Esame di Sistemi Operativi
AA 2017/18
17 Luglio 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** che devono essere eseguiti in uno *scheduler real-time*

Processo	**Tempo di Inizio**	**CPU burst**	**Periodo**
P1	0	3	12
P2	0	2	10
P3	0	1	5
P4	0	1	3

Domanda Siano date le seguenti assunzioni:

- nessuno dei processi debba attendere il rilascio di una risorsa posseduta da un altro processo;
- la politica di selezione dei processi sia la **Earliest Deadline First** - EDF - e che la deadline di un processo coincida con l'inizio di un nuovo periodo;
- i processi in entrata alla CPU "dichiarano" 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 quindi il comportamento dello scheduler in questione nel periodo indicato, avvalendosi degli schemi di seguito riportati (vedi pagina seguente).

Soluzione In base alle specifiche di sopra enunciate, per prima cosa bisogna prima valutare la fattibilita' dello scheduling tramite questo algoritmo. Per fare cio' calcoliamo la percentuale di utilizzo della CPU totale come segue:

$$U = \sum_j \texttt{CPU}\,(p_j) = \sum_j \frac{t_j}{p_j} = \frac{3}{12} + \frac{2}{10} + \frac{1}{5} + \frac{1}{3} = 0.983 \leq 1 \tag{1}$$

Notiamo che in questo caso $U < 1$, ergo e' possible usare *EDF* come algoritmo di scheduling. La traccia di esecuzione dei processi e' riportata nella Figura 1.

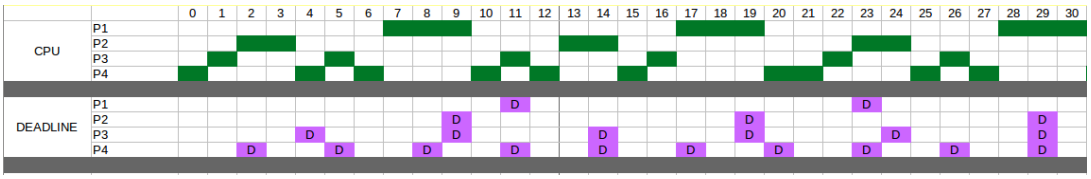


Figure 1: Traccia dei processi schedulati secondo l'algoritmo *Earliest Deadline First*. In verde sono riportati i cicli di CPU, mentre in viola le deadline dei vari processi.

Nome	Cognome	Matricola

Esercizio 2

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

- frame 4KB
- memoria fisica indirizzabile 16GB

Inoltre, ogni indirizzo logico necessita di 48 bit.

Domande

- Quanti bit sono necessari per l'address bus?
- Quanti bit sono necessari per indicizzare una pagina?
- Quanti segmenti ci saranno al massimo in questo sistema?

Soluzione

- L'address bus dovra' essere in grado di indirizzare almeno 16GB di memoria, quindi saranno necessari 34 bit (come minimo).
- $34 - 12 = 22$.
- $48 - 34 = 14$ quindi 2^{14}.

Nome	Cognome	Matricola

Esercizio 3

Dati i seguenti programmi P1 e P2, rispondere alle seguenti domande:

- cosa fanno questi due processi?
- cosa succede se P2 parte prima di P1?

P1

```
#define SHMEM_SIZE 1
#define NUM_ROUNDS 10

char* sem_name_0;
char* sem_name_1;
char* resource_name;

int main(int argc, char** argv) {
  sem_name_0 = "s0";
  sem_name_1 = "s1";
  resource_name = "object";

  sem_t* sem_0 = sem_open(sem_name_0, O_CREAT, 0666, 0);
  sem_t* sem_1 = sem_open(sem_name_1, O_CREAT, 0666, 1);

  int fd=shm_open(resource_name, O_RDWR|O_CREAT, 0666);
  if (fd < 0){
    printf("cannot create shared memory object, error: %s \n", strerror(errno));
    exit(-1);
  }
  int ftruncate_result = ftruncate(fd, SHMEM_SIZE);
  if (ftruncate_result < 0) {
    printf("cannot truncate shared memory object, error: %s \n", strerror(errno));
    exit(-1);
  }

  void * my_memory_area = mmap(NULL, SHMEM_SIZE, PROT_WRITE, MAP_SHARED, fd, 0);
  for (int i=0; i < NUM_ROUNDS; ++i) {
    sem_wait(sem_1);
    char* buffer=(char*) my_memory_area;
    sprintf(buffer, "%d", i);
    printf("[%s]\n", buffer);
    sem_post(sem_0);
    sleep(1);
  }

  int unlink_result=shm_unlink(resource_name);
  if (unlink_result<0) {
    printf("cannot unlink shared memory object, error: %s \n", strerror(errno));
    exit(-1);
  }

  sem_close(sem_0);
  sem_close(sem_1);

  sem_unlink(sem_name_0);
  sem_unlink(sem_name_1);

  return 0;
}
```

