CSE4001: Operating Systems Concepts

Processes

How to create and control processes: Process API

- Create
- Destroy
- Wait
- Miscellaneous control
- Status

Content

- Creating processes with `fork()`

In UNIX, use the fork() system call to create a new process

- This creates an **exact duplicate** of the parent process!!
- Creates and initializes a new PCB
- Creates a new address space
- Copies entire contents of parent's address space into the child
- Initializes CPU and OS resources to a copy of the parent's
- Places new PCB on ready queue

```
int main()
{
    int x = 0;
    fork();
    x = 1;

    return 1;

}
```

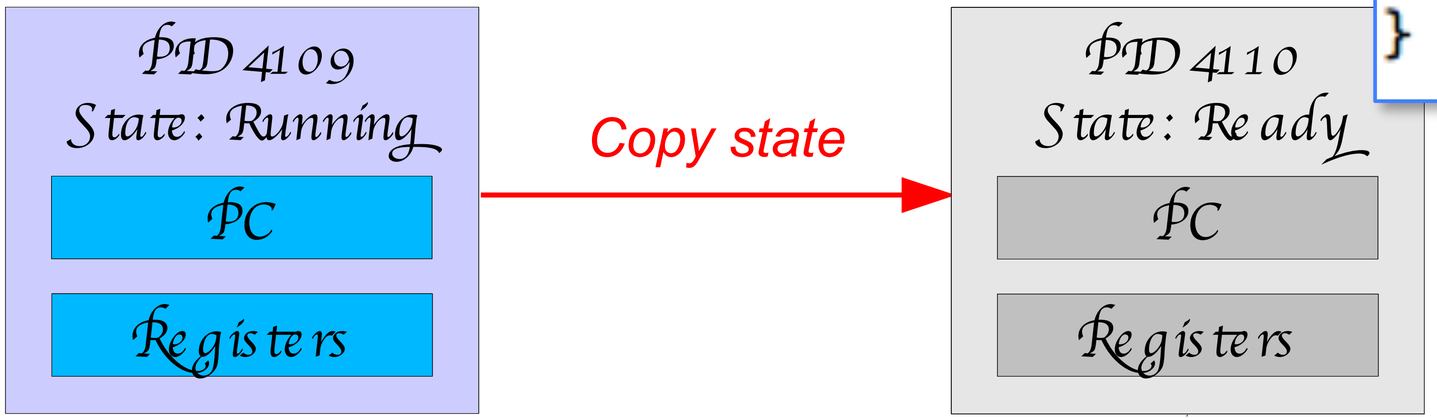
Process calls fork()

In UNIX, use the fork() system call to create a new process

- This creates an **exact duplicate** of the parent process!!
- Creates and initializes a new PCB
- Creates a new address space
- Copies entire contents of parent's address space into the child
- Initializes CPU and OS resources to a copy of the parent's
- Places new PCB on ready queue

```
int main()
{
    int x = 0;
    fork();
    x = 1;

    return 1;

}
```

Process calls fork()

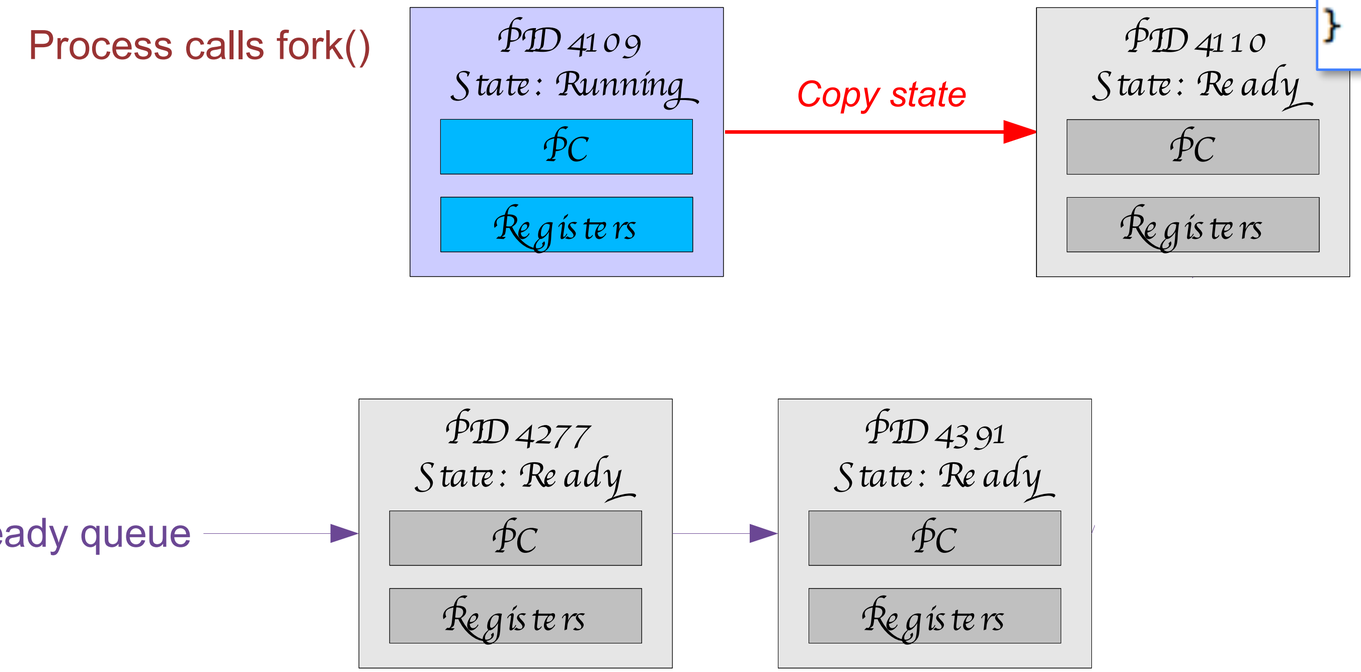


UNIX fork mechanism

In UNIX, use the fork() system call to create a new process

- This creates an **exact duplicate** of the parent process!!
- Creates and initializes a new PCB
- Creates a new address space
- Copies entire contents of parent's address space into the child
- Initializes CPU and OS resources to a copy of the parent's
- Places new PCB on ready queue

```
int main()
{
    int x = 0;
    fork();
    x = 1;

    return 1;

}
```

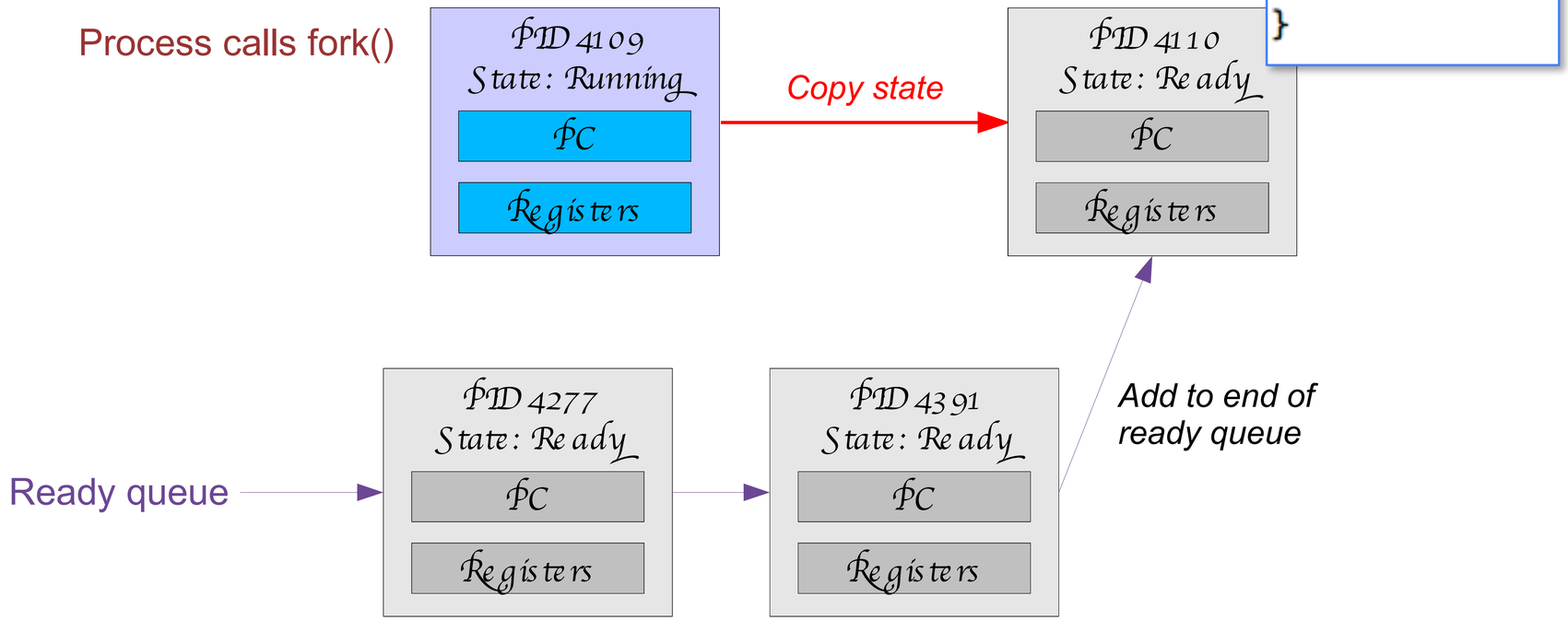


UNIX fork mechanism

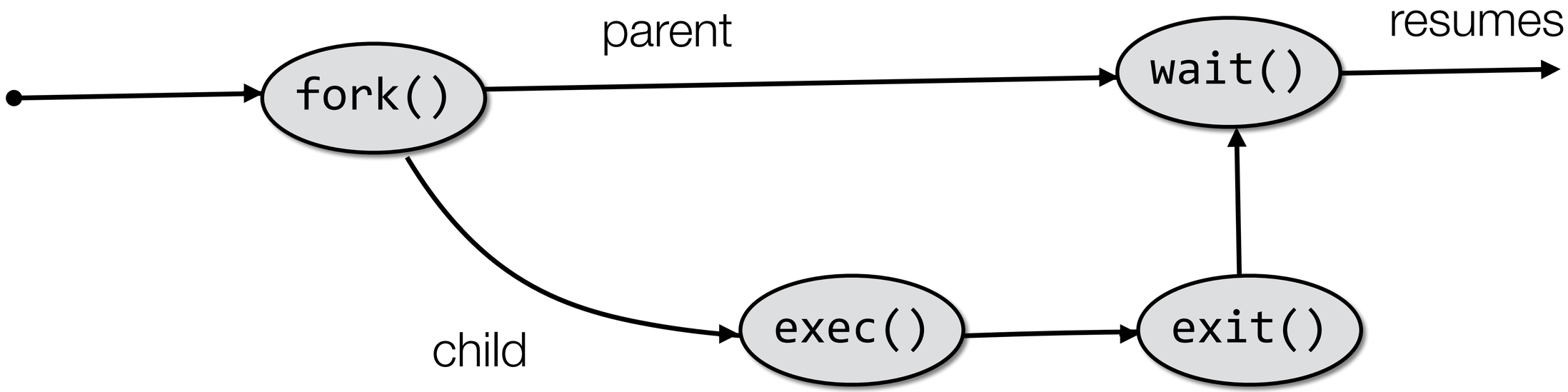
In UNIX, use the fork() system call to create a new process

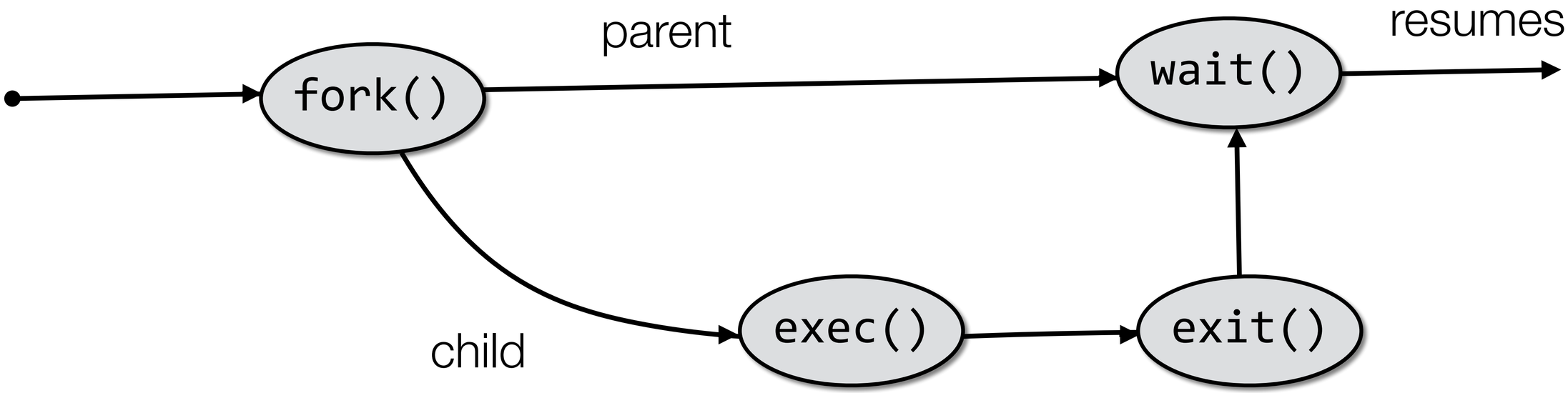
- This creates an **exact duplicate** of the parent process!!
- Creates and initializes a new PCB
- Creates a new address space
- Copies entire contents of parent's address space into the child
- Initializes CPU and OS resources to a copy of the parent's
- Places new PCB on ready queue

