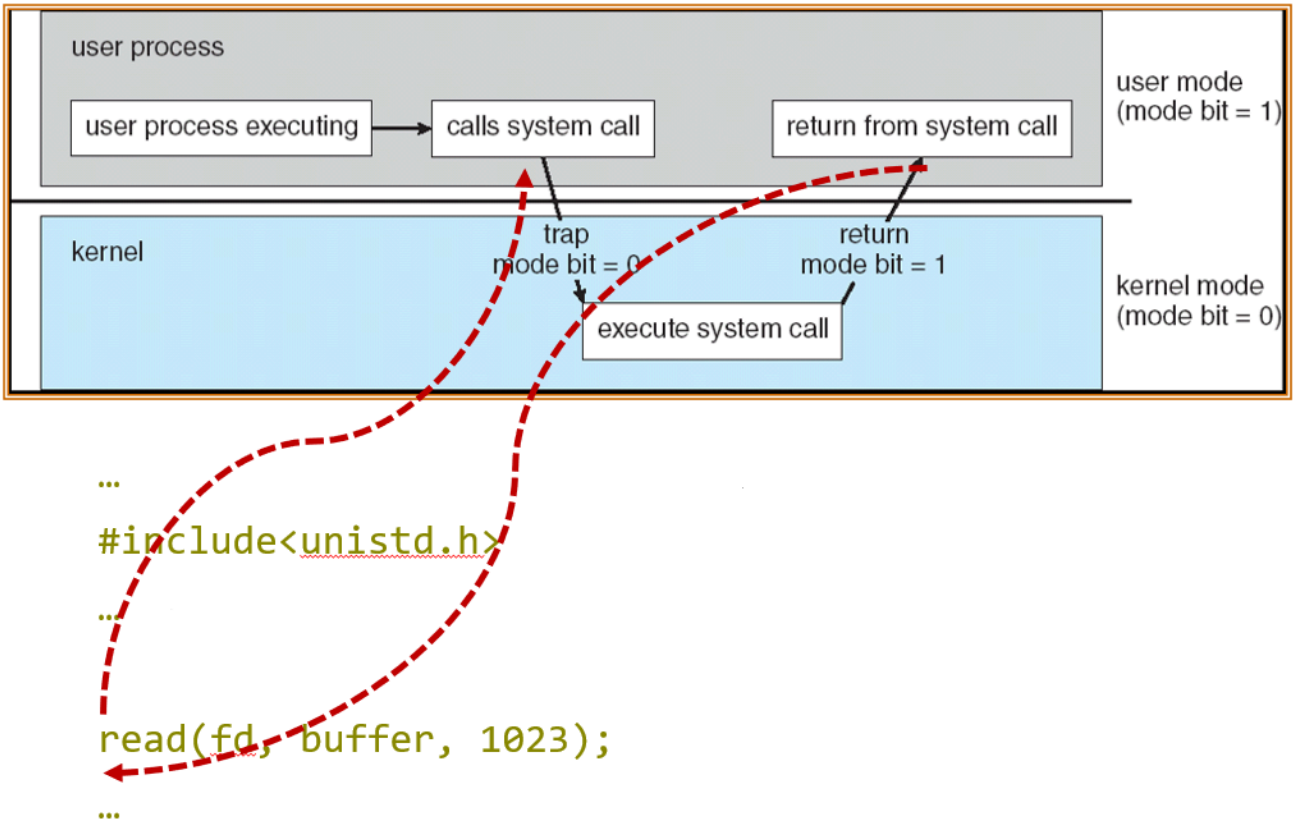
# CH1 Overview

- bootstrap program
  - 存储在**ROM**当中，将操作系统加载到内存中
- 冯诺依曼体系计算机的运行以**内存**为中心
  - 通过总线连接硬件
- 计算机系统的操作
  - I/O设备和CPU可以并发执行
    - 并行（parallel）和并发（concurrent）：任意时刻都在同时运行叫做并行；并发并非并行，只是相对的同时执行
  - 特定设备类型都有一个控制器
  - 每个设备控制器都有一个local buffer
    - buffer解决CPU和设备运行速度之间的mismatch
  - CPU将数据在内存和缓存之间运输
  - I/O是从设备到设备控制器的local buffer
    - 两种I/O方式
      - 同步：I/O开始后，只有当I/O完成时控制信号才会返回用户程序
      - 异步：I/O开始后，控制信号返回用户程序时不会等待I/O完成
  - **设备控制器**通过中断interrupt通知CPU它已经完成了操作
    - 中断通过interrupt vector将控制信息传递给ISR（interrupt service routine），**interrupt vector存储**所有service routine的地址
    - 中断结构必须存储被中断的指令的地址
    - 如果在有中断正在执行时，新的中断会被disable防止中断丢失
    - trap是软件产生的中断，通过错误或者用户请求（系统调用）产生
    - 操作系统是interrupt driven的
    - 中断处理
      - 操作系统通过**寄存器**（当前正在处理的事物）和**program counter**（下一条指令的地址）来保存CPU的状态
      - 两种判断中断类型的方法：polling；vectored
      - 分离的代码段决定每种类型的中断需要执行的动作
- Direct Memory Access Structure（DMA）
  - 为高速I/O设备提供的使其可以用接近内存的速度传递信息
  - 设备控制器在没有CPU干预的情况下从local buffer直接传递数据块给内存
