System calls on Process management

Process

- A process is a program at the time of execution.
- The *fork()* system call creates a new process.
- The process that involves *fork()* is called the *parent process* and the newly created process is called the *child process*.

```
pid =fork();
```

- **`pid` is the process id of child.**

Process

The kernel does the following operations for *fork*:

- It makes an entry in the process table for the new process.
- It assign a unique ID to the child process.
- It makes a logical copy of the context of the parent process.
- It returns the ID of child process to the parent process. A 0 value to the child process on success or -1 to the parent process on failure.

Process termination

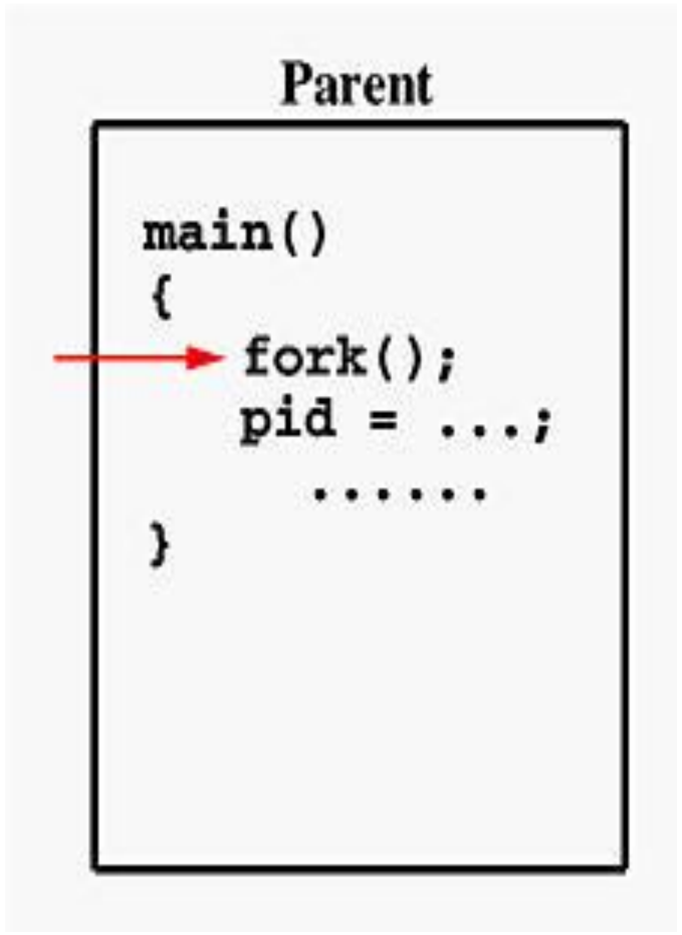
Generally the process terminates when execution finished. Some other reasons are:

- Time slot expire.
- Memory violation.
- I/O failure.
- Parent termination.
- Invalid instruction.

fork()

- If **fork()** returns a negative value, the creation of a child process was unsuccessful.
- **fork()** returns a zero to the newly created child process.
- **fork()** returns a positive value, the ***process ID*** of the child process, to the parent.
- The returned process ID is of type **pid_t** defined in **sys/types.h**.
- A process can use function **getpid()** to retrieve the process ID assigned to this process.

fork()

