Section 1 – True/False (40 pts – 8 Questions)

1. **True** / **False** In a Multitasking OS, there is Context Switching between Processes or Threads to allow the execution of multiple Tasks simultaneously. This does not necessarily mean in parallel, which is something that has to be supported by both the CPU architecture and the OS. Multitasking / Multiprogramming enables us to reduce the Process latency when dealing with multiple Processes. **(L1S18, L3S5)**

2. **True** / **False** To gain direct control of Devices in a more efficient manner, the OS offers specialized operations to User-Space programs, which allow them to circumvent the need to perform a System Call, as this comes with the overhead of having to Trap to Kernel Space and go through the entire System Call Dispatch mechanism. **(L1S22)**

   The only way for an application to invoke OS services is through a system call.

3. **True** / **False** The CPU architecture includes support for different Privilege Levels through dedicated Register fields, such that only certain sets of Instructions and operations are allowed at each Level. Once a System Call is triggered by the User-Level program, the CPU will modify its Privilege Level "bit" and vector to a known address of the Interrupt Dispatch Table which is assumed to be bound to an Interrupt Handler routine. It is the OS' responsibility to setup this mapping. **(L2S20, L2S29)**

4. **True** / **False** Hardware Interrupts are Asynchronous and are the product of the CPU executing certain Instructions. **(L2S33)**

   Interrupts (hardware interrupts) are caused by an external event: they are asynchronous. **Exceptions** (software interrupts) are caused by the CPU executing illegal instructions: they are synchronous.

5. **True** / **False** Inter-Process Communication (IPC) mechanisms include Message-Passing, Signals, and Shared Memory, and provide channels of communication between different Processes, which otherwise operate as Isolated and Protected units. **(L3S12)**

6. **True** / **False** A Process Control Block is a heavyweight data structure as it needs to include all the necessary information that fully defines the Process and its State. A Thread Control Block is a much more lightweight data structure as a Thread (also named a Lightweight Process) shares resources with every other Thread that belongs to the same Process. **(L3S17, L4S6)**

7. **True / False** Process creation in Unix is initiated by setting up a from-scratch and freshly- initialized Virtual Address Space, and then initiating the execution of the desired User-Space program within it. **(L3S37)**

   `pid_t fork(void)` is initialized with a copy of the address space of the parent.

8. **True / False** The Parent-Child Process relationship results in a Process Tree throughout the OS' lifetime. This necessitates that every time a Parent node (Process) is killed, its Children are recursively killed and cleaned up after as well. **(L3S52, L3S53, L3S54)**

   The first statment is true, but the second statement is false as a parent's children are not immediately killed with the parent and processes that remain alive after their parent process has been killed are called orphan processes.

9. **True / False** All Threads of a Process operate under the same Virtual Address Space mapping, within which each Thread gets its own distinct Memory Region with its own separate Stack, Heap, and Data and Code Segment. **(L4S8, L4S41)**

   The first statement is true, but each thread has its own separate stack (and own control block) while the heap, static data, and code are common for the entire process.

10. **True / False** The Many-to-One Thread Multiplexing model allows to tie multiple User-Level Tasks to a single Kernel Thread, which allows us to switch between those Tasks in a Multiprogramming fashion without having to incur the cost of a System Call. The Many-to-Many model gives us the ability to additionally avoid Blocking of all the User-Level Tasks when just one happens to perform a Blocking Call, but at the cost of implementation complexity. **(L4S27, L4S28)**

    This statement is true: one kernel thread may have many user threads to avoid incurring the cost of a system call, but if one of those threads gets blocked, all of those threads will eventually get blocked. A way of avoiding this is to tie multiple kernel threads to the same user threads, but this comes at the cost of implementation complexity.

11. **True / False** The `sched_yield` System Call which is used to relinquish CPU control can be used as the driving mechanism to implement Non-Preemptive Thread Scheduling. Its use is also appropriate every so often in our code, in order to give other Threads of the same Process a chance to run, as the Thread Context Switching in such a case is fast (no change of Virtual Address Space required, etc.) **(L4S33)**

    Making strategic calls to schedule `sched_yield()` can improve performance by giving other threads or processes a chance to run, but spamming calls to it will result in unnecessary context switches, which will degrade system performance.

