Lecture 3
Processes

Prof. Yinqian Zhang

Fall 2025

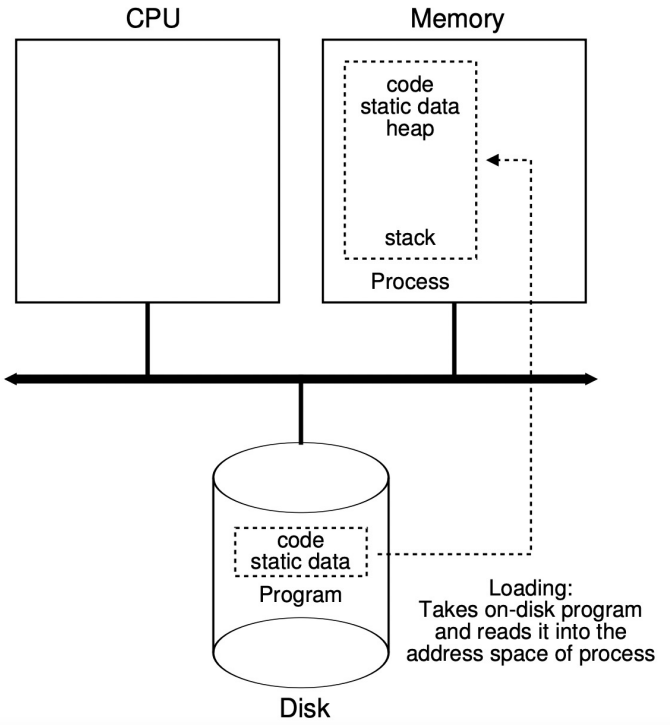
Outline

- Process and system calls
- Process creation
- Kernel view of processes
- Kernel view of fork(), exec(), and wait()
- More about processes

Process and System Calls

What Is a Process

- Process is a program in execution
- A program is a file on the disk
  - Code and static data
- A process is loaded by the OS
  - Code and static data are loaded from the program
  - Heap and stack are created by the OS

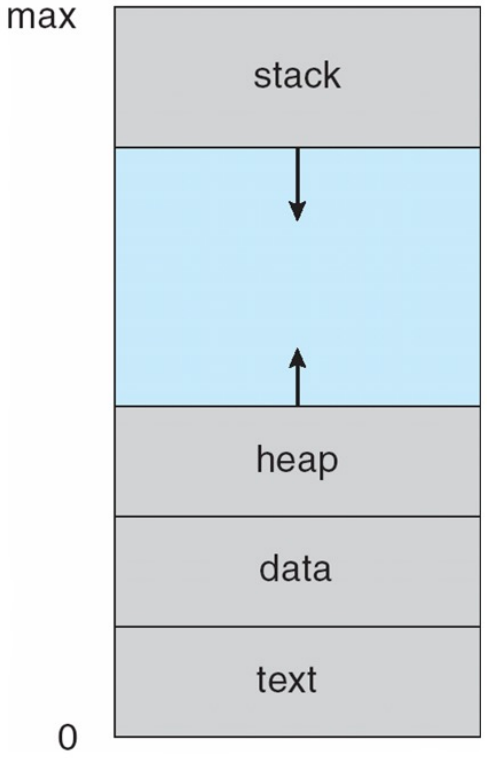


What Is a Process (Cont'd)

os创造了幻觉，看起来process有对于内存的完全控制权

- A process is an abstraction of machine states
  - Memory: address space
  - Register:
    - Program Counter (PC) or Instruction Pointer
    - Stack pointer
    - frame pointer
  - I/O: all files opened by the process

process包含的3部分



Process Identification

- How can we distinguish processes from one to another?
  - Each process is given a unique ID number, and is called the process ID, or the PID.
  - The system call, getpid(), prints the PID of the calling process.

同一个程序加载两次的两个process，不是同一个；而且可以同时存在

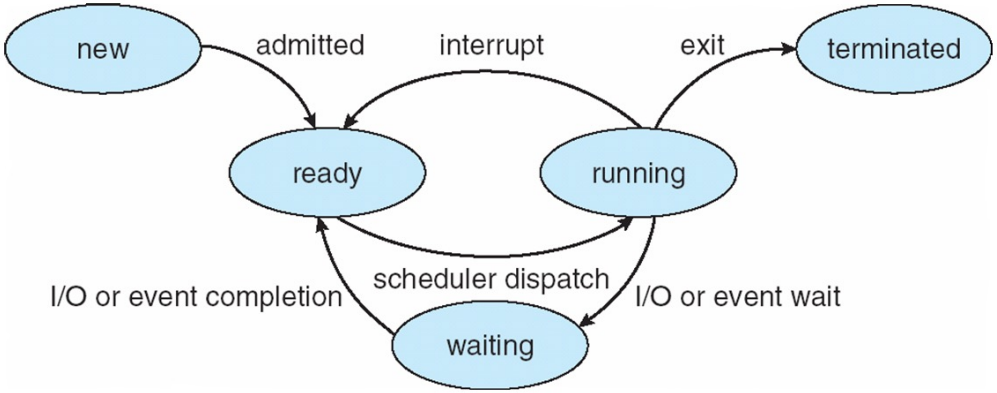
```
// compile to getpid
#include <stdio.h>   // printf()
#include <unistd.h>  // getpid()

int main(void) {
    printf("My PID is %d\n", getpid() );
}
```

```
$ ./getpid
My PID is 1234
$ ./getpid
My PID is 1235
$ ./getpid
My PID is 1237
```

Process Life Cycle

```
int main(void) {
    int x = 1;
    getchar();
    return x;
}
```

Process State

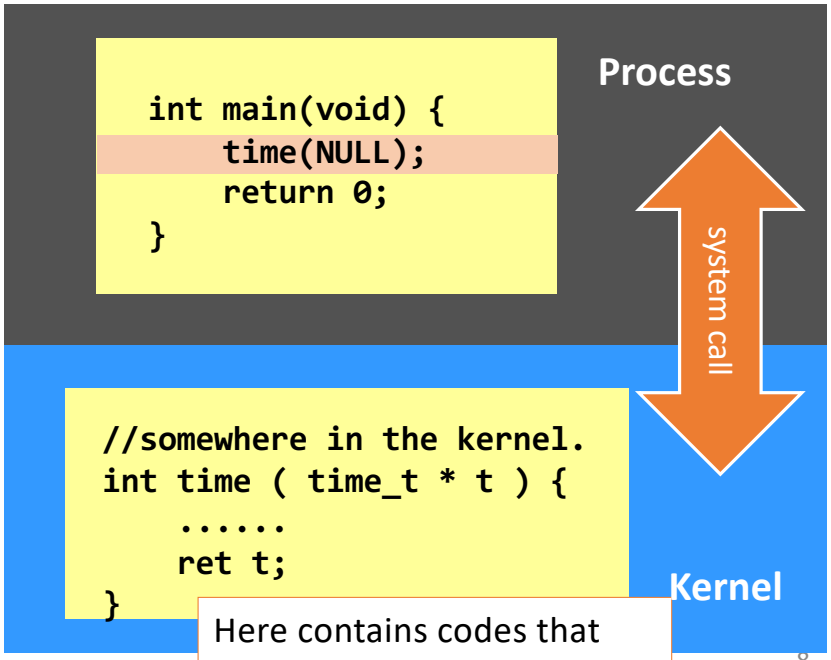


System Call: Process-Kernel Interaction

- System call is a function call.
  - exposed by the **kernel**.
  - abstraction of kernel operations.

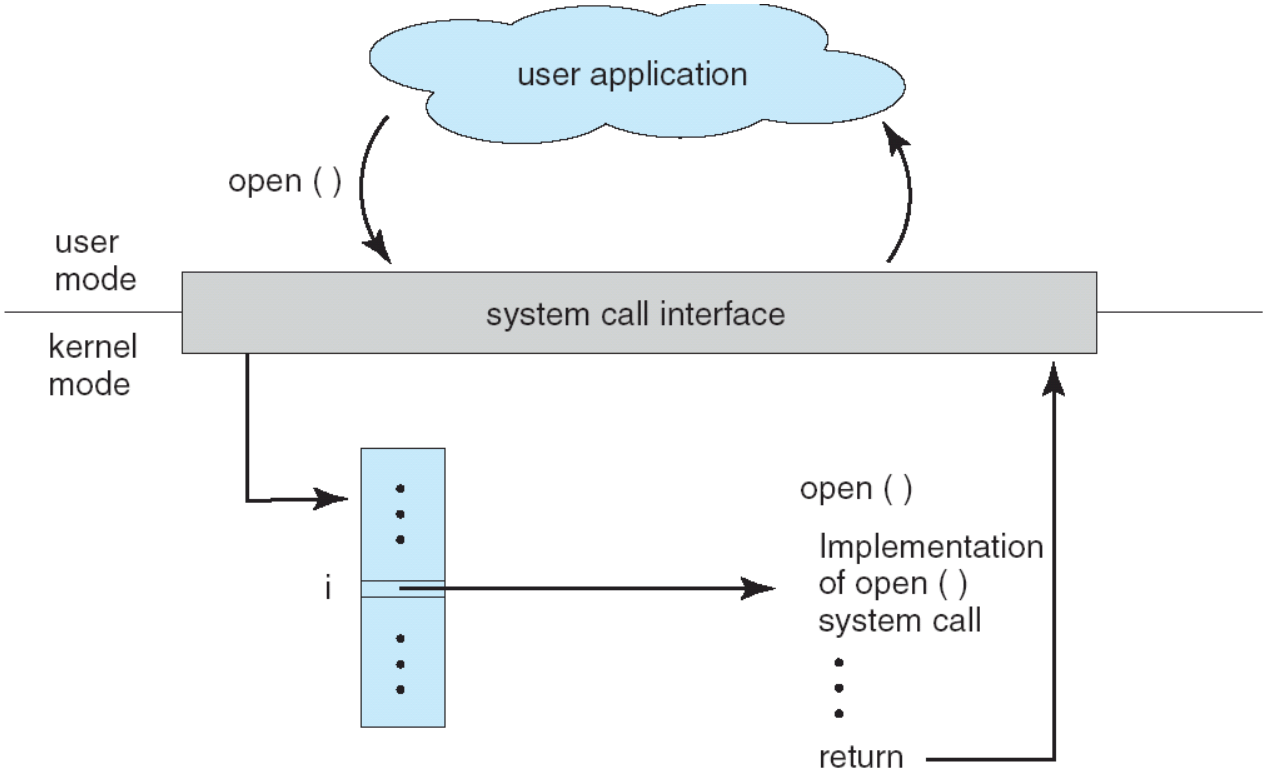
```
int add_function(int a, int b) {
    return (a + b);
}

int main(void) {
   int result;
   result = add_function(a,b);
   return 0;
}

// this is a dummy example…
```

This is a function call.



System Call: Call by Number

- System call is different from function call
- System call is a call by number



System Call: Call by Number

- User-mode code from xv6-riscv

```
int main(void) {
    ……
    int fd = open("copyin1", O_CREATE|O_WRONLY);
    ……
    return 0;
}
```

```
/* kernel/syscall.h */

#define SYS_open 15
```

```
/* user/usys.S */
.global open
open:
    li a7, SYS_open
    ecall
    ret
```

System Call: Call by Number

- Kernel code from xv6-riscv

```
/* kernel/syscall.h */

#define SYS_open 15
```

```
/* kernel/file.c */

uint64 sys_open(void) {
        ……
        return fd;
}
```

```
/* kernel/syscall.c */

static uint64 (*syscalls[])(void) = {
        …….
        [SYS_open] sys_open,
        …….
}

void syscall(void) (
        struct proc *p = myproc();
        num = p->trapframe->a7;
        p->trapframe->a0 = syscalls[num]();
}
```

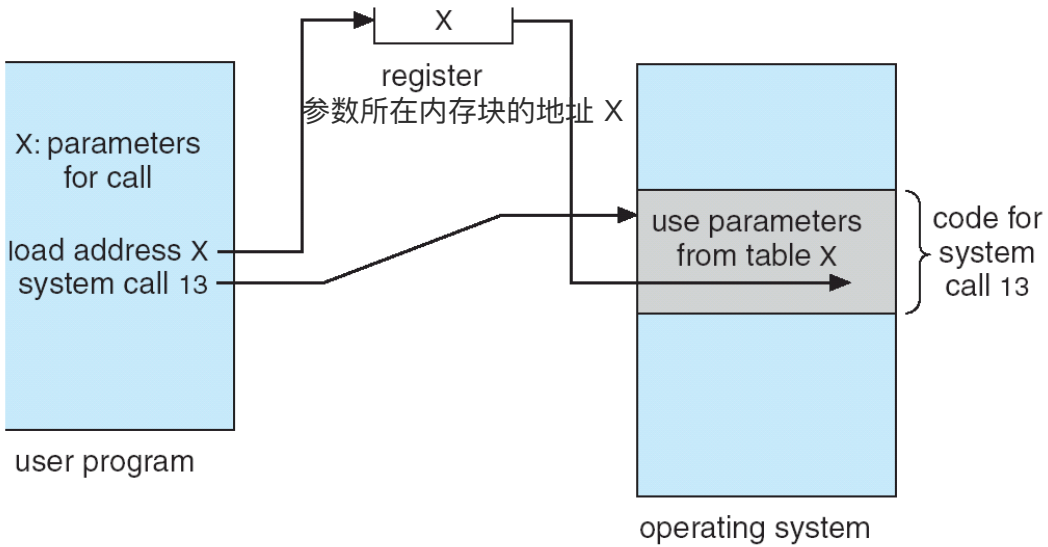
System Call: Parameter Passing

当用户程序要执行系统调用时，光靠“系统调用号”还不够，还需要把参数（比如文件描述符、内存地址、缓冲区长度等）传给内核

- Often, more information is required than the index of desired system call
  - Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
  - **Registers**: pass the parameters in registers
    - In some cases, may be more parameters than registers
    - x86 and risc-v take this approach
  - **Blocks**: Parameters stored in a memory block and address of the block passed as a parameter in a register
  - **Stack**: Parameters placed, or pushed, onto the stack by the program and popped off the stack by the operating system
    - Block and stack methods do not limit the number or length of parameters being passed

System Call: Parameter Passing

- Example: parameter passing via blocks



操作系统内核直接提供的接口

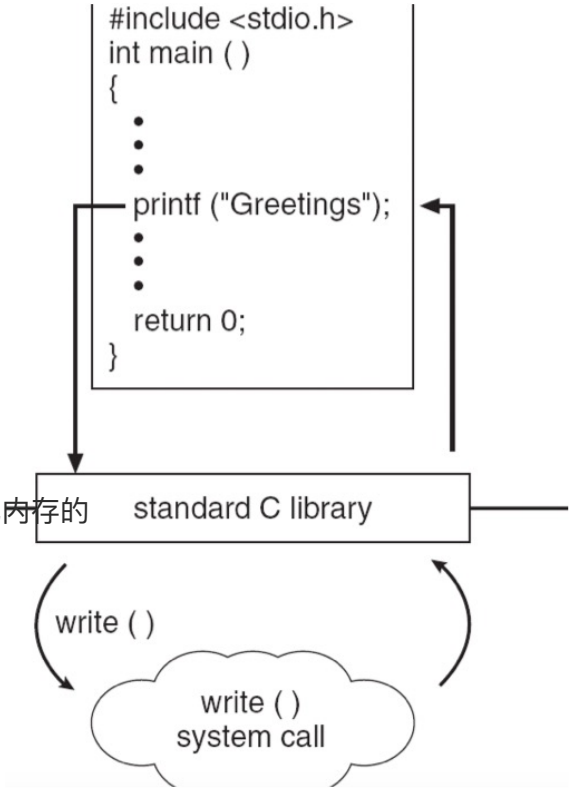
System Call v.s. Library API Call

涉及内核态 ↔ 用户态切换，权限高，可以访问硬件、管理资源

- Most operating systems provide standard C library to provide library API calls
  - A layer of indirection for system calls

Name	System call?
printf() & scanf()	No
malloc() & free()	No
fopen() & fclose()	No
mkdir() & rmdir()	Yes
chown() & chmod()	Yes

system call是以page来分配内存的

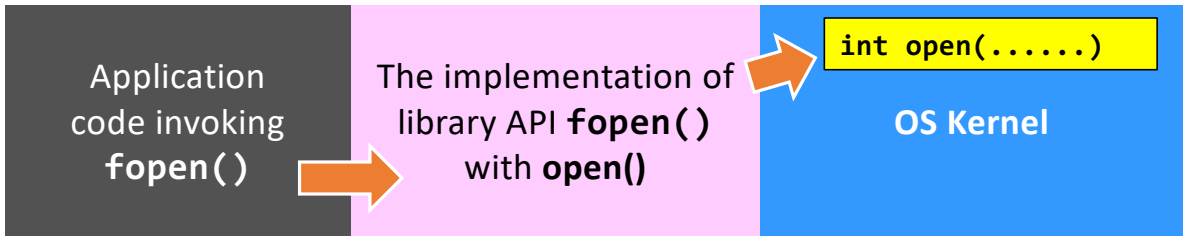


System Call v.s. Library API Call

- Take **fopen()** as an example.
  - **fopen()** invokes the system call **open()**.
  - **open()** is too primitive and is not programmer-friendly!

Library call	`fopen("hello.txt", "w");`
System call	`open("hello.txt", O_WRONLY | O_CREAT | O_TRUNC, 0666);`

0666是权限



Process Creation

Process Creation

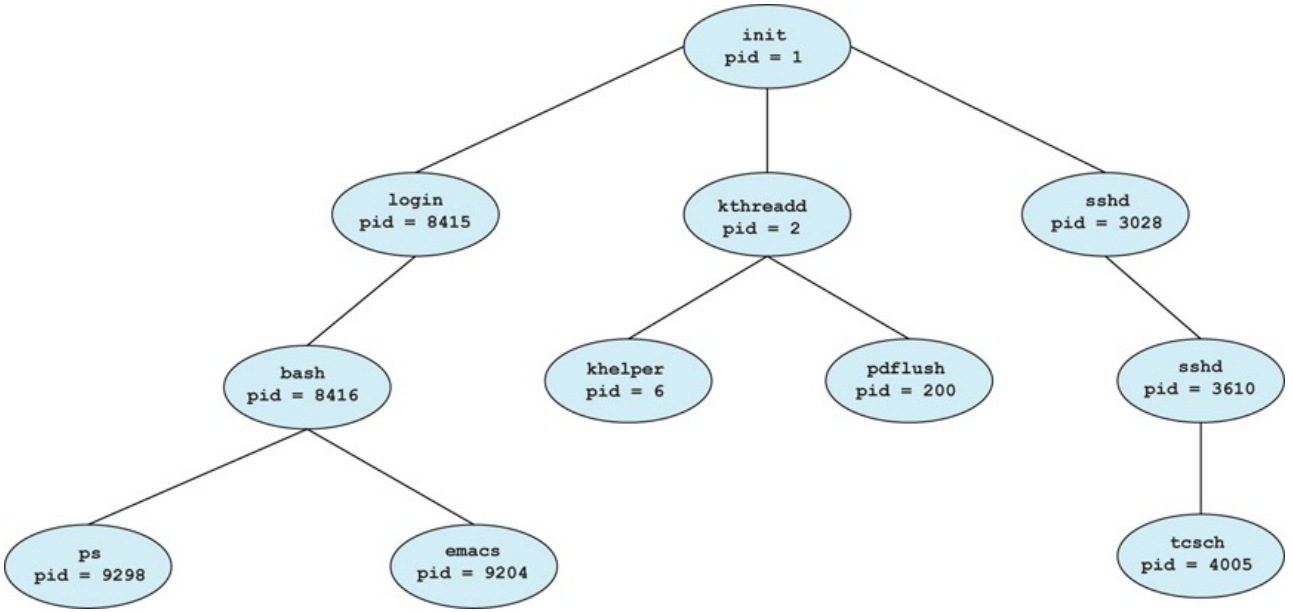
- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- Generally, process identified and managed via a process identifier (pid)
- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - fork system call creates new process
  - exec system call used after a fork to replace the process' memory space with a new program

在 fork() 系统调用里，子进程几乎是父进程的完整拷贝（代码段、数据段、堆栈都复制）
但父子进程有 不同的 PID，可以独立运行

子进程可以调用 exec() 系统调用，把自己当前的地址空间完全替换成一个新的程序。

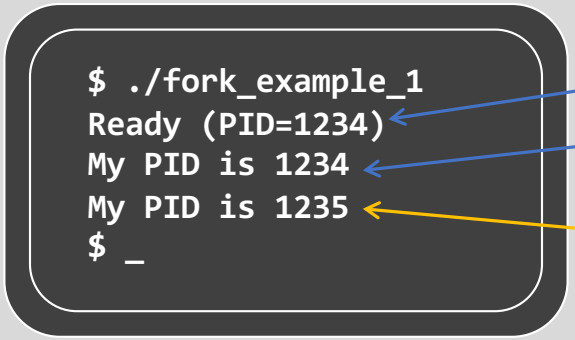
Process Creation (Cont'd)

- A tree of processes in Linux



Creating Processes with fork() System Call

在大多数操作系统中，一个进程都是通过另一个进程调用 fork() 或类似的系统调用创建的。
这就天然形成了 父进程 → 子进程 的关系



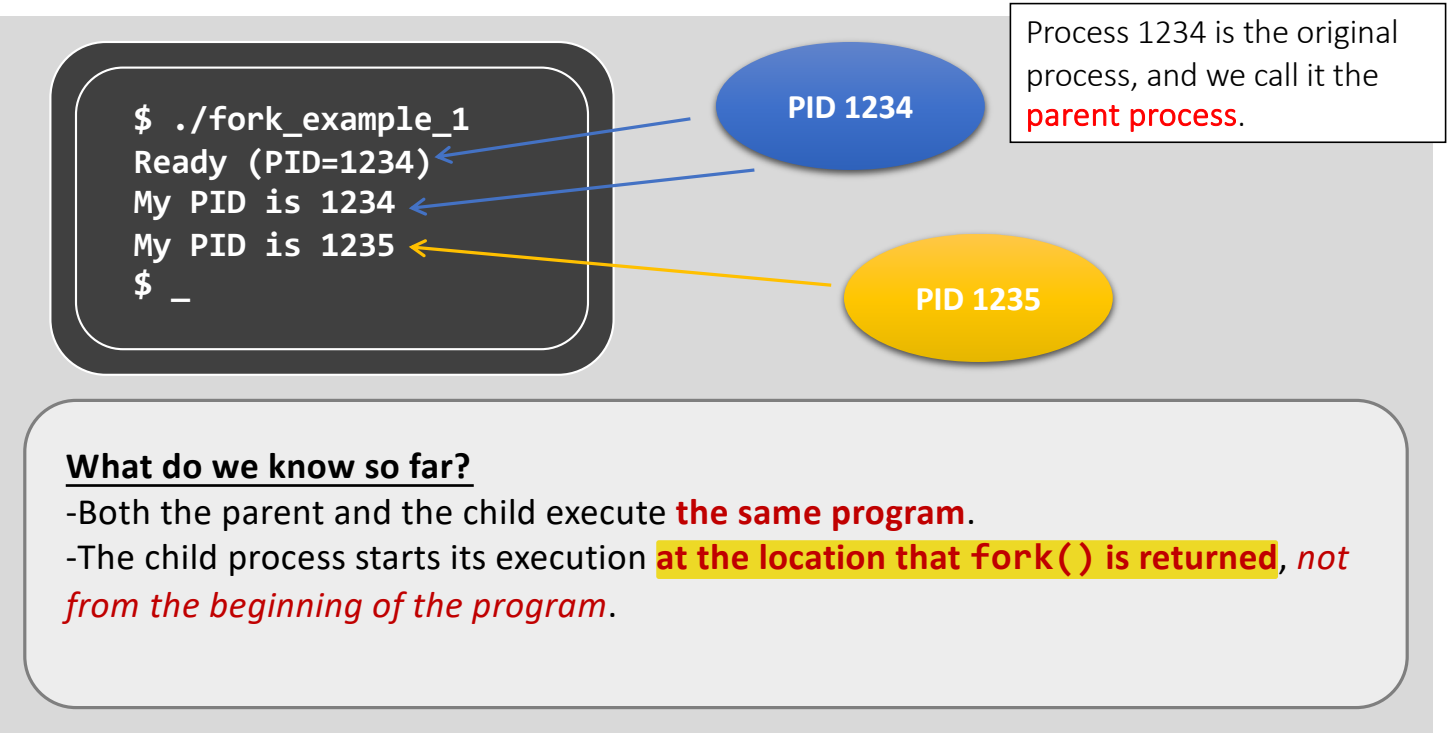
Process 1234 is the original process, and we call it the parent process.


```
int main(void) {
    printf("Ready (PID = %d)\n", getpid());
    fork();
    printf("My PID is %d\n", getpid() );
    return 0;
}
```

Process 1235 is created by the **fork()** system call, and we call it the child process.

