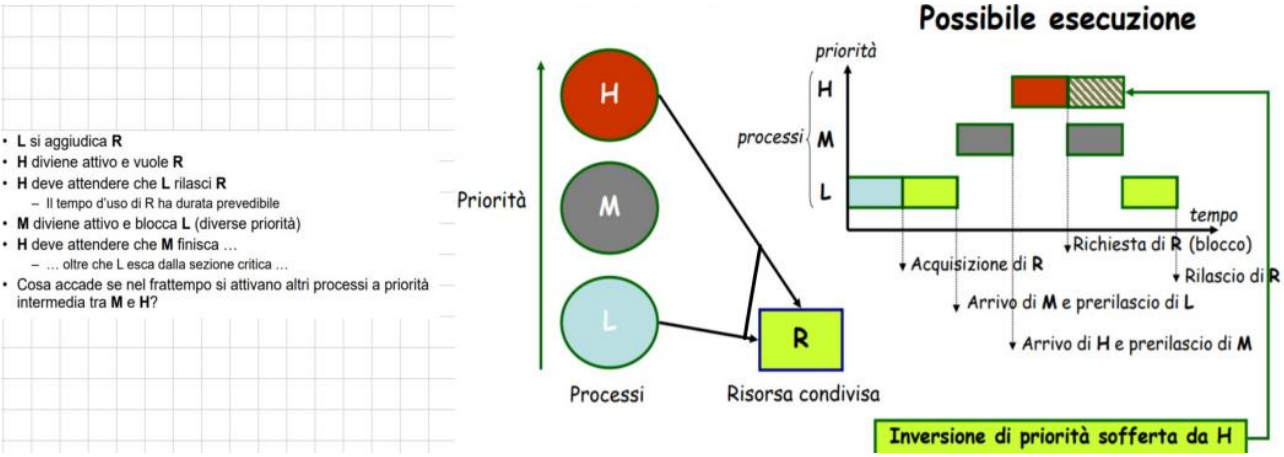
Caso semplice ordinamento a priorità

Processo	Periodo T	Durata C	Priorità
A	80	40	1 ← Bassa
B	40	10	2
C	20	5	3 ← Alta

Il test di ammissibilità fallisce $U = 1 > f(3) = 0{,}78$
ma il sistema è ammissibile!



- **Ordinamento a priorità fissa con prerilascio e scadenze inferiori a periodo (D<T):** assegnazione della priorità a seconda della scadenza del processo, però c'è rischio di *inversione di priorità* ovvero che processi di priorità maggiore siano bloccati dall'esecuzione di processi di priorità minore, questo è causato dall'eccesso esclusivo a risorse condivise e può condurre ad un blocco circolare(deadlock).

