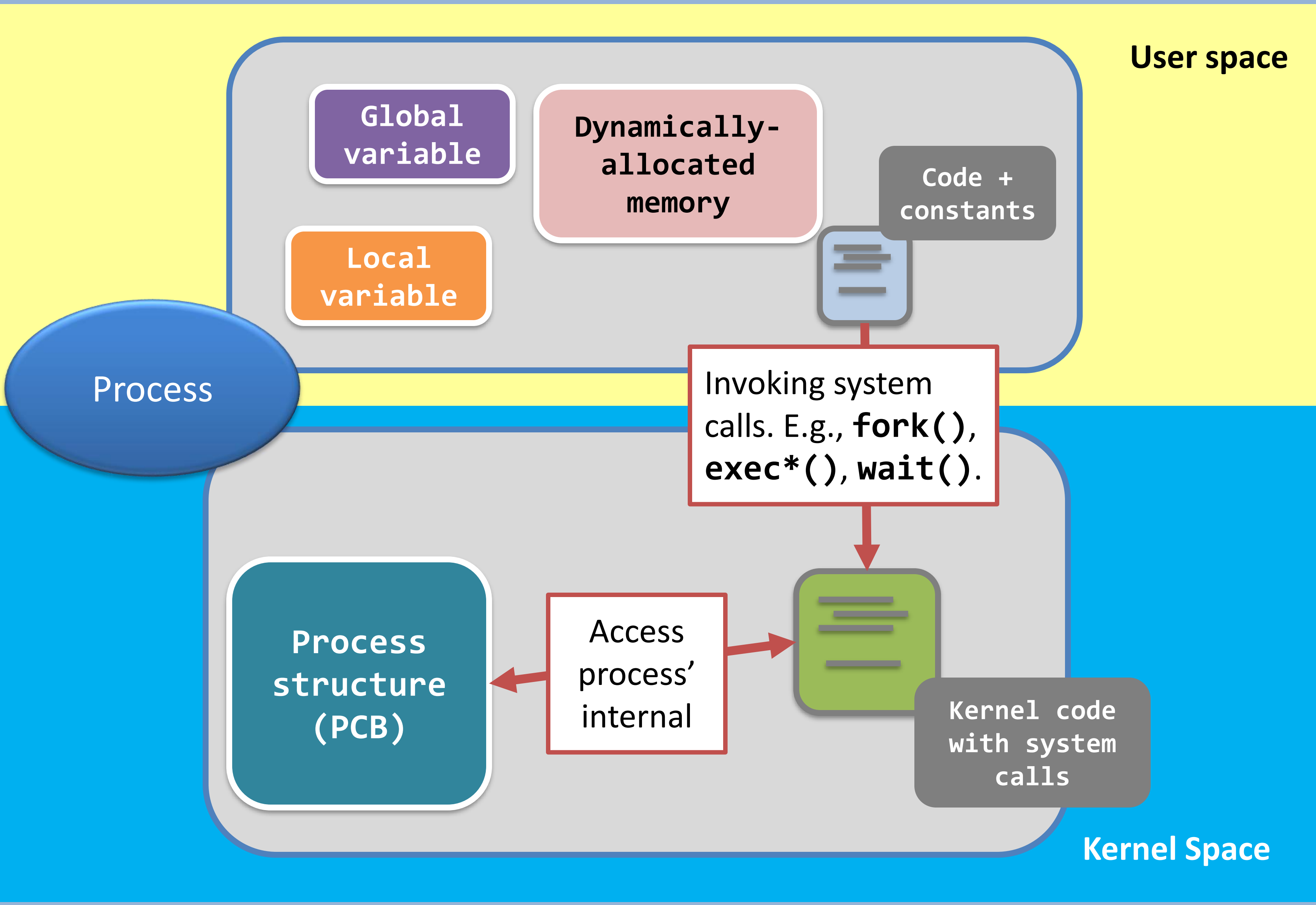
Operating Systems

Prof. Yongkun Li
中国科大-计算机学院 教授
http://staff.ustc.edu.cn/~ykli

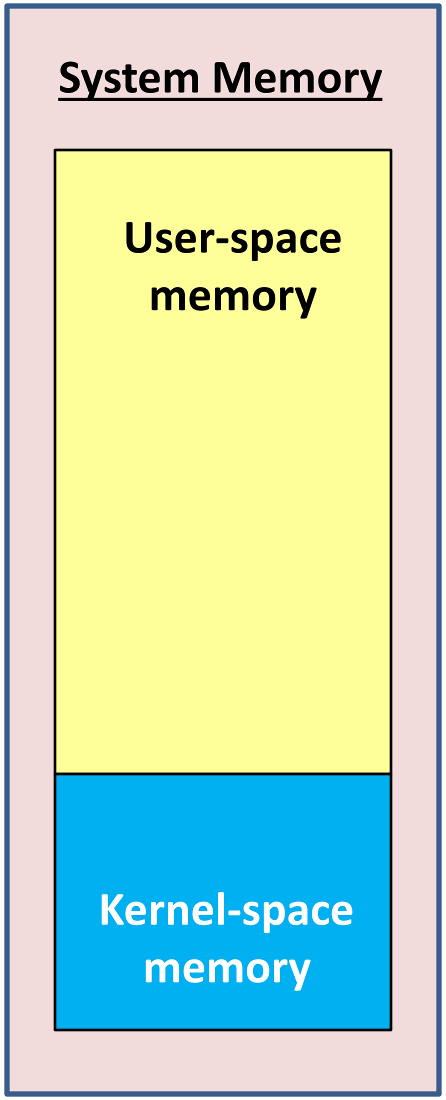
Ch3 - Process Operations

-from kernel's perspective

Process in Memory

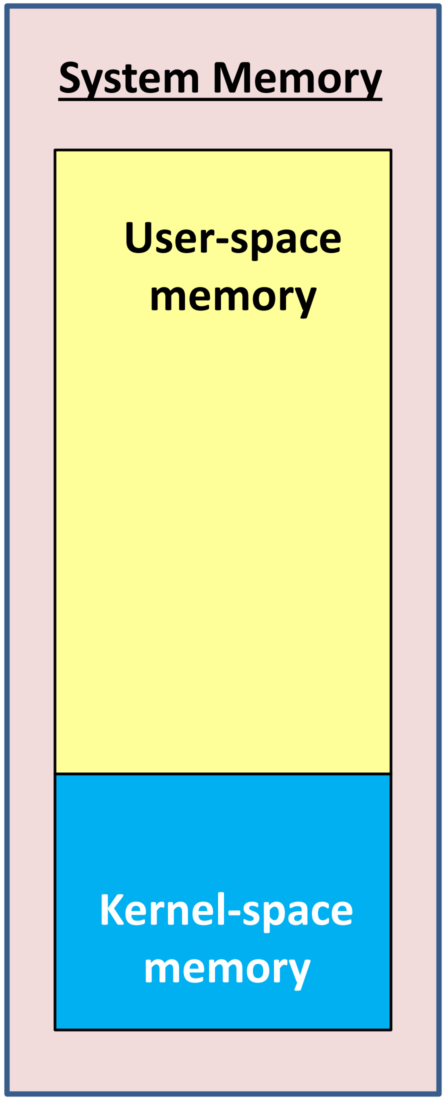


Kernel-space VS User-space



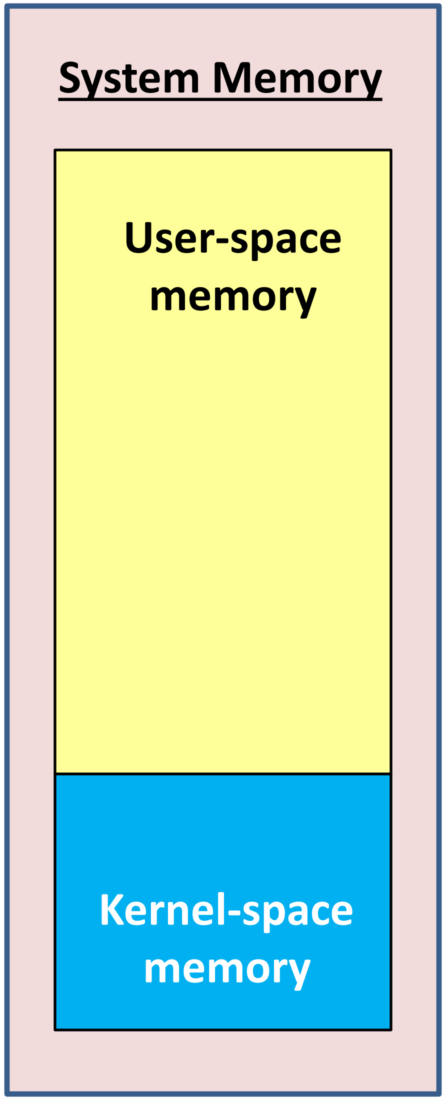
Kernel-space VS User-space

	Kernel-space memory	User-space memory
Storing what		
Accessed by whom		



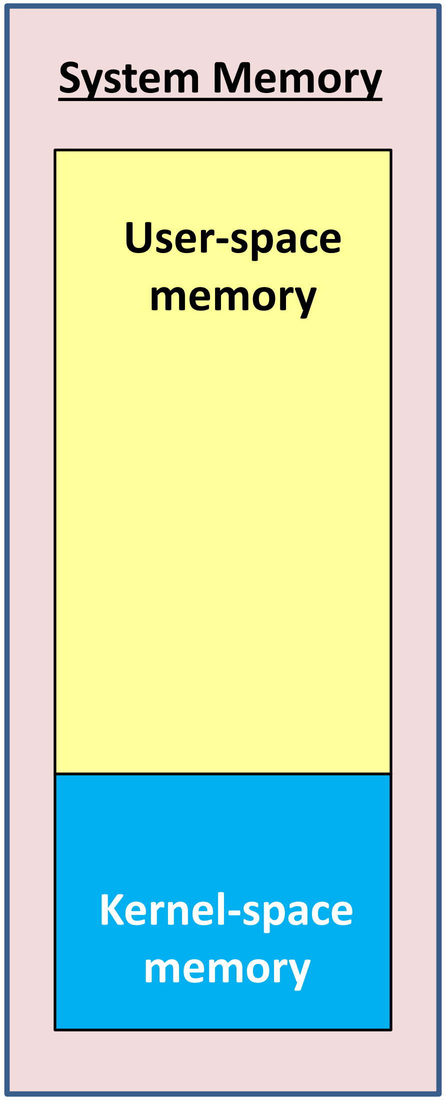
Kernel-space VS User-space

	Kernel-space memory	User-space memory
Storing what	Kernel data structure Kernel code Device drivers	Process' memory Program code of the process
Accessed by whom		



Kernel-space VS User-space

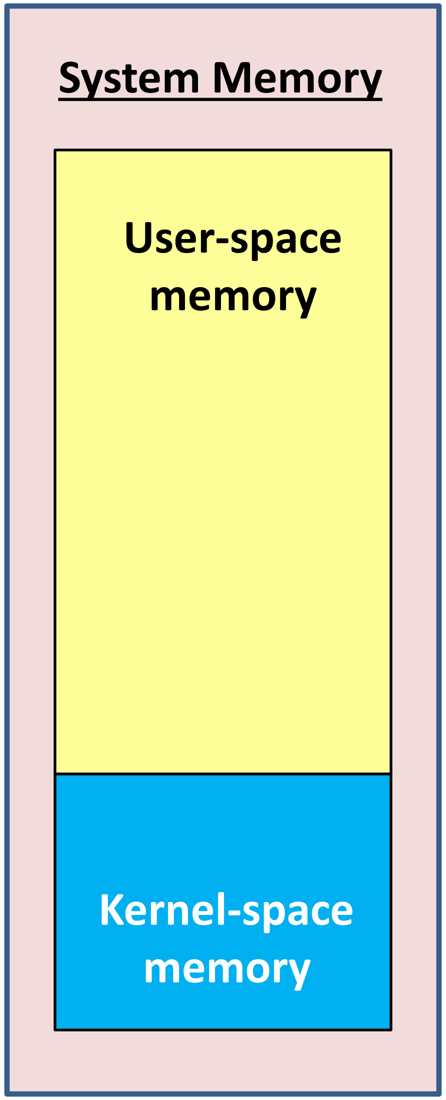
	Kernel-space memory	User-space memory
Storing what	Kernel data structure Kernel code Device drivers	Process' memory. Program code of the process
Accessed by whom	Kernel code	User program code + kernel code



The kernel is invincible!

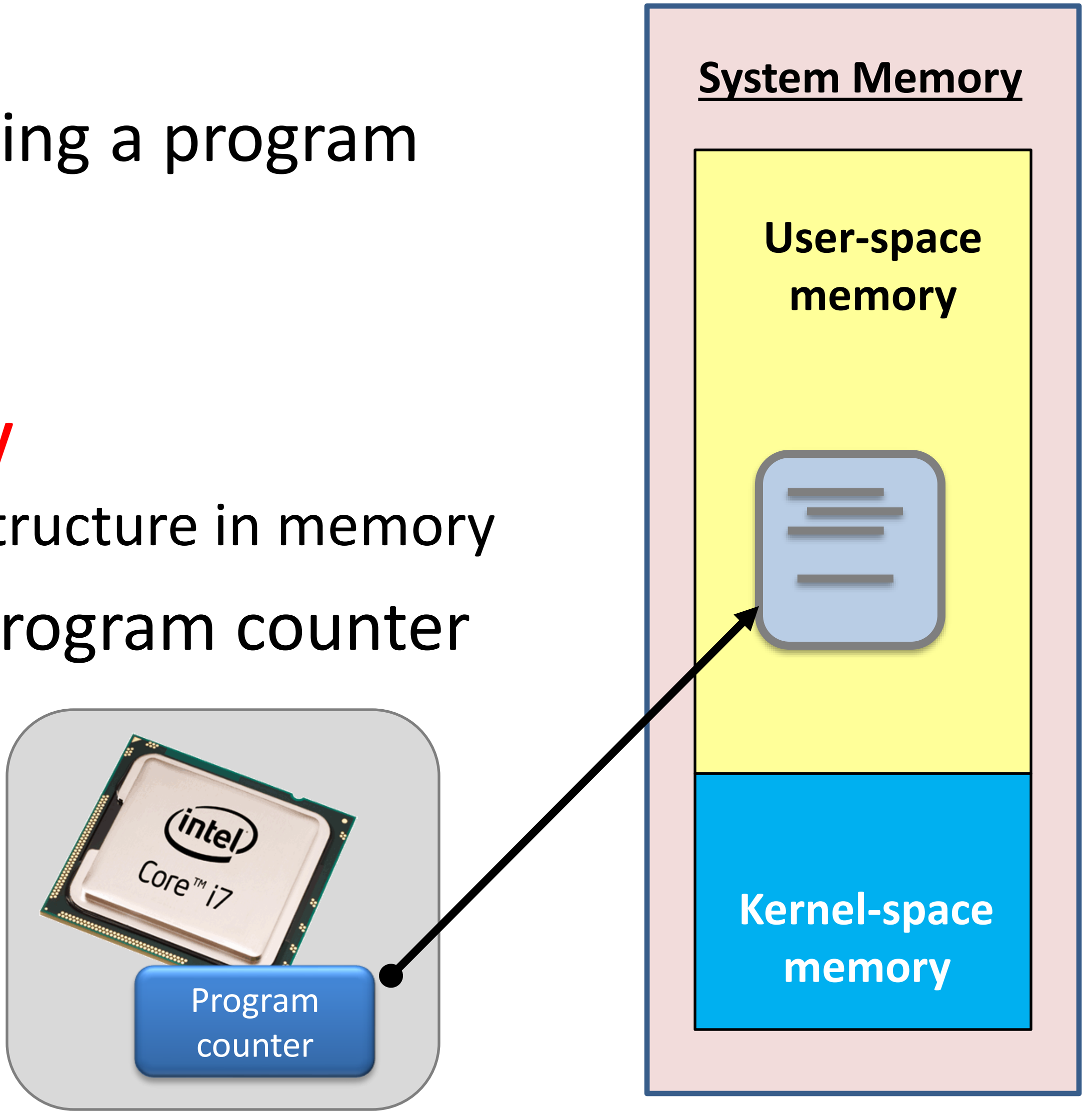
Process is going back and forth...

- A process will switch its execution from user space to kernel space
- **How?**
  - **through invoking system call**



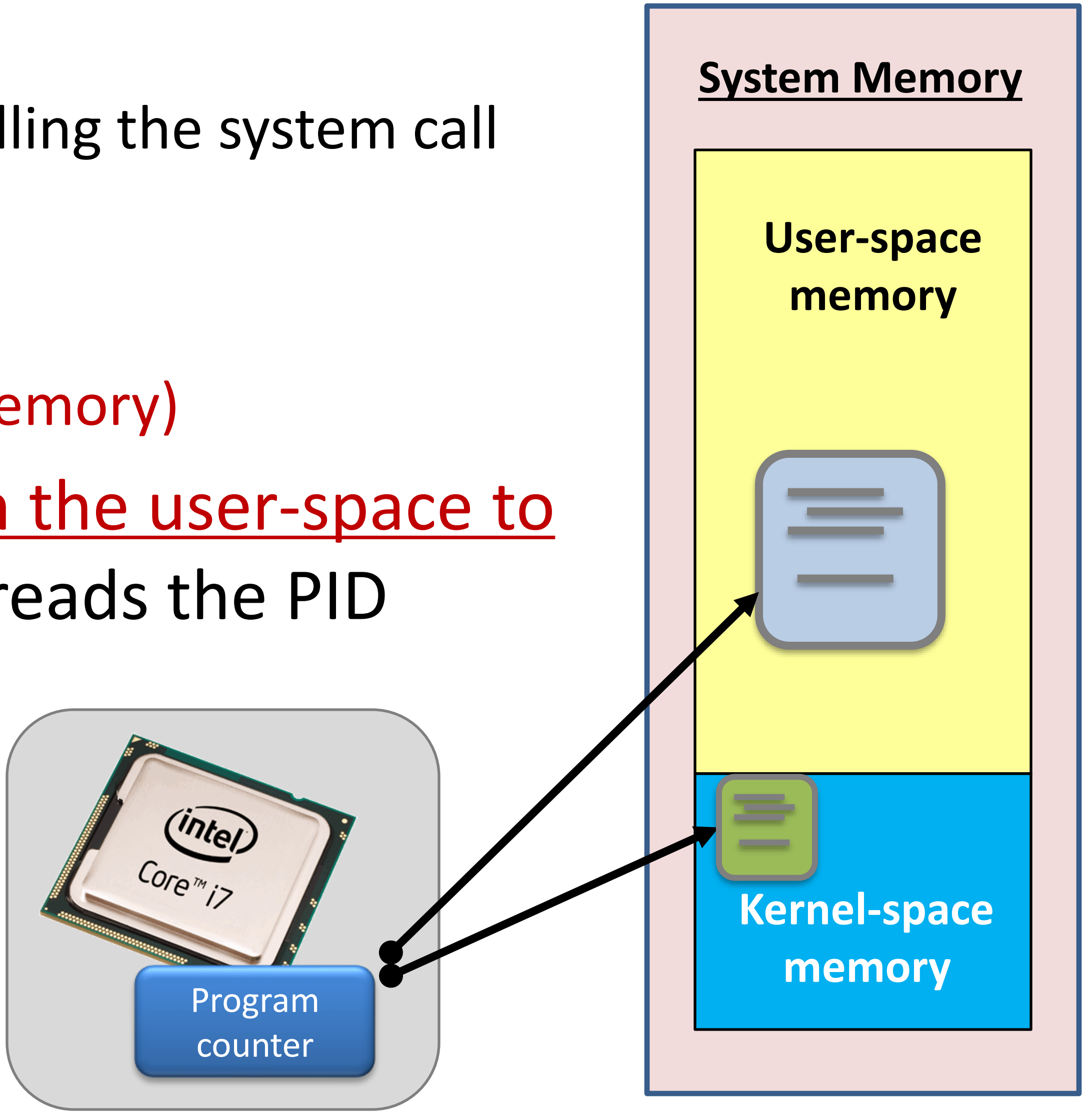
Process is going back and forth...

- Example
  - Say, the CPU is running a program code of a process
  - Where is the code?
    - **User-space memory**
    - Recall the process structure in memory
  - Where should the program counter point to?



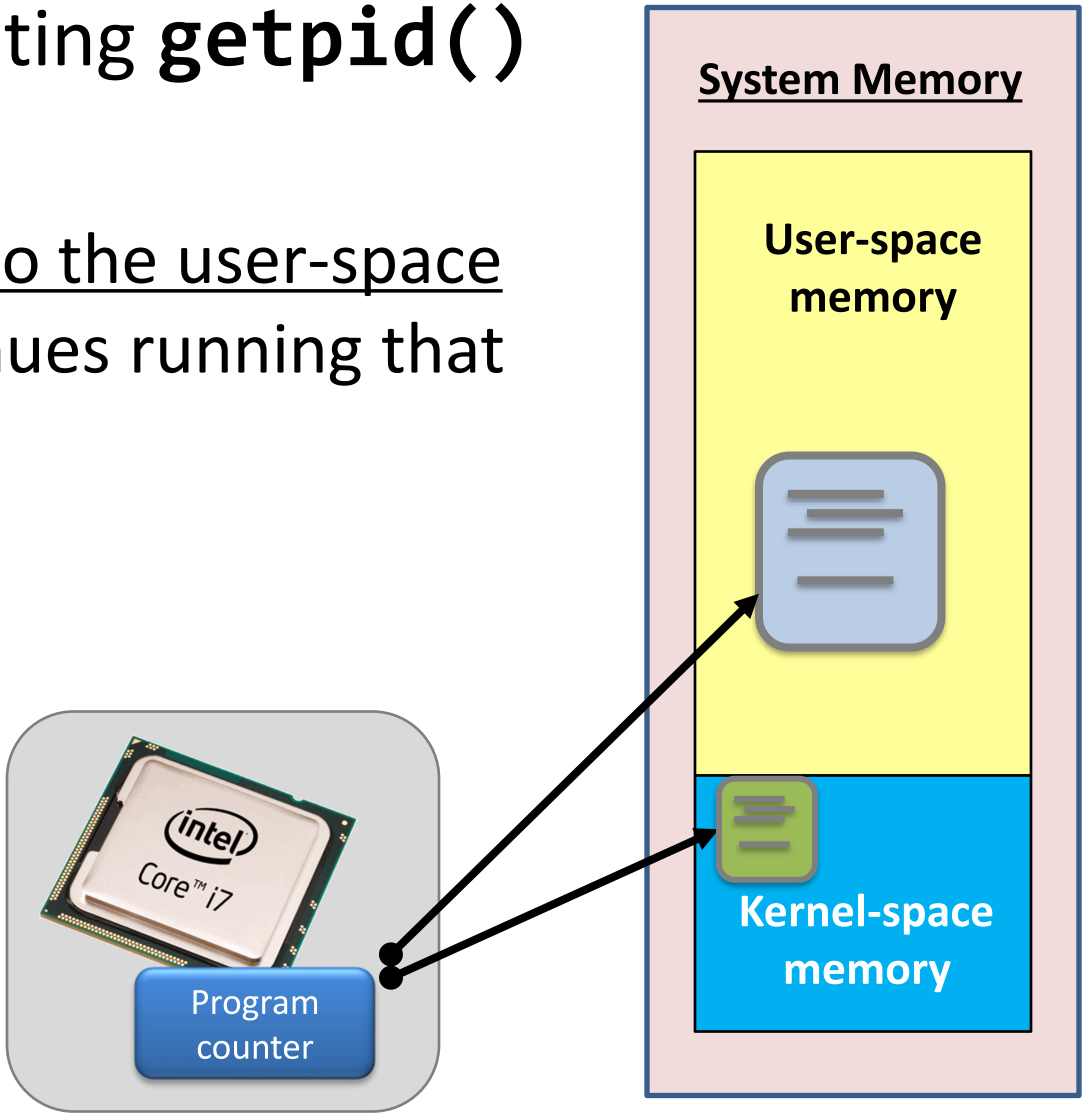
Process is going back and forth...