父进程负责对子进程进行管理：
等待子进程退出 (wait() / waitpid())。
回收子进程的资源（否则子进程会变成“僵尸进程”）。
树状结构使得“谁创建的谁负责”非常直观。
如果一个父进程意外退出，内核会把它的子进程转交给 init（即“孤儿进程”由 init 收养），保证系统稳定。

Creating Processes with fork() System Call

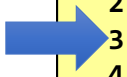


What do we know so far?

-Both the parent and the child execute **the same program**.

-The child process starts its execution **at the location that `fork()` is returned**, *not from the beginning of the program*.

Creating Processes with fork() System Call



```
 1  int main(void) {
 2    int result;
 3    printf("before fork ...\n");
 4    result = fork();
 5    printf("result = %d.\n", result);
 6
 7    if(result == 0) {
 8      printf("I'm the child.\n");
 9      printf("My PID is %d\n", getpid());
10    }
11    else {
12      printf("I'm the parent.\n");
13      printf("My PID is %d\n", getpid());
14    }
15
16    printf("program terminated.\n");
17  }
```

```
$ ./fork_example_2
before fork ...
```

PID 1234

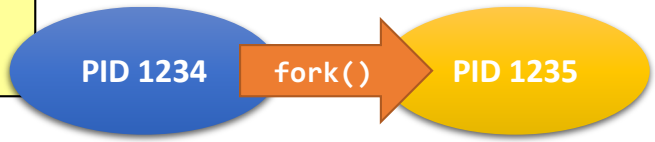
Creating Processes with fork() System Call

```
 1  int main(void) {
 2    int result;
 3    printf("before fork ...\n");
 4    result = fork();
 5    printf("result = %d.\n", result);
 6
 7    if(result == 0) {
 8      printf("I'm the child.\n");
 9      printf("My PID is %d\n", getpid());
10    }
11    else {
12      printf("I'm the parent.\n");
13      printf("My PID is %d\n", getpid());
14    }
15
16    printf("program terminated.\n");
17  }
```

```
$ ./fork_example_2
before fork ...
```

Important

- Both parent and child need to return from fork().
- CPU scheduler decides which to run first.



Creating Processes with fork() System Call

```
 1  int main(void) {
 2    int result;
 3    printf("before fork ...\n");
 4    result = fork();
 5    printf("result = %d.\n", result);
 6
 7    if(result == 0) {
 8      printf("I'm the child.\n");
 9      printf("My PID is %d\n", getpid());
10    }
11    else {
12      printf("I'm the parent.\n");
13      printf("My PID is %d\n", getpid());
14    }
15
16    printf("program terminated.\n");
17  }
```

```
$ ./fork_example_2
before fork ...
result = 1235
```

Important

For parent, the return value of **fork()** is the PID of the created child.

PID 1234 (running)

PID 1235 (ready)

child运行fork后面的代码（program counter也和parent相同）

子进程是父进程的 完整拷贝（包括变量值、堆、栈），只有在fork之后可能有差别（pid等等）

Creating Processes with fork() System Call

```
 1  int main(void) {
 2    int result;
 3    printf("before fork ...\n");
 4    result = fork();
 5    printf("result = %d.\n", result);
 6
 7    if(result == 0) {
 8      printf("I'm the child.\n");
 9      printf("My PID is %d\n", getpid());
10    }
11    else {
12      printf("I'm the parent.\n");
13      printf("My PID is %d\n", getpid());
14    }
15
16    printf("program terminated.\n");
17  }
```

result是0就是child

否则就是parent

```
$ ./fork_example_2
before fork ...
result = 1235
I'm the parent.
My PID is 1234
```

PID 1234 (running)

PID 1235 (ready)

Creating Processes with fork() System Call

```
 1  int main(void) {
 2    int result;
 3    printf("before fork ...\n");
 4    result = fork();
 5    printf("result = %d.\n", result);
 6
 7    if(result == 0) {
 8      printf("I'm the child.\n");
 9      printf("My PID is %d\n", getpid());
10    }
11    else {
12      printf("I'm the parent.\n");
13      printf("My PID is %d\n", getpid());
14    }
15
16    printf("program terminated.\n");
17  }
```

```
$ ./fork_example_2
before fork ...
result = 1235
I'm the parent.
My PID is 1234
program terminated.
```

PID 1234 (stop)

PID 1235 (ready)

Creating Processes with fork() System Call

```
 1  int main(void) {
 2    int result;
 3    printf("before fork ...\n");
 4    result = fork();
 5    printf("result = %d.\n", result);
 6
 7    if(result == 0) {
 8      printf("I'm the child.\n");
 9      printf("My PID is %d\n", getpid());
10    }
11    else {
12      printf("I'm the parent.\n");
13      printf("My PID is %d\n", getpid());
14    }
15
16    printf("program terminated.\n");
17  }
```

```
$ ./fork_example_2
before fork ...
result = 1235
I'm the parent.
My PID is 1234
program terminated.
result = 0
```

Important

For child, the return value of `fork()` is **0**.

PID 1234 (stop)

PID 1235 (running)

Creating Processes with fork() System Call

```
 1  int main(void) {
 2    int result;
 3    printf("before fork ...\n");
 4    result = fork();
 5    printf("result = %d.\n", result);
 6
 7    if(result == 0) {
 8      printf("I'm the child.\n");
 9      printf("My PID is %d\n", getpid());
10    }
11    else {
12      printf("I'm the parent.\n");
13      printf("My PID is %d\n", getpid());
14    }
15
16    printf("program terminated.\n");
17  }
```

```
$ ./fork_example_2
before fork ...
result = 1235
I'm the parent.
My PID is 1234
program terminated.
result = 0
I'm the child.
My PID is 1235
```

PID 1234 (stop)

PID 1235 (running)

Creating Processes with fork() System Call

fork的返回值：
在parent中返回child的pid；
child中返回0

```
 1  int main(void) {
 2    int result;
 3    printf("before fork ...\n");
 4    result = fork();
 5    printf("result = %d.\n", result);
 6
 7    if(result == 0) {
 8      printf("I'm the child.\n");
 9      printf("My PID is %d\n", getpid());
10    }
11    else {
12      printf("I'm the parent.\n");
13      printf("My PID is %d\n", getpid());
14    }
15
16    printf("program terminated.\n");
17  }
```

```
$ ./fork_example_2
before fork ...
result = 1235
I'm the parent.
My PID is 1234
program terminated.
result = 0
I'm the child.
My PID is 1235
program terminated.
$ _
```

PID 1234 (stop)

PID 1235 (stop)

只有get_ppid(),没有get_cpid()
可能有多个children

fork() System Call

- `fork()` behaves like "cell division".
  - It creates the child process by **cloning** from the parent process, including all user-space data, e.g.,

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 "fd", then the child will also have file "fd" opened automatically.

terminal也是file

fork() System Call

- `fork()` does not clone the following…

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.	Parent.
Running time	Cumulated.	Just created, so should be 0.
[Advanced] File locks	Unchanged.	None.

fork() System Call

父进程运行的是什么程序，子进程也只能运行同一个程序。

系统没法启动别的应用

- If a process can only duplicate itself and always runs the same program, it's not quite meaningful
  - how can we execute other programs?

- **exec()**
  - The **exec*()** system call family.

exec()

- **execl()** – a member of the **exec** system call family (execl, execle, execlp, execv, execve, execvp).

```
int main(void) {

  printf("before execl ...\n");

  execl("/bin/ls", "/bin/ls", NULL);

  printf("after execl ...\n");

  return 0;
}
```

```
$ ./exec_example
before execl ...
```

Arguments of the execl() call

1st argument: the program name, "**/bin/ls**" in the example.
2nd argument: argument[0] to the program.
3rd argument: argument[1] to the program.

exec()

- execl() – a member of the exec system call family (and the family has 6 members).

```
int main(void) {

  printf("before execl ...\n");

  execl("/bin/ls", "/bin/ls", NULL);

  printf("after execl ...\n");

  return 0;
}
```

输出里没有“after execl ...”

```
$./exec_example
before execl ...
exec_example
exec_example.c
```

What is the output?

The same as **the output of running “ls” in the shell.**

exec()

用新的程序替换当前进程的代码和数据段。

进程的 PID 不变。

但原来内存中的程序、指令、数据都会被替换成新程序。

- Example #1: run the command "/bin/ls"

```
execl("/bin/ls", "/bin/ls", NULL);
```

Argument Order	Value in above example	Description
1	"/bin/ls"	The file that the programmer wants to execute.
2	"/bin/ls"	When the process switches to "/bin/ls", this string is the **program argument[0]**.
3	NULL	This states the end of the program argument list.

exec()

- Example #2: run the command "/bin/ls -l"

```
execl("/bin/ls", "/bin/ls", "-l", NULL);
```

Argument Order	Value in above example	Description
1	"/bin/ls"	The file that the programmer wants to execute.
2	"/bin/ls"	When the process switches to "/bin/ls", this string is the **program argument[0]**.
3	"-l"	When the process switches to "/bin/ls", this string is the **program argument[1]**.
4	NULL	This states the end of the program argument list.

exec()

- The exec system call family is not simply a function that "invokes" a command.

```
int main(void) {

  printf("before execl ...\n");

  execl("/bin/ls", "/bin/ls", NULL);

  printf("after execl ...\n");

  return 0;
}
```

WHAT?!
The shell prompt appears!

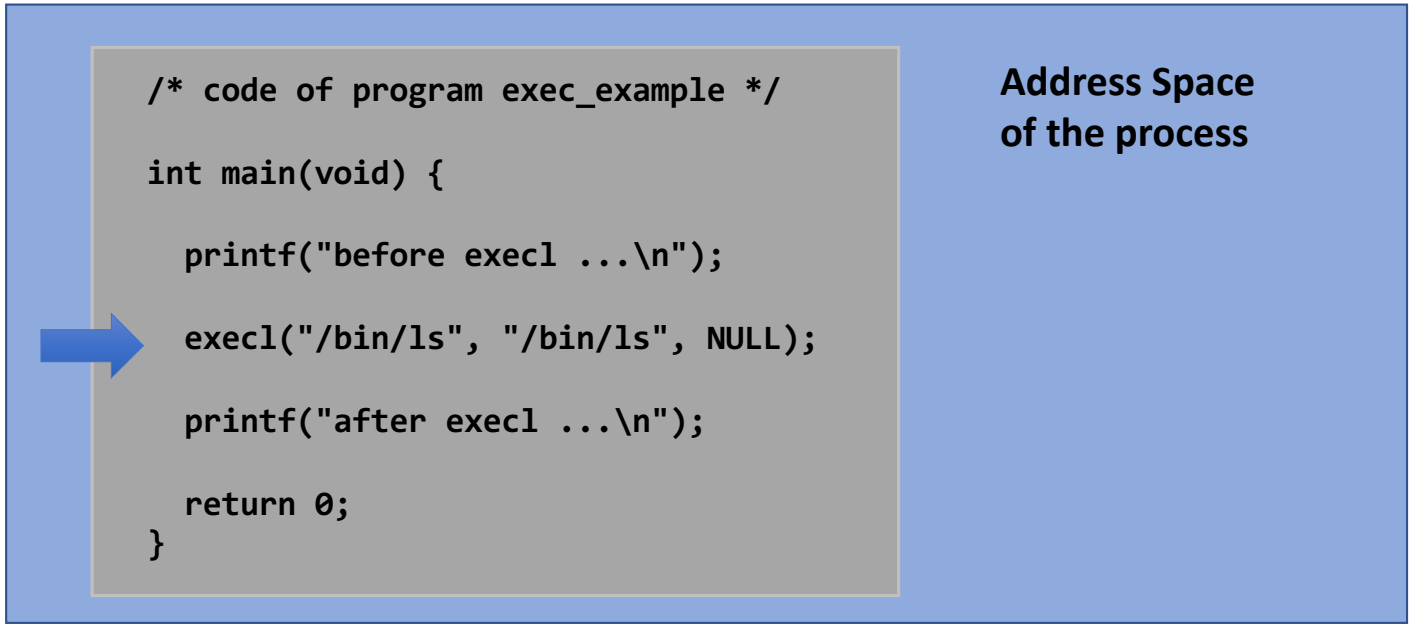
```
$./exec_example
before execl ...
exec_example
exec_example.c
$ _
```

The output says:
(1) The gray code block **is not reached**!
(2) The process is **terminated**!

WHY IS THAT?!

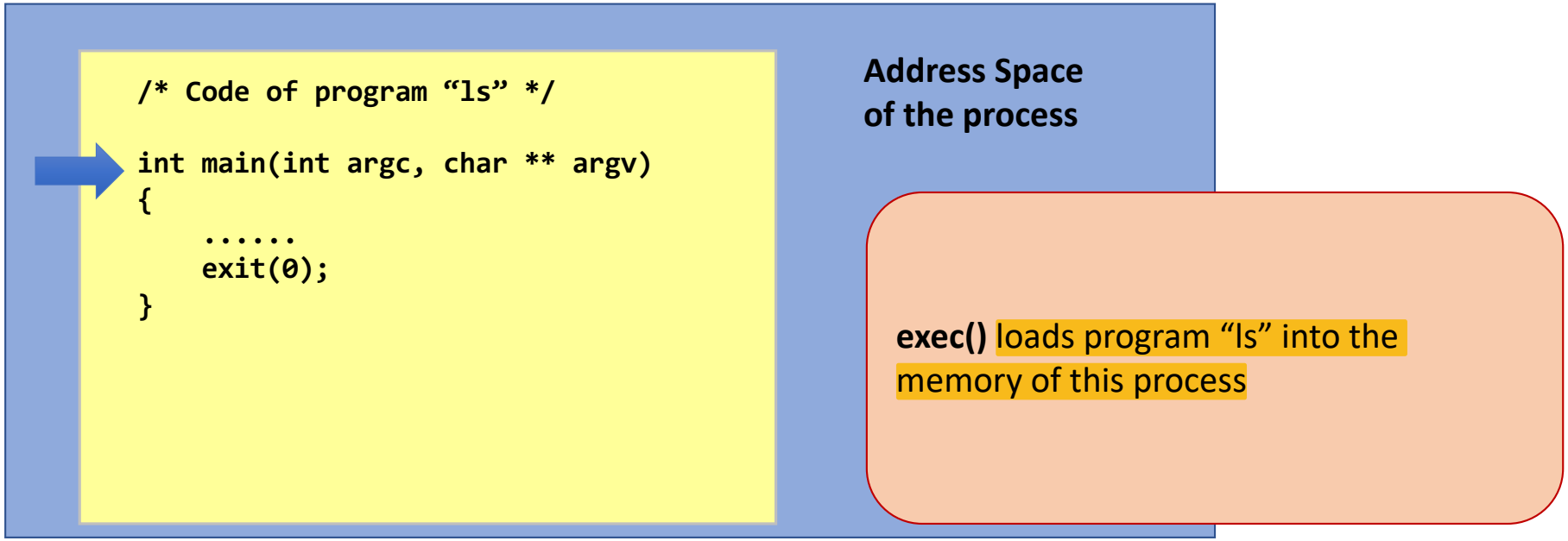
exec()

- The exec system call family is not simply a function that "invokes" a command.



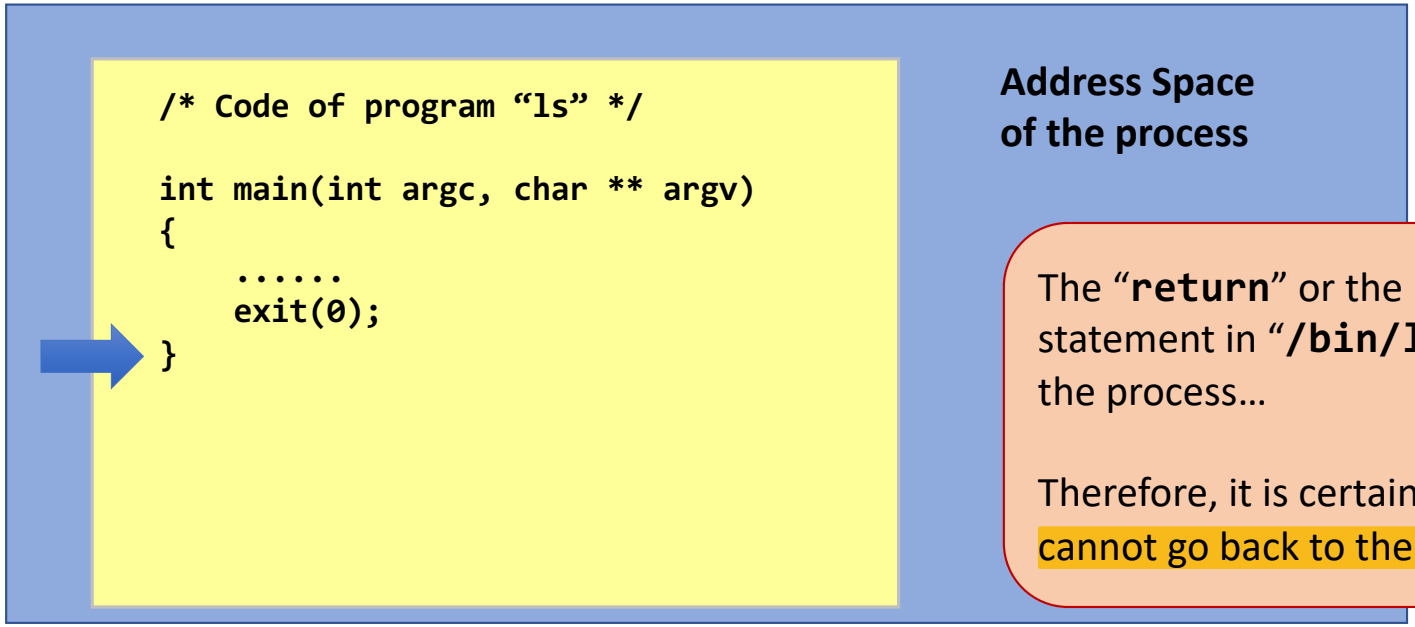
exec()

- The exec system call family is not simply a function that "invokes" a command.



exec()

- The exec system call family is not simply a function that "invokes" a command.



exec() Summary

- The process is changing the code that is executing and never returns to the original code.
  - The last two lines of codes are therefore not executed.
- The process that calls an exec* system call will replace user-space info, e.g.,
  - Program Code
  - Memory: local variables, global variables, and dynamically allocated memory;
  - Register value: e.g., the program counter;
- But, the kernel-space info of that process is preserved, including:
  - PID;
  - Process relationship;
  - etc.

CPU Scheduler and fork()

```
 1  int main(void) {
 2    int result;
 3    printf("before fork ...\n");
 4    result = fork();
 5    printf("result = %d.\n", result);
 6
 7    if(result == 0) {
 8      printf("I'm the child.\n");
 9      printf("My PID is %d\n", getpid());
10    }
11    else {
12      printf("I'm the parent.\n");
13      printf("My PID is %d\n", getpid());
14    }
15
16    printf("program terminated.\n");
17  }
```

Parent return from fork() first

```
$ ./fork_example_2
before fork ...
result = 1235
I'm the parent.
My PID is 1234
program terminated.
result = 0
I'm the child.
My PID is 1235
program terminated.
$ _
```

Child return from fork() first

```
$ ./fork_example_2
before fork ...
result = 0
I'm the child.
My PID is 1235
result = 1235
program terminated.
I'm the parent.
My PID is 1234
program terminated.
$ _
```

并发执行，所以输出顺序不确定

wait(): Sync Parent with Child

一定要让“白发人送黑发人”

```
 1  int main(void) {
 2    int result;
 3    printf("before fork ...\n");
 4    result = fork();
 5    printf("result = %d.\n", result);
 6
 7    if(result == 0) {
 8      printf("I'm the child.\n");
 9      printf("My PID is %d\n", getpid());
10    }
11   else {
12       printf("I'm the parent.\n");
13      wait(NULL);
14      printf("My PID is %d\n", getpid());
15    }
16
17    printf("program terminated.\n");
18  }
```

Parent return from fork() first

```
$ ./fork_example_2
before fork ...
result = 1235
I’m the parent.
result = 0
I’m the child.
My PID is 1235
program terminated.
My PID is 1234
program terminated.
$ _
```

Child return from fork() first

```
$ ./fork_example_2
before fork ...
result = 0
I’m the child.
My PID is 1235
result = 1235
program terminated.
I’m the parent.
My PID is 1234
program terminated.
$ _
```

wait() 系统调用会让 父进程阻塞，直到 子进程结束。

这样就强制保证：子进程先执行完，父进程再继续。

输出顺序就确定了，不再依赖 CPU 调度器。

如果父进程比子进程先结束，子进程就会变成 孤儿进程，被 init 或 systemd 收养。

如果子进程结束了，但父进程没调用 wait()，子进程会变成 僵尸进程 (zombie)，占用 PID 和少量内核资源。

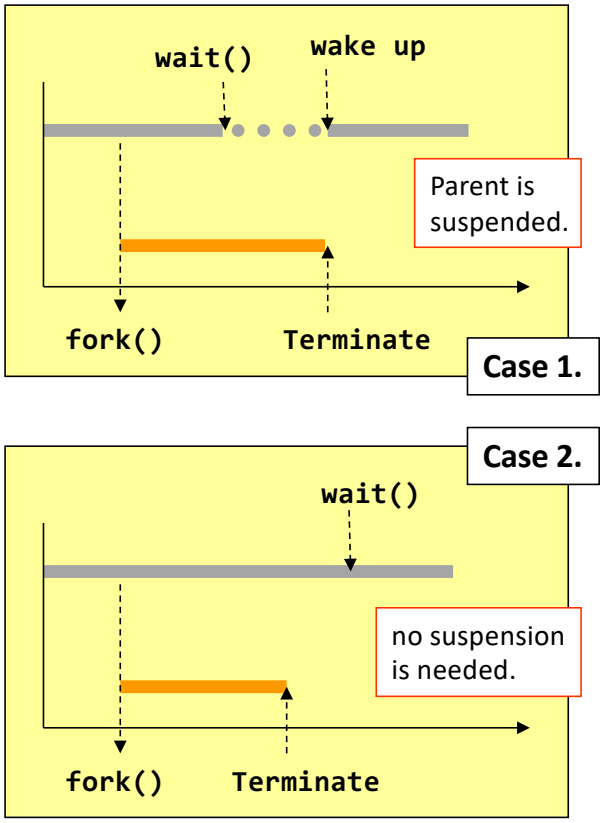
wait() 解决了这两个问题，同时还提供了同步机制（父进程等子进程结束再做后续工作）。

父进程调用 wait() 时，会进入 等待 (waiting) 状态，直到 任意一个子进程结束

wait()

- wait() suspends the calling process to waiting
- wait() returns when
  - one of its child processes changes from running to terminated.

