

Real Time Embedded Homework 2

ECEN 5623

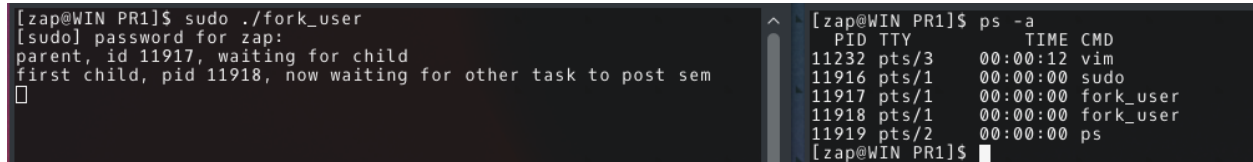
Zachary Vogel

February 24, 2016

Problem 1

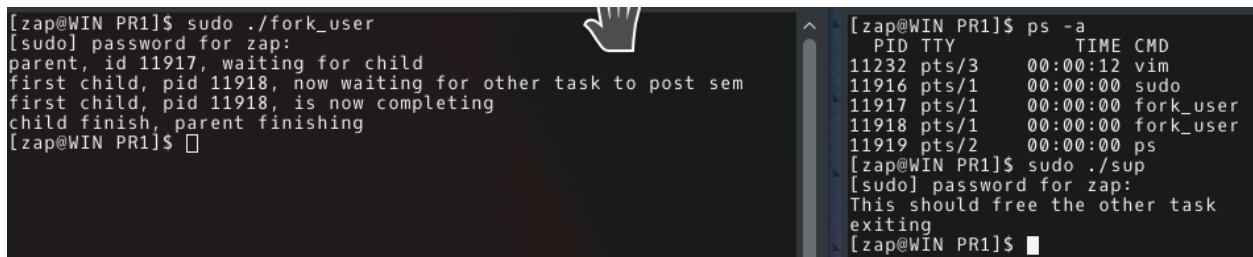
Implement a Linux process that is executed at the default priority for a user-level application and waits on a binary semaphore to be given by another application. Run this process and verify its state using the ps command to list its process descriptor. Now, run a separate process to give the semaphore causing the first process to continue execution and exit. Verify completion.

Here we see two screenshots proving my code does what it says it does. The actual code can be seen in the appendix.



```
[zap@WIN PR1]$ sudo ./fork_user
[sudo] password for zap:
parent, id 11917, waiting for child
first child, pid 11918, now waiting for other task to post sem
[zap@WIN PR1]$ ps -a
  PID TTY          TIME CMD
 11232 pts/3        00:00:12 vim
 11916 pts/1        00:00:00 sudo
 11917 pts/1        00:00:00 fork_user
 11918 pts/1        00:00:00 fork_user
 11919 pts/2        00:00:00 ps
[zap@WIN PR1]$
```

Figure 1: As you can see, the process is waiting to finish



```
[zap@WIN PR1]$ sudo ./fork_user
[sudo] password for zap:
parent, id 11917, waiting for child
first child, pid 11918, now waiting for other task to post sem
first child, pid 11918, is now completing
child finish, parent finishing
[zap@WIN PR1]$ ps -a
  PID TTY          TIME CMD
 11232 pts/3        00:00:12 vim
 11916 pts/1        00:00:00 sudo
 11917 pts/1        00:00:00 fork_user
 11918 pts/1        00:00:00 fork_user
 11919 pts/2        00:00:00 ps
[zap@WIN PR1]$ sudo ./sup
[sudo] password for zap:
This should free the other task
exiting
[zap@WIN PR1]$
```

Figure 2: Semaphore lock opened and everything finishes

Problem 2

Why is the sufficient RM least upper bound so pessimistic?

The RM LUB is pessimistic largely because it is a sufficient not necessary condition. That means that the bound can't pass a set of tasks that won't be schedulable, but it can fail a task set that will be schedulable. The RM LUB is relatively easy to calculate, so it is better off being a simple sufficiency test as opposed to an extremely complex necessary and sufficient test. The other reason for the pessimism is that task sets that aren't very harmonic will fail relatively easily under a RM scheduling algorithm. That means that despite the high utilization of a harmonic task set under RM policy the bound must be lowered for the case where tasks aren't harmonic.

Problem 3

If EDF can be shown to meet deadlines and potentially has 100% CPU utilization, then why is it not typically the hard real-time policy of choice? That is, what are the drawbacks of using EDF compared to RM/DM? In an overload situation, how will EDF fail?

EDF scheduling has several major drawbacks that make it undesirable in real-time systems. First, EDF requires extra computation time because one must compute when the deadline happens and then restructure priorities based on that. The deadline time isn't always deterministic which can lead to problems and that extra computation can slow down the system. One will also notice that for lightly loaded systems EDF and RM scheduling often generate the exact same schedule for tasks. Similarly, the more harmonic a systems tasks are the more likely that EDF and RM produce the same scheduling, so if one makes their tasks harmonic why not use the simpler RM policy. One more issue is that the sufficient bound for EDF scheduling is not easily computed as is the case for RM scheduling. It is also true that for harmonic task sets the RM policy can get an extremely high utilization. This is useful because tasks can often be made to be harmonic with minor adjustments to their computation time.

