Homework 2 Solutions
Operating Systems, V22.0202. Fall 2006, Professor Yap

SOLUTION PREPARED BY Instructor and T.A.s

INSTRUCTIONS:

- This is due on Oct 4.
- Please read questions carefully. When in doubt, ask.

---

1 WRITTEN PART

1. (15 Points)
   Exercise 7.1 (p.273). Show the correctness of Dekker's Algorithm.

   NOTE: Dekkers' Solution is supposed to be Fig.7.43 of the textbook. You see two procedures, enteringCriticalSection(t) and leavingCriticalSection(t). Process t (t=0 or 1) is supposed to call these procedures before and after the critical section. Unfortunately (my apologies for not checking carefully) this printed solution has bugs – you could even see that the braces are not balanced. In view of this, we will grade whatever you submit very generously.

   But we will discuss the correct solution anyway. Please study this discussion carefully for your midterm! Here is Dekker's algorithm (in the same style as the presentation of Peterson's algorithm):

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
     Initially:
               P1wants=P2wants=false (Important!)
               turn=1                (Unimportant)
     -----------------------------------------------------------------
     P1:                            | P2:
                                    |
     Loop forever                   | Loop forever
                                    |
P:       P1wants <-- true           |     P2wants <-- true
                                    |
A:       while (P2wants) {          |     while (P1wants) {
           if (turn=2) {            |        if (turn=1) {
               P1wants <-- false    |            P2wants <-- false
B:             while (turn=2) {}    |            while (turn=1) {}
               P1wants <-- true     |            P2wants <-- true
           }                        |        }
         }                          |     }
                                    |
C:       critical-section           |     critical-section
                                    |
V:       P1wants <-- false          |     P2wants <-- false
         turn <-- 2                 |     turn <-- 1
                                    |
D:       non-critical-section       |     non-critical-section
                                    |
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

   SOLUTION

   Let us notice some salient features of this code. There are six interesting labeled "points": Points C and D correspond to the CS (critical section) and non-CS parts of the code. Points P and V correspond to the usual protocols we execute before and after a CS. Finally, there is a doubly nested while-loop. Points A and B mark the "outer-loop" and the "inner-loop", respectively.

Let us now show that Dekker's solution satisfies the 3 requirements of the text:

1. Mutual Exclusion: We discuss process P1, but the situation is entirely symmetrical with P2. It is important to exploit this symmetry.

(a) First observe that "P1want" is only set by process P1. But "turn" is set by both processes.

(b) When P1 exits from its outer-loop, but before it finishes its critical section (CS), we have the condition [P2wants=false]. However, "turn" can be 1 or 2.

(c) Now [P2wants=false] can mean P2 is either (I) at point B or (II) between points C and D of its code. In other words, P2 is NOT in its critical section.

(d) Can P2 move into its own CS as P1 proceeds to its CS? There are two possibilities.

CASE (I): In this case, turn=1. Moreover, until P1 changed "turn" to 2, we will have turn=1. Thus P2 will remain at point B. Thus P1 can safely proceed to its CS.

CASE (II): In this case, when P2 moves from point D to point P, it will encounter P1wants=true. This means it will enter point B, and will remain there until P1 can finished its CS

(e) We conclude that when P1 is in its CS, it is safe.

2. Progress: a process that is not trying to enter the CS should not hold up other processes.

(a) To be precise, we say that process P1 is not trying to enter the CS when it is in its non-CS (point D in the code).

(b) We must show that P2 will be able to get through both while-loops in finite time as long as P1 remains in its non-CS.

(c) Notice that just before P1 enters its non-CS, it has (P1wants=false and turn=2). Now "turn" might be later changed to 1 by P2, but P1wants=false will not change as long as P1 is in non-CS.

(d) We must prove that P2 will be able to enter its CS if it chooses to. Can P2 get stuck in its outer-loop? This is clearly impossible since P1wants=false.

(e) But can P2 get stuck in its inner-loop? (Remember that because we make no assumptions about relative speeds of the 2 processes, P2 might be inside its outer-loop when P1 is in its non-CS).

(f) But if P2 is inside its inner-loop, it means that turn=1. That means that "turn" has changed by P2 from 2 to 1 while P1 was in its non-CS. This meant that P2 has gone from point D to point B. But that is impossible as P1wants=false throughout this time. P1wants=false means P2 will never ever its outer-loop.

3. Bounded Waiting: we show a very strong version of this property – that P2 cannot get to its CS twice in a row while P1 is waiting to enter its CS

(a) Suppose P1 waited for P2 twice-in-a-row in its outer-loop. THIS IS TRICKY TO STATE PRECISELY SINCE WE DO NOT ASSUME ANYTHING ABOUT RELATIVE SPEEDS OF PROCESSES. SO WHAT DO WE MEAN? It means there are two time instances, $t_1 < t_2$, such that the outer-loop of P1 found P2wants=true. But P2 has gone through its CS twice in the period $[t_1, t_2]$.