- Return immediately (i.e., does nothing) if
  - It has no children
  - Or a child terminates before the parent calls wait for



wait() v.s. waitpid()

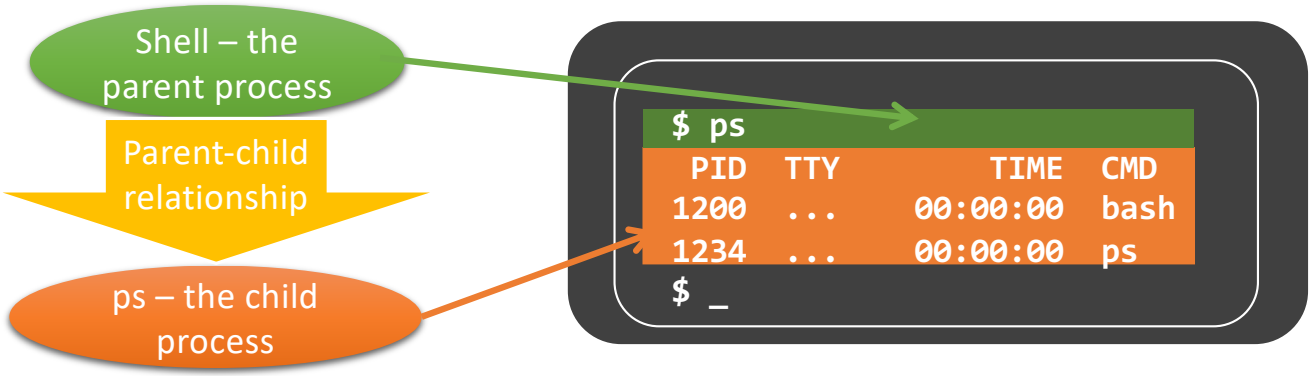
- wait()
  - Wait for any one of the child processes
  - Detect child termination only
- waitpid()
  - Depending on the parameters, waitpid() will wait for a particular child only
  - Depending on the parameters, waitpid() can detect different status changes of the child (resume/stop by a signal)

可以指定 非阻塞模式（父进程不会被卡住，可以继续做别的事）。

可以检测子进程的 不同状态变化（比如收到信号后暂停/恢复），而不仅仅是退出。

Implement Shell with fork(), exec(), and wait()

- A shell is a CLI
  - Bash in linux
  - invokes a function fork() to create a new process
  - Ask the the child process to exec() the target program
  - Use wait() to wait until the child process terminates

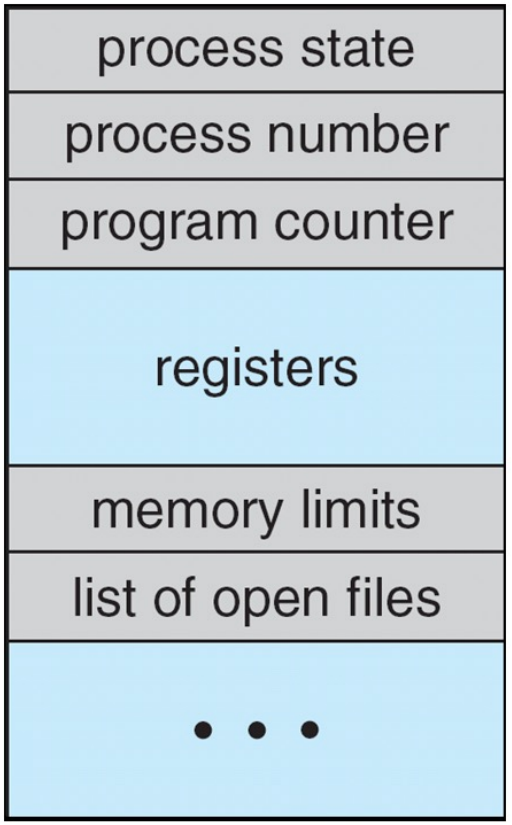


Processes: Kernel View

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
```

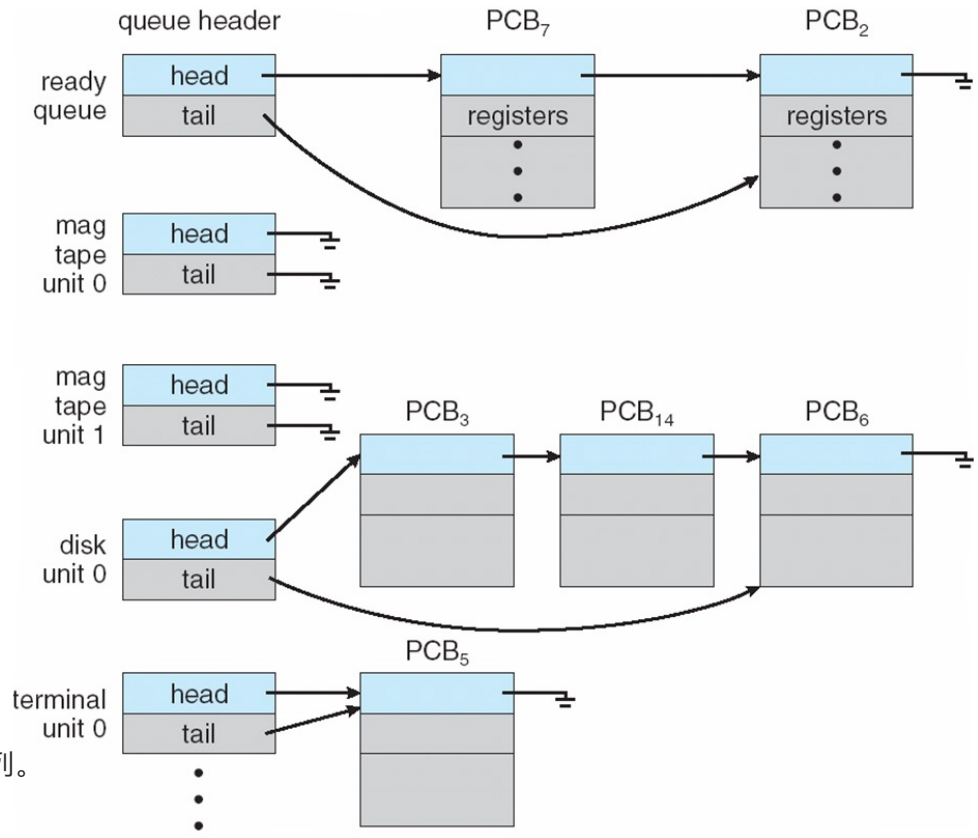
PCB Example: uCore

```
/* 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
};
```

PCB：进程控制块

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) 已经具备运行条件，只是还在等待 CPU
  - Device queue contains processes waiting for I/O events from this device
  - Process may migrate among these queues



每个设备（例如磁带机、磁盘、终端）都有自己的队列。

队列中包含等待该设备 I/O 操作完成的进程。

Threads

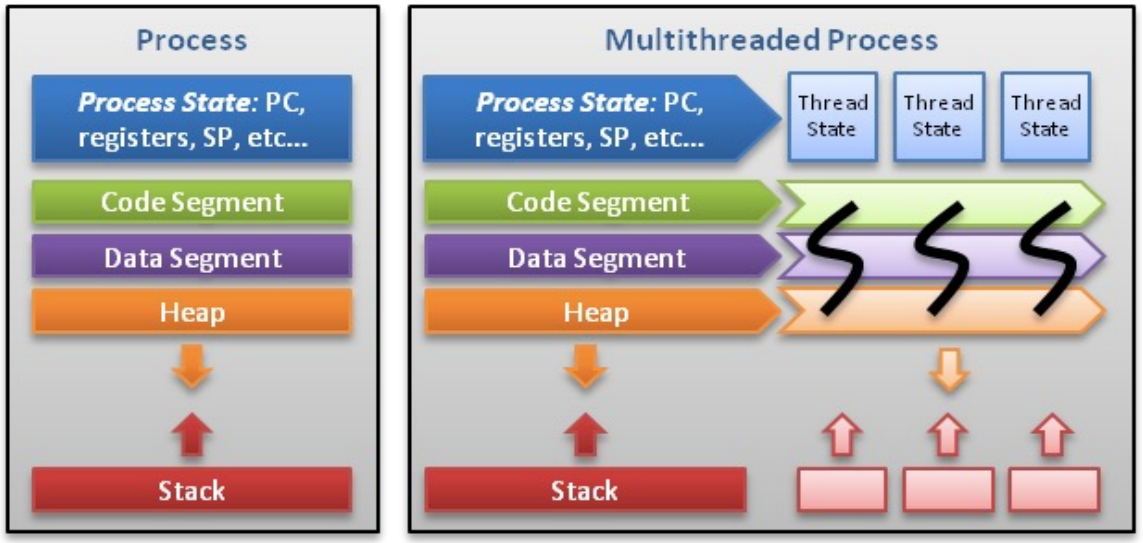
- One process may have more than one threads
  - A single-threaded process performs a single thread of execution
  - A multi-threaded process performs multiple threads of execution "concurrently", thus allowing short response time to user's input even when the main thread is busy

得有空闲的cpu cores，或者time share cpu core

- 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.

© Alfred Park, http://randu.org/tutorials/threads

比较项	单线程进程	多线程进程
执行流数量	1 个	多个
代码段 (Code)	独立	共享
数据段 (Data)	独立	共享
堆 (Heap)	独立	共享
栈 (Stack)	独立	每个线程独立
寄存器、PC、SP	独立	每个线程独立
通信方式	进程间通信（IPC），复杂	共享内存，简单快速

一个进程内部可以有多个线程，多个线程共享该进程的资源（代码段、数据段、堆），但每个线程有自己独立的栈和寄存器状态。

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

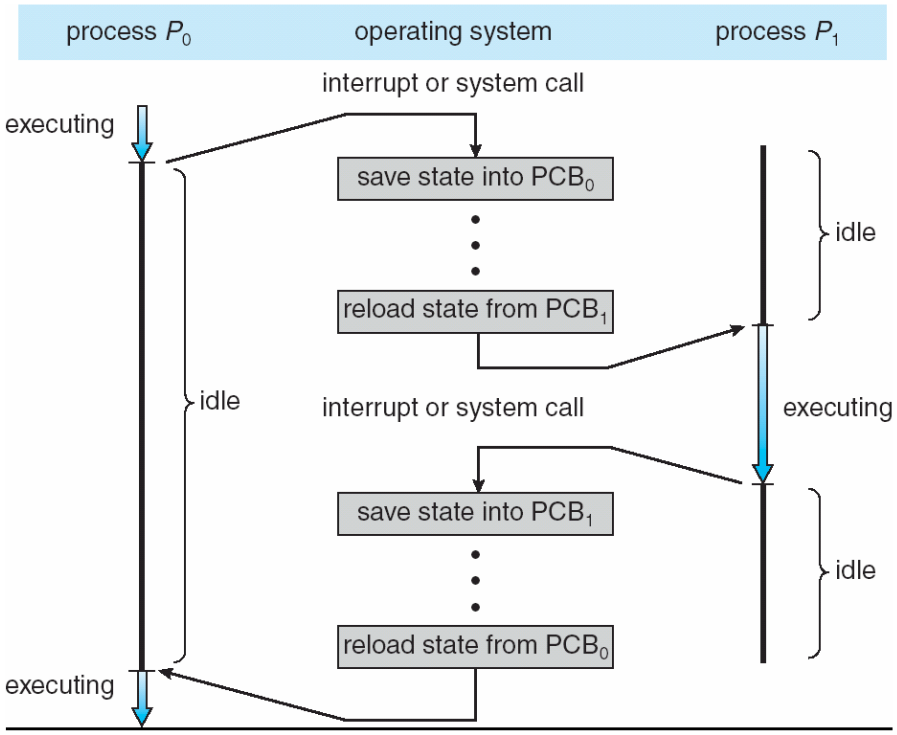
执行 Context Switch（上下文切换）

- 保存当前进程的状态（寄存器、程序计数器等）到其 PCB。
- 从下一个进程的 PCB 中恢复其状态。
- 这样 CPU 就开始执行新的进程了

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);
    }
}
```

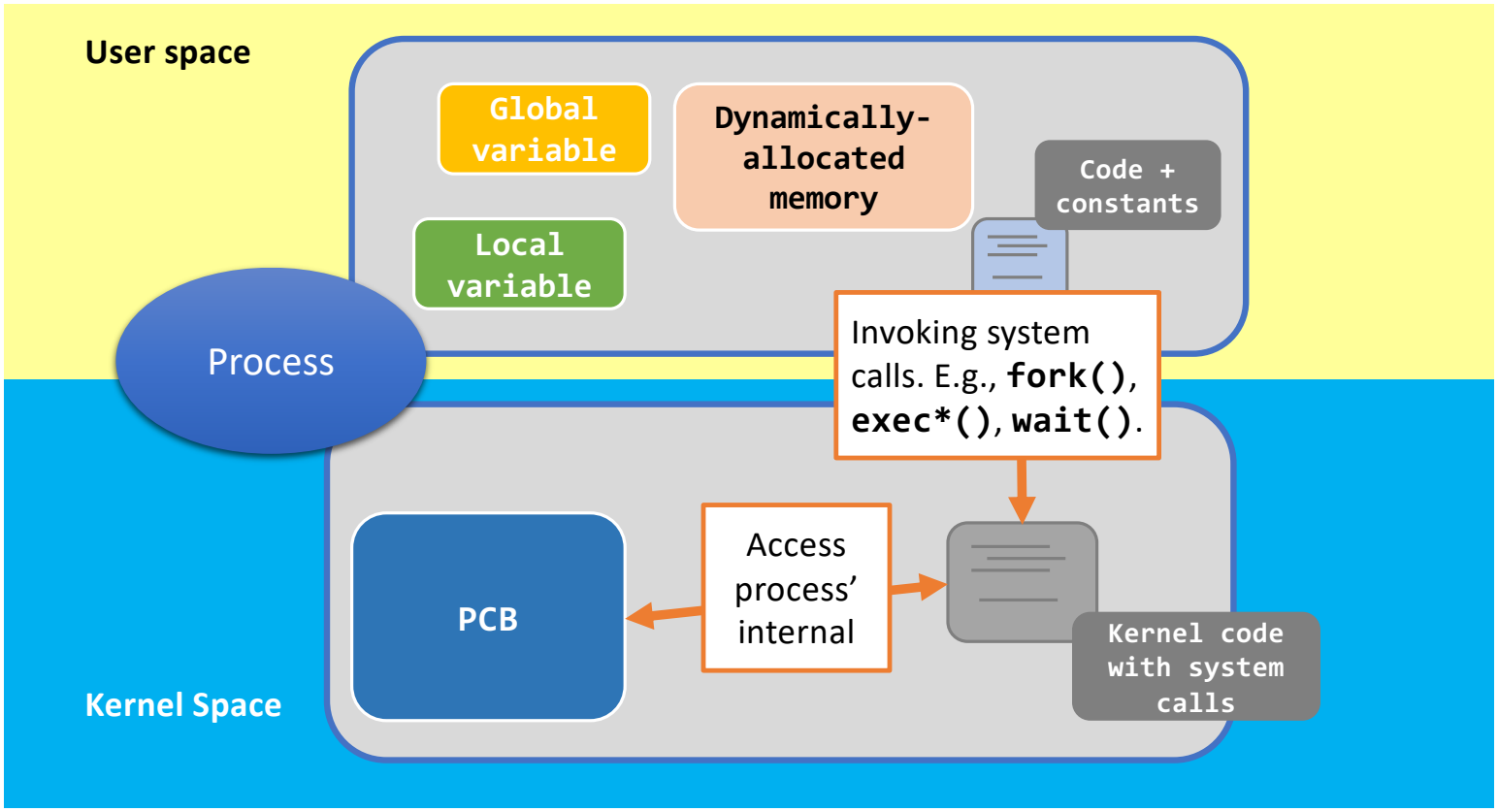
Context Switch: uCore (Cont'd)

```
/* 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)
```

