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

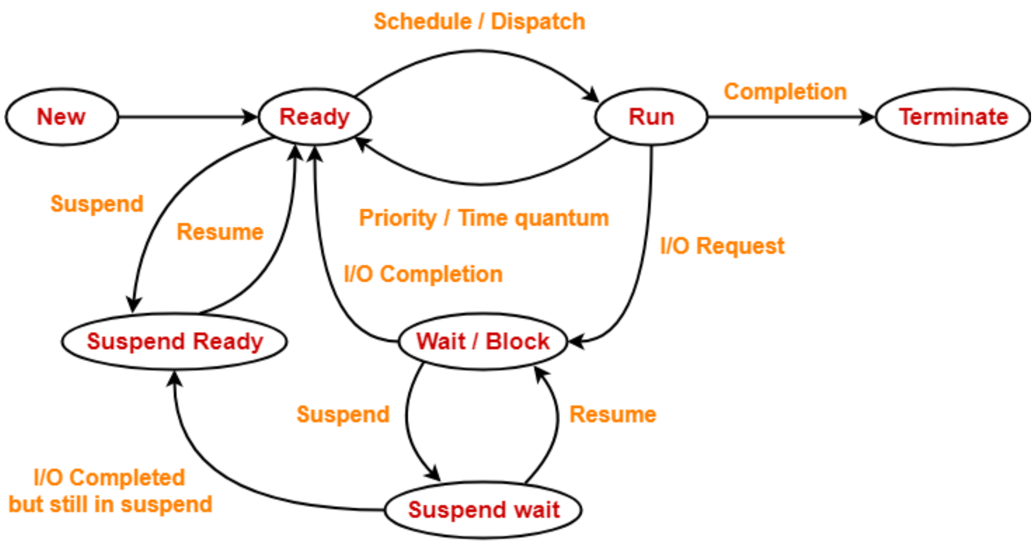
- **进程 (Process)** 是执行的程序

  Process is a program in execution

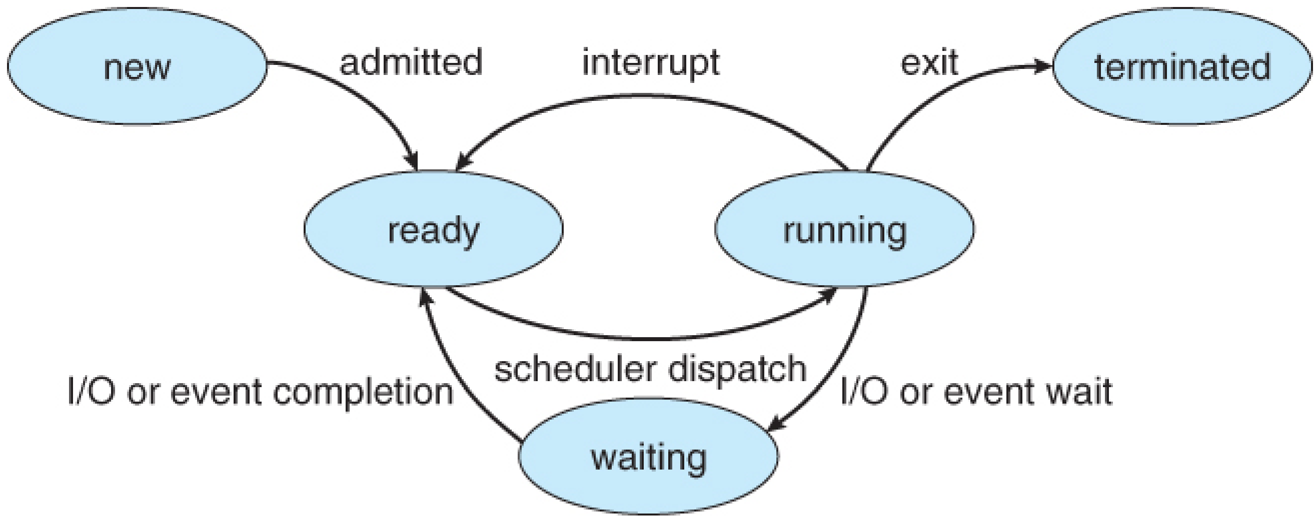
- 进程还包括:
    - 程序计数器 PC
    - 寄存器 (Register) 的内容
    - 堆 Heap
    - 栈 Stack
    - 数据段 Data Section
    - 文本段 Text Section
    - ...
- 程序 (Program) 和进程
    - 程序是**被动实体 (passive entity)**，如存储在磁盘上包含一系列指令的文件，经常称为可执行文件 (executable file)
    - 进程是**活动实体 (active entity)** 或称**主动实体**，具有一个程序计数器用来表示下个执行命令和一组相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

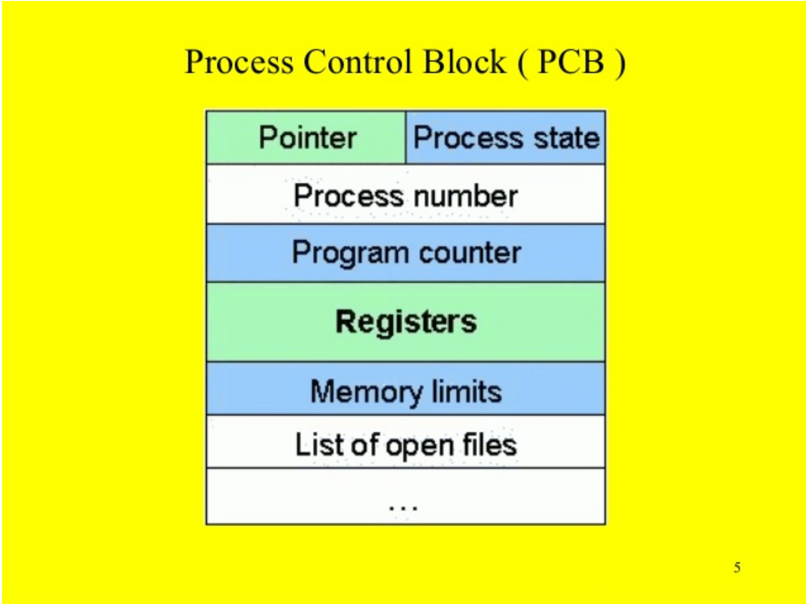


- 等待状态又分为
  - 可中断 (Interruptable)
  - 不可中断 (Un-interruptible)
- 刚 fork 的进程都会变成 ready 状态

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.

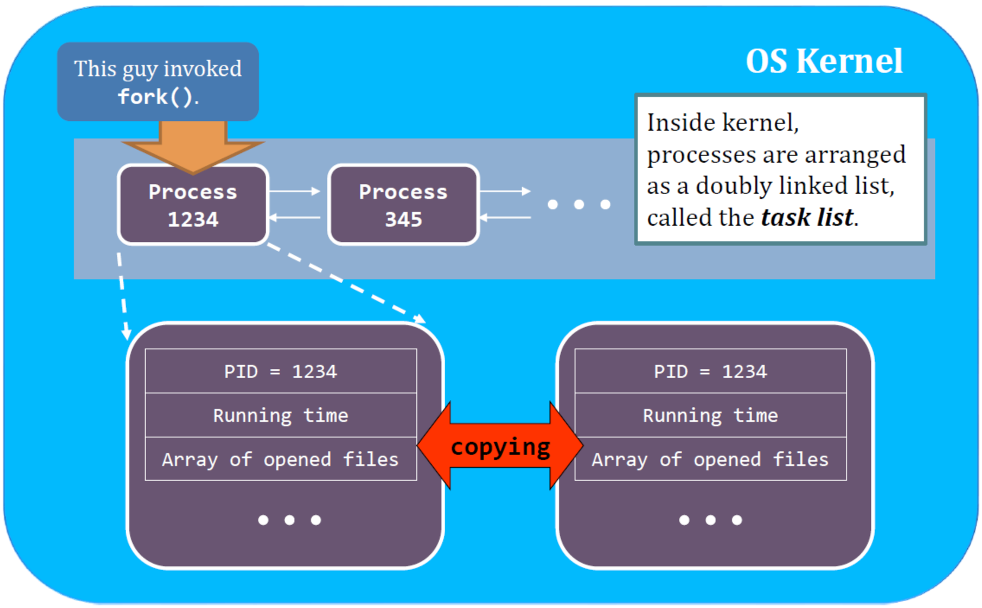
- 在代码中如何区分父进程和子进程

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

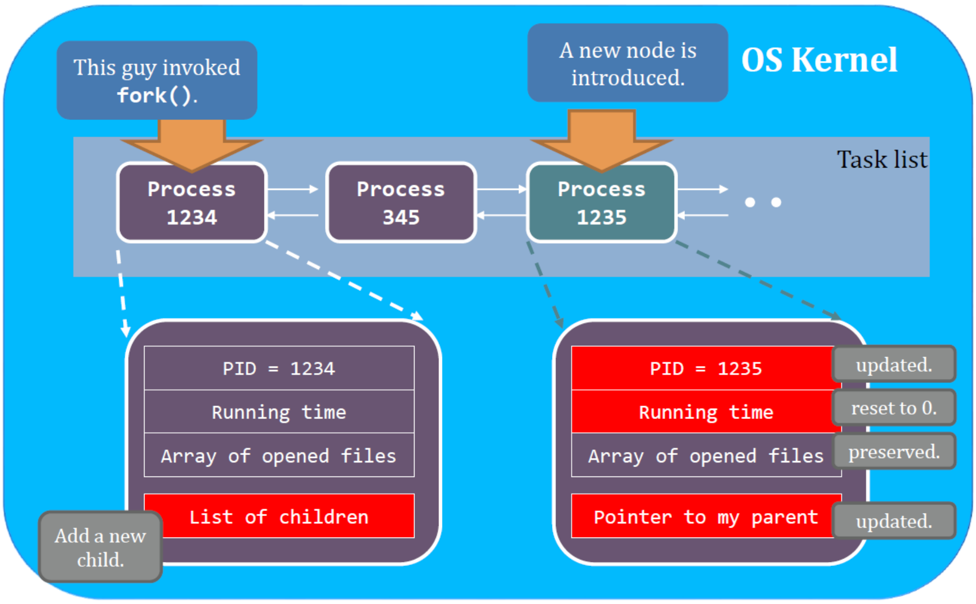
```
if (!pid) {
    // 只有子进程执行
}
else {
    // 只有父进程执行
}
```

