PROCESS MANAGEMENT

UNIX Programming 2014 Fall by Euiseong Seo

Process

- A process is a program in execution
- A procedure is synchronous
  - Calling function stops what it is doing while called function executes
  - When called function returns, calling function resumes original task
- Processes are asynchronous and concurrent

Easy Way to Create a New Process

- system(3) interprets argument with Bourne shell
  - Thus it allows use of shell metacharacters
- system(3) does not return until shell has completed
- Prototype

```
#include <stdlib.h>
int system(const char *string)
```

Creating a Process

- Use fork(2) to create a process
- Prototype

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

- fork() duplicates the exactly same copy of calling process
  - One process calls fork() and two processes return
  - Return value differentiates them
    - -1: fork() failed
    - 0: executing in the child process
    - >0: executing in the parent process (PID of child process)

Creating a Process

- Example

```
pid_t pid;

pid=fork();
switch(pid) {
    case -1: /* fork failed */
        perror("fork");
        exit(1);
    case 0: /* in new child process */
        printf("In Child, my PID is: %d\n", getpid());
        do_child_stuff();
        exit(0);
    default: /* in parent, PID is PID of child */
        printf("In parent, my child is %d\n", pid);
        break;
}
```

Running a New Program

- You can make current process run a new program with one of *exec* functions
- Exec functions operate by destroying old process image and replacing it with one built from new program
- Prototype

```
#include <unistd.h>

int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ..., char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[], char *const envp[]);
```

Running a New Program

- Differences among exec variations
  - Some take an environment to give new program (and does not use environ variable)
  - Some take an argument vector
  - Some take a list of strings as parameters (less flexible, but sometimes easier to use)
  - Some search PATH environment variable
- Some programs change their behavior depending on argv[0]
  - Single program with multiple links
  - E.g) gzip and gunzip are the same program

Running a New Program

- Example

```
if(fork()==0) {
    execlp("ls", "ls", "-F", (char *)0);
    perror("execlp"); /* why no test? */
    exit(1);
}
```

Termination

- _exit(2) system call is used to terminate process
  - Called by exit()
- Prototype

```
#include <unistd.h>
void _exit(int status);
void exit(int status);
```

- To have a function automatically called by exit(), use atexit(3)
- Prototype

```
#include <stdlib.h>
int atexit(void (*func)(void));
```

Termination

- Example

```
void cleanup(void)
{
    char *mesg = "Cleaning up...\n";
    write(STDOUT_FILENO, mesg, strlen(mesg));
}

main()
{
    atexit(cleanup);
    exit(0);
}
```

Parent Cleanup

- When a process terminates, it becomes a zombie
  - Zombie consumes minimal resources
- Zombie dies when parent requests its status
  - Parent can determine how child died
- wait(2) requests zombie to return its status
- Prototype

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

Parent Cleanup

- Both calls return PID of actual child that was returned
- Child will release its resource after sending status
- You can use various macros to determine how it terminated
  - WIFEXITED(status) - True on normal termination
  - WIFSIGNALED(status) - True on termination by a signal
  - WTERMSIG(status) - Return signal that killed the child

Parent Cleanup

- wait() blocks calling process until one of its children is ready to have its status reaped
- waitpid() allows you to specify which process to wait for and whether to block
  - If PID argument is value below, it waits for
    - -1: Any child
    - >0: Child with PID whose value is that of PID
    - 0: Any child in same process group as calling process
    - <-1: Any child in process group |PID|
  - options determine behavior of function
    - Non-block/block
    - And so on

Parent Cleanup

- Example

```
pid_t pid;
int status;

pid = wait(&status);
if(WIFSIGNALED(status))
    termsig = WTERMSIG(status);
else if(WIFEXITED(status))
    exitstatus = WEXITSTATUS(status);
```

Fork, Exec and Wait Example

```
pid_t pid;
int status;

switch(pid = fork()) {
    case -1:
        perror("fork");
        exit(1);
    case 0: /* in child */
        execlp("ls", "ls", "-F", (char *)NULL);
        perror("execlp");
        exit(1);
        break;
    default: /* parent */
        break;
}

if(waitpid(pid, &status, 0) == -1)
    {perror("waitpid"); exit(1);}

if(WIFSIGNALED(status))
    printf("ls terminated by signal %d.\n", WTERMSIG(status));

return 0
```

What Happened to Zombies?

- Zombies remain until parent cleans them up or parent dies
- When a process dies, kernel walks a list of its children and reparents all of them to init
- init peridically waits on all of its children