Università di Verona Mirco De Marchi

Deadlock
ESERCITAZIONE

L'obiettivo dell'esercitazione è implementare un algoritmo di prevenzione al deadlock, in particolare l'***Algoritmo del Banchiere***, in MentOS.

Alcune definizioni di teoria:

- ***Deadlock***: stato di un insieme di processi in concorrenza tra loro, ovvero che utilizzano risorse condivise, in cui almeno un processo è in attesa di un evento che può essere generato da un processo dello stesso insieme. Si ha deadlock se tutte le seguenti condizioni sono vere:
  - Mutua esclusione: esiste una risorsa che non può essere rilasciata.
  - Hold and Wait: un processo detiene una risorsa, ma non può rilasciarla, perché sta aspettando di acquisire una risorsa che è occupata da un altro processo.
  - No preemption: le risorse detenute dai processi non possono essere rilasciate da procedure esterne (es. kernel del sistema operativo), ma possono essere rilasciate solo dal processo stesso.
  - Attesa circolare: esiste un insieme di processi in attesa ciclica sulla liberazione di una risorsa.
- ***Stato safe***: date le risorse necessarie al completamento dell'esecuzione di ogni processo, un insieme di processi è in uno stato safe, se, con tutte le risorse rimanenti è possibile trovare una sequenza di allocazione delle risorse ai processi tale per cui tutti i processi riescono a terminare la loro esecuzione.
- ***Prevenzione statica***: si definiscono delle regole di gestione dei processi o delle procedure kernel che falsificano una delle condizioni necessarie al deadlock.
- ***Detection and recovery***: si implementa un algoritmo di rilevazione del deadlock e si agisce nel momento in cui il deadlock è già avvenuto, applicando un rollback dello stato dei processi o, nella peggiore delle ipotesi, riavviando il sistema.
- ***Prevenzione dinamica***: si implementa un algoritmo di prevenzione al deadlock gestendo l'allocazione delle risorse in modo da evitare uno stato unsafe. Difficile da implementare perché richiede conoscenza approfondita delle richieste di risorse. Un esempio di implementazione di algoritmo di prevenzione dinamica è l'*Algoritmo del Banchiere*.

Algoritmo del Banchiere

Strutture dati del problema:

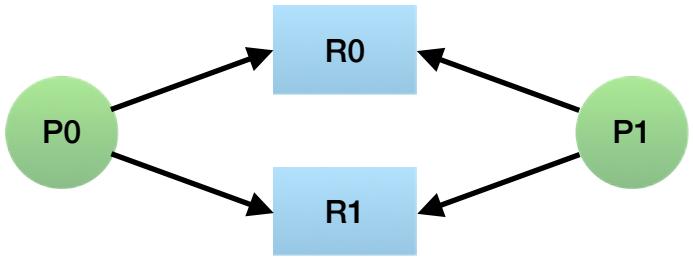
```
int available[m];     //< Istanze di una risorsa m disponibili.
int max[n][m];        //< Matrice delle richieste totali di risorse.
int alloc[n][m];      //< Matrice di allocazione corrente.
int need[n][m];       //< need[i][j] = max[i][j] - alloc[i][j]
```

Pseudocodice (vedi slides teoria):

