Repository Link: https://github.com/Shahriar-0/Operating-System-Lab-Projects-F2024
Latest Commit Hash: f9c6e0d18ddaaf0e2bf6e4d941598f8938e6e520

Synchronization

1. Why are interrupts disabled while running the critical section?

Generally, interrupts are disabled during the execution of a critical section to prevent race conditions and ensure data integrity, more detailed reasons are as follows:

I. **Preventing Context Switches:** Disabling interrupts prevents context switches to another task that may access the same data. This is important because if an interrupt occurs while a process is in a critical section, the interrupt handler could potentially access the same data, leading to inconsistencies.

II. **Avoiding Deadlocks:** If an interrupt occurs during the execution of a critical section and the interrupt handler tries to acquire the same lock, it could lead to a deadlock situation. By disabling interrupts, we ensure that the lock is released properly before the interrupt handler can acquire it.

III. **Ensuring Atomicity:** Critical sections often involve updating shared resources. To ensure these operations are atomic (i.e., completed entirely or not at all), interrupts are disabled. This prevents an interrupt from causing the operation to be in a partially complete state.

IV. **Maintaining System Responsiveness**: Real-time systems must guarantee how long it takes to respond to interrupts, but critical sections can be arbitrarily long. Thus, interrupts are left off for the shortest time possible to maintain system responsiveness.

2. Explain the `pushcli` and `popcli` functions and their difference from `cli` and `sti`

In the xv6 operating system, `pushcli` and `popcli` are used in conjunction with `cli` and `sti` to manage interrupt handling. Here's a brief explanation of these functions:

- `cli`: This function disables interrupts on the CPU.

```
static inline void
cli(void) {
    asm volatile("cli");
}
```

- `sti`: This function enables interrupts on the CPU.

```
static inline void
sti(void) {
    asm volatile("sti");
}
```

`pushcli` and `popcli` are used to track the depth of `cli` calls and ensure that interrupts are enabled or disabled appropriately. Here's how they work:

- `pushcli`: This function disables interrupts (like `cli`) and increments a counter `ncli` to track the depth of `cli` calls. If it's the first `cli` call (`ncli == 0`), it saves the initial interrupt flag state to `intena`.

```
void pushcli(void) {
  int eflags;
  eflags = readeflags();
  cli();
  if (mycpu()->ncli == 0)
    mycpu()->intena = eflags & FL_IF;
  mycpu()->ncli += 1;
}
```

- `popcli`: This function decrements the `cli` counter `ncli`. If the counter reaches zero (meaning we're at the outermost `cli` call), and the initial interrupt flag `intena` was set, it enables interrupts (like `sti`). If interrupts are already enabled when `popcli` is called, it causes a kernel panic.

```
void popcli(void) {
  if (readeflags() & FL_IF)
    panic("popcli - interruptible");
  if (--mycpu()->ncli < 0)
    panic("popcli");
  if (mycpu()->ncli == 0 && mycpu()->intena)
    sti();
}
```

The key difference between `cli/sti` and `pushcli/popcli` is that `pushcli/popcli` are matched: it takes two `popcli` to undo two `pushcli`. Also, `pushcli/popcli` keeps track of the initial interrupt state and restores it when all `cli` calls have been popped. This is useful in situations where we have nested calls to disable/enable interrupts, and we want to ensure that interrupts are re-enabled only when all nested calls have been completed.

3. Why `spinlock` is not appropriate for single-core systems

Let's take a look and `spinlock` struct:

```
struct spinlock {
    uint locked; // Is the lock held?

    // For debugging:
    char* name;      // Name of lock.
    struct cpu* cpu; // The cpu holding the lock.
    uint pcs[10];    // The call stack (an array of program counters)
                     // that locked the lock.
};
```

- `uint locked`: This field indicates whether the lock is currently held or not. If `locked` is `1`, the lock is held; if `locked` is `0`, the lock is not held.
- `char* name`: This field is used for debugging purposes. It holds the name of the lock.
- `struct cpu* cpu`: This field is also used for debugging. It points to the CPU that currently holds the lock.
- `uint pcs[10]`: This field is an array of program counters. It's used to store the call stack that has locked the lock, which can be useful for debugging.

