

Question 2.5. 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.

This implementation involves two separate user-level applications: one acting as a semaphore waiter (Waiter) and the other as a semaphore signaler (Signaler). Both applications are executed with the default priority assigned to user-level processes in Linux.

### 1. Semaphore Waiter (Process Waiter):

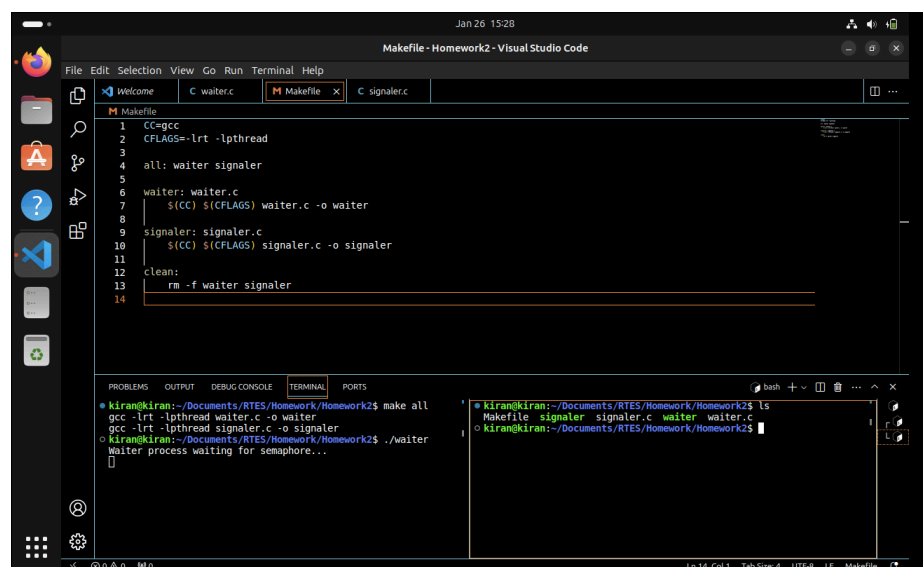
- This process initializes and waits on a binary semaphore. It remains in a blocked state until the semaphore is signalled by another process.
- Figure 1 illustrates the Waiter process in its waiting state. During this phase, the process is inactive, conserving CPU resources while awaiting the semaphore release.

### 2. Semaphore Signaler (Process Signaler):

- Upon execution, this process signals the binary semaphore that the Waiter process is waiting on.
- The moment the Signaler process is instantiated and executes the signal operation, it effectively releases the semaphore.

### 3. Interaction and Conclusion:

- As soon as the semaphore is signalled by the Signaler, the Waiter process receives the semaphore and transitions from a blocked to a running state.
- Subsequently, as shown in Figure 2, the Waiter process completes its execution and exits gracefully.



```
Jan 26 15:28
Makefile - Homework2 - Visual Studio Code

File Edit Selection View Go Run Terminal Help
C: waiter.c  Makefile  C: signaler.c

1 CC=gcc
2 CFLAGS=-lrt -lpthread
3
4 all: waiter signaler
5
6 waiter: waiter.c
7 $(CC) $(CFLAGS) waiter.c -o waiter
8
9 signaler: signaler.c
10 $(CC) $(CFLAGS) signaler.c -o signaler
11
12 clean:
13 rm -f waiter signaler
14

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
• kirankiran~/Documents/RTES/Homework/Homework2$ make all
gcc -lrt -lpthread waiter.c -o waiter
gcc -lrt -lpthread signaler.c -o signaler
• kirankiran~/Documents/RTES/Homework/Homework2$ ./waiter
Waiter process waiting for semaphore...
```

Figure 1 : Waiter Process for Waiting for Semaphore

The screenshot shows the Visual Studio Code editor with a Makefile and a terminal window. The Makefile contains the following rules:

```

1 CC=gcc
2 CFLAGS=-lrt -lpthread
3
4 all: waiter signaler
5
6 waiter: waiter.c
7     $(CC) $(CFLAGS) waiter.c -o waiter
8
9 signaler: signaler.c
10    $(CC) $(CFLAGS) signaler.c -o signaler
11
12 clean:
13     rm -f waiter signaler
14

```

The terminal window shows the following commands and output:

```

kiran@kiran:~/Documents/RTES/Homework/Homework2$ make all
gcc -lrt -lpthread waiter.c -o waiter
gcc -lrt -lpthread signaler.c -o signaler
kiran@kiran:~/Documents/RTES/Homework/Homework2$ ./waiter
Waiter process waiting for semaphore...
Semaphore received. Exiting...
kiran@kiran:~/Documents/RTES/Homework/Homework2$
kiran@kiran:~/Documents/RTES/Homework/Homework2$ ls
Makefile signaler signaler.c waiter waiter.c
kiran@kiran:~/Documents/RTES/Homework/Homework2$ ./signaler
Signaler process posting semaphore...
Semaphore posted. Exiting...
kiran@kiran:~/Documents/RTES/Homework/Homework2$

```

Figure 2 : Waiter process exited once signaler application posted semaphore

#### 4. Before Signalling:

Initially, when you ran `ps aux | grep waiter`, we received two lines in the output as seen in figure 3:

```
kiran 8020 0.0 0.0 2664 1280 pts/1 S+ 15:31 0:00 ./waiter
```

```
kiran 8047 0.0 0.0 6608 2176 pts/2 S+ 15:31 0:00 grep --color=auto waiter
```

The first line represents your waiter process, and the second line represents the grep command searching for the waiter process.