  - 每个块会产生一个中断，而不是每个字节一个中断
- Storage Structure（speed速度 cost成本 volatility易失性）
  - 主存：CPU可以直接访问的大存储媒介
  - 二级存储：主存的扩展，提供大量非易失的存储空间

    - 磁盘：
    - Caching：复制信息到快速存储系统，高速设备缓存低速设备
    - Multitasking
    - Multiprocessor
    - Multicore
    - NUMA Architecture
        - CPU通过共享的system interconnect连接
        - 随着处理器的增加，规模更高效
        - 通过interconnect的远程内存速度很慢
        - 操作系统需要细致的CPU规划和内存管理
- Operating System Structure
    - Multiprogramming（**并发concurrent**）
        - 组织所有的任务，CPU总可以有一个任务去执行
        - 系统中所有任务的子集被存储在内存中
        - 任务通过job scheduling（任务调度）被选择和执行
        - OS切换至其他任务当他必须停止时（例如I/O）
    - **Timesharing（multitasking）**
        - CPU在各个任务之间频繁切换，用户可以在各个任务执行时与之互动，创造可交互计算环境
        - 响应时间应小于1秒
        - 每个用户至少有一个程序在内存中执行，被称之为进程（process）
        - 几个任务同时准备运行，被称之为CPU调度
        - 如果进程不在内存中，**swapping**将进程移入或移出内存
        - **虚存（Virtual memory）**允许进程的执行不完全在内存中
- 操作系统操作
    - Dual-mode：User mode 和 Kernel mode；允许操作系统保护自己和系统其他组件
    - 硬件提供Mode bit：提供了辨别系统正在运行哪种模式的能力；使得一些指令需要更高权限只能在Kernel mode执行；系统调用切换至kernel mode，从调用返回切换这user mode

