

Computing platforms (Spring 2025)

week 1

Exercise 1

Pick an operating system of your choice and answer the following questions.

a) **What is the operating system you chose?**

MacOS

b) **What function/program can you use to find information of the running processes in this OS?**

CLI program on MacOS to monitor running programs is **ps**. System level API for same information is found in **sysctl** call.

- c) Pick one process and list all information of the process you find relevant.

Let's select running VSCode app for inspection (PID 5112 below):

```
$ ps|grep -i "Visual Studio Code.app"
34048 ttys008      0:00.01 grep -i Visual Studio Code.app
5112  ttys011      0:00.02 /opt/homebrew/bin/bash --init-file /Applications/
Visual Studio Code.app/Contents/Resources/app/out/vs/workbench/contrib/
terminal/common/scripts/shellIntegration-bash.sh
```

We can get all the essential information using command:

```
ps -p 5112 -o pid,ppid,uid,user,%cpu,%mem,vsz,rss,TTY,stat,start,time,command
```

which give

```
PID PPID  UID USER  %CPU %MEM  VSZ  RSS TTY      STAT STARTED TIME COMMAND
5112 5097  501 juha   0,0  0,0 410296320 208 ttys011 Ss+ 11Tam25 0:00.02 /
opt/homebrew/bin/bash --init-file /Applications/Visual Studio Code.app/Contents
/Resources/app/out/vs/workbench/contrib/terminal/common/scripts/
shellIntegration-bash.sh
```

Legend for above listed process information:

Parameter	Description
PID	Process ID, a unique identifier for the process.
PPID	Parent Process ID, the ID of the process that started this process.
UID	User ID, the numerical identifier of the process owner.
USER	Username of the process owner.
%CPU	Percentage of CPU being used by the process.
%MEM	Percentage of memory being used by the process.
VSZ	Virtual memory size used by the process, in kilobytes.
RSS	Resident Set Size, the amount of physical memory used by the process, in kilobytes.
TTY	The terminal associated with the process.
STAT	Current status of the process (e.g., S for sleeping, R for running).
STARTED	The time or date when the process started.
TIME	Cumulative CPU time used by the process (in the format HH:MM:SS).
COMMAND	The full command with arguments that was used to start the process.

Exercise 2

Process creation and termination.

a) **Briefly describe what happens when a process is created?**

Process creation involves several well defined steps to prepare a new process for execution. According to Stallings [1], the general steps for process creation are as follows:

1. **Assign a unique identifier:** Each created process is given an unique identifier. This is added to the process table, which has an entry for each process.
2. **Allocate space:** Memory is allocated for the process's image, including program, data, and stack. If the process inherit memory or resources from a parent process, the appropriate linkages are set up.
3. **Initialize process control block content:**
 - Identification portion, contain such as PID and parent PID.
 - Processor state, being mostly zero except for program counter and stack pointers.
 - Control information, such as the initial state (Ready or Ready/-Suspend) and priority.
4. **Set scheduling linkages:** The process is added to appropriate scheduling queue such as the Ready queue or Ready/Suspend queue.
5. **Create or expand other data structures:** Accounting files and other structures are initialized for monitoring, billing, or performance evaluation.

b) List and describe three different reasons leading to process termination

Processes in an operating system can terminate for various reasons. Below are three common causes, along with their descriptions and examples in MacOS:

1. Normal Completion

- **Description:** A process terminates normally when it successfully completes the execution of its program.
- **Example:** In MacOS, a process calls the `exit()` system call to signal successful completion. The operating system cleans up resources and notifies the parent process.

2. Error or Exception

- **Description:** Processes terminate abnormally due to runtime errors or exceptions such as division by zero or segmentation faults.
- **Example:** In MacOS, dereferencing a null pointer results in a segmentation fault terminating the process.

3. External Termination (Kill or Signal)

- **Description:** A process is terminated by an external factor, such as:
 - A parent process sending a termination signal (`SIGKILL`, `SIGTERM`).
 - System-initiated termination during shutdown or when resources are insufficient (For example OOM-killer).
- **Example:** Using the `kill` command to send a `SIGKILL` signal:

```
kill -9 <PID>
```

Exercise 3

Context switch on the TOY processor.

- a) **What information about the context of a process needs to be saved to the process control block (PCB) during a mode switch?**

When a mode switch occurs, the operating system must save the current context of the process to its PCB to ensure it can resume execution later. The following information is saved:

- General-purpose registers R0 .. R3: These contain the current working data of the process.
- Special registers:
 - SP (Stack Pointer): Tracks the top of the user process's stack.
 - FP (Frame Pointer): Tracks the base of the current stack frame.
 - PC (Program Counter): Indicates the next instruction to be executed.
- Process priority, execution state, and any information required for scheduling or resource management.

