In God we trust

# Os lab EX4

Mani Hosseini 810102552
Shayan Maleki 810102515
Hamid Mahmoudi 810102549

## Part 1: Kernel analysis & advanced locks

### Q1: Path Control

Path control is the execution path through which the CPU enters and executes kernel code. It defines how control is transferred to the kernel and what rules and limitations apply during that execution.

There are three main types of control paths in an operating system. The first one is a **system call**, which is explicitly requested by a user process. A system call is **synchronous**, meaning it happens exactly at the point where the process invokes it, and it is done **intentionally** to request a service from the kernel. Since a system call is executed in process context, the process can **sleep or block** while waiting for resources such as I/O.

The second type is an **interrupt**, which is triggered by hardware events like timers or I/O devices. Interrupts are **asynchronous and unplanned**, meaning they can occur at any moment, even while another process or kernel code is running. Because interrupts must be handled quickly and may interrupt critical kernel sections, code executed in interrupt context **must not sleep**.

The third type is an **exception**, which is generated by the CPU when an error or special condition occurs, such as a page fault or division by zero. Exceptions are **synchronous** because they are caused by the currently executing instruction, but they are **unwanted and unplanned**, since the program does not intentionally request them.

### Q2: Kernel Reentrant

A reentrant kernel means that the kernel can be entered again while it is already running. This means that while a process is executing kernel code, another execution path can enter the kernel before the first one finishes.

In xv6, this situation can happen when a process is executing a system call and, during its execution, a hardware interrupt occurs. For example, a process may be inside a `read` system call when a disk interrupt happens. The CPU then pauses the execution of the system call and jumps to the disk interrupt handler, which causes the CPU to re-enter the kernel. At this point, two kernel execution paths exist at the same time: the interrupted system call and the interrupt handler.

If proper synchronization is not used in this situation, both kernel paths may access or modify shared kernel data structures simultaneously. This can lead to race conditions, inconsistent kernel state, and corruption of kernel data. In severe cases, this may cause kernel crashes or unpredictable system behavior.

## Q3: Process Context vs Interrupt Context

Process context means that a process owns what is running at that moment. The kernel is executing code on behalf of a specific process, usually due to a system call. Process context is **synchronous**, and since it is associated with a process, it is allowed to **sleep or block** while waiting for resources such as I/O. Code running in process context can also be **preempted** by interrupts while it is executing.

Interrupt context, on the other hand, occurs when hardware owns what is running. It is triggered by hardware events and is therefore **asynchronous**. Interrupt context is not associated with a specific process, and for this reason, code running in interrupt context **must not block or go to sleep**, because doing so can lead to deadlock or system instability. Interrupt handlers are expected to execute quickly and then return control to the previously running code.

## Q4: Why code in Interrupt Context must not block or sleep

Interrupt handlers should not block or go to sleep for several reasons. First, code running in interrupt context may not belong to a specific process, so putting it to sleep can be meaningless because there is no proper process context to block and later resume.

Second, interrupts are often used for critical system functions such as timer ticks. If a timer interrupt handler is blocked or delayed, the scheduler’s tick counting and time management can be disrupted, which may affect scheduling and prevent sleeping processes from waking up correctly.

Finally, interrupt handlers must execute quickly. If an interrupt handler takes too long or blocks, the system may lose important hardware events. For example, signals indicating that an I/O operation has finished or that a device buffer is full may be delayed or missed, leading to incorrect system behavior or performance problems.

## Q5: Why Spinlock and Disable Interrupts are used in Interrupt Context

As mentioned previously, in interrupt context the code must not go to sleep or block. Therefore, instead of using a sleeping lock, a **spinlock** is used so the handler can **busy-wait** for a short time until the shared resource becomes available. Spinlocks are suitable for interrupt context because they do not put the CPU to sleep and are intended for short critical sections.

Spinlocks are also important in **multi-core systems**, where disabling interrupts on one CPU does not prevent other CPUs from accessing the same shared data. In this case, the spinlock ensures mutual exclusion between different CPUs.