- 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

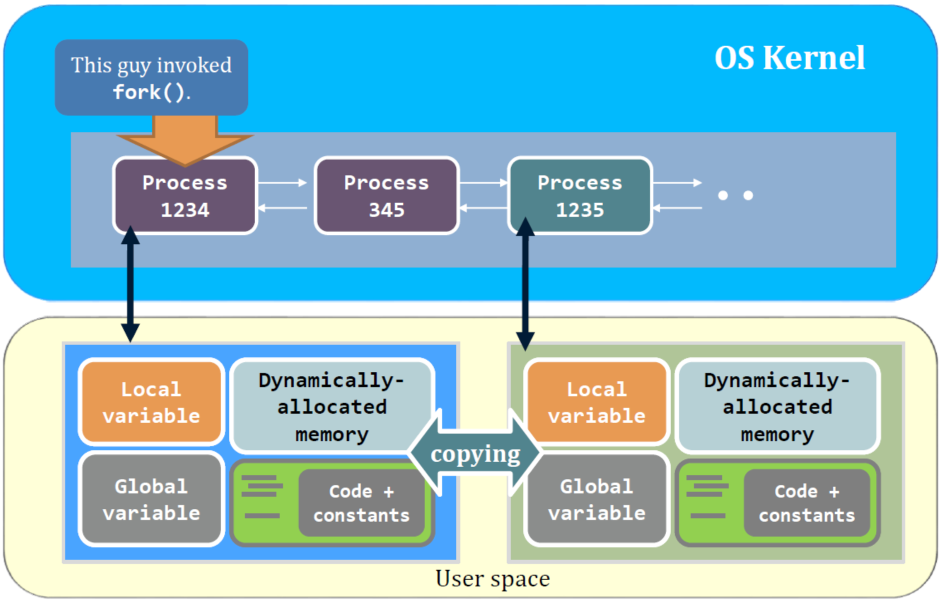
  1. 复制 PCB



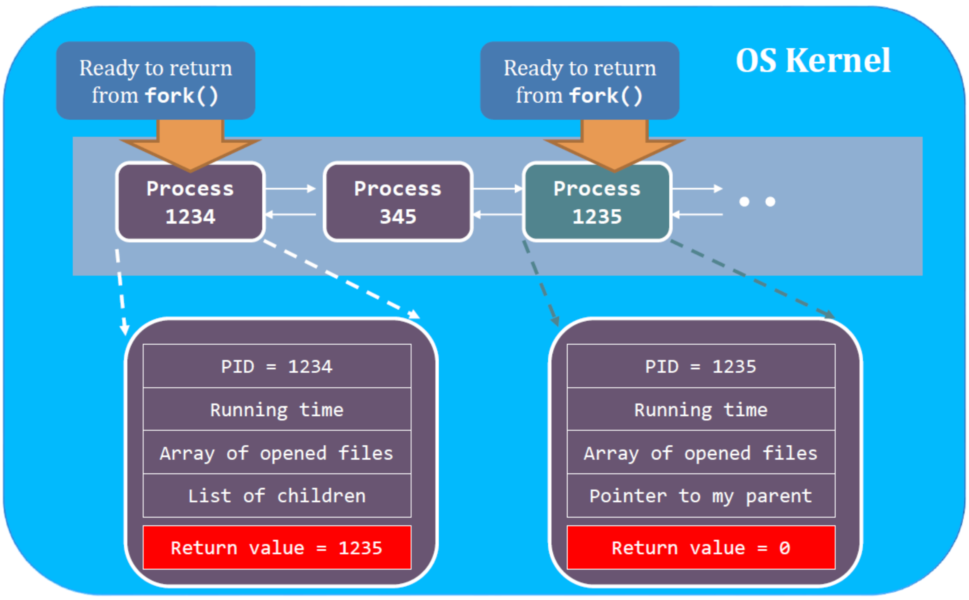
  2. 更新 PCB 和 task list



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);
```

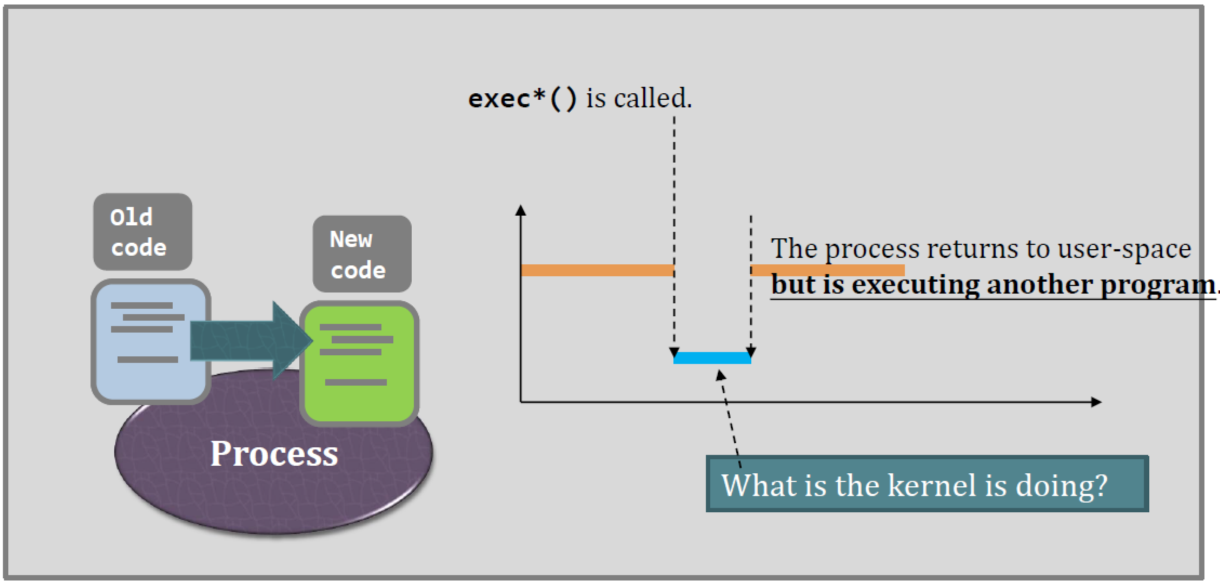
Argument Order	Value in above example	Description
1	"/bin/ls"	The file that the programmer wants to execute.
2	"/bin/ls"	When the process switches to "/bin/ls", this string is the **program argument[0]**.
3	"-l"	When the process switches to "/bin/ls", this string is the **program argument[1]**.
4	NULL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

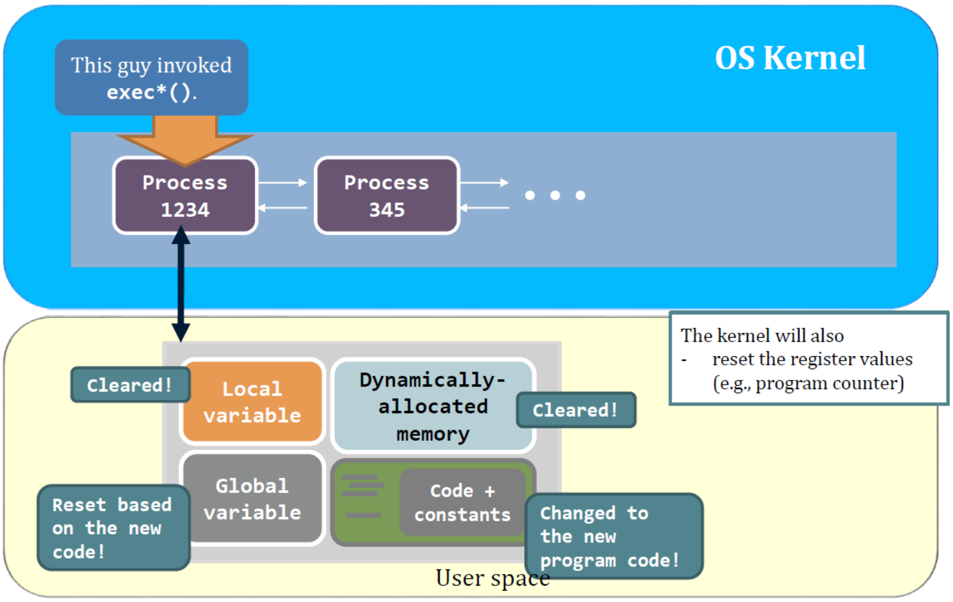
  `args[0]` 是程序的名字

- 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

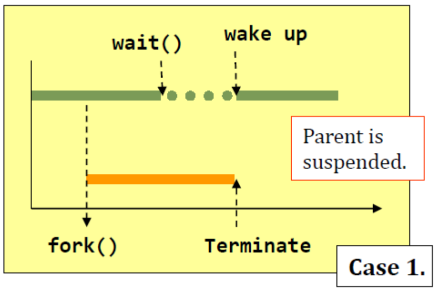


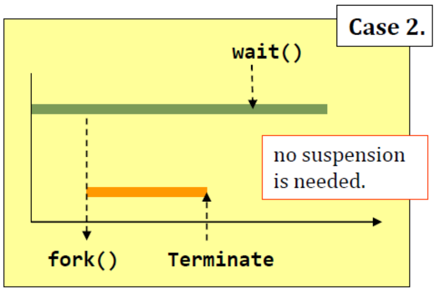


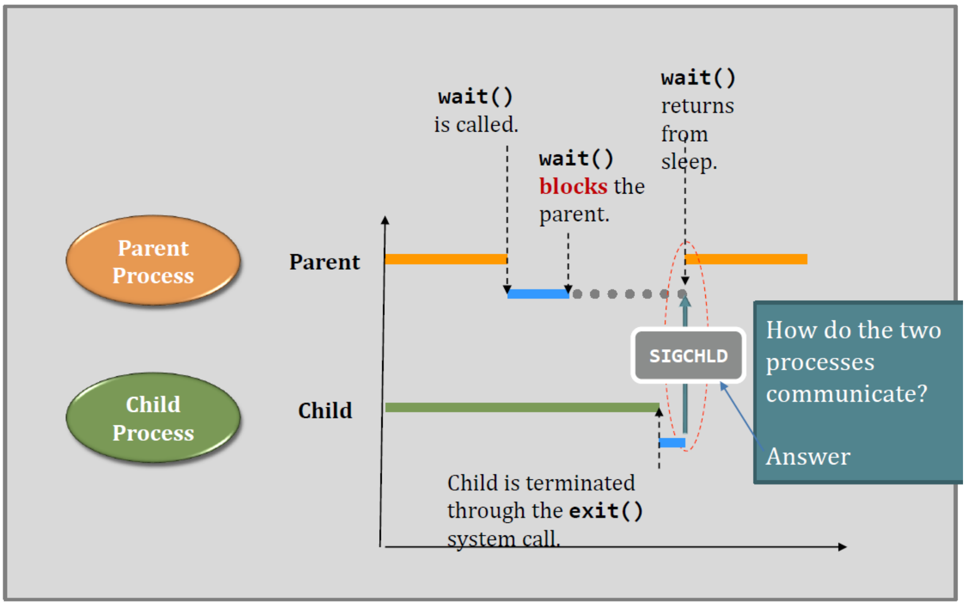
3.2.4 进程等待

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

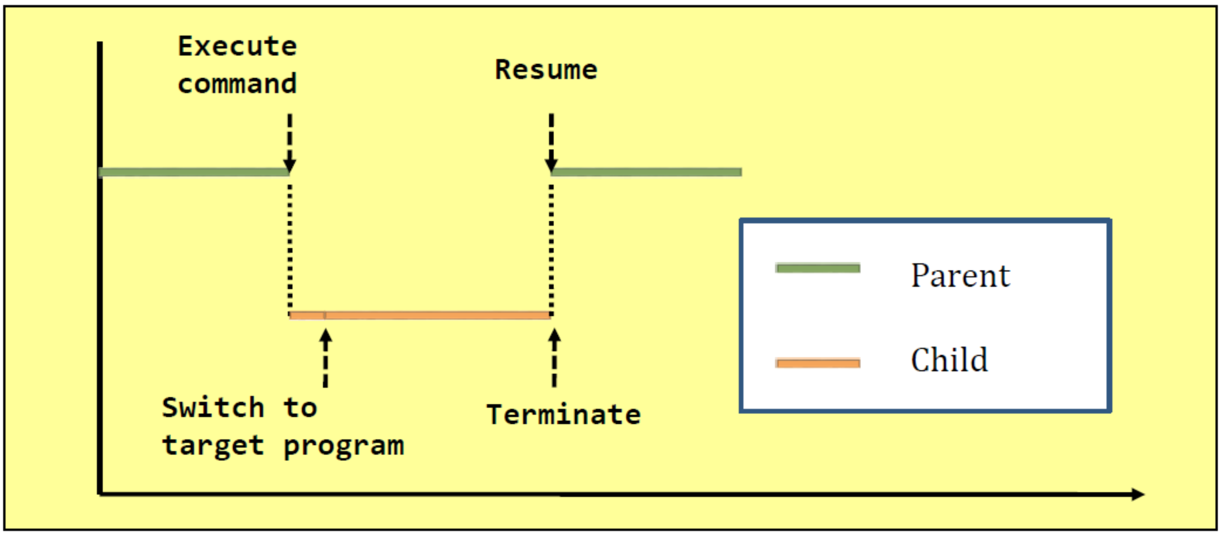
wait()	vs	waitpid()
Wait for any one of the children.		Depending on the parameters, **waitpid()** will wait for a particular child only.
Detect child **termination** only.		Depending on the parameters, **waitpid()** <u>**can detect multiple child's status change**</u>







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

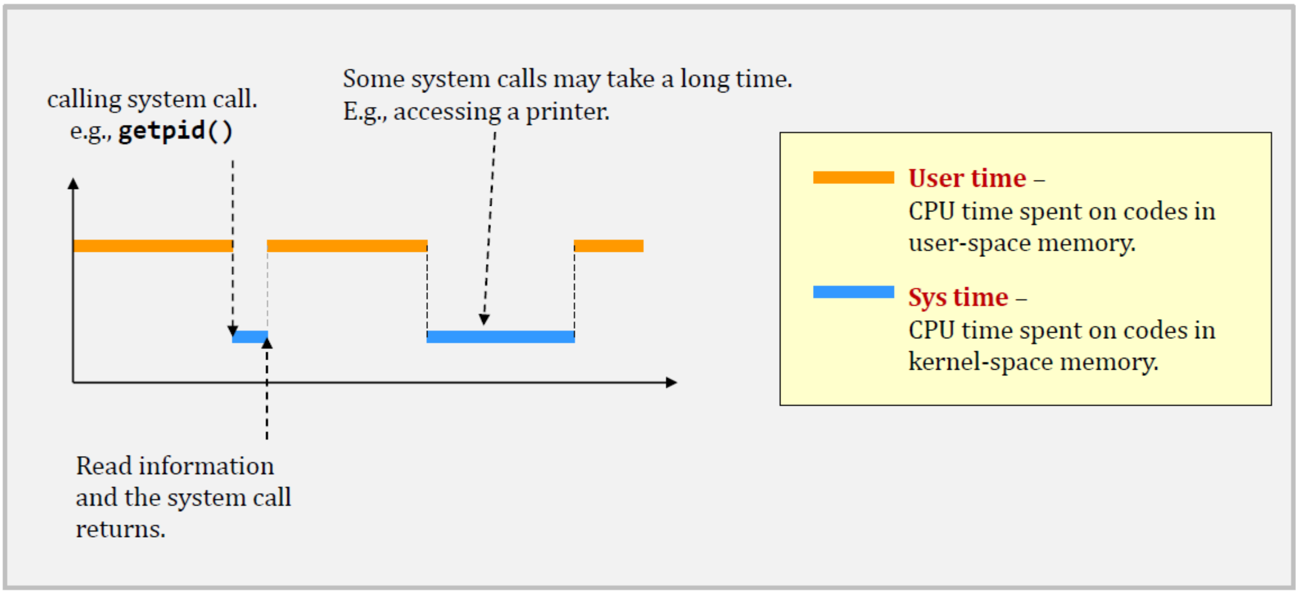
3.2.5 进程时间

- 实际时间 Real Time

  Wall-clock time
- 用户时间 User Time

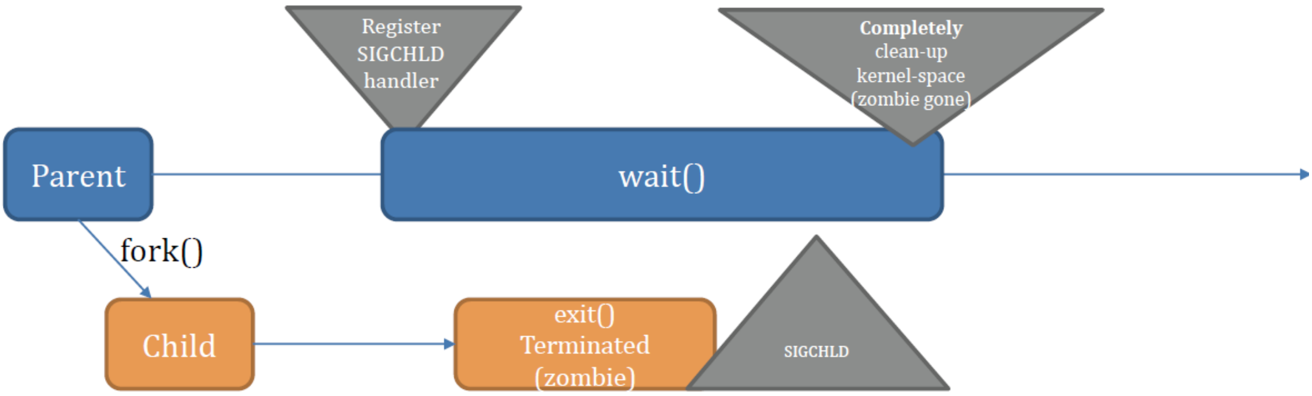
  CPU 在用户空间花费的时间
- 系统时间 System Time

  CPU 在内核空间花费的时间

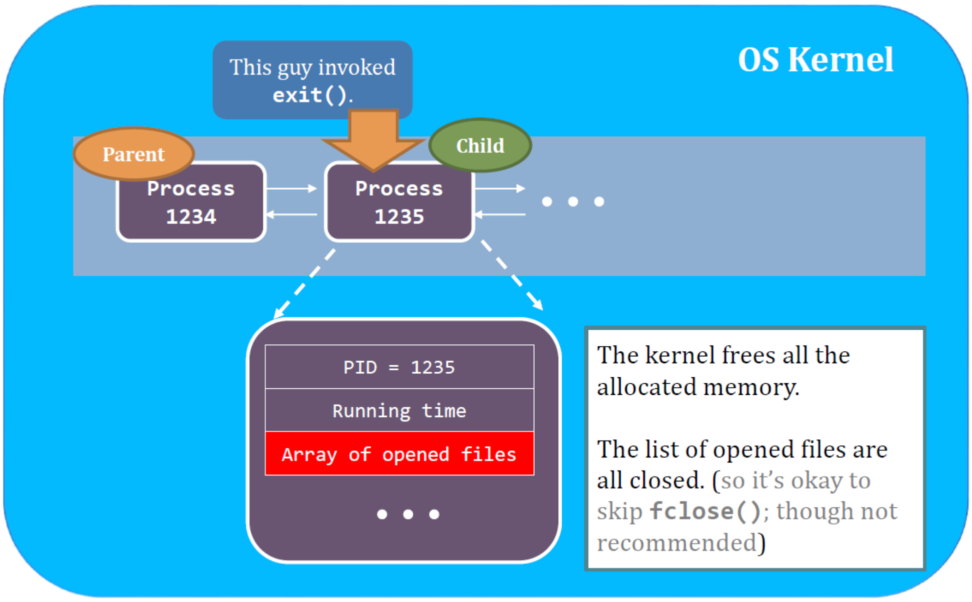


- User Time + Sys Time 决定了程序的性能 (Performance)
  - User Time + Sys Time > Real Time: 单核
  - User Time + Sys Time < Real Time: 多核

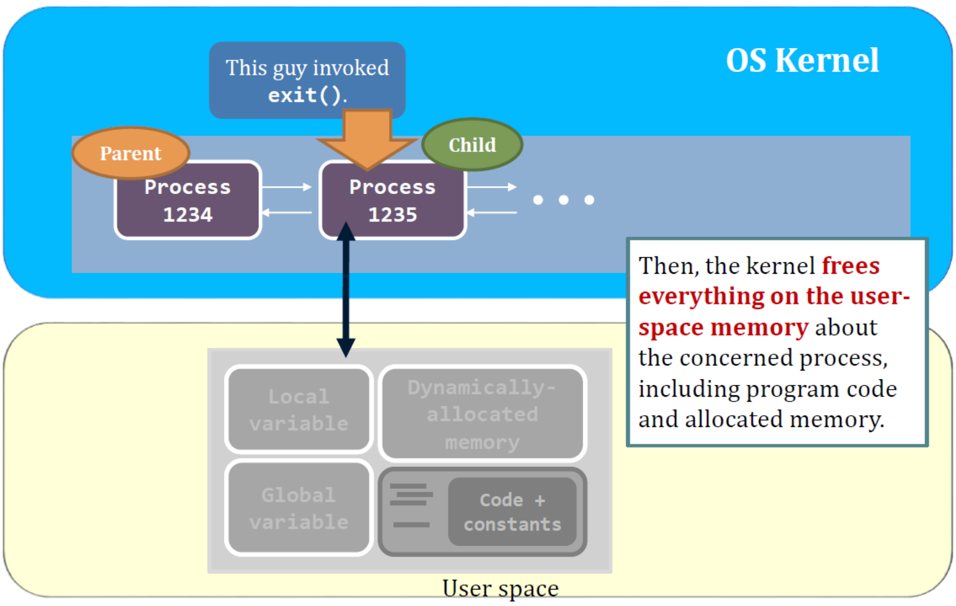
3.2.6 进程终止 Process Termination



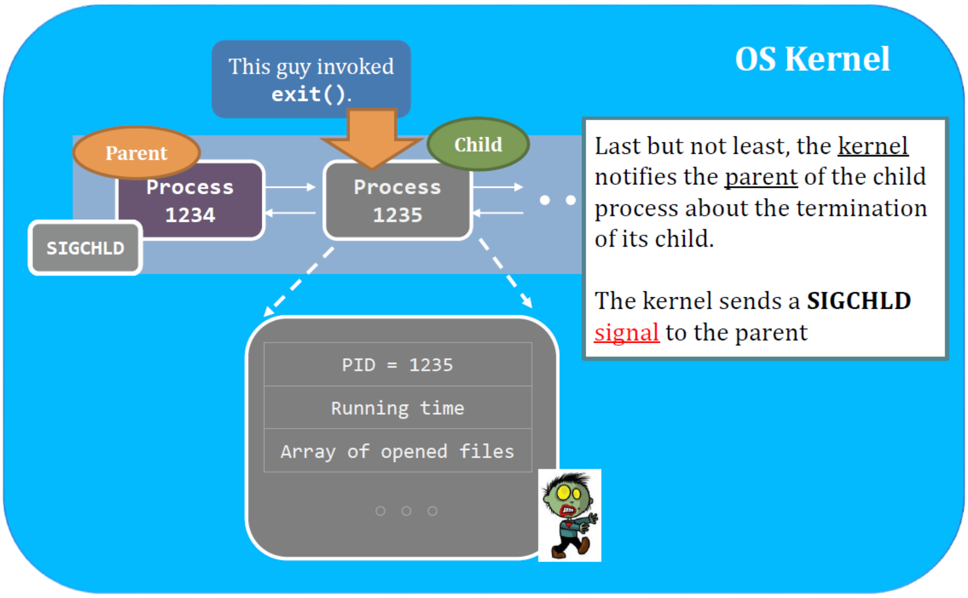
- System call `exit()`: terminate the calling process
- `exit()` 的执行过程
