

CS 333 Operating Systems Lab

Autumn 2023

Lab6: the space-time continuum

The scope of this lab is to understand the CPU scheduler of xv6 and to try out new extensions to scheduling in xv6. In this lab, we will modify the xv6 scheduler and work with context switches, process statistics, priorities and time quanta.

Task 0: Overview

- Setting up xv6 - Set up xv6 the same as previous labs (see included xv6 setup guide file in the assignment folder)
- **In the Makefile, replace `CPUS := 2` with `CPUS := 1`**
The old setting runs xv6 with two CPU cores, to make the scheduling more deterministic the #CPUs of the emulated machine are set to 1.
- The booklet describing/listing all source files is available [here](#).
(also after `make xv6.pdf` in the xv6 dir).
- Chapter 5 (Page 61) of the [xv6 book](#) contains conceptual details about scheduling and multiplexing in xv6.
- The current xv6 scheduler loops through all the processes which are available to run (in the **RUNNABLE** state) and allocates the CPU to each one of them (schedules processes) one at a time — round-robin-like scheduler.
- **proc.c** contains most of the logic for the scheduler in the xv6 and the associated header file, **proc.h** is also quite useful for examining the functioning of the scheduler in xv6.
The scheduler function is called by the **mpmain** function in **main.c** as the last step of initialization. This function will never return. It loops forever to schedule the next available process for execution. If you are curious about how it works, read Chapter 5 of the xv6 book.

Information about the system calls and files in xv6 has been included under the heading “System calls related” in the included “xv6 setup guide” document.

Task 1: context matters

(a) count switch-in and switch-out in xv6

xv6 uses a per-CPU process scheduler. Each CPU calls a **void scheduler(void)** after setting itself up. The scheduler keeps looping infinitely, doing the following:

- choose a process to run
- **switch** to start running that process
- Eventually, that process transfers control via **switch** back to the scheduler.

Refer to sheet 27 to understand the existing scheduler function (present in `proc.c`) in xv6 and get familiar with it.

The functioning of the `switch` system call is as follows:

void switch(struct context **old, struct context *new);

Save the current registers on the stack (populating) struct context, and save its address in *old (this is the old context).

Switch stacks to newcontext (new→esp) and pop previously saved registers.

Refer to implementation in **switch.S**

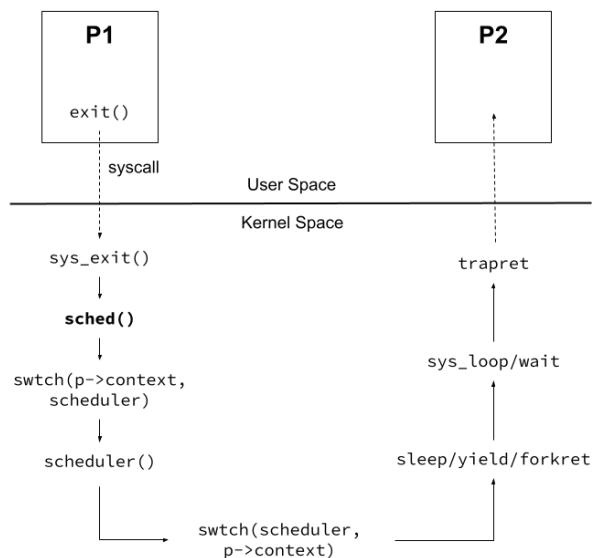


Figure 1: Depiction of system call flow in process scheduling

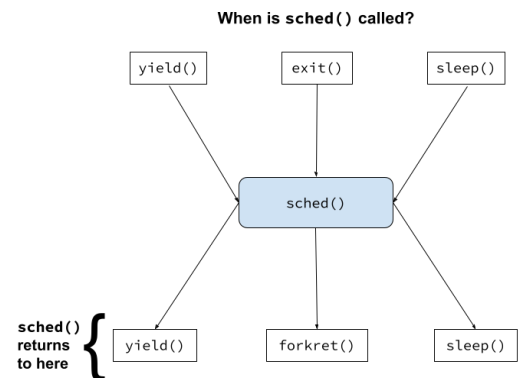


Figure 2: Locations in xv6 code where sched() is invoked and where sched() returns to.

Refer to `switch.S` in the xv6 source code to familiarize yourself with the underlying assembly code for saving and loading registers while context switching.