- What happens...
  - When the process is calling the system call "**`getpid()`**"
- Where to get the PID
  - PCB (in kernel-space memory)
- The CPU switches <u>from the user-space to the kernel-space</u>, and reads the PID



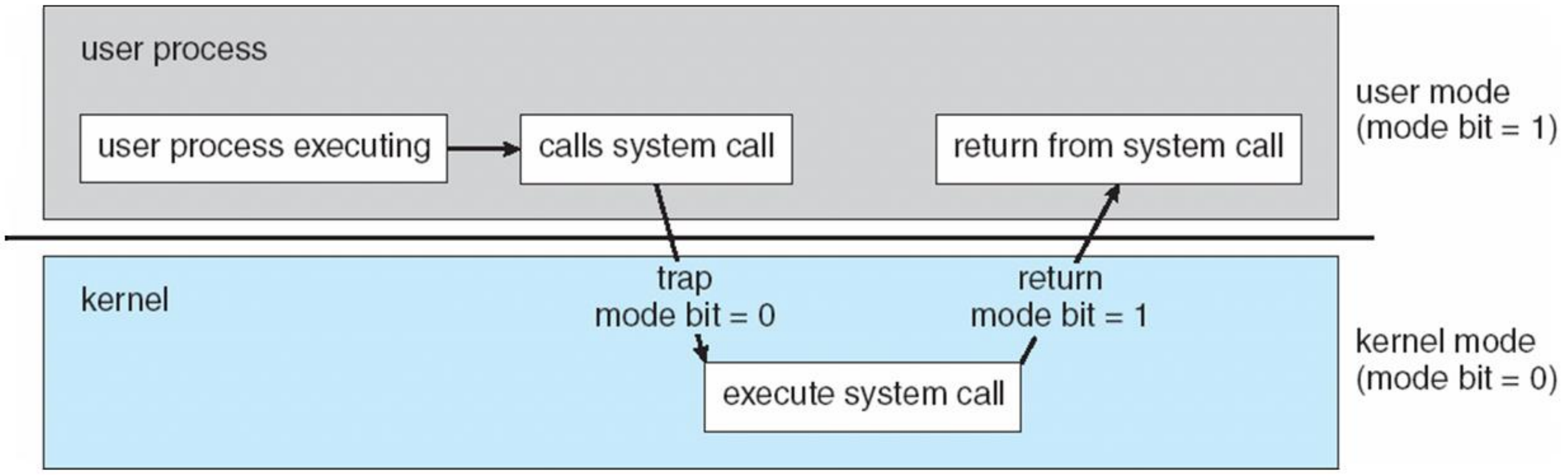
Process is going back and forth...

- After finished executing **`getpid()`**
  - What happens?
  - CPU switches back to the user-space memory, and continues running that program code



User Mode & Kernel Mode

- Remember this?



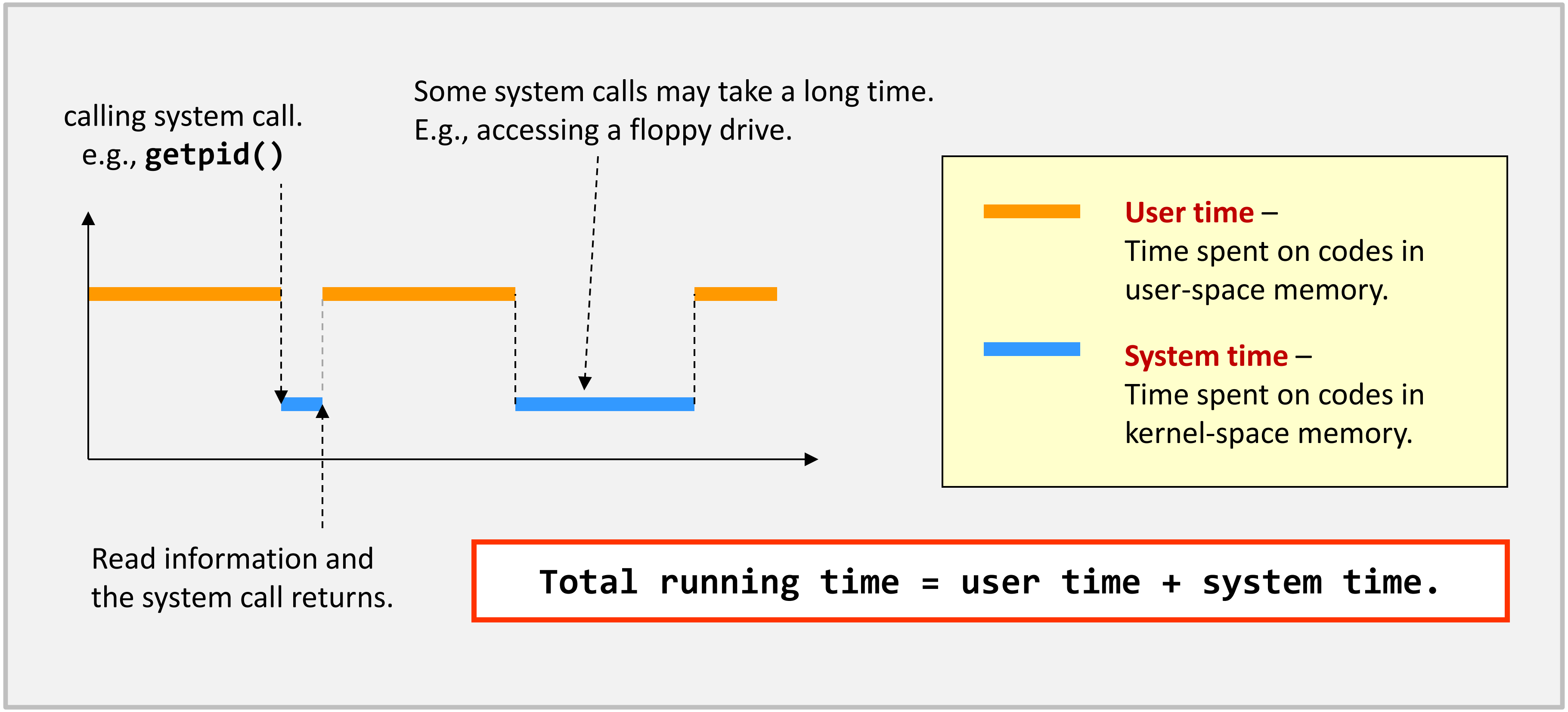
Another question: How much time was spent in each part?

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 and system time

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 and system time



User time VS System time – example 1

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

```
$ time ./time_example

real      0m0.003s
user      0m0.003s
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 <= 100000; i++) {
        x = x + i;
    //  printf("x = %d\n", x);
    }
    return 0;
}
```

Commented on purpose.

Why comment this line???

User time VS System time – example 1

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

```
$ time ./time_example

real      0m0.003s
user      0m0.003s
sys       0m0.000s
$ _
```

```
int main(void) {
    int x = 0;
    for(i = 1; i <= 100000; i++) {
        x = x + i;
    //  printf("x = %d\n", x);
    }
    return 0;
}
```

Commented on purpose.

```
$ time ./time_example

real      0m0.677s
user      0m0.032s
sys       0m0.227s
$ _
```

```
int main(void) {
    int x = 0;
    for(i = 1; i <= 100000; i++) {
        x = x + i;
        printf("x = %d\n", x);
    }
    return 0;
}
```

Comment released.

See? Accessing hardware costs the process more time.

User time VS System time – example 2

- What is the difference of the two programs?

```
#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;
}
```

Lessons learned: When writing a program, you must consider both the user time and the system time

User time VS System time – short summary

- The user time and the system time together define the **performance** of an application
  - System call plays a major role in **performance**.
  - **Blocking system call:** some system calls even **stop your process** until the data is available.
- Programmers should pay attention to system performance
  - Reading a file byte-by-byte
  - Reading a file block-by-block, where the size of a block is 4,096 bytes

Story so far…

User space and Kernel space

User time and system time

Next…

Working of system calls

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

Next…

Working of system calls

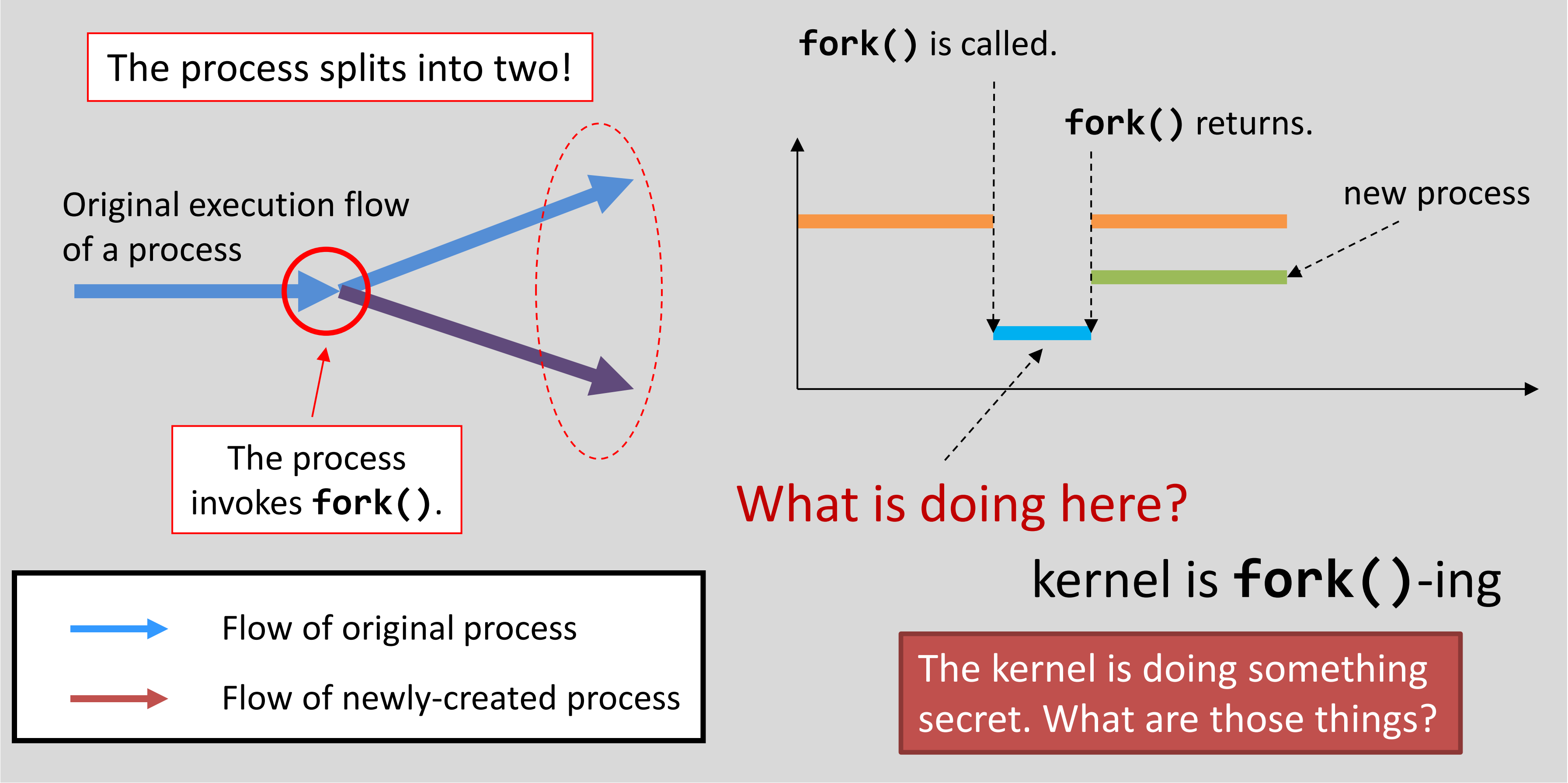
- `fork();`
- `exec*();`
- `wait() + exit();`

`fork()`

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

fork()

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



`fork()`

- From the Kernel's view…

Guess: What will be modified?

Process creation – `fork()` system call

Recall

- **`fork()`** behaves like "*cell division*".
  - It creates the child process by **cloning** from the parent process, including…

Cloned items	Descriptions
Program counter [CPU register]	That's why they both execute from the same line of code after **`fork()`** returns.
Program code [File & Memory]	They are sharing the same piece of code.
Memory	Including local variables, global variables, and dynamically allocated memory.
Opened files [Kernel's internal]	If the parent has opened a file "A", then the child will also have file "A" opened automatically.

Process creation – **`fork()`** system call

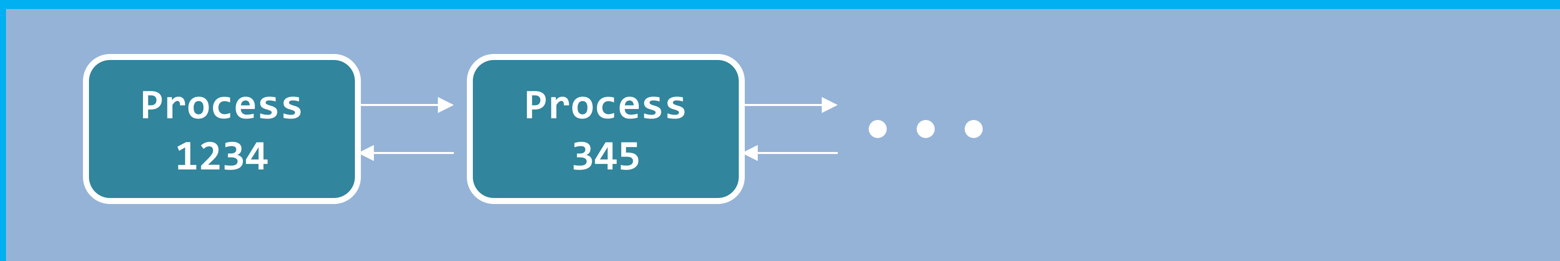
Recall

- However…
  - **`fork()`** does not clone the following...
  - Note: they are all data inside the memory of kernel.

Distinct items	Parent	Child
Return value of `fork()`	PID of the child process.	0
PID	Unchanged.	Different, not necessarily be "Parent PID + 1"
Parent process	Unchanged.	Doesn't have the same parent as that of the parent process.
Running time	Cumulated.	Just created, so should be 0.

`fork()` in action – the start…

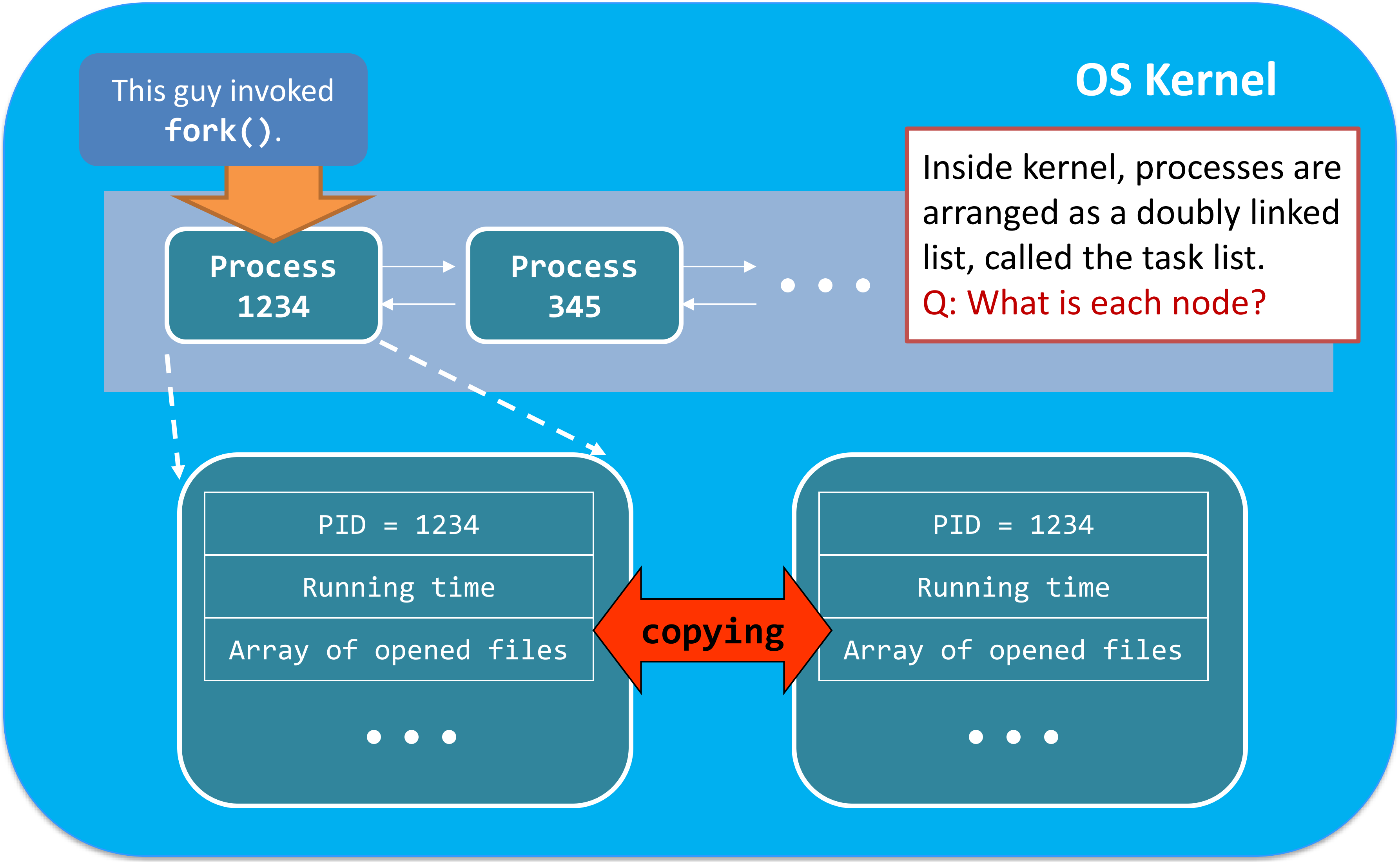
OS Kernel



Inside kernel, processes are arranged as a doubly linked list, called the task list.

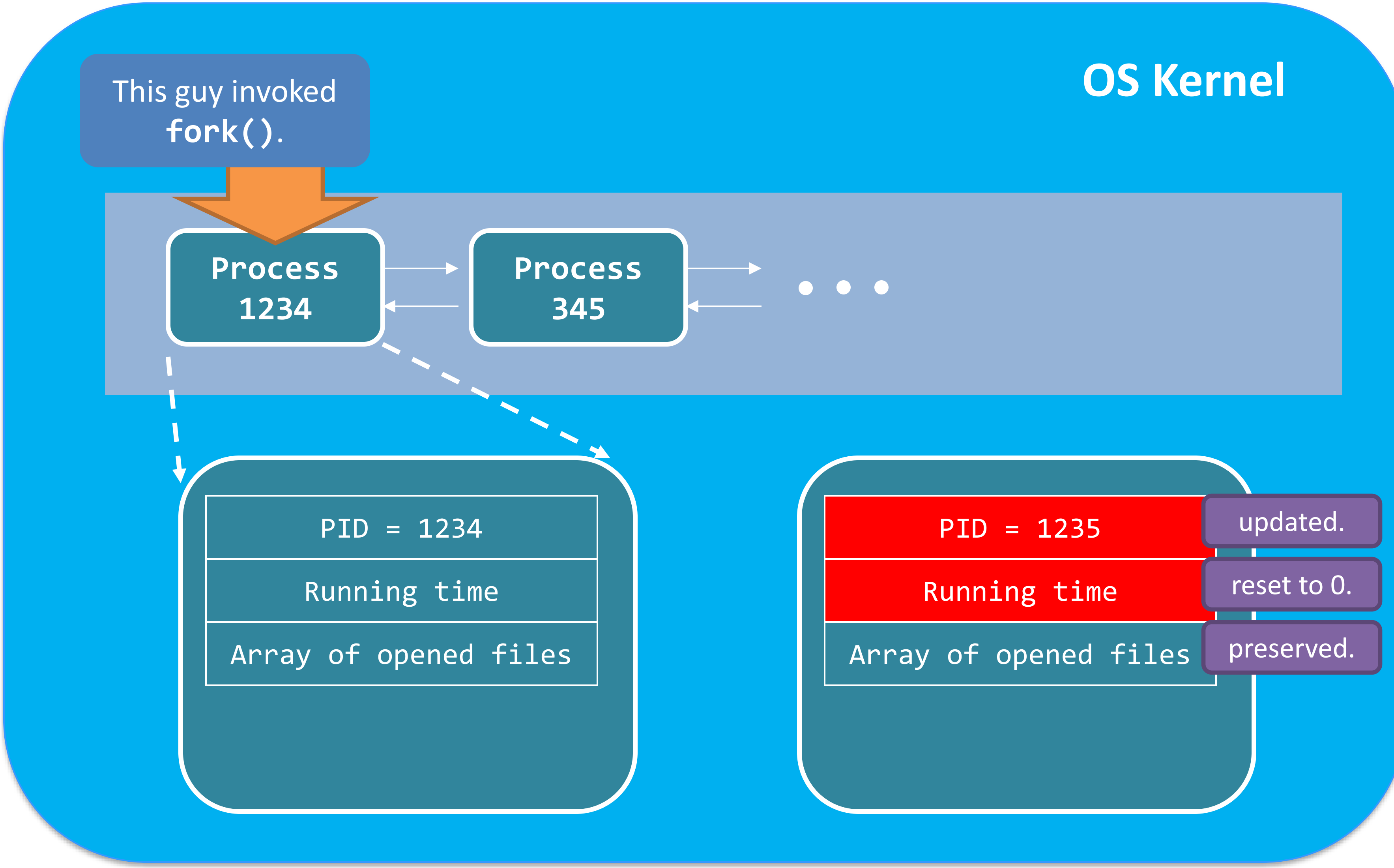
Q: What is each node?

`fork()` in action – the start...