The `acquire` function is used to acquire a lock on a resource.

```
void acquire(struct spinlock *lk) {
  pushcli(); // disable interrupts to avoid deadlock.
  if(holding(lk))
    panic("acquire");

  // The xchg is atomic.
  while(xchg(&lk->locked, 1) != 0)
    ;

  // Tell the C compiler and the processor to not move loads or stores
  // past this point, to ensure that the critical section's memory
  // references happen after the lock is acquired.
  __sync_synchronize();

  // Record info about lock acquisition for debugging.
  lk->cpu = mycpu();
  getcallerpcs(&lk, lk->pcs);
}
```

I. `pushcli()`: This function disables interrupts on the local processor to avoid deadlock.
II. `if(holding(lk)) panic("acquire")`: This line checks if the lock is already held by the current CPU. If it is, the system panics, as this would mean a re-entrant locking situation, which is not supported in xv6.
III. `while(xchg(&lk->locked, 1) != 0)`: This line uses the `xchg` atomic instruction to try to acquire the lock, An atomic instruction in the context of programming refers to an indivisible operation, meaning it's executed in a single step and cannot be interrupted, It sets the `locked` field of the lock to `1` (indicating that the lock is now locked). If the lock was previously unlocked (`lk->locked` was `0`), `xchg` returns `0`, and the loop exits. If the lock was already locked, `xchg` returns `1`, and the loop continues to spin (busy-wait) until the lock becomes available.
IV. `__sync_synchronize()`: This line is a memory barrier. It tells the compiler and the processor to not reorder loads or stores across this point. This ensures that all memory references in the critical section happen after the lock is acquired.
V. The last part of the function records some information about the lock acquisition for debugging purposes.

Spinlocks, as used in the `acquire` function in xv6, are not typically suitable for single-core systems due to several reasons:

1. **Busy-Waiting**: The `acquire` function uses a busy-waiting loop (`while(xchg(&lk->locked, 1) != 0)`) to spin until the lock is available. In a single-core system, this spinning wastes CPU cycles because no other process can run and release the lock as there's only one core.
2. **Preemption and Deadlock**: If a process holding a spinlock is preempted (for example, due to a time slice expiry or an interrupt), no other process can acquire the spinlock and make progress. This can lead to a deadlock situation.
3. **Lack of Real Benefit**: Spinlocks are more beneficial in multi-core systems where another thread on a different core might release the lock while the current thread is spinning. In single-core systems, this isn't possible as only one thread can execute at a time.
4. **Inefficient Use of Resources**: Spinlocks can be much more expensive than other synchronization primitives like mutexes that put the thread to sleep when it cannot acquire the lock. Sleeping threads do not consume CPU cycles.
5. **Interrupt Disabling**: The `acquire` function disables interrupts using `pushcli`. This is done to avoid deadlocks and ensure atomicity. However, disabling interrupts for long periods can lead to missed hardware events and reduced system responsiveness.

Therefore, in single-core systems, other synchronization mechanisms like `mutexes` or `semaphores` are often preferred over spinlocks. These mechanisms put the thread to sleep

when it cannot acquire the lock, allowing the CPU to do other work instead of wasting cycles spinning. This makes them more efficient for single-core systems.

4. `amoswap` command in RISC-V

The `amoswap` command is an atomic instruction in the RISC-V instruction set. It's used for atomic memory operations, which are crucial in multi-threaded programming for synchronizing access to shared resources.

```
amoswap.w rd, rs2, (rs1)
```

This command atomically loads a 32-bit signed data value from the address in `rs1`, places the value into register `rd`, swaps the loaded value and the original 32-bit signed value in `rs2`, then stores the result back to the address in `rs1`.

So steps are:

I. **load:** The command first loads a 32-bit signed data value from the memory address specified by `rs1`.
II. **swap:** It then swaps this loaded value with the value in `rs2`.
III. **store:** Finally, it stores the swapped value (originally in `rs2`) back to the memory address specified by `rs1`.

The entire operation is atomic, meaning it's indivisible and executed in a single step. This ensures that if multiple threads are executing `amoswap` instructions concurrently on different cores, each `amoswap` operation will either complete entirely or not at all, but will never be in a partially complete state.

`amoswap` is typically used in implementing synchronization primitives like spinlocks. For example, it can be used to atomically check if a lock is free and acquire it if it is.

5. Process communication in `sleeplock` functions and `spinlock` usage producer-customer example

First, a brief explanation of `sleeplock` struck:

```
struct sleeplock {
    uint locked;        // Is the lock held?
    struct spinlock lk; // spinlock protecting this sleep lock

    // For debugging:
    char* name; // Name of lock.
    int pid;    // Process holding lock
};
```

- `uint locked;`: This field indicates whether the sleep lock is currently held or not. If `locked` is `1`, the lock is held; if `locked` is `0`, the lock is not held.
- `struct spinlock lk;`: This field is a spinlock that protects the sleep lock, to ensure that only one process at a time can modify the status of the sleep lock.
- `char* name;`: This field is used for debugging purposes. It holds the name of the lock.
- `int pid;`: This field is also used for debugging. It holds the process ID of the process that currently holds the lock.

