Esame di Sistemi Operativi
AA 2018/19
10 Settembre 2019
[soluzione]

Nome	Cognome	Matricola

Esercizio 1

Sia data la seguente tabella che descrive il comportamento di un insieme di processi periodici **real-time**.

Process	T_{start}	**Period**	**CPU Burst**
P1	0	3	1
P2	0	5	2
P3	0	6	1

Si assuma di disporre di uno scheduler preemptive Earliest Deadline First (EDF). Si assuma inoltre che:

- la deadline di ogni processo **coincida** con il suo periodo;
- nessuno dei processi debba attendere il rilascio di una risorsa posseduta da un altro processo;
- 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.

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

Soluzione Dato il requisito di esecuzione dei processi in *real-time* per prima cosa bisogna verificare che lo scheduler in questione possa garantire l'esecuzione di ogni ciclo di CPU burst entro la deadline specificata. Ricordando che in questo caso la deadline coincide con il periodo, calcoliamo quindi la

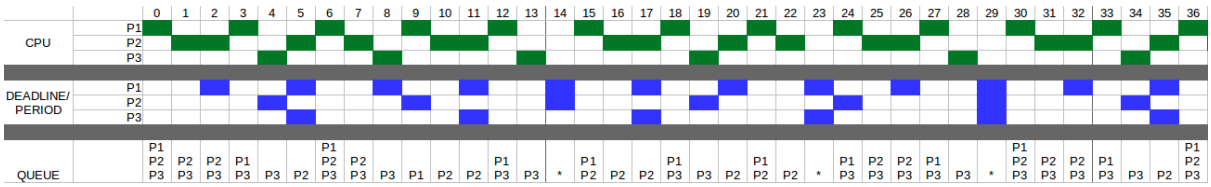


Figure 1: Traccia di esecuzione dei processi secondo l'algoritmo EDF. In blu sono scanditi i periodi dei processi, mentre in verde i cicli di CPU per ogni burst.

percentuale di utilizzo della CPU come

$$U_{CPU} = \sum_p \frac{t_p}{d_p} = \frac{1}{3} + \frac{2}{5} + \frac{1}{6} = 0.9. \tag{1}$$

Come si nota dalla Equazione 1, $U_{CPU} \leq 1$, quindi e' possibile effettuare lo scheduling tramite EDF rispettando il vincolo di esecuzione *real-time*. In base alle specifiche dell'algoritmo EDF, la traccia di esecuzione dei processi e' riportata in Figura 1. In questa traccia, quando due processi si trovano nelle stesse condizioni, si e' scelto di privilegiare il processo con `pid` minore.

Nome	Cognome	Matricola

Esercizio 2

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

- frame → `4 kB`
- memoria fisica indirizzabile → `32 GB`.

Si calcolino:

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

B. Il valore di p_{fault}, considerando che il tempo di accesso medio ad una pagina e' di 150 ns, $T_{RAM} = 100\ ns$ e $T_{TLB} = 10\ ns$.

Soluzione

A. Data la dimensione di ogni pagina pari a `4 kB`, saranno necessari `12 bit` per indicizzare un elemento all'interno della stessa. La memoria fisica, invece, necessita di almeno `35 bit`. Il numero di bit *minimo* per indicizzare tutte le pagine e' quindi pari a `35-12 = 23 bit`.

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

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

Sostituendo i dati della traccia nella (2) avremo $p_{fault} = 1 - p_{hit} = \frac{4}{11} \approx 0.36364$.

Nome	Cognome	Matricola

Esercizio 3

Con riferimento agli algoritmi di *Page Replacement*, enumerare i 4 principali algoritmi usati per tale scopo, ordinandoli in base al loro *page-fault rate* (dal piu' alto al piu' basso). Si evidenzino anche gli algoritmi che soffrono dell'anomalia di Belady.

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

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

Nome	Cognome	Matricola

Esercizio 4

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

Soluzione Tra le piu' annoverate implementazioni di *Memory Allocator* troviamo: *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.

