Gordon Ng gng8@bu.edu

CS350 Undergraduate Operating Systems **Spring 2021**

Written Assignment 1

Assigned: March 24 *Due: April 4, 11:59PM EDT*

1. **Protection and Interrupt-Handling:**

   (a) Consider a dual-mode system in which the kernel is separated from user-level applications using two “rings of protection”.

   - (**4 points**) Is it safe for the kernel to directly access user-level memory in any process address space? Briefly explain your answer.

It is safe for the kernel to directly access user-level memory in any process address space. It needs a virtual address for direct access to memory spaces. User-space programs can't access kernel memory, kernel can access user memory. Kernel shouldn't execute user-memory.
If the kernel has direct access to execute user-level memory, there is a security issue. If theres no request to access user-space memory, a malicious code like ransomware could directly access application data. There's a set of accessors functions such as copy_to_user(), which returns information from a valid user range. There's also a set of features in x86 CPUs like Supervisor Mode Execution Prevention (SMEP) and Access (SMAP) stops kernel from accessing user-level memory.

   - (**6 points**) Is it safe for the kernel to directly execute *user-level code* in *any* process address space? Again, briefly explain your answer.

It is not safe for the kernel to directly execute user-level code. CPU has two modes: Kernel Mode and User Mode. Kernel Mode executing code has access to the hardware and can execute any code and reference any memory address. This is reserved for the lowest rated levels and most trusted commands in the OS. User Mode executing code doesn't have access to hardware or memory. Code in this area must use APIs. The difference between the two modes is that Kernel crashes the device when it fails, and User Mode doesn't crash the device due to the use of APIs. If kernel has direct access to user-level code and it fails, the device will succumb to failure aswell.

   - (**8 points**) Systems such as UNIX allow processes to associate signal handlers with various events. Unlike interrupts, signals are not handled until the target process is active. Briefly explain one CPU, memory and I/O protection problem if we allowed a signal handler for process $p1$ to execute in the context of another process, $p2$. If we could address these protection problems, what would be the advantage of this scheme?

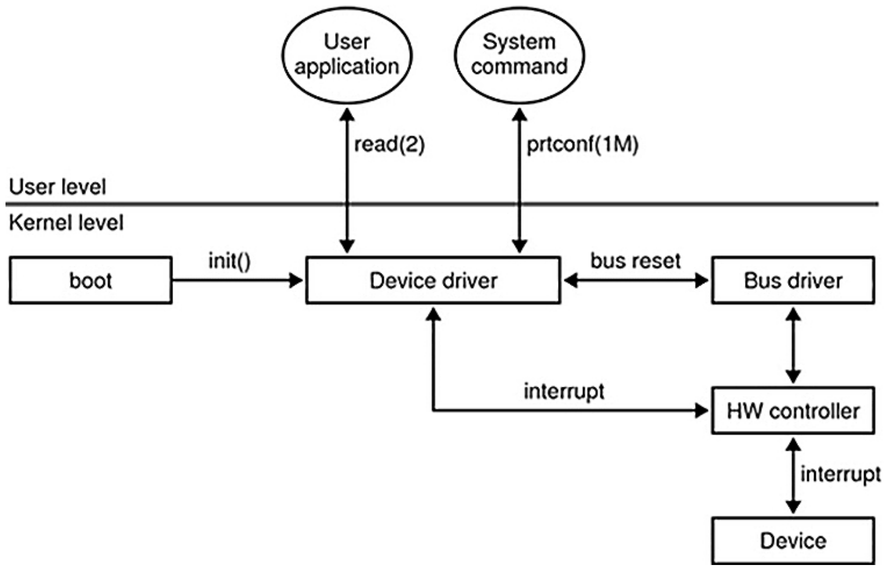
CPU problem: A deadlock occurs when p1 gains access to data structure a and process p2 gains access to b, but p1 waits for b and p2 waits for a. A timer interrupt runs the OS Scheduler which is only allowed with privilege. P1 would essentially interrupt p2 and cause a deadlock on the CPU.
I/O problem: While in user mode the CPU can't run I/O instructions, if p1 and p2 executes in context of the other, then the user can run I/O instructions and there is a security flaw, user could destroy disk data and OS can't check if you're allowed to use I/O instructions or not.
Memory problem: P1 and P2 might write to the same memory location at the same time. Memory is protected by partitions, CPU can only access some of these pieces, controlled by "base" and "limit" registers, which sets a bottom limit and a number of addresses of locations.
The advantage of this scheme would be UNIX provides more advanced signal handling. If processes has multiple signals, it would ignore the issues caused by multi threads. Individual threads can block signals so only one thread handles the signal. Adds asynchronous notifications and synchronous errors/exceptions, adds shared memory with shmget() and other function modules, better memory management and less deadlocks.

