CS 219 Spring 2025 Quiz 2

(6 questions, 30 marks, 15% weightage)

Name:________________________ Roll number:________________________

1. **[4 marks]** Suppose we wish to implement a copy-on-write fork in a simple OS like xv6. Fill in the steps required to do so in the questions below. You may assume that there are no read-only pages in the virtual address space of the parent process before it forks a child. Write your answer briefly in one or two sentences.

   (a) During the fork system call, we begin with a skeleton page table for the child process, which has all the kernel page table mappings. Then we go over every entry in the user part of the page table of the parent process, and add a corresponding entry to the child's page table. For every page table entry `pte` in the parent page table that maps virtual address `va` to physical address `pa` with permission `flags`, describe the entry that is added to the child's page table (mention the address mappings as well as permissions).

   Ans: PTE mapping same VA to same PA is added to child page table, but with read-only permission.

   (b) Describe the changes made to the parent page table entries in the above step.

   Ans: Every page table entry of the parent process is also marked as read-only.

   (c) Describe the changes made to the trap handling code for handling page faults, in order to correctly implement copy-on-write fork.

   Ans: When parent or child tries to write to any page, it traps to the OS. OS allocates a new copy of the page and maps it into the page table of the process that trapped.

   (d) When are the changes made to the parent page table (listed in part (b) above) rolled back, i.e., when are the parent page table entries restored to the state they were in before fork?

   Ans: Once every process has its own copy of the page, the read-only restriction can be removed, and the parent page table goes back to its original state.

2. **[5 marks]** Consider a system running 32-bit xv6 with 4KB pages. Assume that the operating system image is loaded into the first 4MB of physical memory, and is mapped into the virtual address space of every process starting at high virtual address 2GB (KERNBASE). The total physical memory available to xv6 is 32MB (PHYSTOP). Two processes P1 and P2 have been created in this system, each having 3 pages (for the code, guard page, and stack, in that order) in the user part of their address space.

   (a) How many free frames are present in the free list of the OS initially, before any processes are created in the system? You may assume that all free physical memory that the OS can address is part of the free list initially.

   Ans: 7168 (32MB = 8192 pages total, out of which 1024 are for OS, which means 7168 frames are part of the free list)

(b) How many valid/present pages does the virtual address space of a process (P1 or P2) have, across the user and kernel parts of the address space?

Ans: 3 (user) + 8192 (all of 0 to phystop is mapped in kernel space) = 8195

(c) Consider a physical frame at address 2MB, which holds some part of the OS image. At which virtual address is this frame mapped in the page tables of P1 and P2? List all mappings to this frame.

Ans: It is mapped at address 2GB + 2MB in the page tables of P1 and P2

(d) Consider a physical frame at address 4MB + 8KB (i.e., the 3rd frame after the OS image), which is allocated to hold the stack page (i.e., the 3rd page in the user part of the memory image) of process P1. At which virtual address is this frame mapped in the page tables of P1 and P2? List all mappings to this frame.

Ans: It mapped at virtual address 8KB in P1, and also at 2GB+4MB+8KB in the page tables of P1 and P2

(e) Consider the page table of P1. Recall that the inner page table pages are allocated on demand, i.e., only when there is at least one valid entry to store. How many pages does the OS allocate to build the page table of P1, across both the inner and outer levels?

Ans: 8 pages for kernel mappings (each holds 1024 PTE, so maps 4MB of address space) + 1 for user + 1 outer pgdir = 10 pages in total

3. **[6 marks]** Consider a context switch from process P1 to process P2 via the scheduler thread in xv6. Process P1 has a pipe open into which it reads and writes sometimes. Assume that the scheduler thread finds the ready process P2 in the ptable without having to release the `ptable.lock`. Also assume there are no other locks involved during this period of the context switch, besides the ptable lock and (maybe) the pipe lock. Below are listed several events that may happen during the context switch from P1 to P2.

E1. `ptable.lock` is acquired

E2. `ptable.lock` is released

E3. The pipe lock of P1's pipe is acquired

E4. The pipe lock of P1's pipe is released

E5. The `sleep` function is called by P1

E6. The `sched` function is called by P1, which then calls `swtch` to switch to the scheduler thread

E7. The scheduler thread calls `swtch` to switch to process P2

For each of these scenarios described below, list the subset of events that occur during the context switch from P1 to P2 in chronological order (earliest to latest).

(a) P1 receives a timer interrupt and is switched out. (It did not perform any pipe-related system calls.)

Ans: E1, E6, E7, E2

(b) P1 reads on an empty pipe, gets blocked, and is switched out.

Ans: E3, E5, E1, E4, E6, E7, E2

(c) P1 invokes the wait system call to reap a child process that is still running, gets blocked, and is switched out.

Ans: E1, E5, E6, E7, E2

4. **[5 marks]** Consider a program that handles 3 types of requests (let us call them P, Q, and R) that arrive at random times. A new thread is created to handle each request. The program batches requests into groups of three (one each of P, Q, R) and processes them together. A thread waits until the corresponding requests of the other types arrive, and three "matched" threads can run together. The threads must be matched whenever possible, and a thread must not wait unnecessarily if the matching threads have already arrived.

   You are given pseudocode that synchronizes the threads handling the requests, where the synchronization is done using three semaphores, `semp`, `semq`, `semr`, all initialized to 0. The code for a thread handling request P is given below, and the code for requests Q and R is symmetric.

```
up(semp)
up(semp)
down(semq)
down(semr)
```

   Is the code shown above correct? Answer yes or no, with suitable justification. If you answer yes, provide a brief explanation for why the code works. If you answer no, describe one example scenario where the code fails to correctly match threads or leads to some type of deadlock.