  1. Clean up most of the allocated kernel space memory



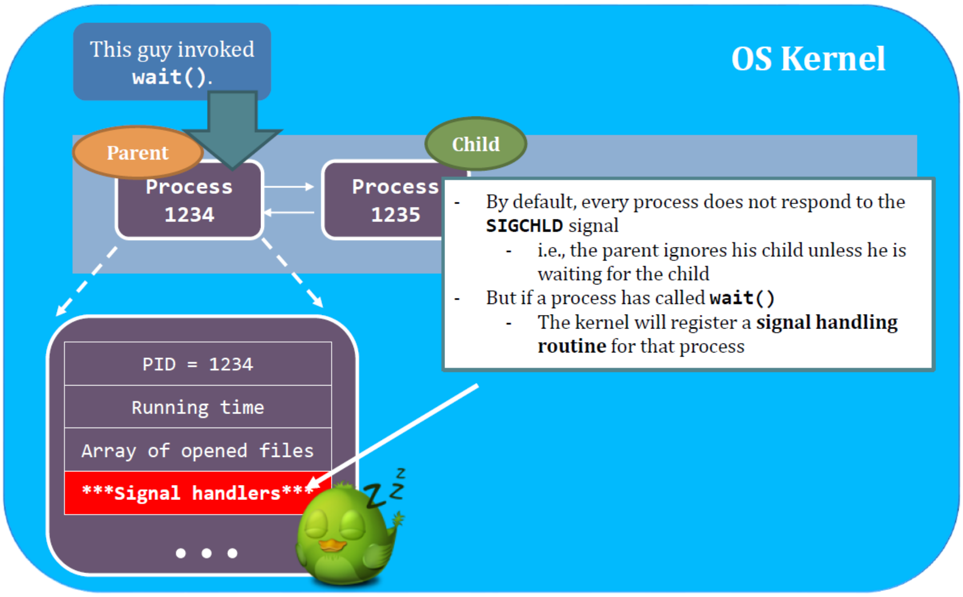
  2. Clean up the exit process's user space memory

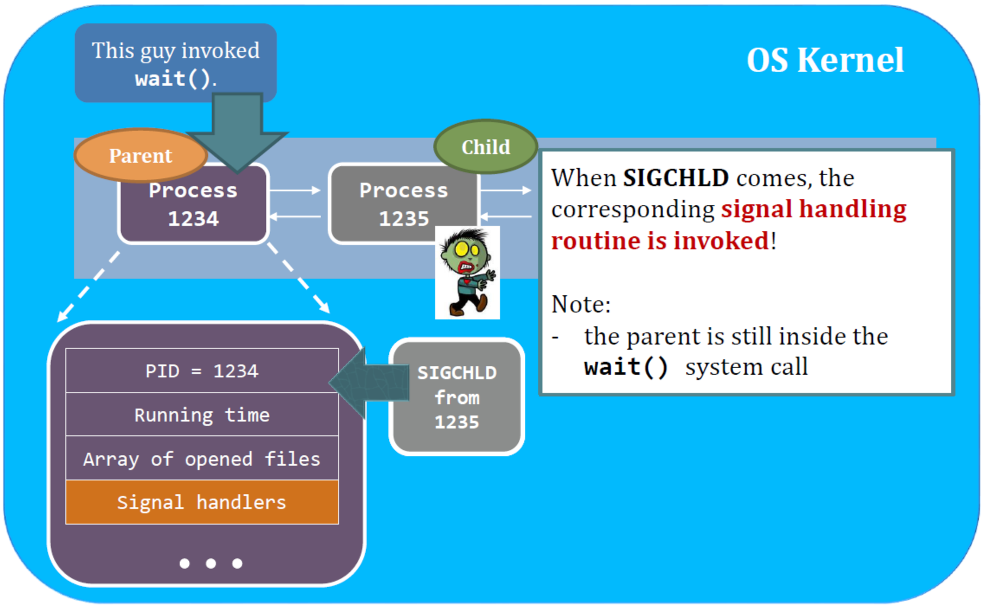


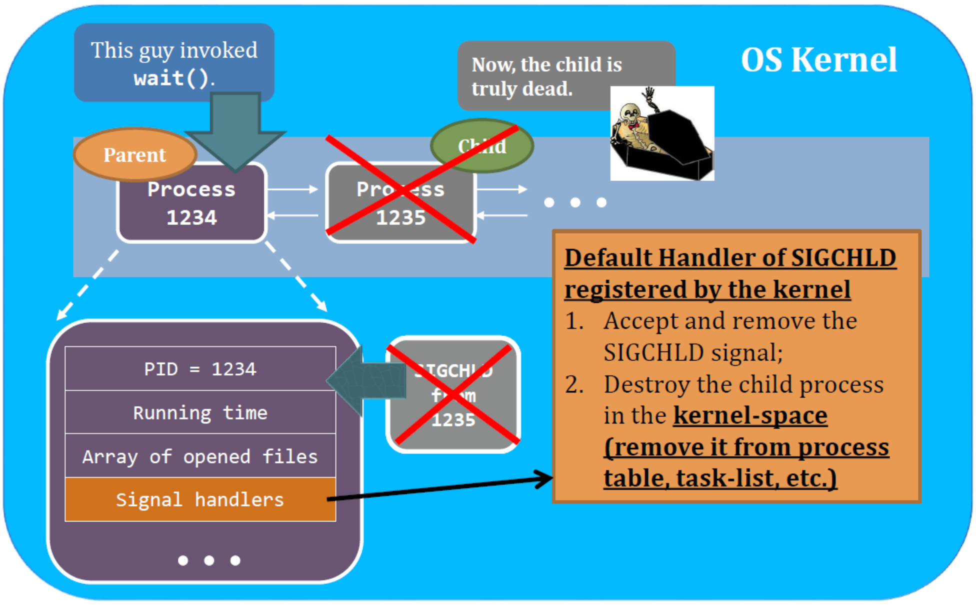
  3. Notify the parent with SIGCHLD.

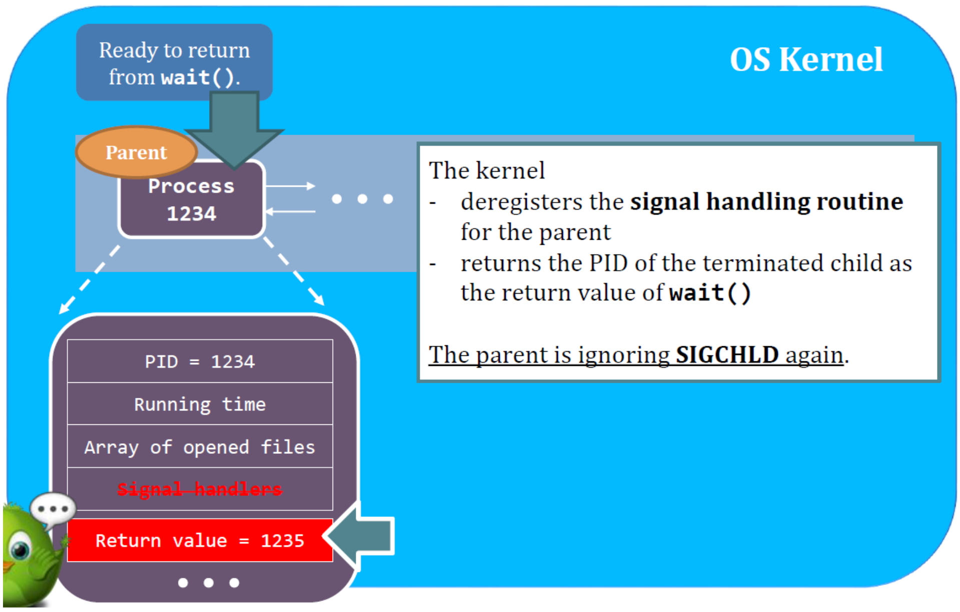


- 僵尸进程 Zombie Process
    - 进程的用户空间和内核空间被释放之后，PID 依然在进程表里，直到父进程调用 `wait()`
    - 当进程已经终止，但是其父进程尚未调用 `wait()`，这样的进程称为僵尸进程
    - 所有进程终止时都会过渡到这个状态
    - 一旦父进程调用 `wait()`，僵尸进程的进程标识符和它在进程表中的条目就会释放









- 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()`

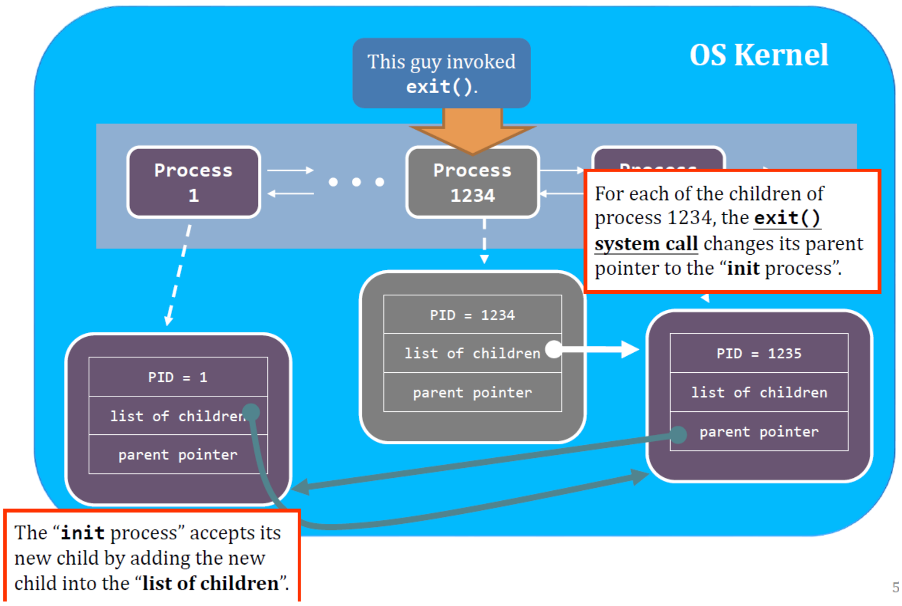
- The fork bomb
  - PID 是有限的，Linux 中 PID 最大值为 32768

    `cat / proc /sys/ pid_max`

  - fork bomb (僵尸大军)

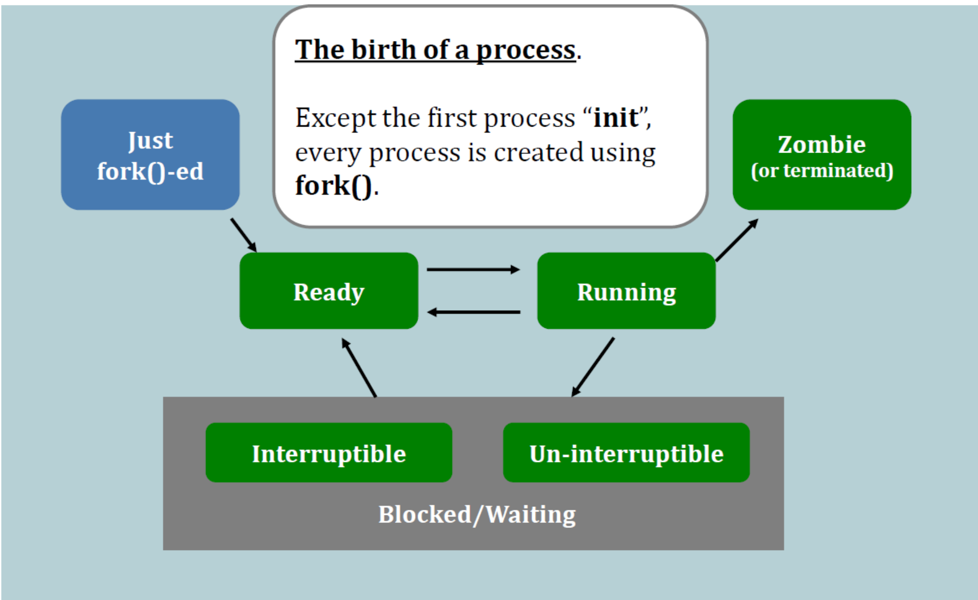
    ```c
    int main() {
        while (fork());
        return 0;
    }
    ```

- 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

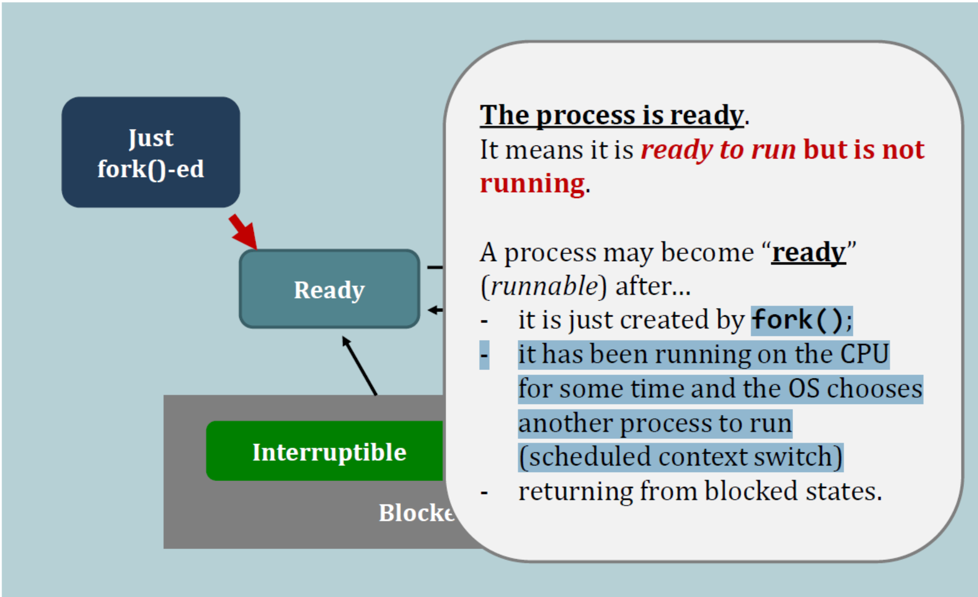


3.2.7 进程生命周期 Process Lifecycle

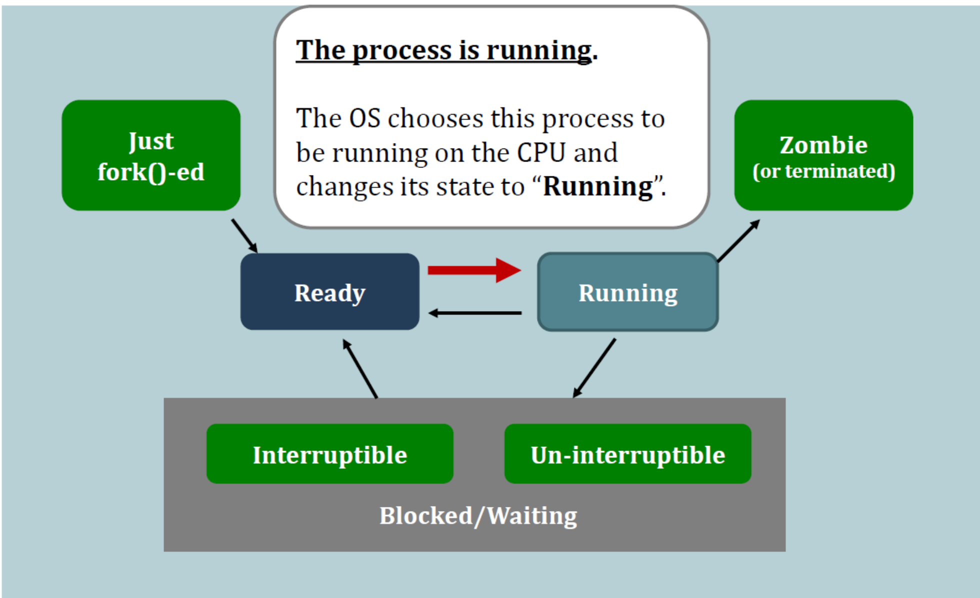
1. forked



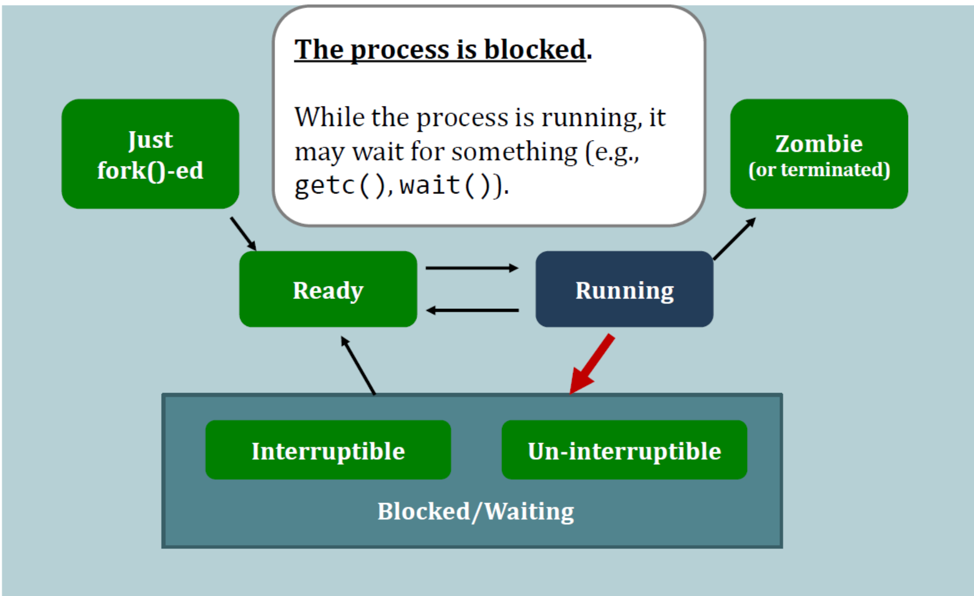
2. Ready



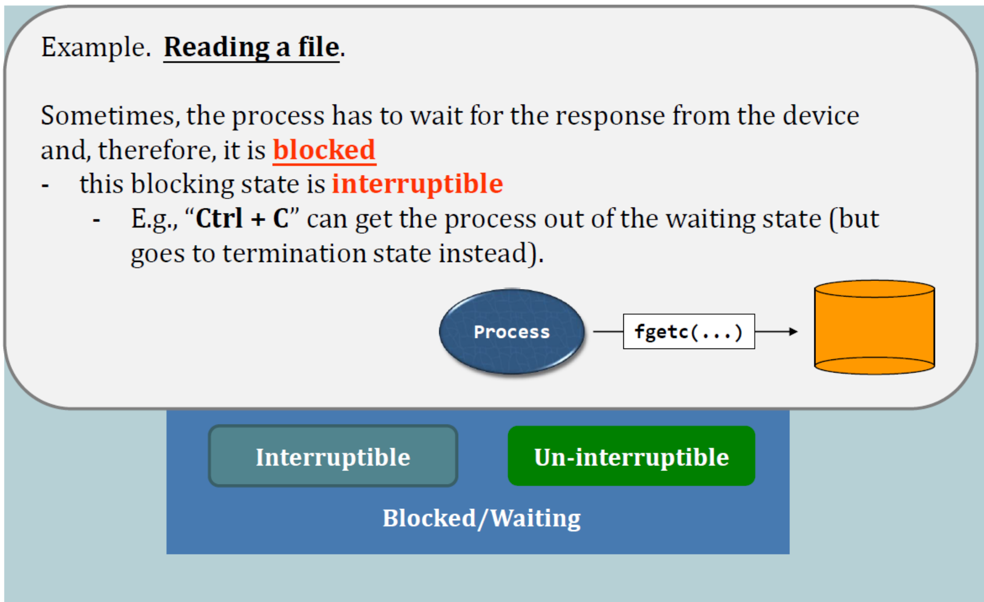
3. Running



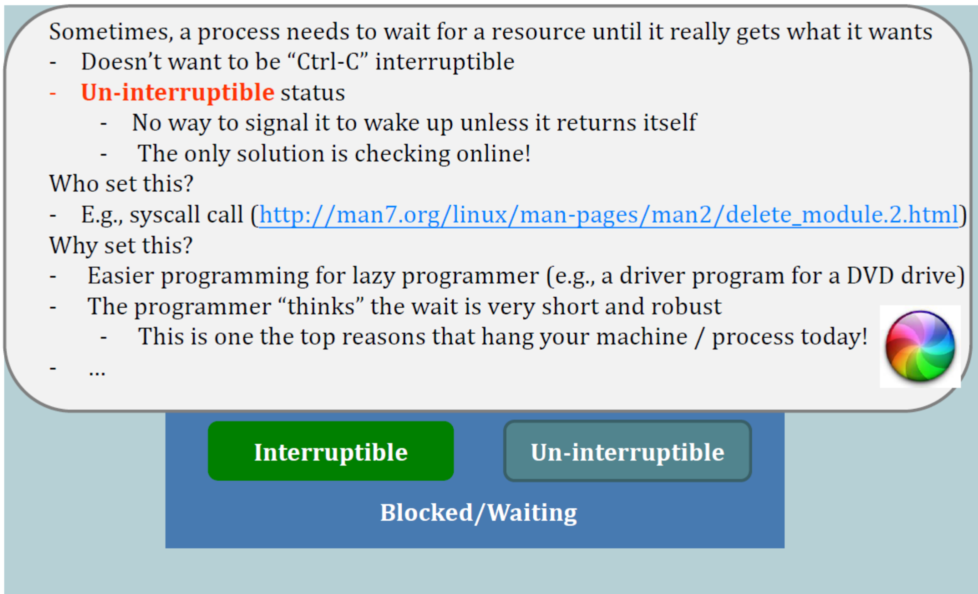
4. Blocked



5. Interruptable waiting

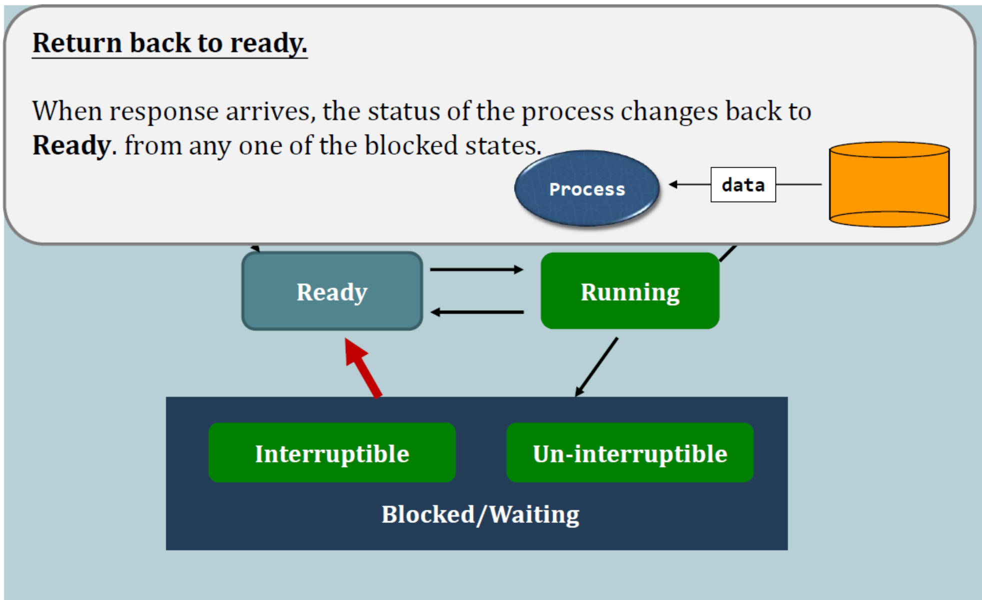


6. Un-interruptable waiting

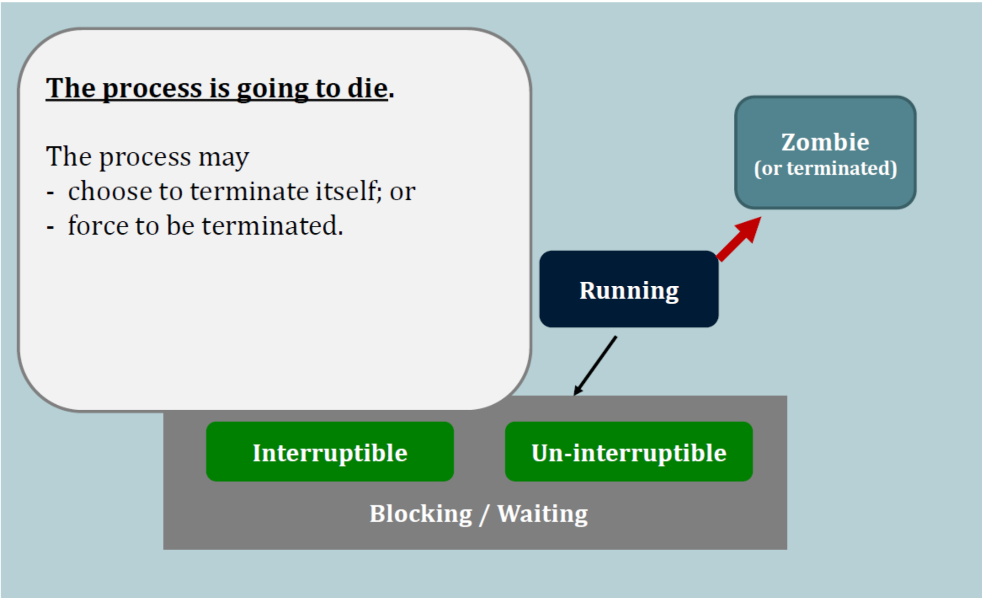


计网的程序里经常碰见，纯贵物，谁设计的抓紧埋了吧

7. Return back to ready



8. Terminated

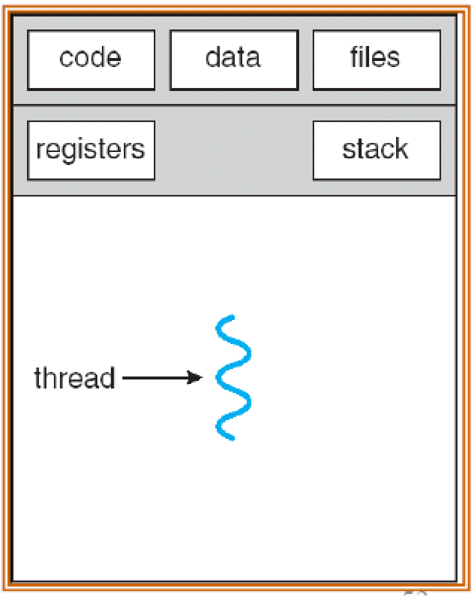


第四章 线程 Thread

4.1 线程的概念

- Heavyweight Process

  A process has a single thread of control



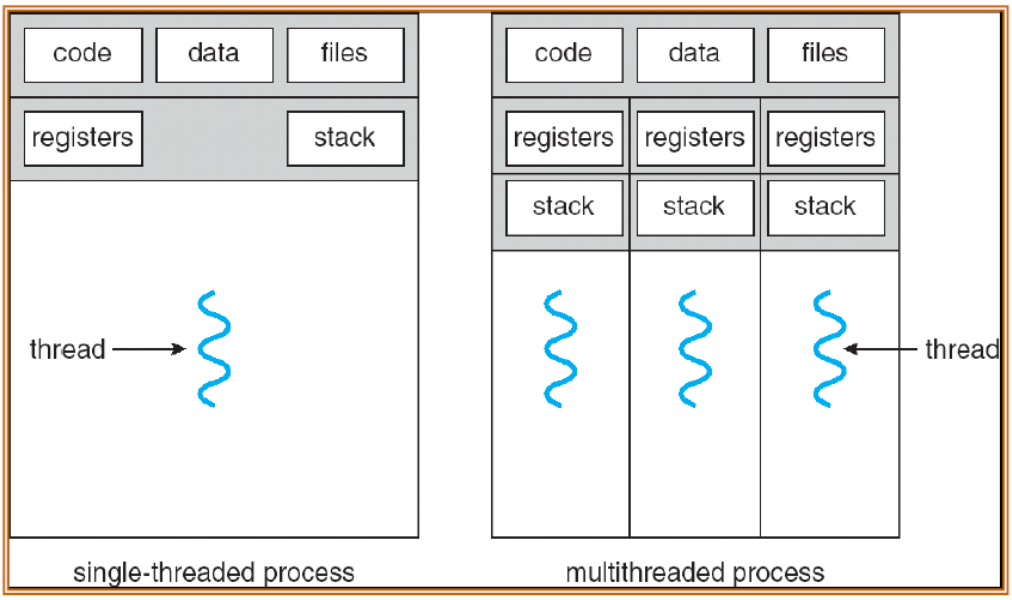
- 线程 Thread

  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

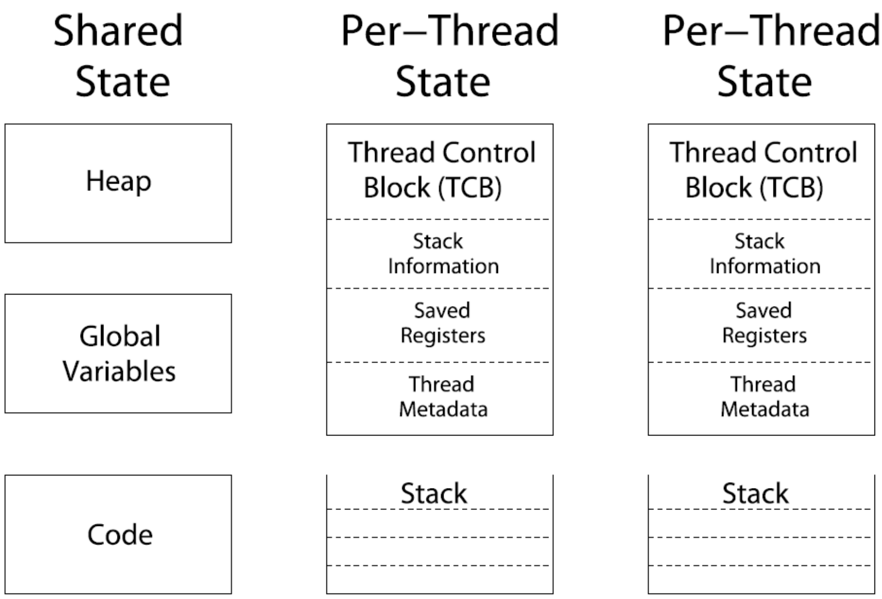
  A single program made up of a number of different concurrent (并发) activities
- 结构图



- 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

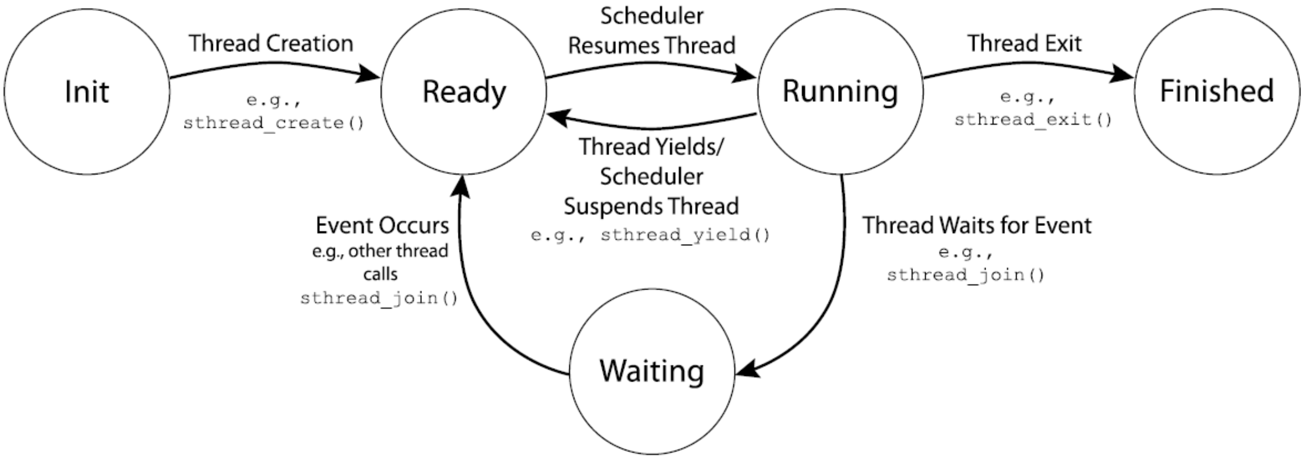