On the other hand, in a **single-processor system**, disabling interrupts is necessary to prevent the currently running code from being interrupted while holding a lock. If interrupts were not disabled, an interrupt handler could run on the same CPU and try to acquire the same lock, leading to a deadlock. By disabling interrupts, the interrupt handler is delayed until the critical section is finished, allowing the handler to execute quickly and safely afterward.

## Q6: Synchronization Approaches

A **spinlock** is suitable when the expected waiting time is short. In this approach, the CPU keeps checking the lock in a busy-wait loop until it becomes available. This works well for short critical sections and can be used in **interrupt context**, where sleeping is not allowed. However, spinlocks perform poorly for long waits because they **burn CPU cycles unnecessarily** and can cause performance and scalability issues under high contention.

A **sleep lock** is used when the waiting time is expected to be long, such as during I/O operations. Instead of busy-waiting, the process goes to sleep and allows the CPU to run other processes, which **saves CPU time and improves overall system throughput**. The disadvantage of sleep locks is their **higher overhead** due to sleeping and waking up processes, and they **cannot be used in interrupt context**. They are also more prone to deadlocks if used incorrectly.

**Lock-free programming** is an advanced synchronization approach that avoids using locks altogether. It relies on atomic operations and careful design to allow multiple threads to make progress without mutual exclusion. This approach **avoids deadlocks completely** and usually **scales better in high-contention environments**. However, lock-free programming is **much harder to design and debug**, and it requires very careful programming to ensure correctness.

## Q7: Why interrupts are disabled before acquiring a lock

In the `acquire` function, interrupts are disabled before taking the lock in order to prevent a deadlock situation on a single-core system. If interrupts are enabled, the CPU may be interrupted while holding a lock.

For example, a process running in kernel mode may successfully acquire a lock and then get interrupted before releasing it. If the interrupt handler that runs next tries to acquire the same lock, it will spin forever waiting for the lock to be released. However, the lock can only be released by the interrupted process, which cannot continue execution until the interrupt handler finishes. This creates a deadlock where neither side can make progress.

By disabling interrupts before acquiring the lock, the kernel ensures that no interrupt handler can interrupt the critical section and attempt to acquire the same lock. This prevents re-entry into kernel code on the same CPU and guarantees that the lock will be released safely before any interrupt handler runs.

## Q8 – Why xv6 uses `pushcli` and `popcli` instead of `cli` and `sti`

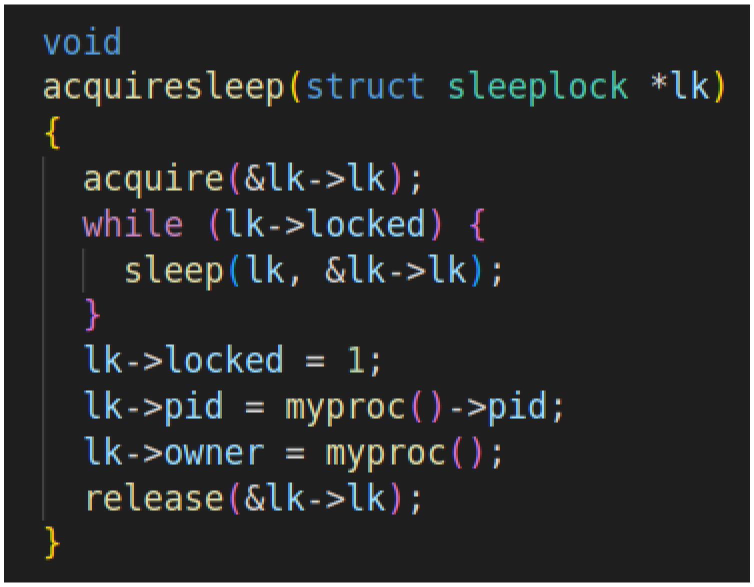
A nested critical section happens when kernel code enters a critical section and, while still inside it, calls another function that also enters a critical section. If raw `cli` and `sti` are used to disable and enable interrupts, this can cause race conditions.

