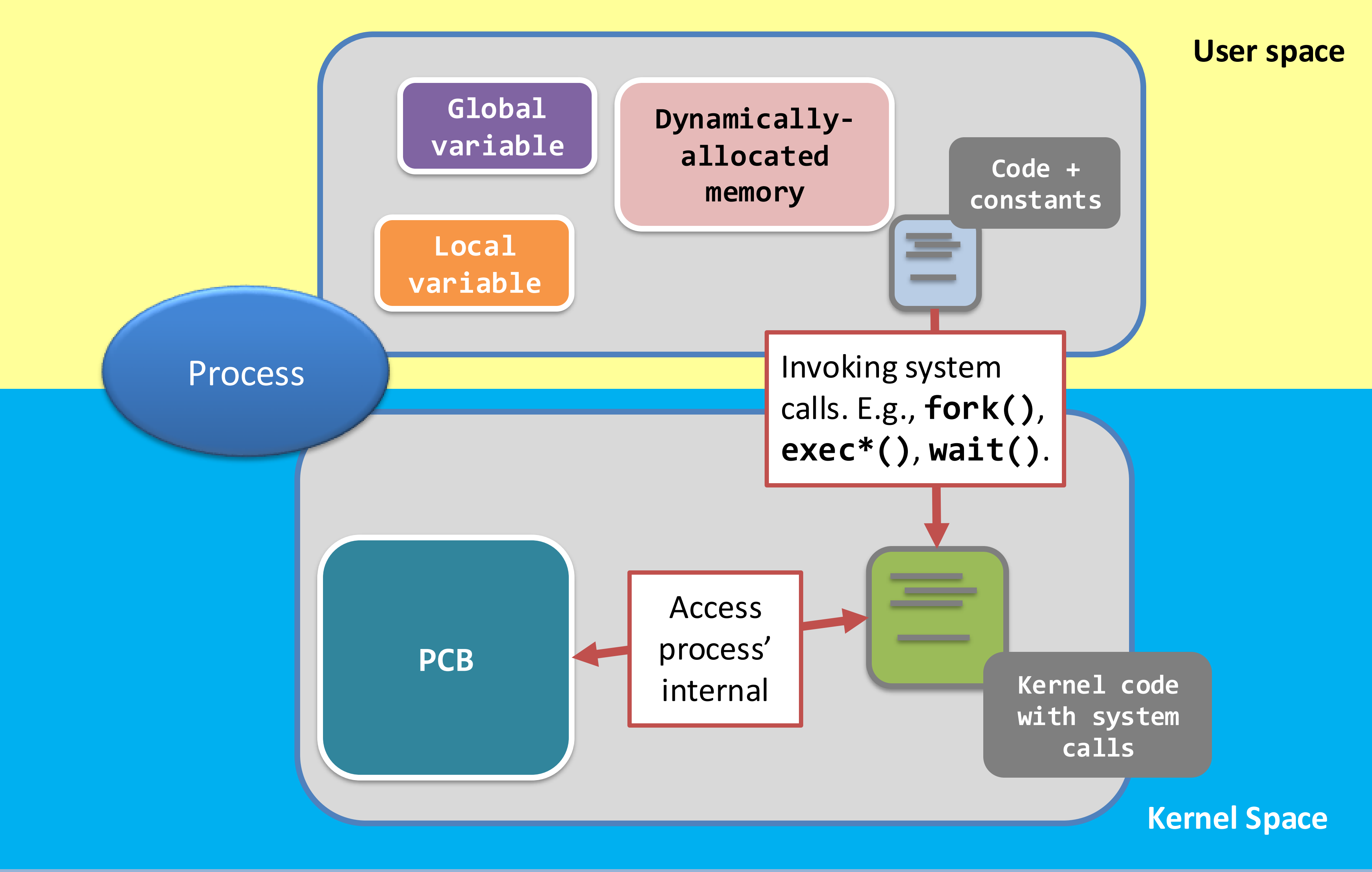
Operating Systems

Eric Lo

4 Process (Kernel Space)

What're happening inside fork, exec, and wait?

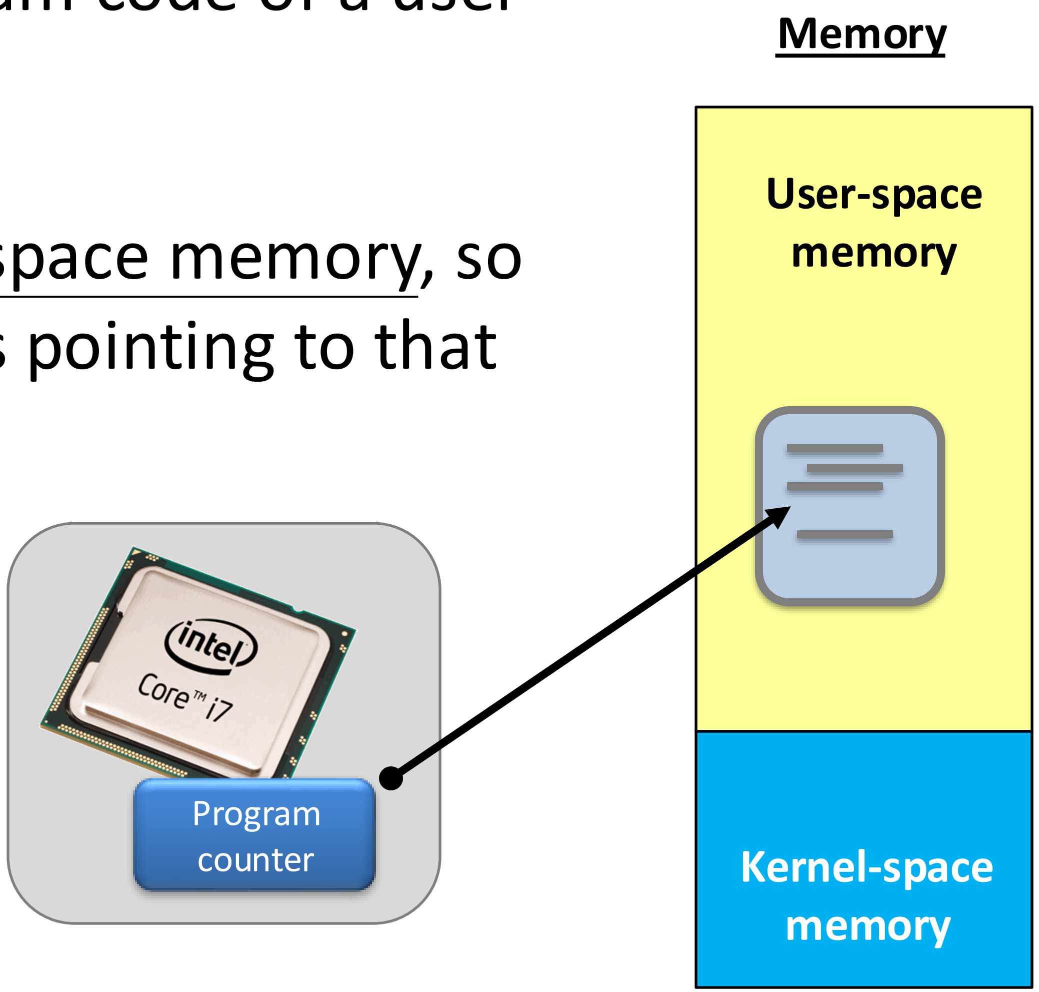
The story so far…



When invoking a system call (memory view)

- When running a program code of a user process.

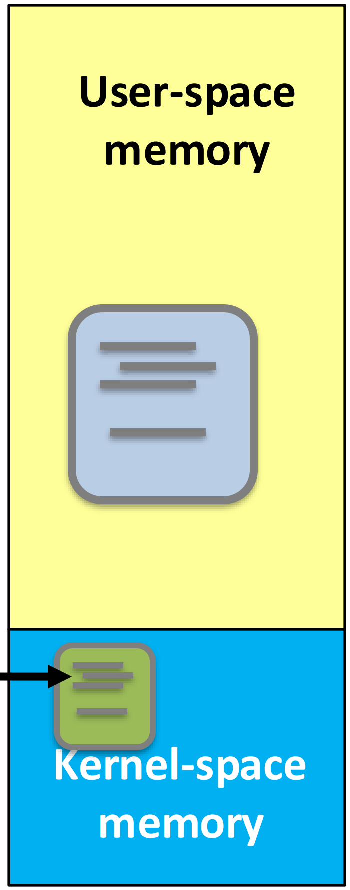
- As the code is in user-space memory, so the program counter is pointing to that region.

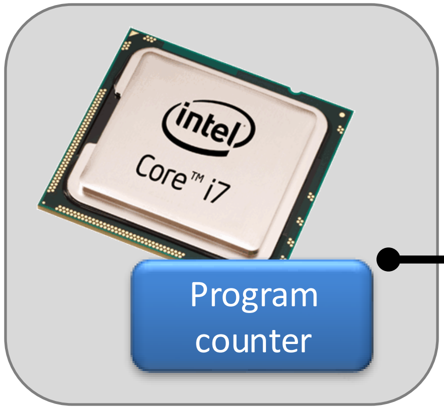


When invoking a system call (memory view)

- 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.

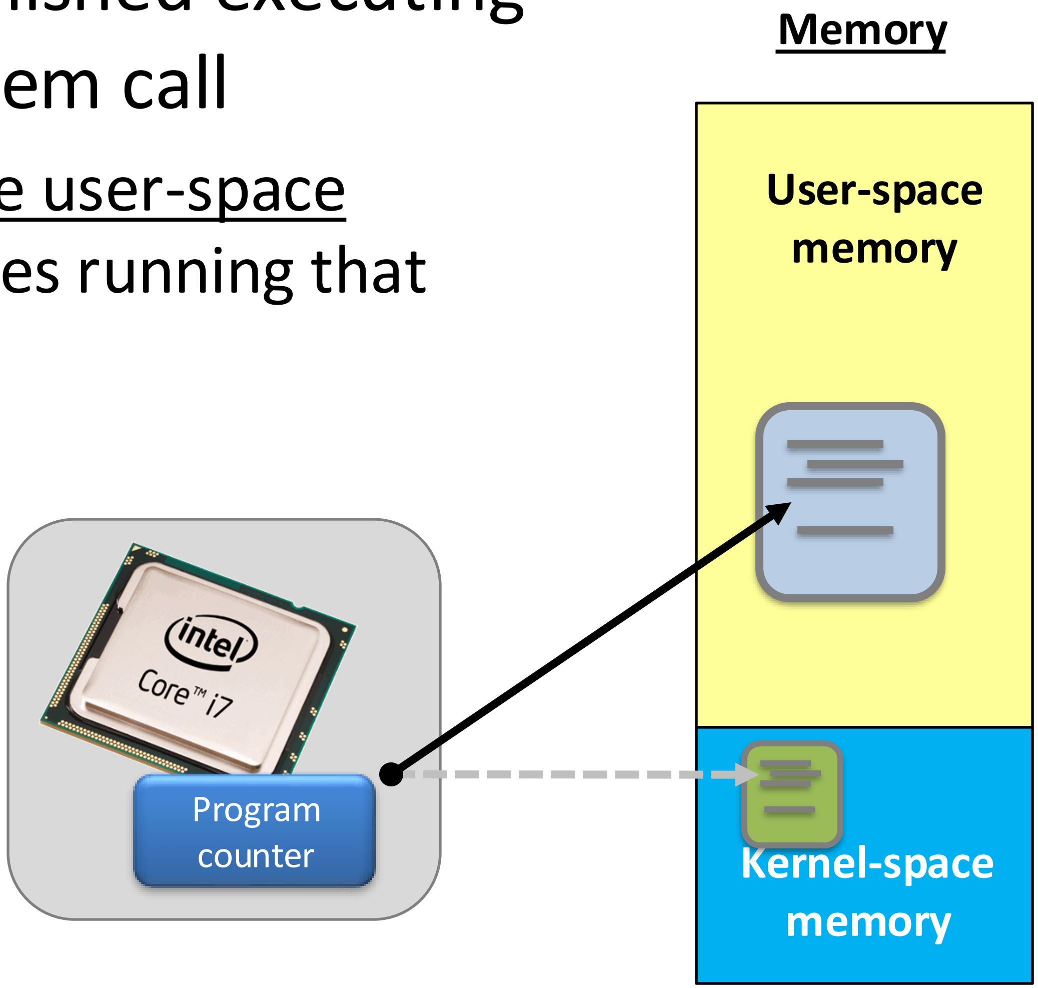
Memory



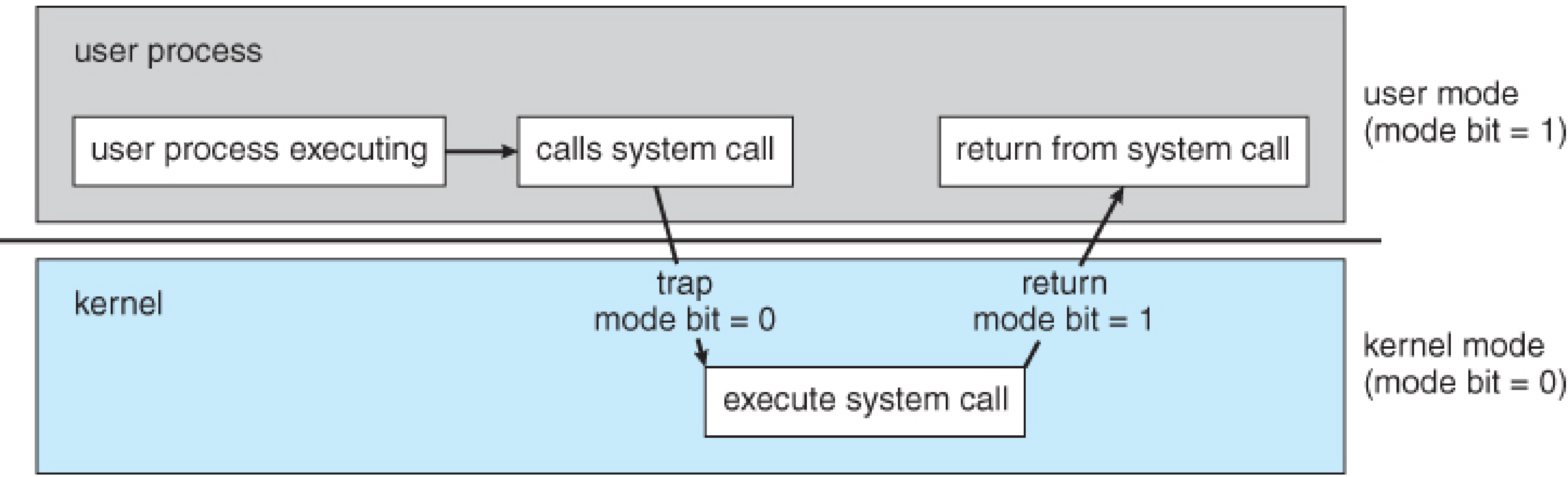


When invoking a system call (memory view)

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



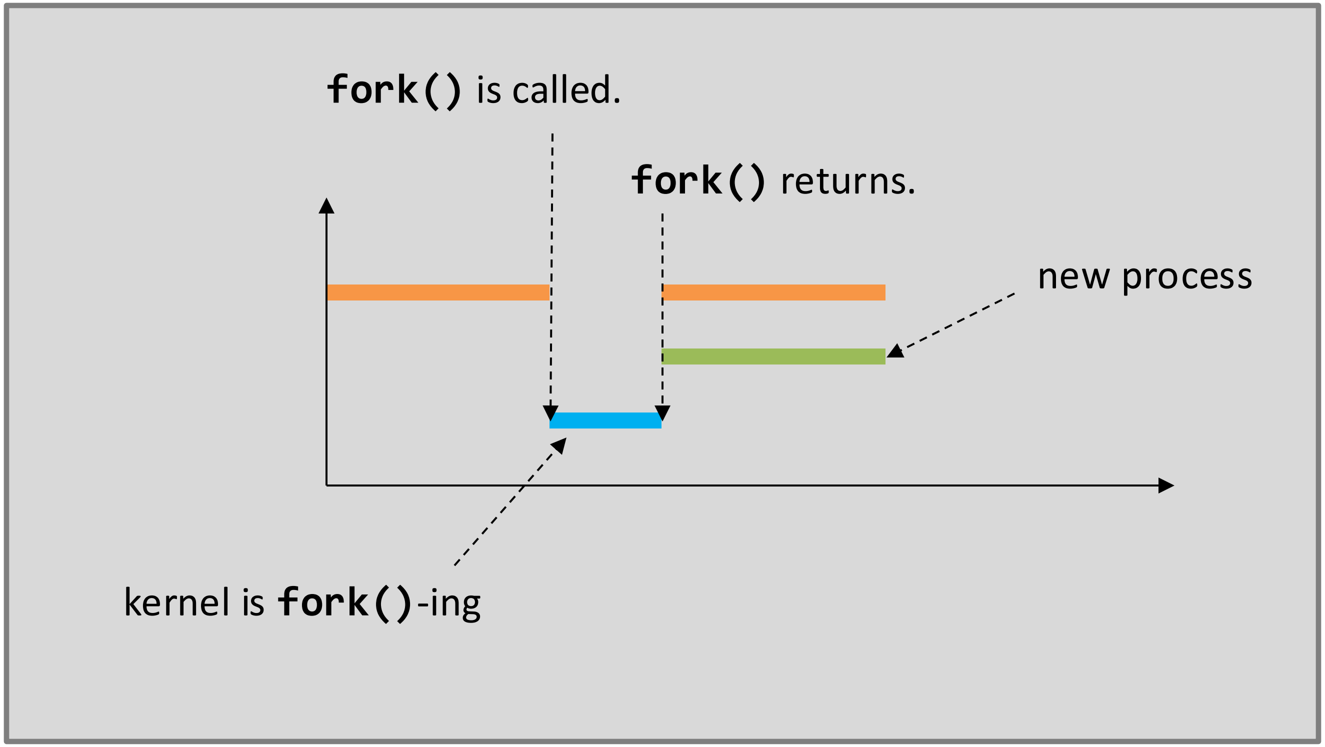
When invoking a system call (CPU view)



