Lectures on Operating Systems (Mythili Vutukuru, IIT Bombay)

Practice Problems: Synchronization in xv6

1. Modern operating systems disable interrupts on specific cores when they need to turn off preemption, e.g., when holding a spin lock. For example, in xv6, interrupts can be disabled by a function call `cli()`, and reenabled with a function call `sti()`. However, functions that need to disable and enable interrupts do not directly call the `cli()` and `sti()` functions. Instead, the xv6 kernel disables interrupts (e.g., while acquiring a spin lock) by calling the function `pushcli()`. This function calls `cli()`, but also maintains of count of how many push calls have been made so far. Code that wishes to enable interrupts (e.g., when releasing a spin lock) calls `popcli()`. This function decrements the above push count, and enables interrupts using `sti()` only after the count has reached zero. That is, it would take two calls to `popcli()` to undo the effect of two `pushcli()` calls and restore interrupts. Provide one reason why modern operating systems use this method to disable/enable interrupts, instead of directly calling the `cli()` and `sti()` functions. In other words, explain what would go wrong if every call to `pushcli()` and `popcli()` in xv6 were to be replaced by calls to `cli()` and `sti()` respectively.

   Ans: If one acquires multiple spinlocks (say, while serving nested interrupts, or for some other reason), interrupts should be enabled only after locks have been released. Therefore, the push and pop operations capture how many times interrupts have been disabled, so that interrupts can be reenabled only after all such operations have been completed.

2. Consider an operating system where the list of process control blocks is stored as a linked list sorted by pid. The implementation of the wakeup function (to wake up a process waiting on a condition) looks over the list of processes in order (starting from the lowest pid), and wakes up the first process that it finds to be waiting on the condition. Does this method of waking up a sleeping process guarantee bounded wait time for every sleeping process? If yes, explain why. If not, describe how you would modify the implementation of the wakeup function to guarantee bounded wait.

   Ans: No, this design can have starvation. To fix it, keep a pointer to where the wakeup function stopped last time, and continue from there on the next call to wakeup.

3. Consider an operating system that does not provide the `wait` system call for parent processes to reap dead children. In such an operating system, describe one possible way in which the memory allocated to a terminated process can be reclaimed correctly. That is, identify one possible place in the kernel where you would put the code to reclaim the memory.

   Ans: One possible place is the scheduler code itself: while going over the list of processes, it can identify and clean up zombies. Note that the cleanup cannot happen in the exit code itself, as the process memory must be around till it invokes the scheduler.

4. Consider a process that invokes the `sleep` function in xv6. The process calling sleep provides a lock `lk` as an argument, which is the lock used by the process to protect the atomicity of its call to sleep. Any process that wishes to call `wakeup` will also acquire this lock `lk`, thus avoiding a call to wakeup executing concurrently with the call to sleep. Assume that this lock `lk` is not `ptable.lock`. Now, if you recall the implementation of the sleep function, the lock `lk` is released before the process invokes the scheduler to relinquish the CPU. Given this fact, explain what prevents another process from running the wakeup function, while the first process is still executing sleep, after it has given up the lock `lk` but before its call to the scheduler, thus breaking the atomicity of the sleep operation. In other words, explain why this design of xv6 that releases `lk` before giving up the CPU is still correct.

   Ans: Sleep continues to hold ptable.lock even after releasing the lock it was given. And wakeup requires ptable.lock. Therefore, wakeup cannot execute concurrently with sleep.

5. Consider the `yield` function in xv6, that is called by the process that wishes to give up the CPU after a timer interrupt. The yield function first locks the global lock protecting the process table (`ptable.lock`), before marking itself as RUNNABLE and invoking the scheduler. Describe what would go wrong if `yield` locked `ptable.lock` AFTER setting its state to RUNNABLE, but before giving up the CPU.

   Ans: If marked runnable, another CPU could find this process runnable and start executing it. One process cannot run on two cores in parallel.

6. Provide one reason why a newly created process in xv6, running for the first time, starts its execution in the function `forkret`, and not in the function `trapret`, given that the function `forkret` almost immediately returns to `trapret`. In other words, explain the most important thing a newly created process must do before it pops the trap frame and executes the return from the trap in `trapret`.

   Ans: It releases ptable.lock and preserves the atomicity of the context switch.

7. Consider a process P in xv6 that acquires a spinlock L, and then calls the function `sleep`, providing the lock L as an argument to `sleep`. Under which condition(s) will lock L be released *before* P gives up the CPU and blocks?

   (a) Only if L is `ptable.lock`

   (b) Only if L is not `ptable.lock`

   (c) Never

   (d) Always

   Ans: (b)

8. Consider a system running xv6. You are told that a process in kernel mode acquires a spinlock (it can be any of the locks in the kernel code, you are not told which one). While the process holds this spin lock, is it correct OS design for it to:

   (a) process interrupts on the core in which it is running?

