Unit 6 Sample Problems - Threads (SOLN)

In this exercise, we will review the ideas of Threads.

- Length: 1:15 minutes with discussion.
- Questions: Q1-Q5. (optional: Q6)

Multicore Programming

1. [Acuña] Consider the algorithmic task of compiling a set of source files in a programming project. For each source file, the compiler needs to emit an object file containing byte-code that represents the original source code at a lower level. Then, each file is linked together (with standard libraries) to generate a final executable file.

   Say we want to design a compiler to compile a project as quickly as possible. Of the five issues in multi-core programming, which is the most problematic for multi-threading this system?

   Ans: [Acuña]

   Balance. Source files can be of vastly different sizes (think 1 method vs 100). Since the length of the files is proportional to how long they take to compile, we might end up with cases where compilation of one file takes so long that other the threads compiling other files end up idling. (A runner up would be Data Dependency, for cases where files dependent on one another. Like inheritance.)

2. [Acuña] Consider the algorithmic task of compressing a folder. The typical threaded implementation usually divides up files in a folder being compressed, and then sends out the work of compressing some sub-set of them to different threads. Each thread will produce compressed versions of each file it is given, which are then combined into a single compression archive. This gives a significant speedup.

   Another compression scheme uses a so-called a "dictionary". In this "dictionary", chunks of uncompressed data are mapped to indices with compressed data. The dictionary will end up stored in the compressed file so it can be used to reconstruct the compressed data. The idea is that if data repeats (e.g., same file twice), both compressed instances of the file can share the same compressed data. This can give a significant improvement in file size

   Say we want to design a dictionary based compression tool. Of the five issues in multi-core programming, which is the most problematic for multi-threading this system?

   Ans [Acuña]:

   Data Dependency. The issue is that we are limited to a single dictionary of pieces of compressed file. If two threads try to compress files and decide that they need to add an entry to the dictionary, then they will compute it in parallel, but only one result will actually get to be stored. Since the purpose of the dictionary is to support reuse, the slower thread will have to discard its result (the dictionary already has an entry), and instead reference the entry produced by the first.

3. [Lisonbee] Consider the following program and answer the question:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int fib(int n) {
    if (n == 0)
        return 0;
    else if (n == 1)
        return 1;
    return fib(n-1) + fib(n-2);
}

void runner(void* param) {
    int n = (int)param;
    printf("Fib @ %d is %d\n", n, fib(n));
}

void threadded_fib(int* arr, int size) {
    pthread_t threads[size];
    for (int i = 0; i < size; i++) {
        pthread_create(&threads[i], NULL, runner, (void*)arr[i]);
    }
    for (int i = 0; i < size; i++) {
        pthread_join(threads[i], NULL);
    }
}

int main() {
    int nums[] = {1, 6, 15, 2, 89};
    int size = 5;
    threadded_fib(nums, size);
    return 0;
}
```

Of the five issues in multi-core programming, which is the most problematic for this problem?

Ans: [Lisonbee]

Balance. Each thread computes a single fibonacci number, but the placement of the number in the sequence will determine how much work that thread does. In the example sequence, the first thread will get the value '1' and return '1' without recursively calling fib at all. The fifth thread gets '89' and returns the result of fib(89) (a very large value; will overflow countless times), but the computation necessary to calculate this value is many times greater than the first thread. That being said, this thread will take much longer to finish it's work than the first thread, leading to imbalance between the amount of work each thread has done.

Thread Libraries

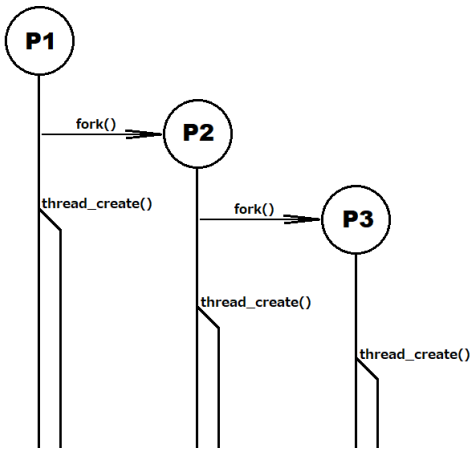
4. [Silberschatz 4.15 edited] Consider the following code fragment and answer the following questions:

```
pid_t pid;
temp = fork();

if (temp == 0) {
        fork();
}

thread_create(...);
```

   (a) Using "lifeline notation", draw the creation of processes and threads during execution.



   (b) How many unique processes are created? (Do not include the initial process.)
   2

   (c) How many unique threads are created? (Hint: processes don't count!)
   3

5. [Acuña] Consider the following program written with the pthreads library:

```
#include <pthread.h>
#include <stdio.h>

int t1, t2, input;

void* runner(void *param) {
    int result, upper = (int)param;
    int a = 1, b = 1, c = 1;

    for (int i = 2; i < input-upper; i++) {
        c = a + b;
        a = b;
        b = c;
    }

    if(upper == 1)
        t1 = c;
    else
        t2 = c;

    pthread_exit(0);
}

int main(int argc, char *argv[]) {
    pthread_t tid1, tid2;
    pthread_attr_t attr;
    input = 10;

    pthread_attr_init(&attr);
    pthread_create(&tid1, &attr, runner, 1);
    pthread_create(&tid2, &attr, runner, 2);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("result = %d\n", t1+t2);
}
```

What is this programming trying to compute and how does it uses threads?

Ans [Acuña]:

This program is computing the 10th Fibonacci number. It uses one thread to compute fib(n-1) and store it in t1, and another to compute fib(n-2) and store it into t2.

6. [Acuña] Consider the problem of finding the maximum element in an array of values, which is solved by the following program:

```
#include <stdio.h>
#define COUNT 1024

int data[COUNT];

int findMaximum(int data[], int size) {
  int max = data[0];
  for(int i = 1; i < size; i++)
    if(data[i] > max)
      max = data[i];
  return max;
{

void main (void) {
  //assume that data is populated here

  printf("max is %d", findMaximum(data, COUNT);
}
```

Using pthreads, rewrite this program to perform this same work but split across two threads.

Ans: [Acuna]

```
#include <pthread.h>
#include <stdio.h>
int data[1024];
struct info { //fancy way
  int* numbers; int start; int end; int result;
};
void* findMaximum(void *param) {
  struct info* in = (struct info*)param;
  int max = data[in->start];
  for(int i = in->start+1; i < in->end; i++)
    if(data[i] > max)
      max = data[i];
  in->result = max;
  pthread_exit(0);
}
void main (void) {
  info ti[2];
  pthread_t tid1, tid2;
  pthread_attr_t attr;
  //assume that data is populated here
  ti[0].numbers = data; ti[0].start = 0; ti[0].end = 1024/2;
  ti[1].numbers = data; ti[1].start = 1024/2; ti[0].end = 1024;
  pthread_attr_init(&attr);
  pthread_create(&tid1, &attr, findMaximum, &ti[0]);
  pthread_create(&tid2, &attr, findMaximum, &ti[1]);
  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);
  int final = ti[1].result > ti[0].result ? ti[1].result : ti[0].result;
  printf("max is %d", final);
}
```