操作系统 Operating System

南方科技大学 计算机科学与工程系 11812804 董正

前言 Preface

笔记结构基于《操作系统概念（第九版）》

Based on *Operating System Concepts Ninth Edition*

第三章 进程 Process

3.1 基本概念

- CPU 活动
  - 批处理系统: 作业 (job)
  - 分时系统: 用户程序 (user program) 或任务 (task)

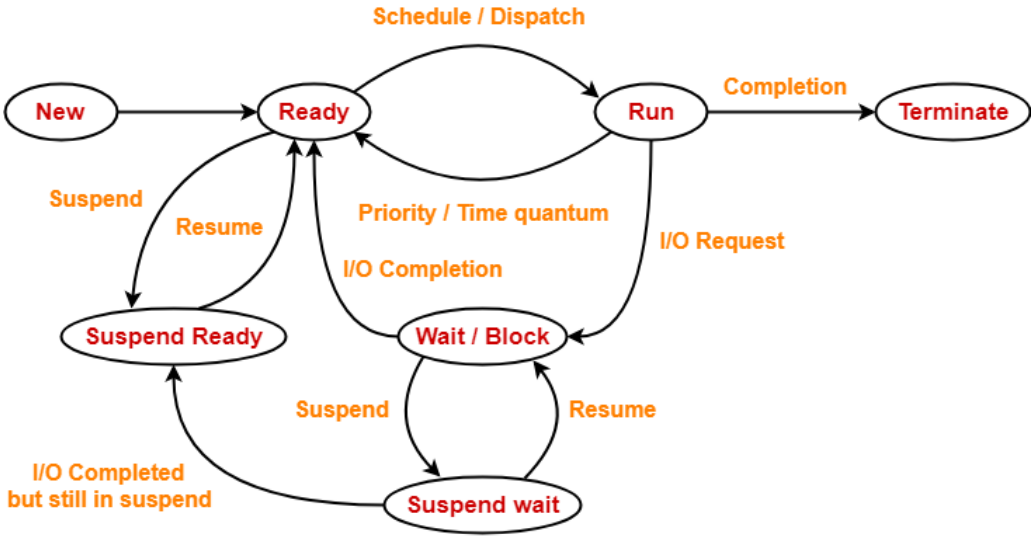
3.1.1 进程的概念

- **进程 (Process)** 是执行的程序

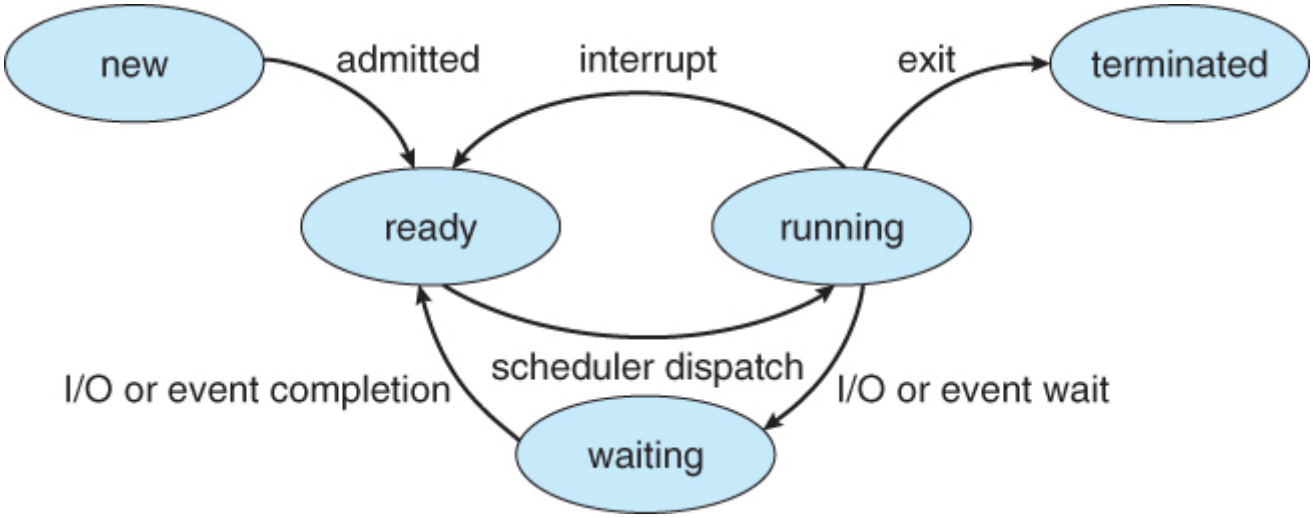
  Process is a program in execution
- 进程还包括:
  - 程序计数器 PC
  - 寄存器 (Register) 的内容
  - 堆 Heap
  - 栈 Stack
  - 数据段 Data Section
  - 文本段 Text Section
  - ...
- 程序 (Program) 和进程
  - 程序是**被动实体 (passive entity)**，如存储在磁盘上包含一系列指令的文件，经常称为可执行文件 (executable file)
  - 进程是**活动实体 (active entity)** 或称**主动实体**，具有一个程序计数器用来表示下个执行命令和一组相关资源
  - 当一个可执行文件被加载到内存时，这个程序就成为进程

3.1.2 进程的状态 Process States

状态	英文	说明
新的	new	进程正在创建
运行	running	指令正在执行
等待	waiting/blocked	进程等待发生某个事件，如 IO 完成或收到信号
就绪	ready	进程等待分配处理器
终止	terminated	进程已经完成执行



Process State Diagram

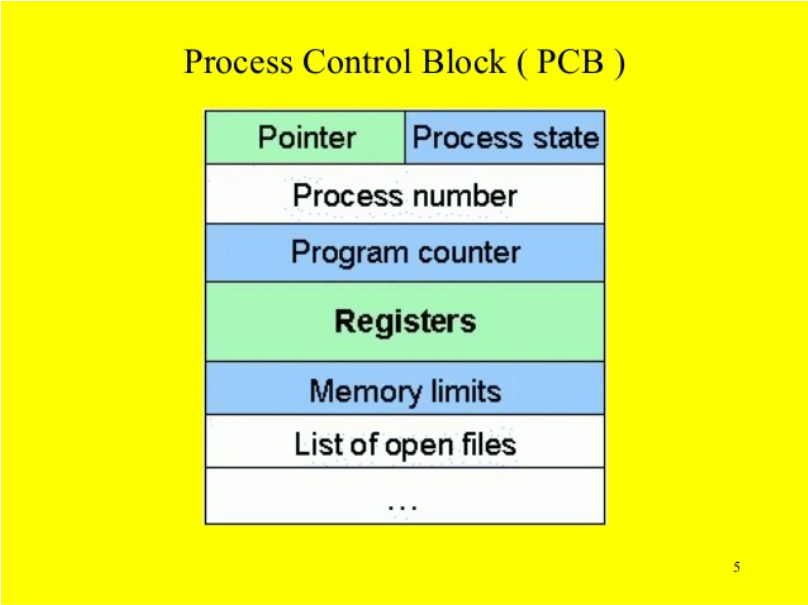


- 等待状态又分为
  - 可中断 (Interruptable)
  - 不可中断 (Un-interruptible)
- 刚 fork 的进程都会变成 ready 状态

3.1.3 进程控制块 Process Control Block

进程控制块 PCB (任务控制块 Task Control Block)

在内存 (Main Memory) 里



PCB 是系统感知进程存在的唯一标志

- 进程状态 Process State
- 程序计数器 PC
- CPU 寄存器 CPU Register
- CPU 调度信息 CPU-scheduling Infomation

  进程优先级，调度队列的指针和其他调度参数
- 内存管理信息 Memory-management Infomation

  基地址，界限寄存器的值，页表或段表等
- 记账信息 Accounting Infomation

  CPU 时间，实际使用时间，时间期限，记账数据，作业或进程数量等
- IO 状态信息 IO Status Infomation

  分配给进程的 IO 设备列表，打开文件列表等

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.

- 几个概念

- PCB = 进程表 = `task_struct` in Linux
- Task list = PCB 组成的双向链表

3.2 进程生命周期

3.2.1 进程标识符 Process Identifier

- 进程标识符 Process Identifier (PID)

  系统的每个进程都有一个唯一的整数 PID

  System call `getpid()`: return the PID of the calling process

- `init` 进程

  PID = 1, 所有用户进程的父进程或根进程

  代码位于 `/sbin /init`

  它的第一个任务是创建进程 `fork()+exec*()`

3.2.2 进程创建 Process Creation

- System call `fork()`: creates a new process by duplicating the calling process.

  fork 出的子进程从 fork 调用的下一行开始执行（因为 PC 也复制了）

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

- **`fork()` does not clone the following...**
- Note: PCB is in the kernel space.

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.

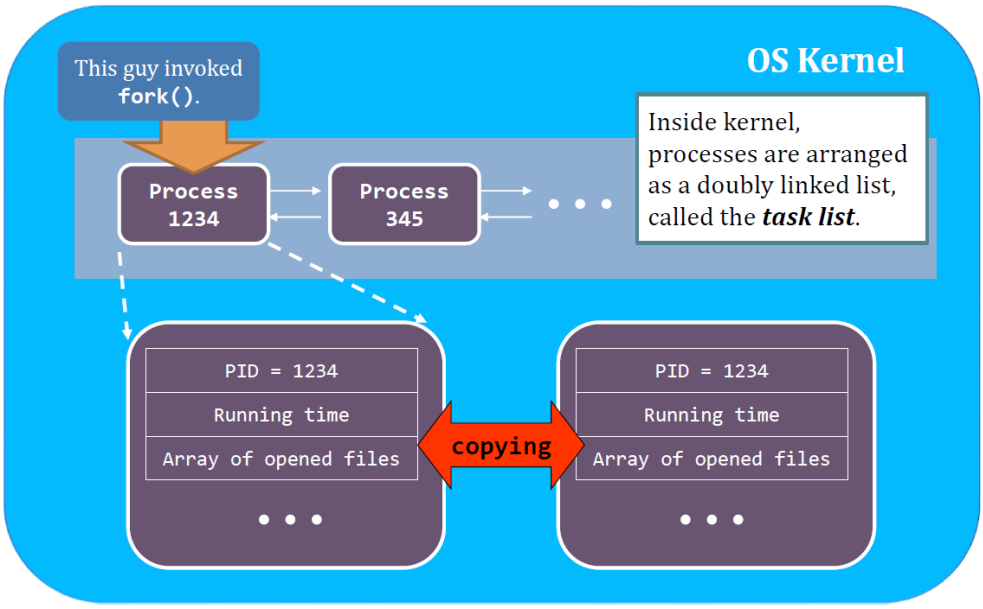
- 在代码中如何区分父进程和子进程

  `pid=fork()`, 则父进程中 `pid` 变量等于子进程的 PID， 子进程中 `pid=0`

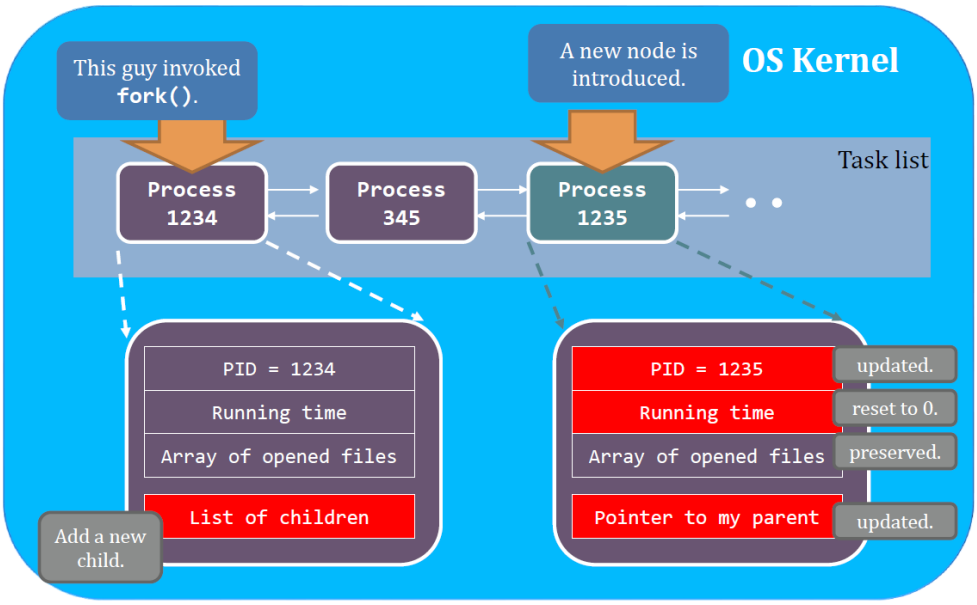
```c
if (!pid) {
    // 只有子进程执行
}
else {
    // 只有父进程执行
}
```

