Creating a new process using *fork()*

fork() is very unusual because it returns *different* values in the (existing) parent process, and the (new) child process:

- the value returned by *fork()* in the parent process will be the process-indentifier, of process-ID, of the child process;
- the value returned by *fork()* in the child process will be 0, indicating that it is the child, because 0 is not a valid process-ID.

Each successful invocation of *fork()* returns a new monotonically increasing process-ID (the kernel 'wraps' the value back to the first unused positive value when it reaches 100,000).

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

void function(void)
{
    int  pid;                  // some systems define a pid_t

    switch (pid = fork()) {
    case -1 :
        printf("fork() failed\n");     // process creation failed
        exit(EXIT_FAILURE);
        break;

    case 0:                    // new child process
        printf("c:  value of pid=%i\n", pid);
        printf("c:  child's pid=%i\n", getpid());
        printf("c:  child's parent pid=%i\n", getppid());
        break;

    default:                   // original parent process
        sleep(1);
        printf("p:  value of pid=%i\n", pid);
        printf("p:  parent's pid=%i\n", getpid());
        printf("p:  parent's parent pid=%i\n", getppid());
        break;
    }
    fflush(stdout);
}
```

produces:

```
c:  child's value of pid=0
c:  child's pid=5642
c:  child's parent pid=5641
p:  parent's value of pid=5642
p:  parent's pid=5641
p:  parent's parent pid=3244
```

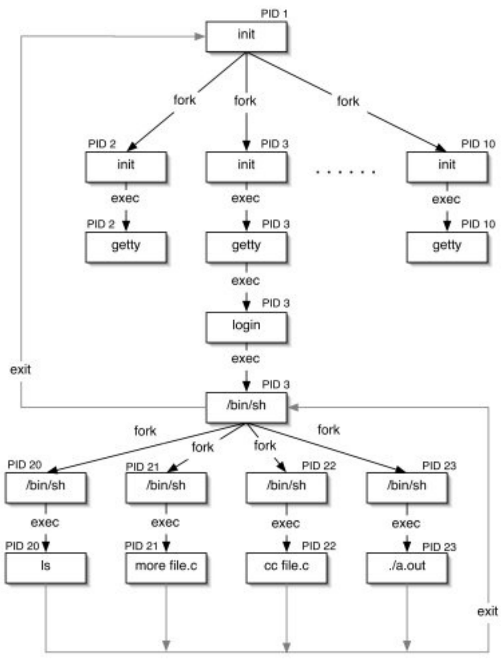
Of note, calling *sleep(1) may* help to separate the outputs, and we *fflush()* in each process to force its output to appear.

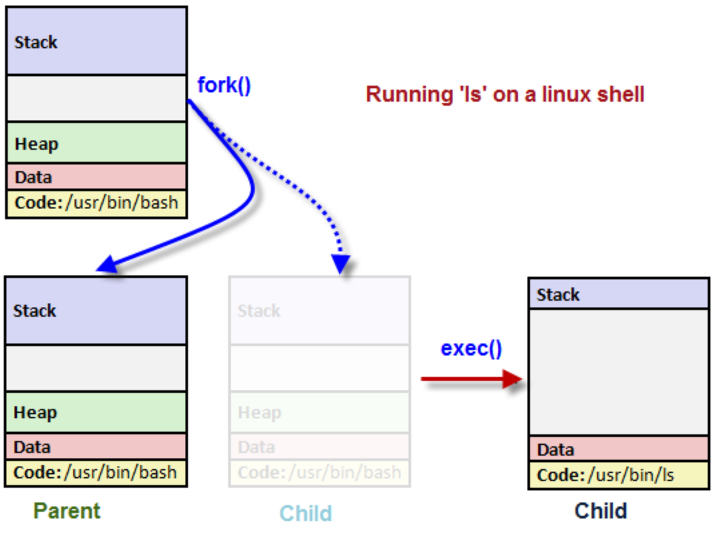
Where does the first process come from?

The *last internal action* of booting a Unix-based operating system results in the first single 'true' process, named *init*.

init has the process-ID of *1*. It is the ancestor process of *all* subsequent processes.

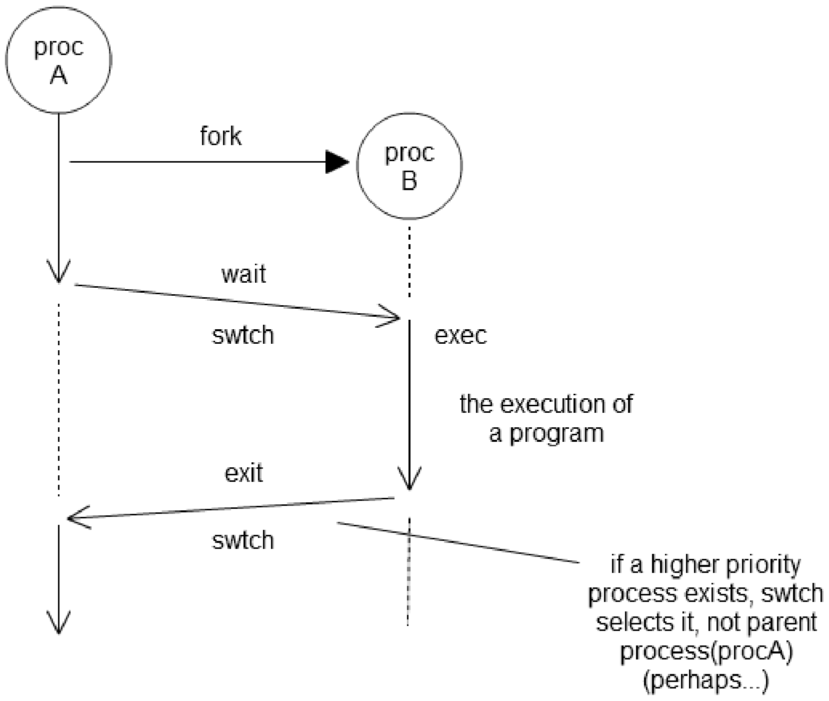
In addition, because the operating system strives to maintain a hierarchical relationship amongst all processes, a process whose parent terminates is 'adopted' by the *init* process.





The general calling sequence of system calls

