}

Get Process ID

Always successfully returns the process ID of caller.

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

int main() {
    pid_t pid = getpid();
    printf("%d\n", pid);

    return 0;
}
```

- The Linux kernel limits process IDs to being less than or equal to 32,767.
- Each time the limit of 32,767 is reached, the kernel resets its process ID counter.
- Once it has reached 32,767, the process ID counter is reset to 300, rather than 1.
- This is done because many low-numbered process IDs are in permanent use by system processes and daemons.
- In Linux 2.4 and earlier, the process ID limit of 32,767 is defined by the kernel constant PID_MAX.
- With Linux 2.6, this limit is adjustable via the value in the Linux-specific `/proc/sys/kernel/pid_max` file. On 32-bit platforms, the maximum value for this file is 32,768, but on 64-bit platforms, it can be adjusted to any value up to 2^{22} (4194304).

Get Parent Process ID

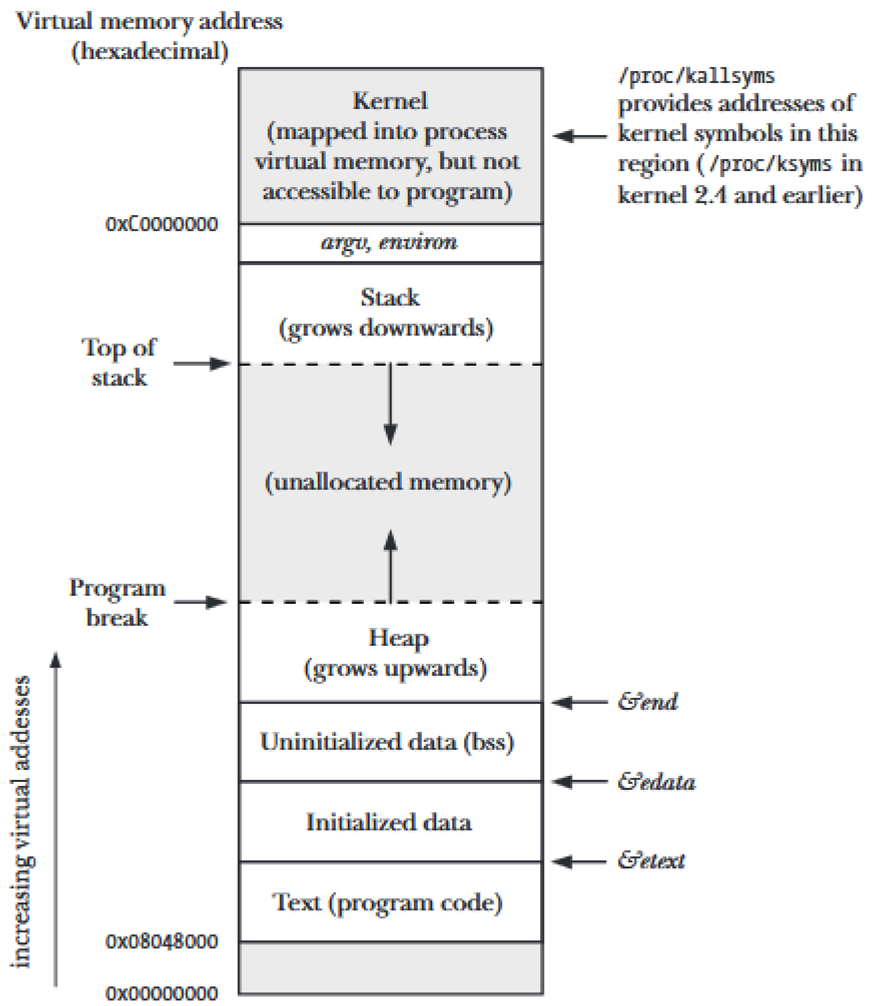
Always successfully returns the process ID of the parent of the caller.

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

int main() {
    pid_t ppid = getppid();
    printf("%d\n", ppid);

    return 0;
}
```

- The parent of each process has its own parent, and so on, going all the way back to process 1, init, the ancestor of all processes.

