Carleton University
Department of Systems and Computer Engineering
SYSC 4001 Operating Systems Fall 2024

Assignment 2 – Process Scheduling, Memory Management

THIS ASSIGNMENT MUST BE DONE BY TEAMS OF TWO STUDENTS in the same lab section.

Please submit the assignment using the electronic submission on Brightspace. The submission process will be closed at the deadline: November 1st, 2024 at midnight. No assignments will be accepted via email.

YOU ARE REQUIRED TO FORM GROUPS ON BRIGHTSPACE BEFORE THE SUBMISSION OF THE ASSIGNMENT. To form groups go to "Tools" > "Groups". Scroll through the list of group options available. The group would be "<Lab section> - Assignment 2". The "Lab Section" will be your respective to the section.

Part I – Concepts [3 marks]

Answer the following questions. Justify your answers. Show all your work.

a) [1 mark] Consider the following set of processes. Each process has a single CPU burst and does not perform any I/O.

Process	Arrival Time (sec)	Execution Time (sec)
P1	0	13
P2	6	4
P3	15	11
P4	21	9
P5	24	13

With the help of Gantt charts, compute the mean turnaround time for the following scheduling algorithms:
(i) FCFS (First Come First Serve) (ii) Round Robin (time slice of 4 sec).
(iii) Shortest Job First with preemption
(iv) Multiple queues with feedback (high priority queue: quantum = 2; mid-priority queue: quantum = 3; low priority queue: FIFO)

b) [1 mark] Now assume that each process in part a) requests to do an I/O every 1 sec., and the duration of each of these I/O is 1 sec. Create new Gantt diagrams considering the I/O operations and repeat all the parts done in part a) using this new input trace.

c) Memory management [1 mark]

Consider a multiprogrammed system that uses multiple partitions (of variable size) for memory management. A linked list of holes called the free list is maintained by the operating system to keep track of the available memory in the system. At a given point in time the free list consists of holes with sizes: 102K, 205K, 43K, 180K, 70K, 125K, 91K, and 150K

The free list is also ordered in the sequence given above: the first hole in the list is of size 102K words, which is followed by a hole of size 205K words and so on. There are a number of Jobs arriving to the system with different memory requirements; they arrive in the following order:

Job No.	Arrival Time	Memory Requirement (words)
1	t1	122K
2	t2	105K
3	t3	203K
4	t4	90K

[Given $t1 < t2 < t3 < t4$]

Determine which free partition will be allocated to each process for the following algorithms:
(i) First Fit (ii) Best Fit (iii) Worst Fit
Show all your work. Based on the calculations and results obtained from these algorithms, analyze which scheduling algorithm and memory management strategy is the most efficient according to different metrics. Justify your answer.

Part II – Concurrent Processes in Unix [2 marks]

1. Run two concurrent processes in C/C++ under Linux

Using the fork system call (page 472 of the Linux book), create two independent processes, which run indefinitely. Process 1 will run forever and will display, on screen "I am Process 1". Process 2 will display, on screen "I am Process 2". Use delay functions to slower the display speed.

To finish the program, use the kill command (man pages), find the pid of both processes (ps) and kill them.

2. Extend the Processes. Now it will display the same message on screen as in 1., and will generate a random number between 0 and 10 using the rand() function in C/C++. If the value is larger than 5, it will now display "High value". If it is equal or lower than 5, it will display "Low value". Process 2 will start only if a random value equal to 9 is generated by Process 1. Process 2 will display "I am process 2" in an infinite loop. Use the exec system call to launch Process 2 (i.e., Process 2 should be a different program/executable).

Use delay functions to slower the display speed.

To finish the program, use the kill command (man pages), find the pid of both processes (ps) and kill them.

3. Extend the processes above once more. Use the wait system call. Process 1 starts as in 2, and when Process 2 starts, it waits for it. Process 2 displays the message 10 times and exits. When this happens, Process 1 should end too.

4. Extend the processes above once more. They should now share memory. The primary functions are listed in the book and course materials. Using shmget, shmctl, shmat, and shmdt, add a common variable shared between the two processes. The variable contains the random number generated. Process 2 starts only when the random value is 9. Each of the processes should now react to the value of the shared variable, and display a message identifying themselves and the random number in shared memory. Both processes finish when the value generated is 0.

5. Extend the processes above once more. They should now protect concurrent access to the shared memory position. On top of the shm instructions, you should protect the shared memory access using semaphores. Use semget, semop, semctl to protect the shared memory section.

As before, you now have a common variable shared between the two processes, and it is protected from concurrent access using semaphores. The behavior is as in 4.

