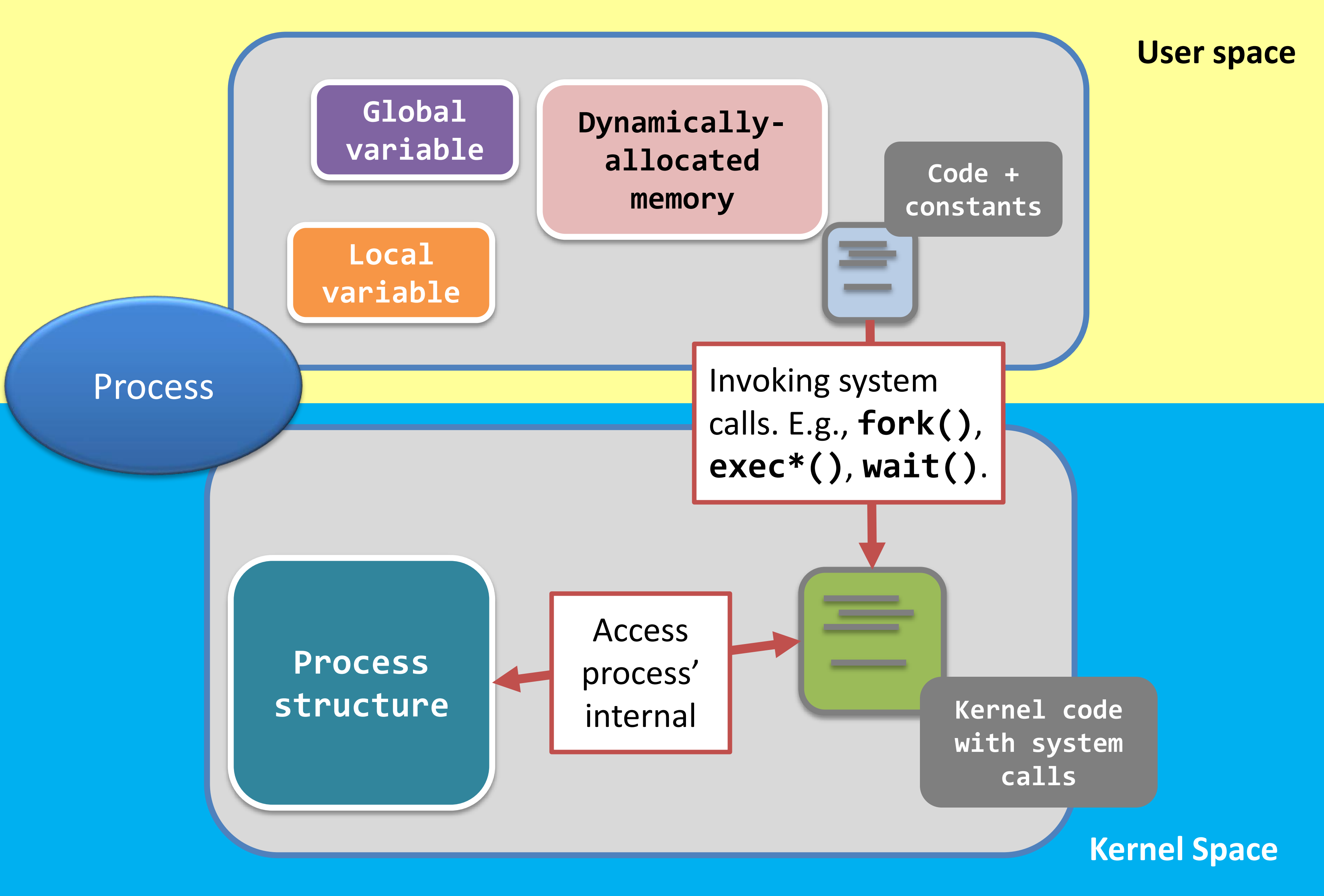
3150 - Operating Systems

Dr. WONG Tsz Yeung

Chapter 2, part 2 - Process and Kernel

- This boring part lets you know the working in the kernel.

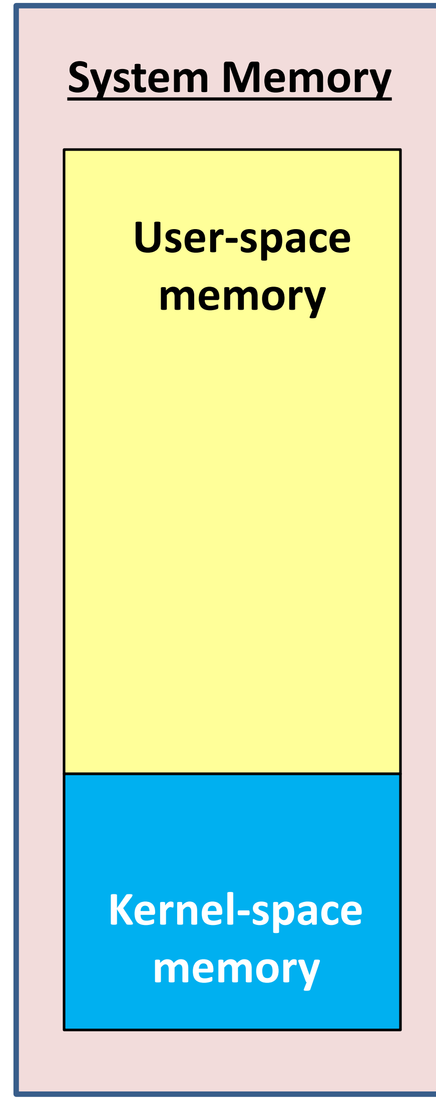
The story so far…



Kernel-space VS User-space

	Kernel-space memory	User-space memory
Storing what	a. Kernel data structure. b. Kernel code. c. Device drivers.	a. Process' memory. b. Program code of the process.
Accessed by whom	Kernel code.	User program code + kernel code.

See? The kernel is invincible!

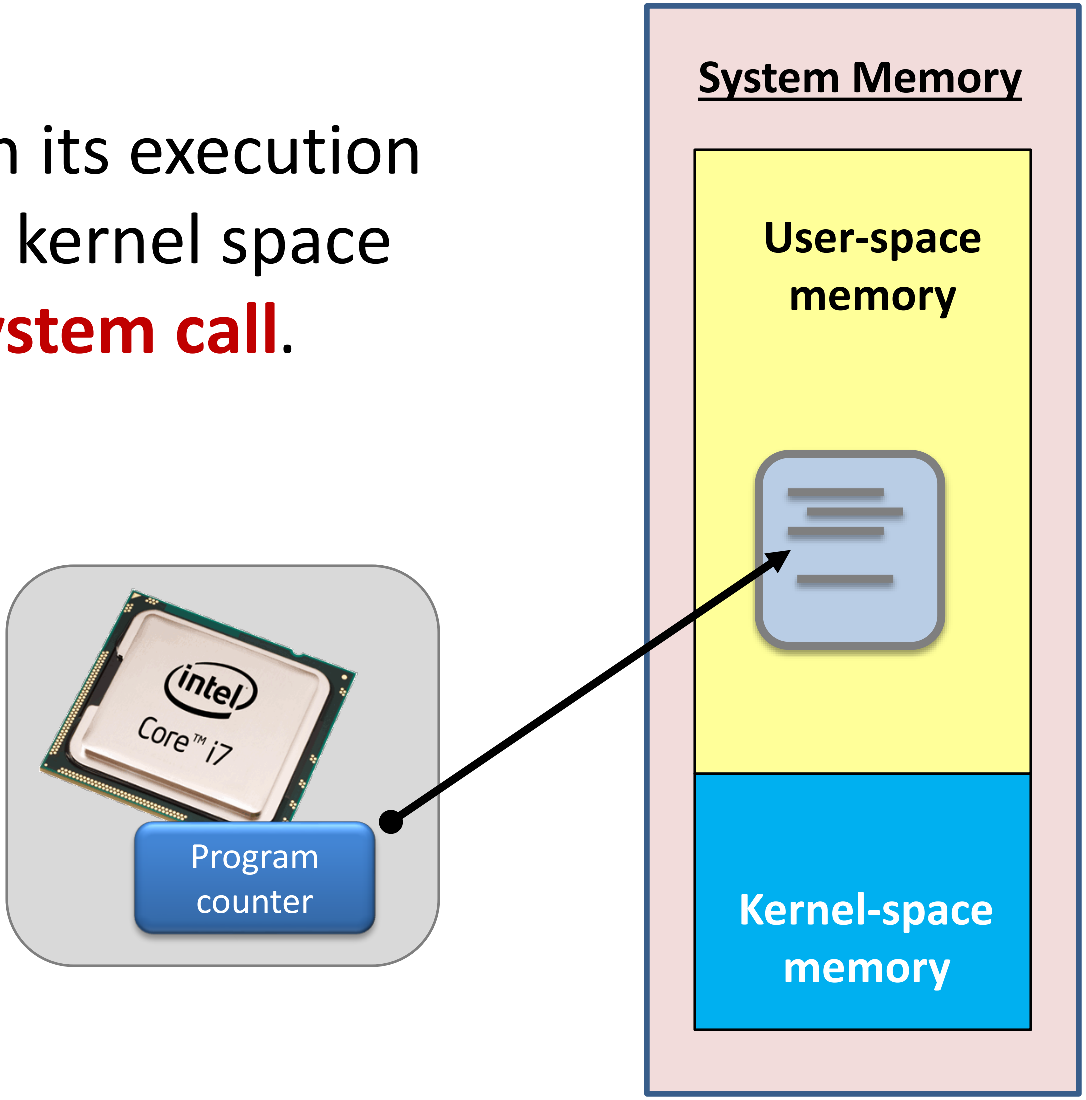


Process is going back and forth...

- We'd better say:
  - A process will switch its execution from using-space to kernel space **through invoking system call**.

Say, the CPU is running a program code of a process.

As the code is in user-space memory, so the program counter is pointing to that region.

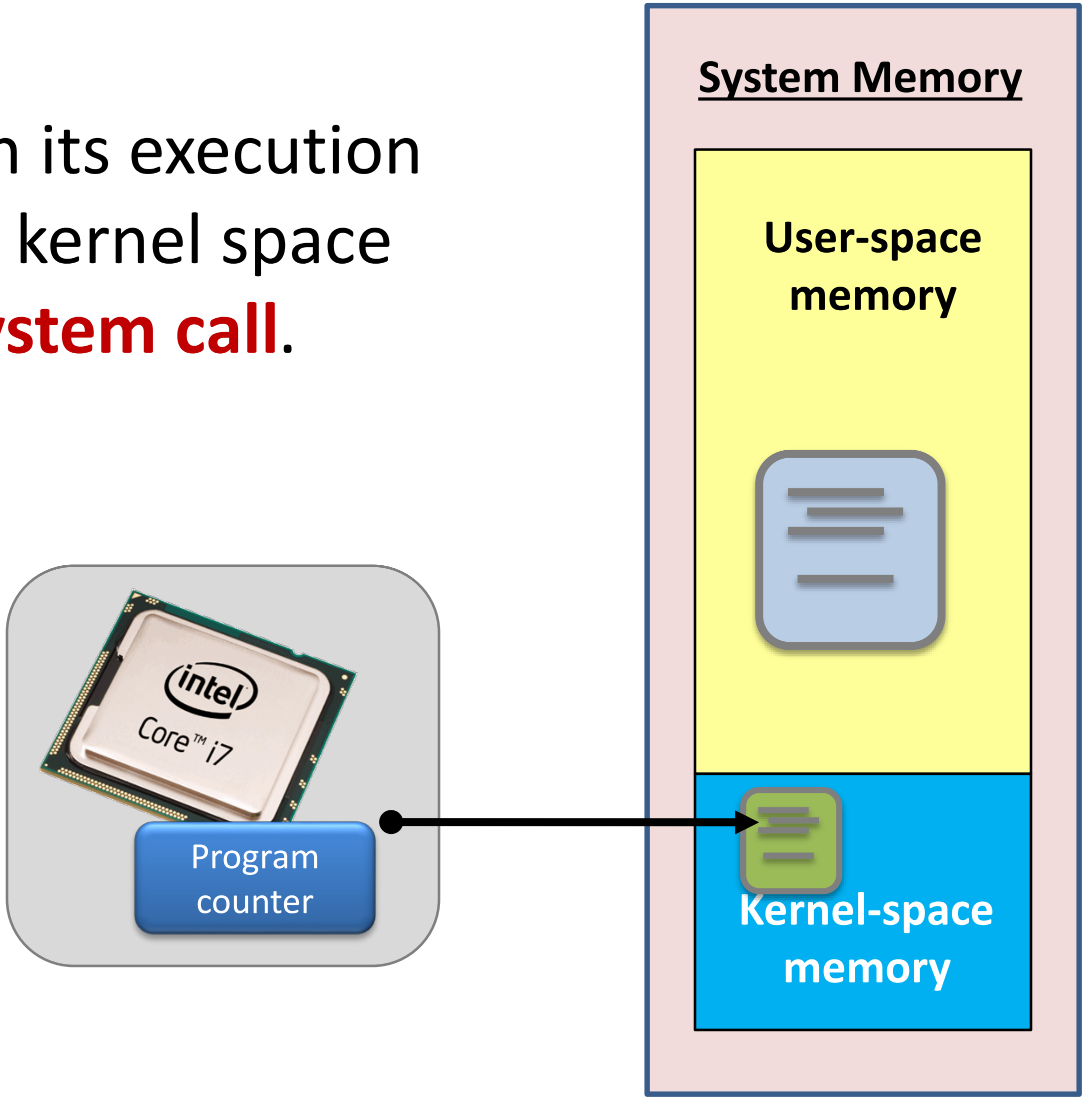


Process is going back and forth…

- We’d better say:
  - A process will switch its execution from using-space to kernel space **through invoking system call**.

When the process is calling the system call “**`getpid()`**”.

