Esame di Sistemi Operativi
AA 2017/18
16 Febbraio 2018

Nome	Cognome	Matricola

Istruzioni

Scrivere il proprio nome e cognome su ogni foglio dell'elaborato. Usare questo testo come bella copia per le risposte, utilizzando l'apposito spazio in calce alla descrizione dell'esercizio.

Esercizio 1

Sia data la seguente tabella che descrive il comportamento di un insieme di processi periodici

processo	**tempo di inizio**	**CPU burst**	**IO burst**
P1	0	5	5
P2	1	1	5
P3	3	10	1

Domanda Si assuma di disporre di uno scheduler preemptive *Round Robin* (RR) con quanto di tempo $T = 5$. Si assuma inoltre che:

- l'operazione di avvio di un processo lo porti nella coda di ready, ma **non** necessariamente in esecuzione
- il termine di un I/O porti il processo che termina nella coda di ready, ma **non** in esecuzione.

. Si illustri il comportamento dello scheduler in questione nel periodo indicato, avvalendosi degli schemi di seguito riportati.

Soluzione Assumendo che all'arrivo contemporaneo di due processi, essi vengano messi nella coda di wait in ordine crescente di PID, in Figura 1 e' illustrata la traccia di esecuzione dello scheduler specificato. I cicli di cpu burst sono indicati in verde ed in rosso l'I/O burst; le caselle azzurre indicano l'arrivo di un nuovo processo.

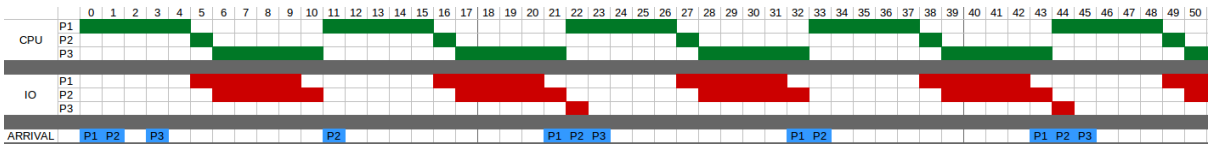


Figure 1: RR Scheduler

Nome	Cognome	Matricola

Esercizio 2

Sia data la seguente traccia di accesso alle pagine di memoria:

1 2 3 7 5 2 3 8 9 1 1 4 8 1 1.

Domanda Si assuma di avere un TLB di 4 slot gestito in modalita' LRU. Di quanto migliorano le prestazioni della memoria raddoppiandolo?

Soluzione Il tempo di accesso medio T^4_{mean} supponendo che il TLB abbia 4 unita' e' dato dalla seguente relazione:

$$T^4_{mean} = \overbrace{4(T_{TLB} + 2T_{fetch})}^{1,2,3,7} + \overbrace{(T_{TLB} + 2T_{fetch})}^{5} + \overbrace{(T_{TLB} + T_{fetch})}^{2} + \overbrace{(T_{TLB} + T_{fetch})}^{3} + \\ + \overbrace{(T_{TLB} + 2T_{fetch})}^{8} + \overbrace{(T_{TLB} + 2T_{fetch})}^{9} + \overbrace{(T_{TLB} + 2T_{fetch})}^{1} + \overbrace{(T_{TLB} + T_{fetch})}^{1} + \\ + \overbrace{(T_{TLB} + 2T_{fetch})}^{4} + \overbrace{(T_{TLB} + T_{fetch})}^{8} + \overbrace{(T_{TLB} + T_{fetch})}^{1} + \overbrace{(T_{TLB} + T_{fetch})}^{1}$$

Analogamente, nel caso in cui il TLB abbia 8 unita', il tempo medio sara' calcolato come segue:

$$T^8_{mean} = \overbrace{5(T_{TLB} + 2T_{fetch})}^{1,2,3,7,5} + \overbrace{(T_{TLB} + T_{fetch})}^{2} + \overbrace{(T_{TLB} + T_{fetch})}^{3} + \\ + \overbrace{(T_{TLB} + 2T_{fetch})}^{8} + \overbrace{(T_{TLB} + 2T_{fetch})}^{9} + \overbrace{(T_{TLB} + T_{fetch})}^{1} + \overbrace{(T_{TLB} + T_{fetch})}^{1} + \\ + \overbrace{(T_{TLB} + 2T_{fetch})}^{4} + \overbrace{(T_{TLB} + T_{fetch})}^{8} + \overbrace{(T_{TLB} + T_{fetch})}^{1} + \overbrace{(T_{TLB} + T_{fetch})}^{1}$$

Il guadagno di prestazioni sara' dato semplicemente dal rapporto tra T^4_{mean} e T^8_{mean}.

