Lesson 2

fork: create a new process. The new process (child process) is almost an exact copy of the calling process (parent process). In this method we create an hierarchy structure for the processes, which is similar to the files structure in Unix. The root node of this tree is the process, which is the execution of the *init* program (pid = 1), is the ancestor of all the system and user processes.

SYNOPSIS
#include <sys/types.h>
#include <unistd.h>

pid_t fork(*void*);

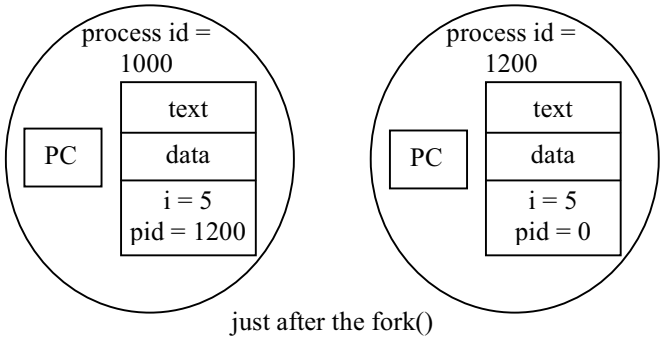
The *fork* command copies all of the process (the code = the program, the data = global variables, the stack = automatic variable [local variables in the function] and the program counter).
The new process has a different pid and its ppid is the same as the pid of the calling process.

RETURN VALUES
Upon successful completion, **fork**() returns a value of 0 to the child process and returns the process ID of the child process to the parent process.
Otherwise, a value of -1 is returned to the parent process, no child process is created, and the global variable *errno* is set to indicate the error.

Note: If the process has open files and performs fork(), then any change in the read/write head in any process will entail the same change in the other processes, since the read/write head is shared for all processes because it is managed by the system.

Example: ***(2_1.c)***

```
#include <sys/types.h>
int main()
{
        pid_t pid;
        int i;
        i = 5;
        pid = fork();
        i++;
        printf("%d", i);
}
```



Output: 66, but it is unknown which process printed the first 6, since it depends on the CPU the system gave to each process (***race condition***).

Note that in the child process there are some lines that are never being executed (all the lines before the fork). In the above example 3 lines are not executed in the child process.

Examples:
(2_2.c)

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

main()
{
        pid_t pid;
        pid = fork();
        if (pid == 0)
                printf("\nI'm the child process");
        else  if (pid > 0)
                printf("\nI'm the parent process. My child pid is %d", pid);
        else
                perror("error in fork");
}
```

(2_3.c)

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

main()
{
        pid_t pid;
        int i = 5;
        pid = fork();
        if (pid == 0) // child process
                i++;
        printf("%d", i);
}
```

Output: 56 or 65

(2_4.c)

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

main()
{
        pid_t pid;
        if ( (pid = fork()) == 0)
                printf("1");
        else
                printf("2");
        printf("3");
}
```

Output: 2133 or 1233 or 2313 or 1323 (3312 is not possible)

(2_5.c)

```
main()
{
        if (fork() == 0)
                while(1);
        else
                while(1);
}
```

State of a process:
O - Running.
S - Sleeping (process is waiting for an event to complete).
R - Runnable (process is on run queue).
Z - Zombie (process terminated and parent not waiting)
T - Stopped.

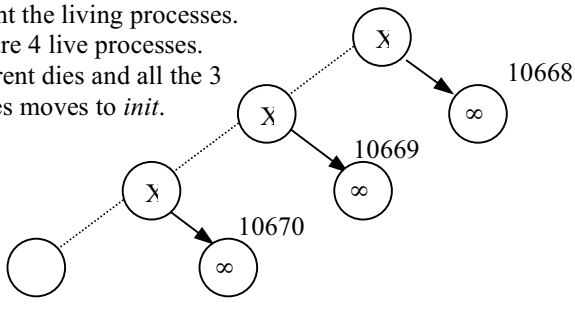
```
1 > gcc -o example1 2_5.c
2 > example1 &
[1] 7580
3 > ps -l
UID   PID   PPID NI STAT TT    TIME COMMAND
8385  4709  4705 20 S    pts/0 0:01 -tcsh
8385  7580  4709 20 R    pts/0 0:21 example1
8385  7581  7580 20 R    pts/0 0:22 example1
4 > kill -KILL %1
[1]   Killed      example1
```

(2_6.c)

```
main()
{
        int i;
        for (i = 0; i < 3; i++)
                if (fork() == 0)
                        while(1);
}
```

```
1 > gcc -o example2 2_6.c
2 > example2
3 > ps -l
```

The leaves represent the living processes.
When i = 2, there are 4 live processes.
Afterwards, the parent dies and all the 3 remaining processes moves to *init*.



