CS345 Operating Systems

Tutorial 2: Producer-Consumer

Threads, Shared Memory, Synchronization

Threads

- Athread is a light - weight process.
- Athread exists within a process, and uses the process resources.
- It is asynchronous.
- The program in C calls the pthread.h header file.

- How to compile:

```
gcc hello.c –pthread –o hello
```

Creating a thread

```
int pthread_create( pthread_t * thread,    pthread_attr_t
*attr,  void * (*func)(void *),
void *arg );
```

Returns 0 for success, (>0) for error.

- 1st arg (*thread) – pointer to the identifier of the created thread.
- 2nd arg (*attr) – thread attributes. If NULL, then the thread is created with default attributes
- 3rd arg (*func) – pointer to the function the thread will execute
- 4th arg (*arg) – the argument of the executed function

Shared memory

- A shared memory segment is a portion of physical memory that is virtually shared between multiple processes.
- In this assignment we are dealing with intra-process communication.
- All the global variables of a program-process are shared memory for it’s threads.

Shared memory - concerns

- Needs concurrency control/synchronization (data inconsistencies are possible)
- Processes should be informed if it's **safe to read and write** data to the shared resource.

Thread synchronization mechanisms

- Mutual exclusion (mutex)

  Used to serialize access to the shared memory.
  It is a locking mechanism.

- Semaphores

  A generalized mutex, that allow us to split the buffer and access separately each resource.
  It is a signaling mechanism.

Mutexes

- guard against multiple threads modifying the same shared data simultaneously
- provide locking/unlocking critical code sections where shared data is modified
- each thread waits for the mutex to be unlocked (by the thread who locked it) before performing the code section

Mutexes-create and initialize

Mutex variables are declared with type pthread_mutex_t, and must be initialized before they can be used.

There are two ways to initialize a mutex variable:

1. Statically, when it is declared. For example: pthread_mutex_t mut= PTHREAD_MUTEX_INITIALIZER;
2. Dynamically, with the pthread_mutex_init()routine. This method permits setting mutex object attributes, attr.

The mutex is initially unlocked.

Routines :

pthread_mutex_init (mutex, attr) pthread_mutex_destroy (mutex)

Mutexes – basic functions

```
int pthread_mutex_lock(pthread_mutex_t*mutex); int
pthread_mutex_trylock(pthread_mutex_t*mutex);  int
pthread_mutex_unlock(pthread_mutex_t*mutex);
```

- a mutex is like a key (to access the code section) that is handed to only one thread at a time
- the lock/unlock functions work together
- a mutex is unlocked **only by the thread that has locked it.**

```
#include <pthread.h>
...
pthread_mutex_t my_mutex;
...
int main()
{
        int tmp;
        ...
        // initialize the mutex
        tmp= pthread_mutex_init( &my_mutex, NULL );
        ...
        // create threads
        ...
        pthread_mutex_lock( &my_mutex); do_something_private();
        pthread_mutex_unlock( &my_mutex);
        … pthread_mutex_destroy(&my_mutex); return 0;
}
```

Whenever a thread reaches the lock/unlock block, it first determines if the mutex is locked. If so, it waits until it is unlocked. Otherwise, it takes the mutex, locks the succeeding code, then frees the mutex and unlocks the code when it's done.

Semaphores

Counting Semaphores:

- permit a limited number of threads to execute asection of the code
- similar to mutexes (if we use binary semaphores it's the same)
- should include the semaphore.h header file
- semaphore functions do not have pthread_prefixes; instead, they have sem_prefixes

Semaphores – basic functions

- Creating a semaphore:
  int sem_init (sem_t*sem, int pshared, unsigned int value);
  –initializes a semaphore object pointed to by sem
  –pshared is a sharing option; a value of *0 means the semaphore is local to the calling process*
  –gives an initial value value to the semaphore

- Terminating a semaphore:
  int sem_destroy (sem_t*sem);
  –frees the resources allocated to the semaphore sem
  –an error will occur if a semaphore is destroyed for which a thread is waiting

Semaphores – basic functions

- Semaphore control:

  int sem_post(sem_t*sem);

  –atomically increases the value of a semaphore by 1, i.e., when 2 threads call sem_post simultaneously, the semaphore's value will also be increased by 2 (there are 2 atoms calling)

  int sem_wait(sem_t*sem);

  –atomically decreases the value of a semaphore by 1; but always waits until the semaphore has a non-zero value first

Mutex vs Semaphores

	Semaphore	**Mutex**
Μέθοδος	Συνηθισμένη μέθοδος **Signal mechanism**	Locking mechanism
Τι είναι	**Integer** Counting or binary semaphore	**Object**
Λειτουργία	Πολλαπλά νήματα προγράμματος μπορούν να έχουν πρόσβαση σε περιορισμένο αριθμό instance πηγών.	Πολλαπλά νήματα προγράμματος μπορούν να έχουν πρόσβαση σε έναν πόρο, αλλά όχι ταυτόχρονα.

