Homework 3 Solutions
Operating Systems, V22.0202. Fall 2006, Professor Yap

SOLUTION PREPARED BY Instructor and T.A.s

INSTRUCTIONS:

- This is due on Nov 1.
- Please read questions carefully. When in doubt, ask.
- The written part must be handed in separately, as a hardcopy. The programming part should be packaged as a tar file as described below.

---

1 WRITTEN PART

1. (15 Points)
   On page 241, the following solution was proposed for the 5 dining philosophers problem: *the odd numbered philosopher picks up her left chopstick first, whereas the even numbered philosopher picks up her right chopstick first.* Show carefully to show that no deadlock can occur.

   HINT: let the philosophers be numbered $0, 1, 2, 3, 4$. Can philosopher 1 (P_1) wait for philosopher 2 (P_2) indefinitely? We are assuming that any processor that enters its CS must exit its CS and release its chopsticks in finite time.

   SOLUTION The answer to the hint question is NO. That is because if P_1 is waiting for P_2, it means P_2 has acquired the second fork and would be out of its CS eventually. Suppose i, j are two neighboring philosophers. Write $i \to j$ to mean "philosopher i is waiting for philosopher j". If there is a deadlock, it MUST mean that there is a cycle of the form $i_1 \to i_2 \to i_3 \to \cdots \to i_n \to i_1$ where $i_1, i_2, \ldots, i_n \in \{0, 1, 2, 3, 4\}$. But notice that $i_1 \neq i_3$ and $i_2 \neq i_4$, etc. In other words, there are no deadlocks of the form where P_i is waiting for P_{i+1} and P_{i+1} is waiting for P_i. This implies that any deadlock must have the form $0 \to 1 \to 2 \to 3 \to 4 \to 0$. But this is impossible – if P_1 is waiting for P_2 then P_2 is not waiting anyone else! ♠

2. (10 points)
   Exercise 6.5, p.211 (Variant of RR).

   Consider a variant of the RR scheduling algorithm where the entries in the ready queue are pointers to the PCBs.
   a. What would be the effect of putting two pointers to the same process in the ready queue?
   b. What would be two major advantages and two disadvantages of this scheme?
   c. How would you modify the basic RR algorithm to achieve the same effect with the duplicate pointers?

   SOLUTION a. This would essentially double the quantum for the process.

   b. Advantages: we can increase the quantum of any process by any positive integer multiple. This method does not need any change the standard RR algorithm. Disadvantages: There is no way to give non-integer multiplies of a quantum to a process. If we want big integer multiples, we can clog up the RR queue. If we do not take care to avoid unnecessary context switches, we may swap a process out, only to swap it right back in. We can avoid this by making sure that multiple copies of the pointers in one process are consecutive in the queue, and before we swap out a process, make sure that the next process is really different.

   c. We can associate a constant $c_i > 0$ for each job/process J_i in the RR queue, and this means that J_i will be allocated c_i times the unit quantum of time. For instance, assuming priority of J_i is $p_i > 0$, we can let $c_i = 1/p_i$ (recall our convention is that smaller numbers have higher priority. ♠

3. (10 points)
   Exercise 6.7, p.211 (Preemptive priority-scheduling)

Consider the following preemptive priority-scheduling algorithm based on dynamically changing priorities. Larger priority numbers imply higher priority. When a process is waiting for the CPU (in the ready queue, but not running), its priority changes at a rate α; when it is running, its priority changes at a rate β. All processes are given a priority of 0 when they enter the ready queue. The parameters α and β can be set to give many different scheduling algorithms.
a. What is the algorithm that results from $\beta > \alpha > 0$?
b. What is the algorithm that results from $\alpha < \beta < 0$?

SOLUTION Preemptive priority means that when a new job arrives, it may preempt the current job if the new job has higher priority. Recall the book's convention that lower number indicates higher priority. However, this question specifically asked for the opposite convention!

a. The result is FCFS since any entering jobs will have lower priority (i.e., 0) compared to jobs already in the system. Also, the running job will have larger and larger priority compare to the non-running jobs, and so will never be preempted.

b. The result is LIFO since the entering job with priority 0 will be higher to jobs already in the system (which will have negative priority). Moreover, if a job is runninging running, it has higher priority. Since $\alpha < \beta < 0$, its priority is becoming negative slower than the ones in the waiting queue, i.e., it continues to run. Thus the behavior is LIFO.

REMARK: In case we use the books normal convention (i.e., a larger priority number means a lower "scheduling priority") then we get the following anomalous behavior:

a. Suppose a job J_1 is currently running because its priority p_1 is smaller than p_2, the priority of the job J_2 at the front of the queue. After t units of time, p_1 becomes $p_1 + t\beta$ and p_2 becomes $p_2 + t\alpha$. If $\beta > \alpha$, then there is a time $t > 0$ when $p_1 + t\beta = p_2 + t\alpha$, or $t(\beta - \alpha) = p_2 - p_1$. Hence, after time $t^* = (p_2 - p_1)/(\beta - \alpha)$, the priority of J_2 would higher than that of J_1. Now we have a problem: after time t^*, there is a kind of deadlock between J_1 and J_2 – the moment we schedule J_2, its priority would become slightly smaller than J_1, and we would have to preempt it and replace it with J_1. Thus there is perpetual preemption and no real progress.

b. With the same scenario as in part a, but suppose $\alpha < \beta < 0$. We see that if $p_1 < p_2$ then $p_1 + t\beta$ will eventually be equal to $p_2 + t\alpha$. Again, after time $t(\beta - \alpha) = p_2 - p_1$, the two processes will have the same priority, and again we have the perpetual preemption. ♠

2 PROGRAMMING PART

We begin to program with threads. In this exercise, you are to repeat the multiple gcd problem of hw2, but using POSIX threads (pthreads) instead of forking multiple processes. This is worth 50 points.