For example, suppose a process enters function A and calls `cli` to disable interrupts. While still inside this critical section, function A calls function B, and function B also calls `cli`. When function B finishes, it calls `sti`, which re-enables interrupts. However, execution is still inside function A's critical section. If an interrupt occurs at this point, it happens inside a critical section, which can lead to race conditions or deadlock.

To avoid this problem, xv6 uses `pushcli` and `popcli`. These functions keep track of how many times interrupts have been disabled using a nesting counter. Interrupts are only re-enabled when the outermost critical section finishes. This guarantees that interrupts are not enabled too early and ensures correct behavior when critical sections are nested.

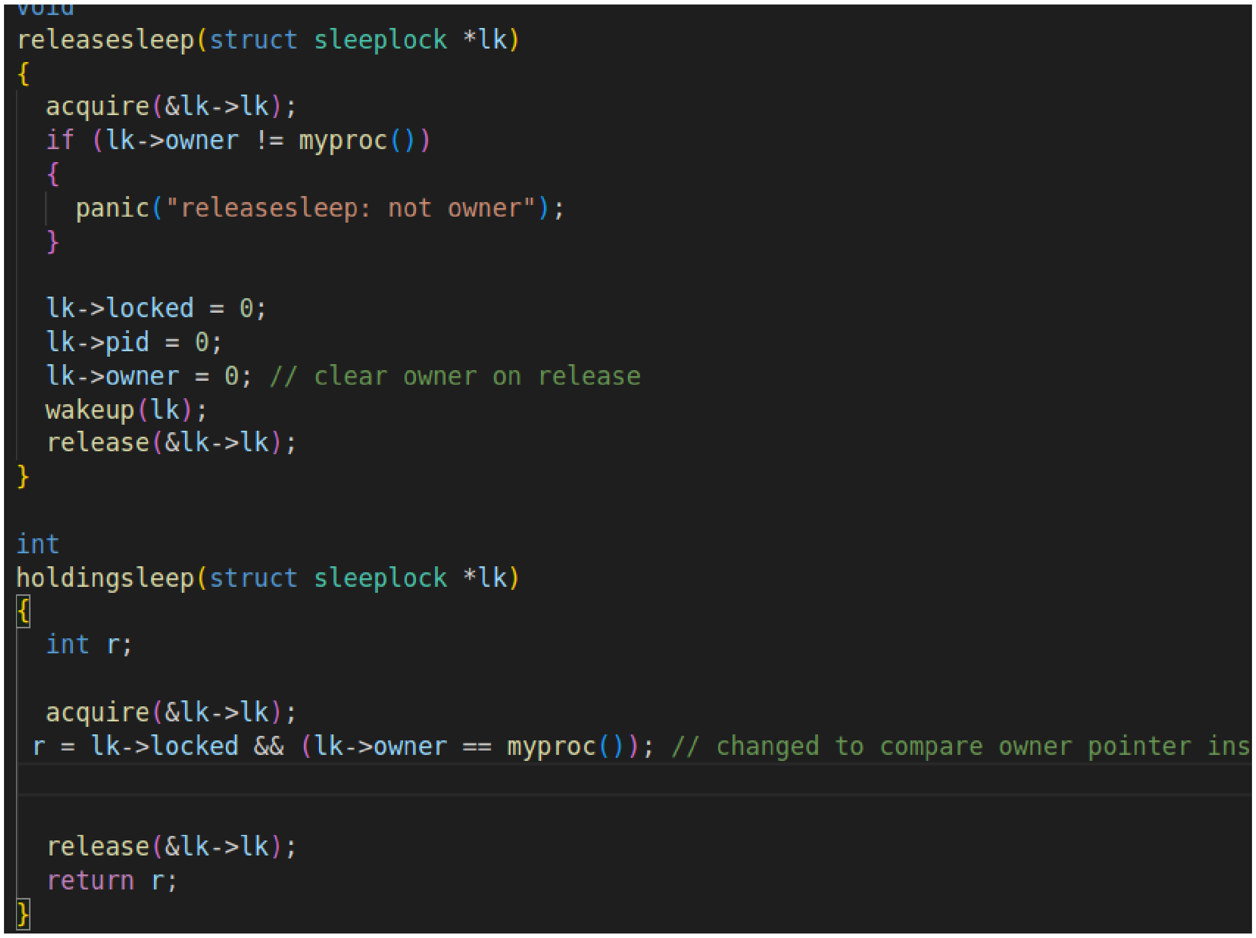
## Practical Implementation:

So first we added an ownership pointer to sleeplock  struct (it already had a pid lock but we think it is better to use a pointer to the proc in case that a pid is used multiple times)
Then we set the owner of the lock when we acquire it in a process, the code for this is shown below:



```c
void
acquiresleep(struct sleeplock *lk)
{
  acquire(&lk->lk);
  while (lk->locked) {
    sleep(lk, &lk->lk);
  }
  lk->locked = 1;
  lk->pid = myproc()->pid;
  lk->owner = myproc();
  release(&lk->lk);
}
```

And finally when releasing we check if the process which is releasing it is its owner



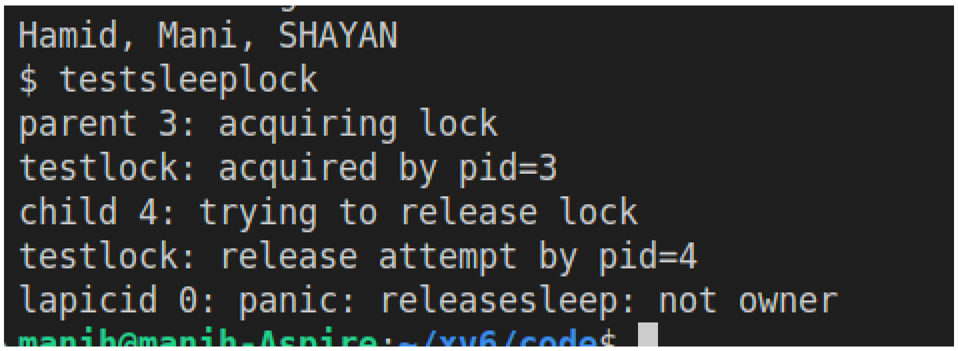
```c
void
releasesleep(struct sleeplock *lk)
{
  acquire(&lk->lk);
  if (lk->owner != myproc())
  {
    panic("releasesleep: not owner");
  }

  lk->locked = 0;
  lk->pid = 0;
  lk->owner = 0; // clear owner on release
  wakeup(lk);
  release(&lk->lk);
}

int
holdingsleep(struct sleeplock *lk)
{
  int r;

  acquire(&lk->lk);
 r = lk->locked && (lk->owner == myproc()); // changed to compare owner pointer ins


  release(&lk->lk);
  return r;
}
```

If it isn't, kernel should panic, the reason we chose to panic is trying to release a wrong lock is a fatal kernel code issue which could have very unpredictable consequence , so we panic to just stop the os right here.

To test the ownership enforcement of the sleeplock, an initial attempt was made to perform the entire test inside a single system call by calling fork() in the kernel and checking for pid == 0. However, this approach did not work because in xv6 the child process does not continue execution inside the same kernel system call. The child only observes a return value of zero when control returns to user space, not inside the kernel syscall implementation.

To correctly test the behavior, the test was redesigned using two separate system calls. The first system call allows the parent process to acquire a global test sleeplock in the kernel. Then, the parent forks in user space, where the child process correctly follows the pid == 0 path. The child then calls a second system call that attempts to release the same sleeplock.

Since the child process is not the owner of the lock, the kernel detects this and triggers a panic, which confirms that sleeplock ownership is properly enforced. This approach correctly matches the intended test scenario, where the parent acquires the lock and the child is prevented from releasing it.



