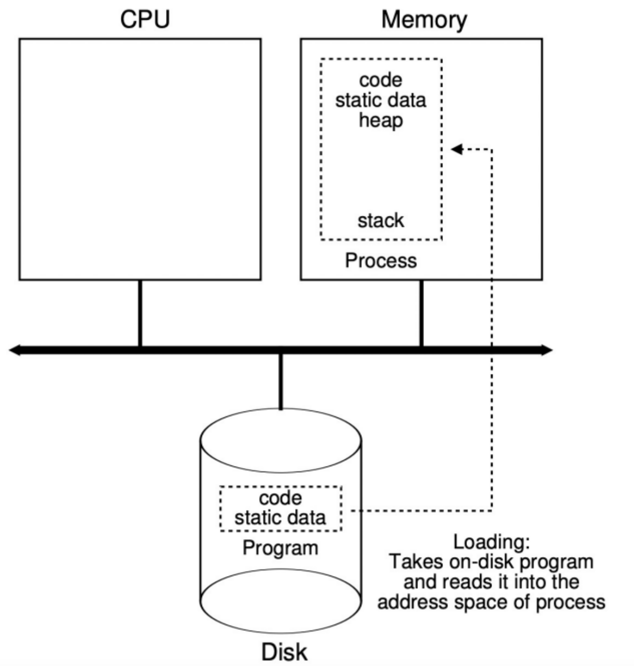
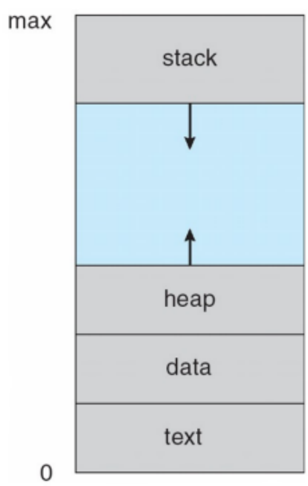
Lecture3 Processes

1. Process and System calls

What is a Process



- Process is a **program in execution**
- Program is a **file on the disk** (code and static data)
- Process is **loaded by the OS**
    - Code and static data are loaded from the program
    - Heap and stack are created by the OS



- 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 Identification

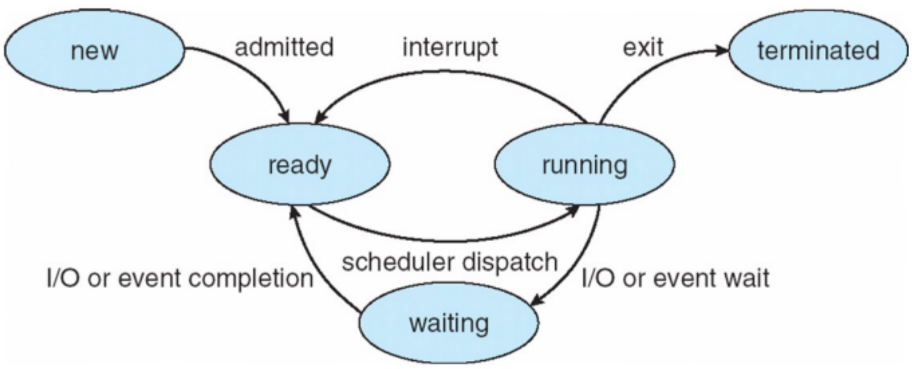
```c
// compile to getpid
#include <stdio.h>    // printf()
#include <unistd.h>   // getpid()

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
```

Each process is given a unique ID number, and is called the process ID, or the PID.

- System call `getpid()` prints the PID of the calling process

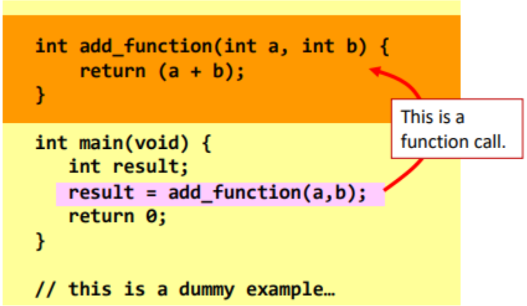
Process Life Cycle

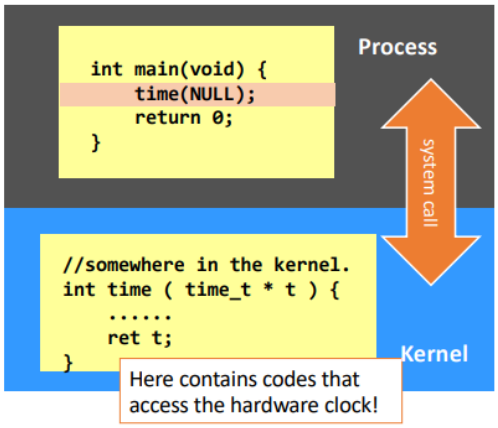


- **interrupt** e.g. time interrupt
- **I/O or event wait** e.g. open a file

System Call

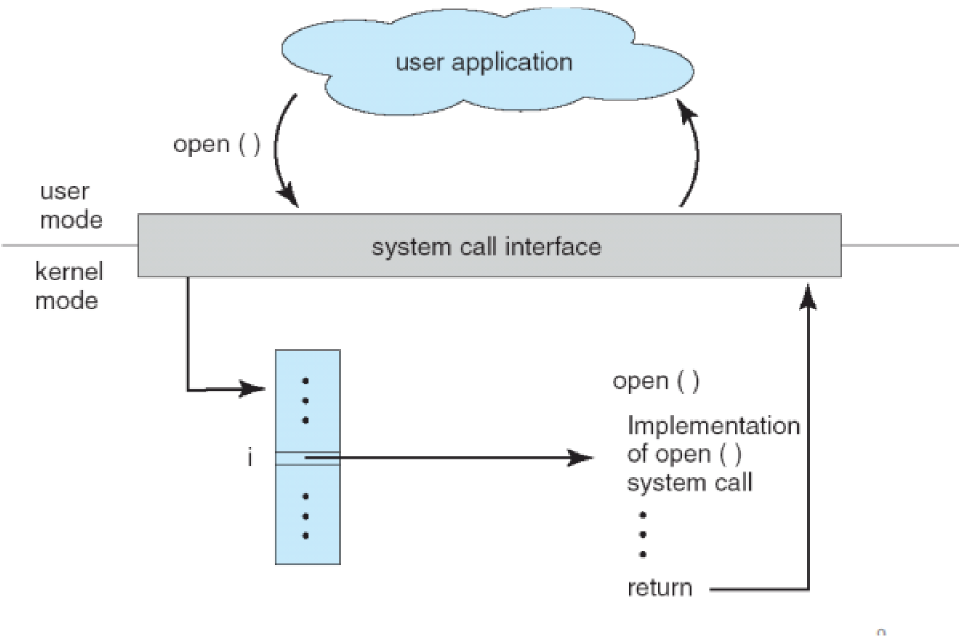
Process-Kernel Interaction





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

Call by Number



- System call is different from function call
- System call is a 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
```

- Kernel code from xv6-riscv

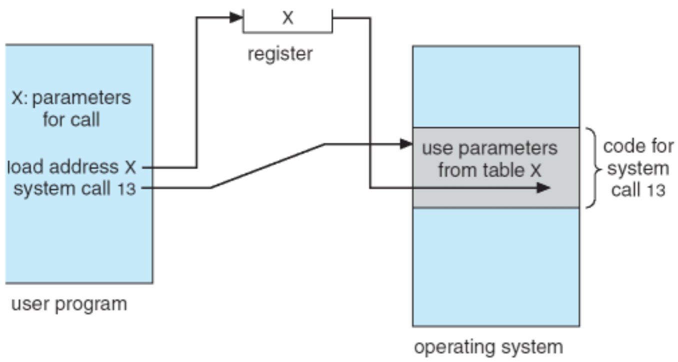
```
/* kernel/syscall.h */

