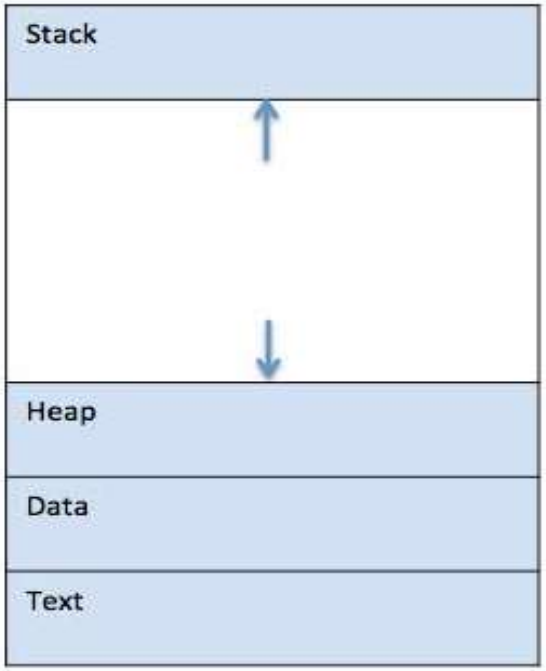
Process Creation
Part 1

Process

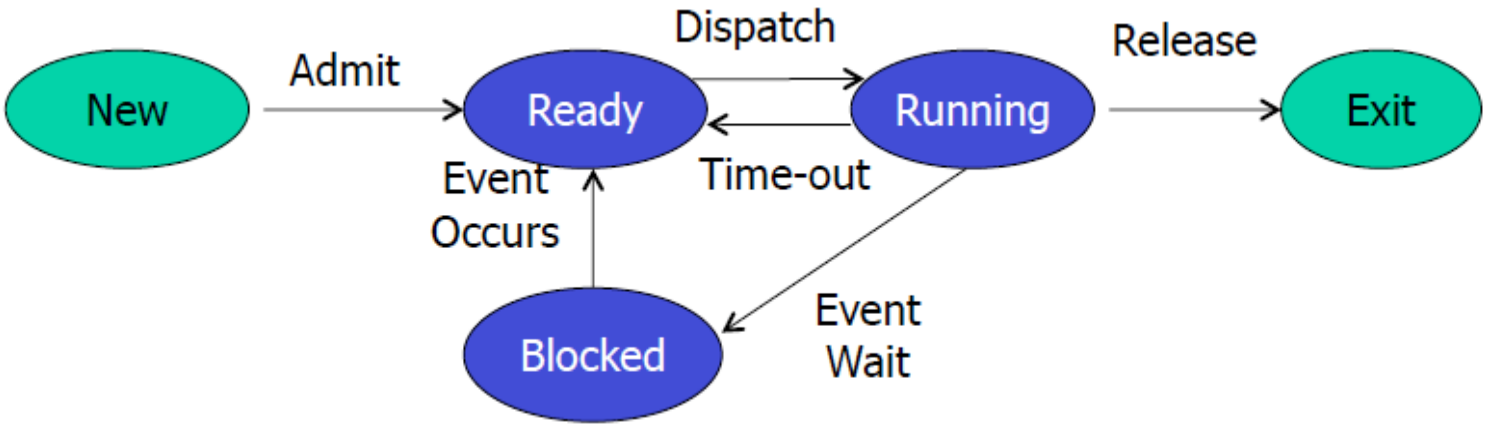
- A process is basically a program in execution.
- To put it in simple terms, we write our computer programs in a text file and when we execute this program, it becomes a process which performs all the tasks mentioned in the program.

When a program is loaded into the memory and it becomes a process, it can be divided into four sections — stack, heap, text and data. The following image shows a simplified layout of a process inside main memory −



Process Life Cycle

- When a process executes, it passes through different states. These stages may differ in different operating systems, and the names of these states are also not standardized.



Process Control Block (PCB)

- A Process Control Block is a data structure maintained by the Operating System for every process.
- The PCB is identified by an integer process ID (PID). A PCB keeps all the information needed to keep track of a process as listed below in the table

S.N.	Information & Description
1	**Process State** The current state of the process i.e., whether it is ready, running, waiting, or whatever.
2	**Process privileges** This is required to allow/disallow access to system resources.
3	**Process ID** Unique identification for each of the process in the operating system.
4	**Pointer** A pointer to parent process.
5	**Program Counter** Program Counter is a pointer to the address of the next instruction to be executed for this process.

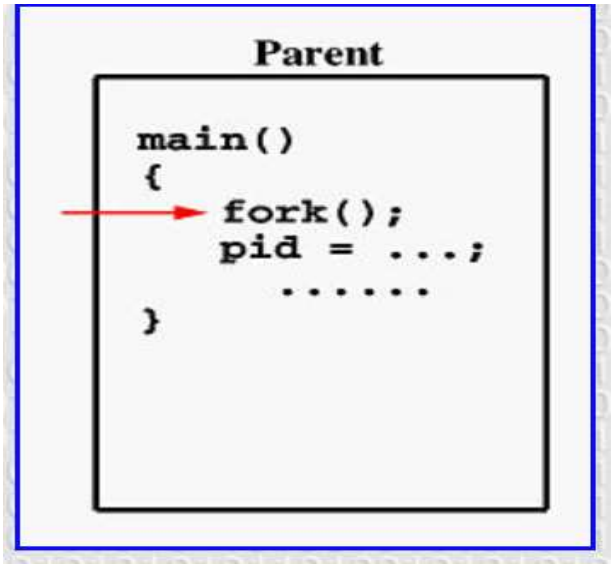
6	**CPU registers** Various CPU registers where process need to be stored for execution for running state.
7	**CPU Scheduling Information** Process priority and other scheduling information which is required to schedule the process.
8	**Memory management information** This includes the information of page table, memory limits, Segment table depending on memory used by the operating system.
9	**Accounting information** This includes the amount of CPU used for process execution, time limits, execution ID etc.
10	**IO status information** This includes a list of I/O devices allocated to the process.

