Sincronização com semáforos

Carlos Alberto Santos de Souza
201110006250
Sistemas Operacionais 2016-1

1. Sincronização de threads

- Threads são escalonadas de formas diferentes a cada execução.
- Não é possível saber quando uma thread será interrompida para que outra seja executada.
- Para sincronizar uma thread precisamos de ferramentas como mutex, semaphores e variáveis condicionais.

3. Conceito de Semáforos

- É uma ferramenta que sinaliza para a thread parar o seu processamento até que esta seja sinalizada para seguir em frente (como um semáforo de trânsito).
- Os sinais podem ser emitidos de forma sincrona (pelo ***sem_wait()***) ou assíncrona (pelo ***sem_trywait()***).

2. Como funciona?

- Eles controlam o fluxo de processamento das threads através das funções ***sem_wait()**** e ***sem_post()**** que, respectivamente, pausam ou libera o processamento de uma thread.
- Os semáforos usam um contador que é decrementado por cada ***sem_wait()*** que é lançado, se o valor do contador for 0, a thread fica no estado de *esperando* até o valor voltar a ser positivo. O ***sem_post()*** é quem incrementa o contador.

- ***** estas funções são apenas as mais comuns.

5. Funções importantes dos semáforos

- int sem_destroy(sem_t *);
- int sem_init(sem_t *, int, unsigned int);
- int sem_post(sem_t *);
- int sem_trywait(sem_t *);
- int sem_wait(sem_t *);

- Obs*:* ***sem_t*** é o tipo de variável criada para o contador do semáforo.

4. Bibliotecas relevantes para a implementação

- <semaphore.h> -> usada para a criar e manipular os semaforos.
- <pthread.h> -> biblioteca de threads implementadas de acordo com o padrão POSIX.
- <malloc.h> -> Usada para requisitar uma porção de memória e retornar um ponteiro pra ela.

6. Implementação

Código

```c
#include <pthread.h>
#include <stdio.h>
#include <malloc.h>
#include <semaphore.h>

/*Aqui criamos a estrutura para a tarefa*/
struct job{
    struct job* next;
    int time;
};

/*Fila de tarefas*/
struct job* job_queue;

/*Semaforo*/
sem_t semaforo;
/*Mutex*/
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
/*Contador de tarefas*/
int cont = 0;
/*Contador de segundos*/
int sec = 0;

/*inicializamos a fila e o semaforo*/
void initialize_job_queue () {
  job_queue = NULL;
   /*Iniciando o Semaforo*/
   sem_init(&semaforo,0,0);
}
```

```c
/*Responsável por enfileirar uma nova tarefa*/
void create_job(int time){
    struct job* new_job = (struct job*) malloc (sizeof (struct job));
    new_job->time = time;

   pthread_mutex_lock (&mutex);
    new_job->next = job_queue;
    job_queue = new_job;
    cont++;
    printf("Tarefas enfileiradas [%d] \n", cont);
   pthread_mutex_unlock (&mutex);

  /*Indica que já existem tarefas disponíveis para serem executadas
    as threads que estiverem esperando por alguma tarefa pode voltar a
    ativa.*/
    sem_post(&semaforo);
}

/*Simula o tempo de execução de uma tarefa*/
void  process_job(struct job* job){
    int id = pthread_self();
      printf("Tarefa sendo executada por PID: %d\n", id);
      sleep(job->time);
      printf("Tarefa Finalizada por PID: %d\n", id);
}
```

6. Implementação

```c
/*Retira uma tarefa da fila*/
void  get_job(){
  pthread_mutex_lock (&mutex);
    cont--;
    printf("Tarefas enfileiradas [%d] \n", cont);
    struct job* job_next = job_queue;
    job_queue = job_queue->next;
  pthread_mutex_unlock (&mutex);

    process_job(job_next);
    free (job_next);
}

/*Função executada pelas threas (simula as requisições de dispositivos)*/
void*  thread(){
  int id = pthread_self();
  printf("Thread iniciada. PID: %d\n", id);
  while(1){
    printf("Thread Esperando. PID: %d\n", id);
    sem_wait (&semaforo);
    int id = pthread_self();
    printf("Tarefa retirada por PID: %d\n", id);
    get_job();
  }
  return NULL;
}
```

```c
/*Função da thread escalonadora, mantém a fila de tarefas*/
void*  thread_scheduler(){
  printf("Programa iniciado.\n");
  int id = pthread_self();
  printf("Scheduler iniciado. PID: %d\n", id );
  int i;
  for(i = 0;i <= 5; i++){
    create_job(1);
    sleep(5+i);
  }
  return NULL;
}

/*Um pequeno pseoudo relógio para compararmos o uso das threads*/
void* time_mensure(){
  while(1){
    sec++;
    printf("sec %d\n",sec);
    sleep(1);
  }
}
```

6. Implementação

função main

```
int main(){
   initialize_job_queue();

   pthread_t thread_id1;
   pthread_t thread_id2;
   pthread_t thread_id3;
   pthread_t thread_id4;

   pthread_create (&thread_id1, NULL, &thread_scheduler, NULL);
   pthread_create (&thread_id2, NULL, &thread, NULL);
   pthread_create (&thread_id3, NULL, &thread, NULL);
   pthread_create (&thread_id3, NULL, &time_mensure, NULL);

   pthread_join (thread_id1, NULL);

   /*Destrói o semáforo*/
   sem_destroy(&semaforo);
  return 0;
}
```

```
sec 1
Thread iniciada. PID: -179529984
Thread Esperando. PID: -179529984
Thread iniciada. PID: -171137280
Thread Esperando. PID: -171137280
Programa iniciado.
Scheduler iniciado. PID: -162744576
Tarefas enfileiradas [1]
Tarefa retirada por PID: -179529984
Tarefas enfileiradas [0]
Tarefa sendo executada por PID: -179529984
sec 2
Tarefa Finalizada por PID: -179529984
Thread Esperando. PID: -179529984
sec 3
sec 4
sec 5
Tarefas enfileiradas [1]
Tarefa retirada por PID: -171137280
Tarefas enfileiradas [0]
Tarefa sendo executada por PID: -171137280
sec 6
```

6. Referências

- **Advanced Linux Programming.** Disponível em: http://advancedlinuxprogramming.com/alp-folder/. Acesso em: 16 ago. 2016.

- **The Single UNIX .** Disponível em: http://pubs.opengroup.org/onlinepubs/7908799/xns/listen.html. Acesso em: 18 ago. 2016.

Gist (gitHub)

- https://github.com/CarlosAlbertoUFS/SOProjects/tree/master/Trabalho04