4.3 线程和进程的区别

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.

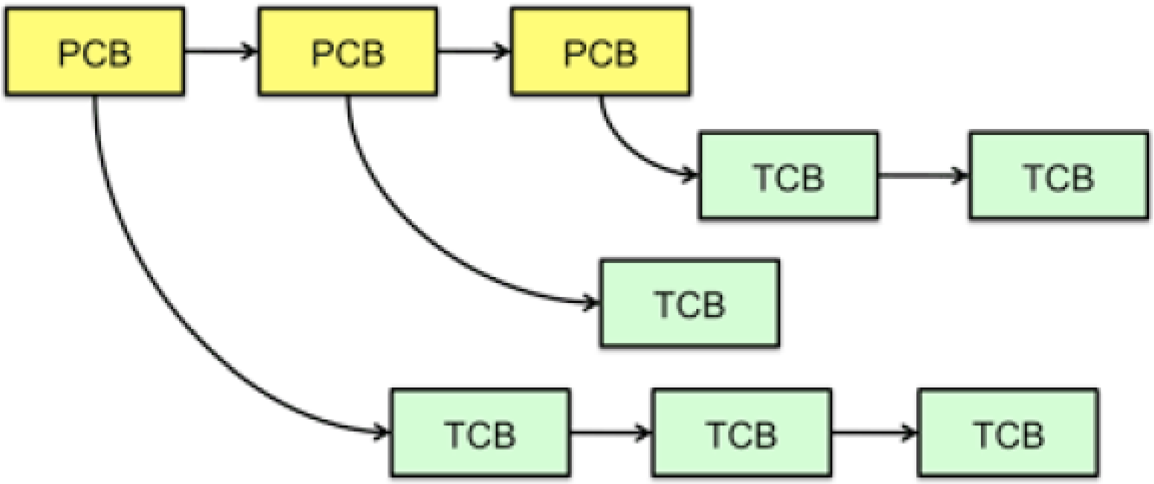
Process	Thread
Process is less efficient in term of communication.	Thread is more efficient in term of communication.
Process consume more resources.	Thread consume less resources.
Process is isolated.	Threads share memory.
Process is called heavy weight process.	Thread is called light weight process.
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

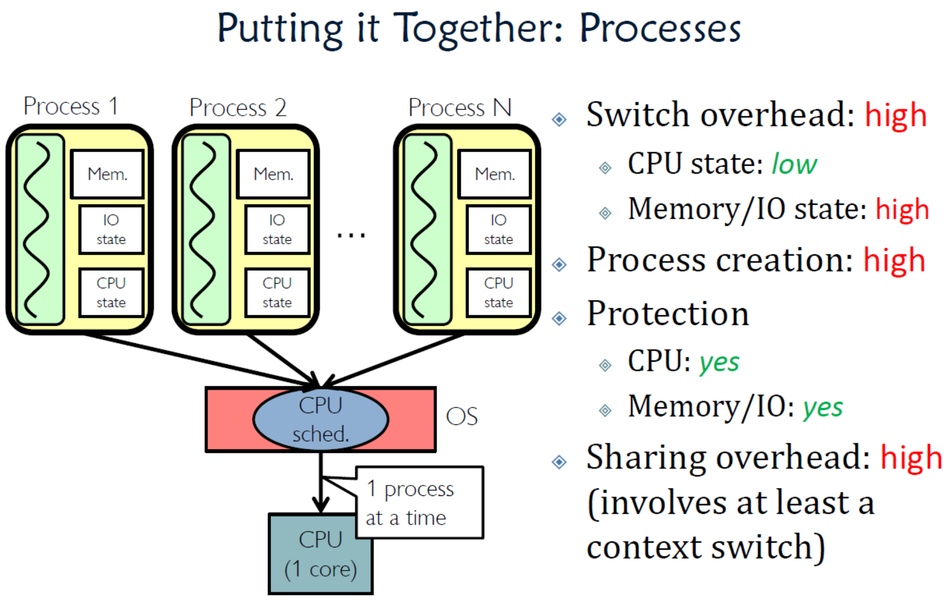
- PCB 指向多个 TCB

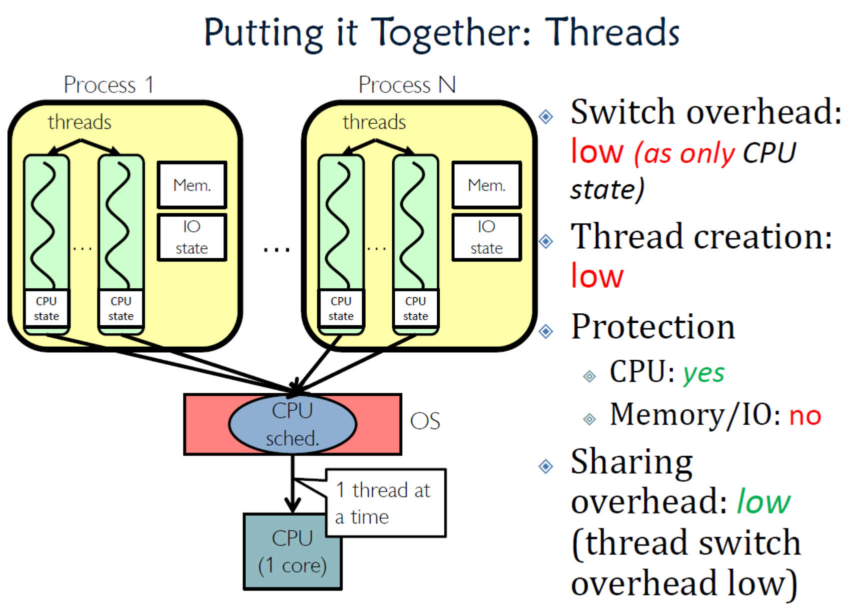


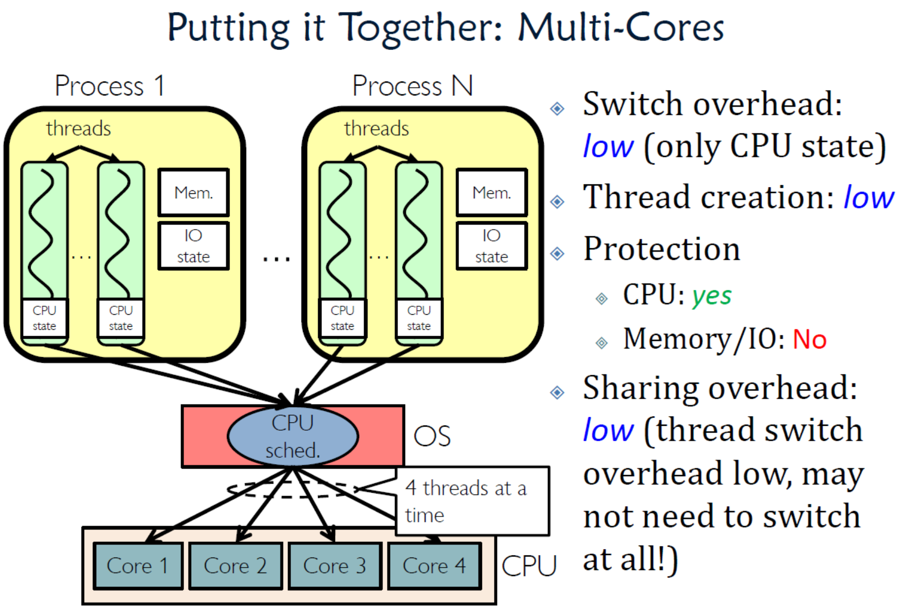
- Switching threads within a block is a simple thread switch
- Switching threads across blocks requires changes to memory and I/O address tables

---

4.6 多线程调度

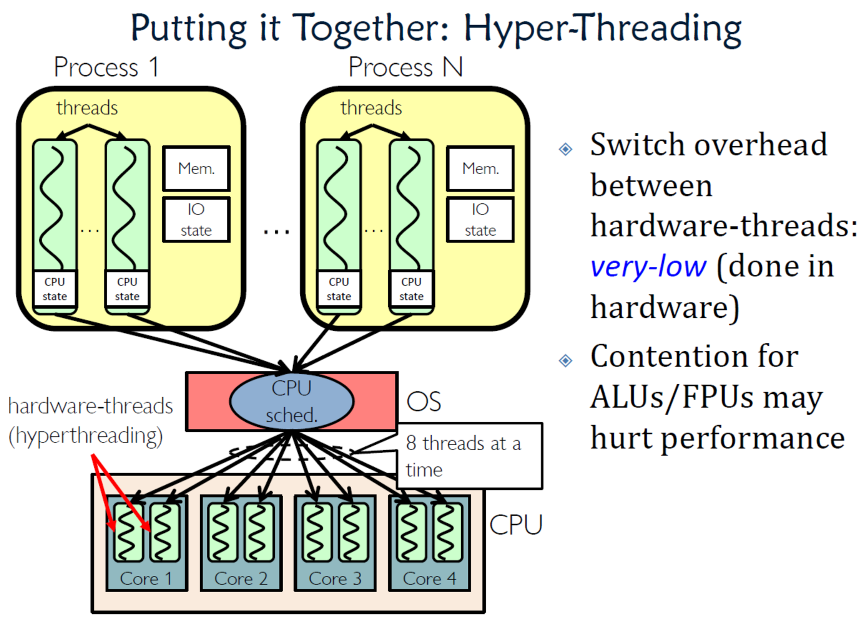






- 超线程 Hyper-Threading

  超线程(hyper-threading)其实就是**同时多线程(simultaneous multi-theading)**，是一项允许一个CPU执行多个控制流的技术。它的原理很简单，就是把一颗CPU当成两颗来用，将一颗具有超线程功能的物理CPU变成两颗逻辑CPU，而逻辑CPU对操作系统来说，跟物理CPU并没有什么区别。因此，操作系统会把工作线程分派给这两颗（逻辑）CPU上去执行，让（多个或单个）应用程序的多个线程，能够同时在同一颗CPU上被执行。注意：两颗逻辑CPU共享单颗物理CPU的所有执行资源。因此，我们可以认为，**超线程技术就是对CPU的虚拟化**



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

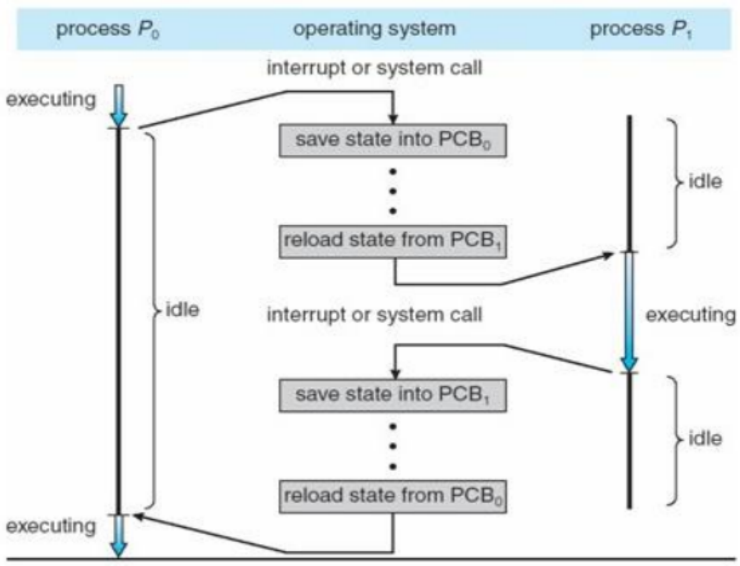
  A computer running more than one program at a time

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

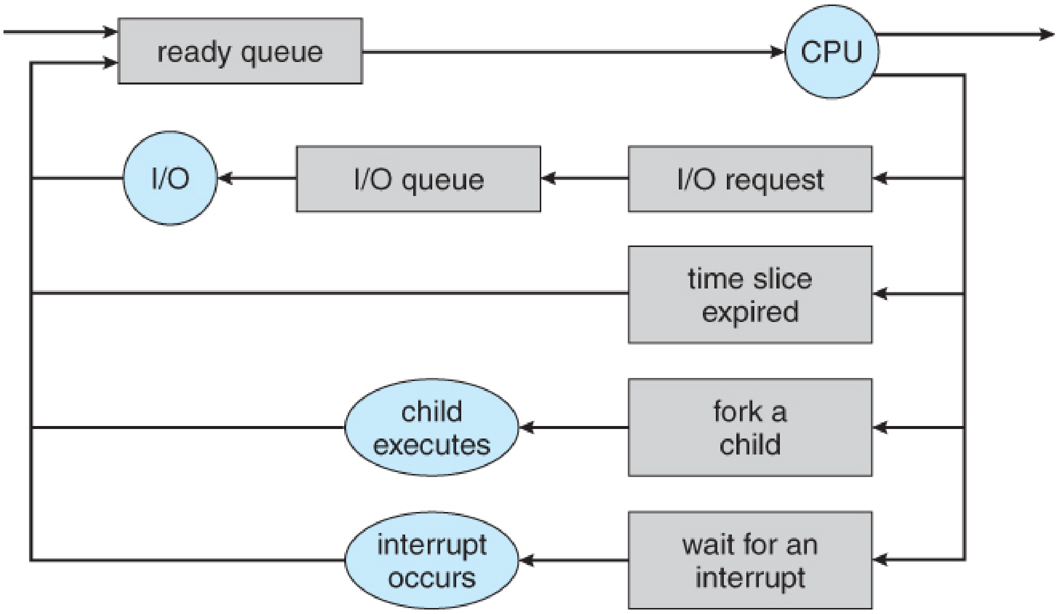
  等待使用该 IO 设备的进程队列

  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

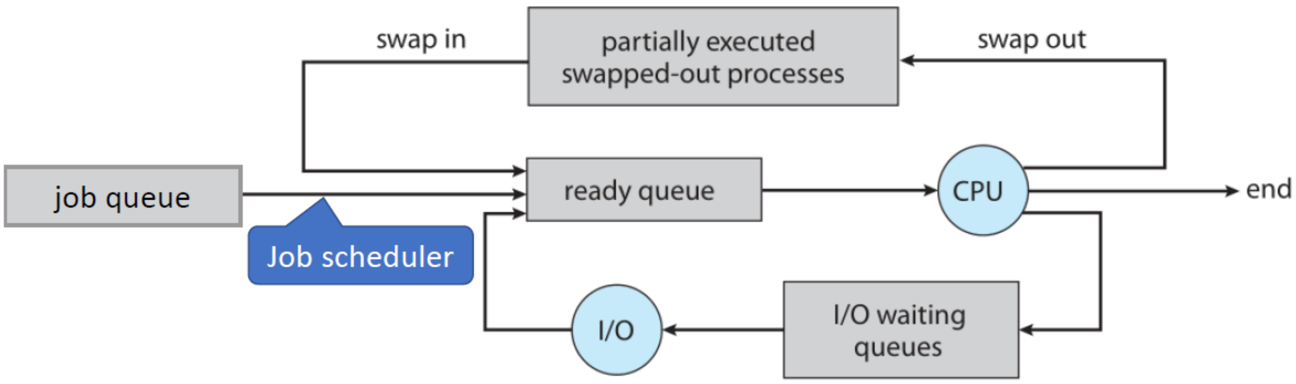
- 调度程序 (调度器)

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

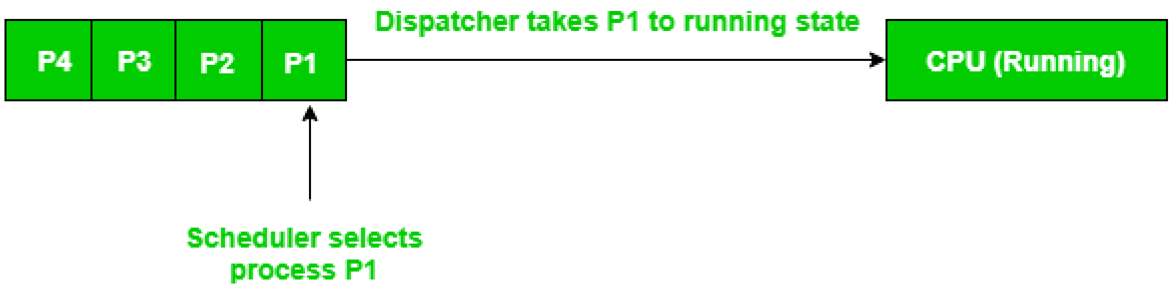
  中文书把 dispatcher 也翻译成调度程序，我真想一拳干碎你的眼镜

  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

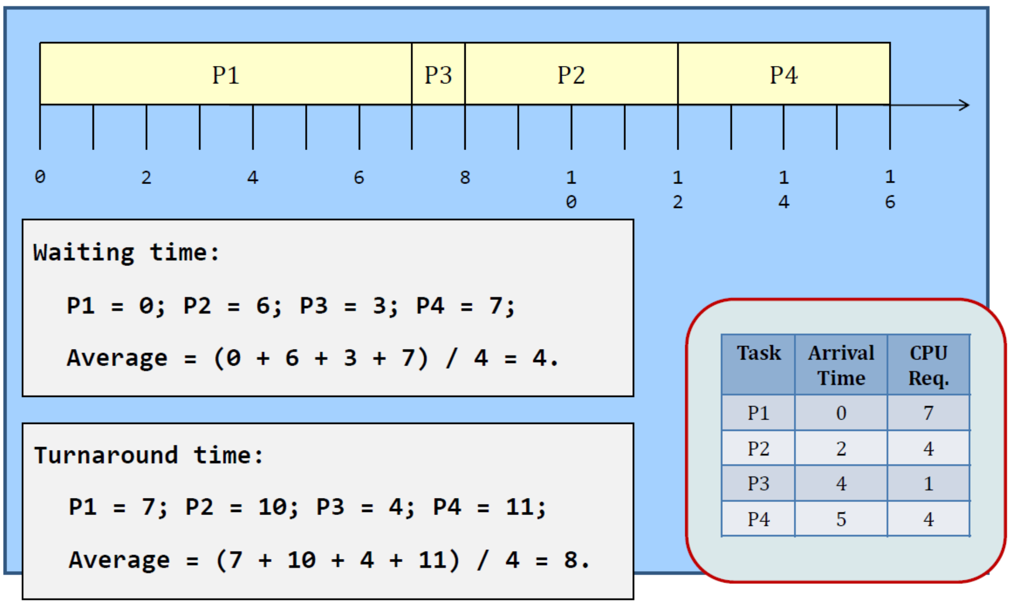
5.3.1 先到先服务调度 First-Come-First-Served (FCFS)

字面意思

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

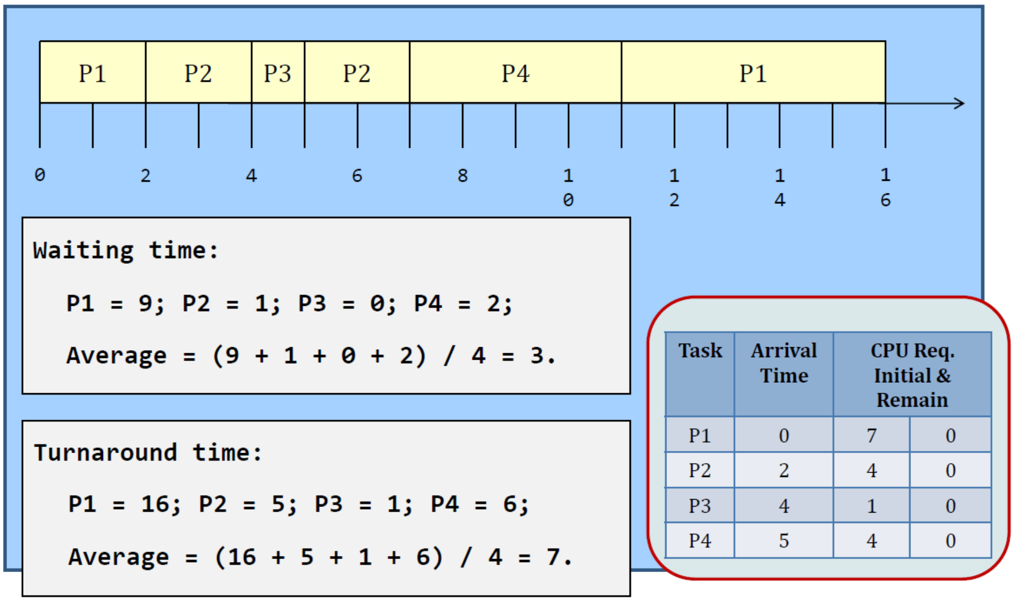
