13

Sincronização, Deadlock, Starvation Mutex Locks e Semaphores (Posix)

Notas: *Linux man pages.* É importante consultar as paginas sobre os comandos
(If you don't run Linux, be sure to check your local man pages or via the Internet!)

>man pthreads
>man sem_wait

13.1 Revisão de Definições

- Deadlock : dois ou mais processos estão a esperar indefinidamente por um evento que só pode ser provocado por um dos processos que está esperando.

- Considere o seguinte exemplo. Sejam P_0 e P_1 dois processos concorrentes e S e Q dois semáforos binários com valor inicial 1, onde temos a seguinte sequência de execução. Deverá agora tentar considerar as várias possibilidades de execução !

P_0	P_1
wait(*S*);	*wait*(*Q*);
wait(*Q*);	*wait*(*S*);
sec. critica	*sec. critica*
signal(*S*);	*signal*(*Q*);
signal(*Q*)	*signal*(*S*);

- Starvation – bloqueio indefinido. Um processo, por qualquer que seja o motivo, não consegue avançar. Por exemplo, um processo que nunca pode ser removido da fila de semáforo na qual está suspenso devido a uma falta duma operação de "signal"
- As operações de semáforos
    1. Wait(). A chamada no sintaxe posix é sem_wait()
    2. Signal(). A chamada no sintaxe posix é sem_post()

P

13.2 O Problema Dos Alunos Esfomeados

Ou Usando Locks e Semáforos Para Gerir Um Problema De Recursos Limitados.

Considere a seguinte situação. Devido ao aumento de propinas chegou-se ao fim de semestre com N alunos esfomeados. O departamento de Informática, sempre a cuidar dos seus alunos, fornece um quantidade ilimitada de comida e cerveja para os seus alunos. Infelizmente para comer, cada aluno precisa de usar um garfo e devido aos cortes no orçamento do departamento foi possível arranjar apenas K (K<N) garfos. Para que todos os alunos possam ir comendo em vez de comer sequencialmente decide-se que cada aluno vai agarrar num garfo comer um pouco e depois largar o garfo para que haja a possibilidade de outro aluno comer. Um programa, fome.c, utilizando a biblioteca de pthreads para simular esta situação será fornecido.

O algoritmo básico de cada aluno é dado em baixo. Na implementação do Professor o ciclo faz-se 100 vezes.

```
Thread aluno:
FAZER {
        Tentar Agarrar num dos Garfos
        a comer
        …
        Largar garfo
        Conversar e beber cerveja
        …
} ENQUANTO ( TEM FOME );
```

Ouput Típico do Programa

Aluno 1 a esperar dum Garfo
Aluno 1 a Comer
Aluno 2 a esperar dum garfo
Total 3 a comer

Exercício.1

Obter o código. O ficheiro fome.c do código pode ser obtido no rep. git.
ou na maquina: alunos.di.ubi.pt no directório ~crocker/so/cprogs/sinc
ou ou ver o anexo 3 deste documento etc.

Estudar, compilar, executar e observar o output.
Repare que o código **ainda** não controle o numero máximo de alunos que podem comer simultaneamente.

```
cc -o fome fome.c -lpthread
./fome
```

Utilizando grep para ver apenas as linhas que fazem printf do total numero de launos a comer num dado instante.

```
./fome | grep "total 4"      ou      ./fome | grep "total 3"
```

O objetivo final é que com K=3 não deve haver output a dizer "total 4"

Exercício.2

O programa tem uma condição de corrida, nomeadamente nas actualizações da variável "numx" que represente o numero de alunos a comer simultaneamente. Insere uma variável do tipo exclusão mútua "mutex" para as duas secções criticas (numx++ e numx--) e guarde a sua solução com o nome fome2.c

```
pthread_mutex_t trinco;                 //declaração global
pthread_mutex_init(&trinco,NULL);       //inicialização na função main
pthread_mutex_lock(&trinco);            //utilização do trinco na função executada pela thread
secções criticas :  numx++ // numx--
pthread_mutex_unlock(&trinco);
```

