UNIX System Programming
Lecture 7: UNIX Environment

● Outline
  ● Handling Program Arguments
  ● Environment Variables
  ● Time and Date Routines
  ● Temporary Files
  ● User and Host Information
  ● Process Priorities and Resource Limits

● Reference
  ● BLP: Chapter 4
Lecture 7: UNIX Environment

The getopt() Routine

- The `getopt()` routine parses command line arguments. Successive calls to `getopt()` return option characters.
- It can detect options of the form “-abc” or “-a -b -c”. It can parse lines with options that require arguments “-f data.txt” or “-f data.txt”.
- Note: There is a `getopt` program (`man 1 getopt`) in addition to the `getopt()` routine (`man 3 getopt`).
The getopt() Routine

- **getopt() permutes** argv[] so that all non-options are at the end.

- The **getopt** prototype is:

  ```c
  #include <unistd.h>
  int getopt(int argc, char *const argv[],
   const char *optstring);
  ```

- See **getopt.cpp** for an example.
Lecture 7: UNIX Environment

The Environment

- UNIX programs receive two collections of data from the process (the parent) that invokes it: the *arguments* and the *environment*.

- The environment is *copied* from the parent. It is not *shared* with the parent.

- A child process cannot change the environment of the parent process.
bash allows assignments to be made to variables in the child's environment:

$ OUTFILE=xxx.dat ./program

The environ variable provides direct access to the environment (see prenv.cpp).

It is usually easier to use the getenv() and putenv() routines. (see environ.cpp).
There are several routines for working with times and dates.

```c
#include <time.h>
time_t time(time_t *t);
struct tm *gmtime(const time_t *timep);
struct tm *localtime(const time_t *timep);
time_t mktime(struct tm *tm);
char *asctime(const struct tm *tm);
char *ctime(const time_t *timep);
```

Note the two data types, a `time_t` type and a `tm` structure.
Lecture 7: UNIX Environment
Date and Time Routines

- A `time_t` type holds the time (in seconds) since the *Epoch* 00:00:00 Jan 1, 1970.
- The `time()` routine returns the current time as a `time_t` type. (`stat()` returns access, mod and change times as `time_t` types.)
- `localtime()` converts a `time_t` value into a programmer-friendly `tm` structure. `mktime()` converts a `tm` structure into a computer friendly `time_t` value.
• Here's the definition of the `tm` structure from the `localtime` man page:

```c
struct tm {
    int tm_sec;    /* seconds */
    int tm_min;    /* minutes */
    int tm_hour;   /* hours */
    int tm_mday;   /* day of the month */
    int tm_mon;    /* month */
    int tm_year;   /* year */
    int tm_wday;   /* day of the week */
    int tm_yday;   /* day in the year */
    int tm_isdst;  /* daylight saving time */
};
```
• **asctime()** and **ctime()** can be used to convert a time type directly into a user friendly **text** string.

```c
    time_t now;
    struct tm *tm_now;
    now = time(NULL);        // time from epoch
    tm_now = localtime(&now); // time_t to tm
    cout << asctime(tm_now); // tm to human
    cout << ctime(&now);     // time_t to human
```

• Refer to the **whattime.cpp** example.
Lecture 7: UNIX Environment
The Y2.038K Problem

- On a 32-bit platform, `time_t` is a 32 bit signed integer. The clock will roll-over at GMT 03:14:07 on Tues, Jan 19, 2038 to GMT 20:45:52 on Fri, Dec 13 1901.

- On a 64-bit platform, `time_t` is a 64 bit signed integer. The clock rolls-over at 3:30 PM Sunday, Dec 4, 292,277,026,596. (Our sun is expected to run out of hydrogen in about 5,000,000,000 years.)
Lecture 7: UNIX Environment

The Y2.038K Problem

- On 32-bit platforms certain date calculations can run into problems due to overflow even now (see `avgtime.cpp`).