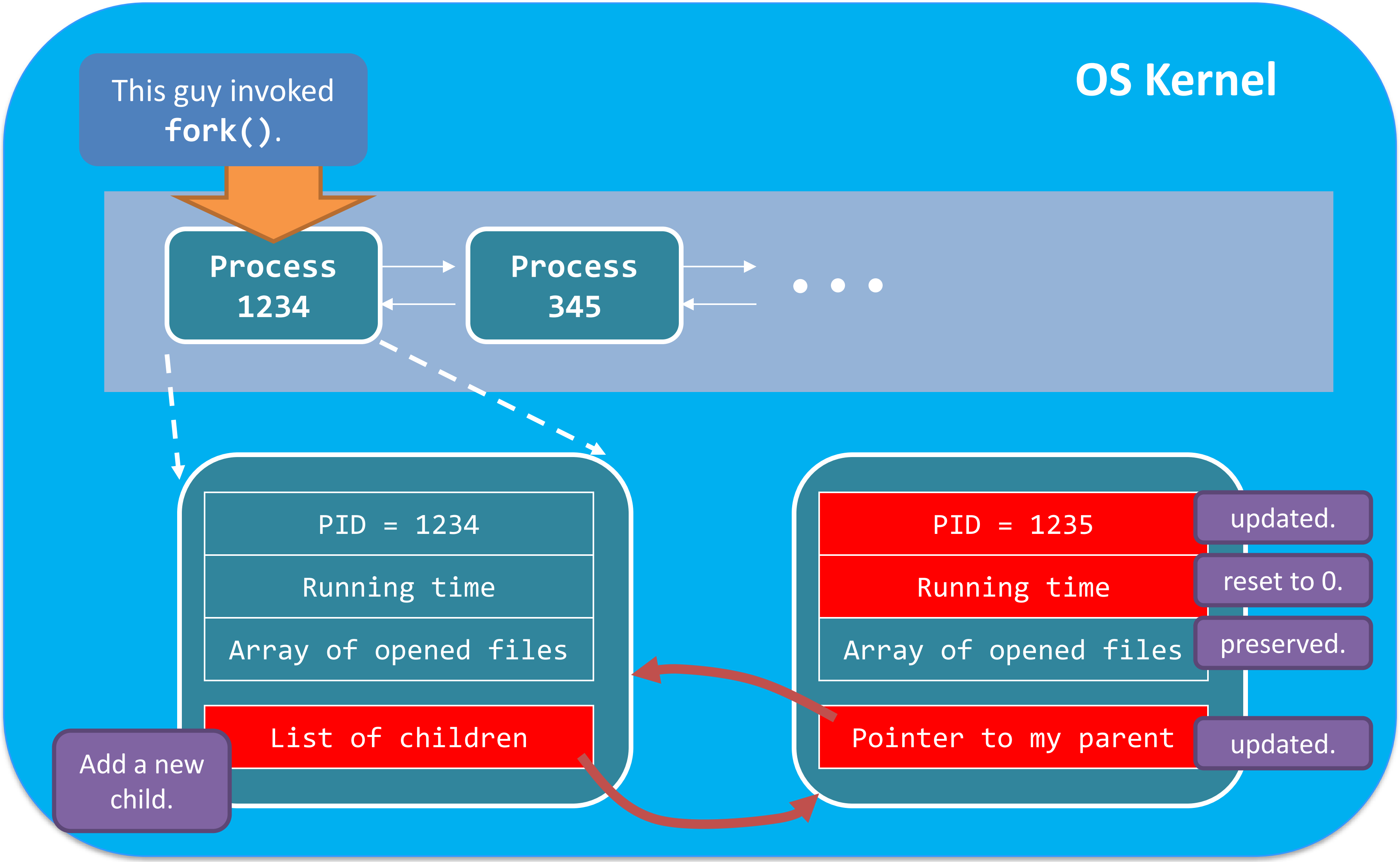


Inside kernel, processes are arranged as a doubly linked list, called the task list.
Q: What is each node?

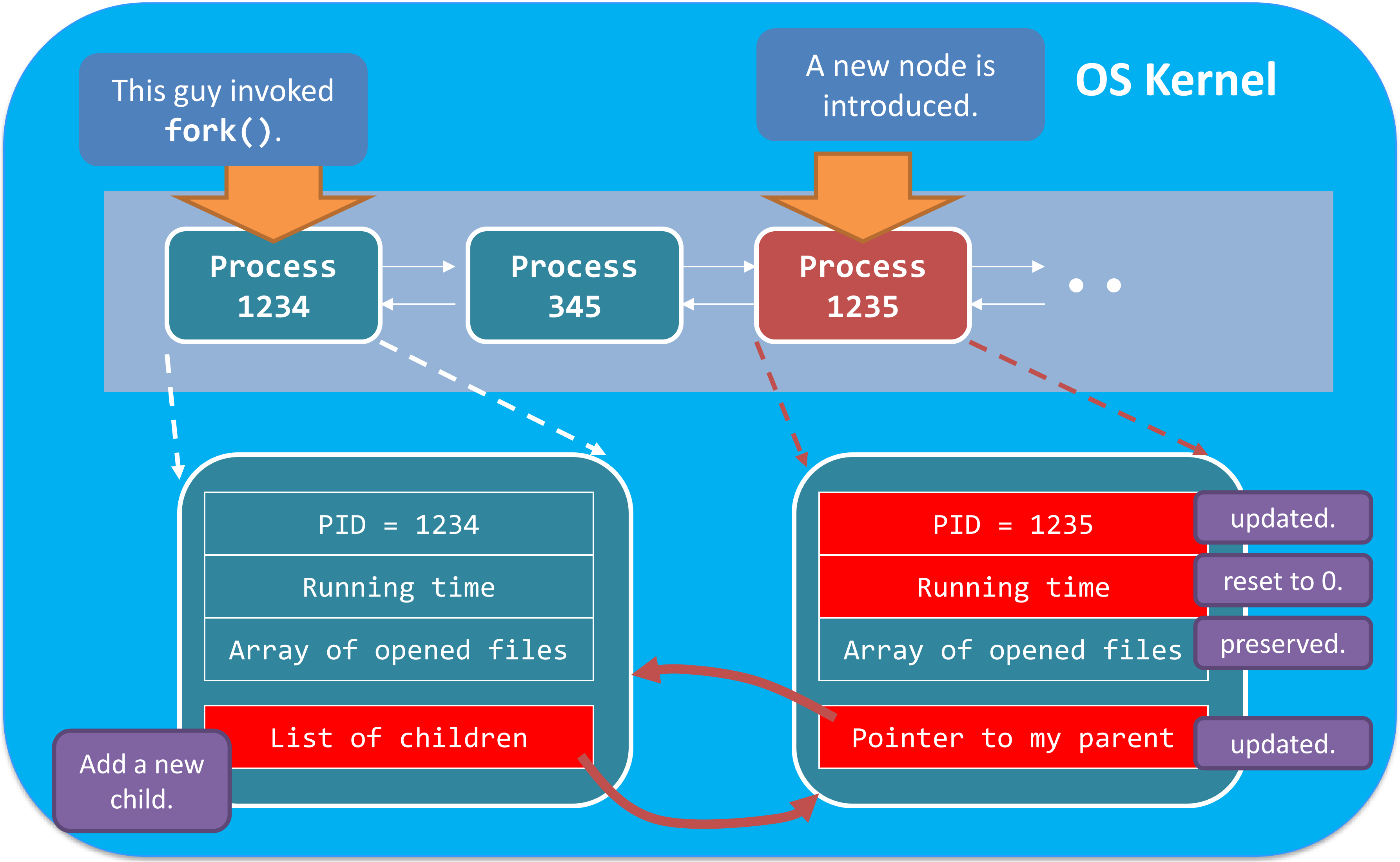
`fork()` in action – kernel-space update



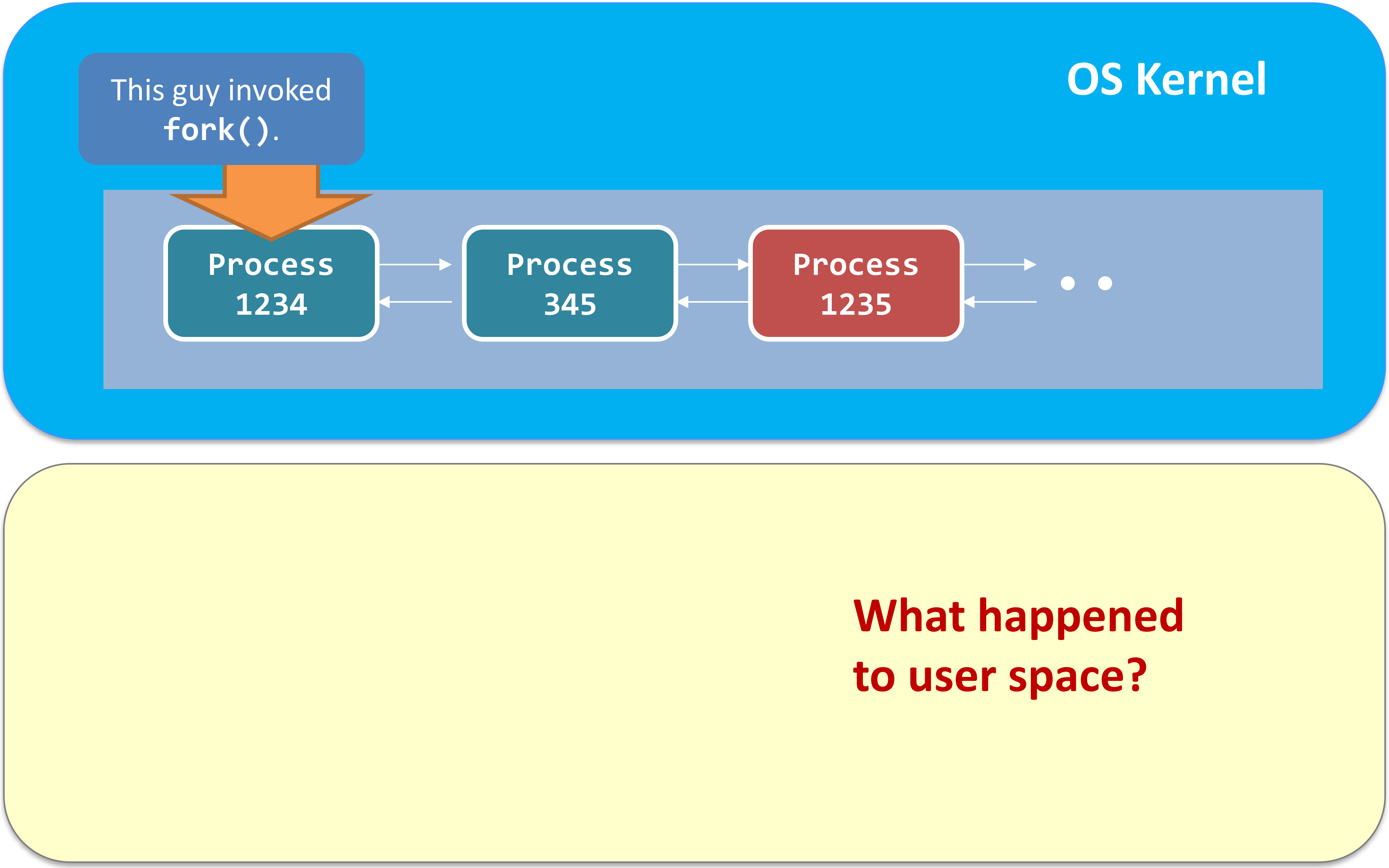
fork() in action – kernel-space update



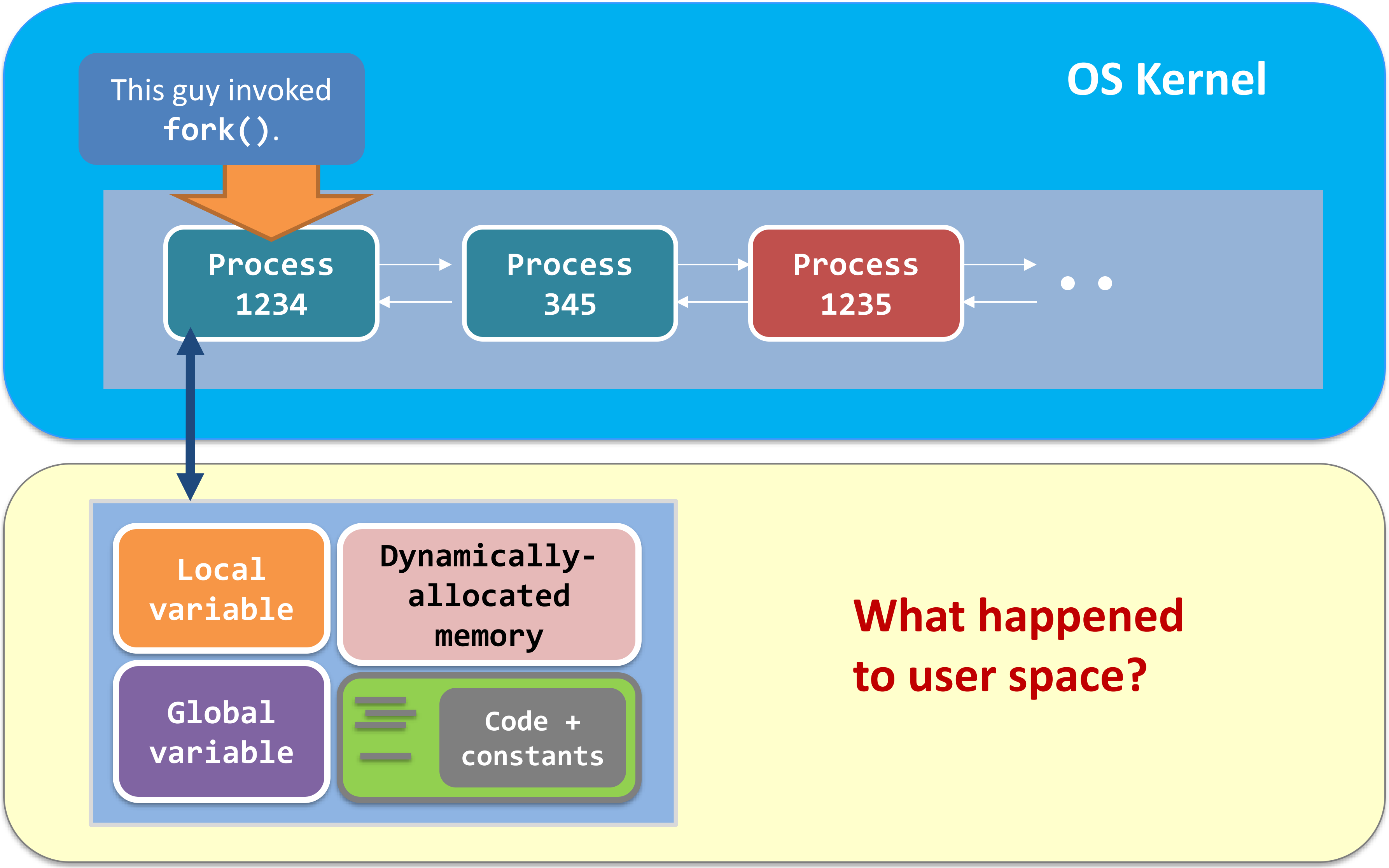
`fork()` in action – kernel-space update



fork() in action – user-space update

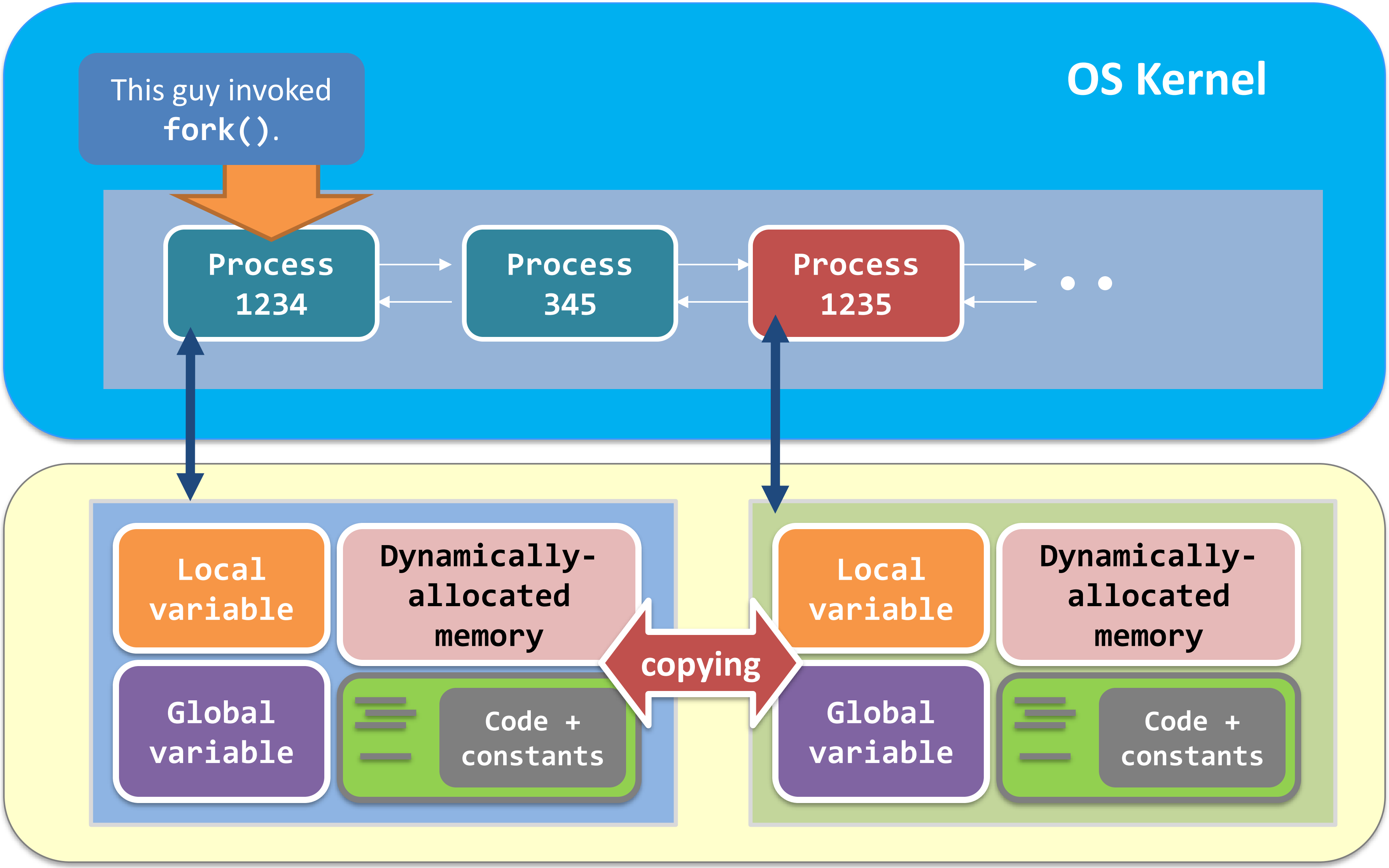


fork() in action – user-space update

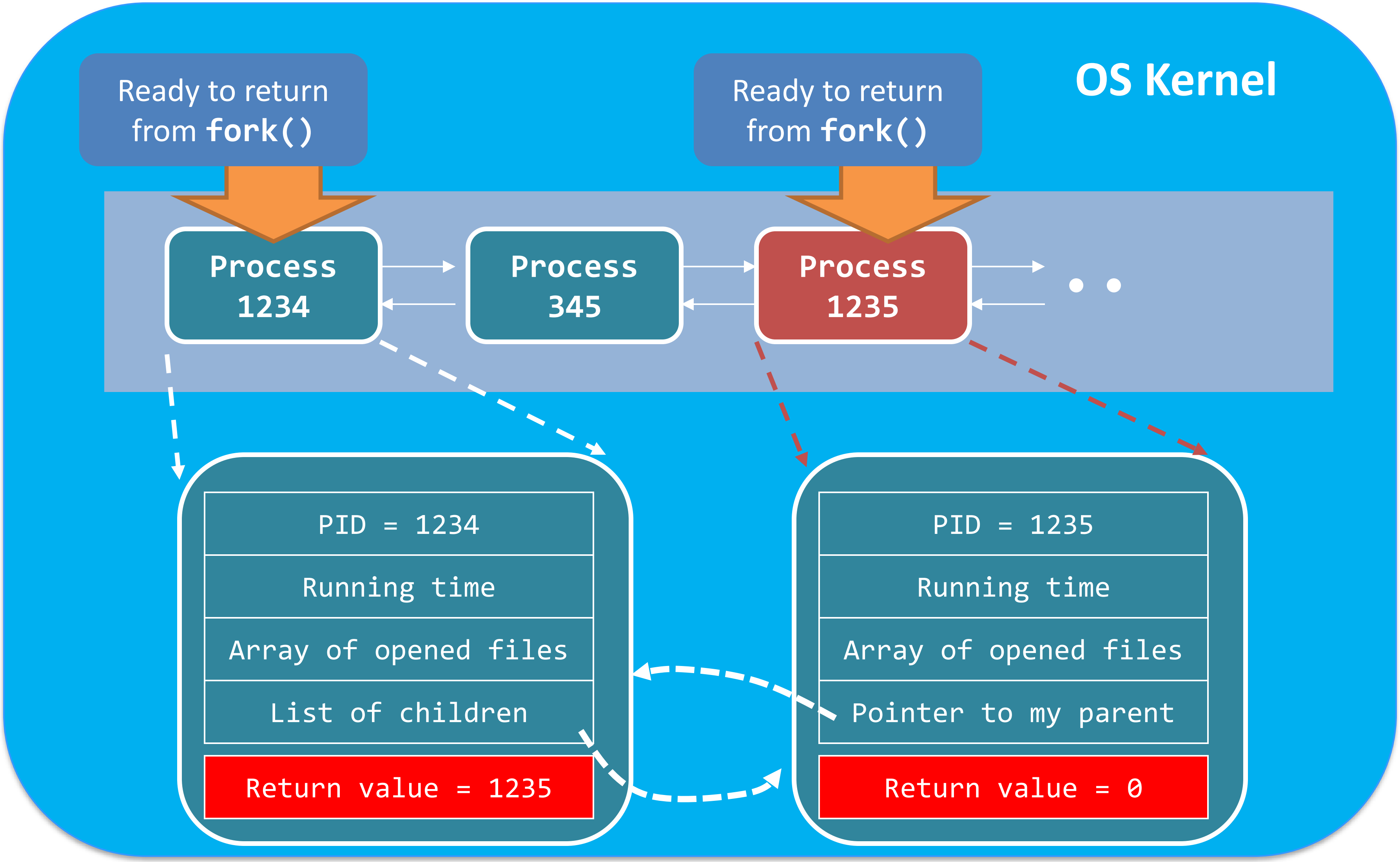


What happened to user space?

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



fork() in action – finish



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

- After `fork()`
  - The child process share a set of opened files
- What are the array of opened files?

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!

