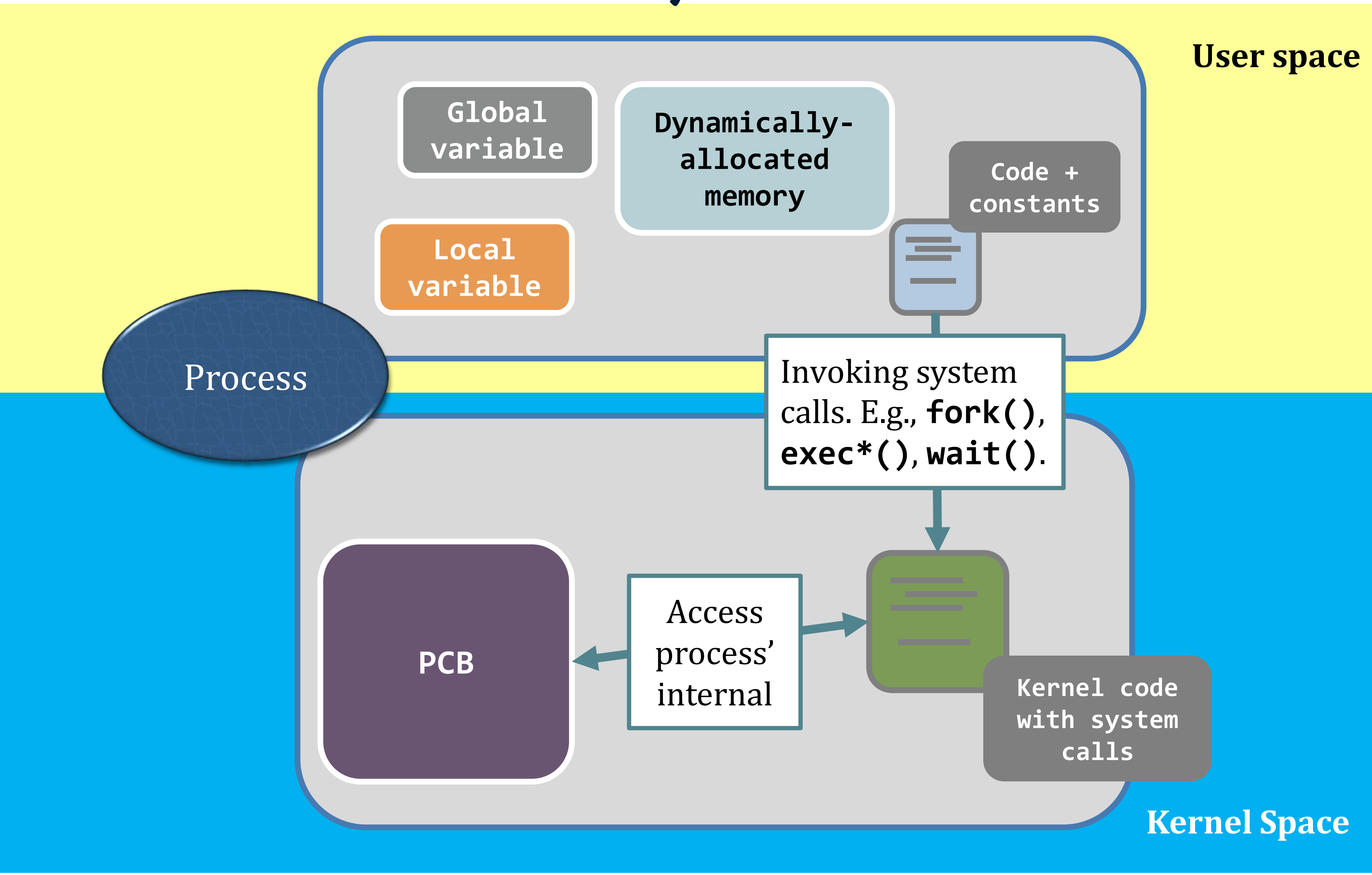
Lecture 4: Process II

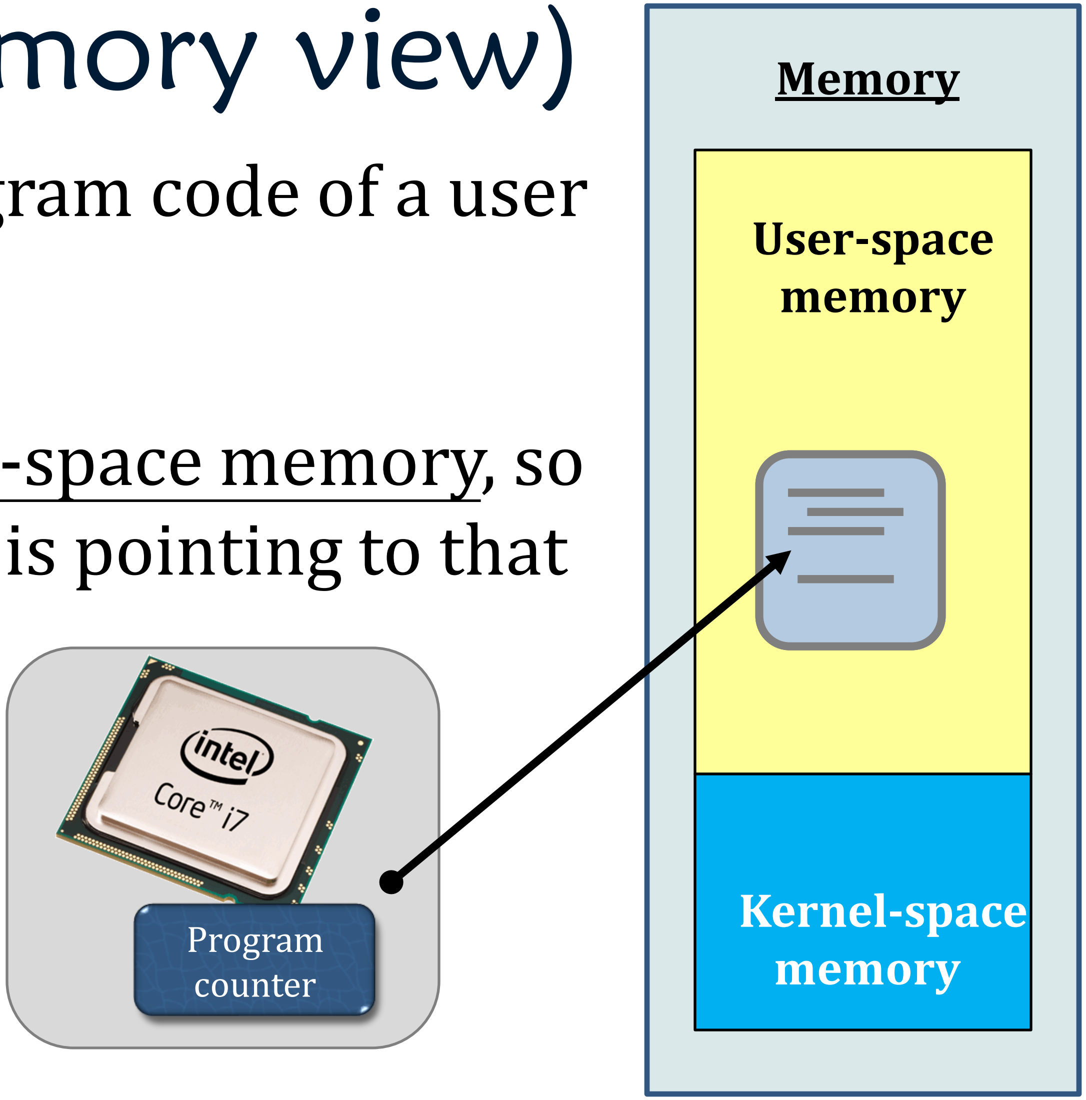
Bo Tang @ 2020, Spring

The story so far…



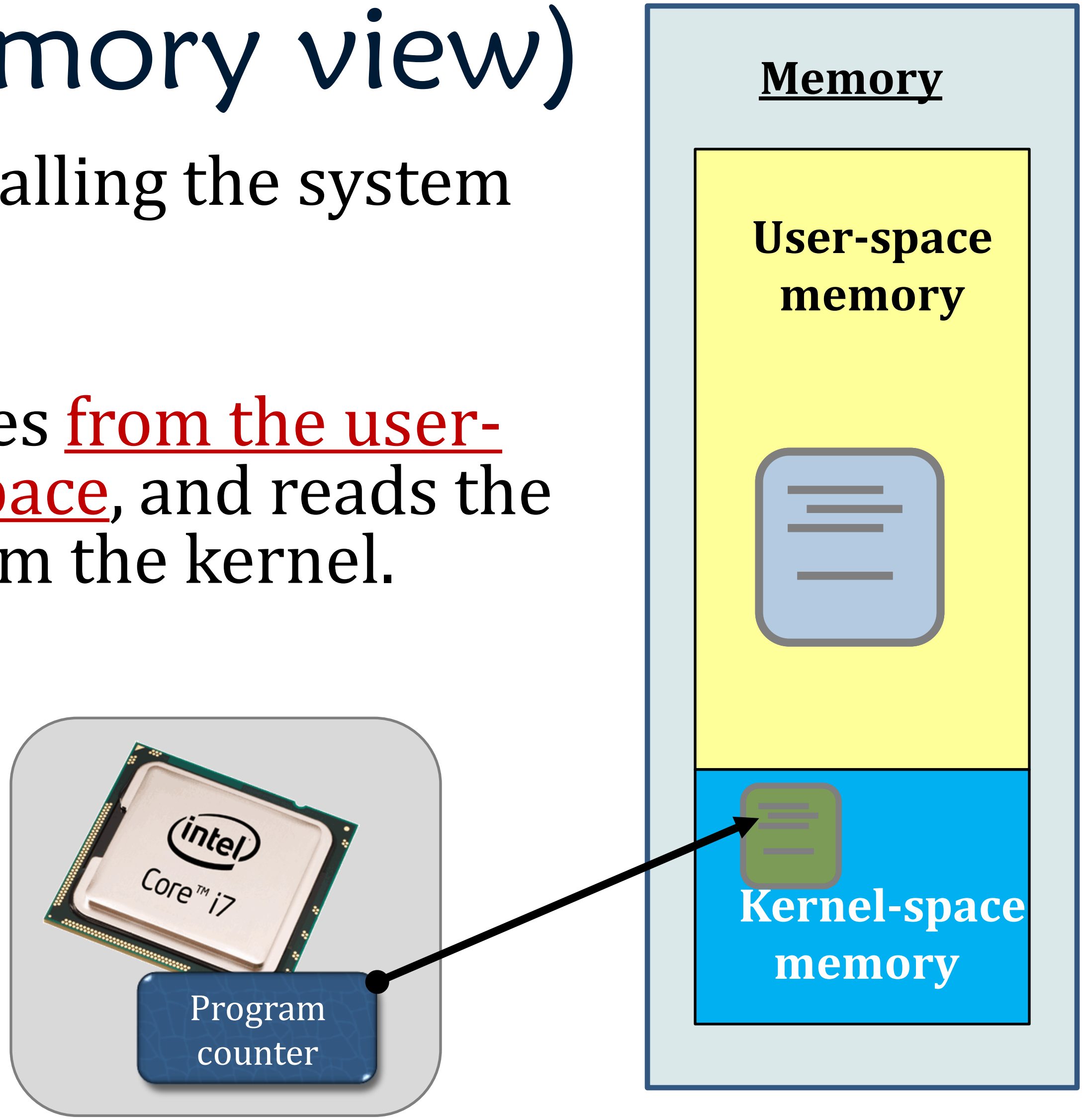
When invoking a system call (memory view)

- When running a program code of a user process.
- As the code is in user-space memory, so the program counter is pointing to that region.



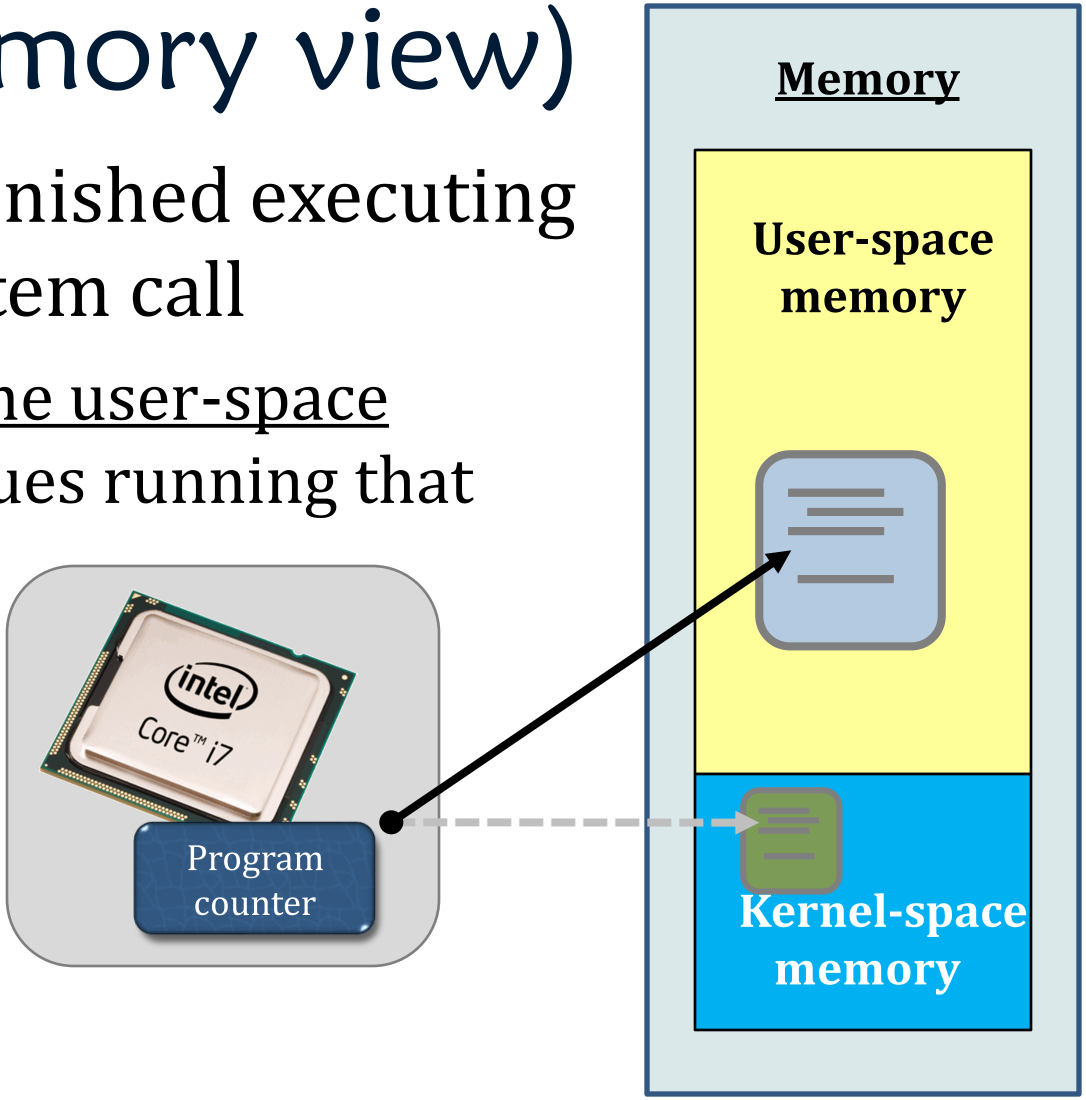
When invoking a system call (memory view)

- When the process is calling the system call "**`getpid()`**".
- Then, the CPU switches from the user-space to the kernel-space, and reads the PID of the process from the kernel.

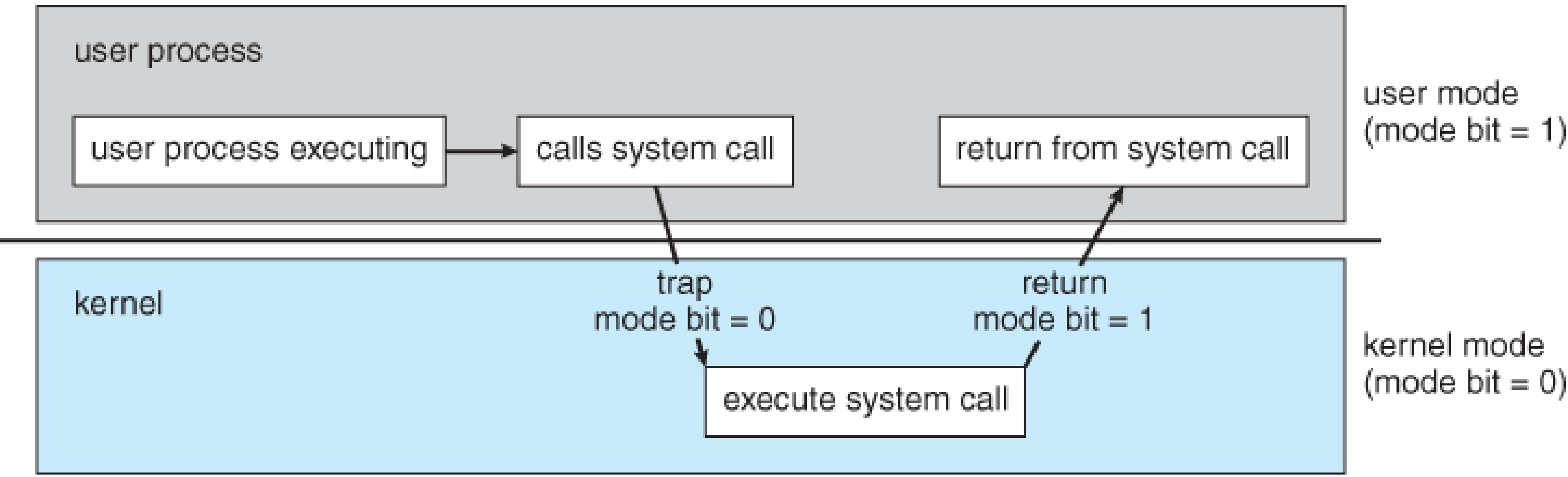


When invoking a system call (memory view)

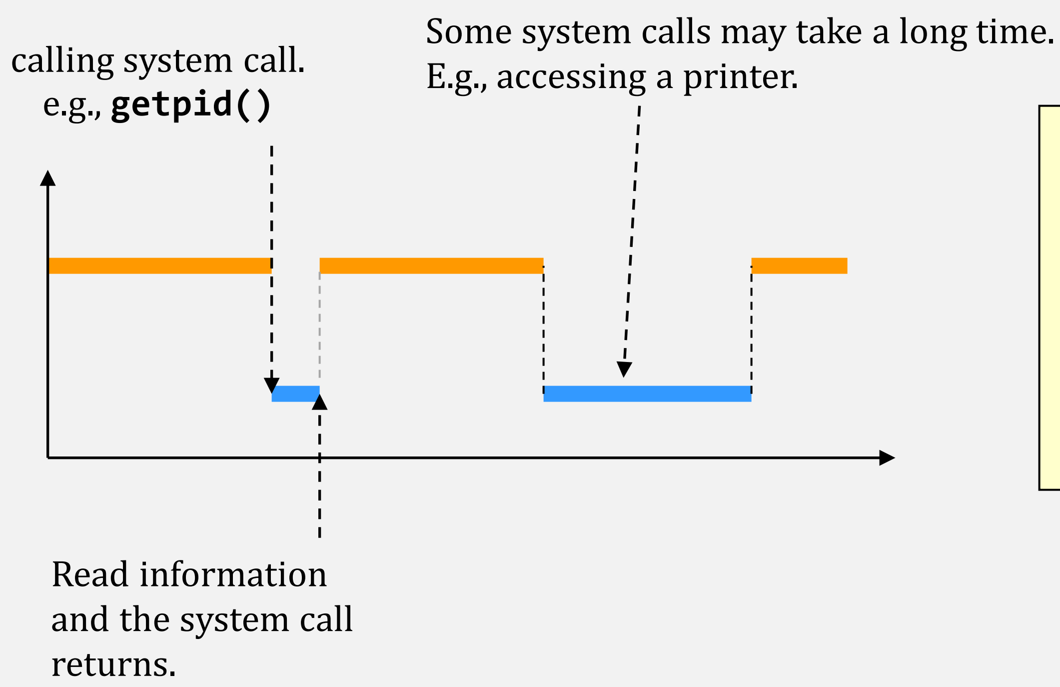
- When the CPU has finished executing the "**`getpid()`**" system call
  - it <u>switches back to the user-space memory</u>, and continues running that program code.



