CMSC 216
Introduction to Computer Systems
Lecture 20
Process Control

Larry Herman & Alan Sussman

Notes

- Project 5 due today
- Project 6 posted soon
- Exam #2 next Thursday
  - Make (lecture 12) through System-level I/O (lecture 21)
  - practice exam posted tomorrow

Sections 8.2-8.5, Bryant and O'Hallaron

PROCESS CONTROL

Creating new processes

- In UNIX, a new process is created by an existing process, making a parent-child relationship between the two processes
- The system call to do this is **`fork()`**; creates a new copy of the parent process
  - all variables (the whole address space) are copied
  - point of execution (PC) is copied
  - file table information is copied

The `fork()` function

- Prototype:

```
#include <unistd.h>
#include <sys/types.h>
pid_t fork(void);
```

- Returns **twice**, in BOTH parent and child
  - -1: error occurred
    - generally due to process table being full or resource limit reached
  - 0: returned to child process
  - > 0: returns pid of child process to parent

`fork()` example

```
/* #include statements omitted */

int main() {
  int var = 216;
  pid_t child_pid;
  if ((child_pid = fork()) < 0)
    err(EX_OSERR, "fork error");
  if (child_pid) {  /* parent code */
    printf("Parent pid = %d; my child has pid = %d\n",
           getpid(), child_pid);
    var++;
    printf("Var in parent = %d\n", var);
  }
  else {  /* child code */
    printf("Child pid = %d; my parent has pid = %d\n",
           getpid(), getppid());
    var--;
    printf("Var in child = %d\n", var);
  }
  return 0;
}
```


Execution order after a `fork()`

- The previous example's output on one machine was:

```
Child pid = 18532; my parent has pid = 18531
Var in child = 215
Parent pid = 18531; my child has pid = 18532
Var in parent = 217
```

- On Grace, it was:

```
Parent pid = 726; my child has pid = 727
Var in parent = 217
Child pid = 727; my parent has pid = 726
Var in child = 215
```

- It could even be:

```
Parent pid = 23892; my child has pid = 23894
Child pid = 23894; my parent has pid = 23892
Var in child = 215
Var in parent = 217
```

- Print order **within** a process is (usually) determinate
- Print order **between** processes is not

`fork()` semantics

- Some things inherited by a child from its parent process:
  - process credentials: user and group ID (UIDs and GIDs in UNIX terminology)
  - environment
  - a copy of the parent's memory contents, including program code, runtime stack, and heap
  - open file descriptors – `FILE *`'s from `fopen()`. The current file position is also shared between the parent and child, which can cause file consistency issues.
  - signal handling settings (a UNIX way of handling events external to the process, from the operating system or another program)

`fork()` semantics, con't.

- Some things inherited by a child from its parent process, con't:
  - current working directory (set with **`cd`**, viewed with **`pwd`**)
  - root directory
  - resource limits (that can be set and viewed with the tcsh **`limit`** command, or **`ulimit`** in bash)
  - the controlling terminal (the program that controls **`stdin`**, **`stdout`**, and **`stderr`** for the process, which is usually a shell), so the child reads input from and prints output to the same devices that the parent does
  - "nice" value (to determine process priority for scheduling by OS)

`fork()` semantics, con't.

- Some things that are unique to a child process:
  - its process ID
  - it has a different parent process ID (the parent, not the parent's parent)
  - it has its own copy of file descriptors and directory streams.
  - its process times are unique to it
  - its resource utilizations are initially set to 0
  - its pending signals are initialized to the empty set

The dangers of `fork()`

- The process table in the kernel can hold only a finite number of processes; what happens if you fill it up?

```
#include <unistd.h>
int main() {
  while (1) fork();
}
```

- That is a fork **bomb**, and it is a Very Bad Thing
- Fork bombs can be unintentional; a loop that doesn't terminate correctly can easily cause one
  - many students write them accidentally
- Often, it can require sysadmin intervention (e.g., reboot, killing all user processes)
- Lesson: Be careful when using **`fork()`** in a loop

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)

`wait()` example

```
/* #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()`**:

```
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;
}
```