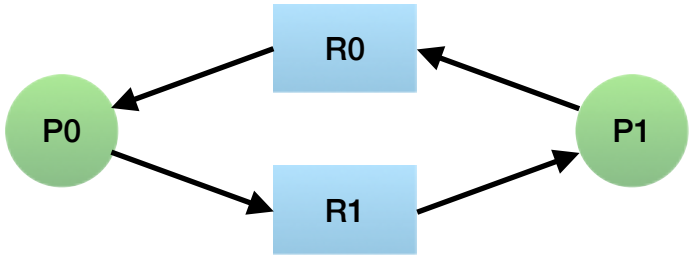
```
void request(int req_vec[]) {
      if (req_vec[] > need[i][])
            error();   //< Superato il massimo preventivato.

      if (req_vec[] > available[])
            wait();    //< Attendo che si liberino risorse.

      // Simulo l'allocazione richiesta.
      available[] = available[] - req_vec[];
      alloc[i][]  = alloc[i][] + req_vec[];
      need[i][]   = need[i][] - req vec[];

      if (!state_safe()) {
            // Se non è safe, rollback.
            available[] = available[] + req_vec[];
            alloc[i][] = alloc[i][] - req_vec[];
            need[i][] = need[i][] + req vec[];
            wait();
      }
}

boolean state_safe() {
      int work[m] = available[];
      boolean finish[n] = (FALSE,...,FALSE);
      int i;

      while (finish != (TRUE,...,TRUE)) {
            // Cerco i processi che possono terminare con le risorse
            // disponibili.
            for (i=0; (i<n)&&(finish[i]||(need[i][]>work[])); i++);
            if (i==n)
                  return FALSE; //< Stato unsafe.
            else {
                  work[] = work[] + alloc[I][];
                  finish[i] = TRUE; //< Processo i ha terminato.
            }
      }
      return TRUE;
}
```

Caso di studio

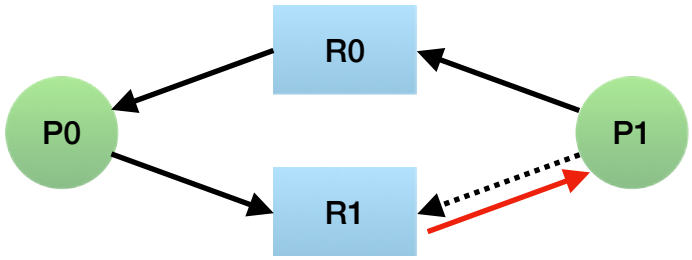
Ipotizziamo di avere 2 processi in esecuzione e in concorrenza tra loro, e 2 risorse condivise con 1 istanza ognuna. Entrambi i processi per completare la loro esecuzione hanno bisogno di entrambe le risorse.



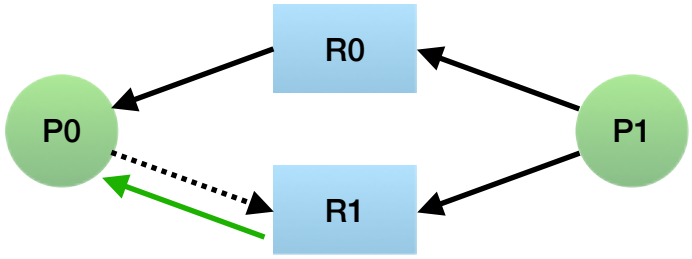
La domanda è se può esistere uno scheduling dei processi che porta l'insieme dei processi {P0, P1} in deadlock. In particolare, tralasciando lo scheduling di esecuzione dei processi, in questo caso è più significativa la sequenza di richieste a risorse generata dallo scheduling. Ad esempio con la sequenza P0 -> R0 e P1 -> R1, si raggiunge il seguente stato, che è uno stato di deadlock:



Tuttavia, visto che l'obiettivo è quello di implementare un algoritmo di prevenzione al deadlock, bisogna agire ancora prima del deadlock, ovvero dobbiamo identificare se una specifica richiesta di risorsa porta l'insieme dei processi {P0, P1} in uno stato unsafe. In questo esempio, la richiesta P1 -> R1 è una richiesta che porterebbe il sistema in uno stato unsafe (che in questo specifico caso corrisponde anche a uno stato di deadlock):



Se invece di P1 -> R1, avessimo P0 -> R0, allora la richiesta manterrebbe l'insieme dei processi {P0, P1} in uno stato safe:



Codice

Per completare questa esercitazione è prima necessario implementare almeno un algoritmo di scheduling e le syscall sui semafori. Quindi se si vuole scaricare da capo MentOS procedere nel seguente modo:

- `git clone https://github.com/mentos-team/MentOS.git`
- `git checkout feature/Feature-DeadlockExercise`
- Implementare ora un algoritmo di scheduling e le syscall dei semafori, oppure sovrascrivere i seguenti file con la propria implementazione precedentemente effettuata in un'altra directory:
  - *mentos/src/process/scheduler_algorithm.c*
  - *mentos/src/experimental/smart_sem_user.c*
  - *mentos/src/system/syscall.c*
  - *mentos/inc/system/syscall.h*

