# Fork() system call

System call fork() is used to create processes.

The purpose of fork() is to create a new process, which becomes the child process of the caller.

After a new child process is created, both processes will execute the next instruction following the fork() system call.

Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of fork():

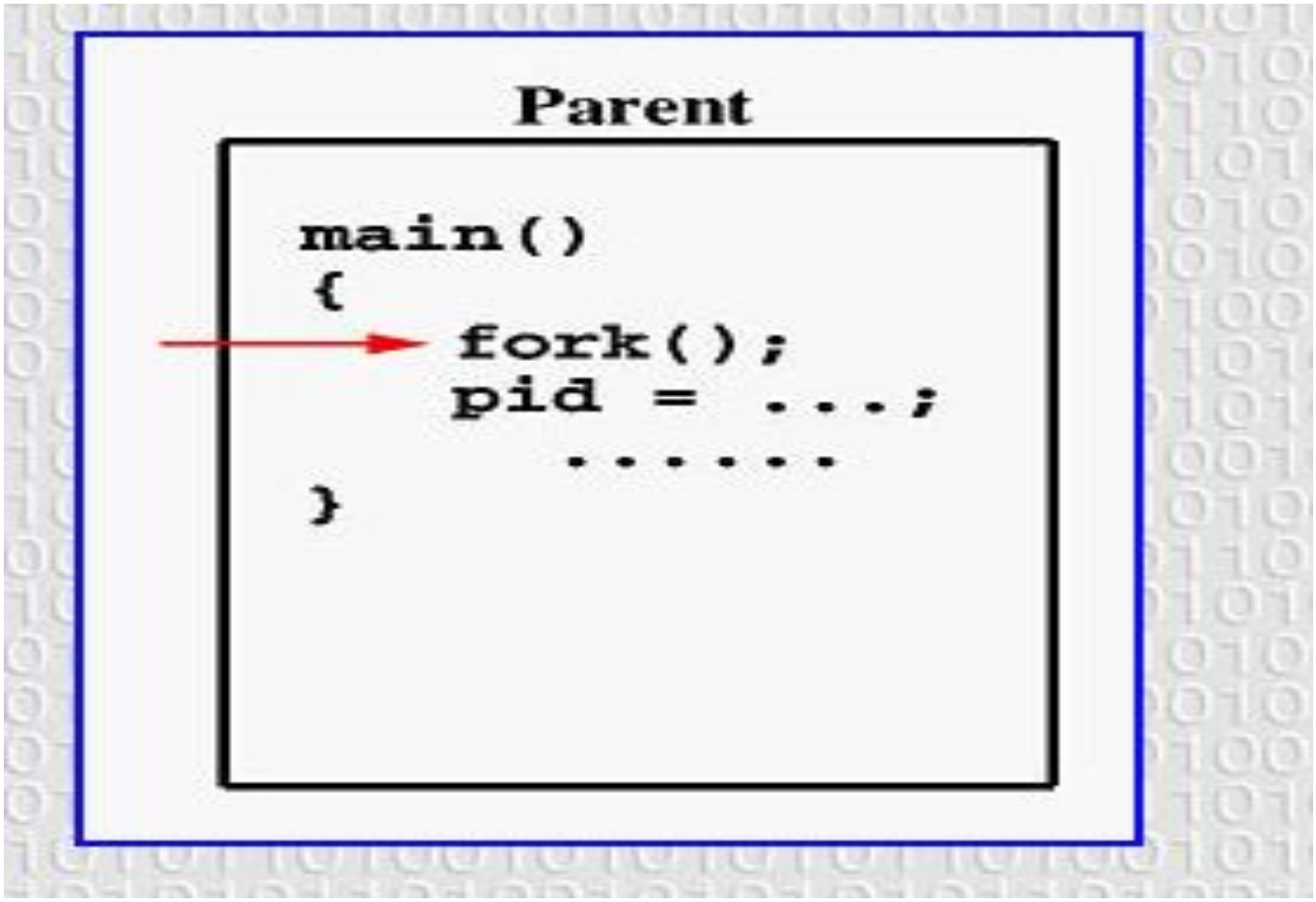
If fork() returns a negative value, the creation of a child process was unsuccessful.

