fork(2)

- Let’s code some examples
- Why is fork**(2)** special?
- Why do man pages have numbers now?

fork(2)

- Order of output?

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

int main ()
{
        int pid;
        int pid_2;

        if ((pid = fork()) == 0){
                printf("I am a smol child\n");
                if ((pid_2 = fork()) == 0) {
                        printf("I am a much smaller child\n");
                } else {
                        wait(NULL);
                        printf("I waited for the smaller child\n");
                }
        } else {
                wait(NULL);
                printf("I am a parent\n");

        }
        return 0;
}
```

wait(2)

- What is wait(2) doing?
- Error handling and exit status
  - Is the wait(NULL) call correct?
  - What could happen here?

fork(2)

- Making behavior more correct
  - Error codes from fork? What would that mean?

RETURN VALUE

On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created, and errno is set appropriately.

ERRORS

EAGAIN A system-imposed limit on the number of threads was encountered. There are a number of limits that may trigger this error:

* the **RLIMIT_NPROC** soft resource limit (set via **setrlimit**(2)), which limits the number of processes and threads for a real user ID, was reached;
* the kernel's system-wide limit on the number of processes and threads, /proc/sys/kernel/threads-max, was reached (see **proc**(5));
* the maximum number of PIDs, /proc/sys/kernel/pid_max, was reached (see **proc**(5)); or
* the PID limit (pids.max) imposed by the cgroup "process number" (PIDs) controller was reached.

EAGAIN The caller is operating under the **SCHED_DEADLINE** scheduling policy and does not have the reset-on-fork flag set. See **sched**(7).

ENOMEM **fork**() failed to allocate the necessary kernel structures because memory is tight.

ENOMEM An attempt was made to create a child process in a PID namespace whose "init" process has terminated. See **pid_namespaces**(7).

ENOSYS **fork**() is not supported on this platform (for example, hardware without a Memory-Management Unit).

ERESTARTNOINTR (since Linux 2.6.17)
System call was interrupted by a signal and will be restarted. (This can be seen only during a trace.)

fork(2)

- Making behavior more correct
  - Error codes from fork? What would that mean?

RETURN VALUE

On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created, and errno is set appropriately.

ERRORS

EAGAIN A system-imposed limit on the number of threads was encountered. There are a number of limits that may trigger this error:

* the **RLIMIT_NPROC** soft resource limit (set via **setrlimit**(2)), which limits the number of processes and threads for a real user ID, was reached;
* the kernel's system-wide limit on the number of processes and threads, /proc/sys/kernel/threads-max, was reached (see **proc**(5));
* the maximum number of PIDs, /proc/sys/kernel/pid_max, was reached (see **proc**(5)); or
* the PID limit (pids.max) imposed by the cgroup "process number" (PIDs) controller was reached.

EAGAIN The caller is operating under the **SCHED_DEADLINE** scheduling policy and does not have the reset-on-fork flag set. See **sched**(7).

ENOMEM **fork**() failed to allocate the necessary kernel structures because memory is tight.

ENOMEM An attempt was made to create a child process in a PID namespace whose "init" process has terminated. See **pid_namespaces**(7).

ENOSYS **fork**() is not supported on this platform (for example, hardware without a Memory-Management Unit).

ERESTARTNOINTR (since Linux 2.6.17)
System call was interrupted by a signal and will be restarted. (This can be seen only during a trace.)

fork(2)

- Errors aren’t always bad

```
1   void unix_error(char *msg) /* Unix-style error */
2   {
3       fprintf(stderr, "%s: %s\n", msg, strerror(errno));
4       exit(0);
5   }
```

Given this function, our call to fork reduces from four lines to two lines:

```
1       if ((pid = fork()) < 0)
2           unix_error("fork error");
```

Advanced wait(2)

- Wait(2) blocks by default
  - Why would you not want to block?

WNOHANG. Return immediately (with a return value of 0) if none of the child processes in the wait set has terminated yet. The default behavior suspends the calling process until a child terminates; this option is useful in those cases where you want to continue doing useful work while waiting for a child to terminate.

WUNTRACED. Suspend execution of the calling process until a process in the wait set becomes either terminated or stopped. Return the PID of the terminated or stopped child that caused the return. The default behavior returns only for terminated children; this option is useful when you want to check for both terminated *and* stopped children.

WCONTINUED. Suspend execution of the calling process until a running process in the wait set is terminated or until a stopped process in the wait set has been resumed by the receipt of a SIGCONT signal. (Signals are explained in Section 8.5.)

Advanced wait(2)

- Capturing error status
  - Macros vs Functions

WIFEXITED(`status`). Returns true if the child terminated normally, via a call to `exit` or a return.

WEXITSTATUS(`status`). Returns the exit status of a normally terminated child. This status is only defined if WIFEXITED() returned true.

WIFSIGNALED(`status`). Returns true if the child process terminated because of a signal that was not caught.

WTERMSIG(`status`). Returns the number of the signal that caused the child process to terminate. This status is only defined if WIFSIGNALED() returned true.

WIFSTOPPED(`status`). Returns true if the child that caused the return is currently stopped.

WSTOPSIG(`status`). Returns the number of the signal that caused the child to stop. This status is only defined if WIFSTOPPED() returned true.

WIFCONTINUED(`status`). Returns true if the child process was restarted by receipt of a SIGCONT signal.

