Semetexes and Mutephores

Zachary Owen

February 27, 2017

1 PID Manager

This program is a simulator for a Kernal PID Manager. It uses threading to simulate diffrent processes, each trying to get a process ID from a process ID manager.

This document comes with a makefile used to make the PID Manager program. To run this type

```
make
./testpid
```

1.1 Functionality: test.c

This File contains the main and pthread:allocator functions. Program execution begins here.

1.1.1 int main()

The main function of this program begins by allocating structures for the threads, allocating our PID map and initiallizing the mutex.

Afterwards it spins up each of the threads.

Finally it cleans up the threads and the mutexs and exits.

1.1.2 void* allocator(void)

Allocators is the Pthread function. It does not use its return or its arguments. It simply goes through a cycle,

1. Sleep
2. Try to get a PID
3. If you get the PID print that you have it. And Sleep.

   If not print "No pid Available" and go back go step 1
4. Finally Release the PID and goto 1

1.2 Functionality: pid.c

This file contains most of the helper functions and allocators called in text.c

1.2.1 allocate_map()

Allocate map simply sets the PID map to zero. Pretty simple.

1.2.2 allocate_pid()

This function is called by allocator and it takes care of allocating the process ID and the mutual exclusion. Before the PID array is touched, The mutex is locked. Then the get_id() is called which does the job of actually geting the ID. The id is marked for the process and the mutex is unlocked.

1.2.3 release_pid(int)

This fucntion is called by allocator at the end of the simulated program's life. It grabs the mutex lock, unsets the processes ID, which was passed to it as it's single argument. and finally returns the mutex to its unlocked state.

1.2.4 get_id()

This function does a linear search on the Process ID pool to find one that is unallocated. If an ID is not found, -1 is returned. This function is called from allocate_pid(). It simply makes getting the PID look nicer.

1.3 Functionality: pid.h

This file only holds defines for the minimum and maximum PID values and prototypes for the pid functions in pid.c

2 Sleeping TA

This function simulates the final hour before the CSC150 program is due. There are 5 students in need of help. At random intervals they go to the TA in an attempt to see him/her about help. However the TA only has 3 chairs and the students are weird. A maximum of 3 students are allowed to queue waiting for the TA's help and any who arrive after decide to come back later.

This document comes with a makefile used to make the PID Manager program. To run this type

```
make
./sleepingta
```

2.1 Functionality: main.c

This file contains main() as well as all of the initialization functions.

2.1.1 int main()

This function begins by calling init() which initializes the mutex's and semaphores.
Then main starts up the TA thread, followed by spinning up a defined number of student threads.
Finally each one of the students is joined back to the main thread after they exit. And then the TA thread is forced to exit by main.
The semaphores and mutexes are destroyed and the program exits.

2.1.2 (

create_ta) Calls the ta() function from ta.c in a pthread. This function has no arguments and no return value.

2.1.3 create_students()

This function iterates through a for loop spinning up a defined number of Student threads passing each one a nullpointer which is actually its ID number because making structs is annoying.

This value is cast in the student function back to an integer.

2.1.4 init()

This function initiallizes two mutexes, and way too many semephores.

2.1.5 dinit()

This function destroys all that init() created for as life begets death, allocation begets deallocation. Such is the will of Kernel

2.2 Functionality: ta.c

This file holds ta_loop which is the TA pthread function.

2.2.1 ta_loop()

The thread begins by locking its door so the TA can sleep. The the TA checks the semaphore that tells it if a student is waiting on the third chair for help.
If it is, the TA follows these steps

1. The TA locks the students ability to sit down.
2. The TA Allows another student into the third seat.
3. The TA waits() the student counter followed by getting the new value.
4. The TA then unlocks the chair mutex so other students can attempt to sit.
5. The TA prints a message letting the user know that a student is recieving help and at the same time the Student lets the user know it is being helped.
6. the TA then closes its door which lets the student out so it can decide what to do next.

After this the TA waits on its semaphore again.

2.3 Functionality: ta.h

This file contains declarations for all functions in ta.c, student.c, help_student.c and hangout.c as well as all of the #defines and declarations of the mutexs and semaphores. This file also contains all of the #includes for the entire program.

2.4 Functionality: hangout.c

This file contains a single function that prints a message to the screen that reads "Student # hanging out for # seconds." Based on the student thread calling and how many seconds it plans to wait. The seconds waited are calculated in a random() call in the function call.

2.5 Functionality: help_student.c

This file contains a single function that prints a message to the screen that reads "Helping a student for # seconds" Based on how many seconds the TA thread calling it plans to wait. The seconds waited are calculated in a random() call in the function call.

2.6 Functionality: student.c

This file contains the student pthread function. It is called and created a number of times in main.

2.6.1 student_loop()

The function begins by casting its argument to an integer since when called it is given a bad pointer that is easy to cast. It is a 64 bit integer because the address is 64 bits.

The function then initializes its times through the loop variable, aptly named, and then calls hang_out() on a random amount of time between 1 and the max time specified in ta.h

After hanging out the student will attempt to grab the lock on chairs. Chairs is a mutex that allows all sitting operations. If it obtains the mutex, the student checks on the number of students currently sitting down. If that number is less than three, the student increments the counter, reobtains it, and then uses it to tell the user that it is waiting and how many students are waiting.

If the number of students sitting is 3 or more the student will opt to try again later.

After this the chairs mutex is unlocked because the student is the semaforrest. It is a 4 times locked sequential semaphores which form a queue

by grabbing the next semaphore in the list and dropping the one before it. Finally when the process hits the front of the queue, it signals the TA semaphore. This wakes the TA process which takes care of the student.

The student is then allowed to leave only when the door mutex is unlocked by TA.

The student will go through this loop a number of times, that number being 5 before exiting and joining the main thread.