

Lecture 3 Processes

Prof. Yinqian Zhang

Spring 2023

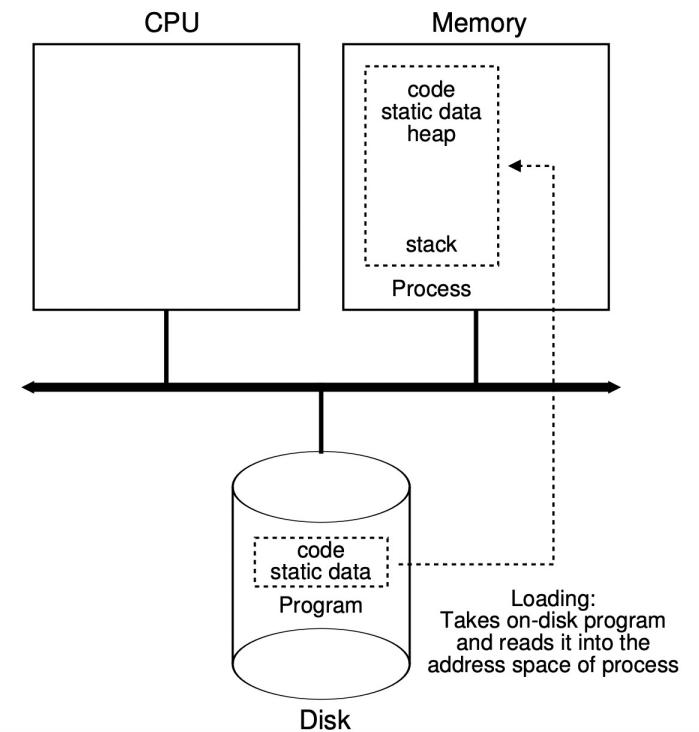
Outline

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

Process and System Calls

What Is a Process

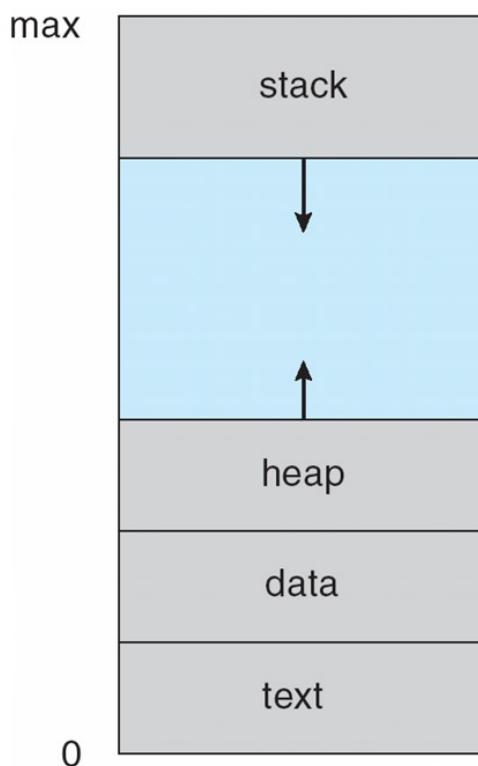
- Process is a program in execution
- A program is a file on the disk
 - Code and static data
- A process is loaded by the OS
 - Code and static data are loaded from the program
 - Heap and stack are created by the OS



What Is a Process (Cont'd)

- 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 是管理文件的最小单位
其中的每个 thread 共享文件。



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.

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

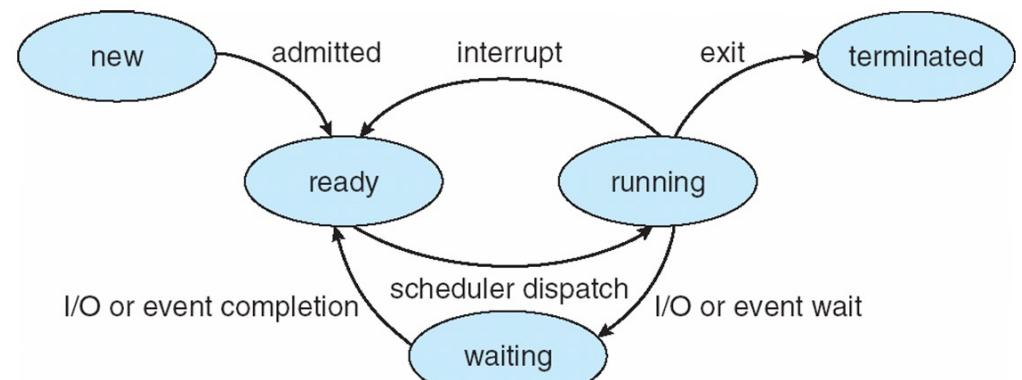
int main(void) {
    printf("My PID is %d\n", getpid());
}
```

```
$ ./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

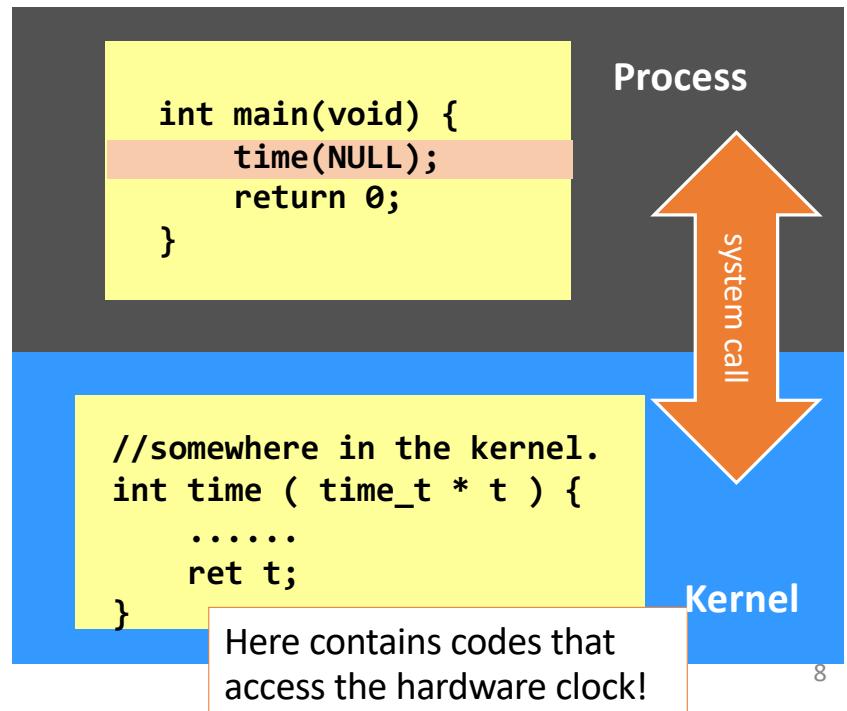


System Call: Process-Kernel Interaction

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

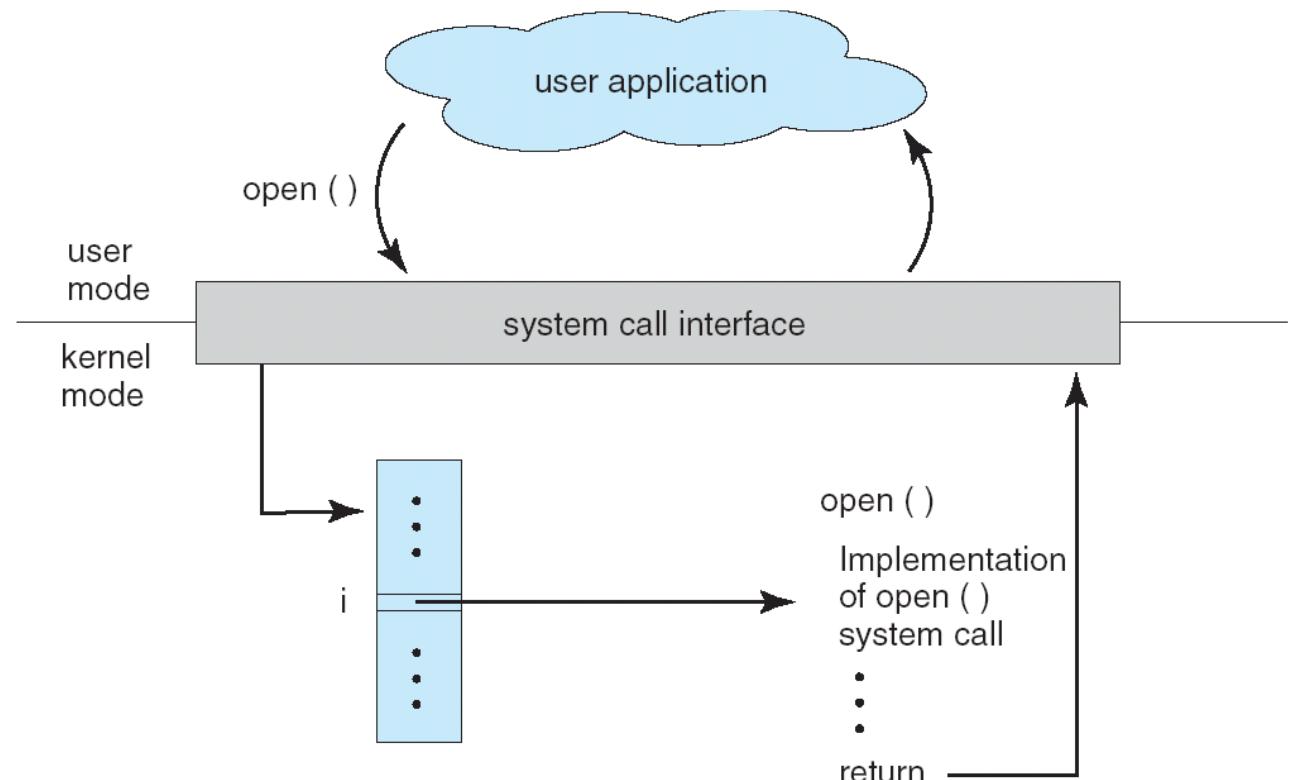
```
int add_function(int a, int b) {  
    return (a + b);  
}  
  
int main(void) {  
    int result;  
    result = add_function(a,b);  
    return 0;  
}  
  
// this is a dummy example...
```

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
```

user mode → supervisor mode

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]();  
}
```

查表提取 parameter

IDT 不用提取 parameter

描述符表是由若干个描述符组成,

每个描述符占用8个字节的内存空间, 每个描述符表内最多可以有8129个描述符。

描述符是描述一个段的大小, 地址及各种状态的。

描述符表有三种, 分别为全局描述符表GDT、局部描述符表LDT和中断描述符表IDT。

1. 全局描述符表GDT:

全局描述符表在系统中只能有一个,且可以被每一个任务所共享.任何描述符都可以放在GDT中,但中断门和陷阱门放在GDT中是不会起作用的.能被多个任务共享的内存区就是通过GDT完成的,

2. 局部描述符表LDT:

局部描述符表在系统中可以有多个,通常情况下是与任务的数量保持对等,但任务可以没有局部描述符表.任务间不相干的部分也是通过LDT实现的.这里涉及到地址映射的问题.和GDT一样,中断门和陷阱门放在LDT中是不会起作用的.

3. 中断描述符表IDT:

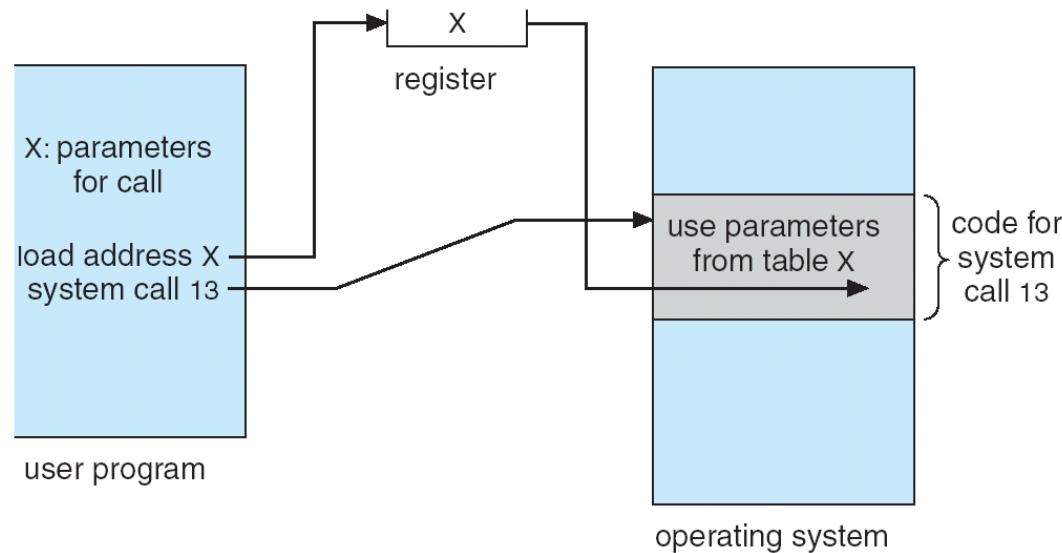
和GDT一样,中断描述符表在系统最多只能有一个,中断描述符表内可以存放256个描述符,分别对应256个中断.因为每个描述符占用8个字节,所以IDT的长度可达2K.中断描述符表中可以有任务门、中断门、陷阱门三个门描述符, 其它的描述符在中断描述符表中无意义。

System Call: Parameter Passing

- 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

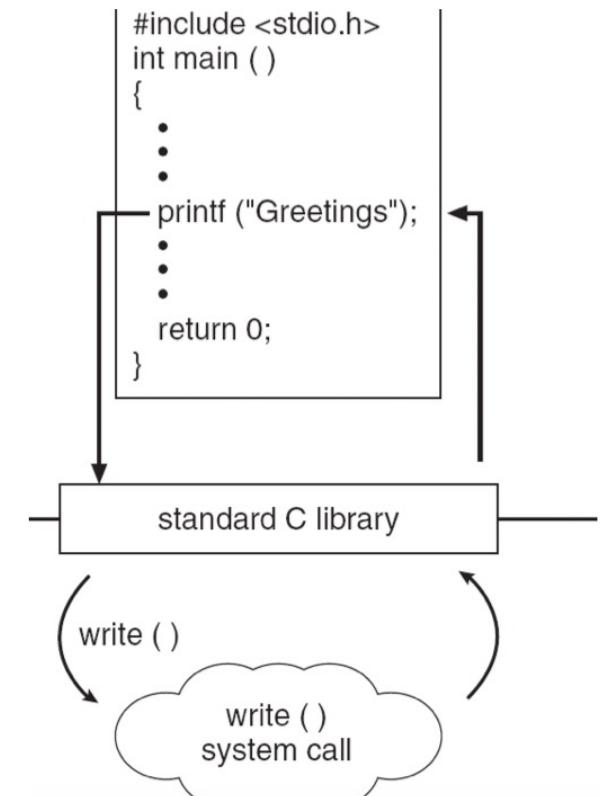
- Example: parameter passing via blocks



System Call v.s. Library API Call

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

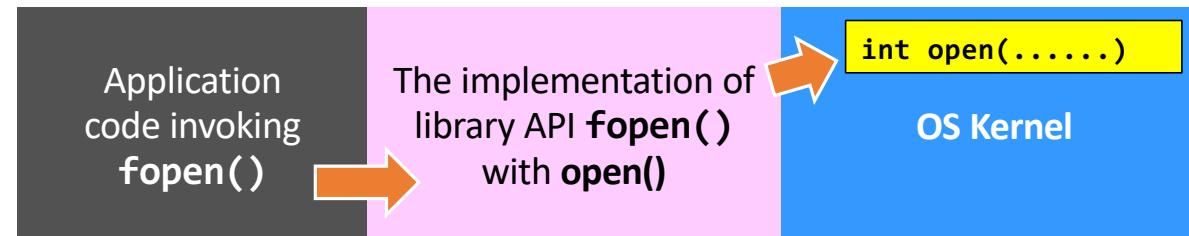
Name	System call?
printf() & scanf()	No
malloc() & free()	No
fopen() & fclose()	No
mkdir() & rmdir()	Yes
chown() & chmod()	Yes



System Call 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	<code>fopen("hello.txt", "w");</code>
System call	<code>open("hello.txt", O_WRONLY O_CREAT O_TRUNC, 0666);</code>



Process Creation

Process Creation

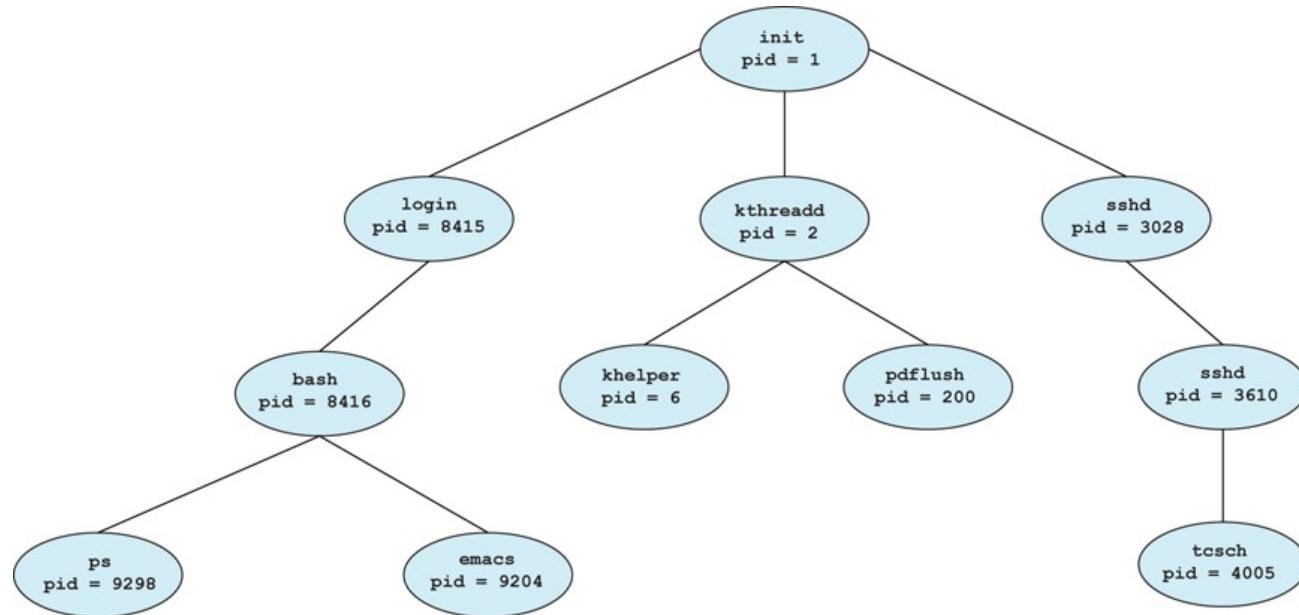
- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- Generally, process identified and managed via a process identifier (pid)
- Resource sharing
 - Parent and children share all resources
 - Children share subset of parent's resources
 - Parent and child share no resources
- Execution
 - Parent and children execute concurrently
 - Parent waits until children terminate

Process Creation (Cont'd)