```
UID   PID  PPID NI STAT TT    TIME COMMAND
8385  4709 4705 20 S    pts/0 0:01 -tcsh
4 > ps -al
UID   PID  PPID NI STAT TT    TIME COMMAND
8385  4709 4705 20 S    pts/0 0:01 -tcsh
8385 10668 1    20 RO   pts/0 0:12 example2
8385 10669 1    20 RO   pts/0 0:14 example2
8385 10670 1    20 RO   pts/0 0:13 example2
5 > kill -KILL 10668 10669 10670
```

(2_7.c)

```
main()
{
        if (fork() != 0)
          while(1);
}
2 > gcc -o example3 2_7.c
3 > example3 &
[1] 11499
4 > ps -al
UID   PID   PPID  NI STAT TT    TIME COMMAND
8385  11500 11499 0  Z          0:00 <defunct>
8385  4709  4705  20 S    pts/0 0:01 -tcsh
8385  11499 4709  20 R    pts/0 0:44 example3
4 > kill -KILL %1
[1]   Killed      example3
```

wait, waitpid, wait4, wait3: wait for process termination.

SYNOPSIS

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

pid_t wait(int *status);
pid_t waitpid(pid_t wpid, int *status, int options); // wait for a specific pid

#include <sys/types.h>
#include <time.h>
#include <sys/resource.h>
#include <sys/wait.h>

pid_t wait3(int *status, int options, struct rusage *rusage);
pid_t wait4(pid_t wpid, int *status, int options, struct rusage *rusage);
```

DESCRIPTION

The **wait**() function suspends execution of its calling process until *status* information is available for a terminated child process, or a signal is received. If there are more than one child process then wait returns when *any* of the child process finishes.

On return from a successful **wait**() call, the *status* area contains termination information about the process that exited (if the value of *status* is 0 → program terminated with no errors).

The **wait4**() call provides a more general interface for programs that need to wait for certain child processes, that need resource utilization statistics accumulated by child processes, or that require options. The other wait functions are implemented using **wait4**().

The *wpid* parameter specifies the set of child processes for which to wait:

If *wpid* is -1, the call waits for any child process.

If *wpid* is 0, the call waits for any child process in the process group of the caller.

If *wpid* is greater than zero, the call waits for the process with process id *wpid*.

If *wpid* is less than -1, the call waits for any process whose process group id equals the absolute value of *wpid*.

The *options* parameter contains the bitwise OR of any of the following options:

- The WNOHANG option is used to indicate that the call should not block if there are no processes that wish to report status.
  When the WNOHANG option is specified and no processes wish to report status, **wait4**() returns a process id of 0.

- The WUNTRACED is used in order to wait also for children which are *stopped*, and whose status has not been reported.

The **waitpid**() call is identical to **wait4**() with an *rusage* value of zero. The **wait3**() call is the same as **wait4**() with a *wpid* value of -1.

The following macros may be used to test the manner of exit of the process.

WIFEXITED(*status*) True if the child exited normally.
WIFSIGNALED(*status*) True if the process terminated due to receipt of a signal.
WEXITSTATUS(*status*) Returns the return code of the process (the least significant eight bits of the return code).
WIFSTOPPED(status) returns true if the child process which caused the return is currently stopped; this is only possible if the call was done using WUNTRACED.
WSTOPSIG(status) returns the number of the signal which caused the child to stop. This macro can only be evaluated if WIFSTOPPED returned non-zero.

If *rusage* is not NULL, the struct rusage as defined in <sys/resource.h> it points to will be filled with accounting information:

```
struct rusage {
        struct timeval ru_utime; /* user time used */
        struct timeval ru_stime; /* system time used */
        …
        long ru_majflt; /* page faults */
        long ru_nswap; /* swaps */
        …
        long ru_nsignals; /* signals received */
        …
};
```

(For more information use: man getrusage)

RETURN VALUES

If **wait**() returns due to a stopped or terminated child process, the process ID of the child is returned to the calling process. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.
If **wait4**(), **wait3**() or **waitpid**() returns due to a stopped or terminated child process, the process ID of the child is returned to the calling process. If there are no children not previously awaited, -1 is returned with *errno* set to [ECHILD]. Otherwise, if WNOHANG is specified and there are no stopped or exited children, 0 is returned. If an error is detected or a caught signal aborts the call, a value of -1 is returned and *errno* is set to indicate the error.