- If a child process becomes orphaned because its “birth” parent terminates, then the child is adopted by the init process, and subsequent calls to getppid() in the child return 1.
- The parent of any process can be found by looking at the Ppid field provided in the Linux-specific `/proc/PID/status` file.

Virtual Memory Management



Virtual memory address (hexadecimal)
Kernel (mapped into process virtual memory, but not accessible to program)
/proc/kallsyms provides addresses of kernel symbols in this region (/proc/ksyms in kernel 2.4 and earlier)
0xC0000000
argv, environ
Stack (grows downwards)
Top of stack
(unallocated memory)
Program break
Heap (grows upwards)
&end
Uninitialized data (bss)
&edata
Initialized data
&etext
Text (program code)
increasing virtual addresses
0x08048000
0x00000000

Figure 6-1: Typical memory layout of a process on Linux/x86-32

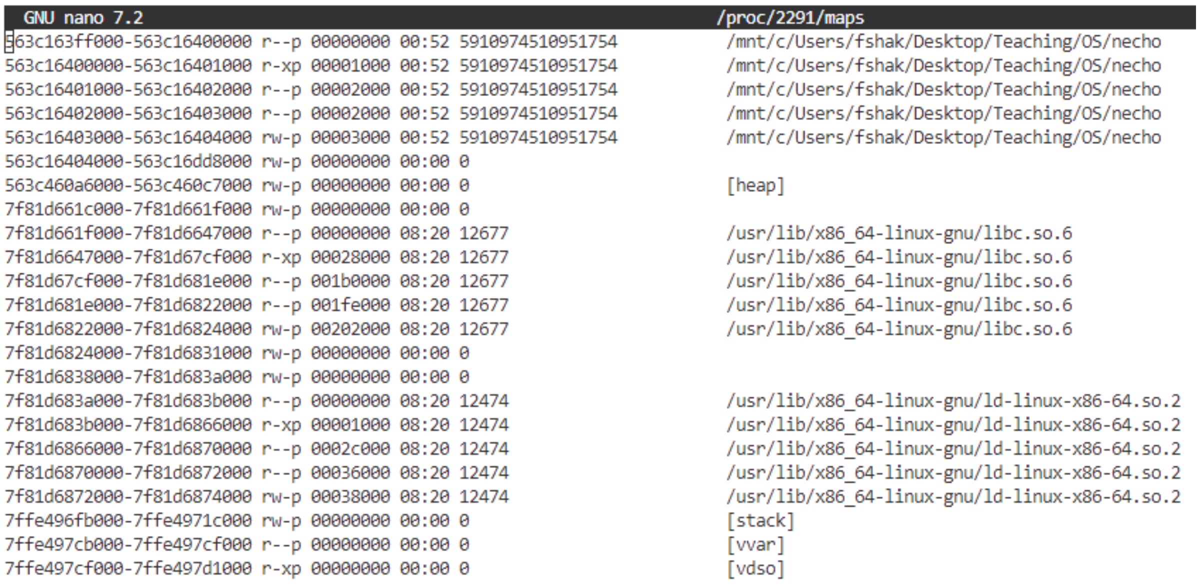
Let’s print memory locations of different variables–

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

char globBuf[65536];         /* Uninitialized data segment */
int primes[] = {2, 3, 5, 7}; /* Initialized data segment */

static int square(int x) /* Allocated in frame for square() */ {
    int result; /* Allocated in frame for square() */
```