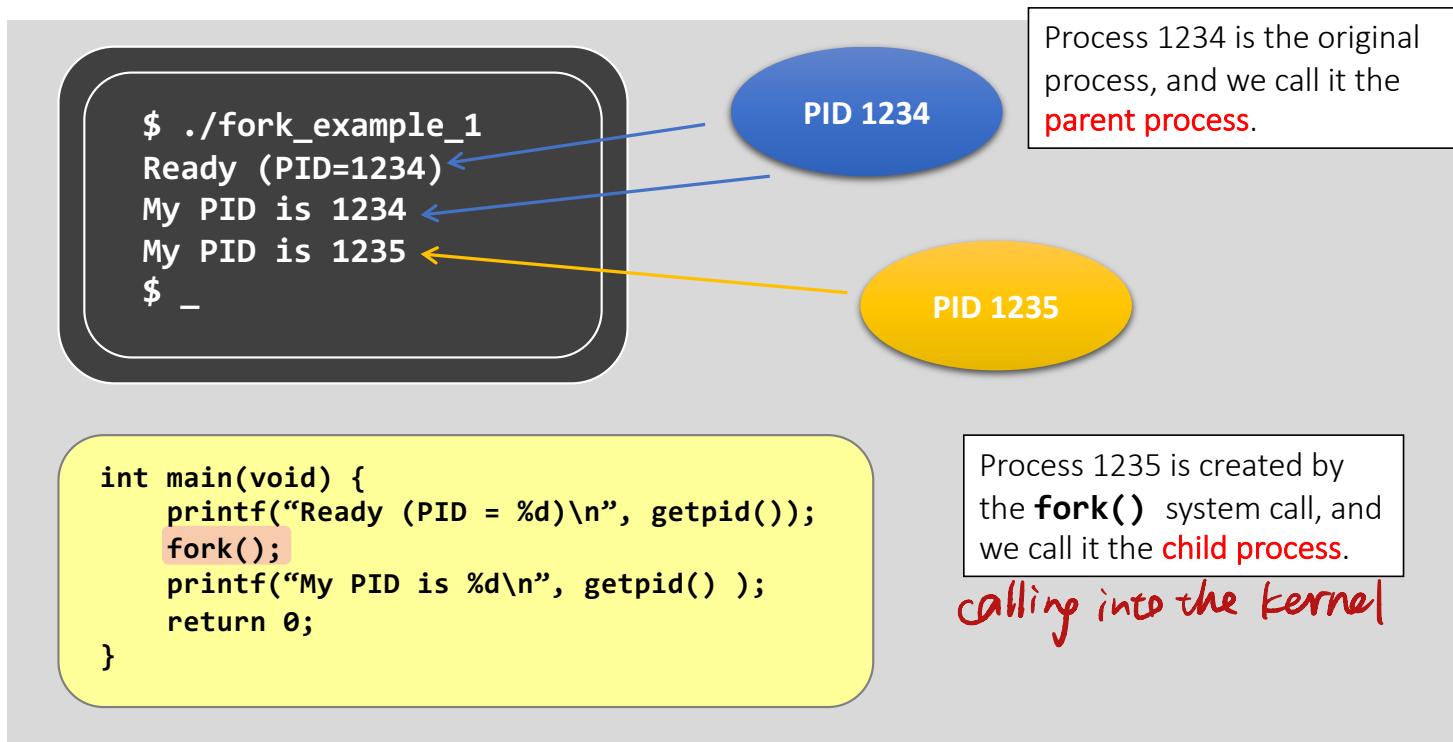
- Address space
 - Child duplicate of parent
 - Child has a program loaded into it
- UNIX examples
 - fork system call creates new process
 - exec system call used after a fork to replace the process' memory space with a new program

Process Creation (Cont'd)

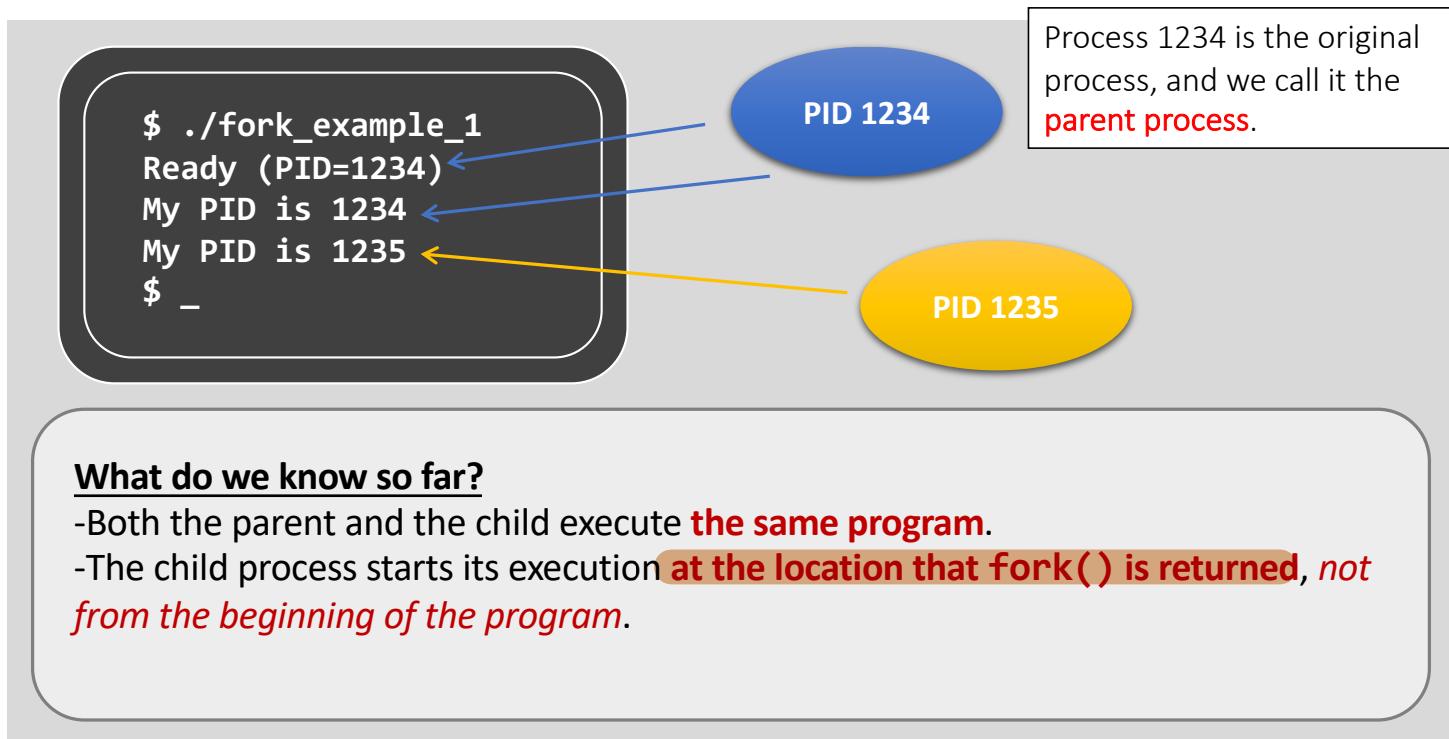
- A tree of processes in Linux



Creating Processes with fork() System Call



Creating Processes with fork() System Call



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.

PID 1234

fork()

PID 1235

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
(waiting)

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
```

PID 1234
(running)

PID 1235
(waiting)

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.
```

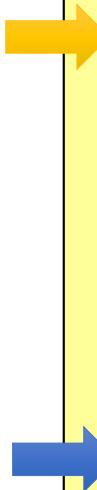
PID 1234
(stop)

PID 1235
(waiting)

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 }
```

return value } parent : pid of child process
of fork () } child : 0 .



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

Important

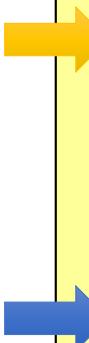
For child, the return value of **fork()** is 0.

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

```
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)

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.

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

- 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()

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

```
int main(void) {  
    printf("before execl ...\\n");  
    load to memory and execute  
    execl('/bin/ls', "/bin/ls", NULL);  
  
    printf("after execl ...\\n");  
  
    return 0;  
}
```



```
$./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()

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

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

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

exec()

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

```
exec("/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");  
    // Both my mem is in the code都被覆盖  
    execl("/bin/ls" "/bin/ls", NULL);  
  
    printf("after execl ...\\n");  
  
    return 0;  
}
```

never get
executed →

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.

```
/* code of program exec_example */

int main(void) {

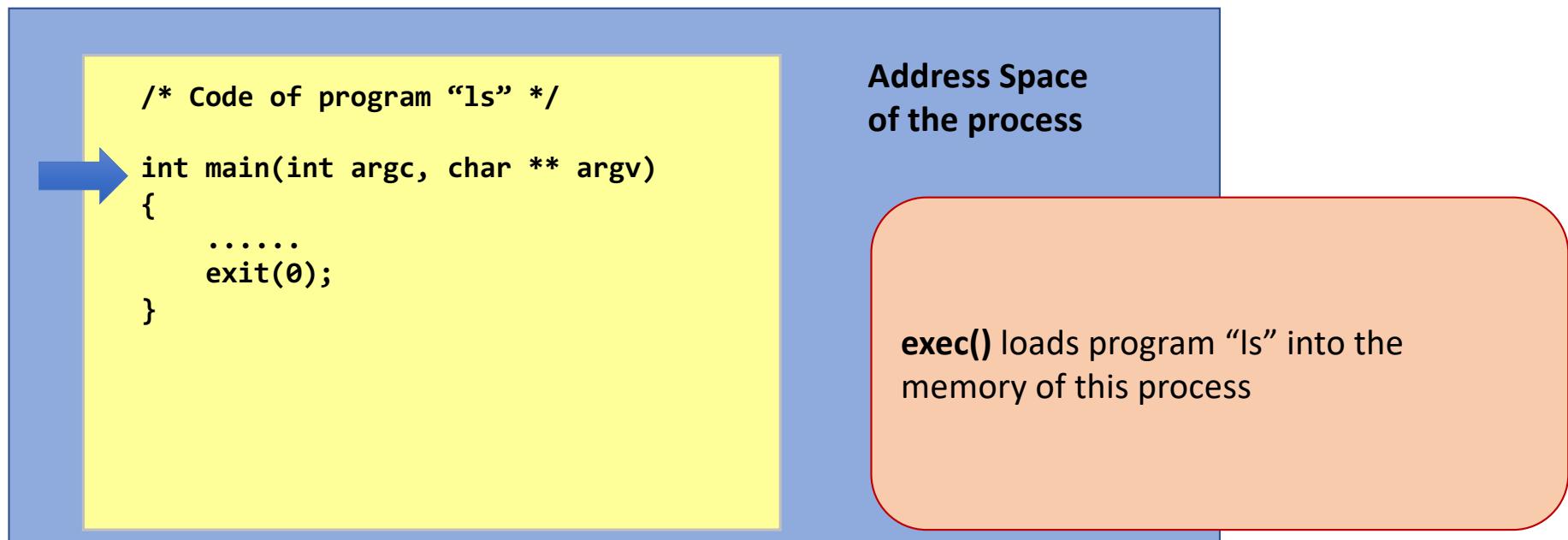
    printf("before execl ...\\n");
    execl("/bin/ls", "/bin/ls", NULL);
    printf("after execl ...\\n");
    return 0;
}
```

Address Space
of the process



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.

```
/* Code of program "ls" */

int main(int argc, char ** argv)
{
    .....
    exit(0);
}
```

Address Space
of the process

The “return” or the “exit()”
statement in “/bin/ls” will terminate
the process...

Therefore, it is certain that the process
cannot go back to the old program!

`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.
$ _
```

wait(): Sync Parent with Child

```
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); wait until child process terminates
14        printf("My PID is %d\\n", getpid());
15    }
16
17    printf("program terminated.\\n");
18 }
```

Parent return
from fork() first

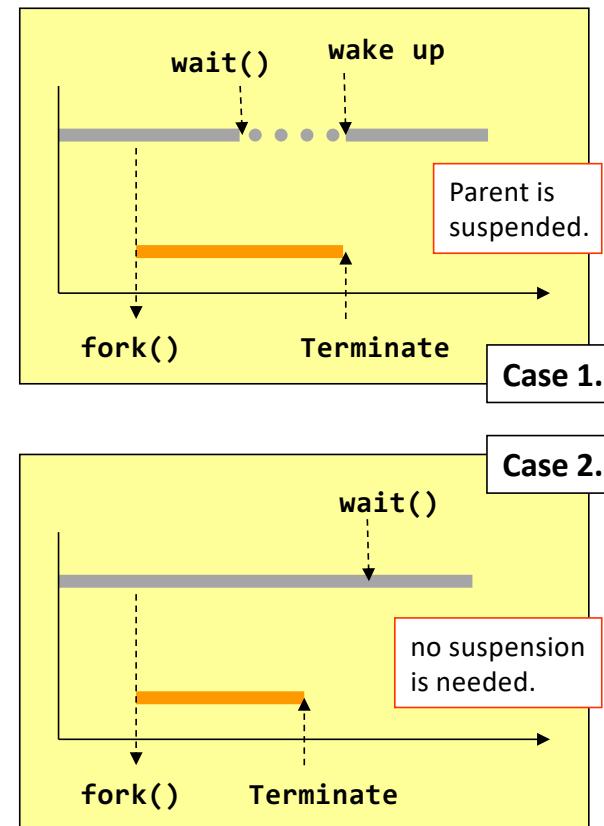
```
$ ./fork_example_2
before fork ...
result = 1235
I'm the parent.
result = 0
I'm the child.
My PID is 1235
program terminated.
My PID is 1234
program terminated.
$ _
```

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()

- 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

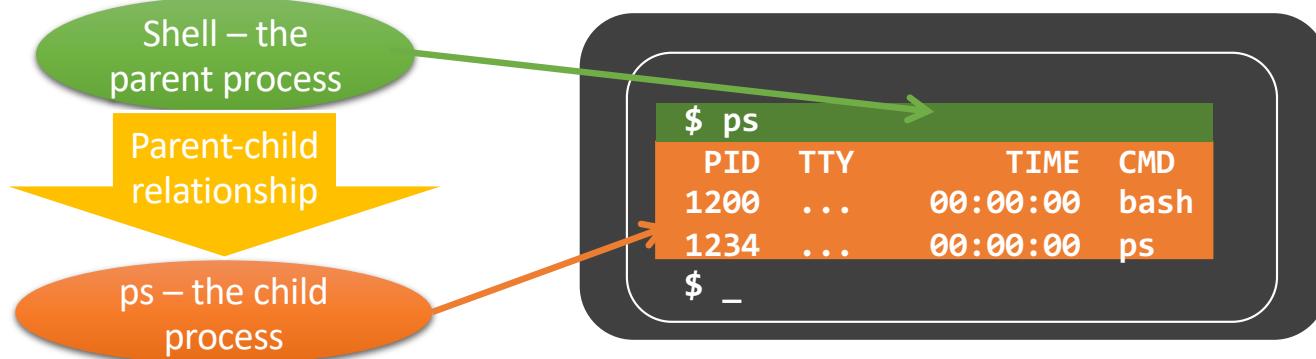


wait() v.s. waitpid()

- **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)

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 child process to `exec()` the target program
 - Use `wait()` to wait until the child process terminates

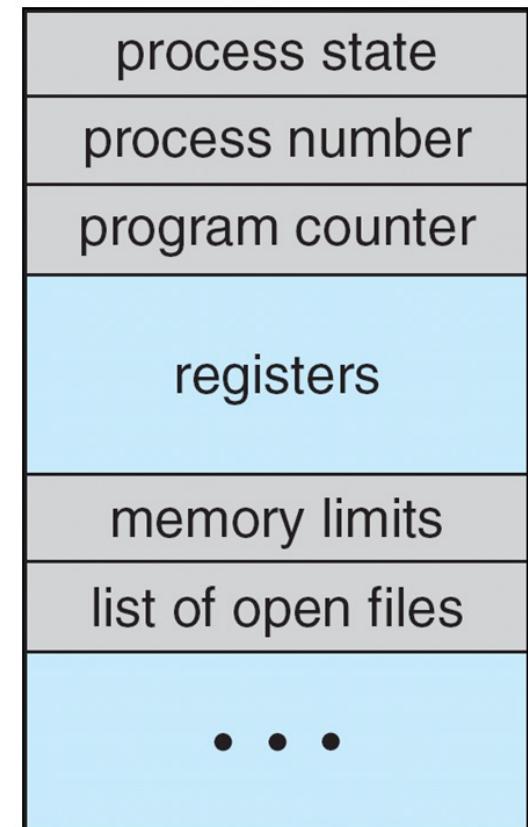


Processes: Kernel View

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
```

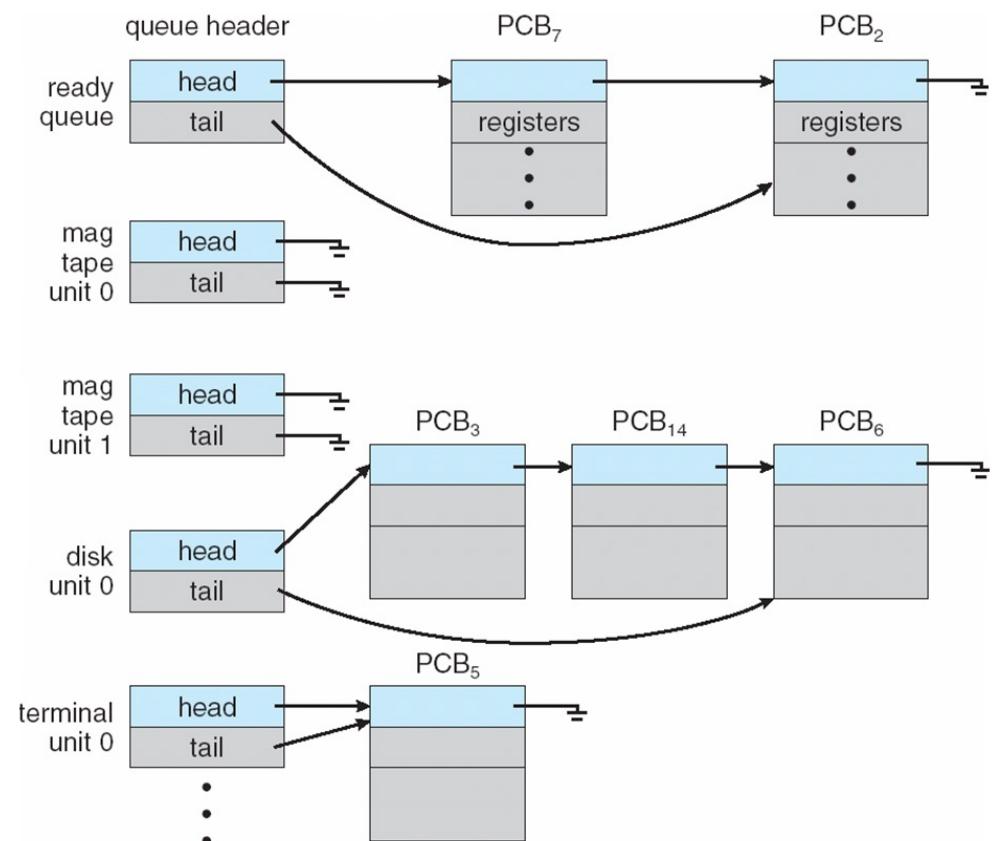
PCB Example: uCore

