# COMP3230 Principle of Operating Systems

## 2020 Midterm Examination

**(Total 11 pages)**

**Submit your answers within 48 hours, i.e., before 11:59AM Oct 24, 2020. No late submission is allowed.**

**Open-book exam:** you can read all the lecture notes and tutorial materials saved in your own computer, however **Internet search (e.g., Google search) is not allowed**. You are also not allowed to communicate with any classmates or friends via any means (e.g., email, WeChat, WhatsApp, Line) during the 48 hours, for the discussion of the exam questions.

**Warning: Plagiarism is Totally Unacceptable.** Students who are suspected for copying (i.e., detected by our plagiarism checker) will be interviewed. The student who offers the solution will receive double penalty.

**Questions during the exam:** we will only answer questions during :

> **Class A**: 2:30-3:00, Oct 23 (Fri) via Class A Zoom meeting
>
> **Class B**: 1:30-2:00, Oct 22 (Thur) via Class B Zoom meeting
>
> In case of any critical correction, it will be broadcast in the news group (Moodle News Announcement). Please check your email or Moodle.

**Answer Book:** We have prepared an answer book in MS Words format for you to fill up the answers. Please download it from Moodle before the exam. You should type all your answers in the answer book and submit it to Moodle before 11:59AM Oct 24, 2020. The submitted file can be a pdf file or a Words file. The file name should be named as COMP3230(A/B)-YourUID-LastName-FirstName (e.g., COMP3230A-1234567-Xu-Pengfei.pdf). We will deduct 5 marks if you didn't follow this naming scheme. **Warning:** DON'T take photos via a mobile phone and embed the photos to the Words file, nor using a Scan App to produce the pdf file for the submission. **You will get 20 marks deducted if you do so** (explained in the Midterm Exam Guide).

**No makeup exam will be arranged.** Absent from the exam will receive zero mark by default. Even you have a good reason (e.g., with the sick leave certificate issued by the doctor, tested positive for COVID-19), you will only get **60% of your final exam mark**.

University Number:

## Problem 1. Multiple Choice Questions (45 marks)

**Each question may have <u>one</u> or <u>more than one</u> incorrect statement to be chosen unless "Choose One" is specified.** [**Marking scheme**: **3 marks** are awarded for each question with all incorrect statement(s) chosen and give corrections (No need for those correct statements, or if you choose **(5)** None of the above). We give at most 1 mark as partial credit for your answer with ALL correct choice(s) chosen but incomplete corrections (i.e., not all incorrect statements are corrected or the correction is wrong). You get a zero mark if you only put your choice(s) without correcting the wrong statement(s).]

1. ( ) Which of the following statements about the use of **Linux command** in workbench is/are **INCORRECT**?

   (1) "lscpu" shows the number of physical cores in workbench.

   (2) With hyper-threading, we can run two threads on the same physical core at the same time and double the processing speed.

   (3) "ps aux" shows all the processes created by a single user.

   (4) "top -u" shows all user-space processes created in workbench.

   (5) None of the above (i.e., (1)-(4) are all **CORRECT**)

2. ( ) Which of the following statements about ***microkernel*** is **INCORRECT**? (Choose One)

   (1) The microkernel OS consists of only the minimal set of vital functions of the operating system, thus the execution time of your program is much faster.

   (2) The primary overhead of the microkernel architecture is inter-process communication (IPC).

   (3) The microkernel architecture is easily extendable, i.e., if any new services are to be added they are added to user address space.

   (4) The microkernel architecture is more secure, i.e., if one service crashes it does not directly affect others.

   (5) None of the above (i.e., (1)-(4) are all **CORRECT**)

3. ( ) Which of the following statements about ***virtual machine*** (VM) and ***container*** is/are **INCORRECT**?

   (1) Multiple containers can run on the same server by sharing a single OS kernel.

   (2) Each VM requires its own operating system.

   (3) workbench is just a "container", not a real machine.

   (4) vmlinux is the virtual machine running on workbench.

