Lecture 4
Processes II

Prof. Yinqian Zhang

Spring 2022

Outline

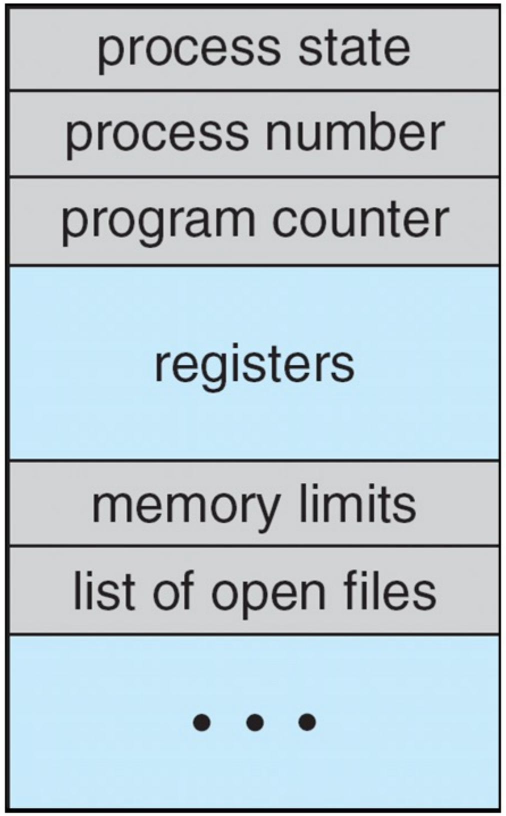
- Kernel view of processes
- Kernel view of fork(), exec(), and wait()
- More about processes

Processes: Kernel View

Process Control Block (PCB)

Information associated with each process

- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information



PCB Example: uCore

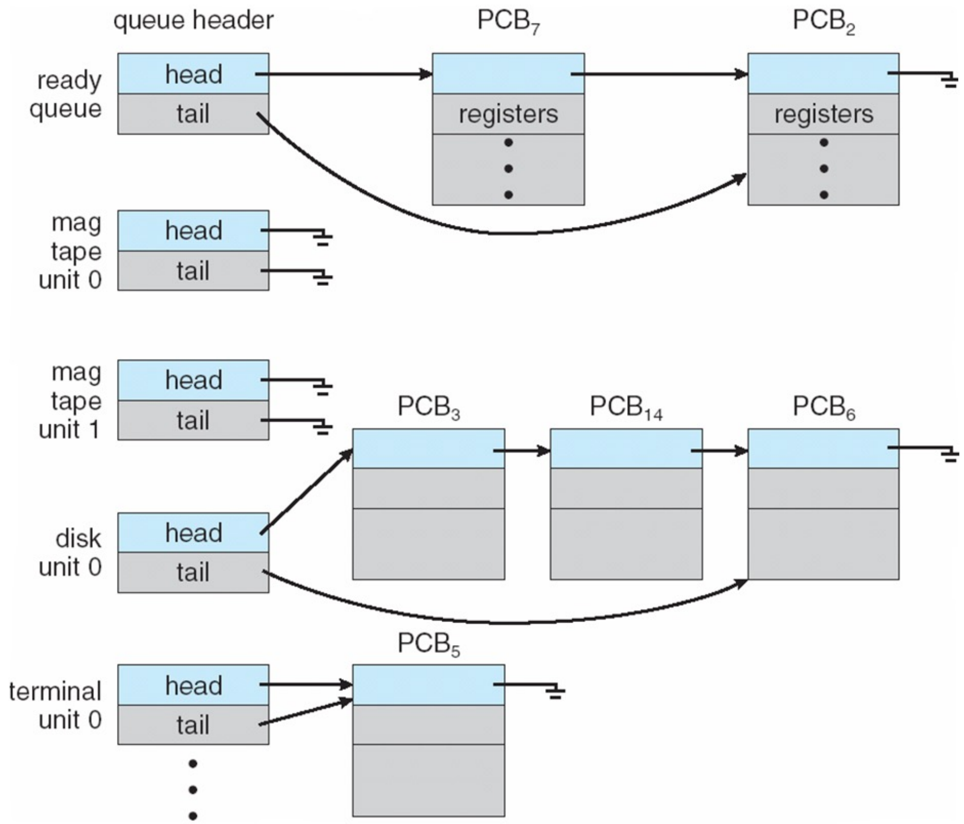
```
/* kern/process/proc.h in ucore */

struct proc_struct {
    enum proc_state state;                  // Process state
    int pid;                                // Process ID
    int runs;                               // the running times of Process
    uintptr_t kstack;                       // Process kernel stack
    volatile bool need_resched;             // bool value: need to be rescheduled to release CPU?
    struct proc_struct *parent;             // the parent process
    struct mm_struct *mm;                   // Process's memory management field
    struct context context;                 // Switch here to run process
    struct trapframe *tf;                   // Trap frame for current interrupt
    uintptr_t cr3;                          // CR3 register: the base addr of Page Directroy Table(PDT)
    uint32_t flags;                         // Process flag
    char name[PROC_NAME_LEN + 1];           // Process name
    list_entry_t list_link;                 // Process link list
```

PCB Example: uCore

```
/* kern/process/proc.h in ucore */

    list_entry_t hash_link;                 // Process hash list
    int exit_code;                          // exit code (be sent to parent proc)
    uint32_t wait_state;                    // waiting state
    struct proc_struct *cptr, *yptr, *optr; // relations between processes
    struct run_queue *rq;                   // running queue contains Process
    list_entry_t run_link;                  // the entry linked in run queue
    int time_slice;                         // time slice for occupying the CPU
    struct files_struct *filesp;            // the file related info of process
};
```

Ready Queue And I/O Device Queues

- PCBs are linked in multiple queues
  - Ready queue contains all processes in the ready state (to run on this CPU)
  - Device queue contains processes waiting for I/O events from this device
  - Process may migrate among these queues

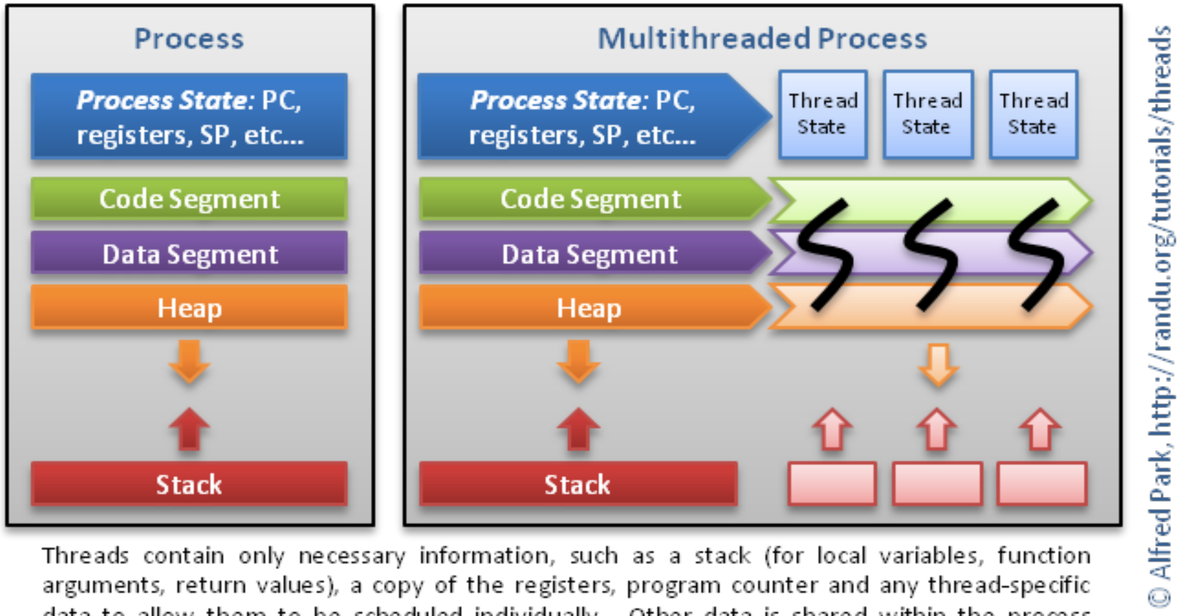


Threads

- One process may have more than one threads
  - A single-threaded process performs a single thread of execution
  - A multi-threaded process performs multiple threads of execution "concurrently", thus allowing short response time to user's input even when the main thread is busy
- PCB is extended to include information about each thread

Process and Thread

- Single threaded process and multi-threaded process



Threads contain only necessary information, such as a stack (for local variables, function arguments, return values), a copy of the registers, program counter and any thread-specific data to allow them to be scheduled individually. Other data is shared within the process between all threads.

Switching Between Processes

- Once a process runs on a CPU, it only gives back the control of a CPU
  - when it makes a system call
  - when it raises an exception
  - when an interrupt occurs
- What if none of these would happen for a long time?
  - Coorperative scheduling: OS will have to wait
    - Early Macintosh OS, old Alto system
  - Non-coorperative scheduling: timer interrupts
    - Modern operating systems

Switching Between Processes (Cont'd)

- When OS kernel regains the control of CPU
  - It first completes the task
    - Serve system call, or
    - Handle interrupt/exception
  - It then decides which process to run next
    - by asking its **CPU scheduler**
    - How does it make decisions?
    - More about CPU scheduler later
  - It performs a **context switch** if the soon-to-be-executing process is different from the previous one

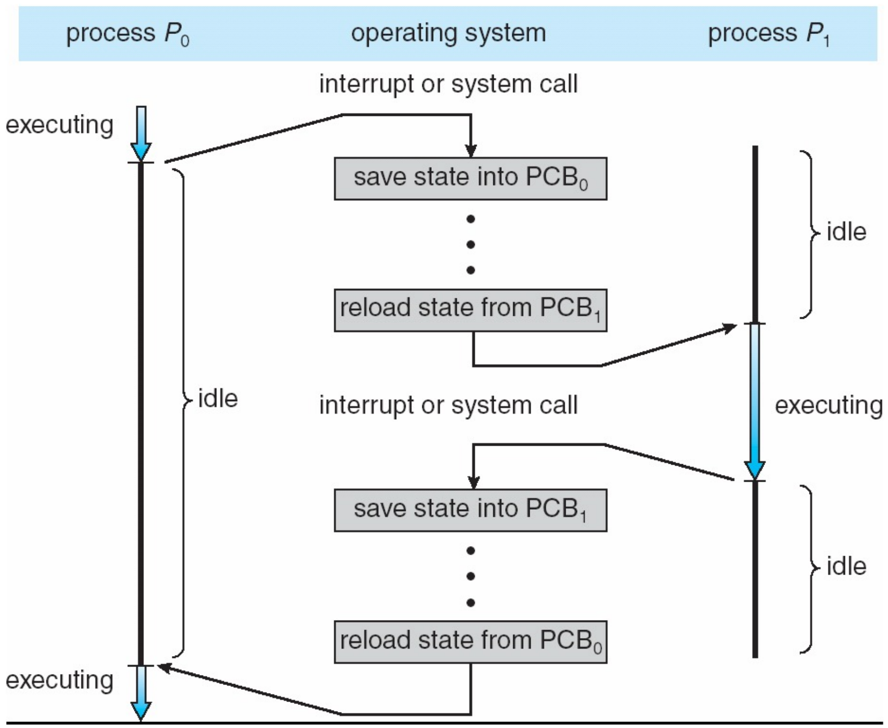
Context Switch

- During context switch, the system must save the state of the old process and load the saved state for the new process

- Context of a process is represented in the PCB

- The time used to do context switch is an overhead of the system; the system does no useful work while switching
  - Time of context switch depends on hardware support
  - Context switch cannot be too frequent

Context Switch (Cont'd)



Context Switch: uCore

```
/* kern/schedule/sched.c */
void schedule(void) {
   bool intr_flag;
   struct proc_struct *next;
   local_intr_save(intr_flag);
   {
      if (current->state == PROC_RUNNABLE)
         sched_class_enqueue(current);

      if ((next = sched_class_pick_next()) != NULL)
         sched_class_dequeue(next);

      if (next != current)
         proc_run(next);
   }
   local_intr_restore(intr_flag);
}
```

```
/* kern/process/proc.c*/



void proc_run(struct proc_struct *proc) {
   if (proc != current) {
      bool intr_flag;
      struct proc_struct *prev = current, *next = proc;
      local_intr_save(intr_flag);
      {
         current = proc;
         lcr3(next->cr3);
         switch_to(&(prev->context), &(next->context));
      }
      local_intr_restore(intr_flag);
   }
}
```

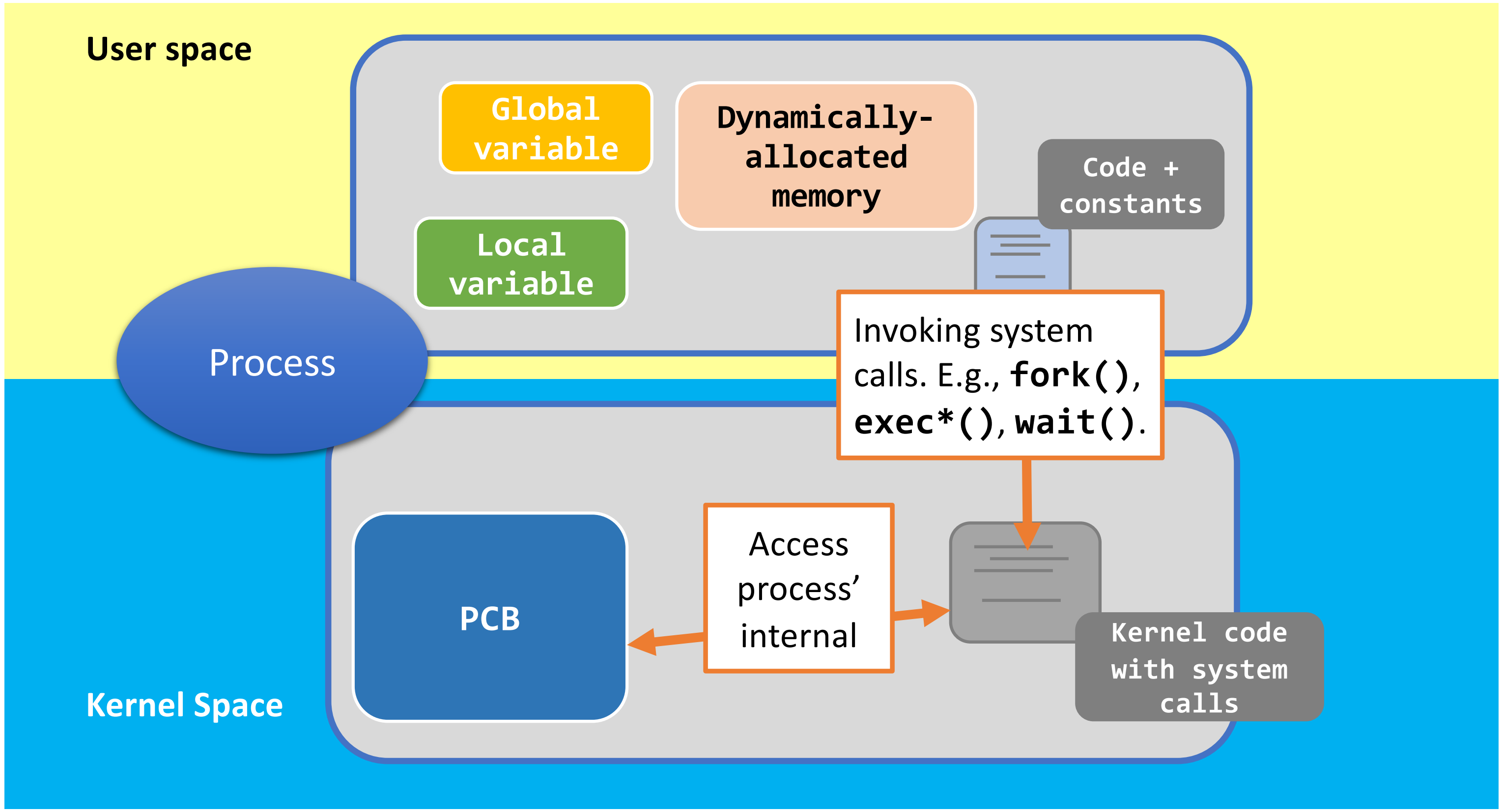
Context Switch: uCore (Cont'd)

```
/* kern/process/switch.S */
.globl switch_to
switch_to:
   # save from's registers
   STORE ra, 0*REGBYTES(a0)
   STORE sp, 1*REGBYTES(a0)
   STORE s0, 2*REGBYTES(a0)
   STORE s1, 3*REGBYTES(a0)
   STORE s2, 4*REGBYTES(a0)
   STORE s3, 5*REGBYTES(a0)
   STORE s4, 6*REGBYTES(a0)
   STORE s5, 7*REGBYTES(a0)
   STORE s6, 8*REGBYTES(a0)
   STORE s7, 9*REGBYTES(a0)
   STORE s8, 10*REGBYTES(a0)
   STORE s9, 11*REGBYTES(a0)
   STORE s10, 12*REGBYTES(a0)
   STORE s11, 13*REGBYTES(a0)
```