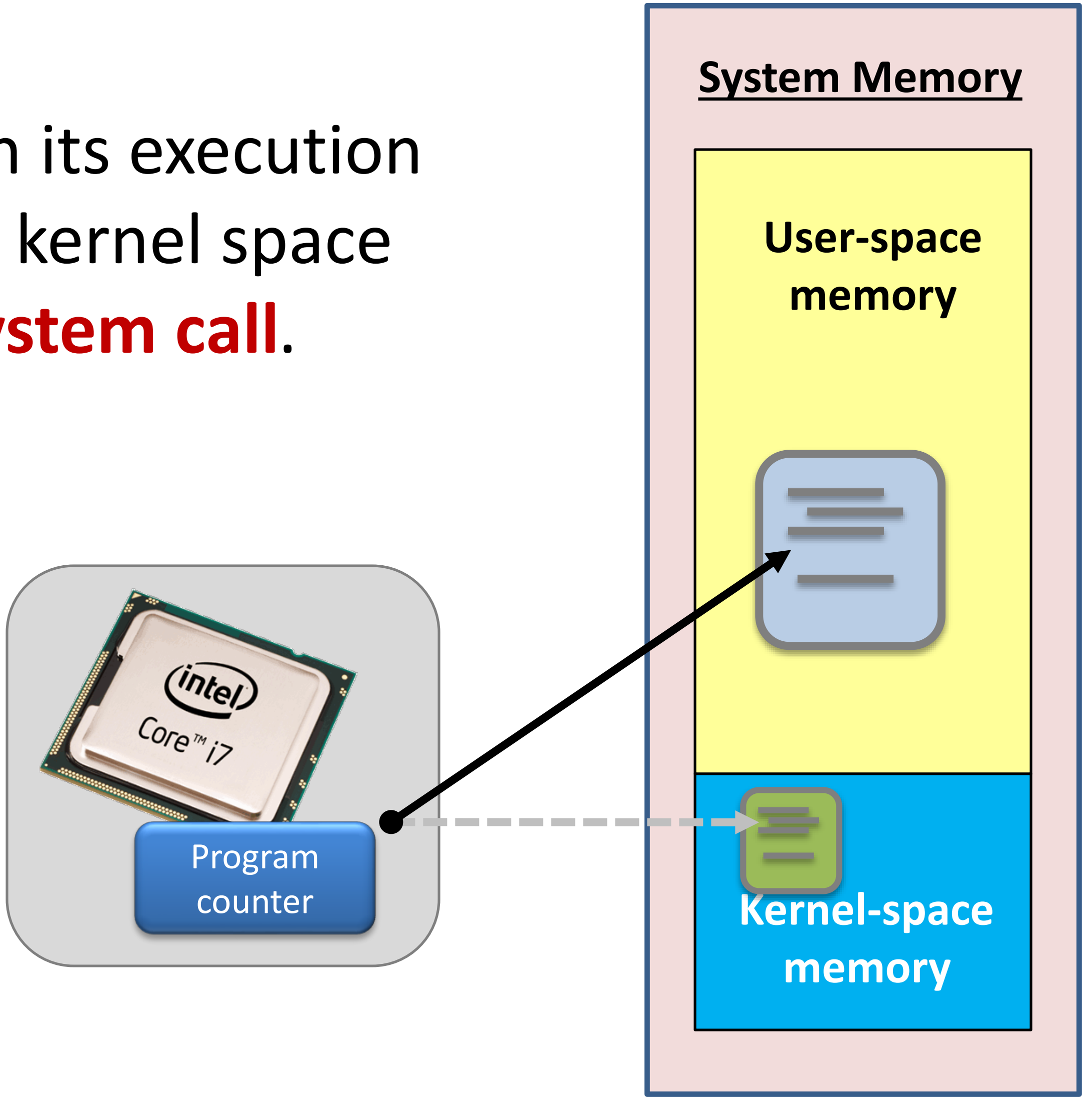
Then, the CPU switches <u>from the user-space to the kernel-space</u>, and reads the PID of the process from the kernel.



Process is going back and forth…

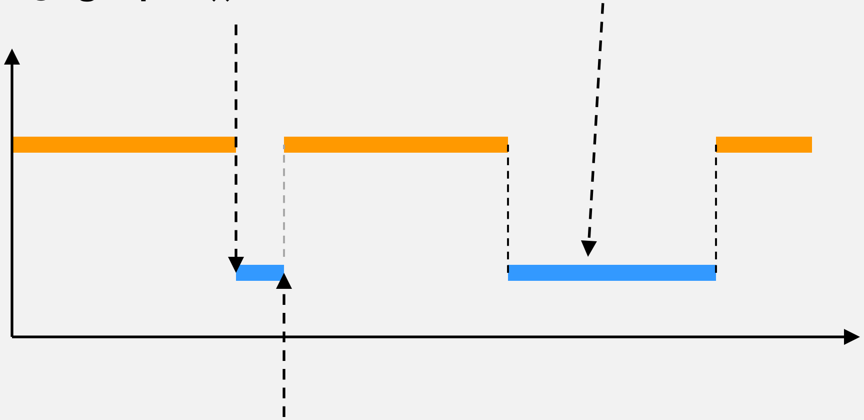
- We'd better say:
  - A process will switch its execution from using-space to kernel space **through invoking system call**.

At last, 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.



User time VS System time

- So, not just the memory, but also the **execution of a process** is also divided into two parts.



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

System time –
Time spent on codes in kernel-space memory.

`Total running time = user time + system time.`

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
$ _
```

Time elapsed when "**./time_example**" terminates.

The user time of "**./time_example**" measured when the process is on CPU.

The system time of "**./time_example**" measured when the process is on CPU.

```
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.

```
[example@3150]$ cat time_example.c
```

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.

```
[example@3150]$ cat time_example.c
```

User time VS System time – example 2

- The user time and the system **time together define the performance of an application**.
  - When writing a program, you must consider both the user time and the system 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;
}
```

```
[example@3150]$ cat printf_example_{slow,fast}.c
```

User time VS System 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
$ _
```

```
[example@3150]$ cat printf_example_{slow,fast}.c
```

User time VS System time – example 2

We will explain the difference later this chapter.

For those who are impatient, I'd say:
we could do nothing about that…

```
$ time ./time_example_slow

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

```
$ time ./time_example_fast

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

```
[example@3150]$ cat printf_example_{slow,fast}.c
```

User time VS System time – short summary

- System call plays a major role in performance.
  - **Blocking system call**: some system calls even **stop your process** until the data is available.

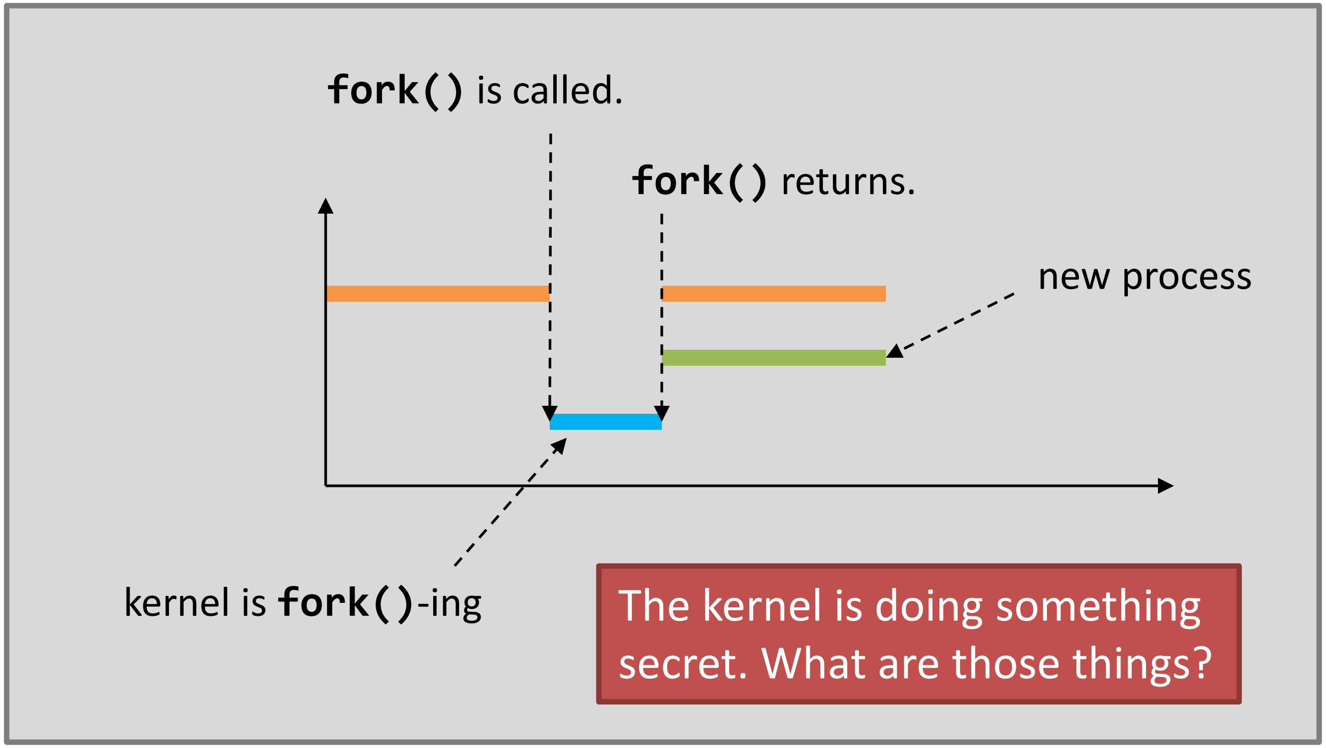
Don’ts	Do’s
Reading a file byte-by-byte.	Reading a file block-by-block, where the size of a block is 4,096 bytes.
A pair of `malloc()` and `free()` on piece of fixed, 1,000-byte memory.	Declare a `char` array of 1,000 bytes.

Can you think of (or you’ve experienced) more examples?

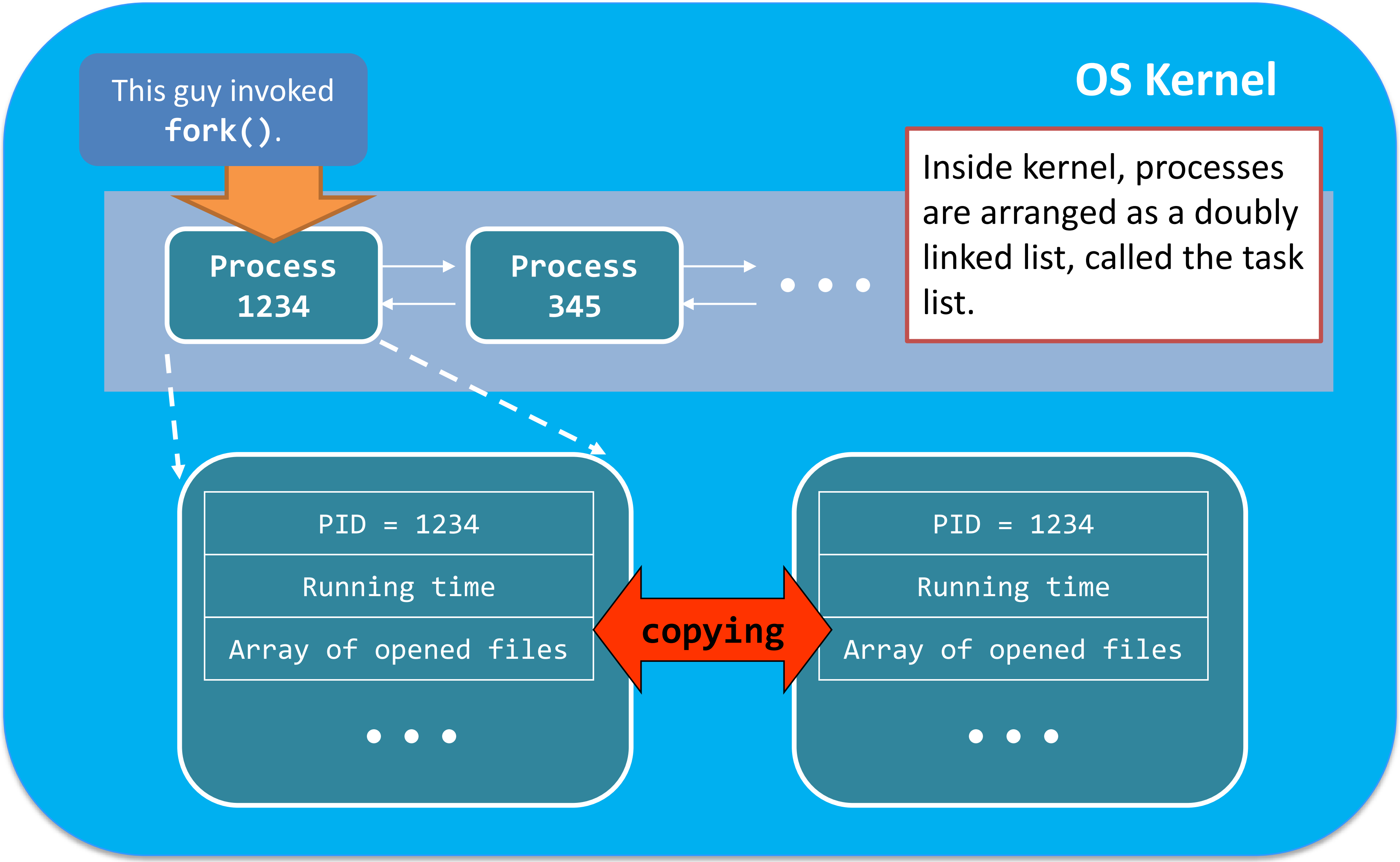
Working of system calls
- `fork();`

`fork()` that you've learnt…

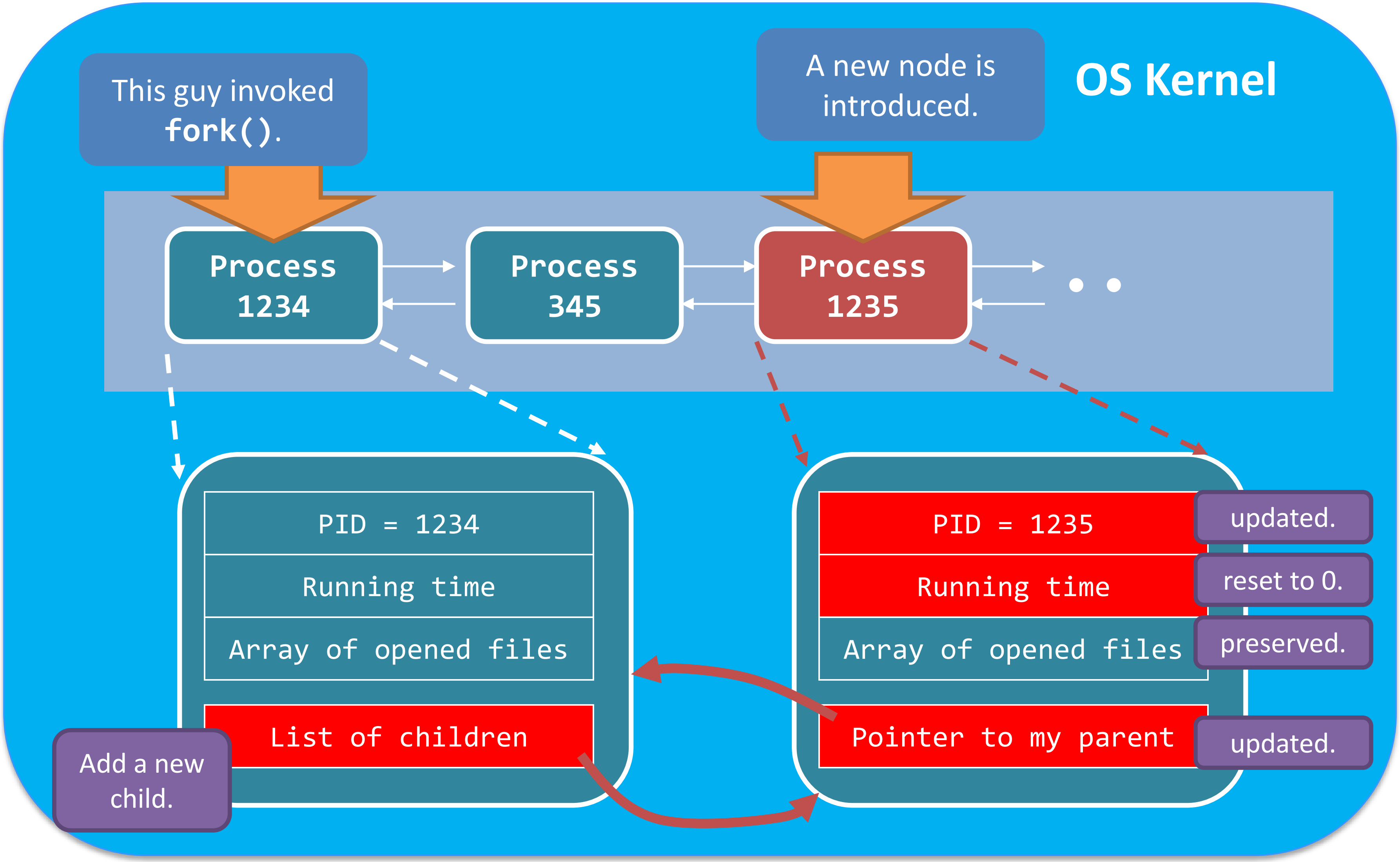
- From a programmer's view, **`fork()`** behaves like the following:



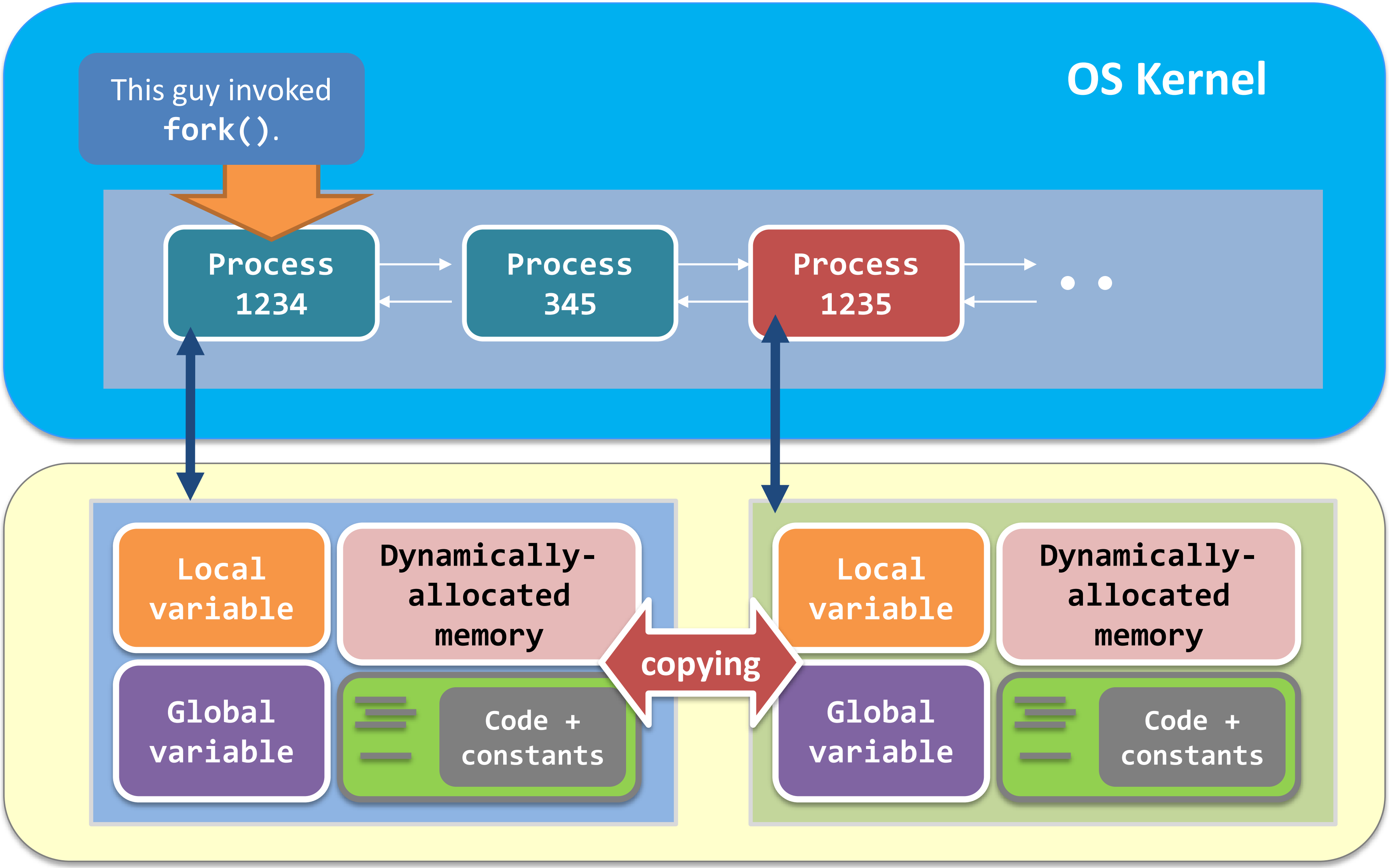
`fork()` in action – the start…



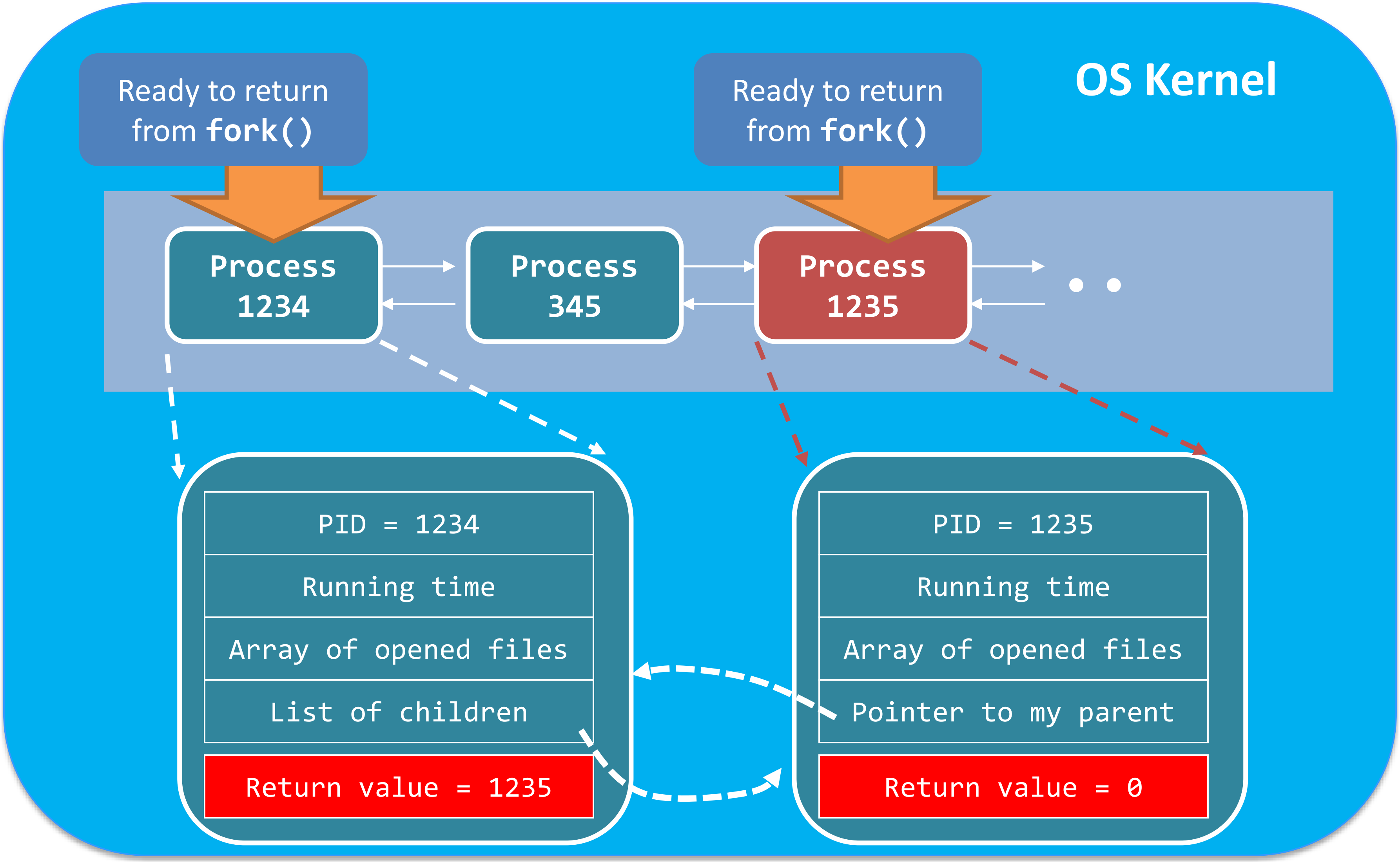
`fork()` in action – kernel-space update



fork() in action – user-space update



fork() in action – finish



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

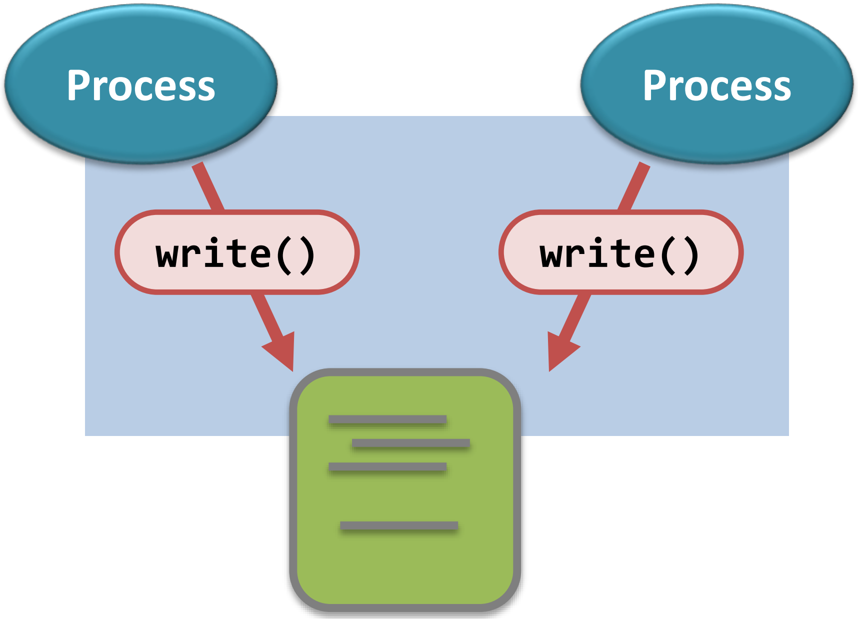
- Array of opened files contains:

Array Index	Description
0	Standard Input Stream; **`FILE *stdin;`** *int x; fscanf(stdin, %d, &x);*
1	Standard Output Stream; **`FILE *stdout;`** *fprintf(stdout, “hello world”); = printf(“hello world”);*
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!

`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!