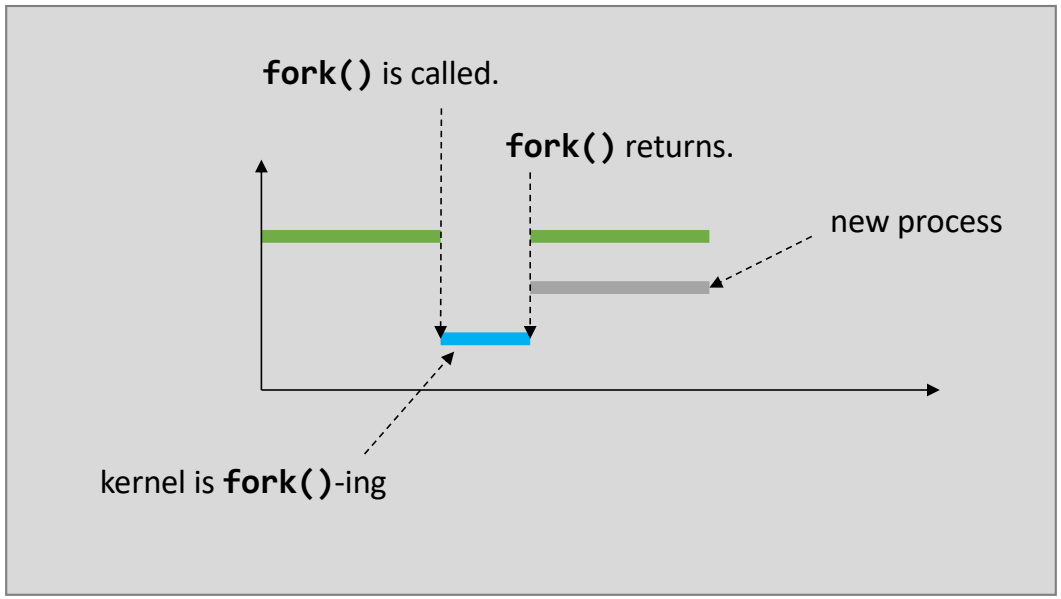
```
    # restore to's registers
    LOAD ra, 0*REGBYTES(a1)
    LOAD sp, 1*REGBYTES(a1)
    LOAD s0, 2*REGBYTES(a1)
    LOAD s1, 3*REGBYTES(a1)
    LOAD s2, 4*REGBYTES(a1)
    LOAD s3, 5*REGBYTES(a1)
    LOAD s4, 6*REGBYTES(a1)
    LOAD s5, 7*REGBYTES(a1)
    LOAD s6, 8*REGBYTES(a1)
    LOAD s7, 9*REGBYTES(a1)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

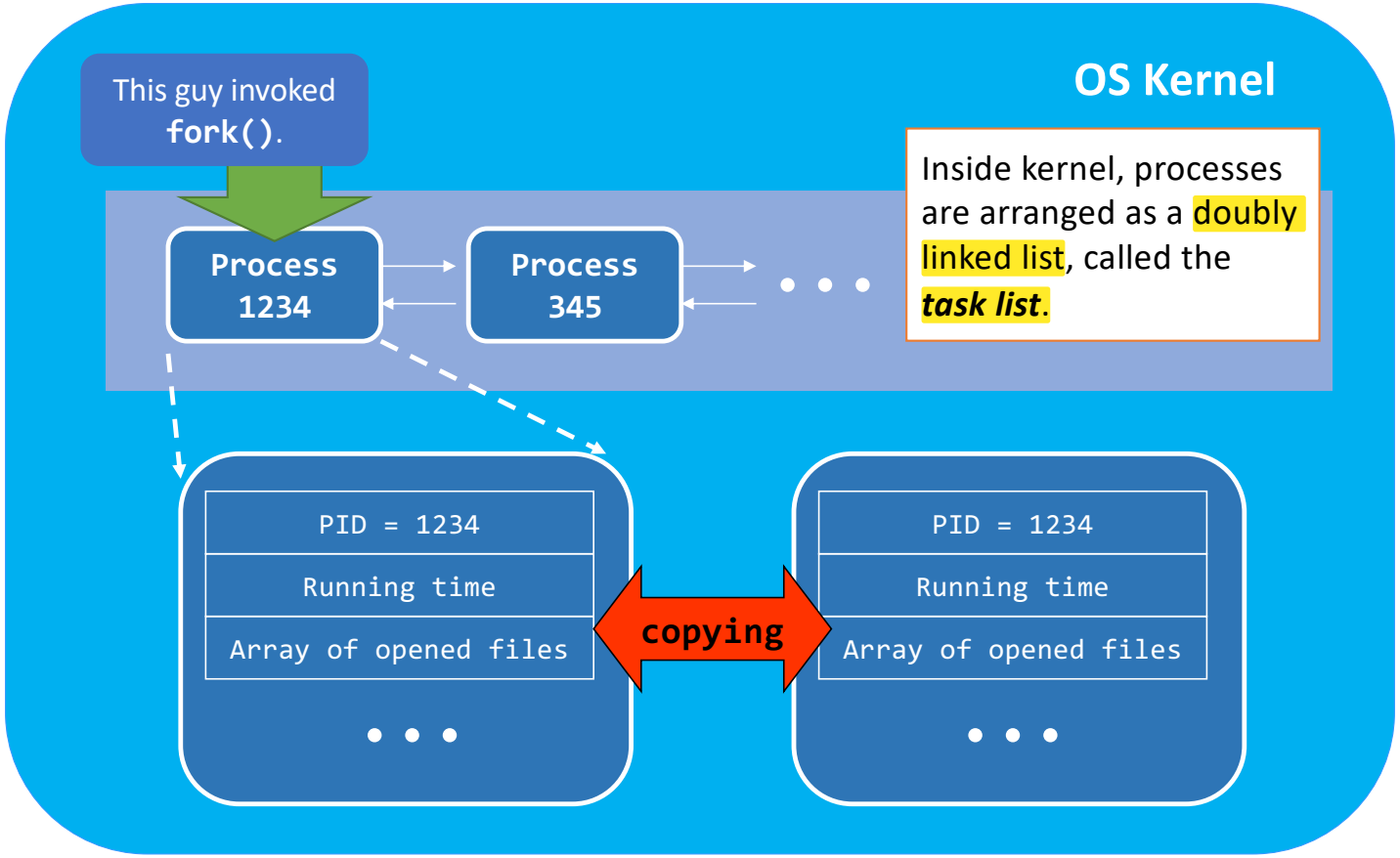
Recall: fork(), exec(), and wait()



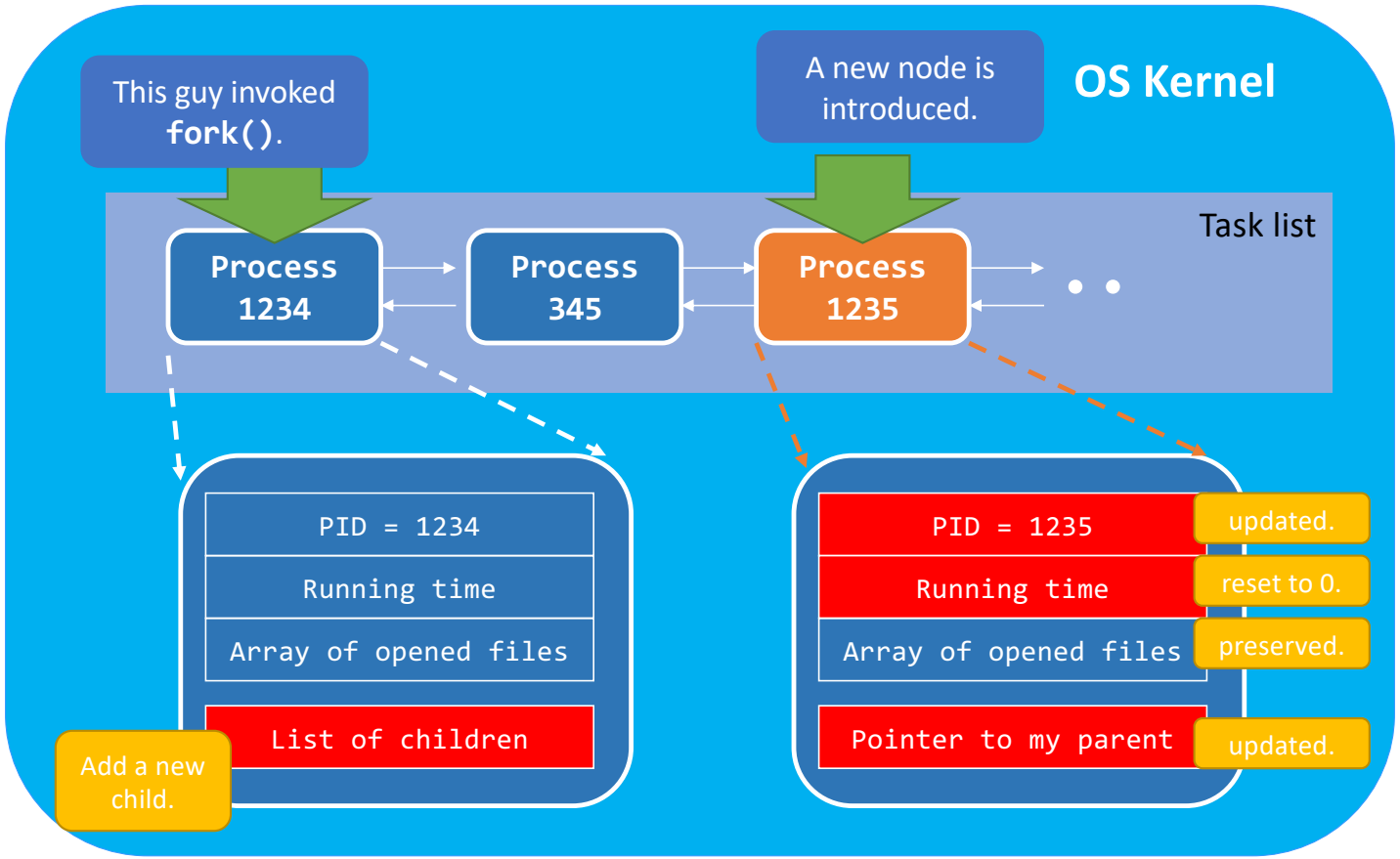
Fork() in User Mode



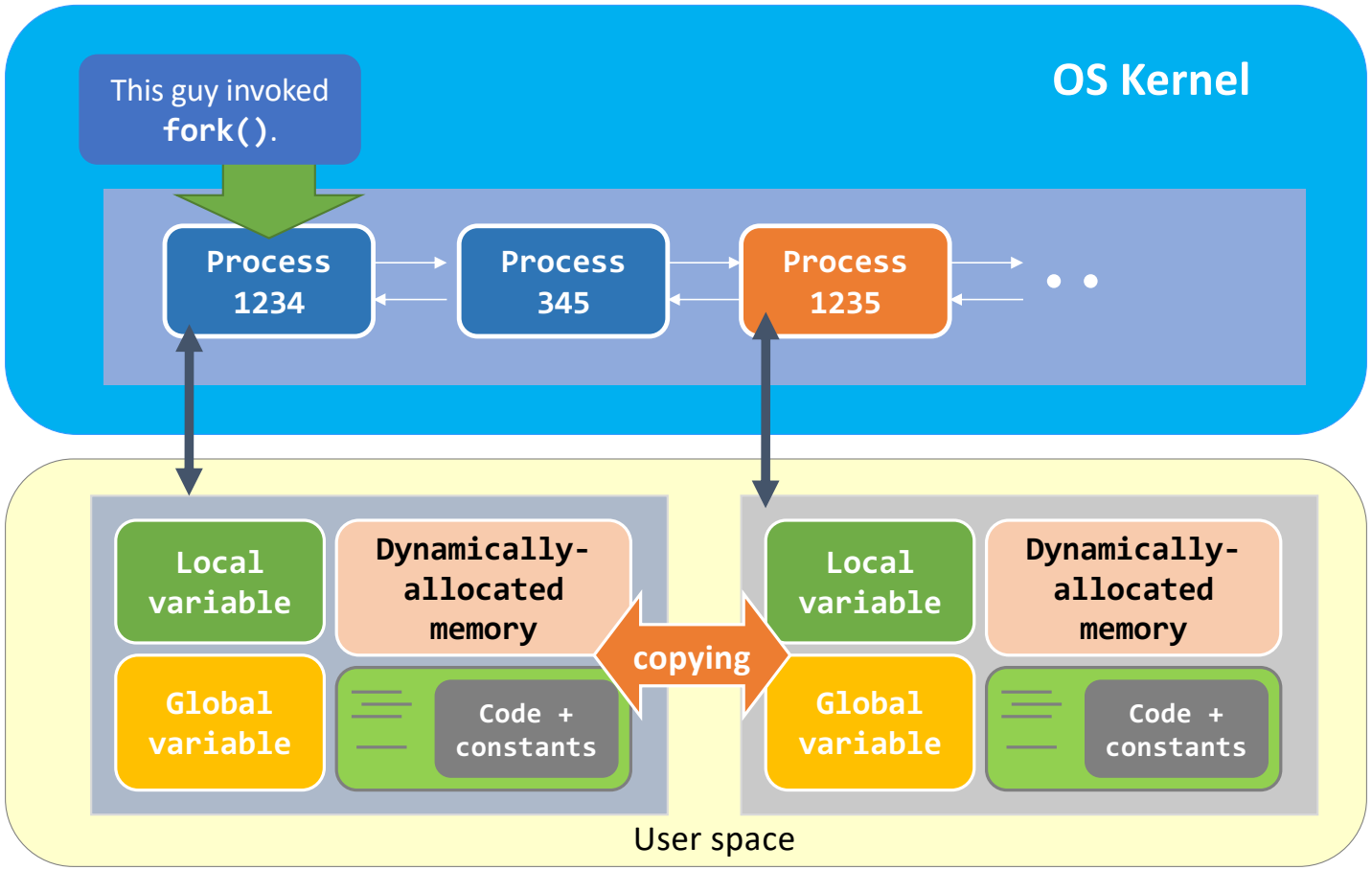
fork(): Kernel View



fork(): Kernel View

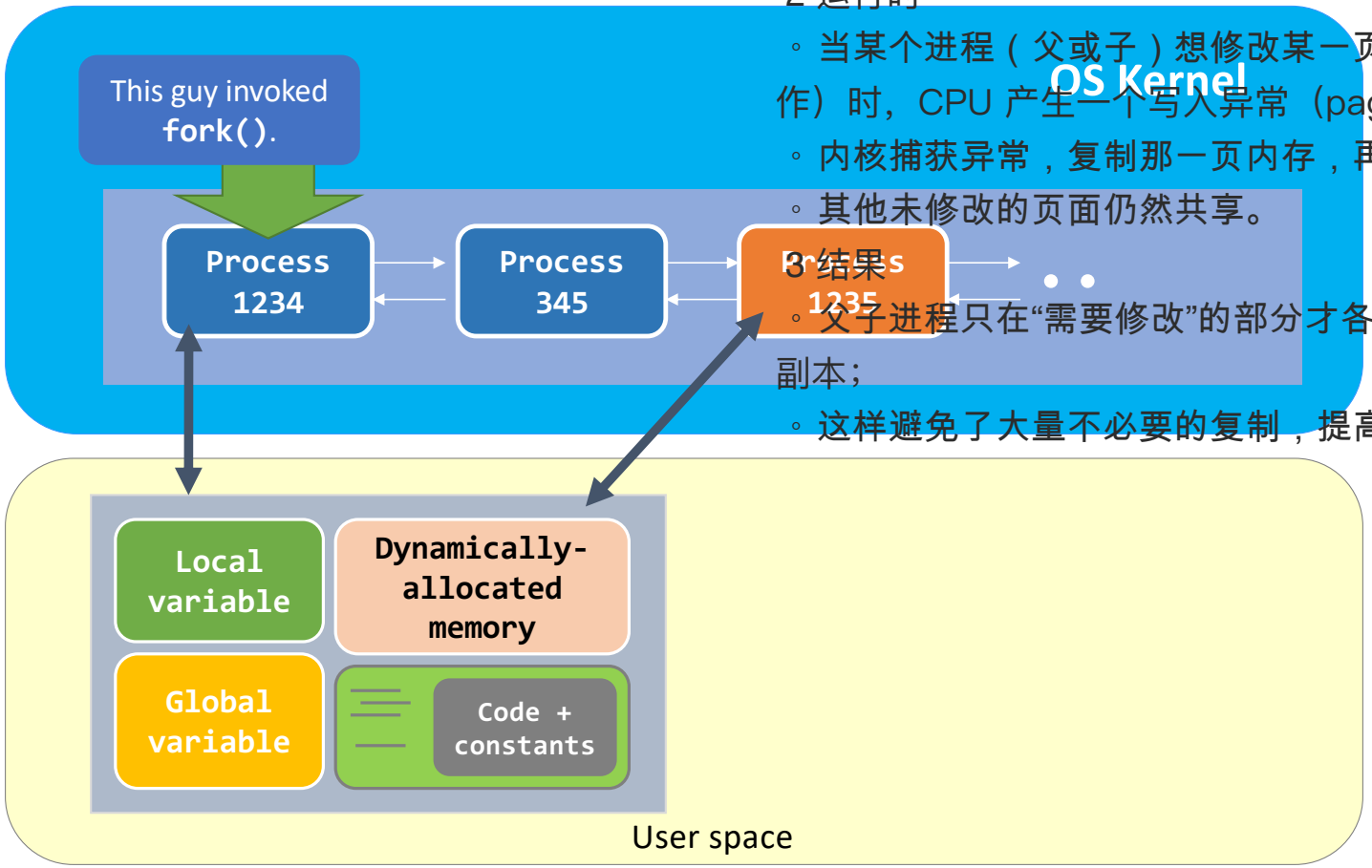


Case 1: Duplicate Address Space



在早期或简单的操作系统中，fork() 会完整地复制父进程的整个用户空间内存。
也就是说，父进程在内存中的所有数据（代码、变量、堆、栈）都会被复制到子进程

Case 2: Copy on Write



具体工作机制：

1 fork() 时

- 子进程的页表复制父进程的页表；
- 父子进程都指向相同的物理内存页；
- 所有页标记为 “只读（read-only）”。

2 运行时

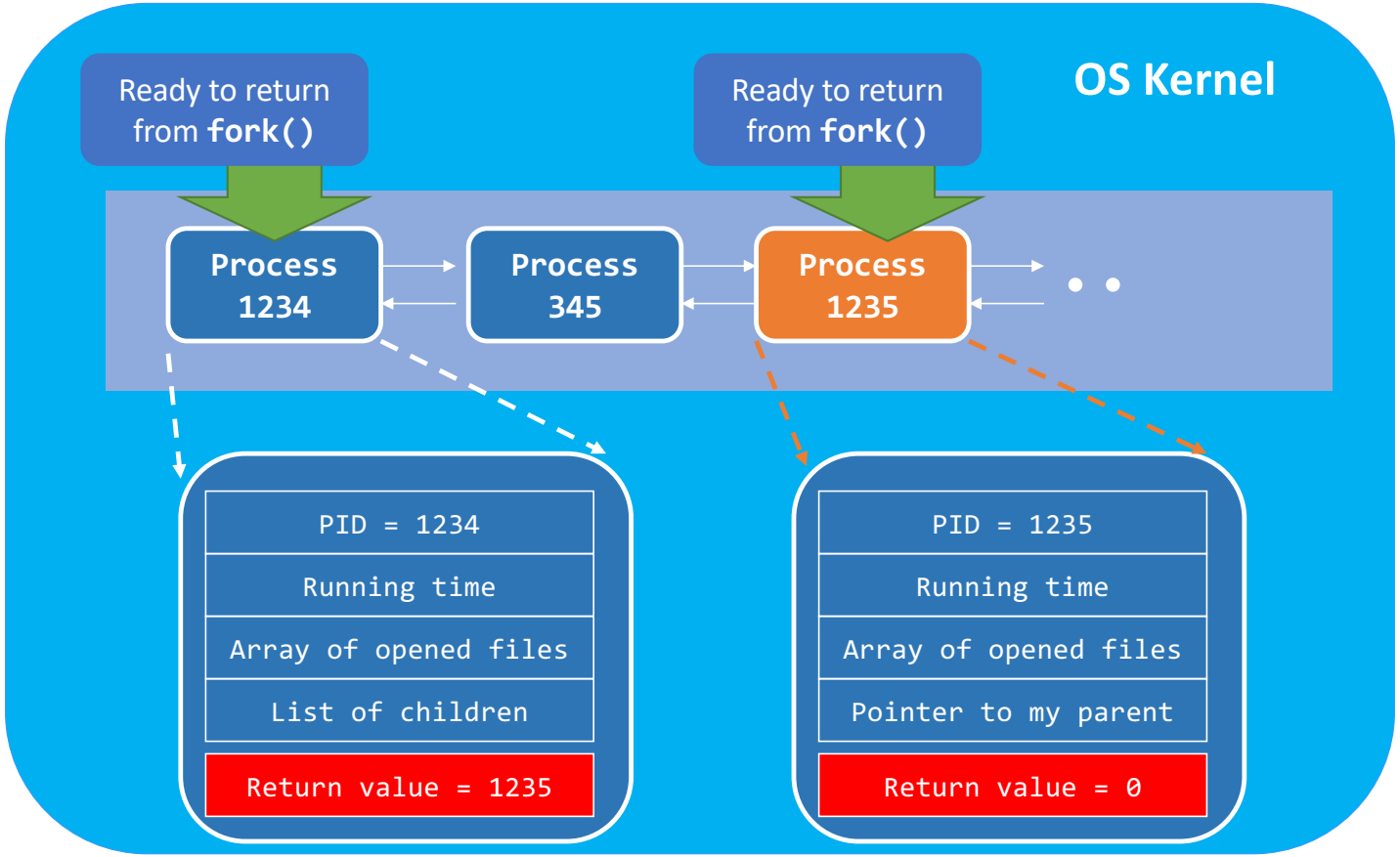
- 当某个进程（父或子）想修改某一页内存（写操作）时，CPU 产生一个写入异常（page fault）；
- 内核捕获异常，复制那一页内存，再允许写入；
- 其他未修改的页面仍然共享。

3 结果

- 父子进程只在“需要修改”的部分才各自拥有独立副本；
- 这样避免了大量不必要的复制，提高性能。

现代操作系统（如 Linux）采用 Copy–on–Write（COW） 技术来优化 fork()。
它不会立即复制父进程的所有内存，而是让父子进程共享同一块物理内存，直到其中一方试图修改数据时，才复制对应的那一页。

fork(): Kernel View



每个进程的 PCB（Process Control Block）中都有一个“文件描述符表 (File Descriptor Tab

- 它是一个数组，保存所有打开文件的引用；

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.

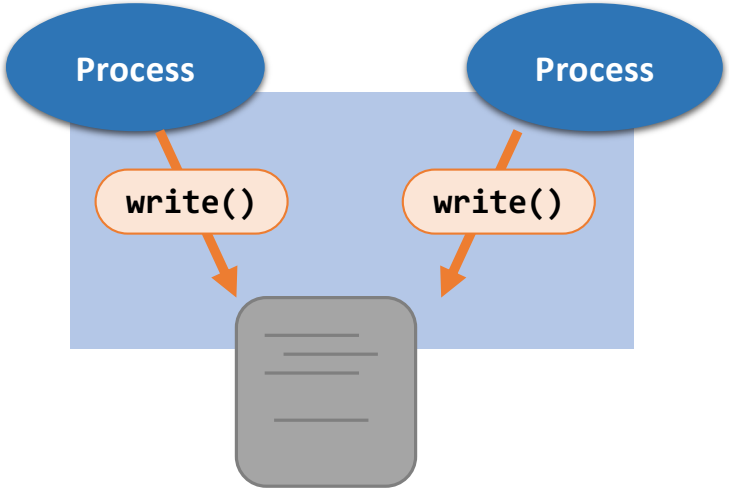
- That's why a parent process shares the same terminal output stream as the child process.

当调用 fork() 时：

- 子进程会复制父进程的文件描述符表；
- 但这只是指针复制，父子都指向同一个内核文件对象；
- 因此，它们共享：
  - 文件位置指针（file offset）
  - 文件状态（如读写模式、锁等）