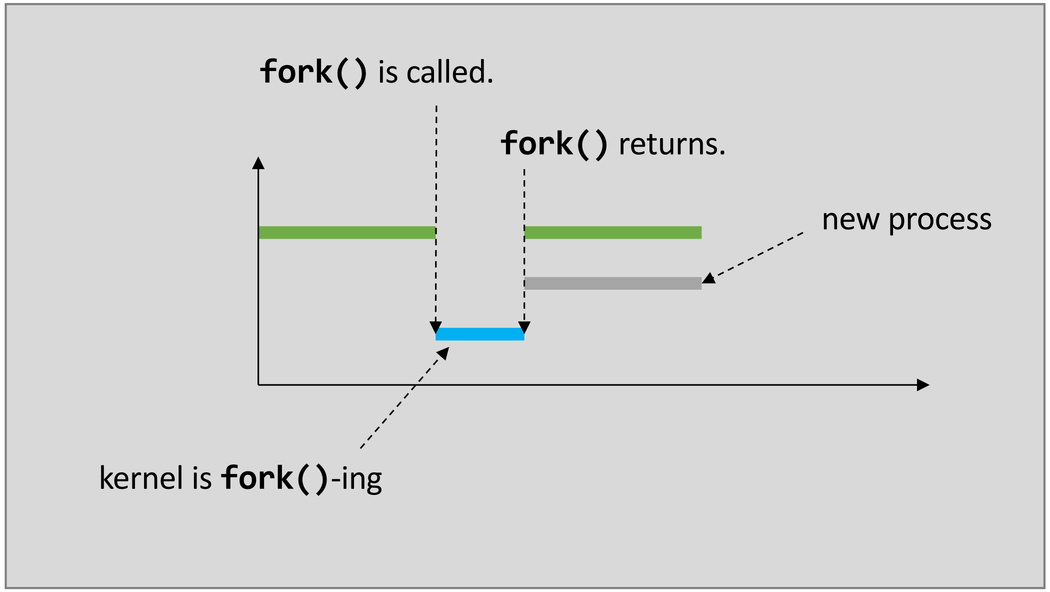
```
# restore to's registers
LOAD ra, 0*REGBYTES(a1)
LOAD sp, 1*REGBYTES(a1)
LOAD s0, 2*REGBYTES(a1)
LOAD s1, 3*REGBYTES(a1)
LOAD s2, 4*REGBYTES(a1)
LOAD s3, 5*REGBYTES(a1)
LOAD s4, 6*REGBYTES(a1)
LOAD s5, 7*REGBYTES(a1)
LOAD s6, 8*REGBYTES(a1)
LOAD s7, 9*REGBYTES(a1)
LOAD s8, 10*REGBYTES(a1)
LOAD s9, 11*REGBYTES(a1)
LOAD s10, 12*REGBYTES(a1)
LOAD s11, 13*REGBYTES(a1)

ret
```

fork(), exec(), wait()
Kernel View

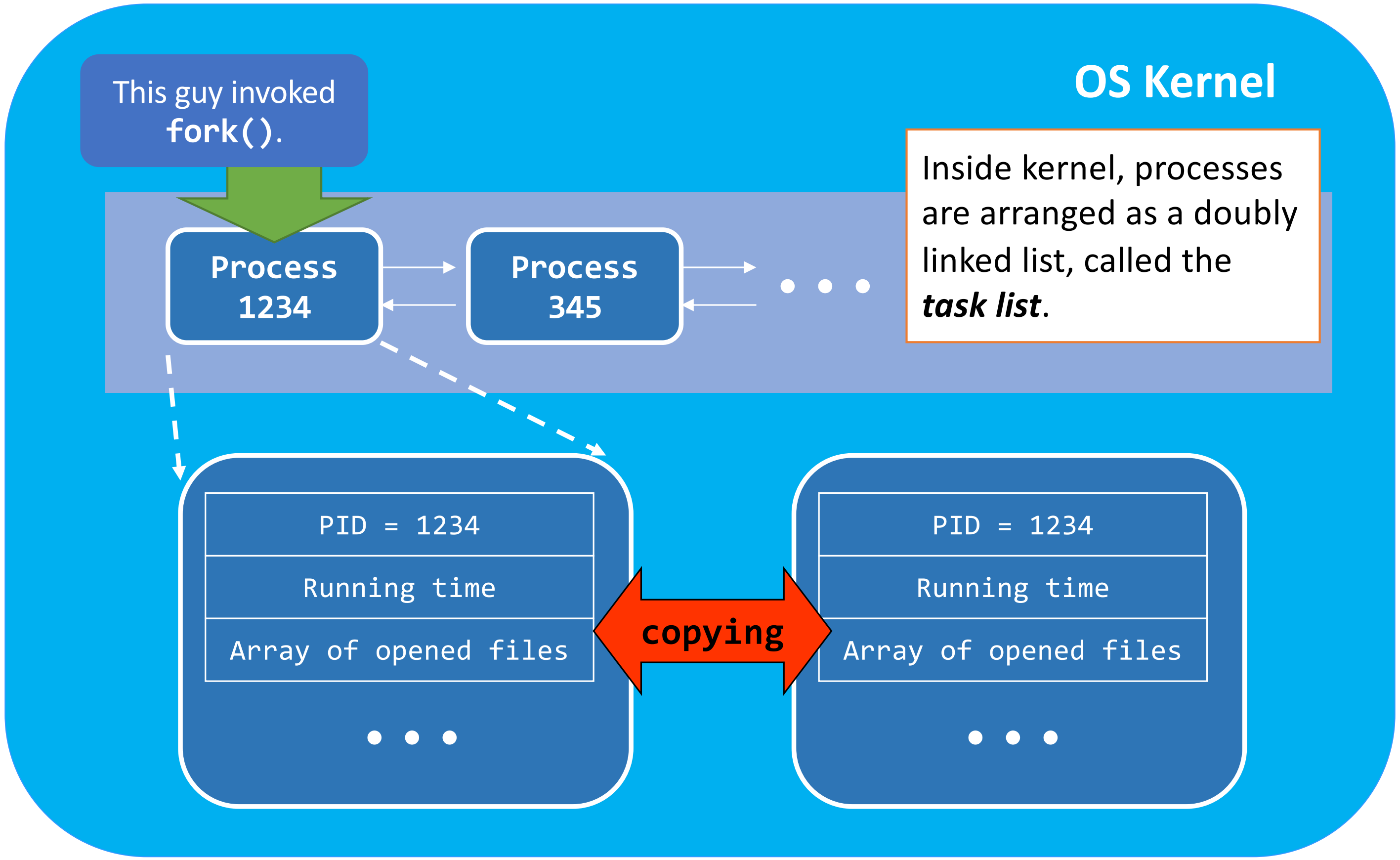
Recall: fork(), exec(), and wait()