Now, a brief explanation of `acquiresleep` and `releasesleep`. They are used to manage sleep locks. Sleep locks are a type of lock that puts the process to sleep if it cannot acquire the lock, allowing the CPU to do other work instead of wasting cycles spinning.

- `acquiresleep`: This function acquires a sleep lock. If the lock is already held, it puts the calling process to sleep, yielding the CPU to other processes. The process will remain asleep until the lock becomes available. This is different from a spinlock's `acquire` function, which would busy-wait (spin) until the lock is available. This is a form of process communication: the calling process is effectively saying, "Wake me up when the lock is available.".

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

- `releasesleep`: This function releases a sleep lock that was previously acquired by the calling process. If any processes are sleeping on the lock (i.e., they called `acquiresleep` but the lock was already held), one of them is woken up, and one of them is woken up. This is another form of process communication: the process is effectively saying, "The lock is now available, so one of the sleeping processes can wake up and acquire it."

```
void releasesleep(struct sleeplock* lk) {
    acquire(&lk->lk);
    lk->locked = 0;
    lk->pid = 0;
    wakeup(lk);
    release(&lk->lk);
}
```

This mechanism of putting processes to sleep and waking them up allows for efficient use of CPU resources. Instead of wasting CPU cycles spinning (busy-waiting) for the lock to become available (as in a spinlock), a process that cannot acquire a sleep lock simply goes to sleep, allowing the CPU to do other work. This makes them more efficient for systems with a single CPU core.

The producer-consumer problem is a classic example of multi-process synchronization in operating systems. It describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue. The producer's job is to generate data, put it into the buffer, and start again. At the same time, the consumer is consuming the data (i.e., removing it from the buffer), one piece at a time.

Spinlocks are not typically suitable for the producer-consumer problem due to several reasons:

I. **Busy-waiting:** Spinlocks use busy-waiting (spinning) when a lock cannot be acquired. If a producer (or consumer) cannot acquire the lock because the buffer is full (or empty), it will spin in a loop, checking the lock continuously. This wastes CPU cycles, especially in a single-core system where no other process can run and make progress (like consuming or producing data) while a process is spinning[4].
II. **Starvation:** If a producer (or consumer) spins waiting for the lock, and the scheduler always chooses another producer (or consumer) when the lock is released, the spinning producer (or consumer) could starve.
III. **No Sleep and Wakeup Mechanism:** Spinlocks do not have a built-in mechanism to put a process to sleep when it cannot acquire a lock and wake it up when the lock is available. This is a crucial requirement for the producer-consumer problem, where producers need to sleep when the buffer is full, and consumers need to sleep when the buffer is empty.

IV. **Inefficient Use of Resources**: As mentioned earlier, busy-waiting can lead to inefficient use of CPU resources. In contrast, other synchronization primitives like semaphores or condition variables allow a process to sleep when it cannot do useful work, thereby allowing the CPU to do other tasks.

Therefore, other synchronization mechanisms like `semaphores` or `condition variables` are often preferred over spinlocks for the producer-consumer problem. These mechanisms provide a way for a process to sleep when it cannot make progress and to wake up at the appropriate time, making them more efficient for this kind of problem.

6. Process state and `sched` function

```
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE }
```

This `enum` defines the possible states of a process:

- `UNUSED`: This state indicates that the process table entry is not currently being used by any process.
- `EMBRYO`: This state indicates that a process is being created. During this state, the process is allocated resources like memory and file descriptors.
- `SLEEPING`: This state indicates that a process is waiting for a resource to become available or an event to occur. A process in the `SLEEPING` state is not scheduled to run.
- `RUNNABLE`: This state indicates that a process is ready to run and is waiting to be assigned to a CPU. The process has all the resources it needs to run and is simply waiting for the scheduler to give it CPU time. A process can transition to the `RUNNABLE` state from other states in several ways:
  I. **Newly Created:** When a process is first created, it enters the `RUNNABLE` state after it has been initialized and is ready to start execution.
  II. **Resuming from Sleep:** If a process is in the `SLEEPING` state, waiting for a resource or an event, it becomes `RUNNABLE` when the resource becomes available or the event occurs.
  III. **Preempted:** If a process was `RUNNING` and is preempted by the scheduler (for example, due to a time slice expiry), it typically transitions back to the `RUNNABLE` state.
  IV. **I/O Completion:** A process that was waiting for an I/O operation to complete becomes `RUNNABLE` when the I/O operation finishes.
  V. **Released Lock:** If a process was waiting to acquire a lock (in a `SLEEPING` state), it becomes `RUNNABLE` when the lock is released.
  VI. **Signal Handling:** If a process is in the `SLEEPING` state and receives a signal that it needs to handle, it can become `RUNNABLE`.