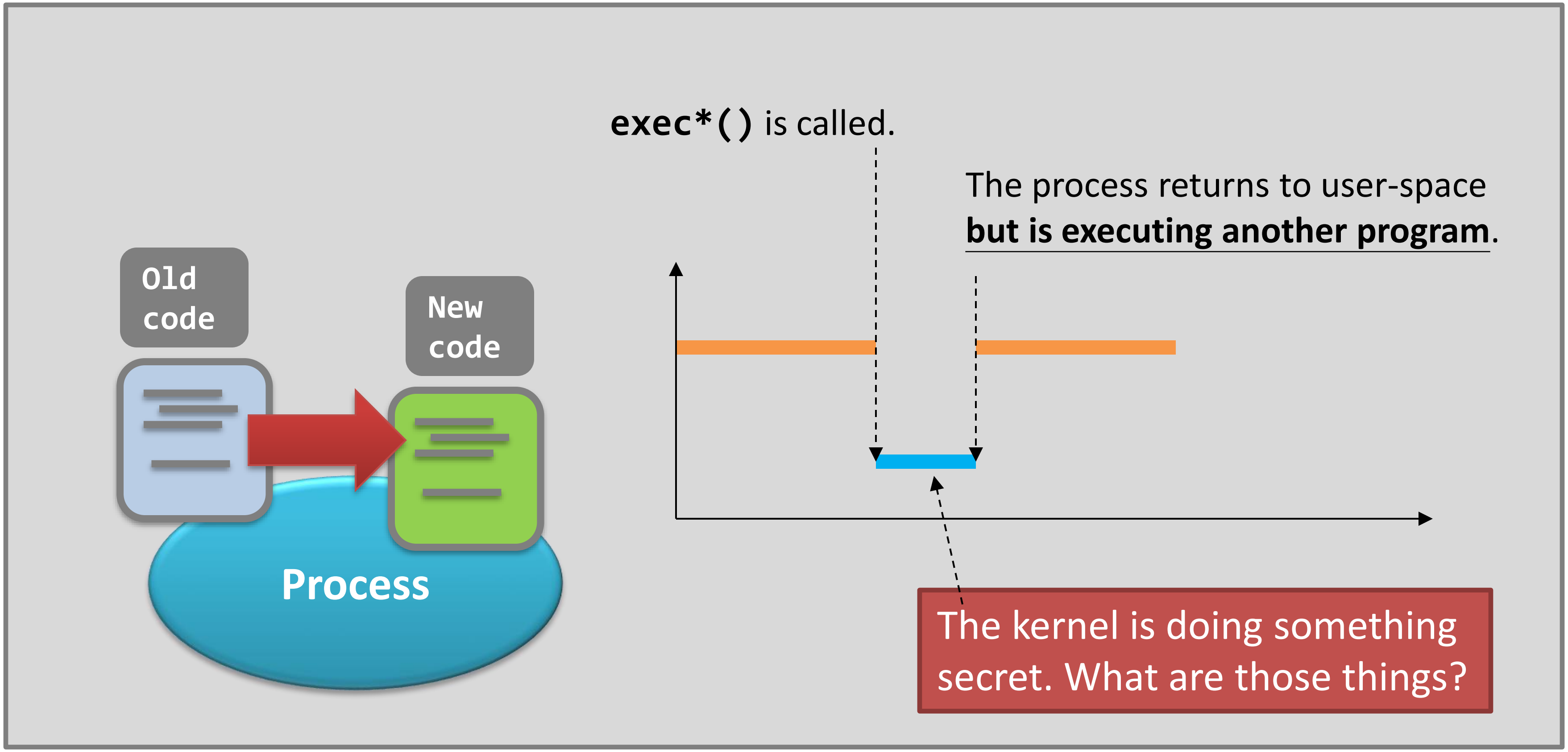
Working of system calls

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

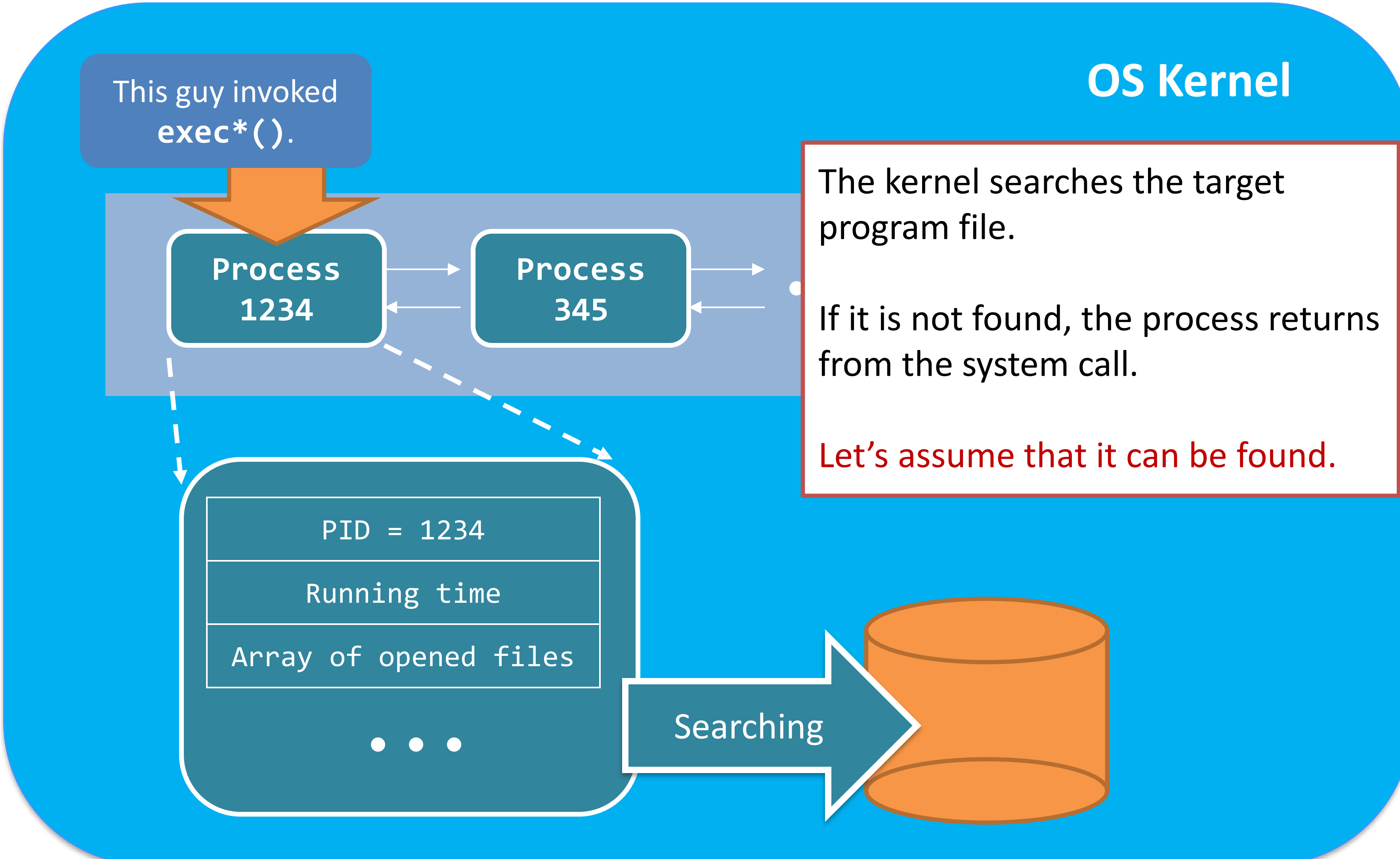
exec*()

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

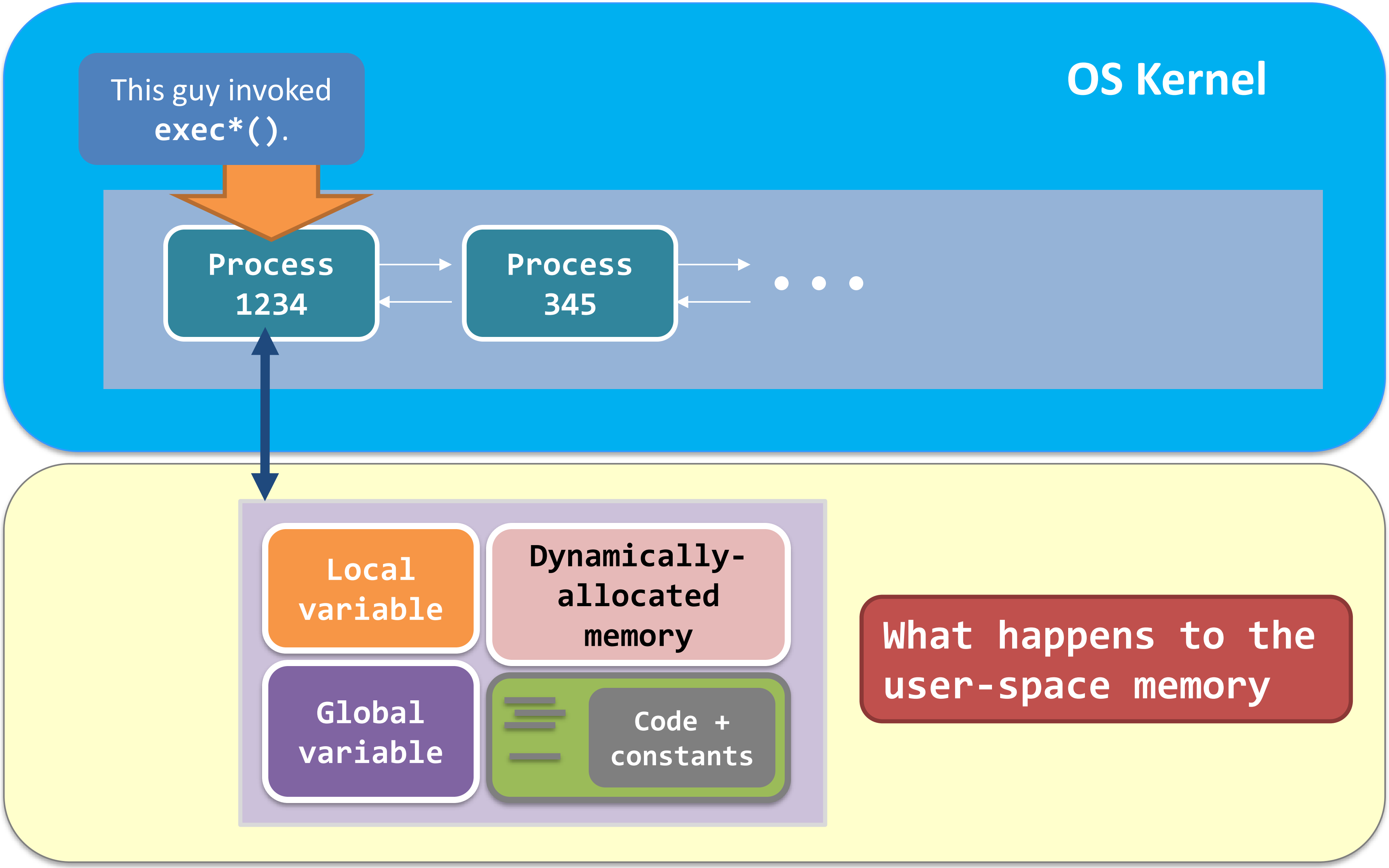
```
e.g., execl("/bin/ls", "/bin/ls", NULL);
```



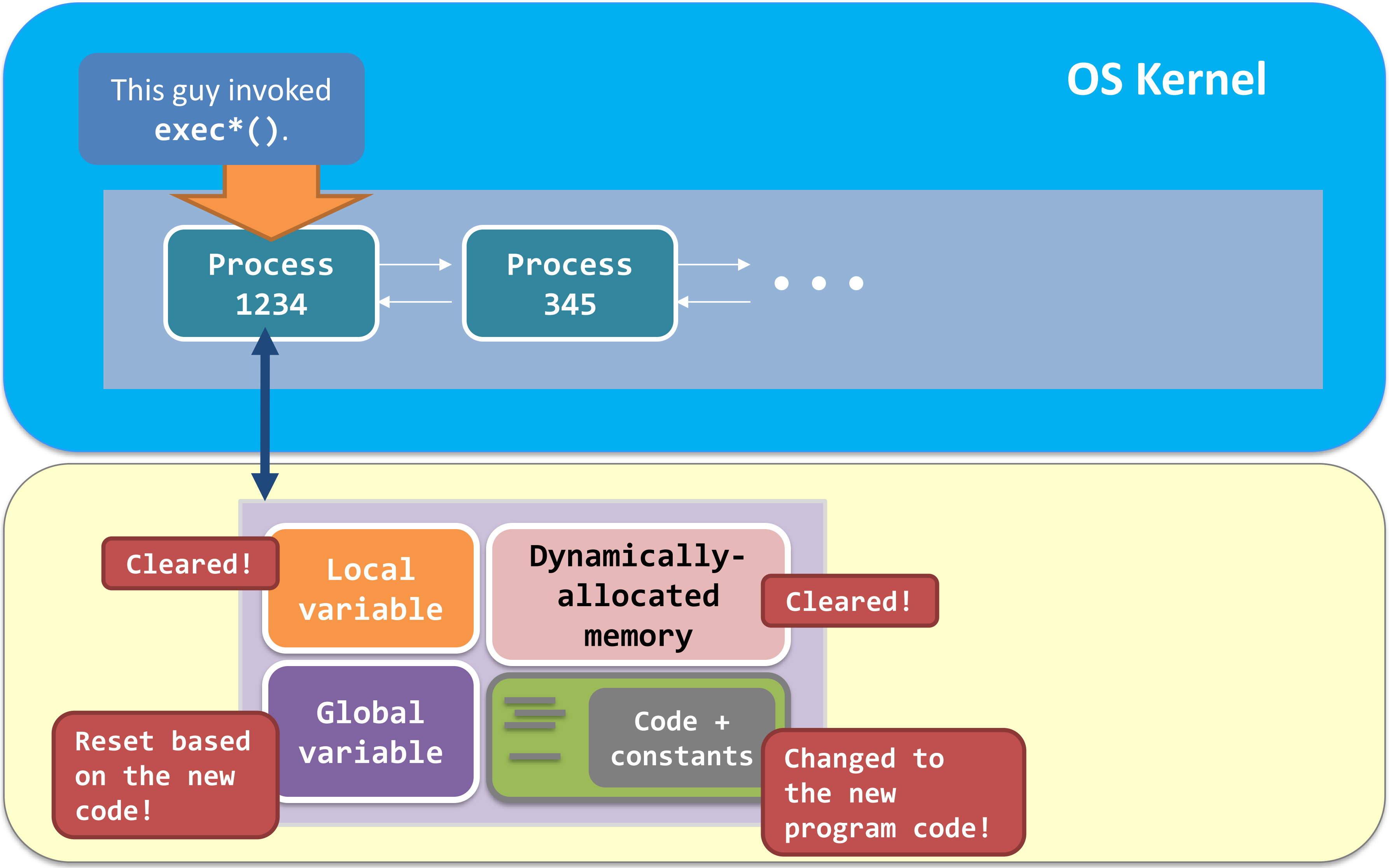
exec*() in action – the start…



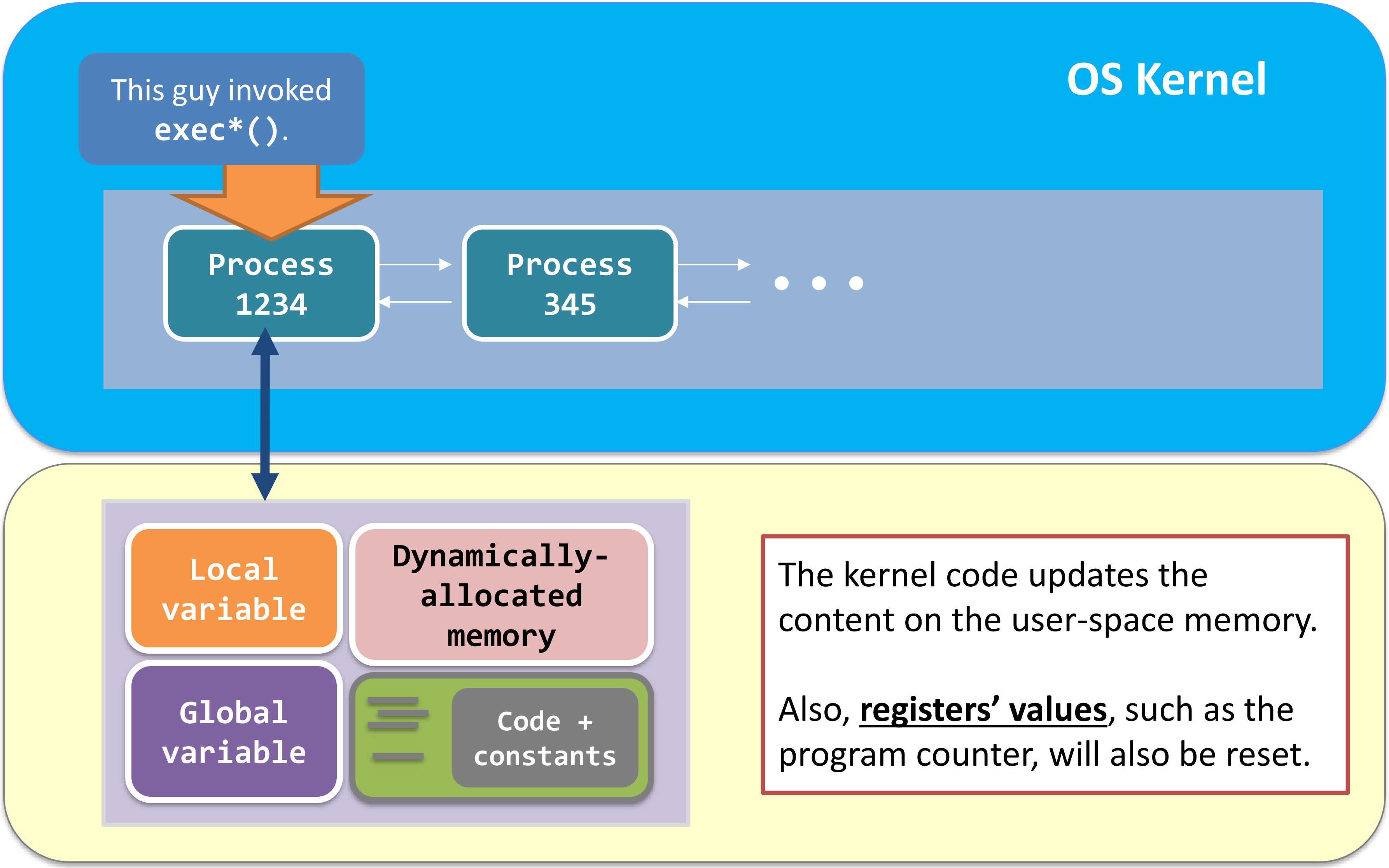
exec*() in action – the end



exec*() in action – the end



exec*() in action – the end

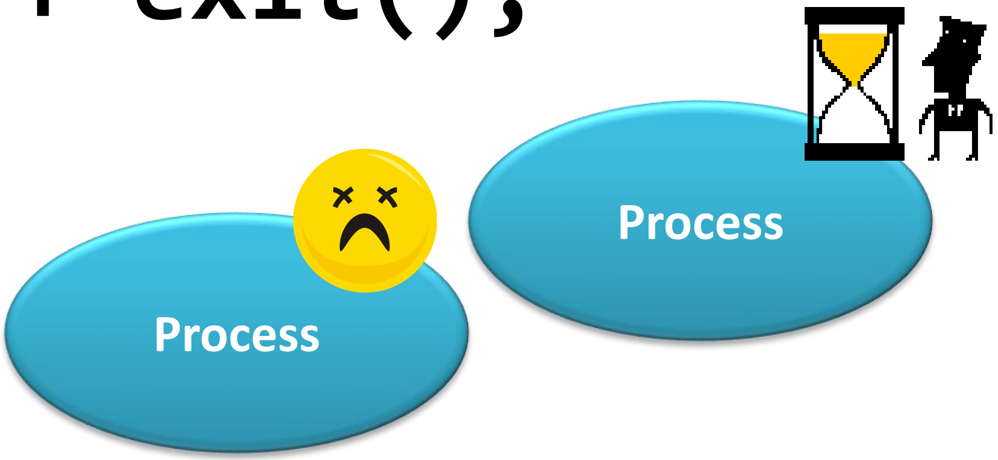


The kernel code updates the content on the user-space memory.

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

Working of system calls

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



Recall the example

```
int system_test(const char *cmd_str) {
    if(cmd_str == -1)
        return -1;
    if(fork() == 0) {
        execl("/bin/sh", "/bin/sh",
              "-c", cmd_str, NULL);
        fprintf(stderr,
           "%s: command not found\n", cmd_str);
        exit(-1);
    }
    wait(NULL);
    return 0;
}

int main(void) {
    printf("before...\n\n");
    system_test("/bin/ls");
    printf("\nafter...\n");
    return 0;
}
```

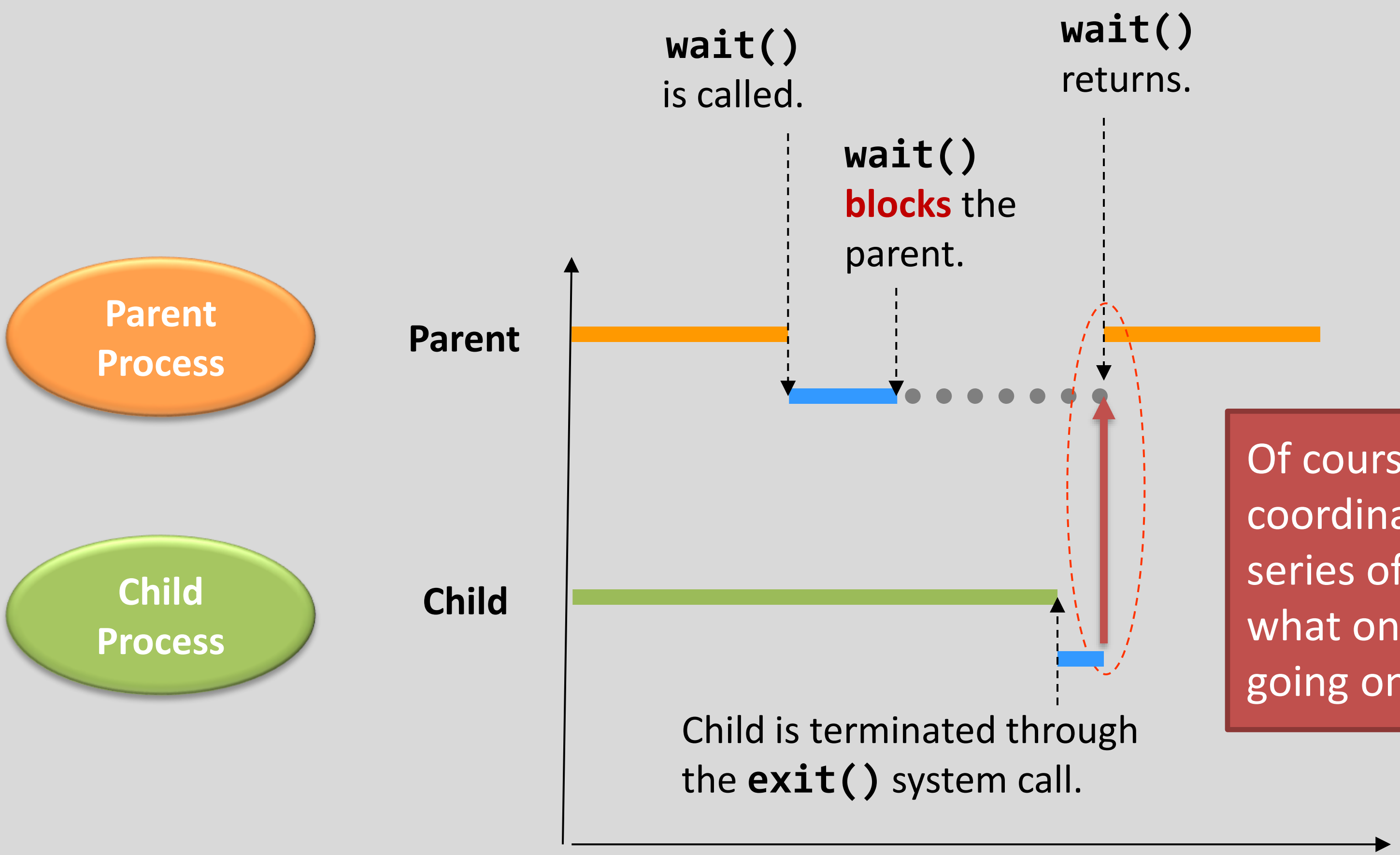
The parent is suspended until the child terminates

```
$ ./system_implement_2
before...

system_implement_2
System_implement_2.c

after...
$ _
```

wait()

- **wait()** system call
  - Suspend the parent process
  - Wake up when one child process terminates
- How to terminate the child process
  - Through the **exit()** system call
- **wait()** and **exit()** – they come together!

