Operating Systems **CSI3310** ***Lab 2*** *Winter 2011*

Interprocess Communications with pipes
Java Threads

Part A – Interprocess Communications with Pipes.

Goal: To explore IPC with UNIX/Linux pipes.

Guideline

We will reuse the files from the lab 1.

Step 1: Start up the Site Linux virtual machine *sitedev* and login into the system. Copy the provided file *lab2a.tar* your working directory. As in Lab 1, extract the files from this archive using *tar –xvf lab2a.tar*. You will find the same files as in Lab 1, but with an additional file *filter* (the code for this executable is in the *code* directory).This is an executable file that is launched without parameters, reads its standard input, assuming it is receiving the output of *procmon* from the last labs and outputs only the lines in which the state of the monitored program changed.

Step 2: Enhance the C program mon.c developed in Lab 1 using the following guidelines:

1. Like *mon* from Lab 1, *mon2* uses one command line argument: the name M of the program it has to launch.
2. It will launch the program M (the program M does not take any parameters) and determine the PID of the process running the program.
3. It will launch *procmon* with the argument PID.
4. It will launch another process to run the provided program *filter* with no arguments.
5. The output of *procmon* should be sent to the input of *filter*, i.e. the *filter* will read from its standard input what *procmon* is writing to its (procmon's) standard output.
6. It will sleep for 20s, then it will terminate program M, will sleep 2 more seconds and will (try to) terminate *procmon* and *filter*.
7. A template for *mon2.c* has been provided to help you develop the code. Note that much of the *mon.c* program can be copied into *mon2.c*.

Step 3: Compile your C progam by entering `cc mon2.c –o mon2` on the command line. This invokes the C compiler (command cc) and tells it to compile file `mon2.c` and produce executable called `mon2`. If the compilation succeeded, an executable called `mon2` has been created in the current directory.

Step 4: Enter mon2 calcloop and observe. This launches the program mon2 you have just created, which in turn launches procmon to monitor calcloop's execution, and filter to output only the lines when the state of calcloop changes.

Step 5. You can experiment with sending signals to the calcloop process: When you launch mon calcloop:

- Learn the pid of calcloop (for example by entering ps –a | grep calcloop)
- Send it a stop signal by entering kill –s SIGSTOP pid, where pid is the pid of calcloop that you learned in the previous step.
- Send calcloop a continue signal by entering kill –s SIGCONT pid

Note that you will have to be reasonably fast, so that you manage to do this within 20 seconds calcloop is running.

Background you might need:

- The execution of a C program starts in procedure main() which has two arguments: integer argc containing the number of command line arguments and an argc-element array of strings argv, containing the command line arguments. By convention, argv[0] is the name of the program, argv[1] is its first argument.
- the fork() command creates a new process. Both the parent and the child continue as if they returned from a fork() call, however the parent gets as return value the PID of the child, while the child gets 0. Type man fork to read more about fork().
- The execl(path, arg1, …) command replaces the current process with the specified program launched with the provided arguments. Type man execl to learn about the exact meaning of its arguments.
  The code 'execl("calcloop", "calcloop", NULL)' will replace the current process with the *calcloop* program. The code 'execl("/bin/ls", "ls", "-l", NULL' will launch ls –l.
- You will need to convert the integer PID into a string to pass to procmon. In C you can do this using the *sprintf* function as in *sprintf(buf,"%d%,pid)* where *pid* is an *int* variable, and *buf* is a character array (*char buf[20];*) that will receive the character string to be passed to *procmon*.
- Function sleep() will cause the program to sleep for the specified number of seconds.
- Function kill(pid, sig) will send a signal sig to process pid.
  - have a look at signal.h to see the different signals
  - google is your friend, this search result on 'signal.h' might be helpful (there are many more)

  http://www.opengroup.org/onlinepubs/009695399/basedefs/signal.h.html
- A file descriptor is an integer number, that serves as a file handle, to identify an open file. The file descriptor is used with library functions and system calls such as *read()* and *write()* to have the OS operate on the corresponding file.

- The function *pipe(int *fd)* takes as an argument a pointer to an integer array (i.e. the name of an integer variable declared as array, i.e. int fd[2] , creates a pipe and sets the fd[0] to be the file descriptor used to access the read end of the pipe and sets fd[1] to be the file descriptor to be the write end of the pipe.
- *dup2(int newfd, int oldfd)* – duplicates the oldfd by the newfd and closes the oldfd. See http://mkssoftware.com/docs/man3/dup2.3.asp for more information. For example, the following program:

```
int main(int argc, char *argv[]) {
        int fd;
        printf(“Hello, world!”)
        fd = open(“outFile.dat”, “w”);
        if (fd != -1) dup2(fd, 1);
                printf(“Hello, world!”);
        close(fd);
}
```

will redirect the standard output of the program into the file outFile.dat, i.e. the first “Hello, world!” will go into the console, the second into the file “outFile.dat”.

