Homework 3 with SOLUTIONS
Operating Systems, V22.0202
Fall 2007, Professor Yap

Due: Wed Oct 10
SOLUTION PREPARED BY Instructor and T.A.s

- Please read questions carefully. When in doubt, please ask.
- The written homework is to be submitted in hardcopy during class, but the programming part sent to us in a single file (as detailed below) by midnite.

Question 1 (5 Points)
Problem 4.2, p.146. (Context switch for user-level threads)
Describe the actions taken by a thread library to context switch between user-level threads.

SOLUTION: The user-level threads are known only within a given process. To context switch, we only need to save the thread-specific context: the program counter, CPU registers, and the thread-specific stack pointer.

◊

Question 2 (5 Points)
Problem 4.5, p.147. (Multiprocessors and user-level threads)
Can a multithreaded solution using multiple user-level threads achieve better performance on a multiprocessor system than on a single processor system? Note: you must give the reasoning behind your "Yes" or "No" answer.

SOLUTION: We assume that user-level threads are not known to the kernel. In that case, the answer is because the scheduling is done at the process level.
On the other hand, some OS allows user-level threads to be assigned to different kernel-level processes for the purposes of scheduling. In this case the multithreaded solution could be faster.
AN ASIDE: Note that multithreaded solutions can still be useful in a uniprocessor environment because they may may simplify programming logic.
Moreover, this can sometimes lead to a more efficient solution: a multi-threaded solution on a uniprocessor can faster may be faster than a single-threaded solution on a uniprocessor. HOW can this happen? If a task can be naturally multi-tasked, and one of the tasks is much slower than another, the multithreaded solution can take advantage of this! It would be much more complicated to figure out the logic for a single-threaded solution to achieve the same efficiency.
Of course this is not directly relevant to our question because this question is comparing uni- with multi-processor environments.

◊

Question 3 (15 Points)
In Homework 2, we implemented the toy shell. Suppose we want to implement pipes between processes. Describe what changes needs to be done to your `tsh.c` program. For simplicity, assume that there is only one use of the pipe directive "|".

HINT: you need to use the `dup()` system call. You need not write out complete programs, but give explicit code fragments to show how the critical parts are implemented.

SOLUTION: Assume the parser has divided the input command line into tokens.
STEP 1: first separate the tokens into **subcommands** that are separated by |. For instance, the input command

```
ls  -a  |  grep  "m*"  |  wc
```

has 7 tokens, but it will be subdivided into 3 subcommand: `ls -a`, `grep "m*"`, `wc`. In particular, if there are no pipes, there is only one subcommand. Let N be the number of subcommands.
STEP 2: Execute a fork(). The parent process returns to the main command loop.
STEP 3: The child process executes the following:

```
int pid;
int pipefd[2];

for (i=2; i<=N; i++)
    pipe(pipefd); // for simplicity, we do check if pipe fail
    pid = fork(); // nor not check if fork fail
    if (pid == 0) // child
        close(0); // close standard input of child
        dup(pipefd[0]); // now pipefd[0] is standard input
        close(pipefd[0]); // close duplicated pipefd[0]
        close(pipefd[1]); // must close pipefd[1] also
    else // parent
        close(1); // close standard output of parent
        dup(pipefd[1]); // now pipefd[1] is standard output
        close(pipefd[1]); // close duplicated pipefd[1]
        close(pipefd[0]); // must close pipefd[0] also
        do "exec" of the (i-1)-st subcommand
do "exec" of the N-th subcommand // the N-th child reaches here
```

$\diamondsuit$

Question 4 (70 Points)

- INTRODUCTION. 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 (greatest common divisor) of one or more pairs of numbers. Given two positive integers m and n, their $GCD(m,n)$ is defined to be the largest number that divides both m and n. Clearly, $GCD(m,n) \geq 1$. For instance, $GCD(15,9) = 3$. There is the well-known Euclidean algorithm for computing GCD. Since $GCD(m,n) = GCD(n,m)$, we may assume that $m \geq n > 0$. Initially, let

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

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

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

  where for each $i \geq 2$, we define $m_{i+1} = m_{i-1} \mod m_i$. Recall that the **modulo operation** $a \bmod b$ 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 `gcd.c` for computing gcd is provided here.

- PROBLEM OVERVIEW. Your main program should be called `mgcd.c` (multiple gcd). The input to `mgcd.c` is a sequence of pairs of numbers. If there are $k \geq 1$ numbers, then the main process will spawn $\lfloor k/2 \rfloor$ 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 mgcd.c -o mgcd
  > mgcd 24 17 18 10 987654321 123456
  ```

  then GCD will spawn 3 processes which eventually prints "1", "2" and "3" (these are 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 to read the parent-to-child pipe in a blocking manner.
  (b) The other way is to use the **signaling mechanism** for unix processes. A child can 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.
  BUT NOTE THAT WE REQUIRE the non-signaling version (i.e., method (a)) 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: similar to previous homeworks, we want a single tar file containing a Makefile file, README file, and all necessary programs. You must give your timings and explain your experiments in the README file. We should be able to duplicate your experiments by typing "make time".

- USEFUL INFORMATION.
  - To read an integer from the command line, it is useful to know the `atoi()` library function to convert a string of digits into an integer. E.g., the following code fragment converts the first two arguments of the command line into integers:

    ```
    int arg1 = atoi(argv[1]);
    int arg2 = atoi(argv[2]);
    ```

– 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 <fcntl.h>   // needed by setFlag()
#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.

♢

1 SOLUTION PROGRAM