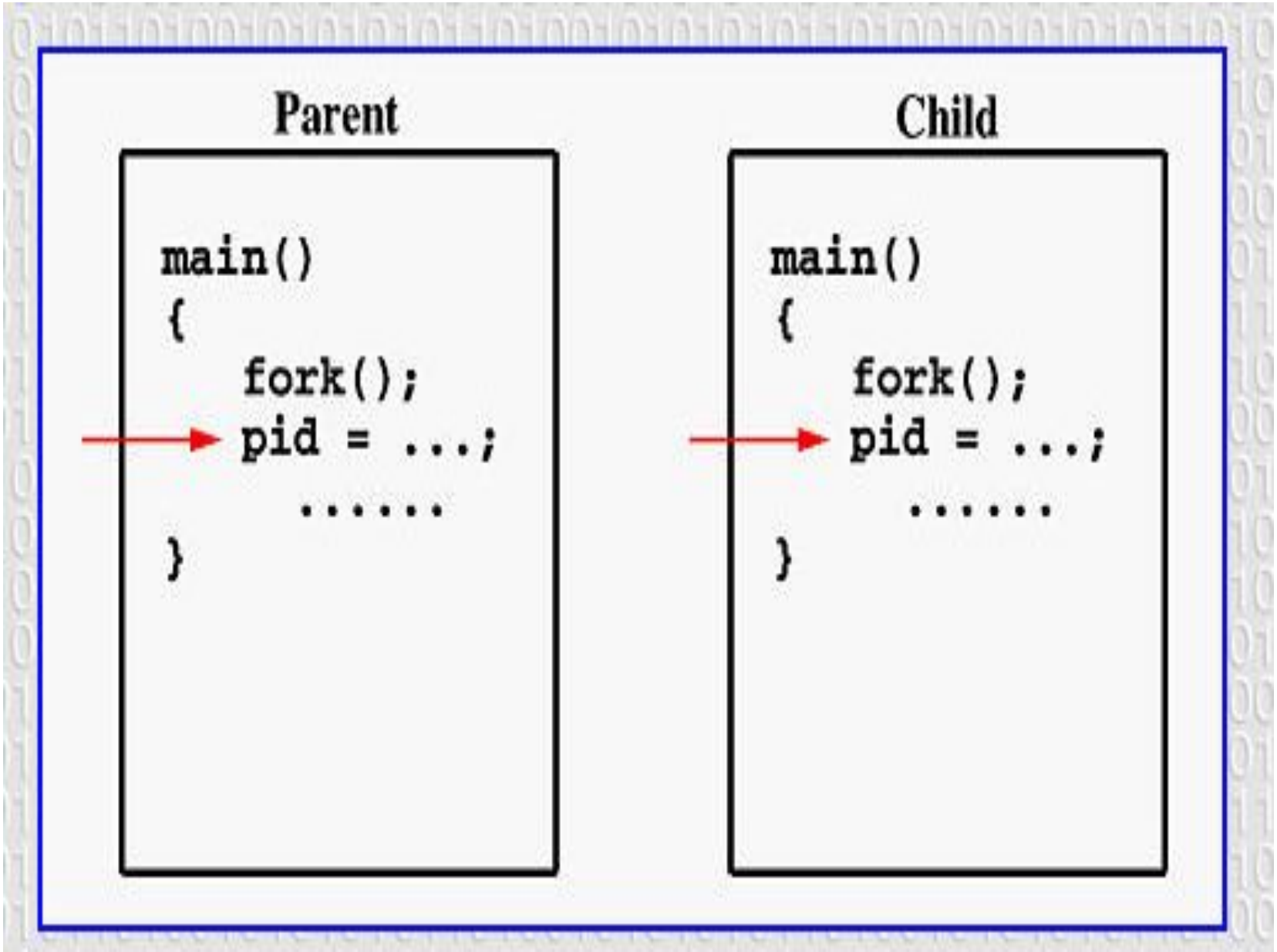
fork() returns a zero to the newly created child process.

