Operating Systems & Systems Programming I

Processes

Keith Bugeja / Joshua Ellul / Alessio Magro / Kevin Vella

March 5, 2018

Department of Computer Science
Faculty of ICT, University of Malta

Overview

Overview

Processes

- Definition and Structure
- Start and Termination
- Process Control

Process definition

Process definition

What is a **process**?

What is a **process**?

A process is an instance of a computer program in execution:

Process definition

What is a **process**?

A process is an instance of a computer program in execution:

- identified by the *process ID* (PID), a non-negative number guaranteed to be unique at any single point in time;

What is a **process**?

A process is an instance of a computer program in execution:

- identified by the *process ID* (PID), a non-negative number guaranteed to be unique at any single point in time;
- each PID is associated with a process descriptor inside the OS scheduler.

Process definition

What is a **process**?

A process is an instance of a computer program in execution:

- identified by the *process ID* (PID), a non-negative number guaranteed to be unique at any single point in time;
- each PID is associated with a process descriptor inside the OS scheduler.

Note

A **binary** is compiled, executable code lying dormant on storage medium such as disk; colloquially it is called a program. Large significant binaries called applications.

Process identifiers

The `ps` command can be used to obtain a list of PIDs.

```
$ ps -aux
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root          1  0.0  0.0 119688  5876 ?        Ss   11:19   0:03 /sbin/init splash
root          2  0.0  0.0      0     0 ?        S    11:19   0:00 [kthreadd]
root          4  0.0  0.0      0     0 ?        S<   11:19   0:00 [kworker/0:0H]
root          6  0.0  0.0      0     0 ?        S    11:19   0:00 [ksoftirqd/0]
root          7  0.0  0.0      0     0 ?        R    11:19   0:00 [rcu_sched]
root          8  0.0  0.0      0     0 ?        S    11:19   0:00 [rcu_bh]
root          9  0.0  0.0      0     0 ?        S    11:19   0:00 [migration/0]
root         10  0.0  0.0      0     0 ?        S<   11:19   0:00 [lru-add-drain]
root         11  0.0  0.0      0     0 ?        S    11:19   0:00 [watchdog/0]
root         12  0.0  0.0      0     0 ?        S    11:19   0:00 [cpuhp/0]
root         13  0.0  0.0      0     0 ?        S    11:19   0:00 [cpuhp/1]
root         14  0.0  0.0      0     0 ?        S    11:19   0:00 [watchdog/1]
root         15  0.0  0.0      0     0 ?        S    11:19   0:00 [migration/1]
root         16  0.0  0.0      0     0 ?        S    11:19   0:00 [ksoftirqd/1]
root         18  0.0  0.0      0     0 ?        S<   11:19   0:00 [kworker/1:0H]
...
root       3378  0.0  0.0      0     0 ?        S    12:17   0:00 [kworker/u256:2]
root       3602  0.0  0.0      0     0 ?        S    12:21   0:00 [kworker/0:2]
keith      3612  0.0  0.0  38584  3424 pts/1    R+   12:21   0:00 ps -aux
$
```

Process identifiers

The `ps` command can be used to obtain a list of PIDs.

```
$ ps -aux
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root          1  0.0  0.0 119688  5876 ?        Ss   11:19   0:03 /sbin/init splash
root          2  0.0  0.0      0     0 ?        S    11:19   0:00 [kthreadd]
root          4  0.0  0.0      0     0 ?        S<   11:19   0:00 [kworker/0:0H]
root          6  0.0  0.0      0     0 ?        S    11:19   0:00 [ksoftirqd/0]
root          7  0.0  0.0      0     0 ?        R    11:19   0:00 [rcu_sched]
root          8  0.0  0.0      0     0 ?        S    11:19   0:00 [rcu_bh]
root          9  0.0  0.0      0     0 ?        S    11:19   0:00 [migration/0]
root         10  0.0  0.0      0     0 ?        S<   11:19   0:00 [lru-add-drain]
root         11  0.0  0.0      0     0 ?        S    11:19   0:00 [watchdog/0]
root         12  0.0  0.0      0     0 ?        S    11:19   0:00 [cpuhp/0]
root         13  0.0  0.0      0     0 ?        S    11:19   0:00 [cpuhp/1]
root         14  0.0  0.0      0     0 ?        S    11:19   0:00 [watchdog/1]
root         15  0.0  0.0      0     0 ?        S    11:19   0:00 [migration/1]
root         16  0.0  0.0      0     0 ?        S    11:19   0:00 [ksoftirqd/1]
root         18  0.0  0.0      0     0 ?        S<   11:19   0:00 [kworker/1:0H]
...
root       3378  0.0  0.0      0     0 ?        S    12:17   0:00 [kworker/u256:2]
root       3602  0.0  0.0      0     0 ?        S    12:21   0:00 [kworker/0:2]
keith      3612  0.0  0.0  38584  3424 pts/1    R+   12:21   0:00 ps -aux
$
```

Example uses BSD option syntax to show all processes from all users.

