CS347: Operating Systems

Problem Set 2

1. 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();

}
```

2. 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);


}
```

3. Consider the readers and writers problem discussed in class. 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.

4. Consider the readers and writers problem as discussed in class. Several reader and writer processes wish to access a critical section. Becasue readers do not modify the critical section, multiple readers can access the critical section concurrently. However, a writer can access the critical section only when no other reader or writer is concurrently accessing it. We wish to implement locking/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.

5. 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


}
```

6. Modern operating systems disable interrupts on specific cores when they need to turn off preemption, e.g., when holding a spin lock. For example, in xv6, interrupts can be disabled by a function call `cli()`, and reenabled with a function call `sti()`. However, functions that need to disable and enable interrupts do not directly call the `cli()` and `sti()` functions. Instead, the xv6 kernel disables interrupts (e.g., while acquiring a spin lock) by calling the function `pushcli()`. This function calls `cli()`, but also maintains of count of how many push calls have been made so far. Code that wishes to enable interrupts (e.g., when releasing a spin lock) calls `popcli()`. This function decrements the above push count, and enables interrupts using `sti()` only after the count has reached zero. That is, it would take two calls to `popcli()` to undo the effect of two `pushcli()` calls and restore interrupts. Provide one reason why modern operating systems use this method to disable/enable interrupts, instead of directly calling the `cli()` and `sti()` functions. In other words, explain what would go wrong if every call to `pushcli()` and `popcli()` in xv6 were to be replaced by calls to `cli()` and `sti()` respectively.

7. Consider an operating system where the list of process control blocks is stored as a linked list sorted by pid. The implementation of the wakeup function (to wake up a process waiting on a condition) looks over the list of processes in order (starting from the lowest pid), and wakes up the first process that it finds to be waiting on the condition. Does this method of waking up a sleeping process guarantee bounded wait time for every sleeping process? If yes, explain why. If not, describe how you would modify the implementation of the wakeup function to guarantee bounded wait.

8. Consider an operating system that does not provide the `wait` system call for parent processes to reap dead children. In such an operating system, describe one possible way in which the memory allocated to a terminated process can be reclaimed correctly. That is, identify one possible place in the kernel where you would put the code to reclaim the memory.

9. Consider a process that invokes the `sleep` function in xv6. The process calling sleep provides a lock `lk` as an argument, which is the lock used by the process to protect the atomicity of its call to sleep. Any process that wishes to call `wakeup` will also acquire this lock `lk`, thus avoiding a call to wakeup executing concurrently with the call to sleep. Assume that this lock `lk` is not `ptable.lock`. Now, if you recall the implementation of the sleep function, the lock `lk` is released before the process invokes the scheduler to relinquish the CPU. Given this fact, explain what prevents another process from running the wakeup function, while the first process is still executing sleep, after it has given up the lock `lk` but before its call to the scheduler, thus breaking the atomicity of the sleep operation. In other words, explain why this design of xv6 that releases `lk` before giving up the CPU is still correct.

10. Consider the `yield` function in xv6, that is called by the process that wishes to give up the CPU after a timer interrupt. The yield function first locks the global lock protecting the process table (`ptable.lock`), before marking itself as RUNNABLE and invoking the scheduler. Describe what would go wrong if `yield` locked `ptable.lock` AFTER setting its state to RUNNABLE, but before giving up the CPU.

11. Provide one reason why a newly created process in xv6, running for the first time, starts its execution in the function `forkret`, and not in the function `trapret`, given that the function `forkret` almost immediately returns to `trapret`. In other words, explain the most important thing a newly created process must do before it pops the trap frame and executes the return from the trap in `trapret`.

12. Consider a kernel design for an SMP (symmetric multiprocessor) system. There are two design choices for the queue of ready processes in the kernel. The kernel could maintain separate ready queues for each processing core, and schedule processes on a core from its local ready queue. Or, the kernel could have a common ready queue across all cores and run a process that is in the common queue on whichever core is free.

    (a) Provide one advantage of separate ready queues over a common queue.

    (b) Provide one advantage of a common ready queue over separate queues.

13. Consider the context switch of a CPU from the context of process P1 to that of process P2 in xv6. Consider the following two events in the chronological order of the events during the context switch: (A) the ESP (stack pointer) shifts from pointing to the kernel stack of P1 to the kernel stack of P2; (B) the EIP (program counter) shifts from pointing to an address in the memory allocated to P1 to an address in the memory allocated to P2. Which of the following statements is/are true regarding the relative ordering of actions A and B?

Answer _____

A. A occurs before B.

B. B occurs before A.

C. A and B occur simultaneously via an atomic hardware instruction.

D. The relative ordering of A and B can vary from one context switch to the other.

14. Which of the following actions by a running process will *always* result in a context switch of the running process, even in a non-preemptive kernel design?

Answer _____

A. Servicing a disk interrupt, that results in another blocked process being marked as ready/runnable.

B. A blocking system call.

C. The system call exit, to terminate the current process.

D. Servicing a timer interrupt.

15. 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?

Answer _____

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.

16. Consider a process P in xv6 that acquires a spinlock L, and then calls the function `sleep`, providing the lock L as an argument to `sleep`. Under which condition(s) will lock L be released *before* P gives up the CPU and blocks?

Answer _____

A. Only if L is `ptable.lock`

B. Only if L is not `ptable.lock`

C. Never

D. Always