CS 4352
Project 2: Development of "At" utility
Due: 10/13/14 at 11:59 pm

Problem: There is a UNIX utility called, "at". It is used to schedule a program for delayed execution when it may not be convenient for the user to interact with the system at odd hours of the day. See its man page. The purpose of this project is to develop a similar utility, which we name "At" here. The syntax of "At" is as follows:

At time command-string

There are two ways to specify the time:

1) @hh:mm:ss (exact time of execution today)
2) +hh:mm:ss (time to lapse before execution)

As expected, 0 <= hh < 24, 0 <= mm, ss < 60.

The command-string is generally a single Shell command with its own options and arguments. Consider the following two examples to understand its use:

At +00:00:30 ping banana //Execute ping command after 30 seconds

At @02:00:00 ls –i /tmp $HOME //execute 'ls' at 2 AM. If current time is already past 2 AM today, wait for 2 AM tomorrow.

If the command-string contains a Shell meta-character or environment variable, it will be processed by the shell as usual at the time of execution of At, but not by At program. Hence, you may assume that the command-string is free of shell meta-characters.

This project is divided into two stages. The summary of each stage is as follows:
Stage 1: Convert the time argument to seconds, save the command-string, and find the executable file of the command. (It does not execute command).
Stage 2: Create a new process, say At_cmd, which would sleep through the delay period and then execute the saved command string. At quits after creating At_cmd. This feature allows the user to log out, and yet be able to keep At_cmd alive for delayed execution of the submitted command-string.

It is a complex project, because its main goal is to teach a number of process management concepts as well as some C coding. A number of small C code files are stored in "~cs4352/tutorials" folder to help you with the C coding.

Both stages will be graded separately; "proj2.grading" file stored in "~cs4352/projects/proj2" contains grading details. Therefore, it is important that you implement features described for a stage in that stage only. The project development is further broken into a number of steps to improve testing.

Create a folder "proj2" in your "cs4352" folder and then implement this project in proj2 folder.

Stage1: The purpose of this stage is to process the command line of *At* and save the command-string. You may assume that here is no ordering error, i.e., the time argument is given first and then the command-string, but the time argument itself may contain errors.

The outline of main() in stage1 is as follows:

1. main( ) first calls a procedure, say process_arguments(argc, argv), to extract arguments.
2. Next it calls another procedure, say delay = get_time_compute_delay(time)), to compute the delay period in seconds from the time extracted as a character string.
3. Next it calls a library function pathfind(…) to determine the path of the executable file.
4. Finally, it sleep through the assigned time and exits.

Execution of the actual command-string is taken up in Stage 2.

```
main ( argc, argv)
{
  Process arguments;
  Compute delay;
  Determine path of the executable of the command.
  sleep (delay);
exit(0)
}
```

Development of stage1 is further divided into following steps.

Step1a: process_arguments() processes argv[ ]. It copies the time argument to a char string, and the rest of the command line as a vector of strings, declared say as, "char **arglist". For this purpose, it first counts the size of the command-string, uses malloc() to allocate an array of character pointers and then saves the arguments. For example, if the command line is "*At* +00:00:02 ls –l /home /tmp", the net result should be that arglist[0]="ls",…, arglist[2] = "/home", arglist[3] = "/tmp", and arglist[3] = '\0'.

Output: Process_arguments( ) should print all items extracted from the command line such as below.

```
Time = +00:00:02
Arglist[0] = "ls", arglist[1] = "-l", …
```

Step1b: This step implements a procedure, "delay=get_time_compute_delay(time)" . It first checks the format of the 'time' argument. If it is well formed, it converts the 'time' into seconds and prints its value. Otherwise, it prints an appropriate error message and returns -1 to the main().

Files "test_time.c" and "seconds.c" in *tutorials* folder contain some useful code to complete this procedure. "test_time.c" shows how to obtain the current time from "tm" structure, and "seconds.c" shows how to convert the current time, obtained using "date" command, into seconds. (Caution – although function call, time(0) returns the number of seconds since mid-night, you need to compute it yourself, because one of the goals of this project is to do learn string processing.)

