CS61 Section Notes

Section 11 (Fall 2011)

File IO
Fork Review
Signals

UNIX System Programming

A Note on Error Handling

Throughout the text, many example programs use versions of the UNIX system calls written with an initial uppercase letter. These versions execute the actual UNIX system call and then check for errors, exiting the program if an error occurs. See pp. 717-718 (1st ed.: p. 599) in the text for further explanation. Note that exiting the program may not always be the proper thing to do when a system call returns an error.

Basic File IO

The following program copies standard input to standard output. Note that read and write are declared in unistd.h, as are STDIN_FILENO and STDOUT_FILENO. The exit system call is declared in stdlib.h.

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

int main(int argc, char **argv) {
      char c;
      while (read(STDIN_FILENO, &c, 1) != 0) {
            if (write(STDOUT_FILENO, &c, 1) < 0) {
                  exit(1);
            }
      }
      exit(0);
}
```

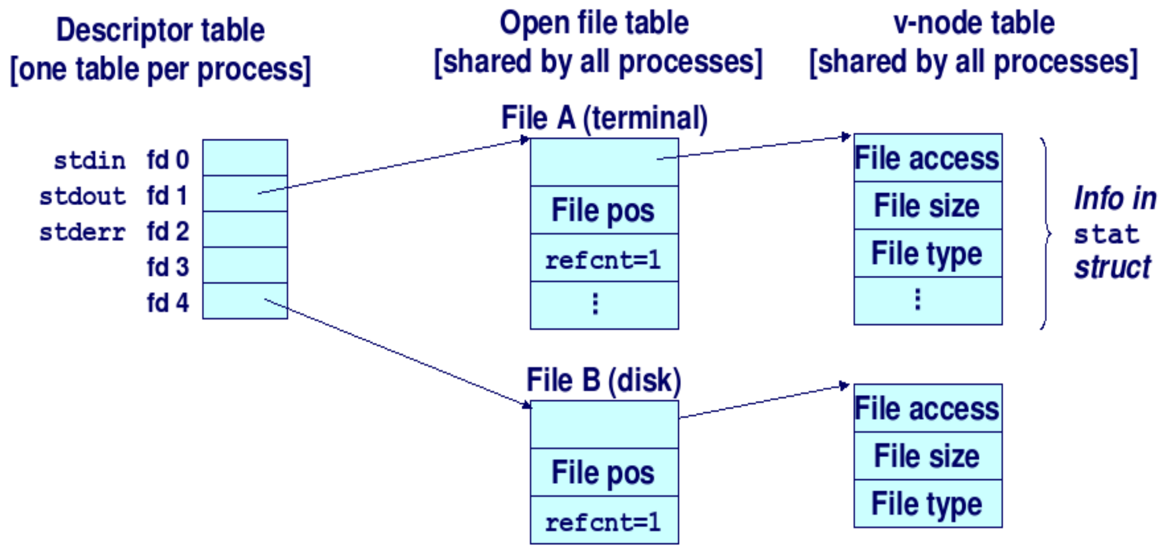
Question 1: As pointed out in lecture, the operations in this program have a lot of overhead. Why is that the case?

In an effort to reduce overhead, a programmer new to UNIX rewrote this program as follows:

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

#define BUF_SIZE 1024

int main(int argc, char **argv) {
      char buf[BUF_SIZE];

      while (read(STDIN_FILENO, buf, BUF_SIZE) != 0) {
            if (write(STDOUT_FILENO, buf, BUF_SIZE) < 0) {
                  exit(1);
            }
      }
      exit(0);
}
```

Question 2: What is wrong with this code? Can you spot another error that both versions share?

Mixing Processes and IO



The kernel keeps track of a process's open files using the three data structures shown above. The filepos field of the file structure keeps track of what position in the file the open file is currently at. The refcnt field keeps track of how many file descriptors are pointing to this open file. This is a reference count.

Question 3: Why would the OS use reference counting for the open files? (Hint: think about the other situation in which we have encountered reference counting.)