   (5) None of the above (i.e., (1)-(4) are all **CORRECT**)

4. ( ) Which of the following statements about ***static*** and ***dynamic*** library is/are **INCORRECT**?

   (1) One key advantage of static libraries is that they are directly executable.

   (2) Every time you change or up-grade the static libraries, you have to recompile your program into a new executable.

   (3) Shared library is loaded by the operating system into the calling process' stack segment.

   (4) The object files of static libraries are inserted into the executable file (a.out), thus the code size is much larger.

   (5) On Linux, the file names of static libraries usually end with a suffix of ".a", while those of shared libraries end with a suffix of ".so".

5. ( ) Which of the following statements about **htop** is **INCORRECT**? (Choose One)

   (1) In htop, RES column is the amount of physical memory this process is using.

   (2) Right after a child process is created by fork(), the child should have the same VIRT as its parent.

   (3) After the child process executes for a while, its VIRT and RES could be different from its parent's.

   (4) When a process performs "malloc(4 * 1024 * 1024)", it's VIRT and RES will immediately increase by 4MB.

   (5) None of the above (i.e., (1)-(4) are all **CORRECT**).

6. ( ) Which of the following statements about **System Calls** is/are **INCORRECT**?

   (1) System call is a user defined function but running in kernel mode.

   (2) All system call functions are defined in GNU C Library (glibc).

   (3) The integer value 80 in "INT $0x80" is a system call number.

   (4) System calls overhead is much less than user-mode procedure calls.

   (5) None of the above (i.e., (1)-(4) are all **CORRECT**).

7. ( ) Which of the following statements about ***process address space*** is/are **INCORRECT**?

   (1) Global variables with non-zero initial value are stored in *data segment*.

   (2) Local variables with non-zero initial value are stored in *data segment*.

   (3) Uninitialized local static variables are stored in *stack segment*.

   (4) The pointer variable returned from malloc() is stored in *heap* or *stack segment*.

   (5) Shared memory created by shmget(), is allocated at *heap segment*.

8. ( ) Which of the following statements about ***stack*** and ***heap*** is/are **INCORRECT**?

(1) Stack access is usually faster than heap.

(2) Variables stored in stack are usually accessed via *stack pointer* (`%esp`).

(3) Arrays that may change size dynamically are better to be allocated on the heap.

(4) The size of heap can be reported by the Linux *size* command.

(5) None of the above (i.e., (1)-(4) are all **CORRECT**)

9. ( ) Which of the following statements about **fork()** is/are **INCORRECT**?

(1) The child process is created by duplicating the parent's task structure (`struct task_struct`) and page table.

(2) During fork(), OS will duplicate the page table and mark all the page table entries *read-only*.

(3) Right after `fork()`, the same variable (e.g., `int a`) defined in parent and child will have the same virtual address but different physical addresses.

(4) The child process needs to wait until parent process changes those read-only pages to writable before it can update the data.

(5) None of the above (i.e., (1)-(4) are all **CORRECT**)

10. ( ) Which of the following statements about **wait()** and **waitpid()** is **INCORRECT**? (Choose One)

(1) The `wait()` function returns child `pid` on success.

(2) The call `wait(&status)` is equivalent to `waitpid(-1, &status, 0)`.

(3) The call `wait(NULL)` waits for all child processes to terminate.

(4) The call `waitpid(-1, &status, WNOHANG)` returns immediately if no child has exited.

(5) None of the above (i.e., (1)-(4) are all **CORRECT**)

11. ( ) Which of the following statements about **zombie process** is/are **INCORRECT**?

(1) If a parent process crashes during the middle of execution, all its child processes that are still running become the zombie processes.

(2) A zombie process is a running process, but whose parent process has finished or terminated.

(3) Zombie processes are harmful as they may still occupy memory (i.e., RSS is usually non-zero).

(4) Zombie process can be killed by "kill -9 [zombiePID]".

(5) None of the above (i.e., (1)-(4) are all **CORRECT**)

12. ( ) Which of the following statements about ***process scheduling*** is **INCORRECT**? (Choose One)

    (1) Shortest Job First (SJF) may lead to process starvation.

