# Monitor

- Con i **semafori** e **mutex** la comunicazione tra processi sembra apparentemente facile, **invece è difficile scrivere programmi corretti**.
- Un **monitor** è un insieme di procedure, variabili e dati strutturati tutti raggruppati in un particolare tipo di modulo o pacchetto.
- I processi possono richiedere le procedure in un **monitor** ogni volta che vogliono, ma non possono accedere direttamente le strutture dati interne del **monitor**.
- I **monitor** hanno una proprietà importante che li rende utili per realizzare la mutua esclusione: in un dato istante, **un solo processo per volta può essere attivo in un monitor**.

# Monitor

- Sono un costrutto del linguaggio di programmazione così il compilatore sa che li deve gestire in modo diverso rispetto alle altre chiamate di procedure.
- Quando un processo chiama una procedura in un **monitor**, esso controlla se c'è un altro processo già nel monitor:
  - il processo chiamante è sospeso finché l'altro non esce dal **monitor**;
  - altrimenti può entrare.
- Anche se i **monitor** forniscono un modo semplice per ottenere la mutua esclusione, occorre un sistema per bloccare i processi quando non possono andare avanti.

# Monitor

- La soluzione è l'uso delle **variabili condizione** abbinate alle due operazioni: **wait**() e **signal**().
- Quando una procedura dentro il monitor scopre che non può continuare (es. buffer pieno o vuoto) esegue una wait() su una variabile condizione e blocca il processo.
- Un altro processo partner può risvegliare il processo inviando una signal() sulla variabile condizione.
- sleep() e wakeup() sembrano simili alle operazioni wait() e signal(), ma le prime avevano il difetto che se un processo stava per addormentarsi non si accorgeva della sveglia dell'altro se non aveva già preso sonno.

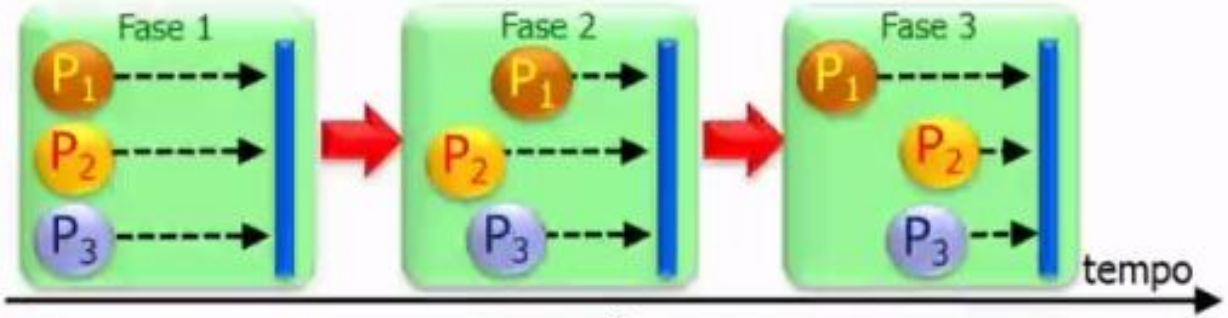
# Scambio di messaggi

- Questo metodo usa due chiamate di sistema: **send**() e **receive**() allo stesso modo dei semafori e differentemente dai monitor che sono un costrutto del linguaggio di programmazione.
- **send**(destinatario, &messaggio):
  - Invia un messaggio ad una destinatario.
- **receive**(fonte, &messaggio):
  - Riceve un messaggio da una fonte.

- Se il messaggio non è disponibile, il ricevente:
  - Rimane bloccato finché non ne riceve uno.
  - oppure può restituire immediatamente un codice di errore.