When invoking a system call (CPU view)



Process real time cost (wall-clock time)



User time –
CPU time spent on codes in user-space memory.

Sys time –
CPU time spent on codes in kernel-space memory.

User time VS System time – example 1

- Let's tell the difference…with the tool "**`time`**".

```
$ time ./time_example

real     0m0.001s
user     0m0.000s
sys      0m0.000s
$ _
```

Real-time elapsed when "**`./time_example`**" terminates.

The **user time** of "**./time_example**".

The **sys time** of "**./time_example**".

It's possible:
real > user + sys
real < user + sys

Why?

```
int main(void) {
    int x = 0;
    for(i = 1; i <= 10000; i++) {
        x = x + i;
    //  printf("x = %d\n", x);
    }
    return 0;
}
```

User time VS System time – example 1

- Let's tell the difference...with the tool "**time**".

```
$ time ./time_example

real      0m0.001s
user      0m0.000s
sys       0m0.000s
$ _
```

```
int main(void) {
    int x = 0;
    for(i = 1; i <= 10000; i++) {
        x = x + i;
//      printf("x = %d\n", x);
    }
    return 0;
}
```

Commented on purpose.

```
$ time ./time_example

real 0m2.795s
user 0m0.084s
sys  0m0.124s
$ _
```

See? Accessing hardware costs the process more time.

```
int main(void) {
    int x = 0;
    for(i = 1; i <= 10000; i++) {
        x = x + i;
        printf("x = %d\n", x);
    }
    return 0;
}
```

Comment released.

User time VS Sys time – example 2

- The user time and the sys time together **define the performance of an application**.
  - When writing a program, you must consider both the user time and the sys time.
    - E.g., the output of the following two programs are exactly the same. But, their running time is not.

```
#define MAX 1000000

int main(void) {
    int i;
    for(i = 0; i < MAX; i++)
        printf("x\n");
    return 0;
}
```

```
#define MAX 1000000

int main(void) {
    int i;
    for(i = 0; i < MAX / 5 ; i++)
        printf("x\nx\nx\nx\nx\n");
    return 0;
}
```

User time VS Sys time – example 2

```
#define MAX 1000000

int main(void) {
    int i;
    for(i = 0; i < MAX; i++)
        printf("x\n");
    return 0;
}
```

```
$ time ./time_example_slow

real 0m1.562s
user 0m0.024s
sys  0m0.108s
$ _
```

```
#define MAX 1000000

int main(void) {
    int i;
    for(i = 0; i < MAX / 5 ; i++)
        printf("x\nx\nx\nx\nx\n");
    return 0;
}
```

```
$ time ./time_example_fast

real 0m1.293s
user 0m0.012s
sys  0m0.084s
$ _
```

User time VS Sys time

- Function calls cause overhead
  - Stack pushing (will see later)
- Sys calls may cause even more

➔Sys call is from another "process" (the kernel)

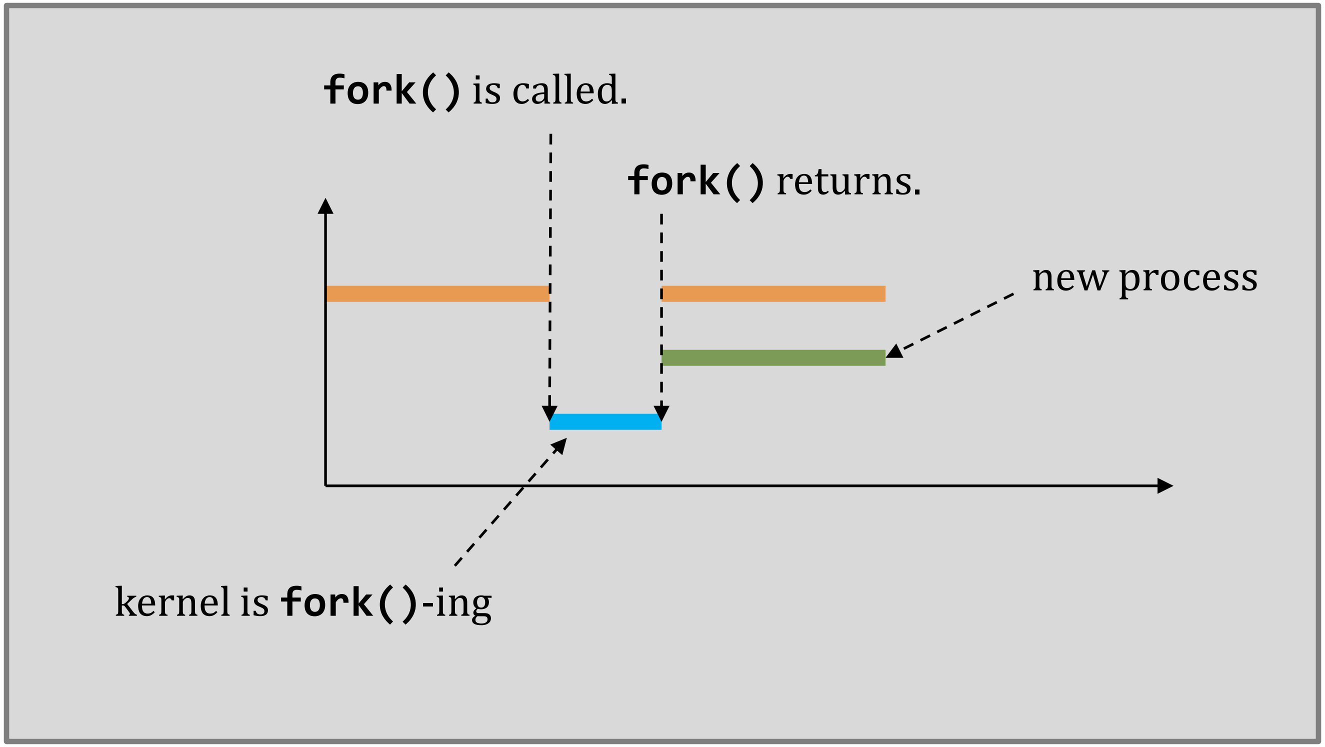
➔Switching to another "process" ➔ context switch (will see later)

https://www.quora.com/Is-an-OS-kernel-itself-a-process

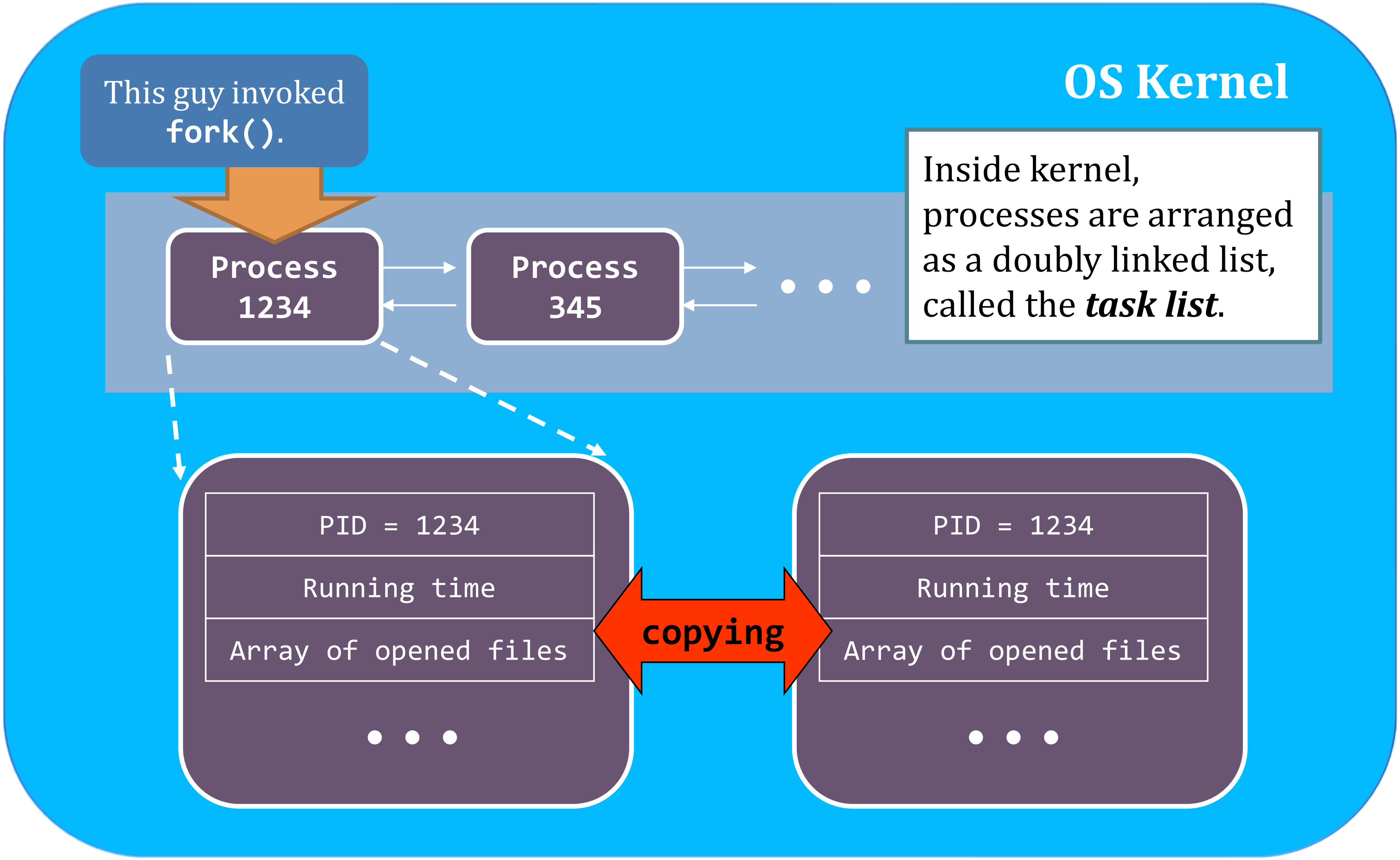
Working of system calls

- `fork();`

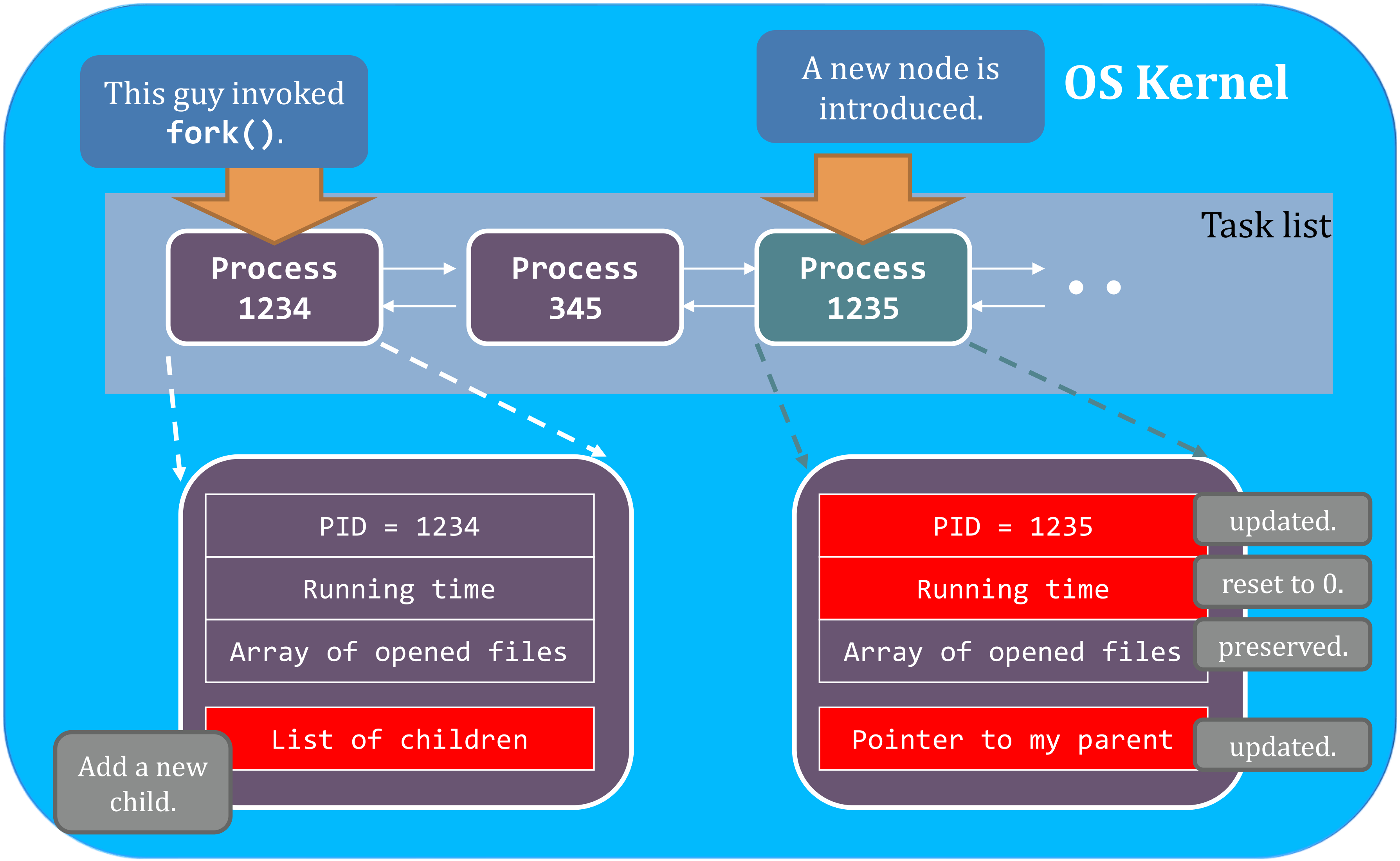
Programmer view of fork()



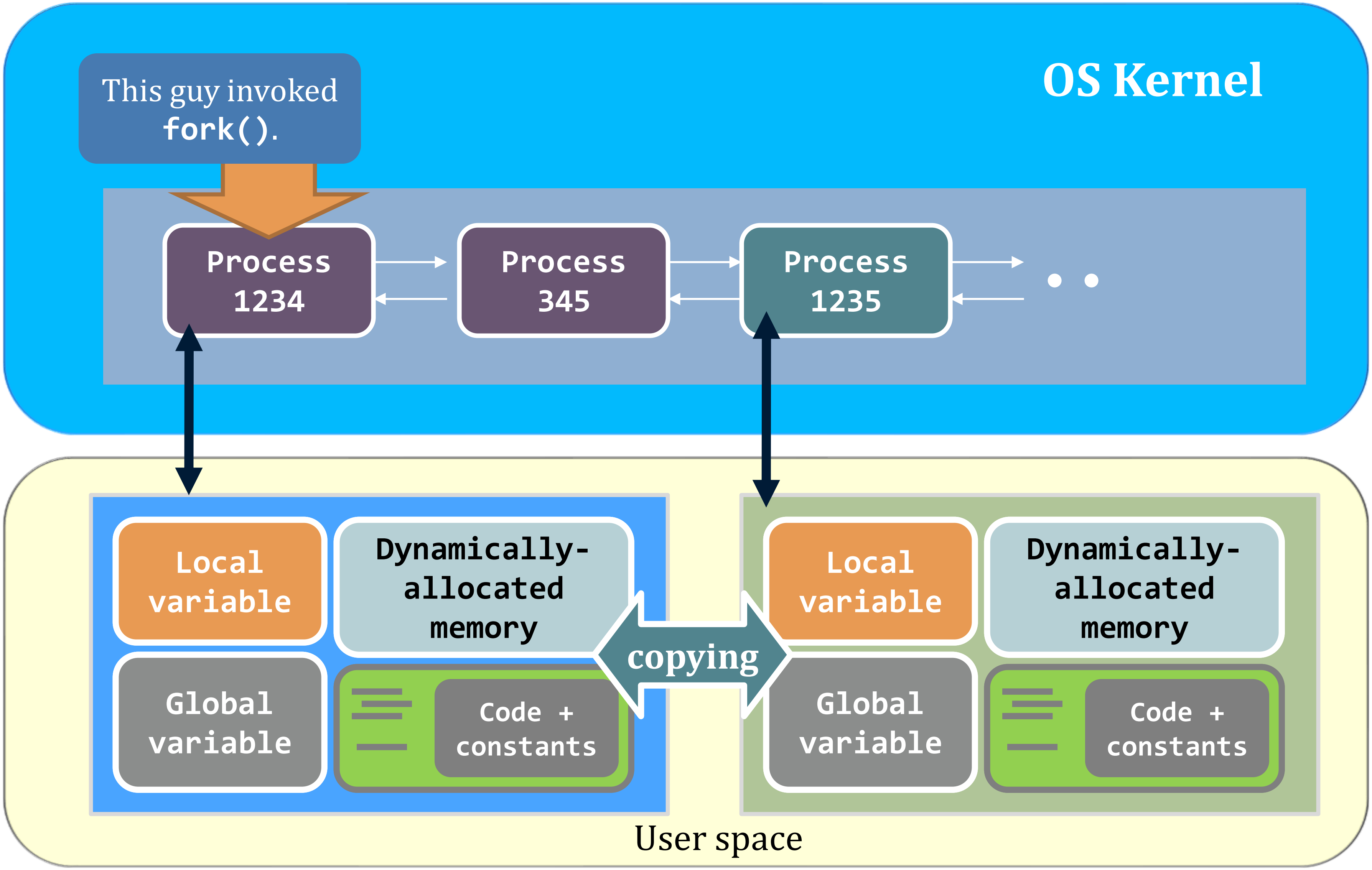
fork() inside the kernel



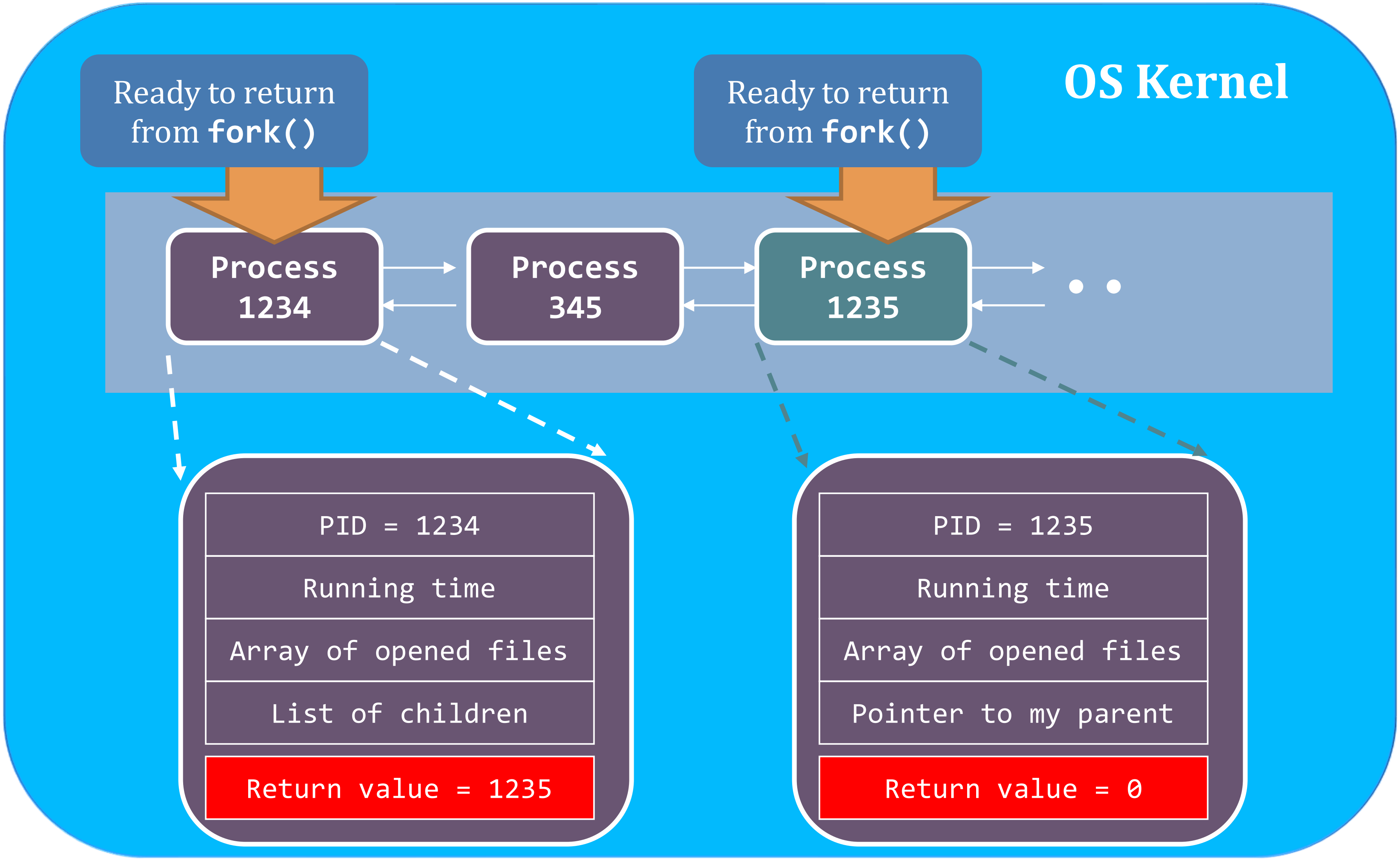
fork() in action – kernel-space update



fork() in action – user-space update



fork() in action – finish



`fork()` in action – array of opened files?

