CS2106: Introduction to Operating Systems, AY 2019/2020 Semester 2
Final Assessment Part 1, time allowed: 60 minutes.

1. A program does a very long computation. You want to parallelize it, and you realize that some parts of the program are ***embarrassingly parallel***, and other parts are inherently serial. You are considering **threading** model **or multi-process (forking)** model as a method of parallelization.

   Select all true statements, if any.

   A. With **threading** it is easier to share state and combine the results of parallelized computations.
   B. With **forking** it is easier to share state and combine the results of parallelized computations.
   C. The advantage of **threading** is that there is less overhead (in terms of worker creation and context switching between workers), in case that the computation is split into more workers than cores on the system.
   D. The advantage of **forking** is that there is less overhead (in terms of worker creation and context switching between different workers), in case that the computation is split into more workers than cores on the system.
   E. The advantage of **threading** is that for the parts of the computation that are done in parallel, if one process crashes the rest can continue while the crashing part is restarted.
   F. The advantage of **forking** is that for the parts of the computation that are done in parallel, if one process crashes the rest can continue while the crashing part is being restarted.

2. You want to build a high-performance network service that can handle many independent connections simultaneously. To implement such a parallel service, you are considering a **threading** model and a **multi-process (forking)** model.

   Select all true statements, if any.

   A. If one worker thread/process is created to handle each new connection, the overhead of establishing a connection would be lower with **threading**.
   B. If handling a network connection could result in a change of globally shared state, it would be easier to keep the state synchronized in case of **threading**.
   C. If handling a network connection could result in a change of globally shared state, it would be easier to keep the state synchronized in case of **forking**.
   D. If one worker thread/process is created to handle each new connection, the overhead of establishing a connection would be lower with **forking**.
   E. **Threading** is more reliable, because if one worker crashes, say due to a bug or malicious connection, the whole service is **not** taken down**.**
   F. **Forking** is more reliable, because if one worker crashes, say due to a bug or malicious connection, the whole service is **not** taken down**.**
   G. **Threading** is more secure, because if one worker is compromised, the attacker cannot as easily read the data of other connections.
   H. **Forking** is more secure, because if one worker is compromised, the attacker cannot as easily read the data of other connections.

The following context applies to the next three questions.

Long-running compute-bound process **Q** has started executing on a single-core **32-bit** Intel x86 processor running Linux. Before loading process **Q**, the system had less than **420 MiB** of free main memory. However, process **Q** needs at least **2.1GiB** of memory to totally eliminate page faults (after reaching a steady state, approximately **42 seconds** after loading). Nevertheless, the process was loaded is about to execute the following code to increment a **local 32-bit integer variable x on the stack:**

```
1. MOV EAX, [EBP-12]; load a variable into a register

2. INC EAX;  increment the value of register EAX

3. MOV [EBP-12], EAX; store the variable to memory
```

Note that Intel processors use the ***write-back*** cache policy, which means that any writes to the data residing in the caches will be propagated to main memory only after the updated data is evicted from the cache. This is in contrast to the ***write-through*** cache policy, in which every update must be propagated to the main memory immediately.

3. After executing the first instruction in the above sequence, where is the variable **likely** to be present? Select all options that apply (if any).

   A. EAX register
   B. EBP register
   C. In CPU caches.
   D. In the swap area of the SSD.
   E. Somewhere on the SSD, but strictly outside of the swap area.
   F. In a DNA-based archival storage system.
   G. On the hard disk.
   H. In DRAM.

4. After executing the second instruction, which location is **likely** to have the **most up-to-date version** of variable x? Select all that applies, if any.

   A. f EAX register
   B. EBP register
   C. In CPU caches.
   D. In the swap area of the SSD.
   E. Somewhere on the SSD, but strictly outside of the swap area.
   F. In a DNA-based archival storage system.
   G. On the hard disk.
   H. In DRAM.

5. After executing the third instruction, which location is **likely** to hold the most **up-to-date** version of variable x? Select all that applies, if any.

   A. EAX register
   B. EBP register
   C. In CPU caches.
   D. In the swap area of the SSD.
   E. Somewhere on the SSD, but strictly outside of the swap area.
   F. In a DNA-based archival storage system.
   G. On the hard disk.
   H. In DRAM.