Information about message passing is on Chapter 14 of the Linux reference book, and in the online materials posted.

Marks to be obtained: 0.4 marks for each correct part.

Part III - Design and Implementation of an API Simulator: fork/exec [5 marks]

The objective of this assignment is to build a small API simulator.

We will reuse Assignment 1, where we simulated an interrupt system.

The system has one CPU. The OS use fixed partitions in memory. The simulator uses the following data structures:

i. Memory Partitions: fixed partitions

You have a simulated space of 100 Mb of user space available, divided in six fixed partitions:

1- 40 Mb
2- 25 Mb
3- 15 Mb
4- 10 Mb
5- 8 Mb
6- 2 Mb

You need to define a table (array of structs, linked list) with the following structure:

Partition Number	Size	Code
Unsigned int	Unsigned int	String. Only use free, init or the program name (this will be clear as you read through the assignment)

ii. PCB:

Using the textbook/slides, you need to define a table (array of structs, linked list) with similar contents of those found in a PCB (only include the information needed: PID, CPU, and I/O information, remaining CPU time – needed to represent preemption –, partition number where the process is located). You should add any other information that your simulator needs.

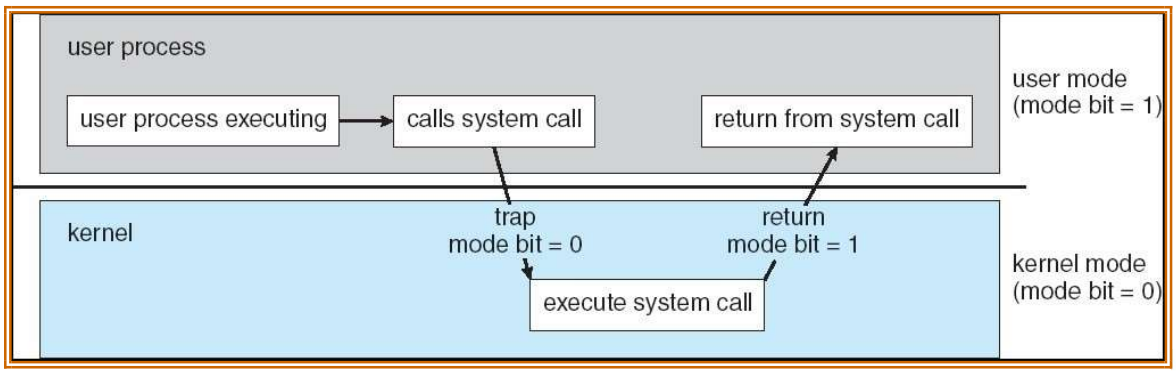
Your simulator will initialize the PCB table and pid 0, called init, which will use Partition 6; it uses 1 Mb of memory.