#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]();
}
```

Parameter Passing

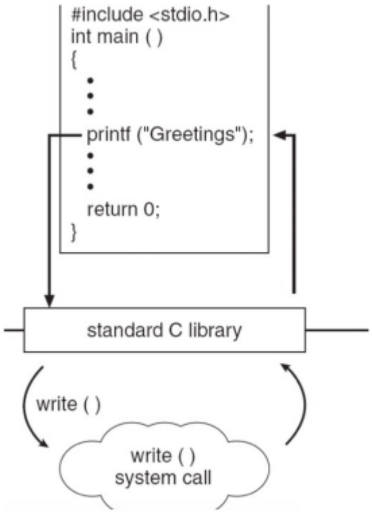
- **more information** is required than the index of desired system call
  - Exact type and amount of information **vary** according to OS and call
- **Three general methods used to pass parameters to the OS**
  - **Registers**: pass the parameters in registers
    - In some cases, may be more parameters 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

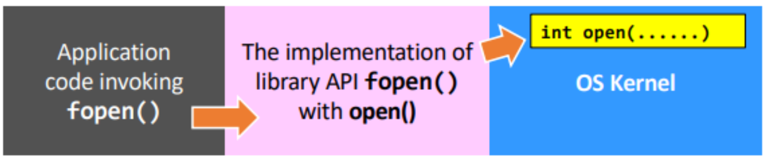
System Call v.s. Library API call



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

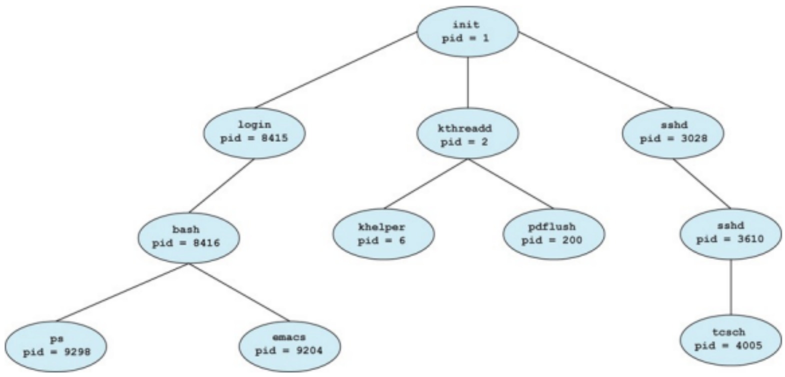
Name	System call?
`printf() & scanf()`	No
`malloc() & free()`	No
`fopen() & fclose()`	No
`mkdir() & rmdir()`	Yes
`chown() & chmod()`	Yes

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



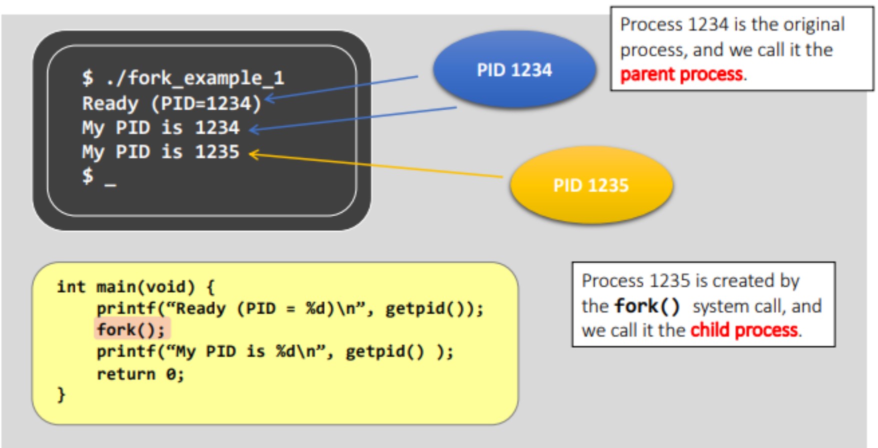
- Take `fopen()` as an example
    - `fopen()` invokes the system call `open()`
    - `open()` is too primitive and is not programmer-friendly

2. Process creation



- Parent process create children processes, which, in turn create other processes, forming **a tree of processes**. We create one process (parent process) from another process (child process).
- Generally, process identified and managed via a **process identifier** (PID)

Creating Process with fork() system call



- 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**

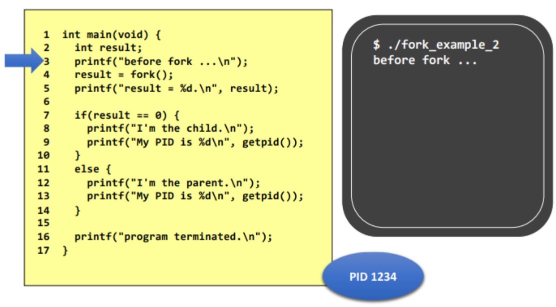
Diagram	Illustration
	

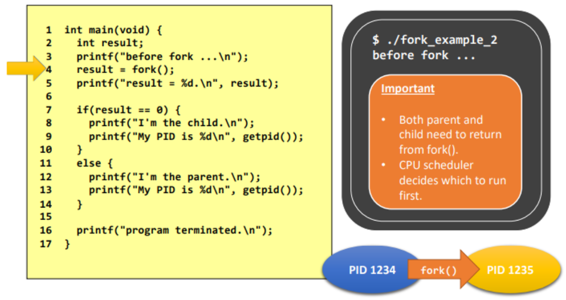
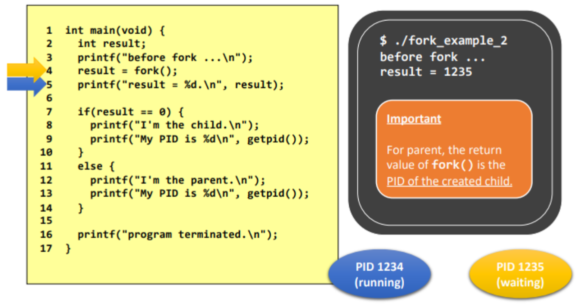
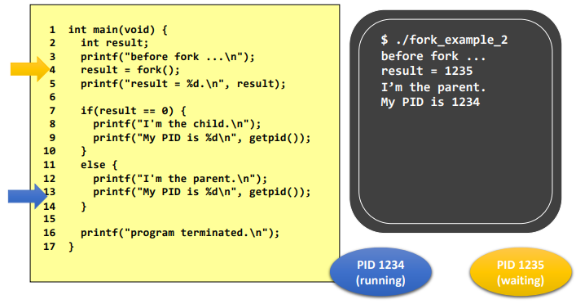
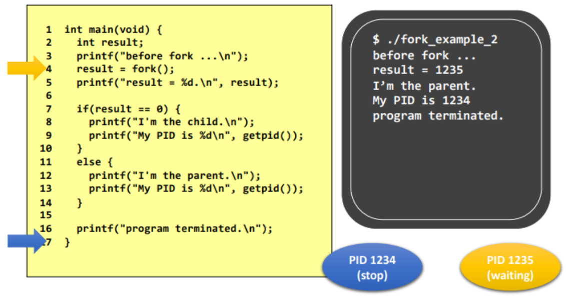
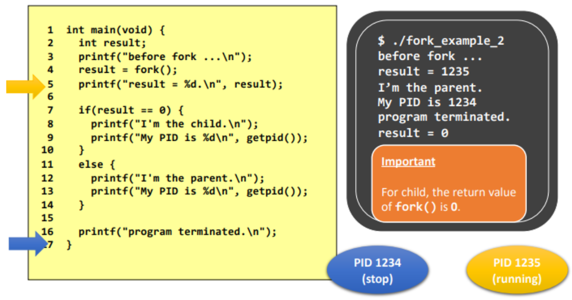