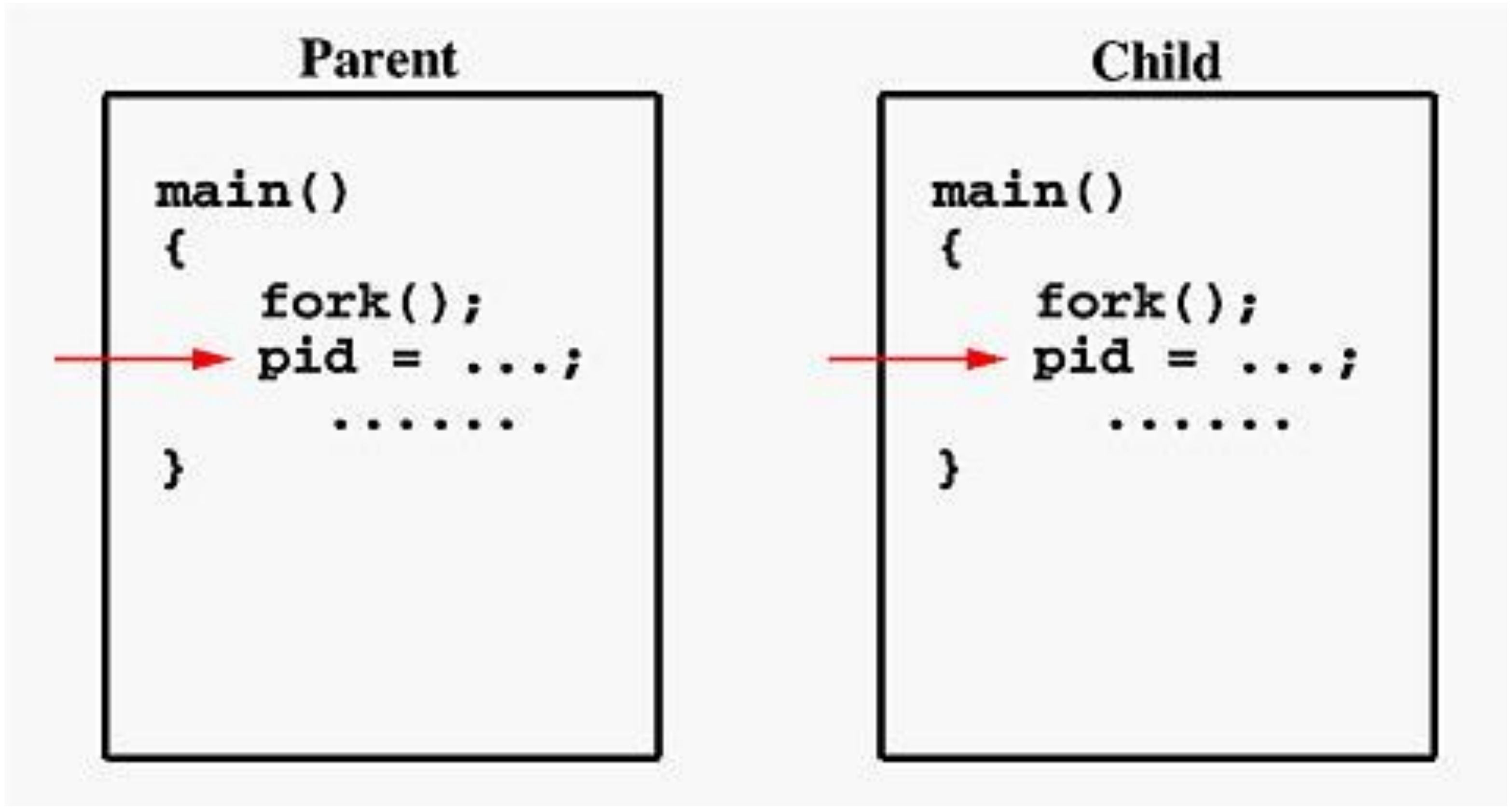


```
main()
{
    fork();
    pid = ...;
       ......
}
```

If the call to **fork()** is executed successfully, Unix will

- make two identical copies of address spaces, one for the parent and the other for the child.
- Both processes will start their execution at the next statement following the **fork()** call.

fork()