    (2) Round Robin (RR) is better than FCFS in terms of response time.

    (3) If the quantum time of Round Robin (RR) is very large, then it is equivalent to SJF.

    (4) Multilevel Feedback Queue (MFQ) is the most general CPU-scheduling algorithm, which is adopted by most modern operating systems.

    (5) None of the above (i.e., (1)-(4) are all **CORRECT**)

13. ( ) Which of the following statements about **Highest Response Ratio Next (HRRN)** scheduling is/are **INCORRECT**?

    (1) HRRN is a preemptive version of Shortest Job First (SJF) algorithm.

    (2) HRRN adopts dynamic priorities, but it still favors the shorter jobs.

    (3) HRRN may lead to process starvation if there are many short jobs.

    (4) HRRN has some overhead in tracking the remaining service time for each process in the ready queue.

    (5) None of the above (i.e., (1)-(4) are all **CORRECT**)

14. ( ) Which of the following statements about ***signal*** is/are **INCORRECT**?

    (1) kill() system call will immediately terminate the identified process.

    (2) signal() system call is used to send a signal to a process.

    (3) Signal handlers can be called anytime when the program is running.

    (4) SIGCHLD is sent from a parent process to a child process.

    (5) None of the above (i.e., (1)-(4) are all **CORRECT**)

15. ( ) Which of the following statements about **multiprocessor scheduling** is/are **INCORRECT**?

    (1) Current Linux uses a global queue to schedule processes.

    (2) A long-running program could be scheduled to run on different CPU cores during its execution.

    (3) “taskset 0x7 [pid]” forces the process with PID=[pid] to run on core #7.

    (4) Gang scheduling is a good strategy for scheduling parallel programs (e.g., parallel 4-way merge sort) on multicore systems.

    (5) None of the above (i.e., (1)-(4) are all **CORRECT**)

## Problem 2: Process Creation using Fork() (15 marks)

The code snippets below omit irrelevant details and error handling due to space constraint. We assume every fork() succeeds. [**Note:** You are not required to run this code in workbench or your own computer to show us the screenshot.]

```
main() {
int count = 0;
pid_t pid1, pid2;

pid1 = fork();
count++;
if (pid1 == 0) {
    pid2 = fork();
    count++;
    if (pid2==0) {
       count=count+10;
       exit(0);
    }
} else {
  count++;
}
printf("Hello world!\n");
printf("Count is  %d\n", count);
}
```

Answer the following two questions:

(a) **(7%)** How many times will the message "Hello world!\n" be displayed? Give detailed reasons.

(b) **(8%)** What will be the largest value of "count" displayed? Give detailed reasons.

[**Note**: You need to show you fully understand how the above program work and explain how it leads to the result. If you only give the answer without explanation (or give wrong explanation), you will get a zero mark. **Write your answers on the provided answer book**. **For this question, you are allowed to draw a figure using any drawing tools (e.g., Paint, PowerPoint) to illustrate your analysis if you prefer to do so (Not compulsory)**.]

University Number:

## Problem 3: Signals and Shared Memory (20%)

In this program, you will create a shared memory segment to be shared by the parent and the child process and allow the child to read the updated values from the shared memory made by the parent.