(b) To show a contradiction, note that P1wants=true during $[t_1, t_2]$. Hence, when P2 goes from its CS back to its CS again, it would have gotten inside its outer-loop. Moreover, turn=1 (set by P2 itself). This meant that P1 would have been stuck inside its inner-loop. This contradicts our assumption that P2 proceeded to its CS the second time.

(c) Now let us consider the other possibility: Suppose P1 waited for P2 twice-in-a-row in its inner-loop. WHAT DO WE MEAN? It means there are two time instances, $t_1 < t_2$, such that the inner-loop of P1 found turn=2. But P2 has gone through its CS twice in the period $[t_1, t_2]$.

(d) After P2 goes through its CS the first time, it would set turn=1 at some time $t_3 \in (t_1, t_2)$. That means "turn" must be reset to 2 again in the period $[t_3, t_2]$. This is impossible since only P1 can reset "turn" but P1 was assumed to be inside the inner-loop.

♠

2. (20 Points)
Peterson's solution to MUTEX was for 2 processes.
(a) Generalize Peterson's solution to an arbitrary number of processes. Please write the solution in a conceptual formulation (as we did in lectures, not as a Java program as in the text).
(b) Show that your solution achieves mutual exclusion.
(c) Show that no process can be locked out in your solution.
(d) Show that your solution is fair. NOTE: (b),(c),(d) correspond to three the criteria in the text.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Multi-Process Peterson Solution:

     int N;                      // N>1 is number of processes
                                 //
     bool wants[N];              // Does Process[i] want to enter CS?
     int  turn[N];               // Who wants a turn?  BUT turn[j] is NOT
                                 //   necessarily associated with Process[j]
                                 //
     int count() {               // subroutine to count number of wants.
        n <-- -1;                // But observe that each processor only counts
        for (j=0; j<N; j++)      // the number of processes OTHER than itself.
           if (wants[j]) n++     // E.g., if there are no OTHER process who
        return(n)                // wants, count()=0.
     }

     Initially, wants[i]=false  for all i.
     -----------------------------------------------------------------
     -----------------------------------------------------------------

     P(i):                       // i=0,...,N-1

       Loop forever

P:       wants[i] <-- true

A:       for (j=0; j<=count()-1;  i++) {
             turn[j] <-- i;
B:           while (turn[j]=i and i<count()-1) {}
         }

C:        Critical Section

V:        wants[i] <-- false

D:        Non Critical Section

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

SOLUTION This is clearly very similar to our previous solutions, with the labels A,B,C,D and P,V.

Let us say that Process i is "in contention" if wants[i]=true. Thus, count() is number the number of processes that a process in contention must compete against, not counting itself.

The for-loop at point A is very interesting: Process i sets turn[j]=i and then enters a while-loop to wait for turn[j] to become non-i. We then say the process is WAITING AT LEVEL j.

First, to get a sense of the dynamics. let us assume that count()=m and does not increase (i.e., no new process tries to enter the CS). What happens in this situation? We see that Process i has to wait at each level in turn. If it were waiting at level m-1, and turn[m-1] becomes non-i, then Process i will be able to get to its CS. After it reaches it non-CS, it will cause count() to decrement. This will now allow another process to enter its CS.

(b) We now show why there is mutual exclusion: Process i enters its CS only when turn[c-1]=j for some j not equal to i. This means Process j is waiting at level c-1. But how did Project j reach level c-1? Because some Process k set turn[c-2] to k. Thus, process k is waiting at level c-2. And so on. In other words, there exactly c-2 processes, each waiting at a distinct level (0 to c-1). Now if some new process enters the contention, count() increases, but increasing count() will not allow any one new to enter the CS. It does allow a process to wait at a higher level.

Only after Process i has finished visiting its CS, it sets its wants[i]=false. This will decrease count(), which eventually permits another process to enter the CS.

(c) We now show that no process can be locked out. As long as every process is making progress, eventually it will wait at the higest level, and it will be admitted into its CS.

(d) Fairness (what we call Bounded Waiting in the previous question): this comes from the fact that a process has to wait at successive "levels" and so no process can "bypass" another process that is already in the contention process. ♠

2 PROGRAMMING PART

This part addresses Interprocess Communication: you will need the tools of fork and exec (as in hw1) and also some facility with pipes. It is worth 50 points.

Please start your programming part immediately! Feel free to send me and the grader emails when you run into problems.

- In this programming homework, you are to simulate the concurrent solution of a computational task using processes.