- Array of opened files contains:

Array Index	Description
0	Standard Input Stream; **`FILE *stdin;`**
1	Standard Output Stream; **`FILE *stdout;`**
2	Standard Error Stream; **`FILE *stderr;`**
3 or beyond	Storing the files you opened, e.g., **`fopen()`**, **`open()`**, etc.

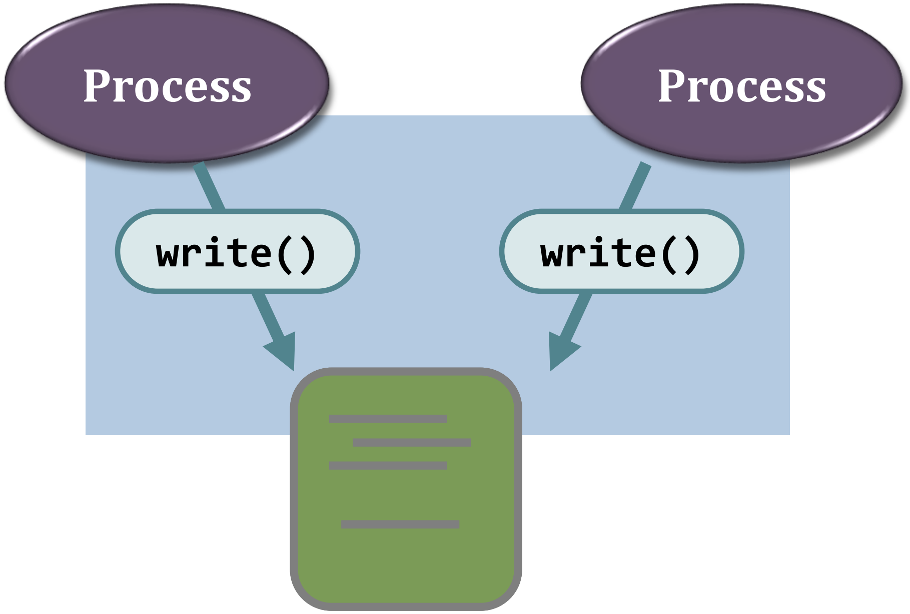
- That's why a parent process **shares the same terminal output stream** as the child process.

Stream is just a **logical** object for you to read as a **sequence of bytes**

So, how can you random access the middle of a file? Read Stream → Array → Array[mid-point]

fork() in action – sharing opened files?

- What if two processes, **sharing the same opened file**, write to that file together?



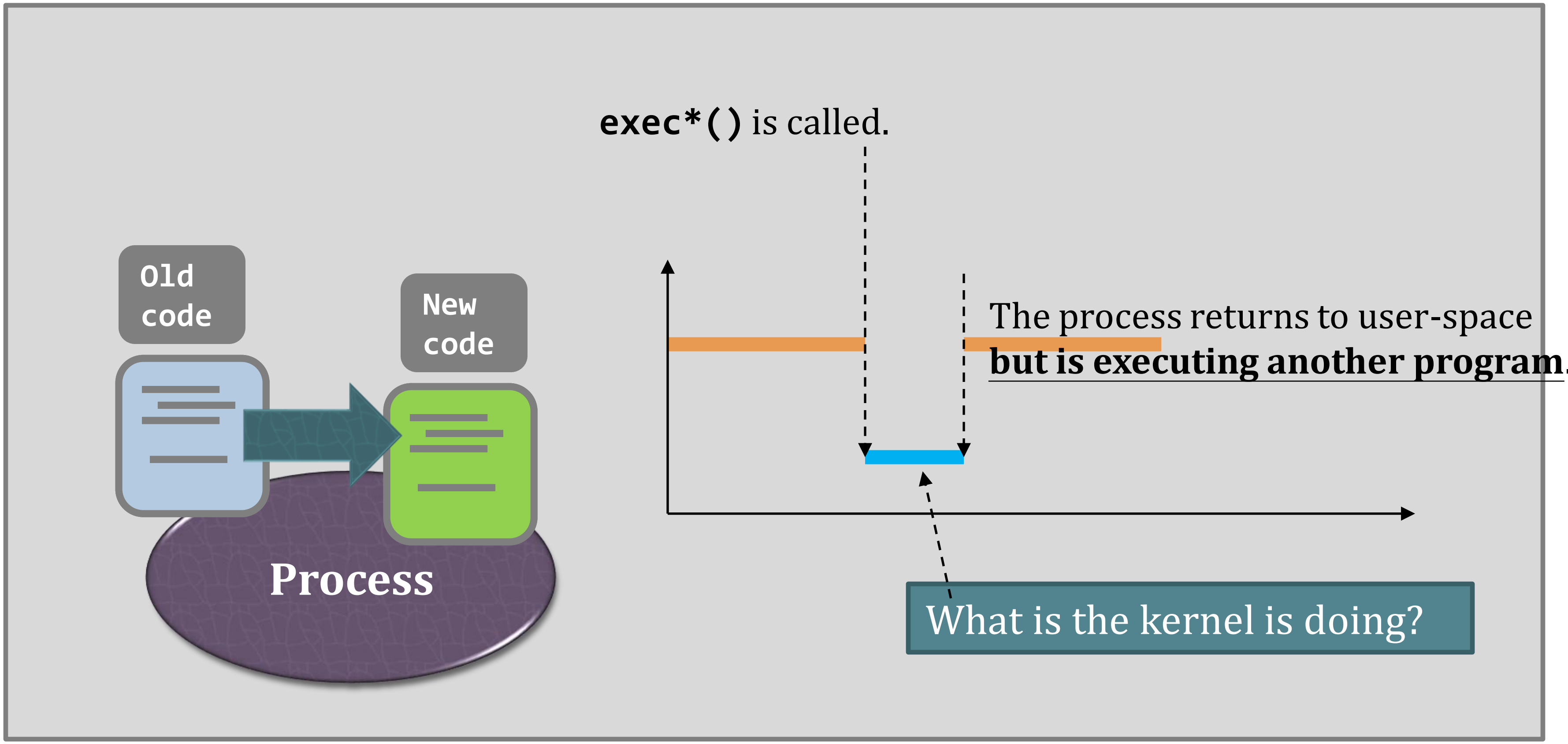
Let's see what will happen when the program finishes running!

Working of system calls

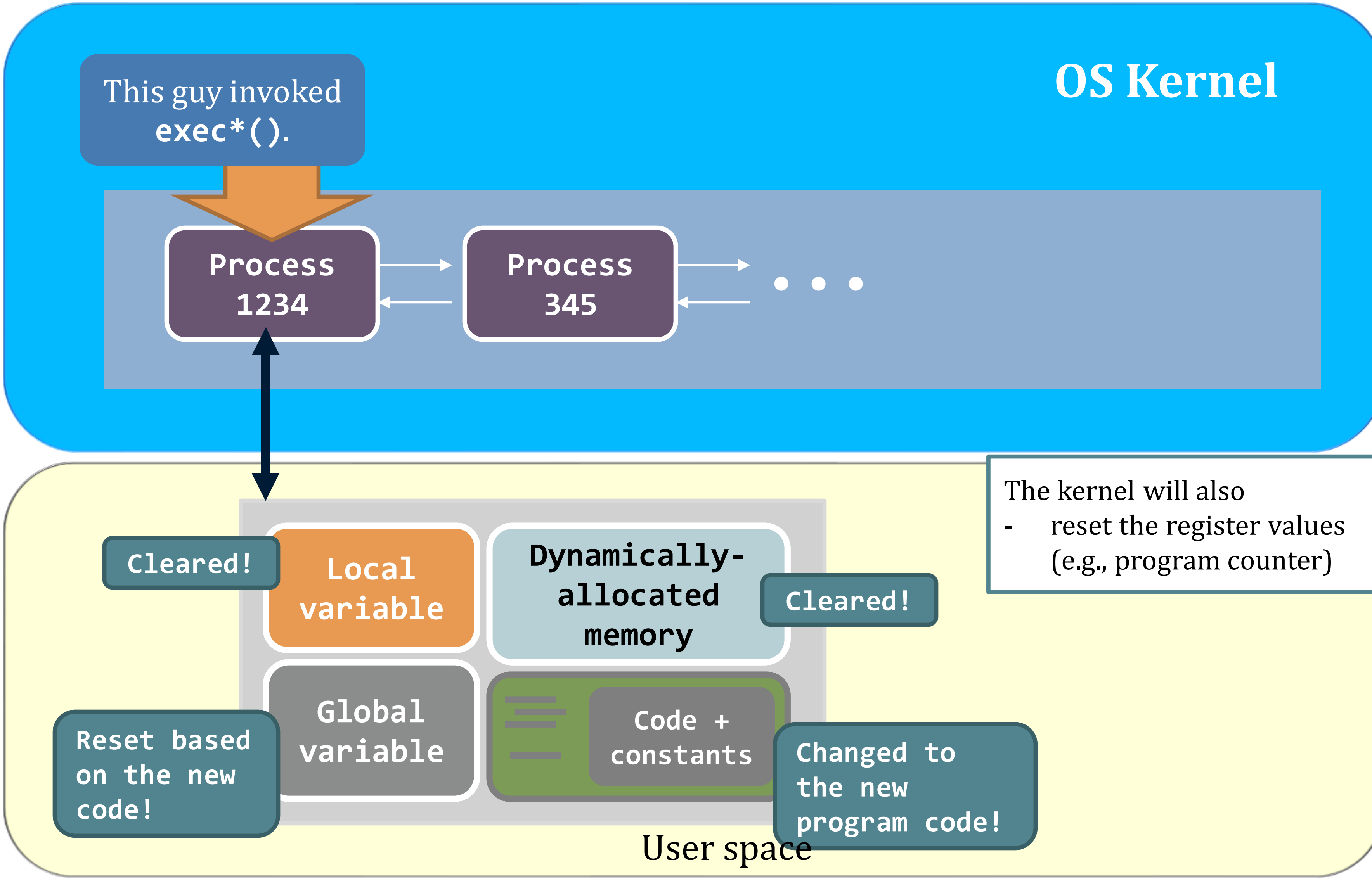
- `fork();`
- `exec*();`

exec*() that you've learnt…

- How about the **exec*()** call family?



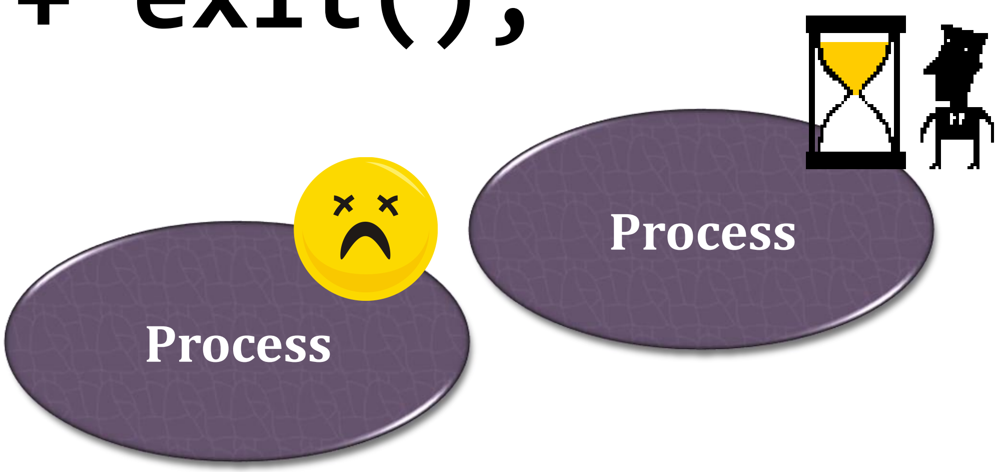
exec*() in action



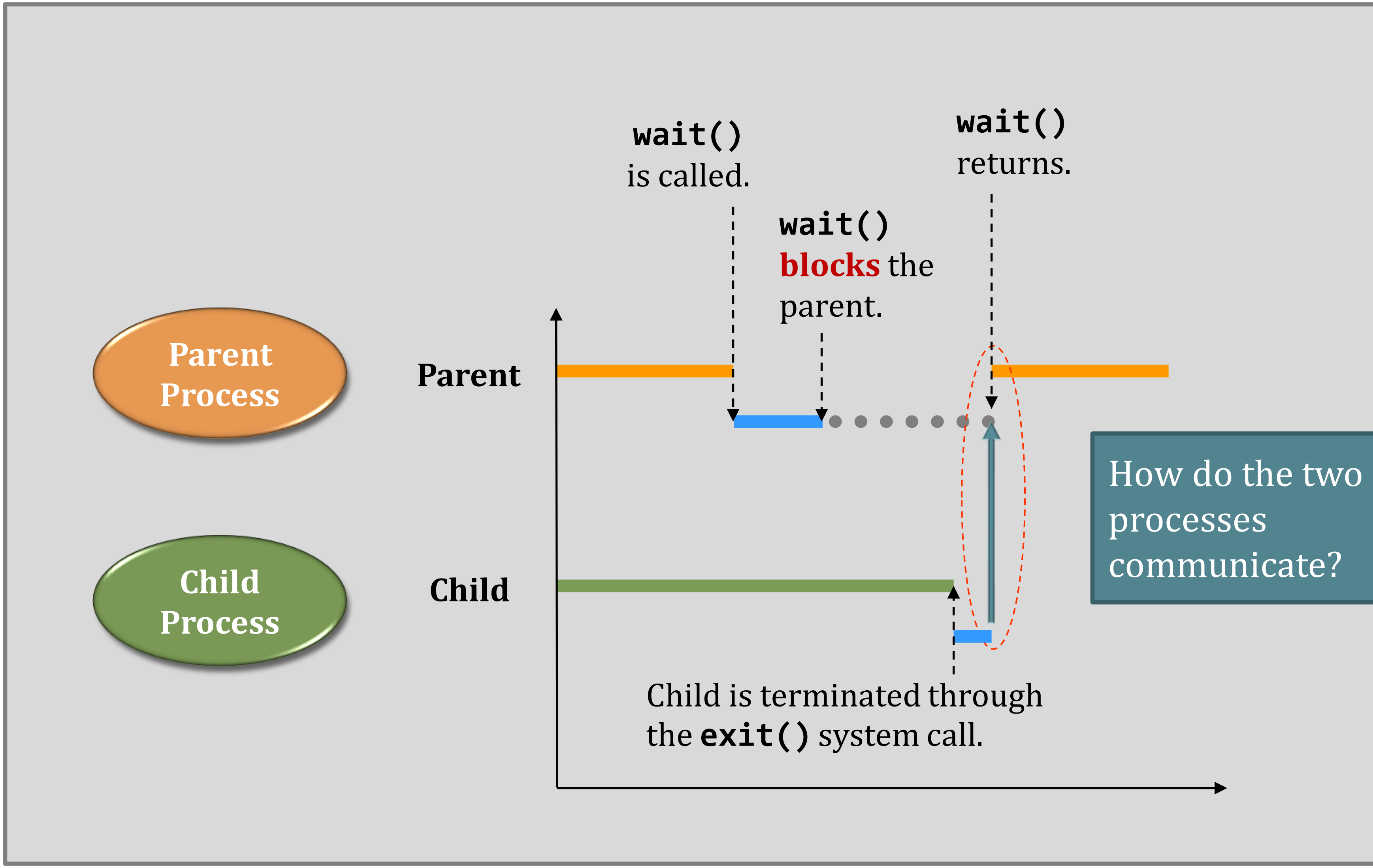
The kernel will also
- reset the register values (e.g., program counter)

Working of system calls

- `fork();`
- `exec*();`
- `wait() + exit();`

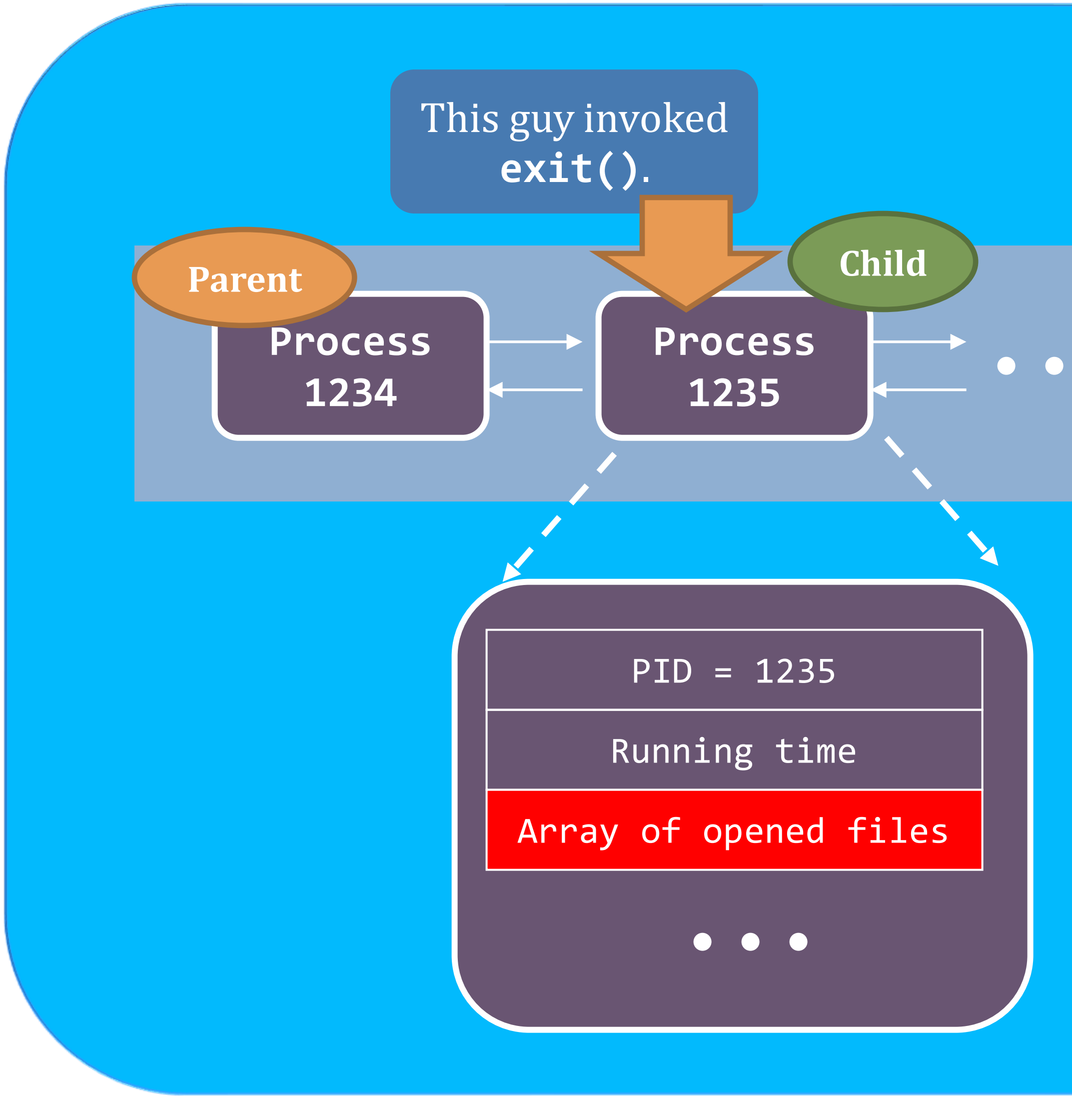


`wait()` and `exit()`



How do the two processes communicate?

