Operating Systems | **CSI3131** ***Assignment 1*** | *Winter 2014*

Process/thread creation and inter-process communication

You must submit your assignment on-line with BlackBoard. This is the only method by which we accept assignment submissions. Do not send any assignment by email. We will not accept them. We are not able to enter a mark if the assignment is not submitted on BlackBoard! The deadline date is firm since you cannot submit an assignment passed the deadline. You are responsible for the proper submission of your assignments and you cannot appeal for having failed to do so. A mark of 0 will be assigned to any missing assignment.

Assignments must be done individually. Any team work, and any work copied from a source external to the student (including solutions of past year assignments) will be considered as an academic fraud and will be forwarded to the Faculty of Engineering for imposition of sanctions. Hence, if you are judged to be guilty of an academic fraud, you should expect, at the very least, to obtain an F for this course. Note that we will use sophisticated software to compare your assignments (with other student's and with other sources...). This implies that you must take all the appropriate measures to make sure that others cannot copy your assignment (hence, do not leave your workstation unattended).

Goals

Practise:

1. process creation in Linux using fork() and exec() system calls
2. thread creation in Linux with the pthread API
3. inter-process communication using pipes

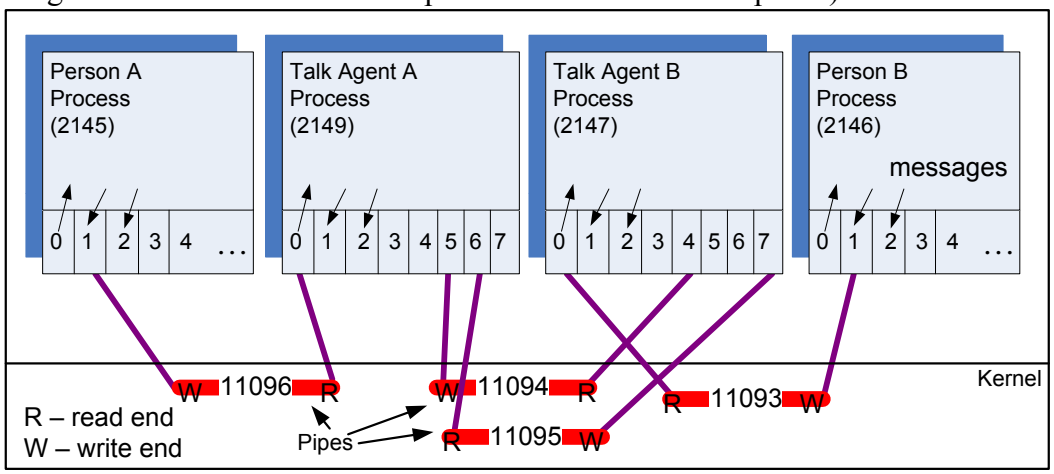
Posted: Jan 20 2014
Due: Feb 2 2014 (midnight)

Description (Please read the complete assignment document before starting.)

Two programs have been provided: `persons.c` and `talkagent.c`. Your task shall be to complete these two programs such that the execution of the program `persons` creates four processes as follows:

1. Person A process: this is the original process invoked when `persons` is run. It is responsible for creating the other three processes after which is simulates Person A that exchanges messages with Person B (simulated by the Person B process).
2. Person B process: this is a child process of the Person A process that simulates Person B who exchanges messages with Person A.
3. Talk Agent A process: This process (that runs the `talkagent` program) provides a talk agent service that services the Person A process. It shall use one thread to read text messages from its standard input which is then send to a communications channel to the Talk Agent B process. A second thread is used to send to its standard output any text messages received from the Talk Agent B process over a communications channel.
4. Talk Agent B process: This process (that runs the `talkagent` program) provides a talk agent service that services the Person B process. It shall use one thread to read text messages from its standard input which is then send to a communications channel to the Talk Agent A process. A second thread is used to send any text messages received from the Talk Agent A process over a communications channel to its standard output.

The `persons` program is designed to test the `talkagent` program by simulating two persons that exchange a sequence of text messages. Four pipes are used as shown below: two pipes allow the Person A and Person B processes to send messages to the standard inputs of the Talk Agent A and Talk Agent B processes respectively. Two other pipes are used as the communication channel between the talk agents. The Person A process that spawns the Talk Agent processes creates the pipes and passes the file descriptors of the communication channel pipes to the `talkagent` programs (see section To complete the assignment and comments in the provided source code templates)..

