Esame di Sistemi Operativi
AA 2018/19
15 Febbraio 2019
[soluzione]

Nome	Cognome	Matricola

Esercizio 1

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

Process	T_{start}	**CPU Burst 0**	**IO Burst 0**	**CPU Burst 1**	**IO Burst 1**
P1	0	3	2	7	1
P2	2	2	2	2	1
P3	3	4	4	6	6
P4	4	10	1	1	1

Domanda Si assuma di disporre di uno *scheduler preemptive* Shortest Remaining Job First (SRJF) con quanto di tempo pari a $T_q = 3$. Si assuma inoltre che:

- i processi in entrata alla CPU dichiarino il numero di burst necessari al proprio completamento;
- 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 necessariamente** in esecuzione.

Si illustri il comportamento dello scheduler in questione nel periodo indicato, avvalendosi degli schemi di seguito riportati (vedi pagina seguente).

Soluzione La traccia di esecuzione dei processi che soddisfi le specifiche di cui sopra e' riportata in Figura 1. Poiche' lo scheduler considera i burst *rimanenti* di ogni processo, il processo P4 non riuscira' ad essere eseguito fino a che gli altri non avranno finito.

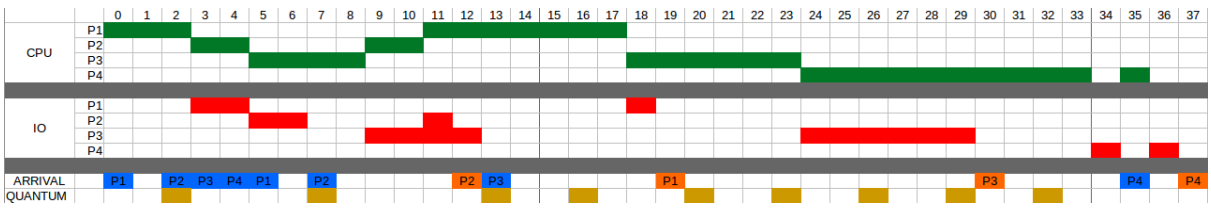


Figure 1: Traccia di esecuzione dei processi con scheduler SRJF e time quantum pari a $T_q = 3$. CPU burst, IO burst, arrivo e fine dei processi sono indicati rispettivamente in verde, rosso, blu e arancione. In giallo viene evidenziato l'intervento del time quantum.

Nome	Cognome	Matricola

Esercizio 2

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

- frame `4MB`
- memoria fisica indirizzabile `128GB`.

Domande Si calcolino quindi:

A. Il numero di bit minimo per indicizzare tutte le pagine

B. Il tempo di accesso medio al TLB, considerando che in media per accedere ad una pagina il sistema impiega 140ns, che $T_{RAM} = 110ns$ e che la probabilita' di trovare una pagina all'interno del TLB sia 0.85.

Soluzione

A. Data la dimensione di ogni pagina pari a `4MB`, saranno necessari `22 bit` per indicizzare un elemento all'interno della stessa. La memoria fisica, invece, necessita di almeno `37 bit`. Il numero di bit *minimo* per indicizzare tutte le pagine e' quindi pari a `37-22 = 15 bit`.

B. La formula per il calcolo del tempo di accesso medio alla memoria - *Effective Access Time* - e' data dalla seguente relazione:

$$T_{EAT} = p_{hit}(T_{TLB} + T_{RAM}) + (1 - p_{hit}) \cdot 2(T_{TLB} + T_{RAM}) \tag{1}$$

Sostituendo i dati della traccia nella (1) avremo $T_{TLB} = \frac{270}{23} \approx 11.739\ [ns]$.

Nome	Cognome	Matricola

Esercizio 3

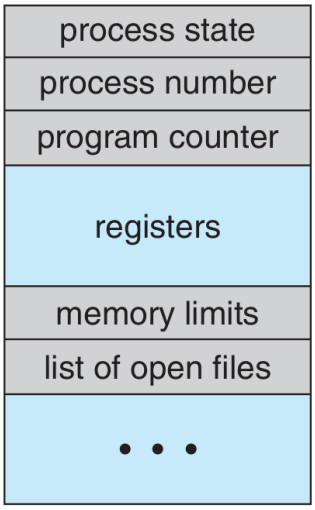
Cosa contiene il Process Control Block (PCB) di un processo? Illustrare inoltre il meccanismo di *Context Switch* (avvalendosi anche di schemi approssimativi).