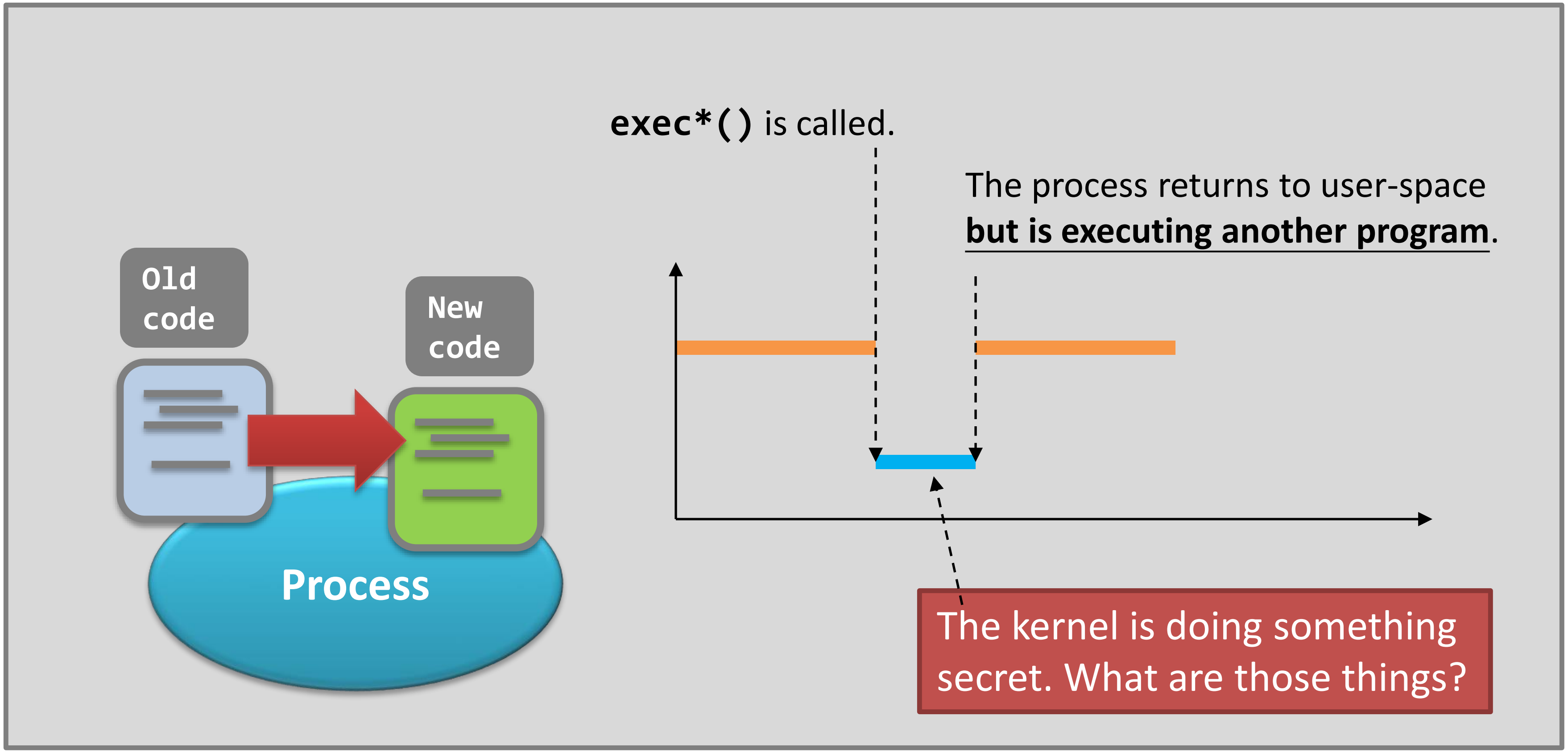
`[example@3150] share_opened_file.c`

Working of system calls

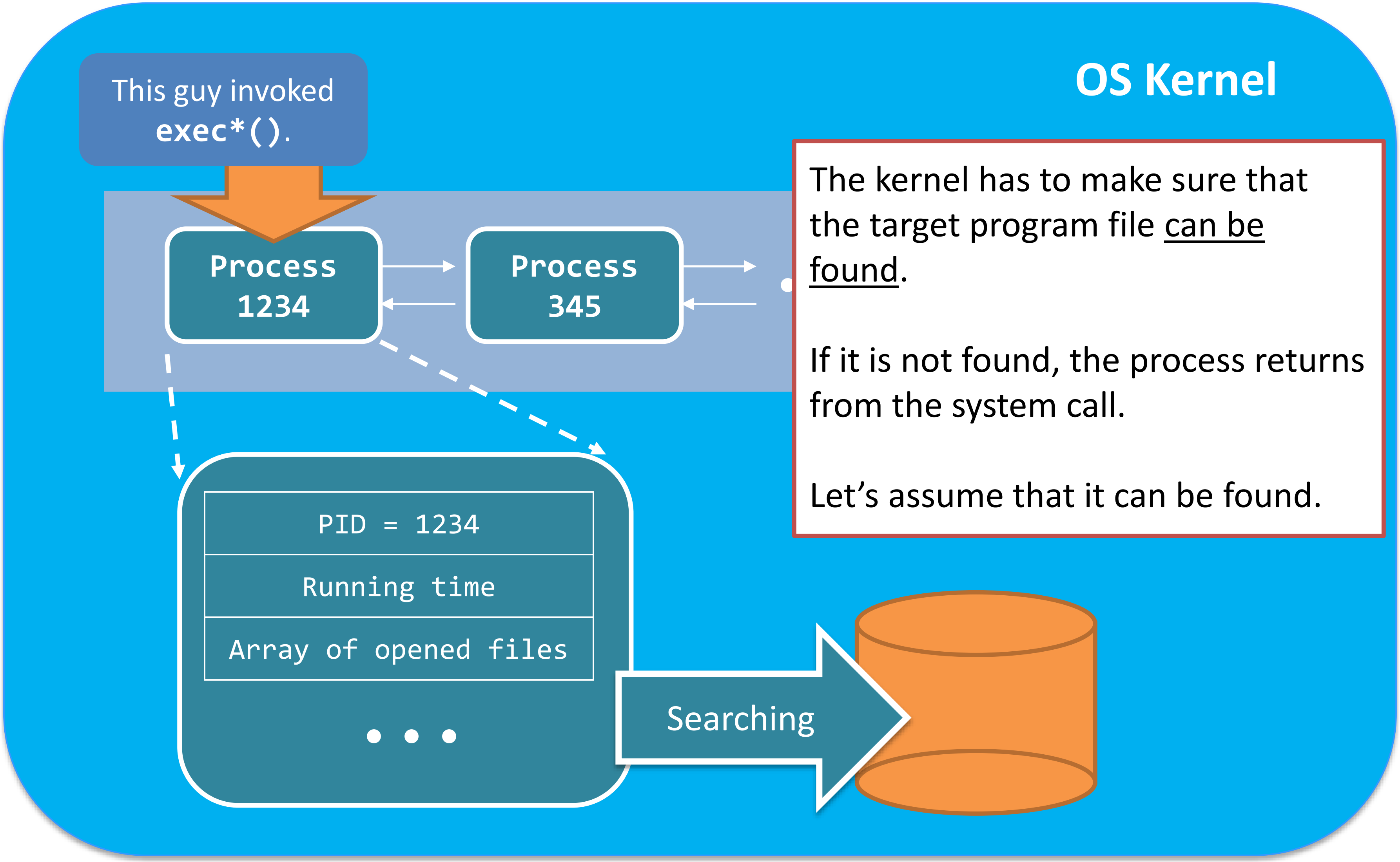
- `fork();`
- `exec*();`

exec*() that you've learnt...

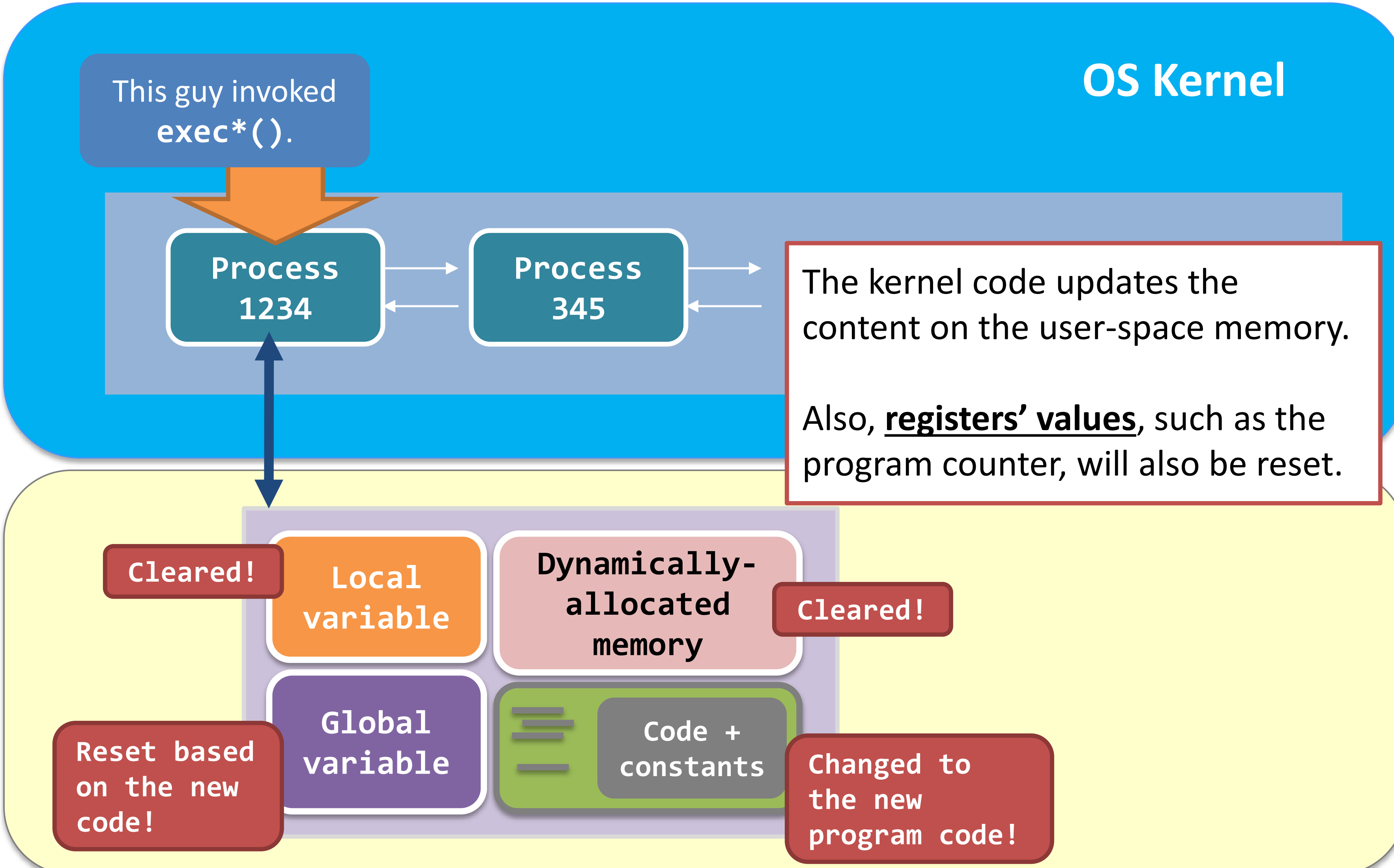
- How about the **exec*()** call family?



exec*() in action – the start…



exec*() in action – the end

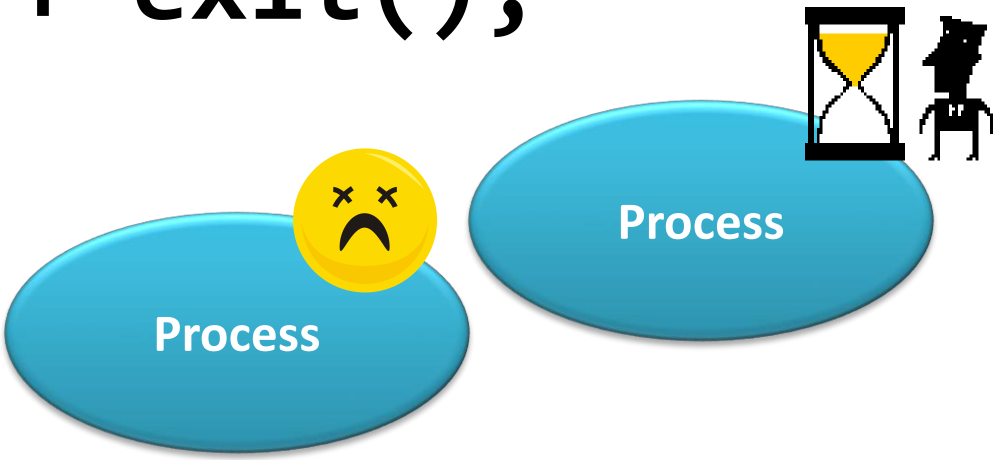


The kernel code updates the content on the user-space memory.

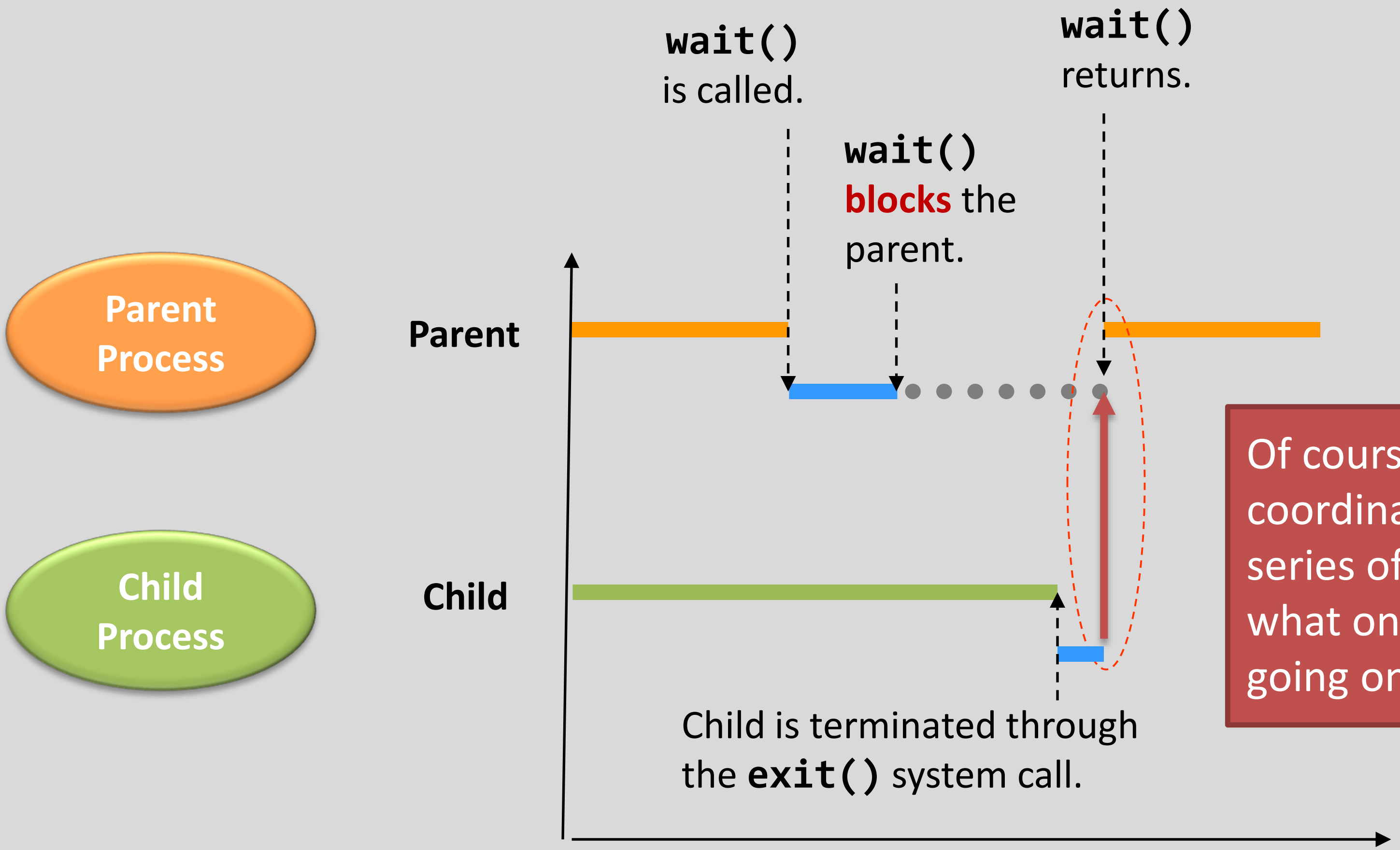
Also, **<u>registers' values</u>**, such as the program counter, will also be reset.