6. A process has **many** threads, which are about to run **the same short piece of code**. There is a critical section to be protected, but your operating system does **not** provide semaphores. Luckily, your processor provides an atomic instruction called **fetch & add,** and C library provides a convenient wrapper function:

```
int fetch_and_add(int* x);
```

   which compiles into the **fetch & add** instruction. The function atomically adds `1` to an integer located at address `x` and returns the original value at the same location.

   Your task is to synchronize the threads such that mutual exclusion is guaranteed with minimal busy waiting. You are allowed to use **at most three global helper integer** variables, which you first need to declare and initialize. Then fill in the code snippet below using `fetch_and_add()` as your main tool to enforce the mutual exclusion.

```
// your code here (critical section entry)

/*----------------------------------------

Critical Section

----------------------------------------*/

// your code here (critical section exit)
```

   Separate your code into the 3 portions (helper variable initialization & declaration, critical section entry, critical section exit) using two horizontal lines like this:

   ----

   and **pay attention to syntax** because the question will be **auto-graded**.

(Hint: The idea here is similar to the queuing solution for bubble tea. You get a ticket number and you get to collect your drink only when your ticket number shows up on the screen.) Good luck!

7. Prof's process attempts to read from an invalid (i.e., unmapped) memory address on the lab machine (Linux, x86_64). What is certain to happen next? Select all options that apply, if any.

   A. The process receives a signal.
   B. The process is terminated by the OS.
   C. The process reads garbage and continues.
   D. The process reads garbage and continues, but an error is reported at the next system call.
   E. The OS will prevent the process from reading data from another process' virtual address space.

8. Consider the following snippet of C code:

```
int a;
int b[16];
int c = 123;

int random(void) {
  return 4;
}

int main(int argc, char *argv[]) {
  int d = 4;
  int *e = malloc(sizeof(int));
  int *f = b;
  return *e + *f;
}
```

In which memory segment is the following data located?

Enter only **`text`**, **`bss`**, **`data`**, **`heap`**, **`stack`**.

1. **`a:`**
2. **`b[0]:`**
3. **`c:`**
4. **`d:`**
5. **`e:`**
6. **`*e:`**
7. **`f:`**
8. **`*f:`**
9. **`argc:`**
10. **`argv:`**
11. **`random:`**

Note: ensure you do not have any typos or extraneous whitespace.

9. In Lab 4, you saw the use of `brk`/`sbrk` Linux system calls to allocate memory. Why do normal programs typically use `malloc` instead of `brk`/`sbrk`?

10. In Lab 4, you saw the use of `brk`/`sbrk` Linux system calls to allocate memory. What would happen if a program mixed calls to `malloc` and `brk`/`sbrk`?

11. Two single-core systems, A and B, are very similar. Both of them have 32MB of main memory and both systems require the entire program to be present in memory at any time, as they do not support virtual memory. However, system A uses paging with 4KB pages, whereas system B uses segmentation. It is observed that System B achieves slightly higher performance across a number of benchmarks.

Compare the number of TLB entries in systems A and B and explain your answer.

12. ________________ is the common design principle behind techniques such as copy-on-write and demand paging.

13. Many systems nowadays use **small page sizes (e.g., 4 KB)**. However, some systems allow for **larger page sizes. e.g., 8KB, 2 MB, 1 GB....** What are the effects of using a **larger page size**? Select all answers that apply, if any. You can assume that the underlying system (e.g., hardware, page table format, etc.) is properly adapted to the larger page size, but the user programs remain the same.

    A. Increased TLB pressure (more TLB entries required)
    B. Decreased TLB pressure (less TLB entries required)
    C. Increased external fragmentation
    D. Decreased external fragmentation
    E. Increased internal fragmentation
    F. Decreased internal fragmentation
    G. Increased virtual address space
    H. Decreased virtual address space
    I. Increased addressable physical memory
    J. Decreased addressable physical memory
    K. Increased page table size
    L. Decreased page table size
    M. Increased latency of page-table walks
    N. Decreased latency of page-table walks
    O. Improved secondary storage performance.
    P. Degraded secondary storage performance.

14. In modern Intel/AMD/ARM processors, there is usually a separate TLB for instruction accesses (iTLB) and data accesses (dTLB). Give two reasons behind this design decision.

CS2106: Introduction to Operating Systems, AY 2019/2020 Semester 2 Final Assessment Part 2, time allowed: 60 minutes.