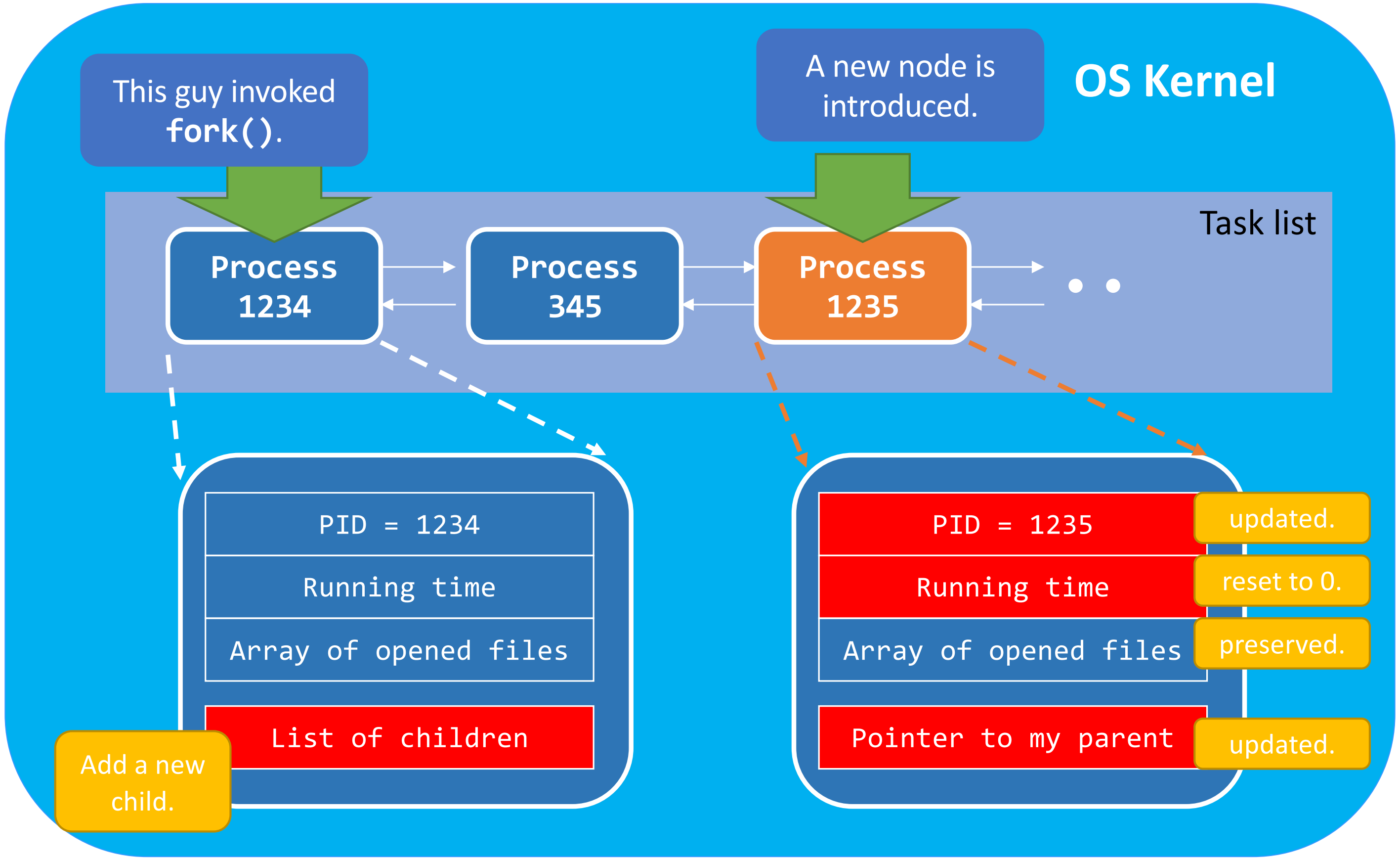
Fork() in User Mode



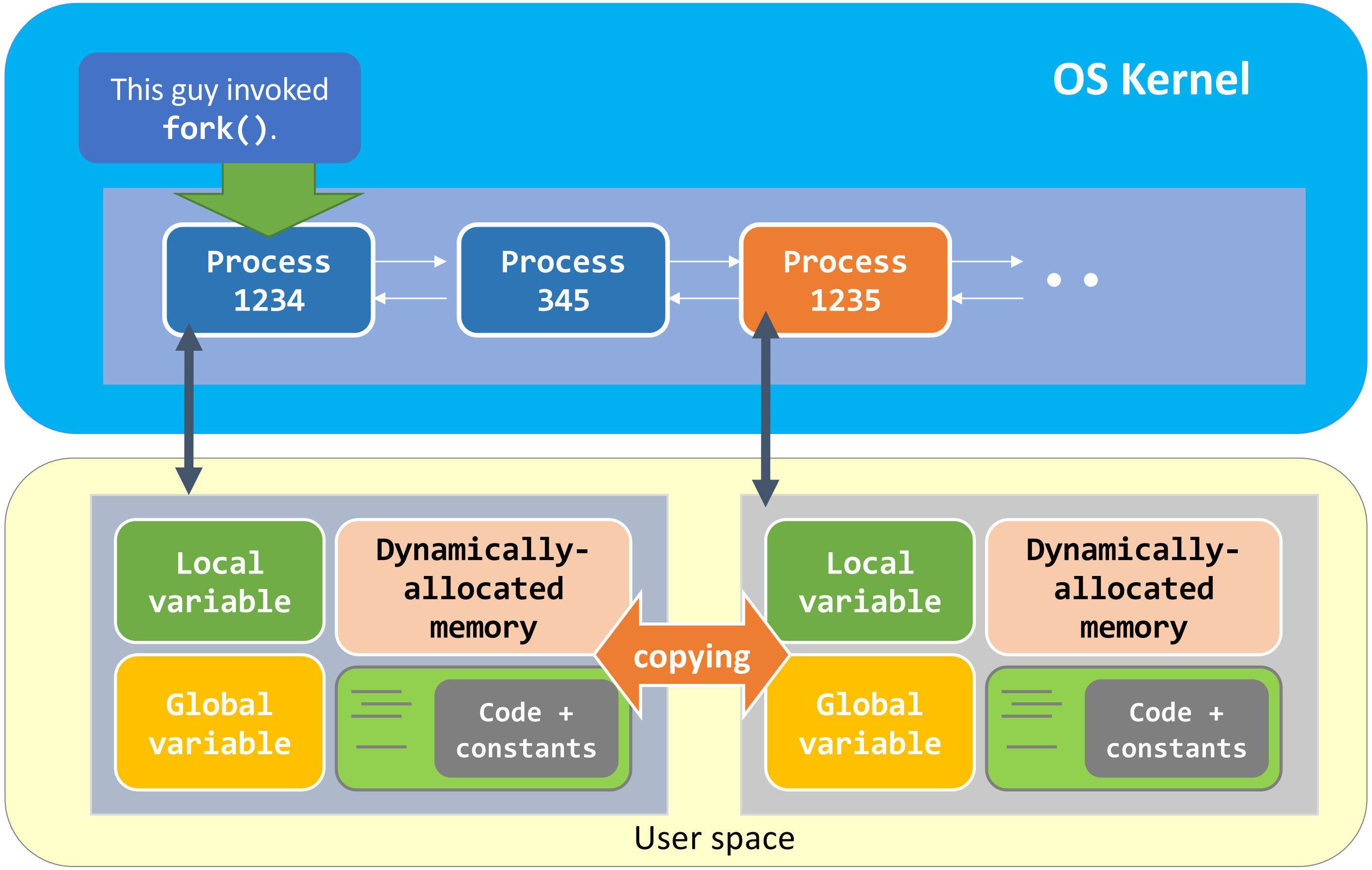
fork(): Kernel View



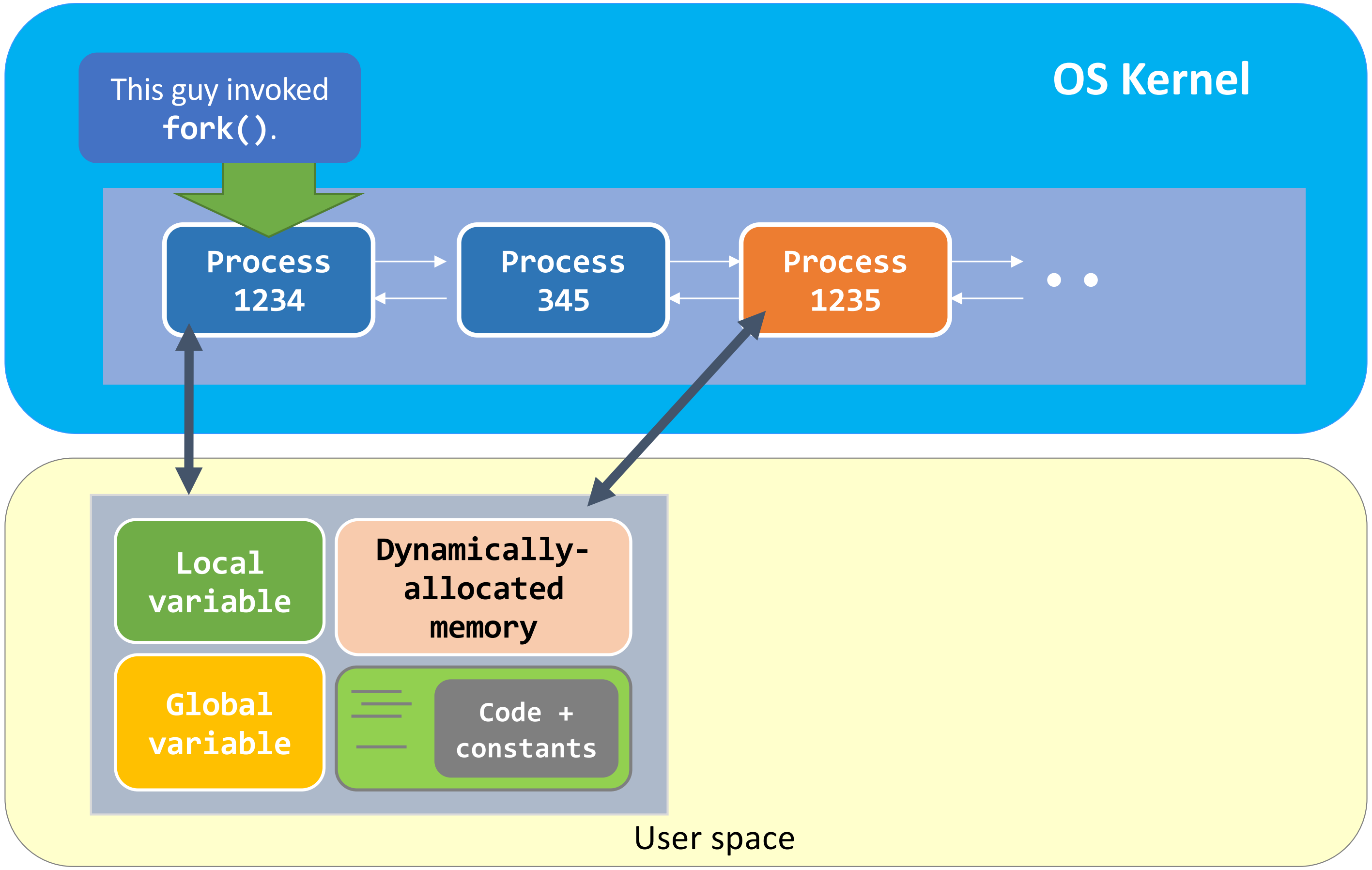
fork(): Kernel View



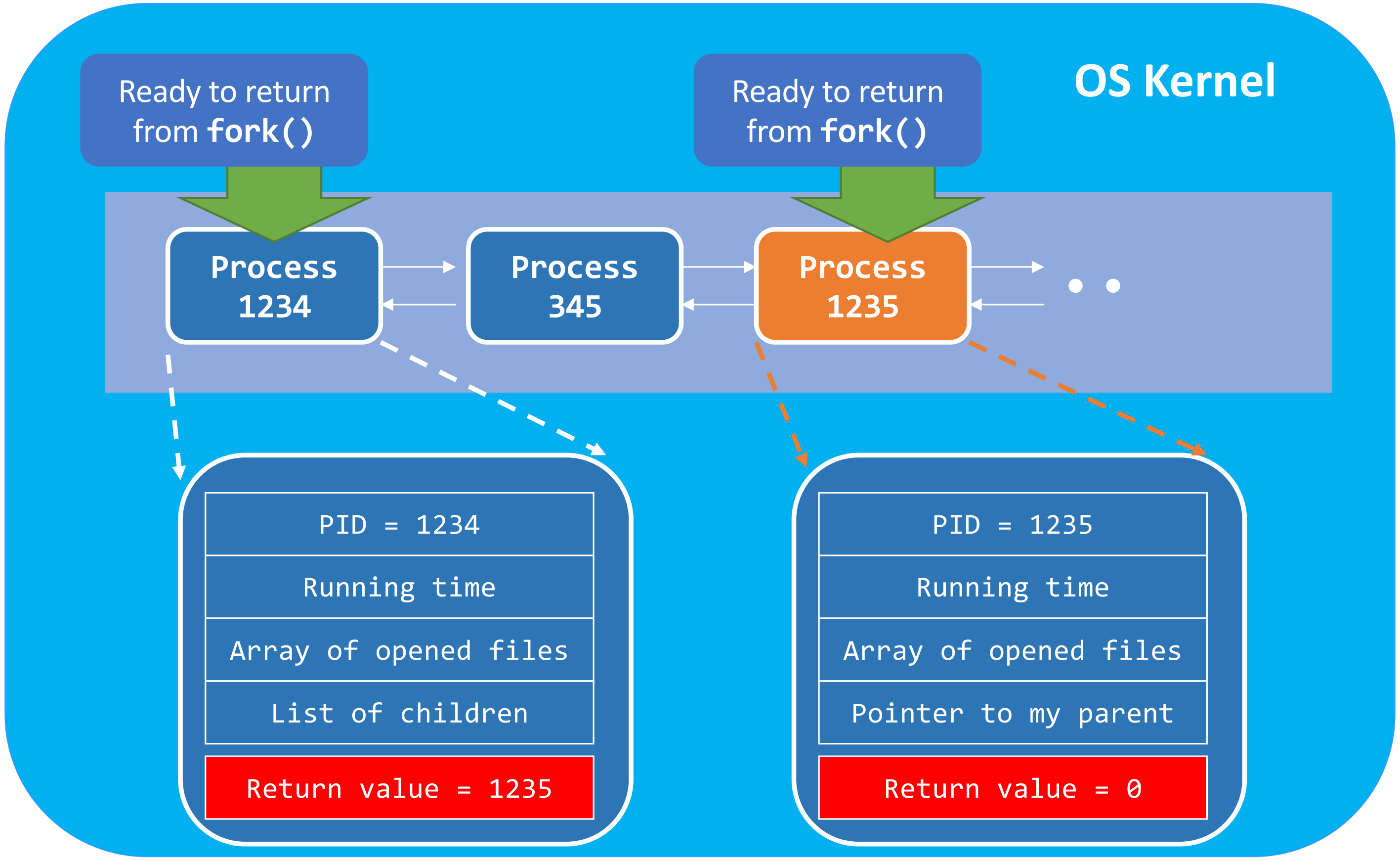
Case 1: Duplicate Address Space



Case 2: Copy on Write



fork(): Kernel View



fork(): Opened Files

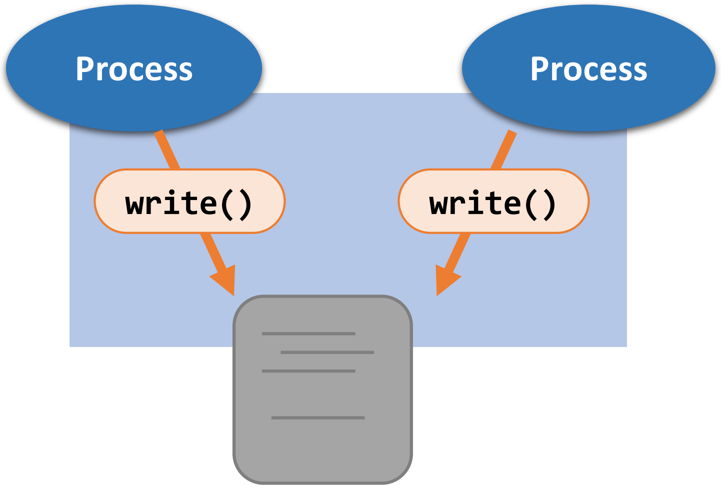
- Array of opened files contains:

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.

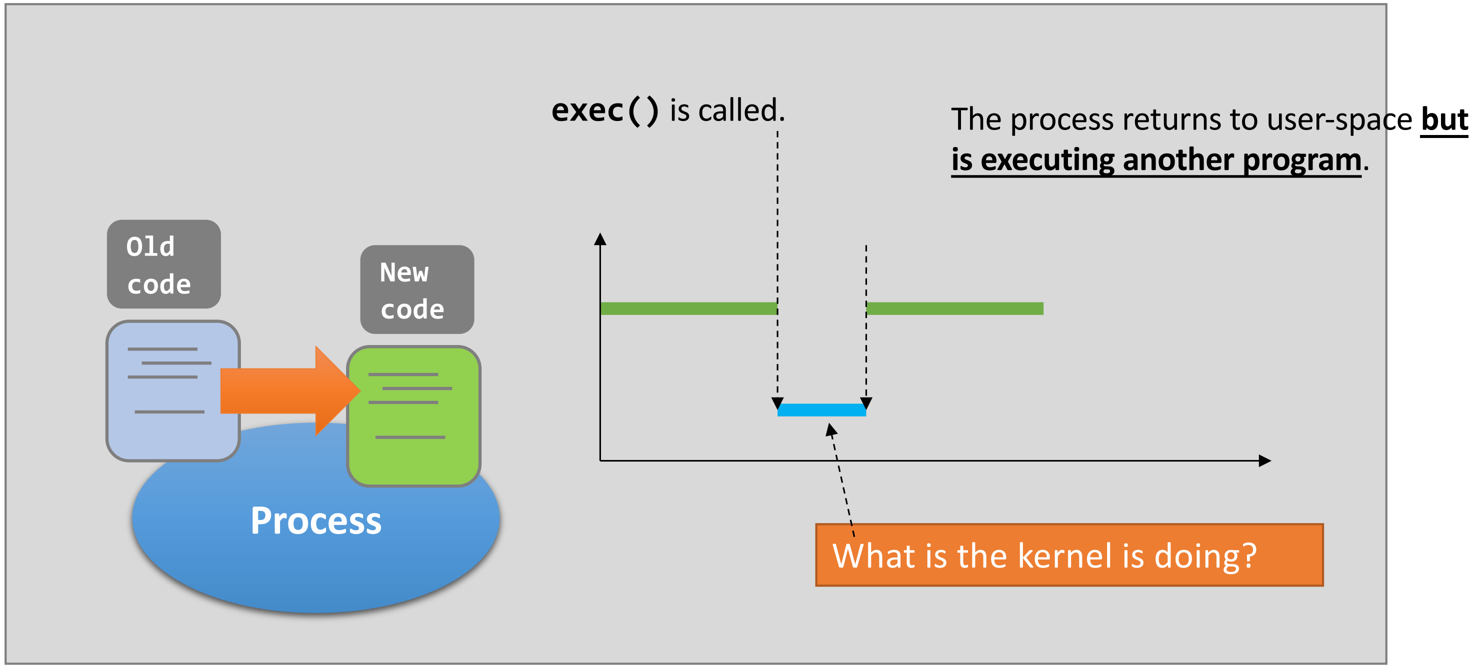
fork(): Opened Files

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

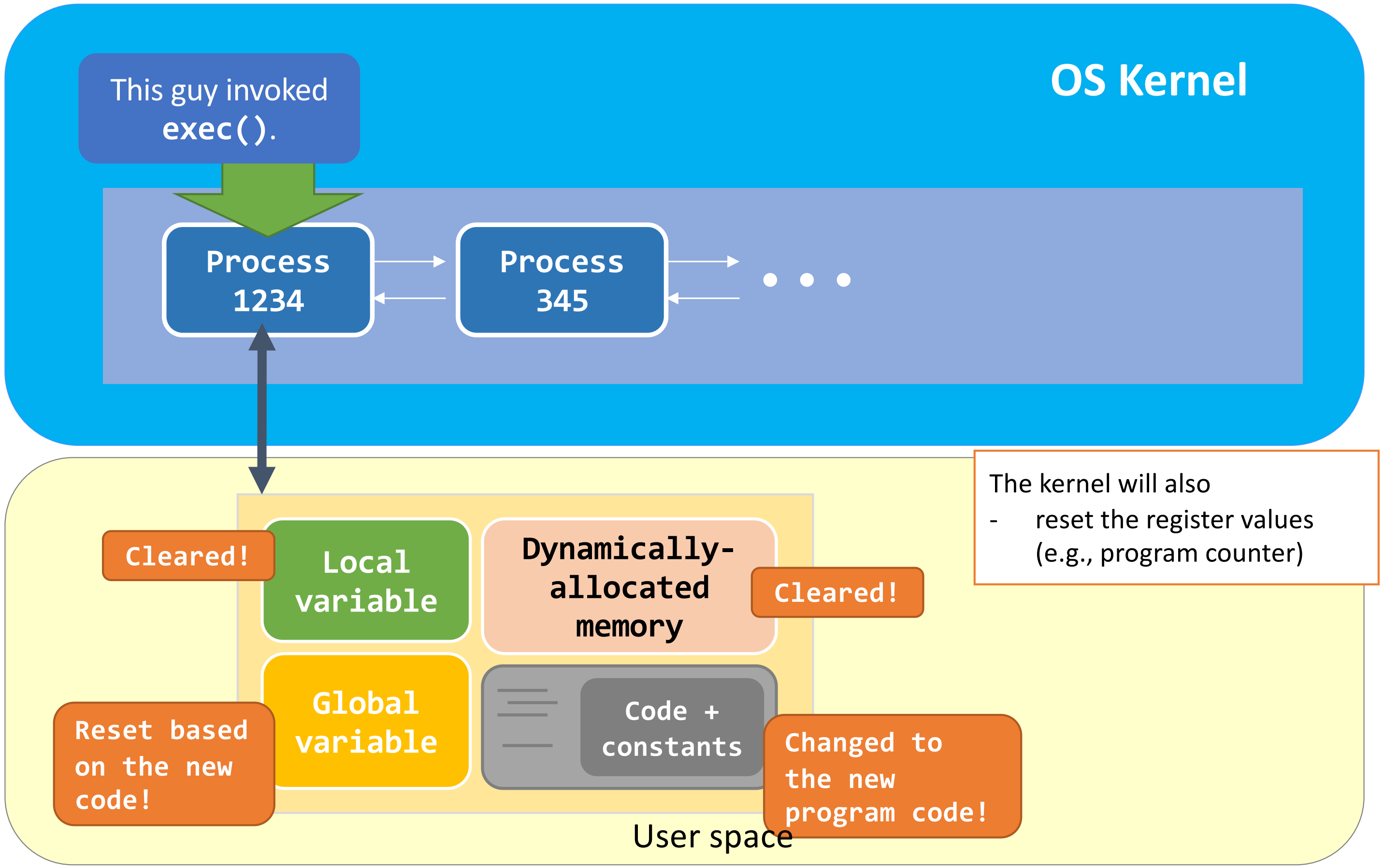


Let's see what will happen when the program finishes running!