```c
    result = x * x;
    return result; /* Return value passed via register */
}

static void doCalc(int val) /* Allocated in frame for doCalc() */ {
    printf("The square of %d is %d\n", val, square(val));
    if (val < 1000)
    {
        int t; /* Allocated in frame for doCalc() */
        t = val * val * val;
        printf("The cube of %d is %d\n", val, t);
    }
}

int main(int argc, char *argv[]) /* Allocated in frame for main() */{
    static int key = 9973;      /* Initialized data segment */
    static char mbuf[10240000]; /* Uninitialized data segment */
    char *p;                    /* Allocated in frame for main() */
    p = malloc(1024);           /* Points to memory in heap segment */

    printf("\nMemory addresses of variables and segments:\n");
    printf("---------------------------------------------------\n");
    printf("Address of function main() (Text): %p\n", (void *)main);
    printf("Address of function doCalc() (Text): %p\n", (void *)doCalc);
    printf("Address of globBuf (BSS): %p\n", (void *)globBuf);
    printf("Address of primes (Initialized Data): %p\n", (void *)primes);
    printf("Address of key (Initialized Data): %p\n", (void *)&key);
    printf("Address of mbuf (BSS): %p\n", (void *)mbuf);
    printf("Address of malloc'd memory (Heap): %p\n", (void *)p);
    printf("Address of local variable p (Stack): %p\n", (void *)&p);

    printf("\nTo see segment boundaries, check /proc/%d/maps\n", getpid());

    doCalc(key);

    printf("\nPress Enter to continue...\n");
    getchar(); // Wait for user input to proceed

    free(p);
    return EXIT_SUCCESS;
}
```

Now, let’s see the memory map by reading the file `/proc/<PID>/maps`

