628 Lecture Notes Week 4

1 Topics

- I/O Redirection
- Notes on Lab 4
- Introduction to Threads

Review

Memory spaces

```
#include <stdlib.h>
#include <string.h>
int g1;
int g2;
char * gcp1;
char * gcp2;

void foo(int p, int q) {
    printf("Parameter p address:   %p\n", &p);
    printf("Parameter q address:   %p\n", &q);
}

/*
 *
 */
int main(int argc, char** argv) {
    int local_1;
    int local_2;

    gcp1 = "abc";
    printf("Function main address:          %08X\n", main);
    printf("Function foo address:           %08X\n", foo);

    printf("Global g1 address:            %p\n", &g1);
    printf("Global g2 address:            %p\n", &g2);
    printf("Global gcp1 address:          %p\n",  &gcp1);
    printf("Global gcp2 address:          %p\n",  &gcp2);
```

```
    printf("Local local_1 address:       %p\n", &local_1);
    printf("Local local_2 address:       %p\n", &local_2);
    foo(local_1, 5);

    printf("String 'abc'  address:  %p\n", gcp1);
    gcp2 = (char *) malloc(strlen(gcp1) + 1);
    strcpy(gcp2, gcp1);
    printf("Dynamically allocated copied string address: %p\n", gcp2);


    *gcp2 = 'x';
    printf("%s\n", gcp2);

//    *gcp1 = 'x';
    printf("Goodbye\n");

    return (EXIT_SUCCESS);
}
```

You can download this file here

2 I/O Redirection

(Note: a more complete examination of this topic will have to wait until we discuss file systems and device drivers later in the course.)

stdin, stdout and stderr

- Any OS that allows C programming opens 3 “files”, *stdin, stdout, stderr* before the program starts.
- The OS maintains a table with an entry for each open file.
- Entry 0 is *stdin.*
- Entry 1 is *stdout*
- Entry 2 is *stderr.*
- Note that a “file” may be a *device* (such as a window on the screen or a keyboard or a printer) as well as (of course) an “ordinary file”.
- By default, *stdin* is the keyboard and both *stdout* and *stderr* are the terminal (window).

A note about C functions that are OS calls versus “user friendly” functions

- An operating system is basically defined by a set of functions that implement OS services.

- These functions usually operate in supervisor mode whereas the "user friendly" functions work in user mode.
- Any programmer can add new user mode functions; only the OS kernel writers can write functions that operate in supervisor mode.
- OS (or kernel) functions tend to be very low level and sometimes difficult to use.
- For example, "printf" is a user-mode function that is "easy" to use.
- But the actual output is done by the much lower-level "write" system call where the bytes to write and an index into the file descriptor table are required. (Of course, the user-level writer of the printf function does invoke "write" but all the lower-level details are hidden from the user.)
- By the way, although *stderr* and *stdout* are both by default connected to the terminal, they do ***not*** behave identically. In particular, stdout is *buffered* whereas stderr is *unbuffered.* (i.e. writing a character to stderr appears immediately whereas a byte to stdout is only written once it is "flushed" either because the buffer is full, a newline is output, the program exits or the programmer explicitly flushes it with "fflush(stdout);"
- In Unix, the "manual" is divided into 8 sections. Section 1 describes user commands (ls, mkdir, or whatever you type to the shell), section 2 describes operating system calls (such as open, close, dup, read, write, fork, exec, etc.) and section 3 describes the functions most programmers use (such as getchar, printf, fopen). Note: it is unlikely that you have ever written a C program before this course that used any Section 2 functions!

Here's a simple example illustrating that stderr and stdout do not behave the same way:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
    int i;
    char ch = 'a';
    char ch2 = 'A';
    for(i = 0; i < 20; i++) {
        putc(ch + i, stdout);
        if(i == 10) {
            fflush(stdout);
        }
        putc(ch2 + i, stderr);
    }
    return (EXIT_SUCCESS);
}
```

Redirection and piping

Some useful functions (not supervisor mode):

- `FILE * fopen(const char * filename, char * mode)`: opens a file returning a pointer to its "handle"; mode can be "r" (read), "w" (write), "rw" (red/write), etc
- `int fileno(FILE * f)`: Get the file descriptor (index into file table) for this file. (OS calls deal with file descriptors, not handles.)

Some useful functions (OS calls or supervisor mode):

- `int open(const char path, int flags, int mode)`:
- `close(int fd)`: Close the file.
- `dup(int fd)`: Duplicates the file descriptor at the lowest numbered unused slot in the table of open files.
- `dup2(int oldfd, int newfd)`: Duplicates the oldfd to newfd (closing newfd first if that slot was in use.)

Here's a simple example of redirecting stderr (file descriptor number 2):

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
    int i;
    char ch = 'a';
    char ch2 = 'A';
    FILE * f = fopen("junk", "w");
    dup2(fileno(f), 2);
    for(i = 0; i < 20; i++) {
        putc(ch + i, stdout);
        if(i == 10) {
            fflush(stdout);
        }
        putc(ch2 + i, stderr);
    }
    return (EXIT_SUCCESS);
}
```