(b) (**8 points**) Consider the implementation of a micro-kernel on an architecture such as the Intel x86. Suppose this OS structure allows device drivers to be implemented at user-level. On the x86 architecture a general protection fault occurs if we attempt to *trap* to a ring of protection less privileged than the current protection domain. Briefly explain how we can invoke user-level device drivers from the kernel (by essentially performing the opposite of a system call). Show pseudo-code or a diagram, if necessary, to help with your explanation.

Driver Directory Organization:
/kernel
These modules are common across most platforms. Modules that are required for booting or for system initialization belong in this directory.
/platform/`uname -i`/kernel
These modules are specific to the platform identified by the command uname -i.
/platform/`uname -m`/kernel
These modules are specific to the platform identified by the command uname -m. These modules are specific to a hardware class but more generic than modules in the uname -i kernel directory.
/usr/kernel
These are user modules. Modules that are not essential to booting belong in this directory. This tutorial instructs you to put all your drivers in the /usr/kernel directory.

Reference: https://docs.oracle.com/cd/E19253-01/817-5789/emjjp/index.html#:~:text=The%20kernel%20calls%20device%20drivers,User%2Dlevel%20requests.

We can invoke user-level device drivers from the kernel by calling functions that point to entry points. The kernel calls device drivers such as prtconf(1M), with the use of User-level requests. There's different modules, private modules for drivers. Device drivers declare entry points in dev_ops(9S) and cb_ops(9S) depending on type of routine of entry point.



Reference: https://docs.oracle.com/cd/E37838_01/html/E61062/emjjp.html

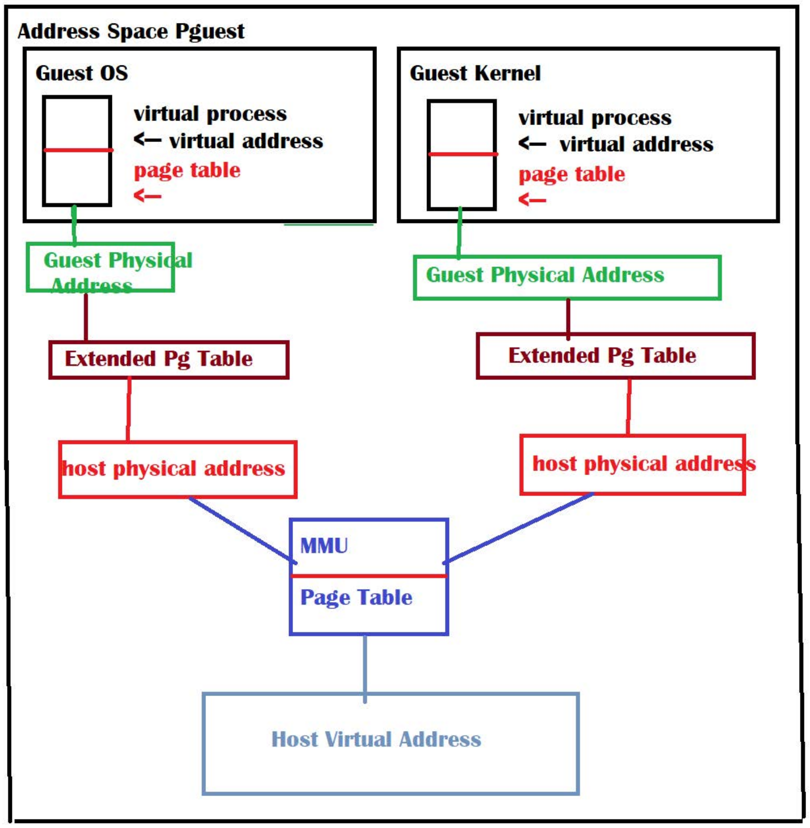
2. **Virtualization:** For this question you will need to study extra sources of information, such as technical papers and web pages. Please cite any sources of information with your answers.

