CMSC 216
Introduction to Computer Systems
Lecture 16
Process Control and System-Level I/O
Sections 8.2-8.5, Bryant and O'Hallaron

PROCESS CONTROL (CONT.)
Reaping child processes

• When a process exits, it is still tracked by the kernel (remember the termination process state?)
• Processes are released from the process table only when their parent reaps the terminated child; until this happens, the terminated process is called a zombie
• A parent can release its zombie children from the process table via either the wait() or waitpid() system calls
• If the parent terminates before the child, the child is orphaned, and then adopted by the init process (pid #1); init will reap children as soon as they terminate
wait() system calls

- Can be used to obtain the exit status of the reaped child (or not, if you don't care)
- `pid_t wait(int *status);`
  - requires `<sys/types.h>` and `<sys/wait.h>`
  - pass in a pointer to an int (or `NULL`) that will be populated by the status of the reaped process
  - will reap any single terminated child
  - **blocking** `wait`; does not return until a terminated child exists (if a child exists)
  - returns -1 on error (e.g., no unwaited-for children exist)
  - returns pid of reaped process on success
wait() system calls, cont.

- `pid_t waitpid(pid_t pid, int *status, int options);`
  - will wait on one specified process
  - `pid` is pid of the child process
  - `options` is a number formed from the bitwise OR of several flags (or just 0); **WNOHANG** is the most useful of these flags (doesn't block)
/* #include statements omitted */

int main() {
    pid_t child_pid;
    if ((child_pid = fork()) < 0)
        err(EX_OSERR, "fork error");
    if (child_pid) { /* parent code */
        int status;
        wait(&status); /* nothing happens until child exits */
        printf("Parent pid = %d; my child had pid = %d\n",
                getpid(), child_pid);
        printf("Child exited with status %d\n", status);
    }
    else { /* child code */
        printf("Child pid = %d; my parent has pid = %d\n",
                getpid(), getppid());
    }
    return 0;
}
Exit status

- The status argument points to an `int`; the `int` value is actually more than just the exit code
- We can use macros defined in `<sys/wait.h>` to learn information about the reaped child
  - `WIFEXITED(status)`: true if child terminated normally (via exit/return)
  - `WEXITSTATUS(status)`: the exit status of the normally terminated child
  - `WTERMSIG(status)`: the signal that caused the child to terminate
Environment variables

• Examples:
  – **PATH** (where does the shell look for a program?)
  – **PAGER** (what program do I want to use to view files one page at a time?) (hint: less)

• Are not shell variables
  – shell vars. only affect current shell, env. vars are copied to all child processes run by shell

• Shell commands to set shell variables
  – tcsh: `set var=value`
  – bash: `var=value`

• Shell commands to set environment variables
  – tcsh: `setenv VAR value`
  – bash: `export VAR=value`
Environment variables, cont.

- In the shell, accessed using `$`
  - try "echo $PATH"
- In C programs, can access with 3-param `main()`:
  ```c
  int main(int argc, char *argv[], char *envp[])
  {
    ...  
  }
  ```
  - `envp` is an array of strings of the form `NAME=VALUE`
- Can use `getenv()` to get value of these variables even without using the modified `main()`
- The `extern char **environ` (declared in `<unistd.h>`) also holds the current environment in the same form as `envp`
Loading a new program

• Since the creation of a new process with `fork()` is just a clone of the original, we need a way to change processes to run other programs

• The `execve()` system call can be used to load a program in the context of the current process (so it is the same process, but different program)

• Prototype:

  ```
  #include <unistd.h>
  int execve(const char *filename, 
              char * const argv[], 
              char * const envp[]);
  ```

• Returns `-1` on error; doesn't return on success
  – doesn't return?!
Using `execve()`

- **filename** has to be the absolute path of the executable
- Both the `argv` and `envp` arrays are arrays of strings, with a **NULL** pointer as the final element
  – Not-so-coincidentally, just like the `argv` and `envp` arrays in the 2- and 3-param forms of `main()`
- **argv**: argument vector for the new program
- **envp**: list of environment variable strings, each in the form "**NAME=VALUE**"
  – can just use `environ` to pass along the environment
execve() example

/* #include statements omitted */
extern char **environ;
int main() {
    char *args[] = { "ls", "-l", NULL };
    pid_t child_pid;
    if ((child_pid = fork()) < 0)
        err(EX_OSERR, "fork error");
    if (child_pid) { /* parent code */
        wait(NULL);
        printf("Parent pid = %d; my child had pid = %d\n", 
               getpid(), child_pid);
    }
    else { /* child code */
        printf("PID %d replacing myself\n", getpid());
        execve("/bin/ls", args, environ);
        err(EX_OSERR, "exec error"); /* why no if statement? */
    }
    return 0;
}
Other exec functions

• There are 5 other functions that perform exec operations
  – **l**: use an argument list, terminated by **NULL**
  – **v**: use an argument vector
  – **p**: search the **PATH** list of directories
  – **e**: can specify environment

• `execle(const char *filename, ..., NULL, char * const envp[])`;
• `execv(const char *filename, char * const argv[])`;
• `execl(const char *filename, ..., NULL)`;
• `execvp(const char *progname, char * const argv[])`;
• `execlp(const char *progname, ..., NULL)`;
How to use the other functions

char *filename = "/bin/ls";
char *argv[] = { "ls", "-l", NULL };

execle("/bin/ls", "ls", "-l", NULL, environ);
execv(filename, argv);
execl("/bin/ls", "ls", "-l", NULL);
execvp(argv[0], argv);
execlp("ls", "ls", "-l", NULL);
Why you specify the program twice

• The first time, you tell the system which program to run
• The second time, it's `argv[0]` for that program
  – used to by the program to know what its name is
  – sometimes the same program runs differently under different names
• Can be used for bizarre means...
An admittedly contrived example

$ cat sleeper.c
#include <stdio.h>
#include <unistd.h>
int main() {
    printf("PID = %d\n", getpid());
    execlp("sleep", "blargh", "60", NULL);
    return 1;
}
$ ./sleeper &
PID = 674
$ ps -fp 674
UID   PID  PPID  C STIME TTY          TIME CMD
bofh  674  26789  0 22:12 pts/6    00:00:00 blargh 60
A simple shell

- With fork, exec, and waiting, we have all the tools we need to build our own shell
- Basic idea: infinite loop with prompt
- Inside loop:
  - read in a command line
  - parse the command line
  - fork; parent waits, child execs command
A simple shell program

```c
while (1) {
    /* prompt and read into buffer */
    /* parse buffer into string vector argv */
    switch (fork()) {
        case -1:
            perror("fork error");
            break;
        case 0:
            execvp(argv[0], argv);
            err(EX_OSERR, "exec error");
            break;
        default:
            wait(NULL);
            break;
    }
}
```