- 

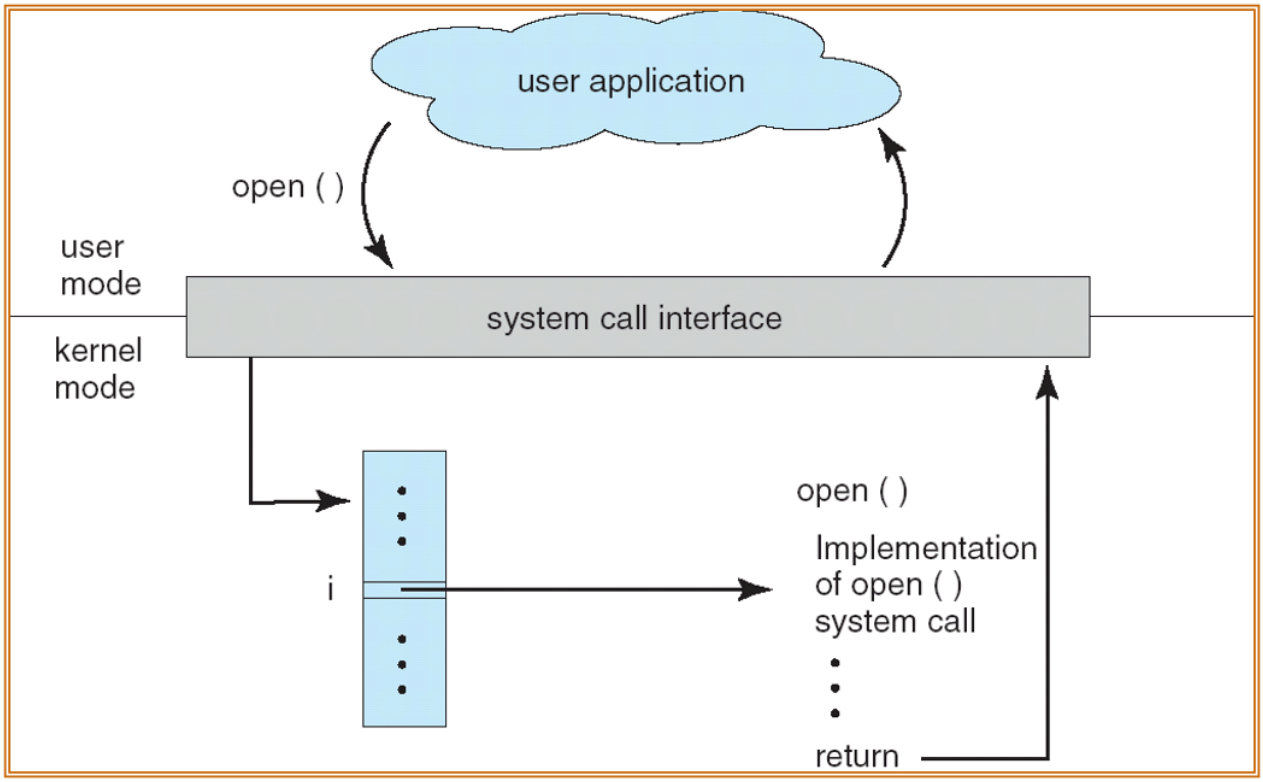

```
…
#include<unistd.h>
…

read(fd, buffer, 1023);
…
```

- Timer & Interrupt
  - 防止死循环
    - 在一定时间后设置中断
    - 操作系统减少计数器，计数器为0是发生中断
    - 在进程调度前开始运作，重新拿回控制权或终止超出规定时间的程序
- 进程管理
  - 进程是正在执行的程序
  - 进程终止需要**回收所有可重用资源**
  - 单线程进程有一个程序计数器，指定要执行的下一条指令的位置
    - 进程线性执行指令，一次一条直到完成
  - 多线程进程每个线程都有一个程序计数器
  - 通常，系统有许多进程、一些用户、一些操作系统在一个或多个CPU上并发运行
    - 通过在进程/线程之间**复用CPU实现并发**
  - 进程管理事务
    - 创建和删除用户和系统进程
    - 暂停和恢复进程
    - 提供进程同步机制
    - 提供进程沟通机制
    - 提供死锁解决机制
- 内存管理
  - 所有数据在处理前后都需要通过内存
  - 所有指令必须在执行中才能执行