If a single program has two distinct execution paths/sequences, then the parent and child may run different parts of the same program. Typically the parent will want to know when the child terminates.


proc A
fork
proc B
wait
swtch
exec
the execution of a program
exit
swtch
if a higher priority process exists, swtch selects it, not parent process(procA) (perhaps...)

The typical sequence of events is:

- the parent process **fork()**s a new child process.
- the parent waits for the child's termination, calling the blocking function **wait(&status)**.
- [optionally] the child process replaces details of its program (code) and data (variables) by calling the **execve()** function.
- the child calls **exit(value)**, with an integer value to represent its success or failure. By convention, zero (= EXIT_SUCCESS) indicates successful execution, non-zero otherwise.
- the child's value given to **exit()** is written by the operating system to the parent's **status**.

Waiting for a Process to Terminate

The parent process typically lets the child process execute, but wants to know when the child has terminated, and whether the child terminated successfully or otherwise.

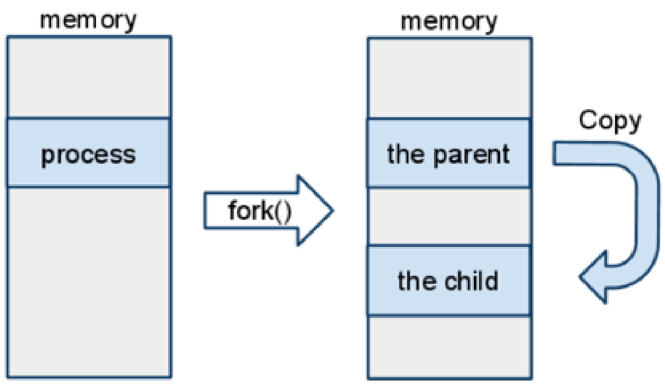
A parent process calls the *wait()* system call to suspend its own execution, and to wait for*any* of its child processes to terminate.

The (new?) syntax **&status** permits the *wait()* system call (in the operating system kernel) to modify the calling function's variable. In this way, the parent process is able to receive information about *how* the child process terminated.

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

void function(void)
{
    switch ( fork() ) {
    case -1 :
        printf("fork() failed\n"); // process creation failed
        exit(EXIT_FAILURE);
        break;

    case 0:                        // new child process
        printf("child is pid=%i\n", getpid() );

        for(int t=0 ; t<3 ; ++t) {
            printf("  tick\n");
            sleep(1);
        }
        exit(EXIT_SUCCESS);
        break;

    default: {                     // original parent process
        int child, status;

        printf("parent waiting\n");
        child = wait( &status );

        printf("process pid=%i terminated with exit status=%i\n",
                child, WEXITSTATUS(status) );
        exit(EXIT_SUCCESS);
        break;
    }

    }
}
```