Figure 1 describes the sequence of **scheduler**, **sched** and **switch** and Figure 2 shows the location in xv6 code where it is called from where it returns to when rescheduled.

Implement a system called **cscount(int pid)** to count the number of context switches for a process with a given pid, i.e. you need to count both the number of switch-in and switch-out of that particular process.

Hint: Can add fields in proc struct to store the per process context switch in and out values. Understand the logic of **sched** in proc.c and when it is called. Note that the **scheduler** picks up a user space process to run on the CPU whereas **sched** switches back to the scheduler again once the process is done. **sched** is called after a timer interrupt when a process becomes a zombie or when a process goes to a blocked state.

NOTE: The user program provided `cscount.c` has a simple implementation to check the `cscount()` system call.

The given user program forks child processes and calls the `cscount()` just before the child starts executing, and calls the system call `cscount()` again just before the child exits.

The output format is pid [**process pid**] switch in [**context switch in count**], switch out [**context switch out count**].

Your output should give a switch in count = 1 and switch out count = 0 just before the process starts executing since this is the first time the process is getting the share of the CPU. The switch in and out count at the end can be any number depending on the time taken by the process to execute and the number of context switches it went through.

Sample usage

```
$ cscount
-----Testcase: context switch-----Scheduler: DEFAULT-----

pid [4] switch in [1],switch out [0]
Child 4 started

pid [5] switch in [1],switch out [0]
Child 5 started

pid [6] switch in [1],switch out [0]
Child 6 started
Child 5 finished

pid [5] switch in [1621],switch out [1620]
Child pid: 5 exited
Child 6 finished

pid [6] switch in [1622],switch out [1621]
Child pid: 6 exited
Child 4 finished

pid [4] switch in [1628],switch out [1627]
Child pid: 4 exited
$
```

(b) waiting for execution statistics

Implement a new system call `wait2` similar to `wait`, but with more functionalities, in order to check the performances of xv6 scheduling algorithms. Specifically, it will have the following interface:

```
int wait2(int *wtime, int *runtime);
```

The call takes two arguments, pointers to variables that denote the amount of time the process spent waiting for the CPU and the time spent executing on the CPU. It waits for a child process to exit and fills the waiting time and run time (both are in terms of ticks counted in the trap handler. Note that xv6 is configured to generate 200 ticks per second for a 2GHz CPU, which corresponds to a tick every 5ms. You may however report just the tick count.) in `wtime` and `runtime` buffer, respectively, for the process that is exiting and return its `pid`. Return `-1` if the calling process has no children.

The waiting time of a process is defined as the time spent by the process in the RUNNABLE state (ready to run and waiting for CPU), and run time is the time spent by the process on the CPU (time is in ticks).

NOTE: The user programs provided `task1b.c` have a simple implementation to check the `wait2()` system call.

The `wait2` call will be useful for understanding how a scheduling policy affects the times of every Process.

Hints:

- Logging of durations/timings will need tracking down all events where the state of the process changes between WAITING, RUNNABLE, RUNNING etc. and appropriate duration updates via variables in the PCB entry for the process.
- An implementation of the `wait` system call exists in xv6 and can be the starting point for the `wait2` implementation — a copy of the code of `wait` is a good starter for the implementation of `wait2`.
- You can initialize `wtime` and `runtime` in the `allocproc` function in the `proc.c` file.

Note: It is important to keep in mind that the process table struct `ptable` is protected by a lock. You must acquire the lock before accessing this structure for reading or writing and must release the lock after you are done.

Sample usage

```
$ task1b
-----Testcase: wait2-----Scheduler: DEFAULT-----
*Case1: Parent has no children*
wait2 status: -1
*Case2: Parent has children*
Child 7 created
Child 8 created
Child 8 finished
Child 7 finished
Child pid: 8 exited with pid: 8, Waiting Time: 58, Run Time: 56
Child pid: 7 exited with pid: 7, Waiting Time: 58, Run Time: 58
$
```

Task 2: time turner

(a) *actions express priorities ...* or is it the other way round?

A priority-based scheduler selects the process with the highest priority for execution. In case two or more processes have the same priority, we choose them in a round-robin fashion. The priority of a process can be in the range [0,100]. The **smaller value will represent a higher priority**. Set the default priority of a process as 60. To change the default priority add a new system call `set_priority` which can change the priority of a process.

int set_priority (int pid, int priority)

The system call should set the priority of a process with a given pid and return the pid of the process. The scheduler should then schedule the process based on the set priority.

