

**THE UNIVERSITY OF HONG KONG**  
**FACULTY OF ENGINEERING**  
**DEPARTMENT OF COMPUTER SCIENCE**  
**COMP3230A Principles of Operating Systems**

**Date: December 09, 2020**

**Time: 2:30 – 5:30pm**

***Special note:** This is an online open-book exam. The exam is **THREE** hours. You are permitted to refer to the following electronic/printed materials in the examination: lecture slides, tutorial/assignment handout and sample solutions, and self-made notes. **Internet searching is not allowed.** **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 upload it to Online Exam (OLEX) system before the end of examination. The submitted file can be a MS Words file (.doc) or a pdf file (.pdf).*

*Only approved calculators as announced by the Examinations Secretary can be used in this examination. It is candidates' responsibility to ensure their calculator operates satisfactorily, and candidates must record the name and type of the calculator after the University Number in their answer script.*

**(Total: 11 pages)**

<b>Problem No.</b>	<b>Score</b>
<b>1 (30%)</b>	<b>/30</b>
<b>2 (10%)</b>	<b>/10</b>
<b>3 (20%)</b>	<b>/20</b>
<b>4 (20%)</b>	<b>/20</b>
<b>5 (20%)</b>	<b>/20</b>
<b>Total</b>	<b>/100</b>

### Problem 1. Multiple Choice Questions with Justifications. (30%)

[Marking scheme: Two marks each. Each question might have one or more than one incorrect statement. To get full marks you must identify **ALL** the incorrect statement(s) and explain why the statement is incorrect, or correct the statement directly to show you understand the problems. Marks will be deducted for incorrect justification or without justification. If you choose “(e) None of the above”, no justification is needed. Read details in our Updated Marking Scheme for MCQ.]

1. (   ) Which of the following statements about *static* and *dynamic* library is/are **INCORRECT**?
  - (a) libTeslaFactory.a used in Assignment 2 is a static library.
  - (b) Static libraries use more memory as they copy code into executable files.
  - (c) One advantage of static libraries is that they are directly executable.
  - (d) Calling shared libraries functions won't generate page faults as they are usually resident in memory.
  - (e) None of the above (i.e., (1)-(4) are all **CORRECT**)
  
2. (   ) Which of the following statements about *copy-on-write (COW)* is/are **INCORRECT**?
  - (a) COW is adopted in fork() to reduce copying memory.
  - (b) To enable COW, the *access* (A) bit in the page table entry must be set to 0.
  - (c) Duplication of physical page frame is deferred until either parent or child writes to the page.
  - (d) COW is also adopted in exec() except no duplication on the code segment.
  - (e) None of the above (i.e., (1)-(4) are all **CORRECT**)
  
3. (   ) Which of the following statements about the *VIRT* and *RES* displayed by top command is/are **INCORRECT**?
  - (a) VIRT includes memory that is swapped out to disk and memory allocated but not used.
  - (b) After executing “malloc(4 \* 1024 \* 1024)”, VIRT is increased by about 4MB which is allocated in swap space.
  - (c) A newly created child process should have the same VIRT as its parent.
  - (d) Child process' VIRT could be different from its parent's after it executes for a while.
  - (e) None of the above (i.e., (1)-(4) are all **CORRECT**).
  
4. (   ) Which of the following statements about **Linux threads** is/are **INCORRECT**?
  - (a) All threads created by the same process share the same task\_struct.
  - (b) getpid() system call returns the PID stored in task\_struct.
  - (c) gettid() system call returns the TID stored in task\_struct.
  - (d) pthread\_self() returns the TID of the calling thread stored in task\_struct.
  - (e) None of the above (i.e., (1)-(4) are all **CORRECT**)
  