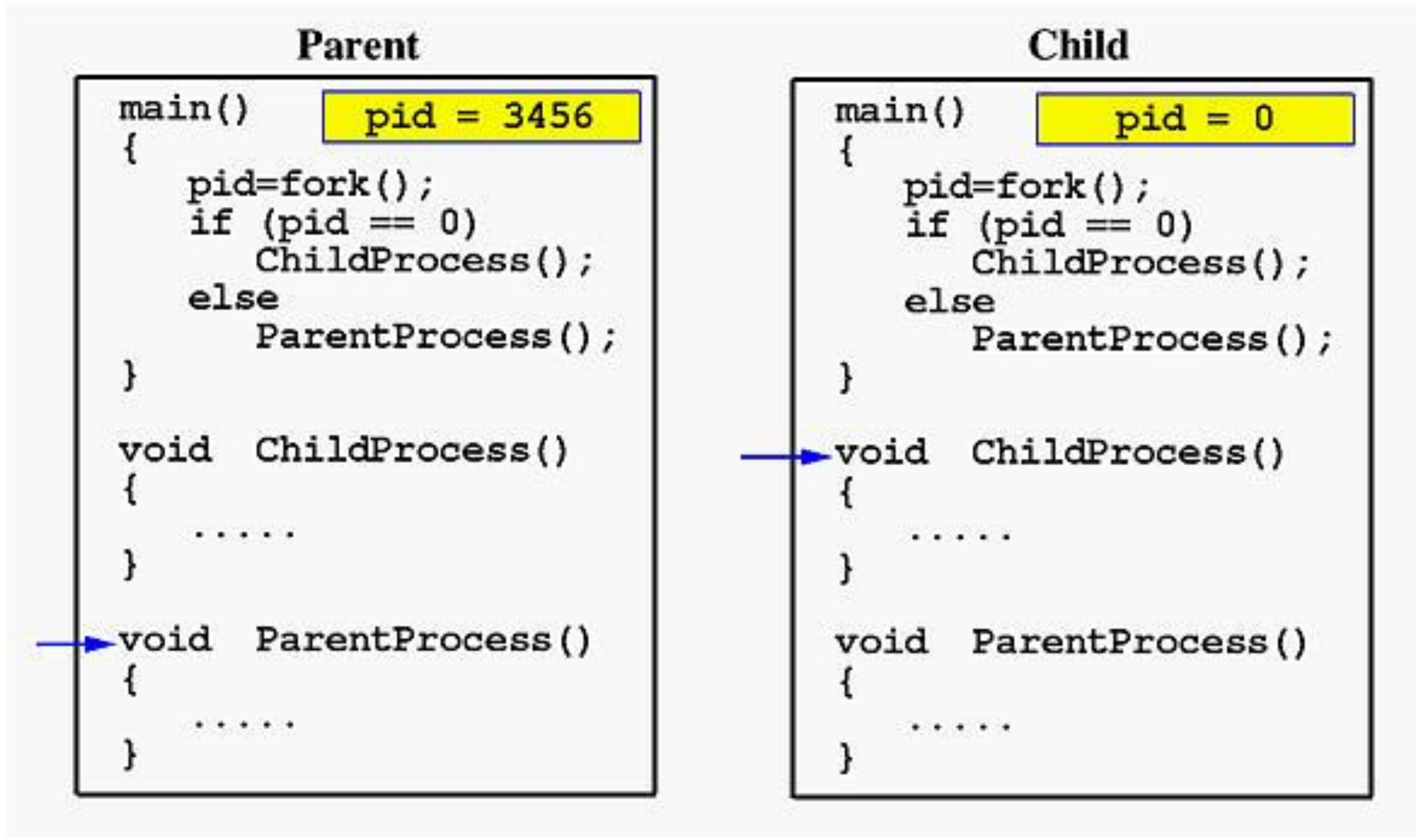
```
main()
{
    fork();
    pid = ...;
      ......
}
```

```
main()
{
    fork();
    pid = ...;
      ......
}
```

