Esame 28/2/2023

Esercizio 1

New → Ready: il thread viene creato, gli vengono allocate le risorse per gestirlo affinchè il processo passi allo stato Ready pronto per essere gestito.
Ready → Running: il thread viene selezionato dallo scheduler per essere eseguito.
Running → Ready: il thread non ha ancora terminato la sua esecuzione, ma ha ricevuto un timer-interrupt e ritorna allo stato Ready dove può essere rischedulato.
Running → Waiting: il thread esegue un'operazione sospensiva, interrompe la sua esecuzione rilasciando la CPU e si mette in attesa del completamento della richiesta.
Waiting → Ready: la richiesta sospensiva è terminata, ritorna allo stato Ready.
Ready, Running, Waiting → Terminated: il thread riceve un segnale di interruzione, terminando in modo anomalo, oppure potrebbe eseguire un'operazione non valida, oppure terminare in modo corretto dopo aver completato tutte le operazioni.

Il Process Control Block è una struttura mantenuta dal sistema operativo, per ogni processo, che mantiene le informazioni sullo stato corrente del processo.
Informazioni per processo: spazio indirizzi, variabili globali, file aperti, processi figlio, allarmi in attesa, segnali
Informazioni per thread: PC, registri, stack, stato

Il context switch è il passaggio dell'esecuzione di un processo/thread corrente all'esecuzione di un altro processo/thread. Durante un context switch, il dispatcher salva lo stato del processo attuale e carica quello del processo in attesa di esecuzione. Il context switch tra processi è un'operazione più dispendiosa in termini di tempo in quanto si devono caricare molte più informazioni rispetto ai thread (solo stack, registri, stato e PC).

Esercizio 2

Con la tecnica della paginazione, ad ogni processo è assegnato un proprio spazio indirizzi virtuale, contiguo e suddiviso in parti di dimensione fissa chiamate pagine logiche. Nella memoria fisica è disponibile un certo spazio di indirizzi fisici, suddivisa in porzioni dette frame, su cui vengono copiati, quando necessario, le pagine logiche. Non tutte le pagine di un processo devono essere presenti in memoria contemporaneamente in quanto le pagine vengono caricate in memoria a runtime in base alle richieste del processo. Per ogni processo viene mantenuta una tabella delle pagine, che indica per ciascuna pagina dello spazio degli indirizzi la presenza o meno in RAM, e in quale frame se presente. La Memory Management Unit (MMU) è un'unità hardware che traduce indirizzi virtuali a fisici. Se un processo prova a far riferimento ad una pagina non presente in memoria, la MMU causa una trap di tipo page fault.

Per indirizzare i byte all'interno di una pagina servono 12 bit, in quanto 4 Kbyte = 2^{12}. Negli indirizzi logici e fisici i 12 bit meno significativi rappresentano l'offset all'interno della pagina/frame, mentre i restanti bit indicizzano la specifica pagina o frame. Negli indirizzi logici, 36 - 12 = 24 bit servono per specificare la pagina logica; negli indirizzi fisici 30 - 12 = 18 bit servono per identificare il frame. L'occupazione di ciascuna entry nella tabella delle pagine è di 4 byte: 3 byte per memorizzare l'indice del frame e un byte aggiuntivo per i bit d'utilità. Il numero totale di entry

è uguale al numero della pagine logiche, ossia 2^{24}. La page table completa occupa quindi $4 * 2^{24}$ byte, circa 64 MB.

Esercizio 5

Thread:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <unistd.h>

typedef struct alarm {
    int seconds;
    char message[64];
} alarm_t;

void *tbody(void *arg){
    alarm_t *alarm;
    alarm = (alarm_t*) arg;
    sleep(alarm->seconds);
    printf("Messaggio: %s\n", alarm->message);
}
int main(int argc, char *argv[]){
    char line[128];
    alarm_t *alarm;
    pthread_t t;
    while(1){
        printf("Allarme >");
        if(fgets(line, sizeof(line), stdin)==NULL) exit(0);
        if(strlen(line)<=1) continue;
        alarm=(alarm_t *) malloc(sizeof(alarm_t));
        if(sscanf(line, "%d %64[^\n]", &(alarm->seconds), &(alarm->message)
)<2) {
            fprintf(stderr, "Comando sconosciuto\n");
        } else {
            pthread_create(&t, NULL, tbody, (void *)alarm);
        }
    }
}
```

Processi:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <unistd.h>

int main(int argc, char **argv) {
    char line[128];
    int seconds;
    char message[64];
    pid_t pid;

    while(1) {
```

```
            printf("Allarme > ");
            if(fgets(line, sizeof line, stdin) == NULL) {
                exit(0);
            }
            if(strlen(line) <= 1) {
                continue;
            }
            if(sscanf(line, "%d %64[^\n]", &seconds, message) < 2) {
                printf("Comando sconosciuto\n");
            }
            else {
                pid = fork();
                if(pid == -1) {
                    printf("Errore fork");
                    exit(1);
                }
                if(pid == 0) {
                    sleep(seconds);
                    printf("Messaggio: %s", message);
                    exit(0);
                }
            }
        }
    }
```

Esercizio 6

Thread:

```
    int n = N;
    arrive = 0;
    sem_t mutex, sem;

    void wall() {
        sem_wait(&mutex);
        arrive++;
        if(arrive < n) {
            sem_post(&mutex);
            sem_wait(&sem);
        }
        else {
            for(int i = 0; i < n; i++) {
                sem_post(&sem);
            }
        }
    }

    int main(int argc, char **argv) {
        pthread_t t[n];

        for(int i = 0; i < n; i++) {
            pthread_create(&t[i], NULL, wall, NULL);
        }
        for(int i = 0; i < n; i++) {
            pthread_join(t[i], NULL);
        }
```

```
}
```

Processi:

```
int n = N;
arrive = 0;
sem_t mutex, sem;

void wall() {
    sem_wait(&mutex);
    arrive++;
    if(arrive < n) {
        sem_post(&mutex);
        sem_wait(&sem);
    }
    else {
        for(int i = 0; i < n; i++) {
            sem_post(&sem);
        }
    }
}

int main(int argc, char **argv) {
    pid_t pid;

    for(int i = 0; i < n; i++) {
        pid = fork();
        if(pid == 0) {
            printf("Sono il processo %ld e sto arrivando alla barriera...\n
", getpid());
            wall(n);
        }
    }
    for(int i = 0; i < n; i++) {
        wait(NULL);
    }
}
```