5. (   ) Which of the following statements about *CPU scheduling* is/are **INCORRECT**?
  - (a) FIFO suffers from *Belady's Anomaly*.
  - (b) SRTF can result in starvation of long tasks if short tasks keep arriving.
  - (c) HRRN has some overhead in tracking the remaining service time of the waiting processes.
  - (d) HRRN suffers from frequent context switching if short tasks keep arriving.
  - (e) None of the above (i.e., (1)-(4) are all **CORRECT**)

6. (   ) Which of the following statements about *shared memory* is/are **INCORRECT**?
- (a) `shmget()` returns a pointer to the starting address of the shared memory segment.
  - (b) You can dynamically `malloc()` additional memory in the shared memory segment whenever needed.
  - (c) Shared memory is used by multiple processes, thus never swapped out to disk.
  - (d) A shared memory segment can be attached as read-only by one process, while writable by another process.
  - (e) None of the above (i.e., (1)-(4) are all **CORRECT**)
7. (   ) Which of the following statements about *Pthreads* is/are **INCORRECT**?
- (a) When the main thread calls `pthread_exit()`, all the threads created by it become *zombie* threads.
  - (b) When the main thread calls `exit()`, all the threads created by it will be terminated immediately.
  - (c) When the main thread calls `pthread_detach()` to de-attach a thread, the thread's stack is immediately freed up.
  - (d) The memory `malloc()`'ed by a thread is automatically freed up when the thread calls `pthread_exit()`.
  - (e) None of the above (i.e., (1)-(4) are all **CORRECT**)
8. (   ) Which of the following statements about *semaphore* is/are **INCORRECT**?
- (a) `sem_trywait(n)` will try to enter the critical section at most *n* times before it gives up.
  - (b) `sem_trywait()` will decrease the semaphore value by one each time it executes until successful.
  - (c) `sem_timedwait(sem_t *sem, &ts)` will only wait for `ts.tv_sec` seconds to get the semaphore designated by `sem`.
  - (d) `sem_getvalue()` returns an integer value that indicates the number of processes blocked outside the critical section.
  - (e) None of the above (i.e., (1)-(4) are all **CORRECT**)
9. (   ) Which of the following statements about *translation lookaside buffer (TLB)* is/are **INCORRECT**?
- (a) TLB is a high-speed cache for storing frequently accessed data.
  - (b) A TLB miss only causes a minor page fault.
  - (c) TLB must be flushed when context switches among threads of the same process.
  - (d) TLB must be flushed whenever switching from user mode to kernel mode (e.g., calling a system call).
  - (e) None of the above (i.e., (1)-(4) are all **CORRECT**).
10. (   ) Which of the following statements about a *4-level paging* in Linux for x86-64 platform is/are **INCORRECT**?
- (a) Not adopted in real systems since it requires 4-level TLB.
  - (b) Need to keep a minimum of four 4KB page tables in memory for address translation.
  - (c) A TLB miss will cause at least 4 memory accesses before the actual data is fetched.
  - (d) Modern CPUs use a large TLB with millions of entries to keep the TLB miss rate low.
  - (e) None of the above (i.e., (1)-(4) are all **CORRECT**)