1. The Linux machines used in the labs have 64-bit processors, but the virtual addresses are 48-bit long. The page size is 4KiB. The system uses a hierarchical direct page table structure, with each table/directory entry occupying 64 bits regardless of the level in the hierarchy. How many levels are there in the page-table structure?
   A. 2
   B. 3
   C. 4
   D. 5
   E. 6
   F. 8
   G. Cannot be determined based on the given information.

2. Assume Process P runs the following simple program on one of the lab machines:

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #define ONE_GIG 1<<30
4.  #define NUM_PAGES 1<< 18
5.  void main(){
6.    char* array = malloc(ONE_GIG);
7.    int i;
8.    for(i=0; i<NUM_PAGES; i++) array[i<<12]='a';
9.    printf("0x%lX\n", array);
10.   // point of interest
11.   free(array);
12. }
```

   The program prints out the following line, which is as the hexadecimal representation of the value stored in variable **`array`**:

   0x7F9B55226010

   When the program execution reaches the point of interest (line 10), in total ____________ **entries in the root directory** are holding information related to process P's **dynamically allocated data**. The answer must be a non-negative integer, or an arithmetic integer expression containing "*" , "-", "+" that can be evaluated in C (e.g., **31*42-53** is an acceptable answer format).

3. With reference to the setup in the previous question, when the program execution reaches the point of interest (line 10), in total ____________ **physical frames** are holding information related to process P's dynamically allocated data in the **second level** of the hierarchical page-table structure (next after the root). The answer must be a non-negative

integer, or an integer expression containing "*" ,"-", "+" that can be evaluated in C (e.g., **31*42-53** is an acceptable answer format).

4. With reference to the setup in the previous question, when the program execution reaches the point of interest (line 10), in total ___________ **physical frames** are holding information related to process P's dynamically allocated data in the **penultimate level of the page-table structure** (i.e., the level before the last one). Expressions containing *, -, + that can be evaluated in C are allowed (e.g., **31*42-53** is an acceptable answer format).

5. With reference to the setup in the previous question, when the program execution reaches the point of interest (line 10), in total __________ **physical frames** are holding information related to process P's dynamically allocated data in the **last level of the page-table structure.** Expressions containing *, -, + that can be evaluated in C are allowed (e.g., **31*42-53** is an acceptable answer format).

6. A machine uses **3-level** paging to organize its virtual memory system. The virtual address space is **16 bits** and the RAM is **4KiB** in size. A page is **32 bytes** in size. A page table entry or page directory entry is **8 bytes** in size.

   The highest-level page directory (the one that is referenced first) of a process occupies ____________ pages.

7. Shown below is the pseudocode for a disk scheduling algorithm.

```
Procedure schedule(requests, head):
    1. current_direction = current direction of head
    2. current_position = current head position
    3. while there are requests in current_direction from
current_position:
                service the request
                update current_position
    4. reverse current_direction
    5. go to step 3
```

Given the following arrival time information, determine the order of requests served according to the above scheduling algorithm. Initial head position = 30, Direction = Descending direction. Assume each request takes 1 time unit.

Time	0	0	2	2	2	2	10	10	13
Request (track #)	31	40	62	14	25	80	43	1	90

(Fill in the blanks): ____ , ____ , ____ , ____ , ____ , ____ , ____ , ____ , ____

8. It is observed that the above algorithm results in **unfair waiting times** for some requests. What changes would you propose to the algorithm to **minimize** this issue?

9. The disk takes considerably longer time to complete write operations compared to read operations. What is the implication of this latency gap between read and write operations on the performance of the above algorithm?

10. Suggest **two** approaches for tackling the issue of **read-write performance gap** and improving the overall performance of servicing disk requests.

11. Over time, it is common for secondary storage disks to develop corrupted storage areas known as **bad sectors**. The operating system tries to mitigate the impact of bad sectors upon detection by **utilizing non-corrupted blocks** instead of corrupted blocks in bad sectors.
    Which of the following is **true** (if any) regarding **file block allocation schemes** in the presence of bad sectors?

    A. Contiguous block allocation is convenient to use because changing start block and end block will easily get rid of corrupted blocks.
    B. Contiguous block allocation is more efficient than linked list allocation in mitigating corrupted blocks.
    C. Linked list allocation is efficient in mitigating corrupted blocks as they could be easily replaced by redirecting list pointers.
    D. FAT allocation scheme is less efficient than regular linked list allocation in mitigating corrupted blocks because in FAT the linked list is traversed in memory.
    E. FAT allocation scheme is more efficient than regular linked list allocation in mitigating corrupted blocks because in FAT the linked list is traversed in memory.

12. In tutorial 10, we explored **buffered reads and writes**. As a short recap, the following is the excerpt from the question (you do not need information in this excerpt to solve the question, but it can help you understand the code):

    Most high-level programming languages therefore provide buffered file operations that wrap around primitive file operations. The buffered version maintains an internal intermediate storage in memory (i.e., buffer) to store values read from/written to the file by the user. For example, a buffered write operation will wait until the internal memory buffer is full before doing a large one-time file write operation to flush the buffer content into file.

    In the following code snippet, we try to implement a set of functions to do something similar. To read a file **byte-by-byte**, we utilize a buffer of size 2 bytes to reduce to number of "read()" operations by approximately half.