   (a) (**10 points**) Before the introduction of hardware virtualization support, x86 architecture was not considered virtualizable in the "trap and emulate" sense. This was because of approximately 17 sensitive, unprivileged instructions. Briefly describe what is meant by a "sensitive instruction" in this case and give an example of how it is potentially problematic when constructing a virtual machine (VM). What is meant by the term "trap and emulate"? How then can the x86 support VMs when not all instructions are virtualizable?

a) Sensitive Instruction is when the virtual machine/hypervisor wants to trap/observe a privilege machine state, to give an unmodified OS the illusion it owns its hardware resources, to virtually run an OS.
b) This can be problematic when constructing a VM because:
In order for sensitive instructions to be virtualizable, they have to be a subset of privileged instructions, or else errors will occur.
Not all architectures are suitable for trap and emulation, and there's a loss in performance not related to the virtualization process. Ex: Different languages or different architecture - no translation.
Example: popf(restore %eflags from stack) - the instruction
We want to have int $13 (General Protection Fault) but we get
%eflags which is all bits on the stack
c) Trap and emulate means to emulate privileged instructions and registers to pretend the OS is still in kernel mode. When instructions are different from the OS, this is especially helpful, it doesn't execute code.
d) The x86 support VMs when not all instructions are virtualizable when:
you add another ring of privelege (-1) for virtual memory and porting the OS. Hardware is also added to the CPU to support virtualization, you need to find out what can be virtualized and what kind of hardware is needed in order to remove instructions.

b) (10 points) Consider that a guest OS is mapped to a process address space Pguest. Both applications and a guest kernel exist in Pguest. Each guest application is associated with a “virtual process” that behaves like a separate thread in Pguest. Likewise, the guest kernel is similar to a library but system calls from a guest application to this kernel cannot behave like direct function calls. They still need to operate like traditional system calls in a dual-mode system (e.g., using a mechanism similar to int 0x80 or sysenter in Linux on the x86).
Explain how one could virtualize system calls issued by a guest application within Pguest, so they are serviced by kernel code of the guest OS. How do you differentiate and control the switching between application and kernel stacks in Pguest? In your answer, include a diagram that shows how control is redirected between various parts of memory, to handle virtualized system calls.
HINT: Early versions of User-Mode Linux used a “tracing thread” technique similar to this.
You can assume this is a form of OS rather than machine virtualization, and there is no special hardware support for virtualization.

One could virtualize system calls issued by a guest application within Pguest, so they are serviced by kernel code of the guest OS by first using the Host Virtual Address to initialize memory management units, then it is connected to the host protected area that is connected to the extended page table, which is connected to the guest physical address to each Virtual machine's Memory management table. You go into the shared MMU page table and then you can differentiate and control switching between application and kernel stacks in Pguest.
Basically find a way to connect the Guest Virtual Addresses in the guest physical addresses, to the guest Host Virtual Address in the Host Physicial Address. In this diagram: is how the ACRN hypervisor system differentiates control between the switching between application and kernel stacks in Pguest. They use the page tables and reference accordingly.



3. **Process/Thread Scheduling:** Consider the following set of processes, with all times in milliseconds.