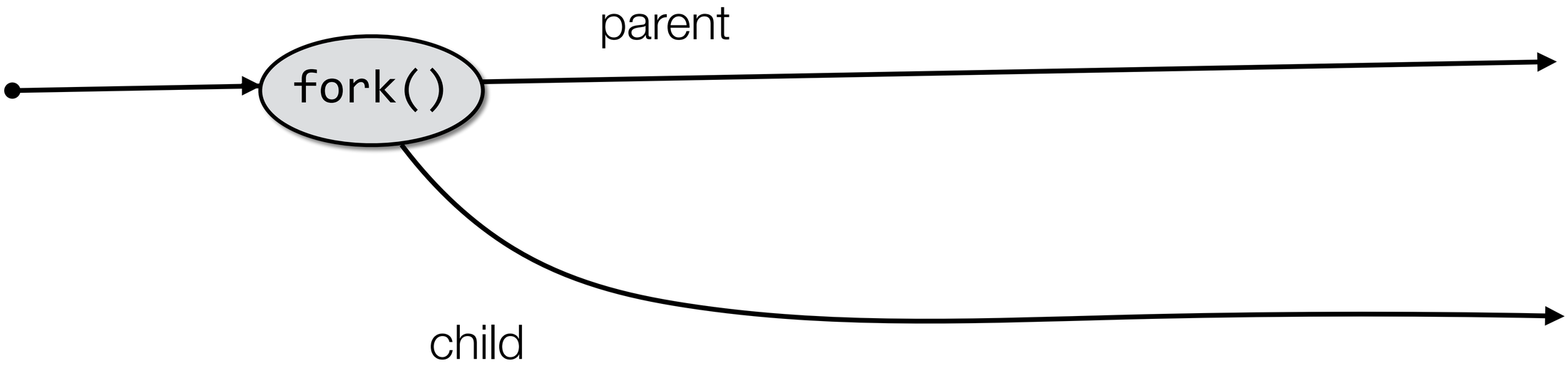
```
int main()
{
    int x = 0;
    fork();
    x = 1;

    return 1;

}
```

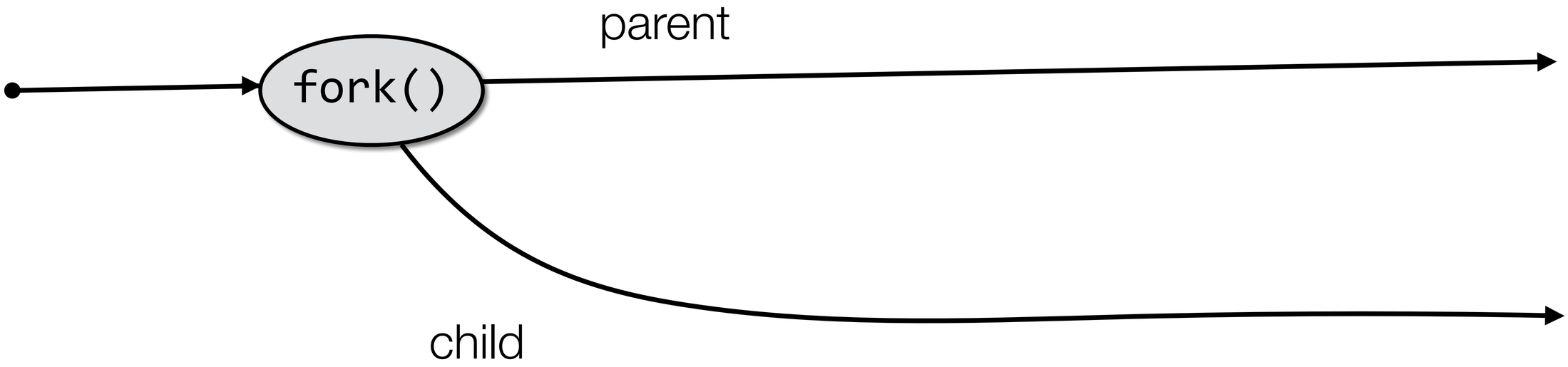


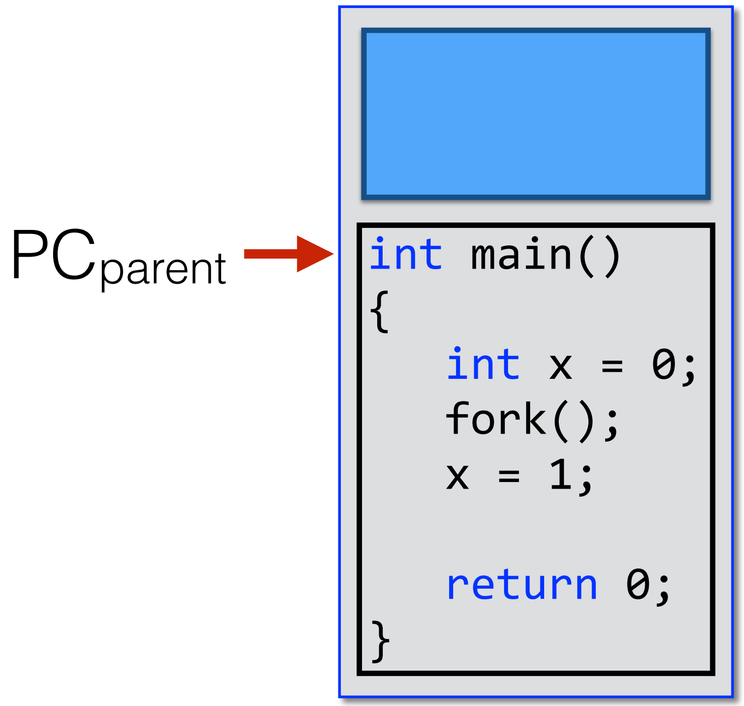
Process API: creation and program execution