- 父进程和子进程执行顺序不确定
- `fork` 的流程, 内核空间的动作

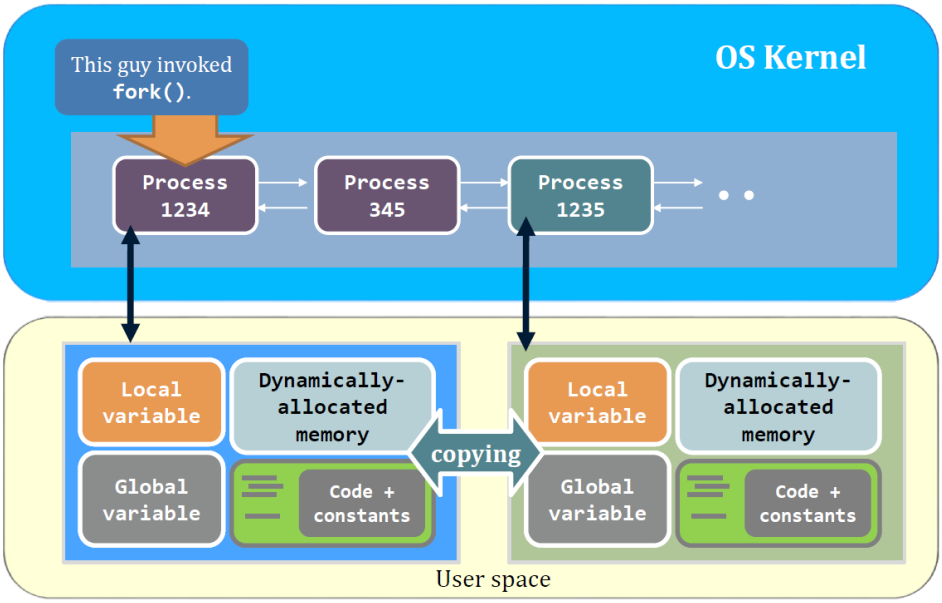
  1. 复制 PCB



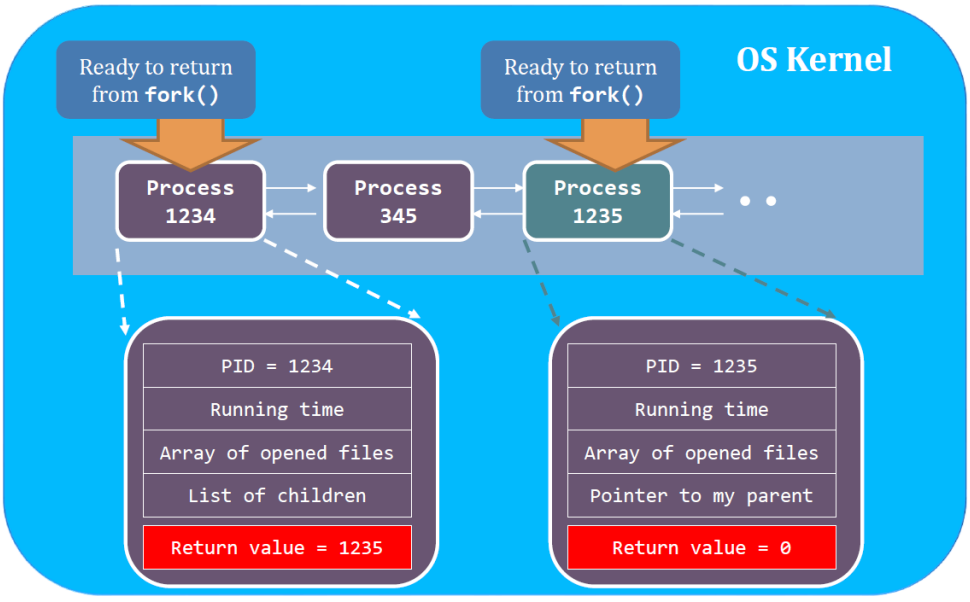
  2. 更新 PCB 和 task list



3. 复制用户空间



4. return



3.2.3 进程执行 Process Execution

- System call `exec*()`
- Example: `ls -l`

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

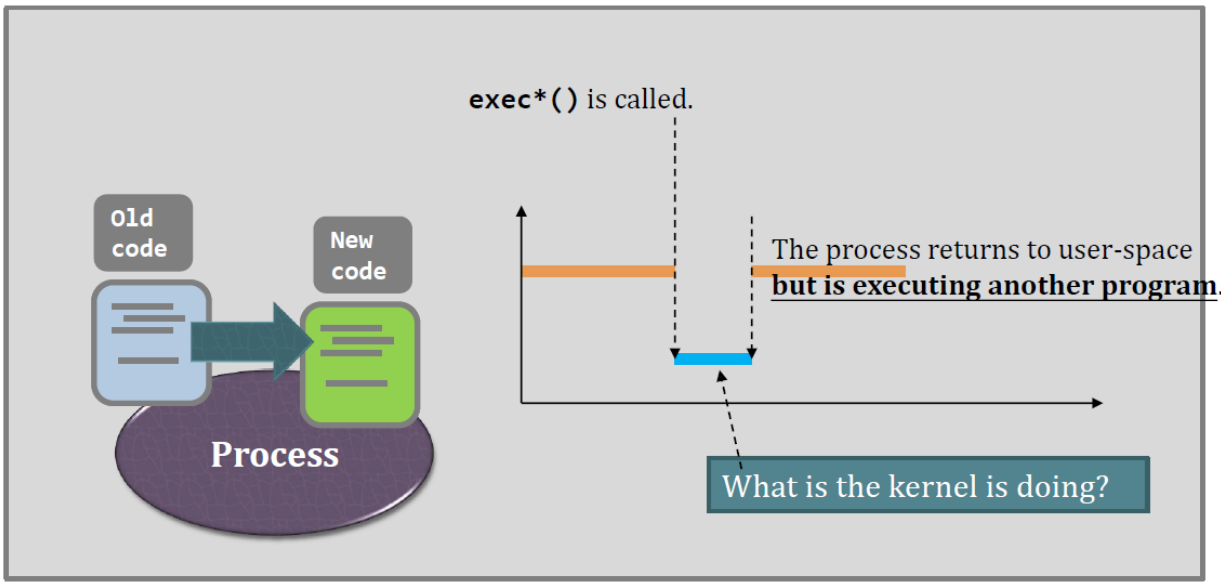
Argument Order	Value in above example	Description
1	`"/bin/ls"`	The file that the programmer wants to execute.
2	`"/bin/ls"`	When the process switches to `"/bin/ls"`, this string is the **program argument[0]**.
3	`"-l"`	When the process switches to `"/bin/ls"`, this string is the **program argument[1]**.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

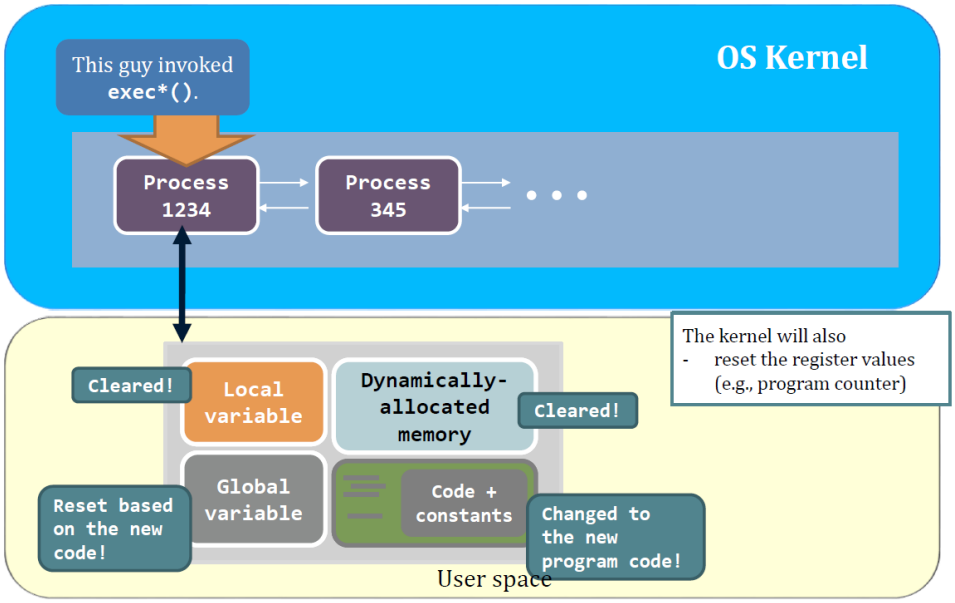
`args[0]` 是程序的名字

- The process is changing the code that is executing and never returns to the original code.

  `exec*()` 之后的代码不会执行了，因为调用之后该进程就去执行 `exec` 指定的程序了

- User space 的信息被覆盖
  - Program Code
  - Memory
    - Local Variables
    - Global Variables
    - Dynamically Allocated Memory
  - Register Value: 如 PC
- Kernel space 的信息保留: PID, 进程关系等
- `exec*()` 的内核执行过程

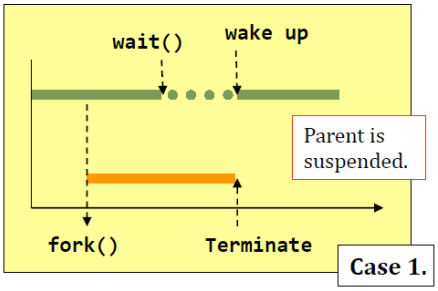


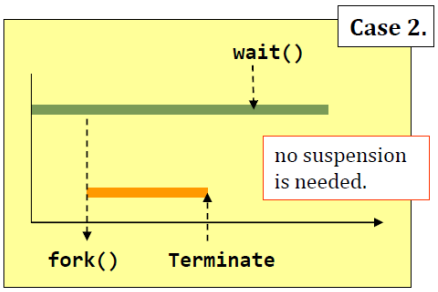


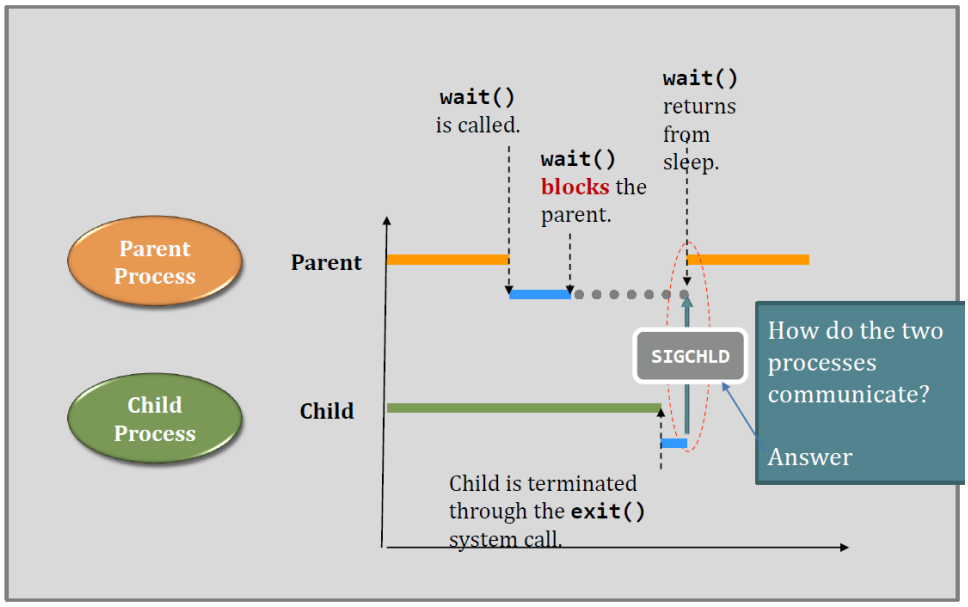
3.2.4 进程等待

1. System call `wait()`
    - Suspend the calling process to waiting state and return (wakes up) when
        - one of its child processes changes from running to terminated
        - received a signal
    - Return immediately (i.e., does nothing) if
        - it has no children
        - a child terminates before the parent calls wait
    - 给子进程收尸 见 3.2.6

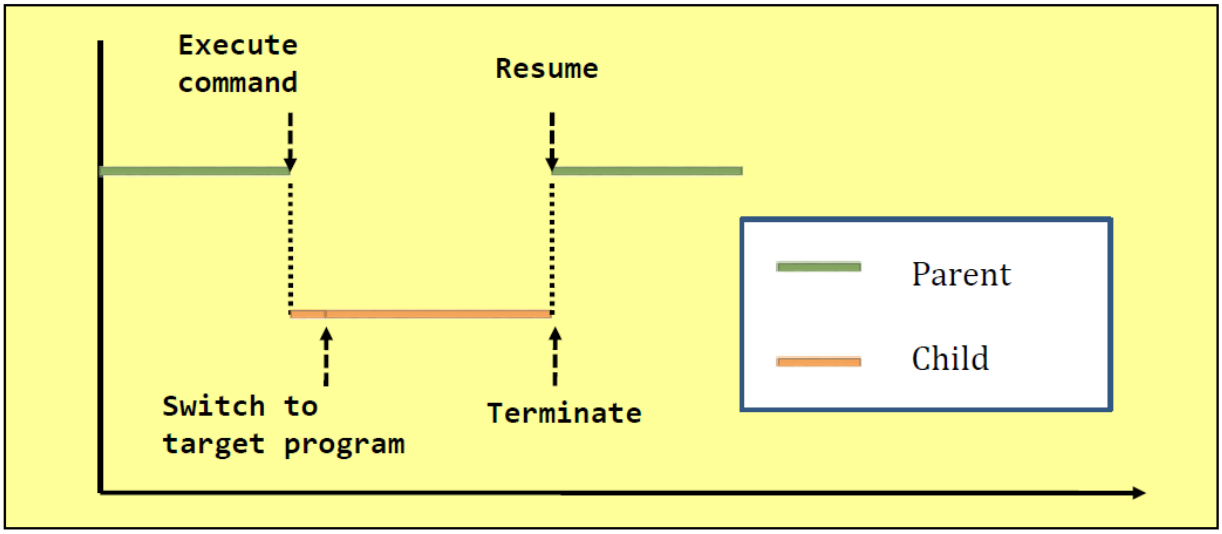
wait()	vs	waitpid()
Wait for any one of the children.		Depending on the parameters, **waitpid()** will wait for a particular child only.
Detect child **termination** only.		Depending on the parameters, **waitpid()** **can detect multiple child's status change**