- `RUNNING`: This state indicates that a process is currently running on a CPU. There can be at most one `RUNNING` process per CPU.
- `ZOMBIE`: This state indicates that a process has finished execution (it has exited), but its parent process has not yet collected its exit status. The process table entry must remain until the parent calls `wait()` to collect the exit status.

The `sched` function is responsible for scheduling processes in the operating system. It's part of the process scheduler, which decides which process to run next. It works like this:

I. **Context Switching:** At a low level, xv6 performs two kinds of context switches, from a process's kernel thread to the current CPU's scheduler thread, and from the scheduler thread to a process's kernel thread. xv6 never directly switches from one user-space process to another; this happens by way of a user-kernel transition (system call or interrupt), a context switch to the scheduler, a context switch to a new process's kernel thread, and a trap return.
II. **Process States:** The scheduler's job is to pick a process from the `RUNNABLE` state and switch it to the `RUNNING` state.
III. **Sleep and Wakeup Mechanism:** xv6 provides sleep and wakeup functions, that are equivalent to the wait and signal functions of a `conditional variable`. The sleep and wakeup functions must be invoked with a lock that ensures that the sleep and wakeup procedures are completed atomically.
IV. **Multiplexing:** If two processes want to run on a single CPU, xv6 multiplexes them, switching many times per second between executing one and the other. This multiplexing creates the illusion that each process has its CPU.

A more detailed explanation can be found on "*EX3 - Report*".

7. Modifying `Sleeplock` functions so that only the owner process can unlock it, what is the equivalent in Linux?

```
void releasesleep(struct sleeplock* lk) {
    acquire(&lk->lk);

    // only the process that has the lock can release it
    if (lk->pid != myproc()->pid)
        return;

    lk->locked = 0;
    lk->pid = 0;
    wakeup(lk);
    release(&lk->lk);
}
```

The equivalent of this in Linux is mutex lock. A mutex (short for mutual exclusion) is a synchronization primitive that can be used to protect shared data from being simultaneously accessed by multiple threads. A calling thread owns a mutex from the time that it successfully calls either `lock` or `try_lock` until it calls unlock. It's implemented like this:

```
struct mutex {
	atomic_long_t		owner;
	raw_spinlock_t		wait_lock;
#ifdef CONFIG_MUTEX_SPIN_ON_OWNER
	struct optimistic_spin_queue osq; /* Spinner MCS lock */
#endif
	struct list_head	wait_list;
#ifdef CONFIG_DEBUG_MUTEXES
	void			*magic;
#endif
#ifdef CONFIG_DEBUG_LOCK_ALLOC
	struct lockdep_map	dep_map;
#endif
};
```

- `atomic_long_t owner`: This field contains a pointer to the current owner of the mutex. It's `NULL` if the mutex is not currently owned.
- `raw_spinlock_t wait_lock`: This is a spinlock that protects the mutex. It's used to ensure that only one process at a time can check or change the status of the mutex.
- `struct optimistic_spin_queue osq`: This field is used when the `CONFIG_MUTEX_SPIN_ON_OWNER` option is enabled. It's a queue of threads spinning on the mutex.
- `struct list_head wait_list`: This is a list of processes waiting for the mutex to become available.
- `void *magic`: This field is used for debugging when the `CONFIG_DEBUG_MUTEXES` option is enabled.
- `struct lockdep_map dep_map`: This field is used for tracking lock dependencies when the `CONFIG_DEBUG_LOCK_ALLOC` option is enabled.

8. `Lock-free` algorithm

`Lock-free` algorithms and `programming with locks` are two different approaches to handling concurrency in multi-threaded programming.

`Lock-free` algorithms are carefully designed data structures and functions that allow for multiple threads to attempt to make progress independently of one another. They use atomic RMW[1] primitives that the hardware must provide, the most notable of which is CAS[2].

A `lock-free` algorithm is non-blocking, meaning that the failure or suspension of any thread cannot cause the failure or suspension of another thread. An algorithm is lock-free if, when the program threads are run for a sufficiently long time, at least one of the threads makes progress. They can improve performance on a multi-core processor because access to the shared data structure does not need to be serialized to stay coherent.

The traditional approach to multi-threaded programming is to use `locks` to synchronize access to shared resources. Synchronization primitives such as `mutexes`, `semaphores`, and critical sections are all mechanisms by which a programmer can ensure that certain sections of code do not execute concurrently if doing so would corrupt shared memory structures.