Question 4: When a process opens the same file twice, how many entries are added to the descriptor and open file tables? What about the V-node table?

Question 5: Assume the programmer uses the dup2 UNIX system call to duplicate a file descriptor. How do the descriptor and open file tables change?

With those answers in mind, what do the following programs output when run? Assume the file foobar.txt contains the word "foobar". Assuming these are the only processes running, how many entries are in the file descriptor and open file tables at the end of execution?

Program 1

```
#include “csapp.h”

int main(int argc, char **argv) {
      int fd1, fd2;
      char c;

      fd1 = Open(“foobar.txt”, O_RDONLY, 0);
      fd2 = Open(“foobar.txt”, O_RDONLY, 0);
      Read(fd1, &c, 1);
      Read(fd2, &c, 1);
      printf(“c = %c\n”, c);
      exit(0);
}
```

Program 2

```
#include “csapp.h”

int main(int argc, char **argv) {
      int fd1, fd2;
      char c;

      fd1 = Open(“foobar.txt”, O_RDONLY, 0);
      fd2 = Open(“foobar.txt”, O_RDONLY, 0);
      Read(fd2, &c, 1);
      Dup2(fd2, fd1);
      Read(fd1, &c, 1);
      printf(“c = %c\n”, c);
      exit(0);
}
```

Fork Review

Consider the C program below. (For space reasons, we are not checking error return codes, so assume that all functions return normally.)

```
int main () {
 if (fork() == 0) {
   if (fork() == 0) {
     printf("3");
   }
   else {
     pid_t pid; int status;
     if ((pid = wait(&status)) > 0) {
       printf("4");
     }
   }
 }
 else {
   printf("2");
   exit(0);
 }
 printf("0");
 return 0;
}
```

Question 6: For each of the following strings, circle whether (Y) or not (N) this string is a possible output of the program.

A.	32040	Y	N
B.	34002	Y	N
C.	30402	Y	N
D.	23040	Y	N
E.	40302	Y	N

Question 7: Consider the following C program below. (For space reasons, we are not checking error return codes, so assume that all functions return normally.) Note: the atexit function takes a pointer to a function and adds it to a list of functions (initially empty) that will be called when the exit function is called.

```
void end(void) {
 printf("2");
}
```

```
int main() {
 if (fork() == 0)
   atexit(end);
 if (fork() == 0)
   printf("0");
 else
   printf("1");
 exit(0);
}
```

For each of the following strings, circle whether (Y) or not (N) this string is a possible output of the program.

A. 112002	Y	N
B. 211020	Y	N
C. 102120	Y	N
D. 122001	Y	N
E. 100212	Y	N

Signal Handling

We covered fork, exec, and wait in lecture. For the shell assignment, you'll also need to know about signals. A signal is a message from the kernel to the process that notifies the process that some event has happened. There are three signals you will need to know about and handle for your lab

SIGINT	Interrupt: received ctrl-c from the keyboard.
SIGTSTP	Suspend execution: received ctl-z from the keyboard.
SIGCHLD	A child process has stopped or terminated.

A program specifies that it wants to handle a signal by using the UNIX signal call:

```
#include <signal.h>
typedef void handler_t(void);

handler_t *signal(int signum, handler_t *handler);
```

The first parameter to signal is the *signal number*, and the second is the function that should be called when the signal is received. Signal numbers are defined in signal.h.
There are three important points to remember when handling signals:

1. Pending signals are blocked. If a signal is received while the handler for the signal is currently executing, the signal becomes *pending* and its handler will not be re-entered until the current execution finishes.

2. Pending signals are not queued. If a signal is received multiple times while it is blocked (because its handler is being executed), the signal handler will only be called once for the pending signal. A pending signal indicates that *at least one* signal of the given type has arrived, but more than one may have arrived.

3. System calls can be interrupted. On some systems, calls to read and write can be interrupted by a signal. In this case they return a value indicating an error and set errno to EINTR.