2. `fork()+exec*()+wait()=system()`



例: shell 里输入命令 -> 执行相应程序 -> 程序终止 -> 返回 shell

- 除了 `init`, 所有的进程都是 `fork()+exec*()` 来的

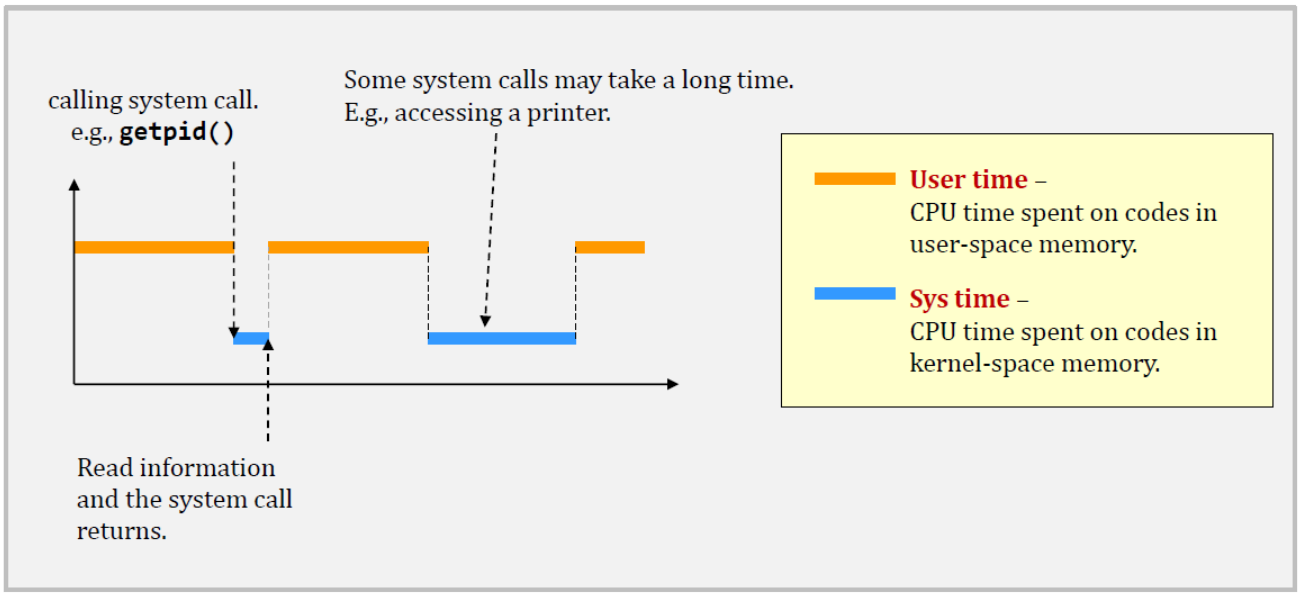
3.2.5 进程时间

- 实际时间 Real Time

  Wall-clock time
- 用户时间 User Time

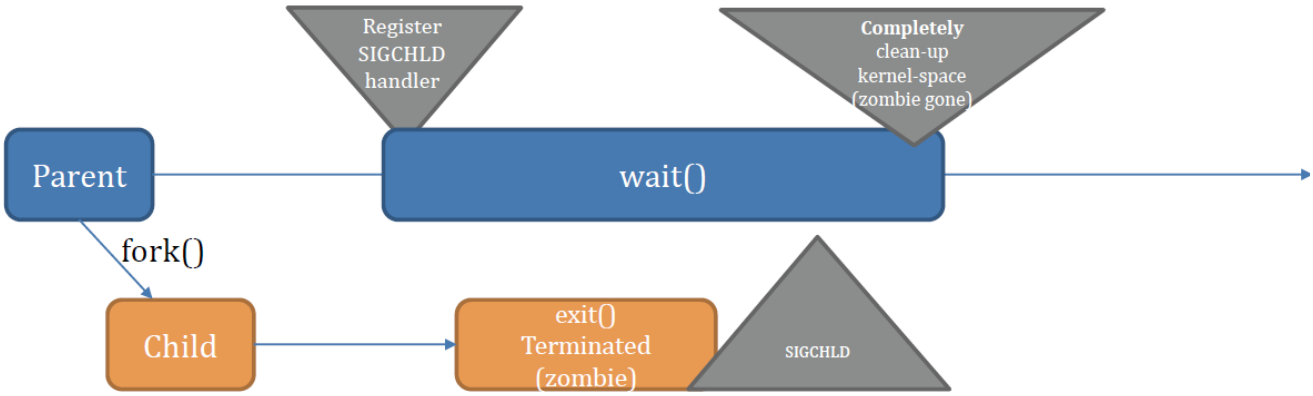
  CPU 在用户空间花费的时间
- 系统时间 System Time

  CPU 在内核空间花费的时间

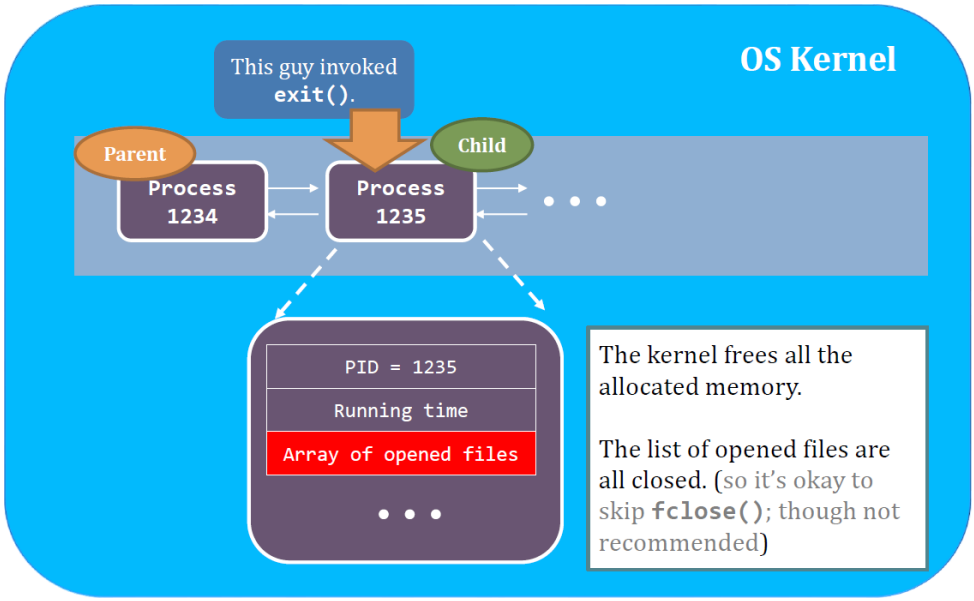


- User Time + Sys Time 决定了程序的性能 (Performance)
  - User Time + Sys Time > Real Time: 单核
  - User Time + Sys Time < Real Time: 多核

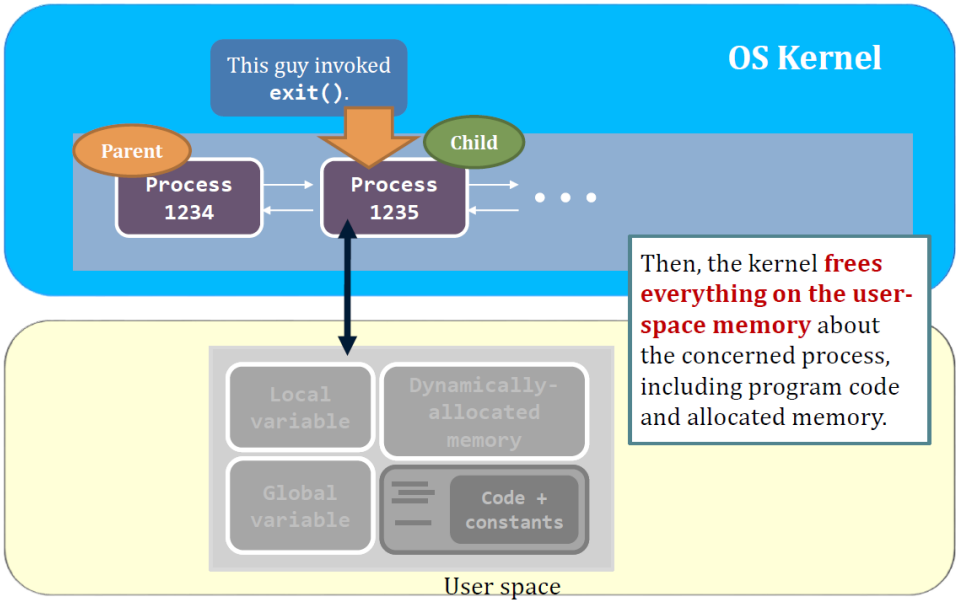
3.2.6 进程终止 Process Termination



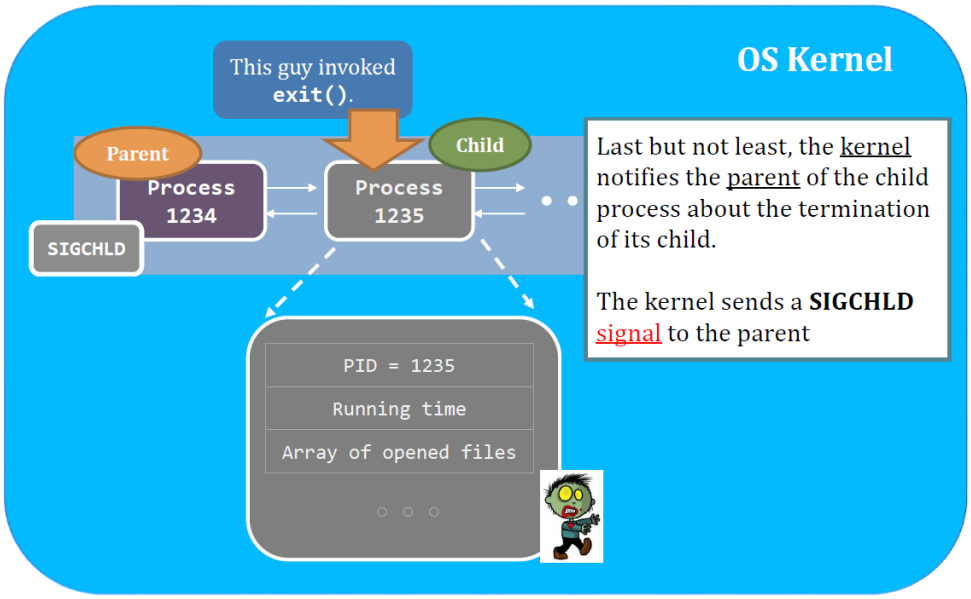
- System call `exit()`: terminate the calling process
- `exit()` 的执行过程
    1. Clean up most of the allocated kernel space memory



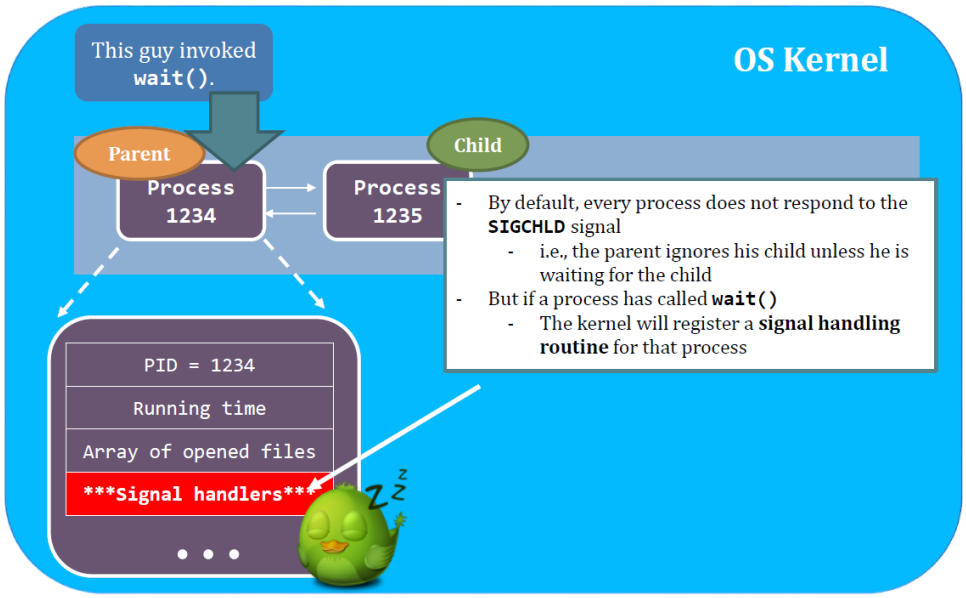
    2. Clean up the exit process's user space memory

