CSU0033 Operating Systems, Homework 4

Department of Computer Science and Information Engineering
National Taiwan Normal University

May 1, 2025

As always, you are encouraged to ask and help answer questions on Moodle.

It becomes important to use AI as a good tool. But AI shall not harm you by reducing your critical thinking and creativity, which are sharpen by working on the problem by yourself first. You must rule over the temptation of simply flying over with AI. If you used AI to help you in answering any of the questions, tell us in which specific ways you used it. Knowing that you used AI will not affect our grading. It is just that we would like to know how you used it.

Contents

1 More on spinlocks (40 points) **1**

2 Learning by teaching (15 points) **2**

3 Multi-thread programming (25 points) **3**

4 Xv6 kernel programming (20 points) **5**

1 More on spinlocks (40 points)

In this part of the homework, we will get some deeper understanding for spinlocks. Before you proceed, study Chapter 6 of the xv6 book. The chapter explains some more details in implementing a spinlock. This, along with the corresponding sections in the Comet book[1], should provide enough background for you to tackle the following questions.

Now, answer the following questions in your own words.

[1]Links to each book can be found in the slides we used on Week 1.

Question-1 (15 points) Review both the idea and usage of *hardware interrupts*. For example, read Chapters 6, 7 and 36 in the Comet book. When a thread of execution is experiencing a hardware interrupt (e.g., a hardware timer), we say the system will trap into the kernel and move into the *interrupt context*, where an ISR (interrupt service routine) will handle the event that caused the interrupt. Afterwards, the execution would resume to the interrupted thread. With this in mind, try to explain why is it that 'while the system is still in the interrupt context, the scheduler will not pick and resume the interrupted thread'. (Hint: recall that an event causing an interrupt might or might not relate to the interrupted thread; try to use some scenarios to illustrate your points)

Question-2 (15 points) Study the xv6's implementation of spinlock in *kernel/spinlock.c*. After you've studied Chapter 6 of the xv6 book, the implementation of *acquire()* (i.e., lock) and *release()* (i.e., unlock) therein will be easier to understand. Now, give one reason why this system implementation could eventually deadlock if 'one thread calls *release()* to unlock the lock currently held by another thread'[2], assuming the thread whose lock is stolen will as usual call *release()* to fufill the need of using *acquire()/release()* to guard a critical section. Give an example to make your argument clear.

Question-3 (10 points) Explain why in a multi-core system it may be advantageous to use a spinlock instead of a sleep-lock, provided that the critical section it protects is tiny in the sense of execution time.

2 Learning by teaching (15 points)

Often, we learn more when we try to teach. Find one of your friends (or a family member) who is not majoring in engineering. Explain to him or her the following three ideas in the context of the producer/consumer problem with an one-unit buffer:

1. Why do we need to synchronize producers and consumers?
2. Why would the use of semaphore help synchronize the producers and the consumers?
3. How could an incorrect use of semaphores cause deadlock?

Record one question your audience asked (instead of 'why do I need to know this?') and also record how you answered that question.

[2]Note that this does not mean xv6 has an erroneous implementation of spinlock; the intention of this exercise is to use xv6's concrete implementation to demonstrate an erroneous use of spinlock, to let us know why such a usage is erroneous.

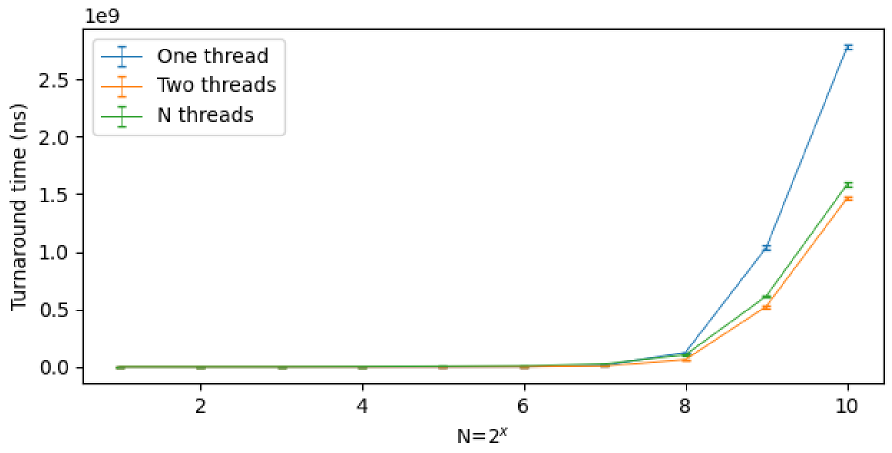
3 Multi-thread programming (25 points)

Do the following in your Linux environment (not xv6). We will use the pthread library to implement a multi-thread program for matrix multiplication.

First, read Chapter 27 in the Comet book and run some example code to get yourself familiar with the pthread library. Then, write a pthread program to compute the multiplication of two n-by-n matrices. In your program, do three versions of the multiplication. In the first version, just use one single thread (i.e., do not create new threads) and do sequential computation for each entry in the matrix of the result. In the second version, use two threads to parallelize the computation. In the third version, use n threads to parallelize the computation. Remember to verify that all the three versions will produce the same results.