# Barriere

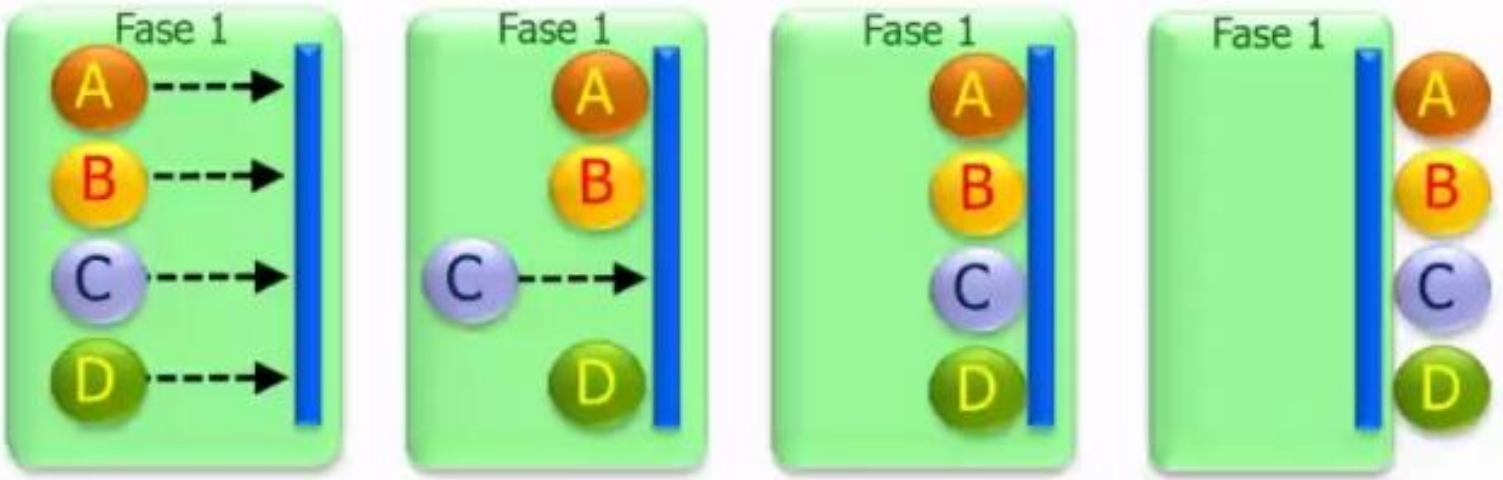
- Il meccanismo di sincronizzazione delle barriere è destinato a gruppi di processi, piuttosto che a comunicazioni tra due processi.
- Alcune applicazioni sono suddivise in fasi ed hanno la regola che nessun processo può procedere alla fase successiva fino a quando tutti i processi di gruppo sono pronti a passare alla fase successiva.




- Questo comportamento può essere ottenuto posizionando una **barriera** al termine di ogni fase.
- Quando un processo raggiunge la barriera, viene bloccato fino a che tutti gli altri processi hanno raggiunto la barriera.

# Esempio di barriere

- Quattro sono nella fase di calcolo, dopo un pò, il processo A termina e esegue la primitiva **barrier()** e viene sospeso.
- Successivamente anche il processo B e D finiscono l'elaborazione della prima fase ed eseguono la stessa primitiva. Anch'essi sono sospesi.
- Solo quando l'ultimo processo (C) arriva alla barriera, tutti i processi possono essere rilasciati.



