Department of Electrical and Computer Engineering

University of Colorado at Boulder

ECEN5623 - Real Time Embedded Systems

Homework 2

Submitted by

Parth Thakkar

Submitted on February 25, 2024

Contents

List of Figures

List of Tables

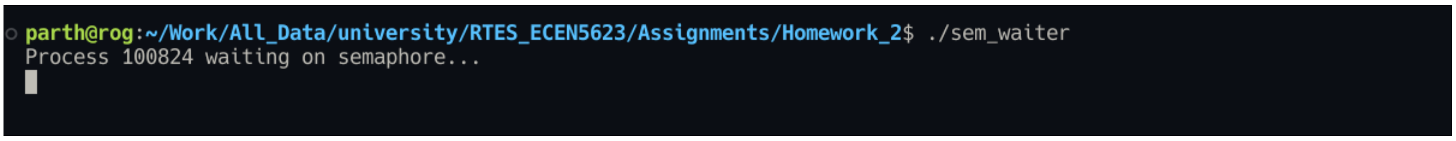
PDF is clickable

1 Question 1

Q: 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.

Answer: I wrote a C program named sem_wait that waits for a semaphore to be signaled by another program. Additionally, I wrote another C program called sem_post that signals this semaphore. Below are the steps and screenshots demonstrating the waiting program until the semaphore is signaled.
here are steps for to run semaphore programs

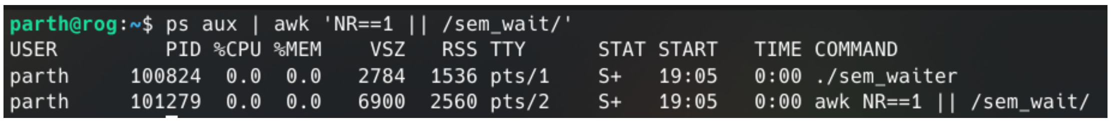
1. First, I executed the sem_wait program. This program will wait for a semaphore signal. A screenshot confirms the program is in a wait state.



```
parth@rog:~/Work/All_Data/university/RTES_ECEN5623/Assignments/Homework_2$ ./sem_waiter
Process 100824 waiting on semaphore...
```

Figure 1: Program is running and is in wait state

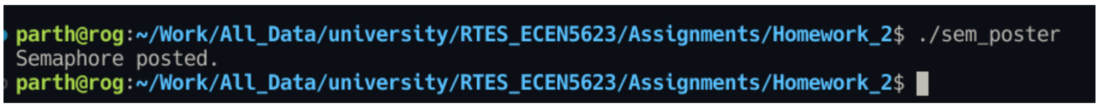
2. Using the ps command, I checked that the sem_wait program is indeed running and has not completed its execution. This is shown in the screenshot below.



```
parth@rog:~$ ps aux | awk 'NR==1 || /sem_wait/'
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
parth     100824  0.0  0.0   2784  1536 pts/1    S+   19:05   0:00 ./sem_waiter
parth     101279  0.0  0.0   6900  2560 pts/2    S+   19:05   0:00 awk NR==1 || /sem_wait/
```

Figure 2: PS command that shows that our program is waiting for semaphore

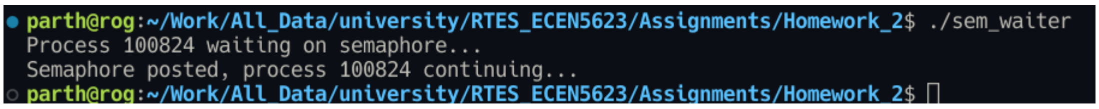
3. Next, I ran the sem_post program, which signals the semaphore for the waiting program.



```
parth@rog:~/Work/All_Data/university/RTES_ECEN5623/Assignments/Homework_2$ ./sem_poster
Semaphore posted.
parth@rog:~/Work/All_Data/university/RTES_ECEN5623/Assignments/Homework_2$
```

Figure 3: Semaphore has been posted by this program

4. After the semaphore was signaled, the sem_wait program completed its execution. This is evident from the following screenshot.

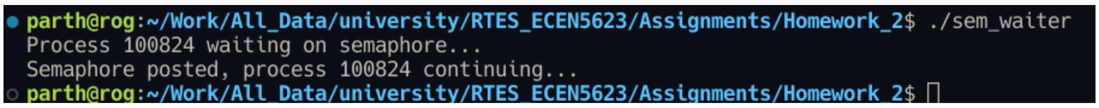


```
parth@rog:~/Work/All_Data/university/RTES_ECEN5623/Assignments/Homework_2$ ./sem_waiter
Process 100824 waiting on semaphore...
Semaphore posted, process 100824 continuing...
parth@rog:~/Work/All_Data/university/RTES_ECEN5623/Assignments/Homework_2$
```

Figure 4: Output of program waiting for semaphore

5. Lastly, I used the ps command again to verify that the waiting program had indeed completed after the semaphore was posted. The screenshot intended to show this step appears to be repeated from step 4. It should ideally confirm the waiting program's completion.