If one thread attempts to acquire a lock that is already held by another thread, the thread will block until the lock is free. Using `locks` also involves a trade-off between coarse-grained locking, which can significantly reduce opportunities for parallelism, and fine-grained locking, which requires more careful design, increases locking overhead, and is more prone to bugs.

Comparison:

- `Lock-free` algorithms eliminate a lot of the contention caused by locking. They handle communication, data sharing, and other mechanisms that usually require threads to be synchronized differently to avoid the need for locks.
- On a system using a `lock-free` algorithm, it is guaranteed that some thread is making progress. This is in contrast to `lock-based` programming where a thread can be blocked, halting its progress.
- `Lock-free` algorithms can be more complex to implement correctly compared to `lock-based` programming.
- `Lock-free` algorithms can potentially provide better performance than `lock-based` programming, especially in systems with high contention and in real-time systems.

[1] read-modify-write

[2] compare and swap

Per-core Variable

9. Cache coherency

Cache coherence refers to the consistency and synchronization of data stored in different caches within a multiprocessor or multicore system. In such systems, each processor or core typically has its cache memory to improve performance. Cache memory plays a crucial role in computer architecture by providing fast access to frequently used data. Brief explanation:

I. **Uniformity of Shared Resource Data:** Cache coherence is the uniformity of shared resource data that ends up stored in multiple local caches. When clients in a system maintain caches of a common memory resource, problems may arise with incoherent data.
II. **Concurrency Issues:** This is particularly the case with CPUs in a multiprocessing system. Suppose both clients have a cached copy of a particular memory block from a previous read. If one client updates/changes that memory block, the other client could be left with an invalid cache of memory without any notification of the change.
III. **Maintaining a Coherent View:** Cache coherence is intended to manage such conflicts by maintaining a coherent view of the data values in multiple caches. Coherent caches mean that the value in all the caches' copies is the same.
IV. **Requirements for Cache Coherence:** The following are the requirements for cache coherence:
    - **Write Propagation:** Changes to the data in any cache must be propagated to other copies (of that cache line) in the peer caches.
    - **Transaction Serialization:** Reads/Writes to a single memory location must be seen by all processors in the same order.
V. **Definition:** Coherence defines the behavior of reads and writes to a single address location. One type of data occurring simultaneously in different cache memory is called cache coherence, or in some systems, global memory.

To address these cache coherency issues at the hardware level, there are several protocols, among which the most commonly used are the `MESI` (Modified, Exclusive, Shared, Invalid) and its successors, `MSI` (Modified, Shared, Invalid), `MOESI` (Modified, Owner, Exclusive, Shared, Invalid), and `MESIF` (Modified, Exclusive, Shared, Invalid, Forward).

I. **Modified:** The processor has modified the block. The block is only in this cache and has to be written back to main memory before it is replaced.

II. **Exclusive:** The block is only in this cache and matches the main memory. The block can be changed to the modified state without informing other processors.
III. **Shared:** This block is unmodified and could be present in other caches.
IV. **Invalid:** This block is either empty or contains data that is potentially incorrect (i.e., it doesn't correspond to the main memory).

The protocol maintains consistency by enforcing rules that dictate how and when a processor can make a transition between states. For example, when a processor modifies a block, it informs all other processors about this change (broadcast), forcing all the other copies of that block to transition to the invalid state. So, if another processor needs that block, it has to fetch it from the main memory or the processor that owns this block, ensuring it always gets the most recent data

10. `Ticket lock`

A ticket lock is a synchronization mechanism or locking algorithm, that is a type of `spinlock` that uses "tickets" to control which thread of execution is allowed to enter a critical section. The basic concept of a ticket lock is similar to the `ticket queue` management system used in bakeries and delis to serve customers in the order that they arrive.

There are two integer values which begin at 0. The first value is the queue ticket, the second is the dequeue ticket. When a thread arrives, it atomically obtains and then increments the queue ticket. It then compares its ticket value, before the increment, with the dequeue ticket's value. If they are the same, the thread is permitted to enter the critical section. If they are not the same, then another thread must already be in the critical section and this thread must be busy-waited or yield.

From a cache coherence perspective, ticket locks can be analyzed as follows:

- Cache coherence ensures that a lock update is seen by other processors. The process that acquires the lock in an exclusive state gets to update the lock first.
- The lock can be cached, and cache coherence ensures that a lock update is seen by other processors. The process that acquires the lock in an exclusive state gets to update the lock first.
- The ticket lock is indeed fair, but its performance is just about on par with the `pthread spinlock` algorithm. The introduction of ticket lock is mainly because of fairness reasons.

