CS 342302 Operating Systems

Fall Semester 2021

Prof. Pai H. Chou

Weekly Review 5

The questions here serve the purpose of reviewing concepts from the lecture, and expect the concepts to be tested on the midterm and final. However, they are by no means exhaustive. Anything covered in the lecture and projects can be tested.

1. Definitions and Short Answers - week 5

1. What is a **thread**? What comprises the state of a thread, and what does a thread share with other threads in the same process?
2. For a server, why is it not a good idea to create a separate process to service each request?
3. What is the difference between **concurrent** and **parallel** execution? Can you have concurrency without parallelism? Can you have parallelism without concurrency?
4. On a uniprocessor, do threads run concurrently? in parallel? What about on a multicore processor? Under what conditions do threads run in parallel or concurrently but not in parallel?
5. What is **data parallelism**? Give an example and explain why.
6. What is **task parallelism**? Give an example and explain why.
7. What is the difference between **user threads** and **kernel threads**? Do both have to make system calls to create and join threads?
8. Do user threads need OS support?
9. Is it true that **kernel threads** always run in kernel mode?
10. What is the difference between **kernel threads** and **hardware threads**?
11. What are advantages with user threads over kernel threads? What is a disadvantage on a single-threaded kernel?
12. What are the three main ways of **mapping** user threads to kernel threads?
	1. if the kernel is not multithreaded, which model can it support? Can it support one-to-one and why or why not?
	2. Can many-to-one support simultaneous execution of user threads and a blocking system call made by one of its threads? Why or why not?
	3. Do kernel threads run in parallel? Can a uniprocessor run an OS with multiple kernel threads? If so, what would be a benefit and for which model?
13. How is **two-level** thread model different from a many-to-many model?
14. Are threads created using fork() like regular processes?
15. Are most threads packages (Pthreads, Python threads, Java threads) **preemptive** or **cooperative**?
16. What is a problem that can happen with cooperative threads but not with preemptive threads?
17. To implement preemptive threads at user level (on a Unix-like system), what is the mechanism for the user code to get the effect similar to a timer-interrupt for context switching? Why can't you use the timer interrupt directly as processes do?
18. What is done by **synchronous threading** that is usually not done by **asynchronous** threading?
19. Does the "P" in Pthreads mean "Python" threads?
20. What does it mean to **spawn** a thread?
21. What is the purpose of calling pthreads\_join() on a created thread?
22. Does pthread\_create() make a clone of the current thread, similar to fork() cloning the current process?
23. Why would you call pthread\_detach() on a thread instead of calling pthread\_join() on it?
24. In a Python generator, if you have a statement of c = **yield**, where does the yield expression get its value from?
25. How do you write the producer-consumer example in Python using *generators* in a **push** style vs. in **pull** style?
26. What are two ways a thread can **exit** **voluntarily**?
27. What are the main reasons to use a **thread pool** as opposed to directly creating threads?
28. In OpenMP, how are threads created?
29. Does OpenMP support the semantics of synchronous or asynchronous threads? What type of parallelism is it called?
30. Do all threads of a process get duplicated by fork(), or what would fork() do?
31. If you use signal() to register a callback, which thread or threads would get the signal?
32. when pthread\_cancel() is called on a thread *t*, is *t* terminated immediately or in what way?
33. What is **thread-local storage** and how is it different from **local variables**?
34. How are **scheduler activations** used with the user-thread scheduler?
35. How does Linux clone() create processes and threads?

2. Thread Exercises

QuickSort is a popular algorithm for sorting. Even though its worst-case runtime complexity is *O*(*n*2), its average complexity is *O*(*n* lg *n*), and in practice it is very fast because it is in-place sorting (i.e., does not need a temporary buffer). Also, as a divide-and-conquer algorithm, it does most of the work in the “divide” stage and no work in the “conquer” stage. This makes it nice for parallelizing, because after forking, there is no dependency after joining.

The following is a Python 3 implementation of Quicksort.

| **def** Partition(A, p, r): x = A[r] i = p - 1 **for** j **in** range(p, r): **if** (A[j] <= x): i = i + 1 A[i], A[j] = A[j], A[i] A[i+1], A[r] = A[r], A[i+1] **return** i + 1 | **def** QuickSort(A, p, r): **if** p < r: q = Partition(A, p, r) QuickSort(A, p, q-1) QuickSort(A, q+1, r) |
| --- | --- |
| **if** \_\_name\_\_ == '\_\_main\_\_': LEN = 10 L = list(range(LEN)) **import** random random.shuffle(L) QuickSort(L, 0, len(L)-1) **if** L == list(range(LEN)): print("successfully sorted") **else**: print("sort failed: %s" % L) |

The test case just says to generate numbers 0..LEN-1, shuffle, and sort. If successful, it says so; otherwise, it dumps the list.

