OS-CA4

Kasra Noorbakhsh - Kourosh Sajjadi - Mehdi Jamalkhah

810100230 - 810100587 - 810100111

Contents

1 XV6 synchronization

1.1 Disable(Q1)

We disable interrupts during critical sections to prevent preemption:

When a thread is executing a critical section protected by a lock, it's crucial to prevent preemption by the scheduler. If interrupts were not disabled, the scheduler could interrupt the currently executing thread and switch to another thread that might also attempt to enter the same critical section, leading to a race condition. By doing this we somewhat make our operation atomic but this comes with trade-offs, The most notable trade-off is that disabling interrupts can impact system responsiveness, especially in real-time systems where timely response to external events is crucial.

1.2 Pushcli and Popcli(Q2)

cli function disables the interrupts and sti function enables the interrupts again. In practice we have a integer valued variable that keeps the number of calls to the functions. pushcli and popcli are somehow activation functions that somewhat wrap's around for cli and sti. every time pushcli is called, it calls cli to disable interrupts and increments the value of that variable but, popcli calls sti to disable interrupts only if the variable is 0.

1.3 Single Core(Q3)

the code for acquire function :

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

   While(xchg(&lk->locked, 1) != 0);

   // references happen after the lock is acquired.
   __sync_synchronize();

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

First every interrupt is disabled by the pushcli function than in a while loop the atomic operation xchg (replaces a block in memory with a reg) the acquire function continuously makes the lk.locked 1 using xchg command this method causes busy waiting in multiprocessor systems this causes a heavy overhead and the efficiency of the system decreases in single-core systems the worst thing that can happen is a deadlock.

1.4 Amoswap(Q4)

The amoswap instruction in RISC-V stands for "atomic memory swap." It is used for implementing atomic read-write operations in multiprocessor systems.

The amoswap instruction provides a way to atomically swap the contents of a memory location with a register value. Its basic syntax is as follows:

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

rd is the destination register that will hold the original value in memory before the swap. rs1 is the source register containing the new value to be swapped into the memory location. rs2 is the base address of the memory location. (aq, rl) are optional acquire (aq) and release (rl) qualifiers that control the ordering of memory operations.

instruction execution by steps : Load the original value from the memory location specified by (rs2) into the destination register rd. Store the value from source register rs1 into the memory location specified by (rs2). The original value (loaded into rd) is now in the register, and the new value (from rs1) is stored in the memory location. The key aspect of amoswap is that these two operations (load and store) are performed atomically.

1.5 Communication(Q5)

In the acquire-sleep function the communication between process can be told like this one process request's the lock if the other processes had the lock the first process will be put to sleep until that lock is released that takes us to released-sleep function that does the same thing the process that is holding the lock releases it and that enables the processes that are sleep wake up (it might wake up one or more processes).

1.6 Conditions(Q6)

we have these conditions in XV6 that are located in procstate: enum procstate UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE ;

- **UNUSED**: This state represents a process that is not in use or has been terminated. It is essentially a free or not allocated process slot.
- **EMBRYO**: The EMBRYO state corresponds to a newly created process that is in the process of being initialized but is not yet fully ready to run.
- **SLEEPING**: When a process is waiting for an event or a certain amount of time to pass, it enters the SLEEPING state. This is typically the state when a process is waiting for I/O operations to complete.
- **RUNNABLE**: A process is in the RUNNABLE state when it is ready to execute but is waiting for the CPU to be scheduled. It has all the resources it needs to run, but the scheduler has not yet selected it for execution.
- **RUNNING**: The RUNNING state indicates that the process is currently being executed by the CPU. It is the active state where the CPU is executing the instructions of the process.
- **ZOMBIE**: After a process has completed its execution, it enters the ZOMBIE state. In this state, the process is not fully terminated, but it has finished its execution. The process stays in the ZOMBIE state until its parent retrieves its exit status.

The sched() function is for context switching to scheduler context. The scheduler iterates through the process table to find the next process that is in the RUNNABLE state and is eligible to run. If a RUNNABLE process is found, a context switch is performed to switch from the current running process to the newly selected process using swtch function.

