1 Programação concorrente no Linux

1.1 Threads

Como apresentado em aula, a linguagem C não possui nenhum recurso nativo para permitir a programação concorrente. Contudo, este problema pode ser contornado utilizando-se uma bibliteca que provê a abstração de **pthreads**. Uma pthread seria uma nova linha de execução, ou seja, uma função que será executada concorrentemente com as outras pthreads/processos que existirem no sistema operacional.

Antes de utilizar threads, devemos fazer a inclusão do arquivo onde existem as definições das funções

`#include <pthread.h>`

1.2 Compilação de programas concorrentes

A biblioteca pthreads precisa ser inserida ao compilar o programa fonte **.c** usando a opção `-l` (library)

`gcc meu_programa.c -lpthreads -o meu_programa`

1.2.1 Criação das threads

A criação de uma thread necesseita de um identificador: `pthread_t id`. Todas as operações serão realizadas sobre este identificador.

A função para criar uma thread é:

```
1 int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)
      (void *), void *arg);
```

O primeiro parâmetro é a identificação da thread, o segundo são os atributos que podem ser deixados como NULL (atributos default), o próximo parâmetro é o endereço da função alvo que será executada e o último parâmetro são os parâmetros desta função alvo. Percebe-se que a função alvo deve ser uma função que retorna um ponteiro genérico (void *) e tem como parâmetro um ponteiro genérico também.

Exemplo:

`pthread_create (&id, NULL, (void *) minha_thread,NULL);`

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4  #include <unistd.h>
5  #include <math.h>
6
7  pthread_t id,id2;
8
9  void * minha_thread (void *apelido) {
10    float k;
11    while(1) {
12       k=sin(k+123);
13    }
14    pthread_exit(NULL);
15 }
16
17 void * minha_thread_2(void *apelido) {
18    while(1)  {
19       printf("2\n");
20       sleep(4);
21    }
22    pthread_exit(NULL);
23 }
24
25 int main(int argc, char *argv[]) {
26     pthread_create (&id,  NULL, (void *) minha_thread,  NULL);
27     pthread_create (&id2, NULL, (void *) minha_thread_2,NULL);
```

```
28      while(1);
29      return 0;
30  }
```

1.2.2 Terminação de uma thread

Uma thread pode ser terminada com a função `pthread_exit(NULL);` o parâmetro desta função será o valor de retorno da thread. Se a thread em questão (como no exemplo) fica num loop sem fim, ela jamais terminará e assim não precisa chamar `pthread_exit`.

Note ainda que quando as 2 threads foram disparadas na função `main` existem 3 linhas de execução. Uma para cada thread e outra da própria função `main`. No exemplo, a função `main` fica num loop sem fim. Se aquele loop não existisse, a função `main` chamaria `return` e terminaria o processo (matando todas as threads que estivessem executando).

A função `main` pode ser codificada para ao invés de ficar num loop sem fim, aguardar o término de uma thread usando a função:

`int pthread_join(pthread_t th, void **thread_return);`

O primeiro parâmetro é a identificação da thread e o segundo parâmetro é o retorno que a tal thread fornece. Quando não existe interesse em verificar o retorno pode-se chamar `pthread_join( thread1, NULL);`

1.2.3 Passagem de parâmetros para threads