fork()

```
#define   MAX_COUNT  20
void  ChildProcess(void);              /* child process prototype  */
void  ParentProcess(void);              /* parent process prototype */
void  main(void)  {
    pid_t  pid;     pid = fork();
    if (pid == 0)
        ChildProcess();
    else
        ParentProcess();      }
void  ChildProcess(void)
{
    int   i;
    for (i = 1; i <= MAX_COUNT; i++)
        printf("   This line is from child, value = %d\n", i);     printf("   *** Child process is done ***\n");
}
void  ParentProcess(void)
{    int   i;
    for (i = 1; i <= MAX_COUNT; i++)
        printf("This line is from parent, value = %d\n", i);     printf("*** Parent is done ***\n");
}
```

fork()



```
main()
{
   pid=fork();
   if (pid == 0)
      ChildProcess();
   else
      ParentProcess();
}

void  ChildProcess()
{
   .....
}

void  ParentProcess()
{
   .....
}
```

```
main()
{
int dd;
printf("%d: I am the parent. Remember my number!\n", getpid());
printf("%d: I am now going to fork ... \n", getpid());
dd = fork();
if (dd != 0)
{ /* the parent will execute this code */
printf("%d: My child's pid is %d\n", getpid(), dd);
}
else /* dd== 0 */
{ /* the child will execute this code */
printf("%d: Hi! I am the child.\n", getpid());
}
printf("%d: like father like son. \n", getpid());
}
```

```
3088: I am the parent. Remember my number!
3088: I am now going to fork ...
3088: My child's pid is 3089
3088: like father like son.
3089: Hi! I am the child.
3089: like father like son.
```

System calls: Process

- fork()
- wait()
- exit()
- sleep()
- exec family [execl, execv, execle, execve, execlp, execvp]

wait()

- **pid_t   wait(int *status)**
- It will force a parent process to wait for a child process to stop or terminate.
- A call to this function causes the parent process to wait until one of its child processes exits.
- The wait call returns the process id of the child process, which gives the parent the ability to wait for a particular child process to finish.
- It return -1 for an error.

```
#include <stdio.h> #include <sys/wait.h> /* contains prototype for wait */
int main(void)
{
int pid;
int status;
printf("Hello World!\n");
pid = fork( );
if (pid == -1) /* check for error in fork */
{
printf(“error");
exit(1);
}
if (pid == 0)
printf("I am the child process.\n");
else
{
wait(&status); /* parent waits for child to finish */
printf("I am the parent process.\n");
}
}
```

```
Hello World!
I'm the child process with PID 3497 and PPID 3438.
I'm the child process with PID 3498 and PPID 3497.

 I am the child process.


 I am the parent process.
```

```
#define  MAX _COUNT  20
void  ChildProcess(void);
void  ParentProcess(void);
void  main(void)
{
pid_t  pid;
pid = fork();
if (pid == 0)
ChildProcess();
else
ParentProcess();
}
```

```
void  ChildProcess(void)
{
int   i;
for (i = 1; i <= MAX_COUNT; i++)
printf("This line is from child, value =
%d\n", i);
printf("*** Child process is done
***\n");
}
void  ParentProcess(void)
{
int   i;  /* int status;  wait(&status);  */
for (i = 1; i <= MAX_COUNT; i++)
printf("This line is from parent, value =
%d\n", i);
printf("*** Parent is done ***\n");
}
```

```
This line is from child, value = 1
This line is from parent, value = 1
This line is from parent, value = 2
This line is from parent, value = 3
This line is from parent, value = 4
This line is from parent, value = 5
This line is from parent, value = 6
This line is from parent, value = 7
This line is from parent, value = 8
This line is from parent, value = 9
This line is from parent, value = 10
This line is from parent, value = 11
This line is from parent, value = 12
This line is from child, value = 2
This line is from parent, value = 13
This line is from parent, value =14
This line is from parent, value = 15
This line is from parent, value = 16
This line is from parent, value = 17
This line is from child, value = 3
This line is from parent, value = 18
This line is from parent, value = 19
This line is from child, value = 4
This line is from child, value = 5
This line is from child, value = 6
This line is from parent, value = 20
*** Parent is done ***
This line is from child, value = 7
This line is from child, value = 8
This line is from child, value = 9
This line is from child, value = 10
This line is from child, value = 11
This line is from child, value = 12
This line is from child, value = 13
This line is from child, value = 14
This line is from child, value = 15
This line is from child, value = 16
This line is from child, value = 17
This line is from child, value = 18
This line is from child, value = 19
This line is from child, value = 20
*** Child process is done ***
```

```
This line is from child, value = 1
This line is from child, value = 2
This line is from child, value = 3
This line is from child, value = 4
This line is from child, value = 5
This line is from child, value = 6
This line is from child, value = 7
This line is from child, value = 8
This line is from child, value = 9
This line is from child, value = 10
This line is from child, value = 11
This line is from child, value = 12
This line is from child, value = 13
This line is from child, value = 14
This line is from child, value = 15
This line is from child, value = 16
This line is from child, value = 17
This line is from child, value = 18
This line is from child, value = 19
This line is from child, value = 20
*** Child process is done ***
```

```
This line is from parent, value = 1
This line is from parent, value = 2
This line is from parent, value = 3
This line is from parent, value = 4
This line is from parent, value = 5
This line is from parent, value = 6
This line is from parent, value = 7
This line is from parent, value = 8
This line is from parent, value = 9
This line is from parent, value = 10
This line is from parent, value = 11
This line is from parent, value = 12
This line is from parent, value = 13
This line is from parent, value = 14
This line is from parent, value = 15
This line is from parent, value = 16
This line is from parent, value = 17
This line is from parent, value = 18
This line is from parent, value = 19
This line is from parent, value = 20
*** Parent is done ***
```

