

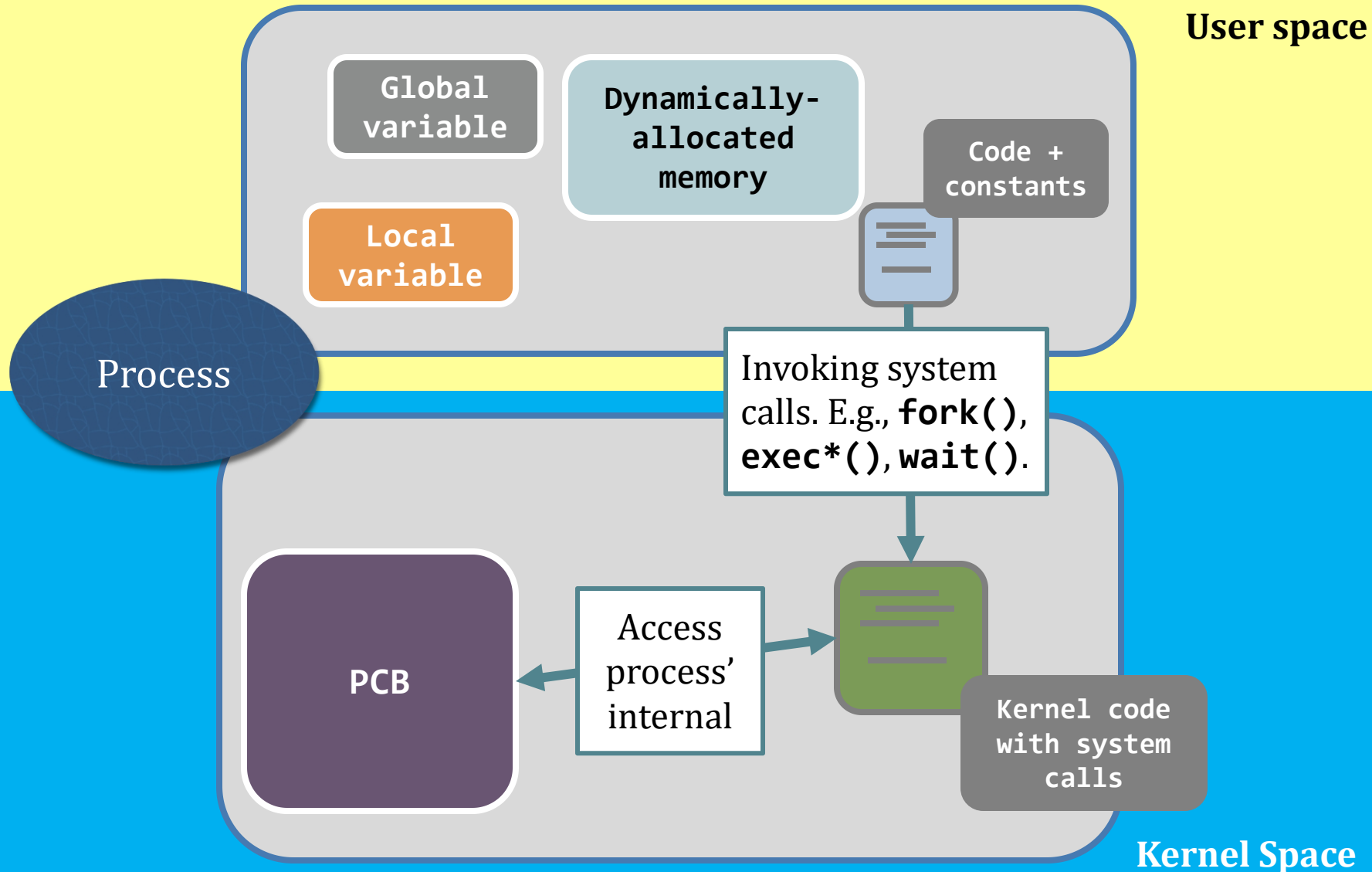
# Lecture 4: Process II

---

Yinqian Zhang@ 2021, Spring

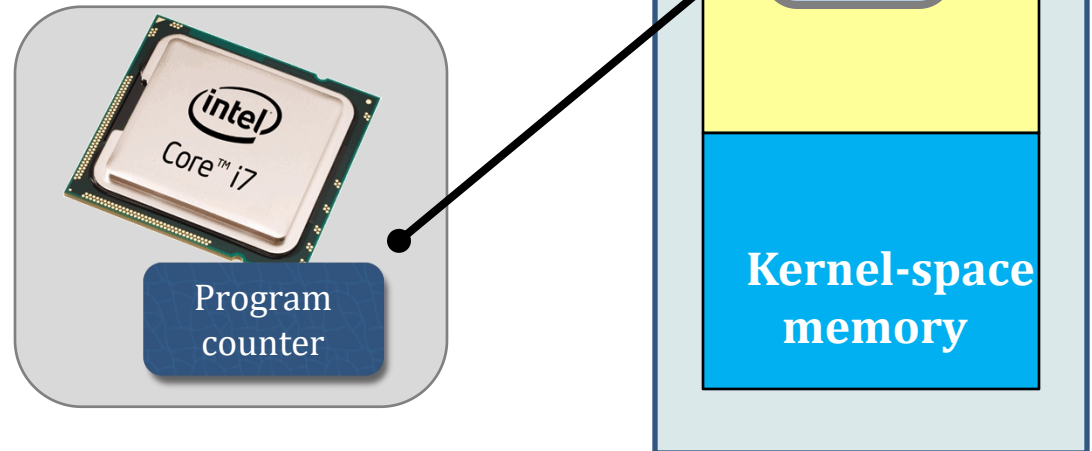
copyright@Bo Tang

# The story so far...



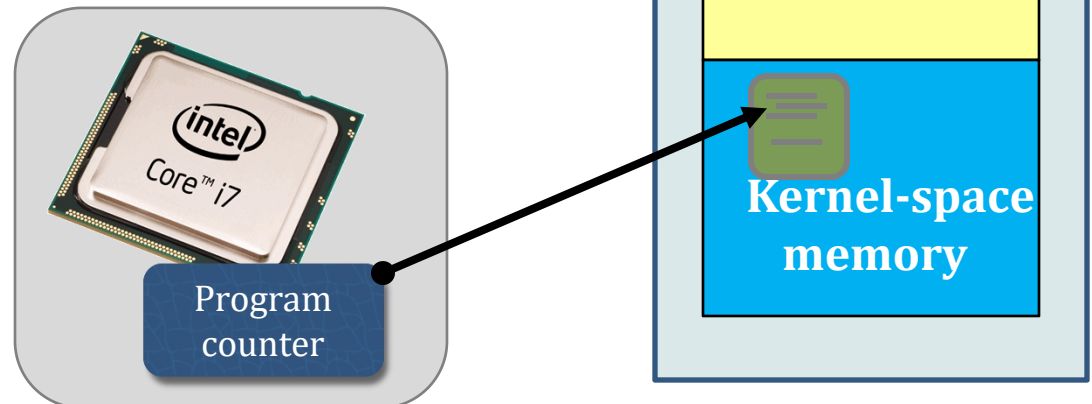
# When invoking a system call (memory view)

- ✧ 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.



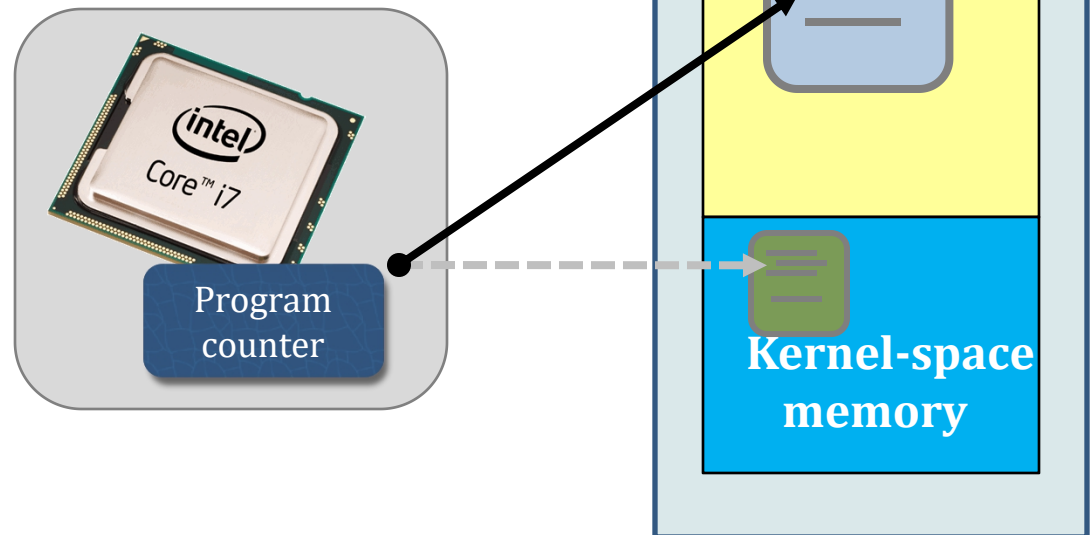
# When invoking a system call (memory view)

- ✧ 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.

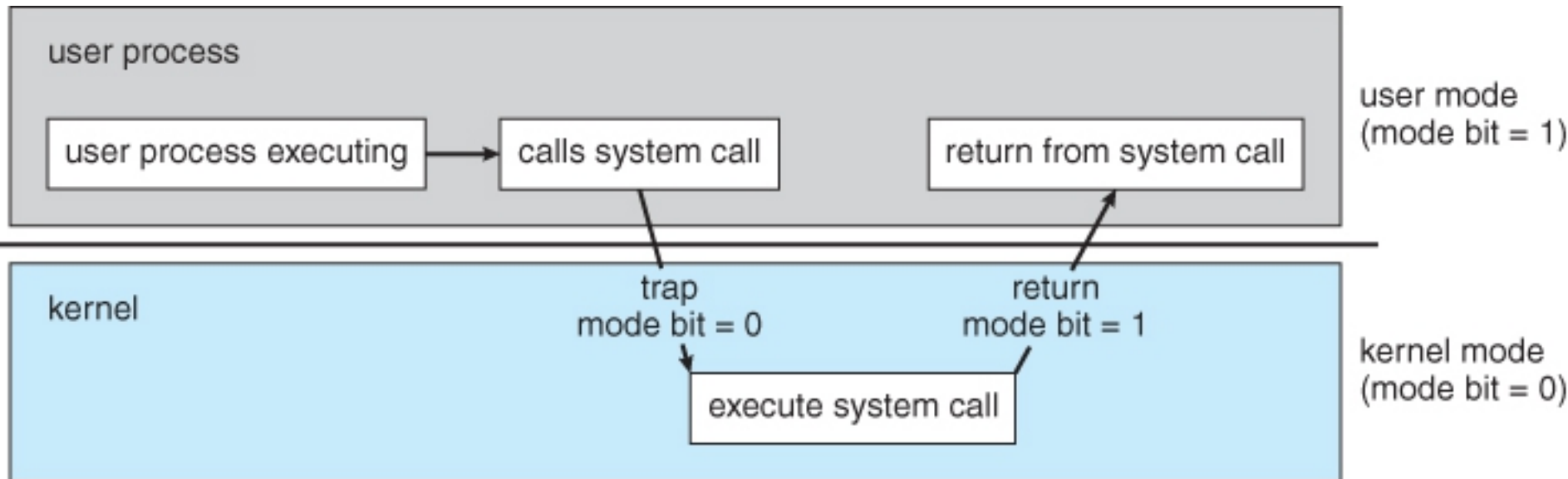


# When invoking a system call (memory view)

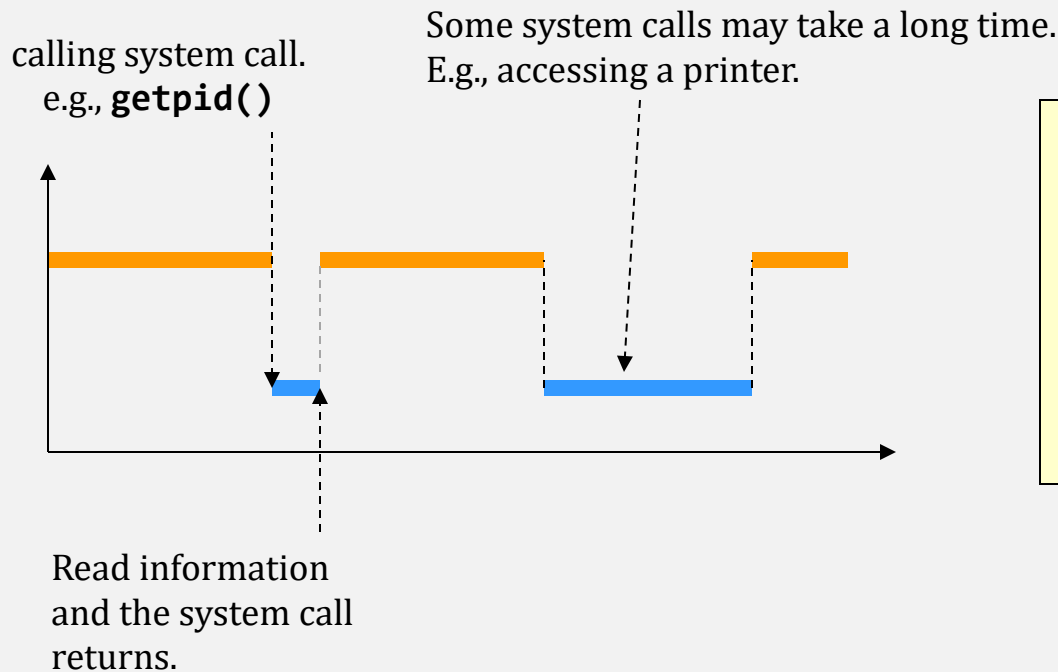
- ✧ When the CPU has finished executing the “**getpid()**” system call
  - ✧ it switches back to the user-space memory, and continues running that program code.



# 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)

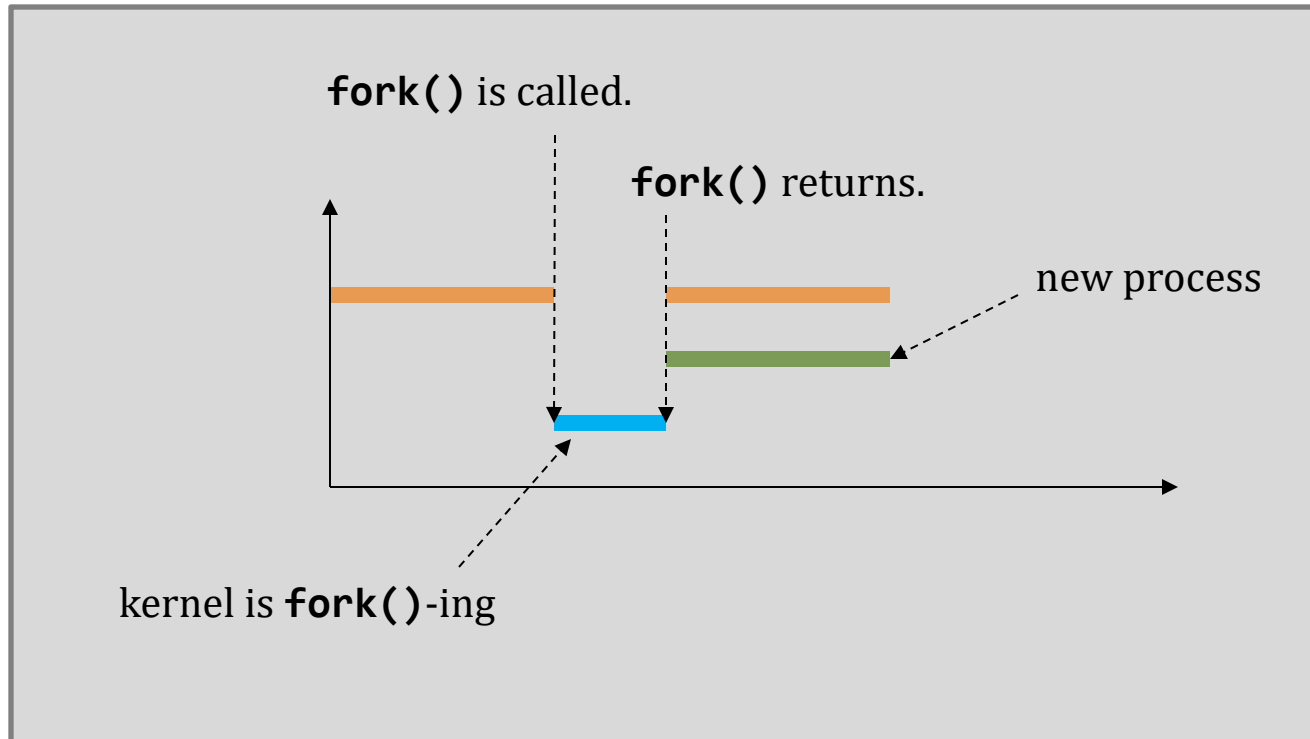
<https://www.quora.com/Is-an-OS-kernel-itself-a-process>