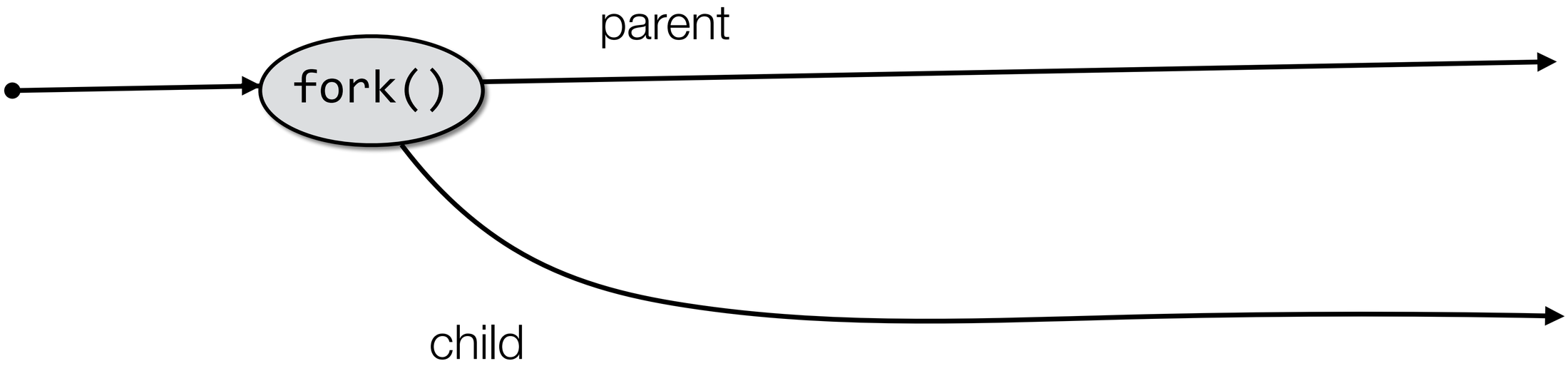


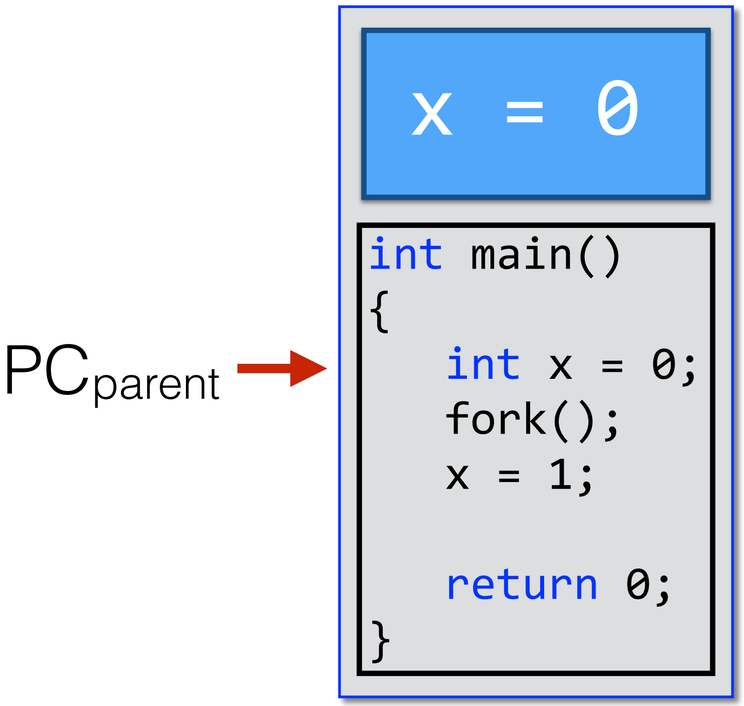

parent
resumes
fork()
wait()
exec()
exit()
child

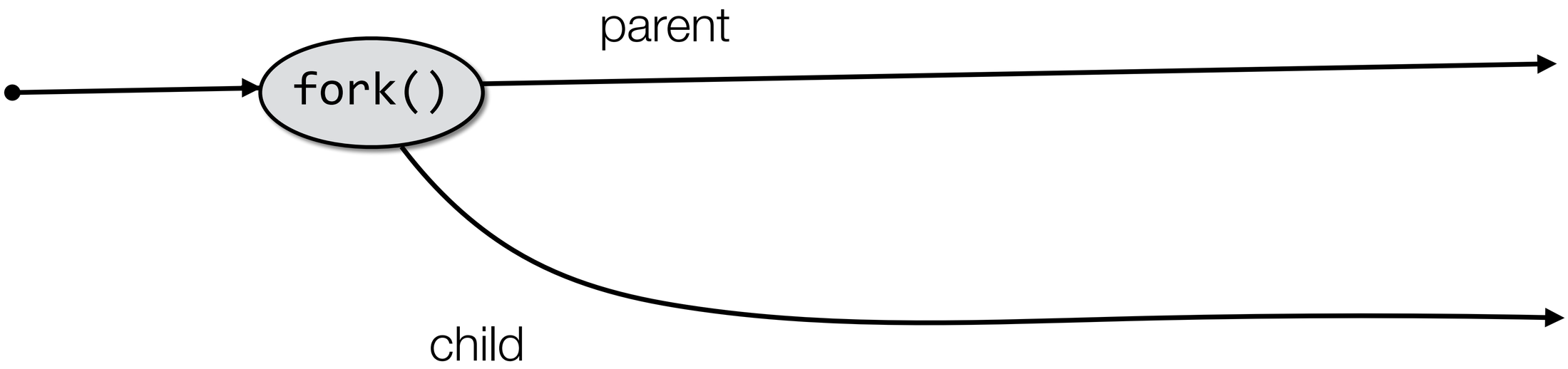

parent
fork()
child

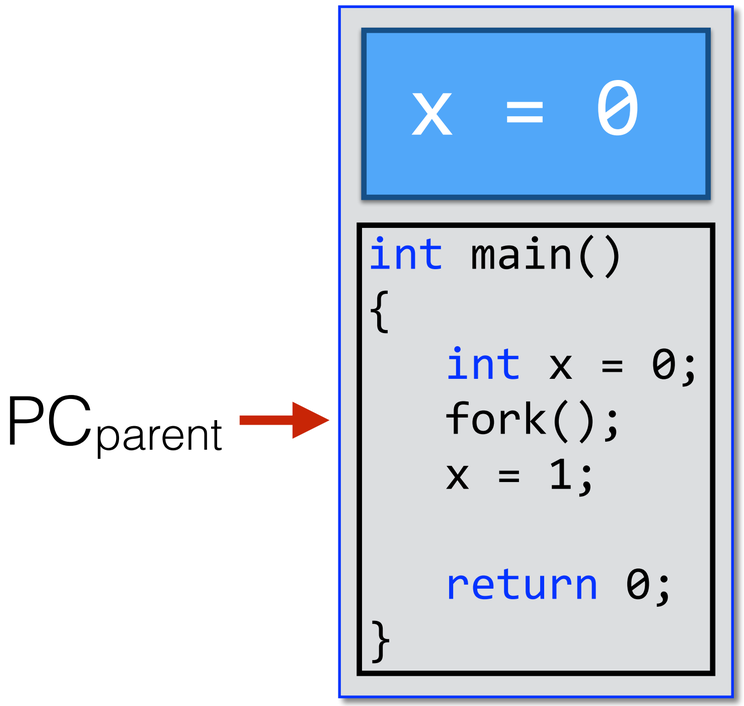

parent
fork()
child

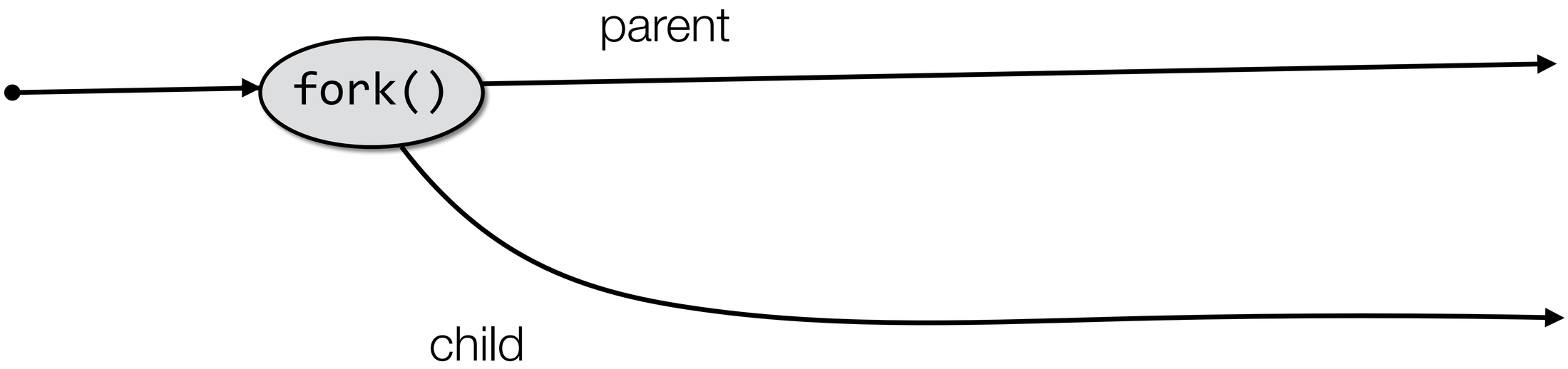

PCparent
int main()
{
int x = 0;
fork();
x = 1;
return 0;
}

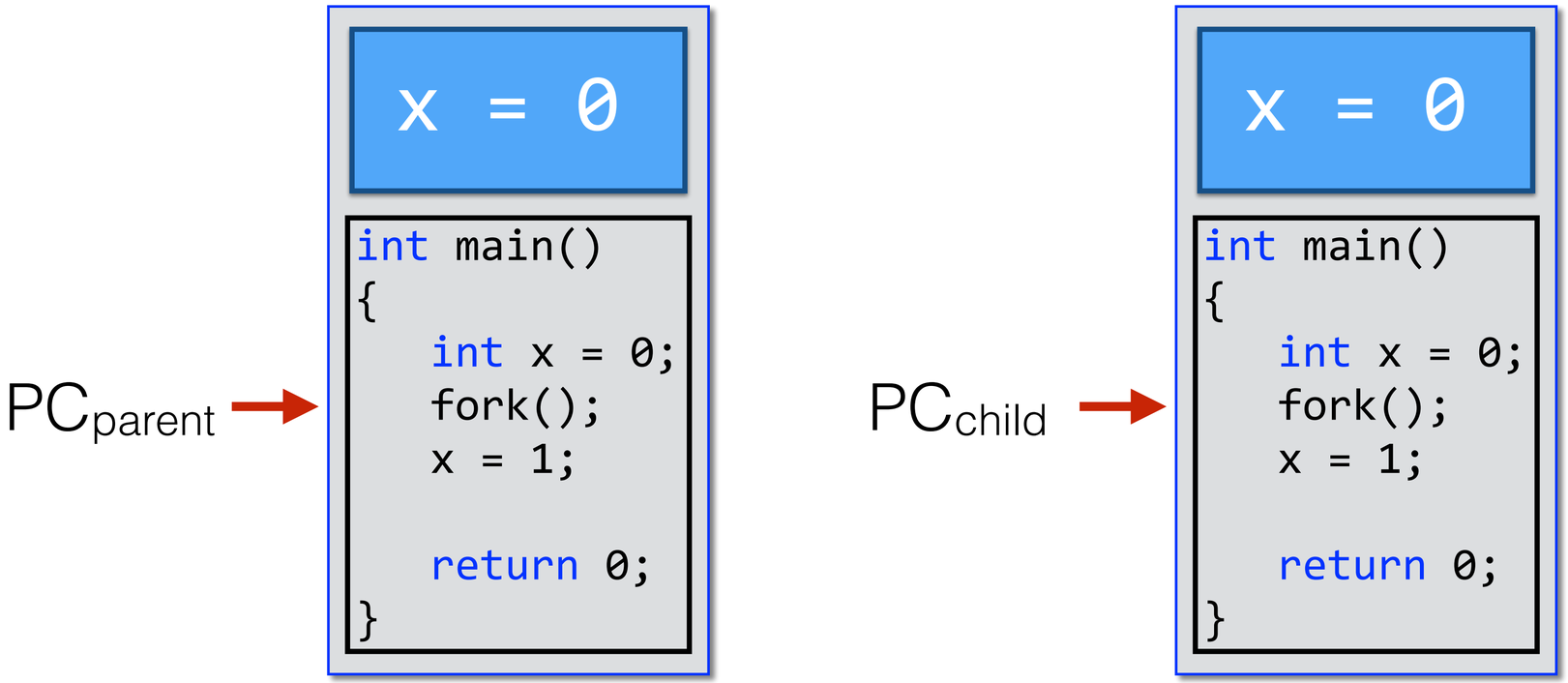

parent
fork()
child

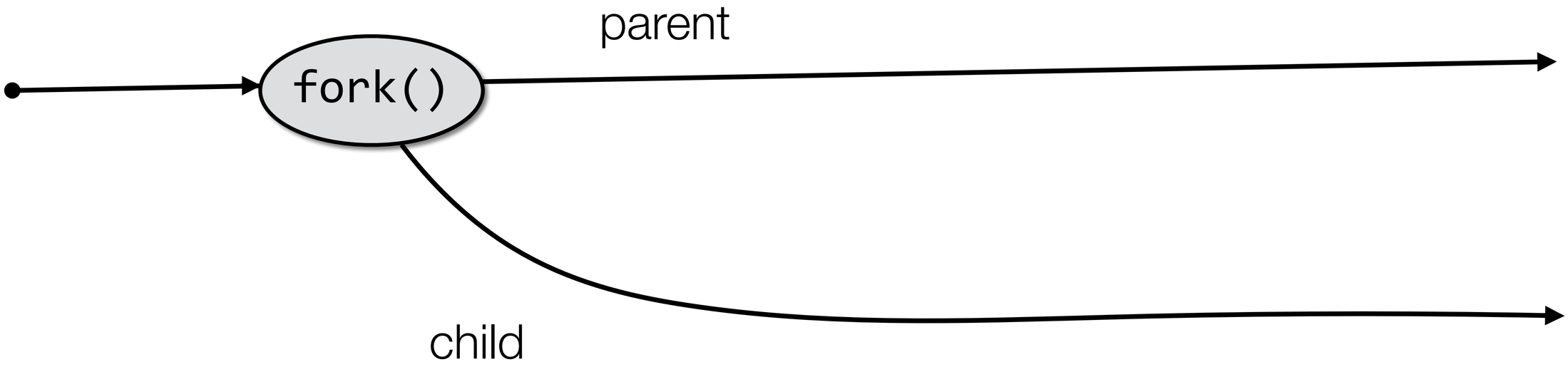

x = 0
int main()
{
int x = 0;
fork();
x = 1;
return 0;
}
PCparent

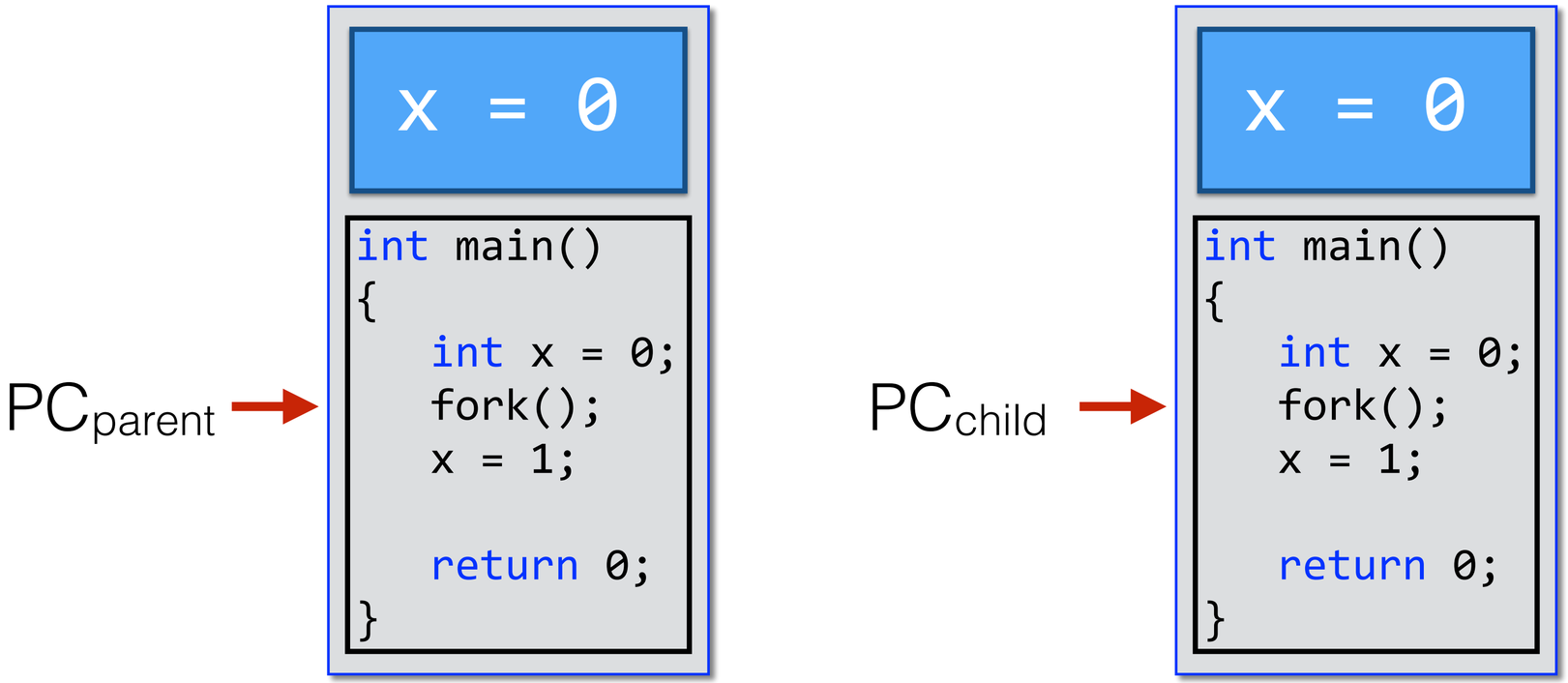

parent
fork()
child


x = 0
int main()
{
int x = 0;
fork();
x = 1;
return 0;
}
PCparent


parent
fork()
child

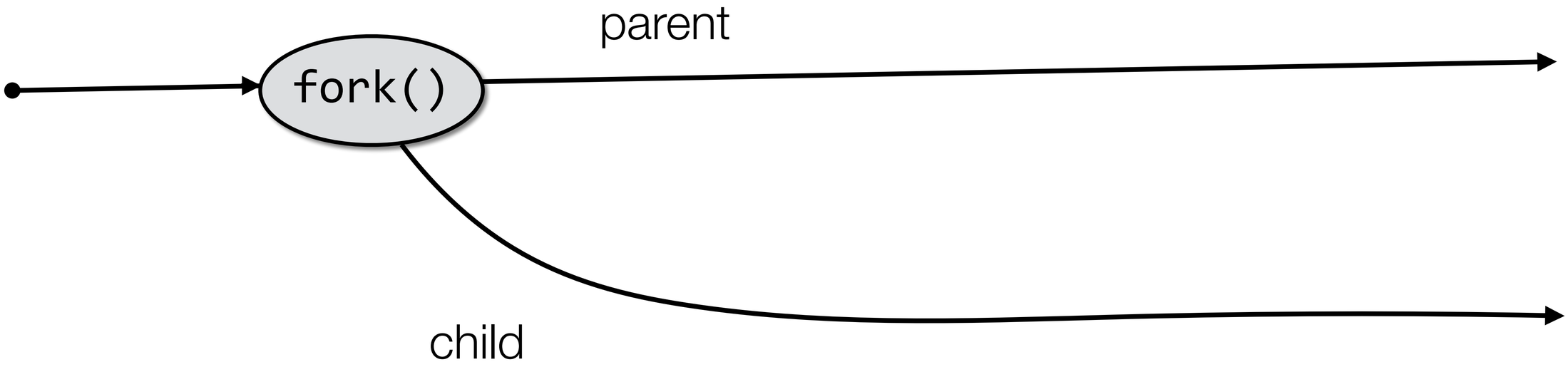

PCparent
x = 0
int main()
{
int x = 0;
fork();
x = 1;
return 0;
}
PCchild
x = 0
int main()
{
int x = 0;
fork();
x = 1;
return 0;
}

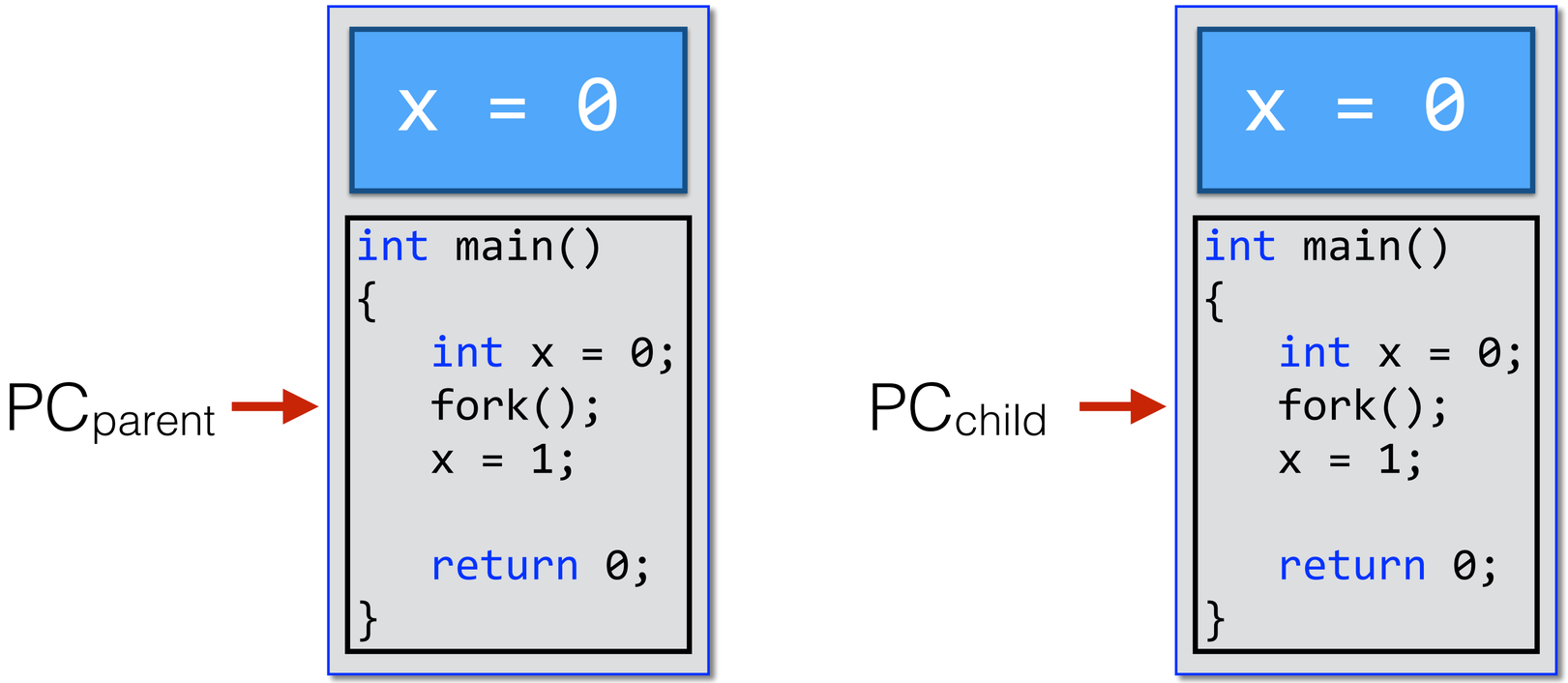




The **`fork()`** call is issued. A new process is created (child).