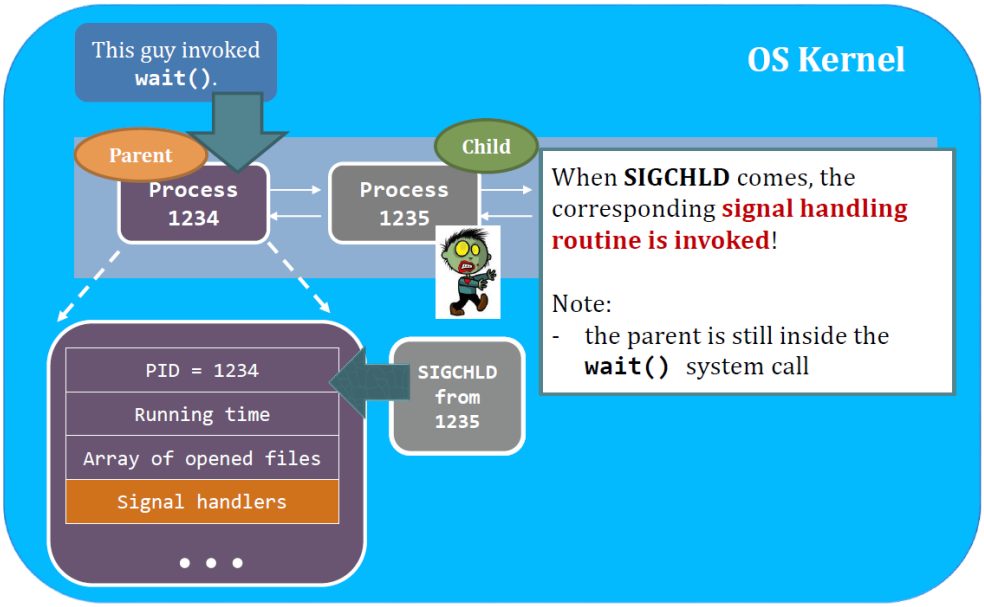


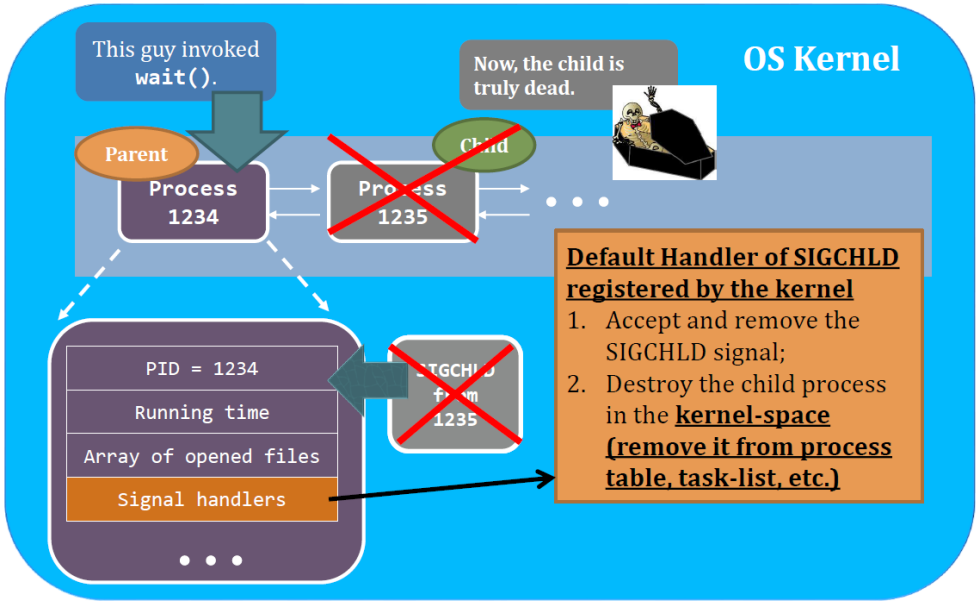
    3. Notify the parent with SIGCHLD.

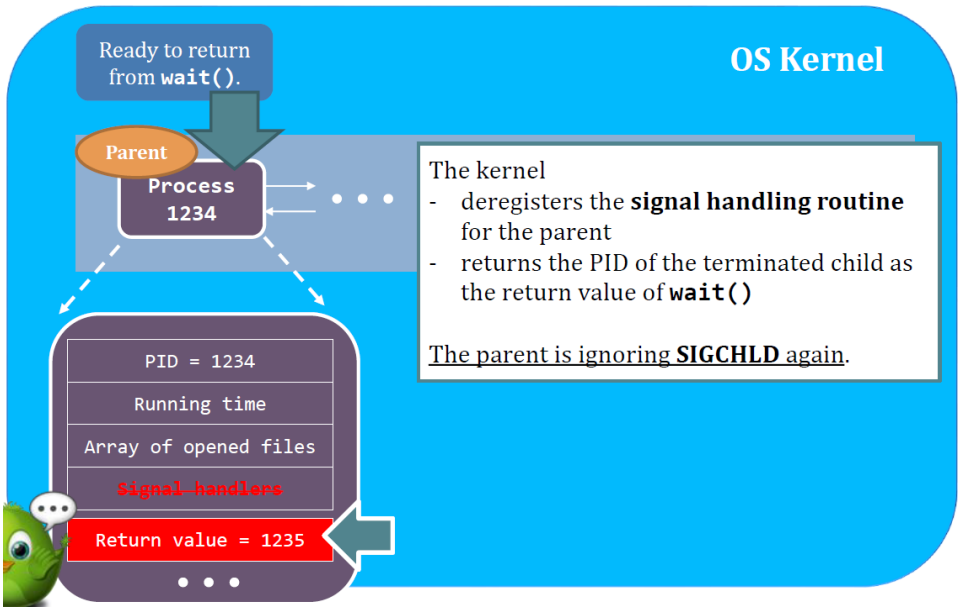


- 僵尸进程 Zombie Process
  - 进程的用户空间和内核空间被释放之后，PID 依然在进程表里，直到父进程调用 `wait()`
  - 当进程已经终止，但是其父进程尚未调用 `wait()`，这样的进程称为僵尸进程
  - 所有进程终止时都会过渡到这个状态
  - 一旦父进程调用 `wait()`，僵尸进程的进程标识符和它在进程表中的条目就会释放









- 子进程先终止，父进程再调用 `wait()` 也可以，SIGCHLD 不会消失，但是这段间隔内僵尸进程就一直存在、占用资源
- Linux 系统中僵尸进程被标为 `<defunct>`

  查看: `ps aux | grep <defunct>`

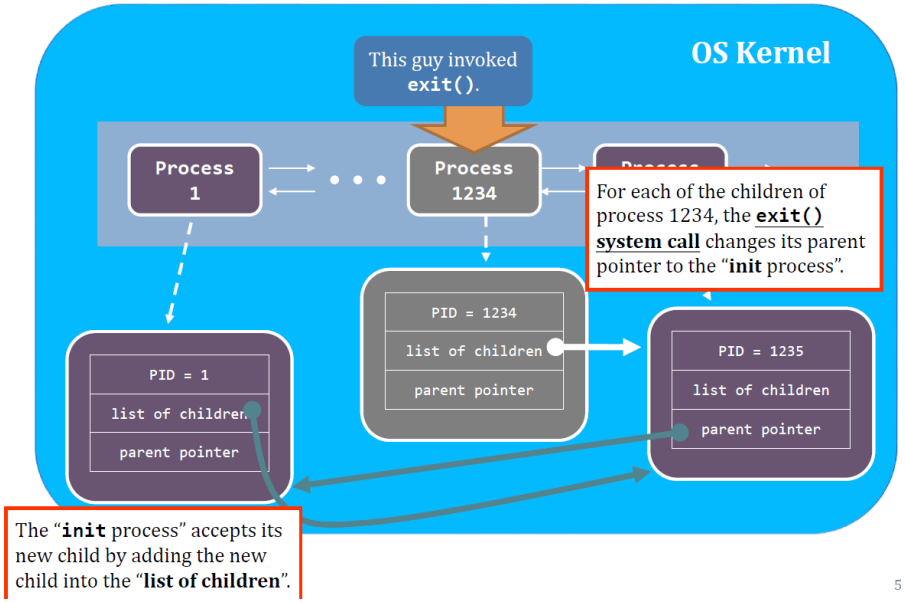
    - `exit()` system call turns a process into a zombie when
        - The process calls `exit()`
        - The process returns from `main()`
        - The process terminates abnormally

          这种情况下 kernel 会帮忙给他调用 `exit()`
- The fork bomb
    - PID 是有限的，Linux 中 PID 最大值为 32768

      `cat / proc /sys/ pid_max`
    - fork bomb (僵尸大军)

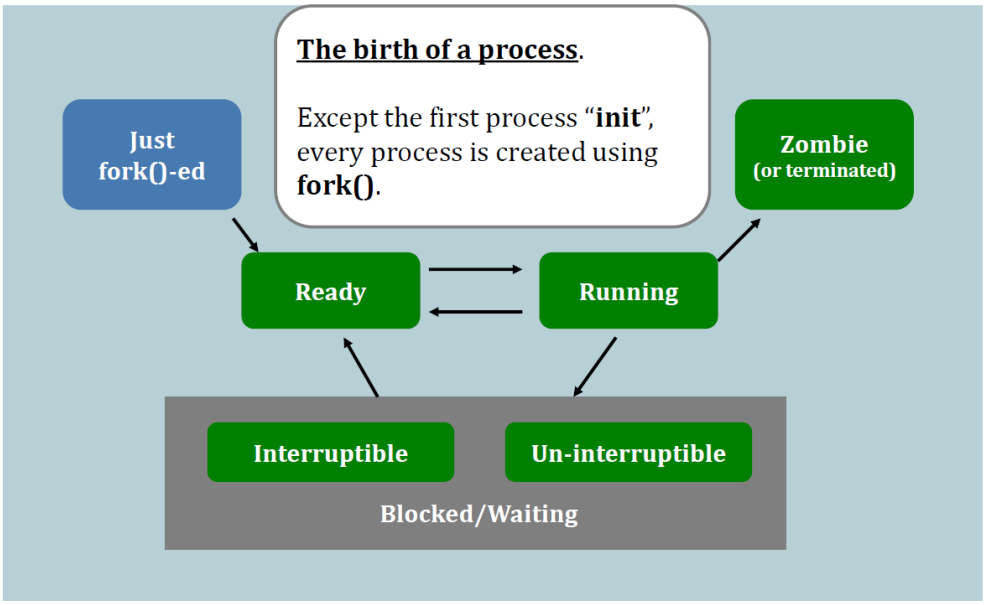
```c
int main() {
    while (fork());
    return 0;
}
```

- 孤儿进程 Orphan Process
    - 父进程没有调用 `wait()` 就终止，子进程变成孤儿进程
    - Linux & UNIX: 将 `init` 进程作为孤儿进程的父进程 (Re-parent)
    - `init` 进程定期调用 `wait()` 以便收集任何孤儿进程的退出状态，并释放孤儿进程标识符和进程表条目