# Working of system calls

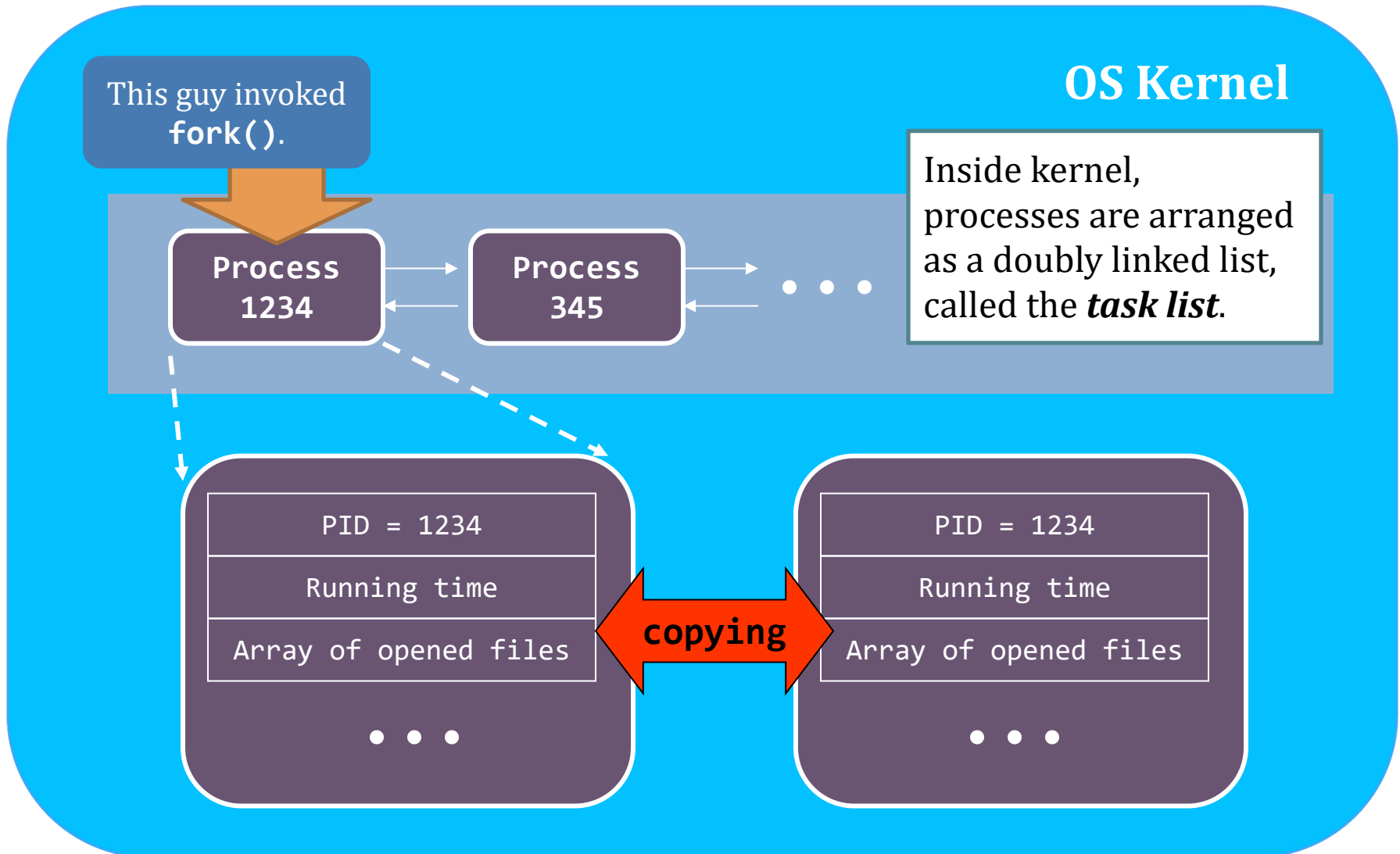
- `fork()`;



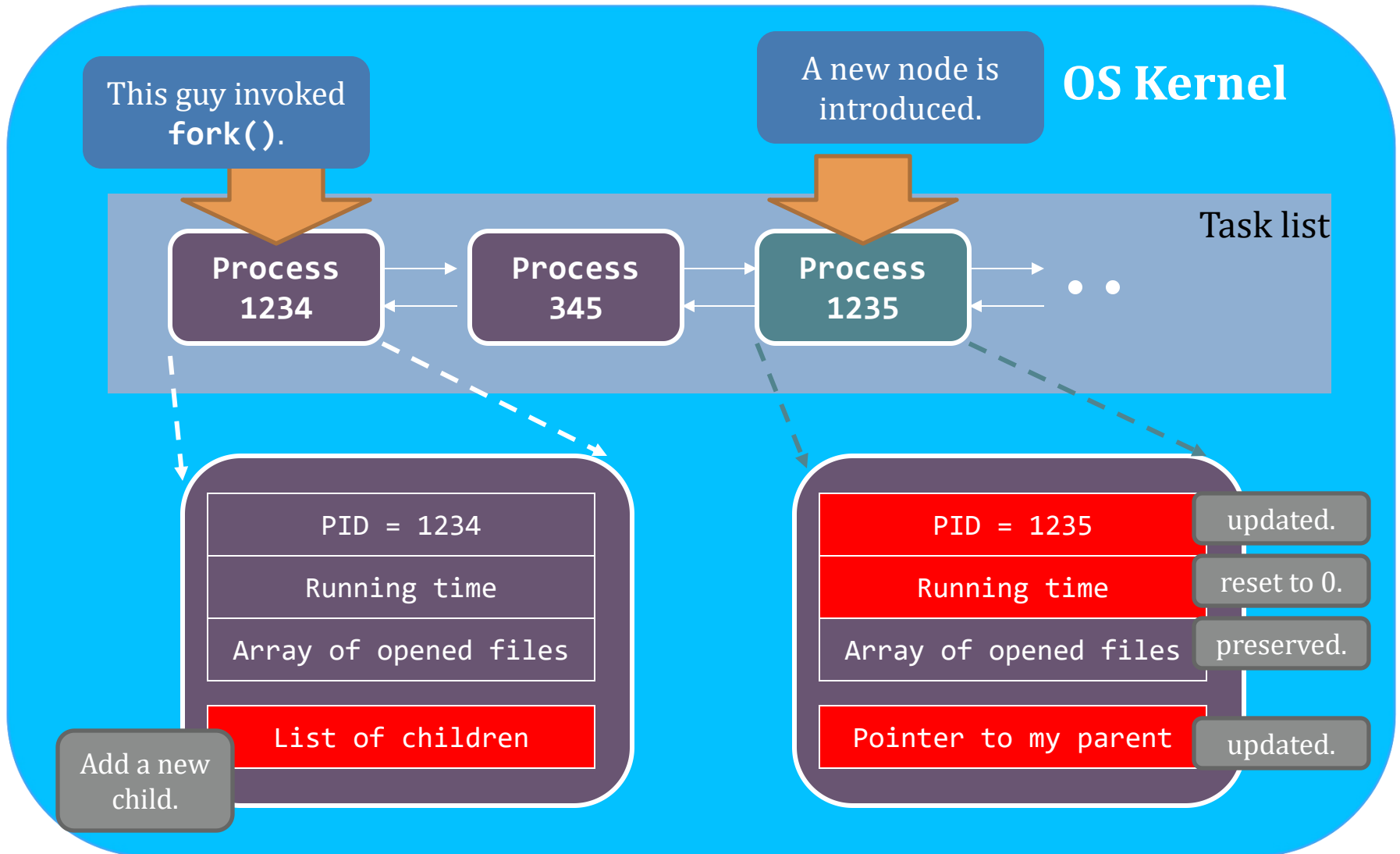
# Programmer view of fork()



# fork() inside the kernel

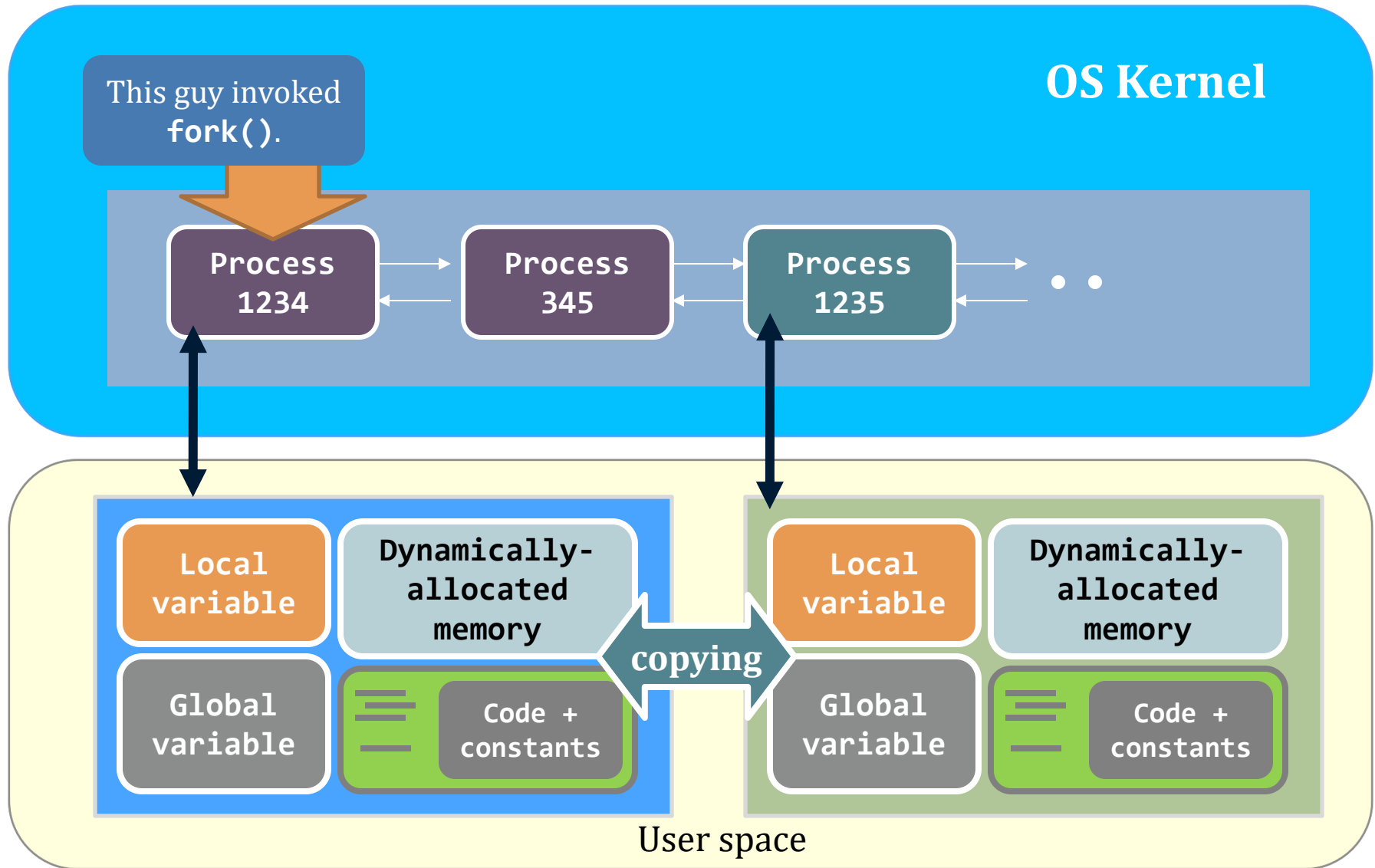


# fork() in action – kernel-space update

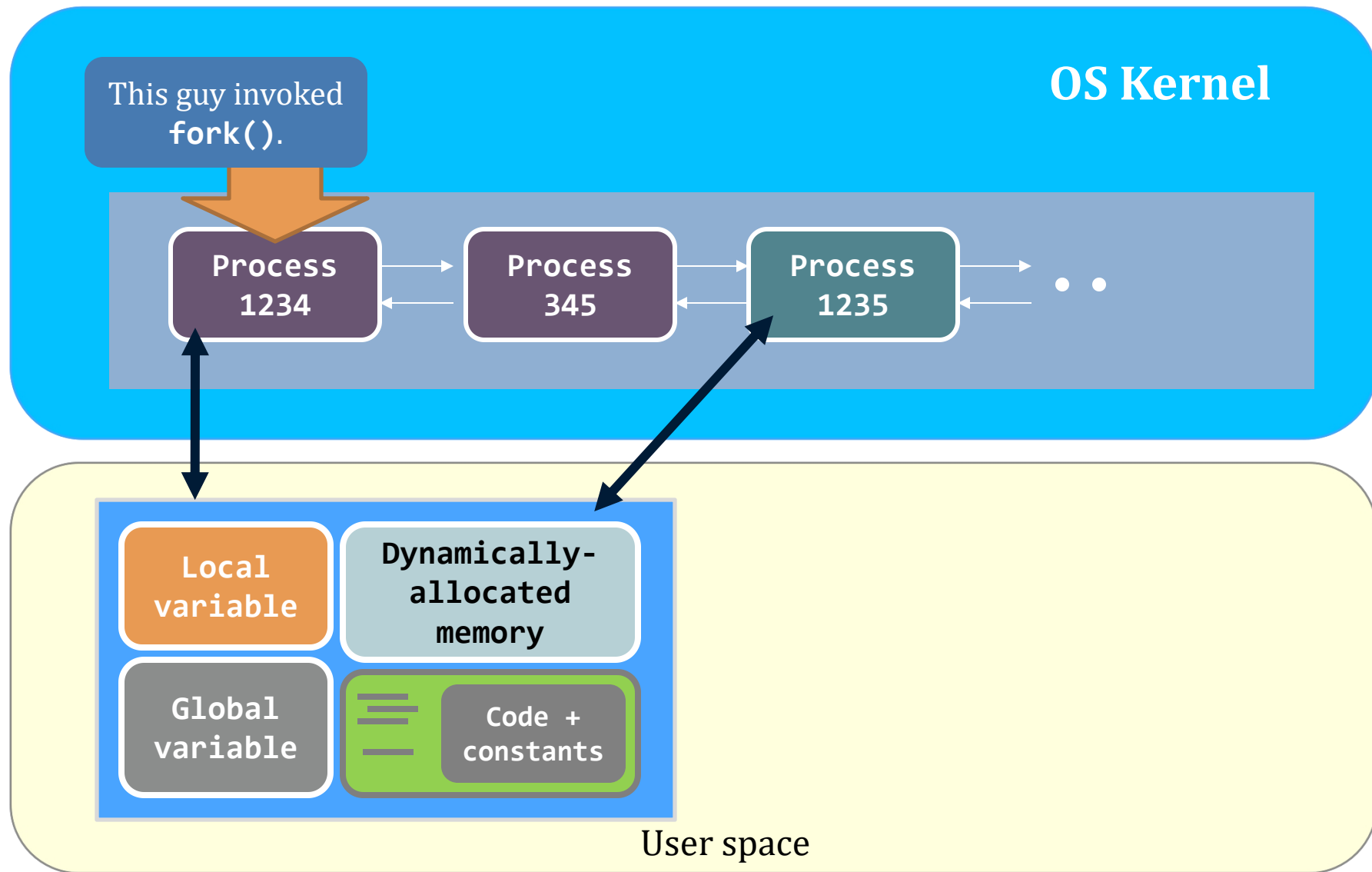




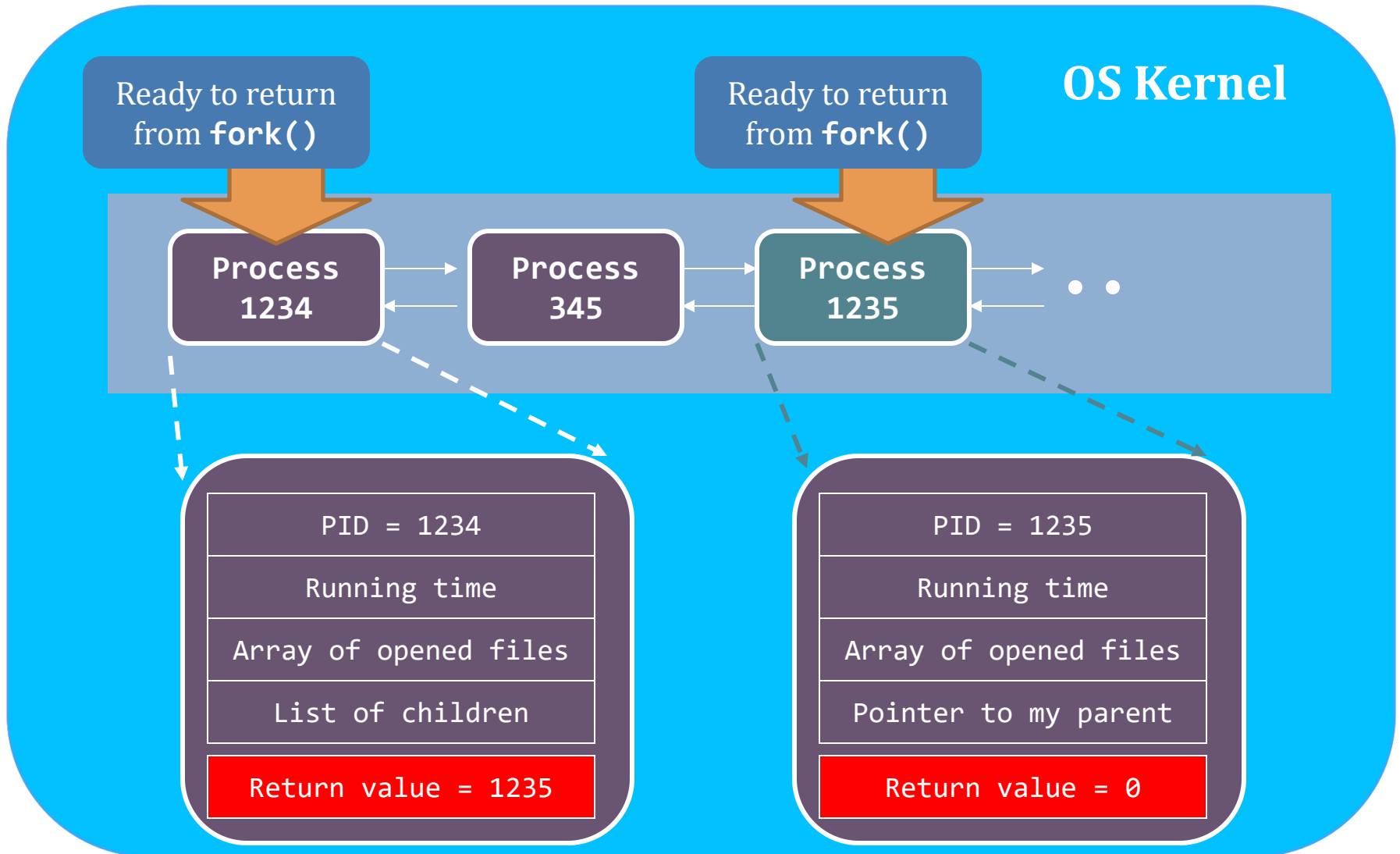
# fork() in action – user-space update



# Or Copy-on-Write



# fork() in action – finish



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

✧ Array of opened files contains:

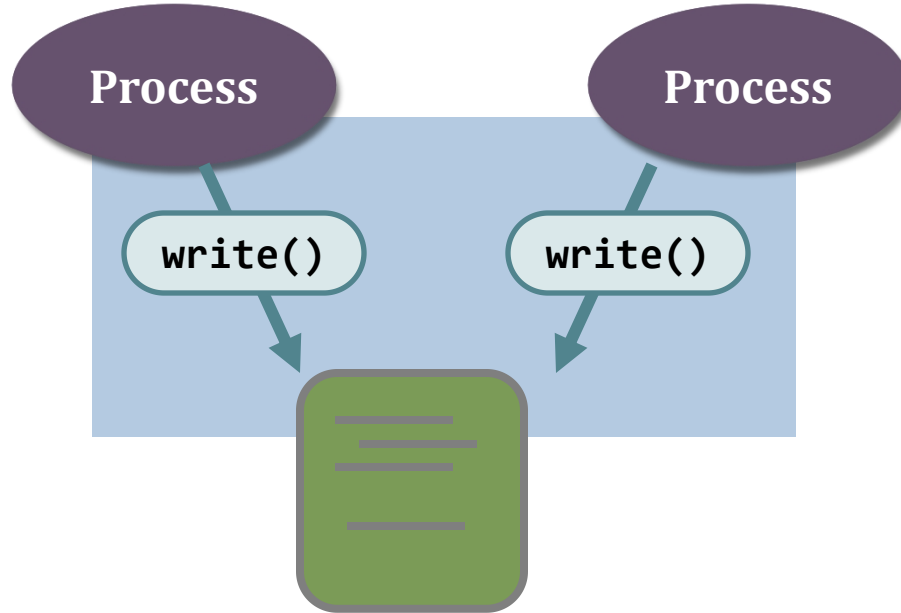
Array Index	Description
0	Standard Input Stream; <b>FILE *stdin;</b>
1	Standard Output Stream; <b>FILE *stdout;</b>
2	Standard Error Stream; <b>FILE *stderr;</b>
3 or beyond	Storing the files you opened, e.g., <b>fopen()</b> , <b>open()</b> , etc.

✧ 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 → ArrayEnd, 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!

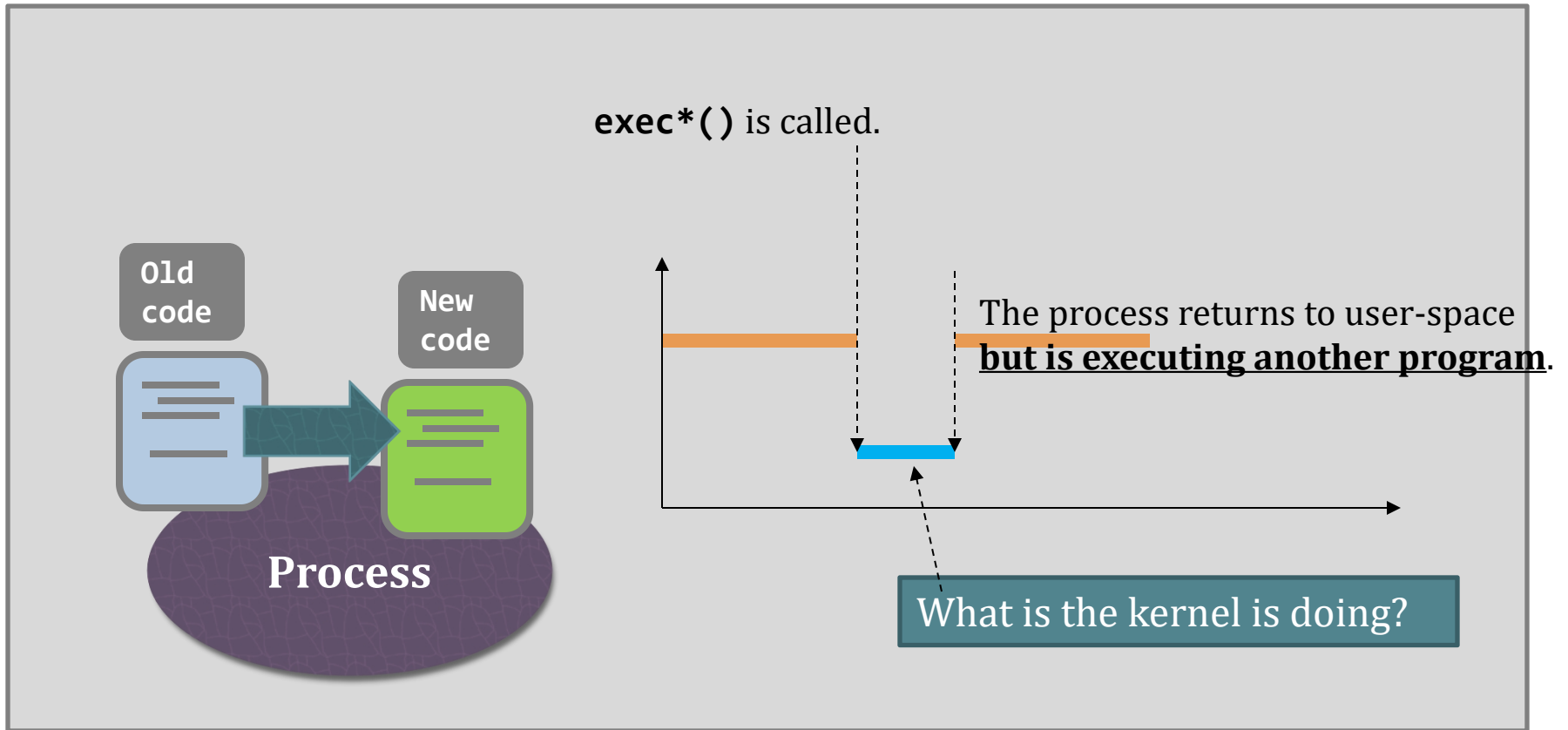
# Working of system calls

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

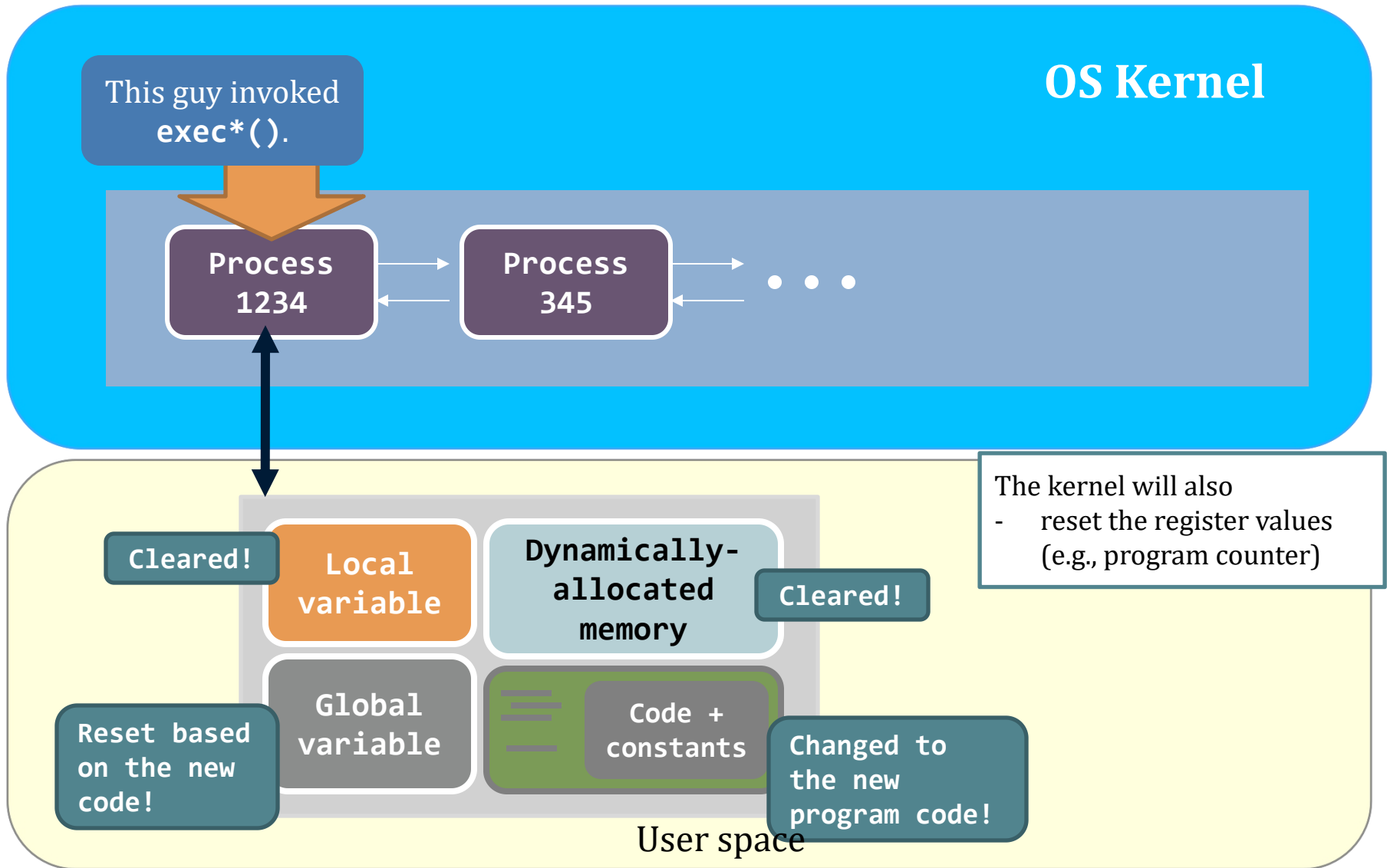


# `exec*()` that you've learnt...

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



# exec\*() in action



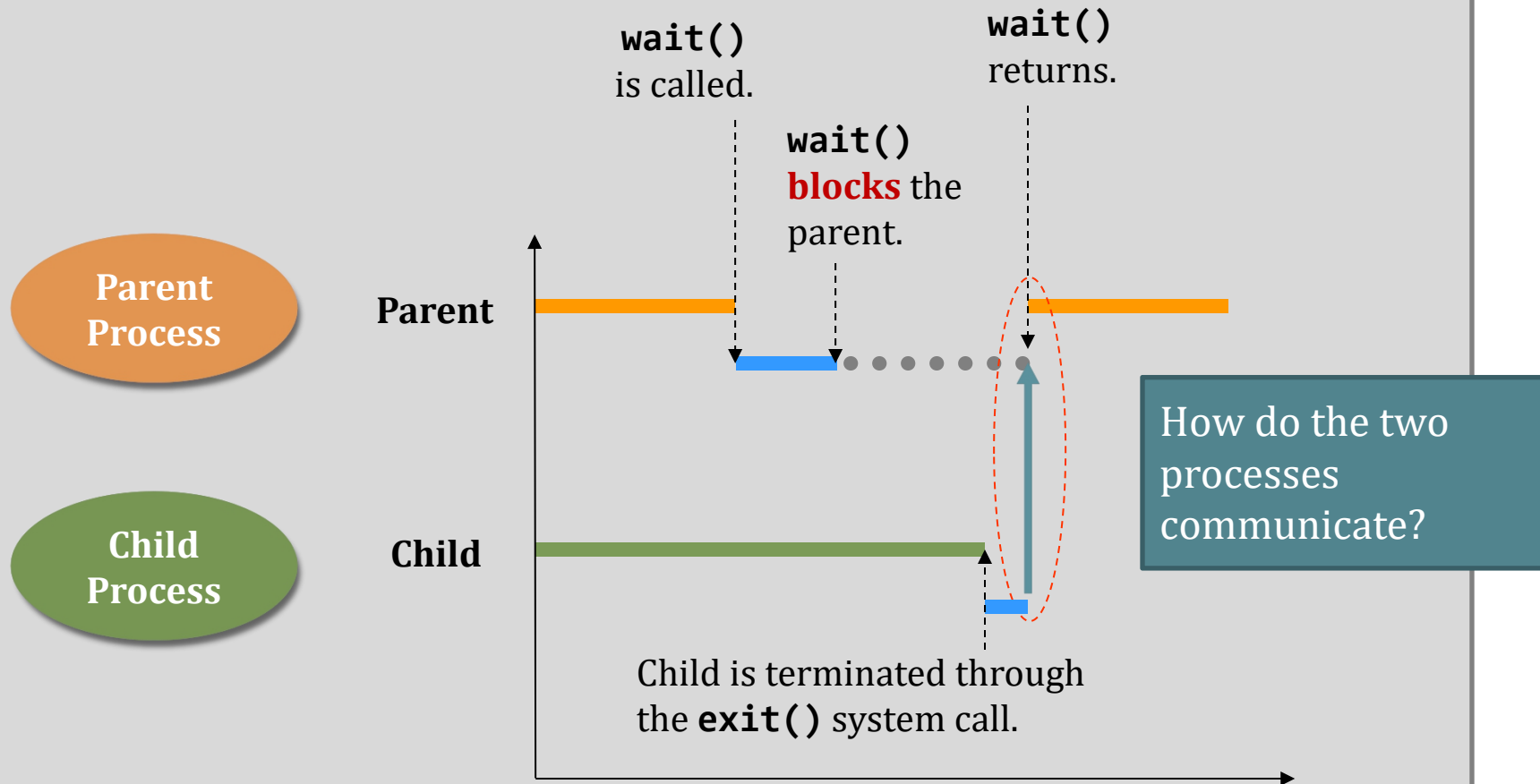


# Working of system calls

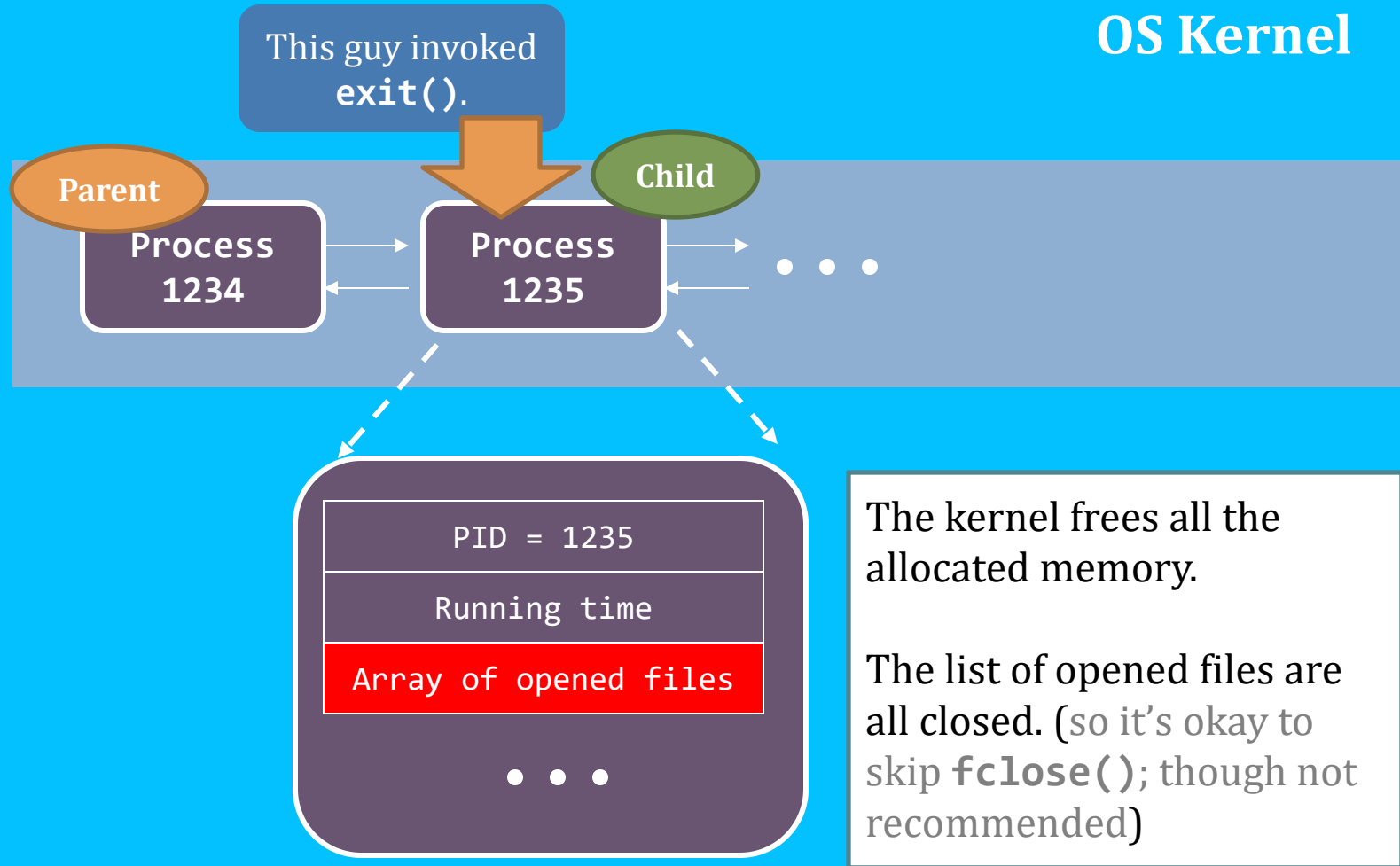
- `fork()`;
- `exec*()`;
- `wait()` + `exit()`;