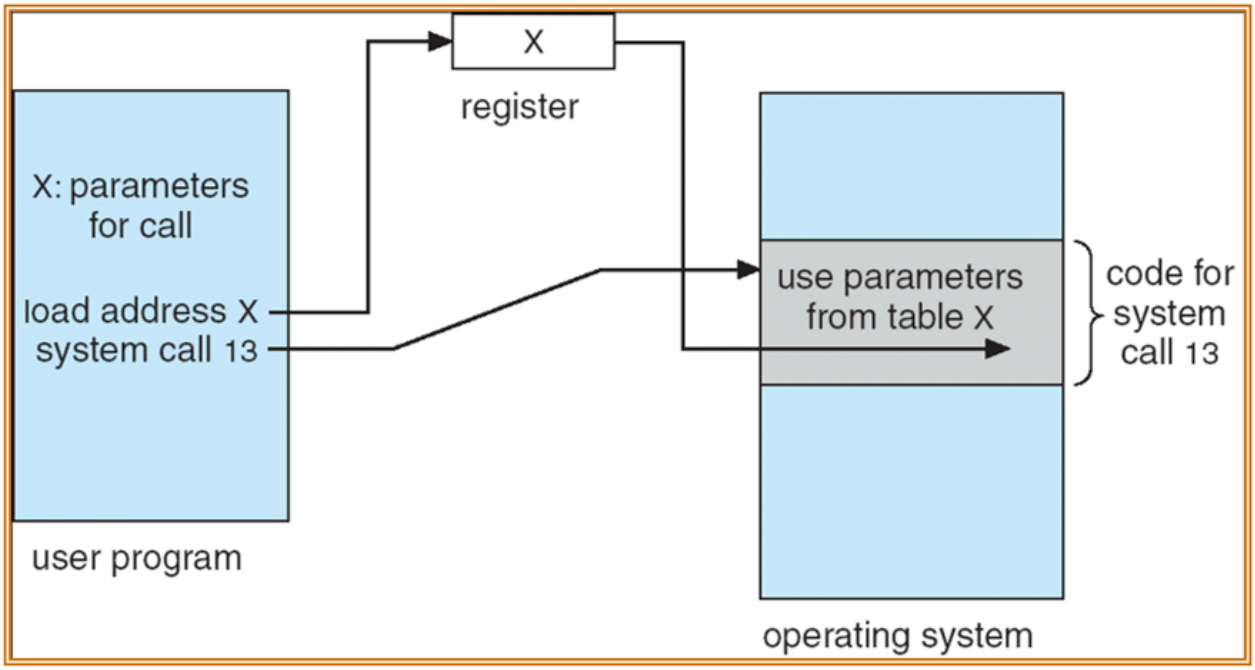
    - 内存管理决定内存中的内容
    - 内存管理事务
        - 跟踪当前正在使用内存的哪些部分以及由谁使用
        - 决定哪些进程（或其部分）和数据移入和移出内存
        - 根据需要分配和释放内存空间
- 存储管理
    - 文件系统管理
        - 访问控制
        - 创建删除文件和目录
        - 将文件映射进二级存储
        - 在非易失性存储媒介备份文件
- I/O子系统
    - I/O内存管理包括buffering，caching，spooling（一个任务的输出与其他任务的输入重叠）
    - 通用设备驱动接口
    - 特定硬件设备驱动

# CH2 Operating-System Structures

- 操作系统服务
    - user interface（UI）
    - 程序执行
    - I/O操作
    - 文件系统管理
    - 沟通
    - 错误检测
    - 资源分配
    - 审计：资源使用情况
    - 保护和安全
- UI
- System call系统调用
    - 操作系统提供的服务的程序接口
    - 大多数系统调用通过高层次的API在程序中来访问，而不是直接使用系统调用
    - 为什么用API而不是系统调用？系统调用更复杂，API更轻松方便
    - 系统调用的实现
        - 每个系统调用都有一个编号，系统调用的接口维护了一张通过这个编号索引的表
        - 系统调用接口在操作系统内核中调用预定的系统调用，并返回系统调用的状态和任何返回值