Hint: You also need to slightly modify the `scheduler()` function in `proc.c` to change the logic to choose the process having higher priority to schedule instead of the one used by the default round robin in xv6.

The user programs provided `task2a.c` have a simple implementation to check the `set_priority()` system call.

Sample usage

Note: If multiple processes have the same priority, they will be executed in the default round-robin fashion (first sample output) and if processes have different priorities (second sample output), the one with a lower priority number (i.e. higher priority) will execute before the ones with a higher priority number (i.e. lower priority).

This effect is visible in the increased waiting time of lower priority processes.

```
$ task2a1
-----Testcase 1: set priority-----Scheduler: PRIORITY BASED-----
Child 8 created priority 4
Child 9 created priority 4
Child 10 created priority 4
Child 9 finished
Child 8 finished
Child 10 finished
Child pid: 8 exited with pid: 8, Waiting Time: 111, Run Time: 56
Child pid: 9 exited with pid: 9, Waiting Time: 110, Run Time: 54
Child pid: 10 exited with pid: 10, Waiting Time: 110, Run Time: 56
$ task2a2

$ task2a2
-----Testcase 2: set priority-----Scheduler: PRIORITY BASED-----
Child 12 created priority 5
Child 12 finished
Child 13 created priority 6
Child 13 finished
Child 14 created priority 7
Child 14 finished
Child pid: 12 exited with pid: 12, Waiting Time: 0, Run Time: 58
Child pid: 13 exited with pid: 13, Waiting Time: 58, Run Time: 56
Child pid: 14 exited with pid: 14, Waiting Time: 114, Run Time: 60
$
```

(b) the big slice

Implement the following system call **int set_quanta(int pid, int quanta)**, which sets the time slice quanta of a process with the given pid. This system call will allow you to give more time to a process overriding the **default slice** of 1 quanta/tick implemented in xv6.

You need to add a specific field to the process structure (e.g. current slice, extra_slice) at `proc.h` to hold the current time slice value. You should create the syscall to set quanta and modify the scheduler function to reset `current_slice` to `extra_slice` at process wakeup.

You also need to modify `trap.c` to handle the time slice logic when timer interrupts come for a process to give up the CPU on the clock tick. Look at the case of what happens when **T_IRQ0 + IRQ_TIMER** interrupt occurs.

Note: Change the modification in `scheduler()` in `proc.c` made for the previous task back to the default scheduling logic and add a line to update the process's current slice value.

The user programs provided `task2b.c` have a simple implementation to check the `set_quanta()` system call.

Note: The process with a higher quanta should have a lesser waiting time than others since it gets more time (opportunity) to execute at a time (and T.Q. in output implies "Time Quanta")

Sample usage:

The process with a higher time quanta (`pid = 4`) finishes before every one as it has the largest time quanta (T.Q. = 8) and has a waiting time 0 since it utilizes its full quanta to complete its execution. Similar is the case process with `pid = 5` with respect to the process with `pid = 6`. ***So you need to make sure that the process with a higher time quanta has a lesser waiting time than those with lesser time quanta.***

```
$ task2b
-----Testcase: set quanta-----Scheduler: DEFAULT-----
Child [4] created T.Q. [8]
Child [4] finished
Child [5] created T.Q. [4]
Child [5] finished
Child [6] created T.Q. [2]
Child [6] finished
Child pid: 4 exited with pid: 4, Waiting Time: 0, Run Time: 34
Child pid: 5 exited with pid: 5, Waiting Time: 34, Run Time: 34
Child pid: 6 exited with pid: 6, Waiting Time: 68, Run Time: 34
$ task2b
```

Submission instructions

- All submissions have to be done via Moodle. Name your submission as `<rollnumber_lab6>.tar.gz`
(e.g., 2100500ab_lab6.tar.gz)
- The tar should contain the xv6 source code files in the following directory structure: **<rollnumber_lab6>/xv6**
- The submission must contain the **entire xv6 source code** with necessary changes to it.
- Your code should be well-commented and readable.
- `tar -czvf <rollnumber_lab6>.tar.gz <rollnumber_lab6>`

Deadline:

Wednesday 4th October 2023 4:59 PM via Moodle. (no extensions)