# Scheduling

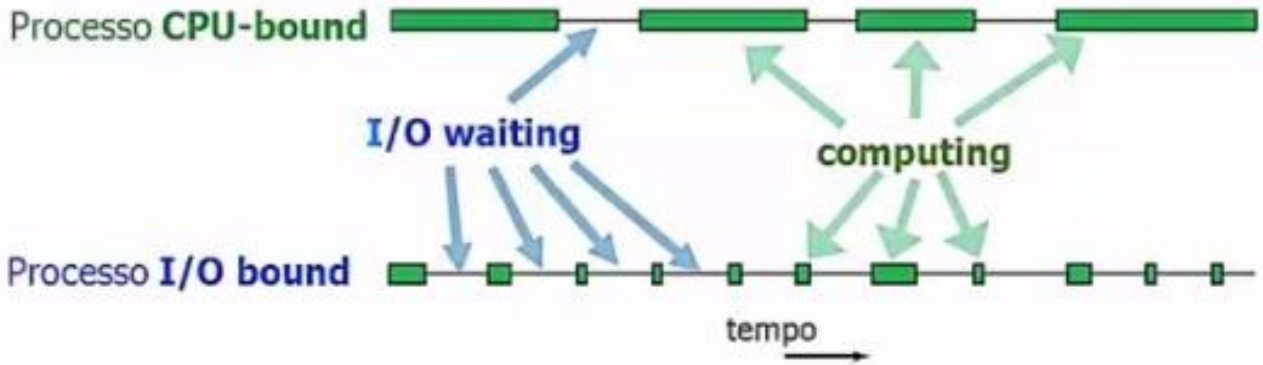
- Quando un computer è multiprogrammato, ha più processi o thread che competono per ottenere la CPU nel medesimo istante.
- Questa situazione si verifica quando due o più di essi sono contemporaneamente in stato di pronto.
- Con una CPU disponibile un solo processo/thread può essere selezionato per entrare in esecuzione.
- La parte del sistema operativo che effettua questa scelta è chiamato **scheduler** e l'algoritmo utilizzato è detto **algoritmo di scheduling**.
- Molti dei problemi che si applicano allo scheduling dei processi si applicano anche ai thread. Quando il kernel gestisce i thread, lo scheduling è fatto per thread indipendentemente dal processo di appartenenza.

# Introduzione allo scheduling

- Ai tempi dei sistemi batch l'algoritmo di scheduling era semplice: bastava eseguire il prossimo job sul nastro.
- Con i sistemi multiprogrammati, l'algoritmo di scheduling è diventato più complesso: più utenti sono in attesa del servizio.
- Con l'avvento del PC, la situazione si è evoluta in due modi:
  1) Il più delle volte lo scheduler ha un compito semplice perché c'è c'è solo un candidato tra i processi pronti: un utente che sta utilizzando un word processor non utilizza concorrentemente un'altra applicazione.
  2) Oggi i computer sono diventati così veloci che raramente la CPU è una risorsa scarsa. La maggior parte dei programmi per PC sono limitati dalla velocità di inserimento dati dell'utente.
- La situazione è completamente diversa se si considerano i server collegati in rete.

# Processi CPU-bound e I/O-bound

- Tutti i processi di alternano fasi di attesa di I/O con fasi di calcolo (uso intensivo della CPU):




- Alcuni processi spendono la maggior parte del loro tempo in elaborazione mentre altri in attesa dell'I/O.
- I processi limitati nel calcolo utilizzano la CPU per lunghi periodi ed hanno attese di I/O poco frequenti, al contrario gli altri usano la CPU per brevi periodi ma hanno attese di I/O frequenti.
- Con l'incremento delle performance delle CPU i processi tendono ad essere più I/O bound che non CPU-bound.

# Quando eseguire lo scheduling?

- Un aspetto cruciale dello scheduling è stabilire quando è il momento opportuno per eseguirlo:
  1) Quando viene creato un nuovo processo, occorre scegliere se passare l'esecuzione al processo figlio oppure continuare l'esecuzione di quello genitore.
  2) Se un processo termina, qualche altro processo deve essere scelto nell'insieme dei processi pronti.
  3) Se un processo di blocca su una operazione di I/O o su un semaforo bisogna selezionare un altro processo dall'elenco dei processi pronti.
  4) Quando si verifica un interrupt di I/O, può essere presa una decisione di programmazione (che dipende dal tipo di interrupt).

# Tipi di scheduling

- Gli algoritmi di scheduling possono essere suddivisi in due categorie rispetto alla capacità dello scheduling di interrompere l'esecuzione dei processi:

1) **non preemptive**: lo scheduler una volta mandato in esecuzione un processo lo lascia andare finché non si blocca (attesa di I/O o di un altro processo) o rilascia spontaneamente la CPU.
2) **preemptive**: lo scheduler sceglie un processo da eseguire per un tempo massimo stabilito. Se il processo è ancora in esecuzione al termine dell'intervallo di tempo, lo scheduler lo sospende e sceglie un altro processo da eseguire (se disponibile).