1.7 Owner(Q7)

In the sleep lock structure we can simply use the pid variable that was in the original structure in the release-sleep we need to check for that pid the below code shows the modification needed:

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

    if (lk->pid != myproc()->pid) { //Check for PID
        release(&lk->lk);
        return;
    }

    lk->locked = 0;
    lk->pid = 0; //Reset owner's PID
```

```
    wakeup(lk);
    release(&lk->lk);
}
```

In the Linux kernel the equivalent is the mutex lock (its available at the mutex.h file).

1.8 Lock-Free(Q8)

Lock-free algorithms are a class of concurrent algorithms designed to enable multiple threads to make progress without the use of locks. In traditional multi-threading programming, using locks can introduce contention, synchronization overhead, and potential deadlocks.

Lock-free algorithms aim to overcome these limitations by providing a way for multiple threads to progress without waiting for a lock to be released. Instead of relying on locks, lock-free algorithms often use low-level atomic operations, such as compare-and-swap (CAS), to ensure that operations can proceed without interference from other threads. These algorithms are designed to guarantee that at least one thread makes progress within a finite number of steps, even in the presence of concurrent execution.

- **Pros**: Scalability: Lock-free algorithms can often achieve better scalability than lock-based approaches. Since threads are not blocked waiting for locks, they can continue to make progress even in the presence of contention.

  Avoidance of Deadlocks: Lock-free algorithms eliminate the possibility of deadlocks, which can occur in lock-based systems. With lock-free algorithms, there are no locks to hold or release, reducing the chance of encountering deadlocks.

- **Cons**: Increased Complexity: Lock-free algorithms are often more complex to design and implement than lock-based counterparts. Ensuring correctness in the presence of concurrent execution requires careful consideration of possible inter-leavings and edge cases.

  Limited Applicability: Lock-free algorithms may not be suitable for all scenarios. In some cases, the performance benefits of lock-free designs may be negligible or even detrimental. Additionally, some algorithms and data structures may not have straightforward lock-free equivalents, making it challenging to apply this approach uniformly across all parts of a system.

2 Cache

2.1 Hardware(Q9)

We learned in Computer-Architecture course that validating all invalided datum is a time-consuming procedure. we can use this approach : whenever a data is modified and thus invalid, do nothing! at each moment that a process needs the value of an variable, it checks and if it is invalid it must run somehow a protocol to update and validate the new value of that variable. so we just validate and update that data whenever it is needed.

2.2 Ticket(Q10)

Ticket lock is designed to address the fairness issue found in simpler locks like spin locks. The primary goal of a it is to ensure that threads or processes obtain access to the shared resource in a fair and orderly manner, preventing some threads from experiencing indefinite delays.

The ticket lock ensures that threads enter the critical section in the order of their ticket numbers, which guarantees fairness. However, this fairness comes at a cost, especially when dealing with validation or synchronization issues.

To address the slowness issue during local cache validation, you may want to consider the following:

- **Optimize Critical Section**: Analyze the code within the critical section to identify any bottlenecks or inefficient operations. Optimize the validation process itself to minimize the time threads spend in the critical section.

- **Caching Strategies**: Explore caching strategies that can reduce the need for frequent validation. Depending on the nature of the data and access patterns, you might be able to employ techniques such as lazy loading or caching with expiration policies.

2.3 Linux(Q11)

One commonly used method is through the use of OpenMP (Open Multi-Processing) directives, which is a set of compiler directives and library routines for parallel programming.

Here's a basic example of how you can use OpenMP to declare data placement for each core at compile time:

```
// Define a structure to hold your data
typedef struct {
    int value;
    // Add other data members as needed
} MyData;

// Specify the number of cores to use
int num_cores = omp_get_max_threads();

MyData data_per_core[num_cores];

#pragma omp parallel for
for (int i = 0; i < num_cores; ++i) {
    // Initialize data for each core
    data_per_core[i].value = i * 10;
}

