Algoritmos de SO – Threads e Semaphores

Banquete do Rei

O objetivo deste projeto é escrever um programa em C para simular a execução de um conjunto de tarefas em paralelo usando uma thread Posix ou Win32 para modelar cada convidado como uma thread um semáforo para o tacho de **Paella Espanhola**

. Sugiro, ser no ambiente Linux, mas pode ser feito no Windows

Data entrega 20/abr/2018, com defesa na aula de laboratório seguinte.

Pode ser feito por até 3 componente. Todos tem que apresentar. Faltar na apresentação implica em ter zero como avaliação do trabalho.

O Banquete do Rei

(problema adaptado do *Little Book of Semaphores*, de A. Downey)

Num reino muito distante, tem um rei chamado Stanizlaw que tem uma corte com 30 membros. Este rei resolveu dar um banquete e convidou os 30 membros para jantar.

O tacho com a Paella serve até 12 comensais por vez. Quando um comensal quer comer, vai até o tacho e pega uma porção.

Não há uma ordem para os convidados, eles acessam o tacho de forma aleatória;

Informações do banquete:

- Um tacho serve 12 comensais;
- Quando acaba a comida, deve ser chamado o cozinheiro para repor a Paella;
- Se o convidado se dirigir a mesa e não tiver comida, volta para o lugar sem comer e aguarda o cozinheiro avisar que o jantar está servido.
- O Cozinheiro leva 3 tempos para preparar o tacho e mais 2 para trocar de tacho na mesa. Quando está tudo pronto o cozinheiro grita para todos que a comida está servida e volta para a cozinha.

O comportamento de cada *thread* “convidado” é o seguinte:

```
while True:
  Se tiver comida
        servir()
        comer()
  Senão
        chamar_ozinheiro()
```

A *thread* “cozinheiro” tem o seguinte comportamento:

```
while True:
  preparar_comida()
  encher_tacho()
  avisar_corte()
  voltar_cozinha()
```

As restrições de sincronização são:

Comensais:

- Não podem se servir ao mesmo tempo (mas podem comer ao mesmo tempo).
- Podem repetir o prato o quanto quiserem.
- Não pode comer na frente do tacho, ou seja, após se servir deve ir comer e deixar outro convidado se servir.
- Levam de 2 a 5 tempos para comer e 1 tempo para se servir
- Não podem se servir se o tacho estiver vazio.
- O cozinheiro só pode trocar de tacho quando este estiver vazio.
- Um segundo tem 60 tempos

Desenvolver um programa para resolver o problema do Banquete do Rei e do cozinheiro para atender as restrições de sincronização.

Roteiro

1. Implementar uma solução com controle de impasses que permita o máximo paralelismo entre os Convidados comendo.
2. Pesquise sobre semáforos Posix (TDE de estudo prévio para o trabalho);
   https://computing.llnl.gov/tutorials/pthreads/index.html

   http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html
3. Recomendo que implemente o programa usando semáforos POSIX em ambiente Linux.
4. Medir o número de Convidados que podem comer por segundo
5. Para medir o desempenho de cada solução, a *thread* principal do Rei (`main`) pode executar o seguinte laço, após ter criado as demais *threads* dos convidados:

```
while(1)
{
   sleep (1) ;
   pthread_mutex_lock (&mut) ;
   printf ("Refeições por segundo: %d\n", numRefeicoes) ;
   numRefeicoes = 0 ;
   pthread_mutex_unlock (&mut) ;
}
```

O contador `numRefeicoes` deve ser incrementado cada vez que um selvagem se servir. O *Mutex* `mut` serve para evitar condições de disputa envolvendo esse contador. Ambos devem ser variáveis globais para serem vistos por todas as threads.

Entrega:

- Deve ser entregue um programa fonte, codificado na linguagem C;
- O programa terá seu código revisto pelo professor (que não fará correções no código) e será executado no ambiente Linux Ubuntu da máquina do professor ou Windows 10;
- Será compilado e executado e se não executar ou contiver erros de compilação terá avaliação 0 (zero);

Material de apoio do livro do ABRAHAM SILBERSCHATZ, 9th edition.

4.4.1 Pthreads

Pthreads refers to the POSIX standard (IEEE 1003.1c) defining an API for thread creation and synchronization. This is a ***specification*** for thread behavior, not an ***implementation***. Operating-system designers may implement the specification in any way they wish. Numerous systems implement the Pthreads specification; most are UNIX-type systems, including Linux, Mac OS X, and Solaris. Although Windows doesn't support Pthreads natively, some thirdparty implementations for Windows are available.