Mentre se si vuole utilizzare la stessa directory di lavoro che già in precedenza avete usato per implementare le syscall:

- `cd <mentos-proj-dir>`

- `git checkout feature/Feature-DeadlockExercise`
- `git pull`

Per procedere con l'esercitazione di Deadlock Prevention aprire il file *mentos/src/experimental/deadlock_prevention.c*. La richiesta è quella di completare le funzioni ***request*** e ***state_safe*** seguendo lo schema del template dato***.***

Per completare l'esercitazione è necessario usare delle funzioni algebriche su array e matrici. Queste funzioni sono già disponibili in MentOS in una libreria chiamata *arr_math.h*. Trovate la documentazione della libreria all'interno del header file *mentos/inc/experimental/math/arr_math.h*. Lascio qui una breve spiegazione di alcune di queste funzioni:

- `arr_g_any(op1, op2, length)`: ritorna `true` se esiste un elemento nel array `op1` maggiore del rispettivo elemento in `op2`, altrimenti `false`.
- `arr_g(op1, op2, length)`: ritorna `true` se ogni elemento nel array `op1` è maggiore del rispettivo elemento in `op2`, altrimenti `false`.
- `arr_add(dst_op1, op2, length)`: esegue la somma elemento per elemento del array `dst_op1` e `op2`, e lo salva in `dst_op1`.
- `arr_sub(dst_op1, op2, length)`: esegue la sottrazione elemento per elemento del array `dst_op1` e `op2`, e lo salva in `dst_op1`.
- `all(dst, value, length)`: inizializza l'array `dst` con il valore `value`.
- `arr_ne(op1, op2, length)`: ritorna `true` se esiste un elemento nel array `op1` che è diverso del rispettivo elemento in `op2`, altrimenti `false`.

Test

L'*Algoritmo del Banchiere* implementato viene testato esattamente sul caso di studio descritto sopra.

- Condizioni iniziali: 2 processi P0 e P1, e 2 risorse condivise R0 e R1, di cui ognuna contenente un'istanza di risorsa.
- Conoscenza a priori: entrambi i processi hanno bisogno per il loro completamento di entrambe le risorse (come si può acquisire questa informazione in un sistema operativo?).

Compilazione ed esecuzione:

```
cd <mentos-proj-dir>
mkdir build
cd build
cmake -DENABLE_DEADLOCK_PREVENTION=ON ..
make
make qemu
```

Attenzione: se l'algoritmo di Deadlock Prevention non è ancora stato implementato, i comandi precedenti daranno errore di compilazione.

Simulazione deterministica del Algoritmo del Banchiere

Mentre MentOS si avvia, viene eseguita una simulazione deterministica del Algoritmo del Banchiere da voi implementato. Potete vedere l'output di questa simulazione nella finestra di debug, quella dove vengono stampate tutte informazioni aggiuntive relativa al funzionamento di MentOS. Scorrendo l'output un po' più in alto, subito prima del messaggio di creazione del processo *init,* troverete anche l'output di simulazione del algoritmo di Deadlock Prevention. Se l'implementazione del Algoritmo del Banchiere è corretta, l'output di simulazione dovrà essere uguale a quello mostrato sotto.

Durante la simulazione i processi possono effettuare 2 tipi di richieste:

- LOCK: richiesta di allocazione di una risorsa in cui viene specificato il processo che effettua la richiesta e quali risorse si vogliono utilizzare. Per semplicità durante la simulazione verranno allocate 1 sola risorsa alla volta.

- FREE: richiesta di liberazione di una risorsa in cui viene specificato il processo che effettua la richiesta e quali risorse non si vogliono più utilizzare.

Dopodiché il sistema risponderà con lo stato del sistema, che può essere:
- SAFE: il processo può prendersi la risorsa.
- WAIT: il processo deve aspettare.
- WAIT UNSAFE: il processo deve aspettare altrimenti porterebbe il sistema in uno stato unsafe.
- ERROR: errore di sistema.

