CSCI 493.66 Unix Tools

with emphasis on string algorithms and Bioinformatics

Homework 5
Due 04/30/09

Saad Mneimneh
Computer Science
Hunter College of CUNY

Problem 1: The yes/no question
Consider the following C++ program to answer a yes/no question:

```
#include <iostream>
#include <termios.h>
#include <fcntl.h>

using std::cout;
using std::cin;

void question(const char * q) {
  cout<<q;
  cout.flush();
}

int response() {
  int ans=-1;
  int maxtry=3;
  while (1) {
    sleep(2);
    char a;
    cin>>a;
    if (a=='y') {
      ans=1;
      break;
    }
    if (a=='n') {
      ans=0;
      break;
    }
    cout<<"\a";
    cout.flush();
    cin.clear(); //in case read nothing, it sets error flag
    if (--maxtry==0)
      break;
  }
  return ans;
}
```

```
struct termios_flags {
  termios termsettings;
  int termflags;
};

termios_flags noncanontermmode() {
  //save old terminal setting
  termios oldtermsettings;
  tcgetattr(0, &oldtermsettings);

  //new terminal settings
  termios newtermsettings=oldtermsettings;
  newtermsettings.c_lflag &= ~ICANON; //no buffering
  newtermsettings.c_cc[VMIN] = 1; //get 1 char at a time
  newtermsettings.c_lflag &= ~ECHO; //no echo
  newtermsettings.c_iflag |= IUCLC;
  tcsetattr(0, TCSADRAIN, &newtermsettings);

  //make it non-blocking
  int oldtermflags=fcntl(0, F_GETFL);
  int newtermflags=oldtermflags|O_NDELAY;
  fcntl(0, F_SETFL, newtermflags);

  termios_flags tf;
  tf.termsettings=oldtermsettings;
  tf.termflags=oldtermflags;
  return tf;
}

void settermmode(termios_flags tf) {
  tcsetattr(0, TCSANOW, &tf.termsettings);
  fcntl(0, F_SETFL, tf.termflags);
}

int main() {
  termios_flags tf=noncanontermmode();
  question("yes or no (y/n)?");
  int ans=response();
  settermmode(tf);
  if (ans==-1)
    cout<<"question not answered\n";
}
```

(a) In the program above, cout.flush() is used twice. Explain the need for this function.

(b) The program implements a timeout feature that gives the user three chances to answer the question. This timeout implementation relies on two mechanisms:

- sleeping
- non-blocking input

Discuss the main problem of this implementation in terms of responsiveness to the user, and discuss the difference between non-blocking input and asynchronous input.

(c) Use asynchronous input and signals to solve this responsiveness problem.

(d) Research the use of VMIN and VTIME in the terminal settings to create a better timeout feature that is more responsive to the user. This will not require you to explicitly use non-blocking input and/or signals. Learn about the different ways VMIN and VTIME can be combined to create different behaviors.

(e) Protect the terminal settings by properly handling SIGINT and SIGQUIT. It is acceptable if you modify the code and use global variables.

Problem 2: Conway's game of life
In 1970, John Conway presented the game of life. In a two dimensional grid of cells, each cell is in one of two possible states: live or dead. Given this initial state, the cells evolve in each time step as follows:

- Live cell: a live cell with less than 2 live neighbors or more than 3 live neighbors dies
- Dead cell: a dead cell with exactly 3 live neighbors becomes alive.

neighbor	neighbor	neighbor
neighbor	cell	neighbor
neighbor	neighbor	neighbor

Here's a C++ implementation of these rules using vectors (you may use this code or a modification of it for this problem).

```
using std::vector;

void update(vector<vector<bool> >& grid) {

  /*
  assume grid has the following form:
   0000000
   0     0
   0     0
   0000000
  */

  int m=grid.size()-2;
  int n=grid[0].size()-2;

  //now access grid using rows 1 to m and colums 1 to n

  vector<vector<bool> > window;
  window.push_back(grid[0]);
  window.push_back(grid[1]);

  for (int i=1; i<=m; i++) { //row
    for (int j=1; j<=n; j++) { //col
      int count=
        window[0][j-1]+
        window[0][j]+
        window[0][j+1]+
        window[1][j-1]+
        window[1][j+1]+
        grid[i+1][j-1]+
        grid[i+1][j]+
        grid[i+1][j+1];
      if (grid[i][j])   //live
        if (count<2 || count>3)
          grid[i][j]=false;  //die
      if (!grid[i][j])  //dead
        if (count==3)
          grid[i][j]=true;   //alive
    }

    //slide window
    window[0]=window[1];
    window[1]=grid[i+1];
  }
}
```

(a) Implement the game of life using the curses library. Use timers and signals to animate, and asynchronous input to accept the following user commands:

- f: faster, set timer for shorter intervals
- s: slower, set timer for longer intervals
- q: quit, end the program

(b) If the timer is fast enough, the game may be interrupted in the middle of an update and, therefore, an update may start before a previous update terminates. This can put the grid in a wrong state. Use sigaction to make sure the handler is non-recursive.