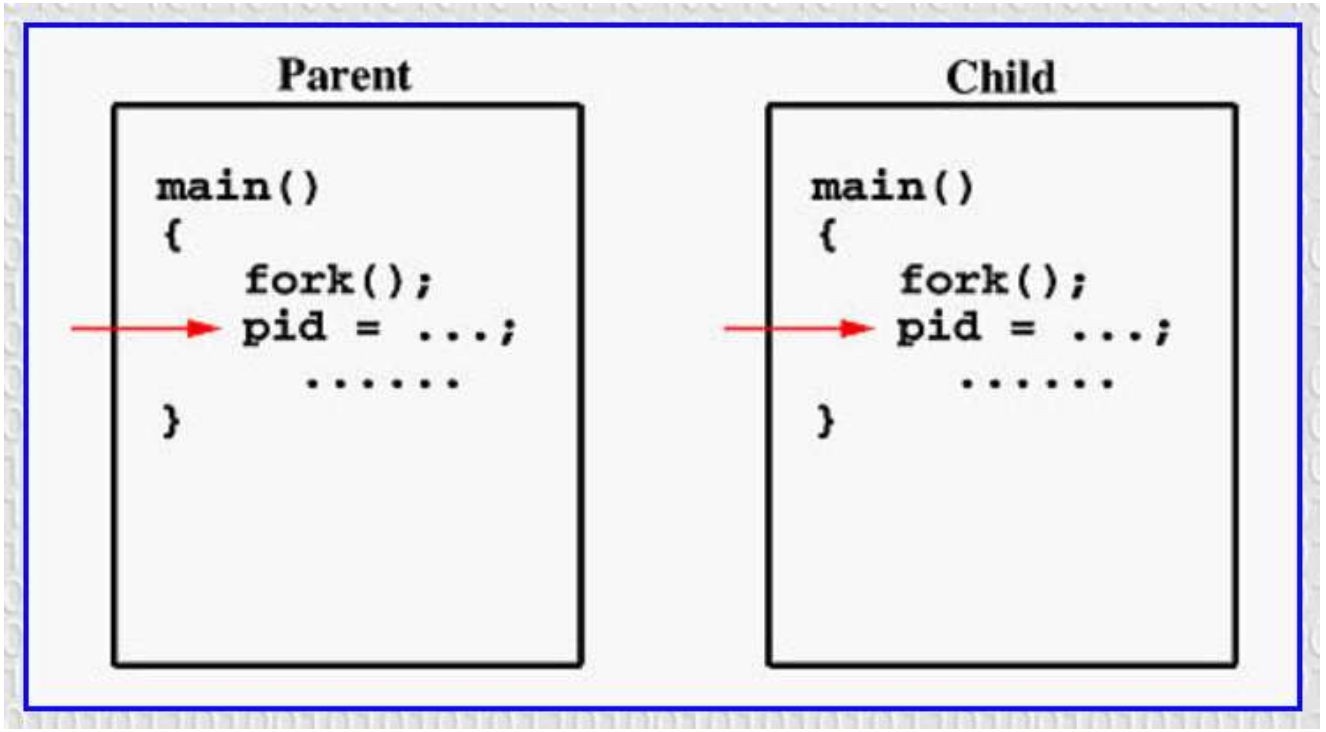
System Calls

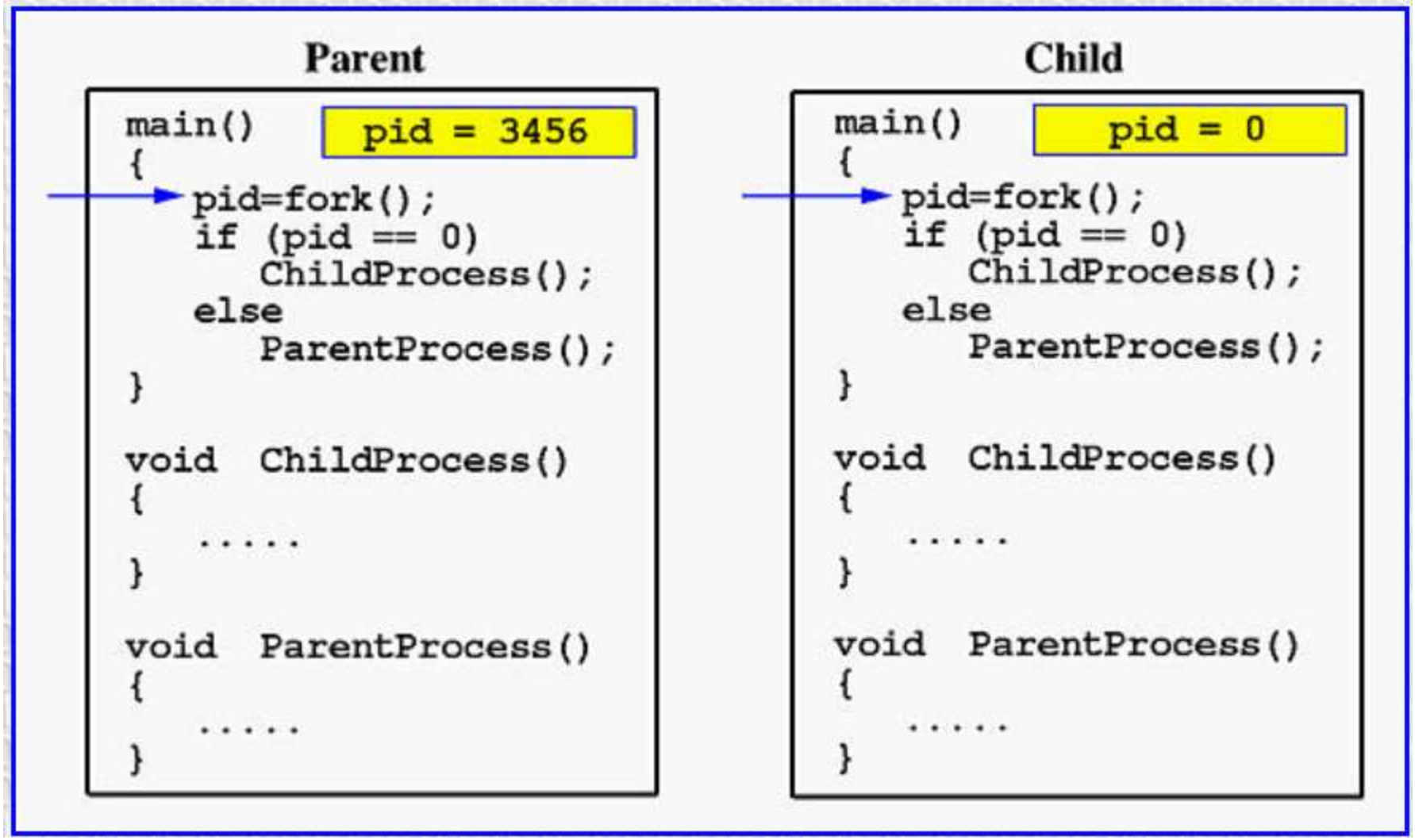
- fork()
- exec()
- wait()
- exit()
- getpid(), getppid()
- getpgrp()

