CS 3423 Operating Systems
Fall Semester 2017
Prof. Pai H. Chou

Assignment 4

Due Date: Sunday, October 15, 2017, 11:59pm
Submit electronically on iLMS

What to submit: One zip file named <studentID>-`hw4.zip` (replace <studentID> with your own student ID). It should contain four files:

- one PDF file named **hw4.pdf** for Section 1 and Section 3. Write your answers in **English**. Check your spelling and grammar. Include your name and student ID.
- Section 2: Turn in
  - Section 2.1: `quicksort.c`, `typescript1`
  - Section 2.2: `qsortTh.py`, `typescript2`
  - Section 2.3: `qsortTh.c`, `typescript3`
- Section 3: (part of hw4.pdf)

1. [40 points] Problem Set

1. [20 points] Under what circumstances does a multithreaded solution using multiple kernel threads provide better performance than a single-threaded solution on a single-processor system?
2. [20 points] Which of the following components of program state are shared across threads in a multithreaded process?
   a. Register values
   b. Heap memory
   c. Global variables
   d. Stack memory

2. [50 points] Programming Exercise

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 2 implementation of Quicksort. For Python3, see the comments about changing range(LEN) into list(range(LEN)).

```
def Partition(A, p, r):
    x = A[r]
    i = p - 1
    for j in range(p, r):
```

```
def QuickSort(A, p, r):
    if p < r:
        q = Partition(A, p, r)
        QuickSort(A, p, q-1)
```

```
        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
```

```
        QuickSort(A, q+1, r)
```

```
if __name__ == '__main__':
    LEN = 10
    L = range(LEN) # in Python3, do L = list(range(LEN)) instead
    import random
    random.shuffle(L)
    QuickSort(L, 0, len(L)-1)
    if L == range(LEN):  # Python3 list(range(LEN)) instead of 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 [10 points] Convert quicksort.py to C and call it `quicksort.c`. Turn in the source file and a typescript file named `typescript1` showing how you build and run it.

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 = 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. Capture this in typescript.

2.2 [20 points] 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.

Turn in the python source file (`qsortTh.py`) and a typescript file named `typescript2` showing that you run it successfully. Then show the `time` result (Section 2.1.3)

2.3 [20 points] 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.

Turn in the C source file (`qsortTh.c`) and a typescript file named `typescript3` showing that you run it successfully, and then show the `time` result (Sec. 2.1.3).

3. [10 points] 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.