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!

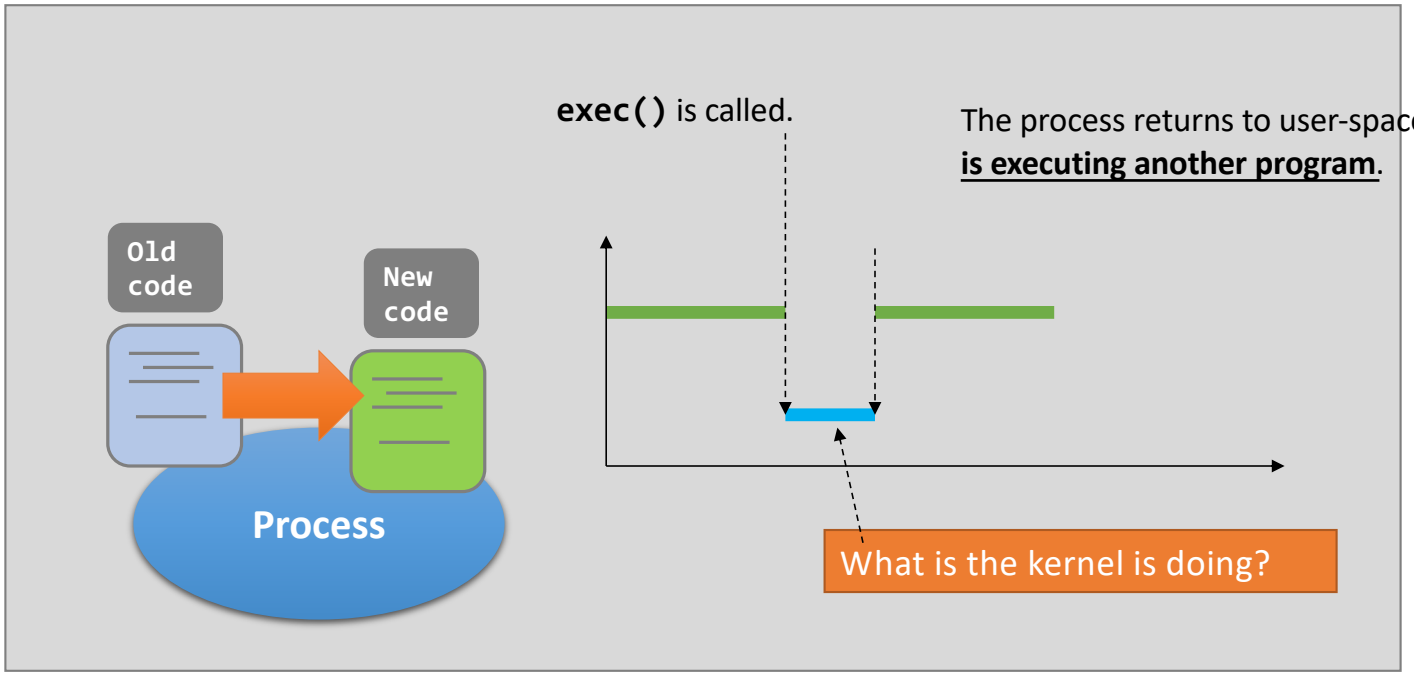
我们将观察到两个进程的输出混合在一起，
因为它们写入的是同一个文件流，共享同一个文件位置指针 (file offset)。

并且写入的顺序是不确定的

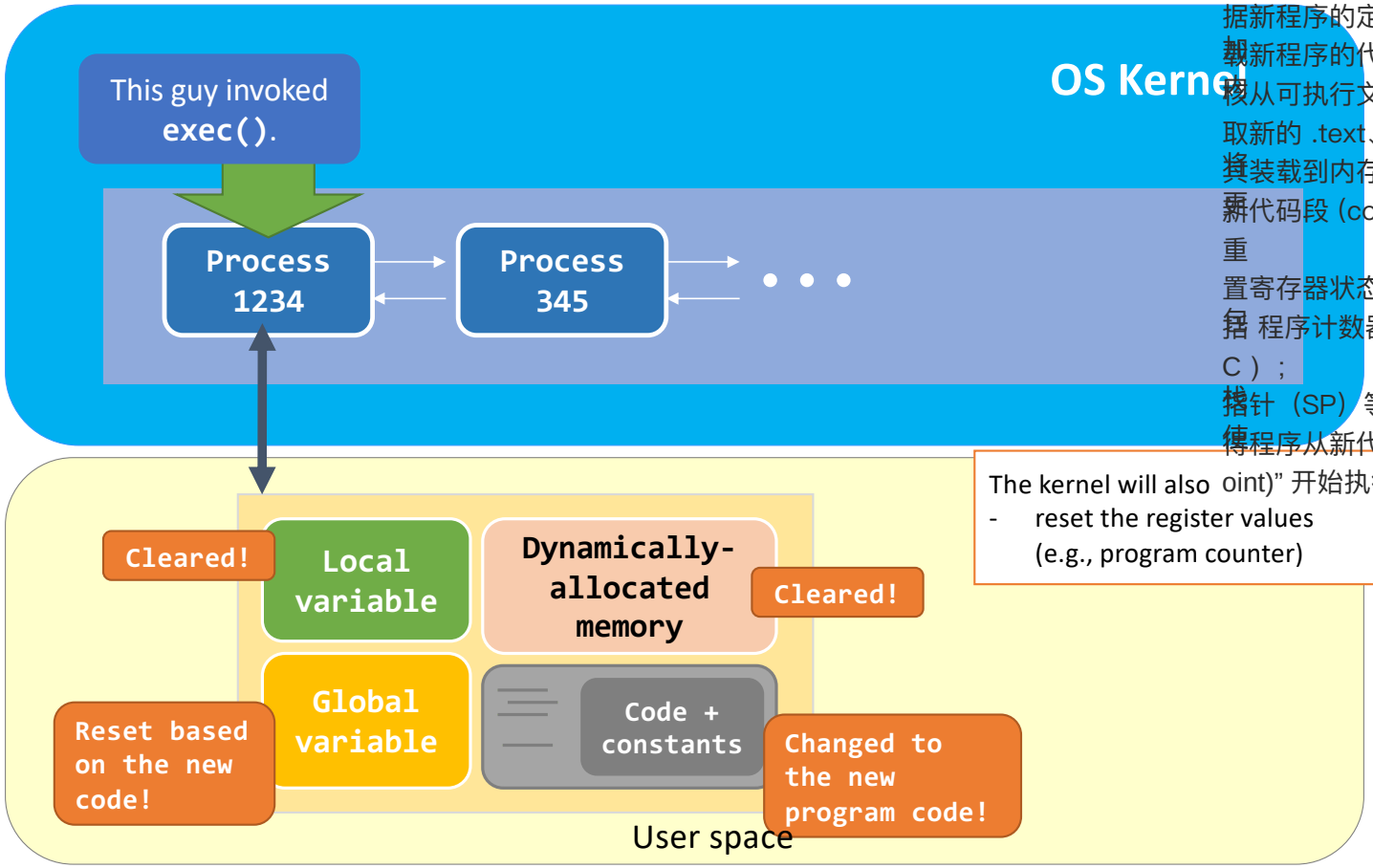
前面提过，在执行 fork() 后，父子进程会共享同一个打开文件表 (file descriptor table)。
这意味着它们都指向同一个内核文件对象（同一个文件描述符结构）。

所以当两个进程同时调用 write() 时，它们实际上在操作同一个文件

exec() in the User Mode



exec(): Kernel View



The kernel will also
- reset the register values (e.g., program counter)

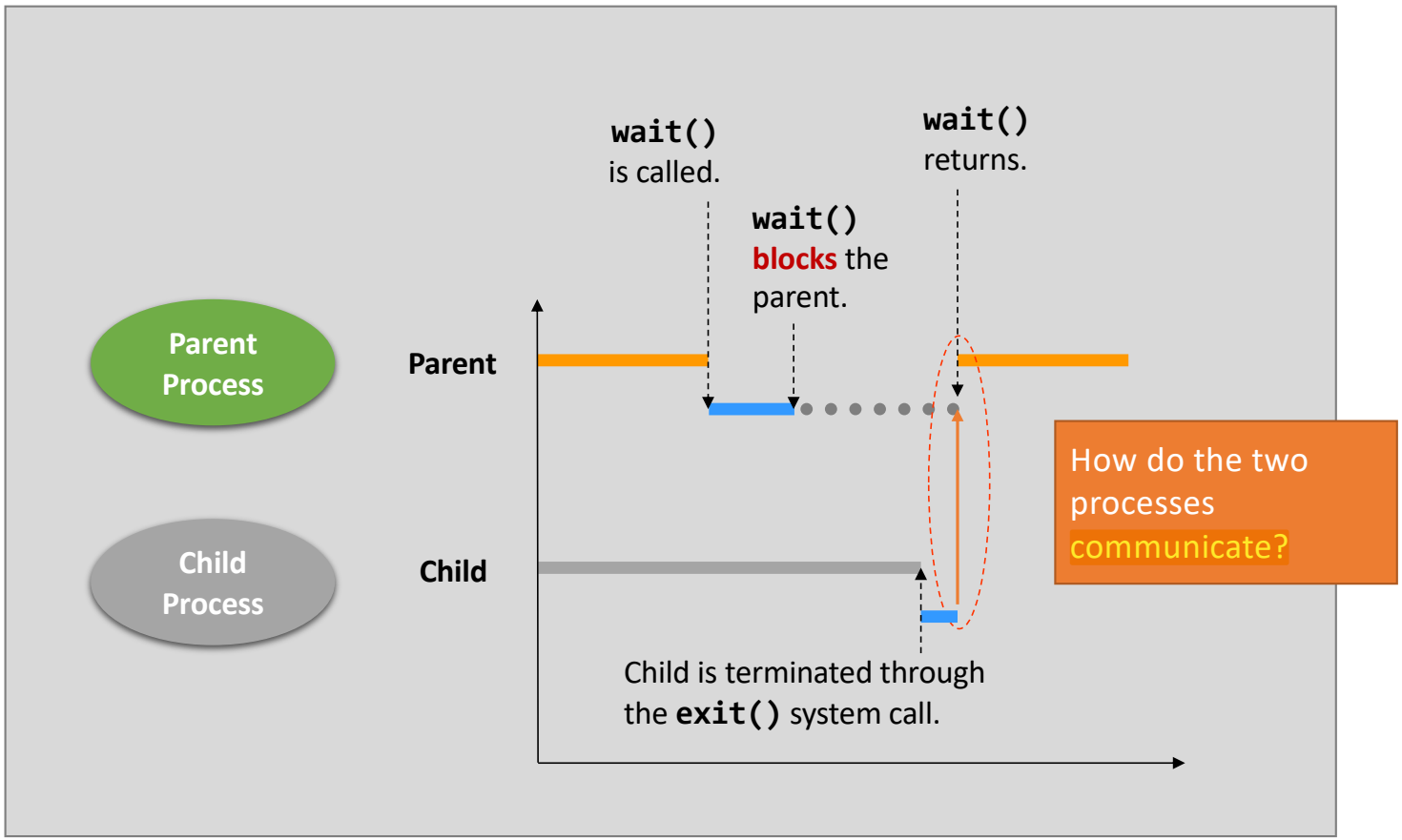
在图中（黄色部分表示用户空间）：
清空原有的用户态内存结构：
局部变量（local variables） → 清除；
动态分配的内存（heap） → 清除；
栈（stack） → 清除；
全局变量（global variables） → 根据新程序的定义重新设置。
加载新程序的代码：
内核从可执行文件（如 ELF 文件）中读取新的 .text、.data 段；
将其装载到内存；
更新代码段 (code + constants)。
重置寄存器状态：
包括 程序计数器（Program Counter, PC）；
栈指针（SP）等；
使得程序从新代码的“入口点 (entry point)” 开始执行。

进程号 (PID) 保持不变；

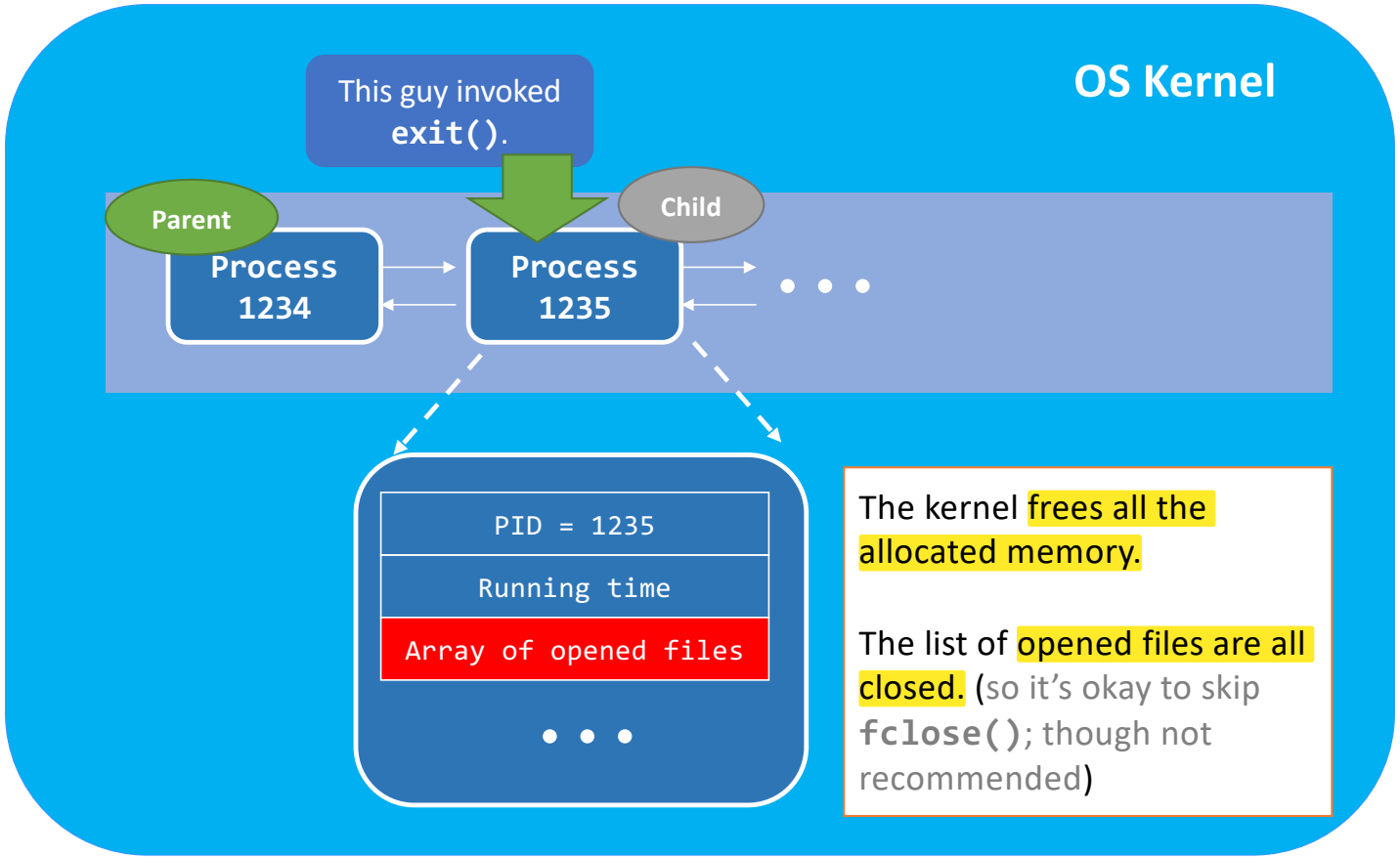
打开的文件描述符 (open files) 仍然保留；

但进程的代码、数据、堆栈都被新程序完全替换。

wait() and exit()

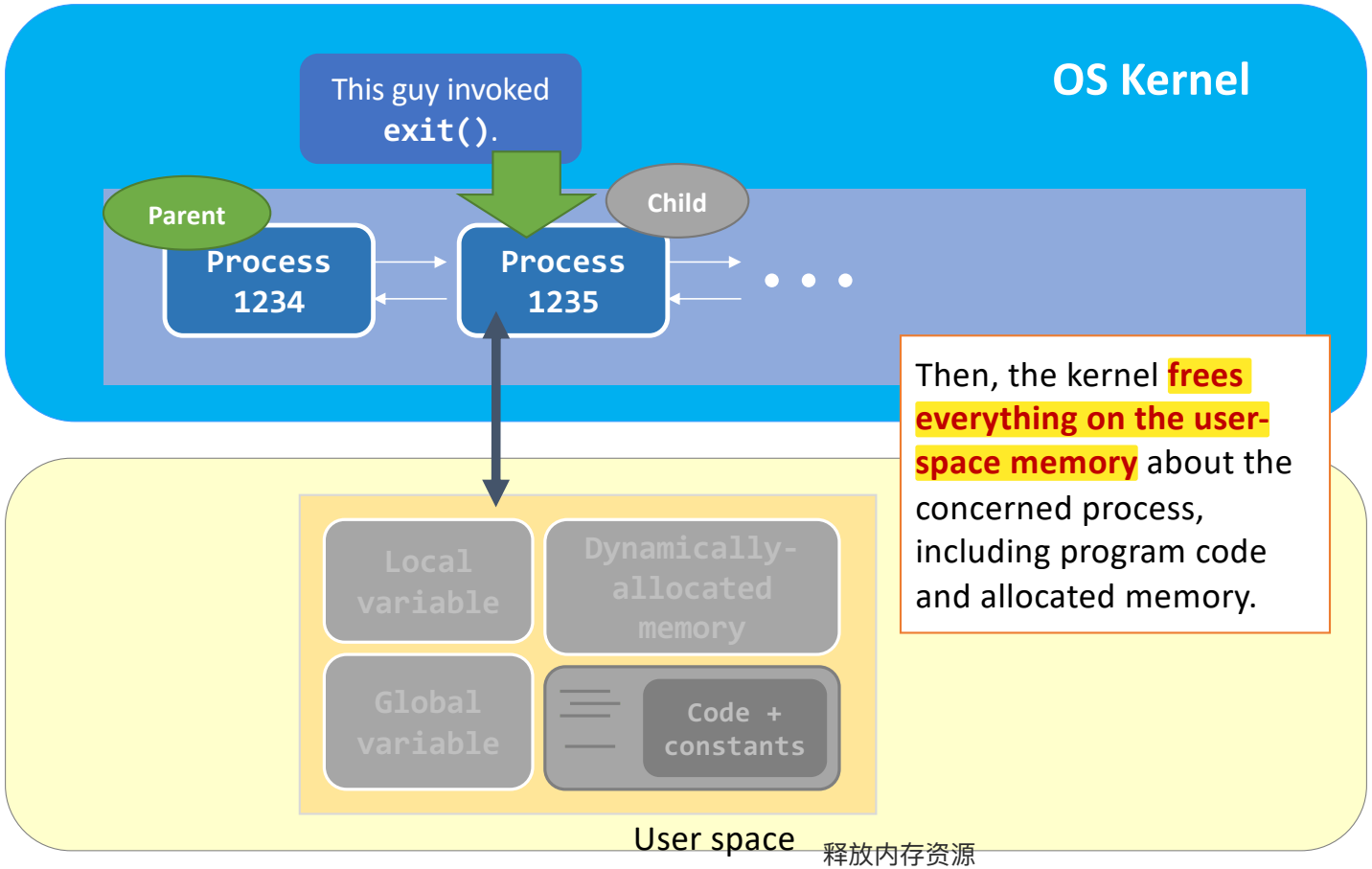


exit(): Kernel View



关闭所有打开的文件：
内核会遍历进程控制块（PCB）中的“文件描述符表 (Array of opened files)”；
逐个调用文件关闭函数（类似于 fclose() 的底层实现）；
文件引用计数（file reference count）减少；
如果某个文件没有其他进程打开，就会真正关闭。

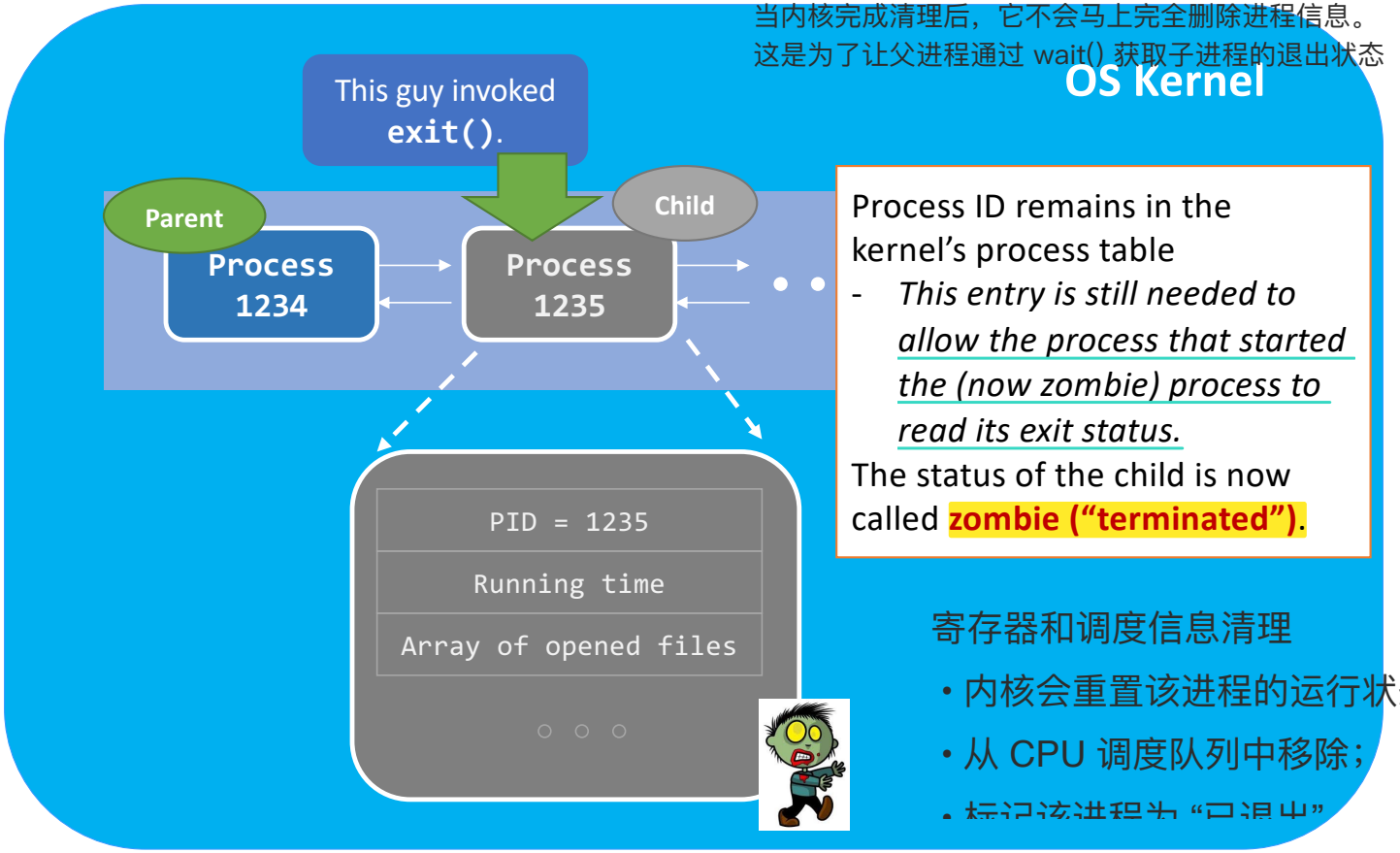
exit(): Kernel View



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

释放内存资源
内核回收该进程分配的所有用户态内存：
栈 (stack)
堆 (heap)
全局数据段 (global/data)
代码段 (text/code)
这些都属于进程的“用户空间 (User Space)”，由内核负责释放。

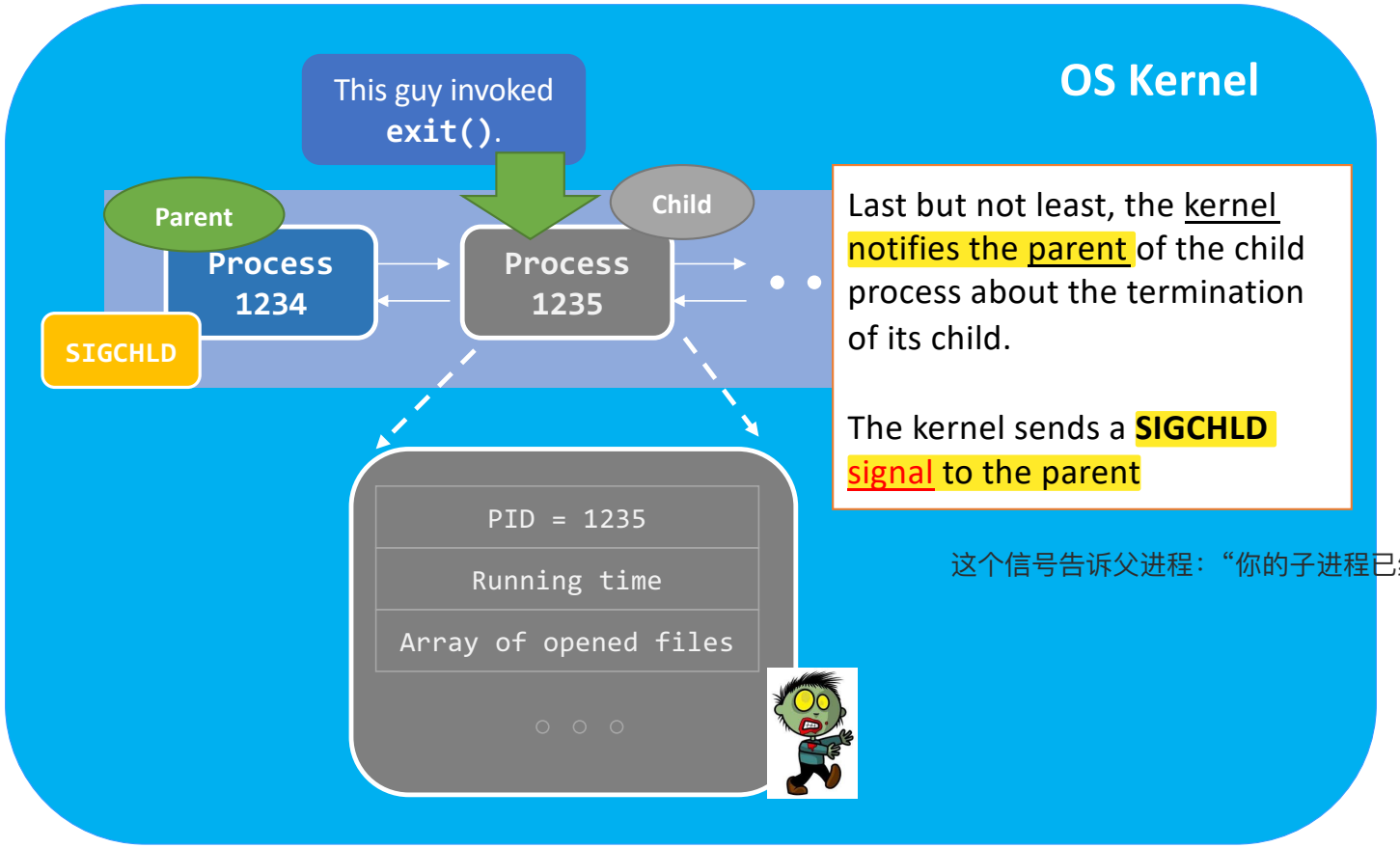
exit(): Kernel View



僵尸进程并没有在运行，它只是一个“死亡的壳”：

没有占用 CPU；
内存和文件都已释放；
只是留下一条进程表记录，等待父进程来“收尸”。

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

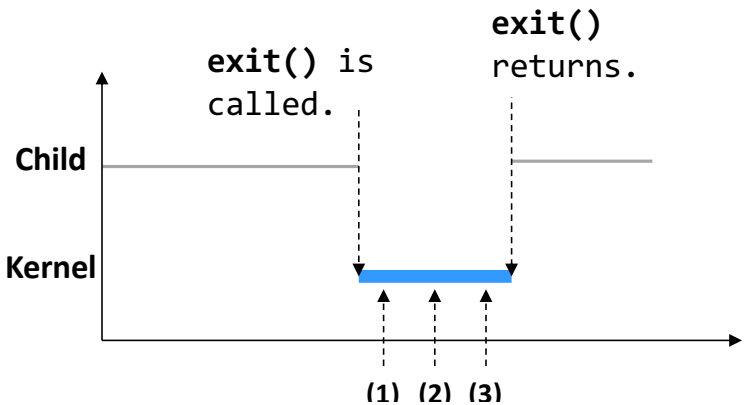