The requirements:

1. Randomly generate the entries in the two input matrices, with each entry being an integer between 0 and 9.
2. Use Linux's *clock_gettime()* to take time-stamps both at the beginning and at the end of each version of matrix multiplication. Take the time difference and print the turnaround time in terms of nanoseconds. Use the `CLOCK_MONOTONIC_RAW` clock; type `man clock_gettime` to learn more.
3. For each version of matrix multiplication, compute the result for n=*2, 4, 8, 16, 32, 64, 128, 256, 512,* and *1024*.
4. Repeat the experiment 50 times for each case of n the matrix dimension. Each time creates a random matrix and use the same matrix for all the three versions of implementation. Compute both the average turnaround time for each n and *the 95% confidence interval* for each n. The 95% confidence interval describes the range of value that the result of the next experiment has 95% probability to fall into. See the following example Python script for a way to compute and plot this interval.
5. Generate a figure to visualize the overall experiment results. The figure must have three line charts, representing the result of single-thread, two-thread, and n-thread cases. In the figure, the x-axis is the different value of n, and the y-axis is the turnaround time in the linear scale. See below for an example figure.



To plot the results, you may use whichever data visualization tools you like. The following is the Python script I used for generating the above figures.

```
import matplotlib.pyplot as plt
import numpy as np

d = []
v1avg = []
v2avg = []
vnavg = []
v1err = []
v2err = []
vnerr = []
x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
for i in range(1,11):
    p = 2**i
    d = np.loadtxt("./" + str(p) + ".out", delimiter=" ")
    v1avg.append(np.mean(d[:,0]))
    v2avg.append(np.mean(d[:,1]))
    vnavg.append(np.mean(d[:,2]))
    # Calculate the 95% confidence interval
    v1err.append(1.96*np.std(d[:,0],ddof=1)/np.sqrt(len(d[:,0])))
    v2err.append(1.96*np.std(d[:,1],ddof=1)/np.sqrt(len(d[:,1])))
    vnerr.append(1.96*np.std(d[:,2],ddof=1)/np.sqrt(len(d[:,2])))

plt.errorbar(x, v1avg, yerr=v1err, lw=0.7,\
 elinewidth=0.7,capsize=2, markersize=5, label='One thread')
plt.errorbar(x, v2avg, yerr=v2err, lw=0.7,\
 elinewidth=0.7,capsize=2, markersize=5, label='Two threads')
plt.errorbar(x, vnavg, yerr=vnerr, lw=0.7,\
 elinewidth=0.7,capsize=2, markersize=5, label='N threads')
```

```
plt.legend(loc='best')
plt.xlabel('N=$2^x$')
plt.ylabel('Turnaround time (ns)')
plt.tight_layout()
plt.show()
```

The above script requires both the *matplotlib* library and the *numpy* library.

Submit your code, your figure, and your analysis. Analyze your result based on the setting of your system; for example, how many CPUs in your Linux, and how may that impact the performance of your code?

4 Xv6 kernel programming (20 points)

First, boot up your Linux environment and try to run some commands in the terminal. Then press the `up-arrow` key and the `down-arrow` key several times, to see how your Linux shell provides a history of commands for you. In this part of the assignment, we are going to implement a similar feature for our xv6.

In your xv6 environment (not Linux), do the following.

Take a look at *kernel/console.c* to see how xv6 handles our keyboard input. To implement command-line history, it is sufficient to only modify that file.

Under the hood, the QEMU emulates a UART[3] device and our xv6 is attached to it (or the device is attached to our xv6, if you'd like). Each of our key press in effect will trigger a command sent to that UART device, which then will encode the pressed key by one or a series of ASCII characters. Then the device will trigger a hardware interrupt to the xv6 OS. The kernel then quickly handles each such interrupt. That is why you can just press `Ctrl`+`P`, with no press of the `Enter` key, and the OS will still respond. See *kernel/console.c* for detail.

Your implementation of the command-line history support must meet the following requirements:

1. Press `Ctrl`+`W` and the shell will show you the most recent command entered. Then you can press `Ctrl`+`W` again to get an older command, like the `up-arrow` key in Linux. Press `Ctrl`+`S` one or multiple times to get a more recent command, like the `down-arrow` key in Linux. Press the `Enter` key and the shell will run the shown command for you.
2. The system keeps at most *five* most-recent commands. In the beginning, the history is empty.
3. Should we move beyond the available history, the system will beep and will either show the oldest command (too many `Ctrl`+`W`) or show nothing

[3]https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter

(too many `Ctrl+S`). The sound will work on a remote terminal (e.g., via *ssh* from Windows PowerShell) as long as the terminal supports audio.

4. Each `Ctrl+W` or `Ctrl+S` will clear the current command shown on the prompt; that is, your intermediate input will be lost.

The following is the test data we will use to validate your implementation:

```
$ echo 1111111111111111111111111111111
1111111111111111111111111111111
$ echo 2222222222222222222222222222222
2222222222222222222222222222222
$ echo 3333333333333333333333333333333
3333333333333333333333333333333
$ echo 4444444444444444444444444444444
4444444444444444444444444444444
$ echo 5555555555555555555555555555555
5555555555555555555555555555555
$ echo last
last
$
```

After the above six input commands, your code shall be able to give the five most-recent commands entered, according to the requirements.