11. ( ) Which of the following statements about **Page-Fault-Frequency (PFF)** algorithm is/are **INCORRECT**?
- (a) PFF relies on page fault rate to determine which pages should be kept in the working set.
  - (b) PFF is designed based on the LFU replacement algorithm. A page fault count is associated with each page.
  - (c) PFF is activated upon every page fault, thus the overhead is quite high.
  - (d) PFF is used in Windows as a global replacement algorithm.
  - (e) None of the above (i.e., (1)-(4) are all **CORRECT**)
12. ( ) Which of the following statements about **malloc()** in Linux is/are **INCORRECT**?
- (a) malloc() only reserves memory and does not allocate any physical memory.
  - (b) The returned pointer of malloc() doesn't point to any physical address at the time the call returns.
  - (c) Memory malloc()'ed by a child process is shared with the parent via *copy-on-write*.
  - (d) The C library function calloc() usually takes longer time than malloc() as it zero-initializes the memory before it returns.
  - (e) None of the above (i.e., (1)-(4) are all **CORRECT**)
13. ( ) Which of the following statements about **swap space** is/are **INCORRECT**?
- (a) When the swap space is full, *thrashing* starts happening.
  - (b) The recommended swap size is 2x the size of RAM for a high-end server with large memory.
  - (c) When swap size is set 0, you won't observe any major page faults during the program execution.
  - (d) Increase the swap space can improve the execution speed of your programs.
  - (e) None of the above (i.e., (1)-(4) are all **CORRECT**)
14. ( ) Which of the following statements about **thrashing** is/are **INCORRECT**?
- (a) *Thrashing* starts happening when the total virtual memory size of all running processes is larger than the available physical memory size.
  - (b) Using faster CPU could reduce *thrashing*.
  - (c) Increase the swap size could reduce *thrashing*.
  - (d) *Thrashing* problem can be avoided by using a faster hard disk.
  - (e) None of the above (i.e., (1)-(4) are all **CORRECT**)
15. ( ) Which of the following statements about **working set** is/are **INCORRECT**?
- (a) A process' working set size shrinks when more processes run in the system.
  - (b) Working set size of a process never changes during its execution.
  - (c) When temporal locality is getting poorer, the working set size starts increasing.
  - (d) LRU-like page replacement algorithms are commonly used to track the working set.
  - (e) None of the above (i.e., (1)-(4) are all **CORRECT**)

## Problem 2: Threads (10%)

The following Pthreads program creates three threads to access three types of resources, namely **A**, **B**, and **C**. The program can be compiled without any error, but could potentially lead to some problem(s).

**Tasks to be done:** (a) Analyze the code and discuss the potential problem(s) based on the *Resource Allocation Graphs* (RAG). (5%). (b) Fix the problem(s) with minimal change of code without introducing any new variables nor adding new functions. Explain why your solution works. (5%)

[Note: you only need to show the re-written part of the code. Don't rewrite the whole program in the answer book.]

```
// We only show portion of the code.
typedef struct _resource_t {
    char name[20];
    pthread_mutex_t m;
} resource_t;

typedef struct _arg_t {
    resource_t* read;
    resource_t* write;
} arg_t;

void* entry_point(void* args) {
    arg_t* arg = (arg_t*)(args);

    pthread_mutex_lock(&arg->read->m);
    pthread_mutex_lock(&arg->write->m);

    // Do something here //
    pthread_mutex_unlock(&arg->write->m);
    pthread_mutex_unlock(&arg->read->m);
    return NULL;
}

void create_resource(resource_t* rs, const char* name) {
    strncpy(rs->name, name, sizeof(rs->name));
    pthread_mutex_init(&rs->m, NULL);
}

int main(int argc, char* argv[]) {
    int rc;
    pthread_t p1, p2, p3;
    resource_t rs1, rs2, rs3;

    create_resource(&rs1, "A");
    create_resource(&rs2, "B");
    create_resource(&rs3, "C");

    arg_t a1 = {&rs1, &rs3};
    arg_t a2 = {&rs3, &rs2};
    arg_t a3 = {&rs2, &rs1};

    rc = pthread_create(&p1, NULL, entry_point, &a1);
    rc = pthread_create(&p2, NULL, entry_point, &a2);
    rc = pthread_create(&p3, NULL, entry_point, &a3);

    rc = pthread_join(p1, NULL);
    rc = pthread_join(p2, NULL);
    rc = pthread_join(p3, NULL);

    return 0;
}
```

Problem 3. Page Replacement Algorithms (20%)

In the Clock algorithm, each page has an associated reference bit (R-bit) that is set whenever the particular page is referenced. Based on the Clock algorithm we discussed in COMP3230, complete the following table and answer the questions (a)-(e). Asterisk (\*) indicates the current R-bit is set to 1. The arrow sign (→) indicates the current position of the clock arm. [Note: We have completed the first few page references for you. Please complete the rest.]