```
nano /proc/<PID>/maps
```

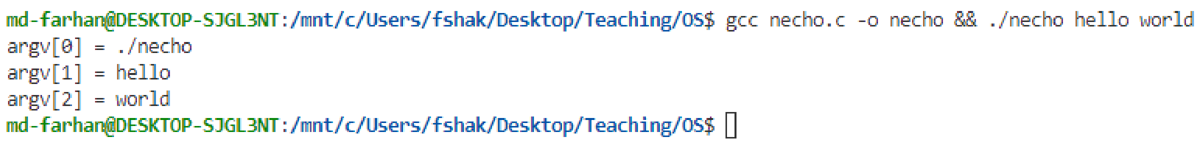


```
  GNU nano 7.2                                                        /proc/2291/maps
563c163ff000-563c16400000 r--p 00000000 00:52 591097451095175 4          /mnt/c/Users/fshak/Desktop/Teaching/OS/necho
563c16400000-563c16401000 r-xp 00001000 00:52 591097451095175 4          /mnt/c/Users/fshak/Desktop/Teaching/OS/necho
563c16401000-563c16402000 r--p 00002000 00:52 591097451095175 4          /mnt/c/Users/fshak/Desktop/Teaching/OS/necho
563c16402000-563c16403000 r--p 00002000 00:52 591097451095175 4          /mnt/c/Users/fshak/Desktop/Teaching/OS/necho
563c16403000-563c16404000 rw-p 00003000 00:52 591097451095175 4          /mnt/c/Users/fshak/Desktop/Teaching/OS/necho
563c16404000-563c16dd8000 rw-p 00000000 00:00 0
563c460a6000-563c460c7000 rw-p 00000000 00:00 0                          [heap]
7f81d661c000-7f81d661f000 rw-p 00000000 00:00 0
7f81d661f000-7f81d6647000 r--p 00000000 08:20 12677                      /usr/lib/x86_64-linux-gnu/libc.so.6
7f81d6647000-7f81d67cf000 r-xp 00028000 08:20 12677                      /usr/lib/x86_64-linux-gnu/libc.so.6
7f81d67cf000-7f81d681e000 r--p 001b0000 08:20 12677                      /usr/lib/x86_64-linux-gnu/libc.so.6
7f81d681e000-7f81d6822000 r--p 001fe000 08:20 12677                      /usr/lib/x86_64-linux-gnu/libc.so.6
7f81d6822000-7f81d6824000 rw-p 00202000 08:20 12677                      /usr/lib/x86_64-linux-gnu/libc.so.6
7f81d6824000-7f81d6831000 rw-p 00000000 00:00 0
7f81d6838000-7f81d683a000 rw-p 00000000 00:00 0
7f81d683a000-7f81d683b000 r--p 00000000 08:20 12474                      /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f81d683b000-7f81d6866000 r-xp 00001000 08:20 12474                      /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f81d6866000-7f81d6870000 r--p 0002c000 08:20 12474                      /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f81d6870000-7f81d6872000 r--p 00036000 08:20 12474                      /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f81d6872000-7f81d6874000 rw-p 00038000 08:20 12474                      /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7ffe496fb000-7ffe4971c000 rw-p 00000000 00:00 0                          [stack]
7ffe497cb000-7ffe497cf000 r--p 00000000 00:00 0                          [vvar]
7ffe497cf000-7ffe497d1000 r-xp 00000000 00:00 0                          [vdso]
```

Task: Check whether or not all variable locations fall in the range or not?

Command-Line Arguments

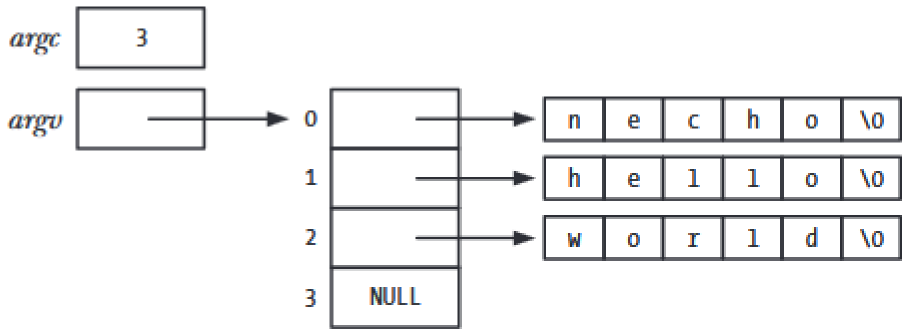
```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int j;
    for (j = 0; j < argc; j++)
        printf("argv[%d] = %s\n", j, argv[j]);
    exit(EXIT_SUCCESS);
}
```



```
md-farhan@DESKTOP-SJGL3NT:/mnt/c/Users/fshak/Desktop/Teaching/OS$ gcc necho.c -o necho && ./necho hello world
argv[0] = ./necho
argv[1] = hello
argv[2] = world
md-farhan@DESKTOP-SJGL3NT:/mnt/c/Users/fshak/Desktop/Teaching/OS$
```

- The first argument, `int argc`, indicates how many command-line arguments there are.
- The second argument, `char *argv[]`, is an array of pointers to the command-line arguments, each of which is a null-terminated character string.

- The first of these strings, in `argv[0]`, is (conventionally) the name of the program itself. The list of pointers in `argv` is terminated by a NULL pointer (i.e., `argv[argc]` is NULL).



Since the argv list is terminated by a NULL value, we could alternatively code the body of the program as follows, to output just the command-line arguments one per line:

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

int main(int argc, char *argv[])
{
    char **p;
    for (p = argv; *p != NULL; p++)
        puts(*p);
    exit(EXIT_SUCCESS);
}
```

One limitation of the argc/argv mechanism is that these variables are available only as arguments to main(). To portably make the command-line arguments available in other functions, we must either pass argv as an argument to those functions or set a global variable pointing to argv.
The command-line arguments of any process can be read via the Linux-specific `/proc/PID/cmdline` file, with each argument being terminated by a null byte. (A program can access its own command-line arguments via `/proc/self/cmdline`.)

Environment List

Each process has an associated array of strings called the environment list, or simply the environment. Each of these strings is a definition of the form name=value.

```
$ export SHELL=/bin/bash
```

The above commands permanently add a value to the shell’s environment, and this environment is then inherited by all child processes that the shell creates. At any point, an environment variable can be removed with the `unset` command.