P2

```c
sem_t* sem_0 = NULL;
sem_t* sem_1 = NULL;
char* resource_name;

int main(int argc, char** argv) {
  char* sem_name_0 = "s0";
  char* sem_name_1 = "s1";
  resource_name = "object";

  sem_0 = sem_open(sem_name_0, O_CREAT, 0666, 0);
  sem_1 = sem_open(sem_name_1, O_CREAT, 0666, 1);

  int fd = shm_open(resource_name, O_RDONLY, 0666);
  if (fd < 0){
    printf("cannot link to shared memory object, error: %s \n", strerror(errno));
    exit(-1);
  }

  int SHMEM_SIZE = 0;

  struct stat shm_status;
  int fstat_result = fstat(fd, &shm_status);
  if (fstat_result < 0){
    printf("cannot get stats from memory object, error: %s \n", strerror(errno));
    exit(-1);
  }

  SHMEM_SIZE = shm_status.st_size;

  void* my_memory_area = mmap(NULL, SHMEM_SIZE, PROT_READ, MAP_SHARED, fd, 0);

  while (1) {
    sem_wait(sem_0);
    char* buffer = (char*) my_memory_area;
    printf("%s\n", buffer);
    sem_post(sem_1);
    usleep(100000);
  }

  int unlink_result = shm_unlink(resource_name);
  if (unlink_result < 0) {
    printf("cannot unlink shared memory object, error: %s \n", strerror(errno));
    exit(-1);
  }

  sem_close(sem_0);
  sem_close(sem_1);

  return 0;
}
```

Soluzione Analizzando il codice di `P1` e `P2` possiamo affermare quanto segue:

- ⋆ I due processi descrivono un semplice problema *produttore-cconsumatore* . In questo caso, `P1` rappresenta il processo produttore mentre `P2` un generico consumatore. I due processi comunicano tramite una *shared-memory*, mentre la sincronizzazione degli stessi e' ottenuta tramite due semafori POSIX (anche essi ovviamente condivisi tra i processi). E' bene notare che in questo caso, poiche' i semafori sono condivisi, essi avranno un nome univoco associato. Per rimuoverli dal sistema, quindi, oltre ad effettuare una `sem_close`, bisognera' anche eseguire una `sem_unlink` - **una sola volta**.
- ⋆ Se `P2` parte prima di `P1` si ricevera' un errore nel momento in cui si tentera' di linkarsi alla shared memory, poiche' nessuno l'ha ancora creata - solo `P1` la apre con il flag `O_CREAT`.

Nome	Cognome	Matricola

Esercizio 4

Cos 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:

$$\texttt{GAIN} \leq \frac{1}{\mathcal{S} + \frac{1-\mathcal{S}}{N}} \tag{2}$$

E' bene notare che

$$\lim_{N \to \text{inf}} \texttt{GAIN} = \lim_{N \to \text{inf}} \frac{1}{\mathcal{S} + \frac{1-\mathcal{S}}{N}} = \frac{1}{\mathcal{S}} \tag{3}$$

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

Nome	Cognome	Matricola

Esercizio 5

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 $\overbrace{1}^{\texttt{segnum}}\ \underbrace{00}_{\texttt{offset}}\ \overbrace{12}^{\texttt{spiazzamento}}$.

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 6

Si consideri l'implementazione di un file system con allocazione concatenata (Linked Allocation) ed un file system che invece utilizzi una allocazione ad indice (Indexed Allocation).

Domanda Illustrare brevemente i vantaggi dell'uno e dell'altro nell'eseguire le seguenti operazioni:

1. accesso sequenziale
2. accesso indicizzato
3. operazioni su file di testo

Soluzione

1. **Accesso Sequenziale**: in questo caso, il file system che usa la *lista concatenata* sarà favorito, garantendo una maggiore velocità dell'operazione. Ciò poiché non bisogna effettuare alcuna ricerca per trovare il blocco successivo, poiché esso sarà semplicemente il blocco `next` nella lista.
2. **Accesso Indicizzato**: questa operazione - contrariamente alla precedente - risulta essere molto onerosa per il file system che usa la *linked list*. Infatti, per ogni accesso, bisognerà scorrere tutta la lista finchè non viene trovato il blocco desiderato. La ricerca tramite `inode` risulterà molto più efficiente.
3. **Accesso su file di testo**: per la natura del tipo di file (testo), la *linked list* risulterà più efficiente ancora una volta. Questo poiché i file di testo sono memorizzati in maniera sequenziale sul disco, riportandoci al caso 1.