12. **True / False** The reasons why we need Synchronization mechanisms include the fact that Thread Scheduling can non-deterministic, the fact that Multi-Word operations are not implemented Atomically, and finally the fact that the compiler is free to Reorder Instructions for optimization purposes, and the Hardware may also Reorder Memory Accesses. **(L5S10)**

13. **True / False** The Mutex (`mutex_t`) mechanism is an alternative to Lock-based Spin-Waiting, which performs Blocking. This is highly inefficient as it involves having to maintain extra state variables for the Mutex as well as perform enqueuing and dequeuing operations. **(L5S39, L5S40)**

    The implementation of the mutex involves a lock and is more efficient than an endless while loop that invokes `sched_yield()` even though it involves more logic and data members.

```
void lock(mutex_t *m) {
  //acquire guard lock by spinning
  while (test_and_set(m->guard));

  if (m->flag == 0) {
    m->flag = 1; // mutex acquired

    unset(m->guard);

  } else {
    // put self into waitlist
    enqueue(m->queue, self);

    unset(m->guard);

    pause(); // suspend until signal
  }
}
```

```
void unlock(mutex_t *m) {
 //acquire guard lock by spinning
  while (test_and_set(m->guard));

  if (empty(m->queue))
    // release mutex; no one wants mutex
    m->flag = 0;
  else
    // remove from waitlist & send signal
    wakeup( dequeue(m->queue) );

  unset(m->guard);
}
```

14. **True / False** A Semaphore is a Synchronization mechanism that internally leverages an integer value that is never allowed to become negative. The integer variable itself cannot be accessed or manipulated directly, only through the Semaphore interface functions `sem_post()` and `sem_wait()`. **(L6S3)**

15. **True / False** A Condition Variable is a Synchronization mechanism that has no "memory". This means that if at the time that a `pthread_cond_signal()` operation takes place in one Thread, there is no other Thread that has is Blocked as a result of a `pthread_cond_wait()`, the signal is just lost. **(L6S11)**

16. **True / False** One way to prevent Deadlocks is to ensure that no cycles in the Resource Allocation Graph ever occur, by statically numbering all Resources and enforcing that our application only attempts to take them in that order (such that attempting to take a Resource ordered lower than a Resource currently held, will not proceed to take and Lock it). This is method is easily generalizable in practice and most OSs provide such a strict standard for Application developers to follow. **(L7S14)**

    The first part statement is true, but such a method is not generalizable in practice and a strict standard will not be followed by every developer on the OS.

17. **True / False** The Eraser -or- Lockset algorithm allows us to infer Data Locking relationships during the program's runtime. This can be implemented in the form of a binary tool that allows to infer Data Race conditions (instead of detecting actual Data Races elicited), by inferring the Sets of Locking mechanisms protecting shared data at runtime. **(L7S21, L7S23, L7S26)**

18. **True / False** The Shortest Remaining Time First (SRTF) Scheduling algorithm is the Preemptive variant of Shortest Job First (SJF) Scheduling. Without Preemption allowed, SJF is the optimal algorithm in terms of Average Waiting Time, a property which it loses in case of Preemptive Scheduling. In that case, SRTF is capable of improving the Average Waiting Time. Both algorithms have to rely on estimating the actual Task Burst Time through techniques that leverage past values. **(L8S15, L8S16, L8S18)**

19. **True / False** The Round-Robin (RR) Scheduling algorithm falls under the category of Non-Preemptive Scheduling, as each Task is responsible for keeping track of its Time Quantum and Yielding CPU back to the Scheduler once it expires. **(L8S20)**

    Every task is given a time quantum and it's always preempted at the end of each time quantum. It is not each task's responsibility to keep track of their own time quantum and instead is the scheduler's.

20. **True / False** In Multi-Level Feedback Queue (MLQ) Scheduling we have multiple queues of Tasks, with each queue associated to a Priority Level. Higher Priority-Level Tasks always Preempt Lower Priority-Level ones. In such a scheme, we typically define a Static Priority for a Task through its nice-ness value, and then dynamically adapt the queue that it is moved to (i.e. the Task's dynamic Priority Level) once it consumes its Time Quantum and is Preempted. **(L8S33, L8S34, L838, L8S50, L8S51, L8S54)**

21. **True / False** The Linux O(1) Scheduler has become deprecated because one of its main downsides was that it had to dynamically compute Task Aging each time a Scheduling decision was taken. **(L8S51, L8S49)**

The Linux O(1) scheduler was very efficient in how it computed task aging. It did not have to compute dynamic task aging for each individual process. All it was doing was simulated aging by moving processes to the expired array and then swapping the active and the expired array.

