RTES- HW2

1. 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.  
Ans:

A computer screen with text on it

Description automatically generated  
First the process1 is started which is waiting for binary semaphore which is not existed. Then process2 is allowed to post the semaphore which become available for process1 and able to complete further execution.

2. 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? process to continue execution and exit. Verify completion.

Ans:

* Because the slightest miscalculation of resources required can cause overuse of resources allotted for other tasks. This leads to overload situations, which intern results in non-deterministic failure.
* In RM we can easily identify that all services of lower priority than the service that is overrunning may miss deadlines, yet all services of higher priority are guaranteed not to be affected. But in EDF it is hard to predict exactly which and how many services will miss their deadlines in an overload.
* In an overload situation, the stalling service cause all services in ready queue to miss their deadlines. Even if a new task is added to the ready queue, recalculation of priority will take place and result in assigning highest priority to stalling service, which will not prevent/preempt it from running further. So other services in ready queue will fail to meet deadlines. In this kind of scenarios, terminating the overload service is best idea, but it requires CPU resources for some time, which intern anyway leads to failure of next few services in the ready queue. Afterwards it is good to execute further.

3. 4.2 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?

Ans:

CPI = Total cycles per second / Instructions per second.

Total instruction counts per second = number of frames completed \* number of instructions in each frame

T = 100000 \* 2120 = 212000000 instructions/second.

Therefore, CPI = 1G / 212000000 = **4.716 cycles per instruction**

From textbook, OR + NOA = 1, where OR is Overlap required and NOA is non-overlap allowable.

Here we must find OR, so OR = 1 – NOA

NOA = (time to complete each frame – time consumed by IO)/ time to complete each frame = ((1/100000) – 4.5u)/ (1/100000) = (10u – 4.5u)/10u = 0.55.

Finally, OR = (1 – 0.55) \* 100 = **0.45%  
  
APPENDIX:**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 \* Copyright (C) 2023 by Jithendra

 \*

 \* 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. Jithendra and the University of Colorado are not liable for

 \* any misuse of this material.

 \* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

 \* @file process1.c

 \* @brief This program demonstrates the usage of a binary semaphore for synchronization.

 \* @author Jithendra H S

 \* @date 02-25-2024

 \*/

#include <stdio.h>

#include <semaphore.h>

#include <fcntl.h>

#include <syslog.h>

#define BINARY\_SEMAPHORE ("bin\_sem")

int main() {

    // Open or create a binary semaphore with initial value 0

    sem\_t \*binary\_semaphore = sem\_open(BINARY\_SEMAPHORE, O\_CREAT, 0777, 0);

    // Display a message indicating that Process 1 has started and is waiting for the binary semaphore

    printf("Process 1 started and waiting for binary\_semaphore\n");

    syslog(LOG\_INFO, "Process 1 started and waiting for binary\_semaphore");

    // Wait for the binary semaphore

    sem\_wait(binary\_semaphore);

    // Display a message indicating that Process 1 has completed after receiving the binary semaphore

    printf("Process 1 completed after receiving binary\_semaphore\n");

    syslog(LOG\_INFO, "Process 1 completed after receiving binary\_semaphore");

   sem\_close(binary\_semaphore);

    return 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 \* Copyright (C) 2023 by Jithendra

 \*

 \* 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. Jithendra and the University of Colorado are not liable for

 \* any misuse of this material.

 \* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

 \* @file process2.c

 \* @brief This program releases a binary semaphore for synchronization.

 \* @author Jithendra H S

 \* @date  02-25-2024

 \*/

#include <stdio.h>

#include <semaphore.h>

#include <fcntl.h>

#include <syslog.h>

#define BINARY\_SEMAPHORE ("bin\_sem")

int main() {

    // Open or create the binary semaphore

    sem\_t \*binary\_semaphore = sem\_open(BINARY\_SEMAPHORE, O\_CREAT, 0777, 0);

    // Display a message indicating that Process 2 is releasing the binary semaphore

    printf("Process 2 releasing the binary\_semaphore\n");

    syslog(LOG\_DEBUG, "Process 2 started and waiting for binary\_semaphore");

    // Release the binary semaphore

    sem\_post(binary\_semaphore);

    return 0;

}