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

Assignment 1

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

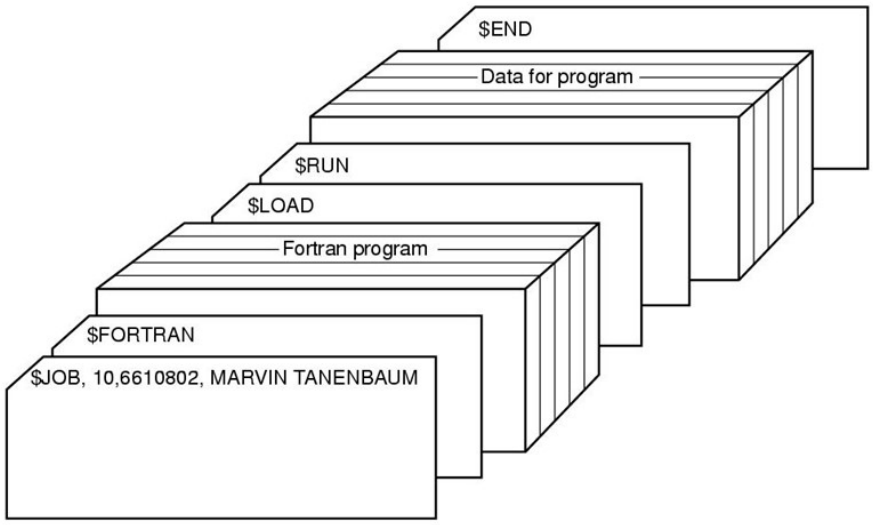
Please submit the assignment using the electronic submission on Brightspace. The submission process will be closed at the deadline: Friday, October 4th, 2024 at midnight. No assignments will be accepted via email.

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 1". The "Lab Section" will be your respective to the section.

Part I – Concepts [4.5 marks]

Answer the following questions (marks in brackets)

a) [1 mark] Batch Operating Systems used special cards to automate processing and to identify the jobs to be done. A new job started by using a special card that contained a command, starting with $, like:



For instance, the $FORTRAN card would indicate to start executing the FORTRAN compiler and compile the program in the cards and generate an executable. $LOAD loads the executable and $RUN, starts the execution.

i. [0.5 marks] Explain why these special cards have a special sign in front ($) and describe the software components used to interact with those cards. Explain all the software components that are needed for the card $RUN, how they interact with the card, and describe what action the OS takes after all the software components have been executed.
ii. [0.5 marks] Explain what would happen if, in the middle of the FORTRAN compilation process, we detect the card $LOAD. What should the Operating System do in that case?

b) [0.5 marks] Explain how kernel and user modes, combined with Memory Protection prevent a user from accessing (reading/writing to) the memory area assigned to the OS.

c) [0.4 marks] Write examples of four privileged instructions and explain what they do and why they are privileged instructions.

d) [0.5 mark] Explain, in detail, the complete Interrupt mechanism, starting from an external signal until completion. Differentiate clearly what part of the process is carried out by hardware components, and what part is executed by software.

e) [0.5 marks] Discuss what is a System Call. Explain, in detail, what a System Call is and give examples of known system calls. Explain how system Calls are related to Interrupts and explain how the Interrupt hardware mechanism is used to implement System Calls.

f) [0.6 marks] Explain briefly how a Time-Sharing Operating System works. How does it manage to handle multiple interactive users with a single CPU?

g) [0.5 marks] (Kernel structure) Let us suppose we have a process in a multitasking OS.

i. What are the events needed to make a new process move from the "Ready" to the "End" state?
ii. How about events to move the process from "Running" to "Ready"?

Explain how the OS Kernel will react to these different events (in terms of the states of the system and the transitions between these states, and the routines of the kernel involved in the process).

Hint: explain at least *two* kinds of events to be considered, which will produce different state changes and their corresponding transitions. Explain how the OS Kernel will react to this event in detail. Include the OS components involved in these transitions.

h) [1 marks] Let us assume you have an interrupt-based interface/controller to a dumb terminal, 40 characters wide by 20 lines height. The terminal displays one character and sends an interrupt when it is done. Displaying a new character takes 1 millisecond and processing an interrupt takes 50 microseconds.

How long will it take to display a full screen? Calculate how many interrupts will be needed. Calculate how much time will be used for displaying, and how much time for processing interrupts.

Now, let us assume that we change the terminal and have a high-resolution screen, 1000 pixels wide by 400 pixels height. Displaying one pixel takes 1 microsecond and processing an interrupt takes 50 microseconds. Repeat the calculation above.

Discuss the results obtained, and, if any problems are detected, discuss a mechanism to solve the problem, repeat the calculation above, and discuss the results obtained in all the scenarios.