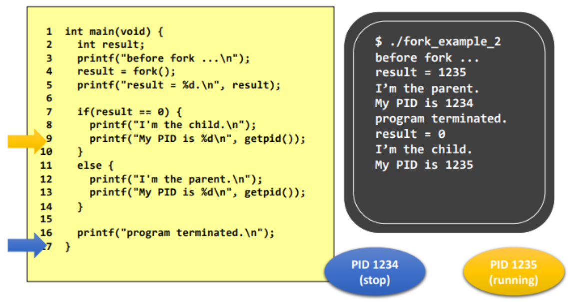
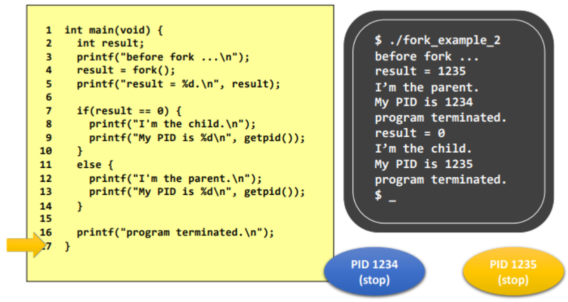
Diagram	Illustration
	Both parent and child need to return from `fork()` **CPU scheduler decides** which to run first
	For parent, the return value of `fork()` is **the PID of the created child**
	
	
	For child, the return value of `fork()` is **0**
	

Diagram	Illustration
	

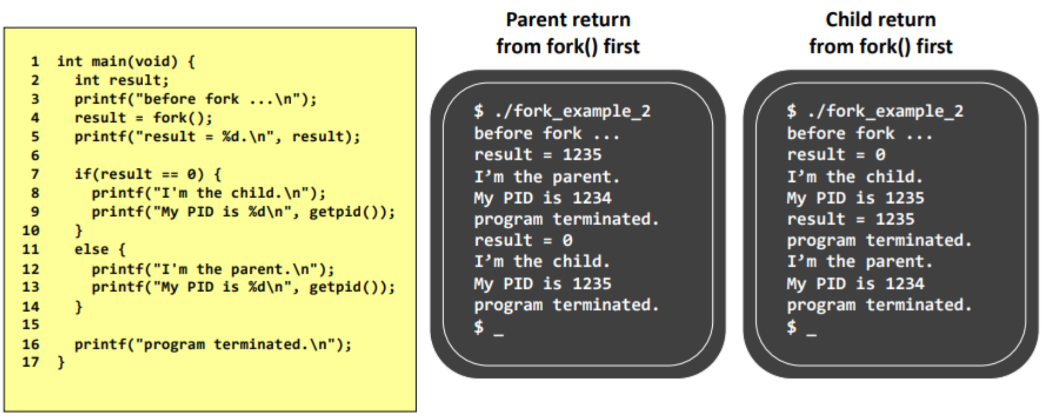
fork() system call

`fork()` does not clone the following

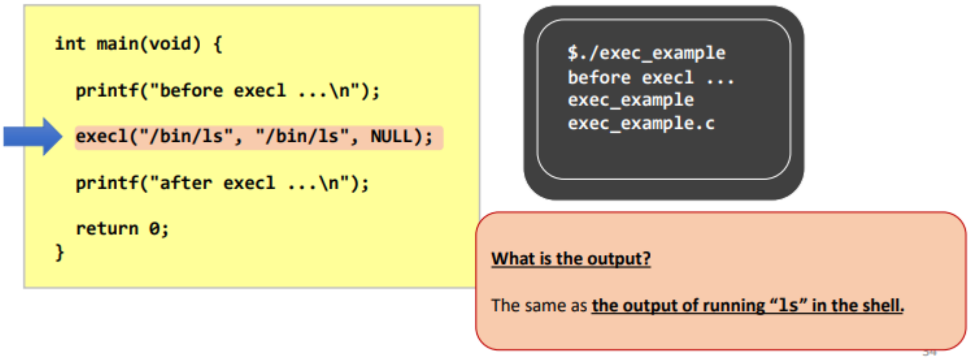
Distinct items	Parent	Child
Return value of fork()	PID of the child process.	0
PID	Unchanged.	Different, not necessarily be "Parent PID + 1"
Parent process	Unchanged.	Parent.
Running time	Cumulated.	Just created, so should be 0.
[Advanced] **File locks**	Unchanged.	None.

- `fork()` duplicates the current process itself
- If a process can only duplicate itself and always runs the same program, it's not quite meaningful

CPU Scheduler and fork()



exec() system call



- `execl()`: a member of the exec system call family (and the family has 6 members)
  - 1st argument: program name `/bin/ls` in the example
  - 2nd argument: argument[0] to the program
  - 3rd argument: argument[1] to the program

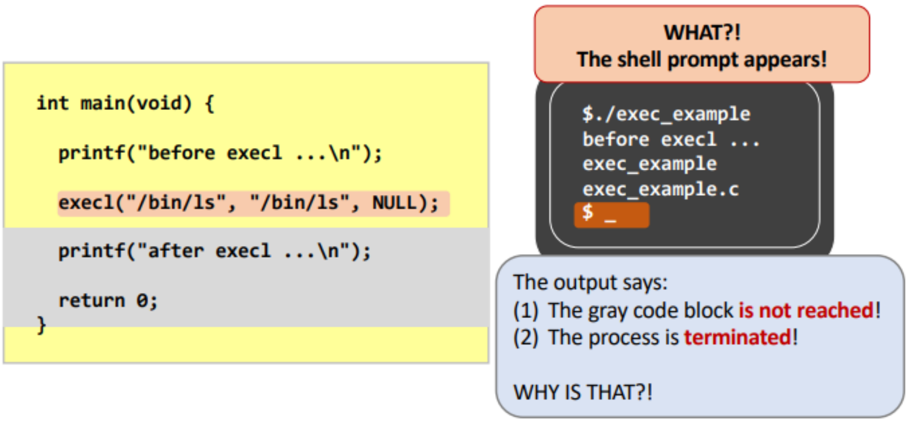
```
execl("/bin/ls", "/bin/ls", NULL);
```

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.

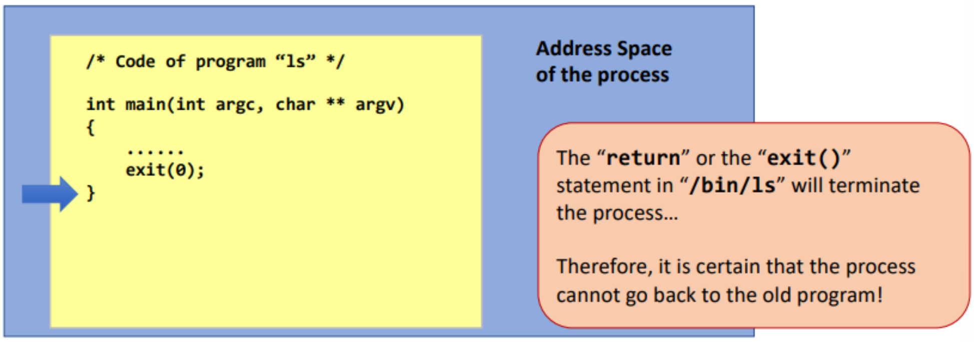
```
execl("/bin/ls", "/bin/ls", "-l", NULL);
```

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.

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

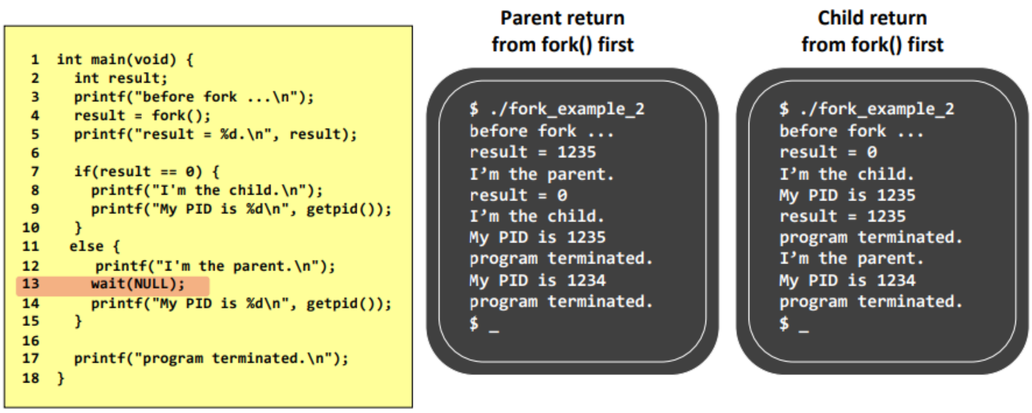


- `exec()` loads program "ls" into the **memory** of this process

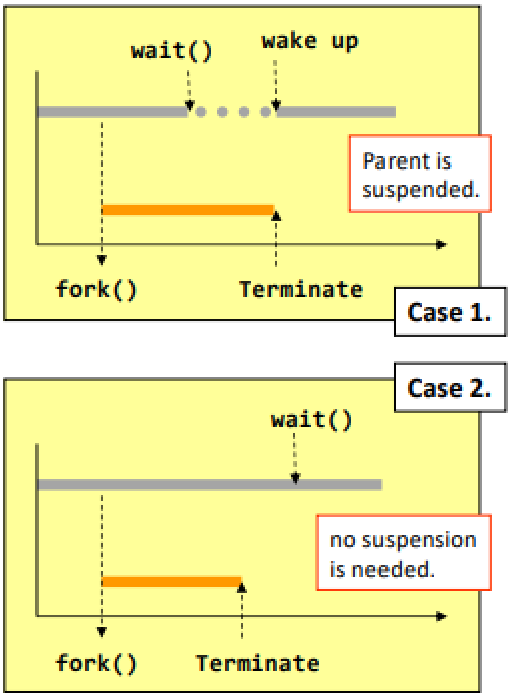