Working of system calls

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

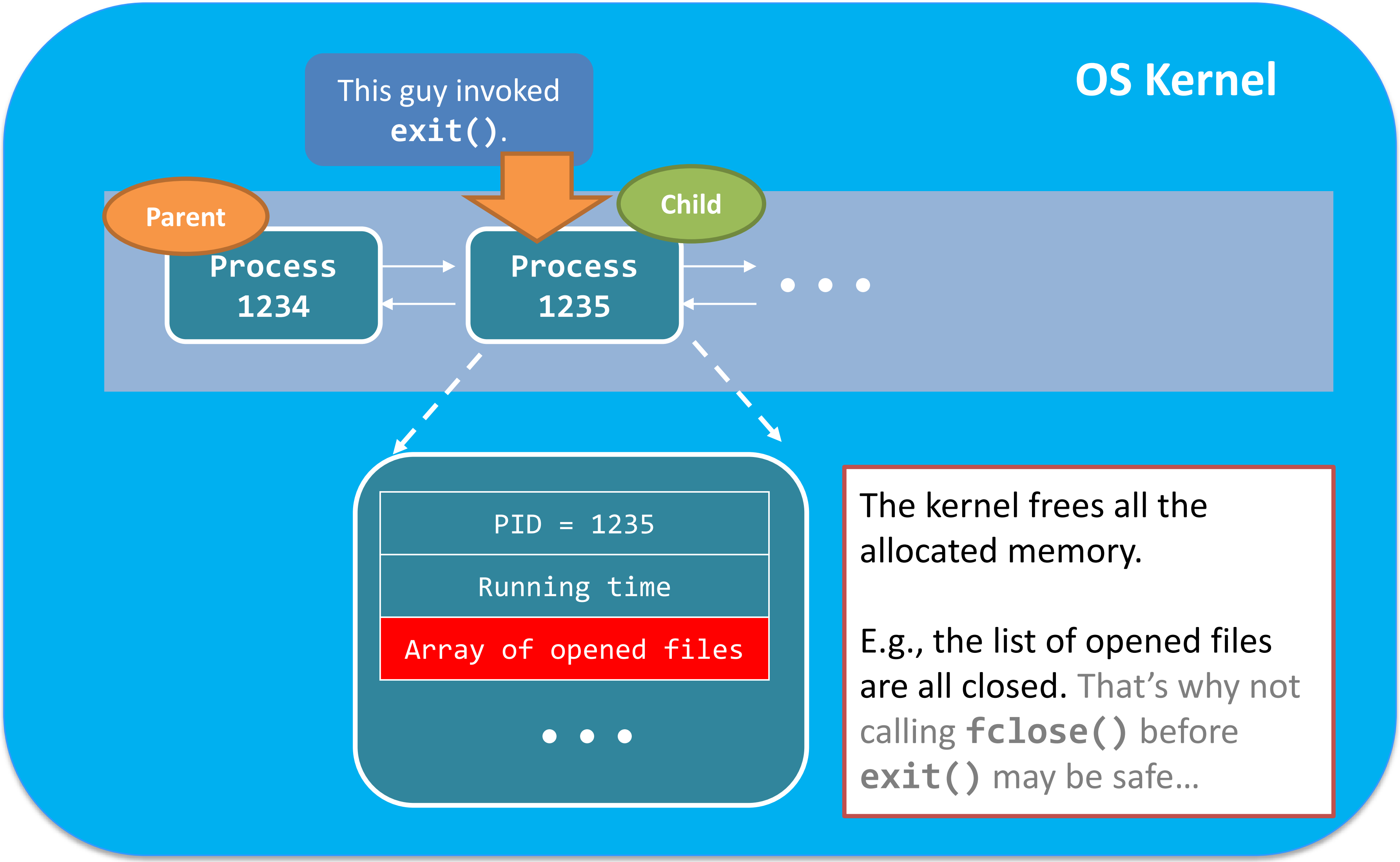


`wait()` and `exit()` – they come together!



Of course, the kernel coordinates the series of events. But, what on earth is going on?

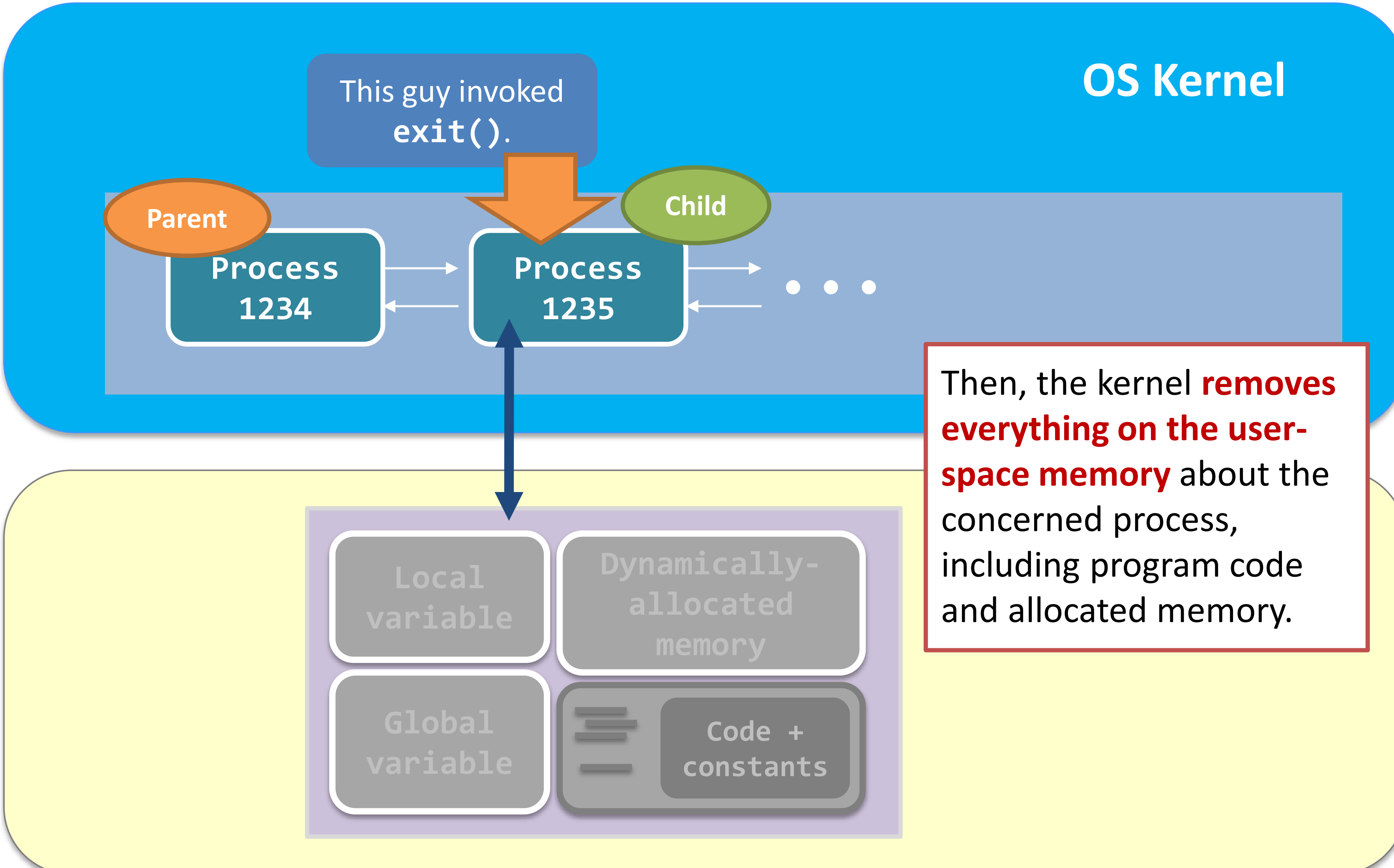
`wait()` and `exit()` – child side



The kernel frees all the allocated memory.

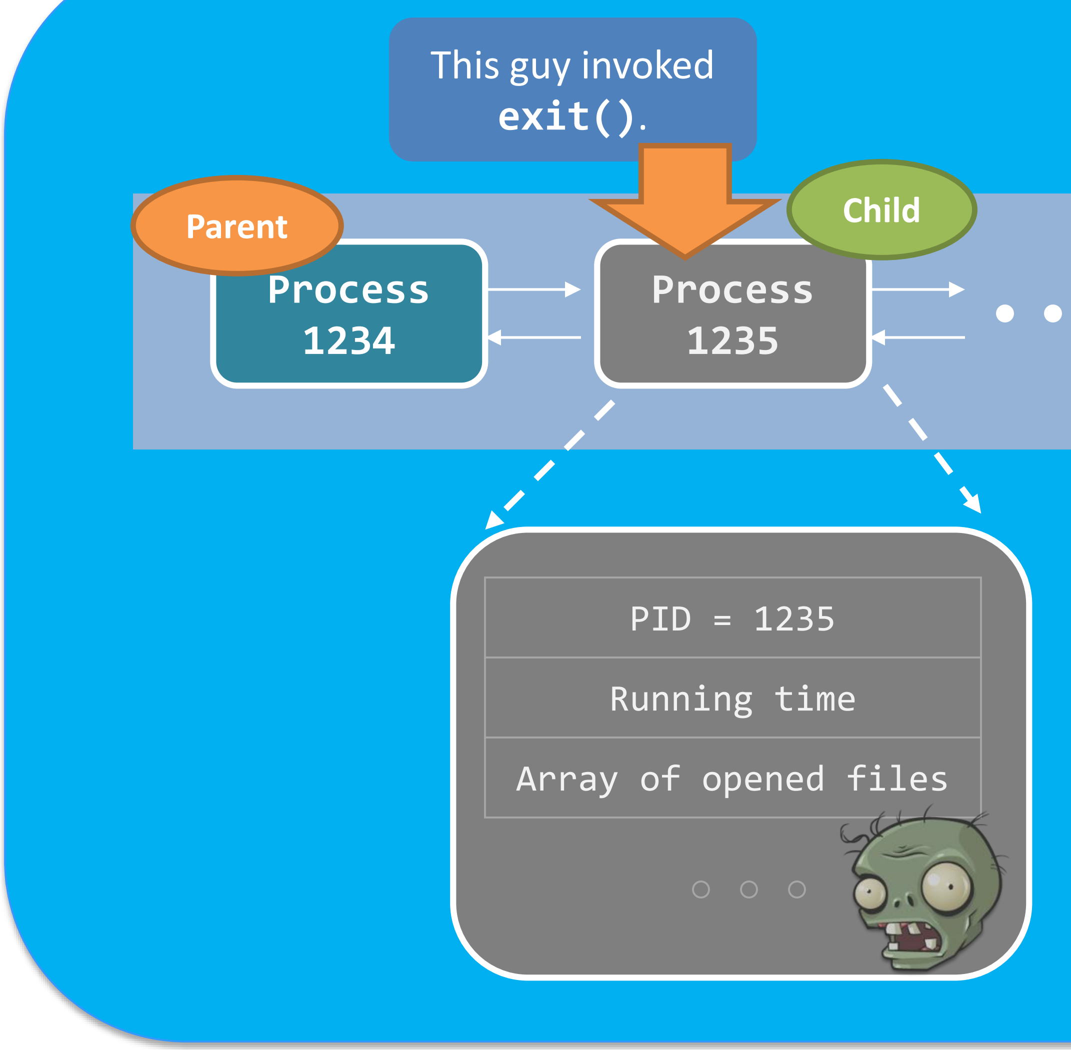
E.g., the list of opened files are all closed. That's why not calling **`fclose()`** before **`exit()`** may be safe…

`wait()` and `exit()` – child side



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

`wait()` and `exit()` – child side

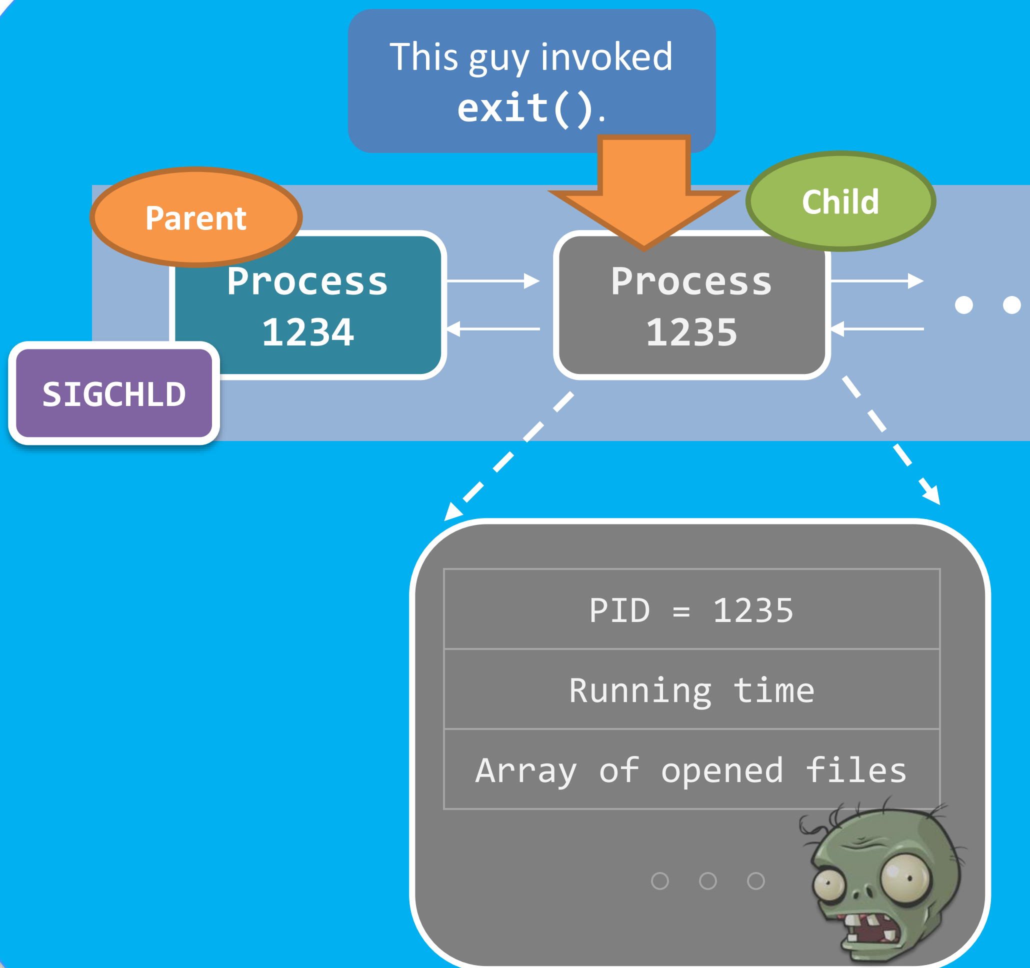


For the child, its storage in the kernel-space memory is kept to a minimum…

Nevertheless, the **PID** (1235 in this example) and the **process structure** is owned by the child.

The status of the child is now called **zombie**.

`wait()` and `exit()` – child side



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

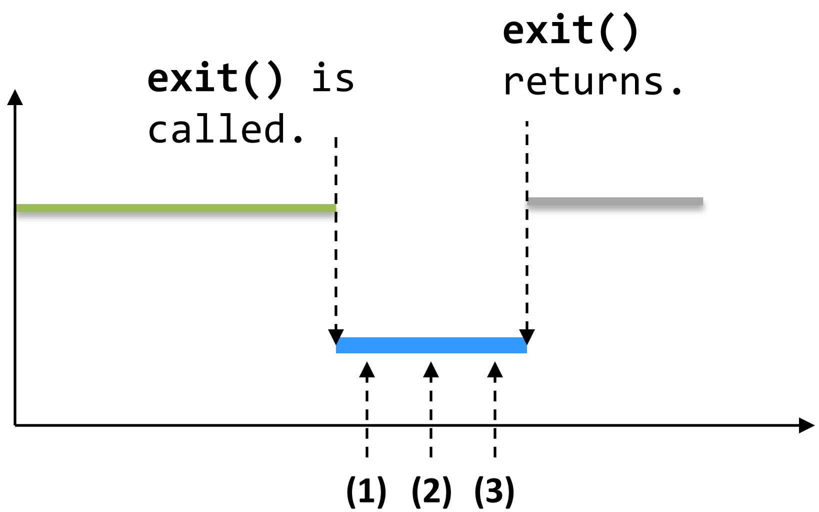
The notification is a **signal** called **SIGCHLD** (pronounced as **sig-child**).