Soluzione Il PCB di un processo contiene tutte le informazioni relative al processo a cui e' associato. Esempi di informazioni contenute nel PCB sono:

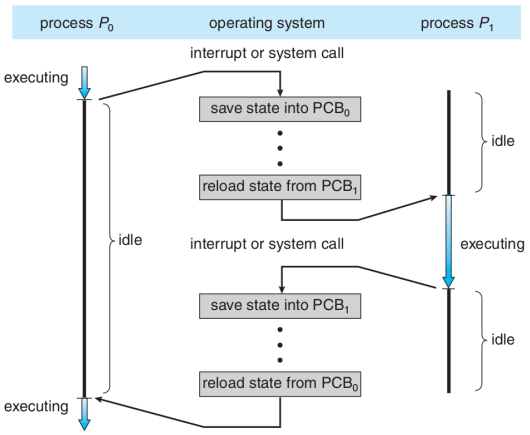
- stato del processo (running, waiting, zombie ...)
- Program Counter (PC), ovvero il registro contente la prossima istruzione da eseguire
- registri della CPU
- informazioni sulla memoria allocata al processo
- informazioni sull'I/O relativo al processo.

Una illustrazione di tale struttura dati e' riportata in Figura 2a.

Supponendo di avere due processi `P0` e `P1` e che il primo sia in running. Per mettere in esecuzione `P1`, l'Operating System (OS) deve salvare lo stato corrente di `P0` in modo da poter rirpistinare l'esecuzione dello stesso in un secondo momento. Quindi, il *context switch* può essere riassunto visivamente nella Figura 2b. E' bene notare che il *context switch* e' 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.



(a) Una rappresentazione grafica del PCB. Esso contiene tutte le informazioni relative alla gestione di un processo da parte dell'OS.



(b) Esempio di *context switch*.

Figure 2: Processi: PCB e *context switch*.

Nome	Cognome	Matricola

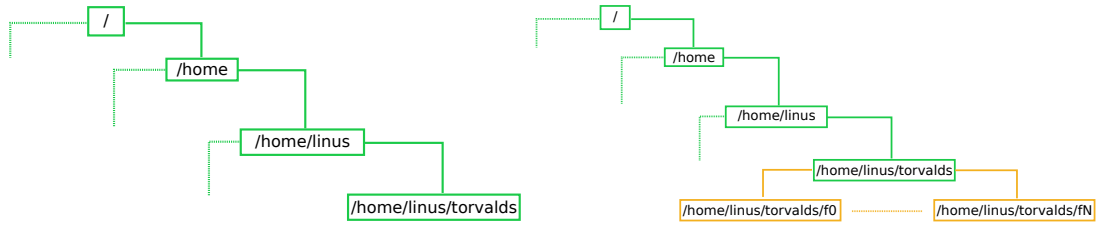
Esercizio 4

Sia dato un OS che operi su un disco primario `/dev/sda1`, quest'ultimo gestito da un File System (FS) `ext4`. Supponendo di collegare un secondo disco `/dev/sda2` formattato in `ext4`, spiegare in dettaglio cosa succede quando viene eseguito il comando

```
sudo mount /dev/sda2 /home/linus/torvalds
```

Si schematizzi, inoltre, l'albero delle directory prima e dopo l'esecuzione di tale comando. Cosa sarebbe cambiato se il disco `/dev/sda2` fosse stato formattato in `ExFAT`?

Soluzione Tramite il comando `mount` l'OS viene informato che un nuovo FS e' pronto per essere usato. L'operazione, quindi, provvedera' ad associarlo con un dato `mount-point`, ovvero la posizione all'interno della gerarchia del FS del sistema dove il nuovo FS verra' caricato. Prima di efferruare questa operazione di *attach*, ovviamente bisognera' controllare la tipologia e l'integrita' del FS. Una volta fatto cio', il nuovo FS sara' a disposizione del sistema (e dell'utente). In Figura 3 viene schematizzato tale processo.



(a) Overview prima dell'operazione di `mount`.

(b) Overview dopo l'operazione di `mount`.

Figure 3: Schematizzazione dei FS del sistema prima e dopo l'operazione di `mount`. La directory `/` rappresenta il `mount-point` del disco principale `/dev/sda1`, mentre il disco secondario `/dev/sda2` avra' come `root` la directory `/home/linus/torvalds`.

Ovviamente se il disco `/dev/sda2` fosse formattato in `ExFAT` non sarebbe cambiato niente per l'utente. Cio' poiche' tramite il layer denominato Virtual File System (VFS), diversi FS vengono uniformati sotto un'unica interfaccia. Cio' significa che qualunque tipo di FS supportato dal sistema verra' "visto" come una serie di file e directory, anche se organizzato in maniera diversa internamente.

Nome	Cognome	Matricola

Esercizio 5

A cosa serve la syscall `ioctl`? Come si puo' configurare la comunicazione con devices a caratteri e seriali in Linux?

Soluzione La syscall `ioctl` permette di interagire con il driver di un device generico - e.g. una webcam. Tramite essa sara' possibile ricavare e settare i parametri di tale device - e.g. ricavare la risoluzione della webcam o settarne la tipologia di acquisizione dati.

Per configurare devices seriali a caratteri - e.g. terminali - e' possibile usare le API racchiuse nella interfaccia `termios`. Tramite di essa, avremo accesso a tutte le informazioni relative al device - e.g. baudrate, echo, ...