  - The `return` or the `exit()` statement in `/bin/ls` will terminate the process
  - Therefore, it is certain that the process **cannot go back to the old program**
- The process is changing the code that is executing and never returns to the original code
- The process that calls an `exec` system call will **replace user-space info**, e.g.,
  - **Program Code**
  - **Memory**: local variables, global variables, and dynamically allocated memory
  - **Register value**: e.g. PC
- But, the **kernel-space info** of that process is **preserved**, including
  - PID
  - Process relationship

wait() system call: Sync Parent with Child



- `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

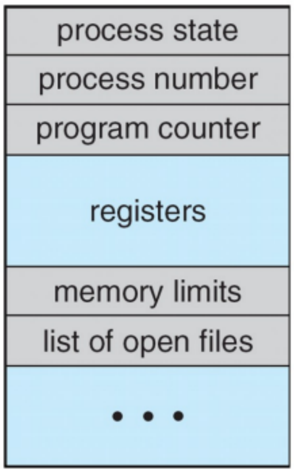
wait() v.s. waitpid()

- `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)

3. Processes: Kernel view

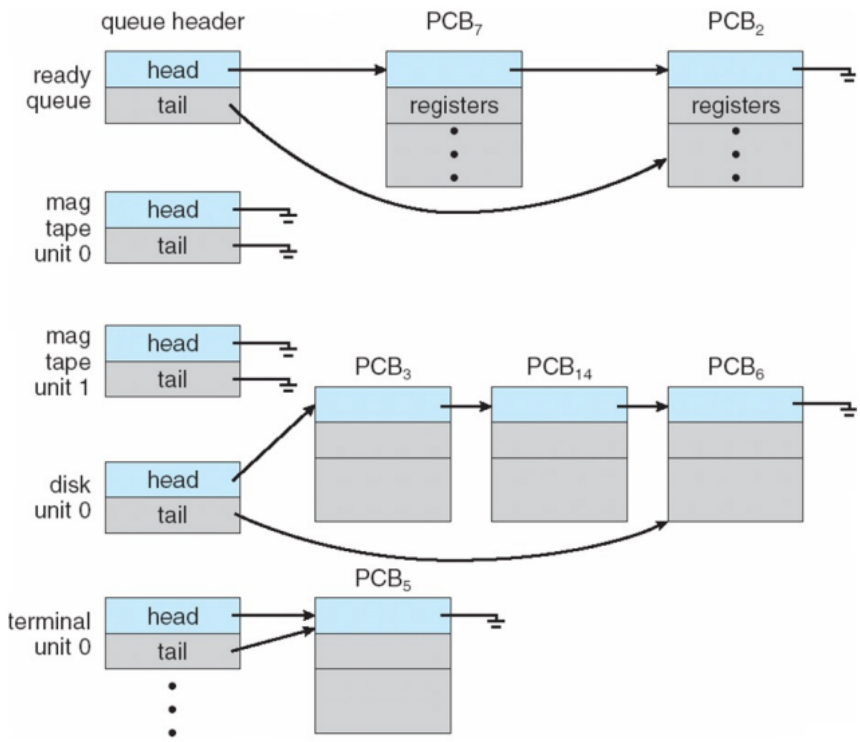
Process Control Block (PCB)



Information associated with each process

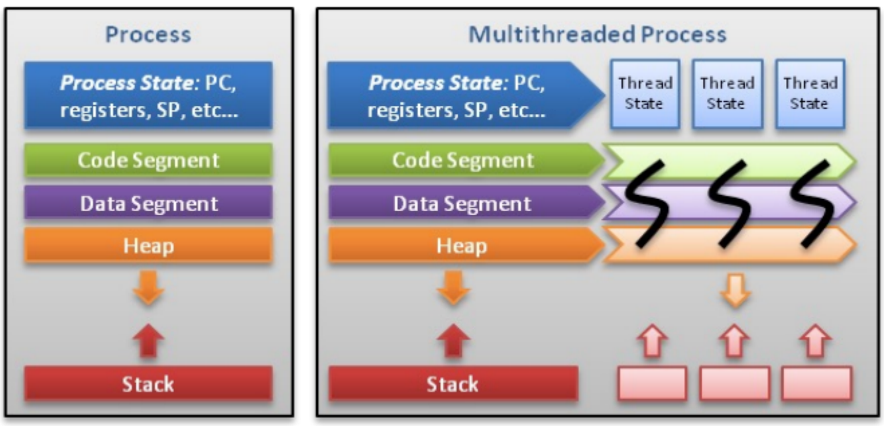
- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information

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

Process and Threads



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

- 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

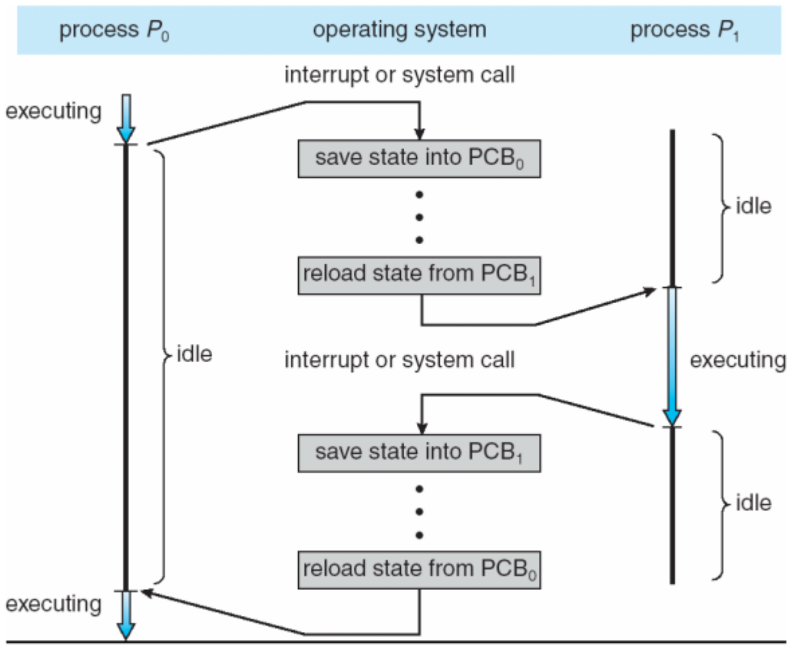
even when the main thread is busy

- PCB is extended to **include information about each thread**

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

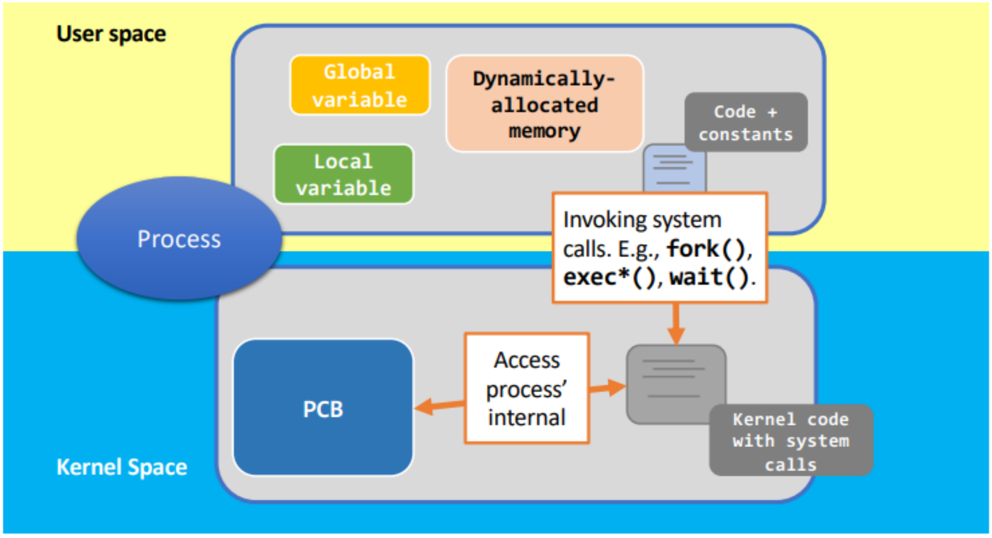
Context Switch



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

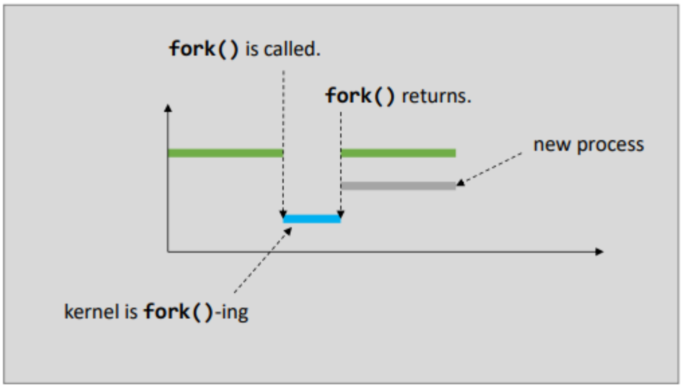
- Context of a process is represented in the **PCB**
- The time used to do context switch is an overhead of the system; the system does no useful work while switching