wait waits only for one child. In order to wait for all children processes:

```
while (wait(&status) != -1)
        ;
```

Examples:

(2_8.c)

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

main()
{
        pid_t pid;
        int stat;
        if ( (pid = fork() ) == 0)
                printf("1"); // child process
```

```
        else // parent process
        {
                wait(&stat);
                printf("2");
        }
}
```

Output: 12 (21 is ***not possible***).

(2_9.c)

```
main()
{
        int i, s;
        for (i = 1; i <= 3; i++)
                if (fork() == 0)
                        while(1);
                else
                        if (i == 3)
                                wait(&s);
}
```

```
1 > gcc -o example4 2_9.c
2 > example4 &
[1] 13422
4 > ps -l
UID   PID   PPID  NI STAT TT    TIME COMMAND
8385  4709  4705  20 R    pts/0 0:02 -tcsh
8385 13422  4709  20 S    pts/0 0:00 example4
8385 13423 13422  20 R    pts/0 0:05 example4
8385 13425 13422  20 R    pts/0 0:05 example4
8385 13426 13422  20 R    pts/0 0:04 example4
5 > kill -KILL %1
[1] Killed  example4
```

(2_17.c)

```
main()
{
        int status;
        pid_t pid, pid1, pid2;

        if ((pid1 = fork()) == 0)
                printf("in child 1\n");
        else
                if ((pid2 = fork()) == 0)
                        printf("in child 2\n");
                else
                {
                        pid = wait(&status);
                        if (pid == pid1)
                                printf("child 1 finished\n");
                        if (pid == pid2)
                                printf("child 2 finished\n");
                }
}
```

(2_18.c)

```
main()
{
        int status;
        pid_t pid, pid1, pid2;
```

```
        if ((pid1 = fork()) == 0)
                printf("in child 1\n");
        else
                if ((pid2 = fork()) == 0)
                        printf("in child 2\n");
                else
                {
                        pid = wait(&status);
                        pid = wait(&status);
                        if (pid == pid1)
                                printf("child 1 finished\n");
                        if (pid == pid2)
                                printf("child 2 finished\n");
                }
}
```

(2_19.c)

```
main()
{
        int status;
        pid_t pid_a, pid_b, pid1, pid2;

        if ((pid1 = fork()) == 0)
                printf("in child 1\n");
        else
                if ((pid2 = fork()) == 0)
                        printf("in child 2\n");
                else
                {
                        pid_a = wait(&status);
                        if (pid_a == pid1)
                                printf("child 1 finished\n");
                        if (pid_a == pid2)
                                printf("child 2 finished\n");

                        pid_b = wait(&status);
                        if (pid_b == pid1)
                                printf("child 1 finished\n");
                        if (pid_b == pid2)
                                printf("child 2 finished\n");
                }
}
```

What happens if the child terminated before the parent process did wait()?

When a process terminates the system releases all of its resources (variables and PC), except for the termination information of the process, i.e., the process is empty of content but there is still a pid and termination information → the process is a ***zombie***. The process will be freed only when the parent process does wait().

If the parent process terminates before the child process then the child process becomes an ***orphan process*** (child of the init process). The init process executes all the time a wait loop. This will eventually free the orphan process without collecting the termination information.

execl, execlp, execle, exect, execv, execvp: execute a file. These functions replace the content of the program with a new program that will run from its beginning with its variables. The function doesn't create a new process, it just replaces the current executing program.

SYNOPSIS

```
#include <unistd.h>

extern char **environ;
```

Receives the arguments as a NULL terminated list:
int execl(*const char **path*, *const char *arg0*, const char *arg1, …, const char *argn, NULL);
int execlp(*const char **file*, *const char *arg0*, const char *arg1, …, const char *argn, NULL);

Receives the arguments as an arguments' array:
int execv(*const char **path*, *char *const argv[]*);
int execvp(*const char **file*, *char *const argv[]*);

Specifies environment:
int execle(*const char *path*, *const char *arg*, *...*, *char *const envp[]*);

The path can be absolute or relative. The program needs to be executed must have execute permission.

DESCRIPTION
The **exec** family of functions replaces the current process image with a new process image.
The initial argument for these functions is the pathname of a file which is to be executed.
The first argument of the argv list/array, by convention, should point to the file name associated with the file being executed. The list of arguments *must* be terminated by a NULL pointer.
The **execle**()function also specify the environment of the executed process by following the NULL pointer that terminates the list of arguments in the parameter list or the pointer to the argv array with an additional parameter. This additional parameter is an array of pointers to null-terminated strings and *must* be terminated by a NULL pointer. For example:

