Programming Project: watch

Introduction

For this assignment you will write a program that implements some of the functions of the **watch** feature of the Unix tcsh shell. After you get the basic program working, you will make it more efficient and think about some ideas that can make the program even more useful.

Sometimes you are waiting for one or more users to login to the system. There are lots of reasons why you might want to know when someone logs in. You might want to start a chat session with that user. You might be waiting for that person to login and download a file you need. The **who** command might be useful; it tells you which users are logged in right now. You could type 'who' every few minutes and see if the logname appeared on the list. That would be pretty tedious. The shell called tcsh has a built-in feature that watches for logins and logouts for a specified list of users. The tcsh manual page has all the details.

For this assignment, you will write a program that performs the basic functions of the watch feature. Invoke the watch command with a list of lognames, and it tells you, by checking the utmp file at regular intervals, when anyone on the list logs in or logs out of the system.

Functional Overview

Write a program called **watch** that takes as command line arguments a list of users you want the program to watch for. Every five minutes the program wakes up and checks the utmp file. It compares the list of active users it finds there to the list list of active users it found last time.

If it finds a user is not logged on but had been logged on before, it should tell you. If it finds a user is logged on now and had not been logged on before, it should tell you.

The interval of checking defaults to five minutes. If you prefer a different interval, you should be able to specify the interval on the command line.

Under Unix, a user may login to several terminals at the same time, so the program should only report when a user goes from no logins to one or more logins or when a user goes from one or more logins to no logins. If you do not make this restriction, each time a user opens a new terminal window on a graphics workstation, you will be notified.

Part I: Specific Details

The **watch** program you write must meet the following specifications:

[a] It takes one or more lognames as command line arguments. It watches for the comings and goings of the lognames specified on the command line. It does not report on users you do not specify.

[b] When the program starts, it prints the lognames of users on the given list who are currently logged in.

[c] It checks the utmp file every 300 seconds to see if anyone on the list of lognames has logged in or logged out. If the first command line argument is an integer, it uses that number as the number of seconds to sleep between checking the utmp file.

[d] The program reports when a user on the list logs in. A user is considered to have logged in if, when the utmp file was last checked, that user was not logged in anywhere, but now the user is logged in at one or more terminals.

[e] The program reports when a user on the list logs out. A user is considered to have logged out if, when the utmp file was last checked, that user was logged in, but now the user is not logged in at any terminals.

The program should produce output of the form (the ... indicates five-minute pauses):

```
% watch betsy happy maya fido king susie
betsy happy are currently logged in
 ...
happy logged out
susie logged in
 ...
happy king logged in
 ...
susie logged out
fido betsy logged in
 ...
```

If you invoke the program with

```
% watch 120 betsy happy maya fido king susie
```

then it checks for changes every 120 seconds.

In practice, one would run the program in the background by typing a command with a trailing ampersand:

```
% watch 120 betsy happy maya fido king susie &
```

[f] The program buffers disk access to the utmp file by using the functions in the file utmplib.c .

[g] The program exits when the person who launched the program is no longer logged in. (There are lots of ways of doing this part.)

Getting Started

Make a subdirectory for this project. Change into this directory and link to the sample files and reference material. Type

```
mkdir watch
cd watch
ln   -s   ~COURSE/hw/watch/*    .       <- that's a dot at the end
```

Read about how tcsh handles the watch feature by typing `man tcsh` and then typing `/watch` to search for the word "watch" . When you are done reading one section, you can type `/watch` again to look for the next reference.

Note that the version of watch you are asked to write differs from the tcsh version of watch.

The program **dumputmp.c** that appears in the watch directory will read the login info file record by record and print that info to the screen. Look at the program and look at the header file to see how it works. Study the data structure and see what it contains. Run the program with the command

```
dumputmp /var/adm/utmp
```

to see the stuff in the file. A LOGIN entry means the system prints the login: prompt, a USER process means a user has logged in and is running user commands at the terminal, and a DEAD process means the user process has died. Usually, a dead process on a line causes a new LOGIN process to appear.

Now you need to develop a system to read the file with an eye for particular lognames. The lognames will be given on the command line. Your program should print an error message if it gets no lognames as arguments.

Your program should start by reporting which among the named users are currently logged in.

Your program should then sleep for 300 seconds, or the number of seconds specified on the command line as the first argument. Type `man 3 sleep` to learn about the sleep function.

When your program wakes up, it should print out the list of users who have logged out since it last checked. It should then print out a list of users who have logged in since it last checked.

Your program should then go to sleep and repeat the loop endlessly. The program should only exit if it detects any errors with the utmp file or with printf().