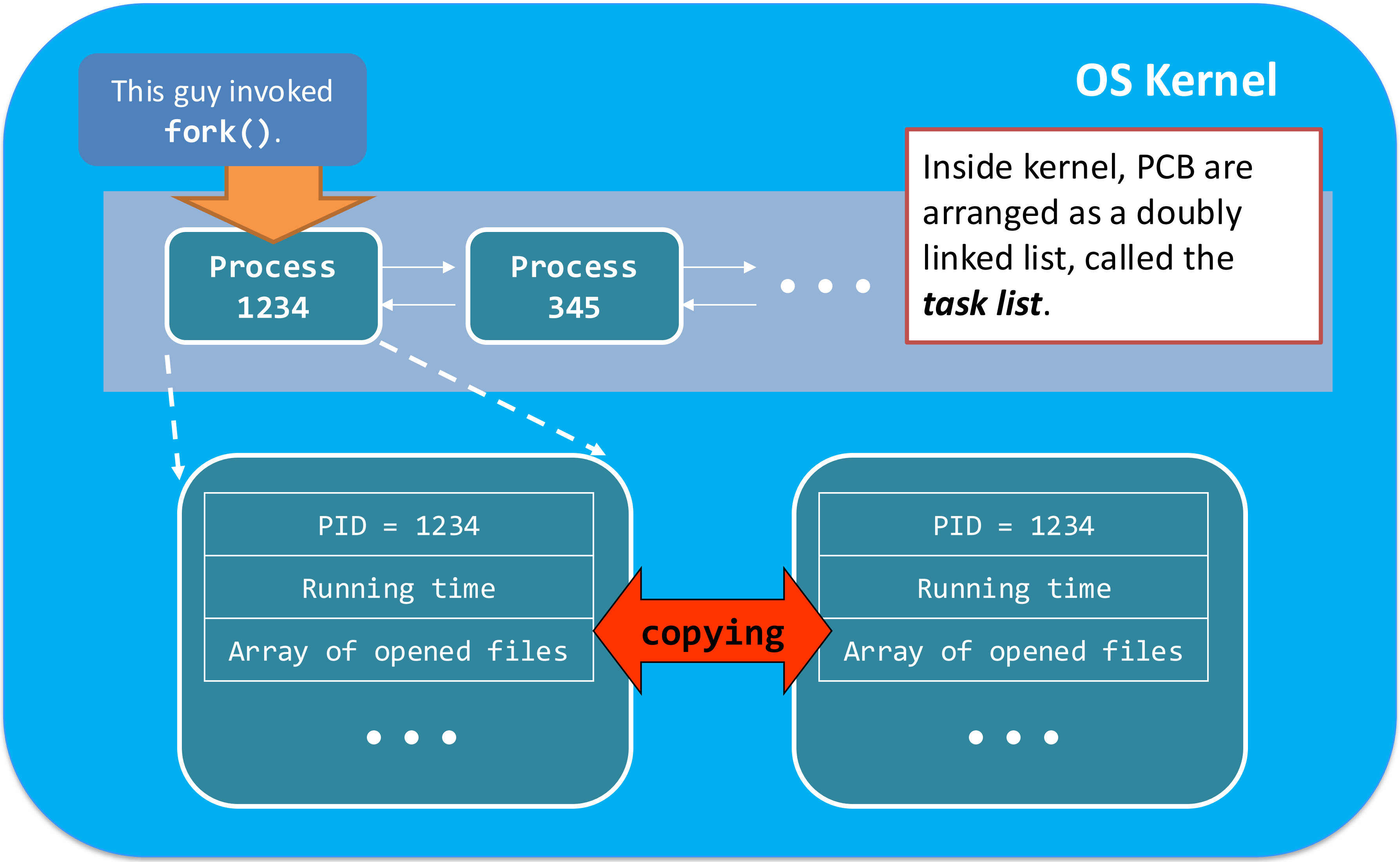
Working of system calls

- `fork();`

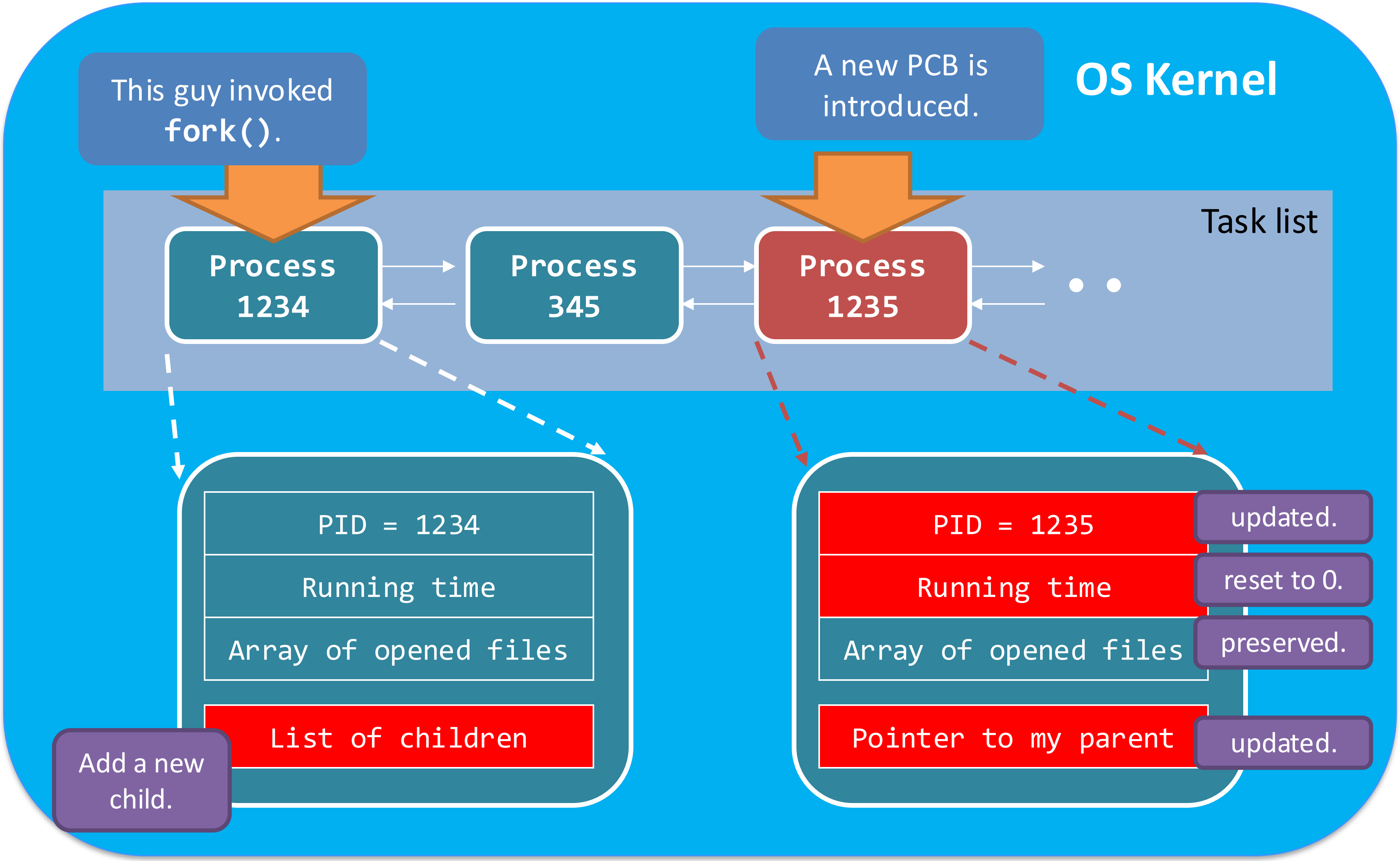
Programmer view of fork()



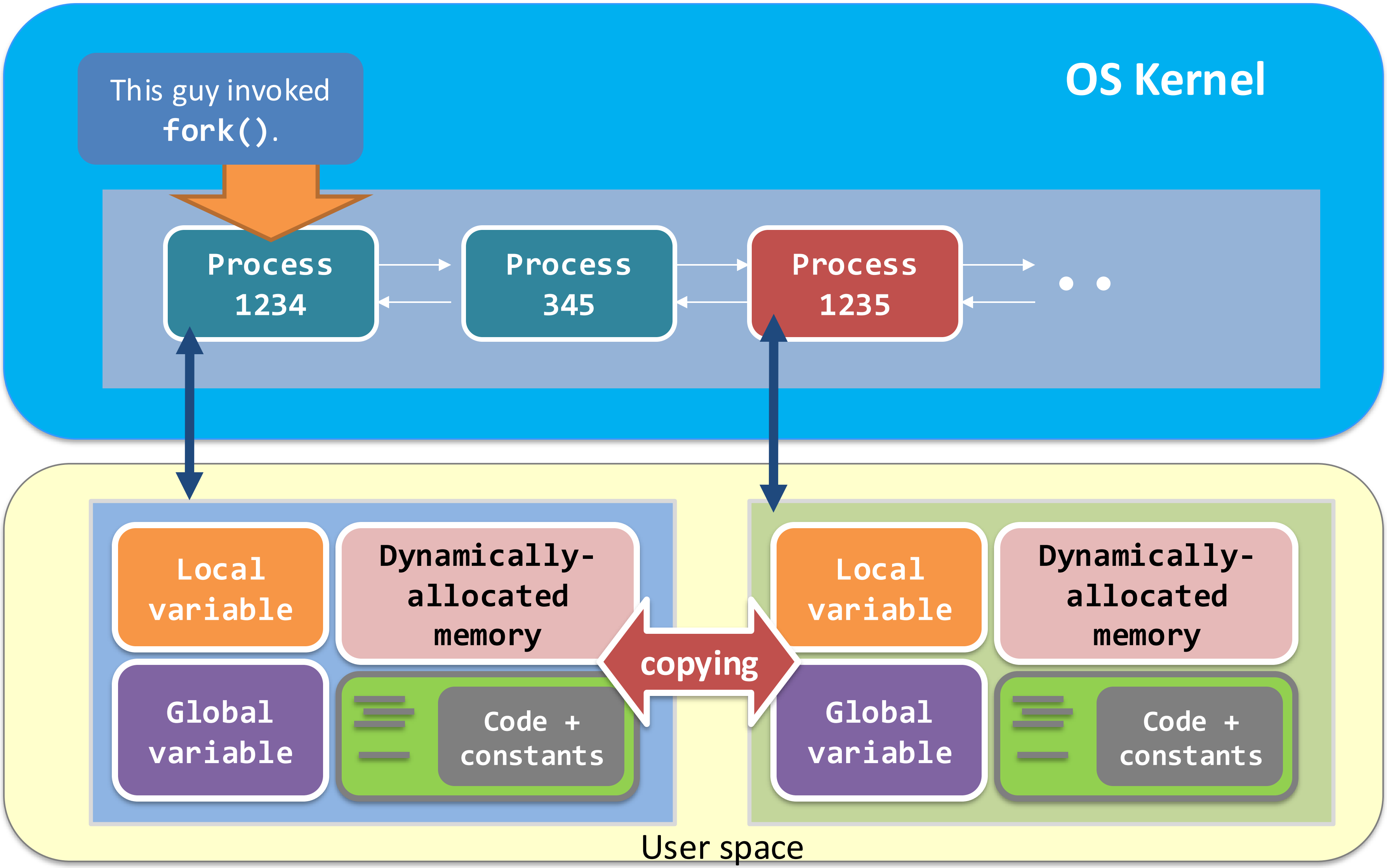
`fork()` inside the kernel



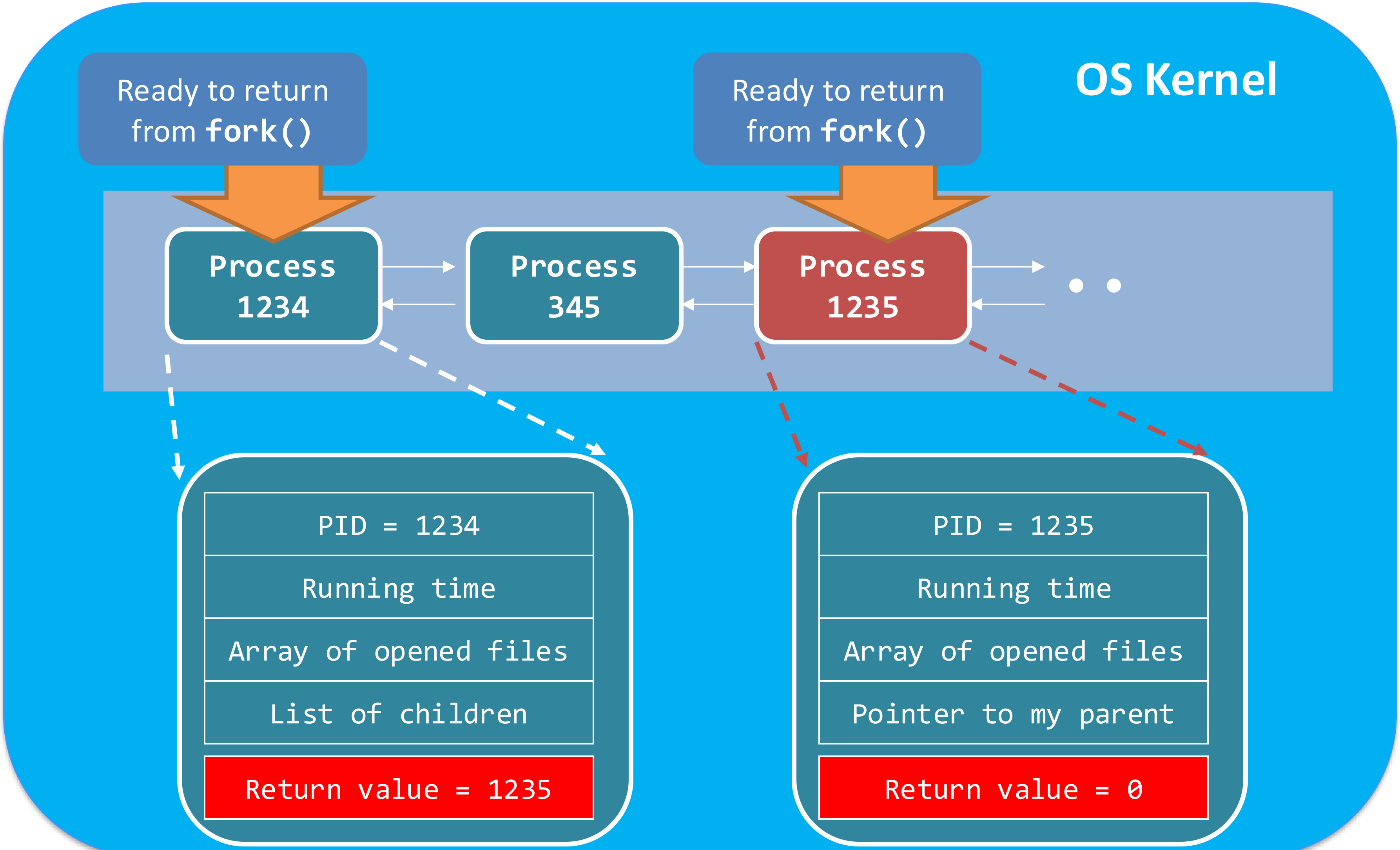
fork() in action – kernel-space update



`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.

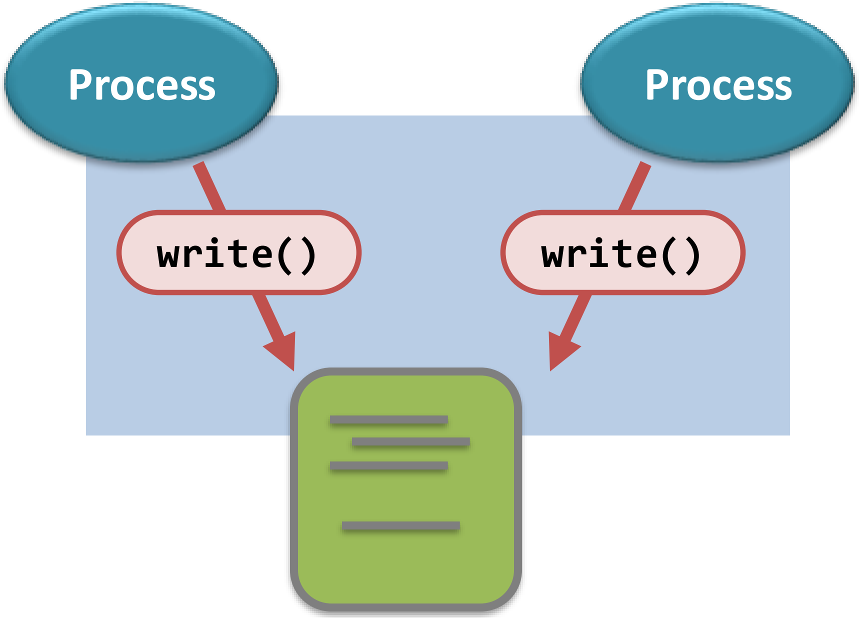
  – That's why a parent process **shares the same terminal output stream** as the child process.

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

Some streams (e.g., network can't random access. Why?

`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!