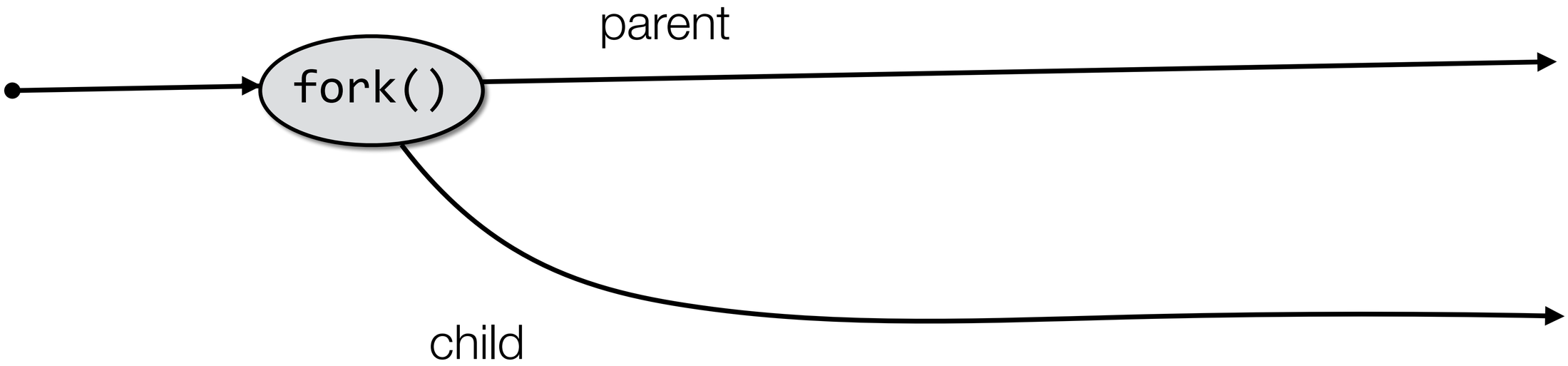
What are the states of the parent and child processes?





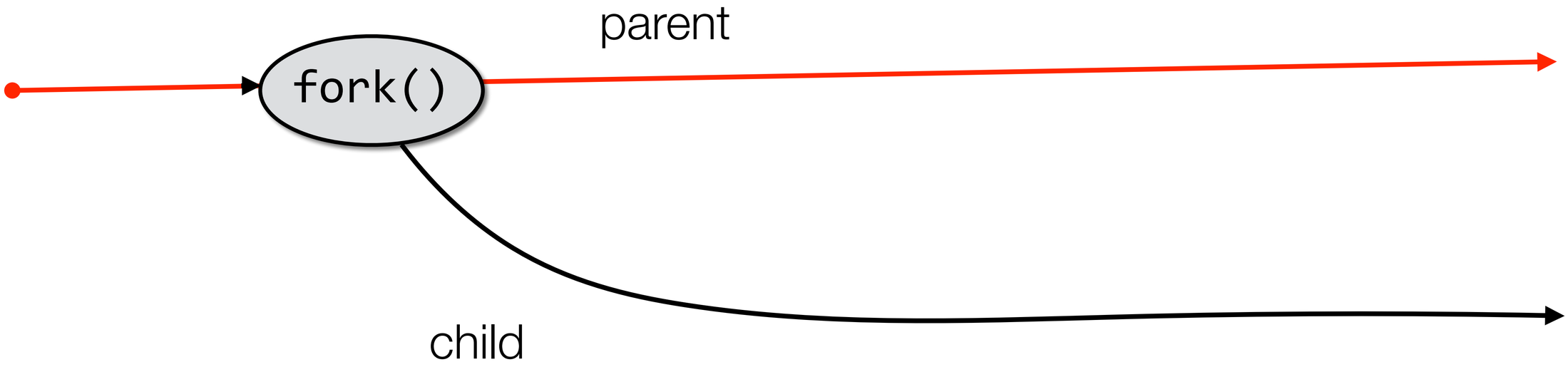
The `fork()` call is issued. A new process is created (child).

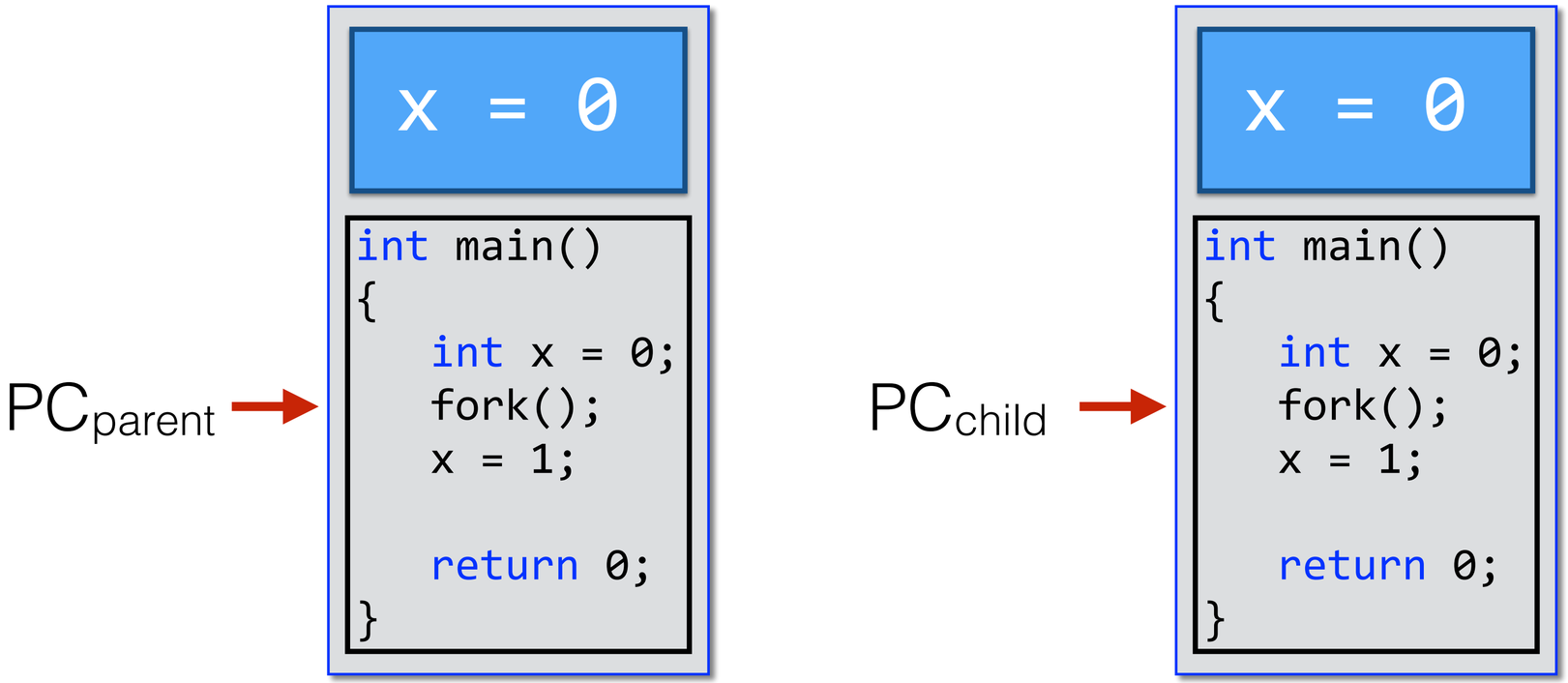
Which one of the two process will be scheduled to run first?



After Creation

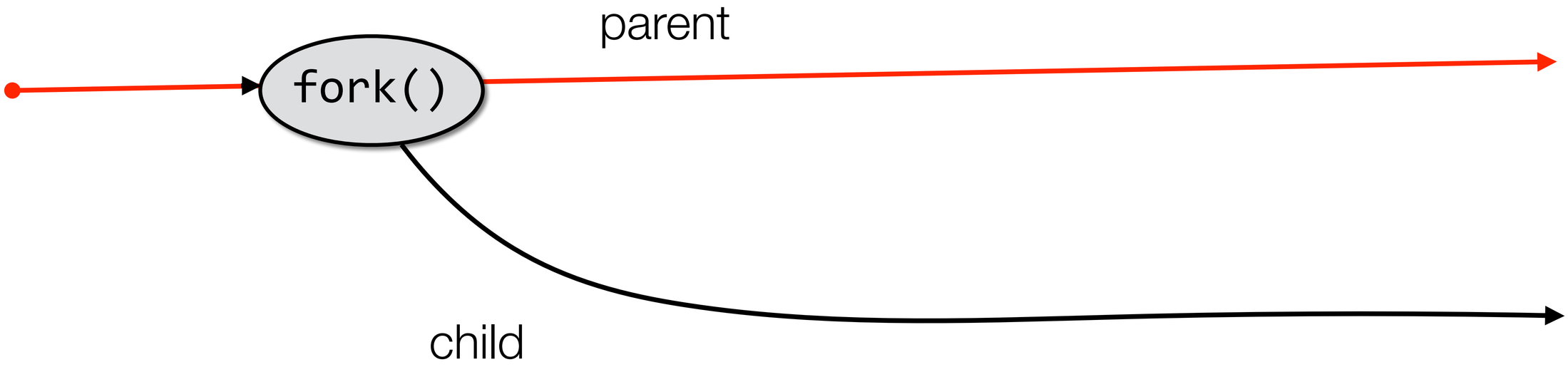
- After creating the process the Kernel can do one of the following, as part of the dispatcher routine:
  - Stay in the parent process.
  - Transfer control to the child process
  - Transfer control to another process.