- 系统调用的参数传递
  - 通过寄存器传递
  - 参数存储在内存中的块或表中，块的地址作为调用参数在寄存器中传递



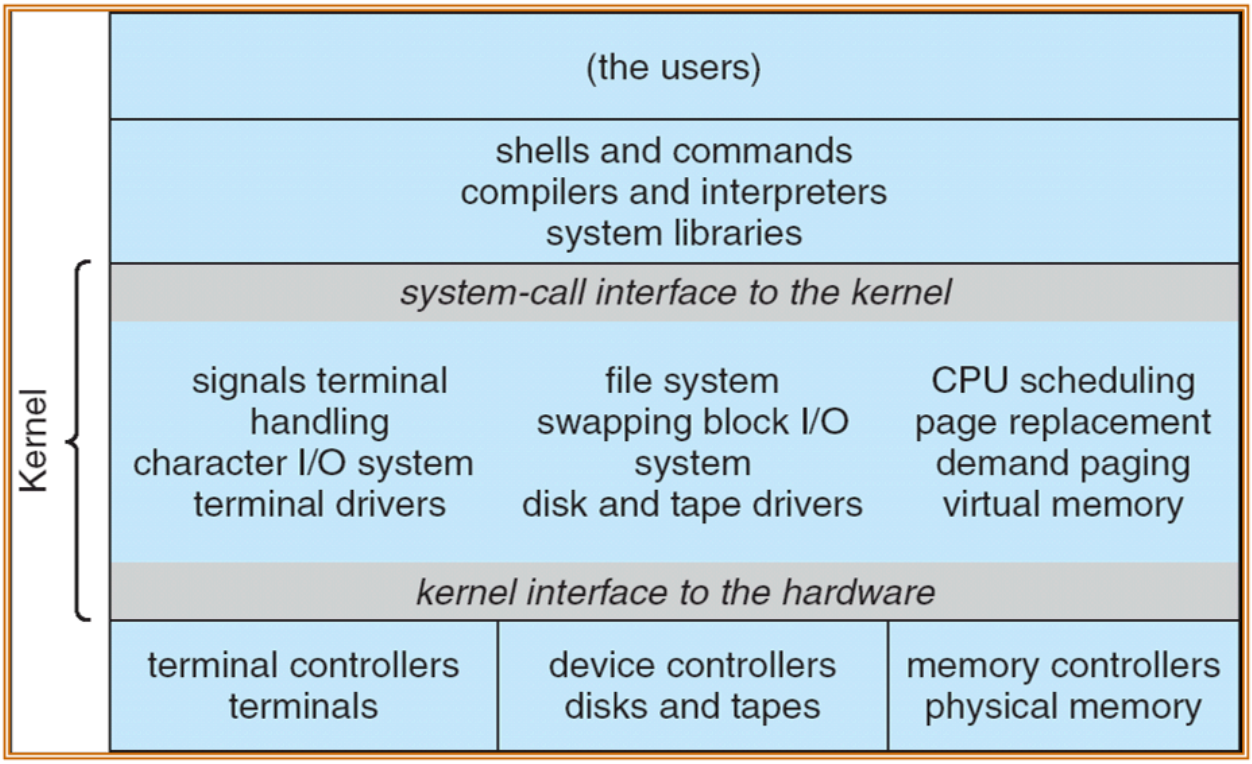

  - 由程序放置或推送到堆栈上的参数，由操作系统弹出堆栈

- 系统调用的类型

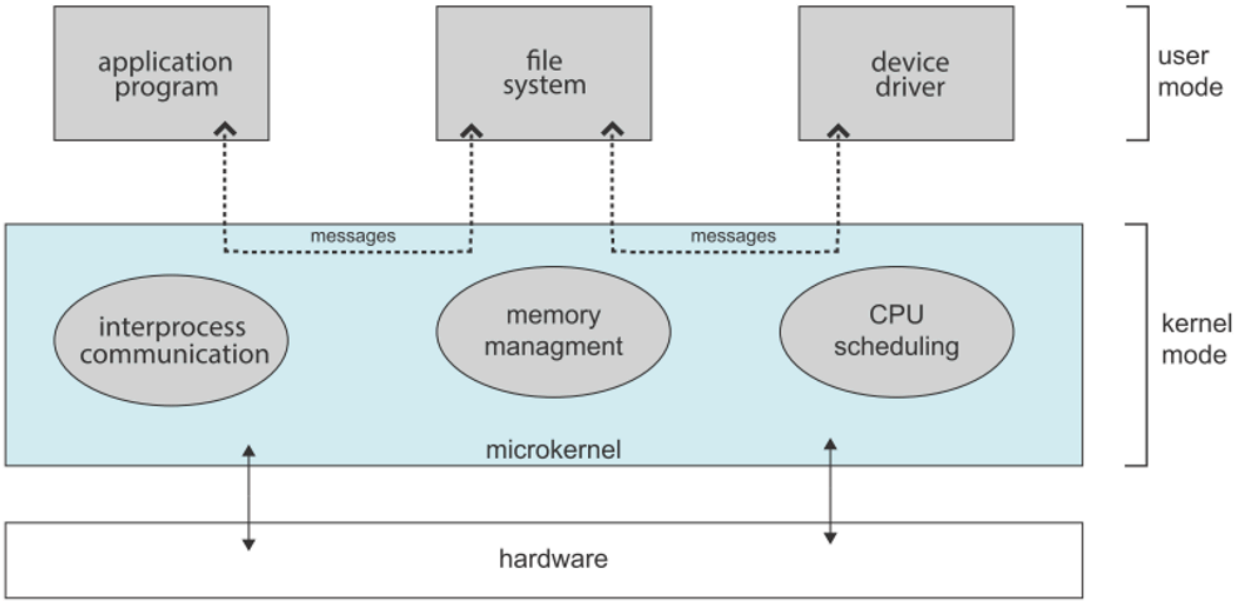
| | Windows | Unix |
|---|---|---|
| Process control | CreateProcess()<br>ExitProcess()<br>WaitForSingleObject() | fork()<br>exit()<br>wait() |
| File management | CreateFile()<br>ReadFile()<br>WriteFile()<br>CloseHandle() | open()<br>read()<br>write()<br>close() |
| Device management | SetConsoleMode()<br>ReadConsole()<br>WriteConsole() | ioctl()<br>read()<br>write() |
| Information maintenance | GetCurrentProcessID()<br>SetTimer()<br>Sleep() | getpid()<br>alarm()<br>sleep() |
| Communications | CreatePipe()<br>CreateFileMapping()<br>MapViewOfFile() | pipe()<br>shm_open()<br>mmap() |
| Protection | SetFileSecurity()<br>InitlializeSecurityDescriptor()<br>SetSecurityDescriptorGroup() | chmod()<br>umask()<br>chown() |