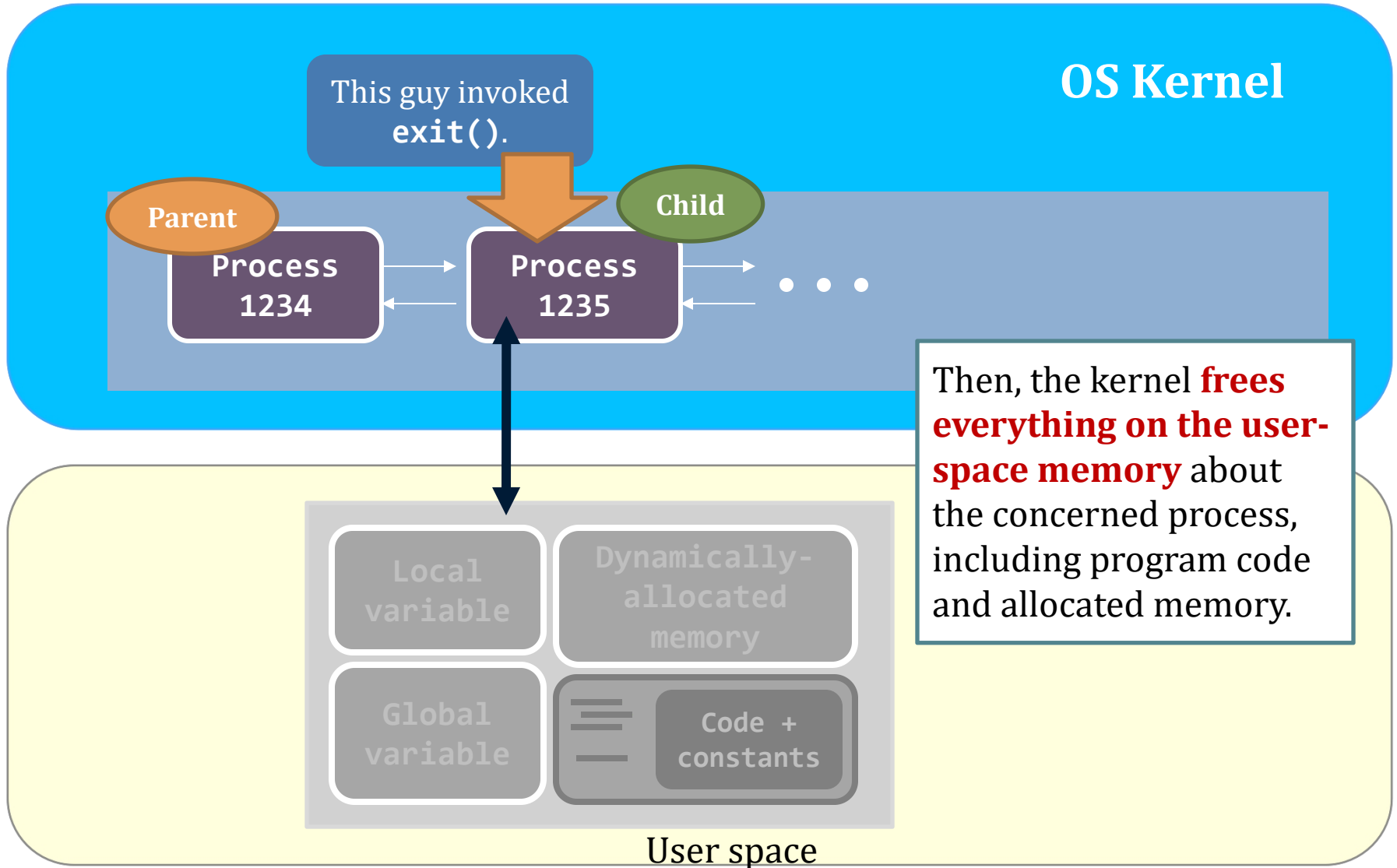
# wait() and exit()



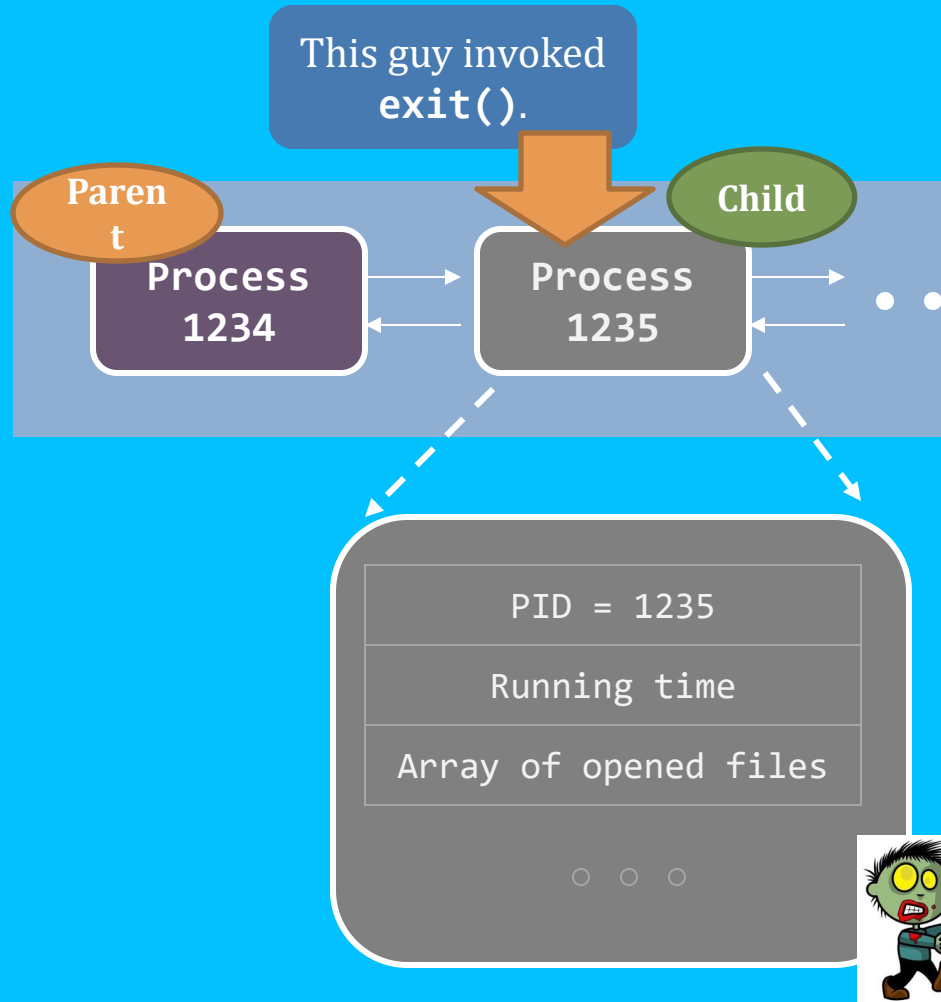
# exit() (kernel-view)



# exit() (kernel-view)



# exit() (kernel-view)



## OS Kernel

Process ID stills in the kernel's process table

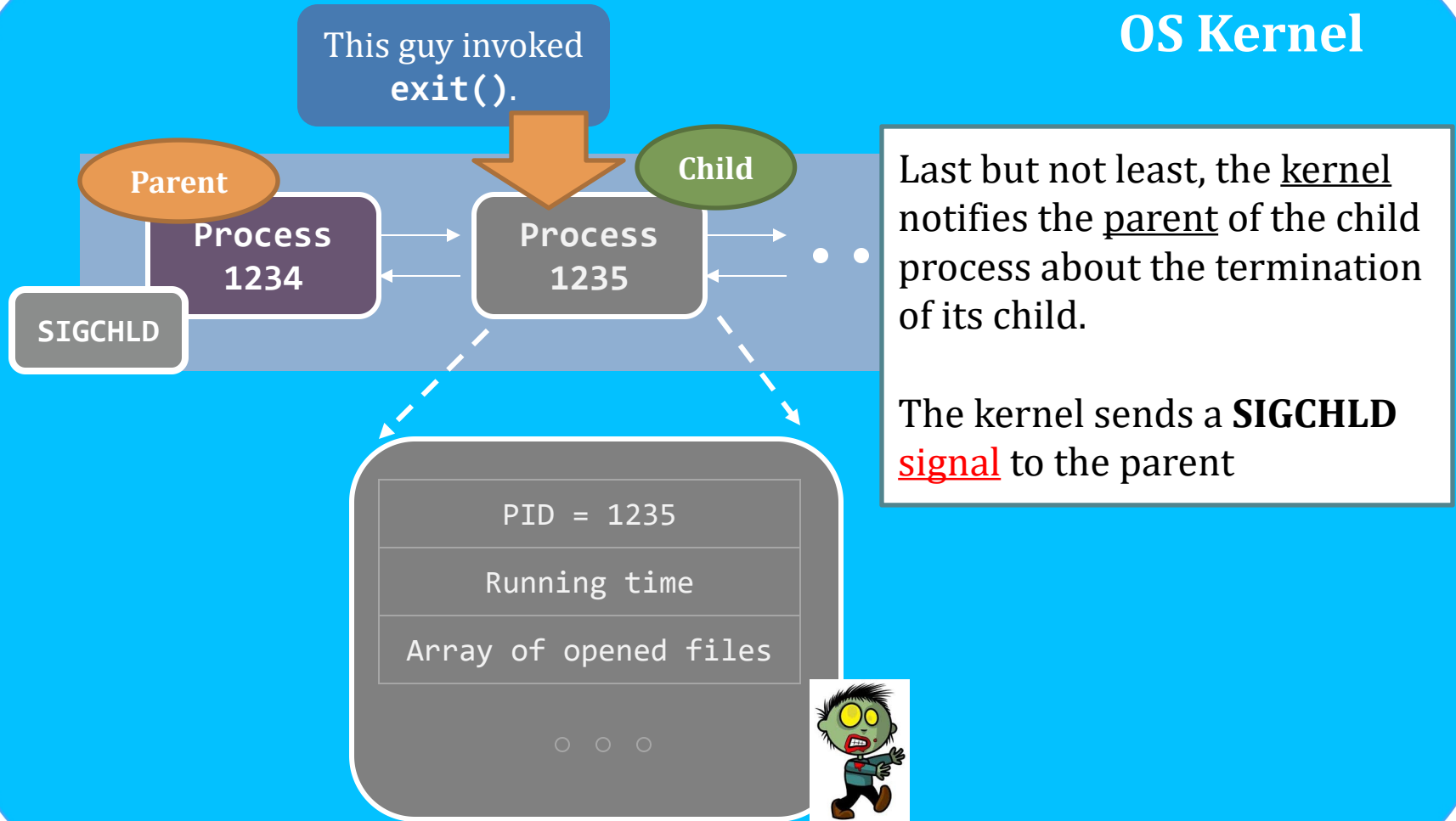
- Why?

[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)

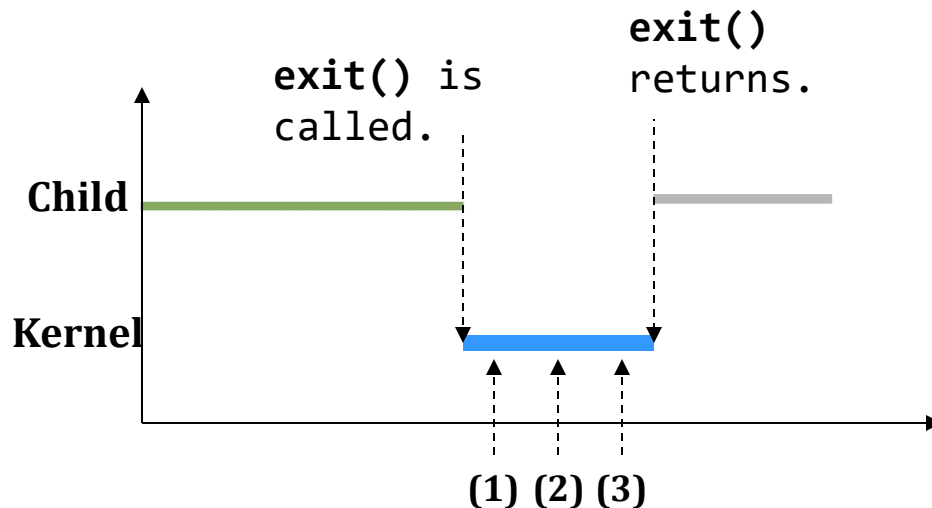


# Summary -- what the kernel does for `exit()`

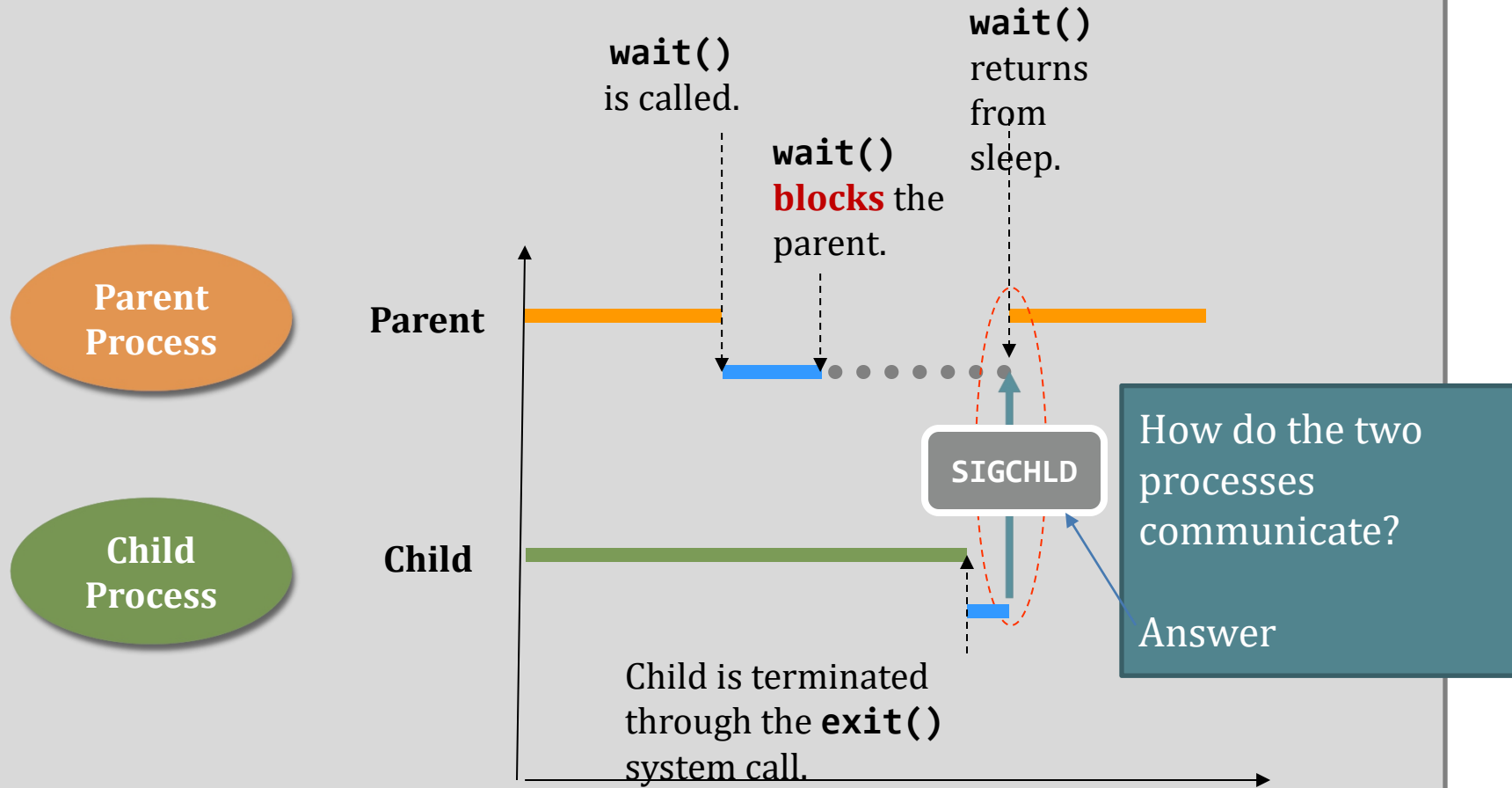
Step (1) Clean up most of the allocated kernel-space memory (e.g., process's running time info).