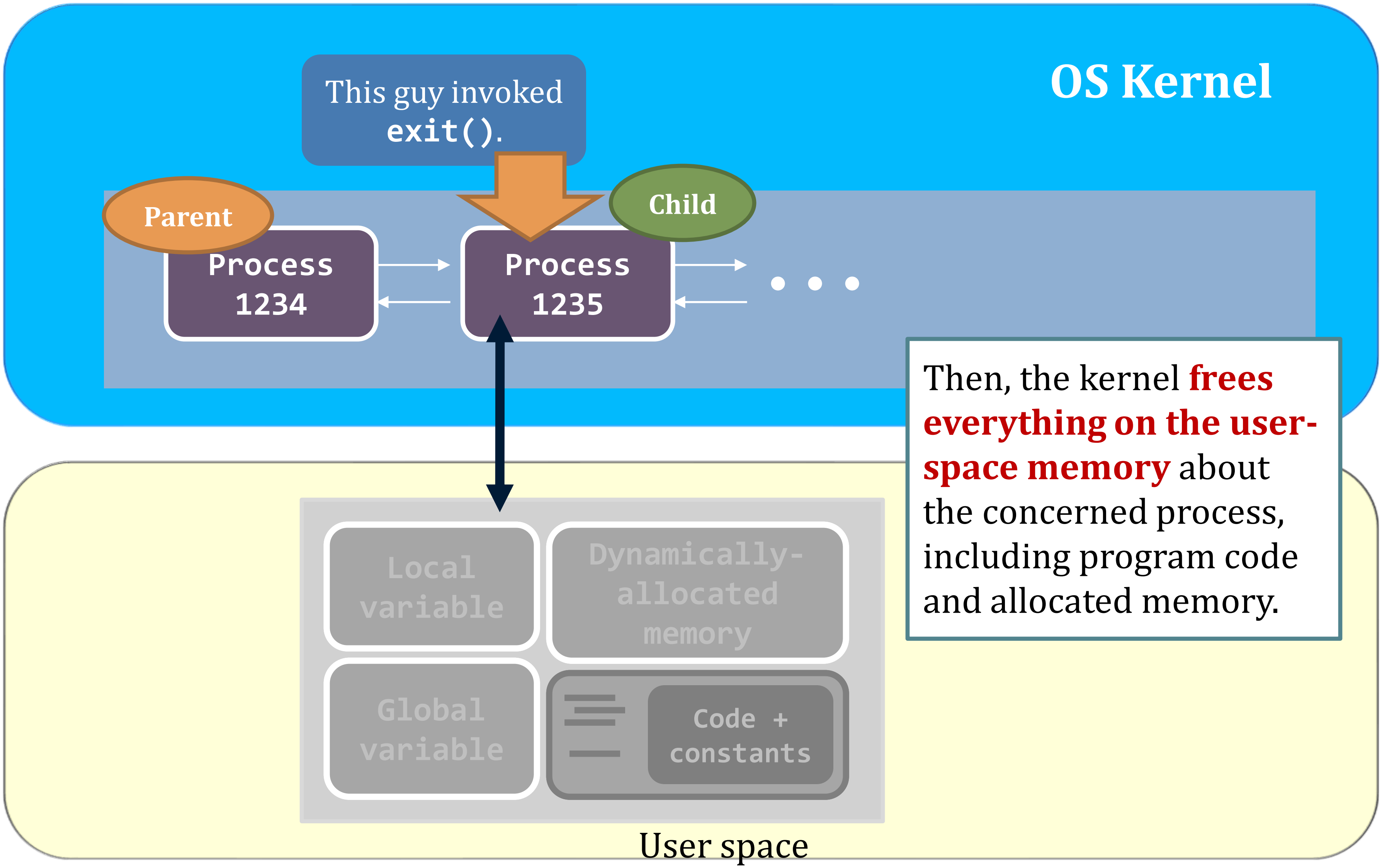
`exit()` (kernel-view)



The kernel frees all the allocated memory.

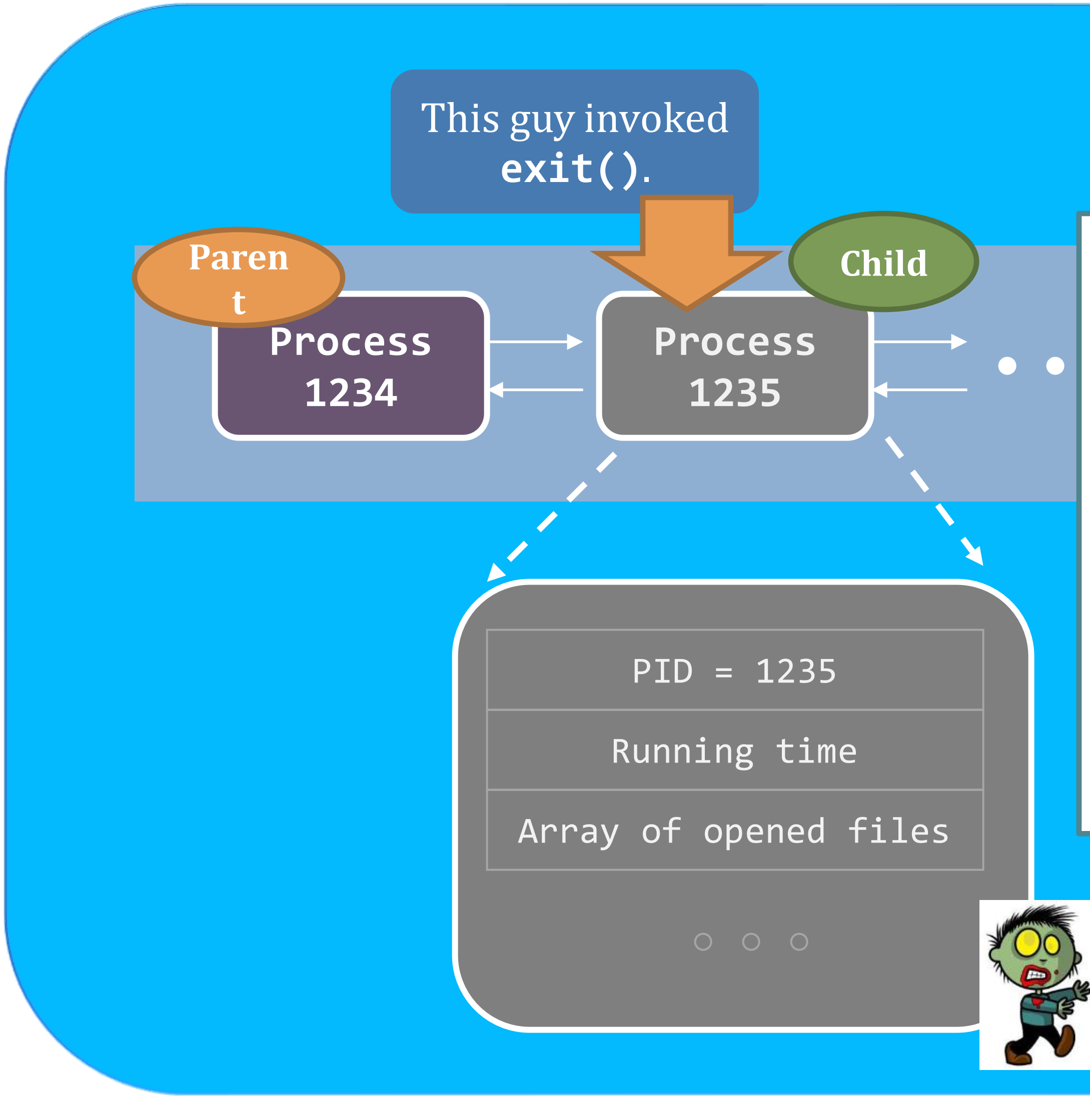
The list of opened files are all closed. (so it's okay to skip **`fclose()`**; though not recommended)

`exit()` (kernel-view)



Then, the kernel **frees everything on the user-space memory** about the concerned process, including program code and allocated memory.

exit() (kernel-view)

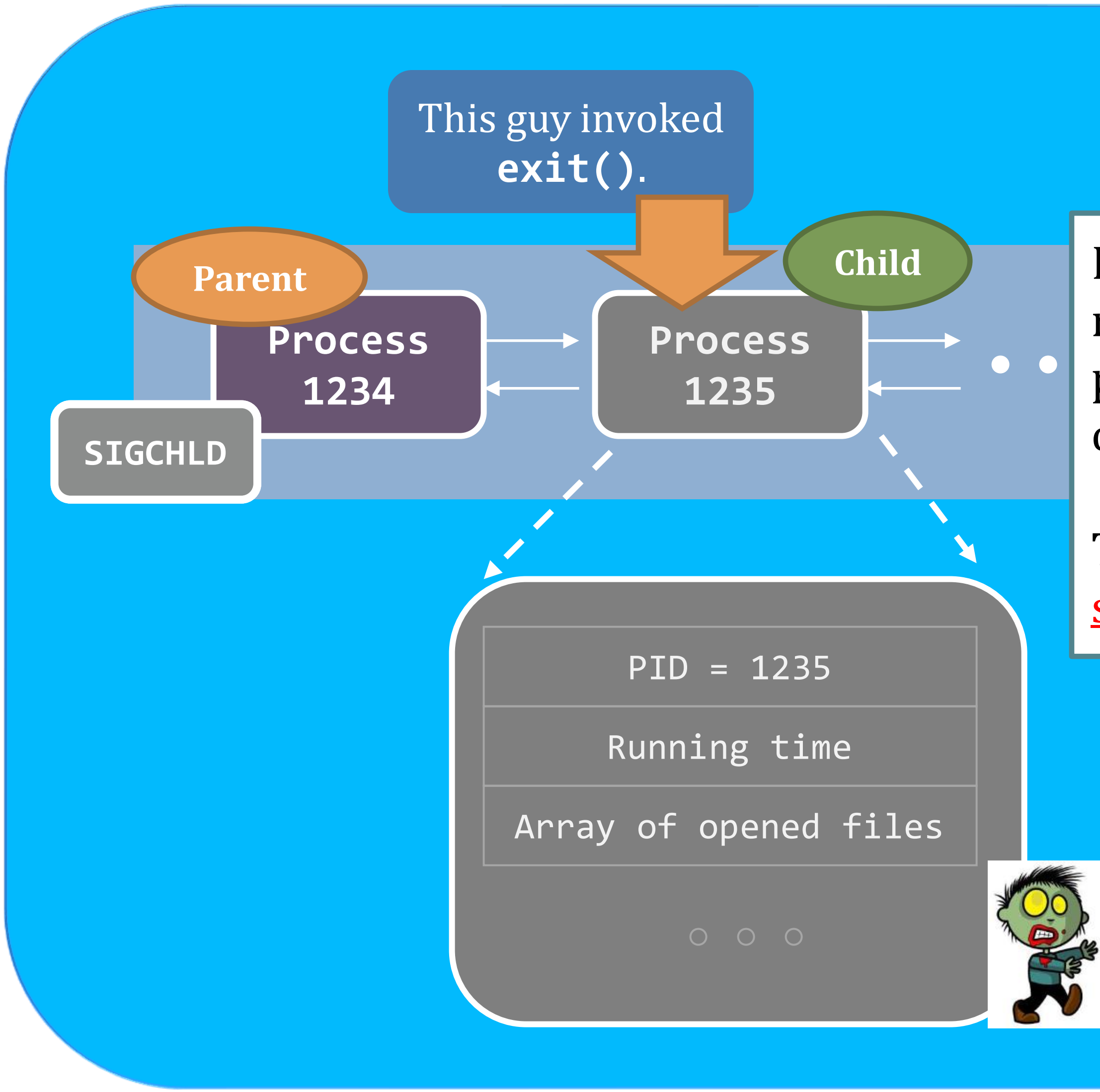


Process ID stills in the kernel’s process table

- Why?
  [Wiki] *This entry is still needed to allow the process that started the (now zombie) process to read its exit status.*

The status of the child is now called **zombie (“terminated”)**.

exit() (kernel-view)



Last but not least, the kernel notifies the parent of the child process about the termination of its child.

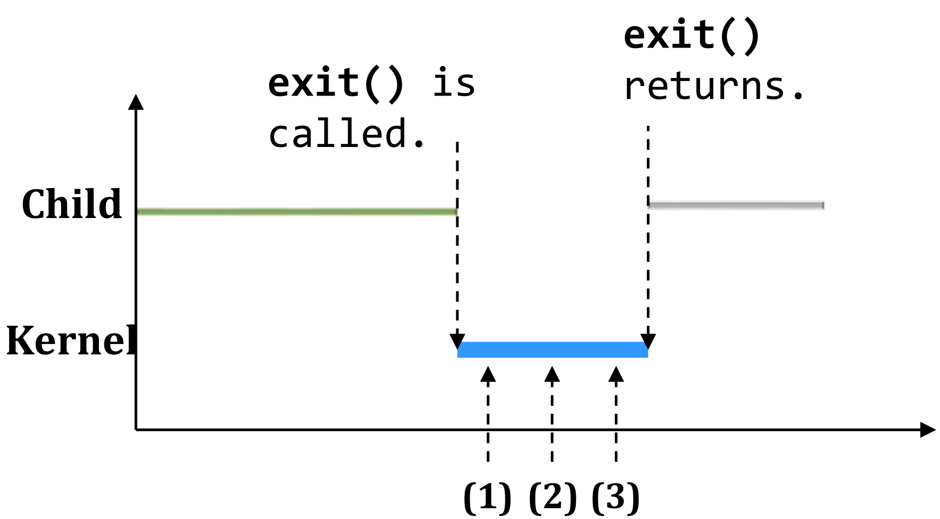
The kernel sends a **SIGCHLD** signal to the parent

Summary -- what the kernel does for **exit()**

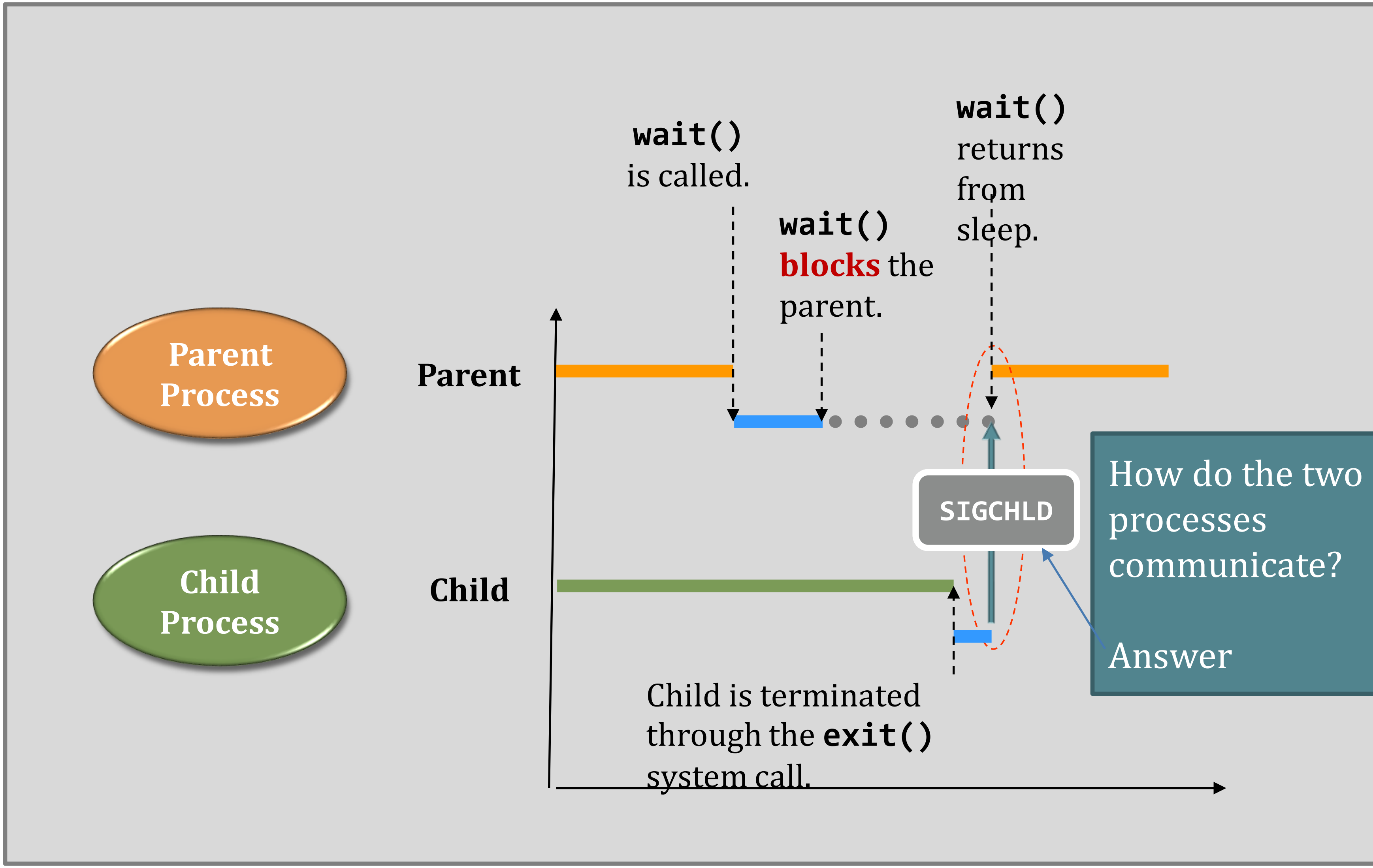
Step (1) Clean up most of the allocated kernel-space memory (e.g., process's running time info).

Step (2) Clean up the exit process's user-space memory.

Step (3) Notify the parent with SIGCHLD.