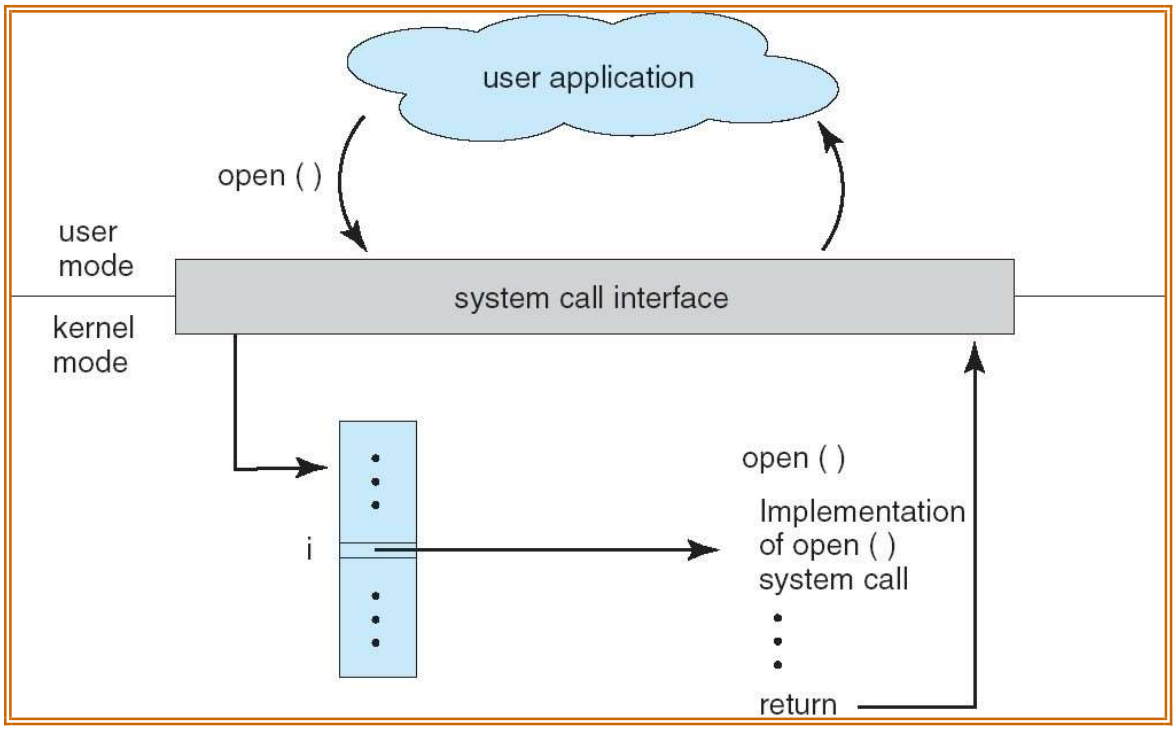
iii. List of External Files

This is another table (implemented as an array of structs, linked list, or your choice of data structure for the table). Its initial contents are loaded from an external text file (*external_files.txt*) at the start of the simulation. It simulates your persistent memory (hard drive, USB, flash drive…). It only contains a list of programs as follows:

Program Name	Size of the program when loaded in memory
String; 20 chars	Unsigned Integer

The simulator will implement simulated versions of the fork, exec system calls. For all the system calls, the simulation should try to reproduce the behavior of this state diagram (from Silberschatz et al.) which was implemented in Assignment 1, for both system calls and interrupts.





In this simulator, we implement the two system calls as follows:

a. fork ()

a. Simulates SYSCALL (Assignment 1)
b. The ISR copies the PCB of the parent to the child process
c. At the end of the ISR, you call the routine scheduler (), which is, for now, empty (it just displays "scheduler called").
d. Return from the ISR (Assignment 1)

b. exec (file_name)

a. Simulates SYSCALL (Assignment 1)
b. The ISR uses the size of the new executable; should search the file in the file list, and obtain the memory size

c. It finds an empty partition where the executable fits (Assume best-fit policy)
d. It marks the partition as occupied
e. It records which process is using that partition
f. It updates the PCB with the new information
g. At the end of the ISR, you call the routine scheduler (), which is, for now, empty (it just displays "scheduler called").
h. Return from the ISR (Assignment 1)

All the activities by fork and exec must be logged by the simulator. Each step is associated with a random execution time between 1 and 10 milliseconds.

The simulator checks the Ready Queue (in the PCB table you defined), grabs the first process available, and simulates its execution. How? By reading the program name from the *external_files* table and loading the appropriate program into the simulator. Other than that, you execute the fork, exec and exit syscall as above.

Assumptions to be made:

- When the EXEC system call is invoked, assume it is only called by the child process and not by the parent.
- The child process has a higher priority than the parent process. Therefore, the child process should be executed until completion before resuming the parent process. This means that the simulator does not account for orphaned processes yet; the child must finish execution before the parent process can terminate.
- EXEC is found at vector 3 and for FORK is found at vector 2

At the end of each simulated system call, the simulator saves, in a file called system_status.txt the following information:

. Current simulated time
. PCB table

<u>Mandatory Test Scenarios</u>

1. Only Init is in the system. Init runs this code:

FORK, 10
EXEC program1, 50
FORK, 15
EXEC program2, 25

Contents of program1:
CPU, 100

Contents of program2:
SYSCALL 4, 125

2. Only Init is in the system. Init runs this code:
FORK, 17
EXEC program1, 16

Contents of program1:

FORK, 15
EXEC program2, 33

Contents of program2:
CPU, 53
SYCALL 5, 128
END_IO 11, 115

3. Only Init is in the system. Init runs this code:
FORK, 20
EXEC program1, 60

Program 1 runs this code:
FORK, 17
EXEC program2, 46

Program 2 runs this code
FORK, 16
EXEC program3, 58

Program 3 runs this code:
CPU, 50
SYSCALL 6, 110
CPU, 15
END_IO 10, 220

Your Test Scenarios

After these three tests, you should create at least 2 other tests to be submitted.

The input to your program:

- List of external files in *external_files.txt*
- Initial trace file in *trace.txt*
- Your vector table in *vector_table.txt*
- This is not an input, but you need the traces of your external programs accessible by your simulator. The simulator will look at the respective program name from the EXEC call, check memory requirements from the external_files table and fetch the corresponding program.

Test cases and an example document will be posted on Brightspace, as in Assignment 1.