- b) **What happens during a mode switch when an interrupt occurs?**

When an interrupt occurs, the TOY processor performs the following steps:

- (a) Save the Program Counter (PC): The address of the next instruction in the user process is saved to the **EPC** register.
- (b) Load Interrupt Handler Address: Address of interrupt handler is loaded into the **PC** register, switching execution to interrupt handler.
- (c) Switch to System Mode: Processor changes its mode from user to system, enabling access to all registers, including **K0** and **K1**.
- (d) Execute Interrupt Handler: Processor begin executing instructions from interrupt handler.

- c) **What happens during a mode switch when the operating system returns to the interrupted process?**

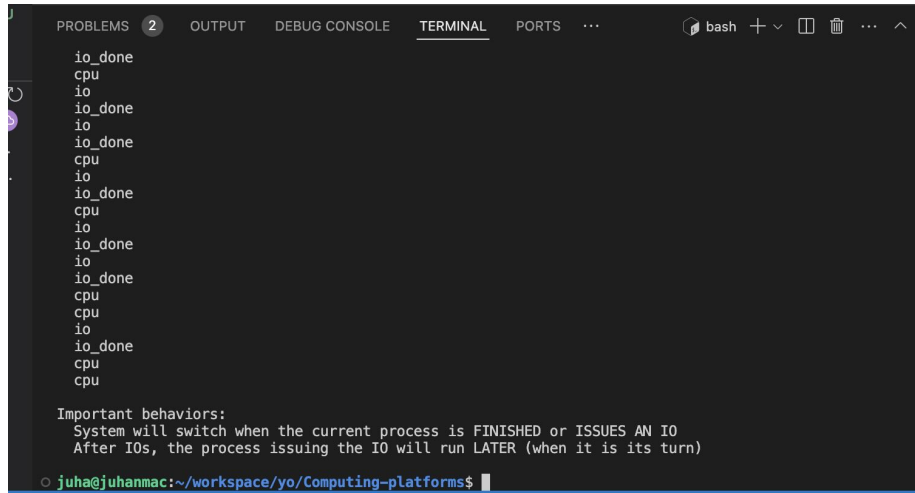
After interrupt is handled operating system restores the context of interrupted process:

- (a) Restore PC: Address saved in the **EPC** register in part b is copied back to the **PC** register.
- (b) Switch to User Mode: Processor changes mode from system to user, restricting access to **EPC**, **K0**, and **K1**.
- (c) Resume Execution: Process continues execution from address now stored in the **PC** register, resuming where it was interrupted.

Exercise 4

Processes and process states.

a) process-run.py execution



```
io_done
cpu
io
io_done
io
io_done
cpu
io
io_done
cpu
io
io_done
io
io_done
cpu
cpu
io
io_done
cpu
cpu
io
io_done
cpu
cpu

Important behaviors:
System will switch when the current process is FINISHED or ISSUES AN IO
After IOs, the process issuing the IO will run LATER (when it is its turn)
```

Figure 1: Process Execution Diagram

b) cpu-intro.pdf questions 1, 2

1) Run process-run.py with the following flags: -l 5:100,5:100.
What should the CPU utilization be

100% because flags say cpu utilization is 100%

2) Now run with these flags: ./process-run.py -l 4:100,1:0.

As it's seen in results, second PID get to execute after first PID and will be blocked while IO is in progress. Both processes complete after 11 cycles.

```
$ python3 ./process-run.py -l 4:100,1:0 -c -p
```

Time	PID: 0	PID: 1	CPU	IOs
1	RUN:cpu	READY	1	
2	RUN:cpu	READY	1	
3	RUN:cpu	READY	1	
4	RUN:cpu	READY	1	
5	DONE	RUN:io	1	
6	DONE	BLOCKED		1
7	DONE	BLOCKED		1
8	DONE	BLOCKED		1
9	DONE	BLOCKED		1
10	DONE	BLOCKED		1
11*	DONE	RUN:io_done	1	

```
Stats: Total Time 11
Stats: CPU Busy 6 (54.55%)
Stats: IO Busy 5 (45.45%)
```

Exercise 5

Processes and process states (cont'd)

- 3) **Switch the order of the processes: `-l 1:0,4:100`. What happens now?**

Now first PID get blocked when it start to do I/O which follow there's context switch to second PID which get to use CPU. This interleaves two processes making overall process faster. Now both PIDs finish in 7 cycles instead of previously seen 11 cycles.

- 4) **We'll now explore some of the other flags. What happens when you run the following two processes (`-l 1:0,4:100 -c -S SWITCH_ON_END`), one doing I/O and the other doing CPU work?**

Now context switching is blocked until one PID is finished, this cause second PID to wait until I/O on first PID is finished. Effect of this for final result is overall process take 11 cycles, e.g. same as seen in **2**).

- 5) **Now, run the same processes, but with the switching behavior set to switch to another process whenever one is WAITING for I/O (`-l 1:0,4:100 -c -S SWITCH_ON_IO`). What happens now?**

Now we see same result as on **3**), ie. while PID become blocked by I/O another PID will be switched on to.