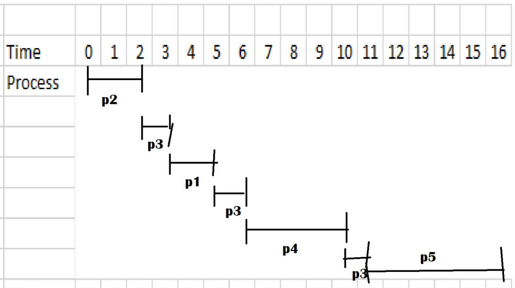
Process	Arrival Time	Burst/Computation Time	Priority (1=highest)	Deadline
P1	3	2	1	7
P2	0	2	3	5
P3	2	3	4	15
P4	6	4	2	11
P5	1	5	5	20

(a) (**20 points**) In the space below draw the Gantt charts illustrating the execution of these processes using the following scheduling algorithms: (1) Static Priority with preemption, (2) Shortest Job First, no preemption, (3) Round-Robin, no priority, quantum = 1, and (4) Earliest Deadline First, with preemption.

(b) (**8 points**) What are the *average waiting* and *average turnaround* times for each of the processes, using the scheduling algorithms above?

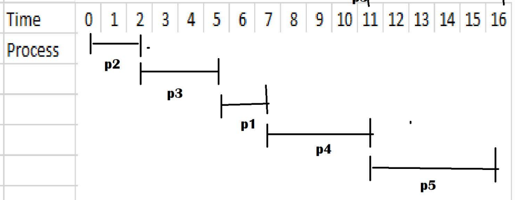
Static Priority

Process No.	Arrival Time	Burst Time	Completed Time	Turn Around Time	Waiting Time
P1	3	2	5	2	0
P2	0	2	2	2	0
P3	2	3	11	9	6
P4	6	4	10	4	0
P5	1	5	16	15	10



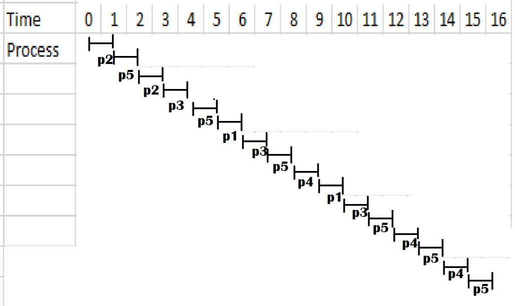
Shortest Job First, no preemption

Process No.	Arrival Time	Burst Time	Completed Time	Turn Around Time	Waiting Time
P1	3	2	7	4	2
P2	0	2	2	2	0
P3	2	3	5	3	0
P4	6	4	11	5	1
P5	1	5	16	15	10



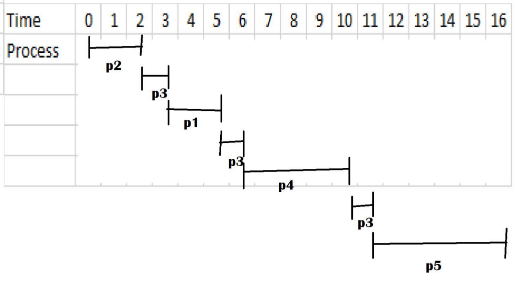
Round-Robin, no priority, quantum = 1

Process No.	Arrival Time	Burst Time	Completed Time	Turn Around Time	Waiting Time
P1	3	2	10	7	5
P2	0	2	3	3	1
P3	2	3	11	9	6
P4	6	4	16	10	6
P5	1	5	14	13	8



Earliest Deadline First, with preemption

Process No.	Arrival Time	Burst Time	Completed Time	Turn Around Time	Waiting Time
P1	3	2	5	2	0
P2	0	2	2	2	0
P3	2	3	11	9	6
P4	6	4	10	4	0
P5	1	5	16	15	10



Turn Around Time = Burst Time + Wait Time = Completed Time - Arrival Time

Static Priority			
Average Waiting Time	0+0+6+0+10 / 5	3.2	
Average Turnaround Time	2+2+9+4+15 / 5	6.4	
Shortest Job First, no preemption			
Average Waiting Time	2+0+0+1+10 / 5	2.6	
Average Turnaround Time	4+2+3+5+15 / 5	5.8	
Round-Robin, no priority, quantum = 1			
Average Waiting Time	5+1+6+6+8 / 5	5.2	
Average Turnaround Time	7+3+9+10+13 / 5	8.4	
Earliest Deadline First, with preemption			
Average Waiting Time	0+0+6+0+10 / 5	3.2	
Average Turnaround Time	2+2+9+4+15 / 5	6.4	

(c) **(5 points)** Consider a scheduling algorithm with $O(n)$ time complexity to schedule n processes (i.e., the scheduling latency is linearly-proportional to the number of processes ready for execution). Will the *average* scheduling latency be more or less for CPU-bound processes than I/O-bound processes? Briefly explain your answer.