# Categorie di algoritmi di scheduling

- In diverse aree applicative sono necessari differenti sistemi operativi e, conseguentemente, specifici algoritmi di scheduling.

- Saranno trattati tre differenti ambienti:
  1) **Sistemi batch.**
  2) **Sistemi interattivi.**
  3) **Sistemi in tempo reale.**

# Sistemi batch

- I sistemi batch sono ancora utilizzati nel mondo bancario e assicurativo per il calcolo delle buste paga, l'inventario, il calcolo degli interessi, la gestione dei reclami e altre attività periodiche.
- Nei sistemi batch, non ci sono utenti impaziente in attesa di una risposta rapida al loro breve quesito.
- In questo contesto, gli algoritmi non preeventive sono spesso accettabili.

# Sistemi interattivi

- In un ambiente con utenti interattivi, la preemption è essenzialmente per evitare che un processo utente monopolizzi la CPU bloccando gli altri utenti.
- La preemption è essenziale anche per i server che gestiscono normalmente molti utenti remoti.

# Sistemi real-time

- Nei sistemi real-time, la preemption potrebbe non essere necessaria perché i processi sanno che devono dare dei risultati in tempi brevi.

# Obiettivi degli algoritmi di scheduling

- Per progettare un algoritmo di scheduling ci sono obiettivi comuni e altri che dipendono dal tipo di ambiente (batch, interattivo o real-time).
- In ogni caso, l'**equità** (dando ad ogni elaborare una congrua parte della CPU) è molto importante.
- Forzare le **politiche** del sistema è scorretto. In un centro di calcolo reattore nucleare, se la politica locale di sicurezza è che il ciclo di controllo deve prendere 10 msec, lo scheduler deve assicurarsi questa politica venga **sempre** rispettata.
- Tutte le parti del sistema devono essere impegnate in modo **bilanciato**: se la CPU e i dispositivi di I/O sono tenuti sempre impegnati vengono eseguiti più lavori al secondo rispetto ad avere componenti inattivi.

# Obiettivi degli algoritmi di scheduling

- I data center che eseguono molti lavori batch sono interessati a due metriche principali:
  1) **throughput**: numero di job completati nell'unità di tempo.
  2) **tempo di turnaround**: tempo medio di esecuzione dei processi batch.
- Anche se è metrica non appropriata per i sistemi batch, viene spesso utilizzata la percentuale di **utilizzo della CPU**.
- Nei sistemi interattivi è fondamentale il **tempo di risposta**: il tempo che intercorre dall'invio del comando a quando si ottiene il risultato.
- L'**adeguatezza della risposta** è, invece, la percezione degli utenti rispetto al tempo impiegato per svolgere quel compito (lo scheduler potrebbe assegnare male le priorità di esecuzione e causare un degrado delle risposte).
- Per i sistemi in tempo reale è essenziale: soddisfare le scadenze ed evitare errori durante l'esecuzione del processo.

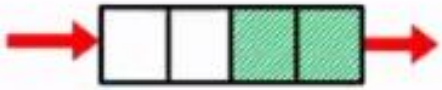
# Scheduling nei sistemi batch

- Gli algoritmi di scheduling utilizzati nei sistemi batch sono:
  - **First-come first-served.**
  - **Shortest job first.**
  - **Shortest remaining time next.**

- Tra questi alcuni sono validi anche nei sistemi interattivi.

# First-come first-served

- È il più semplice algoritmi di scheduling nonpreemptive: il primo che arriva è il primo ad essere servito.



- La CPU è assegnata ai processi in ordine di arrivo.
- C'è un'unica coda dei processi pronti.
- Questo algoritmo è facile da capire e da programmare.

- **Problema**
  - un sistema con processi eterogenei per tempo di esecuzione potrebbe rallentare i processi veloci in assenza di preemption.