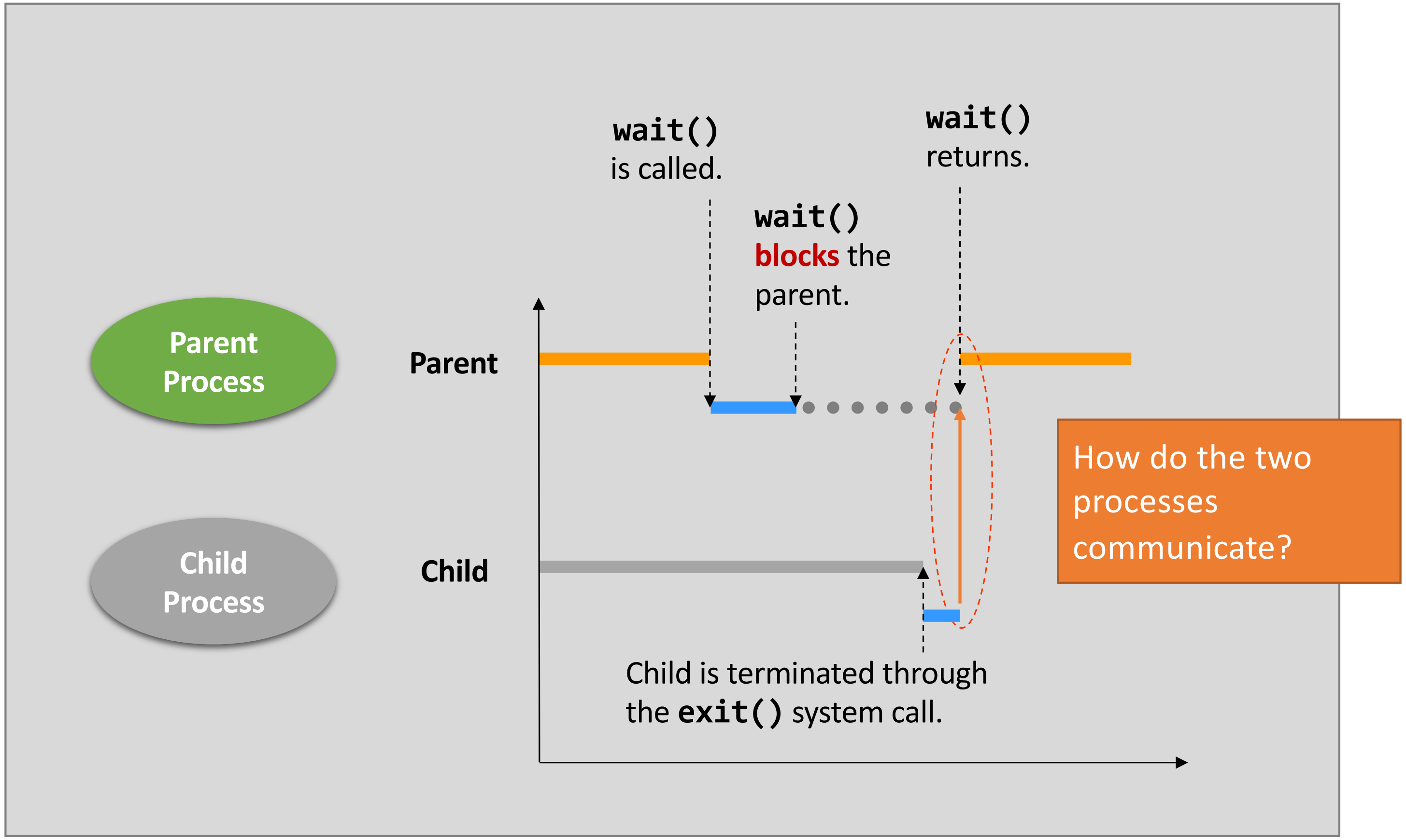
exec() in the User Mode



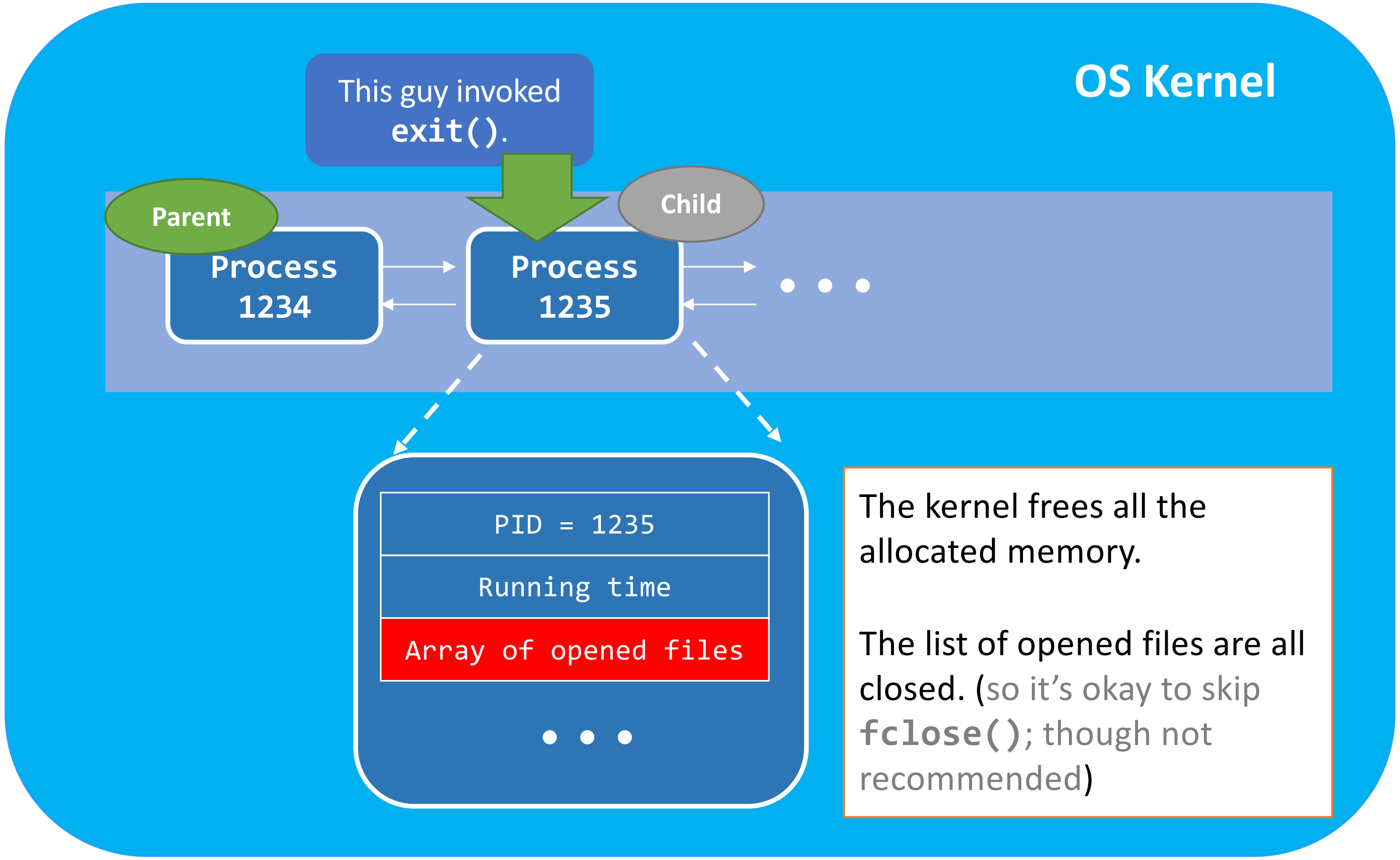
exec(): Kernel View



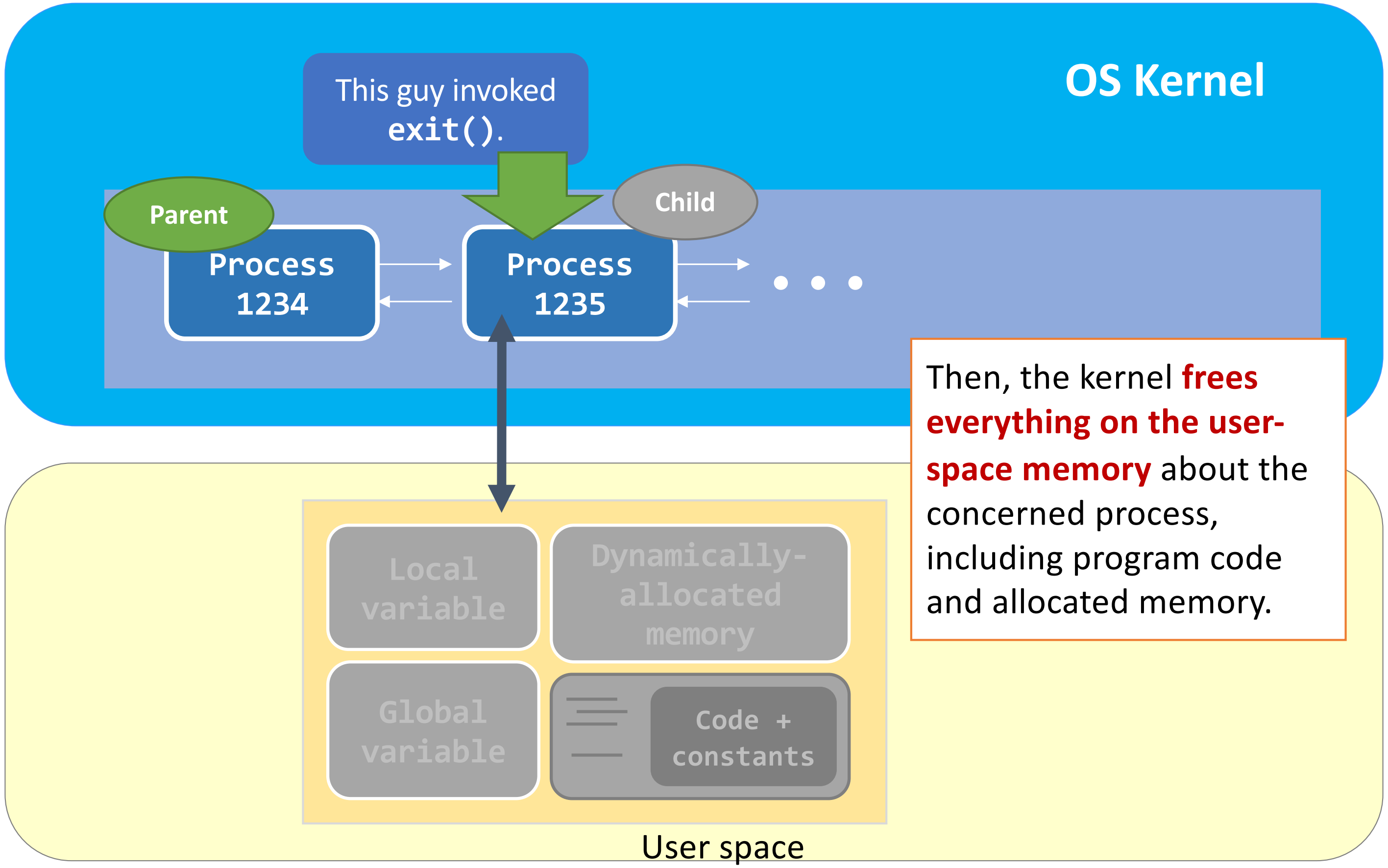
wait() and exit()



