Sistemi Operativi **(M. Cesati)**

Compito scritto del 16 luglio 2014

Nome: _____ **Cognome:** _____

Matricola: _____ **Corso di laurea:** _____

Crediti da conseguire: 5 6 9

Scrivere i dati richiesti in stampatello. Al termine consegnare tutti i fogli. Per conseguire un voto sufficiente è necessario ottenere un voto non gravemente insufficiente in entrambi gli esercizi. Tempo a disposizione: 2,5 ore.

Esercizio 1. Scrivere una applicazione C/POSIX che simula il comportamento del sistema dei Cinque Filosofi a Cena (E. W. Dijkstra, 1965). I cinque filosofi sono seduti a cena ad un tavolo rotondo. Davanti a ciascun filosofo c'è un piatto di spaghetti, e tra un piatto e l'altro c'è una bacchetta (quindi in totale esistono cinque bacchette). Ciascun filosofo pensa per un tempo casuale, poi cerca di prendere le bacchette a destra e sinistra del proprio piatto, mangia per un tempo casuale, lascia le due bacchette, e torna a pensare. Ovviamente se l'una o l'altra delle due bacchette non è disponibile il filosofo dovrà aspettare prima di iniziare a mangiare.

L'applicazione deve fare uso di cinque processi, ciascuno dei quali realizza il comportamento di un filosofo. In ciascuna iterazione ogni processo scrive in standard output l'istante di tempo in cui inizia a pensare, smette di pensare, ed inizia a mangiare. Le "bacchette" condivise tra i processi devono essere realizzate con opportuni meccanismi di sincronizzazione, e si devono evitare le situazioni di stallo dovute a "deadlock".

Nota 1: per misurare il tempo si può utilizzare la API:

```
#include <time.h>
time_t time(time_t *t);
```

In pratica `time(NULL)` restituisce il numero di secondi trascorsi da una data prefissata (1 gennaio 1970).

Nota 2: per ottenere un valore casuale si può utilizzare la API:

```
#include <stdlib.h>
long random(void);
```

In pratica `random()` restituisce un numero pseudo-casuale compreso tra 0 ed il valore `RAND_MAX`.

Esercizio 2. Si descrivano in modo sintetico ed esauriente i motivi per i quali un moderno sistema operativo deve disporre di primitive di sincronizzazione tra flussi di esecuzione, e di come siano realizzate tali primitive.

Sistemi Operativi (M. Cesati)

Esempio del programma del compito scritto del 16 luglio 2014

L'esercizio richiede di simulare il comportamento dei cinque filosofi a cena utilizzando cinque processi. È facile rendersi conto che per realizzare il programma si deve prevedere un meccanismo di sincronizzazione che consenta di tenere traccia dello stato libero/occupato di ciascuna delle cinque bacchette a disposizione dei filosofi. Facciamo la scelta di utilizzare come primitiva di sincronizzazione un set di cinque semafori IPC, un semaforo per ciascuna bacchetta.

L'altro problema da risolvere è quale strategia adottare per evitare le situazioni di stallo (deadlock). Ciascun filosofo deve utilizzare le bacchette poste alla propria destra ed alla propria sinistra. Se assegniamo ai filosofi indici da zero a quattro, ed alle cinque bacchette indici da zero a quattro, potremo avere:

Filosofo	Bacchetta a sinistra	Bacchetta a destra
0	4	0
1	0	1
2	1	2
3	2	3
4	3	4

Se ciascun filosofo prende le bacchette in un ordine relativo prefissato, ad esempio prende sempre prima la bacchetta a sinistra e poi quella a destra, allora è possibile che si verifichino situazione di stallo. D'altra parte, se ciascun filosofo prende sempre prima la bacchetta con indice minore (ossia se rispetta l'ordine globale indotto dagli indici delle bacchette), nessuno stallo si verificherà mai. Pertanto l'ordine con cui i filosofi prenderanno le bacchette dovrà essere il seguente:

Filosofo	Prima bacchetta	Seconda bacchetta
0	0	4
1	0	1
2	1	2
3	2	3
4	3	4

Memorizziamo l'ordine in cui i processi dovranno acquisire i semafori in due vettori costanti:

```
#define num 5
const int first_sem[num]  = { 0, 0, 1, 2, 3 };
const int second_sem[num] = { 4, 1, 2, 3, 4 };
```