#pragma omp parallel
{
    int thread_id = omp_get_thread_num();
    // Access and use other data members as needed
}
```

3 Implementations

3.1 Multi-Core

For this we added a global number of works and we added a number of works variable for each process (in the CPU file) and every-time someone calls a system call we have added a few lines of code that increments these numbers that we have added we also lock them so it doesn't induce a race-condition.

For the user-program we have added workload that first resets the numbers then it forks a number of children and attempts to write on a file which simulates workload on different cores and the protection of locks on it based on the fact that they are writing on a file.

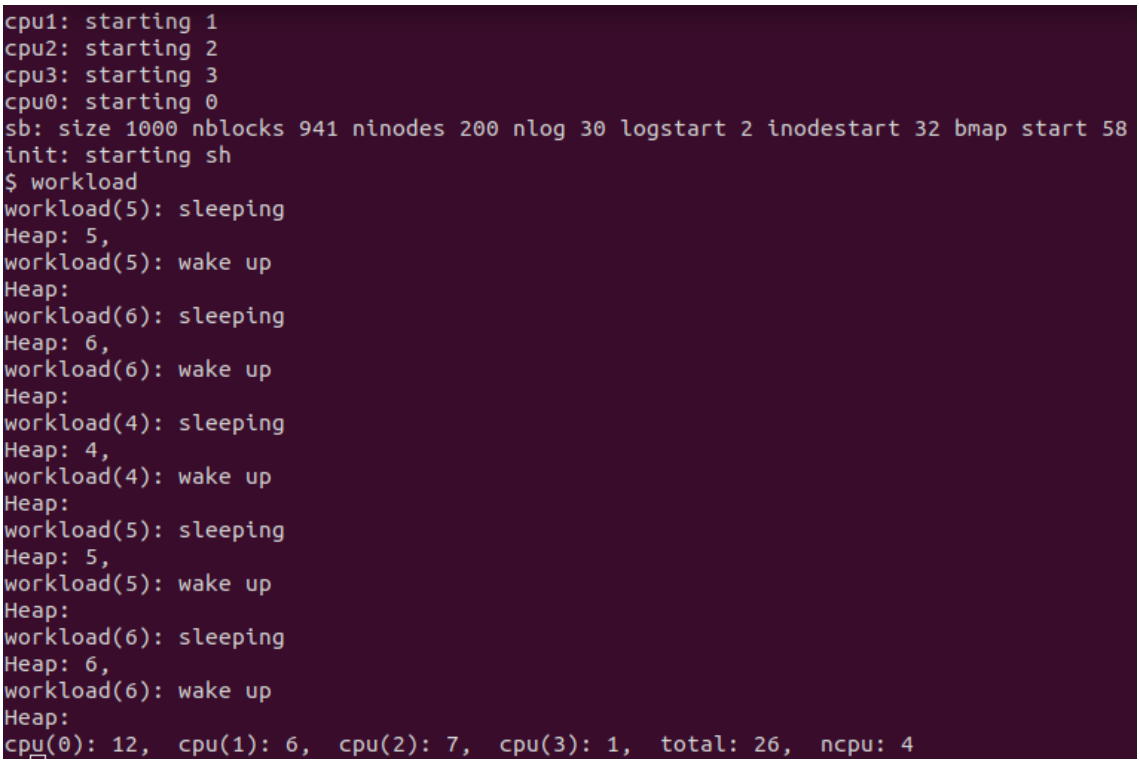

Figure 1: Multi-Core

3.2 Synchronization

For this we have added a heap data-structure and in the sleeplock file each time someone goes to sleep we add it to our heap then when someone gives a signal to wake up we pop the heap and that process (using its PID) wakes up.

