Esame di Sistemi Operativi
AA 2017/18
22 Ottobre 2018

Nome	Cognome	Matricola

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	4	6
P2	1	2	3
P3	3	10	2

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 Date queste premesse, la traccia di esecuzione dei processi e riportata nella Figura 1

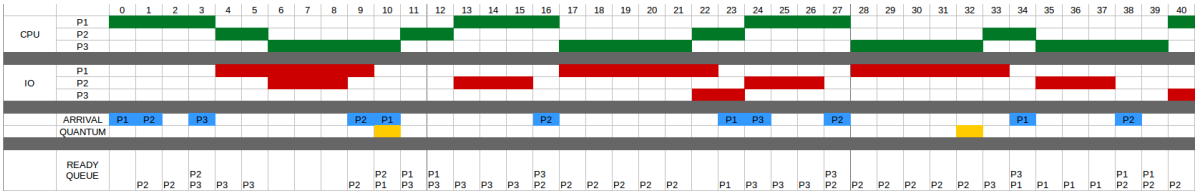

Figure 1: Traccia di esecuzione dei processi con *Round Robin* e time quantum $T = 5$. In verde sono indicati i cicli di `CPU burst` ed in rosso quelli di `I/O`. L'arrivo ed il termine di un quanto sono indicati reispettivamente in blu e giallo.

Nome	Cognome	Matricola

Esercizio 2

Si consideri in sottosistema di memoria il caratterizzato dalle seguenti tabelle

Segments:

Number	Base	Limit
0x0	0x00	0x02
0x1	0x02	0x01
0x2	0x04	0x01
0x3	0x05	0x02

Pages:

Page	Frame
0x00	0x07
0x01	0x06
0x02	0x05
0x03	0x04
0x04	0x03
0x05	0x02
0x06	0x01
0x07	0x00

Domanda Assumendo che le pagine abbiano una dimensione di 256 byte, che la tabella delle pagine consista di 256 elementi e che la tabella dei segmenti possa contenere 16 elementi, come vengono tradotti in indirizzi fisici i seguenti indirizzi logici?

- 0x10012
- 0x00134
- 0x30156
- 0x30300

Soluzione Ogni indirizzo virtuale è così composto: 0x10012 $\rightarrow$ 0x $\overset{\texttt{segnum}}{1}$ $\underbrace{00}_{\texttt{offset}}$ $\overset{\texttt{spiazzamento}}{12}$.

Usiamo il primo *nibble* per individuare la `base` dalla tabella dei segmenti. In seguito, il `frame` sarà individuato da confrontando `base` + `offset` con le entries della tabella delle pagine. Infine, l'indirizzo fisico sarà semplicemente 0x[`frame spiazzamento`]. Avremo quindi:

- 0x10012 $\rightarrow$ 0x0512
- 0x00134 $\rightarrow$ 0x0634
- 0x30156 $\rightarrow$ 0x0156
- 0x30300 $\rightarrow$ invalid address - limit exceeded.

Nome	Cognome	Matricola

Esercizio 3

Si consideri un file consistente in 100 blocchi e che il suo *File Control Block* (FCB) sia gia' in memoria. Siano dati due file-systems gestiti rispettivamente tramite allocazione a lista concatenata (*Linked List Allocation*) e allocazione indicizzata (*Indexed Allocation*). Si assuma che nel caso indicizzato il FCB sia in grado di contenere i primi 200 blocchi del file.

Domanda Si calcolino le operazioni di I/O su disco necessarie per eseguire le seguenti azioni in entrambi i file-systems:

- Rimozione di un blocco all'inizio del file
- Rimozione di un blocco a meta' del file
- Rimozione di un blocco alla fine del file

Soluzione Nel caso di file-system con *Indexed Allocation*, ogni file conterra un index block, ovvero un blocco contenente i puntatori a tutti gli altri blocchi componenti il file. Cio' implica che per raggiungere un dato blocco bastera' effettuare una semplice indicizzazione. Nell'implementazione con *Linked List Allocation* invece, ogni blocco contiene il riferimento al precedente ed al successivo. In questo caso, per effettuare operazioni che non siano all'inizio del file bisognera' scorrere la lista fino al blocco desiderato ed in seguito eseguire l'operazione necessaria.

Date queste premesse, i risultati sono:

1. Linked Allocation: 1 `IO-ops`; Indexed Allocation: 0 `IO-ops`
2. Linked Allocation: 52 `IO-ops`; Indexed Allocation: 0 `IO-ops`
3. Linked Allocation: 100 `IO-ops`; Indexed Allocation: 0 `IO-ops`

Nome	Cognome	Matricola

Esercizio 4

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.