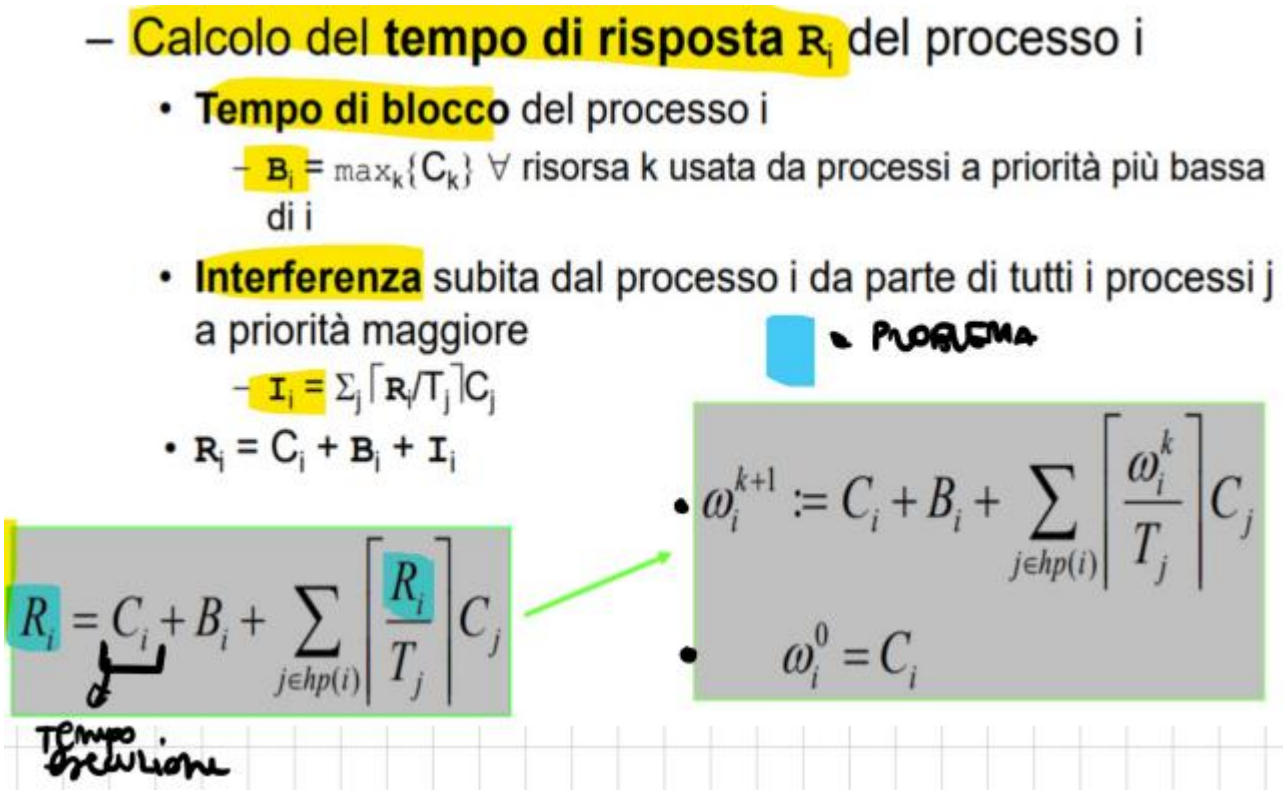


- **Soluzioni all'inversione di priorità:**
  - **Innalzamento delle priorità:** *basic priority inheritance*, la BPI non impedisce il deadlock, nel caso un processo a priorità maggiore si blocca nell'acceso ad una risorsa presa da uno con priorità inferiore la BPI permette di innalzare la priorità del processo che ha la risorsa per permettergli di terminare senza essere bloccato dalla politica di priorità.
  - **Innalzamento della priorità(versione avanzata):** *immediate ceiling priority* ogni processo ha una priorità statica PBj, e ogni risorsa i ha una priorità(ceiling) PCi uguale alla massima priorità che hanno i processi che possono usarla, ogni processo ha anche una priorità **dinamica** Pj = max{PBj, PCi} per ogni risorsa a cui vuole accedere. Un processo può accedere ad una risorsa solo se la sua priorità dinamica attuale è maggiore del *ceiling* di tutte le risorse attualmente in possesso di altri processi.

- La tecnica **IPC** evita il *deadlock*
- **Esempio IP2**
  - Consideriamo tre processi **L**, **M**, **H** con priorità crescente
  - Assumiamo che tutti condividano le risorse **R1** e **R2** (entrambe *Mutex*)
    - Il *priority ceiling* di **R1** e **R2** è superiore alla priorità di **H**
  - **L** acquisisce **R1** e ne assume il *ceiling* poi si accinge a richiedere **R2**
  - **H** diventa pronto a questo istante e vorrebbe prerilasciare **L**
    - Ma non può perché la priorità di **H** non è superiore al *ceiling* di **R1** e **R2**
  - Quindi **H** resta pronto ma non riesce a prerilasciare **L**
  - **L** acquisisce anche **R2** e poi prosegue fino a rilasciare **R1** e **R2**
    - La priorità di **L** ritorna al valore originale
  - **H** ha ora priorità maggiore di **L** e di ogni altro eventuale **M**
    - **H** può acquisire **R2** proseguire e completare
- La tecnica **IPC** impedisce il formarsi di catene di blocchi …
  - I processi $\{H\}_i$ subiscono al più 1 blocco da parte di 1 processo L in possesso di risorsa R condivisa con {H}
    - Blocco = ritardo nel primo prerilascio

Esercizio con inversion priority risolto con ICP/IPC.

