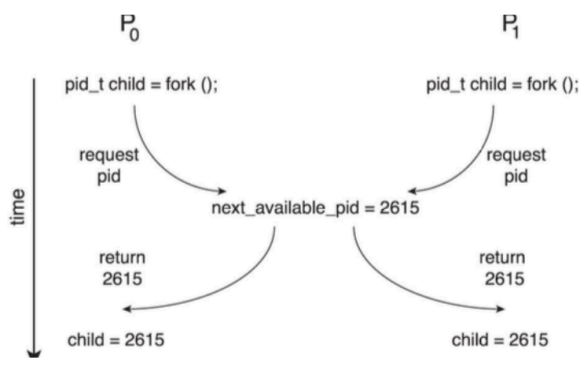
23 Maggio 2023

ARCHITETTURE DI CALCOLO LEZIONE 19

Race condition, sezione critica e algoritmi - Esercizi

RACE CONDITION

Letteralmente "condizione che si verifica in corsa": situazione in cui più processi accedono e manipolano dati condivisi in modo concorrente e il valore finale dei dati condivisi dipende dall'ordine con cui sono avvenuti gli accessi. Può portare ad inconsistenza.
Per eliminare le race condition, i processi concorrenti devono essere sincronizzati in modo che la manipolazione delle variabili condivise sia consentita ad un solo processo alla volta.
Un esempio di race condition lo si trova nella funzione di fork () durante la creazione sincrona di processi figli da parte di due processi padre. In particolare, alla creazione di un processo figlio segue l'assegnazione di un identificatore (pid) che deve essere univoco: senza sincronizzazione, lo stesso pid potrebbe essere assegnato a due processi diversi.



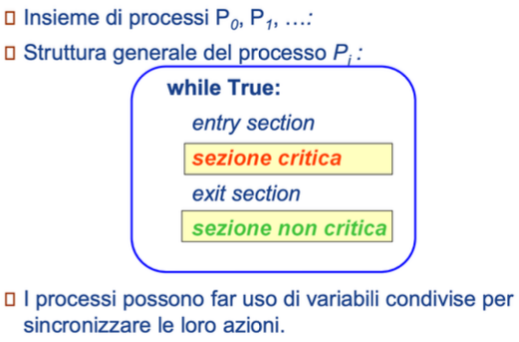
La race condition può verificarsi nella cosiddetta sezione critica, un segmento di codice nel quale i dati condivisi vengono acceduti e modificati da più processi. La soluzione è l'esecuzione delle sezioni critiche da parte dei processi in modo mutuamente esclusivo nel tempo: solo un processo per volta può accedere alla sezione critica.
Una soluzione al problema dell'accesso sincrono alla sezione critica deve soddisfare i seguenti requisiti:

- **Mutua esclusione (ME)**: Se il processo Pi è in esecuzione nella sua sezione critica, nessun altro processo può essere in esecuzione nella propria sezione critica.
- **Progresso (P)**: Se nessun processo è in esecuzione nella propria sezione critica, allora soltanto i processi che cercano di entrare nella sezione critica possono partecipare alla decisione di chi entrerà nella propria sezione critica, e questa decisione deve avvenire in un tempo finito.
- **Attesa limitata (AL)**: Deve esistere un limite nel numero di volte che altri processi sono autorizzati ad entrare nelle rispettive sezioni critiche dopo che un processo Pi ha fatto richiesta di entrare nella propria sezione critica e prima che quella richiesta sia soddisfatta.
  - Si assume che ogni processo esegua a velocità non nulla.
  - Non è possibile fare assunzioni sulla velocità relativa degli N processi.

Vedremo diverse applicazioni di soluzioni del problema della sezione critica, ma tutte prevedono:

- **While True**
- **Entry section:** parte di codice che esegue prima di entrare nella sezione critica

- **Sezione critica:** porzione di codice in cui si modificano le variabili condivise.
- **Exit section:** parte di codice in cui si dichiara l'uscita del programma dalla sezione critica.
- **Sezione non critica:** parte in cui il programma lavora su variabili non condivise.