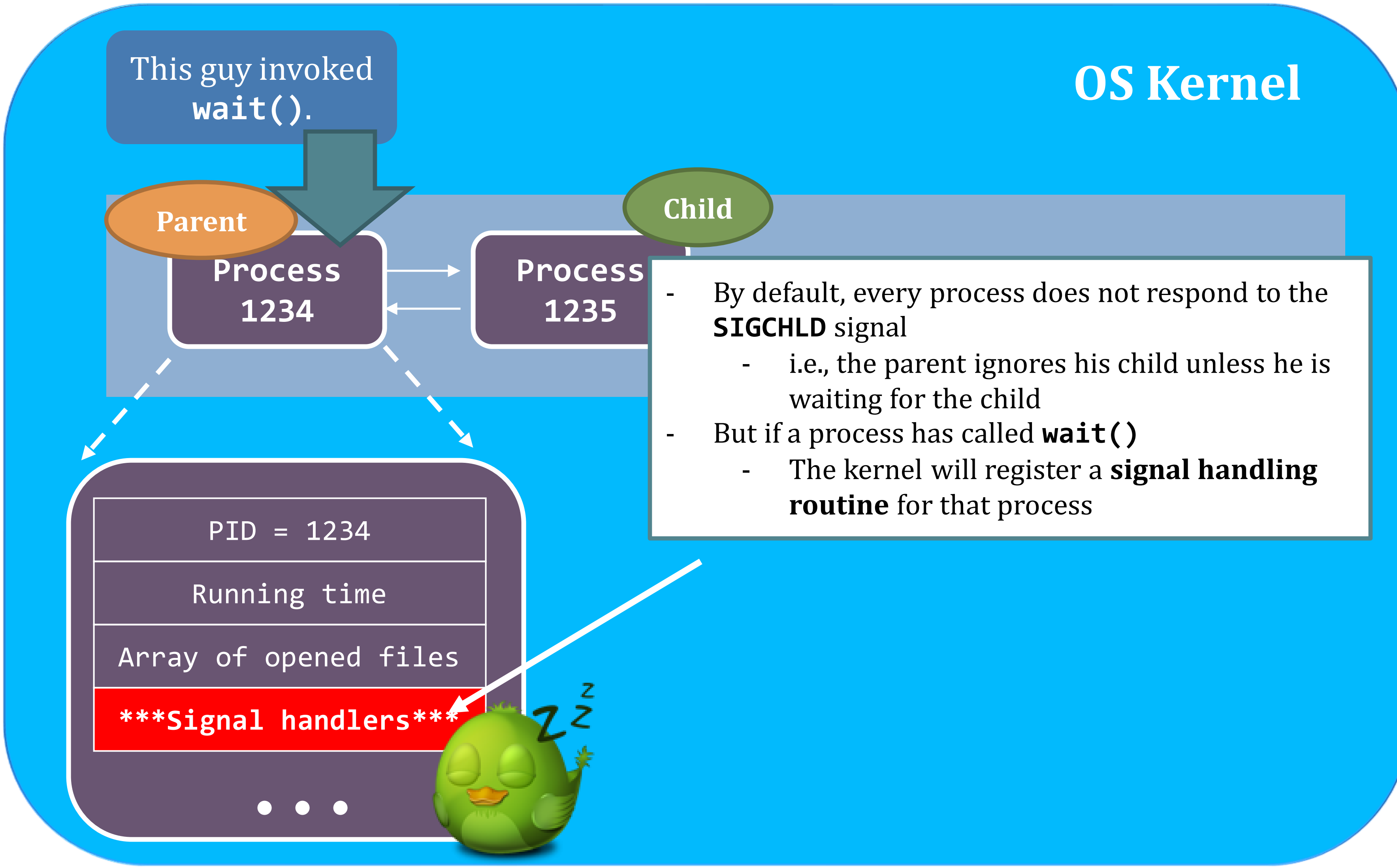
wait() and exit()



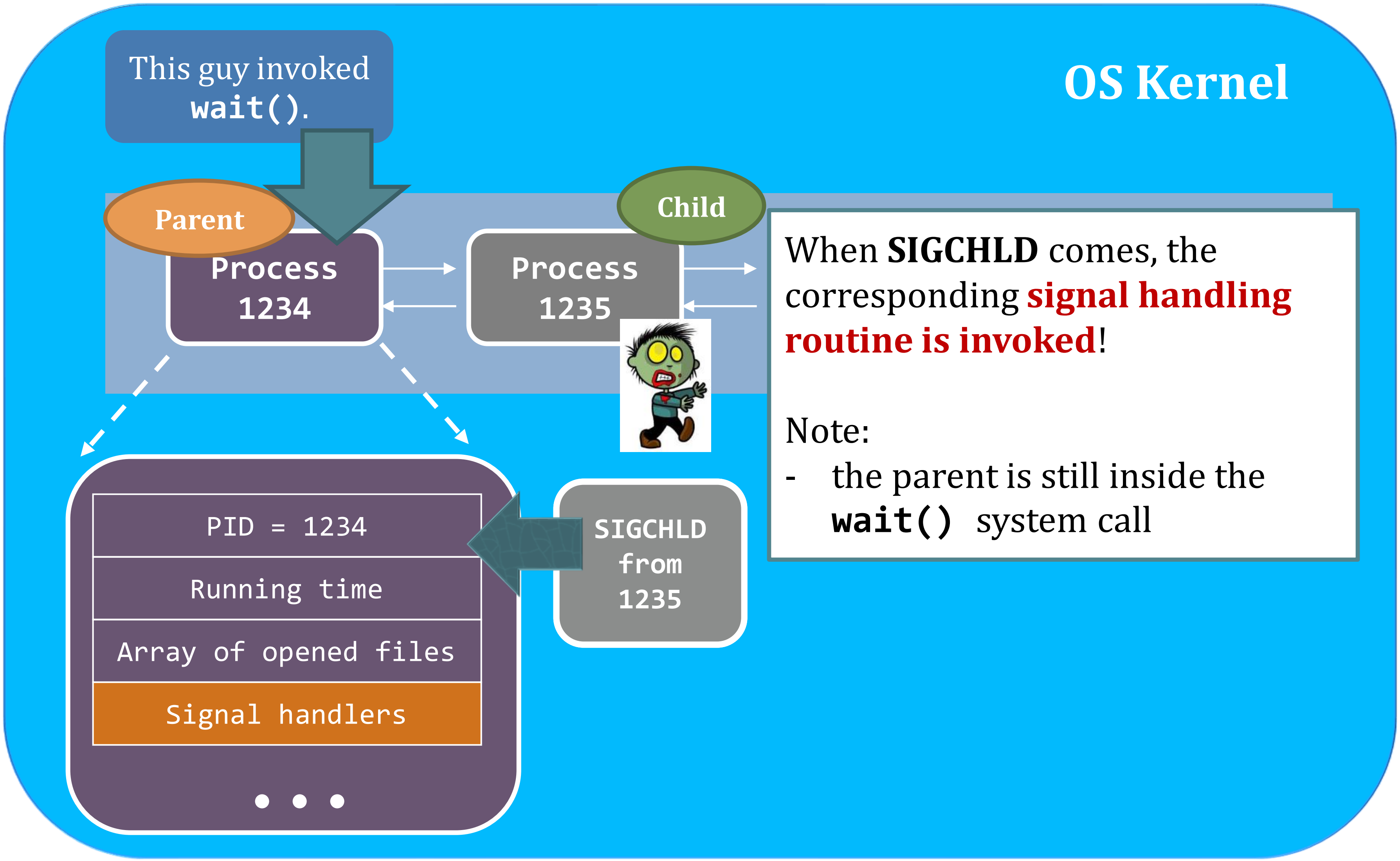
How do the two processes communicate?

Answer

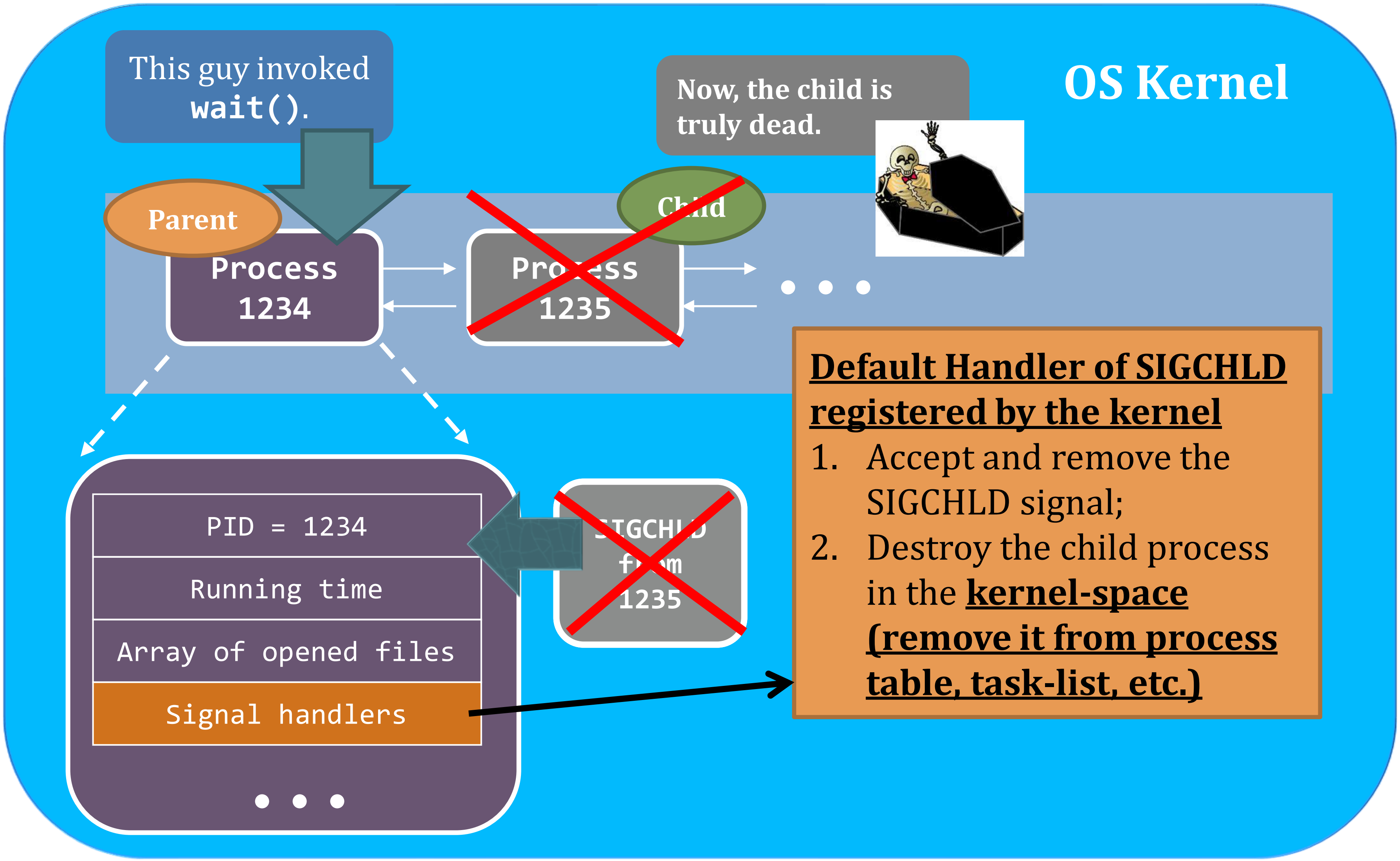
`wait()` kernel view's – registering signal handling routine



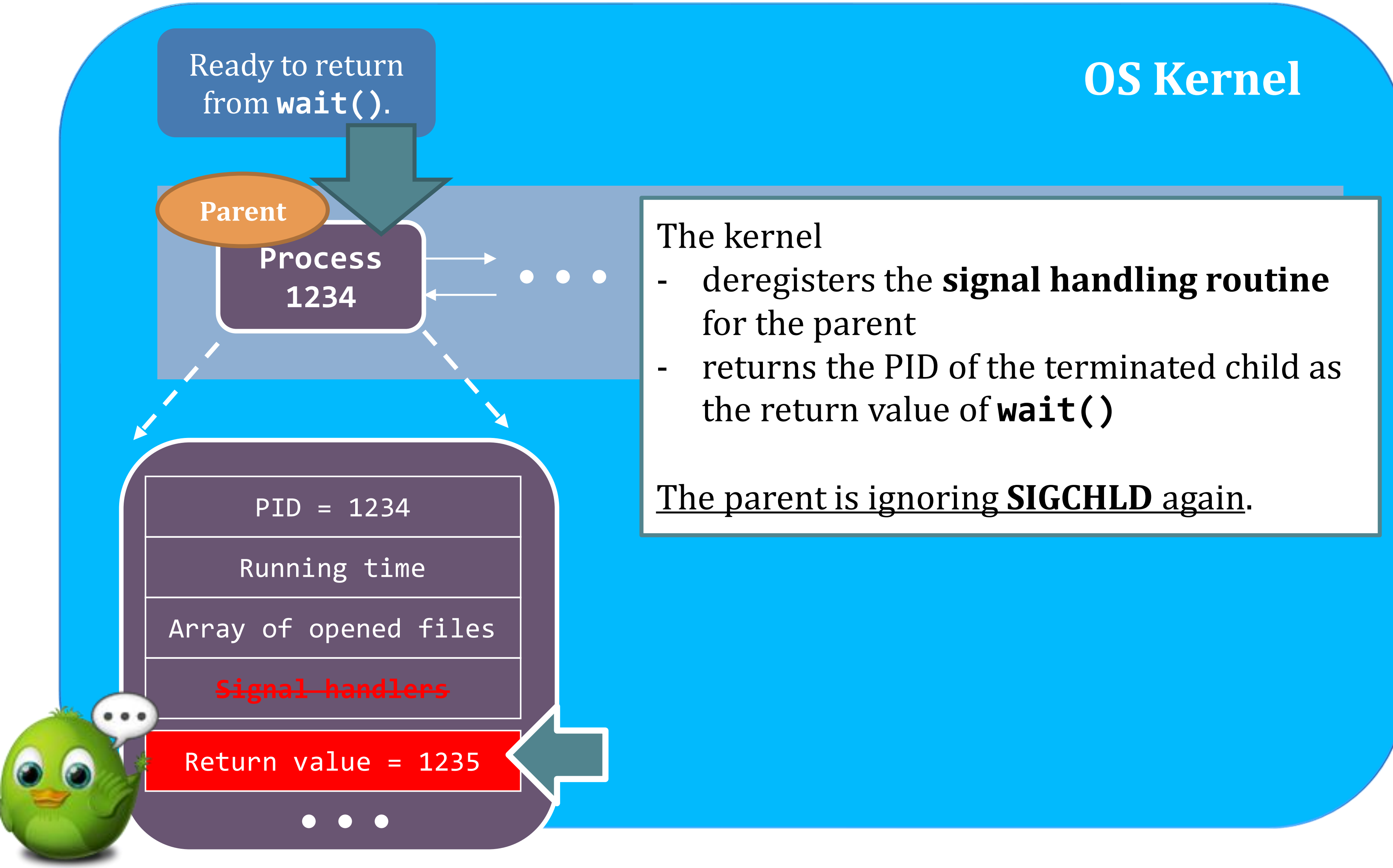
`wait()` kernel's view



wait() kernel's view



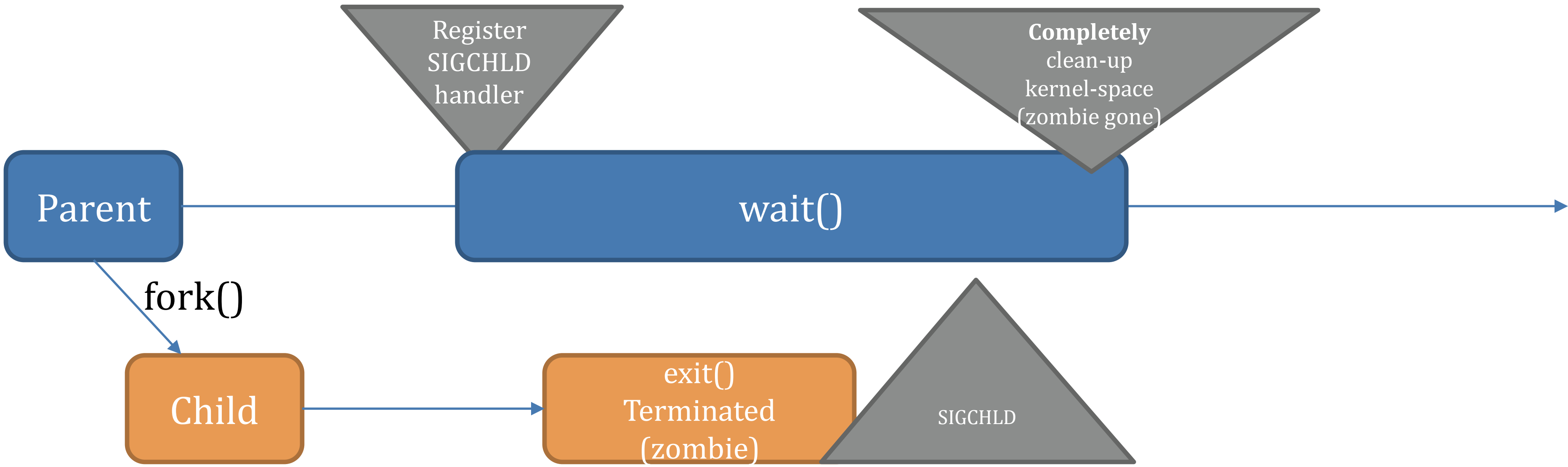
`wait()` kernel's view



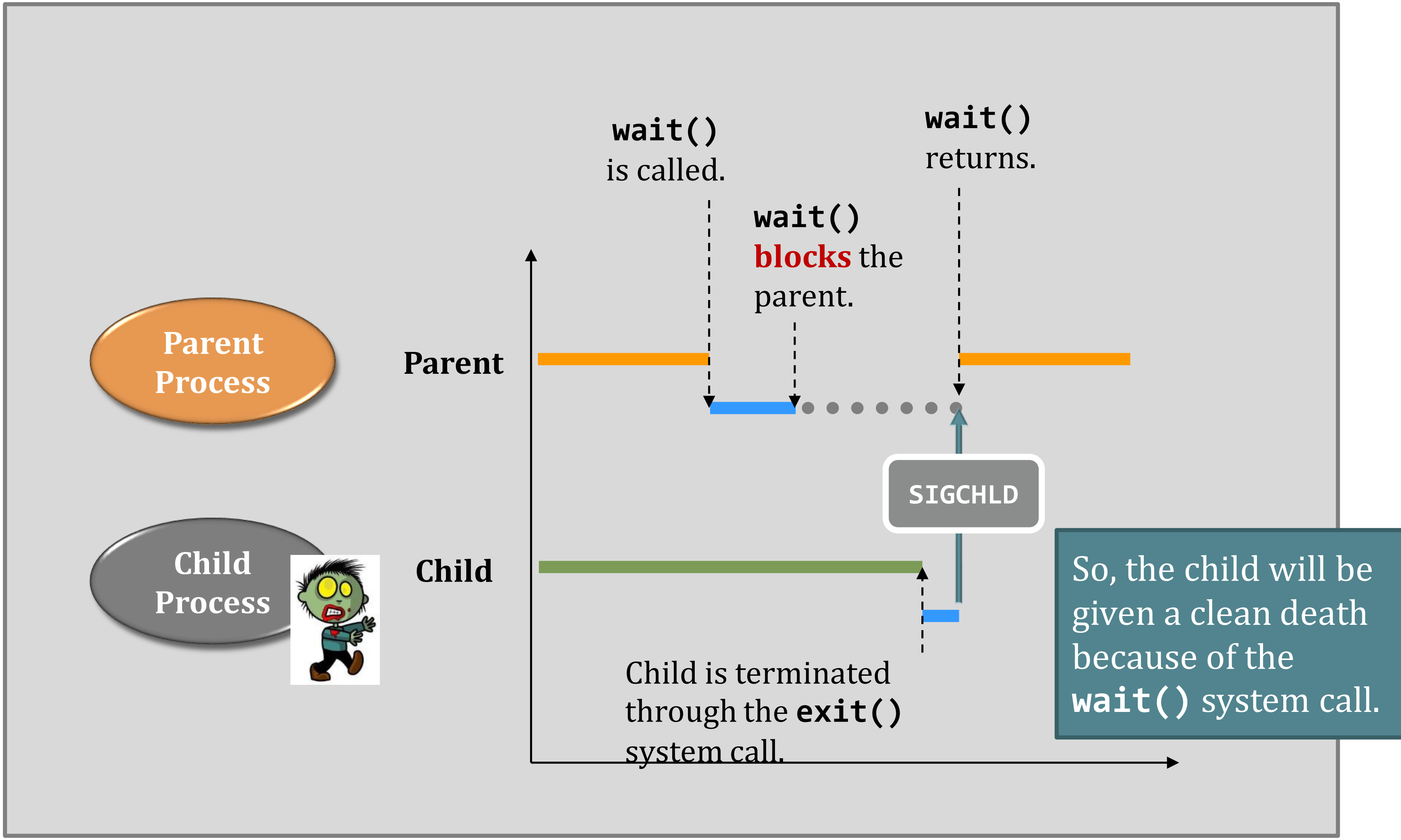
The kernel
- deregisters the **signal handling routine** for the parent
- returns the PID of the terminated child as the return value of **`wait()`**

The parent is ignoring **SIGCHLD** again.