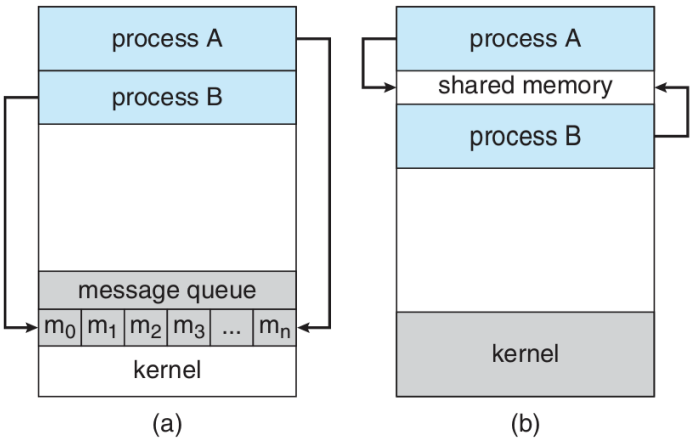

Figure 2: Diversi metodi di IPC: l'immagine *a* raffigura un IPC message-based; l'immagine *b* invece, una comunicazione basata su shared memory.

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 2.

Nome	Cognome	Matricola

Esercizio 5

Siano dati i seguenti algoritmi di *page replacement*:

- LRU
- FIFO
- Second-chance
- Optimal

Domande

(A) Ordinare gli algoritimi in base al loro *page-fault rate*.

(B) Evidenziare gli algoritmi che soffrono dell'anomalia di Belady.

Soluzione Partendo dall'algoritmo con le migliori performances in termini di *page-fault rate*, avremo:

#	**Algorithm**	**Belady**
1.	Optimal	no
2.	LRU	no
3.	Second Chance	si'
4.	FIFO	si'

Nome	Cognome	Matricola

Esercizio 6

Sia dato il seguente schema produttore consumatore:

```
#define BUFFERSIZE 256
int data[BUFFERSIZE];
uint8_t first_idx;
uint8_t size;
pthread_mutex_t mtx;

void producer(void) {
  // do stuff;
  int d = computeNewData();
  int ok = 0;
  while(!ok) {
    pthread_mutex_lock(&mtx);
    if (size < BUFFERSIZE) {
      buffer[(first_idx+size)%BUFFERSIZE]=d;
      size++;
      ok=true;
    }
    pthread_mutex_unlock(&mtx);
  }
}

void consumer(void) {
  int ok = 0;
  int d = 0;
  while(!ok){
    pthread_mutex_lock(&mtx);
    if (size > 0) {
      d=buffer[first_idx%BUFFERSIZE];
      ++first_idx;
      --size;
      ok=true;
    }
    pthread_mutex_unlock(&mtx);
  }
  doStuff(d);
}

```

Domanda Come si possono modificare i due processi in modo da aumentare l'efficienza del sistema?

Soluzione E' possibile usare i semofori per tale problema, aggiungendo due semafori: `semFill = 0` e `semEmpty = BUFFERSIZE`. Quindi le funzioni `producer()` e `consumer()` diventano:

```
void producer(){
  semWait(semEmpty);
  pthread_mutex_lock(&mtx);

  // do stuff;
  int d = computeNewData();
  buffer[(first_idx+size)%BUFFERSIZE]=d;
  size++;

  pthread_mutex_unlock(&mtx);
  semPost(semFill);
}

```

```
1  void consumer(){
2    int d=0;
3
4    semWait(semFill);
5    pthread_mutex_lock(&mtx);
6    d=buffer[first_idx%BUFFERSIZE];
7    ++first_idx;
8    pthread_mutex_unlock(&mtx);
9    semPost(semEmpty);
10
11   doStuff(d);
12 }
13
```

Nome	Cognome	Matricola

Esercizio 7

Illustrare le differenze tra `fork()` e `vfork()`. Che cosa succede se a seguito di una `vfork` non viene fatta immediatamente una `exec()`?

Soluzione `vfork()` e' progettata come variante piu' efficiente della `fork()`, specializzata nel caso l'istruzione immediatamente successiva alla `fork()`, nel processo figlio sia una `exec()`. A differenza della `fork()`, la `vfork()` non replica la memoria del processo padre. Se la prima istruzione eseguita e' una `exec()`, l'operazione di copia non e' necessaria in quanto l'immagine del processo figlio verra' sovrascritta dalla `exec()`. Non chiamare la `exec()` dopo una `vfork()` genera comportamenti non specificati, in quanto la memoria del padre non e' replicata.

Nome	Cognome	Matricola

Esercizio 8

Cos'e' la legge di Amdahl? Cosa descrive tale legge?