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

Step (3) Notify the parent with SIGCHLD.

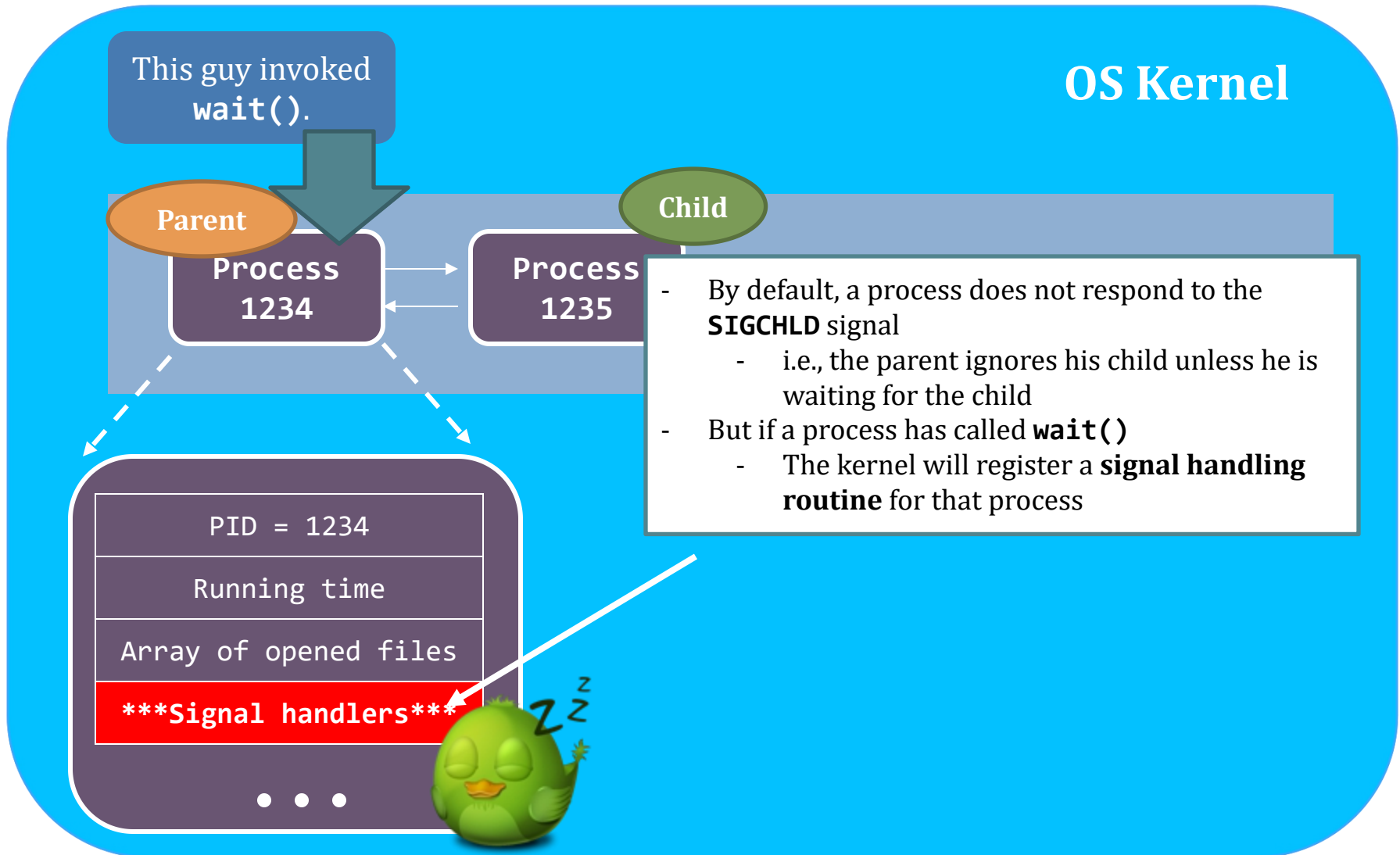


# wait() and exit()

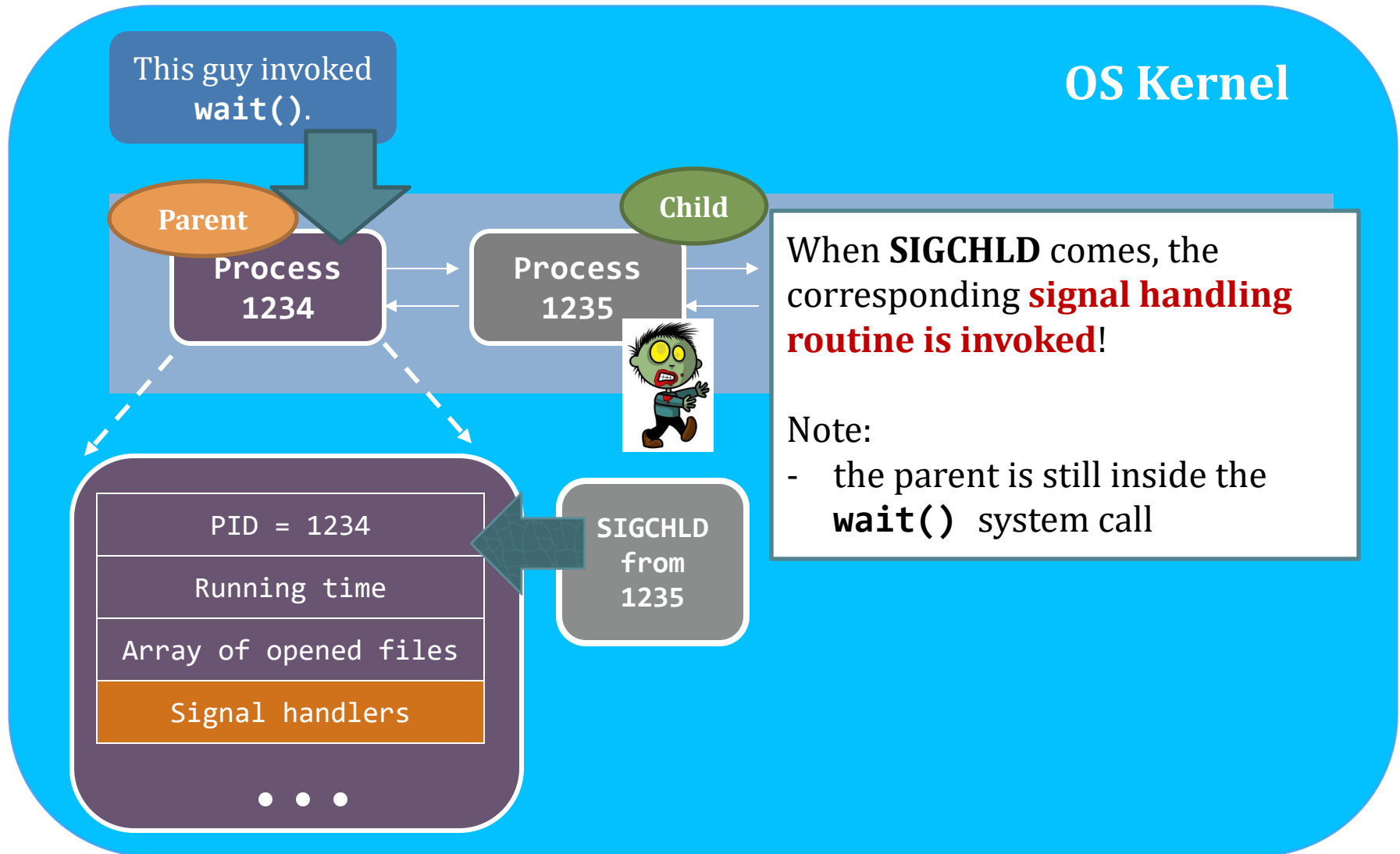




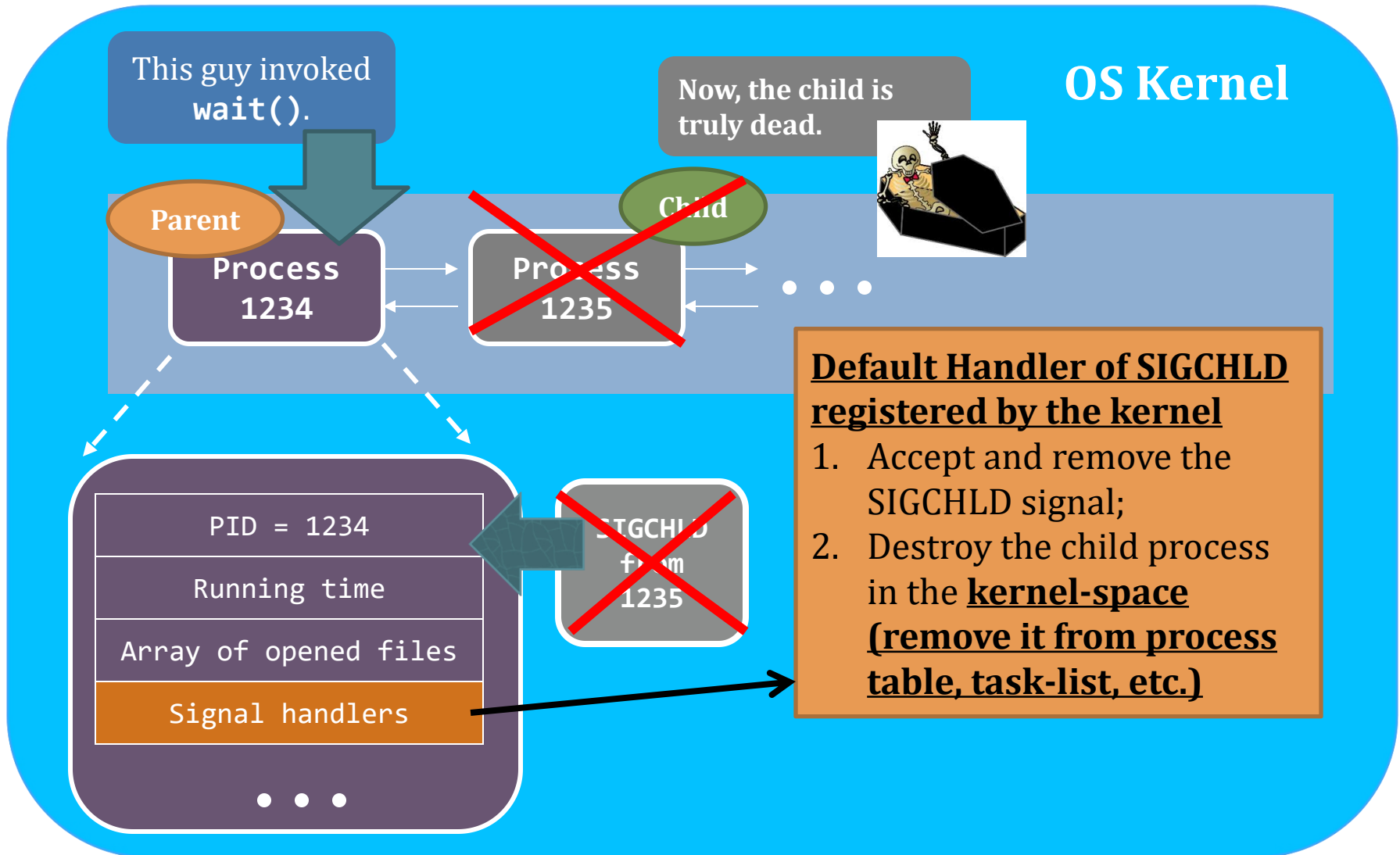
# wait() kernel view's – registering signal handling routine



# wait() kernel's view



# wait() kernel's view



# wait() kernel's view

## OS Kernel

Ready to return  
from `wait()`.

Parent

Process  
1234

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.