Nome	Cognome	Matricola

Esercizio 5

Sia dato il seguente programma:

```
1
2   int main(int argc, char** argv) {
3     if (fork() == 0) {
4       if (fork() == 0) {
5         printf("A");
6         return 0;
7       } else {
8         wait(NULL);
9         printf("B");
10      }
11    } else {
12      if (fork() == 0) {
13        printf("C");
14        exit(EXIT_SUCCESS);
15      } else {
16        wait(NULL);
17      }
18      wait(NULL);
19      printf("D");
20    }
21
22    printf("E");
23    return 0;
24  }
25
```

Indicare quale dei seguenti puo' essere un possibile output sulla shell:

- `A`
- `CABEDE`
- `C`
- `CDA`
- `BA`
- `BAC`

Soluzione Tra quelli elencati, l'unico output valido sara' `CABEDE`, a causa delle `wait` posizionate in ogni processo padre. E' bene notare che la `return` a riga 6 e la `exit` a riga 14 non interrompono l'intero programma, ma soltanto l'esecuzione dei processi figli in cui sono locate - che *non* stampano il carattere '`E`' quindi.

Nome	Cognome	Matricola

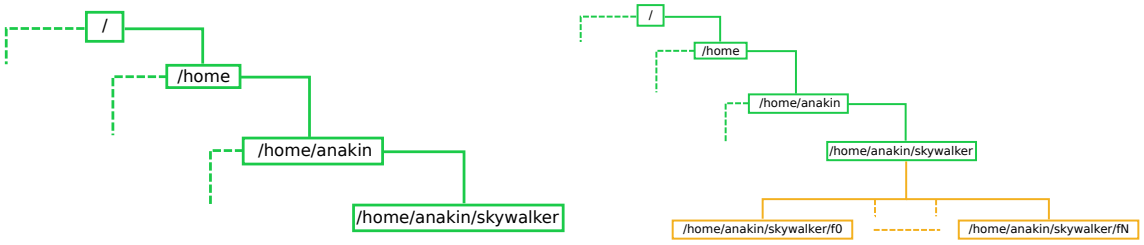
Esercizio 6

Sia dato una macchina con un generico Operating System (OS) installato. Si supponga che il disco primario `/dev/sda0` sia gestito da un File System (FS) `ext4` e che il suo *mount-point* sia `/`. Si supponga di collegare fisicamente alla macchina due dischi secondari - `/dev/sda1` e `/dev/sda2` - rispettivamente formattati in `FAT32` e `NTFS`.

Si schematizzi l'albero delle directory prima e dopo l'esecuzione del comando di cui sotto, spiegando brevemente come viene gestita dall'OS tale richiesta:

```
$ sudo mount /dev/sda2 /home/anakin/skywalker
```

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 effettuare 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 2 viene schematizzato tale processo. L'omogeneita' dell'interfaccia offerta all'utente e' garantita dal layer di Virtual File System (VFS) dell'OS. Tramite il VFS, dischi con FS diversi possono coesistere nello stesso sistema ed essere esplorati senza complicazioni da parte dell'utente.



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

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

Figure 2: Schematizzazione dei FS del sistema prima e dopo l'operazione di `mount`. La directory `/` rappresenta il `mount-point` del disco principale `/dev/sda0`, mentre il disco secondario `/dev/sda2` avra' come `root` la directory `/home/anakin/skywalker`. Gli elementi di quest'ultimo sono visualizzati in giallo nella Figura 2b.

Si noti che il disco `/dev/sda1` non e' visibile all'utente. Cio' poiche' il disco e' fisicamente attaccato alla macchina ma *non* e' stata effettuata alcuna operazione di `mount` per esso.

Nome	Cognome	Matricola

Esercizio 7

Richiede meno risorse la creazione di un nuovo `thread` o di un nuovo `processo`? Si motivi la risposta, evidenziando quali risorse vengono utilizzate in entrambi i casi.