Process hierarchy

The first process that the kernel executes after booting the system is called the *init process* (pid=1):

- kernel searches for `init` in `/sbin`, `/etc`, `/bin`;
- unless specified in `init` kernel command-line parameter.

Process hierarchy

The first process that the kernel executes after booting the system is called the *init process* (pid=1):

- kernel searches for `init` in `/sbin`, `/etc`, `/bin`;
- unless specified in `init` kernel command-line parameter.

The kernel runs the *idle "process"* (pid=0) when there are no *runnable* processes.

The first process that the kernel executes after booting the system is called the *init process* (pid=1):

- kernel searches for `init` in `/sbin`, `/etc`, `/bin`;
- unless specified in `init` kernel command-line parameter.

The kernel runs the *idle "process"* (pid=0) when there are no *runnable* processes.

Note

If the kernel fails to find an init process, it will try to load the Bourne shell (`/bin/sh`). Failing that, system is halted.

Process hierarchy

A process that **spawns** a new process is known as the *parent*

- spawned process is known as the *child*
- relationship captured by parent PID (ppid)

A process that `spawns` a new process is known as the *parent*

- spawned process is known as the *child*
- relationship captured by parent PID (ppid)

All processes except `init` (pid=1) and `kthreadd` (pid=2) have a parent with a non-zero ppid:

- think of them as spawned directly by the kernel (ppid=0)

Programmatically, we can get the PID as follows:

Programmatically, we can get the PID as follows:

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

int main(int argc, char **argv) {
    printf("PID is [%d]; parent PID is [%d]\n"
        , getpid()   // returns the pid
        , getppid()  // returns the parent pid
    );

    return 0;
}
```

Programmatically, we can get the PID as follows:

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

int main(int argc, char **argv) {
    printf("PID is [%d]; parent PID is [%d]\n"
        , getpid()   // returns the pid
        , getppid()  // returns the parent pid
    );

    return 0;
}
```

`getpid` and `getppid` return `pid_t`, defined in `<sys/types.h>`

Process hierarchy

```
$ # compile and run pid program
$ gcc -o pid pid.c
$ ./pid
PID is [45044]; parent PID is [30842]
$
```

Process start-up and termination

How does a program transition into becoming a process?

How does a program transition into becoming a process?

At a very high level, it involves the following steps:

1. a process is created to hold the program image
2. program image is loaded/mapped in the process address space
3. before `main` executes, start-up code is invoked
4. on termination of `main`, finalisation code is invoked

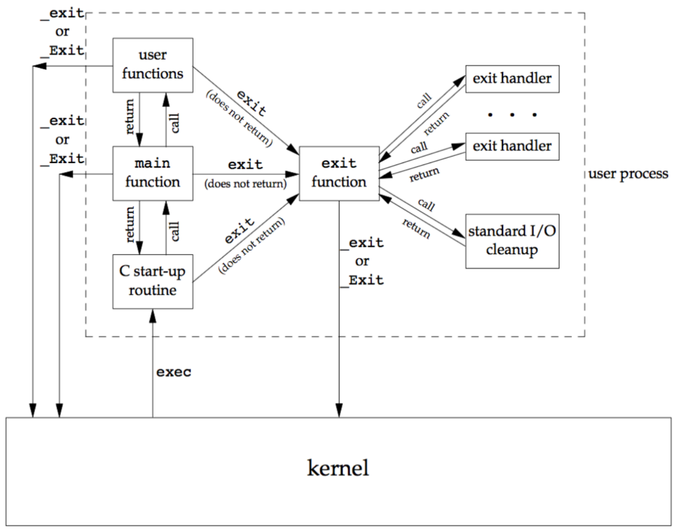

Figure 1: Process Start-up and Termination

Process address space

The process address space contains the following segments:

- text segment for the machine instructions that the CPU executes
- initialised data segment for variables that are specifically initialised in the program
- unitialised data segment (bss) which is initialised by the kernel to arithmetic 0 or null pointers before program execution
- stack, where automatic variables and function information are stored (see stack frame)
- heap, for dynamic memory allocation