PID = 1234

Running time

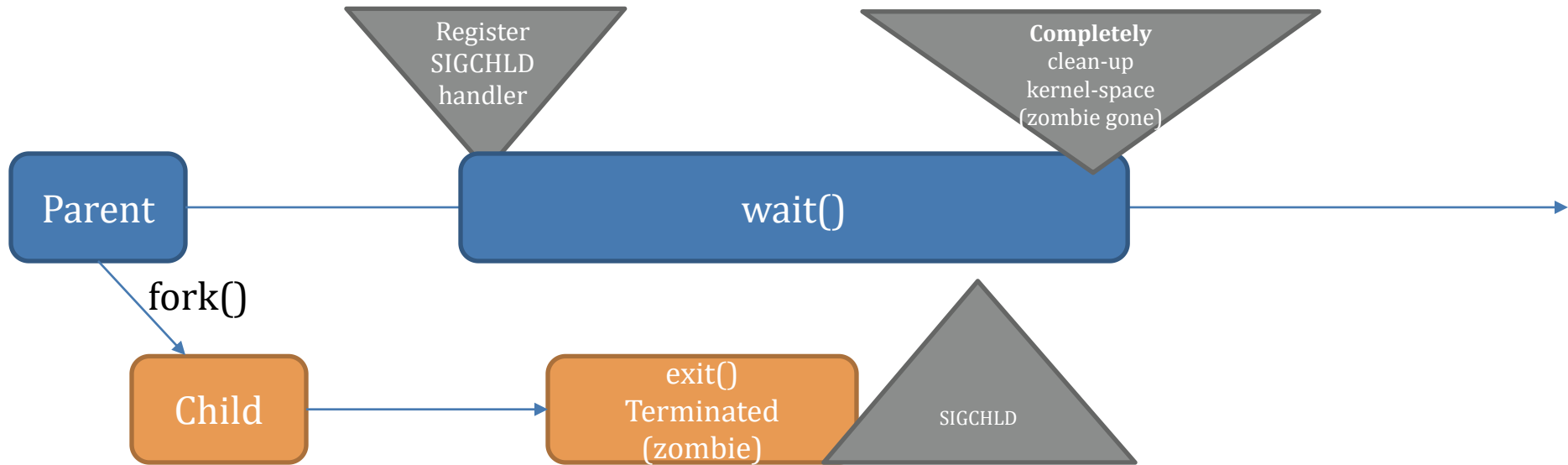
Array of opened files

~~Signal handlers~~

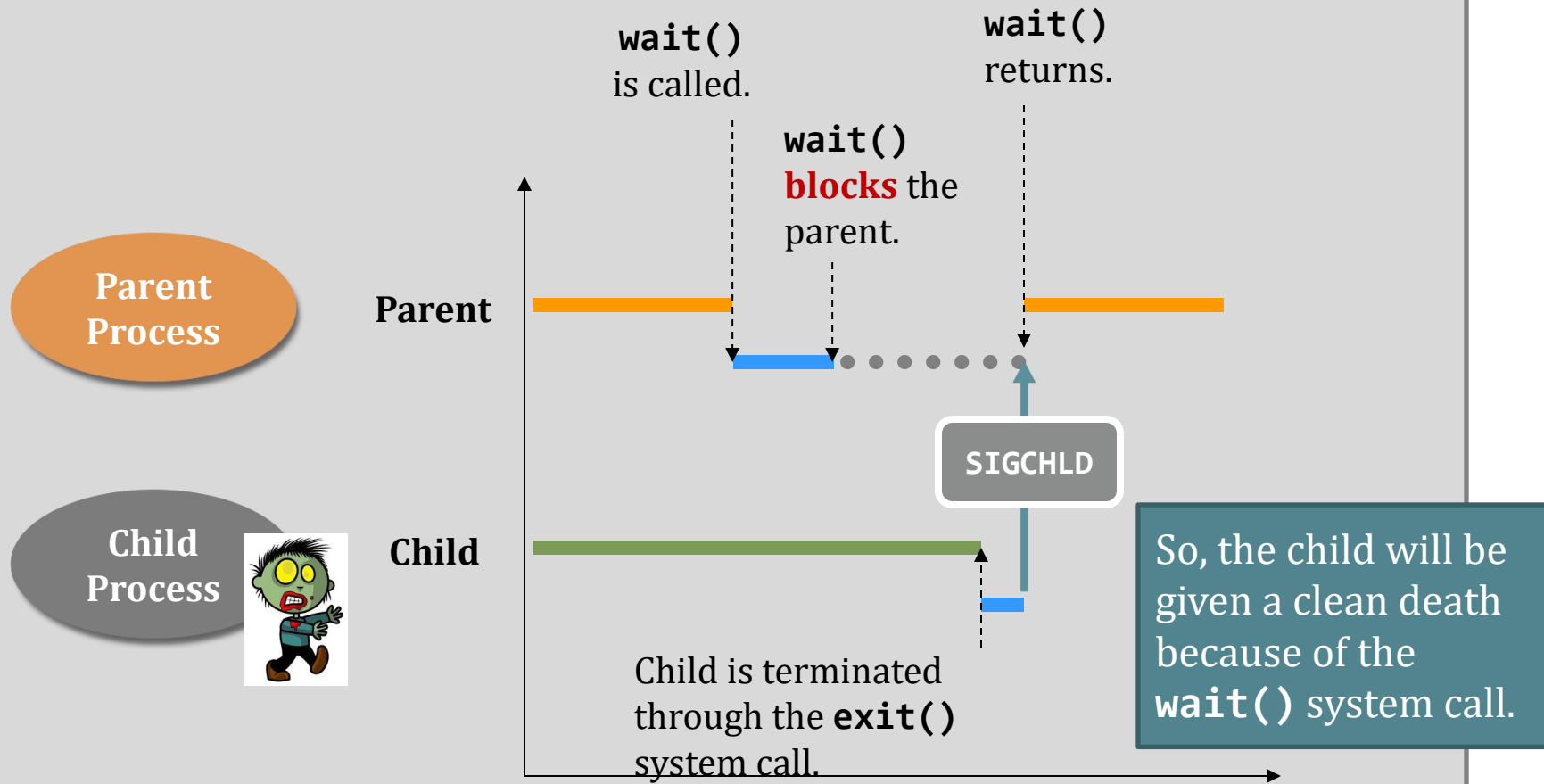
Return value = 1235



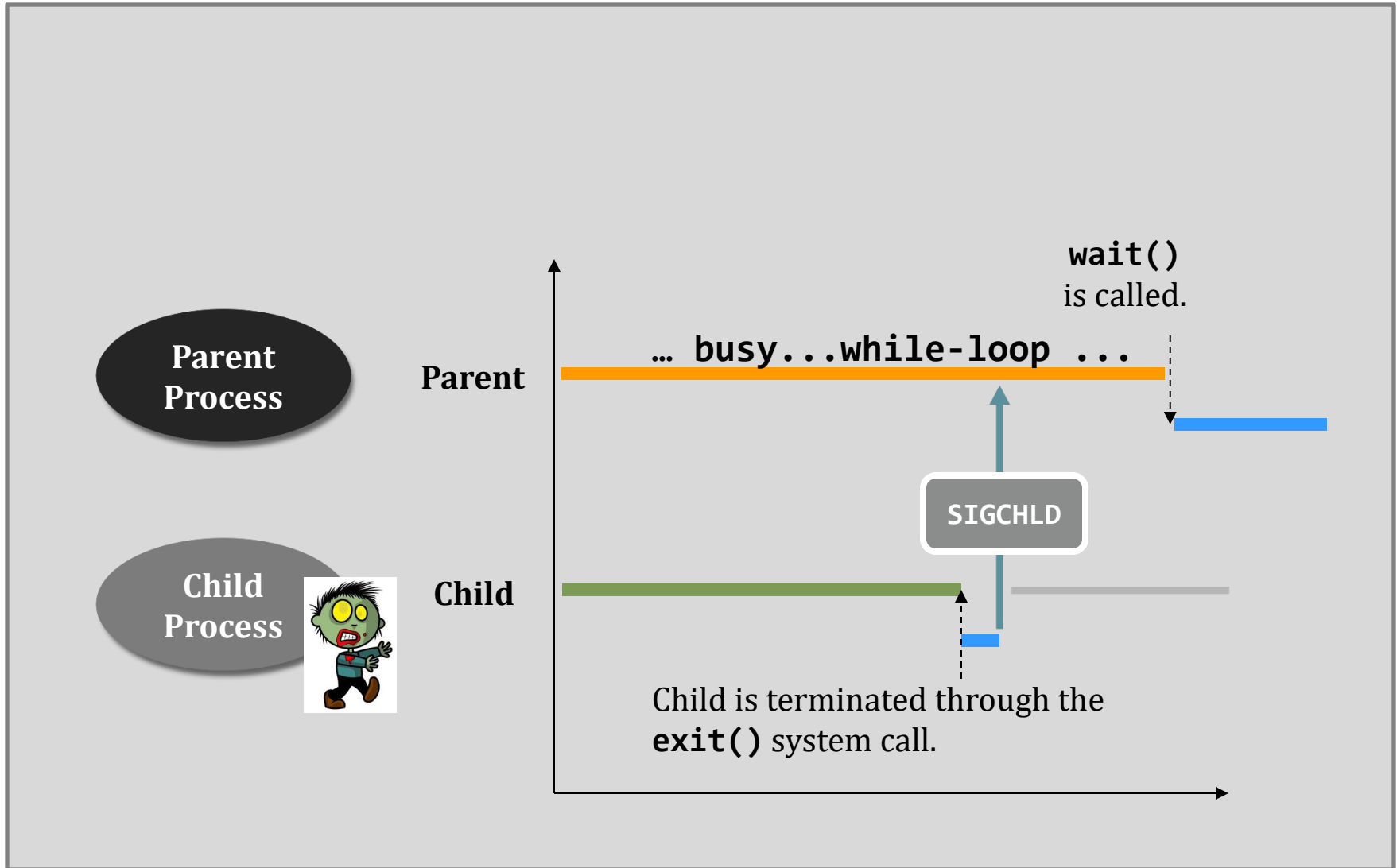
# Overall – normal case



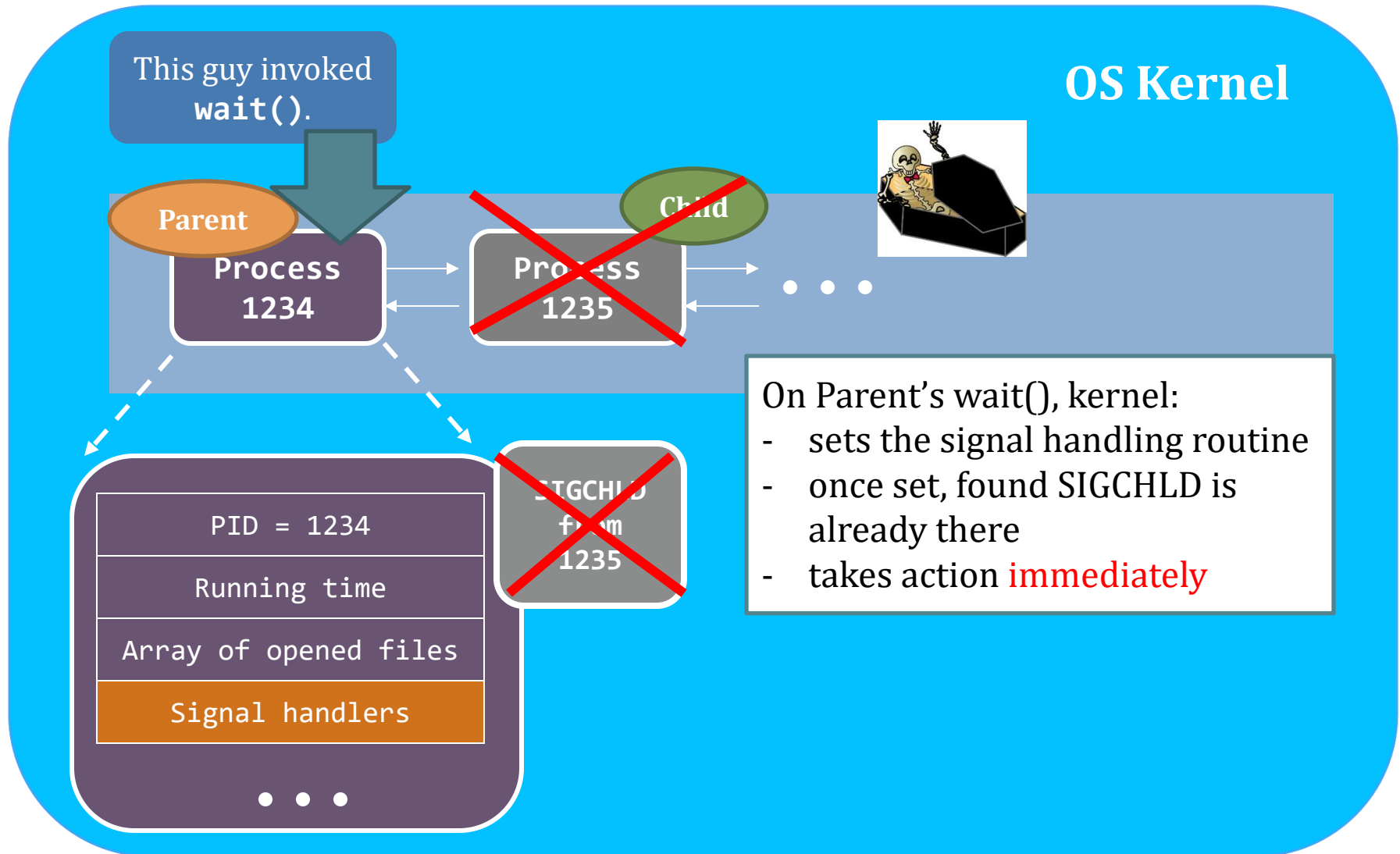
# Normal Case



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

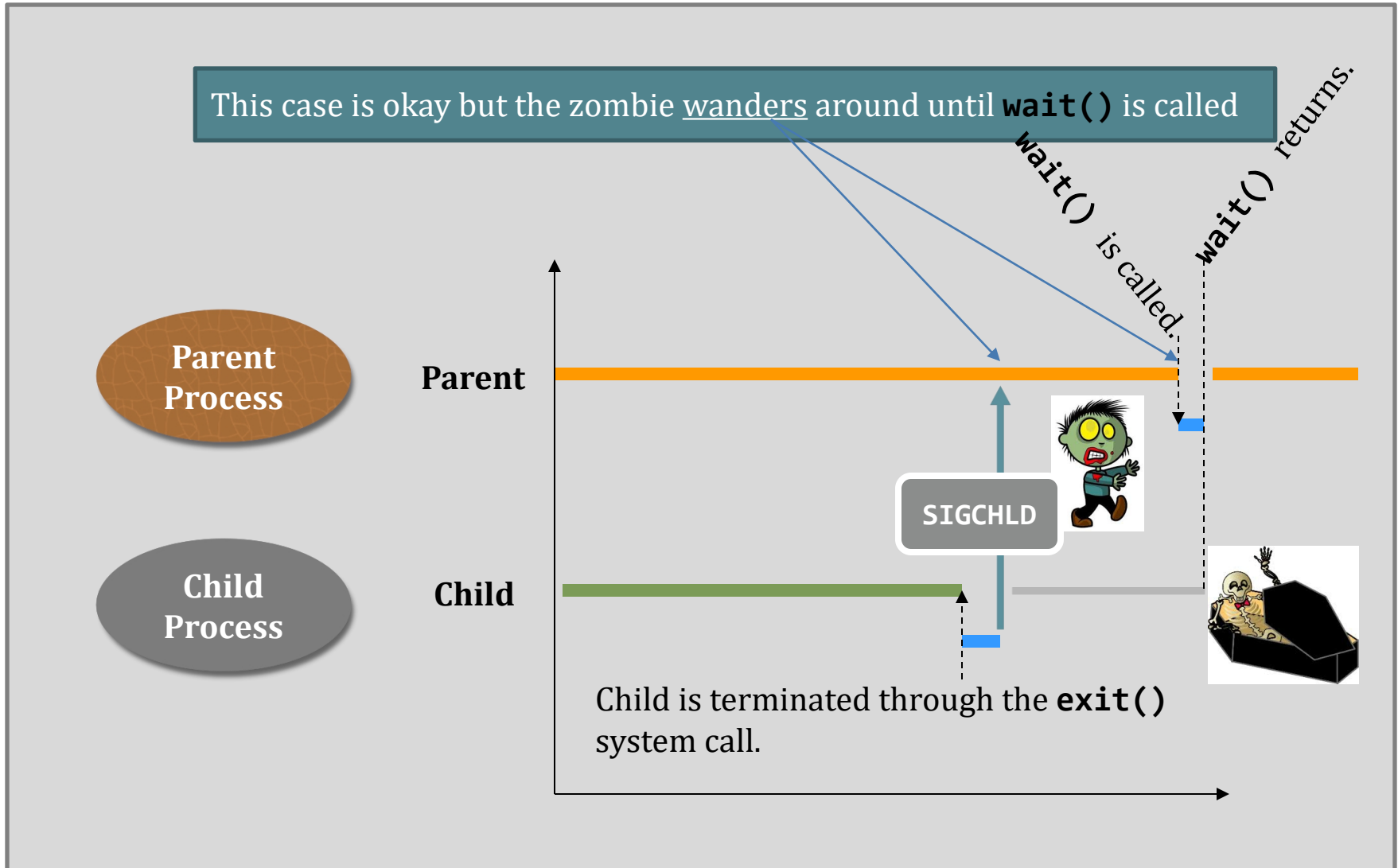


# Parent's Wait() after Child's exit()





# 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

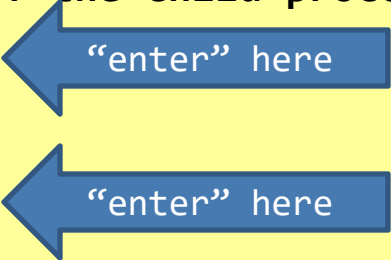
- ✧ **wait() & waitpid()** are to reap zombie child processes.
  - ✧ It is a must that you should never leave any zombies in the system.
  - ✧ **wait() & waitpid()** pause the caller until
    - ✱ A child terminates/stops, OR
    - ✱ The caller receives a signal (i.e., the signal interrupted the wait())
- ✧ 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

```
1 int main(void)
2 {
3     int pid;
4     if( (pid = fork()) !=0 ) {
5         printf("Look at the status of the child process %d\n", pid);
6         while( getchar() != '\n' );
7         wait(NULL);
8         printf("Look again!\n");
9         while( getchar() != '\n' );
10    }
11    return 0;
12 }
```



The diagram shows two blue arrows pointing from the text "enter here" to the while loops in the code. The first arrow points to the while loop on line 6, and the second arrow points to the while loop on line 9.

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 1<sup>st</sup> and the 2<sup>nd</sup> “enter”.

# Working of system calls

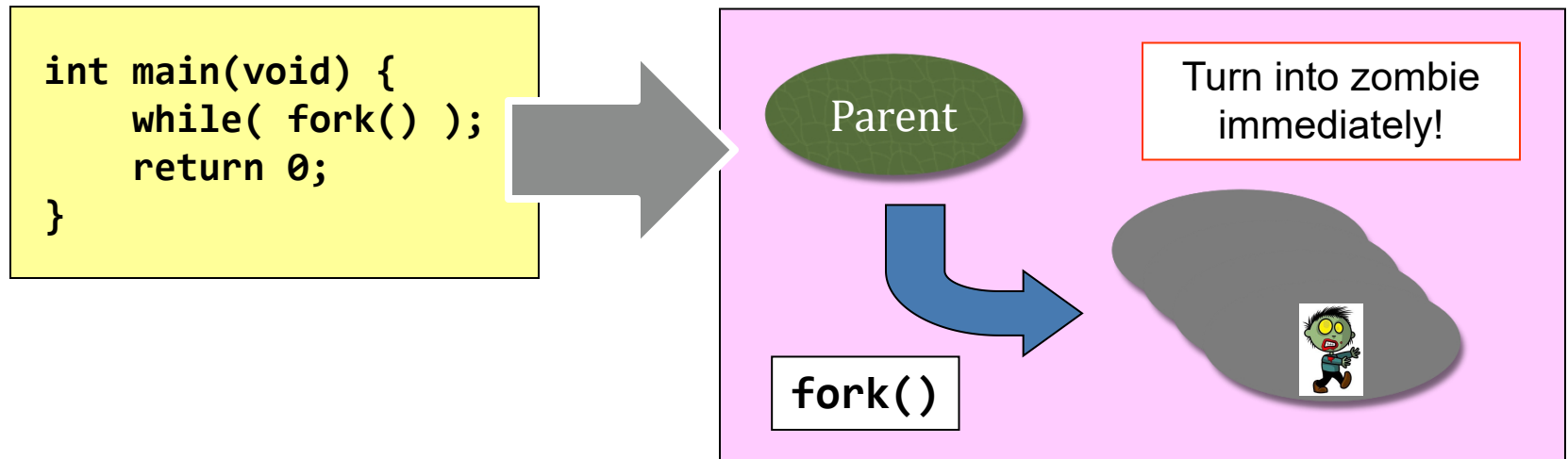
- `fork()`;
- `exec*()`;
- `wait()` + `exit()`;
- **importance/fun in knowing the above things?**

# 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

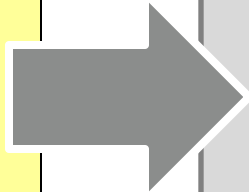
- ✧ Deliberately missing wait()
- ✧ Do not try this on department's machines...



**An infinite, zombie factory!**

# When `wait()` is absent...

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



\$ ./interesting

—

Terminal A

\$ **ls**

No process left.

\$ **poweroff**

No process left.

\$ **=\_\_=**

No process left.

\$ —

Terminal B

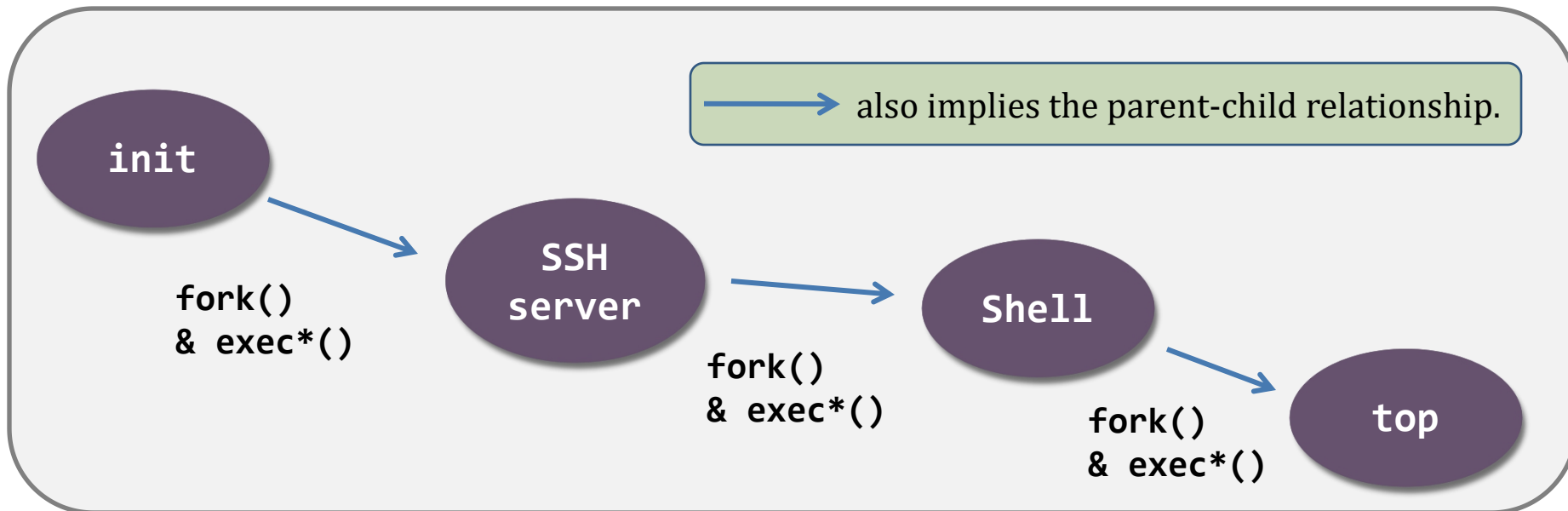


# The first process

- ✧ 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\*()**.