Part II – [5.5 marks] Design and Implementation of an Interrupt Simulator

The objective of this section is to build a small simulator of an interrupt system, which could be used for performance analysis of different parts of the interrupt process. This simulator will also be used in Assignment 3.

i) Input data

As an input, the simulator will receive a trace of a single program. The system has one CPU, one Input device (a sensor that detects individuals coming into a room) and one Output device (an LED-based display that shows how many people are in the room). A pseudocode of the program is as follows:

```
main () {
   initialize variables;           // uses CPU (*1)

   while () {
   x = read_from_sensor();         // reads sensor data into variable x: system call (*2)
   y = calculate (x);              // (*3) uses value of x to calculate info to display
   display(y);                     // it displays the results (*4)
   }
}
```

The trace information is stored in a Table 1 below, as follows:

Activity	Duration

The Activity refers to the type of activity carried out by the program (CPU use or I/O), and the duration is in milliseconds; the data is comma separated. For example, a table looks like this:

```
CPU, 50
SYSCALL 7, 110
END_IO 20, 170
CPU, 100
SYSCALL 12, 250
END_IO 22, 328
CPU, 20
…
```

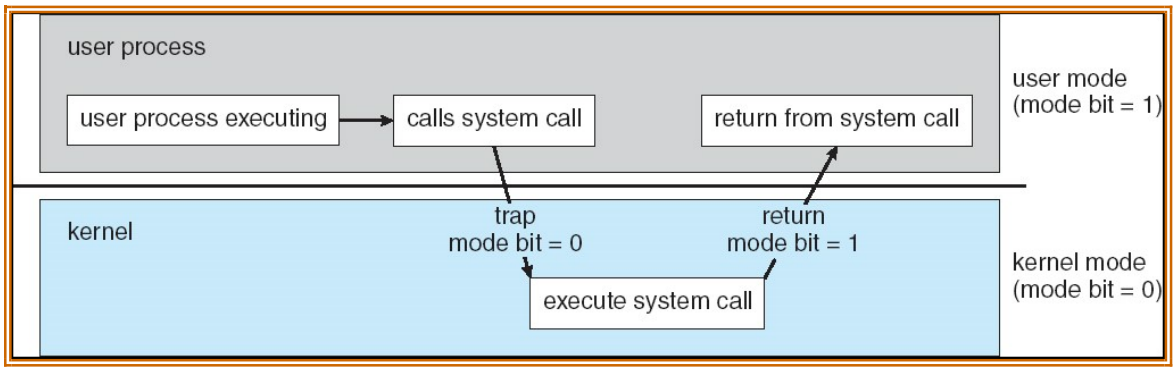
First, we start with program initialization (50 time units of CPU, i.e., *1 in the program above); then, we read from the sensor: the system call is in position 7 of the vector table (SYSCALL 7, called by *2 above). It takes 110 ms to service the call for ISR SYSCALL. Then, it takes 170 ms to process the interrupt, after which the end of interrupt (END_IO, which is in position 20 of the vector table) is executed. The next line corresponds to *3, i.e., processing of the data by the *calculate* function. Finally, another SYSCALL (12, the one to display the results) is called and we repeat.

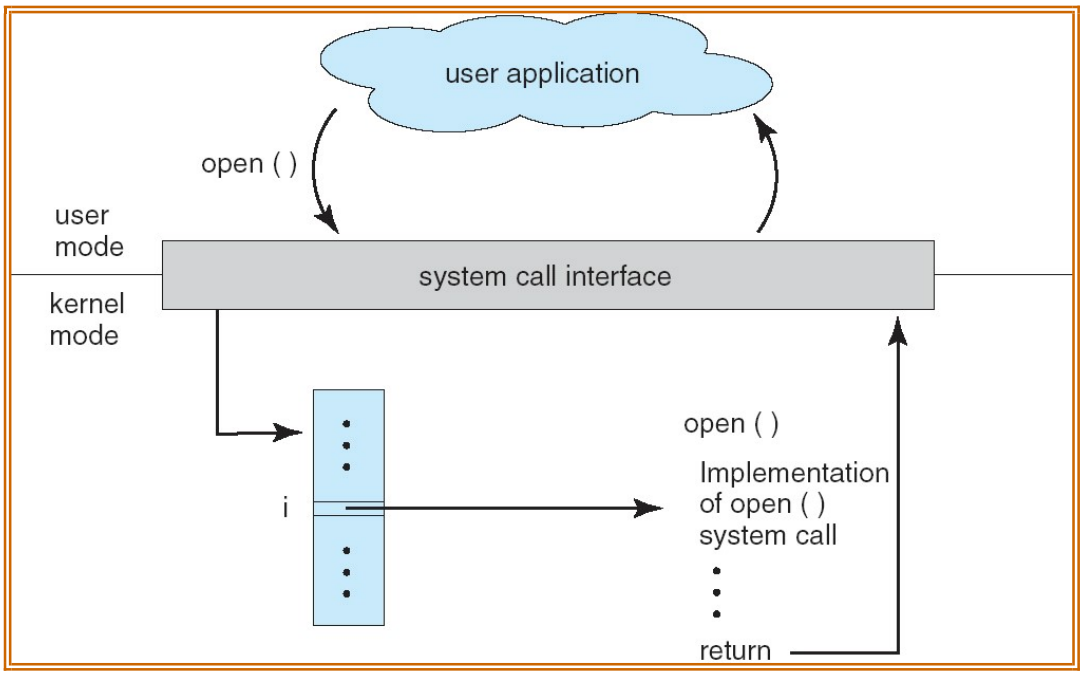
The table must be written in a text file called *trace.txt,* and loaded at the start of your simulation program. This file (and all case studies) must be submitted in the zip folder.