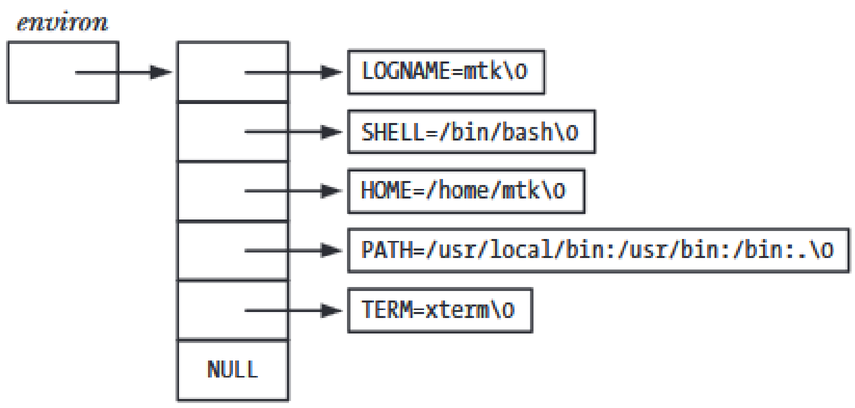
```
$ unset VAR_NAME
```

The `printenv` command displays the current environment list.

```
$ printenv
```

The environment list of any process can be examined via the Linux-specific `/proc/ PID/environ` file, with each NAME=value pair being terminated by a null byte.

Accessing the environment from a program



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

extern char **environ;

int main(int argc, char *argv[]) {
    char **ep;
    for (ep = environ; *ep != NULL; ep++)
        puts(*ep);
    return EXIT_SUCCESS;
}
```

The getenv() function retrieves individual values from the process environment. Returns pointer to (value) string, or NULL if no such variable.

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

int main(int argc, char *argv[]) {
    if (getenv("SHELL") != NULL)
        puts(getenv("SHELL"));
    return EXIT_SUCCESS;
}
```

The putenv() function adds a new variable to the calling process's environment or modifies the value of an existing variable. Returns 0 on success, or nonzero on error.

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

int main(int argc, char *argv[]) {
    if (!putenv("TEXT=hello"))
        puts(getenv("TEXT"));
    return EXIT_SUCCESS;
}
```

The unsetenv() function removes the variable identified by name from the environment. Returns 0 on success, or –1 on error.

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

int main(int argc, char *argv[]) {
    // set and check if saved
    if (!putenv("TEXT=hello"))
        puts(getenv("TEXT"));

    // unset and check if saved
    if (!unsetenv("TEXT")) {
        if (getenv("TEXT") != NULL)
            puts(getenv("TEXT"));
        else
            puts("Not found");
    }
    return EXIT_SUCCESS;
}
```

On occasion, it is useful to erase the entire environment, and then rebuild it with selected values/ We can erase the environment by assigning NULL to `environ`. This is exactly the step performed by the `clearenv()` library function. Returns 0 on success, or a nonzero on error.

Following code demonstrates the use of all of the functions discussed in this section.

```c
#define _GNU_SOURCE /* To get various declarations from <stdlib.h> */
#include <stdio.h>
#include <stdlib.h>

extern char **environ;

void errExit(const char *msg, const char *arg) {
    fprintf(stderr, msg, arg);
    fprintf(stderr, "\n");
    exit(EXIT_FAILURE);
}

int main(int argc, char *argv[]) {
    int j;
    char **ep;

    clearenv();

    for (j = 1; j < argc; j++) {
        if (putenv(argv[j]) != 0)
        {
            errExit("putenv: %s", argv[j]);
        }
    }

    if (setenv("GREET", "Hello world", 0) == -1) {
        errExit("setenv", NULL);
    }

    unsetenv("BYE");
    for (ep = environ; *ep != NULL; ep++) {
        puts(*ep);
    }

    exit(EXIT_SUCCESS);
}
```

fork(), exit(), wait(), and execve()