Through these steps and screenshots, it's demonstrated how one program can wait for a semaphore to be signaled by another, showcasing basic inter-process communication and synchronization in Linux.



```
● parth@rog:~/Work/All_Data/university/RTES_ECEN5623/Assignments/Homework_2$ ./sem_waiter
  Process 100824 waiting on semaphore...
  Semaphore posted, process 100824 continuing...
○ parth@rog:~/Work/All_Data/university/RTES_ECEN5623/Assignments/Homework_2$ ▯
```

Figure 5: PS output after Semaphore posted

2 Question 2

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

1. **Sensitivity to Task Overruns:** In EDF scheduling, if a task overruns its expected execution time, it can lead to a domino effect where subsequent tasks also miss their deadlines. This is because EDF always prioritizes the task with the closest deadline, meaning an overrunning task continues to preempt others, potentially leading all tasks in the queue to miss their deadlines.

2. **Complexity in Handling New Tasks:** Adding a new task to the queue requires adjusting priorities across all tasks. If there's an overrunning task, it won't be preempted by newly added tasks, even if they have sooner deadlines, because the overrunning task has a 'negative' time to its deadline, keeping it at the highest priority.

3. **Difficulty in Overrun Detection and Management:** Managing task overruns in EDF requires detecting the overrun and potentially terminating the overrunning task. This process itself consumes CPU resources, which can further compromise the system's ability to meet deadlines, especially without any margin for error.

4. **Challenges in Predicting Affected Services:** In overload scenarios, while EDF's adjustment to focus on tasks with the soonest deadlines seems logical, predicting which tasks will fail to meet their deadlines due to dynamic releases and priority adjustments is difficult.

EDF Failure in Overload Situations In an overload scenario, EDF can lead to a cascading failure where multiple services miss their deadlines because of a single overrunning service. This failure mode is more severe in EDF compared to RM or DM because RM/DM schedules are static and prioritize tasks based on their fixed rates or deadlines, not dynamically adjusting to the current load or task overruns. This makes RM/DM potentially more predictable and stable under overload conditions, even if they might not utilize CPU resources as efficiently as EDF under normal conditions.

The book by sam siewert suggests that, in the face of an overload, the best course of action for an EDF scheduler might be to drop all services in the queue, aiming for a more deterministic outcome. While this approach ensures predictability, it also indicates a significant limitation of EDF in handling overloads effectively without compromising system reliability.

In summary, while EDF can theoretically achieve 100% CPU utilization and meet deadlines under ideal conditions, its sensitivity to task overruns, complexity in handling dynamic priority adjustments, and challenges in managing overload situations make it less desirable for hard real-time systems compared to RM or DM. RM and DM's predictability and stability, even at the cost of potentially lower CPU utilization, are often more critical in environments where meeting every deadline is paramount.

3 Question 3

Q: 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 1 GHz processor core, what is the CPI required for this system? What is the overlap between instructions and IO time if the intermediate IO time is 4.5 microseconds?
Answer: The total number of instructions processed per second is the product of the frame processing rate and the instruction count per frame. The CPI (Cycles Per Instruction) can then be determined by dividing the total number of cycles available per second by the total number of instructions processed per second.

$$Instructions\_per\_second = Frame\_processing\_rate \cdot Instruction\_per\_frame$$
$$Instructions\_per\_second = 10000 \cdot 2120$$
$$\boxed{Instructions\_per\_second = 21200000}$$

Now for CPI

$$CPI = \frac{Clock\_speed}{Instructions\_per\_second}$$
$$CPI = \frac{1 \cdot 10^9}{21200000}$$
$$\boxed{CPI = 4.71698 \quad C/I}$$

Overlap Calculation The overlap between instructions and I/O time can be considered as the time during which the processor can perform other tasks while waiting for I/O operations to complete. Given the intermediate I/O time, we can calculate the amount of overlap by considering the total time available for processing each frame and the I/O time.
The instruction execution time would be

$$Instruction\_execution\_time = \frac{Instruction\_per\_frame \cdot CPI}{Processor\_clock}$$
$$Instruction\_execution\_time = \frac{2120 \cdot 4.71698}{10^9}$$
$$\boxed{Instruction\_execution\_time = 1 \cdot 10^{-5} \quad sec}$$

And overlapping percentage would be

$$Overlap\_percentage = \frac{IO\_time}{Instruction\_execution\_time} \cdot 100$$
$$Overlap\_percentage = \frac{4.5 \cdot 10^{-6}}{1 \cdot 10^{-5}} \cdot 100$$
$$Overlap\_percentage = 0.45 \cdot 100$$
$$\boxed{Overlap\_percentage = 45\%}$$

4 Referance

1. REAL-TIME EMBEDDED COMPONENTS AND SYSTEMS with LINUX and RTOS by Sam Siewert John Pratt

2. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment C. L. LIU AND JAMES W. LAYLAND

