Esame 14/7/2023

Esercizio 1

Una chiamata di sistema viene eseguita da un processo per richiedere servizi dal sistema operativo. Le chiamate di sistema vengono implementate tramite trap; una trap è un interrupt software provocato dall'esecuzione di una determinata istruzione del programma in esecuzione. Per effetto della trap, la CPU passa da user mode a kernel mode, dove il trap handler gestisce la chiamata. Completata l'operazione, il controllo viene ritornato al processo passando in user mode.

Per quanto riguarda le gestione dei processi, il sistema operativo consuma tempo per svolgere i context switch (passaggio di esecuzione da un processo ad un altro), che quindi non devono essere troppo frequenti, e consuma spazio nella gestione della tabella dei processi e delle pagine, che non deve essere quindi allocata per tutto lo spazio di indirizzi poteziale del processo. Per quanto riguarda la gestione della mamoria, il SO consuma tempo nella traduzione degli indirizzi, specialmente con la paginazione, che viene velocizzata con il TLB; nella paginazione su richiesta, in quanto pagine non presenti in memoria (page fault) vanno caricate, e il tempo d'accesso medio alla memoria rischia di salire molto se i page fault sono frequenti (vanno ottimizzati quindi tramite algoritmi di rimpiazzamento ed eventualmente facendo swap out dei processi per evitare thrashing).

Esercizio 2

Test and Set Lock è un'istruzione di livello ISA (TSL RX,LOCK). Legge i contenuti della parola LOCK nel registro RX e memorizza un valore non nullo all'indirizzo di memoria LOCK. L'operazione di lettura e memorizzazione sono indivisibili, poichè la CPU blocca il bus di memoria. Quando LOCK è 0, qualsiasi processo può impostarlo ad 1 usando l'istruzione TSL e procendendo con la lettura o scrittura della memoria condivisa; alla fine dell'operazione, il processo imposta LOCK a 0.

```
enter_region:
    TSL REGISTER,LOCK   //copia LOCK in registro e impostalo a 1
    CMP REGISTER,#0     //controlla se lock = 0
    JNE enter_region    //se != 0, loop
    RET                 //ritorna al chiamante, entra in regione critica

leave_region:
    MOVE LOCK,#0        //memorizza 0 nel LOCK
    RET                 //return
```

In linguaggio di alto livello:

```
Inizializzazione:
lock = 0;

Per entrare in sezione critica:
while(TestAndSet(&lock));

In uscita dalla sezione critica:
lock = 0;
```

Questa soluzione non soddisfa il requisito di attesa limitata.

Esercizio 3

Esercizio 4

First-fit: 3, 1, 3
Best-fit: 7, 1, 6
Worst-fit: 3, 4, 8
Dato che le pagine che si trovano in memoria sono mantenute in una lista ordinata in base all'istante di caricamento in RAM, al momento di rimpiazzamento viene scartata la pagina in testa alla lista, indipendentemente da quando sia stata riferita l'ultima volta. Per ovviare a questo problema si usa l'algoritmo della seconda possibilità, che utilizza il bit R: se il suo R è 1, la pagina non viene scartata, ma viene portata in fondo alla lista (come se fosse appena stata caricata un memoria) e R viene azzerato.

Esercizio 5

```
int main(int arg, char **argv) {
    char line[64];
    pid_t pid;
    int status;
    FILE *fp = fopen(argv[1], "w");
    char *n, *s;

    printf("> ");
    while(fgets(line, sizeof line, stdin) != NULL) {
        n = strtok(line, " ");
        s = strtok(NULL, " ");
        for(int i = 0; i < atoi(n); i++) {
            if(fork() == 0) {
                execlp(s, s, (char*)0);
                exit(0);
            }
        }
        printf("> ");
    }

    fclose(fp);
    exit(0);
}
```

Si può aggiungere error-checking per fork (errore con fork == -1) e execlp (errore con execlp $<$ 0).

Esercizio 6

Il valore inizializzato da init può essere modificato solo tramite procedure di up e down:

- down: se il valore del semaforo è maggiore di 0, il valore viene decrementato, altrimenti il processo/thread che esegue l'operazione deve attendere
- up: se vi sono altri processi/thread in attesa per effetto di una down, uno di questi termina l'attesa e conclude l'esecuzione della down, altrimenti il valore del semaforo viene incrementato

Controllare il valore, cambiarlo ed eventualmente sospendersi sono operazioni atomiche al fine di risolvere i problemi di sincronizzazione e le race condition.

```
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <pthread.h>

sem_t mutex;

void *thr_fn1(void *arg) {
    printf("Part A of thread 1\n");
    sem_post(&mutex);
    printf("Part B of thread 1\n");
    return ((void*)0);
}

void *thr_fn2(void *arg) {
    printf("Part A of thread 2\n");
    sem_wait(&mutex);
    printf("Part B of thread 2 should be last!\n");
    return ((void*)0);
}

int main(void) {
    pthread_t t1, t2;

    sem_init(&mutex, 0, 0);

    pthread_create(&t1, NULL, thr_fn1, NULL);
    pthread_create(&t2, NULL, thr_fn2, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    exit(0);
}
```