这个信号告诉父进程：“你的子进程已经结束了。”

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.



SIGCHLD 信号的作用

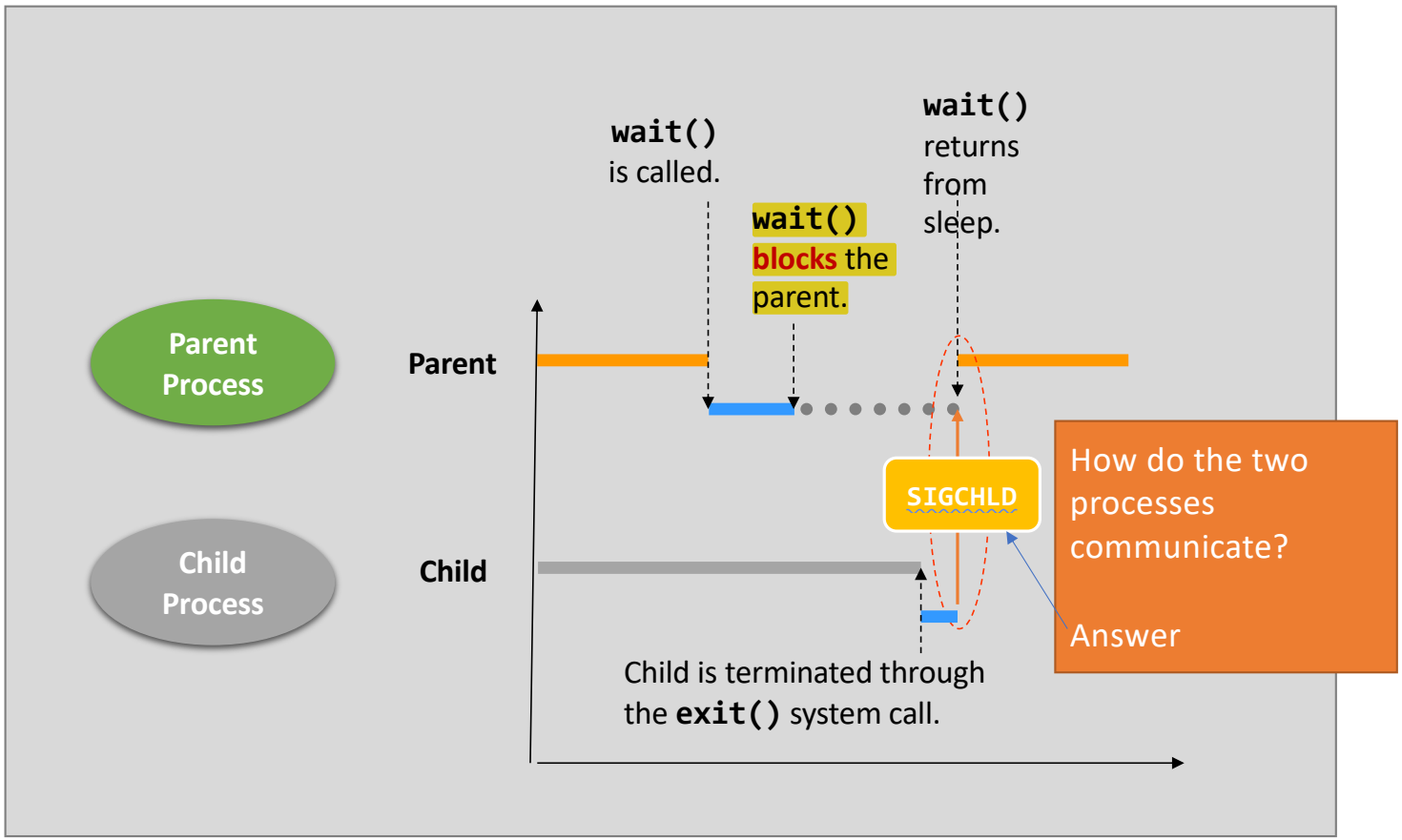
父进程收到信号后，可以选择：

调用 wait() 或 waitpid() 来回收子进程；

忽略信号（如果父进程不关心子进程状态）；

如果父进程没有及时调用 wait()，子进程就会一直保持在“僵尸”状态。

wait() and exit()



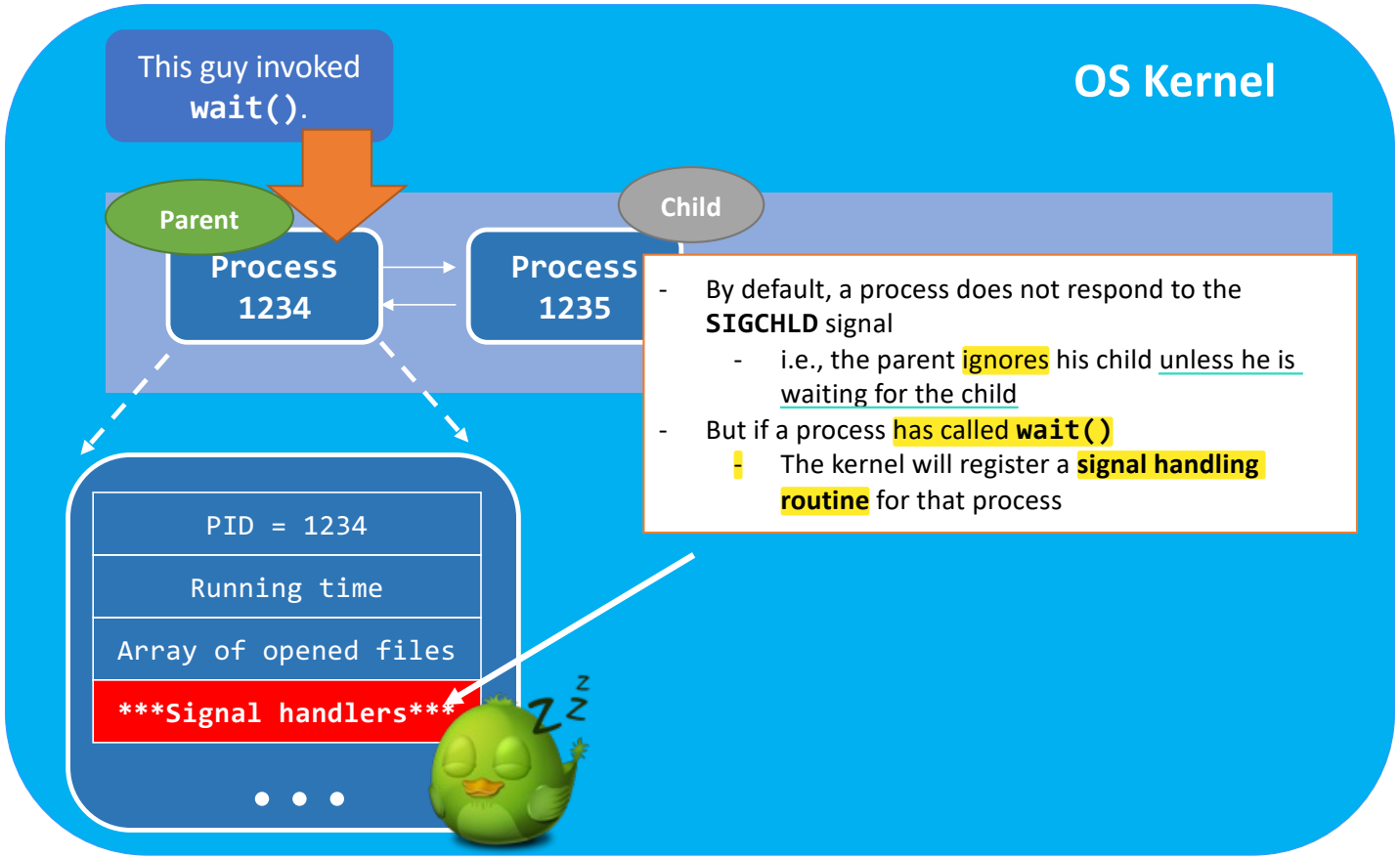
wait() Kernel View

当父进程调用 wait()（或 waitpid()）时，它会：

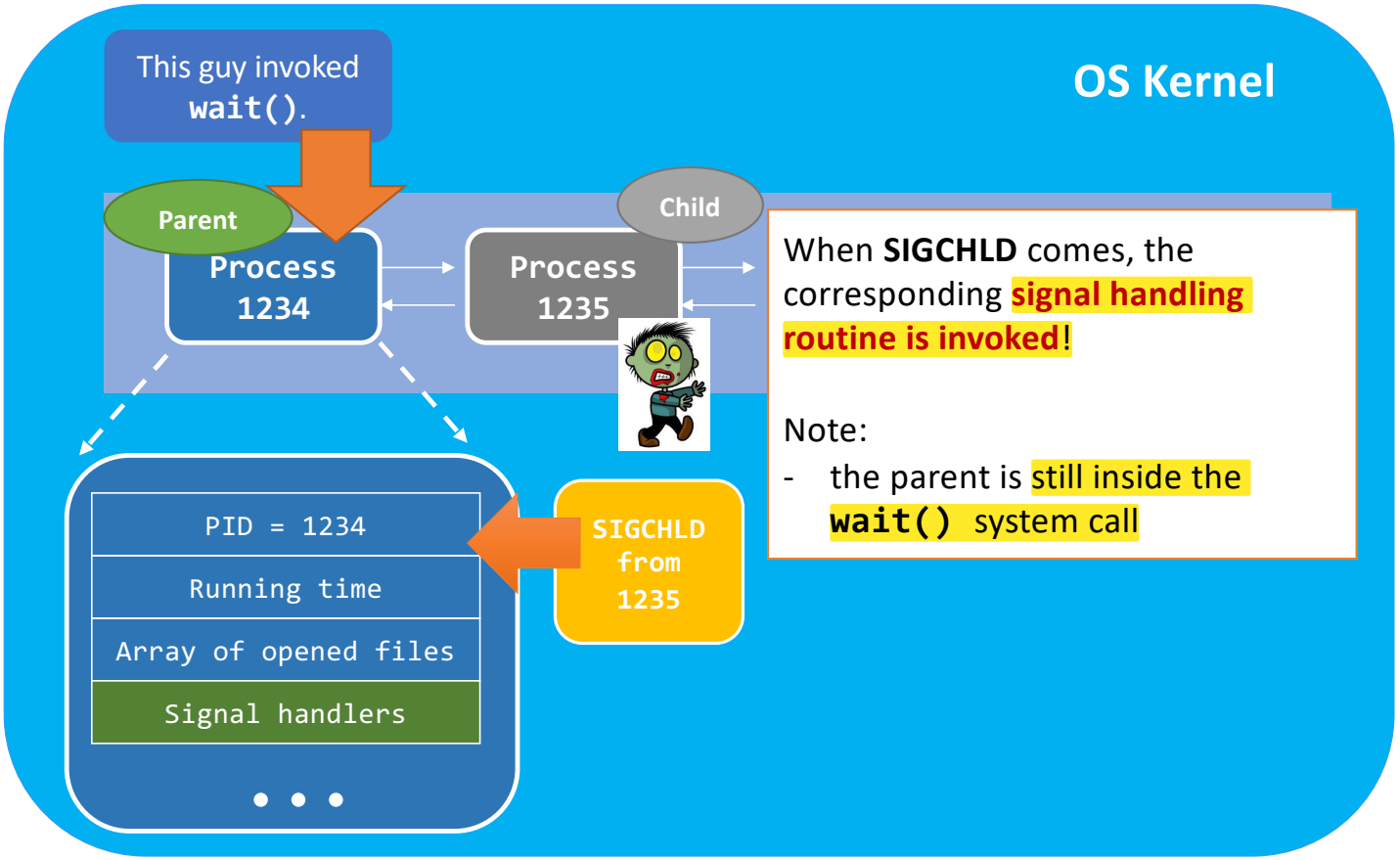
1 进入内核态；2

挂起自己（block）等待子进程退出；3 一旦

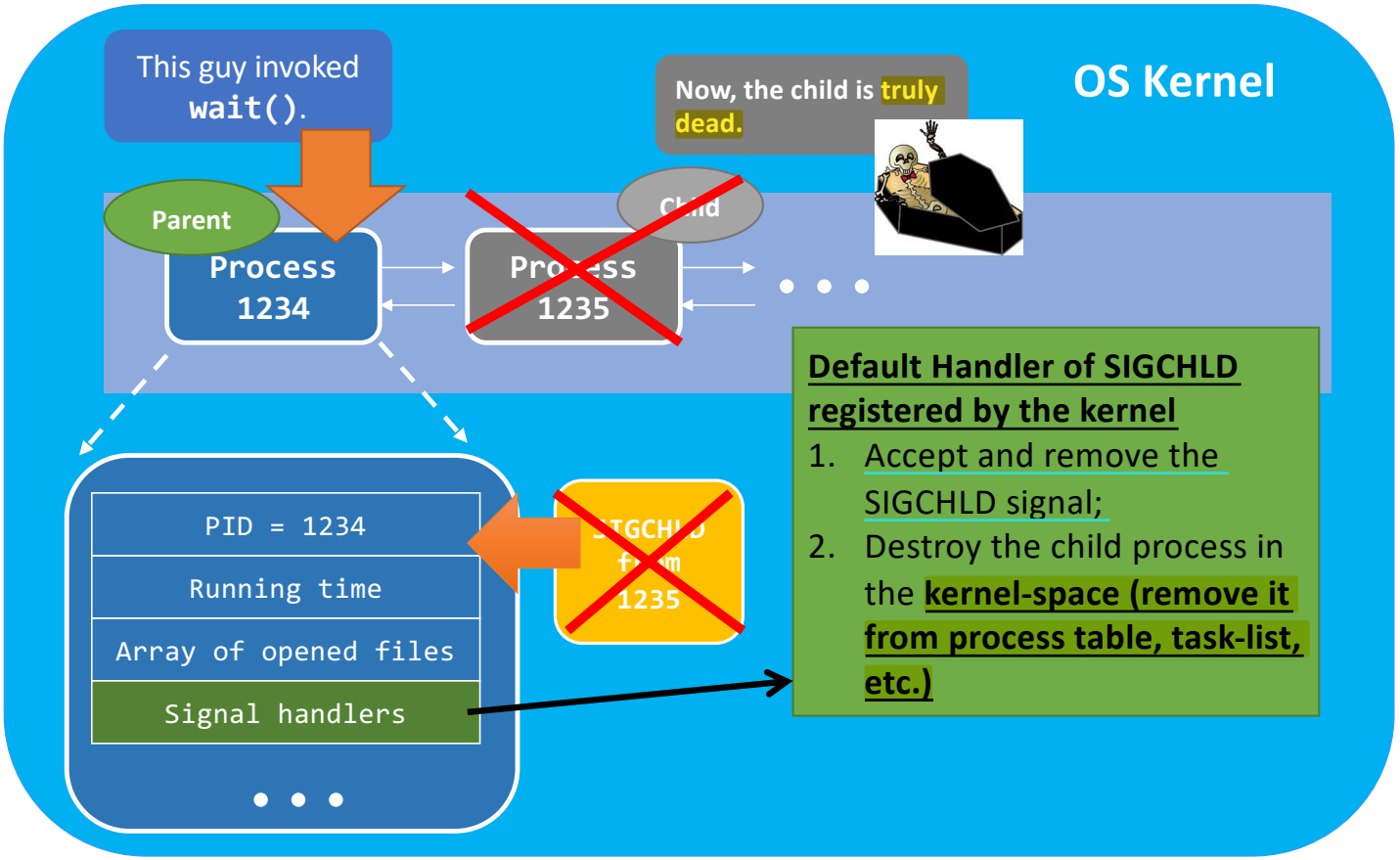
子进程退出并发送 SIGCHLD 信号，父进程被唤醒；4 父进程获



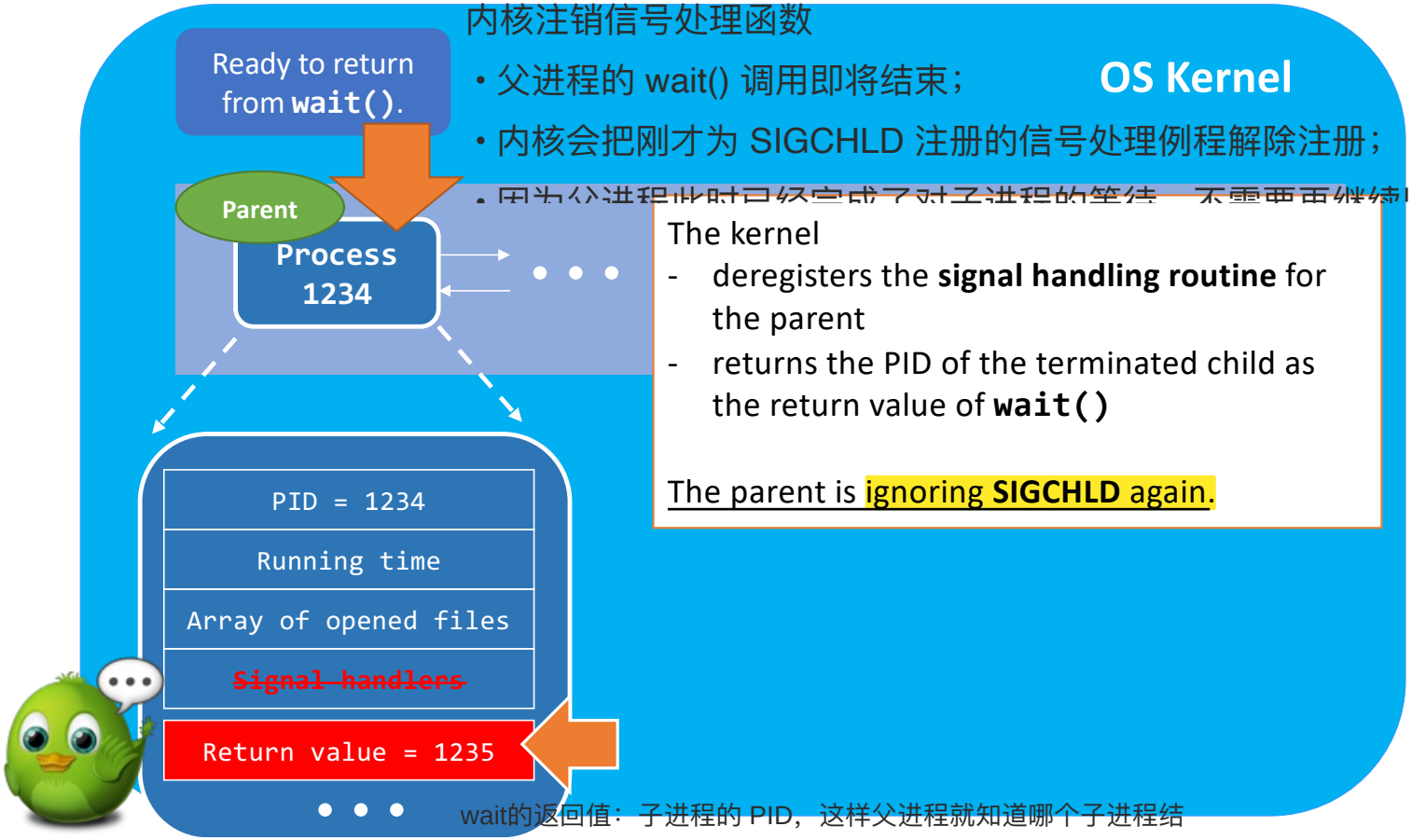
wait() Kernel View



wait() Kernel View

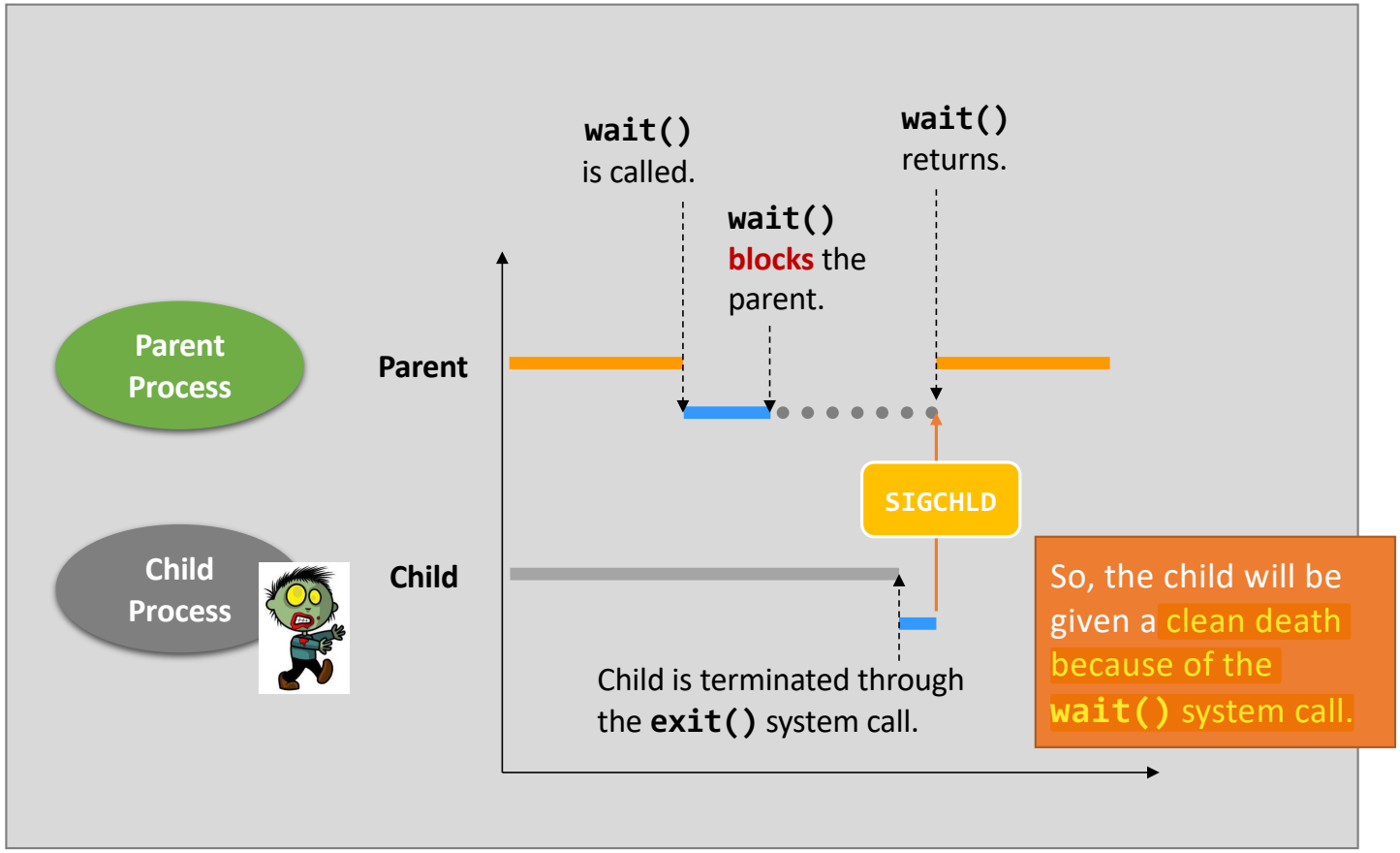


wait() Kernel View

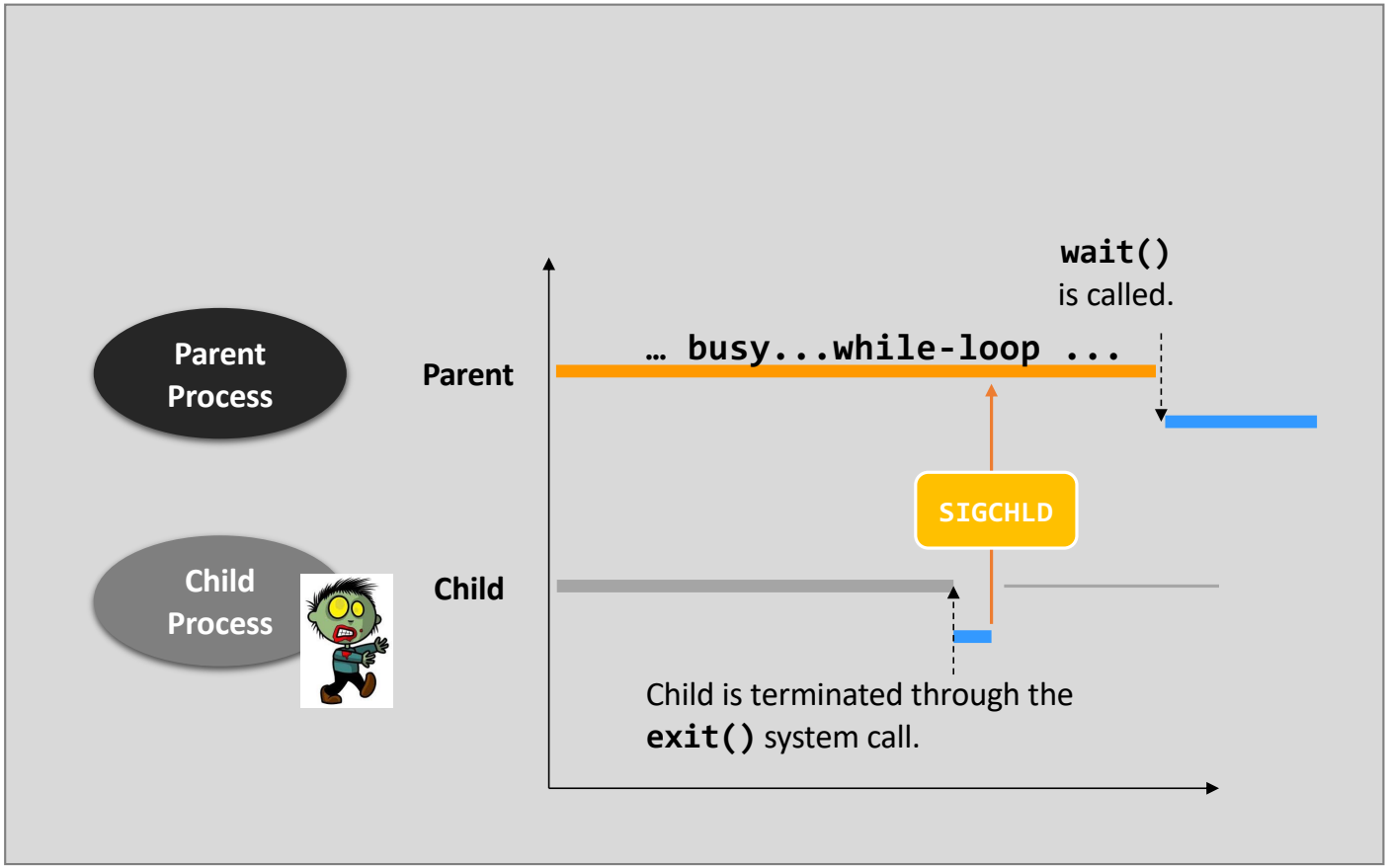


wait的返回值：子进程的 PID，这样父进程就知道哪个子进程结束了

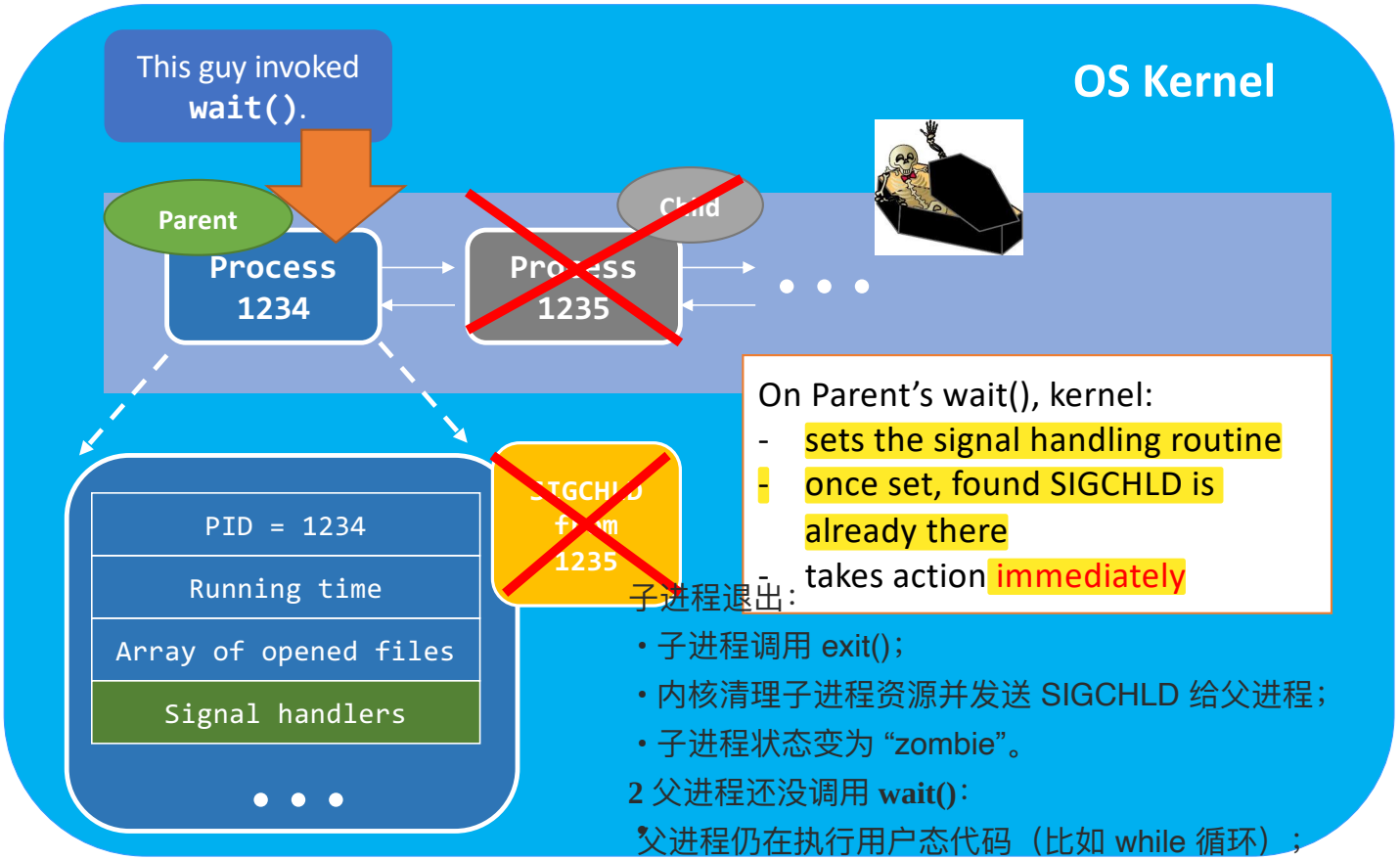
Normal Case


wait()
is called.
wait()
returns.
wait()
blocks the
parent.
Parent
Process
Parent
SIGCHLD
Child
Process
Child
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()



子进程退出：

- 子进程调用 exit()；
- 内核清理子进程资源并发送 SIGCHLD 给父进程；
- 子进程状态变为 “zombie”。

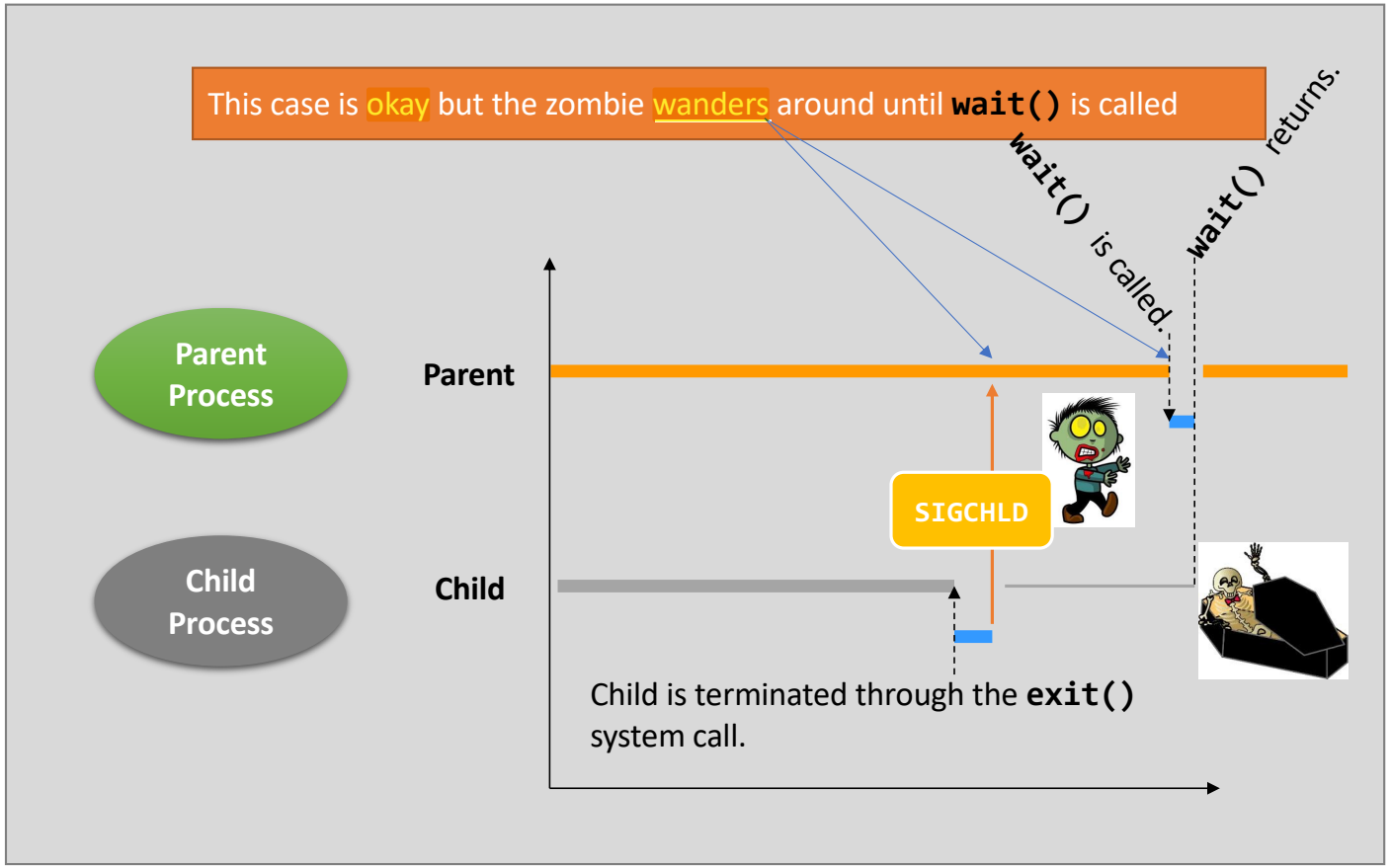
2 父进程还没调用 **wait()**：

- 父进程仍在执行用户态代码（比如 while 循环）；
- 它暂时忽略了 SIGCHLD 信号；
- 子进程此时处于“僵尸状态”，等待父进程来收尸。

3 父进程终于调用 wait()：

- 父进程进入内核态；

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()**. 当 main() 函数执行到结尾并 return 时，
  - The process terminates abnormally.
    - The kernel knows that the process is terminated abnormally. Hence, the kernel invokes **exit()** for it.

当程序异常终止（例如收到致命信号）时，
进程没有机会自己调用 exit()。
但内核会“代为调用” exit() 来清理资源。

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:

僵尸进程不占内存、不耗 CPU，但占系统名额。

Summary of wait() and exit()