```
/* 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
};
```

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

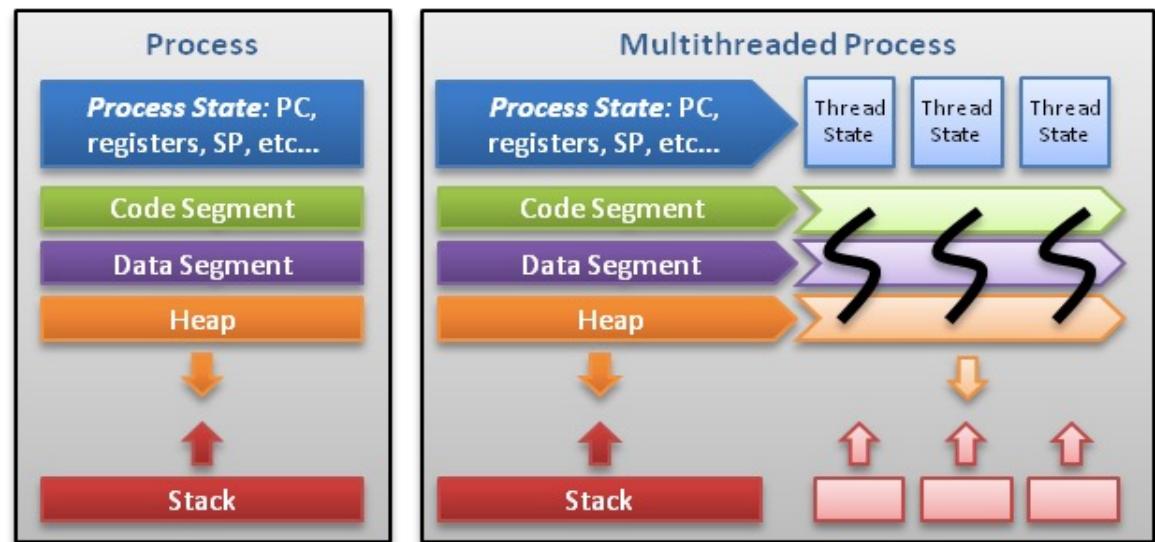
表示执行逻辑

多线程共享数据. 多进程通讯需特定 API.

- 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?
 - Cooperative scheduling: OS will have to wait
 - Early Macintosh OS, old Alto system
 - Non-cooperative scheduling: timer interrupts
 - Modern operating systems

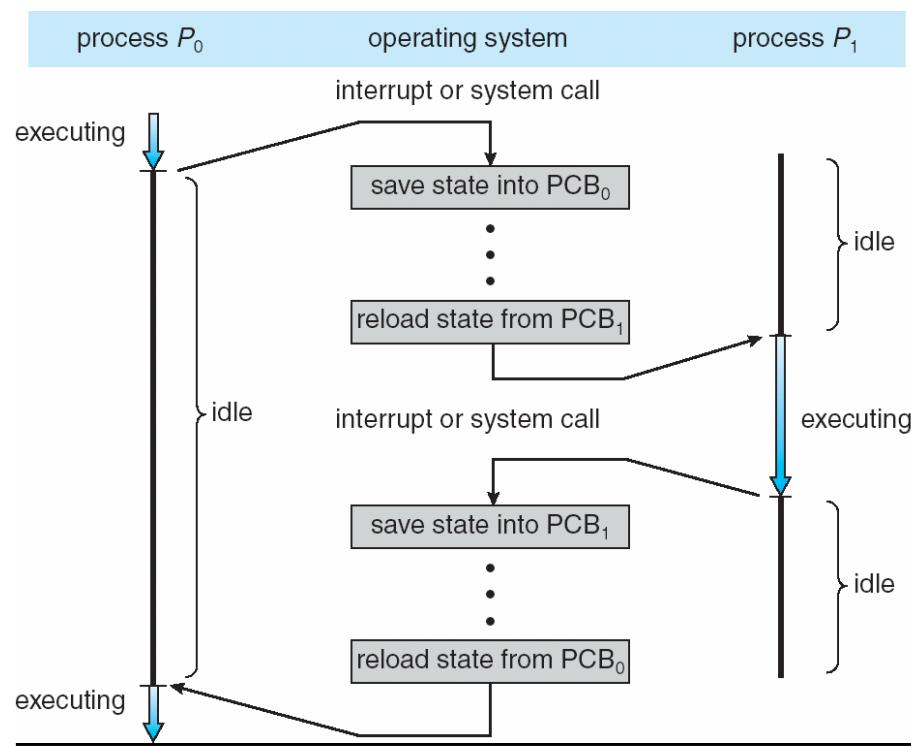
Switching Between Processes (Cont'd)

- 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)
```

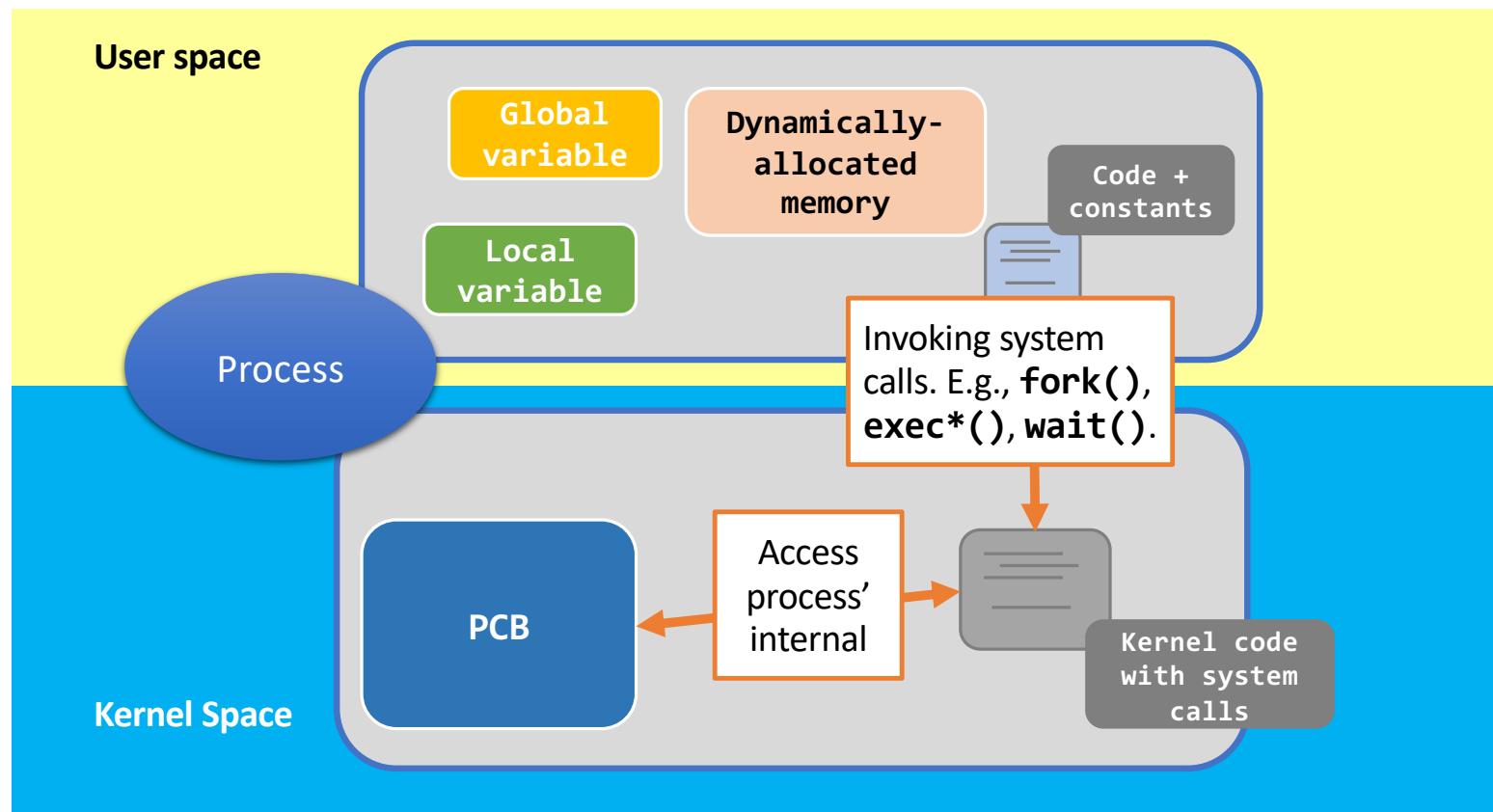
```
# restore to's registers
LOAD ra, 0*REGBYTES(a1)
LOAD sp, 1*REGBYTES(a1)
LOAD s0, 2*REGBYTES(a1)
LOAD s1, 3*REGBYTES(a1)
LOAD s2, 4*REGBYTES(a1)
LOAD s3, 5*REGBYTES(a1)
LOAD s4, 6*REGBYTES(a1)
LOAD s5, 7*REGBYTES(a1)
LOAD s6, 8*REGBYTES(a1)
LOAD s7, 9*REGBYTES(a1)
LOAD s8, 10*REGBYTES(a1)
LOAD s9,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

