Operating System Lab Assignment 04: Design and Deploy Another Process Scheduling (FCFS, or Priority), Performance Testing,Synchronization and Deadlock detection

Dr. Mosaddek Tushar, Professor
Computer Science and Engineering, University of Dhaka,
Version 1.0
Demo Due Date: The following weeks as separate parts.

October 17, 2023

Contents

1 Objectives and Policies

1.1 General Objectives

The objectives of the lab assignment are to understand and have hands-on training to understand operating system component design and implementation.

1.2 Assessment Policy

The assignment has three level objectives: (i) primary objectives, (ii) advanced objectives, and (iii) optional boost objectives. Every student must complete the primary objective; however, they

can attempt advanced and optional Boost-up objectives. The advanced objective will be a primary objective in the subsequent assignment. Further, you can achieve five (5) marks for completing the optional objectives and add these marks to your total lab marks at the end of the semester. However, you can get up to 100

2 What to do?

- Deploy OS Process Scheduling other than round-robin
- Deploy semaphore for synchronization and deadlock
- Compare the performance of round robin, FCFS/Priority for average response time, turn around time, and waiting time.

2.1 Minimum requirement for this assignment

The following description envisions implementing SVC and PendSV to enable unprivileged user applications to access kernel services and schedule user tasks. You obviously need the solution of assignment -03 to start assignment 04.

3 Detail description – What to do

This section describes the detail of the activities of the assignment -04. You must implement and test each section of the assignment before the submission. It would be best if you prepared an interactive presentation and demonstration of the design and implementation for the demonstration. Carefully design and implement your design to reduce the development time.

3.1 Deploy OS Process Scheduling other than round-robin

I hope you have completed assignment -03 for system call and task scheduling for round-robin scheduling. To keep the accounting information like start, response, execution, etc, you need to modify the task structure as follows:

```
struct t_task_tcb{
        uint32_t magic_number; //here it is 0xFECABAA0
        uint16_t task_id; //a unsigned 16 bit integer starting from 1000
        void *psp; //task stack pointer or stackframe address
        uint16_t status; //task status: running, waiting, ready, killed, or terminated
        uint32_t start_time_t; //process creation time
        uint32_t reponse_time_t; //first time CPU allocation (execution) time
        uint32_t execution_time_t; //total execution time (in ms)
        uint32_t waiting_time_t//total waiting time (in ms)
        uint32_t priority; //task priority
        uint32_t digital_sinature; //current value is 0x00000001
} TCB_TypeDef
```

3.1.1 FCFS/Priority Scheduling and performance comparison

The FCFS scheduling algorithm/policy picks a task from the ready queue and starts execution until it terminates. The priority scheduling algorithm assigns non-zero positive priority to each of the processes. In this case, use the priority value to the 'uint32_t priority' in the TCB. The scheduling algorithm updates the accounting information given in the above TCB structure. At the end of all processing, the developed system displays the statistics of the process as

$< ProcessId >< starttime >< responsetime >< waitingtime >< turnaroundtime >$

for each of the process in line. At the end display summarize statistics

$< Scheuling\ Policy >\ \ < average\ response\ time >\ \ < average\ waiting\ time >\ \ < average\ turnaround\ time >$

for each of the scheduling policy . See the following example

Process Statistics

Process Id	Start time	Response time	Waiting time	Execution Time	Turn around time
10023	20ms	23ms	234ms	216ms	450ms
10024	20ms	33ms	264ms	275ms	539ms
–	–	–	–	–	–

Scheduling Policy Performance

Scheduling Policy	Av, Response	Avg. Waiting time	Avg. Execution Time	Avg Turn around time
Round-Robin	35ms	220ms	216ms	600ms
FCFS	50ms	300ms	275ms	1000ms
–	–	–	–	–

3.2 Deploy semaphore for synchronization and deadlock

Assignment 03 displays a few error texts for updating the 'count' variables. Moreover, the 'kprintf' has synchronization issues. You have to implement a semaphore and use it to synchronize all resource uses. First, find out the segment of codes that need synchronization; second, design functions related to semaphore wait and signal; third, use them for data consistency. The semaphore must have a queue, such as a waiting queue, that stores one or more blocking processes out of the ready queue whenever the related resources are not available, or a task is in the critical region. You may have multiple critical regions. Be careful when using semaphore; you may be in a deadlock situation. You will get an extra five marks for demonstrating the deadlock and writing code to break it. Display a set of processes responsible for deadlock. Your deadlock detection and recovery from deadlock must fulfill the four features discussed in the class.