Here's an example of piping two commands:

```
#include <stdlib.h>
#include <stdio.h>
```

```
  char *cmd1[] = {"ls", 0}; //Output goes to pipe input
  char *cmd2[] = {"/usr/bin/tr", "a-z", "A-Z", 0};//Input comes from pipe
output

  int
  main(int argc, char **argv) {
      int pid, status;
      int fd[2];

      pipe(fd); //Create a pipe; fd[0] is input, fd[1] is output

      switch (pid = fork()) {

          case 0: /* child */

              /* Using close/dup */
              close(0);
              dup(fd[0]);
              /* Preferred: use dup2*/
  //            dup2(fd[0], 0); //pipe input is now stdin for child
              close(fd[1]);   //child does not use pipe output
              execvp(cmd2[0], cmd2); //child executes "tr" command
              perror(cmd2[0]); //SHOULD NOT GET HERE!
              exit(1);

          default: /* parent */
              dup2(fd[1], 1);
              close(fd[0]); /* the parent does not need this end of the pipe */
              execvp(cmd1[0], cmd1);
              perror(cmd1[0]);  //SHOULD NOT GET HERE (exec failure)
              break;

          case -1://SHOULD NOT GET HERE (indicates fork failure)
              perror("fork");
              exit(1);
      }
      exit(0);
  }
```

You can download this file here.

3 Notes on Lab 4

4 An introduction to Threads

Summary of process characteristics

- A process can be *Running*, *Blocked* or *Ready.*
- A system call may result in the process being *Blocked.*
- Each process has an entry on the *Process Table*.
- Each process has its own memory spaces (segments).
- Only one process at a time can be running (assuming a single core CPU).
- Processes are usually independent of each other and rarely need to communicate using some form of *Inter Process Communication (IPC)*.
- Creating a process (via fork/exec, for example) requires a lot of work from the kernel (i.e. memory has to be allocated, the text segment has to be read from disk, a process table entry filled in, etc.)

Overview of Threads

- A Thread, like a process, can be *Running*, eligible to run (*Ready*), or waiting for something (*Blocked*) .
- Only one Thread at a time can be running (like a process).
- Each Thread has an entry in the *Thread Table* which indicates its state (Run, Blocked, Ready), the values the machine registers should have the next time it runs, which process created it, its priority, etc.
- ***However,*** a Thread (unlike a process) does ***not*** have its own memory segments (except for its stack; each thread has its own stack.)
- If a process has multiple Threads and the running thread *blocks* to wait for (say) some I/O, the process itself is not blocked and other Threads may run.
- Threads share the same global data and functions. Any Thread can invoke any function at any time and read or modify any global variable.
- Consequently, IPC for Threads is almost trivial. ***But it has its dangers!*** (A topic we shall examine in detail next week.)
- Creating a new Thread is much less CPU-intensive than creating a process; memory for the Thread's stack has to be allocated and a Thread Table entry filled in with the initial values of the new Thread's registers.

Why Threads are useful

First, a classic example of using Threads is any application that has a Graphical User Interface (GUI)

such as a word processor.

- If a word processor were a process with only a single thread, the whole thing would block whenever it was waiting for the next user keyboard entry.
- However, if a high priority thread is created that detects user input (keyboard, mouse, etc.), it will be blocked most of the time. A "work thread" of lower priority can continue working to do things like page formatting, grammar verification, saving to disk periodically, etc.

More generally, even for CPU-intensive tasks:

- Any task that has components that can execute in parallel can benefit from placing them in separate concurrent threads.
- Modern CPUs have multiple "cores" (even smart phones have up to 8 cores; high-end servers can have hundreds of cores; IBM's Watson computer that won Jeopardy has 2880 cores and 16 Terabytes of RAM) and hence Threads really can run concurrently.

Amdahl's Law (How much speed-up with multiple cores)

The extent to which a task can be speeded up with mutiple cores (or CPUs) depends on the fraction of the task that can run in parallel versus the fraction that must run sequentially (serially).

Gene Amdahl (IBM's chief computer architect in the 1960's) gave the following equation:

$$S(n) = \frac{T(1)}{T(n)} = \frac{T(1)}{T(1)\left(B+\frac{1}{n}(1-B)\right)} = \frac{1}{B+\frac{1}{n}(1-B)}$$

where: $S(n)$: Speed-up for *n* processors
B: the fraction of the algorithm that is serial.
T(1): Time for one core.
T(n): Time with *n* cores.

For example:

- If it is *all* serial (B = 1), then *S(n) = 1*
- If *all* can be run concurrently (B = 0), then *S(n) = n*
- If 20% is serial and there are 10 cores, then *S(10) = 3.5*
- If 10% is serial and there are 10 cores, then *S(10)* = 5.2
- If 10% is serial and there are 100 cores, then *S(10)* = 9.2

Why threads can be tricky

Kernel Threads vs. User Threads

5 Interprocess Communication (IPC)

A basic problem: critical sections and atomicity

6 Just for interest..source code and commentary early Unix

- Unix (v6) source code The entire kernel for an early version of Unix (6th edition) was only about 9000 lines long (including comments). Contrast with Windows, estimated to be well over 50,000,000 lines of code!
- Commentary on Unix Source code (v6)