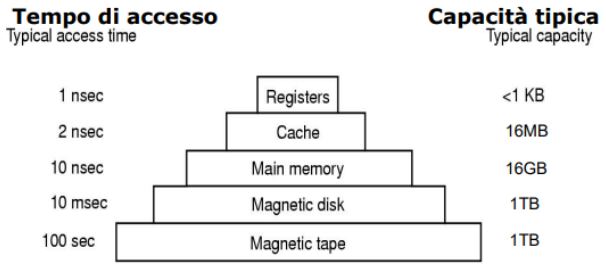
Calcolo del tempo di risposta:



-ESERCIZI ORDINAMENTO: *PDF SITO PALAZZI: https://www.math.unipd.it/~cpalazzi/sistemioperativi.html*

-GERARCHIE DI MEMORIA:

gerarchia fisica di memoria: esistono diverse tipologie di memoria dentro la macchina utilizzate per scopi diversi.



I **registri** sono delle celle interne alla CPU di dimensione 32 o 64 bit a seconda dell'architettura del processore(32 o 64).

La **cache** è una memoria divisa in blocchi detti line, di solito di 64 B, e controllata da hardware. Ci sono vari livelli di cache: L1 dentro la CPU, L2 condivisa e così via. E varie tecniche di lettura e scrittura dei dati.

I dischi magnetici che sono usati prevalentemente per essere letti hanno una capienza enorme e un costo inferire alle memorie RAM ma un tempo di accesso peggiore. I dischi magnetici sono composti da una serie di dischi e da una testina attaccata ad un braccio che si muove che gli legge.

La loro evoluzione sono le SSD che sono più veloci, silenziose perché non hanno parti in movimento, senza frammentazione e più solide. Però hanno un costo per capacità più elevato e le celle si usurano con continua lettura e scrittura.

La soluzione migliore è un mix SSD+HDD il primo per sistema operativo e programmi di uso frequente mentre il secondo per file, media ecc.

Il **CMOS**: è una memoria che salva l'ora, le impostazioni del BIOS e da dove fare il boot del sistema operativo.

Interazione memoria-programma:

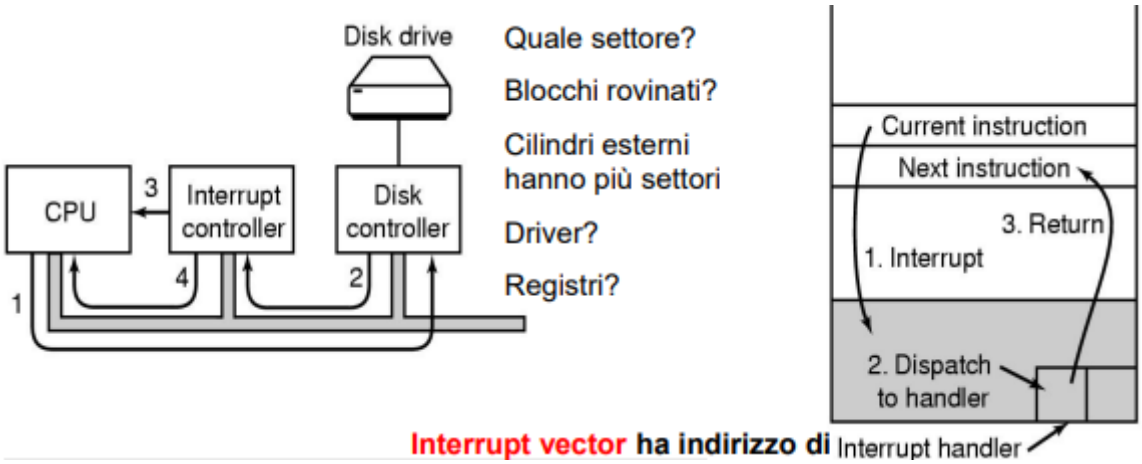
- Come proteggere i programmi tra loro e il kernel dai programmi
- Come gestire la rilocazione.
- Quando si compila un programma non si sa l'area di memoria in cui verrà caricato, soluzione: hardware con due registri base e limite o verifica continua di base+ indirizzo con però spreco CPU.