exit(): Kernel View

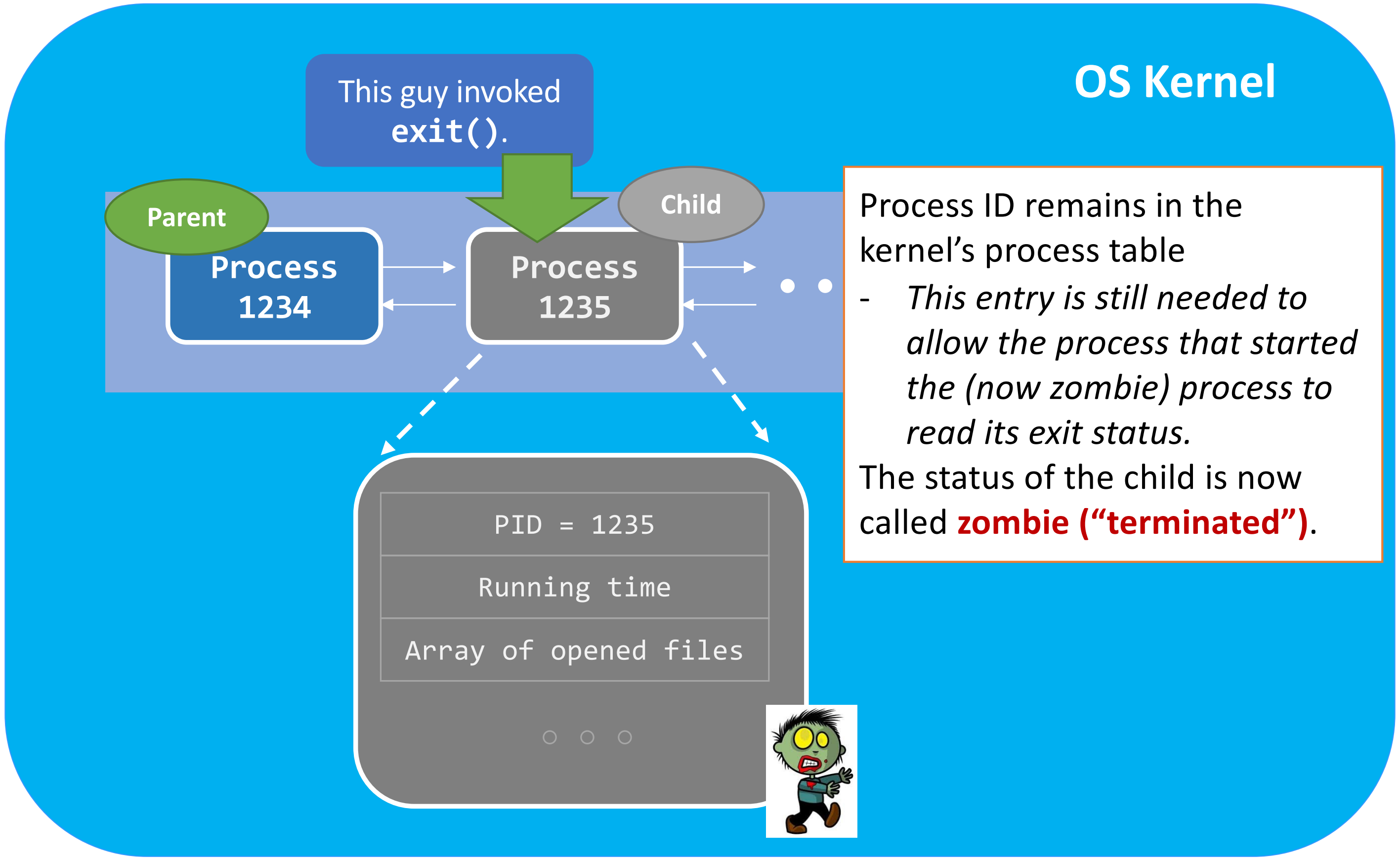


exit(): Kernel View

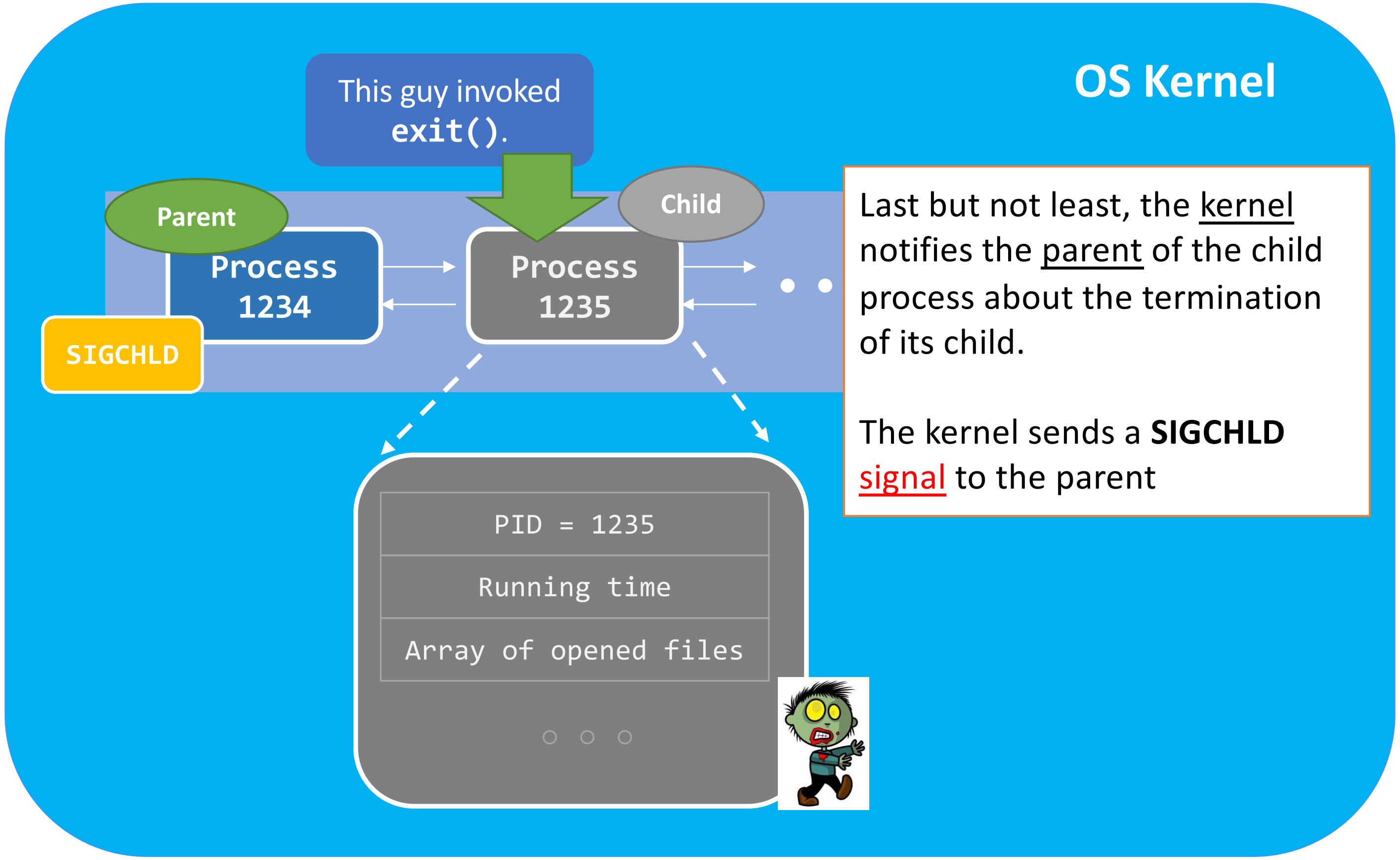


Then, the kernel **frees everything on the user-space memory** about the concerned process, including program code and allocated memory.

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.

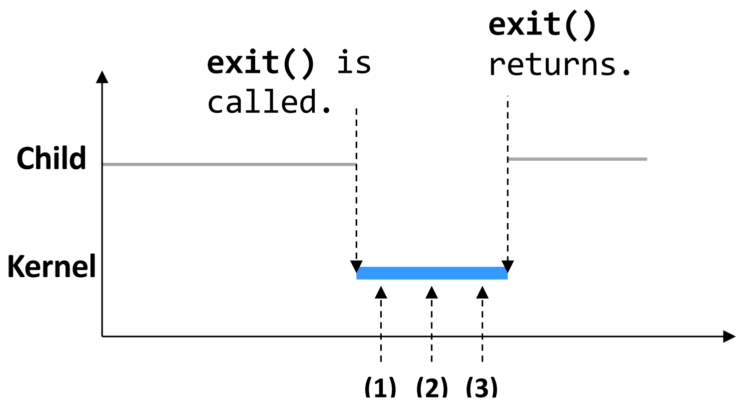
The kernel sends a **SIGCHLD** signal to the parent

exit(): Summary

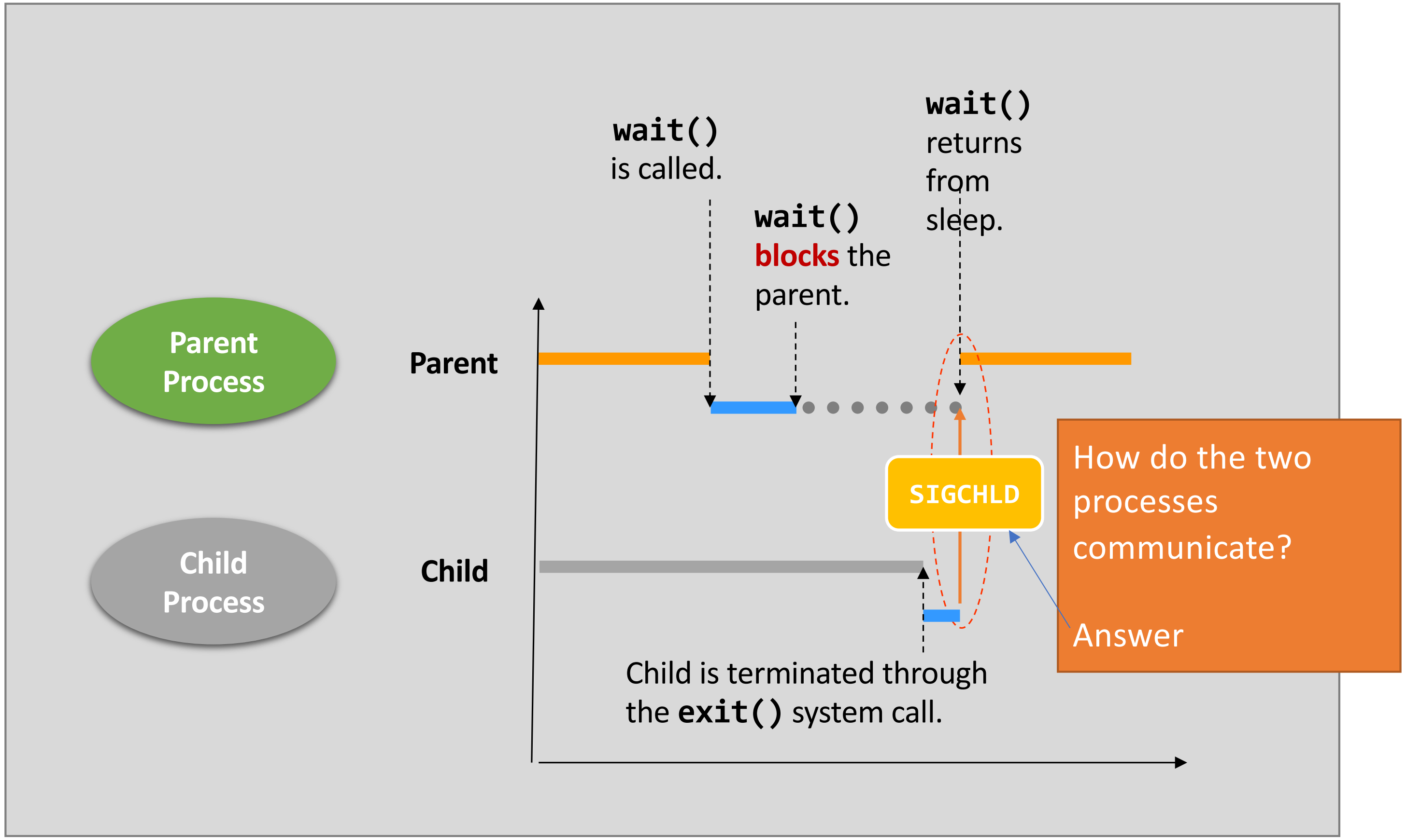
Step (1) Clean up most of the allocated kernel-space memory (e.g., process's running time info).

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

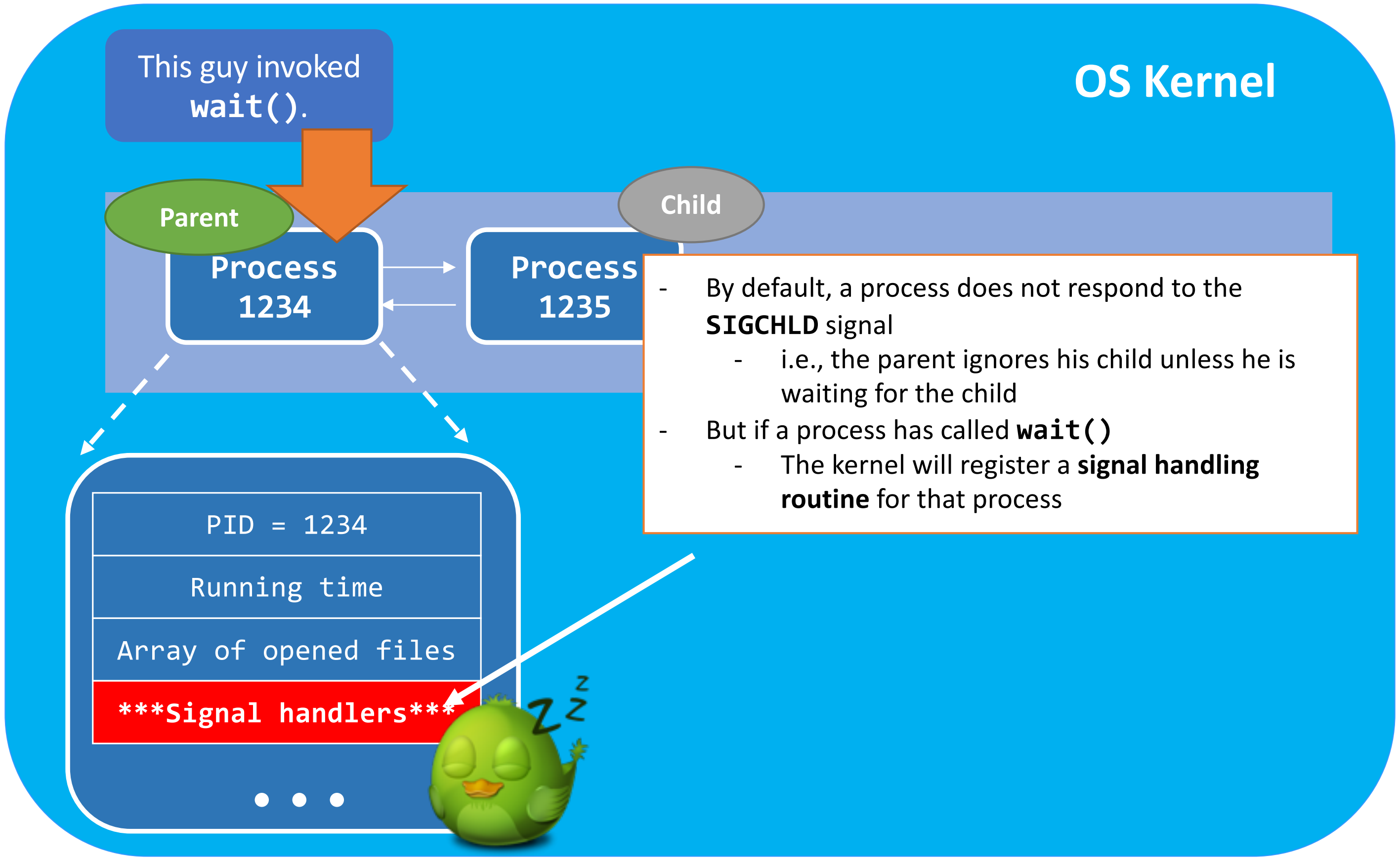
Step (3) Notify the parent with SIGCHLD.