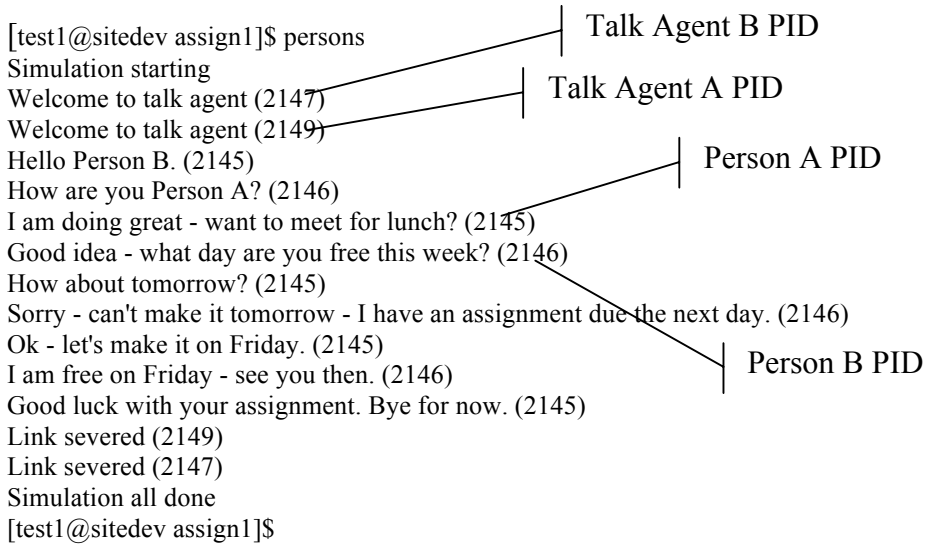


The pipe identifiers and process identifiers (PIDS) shown in the above diagram are specific to a run and correspond to the identifiers shown in the output of item 4 in the section "Background Information". All standard error file descriptors (2) and the standard

outputs of the Talk Agent processes are connected to the terminal. The path of messages from Person B process is shown (a similar path is taken for messages from the Person A process).

To complete the assignment:

1. Start with the provided files `persons.c` and `talkagent.c`. Complete the documentation in each file to indicate your name and student number. Take the time to document well your code. Consider taking the time to design your code before you start coding.
2. Complete the following functions:
   a. persons.c: doPersonB(), setupTalkAgentA(), setupTalkAgentB(), doPersonA() (also complete main to have these functions run properly)
   b. talkagent.c: createThreads()
3. The programs should be compiled using the commands "`cc -o persons persons.c`" and "`cc -o talkagent talkagent.c -lpthreads`" (note that for talkagent compilation, the pthreads library must be explicitly specified). The file `makefile` has been provided to compile both files – simply type in the command `make`.
4. To submit the assignment, upload the files *persons.c* and *talkagent.c*. Do not forget to click on the submit button after (and only after) you have uploaded the file.
5. A word of caution, for debugging the programs, if you wish to write messages to the terminal, you should write to the standard error using the following library call: `fprintf(stderr,"message string\n")` as the standard input and standard output are to be modified.
6. See point 4 in "Background Information" for hints on how to observe processes/threads and pipes to help debug your programs.
7. When `persons` is run, the following output should appear on your screen (Note that PIDs shall be specific to your execution).



Background information:

1. An open file descriptor is an integer, used as a handle to identify an open file. Such descriptors are used in library functions or system calls such as `read()` and `write()` to perform I/O operations.

2. In Unix/Linux, each process has by default three open file descriptors:
    a. Standard input (file descriptor 0, i.e. read(0,buf, 4) reads 4 characters from the standard input to the buffer buf). Typically, the standard input for a program launched from the command line is the keyboard input.
    b. Standard output (file descriptor 1).
    c. Standard error (file descriptor 2).
    d. When a command is run from the shell, the standard input, standard output and standard error are connected to the shell tty (terminal). So reading the standard input reads from the keyboard and writing to the standard output or standard error writes to the display.
    e. Note that many library functions used these file descriptors by default. For example *printf("String")* writes "String" to the standard output.
    f. From the shell it is possible to connect the standard output from one process to the standard output to another process using the pipe character "|". For example, the command "who | wc" connects the standard output from the who process to the standard input of the wc process such that any data written to the standard output by the who process is written (via a pipe) to the standard input of the wc process.