- 系统程序
  - 系统程序给程序开发和执行提供了便利的环境，如：文件管理、状态信息、文件修改、编程语言支持、程序加载和执行、交流、应用程序
- 操作系统的设计和实现
  - 先定义目标和愿景
  - 硬件、系统类型的选择影响操作系统的设计和实现
  - 用户目标和系统目标
  - **重要原则：分离**
    - 策略：确定具体做什么事
    - 机制：定义做事方法
  - 操作系统提供特定的机制作为工具，以满足不同的用户定义的策略
- 操作系统结构
  - 最简单的结构：MS-DOS
  - 整体结构：UNIX（Monolithic Structure）

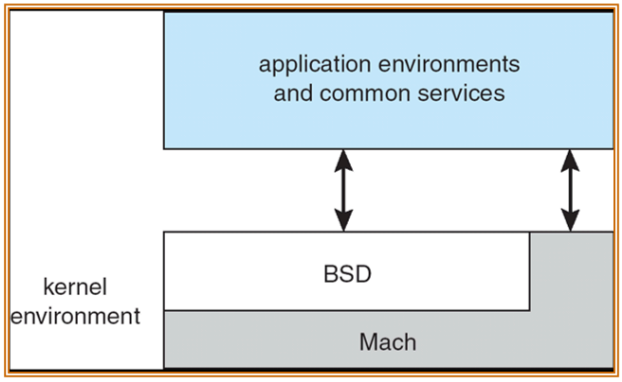
- 优点：直接的内核服务调用带来的高性能、上下文切换更少的简化的系统架构、因为所有东西都在一个空间所以开发和纠错很方便
- 缺点：内核中的一个bug可能导致整个系统的崩溃、服务几乎不独立带来的不安全性、在不影响整个系统的前提下更难维护和更新
- 系统程序
- 内核：包含了低于系统调用接口、高于物理硬件的所有内容

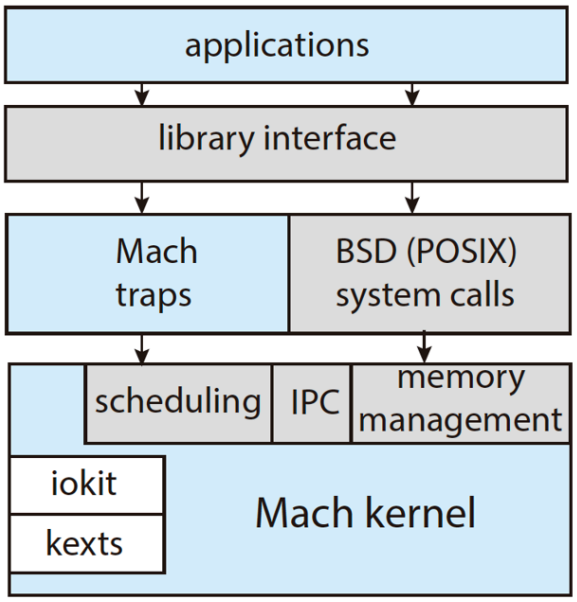



- 微内核系统结构
  - 作用：提供小部分关于进程调度或IPC的服务
  - 使用消息传递在用户模块之间进行通信
  - 优点：服务独立带来的更高的安全性、故障控制带来的系统稳定性的提升、增加或修改服务更加灵活
  - 挑战：用户空间到内核空间通信的性能开销、驱动开发和系统服务的复杂性、上下文切换带来的低效率