```
//////////////////////////////////////////////
// Solution to hw2
// Chee Yap, Fall 2006
//////////////////////////////////////////////
/*
 * mgcd.c
 *
 * MULTI-PROCESS GCD COMPUTATION
 *
 *  When you call "mgcd" with n pairs of integers,
 *  the parent process will spawn n child processes.  Each child process
 *  will compute the gcd of a pair of integers.
 *
 *  The Euclidean algorithm is used: the GCD
 *  of m, n (where m>n>=0) is computed via a sequence
 *
 *  m_0 > m_1 > m_2 >...> m_k > m_{k+1} = 0
 *
 *  where m_{i+1} = m_{i-1} mod m_i for all i.
 *  We return m_k as the GCD of m,n.
 *
 * However, each child process must call the parent process
 * to compute the mod operation.  We set up two arrays of pipes
 * pipe2parent[20][2] and pipe2child[20][2]
 * between each child and the parent.  The i-th child will write
 * a pair of numbers (a,b) into the pipe2parent[i][1],
 * and the parent will return the result (a mod b) into pipe2child[i][0].
 * To find such pairs (a,b), the parent will read
 * each pipe2parent[j] (j=0,1,2,...) in a non-blocking way.
 * The i-th child will keep reading pipe2child[i][0] until it receives
 * the answer from its parent.  When the child has found
 * the GCD, it prints this "GCD(m,n)=d" for suitable m,n,d, and exits.
 *
 * The parent also keeps track of the number of child that
 * has found its GCD (that corresponds to the number of times it
 * returns a 0 back to a child).  The parent exits when this
 * is done.
 *
 *  E.g.,
 *              > gcd_par 100 154 3 2 25 15
 *
 *              gcd(100,154)=4
 *
 *              gcd(3,2)=1
 *
 *              gcd(25,15)=5
 *
 *              parent exits!
 *
 * SYSTEM BUG:  We need to call "sleep(1)" to slow down
 * down the processes, or else program hangs
 * when there are more than 2 processes
 *
 * Chee Yap (Oct 2006)
 **/

#include <stdio.h>
#include <sys/types.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>

////////////////////////////////////////////////////
// global variables
////////////////////////////////////////////////////
char buf[1024]; // buffer

#define N 20 // maximum number of processes
int pipe2parent[N][2];  // array of pipes
int pipe2child[N][2];

////////////////////////////////////////////////////
// Routine to set status flag for files
////////////////////////////////////////////////////
void
set_fl(int fd, int flags) // flag=file status flags to turn on
{
int val;

if ((val = fcntl(fd, F_GETFL, 0)) < 0)
perror("fcntl F_GETFL error");
val |= flags; // turn on flags
if (fcntl(fd, F_SETFL, val)<0)
perror("fcntl F_SETFL error");
}//

////////////////////////////////////////////////////
// Swap values
////////////////////////////////////////////////////
void
swap(int *a, int *b) {
int tmp;
tmp=*a; *a=*b; *b=tmp;
}//

////////////////////////////////////////////////////
// Clear buffer
////////////////////////////////////////////////////
void
clearbuf(char *buff){
int i;
for (i=0; i<1024; i++) buff[i]='\0';
}
```

```
////////////////////////////////////////////////////
// main
////////////////////////////////////////////////////
int
main(int argc, char ** argv)
{
int kk;
int NumPairs = (argc-1)/2;
printf("Number of Pairs = %d\n", NumPairs);
if (NumPairs> N)
perror("too many pairs");

int i;
int m, n, p, mm, nn;
int len;
int child_pid, parent_pid;
int countdown = NumPairs;

////////////////////////////////////////////////////
//fork children processes
////////////////////////////////////////////////////
for (i=0; i< NumPairs; i++){
// get args
m = atoi(argv[2*i + 1]);
n = atoi(argv[2*i + 2]);
if (m<n) swap(&m, &n);

if ((pipe(pipe2parent[i]) == -1)
|| (pipe(pipe2child[i]) == -1))
perror("pipe");

set_fl(pipe2parent[i][0], O_NONBLOCK); // set nonblocking
set_fl(pipe2child[i][0], O_NONBLOCK); // set nonblocking
child_pid = fork();
switch(child_pid){
  case -1: // error
   perror("fork");
exit(1);
break;
  case 0:  // child
printf("Child %d\n", i+1);
close(pipe2parent[i][0]);
close(pipe2child[i][1]);
mm=m; nn=n;
while (nn>0) {
   sleep(1); // hack to overcome some system bug
   // write m and n
   clearbuf(buf);
   sprintf(buf,"%d %d ",mm,nn);
   len = strlen(buf);
   if (len != write(pipe2parent[i][1],buf,len))
perror("write to parent");
   // read p
   clearbuf(buf);
   while (read(pipe2child[i][0], buf, 1024)<= 0);
   mm=nn;
   sscanf(buf,"%d", &nn);
   printf("got it!\n");
}
printf("gcd(%d, %d)= %d\n", m,n,mm);
exit(0);
break;
   default: // parent
close(pipe2parent[i][1]);
close(pipe2child[i][0]);
}//switch
}//for i

////////////////////////////////////////////////////
//main parent loop
////////////////////////////////////////////////////
printf("parent loop\n");
while (countdown>0) {  // countdown=number of live children
  for (i=0; i< NumPairs; i++){
clearbuf(buf);
if (read(pipe2parent[i][0], buf, 1024)>0) {
   sscanf(buf,"%d %d", &m, &n);
   clearbuf(buf);
   sprintf(buf,"%d", m%n);
   if (m%n == 0) countdown--;
   len = strlen(buf);
   if (len != write(pipe2child[i][1],buf,len))
perror("write to child");
}
  }
}
printf("Parent die\n");
}//main
////////////////////////////////////////////////////
// END
////////////////////////////////////////////////////
```