Please read Lectures 7 and 8 for information on pthreads. Another excellent web reference is `http://www.cs.cf.ac.uk/Dave/C/`. This site gives you not just general C programming information, but also thread and synchronization information. You will need to understand the basics of pointers to do this problem. Please start your programming part immediately! Feel free to email the grader and me if you run into problems.

- OVERVIEW. Your main program should be called `tgcd.c`. The input to `tgcd.c` is a sequence of pairs of positive integers, as before. If there are k pairs, then the main thread will spawn k "gcd threads" and one "worker thread". Each gcd thread will compute the GCD of one pair of integers. However, the gcd thread does not know how to compute the modulo operation – only the worker thread knows how. Hence the gcd thread must submit pairs (a, b) of integers to the worker thread, who will compute and return the modulus $a \mod b$. When a gcd thread has computed the GCD of its pair, it prints its answer and exits. We want this printout to be informative – it should tell us (1) the thread ID (not it's index), (2) the original arguments and the gcd, and (3) the number of mod operations. E.g.,

```
Thread 12345:  gcd(9,12)=3 using 3 mod operations
```

  The main thread should join to each of the gcd threads. When all the gcd threads have exited, the main thread will send a thread cancellation command the worker thread and exit.

- COMMUNICATION. The chief difference from the process version of this problem is that you no longer need pipes. Instead, global variables can be seen by all the threads. To achieve synchronization, we want you to use mutexes. There are different ways to achieve this, so you MUST design a solution, and include its description in the README file.
- Finally, you are to create several runs using various sets of input pairs, and give the timing for these runs. We will give you all the hints necessary to do the programming part.
- CHECKLIST for what to hand-in:
  1. A single tar file called [your-last-name]-hw3.tar The [your-last-name] should be in small letters (no caps).
  2. The tar file should contain a subdirectory called [your-last-name]-hw3. You can check if your tar file is in conformance by typing "tar tvf [your-last-name]-hw2.tar" in any directory containing the tar file. It will list the files in the tar file, including any SUBDIRECTORY structure.
  3. This subdirectory should contain a Makefile file, a README file, the main program "tgcd.c" and any associated files.
  4. The README file must contain a description of your thread synchronization, your timings, and any comments on your experiments.
  5. The Makefile should have these targets: the default target (a plain "make") should compile your program. [If we cannot compile the program, the grade is automatically 0.] If we type "make test", it should be equivalent to the following test:

     ```
     tgcd.exe 2 3 15 9 123456 7890 123 456789 98765 4321 987654 3210
     ```

     and finally, we should duplicate your timing experiments by typing "make time".
  6. Email to grader, cc to me.

3 USEFUL INFORMATION

- It may be useful to use structures (or ”structs”) to encode pairs of integers to be passed around. Please see my notes on C programming on the web for more information on structures. In your case, you do:

  `struct pair {int first; int second;}; // define "pair" structure`

  `struct pair onePair; // declare variable "onePair"`

  `onePair.first = 1; onePair.second=2; // assignment and reference`

- To get one’s thread ID,

  `pthread_t tid;`

  `tid = pthread_self();`

- To join to a child (tid) thread, ignoring the return status:

  `pthread_join( tid, NULL);`

- Here is the Hello World Prograd for pthreads:

```
#########################################
#include <pthread.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define N 5

void *PrintHello(void *threadid);        // actual code below!

// MAIN THREAD ROUTINE
int main (int argc, char * argv[]) {
   pthread_t threads[N];
   int rc, t;
   for (t=0; t<N; t++) {
      printf("Creating thread %d\n", t);
      rc = pthread_create( &threads[t], NULL, PrintHello, (void *)&t);
      if (rc){
         printf("ERROR; return code of pthread_create() is %d\n", rc);
         exit(-1);
      }
   }
   pthread_exit(NULL);
}//main

// CHILD THREAD ROUTINE
void *PrintHello(void *threadid) {
   printf("\n%d: hello world!\n", threadid);
   pthread_exit(NULL);
}

#########################################
```

To compile this, type ”gcc thello.c -lpthread”.

4 SOLUTION PROGRAM

```
//////////////////////////////////////////////
// Solution to hw3
// Chee Yap, Fall 2006
//////////////////////////////////////////////
/*
 * tgcd.c
 *
 * MULTI-THREADED MULTI-GCD COMPUTATION
 *
 *  When you call "tgcd" with n pairs of integers,
 *  the main thread will spawn n gcd threads.  Each gcd thread
 *  will compute the gcd of a pair of integers.
 * However, each gcd process must call a worker thread
 * to compute the mod operation.

int main(){
}//main

///////////////////////////////////////////////////
// END
///////////////////////////////////////////////////
```