`wait()` and `exit()` – Time Analysis

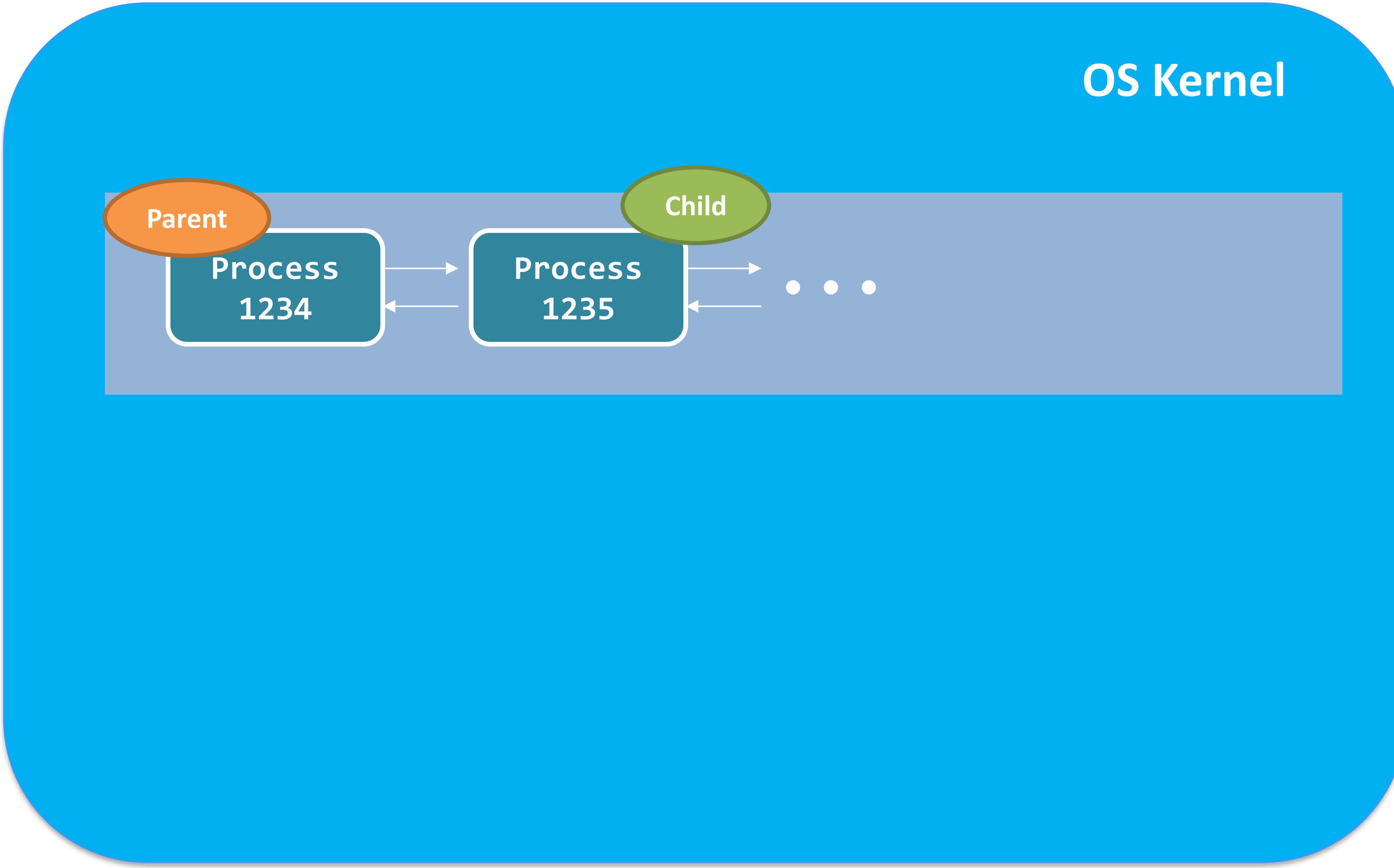


Of course, the kernel coordinates the series of events. But, what on earth is going on?

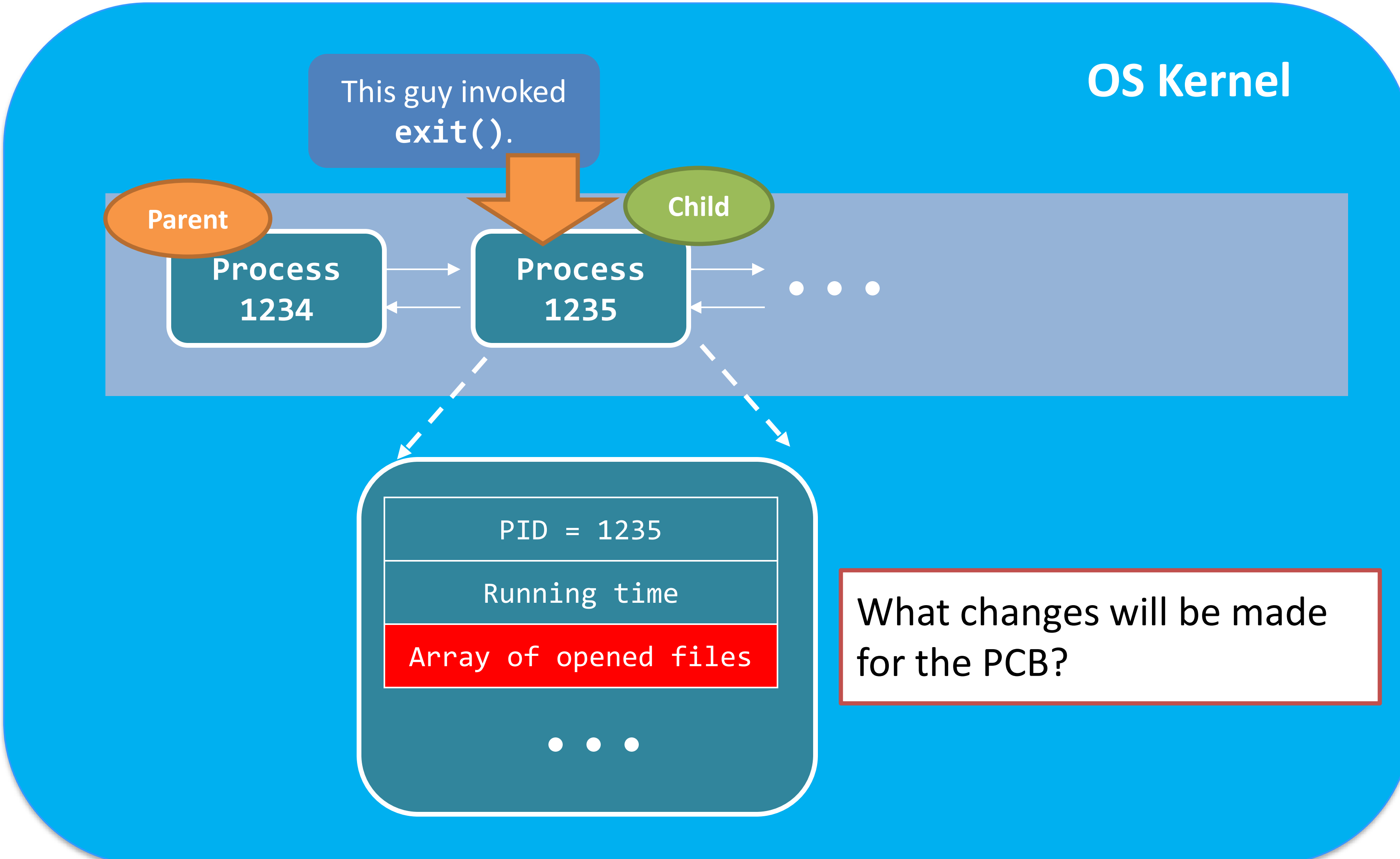
Guess…

- What is going on inside kernel?
  - Child: **`exit()`**
    - Process data + PCB
  - Parent: **`wait()`**
    - Process data + PCB

`wait()` and `exit()` – child side

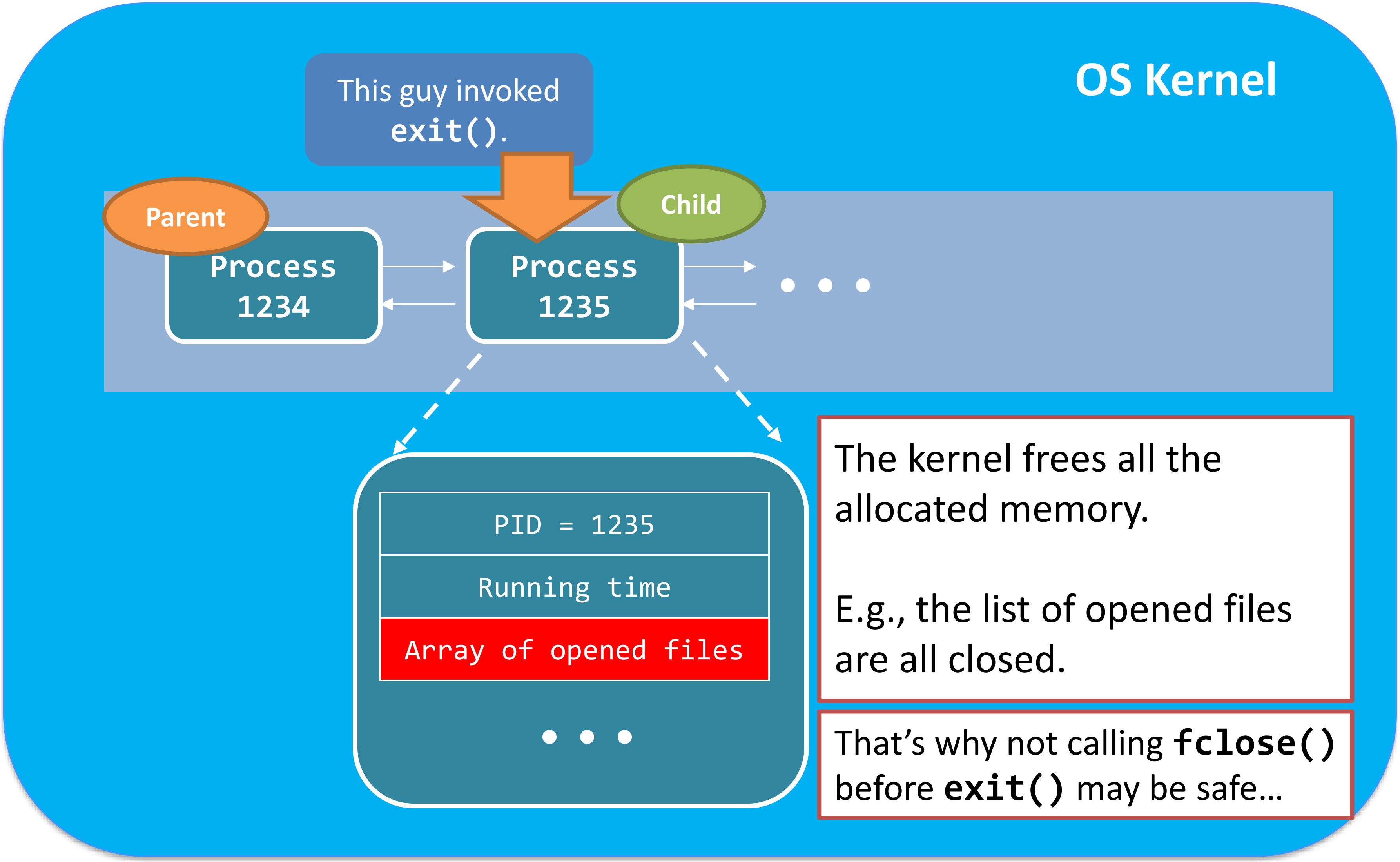


`wait()` and `exit()` – child side



What changes will be made for the PCB?

`wait()` and `exit()` – child side

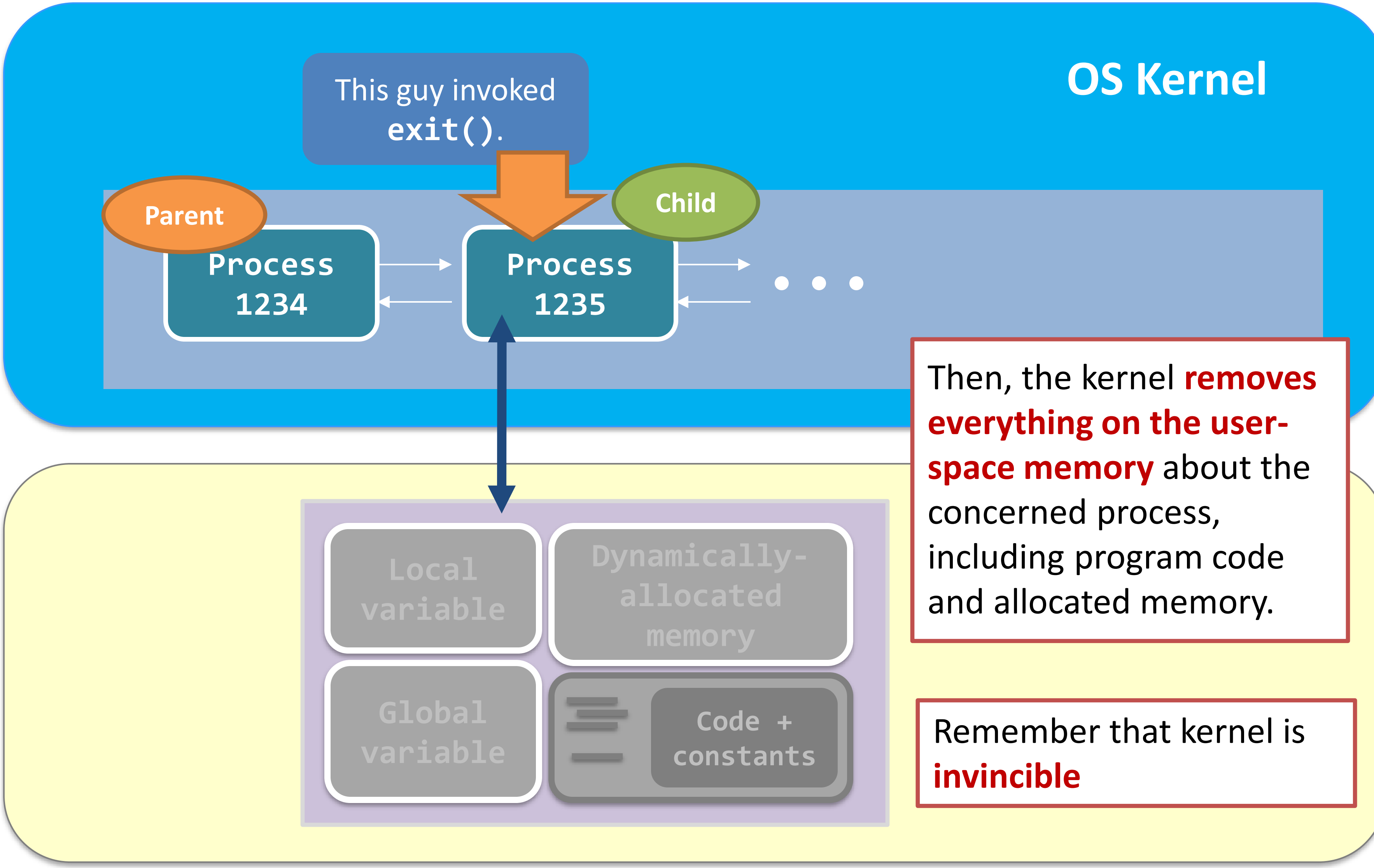


The kernel frees all the allocated memory.

E.g., the list of opened files are all closed.

That's why not calling **`fclose()`** before **`exit()`** may be safe…

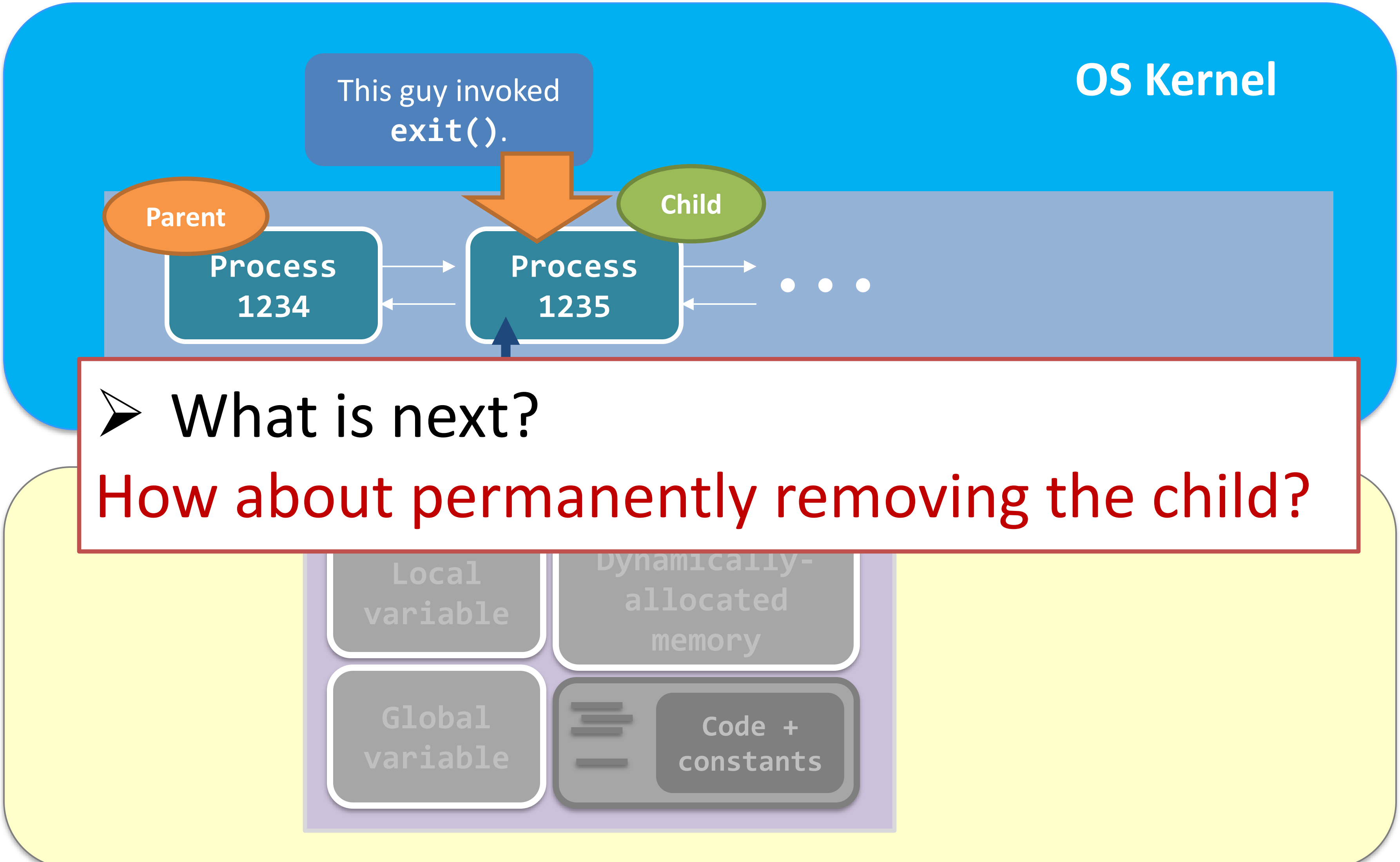
`wait()` and `exit()` – child side



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

Remember that kernel is **invincible**

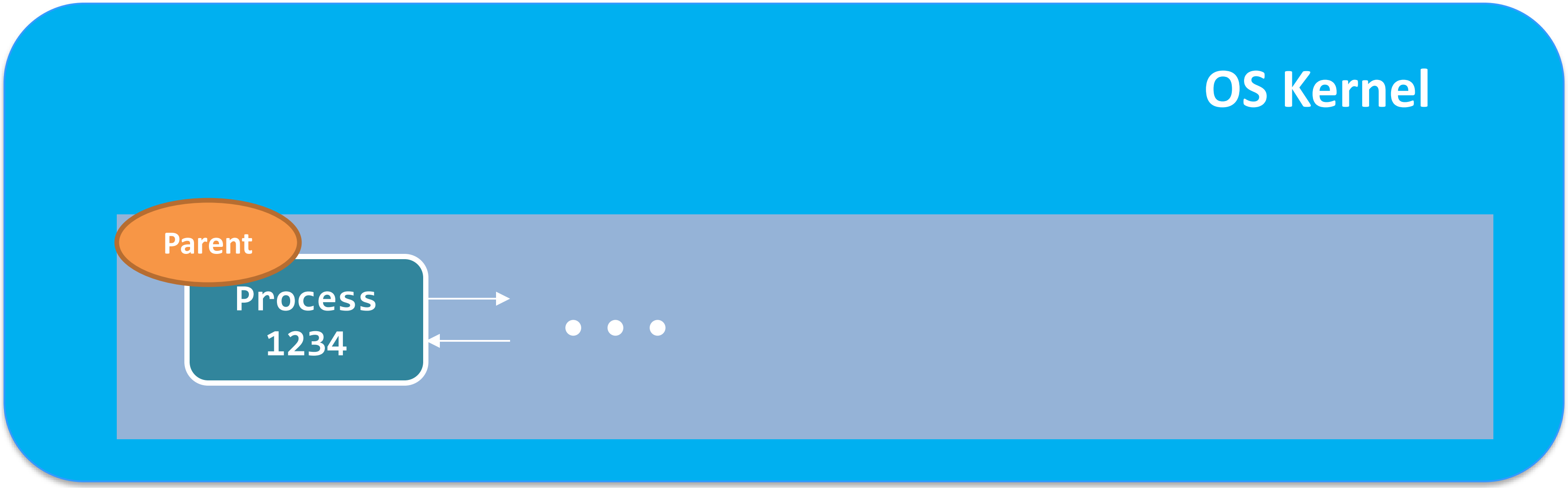
`wait()` and `exit()` – child side



- What is next?