Output corretto di simulazione:

```
Tasks N:      2
Resources M:  2
AVAILABLE: { R_0: 1, R_1: 1 }
 --------------------
|        MAX         |
| Task | R_0  | R_1  |
|    0 |    1 |    1 |
|    1 |    1 |    1 |
 --------------------
 --------------------
|       ALLOC        |
| Task | R_0  | R_1  |
|    0 |    0 |    0 |
|    1 |    0 |    0 |
 --------------------
 --------------------
|        NEED        |
| Task | R_0  | R_1  |
|    0 |    1 |    1 |
|    1 |    1 |    1 |
 --------------------
SIMULATION START
LOCK (task: 0; req_vec: { R_0: 1, R_1: 0 }) SAFE: enjoy your resource
available: { R_0: 0, R_1: 1 }
 --------------------
|       ALLOC        |
| Task | R_0  | R_1  |
|    0 |    1 |    0 |
|    1 |    0 |    0 |
 --------------------
LOCK (task 1; rec_vec: { R_0: 0, R_1: 1 }) WAIT UNSAFE: deadlock detected
available: { R_0: 0, R_1: 1 }
 --------------------
|       ALLOC        |
| Task | R_0  | R_1  |
|    0 |    1 |    0 |
|    1 |    0 |    0 |
 --------------------
LOCK (task: 0; req_vec: { R_0: 0, R_1: 1 }) SAFE: enjoy your resource
available: { R_0: 0, R_1: 0 }
 --------------------
|       ALLOC        |
| Task | R_0  | R_1  |
|    0 |    1 |    1 |
|    1 |    0 |    0 |
 --------------------
```

Matrice delle richieste massime di risorse che i processi potrebbero richiedere per il loro completamento.
Conoscenza a priori del sistema.

Matrice delle allocazioni correnti delle risorse.

Matrice delle risorse necessarie ai processi al tempo corrente per il completamento dell'esecuzione.

Se permettessi a questo processo di prendersi la risorsa, porterei il sistema in una possibile situazione di deadlock.

```
LOCK (task 1; req_vec: { R_0: 0, R_1: 1 }) WAIT: resource busy
available: { R_0: 0, R_1: 0 }
 --------------------
|       ALLOC        |
| Task | R_0  | R_1  |
|    0 |    1 |    1 |
|    1 |    0 |    0 |
 --------------------
LOCK (task 0; rec_vec: { R_0: 0, R_1: 1 }) ERROR: max matrix overflow
available: { R_0: 0, R_1: 0 }
 --------------------
|       ALLOC        |
| Task | R_0  | R_1  |
|    0 |    1 |    1 |
|    1 |    0 |    0 |
 --------------------
FREE (task 0; rec_vec: { R_0: 0, R_1: 1 })
available: { R_0: 0, R_1: 1 }
 --------------------
|       ALLOC        |
| Task | R_0  | R_1  |
|    0 |    1 |    0 |
|    1 |    0 |    0 |
 --------------------
LOCK (task: 1; req_vec: { R_0: 0, R_1: 1 }) SAFE: enjoy your resource
available: { R_0: 0, R_1: 0 }
 --------------------
|       ALLOC        |
| Task | R_0  | R_1  |
|    0 |    1 |    0 |
|    1 |    0 |    1 |
 --------------------
FREE (task 0; rec_vec: { R_0: 1, R_1: 0 })
available: { R_0: 1, R_1: 0 }
 --------------------
|       ALLOC        |
| Task | R_0  | R_1  |
|    0 |    0 |    0 |
|    1 |    0 |    1 |
 --------------------
LOCK (task: 1; req_vec: { R_0: 1, R_1: 0 }) SAFE: enjoy your resource
available: { R_0: 0, R_1: 0 }
 --------------------
|       ALLOC        |
| Task | R_0  | R_1  |
|    0 |    0 |    0 |
|    1 |    1 |    1 |
 --------------------
FREE (task 1; rec_vec: { R_0: 1, R_1: 0 })
available: { R_0: 1, R_1: 0 }
 --------------------
|       ALLOC        |
| Task | R_0  | R_1  |
|    0 |    0 |    0 |
|    1 |    0 |    1 |
 --------------------
```

Errore perché il processo P0 sta richiedendo più risorse di quante ne necessita per arrivare a completamento. Situazione che non si dovrebbe mai verificare in un sistema operativo, per questo è così importante la conoscenza a priori.