- 6) One other important behavior is what to do when an I/O completes. What happens when you run this combination of processes? (`./process-run.py -l 3:0,5:100,5:100,5:100 -S SWITCH_ON_IO -c -p -I IO_RUN_LATER`). Are system resources being effectively utilized?

No, system resources are not being effectively utilized. This can be seen from table below where interleaving CPU and I/O is not perfect:

```
$ python3 ./process-run.py -l 3:0,5:100,5:100,5:100 -S SWITCH_ON_IO -c -p -I IO_RUN_LATER
```

Time	PID: 0	PID: 1	PID: 2	PID: 3	CPU	IOs
1	RUN:io	READY	READY	READY	1	
2	BLOCKED	RUN:cpu	READY	READY	1	1
3	BLOCKED	RUN:cpu	READY	READY	1	1
4	BLOCKED	RUN:cpu	READY	READY	1	1
5	BLOCKED	RUN:cpu	READY	READY	1	1
6	BLOCKED	RUN:cpu	READY	READY	1	1
7*	READY	DONE	RUN:cpu	READY	1	
8	READY	DONE	RUN:cpu	READY	1	
9	READY	DONE	RUN:cpu	READY	1	
10	READY	DONE	RUN:cpu	READY	1	
11	READY	DONE	RUN:cpu	READY	1	
12	READY	DONE	DONE	RUN:cpu	1	
13	READY	DONE	DONE	RUN:cpu	1	
14	READY	DONE	DONE	RUN:cpu	1	
15	READY	DONE	DONE	RUN:cpu	1	
16	READY	DONE	DONE	RUN:cpu	1	
17	RUN:io_done	DONE	DONE	DONE	1	
18	RUN:io	DONE	DONE	DONE	1	
19	BLOCKED	DONE	DONE	DONE		1
20	BLOCKED	DONE	DONE	DONE		1
21	BLOCKED	DONE	DONE	DONE		1
22	BLOCKED	DONE	DONE	DONE		1
23	BLOCKED	DONE	DONE	DONE		1
24*	RUN:io_done	DONE	DONE	DONE	1	
25	RUN:io	DONE	DONE	DONE	1	
26	BLOCKED	DONE	DONE	DONE		1
27	BLOCKED	DONE	DONE	DONE		1
28	BLOCKED	DONE	DONE	DONE		1
29	BLOCKED	DONE	DONE	DONE		1
30	BLOCKED	DONE	DONE	DONE		1
31*	RUN:io_done	DONE	DONE	DONE	1	

```
Stats: Total Time 31
Stats: CPU Busy 21 (67.74%)
Stats: IO Busy 15 (48.39%)
```


- 7) Now run the same processes, but with **-I IO_RUN_IMMEDIATE** set, which immediately runs the process that issued the I/O. How does this behavior differ? Why might running a process that just completed an I/O again be a good idea?

Now interleaving of PIDs is well aligned which is seen at 100% CPU utilization. Switching to process that just completed I/O maybe good idea if it has more I/O operations in queue. This will allow slow I/O operations to be running continuously.

```
$ python3 ./process-run.py -l 3:0,5:100,5:100,5:100 -S SWITCH_ON_IO -c -p -I IO_RUN_IMMEDIATE
```

Time	PID: 0	PID: 1	PID: 2	PID: 3	CPU	IOs
1	RUN:io	READY	READY	READY	1	
2	BLOCKED	RUN:cpu	READY	READY	1	1
3	BLOCKED	RUN:cpu	READY	READY	1	1
4	BLOCKED	RUN:cpu	READY	READY	1	1
5	BLOCKED	RUN:cpu	READY	READY	1	1
6	BLOCKED	RUN:cpu	READY	READY	1	1
7*	RUN:io_done	DONE	READY	READY	1	
8	RUN:io	DONE	READY	READY	1	
9	BLOCKED	DONE	RUN:cpu	READY	1	1
10	BLOCKED	DONE	RUN:cpu	READY	1	1
11	BLOCKED	DONE	RUN:cpu	READY	1	1
12	BLOCKED	DONE	RUN:cpu	READY	1	1
13	BLOCKED	DONE	RUN:cpu	READY	1	1
14*	RUN:io_done	DONE	DONE	READY	1	
15	RUN:io	DONE	DONE	READY	1	
16	BLOCKED	DONE	DONE	RUN:cpu	1	1
17	BLOCKED	DONE	DONE	RUN:cpu	1	1
18	BLOCKED	DONE	DONE	RUN:cpu	1	1
19	BLOCKED	DONE	DONE	RUN:cpu	1	1
20	BLOCKED	DONE	DONE	RUN:cpu	1	1
21*	RUN:io_done	DONE	DONE	DONE	1	

```
Stats: Total Time 21
Stats: CPU Busy 21 (100.00%)
Stats: IO Busy 15 (71.43%)
```

References

- [1] William Stallings. *Operating Systems: Internals and Design Principles*. Pearson, 9th edition, 2018.