The screenshot shows the Visual Studio Code editor with the same Makefile and a terminal window. The terminal window shows the following commands and output:

```

kiran@kiran:~/Documents/RTES/Homework/Homework2$ ./waiter
Waiter process waiting for semaphore...
kiran@kiran:~/Documents/RTES/Homework/Homework2$ ps aux | grep waiter
kiran 8020 0.0 0.0 2664 1280 pts/1 S+ 15:36 0:00 ./waiter
kiran 8047 0.0 0.0 6608 2176 pts/2 S+ 15:36 0:00 grep --color=auto waiter
kiran@kiran:~/Documents/RTES/Homework/Homework2$

```

Figure 3 : Process status before running signaler application

## 5. After Signalling and Exiting:

After you executed `./signaler`, the waiter process received the semaphore, completed its execution, and then exited. This is confirmed by the absence of the waiter process in the subsequent `'ps aux | grep waiter'` output:

```

Makefile
1 CC=gcc
2 CFLAGS=-lrt -lpthread
3
4 all: waiter signaler
5
6 waiter: waiter.c
7     $(CC) $(CFLAGS) waiter.c -o waiter
8
9 signaler: signaler.c
10    $(CC) $(CFLAGS) signaler.c -o signaler
11
12 clean:
13     rm -f waiter signaler
14
Terminal
kiran@kiran:~/Documents/RTES/Homework/Homework2$ ./waiter
Waiter process waiting for semaphore...
Semaphore received. Exiting...
kiran@kiran:~/Documents/RTES/Homework/Homework2$ ./signaler
Signaler process posting semaphore...
Semaphore posted. Exiting...
kiran@kiran:~/Documents/RTES/Homework/Homework2$ ps aux | grep waiter
kiran  8820  0.0  0.0  2664 1280 pts/1  S+  15:36  0:00  ./waiter
kiran  8847  0.0  0.0  6608 2048 pts/2  S+  15:36  0:00  grep --color=auto waiter
kiran@kiran:~/Documents/RTES/Homework/Homework2$

```

Figure 4 : Process status after running signaler application

In this output as seen in figure 4, we only see the line for the `grep` command itself. The waiter process (previously with PID 8020) is no longer listed, indicating that it has successfully terminated after being signalled.

The waiter process was initially in a sleeping state, waiting for the semaphore. After you ran the 'signaler' program, it signalled the semaphore, allowing the 'waiter' process to proceed. Once the 'waiter' completed its execution, it exited as expected. The 'ps' command output confirms this by no longer listing the waiter process, indicating that your semaphore signalling and process completion worked as intended.

Question 3.5 : 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?

Earliest Deadline First (EDF) is a dynamic scheduling algorithm that prioritizes tasks based on their deadlines. It's known for its theoretical efficiency in CPU resource utilization, where it can potentially handle 100% CPU load by prioritizing the task with the closest deadline. The primary reason Earliest Deadline First (EDF) is not typically the hard real-time policy of choice, despite its ability to meet deadlines and achieve potentially 100% CPU resource utilization, revolves around its dynamic nature and the complexity associated with it.

### **Drawbacks of Earliest Deadline First (EDF)**

#### *1. Dynamic Priority Management*

- Complexity: EDF's dynamic nature, where priorities are constantly adjusted based on task deadlines, introduces significant computational overhead. This complexity is a major drawback in hard real-time systems where minimal processing latency and predictability are crucial [1].
- Resource Intensiveness: In EDF, the system spends considerable resources recalculating priorities, which could otherwise be used for executing time-critical tasks.

#### *2. Predictability and System Analysis*

- Difficulty in Predictability: Hard real-time systems benefit from predictable behavior to ensure reliability. EDF, with its variable priorities, makes it more challenging to perform accurate worst-case execution time analyses.
- Analysis Complexity: The dynamic priority model of EDF complicates the analysis process, especially compared to fixed-priority models like RM/DM, which allow for more straightforward worst-case scenario planning [2].

### **Behaviour of EDF in Overload Situations**

#### *1. Task Prioritization Challenges*

- Inefficiency Under Load: When the system is overloaded with tasks, EDF prioritizes tasks with imminent deadlines. However, this can lead to scenarios where many tasks end up missing their deadlines as the system struggles to keep up with the most urgent tasks, neglecting others in the queue.
- Missed Deadlines: EDF's approach in overload situations can cause a cascade of missed deadlines, where the system becomes increasingly unable to recover and process queued tasks effectively.