    - Time of context switch depends on **hardware** support
    - Context switch **cannot be too frequent**

4. Kernel view of fork(), exec() and wait()

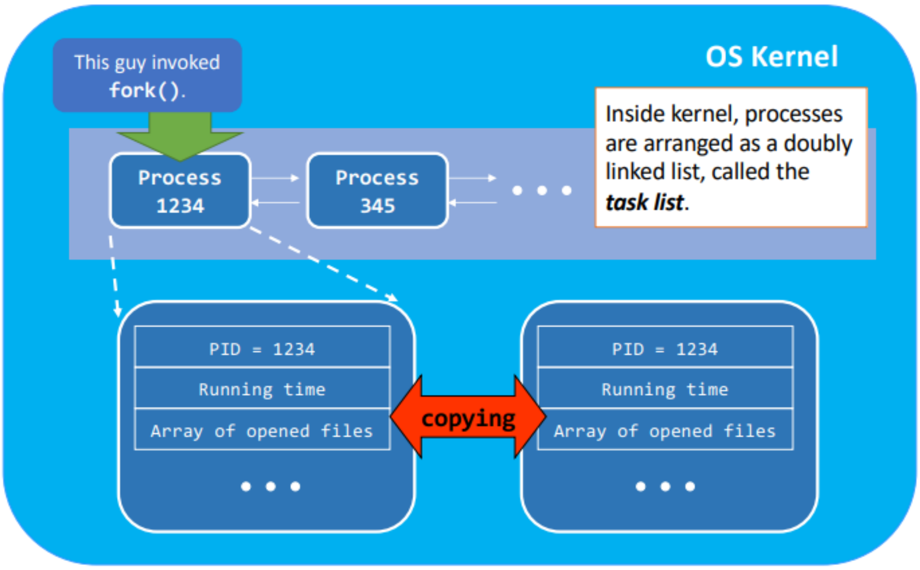


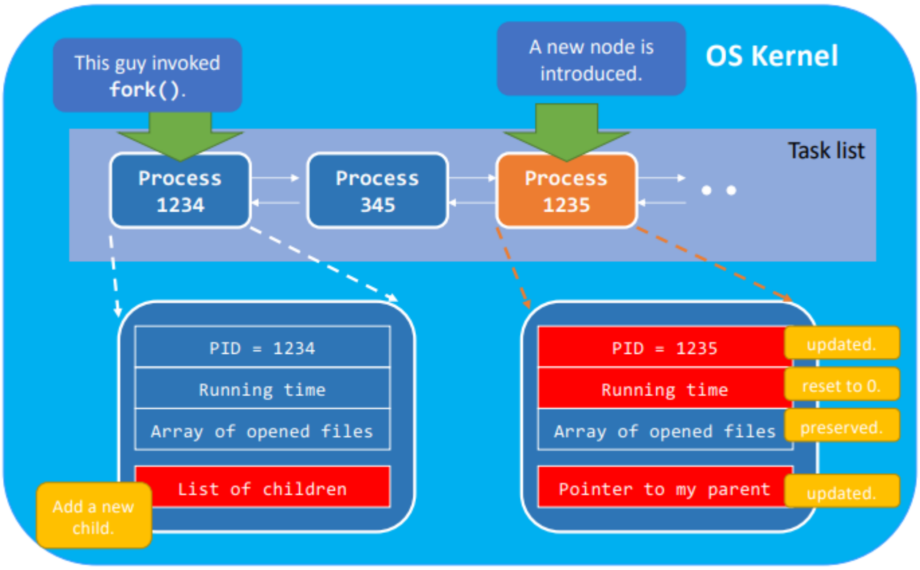
Fork

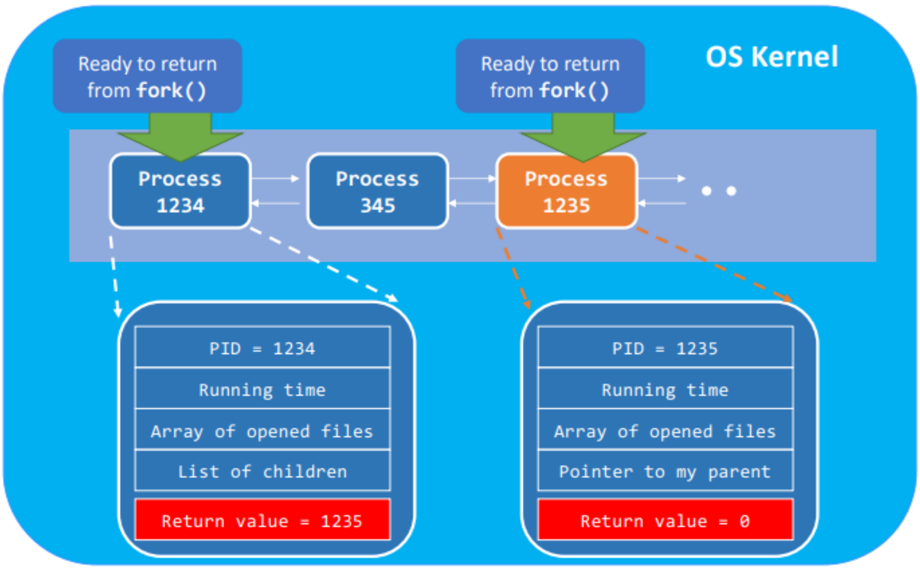
In User Mode



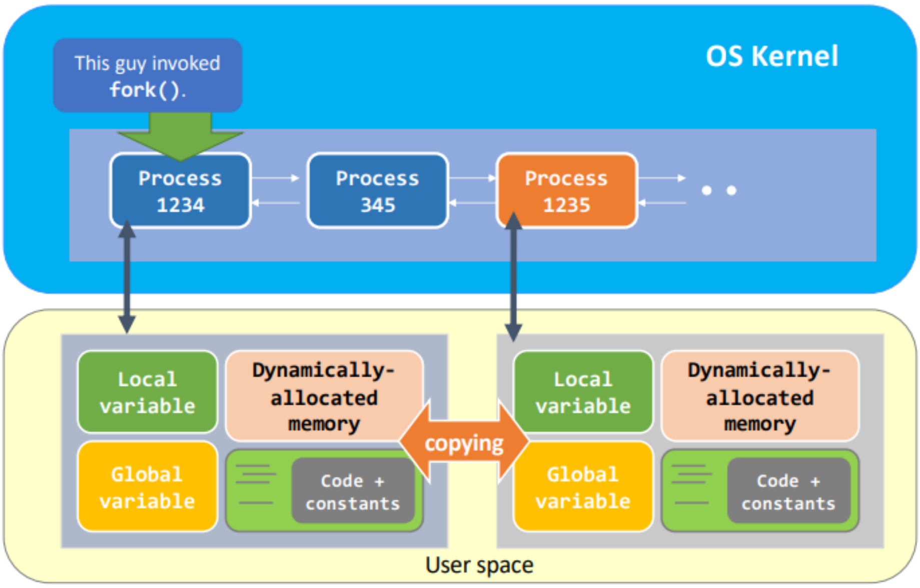
In Kernel Mode



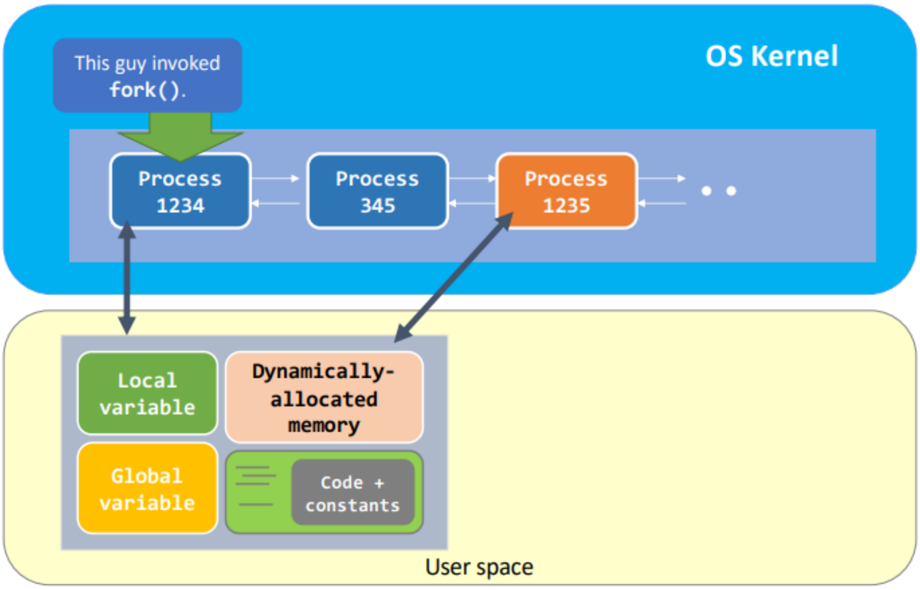




Case 1: Duplicate Address Space

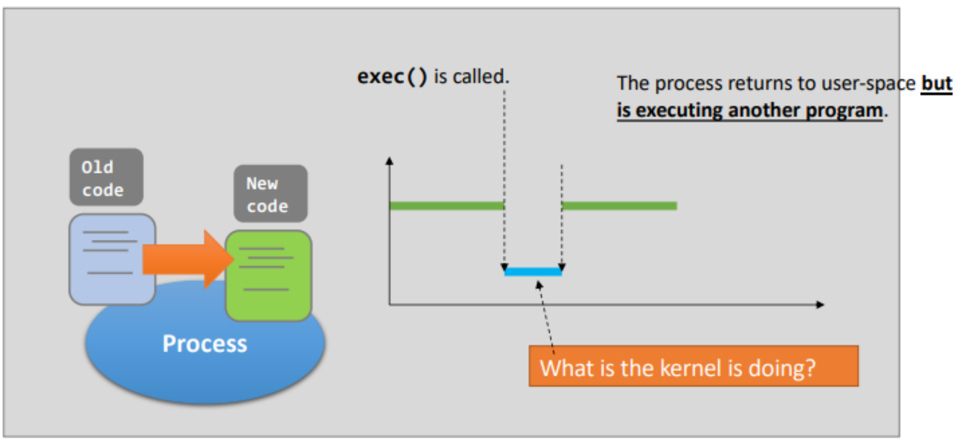


Case 2: Copy on Write

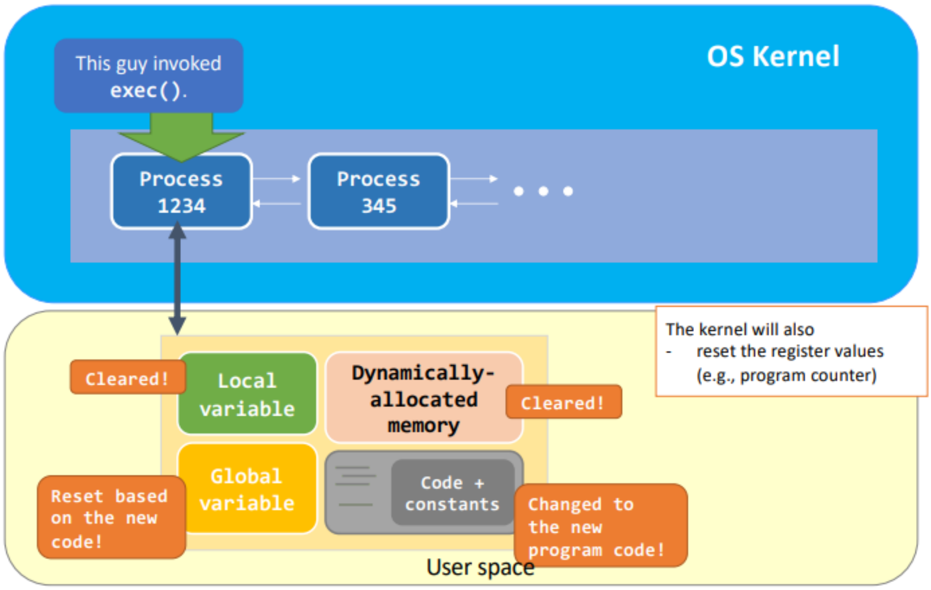


Exec

In User Mode

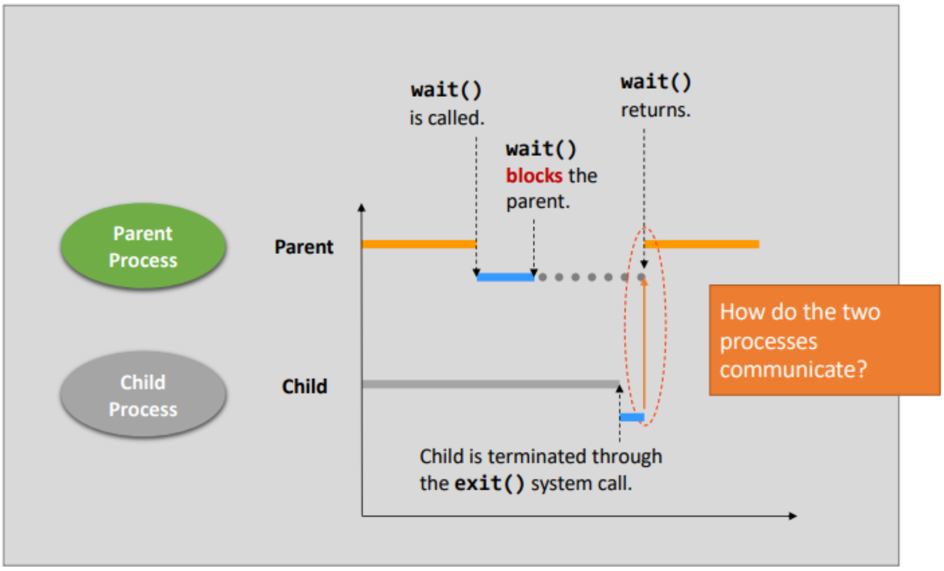


In Kernel Mode

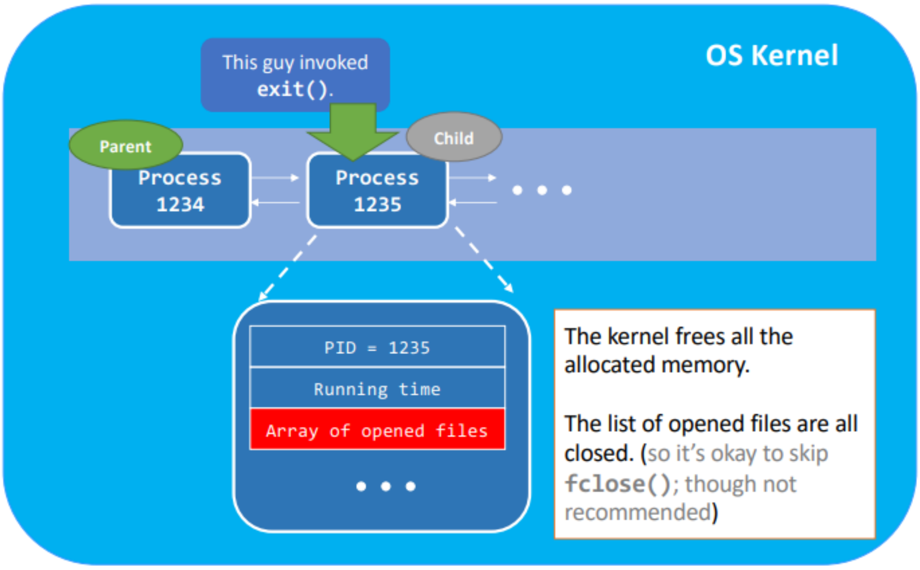


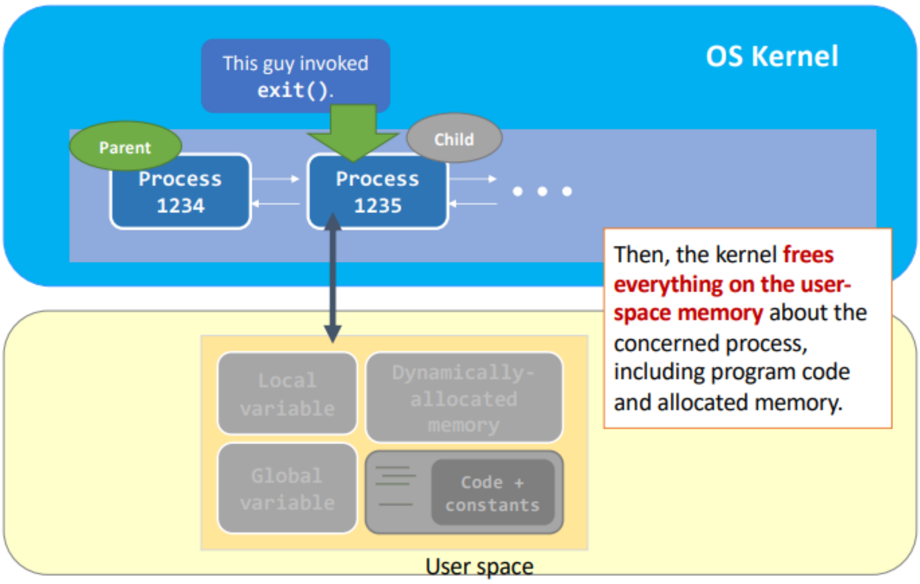
Exit

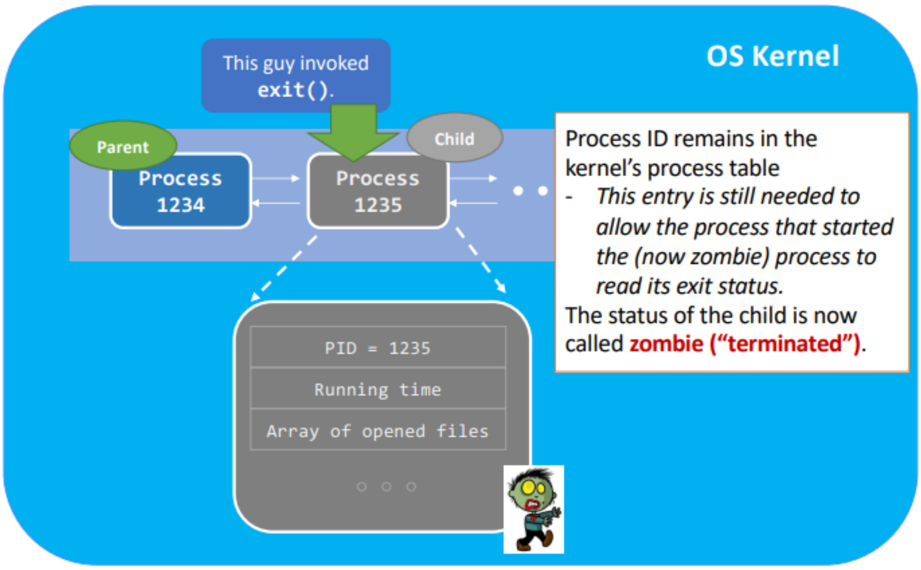
In User Mode

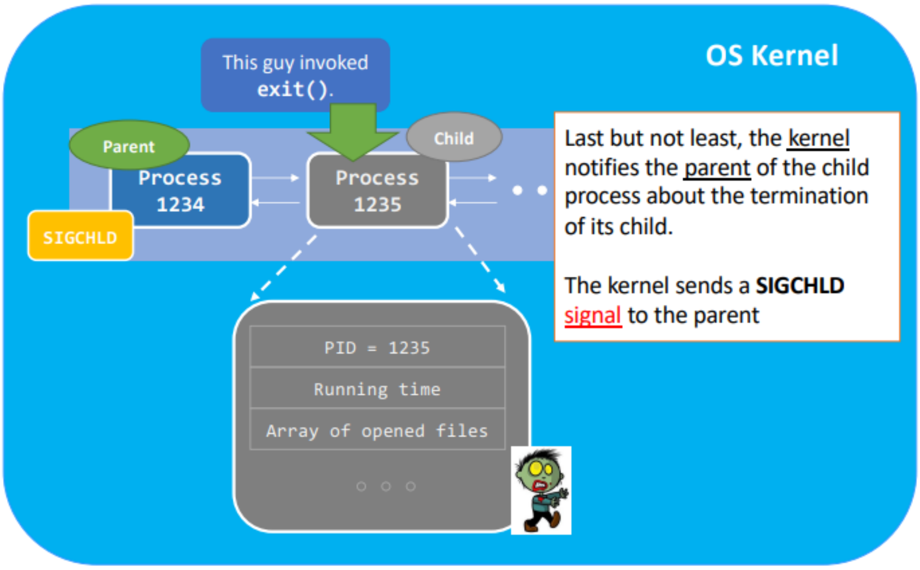


In Kernel Mode


OS Kernel
This guy invoked exit().
Parent
Child
Process 1234
Process 1235
PID = 1235
Running time
Array of opened files
The kernel frees all the allocated memory.
The list of opened files are all closed. (so it's okay to skip fclose(); though not recommended)


OS Kernel
This guy invoked exit().
Parent
Child
Process 1234
Process 1235
Then, the kernel frees everything on the user-space memory about the concerned process, including program code and allocated memory.
Local variable
Dynamically-allocated memory
Global variable
Code + constants
User space


OS Kernel
This guy invoked exit().
Parent
Child
Process 1234
Process 1235
PID = 1235
Running time
Array of opened files
Process ID remains in the kernel's process table
- This entry is still needed to allow the process that started the (now zombie) process to read its exit status.