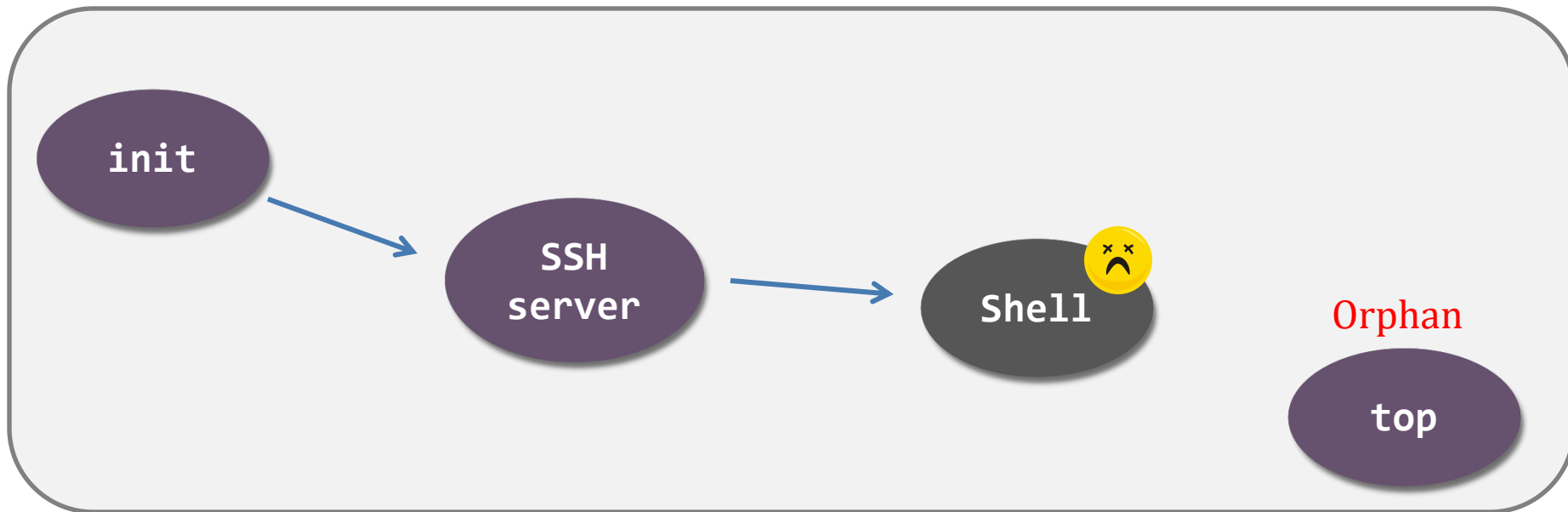
# Process blossoming

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



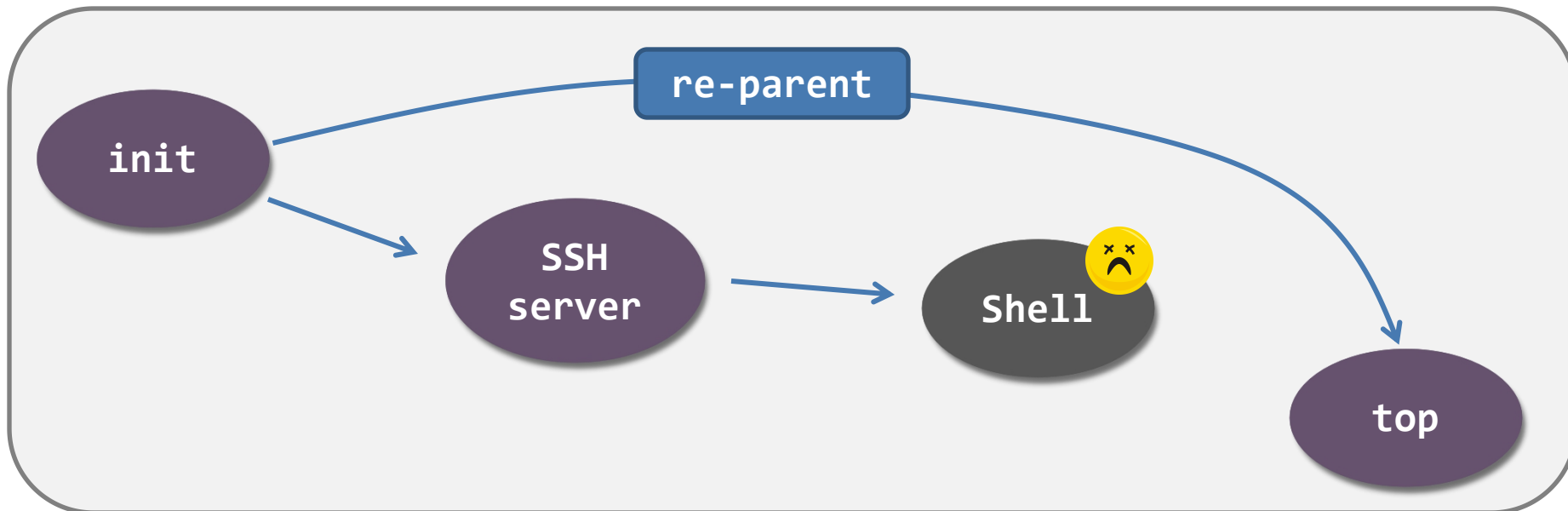
# Process blossoming...with orphans?

- ✧ 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)

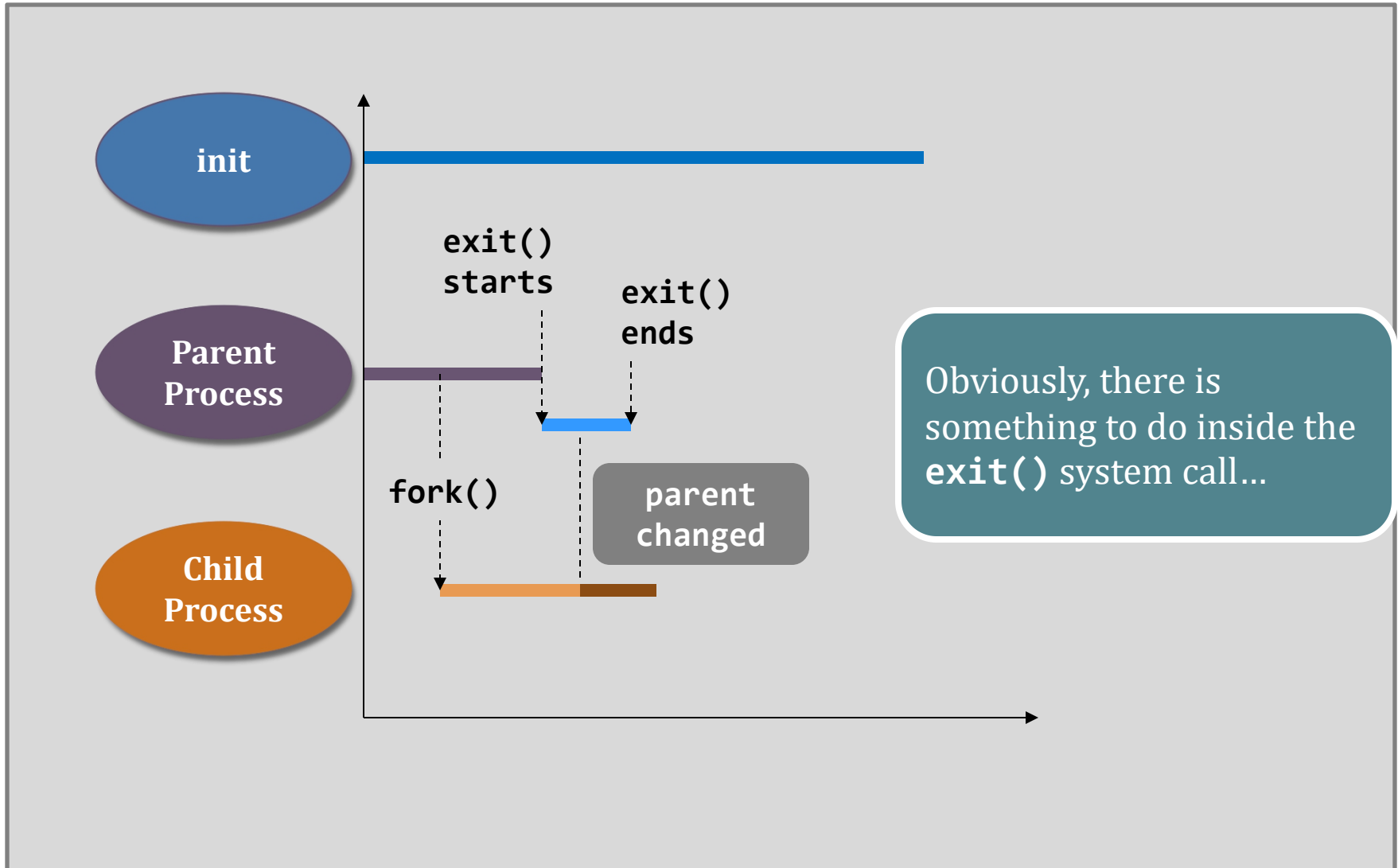
# Re-parenting example

```
1  int main(void) {
2      int i;
3      if(fork() == 0) {
4          for(i = 0; i < 5; i++) {
5              printf("(%d) parent's PID = %d\n",
6                  getpid(), getppid() );
7              sleep(1);
8          }
9      }
10     else
11         sleep(1);
12     printf("(%d) bye.\n", getpid());
13 }
```

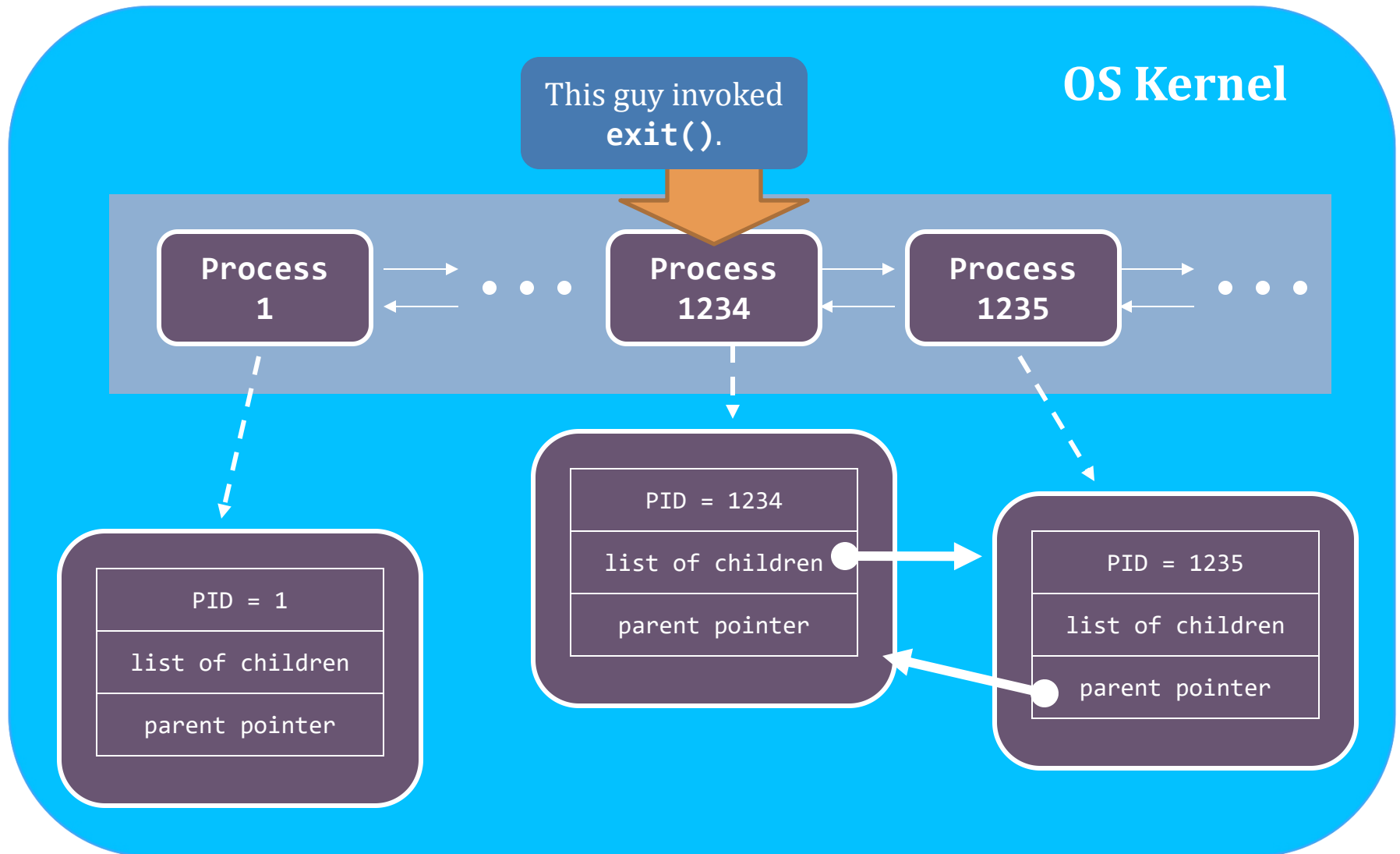
`getppid()` is the system call that returns the parent's PID of the calling process.

```
$ ./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.
$ _
```

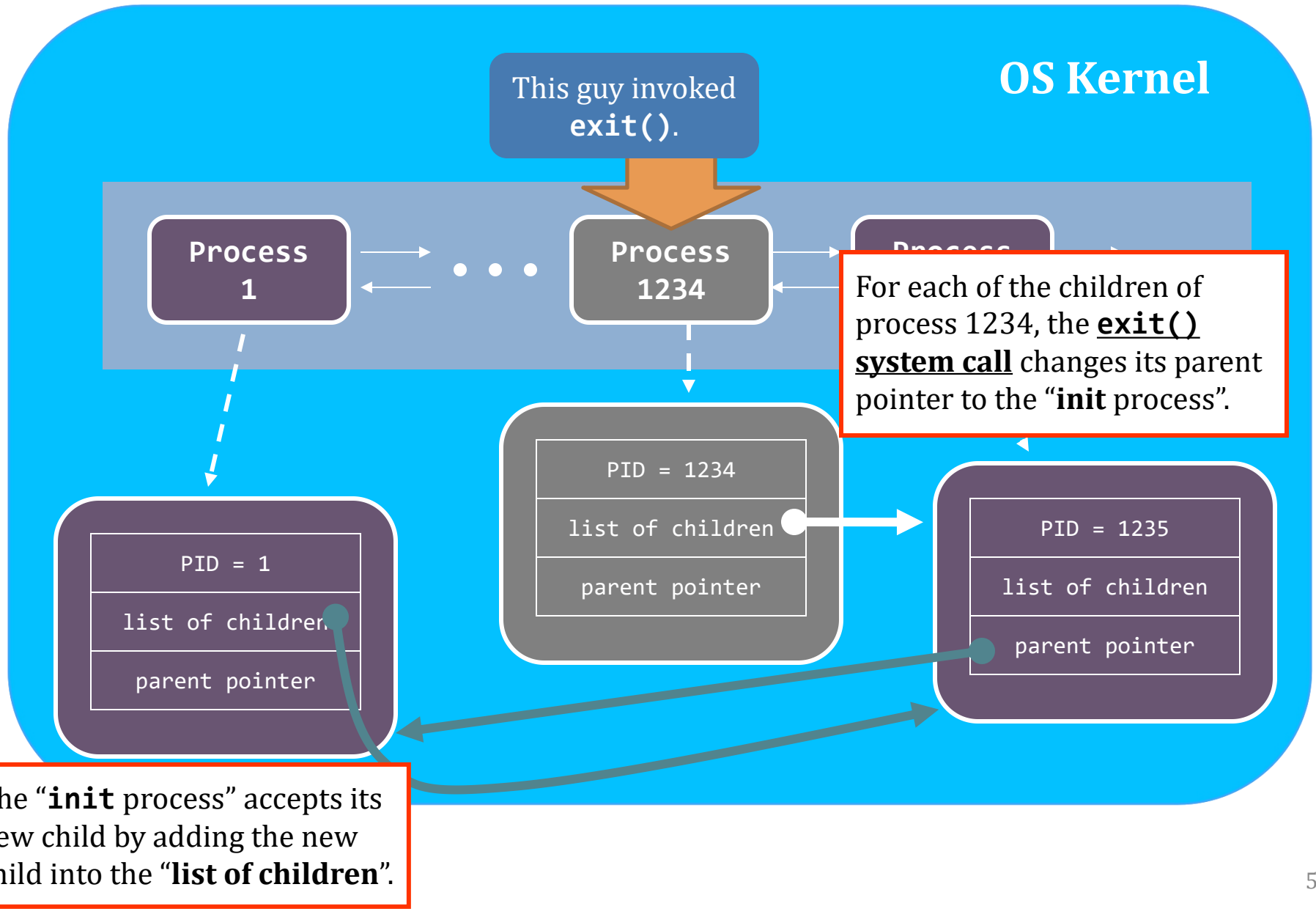
# What had happened during re-parenting?