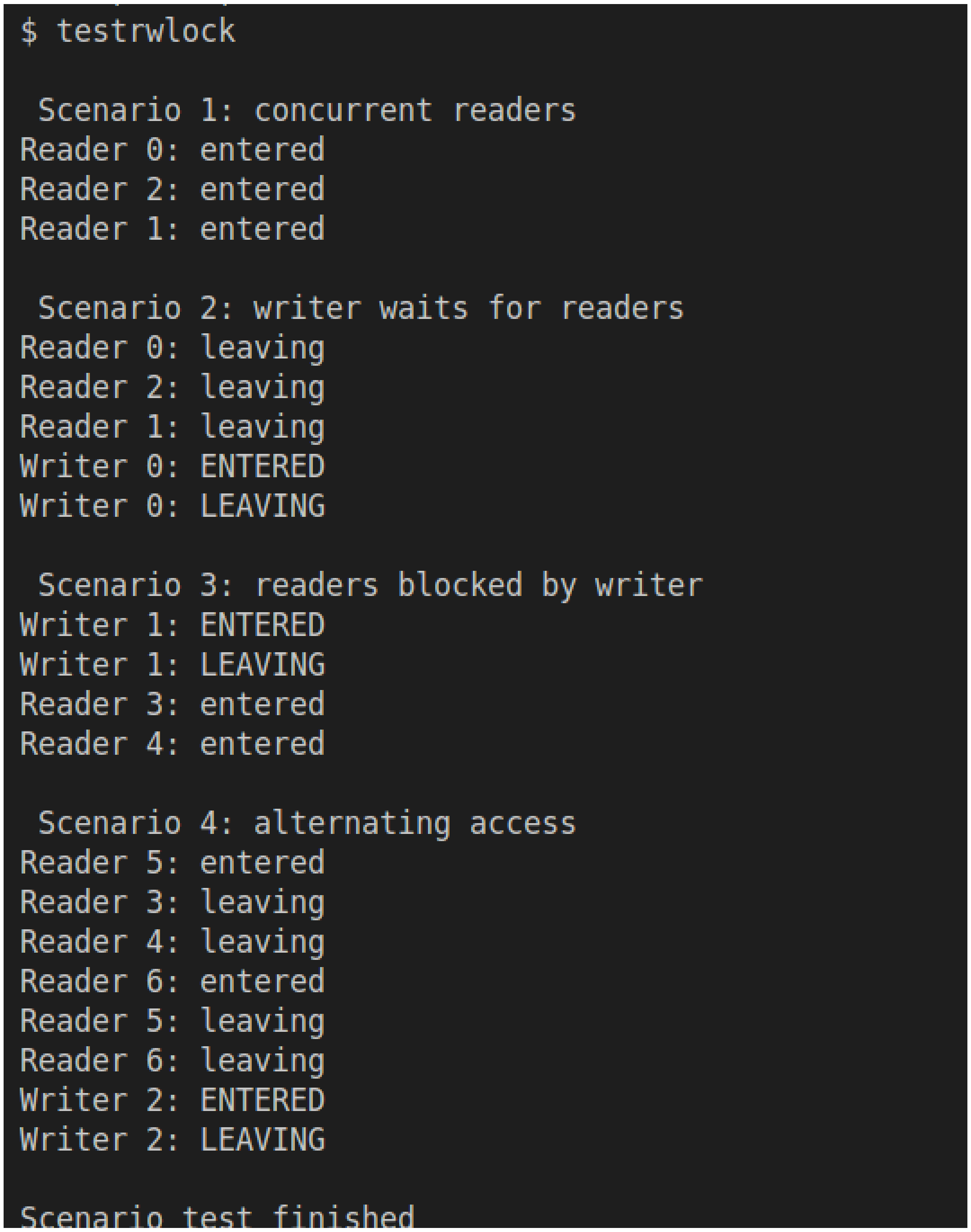
```
Hamid, Mani, SHAYAN
$ testsleeplock
parent 3: acquiring lock
testlock: acquired by pid=3
child 4: trying to release lock
testlock: release attempt by pid=4
lapicid 0: panic: releasesleep: not owner
```

## Bonus: Reader–Writer Lock Design and Test

To improve performance in cases where shared data is read more frequently than written, a reader–writer lock was designed and implemented. Unlike mutual exclusion locks, this lock allows multiple reader processes to enter the critical section simultaneously, while ensuring that writer processes have exclusive access. This design is useful for kernel data structures such as file lists, where concurrent reads are safe but writes must be serialized.