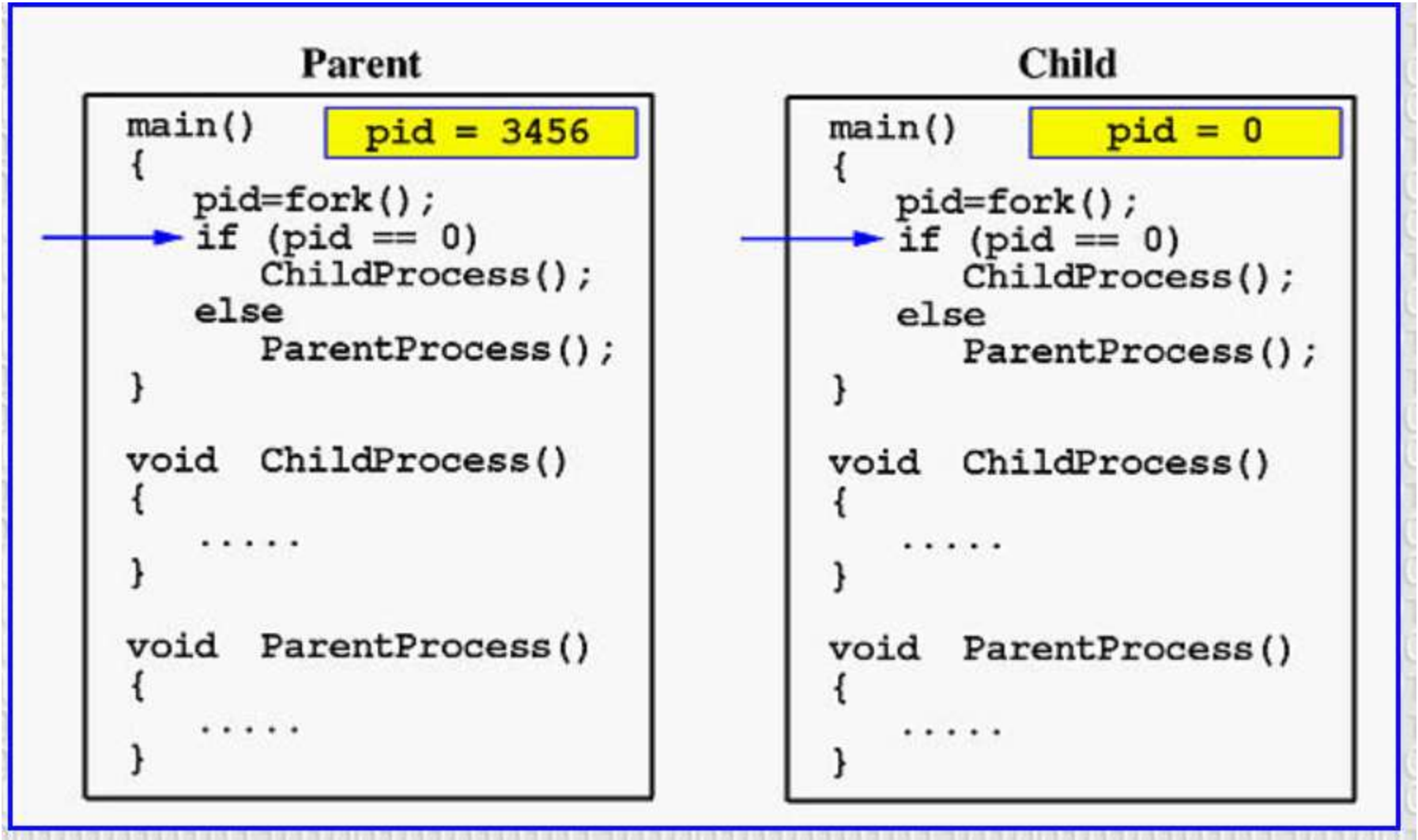
The "fork()" system call

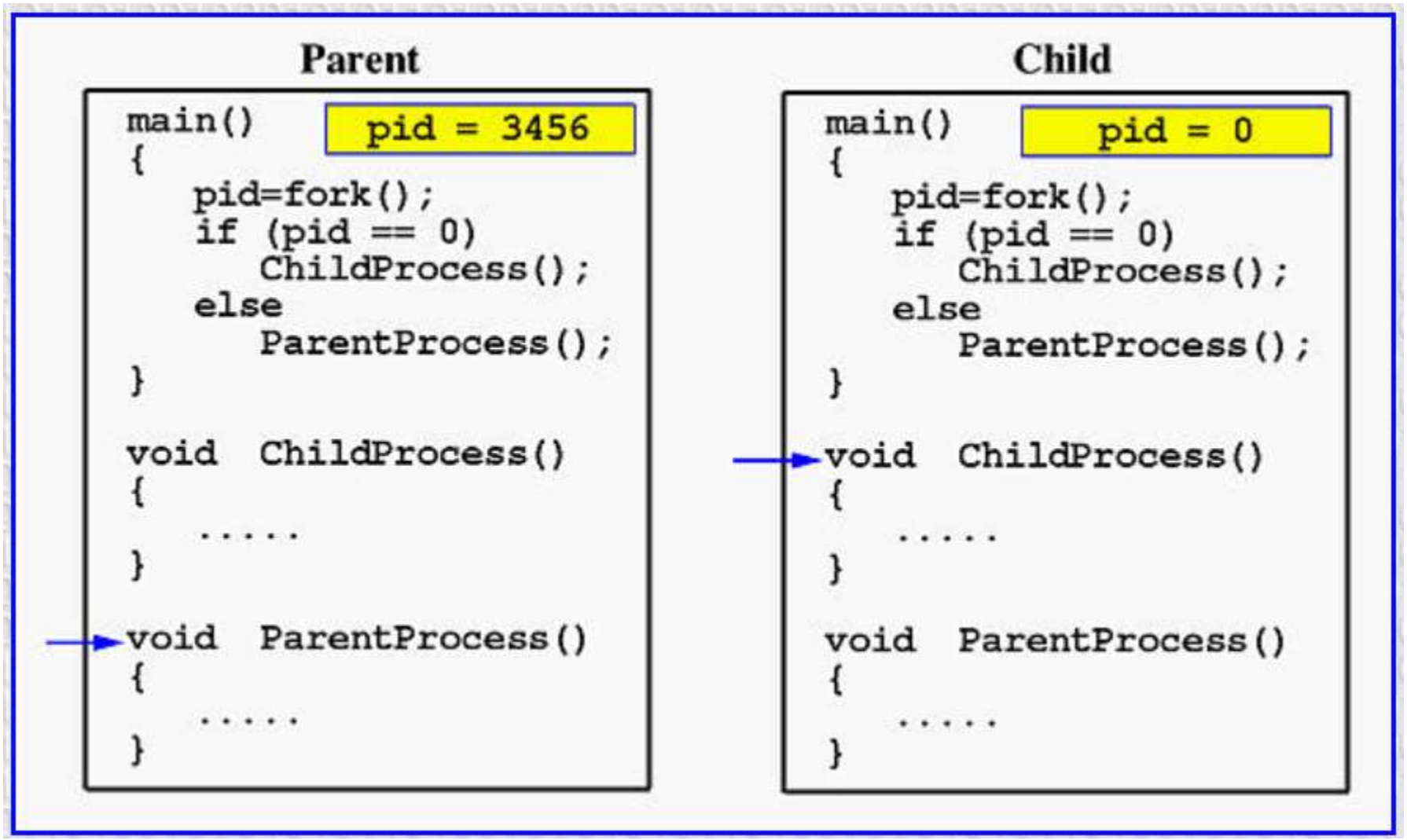
- A process calling `fork()` spawns a child process.
- The child is almost an identical *clone* of the parent:
  - Program Text (segment .text)
  - Stack (ss)
  - PCB(eg. registers)
  - Data (segment .data)
- The `fork()` is called once, but returns twice!
- After `fork()` both the parent and the child are executing the same program.