   Ans: The code is incorrect. While it may work fine in simple cases (P, Q, R arrive one after other, for example), it fails in some cases. Consider the scenario where two P, and one each of Q, R arrive. By the time R arrives, 2 P and one are Q waiting in down R. If two P get released by the two up operations of R, then Q and R deadlock. So, we will not be able to match one set of P, Q, R, even though we could have. Think about what the correct solution to this question is — it may be an endsem question!

5. **[4 marks]** Consider an application that stores key-value pairs in a hash table. The application receives requests to get (i.e., read) or put (i.e., write) key-value pairs into the hash table. Every request is handled by a separate thread. In order to improve performance, the program allows multiple threads handling get requests to concurrently access the hash table, because a get request only reads the data structure. However, whenever a thread handling a put request needs to access the hash table, it must do so exclusively, without executing concurrently with any other get or put thread. Further, because the application wishes to avoid reading stale data, it must ensure that put requests are prioritized over get requests, when multiple requests are waiting to access the hash table. That is, put requests must not wait for access any longer than they need to. You are given incomplete pseudocode for the threads that handle get and put requests below. You must complete the code to ensure the synchronization described above between the threads happens correctly.

   You must use only the following variables in your solution: a mutex lock `L`, two condition variables `CVGet` and `CVPut`, and three counters `getcount`, `putcount` and `putwaiting`, all initialized to 0.

   (a) Complete the code for a thread handling a get request below.

```
lock(L)




unlock(L)

//Access hash table to get

lock(L)




unlock(L)
```

(b) Complete the code for a thread handling a put request below.

```
lock(L)



unlock(L)

//Access hash table to put

lock(L)



unlock(L)
```

Ans: This is similar to the reader-writer locks discussed in class, with writer preference.

```
//code for get
lock(L)
if(putcount>0 || putwaiting>0) wait(CVGet, L)
getcount++
unlock(L)
//Access hash table to get
lock(L)
getcount--
if(getcount==0) signal(CVPut)
```

```
unlock(L)

//code for put
lock(L)
putwaiting++
if(getcount>0 || putcount>0) wait(CVPut, L)
putwaiting--
putcount++
unlock(L)
//Access hash table to put
lock(L)
putcount--
if(putwaiting>0) signal(CVPut)
else signal_broadcast(CVGet)
unlock(mutex)
```

6. **[6 marks]** Complete the synchronization logic in the questions below using semaphores. The semaphores you must use in your solution are already declared for you. You must not use any other variables other than those provided in the question.

   (a) A trading engine has two types of requests coming in, to buy and sell stocks. Every request waits until it is paired up with a request of the opposite type. That is, a buy request must wait till it can be paired up with a sell request and vice versa. Each request is handled by a separate thread, the requests can arrive in any order, and any buy request can be paired up with any sell request. Complete the code for the thread handling a buy request below, using two semaphores `sembuy` and `semsell`, both initialized to 0. You can assume that the code for the thread handling sell requests will be symmetrical, with the buy and sell semaphores swapped. Your code must ensure that the thread waits until paired up suitably.

   ```
   //complete code for buy thread
   ```

   Ans: The other symmetric answer is also possible.

   ```
   up(sembuy)
   down(semsell)
   ```

   (b) A program receives multiple requests which are handled by separate threads. The program batches requests into groups of three before processing. So threads that arrive must wait until they can form a group of three, after which they can complete execution. Complete the code that each thread must run at the start of its execution to achieve this synchronization. You must use two semaphores in your solution: semaphore `waitfor3` (initialized to 0) and semaphore `mutex` (initialized to 1). This `mutex` semaphore acts as a lock to protect a counter `count` (initialized to 0). You must write your solution only between the `down(mutex)` and `up(mutex)` statements below.

```
down(mutex)
count++
```

```
up(mutex)
down(waitfor3)
```

Ans:

```
down(mutex)
count++
if(count %3 == 0)
  do 3 times: up(waitfor3)
  //optional to decrement count by 3
up(mutex)
down(waitfor3)
```

(c) Consider a program that has two types of threads: multiple worker threads and one batch processing thread. Worker threads are spawned to handle specific tasks, which must be completed in batches of size N. So worker threads that arrive must first wait to be grouped into batches. The batch processing thread runs periodically from time to time on its own (it need not be woken up by the worker threads), and processes a batch of N worker threads in each round. After it wakes up the N worker threads in a batch, it prints a message indicating that the batch has been completed.

The synchronization logic for the worker threads is given below, which you must not change. Each worker waits in a semaphore and prints a message when woken up. You must complete the corresponding code for the batch processor thread. The final message of the batch processor thread must print after all workers have woken up and printed their messages. You may assume that there are a large number of worker threads waiting always, so the batch processor thread will always find N worker threads to wakeup and execute in the batch.

You are given semaphores `worker` and `batch`, both initialized to 0. You may use a local loop counter in your solution if required. You must not use any other variables.

```
//worker thread code below; do not modify

down(worker)
print ``I am awake!''
up(batch)

//batch processor thread; complete code below
```

```
//this statement must print after N workers are woken up
print "Batch completed"
```

Ans: (Can also do the up worker and down batch in separate loops)

```
do N times {
  up(worker)
  down(batch)
}
```