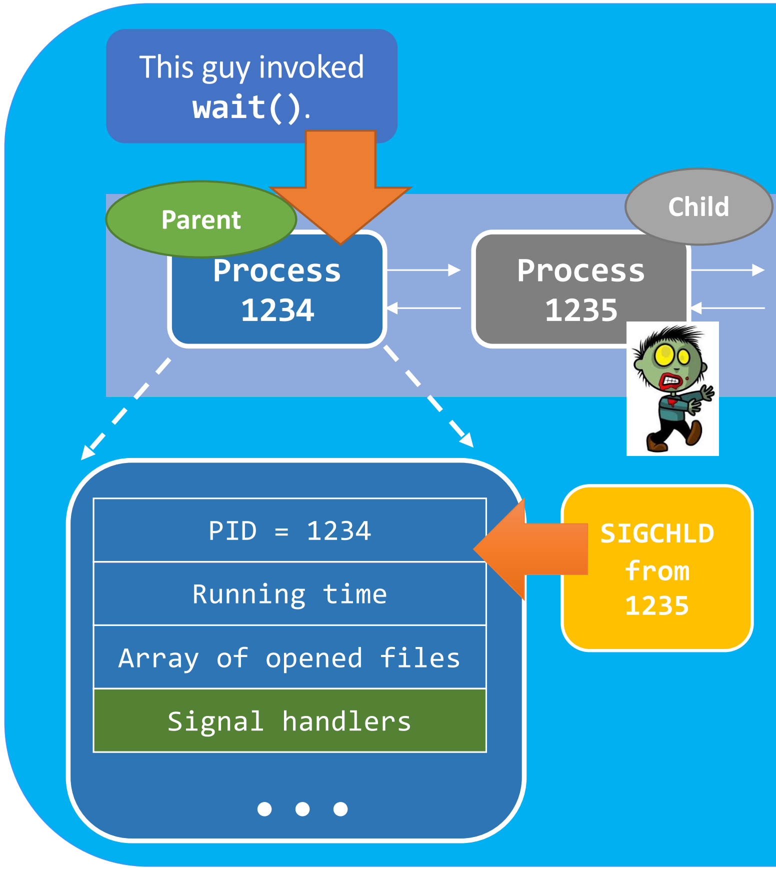
wait() and exit()



wait() Kernel View



wait() Kernel View

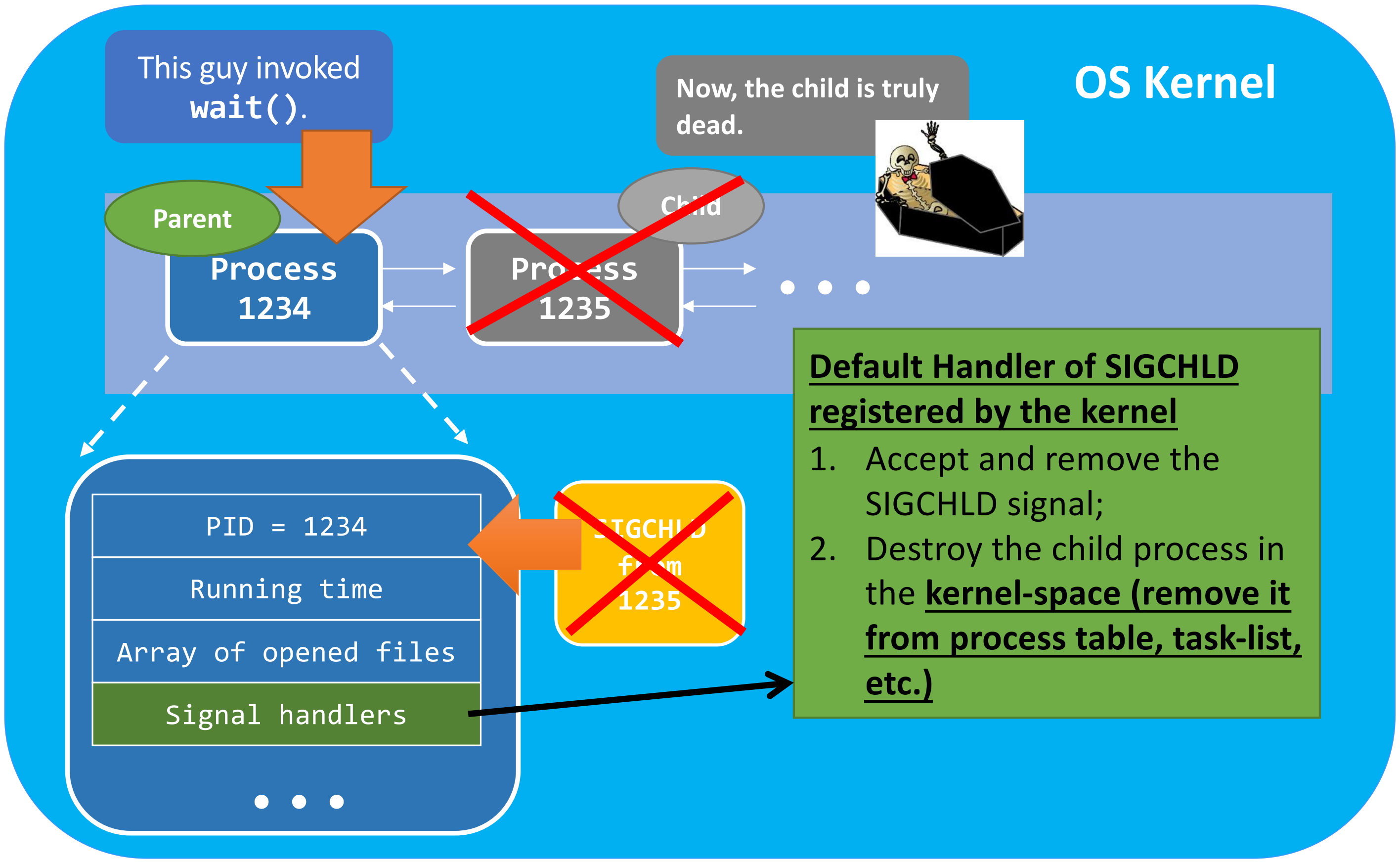


When **SIGCHLD** comes, the corresponding **signal handling routine is invoked**!

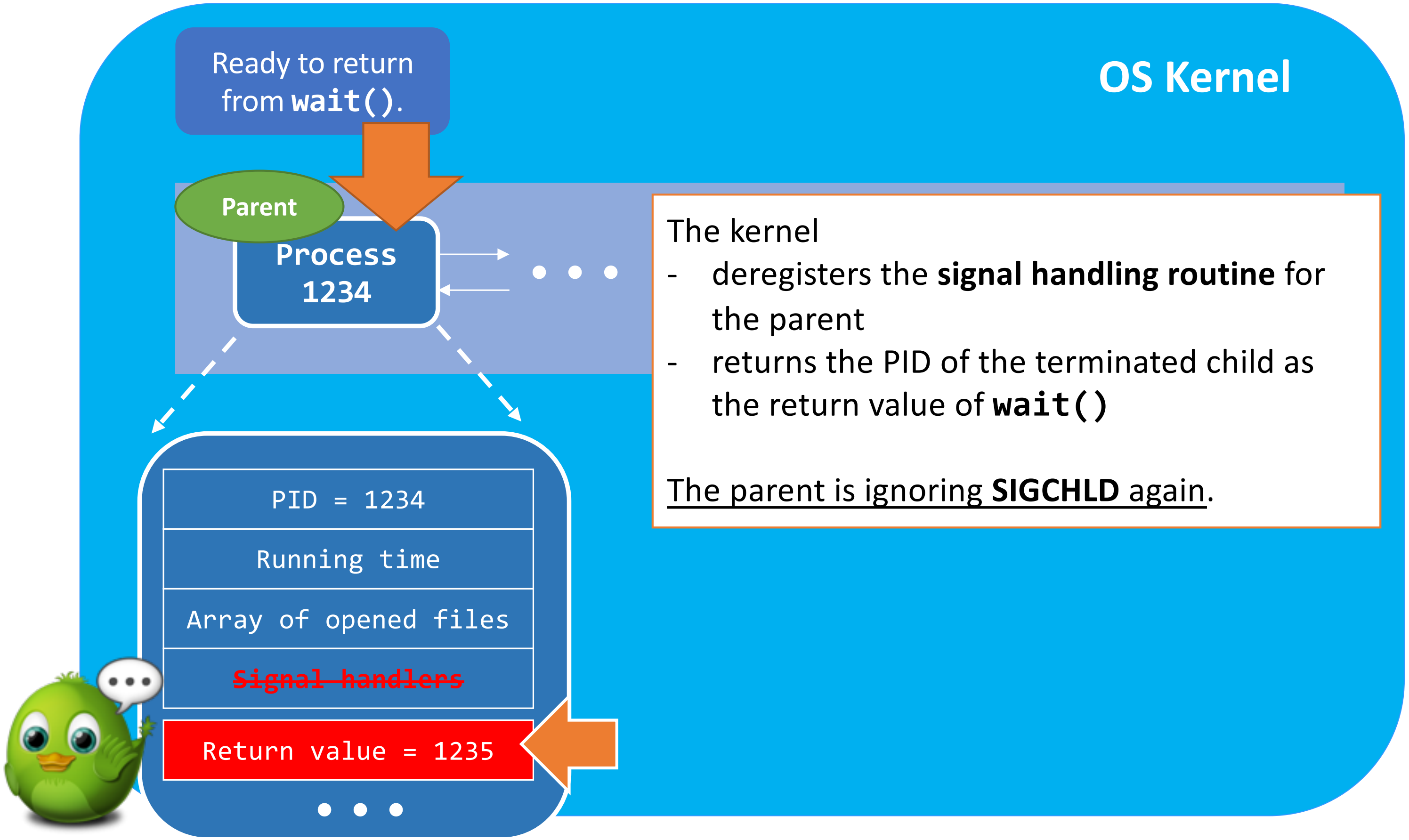
Note:

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

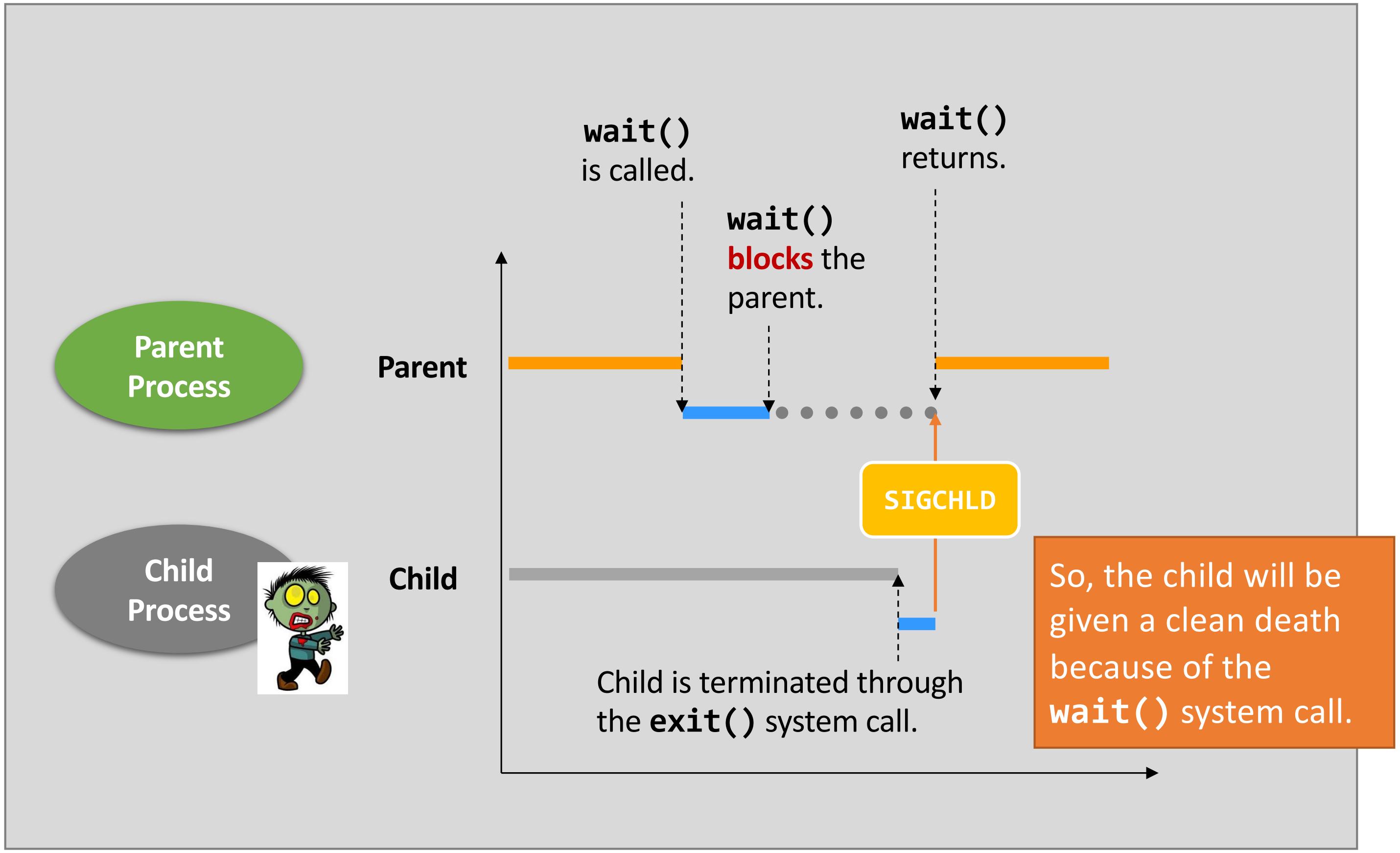
wait() Kernel View



wait() Kernel View

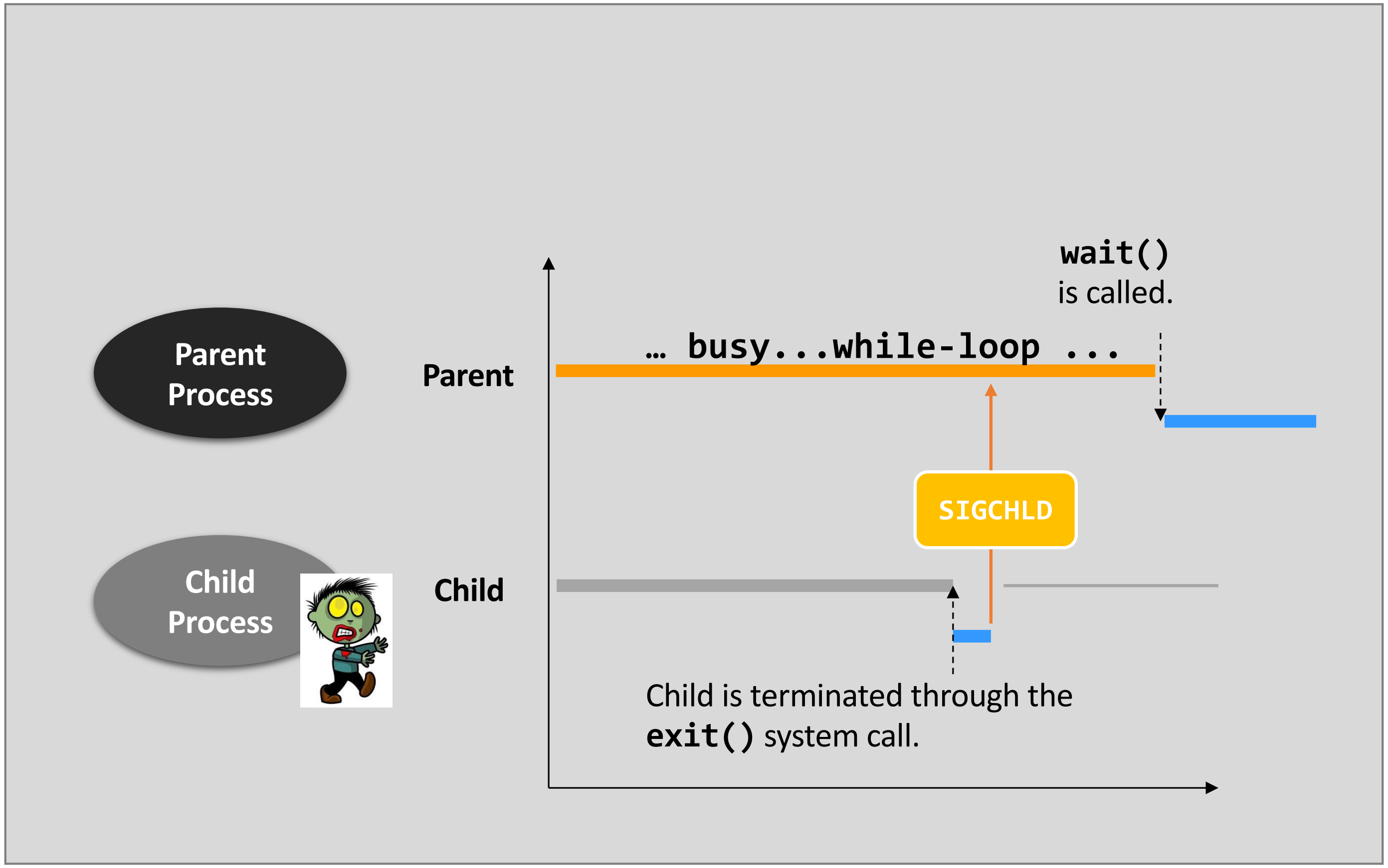


Normal Case

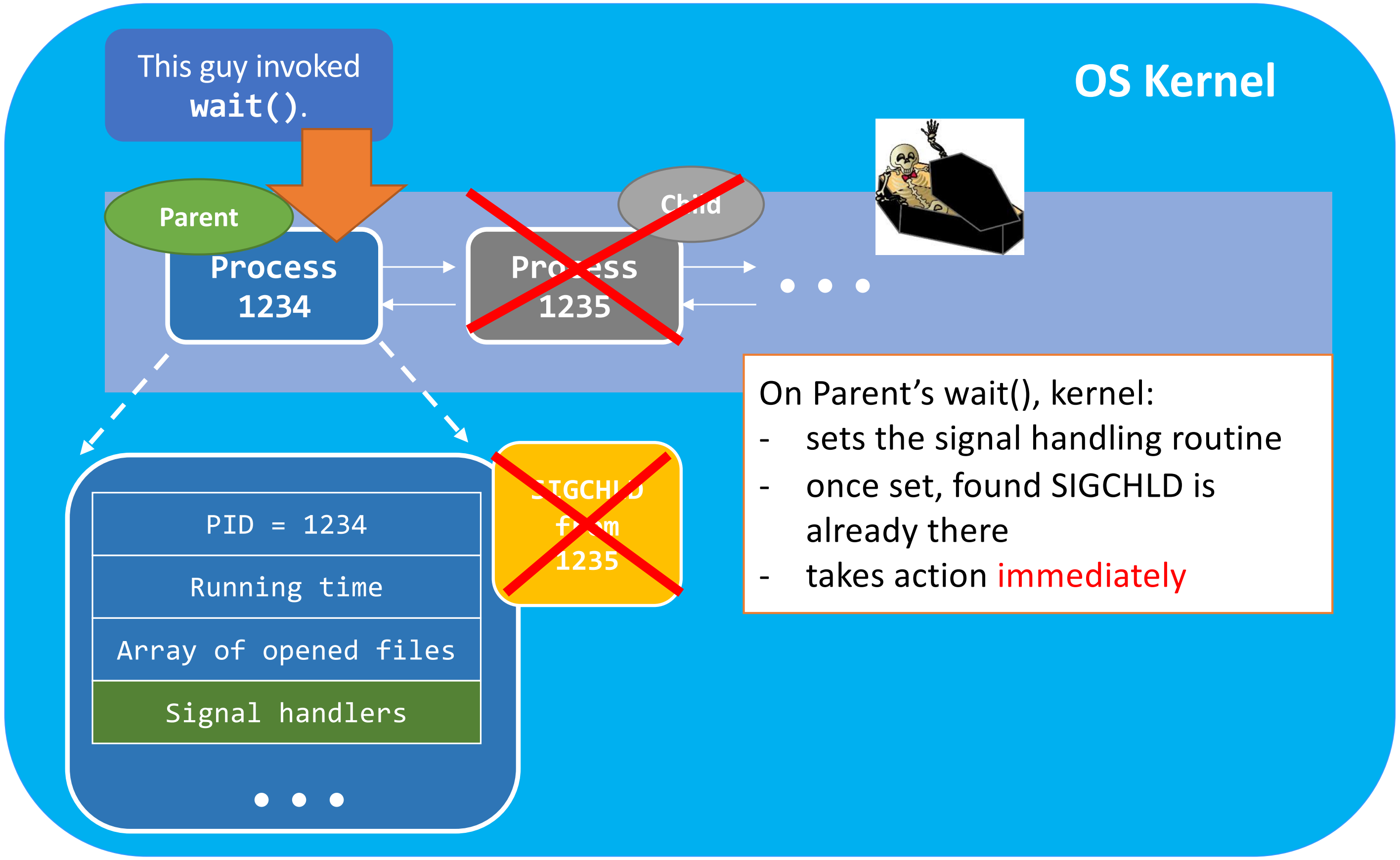


Parent Process
Child Process
Parent
Child
wait() is called.
wait() blocks the parent.
wait() returns.
SIGCHLD
Child is terminated through the exit() system call.
So, the child will be given a clean death because of the wait() system call.

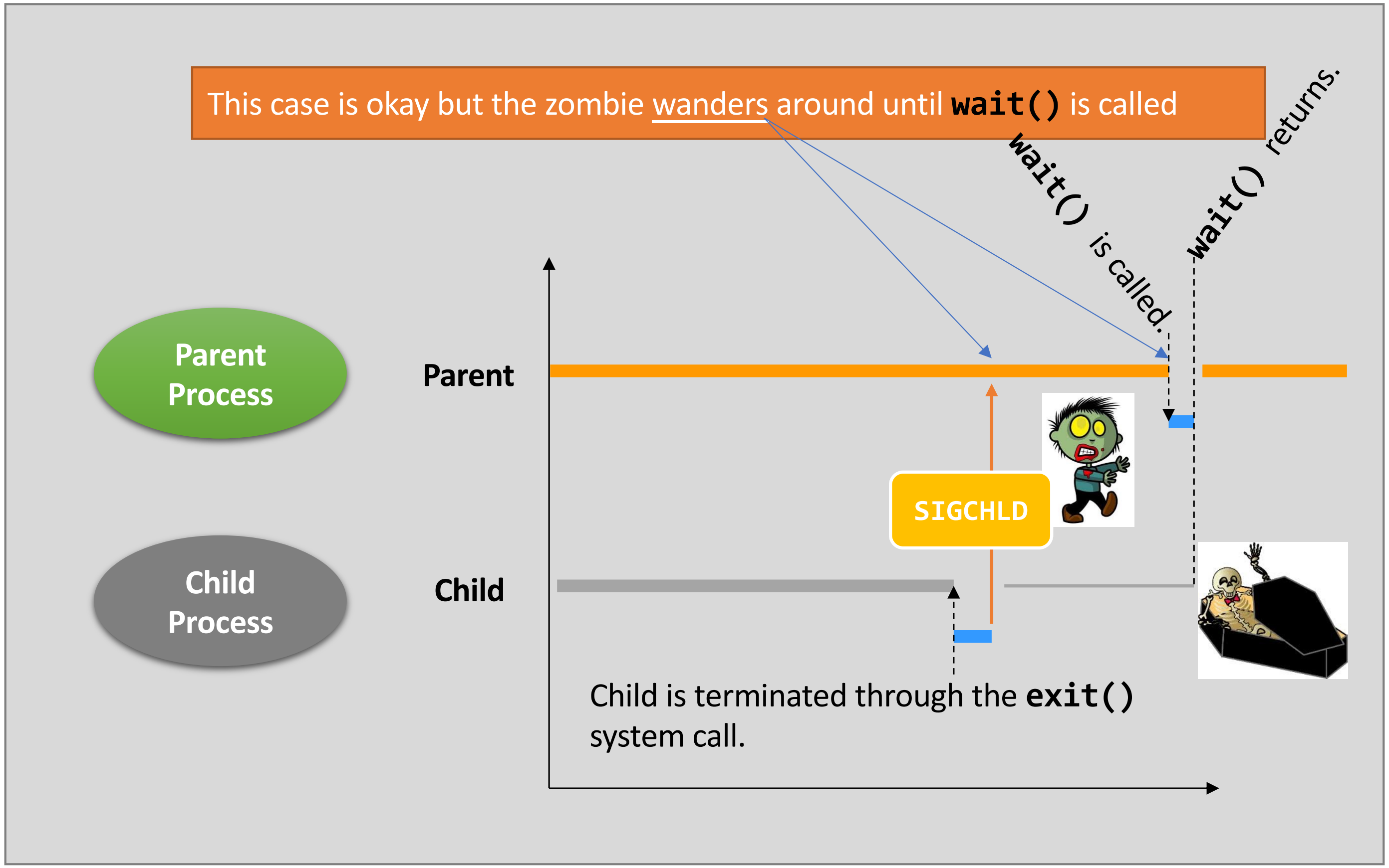
Parent's wait() after Child's exit()



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



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



Summary of wait() and exit()

- **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.

Summary of wait() and exit()

- **`wait()`** & **`waitpid()`** 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

Summary of wait() and exit()

```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.

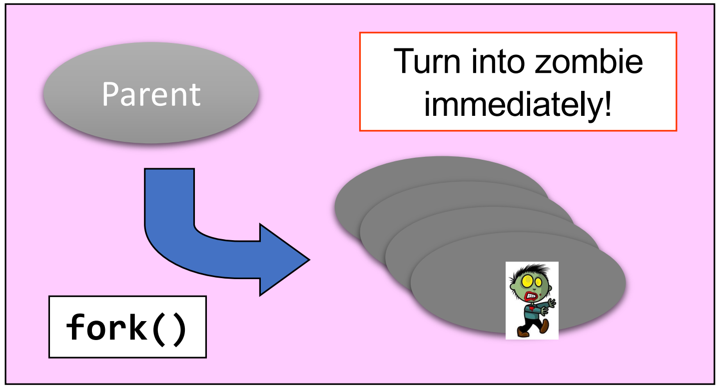
You are expected to see **the status of the child process changes (ps aux [PID])** between the 1st and the 2nd “enter”.

Using wait() for Resource Management

- 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?

Using wait() for Resource Management

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




```
$ ./interesting
_
```

Terminal A

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

Terminal B

More about Processes

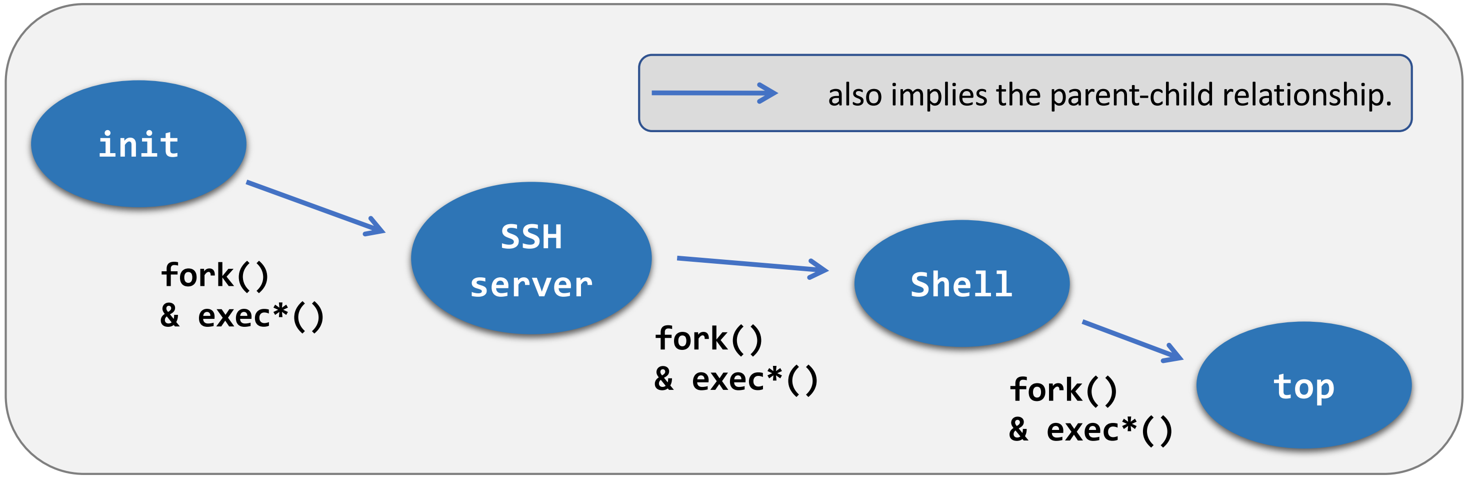
The first process

- 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().

How does uCore create the first process?

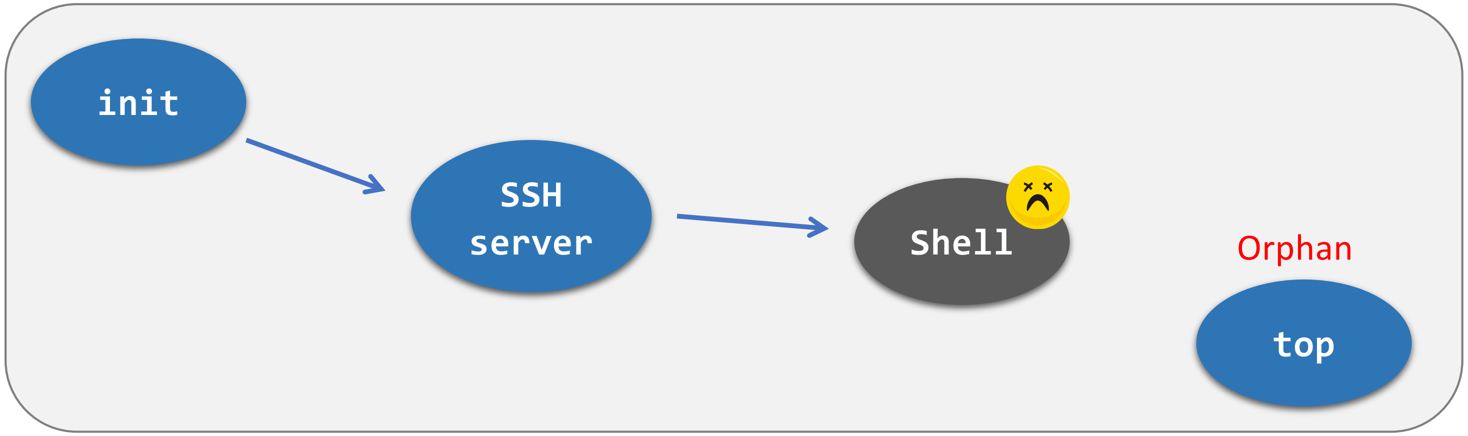
A Tree of Processes

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



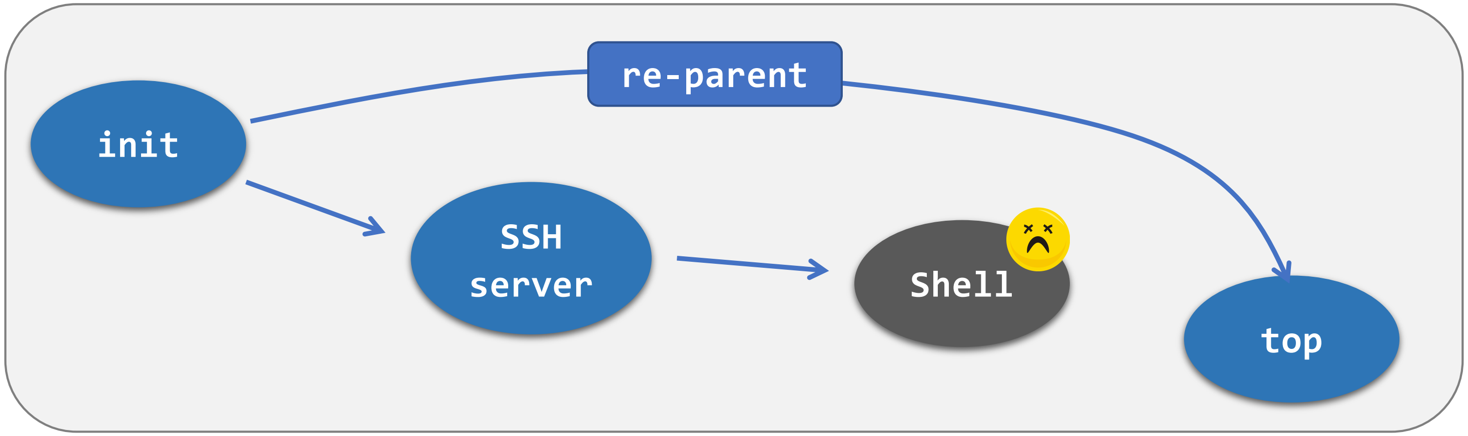
Orphans

- 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.



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*……………



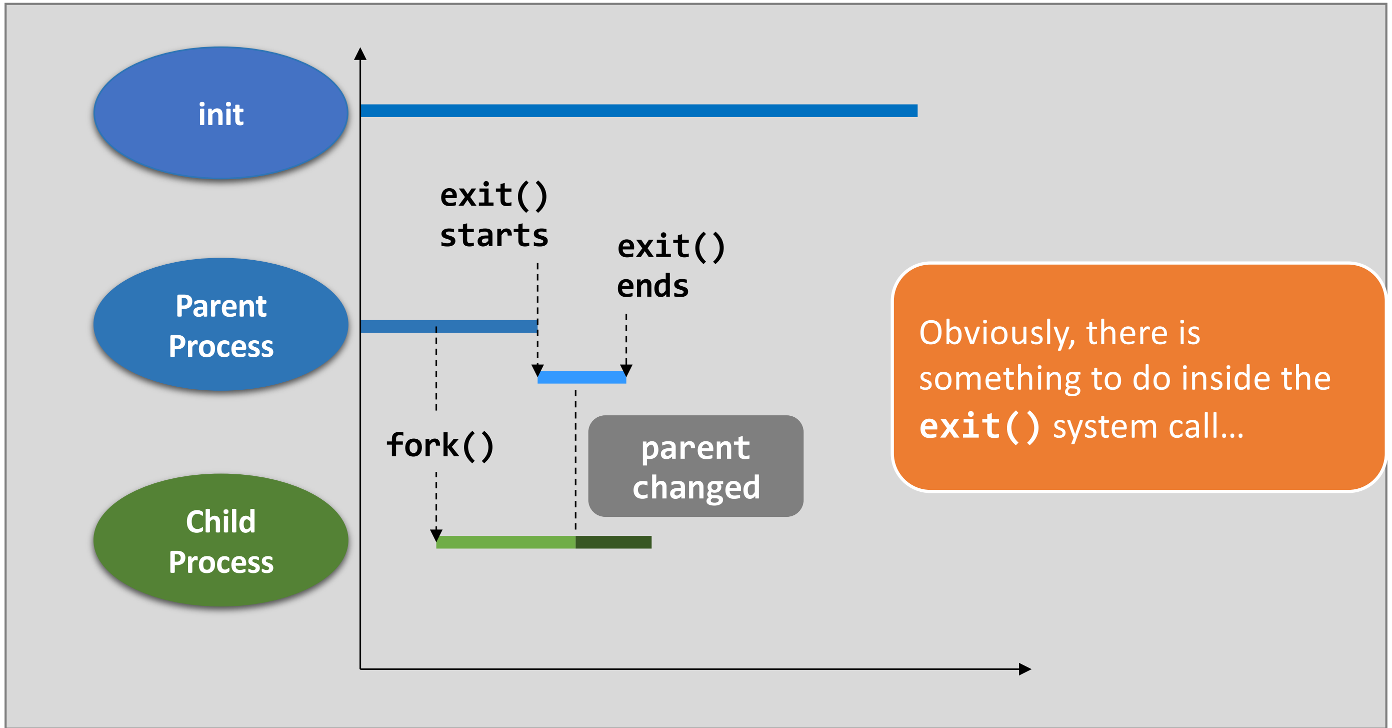
An Example