Reference string	2	3	2	1	5	2	4	5	3	5
Frame 1	2*	2*								
Frame 2		3*								
Frame 3										
Fault count	1	2								

- (a) When **page 5** is first referenced, discuss how the Clock algorithm works on the circular list to choose a victim page? What is the new position of the clock arm? (4%) [Note: you are expected to draw the circular list to show how the pages are linked together and the value of R-bit associated with each page **before** and **after** page 5 is referenced.]
- (b) When **page 3** is referenced the second time (near the end), discuss how the Clock algorithm works on the circular list to choose a victim page? What is the new position of the clock arm? (2%)
- (c) What is the total number of page faults? (2%)
- (d) Identify two potential problems when Clock is applied in a real system (e.g., the number of page frames is much larger than 3). (6%) [Hint: you could make inferences based on the observations of the above test case for identifying the potential problems.]
- (e) How to fix the problems you identified in (d)? (6%) [Note: marks will be given based on the feasibility of the proposed solution with the consideration of the implementation cost.]

## Problem 4: Data Locality (20%)

Consider the two versions of matrix multiply programs which multiply two square matrices of size  $n \times n$ . Assume the 2D integer arrays A, B, and C are arranged in **row-major order** and  $n > 500$  (i.e.,  $n$  is large enough). All the three 2D arrays have been allocated in memory and initialized without any problem. Answer the following questions:

- (a) If we run the two versions of code on **workbench** server, can you estimate which version runs faster? Justify your answer. (8%) [Note: we don't expect you to run the code on any machine. Also be aware that the swap size in workbench is 0.]
- (b) Based on the version you selected in (a), is it possible to further improve its execution speed? Rewrite the code snippet **without introducing any new variable** and justify your answer. (12%) [Note: marks will be given based on the level of details of your analyses, not solely on your code.]

**// Version A**

```
int i, j, k, s;
// Arrays: A, B, C are allocated and initialized

for (j = 0; j < n; j++)
    for (k = 0; k < n; k++) {
        s = B[k][j];
        for (i = 0; i < n; i++)
            C[i][j] += A[i][k]*s;
    }
```

**// Version B**

```
int i, j, k, s;
// Arrays: A, B, C are allocated and initialized

for (i = 0; i < n; i++)
    for (j = 0; j < n; j++) {
        s = 0;
        for (k = 0; k < n; k++)
            s += A[i][k]*B[k][j];
        C[i][j] += s;
    }
```

## Problem 5: Semaphores (20%)

Tesla Motors decided to produce electric bicycles in 2021. The new production line consists of 4 types of production jobs: (1) making wheels, (2) making skeletons, (3) making batteries, and (4) assembling bicycles.

**Task 1 (10%):** You will create 4 threads (robots), and each thread will only focus on one type of production jobs. The code snippet below shows the process of each job.

*/\* You are not allowed to modify these functions \*/*

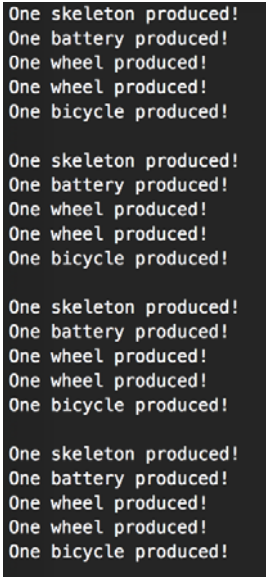
```
void makeWheel() {
    sleep(1);
    printf("One wheel produced!\n");
}

void makeSkeleton() {
    sleep(1);
    printf("One skeleton produced!\n");
}

void makeBattery() {
    sleep(1);
    printf("One battery produced!\n");
}

void makeBicycle() {
    sleep(1);
    printf("One bicycle produced!\n\n");
}
```