The status of the child is now called zombie ("terminated").



Summary

Exit() will execute the following processes

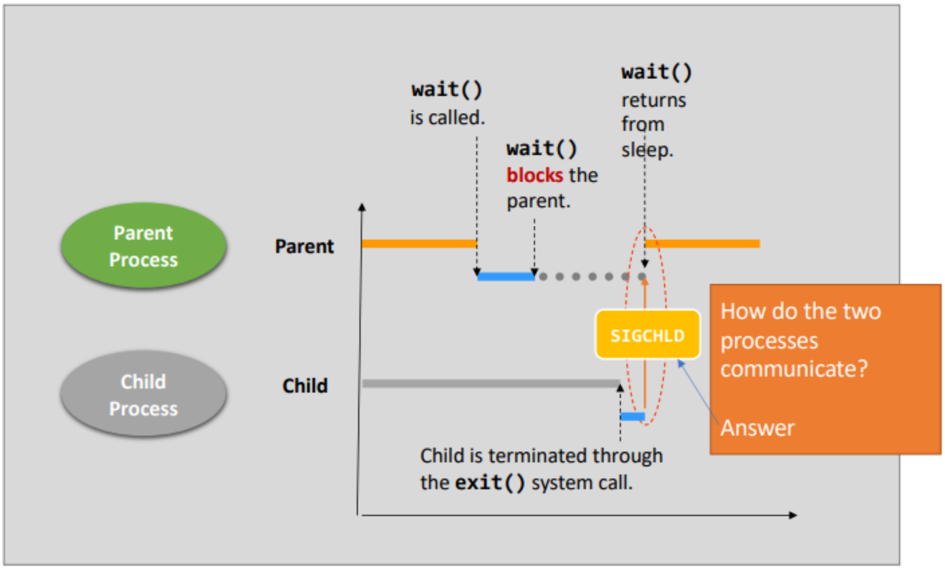
1. Clean up **most of** the allocated **kernel-space memory** (e.g., process's running time info).
2. Clean up the exit process's **user-space memory**
3. Notify the parent with `SIGCHLD`

Exit() system call turns a process into a zombie when

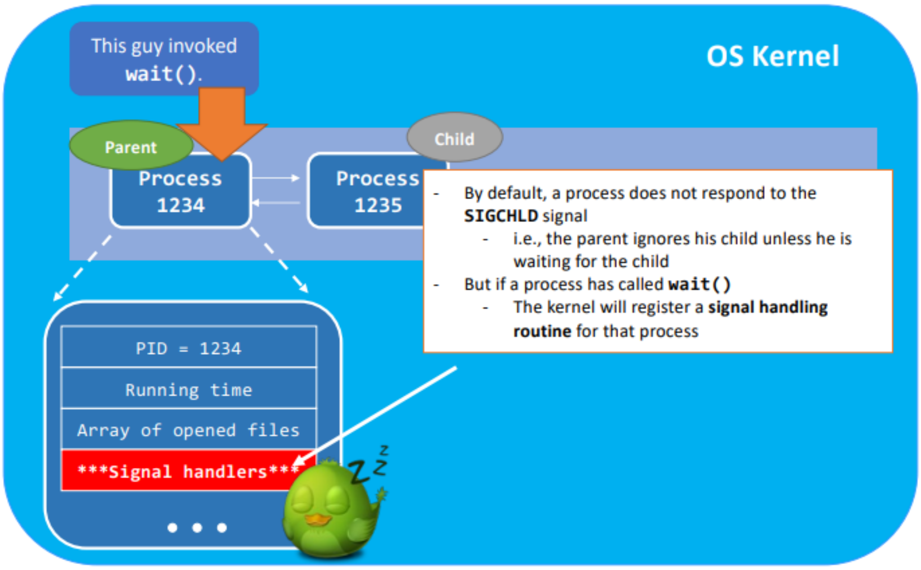
- The process calls exit()
- The process returns from main()
- The process terminates abnormally
    - The kernel knows that the process is terminated abnormally. Hence, the kernel invokes exit() for it

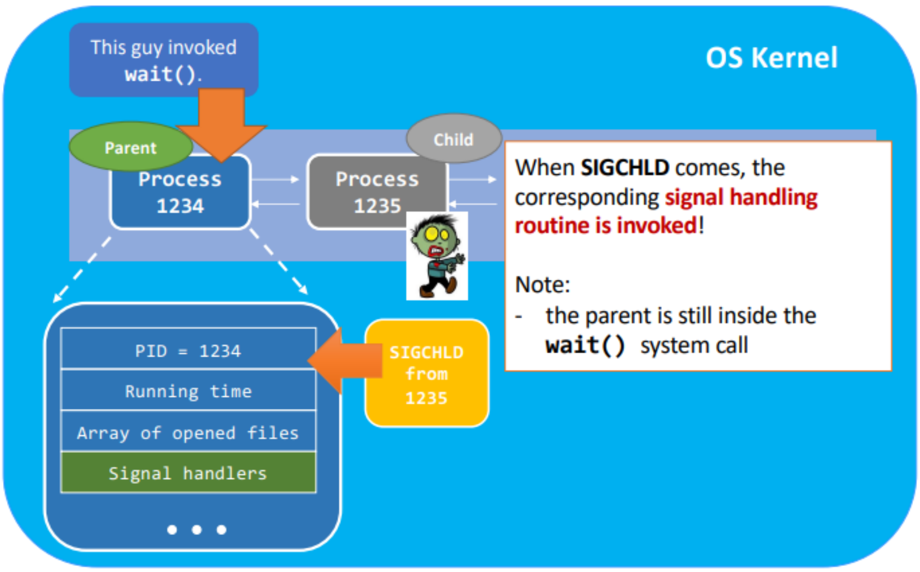
Wait

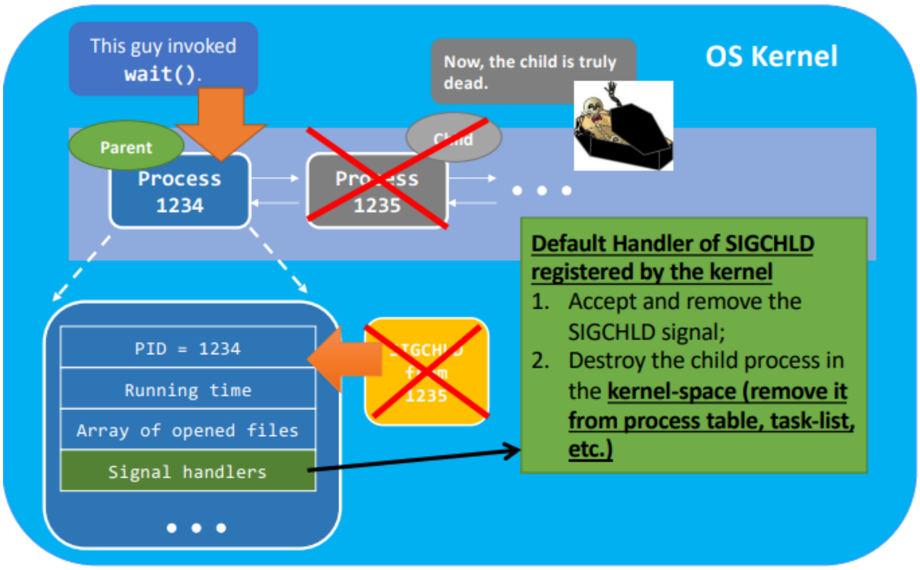
In User Mode

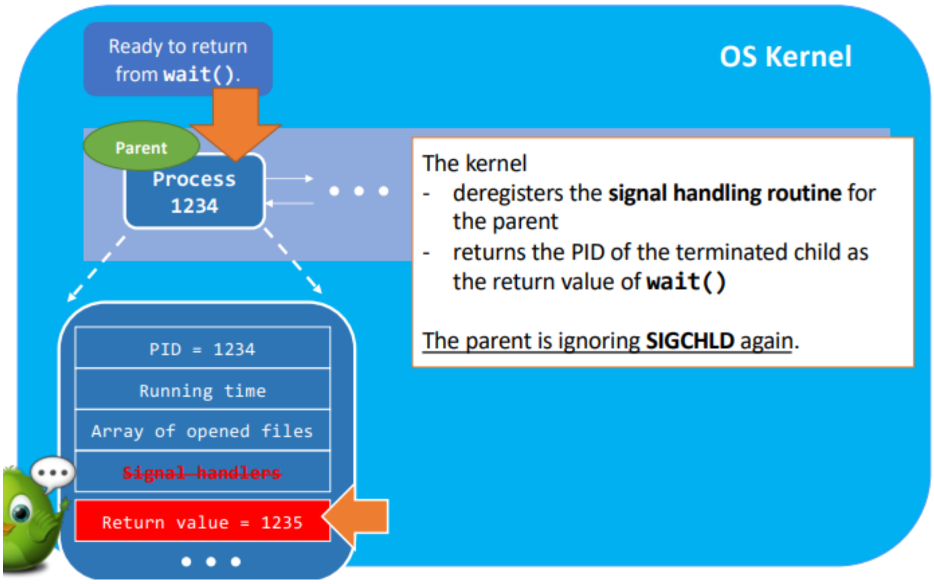


In Kernel View

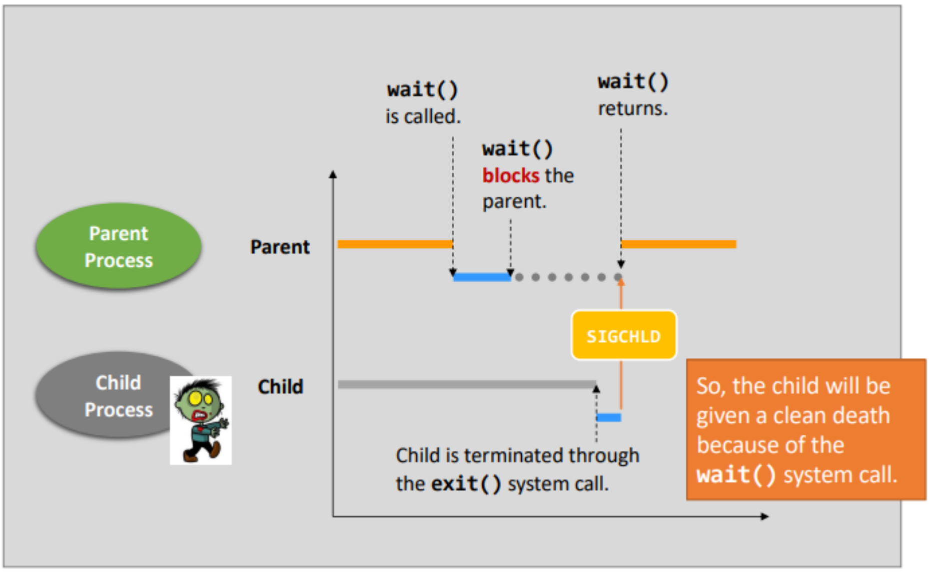





This guy invoked wait().
Now, the child is truly dead.
OS Kernel
Parent
Process 1234
Default Handler of SIGCHLD registered by the kernel
1. Accept and remove the SIGCHLD signal;
2. Destroy the child process in the kernel-space (remove it from process table, task-list, etc.)
PID = 1234
Running time
Array of opened files
Signal handlers

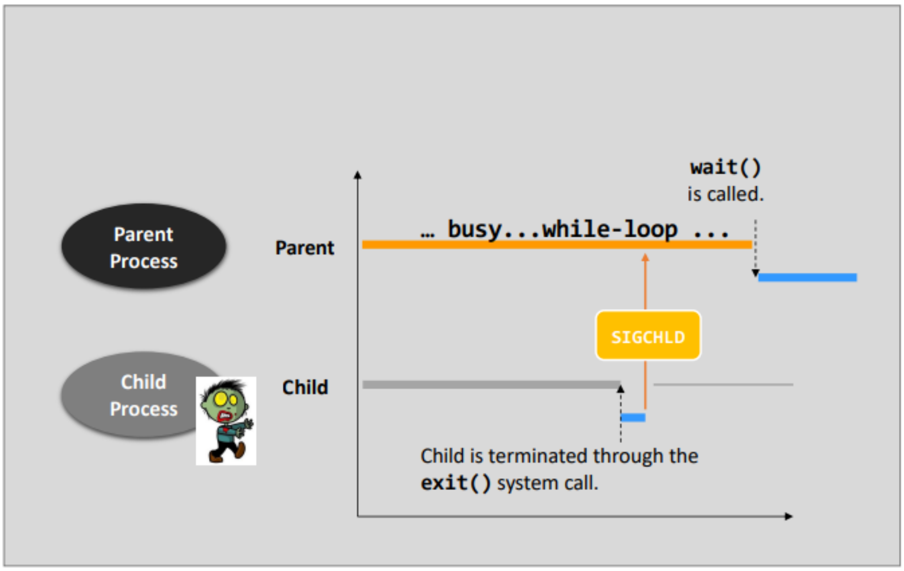

Ready to return from wait().
OS Kernel
Parent
Process 1234
The kernel
- deregisters the signal handling routine for the parent
- returns the PID of the terminated child as the return value of wait()
The parent is ignoring SIGCHLD again.
PID = 1234
Running time
Array of opened files
~~Signal handlers~~
Return value = 1235

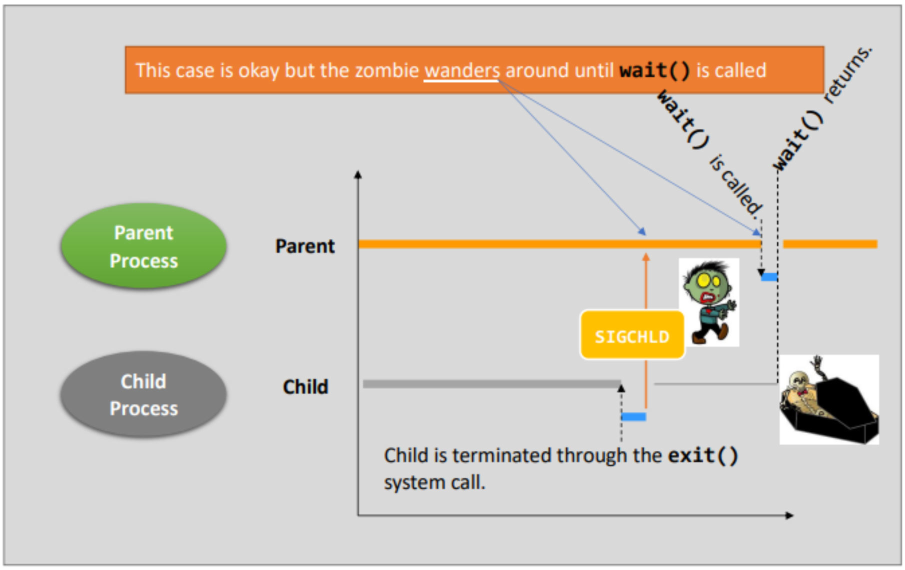
Normal Case

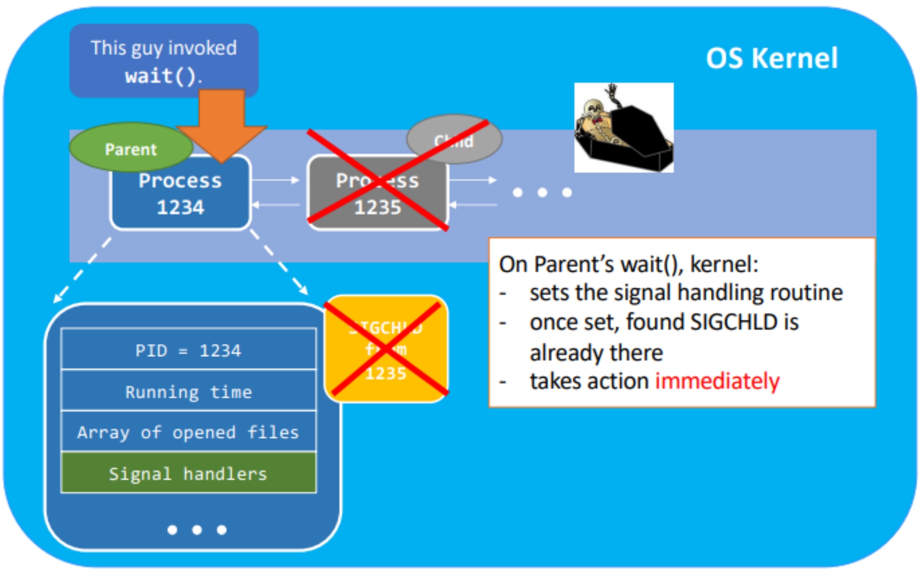

wait() is called.
wait() returns.
wait() blocks the parent.