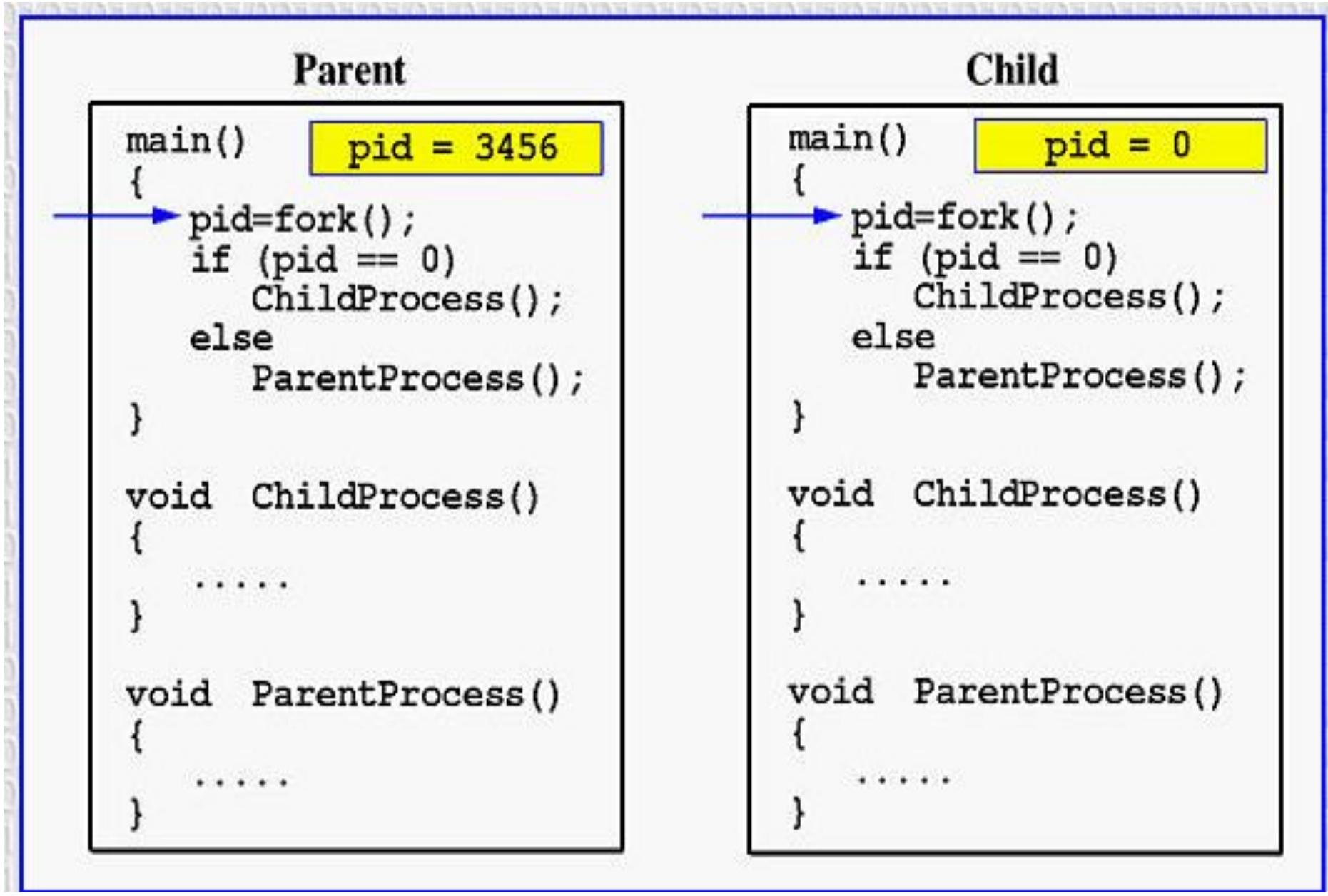
fork() returns a positive value, the process ID of the child process, to the parent. The returned process ID is of type pid_t defined in sys/types.h