Caution: Do not declare just 2-byte space to store hh, mm, or ss of the time argument. The reason is that a string ends with '\0'. For example, if you declare
"char MM[2];" to store minutes component, set MM[0]=time[4] and MM[1]=time[5], and then do 'atoi(MM)' to convert the character representation of minute part to integer, you will not get

the correct result, because ‘atoi()’ will not just process 2-byte MM, but keep scanning until it gets ‘\0’. Correct coding is: char MM[3]; MM[0]=time[4], MM[1]=time[5], MM[2]=’\0’, and then use atoi().

Test step1b using the following:

```
At +00:70:09 ls          //error - minutes exceeded 59
At +25:07:02 ls          //error – hour exceeded 24
At @00:1:25 ls           //error - minutes should be given as 2-digit number
At *00:00:02             //error – time argument should begin with ‘+’ or ‘@’ only
At +01:10:12 man ls      // correct input ; delay = 4212 seconds
At @00:00:02 man ls      //correct command line;
At +00:00:02 cat –n –b stage1.sh stage2.sh        //correct command line –check arglist
```

Note that @time gives the actual time of execution of the command-string. For example, if time argument is @00:00:01, the time may already be passed today (unless you started *At* dead at the mid-night). In this case, you should compute the number of seconds to lapse until 00:00:01 of the next day.

Output: Print meaningful informative error message (and not “error”), in case of errors. If there is no error in the time argument, print the delay period. For example, if you execute at 11:00:00 am, the output for the fifth test should be “delay= 3602 seconds” and this is what your program should print and not 3602 alone.

Step1c: This step verifies whether the command in the command-string is a valid command. It calls a library function, pathfind( ) to verify if an executable file corresponding to the command name exists on the user’s PATH. Follow the little example given in the man page of pathfind ( ). Do not forget to include the header “libgen.h” to your source file and use “-lgen” in the gcc command.

Test step1c with the following:

```
At +00:00:03 LS                  // Bad command name
At +00:00:03 ping bigsun         // correct command.
```

Output: Print the pathname if the executable is found on the path; Else, print appropriate error message that command is not found on PATH.

Stage1d: Add sleep() to main().
Test step 1d with the following:

```
At +00:00:05 date
At @00:00:01 date        //should wait too long, kill it.
```

Before submitting your programs, remove debugging statements you may have added on your own (but not the print statements suggested above to show the output).

Stage2: The purpose of this stage is to create a child process, denoted as “At_cmd” for execution of the given command-string.

Step2a: Copy stage1.c to stage2.c and reorganize it as the following.

```
main (argc, argv )
{
   Process arguments,
   compute delay, and
   verify path of the command.

   if ((At_cmd_id = fork()) == 0)  {print identities of the child; At_cmd (delay) } //child's portion
    else { print At_cmd_id; …} // print and exit
}

At_cmd (delay)              // procedure executed by the child
{ sleep (delay);
  exit(0)
}
```

In this step, At creates a child process, prints some information (described later), and exits. It is the child process which calls At_cmd() to sleep through the "delay" period. The logic of At_cmd( ) will evolve as you implement more features in later steps.

Note that the parent portion should not contain wait(0), because At is supposed to *exit* once the child is created.

When a user logins to a system, the login Shell opens three files, stdin, stdout, and stderr, and its children processes inherit them. Therefore, when *At* starts, these files are already opened and At_cmd inherits them from *At*. As a result, At_cmd would print the output of "date" on the same terminal (stdout), although the parent would have already completed. To verify this statement, as soon *At* completes, immediately run command, "ptree" before the delay period expires. You should see a line containing the process-id printed by the parent. It means that although *At* has completed, the child is not dead. If you like, you can kill the child process before it completes, by running "kill –9 At_cmd-id" (killing a process is not an offence).