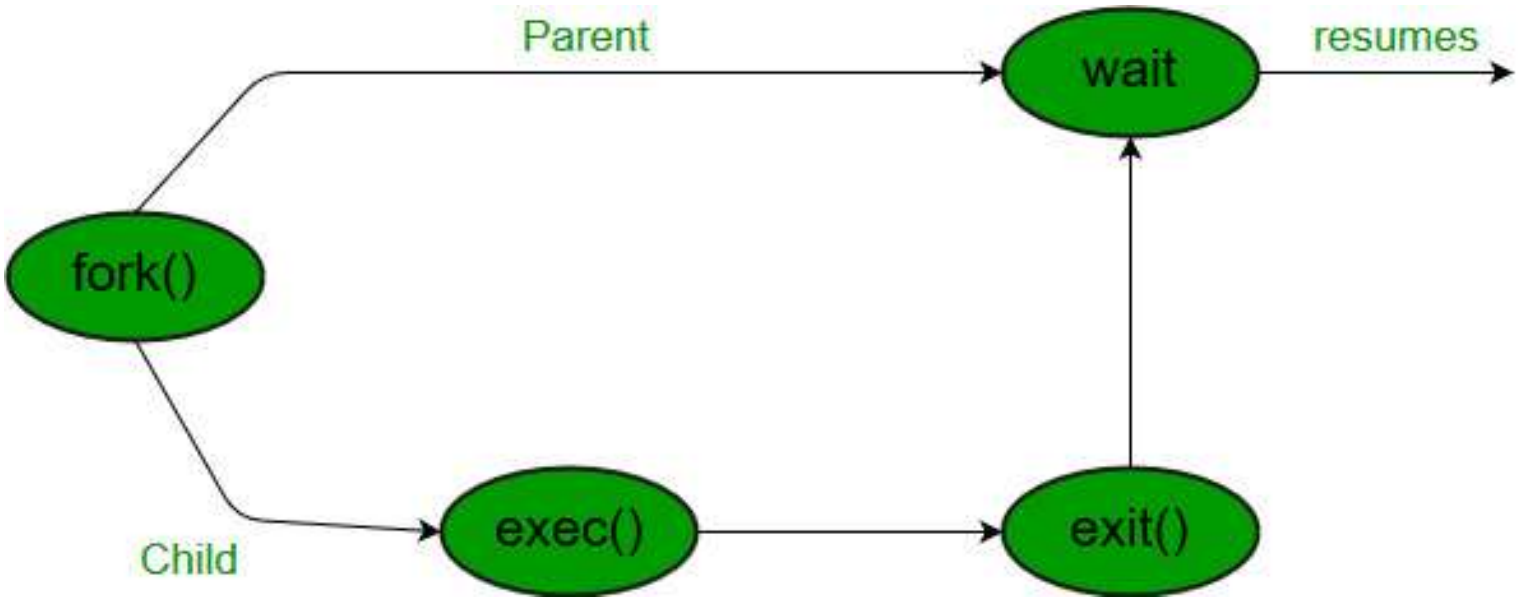

Parent
main()
{
fork();
pid = ...;
.....
}


Parent
main()
{
fork();
pid = ...;
.....
}
Child
main()
{
fork();
pid = ...;
.....
}


Parent
main()
pid = 3456
{
pid=fork();
if (pid == 0)
ChildProcess();
else
ParentProcess();
}
void ChildProcess()
{
.....
}
void ParentProcess()
{
.....
}
Child
main()
pid = 0
{
pid=fork();
if (pid == 0)
ChildProcess();
else
ParentProcess();
}
void ChildProcess()
{
.....
}
void ParentProcess()
{
.....
}

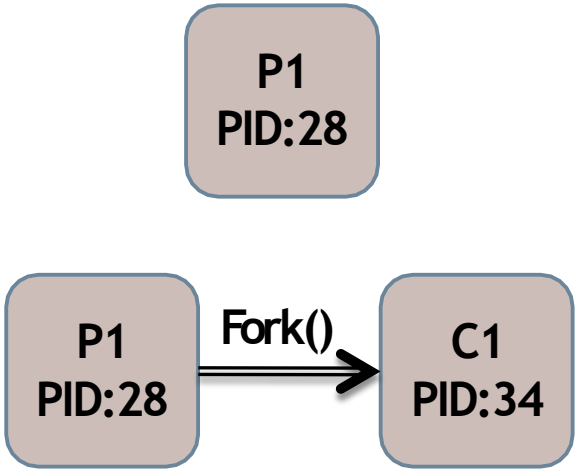

Parent
main()
pid = 3456
{
pid=fork();
if (pid == 0)
ChildProcess();
else
ParentProcess();
}
void ChildProcess()
{
.....
}
void ParentProcess()
{
.....
}
Child
main()
pid = 0
{
pid=fork();
if (pid == 0)
ChildProcess();
else
ParentProcess();
}
void ChildProcess()
{
.....
}
void ParentProcess()
{
.....
}


Parent
main()
pid = 3456
{
pid=fork();
if (pid == 0)
ChildProcess();
else
ParentProcess();
}
void ChildProcess()
{
.....
}
void ParentProcess()
{
.....
}
Child
main()
pid = 0
{
pid=fork();
if (pid == 0)
ChildProcess();
else
ParentProcess();
}
void ChildProcess()
{
.....
}
void ParentProcess()
{
.....
}


Parent
wait
resumes
fork()
exec()
exit()
Child

The "fork()" system call - PID

- pid<0: the creation of a child process was unsuccessful.
- pid==0: the newly created child.
- pid>0: the *process ID* of the child process passes to the parent.