```
ret
```

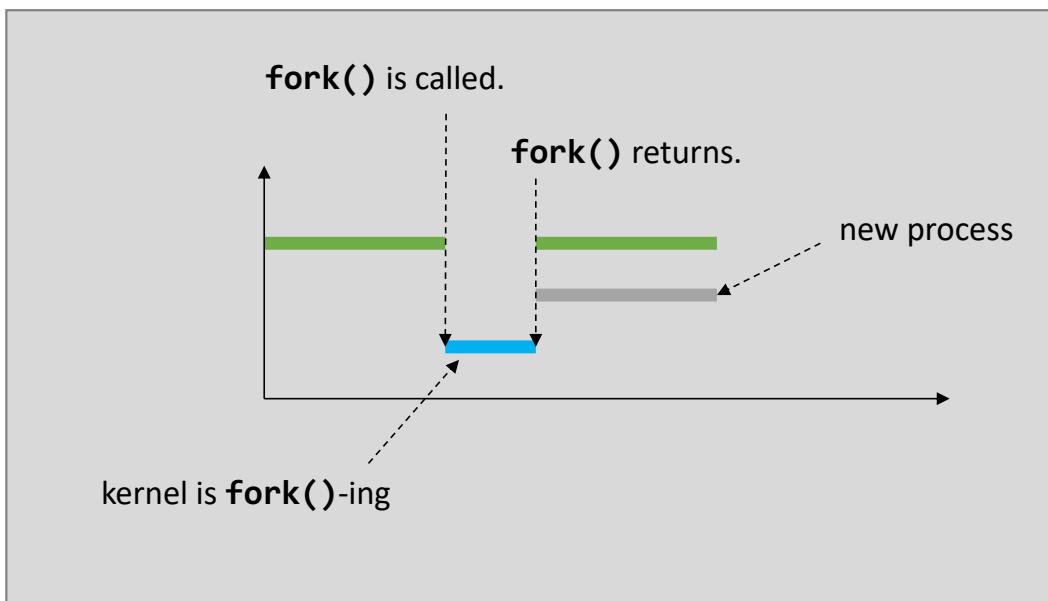
fork(), exec(), wait()

Kernel View

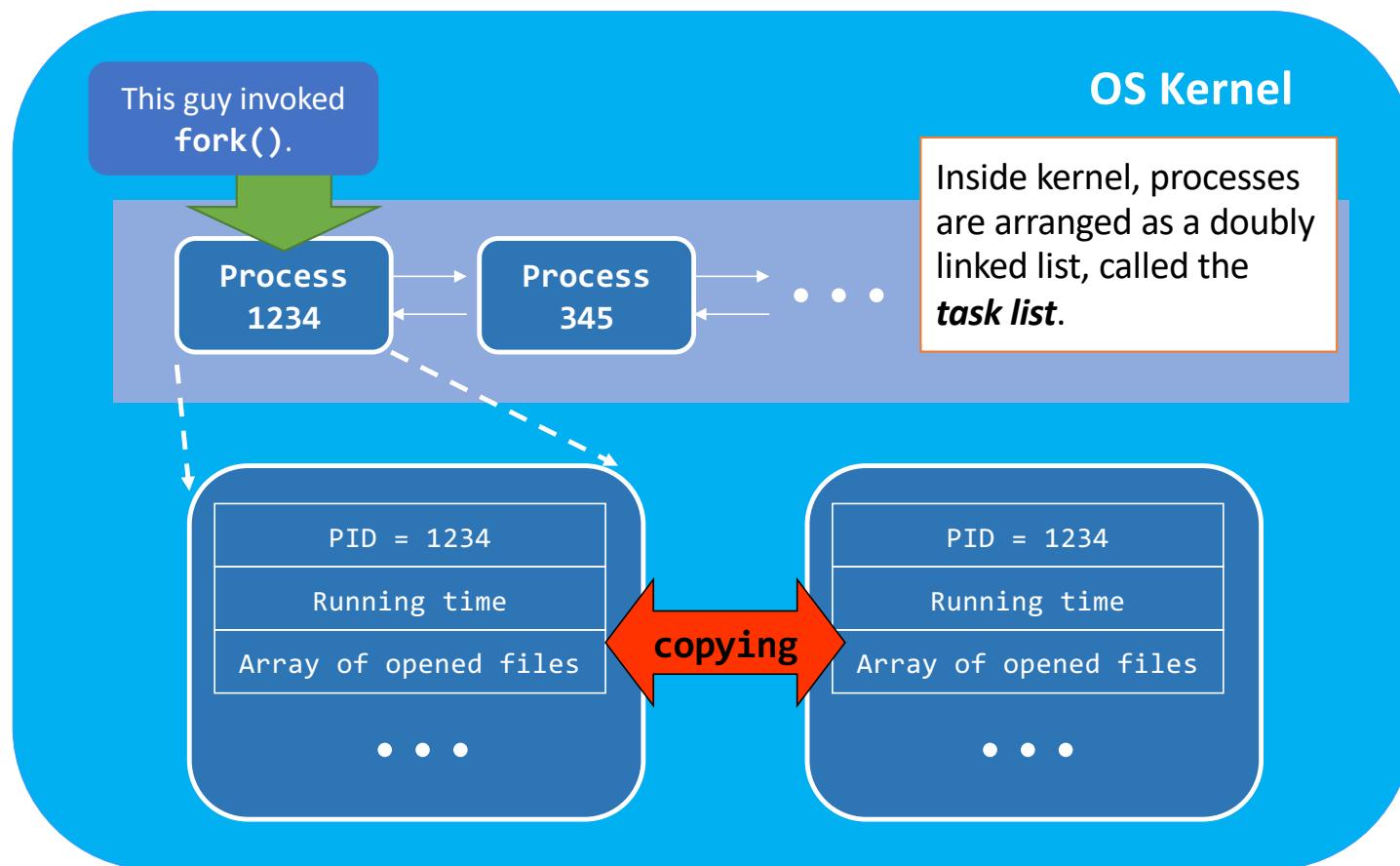
Recall: fork(), exec(), and wait()



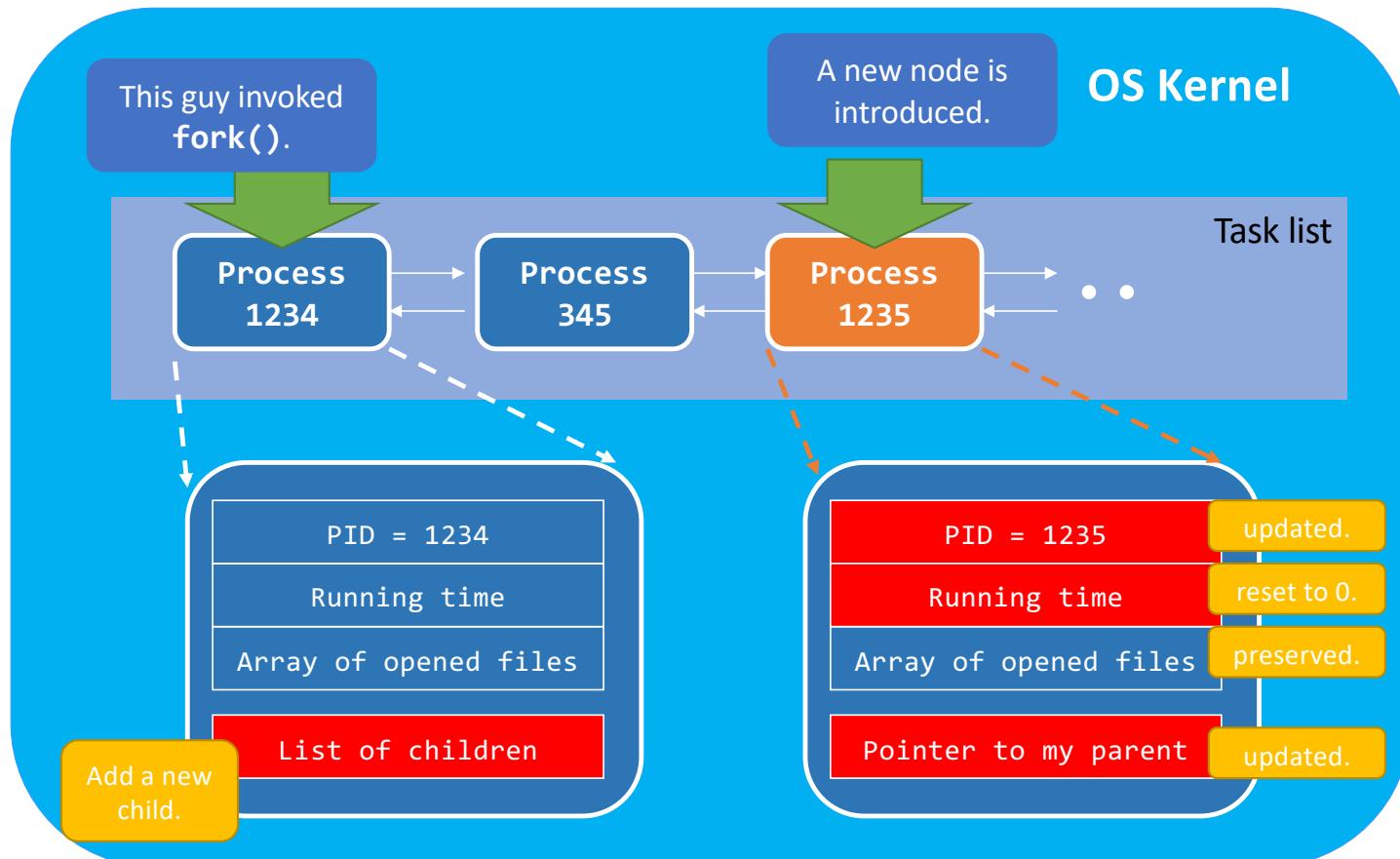
Fork() in User Mode



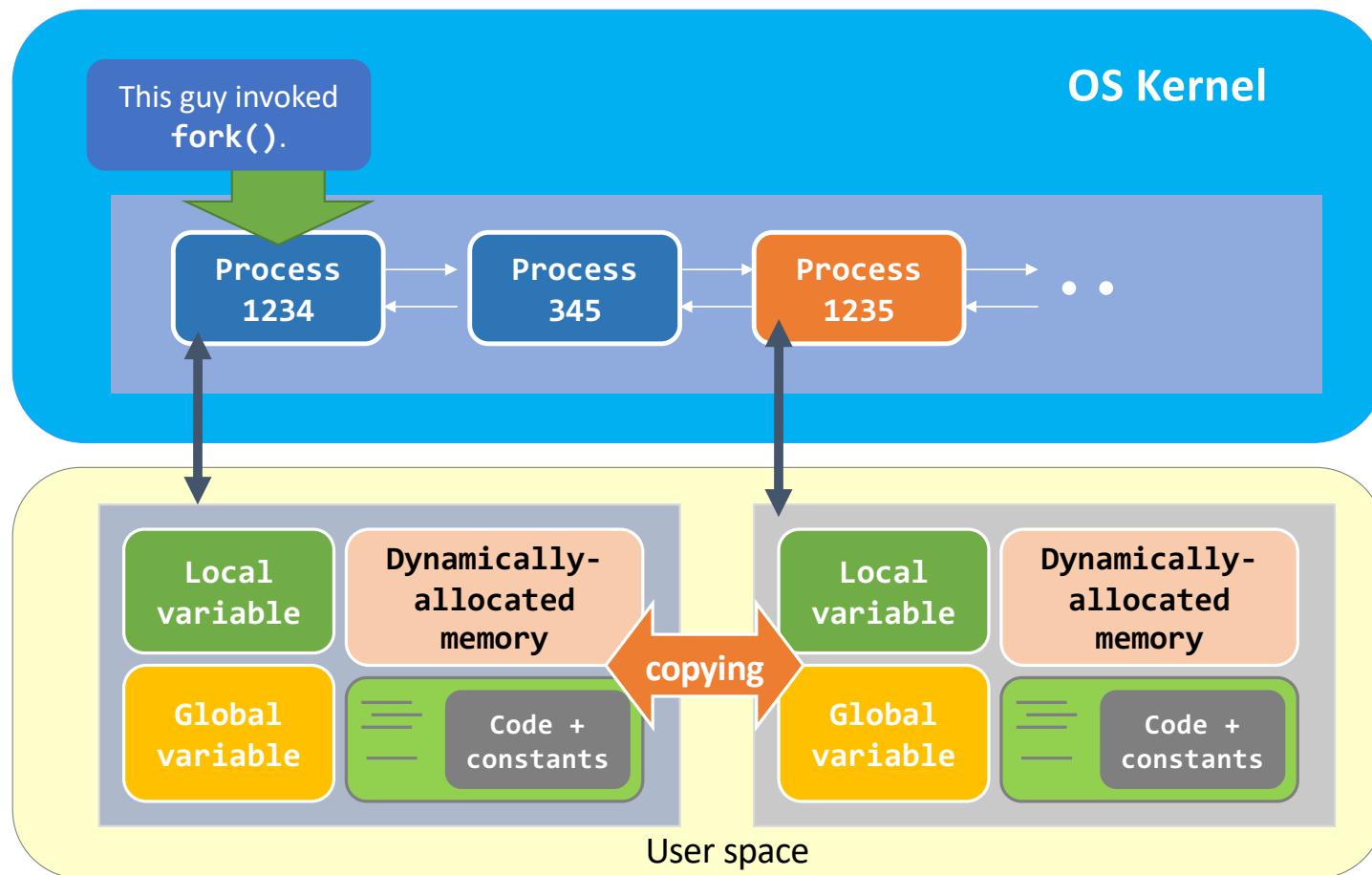
fork(): Kernel View



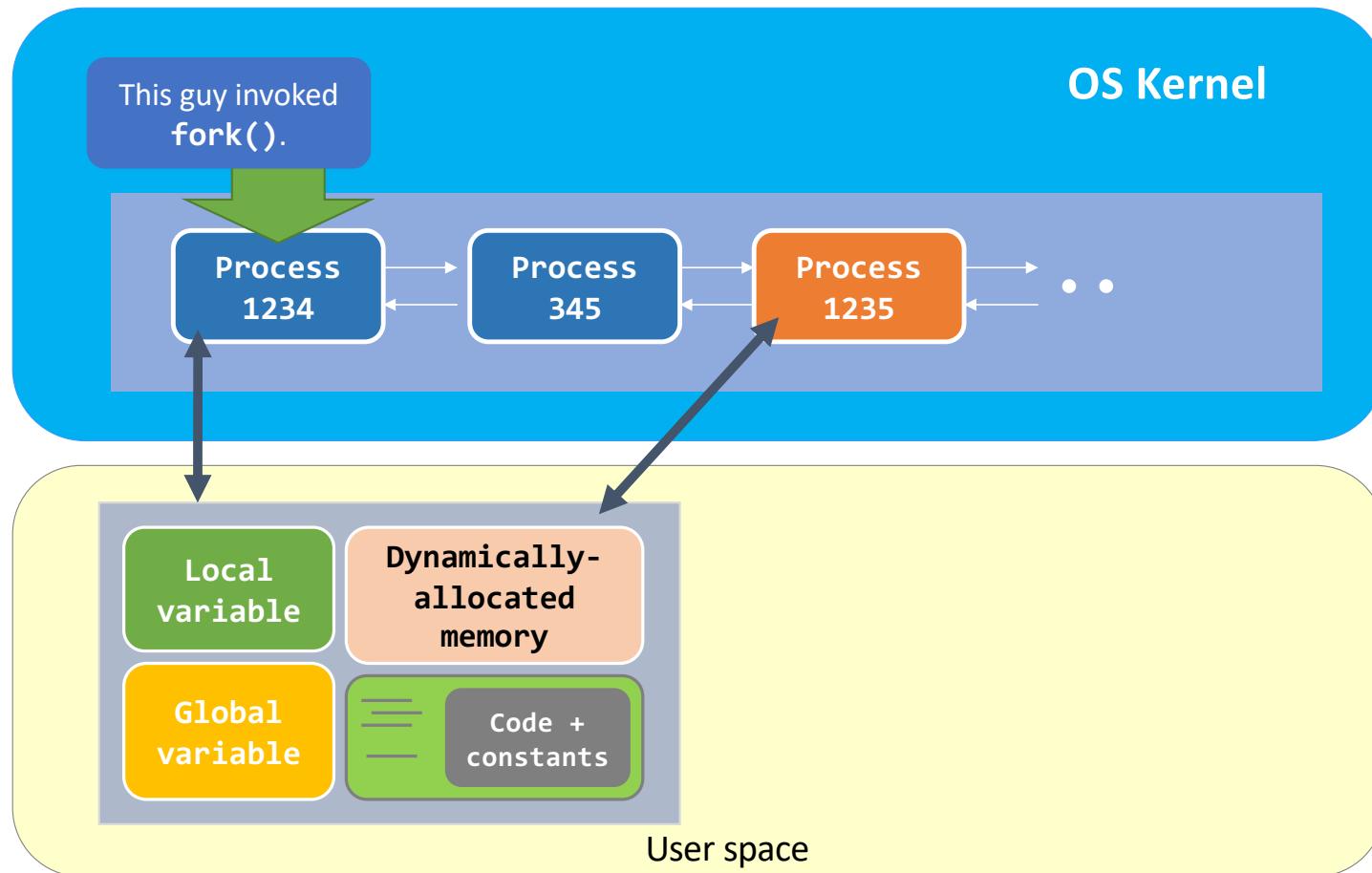
fork(): Kernel View



Case 1: Duplicate Address Space

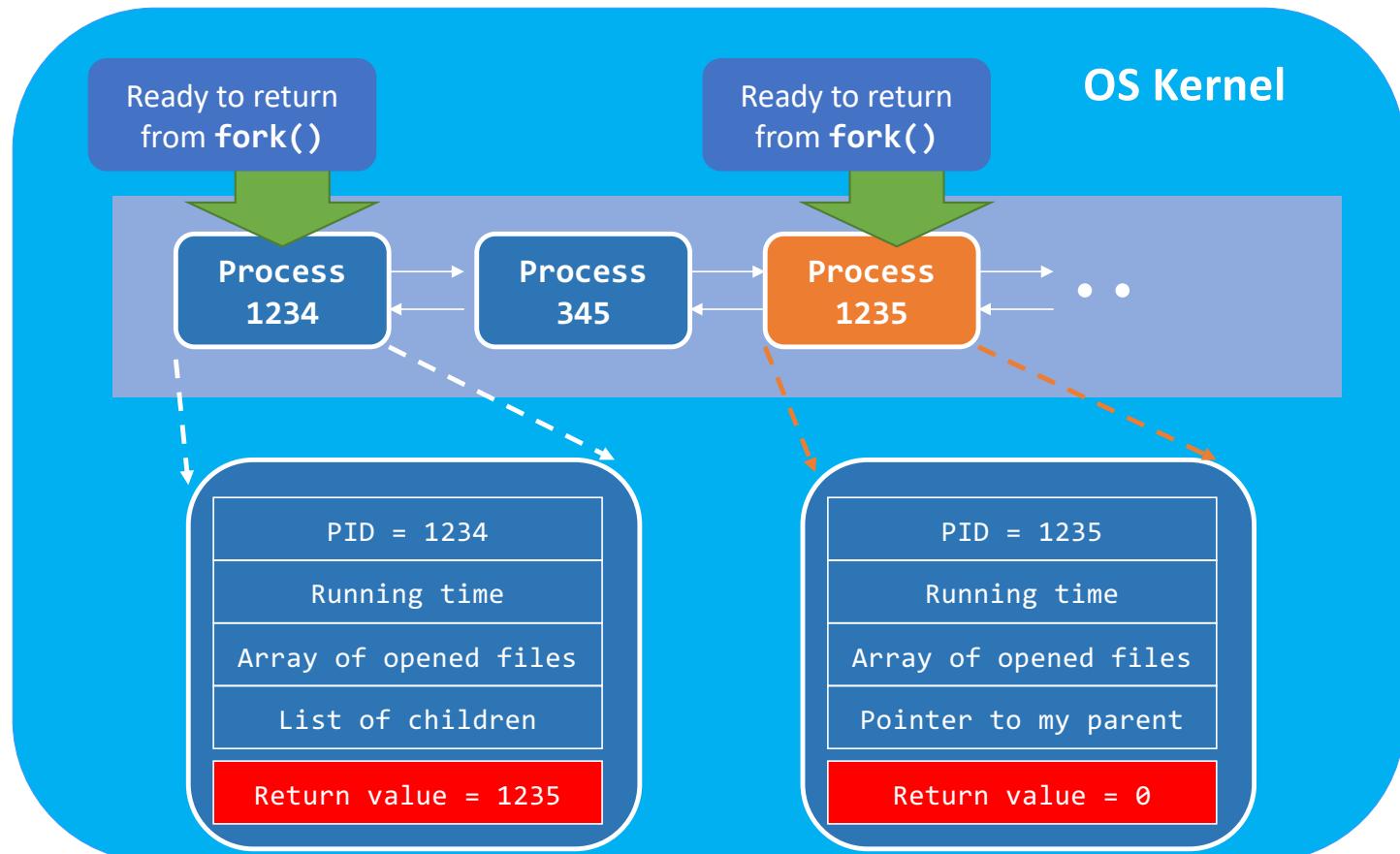


Case 2: Copy on Write



每次需要写才 copy 出新内存空间。

fork(): Kernel View



fork(): Opened Files

- Array of opened files contains:

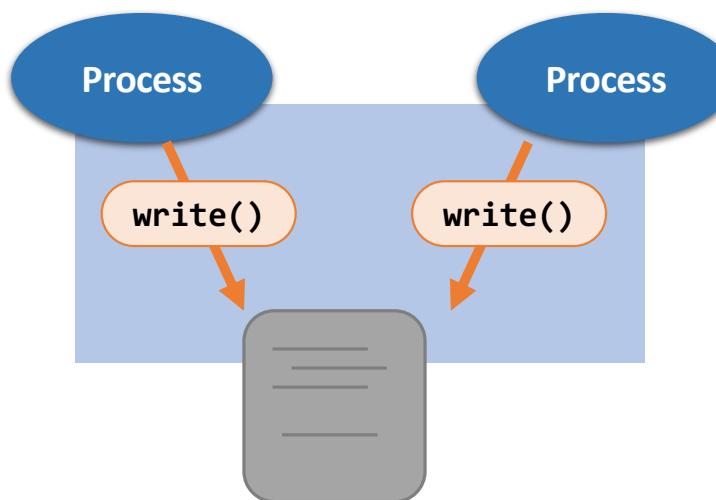
Array Index	Description
0	Standard Input Stream; FILE *stdin;
1	Standard Output Stream; FILE *stdout;
2	Standard Error Stream; FILE *stderr;