Soluzione La legge di Amdahl permette di valutare il guadagno di performance derivante dal rendere disponibili più core computazionali ad una applicazione che ha componenti sia *seriali* che *parallele*. Indicando con $\mathcal{S}$ la porzione seriale dell'applicazione e con N il numero di core a disposizione, si avrà quindi la seguente relazione:

$$\texttt{GAIN} \leq \frac{1}{\mathcal{S} + \frac{1-\mathcal{S}}{N}} \tag{1}$$

E' bene notare che

$$\lim_{N \to \text{inf}} \texttt{GAIN} = \lim_{N \to \text{inf}} \frac{1}{\mathcal{S} + \frac{1-\mathcal{S}}{N}} = \frac{1}{\mathcal{S}} \tag{2}$$

Ovviamente il `GAIN` dipende anche da come è implementato nel dettaglio il sistema multi-core.

Nome	Cognome	Matricola

Esercizio 9

(A) Illustrare in modo conciso e preciso il meccanismo di *context switch*, avvalendosi di semplici illustrazioni.

(B) E' possibile implementare ugualmente un multitasking preemptive su una CPU priva di MMU? Motivare in modo sintetico la risposta.

Soluzione Per poter rimuovere dall'esecuzione un processo $\texttt{P}_0$ e quindi eseguire un nuovo processo $\texttt{P}_1$, il SO deve salvare lo stato corrente del processo $\texttt{P}_0$ in modo da poter rirpistinare l'esecuzione dello stesso in un secondo momento. Le informazioni di un processo sono contenute nel suo Process Control Block (PCB). Quindi, il *context switch* può essere riassunto visivamente nella Figura 3.

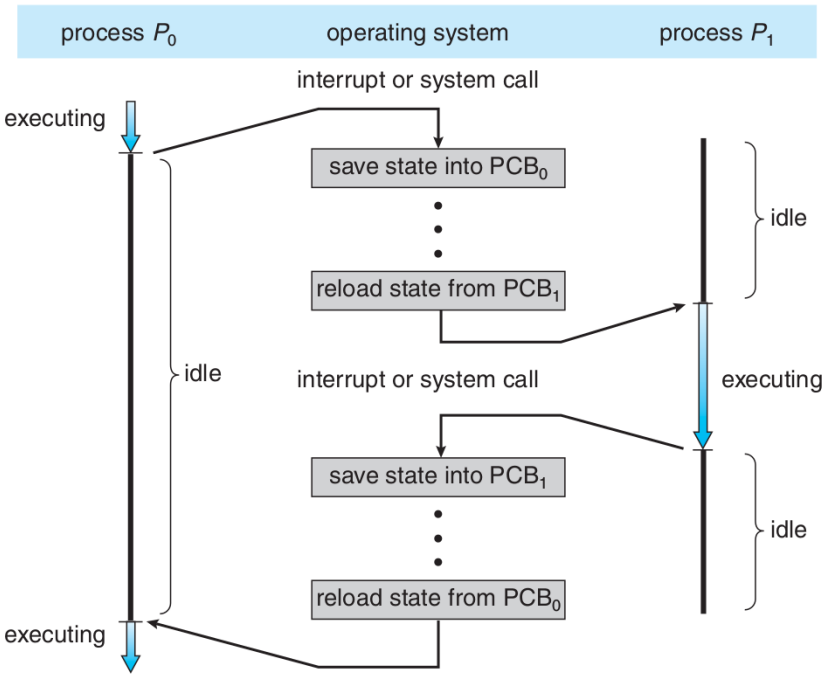

Figure 3: Esempio di *context switch*.

E' bene notare che il *context switch* è fonte di overhead a causa delle varie operazioni di preambolo e postambolo necessarie allo switch - e.g. salvare lo stato, blocco e riattivazione della pipeline di calcolo, svuotamento e ripopolamento della cache.

E' possibile implementare multitasking preemptive su una CPU priva di MMU poiché essa non è necessaria per il context switch.

Nome	Cognome	Matricola

Esercizio 10

Che relazione ce tra un *File Descriptor* ed una entry nella tabella globale dei file aperti del file system?

Soluzione Un File Descriptor consiste in un file handler che viene restituito ad un processo in seguito ad una chiamata alla syscall `open()`. In seguito a tale chiamata, il sistema scandisce il FS in cerca del file e, una volta trovato, il FCB e' copiato nella tabella globale dei file aperti. Per ogni singolo file aperto, anche se da piu' processi esiste una sola entry nella tabella globale dei file aperti.

Viene, quindi, creata una entry all'interno della tabella dei file aperti detenuta dal processo, la quale puntera' alla relativa entry nella tabella globale, insieme ad altre informazione - e.g. permessi, locazione del cursore all'interno del file, ecc. La syscall `open()` restituisce per l'appunto l'entry all'interno della tabella del processo - il File Descriptor. Piu' `open()` su uno stesso file da parte di uno stesso processo generano descrittori diversi.