Overall – normal case

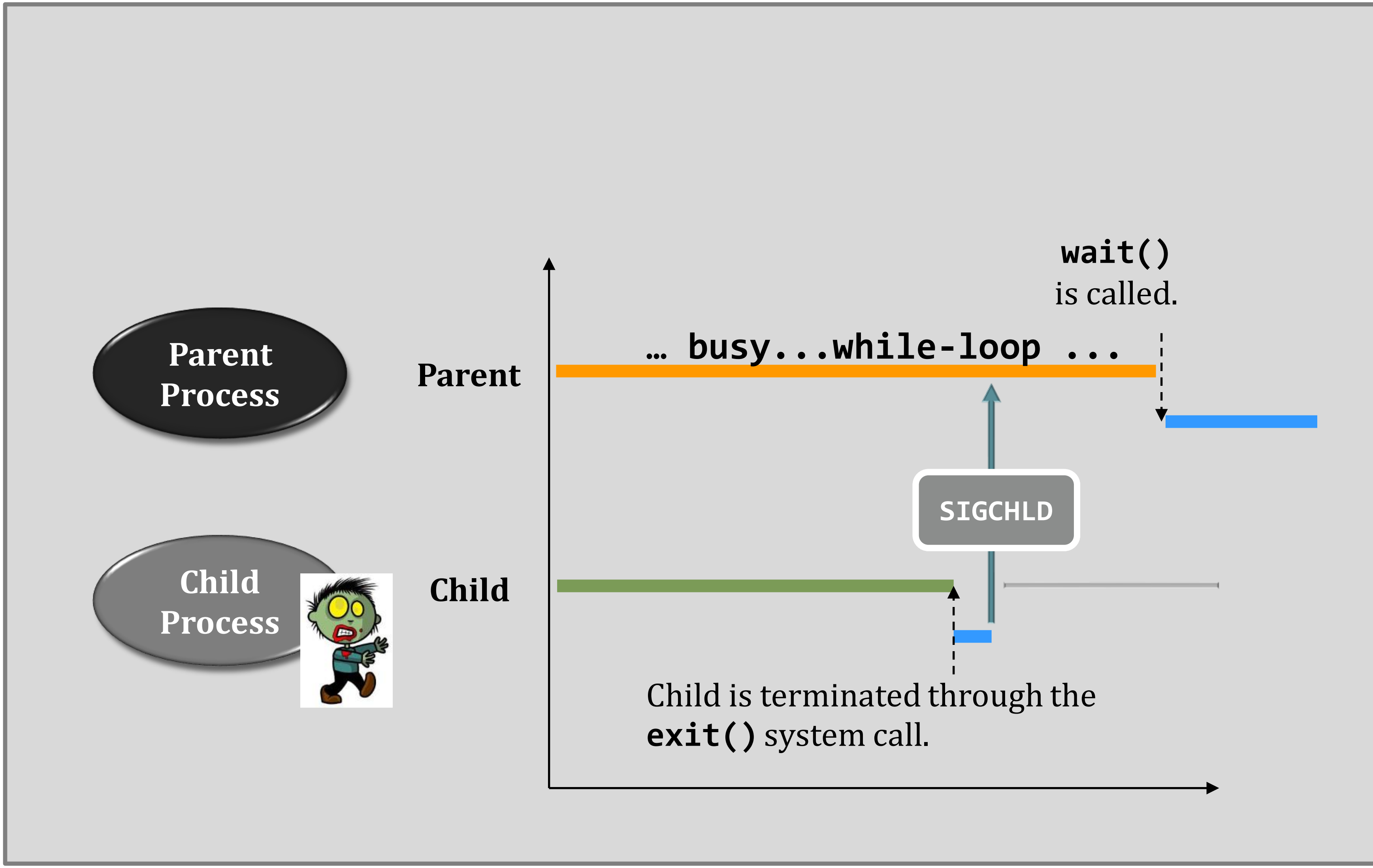


Normal Case

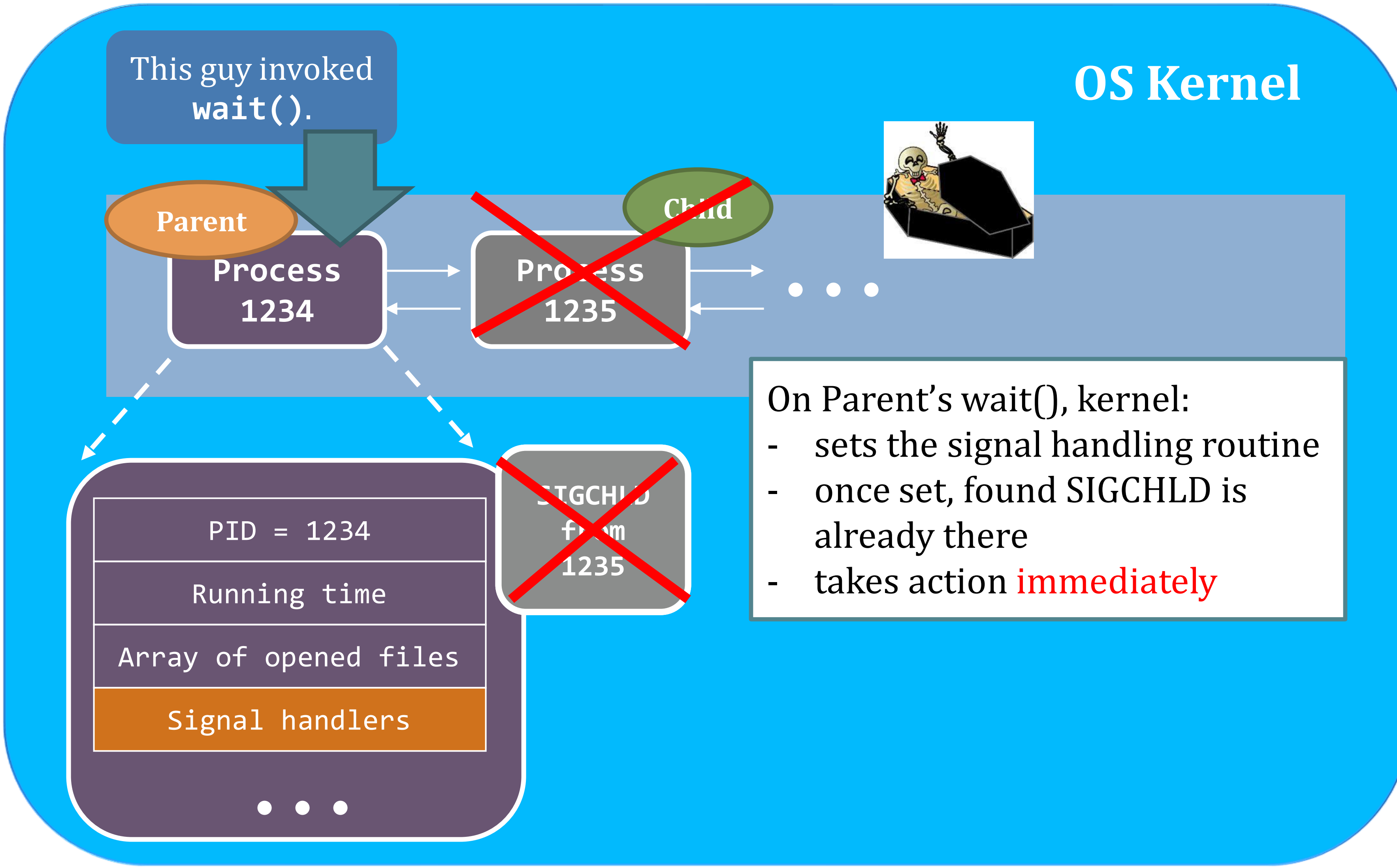


Parent Process
Child Process
Parent
Child
wait() is called.
wait() blocks the parent.
wait() returns.
SIGCHLD
Child is terminated through the exit() system call.
So, the child will be given a clean death because of the wait() system call.

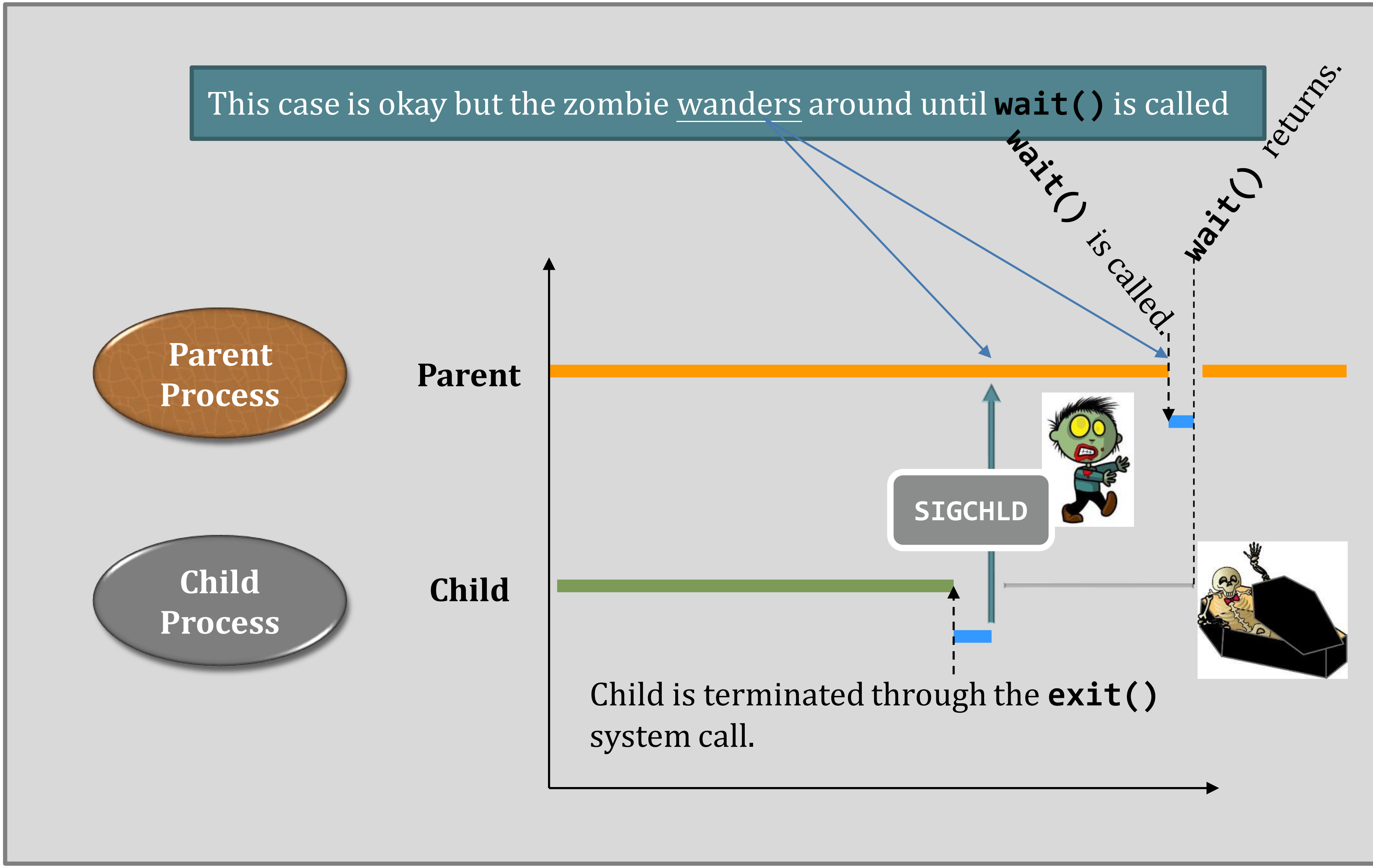
Parent's wait() after Child's exit()



Parent's Wait() after Child's exit()



Parent's Wait() after Child's exit()



`wait()` and `exit()` – short summary

- **`exit()`** system call turns a process into a zombie when…
  - The process calls **`exit()`**.
  - The process returns from **`main()`**.
  - The process terminates abnormally.
    - The kernel knows that the process is terminated abnormally. Hence, the kernel invokes **`exit()`** for it.

`wait()` and `exit()` – short summary

- **`wait()`** & **`waitpid()`** are to reap zombie child processes.
  - It is a must that you should never leave any zombies in the system.
  - **`wait()`** & **`waitpid()`** pause the caller until
    - A child terminates/stops, OR
    - The caller receives a signal (i.e., the signal interrupted the `wait()`)
- Linux will label zombie processes as "**`<defunct>`**".
  - To look for them:

```
$ ps aux | grep defunct
……..... 3150 ... [ls] <defunct>
$ _
```

PID of the process

wait() and exit() – short summary

```c
int main(void)
{
    int pid;
    if( (pid = fork()) !=0 ) {
        printf("Look at the status of the child process %d\n", pid);
        while( getchar() != '\n' );
        wait(NULL);
        printf("Look again!\n");
        while( getchar() != '\n' );
    }
    return 0;
}
```

“enter” here

“enter” here

This program requires you to type “enter” twice before the process terminates.

You are expected to see **the status of the child process changes (ps aux [PID])** between the 1st and the 2nd “enter”.

Working of system calls

```
- fork();
- exec*();
- wait() + exit();
- importance/fun in knowing
  the above things?
```

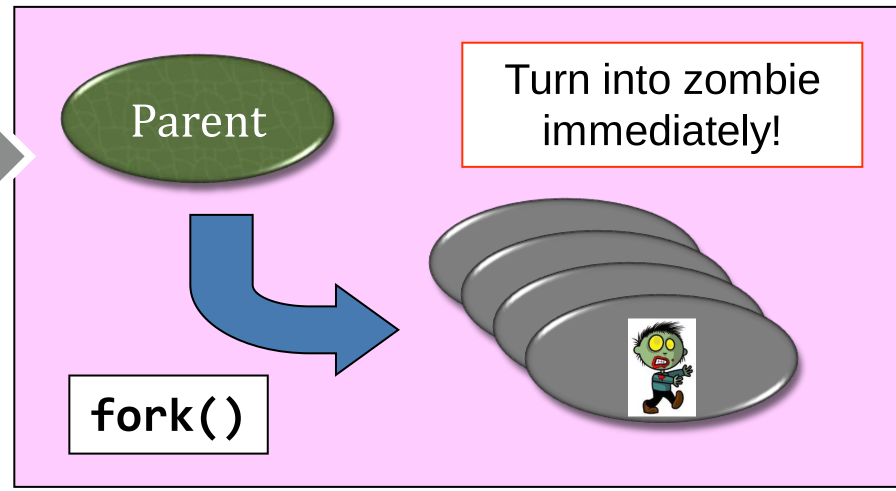
Calling **wait()** is important.

- It is not only about process execution / suspension…
- It is about **system resource management**.
  - A zombie takes up a PID;
  - The total number of PIDs are limited;
    - Read the limit: "**`cat /proc/sys/kernel/pid_max`**"
    - It is 32,768.
  - What will happen if we don't clean up the zombies?

The fork bomb

- Deliberately missing wait()
- Do not try this on department's machines…

```c
int main(void) {
    while( fork() );
    return 0;
}
```



An infinite, zombie factory!

When `wait()` is absent...

```c
int main(void) {
    while( fork() );
    return 0;
}
```

```
$ ./interesting
_
```

Terminal A

```
$ ls
No process left.
$ poweroff
No process left.
$ =__=
No process left.
$ _
```

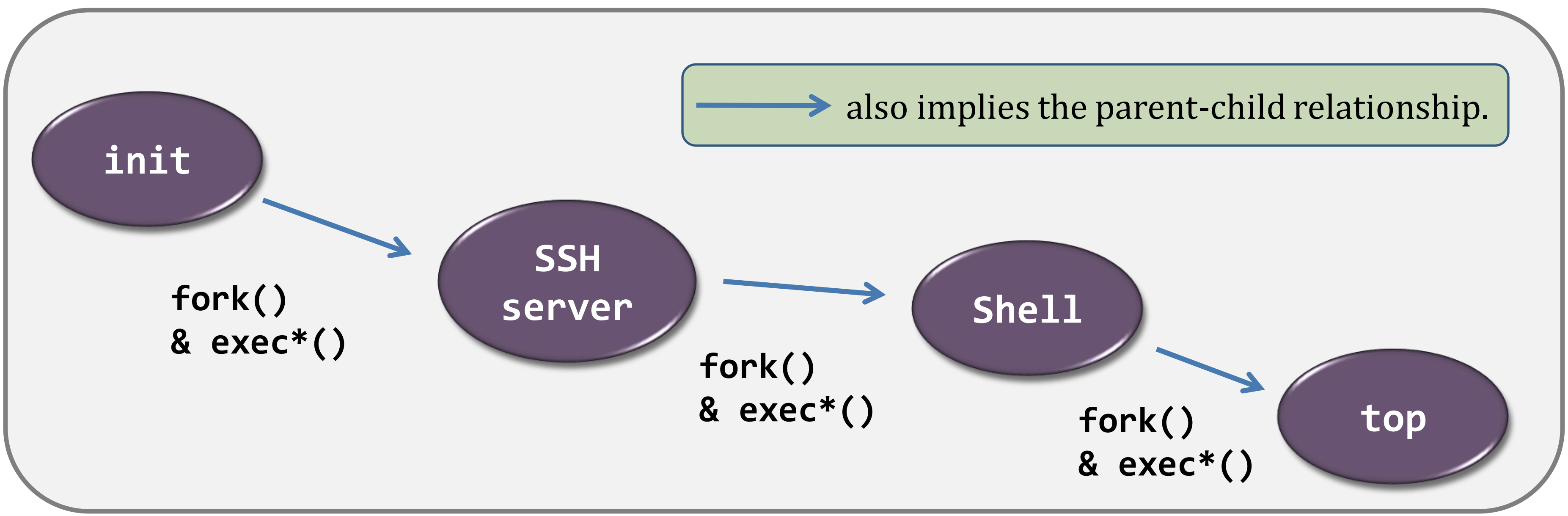
Terminal B

The first process

- We now focus on the process-related events.
  - The kernel, while it is booting up, creates the first process – **`init`**.
- The "**`init`**" process:
  - has **PID = 1**, and
  - is running the program code "**`/sbin/init`**".
- Its first task is to **create more processes**…
  - Using **`fork()`** and **`exec*()`**.