# What had happened during re-parenting?



# What had happened during re-parenting?





# Background jobs

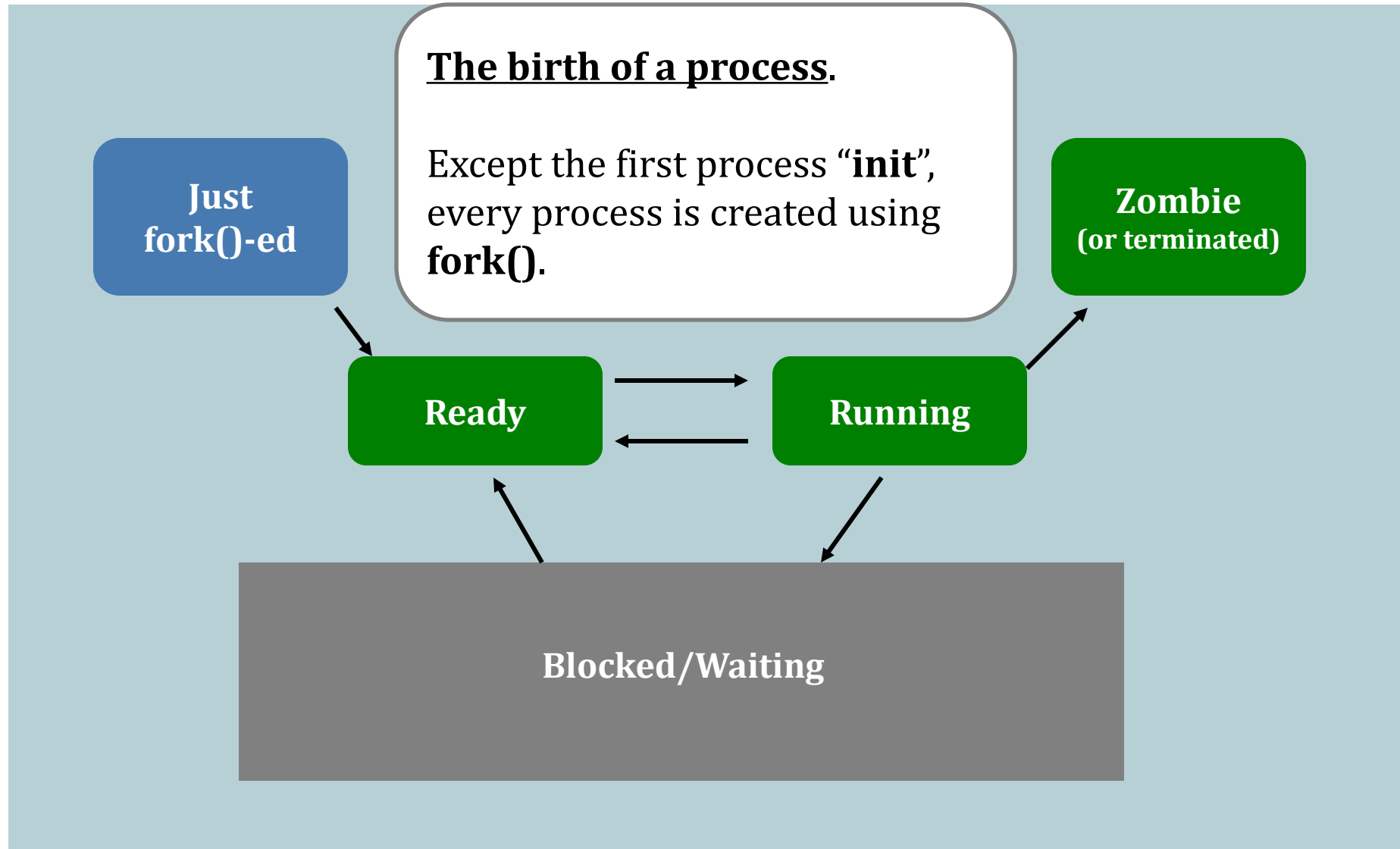
- ✧ The re-parenting operation enables something called **background jobs** in Linux
  - ✧ It allows a process runs **without a parent terminal/shell**

[Back to home](#)

```
$ ./infinite_loop &  
$ exit  
  
[ The shell is gone ]
```

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

# Process lifecycle



# Process lifecycle - Ready

Just  
fork()-ed

Ready

Block

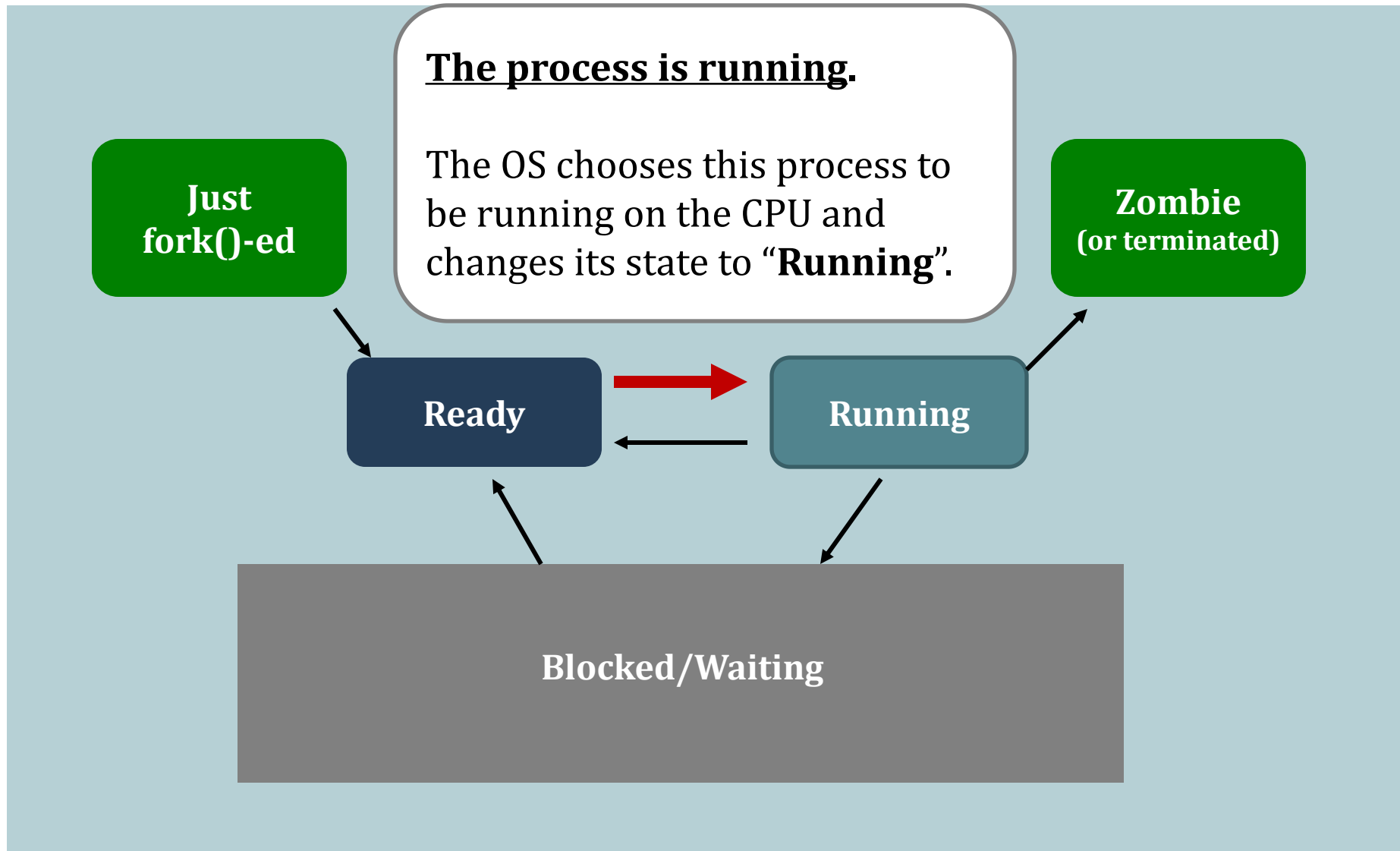
The process is ready.

It means it is **ready to run but is not running.**

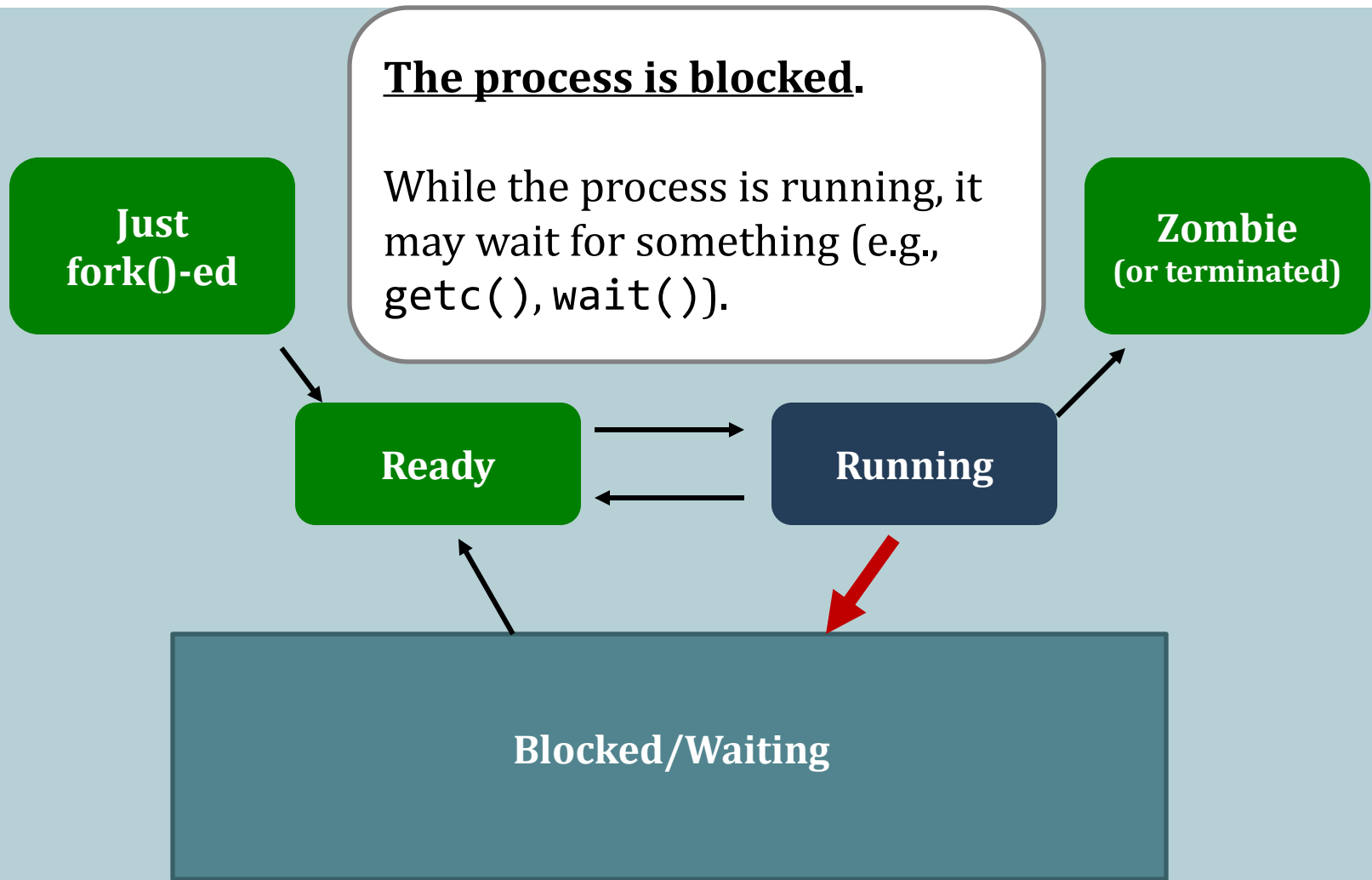
A process may become “ready” (*runnable*) after...

- 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

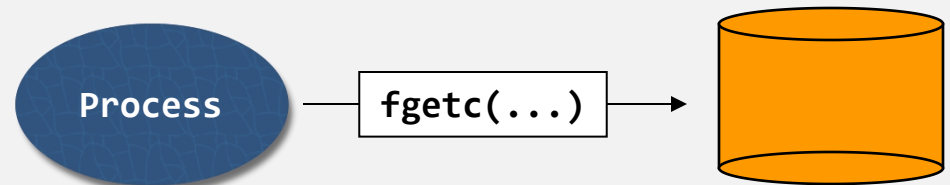


# Process lifecycle – Interruptible wait

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).

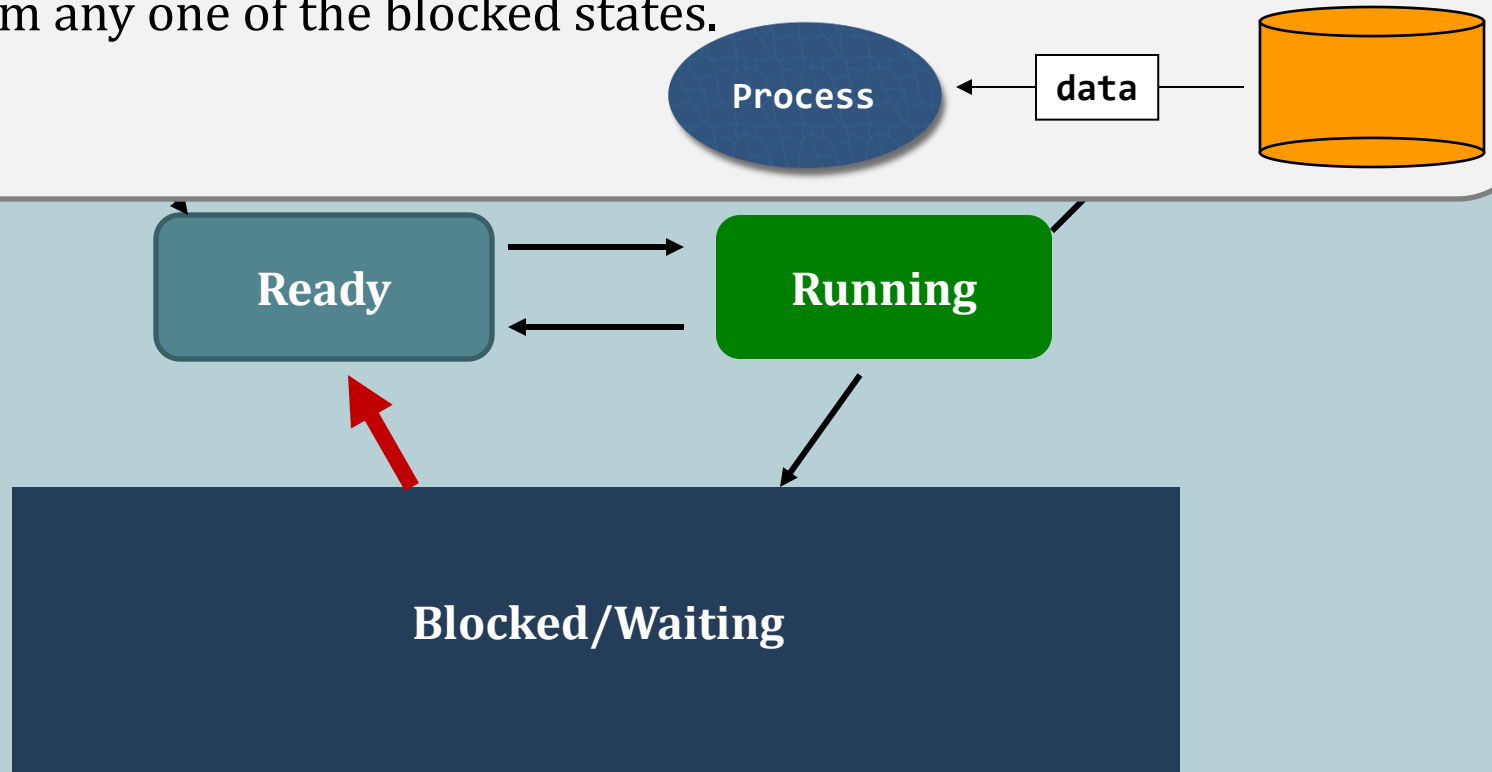


**Blocked/Waiting**

# 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

## The process is going to die.

The process may

- choose to terminate itself; or
- force to be terminated.

**Running**

**Zombie  
(or terminated)**

**Blocked/Waiting**



**Thank You!**