Operating Systems 2017/2018 | API Process Control

Ricardo Rocha

Department of Computer Science

Faculty of Sciences

University of Porto

For more information please consult

'Advanced Programming in the UNIX® Environment, 3rd Edition,

W. Richard Stevens and Stephen A. Rago, Addison Wesley'

Sections 8.1–8.3, 8.5–8.6 and 8.10

Process Identifiers

```
#include <unistd.h>

pid_t getpid(void);
// returns process ID of calling process

pid_t getppid(void);
// returns parent process ID of calling process

pid_t getuid(void);
// returns user ID of calling process

pid_t getgid(void);
// returns group ID of calling process
```

Process Creation

```
#include <unistd.h>

pid_t fork(void);
// * creates a new process called the child process
// * both the child and parent continue executing with the
//   instruction that follows the call to fork
// * the child gets a copy of the parent's data space, heap
//   and stack (note that they do not share these portions of
//   memory, they only share the text segment)
// * returns 0 in child process, process ID of child in
//   parent process, -1 on error
```

Process Creation: Code Skeleton

```
...
// parent code before fork
if ((pid = fork()) < 0) {
  // fork failed
} else if (pid = = 0) {
  // child code after fork
} else {
  // parent code after fork
}
// common code after fork
...
```

Process Creation: Example

```
int main () {
  pid_t pid;
  int a = 0, b = 0, c = 0;
  printf("before fork\n");
  if ((pid = fork()) < 0)
    { /* fork failed */ }
  else if (pid = = 0)
    a++;
  else
    b++;
  a++; b++; c++;
  printf("pid: %d, a: %d, b: %, c: %d\n", getpid(), a, b, c);
}
```

Running example:

```
$ ./a.out
before fork
pid: 1730, a: 1, b: 2, c: 1
pid: 1731, a: 2, b: 1, c: 1
$ ./a.out
before fork
pid: 1733, a: 2, b: 1, c: 1
pid: 1732, a: 1, b: 2, c: 1
```

Process Termination

```
#include <stdlib.h>

void exit(int status);
// * causes normal process termination
// * the constants EXIT_SUCCESS (0) and EXIT_FAILURE (!=0) can
//   be used to indicate successful/unsuccessful termination
// * the parent can then obtain the exit status of the child
// * the call to exit() does not return
```

Process Termination

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

pid_t wait(int *status);
// * waits for a child to terminate or returns an error (-1)
//   if it does not have any (more) children
// * if at least one child has terminated, returns the child
//   process ID and fetches its termination status
// * otherwise, it blocks while all children are still running

pid_t waitpid(pid_t pid, int *status, int options);
// * allows to specify which child process it waits for
// * allows to prevent blocking when all children are still
//   running (option WNOHANG)
```

Termination Status Macros

Macro	Description
`WIFEXITED(`*status*`)`	True if status was returned for a child that terminated normally. In this case, we can execute `WEXITSTATUS(`*status*`)` to fetch the low-order 8 bits of the argument that the child passed to `exit`, `_exit`, or `_Exit`.

`WIFSIGNALED(`*status*`)`	True if status was returned for a child that terminated abnormally, by receipt of a signal that it didn't catch. In this case, we can execute
	`WTERMSIG(`*status*`)`
	to fetch the signal number that caused the termination.
	Additionally, some implementations (but not the Single UNIX Specification) define the macro
	`WCOREDUMP(`*status*`)`
	that returns true if a core file of the terminated process was generated.

`WIFSTOPPED(`*status*`)`	True if status was returned for a child that is currently stopped. In this case, we can execute `WSTOPSIG(`*status*`)` to fetch the signal number that caused the child to stop.
`WIFCONTINUED(`*status*`)`	True if status was returned for a child that has been continued after a job control stop (XSI option; `waitpid` only).

Process Termination: Example

```
int main () {
  pid_t pid;
  int status;
  if ((pid = fork()) < 0)
    { /* fork failed */ }
  else if (pid = = 0) {
    printf("pid %d exiting ...\n", getpid());
    exit(5);
  } else {
    wait(&status);
    printf("child %d exit status %d\n", pid, WEXITSTATUS(status));
  }
  exit(0);
}
```

Running example:

```
$ ./a.out
pid 1735 exiting ...
child 1735 exit status 5
```

Starting a New Program

```
#include <unistd.h>

int execl(char *pathname, char *arg, ...);
int execlp(char *filename, char *arg, ...);
int execle(char *pathname, char *arg, ..., char *envp[]);
int execv(char *pathname, char *arg[]);
int execvp(char *filename, char *arg[]);
int execve(char *pathname, char *arg[], char *envp[]);
// * replaces the current process segments -- text, data, heap
//   and stack segments -- with a brand new program from disk
//   and starts executing the new program at its main function
// * keeps the process ID since no new process is created
// * returns an error (-1) if for some reason it fails
```

Exec Differences

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

Starting a New Program: Example

```
int main () {
  pid_t pid;
  if ((pid = fork()) < 0)
    { /* fork failed */ }
  else if (pid = = 0) {
    if (execlp("ls", "ls", "-l", NULL) < 0) {
      printf("shouldn't be here!\n");
      exit(1);
    }
  }
  ...  // parent code after fork
  exit(0);
}
```