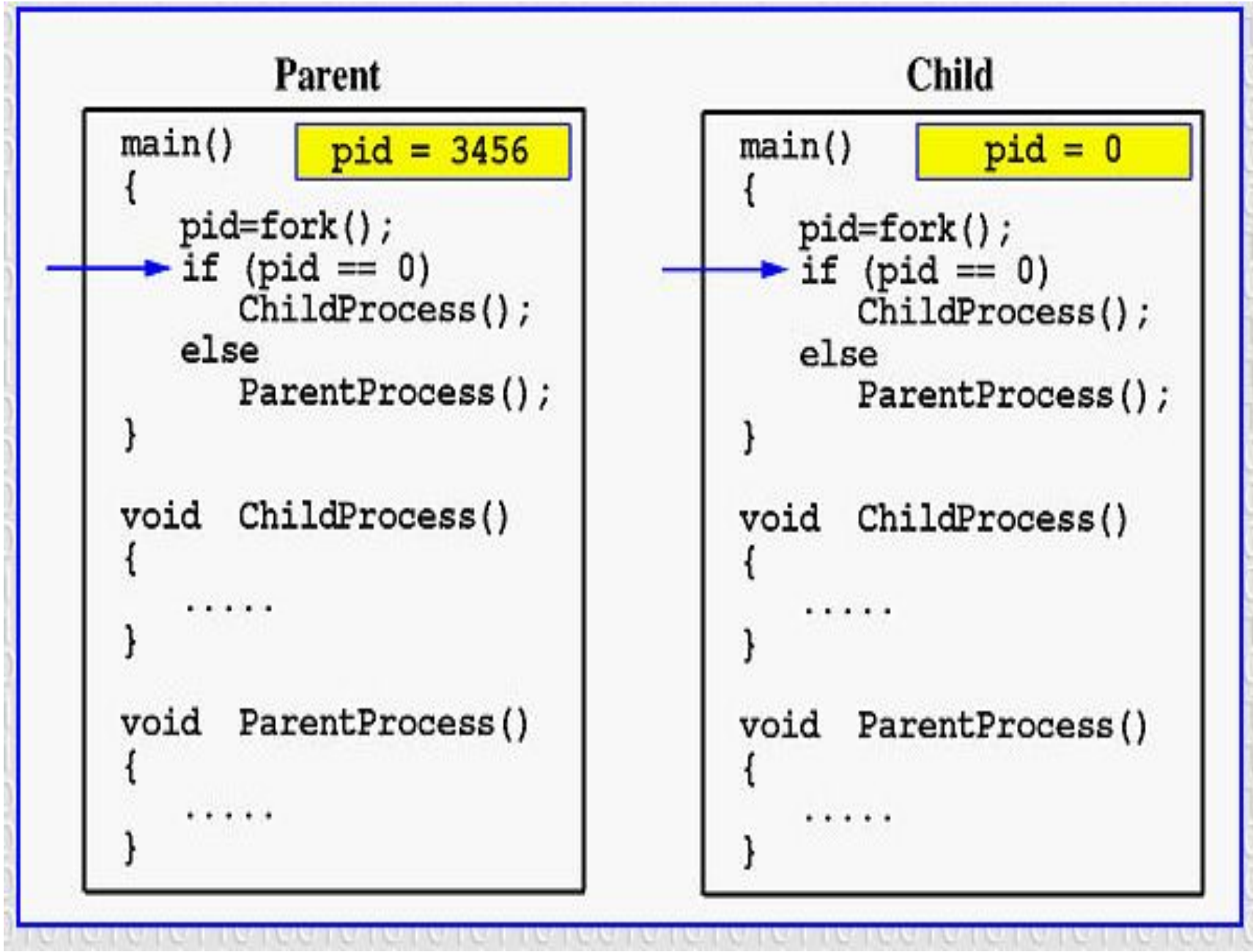
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int) getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int) getpid
  } else {
    // parent goes down this path (main)
    printf("hello, I am parent of %d (pid:%d)\n",
           rc, (int) getpid());
  }
  return 0;
}
```

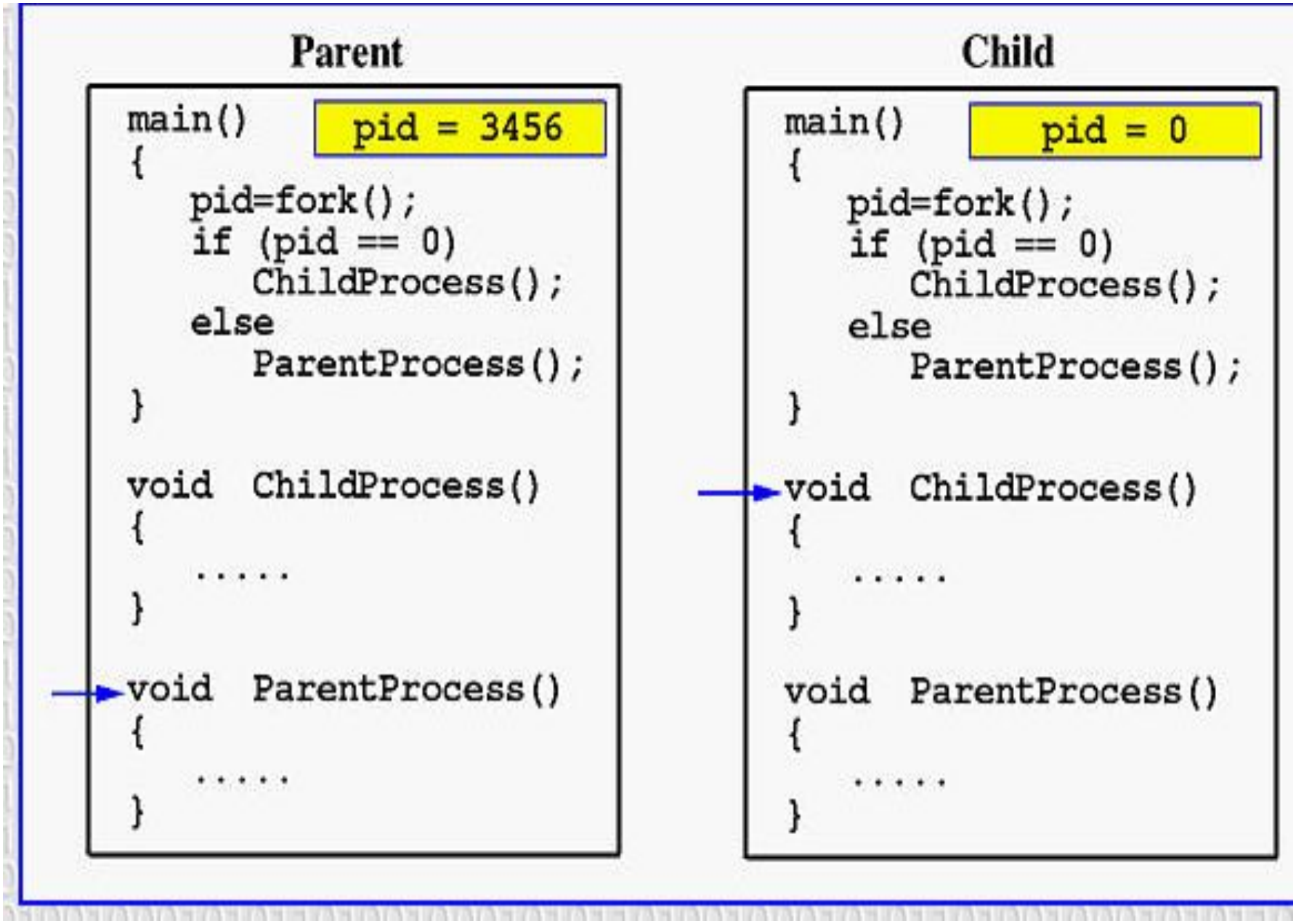
```
prompt> ./p1
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
prompt>
```


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