However, the compulsory part of calculating the overhead in the scheduling policies' performance is using a semaphore for synchronization. The best way to demonstrate the percent of overhead incurred on the performance is illustrated in the previous sections.

3.2.1 Semaphore implementation in ARM example

: You should consult the ARM development manual. Your implementation may be different from the one below. It is up to you to implement the signal and wait module properly. The following are a few examples only to guide you. Present your development criterion, design, and implementation. Your code must maintain the system throughput and parallelism. Add 'sem.h' and 'sem.c' in the

'kern/lib' and 'kern/include' directories; these files should have all the data structure and code to implement semaphores.

Semaphore decrement function:

```
; sem_dec
; Declare for use from C as extern void sem_dec(void * semaphore);
EXPORT sem_dec
sem_dec PROC
1   LDREX   r1, [r0]
CMP             r1, #0        ; Test if semaphore holds the value 0
BEQ     %f2           ; If it does, block before retrying
SUB     r1, #1        ; If not, decrement temporary copy
STREX   r2, r1, [r0]  ; Attempt Store-Exclusive
CMP     r2, #0        ; Check if Store-Exclusive succeeded
BNE     %b1           ; If Store-Exclusive failed, retry from start
DMB                   ; Required before accessing protected resource
BX      lr
```

Semaphore increment function:

```
; sem_inc
; Declare for use from C as extern void sem_inc(void * semaphore);
EXPORT sem_inc
sem_inc PROC
1   LDREX   r1, [r0]
ADD     r1, #1        ; Increment temporary copy
STREX   r2, r1, [r0]  ; Attempt Store-Exclusive
CMP     r2, #0        ; Check if Store-Exclusive succeeded
BNE     %b1           ; Store failed - retry immediately
CMP         r0, #1        ; Store successful - test if incremented from zero
DMB                   ; Required before releasing protected resource
BGE         %f2           ; If initial value was 0, signal update
BX      lr

2   ; Signal waiting processors or processes
SIGNAL_UPDATE
BX      lr
```

Semaphore add task to semaphore waiting queue

```
extern void sem_inc(void * semaphore);
extern void sem_dec(void * semaphore);

unsigned int task_semaphore = 0;

void add_task(TCB_TypeDef * task)
{
        /* Add task to queue */
        ...
```

```
        /* Increment semaphore to show task has been added*/
        sem_inc(&task_semaphore);

        return;
}
```

Semaphore : get/delete task to semaphore waiting queue

```
struct task * void get_task(void)
{
        TCB_TypeDef * tmptask;

        /* Decrement semaphore, or block until it indicates a task is available */
        sem_dec(&task_semaphore);

        /* Take task from queue */
        ...

        return tmptask;
}
```

Wait for update function – low power

```
 MACRO
WAIT_FOR_UPDATE
WFI                     ; Indicate opportunity to enter low-power state
MEND
```

Signal update function – low power

```
MACRO
SIGNAL_UPDATE           ; No software signalling operation
MEND
```

4 What to submit

Submit the source code, including 'sem.h' and 'sem.c', and the presentation on Google Classroom. Before submitting, run 'make clean' to reduce the source code size. The presentation must be in pdf format. I advise you to use a latex beamer for the presentation. (i) performance display before and after semaphore implementation; (ii) implement at least one scheduling policy (FCFS/Priority) along with round-robin scheduling. You should read the assignment carefully and submit it as discussed in this document.

Alert: You can discuss with your classmates, however, do not copy code from others.