Interacting with Unix

Getting the Process ID

- Synopsis

```
#include <unistd.h>
pid_t getpid(void);
```

- Example:

```
#include <stdio.h>
#include <unistd.h>
int main(){
   pid_t  n = getpid();
   printf("Process id is %d\n", n);
}
```

Getting and Changing the Current Directory

- SYNOPSIS

```
#include <unistd.h>
char *getcwd(char *buf, size_t size);
int chdir(const char *path);
```

Example

```
#include <stdio.h>
#include <unistd.h>
int main(){
    char str[1000];
    char*p=getcwd(str,1000);
    if(p!=str){
      printf("Could not get cwd!");
      exit(1);
    }
    printf("cwd is %s\n", str);
    chdir("/usr/bin");
    printf("cwd is now %s\n",getcwd(str,1000));
}
```

Getting the Current System Time (1)

- There are a number of library functions relating to time in C. Their prototypes are found in <time.h>.
- Two data types are the most important for those functions:
  - time_t /* Typically same as long. It is the number of seconds since epoch: 00:00:00 UTC, January 1, 1970 */
  - struct tm /* See next slide. */
- Can go the microsecond or nanosecond accuracy with other structures and functions.

Getting the Current System Time (2)

- struct tm contains time information broken down:

```
struct tm{
    int tm_sec; // seconds [0,61]
    int tm_min; // minutes [0,59]
    int tm_hour; // hour [0,23]
    int tm_mday; // day of month [1,31]
    int tm_mon; // month of year [0,11]
    int tm_year; // years since 1900
    int tm_wday; // day of week [0,6] (Sunday = 0)
    int tm_yday; // day of year [0,365]
    int tm_isdst; // daylight savings flag
}
```

Getting the Current System Time (3)

- Most of the time, you only need the following two functions, but there are others:

```
#include <time.h>
time_t time(time_t * time);
struct tm *localtime(const time_t * time);
```

An Example and a Question

```
#include <stdio.h>
#include <time.h>
int main(){
    time_t t = time(NULL);
    struct tm * p = localtime(&t);
    if( p->tm_year >= 102 ){
        printf("Trial version expired!\n");
        exit(0);
    }
    return 0;  /* Question: why don’t we free(p)? */
}
```

The Answer

- localtime() looks like the following:

```
struct tm * localtime(const time_t * time){
    static struct tm t;
    t.tm_year = ……;
    ……
    return & t;
}
```

- Suggestion: Use man localtime or look up a manual page to find out the exact behavior of a function.

Calling a Command from a C Program

- In a C program, we can invoke a subshell and let it run a Unix command using the system() function:

  ```
  #include <stdlib.h>
  int system(const char *);
  ```

- Example:

  ```
  #include <stdio.h>
  #include <stdlib.h>
  int main() {
      int k;
      printf("Files in Directory are: \n");
      k = system("ls -l");
      printf("%d is returned.\n", k);
      return k;
  }
  ```

Piping to and from Other Programs (1)

- A command executed by the system() function uses the same standard input and output as the calling program.
- Sometimes, we want to pipe output from the calling program to the new command, or pipe input from the new command to the calling program.
- This can be done using the popen() function:

```
#include <stdio.h>
FILE *popen(const char *command, const char *mode);
int pclose(FILE *fp);
```

- If mode is "r", popen() returns a file pointer that can be used to read the standard output of command.
- If mode is "w", popen() returns a file pointer that can be used to write to the standard input of command.
- popen() returns NULL on error.

Piping to and from Other Programs (2)

```
#include<stdio.h>
int main() {
   FILE *fp;
   char buffer[100];
   if ((fp = popen("ls -l", "r")) != NULL) {
     while(fgets(buffer, 100, fp) != NULL) {
        printf("Line from ls:\n");
        printf("  %s\n", buffer);
      }
     pclose(fp);
   }
   return 0;
}
```

execl (1)

- The system() function returns control to the program it was called from.
  - Immediately, if you background the command with an &.
  - When the command completes, otherwise.
- Occasionally, you do not want to get the control back.
  - For example, when your program is a loader of another program.
- execl() is suitable for such purposes. It loads the new program and uses it to *replace* the current process.

execl (2)

- Synopsis

```
#include <unistd.h>
int execl(const char *path, const  char  *arg0,
   ...,  const char *argn, char * /*NULL*/);
```

- path is the pathname of the executable file.
- arg0 should be the same as path or the filename.
- arg1 to argn are the actual arguments
- The last parameter must be NULL (or 0).

Example

```
#include <stdio.h>
#include <unistd.h>
int main() {
    printf("Files in Directory are:\n");
    execl("/bin/ls", "ls”, "-l", NULL);
    printf("This line should not be printed out!\n");
    return 0;
}
```

- All statements after execl() will not be executed.

Multi-process Programming

- With a Unix system, you can write programs that run several processes in parallel.

- For example, a web-server can invoke child processes, each of which responses to the requests from a different web-browser.

- We will not get into the detail of this (see CS305a/b). But, we tell you the first step of multi-process programming, so you know where to start.

The fork() Function (1)

- Synopsis

  ```
  #include <unistd.h>
  pid_t fork()
  ```

- The fork() function creates a new process. The new process (child process) is an exact copy of the calling process (parent process).
- The only difference between the child and parent processes is the return value of fork().
  - Child process gets 0 if fork is successful.
  - Parent gets process id of child or -1 on errors.
- You can do different things depending on whether it is a child or a parent process.

The fork() Function (2)

```
#include <stdio.h>
#include <unistd.h>
int main(){
    int pid; /* Process identifier */
    pid = fork();
    if ( pid < 0 ) {
            printf("Cannot fork!!\n"); exit(1);
    } else if ( pid == 0 ) {
            /* Child process */ ......
    } else {
            /* Parent process */ ....
    }
}
```