```
#include “sys/types.h"
int globvar = 6;  /* external variable in initialized data */
int  main(void)
{
int var; /* automatic variable on the stack */
pid_t pid;
var = 88;
printf("before fork\n"); /* we don’t flush stdout */
if ((pid = fork()) < 0)
{
printf("fork error");
}
       else if (pid == 0) { /* child */
        globvar++; /* modify variables */
        var++;
        } else {
                sleep(2); /* parent */
                 }
printf("pid = %ld, glob = %d, var = %d\n", (long)getpid(), globvar,var);
exit(0);
 }
```

- When it first started running, the process prints out a hello world message; included in that message is its process identifier 29146;

- The process calls the fork() system call, which the OS provides as a way to create a new process. The process that is created is an (almost) exact copy of the calling process.

- It now looks like there are two copies of the program p1 running, and both are about to return from the fork() system call.

- The newly-created process (called the child, in contrast to the creating parent) doesn't start running at main()  (note, the "hello, world" message only got printed out once); *rather, it just comes into life as if it had called fork() itself.*

- Specifically, although it now has its own copy of the address space (i.e., its own private memory), its own registers, its own PC, and so forth.

p1.c is not deterministic. When the child process is created, there are now two active processes the parent and the child.

Assuming we have a single CPU (for simplicity), then either the child or the parent might run at that point.

In our example (above), the parent did and thus printed out its message first

# wait( )

- Sometimes, it is quite useful for a parent to wait for a child process to finish what it has been doing. This task is accomplished with the wait() system call.
- In this example (p2.c), the parent process calls wait() to delay its execution until the child finishes executing.
- When the child is done, wait() returns to the parent. Adding a wait() call to the code above makes the output deterministic.

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

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int) getpid());
  int rc = fork();
  if (rc < 0) {          // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) { // child (new process)
    printf("hello, I am child (pid:%d)\n", (int) getpid());
  } else {               // parent goes down this path (main)
    int rc_wait = wait(NULL);
    printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n",
           rc, rc_wait, (int) getpid());
  }
  return 0;
}
```

```
prompt> ./p2
hello world (pid:29266)
hello, I am child (pid:29267)
hello, I am parent of 29267 (rc_wait:29267) (pid:29266)
prompt>
```

