Operating systems and concurrency - B03

David Kendall

Northumbria University

Introduction

This lecture gives a more detailed view of Linux processes than you have seen in lecture A02. It includes details about:

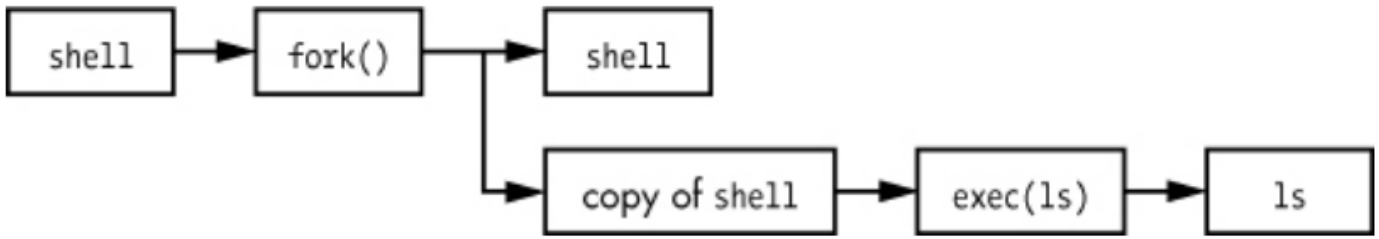
- Processes and multitasking
- `fork()` and `exec()`
- Process identifiers
- `fork()` example
- `exit()`
- `exec()` example
- `wait()`, `waitpid()`

Processes and multitasking

- A key function of the OS is to share the CPU, or a set of CPUs, between many different tasks, also called *processes*
- A process is an instance of a program in execution
- It has its own data: static variables, stack, heap, open files etc
- There can be several processes running based on the same program, e.g.

```
for i in {1..4}
do
  xterm&
done
```

Fork and exec



Ward, B., How Linux Works, 2nd edition, No Starch Press, 2014, § 1.3.4

- In Unix, all user processes, except `init`, are created by `fork()`
- `fork()` is a system call
- When a process calls `fork()` the kernel creates an almost identical copy of the process
- When a process calls `exec(`*`program`*`)`, the kernel starts the execution of *program*, replacing the current process
- Assume we type `ls` into our terminal; the command is passed to the shell, which forks itself, creating a copy of itself, which then runs `exec(ls)` to start the `ls` program, which replaces the copy of the shell

Process identifiers

- Every process has a unique *process identifier*, a non-negative integer
- Every process also has other identifiers associated with it:

```
#include <unistd.h>

pid_t getpid(void)     // process id
pid_t getppid(void)    // parent process id
uid_t getuid(void)     // real user id
uid_t geteuid(void)    // effective user id
gid_t getgid(void)     // real group id
gid_t getegid(void)    // effective group id
```

- There are some special processes:
  - process 0 - usually a system process called the *swapper*
  - process 1 - *init*, called by the kernel at the end of the boot process

Process identifiers example

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

int main(void) {
  printf("My process id is %d\n", getpid());
  printf("The id of my parent process is %d\n",
         getppid());
  printf("My real user id is %d\n", getuid());
  printf("and my group id is %d\n", getgid());
  return(0);
}
```

```
$ gcc -o hellopid hellopid.c
$ ./hellopid
My process id is 25419
The id of my parent process is 2924
My real user id is 1000
and my group id is 1000
$ ls -n hellopid
-rwxrwxr-x 1 1000 1000 8733 Feb  5 17:13 hellopid
```

Fork example

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>

int globvar = 6;

int main(void) {
  int var = 88;
  pid_t pid;
  int status;

  printf("before fork\n");
  if ((pid = fork()) < 0) {
      fprintf(stderr, "fork failed\n");
      exit(-1);
  } else if (pid == 0) { /* child */
      globvar++;
      var++;
      printf("pid = %d, globvar = %d, var = %d\n",
              getpid(), globvar, var);
      exit(0);
  } else { /* parent */
      waitpid(pid, &status, 0); // parent
      printf("pid = %d, globvar = %d, var = %d, child status = %d\n",
              getpid(), globvar, var, status);
  }
}
```

Fork example compilation and output

```
$ gcc -o forkexample forkexample.c
$ ./forkexample
pid = 14491, globvar = 7, var = 89
pid = 14490, globvar = 6, var = 88, child status = 0
```