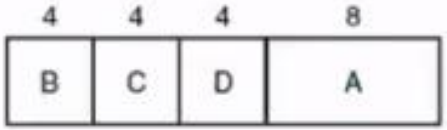
# Shortest job first

- Si tratta di un algoritmo batch senza preemption in cui i tempi di esecuzione dei processi sono noti in anticipo (la predizione del tempo di esecuzione per job ripetitivi non è difficile).
- I job entrano nella coda in ordine, ma lo scheduler sceglie quello che termina prima.

# Shortest job first

- Si supponga che arrivino 4 job A, B, C, e D con la rispettiva stima dei tempi di esecuzione (in minuti):


- Se si facessero entrare in esecuzione nell'ordine di arrivo A avrebbe un tempo di turnaround di 8', B di 12', C di 16' e D di 20' minuti (tempo medio=14').
- Usando l'algoritmo shortest job first, i tempi di turnaround diventano 4', 8', 12', and 20' minutes (tempo medio=11'):




- Attenzione: l'algoritmo **shortest job first** è ottimale solo se i job sono tutti disponibili al momento della scelta.

# Shortest Remaining time next

- È una versione preemptive del precedente algoritmo.
- Il tempo di esecuzione è noto a priori.
- Lo scheduler sceglie il processo a cui manca meno tempo al termine dell'esecuzione.
- Se arriva un nuovo job che ha bisogno di meno tempo per terminare rispetto al job corrente, quest'ultimo è sospeso e viene eseguito il nuovo arrivato.

# Scheduling nei sistemi interattivi

- Nei sistemi interattivi possono essere utilizzati i seguenti algoritmi:
  - **Scheduling Round-Robin.**
  - **Scheduling con priorità.**
  - **Shortest process next.**
  - **Scheduling garantito.**
  - **Scheduling a lotteria.**
  - **Scheduling Fair-Share.**

# Scheduling round-robin

- È uno degli algotirmi più vecchi, più semplice, più equilibrato e diffusamente utilizzato.

- Ad ogni processo è assegnato un «quantum» di tempo di CPU per l'esecuzione, allo scadere del quale viene interrotto e si passa, con modalità circolare e parietaria, al successivo processo.

- La scelta del «quantum» è un fattore critico: un valore troppo breve provoca troppe interruzioni dei process causando **overhead**, mentre un valore troppo alto provoca una coda di attesa eccessiva.

# Scheduling con priorità

- Nello scheduling round-robin c'è l'assunzione implicita che tutti i processi sono ugualmente importanti, in molti contesti reali questa è una ipotesi troppo restrittiva.
- Un approccio differente assegna una priorità a ciascun processo e lo scheduling tiene conto della priorità nel momento in cui deve scegliere il processo da mandare in esecuzione.
- Per evitare che i processi ad alta priorità girino per un tempo indeterminato, lo scheduler può:
  1) diminuire la priorità del processo in esecuzione ad ogni ciclo di clock;
  2) utilizzare un quantum di tempo massimo di CPU.
- Le priorità possono essere assegnate in modo statico o dinamico.
- È conveniente dividere i processi per classi di priorità utilizzando lo scheduling con priorità per le classi e quello round-robin all'interno di ciascuna classe.

# Shortest process next

- Poiché l'algoritmo **shortest job first ha sempre il minor** tempo medio di risposta nei sistemi batch, sarebbe interessante poterlo utilizzare anche per i processi interattivi.
- La difficoltà nei sistemi interattivi è nel calcolare il tempo di esecuzione che non è sempre lo stesso.
- Un possibile approccio è di misurare i tempi di esecuzione e di utilizzarli per le stime successive attraverso una media pesata tenendo conto che le misure più recenti sono più attendibili rispetto a quelle passate (**aging**):

| iterazione | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| Tempo di esecuzione | $T_0$ | $T_1$ | $T_2$ | $T_3$ |
| Calcolo | $T_0$ | $T_0/2+T_1/2$ | $T_0/4+T_1/4+T_2/2$ | $T_0/8+T_1/8+T_2/4+T_3/2$ |
| stima di | $T_1$ | $T_2$ | $T_3$ | $T_4$ |