- The `fork()` system call allows one process, the parent, to create a new process, the child. This is done by making the new child process an (almost) exact duplicate of the parent: the child obtains copies of the parent's stack, data, heap, and text segments.
- The `exit(status)` library function terminates a process, making all resources (memory, open file descriptors, and so on) used by the process available for subsequent reallocation by the kernel. The status argument is an integer that determines the termination status for the process.
- The `wait(&status)` system call has two purposes. First, if a child of this process has not yet terminated by calling `exit()`, then `wait()` suspends execution of the process until one of its children has terminated. Second, the termination status of the child is returned in the status argument of `wait()`.
- The `execve(pathname, argv, envp)` system call loads a new program (pathname, with argument list `argv`, and environment list `envp`) into a process's memory. The existing program text is discarded, and the stack, data, and heap segments are freshly created for the new program.

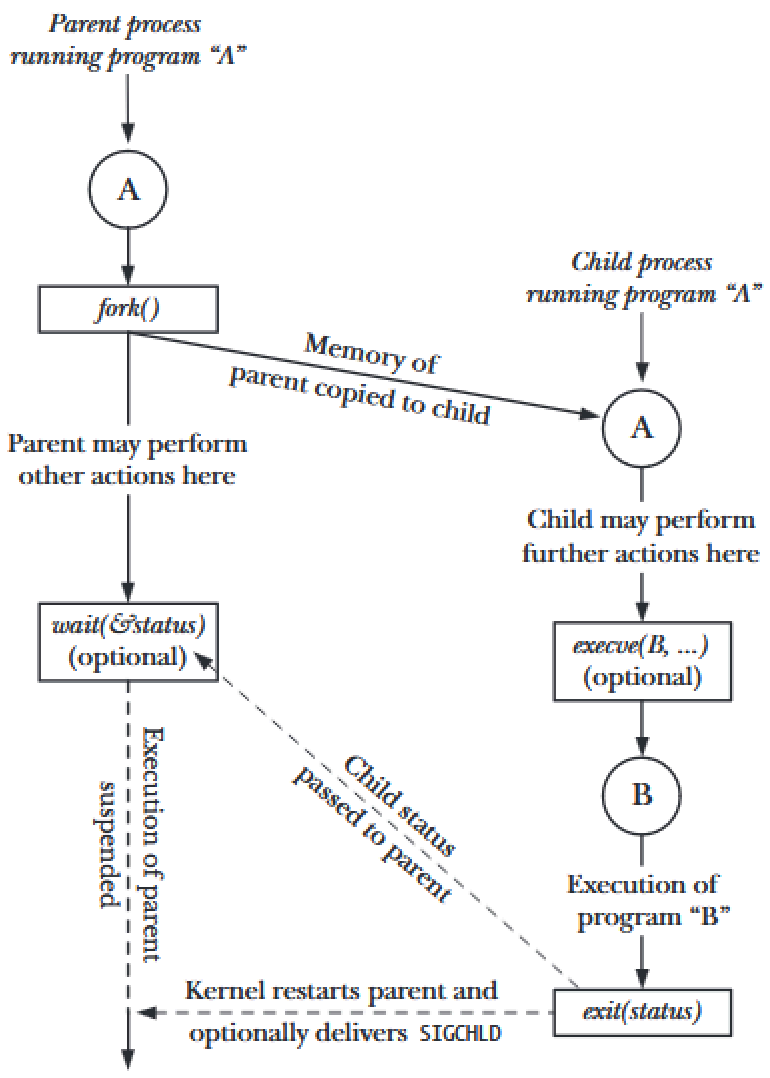

Parent process running program "A"
A
fork()
Memory of parent copied to child
Child process running program "A"
A
Parent may perform other actions here
Child may perform further actions here
wait(&status) (optional)
execve(B, ...) (optional)
B
Execution of parent suspended
Child status passed to parent
Execution of program "B"
Kernel restarts parent and optionally delivers SIGCHLD
exit(status)