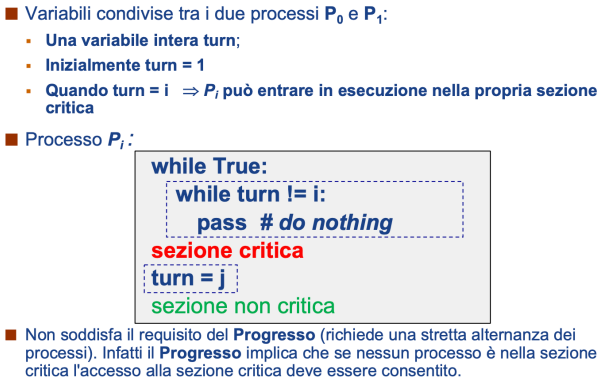
Soluzioni software

- Si considerano solo due processi P0, P1
  - Algoritmi di Dijkstra (Algoritmo 1 e Algoritmo 2)
  - Algoritmo di Peterson
- Considera n-processi P0, P1, ... Pn-1
  - Algoritmo di Lamport Soluzioni hardware

Soluzioni con API software

- Lock mutex (o Mutex)
- Semafori
- Monitor

ALGORITMO 1 DI DIJKSTRA



Questo primo algoritmo prevede nella entry section un ciclo while che permette, tramite una variabile condivisa turn, di indicare il turno del processo da eseguire: fintanto che il turn è 1 ha accesso alla sezione critica il processo 1 e si mantiene il processo 0 in waiting; viceversa per turn=0.

```
P0

while True:

   while turn != 0:
      pass
      <sezione critica>
      turn = 1
      ...
```

```
P1

while True:

   while turn != 1:
      pass
      <sezione critica>
      turn = 0
      ...
```

Può capitare che il processo 0 voglia accedere ma, non essendo in esecuzione il processo 1, il turno del processo 0 non può avvenire tempestivamente: se la simmetria tra processi non è rispettata allora non viene soddisfatto il requisito del Progresso. Vale solo per 2 processi, non funziona con più processi.

ALGORITMO 2 DI DIJKSTRA

- Variabili condivise:
  - **flag = [False] * 2 # Inizialmente flag[0] e flag[1] sono False.**
  - **Quando flag[i] == True ⇒ P_i è pronto ad entrare nella sua sezione critica**
- Processo P_i:

```
while True:
    flag[i] = True
    while flag[j]:
        pass // no-op
    sezione critica
    flag[i] = False
    sezione non critica
```

*I processi possono entrare in un loop infinito quando ai due flag viene assegnato il valore **True**.*

P0

```
while True:
  flag[0] = true
  while flag[1]:
    pass
  <sezione critica>
  flag[0] = False
```

P1

```
while True:
  flag[1] = true
  while flag[0]:
    pass
  <sezione critica>
  flag[1] = False
```

Cerca di risolvere il problema del primo algoritmo tramite inserimento di una variabile condivisa (una per ogni processo, solo 2 processi) detta flag contenente un booleano; quando flag[i] == True allora il processo i-esimo è pronto ad entrare nella sua sezione critica.
Il problema di tale algoritmo sta nel fatto che se i due processi vogliono entrare contemporaneamente si verifica un deadlock, situazione in cui ogni processo è in attesa di un evento da parte di un altro processo che, a sua volta, attende un evento dell'altro processo, creando un ciclo infinito (viene killato il programma).

ALGORITMO DI PETERSON

- Usa le variabili condivise degli algoritmi 1 e 2.
- Processo P_i:

```
while True:
    flag[i] = True
    turn = j
    while flag[j] and turn == j:
        pass
    sezione critica
    flag[i] = false
    sezione non critica
```

L'array *flag* si usa per indicare se un processo è pronto ad entrare nella propria sezione critica

La variabile *turn* indica il processo che è «di turno» per accedere alla sezione critica

Pi assegna innanzitutto a flag[i] il valore true; quindi attribuisce a turn il valore j, conferendo così all'altro processo la facoltà di entrare nella sezione critica.

- Soddisfa tutti e tre i requisiti (ME, P, AL); tuttavia vale soltanto per due processi.

P0

```
while True:
  flag[0] = True
  turn = 1
  while flag[1] and turn == 1:
    pass
  <sezione critica>
  flag[0] = False
```

P1

```
while True:
  flag[1] = True
  turn = 0
  while flag[0] and turn == 0:
    pass
  <sezione critica>
  flag[1] = False
```

- non accade mai che un processo si blocchi se l'altro non è nella sezione critica
- Un processo in attesa, prima o poi si sblocca.

Peterson risolve il problema del deadlock tramite l'aggiunta di una variabile turn (turno) che fa si che il processo venga eseguito solo nel caso in cui si ha che:
• La sua flag è True
È il proprio turno
In caso di contemporaneità, non si ha il deadlock in quanto, anche se entrambe le rispettive flag sono a True, il turn andrà casualmente a favore di uno solo dei due processi e quindi solo uno dei due processi entra nella sezione critica. Nello specifico:
•Qualora entrambi i processi tentino l'accesso contemporaneo, all'incirca nello stesso momento sarà assegnato a turno sia il valore i sia il valore j. Soltanto uno dei due permane: l'altro sarà immediatamente sovrascritto. Il valore definitivo di turn stabilisce quale dei due processi sia autorizzato a entrare per primo nella propria sezione critica.