The lock was implemented using a combination of a spinlock and sleep–wakeup mechanisms, similar to the `sleeplock` design. An internal spinlock protects the lock's state variables, including the number of active readers and the presence of an active writer. Readers are allowed to enter as long as no writer is holding the lock, and writers are allowed to enter only when there are no active readers or writers. Processes that cannot enter the critical section are put to sleep and woken up when the lock state changes.

To test the correctness of the implementation, simple system calls were added to allow user programs to acquire and release the reader and writer modes of the lock. A user-level test program was then written to simulate multiple reader processes entering the critical section concurrently, followed by writer processes that only enter when the critical section is empty. The observed behavior confirms that readers can execute in parallel, while writers maintain exclusive access, demonstrating the correctness of the reader–writer lock implementation.



```
$ testrwlock

 Scenario 1: concurrent readers
Reader 0: entered
Reader 2: entered
Reader 1: entered

 Scenario 2: writer waits for readers
Reader 0: leaving
Reader 2: leaving
Reader 1: leaving
Writer 0: ENTERED
Writer 0: LEAVING

 Scenario 3: readers blocked by writer
Writer 1: ENTERED
Writer 1: LEAVING
Reader 3: entered
Reader 4: entered

 Scenario 4: alternating access
Reader 5: entered
Reader 3: leaving
Reader 4: leaving
Reader 6: entered
Reader 5: leaving
Reader 6: leaving
Writer 2: ENTERED
Writer 2: LEAVING

Scenario test finished
```

# Part 2: Scalability, CPU-per data & lock profiling

## Q1:

**Disabling Interrupts for Spinlocks:**

A processor must disable interrupts while holding a spinlock to prevent a deadlock on the same CPU. If interrupts were left enabled, an interrupt could arrive while the kernel is holding a lock. The CPU would then pause the current execution to run the interrupt handler. If that interrupt handler tries to acquire the *same* lock, it will enter an infinite spin loop waiting for the lock to be released. However, the lock will never be released because the thread holding it has been suspended by the interrupt handler itself.

## Q2:

**Deadlock with Interrupt Handlers:**

If interrupts remain active, a "single-CPU deadlock" occurs. **Example:**

1. **Thread A** acquires **Lock X**.
2. A timer interrupt fires. The CPU suspends **Thread A** and jumps to the **Interrupt Handler**.
3. The **Interrupt Handler** code tries to acquire **Lock X**.
4. The Handler sees the lock is held and begins **busy-waiting** (spinning) for it to open.
5. **Thread A** cannot run to release the lock because the CPU is stuck inside the Handler. The system freezes because the Handler is waiting for the Thread, and the Thread is waiting for the Handler to finish.

## Q3:

**Global Variables and Cache Coherence:**

When multiple cores constantly modify a shared global variable, a phenomenon called **Cache Thrashing** or **Cache Line Bouncing** occurs. Under the **MESI protocol**, before a core can write to the variable, it must invalidate copies of that data in all other caches (setting them to the **Invalid** state). If Core 1 writes to the variable, it invalidates Core 2's cache. When Core 2 tries to write immediately after, it must fetch the data back and invalidate Core 1's cache. This causes the data to "bounce" back and forth between caches, flooding the system bus with traffic and forcing the CPUs to stall constantly, which severely degrades performance.

## Q4:

**Per-CPU Variables** Using **Per-CPU:**

variables solves this by assigning a separate counter to each processor, stored at different memory addresses (and ideally different cache lines).

With per-CPU counters, Core 1 only writes to its local counter and Core 2 only writes to its local counter. Since they are modifying different memory locations, Core 1's write does not force an **invalidation** of Core 2's cache. Each core can keep its own cache line in the **Modified** state locally. This eliminates the "bouncing" on the bus, allowing all cores to increment their counters in parallel at full speed.