3 or beyond	Storing the files you opened, e.g., fopen() , open() , etc.

3以上 : 真実文件 index

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

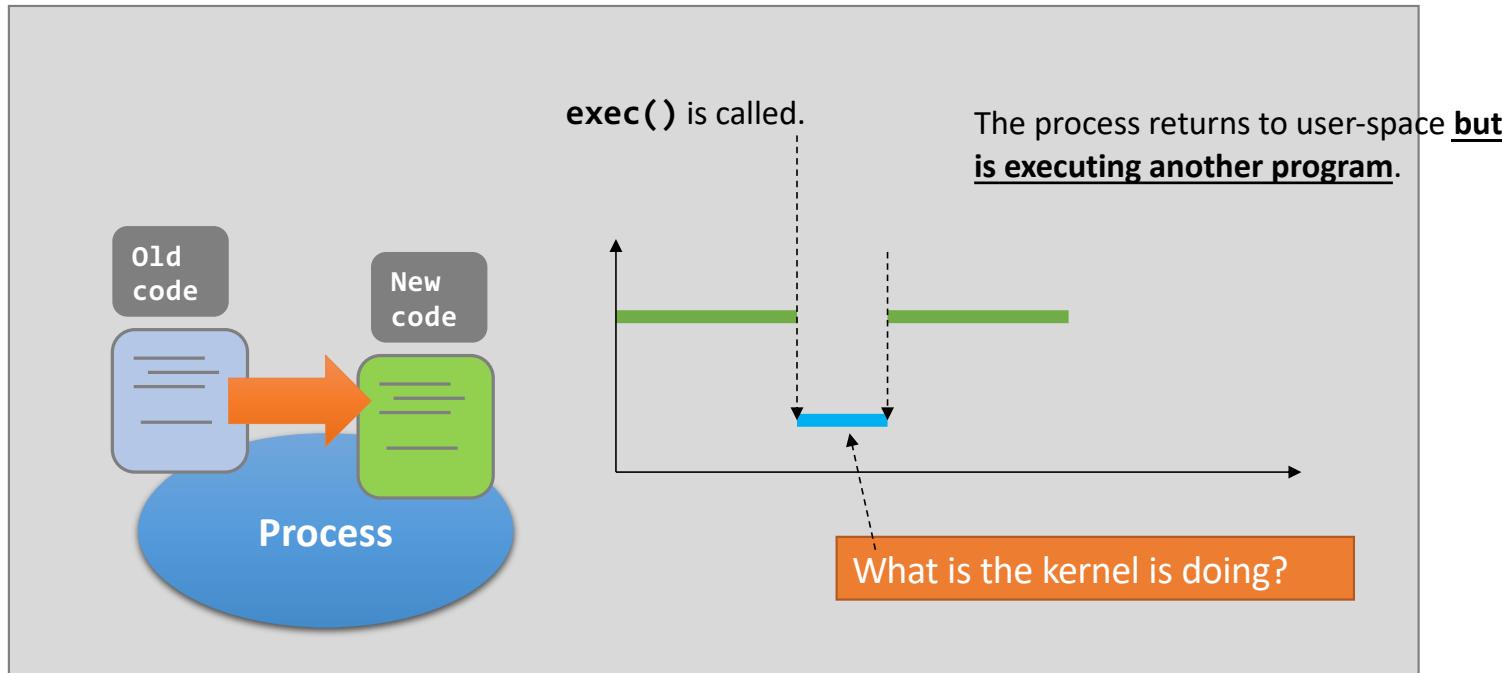
fork(): Opened Files

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

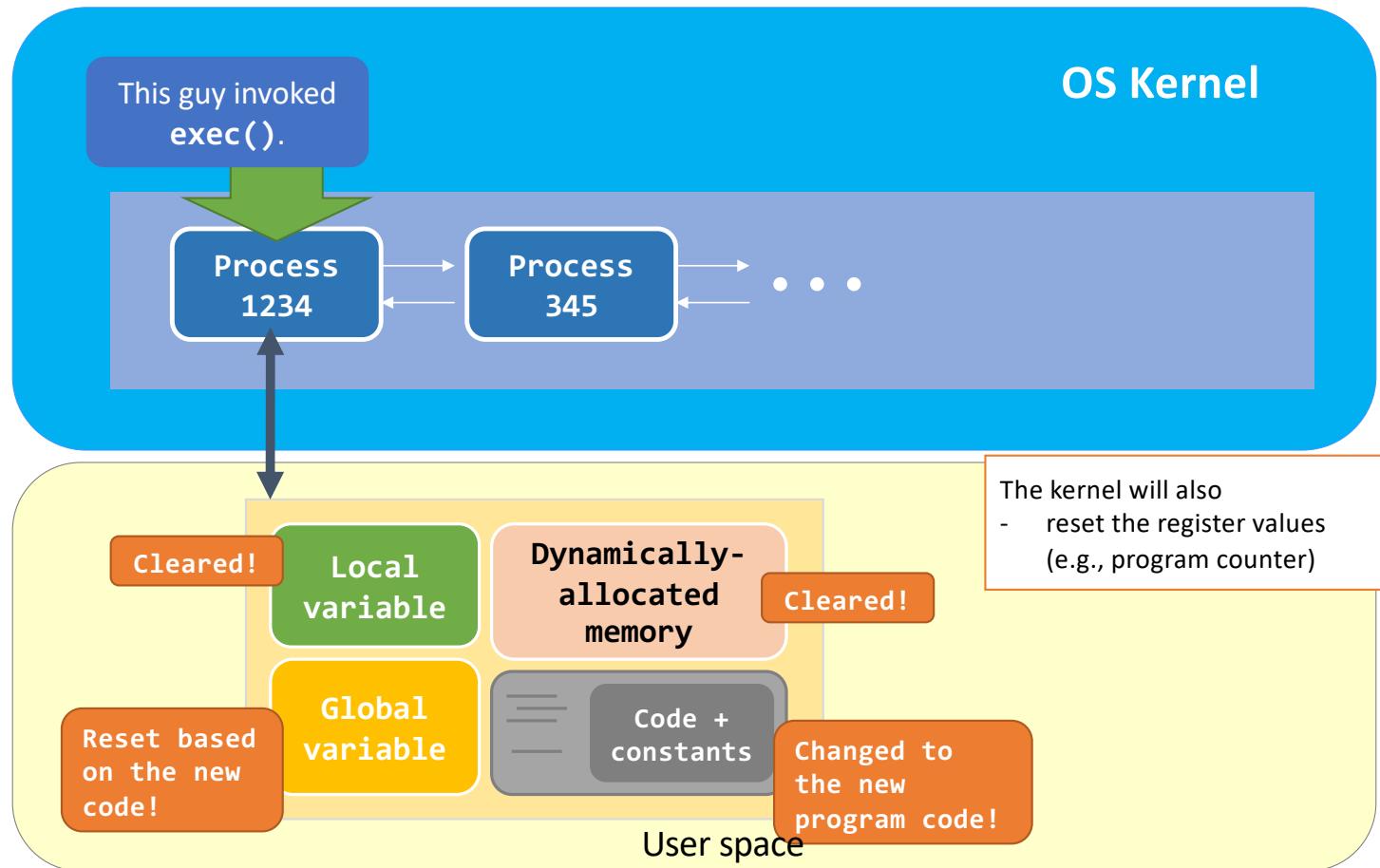


Let's see what will happen when the program finishes running!

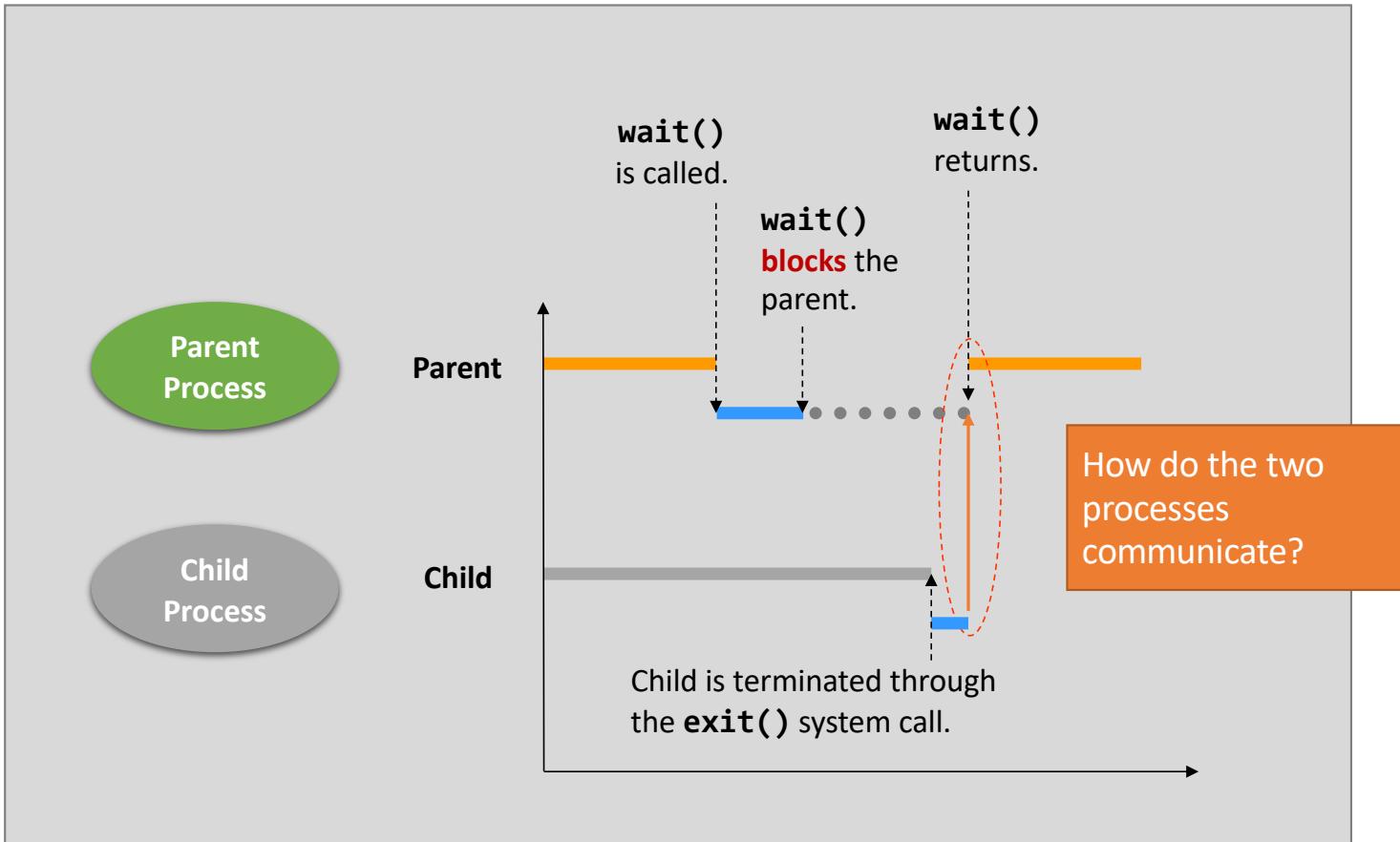
exec() in the User Mode



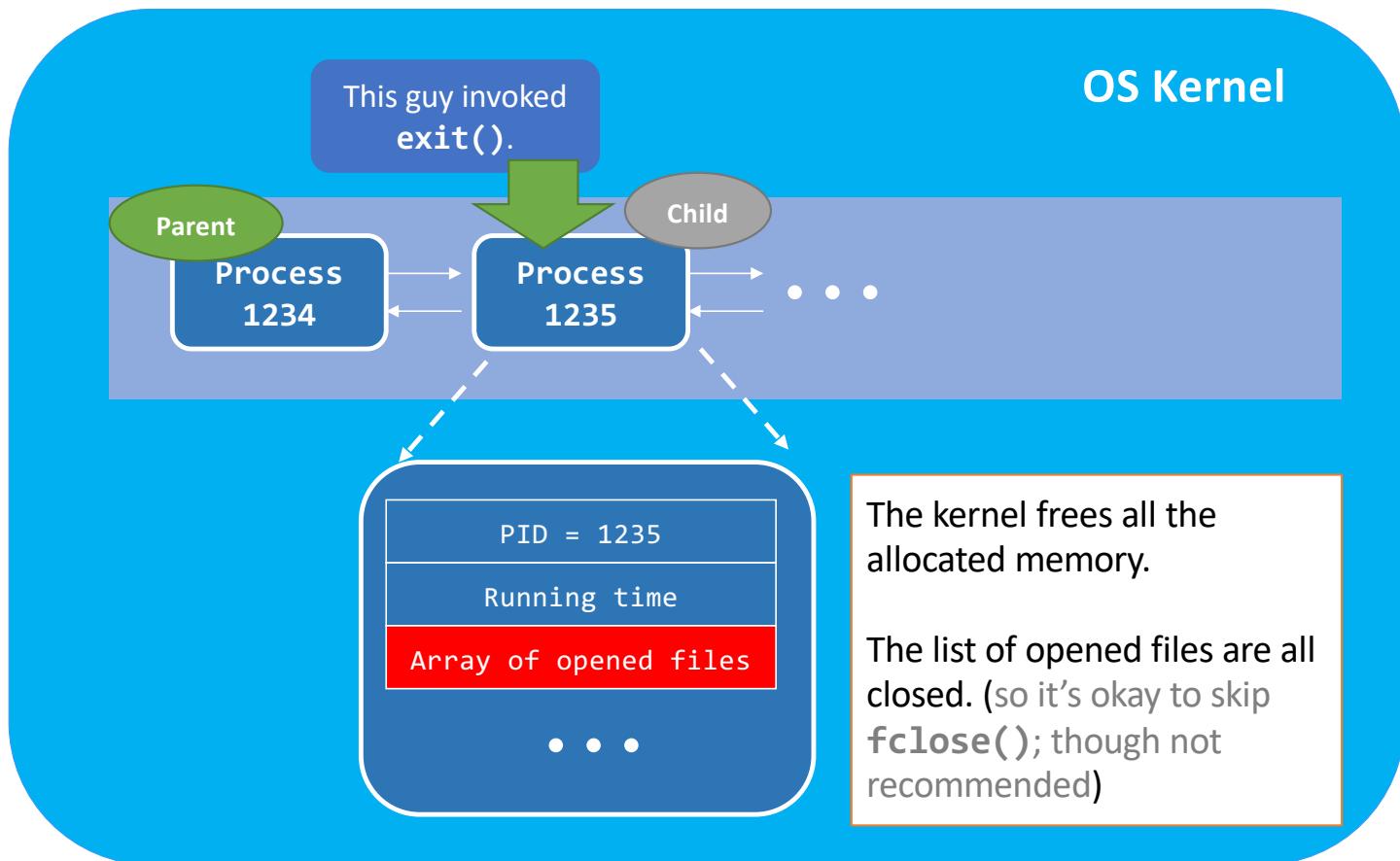
exec(): Kernel View



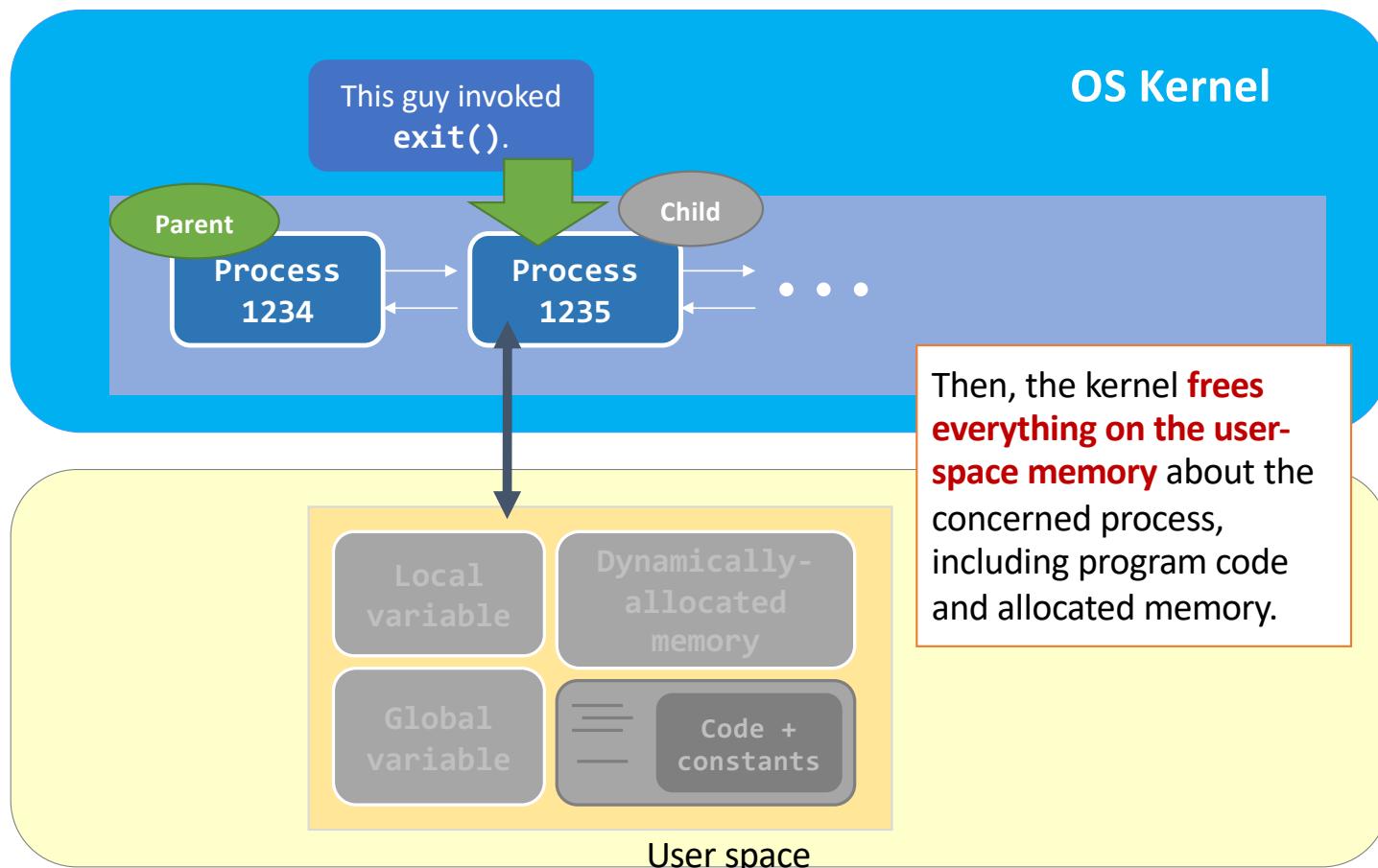
`wait()` and `exit()`



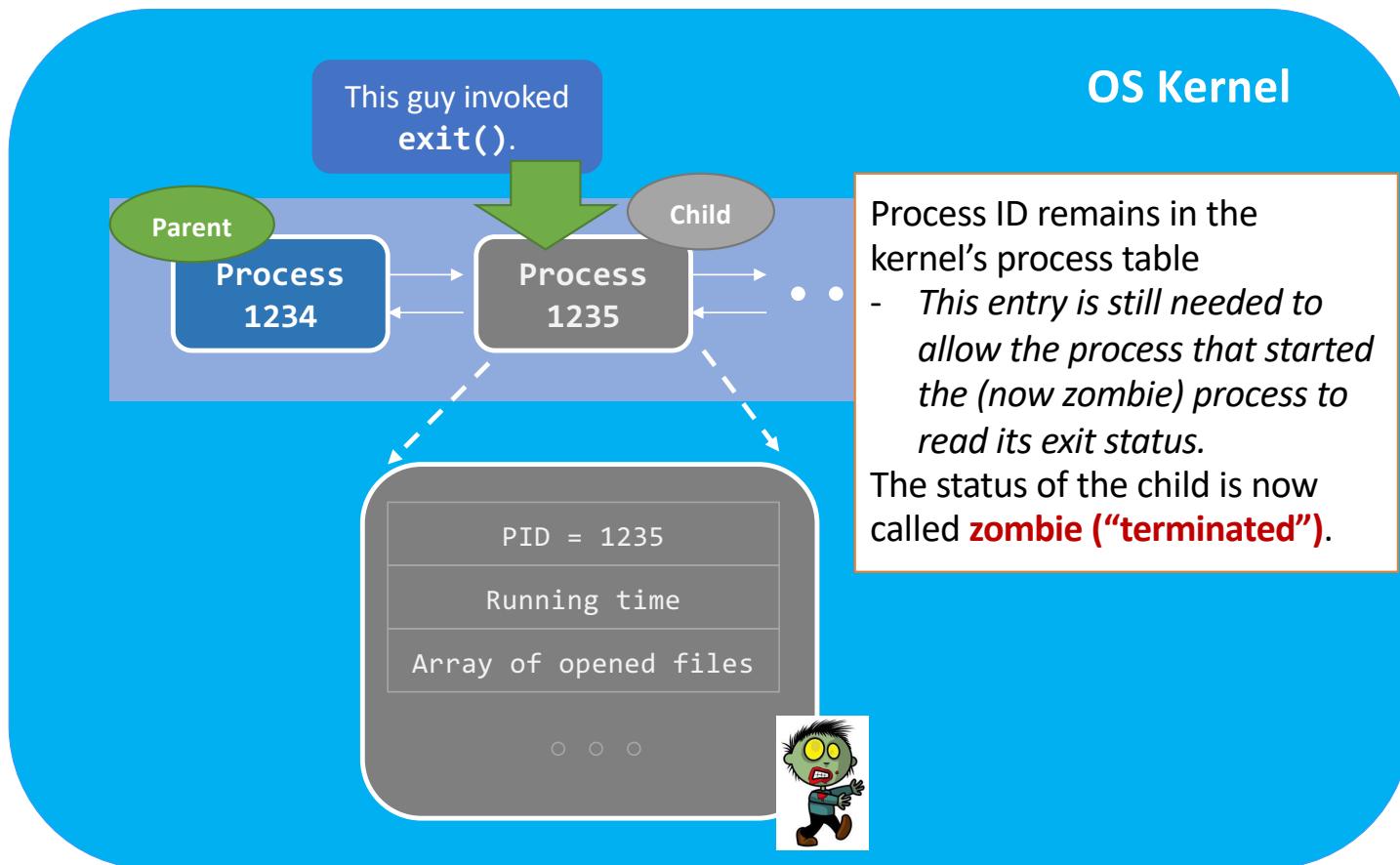
exit(): Kernel View



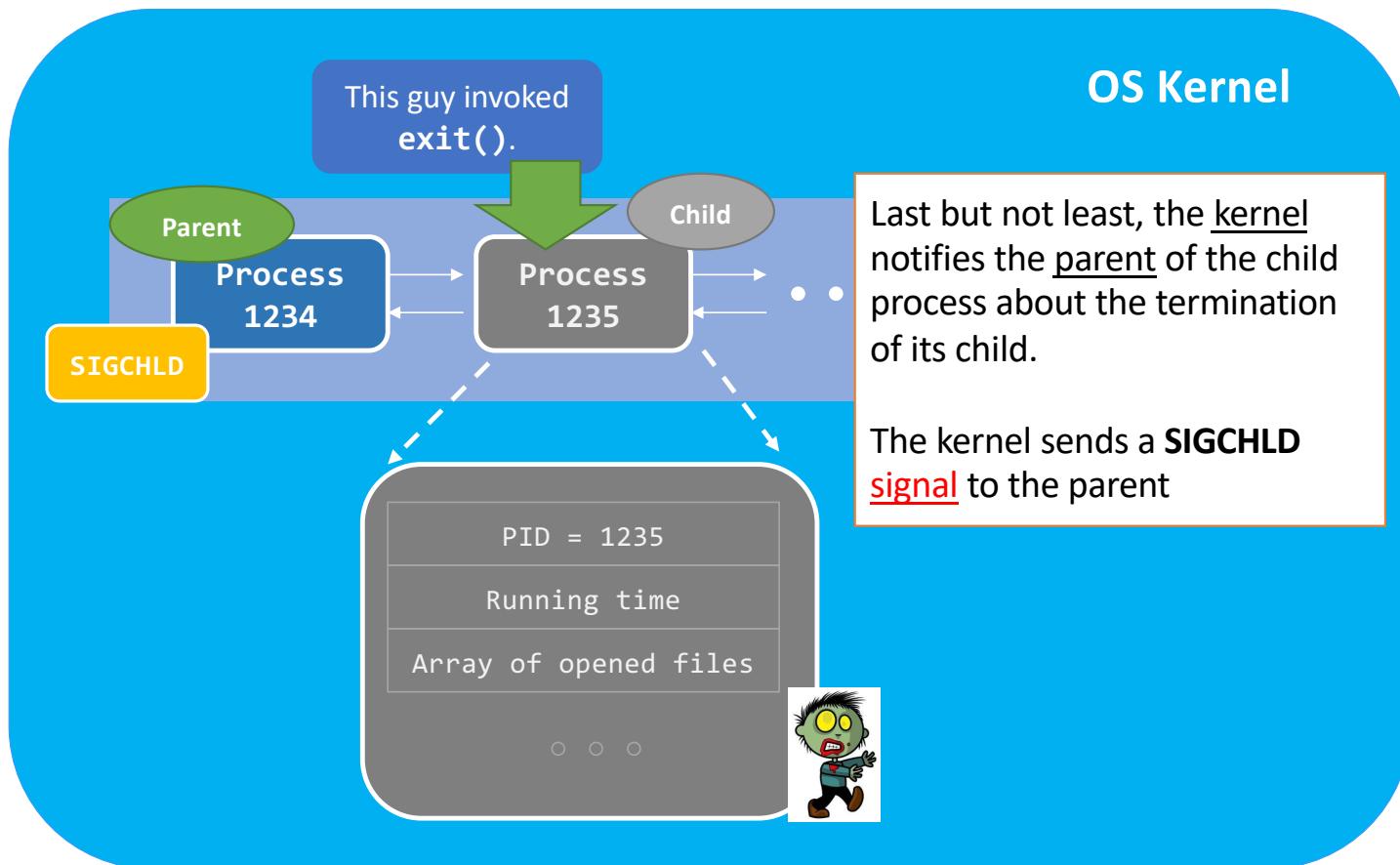
exit(): Kernel View



exit(): Kernel View



exit(): Kernel View

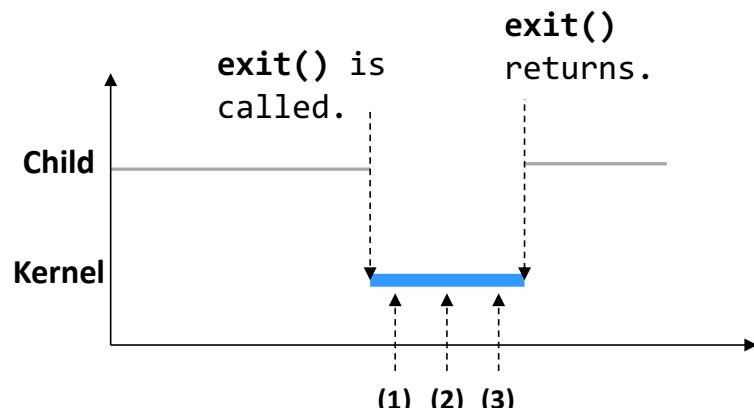


exit(): Summary