Nome	Cognome	Matricola

Esercizio 3

Cosa succede al seguente programma se si eliminano i mutex?

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NUM_THREADS 10
#define M 1000000
typedef struct {
  float* src;
  float* dst;
  int dim;
  pthread_mutex_t mtx;
} ThreadArgs;

void* runner(void* arg_) {
  ThreadArgs* args=(ThreadArgs*) arg_;
  pthread_mutex_lock(&args->mtx);
  for(int i=0; i<args->dim; ++i)
    args->dst[i]+=args->src[i];
  pthread_mutex_unlock(&args->mtx);
  return 0;
}

float srcs[NUM_THREADS][M];
float dests[NUM_THREADS][M];
ThreadArgs args[NUM_THREADS];
pthread_t threads[NUM_THREADS];

int main () {
  for (int i=0; i<NUM_THREADS; ++i) {
    printf("%d \n", i);
    args[i].src=srcs[i];
    args[i].dst=dests[i];
    args[i].dim=M;
    pthread_mutex_init(&args->mtx, NULL);
    pthread_create(&threads[i], NULL, runner, (void*) &args[i]);
  }
  for (int i=0; i<NUM_THREADS; ++i) {
    void* retval=0;
    pthread_join(threads[i], &retval);
    pthread_mutex_destroy(&args->mtx);
  }
}
```

Soluzione Eliminando i mutex, non succede **niente**. Essi infatti non sono utili poiche' ogni thread ha un suo vettore `src` e `dest`.

Nome	Cognome	Matricola

Esercizio 4

Sia dato un sottosistema di memoria con paginazione, caratterizzato dalle seguenti dimensioni:

- frame 4KB
- memoria fisica indirizzabile 16MB

Domande

- Si calcoli il numero di bit necessari per individuare una pagina in un indirizzo virtuale
- Si calcoli inoltre la probabilità di page fault considerando che un accesso al TLB impiega 1 ns, un ciclo di lettura scrittura impieghi 100 ns ed il tempo di accesso medio alla memoria sia di 120 ns.

Soluzione

- poiche' ogni frame necessita di 12 bit - 4KB = 2^{12} - e 16MB di memoria fisica necessitano di 24 bit per lindirizzamento, il numero di bit per il page number sara' $24 - 12 = 12$.
- Supponendo che l'*hit ratio* abbia valore p, avremo il seguente tempo di accesso in memoria effettivo - aka Effective Access Time:

$$\texttt{EAT} = [p \cdot (1 + 100) + (1 - p) \cdot (2 + 200)]\ \texttt{ns} \tag{1}$$

  Inoltre, sappiamo che il tempo di accesso medio e' di 120 `ns`, da cui la seguente relazione:

$$[p \cdot (1 + 100) + (1 - p) \cdot (2 + 200)]\ \texttt{ns} = 120\ \texttt{ns} \tag{2}$$

  Risolviamo quindi la Equazione 2 in p per trovare la probabilita' desiderata come $q = 1 - p$.

Nome	Cognome	Matricola

Esercizio 5

Cos'e' la legge di Little (Little's Law). (hint: teoria delle code, produttori/consumatori).

Soluzione La legge di Little e' usata per valutare i vari algoritmi di scheduling, mettendo in relazione la dimensione media della coda n con il tempo di attesa medio W e frequenza media di arrivo dei processi nella coda λ. In particolare, essa e' descritta dalla relazione:

$$n = \lambda \cdot W \tag{3}$$

Tale legge si inserisce nello studio degli algoritmi di scheduling mediante *Queuing Models*. A differenza di una valutazione analitica in cui bisogna conoscere il workload, un'analisi tramite Queuing Models assume che quest'ultimo sia sconosciuto. In tal caso, si effettuera' una stima probabilistica basata sulla distribuzione di CPU ed I/O bursts e sulla distribuzione dei tempi di arrivo dei processi.

Nome	Cognome	Matricola

Esercizio 6

Sia data la seguente tabella di processi/risorse.

Allocated:

Processo	R1	R2	R3
P0	0	0	1
P1	0	0	0
P2	6	0	2

Max:

Processo	R1	R2	R3
P0	5	4	1
P1	2	0	3
P2	6	6	2

Available:

R1	R2	R3
0	6	1

Domande Si illustri l'evoluzione dell'algoritmo del banchiere nella gestione dei deadlock.

Soluzione Una soluzione safe calcolata tramite dall'Algoritmo del Banchiere e' la seguente: $P_2 \rightarrow P_0 \rightarrow P_1$. Di seguito l'evoluzione dell'algoritmo