```
1
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <pthread.h>
5  #include <unistd.h>
6
7  #define NUM_THREADS     5
8
9  void *PrintHello(void *threadid){
10
11 int tid;
12
13     usleep(10000);
14     tid = (int)threadid;
15     printf("Ola, eu sou a Thread %d!\n", tid);
16     pthread_exit(NULL);
17 }
18
19 int main (int argc, char *argv[]){
20
21     pthread_t threads[NUM_THREADS];
22     int rc, t;
23
24     for(t=0; t<NUM_THREADS; t++){
25         printf("Criando a thread numero %d\n", t);
26         rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
27         if (rc){
28             printf("Erro: Problema na criacao da thread %d\n", rc);
29             exit(-1);
30         }
31     }
32     while(1);
33     return 0;
34 }
```

Discussão sobre parâmetros para threads:

- Se o parâmetro for um double ?
- Se for necessário passar mais parâmetros ?

1.3 Exclusão mútua e sincronização de threads: Semáforos

Para permitir a exclusão mútua e sincronização de threads num programa concorrente pode-se utilizar semáforos. Da mesma forma que threads, semáforos não existem nativamente na linguagem C mas foram implementados como uma biblioteca externa.

Para utilizar semáforos no Linux utiliza-se `#include <semaphore.h>`

1.3.1 Criação e inicialização de um semáforo

Nas aulas teóricas de Sistemas Operacionais foi apresentada uma instrução para inicialização de semáforos. Naquela momento por se tratar de um pseudo-código os semáforos foram inicializados diretamente, exemplo: `semaphore mutex=1`.

Na pratica para criar um semáforo, deve-se declarar: `sem_t mutex;` e para inicializar o semáforo `sem_init (&mutex,0,1);`. Nesta chamada o semáforo **mutex** está sendo inicializado com o valor 1. O segundo parâmetro que vale 0 serve para indicar que o semáforo criado será compartilhado com todas as threads que pertencem a um determinado processo e assim o semáforo será alocado numa memória pertencente ao processo. Se o valor for diferente de 0 significa que o semáforo será compartilhado com outros processos no sistema e precisará ser alocado numa memória compartilhada por estes processos.

1.3.2 Compilação de programas com semáforos

A biblioteca de semáforos está junto com a biblioteca das pthreads, assim, usa-se `gcc main.c -lpthreads -o main`

1.3.3 Destruição de um semáforo

Caso um semáforo não esteja mais em uso o mesmo pode ser destruído com `int sem_destroy(sem_t *sem);` Somente semáforos que tenham sido anteriormente inicializados podem ser destuídos pois é na operação de inicialização que a memória necessária para o semáforo é alocada. A rotina de destruição irá desalocar esta memória.

1.3.4 Operações sobre um semáforo

`int sem_wait(sem_t * sem);` Utilizada para fazer um `P/down` sobre um semáforo, ou seja, decrementa seu valor e bloqueia se o valor estava em zero.

`int sem_post(sem_t * sem);` Utilizada para fazer um `V/up` sobre um semáforo, ou seja, incrementa o seu valor. Se o semáforo valia 0 e algum processo/thread estava bloqueado neste semáforo então será liberado e poderá executar quando selecionado pelo escalonador.

`int sem_getvalue(sem_t *sem, int *sval);` Utilizada para capturar (ler) o valor de um semáforo sem realizar nenhuma operação sobre ele. O valor lido é armazenado no segundo parâmetro.

```
1
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <pthread.h>
5  #include <unistd.h>
6  #include <semaphore.h>
7
8  sem_t S;
9  int vet[2];
10
11 void * th1 (void *s)
12 {
13     sem_wait (&S);
14     vet[0]=0;
```

```
15      usleep(1000);
16      if (vet[0]==0) {
17          printf("esta valendo 0");
18      }
19      else printf("nao esta valendo 0");
20      sem_post (&S);
21      pthread_exit(NULL);
22  }
23  void * th2 (void *s)
24  {
25      sem_wait(&S);
26      vet[0]=1;
27      sem_post(&S);
28  }
29
30  int main(int argc, char *argv[])
31  {
32      pthread_t id1, id2;
33      vet[0]=0;
34
35      sem_init (&S,0,1);
36      pthread_create (&id1,NULL,th1,NULL);
37      pthread_create (&id2,NULL,th2,NULL);
38      while(1);
39      return 0;
40  }
```

1.4 Exercícios:

1) Implemente o problema do produtor consumidor usando programação concorrente e semáforos.
2) Implemente o problema dos leitores e escritores usando programação concorrente e semáforos.
3) Implemente o problema do jantar dos filósofos usando programação concorrente e semáforos.