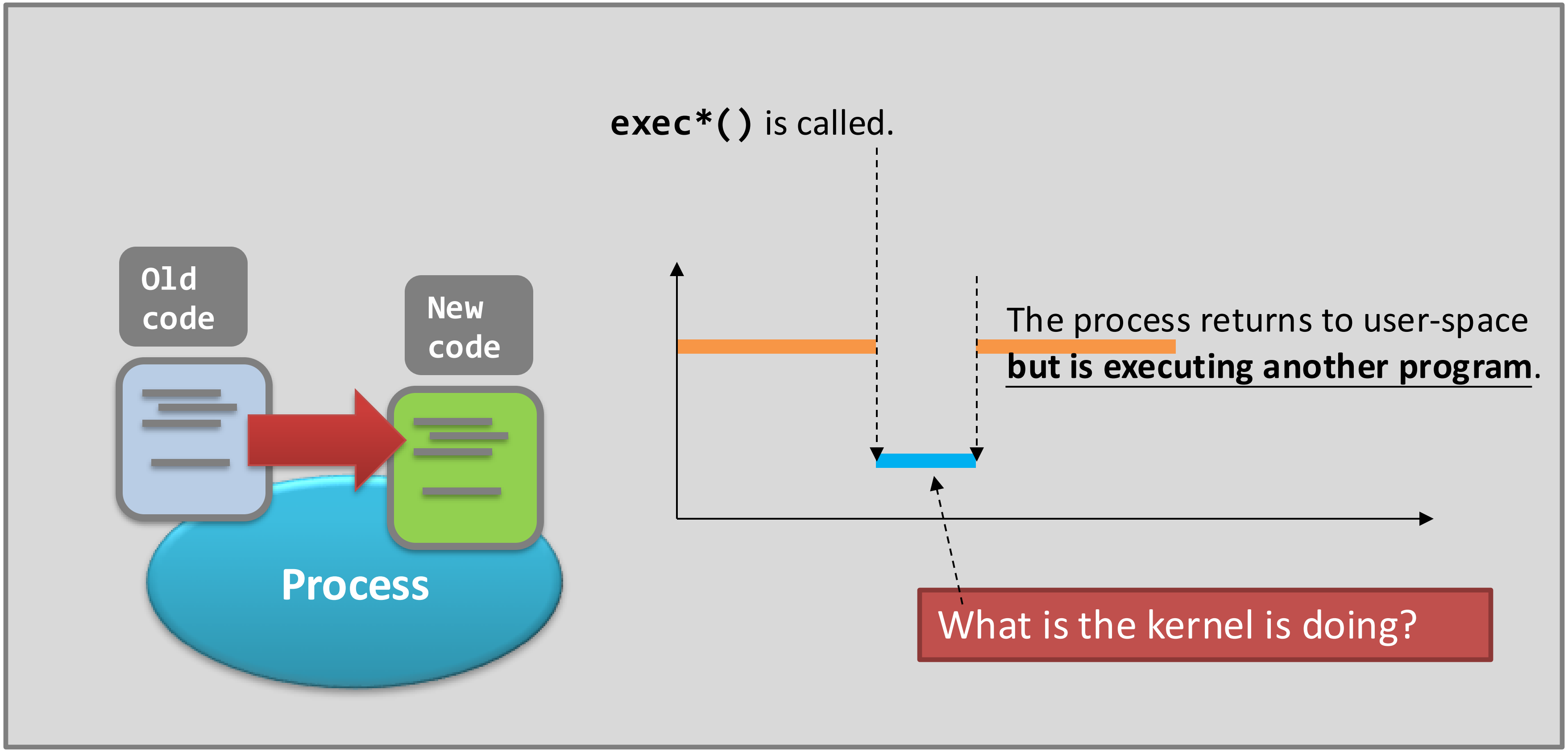
`[example@3150] share_opened_file.c`

Working of system calls

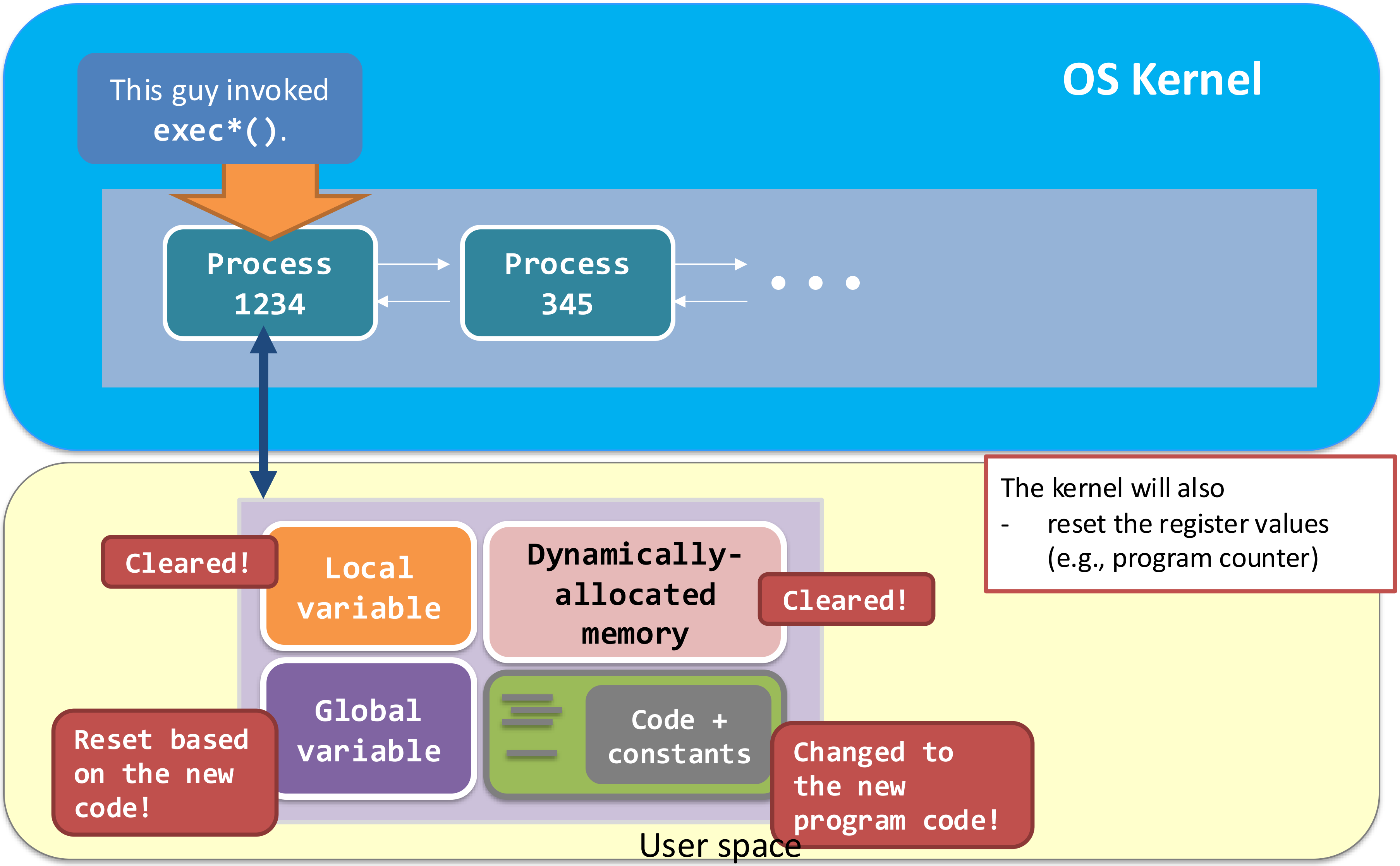
- `fork();`
- `exec*();`

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

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

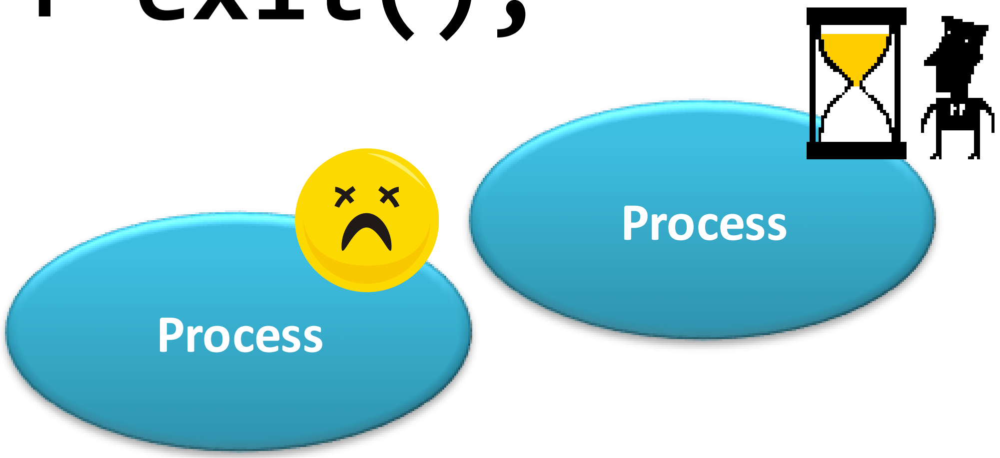


exec*() in action

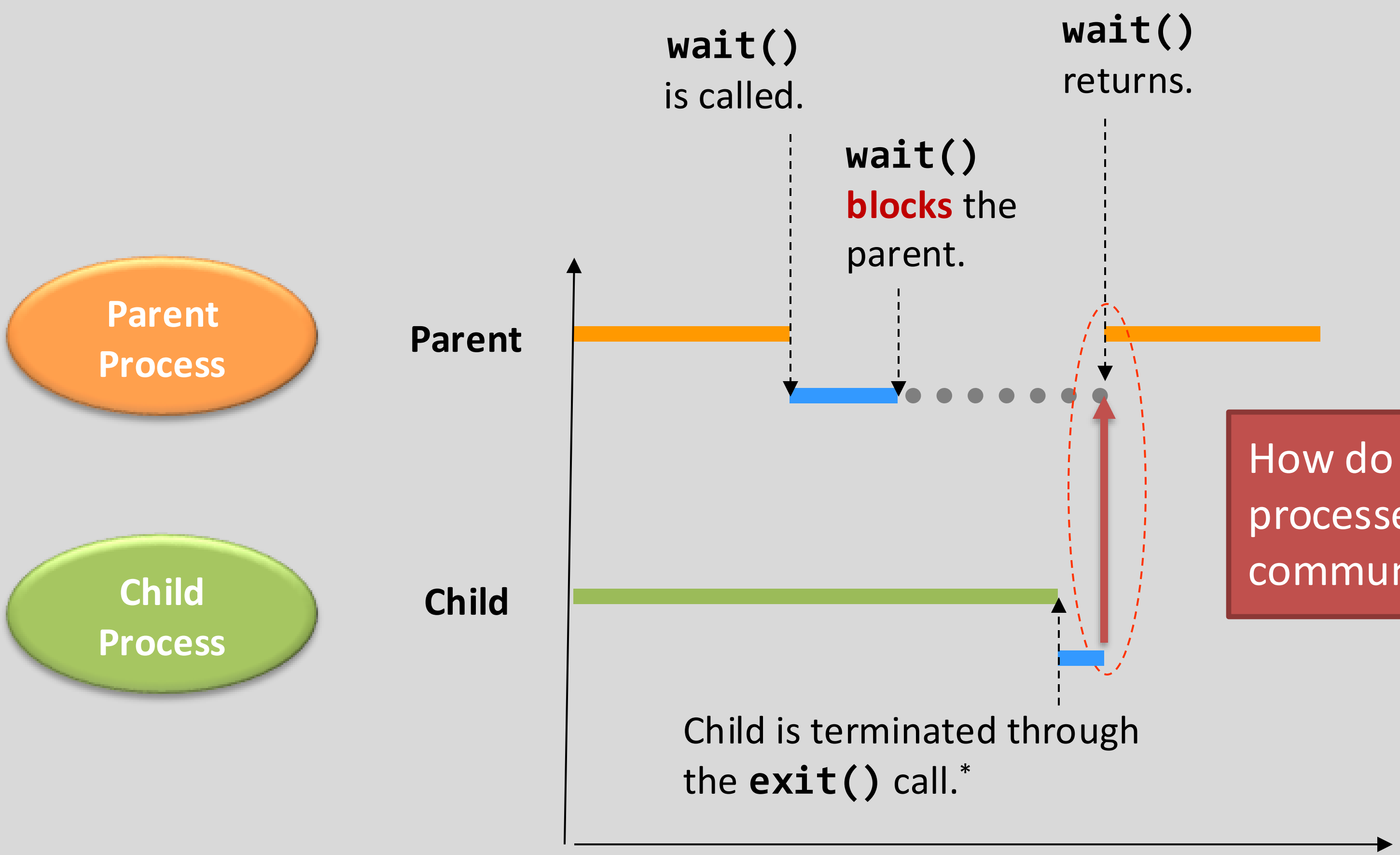


Working of system calls

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



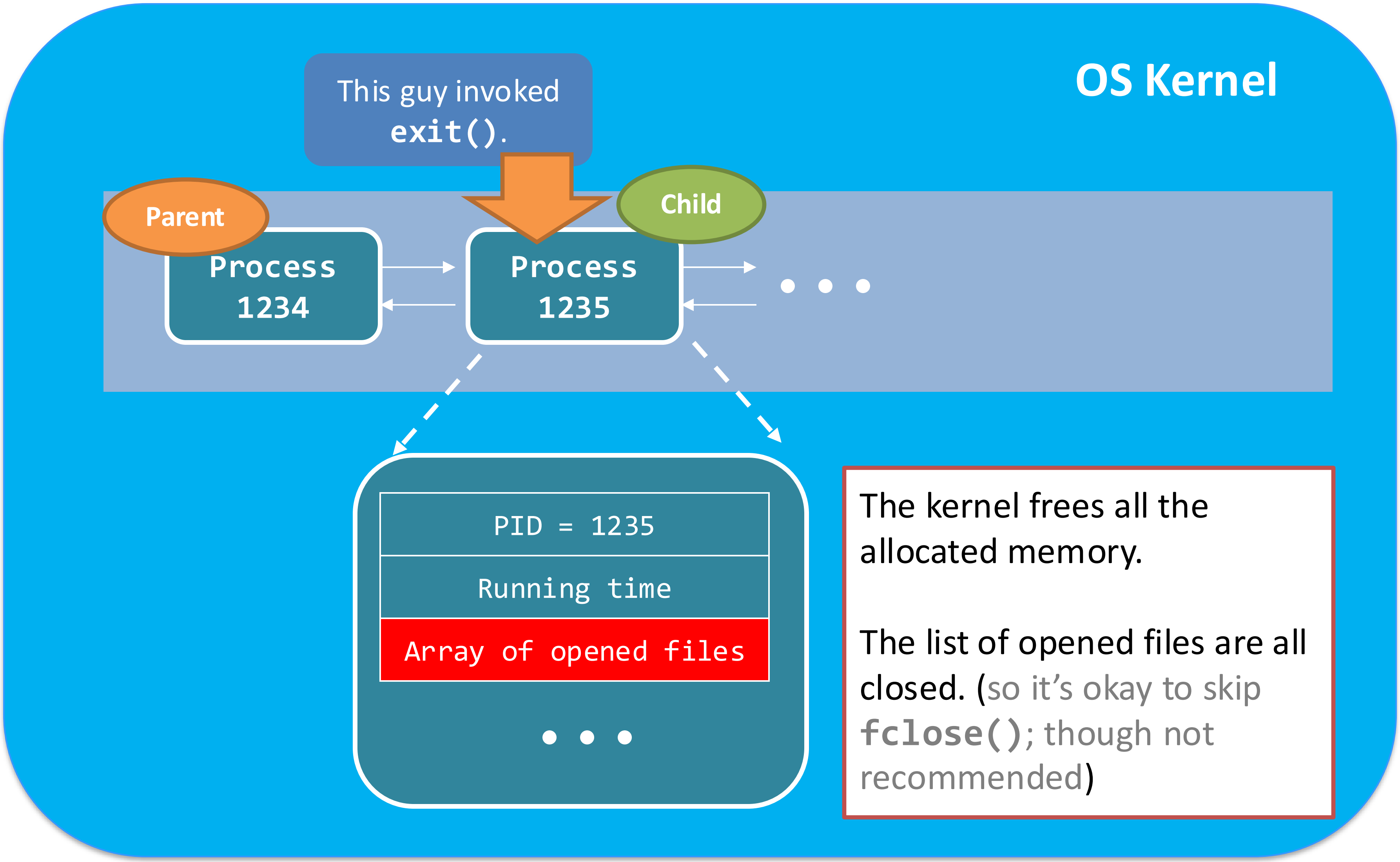
`wait()` and `exit()`



How do the two processes communicate?

*Inserted by the compiler if you don't write it.

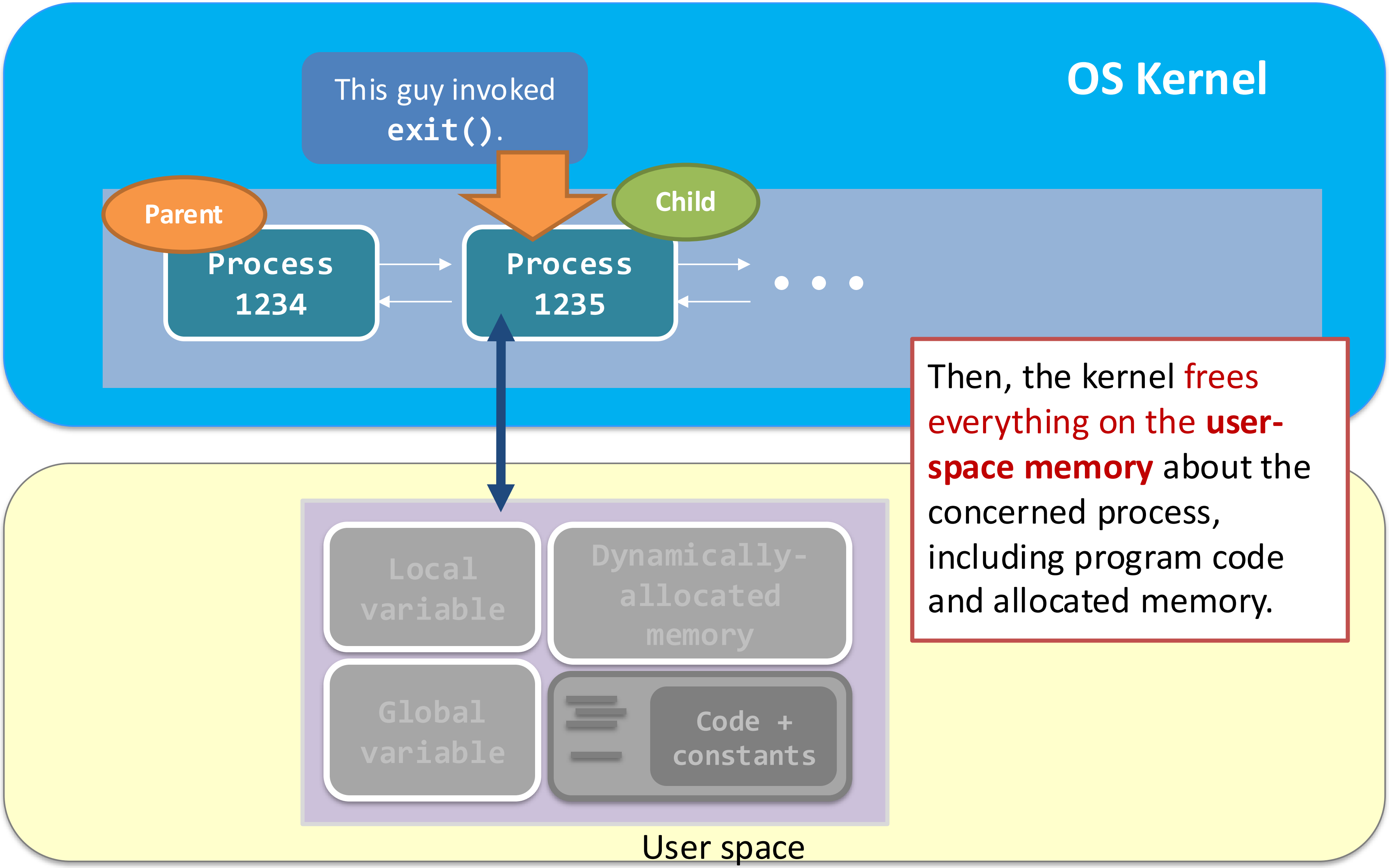
`exit()` (kernel-view)



The kernel frees all the allocated memory.

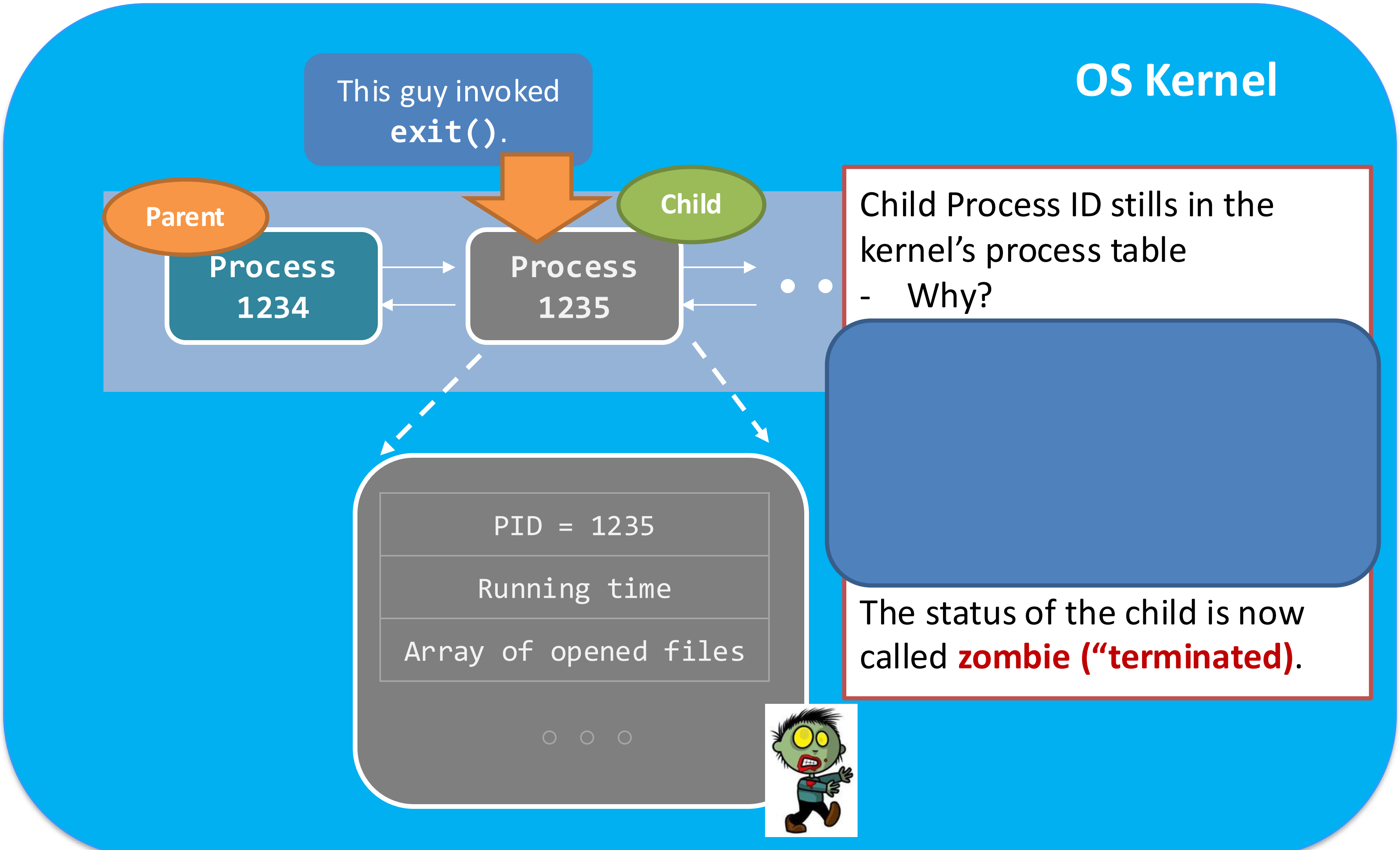
The list of opened files are all closed. (so it's okay to skip `fclose()`; though not recommended)

`exit()` (kernel-view)

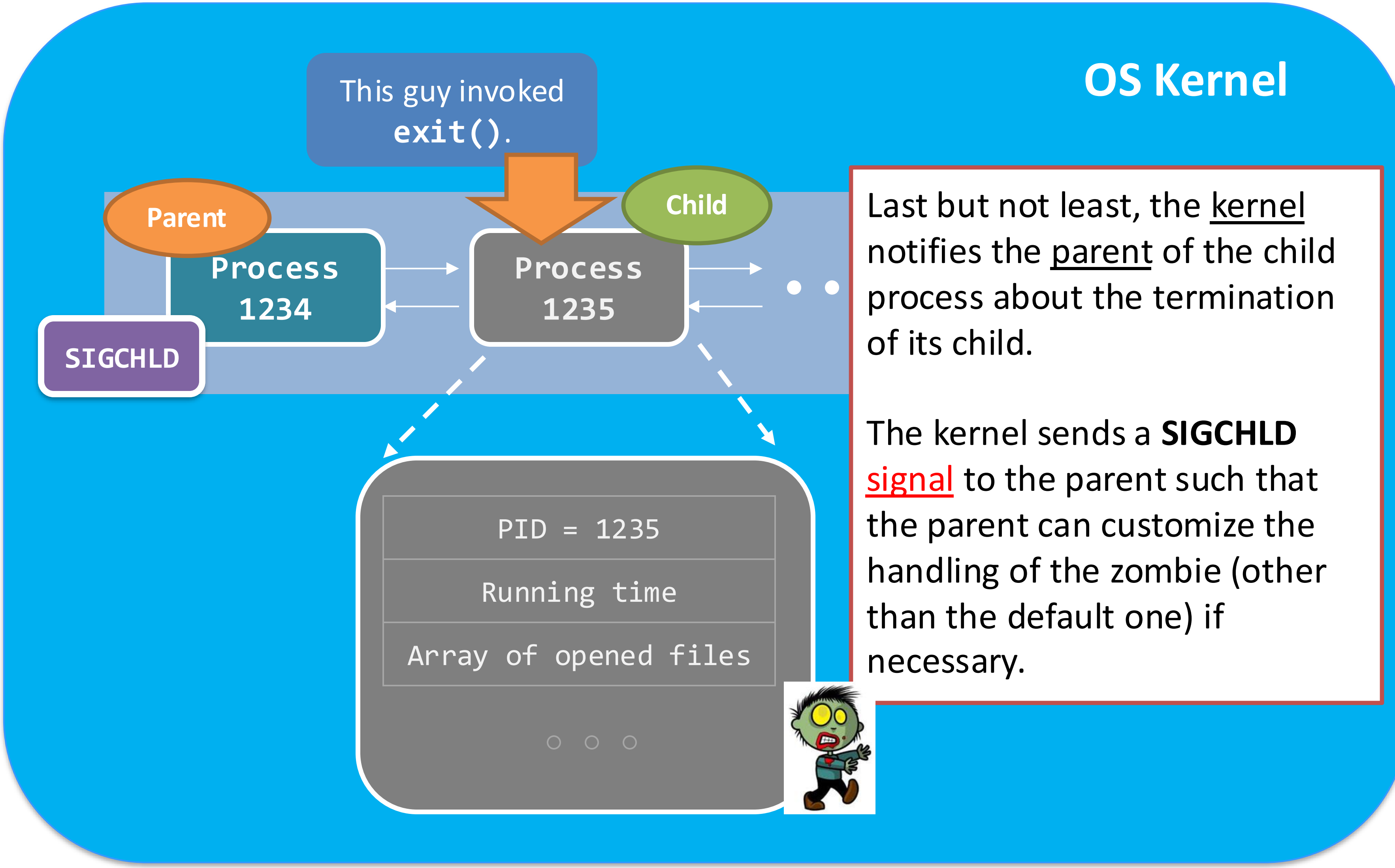


Then, the kernel frees everything on the **user-space memory** about the concerned process, including program code and allocated memory.