Nome	Cognome	Matricola

Esercizio 6

Sia dato il seguente programma:

```
#define A_STEPS 2
#define B_STEPS 5

const char name_0[] = "A|";
const char name_1[] = "B|";
const char* name = name_0;

void fn0() {
  for (unsigned int i = 0; i < A_STEPS; i++) {
    printf("%s round: #%d \n", name, i);
    sleep(1);
  }
}

void fn1() {
  for (unsigned int i = 0; i < B_STEPS; i++) {
    printf("%s round: #%d \n", name, i);
    sleep(1);
  }
}

int main(int argc, char** argv) {
  printf("hello!\n");
  int f = fork();

  if (f < 0) {
    printf("%s exit", name);
    exit(EXIT_FAILURE);
  }

  if(0 == f) {
    name = name_1;
    fn1();
  } else {
    fn0();
  }

  printf("see ya!\n");
  exit(EXIT_SUCCESS);
}

```

Domanda Cosa stampa il programma? Che cosa succede quando `fn0` arriva alla fine del ciclo?

Soluzione Il programma esegue una `fork()`, andando a creare un processo figlio e, quindi, duplicando le variabili in memoria, eventuali *file descriptor* aperti, ecc. E' bene notare che il processo parent non effettua una `wait()`, quindi non aspettera' fine del child. Dato cio', il programma stampera' quanto segue:

```
hello!
A| round: #0
B| round: #0
A| round: #1
B| round: #1
see ya!
B| round: #2
B| round: #3
```

```
B| round: #4
see ya!
```

Il lettore noti che poiche' il parent non effettua la `wait()`, una volta terminato il suo ciclo, esso terminera' brutalmente, lasciando il child creato in precedenza orfano. Il child quindi verra' assegnato al processo master `init/systemd` - il primo processo spawnato all'avvio della macchina.

Nome	Cognome	Matricola

Esercizio 7

In relazione alla gestione della memoria, evidenziare le differenze tra *segmenti* e *pagine*.

Soluzione Gli indirizzi virtuali sono resi necessari per indicizzare un address space piu' grande del numero di registri disponibili. La memoria quindi viene organizzata in *segmenti* o *pagine* - o una combinazione di entrambe. Di conseguenza, i primi bit dall'indirizzo virtuale faranno riferimento ad uno di essi, mentre la restante parte indichera' lo spiazzamento all'interno del segmento o pagina selezionata.

I segmenti sono stati il primo approccio a tale problema. Ogni segmento ha dimensione variabile e percio' e' identificato da indirizzo di base e limite. Tali informazioni sono contenute in una tabella apposita (Segment Table). Ovviamente il sistema puo' proteggere alcuni segmenti non rendendoli accessibili tramite un bit di controllo. I contro della segmentazione sono principalmente frammentazione esterna e la necessita' di essere ricompattati.

Per far fronte alle pecche della segmentazione, architetture moderne usano la paginazione per virtualizzare l'address space. La memoria viene quindi divisa in frames (pagine) di dimensione uguale (prestabilita). La parte alta dell'indirizzo virtuale verra' usata come key per la tabella delle pagine (Page Table), andando ad evidenziare la pagina corrente. La rimanente parte dell'indirizzo verra' usata come offset all'interno della pagina. La paginazione permette anche di proteggere zone di memoria (tramite bit di protezione) e puo' essere implementata anche in maniera gerarchica (con n livelli di indirezione).

E' bene notare che entrambi gli approcci necessitano di strutture hardware dedicate per funzionare efficientemente.

Nome	Cognome	Matricola

Esercizio 8

Illustrare brevemente SLAB e Buddy allocator, sottolineandone le differenze ed i casi di utilizzo.

Soluzione Tra le piu' annoverate implementazioni di *Memory Allocator* traviamo: *slab* e *buddy* systems.

I. *Slab Allocator*: usato per allocare oggetti di dimensione fissa, puo' allocarne fino ad un numero massimo fissato. Il buffer viene quindi diviso in chunk di dimensioni `item_size`. Per poter organizzare il buffer viene quindi usata una struttura ausiliaria che tenga l'indice dei blocchi ancora liberi. Una *array list* soddisfa tale richiesta.

II. *Buddy Allocator*: usato per allocare oggetti di dimensione variabile. Il buffer viene partizionato ricorsivamente in 2 - creando di fatto un albero binario. La foglia piu' piccola che soddisfa la richiesta di memoria sara' ritornata al processo. Il *buddy* associato ad una foglia sara' l'altra regione ottenuta dalla divisione del parent. Ovviamente, se un oggetto e' piu' piccolo della minima foglia che lo contiene, il restante spazio verra' sprecato. Quando un blocco viene rilasciato, esso verra' ricompattato con il suo buddy (se libero), risalendo fino al livello piu' grande non occupato.