P0: flag[0]=True P1: flag[1]=True P1: turn=0
P1: while flag[0] and Turn==0: pass P0: turn=1
P0: while flag[1] and Turn==1: pass

- Mutua esclusione: turn può valere 0 o 1, ma non entrambi.
- Progresso: l'ingresso di un processo Pi nella propria sezione critica può essere impedito solo se il processo è bloccato nella sua iterazione while, con le condizioni flag[j] == True e turn == j; questa è l'unica possibilità.
- Attesa limitata: Pi entrerà nella sezione critica (progresso) dopo che Pj ha effettuato non più di un ingresso (attesa limitata).

ALGORITMO DEL FORNAIO (DI LAMPORT)

Tale algoritmo risolve il problema della sezione critica per N processi:
prima di entrare nella sezione critica, il processo "riceve" un numero. Il possessore del numero più piccolo entra nella sezione critica.
Lo schema di numerazione non assicura che ogni processo riceva un numero diverso, ma garantisce che genererà sempre numeri in ordine crescente; ad esempio: 1,2,3,3,3,3,4,5...
In questo caso, passerà il processo con ID interno minore.
L'ALGORITMO VIENE SPIEGATO NELLA PROSSIMA LEZIONE

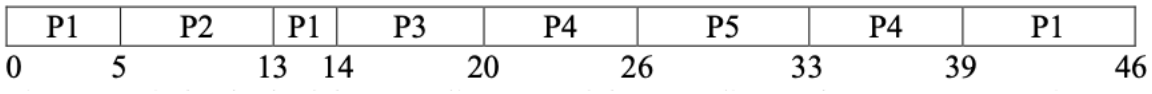
ESERCIZI SULLO SCHEDULING DELLA CPU

Primo esercizio
Si supponga di avere cinque processi che arrivano nel sistema al tempo di arrivo specificato nella seguente tabella, dove sono indicate anche le durate dei CPU burst e le priorità (dove 3 è la priorità massima):

Processo	*Tempo di Arrivo*	*CPU Burst*	*Priorità*
P1	*0*	*13*	*1*
P2	*5*	*8*	*3*
P3	*14*	*6*	*2*
P4	*18*	*12*	*2*
P5	*26*	*7*	*3*

Si mostri la sequenza di esecuzione e si calcolino il tempo di attesa e il tempo di completamento di ciascun processo, considerando i seguenti algoritmi di scheduling:
A. Priorità con prelazione
B. SJF senza prelazione

A. Nel metodo di scheduling per priorità, i processi da eseguire vengono messi in coda in base a chi ha la priorità maggiore (nell'esercizio 3=max priorità, 1=min priorità); la prelazione permette di interrompere un processo in esecuzione qual ora sopraggiunga un processo che ha maggiore priorità. Secondo questi criteri, si ha la seguente sequenza di esecuzione:

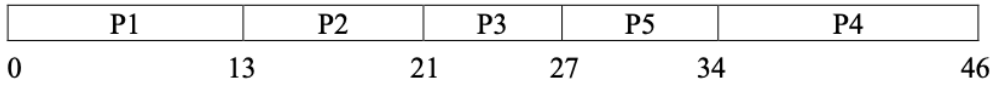


Si passa poi al calcolo del tempo di attesa e del tempo di completamento per ogni processo:

- Il tempo di attesa è dato dalla somma degli intervalli di tempo, dall'arrivo del processo al suo completamento, in cui il processo stesso non è in esecuzione, risultando in attesa. Ad esempio, il processo P1 ha atteso dal tempo 5 al tempo 13 (8) e poi dal tempo 14 al tempo 39 (25)àTempo d'attesa = 8+25=33
- Il tempo di completamento è dato dalla differenza del momento in cui il processo è terminato con il tempo di arrivo. Ad esempio, il processo P4, terminato a tempo 39 e con tempo di arrivo 18, ha tempo di completamento uguale a 39-18 = 21.