# Scheduling garantito

- Un approccio completamente diverso per lo scheduling è quello di fare promesse sulle performance reali dei processi utente e poi mantenerle.
- Se ci sono **n** utenti collegati dovrebbero ricevere 1/**n** del tempo della CPU.
- Per mantenere questa promessa, il sistema memorizza quanta CPU ogni processo ha avuto fin dalla sua creazione e calcola il rapporto tra avuto/promessa:
  - Se un qualsiasi processo ha un valore più basso di tale rapporto è il successivo candidato ad essere messo in esecuzione fino a quando il suo rapporto è il più basso tra tuttu quelli pronti.

# Scheduling a lotteria

- Fare promesse agli utenti è una bella idea è ma difficile da realizzare.
- Si può utilizzare un algoritmo più semplice: si assegna un biglietto della lotteria alle varie risorse del sistema, come il tempo di CPU.
- Ogni volta che lo scheduling deve fare una scelta pesca caso un biglietto e il processo che ha quel biglietot si aggiudica la risorsa.
- Differentemente dallo scheduling con le priorità in cui è difficile dire cosa significa avere una certa priorità, in questo algoritmo se un processo possiede 20 biglietti su 100 ha il 20% delle probabilità di ottenere la risorsa (che al crescere del tempo approssima la frequenza).

# Vantaggi dello scheduling a lotteria

- L'algoritmo di scheduling a lotteria ha due proprietà interessanti:
  - È reattivo: anche i processi neonati possono vincere la lotteria, fin dalle prime scelte dello scheduler.
  - I processi che cooperano, se lo desiderano, possono scambiarsi biglietti che detengono per alterare le priorità di esecuzione.
- Può essere utilizzato per problemi che sono di difficile risoluzione con altri metodi come ad esempio lo **streaming video** dove sono necessarie differenti velocità di trasferimento (espresse in frame al secondo) a seconda del processo. Assegnando tanti biglietti quanti sono i frame automaticamente la CPU approssima le proporzioni desiderate.

# Scheduling fair-share (a quota equa)

- Fino ad ora ogni processo è schedulato per proprio conto indipendentemente dal suo proprietario.
- Se un utente X avvia 4 processi (A, B, C e D) ed un altro Y uno soltanto (E), il primo utente si prenderà l'80% del tempo di CPU.

# Scheduling nei sistemi real-time

- In un sistema in tempo reale, il tempo gioca un ruolo essenziale.
- I sistemi real-time sono generalmente suddivisi in categorie:
  - **Hard real-time**: le scadenze devono essere sempre rispettate.
  - **Soft real-time**: il mancato rispetto di una scadenza non è auspicabile, ma comunque tollerabile.
- Gli eventi in un sistema real-time possono essere classificati come:
  - **Periodici**, quando si verificano a intervalli di tempo regolari.
  - **Non periodici**, quando si verificano in modo imprevedibile.

# Sistemi real-time con eventi periodici

- Un sistema **soft real-time con eventi periodici** è sostenibile se riesce a far fronte agli eventi stessi, ovvero se riesce a trattare un evento prima che ne arrivi un altro.

- Se ci sono «m» eventi periodici e ogni evento avviene con frequenza pari a $1/P_i$ , supponendo che ciascun evento richieda $C_i$ secondi di tempo CPU per gestirlo, allora il sistema è in grado di reggere il carico se e solo se:

$$\sum_{i=0}^{m} \frac{C_i}{P_i} < 1$$

# Problema

- Si consideri un sistema soft real-time con tre eventi periodici:
  - P0 = 100 ms, P1 = 200 ms, P2 = 500 ms

  e tempi di elaborazione per ciascun evento rispettivamente di:
  - C0 = 50 ms, C1 = 30 ms, C2 = 100 ms