(c) A safe way to handle a signal is to have a handler that only sets a global flag. The main loop repeatedly checks the flag and then handles the signal inside the normal program flow. Use this paradigm with the basic signal handling mechanism.

PS: For added safety, the global flag needs to be so small that the kernel can guarantee it will be set without interruption, i.e. in one machine cycle. This is usually an integer. The header files in C/C++ typedef the appropriate type for this to sig_atomic_t. Use that type for your flags.

Problem 3: Shared memory and mutual exclusion
Consider the following program with two processes:

```
#include <iostream>
#include <wait.h>

using std::cout;

void inc(int * p) {
  ++*p;
}
```

```
int main() {
  int count=0;
  if (fork()==0)  //now we have 2 processes
    inc(&count);
  else {
    wait(NULL);
    cout<<count<<''\n'';
  }
}
```

After the fork, the child process increments the count, while the parent process simply waits for the child to be done, and then outputs the value of count.

(a) Explain why this program outputs 0.

(b) The signal SIGCHLD is sent to a parent process when a child process is done. However, the default handling of SIGCHLD is to ignore it. Modify the implementation of the above program by replacing the wait function with a proper handling of SIGCHLD and the pause() command.

PS: this is not as trivial as it sounds. I will let you experiment.

(c) The mmap function can be used to map a portion of a file to an array. This is one way to create shared memory between a parent process and a child process. The following program illustrate the use of mmap.

```
#include <iostream>
#include <wait.h>
#include <sys/mman.h>
#include <fcntl.h>

using std::cout;

void inc(int * p) {
  ++*p;
}

int main() {
  int fd=open("shared",
            O_RDWR | O_CREAT,
            S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);

  write(fd, "\n", 1); //make sure file is not empty

  int * p=(int *)mmap(0,
                      sizeof(int),
                      PROT_READ | PROT_WRITE, MAP_SHARED,
                      fd,
                      0);
  *p=0;
  if (fork()==0)  //now we have 2 processes
    inc(p);
  else {
    wait(NULL);
    cout<<*p<<"\n";
  }
}
```

Run the program and interpret the result.

(d) Consider the following program:

```
#include <iostream>
#include <wait.h>
#include <sys/mman.h>
#include <fcntl.h>

using std::cout;

void inc(int * p) {
  int a=++*p;

  //something that takes time
  for (int i=0; i<10000000; i++);

  cout<<*p-a<<'' '';
  cout.flush();
}

int main() {
  int fd=open("shared",
              O_RDWR | O_CREAT,
              S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);

  write(fd, "\n", 1); //make sure file is not empty

  int * p=(int *)mmap(0,
                      sizeof(int),
                      PROT_READ | PROT_WRITE, MAP_SHARED,
                      fd,
                      0);
  *p=0;
  fork();  //now we have 2 processes
  while(true)
    inc(p);
}
```

Both the parent and the child process repeatedly call the function inc(). Therefore, we expect this program to output an infinite sequence of zeros. But this is not what we observe when we run the program. Explain why.

(e) Use the link and unlink system calls to make the inc() function atomic. Make sure the output is consistent with your expectation. Also, add the necessary code so that the program can run again if killed.

(f) The inc() function represents a critical region, i.e. we would like at most one process to be in that region at any point in time. This is known as mutual exclusion (processes exclude each other from being in the critical region). Look for the Peterson's algorithm to achieve mutual exclusion, and make inc() an atomic function using shared memory instead of the link and unlink system calls.

Problem 4: Z algorithm
Implement the Z algorithm described in class. Given a string $s = s_1 \dots s_n$, compute Z_k for $k = 2 \dots n$, where:

$$Z_k = \text{length of longest prefix of } s_k \dots s_n \text{ that is also a prefix of } s$$

```
$l = r = 0$
**for** each $k = 2 \dots n$
    **if** $k > r$
        compute $Z_k$ explicitly
        **if** $Z_k > 0$
            $l = k$
            $r = k + Z_k - 1$
    **else**
        $\beta = r - k + 1$
        $k' = k - l + 1$
        **if** $Z_{k'} < \beta$
            $Z_k = Z_{k'}$
        **else**
            compute $Z_k$ explicitly starting at positions $\beta + k'$ and $r + 1$
            $l = k$
            $r = k + Z_k - 1$
```