Exercício.3

Uma primeira (e errada!) tentativa de controlar o numero de alunos a comer é inserir um ciclo do tipo "espera ocupado" antes dum aluno pegar num garfo. Experimente este solução no ficheiro fome3.c. Existem casos onde K+1 alunos podem comer ? Explique o sucedido !

```
while ( numx >= K )
    ;  /* spin lock ou espera ocupado*/
pausa
 pthread_mutex_lock(&trinco);
 numx++;
 pthread_mutex_unlock(&trinco);
```

3.1 Consistência.

- Explique como, usando mais uma variável do tipo *mutex*, se possa corrigir o problema descrito anterior fazendo lock a volta do *spin-lock* e instrução da incrementação.
- Implemente esta solução usando a variável de exclusão mútua *trinco2* e guarde a solução com o nome fome3-1.c

```
pthread_mutex_lock(&trinco2);
  while…
  lock …
    numx++
  unlock
pthread_mutex_unlock(&trinco2);
```

- Porquê é que não podemos ter usado apenas o *mutex* trinco. ?

3.2 Starvation.

- Se um aluno/thread não comer durante um certo período do tempo morre de fome (starvation) ! Existe esta possibilidade usando esta segunda variável de exclusão mútua de simulação? Explique.
- Substitui a segunda variável de exclusão mutua (trinco2) por um semáforo binário e guarde a sua solução com o nome fome3-2.c

```
sem_wait(&trinco2);
  while…
  lock …
    numx++
 unlock
sem_post(&trinco2);
```

Exercício.4

Implemente uma nova solução usando um semáforo para representar os garfos disponíveis evitando assim a necessidade do ciclo de espera ocupada (*while..)* e semáforo binário (*trinco2*).

Nota que ainda são necessárias as variáveis *numx* e a variável *mutex trinco*

Repare que o valor de inicialização do semáforo é "K".

Guarde a sua solução como o nome fome4.c
(ver programa fome4.c em anexo)

Na função main()

```
sem_init(&garfo,0,K)
```

Na função da thread

```
sem_wait(&garfo)

...lock
    numx++
 unlock .
.printf ...
lock
  numx—
unlock...

sem_post(&garfo)
```

Exercício.5

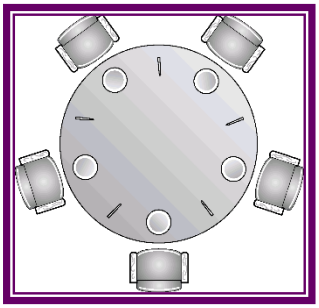
Dado que o semáforo represente o numero de garfos disponíveis podemos simplesmente calcular o número de alunos a comer simultaneamente através do próprio valor de semáforo !!
Investigue a função sem_getvalue() da library de pthreads e implemente uma solução com esta função removendo a variável *numx* e a sua variável de exclusão mútua *trinco*.
Guarde a sua solução como o nome fome5.c.

Bom Trabalho !

P

13.3 O Problema de Jantar dos Filósofos

Este é um problema clássico de sincronização devido ao facto de modelar uma grande classe de problemas de concorrência onde há necessidade de reservar recursos entre vários processos sem deadlock ou starvation. O jantar consiste em, por exemplo, 5 filósofos a comer num restaurante chinês numa mesa redonda onde há - Um "pau" para cada par de filósofos. Para comer um filósofo precisa de ter dois paus. Os filósofos alternam entre pensar e comer



```
do {
        wait( chopstick[i] )                    //agarrar pau à esquerda
        wait( chopstick[(i+1) % N] )            //agarrar pau à direira
        …
        comer
        …
        signal( chopstick[i] );                 //largar pau à esquerda
        signal( chopstick[(i+1) % N] );         //largar pau à direita
        …
        pensar
        …
} while (1);
```

Algoritmo Geral do "Jantar dos Filósofos".

O Jantar dos filósofos será modelado usando uma thread para representar um filósofo e semáforos para representar os "paus". O acto de agarrar um pau é modelado pelo acto de obter um semáforo (primitiva de wait). O acto de largar um pau é representado pela acto de deixar de utilizar o semáforo (primitiva de signal). O psuedo-código para um filosofo é dado em cima. Aqui cada filósofo (cada thread) vai fazer o algoritmo, portanto todos fazem a mesma sequência de acções.