3. You will need the following C library functions:
    a. fork() – should be familiar from lectures
    b. pthread_create() – should be familiar from lectures
    c. pipe()
        - should be familiar from lectures
        - note that multiple process can be attached to each end of the pipes, which means that a pipe is maintained until no processes are connected at either end of the pipe
    d. execvp(const char * program, const char *args[]) (or execlp)
        - replaces the current process with the program from the file specified in the first argument
        - the second argument is a NULL terminated array of strings representing the command line arguments
        - by convention, args[0] is the file name of the file to be executed
    e. execlp(const char * program, const char *arg1, const char *arg2, … NULL)
        - replaces the current process with the program from the file specified in the first argument
        - the second argument subsequent arguments are strings representing the command line arguments.
        - by convention, arg1 is the file name of the file to be executed

f. 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".

g. 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).
h. write(int fd, char *buff, int bufSize) – write into the file/pipe bufSize characters from the buffer buff
i. close(int fd) – closes an open file descriptor
j. printf(): You may want to use the printf() function to format output. This function writes to the standard output (fd 1). But be careful since this function buffers output and does not write immediately to the standard output. To force and immediate write, used fflush(stdout). Alternatively, you may used sprintf(), to format the an output into a buffer and use write() to write to the standard output.
k. fprintf(): this is a version of *printf()* that provides the means to specify where output should be send. Use it to write to the standard error with *fprintf(stderr,"a message", arg, arg,...)*. This function is useful for debugging as it will write to the terminal in processes where the standard output has been redirected to a pipe.
l. getpid(): this function returns the PID of the current process. It is useful in creating messages printed on the screen to identify the source of the message.
m. Consult the manual pages (by typing 'man function_name', i.e. 'man fork') and/or web resources for more information.

4. Here is a hint at how you can observe the connection of processes to pipes
    a. Insert long delays using the standard library function `sleep` (e.g. `sleep(300)`) to allow observation of processes, threads and pipes at different points in the execution of the programs (particularly in `persons.c`).
    b. To see the processes and threads created using the command "`ps -Hmu test1`" (replace test1 with your user name if you are using one of the SITE Linux platforms). The option H has `ps` print out a tree of processes/threads and the option m includes all threads. In fact, Linux treats all processes and threads as tasks assigning each a PID. See below for expected output.

```
[test1@sitedev assign1]$ ps -Hmu test1
 PID TTY          TIME CMD
1114 ?        00:00:56 sshd
1115 pts/0    00:00:01   bash
2103 pts/0    00:00:00     vim
2112 pts/1    00:00:00   bash
2145 pts/1    00:00:00     persons
2146 pts/1    00:00:00       persons
2147 pts/1    00:00:00       talkagent
2148 pts/1    00:00:00         talkagent
2150 pts/1    00:00:00         talkagent
2149 pts/1    00:00:00       talkagent
2151 pts/1    00:00:00         talkagent
2152 pts/1    00:00:00         talkagent
2153 pts/1    00:00:00     ps
1833 tty1     00:00:00 bash
```

Person A process that forks three processes

Person B process forked by Person A process 2145

2148 and 2150 are threads created by 2147 (using pthreads)

Talk Agent B process

Talk Agent A process

See the next page for a hint on observing file descriptors and pipes.

c. To see how pipes and standard input, standard output, and standard error are set up for the various processes, use the command "`ls -l /proc/xxx/fd`" where `xxx` is replaced with the PID of a process. This will display how the various file descriptors of the identified process are connected. See below for expected output from the programs of this assignment.