`exit()` (kernel-view)



exit() (kernel-view)



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

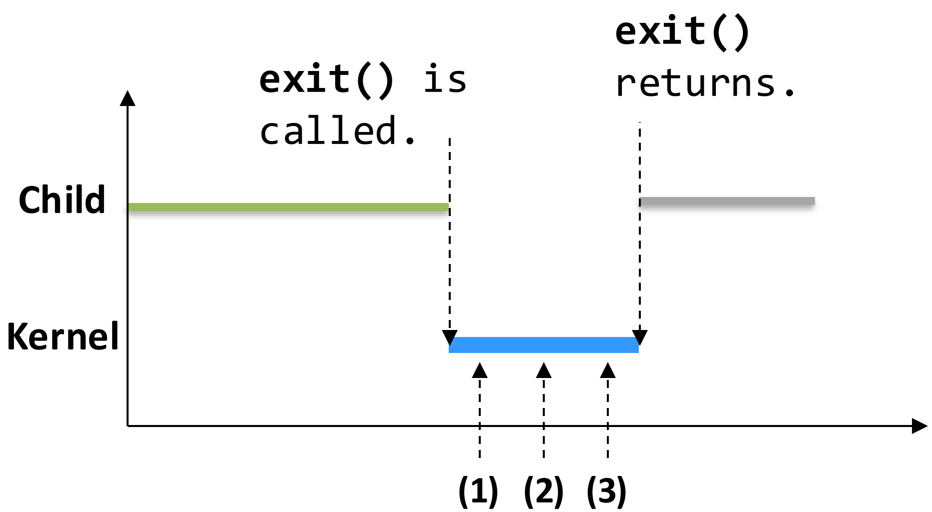
The kernel sends a **SIGCHLD** signal to the parent such that the parent can customize the handling of the zombie (other than the default one) if necessary.

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

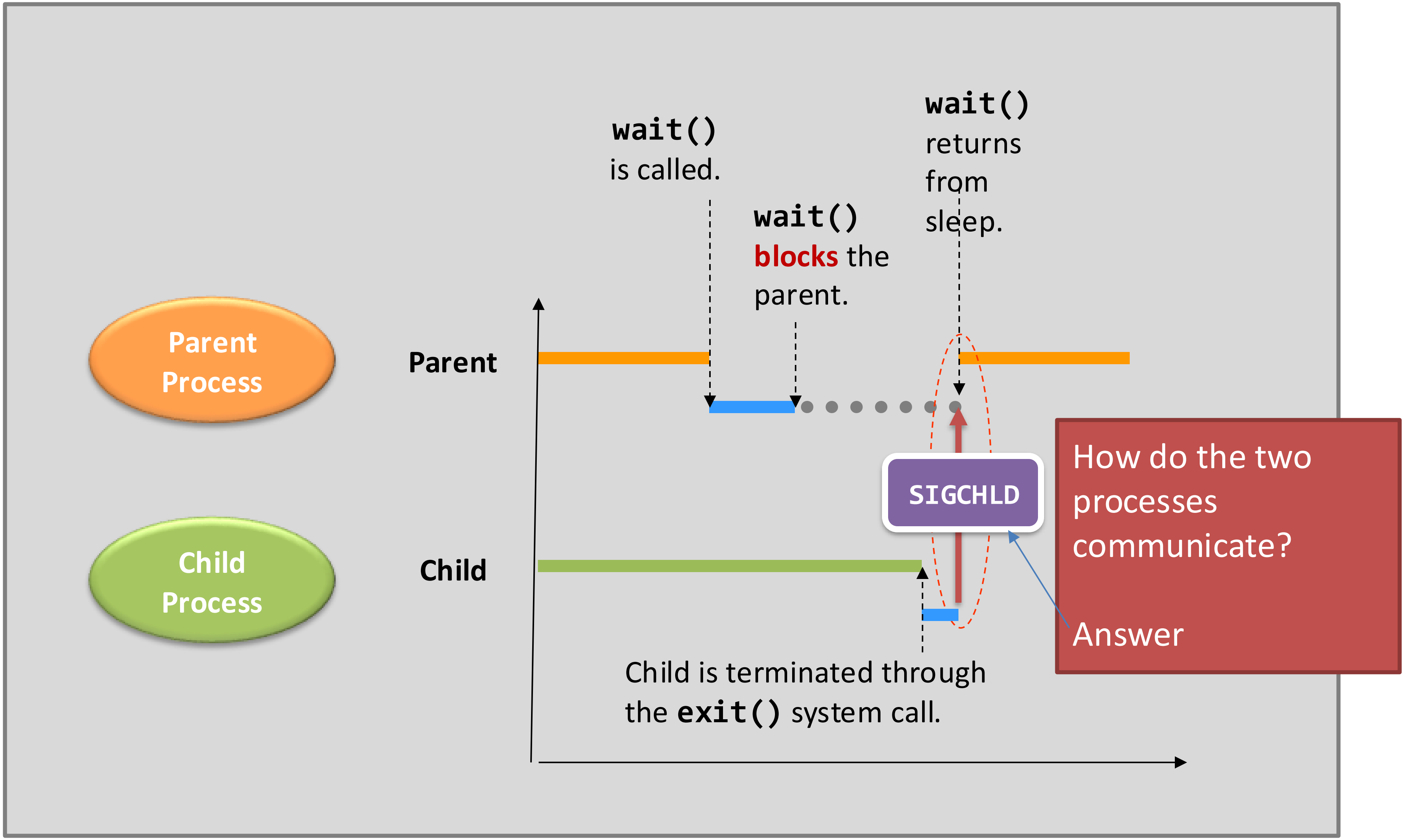
Step (1) Clean up most of the child's PCB data in the kernel

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

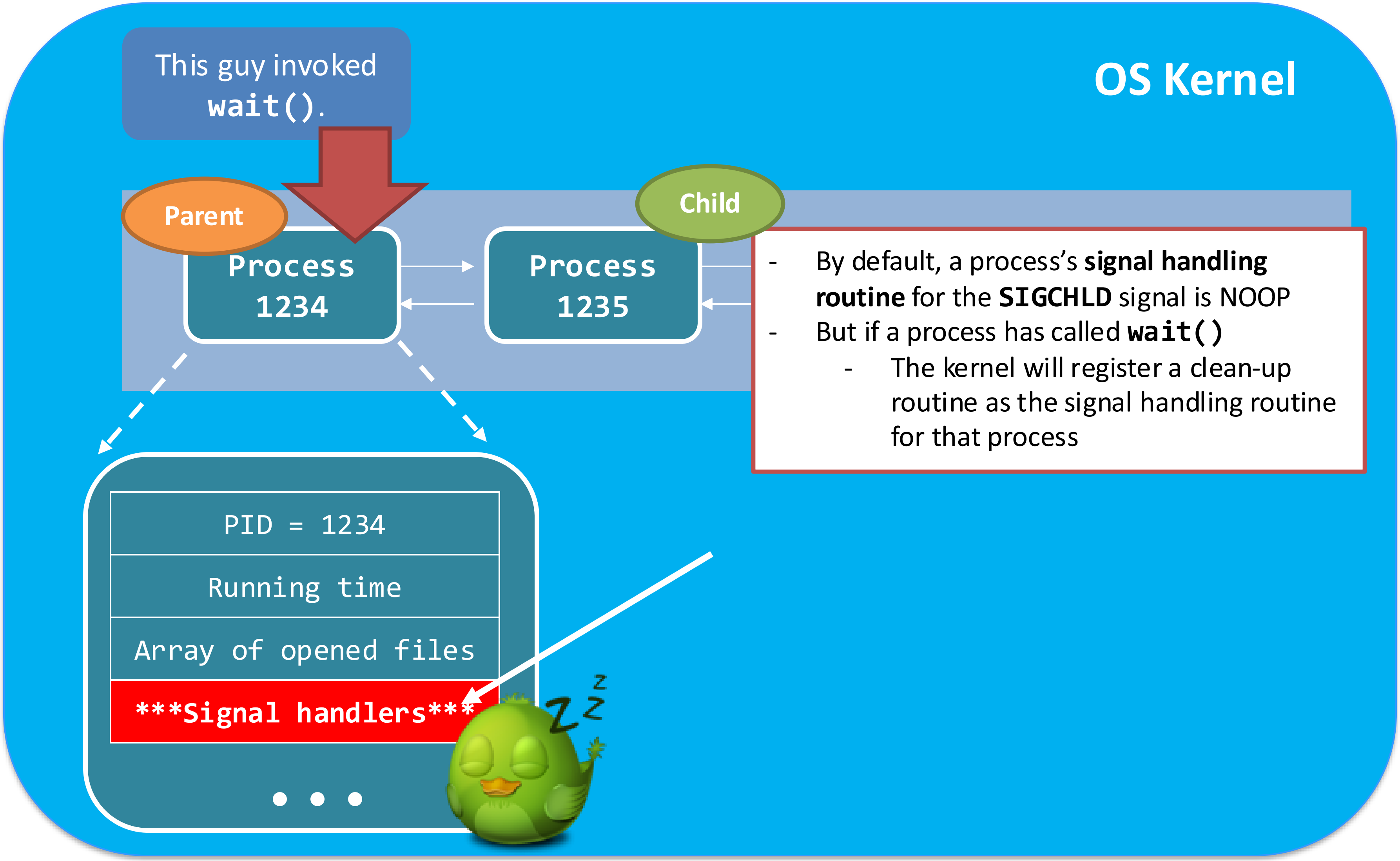
Step (3) Notify the parent with SIGCHLD.



`wait()` and `exit()`



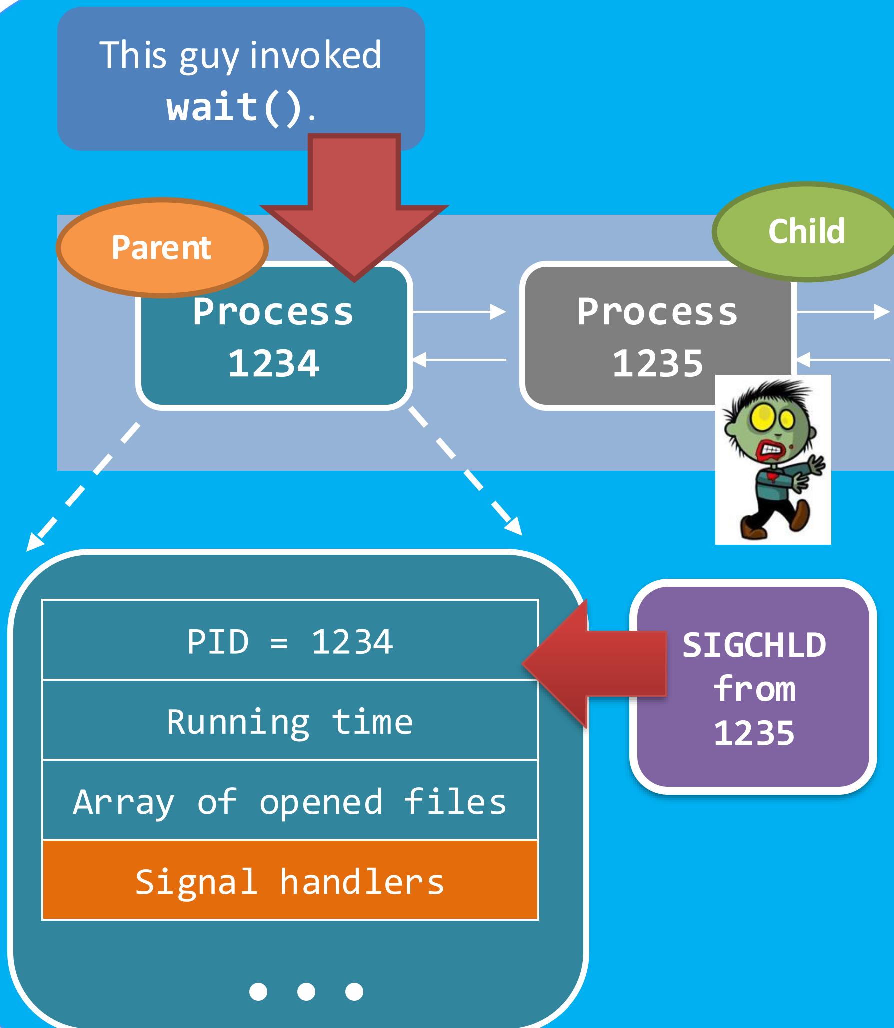
wait() kernel view's – registering signal handling routine



- By default, a process's **signal handling routine** for the **SIGCHLD** signal is NOOP
- But if a process has called **wait()**
  - The kernel will register a clean-up routine as the signal handling routine for that process

`wait()` kernel's view

OS Kernel

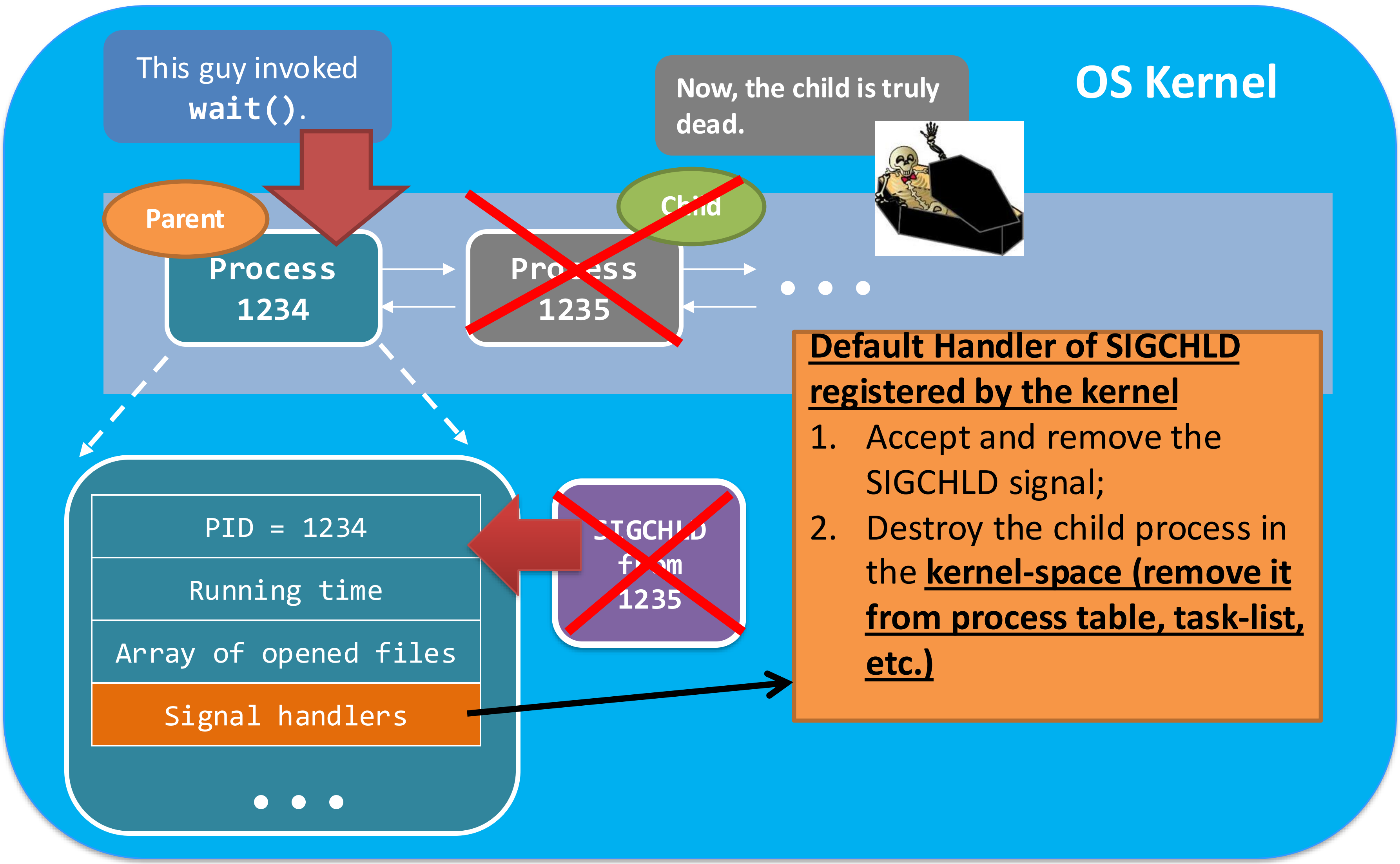


When **SIGCHLD** comes, the corresponding **signal handling routine is invoked**!

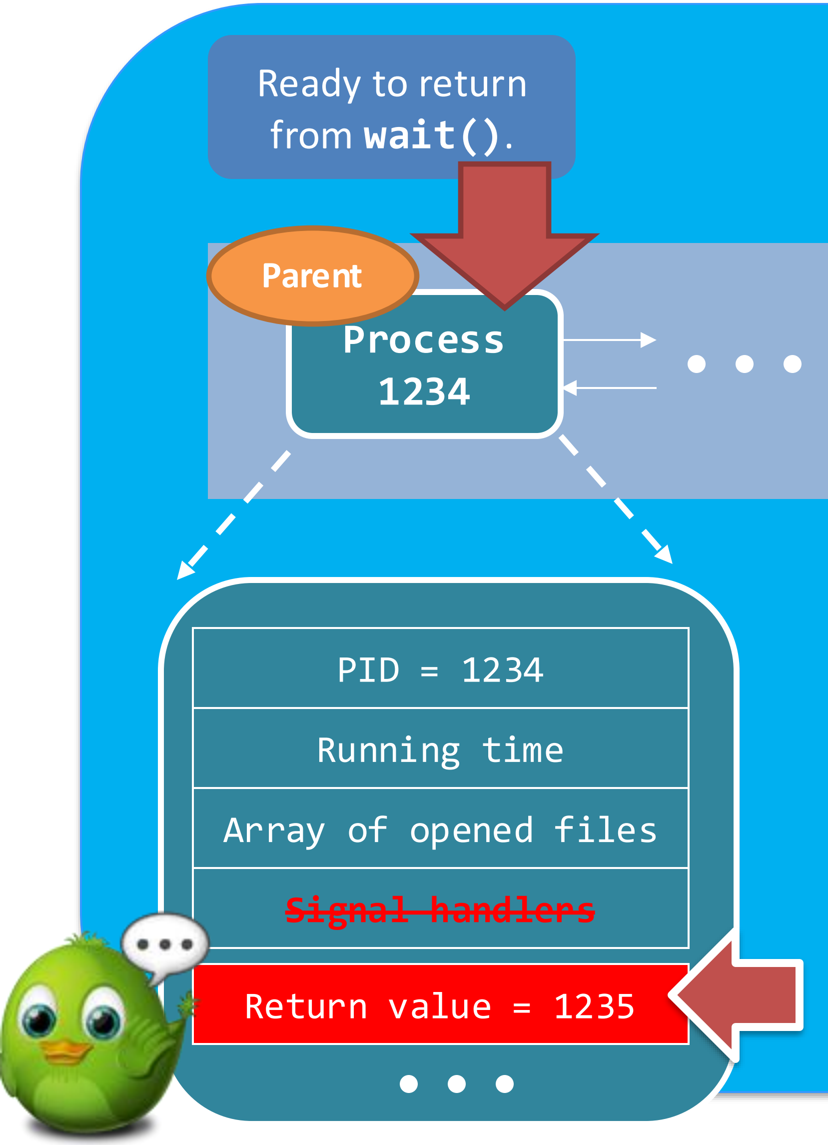
Note:

- the parent is still inside the **`wait()`** system call

`wait()` kernel's view



`wait()` kernel's view

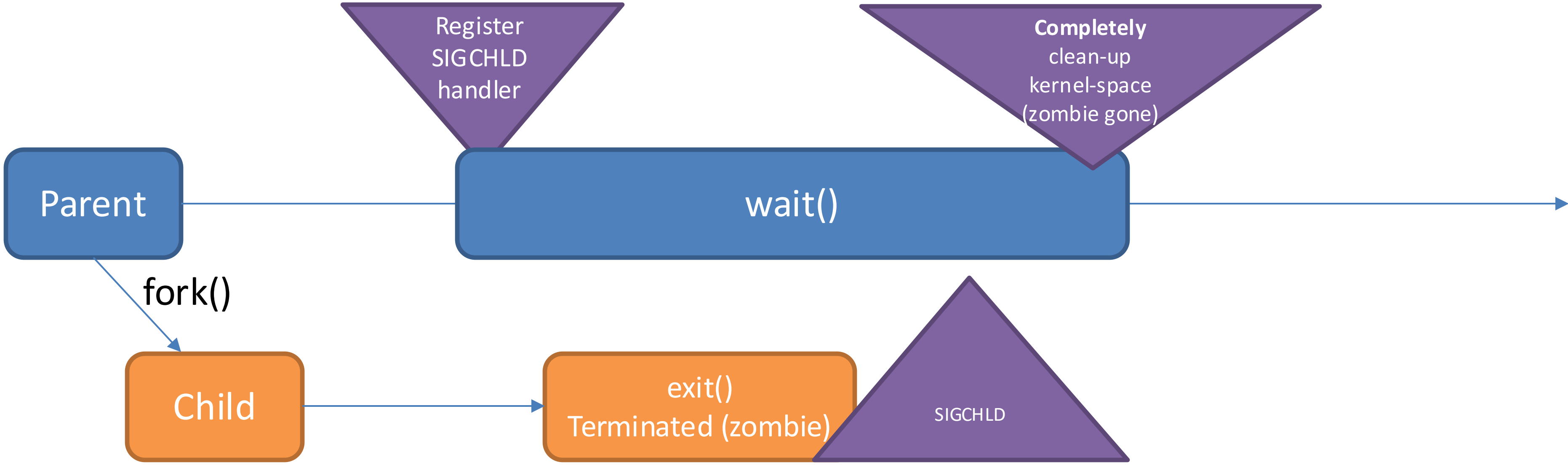


The kernel

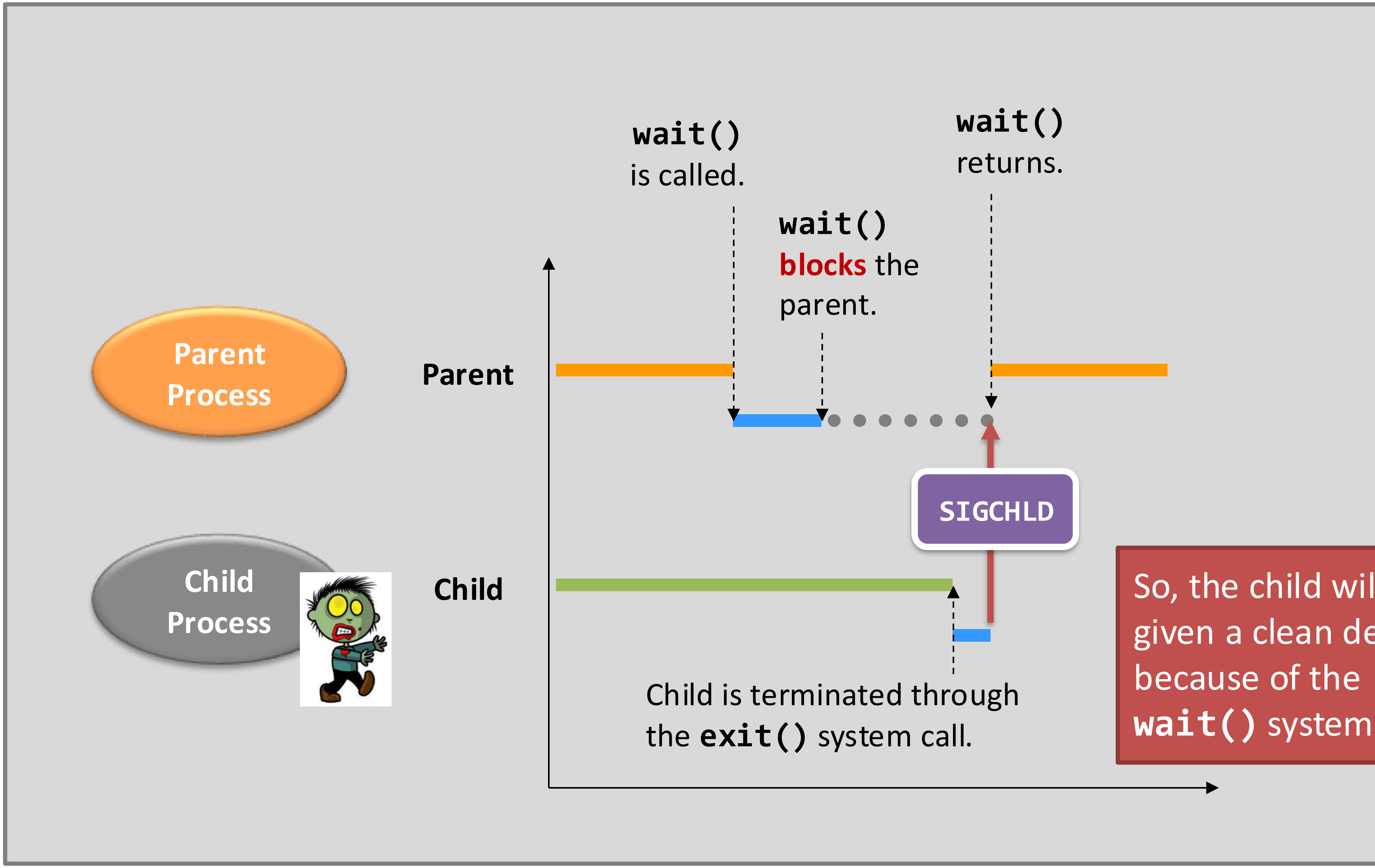
- deregisters the **signal handling routine** for the parent
- returns the PID of the terminated child as the return value of **`wait()`**

The parent's **SIGCHLD handling routine** is NOOP again.

Overall – normal case

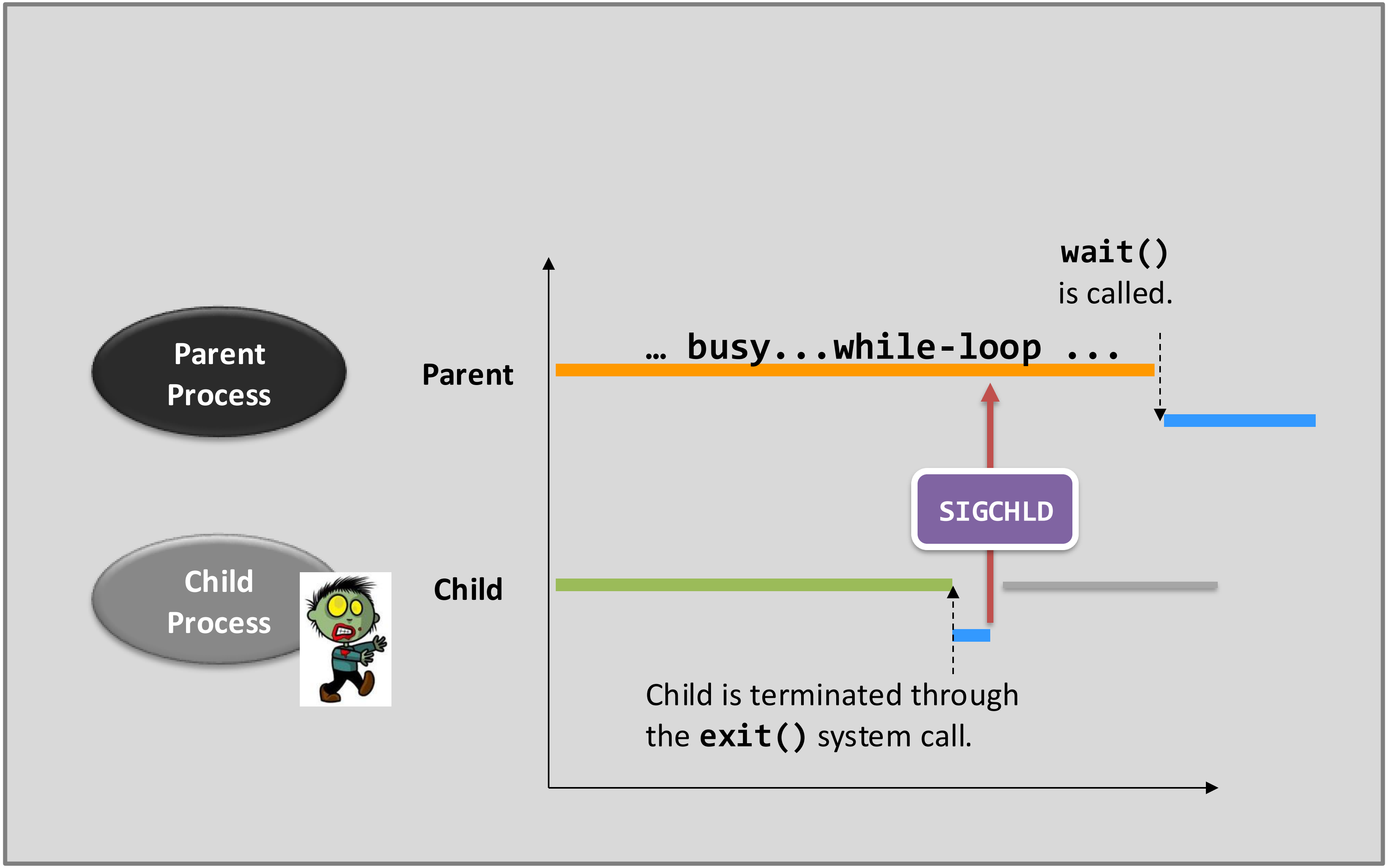


Normal Case

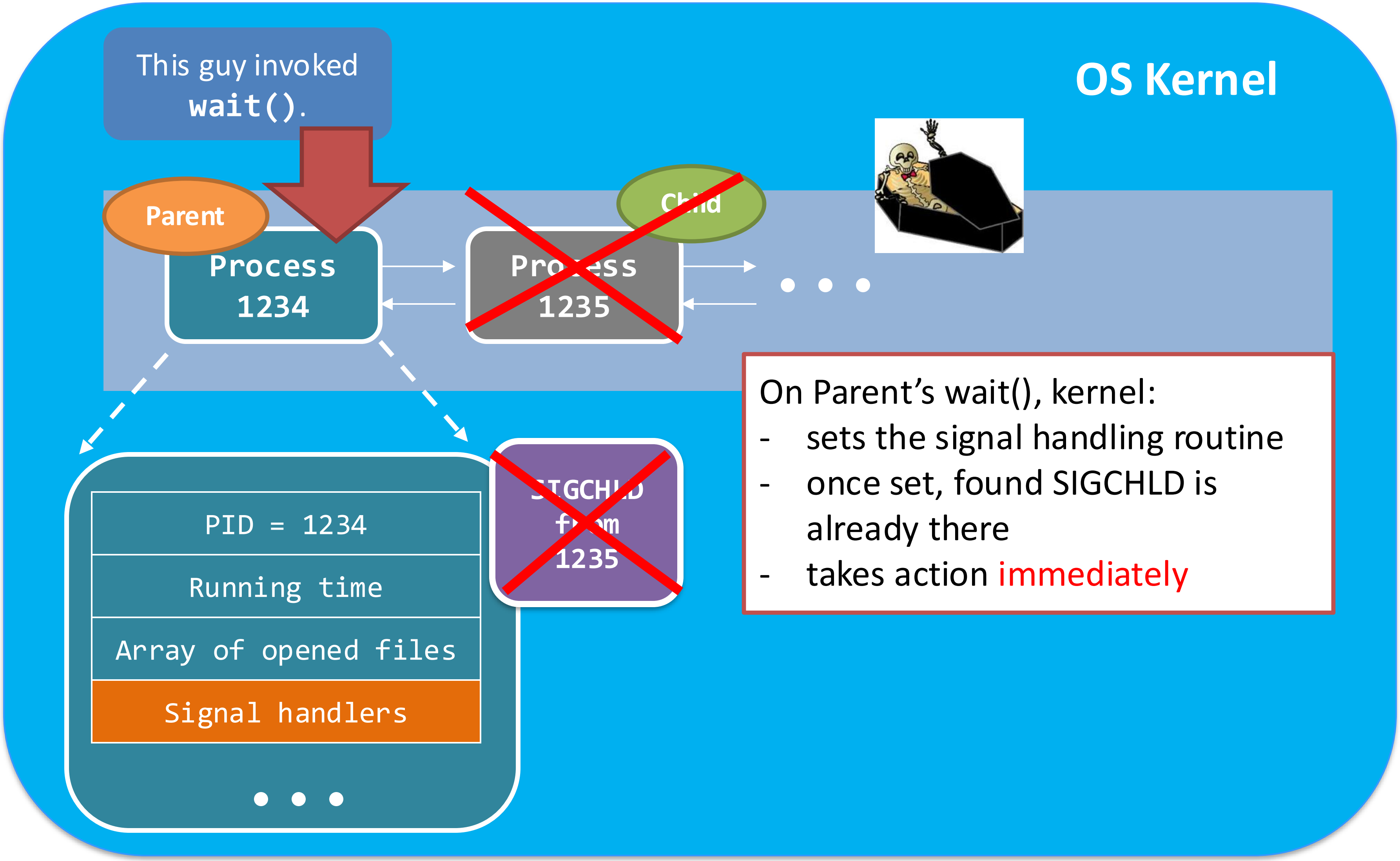


So, the child will be given a clean death because of the **wait()** system call.

Another case:
Parent's wait() after Child's exit()



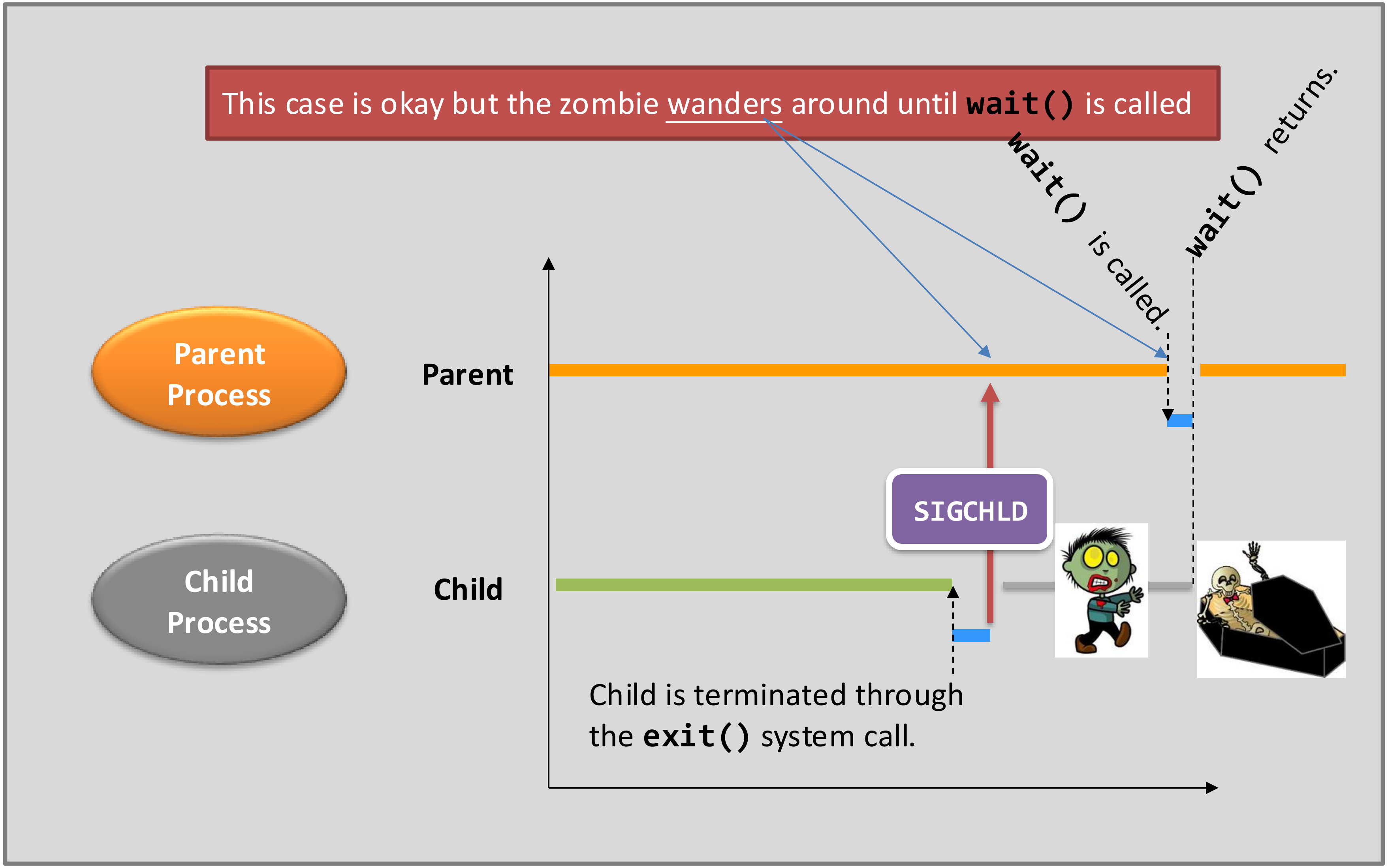
Parent's Wait() after Child's exit()



On Parent's wait(), kernel:

- sets the signal handling routine
- once set, found SIGCHLD is already there
- takes action immediately