Com este algoritmo vai haver uma situação de "Deadlock" (bloquei mútua). Soluções podem ser

- Simétricas. Aqui todos os filósofos fazem a mesma coisa → seguem o mesmo algoritmo) através a utilização de mais primitivas de sincronização ou até terceira entidades (árbitros!)
- Assimétricas. Aqui os filósofos podem seguir algoritmos diferentes, por exemplo alguns podem agarrar pelo ordem direita/esquerda em vez de esquerda/direita

Exercícios

1. Implementar utilizando pthreads o algoritmo em cima (o código é fornecido em baixo ou na Internet) – experimente executando o código para ver situações de bloqueio. Deverá comentar o código com os seus comentários !

2. Uma solução assimétrica é deixar um, (ou mais ?) por exemplo o primeiro dos filósofos, agarrar e largar os paus por outra ordem (da direita para a esquerda em vez da esquerda para a direita). Modificar o código para implementar este algoritmo e experimente. Há deadlock ? Pode haver starvation ?

3. Implemente uma solução simétrica. Uma ideia é de considerar o caso dum arbitro que indica se um filosofo pode ou não tentar agarrar os "dois" paus. A ideai deste arbitro é simples de implementar .. baste incluir um lock de exclusão mútua a volta do código para agarrar os dois paus. Com esta solução Há deadlock ? Pode haver starvation ?

Implementação do Jantar dos Filósofos com Pthreads : O exemplo que segue vai "bloquear"

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>


#define PAUSA(A) {fflush(stdin);printf("Pausa %s\n",A);getchar();}

#define pausa {double \
indicez1;for(indicez1=0;indicez1<1000;indicez1=indicez1+0.9);}

#define N 4  //numero de filosofos

sem_t pau[N];   //variavel partilhada

void *filosofo (void *args);
```

```
int main (int argc, char *argv[])
{
  pthread_t th[N];  //identificadores dos threads
  int i, id[N];

  for (i = 0; i < N; i++) {
    id[i] = i;
    sem_init ( &pau[i], 0, 1);
  }

  PAUSA ("Start: Iniciar press return ")

  for (i = 0; i < N; i++)
    pthread_create (&th[i], NULL, filosofo, &id[i]);

  for (i = 0; i < N; i++)
    pthread_join (th[i], NULL);

  return 0;
}
```

P

```
void *filosofo (void *args)
{
  int id,esq,dir, comer=10000;

  id = *((int *) args);

  esq=id;
  dir=(id+1)%N;

  while (comer > 0) {
      printf ("Estou a pensar thread %d %d\n", id,comer);
      pausa
      sem_wait (&pau[esq]);
      pausa
      sem_wait(&pau[dir]);

         pausa
         printf ("Estou a comer -thread %d %d\n", id,comer);
         pausa
         comer = comer - 1;
         pausa