- 选择最短 CPU 执行时间的进程
- 相同，可以使用 FCFS 规则选择
- 又称最短下次 CPU执行 (Shortest-Next-CPU-Burst) 算法

5.3.2.1 非抢占 (Non-Preemptive) SJF



5.3.2.2 抢占 SJF

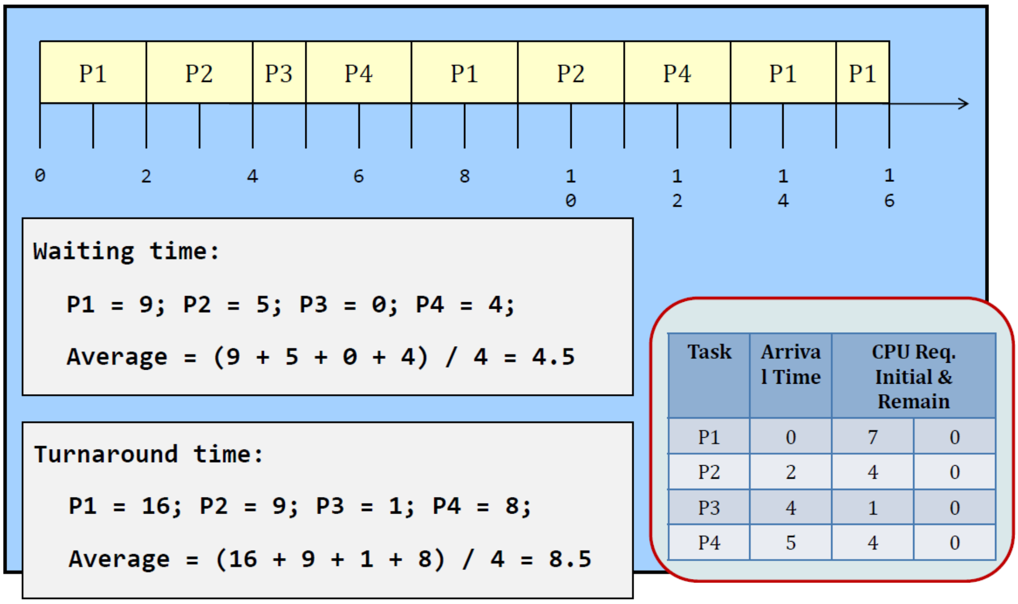
- 最短剩余时间优先 (Shortest-Remaining-Time-First)



- 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



- 缺点：性能较差
- 优点：公平

5.3.4 优先级调度 Priority Scheduling

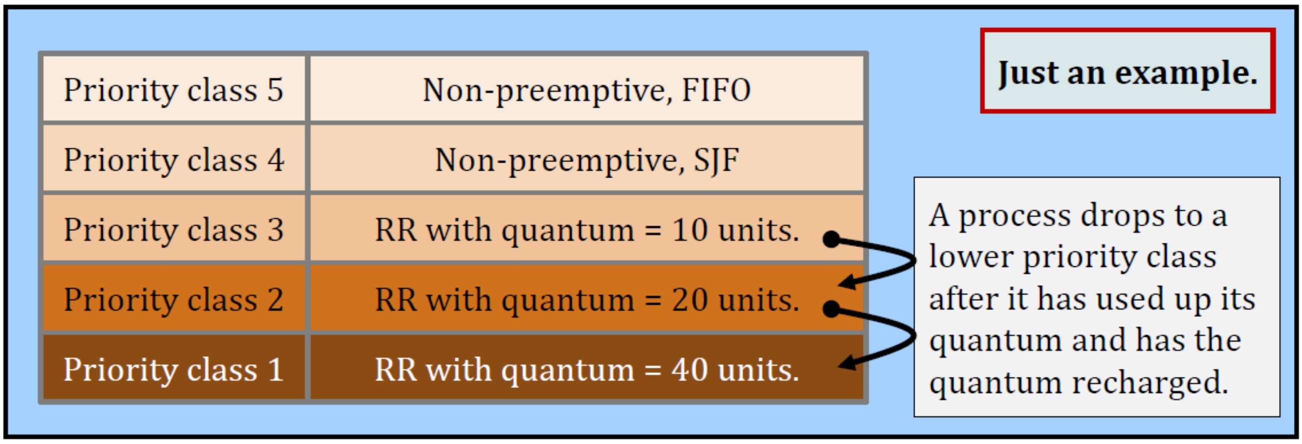
- 每个进程都有一个优先级
- 调度程序根据优先级选择进程
- 优先队列
- 分类

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.

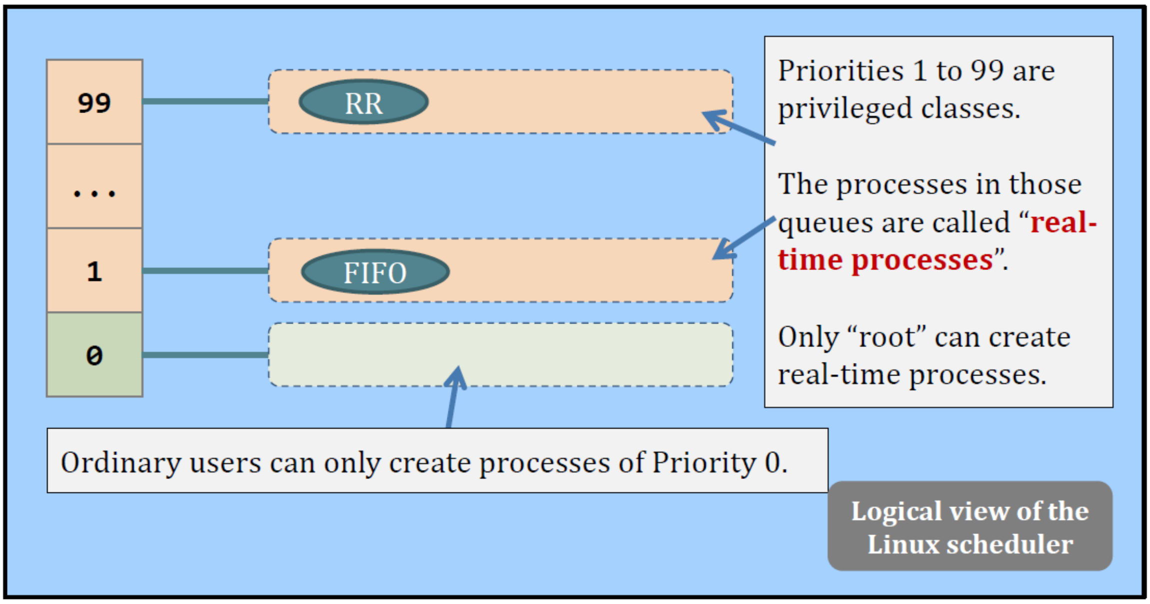
- 新进程到来时，重新 schedule (这里会发生抢占)
- 如果当前进程被抢占，它先出队再入队

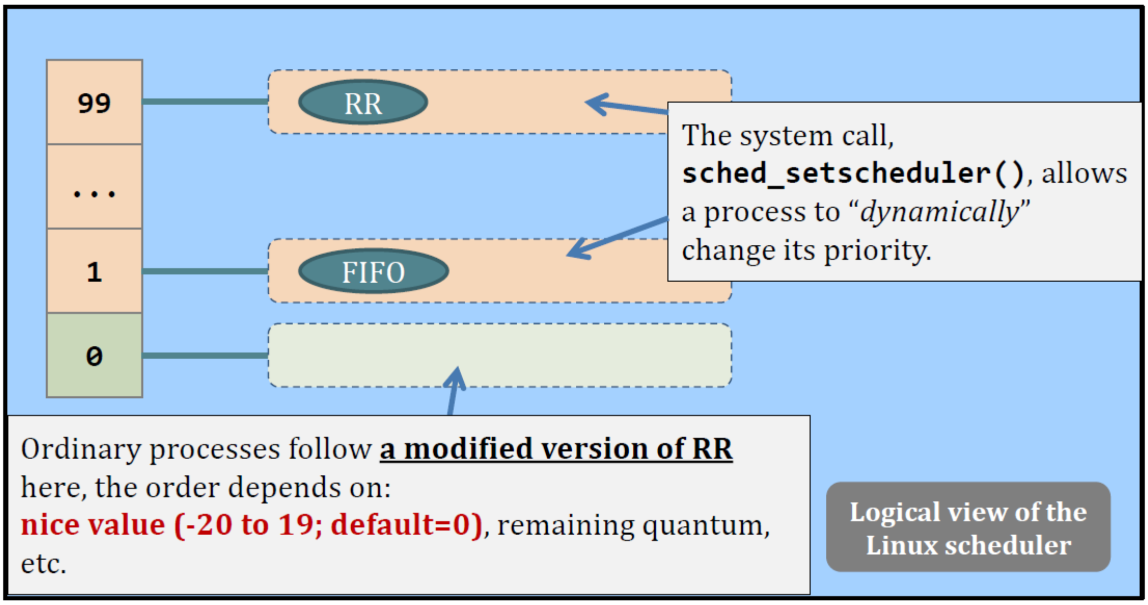
5.3.4.1 Multiple Queue Priority Scheduling

- 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

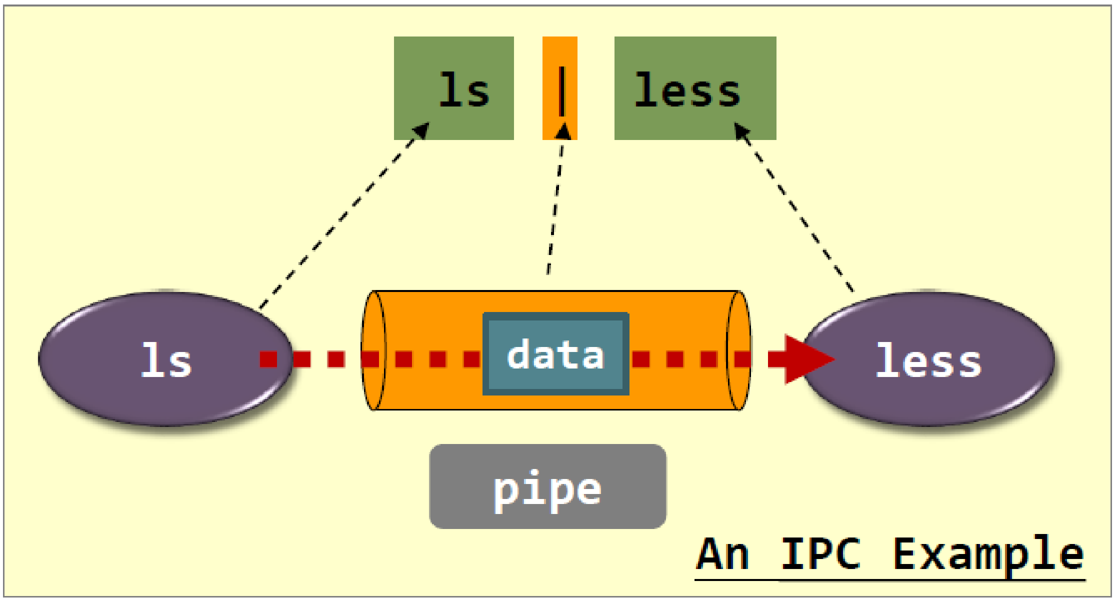

99
...
1
0
RR
FIFO
Priorities 1 to 99 are privileged classes.
The processes in those queues are called "**real-time processes**".
Only "root" can create real-time processes.
Ordinary users can only create processes of Priority 0.
Logical view of the Linux scheduler


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

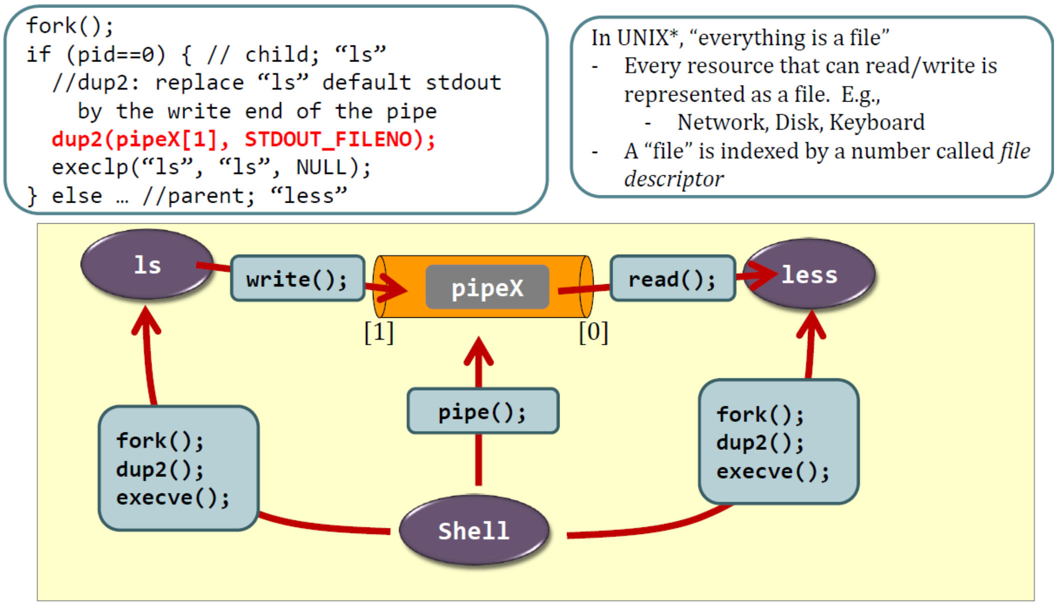
第六章 同步 Synchronization

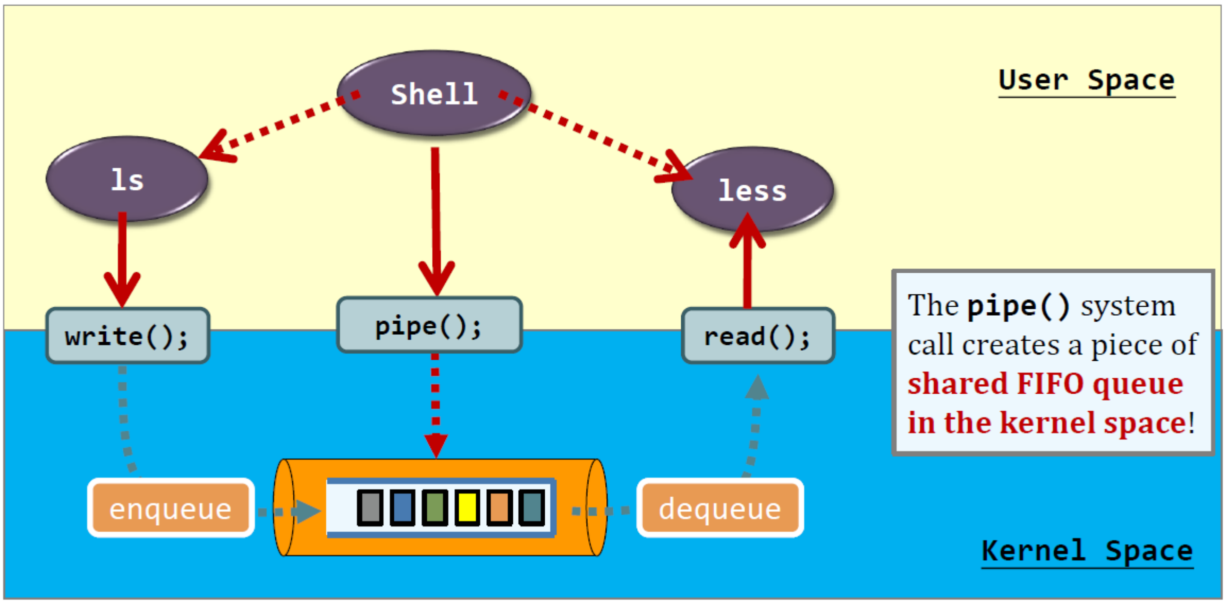
6.1 进程间通信 Inter-Process Communication (IPC)

- 匿名管道 Pipe
  - 单向 Unidirectional
  - 匿名管道只能在祖先相同的进程之间建立

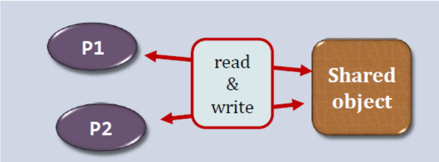
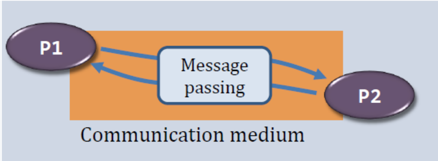


- 信号 Signal
  - More kernel-level
  - Limited (SIGKILL, SIGCHLD, ...)
- 例: `ls | less`





- IPC Models

Shared Objects	Message Passing
	
• shared files (on disk; slow) • pipes (restricted, but OS takes care of synchronization for you) • shared memory (primitive, general, but synchronization is on you) • shared address space (threading)	• socket programming • message passing interface (MPI) library for computing clusters.