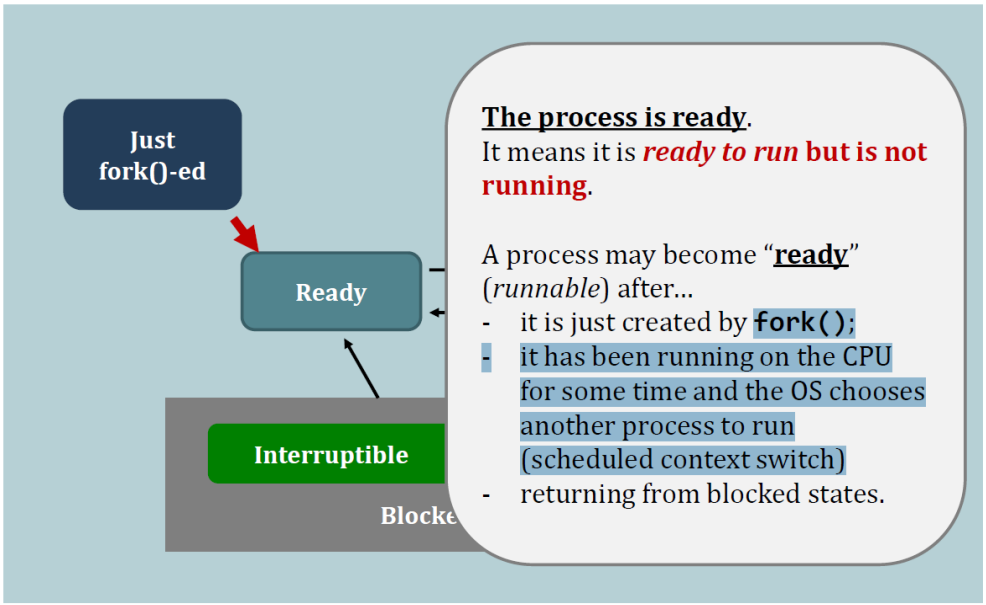


3.2.7 进程生命周期 Process Lifecycle

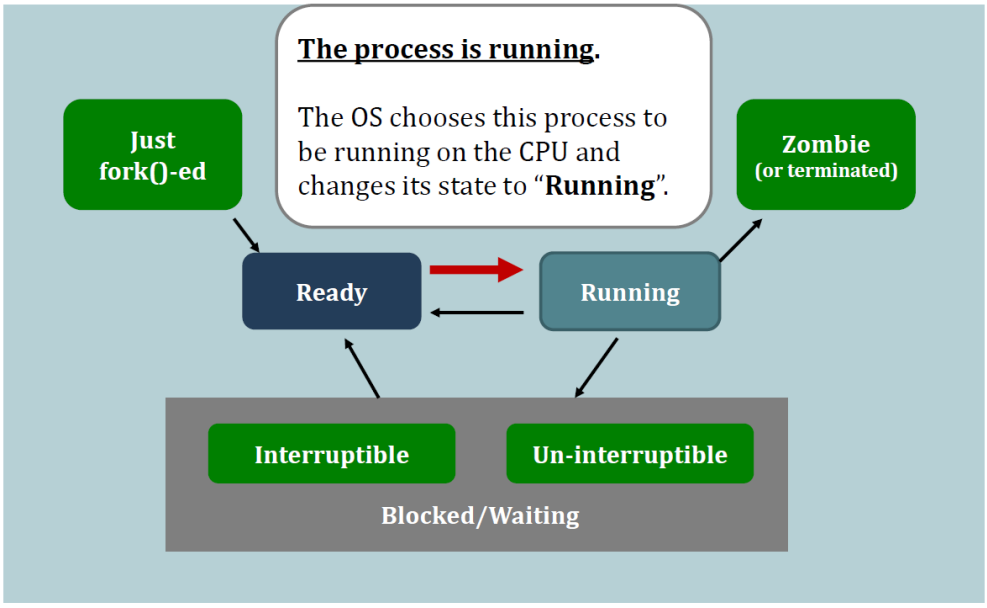
1. forked



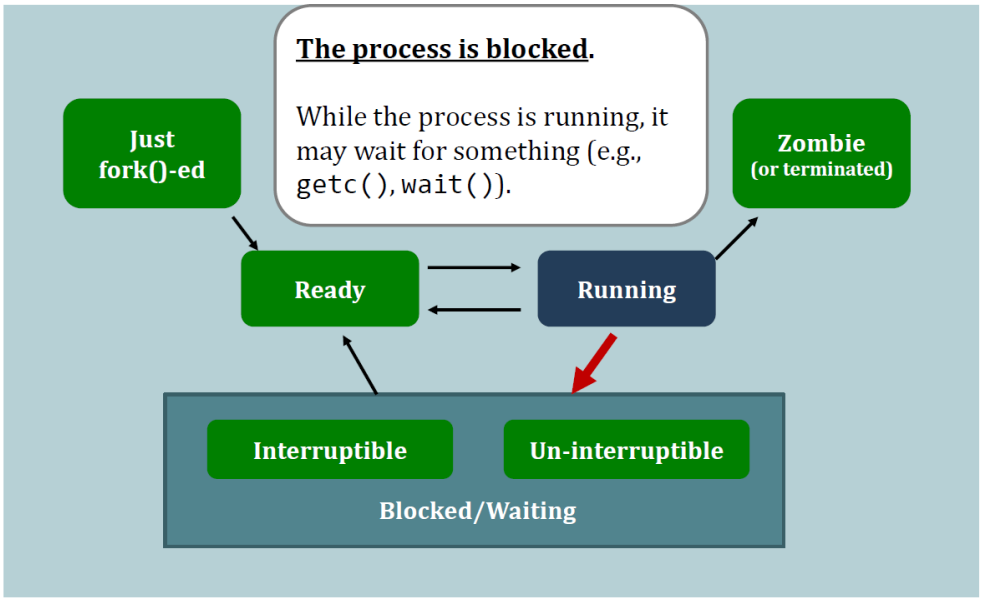
2. Ready



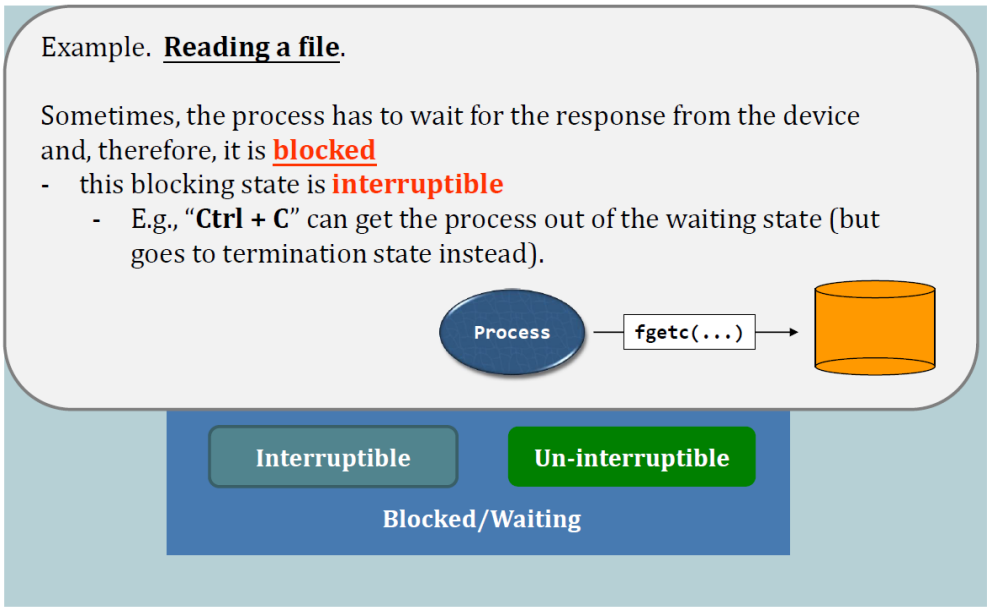
3. Running



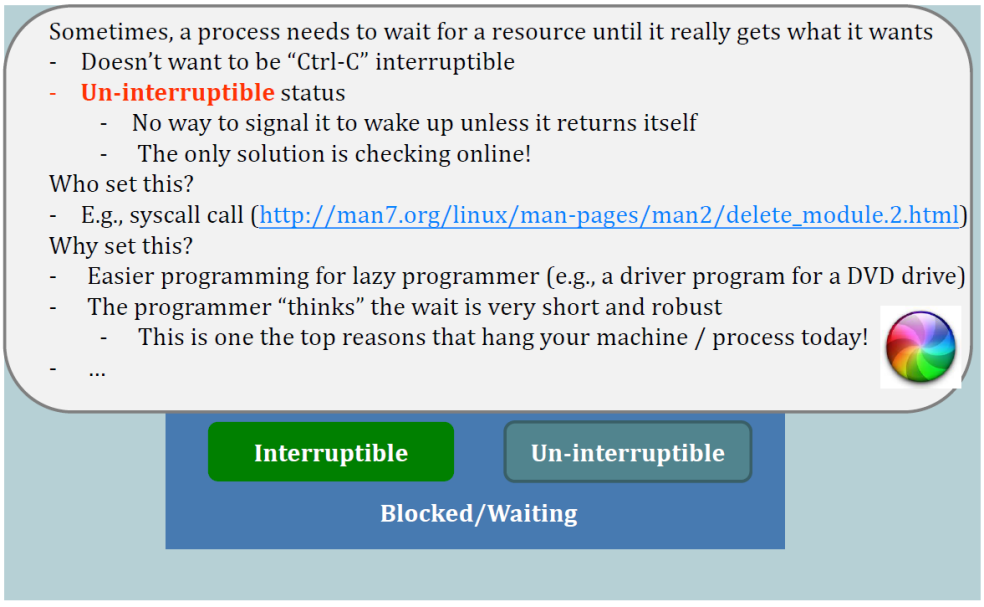
4. Blocked



5. Interruptable waiting

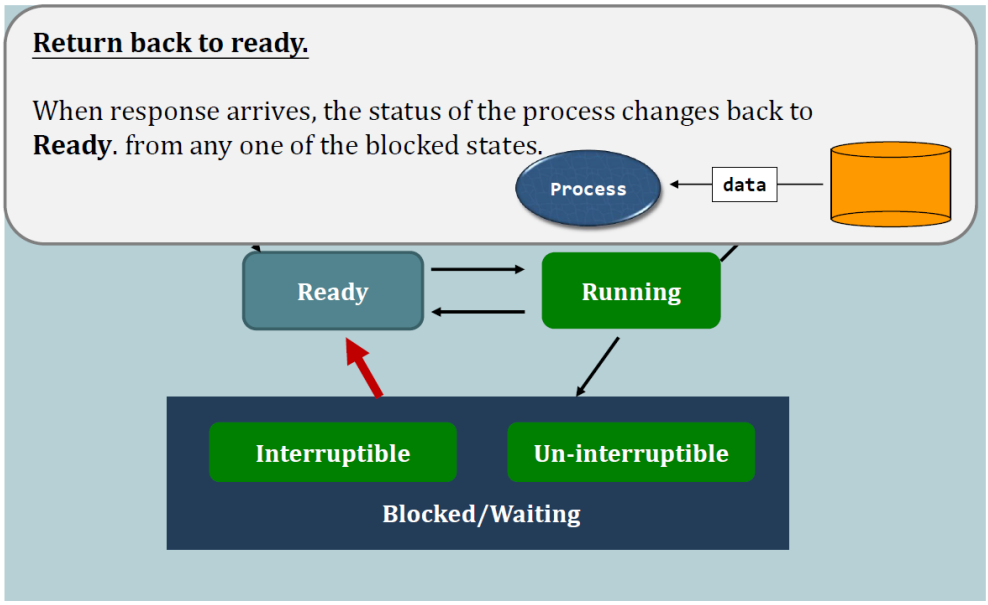


6. Un-interruptable waiting

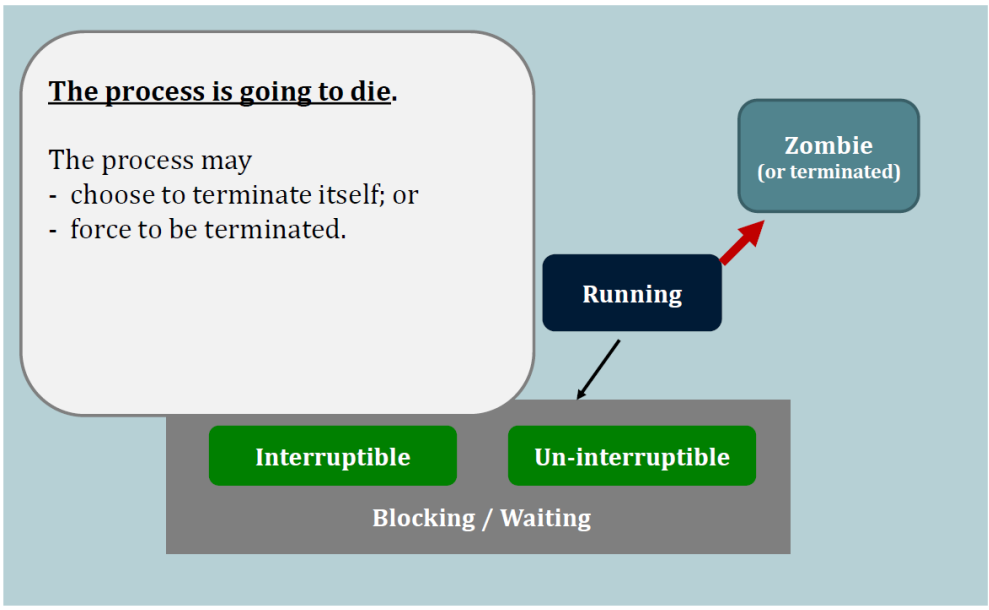


计网的程序里经常碰见，纯贵物，谁设计的抓紧埋了吧

7. Return back to ready



8. Terminated

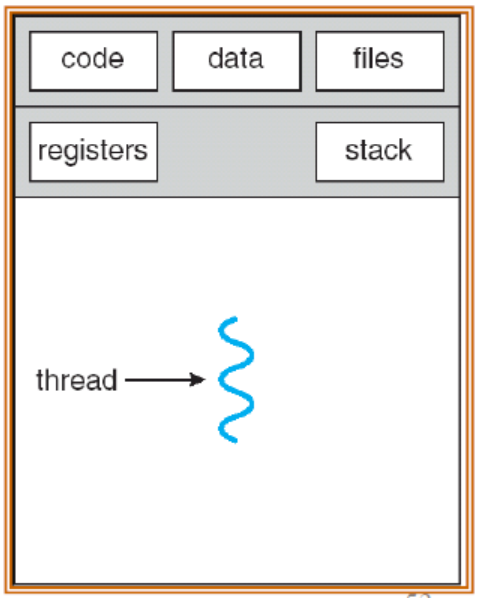


第四章 线程 Thread

4.1 线程的概念

- Heavyweight Process

  A process has a single thread of control



- 线程 Thread

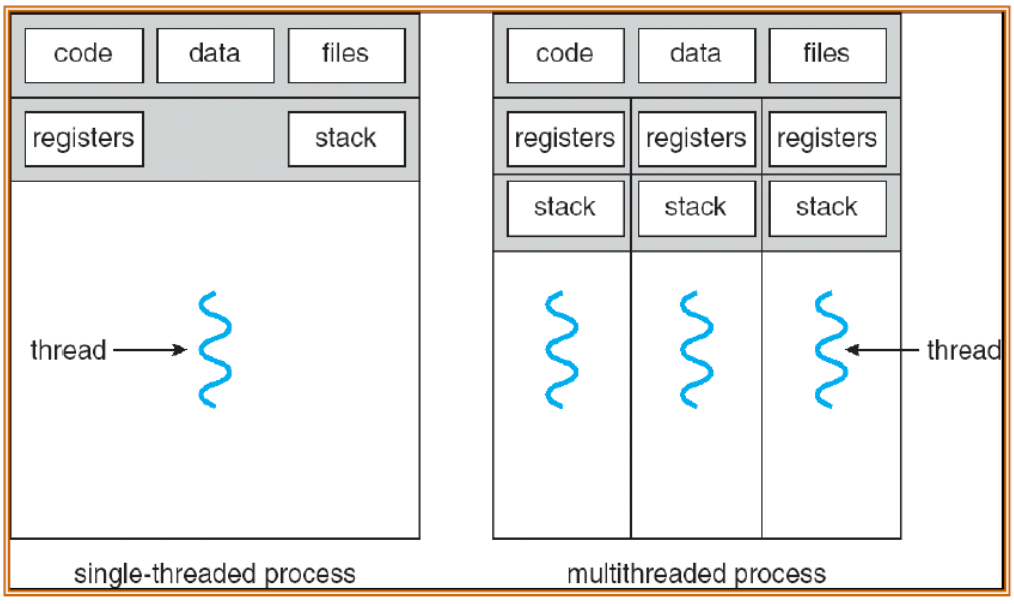
  A sequential execution stream within process