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

- Parent's **wait()** & **waitpid()** syscall are the ones who register the default handler that reaps zombie child processes.
- 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

```
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".

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

Working of system calls

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

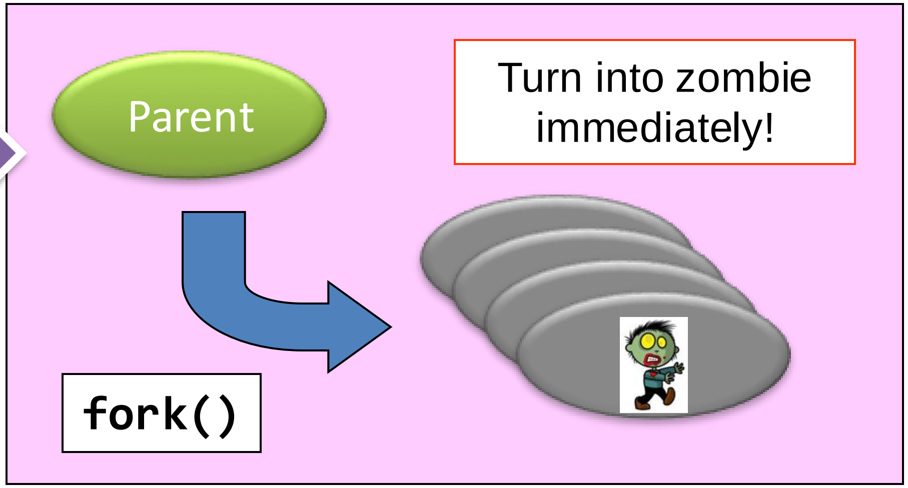
Calling **wait()** is important.

- 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 fork() without wait()
- **Don't try this on department's machines…**

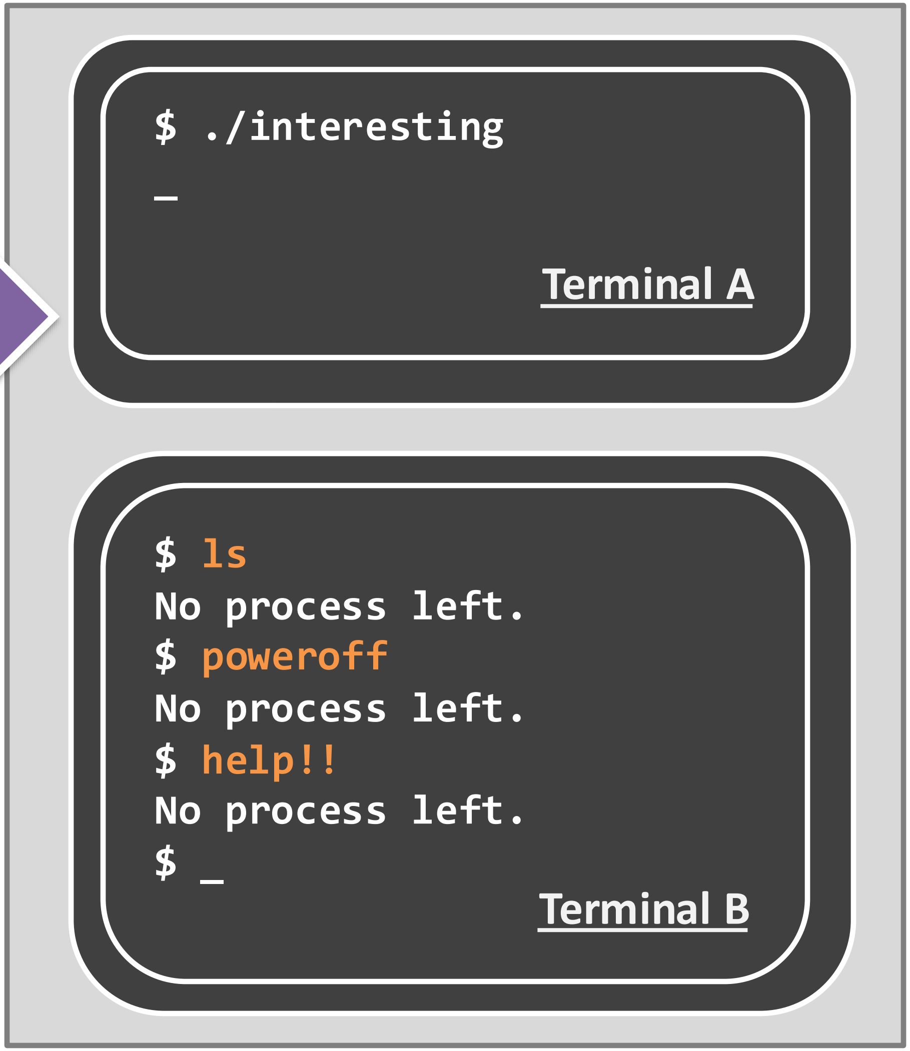
```c
int main(void) {
    while( fork() );
    return 0;
}
```



An infinite, zombie factory!

The fork bomb

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



The first process

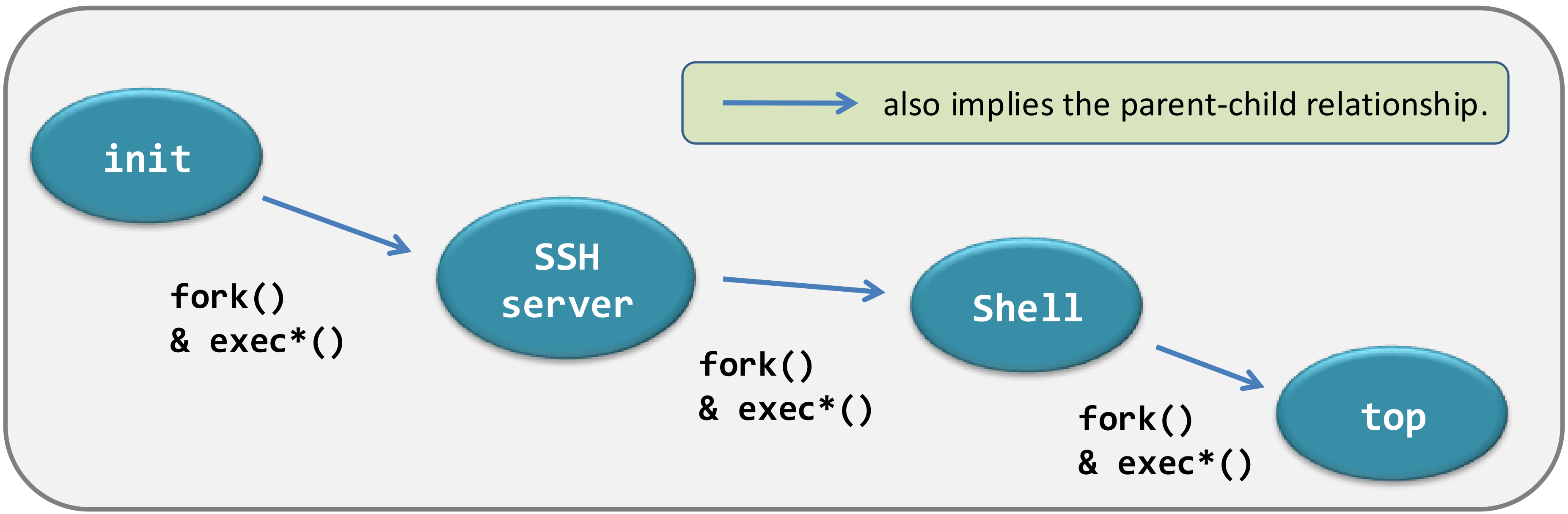
- We now focus on the process-related events.
  - The kernel, while it is booting up, creates the first process – **`init`**.

- 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*()`**.

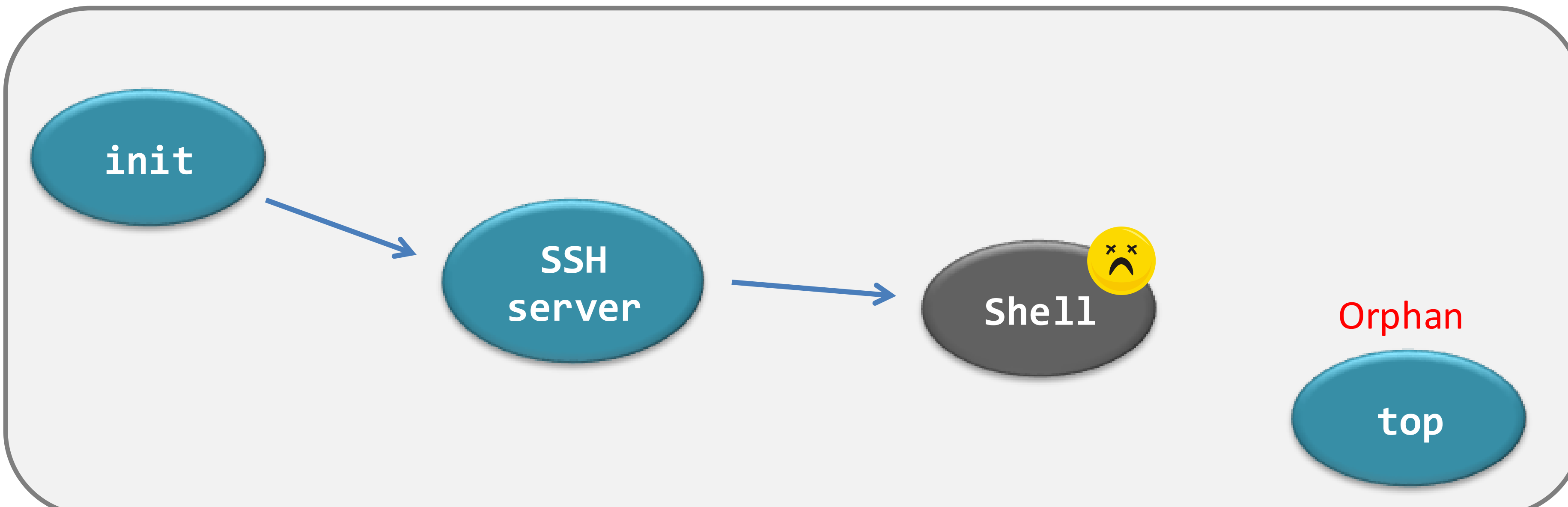
Process blossoming

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



Process blossoming…with orphans?

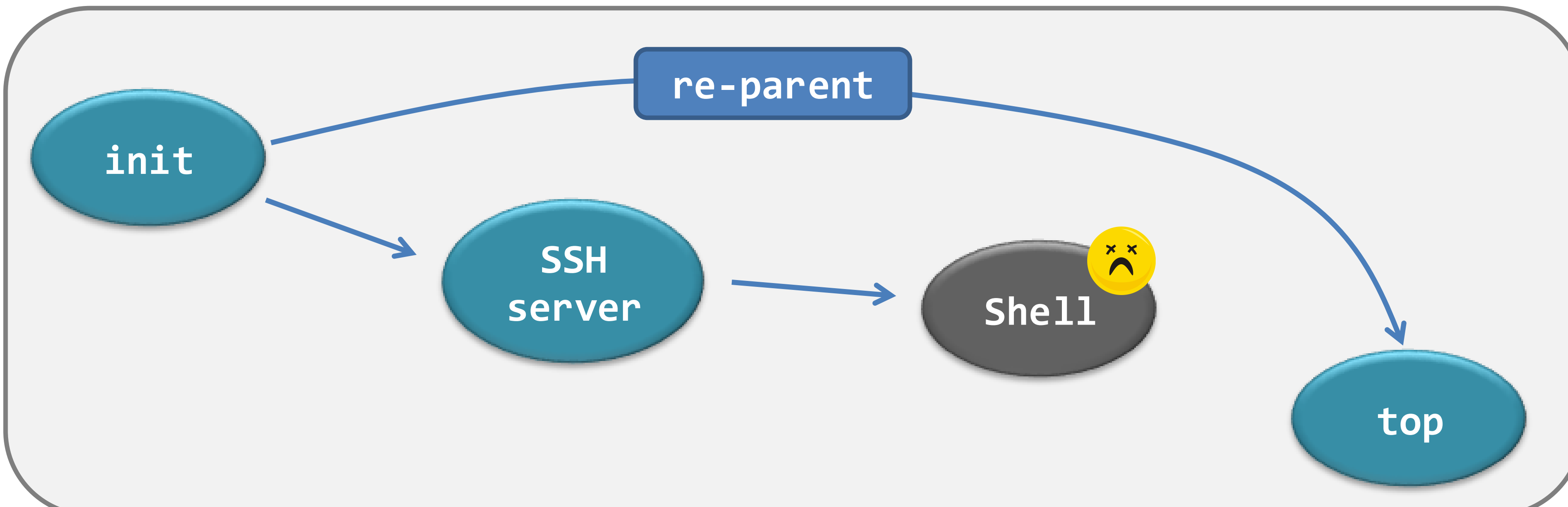
- However, termination can happen, at any time and in any place…
  - This is no good because an orphan turns the hierarchy from a **tree** into a **forest**!
  - Plus, no one would know the termination of the orphan.



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

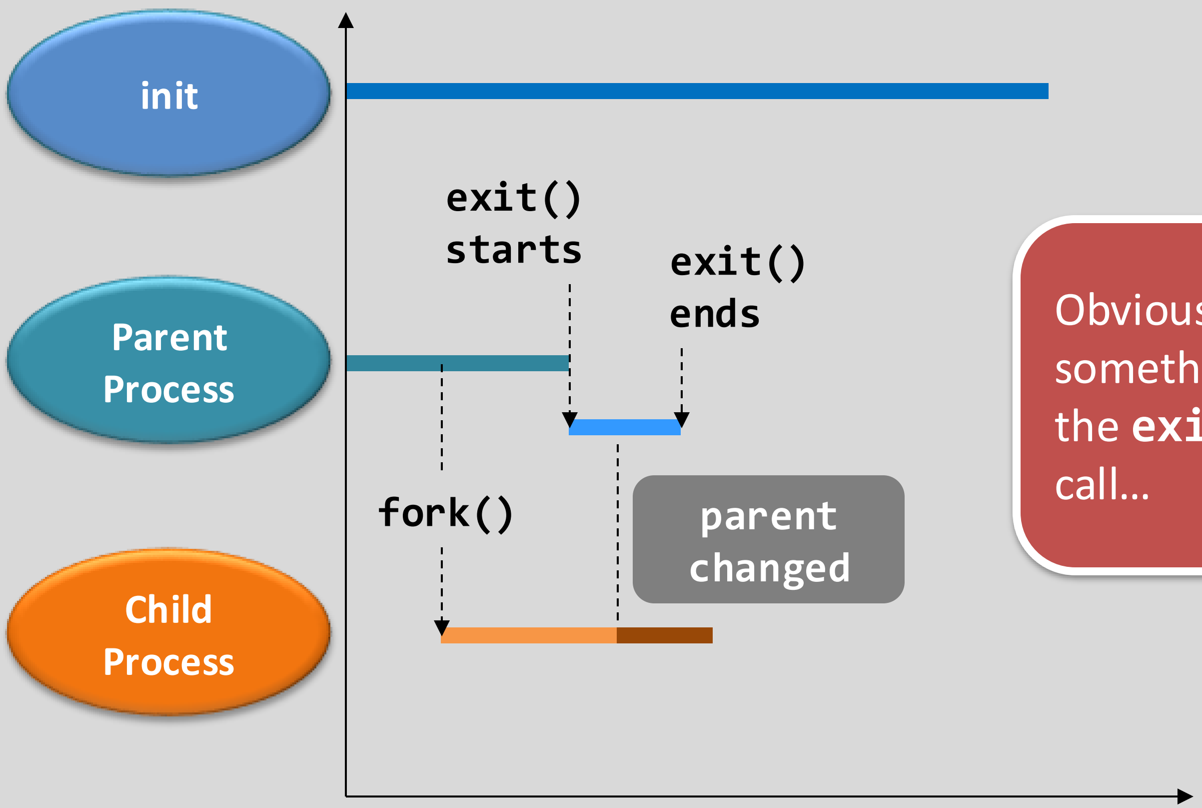
```c
int main(void) {
    int i;
    if(fork() == 0) {
        for(i = 0; i < 5; i++) {
            printf("(%d) parent's PID = %d\n",
                    getpid(), getppid() );
            sleep(1);
        }
    }
    else
        sleep(1);
    printf("(%d) bye.\n", getpid());
}
```

```
$ ./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.

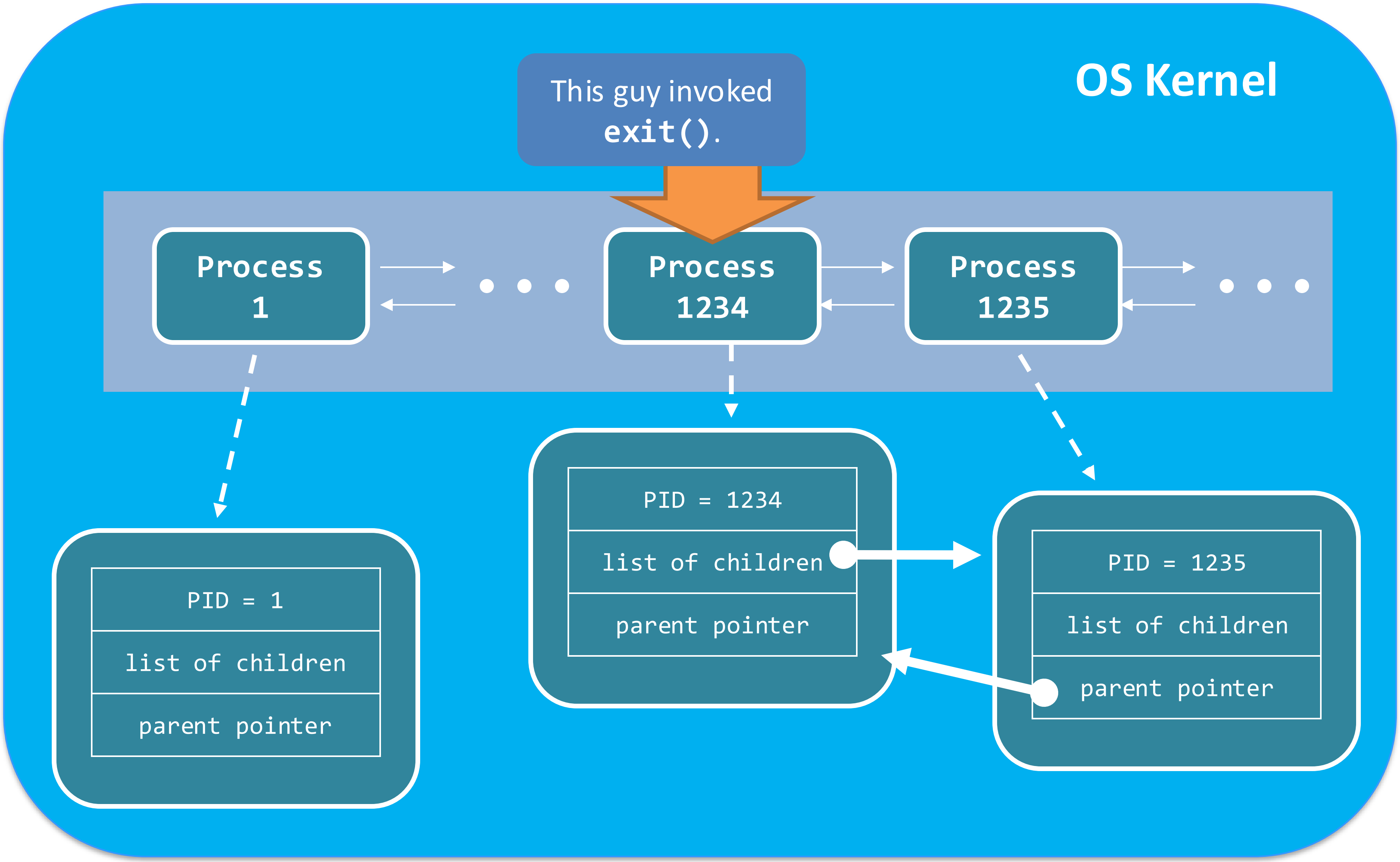
`[examples@3150] cat reparent.c`

What had happened during re-parenting?

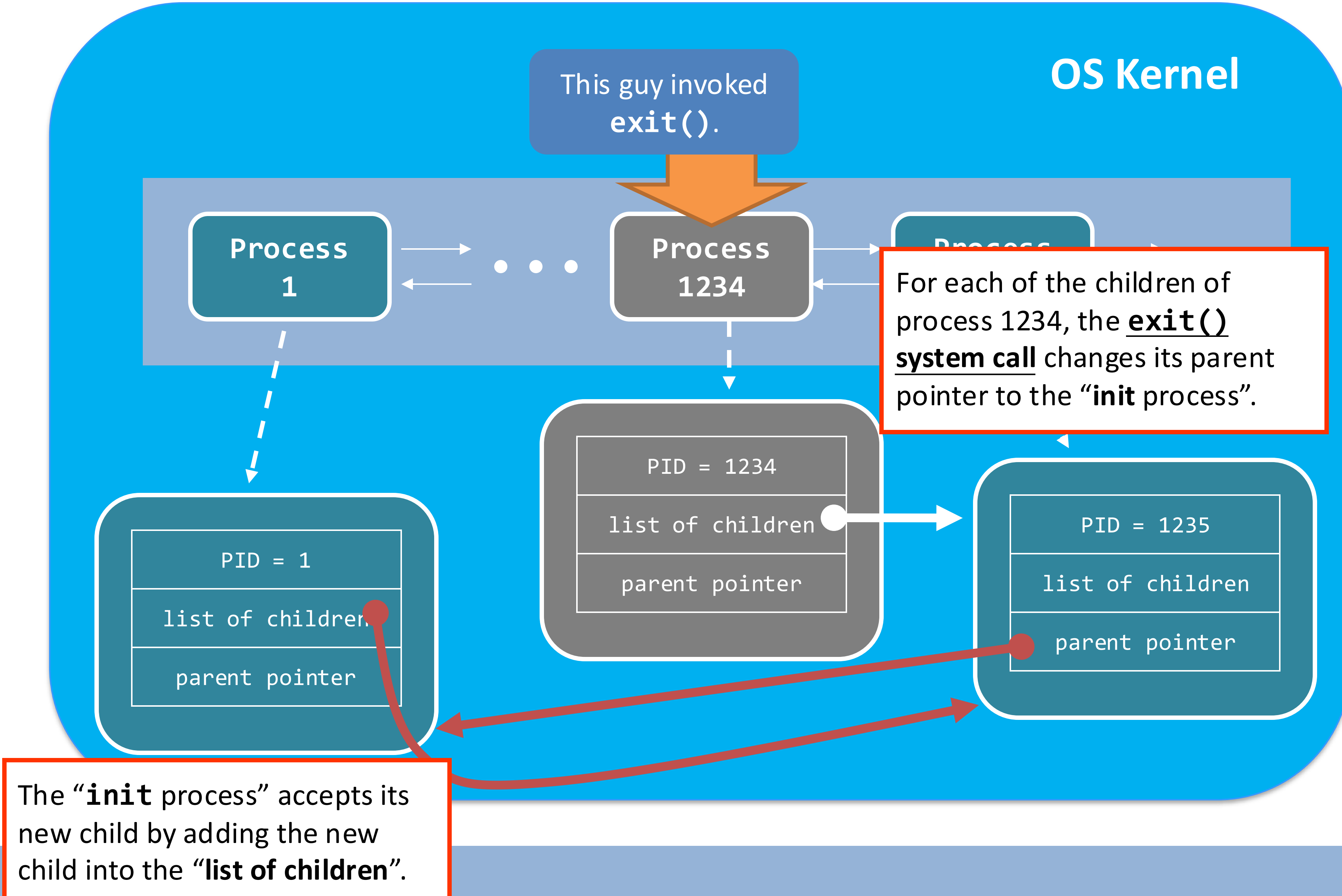


Obviously, there is something to do inside the **`exit()`** system call…

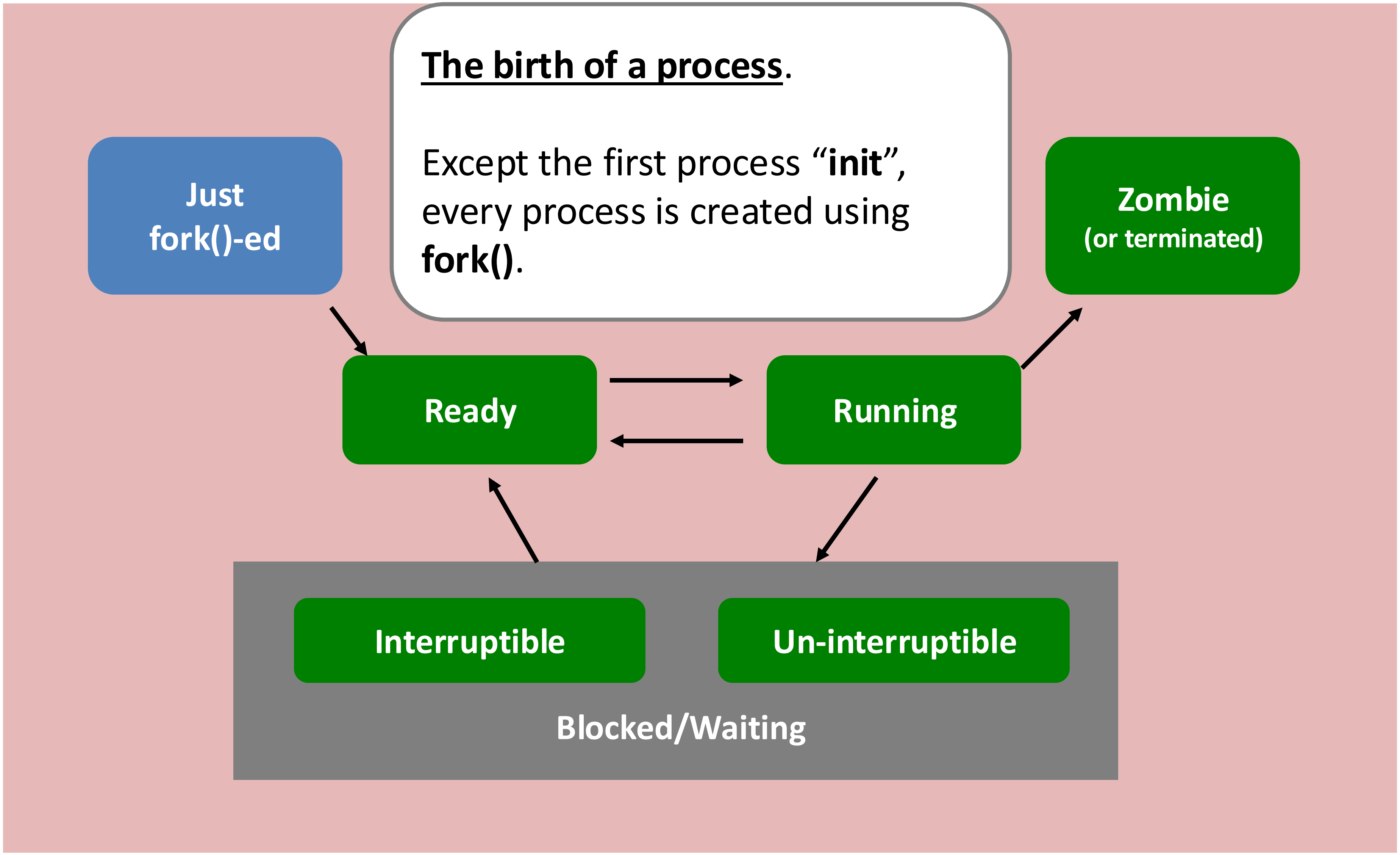
What had happened during re-parenting?