- Usually single-node communication - More efficient - Need to take great care of synchronization because of sharing the same object	- Usually multi-node communication - Less efficient - Less troublesome in synchronization - But need to care of other faults (e.g., what if a network link is broken?)

6.2 临界区 Critical Section

6.2.1 竞争条件 Race Condition

- 多个进程并发访问和操作同一数据并且执行结果与特定访问顺序有关，称为竞争条件 (Race Condition)
- Shared Object + Multiple Process + Concurrency

6.2.2 临界区问题 Critical Section Problem

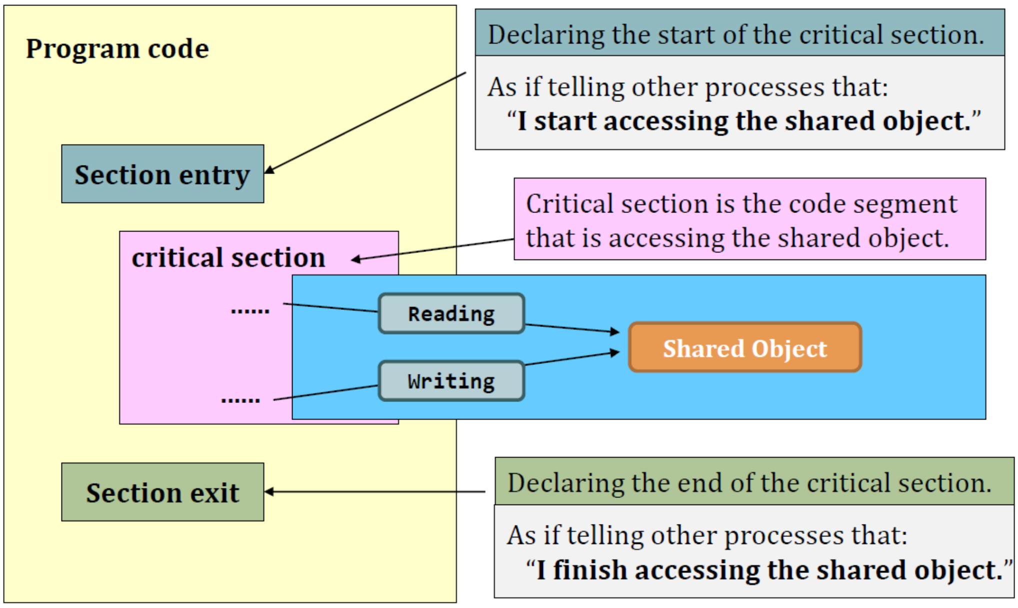
- 临界区 Critical Section

  每个进程有一段代码，进程在执行该段代码时可能修改公共变量、更新一个表、写一个文件等

- 进入区 Entry Section

  进入临界区前，请求许可的代码段

- 退出区 Exit Section
- 剩余区 Remainder Section



- 临界区问题 (Critical-Section Problem) 指设计一个协议以便协作进程，使得没有两个进程可以在它们的临界区内同时执行
- 临界区要尽可能紧凑
- 一个临界区里可以访问多个 shared object
- 重点是进入区和退出区的实现

6.2.3 临界区问题的要求

1. 互斥 Mutual Exclusion

   如果一个进程在其临界区内执行，那么其他进程都不能在临界区内执行

2. 进步 Progress

   如果没有进程在临界区内执行，并且有进程需要进入临界区，那么只有那些不在剩余区内的进程可以参加选择，以便确定谁下次进入临界区，而且这种选择不能无限推迟

   别让执行临界区的进程空着，除非大家都不想进临界区

3. 有限等待 Bounded Waiting

   从一个进程做出进入临界区的请求直到这个请求允许为止，其他进程允许进入其临界区的次数有上限

   别让一个进程等一辈子

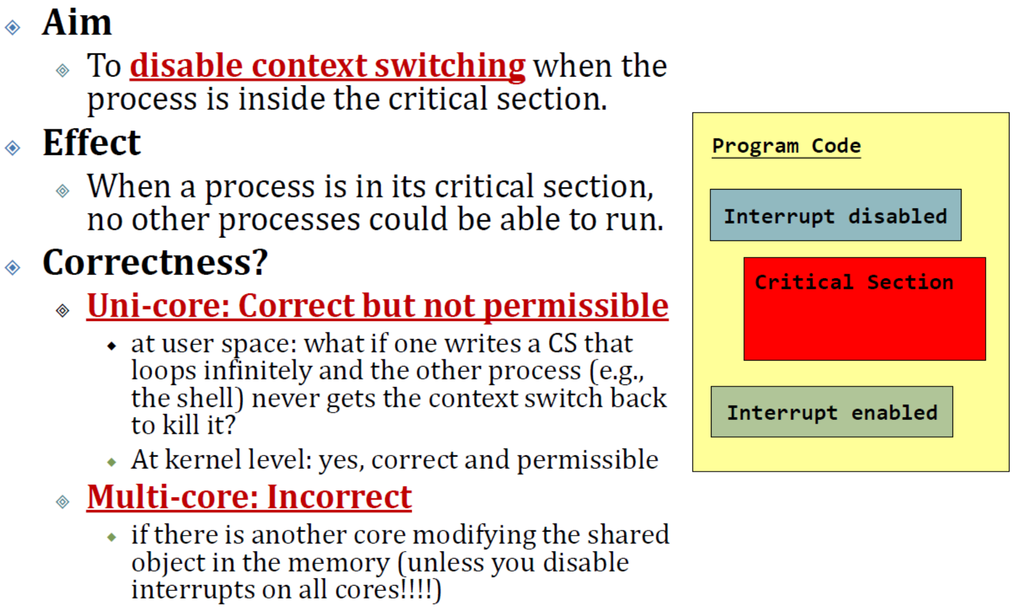
6.3 临界区问题的解决方案 Solutions for Critical Section Problem

- Lock-based
  - Spin-based Lock
    - Basic spinning

        - Peterson's solution
    - Sleep-based Lock
        - POSIX semaphore
        - `pthread_mutex_lock`
- Lock-free

6.3.1 硬件同步 (×) Hardware Synchronization

- 禁止中断



- 单核

  正确，但是不能接受

  如果有个进程在 CS 里写个死循环就全卡这了

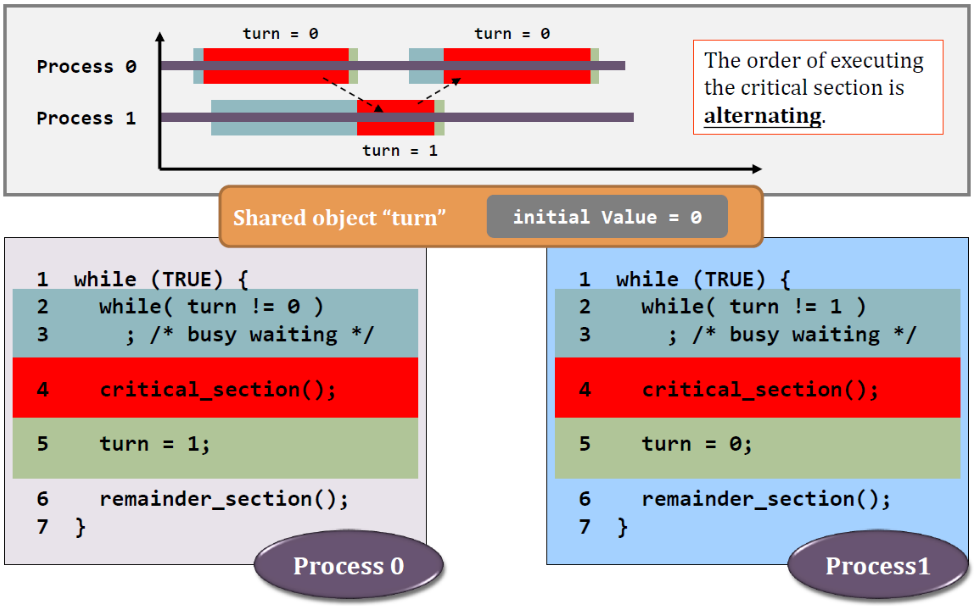
- 多核

  不正确，除非把所有核的中断全都禁止

6.3.2 基本自旋锁 (×) Basic Spin Lock

- 原理

  设置一个公共变量 `turn` 来决定哪个进程可以进 CS

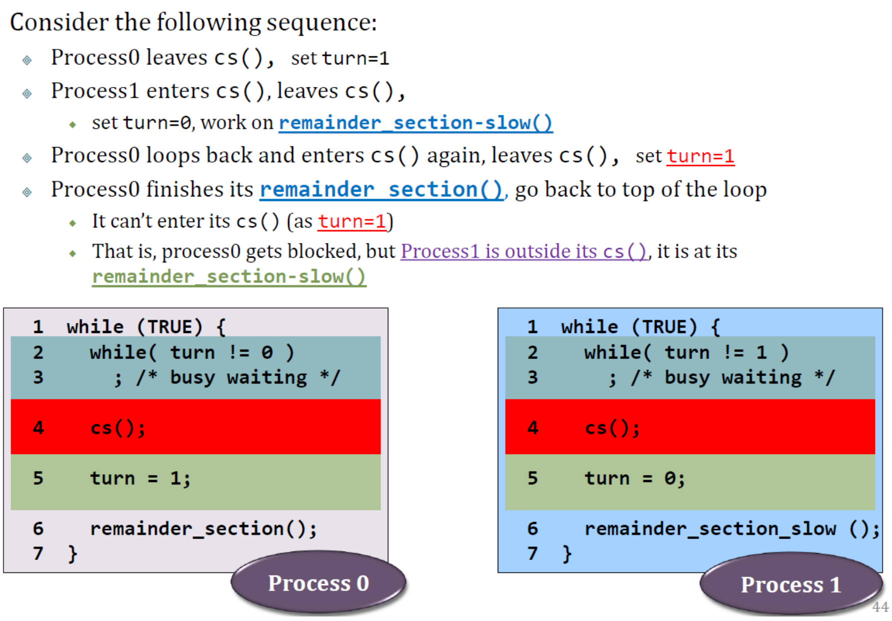


- 太浪费 CPU
- 违反 Progress

  多个进程一定是交替执行

  如果一个进程不打算进 CS 但是另一个进程交出了权限，那就要等很长时间 (**no progress**)

  Example: 这种情况下没人在 CS 里。不能让执行 CS 的进程空着



6.3.3 Peterson's Solution

- 在 `turn` 的基础上新加一个布尔数组 `interested`
  - If I don't show interest

    I let you all go
  - If we both show interest

    Take turns