  又称 Lightweight Process

  - Process still contains a single Address Space
  - No protection between threads
- 多线程 Multithreading

  A single program made up of a number of different concurrent (并发) activities

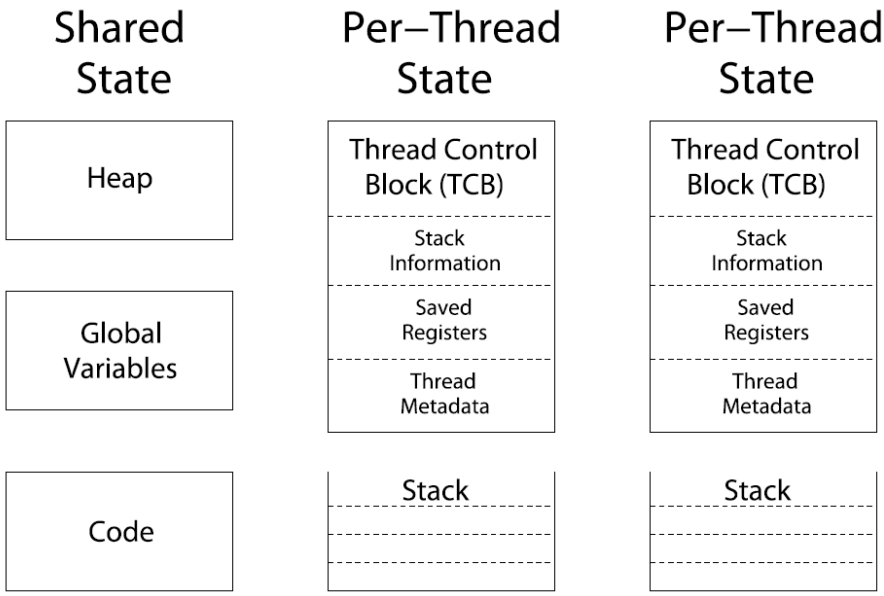
- 结构图



    - Threads encapsulate **concurrency**: "Active" component
    - Address spaces encapsulate **protection**: "Passive" part

4.2 线程的组成

- State shared by all threads in process/address space
    - Content of memory (global variables, heap)
    - I/O state (file descriptors, network connections, etc)
- State "private" to each thread
    - Kept in TCB (Thread Control Block)
    - CPU registers (including PC)
    - Execution stack
- 栈
    - Parameters, temporary variables
    - Return PCs are kept while called procedures are executing
    - 回忆计组学的，不多说

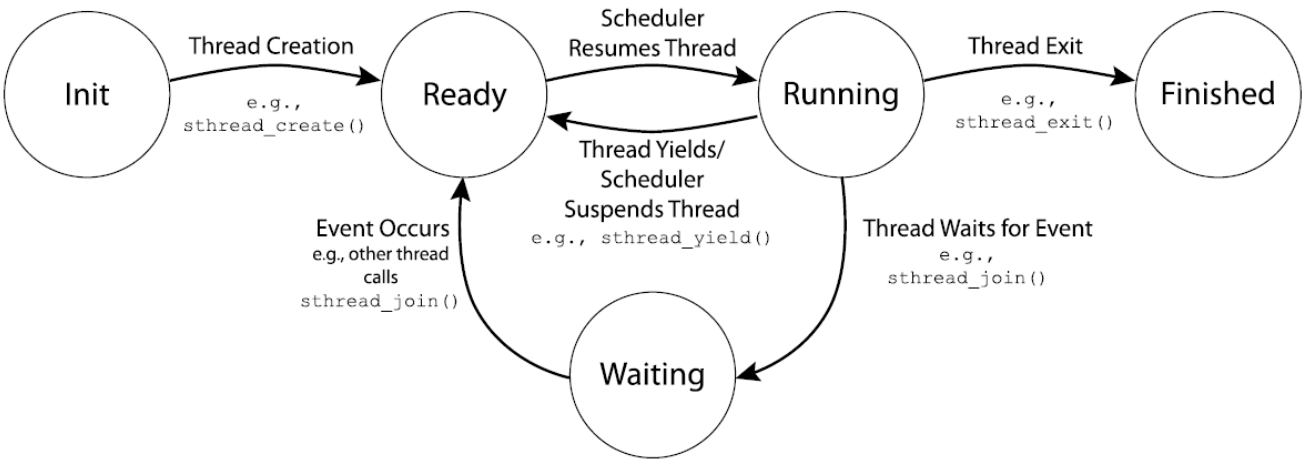


4.3 线程和进程的区别

Process	Thread
Process means any program is in execution.	Thread means segment of a process.
Process takes more time to terminate.	Thread takes less time to terminate.
It takes more time for creation.	It takes less time for creation.
It also takes more time for context switching.	It takes less time for context switching.

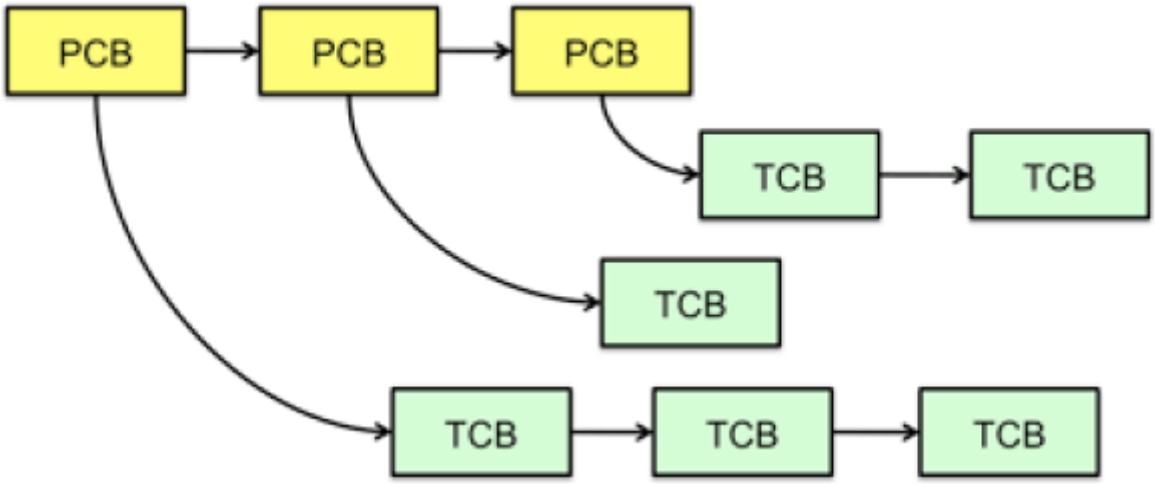
Process	Thread
Process is less efficient in term of communication.	Thread is more efficient in term of communication.
Process consume more resources.	Thread consume less resources.
Process is isolated.	Threads share memory.
Process is called heavy weight process.	Thread is called light weight process.
Process switching uses interface in operating system.	Thread switching does not require to call a operating system and cause an interrupt to the kernel.
If one process is blocked then it will not effect the execution of other process	Second thread in the same task couldnot run, while one server thread is blocked.
Process has its own Process Control Block, Stack and Address Space.	Thread has Parents' PCB, its own Thread Control Block and Stack and common Address space.

4.4 线程的生命周期 Thread Lifecycle



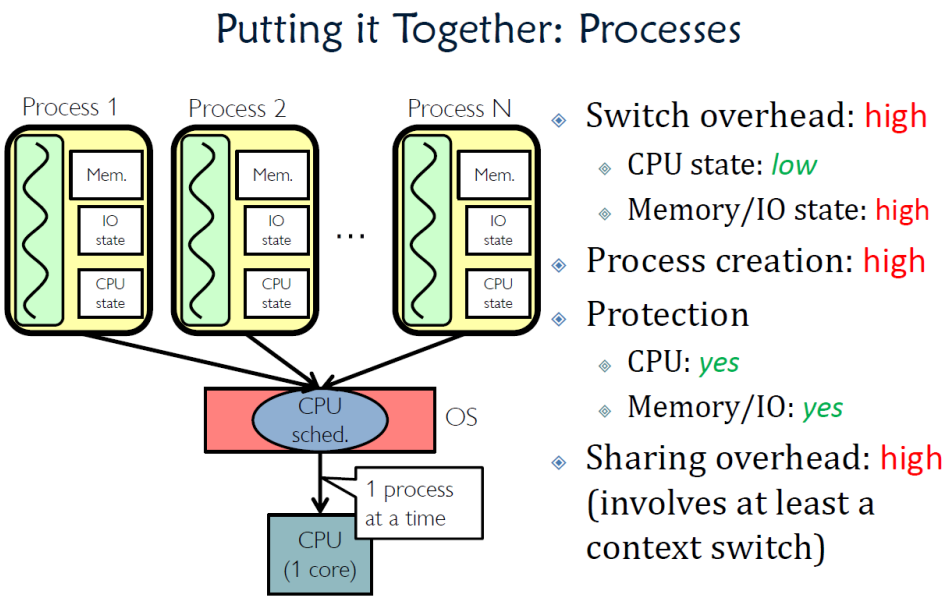
4.5 多线程进程 Multithreaded Process

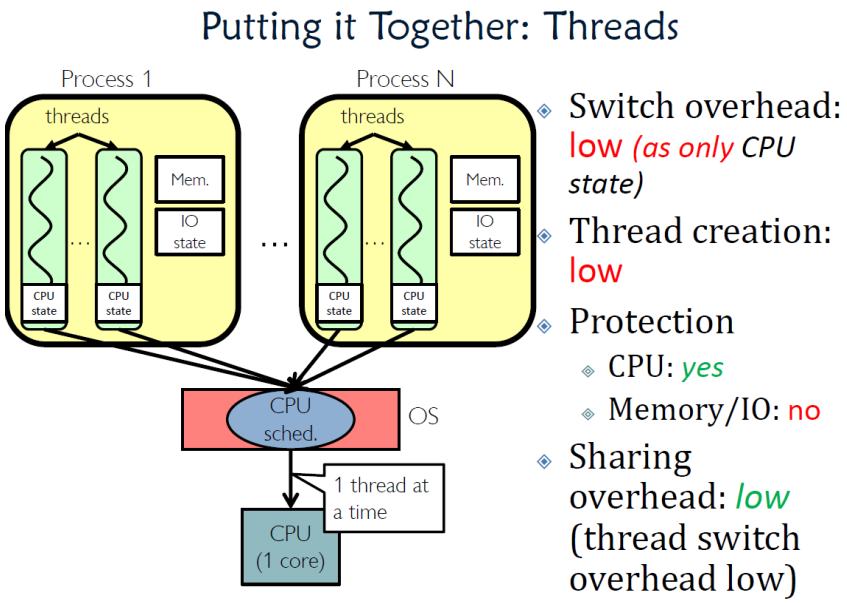
- PCB 指向多个 TCB

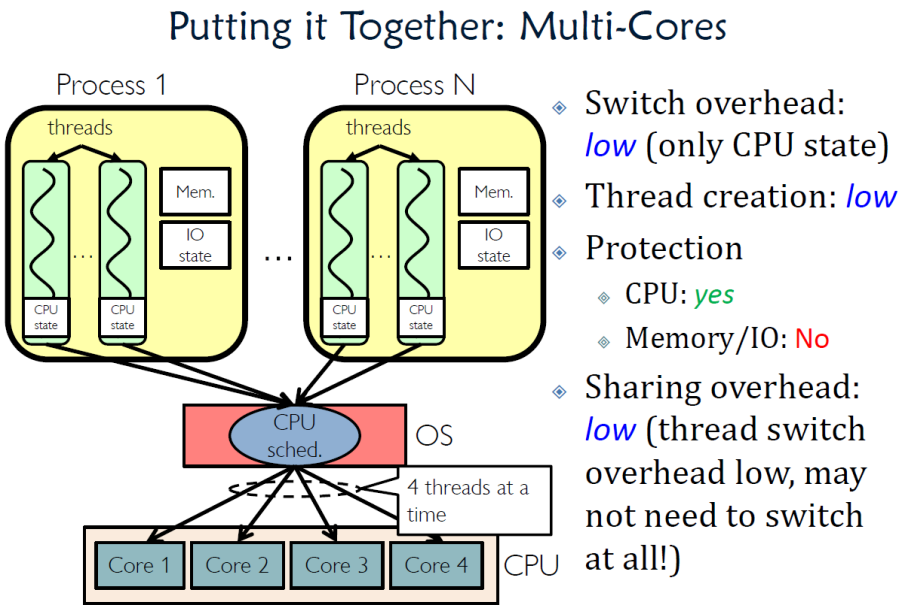


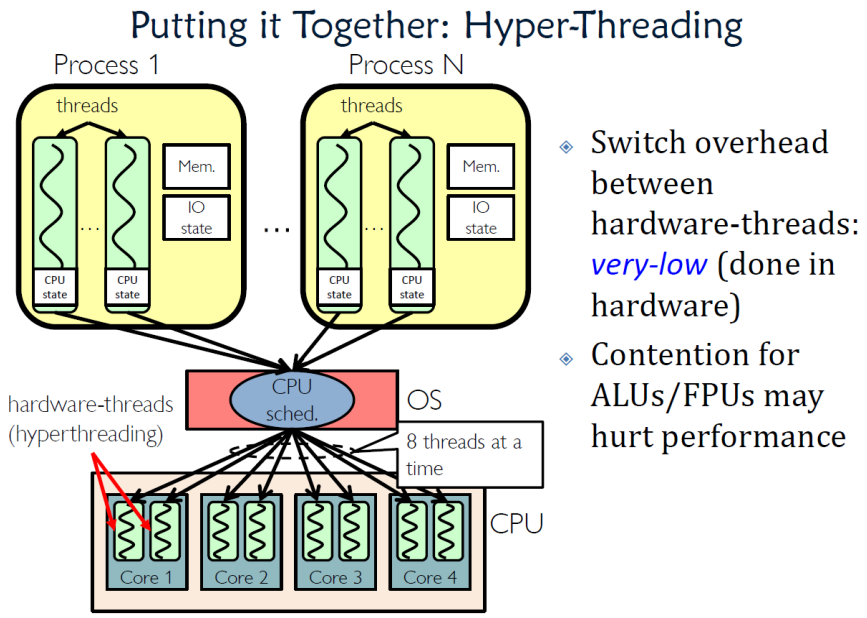
- Switching threads within a block is a simple thread switch
- Switching threads across blocks requires changes to memory and I/O address tables