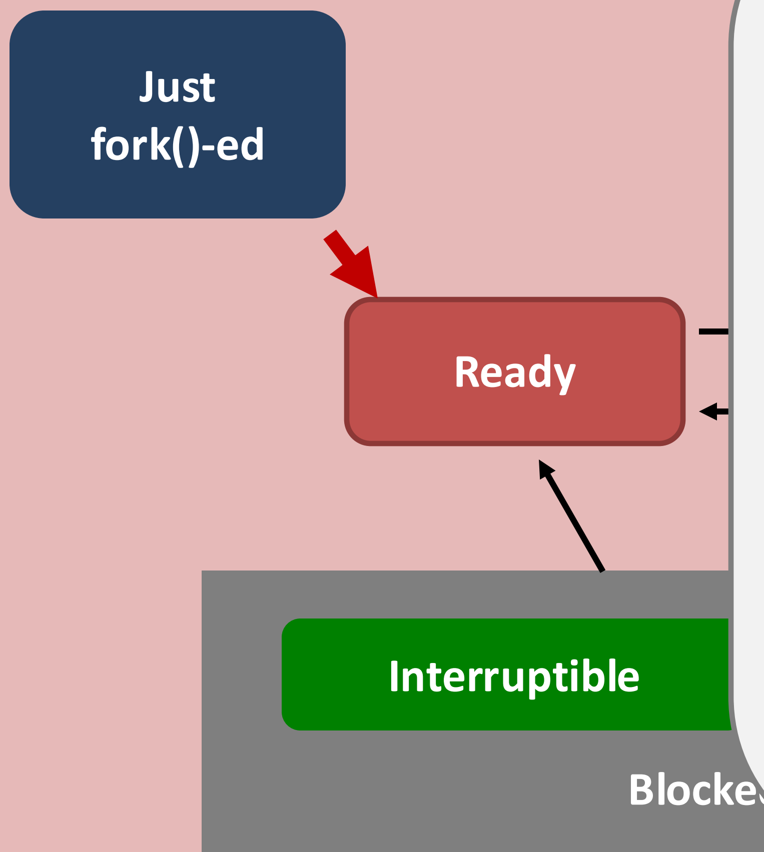
What had happened during re-parenting?



Process lifecycle – Kernel View



Process lifecycle - Ready

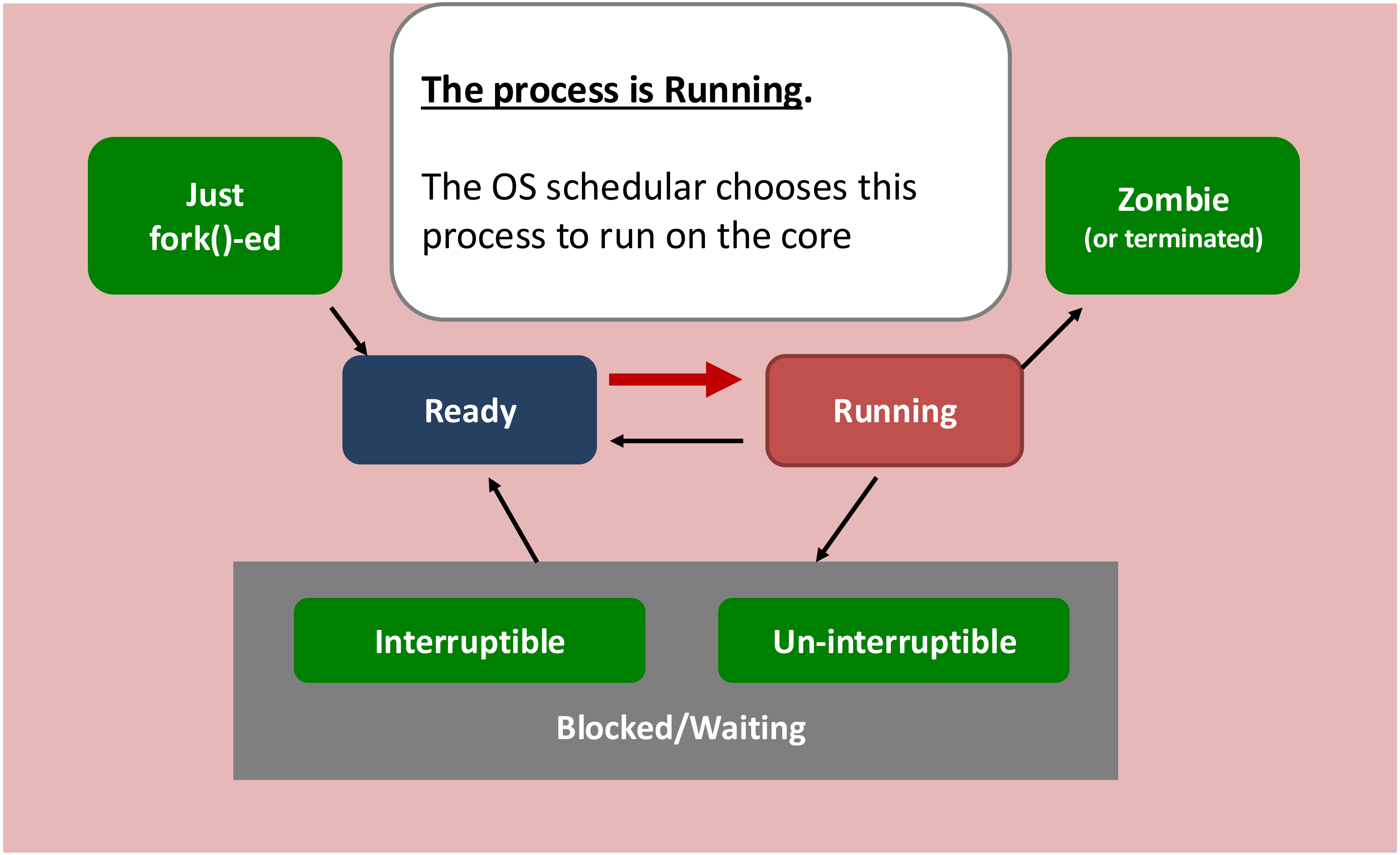


<u>The process is ready</u>.
It means it is ***ready to run* but is not running**.

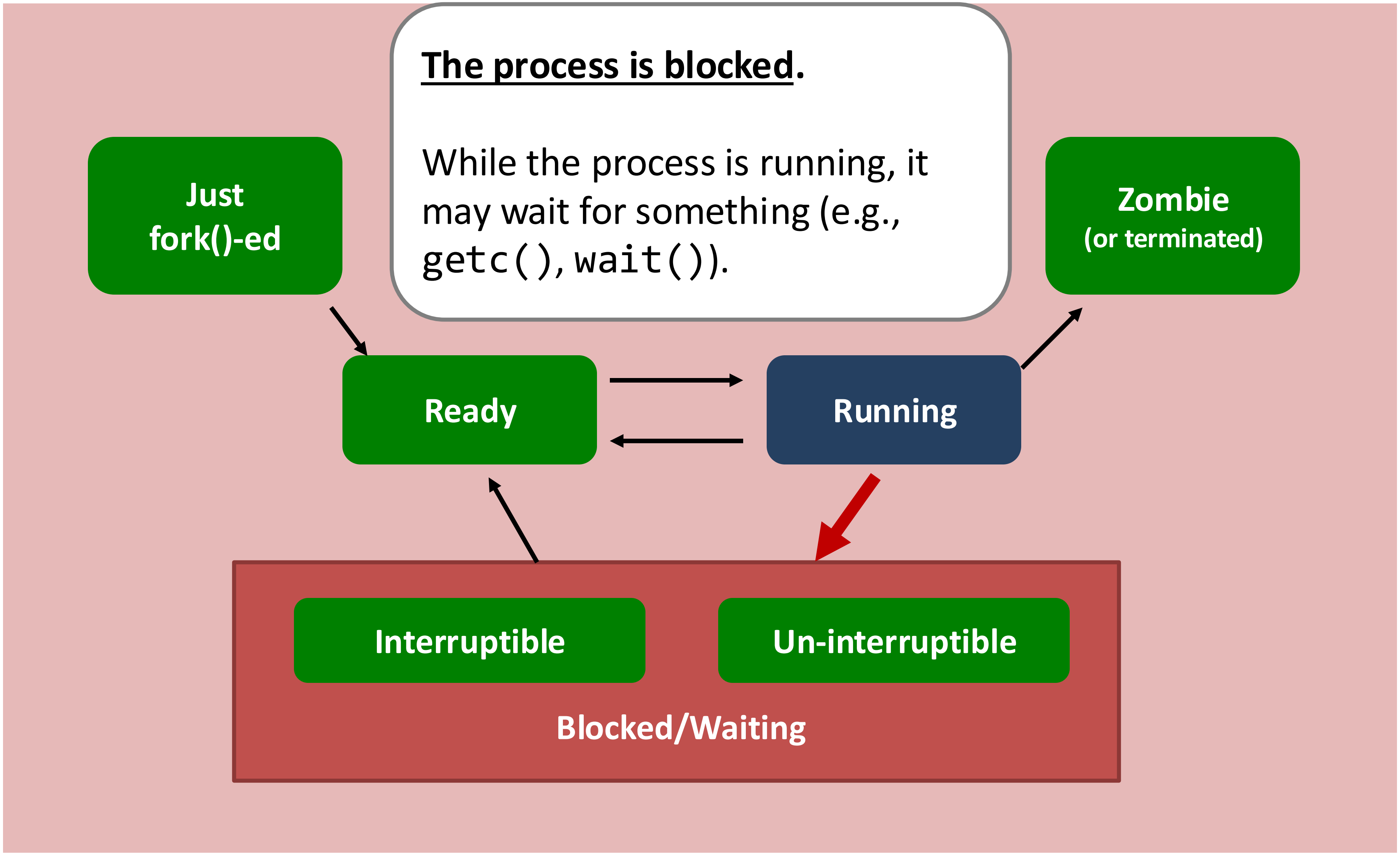
A process may become "**<u>ready</u>**" (*runnable*) after…

- 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

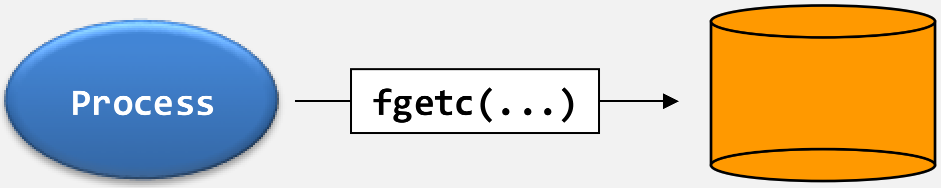


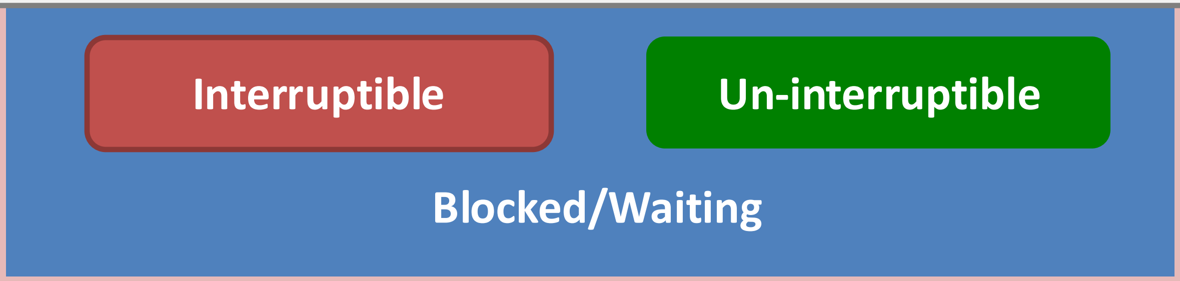
Process lifecycle – Interruptible wait

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., Some syscall call (http://man7.org/linux/man-pages/man2/delete_module.2.html)

Why set this?

- Interruptible means you need to checkpoint and recovery…
- Easier programming for lazy programmers (e.g., a driver program for a printer)
- 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

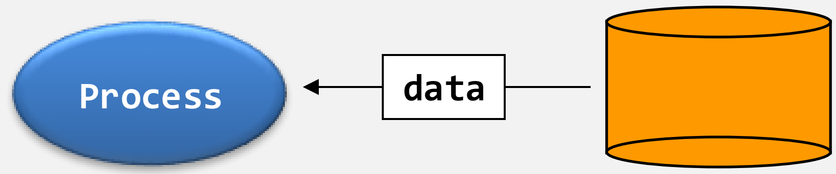
http://unix.stackexchange.com/questions/96797/what-does-the-interruptible-sleep-state-indicate

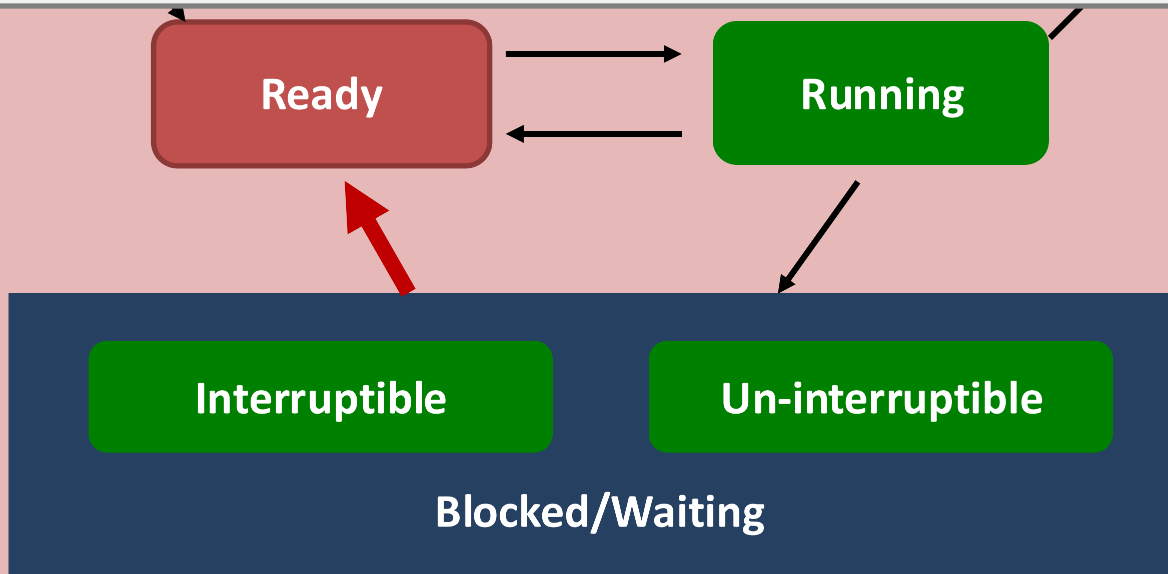
http://stackoverflow.com/questions/767551/how-to-stop-uninterruptible-process-on-linux

Process lifecycle

<u>Return back to ready.</u>

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

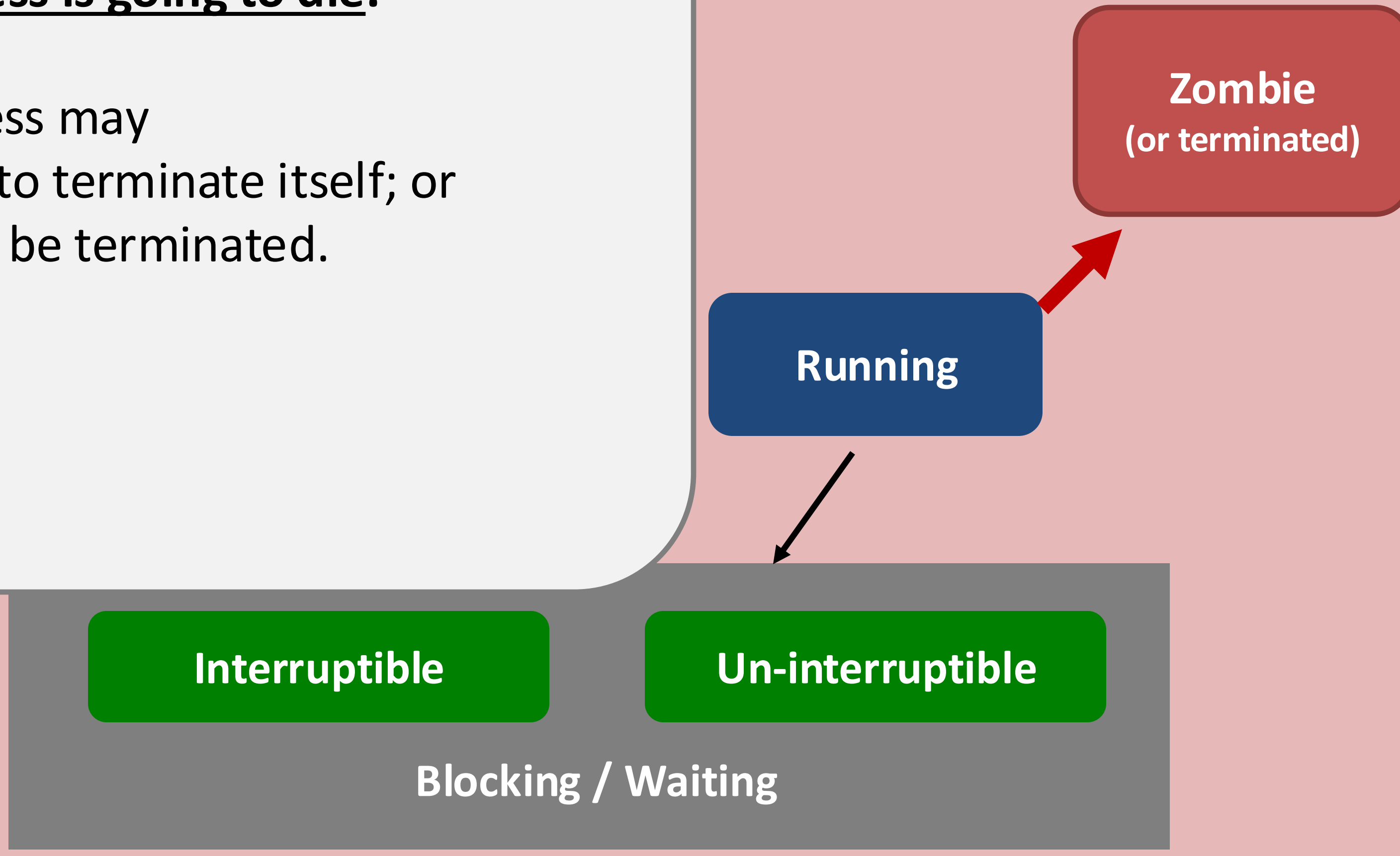




Process lifecycle

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

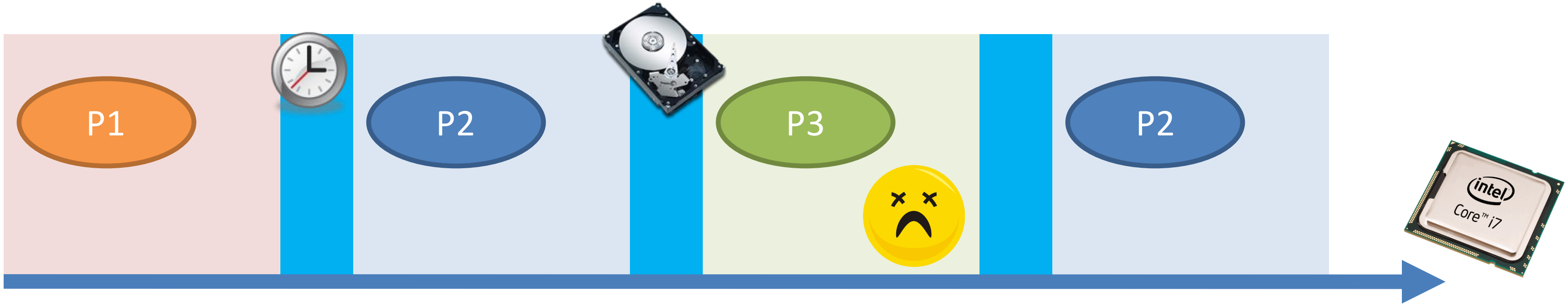
The process may
- choose to terminate itself; or
- force to be terminated.