Consider a piece of program

```
...
pid_t pid = fork();
printf("PID: %d\n", pid);
...
```

The parent will print:

```
PID: 34
```

And the child will **always** print:

```
PID: 0
```

“fork()” Example

```
                                                    void main() {
                                                                int i;
        printf("simpfork: pid =  %d\n", getpid());   i
         = fork();
        printf("Did a fork. It returned %d.
                        getpid =  %d. getppid = %d\n“
                        , i, getpid(),  getppid());
}
```

Returns:

simpfork: pid = 914

Did a fork.It returned 915. getpid=914.

getppid=381 Did a fork. It returned 0.

getpid=915. getppid=914

When simpfork is executed, it has a pid of 914. Next it calls **fork()** creating a duplicate process with a pid of 915. The parent gains control of the CPU, and returns from **fork()** with a return value of the 915 -- **this is the child's pid.** It prints out this return value, its own pid, and the pid of Cshell, which is 381.

Note: there is no guarantee which process gains control of the CPU first after a **fork()**. It could be the parent, and it could be the child.

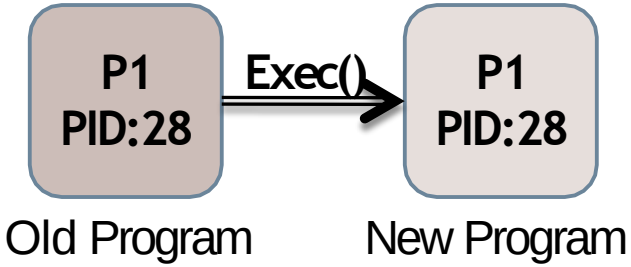
The "exec()" System Call

- The `exec()` call replaces a current process' image with a new one (i.e. loads a new program within current process).
- The new image is either regular executable **binary file** or a **shell script**.
- There's **not** a syscall under the name `exec()`. By `exec()` we usually refer to a family of calls:
  - int execl(char *path, char *arg, ...);
  - int execv(char *path, char *argv[]);
  - int execle(char *path, char *arg, ..., char *envp[]);
  - int execve(char *path, char *argv[], char *envp[]);
  - int execlp(char *file, char *arg, ...);
  - int execvp(char *file, char *argv[]);

Where l=argument list, v=argument vector, e=environmental vector, and p=search path.