For the user-program we have added race that which create a number of child's in which they simulate a virtual race such that some child goes to the critical section and sleeps (we used sleep so that it doesn't end instantly) and they print they're statuses.

This can lead to starving cause if some lock has a lower priority than others lets say 1 and other's constantly have a higher priority than that it can lead to starvation and a way to fix it is to use the same idea in aging after a certain amount of time we make its priority +1 or put it at the highest priority this also handles the cases where the locks have the same priority.

The difference between this and ticket lock is in the lock we implemented we use the PID as a priority selector but in ticket lock we assign a value then increment it this kinda acts like a FIFO.

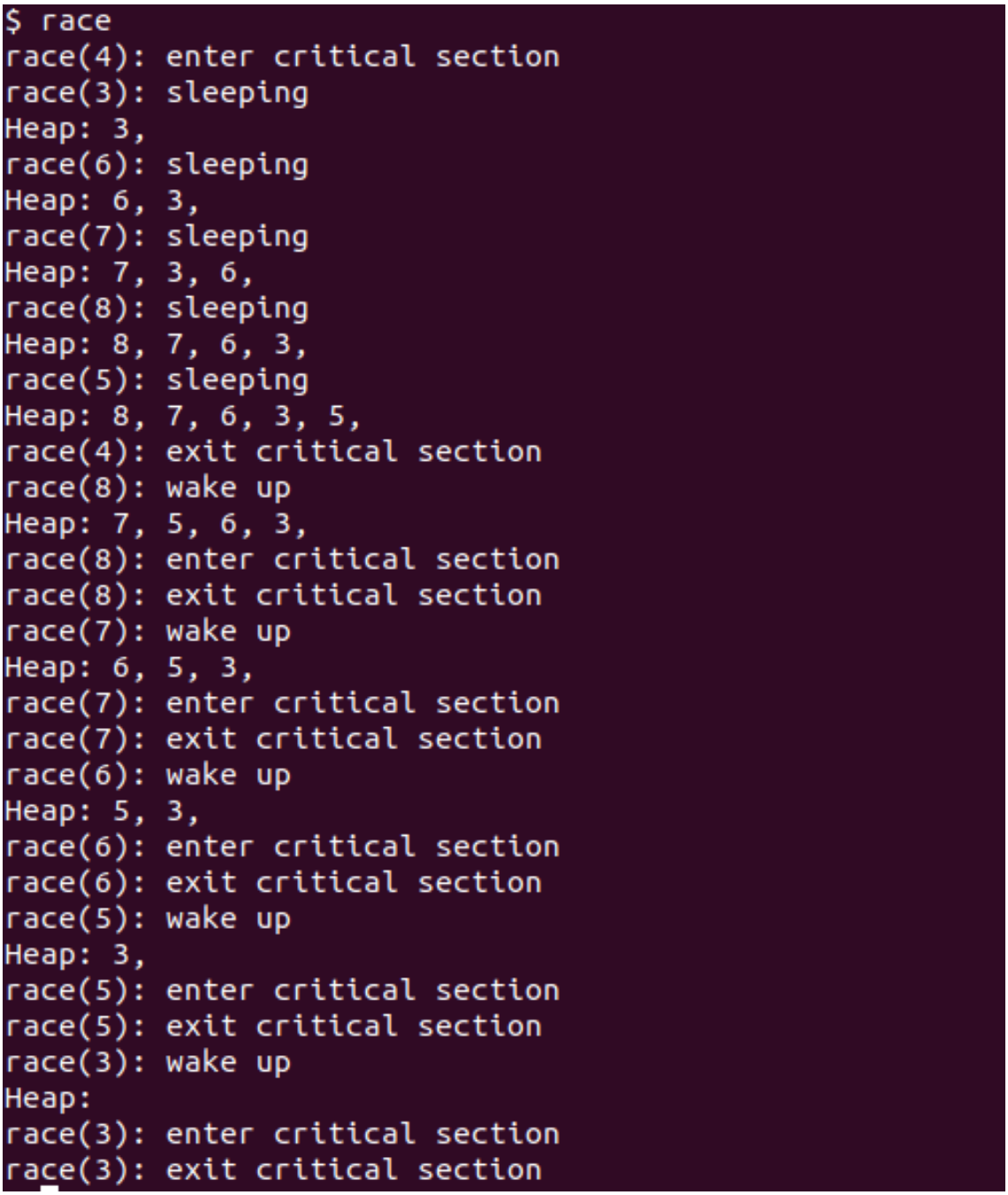

Figure 2: Race-User-Program