```
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());
}
```

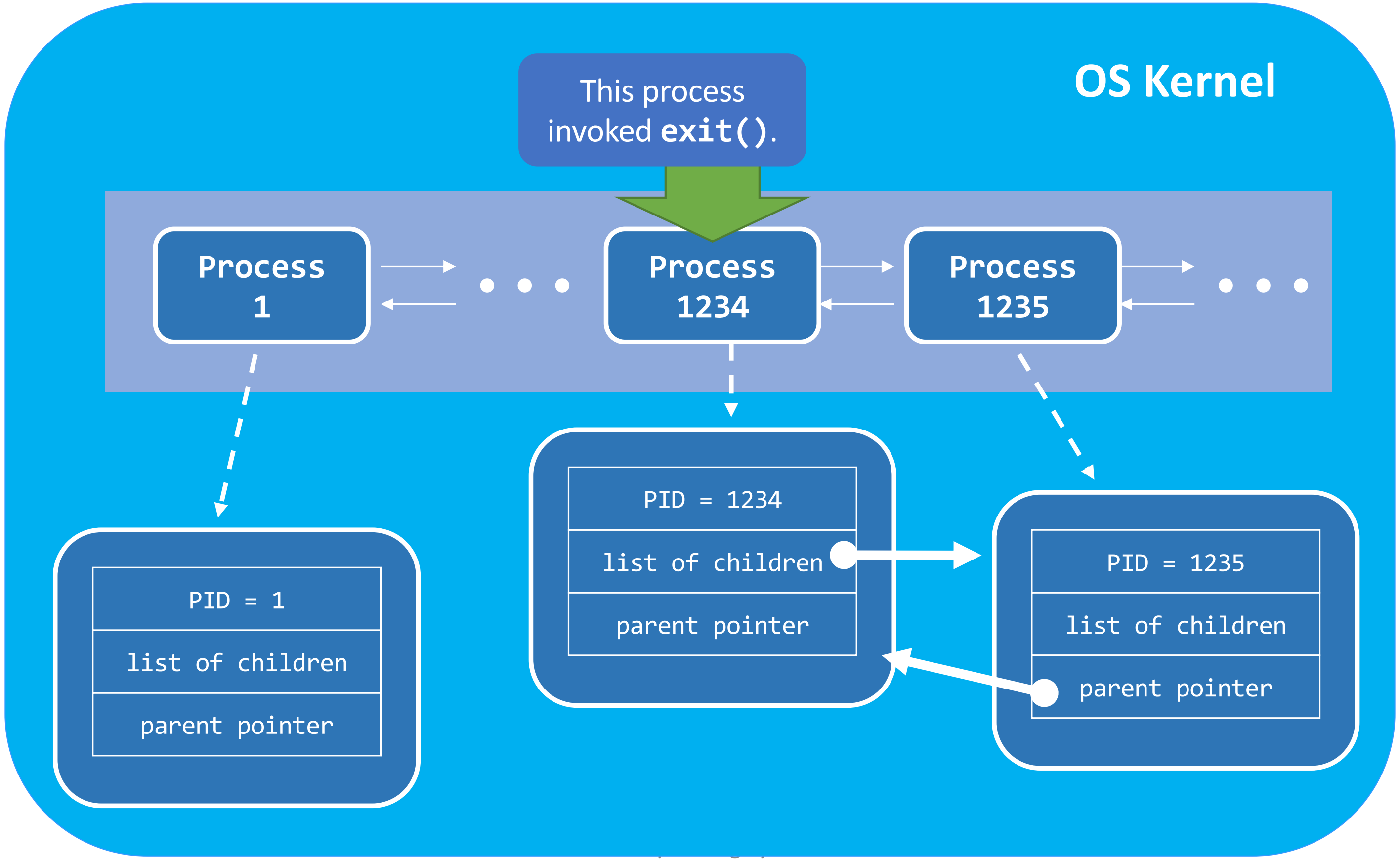
```
$ ./reparent
(1235) parent's PID = 1234
(1235) parent's PID = 1234
(1234) bye.
$ (1235) parent’s PID = 1
(1235) parent’s PID = 1
(1235) parent’s PID = 1
(1235) bye.
$ _
```

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

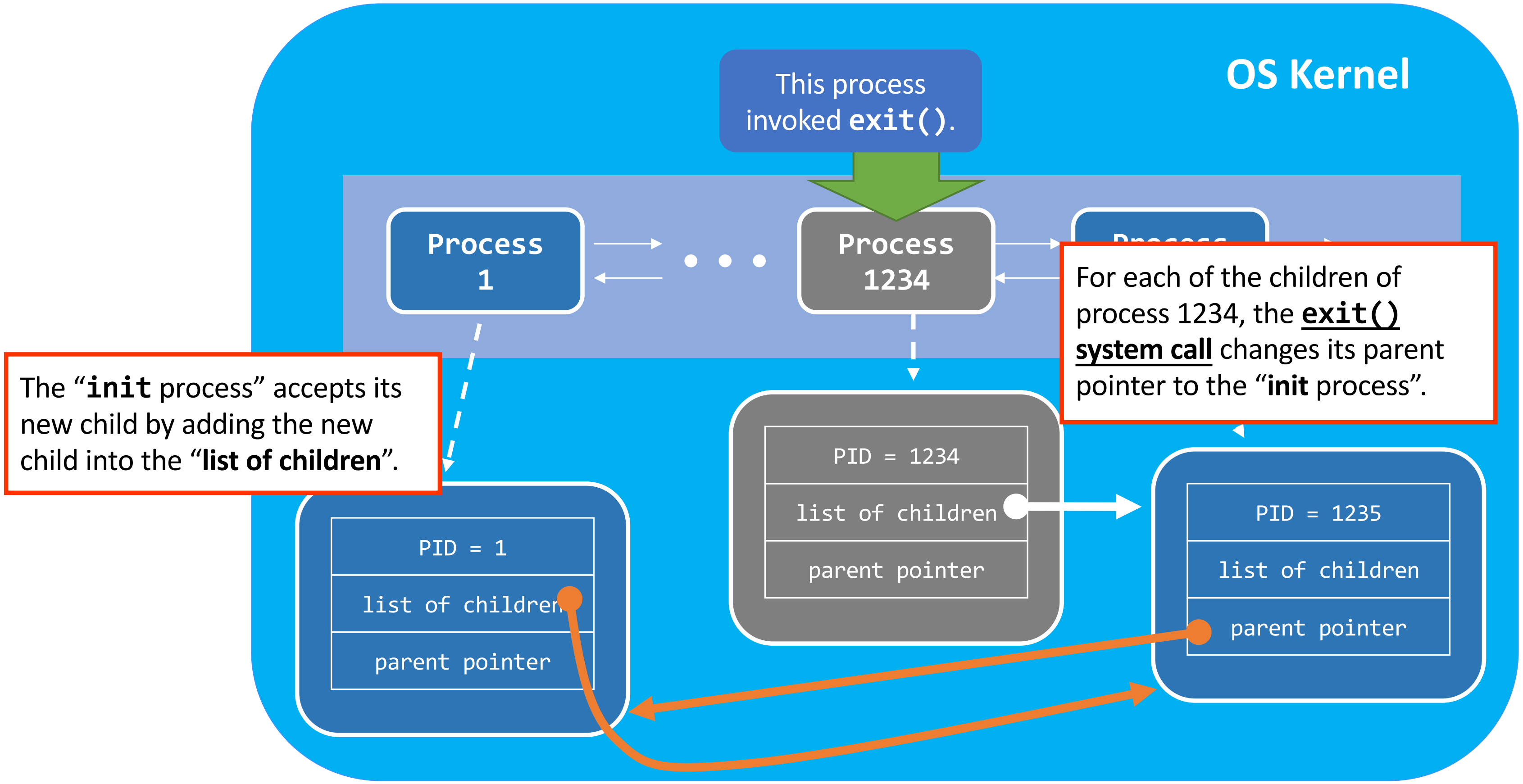
Re-parenting Explained



Re-parenting Explained (Cont'd)



Re-parenting Explained (Cont'd)



Background Jobs

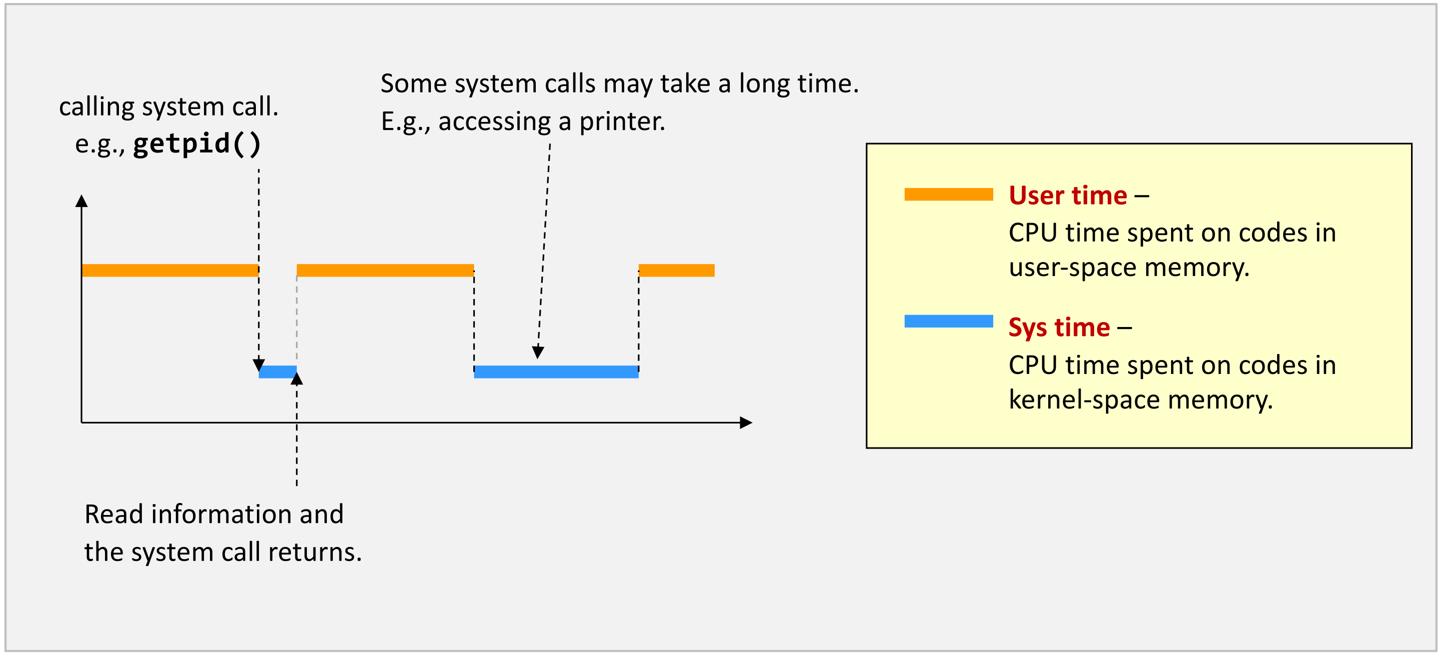
- The re-parenting operation enables something called **background jobs** in Linux
  - It allows a process runs **without a parent terminal/shell**

Back to home

```
$ ./infinite_loop &
$ exit

[ The shell is gone ]
```

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

Measure Process Time



User Time v.s. System Time (Case 1)

```
$ 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?

- real>user+sys
  I/O intensive
- real<user+sys
  multi-core

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

User Time v.s. System Time (Case 1)

```
$ 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;
}
```

User Time v.s. System Time (Case 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 v.s. System Time (Case 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
$ _
```

Thank you!