Memory in Parent and Child Processes

The (existing) parent process and the (new) child process continue their own execution.



Importantly, both the parent and child have *their own copy* of their program's memory (variables, stack, heap).

The parent naturally uses the memory that it had before it called *fork();* the child receives its own copy of the same memory. The copy is made at the time of the *fork()*.

As execution proceeds, each process may update its own memory without affecting the other process.

[OK, I lied - on contemporary operating systems, the child process does *not* receive a full copy of its parent's memory at the time of the *fork()*:

- the child can *share* any read-only memory with its parent, as neither process can modify it.
- the child's memory is only copied from the parent's memory if either the parent modies its (original) copy, or if the child attempts to write to its copy (that it hasn't yet received!)
- this sequence is termed *copy-on-write*.

]

Running a New Program

Of course, we do not expect a single program to meet all our computing requirements, or for both parent and child to conveniently execute different paths through the same code, and so we need the ability to commence the execution of *new programs* after a *fork()*.

Under Unix/Linux, a new program may replace the currently running program. The new program runs as the same process (it has the same *pid*, confusing!), by overwriting the current process's memory (instructions and data) with the instructions and data of the new program.

The single system call **execv()** requests the execution of a new program as the current process:

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

char *program_arguments[] = {
    "ls",
    "-l",
    "-F",
    NULL
};

    ....
    execv( "/bin/ls", program_arguments );
    // A SUCCESSFUL CALL TO exec() DOES NOT RETURN

    exit(EXIT_FAILURE);  // IF WE GET HERE, THEN exec() HAS FAILED
```

On success, **execv()** does not return (to where would it return?)
On error, -1 is returned, and *errno* is set appropriately (EACCES, ENOENT, ENOEXEC, ENOMEM, ...).

The single system call is supported by a number of library functions (see *man execl*) which simplify the calling sequence.

Typically, the call to **execv()** (via one of its library interfaces) will be made in a child process, while the parent process continues its execution, and eventually waits for the child to terminate.

Why the exit status of a program is important

To date, we've always used `exit(EXIT_FAILURE)` when a problem has been detected, or `exit(EXIT_SUCCESS)` when all has gone well. Why?

The operating system is able to use the *exit status* of a program to determine if it was successful.

Consider the following program which exits with the integer status provided as a command-line argument:

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

int main(int argc, char *argv[])
{
    int status = EXIT_SUCCESS;    // DEFAULT STATUS IS SUCCESS (=0)

    if(argc > 1) {
        status = atoi(argv[1]);
    }
    printf("exiting(%i)\n", status);

    exit(status);
}
```

Why the exit status of a program is important, *continued*

Most operating system *shells* are, themselves, programming languages, and they may use a program's*exit status* to direct control-flow within the shells - thus, the programming language that is the shell, is treating your programs as if they are external functions.

Shells are typically programmed using files of commands named *shellscripts* or *command files* and these will often have conditional constructs, such as *if* and *while*, just like C. It's thus important for our programs to work with the shells that invoke them.

We now compile our program, and invoke it with combinations of zero, and non-zero arguments:

```
prompt>  mycc -o status status.c

prompt>  ./status 0 && ./status 1
exiting(0)
exiting(1)

prompt>  ./status 1 && ./status 0
exiting(1)

prompt>  ./status 0 || ./status 1
exiting(0)

prompt>  ./status 1 || ./status 0
exiting(1)
exiting(0)
```

- Example1 - consider the sequence `prompt>` cd mydirectory && rm -f *
- Example2 - consider the *actions* in a *Makefile* (discussed in a later lecture).
  If a *target* has more than one action, then the*make* program executes each until one of them fails (or until all succeed).