- Il sistema è sostenibile?
  - Sì perché il carico è dell'**85%**:

$$0{,}5 + 0{,}15 + 0{,}2 = 0{,}85 < 1$$

- Se viene aggiunto un quarto evento con un periodo di **1 sec** quanto è il tempo limite di processamento affinché il sistema rimanga sostenibile?
  - <**150** ms del tempo di CPU:

$$1 - (0{,}5 + 0{,}15 + 0{,}2) = 0{,}15$$

# Scheduling dinamico e statico

- Gli algoritmi di scheduling per i sistemi real-time possono essere statici o dinamici:

1. Gli algoritmi statici prendono le decisioni di scheduling prima che il sistema inizia a funzionare:
   - è applicabile solo quando ci sono informazioni disponibili in anticipo riguardo il lavoro da svolgere e le scadenze da rispettare.

2. Gli algoritmi dinamici prendono le decisioni di scheduling in fase di esecuzione:
   - non hanno bisogno di conoscere in anticipo alcuna informazione sul compito da svolgere e sui tempi.

# La politica contro il meccanismo

- Fino ad ora, abbiamo assunto che tutti i processi appartengono a diversi utenti e quindi competano per ottenere la CPU.
- A volte un processo può avere molti processi figli eseguiti sotto il suo controllo (un dbms può avere molti processi figli).
- Il processo genitore conosce bene le priorità da assegnare ai processi figli, per lo scheduler i processi sono tutti allo stesso livello.
- La soluzione a questo problema consiste nel separare il meccanismo di scheduling dalla politica di scheduling.
- L'algoritmo di scheduling ha parametri che possono essere compilati dai processi utente.

# Scheduling a thread

- Quando i processi hanno più thread, possiamo definire due livelli di parallelismo: sui **processi** e sui **thread**.

- Lo scheduling in questi ambienti si differenzia a seconda che siano supportati **thread utente** o a **livello kernel** (o entrambi).

# Scheduling dei thread a livello utente

- Il kernel non conosce l'esistenza dei thread perché conosce solo i processi che li contengono.
- Supponiamo che:
  - Il processo di A ha tre thread: A1, A2 e A3.
  - Il processo di B dispone di tre thread: B1, B2 e B3.
- Lo scheduler sceglie il processo A e dà al processo il controllo per il suo quantum di tempo.
- Ora lo scheduler di thread interno ad A decide quale thread eseguire (ad esempio A1). Poiché per i thread non ci sono interrupt esso può continuare a funzionare finché vuole (al max il quantum di A).
- Se consuma tutto il quantum di A, il kernel sceglie un altro processo per l'esecuzione.
- La sequenza di attivazione A1 → B1 → A2 → B2 → A3 → B3 non è possibile!

# Scheduling dei thread a livello kernel

- Il kernel conosce i thread e ne prende uno per l'esecuzione.
- Al thread è assegnato un quantum di tempo e viene forzatamente sospeso se supera quel valore di tempo di esecuzione.
- La sequenza di attivazione A1 → B1 → A2 → B2 → A3 → B3 è ora possibile!

# Vantaggi e svantaggi

- **Thread a livello utente**
  - lo scambio di esecuzione tra thread è veloce e sono sufficienti poche istruzioni macchina.
  - Se un thread va in blocco su una operazione di I/O si deve sospendere l'intero processo.
  - Si può utilizzare uno scheduler di thread specifico dell'applicazione (es. web server) e che abiliti una strategia di attivazione migliore del kernel, poiché conosce ciò che fanno i vari thread.

# Vantaggi e svantaggi

- **Thread a livello del kernel**
  - Lo scambio di contesto è oneroso di molti ordini di grandezza rispetto all'altro caso, poiché occorre invalidare la mappa di memoria e la cache.
  - Il kernel può decidere quale thread mandare in esecuzione tenendo conto anche dell'overhead causato da un eventuale cambio di contesto.
  - Se un thread va in blocco non sospende l'intero processo.