ii) Simulation implementation

Using the information on the input file, you must simulate the advance of the program, logging every activity carried out by the program in an output file.

For system calls, the simulation should try to reproduce the behavior of this state diagram (from Silberschatz *et al.*). The complete detailed behavior of interrupts is found in the video posted on Brightspace.





Your code should use the trace table to simulate all the execution steps. Every step with its simulated time should be stored in an output file called *execution.txt*. **This file should be submitted.**

ASSUME THAT YOU ALWAYS HAVE AN I/O DEVICE AVAILABLE (that is, do not worry about waiting queues at the I/O devices: whenever you request an I/O, the I/O starts immediately).

Each vector is 2 bytes long.

You will need to include a Vector Table as follows (you can change this and have it as an input file or you can hardcode the table into your simulator):

Initial Memory Address	Interrupt Number	ISR Address
	…	
0x00	7	0x0E
	…	
0x00	12	0x18
	…	
0x00	20	0x28
	…	
0x00	22	0x16

The simulation should include all the steps in the interrupt process, which should be in the output trace:

- switch to/from kernel mode (time: negligible; use a duration of 1 ms)
- save/restore context (random value: 1-3 ms)
- depending on the interrupt number, calculate where in memory is the ISR start address (time: negligible; use a duration of 1 ms)
- get ISR address from vector table (time: negligible; use a duration of 1 ms)
- execute ISR body. Include all the activities carried by the ISR (each activity: should be a random value 100-400 ms that you include in the Table 1)
- execute IRET (time: negligible; use a duration of 1 ms)

The *execution.txt* file should include the following information, comma separated:

Time of the event	Duration of the event	Event type

The duration is in ms. Example: for the trace above, the output execution trace in file *execution.txt* could look as follows (**comments are to clarify; should not be included in the file)**:

```
...
590, 1, switch to kernel mode
591, 2, context saved
593, 1, find vector 7 in memory position 14
594, 1, obtain ISR address
595, 24, call device driver          // include all activities of the ISR (takes 110ms)
...
705, 1, IRET                         // IRET after 110ms
712, 170, end of I/O 20: interrupt   // this information is from the trace.txt file
882, 1, switch to kernel mode
883, 1, context saved
884, 1, find vector 20 in memory position 40
...
```

We will submit test cases in BrightSpace; the TAs will mark these cases first. You must include your own test cases, too.

The program must be written in C/C++. You must submit an executable, the source code, all files needed to compile, test scenarios, and scripts to run them. Include, at least, two scripts named "test1" and "test2" that will be used to run 2 different tests automatically. The TAs will use these two to run basic tests, and then will modify your input files to test the correctness of the results. Every test case should be included in a different subdirectory (the *trace.txt* and the *execution.txt* corresponding to the test should be in the corresponding directory). All tests should be submitted in a zip file, *tests.zip*. Your source code should be named *interrupts.cpp* and your header files *interrupts.hpp.* Only **one version** of your code should be submitted. The executable should be named *interrupts* and submitted with the rest of the files.

You must run at least 20 simulation tests, analyze the results, and write a short report (1-2 pages long) in text format. In your report, you should discuss the influence of different steps in the interrupt process:

- What happens if the execution of the ISR is long?
- How about the different steps of the ISR? (saving information in the PCB, calling the scheduler, executing the scheduler, save/restore context). How does the difference in speed in these steps affect the overall execution time of the process?
- Identify in your program ALL sources of overhead, and calculate the ratio between actual CPU use, I/O activities and overhead
- You can process this data using a Python script, Excel spreadsheet or any other tools for separating the overhead from the actual work of your program (i.e., CPU use for processing and actual I/O needed by the program).
- Ask yourselves other interesting questions and try to answer them through simulations. For instance: what happens if we have addresses of 4 bytes instead of 2? What if we have a faster CPU

and now all processing takes ½ of the time? (this is non-mandatory but you should be asking yourselves these kinds of questions and answering them to learn and to prepare for exams).

Submit your report in the *report.txt* file.

Marking Scheme:

TOTAL: 10 marks

For Part II:

- Correctness (including error checking): 65%
- Documentation, output and report: 25% (including proper comments)
- Program structure: 5%
- Style and readability: 5%

Programs that do not compile will receive a mark of zero.