exit()

- It terminates the process which calls this function and returns the exit status value.
- By convention, a status of 0 means normal termination. Any other value indicates an error.
- Many standard library calls have errors defined in the sys/stat.h header file.
- The *exit status/return code* of a process is a small number passed from a child process to a parent process when it has finished executing a specific procedure or delegated task.

exit()

- On OS, a process terminates its execution by making an exit() system call.
- The operating system reclaims resources (memory, files, etc.) that were used by the process.
- *The process is said to be a dead process after it terminates.*

sleep()

- A process may sleep, which places it into an *inactive state* for a period of time.
- Eventually the expiration of an interval timer, or the receipt of a signal or interrupt causes the program to resume execution.
- A process may suspend for a period of time using the sleep command .

sleep()

- unsigned int sleep (time)
- A typical sleep system call takes a time value as a parameter, specifying the minimum amount of time that the process is to sleep before resuming execution.
- The parameter typically specifies seconds, although some OSs provide finer resolution, such as milliseconds or microseconds.

Orphan process

- When a parent dies before its child, the child is automatically adopted by the original “init” process whose PID is 1.
- An orphan process is a computer process whose parent process has finished, though it remains running itself.

```
void main()
{
int pid ;
printf("I am original process with PID %d & PPID %d.\n",  getpid(), getppid());
pid = fork ( ) ;
if ( pid != 0 )
{
printf("I'am the parent with PID %d and PPID %d.\n", getpid(), getppid()) ;
printf("My child's PID is %d\n", pid ) ;
}
else /* pid is zero, so I must be the child */
{
sleep(40);                    /* make sure that the parent terminates first */
printf("I'm the child with PID %d and PPID %d.\n",  getpid(), getppid()) ;
}
printf ("PID %d terminates.\n",  getpid()) ;
}
```

```
I am original process with PID 2845 & PPID 2662.
I'am the parent with PID 2845 and PPID 2662.
My child's PID is 2846
PID 2845 terminates.
I'm the child with PID 2846 and PPID 1.
PID 2846 terminates.
```