```
char* env_list[] = { "SOURCE=MYDATA", "TARGET=OUTPUT", "lines=65",
                     NULL};
execle( "myprog", "myprog", "ARG1", "ARG2", NULL, env_list );
```

In this example, `myprog` will be found if it exists in the current working directory. The environment for the invoked program consists of the three environment variables **SOURCE**, **TARGET** and **lines**.

The other functions take the environment for the new process image from the external variable *environ* in the current process.
The functions **execlp**() and **execvp**() will duplicate the actions of the shell in searching for an executable file if the specified file name does not contain a slash "/" character. The search path is the path specified in the environment by "PATH" variable. If this variable isn't specified, the default path "/bin:/usr/bin:" is used.

RETURN VALUES
If any of the exec functions returns, an error will have occurred. The return value is -1, and the global variable *errno* will be set to indicate the error.

Examples:
(2_10.c) - compiled to "prog1"

```
#include <sys/types.h>
int main()
{
        printf("1");
        execl("prog2", "prog2", NULL);
        printf("11");
}
```

(2_11.c) - compiled to "prog2"

```
int main()
{
        printf("2");
}
```

```
% prog1
```

Output:
12

The command *ps* will show that the executing program at first is ***prog1 (pid = 1000)*** and afterwards ***prog2 (pid = 1000)***.
When prog2 terminates the process terminates (there is no return back to the program prog1).
Only if the *execl* fails will the output be "111".

(2_12.c)

```
#include <sys/types.h>
main()
{
        execl("/bin/cal", "cal", "5", "1999", NULL); // executing the command % cal 5 1999
        /* equivalents to:
                char *const ar[] = {"cal", "5", "1999", NULL};
                execv("/bin/cal", ar);
        */
        perror("execl failed");
}
```

(2_13.c)

```
#include <sys/types.h>
main() /* assuming program "fork4" doesn't exist */
{
        int stat;

        if (fork() == 0) {
                execl("/u/usr/fork4", "fork4", NULL); // fails
                printf("1");
        }
        else {
                wait(&stat); // waiting for child's termination
                printf("2");
        }
}
```

Output: 12

(2_14.c)

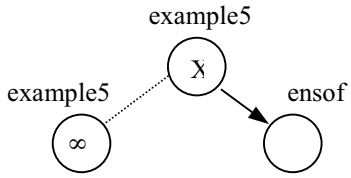
```
#include <sys/types.h>
main() /* assuming program "fork5" doesn't exist */
{
        int stat;

        if (fork() == 0) {
                execl("/u/usr/fork5", "fork5", NULL);
                printf("1");
                exit(1);
        }
        else {
                wait(&stat);
                printf("2");
                if (stat != 0) //
                        printf("3");
        }
}
```

Output: 123

(2_15.c)

```
#include <sys/types.h>
main()
{
        if (fork() == 0) {
                execl("/u/usr/ensof", "ensof", NULL); // the program "ensof": while(1);
                exit(1);
        }
        else
                while(1);
}
```

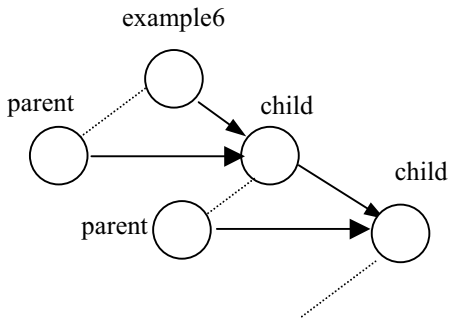
```
2 > gcc -o example5 2_15.c
3 > example5 &
[1] 15202
4 > ps
PID   TT    STAT  TIME  COMMAND
4709  pts/0 S     0:02  -tcsh
15202 pts/0 R     0:01  example5
15203 pts/0 R     0:01  ensof
5 > kill -KILL %1
[1] Killed  example5
```



(2_16.c)

```
#include <sys/types.h>
main()
{
        int s;
        if (fork() == 0)
                execl("/u/usr/example6", "example6", NULL);
        else
                wait(&s);
}
```

```
2 > gcc -o example6 2_16.c
3 > example6
```



Numerous processes will be created (how many - depends on how many resources the system gives us).
Eventually *fork()* will fail and return -1. We then go to the else block where we do *wait*. Since this process has no child (fork failed) *wait* fails and the process terminates.
Then, in a chain reaction, all other processes will do wait, and finally we will remain with 0 processes.

getpid(): get process id
getppid(): get parent process id

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

pid_t getpid(void);
pid_t getppid(void);
```