4.6 多线程调度









4.7 Multiprocessing, Multithreading and Multiprogramming

- 多进程 Multiprocessing

  Multiple CPUs

  A computer using more than one CPU at a time.
- 多线程 Multithreading

  Multiple threads per Process
- 多道程序设计 Multiprogramming

  Multiple Jobs or Processes

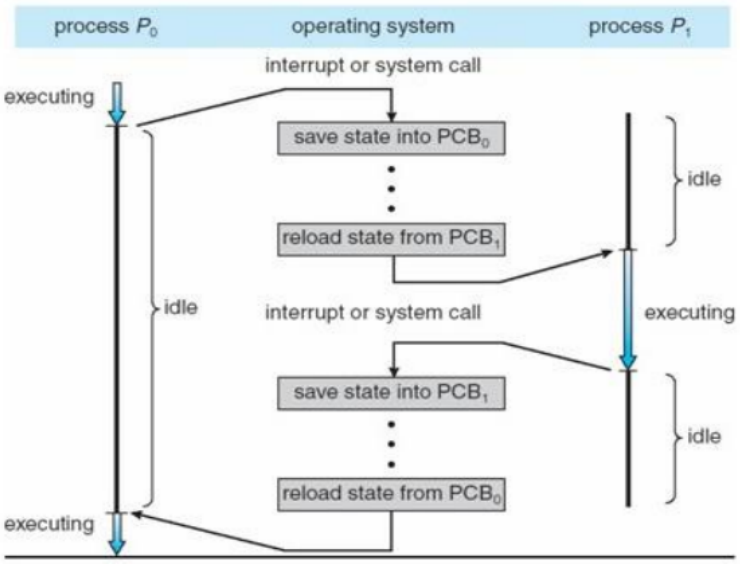
  A computer running more than one program at a time

第五章 进程调度 Process Scheduling

5.1 基本概念

5.1.1 上下文切换 Context Switch

- 切换 CPU 到另一个进程需要**保存**当前进程状态和**恢复**另一个进程的状态，这个任务称为上下文切换
- 当进行上下文切换时，内核会将旧进程的状态保存在其 **PCB** 中，然后加载经调度而要执行的新进程的上下文
- 上下文切换是纯粹的时间开销 (Overhead)，因为 CPU 在此期间没有做任何有用工作
- 上下文切换非常耗时



5.1.2 调度队列

- 作业队列 Job Queue

  包含所有进程
- 就绪队列 Ready Queue

  等待运行的进程

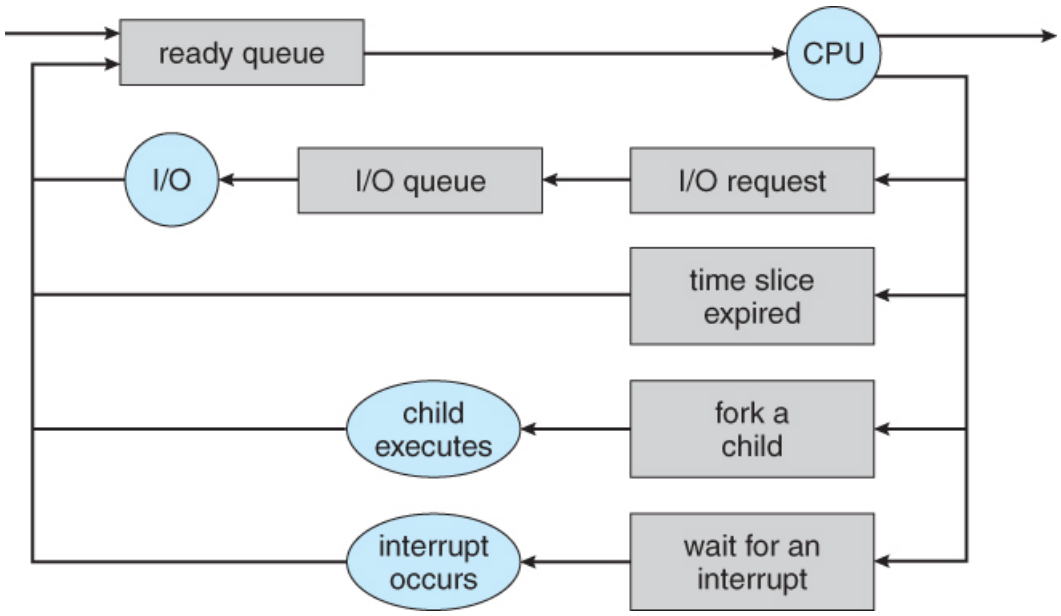
  PCB 构成的链表
- 设备队列 Device Queue

  等待使用该 IO 设备的进程队列

  每个设备都有
- 队列图 Queueing Diagram

  圆圈代表服务队列的资源， 箭头代表系统内的进程流向

  ↓执行或分派 (Dispatch)



5.1.3 调度程序 Scheduler

- 缓冲池

  通常来说，对于批处理系统，提交的进程多于可执行的，这些进程被保存到大容量存储设备（如磁盘）的缓冲池，以便以后执行

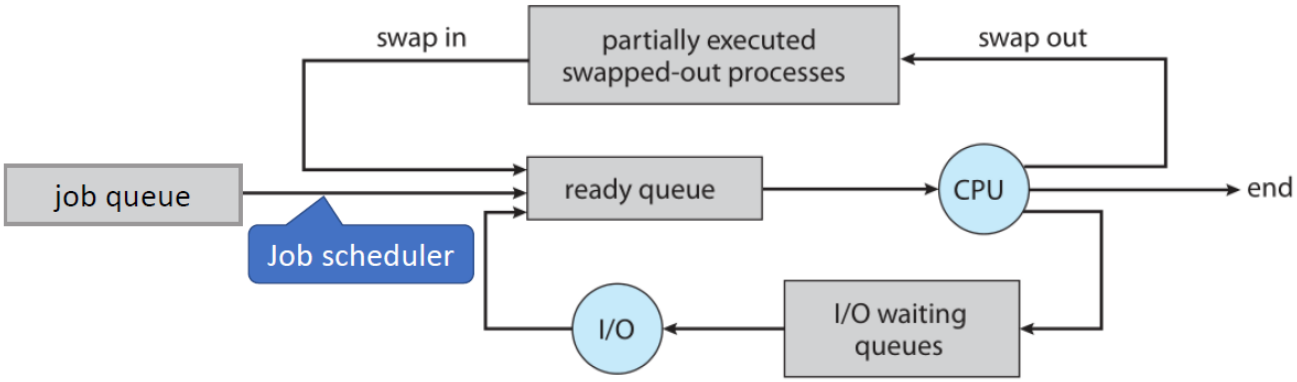
- 调度程序 (调度器)

调度器	别名	作用
长期调度程序 Long-term Scheduler	作业调度程序 Job Scheduler	从缓冲池中选择进程加载到内存
短期调度程序 Short-term Scheduler	CPU 调度程序	从 Ready Queue 中选择进程并分配 CPU
中期调度程序 Medium-term Scheduler		进程交换

- 进程分类

中文	英文	特点
I/O 密集型进程	I/O Bounded Process	执行 I/O 比执行计算耗时
CPU 密集型进程	CPU Bounded Process	很少I/O, 执行计算用时长

  长期调度程序需要选择这两种进程的合理组合才能最大化 CPU 和 IO 设备的利用



5.1.4 Dispatcher

Dispatcher 是一个模块，用来将 CPU 控制交给由 CPU 调度程序选择的进程

- 功能
  - 切换上下文
  - 切换到用户模式
  - 跳转到用户程序的合适位置，以便重新启动程序
- 调度延迟 Dispatch Latency

  Dispatcher 停止一个进程而启动另一个进程所需的时间
- Dispatcher 和 Scheduler 的区别

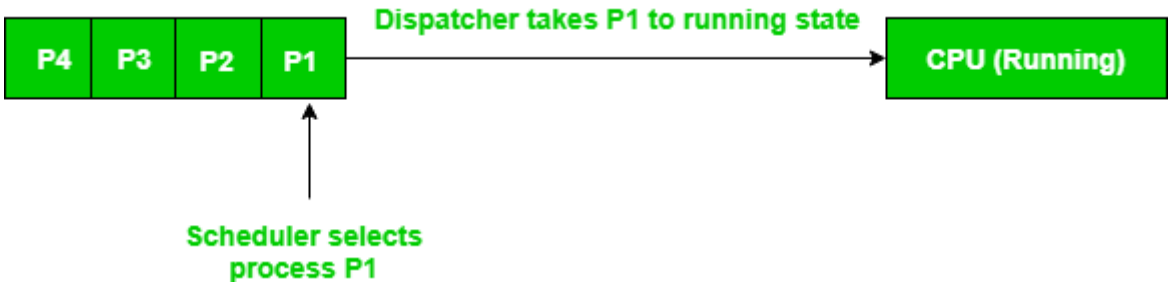
  中文书把 dispatcher 也翻译成调度程序，我真想一拳干碎你的眼镜

  https://www.differencebetween.com/difference-between-scheduler-and-vs-dispatcher

  https://www.geeksforgeeks.org/difference-between-dispatcher-and-scheduler

> The **key difference** between scheduler and dispatcher is that **the scheduler selects a process out of several processes to be executed while the dispatcher allocates the CPU for the selected process by the scheduler.**

Scheduler vs Dispatcher	
A scheduler is special system software that handles process scheduling by selecting the process to execute.	The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler.
Types	
There are three types of schedulers known as; • long-term scheduler, • short-term scheduler • medium term scheduler.	There is no categorization for a dispatcher.
Main Tasks	
The **long-term scheduler** selects the process from the job queue and brings it to the ready queue. The **short term scheduler** selects a process in the ready queue. The **medium scheduler** carries out the swap in, swap out of the process.	The dispatcher allocates the CPU to the process selected by the short-term scheduler.



Properties	DISPATCHER	SCHEDULER
Definition	Dispatcher is a module that gives control of CPU to the process selected by short term scheduler	Scheduler is something which selects a process among various processes
Types	There are no different types in dispatcher. It is just a code segment.	There are 3 types of scheduler i.e. Long-term, Short-term, Medium-term
Dependency	Working of dispatcher is dependent on scheduler. Means dispatcher have to wait until scheduler selects a process.	Scheduler works independently. It works immediately when needed
Algorithm	Dispatcher has no specific algorithm for its implementation	Scheduler works on various algorithm such as FCFS, SJF, RR etc.
Time Taken	The time taken by dispatcher is called dispatch latency.	Time taken by scheduler is usually negligible. Hence we neglect it.
Functions	Dispatcher is also responsible for: Context Switching, Switch to user mode, Jumping to proper location when process again restarted	The only work of scheduler is selection of processes.