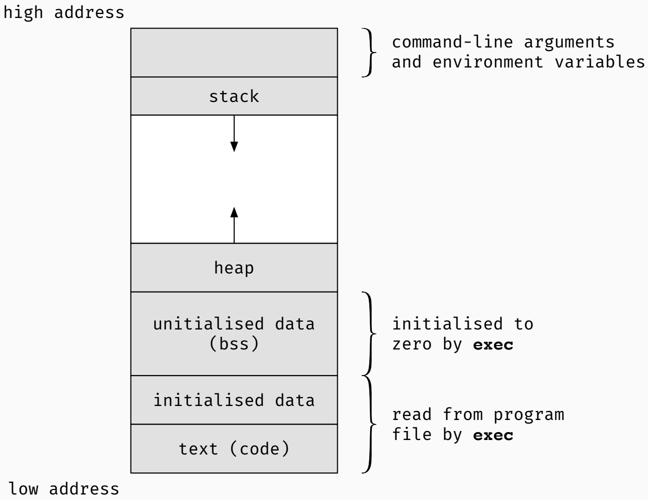

Figure 2: Typical memory arrangement

To examine the size of the respective sections in a program binary, use the `size` command:

```
$ # show the section sizes for the pid program
$ size pid
   text    data     bss     dec     hex filename
   1404     568       8    1980     7bc pid
$
```

Process environment

During launch, our program is passed important information

- launch arguments, e.g.: `gcc, -o, myprog, myprog.c`
- environment strings, e.g.: `HOME=/home/user, ...`

Process environment

During launch, our program is passed important information

- launch arguments, e.g.: `gcc, -o, myprog, myprog.c`
- environment strings, e.g.: `HOME=/home/user, ...`

Note

Both lists (of strings), when made available to a process are terminated with a `NULL` entry.

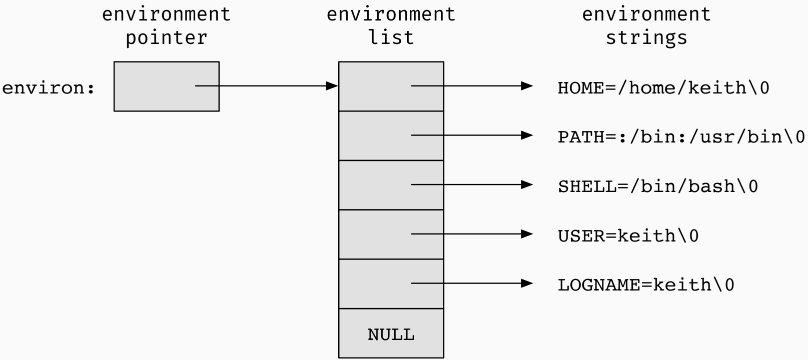

Figure 3: User environment made available to the program by `exec`

Process environment

```c
#include <stdio.h>

extern char **environ;

int main(int argc, char **argv) {
    // Print out command line arguments
    for (int i = 0; i < argc; ++i)
        printf("argv[%d] = %s\n", i, argv[i]);

    // Print out environment strings
    for (int i = 0; environ[i] != NULL; ++i)
        printf("environ[%d] = %s\n", i, environ[i]);

    return 0;
}
```

Note

Alternatively, the main function can be passed a third argument:

`int main(int argc, char **argv, char **env)`

Process environment

```
$ gcc -o argenv argenv.c
$ ./argenv arg1 arg2 another_arg and another
argv[0]=./argenv
argv[1]=arg1
argv[2]=arg2
argv[3]=another_arg
argv[4]=and
argv[5]=another
environ[0]=XDG_VTNR=7
environ[1]=LC_PAPER=mt_MT.UTF-8
environ[2]=XDG_SESSION_ID=2
environ[3]=SSH_AGENT_PID=1747
environ[4]=PAM_KWALLET5_LOGIN=/run/user/1000/kwallet5.socket
environ[5]=LC_ADDRESS=mt_MT.UTF-8
environ[6]=KDE_MULTIHEAD=false
environ[7]=LC_MONETARY=mt_MT.UTF-8
...
$
```

Process termination

A single-threaded process can normally terminate in 3 ways:

- Executing a `return` from main
- Calling the `exit` function
- Calling `_exit` or `_Exit`

A call to `exit` performs basic shutdown steps and instructs the kernel to terminate the process: thus, `exit` does not return.

```c
#include <stdlib.h>

void exit(int status);
```

The `status` parameter denotes the process exit status, which can be checked by other processes, including the shell.

- Two macros, `EXIT_SUCCESS` and `EXIT_FAILURE`, are defined as portable ways to represent success and failure respectively.