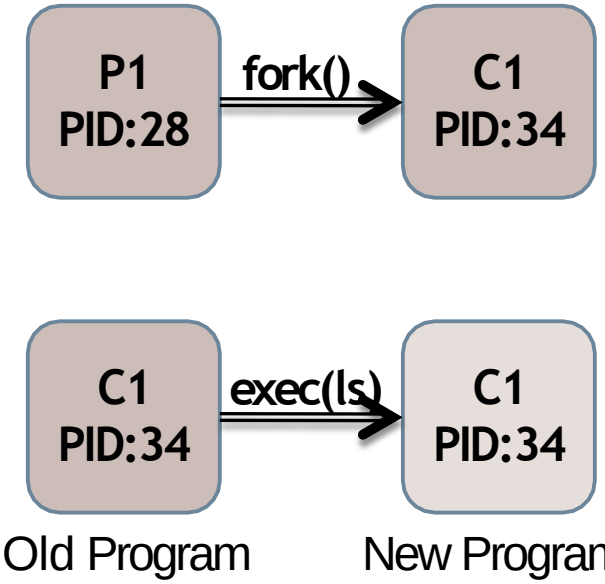
The "exec()" System Call

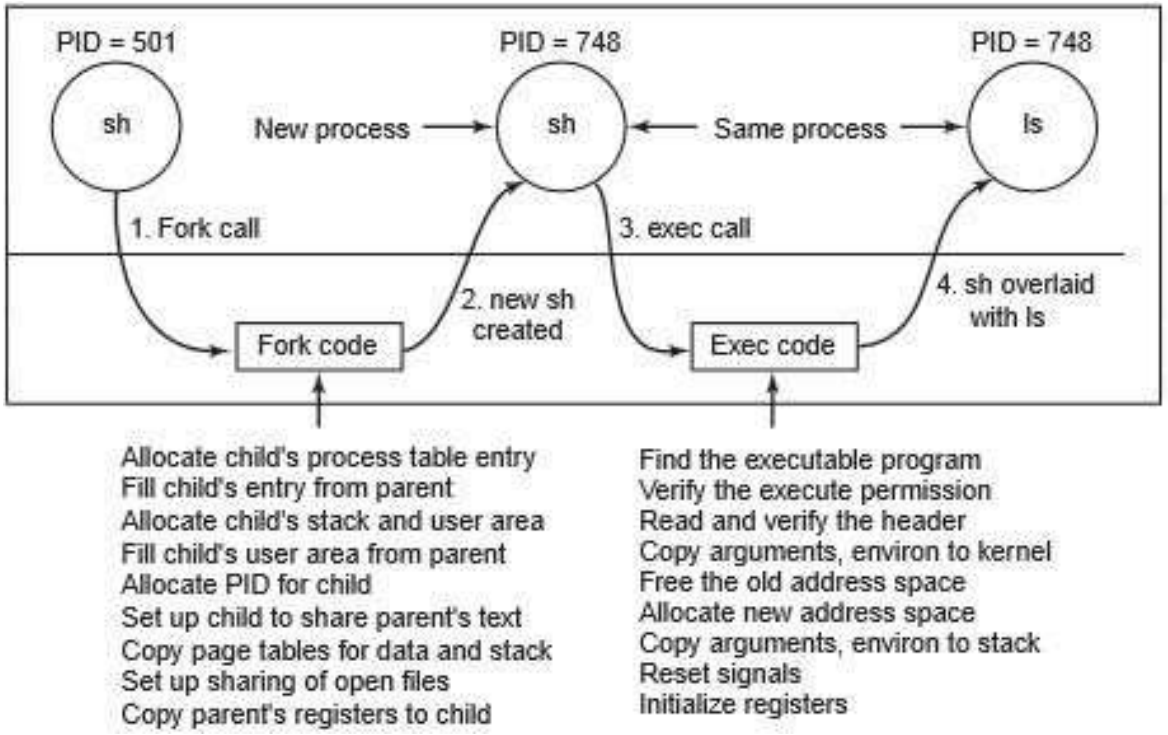
- Upon success, `exec()` never returns to the caller. It replaces the current process image, so it cannot return anything to the program that made the call. If it does return, it means the call failed. Typical reasons are: non-existent file (bad path) or bad permissions.
- Arguments passed via `exec()` appear in the `argv[]` of the `main()` function.
- 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.
- For more info: `man 3 exec;`



“fork()” and “exec()” combined

- Often after doing `fork()` we want to load a new program into the child. *E.g.*: a shell





The "wait()" system call

- Forces the parent to suspend execution, i.e. wait for its children or a specific child to die (*terminate).*
- When the child process dies, it returns an exit status to the operating system, which is then returned to the waiting parent process. The parent process then resumes execution.
- A child process that dies but is never waited on by its parent becomes a **zombie process**. Such a process continues to exist as an entry in the system process table even though it is no longer an actively executing program.

The "wait()" system call

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

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