```
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>

int fill_buffer(int fd, char* buf){
   return (int) read(fd, buf, 2);
}

char read_char(int fd, char* buf, int* buf_ptr){
    int chars_filled;
    if (*buf_ptr >= 2){
        chars_filled = fill_buffer(fd, buf);
        if (chars_filled == 0){
            return '#'; //To indicate the end of file
        } else if (chars_filled == 1){
            *buf_ptr = 2;
            return buf[0];
        } else{
            *buf_ptr = 1;
```

```
                return buf[0];
            }
        } else {
            *buf_ptr = *buf_ptr + 1;
            return buf[*buf_ptr - 1];
        }
    }
```

The following is an example run using the functions. The file "secret" is **exactly** 4 bytes long and contains "abcd".

```
    int main(){

        char buf[2];
        int buf_ptr = 2 ;
        char id = 'P';
        int fd = open("./secret", O_RDONLY);

        /* Also fills the buffer for the first time*/
        char c = read_char(fd, buf, &buf_ptr);
        printf("%c: %c \n", id, c);

        if (fork() == 0) id = 'C';

        else  id = 'P';

        for (int i = 0; i < 3; i++){
            c = read_char(fd, buf, &buf_ptr);
            printf("%c: %c\n", id, c);
        }
    }
```

Give a possible **output sequence**, and explain your answer. The first line of output is given to you below:

P: a

The following context applies to the next three questions:

We discussed three techniques for allocation of disk blocks for files: **contiguous, linked-list, and indexed allocation**. **Rank** the three allocation techniques according to their suitability for various user cases below, **and justify your answer.**

13. Use Case #1: You want to maximize the performance of sequential accesses to very large files.

14. Use Case #2: You want to maximize the performance of random access to very large files.

15. Use Case #3: You want to maximize the utilization of the disk capacity (i.e., getting the most file bytes on the disk).

16. Process A reads from a Unix **pipe** using the `read` syscall. What are the possible scenarios? Select all answers that apply (if any), and note that each scenario is independent of each other. Assume that the scheduler does not pre-empt the CPU from process A.

    A. Process A transitions to state **BLOCKED**. The syscall eventually returns with no data, and process A continues running.
    B. Process A transitions to state **BLOCKED**. The syscall eventually returns with some data, and process A continues running.
    C. Process A does **not** transition state **BLOCKED**. It receives no data from the syscall and continues running.
    D. Process A does **not** transition to state **BLOCKED**. It receives some data from the syscall and continues running.

17. Prof's process makes a `read` system call on Linux, but passes in an **invalid argument**. What is certain to happen next? Select all correct answers, if any.
    A. The process receives a signal.
    B. The process crashes.
    C. The process is killed.
    D. The system call returns with an error.

18. Which of these operations cannot execute without involving the kernel? Select all answers that apply, if any. Assume the operations are performed on Linux.
    A. Calling a function defined in the same program.
    B. Reading a file from the disk.
    C. Writing to a file on the disk.
    D. Calling `malloc()` to allocate memory.
    E. Communicating with another thread using a global variable.
    F. Communicating with another process using already-existing shared memory.
    G. Communicating with another process using an already-existing pipe.
    H. Waiting for approximately 30 seconds to pass.

END OF ASSESSMENT