Expected output of **Task 1**:



```
One skeleton produced!
One battery produced!
One wheel produced!
One wheel produced!
One bicycle produced!

One skeleton produced!
One battery produced!
One wheel produced!
One wheel produced!
One bicycle produced!

One skeleton produced!
One battery produced!
One wheel produced!
One wheel produced!
One bicycle produced!

One skeleton produced!
One battery produced!
One wheel produced!
One wheel produced!
One bicycle produced!
```

You will use 4 semaphores (*sem0*, *sem1*, *sem2*, and *sem3*) to enforce the production order and correct ration for each type of parts (1 skeleton : 1 battery : 2 wheels). The expected production order is: *skeleton* → *battery* → *wheel* → *wheel* → *bicycle*. The expected output and printing order are shown in the above screenshot. The job IDs are defined as follows: 0 = Wheel, 1 = Skeleton, 2 = Battery, 3 = Bicycle.

To complete Task 1, you only need to set the initial value of each semaphore in the `main()` function and add `sem_wait()` and `sem_post()` function calls (with correct semaphore variables specified) in `robotProduction()` where you think necessary. **You are not allowed to introduce additional semaphores or use other function calls** (e.g., `pthread_mutex_lock/unlock()`, `sleep()`). You can assume the storage space is always available to keep the produced part. Please add the needed code in the answer book we provided and explain how your program could ensure the expected production order (*skeleton* → *battery* → *wheel* → *wheel* → *bicycle*). [Note: we don't expect you to write the code and test on any machine.]



Job ID definition: 0 = Wheel, 1 = Skeleton, 2 = Battery, 3 = Bicycle

```
sem_t sem0, sem1, sem2, sem3;
void *robotProduction(void *arg) {
    int jobID = *(int *)arg;
    while (1) {
        switch (jobID) {
            case 0: {
                /* A: insert necessary code if any*/
                makeWheel();
                /* B: insert necessary code if any*/
                break;
            }
            case 1: {
                /* C: insert necessary code if any*/
                makeSkeleton();
                /* D: insert necessary code if any*/
                break;
            }
            case 2: {
                /* E: insert necessary code if any*/
                makeBattery();
                /* F: insert necessary code if any*/
                break;
            }
            case 3: {
                /* G: insert necessary code if any*/
                makeBicycle();
                /* H: insert necessary code if any*/
                break;
            }
            default:
                printf("Unknown Job ID: %d\n", jobID);
        }
    }
    return NULL;
}

int main(int argc, char *argv[]) {
    /* I: add the initial value of each semaphore */
    sem_init(&sem0, 0, __); /*set initial value of sem0*/
    sem_init(&sem1, 0, __); /*set initial value of sem1*/
    sem_init(&sem2, 0, __); /*set initial value of sem2*/
    sem_init(&sem3, 0, __); /*set initial value of sem3*/

    pthread_t robots[4];
    int jobIDs[4];

    for (int i = 0; i < 4; ++i) {
        jobIDs[i] = i;
        pthread_create(&robots[i], 0, robotProduction, &jobIDs[i]);
    }

    for (int i = 0; i < 4; ++i) {
        pthread_join(robots[i], 0);
    }
    return 0;
}
```

**Task 2 (10%):** Tesla Motors manager finds that restricting the part production order (skeleton→ battery→ wheel→ wheel→ bicycle) isn't efficient as the part production can be done concurrently to improve the production speed. Therefore, the factory decides to introduce 3 new semaphores (**wheel**, **skeleton**, and **battery**) to track the number of produced parts as shown in code snippet below; and uses only **3 semaphores** (**sem0**, **sem1**, and **sem2**) for robots (threads) synchronization to make all robots be able to run concurrently to maximize the production speed, while keeping the production ratio (**wheel : skeleton : battery : bicycle = 2 : 1 : 1 : 1**) in each run to avoid wasted parts. The screenshot below shows the expected outputs and possible printing orders if the program executes correctly.