The final reason that I found was in the case of a failing system. Rate monotonic scheduling guarantees that the highest priority task will be the first task to miss a deadline if a deadline is going to be missed. This is very convenient for real time systems because of its determinism. Since one knows how the system will fail, they can build their system to react to that failure intelligently. This is not the case for EDF scheduling. In EDF scheduling, one does not know how a system will fail. It might fail the highest period task or the lowest period task first. Since one does not know which task will fail it becomes much more difficult to "fail in an elegant way." Basically, EDF scheduling does not fail in a deterministic way, which makes it harder to work with in the case of failures and will often result in cascading failures. These result in many more missed deadlines which might end up breaking even a soft real time system.

Problem 4

If a system must complete frame processing so that 100,000 frames are completed per second and the instruction count per frame processed is 2,120 instructions on a 1GHz processor core, what is the CPI required for the system? What is the overlap between instructions and IO time if the intermediate IO time is $4.5 \mu s$?

For, $f = 10^9 \text{cycles/second}$, $IPF = 2,120 \text{instructions per frame}$, and $FPS = 100,000 \text{frames per second}$, what is CPI. We need $2,120 * 100,000 = 212,000,000$ instructions per second. Then, $CPI = \frac{10^9}{212,000,000} = 4.7169 \text{CPI}$ which is about 5 cycles per instruction. Given intermediate IO time is $4.5 \mu s$, what is the overlap. The deadline time here is $10 \mu s$ because we want to complete 100,000 frames every second at even spacing. Instruction count time is the same based on the definition of the system. Thus overlap percent is $1 - \frac{10^{-5} - 4.5 * 10^{-6}}{10^{-5}} = 45\%$.

Problem 5

Review the DVD code for `heap_mq.c` and `posix_mq.c`. Write a brief paragraph describing how the two message queue applications are similar and how they are different. Make sure you not only read the code but also build it, load it and execute it to make sure you understand how both applications work.

The first major difference between the two implementations is that `heap_mq.c` creates and opens the message queue before running the sender and receiver functions. The `posix_mq.c` code opens and creates the message queue in the receiver function and it is again opened in the sender function. The other major difference is that `heap_mq.c` uses dynamically allocated memory to be used as the buffer, while `posix_mq.c` uses stack memory. `heap_mq.c` also implements the sender and receiver functions as infinite loops that must be closed by deleting the tasks, while `posix_mq.c` only runs through the sending and receiving once. Both

functions give the receiver a higher priority than the sender, both use tasks to run the sender and receiver functions, and both use the posix queue with the function calls mq_open, mq_receive, and mq_send.

Code Appendix

Waiting function at user priority

```

1 #include <unistd.h>
2 #include <sys/types.h>
3 #include <errno.h>
4 #include <stdio.h>
5 #include <sys/wait.h>
6 #include <stdlib.h>
7 #include <sys/resource.h>
8 #include <sys/time.h>
9 #include <semaphore.h>
10 #include <string.h>
11 #include <fcntl.h>
12
13 #define SNAME "/mysem"
14
15 int main(void)
16 {
17     sem_t * sem1=sem_open(SNAME,O_CREAT, 0644,0);
18
19     pid_t user_lev=fork();
20     if (user_lev>=0)
21     {
22         if (user_lev==0)
23         {
24             setpriority(PRIO_USER,user_lev,-11); //sets priority to default user level with
standard niceness
25             printf("first child, pid %d, now waiting for other task to post sem\n",getpid());
26             sem_wait(sem1);
27             printf("first child, pid %d, is now completing\n",getpid());
28             exit(0);
29         }
30         //parent, generate another child and then finish
31         else
32         {
33             int returnstat;
34             printf("parent, id %d, waiting for child\n",getpid());
35             waitpid(-1,&returnstat,0);
36             printf("child finish, parent finishing\n");
37         }
38     }
39     else{
40         printf("First fork failed\n");
41         return 1;
42     }
43     return 0;
44 }
```

Function to post the semaphore

```

1 #include <sys/resource.h>
2 #include <stdio.h>
3 #include <unistd.h>
4 #include <sys/types.h>
5 #include <errno.h>
6 #include <sys/wait.h>
7 #include <stdlib.h>
8 #include <sys/time.h>
9 #include <semaphore.h>
10 #include <string.h>
11 #include <fcntl.h>
12
```

```
13
14
15 #define SNAME "/mysem"
16
17 int main() {
18     sem_t * sem=sem_open(SNAME,0);
19     printf("This should free the other task\n");
20     sem_post(sem);
21     printf("exiting\n");
22
23     return 0;
24 }
```