Zombie process

- A process that terminates cannot leave the system until its parent accepts its return code.
- If its parent process is already dead, it’ll already have been adopted by the “init” process, which always accepts its children’s return codes.
- However, if a process’s parent is alive but never executes a wait(), the process’s return code will never be accepted and the process will remain a *zombie.*

```
#include <stdio.h>
main ( )
{
int pid ;
pid = fork();               /* Child and parent continue from here */
if ( pid != 0 )          /* pid is non-zero, so I must be the parent */
{
while (1)             /* Never terminate and never execute a wait () */
sleep (100) ;                     /* stop executing for 100 seconds */
}
else               /* pid is zero, so I must be the child */
{
exit (1) ;
}
}
```

exec family

- In computing, exec(execute) is a functionality of an OS that runs an executable file in the context of an already existing process, replacing the previous executable.
- This act is also referred to as an overlay.
- As a new process is not created, the process identifier (PID) does not change, but the machine code, data, heap, and stack of the process are replaced by those of the new program.
- The exec functions are defined in *unistd.h header file*

exec family

- The base of each is exec, followed by one or more letters:
- e – An array of pointers to environment variables is explicitly passed to the new process image.
- l – Command-line arguments are passed individually to the function.
- p – Uses the PATH environment variable to find the file named in the path argument to be executed.
- v – Command-line arguments are passed to the function as an array of pointers.

execv()

- **int execv (const char *filename, char *const argv[ ])**
- The execv() function executes the file named by filename as a new process image.
- The argv argument is an array of null-terminated strings that is used to provide a value for the argv argument to the main function of the program to be executed.
- By convention, the first element of this array is the file name of the program.

execv()

```
#include <stdio.h>
#include <unistd.h>
void main (argc, argv)
int argc ;
char *argv[] ;
{
execv ("/bin/ls",    /* program to load - full path only */
&argv[0] ) ;
printf ("EXEC Failed\n") ;
/* This above line will be printed only on error and not otherwise */
}
```

It will return all the files as done by $ ls

execl()

- The environment for the new process image is taken from the environ variable of the current process image.
- **int execl (const char *filename, const char *arg0, ...)**
- This is similar to execv,() but the argv strings are specified individually instead of as an array.
- A null pointer must be passed as the last argument.

execl()

```
#include <stdio.h>
#include <unistd.h>
void main ( )
{
execl ("/bin/ls",          /* program to run - give full path */
 "ls",                     /* name of program sent to argv[0] */
"-l",                      /* first parameter (argv[1])*/
"-a",                      /* second parameter (argv[2]) */
NULL) ;                    /* terminate arg list */
printf ("EXEC Failed\n") ;
/* This above line will be printed only on error and not otherwise */
}
```

$ ls –l –a

$ ls –a Displays all files.

$ls –l Displays the long format listing.

execvp()

- **int execvp (const char *filename, char *const argv[ ])**
- The execvp() function is similar to execv(), except that it searches the directories listed in the PATH environment variable to find the full file name of a file from filename if filename does not contain a slash.
- This function is useful for executing system utility programs, because it looks for them in the places that the user has chosen.

execvp()

```
#include <stdio.h>
#include <unistd.h>
main (int argc, char *argv[] )
{
execvp (“ls",       /* program to load - PATH searched */
&argv[0] ) ;
printf ("EXEC Failed\n") ;
/* This above line will be printed only on error and not otherwise */
}
```

execlp()

- **int execlp (const char *filename, const char *arg0, ...)**
- This function is like execl(), except that it performs the same file name searching as the execvp() function.

execlp()

```
#include <stdio.h>
#include <unistd.h>
void main ( )
{
execlp ("ls",      /* program to run - PATH Searched */
"ls",              /* name of program sent to argv[0] */
"-l",              /* first parameter (argv[1])*/
"-a",              /* second parameter (argv[2]) */
NULL) ;            /* terminate arg list */
printf ("EXEC Failed\n") ;
/* This above line will be printed only on error and not otherwise */
}
```

$ ls –l –a

$ ls –a Displays all files.

$ls –l Displays the long format listing.