Step2b: The purpose of this and the next step is to print a number of process identities. To determine what
At (the parent process) and At_cmd (the child process) portions should print, run my program for stage2. You would see that it prints output by giving appropriate labels such as "parent: At_cmd_id=" to tell what the output means and who printed it. Follow it to code your print statements.

Step2c: In step2a, the child process would die without completing the delay, if you log out before the end of the delay period. The reason is that At_cmd belongs to the process hierarchy which is headed by the login shell. It is therefore necessary to separate the child process and make it the head of its own process group so that when the user logs off and the login shell dies, the child would survive on its own. Run "man –s 2 getpgrp" to learn about process group-id. There is a companion functions setpgrp().

To test this step, open two windows to the same UNIX host, run *At* command on one and close the window. Run "ps-ef |grep At_cmd_id" on the other window to verify if the child is still alive.

Step2d: Irrespective of the time format, delayed execution should produce the same output as if the commands were executed immediately, i.e., there is no effect of delayed execution. Therefore, the environment at the time of execution should be the same that existed at the time of submission of the command-string. The purpose of this step is to verify whether the environment of a process changes after it is created. To do so, add getenv ("SHELL"); putenv ("SHELL=/usr/bin/sh"); in the parent's portion, and getenv("SHELL"); right after the sleep ( ) in the At_cmd( ). Label the output of two getenv( ) appropriately to show which of the two processes printed them.

Step2e: Finally, revise At_cmd( ) to complete stage 2.

1. Pass pathname, the arglist, and the delay period as arguments to At_cmd().
2. Add execv ( ) system call to run the given command_string.

Run the following to test stage2:
At +00:00:10 cal 2008
At +00:00:03 At +00:00:04 ping ficus //This command would not run unless "." (current directory) is on your PATH. Edit your .bashrc and add ":." at the end of line defining PATH. Does output of this command show two output of pathname, ./At and /usr/sbin/ping?

The logic step2e and step2a could be easily merged into one step, but the purpose is to emphasize that fork() and exec() are two separate process management concepts in UNIX**..**

Final testing: Your program should produce appropriate error message in case of following:

1) If time does not beging with + or @, it should print usage of At and quit.
2) If hh, mm, or ss, is out of bound, print message and quit.
3) Print delay, if argument is correct such as @00:00:01.
4) It should print pathname of command, if found, else print error message.
5) It uses dynamic memory allocation for arglist.
6) Print arglist.
7) Print parent and child process-id.
8) After resetting group process-id, prints the new id.
9) Prints old and new environment variable.
10) Produces correct output for step 2e tests.

Submission: Develop stage1.c and stage2.c in proj2 folder. Name the source and executable files of different stages appropriately such as At1.c and At1 for stage 1. Next remove all files except the source files and their executables from your "proj2" folder. Use the given script to submit proj2. Late completion penalty is 10% of your grade per day.

EXTRA CREDIT (30%)

Stage3: The <u>purpose of this stage is to teach you use of the signal handling</u>.

Step3a: At_cmd() uses sleep() for the entire delay period, but what if the user desires to change the delay period after starting *At*. A trivial solution is to kill At_cmd process and run At again with the new time argument, but obviously it is not a good use of resources. A better solution is that At_cmd process should set up an alarm and that if the user sends a wakeup call, it should get out of the sleep and change its sleep time. (Note that a UNIX process has no *hearing* ability and therefore it would not hear an audible alarm when one is generated!)

Replace sleep( ) by the following three lines in At_cmd( ).

```
signal (SIGALRM, sig_catcher); //Tell kernel to transfer to sig_catcher(), when alarm goes off
alarm (delay); //set an alarm to go off at the end of given delay period
pause ( ); //go to sleep until interrupted by alarm or some other signal
```

Code a procedure, sig_catcher ( ) (or whatever name you like). It has no argument. In this step, it just prints a dummy message, say "snooze". See "test_fork.c" to learn use of signal ( ). Note that the kernel would transfer the control automatically to Sig_catcher() when At_cmd() receives an alarm, and therefore there is no call statement in At_cmd for sig_catcher().