```
*****************************************
op: 0
next_op: 1
p: 0
work:
SIZE: 3
0 6 1
needed:
SIZE: 3
5 4 0
reject

*****************************************
op: 0
next_op: 2
p: 1
work:
SIZE: 3
0 6 1
needed:
SIZE: 3
2 0 3
reject

*****************************************
op: 0
next_op: 3
p: 2
work:
SIZE: 3
```

```
0 6 1
needed:
SIZE: 3
0 6 0
accept

****************************************
op: 1
next_op: 2
p: 0
work:
SIZE: 3
6 6 3
needed:
SIZE: 3
5 4 0
accept

****************************************
op: 2
next_op: 3
p: 1
work:
SIZE: 3
6 6 4
needed:
SIZE: 3
2 0 3
accept
is_safe: 1
final_order:
SIZE: 3
2 0 1
```

Nome	Cognome	Matricola

Esercizio 7

Illustrare un esempio di mutex implementato mediante l'istruzione atomica `testAndSet`.

Soluzione L'istruzione atomica `testAndSet` permette di testare e modificare il contenuto di una word tramite una singola azione che non e' possibile interrompere. Quindi, se piu' di una `testAndSet` viene eseguita da diversi processori, queste verranno eseguite *sequenzialmente* - secondo un'ordine arbitrario. Per questo motivo `testAndSet` e' usata come base dei *mutex*, in modo da regolare l'accesso a sezioni critiche di codice. L'istruzione e' cosi' implementata:

```
boolean test_and_set(boolean *target) {
  boolean rv = *target;
  *target = true;
}
```

Quindi, un *mutex* potra' essere implementato tramite l'istruzione `testAndSet` come riportato nella seguente porzione di codice:

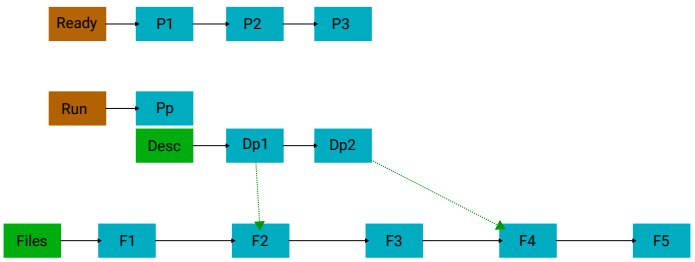
```
do {
  while (test_and_set(&lock))
    ; /* do nothing */
  /* critical section */
  lock = false;
  /* remainder section */
} while (true);
```

Nome	Cognome	Matricola

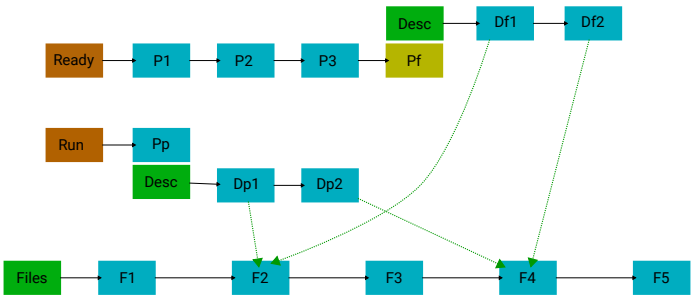
Esercizio 8

Domanda Cosa succede durante la system call `fork`? Illustrare in dettaglio i passi necessari alla sua esecuzione, evidenziando come vengono modificate le strutture nel kernel.

Soluzione La syscall `fork` e' usata per creare un nuovo processo - *children* - a partire da un processo *parent*. Il processo creato, avra' una copia dell'adress space del parent, consentendo di comunicare facilmente tra loro. I processi, in questo caso, continueranno l'esecuzione concorrentemente. Il child, inerita anche i privilegi e gli attributi nel parent, nonche' alcune risorse (quali i file aperti). Una volta creato il processo, se non viene invocata l'istruzione `exec()` il child sara' una mera copia del parent (con una propria copia dei dati); in caso contrario sara' possibile eseguire un diveso comando.



(a)



(b)

Figure 2: Questa immagine schematizza cio' che avviene durante la chiamata a `fork()`. In Figura 2a, il processo P_P esegue una fork. Il processo creato P_F viene portato in coda di ready - o comunque schedulato secondo l'algoritmo utilizzato - come riportato in Figura 2b. P_F eredita, oltre ad una copia dello stack di P_P, le sue risorse.

Il parent aspetta che il child completi il suo task tramite l'istruzione `wait()`; quando il child finisce la sua esecuzione (o avviene una chiamata `exit()`), il parent riprende il suo flusso, come riportato in Figura 3.

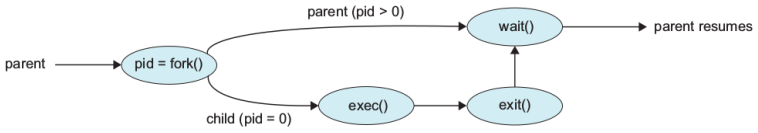


Figure 3: Creazione di un processo tramite syscall `fork()`.

Nome	Cognome	Matricola

Esercizio 9

Domanda Cos'e' una *Shared Memory*? Fornire un breve esempio del suo utilizzo.

Soluzione La *shared memory* e' un meccanismo usato per permettere a processi diversi di comunicare tra loro (Interprocess Communication - IPC). In questo caso, viene riservata una porzione di memoria condivisa tra i vari processi, i quali potranno scambiarsi informazioni semplicemente scrivendo e leggendo in tale porzione di memoria. I processi sceglieranno la locazione di memoria ed la tipologia di dati; essi dovranno anche sincronizzarsi in modo da non operare contemporaneamente sugli stessi dati. La shared memory, per esempio, e' molto utile nel caso di problemi *producer/consumer* - o analogamente *client/server*. In questo caso, infatti, sara' necessario istanziare un buffer condiviso da entrambi i processi, in modo che il produttore possa rendere disponibile ai consumatori cio' che ha prodotto.

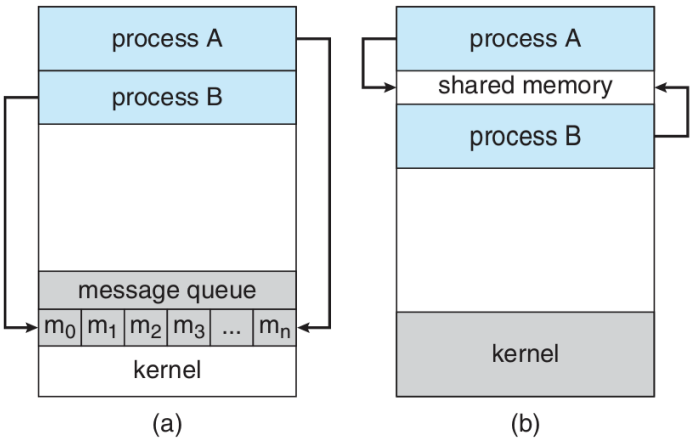


Figure 4: Diversi metodi di IPC: l'immagine *a* raffigura un IPC message-based; l'immagine *b* invece, una comunicazione basata su shared memory.

Un altro metodo di IPC prevede l'uso di *messaggi* (Message Passing). In questo caso, i processi comunicheranno inviandosi dei messaggi che saranno gestiti tramite opportune syscall - creando quindi overhead. Questi ultimi sono da favorire nel caso in cui i dati da veicolare abbiano una dimensione ridotta o nel caso di architetture fortemente multicore - per evitare problemi di coerenza delle cache. Entrageditmbi i metodi sono riportati nella Figura 4.

Nome	Cognome	Matricola

Esercizio 10

Domanda Cos'e' la tabella delle syscall? Cos'e' invece l'interrupt vector?

Soluzione L'interrupt vector e' un vettore di puntatori a funzioni; queste ultime saranno le *Interrupt Service Routine* (ISR) che gestiranno i vari interrupt.

Analogamente, la tabella delle syscall conterra' in ogni locazione il puntatore a funzione che gestisce quella determinata syscall. Alla tabella verra' associata anche un vettore contenente il numero e l'ordine di parametri che detta syscall richiede.