A short summary for **`exit()`**

Step (1) Clean up most of the allocated kernel-space memory.

Step (2) Clean up all user-space memory.

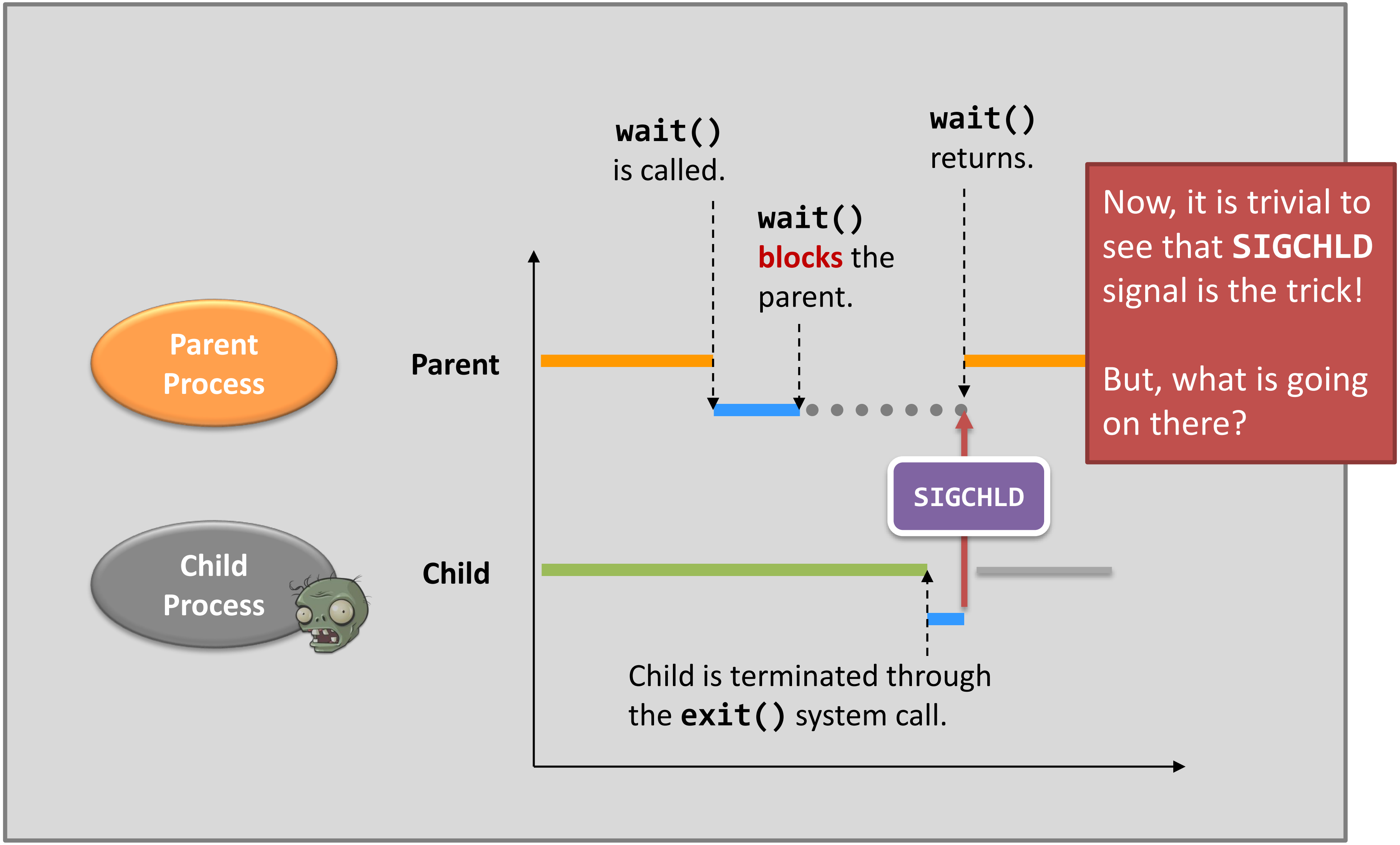
Step (3) Notify the parent with SIGCHLD.



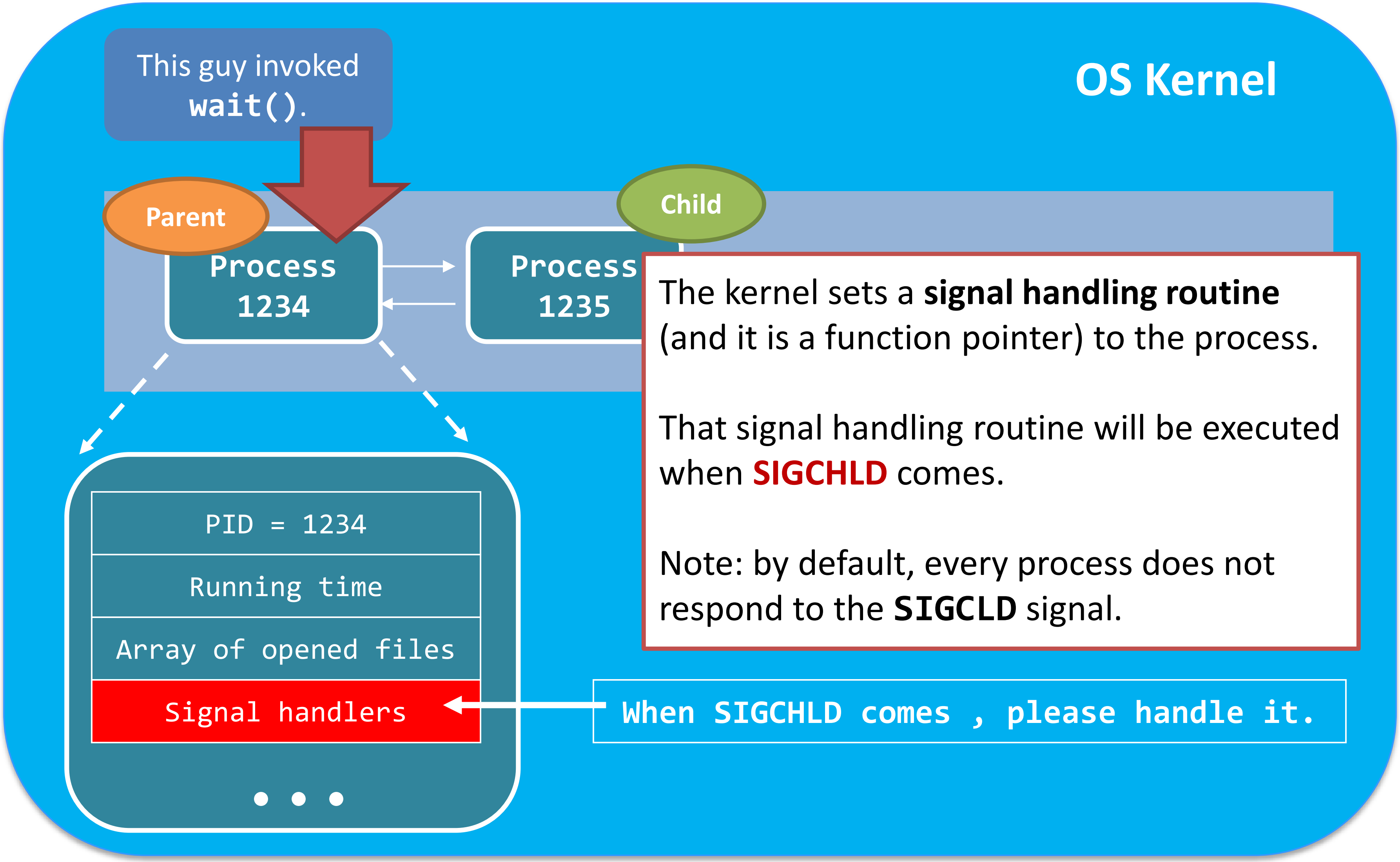
Although the child is still in the system, it is no **longer running**. There is no program code, how come it is running?!

It turns into a **mindless zombie**...

`wait()` and `exit()` – they come together!



`wait()` and `exit()` – parent side

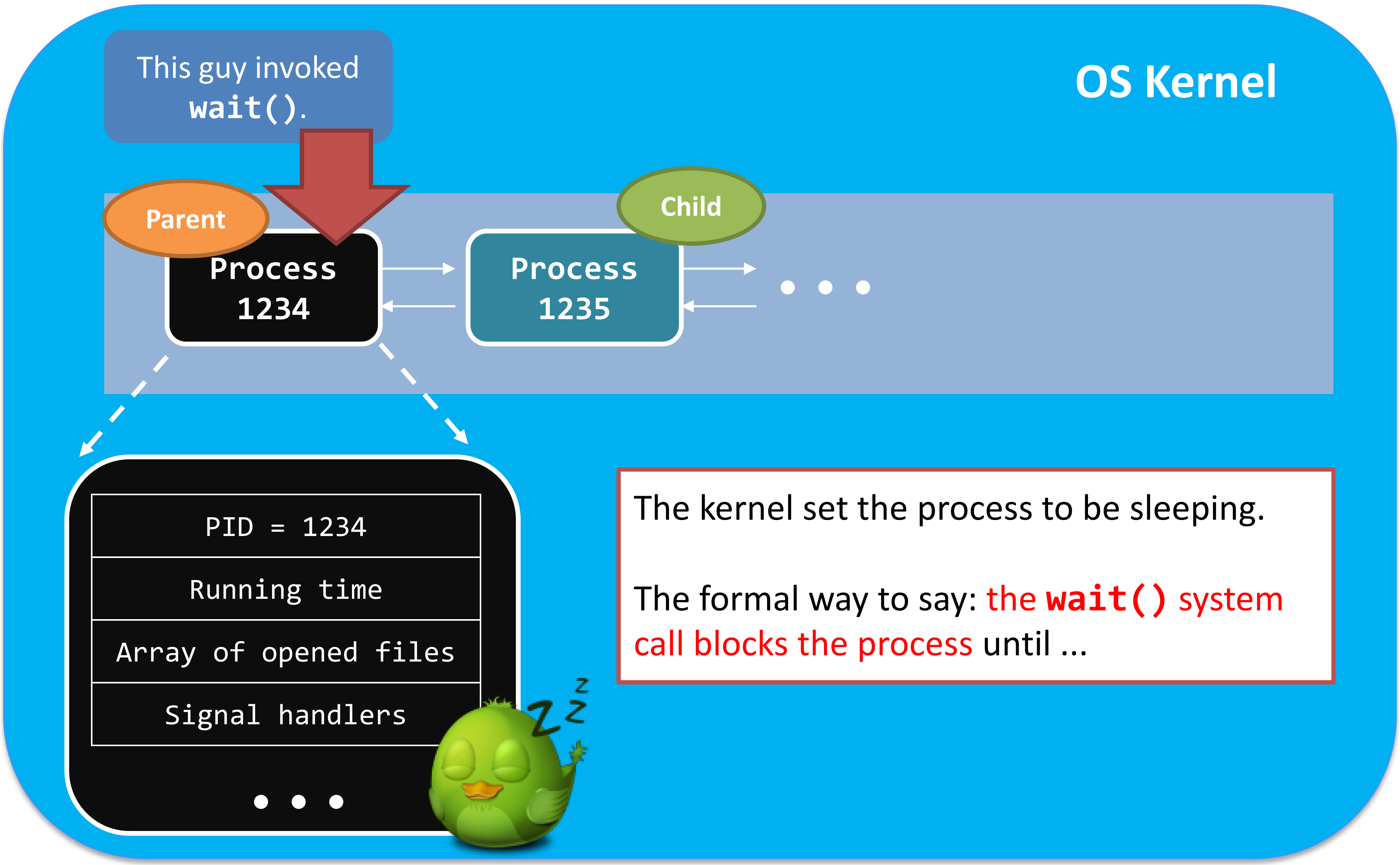


The kernel sets a **signal handling routine** (and it is a function pointer) to the process.

That signal handling routine will be executed when **SIGCHLD** comes.

Note: by default, every process does not respond to the **`SIGCLD`** signal.

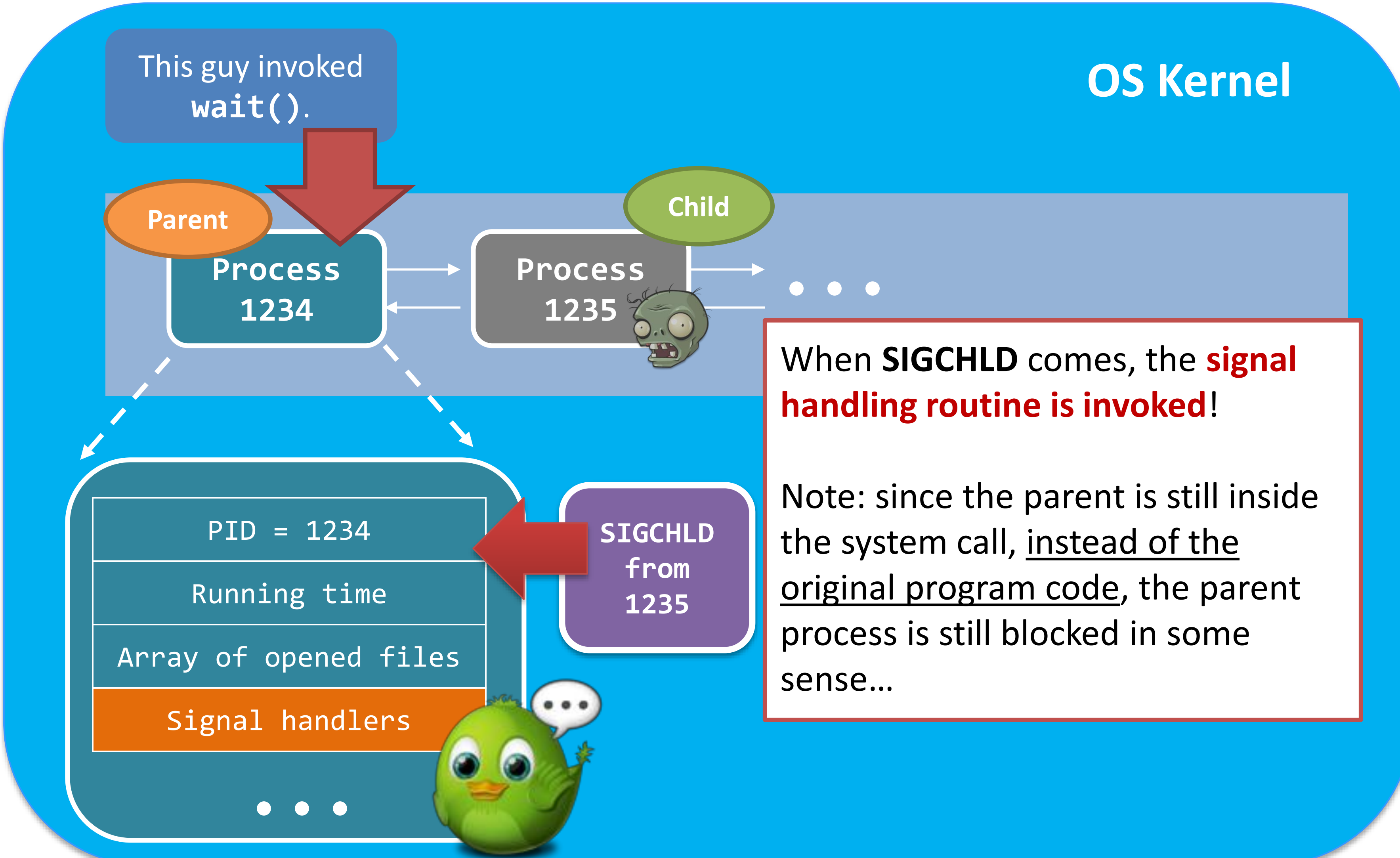
`wait()` and `exit()` – parent side



The kernel set the process to be sleeping.