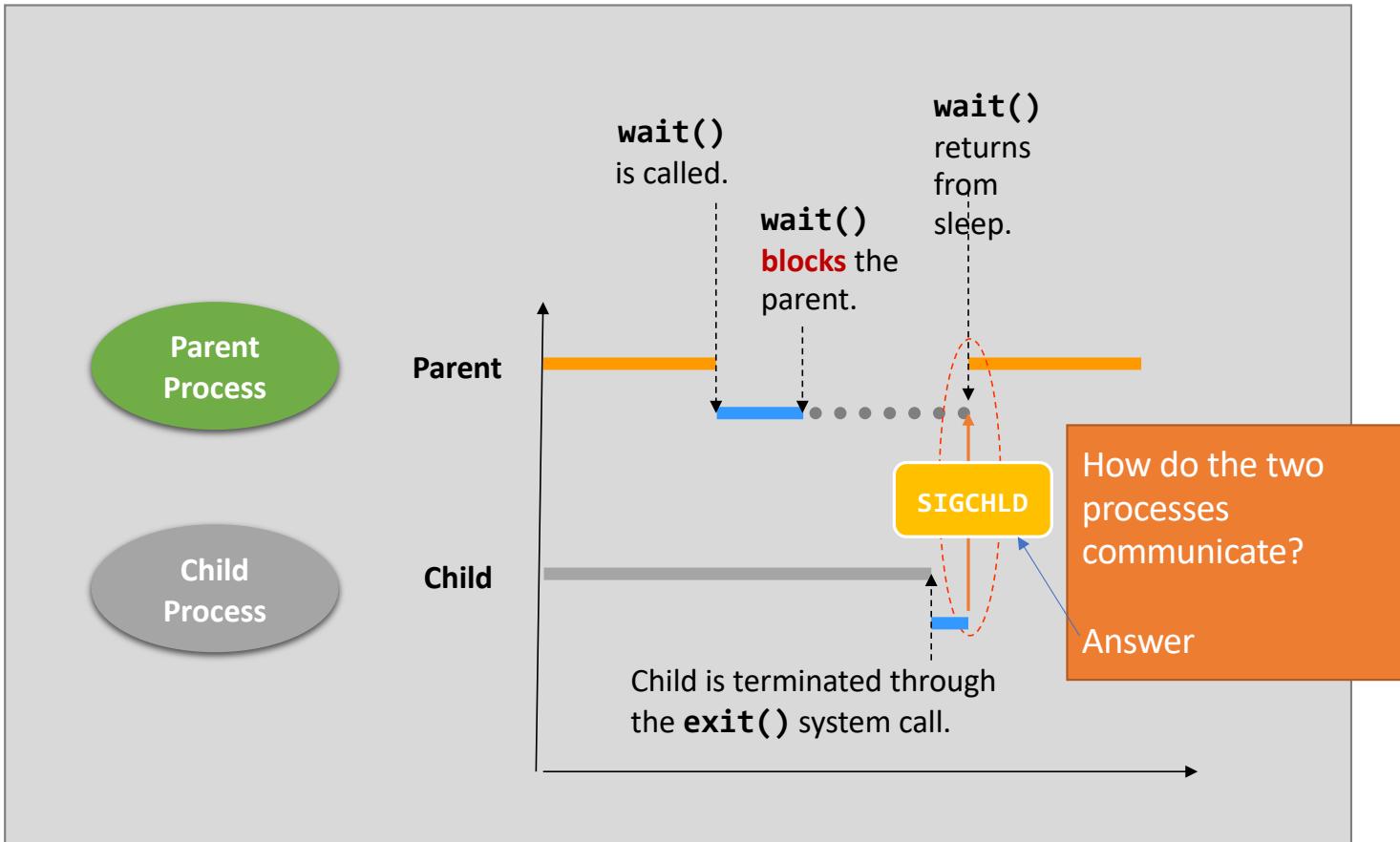
Step (1) Clean up most of the allocated kernel-space memory
(e.g., process's running time info).

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

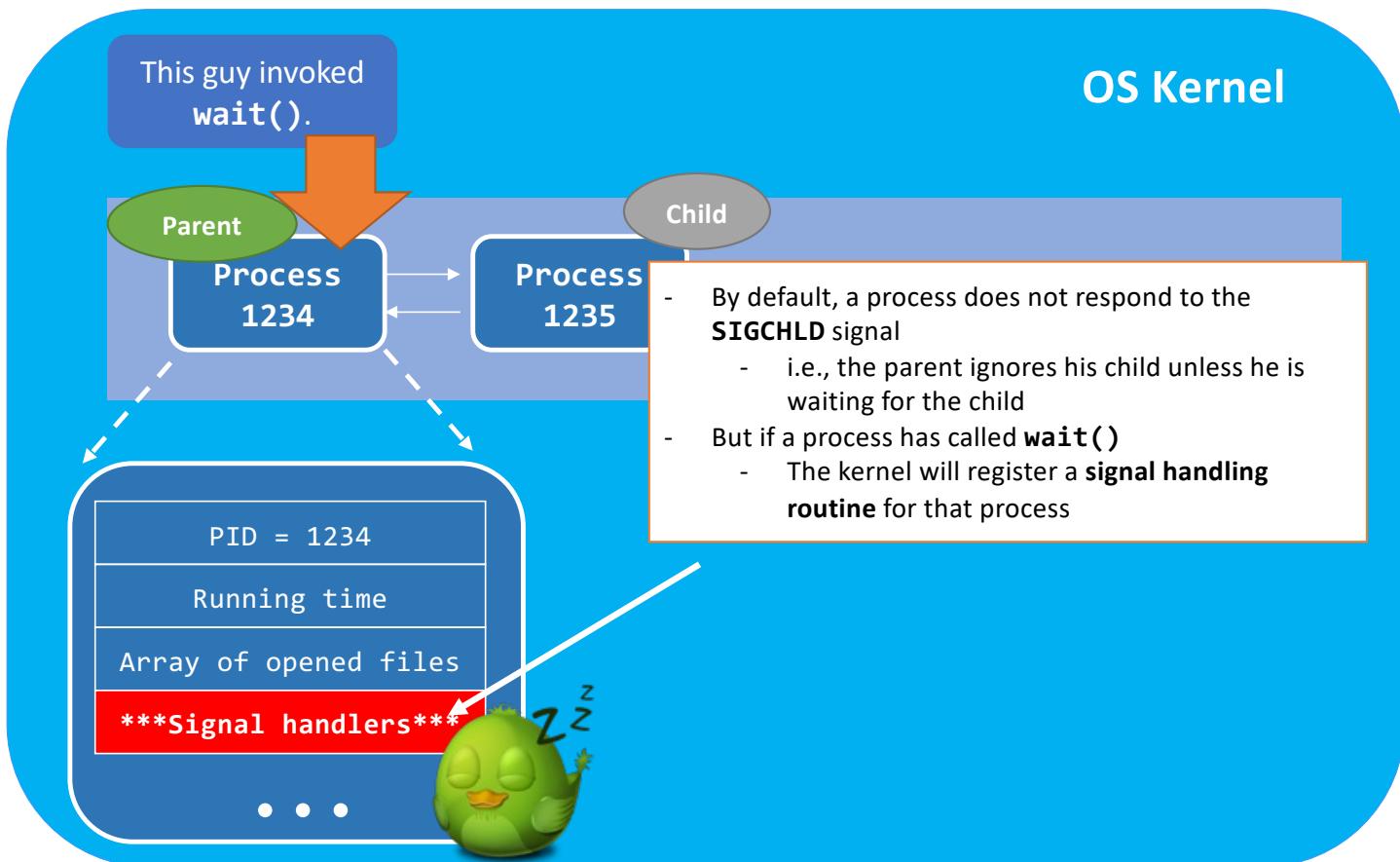
Step (3) Notify the parent with SIGCHLD.



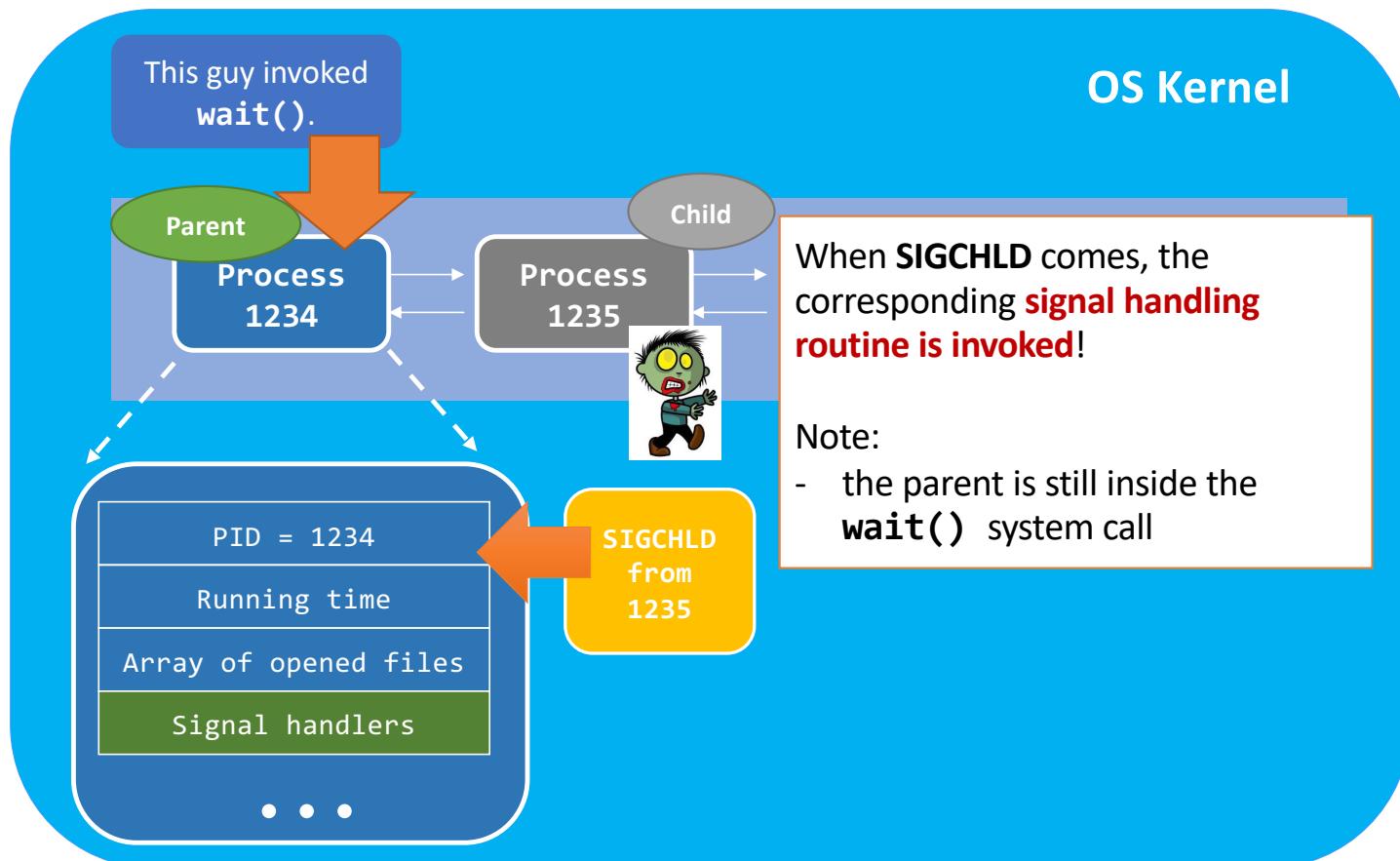
`wait()` and `exit()`



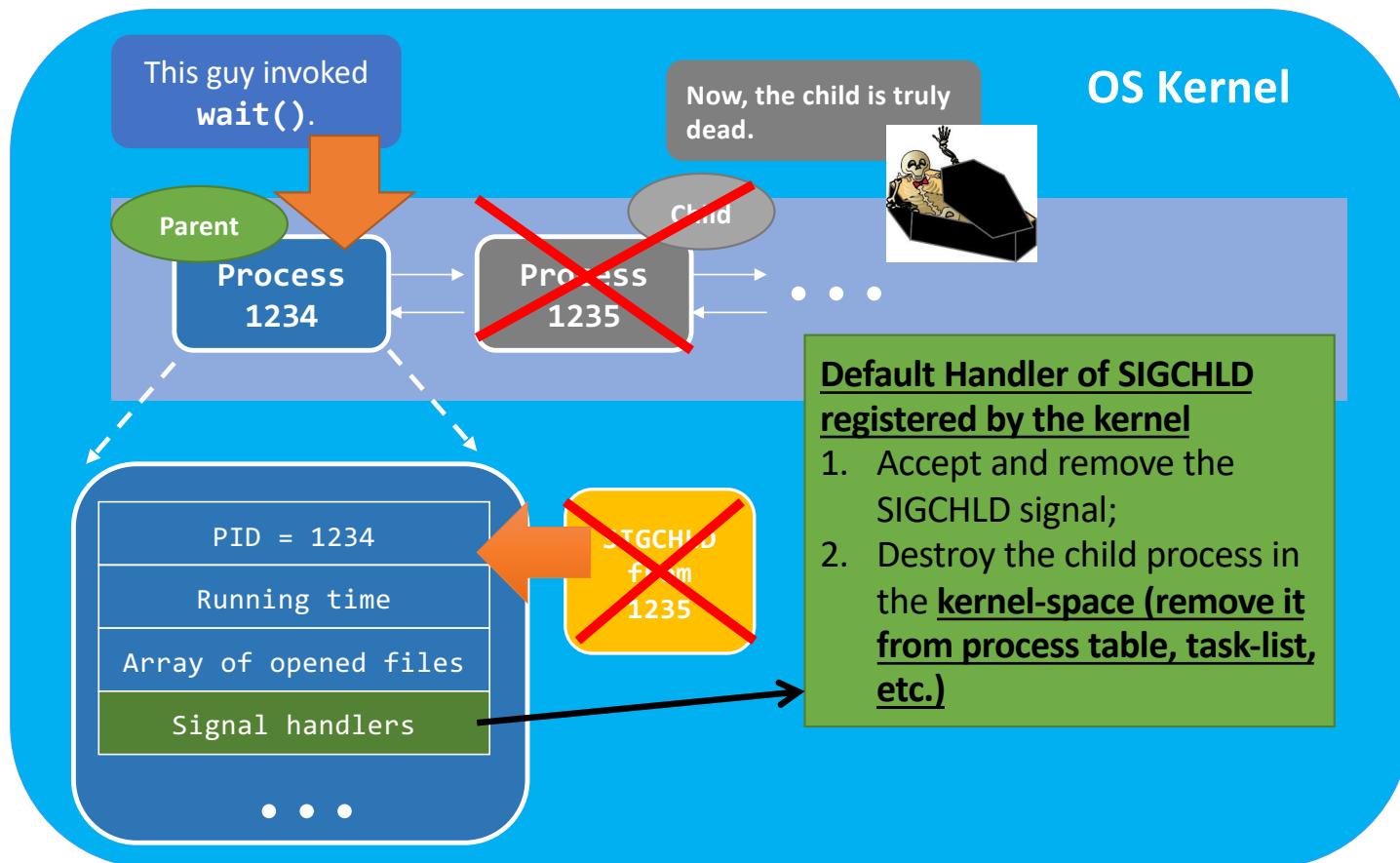
wait() Kernel View



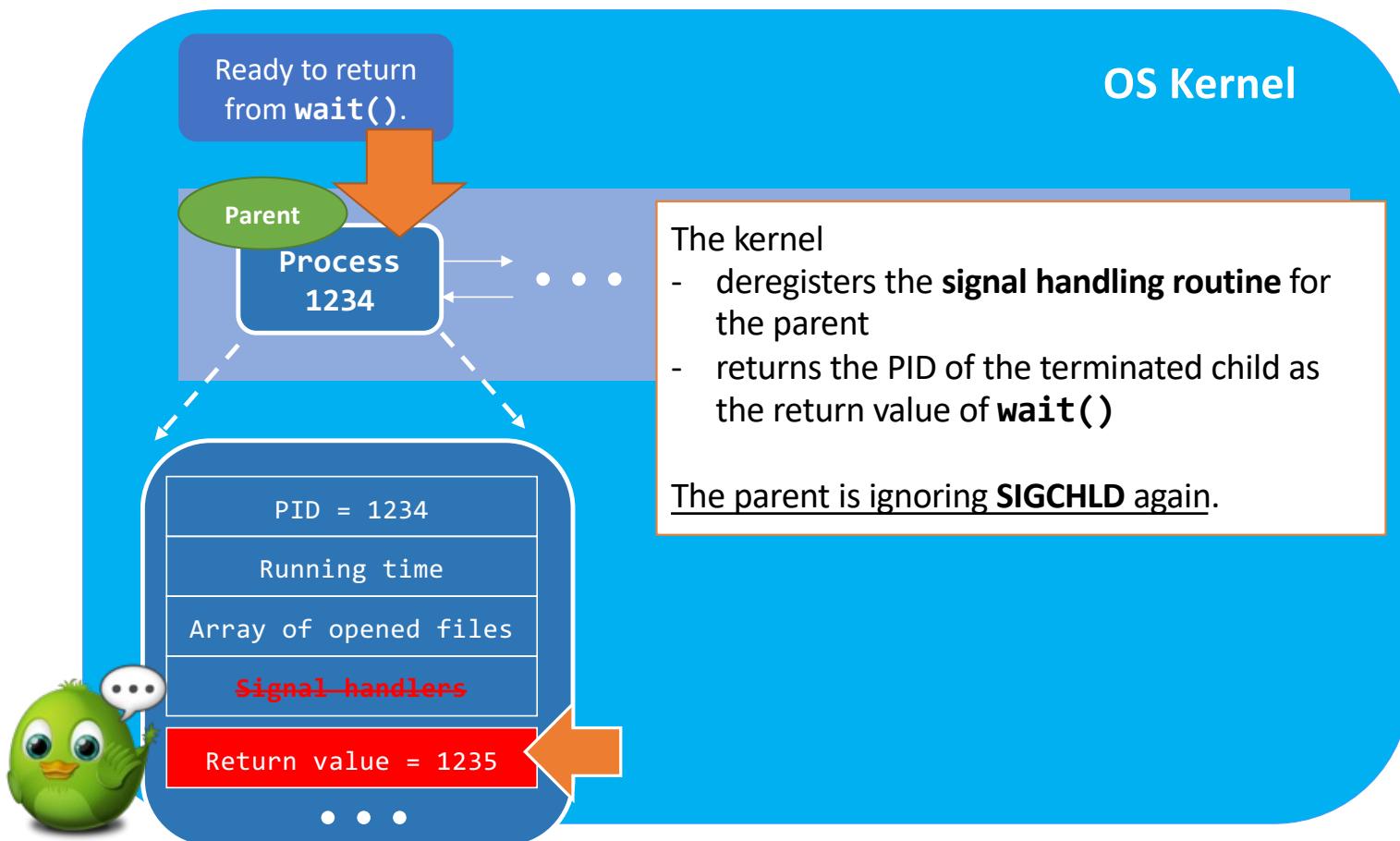
wait() Kernel View



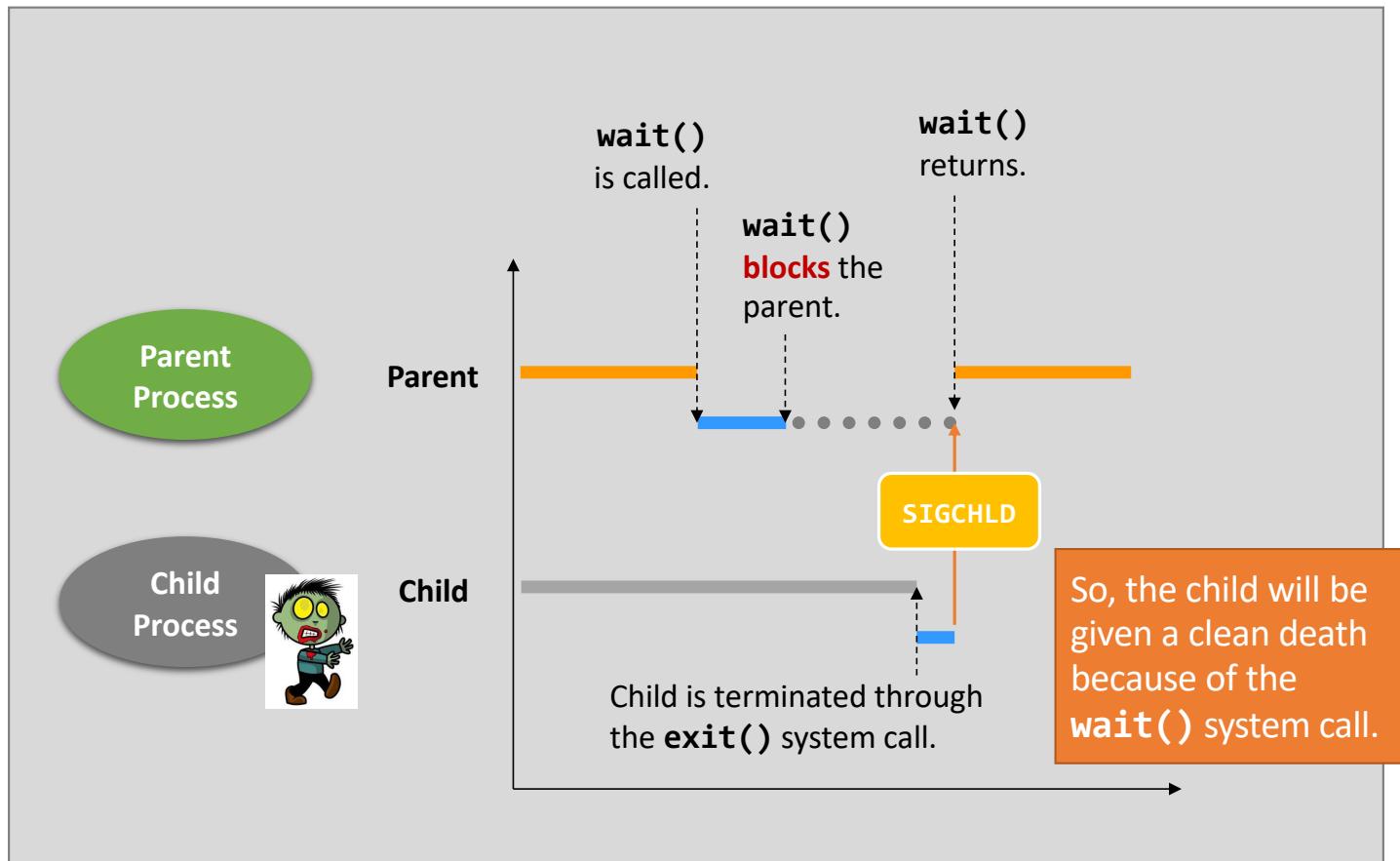
wait() Kernel View



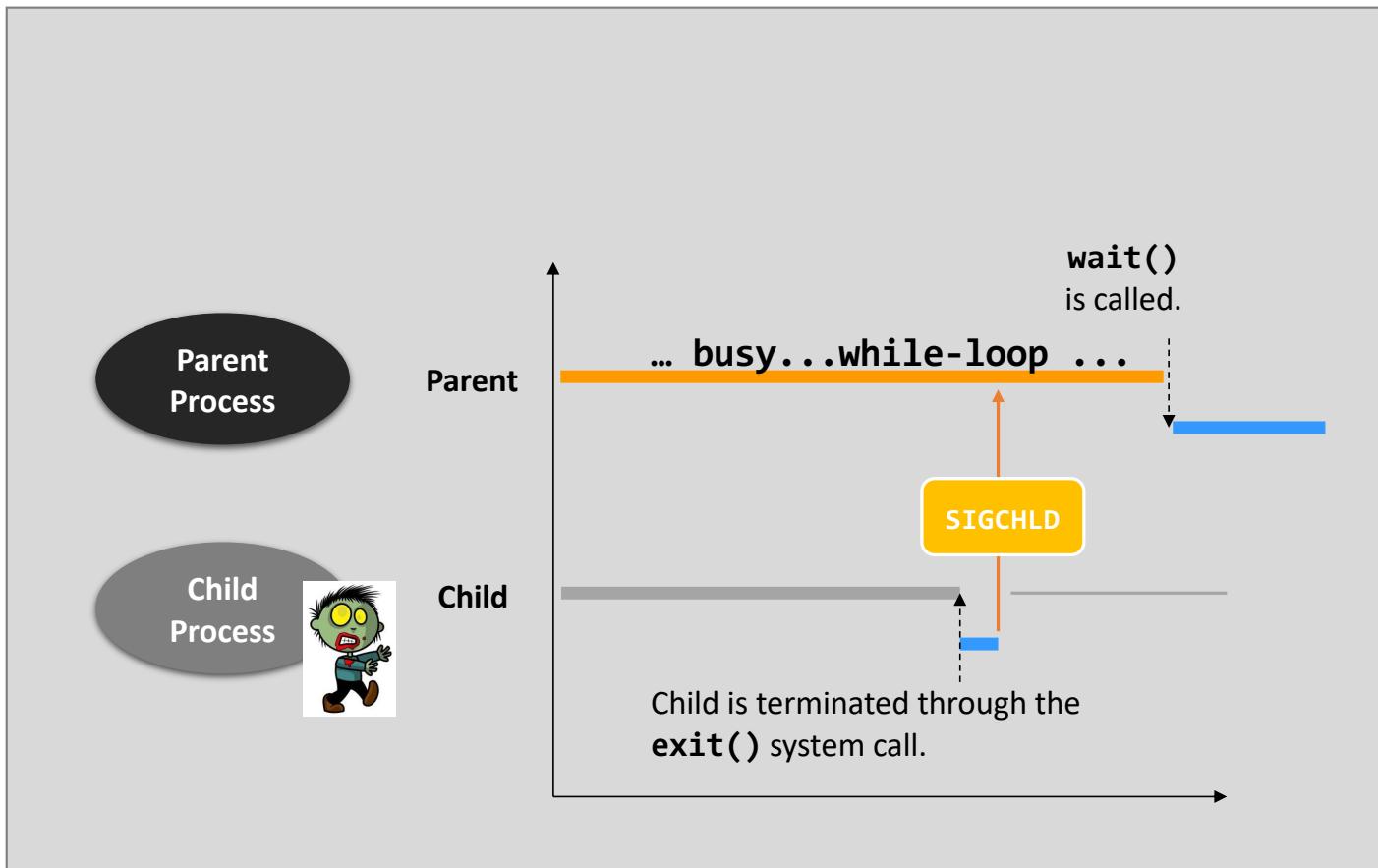
wait() Kernel View



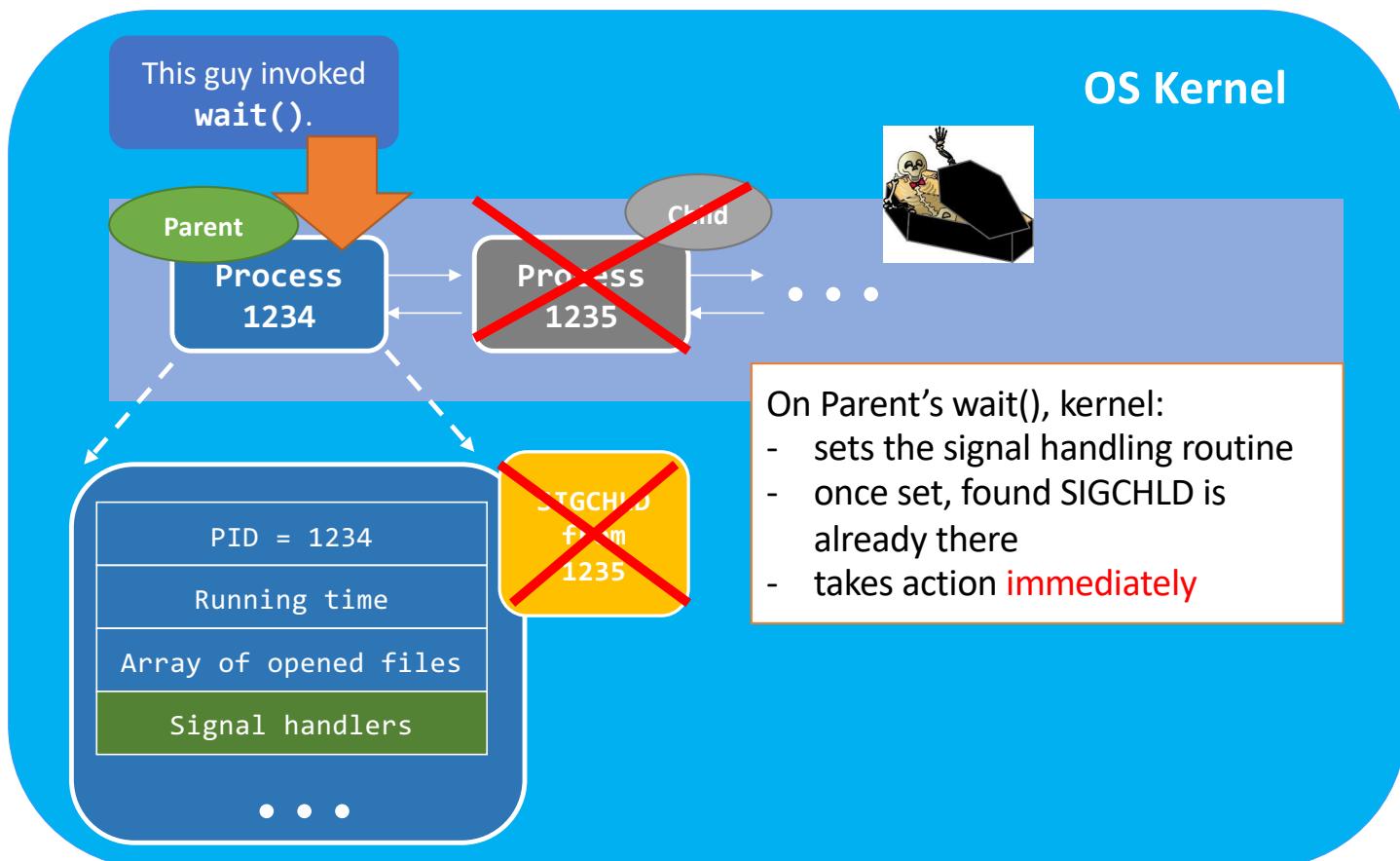
Normal Case



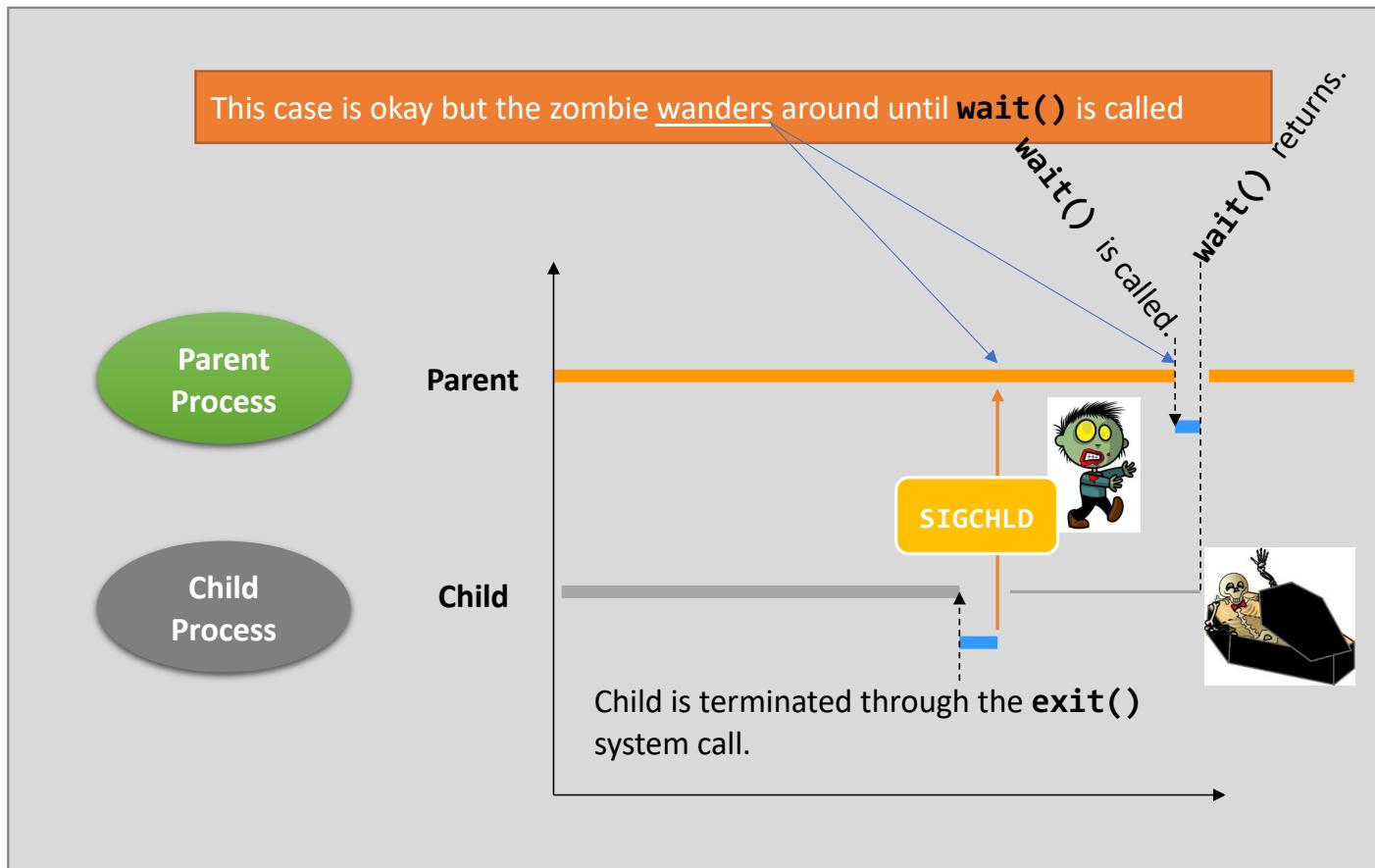
Parent's `wait()` after Child's `exit()`



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()