#### *3. Comparison with Fixed-Priority Systems*

- Lack of Structured Overload Management: Unlike fixed-priority systems like RM and DM, EDF lacks a structured approach to managing tasks during high load periods. Fixed-priority systems can offer a more controlled handling of tasks by ensuring higher priority (and often more critical) tasks are more likely to be completed, even in overload scenarios.

## Conclusion

While EDF has the theoretical advantage of handling 100% CPU utilization and effectively managing task deadlines under optimal conditions, its drawbacks in terms of computational complexity and predictability, along with its less effective handling of overload scenarios, often make it a less favoured choice for hard real-time applications. The need for a predictable and analysable system in a hard real-time environment often outweighs the theoretical efficiency of EDF, favouring fixed-priority scheduling methods.

## References :

- [1] S. Siewert, "Real-Time Embedded Components and Systems with Linux and RTOS," 2nd ed. Mercury Learning & Information, 2016.
- [2] G. Buttazzo, "Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications," 3rd ed. Springer, 2011.

Question 4.3: 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?

**Required CPI Calculation :**

Assumptions:

- Processor Speed: 10 GHz.
- Instructions per frame: 2,120.
- Frame Processing Rate: 100,000 frames per second.

Calculations:

1. *Total Instructions per Second*: Calculated as the product of instructions per frame and frames per second.
  - Instructions per Second = 2,120 instructions/frame \* 100,000 frames/second
  - Instructions per Second = 212,000,000 instructions/second.
2. *Total Cycles per Second*: Equivalent to the clock frequency.
  - Total Cycles per Second = 10 GHz =  $10 \times 10^9$  cycles/second.
3. *Computing CPI*: The ratio of total cycles available to total instructions needed.
  - $CPI = \text{Total Cycles per Second} / \text{Instructions per Second}$
  - $CPI = 10 \times 10^9 / 212 \times 10^6$
  - $CPI \approx 4.71$  cycles per instruction.

The calculated CPI is approximately **4.71**, indicating the average number of clock cycles required for each instruction.

**Overlap Ratio Calculation:**

Given

- IO Time: 4.5 microseconds ( $\mu\text{s}$ )

Calculations:

1. *Instruction Completion Time (ICT)*: Determined by multiplying the CPI by the instructions per frame, then dividing by the clock frequency.
  - $ICT = (CPI * \text{Instructions per Frame}) / \text{Clock Frequency}$
  - $ICT = (4.71 * 2,120) / 10^9$
  - $ICT = 10^{-5}$  seconds.
2. *Deadline per Frame*: The inverse of the frame processing rate.
  - $\text{Deadline} = 1 / \text{Frame Rate}$
  - $\text{Deadline} = 1 / 10^5$

- Deadline =  $10^{-5}$  seconds.
3. *Overlap Ratio (OR)*: Calculated by one minus the ratio of the difference between the deadline and IO time to the ICT.
- $OR = 1 - (\text{Deadline} - \text{IO Time}) / \text{ICT}$
  - $OR = 1 - (10^{-5} \text{ sec} - 4.5 \times 10^{-6} \text{ sec}) / 10^{-5} \text{ sec}$
  - $OR = 0.45$
  - $OR (\text{in percent}) = 0.45 * 100 = 45\%$ .

The overlap ratio is approximately **45%**, indicating the percentage of the instruction completion time that overlaps with the IO time.

### Code :

#### Waiter.c :

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <semaphore.h>

#define SEM_NAME "/my_semaphore"

int main() {
    sem_t *sem = sem_open(SEM_NAME, O_CREAT, 0644, 0);
    if (sem == SEM_FAILED) {
        perror("sem_open");
        return 1;
    }

    printf("Waiter process waiting for semaphore...\n");
    if (sem_wait(sem) < 0) {
        perror("sem_wait");
        return 1;
    }

    printf("Semaphore received. Exiting...\n");
    sem_close(sem);
    return 0;
}
```

#### Signaler.c :

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <semaphore.h>

#define SEM_NAME "/my_semaphore"

int main() {
    sem_t *sem = sem_open(SEM_NAME, 0);
    if (sem == SEM_FAILED) {
        perror("sem_open");
        return 1;
    }
}
```

```
printf("Signaler process posting semaphore...\n");
if (sem_post(sem) < 0) {
    perror("sem_post");
    return 1;
}

printf("Semaphore posted. Exiting...\n");
sem_close(sem);
return 0;
}
```

### Makefile :

```
CC=gcc
CFLAGS=-lrt -lpthread

all: waiter signaler

waiter: waiter.c
    $(CC) $(CFLAGS) waiter.c -o waiter

signaler: signaler.c
    $(CC) $(CFLAGS) signaler.c -o signaler

clean:
    rm -f waiter signaler
```