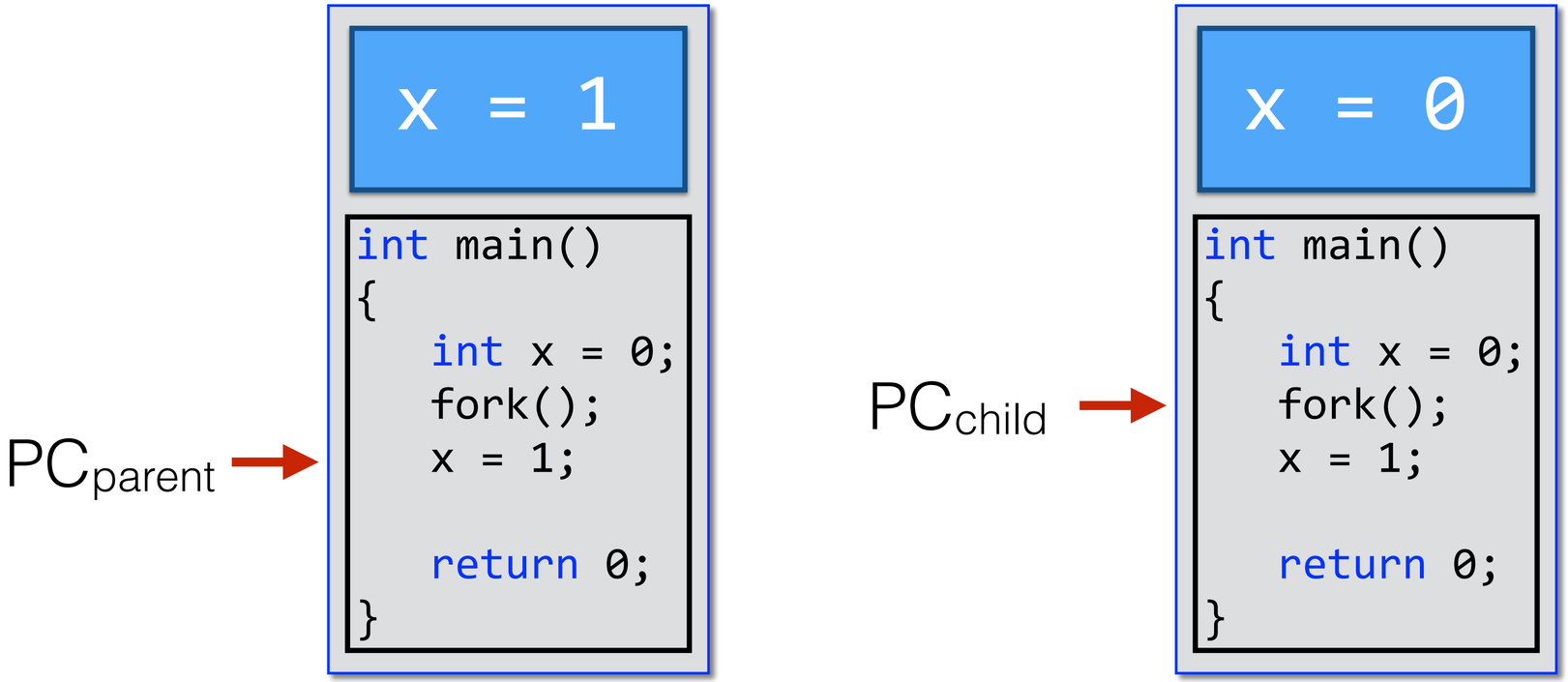


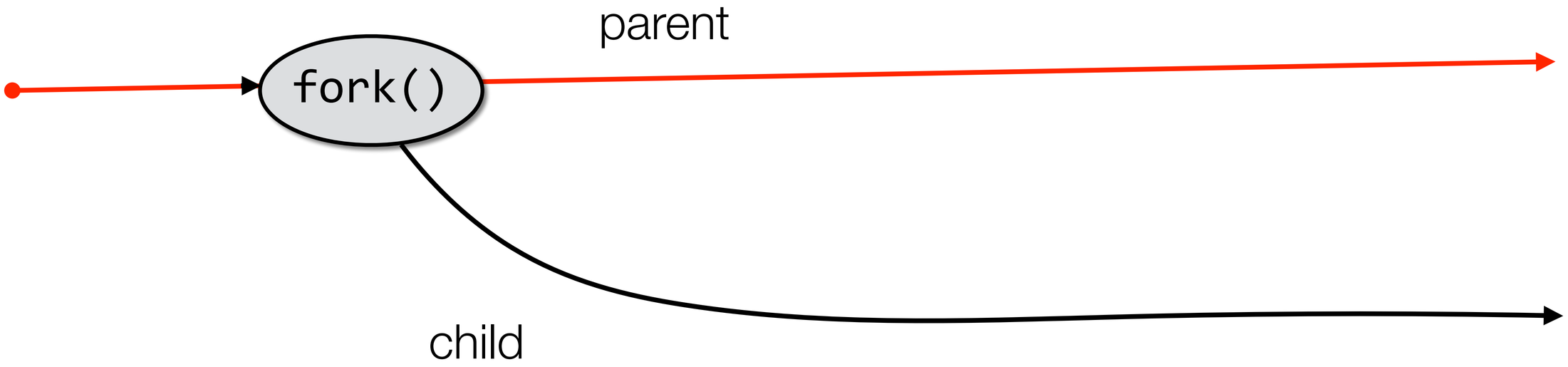


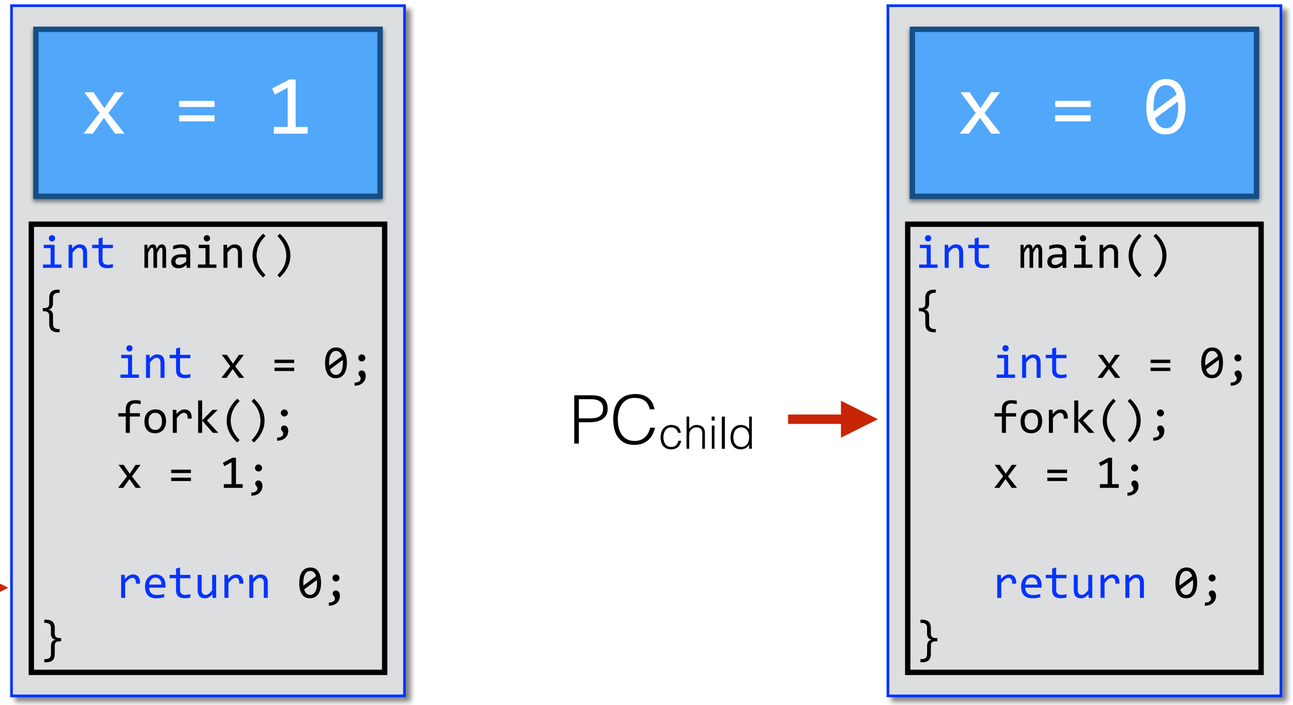
The **`fork()`** call is issued. A new process is created (child).

Assuming that the parent gets to run, what happens to it?

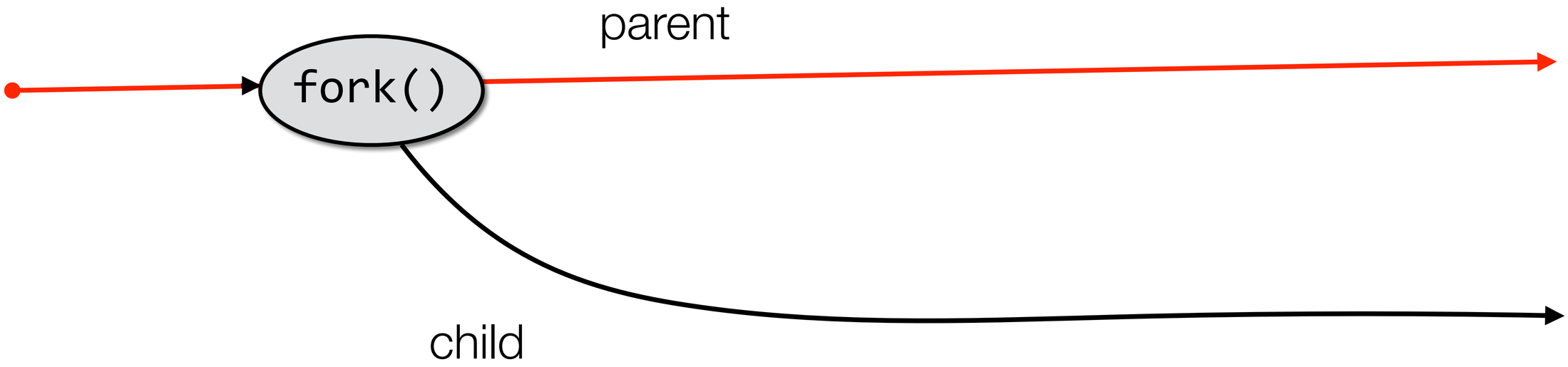

parent
fork()
child

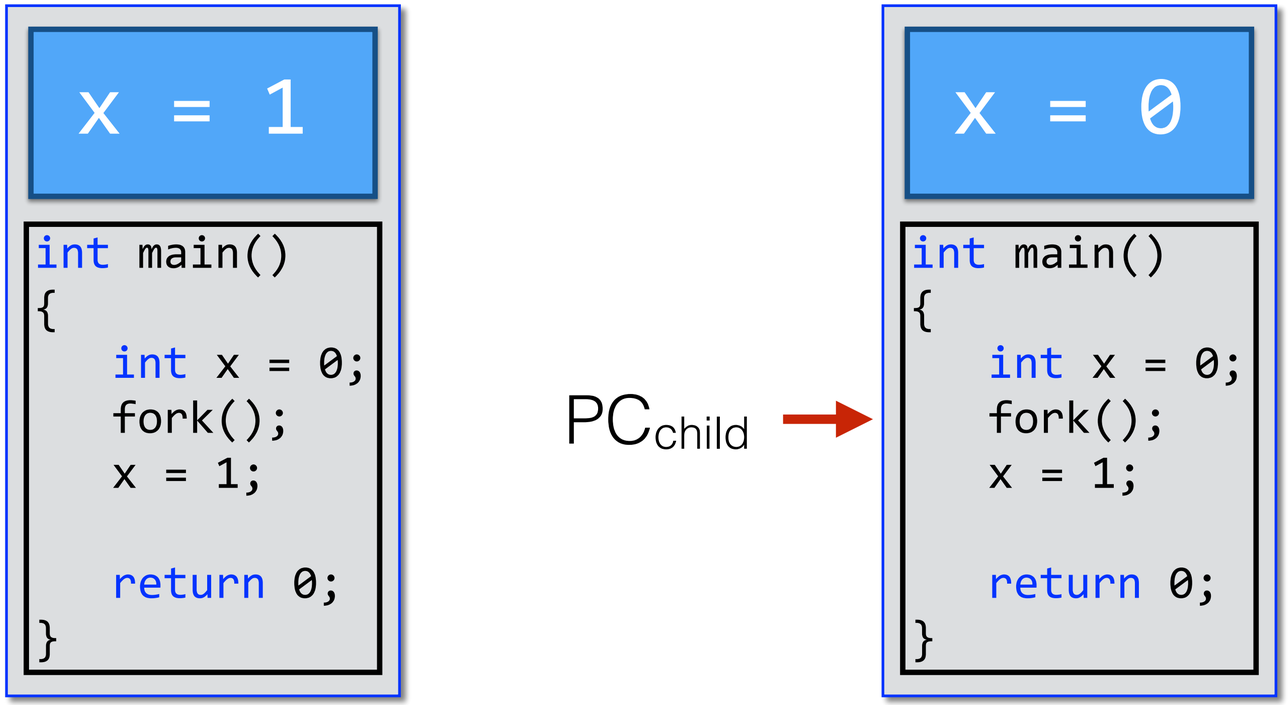

x = 1
int main()
{
int x = 0;
fork();
x = 1;
return 0;
}
PCparent
PCchild
x = 0
int main()
{
int x = 0;
fork();
x = 1;
return 0;
}



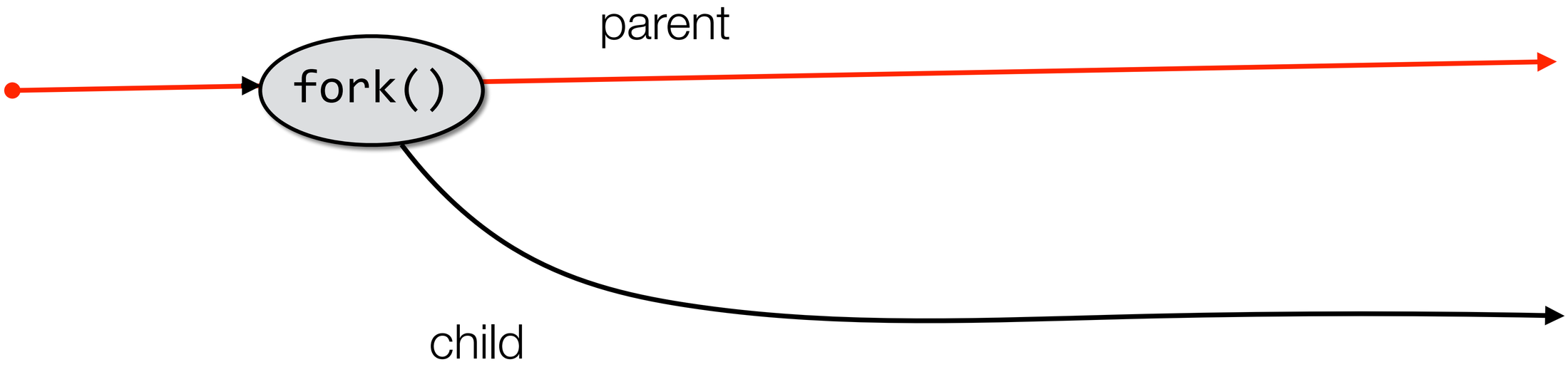


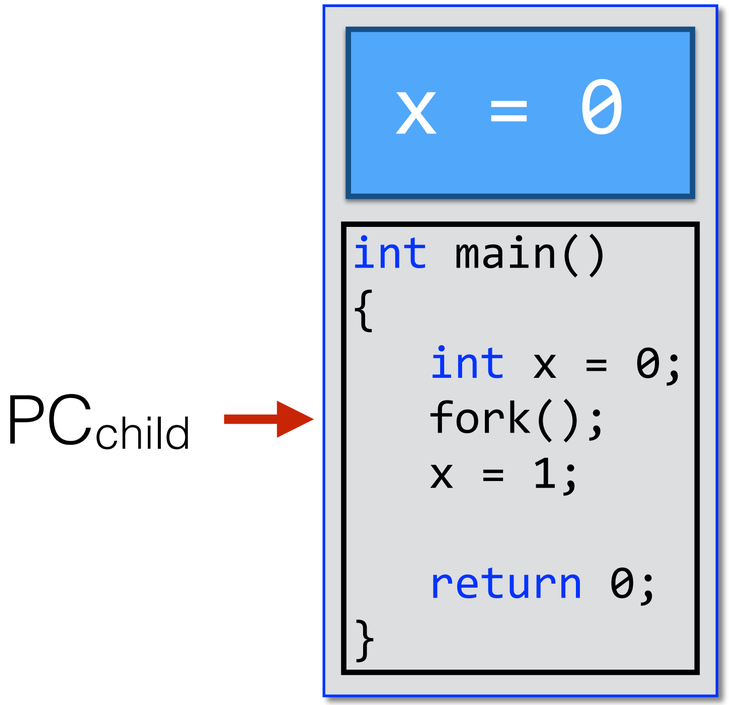
Why didn't the value of **x** in the child process change?





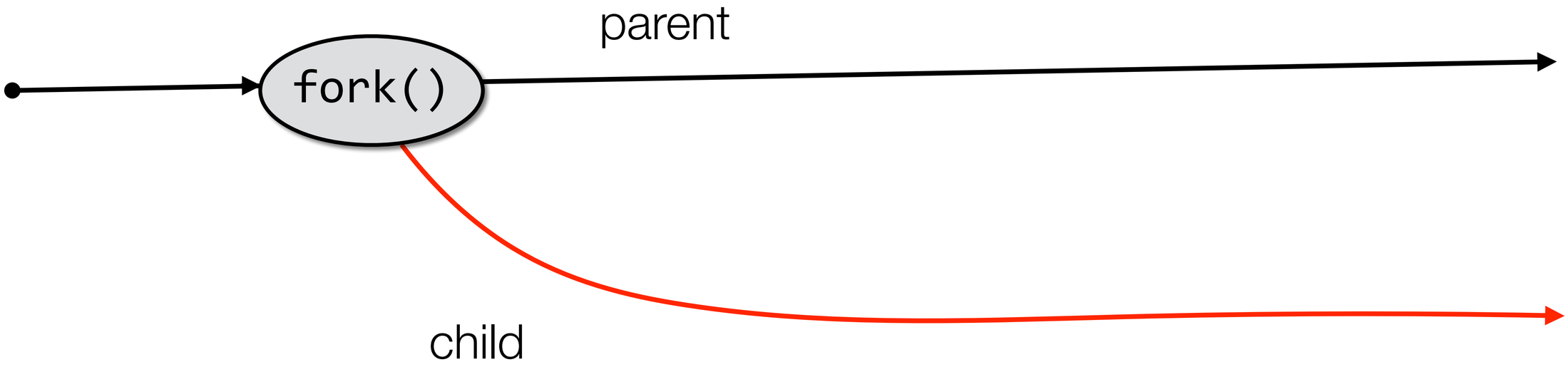
The parent process terminates. What happens to the child?