Processo	***Tempo di Attesa***	***Tempo di Completamento***
P1	*33*	*46*
P2	*0*	*8*
P3	*0*	*6*
P4	*9*	*21*
P5	*0*	*7*

B. Secondo la logica SJF vengono eseguiti per primi i processi più brevi; essendo senza prelazione, un processo già in esecuzione non può essere interrotto anche se dovesse arrivare un processo più breve. Secondo questi criteri, si ha la seguente sequenza di esecuzione:



Processo	***Tempo di Attesa***	***Tempo di Completamento***
P1	*0*	*13*
P2	*8*	*16*
P3	*7*	*13*
P4	*16*	*28*
P5	*1*	*8*

Secondo esercizio

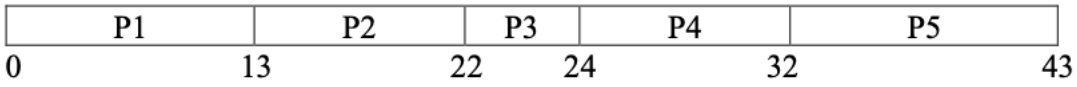
Si supponga di avere cinque processi che arrivano nel sistema al tempo di arrivo specificato nella seguente tabella, dove sono indicate anche le durate dei CPU burst:

Processo	***Tempo di Arrivo***	***CPU Burst***
P1	*0*	*13*
P2	*3*	*9*
P3	*8*	*2*
P4	*12*	*8*
P5	*16*	*11*

Si mostri la sequenza di esecuzione e si calcolino il tempo di risposta ed il tempo di completamento di ciascun processo, considerando i seguenti algoritmi di scheduling:

A. FCFS
B. SJF con prelazione

A. Secondo lo scheduling FCFS i processi vengono eseguiti secondo tempo di arrivo, senza la possibilità di interrompere un processo in esecuzione. Secondo tali criteri, si ha la seguente sequenza di esecuzione:

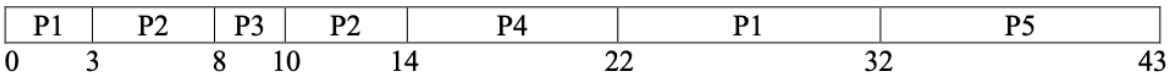


Il tempo di risposta corrisponde alla differenza tra il tempo di avvio dell'esecuzione del processo con il tempo di arrivo. Ad esempio, per il P2 il tempo di risposta è dato da 13-3 = 10.

Processo	***Tempo di Risposta***	***Tempo di Completamento***
P1	*0*	*13*
P2	*10*	*19*
P3	*14*	*16*
P4	*12*	*20*
P5	*16*	*27*

Nota: nello scheduling FCFS il tempo di risposta è uguale al tempo di attesa.

B. SJF con prelazione: prima i più brevi, un processo può essere interrotto. Sequenza di esecuzione:



Processo	***Tempo di Risposta***	***Tempo di Completamento***
P1	*0*	*32*
P2	*0*	*11*
P3	*0*	*2*
P4	*2*	*10*
P5	*16*	*27*

Terzo esercizio

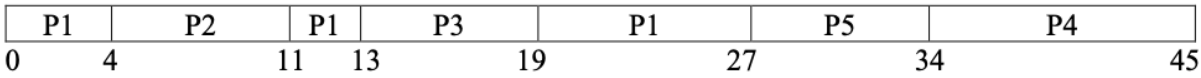
Si supponga di avere cinque processi che arrivano nel sistema al tempo di arrivo specificato nella seguente tabella, dove sono indicate anche le durate dei CPU burst e le priorità (dove 3 è la priorità massima):

Processo	*Tempo di Arrivo*	*CPU Burst*	*Priorità*
P1	*0*	*14*	*1*
P2	*4*	*7*	*3*
P3	*13*	*6*	*2*
P4	*18*	*11*	*2*
P5	*26*	*7*	*3*

Si mostri la sequenza di esecuzione e si calcolino il tempo di attesa e il tempo di completamento di ciascun processo, considerando i seguenti algoritmi di scheduling:

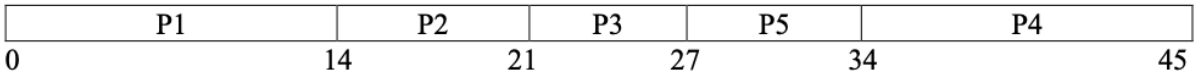
A. SJF con prelazione
B. Priorità senza prelazione

A. Sequenza di esecuzione:



Processo	*Tempo di Attesa*	*Tempo di Completamento*
P1	*13*	*27*
P2	*0*	*7*
P3	*0*	*6*
P4	*16*	*27*
P5	*0*	*8*

B. Sequenza di esecuzione:



Processo	*Tempo di Attesa*	*Tempo di Completamento*
P1	*0*	*14*
P2	*10*	*17*
P3	*8*	*14*
P4	*16*	*27*
P5	*1*	*8*