- The `wait()` causes the parent to wait for any child process.
- The `waitpid()` waits for the child with specific PID.
  - pid: pid of (child) process that the calling process waits for.
  - status: a pointer to the location where status information for the terminating process is to be stored.
  - options: specifies optional actions.
- The return value is:
  - PID of the exited process, if no error
  - (-1) if an error has happened

- When wait() returns they also define **exit status** (which tells us, a process why terminated) via pointer, If status are not **NULL**.
- If any process has no child process then wait() returns immediately “-1”.

Child status information:

Status information about the child reported by wait is more than just the exit status of the child, it also includes

- normal/abnormal termination
- termination cause
- exit status

The "exit()" system call

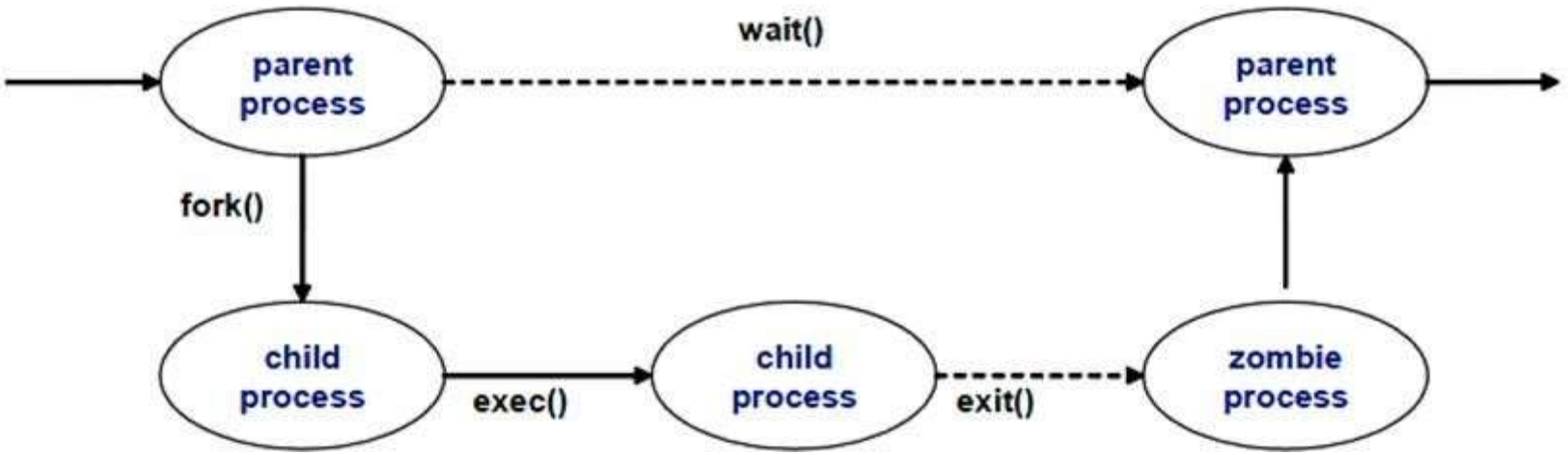
- This call **gracefully** terminates process execution. Gracefully means it does clean up and release of resources, and puts the process into the **zombie state**.
- By calling `wait()`, the parent cleans up all its zombie children.
- When the child process dies, an exit status is returned to the operating system and a signal is sent to the parent process. The exit status can then be retrieved by the parent process via the ***wait*** system call.