```
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 }
```

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

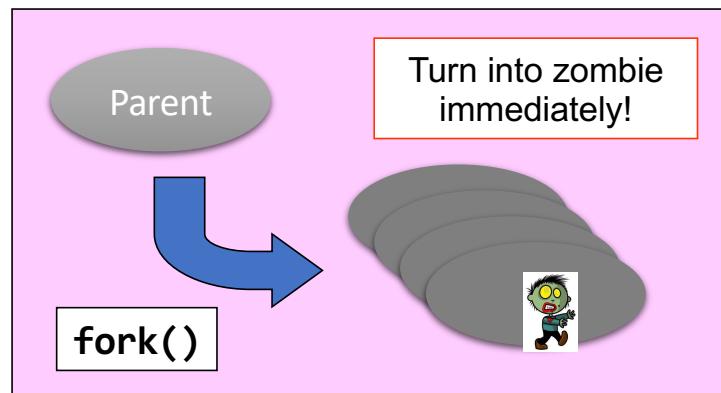
You are expected to see **the status of the child process changes (ps aux [PID])** between the 1st and the 2nd “enter”.

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

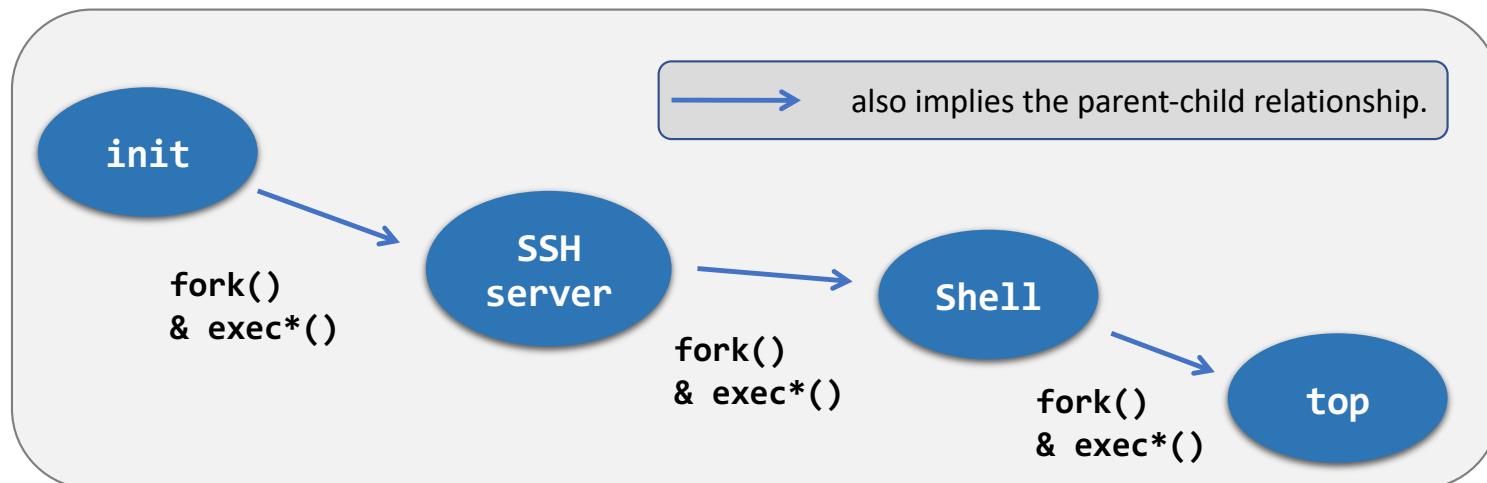
The first process

- 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?

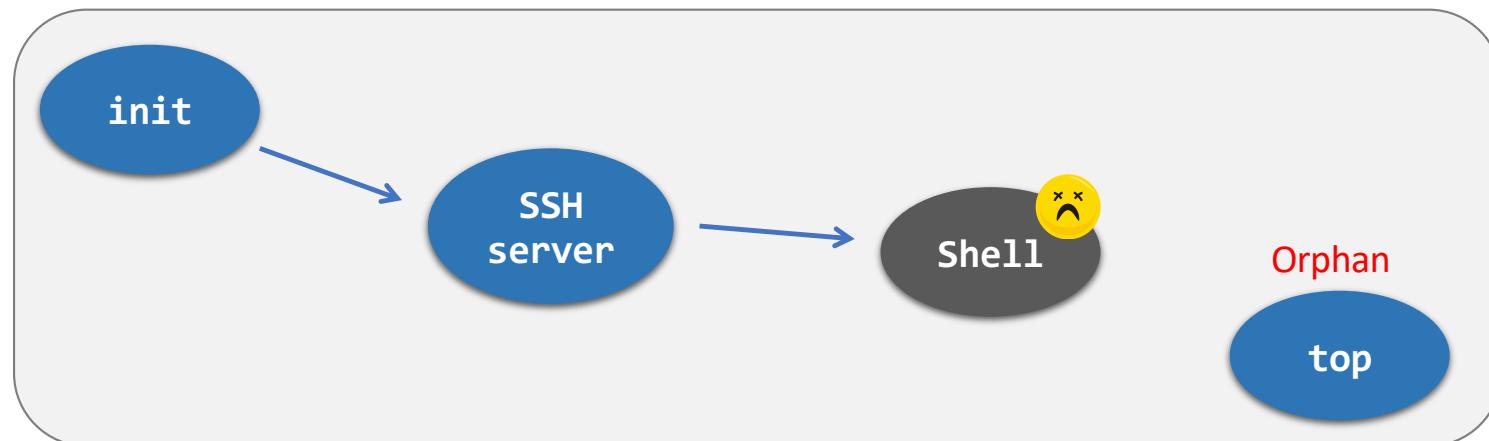
A Tree of Processes

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



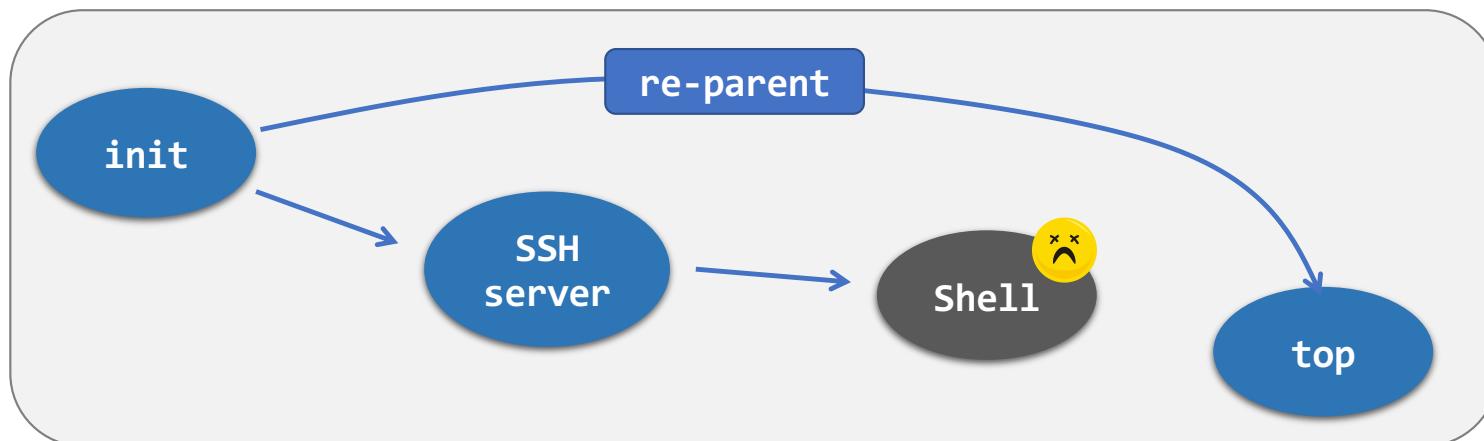
Orphans

- 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.



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

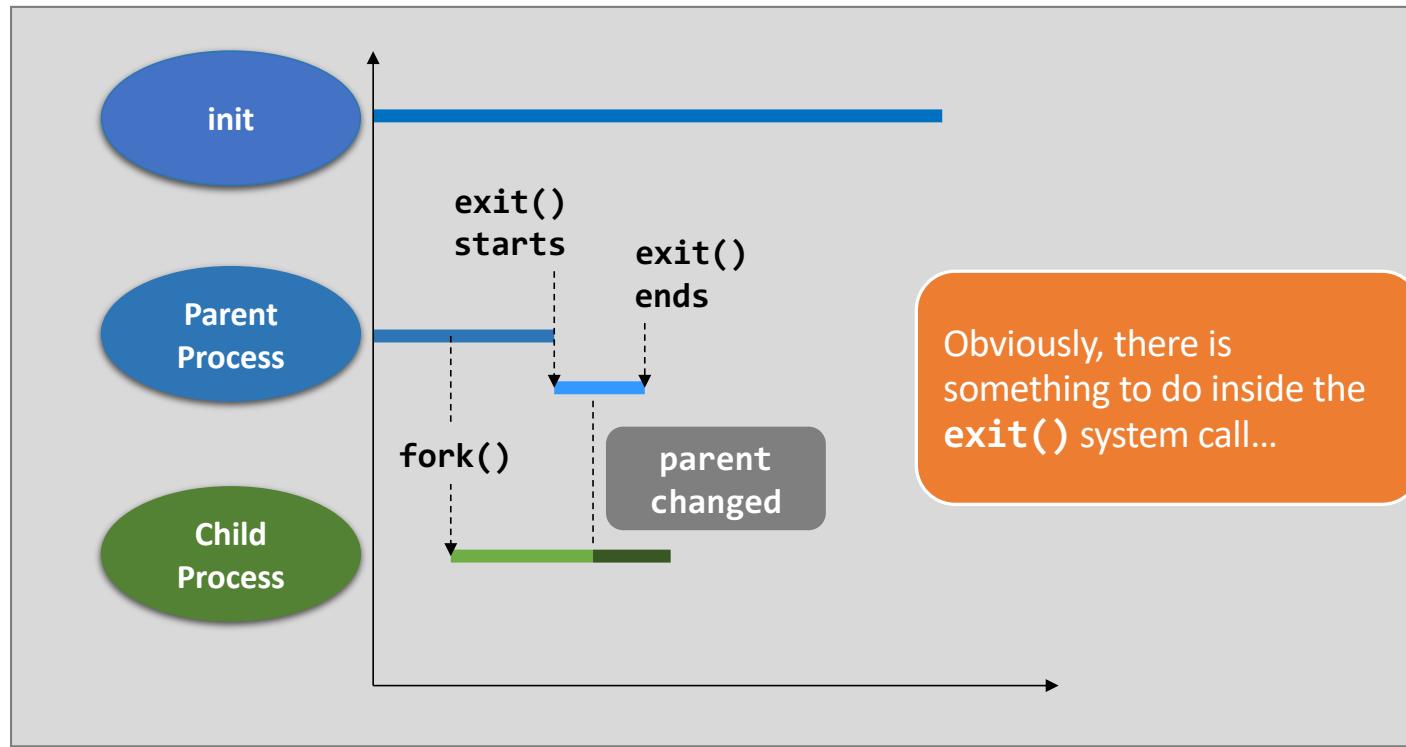
```
1 int main(void) {
2     int i;
3     if(fork() == 0) {
4         for(i = 0; i < 5; i++) {
5             printf("(%) parent's PID = %d\n",
6                     getpid(), getppid());
7             sleep(1);
8         }
9     }
10    else
11        sleep(1);
12    printf("(%) bye.\n", getpid());
13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