Parent Process
Parent
Child Process
Child
SIGCHLD
Child is terminated through the exit() system call.
So, the child will be given a clean death because of the wait() system call.

Abnormal Case

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

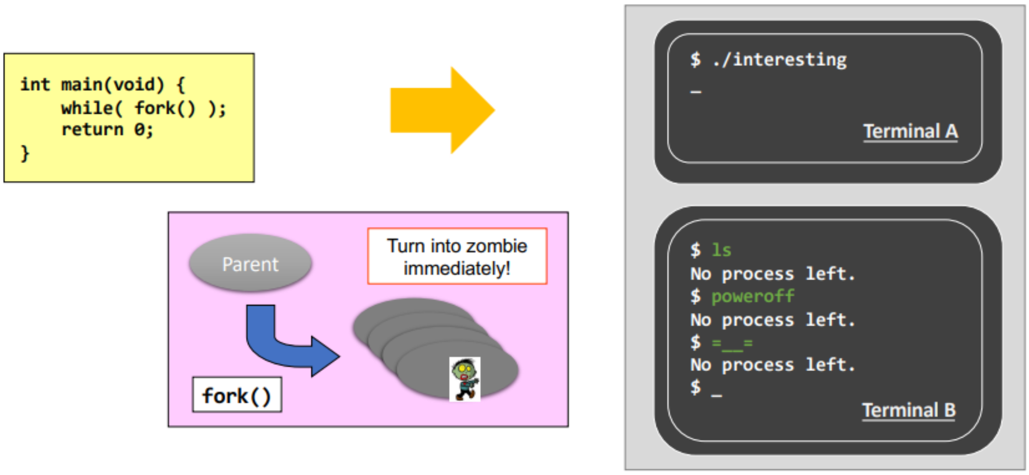






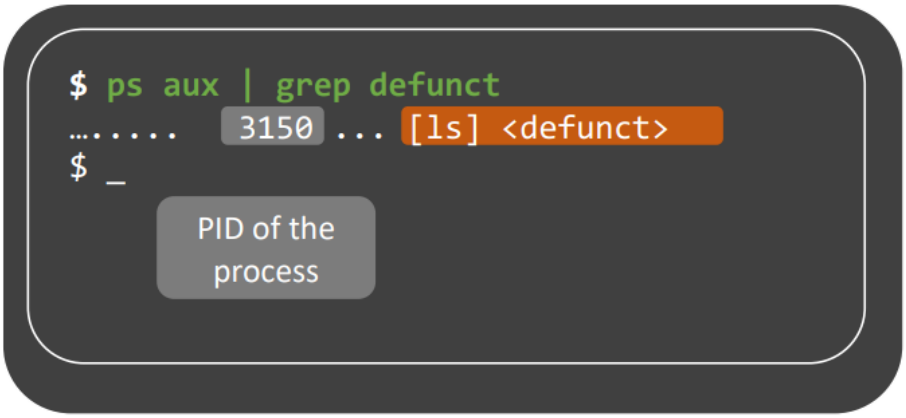
Using wait() for Resource Management

- 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



Summary

- 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
        - The caller receives a signal (i.e., the signal interrupted the wait())
- Linux will label zombie processes as `<defunct>`

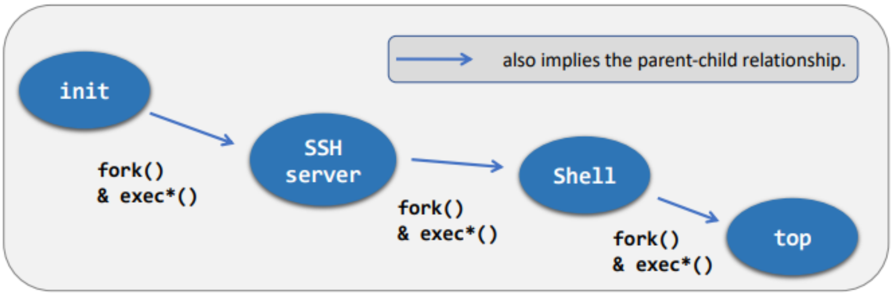


5. More about processes

The first process

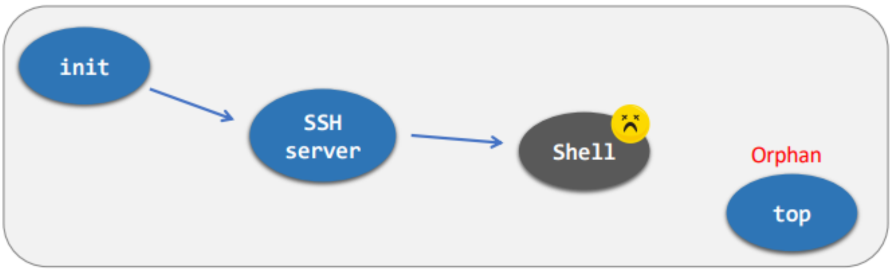
- We now focus on the process-related events
  - The kernel, while it is booting up, creates **the first process** – `init`
- The `init` process
  - has PID=1
  - is running the program code `/sbin/init`
- Its first task is to create more processes
  - using `fork()` and `exec()`

A Tree of Processes



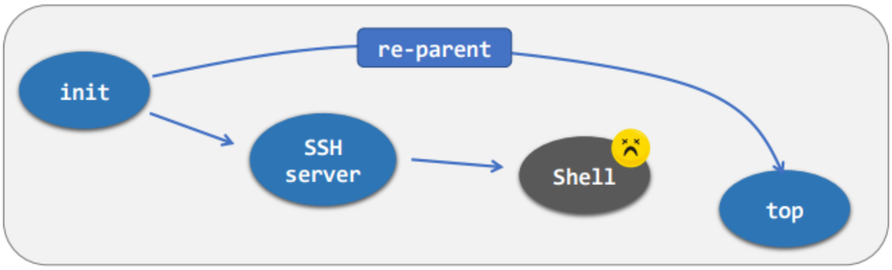
- You can view the tree with the command:
  - `pstree`
  - `pstree -A`: for ASCII-character-only display

Orphans



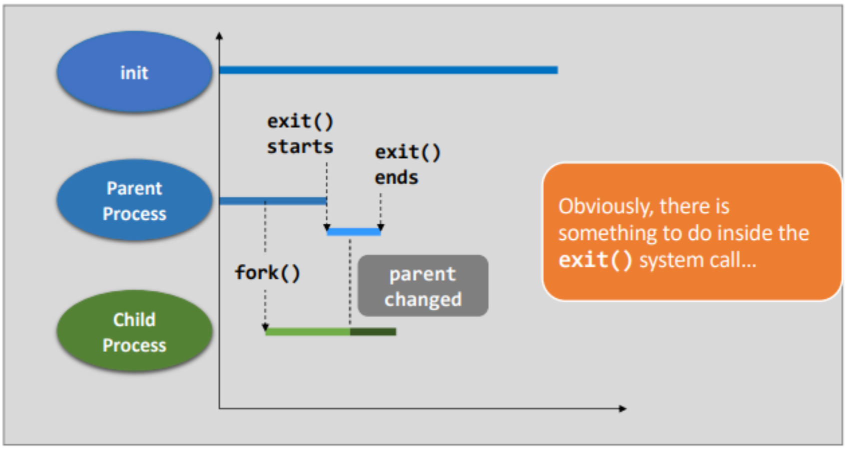
- However, termination can happen, at any time and in any place
  - This is no good because an orphan turns the hierarchy from a tree into a forest
  - no one would know the termination of the orphan

Re-parent

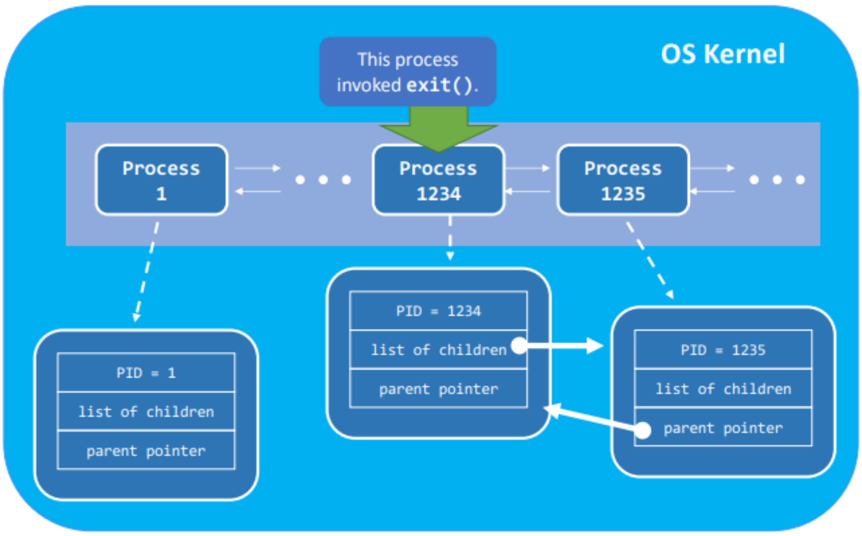


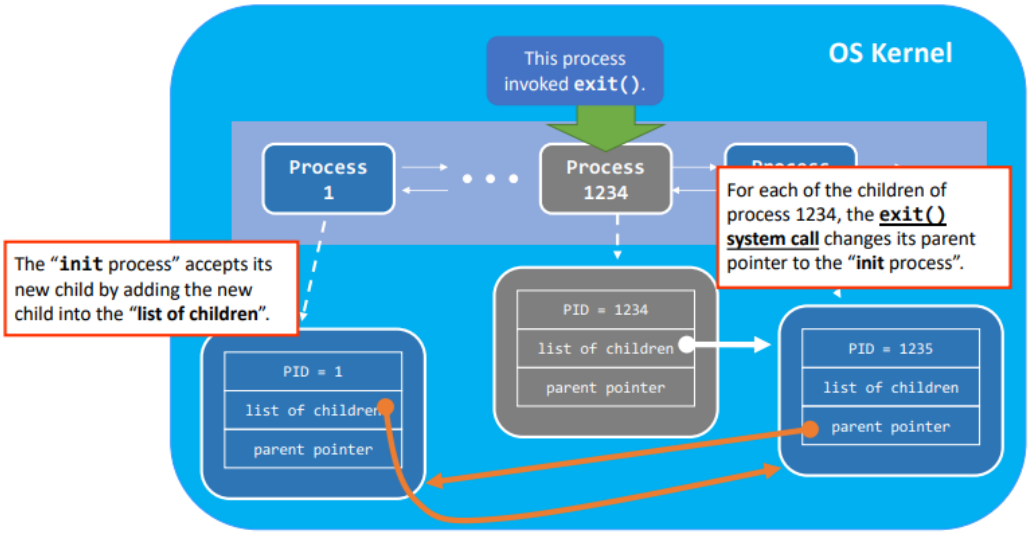
- In Linux
  - The `init` process will become the step-mother of all orphans
  - It's called **re-parenting**
- In Windows
  - It maintains a forest-like process hierarchy