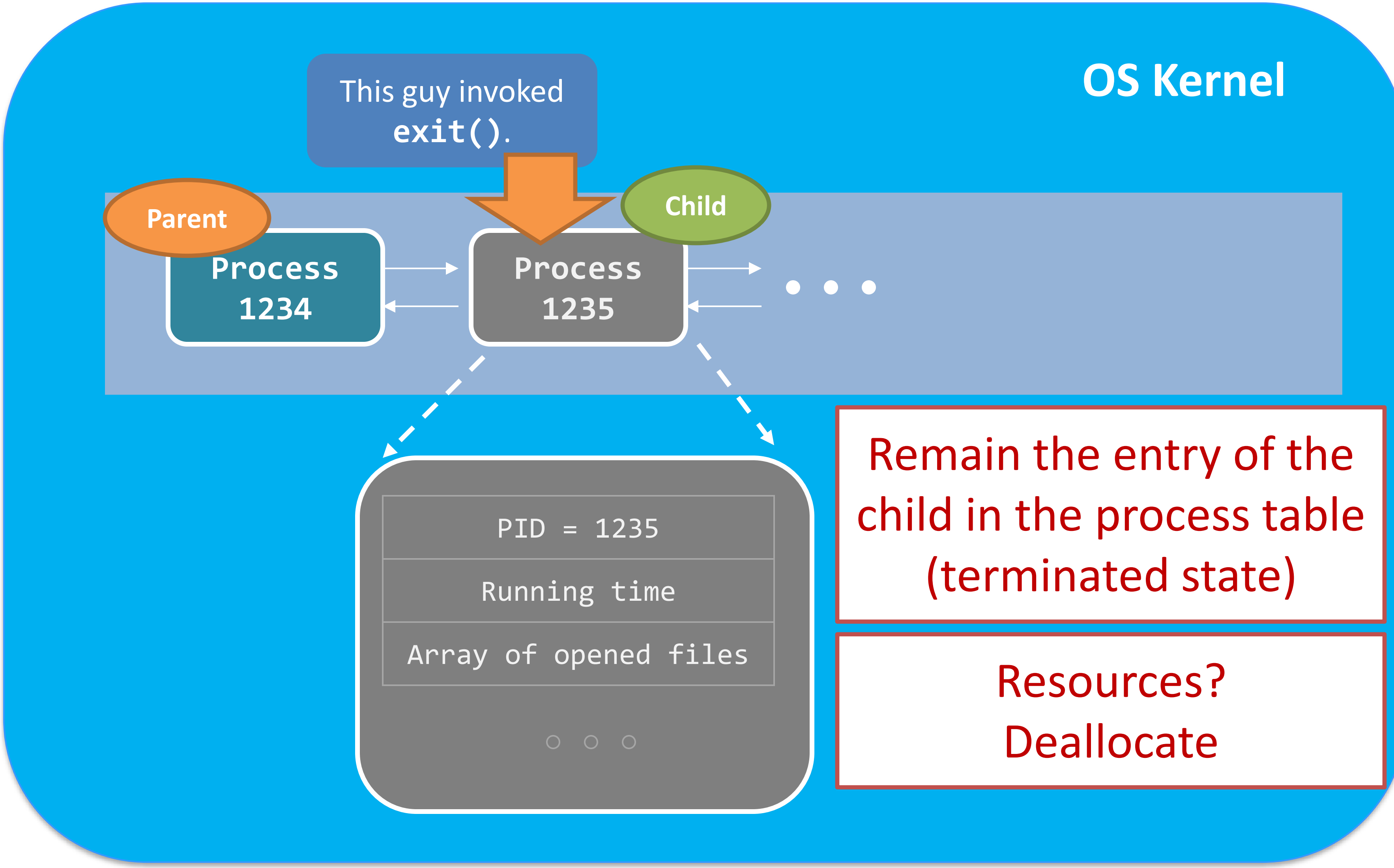
How about permanently removing the child?

`wait()` and `exit()` – child side

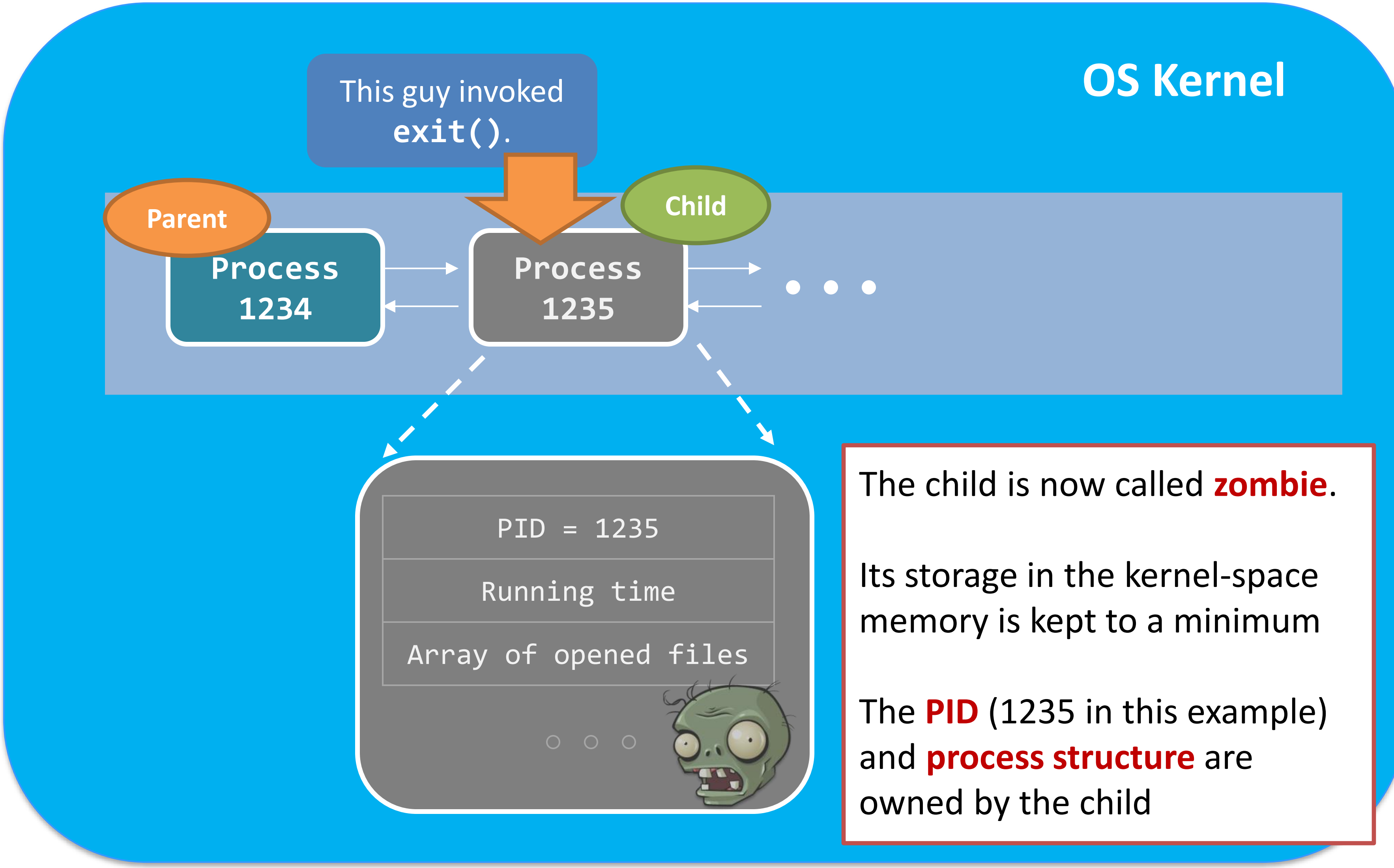


Removed from the process table immediately?
Not really! Why?

`wait()` and `exit()` – child side



`wait()` and `exit()` – child side

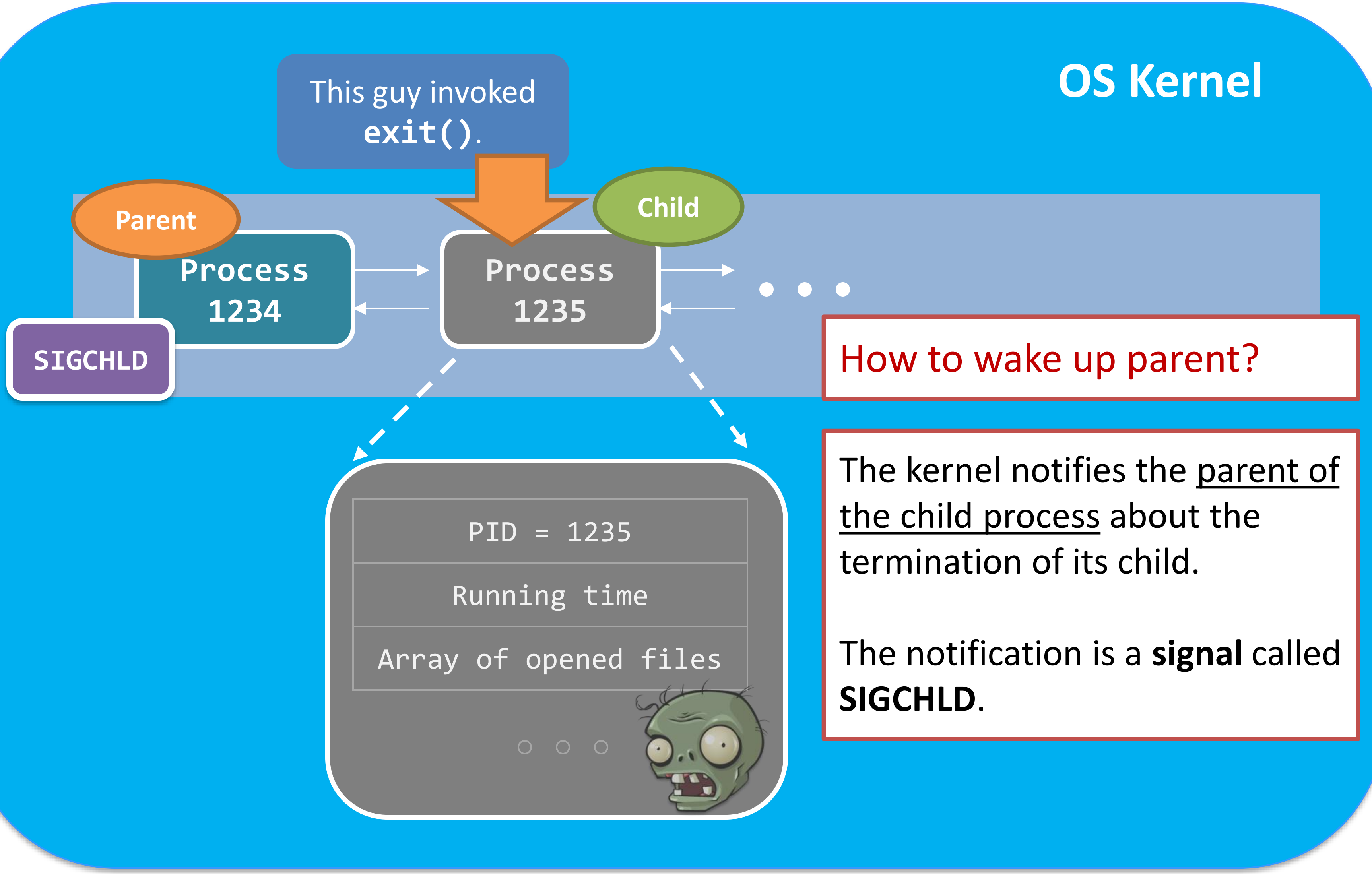


The child is now called **zombie**.

Its storage in the kernel-space memory is kept to a minimum

The **PID** (1235 in this example) and **process structure** are owned by the child

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



How to wake up parent?

The kernel notifies the <u>parent of the child process</u> about the termination of its child.

The notification is a **signal** called **SIGCHLD**.

Signal

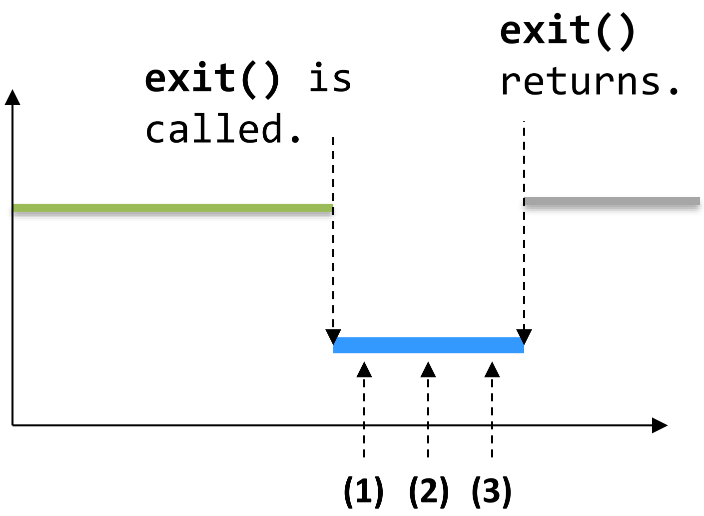
- What is signal?
  - A software interrupt
  - It takes steps as in the hardware interrupt
- Two kinds of signals
  - Generated from user space
    - **Ctrl+C**, **kill()** system call, etc.
  - Generated from kernel and CPU
    - Segmentation fault (**SIGSEGV**), Floating point exception (**SIGFPE**), child process termination (**SIGCHLD**), etc.
- Signal is very hard to master, will be skipped in this course
  - Reference: Advanced Programming Environment in UNIX
  - Linux manpage

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.

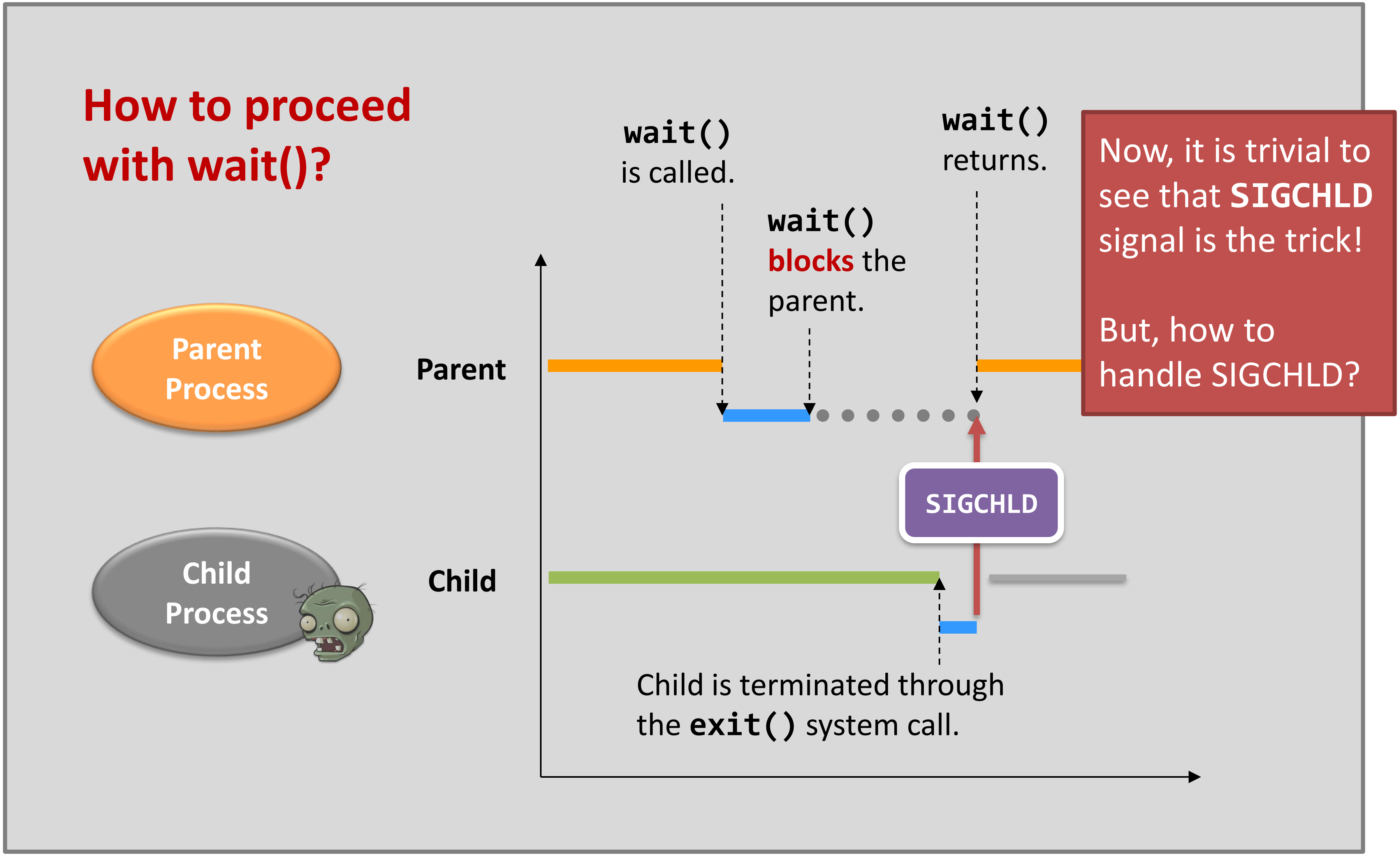


Although the child is still in the system, it is no **longer running**. There is no program code!!!

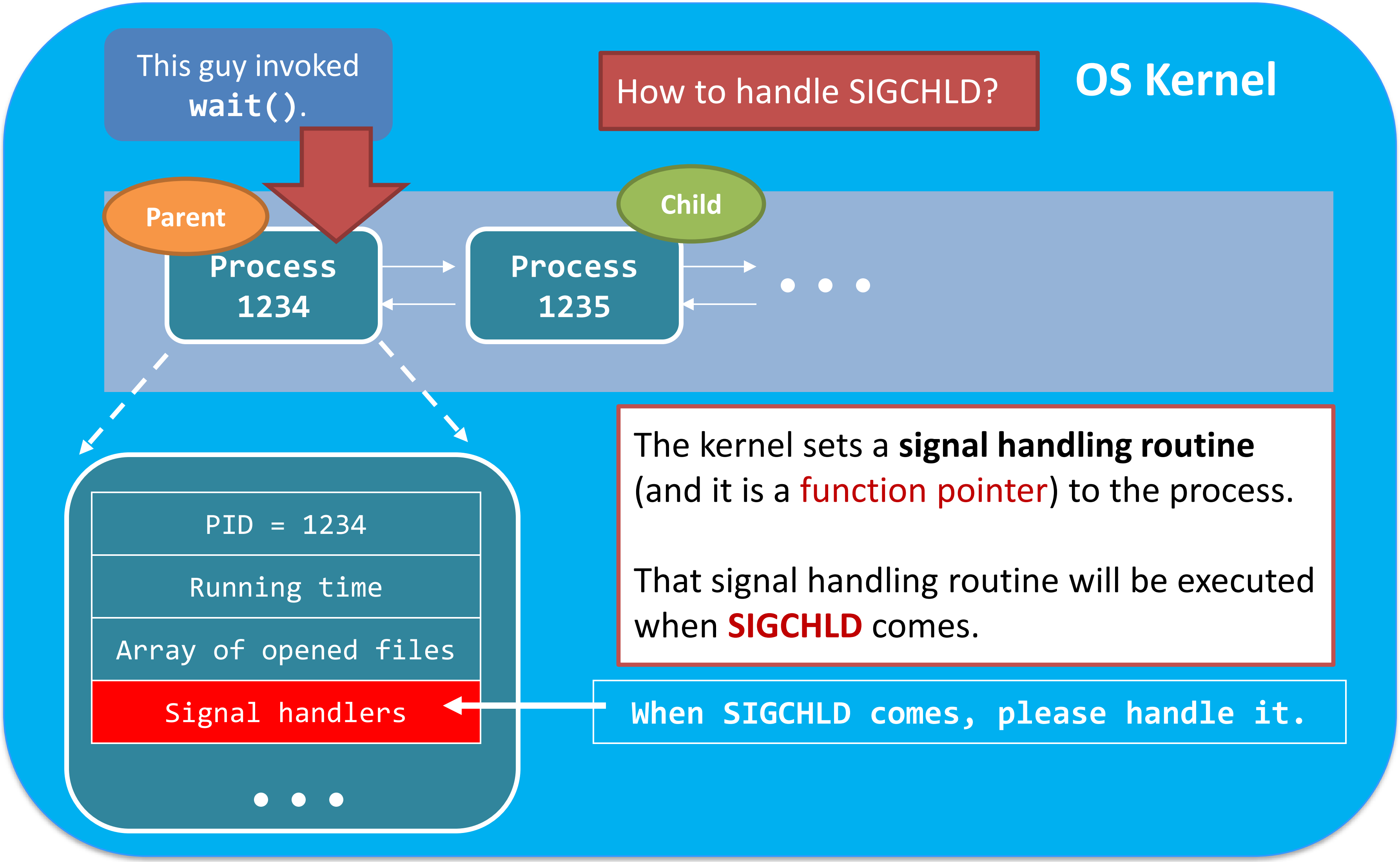
It turns into a **mindless zombie**…

You cannot kill a zombie process, as it is already dead. Then how to eliminate it?

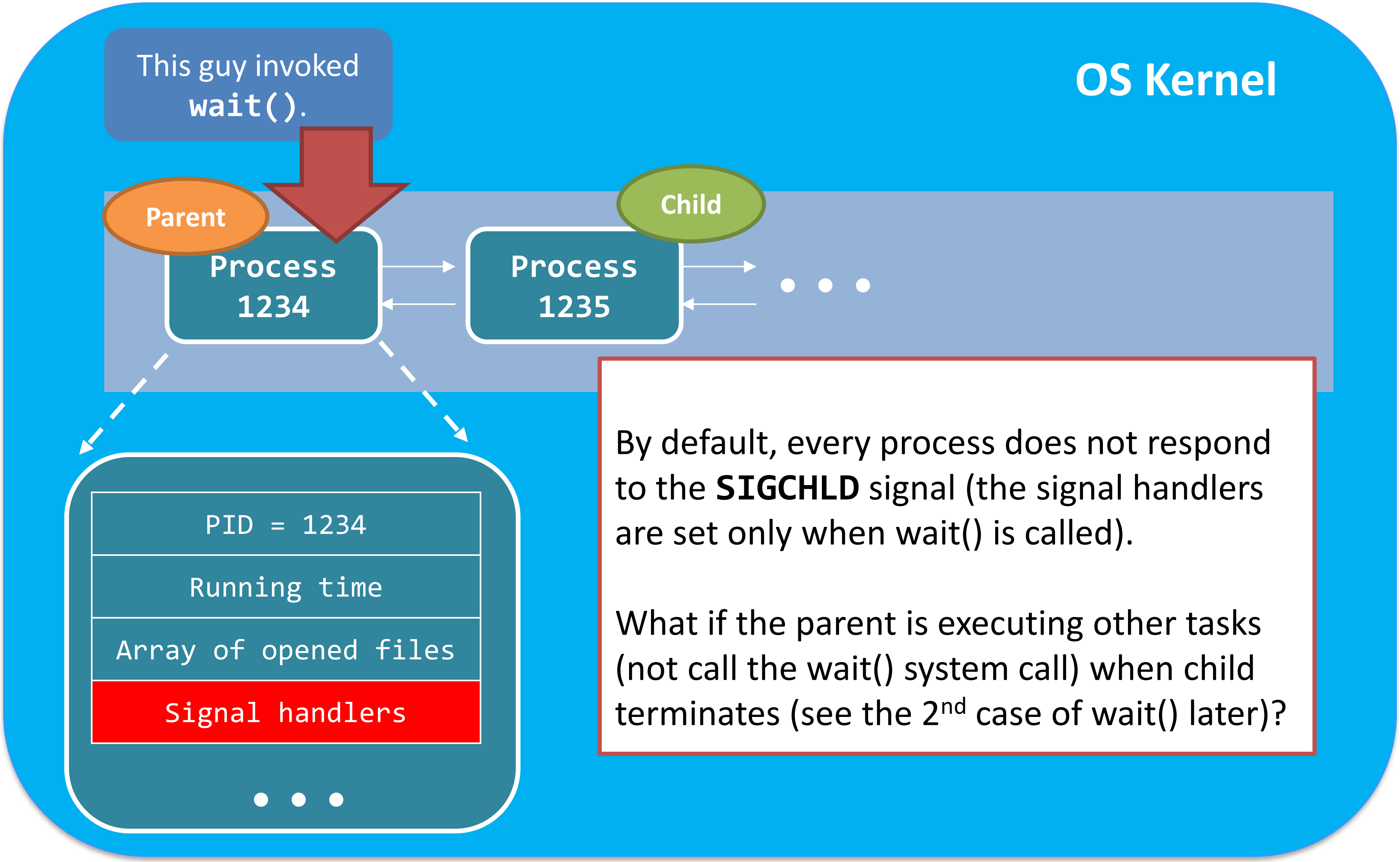
`wait()` and `exit()` – they come together!