```
/* This will define all the `__W*' macros.  */
# include <bits/waitstatus.h>

# define WEXITSTATUS(status)     __WEXITSTATUS (status)
# define WTERMSIG(status)        __WTERMSIG (status)
# define WSTOPSIG(status)        __WSTOPSIG (status)
# define WIFEXITED(status)       __WIFEXITED (status)
# define WIFSIGNALED(status)     __WIFSIGNALED (status)
# define WIFSTOPPED(status)      __WIFSTOPPED (status)
# ifdef __WIFCONTINUED
#  define WIFCONTINUED(status)   __WIFCONTINUED (status)
# endif
```

Advanced wait(2)

- Ugh … its bit operators again!!

```
/* This will define all the `__W*' macros.  */
# include <bits/waitstatus.h>

# define WEXITSTATUS(status)     __WEXITSTATUS (status)
# define WTERMSIG(status)        __WTERMSIG (status)
# define WSTOPSIG(status)        __WSTOPSIG (status)
# define WIFEXITED(status)       __WIFEXITED (status)
# define WIFSIGNALED(status)     __WIFSIGNALED (status)
# define WIFSTOPPED(status)      __WIFSTOPPED (status)
# ifdef __WIFCONTINUED
#  define WIFCONTINUED(status)   __WIFCONTINUED (status)
# endif
```

```
/* If WIFEXITED(STATUS), the low-order 8 bits of the status.  */
#define __WEXITSTATUS(status)   (((status) & 0xff00) >> 8)

/* If WIFSIGNALED(STATUS), the terminating signal.  */
#define __WTERMSIG(status)      ((status) & 0x7f)

/* If WIFSTOPPED(STATUS), the signal that stopped the child.  */
#define __WSTOPSIG(status)      __WEXITSTATUS(status)

/* Nonzero if STATUS indicates normal termination.  */
#define __WIFEXITED(status)     (__WTERMSIG(status) == 0)

/* Nonzero if STATUS indicates termination by a signal.  */
#define __WIFSIGNALED(status) \
  (((signed char) (((status) & 0x7f) + 1) >> 1) > 0)
```

fork(2)

- Checking returns
- Showing PID

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

int main ()
{
        int pid;
        int pid_2;

        if ((pid = fork()) == 0){
                printf("I am a smol child\n");
                if ((pid_2 = fork()) == 0) {
                        printf("I am a much smaller child\n");
                        sleep(30);
                } else if (pid_2 > 0) {
                        printf("I waited for the smaller child pid_2(%d)\n", pid_2);
                        wait(&pid_2);
                } else { // pid_2 < 0 is an error
                        printf("Whoops, fork failed, let's go away\n");
                        return(-1);
                }
        } else if (pid > 0) {
                printf("I am a parent child pid (%d)\n", pid);
                wait(&pid);

        } else { // pid < 0 is an error
                printf("Whoops, parent fork failed, let's go away\n");
                return(-1);
        }
        return 0;
}
```

signals

- Kill(1)
- Default is TERM
- Invisible handler
  - Default is to exit

```
sbruno@iwilcox-XPS-8940:~$ ./test_sig
I am a parent child pid (156271)
I am a smol child
I waited for the smaller child pid_2(156272)
I am a much smaller child
```

```
sbruno@iwilcox-XPS-8940:~$ ps -ef|grep 156271
sbruno    156271  156270  0 09:21 pts/1    00:00:00 ./test_sig
sbruno    156272  156271  0 09:21 pts/1    00:00:00 ./test_sig
sbruno    156274  156227  0 09:21 pts/3    00:00:00 grep 156271
sbruno@iwilcox-XPS-8940:~$ kill 156271
sbruno@iwilcox-XPS-8940:~$
```