The formal way to say: the **`wait()`** system call blocks the process until ...

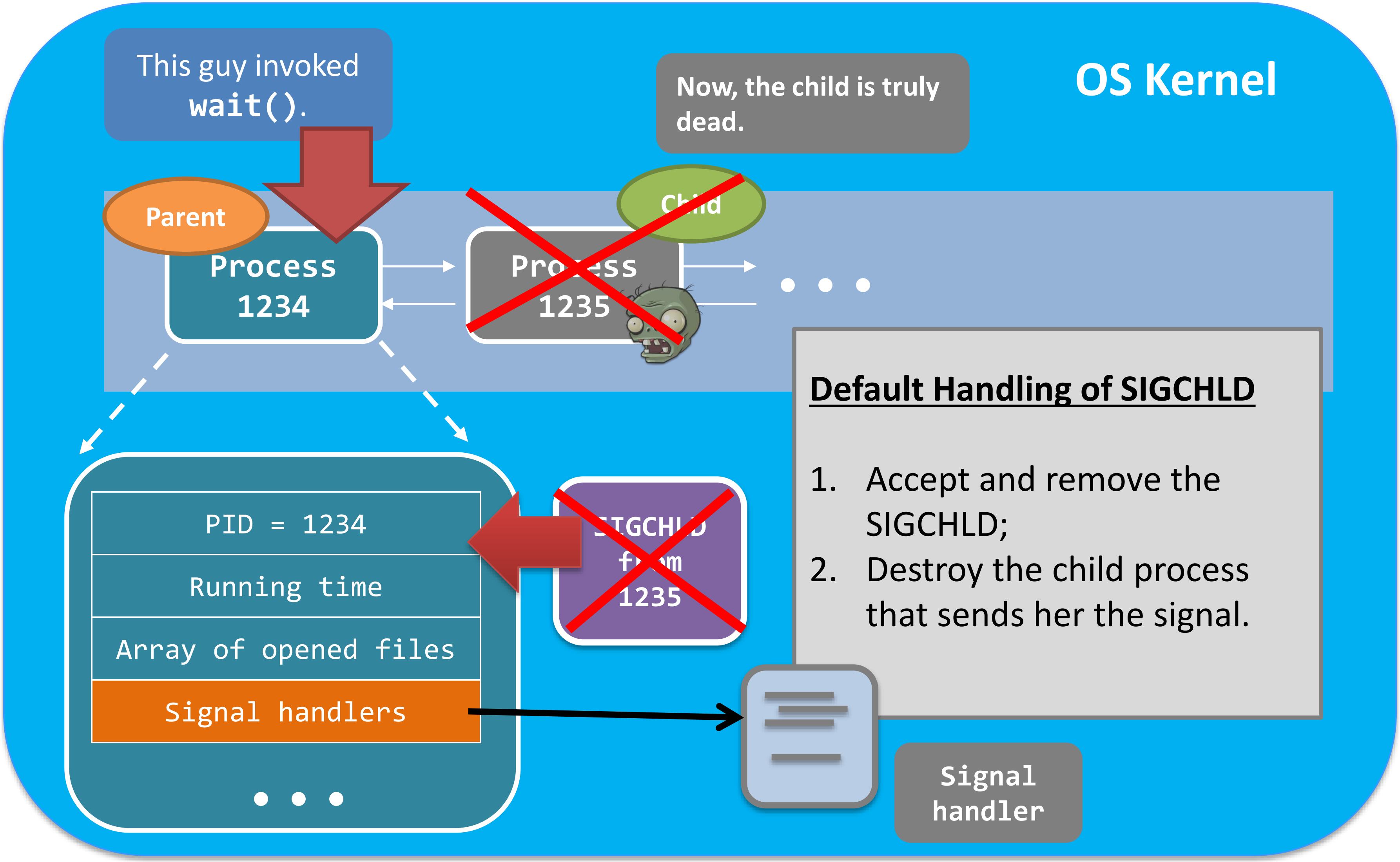
`wait()` and **`exit()`** – parent side



When **SIGCHLD** comes, the **signal handling routine is invoked**!

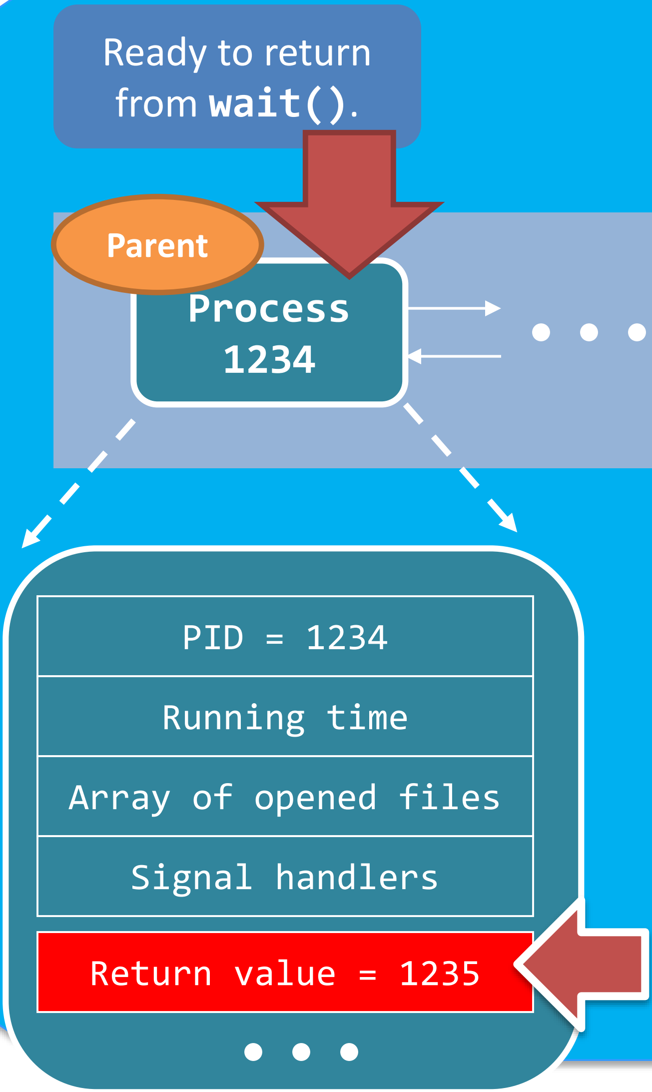
Note: since the parent is still inside the system call, <u>instead of the original program code</u>, the parent process is still blocked in some sense…

wait() and exit() – parent side



`wait()` and `exit()` – parent side

OS Kernel



The signal handler is then removed, i.e., the process is ignoring **SIGCHLD** again.

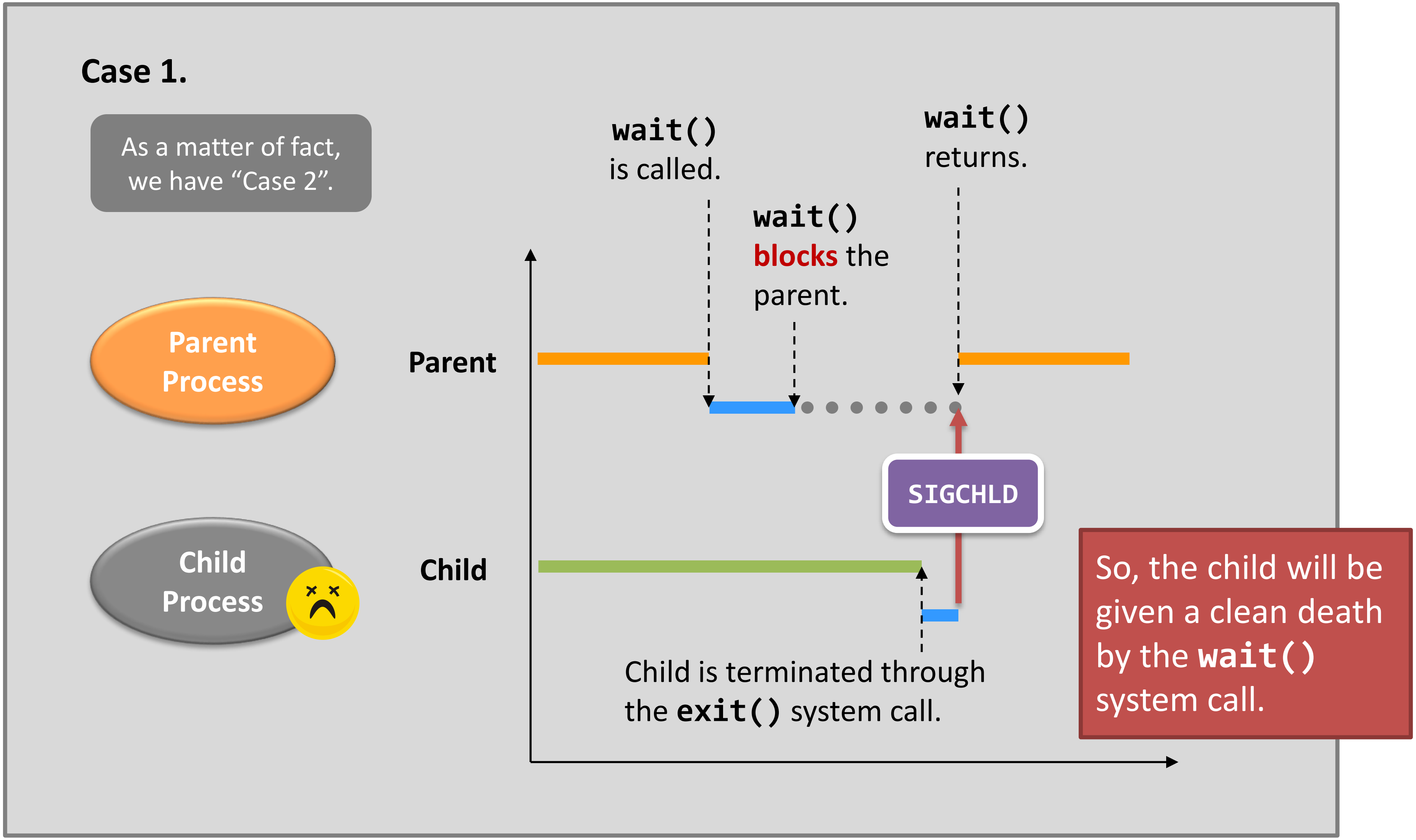
It returns to the previously-executing code, going back to the user space.

So, it looks like "**wait()** is returned from its invocation".

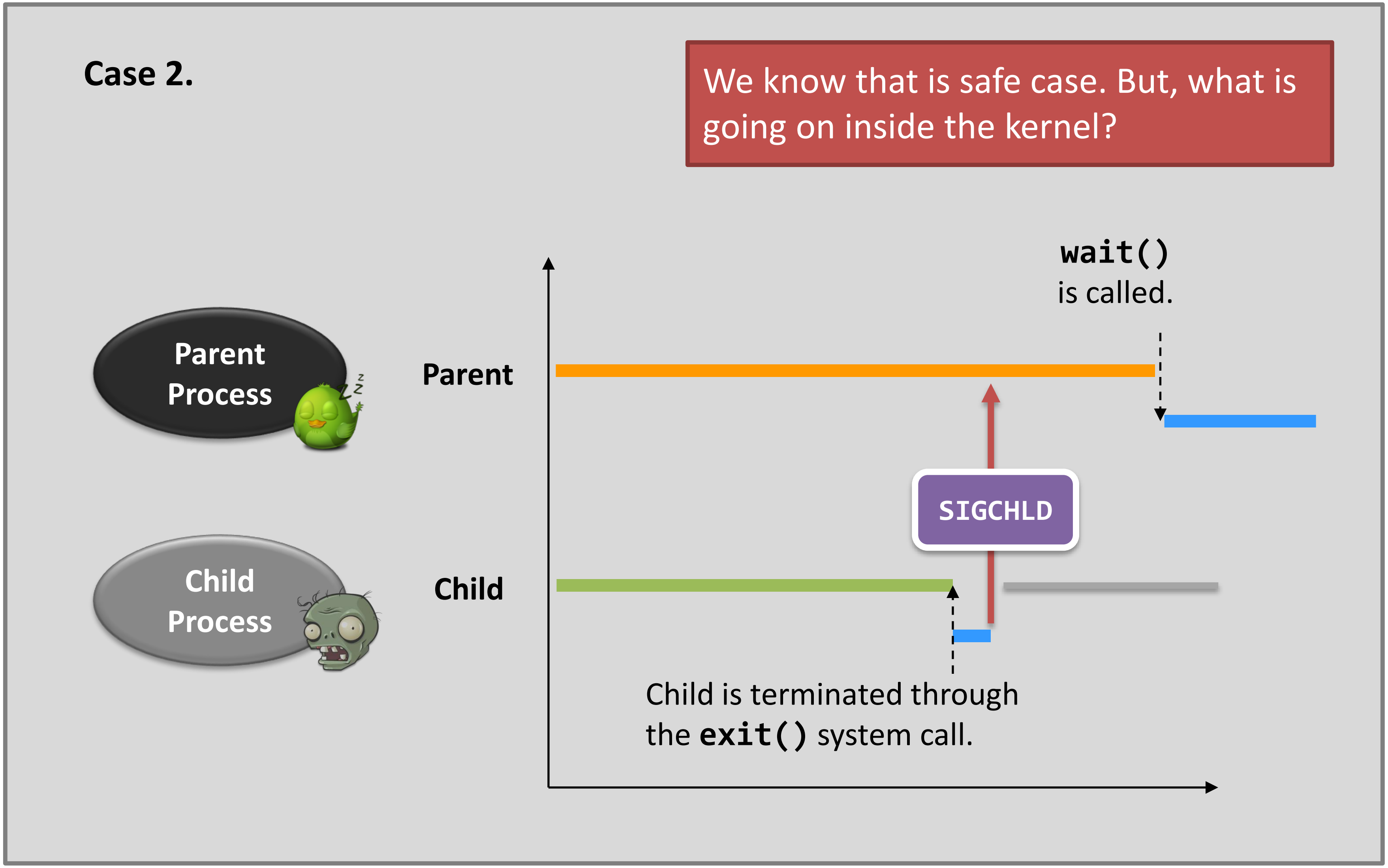
The return value of **`wait()`** system call is the PID of the terminated child.