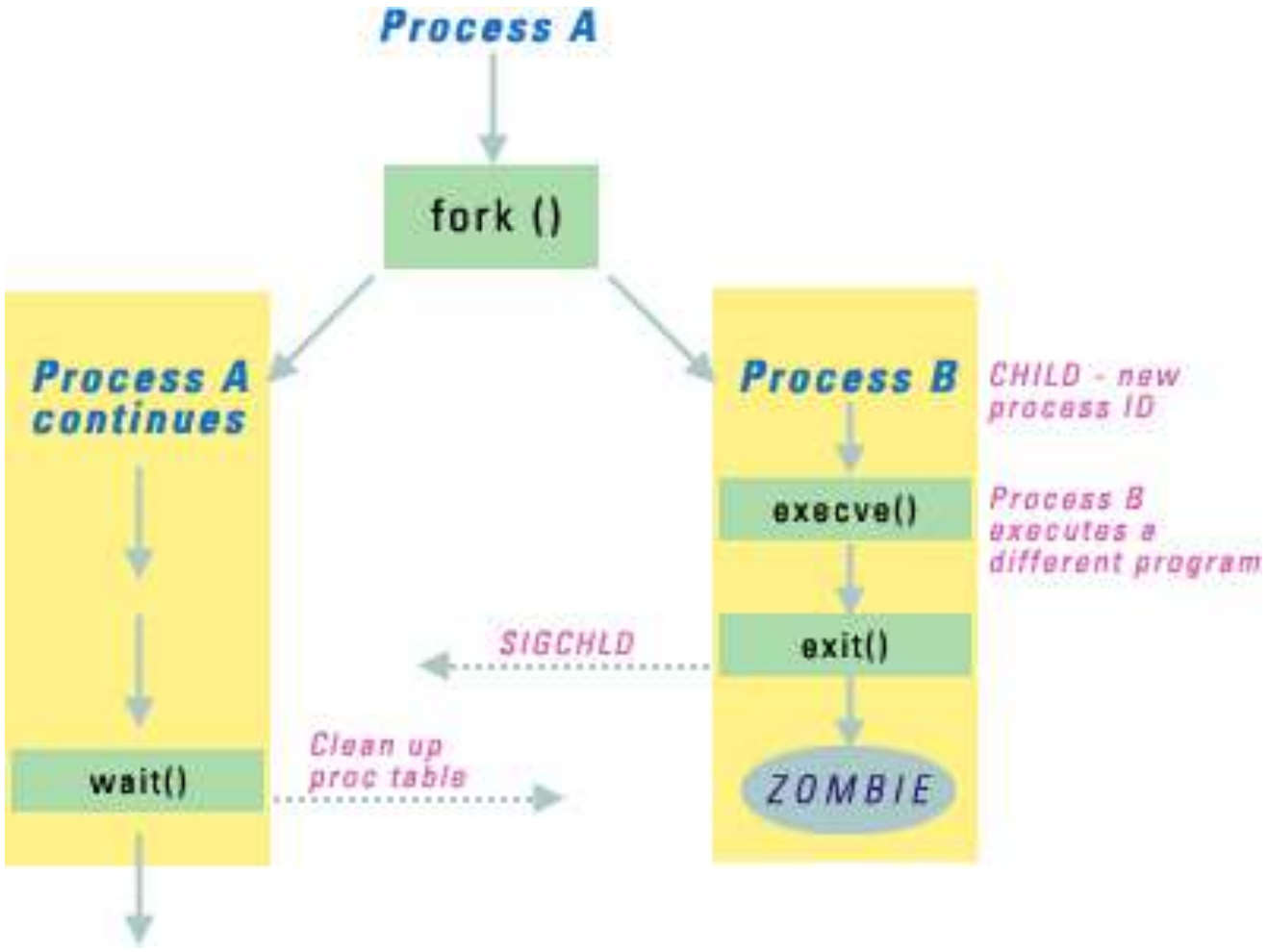
```
void exit ( int status );
```

The process states

- **Zombie**: has completed execution, still has an entry in the process table
- **Orphan**: parent has finished or terminated while this process is still running
- **Daemon**: runs as a background process, not under the direct control of an interactive user

A zombie process




Process A
fork ()
Process A continues
wait()
Process B
CHILD - new process ID
execve()
Process B executes a different program
exit()
SIGCHLD
Clean up proc table
ZOMBIE

https://www.geeksforgeeks.org/wait-system-call-c/
https://www.geeksforgeeks.org/exit-status-child-process-linux/
https://www.softprayog.in/programming/creating-processes-with-fork-and-exec-in-linux
https://www.csl.mtu.edu/cs4411.ck/www/NOTES/process/fork/create.html
https://aljensencprogramming.wordpress.com/2014/03/
https://www.geeksforgeeks.org/understanding-exit-abort-and-assert/
http://www.embhack.com/the-wait-system-call/