Soluzione Creare un thread - sia esso kernel o user - richiede l'allocazione di una data structure contenente il register set, lo stack e altre informazioni quali la priorita', come riportato in Figura 3.

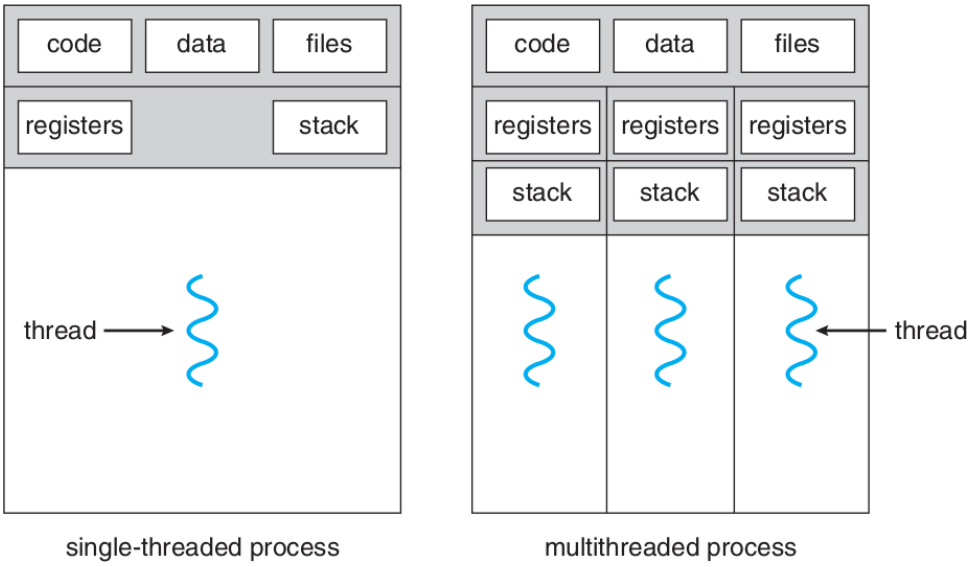


Figure 3: Processo single thread vs. multi-thread.

Creare un nuovo processo invece, richiede l'allocazione di un nuovo PCB (una data structure molto piu' pesante). Il Process Control Block (PCB) contiene tutte le informazioni del processo, quali `pid`, stato del processo, informazioni sull'`I/O`, il Program Counter (PC) e la lista delle risorse aperte dal processo. Inoltre il PCB include anche informazioni sulla memoria allocata dal processo (tabella delle pagine, regioni mmapped etc.). Tale operazione e' relativamente costosa.

In definitiva, quindi, la creazione di un thread e' piu' *leggera* rispetto a quella di un nuovo processo.

Nome	Cognome	Matricola

Esercizio 8

Si consideri limplementazione di un FS con Linked List Allocation (LLA) ed un FS che invece utilizzi una Indexed Allocation (IA). Illustrare brevemente i vantaggi dell'uno e dell'altro nell'eseguire le seguenti operazioni:

A. accesso sequenziale

B. accesso indicizzato

C. operazioni su file di testo.

Soluzione

1. **Accesso Sequenziale**: in questo caso, il FS che usa la LLA sara' favorito, garantendo una maggiore velocita' dell'operazione. Cio' poiche' non bisogna effettuare alcuna ricerca per trovare il blocco successivo, poiche' esso sara' semplicemente il blocco `next` nella lista.

2. **Accesso Indicizzato**: questa operazione - contrariamente alla precedente - risulta essere molto onerosa per il FS che usa la LLA. Infatti, per ogni accesso, bisognera' scorrere tutta la lista finche' non viene trovato il blocco desiderato. La ricerca tramite `inode` risultera' molto piu' efficiente.

3. **Accesso su file di testo**: per la natura del tipo di file (testo), la LLA risultera' piu' efficiente ancora una volta. Questo poiche' i file di testo sono memorizzati in maniera sequenziale sul disco, riportandoci al caso 1.