- Redefining `time_t` as a 64-bit type on 32-bit platforms would break too many existing binary applications. (Recompiling from source would work as long as no assumptions were made about `time_t` being a 32-bit type.)
If you expect your application to still be running on a 32-bit platform in 2038 you might want to use something other than a `time_t` time reference. (See the `libtai` Open Source library.)
Lecture 7: UNIX Environment

Temporary Files

- The `tmpnam()` and `tempnam()` routines generate unique file names. (They may not be unique when the file is opened later.)

- `tmpfile()` opens a temporary file with a guaranteed unique name, but returns a C FILE* stream.

- `mkstemp()` generates a unique name from a template, opens the file, and returns a file descriptor.
Lecture 7: UNIX Environment

Temporary Files

- **mkstemp()** is guaranteed to provide access to a file with a unique name.

```c++
// must be a char array and not char *
// last 6 letters must be XXXXXX
char nametemp[ ] = "\tmp/tmpXXXXXX";
int fd = mkstemp(nametemp);

cout << "Temp file name is "
    << nametemp << endl;

string output("Hello world!\n");
write(fd, output.c_str(), output.length());
```
Lecture 7: UNIX Environment

User Information

- The following routines return user info:
  - `getuid()`  // get real user id
  - `geteuid()` // get effective user id (setuid programs)
  - `getlogin()` // get real username
  - `cuserid()`  // get effective username
  - `getpwuid()` // get passwd struct from user id
  - `getpwnam()` // get passwd struct from username
  - `getpwent()` // get successive passwd entries

- See the man pages for details ...
The **passwd** struct contains:

```c
struct passwd {
    char    *pw_name;    /* user name */
    char    *pw_passwd;  /* user password */
    uid_t   pw_uid;      /* user id */
    gid_t   pw_gid;      /* group id */
    char    *pw_gecos;   /* real name */
    char    *pw_dir;     /* home directory */
    char    *pw_shell;   /* shell program */
};
```
• The following routines return information about the \textbf{machine}:

\begin{itemize}
  \item \texttt{gethostname()}: \texttt{get machine name (csserver)}
  \item \texttt{sethostname()}: \texttt{set machine name (root only)}
  \item \texttt{getdomainname()}: \texttt{get the domain (evansville.edu)}
  \item \texttt{setdomainname()}: \texttt{set the domain name (root only)}
  \item \texttt{uname()}: \texttt{get OS info (version, etc)}
  \item \texttt{gethostbyname()}: \texttt{get IP info (via DNS)}
\end{itemize}
The `syslog()` routine can be used to send a message to one of the log files (usually in `/var/log`). This facility is intended for use by system applications and utilities and not for normal user applications.

// Depending on syslogd configuration this may or may not appear in /var/log/messages
syslog(LOG_INFO, "Warning Will Robinson!!");
Processes have priorities between -20 (high) and +19 (low). The default priority is 0. A user can run a program at a lower priority using the `nice` command. You can change the priority of an already running process with `renice`. Only the root user can decrease a program's priority below 0.

`getpriority()` and `setpriority()` are used to determine and alter process priority.
Lecture 7: UNIX Environment Resource Limits

- Limits can be placed on process and user resources: (cpu time, memory, file size, number of open files, etc)
- The bash commands `ulimit -Sa` and `ulimit -Ha` will list the default soft and hard limits.
- Soft limits can be changed (up to the hard limit) by users. Users can only decrease the hard limit (but not below the soft limit).
Lecture 7: UNIX Environment

Resource Limits

- Only privileged processes (root or setuid processes) can increase the hard limit.
- `getrlimit()` and `setrlimit()` routines are used to get and set resource limits.
- `getrusage()` routines can be used to get resource usages.
- The administrator can control resource limits through the `/etc/security/limits.conf` configuration file (via PAM).
In Class Exercises

- Write a program that displays the GECOS field from the passwd entry corresponding to the effective user.
- Write a program to determine the soft and hard limits on the number of files a process may have open.
- Use the shell's `ulimit` command (help `ulimit`) to change the soft and hard limits on the number of files and then rerun your program.