- *read(int fd, char *buff, int bufSize)* – read from the file (or pipe) identified by the file descriptor fd bufSize characters into the buffer buff. Returns the number of bytes read, or -1 if error or 0 if the end of file has been reached (or the write end of the pipe has been closed and all data read).
- *write(int fd, char *buff, int buffSize)* – write into the file/pipe buffSize characters from the buffer buff
- *close(int fd)* – closes an open file descriptor
- perhaps this link might help you a bit with C: http://www.acm.uiuc.edu/webmonkeys/book/c_guide/

Consult the manual pages (by typing ‘man function_name’, i.e. ‘man fork’) and/or web resources for more information. The lecture slides from the chapter 3 and the exercise before the chapter 4 might be also quite helpful.

Part B – Java Threads.

Goals

1. Learn how to use threads – the Java Thread class.
2. Learn how to use thread pools – the Java Executors class.

.

Guideline

For this part of the lab, you will be working with an application that creates an illustration of the Mandelbrot set. The image is generated by painting a set of rectangles in a Windows frame. By using threads (and thread pools) you will be able to visually see the execution of threads. For information on the Mandelbrot set, consult the following site:
 http://www.ddewey.net/mandelbrot/

Step1 - Generating your first diagram: Unzip the java files from MandelBrot.zip. Compile the following java files using you favourite Java development environment.

MBGlobals.java	- provides a class with global definitions
MBPaint.java	- provides the class for computing and setting colors for pixels and filling a given square of the canvas.
MBCanvas.java	- provides the class that extends Canvas, a component that provides a drawing canvas.
MBFrame.java	- provides the class that extends JFrame, an object for creating a window (an MBCanvas object is added to the window frame).
MandelBrot.java	- The main application – reads arguments to set the global values in MBGlobals, and then creates an MBFrame object to display illustration of Mandelbrot set.

After compiling the java application, experiment with creating different views of the set using the following command:

```
java MandelBrot  <Upper x Coord> <Upper y Coord> <Real Dim> <Pixel Dim> <Fill Dim>
```

where

<Upper x Coord> and **<Upper y Coord>**: are the coordinates of the upper left hand corner of the diagram in terms of real values. To get an overall view of the set, use an upper corner of -2,2 with a dimension of 4 (see Real Dim).
<Real Dim> - The application displays a square diagram. This argument gives the dimension of the square in terms of real values. Thus, a value of 4 produces a diagram that will be 4 X 4 units. Using -2, 2, as the upper left hand corner coordinate thus produces a diagram that has as coordinates -2,2 (upper left hand corner), 2,2 (upper right hand corner), -2,-2 (lower right hand corner), and (2,-2) (lower right hand corner).

<Pixel Dim> - This is the dimension of the displayed canvas in terms of pixels. This argument controls the size of the window containing the diagram.
<Fill Dim> - This is the dimension of the square for filling in the pixels. In the provided code, recursion is used to create a number of MBPaint objects for filling in the various sections of the Canvas. The objective of the lab is to change the code such that threads are used to fill in these squares.

Try the following parameters:

java MandelBrot -2 2 4 600 50 - For a general view. Modify 600 to get different size windows.

Other interesting views can be generated using the following:

```
java MandelBrot  -2 1 1 600 50
java MandelBrot  -1 1 1 600 50
java MandelBrot  -2 0.5 1 600 50
```

Step 2 - Using Threads : Using the Java Thread Class, modify the provided java code to use threads for filling in the diagram. Try modifying the <Fill dim> argument to see the effect of filling in the diagram. Note that as you reduce this value, the number of threads used is increased.

Step 3 - Using a Thread pool: Java provides a class "Executors" for creating Thread pools. The static method "newFixedThreadPool(int sz)" can be used to create an object of class *ExecutorService* that manages a thread pool consisting of *sz* Threads. The object also manages a queue for queuing tasks when all threads are busy. To create such an object that uses a thread pool of 20 threads, use:

```
ExecutorService thpool = Executors.newFixedThreadPool(20);
```

Tasks can then be executed using:

```
thpool.execute(Runnable task)
```

Note that task is an object that implements the Runnable interface. Consult the Java documentation for details on the use of the Executors class.

Modify the Java code to use a thread pool for executing the MBCompute objects as runnable tasks. You should see the effect of limiting the number of tasks that can be created when comparing to the application in B). Try varying the size of the thread pool to see its effect on the execution.