Esame di Sistemi Operativi
AA 2018/19
18 Giugno 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**	**IO Burst**	**Priority**
P1	0	6	5	3
P2	0	5	3	1
P3	3	1	10	0
P4	4	3	1	2

Domanda Si assuma di disporre di uno scheduler preemptive con politica di selezione dei processi basata sulla priorita di un processo - Priority Scheduling 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). Si supponga che i processi **si ripresentino con le stesse specifiche** una volta finito l'I/O.

Soluzione In base a quanto specificato nella consegna, la traccia di esecuzione dei processi sara' quella indicata in Figura 1. Si noti come `P2` "monopolizzi" la CPU a causa della sua lunga durata combinata con una alta priorita', portando `P1` ad essere eseguito molto lentamente.

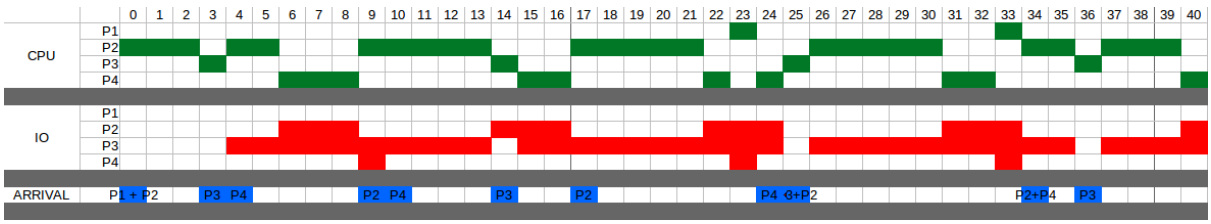


Figure 1: Traccia dei processi usando un algoritmo Priority Scheduling. In verde e in rosso sono indicati rispettivamente i cicli di utilizzo della CPU e dell'I/O; in blu e' scandito il presentarsi di un processo.

Nome	Cognome	Matricola

Esercizio 2

Si consideri in sottosistema di memoria il caratterizzato dalle seguenti tabelle

Number	Base	Limit
0x00	0x000	0x002
0x01	0x002	0x006
0x02	0x008	0x001
0x03	0x009	0x003
0x04	0x00C	0x001
0x05	0x00E	0x001

Table 1: Segmenti

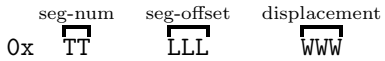
Page	Frame
0x000	0x010
0x001	0x00F
0x002	0x00E
0x003	0x00D
0x004	0x00C
0x005	0x00B
0x006	0x00A
0x007	0x009
0x008	0x008
0x009	0x007
0x00A	0x006
0x00B	0x005
0x00C	0x004
0x00D	0x003
0x00E	0x002
0x00F	0x001
0x010	0x000

Table 2: Pagine

Domanda Assumendo che le pagine abbiano una dimensione di 512 byte, che la tabella delle pagine consista di 512 elementi e che la tabella dei segmenti possa contenere 256 elementi, come vengono tradotti in indirizzi fisici i seguenti indirizzi logici?

- `0x05001FA1`
- `0x01007001`
- `0x00000000`
- `0x01005AAC`
- `0x03F02000`

Soluzione In base alle specifiche del sistema, ogni indirizzo virtuale si potra' scomporre come segue:



I primi due digit piu' significativi individuano il `segment number`. Useremo quindi `base + seg-offset` per individuare il frame all'interno della tabella delle pagine. Ovviamente, se il `seg-offset` eccede il `limit` di quel segmento, si avra' un errore. L'indirizzo finale sara' dato da `frame + displacement`.

Sulla base di quanto detto, avremo i seguenti indirizzi fisici:

- ⊙ `0x05001FA1` → `0x001FA1`
- ⊙ `0x01007001` → invalido
- ⊙ `0x00000000` → `0x010000`
- ⊙ `0x01005AAC` → `0x009AAC`
- ⊙ `0x03F02000` → invalido

Nome	Cognome	Matricola

Esercizio 3

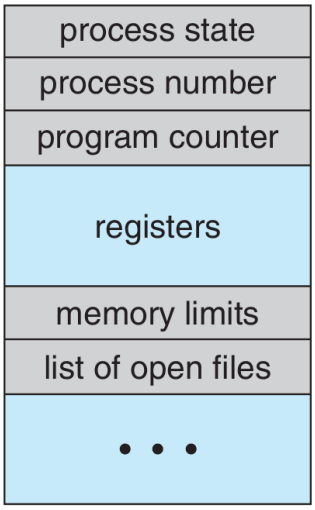
Cosa contiene il Process Control Block (PCB) di un processo? Illustrare inoltre il meccanismo di Context Switch (avvalendosi anche di schemi approssimativi).