Questa operazione è SAFE perché assumiamo che quando un processo ha finito di usare una risorsa, non gli servirà mai più.

```
FREE (task 1; rec_vec: { R_0: 0, R_1: 1 })
available: { R_0: 1, R_1: 1 }
 --------------------
|       ALLOC        |
| Task | R_0  | R_1  |
|    0 |    0 |    0 |
|    1 |    0 |    0 |
 --------------------
FREE (task 1; rec_vec: { R_0: 0, R_1: 1 }) ERROR: try to free a resource not own
available: { R_0: 1, R_1: 1 }
 --------------------
|       ALLOC        |
| Task | R_0  | R_1  |
|    0 |    0 |    0 |
|    1 |    0 |    0 |
 --------------------
```


Errore perché il processo P1 ha tentato di liberare una risorsa che non possedeva.

Simulazione realistica del Algoritmo del Banchiere

Tra i vari comandi shell che MentOS rende disponibili, esiste anche un comando che esegue esattamente la situazione descritta precedentemente nel nostro caso di studio, però con processi veri. Il comando è il seguente:

```
deadlock [-i <iter>]
```

Il comando esegue 2 processi che sono sincronizzati in modo errato su 2 risorse condivise, e quindi c'è il rischio che possano andare in deadlock. Opzionalmente si può specificare il numero di iterazioni di sincronizzazione che i 2 processi eseguono sulle 2 risorse attraverso con il parametro -i <iter>, mentre senza il parametro specificato eseguirebbe solo 1 iterazione. Questo parametro aggiuntivo ci permette di raggiungere più velocemente la condizione di deadlock, senza dover eseguire ripetutamente lo stesso comando.

Se avete implementato correttamente l'algoritmo di prevenzione al deadlock, questo comando dovrebbe sempre terminare e ritornare il controllo alla shell. Altrimenti se provate ad avviare MentOS senza Deadlock Prevention è possibile, con questo comando, mandare il sistema in deadlock:

```
cd <mentos-proj-dir>
mkdir build
cd build
cmake ..
make
make qemu
```

Eseguito poi il login in MentOS si può eseguire il comando ripetutamente:

```
deadlock
```

Oppure per velocizzare il raggiungimento dello stato di deadlock

```
deadlock -i 100
```

Osservate che non si può determinare in quale momento dell'esecuzione del comando avviene deadlock, poiché lo schedule dei processi non è deterministico.

Riassunto info utili

Setup repository MentOS:

- `git clone https://github.com/mentos-team/MentOS.git`
- `git checkout feature/Feature-DeadlockExercise`

Codice da modificare per implementare l'algoritmo del banchiere:

<mentos-proj-dir>/mentos/src/experimental/deadlock_prevention.c

Codice e documentazione libreria **arr_math.h**:

<mentos-proj-dir>/mentos/inc/experimental/math/arr_math.h
<mentos-proj-dir>/mentos/src/experimental/math/arr_math.c

Codice parte Syscall semafori:

<mentos-proj-dir>/mentos/src/experimental/smart_sem_user.c
<mentos-proj-dir>/mentos/src/system/syscall.c
<mentos-proj-dir>/mentos/inc/system/syscall.h

Eseguire MentOS **senza** deadlock prevention:

```
cd <mentos-proj-dir>
mkdir build && cd build
cmake ..
make qemu
```

Eseguire MentOS **con** deadlock prevention:

```
cd <mentos-proj-dir>
mkdir build && cd build
cmake -DENABLE_DEADLOCK_PREVENTION=ON ..
make qemu
```

Comando shell di MentOS per simulazione realistica:

```
deadlock [-i <iterations>]
```