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

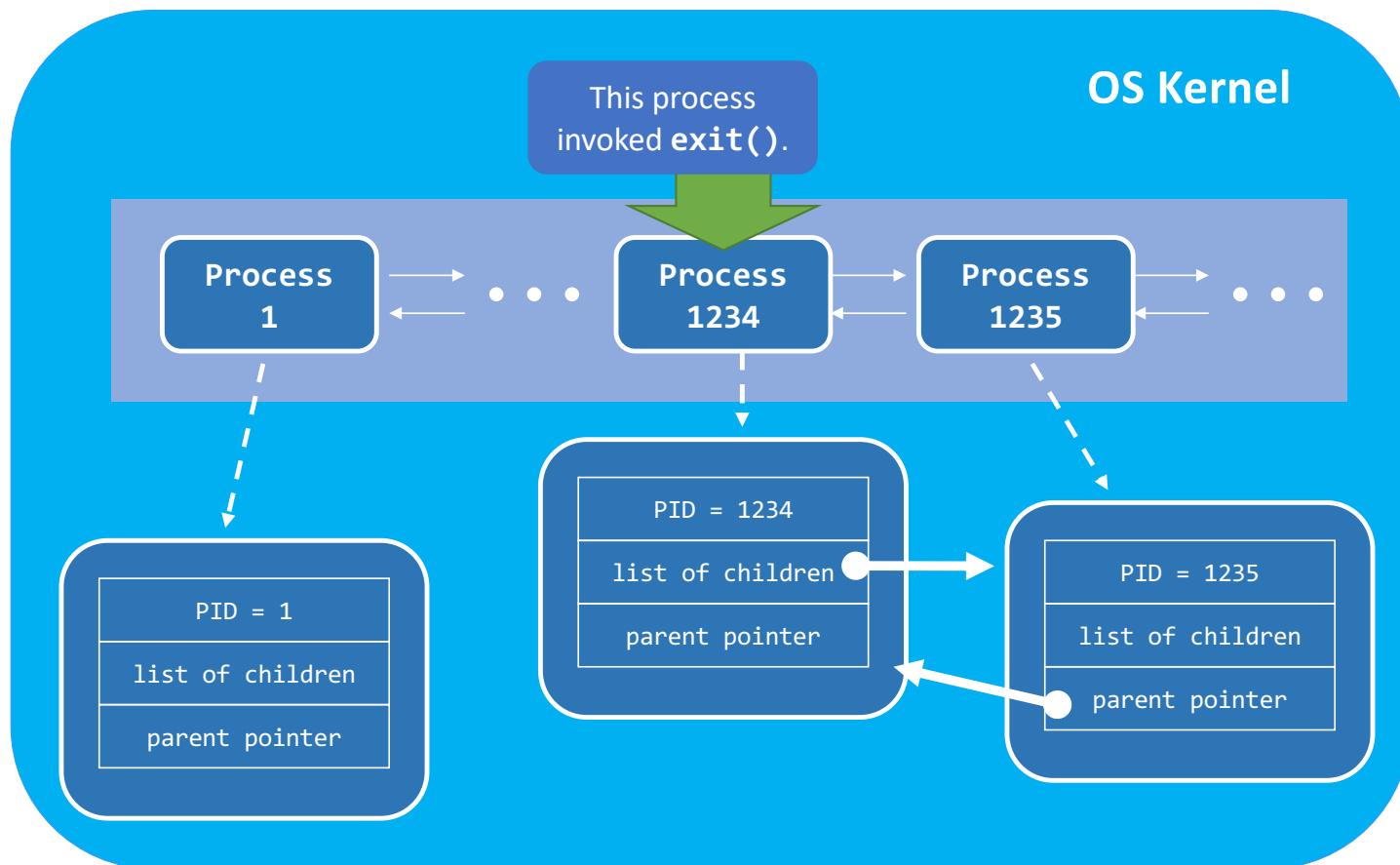
```
$ ./reparent
(1235) parent's PID = 1234
(1235) parent's PID = 1234
(1234) bye.
$ (1235) parent's PID = 1
(1235) parent's PID = 1
(1235) parent's PID = 1
(1235) bye.
$ _
```

→ parent
is gestorben.
} die für reparenting

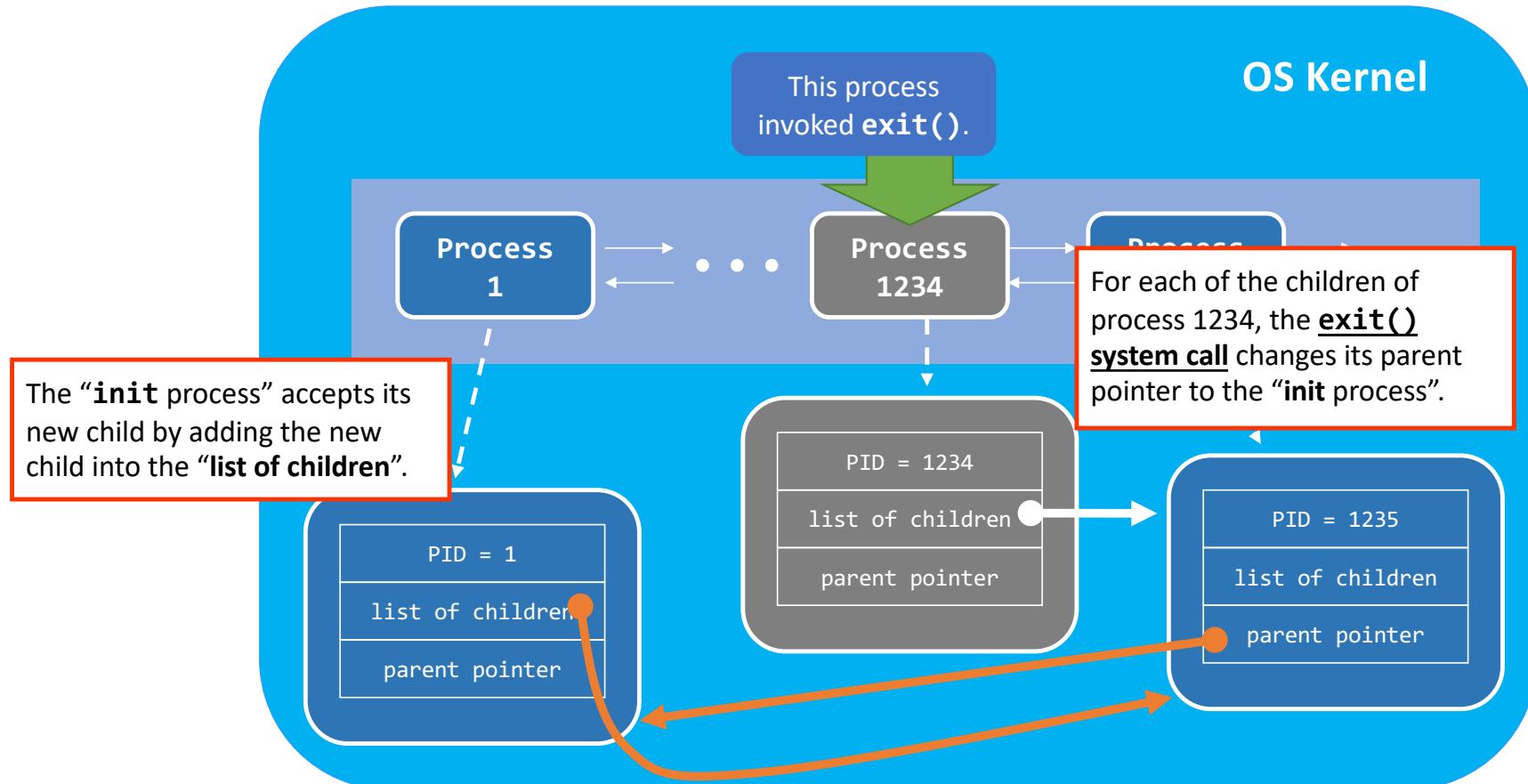
Re-parenting Explained



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**

reparenting
把父进程

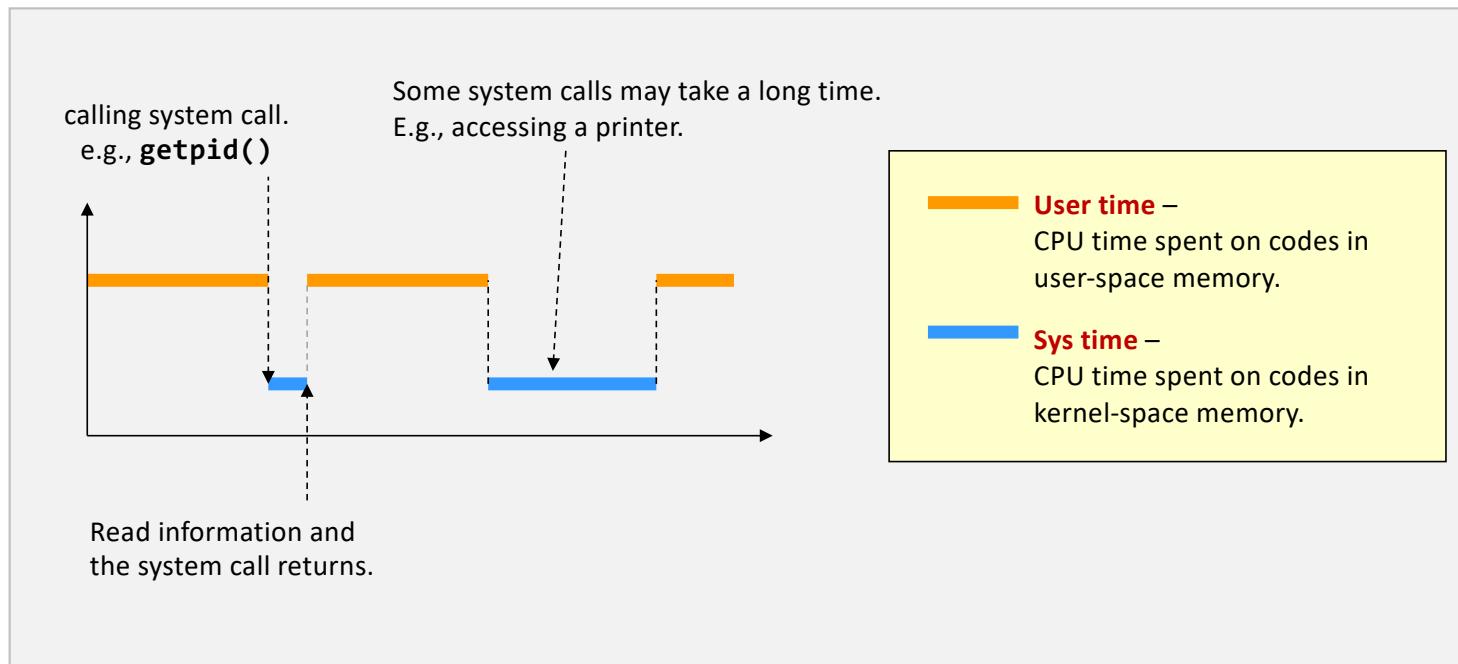
```
$ ./infinite_loop &
$ exit

[ The shell is gone ]
```

[Back to home](#)

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

Measure Process Time



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

sys time
有进阶切换

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

计算 real-time RT
有叠加

The diagram illustrates the output of the `time ./time_example` command:

```
$ time ./time_example
real    0m0.001s
user    0m0.000s
sys     0m0.000s
$ _
```

Annotations explain the components of the output:

- Real-time elapsed when "./time_example" terminates.** Points to the "real" line.
- The user time of "./time_example".** Points to the "user" line.
- The sys time of "./time_example".** Points to the "sys" line.

A blue box contains the text: "It's possible:
real > user + sys
real < user + sys". An arrow from this box points to the "real" line in the output.

A blue box contains the word "Why?"

A yellow box contains C code for a summation loop:

```
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;
}
```

printf() 本身是API，有缓冲区

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
$ _
```

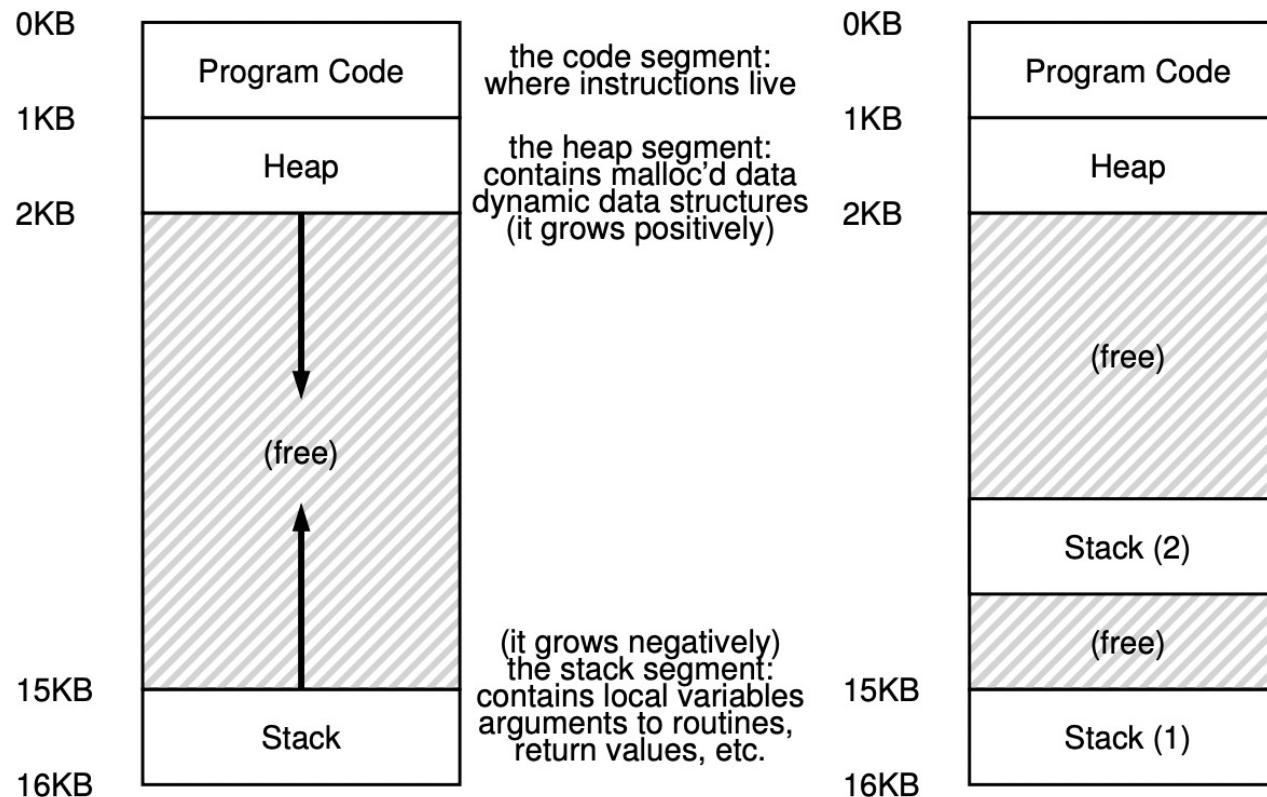
Threads

What is a Thread?

process 是 machine 的工作
thread 是 execution 的工作

- Thread is an **abstraction** of the execution of a program
 - A single-threaded program has one point of execution
 - A multi-threaded program has more than one points of execution
- Each thread has its own **private execution state**
 - Program counter and a private set of registers
 - A private stack for thread-local storage
 - CPU switching from one thread to another requires context switch
- Threads in the same process **share** computing resources
 - Address space, files, signals, etc.

Single-Threaded and Multi-Threaded



Why Use Thread?

- Increase parallelism
 - One thread per CPU makes better use of multiple CPUs to improve efficiency
- Avoid blocking program progress due to slow I/O
 - Threading enables overlap of I/O with other activities within a single program
 - e.g., many modern server-based applications (web servers, database management systems, and the like) make use of threads
- And allow resource sharing !!!

Thread Implementation

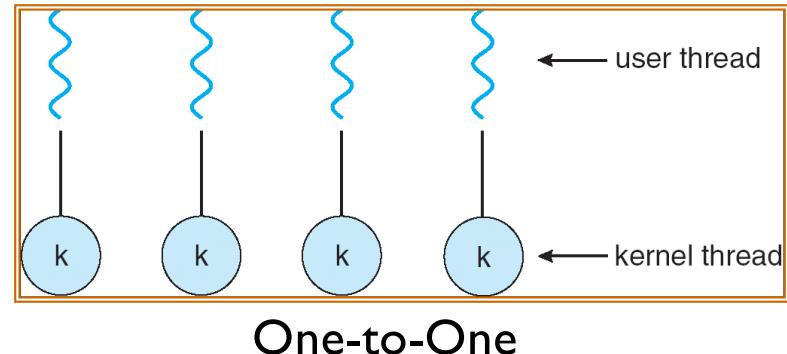
不同的有不同实现方式

- User-level thread
 - Thread management (e.g., creating, scheduling, termination) done by user-level threads library
 - OS does not know about user-level thread
- Kernel-level thread
 - Threat management done by kernel
 - OS is aware of each kernel-level thread

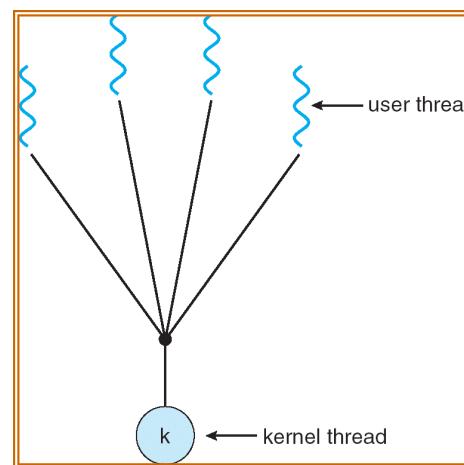
Thread Models

- One-to-one mapping
 - One user-level thread to one kernel-level thread
- Many-to-one mapping
 - Many user-level threads to one kernel-level thread
- Many-to-many mapping
 - Many user-level threads to many kernel-level threads

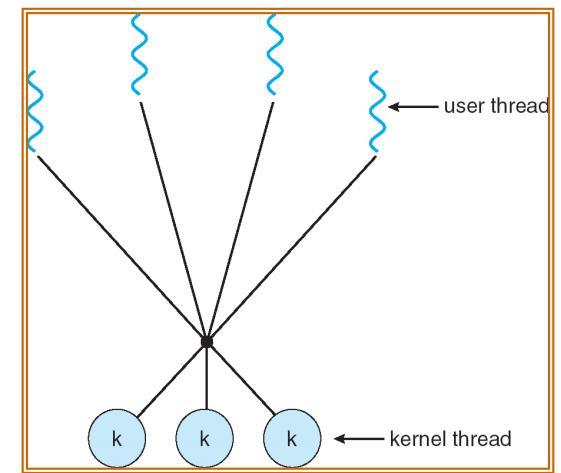
user > kernel



One-to-One



Many-to-One



Many-to-Many

Pros and Cons

- Many-to-one mapping 在用户态实现
 - Pros: context switch between threads is cheap 动操作便利
 - Cons: When one thread blocks on I/O, all threads block 一个block,全部block
- One-to-one mapping 常用
 - Pros: Every thread can run or block independently
 - Cons: Need to make a crossing into kernel mode to schedule
- Many-to-many mapping
 - Many user-level threads multiplexed on less or equal number of kernel-level threads
 - Pros: best of the two worlds, more flexible
 - Cons: difficult to implement

Thank you!