Mutex vs Semaphores

	Semaphore	**Mutex**
Δικαιώματα πρόσβασης	Η τιμή της semaphore μπορεί να ανανεωθεί από οποιοδήποτε διεργασία.	Μόνο η διεργασία που κλείδωσε το mutex, μπορεί να το ξεκλειδώσει.
Τύποι	Binary (0 και 1) Counting (1,2,3…n)	No types
Πώς	Signal/Post and wait χρησιμοποιούνται για να αλλάξουν την τιμή της	Request the lock : trylock, lock Release : unlock
Wait? (πότε μπαίνω στο critical area)	Περιμένω μέχρι το counter του resource να αυξηθεί.	Περιμένω μέχρι το lock να γίνει release.

```
#include <pthread.h> #include <semaphore.h>
...
void *thread_function( void *arg );
...
sem_t semaphore; // also a global variable just like mutexes
…

void *thread_function( void *arg )
{
        sem_wait( &semaphore );
        perform_task_when_sem_ope  n();
        ...
        pthread_exit( NULL );
}
int main()
{
        int tmp;
        ...
        // initialize the semaphore
        tmp = sem_init( &semaphore, 0, 0 );
        ...
        // create threads
        pthread_create( &thread[i], NULL, thread_function,  NULL );
        ...
        while ( still_has_something_to_do() )
        {
                sem_post( &semaphore );
                ...
        }
        ...
        pthread_join( thread[i], NULL );  sem_destroy(
        &semaphore ); return 0;
}
```

the main thread increments the semaphore's count value in the while loop

the threads wait until the semaphore's count value is non-zero before performing perform_task_when_sem_open() and further

A Simple working Example

Creating a thread that prints “Hello World”

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

void *hello_world(void * ptr) {
        printf("Hello World! I am a thread!\n");
        pthread_exit(NULL);
}

int main(int argc, char * argv[]){
        pthread_t thread;
        int rc;

        rc = pthread_create(&thread, NULL, hello_world, NULL);
        if (rc) {
                printf("ERROR: return code from pthread_create() is
                %d\n",
rc);
                exit(-1);
        }

        pthread_exit(NULL);
}
```

A Simple working Example

Creating two threads: The first prints "Hello" and the second prints "World".

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


void *print_Hello( void *ptr ){
        printf(“Hello”);
}
void *print_World( void *ptr ){
        printf(“World”);
}


int main(int argc, char * argv[] ){
        pthread_t t1, t2;
        int rc, rc2;


        rc = pthread_create(&t1, NULL, print_Hello, NULL);
        if (rc) {
                printf("ERROR: return code from pthread_create() is %d\n",
                rc);  exit(-1);
        }
        rc2 = pthread_create(&t2, NULL, print_World, NULL);
        if (rc2) {
                printf("ERROR: return code from pthread_create() is %d\n",
                rc);
                exit(-1);
        }
pthread_join( thread1, NULL); /*Wait for the thread to finish*/
pthread_join( thread1, NULL);

}
```

A Simple working Example

This program sometimes prints "Hello World", sometimes prints "World Hello".
Using a semaphore with intraprocess Scope can syncronize them. Now the thread t2 will never be
executed before the first threat t1.

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


sem_t sem;
void *print_Hello( void *ptr ){
        printf("Hello ");
        sem_post(&sem); //semaphore unlocked (Up)!
}
void *print_World( void *ptr ){
        sem_wait(&sem); //semaphore locked (Down)!
        printf("World\n");
}


int main(int argc, char * argv[] ){
        pthread_t t1, t2;
        int rc, rc2;
        sem_init(&sem, 0, 0 ); /*Initialize semaphore with intraprocess scope*/


        rc = pthread_create(&t1, NULL, print_Hello, NULL);
        rc2 = pthread_create(&t2, NULL, print_World,
        NULL);


pthread_join(t1, NULL); /*Wait for the thread to finish*/
pthread_join(t2, NULL);
```