```
 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 }
```

"enter" here

"enter" here

This program requires you to type "enter" twice before the process terminates.

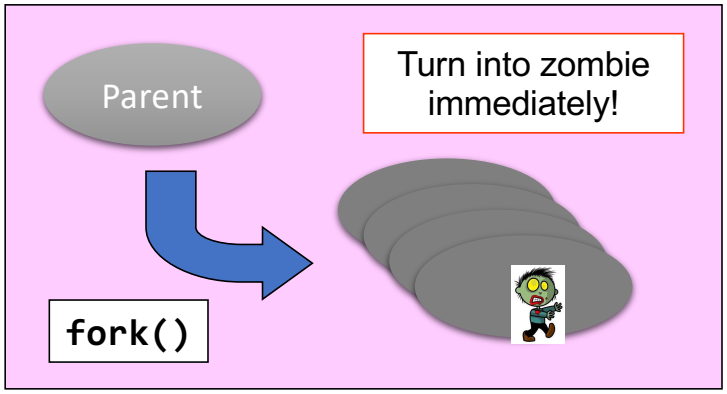
You are expected to see **the status of the child process changes (ps aux [PID])** between the 1st and the 2nd "enter".

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

```
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.

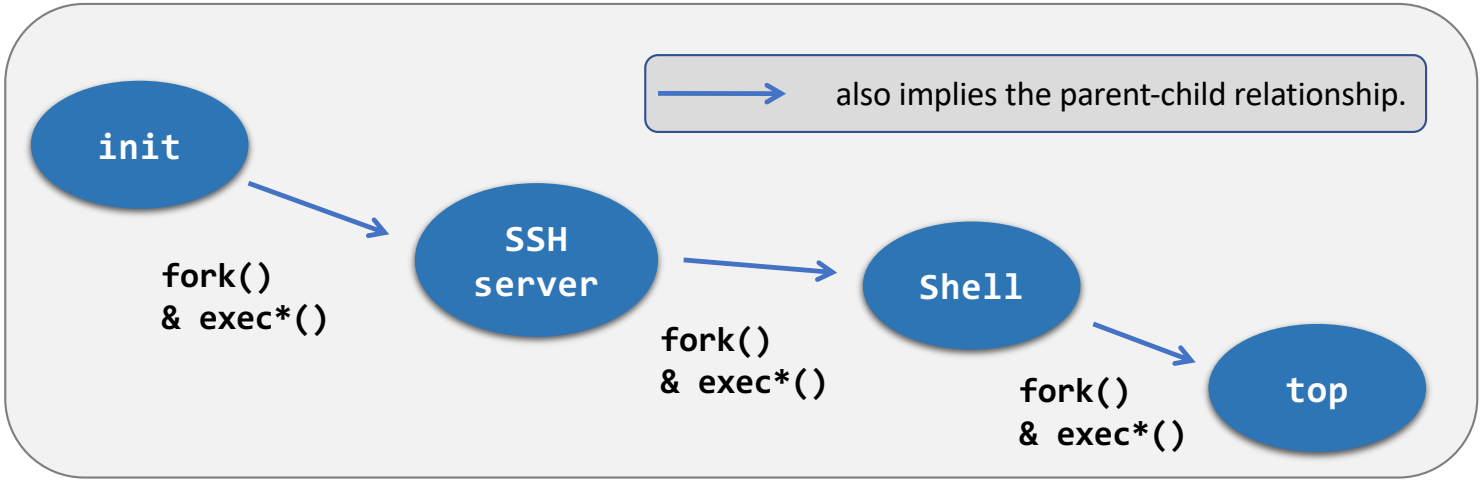
- The "init" process:
  - has PID = 1, and
  - is running the program code "/sbin/init".

- Its first task is to create more processes…
  - Using fork() and exec().

How does uCore create the first process?

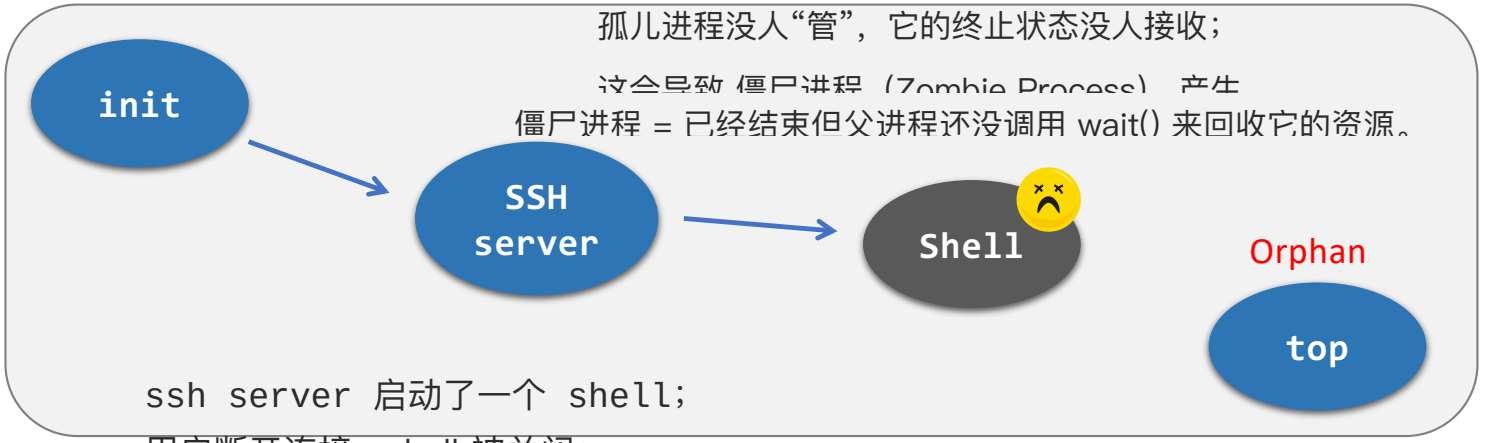
A Tree of Processes

- 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.