      sem_post (&pau[esq]);
      pausa
      sem_post (&pau[dir]);
      pausa
    }
  return (NULL);
}
```

Obter o Código pela Página do Professor aqui: Código para exercício de jantar

O código também pode ser obtido na maquina: alunos.di.ubi.pt directório ~crocker/so/cprogs/sinc/filosofo.c

P

Anexo 1 Rotinas (Posix) da pthreads

Rotinas de criação e juncção duma thread e exclusão mutua <pthreads.h>

Tipo: pthread_t - representação duma thread

- int pthread_create(pthread_t * thread, pthread_attr_t *attr, void * (*start_routine)(void *), void * arg);
- int pthread_join(pthread_t th, void **thread_return);

Tipo: pthread_mutex_t - representação do tipo de exclusão mútua

- int pthread_mutex_init(pthread_mutex_t * mutex,const pthread_mutexattr_t * attr);
- int pthread_mutex_lock(pthread_mutex_t *mutex);
- int pthread_mutex_unlock(pthread_mutex_t *mutex);

SEMAPHORES operations on semaphores <semaphore.h>

Tipo: sem_t representação dum semáforo

- int sem_init(sem_t *sem, int pshared, unsigned int value);
- int sem_wait(sem_t * sem);
- int sem_post(sem_t * sem);

Também:

- int sem_trywait(sem_t * sem);
- int sem_getvalue(sem_t * sem, int * sval);
- int sem_destroy(sem_t * sem);

Semaphores are counters for resources shared between threads. The basic operations on semaphores are: increment the counter atomically, and wait until the counter is non-null and decrement it atomically. This manual page documents POSIX 1003.1b semaphores, not to be confused with SystemV semaphores as described in ipc(5), semctl(2) and semop(2).

sem_init initializes the semaphore object pointed to by sem. The count associated with the semaphore is set initially to value. The pshared argument indicates whether the semaphore is local to the current process (pshared is zero) or is to be shared between several processes (pshared is not zero). LinuxThreads currently does not support process-shared semaphores, thus sem_init always returns with error ENOSYS if pshared is not zero.

sem_wait suspends the calling thread until the semaphore pointed to by sem has non-zero count. It then atomically decreases the semaphore count.

sem_trywait is a non-blocking variant of sem_wait. If the semaphore pointed to by sem has non-zero count, the count is atomically decreased and sem_trywait immediately returns 0. If the semaphore count is zero, sem_trywait immediately returns with error EAGAIN.

sem_post atomically increases the count of the semaphore pointed to by sem. This function never blocks and can safely be used in asynchronous signal handlers.

sem_getvalue stores in the location pointed to by sval the current count of the semaphore sem.

sem_destroy destroys a semaphore object, freeing the resources it might hold. No threads should be waiting on the semaphore at the time sem_destroy is called. In the LinuxThreads implementation, no resources are associated with semaphore objects, thus sem_destroy actually does nothing except checking that no thread is waiting on the semaphore.

RETURN VALUE
The sem_wait and sem_getvalue functions always return 0. All other semaphore functions return 0 on success and -1 on error, in addition to writing an error code in errno.

P

Anexo 2 Linux / MacOS

Because of the difficulty of using the pthread semaphores on MacOs X (for instance if you just port your code with semaphores directly from Linux to MacOS X it just won't work properly although, somewhat irritating, it will compile ok) I decided to write a simple mechanism that lets one compile simple code on either Linux or MacOSx by simply flipping a macro value. The usual semaphore definition and function calls are simply replaced by capital letters and the removal of the address sign. It can easily be extended for the other semaphore function calls. I prefer to use the Mach OSX kernel semaphore synchronization primitives rather than the Cocoa Posix supported named semaphores, but whatever you use remember that semaphores can be implemented in several ways so Mach/Cocoa semaphores may not behave in the same way as semaphores on other platforms. References are given below for using semaphores on MacOS.
http://developer.apple.com/MacOsX/multithreadedprogramming.html
http://developer.apple.com/documentation/Darwin/Conceptual/KernelProgramming/synchronization/synchronization.html

The Basic semaphore declaration and function calls will be replaced by the following:

Declaration:	SEM_T nome;
Initialization:	SEM_INIT(nome, attributes, valor inicial);
Usage:	SEM_POST(nome);
Usage:	SEM_WAIT(nome);

In this simple example the usual posix calls are replaced by the macros defined above.

File ex1.c

```
#include "semmaps.h"
#define N 20                    //number of threads

SEM_T  smutex;                  //declaration of a binary semaphore for mutual exclusion
int x = 0;                      //global  variable

void *f (void *args)
{
  SEM_WAIT (smutex);            //each thread adds one to a shared variable
  x = x + 1;
  SEM_POST (smutex);
}

int main (int argc, char *argv[])
{
  pthread_t th[N];
  int i;

  SEM_INIT (smutex, 0, 1);      //initialization of the binary semaphore.

  for (i = 0; i < N; i++)
    pthread_create (&th[i], NULL, f, NULL);
  for (i = 0; i < N; i++)
    pthread_join (th[i], NULL);
  printf("x = %d  (=%d)\n",x,N);
  return 0;
}
```

P

The file semmaps.h which maps between the Linux/MacOsX thread calls
You can chose between the Linux Posix or linux Posix Named threads and the Darwin (MacOS) Mach semaphores or Darwin Posix named semaphores,

```
#ifndef _SEMMAPS
#define SEMMAPS 1

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

//#define LINUX
//#define LINUX_POSIXNAMED
//#define DARWIN_MACH
//#define DARWIN_POSIXNAMED

#ifdef LINUX
  #include <semaphore.h>
  #define SEM_T sem_t
  #define SEM_INIT(SEM,V,INITIAL)        sem_init(&SEM, V, INITIAL);
  #define SEM_WAIT(SEM)                  sem_wait(&SEM);
  #define SEM_POST(SEM)                  sem_post(&SEM);
  #define  _REENTRANT
#endif