Figure 24-1: Overview of the use of *fork()*, *exit()*, *wait()*, and *execve()*

Creating a New Process: fork()

In parent: returns process ID of child on success, or –1 on error; in successfully created child: always returns 0.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <errno.h>
#include <string.h>

static void errExit(const char *msg) {
    perror(msg);
    exit(EXIT_FAILURE);
```

```c
}

static int idata = 111; /* Allocated in data segment */

int main(int argc, char *argv[]) {
    int istack = 222; /* Allocated in stack segment */
    pid_t childPid;

    switch (childPid = fork()) {
        case -1:
            errExit("fork");
        case 0: /* Child process */
            idata *= 3;
            istack *= 3;
            break;
        default: /* Parent process */
            sleep(3); /* Give child a chance to execute */
            break;
    }

    /* Both parent and child come here */
    printf("PID=%ld %s idata=%d istack=%d\n",
           (long) getpid(),
           (childPid == 0) ? "(child) " : "(parent)",
           idata, istack);
    exit(EXIT_SUCCESS);
}
```

File Sharing Between Parent and Child

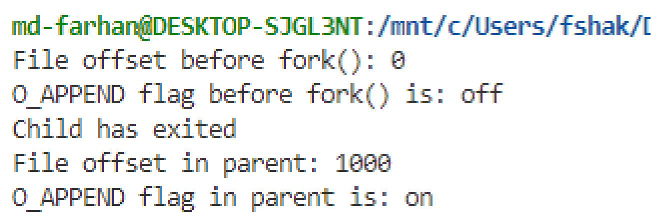
This program opens a temporary file using `mkstemp()`, and then calls `fork()` to create a child process. The child changes the file offset and open file status flags of the temporary file, and exits. The parent then retrieves the file offset and flags to verify that it can see the changes made by the child.

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

```c
#include <string.h>

// Define errExit function to handle errors
void errExit(const char *msg) {
    perror(msg);
    exit(EXIT_FAILURE);
}

int main(int argc, char *argv[]) {
    int fd, flags;
    char template[] = "/tmp/testXXXXXX";

    setbuf(stdout, NULL); // Disable buffering of stdout

    fd = mkstemp(template);
    if (fd == -1)
        errExit("mkstemp");

    printf("File offset before fork(): %lld\n", (long long)lseek(fd, 0,
SEEK_CUR));

    flags = fcntl(fd, F_GETFL);
    if (flags == -1)
        errExit("fcntl - F_GETFL");

    printf("O_APPEND flag before fork() is: %s\n", (flags & O_APPEND) ? "on" :
"off");

    switch (fork()) {
    case -1:
        errExit("fork");

    case 0: // Child: change file offset and status flags
        if (lseek(fd, 1000, SEEK_SET) == -1)
            errExit("lseek");

        flags = fcntl(fd, F_GETFL); // Fetch current flags
        if (flags == -1)
            errExit("fcntl - F_GETFL");

        flags |= O_APPEND; // Turn O_APPEND on
        if (fcntl(fd, F_SETFL, flags) == -1)
```

```
            errExit("fcntl - F_SETFL");

        _exit(EXIT_SUCCESS);

    default: // Parent: can see file changes made by child
        if (wait(NULL) == -1)
            errExit("wait"); // Wait for child exit

        printf("Child has exited\n");
        printf("File offset in parent: %lld\n", (long long)lseek(fd, 0,
SEEK_CUR));

        flags = fcntl(fd, F_GETFL);
        if (flags == -1)
            errExit("fcntl - F_GETFL");

        printf("O_APPEND flag in parent is: %s\n", (flags & O_APPEND) ? "on" :
"off");