3. https://bears.ece.ucsb.edu/class/ece253/lect7.pdf

Appendices

A C Code for the Implementation

sem_waiter.c

```c
/*******************************************************************************
 * Copyright (C) 2023 by Parth Thakkar
 *
 * Redistribution, modification or use of this software in source or binary
 * forms is permitted as long as the files maintain this copyright. Users are
 * permitted to modify this and use it to learn about the field of embedded
 * software. Parth Thakkar and the University of Colorado are not liable for
 * any misuse of this material.
 * ****************************************************************************/

/**
 * @file    sem_waiter.c
 * @brief   This program demonstrates the use of POSIX semaphores for inter-process
 *          communication and synchronization. It waits on a semaphore until it's
 *          posted by another process, allowing for coordinated execution.
 *
 *          This could be used in scenarios where it's necessary to ensure that
 *          certain resources are not accessed by multiple processes simultaneously
 *          or to synchronize the execution order of processes.
 *
 *
 * @author  Parth Thakkar
 *
 */

#include <stdio.h>
#include <fcntl.h>    // For O_* constants
#include <sys/stat.h> // For mode constants
#include <semaphore.h>
#include <stdlib.h>
#include <unistd.h>

#define SEM_NAME "/semaphore_custom"

int main()
{
    printf("Process with PID %d waiting on semaphore...\n", getpid());

    // Open or create the semaphore
    /*
     * 0644 These are the permissions for the new semaphore if it is created. This is a octal number
     * representing the semaphore's permissions in a Unix/Linux environment. The first digit is always
     * zero, the second digit represents permissions for the owner (read and write), the third digit
     * represents permissions for the owner's group (read), and the fourth digit represents permissions for
     * others (read). This means the owner can read and modify the semaphore, while others can only read
     * its value.
     *
     * O_CREAT: This flag indicates that the semaphore should be created if it does not already exist.
     * sem_open can take multiple flags, combined using the bitwise OR operator (|), but in this case, only
     * O_CREAT is used. Other flags could include O_EXCL, which, when used with O_CREAT, will make sem_open
     * fail if the semaphore already exists, ensuring that the semaphore is newly created.
     *
     * 0: This is the initial value for the semaphore if it is being created. In this case, the semaphore is initialized to 0. This
     * value can be used to control access to a resource by having threads or processes wait until the semaphore's value is greater
     * than zero.
     */
    sem_t *sem = sem_open(SEM_NAME, O_CREAT, 0644, 0);
    if (sem == SEM_FAILED)
    {
        perror("sem_open failed");
        exit(EXIT_FAILURE);
    }

    // Wait on the semaphore
    if (sem_wait(sem) < 0)
    {
        perror("sem_wait failed");
        exit(EXIT_FAILURE);
    }

    printf("Semaphore posted, process %d continuing...\n", getpid());
    // Close the semaphore to release resources. This does not remove the semaphore,
    // but it detaches it from the process that called sem_close.
    sem_close(sem);

    // Unlink the semaphore, removing its name from the system. This is necessary
    // to clean up and ensure that the semaphore does not persist after all
    // processes using it have terminated.
    sem_unlink(SEM_NAME);

    return EXIT_SUCCESS;
}
```

sem_poster.c

```c
/****************************************************************************
 * Copyright (C) 2023 by Parth Thakkar
 *
 * Redistribution, modification or use of this software in source or binary
 * forms is permitted as long as the files maintain this copyright. Users are
 * permitted to modify this and use it to learn about the field of embedded
 * software. Parth Thakkar and the University of Colorado are not liable for
 * any misuse of this material.
 * ***************************************************************************/

/**
 * @file    sem_poster.c
 * @brief   This program demonstrates posting (incrementing) a POSIX semaphore.
 *          It is typically used in conjunction with another process that waits
 *          on this semaphore. The posting operation signals that an event has
 *          occurred or a resource is available, allowing the waiting process
 *          to proceed.
 *
 *
 * @author  Parth Thakkar
 *
 */

#include <stdio.h>
#include <fcntl.h>    // For O_* constants
#include <sys/stat.h> // For mode constants
#include <semaphore.h>
#include <stdlib.h>

#define SEM_NAME "/semaphore_custom" // Name of the semaphore to post

int main()
{
    // Open the semaphore that has been created by other file
    sem_t *sem = sem_open(SEM_NAME, 0);
    if (sem == SEM_FAILED)
    {
        perror("sem_open failed");
        exit(EXIT_FAILURE);
    }

    // Post the semaphore
    if (sem_post(sem) < 0)
    {
        perror("sem_post failed");
        exit(EXIT_FAILURE);
    }

    printf("Semaphore posted.\n");

    // Close the semaphore to release resources. This operation does not remove
    // the semaphore but detaches it from the process that called sem_close.
    sem_close(sem);

    return EXIT_SUCCESS;
}
```