In particular, the last point is burdensome to deal with. Because the exact semantics of signal differ from system to system, we provide a wrapper, Signal, that only causes signals to be blocked when their signal handlers are currently being executed and that restarts interrupted system calls.

```
handler_t *Signal(int signum, handler_t *handler) {
    struct sigaction action, old_action;

    action.sa_handler = handler;
    sigemptyset(&action.sa_mask); // block sigs of type being handled
    action.sa_flags = SA_RESTART; // restart syscalls if possible

    if (sigaction(signum, &action, &old_action) < 0)
        unix_error("Signal error");
    return (old_action.sa_handler);
}
```

Adding a Signal Handler

The following program adds a signal handler that catches SIGINT (ctrl-c) and prints out a message:

```
#include "csapp.h"

void sigint_handler(int sig);

int main(int argc, char **argv) {
  Signal(SIGINT, sigint_handler);
  for (;;);
}
```

```
void sigint_handler(int sig) {
    printf("No thanks!\n");
}
```

Question 8: Let's say you're writing a UNIX program that forks multiple children -- a shell perhaps. What happens if your SIGCHLD signal handler is called because a child dies, and while it's executing, several more children die? How can you make sure you clean up after all deceased children?

Sending Signals to Other Processes

Signals are sent to other processes using the kill function:

```
#include <signal.h>

int kill(pid_t pid, int sig);
```

If pid is negative, the signal is sent to all processes in the process group whose parent is abs(pid).

Blocking and Unblocking Signals

Consider the following situation in which a program forks children but also want to maintain state associated with each child.

1. The parent process executes fork, and the kernel creates a child process.

2. Before the parent is able to run, the child terminates and causes a SIGCHLD to be delivered to the parent.

3. When the parent becomes runnable, the SIGCHLD signal is delivered and the parent tries to clean up the state associated with the child. OOPS! The child died before the parent got a chance to set up the state associated with the child!

This is an example of a *race condition*. One way of avoiding this particular race condition is to temporarily block the SIGCHLD signal, and then unblock it when we're ready to handle it. This can be done with sigprocmask and related functions:

```
#include <signal.h>

int sigemptyset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigprocmask(int action, const sigset_t* set,
                sigset_t* oldest);
```

sigemptyset	Initialize a signal set to be empty.

sigaddset	Add a signal to a signal set.
sigprocmask	Change the set of currently blocked signals. If how is SIG_BLOCK, block the specified signals. If how is SIG_UNBLOCK, unblock them. The third parameter, if non-NULL, specifies a destination for the previous signal mask.

Question 9: Suppose you wanted to block detection of a power failure(SIGPWR). Write the code to do that:

Question 10: Consider the following C program. (For space reasons, we are not checking error return codes. You can assume that all functions return normally.)

```
int val = 10;
void handler(sig) {
    val += 5;
    return;
}

int main() {
    int pid;

    signal(SIGCHLD, handler);
    if ((pid = fork()) == 0) {
       val -= 3;
       exit(0);
    }
    waitpid(pid, NULL, 0);
    printf("val = %d\n", val);
    exit(0);
}
```

What is the output of this program? val = ____________

Question 11: Consider the following program:

```
pid_t pid;
int counter = 0;
void handler1(int sig)
{
    counter ++;
    printf("counter = %d\n", counter);
    fflush(stdout); /* Flushes the printed string to stdout */
    kill(pid, SIGUSR1);
}
void handler2(int sig)
{
    counter += 3;
    printf("counter = %d\n", counter);
    exit(0);
}
main() {
    signal(SIGUSR1, handler1);
    if ((pid = fork()) == 0) {
        signal(SIGUSR1, handler2);
        kill(getppid(), SIGUSR1);
        while(1) {};
    }
    else {
        pid_t p; int status;
        if ((p = wait(&status)) > 0) {
            counter += 4;
        printf("counter = %d\n", counter);
        }
    }
}
```
What is the output of this program?