CMSC 321: Operating Systems

Lecture 2: Pipes, Signals

Thought for the day:
Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live.

-- John F. Woods
Recap

- Use `fork()` to create a new (child) process
- Child is an exact copy of the parent
- Both then execute starting beyond `fork()`
- Return value determines if parent or child
- Child uses `execl()` to overlay itself with a new executable
```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char* argv[]) {
    pid_t child_pid;
    int child_status;

    for (int i = 0; i < 3; i++) {
        printf("I'm in the parent for %d sec\n", i);
        sleep(1);
    }

    if ((child_pid = fork()) == 0) {
        execl("./child_print", ./child_print", argv[1], NULL);
    // should never get here, but check to be robust
    } else {
        wait(&child_status);
        printf("Parent off the wait() with child status %d\n", WEXITSTATUS(child_status));
    }

    return 0;
}
```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char* argv[]) {
    pid_t child_pid;
    int child_status;

    for (int i = 0; i < 2; i++) {
        printf("I'm in the parent for %d sec\n", i);
        sleep(1);
    }

    if ((child_pid = fork()) == 0) {
        execl("./child_print", "/child_print", argv[1], NULL);
    } else {
        wait(&child_status);
        printf("Parent off wait() with child status %d\n", WEXITSTATUS(child_status));
    }

    return 0;
}

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char* argv[]) {
    pid_t child_pid;
    int child_status;

    for (int i = 0; i < 2; i++) {
        printf("I'm in the parent for %d sec\n", i);
        sleep(1);
    }

    if ((child_pid = fork()) == 0) {
        execl("./child_print", "/child_print", argv[1], NULL);
    } else {
        wait(&child_status);
        printf("Parent off wait() with child status %d\n", WEXITSTATUS(child_status));
    }

    return 0;
}
```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main(int argc, char* argv[]) {
    pid_t child_pid;
    int child_status;

    for (int i = 0; i < 2; i++) {
        printf("I'm in the parent for %d sec\n", i);
        sleep(1);
    }

    if ((child_pid = fork()) == 0) {
        execle("./child_print", "/child_print", argv[1], NULL);
        // should never get here, but check to be robust
    } else {
        wait(&child_status);
        printf("Parent off wait() with child status %d\n", WEXITSTATUS(child_status));
    }

    return 0;
}
```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char* argv[]) {
    pid_t child_pid;
    int child_status;

    for (int i = 0; i < 2; i++)
    {
        printf("I'm in the parent for %d sec\n", i);
        sleep(1);
    }

    if ((child_pid = fork()) == 0)
    {
        execl("./child_print", "/child_print", argv[1], NULL);
        // should never get here, but check to be robust
    }
    else
    {
        wait(&child_status);
        printf("Parent off wait() with child status %d\n", WEXITSTATUS(child_status));
    }

    return 0;
}
Inter-Process Communication

- In C, processes communicate via pipes
- C treats these pipes like files
- Always remember, communication via a pipe occurs in one direction only!
Setting Up A Pipe

- Before parent forks, it should create the pipe
  
  ```c
  int fileDes[2];   // two ends of comm.
  pipe(fileDes);   // see: man 2 pipe
  ```

- 0 entry of `fileDes` is for reading;
- 1 entry is for writing

- After forking, parent and child will have their own copies of `fileDes`
- Each copy associated with same pipe
Setting Up A Pipe

- **Important**: for each process, close the end of the pipe you won’t be using

- For pipe that parent can use to write to child:
  - parent must `close(2)` the 0 (read) entry of `fileDes`
  - child must `close(2)` the 1 (write) entry of `fileDes`
  - parent writes to 1 entry, child reads from 0 entry

- Communication via pipe occurs in *one direction only*
  - To read from and write to a child, need two pipes
  - For pipe that parent can use to read from child, reverse process above

- Close any pipe when finished using it
Duplicating File Descriptors

• What if we `exec`l a child that reads/writes from/to stdin/stdout?
  – i.e., not our `fileDes`

• We can set up stdin/stdout of the child to be associated with the pipe
  – see: `man 2 dup2`

• In child (before `execl`)
  – associate the 0 entry of the pipe with stdin (fd 0)
    `dup2(fileDes[0], 0);`

• Child can read from stdin & it will be using the pipe
  – e.g., via `cin` or `scanf`
int fileDes[2]; // parent will write to, child will read from
duplicate(fileDes);

if (fork() == 0) {

    // CHILD PROCESS:
    close(fileDes[1]); // close the write end of the pipe
dup2(fileDes[0], 0); // dup the read end as stdin
    execl("./child", ".child", ".child", ".It's a girl!", NULL);
    // child can now read from parent via stdin

} else {

    // PARENT PROCESS:
    close(fileDes[0]); // close the read end of the pipe
    // parent can now write to child via fileDes[0]
}
Communicating Via Pipes

• Communication via a pipe is done using
  – read(int fd, void *buf, size_t count);
  – write(int fd, const void *buf, size_t count);

• First arg: the file descriptor
  – 0 for stdin, 1 for stdout
  – perhaps fileDes[0] or fileDes[1] for our new pipe

• Second arg: a pointer
  – to character string, to an integer, whatever

• Third arg: # of bytes to read/write
  – a char is 1 byte, an int is 4 bytes, etc.
  – use sizeof for integral types, strlen(3) for strings
A Writing Example

• In parent (after forking), write to the child via pipe we created earlier

```c
char* myString = new char[128];
int myInt = 8675309;

sprintf(myString, "If it works, it's obsolete.");

write(fileDes[1], myString, strlen(myString));
write(fileDes[1], &myInt, sizeof(int));
```

• Be aware of exactly what `strlen` does
• Notice that both 2nd args are pointers!
A Reading Example

• In child’s code (after `dup2`, `execl` in parent)

```c
char* buffer = new char[128];
int someInt = 0;

// read the character string from stdin (0)
read(0, buffer, 27); // where does 27 come from?
printf("Buffer = <%s>\n", buffer);

// read the integer from stdin
read(0, &someInt, sizeof(int));
printf("someInt = %d\n", someInt);
```
Using Signals

- C allows processes to signal one another
  ```c
  signal(int signum, sighandler_t handler);
  kill(pid_t pid, int sig);
  - (returns types omitted)
  ```
- Use `signal` to prepare to handle an incoming signal

- In parent:
  ```c
  signal(SIGTERM, myHandlerFunc);
  - where SIGTERM is the C-supplied termination signal
    (man 7 signal for a list of signals)
    - and myHandlerFunc is a function you write
  ```

- Child could signal parent at some point by
  ```c
  kill(getppid(), SIGTERM);
  ```
void myHandlerFunc(int theSignal) {
    // whatever needs to be done to handle the signal
    fprintf(stderr, "Handling the signal\n");
}

int main()
{
    if (fork() == 0) {
        execl("./child", "./child", "It's a girl!", NULL);
        // should never get here
    } else {

        signal(SIGTERM, myHandlerFunc);
        int retval = -1;
        wait(&retval);
        printf("Parent is back from signal!\n");

    }
    return 0;
}
Things To Remember

- Always close any ends of pipes that you aren’t going to use in *that* process

- When a writing process closes its end of a pipe, the reading end will then (and only then) receive EOF

- Always close pipes (and files) when you are finished with them

- Always check return values to make sure things worked properly
  - if not, refer to *errno*(3) and *perror*(3) to determine problem

- Other relevant calls listed at end of proj 1 specs
Current Assignments

• Proj 1: due one week from today