Run "At +00:00:10 date" to test this step. When the alarm goes off, At_cmd blocked due to pause() would get out of the blocked state. Before returning control to At_cmd, the OS will notice that user had set a signal catcher for SIGALRM. Hence, the control will be returned first to the sig_catcher(), which would print "snooze". The control would be transferred to At_cmd to the statement next to pause() and the command-string would be executed.

Run "ps –efl | grep At_cmd _id" from another window to check if At_cmd process is indeed asleep. You should see a line with "S" as the second item, which means that At_cmd is asleep (S is for sleeping).

Step3b: A process can catch selected signals and ignore other (but it cannot ignore all). To incorporate this feature, At_cmd( ) should handle two signals SIGALRM and SIGINT; it catches the first and ignores the second. SIGINT is generated when user hits control-\. To realize this step, add the following statement next to the other signal().

```
        signal (SIGINT, SIG_IGN); //tell kernel to  ignore SIGINT
```

Next you should test if At_cmd would catch SIGALRM and ignore SIGINT. There are two ways to test this part. Send the chosen signals manually, or expand your main() to send such signals automatically. This step describes and implementation of the later choice.

Code a procedure test_signals(). main() calls it just before exiting. Logic of test_signal() is as follows. {sleep(2); kill(At_cmd_id, SIGINT); sleep(2); kill(At_cmd_id, SIGALRM); sleep(2); kill(At_cmd_id, SIGALRM);}

At_cmd should not come out of pause() when SIGINT is sent, but it should come out of pause ( ), when SIGALRM is sent. sig_catcher() is executed automatically, which will print 'snooze' message. At_cmd will then execute the given command-line. The second SIGALRM will be wasted because At_cmd does not go back to sleep again.

To learn about signals, "man –s 3HEAD signal" (-s option is needed here to search section 3HEAD, since this section might not be on your MANPATH.).

Step3d: The purpose of this step is to make At_cmd go back to sleep for the remaining period after it receives a SIGALRM. For this reason, put the alarm() and pause() statements in a loop as shown below. Note that execv ( ) should be the last statement of At_cmd ( ). Why?

```
moredelay = delay;
while (moredelay > 0)
   {printf ("inside loop: moredelay=%d\n", moredelay); //This statement should be in this loop.
    alarm (moredelay);
    pause ( );
   }
```

Since the effect of signal() lasts through only one call, add "signal (SIGALRM, sig_catcher);" in the signal_catcher() for At_cmd to catch the same signal every time. Expand sig_catcher ( ) as per below:

```
sig_catcher ( )
{
Print "snooze";
Get the current time;
Call a function to convert it to number of seconds to lapse since the midnight.
moredelay = how much more to sleep;
signal (SIGALRM, sig_catcher);
}
```

Every time At_cmd receives a SIGALRM, it would come out of pause and execute sig_catcher ( ). sig_catcher ( ) would then obtain the current time, compute the remainder of the delay period, and reset SIGALRM. If (moredelay > 0), At_cmd will go back to sleep for the next SIGALRM.

To realize computation of moredelay, consider the following example. Let *At* be run at 10:00:00 to schedule a command to run at time 19:00:00. Let SIGALRM be sent at14:00:00 and 17:00:00. In this case, your program should compute moredelay = 32400 (=9*60*60) initially. Upon receiving the first alarm, sig_catcher should compute modedelay = 18000 (=5*60*60) and set moredelay = 7200 (=2*60*60) the second time and print it for purpose of debugging.

Since At_cmd does not call signal catcher, it can neither pass an argument nor can it receive any data from the catcher. Therefore, moredelay should be declared as a global variable.

To test this step:

```
At +00:00:30 date
```

Your output should contain two ‘snooze’ statements because the second SIGALRM sent by test_signal() is also caught.

Note that signal catchers are used for very brief processing, and therefore, any post processing of the application should be done as part of application code and not inside the signal catcher code.