**Fork**

fork - create a child process

#include <sys/types.h>

#include <unistd.h>

pid\_t fork(void);

**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.

fork() creates a new process by duplicating the calling process.

* Child process - The new process
* Parent process - The calling process

After a parent process fork()s a child process, it must wait() (or waitpid()) for that child process to exit. It is this act of wait()ing that allows all remnants of the child to vanish.

Exception: the parent can ignore the SIGCHLD signal (SIGCLD on some older systems) and then it won't have to wait(). This can be done (on systems that support it) like this:

**Note:** The termination signal of the child is always SIGCHLD

main() {

signal(SIGCHLD, SIG\_IGN); /\* now I don't have to wait()! \*/

.

.

fork(); fork(); fork(); /\* Rabbits, rabbits, rabbits! \*/

}

Now, when a child process dies and has not been wait()ed on, it will usually show up in a ps listing as “<defunct>”. It will remain this way until the parent wait()s on it, or it is dealt with as mentioned below.

When the parent dies before it wait()s for the child (assuming it is not ignoring SIGCHLD), the child is reparented to the init process (PID 1).

**How does init know to wait() for these zombie processes?**

* On some systems, init periodically destroys all the defunct processes it owns.
* On other systems, it outright refuses to become the parent of any defunct processes, instead destroying them immediately.

# Some points

The child process and the parent process run in separate memory spaces. At the time of fork() both memory spaces have the same content.

Memory writes, file mappings (mmap(2)), and unmappings (munmap(2)) performed by one of the processes do not affect the other.

**Child is different from it's parent in terms of:**

1. The child has its own unique process ID
2. The child's parent process ID is the same as the parent's process ID
3. Process resource utilizations (getrusage(2)) and CPU time counters (times(2)) are reset to zero in the child
4. The child's set of pending signals is initially empty

**Child inherits from it's parent:**

1. Open file description locks fcntl(2) and locks flock(2)

**Child does not inherits from it's parent:**

1. Timers (setitimer(2), alarm(2), timer\_create(2))
2. Process-associated record locks fcntl(2)
3. Outstanding asynchronous I/O operations (aio\_read(3), aio\_write(3))
4. Asynchronous I/O contexts from its parent (io\_setup(2))
5. Parent's memory locks (mlock(2), mlockall(2))
6. Semaphore adjustments from its parent (semop(2))

# Further points

* The child process is created with a single thread the one that called fork(). The entire virtual address space of the parent is replicated in the child, including the states of mutexes, condition variables, and other pthreads objects; the use of pthread\_atfork(3) may be helpful for dealing with problems that this can cause.
* After a fork() in a multithreaded program, the child can safely call only async-signal-safe functions (see signal-safety(7)) until such time as it calls execve(2).
* The child inherits copies of the parent's set of open file descriptors. Each file descriptor in the child refers to the same open file description (see open(2)) as the corresponding file descriptor in the parent. This means that the two file descriptors share open file status flags, file offset, and signal-driven I/O attributes (see the description of F\_SETOWN and F\_SETSIG in fcntl(2)).
* The child inherits copies of the parent's set of open message queue descriptors (see mq\_overview(7)). Each file descriptor in the child refers to the same open message queue description as the corresponding file descriptor in the parent. This means that the two file descriptors share the same flags (mq\_flags).
* The child inherits copies of the parent's set of open directory streams (see opendir(3)). POSIX.1 says that the corresponding directory streams in the parent and child may share the directory stream positioning; on Linux/glibc they do not.

# Example

#include <stdio.h>

#include <stdlib.h>

#include <errno.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

int main(void) {

pid\_t pid;

int rv;

switch(pid = fork()) {

case -1:

perror("fork"); /\* something went wrong \*/

exit(1); /\* parent exits \*/

case 0:

printf(" CHILD: This is the child process!\n");

printf(" CHILD: My PID is %d\n", getpid());

printf(" CHILD: My parent's PID is %d\n", getppid());

printf(" CHILD: Enter my exit status (make it small): ");

scanf(" %d", &rv);

printf(" CHILD: I'm outta here!\n");

exit(rv);

default:

printf("PARENT: This is the parent process!\n");

printf("PARENT: My PID is %d\n", getpid());

printf("PARENT: My child's PID is %d\n", pid);

printf("PARENT: I'm now waiting for my child to exit()...\n");

wait(&rv);

printf("PARENT: My child's exit status is: %d\n", WEXITSTATUS(rv));

printf("PARENT: I'm outta here!\n");

}

return 0;

}

# END