Vista logica della RAM:

la ripartizione della RAM tra due processi distinti utilizza due registri, base dove inizia la partizione utilizzata, e limite, dove finisce. L'allocazione del processo in RAM richiede la rilocazione della sua memoria virtuale. La gestione dello spazio virtuale dei processi utilizza un dispositivo **MMU**, memory managemanet unit, messo tra CPU e memoria e controllato dal sistema operativo.

Trattamento delle interruzioni:

1)driver dice a controller cosa fare -> 2)segnale finito su certe linee bus -> 3)imposta pin in CPU -> 4)mette nome dispositivo su bus.



Le interruzioni evita il ricorso al **polling**, l'interazione tipica avviene in 4 passi(*photo on top):*

- Il gestore del dispositivo programma il controllore di dispositivo scrivendo nei suoi registri di interfaccia.
- Il controllore agisce sul programma e poi informa delle interruzioni.
- Il controller delle interruzioni da un valore *pin* dei notifica alla CPU.
- Quando la CPU riceve la notifica il controller delle interruzioni comunica anche l'identità del dispositivo, in questo modo il trattamento delle interruzioni viene attribuito al gestore appropriato.

All'arrivo di una interruzione:

- I registri PC e PSW (program status word) sono messi nello stack del processo in esecuzione.
- La CPU passa *modalità operativa protetta.*
- Il parametro che denota l'interruzione serve come indice nel vettore delle interruzioni per poter individuare il gestore designato a servire l'interruzione.

- La parte immediata del gestore esegue nel contesto del processo interrotto, la parte meno urgente invece è differita e data ad un processo dedicato.

Plug & play: il sistema assegna i livelli di interrupt e di indirizzi di I/O e poi li rivela alle schede. Prima ogni scheda io aveva un proprio livello di interrupt fisso.

BIOS:

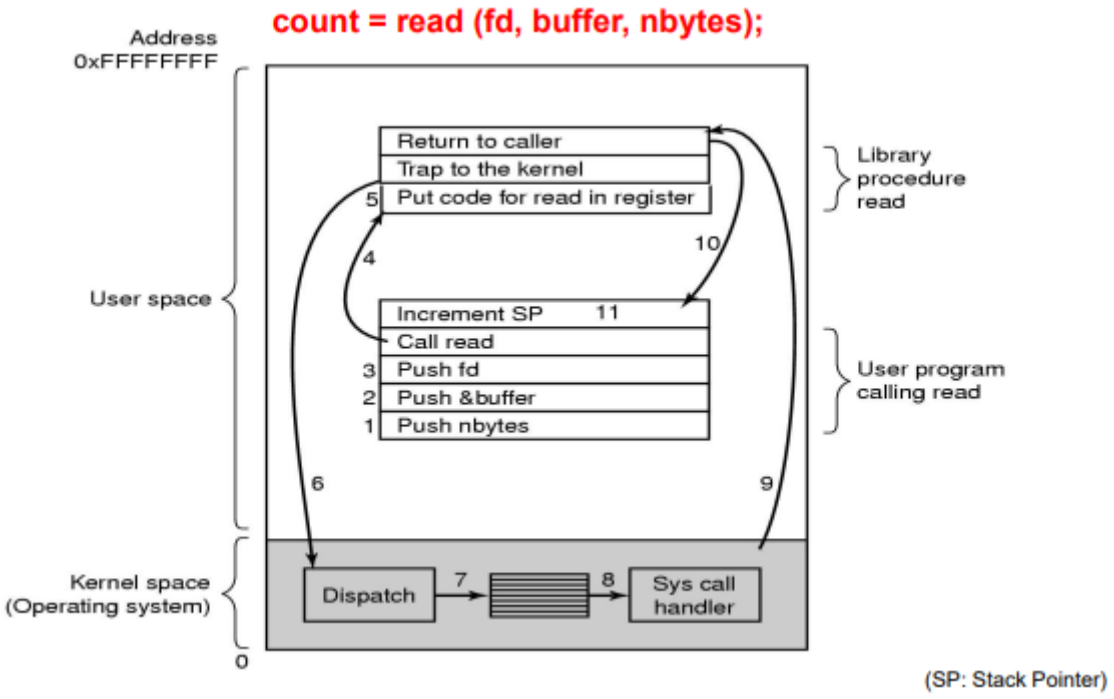
contiene SW a basso livello per la gestione degli I/O, viene caricato all'avvio del computer , verifica la quantità di RAM e i dispositivi collegati presenti, fa lo scan dei bus ISA e PCI per rilevare i dispositivi connessi ad essi, i dispositivi vecchi sono registrati e quelli nuovi vengono configurati, in più determina il boot, la memoria da cui caricare il SO, dalla lista in memoria CMOS.