   (b) call the `sched` function to give up the CPU?

For each question above, you must first answer Yes or No. If your answer is yes, you must give an example from the code (specify the name of the lock, and any other information about the scenario) where such an event occurs. If your answer is no, explain why such an event cannot occur.

Ans:

(a) No, it cannot. The interrupt may also require the same spinlock, leading to a deadlock.

(b) Yes it is possible. Processes giving up CPU call `sched` with `ptable.lock` held.

9. Consider the following snippet of code from the `sleep` function of xv6. Here, `lk` is the lock given to the sleep function as an argument.

```
if(lk != &ptable.lock){
  acquire(&ptable.lock);
  release(lk);
}
```

For each of the snippets of code shown below, explain what would happen if the original code shown above were to be replaced by the code below. Does this break the functionality of `sleep`? If yes, explain what would go wrong. If not, explain why not.

(a)
```
acquire(&ptable.lock);
release(lk);
```

(b)
```
release(lk);
acquire(&ptable.lock);
```

Ans:

(a) This code will deadlock if the lock given to sleep is `ptable.lock` itself.

(b) A wakeup may run between the release and acquire steps, leading to a missed wakeup.

10. In xv6, when a process calls `sleep` to block on a disk read, suggest what could be used as a suitable channel argument to the `sleep` function (and subsequently by `wakeup`), in order for the sleep and wakeup to happen correctly.

Ans: Address of struct buf (can be block number also?)

11. In xv6, when a process calls `wait` to block for a dead child, suggest what could be used as a suitable channel argument in the `sleep` function (and subsequently by `wakeup`), in order for the sleep and wakeup to happen correctly.

Ans: Address of parent struct proc (can be PID of parent?)

12. Consider the exit system call in xv6. The exit function acquires ptable.lock before giving up the CPU (in the function sched) for one last time. Who releases this lock subsequently?

Ans: The process that runs immediately afterwards (or scheduler)

13. In xv6, when a process calls wakeup on a channel to wakeup another process, does this lead to an immediate context switch of the process that called wakeup (immediately after the wakeup instruction)? (Yes/No)

    Ans: No

14. When a process terminates in xv6, when is the struct proc entry of the process marked as unused/free?

    (a) During the execution of exit

    (b) During the sched function that performs the context switch

    (c) In the scheduler, when it iterates over the array of all struct proc

    (d) During the execution of wait by the parent

    Ans: (d)

15. In which of the following xv6 system call implementations is there a possibility of the process calling `sched()` to give up the CPU? Assume no timer interrupts occur during the system call execution. Tick all that apply: `fork` / `exit` / `wait` / none of the above

    Ans: exit, wait

16. In which of the following xv6 system call implementations will a process *always* invoke `sched()` to give up the CPU? Assume no timer interrupts occur during the system call execution. Tick all that apply: `fork` / `exit` / `wait` / none of the above

    Ans: exit

17. Under which conditions does the `wait()` system call in xv6 block? Tick all that apply: when the process has [no children / only one running child / only one zombie child / two children, one running and one a zombie]

    Ans: Only one running child

18. Consider a system with two CPU cores (C0 and C1) that is running the xv6 OS. A process P0 running on core C0 is in kernel mode, and has acquired a kernel spinlock. Another process P1 on core C1 is also in kernel mode, and is busily spinning for the same spinlock that is held by P0. Which of the following best describes the set of cores on which interrupts are disabled?

    (A) C0 and C1 (B) Only C0 (C) Only C1 (D) Neither C0 nor C1

    Ans: A, interrupts need to be disabled before starting to spin also.

19. Consider a process in xv6 that invokes the wakeup function in kernel mode. Which of the following statements is/are true?

    (a) The wakeup function immediately causes a context switch to one of the processes sleeping on a particular channel value.

    (b) The wakeup function wakes up (marks as ready) all processes sleeping on a particular channel value.

    (c) The wakeup function wakes up (marks as ready) only the first process that is blocked on a particular channel value.

(d) The wakeup function wakes up (marks as ready) only the last process that is blocked on a particular channel value.

Ans: (b)

20. Consider a process in kernel mode in xv6, which invokes the sleep function to block. One of the arguments to the sleep function is a kernel spinlock. Which of the following statements is/are true?

    (a) If the lock given to sleep is ptable.lock, then the lock is not released before the context switch.

    (b) If the lock given to sleep is not ptable.lock, then the lock is not released before the context switch.

    (c) If the lock given to sleep is not ptable.lock, then the lock is released before the context switch, but only after acquiring ptable.lock.

    (d) If the lock given to sleep is not ptable.lock, then the lock is released before the context switch and ptable.lock need not be acquired.

    Ans: (a), (c)