# exec ()

- Exec( ) causes another program to be executed.
- When a process calls one of the exec functions, that process is completely replaced by the new program, and the new program starts executing at its main function.
- The process ID does not change across an exec, because a new process is not created;
- exec merely replaces the current process — its text, data, heap, and stack segments — with a brand-new program from disk.

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

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int) getpid());
  int rc = fork();
  if (rc < 0) {          // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) { // child (new process)
    printf("hello, I am child (pid:%d)\n", (int) getpid());
    char *myargs[3];
    myargs[0] = strdup("wc");   // program: "wc" (word count)
    myargs[1] = strdup("p3.c"); // argument: file to count
    myargs[2] = NULL;           // marks end of array
    execvp(myargs[0], myargs);  // runs word count
    printf("this shouldn't print out");
  } else {                  // parent goes down this path (main)
    int rc_wait = wait(NULL);
    printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n",
           rc, rc_wait, (int) getpid());
  }
  return 0;
```

given the name of an executable (e.g., wc), and some arguments (e.g., p3.c), it loads code (and static data) from that executable and overwrites its current code segment (and current static data) with it; the heap and stack and other parts of the memory space of the program are re-initialized.

```
prompt> ./p3
hello world (pid:29383)
hello, I am child (pid:29384)
      29     107    1030 p3.c
hello, I am parent of 29384 (rc_wait:29384) (pid:29383)
prompt>
```

- Then the OS simply runs that program, passing in any arguments as the argv of that process.

- Thus, it does not create a new process; rather, it transforms the currently running program (formerly p3) into a different running program (wc).

- After the exec() in the child, it is almost as if p3.c never ran; a successful call to exec() never returns.