22. **True / False** The Linux Completely Fair Scheduler (CFS) relies on Red-Black Tree operations in order to always maintain a balanced Tree of Tasks in which the leftmost node corresponds to the Task that has received the least Virtual Execution Time. **(L8S56)**

23. **True / False** The Linux Completely Fair Scheduler (CFS) is the default policy for `SCHED_NORMAL`-class Tasks. For Real-Time Tasks, the OS maintains the strategy of Multi-Level Priority Queues, within which at each Priority Level a Task can either be Preemptible (`SCHED_RR`) or Non-Preemptible (`SCHED_FIFO`). Tasks at a Higher Priority-Level always Preempt Lower Priority-Level ones, and Real-Time Tasks always Preempt and are Prioritized over Normal Tasks. **(L8S57, L8S59)**

24. **True / False** In the Virtual Memory Segmentation model, each Process' Virtual Address Space has certain contiguous Regions allocated to each of its Virtual Memory Segments (Code Segment, Data Segment, Stack Segment, etc). The problem with this model is that each contiguous Segment cannot have a fixed size, and therefore External Fragmentation of Memory occurs. **(L9S11, L9S15)**

25. **True / False** In the Virtual Memory Paging model, each Process' Virtual Address Space is broken down into fixed-size Pages, which are associated to Physical Memory Pages via a mapping that is stored and maintained by the OS in the Process' Page Directory and Page Tables. The fixed-size split of Memory does away with External Fragmentation, but an average Internal Fragmentation of 0.5 Page per contiguously allocated Linear Virtual Memory Region still exists. **(L9S16, L9S17, L9S27)**

26. **True / False** To better Protect the Kernel from User-Space Processes, the Kernel's entire Memory is mapped with Virtual Memory at randomized Addresses at the initial Boot-time of the system. **(L9S35, L9S40)**

    Part of the kernel, at least in the beginning of put time, is mapped to a specific location of physical memory with identity mapping or 1 to 1 mapping. The kernel address space is mapped partially with identity mapping on physical addresses to virtual addresses 1 to 1 correspondences, and after that identity mapping there can be some virtual address mapping.

27. **True / False** Process Page Directories remain in the Kernel Memory at all times, while the Page Tables that contain the actual Physical Frame mappings can be Paged-Out/Paged-In on demand. Context Switching to a different Process involves changing the Virtual Address Space, which essentially means loading the appropriate CPU Register (e.g. CR3 in x86) with the new Process' Page Directory Address. **(L9S39, L9S40)**

28. **True / False** The Translation Lookaside Buffer (TLB) is a cache of recent Virtual Memory -to- Page Table Entry translations. This facilitates faster translation than having to perform multiple indirections (first to Page Directory Entry (PDE) – then to Page Table Entry (PTE)). This however also means that the OS is responsible for maintaining its consistency between the cached PTE in the TLB, and the actual PTE (e.g. when Page Table Entry is updated), as well as invalidate the corresponding TLB cache of PTEs when a new Process is Context Switched-in. **(L9S42, L943, L9S45)**

29. **True / False** The Translation Lookaside Buffer (TLB) is responsible for translating from Virtual Memory to Physical Memory Address, and the Memory Management Unit (MMU) is responsible for performing Bounds checking. **(L9S42, L9S20)**

    The translation lookaside buffer is responsible for translating from virtual memory or virtual page number to page table entry, not the final physical memory address. It is the MMU's responsibility, not only just perform bounce checking (depending on the model), but to resolve to the final physical memory address.

30. **True / False** In Software-Managed Translation Lookaside Buffers (TLB)s there is no Kernel intervention once a TLB Miss occurs. The Process' Page Tables have to be maintained in Memory by the Kernel in the appropriate expected format, and the MMU immediately loads them up into the TLB cache from their known Memory Addresses. **(L9S44)**

    The process of loading up the appropriate page table entry involves the TLB faulting to the OS and the OS taking up responsibility to intervene and do the appropriate loading of a page table entry. This gives flexibility. The alternative, which is being described in this question, is about hardware managed TLB loading, which does not involve any OS intervention other than the fact that the OS is responsible for maintaining those page table entries in a hardware defined format in memory for the hardware to fetch directly, once the TLB miss occurs.