Process exit code

Return command line argument as exit code:

- second argument (first, excluding binary name)
- convert argument (`char*`) to an integer
- return value from `main`

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

int main(int argc, char **argv) {
    // return a value only if a valid argument is passed
    if (argc >= 2)
        return atoi(argv[1]);
}
```

Process exit code

Test for a number of exit codes:

```
$ # compile and run exit code program
$ gcc -o exitcode exitcode.c
$ ./exitcode 32
$ # print program exit code
$ echo $?
32
$ ./exitcode 257
$ echo $?
1
$ ./exitcode -1
$ echo $?
255
$ # under C99, implicitly call exit(0)
$ ./exitcode
$ echo $?
0
$
```

Before terminating the process, the C library performs the following steps:

1. call user-defined clean-up functions, registered with `atexit` or `on_exit`, in the reverse order of their registration.
2. flush all open standard I/O streams
3. remove any temporary files created with `tmpfile`

Installing an exit handler:

```c
#include <stdio.h>
#include <stdlib.h>

static void exit_handler(void) {
    printf("exit_handler() called from exit()\n");
}

int main(int argc, char **argv) {
    if (atexit(exit_handler))
        perror("Cannot register exit handler");

    printf("Hello, world!\n");
    return 0;
}
```

Process exit handlers

Execute the exit handler example code:

```
$ gcc -o exit_handler exit_handler.c
$ ./exit_handler
Hello, world!
exit_handler() called from exit()
$
```

Execute the exit handler example code:

```
$ gcc -o exit_handler exit_handler.c
$ ./exit_handler
Hello, world!
exit_handler() called from exit()
$
```

Exercise

Write a program with multiple exit handlers, to verify the order in which they are called.

Launching a program

In Unix, running a program is a two-step procedure:

1. Create a new process, a near identical duplicate of the parent;
2. Load program binary into memory, replacing process address space, and begin execution.

Running programs

In Unix, running a program is a two-step procedure:

1. Create a new process, a near identical duplicate of the parent;
2. Load program binary into memory, replacing process address space, and begin execution.

Note

These two steps are referred to *forking* [a process] and *executing* [a new program] respectively.

EXEC

The execution of a program is accomplished through the `exec` family of functions:

- completely replace a process with the new program (text, data, heap and stack segments are replaced)
- new program starts executing its `main` function
- PID does not change because no new process is being created

Exec family

```c
#include <unistd.h>

int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ..., char *const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[], char *const envp[]);
```

Note

The `exec` function variants only return if an error has occurred. The return value is `-1`, and `errno` is set to indicate the error.

The naming convention of the exec family of functions reflects the arguments the particular function takes:

Function	`path`	`file`	args	`argv[]`	`environ`	`envp[]`
`execl`	·		·		·	
`execlp`		·	·		·	
`execle`	·		·			·
`execv`	·			·	·	
`execvp`		·		·	·	
`execve`	·			·		·
		`p` (path)	`l` (list)	`v` (vector)		`e` (env)

Table 1: Family of `exec` functions

EXECL

```c
#include <unistd.h>

int execl(const char *path, const char *arg, ...);
```

`execl` is a *variadic* function (takes a variable number of arguments)

- current process image is replaced with binary specified by `path`
- arguments are passed to the program via consecutive `arg`[s]
- argument list should be terminated by `NULL` (passed as the last argument to the function)

EXECL

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

int main(int argc, char **argv) {
    printf("Running execl()...\n");

    // call execl followed by argument list and NULL
    if (execl("/bin/ls", "ls", NULL)) {
        perror("execl failed:");
        exit(EXIT_FAILURE);
    }

    // never executes
    printf("This message will never be shown.\n");
}
```

EXECVP

```c
#include <unistd.h>

int execvp(const char *file, char *const argv[]);
```

`execvp` differs from `execl` in two major ways:

- if the filename specified does not contain a slash (/) character, the executable is sought in the colon-separated list of directory pathnames specified in the PATH environment variable;
- arguments to the new program are passed as an array of pointers to null-terminated strings; the array of pointers must be terminated by a NULL pointer.