BOOT:

Il primo settore del dispositivo di boot viene letto in memoria ed eseguito. Contiene un programma che esamina la tabella di partizione e determina quale partizione sia attiva dalla quale viene caricato un altro boot loader che legge ed esegue il sistema operativo che è presente nella partizione. Il sistema operativo quando viene caricato interroga il BIOS per ottenere informazioni sulla configurazione del sistema, per ogni dispositivo controlla l'esistenza dei driver e se non ci sono chiede di installarli, sennò gli carica nel kernel. Dopo di che esegue varie inizializzazioni e il programma iniziale.

CHIAMATE DI SISTEMA:

la maggior parte dei servizi del SO sono eseguiti in risposta a invocazioni esplicite di processi, chiamata di sistema. Le chiamate sono nascoste in procedure di libreria predefinite. Non è l'applicazione ad effettuare direttamente le chiamate di sistema ma la libreria che svolge il lavoro di preparazione necessario per l'invocazione. *TRAP* è la prima chiamata di sistema e attiva il modo operativo privilegiato, inizia l'esecuzione da un indirizzo prefissato dal kernel, il parametro della chiamata designa l'azione da svolgere e la convenzione per trovare gli altri parametri. Il meccanismo usato per trattare le chiamate è simile a quello delle interruzioni.



1- Il programma applicativo effettua una chiamata di sistema poi pone sullo *stack* i parametri secondo un antica convenzione C/UNIX.
2- Poi invoca la procedura di librerie corrispondente alla chiamata.
3- Questa pone l'ID della chiamata in luogo noto al S/O.
4- Poi esegue la *trap* per passare all'esecuzione in modo operativo privilegiato.
5- Il S/O individua la chiamata da eseguire e la esegue.
6- Poi ritorna al chiamante oppure a un nuovo processo.
7- Ritorna come farebbe da return di procedura.
8- Cancella dati nello stack facendo avanzare il puntatore.

Alcune Chiamate di Sistema (POSIX)

Process management

Call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

File management

Call	Description
fd = open(file, how, ...)	Open a file for reading, writing or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

Directory and file system management

Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

Miscellaneous

Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970

• Una shell base tramite fork (UNIX):

```
• while (TRUE) {                                   /* repeat forever
  */
  type_prompt( );                                  /* display prompt */
  read_command (command, parameters)               /* input from terminal */

if (fork() != 0) {                                 /* fork off child process */
  /* Parent code */
  waitpid( -1, &status, 0);                        /* wait for child to exit */
} else {
  /* Child code */
  execve (command, parameters, 0);                 /* execute command */
 }
}
```