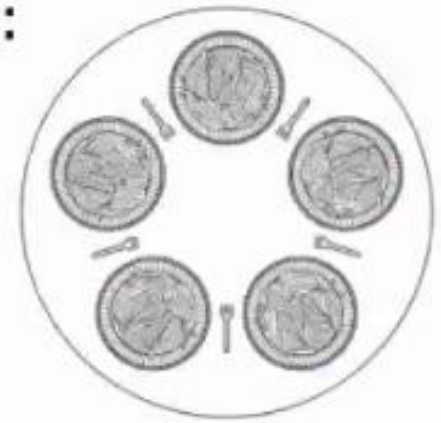
# Problemi classici di IPC

- La letteratura dei sistemi operativi è piena di problemi interessanti che sono stati discussi utilizzando una varietà di metodi di sincronizzazione.

- Nel seguito assegneremo come esercizio due dei problemi più noti:
  - Il problema dei 5 filosofi.
  - Il problema dei lettori e scrittori.

# Il problema dei 5 filosofi

- Cinque filosofi sono seduti a un tavolo circolare con un piatto di spaghetti di fronte a loro.
- Per mangiare gli spaghetti sono necessarie due forchette e sul tavolo ci sono solo 5 forchette, quindi solo due filosofi possono mangiare contemporaneamente.
- La vita di un filosofo si alterna a momenti in cui pensa e in cui mangia.
- Siamo in grado di scrivere un programma:
  per ciascun filosofo che gli permetta di
  pensare per il tempo a lui necessario
  e mangiare senza che il sistema
  si blocchi?



# Una soluzione errata

- La procedura **take_fork(i)** aspetta finché la i-esima forchetta non è disponibile, quando è libera la solleva.

```
#define N 5                          /* numero dei filosofi                          */
void philosopher(int i) {            /* i identifica il filosofo da 0 a 4            */
    while (1) {                      /* ciclo infinito                               */
        think();                     /* il filosofo pensa per il tempo che vuole     */
        take_fork(i);                /* prende la forchetta alla sua sinistra        */
        take_fork( (i+1) % N);       /* prende la forchetta alla sua destra          */
        eat();                       /* mangia gli spaghetti                         */
        put_fork(i);                 /* pone sul tavolo la forchetta di sinistra     */
        put_fork( (i+1) % N);        /* pone sul tavolo la forchetta di destra       */
}
```

- Sfortunatamente questa è una soluzione è sbagliata: se tutti i filosofi prendono la forchetta di sinistra contemporaneamente nessuno sarà in grado di trovare libera l'altra e ci sarà uno stallo (o **deadlock**).

# Starvation

- Potremmo modificare il programma in modo che un filosofo, dopo aver preso la forchetta sinistra, controlli quella di destra:
  - Se disponibile, la prenda.
  - altrimenti, riponga la sinistra sul tavolo, attenda per un certo tempo e ripeta ciclicamente il processo.
- Tuttavia questa soluzione non funziona per un altro motivo:
  - Se tutti i filosofi iniziano l'algoritmo contemporaneamente prendono la forchetta di sinistra e guardano che manca quella alla loro destra, allora posano quella alla loro sinistra e ripetono questa sequenza di passaggi in modo infinito.
- Una situazione come questa, in cui tutti i programmi continuano a essere eseguiti indefinitivamente, senza riuscire a fare alcun progresso si chiama **Starvation**.

# Alcune soluzione al problema

- Una possibile soluzione sfrutta un'attesa casuale tra il momento in cui viene rilasciata la forchetta alla sinistra perché quella alla destra non è disponibile e il nuovo tentativo.
- In molti campi di applicazione non è un problema riprovare finché non si estrae un numero casuale vincente, tuttavia in alcuni (come ad esempio il sistema di controllo di una centrale nucleare) non è accettabile.
- Un'altra soluzione sfrutta il miglioramento di quella con deadlock: si proteggono le istruzioni che seguono **think**() con un semaforo binario.
- Prima di iniziare ad acquisire le forchette, un filosofo esegue una **down**() su un **mutex** e dopo aver riposto la forchetta un **up**() sullo stesso **mutex**.
- Questa soluzione è idonea ma non efficiente: mangia un solo filosofo per volta!