  The task is to to compute the GCD of various pairs of numbers. Given two numbers m and n where $m \geq n > 0$ are two integers, their $GCD(m,n)$ is defined to be the largest number that divides both m and n. For instance, $GCD(15,9) = 3$. There is the well-known Euclidean algorithm for computing GCD. Starting from the numbers

  $$m_0 = m, \qquad m_1 = n$$

  Euclid's algorithm says to compute the sequence of integers,

  $$(m_0, m_1, m_2, \ldots, m_k, 0) \tag{1}$$

  where $m_{i+1} = m_{i-1} \mod m_i$ for all $i = 2, 3, \ldots$. Recall that the **modulo operation** $a \mod b$ simply returns the remainder of a divided b. Hence $0 \leq (a \mod b) < b$. Thus $0 \leq m_{i+1} < m_i$ for $i \geq 2$. Hence the sequence (1) must eventually reach 0. If $m_{k+1} = 0$ (for some $k \geq 1$) then it is easy to show that the previous number m_k is the GCD.

  E.g., for $(m,n) = (15,24)$, we get the sequence $(24,15,9,6,3,0)$ and so $k = 4$ and $m_4 = \texttt{GCD}(15,9) = 3$. A sample program (testGCD.c) for computing gcd is provided here.

- OVERVIEW. Your main program should be called `gcd.c`. The input to `gcd.c` is a sequence of pairs of numbers. If there are $k \geq 1$ pairs, then the main process will spawn k children processes. Each child process will compute the GCD of one pair of numbers. However, the child process does not know how to compute the modulo operation. Only the parent process knows how. Hence the child must submit pairs (a,b) of integers to the parent who will compute and return the modulus $a \mod b$. When a child process has computed the GCD of its pair, it exits. When all the children has exited, the parent exit. For instance, if we type

  ```
  > gcc -o GCD GCD.c
  > GCD 24 17 18 10 987654321 123456
  ```

then GCD will spawn 3 processes which eventually prints ”1”, ”2” and ”3” (as the GCD’s of the three pairs).

We want the parent process to implement the $a \mod b$ operation by repeated subtractions. Moreover, the parent should use round robin method to service the children.

- COMMUNICATION. The parent and each child communicates through a two-way **pipe** (this is just two pipes, one from parent-to-child, and another from child-to-parent). That is, the child will write a pair (a, b) of integers in the child-to-parent pipe, and the parent will respond by writing the integer ($a \mod b$) in the parent-to-child pipe. There are two ways to do this:
  (a) One way is for the parent to try to read from each of its ”children-to-parent” pipes in the round-robin fashion. To accomplish this, the parent must perform what is known as a **nonblocking read**. Such a read will never block – even if there is nothing to read. The other kind of reading is called a **blocking read**. Note that it is quite acceptable for the child read the parent-to-child pipe in a blocking manner.
  (b) The other way is for a child to signal the parent after it has placed a pair (a, b) in the child-to-parent pipe. The parent responds to the signal by searching through the pipes from each of the children to find a pair (a, b) to work on. In order to avoid busy waiting, the parent will only do this search in response to a signal.

- We recommend using the non-signaling version for this homework.

- Finally, you are to create several runs using various sets of input pairs, and give the timing for these runs.

- Thus, you must know how to:

  – set up pipes
  – how to read/write from and to pipes
  – how to do non-blocking reads
  – time the running time of your program.

  We will give you all the hints necessary to do the programming part.

- WHAT TO HAND IN: a single tar file containing a Makefile file, README file, and all necessary programs.

  UPDATED INSTRUCTION: you need to hand in the written part in hard copy. The tar file should be called [your-last-name]-hw2.tar where [your-last-name] are all in small letters. Moreover, all the files must be in a SUBDIRECTORY called [your-last-name]-hw2. You can check if your tar file is in conformance by typing ”tar tvf [your-last-name]-hw2.tar” in the directory containing this tar file. It will list the files in the tar file, but it will also show you any SUBDIRECTORY structure as well.

  You must give your timings and explain your experiments in the README file.

  I should be able to duplicate your experiments by typing ”make time”.

3 USEFUL INFORMATION

- Here is a routine to set the status flag for files or pipes.

```
#include <stdio.h>
#include <sys/types.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>        // include all these for convenience

void setFlag(int fd, int flags) {
                             // flag is a bitvector for bits to turn on
  int val;
  if ((val = fcntl(fd, F_GETFL, 0)) < 0)  // get original bits
   perror("fcntl F_GETFL error");     // F_GETFL are predefined
  val |= flags;                            // turn on the bits in flag;
  if (fcntl(fd, F_SETFL, val) <0)          // set the new bits
   perror("fcntl F_GETFL error");
}
```

  For the parent to read the ”child2parent pipe” in a nonblocking way, we execute:

```
setFlag( child2parent[0], O_NONBLOCK);     // O_NONBLOCK are predefined
```

- How do you send a pair of integers, m and n, to the parent on the pipe? Here is a solution sprintf and sscanf.

  1. Child writes the values of m and n in buffer:
     `sprintf(buf,"%d,%d", m, n);`
  2. Child writes buf into the pipe to the parent.
  3. Parent reads from pipe into its own BUF.
  4. Parent use sscanf to decode from BUF:
     `sscanf(BUF,"%d,%d", &M, &N);`
     where M, N are integers to hold the values of m and n.