**Detailed Description:** After the parent creates a child process, both the parent (writer) and child (reader) will attach the shared memory segment to their address space at ***a*** and ***b*** respectively. Each will then perform 10 iterations to access the shared memory as follows: in the *i*th iteration ($0 \leq i < 10$), the parent process (writer) will first update the value of a[i] (i.e., a[i]=i), then let the child process (reader) read the updated value via reading b[i]. As discussed in Lecture 2, if there is no proper synchronization between the two processes, there could be a ***race condition*** since parent and child could run at different speed. To guarantee the child always reads the updated value written by the parent, you need to arrange **SIGSTOP** and **SIGCONT** signals by adding proper function calls (more details to come) in the given code template to make sure whenever the parent updates one entry in the shared memory the child should read the updated value afterwards before they (parent and child) advance to the next iteration (i.e., i → i++). To be more precise, the expected output and printing order are shown in the screenshot below:

```
1.  int main()
2.  {
3.  pid_t pid;
4.  int shmid, status;
5.  int *a, *b, i;
6.
7.  /* A: What to be done initially? */
8.
9.  pid = fork();
10.if (pid == 0) {   /* Child Process */
11.    /* B: what to be done by Child before entering for-loop */
12.
13.    for( i=0; i< 10; i++) {
         /* C: Anything to be done here? */
14.      printf("\t Child at iteration %d, read b[%d] = %d.\n", i, i, b[i]);
       }
15.    /* D: Anything to be done after the for-loop? */
16.
17. }
18. else {  /* Parent Process */
19.    /* E: What to be done by parent before entering for-loop? */
20.
21.    sleep(1); /* intentionally added. Don't remove it! */
22.    for( i=0; i< 10; i++) {
23.      /* F: Anything to be done here? */
24.      a[i]= i;
25.      printf("Parent at iteration %d, writes a[%d]= %d.\n", i, i, a[i]);
26.      /* G: Anything to be done here? */
27.   }
28. /* H: What to be done at the end of the program? */
29.
30. }
31.}
```

```
jpwang@workbench:~$ gcc q1.c -o main
jpwang@workbench:~$ ./main
Parent at iteration 0, writes a[0]= 0.
        Child at iteration 0, read b[0] = 0.
Parent at iteration 1, writes a[1]= 1.
        Child at iteration 1, read b[1] = 1.
Parent at iteration 2, writes a[2]= 2.
        Child at iteration 2, read b[2] = 2.
Parent at iteration 3, writes a[3]= 3.
        Child at iteration 3, read b[3] = 3.
Parent at iteration 4, writes a[4]= 4.
        Child at iteration 4, read b[4] = 4.
Parent at iteration 5, writes a[5]= 5.
        Child at iteration 5, read b[5] = 5.
Parent at iteration 6, writes a[6]= 6.
        Child at iteration 6, read b[6] = 6.
Parent at iteration 7, writes a[7]= 7.
        Child at iteration 7, read b[7] = 7.
Parent at iteration 8, writes a[8]= 8.
        Child at iteration 8, read b[8] = 8.
Parent at iteration 9, writes a[9]= 9.
        Child at iteration 9, read b[9] = 9.
jpwang@workbench:~$
```

**Some functions to be added to the given code template:**

**Shared memory creation**: shmid = shmget(?, 10*sizeof(int),?). You need to figure out where to add this line and the right arguments to be added in the shmget() function call by yourself.

**Shared memory attachment**: Parent and child should attach the shared memory (shmid) to ***a*** and ***b*** respectively using (int *) shmat(shmid, 0, 0) and meet the two restrictions:

- Parent can only access the shared memory via pointer ***a***

- Child can only access the shared memory via pointer ***b***.

**Shared memory de-attachment:** Parent and child should separately de-attach the shared memory before they exit using shmdt(?).

**Other system calls or library functions that are needed:** kill(), getpid(), wait() (or waitpid()), exit(), shmctl(). **That is all, NOTHING ELSE.** You have to figure out the right arguments to be passed to these function calls by yourself.

**A Few More Notes:**

- **No additional variables needed:** You are not allowed to add new variables in the given program code. "&status" could be used in wait() or waitpid() whenever needed.
- **No error checking needed:** We assume all the system calls and function calls always succeed and never fail.
- **No actual code execution:** We don't need you to write the complete code nor test the code in workbench. Some minor syntax errors are tolerable, but marks could be deducted if the error is severe.

**Marking scheme:**

- **(14%) Code part:** To get the full mark (14 marks), the code you added in **A-H** should be able to make the main program generate the exact outputs as shown in the screenshot. If some critical steps are missing or added at a wrong place, you are likely to get a zero mark as your program won't work correctly. In this case, no partial credit will be given.
- **(6%) Analysis Part:** You need to explain how your program would work correctly, particularly how the parent could ensure the child has firmly stopped and be instructed to advance to the next iteration at a proper time. We also expect you to explain how the shared memory segment was handled before the processes terminate. [**Note:** Please address the issues directly. Don't waste the space for describing the basic knowledges of signals or shared memory that can be copied from the lecture note.]