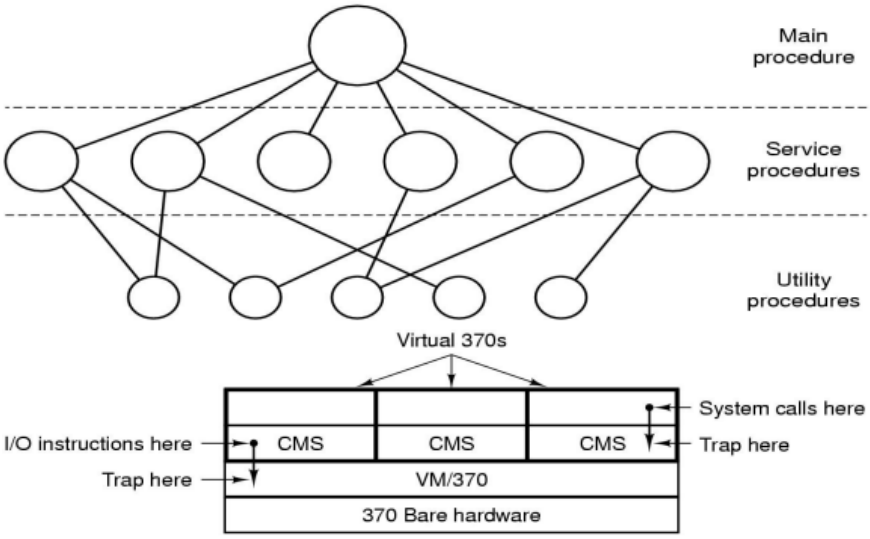
DEVERSE ARCHITETTURE DELLA LOGICA DEL SO:

Struttura monolitica:

non c'è una struttura il sistema operativo è una collezione piatta di procedure, ogni procedura può chiamare un'altra senza *information hiding.* L'unica struttura riconoscibile è data dalla convenzione di attivazione delle chiamate di sistema, i parametri vengono messi in un posto preciso(stack) e poi esegue *trap.*

Organizzazione:

1- Programma principale che invoca le procedure di servizio richieste.
2- Le procedure di servizio eseguono le system calls.
3- Procedure di utilità che sono di ausilio per le procedure di servizio.



Sistema a macchina virtuale:

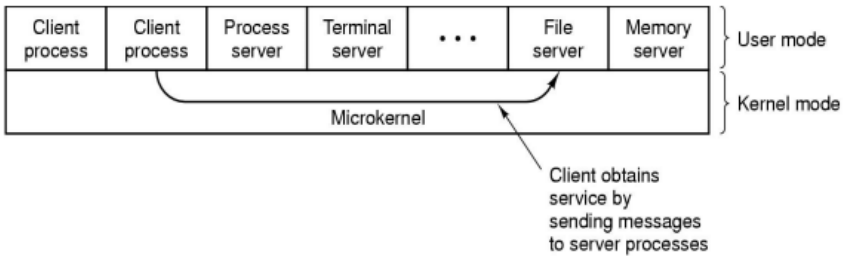
il sistema ha due fondamentali funzioni, multiprogrammazione e virtualizzazione dell'elaboratore fisico, questo schema ha avuto molto seguito nella realizzazione delle architetture dei SO.

Si possono offrire coppie identiche di MV(macchine virtuali) a sistemi operativi diversi per illudere più utenti di avere una macchina tutta per loro.

Il *CMS,* conversational monitor system, è il S/O interattivo a divisione di tempo monoutente.

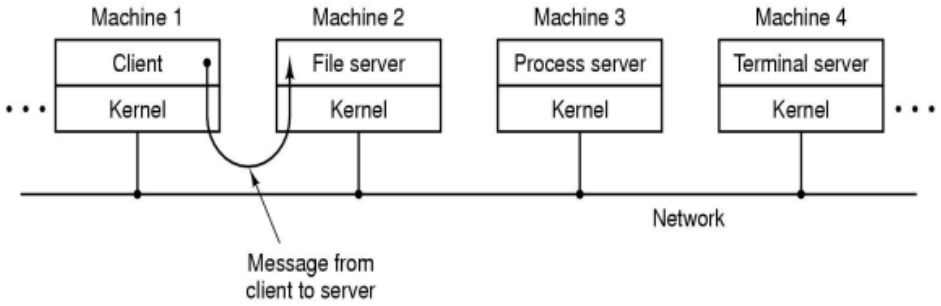
Architettura di tipo client-server:

Serie di processi che si trovano nella parte non protetta del sistema operativo collegati da un *kernel* che fa solo da collegamento tra i vari processi. Il continuo switch tra modalità causa rallentamento, ma come vantaggio ha semplicità e non interruzione del sistema quando un processo si interrompe. I processi di sistema sono visti come serventi, i processi utenti sono visti come clienti



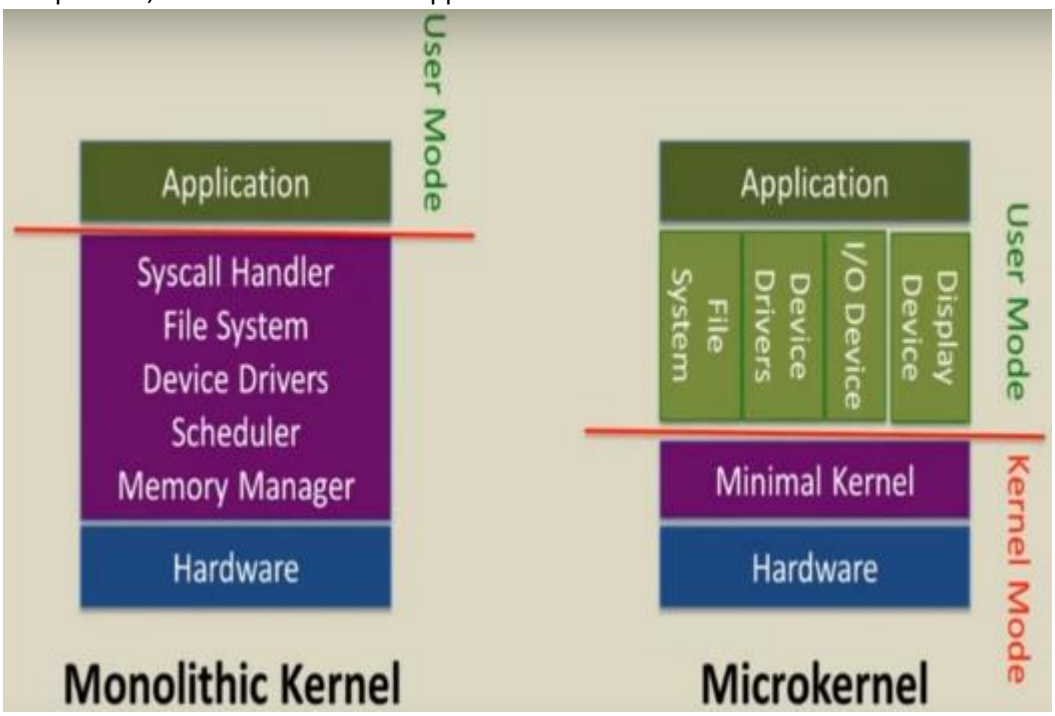
Struttura distribuita:

Simile al client-server ma ogni macchina ha un proprio kernel e collegate tra loro con un network.



Monolitico VS Microkernel:

- I kernel monolitici sono più semplici da realizzare e mantenere.
- i microkernel consentono gestione più flessibile ma hanno problemi di sincronizzazione tra le varie componenti, che ne rallentano sviluppo e mantenimento.



Unità di misura:

Exp.	Explicit	Prefix	Exp.	Explicit	Prefix
10^{-3}	0.001	milli	10^{3}	1,000	Kilo
10^{-6}	0.000001	micro	10^{6}	1,000,000	Mega
10^{-9}	0.000000001	nano	10^{9}	1,000,000,000	Giga
10^{-12}	0.000000000001	pico	10^{12}	1,000,000,000,000	Tera
10^{-15}	0.000000000000001	femto	10^{15}	1,000,000,000,000,000	Peta
10^{-18}	0.000000000000000001	atto	10^{18}	1,000,000,000,000,000,000	Exa
10^{-21}	0.000000000000000000001	zepto	10^{21}	1,000,000,000,000,000,000,000	Zetta
10^{-24}	0.000000000000000000000001	yocto	10^{24}	1,000,000,000,000,000,000,000,000	Yotta