        exit(EXIT_SUCCESS);
    }
}
```

The output will be like this



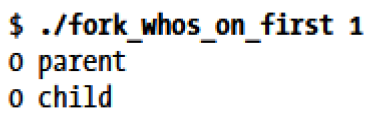
```
md-farhan@DESKTOP-SJGL3NT:/mnt/c/Users/fshak/[
File offset before fork(): 0
O_APPEND flag before fork() is: off
Child has exited
File offset in parent: 1000
O_APPEND flag in parent is: on
```

Race Conditions After fork()

After a fork(), it is indeterminate which process—the parent or the child—next has access to the CPU. We can use the program to demonstrate this indeterminacy. This program loops, using fork() to create multiple children. After each fork(), both parent and child print a message containing the loop counter value and a string indicating whether the process is the parent or child.

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

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

void errExit(const char *msg) {
    perror(msg); // Print system error message
    exit(EXIT_FAILURE);
}

void usageErr(const char *msg, const char *arg) {
    fprintf(stderr, msg, arg); // Print usage message
    exit(EXIT_FAILURE);
}

int getInt(const char *arg, int greaterThanZero, const char *name) {
    char *endptr;
    int value = strtol(arg, &endptr, 10);

    if (*endptr != '\0' || errno != 0 || (greaterThanZero && value <= 0)) {
        fprintf(stderr, "Invalid value for %s: %s\n", name, arg);
        exit(EXIT_FAILURE);
    }

    return value;
}

int main(int argc, char *argv[]) {
    int numChildren, j;
    pid_t childPid;

    if (argc > 1 && strcmp(argv[1], "--help") == 0)
        usageErr("%s [num-children]\n", argv[0]);

    numChildren = (argc > 1) ? getInt(argv[1], 1, "num-children") : 1;

    setbuf(stdout, NULL); // Make stdout unbuffered

    for (j = 0; j < numChildren; j++) {
        switch (childPid = fork()) {
        case -1:
            errExit("fork");
        case 0:
            printf("%d child\n", j);
            _exit(EXIT_SUCCESS);
        default:
            printf("%d parent\n", j);
            wait(NULL); // Wait for child to terminate
            break;
        }
    }

    exit(EXIT_SUCCESS);
}
```



```
$ ./fork_whos_on_first 1
0 parent
0 child
```

Try with 100, you will notice in some cases, the child ran before the parent.

Terminating a Process using exit()

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

int main() {
    int choice;

    printf("Choose an exit scenario:\n");
    printf("1. Normal exit (EXIT_SUCCESS)\n");
    printf("2. Failure exit (EXIT_FAILURE)\n");
    printf("3. Custom exit code (e.g., 42)\n");
    scanf("%d", &choice);

    switch (choice) {
        case 1:
            printf("Exiting with EXIT_SUCCESS (0)...\n");
            exit(EXIT_SUCCESS);
        case 2:
            printf("Exiting with EXIT_FAILURE (1)...\n");
            exit(EXIT_FAILURE);
        case 3:
            printf("Exiting with a custom exit code (42)...\n");
            exit(42);
        default:
            printf("Invalid choice! Exiting with EXIT_FAILURE...\n");
            exit(EXIT_FAILURE);
    }
}
```

Exercises:

1. Write a program that prints its process ID and parent process ID. Extend it to periodically print the parent process ID. What happens if you terminate the parent process?
2. Write a function that determines whether a given variable belongs to the stack, heap, or global segments.
3. Create a program that takes command-line arguments to:
    a. Reverse a string.
    b. Count vowels and consonants in a given string.
    c. Perform arithmetic operations like addition and multiplication.
4. Use the `fork()` example to open a file, write to it from the child, and read it from the parent. Synchronize their actions using `wait()` to ensure data consistency.
5. Create a program that spawns multiple child processes using `fork()`. Each child should try to update a shared counter in a loop. Introduce a delay (e.g., `sleep()`) to observe race conditions. Experiment with and without locks to resolve the issue.
6. Build a calculator where each arithmetic operation (e.g., addition, subtraction) spawns a new process to perform the calculation. The parent gathers results and displays them.