University Number:

**Answer: Please write your answers on the provided answer book.**

| Code segment | C code to be added by you (leave it blank if nothing is needed) |
|---|---|
| **A** | // initialization |
| **B** | // child: |
| **C** | // child: for-loop |
| **D** | // child: after for-loop |
| **E** | // parent |
| **F** | // parent: for-loop (before a[i]= i) |
| **G** | // parent: for-loop (after printf()) |
| **H** | // before the end of the program |
| **Analysis Part (6%)** | |

University Number:

## Problem 4: Process Scheduling (20 marks)

Consider the following processes, arrival times, and CPU processing time requirements:

| Process ID (PID) | Arrival Time | CPU Time |
|---|---|---|
| 1 | 0 | 4 |
| 2 | 1 | 4 |
| 3 | 4 | 3 |
| 4 | 8 | 2 |

Show the scheduling order for these processes under **(1) First-In-First-Out (FIFO)**, **(2) Round-Robin (RR)** with a **quantum = 1 time unit**, and **(3) Shortest-Remaining-Time First (SRTF)**. We assume the context switch overhead is 0. For each of the three scheduling algorithms, fill up the Gantt chart, calculate the *turnaround time* and *waiting time* for each process, and report the ***average turnaround time*** and ***average waiting time***. [**Note:** We have added some tie-breaking rules for you already. If additional tie-breaking rules are still needed, you can use those we discussed in the Midterm Exam Guide. The first few entries have been filled for you. You can add more columns if needed. **Write your answers on the provided answer book.**]

**FIFO: (6%)**

**(2%)** Fill up the Gantt chart with the process ID that is running on the time slot.

| P1 | P1 | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

**(2%) fill up the table**

| Process ID (PID) | Arrival Time | CPU Time | Completion Time | Turn Around Time | Waiting Time |
|---|---|---|---|---|---|
| 1 | 0 | 4 | | | |
| 2 | 1 | 4 | | | |
| 3 | 4 | 3 | | | |
| 4 | 8 | 2 | | | |

**(1%) Average Turn Around time = _____**
**(1%) Average Waiting time = _____**

**RR: (6%)**

**(2%)** Fill up the Gantt chart with the process ID that is running on the time slot. To break the tie (if needed): if a new process arrives at the same time as a running process is preempted due to the time quantum expiration, the arriving process will always be placed ahead of the preempted process at the end of the ready queue.

| P1 | P2 | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

University Number:

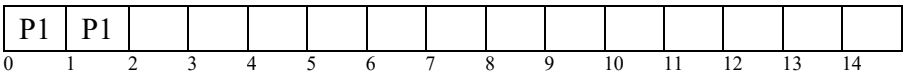
**(2%) fill up the table**

| Process ID (PID) | Arrival Time | CPU Time | Completion Time | Turn Around Time | Waiting Time |
|---|---|---|---|---|---|
| 1 | 0 | 4 | | | |
| 2 | 1 | 4 | | | |
| 3 | 4 | 3 | | | |
| 4 | 8 | 2 | | | |

**(1%) Average Turn Around time = _____**
**(1%) Average Waiting time = _____**

**SRTF: (8%)**

**(4%)** Fill up the Gantt chart with the process ID that is running on the time slot. To break the tie (if needed), we assume an arriving process has a higher priority than the running process.




**(2%) fill up the table**

| Process ID (PID) | Arrival Time | CPU Time | Completion Time | Turn Around Time | Waiting Time |
|---|---|---|---|---|---|
| 1 | 0 | 4 | | | |
| 2 | 1 | 4 | | | |
| 3 | 4 | 3 | | | |
| 4 | 8 | 2 | | | |

**(1%) Average Turn Around time = _____**
**(1%) Average Waiting time = _____**

**---------- End of the exam paper ----------**