Lecture Notes on Operating Systems
Problem Set: Concurrency

1. Answer yes/no, and provide a brief explanation.

   (a) Is it necessary for threads in a process to have separate stacks? *yes*

   (b) Is it necessary for threads in a process to have separate copies of the program executable? *No*

2. Can one have concurrent execution of threads/processes without having parallelism? If yes, describe how. If not, explain why not.

3. Consider a multithreaded webserver running on a machine with N parallel CPU cores. The server has M worker threads. Every incoming request is put in a request queue, and served by one of the free worker threads. The server is fully saturated and has a certain throughput at saturation. Under which circumstances will increasing M lead to an increase in the saturation throughput of the server?

4. Consider a process that uses a user level threading library to spawn 10 user level threads. The library maps these 10 threads on to 2 kernel threads. The process is executing on a 8-core system. What is the maximum number of threads of a process that can be executing in parallel?

5. Consider a user level threading library that multiplexes $N > 1$ user level threads over $M \geq 1$ kernel threads. The library manages the concurrent scheduling of the multiple user threads that map to the same kernel thread internally, and the programmer using the library has no visibility or control on this scheduling or on the mapping between user threads and kernel threads. The N user level threads all access and update a shared data structure. When (or, under what conditions) should the user level threads use mutexes to guarantee the consistency of the shared data structure?

   A. Only if $M > 1$.

   B. Only if $N \geq M$.

   C. Only if the M kernel threads can run in parallel on a multi-core machine.

   D. User level threads should always use mutexes to protect shared data.

6. Creating user level threads in a Linux application via any threading library always leads to the creation of corresponding kernel-level threads. [T/F]

7. Consider a Linux application with two threads T1 and T2 that both share and access a common variable x. Thread T1 uses a `pthread` mutex lock to protect its access to x. Now, if thread T2 tries to write to x without locking, then the Linux kernel generates a trap. [T/F]

8. In a single processor system, the kernel can simply disable interrupts to safely access kernel data structures, and does not need to use any spin locks. [T/F]

9. In the `pthread` condition variable API, a process calling wait on the condition variable must do so with a mutex held. State one problem that would occur if the API were to allow calls to wait without requiring a mutex to be held.

10. Consider N threads in a process that share a global variable in the program. If one thread makes a change to the variable, is this change visible to other threads? (Yes/No)

11. Consider N threads in a process. If one thread passes certain arguments to a function in the program, are these arguments visible to the other threads? (Yes/No)

12. Consider a user program thread that has locked a pthread mutex lock (that blocks when waiting for lock to be released) in user space. In modern operating systems, can this thread be context switched out or interrupted while holding the lock? (Yes/No)

13. Repeat the previous question when the thread holds a pthread spinlock in user space.

14. Consider a process that has switched to kernel mode and has acquired a spinlock to modify a kernel data structure. In modern operating systems, will this process be interrupted by external hardware before it releases the spinlock? (Yes/No)

15. Consider a process that has switched to kernel mode and has acquired a spinlock to modify a kernel data structure. In modern operating systems, will this process initiate a disk read before it releases the spinlock? (Yes/No)

16. When a user space process executes the wakeup/signal system call on a pthread condition variable, does it always lead to an immediate context switch of the process that calls signal (immediately after the signal instruction)? (Yes/No)

17. Several processes wish to read and write data shared between them. Some processes only want to read, while others want to update the shared data. Multiple readers may concurrently access the data. However, a writer must not access the data concurrently with anyone else, either a reader or a writer. Write down pseudocode to implement the functions readLock, readUnlock, writeLock, and writeUnlock that are invoked by the readers and writers to realize read/write locks. You must use condition variables and mutexes only in your solution.

18. Consider the readers and writers problem discussed above. Recall that multiple readers can be allowed to read concurrently, while only one writer at a time can access the critical section. Write down pseudocode to implement the functions readLock, readUnlock, writeLock, and writeUnlock that are invoked by the readers and writers to realize read/write locks. You must use **only** semaphores, and no other synchronization mechanism, in your solution. Further, you must avoid using more semaphores than is necessary. Clearly list all the variables (semaphores, and any other flags/counters you may need) and their initial values at the start of your solution. Use the notation `down(x)` and `up(x)` to invoke atomic down and up operations on a semaphore `x` that are available via the OS API. Use sensible names for your variables.

19. Consider the readers and writers problem as discussed above. We wish to implement synchronization between readers and writers, while giving **preference to writers**, where no waiting writer should be kept waiting for longer than necessary. For example, suppose reader process R1 is actively reading. And a writer process W1 and reader process R2 arrive while R1 is reading. While it might be fine to allow R2 in, this could prolong the waiting time of W1 beyond the absolute minimum of waiting until R1 finishes. Therefore, if we want writer preference, R2 should not be allowed before W1. Your goal is to write down pseudocode for read lock, read unlock, write lock, and write unlock functions that the processes should call, in order to realize read/write locks with writer preference. You must use only simple locks/mutexes and conditional variables in your solution. Please pick sensible names for your variables so that your solution is readable.