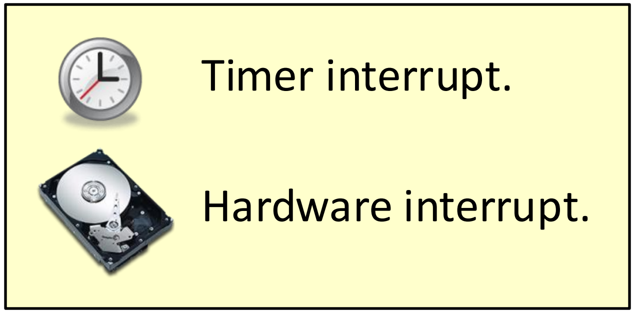


What is context switching?

Scheduling is the procedure that decides which process to run next.

Context switching is the actual switching procedure, from one process to another.

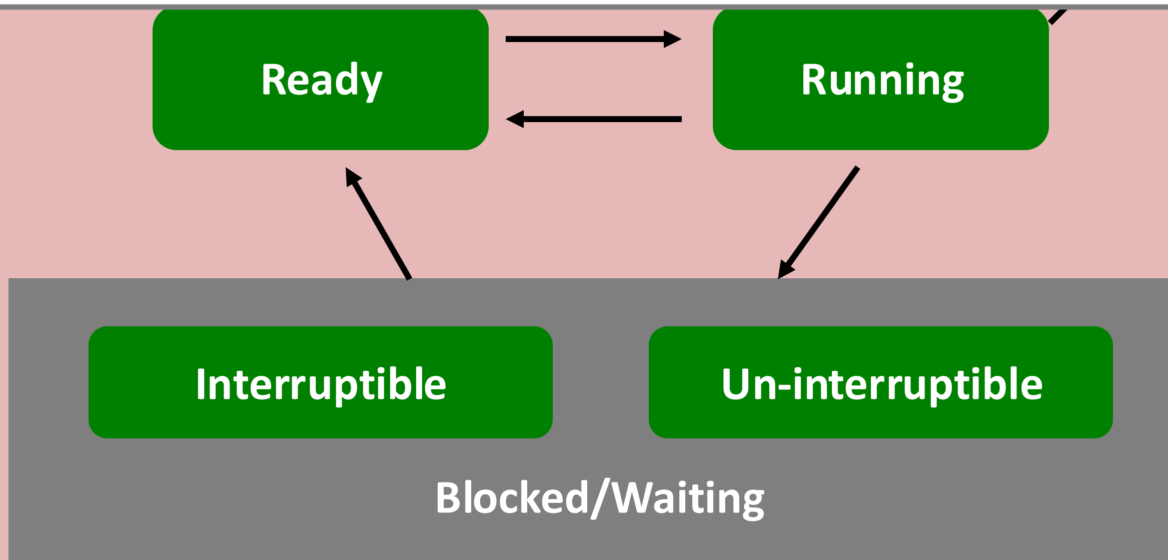




When context switch happens?

Why?
- For multi-tasking
- For fully utilize the CPU

- Whenever a process goes to blocking / waiting state; e.g., `wait()`/`sleep()` is called
- A POSIX signal arrives (e.g., SIGCHLD)
- An interrupt arrives (e.g., keystroke)
- When the OS scheduler says "time's up!" (e.g., round-robin)
  - Put it back to "ready"
- When the OS scheduler says "hey, I know you haven't finished, but the PRESIDENT just arrives, please hold on" (e.g., preemptive, round-robin with priority)
  - Put it back to "ready"
- ...

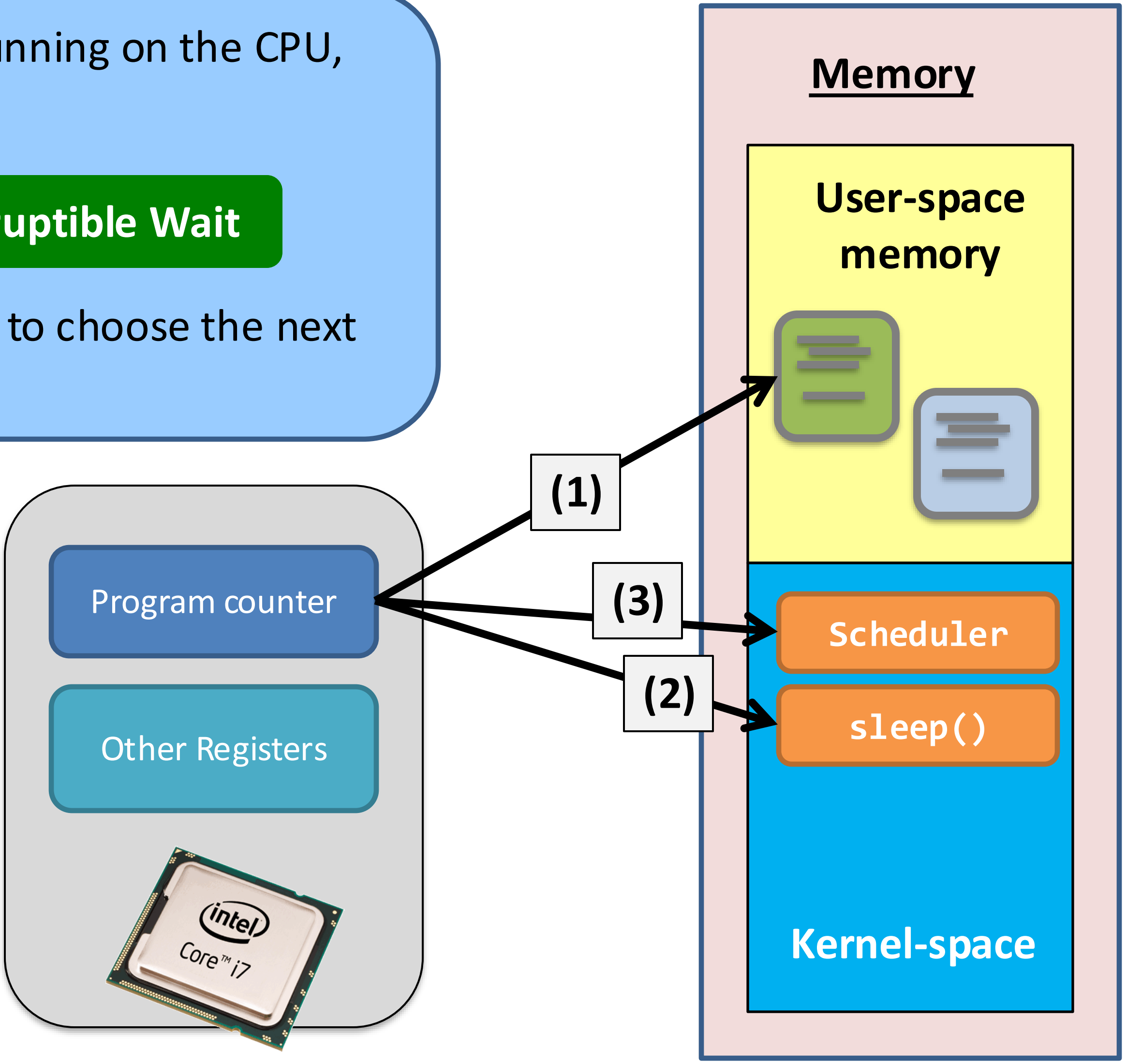


Context switching

Suppose this process gives up running on the CPU, e.g., calling **`sleep()`**. Then:

Running → **Interruptible Wait**

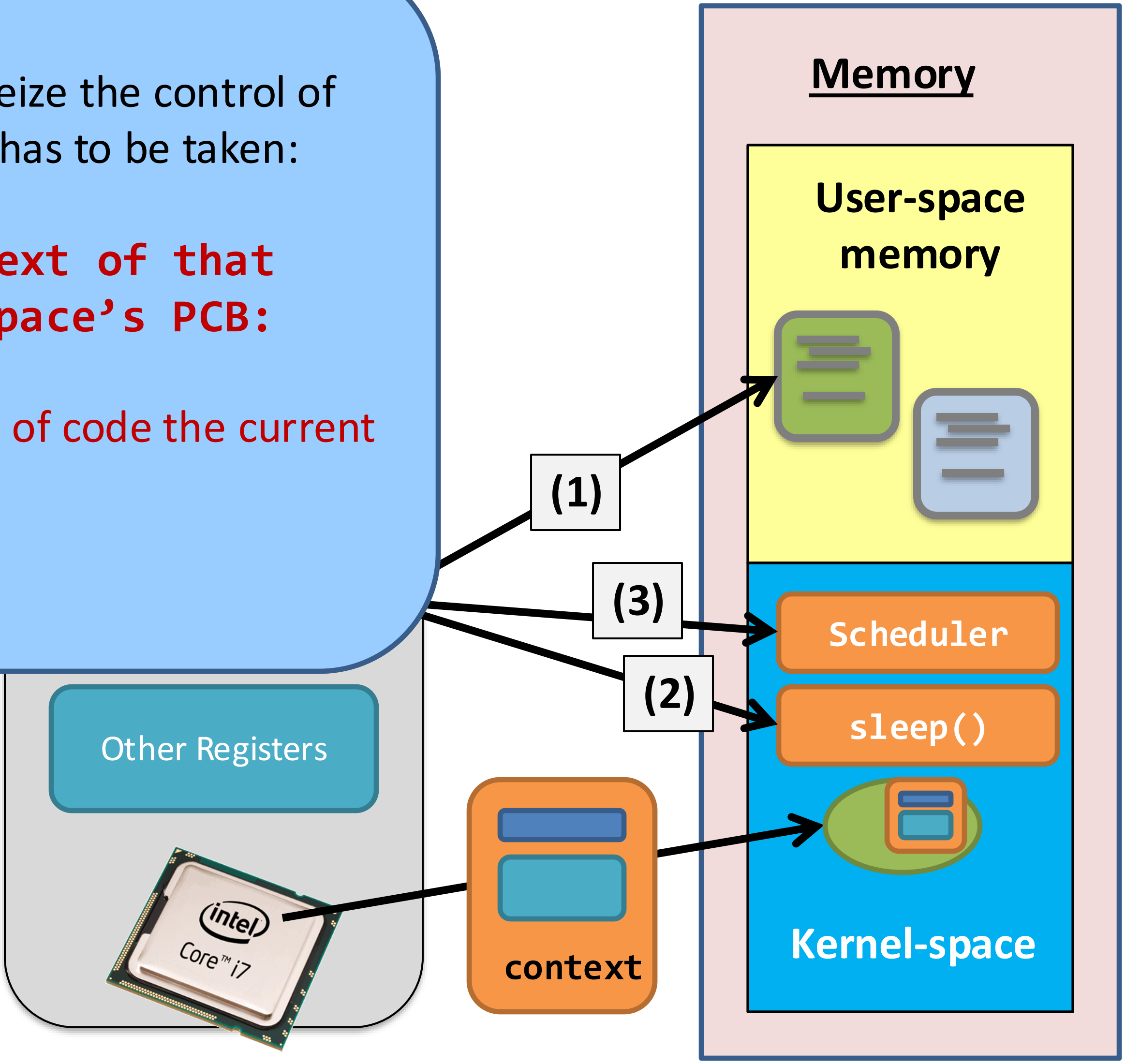
Now, it is time for the scheduler to choose the next process to run.

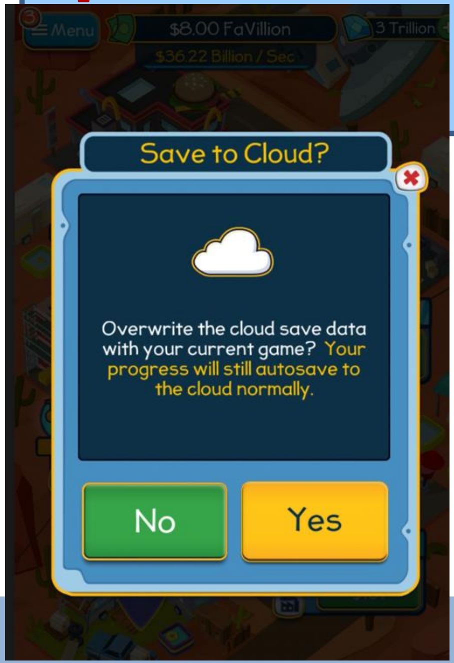


Context switching

But, before the scheduler can seize the control of the CPU, a very important step has to be taken:

Backup all current context of that process to the kernel-space's PCB:
- current register values
- program counter (which line of code the current program is at)
-

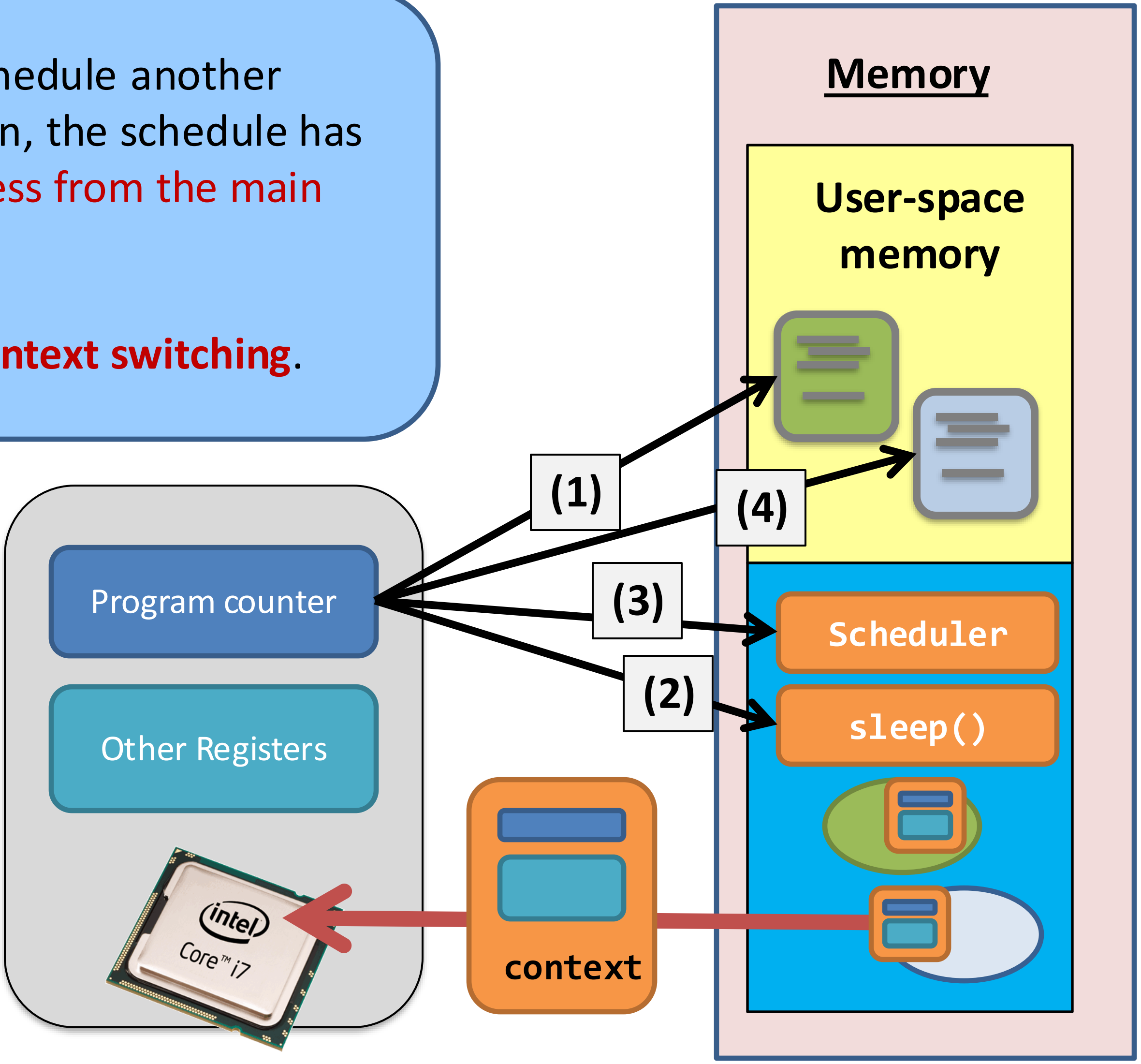




Context switching

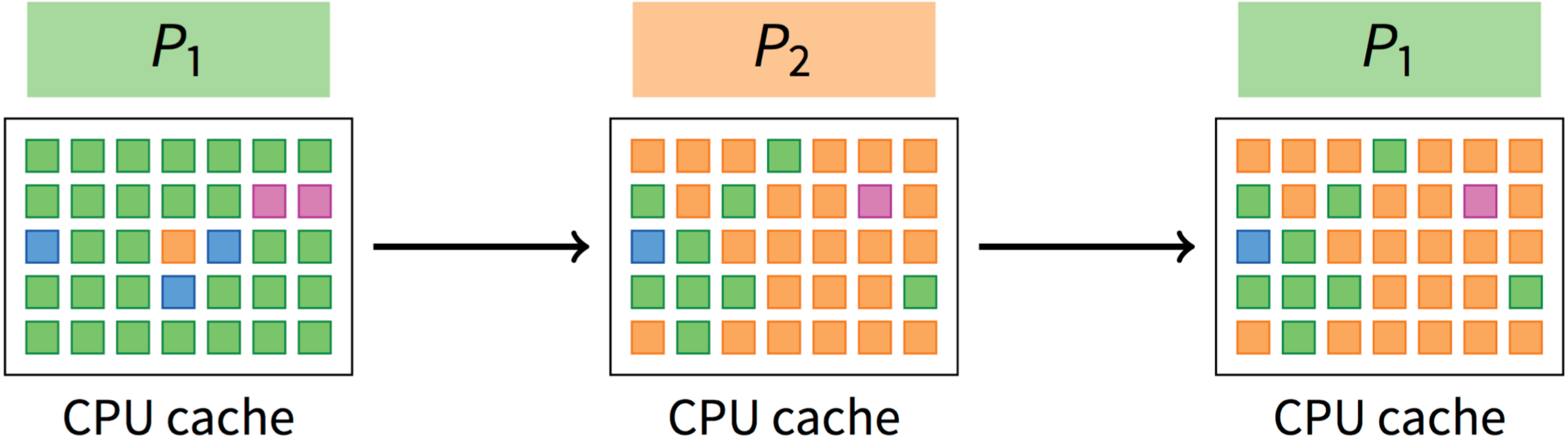
Say, the scheduler decides to schedule another process in the ready queue. Then, the schedule has to load the context of that process from the main memory to the CPU.

We call the entire operation: context switching.



Context switch is expensive

- Direct costs in kernel:
  - Save and restore registers, etc.
  - Switch process address space (will see when learning about memory management)
- Indirect costs: cache misses, etc.



User time VS Sys time

- `time`: tool to report about the time of your program
  - Real time: wall clock time
  - User time: CPU time on user-space (running state)
    - CPU time excludes sleep time (e.g., waits for I/O)
  - Sys time: CPU time on kernel-space (running state)
- Sys call is expensive
  - Function calls cause overhead on Stack
  - Cause context switch from user-code to kernel-code

Real time vs User-time + Sys time

- It's possible that
  - Real time > User-time + Sys time

When?

  - Real time < User-time + Sys time

When?