The C program shown in Figure 4.9 demonstrates the basic Pthreads API for constructing amultithreaded program that calculates the summation of a nonnegative integer in a separate thread. In a Pthreads program, separate threads

4.4 Thread Libraries 173

```
#include <pthread.h>
#include <stdio.h>
int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */
int main(int argc, char *argv[])
{
        pthread t tid; /* the thread identifier */
        pthread attr t attr; /* set of thread attributes */
        if (argc != 2) {
        fprintf(stderr,"usage: a.out <integer value>₩n");
        return -1;
        }
        if (atoi(argv[1]) < 0) {
        fprintf(stderr,"%d must be >= 0₩n",atoi(argv[1]));
        return -1;
        }
        /* get the default attributes */
        pthread attr init(&attr);
        /* create the thread */
        pthread create(&tid,&attr,runner,argv[1]);
        /* wait for the thread to exit */
        pthread join(tid,NULL);
        printf("sum = %d₩n",sum);
}
/* The thread will begin control in this function */
void *runner(void *param)
{
        int i, upper = atoi(param);
        sum = 0;
        for (i = 1; i <= upper; i++)
        sum += i;
        pthread exit(0);
}
```

Figure 4.9 Multithreaded C program using the Pthreads API.
begin execution in a specified function. In Figure 4.9, this is the runner() function. When this program begins, a single thread of control begins in main(). After some initialization, main() creates a second thread that begins control in the runner() function. Both threads share the global data sum.
Let's look more closely at this program. All Pthreads programs must include the pthread.h header file. The statement pthread t tid declares

```
#define NUM THREADS 10
/* an array of threads to be joined upon */
pthread t workers[NUM THREADS];
for (int i = 0; i < NUM THREADS; i++)
pthread join(workers[i], NULL);
```

Figure 4.10 Pthread code for joining ten threads.
the identifier for the thread we will create. Each thread has a set of attributes, including stack size and scheduling information. The pthread attr t attr declaration represents the attributes for the thread.We set the attributes in the function call pthread attr init(&attr). Because we did not explicitly set any attributes, we use the default attributes provided. (InChapter 6, we discuss some of the scheduling attributes provided by the Pthreads API.) A separate thread is created with the pthread create() function call. In addition to passing the thread identifier and the attributes for the thread, we also pass the name of the function where the new thread will begin execution—in this case, the runner() function. Last, we pass the integer parameter that was provided on the command line, argv[1].
At this point, the program has two threads: the initial (or parent) thread in main() and the summation (or child) thread performing the summation operation in the runner() function. This program follows the fork-join strategy described earlier: after creating the summation thread, the parent thread will wait for it to terminate by calling the pthread join() function. The summation thread will terminate when it calls the function pthread exit(). Once the summation thread has returned, the parent thread will output the value of the shared data sum.
This example program creates only a single thread. With the growing dominance of multicore systems, writing programs containing several threads has become increasingly common. A simple method for waiting on several threads using the pthread join() function is to enclose the operation within a simple for loop. For example, you can join on ten threads using the Pthread code shown in Figure 4.10.

4.4.2 Windows Threads

The technique for creating threads using theWindows thread library is similar to the Pthreads technique in several ways. We illustrate the Windows thread API in the C program shown in Figure 4.11. Notice that we must include the windows.h header file when using theWindows API.
Just as in the Pthreads version shown in Figure 4.9, data shared by the separate threads—in this case, Sum—are declared globally (the DWORD data type is an unsigned 32-bit integer). We also define the Summation() function that is to be performed in a separate thread. This function is passed a pointer

to a void, which Windows defines as LPVOID. The thread performing this function sets the global data Sum to the value of the summation from 0 to the parameter passed to Summation().

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */
/* the thread runs in this separate function */
DWORD WINAPI Summation(LPVOID Param)
{
        DWORD Upper = *(DWORD*)Param;
        for (DWORD i = 0; i <= Upper; i++)
        Sum += i;
        return 0;
}
int main(int argc, char *argv[])
{
        DWORD ThreadId;
        HANDLE ThreadHandle;
        int Param;
        if (argc != 2) {
        fprintf(stderr,"An integer parameter is required₩n");
        return -1;
        }
        Param = atoi(argv[1]);
        if (Param < 0) {
        fprintf(stderr,"An integer >= 0 is required₩n");
        return -1;
        }
        /* create the thread */
        ThreadHandle = CreateThread(
        NULL, /* default security attributes */
        0, /* default stack size */
        Summation, /* thread function */
        &Param, /* parameter to thread function */
        0, /* default creation flags */
        &ThreadId); /* returns the thread identifier */
        if (ThreadHandle != NULL) {
        /* now wait for the thread to finish */
        WaitForSingleObject(ThreadHandle,INFINITE);
        /* close the thread handle */
        CloseHandle(ThreadHandle);
        printf("sum = %d₩n",Sum);
        }
}
```

Figure 4.11 Multithreaded C program using the Windows API.

Threads are created in the Windows API using the CreateThread() function, and—just as in Pthreads—a set of attributes for the thread is passed

to this function. These attributes include security information, the size of the stack, and a flag that can be set to indicate if the thread is to start in a suspended state. In this program, we use the default values for these attributes. (The default values do not initially set the thread to a suspended state and instead make it eligible to be run by the CPU scheduler.) Once the summation thread is created, the parent must wait for it to complete before outputting the value of Sum, as the value is set by the summation thread. Recall that the Pthread program (Figure 4.9) had the parent thread wait for the summation thread using the pthread join() statement.We perform the equivalent of this in the Windows API using the WaitForSingleObject() function, which causes the creating thread to block until the summation thread has exited.
In situations that require waiting for multiple threads to complete, the WaitForMultipleObjects() function is used. This function is passed four parameters:

1. The number of objects to wait for
2. A pointer to the array of objects
3. A flag indicating whether all objects have been signaled
4. A timeout duration (or INFINITE)

For example, if THandles is an array of thread HANDLE objects of size N, the parent thread can wait for all its child threads to complete with this statement:
WaitForMultipleObjects(N, THandles, TRUE, INFINITE);