```c
int turn;
int interested[2] = {false, false};

void lock(int process) {
    int other = 1 - process;
    interested[process] = true;
    turn = other;
    while (turn == other && interested[other]); // busy waiting
}

void unlock(int process) {
    interested[process] = false;
}
```

- 会产生优先级翻转问题 (Priority Invasion)

  优先级 $A < B < C$

  1. A 获得锁, C 来了, C 申请锁
  2. 按理来说 C 应该抢占 A, 但是锁在 A 手里, C 就只能等待
  3. B 不要锁，所以 B 可以被调度上去
  4. 明明 B 优先级低，却比 C 先执行
- 为什么 `turn=other` 不是 `turn=process`

  我们假设是这样

  ```
  turn=自己;
  while (turn==自己 && interested[别人]);
  ```

  如果现在有三个进程 P_1, P_2, P_3

  1. 我们脸比较黑，这三个进程经过调度，都该执行 `turn=自己` 这一行
  2. 那么最终 `turn` 是几，就取决于调度器了
  3. 假设调度器就是按 P_1, P_2, P_3 的顺序调度的，那么最后 `turn=3`
  4. 现在我们检查 `while` 的条件
     - 对于 P_1, `turn=3`, 前半句不成立, 不需要 wait
     - 对于 P_2, `turn=3`, 前半句不成立, 不需要 wait
     - 对于 P_3, 条件成立, 需要 wait
  5. 那么现在 P_1, P_2 都被许可进入 CS，违反了**互斥**原则

  正确是这样:

  ```
  turn=别人;
  while (turn==别人们 && interested[别人们]);
  // while ((turn==x || turn==y) && (interested[x] || interested[y]))
  // while (turn!=自己 && interested[别人们])
  ```

  还是这个例子，我们假设 `turn` 的赋值是 1 给 2，2 给 3，3 给 1

1. 还是都执行到 `turn=别人` 这一行，还是按 123 的顺序调度的
2. 那最终 `turn=1`
3. 检查 `while` 的条件，只有 P_1 可以进 CS

6.3.4 信号量 Semaphore

- 信号量是一个 Structure
  - 一个 `int`，表示剩余多少资源可用
  - 一个等待队列

```c
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

- Wait (P 操作)

```c
wait(semaphore *s) {
    s->value--;
    if (s->value<0) {
        add this process to s->list;
        block();
    }
}
```

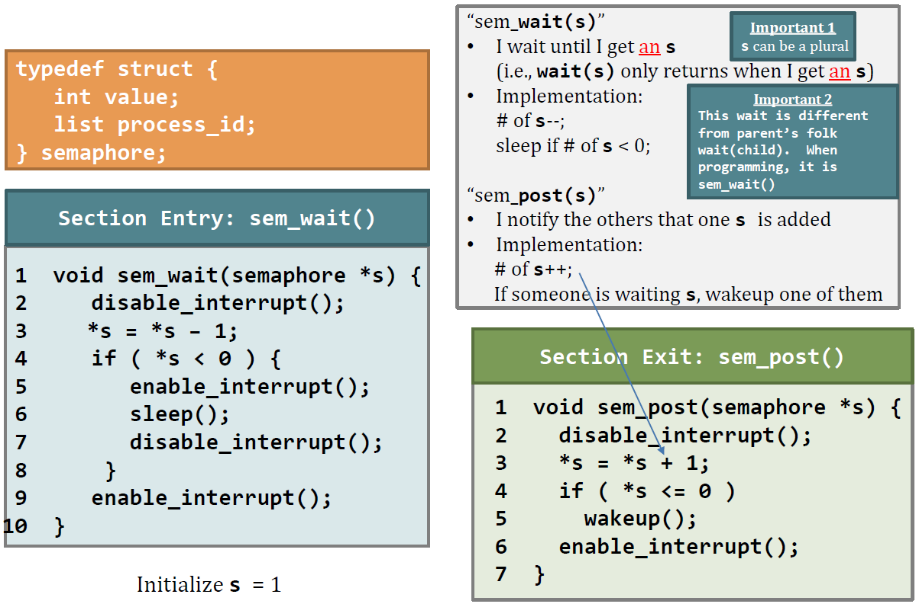
- Post (V 操作)

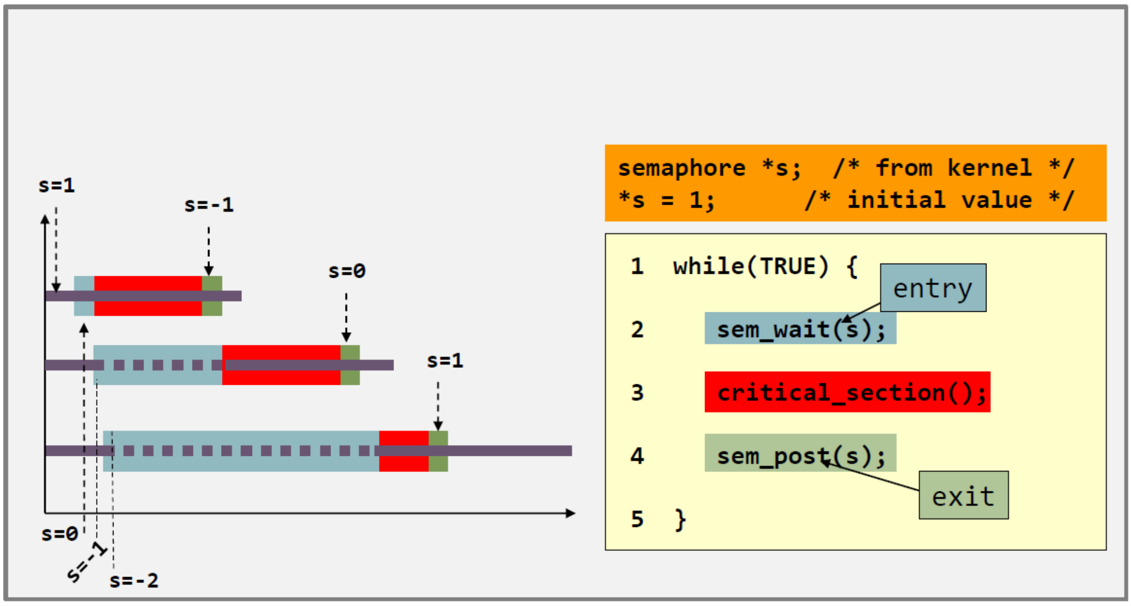
```c
post(semaphore *s) {
    s->value++;
    if (s->value<=0){
        remove a process p from s->list;
        wakeup(p);
    }
}
```

- 分类
  - 二进制信号量 Binary Semaphore

    只能 0 或 1
  - 计数信号量 Counting Semaphore

    可以 > 1



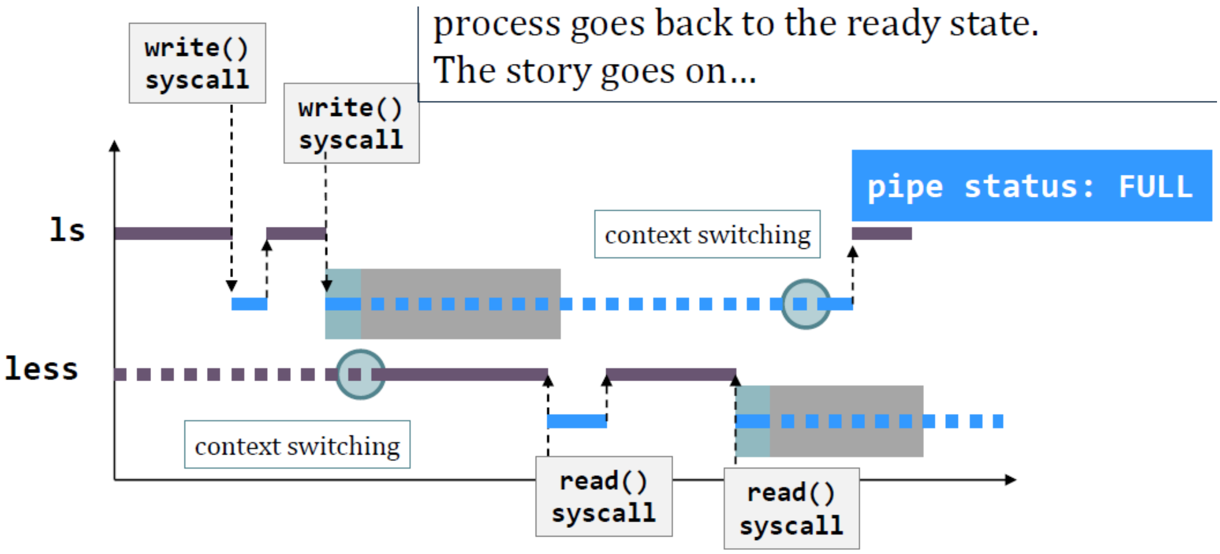


6.4 经典同步问题

6.4.1 有界缓冲问题 Bounded-Buffer Problem

- 又称生产者-消费者问题 (Producer-Consumer Problem)
- 组成
    1. Bounded Buffer
        - Shared object
        - Limited size
        - Queue
    2. Producer Process
        - Produce a unit of data and writes that piece of data to the tail of the buffer at one time
    3. Consumer Process
        - Remove a unit of data from the head of the buffer at one time

- 要求
  1. Producer
     - 当 producer 向 buffer 里放入数据，但是 buffer 已经满的时候，他需要 wait
     - 放入数据后，通知 consumer (wake up)
  2. Consumer
     - 当 consumer 要消费数据，但是 buffer 是空的，他需要 wait
     - 消费数据之后，通知 producer (wake up)
- 例子



- Semaphore 实现

```c
semaphore mutex=1;
semaphore avail=N;
semaphore fill=0;

void producer() {
    int item;

    while (true) {
        item=produce_item();

        wait(&avail);
        wait(&mutex);

        insert_item(item);

        post(&mutex);
        post(&fill);
    }
}

void consumer() {
    int item;

    while (true){
```

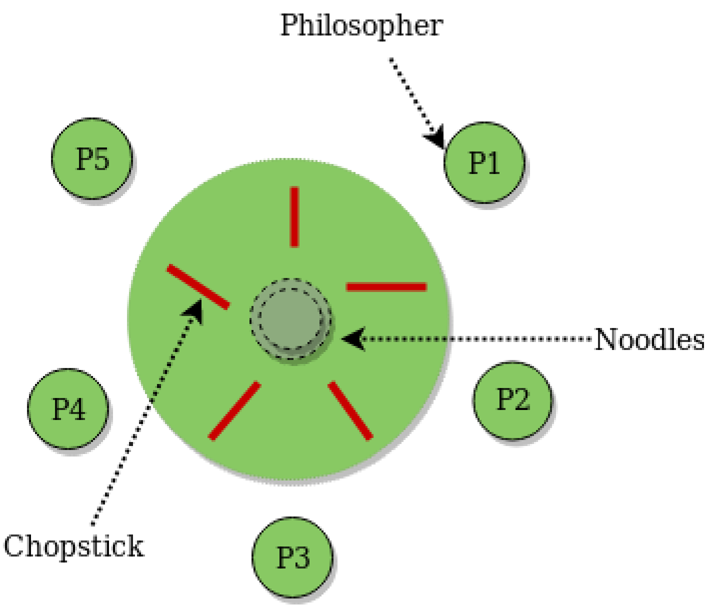
```
        wait(&fill);
        wait(&mutex);

        item=remove_item();

        post(&mutex);
        post(&avail);
    }
}
```

6.4.2 读者-作者问题 Reader-Writer Problem

- 要求
  - 任何数量的 reader 都可以同时 read
  - 同时只能有一个 writer 写
  - 如果有 writer 在写，那么所有 reader 都不能读