Sviluppiamo il programma con un approccio "top-down". La funzione principale del programma dovrà inizializzare le strutture di dati e creare cinque processi per la simulazione di ciascun filosofo:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int main()
{
    int semid;
    initialize(&semid);
    create_phils(semid);
    return EXIT_SUCCESS;
}
```

Il processo principale termina subito dopo aver creato i cinque processi figli.

Il compito più importante della funzione `initialize()` è creare un nuovo set di cinque semafori IPC:

```
time_t base_time;

void initialize(int *psemid)
{
    union semun {
        int val;
        struct semid_ds *buf;
        unsigned short *array;
        struct seminfo *__buf; /* Linux-specific */
    } su;
    int sid, i;

    /* create a new semaphore set with num semaphores */
    sid = semget(IPC_PRIVATE, num, 0666);
    if (sid == -1) {
        perror("semget");
        exit(EXIT_FAILURE);
    }
    *psemid = sid;

    /* initialize the semaphores as free mutexes */
    su.val = 1;
    for (i=0; i<num; ++i) {
        int rc = semctl(sid, i, SETVAL, su);
        if (rc == -1) {
```

```
            perror("semctl");
            exit(EXIT_FAILURE);
        }
    }

    base_time = time(NULL);
}
```

I cinque semafori sono impostati al valore iniziale 1, quindi funzioneranno come mutex. In questo modo viene modellato il fatto che ciascuna bacchetta può essere utilizzata al massimo da un filosofo. In alternativa all'uso ripetuto del comando `SETVAL` avremmo potuto anche utilizzare per una sola volta il comando `SETALL`; quest'ultimo richiede però la definizione di un array con i valori iniziali di tutti i semafori.

La funzione `initialize()` scrive inoltre nella variabile globale `base_time` l'ora corrente (come numero di secondi trascorso dal 1 gennaio 1970). La funzione di servizio `now()` potrà perciò restituire il numero di secondi trascorso dall'inizio dell'esecuzione del programma:

```
time_t now(void)
{
    return time(NULL) - base_time;
}
```

La funzione `create_phils()` si occupa di creare i cinque processi e fare in modo che ciascuno di essi "impersoni" uno dei cinque filosofi:

```
void create_phils(int semid)
{
    int i;
    pid_t pid;

    for (i=0; i<num; i++) {
        pid = fork();
        if (pid == -1) {
            perror("fork");
            exit(EXIT_FAILURE);
        }
        if (pid == 0) {
            play_phil(i,semid,first_sem[i],second_sem[i]);
            /* never returns */
        }
    }
}
```

Il cuore del programma è quindi la funzione `play_phil()`, che riceve quattro argomenti:

1. L'indice (nome) del filosofo
2. Il descrittore del set di semafori IPC
3. L'indice del semaforo che il processo deve acquisire per primo
4. L'indice del semaforo che il processo deve acquisire per secondo

L'implementazione della funzione è molto semplice:

```
void play_phil(int phn, int semid, int first_sem,
               int second_sem)
{
    srandom(base_time+phn);
    for (;;) {
        printf("P%d: start thinking at %ld\n", phn, now());
        random_sleep();
        printf("P%d: stop  thinking at %ld\n", phn, now());
        acquire_sem(semid, first_sem);
        acquire_sem(semid, second_sem);
        printf("P%d: start eating   at %ld\n", phn, now());
        random_sleep();
        release_sem(semid, second_sem);
        release_sem(semid, first_sem);
    }
}
```

L'invocazione di `srandom()` serve ad impostare un seme del generatore pseudo-casuale diverso da processo a processo, così che i cinque filosofi esibiscano un comportamento differente l'uno dall'altro.

La funzione `random_sleep()` sospende il processo per un tempo casuale:

```
void random_sleep(void)
{
    const int max_sleep = 20; /* seconds */
    int delay = random() % max_sleep + 1;
    while (delay != 0)
        delay = sleep(delay);
}
```

Si noti che la funzione di libreria `sleep()` può restituire zero se l'attesa è finita, oppure un valore positivo se l'attesa è stata interrotta dalla ricezione di un segnale. In quest'ultimo caso il ciclo `while` consente di riprendere l'attesa interrotta.

Per acquisire e rilasciare un semaforo si utilizzano due macro che, in realtà, vengono tradotte nella invocazione di una medesima funzione:

```
#define acquire_sem(s,i) operate_sem((s), (i), -1)
#define release_sem(s,i) operate_sem((s), (i), +1)

void operate_sem(int semid, int idx, int op)
{
    struct sembuf sb;

    sb.sem_num = idx;
    sb.sem_op =  op;
    sb.sem_flg = 0;

    if (semop(semid, &sb, 1) == -1) {
        perror("semop");
        exit(EXIT_FAILURE);
    }
}
```