11. Defining per-core data in Linux at compile time

Defining per-core data can help reduce cache coherence overhead in multi-core systems. Here's how:

I. **Reduced Cache Invalidation:** When data is shared among cores, any modification by one core leads to invalidation of the cached data in other cores to maintain coherence. This is known as the `cache invalidation` problem. By defining per-core data, each core operates on its own data, reducing the need for invalidation and thus reducing the overhead associated with maintaining cache coherence.
II. **Decreased Inter-Core Communication:** Shared data requires cores to communicate frequently to ensure that all cores have a consistent view of the data. This communication can be expensive in terms of time and resources. Per-core data reduces the need for such communication, thereby reducing the overhead.
III. **Avoidance of False Sharing:** False sharing occurs when cores inadvertently share cache lines, leading to unnecessary cache coherence traffic. By aligning per-core data structures with cache line boundaries, false sharing can be avoided.
IV. **Improved Cache Utilization:** By ensuring that each core operates on its own data, the efficiency of the cache can be improved. This is because the data a core needs is more likely to be in its cache, reducing the need for expensive memory accesses.
V. **Increased Parallelism:** Per-core data can increase the level of parallelism in the system. Since each core operates on its own data, more operations can be performed in parallel without the need for synchronization.

For implementing this in Linux we follow these steps

I. *Identifying the data*: Determine what data we want to associate with each core. This could be performance counters, process information, or other kernel data.

Using the CPU-specific data structure: Linux provides a `per_cpu` macro that can be used to define CPU-specific variables. so We can declare a per-CPU variable using the `DEFINE_PER_CPU` macro, or an array of per-CPU variables using the `DEFINE_PER_CPU_SHARED_ALIGNED` macro.

II. *Accessing the data:* We can access per-CPU variables using the `get_cpu_var(var)` and `put_cpu_var(var)` functions. These functions disable preemption, so we should minimize the code between these calls.
III. *Compiling the kernel:* Once we've made our changes, we'll need to compile the kernel. This typically involves running make, possibly with the -jN option to speed up the build on multi-core systems.

Counting the Number of System Calls

At first we change the number of `CPUs` in `makefile` to make sure it will run with 4 cores:

```
CPUS := 4
```

And we limit the maximum number of CPUs to 4:

```
#define NCPU        4                // maximum number of CPUs
```

Then we define a struct to help us with this task, and we `extern` it to use it another files:

```
struct nsyslock {
    struct spinlock lk;
    int n;
};
extern struct nsyslock nsys;
```

```
struct cpu {
    uchar apicid;               // Local APIC ID
    struct context* scheduler; // swtch() here to enter scheduler
    struct taskstate ts;        // Used by x86 to find stack for
interrupt
    struct segdesc gdt[NSEGS]; // x86 global descriptor table
    volatile uint started;      // Has the CPU started?
    int ncli;                   // Depth of pushcli nesting.
    int intena;                 // Were interrupts enabled before
pushcli?
    struct proc* proc;          // The process running on this cpu or
null
    int nsyscall;               // number of system calls
};
```

```
struct cpu cpus[NCPU];
int ncpu;
```

For initializing we reset variables before any execution, so we add this to the end of `exec` function:

```
acquire(&nsys.lk);
nsys.n = 0;
release(&nsys.lk);
cpus[0].nsyscall = 0;
cpus[1].nsyscall = 0;
cpus[2].nsyscall = 0;
cpus[3].nsyscall = 0;
```

exec.c: 104

Then we define a system call like this to counter the number of calls

```
void getnsyscall(void) {
    cprintf("%d, %d, %d, %d, ",
        cpus[0].nsyscall,
        cpus[1].nsyscall,
        cpus[2].nsyscall,
        cpus[3].nsyscall
    );
    acquire(&nsys.lk);
    cprintf("%d\n",
        nsys.n
    );
    release(&nsys.lk);
}
```

In each system call we update it like this, due to `syscall(void)` will be called at each system call.

```
void syscall(void) {
    int num;
    struct proc* curproc = myproc();

    num = curproc->tf->eax;
    if (num > 0 && num < NELEM(syscalls) && syscalls[num]) {
        curproc->tf->eax = syscalls[num]();
    }
    else {
        cprintf("%d %s: unknown sys call %d\n",
                curproc->pid, curproc->name, num);
        curproc->tf->eax = -1;
    }
```

```
    cli();
    int CPUid = cpuid();
    sti();
    cpus[CPUid].nsyscall++;
    acquire(&nsys.lk);
    nsys.n++;
    release(&nsys.lk);
}
```