20. Write a solution to the readers-writers problem with preference to writers discussed above, but using only semaphores.

21. Consider the famous dining philosophers' problem. N philosophers are sitting around a table with N forks between them. Each philosopher must pick up both forks on her left and right before she can start eating. If each philosopher first picks the fork on her left (or right), then all will deadlock while waiting for the other fork. The goal is to come up with an algorithm that lets all philosophers eat, without deadlock or starvation. Write a solution to this problem using condition variables.

22. Consider a clinic with one doctor and a very large waiting room (of infinite capacity). Any patient entering the clinic will wait in the waiting room until the doctor is free to see her. Similarly, the the doctor also waits for a patient to arrive to treat. All communication between the patients and the doctor happens via a shared memory buffer. Any of the several patient processes, or the doctor process can write to it. Once the patient "enters the doctors office", she conveys her symptoms to the doctor using a call to `consultDoctor()`, which updates the shared memory with the patient's symptoms. The doctor then calls `treatPatient()` to access the buffer and update it with details of the treatment. Finally, the patient process must call `noteTreatment()` to see the updated treatment details in the shared buffer, before leaving the doctor's office. A template code for the patient and doctor processes is shown below. Enhance this code to correctly synchronize between the patient and the doctor processes. Your code should ensure that no race conditions occur due to several patients overwriting the shared buffer concurrently. Similarly, you must ensure that the doctor accesses the buffer only when there is valid new patient information in it, and the patient sees the treatment only after the doctor has written it to the buffer. You must use **only semaphores** to solve this problem. Clearly list the semaphore variables you use and their initial values first. Please pick sensible names for your variables.

(a) Semaphore variables and initial values:

(b) Patient process:

```
consultDoctor();


noteTreatment();
```

(c) Doctor process:
```
while(1) {

treatPatient();

}
```

23. Consider a producer-consumer situation, where a process P produces an integer using the function `produceNext()` and sends it to process C. Process C receives the integer from P and consumes it in the function `consumeNext()`. After consuming this integer, C must let P know, and P must produce the next integer only after learning that C has consumed the earlier one. Assume that P and C get a pointer to a shared memory segment of 8 bytes, that can store any two 4-byte integer-sized fields, as shown below. Both fields in the shared memory structure are zeroed out initially. P and C can read or write from it, just as they would with any other data object. Briefly describe how you would solve the producer-consumer problem described above, using *only* this shared memory as a means of communication and synchronization between processes P and C. You must not use any other synchronization or communication primitive. You are provided template code below which gets a pointer to the shared memory, and produces/consumes integers. You must write the code for communicating the integer between the processes using the shared memory, with synchronization logic as required.

```
struct shmem_structure {
int field1;
int field2;
};
```

(a) Producer:

```
struct shmem_structure *shptr = get_shared_memory_structure();

while(1) {
int produced = produceNext();



}
```

(b) Consumer:

```
struct shmem_structure *shptr = get_shared_memory_structure();

while(1) {
int consumed; //fill this value from producer


consumeNext(consumed);



}
```

24. Consider a multithreaded banking application. The main process receives requests to tranfer money from one account to the other, and each request is handled by a separate worker thread in the application. All threads access shared data of all user bank accounts. Bank accounts are represented by a unique integer account number, a balance, and a lock of type `mylock` (much like a `pthreads` mutex) as shown below.

```
struct account {
int accountnum;
int balance;
mylock lock;
};
```

Each thread that receives a transfer request must implement the transfer function shown below, which transfers money from one account to the other. Add correct locking (by calling the `dolock(&lock)` and `unlock(&lock)` functions on a `mylock` variable) to the tranfer function below, so that no race conditions occur when several worker threads concurrently perform transfers. Note that you must use the fine-grained per account lock provided as part of the account object itself, and not a global lock of your own. Also make sure your solution is deadlock free, when multiple threads access the same pair of accounts concurrently.

```
void transfer(struct account *from, struct account *to, int amount) {

from->balance -= amount; // dont write anything...
to->balance += amount; // ...between these two lines

}
```

25. Consider a process with three threads A, B, and C. The default thread of the process receives multiple requests, and places them in a request queue that is accessible by all the three threads A, B, and C. For each request, we require that the request must first be processed by thread A, then B, then C, then B again, and finally by A before it can be removed and discarded from the queue. Thread A must read the next request from the queue only after it is finished with all the above steps of the previous one. Write down code for the functions run by the threads A, B, and C, to enable this synchronization. You can only worry about the synchronization logic and ignore the application specific processing done by the threads. You may use any synchronization primitive of your choice to solve this question.

26. Consider two threads A and B that perform two operations each. Let the operations of thread A be A1 and A2; let the operations of thread B be B1 and B2. We require that threads A and B each perform their first operation before either can proceed to the second operation. That is, we require that A1 be run before B2 and B1 before A2. Consider the following solutions based on semaphores for this problem (the code run by threads A and B is shown in two columns next to each other). For each solution, explain whether the solution is correct or not. If it is incorrect, you must also point out why the solution is incorrect.