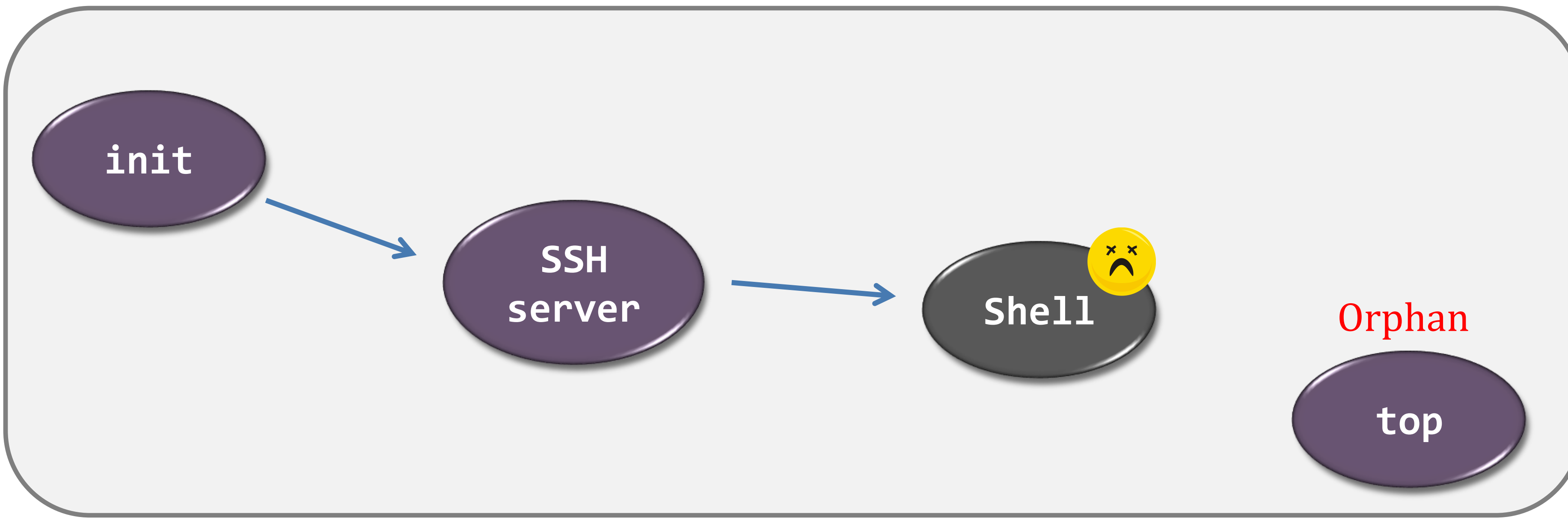
Process blossoming

- You can view the tree with the command:
  - "**pstree**"; or
  - "**pstree -A**" for ASCII-character-only display.



Process blossoming…with orphans?

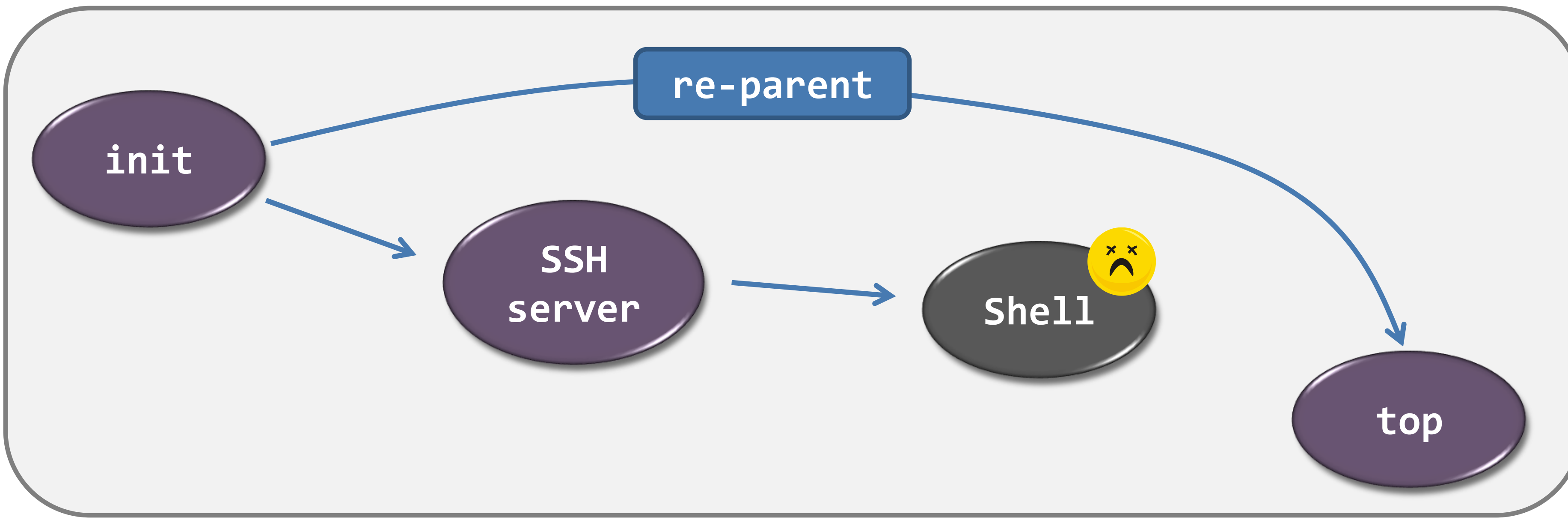
- However, termination can happen, at any time and in any place…
  - This is no good because an orphan turns the hierarchy from a **tree** into a **forest**!
  - Plus, no one would know the termination of the orphan.



Process blossoming…with re-parent!

- In Linux
  - The "**`init`**" process will become the step-mother of all orphans
  - It's called **re-parenting**

- In Windows
  - It maintains a *forest-like process hierarchy*…………..



*New Linux kernels may choose someone else (e.g., the grandparent, user-level init)

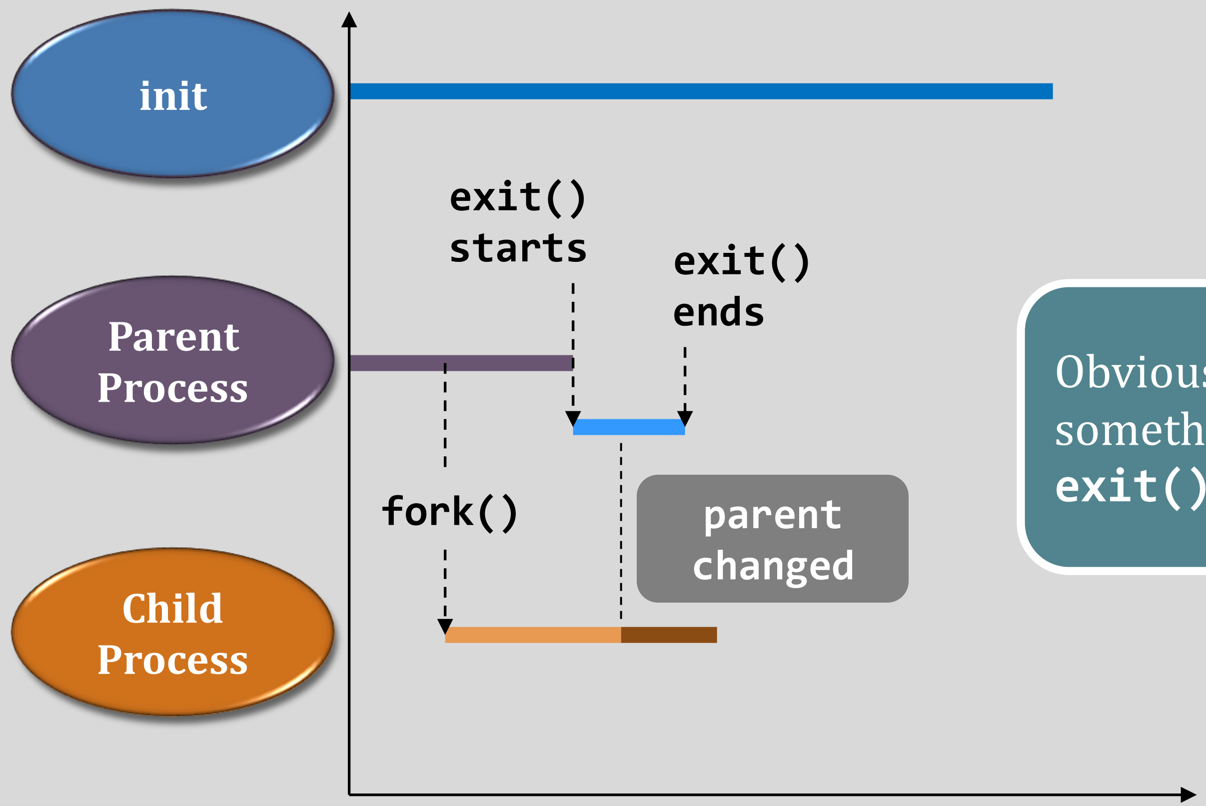
Re-parenting example

```c
int main(void) {
    int i;
    if(fork() == 0) {
        for(i = 0; i < 5; i++) {
            printf("(%d) parent's PID = %d\n",
                   getpid(), getppid() );
        sleep(1);
        }
    }
    else
        sleep(1);
    printf("(%d) bye.\n", getpid());
}
```

```
$ ./reparent
(1235) parent's PID = 1234
(1235) parent's PID = 1234
(1234) bye.
$ (1235) parent’s PID = 1
(1235) parent’s PID = 1
(1235) parent’s PID = 1
(1235) bye.
$ _
```

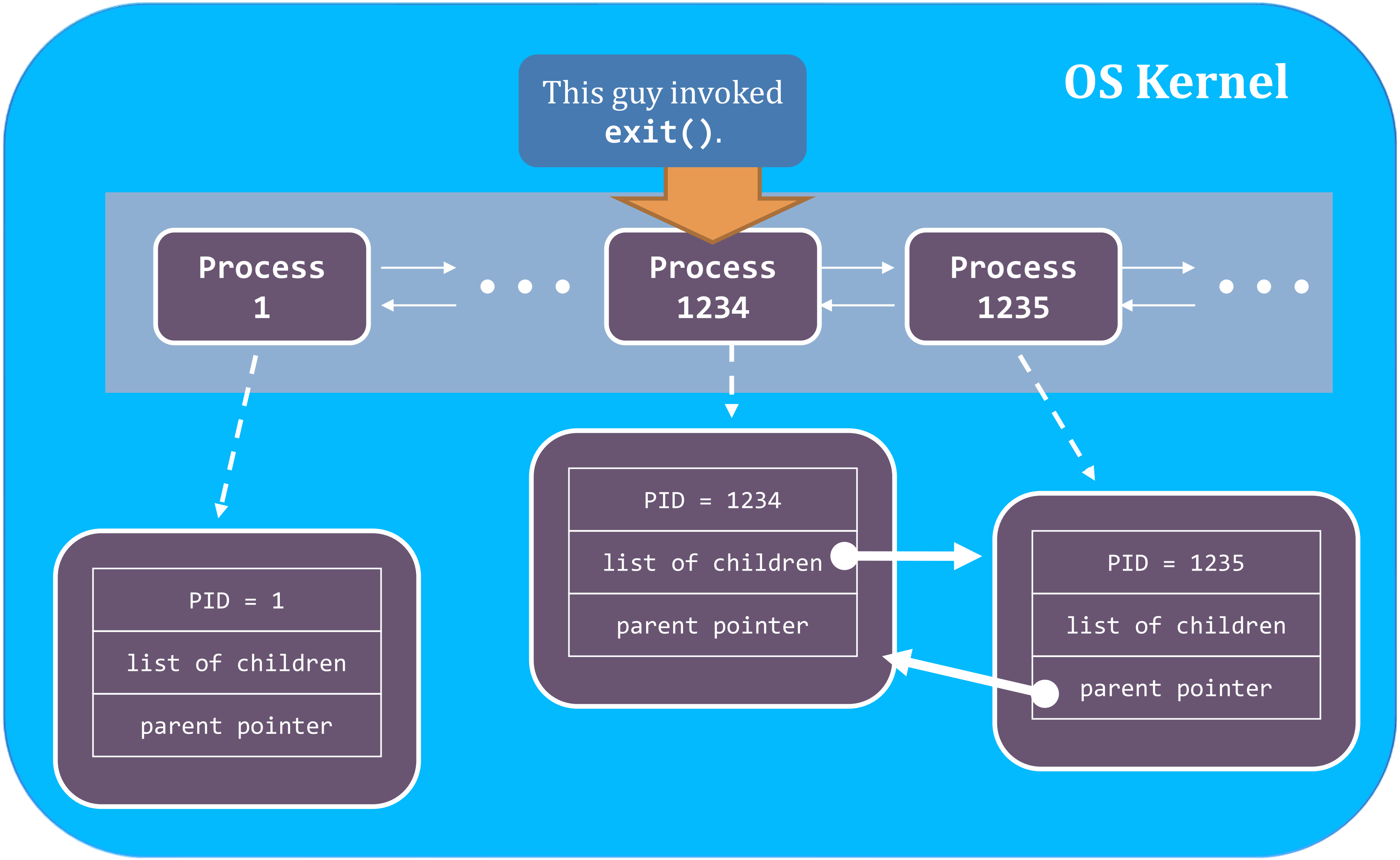
getppid() is the system call that returns the parent’s PID of the calling process.

What had happened during re-parenting?

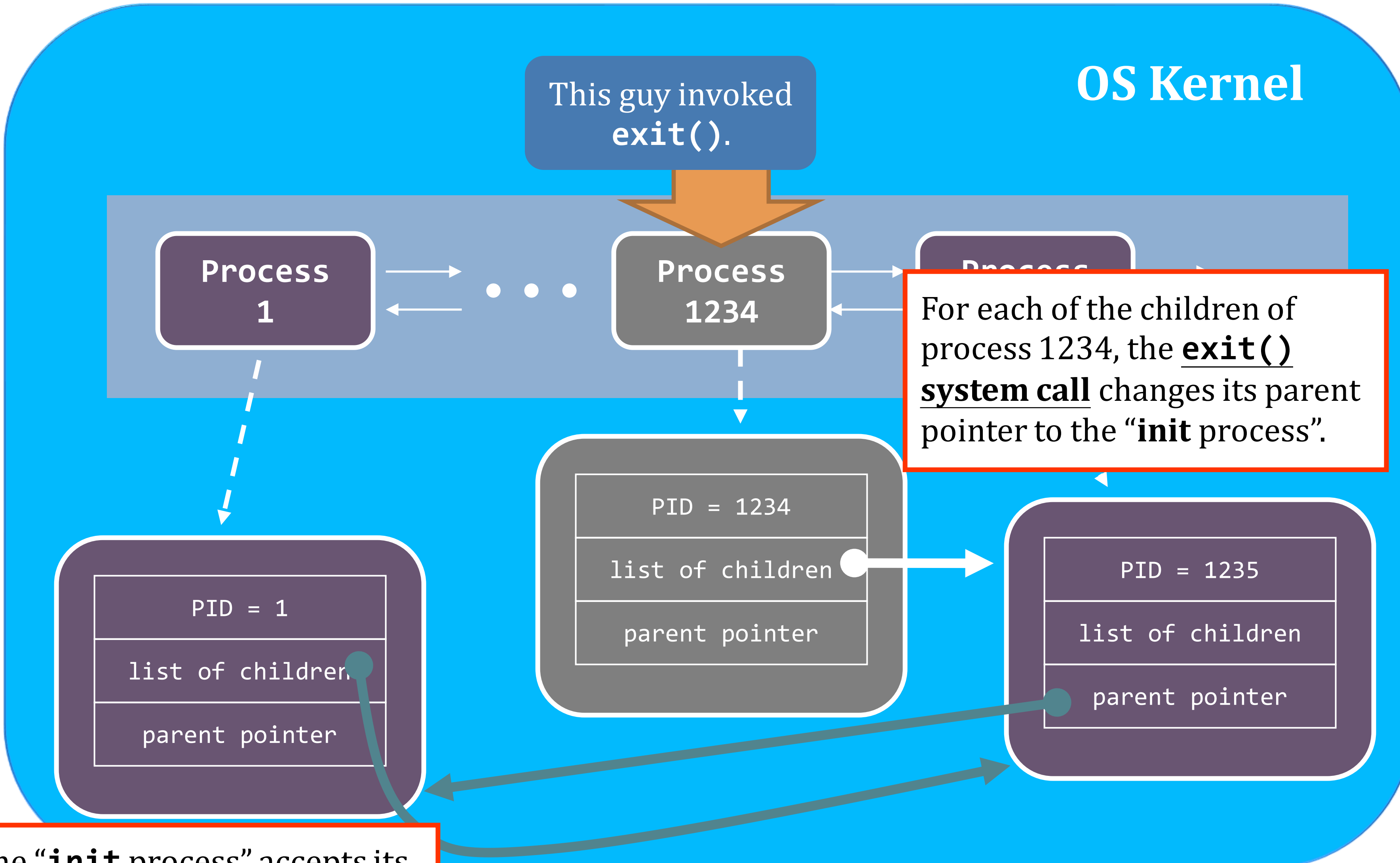


Obviously, there is something to do inside the **`exit()`** system call...

What had happened during re-parenting?



What had happened during re-parenting?



For each of the children of process 1234, the <u>**exit()** **system call**</u> changes its parent pointer to the “**init** process”.

The “**init** process” accepts its new child by adding the new child into the “**list of children**”.

Background jobs

- The re-parenting operation enables something called **background jobs** in Linux
  - It allows a process runs **without a parent terminal/shell**

Back to home

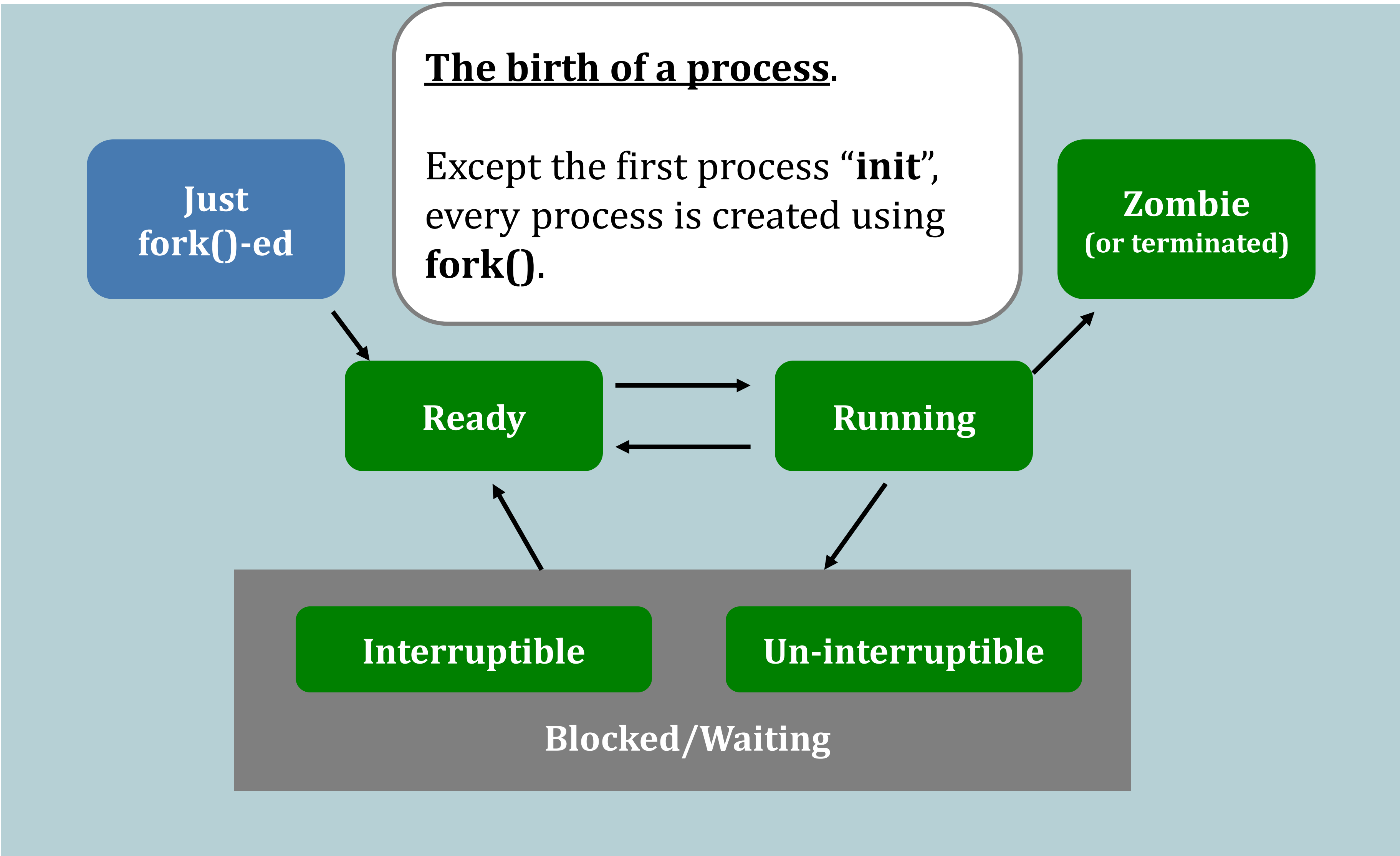
```
$ ./infinite_loop &
$ exit

[ The shell is gone ]
```