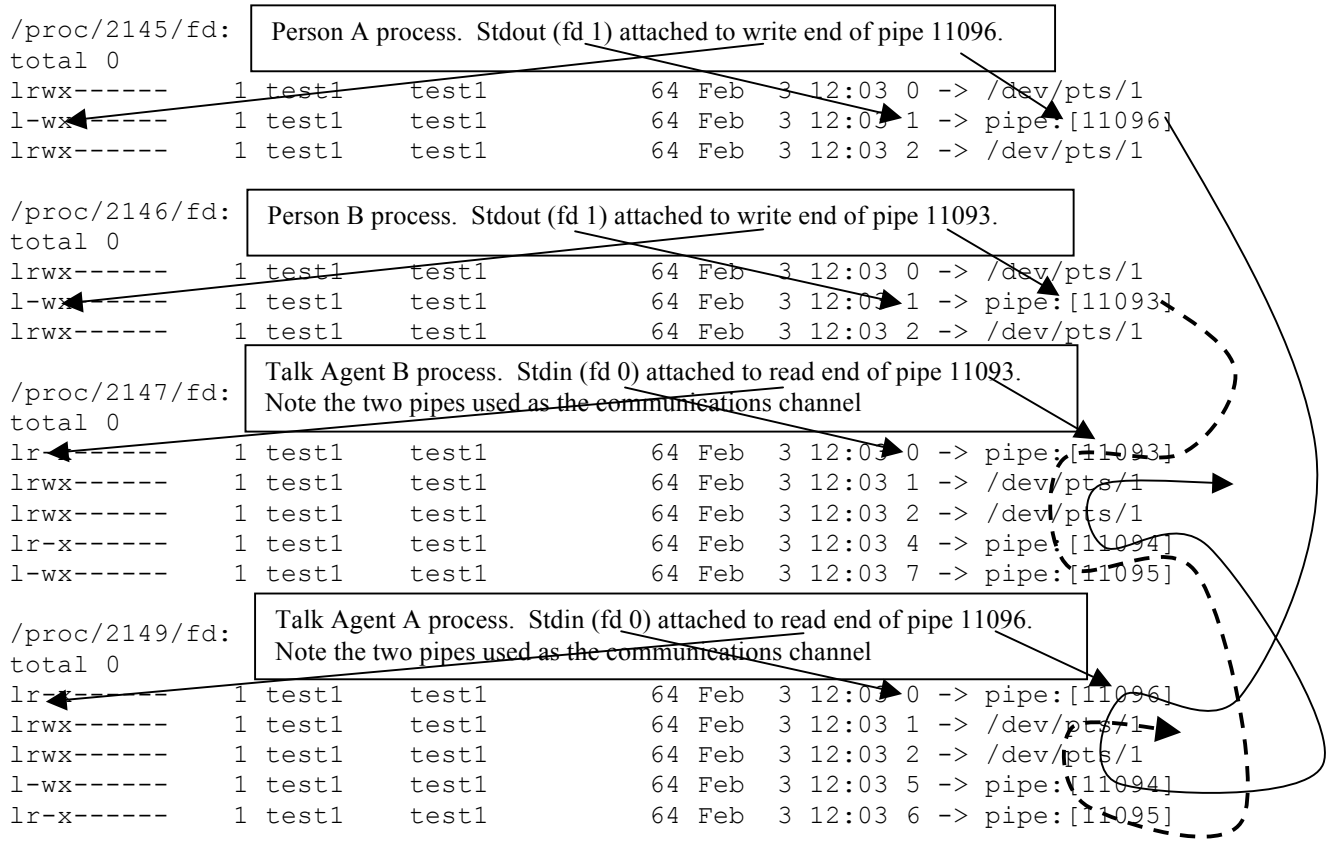
```
[test1@sitedev]$ ls -l /proc/2145/fd /proc/2149/fd /proc/2147/fd /proc/2146/fd

/proc/2145/fd:
total 0
lrwx------    1 test1    test1           64 Feb  3 12:03 0 -> /dev/pts/1
l-wx------    1 test1    test1           64 Feb  3 12:03 1 -> pipe:[11096]
lrwx------    1 test1    test1           64 Feb  3 12:03 2 -> /dev/pts/1

/proc/2146/fd:
total 0
lrwx------    1 test1    test1           64 Feb  3 12:03 0 -> /dev/pts/1
l-wx------    1 test1    test1           64 Feb  3 12:03 1 -> pipe:[11093]
lrwx------    1 test1    test1           64 Feb  3 12:03 2 -> /dev/pts/1

/proc/2147/fd:
total 0
lr-x------    1 test1    test1           64 Feb  3 12:03 0 -> pipe:[11093]
lrwx------    1 test1    test1           64 Feb  3 12:03 1 -> /dev/pts/1
lrwx------    1 test1    test1           64 Feb  3 12:03 2 -> /dev/pts/1
lr-x------    1 test1    test1           64 Feb  3 12:03 4 -> pipe:[11094]
l-wx------    1 test1    test1           64 Feb  3 12:03 7 -> pipe:[11095]

/proc/2149/fd:
total 0
lr-x------    1 test1    test1           64 Feb  3 12:03 0 -> pipe:[11096]
lrwx------    1 test1    test1           64 Feb  3 12:03 1 -> /dev/pts/1
lrwx------    1 test1    test1           64 Feb  3 12:03 2 -> /dev/pts/1
l-wx------    1 test1    test1           64 Feb  3 12:03 5 -> pipe:[11094]
lr-x------    1 test1    test1           64 Feb  3 12:03 6 -> pipe:[11095]
```



Notice that the Person A process messages written to the standard output traverses pipe 11096 to the Talk Agent A process where the message is sent to the Talk Agent B process who then writes the message to its standard output. A similar path with different pipes is taken by messages leaving the Person B process. Notice that both talk agents have their standard output attached to /dev/pts/1 that corresponds to a terminal (actually a pseudo-terminal connected to an ssh client). So messages written to the standard output by the Talk Agent processes shall appear on the terminal.