6.4.3 哲学家就餐问题 Dining-Philosophers Problem

- 问题描述
  - 有 5 个哲学家，5 根筷子，1 盘面条

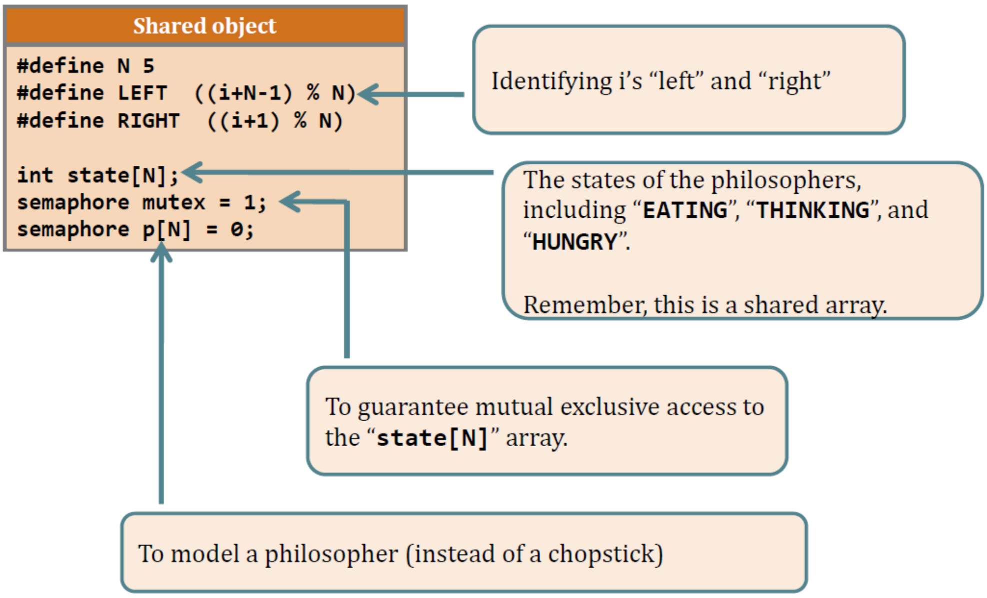


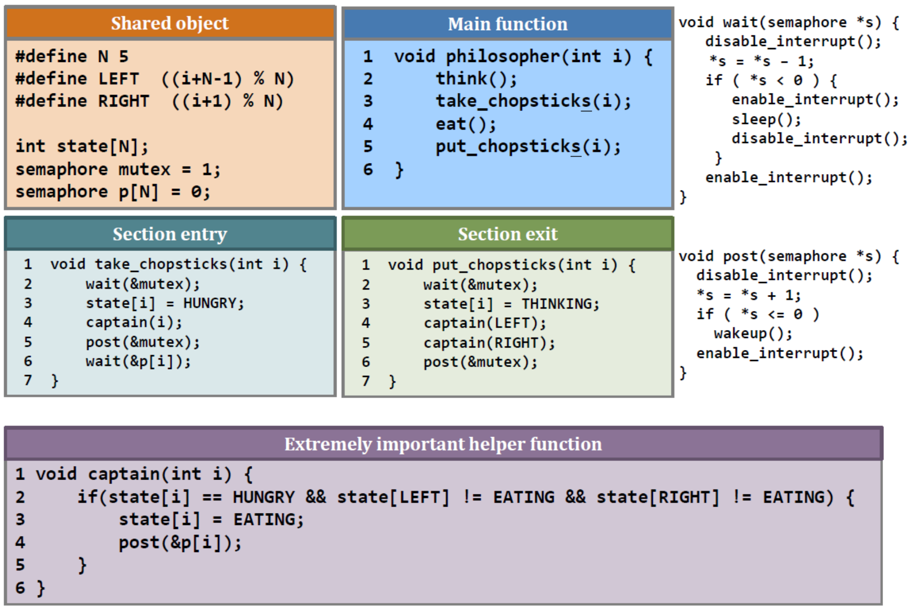
  - 每个哲学家有两个可能的动作
    - Think
    - Eat
  - 如果一个哲学家要吃面条，他必须同时获得左右两根筷子
  - 拿起来的筷子不会被别人抢
- 要求

  设计一个 Protocol，保证所有哲学家

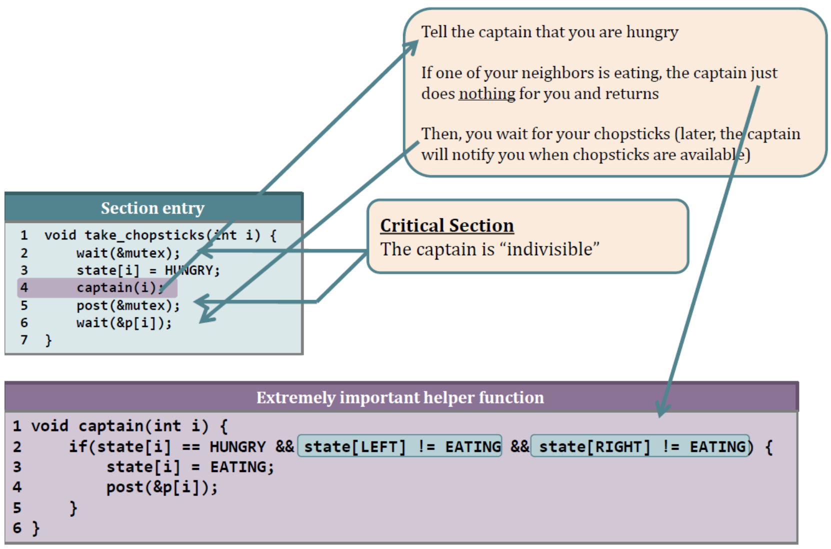
  - 不会饿死
  - 不会死锁
- 解决方案设计

- 如果一个哲学家想吃面条，那么他先问左右
- 如果左右都不在吃，那么他拿两根筷子吃
- 如果左右有人在吃，他就饿着等着，直到别人吃完了通知他
- 吃完之后，他放下筷子并且通知左右他吃完了





Dining philosopher – Hungry



- Finish Eating

Tell the captain
Try to let your **left neighbor** to eat.

Tell the captain
Try to let your right **neighbor** to eat.

Section exit

```
void put_chopsticks(int i)
{
    wait(&mutex);
    state[i] = THINKING;
    captain(LEFT);
    captain(RIGHT);
    post(&mutex);
}
```

Extremely important helper function

```
void captain(int i) {
    if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        post(&p[i]);
    }
}
```

Wake up the one who is sleeping

第七章 死锁 Deadlock

7.1 死锁的概念

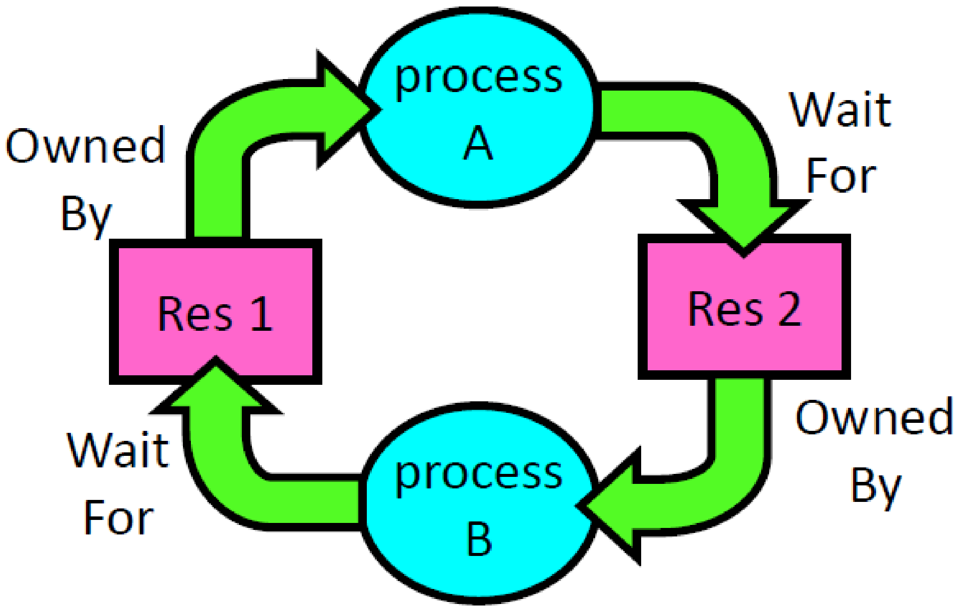
- 在正常操作模式下，进程只能按如下顺序使用资源:
  1. 申请

     进程请求资源。如果进程不能立即被允许，那么它应该等待，直到获取该资源
  2. 使用

     进程对资源进行操作
  3. 释放

     进程释放资源
- 死锁 Deadlock

  Deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.



- 饥饿 Starvation

  Indefinite Blocking

  A condition in which a process is indefinitely delayed because other processes are always given preference.

  Starvation is the problem that occurs when high priority processes keep executing and low priority processes get blocked for indefinite time.

  Deadlock 一定会造成 starvation

7.2 死锁的特征

7.2.1 死锁的必要条件

1. 互斥 Mutual Exclusion

   Only one thread at a time can use a resource.

2. 占有并等待 Hold and Wait

   一个进程应占有至少一个资源并等待另一个资源，而该资源为其他进程所占有

3. 非抢占 No Preemption

   资源不能被抢占，即资源只能被进程在完成任务后自愿释放

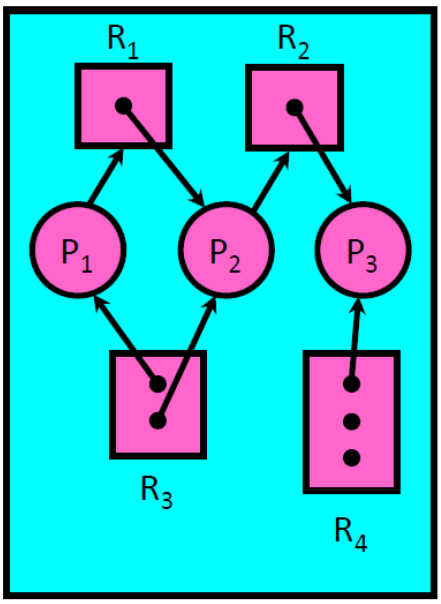
4. 循环等待 Circular Wait

   有一组等待进程 $\{P_0, P_1, P_2, ..., P_n\}$

   - P_0 等待的资源被 P_1 占有
   - P_1 等待的资源被 P_2 占有
   - ...
   - P_n 等待的资源被 P_0 占有

注意是必要条件，即使这些条件都满足也不一定死锁，还需要运气比较背

7.2.2 资源分配图 Resource-Allocation Graph



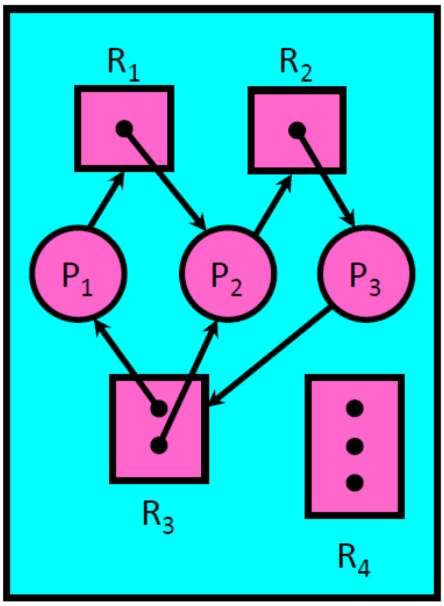
- 圆表示进程
- 矩形表示资源
- 矩形内的点表示资源实例
- 申请边 Request Edge

  进程指向资源的边

- 分配边 Assignment Edge

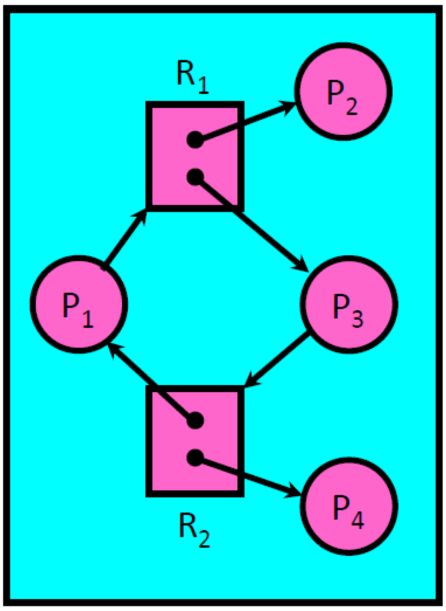
资源指向进程的边

- 如果分配图没有环，那么系统一定没有死锁；如果有环，那么可能存在死锁
- 死锁的例子



- 有环没死锁的例子

  让 P_2, P_4 先执行完



7.3 死锁的处理方法

- 通过协议来预防或避免死锁，确保系统不会进入死锁状态
- 允许系统进入死锁状态，然后检测并恢复
- 忽视，认为死锁不可能在系统内发生

  这种方案被 Linux, Windows 等大多数 OS 采用

  就算出现了死锁，OS 也不管

7.4 死锁检测 Deadlock Detection

7.4.1 死锁检测算法

[xxx] 表示数组

- [FreeResources]: current free resources each type
- [Request$_X$]: current requests from process X
- [Alloc$_X$]: current resources held by process X

```
[Avail] = [FreeResources]
Add all nodes to UNFINISHED

do {
    done = true
    Foreach node in UNFINISHED {
        if ([Request_node] ≤ [Avail]) {
            remove node from UNFINISHED
            [Avail] = [Avail] + [Alloc_node]
            done = false
        }
    }
} until(done)
```

7.4.2 死锁恢复

当检测到死锁后:

- 进程终止

  Terminate thread, force it to give up resources
- 资源抢占

  Preempt resources without killing off process
- 回滚

  Roll back actions of deadlocked threads
- Many operating systems use other options

7.5 死锁预防 Deadlock Prevention

核心：打破四个必要条件

1. 互斥
    - 大家都用只读文件
    - 给足够多的资源
2. 持有且等待

    - 每个进程在执行前申请并获得所有资源
    - 进程仅在没有资源时才申请资源
3. 无抢占
    - 如果一个进程持有资源并申请一个不能被立即分配的资源，那么它现在分配的资源都可以被抢占

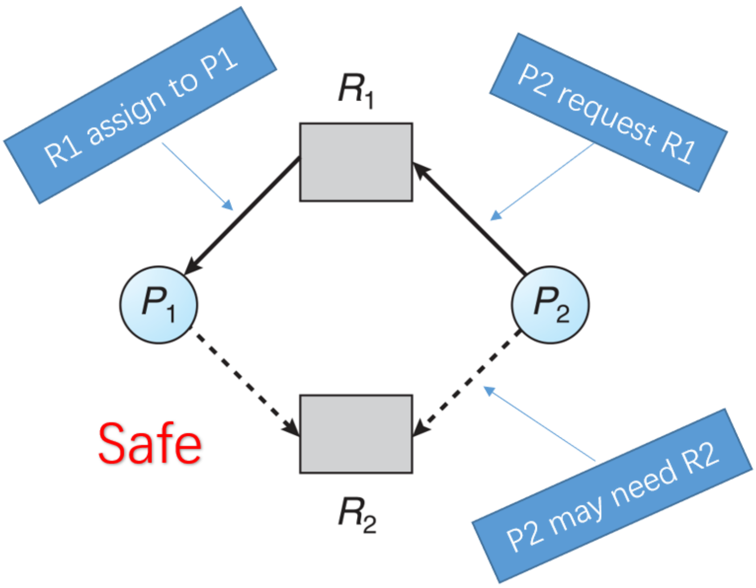
      相当于把它现有的资源都释放了
4. 循环等待
    - 给所有进程一个指定的顺序来申请资源

7.6 死锁避免 Deadlock Avoidance

7.6.1 安全状态

- 如果系统能按一定顺序为每个进程分配资源（不超过其最大需求），可以避免死锁，那么系统的状态就是安全的 (safe)
- 只有存在一个安全序列 (safe sequence)，系统才处于安全状态
- 如果没有这样的序列存在，那么系统状态就是非安全 (unsafe)
- 非安全状态只是可能会导致死锁，不是一定

7.6.2 资源分配图算法



- 需求边 Claim Edge

  进程指向资源，虚线

  进程 P_i 可能在将来申请某个资源 R_j
- 只有在将申请边变成分配边 (反向实线箭头) 并且不会导致资源分配图形成环时，才能允许申请
- 时间复杂度

  $O(n^2)$, n 为进程数量

7.6.3 银行家算法 Banker's Algorithm

n 个进程, m 种资源

- $Available$: 行向量，表示每种资源的可用实例数量
- Max: $n \times m$ 矩阵，每个进程的最大需求
- $Allocation$: $n \times m$ 矩阵，每个进程已经分配的实例数量
- $Need = Max - Allocation$，还缺多少实例才能完事

```
Add all nodes to UNFINISHED

do {
    done = true
    Foreach node in UNFINISHED {
        if ([Max_node] - [Alloc_node] ≤ [Avail]) {
            remove node from UNFINISHED
            [Avail] = [Avail] + [Alloc_node]
            done = false
        }
    }
} until(done)
```

- 例: 可以按 0213 或 0231 的顺序执行完

<table>
<tr><th rowspan="2"></th><th colspan="4">Allocation</th><th colspan="4">Max</th><th colspan="4">Available</th></tr>
<tr><th>A</th><th>B</th><th>C</th><th>D</th><th>A</th><th>B</th><th>C</th><th>D</th><th>A</th><th>B</th><th>C</th><th>D</th></tr>
<tr><td>P0</td><td>0</td><td>0</td><td>1</td><td>2</td><td>0</td><td>0</td><td>1</td><td>2</td><td rowspan="4">1</td><td rowspan="4">5</td><td rowspan="4">2</td><td rowspan="4">0</td></tr>
<tr><td>P1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>7</td><td>5</td><td>0</td></tr>
<tr><td>P2</td><td>1</td><td>3</td><td>5</td><td>4</td><td>2</td><td>3</td><td>5</td><td>6</td></tr>
<tr><td>P3</td><td>0</td><td>0</td><td>1</td><td>4</td><td>0</td><td>6</td><td>5</td><td>6</td></tr>
</table>

- 现在有个新的 $Request$，比如 $P_0\ (1,3,1,0)$
    1. 检查 $Request_0 < Available$
    2. 检查 $Allocation_0 + Request_0 < Max_0$
    3. 假设把资源分配给 P_0
        - 如果分配之后还是 safe（能找到安全序列），那就真的分配给它
        - 如果分配之后 unsafe，拒绝请求