僵尸进程 = 已经结束但父进程还没调用 wait() 来回收它的资源。

ssh server 启动了一个 shell；

用户断开连接，shell 被关闭；

但 shell 下面的某个进程（比如 top）还在运行；

这时 top 的父进程（shell）已经不存在；

Re-parent

当某个进程成为孤儿时，系统会自动把它的父进程改为 init。

init 是 PID=1 的进程，它会定期调用 wait() 来清理孤儿。

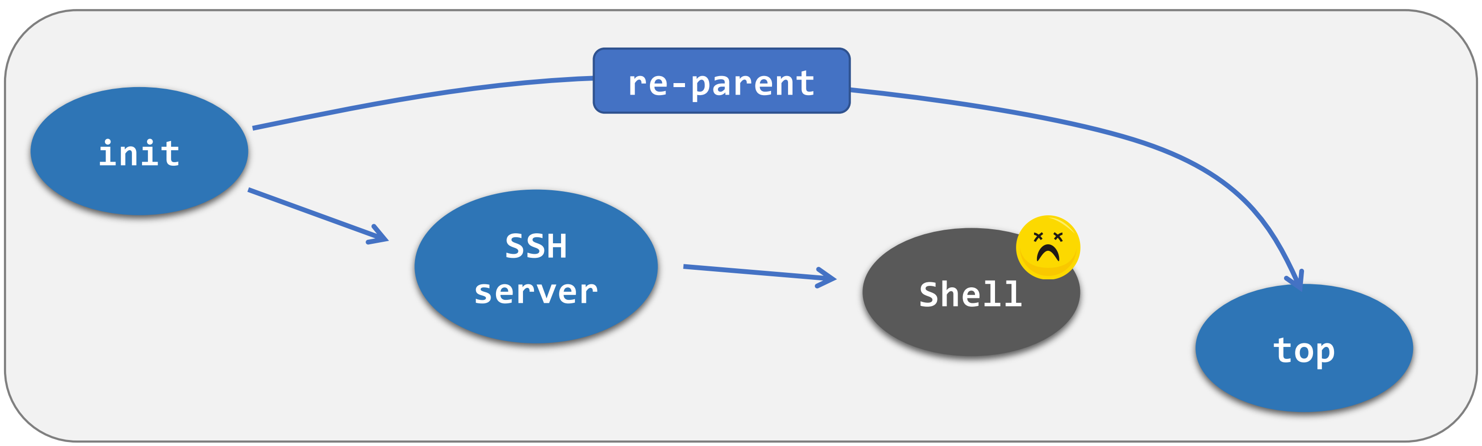
所以 init 被称为所有孤儿进程的“继母（step- mother）”

- In Linux
  - The **“init”** process will become the step-mother of all orphans
  - It's called **re-parenting**

Windows 的进程层级管理方式不同，它维护的是一个 forest-like hierarchy（森林式层级）；

每个进程的父子关系更加松散；

- In Windows
  - It maintains a *forest-like process hierarchy*………系统用不同的引用计数机制来管理



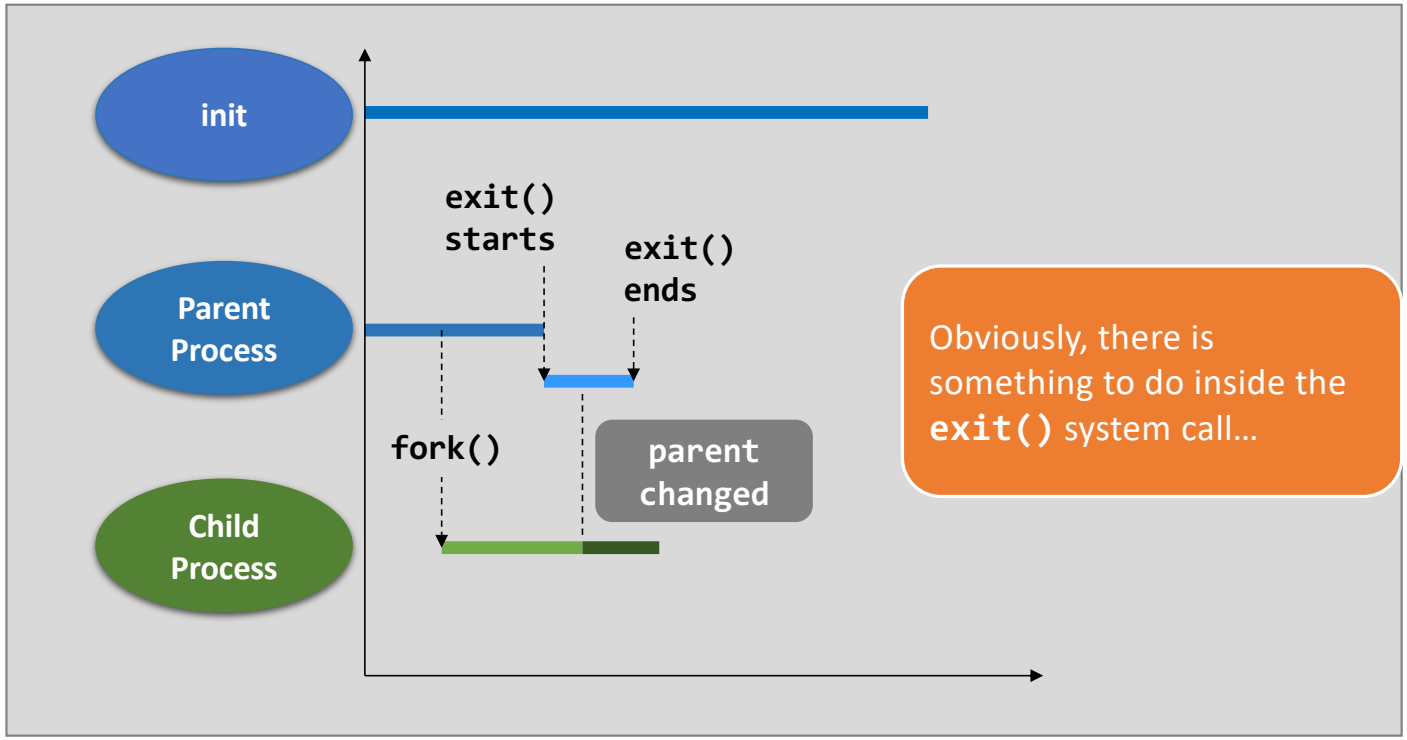
An Example

```
 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  }
```

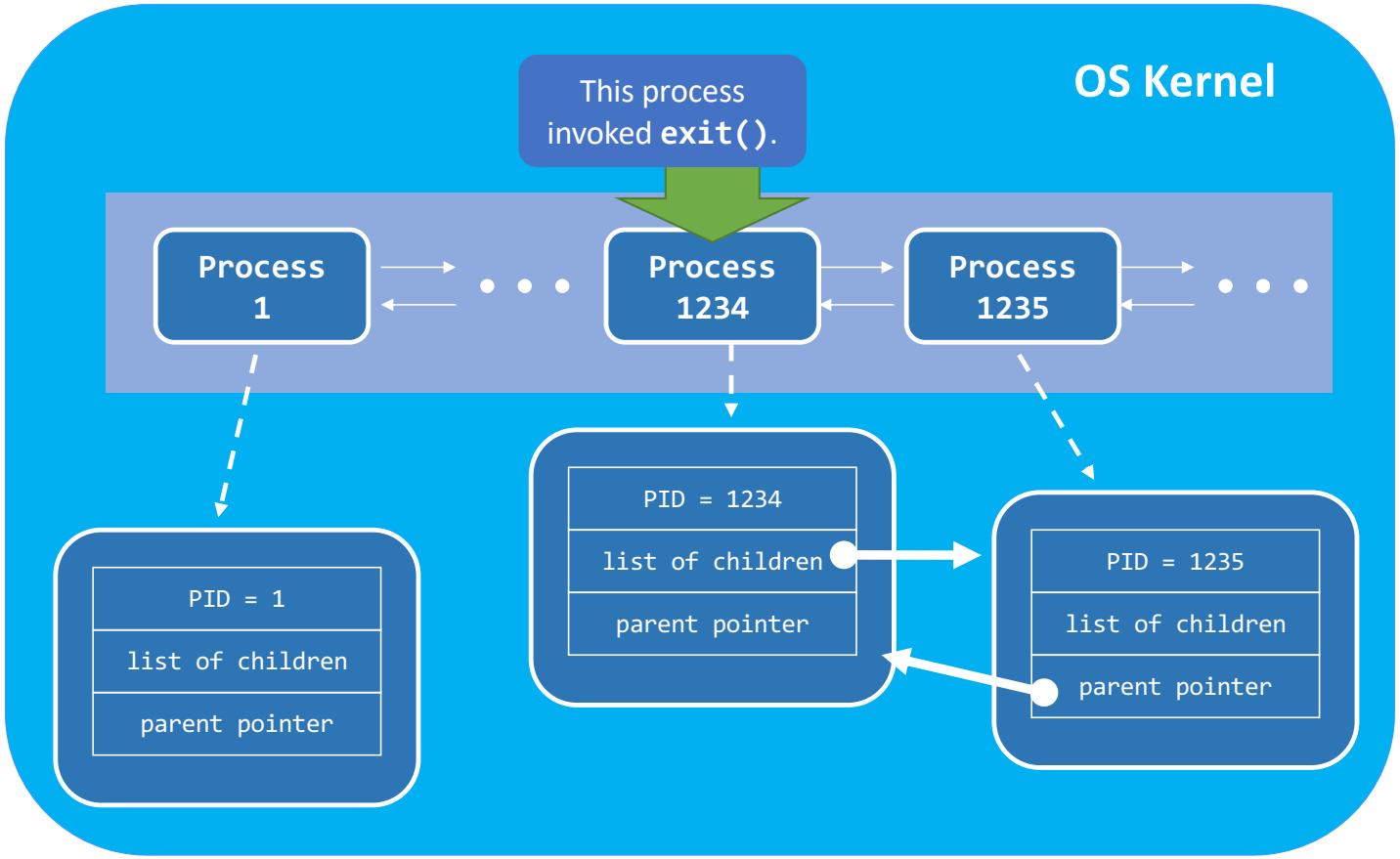
getppid() is the system call that returns the parent's PID of the calling process.

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

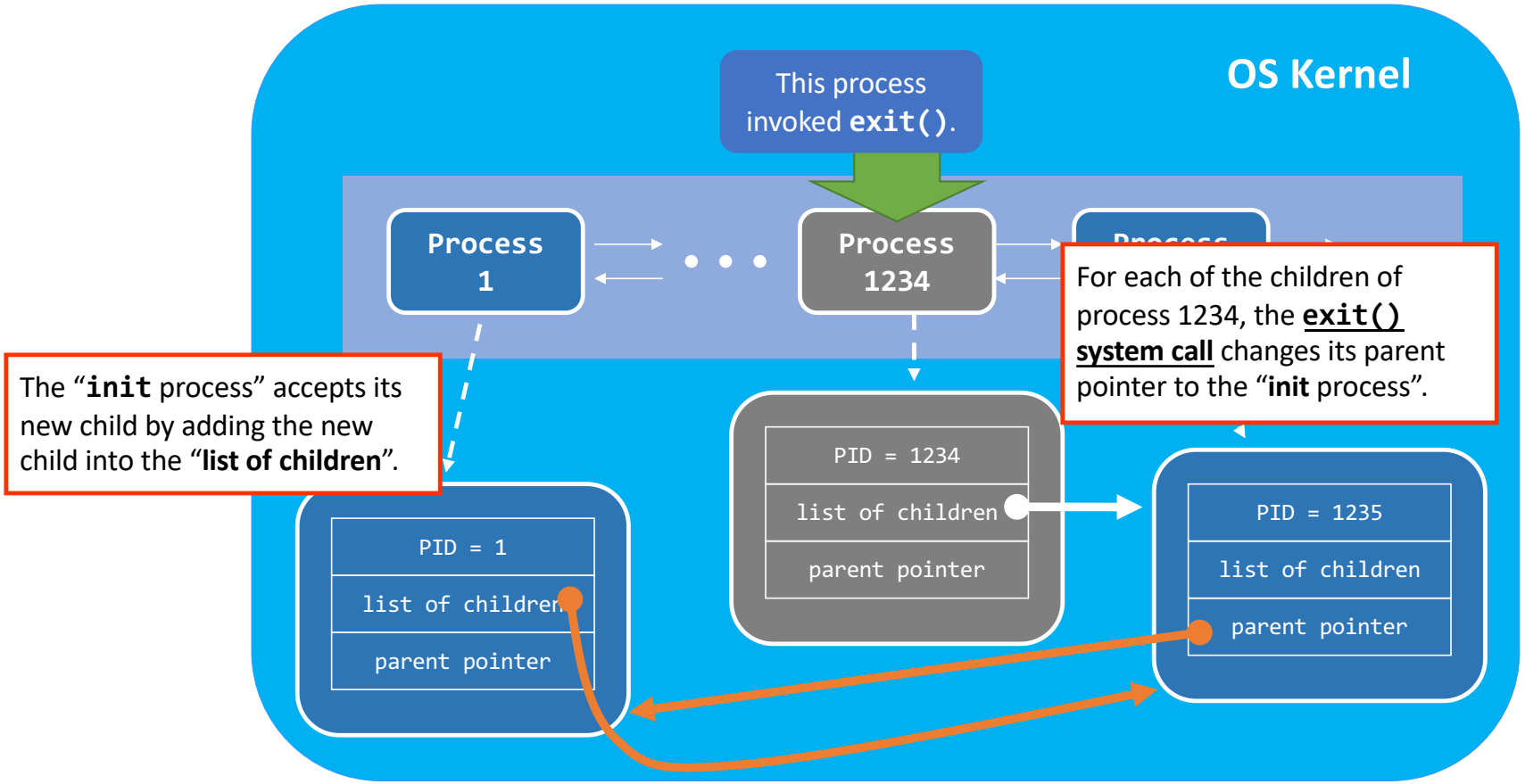
Re-parenting Explained



Re-parenting Explained (Cont'd)

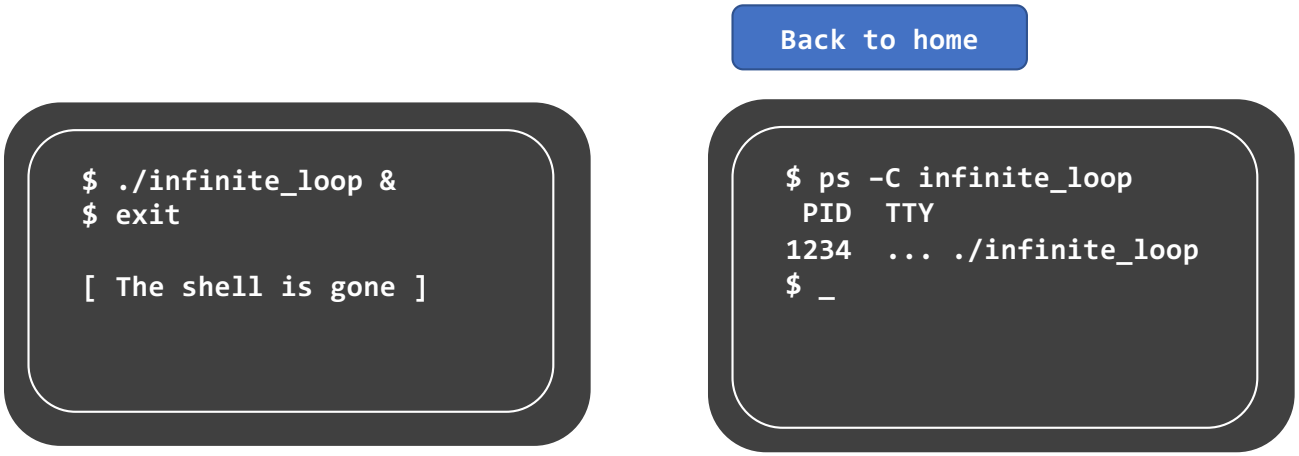


Re-parenting Explained (Cont'd)



Background Jobs

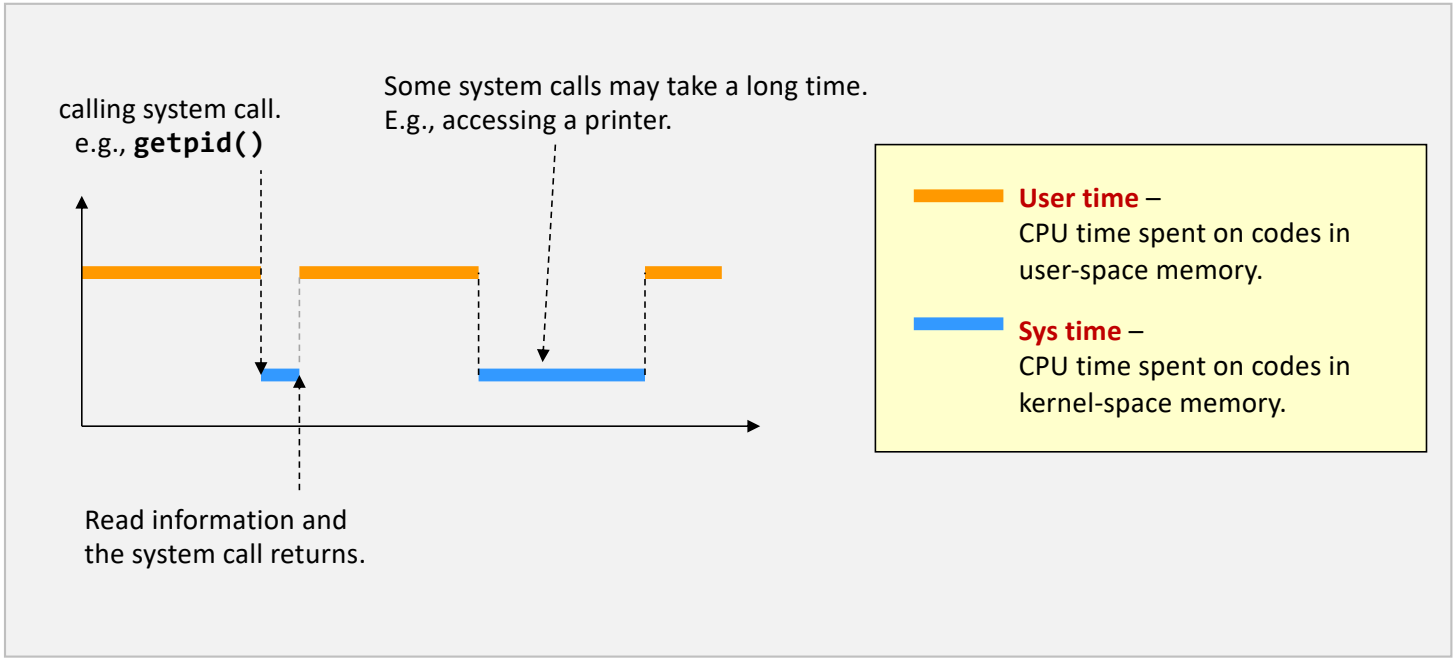
- The re-parenting operation enables something called **background jobs** in Linux
  - It allows a process runs **without a parent terminal/shell**



当一个前台进程被放到后台运行后，如果 终端关闭（shell 退出）：

- 该进程的父进程（shell）会消失；
- 操作系统就会触发 re-parenting（重新收养）；
- 这个后台进程会被 init（PID=1） 收养；
- 从此，它可以在没有终端的情况下继续运行。

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?

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

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

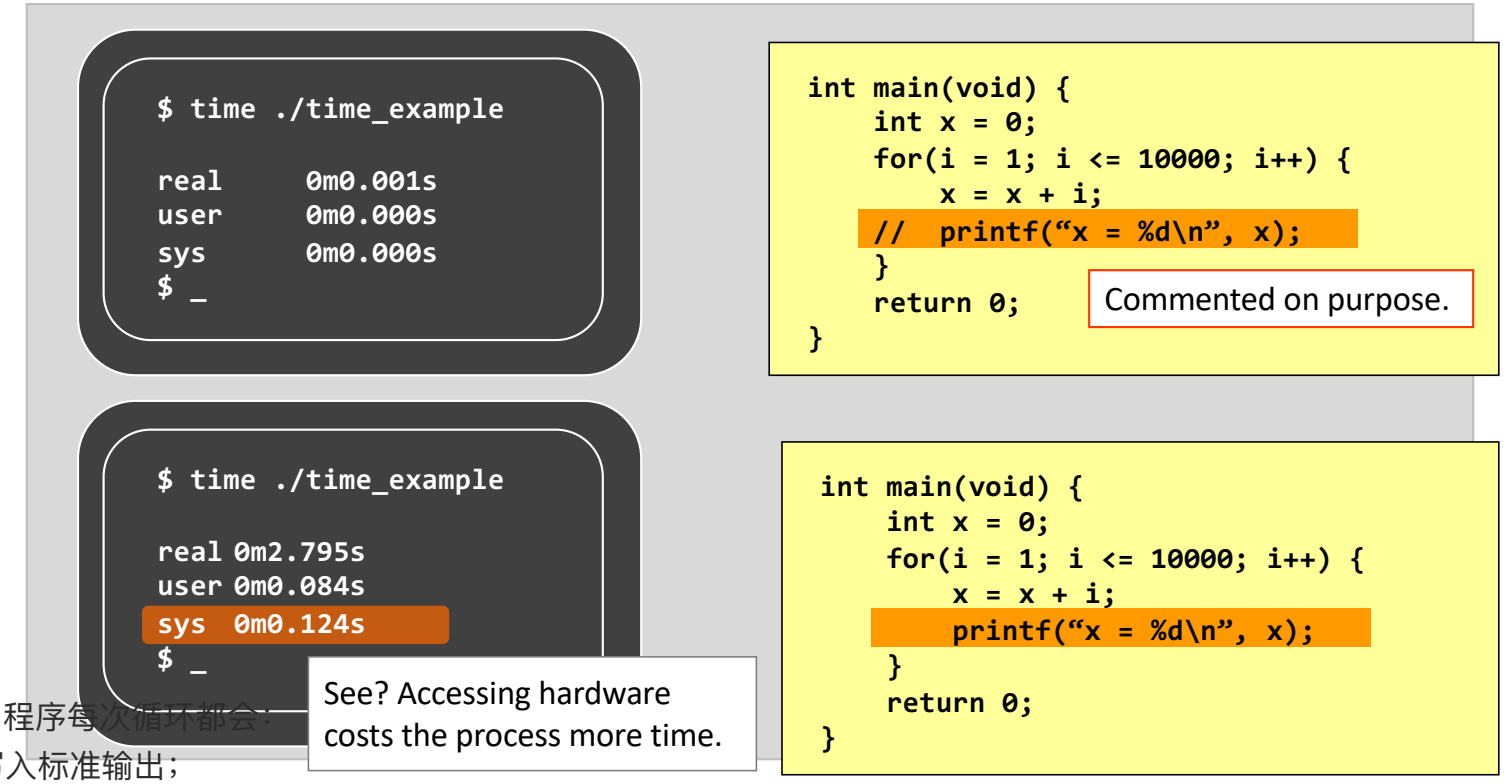
程序只做 简单加法运算；

没有任何系统调用（如 printf、文件访问等）；

所以几乎所有时间都花在 用户态计算（user time）；

程序非常快，以至于 real ≈ user ≈ sys ≈ 0。

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



加上 printf() 后，程序每次循环都会：

- 调用系统函数写入标准输出；
- 从用户态切换到内核态（进行 I/O 操作）。

这种频繁的 系统调用 会带来：

- CPU 切换开销；
- 与终端设备（硬件）交互的延迟。

所以：

- user time 增加（程序计算 + 调用接口）；

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\n");
    return 0;
}
```

• 这个程序一共调用了 1,000,000 次 printf()；
• 每次调用 printf() 都会触发一次 系统调用（syscall）；
• 每次系统调用都需要：
1 用户态 → 内核态切换；
2 写入输出缓冲区；
3 返回用户态；
• 所以大部分时间都浪费在系统调用切换上；

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\n");
    return 0;
}
```

```
$ time ./time_example_fast

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

Thank you!