- A *copy* has been made of the `forkexample` process
  - The global and local variables in the child process are different from the parent
  - Notice that only the child process increments these values; both processes print them; the parent process retains the original values

Fork example compilation and output

```
$ ./forkexample > forkexample.out
$ less forkexample.out
before fork
pid = 25838, globvar = 7, var = 89
before fork
pid = 25837, globvar = 6, var = 88, child status = 0
```

- The file descriptors of the parent have also been copied to the child
- So if the parent process has redirected `stdout`, the child process also redirects `stdout`
- Notice the slight difference in behaviour when `stdout` has been redirected; the output `before fork` appears twice this time.
- This is because the `\n` causes the output buffer to be flushed in interactive mode, but not when output is sent to a file – and even the output buffer of the parent is copied to the child!

The `exit` function

- Using the `exit` function requires the inclusion of the header file `stdlib.h`
- `exit` is called with a single integer argument that indicates the termination status of the calling process; usually, 0 is used to indicate successful termination and some non-zero value, e.g. -1, is used to indicate unsuccessful termination (see the example)
- When a process calls `exit`, all its I/O streams are closed and flushed
- The process that calls `exit` terminates and returns its termination status to its parent. If its parent is not waiting for the status, and has not indicated that it is not interested in the status, then the exiting process becomes a 'zombie' and exists in this state until its parent eventually waits for its status, otherwise the exiting process dies immediately.
- See `man exit` for more details

The `wait` and `waitpid` functions

- In the `fork` example above, once the parent process has forked the child process, it *waits* for the child to finish by calling the `waitpid` function.
- Using the `wait` functions requires the inclusion of the header file `sys/wait.h`
- `waitpid` is called with
  - the id of the process to wait for
  - the address of an integer variable into which the termination status of the child will be stored
  - an integer for a set of flags that gives fine-grained control over the behaviour of the caller, e.g.
    `waitpid(10000, &status, WNOHANG)`
    will wait for the termination of its child process with id 10000, storing its termination status into the integer variable `status`, and returning immediately if the process with id 10000 has not terminated yet.
- The `wait` function waits for *any* terminated child process. It's not possible to specify optional behaviour for `wait`.

exec

- A process can choose to replace itself - text and data - using the `exec()` system call
- Actually there is a family of system calls - `execl`, `execlp`, `execle`, `execv`, `execvp`, `execve`, `fexecve`, which differ in
  - How the command arguments are presented
  - Which environment is used to run the new program
- Usually, the new process runs using the same *environment* as its parent
  - Discover the environment of the shell using the `env` command, e.g. `HOME=/home/cgdk2`, `SHELL=/bin/bash`, `USER=cgdk2`, `PATH=/home/cgdk2/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin`
- `PATH` is particularly important; used to determine where the shell looks for its commands; can be set per user using `~/.bashrc`, e.g. you can add a line like this to your `~/.bashrc`

```
export PATH=/home/cgdk2/special/bin:$PATH
```

exec() example

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>

int main(void) {
  pid_t pid;
  char *argv[] = {"ls", "-l"};

  if ((pid = fork()) < 0) {
      fprintf(stderr, "fork failed\n");
      exit(-1);
  } else if (pid == 0) { // child
      execl("/bin/ls", "ls", "-l", NULL);
  } else {               // parent
      waitpid(pid, NULL, 0);
  }
  exit(0);
}
```

The `exec` functions

Function	*pathname*	*filename*	*fd*	Arg list	*argv[]*	environ	*envp[]*
execl	•			•		•	
execlp		•		•		•	
execle	•			•			•
execv	•				•	•	
execvp		•			•	•	
execve	•				•		•
fexecve			•		•		•
(letter in name)		p	f	l	v		e

- Alternative forms of `exec()`

```
execl("/bin/ls", "ls", "-l", NULL);
execlp("ls", "ls", "-l", NULL);
execv("/bin/ls", argv);
execvp("ls", argv);
```

Summary of process management in C

- Process creation
  `fork()`
  More information - `man fork`
- Process replacement
  `execl()`
  More information - `man execl`
- Process wait
  `wait(), waitpid()`
  More information - `man wait`
- Process termination
  `exit()`
  More information - `man exit`