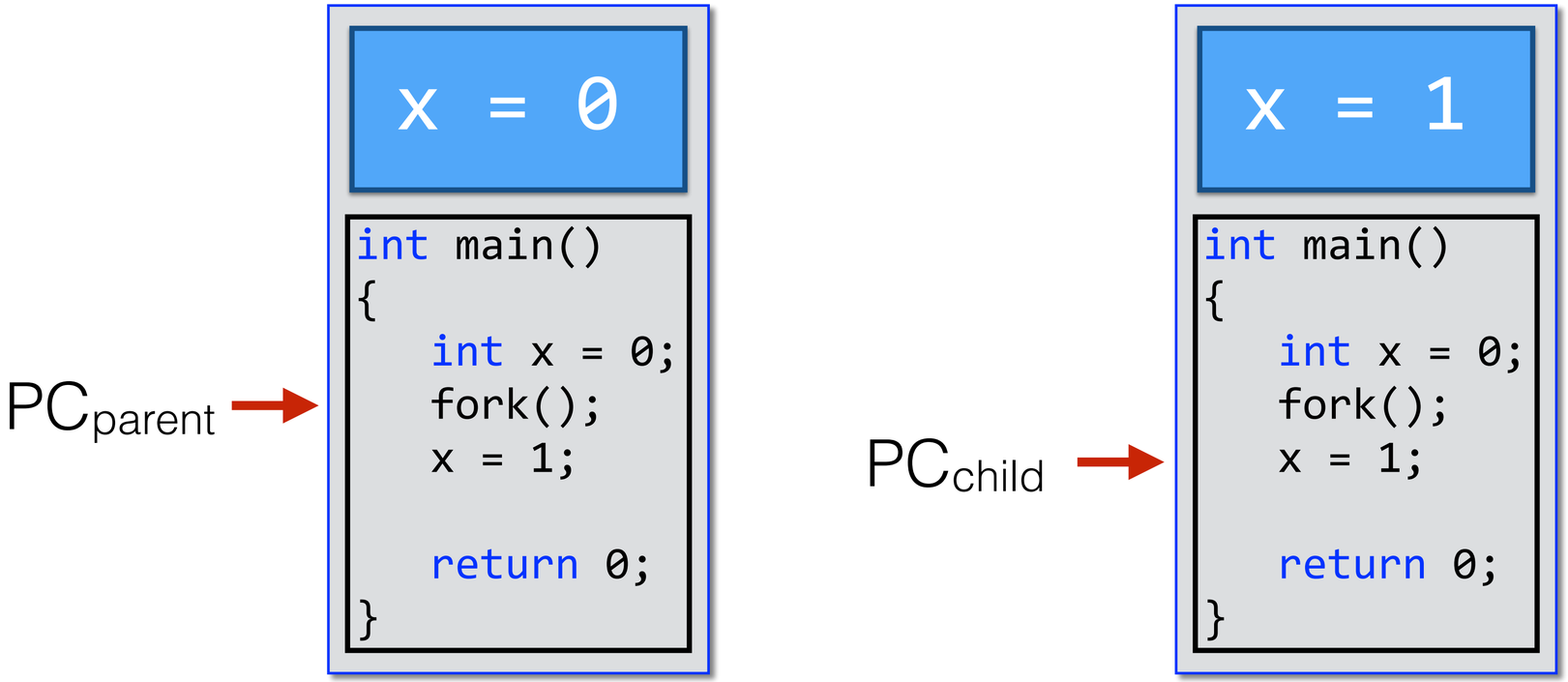




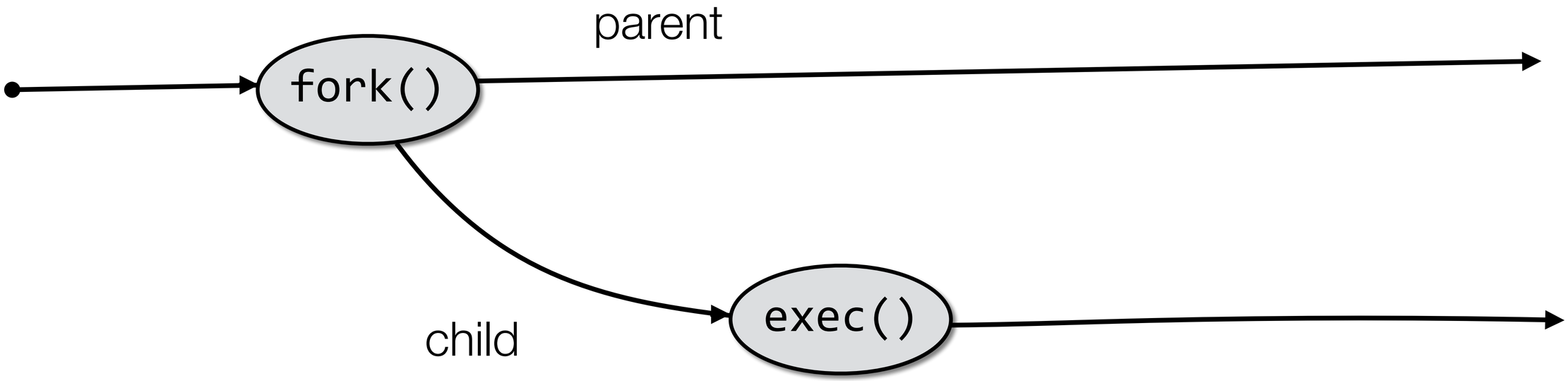
The parent process terminates. What happens to the child?

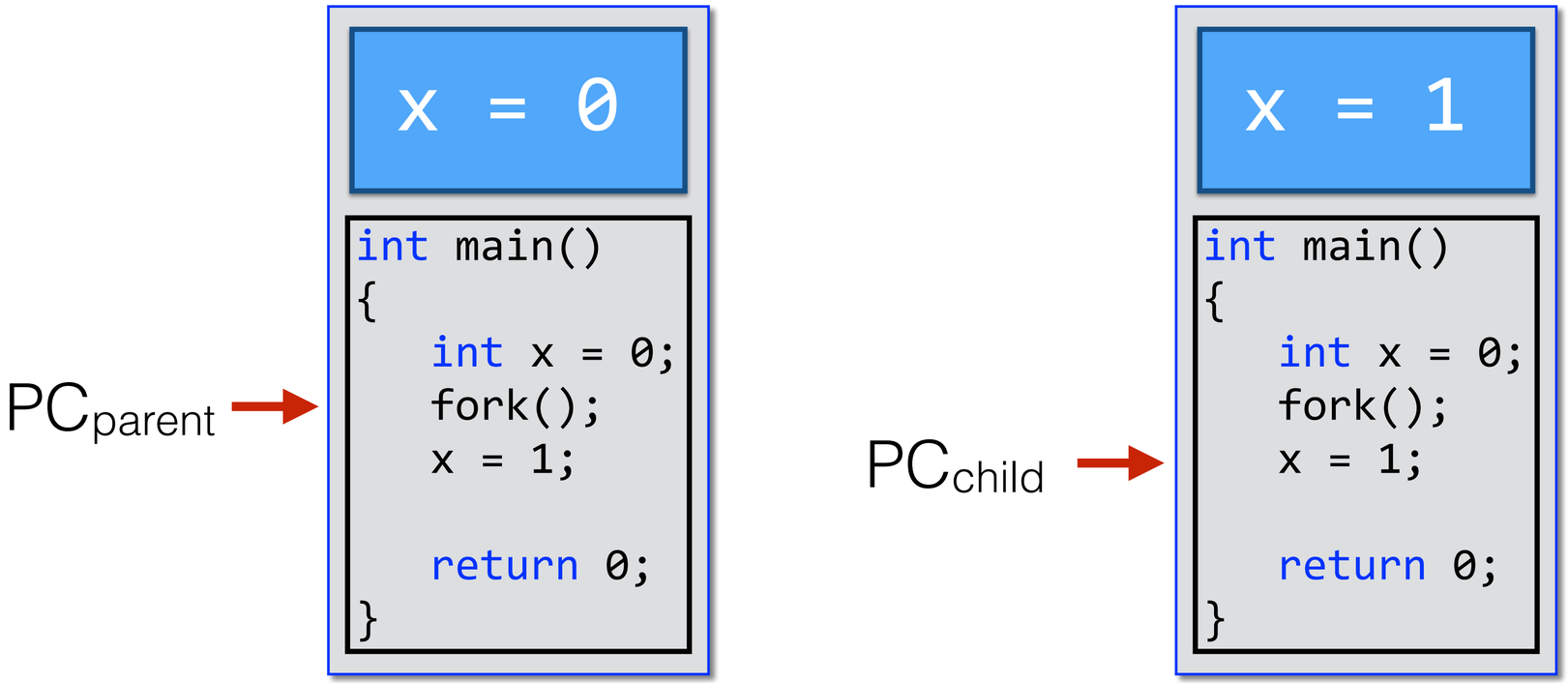
Who is the parent of the child now?



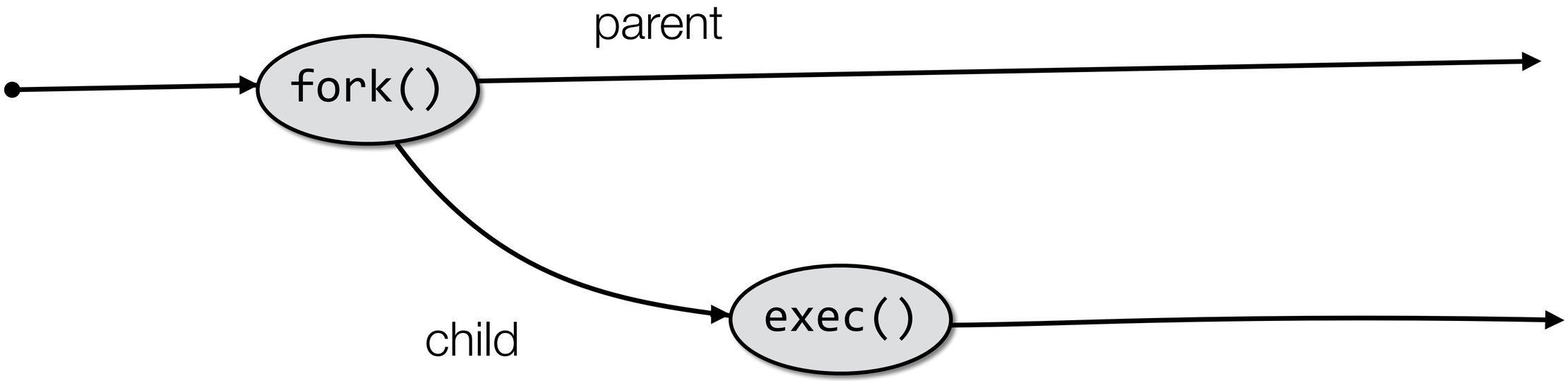


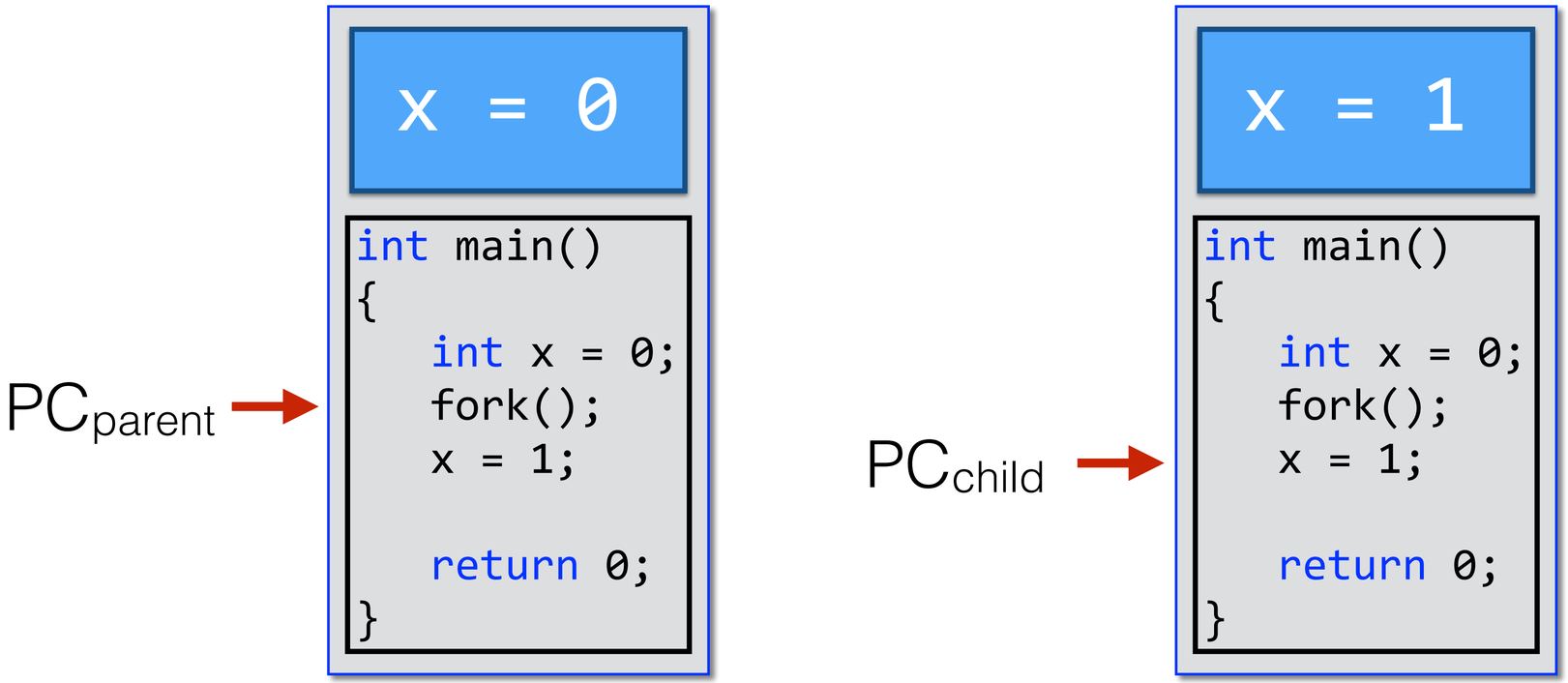
If the child process is scheduled to run, it executes just like the parent, i.e., it starts from the next instruction after the **`fork()`**. Then, it eventually terminates.