31. **True / False** In Shared Memory models where Shared Memory is mapped to different Virtual Address ranges for each Process, we gain flexibility but lose the ability use Pointer values with any amount of consistency. **(L9S66)**

Section 2 – General Questions (20 pts – 2 Questions)

1. Describe the problem that may arise with Zombie Children Processes, and how the OS deals with it. **(L3S52)**

   Zombie processes are the terminated child processes that have not been waited on by a parent yet. The problem with zombie processes is that although the program that was originally running has finished, the operating system still needs to maintain it within the process table because it needs to preserve minimal information about the process itself, such as a the process ID and its termination status. At a later time, the original child process' parent might want to wait on that child process in order to acquire information about it, such as its termination status. As long as these zombie processes are maintained within the kernel's process table, the table might eventually get filled up, so the operating system mitigates this issue by making sure that if the parent process never waits on the child (i.e. it terminates without waiting on it) then any zombie children are adopted by the `init` process. The `init` process automatically removes the zombies itself, periodically.

2. Describe what is the potential problem we need to watch out for when using the `fork()` mechanism to create a new Process from a Parent Process that is Multithreaded. **(L4S31)**

   Whenever we call `fork()`, the `fork()` sys call will only copy the calling thread, which means that the child process will be an exact copy of the parent process, but only the calling thread of `fork()` will be in a non-suspended state, therefore, in the child process, if there are any locks being held by non-calling threads, these ones will be forever locked in the child process. It is therefore safe to call `fork()` from a multithreaded process as long as we only execute a signal safe operations, until the system takes over the entire child process from a different program by calling one of the exec variants.

3. Describe (in words) how Context Switching between Threads takes place. **(L4S42)**

   Context switching is performed by a function that takes two arguments: the current thread and the next thread. First, it pushes the next and current arguments of the function as well as the return address onto the stack. Then, it saves the callee-saved registers on the stack, effectively pushing the register values onto the stack of the current thread. Then, it saves the stack pointer of the current thread into the thread into the controlled block of the current thread. It then takes the stack pointer of the next thread from its corresponding thread controlled block and loads it out onto the actual stack pointer (ESP). At this point, it has effectively switched the stacks between the two threads. Finally, returning from a function onto the next thread, it pops the values of the callee-saved registers of the next thread that come from the point that it was itself preempted in the past, and returns from that function based on the return address that was saved onto the stack as next was preempted itself.

4. Describe (in words) how the Atomic Test-And-Set Instruction facilitates Mutual Exclusion. **(L5S30)**

   The atomic test and set instruction is capable of immediately setting the value of the lock to 1, which indicates that this is now held by a thread. At the same time, it returns the value of the lock before it was set. This can be used for testing to see whether the spin lock should proceed or not, so if the value of the lock was already one, it will just maintain its value and the spin lock will see that it has to keep spinning. However, if the vale of the lock is zero, i.e. is it free, it will be immediately marked as held and its prior value will be returned back which will indicate to the spin lock that it is capable of proceeding, i.e. exiting the lock.

5. Describe why when we perform a Blocking `pthread_cond_wait` on a Condition Variable as a result of its Predicate being true, we need to do so with a while instead of a single if which decides whether to perform the Blocking action just once. **(L6S14)**

   Whenever we perform a Blocking `pthread_cond_wait`, it is possible that it returns even if no `pthread` condition signal has been called, which means that it is possible to return from the otherwise blocking call even though the predicate has not been fulfilled. So it instead returns back to the while, re-checks the predicate, and if the predicate is indeed fulfilled, it then proceeds to do what was going to happen after the blocking wait call.

```
int nfilled = 0;
  cond has_empty, has_filled;

  void produce() {
    while (nfilled == N)
      wait (has_empty);
    ... // fill a slot
    ++ nfilled;
    signal (has_filled);
  }
```

```
void consume() {
    while (nfilled == 0)
      wait (has_filled);
    ... // empty a slot
    -- nfilled;
    signal (has_empty);
  }
```

6. Describe what the problem of Priority Inversion is in Priority-based Preemptive Scheduling with an example. **(L8S29)**

   Assuming we have a low priority task that holds a resource or a lock that's also needed by a high priority task and there's a third, medium priority task that comes in, its possible that the medium priority task preempts the low priority task and never lets it reach the point where it unlocks the resource that's being wanted by the high priority task. Effectively, this is to the medium priority task never allowing the high priority task to happen.