/\* You are not allowed to modify these functions \*/

```
sem_t wheel, skeleton, battery;
void makeWheel() {
    sleep(1);
    printf("One wheel produced!\n");
    sem_post(&wheel);
}

void makeSkeleton() {
    sleep(1);
    printf("One skeleton produced!\n");
    sem_post(&skeleton);
}

void makeBattery() {
    sleep(1);
    printf("One battery produced!\n");
    sem_post(&battery);
}

void makeBicycle() {
    sem_wait(&wheel);
    sem_wait(&wheel);
    sem_wait(&skeleton);
    sem_wait(&battery);
    sleep(1);
    printf("One bicycle produced!\n\n");
}
```

Expected output of **Task 2**:

```
One battery produced!
One skeleton produced!
One wheel produced!
One wheel produced!
One bicycle produced!

One skeleton produced!
One battery produced!
One wheel produced!
One wheel produced!
One bicycle produced!

One skeleton produced!
One wheel produced!
One battery produced!
One wheel produced!
One bicycle produced!

One skeleton produced!
One battery produced!
One wheel produced!
One wheel produced!
One bicycle produced!

One wheel produced!
One skeleton produced!
One battery produced!
One wheel produced!
One bicycle produced!
```

To complete Task 2, you only need to set the initial value of each semaphore in the main() function and add **sem\_wait()** and **sem\_post()** function calls (with correct semaphore variables specified) in robotProduction() where you think necessary. **You are not allowed to introduce additional semaphores or use other function calls** (e.g., pthread\_mutex\_lock/ unlock(), sleep()). You can assume the storage space is always available to keep the produced parts. Please add the needed code in the answer book we provided, and explain how the **two requirements** could be achieved based on your new design: **(1)** improved production speed with maximal concurrency, **(2)** production ratio (wheel : skeleton : battery : bicycle = 2 : 1 : 1 : 1). [Note: we don't expect you to run the code on any machine.]

Job ID definition: 0 = Wheel, 1 = Skeleton, 2 = Battery, 3 = Bicycle

```
sem_t sem0, sem1, sem2;
void *robotProduction(void *arg) {
    int jobID = *(int *)arg;
    while (1) {
        switch (jobID) {
            case 0: {
                /* A: insert necessary code if any*/
                makeWheel();
                /* B: insert necessary code if any*/
                break;
            }
            case 1: {
                /* C: insert necessary code if any*/
                makeSkeleton();
                /* D: insert necessary code if any*/
                break;
            }
            case 2: {
                /* E: insert necessary code if any*/
                makeBattery();
                /* F: insert necessary code if any*/
                break;
            }
            case 3: {
                /* G: insert necessary code if any*/
                makeBicycle();
                /* H: insert necessary code if any*/
                break;
            }
            default:
                printf("Unknown Job ID: %d\n", jobID);
        }
    }
    return NULL;
}

int main(int argc, char *argv[]) {
    /* I: add the initial value of sem0, sem1, and sem2 */
    sem_init(&sem0, 0, __); /*set initial value of sem0*/
    sem_init(&sem1, 0, __); /*set initial value of sem1*/
    sem_init(&sem2, 0, __); /*set initial value of sem2*/

    /** You are not allowed to change the initial value of wheel,
        skeleton, and battery **/
    sem_init(&wheel, 0, 0);
    sem_init(&skeleton, 0, 0);
    sem_init(&battery, 0, 0);

    pthread_t robots[4];
    int jobIDs[4];

    for (int i = 0; i < 4; ++i) {
        jobIDs[i] = i;
        pthread_create(&robots[i], 0, robotProduction, &jobIDs[i]);
    }

    for (int i = 0; i < 4; ++i) {
        pthread_join(robots[i], 0);
    }
    return 0;
}
```

----- END OF PAPER -----