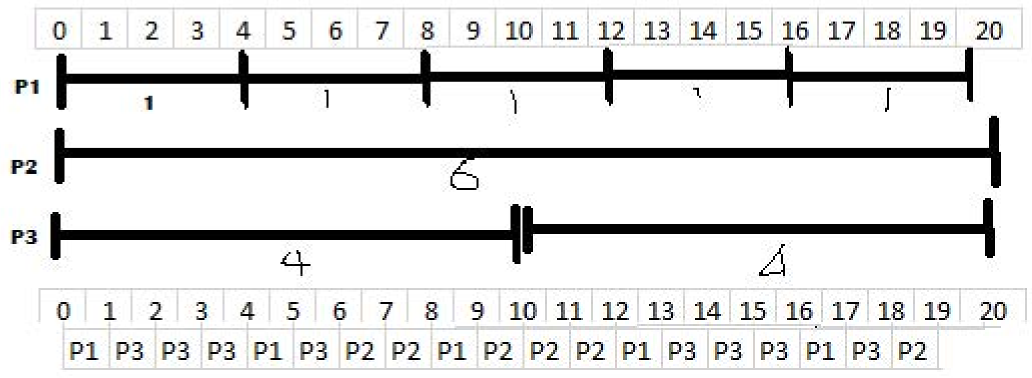
The average scheduling latency will be more and more I/O bound processes. As CPU gets faster processes tend to not increase in speed in proportion to the CPU, when CPU increases this allows more instructions to be executed in a window, so it goes more I/O bound as it doesn't shift in favor of the CPU.

(d) Rate Monotonic Scheduling (RMS) is a well-known, *preemptive* scheduling algorithm for real-time, *periodic* processes.

- (**6 points**) If the following three periodic processes all require scheduling for the first time, at time t, can a feasible schedule be constructed with RMS? Briefly explain your answer by showing the timeline sequence or proving schedulability using the completion time test. What happens if we use earliest deadline first scheduling, assuming deadlines are at the ends of request periods? Again, prove your answer either mathematically or by example.

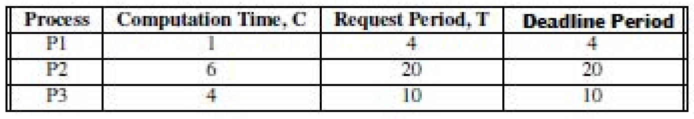
Process	Computation Time, C	Request Period, T
P1	1	4
P2	6	20
P3	4	10