7. Assume 3 Tasks: L, M, H with Priorities (lower number: higher Priority): L(5), M(3), H(1)

   Assume:
   L holds Lock K1
   M holds Lock K2
   What is going to happen to the Task Priorities once:
   H requests Lock K2: M (which is holding lock K2) has its priority set to 1
   M requests Lock K1: L (which is holding lock K1) has its priority set to 1
   (L8S31)

   When the high priority task requests the lock K2, which is held by the medium priority task, the medium priority task is going to be raised to the priority of the high priority task (1). When the medium priority task requests the lock K1, which is held by the low priority task, the low priority task is going to receive the priority of the medium priority task which was raised to priority 1, so all the tasks will end up having a priority of 1.

8. Describe how in Symmetric Multi-Processing (SMP) Processor Affinity-based Scheduling can benefit execution speed. **(L8S44, L8S45)**

   Processor Affinity based scheduling allows us to add processes to a specific CPU's queue if it has already recently run on that same CPU. This benefits from CPU cache states being present. If the same tasks are scheduled as much as possible on the same CPU, the process will benefit from the locality of data by using the cache state that's already present there from its previous executions.

9. Assuming a 4KiB Page size, and that a Process' Page Directory is desired to take up a single Page of Memory, for a 32-bit architecture, show how many bits are required in the Virtual Address fields for:

   (a) The Page Directory (Master Page Table)

   (b) The Page Table (Secondary Page Table)

   (c) The Page Offset

   (L9S26)

   Note: $1Ki$ denotes 1 Kibibit $= 1,024\ \ bits = 128\ \ bytes\ \ (2^7\ \ bytes)$.

   $1KiB$ denotes $Kibibytes$, $\ 1KiB = 8,192\ \ bits = 1,024\ \ bytes\ \ (2^{10}\ \ bytes)$.

   $1KB$ denotes 1 $\ Kilobyte = 1000\ \ bytes\ \ (10^3\ \ bytes)$

   Page: 4KiB, Page Table Entry (PTE) Size: 4B (32 bit system)

   Offset size $= \log_2(\text{page size}) = \log_2(4KiB) = \log_2(4,096\ \ bytes) =$ 12 bits

   The system wants to fit the entire Master Page Table in 1 Page: $\frac{\text{page size}}{\text{page table entry size}} = \frac{4KiB}{4B} = \frac{4 * 8,192\ \ bits}{32\ \ bits} =$ $1,024\ \ bits = 1Ki =$ 1024 entries (Secondary Page Table)

   For 1024 entries, $x = \log_2(1Ki) \rightarrow 2^x = 1Ki\ \ or\ \ 2^x = 1,024\ \ bits \rightarrow x =$ 10 bits for the Master VPN and the remaining 10 bits are for the secondary VPN. Note how 10 bits + 10 bits + 12 bits = 32 bits (i.e. 32 bit architecture)

	Master VPN	**Secondary VPN**	**Offset**
32-bit Virtual Address	10-bit long	10-bit long	12-bit long

2^x	3	4	5	6	7	8	9	10	11	12
=	8	16	32	64	128	256	512	1024	2048	4096

10. Describe how the Copy-on-Write "trick" can be used to facilitate efficient Process creation as done through the `fork()`ing mechanism. **(L9S57)**

In theory, the forking mechanism should duplicate, i.e. copy the entire address space of the parent process to the child process. A more efficient way to do that is to create shared mappings of the parent's pages in the child's virtual address space. Effectively, we do that by marking the page table entry both in the parent and the child as read-only. This means that whenever there is an attempt, either by the parent or the child, to to differentiate the content of the memory, the system will generate a protection fault because the differentiation will be an attempted write to a read-only page. This will trigger a trap and the OS which will effectively copy the page and change the page mapping in the child's page table, and restart the write instruction.