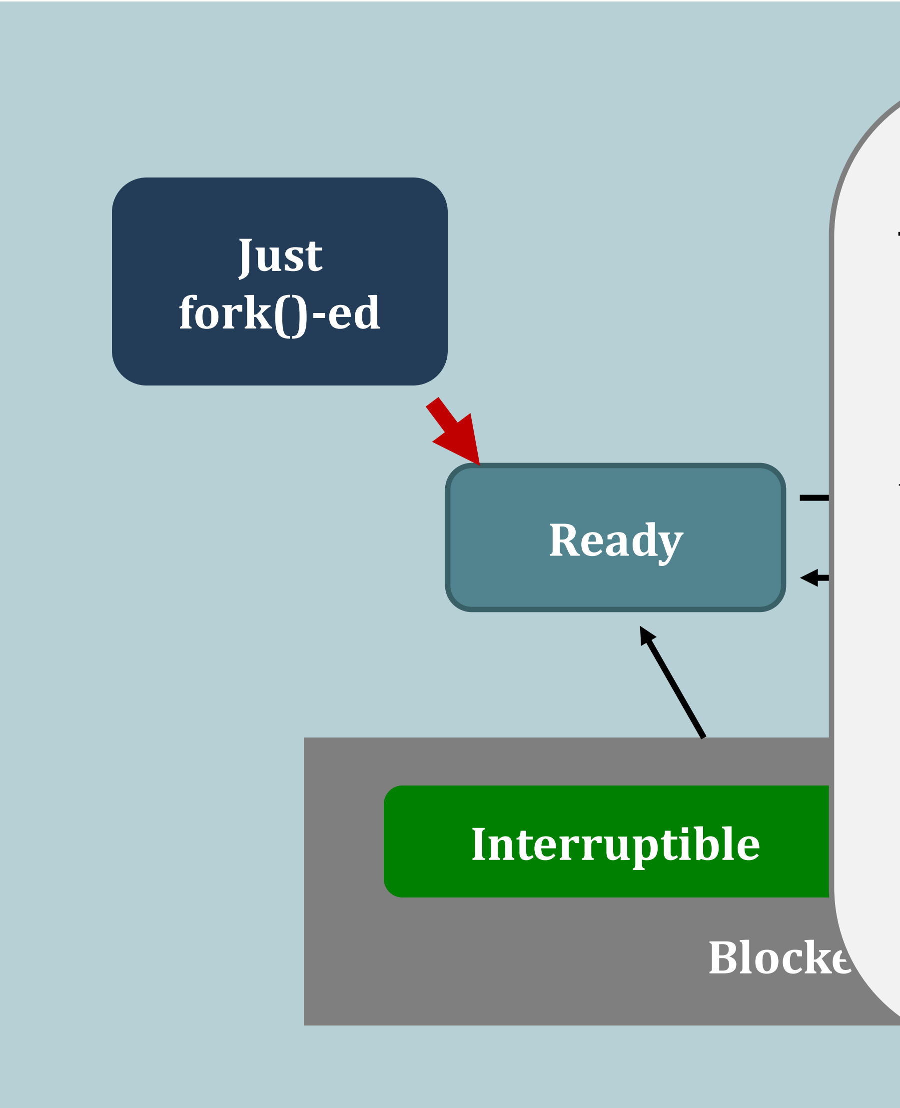
```
$ ps -C infinite_loop
 PID  TTY
1234  ... ./infinite_loop
$ _
```

Will see more in detail soon

Process lifecycle



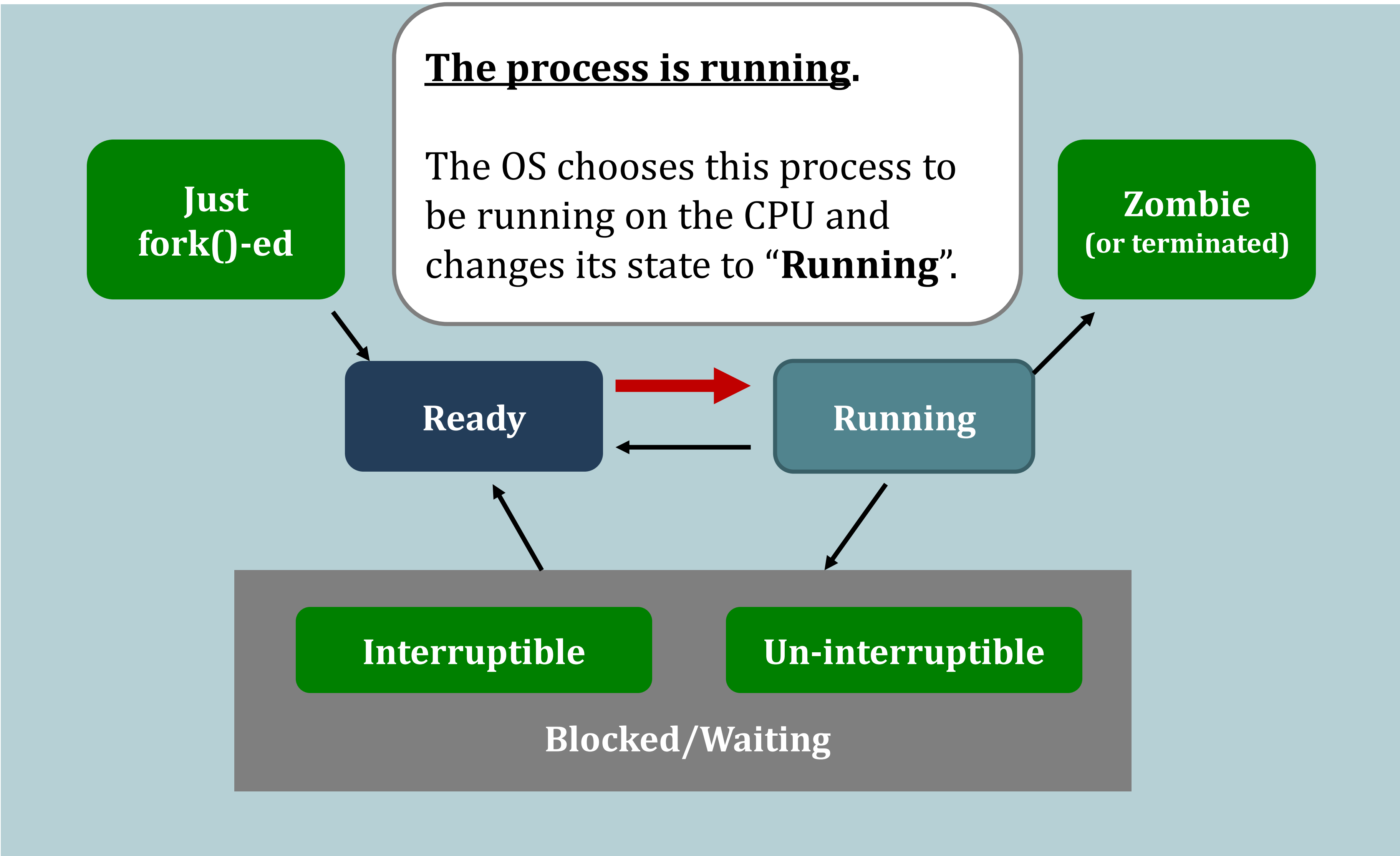
Process lifecycle - Ready



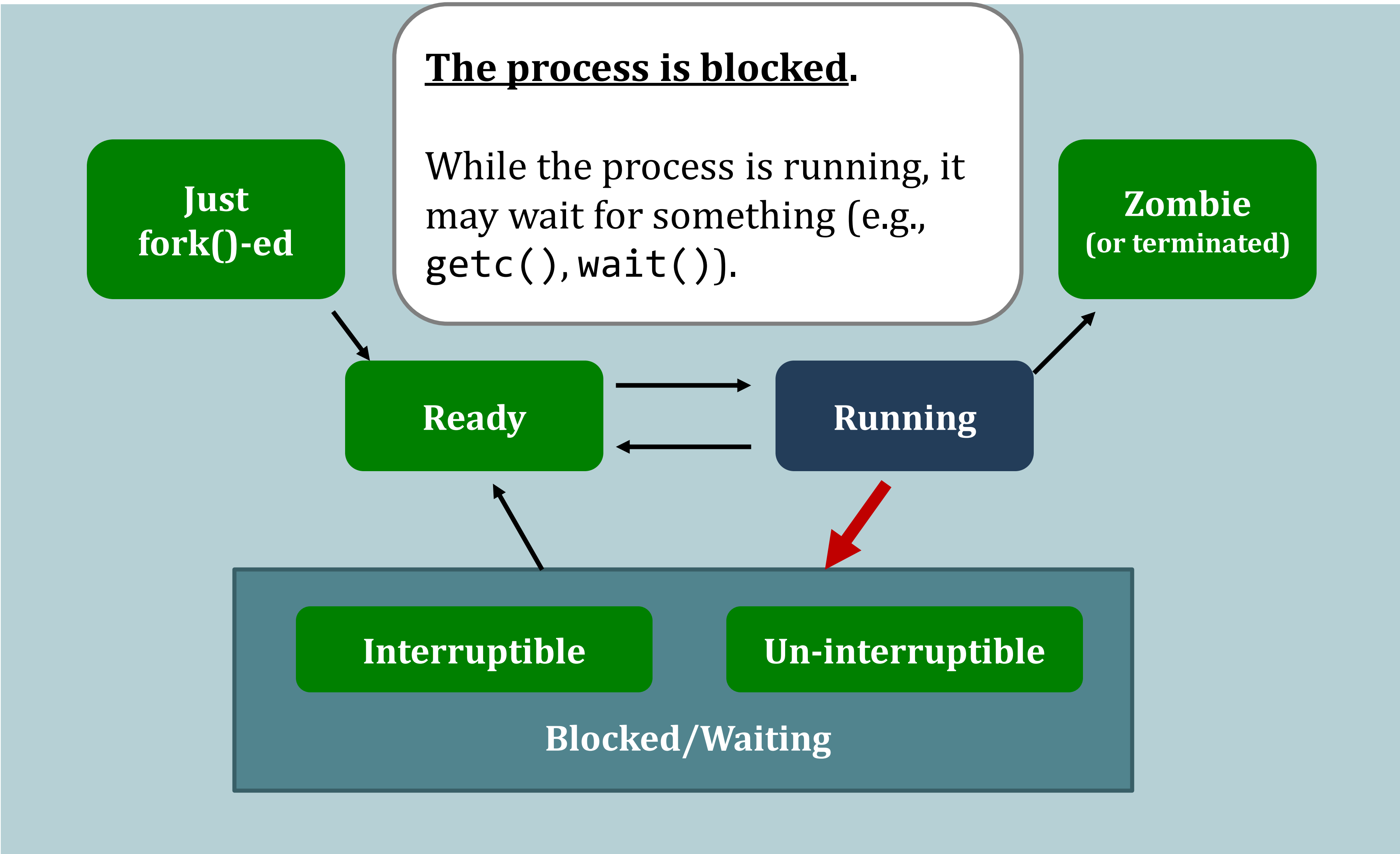
<u>**The process is ready**</u>.
It means it is ***ready to run* but is not running**.

A process may become "<u>**ready**</u>" (*runnable*) after...
- it is just created by **`fork()`**;
- it has been running on the CPU for some time and the OS chooses another process to run (scheduled context switch)
- returning from blocked states.

Process lifecycle - Running



Process lifecycle - Blocking

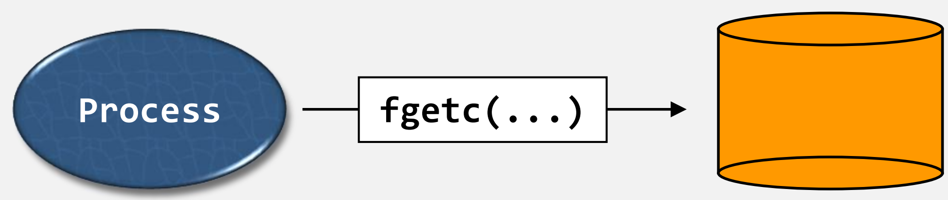


Process lifecycle – Interruptible wait

Example. **Reading a file**.

Sometimes, the process has to wait for the response from the device and, therefore, it is **blocked**

- this blocking state is **interruptible**
  - E.g., "**Ctrl + C**" can get the process out of the waiting state (but goes to termination state instead).



Process lifecycle – Un-Interruptible wait

Sometimes, a process needs to wait for a resource until it really gets what it wants
- Doesn't want to be "Ctrl-C" interruptible
- **Un-interruptible** status
  - No way to signal it to wake up unless it returns itself
  - Check online! The only solution is …

Who set this?
- E.g., syscall call (http://man7.org/linux/man-pages/man2/delete_module.2.html)

Why set this?
- Easier programming for lazy programmer (e.g., a driver program for a DVD drive)
- The programmer "thinks" the wait is very short and robust
  - This is one the top reasons that hang your machine / process today!
- …

Interruptible **Un-interruptible**

Blocked/Waiting

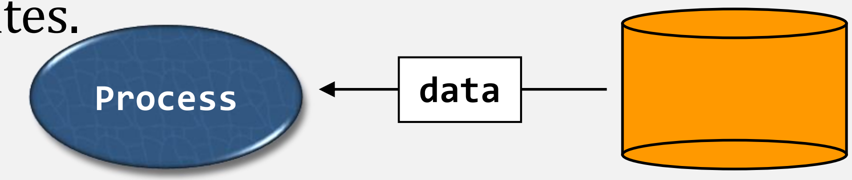
http://unix.stackexchange.com/questions/96797/what-does-the-interruptible-sleep-state-indicate

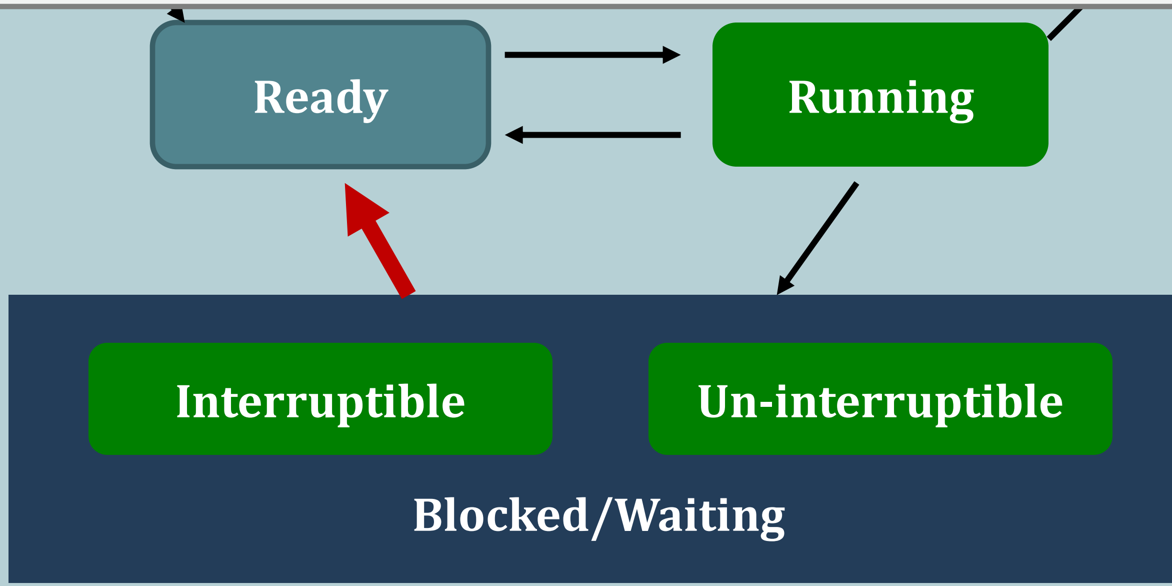
http://stackoverflow.com/questions/767551/how-to-stop-uninterruptible-process-on-linux

Process lifecycle

Return back to ready.

When response arrives, the status of the process changes back to **Ready**. from any one of the blocked states.

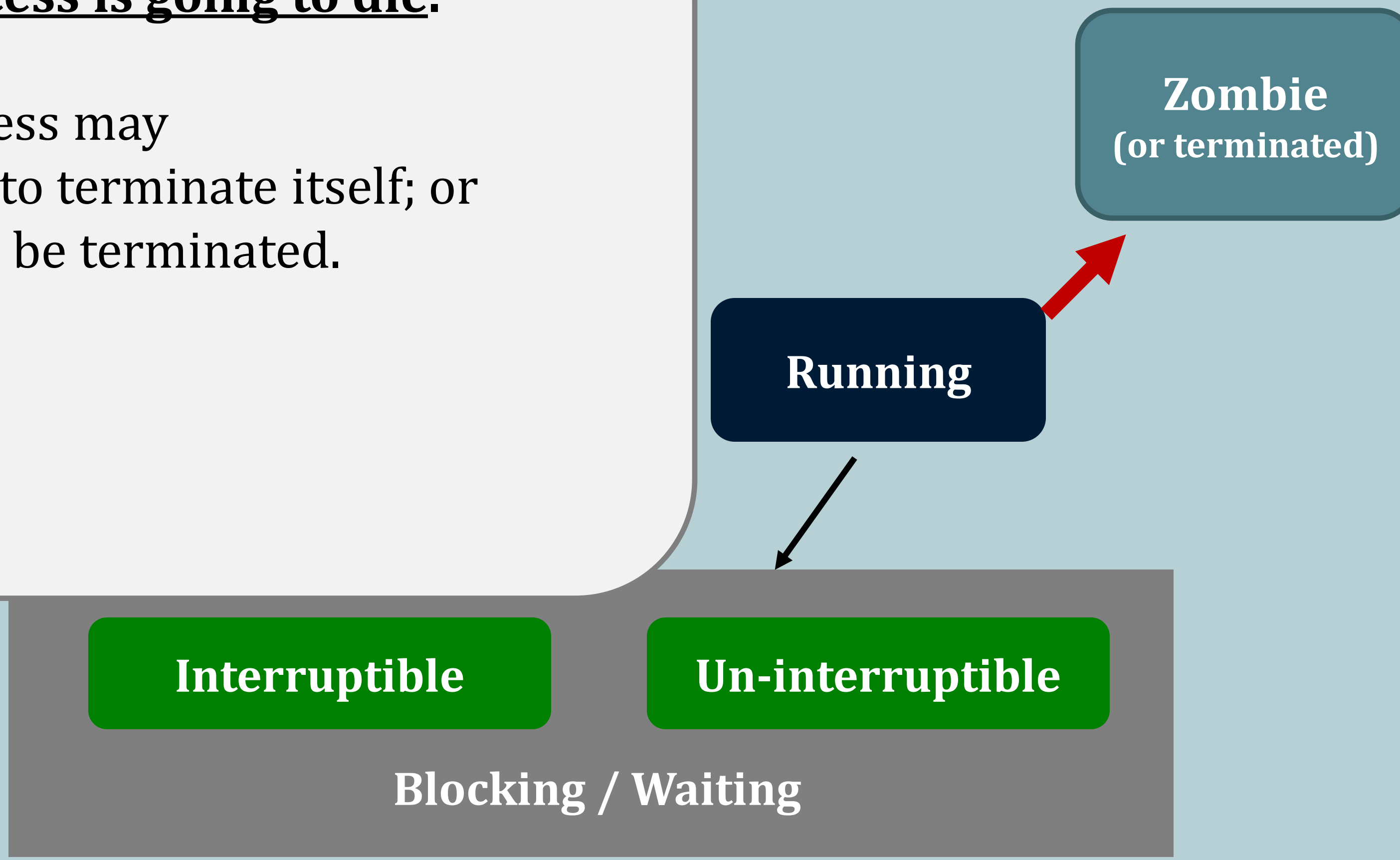




Process lifecycle

<u>The process is going to die</u>.

The process may
- choose to terminate itself; or
- force to be terminated.



Thank You!