CS415 Module 2 Part C - Process Control

Athens State University

Contents

1 The `wait()` System Call

It's a long wait

- We noted in our discussion about `fork()` and `exec()` that most cases require that the parent keep track of its children to find out when and how they terminate

```
#include <sys/wait.h>
pid_t wait(int *status)
```

What's happening with `wait()`

- If no previously waited-upon child of the calling process has yet to terminate, block the caller until one of the children terminate.
- If any child has terminated at the time of the call, immediately return to the calling process
- If the `status` pointer is not `NULL`, information about what caused the child to terminate is placed into the integer pointed to by `status`.
- The kernel update process CPU and resource usage stats to running totals for all of the children of this parent process
- The return type of the system call is the process ID of the child that terminated.
- Upon error, the system call returns -1 to the caller

Note that this means that we use the following loop as a means to force the parent to wait until all of its children have terminated:

```
pid_t childPid;
child_pid = wait(NULL);
while (child_pid == -1) {
  child_pid = wait(NULL);
}
if (errno != ECHILD) {
  errExit("wait");
}
```

Using `wait()`

```
for (j = 1; j < argc; j++)
{
 /* Create one child for each argument */
 switch (fork()) {
   case -1:
     errExit("fork");
   case 0: /* Child sleeps for a while then exits */
     printf("[%s] child %d started with PID %ld, sleeping %s "
             "seconds\n", currTime("%T"), j, (long) getpid(), argv[j]);
     sleep(getInt(argv[j], GN_NONNEG, "sleep-time"));
     _exit(EXIT_SUCCESS);
   default: /* Parent just continues around loop */
   break;
 }
}
```

Using `wait()`

```
numDead = 0;
for (;;) { /* Parent waits for each child to exit */
   childPid = wait(NULL);
   if (childPid == -1) {
   if (errno == ECHILD) {
       printf("No more children - bye!\n");
       exit(EXIT_SUCCESS);
   } else { /* Some other (unexpected) error */
       errExit("wait");
   }
}

numDead++;
printf("[%s] wait() returned child PID %ld (numDead=%d)\n",
currTime("%T"), (long) childPid, numDead);
}
```

2 Waiting for a specific process

Limitations of the `wait()` system call

- If a process creates more than one child, the `wait()` does not allow us to wait for termination of a specific child

- If no child has yet to terminate, then `wait()` always blocks.
  - There are use cases where it's better to just say that no children have terminated
  - We don't always want to block; i.e., stop everything
- The `wait()` system call can tells you what's happening with children that have terminated

The `waitforpid()` system call

```
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *status, int options)
```

- Returns the process ID of the child, 0, or -1 (on error)

The meaning of `pid` variable in `waitforpid()`

- If `pid` is greater than 0, wait for the child whose process ID is equal to *pid*
- If `pid` is equal to 0, wait for any child in the same *process group* as the parent.
- If `pid` is less than -1, wait for for any child whose process group id equals the absolute value of `pid`
- If `pid` is equal to -1, wait for any child

Note that the call `wait(&status)` is equivalent to calling `waitpid(-1, &status, 0)`.

The meaning of the `options` variable in `waitforpid()`

- The options argument is a bitmask that that you can `OR` with one or more of the following flags:
  - `WUNTRACED`: Include status information about whether a process terminated or stopped
  - `WCONTINUED`: Include status information about children who have been restarted by a signal
  - `WNOHANG`: Do not block, always return 0

The `options` variable is important to us due to the `WNOHANG` bitmask. Setting this option converts the system call from blocking to polling, which means that we will not wait until child terminates or stops.

Interpreting the status from `wait()` and `waitforpid()`

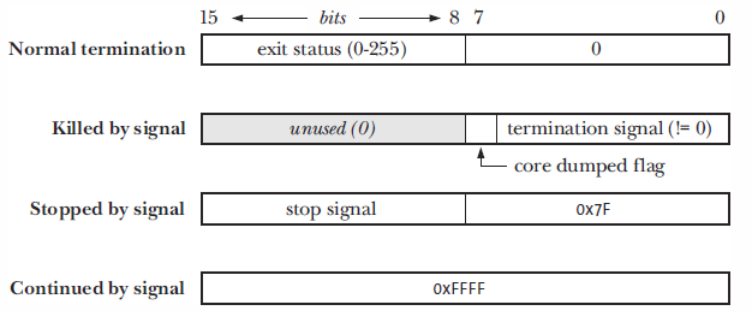

Figure 26-1: Value returned in the *status* argument of *wait()* and *waitpid()*

Fortunately, the header file for the `wait()` and `waitforpid()` system calls provide C macros that parse status codes:

`WIFEXITED(status)` Return true if the child process exited normally

`WEXITSTATUS(status)` Returns the exit status of the child process

`WIFSIGNALED(status)` Returns true if the child process was called by a signal

`WIFSTOPPED(status)` Returns true if the child process was stopped by a signal

```c
#include <sys/wait.h>
#include "print_wait_status.h"          /* Declares printWaitStatus() */
#include "tlpi_hdr.h"
int main(int argc, char *argv[])
{
    int status;
    pid_t childPid;
    if (argc > 1 && strcmp(argv[1], "--help") == 0)
        usageErr("%s [exit-status]\n", argv[0]);
    switch (fork()) {
    case -1: errExit("fork");
    case 0:  /* Child: either exits immediately with given
                       status or loops waiting for signals */
        printf("Child started with PID = %ld\n", (long) getpid());
        if (argc > 1) /* Status supplied on command line? */
            exit(getInt(argv[1], 0, "exit-status"));
        else                    /* Otherwise, wait for signals */
            for (;;)
                pause();
        exit(EXIT_FAILURE); /* Not reached, but good
                               practice */
```

```c
    default: /* Parent: repeatedly wait on child until it
                either exits or is terminated by a signal */
        for (;;) {
            childPid = waitpid(-1, &status, WUNTRACED
#ifdef WCONTINUED       /* Not present on older versions of Linux */
                                                | WCONTINUED
#endif
                    );
            if (childPid == -1)
                errExit("waitpid");
            /* Print status in hex, and as separate decimal bytes */
            printf("waitpid() returned: PID=%ld; status=0x%04x (%d,%d)\n",
                    (long) childPid,
                    (unsigned int) status, status >> 8, status & 0xff);
            printWaitStatus(NULL, status);
            if (WIFEXITED(status) || WIFSIGNALED(status))
                exit(EXIT_SUCCESS);
        }
    }
}
```

3 Orphans and Zombies

Orphaned Children

- A process is *orphaned* when its parent terminates before its children
- In this case, the orphaned child is adopted by the system `init` process
- The orphan's parent process id is set to 1. This provides a means for one to determine if a child's parent is still alive

Zombie Processes

- What happens to a child that terminates before its parent has a chance to perform a `wait()`?
- The child may have finished, but the parent should still be permitted to perform a `wait()` at some later time to determine what happened
- The kernel addresses this question by turning the child into a *zombie process*
  - The kernel keeps an entry for terminated process in its process table
  - The downside is that if the parent never calls `wait()` then the entry is maintained indefinitely in the process table
  - Killing zombie processes require you to kill their parent (like an episode of *Walking Dead*, Linux style)

4 Key Points

Key Points

- The `wait()` and `waitforpid()` system calls
- Orphans and zombie processes