LCM(4, 20 ,10) = 20
Priority: P1, P3, P2

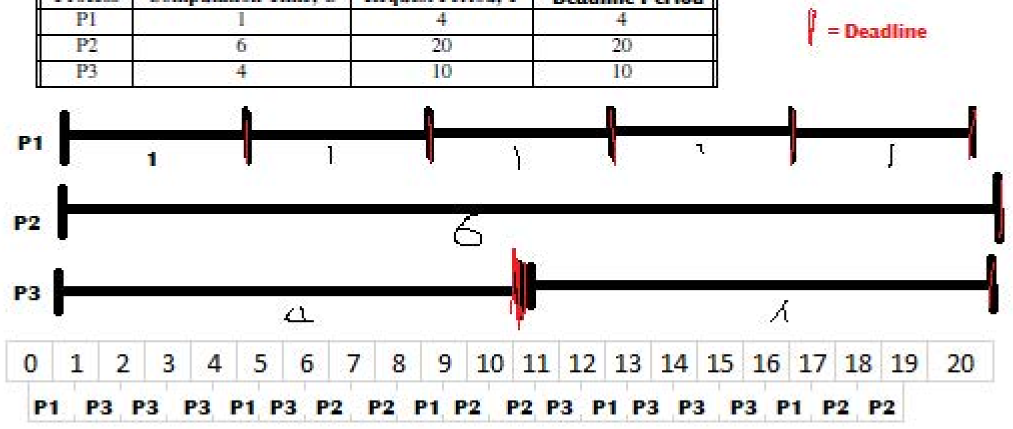


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
P1	P3	P3	P3	P1	P3	P2	P2	P1	P2	P2	P2	P1	P3	P3	P3	P1	P3	P2	

This is feasible when constructed with RMS. As shown in the figure above.



Process	Computation Time, C	Request Period, T	Deadline Period
P1	1	4	4
P2	6	20	20
P3	4	10	10

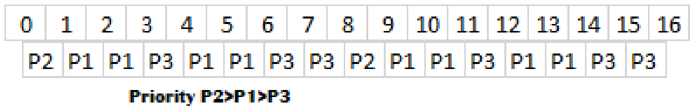


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
P1	P3	P3	P3	P1	P3	P2	P2	P1	P2	P2	P3	P1	P3	P3	P3	P1	P2	P2	

This is feasible with Earliest Deadline first scheduling as shown in the figure above, where at time 11, there's a tie breaker between P2 and P3.

- (**3 points**) Can RMS yield a feasible schedule for the following periodic processes? Briefly explain your answer.

Process	Computation Time, C	Request Period, T
P1	2	4
P2	1	8
P3	6	16



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
P2	P1	P1	P3	P1	P1	P3	P3	P2	P1	P1	P3	P1	P1	P3	P3	

Priority P2>P1>P3

Yes, RMS can yield a feasible schedule for the chart above, as shown in the Gaant chart also, above. It fits all the scheduling without breaking the rules.

- (**6 points**) One a single CPU, what is the least upper bound on utilization by a set of processes that yields a feasible schedule with RMS, if all processes have *harmonically-related* request periods? Prove your answer. NOTE: harmonically-related request periods implies each period is a multiple of all smaller periods.

The least upper bound on utilization is U = m(2^1/m - 1)
Proof:
base case
when M=1, we have 1 task and we can schedule it on 1 processor, Ci/Ti <= 1
Induction case
m=k+1, m >=2
k+1 summation(i=1)   Ci/Ti <= m(2^1/2 - 1)
k+1 summation(i=1)   Ci/Ti <= (n-1)(2^1/2-1)+(2^1/2-1)-Ck+1/Tk+1

Either 1) Ck+1/Tk+1 >= 2^1/2 -1 or 2) Ck+1/Tk+1 < 2^1/2 -1

1) holds for the equation above for k+1 summation(i=1) for the induction step for least upper bound.
2) Ck+1/Tk+1 < 2^1/2 -1 does not hold, therefor when it can't schedule,
   mi >= 2 and ui +ck+1/Tk+1 >mi+1 (2^(1/m^(i+1)) - 1), where ui <= 2^1/2 -1, this contradicts that the second processor must be uj <= 2^1/2 -1, because U would have to be U>(n^1/2 -1) for this to happen. So by proof by contradiction Tk+1 is the correct least lower bound for the processor.

U= m(2^1/m-1) is true.

Reference Links: https://www.cs.fsu.edu/~baker/papers/mpsched.pdf
https://www.sciencedirect.com/topics/computer-science/rate-monotonic-scheduling

(e) (**10 points**) Consider a proportional share scheduler (e.g., EEVDF) for a set of n tasks competing for a single CPU. If each task has a time quantum, q, what is the worst-case *lag* between ideal and actual usage of the CPU experienced by any one task? **Prove your answer**. NOTE: You can assume all tasks have equal weights (and, hence, priorities).

If each task has a time quantum, q, the worst case lag between ideal and actual usage of the CPU experienced by any one task is -q. A EEVDF provides lag bounds +/- q, but maintaining virtual time is minor but a significant part in proportional resource allocation, with it being better than sorting, O(n) in the worst case.