(a)
```
sem A1Done = 0;   sem B1Done = 0;
//Thread A                 //Thread B
A1                         B1
down(B1Done)               down(A1Done)
up(A1Done)                 up(B1Done)
A2                         B2
```

(b)
```
sem A1Done = 0;   sem B1Done = 0;
//Thread A                 //Thread B
A1                         B1
down(B1Done)               up(B1Done)
up(A1Done)                 down(A1Done)
A2                         B2
```

(c)
```
sem A1Done = 0;   sem B1Done = 0;
//Thread A                 //Thread B
A1                         B1
up(A1Done)                 up(B1Done)
down(B1Done)               down(A1Done)
A2                         B2
```

27. Now consider a generalization of the above problem for the case of N threads that want to each execute their first operation before any thread proceeds to the second operation. Below is the code that each thread runs in order to achieve this synchronization. `count` is an integer shared variable, and `mutex` is a mutex binary semaphore that protects this shared variable. `step1Done` is a semaphore initialized to zero. You are told that this code is wrong and does not work correctly. Further, you can fix it by changing it slightly (e.g., adding one statement, or rearranging the code in some way). Suggest the change to be made to the code in the snippet below to fix it. You must use only semaphores and no other synchronization mechanism.

```
//run first step

down(mutex);
count++;
up(mutex);
if(count == N)
        up(step1Done);
down(step1Done);

//run second step
```

28. The cigarette smokers problem is a classical synchronization problem that involves 4 threads: one agent and three smokers. The smokers require three ingredients to smoke a cigarette: tobacco, paper, and matches. Each smoker has one of the three ingredients and waits for the other two, smokes the cigar once he obtains all ingredients, and repeats this forever. The agent repeatedly puts out two ingredients at a time and makes them available. In the correct solution of this problem, the smoker with the complementary ingredient should finish smoking his cigar. Consider the following solution to the problem. The shared variables are three semaphores `tobacco`, `paper` and `matches` initialized to 0, and semaphore `doneSmoking` initialized to 1. The agent code performs `down(doneSmoking)`, then picks two of the three ingredients at random and performs `up` on the corresponding two semaphores, and repeats. The smoker with tobacco runs the following code in a loop.

```
down(paper)
down(matches)
//make and smoke cigar
up(doneSmoking)
```

Similarly, the smoker with matches waits for tobacco and paper, and the smoker with paper waits for tobacco and matches, before signaling the agent that they are done smoking. Does the code above solve the synchronization problem correctly? If you answer yes, provide a justification for why the code is correct. If you answer no, describe what the error is and also provide a correct solution to the problem. (If you think the code is incorrect and are providing another solution, you may change the code of both the agent and the smokers. You can also introduce new variables as necessary. You must use only semaphores to solve the problem.)

29. Consider a server program running in an online market place firm. The program receives buy and sell orders for one type of commodity from external clients. For every buy or sell request received by the server, the main process spawns a new buy or sell thread. We require that every buy thread waits until a sell thread arrives, and vice versa. A matched pair of buy and sell threads will both return a response to the clients and exit. You may assume that all buy/sell requests are identical to each other, so that any buy thread can be matched with any sell thread. The code executed by the buy thread is shown below (the code of the sell thread would be symmetric). You have to write the synchronization logic that must be run at the start of the execution of the thread to enable it to wait for a matching sell thread to arrive (if none exists already). Once the threads are matched, you may assume that the function `completeBuy()` takes care of the application logic for exchanging information with the matching thread, communicating with the client, and finishing the transaction. You may use any synchronization technique of your choice.

```
//declare any variables here

buy_thread_function:
  //start of sync logic


  //end of sync logic
  completeBuy();
```

30. Consider the following classical synchronization problem called the barbershop problem. A barbershop consists of a room with N chairs. If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop. If the barber is busy, but chairs are available, then the customer sits in one of the free chairs and awaits his turn. The barber moves onto the next waiting seated customer after he finishes one hair cut. If there are no customers to be served, the barber goes to sleep. If the barber is asleep when a customer arrives, the customer wakes up the barber to give him a hair cut. A waiting customer vacates his chair after his hair cut completes. Your goal is to write the pseudocode for the customer and barber threads below with suitable synchronization. You must use only semaphores to solve this problem. Use the standard notation of invoking up/down functions on a semaphore variable.

The following variables (3 semaphores and a count) are provided to you for your solution. You must use these variables and declare any additional variables if required.

```
semaphore mutex = 1, customers = 0, barber = 0;
int waiting_count = 0;
```

Some functions to invoke in your customer and barber threads are:

- A customer who finds the waiting room full should call the function `leave()` to exit the shop permanently. This function does not return.

- A customer should invoke the function `getHairCut()` in order to get his hair cut. This function returns when the hair cut completes.

- The barber thread should call `cutHair()` to give a hair cut. When the barber invokes this function, there should be exactly one customer invoking `getHairCut()` concurrently.