The test program for this is written as follows:

```
int main() {
    unlink("ns.txt");
    int fd = open("ns.txt", O_CREATE | O_WRONLY);

    for (int i = 0; i < NPROCESS; i++) {
        if (!fork()) {
            write(fd, "G#17", 4);
            exit();
        }
        else
            continue;
    }
    for (int i = 0; i < NPROCESS; i++)
        wait();

    write(fd, "\n", 1);
    close(fd);
    nsyscalls();
    exit();
}
```

```
$ ns
11, 9, 13, 2, 35
$ ns
12, 16, 4, 3, 35
$ ns
13, 9, 12, 1, 35
$ ns
16, 10, 7, 2, 35
$ ns
7, 5, 17, 6, 35
$ ns
6, 19, 5, 5, 35
$
```

Synchronization with Priority Lock

At first we define `prioritylock` just like `spinlock`:

```
struct prioritylock {
    uint locked;        // Is the lock held?
    struct spinlock lk; // spinlock protecting this priority lock
    int inqueue;        // number of processes waiting in queue
    int lockreq[NPROC]; // pid of processes which requested priority
lock

    int pid;    // Process holding lock
    char* name; // Name of lock

};
```

- `uint locked`: This is a flag indicating whether the lock is currently held or not. If the lock is held, this value is 1; otherwise, it's 0.
- `struct spinlock lk`: This is a spinlock that protects the priority lock. A spinlock is a simple type of lock where a thread simply waits in a loop ("spins") repeatedly checking until the lock becomes available.
- `int inqueue`: This is the number of processes that are currently waiting in the queue for the lock.
- `int lockreq[NPROC]`: This is an array that holds the process IDs of the processes which have requested the priority lock. NPROC is a constant that represents the maximum number of processes.
- `int pid`: This is the process ID of the process that is currently holding the lock.
- `char* name`: This is the name of the lock. It's used for debugging purposes.

This structure is used in conjunction with the functions that are going to be elaborated to manage access to shared resources in a way that gives priority to certain processes (those with smaller process IDs).

```
void showlockqueue(struct prioritylock* lk) {
    acquire(&lk->lk);
    cprintf("[");
    for (int i = 0; i < lk->inqueue; i++) {
        cprintf("%d", lk->lockreq[i]);
        if (i != lk->inqueue - 1)
            cprintf(", ");
    }
    cprintf("]\n");
    release(&lk->lk);
}
```

- `void showlockqueue(struct prioritylock* lk)`: This function prints the queue of lock requests. It acquires the lock, prints the process IDs in the queue, and then releases the lock.

```
void enqueue(struct prioritylock* lk, int pid) {
    if (lk->inqueue == NPROC)
        return;

    int pos = 0;
    while (lk->lockreq[pos] > pid)
        pos++;

    for (int i = lk->inqueue; i > pos; i--) {
        lk->lockreq[i] = lk->lockreq[i - 1];
    }

    lk->lockreq[pos] = pid;
    lk->inqueue++;
}
```

- `void enqueue(struct prioritylock* lk, int pid)`: This function adds a process ID to the queue of lock requests in a sorted order (smaller process IDs have higher priority). If the queue is full, it does nothing.

```
int isprior(struct prioritylock* lk) {
    if (lk->inqueue == 0 || lk->lockreq[0] != myproc()->pid) {
        return 0;
    }

    for (int i = 0; i < lk->inqueue - 1; i++) {
        lk->lockreq[i] = lk->lockreq[i + 1];
    }
    lk->lockreq[lk->inqueue - 1] = 0;
    lk->inqueue--;

    return 1;
}
```

- `int isprior(struct prioritylock* lk)`: This function checks if the first process in the queue is the current process. If it is, it removes the process from the queue and returns 1. Otherwise, it returns 0.

```
void initprioritylock(struct prioritylock* lk, char* name) {
    initlock(&lk->lk, "p_s_lock");
    lk->name = name;
    lk->locked = 0;
    lk->pid = 0;
    memset(lk->lockreq, 0, NPROC);
}
```

- `void initprioritylock(struct prioritylock* lk, char* name)`: This function initializes a priority lock. It sets the name, resets the lock status, and clears the queue.

```
void acquirepriority(struct prioritylock* lk) {
    enqueue(lk, myproc()->pid);
    while (1) {
        acquire(&lk->lk);
        if (lk->locked == 1 || !isprior(lk)) {
            release(&lk->lk);
            continue;
        }
        else
            break;
    }
    lk->locked = 1;
    lk->pid = myproc()->pid;
    release(&lk->lk);
}
```

- `void acquirepriority(struct prioritylock* lk)`: This function tries to acquire the priority lock. It adds the current process to the queue and then enters a loop where it tries to acquire the lock only if it's not locked and the current process has the highest priority.

```
void releasepriority(struct prioritylock* lk) {
    acquire(&lk->lk);
    lk->locked = 0;
    lk->pid = 0;
    release(&lk->lk);
}
```

- `void releasepriority(struct prioritylock* lk)`: This function releases the priority lock. It sets the lock status to unlocked and clears the process ID.

```
int isprioritylocked(struct prioritylock* lk) {
    int r;

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

- `int isprioritylocked(struct prioritylock* lk)`: This function checks if the priority lock is locked. It returns 1 if the lock is locked, and 0 otherwise.