Section 3 – Practical Questions (40 pts – 4 Questions)

1. The following Mutual Exclusion Locking mechanism called Strict Alternation has an issue. Write down a sequence of operations for Threads 1 and 2 that will lead to a problem situation: **(L5S27)**

```
int turn = 0; // 0: Thread 0, 1: Thread 1
void lock() {
    while (turn == 1 - self);
}

void unlock() {
    turn = 1 - self;
}
```

thread 0	shared vars	thread 1
	turn = 0;	
while(turn == 1 - self)		while(turn == 1 - self)
spin		**proceed**
while(turn == 1 - self)		
spin		
		...
		turn = 1 - self;
	turn = 1;	
while(turn == 1 - self);		
proceed		
unlock()		
turn = 1 - self;		
	turn = 0;	
while(turn == 1 - self);		
spin		
while(turn == 1 - self);		
spin		
while(turn == 1 - self);		
spin		
...		

The value of turn is initially 0. Both threads then attempt to acquire the lock. The first one fails and keeps spinning one while the second one grabs it and proceeds past the while. It then does some operations in its critical section while the first thread keeps trying and failing to acquire the lock (spin waiting or busy waiting). At some point, the second thread finishes and turns the lock over to the other thread. The subsequent while will manage to grab the lock and perform its own critical section, after which it'll unlock. If the other thread never returns back to execute its own lock and unlock operation, i.e. perhaps after it's done executing its critical section (after the unlock operation) it terminates or goes into an infinite loop, then it'll never turn over the turn to the first thread, which is a problem with alternation. The turn will keep remaining set to zero and the first thread will keep spin waiting forever and never reacquire the lock.

2. The following Threading implementation has an issue. What has happened here? Every single function uses Mutual Exclusion mechanisms appropriately. So what was the flaw in this design? **(L7S6)**

```
Translation Unit: my_program.c
#include external_lib.h
pthread_mutex_t mut;
void thread_1() {
    pthread_mutex_lock(mut);
    consume();
    pthread_mutex_unlock(mut);
}

void thread_2() {
    pthread_mutex_lock(mut);
    produce();
    pthread_mutex_unlock(mut);
}

Translation Unit: external_lib.c
pthread_mutex_t prod_cons_mut;
pthread_cond_t cv;
void consume() {
    pthread_mutex_lock(prod_cons_mut);
    while (!ready)
    pthread_cond_wait(cv, prod_cons_mut);
    ...
    pthread_mutex_unlock(prod_cons_mut);
}

void produce() {
    pthread_mutex_lock(prod_cons_mut);
    ...
    ready = true;
    pthread_cond_signal(cv);
    pthread_mutex_unlock(prod_cons_mut);
}
```

thread_1	vars	thread_2
pthread_mutex_lock(mut);	**mut**: locked	
pthread_mutex_lock(prod_cons_mut);	**prod_cons_mut**: locked	
pthread_cond_wait(cv, prod_cons_mut);	thread_1 blocked	
	mut: locked	pthread_mutex_lock(mut);
	thread_2 blocked	

Each standalone module seems to be using its own internal synchronization mechanism appropriately. For example, thread one locks a mutex, invokes a function, and the unlocks a mutex. A second thread does the same. The issue arises from the fact that the functions invoked by the threads have their own internal synchronization. The `consume()` and `produce()` functions rely on each other to continue, and will enter a deadlock as the first level of synchronization never lets the two threads call both functions simultaneously.

3. The following is a Reader(s)-Writer(s) implementation. What would happen if we moved the `pthread_mutex_unlock` above the check: if (`readcount == 1`)? **(L7S39)**

```
// number of readers
int readCount = 0;

// mutual exclusion to readcount
pthread_mutex_t mutex;

// exclusive writer or reader (binary sem { initialized to 1)
sem_t w_or_r;

void writer() {
```

```
    sem_wait(&w_or_r); // lock out readers
    write();
    sem_post(&w_or_r); // up for grabs
}

void reader() {
    pthread_mutex_lock(&mutex); // lock readcount
    readcount += 1; // have one more reader

    if (readcount == 1) // NOT(!): >= 1                    (was above this line)
        sem_wait(&w_or_r); // synch w/ writers

    pthread_mutex_unlock(&mutex); // unlock readcount      (what if this line)

    read();

    pthread_mutex_lock(&mutex); // lock readcount
    readcount -= 1; // have one less reader

    if (readcount == 0)
        sem_post(&w_or_r); // up for grabs

    pthread_mutex_unlock(&mutex); // unlock readcount
}
```

