Universidade Federal de Santa Catarina

INE 5611-04238 B - Sistemas Operacionais

Trabalho 1

José Ribamar Marçal Martins Junior (15101188)
Luiz Andre Ventura Gosling (16203896)
Willian Bernardo Silva (17200721)

Florianópolis - SC
2021

Objetivo do trabalho

Explorar aspectos de concorrência com programação multi-threaded usando POSIX threads:

- Identificar condições de corrida e implementar regiões críticas com o uso de estruturas de sincronização (mutexes, semáforos, etc.)

Descrição do problema

Suponha um projeto de SO hipotético que divide o espaço de endereçamento do sistema em blocos de memória. Os blocos de memória possuem tamanho *k bytes* (com k configurável). O SO deve implementar as chamadas de sistema:

Escreve(Posicao1, &buffer, tam_buffer)
Le(Posicao4,tamanho)

O sistema admite execução concorrente das chamadas 'Escreve' e 'Le', mas ele deve garantir que as operações sejam atômicas, ou seja, regiões de memória acessadas por uma operação não devem ser modificadas por outra operação durante a sua execução. Além disso é importante observar que:

1) A quantidade total de memória é dada por N (número de blocos) x k (tamanho de cada bloco).
2) Operações sobre blocos independentes podem ocorrer em paralelo.
3) Operações que acessem mais de um bloco devem garantir que outras operações não atualizem os blocos acessados enquanto a operação estiver em curso.
4) Operações de leitura podem acessar um mesmo bloco em paralelo (este é um requisito opcional).

Serviço de backup

Um serviço de log deve salvar cópias do estado da memória global em disco:

- Uma thread é executada exclusivamente para este propósito.
- É importante que durante o registro dos blocos de memória em arquivo nenhuma atualização seja feita nos blocos de memória, garantindo que o salvamento do estado seja consistente.

Código implementado e comentado:

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#include <malloc.h>
#define NUMLIDO 1
#define NUMESCRITO 2
#define BUFFERSIZE 4
#define COUNT_MAX 4
pthread_t lido[NUMESCRITO];
pthread_t escrito[NUMLIDO];
pthread_mutex_t buffer_mutex;
int buffer[BUFFERSIZE];
int currentidx;
sem_t buffer_full, buffer_empty;
static pthread_key_t thread_log_key;
int count;
```

No bloco acima está sendo feita a declaração das inclusões de biblioteca e as constantes que serão utilizadas no programa. Também são declarados o mutex (`pthread_mutex_t`), semáforo (`sem_t`) que gerenciará o buffer nos estados *full* e *empty*, as variáveis de armazenamento do log (`pthread_key_t thread_log_key`) e as threads que serão criadas estão recebendo o valor das constantes anteriormente definidas.

```
void *escritor(void *arg) {
   int n;
   while(count < COUNT_MAX) {
        n = rand()%1000;
        sem_wait(&buffer_full);
        pthread_mutex_lock(&buffer_mutex);
        count = count +1;
        buffer[currentidx++] = n;
        pthread_mutex_unlock(&buffer_mutex);
        sem_post(&buffer_empty);
        printf("Thread 1 escrevendo numero %d\n", n);
```

```
        sleep((1 + rand()%5));
    }
    pthread_exit(NULL);
}
```

Acima está sendo declarada a função que a thread 1 utilizará para escrever na memória.

```
void *leitor(void *arg) {
    int n;
    while(count < COUNT_MAX) {
        sem_wait(&buffer_empty);
        pthread_mutex_lock(&buffer_mutex);
        count = count +1;

        n = buffer[--currentidx];
        pthread_mutex_unlock(&buffer_mutex);
        sem_post(&buffer_full);
        printf("Thread 2 lendo numero %d\n", n);
        sleep(1 + (rand()%5));
    }
    pthread_exit(NULL);

}
```

Acima está sendo declarada a função que a thread 2 utilizará para ler da memória.

```
void write_to_thread_log (const char* message)
{
 FILE* thread_log = (FILE*) pthread_getspecific (thread_log_key);
 fprintf (thread_log, "%s\n", message);
}

void close_thread_log (void* thread_log)
{
 fclose ((FILE*) thread_log);
}
void* thread_function (void* args)
```

```
{
 char thread_log_filename[20];
 FILE* thread_log;

 /* Generate the filename for this thread.s log file. */
 sprintf (thread_log_filename, "thread%d.log", (int) pthread_self ());


 thread_log = fopen (thread_log_filename, "w");

 /* Store the file pointer in thread-specific data under
thread_log_key. */
 pthread_setspecific (thread_log_key, thread_log);
 write_to_thread_log ("Thread starting.");
}
```

Acima estão sendo declarados 3 métodos que serão utilizados pela thread 3 para escrita do log. A thread criará um log para cada escrita e lida realizada pelas outras threads do programa.

```
int main(int argc, char **argv) {
   int i;
   pthread_mutex_init(&buffer_mutex, NULL);
   sem_init(&buffer_full, 0, BUFFERSIZE);
   sem_init(&buffer_empty, 0, 0);
   pthread_t threads[i];
```

Acima o mutex e o semáforo são iniciados e o array de thread `pthread_t` é declarado.

```
   for(i=0; i<NUMLIDO; i++)
       pthread_create(&(lido[i]), NULL, escritor, NULL);
```

O laço acima popula o array da thread referente aos números escritos.

```
   for(i=0; i<NUMESCRITO; i++)
       pthread_create(&(escrito[i]), NULL, leitor, NULL);
```

O laço acima popula o array da thread referente aos números lidos.

```
   for(i=0; i<NUMLIDO; i++)
       pthread_join(lido[i], NULL);

   for(i=0; i<NUMESCRITO; i++)
       pthread_join(escrito[i], NULL);
```

A função `pthread_join` utilizada em ambos os laços acima suspende a execução de uma thread até que a outra termine sua execução.

```
       /* Create a key to associate thread log file pointers in
  * thread-specific data. Use close_thread_log to clean up the file
  * pointers. */
 pthread_key_create (&thread_log_key, close_thread_log);

 /* Create threads . */
 for (i = 0; i < COUNT_MAX; ++i)
   pthread_create (&(threads[i]), NULL, thread_function, NULL);


 for (i = 0; i < COUNT_MAX; ++i)
   pthread_join (threads[i], NULL);
 return 0;


}
```

Os laços acima incrementam as contagens feitas nos métodos de escrita e leitura para que as threads possam ser finalizadas em determinado momento.