2.1 Convert quicksort.py to C and call it quicksort.c.

1. Since you will need to measure the execution time of the code, you will need a large enough dataset. However, shuffling numbers can take a long time. Instead of shuffling numbers, have the numbers be pre-generated by the following script (genrandom.py) just once before you run your own code any number of times.
**import** random
LEN = 20000000
L = list(range(LEN))
random.shuffle(L)
fh = open("randomInt.txt", "w")
# first write is the length
fh.write(str(LEN)+'\n')
**for** i **in** L:
 fh.write(str(i)+'\n')
fh.close()
Run this python program and it will create a text file named randomInt.txt. The first line is the number for LEN, followed by the shuffled numbers in the range 0..LEN-1.
2. You can use the following template (quicksort.c) to read in the data into array A, or feel free to write your own code:
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
**int** \*A; // array
**int** Partition(**int** A[], **int** p, **int** r) {
 // your code
}

**void**\* QuickSort(**int** A[], **int** p, **int** r) {
 // your code
}

**int** main(**int** argc, **char** \*argv[]) {
 FILE\* fh = fopen("randomInt.txt", "r");
 **int** len;
 fscanf(fh, "%d", &len);
 A = calloc(len, **sizeof**(**int**));
 **for** (**int** i = 0; i < len; i++) {
 fscanf(fh, "%d", A+i);
 }
 fclose(fh);
 QuickSort(A, 0, len-1);
 // check if they are sorted
 **for** (**int** i = 0; i < len; i++) {
 **if** (A[i] != i) {
 fprintf(stderr, "error A[%d]=%d\n", i, A[i]);
 }
 }
}
3. compile and run your program. Compile by $ cc quicksort.c -o quicksort so it generates the executable file named quicksort. Run it and type
$ time ./quicksort to see how much time it takes.

2.2 Convert quicksort.py to threaded version (name it qsortTh.py) using Python’s threading package. Good places to convert to threads is one of the recursive calls in QuickSort, since the two work on two disjoint parts of the array and are therefore independent of each other. The steps are

1. Create a new thread for one of the two recursive calls by calling threading.Thread(), and assign it to a variable. The target parameter is the function for the thread to call, and the args parameter is the tuple of parameters to pass.
2. Unlike POSIX threads, instantiating a thread does not start running it; you have to explicitly call the .start() method on the thread to start running it. The parent thread itself can do the other recursive call concurrently. (The parent could create two threads but it would be wasteful, since the parent would have nothing else to do).
3. (parent) wait for the (child) thread to complete by calling the .join() method on it.
4. When the data size is small (e.g., 10), it probably does not hurt to create threads for recursive calls, but when the data size is large (e.g., 20 million), then you want to limit the number of threads you create. Add code to limit thread creation based on the number of threads currently running. If it exceeds the (self-imposed) maximum number of threads (that you allow), then don’t make a new thread for recursive call; instead, just call QuickSort normally. Otherwise, make a new thread, start it, and join it.

Show the time result (Section 2.1.3 above)

2.3 Convert qsortTh.py from Part 2.2 to C (and name it qsortTh.c) using Pthreads. Note that the idea is similar to the Python version but slightly different. Here is a template: for qsortTh.c

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

**int** \*A; // array

**int** Partition(**int** p, **int** r) {

 // your code from Sec. 2.1 except array param is now global A

}

**void**\* QuickSort(**void** \*arg) {

// your code

}

**int** main(**int** argc, **char** \*argv[]) {

 // read randomInt.txt into array A

 // same as Sec 2.1.

 **int** args[2] = { 0, len-1 };

 QuickSort(args);

 // check if they are sorted. This part is same as Sec 2.1

 **for** (**int** i = 0; i < len; i++) {

 ...

 }

}

1. Declare a variable of type pthread\_t and call pthread\_create() by passing the pointer of the pthread\_t variable as first param; you can use NULL as the 2nd parameter; the name of the function for the thread to call as the 3rd parameter;

2. The fourth parameter to pthread\_create() is a pointer to the arguments. This means your QuickSort function cannot take (**int** A[], **int** p, **int** r) as its argument list; instead, they have to be a pointer to some array or struct where the value of these parameters are found. (see template code)

3. Note that unlike Python, as soon as you call pthread\_create(), the thread starts running right away. However, thread creation could fail, so you should check the return value and report an error if the thread cannot be created and exit.

4. You can use pthread\_join() to join the threads before returning.

5. Compile your code by $ cc -pthread qsortTh.c -o qsortTh -- note the -pthread flag to make sure it is linked properly.

Show the time result (Sec. 2.1.3).

3. Performance Analysis

Present a table of runtime that you measured using the time command for running

* $ time python3 quicksort.py # provided
* $ time python3 qsortTh.py # from section 2.2
* $ time ./quicksort # from section 2.1
* $ time ./qsortTh # from section 2.3

Is the threaded version faster or slower than the unthreaded version in C? in Python? Explain why in each case.