It is possible for the thread to begin with `readcount == 0`. One thread goes in, checks to unlock the mutex, acquires it, increments `readcount` to 1, and unlocks the mutex. At this point, it's possible that the system is context switched to another reader thread before running the writer synchronization line (this is what's affected by moving the `unlock` line). The new thread will see that the mutex is unlocked, and increase `readcount` to 2. Regardless of whether that new thread proceeds or process gets context switched back to the other thread, the writer synchronization flag has been skipped as the condition `readcount == 1`.

4. The following is a Thread-Safe Bounded-Buffer implementation. Since synchronization is achieved by the 2 Semaphores, why do we even need the `pthread_mutex_t`? **(L7S40)**

```
pthread_mutex_t mutex;
// count of empty slots (counting semaphore, initialized to N)
sem_t empty;
// count of filled-up slots (counting semaphore, initialized to 0)
sem_t filled;
    void producer() {
        while (1) {
            produce_new_resource();
            sem_wait(&empty); // wait for 1 empty slot
            pthread_mutex_lock(&mutex);
            insert_resource_to_buffer();
            pthread_mutex_unlock(&mutex);
            sem_post (&filled); // notify +1 filled slot
        }
    }

    void consumer() {
        while (1) {
        sem_wait(&filled); // wait for 1 filled slot
        pthread_mutex_lock(&mutex);
        extract_resource_from_buffer();
        pthread_mutex_unlock(&mutex);
        sem_post(&empty); // notify +1 empty slot
        consume_resource();
    }
}
```

The semaphores `empty` and `filled` are enough to martial the operation between `producer` and `consumer`. The producer will wait until there's at least one empty slot, which will be achieved at the point where the consumer will post that there is one additional empty slot. Then the consumer itself also waits until there is at least one filled slot. This at least one filled slot will be generated once the producer reaches the point where it posts that sempahore to notify that one additional filled slot has been created. So, the question is, "why do we need the mutex at all?". The mutex is there to protect the internal of the buffer, the operations of `insert_resource_to_buffer();` and `extract_resource_from_buffer();`, which mutate the buffer and need to be done in a thread safe manner.

5. The following is a Threaded H2O Producer implementation. In either of `HArrives()` and `OArrives()`, the `pthread_mutex_t` becomes `pthread_mutex_lock`-ed first thing. But if there aren't enough H and O threads, then we will enter the else branch which immediately blocks on `pthread_cond_wait`, not allowing to proceed to the `thread_mutex_unlock` at the very end of the function. Is this not a Deadlock condition? Why not? **(L7S42)**

```
typedef struct _thread_status {
    bool ready;
    pthread_cond_t cv; // condition variable (controls thread suspension)
} thread_status ;

pthread_mutex_t mutex;
int numH, numO;
list<thread_status *> waitingH;
list<thread_status *> waitingO;

void HArrives() {
    pthread_mutex_lock(&mutex);
    numH++;
    if (numH == 2 && numO >= 1) {
        h = waitingH.pop();
        o = waitingO.pop();
        h->ready = true;
        o->ready = true;
        pthread_cond_signal(&h->cv);
        pthread_cond_signal(&o->cv);
        numH -= 2;
        numO -= 1;
        make_water();
    } else {
        thread_status* h = (thread_status*)malloc(sizeof(thread_status));
        h->ready = false;
        pthread_cond_init(h->cv, NULL);
        waitingH.push(h);
        while (!h->ready)
        pthread_cond_wait(&h->cv, &mutex);
        free(h);
    }

    pthread_mutex_unlock(&mutex);
}

void OArrives() {
    pthread_mutex_lock (&mutex);
    numO++;
    if (numH >= 2) {
        h1 = waitingH.pop();
        h2 = waitingH.pop();
        h1->ready = true;
        h2->ready = true;
        pthread_cond_signal (&h1->cv);
        pthread_cond_signal (&h2->cv);
        numH -= 2;
        numO -= 1;
```

```
            make_water();
        } else {
            thread_status* o = (thread_status*)malloc(sizeof(thread_status));
            o->ready = false;
            pthread_cond_init(o->cv, NULL);
            waitingO.push(o);
            while (!o->ready)
            pthread_cond_wait(&o->cv, &mutex);
            free(o);
        }

        pthread_mutex_unlock (&mutex);
    }
```