Things to Think About

The same user can be logged in at several terminals at the same time. For example, if someone is using a graphical user interface, he or she can run several windows at the same time. Each terminal window is a separate login. Your program does not need to keep track of how many windows the user has open, just whether or not the user is logged in anywhere.

Decide on a data structure for storing information. Your program should be flexible enough to work even if the user specifies a *lot* of names on the command line. You could use an array with a big number of entries and complain if the user specifies more than that number on the command line (a bad idea) or you could use `malloc()` to allocate enough storage to handle the number of names specified (a better idea.)

What do you need to record about each user to determine when he or she logs in? How do you obtain that information, and how do you store it?

Usually you will run this program in the background. It will snooze there, waking up every so often, tell you what has happened, then go back to sleep. There is a problem with background processes; they stay around when you logout. Unless you kill them or program them to exit when they find you gone, they will continue to run until the system goes down.

There is good news, though. A program that writes to standard output can detect when you logout. On many systems, when you logout, the standard output is no longer owned by the user who launched the process. When the program uses printf() to write to standard output, printf() returns EOF. Write your program so it exits if printf() returns EOF. That will cause it to exit when you logout.

Using utmplib.c

The file `utmplib.c` contains functions that provide buffered disk access to the utmp file. You should use these functions for your program. To use your code with that file, you can compile the program with the command:

```
% gcc watch.c utmplib.c -o watch
```

You can also compile utmplib.c first and link the .o file with your code. If you know know to use make, set up a makefile. If you do not, ask someone.

Testing Your Program

You can check your program on all currently-logged-in users by typing

```
% watch 30 `users` &
```

which will check every 30 seconds for the status of all people logged in now. If you test it this way, do so on a busy machine like one of the regular student systems. The command **users** is a short version of **who**, it lists lognames only. By enclosing the command name in backward apostrophes, you tell the shell to use the output of the command as command-line arguments to watch.

To determine if **watch** is doing its job correctly, you can wait for it to report a logout or login. Say it reports that user smith has logged out or logged in. Look at the login history of smith with the command `last -3 smith` (where smith is an actual logname.)

To kill the watch job, type `ps` . This command will list your currently-running processes, then you should type `kill ##` where ## is the PID of the watch process.

Part II: Increasing Efficiency

The program, as written, checks the utmp file at regular intervals and processes all the data it finds there. What if the file has not been modified since the last time it was checked? There is not much point in opening the file and processing data that has not changed since the last time.

The **stat()** system call gets information about a file, including the modification time of the file. By checking the modification time of the utmp file, you can see if it has been modified since the last time you processed the data in the file.

Modify your program so it uses **stat()** to determine if it needs to examine and process the utmp file.

Using **stat()** is easy. Read the manual page for complete details, but the following code gets the modification time of a file:

```
#include  <sys/types.h>
#include  <sys/stat.h>

/*
 * returns mod time of named file or 0 if cannot stat the file
 */

time_t get_modtime( char *filename )
{
   struct stat infobuf;
   if ( stat( filename, &infobuf ) == -1 )
     return 0;
   return infobuf.st_mtime;
}
```

Part III: Thoughts for Future Enhancements

You do not have to write any code for this part of the assignment. Instead, you need to think about the questions and write up short, but useful answers to the following questions:

[a] The watch program you write checks for specified users on *any* terminals. The complete watch feature in tcsh allows you to specify lognames *and/or* terminal names. There are four combinations.

You can watch for user smith to login at the console. You can also watch for user smith to login at any terminal. Furthermore, you can watch for any user to login at a specified terminal. For example, you can ask to be notified when anyone logs in on ttyp3 .

What changes would you have to make to the data structures in your program to implement this additional flexibility? How would your code have to change? You do not need to specify low-level details, but you need to include an outline of the main issues and solutions.

[b] In many settings there are lots of machines a user can log into. For example, if the machine called host1 is busy, a user may log into host2 or host3 or host3 or any of several networked machines. Checking logins on the 'local' machine is almost useless in this kind of networked setting. We shall discuss network programming later in the course, so you are not expected to have a 'correct answer' to this problem. Nonetheless, use your imagination to think of some way to check for user smith on several machines. You may invent software ideas as long as they help solve the problem.

[c] People can evade your watch program if they time it right. If you check utmp every five minutes, someone could login right after you check, stay logged in for four minutes, then logout, and never be detected. What information would you need to be able to detect all logins? How could you get that information? Read about the `last` command to find the answers to these questions.

What to Hand in

Hand in the following:

[a] Source code to your program that implements parts I and II.
[b] Sample output to show it in action.
[c] Answers to the questions in part III.