```
int holdingpriority(struct prioritylock* lk) {
    int r;

    acquire(&lk->lk);
    r = lk->locked && (lk->pid == myproc()->pid);
    release(&lk->lk);
    return r;
}
```

- `int holdingpriority(struct prioritylock* lk)`: This function checks if the current process is holding the priority lock. It returns 1 if the current process is holding the lock, and 0 otherwise.

At each acquire and release in priority lock we need to check 3 things:

1. we need to check if a process has already acquired the lock, it can not acquire it again.

```
int pacquire(void) {
    if(holdingpriority(&plock))
        return -1;
    acquirepriority(&plock);
    return myproc()->pid;
}
```

2. A lock can only be released by its owner.

```
int prelease(void) {
    if(!isholdingpriority(&plock))
        return -1;
    releasepriority(&plock);
    return 0;
}
```

3. A process must release a lock before exiting, so if a process try to exit and it has held the lock, we release it in `exit()`:

```
if(isholdingpriority(&plock))
        releasepriority(&plock);
```

proc.c:254

For testing the priority lock:

```
int main() {
    char numstr[NUMBUF];
    unlink("plock.txt");
    int fd = open("plock.txt", O_CREATE | O_WRONLY);
    for (int i = 0; i < NPROCESS; i++) {
        if (!fork()) {
            int pid = pacquire();
            sleep(100);
            pqueue();
            memset(numstr, 0, NUMBUF);
            intToChar(pid, numstr);
            numstr[strlen(numstr)] = '-';
            write(fd, numstr, strlen(numstr));
            printf(1, "process %d done!\n", pid);
            prelease();
            exit();
        }
        else
            continue;
    }
    while (wait() != -1) ;
    write(fd, "\n", 1);
    close(fd);
    exit();
}
```

```
$ prio
[13, 12, 11, 10, 9, 8, 7, 6, 5]
process 4 done!
[12, 11, 10, 9, 8, 7, 6, 5]
process 13 done!
[11, 10, 9, 8, 7, 6, 5]
process 12 done!
[10, 9, 8, 7, 6, 5]
process 11 done!
[9, 8, 7, 6, 5]
process 10 done!
[8, 7, 6, 5]
process 9 done!
[7, 6, 5]
process 8 done!
[6, 5]
process 7 done!
[5]
process 6 done!
[]
process 5 done!
$ cat plock.txt
4-13-12-11-10-9-8-7-6-5-
$
```

Process Starvation in Priority Lock

In our implementation, processes with smaller process IDs always have higher priority. This means that if a process with a larger process ID requests the lock, but processes with smaller IDs keep coming in and requesting the lock, the process with the larger ID could potentially wait indefinitely, leading to starvation.

This is a common problem in priority scheduling and priority locking systems. One common solution is to implement some form of aging, where the priority of waiting processes increases the longer they wait. However, this would require modifying current implementation. Another solution could be to limit the number of times a process can acquire the lock consecutively, but this might not be suitable for all scenarios.

Ticket Lock vs Priotiy Lock

I. **Ticket Lock:**
1) A `ticket lock` is a type of spinlock that uses “tickets” to control which thread of execution is allowed to enter a critical section.
2) It works on a `FIFO`[3] basis, ensuring fairness in lock acquisition.
3) When a thread arrives, it atomically obtains and then increments the `queue ticket`. It then compares its `ticket value`, before the increment, with the `dequeue ticket`’s value.
4) If they are the same, the thread is permitted to enter the critical section. If they are not the same, then another thread must already be in the critical section and this thread must `busy-wait` or `yield`.

II. **Priority Lock:**
1) A `priority lock` is a type of lock where priority is given to certain threads (in your case, those with smaller process IDs).
2) When a thread arrives, it is placed in the `queue` according to its priority.
3) The thread with the highest priority is allowed to enter the critical section first.
4) This can lead to `starvation` if higher priority threads keep arriving, as lower priority threads may never get the chance to acquire the lock.

[3] first-in, first-out