Nome	Cognome	Matricola

Esercizio 7

Illustrare un esempio di mutex implementato mediante la 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 testAndSet(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 (testAndSet(&lock))
    ; /* do nothing */
  /* critical section */
  lock = false;
  /* remainder section */
} while (true);
```

Nome	Cognome	Matricola

Esercizio 8

Spiegare brevemente la differenza tra `open(...)` e `fopen(...)`.

Soluzione `fopen(...)` una funzione di alto livello che ritorna una stream, mentre `open(...)` una syscall di basso livello che ritorna un *file descriptor*. `fopen(...)`, infatti, nasconde una chiamata alla syscall `open(...)`.

Nome	Cognome	Matricola

Esercizio 9

In cosa consiste loperazione di `mount` di un file system?

Soluzione Tramite tale operazione il sistema operativo viene informato che un nuovo file system pronto per essere usato. L'operazione, quindi, provvederà ad associarlo con un dato `mount-point`, ovvero la posizione all'interno della gerarchia del file system del sistema dove il nuovo file system verrà caricato. Prima di efferruare questa operazione di *attach*, ovviamente bisognerà controllare la tipologia e l'integrità del file system. Una volta fatto ciò, il nuovo file system sarà a disposizione del sistema (e dell'utente), come raffigurato nella Figura 2.

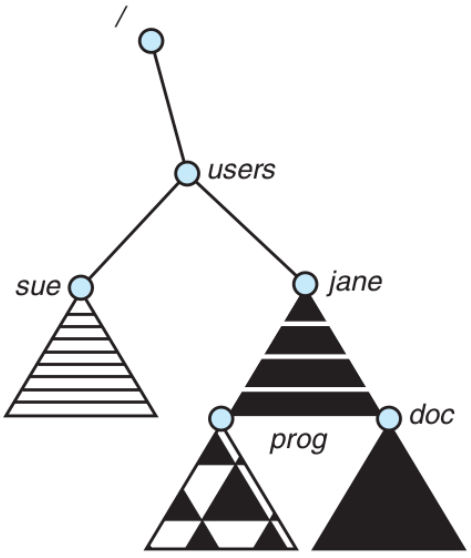


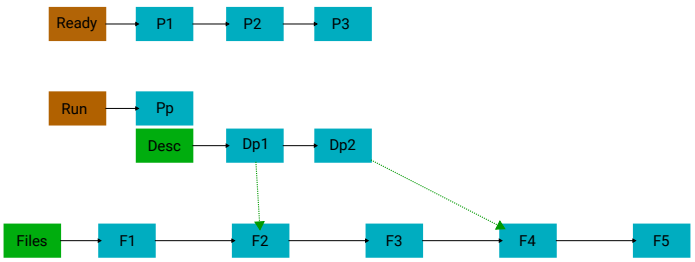
Figure 2: Il file system `jane` viene montato nella directory `/users/jane`

Nome	Cognome	Matricola

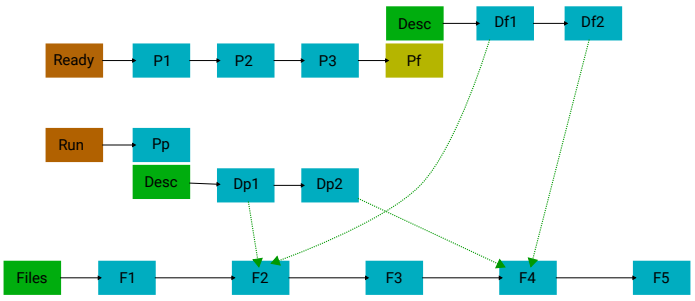
Esercizio 10

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 3: Questa immagine schematizza cio' che avviene durante la chiamata a `fork()`. In Figura 3a, 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 3b. 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 4.

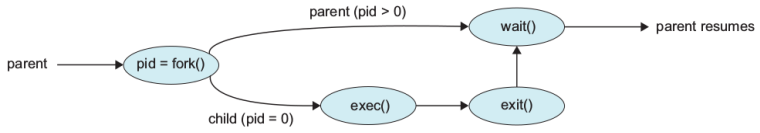


Figure 4: Creazione di un processo tramite syscall `fork()`.