Schedule tasks such that performance is as close as possible in the fluid system
Si(t1,t2) = t1(integral)t2 (wi/jsummation(wj)) dt

Allocation error for task i at time t as:
lagi(t) = Si(t0,t) - si(t0,t)

If allocation is quantum based tasks can be behind or ahead of fluid schedule.

-lag is positive then task has more service time than in the fluid system

-lag is negative then task has less service time than in the fluid system

Some more properties of lag(t):

V(e) = V(t0) + s(t0,t)/w

V(d) = V(e)+c/w

Eligibility law: If a task has non-negative lag then it is eligible

Lag conservation law: for all t, i summation (lag(t)) = 0

Missed Deadline law: if task misses its deadline d then, lag(t) = remaining required service time

Preserved lateness law: If a task that misses a deadline at d completes execution at T, then for all t, T ≥ t > d, lagi (t) > 0
lagi (t) > remaining service time

Reference: http://www.cs.unc.edu/~jeffay/courses/pisa/eevdf.pdf

4. **Threads (15 points)** For this question you will have to study pthreads. The following piece of POSIX code implements a consumer, in a producer-consumer problem, using a shared circular buffer of N slots. Briefly explain what is wrong with the consumer and how it should be fixed:

```
typedef struct {
  char *data; // Slot data.
  size_t size; // Size of data.
  pthread_t id; // ID of destination thread.
} slot_t;

typedef struct {
  slot_t slot[N];
  int count; // Number of full slots in buffer.
  int in_marker; // Next slot to add data to.
  int out_marker; // Next slot to take data from.
  pthread_mutex_t mutex; // Mutex for shared buffer.
  pthread_cond_t occupied_slot; // Condition when >= 1 full buffer slot.
  pthread_cond_t empty_slot;    // Condition when 1 empty buffer slot.
} buffer_t;

char *consume (buffer_t *b) {
  char *item;
  pthread_t self=pthread_self();

  pthread_mutex_lock (&b->mutex); // Don't worry about checking return values
  if (b->count <= 0)
    pthread_cond_wait (&b->occupied_slot, &b->mutex);
  assert (b->count > 0);

  // Check id of target thread to see message is for this thread.
  if (b->slot[b->out_marker].id != self) {
    // Data is not for this thread.
    pthread_mutex_unlock (&b->mutex);
    return NULL;
  }
  item = (char *) malloc (b->slot[b->out_marker].size);
  memcpy (item, b->slot[b->out_marker].data, b->slot[b->out_marker].size);
  free (b->slot[b->out_marker].data);
  b->out_marker++;
  b->out_marker %= BUFFER_SIZE;
  b->count--;

  pthread_mutex_unlock (&b->mutex);
  pthread_cond_signal (&b->empty_slot);

  return (item);
}
```

In the code above, the error is that it consumes before it checks buffer. It might consume an empty buffer. It also signals late. How to fix it is to move the consuming part after you check the buffer. They should signal to the consumer right after checking buffer if its >= 0 for it to continue.

5. **Synchronization.**
(16 points) Consider the following assembly code to implement a spinlock on an x86 multicore processor:

```
spinlock_lock:
.test:
  bts $0, (mlock) // Assume mlock is a memory variable
  jc .test
  ret

spinlock_unlock:
  btr $0, (mlock)
  ret
```

State what is wrong with the above code to implement the lock and unlock functionality, and explain how to fix it. How would you use `bts` and `btr` to implement an improved version of `spinlock_lock` and `spinlock_unlock`, respectively? Show the assembly code for the improved versions.

There might be a deadlock issue, you must use lock or mutex to only have one CPU running on the system bus at one time.

```
spinlock_lock:
.test:
  bts $0, (mlock) // Assume mlock is a memory variable
  jc .test
  jc  .spinlock_unlock
  ret
spinlock_unlock:
  lock btr $0, (mlock)
  j2c  .spinlock_unlock
  ret
```