#ifdef DARWIN_MACH
  #include <mach/mach_init.h>
  #include <mach/task.h>
  #include <mach/semaphore.h>
  #define  _REENTRANT
  #define SEM_T semaphore_t
  #define SEM_INIT(SEM,V,INITIAL)  semaphore_create(current_task(), &SEM, SYNC_POLICY_FIFO, INITIAL);
  #define SEM_WAIT(SEM)          semaphore_wait(SEM);
  #define SEM_POST(SEM)          semaphore_signal(SEM);
#endif
```

```
#ifdef LINUX_POSIXNAMED
#include <semaphore.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
/*
  notes
  tmpnam(NULL) will not return a correctly formed string name
   has to start with /xyz
  note that this will create a kernel persistant semaphore in /dev/shm whose name is sem.xyz
   will need to reboot or sem_unlink to remove this
   However .. only get one semaphore..
*/
#define SEM_T sem_t *
#define SEM_INIT(SEM,V,INITIAL)    {char *sname="/tmp1"; \
                                        SEM = sem_open( sname , O_CREAT, S_IRUSR|S_IWUSR, INITIAL);\
                                        if (SEM==SEM_FAILED) printf("sem failed %d %s\n",errno,strerror(errno)); }
#define SEM_WAIT(SEM)             sem_wait(SEM);
#define SEM_POST(SEM)             sem_post(SEM);
#endif


#ifdef DARWIN_POSIXNAMED
#include <semaphore.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
/* Notes
 sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int initialValue);
 tmpnam(NULL) this will create a semaphore with a unique name
 see man pages for details
  sem_open return an address but in <sys/semaphore.h> we have define SEM_FAILED -1  .. strange!
  so i put the typecast in below just to avoid seeing the warning
*/
#define SEM_T sem_t *
#define SEM_INIT(SEM,V,INITIAL)     {SEM = sem_open(  tmpnam(NULL), O_CREAT, S_IRUSR|S_IWUSR, INITIAL); \
                          if ((int)SEM==SEM_FAILED) printf("sem failed %d %s\n",errno,strerror(errno)); }
#define SEM_WAIT(SEM)             sem_wait(SEM);
#define SEM_POST(SEM)             sem_post(SEM);
#endif

#endif
```

Compilation

Using : semmaps.h

Either un-comment out the line with the #define ARCHITECTURE

Or pass the flag to the compiler, for instance CC –DLINUX_POSIXNAMED

Anexo 3 fome4.c

Here we have 10 hungry and thirsty students but only have three sets of knives and forks.At any one time we can have a maximum of three students eating .. on a Mac try flipping the #defines and watch the results.

```
#define N 10  //number de students - one student is one thread
#define K 3   //number of sets of knives and forks

#define PAUSA(A) {fflush(stdin);printf("Pausa %s\n",A);getchar();}

pthread_mutex_t        mutex;
int                    numx = 0;
SEM_T                  garfo;                          //sem_t garfo posix

int main(int argc, char *argv[])
{
        pthread_t th[N];
        int  i, id[N];
        for (i = 0; i < N; i++) id[i] = i;
        pthread_mutex_init(&mutex, NULL);
        SEM_INIT(garfo,0,K);                           //sem_init(&garfo,0,K)
        PAUSA("Start: Iniciar press return ")
        for (i = 0; i < N; i++)
                pthread_create(&th[i], NULL, student, &id[i]);
        for (i = 0; i < N; i++)
                pthread_join(th[i], NULL);
        return 0;
}

void *student(void *args)
{
        int comer = 100, id = *((int *) args);
        while (comer > 0) {

           printf("Aluno %d a espera\n",id);
           SEM_WAIT(garfo);                            //sem_wait(&garfo);

           pthread_mutex_lock(&mutex);
           numx++;
           pthread_mutex_unlock(&mutex);

           printf("Aluno %d a beber comer : ****%d a comer\n", id, numx);

           pthread_mutex_lock(&mutex);
           numx--;
           pthread_mutex_unlock(&mutex);

           SEM_POST(garfo);                            //sem_post(&garfo);

           printf("Aluno %d a beber cerveja\n", id);
           comer--;
        }
        return NULL;
}
```