In User Mode



In Kernel Mode

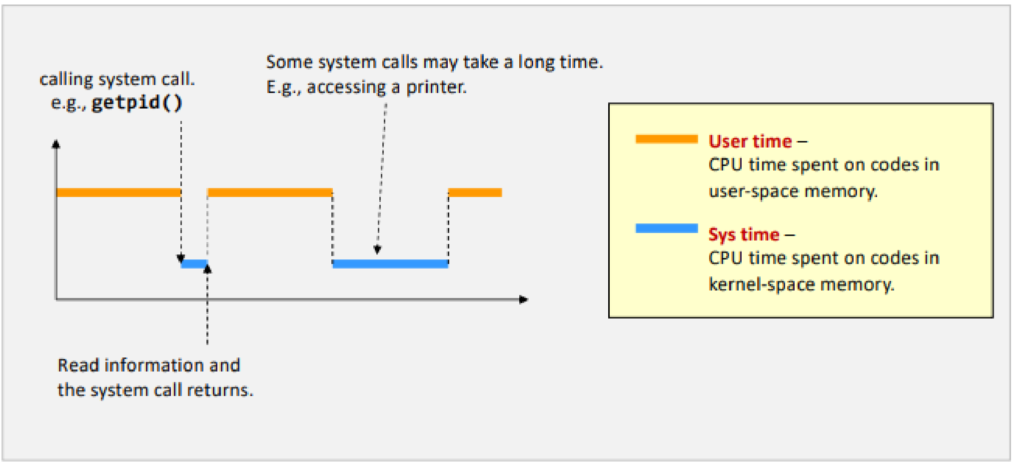




Background Jobs

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

Measure Process Time



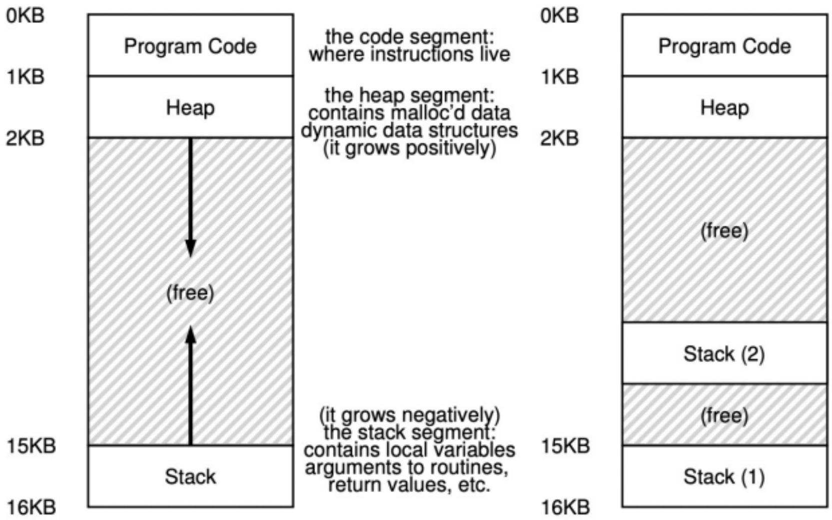
- The user time and the sys time together define **the performance of an application**
  - When writing a program, you must consider both the user time and the sys time
- Ideally, Real time = User time + Sys time
  - real > user + sys: **I/O intensive**
  - real < user + sys: **multi-core**

6. Threads

What is a Thread

- 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 register
  - 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

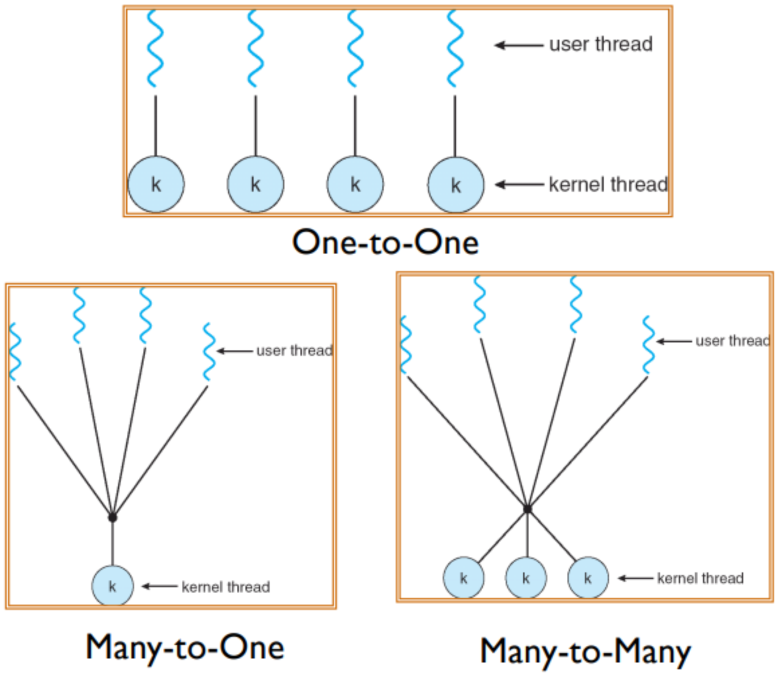
- Increase **parallelism**
  - One thread per CPU makes better use of multiple CPUs to improve efficiency
- Avoid **blocking problem** progress due to slow I/O
  - Threading enables overlap of I/O with other activities within a single program
  - many modern server-based applications (web servers, database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

Thread Implementation

- **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**
    - Thread 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
    - Pros: Every thread can run or block independently
    - Cons: Need to make a crossing into kernel mode to schedule
- **Many-to-one mapping**
    - Many user-level thread to one kernel-level thread
    - Pros: context switch between threads is cheap
    - Cons: When one thread blocks on I/O, all threads block
- **Many-to-many mapping**
    - Many user-level thread to many kernel-level thread
    - Pros: best of the two worlds, more flexible
    - Cons: difficult to implement