

Processes

waitpid(), errno

Instructor: Linyi Li

Slides adapted from Dr. B. Fraser

Topics

- 1) How can a parent process wait for a child?
- 2) How can we know what errors have happened?

Waiting for a child:
`wait()`

wait()

- wait()

..

–Family of calls; we'll usually use `waitpid()`, but refer to them as just `wait()`

- Common usage

```
pid_t pid = fork();
if (pid != 0) {

    // Parent waits for child process to finish
    if (waitpid(pid, ...) == -1) {
        // Exit on error
    }

} else {
    // Child does something.. exec?
}
```

man 2 wait

wait(2)

System Calls Manual

wait(2)

NAME

wait, waitpid, waitid - wait for process to change state

LIBRARY

Standard C library (libc, -lc)

SYNOPSIS

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

```
pid_t wait(int *_Nullable wstatus);
```

```
pid_t waitpid(pid_t pid, int *_Nullable wstatus, int options);
```

DESCRIPTION

All of these system calls are used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed. A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal. In the case of a terminated child, performing a wait allows the system to release the resources associated with the child; if a wait is not performed, then the terminated child remains in a "zombie" state (see NOTES below).

A lot to understand in just a single syscall!
What are these options?

Parts of waitpid()

```
pid_t waitpid(pid_t pid, int *_Nullable wstatus, int options);
```

- **pid**
 - ..
- **wstatus**
 - pointer to an `int` to store..
 - **_Nullable** tells reader OK to be **NULL**
- **options**
 - we'll leave as 0; can specify non-blocking (don't wait)
e.g., **WNOHANG**

wstatus

- `waitpid()` takes a pointer for `wstatus`
 - Calling code (e.g., `main()`)
 - ..
 - `waitpid()` given a pointer to this space
 - `waitpid()` writes an answer into that space
- Effectively, `main()` declares a variable so `waitpid()` has somewhere to write info; called an.. **output parameter**

```
pid_t pid = fork();
if (pid) {
    int wstatus = 0;
    if (waitpid(pid, &wstatus, 0) == -1) {
        perror("waitpid");
        exit(EXIT_FAILURE);
    }
}
```

wait() Status Check Macros

- Why did the child terminate?
(*wstatus(): is a complicated value*)
 - Normally: **exit()** or **return** from main

```
if (WIFEXITED(wstatus)) {  
    printf("Reason: %d\n", WEXITSTATUS(wstatus));  
}
```

- Terminated by Signal?

```
if (WIFSIGNALED(wstatus)) {  
    printf("Terminated by signal # %d\n",  
          WTERMSIG(status));  
}
```


Activity: wait()

- (10 mins) Write a program that:
 - Creates a child process
 - Child process runs `ls -a -l`
 - Parent process waits for the child process to terminate using `waitpid()`
 - If child exits normally, print the exit status.
- Hints:
 - OK to reuse previous code examples from class.
 - Use `execl()`; pass in arguments separately

See code slide: “waitpid() on child”

waitpid() on child

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

int main() {
    pid_t pid = fork();

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

        if (WIFEXITED(wstatus)) {
            printf("Child done with exit status: %d\n", WEXITSTATUS(wstatus));
        } else {
            printf("Child did not exit normally.\n");
        }
    } else {
        if (execl("/usr/bin/ls", "/usr/bin/ls", "-a", "-l", NULL) == -1) {
            perror("execl");
            exit(EXIT_FAILURE);
        }
    }

    return 0;
}
```

Zombies and Orphans

Zombies

- What happens when an application terminates?
 - OS retains some state information of terminated processes (so parent can find out reason for exiting)
 - This takes up some memory.
 - Calling `wait()` on a terminated process frees this memory.
- Zombie
 - Process state where child process terminates
 - ..
 - (It's dead, but not *completely*)
 - Having many zombies uses kernel resources; so important to always `wait()` on child process.

Orphans

- Orphan
 - This is the state where..
 - Orphan processes no longer have a parent process.
- Linux handling of Orphan Processes
 - Orphan child process becomes a child process of `init`
 - `init` calls `wait()` on all child processes



ABCD: wait()

- Which of the following is true about `wait()`?
 - (a) `wait()` takes care of orphans.
 - (b) `wait()` combats the spread of zombies.
 - (c) `wait()` is a replacement for ``sleep()``.
 - (d) `wait()` allows child process to get input from parent.

What went wrong?
errno

man errno

- Run:
man errno
 - What do you notice about it?
- Look at:
 - Description
 - When is it useful?
 - What is its type?
 - How can my program get access to it?

errno & perror

- `errno` is an integer variable that is..
 - Adds more information about which error has occurred.
 - It is defined in `errno.h`
 - C can print an explanation for you from just the `errno` using `perror("your message here")`

```
if (somecall() == -1) {  
    if (errno == EACCESS) {  
        printf("You don't have access.\n");  
    } else {  
        perror("somecall() failed")  
    }  
}
```

- `errno` is similar to `wstatus` from `wait()`:
 - Status code set by a system call if there's an error.

Demo: fork-bomb with errors

- `fork()` sets `errno` on failure

–`man fork`

Checkout possible
`errno` values.

- Demo?

–`ulimit -S -u 100`

fork-bomb with error output

```
fork: Resource temporarily unavailable
EAGAIN
fork: Resource temporarily unavailable
EAGAIN
fork: Resource temporarily unavailable
fork: Resource temporarily unavailable
EAGAIN
EAGAIN
fork: Resource temporarily unavailable
EAGAIN
fork: Resource temporarily unavailable
EAGAIN
```

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

int main() {
    while (1) {
        if (fork() == -1) {
            char *str = NULL;
            switch (errno) {
                case EAGAIN:
                    str = "EAGAIN";
                    break;
                case ENOMEM:
                    str = "ENOMEM";
                    break;
                case ENOSYS:
                    str = "ENOSYS";
                    break;
                default:
                    break;
            }
            perror("fork");
            printf("%s\n", str);
        }
    }
}
```

Summary

- Waiting on your children:
 `wait()`, `waitpid()`
 - Pass `&wstatus` to find out why child terminated.
 - Terminated process becomes a `zombie` until waited on.
 - Terminating the parent creates `orphans` processes.
- Use `errno` to find out info
 - Print error message to screen with `perror()`.