EXECVP

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

int main(int argc, char **argv) {
    printf("Running execvp()...\n");

    // prepare args list to be passed as an array
    char * const args[] = {
        "ls", "-la", NULL
    };

    // execute ls (search PATH for match); pass args to ls
    if (execvp("ls", args)) {
        perror("execvp failed:");
        exit(EXIT_FAILURE);
    }

    // never executes
    printf("This message will never be shown.\n");
}
```

```c
#include <unistd.h>

int execvpe(const char *file, char *const argv[], char *const env[]);
```

`execvpe` also passes a list of environment variables to the program being executed (`char *const env[]`).

EXECVPE

```c
#include <unistd.h>

int execvpe(const char *file, char *const argv[], char *const env[]);
```

`execvpe` also passes a list of environment variables to the program being executed (`char *const env[]`).

Note

The `execvpe` function is a GNU extension that first appeared in glibc 2.11. The compiler will give a warning if `_GNU_SOURCE` is not defined before inclusion.

EXECVPE

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char **argv) {
    printf("Running execvpe()...\n");

    // make sure we have the correct number of args
    if (argc < 2) {
        printf("No program image supplied!\n");
        exit(EXIT_FAILURE);
    }

    // environment
    char *const env[] = {"HOME=/user/onionbro", "USER=onionbro", NULL};

    // call exec with one less argument (arg[0])
    if (execvpe(argv[1], argv + 1, env)) {
        perror("execvpe failed");
        exit(EXIT_FAILURE);
    }
}
```

EXECVPE

Execute the program that prints arguments and environment strings (`argenv`) through `execvpe`:

```
$ gcc -o execvpe execvpe_arg.c
$ ./execvpe ./argenv one two 3 4 5
Running execvpe()...
argv[0]=./argenv
argv[1]=one
argv[2]=two
argv[3]=3
argv[4]=4
argv[5]=5
environ[0]=HOME=/user/onionbro
environ[1]=USER=onionbro
$
```

Execute the program that prints arguments and environment strings (`argenv`) through `execvpe`:

```
$ gcc -o execvpe execvpe_arg.c
$ ./execvpe ./argenv one two 3 4 5
Running execvpe()...
argv[0]=./argenv
argv[1]=one
argv[2]=two
argv[3]=3
argv[4]=4
argv[5]=5
environ[0]=HOME=/user/onionbro
environ[1]=USER=onionbro
$
```

Note

The code for `argenv.c` can be found in an earlier section of this slide set.

EXECVPE

What happens if we specify just the binary filename, without any path qualifying characters?

What happens if we specify just the binary filename, without any path qualifying characters?

```
$ ./execvpe argenv
Running execvpe()...
execvpe failed:: No such file or directory
$
```

What happens if we specify just the binary filename, without any path qualifying characters?

```
$ ./execvpe argenv
Running execvpe()...
execvpe failed:: No such file or directory
$
```

Note

The `exec` function variants with path resolution only attempt to resolve paths for filenames that do not contain a slash character.

Exec path resolution

Running the program again through `strace` (trace system calls and signals) and narrowing the output to `exec`, we observe the following:

```
$ strace ./execvpe argenv
execve("./execvpe", ["./execvpe", "argenv"], [/* 65 vars */]) = 0
...
execve("/home/keith/bin/argenv", ["argenv"], [/* 2 vars */]) = -1 ENOENT (No such file or directory)
execve("/usr/local/sbin/argenv", ["argenv"], [/* 2 vars */]) = -1 ENOENT (No such file or directory)
execve("/usr/local/bin/argenv", ["argenv"], [/* 2 vars */]) = -1 ENOENT (No such file or directory)
execve("/usr/sbin/argenv", ["argenv"], [/* 2 vars */]) = -1 ENOENT (No such file or directory)
execve("/usr/bin/argenv", ["argenv"], [/* 2 vars */]) = -1 ENOENT (No such file or directory)
execve("/sbin/argenv", ["argenv"], [/* 2 vars */]) = -1 ENOENT (No such file or directory)
execve("/bin/argenv", ["argenv"], [/* 2 vars */]) = -1 ENOENT (No such file or directory)
execve("/snap/bin/argenv", ["argenv"], [/* 2 vars */]) = -1 ENOENT (No such file or directory)
...
+++ exited with 1 +++
$
```

In Linux, only `execve` is a system call:

- rest are wrappers in the C library around `execve`
- variadic system calls are difficult to implement
- concept of user's path exists solely in user space

Last error number (errno)

A word on `errno`:

- an integer variable set by system calls and some library functions in the event of an error, to indicate what went wrong.
- defined in `<errno.h>`
- significant only when the return value of the call indicates an error (i.e., `-1` from most system calls, `-1` or `NULL` from most library functions)
- the value is never set to zero by any system call or library function

Last error number (errno)

A word on `errno`:

- an integer variable set by system calls and some library functions in the event of an error, to indicate what went wrong.
- defined in `<errno.h>`
- significant only when the return value of the call indicates an error (i.e., `-1` from most system calls, `-1` or `NULL` from most library functions)
- the value is never set to zero by any system call or library function

Note

The `perror` function produces a message on standard error describing the last error encountered during a call to a system or library function.

FORK

Forking a process

The `fork` function creates a new process by duplicating the calling process.