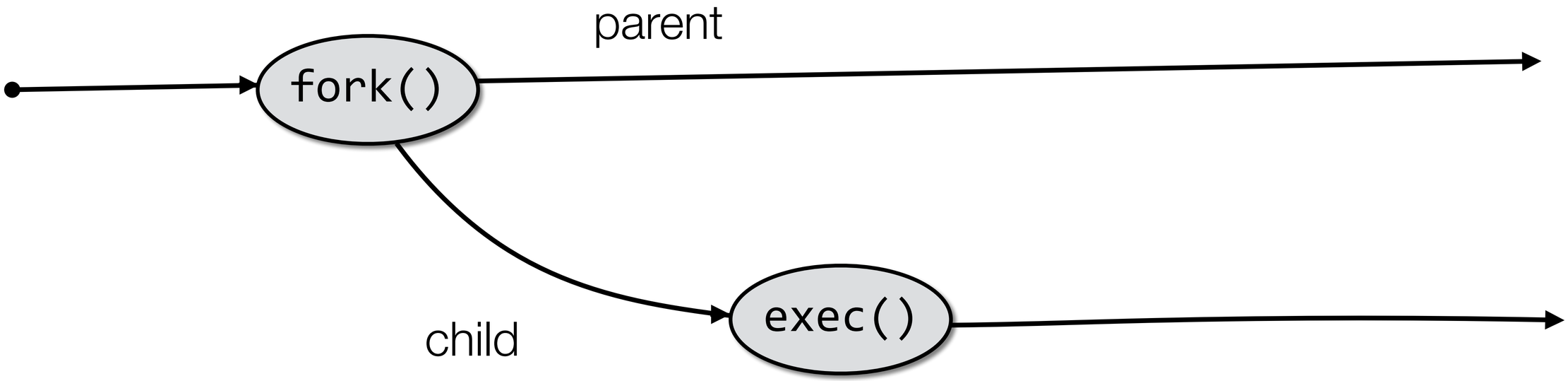


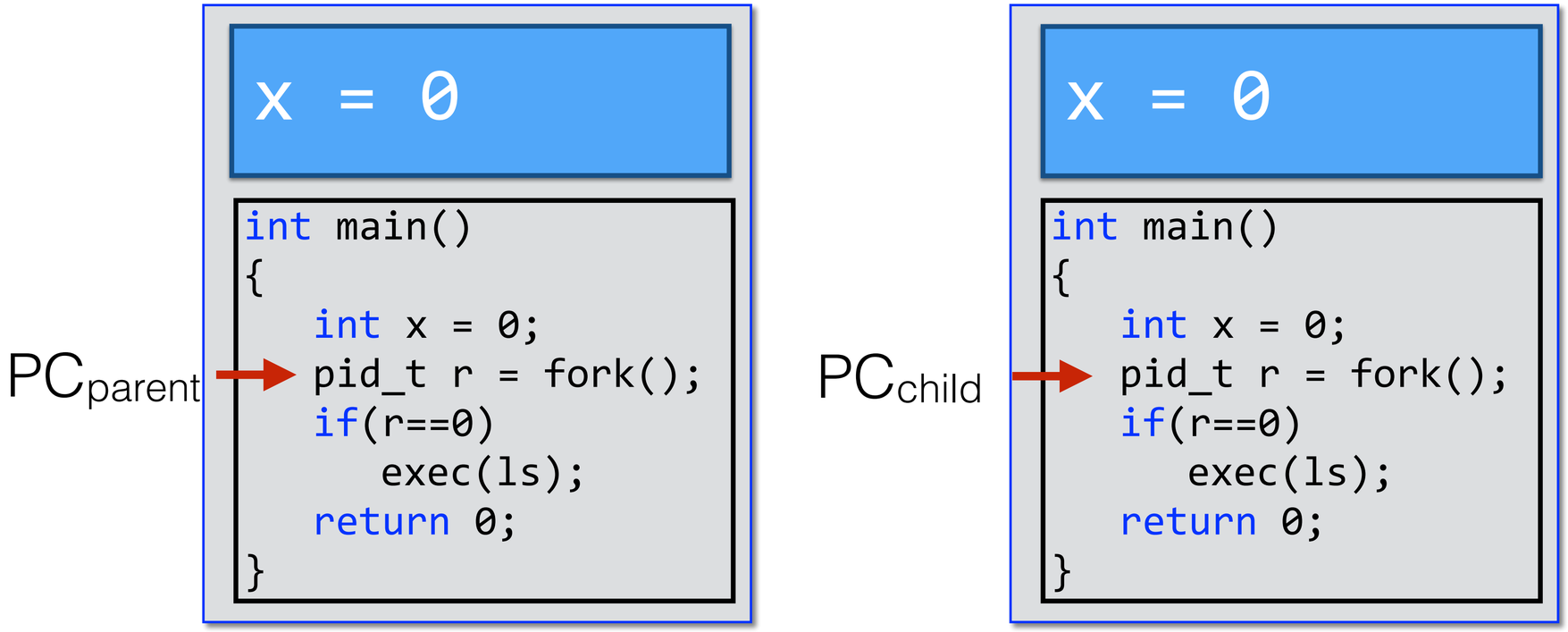
If we want the child process to execute code that is different from the parent's, we can call **`exec()`** in the child process.





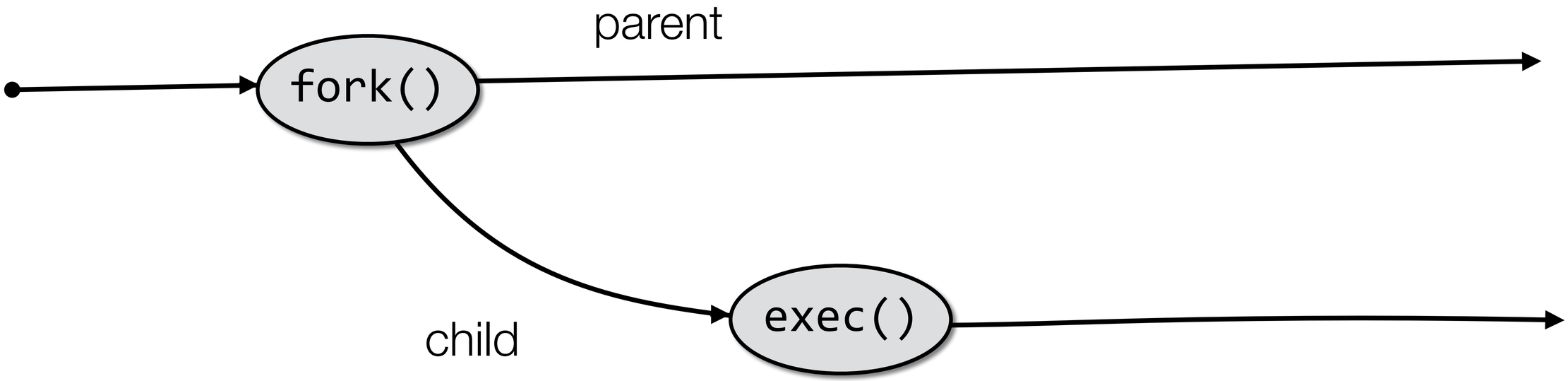
If we want the child process to execute code that is different from the parent's, we can call `exec()` in the child process. **But how can we tell parent from child?**





But how can we tell parent from child?

`fork()` returns `0` to the child process and a `non-zero` value to parent process.



PC$_{parent}$

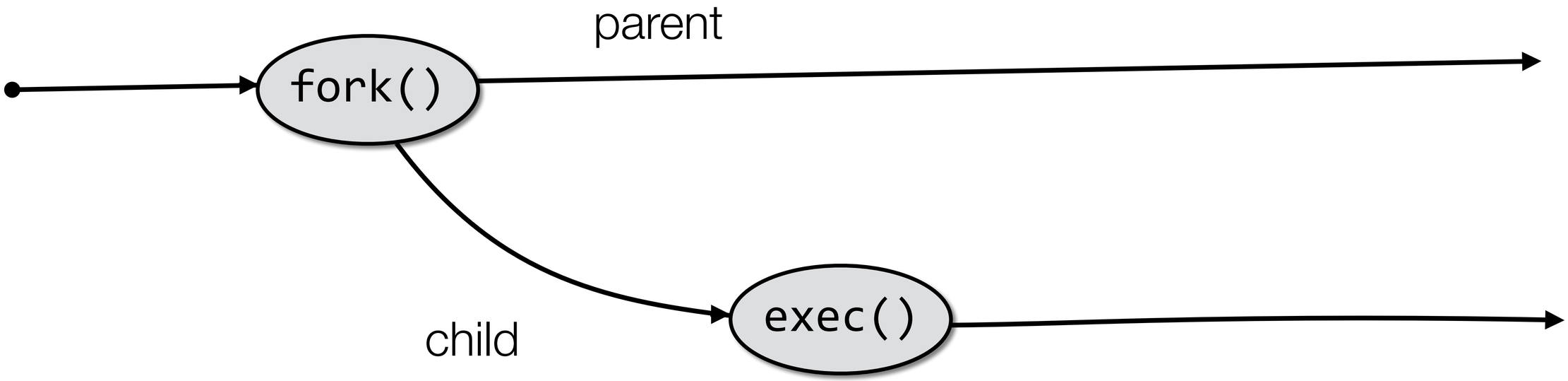
```
x = 0, r=10
int main()
{
    int x = 0;
    pid_t r = fork();
    if(r==0)
        exec(ls);
    return 0;
}
```

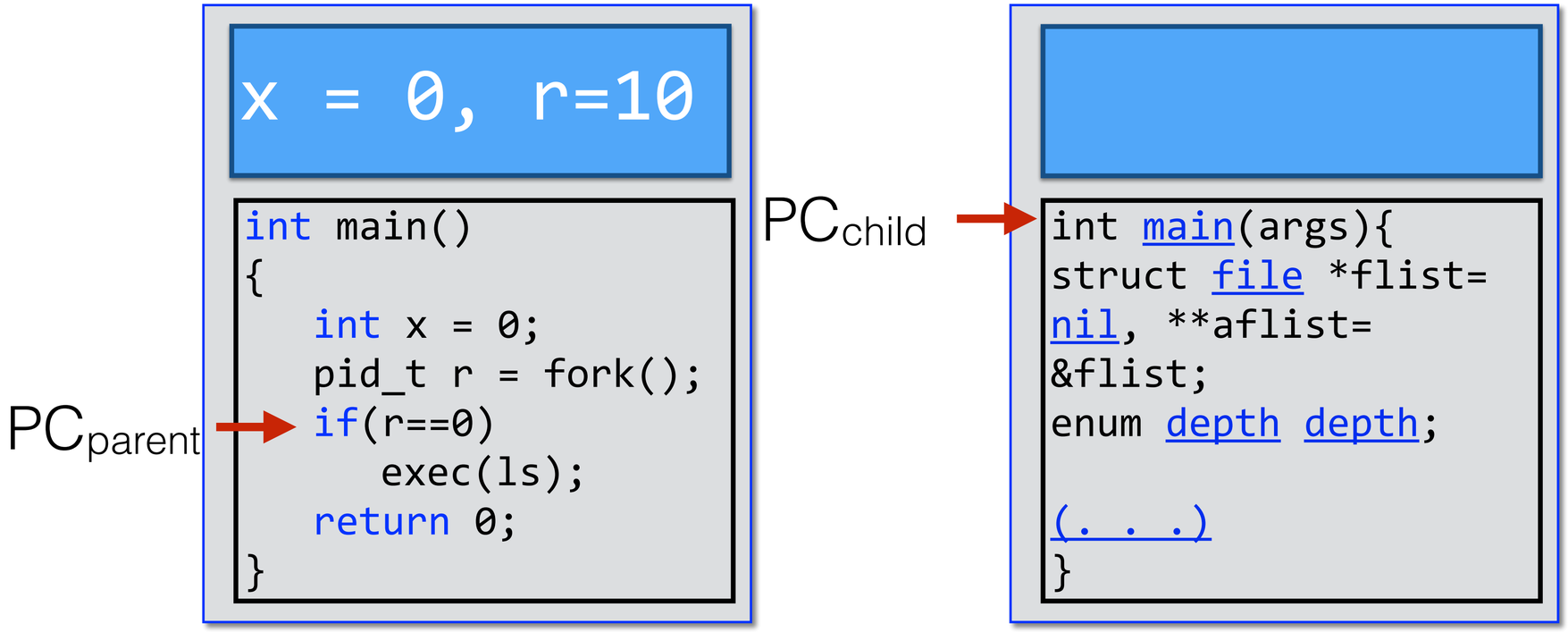
PC$_{child}$

```
x = 0, r=0
int main()
{
    int x = 0;
    pid_t r = fork();
    if(r==0)
        exec(ls);
    return 0;
}
```

But how can we tell parent from child?