`wait()` and `exit()` – parent side



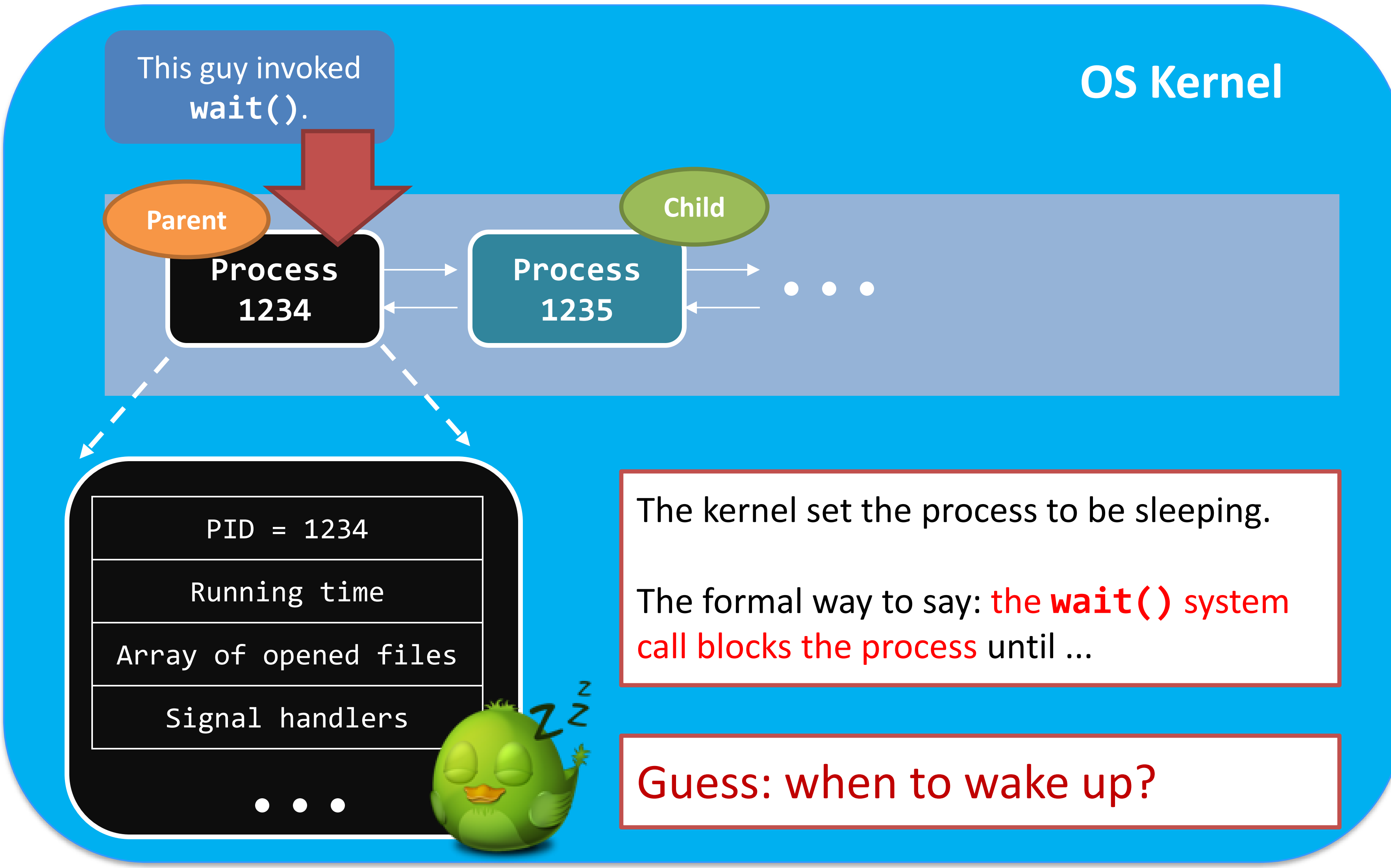
wait() and exit() – parent side



By default, every process does not respond to the **SIGCHLD** signal (the signal handlers are set only when wait() is called).

What if the parent is executing other tasks (not call the wait() system call) when child terminates (see the 2nd case of wait() later)?

`wait()` and `exit()` – parent side

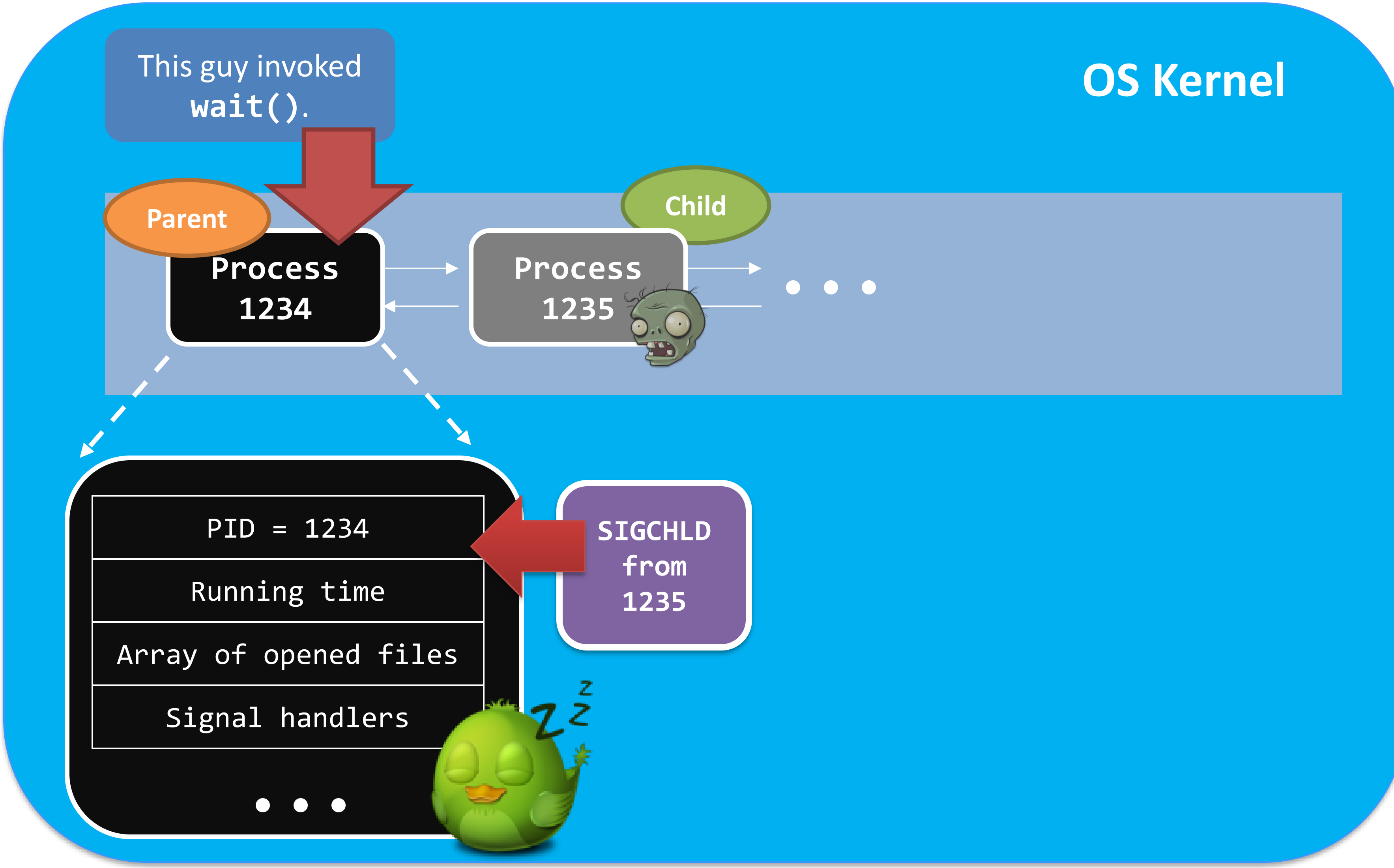


The kernel set the process to be sleeping.

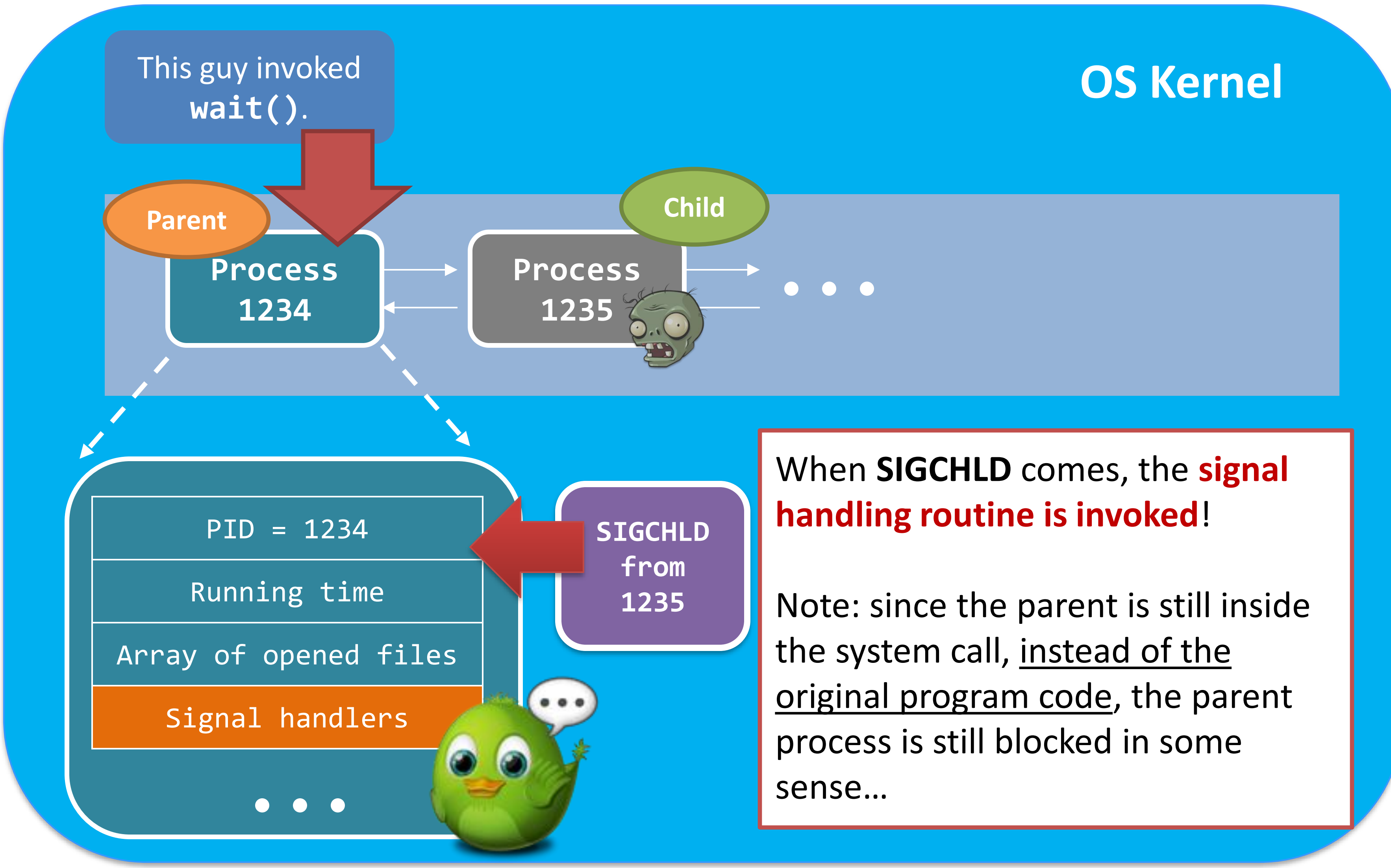
The formal way to say: **the `wait()` system call blocks the process** until ...

Guess: when to wake up?

`wait()` and `exit()` – parent side



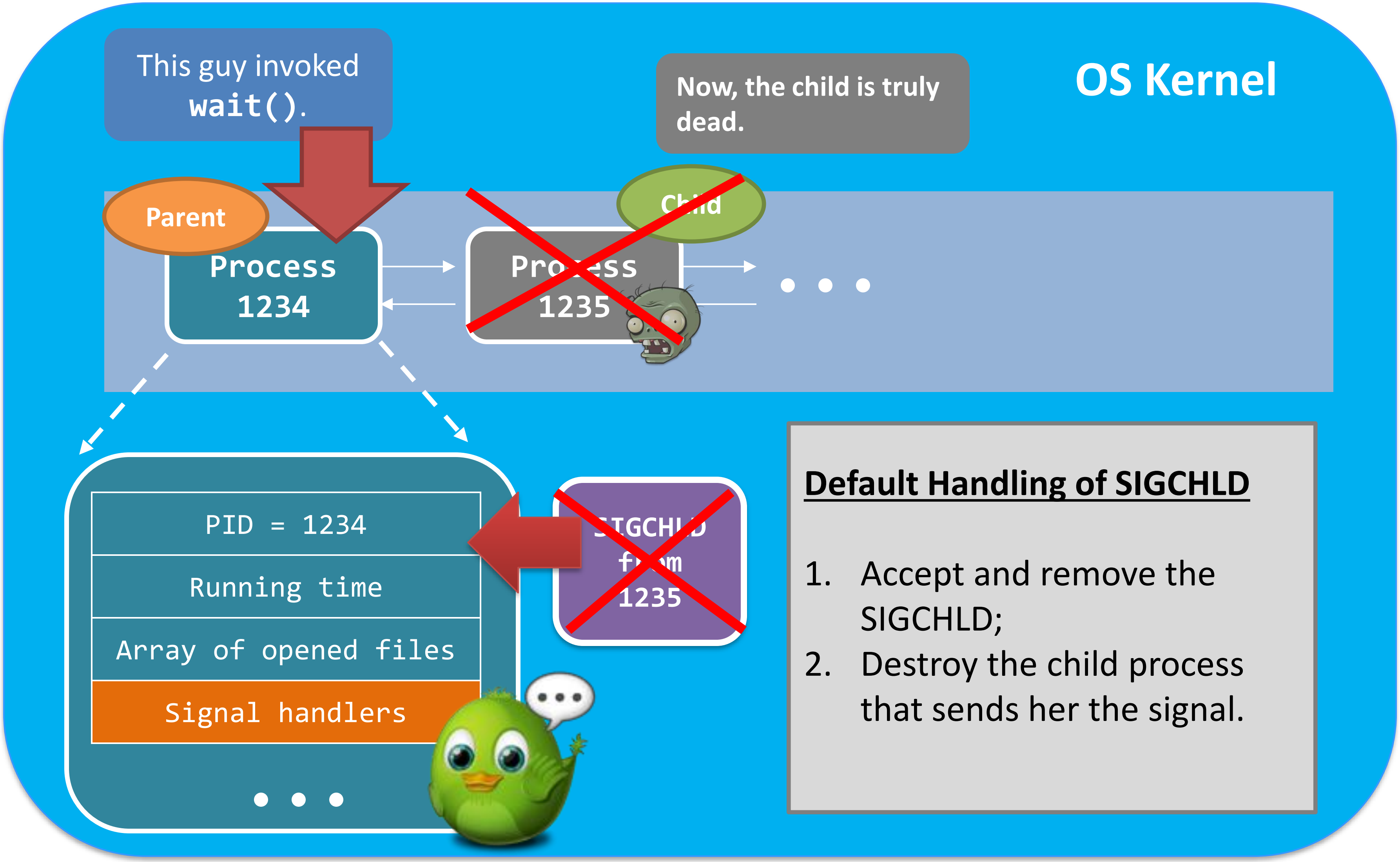
`wait()` and `exit()` – parent side



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

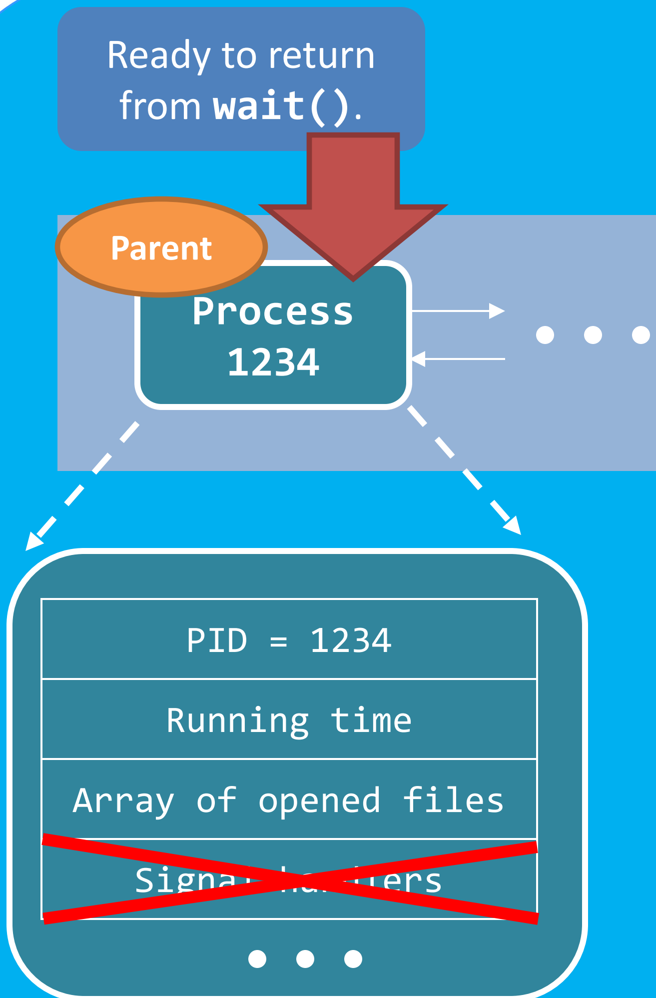
Note: since the parent is still inside the system call, <u>instead of the original program code</u>, the parent process is still blocked in some sense…

wait() and exit() – parent side



`wait()` and `exit()` – parent side

OS Kernel



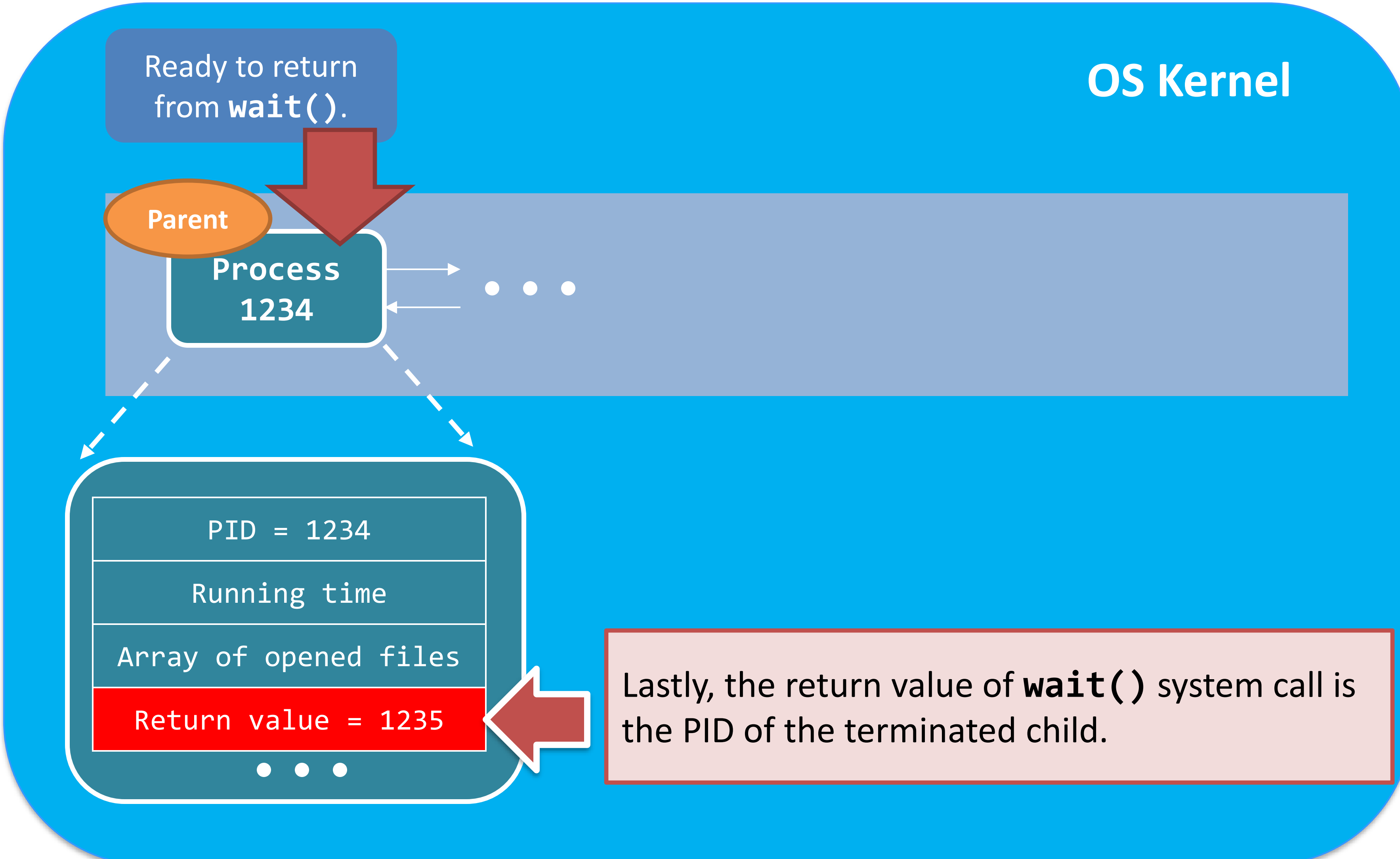
The signal handler is then removed, i.e., <u>the process is ignoring **SIGCHLD** again</u>.