Forking a process

The `fork` function creates a new process by duplicating the calling process.

- child has its own unique PID

Forking a process

The `fork` function creates a new process by duplicating the calling process.

- child has its own unique PID
- child's parent PID is the same as the PID of the calling process

Forking a process

The `fork` function creates a new process by duplicating the calling process.

- child has its own unique PID
- child's parent PID is the same as the PID of the calling process
- resource statistics of the child (e.g. CPU time counters) are reset to zero

Forking a process

The `fork` function creates a new process by duplicating the calling process.

- child has its own unique PID
- child's parent PID is the same as the PID of the calling process
- resource statistics of the child (e.g. CPU time counters) are reset to zero
- child does not inherit resources such as memory locks, timers, asynchronous I/O operations, pending signals and semaphore adjustments

Forking a process

The `fork` function creates a new process by duplicating the calling process.

- child has its own unique PID
- child's parent PID is the same as the PID of the calling process
- resource statistics of the child (e.g. CPU time counters) are reset to zero
- child does not inherit resources such as memory locks, timers, asynchronous I/O operations, pending signals and semaphore adjustments
- entire virtual address space of parent is replicated in the child, including pthreads objects such as mutexes

The `fork` function creates a new process by duplicating the calling process.

- child has its own unique PID
- child's parent PID is the same as the PID of the calling process
- resource statistics of the child (e.g. CPU time counters) are reset to zero
- child does not inherit resources such as memory locks, timers, asynchronous I/O operations, pending signals and semaphore adjustments
- entire virtual address space of parent is replicated in the child, including pthreads objects such as mutexes
- child inherits copies of the parent's set of open file descriptors, open message queue descriptors and open directory streams

FORK

The `fork` function creates a new process by duplicating the calling process.

```c
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

`fork` takes no arguments

- if successful, it creates a new process, identical in almost all aspects to the caller
- both processes continue to run, returning from `fork()`
- in the child process, `fork` returns `0`
- in parent process, `fork` returns the PID of child

If the `fork` call fails, it returns `-1` and sets the `errno` to one of the following values:

`errno`	Description
`EAGAIN`	The kernel failed to allocate certain resources.
`ENOMEM`	Insufficient kernel memory was available to complete the request.

Table 2: Possible `fork` error codes

FORK EXAMPLE

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

int main(int argc, char **argv) {
    printf("Parent process before fork()\n");

    pid_t pid = fork();

    // parent process
    if (pid > 0) {
        printf("This is the parent process! PID is [%d]\n", pid);
    // child process
    } else if (pid == 0) {
        printf("This is the child process! PID is [%d]\n", pid);
    // error
    } else {
        perror("fork() failed");
        exit(EXIT_FAILURE);
    }
}
```

Running the `fork` example program returns a non-zero PID for the parent and zero for the child:

```
$ gcc -o fork fork.c
$ ./fork
Parent process before fork()
This is the parent process! PID is [49911]
This is the child process! PID is [0]
$
```

FORK EXAMPLE

Running the `fork` example program returns a non-zero PID for the parent and zero for the child:

```
$ gcc -o fork fork.c
$ ./fork
Parent process before fork()
This is the parent process! PID is [49911]
This is the child process! PID is [0]
$
```

…provided the call to `fork` doesn't fail!

When a process needs to launch a new program without replacing itself, the *fork-plus-exec* pattern is used:

- a call to `fork` spawns a new child process
- a subsequent call to `exec` from the child replaces it with the desired program binary image

When a process needs to launch a new program without replacing itself, the *fork-plus-exec* pattern is used:

- a call to `fork` spawns a new child process
- a subsequent call to `exec` from the child replaces it with the desired program binary image

Note

Alternatively, on POSIX-compliant Unix and Unix-like systems, one may use `posix_spawn`.

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

int main(int argc, char **argv) {
    pid_t pid = fork();
    if (pid == -1) { // check for error
        perror("fork() failed");
        exit(EXIT_FAILURE);
    } else if (pid == 0) { // child pid, exec ps
        if (execlp("ps", "ps", "-f", NULL)) {
            perror("execlp() failed");
            exit(EXIT_FAILURE);
        }
        // dead code
        printf("This string should never get printed\n");
    }
    // print child PID - should match PID of ps
    printf("Parent process after fork(); child PID is [%d]\n", pid);
}
```

FORK-PLUS-EXEC

The example forks a child process and then executes `ps`:

```
$ gcc -o forkexec forkexec.c
$ ./forkexec
Parent process before fork()
Parent process after fork(); child PID is [50493]
UID         PID   PPID  C STIME TTY          TIME CMD
keith     30842   1927  0 03:52 pts/0    00:00:00 /bin/bash
keith     50493      1  0 15:02 pts/0    00:00:00 ps -f
$
```

FORK SEMANTICS

The `fork` function duplicates the memory address space of the parent process

- in early Unix systems, the kernel created copies of all internal data structures, and copied the memory of the parent process into the child's address space
- this is naïve and wasteful when using fork-plus-exec – parent's address spaces is replicated in child only to be discarded immediately after

VFORK

Unix designers concerned with the wasteful address space copy during fork-plus-exec developed `vfork`:

- the calling process is suspended until the child terminates (normally, by calling `_exit` or abnormally, after the delivery of a fatal signal), or it makes a call to `execve`
- until that point, child shares all memory with the parent, including the stack
- the child must **not** return from the current function or call `exit`
- otherwise, behaviour is similar to `fork`

```c
#include <sys/types.h>
#include <unistd.h>

pid_t vfork(void);
```

Copy-on-write

Modern Unix systems employ a technique call *copy-on-write* (COW) in the `fork` implementation.

- a lazy optimisation strategy designed to mitigate the overhead of duplicating resources
- resources (memory pages) are only copied if a process attempts to modify *its* copy
- modern memory management units (MMUs) provide hardware-level support for copy-on-write

Copy-on-write

Modern Unix systems employ a technique call *copy-on-write* (COW) in the `fork` implementation.

- a lazy optimisation strategy designed to mitigate the overhead of duplicating resources
- resources (memory pages) are only copied if a process attempts to modify *its* copy
- modern memory management units (MMUs) provide hardware-level support for copy-on-write

Note

Copy-on-write reduces copy overhead in fork-plus-exec scenarios.

(Modern) `fork` versus `vfork`

The 4.2BSD man page stated: `"This system call will be eliminated when proper system sharing mechanisms are implemented. Users should not depend on the memory sharing semantics of vfork() as it will, in that case, be made synonymous to fork(2)."`

(Modern) `fork` versus `vfork`

The 4.2BSD man page stated: `"This system call will be eliminated when proper system sharing mechanisms are implemented. Users should not depend on the memory sharing semantics of vfork() as it will, in that case, be made synonymous to fork(2)."`

The child process should take care not to modify the memory in unintended ways

FORK VERSUS VFORK

(Modern) `fork` versus `vfork`

The 4.2BSD man page stated: `"This system call will be eliminated when proper system sharing mechanisms are implemented. Users should not depend on the memory sharing semantics of vfork() as it will, in that case, be made synonymous to fork(2)."`

The child process should take care not to modify the memory in unintended ways

- changes will be seen by the parent when it is given back control

(Modern) `fork` versus `vfork`

The 4.2BSD man page stated: `"This system call will be eliminated when proper system sharing mechanisms are implemented. Users should not depend on the memory sharing semantics of vfork() as it will, in that case, be made synonymous to fork(2)."`

The child process should take care not to modify the memory in unintended ways

- changes will be seen by the parent when it is given back control
- may result in inconsistent process state w.r.t. parent process (e.g. during signal handling by the child)

(Modern) `fork` versus `vfork`

Notwitstanding, `vfork` still has some advantages over `fork` in fork-plus-exec scenarios:

(Modern) `fork` versus `vfork`

Notwitstanding, `vfork` still has some advantages over `fork` in fork-plus-exec scenarios:

- performance critical applications benefit from the small performance advantage (since no page tables are copied)

(Modern) `fork` versus `vfork`

Notwitstanding, `vfork` still has some advantages over `fork` in fork-plus-exec scenarios:

- performance critical applications benefit from the small performance advantage (since no page tables are copied)
- it can be implemented on systems that lack an MMU with copy-on-write support

(Modern) `fork` versus `vfork`

Notwitstanding, `vfork` still has some advantages over `fork` in fork-plus-exec scenarios:

- performance critical applications benefit from the small performance advantage (since no page tables are copied)
- it can be implemented on systems that lack an MMU with copy-on-write support
- can be used on memory-constrained systems, since memory is not overcommitted (e.g. a large program that wants to fork and execute a small program)

WAIT

Process termination

When a process terminates the kernel notifies the parent by sending the `SIGCHLD` signal

- parent can elect to handle this signal; by default it is ignored
- child termination is asynchronous w.r.t. parent; signal may be dispatched at any time

Often, the parent wants to explicitly wait for its child's termination

- parent processes might want to obtain information pertaining to the termination of a child process
- this goes beyond receiving a signal on termination: for instance, getting the child's return value

In Unix, when a child dies before its parent, it's put in a special state:

- process becomes a *zombie*
- only a minimal skeleton of the process is retained
- zombie process ceases to exist when parent inquires about its state (process is reaped)

WAIT

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

pid_t wait(int *status);
```