5.2 调度准则

- CPU 使用率

  应该使 CPU 尽可能忙碌
- 吞吐量

  一个时间单元内进程完成的数量
- 周转时间

从进程提交到完成的时间段称为周转时间 (Turnaround Time)

- 等待时间

  在就绪队列中等待所花时间之和

- 响应时间

  从提交请求到产生第一响应的时间

- Number of Context Switches (from 课件)

  尽可能少做上下文切换

5.3 调度算法 Scheduling Algorithm

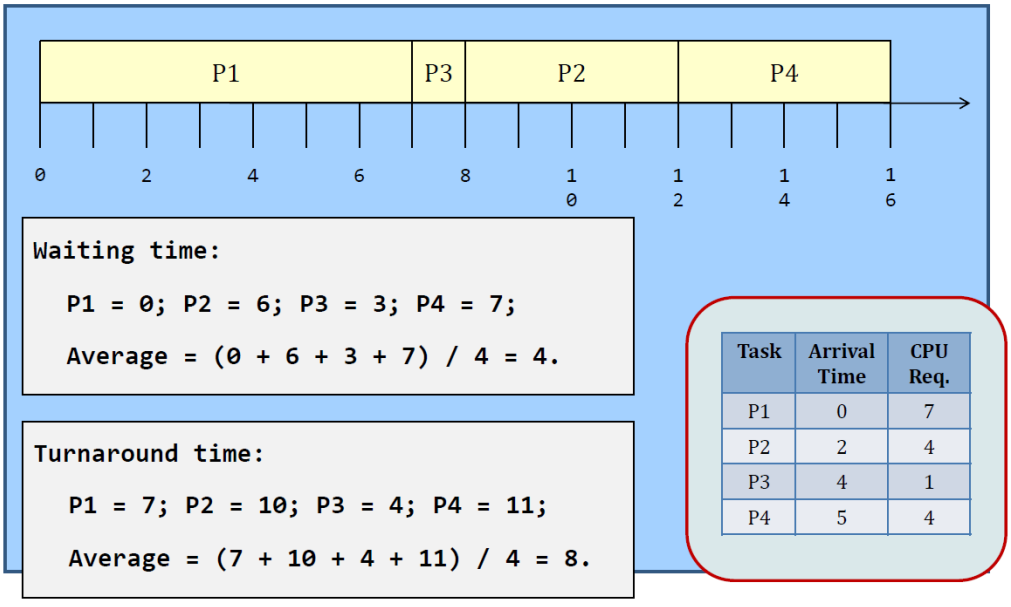
5.3.1 先到先服务调度 First-Come-First-Served (FCFS)

字面意思

5.3.2 最短作业优先调度 Shortest-Job-First (SJF)

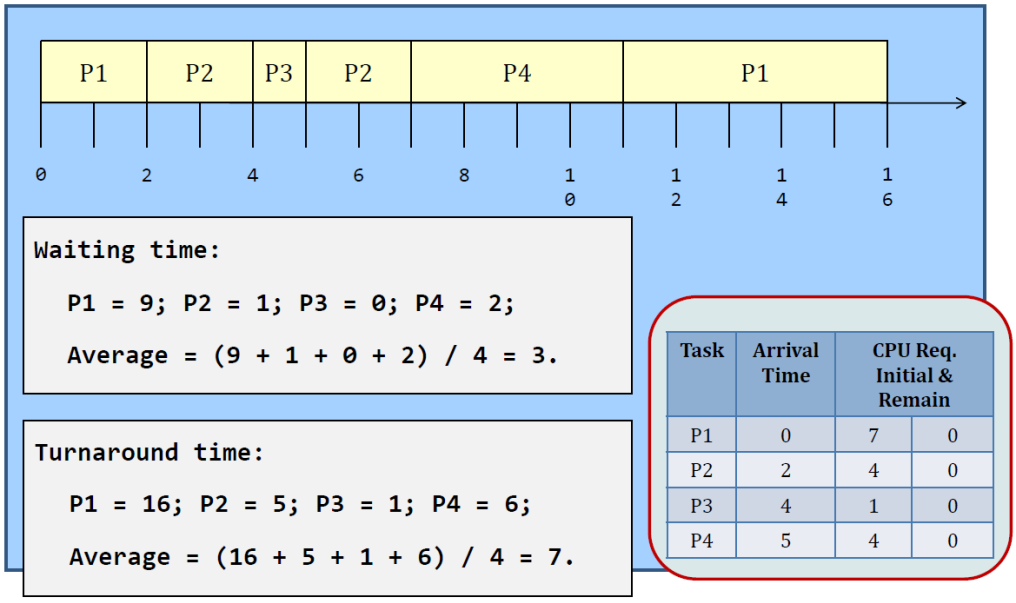
- 选择最短 CPU 执行时间的进程
- 相同，可以使用 FCFS 规则选择
- 又称最短下次 CPU执行 (Shortest-Next-CPU-Burst) 算法

5.3.2.1 非抢占 (Non-Preemptive) SJF



5.3.2.2 抢占 SJF

- 最短剩余时间优先 (Shortest-Remaining-Time-First)

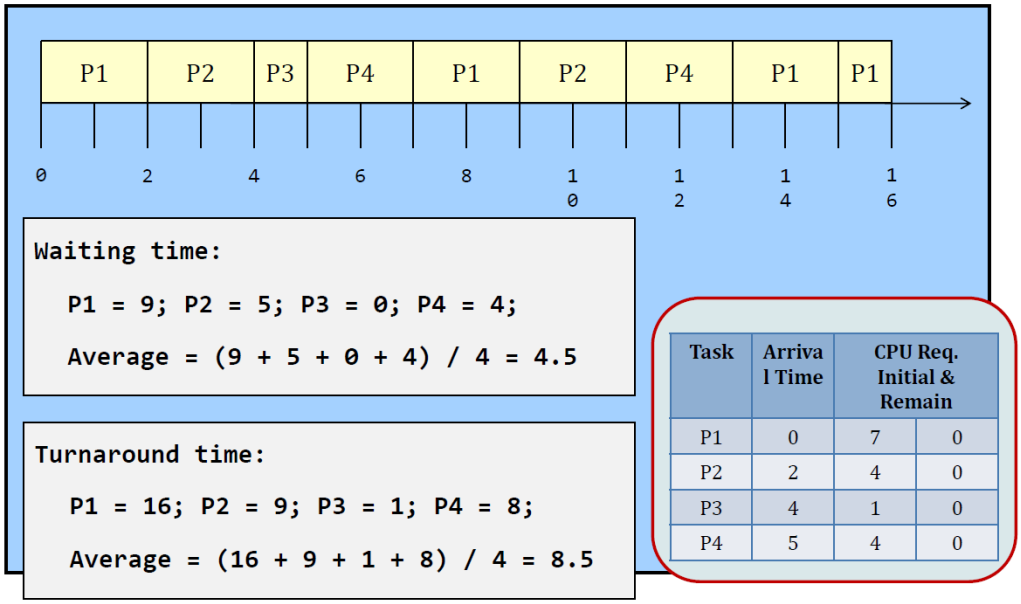


Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	0
P2	2	4	0
P3	4	1	0
P4	5	4	0

- 缺点：上下文切换多

5.3.3 轮转调度 Round Robin (RR)

- 每个进程都有一个时间量 (Time Quantum) 或时间片 (Time Slice)
  通常 10~100ms
- 当时间片用完时，该进程就会释放 CPU (相当于抢占)
- 调度程序选择下一个时间片 > 0 的进程
- 如果所有进程都用完了时间片，它们的时间片同时被 recharge 到初始值
- 就绪队列为循环队列，进程被依次执行



Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	0
P2	2	4	0
P3	4	1	0
P4	5	4	0

- 缺点：性能较差
- 优点：公平

5.3.4 优先级调度 Priority Scheduling

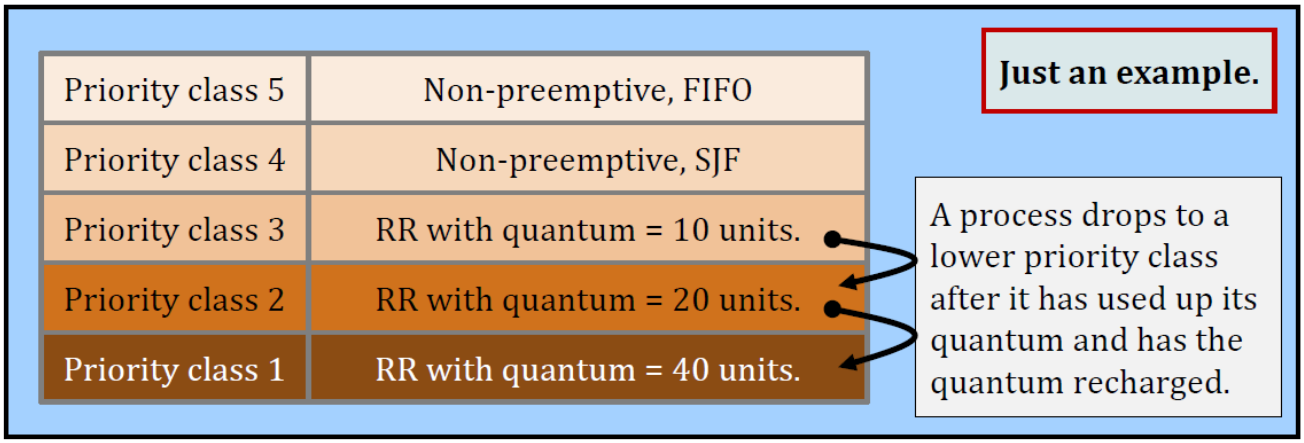
- 每个进程都有一个优先级
- 调度程序根据优先级选择进程
- 优先队列
- 分类

2 Classes	
Static priority	**Dynamic priority**
Every task is given a fixed priority.	Every task is given an initial priority.
The priority is **fixed** throughout the life of the task.	The priority is **changing** throughout the life of the task.

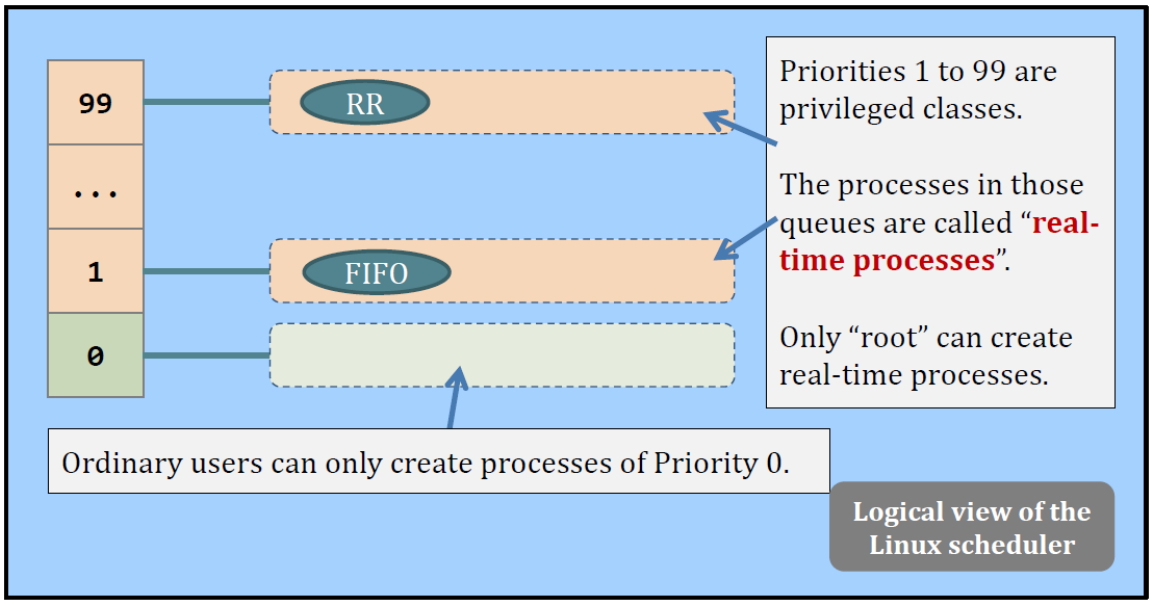
- 新进程到来时，重新 schedule (这里会发生抢占)
- 如果当前进程被抢占，它先出队再入队

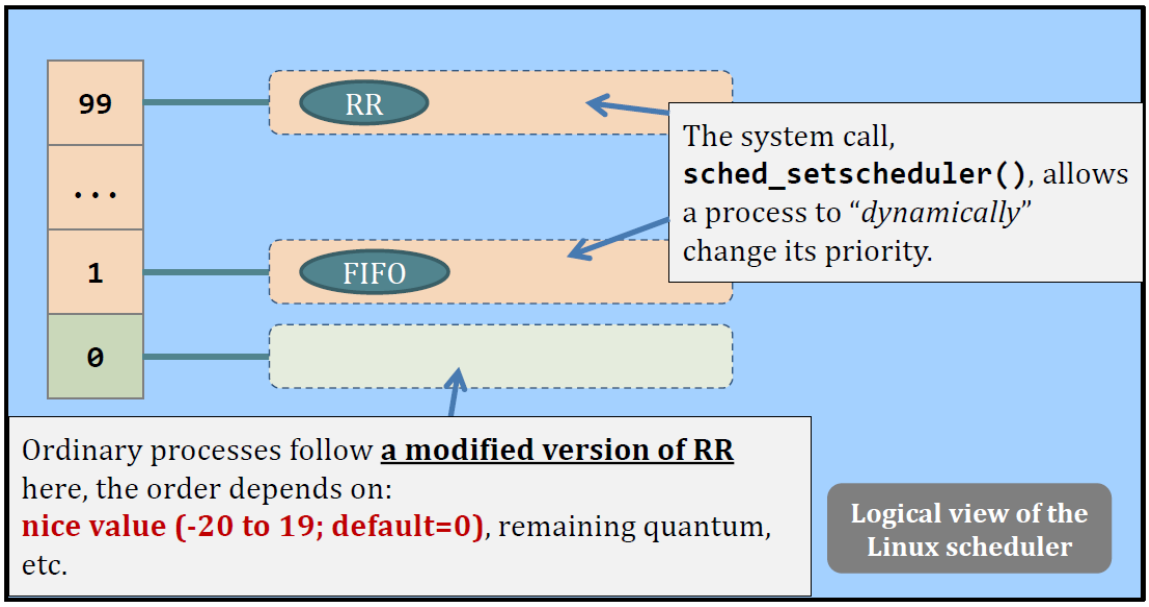
5.3.4.1 Multiple Queue Priority Scheduling

- 依然是 priority scheduler
- 每个优先级有不同的调度方式
- 可以是静态优先级和动态优先级混合



- Linux Scheduler


99
...
1
0
RR
FIFO
Priorities 1 to 99 are privileged classes.
The processes in those queues are called "real-time processes".
Only "root" can create real-time processes.
Ordinary users can only create processes of Priority 0.
Logical view of the Linux scheduler


99
...
1
0
RR
FIFO
The system call, sched_setscheduler(), allows a process to "dynamically" change its priority.
Ordinary processes follow a modified version of RR here, the order depends on:
nice value (-20 to 19; default=0), remaining quantum, etc.
Logical view of the Linux scheduler