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

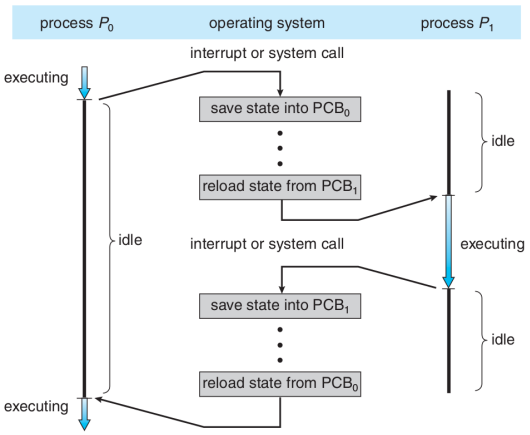
- stato del processo (running, waiting, zombie ...)
- Program Counter (PC), ovvero il registro contente la prossima istruzione da eseguire
- registri della CPU
- informazioni sulla memoria allocata al processo
- informazioni sull'I/O relativo al processo.

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

Si supponga che un OS abbia come disco primario `/dev/sda0` formattato in `ext4` e venga inserito il disco secondario `/dev/sda1` formattato in `NTFS`. Spiegare come viene gestito l'evento dal OS, sottolineando la funzione svolta dal layer del Virtual File System (VFS).

Soluzione Una volta inserito il disco, l'OS - o l'utente - provvedera' ad effettuare una operazione di `mount`. Quindi, l'OS viene informato che un nuovo File System (FS) e' pronto per essere usato. L'operazione, 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).

Il disco secondario `/dev/sda1`, tuttavia, e' formattato con un FS diverso da quello principale. Nonostante cio', l'utente sara' in grado di esplorare il `/dev/sda1` senza notare la differenza (apparentemente). Cio' poiche' l'OS avra' uno strato intermedio costruito *on top* dei vari FS supportati (`ext4`, `NTFS`, `FAT32`, `exFAT`, `APFS`, ...), ovvero il layer del VFS. Esso fungera' da cuscinetto ai vari FS presenti nel sistema, fornendo un accesso uniforme e consistente a tutti gli strati superiori dell'OS che devono operare con tali FS.

Nome	Cognome	Matricola

Esercizio 5

Sia dato il seguente programma:

```
#define NUM_STEPS 5
unsigned int value = 0;
pid_t pid;
pthread_t tid;

void* runner(void* param);

int main(int argc, char *argv[]) {
  pthread_attr_t attr;
  pid = fork();

  if (pid < 0)
    return -1;

  if (pid == 0) {
    pthread_attr_init(&attr);
    pthread_create(&tid,&attr,runner,NULL);
    pthread_join(tid,NULL);
    printf("line A, value = %d\n",value); /* LINE A */
  } else {
    wait(NULL);
    printf("line B, value = %d\n",value); /* LINE B */
  }
  return 0;
}

void* runner(void* param) {
  for (int s = 0; s < NUM_STEPS; ++s) {
    if (pid) {
      value++;
    }
  }
  return param;
}

```

Domanda Cosa stampa il programma in `LINE A` e in `LINE B`?

Soluzione Il programma in traccia crea un processo `child` tramite `fork`, andando a sdoppiare le variabili del processo `parent`. Inoltre il `child` andra' a creare un nuovo thread, che incrementa - eventualmente - la variabile `value`. Tuttavia la funzione eseguita in questo thread - `runner` - scrivera' solo se il `pid != 0`, condizione mai verificata poiche' il thread fa parte del processo `child`.

Date queste considerazioni, l'output del programma sara' il seguente:

```

line A, value = 0
line B, value = 0

```

Nome	Cognome	Matricola

Esercizio 6

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

- frame `16MB`
- memoria fisica indirizzabile `16GB`.

Domande Si calcolino:

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

B. Il tempo di accesso medio alla RAM, considerando che in media per accedere ad una pagina il sistema impiega 150ns, che $T_{TLB} = 10ns$ e che la probabilita' di trovare una pagina all'interno del TLB sia 0.75.

Soluzione

A. Data la dimensione di ogni pagina pari a `16MB`, saranno necessari `24 bit` per indicizzare un elemento all'interno della stessa. La memoria fisica ha dimensione `16GB` e, quindi, necessita di almeno `34 bit`. Il numero di bit *minimo* per indicizzare tutte le pagine e' quindi pari a `34-24 = 10 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 $T_{RAM} = 110\ [ns]$.

Nome	Cognome	Matricola

Esercizio 7

Spiegare brevemente cos'e' il Direct Memory Access (DMA) e quando viene usato.

Soluzione Alcuni device necessitano di trasferire grandi quantita' di dati a frequenze elevate. Effettuare tali trasferimenti tramite il protocollo genericamente usato per altri tipi di periferiche richiederebbe l'intervento della CPU per trasferire un byte alla volta i dati - tramite un processo chiamato Programmed I/O (PIO). Cio' risulterebbe in un overhead ingestibile per l'intera macchina, consumando inutilmente la CPU. Per consentire il corretto funzionamento di tali device evitando gli svantaggi del PIO, tali periferiche possono avvalersi di controllori dedicati che effettuano DMA, cioe' andando a scrivere direttamente sul bus di memoria. La CPU sara' incaricata soltanto di "validare" tale trasferimento e poi sara' di nuovo libera di eseguire altri task.

Questo tipo di periferiche sono molto comuni ai giorni nostri e sono usate nella maggior parte dei dispositivi elettronici - pc, smartphones, servers, ... Esempi di periferica che si avvalgono di controller DMA sono videocamere, dischi, schede video, schede audio, ecc.

Nome	Cognome	Matricola

Esercizio 8

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.