## Q5:

**Spinlock vs. Sleeplock:**

The fundamental difference is their waiting mechanism. **Spinlocks** (the first set in xv6) rely on **Busy Waiting**. The CPU stays active in a tight loop, repeatedly checking if the lock is free. This wastes CPU cycles but is faster for very short critical sections. **Sleeplocks** (the second set) yield the processor. If the lock is unavailable, the process puts itself to **sleep**, allowing the scheduler to switch context and run a different process.

- **Causes Busy Waiting:** Spinlock
- **Yields Processor:** Sleeplock

## Part 3: Priority locks & starvation

### Q1:

Yes, in the design of plock, starvation is possible for low-priority processes because the lock allocation policy strictly favors higher priorities. If a mechanism like aging is not implemented, a low-priority process might never receive the lock. For example, a process with low priority could wait indefinitely if a continuous stream of higher-priority processes keeps arriving and acquiring the lock before it has a chance.

### Q2:

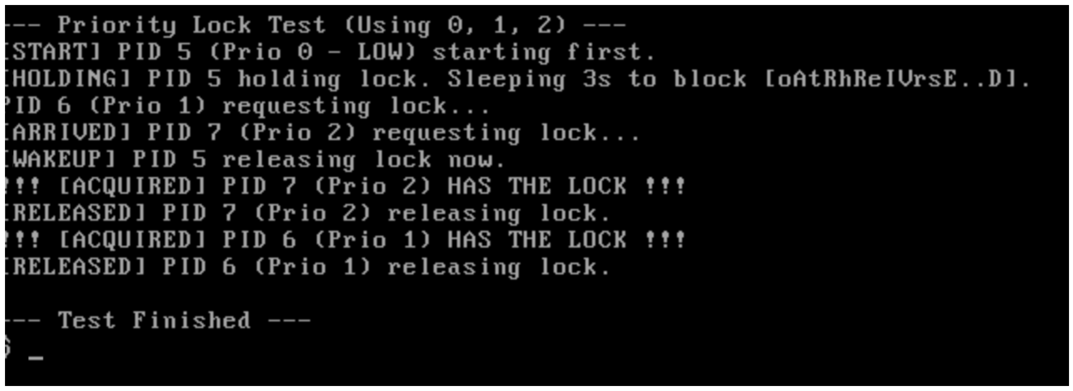
The ticket lock mechanism functions similarly to a numbering system in a bakery, designed to enforce fairness and order in resource access. Unlike standard spinlocks where processes race to acquire the lock, ticket lock assigns a unique sequential number to each arriving process. A process must wait until the current service counter matches its ticket number, ensuring a strict First-In-First-Out (FIFO) execution order that guarantees every process eventually acquires the lock, thereby eliminating the possibility of starvation.

### Q3:

In terms of fairness, the ticket lock mechanism is superior because it strictly follows a First-In-First-Out (FIFO) order, ensuring that every process eventually acquires the lock and eliminating the risk of starvation found in priority-based systems. Regarding complexity, the ticket lock is significantly simpler to implement as it only requires basic atomic arithmetic on counters, whereas a priority lock introduces higher overhead and complexity by necessitating the management of data structures like linked lists to store, search, and sort processes based on their priority levels.

### Practical section:

What we essentially did was that made a new lock that has linked list of processes and their priorities waiting for it to be free.
Then in the release syscall we assign the process that has the highest priority.
Here is the result for our test program:



```
-- Priority Lock Test (Using 0, 1, 2) ---
[START] PID 5 (Prio 0 - LOW) starting first.
[HOLDING] PID 5 holding lock. Sleeping 3s to block [oAtRhReIVrsE..D].
PID 6 (Prio 1) requesting lock...
[ARRIVED] PID 7 (Prio 2) requesting lock...
[WAKEUP] PID 5 releasing lock now.
!!! [ACQUIRED] PID 7 (Prio 2) HAS THE LOCK !!!
[RELEASED] PID 7 (Prio 2) releasing lock.
!!! [ACQUIRED] PID 6 (Prio 1) HAS THE LOCK !!!
[RELEASED] PID 6 (Prio 1) releasing lock.

-- Test Finished ---
$ _
```