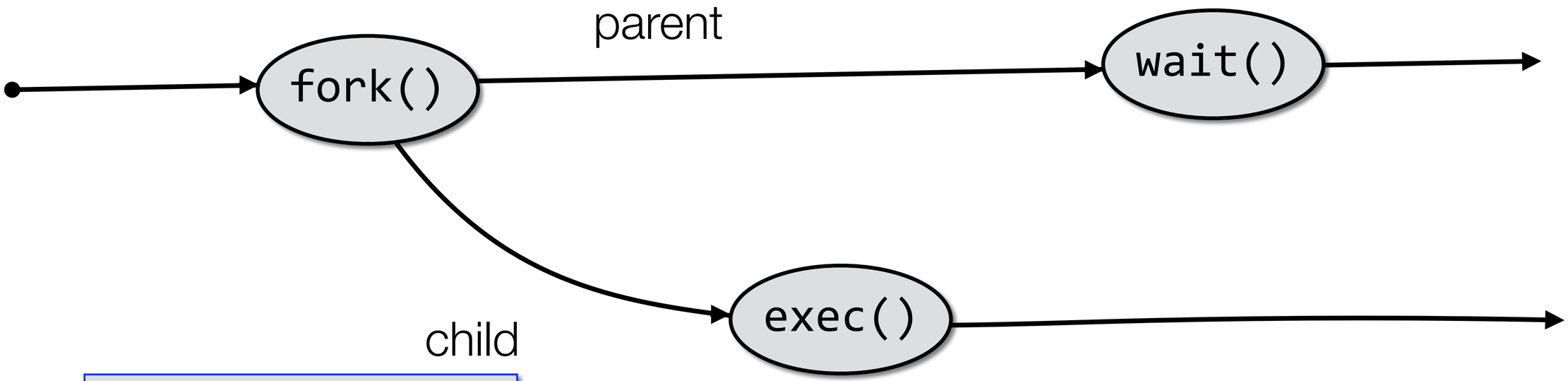
To the parent, `fork()` returns the child's `PID` or a negative number (i.e., child couldn't be created).





The `exec()` function replaces the code section of the child process with the code of the new program.

The PC is reset to the first instruction.



x = 0, r=10

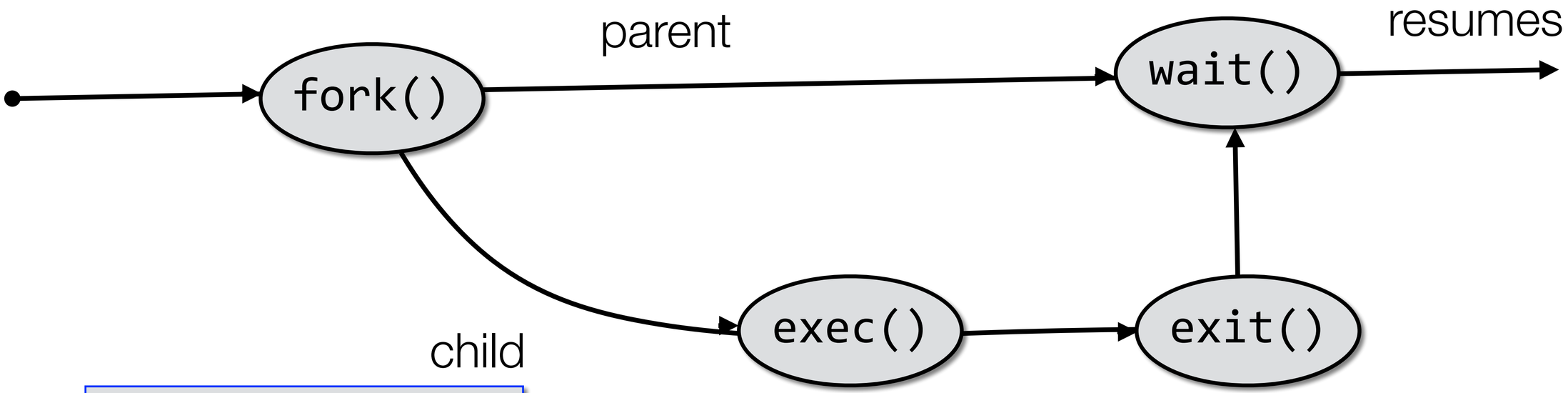
```
int main()
{
    int x = 0;
    pid_t r = fork();
    if(r==0)
        exec(ls);
    else
        wait()
    return 0;
}
```

PC$_{parent}$

PC$_{child}$

```
int main(args){
struct file *flist=
nil, **aflist=
&flist;
enum depth depth;

(. . .)
}
```

Parent can issue a **`wait()`**. This will make parent wait until the child process terminates.



child

x = 0, r=10

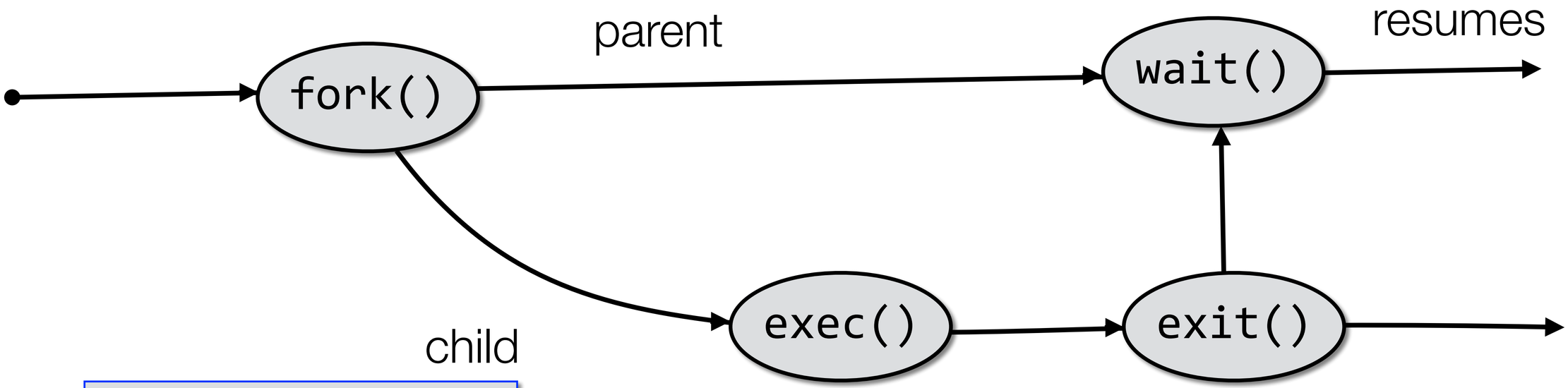
```
int main()
{
    int x = 0;
    pid_t r = fork();
    if(r==0)
        exec(ls);
    else
        wait()
    return 0;
}
```

PC$_{parent}$

PC$_{child}$

```
int main(args){
struct file *flist=
nil, **aflist=
&flist;
enum depth depth;

(. . .)
}
```

Parent can issue a **`wait()`**. This will make parent wait until the child process terminates.



child

x = 0, r=10

```
int main()
{
    int x = 0;
    pid_t r = fork();
    if(r==0)
        exec(ls);
    else
        wait()
    return 0;
}
```

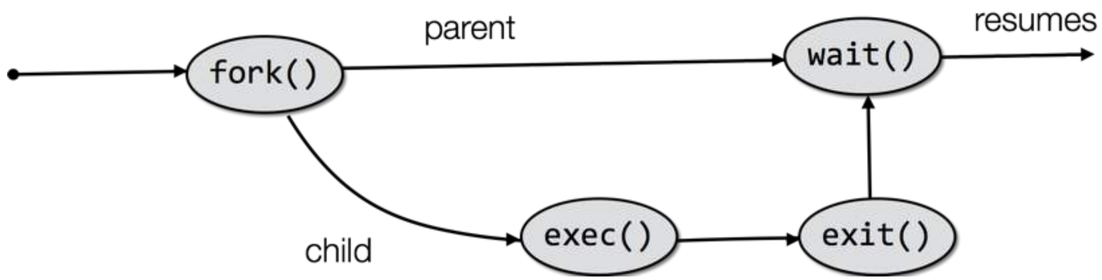
PC$_{parent}$

PC$_{child}$

```
int main(args){
struct file *flist=
nil, **aflist=
&flist;
enum depth depth;

(. . .)
}
```

What are the advantages of this apparently complex interface?

Standard fork pattern




```
int main()
{
pid_t  pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
          fprintf(stderr, "Fork Failed");
          exit(-1);
    }
    else if (pid == 0) { /* child process */
          execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
          /* parent will wait for the child to
    complete */
          wait (NULL);
          printf ("Child Complete");
          exit(0);
    }
}
```

Process Creation in Unix

- Process creation is by means of the system call fork( ).
- This causes the OS, in Kernel Mode, to:
  1. Allocate a slot in the process table for the new process.
  2. Assign a unique process ID to the child process.
  3. Copy of process image of the parent, with the exception of any shared memory.
  4. Increment the counters for any files owned by the parent, to reflect that an additional process now also owns those files.
  5. Assign the child process to the Ready to Run state.
  6. Returns the ID number of the child to the parent process, and a 0 value to the child process.

Why have fork() at all?

Why make a copy of the parent process?

Don't you usually want to start a new program instead?

Where might “cloning” the parent be useful?

- Web server – make a copy for each incoming connection
- Parallel processing – set up initial state, fork off multiple copies to do work

UNIX philosophy: System calls should be minimal.

- Don't overload system calls with extra functionality if it is not always needed.
- Better to provide a flexible set of simple primitives and let programmers combine them in useful ways.

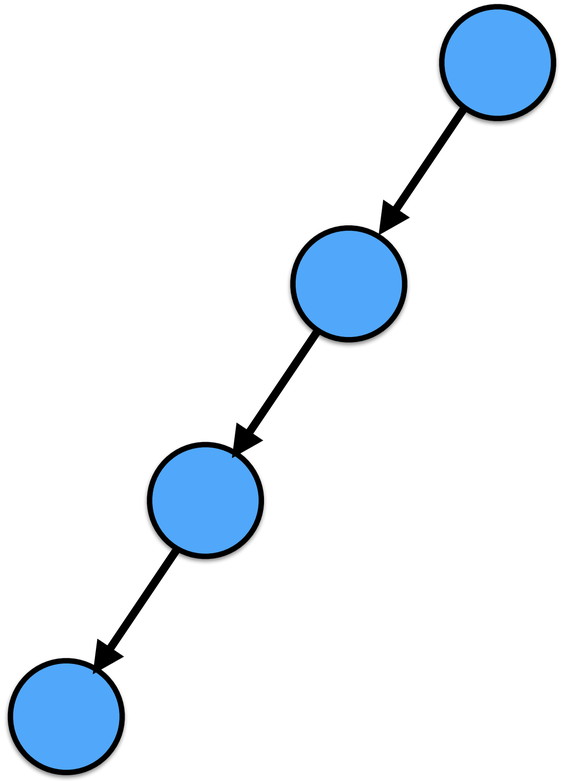
Output of sample program

```
Process 4530: value is 0
Process 4530: value is 1
Process 4530: value is 2
Process 4530: value is 3
Process 4530: value is 4
Process 4530: value is 5
Process 4530: About to do a fork...
Process 4531: value is 6
Process 4530: value is 6
Process 4530: value is 7
Process 4531: value is 7
Process 4530: value is 8
Process 4531: value is 8
Process 4530: value is 9
Process 4531: value is 9
```

What determines the order in which the two processes run???

queue of processes

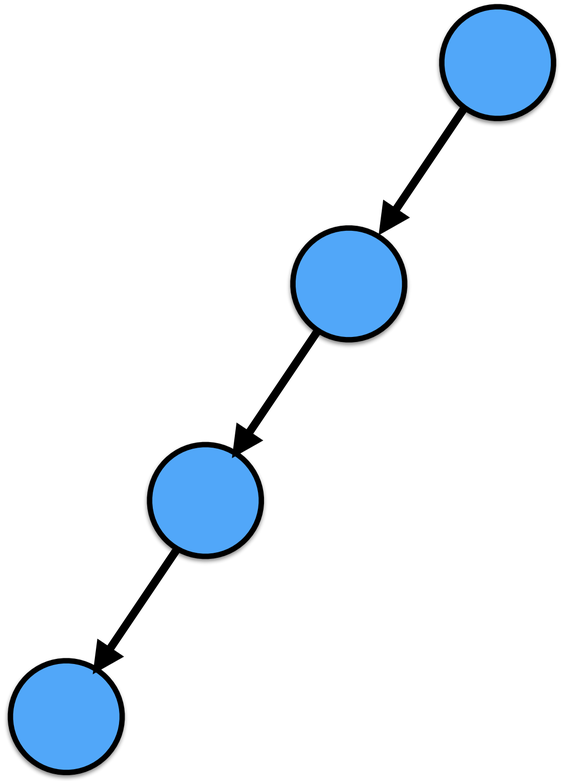
```
int i, n = 4;
pid_t pid;
for ( i = 1; i < n; ++i )
    if ( pid = fork() )
        break;
```



Parent breaks because when the child process is created, `fork()` returns the PID of the child to the parent process. The PID of any process is nonzero.

queue of processes

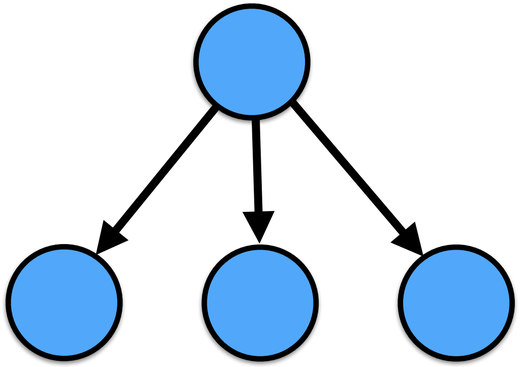
```
int i, n = 4;
pid_t pid;
for ( i = 1; i < n; ++i )
    if ( pid = fork() )
        break;
```



Child processes run the for-loop because `fork()` returns 0 (zero) to the child process.

fan of processes

```
int i, n = 4;
pid_t pid;
for ( i = 1; i < n; ++i )
    if ( ( pid = fork() ) <= 0 )
        break;
```



This time, child processes break, and parent runs the loop.

exit()

- When calling **exit()**, a process voluntarily release all its resources, e.g., address space returned, files closed, etc.
- But not everything can be cleaned by the process itself, it has to be cleaned by someone else.
- Also, I **should not** be all cleaned by the process because the parent process may be waiting for the return value.
- So, there is a separate state for the period after the process calls exit() and before someone else cleans it up.
- In Unix, it's called **zombie** state.

http://www.cs.toronto.edu/~ylzhang/csc369f15/files/lec01-intro-processes.pdf

Zombie

- When a process exits, almost all of its resources are deallocated
- Address space is returned, files are closed, etc.
- PCB retains information about the process's exit state
- The process retains its PID
- The process is a **zombie** until its parent cleans it up

http://www.cs.toronto.edu/~ylzhang/csc369f15/files/lec01-intro-processes.pdf