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

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

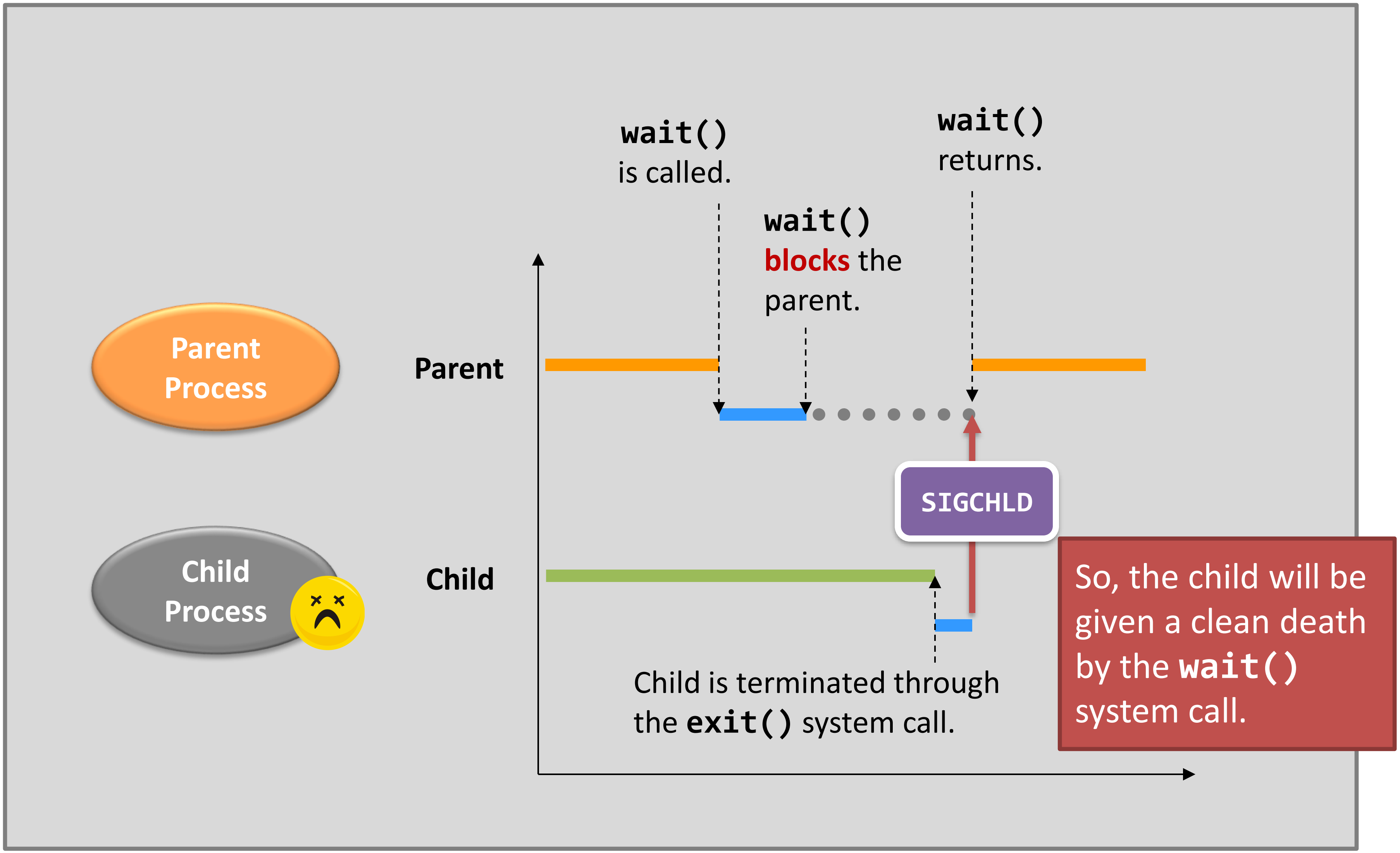
This is the reason why wait() system call waits for any one of the child processes.

`wait()` and `exit()` – parent side



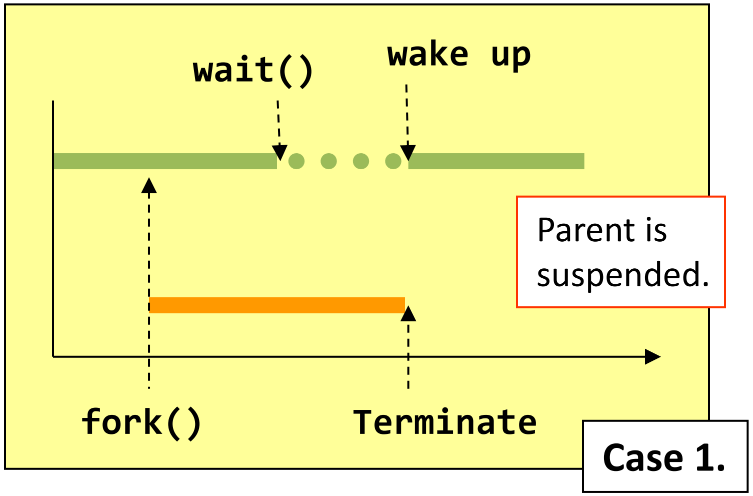
Lastly, the return value of **`wait()`** system call is the PID of the terminated child.

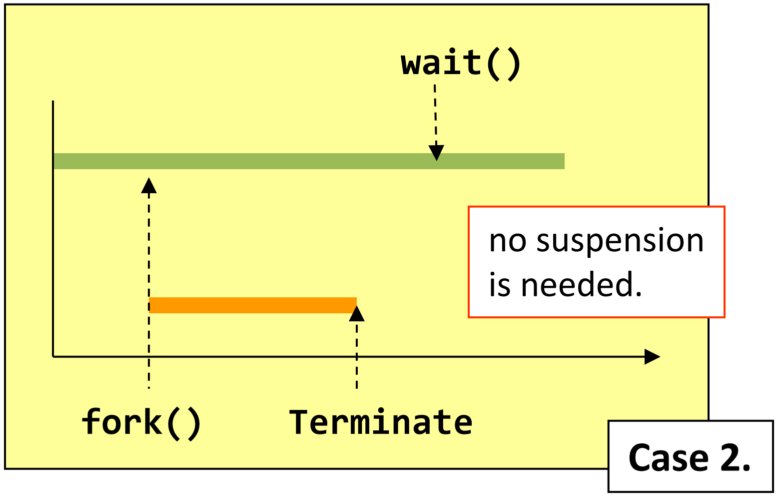
`wait()` and `exit()` – parent side



Is it done?

- How about `wait()` is called after the child already terminated?
  - Remember the case 2 (which is safe)

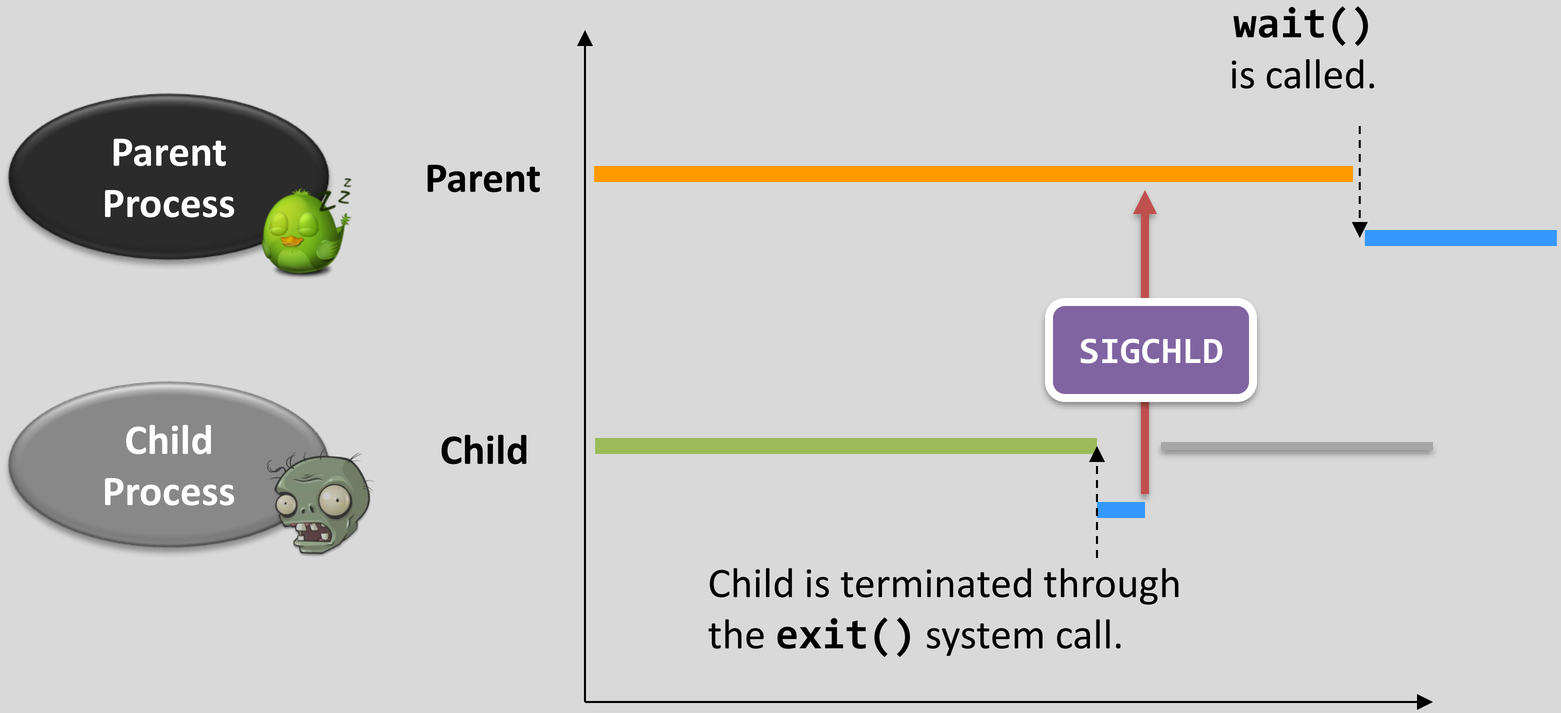




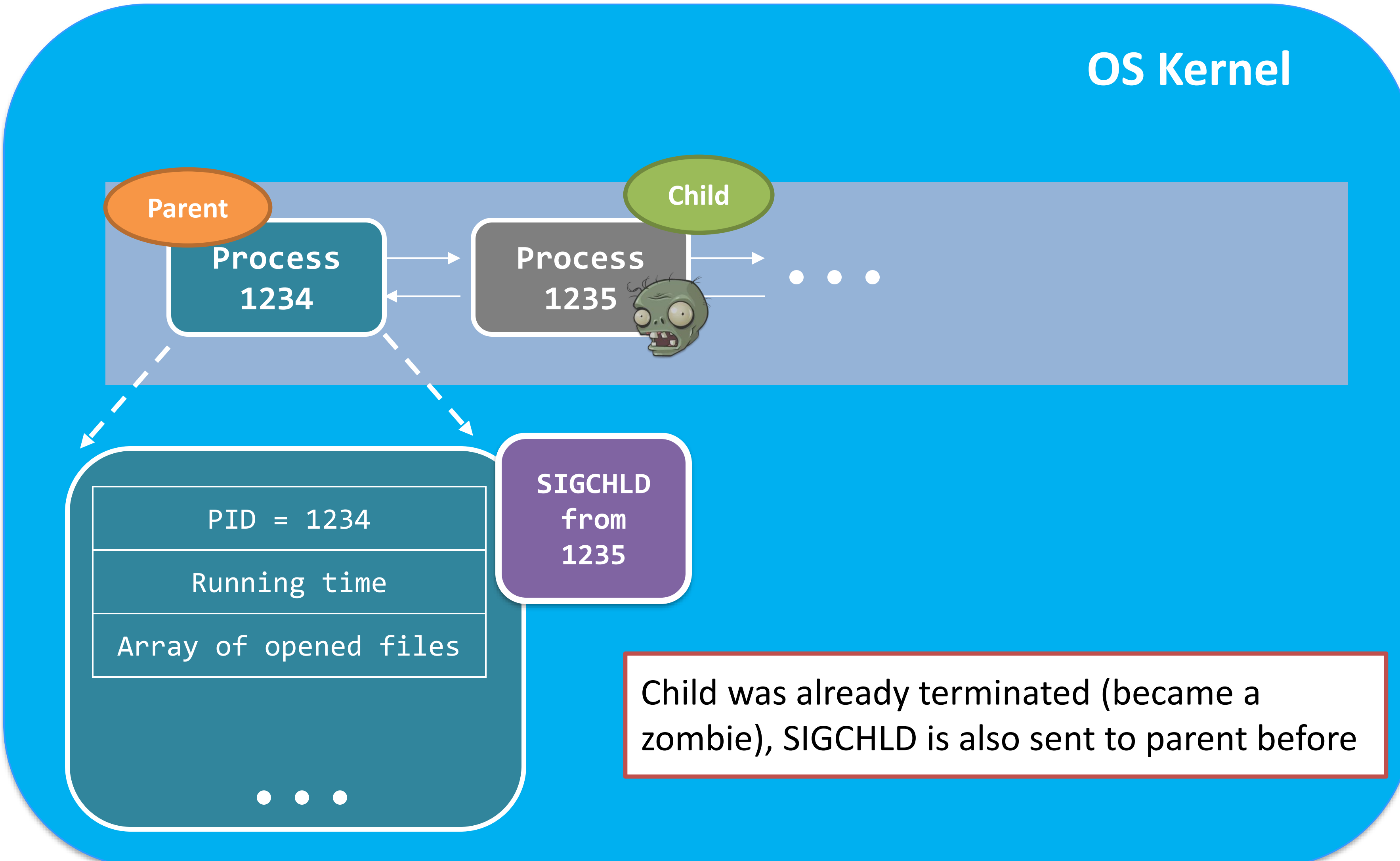
wait() and exit() – parent side

Case 2.

What is going on inside the kernel?

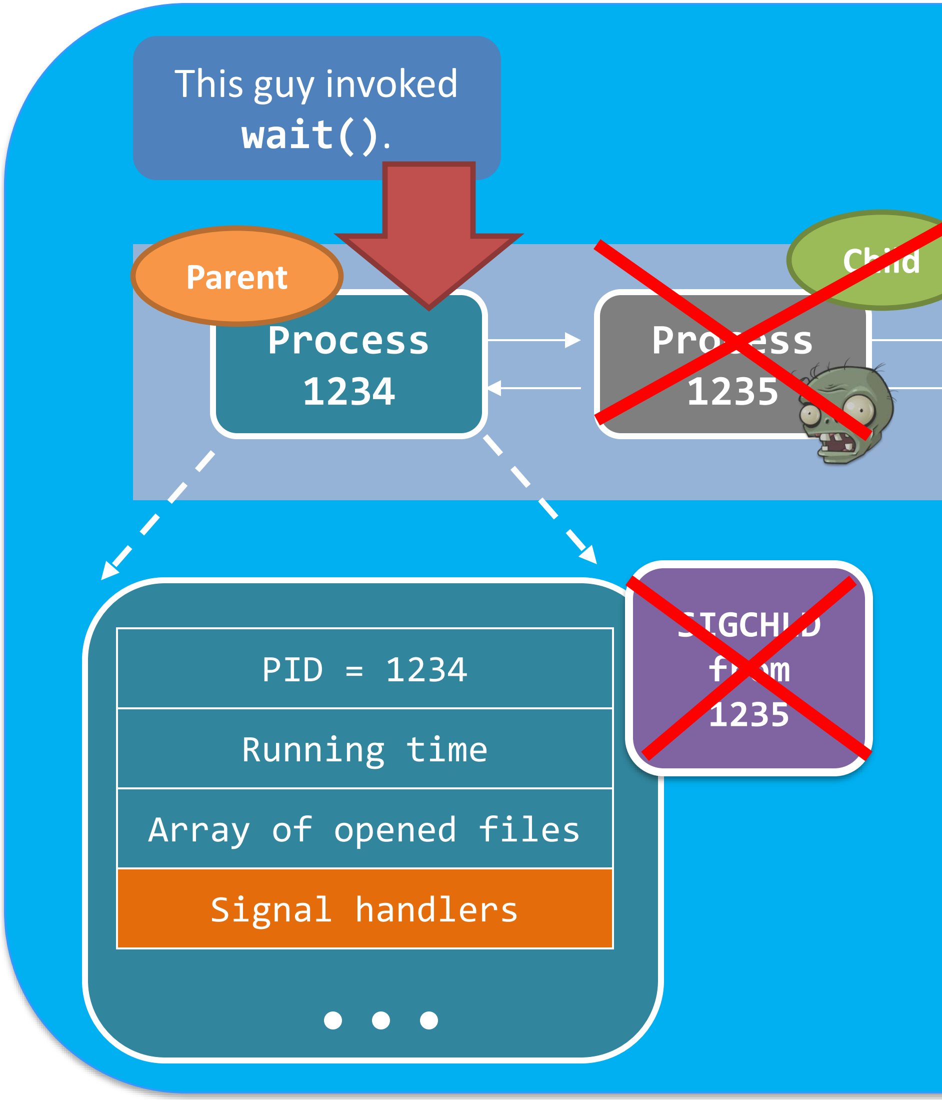


wait() and exit() – parent side



Child was already terminated (became a zombie), SIGCHLD is also sent to parent before

wait() and **exit()** – parent side

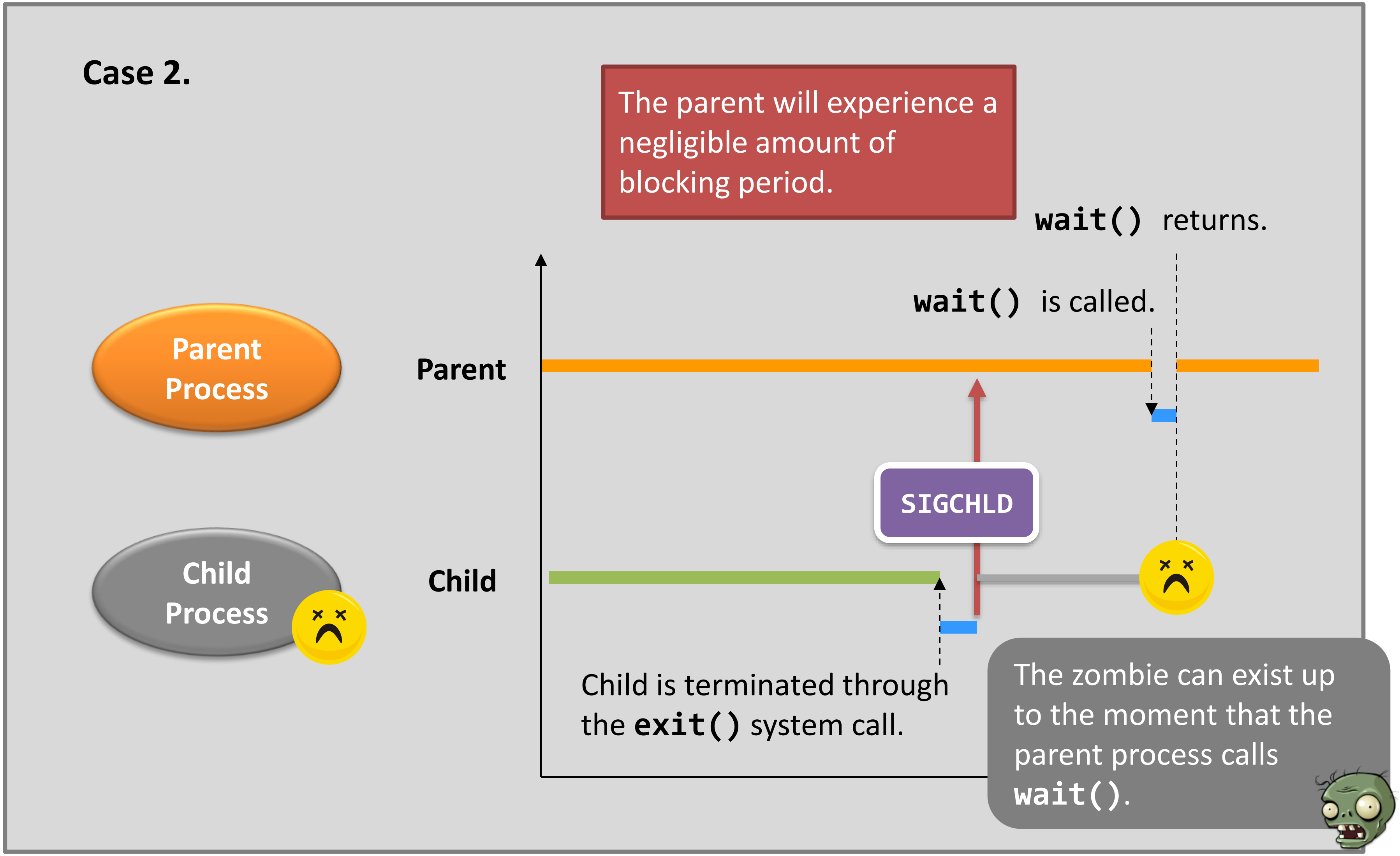


Similar to Case 1, the kernel sets the signal handling routine...

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



Orphans (zombies)

- What would happen if a parent did not invoke `wait()` and terminated?
  - Remember the reparent operation in Linux?

- `init` is the new parent, and it **periodically** invokes `wait()`

`wait()` and **`exit()`** – short summary

- A process is turned 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.
- Remember why **`exec*()`** does not return to its calling process in previous example…

`wait()` and **`exit()`** – short summary

- **`wait()`** is to reap zombie child processes
  - You should never leave any zombies in the system.
- Linux will label zombie processes as "**`<defunct>`**".
  - To look for them: **`ps aux | grep defunct`**
- Learn **`waitpid()`** by yourself…

wait() and exit() – Example

```
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;
}
```

What is the purpose of this program?

wait() and exit() – Example

```
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;
}
```

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”.

Working of system calls

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

The role of `wait()` in the OS…

- Why 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`**”
  - What will happen if we don’t clean up the zombies?

When `wait()` is absent…

- What is the result of this program?
  - Do not try to know the result by running it

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

<u>**Think about what will be happened to both parent and child processes?**</u>

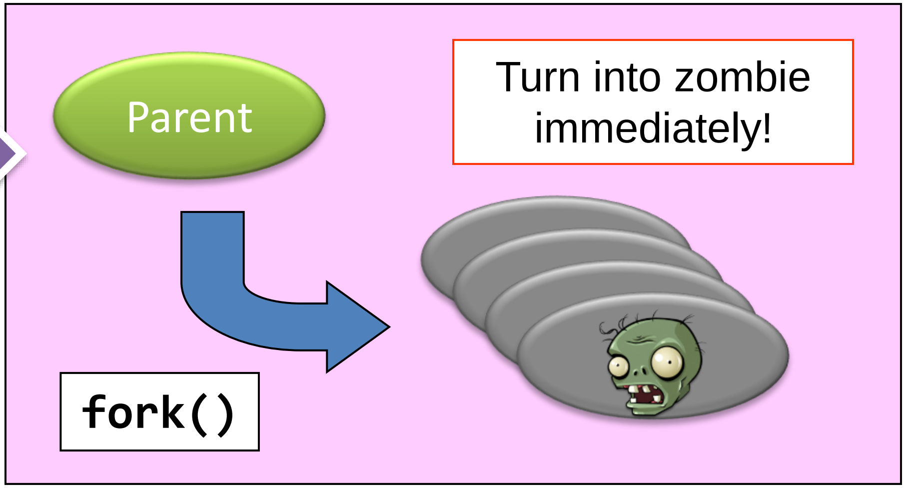
When **`wait()`** is absent…

- Don't try this…

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

Parent: never reach here.

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



An infinite, zombie factory!

Summary

- Process concept
  - Process vs program
  - User-space memory + PCB
- Process operations
  - Creation, program execution, termination
  - The internal workings of
    - **`fork()`**
    - **`exec*()`**
    - **`wait()+exit()`:** come together
- Calling **`wait()`** is important