When H arrives, the first thing occurs is the locking of the mutex. Then, it performs a check to see if there are enough H and O molecules. At first, it might seem that if the process goes to the `else` branch and it reaches the wait, it will arrive at a blocking call and never move past it, thus never allowing any other threads to run logic. This is not the way that `pthread_cond_wait` operates. It immediately releases the mutex but blocks at its call line, as it assumes that it's called with the mutex in a locked state. It will later return with the mutex owned by it when signaled.

6. For the following sequence of Tasks, their Arrival Times, and their Burst Times, draw the Gannt Chart of their Scheduling using the Shortest Remaining Time First algorithm and calculate the Average Waiting Time. **(L8S17)**

Task	Arrival Time	Burst Time
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Remember that processes arrive at the start of the time slice and finish at the end of the time slice. When a process arrives and is switched to, it will run for that time slice as the switch occurred at the beginning of the time slice. When a process finishes during a time slice, it needs to take the entire time slice to do so.

Time	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Process	P1	P1	P2	P2	P3	P2	P2	P4	P4	P4	P4	P1	P1	P1	P1	P1

Time	Process	Notes
0	P1	P1 arrives (7s remain)
1	P1	P1 running (6s remain)
2	P2	P2 arrives, switch to P2 (P2 (4s) < P1 (5s))
3	P2	P2 running (3s remain)
4	P3	P3 arrives, switch to P3 (P3 (1s) < P2 (2s))
5	P2	P3 done, P4 arrives, switch to P2 (P2 (2s) has less time remaning than P1 and P4)
6	P2	P2 done, switch to P4 (P4 (4s) < P1 (5s))
7	P4	P4 running (3s remain)
8	P4	P4 running (2s remain)
9	P4	P4 running (1s remains)
10	P4	P4 done, switch to P1 (P1 (5s) is the only process that remains)
11	P1	P1 running (4s remain)
12	P1	P1 running (3s remain)
13	P1	P1 running (2s remain)
14	P1	P1 running (1s remains)
15	P1	P1 done

Process	Time Spent Waiting
P1	9 seconds
P2	1 second
P3	0 seconds
P4	2 seconds
Average	3 seconds

7. For a Virtual Memory Paging model, calculate the Physical Frame Address of the following Virtual Address Memory access: **(L9S26)**

Master VPN	Secondary (Page Table) VPN	Offset
0x 001	0x 003	0x A12

Page Directory

VPN	Address Memory Access	Status Bits
0	0x F FF0A	Protection bits
1	0x F FA0B	Protection bits
2	0x F FB00	Protection bits
n	...	Protection bits
$2^{20} - 1$	0x F F90E	Protection bits

Page Table – Address 0x F FF0A

VPN	Address Memory Access	Status Bits
0	0x F AF0A	Protection bits
1	0x F BA0B	Protection bits
2	0x 3 6375	Protection bits
3	0x E 0000	Protection bits
n	...	
$2^{20} - 1$	0x F F90E	Protection bits

Page Table – Address 0x F FB00

VPN	Address Memory Access	Status Bits
0	0x E 0001	Protection bits
1	0x A B10B	Protection bits
2	0x 0 44FF	Protection bits
3	0x F FB00	Protection bits
n	...	
$2^{20} - 1$	0x F A90E	Protection bits

Page Table – Address 0x F FA0B

VPN	Address Memory Access	Status Bits
0	0x A F00E	Protection bits
1	0x 6 FA0B	Protection bits
2	0x 3 FAFF	Protection bits
3	0x 5 FB11	Protection bits
n	...	
$2^{20} - 1$	0x 1 FN00	Protection bits

Page Table – Address 0x F F90E

VPN	Address Memory Access	Status Bits
0	0x 1 0000	Protection bits
1	0x 0 1000	Protection bits
2	0x 0 7FAB	Protection bits
3	0x 0 0011	Protection bits
n	...	
$2^{20} - 1$	0x 0 2345	Protection bits

Page Table – Address 0x...

(0x 5 FB11 << 12) | 0x A12 = 0x 5FB1 1A12