The Linux kernel provides a number of interfaces for querying information about terminated child processes, such as `wait`

- if `wait` is successful, it returns the PID of a terminated child; otherwise, it returns `-1`, signifying that an error has occurred

A process that calls `wait` can:

- block if all its children are still running
- return immediately with the termination status of a child
- return immediately with an error if it doesn't have any child processes

WAIT

A process that calls `wait` can:

- block if all its children are still running
- return immediately with the termination status of a child
- return immediately with an error if it doesn't have any child processes

Note

Calling `wait` in response to `SIGCHLD` will always return without blocking.

WAIT EXAMPLE

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

int main(int argc, char **argv) {
    // fork child process and execute "ps -f"
    pid_t pid = fork();
    if (pid == -1) {
        perror("fork() failed");
        exit(EXIT_FAILURE);
    } else if (pid == 0) {
        if (execlp("ps", "ps", "-f", NULL)) {
            perror("execlp() failed");
            exit(EXIT_FAILURE);
        }
    }
    // print child PID - should match PID of ps
    printf("Parent process after fork(); child PID is [%d]\n", pid);
    // wait for child process to terminate
    int status;
    if (wait(&status) == -1) {
        perror("wait() failed");
        exit(EXIT_FAILURE);
    }
    // message is shown after child terminates
    printf("Parent process after wait()\n");
}
```

WAIT EXAMPLE

Running the example, note how the `ps` executed by the child will list the parent process (`forkexecwait`) as still running:

```
$ gcc -o forkexecwait forkexecwait.c
$ ./forkexecwait
Parent process after fork(); child PID is [81483]
UID          PID    PPID  C STIME TTY          TIME CMD
keith      30842    1927  0 Mar01 pts/0    00:00:01 /bin/bash
keith      81482   30842  0 16:14 pts/0    00:00:00 ./forkexecwait
keith      81483   81482  0 16:14 pts/0    00:00:00 ps -f
Parent process after wait()
$
```

Running the example, note how the `ps` executed by the child will list the parent process (`forkexecwait`) as still running:

```
$ gcc -o forkexecwait forkexecwait.c
$ ./forkexecwait
Parent process after fork(); child PID is [81483]
UID          PID    PPID  C STIME TTY          TIME CMD
keith      30842    1927  0 Mar01 pts/0    00:00:01 /bin/bash
keith      81482   30842  0 16:14 pts/0    00:00:00 ./forkexecwait
keith      81483   81482  0 16:14 pts/0    00:00:00 ps -f
Parent process after wait()
$
```

Note

The parent process resumes execution only once the child process terminates and is reaped by `wait`.

WAIT VARIANTS

There are a number of variants of `wait` such as `wait3` and `wait4` which provide a summary of the resource usage of a process

- these are BSD style functions
- `man wait3`

WAIT VARIANTS

There are a number of variants of `wait` such as `wait3` and `wait4` which provide a summary of the resource usage of a process

- these are BSD style functions
- `man wait3`

Another useful variant allows waiting for a specific process:

- a process might have multiple children but wants to wait for a specific one

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

pid_t waitpid(pid_t pid, int *status, int options);
```

WAITPID

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

pid_t waitpid(pid_t pid, int *status, int options);
```

The `pid` parameter specifies exactly which process(es) to wait for:

- `< -1` wait for any child in the given process group
- `-1` wait for any child process (same as `wait`)
- `0` wait for any child in the same process group as caller
- `> 0` wait for the process with the exact `pid` value

WAITPID

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

pid_t waitpid(pid_t pid, int *status, int options);
```

If successful, `waitpid` returns the PID of the process

- if `options` specifies `WNOHANG`, the call does not block;
- if the specified child or children have not yet changed state and the call is non-blocking, then the function returns `0`.

On error, the method returns `-1` and sets `errno`.

WAIT EXIT STATUS

The `status` value returned by `wait` functions, encodes all sorts of useful information about the child process state changes:

- may be queried using macros defined in `sys/wait.h`
- examples include:

  `WIFEXITED`, which returns a non-zero value if the child process terminated normally with `exit` or `_exit`

  `WEXITSTATUS`, which returns the low-order 8 bits of the exit status value from the child process

- see `wait` manpage for a comprehensive list of these macros

Exercise

Use the *fork-plus-exec* pattern and `waitpid` to write a simple command interpreter

- Use `linenoise` to read user input
  `https://github.com/antirez/linenoise`

Questions?