See manpage for details.

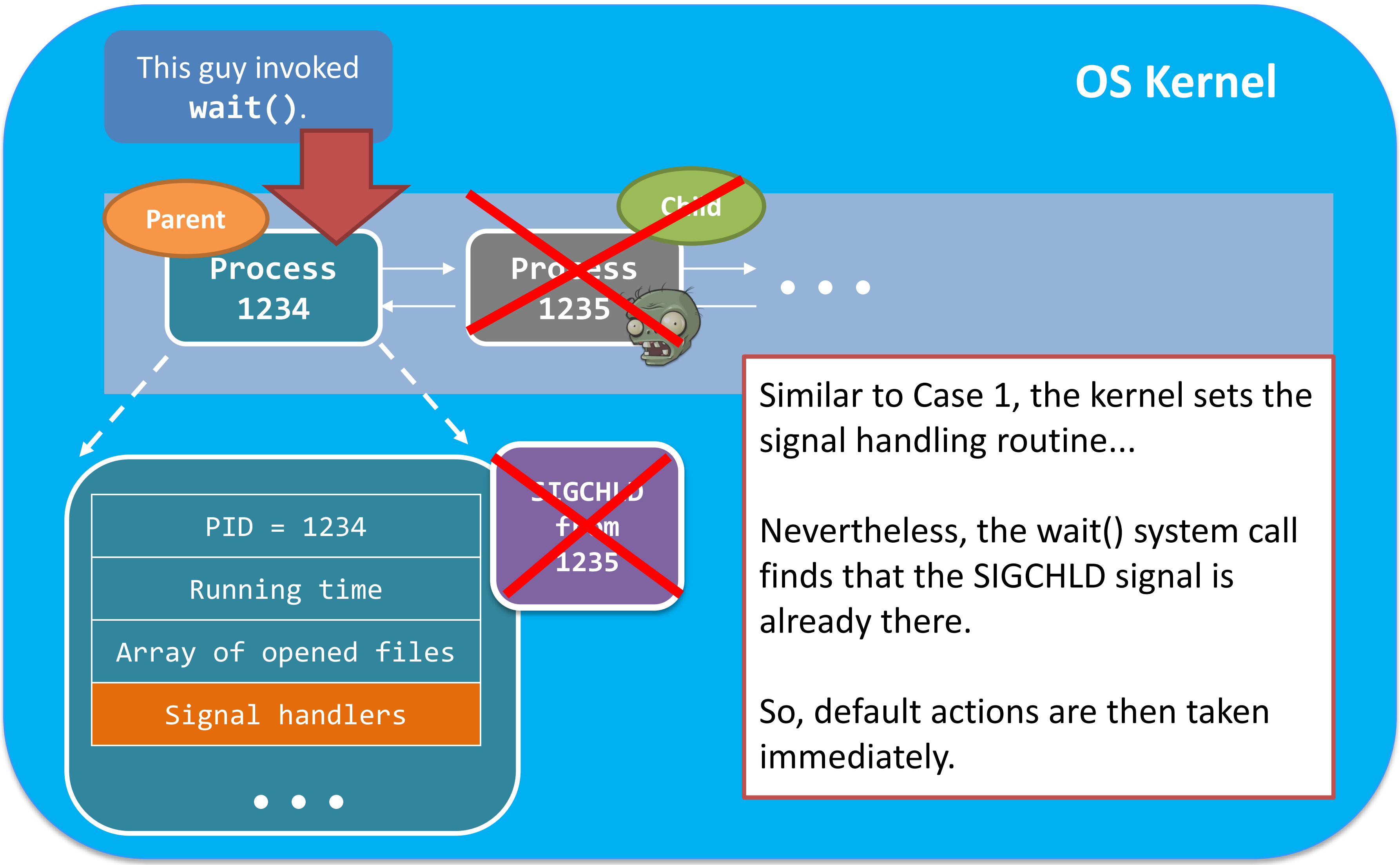
`wait()` and `exit()` – parent side



`wait()` and `exit()` – parent side



`wait()` and `exit()` – parent side

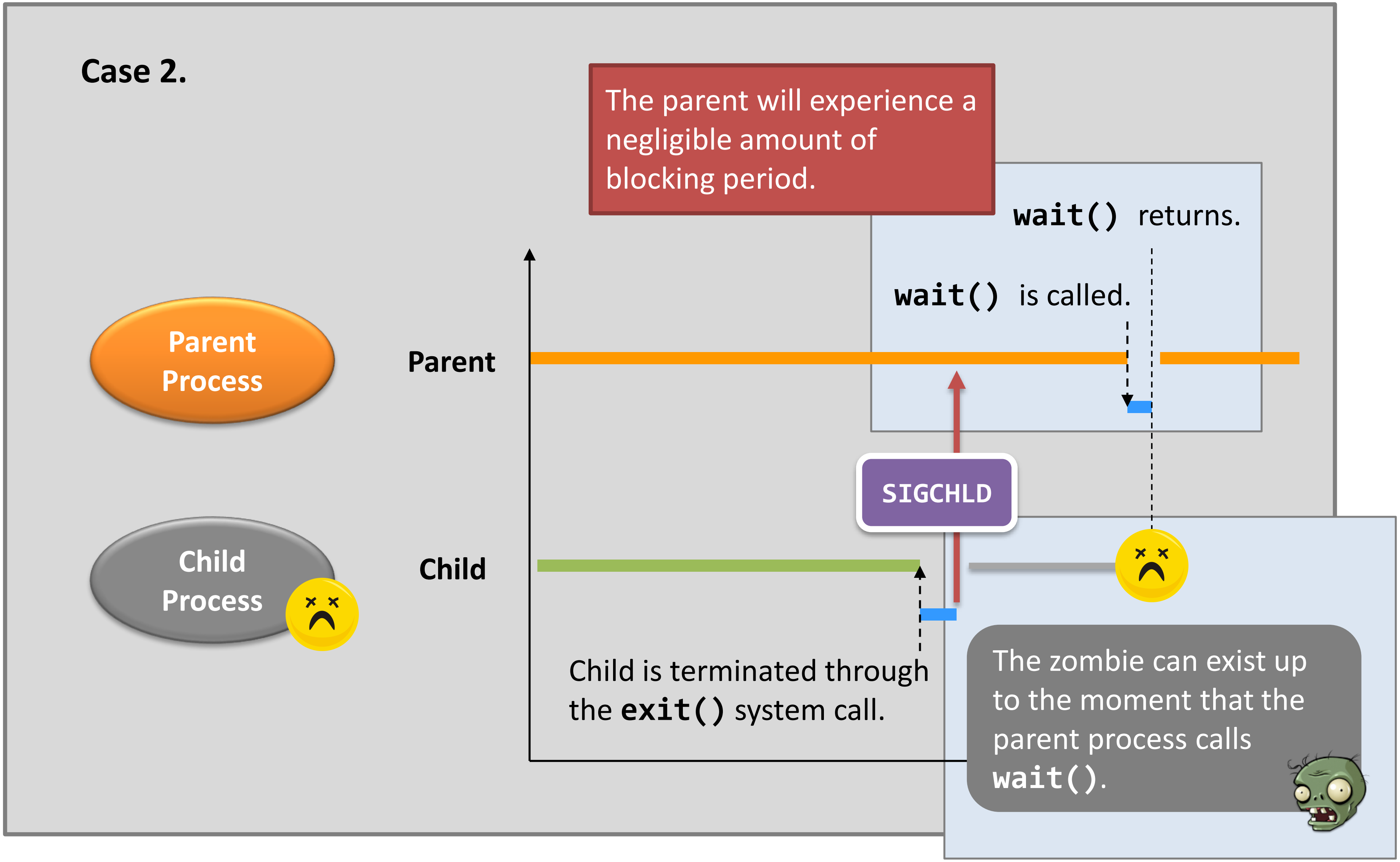


Similar to Case 1, the kernel sets the signal handling routine...

Nevertheless, the wait() system call finds that the SIGCHLD signal is already there.

So, default actions are then taken immediately.

`wait()` and `exit()` – parent side



`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.
    - You know, the kernel knows that the process is terminated abnormally. Hence, the kernel invokes **`exit()`** by itself.

`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.
- Linux will label zombie processes as “**<defunct>**”.
  - To look for them:

```
$ ps aux | grep defunct
tywong  3150 ... [test] <defunct>
$ _
```

PID of the process

Name of the program, i.e., test in the example

`wait()` and `exit()` – short summary

```c
int main(void)
{
    int pid;
    if( (pid = fork()) ) {
        printf("Look at the status of the 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** between the 1st and the 2nd “enter”.

```
[examples@3150]$ cat zombie.c
```

Working of system calls

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

The role of `wait()` in the OS…

- Calling **wait()** is important.
  - It is not about process execution/suspension…
  - It is about **system resource management**.

- Think about it:
  - 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?

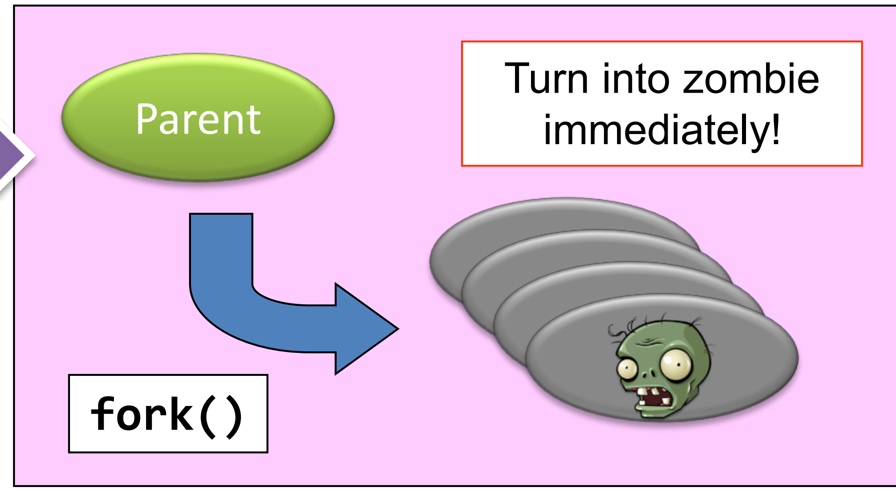
When `wait()` is absent…

- Don’t try this in department’s machines…

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

Parent: never reach here.

Child: reached immediately, but no corresponding **`wait()`** for the parent.



An infinite, zombie factory!

`No matter which department you are from. DON’T TRY!`

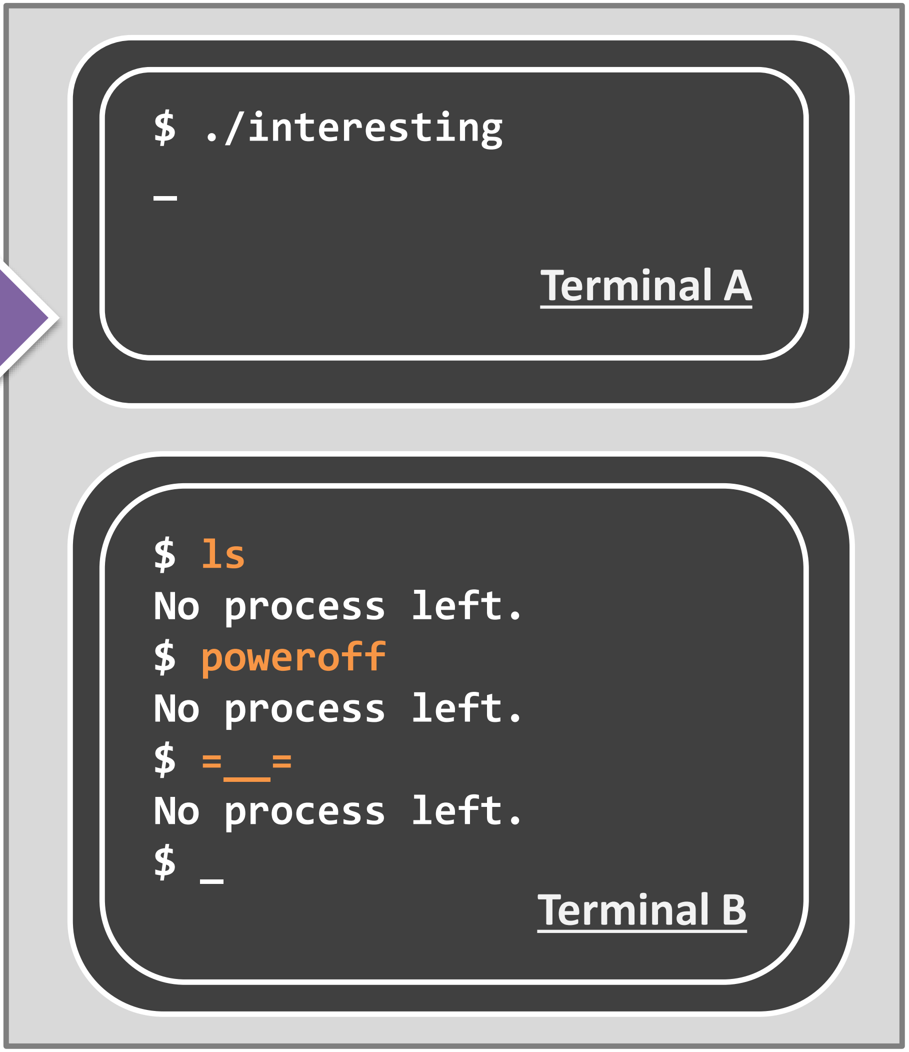
When `wait()` is absent...

- Don't try this in department's machines...

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

Parent: never reach here.

Child: reached immediately, but no corresponding **`wait()`** for the parent.



Summary

- Knowing the internal workings of the system calls is important.
  - System calls such as **`exit()`** produces surprising results.
  - Understanding those system calls also let you to develop a better concept over process control.
  - At least, you don't have to guess what will happen after calling a system call.