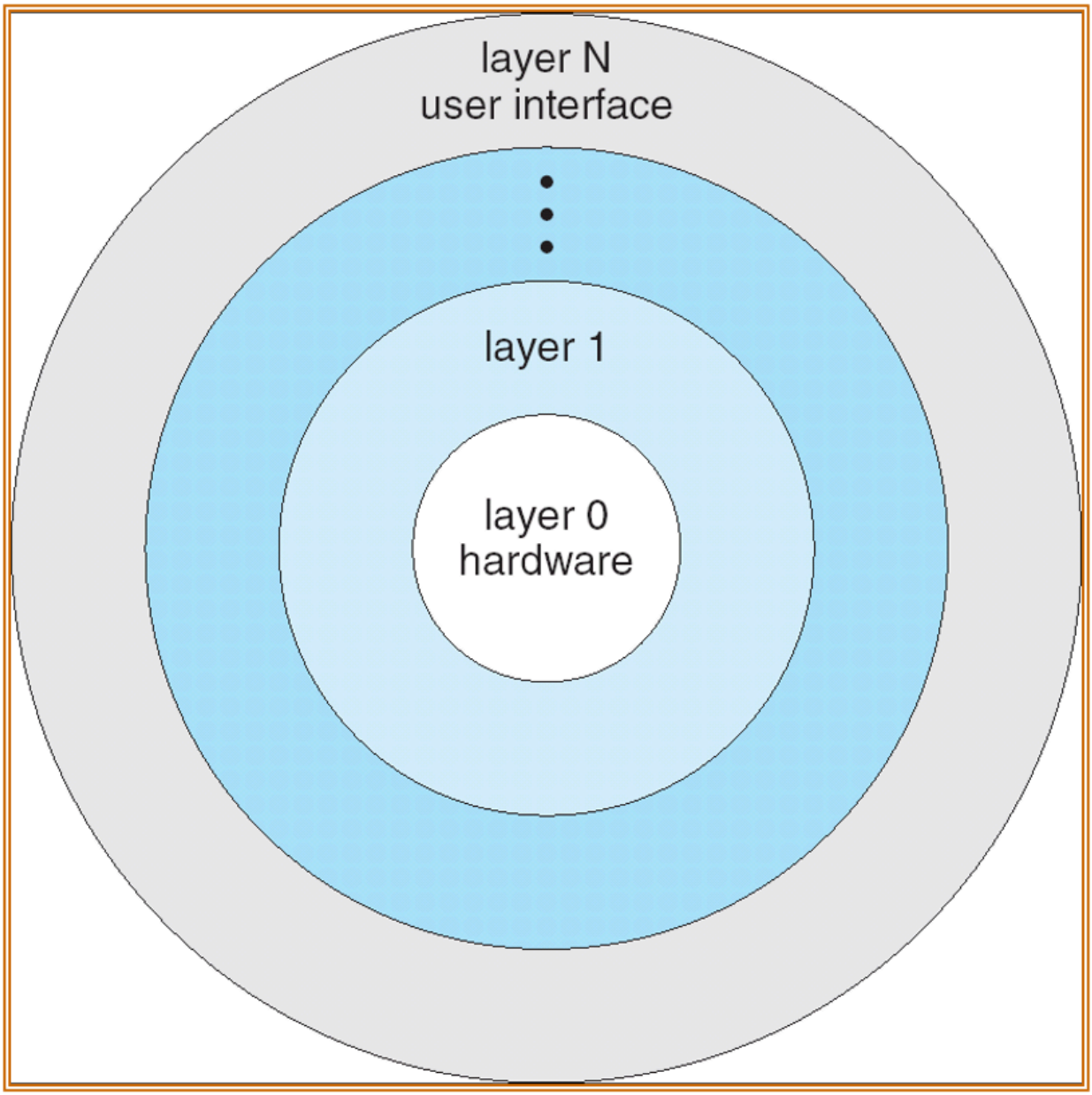



- 混合结构Darwin（Mac OS / iOS）
  - 在内核空间中运行核心功能，不那么重要的服务在用户空间运行

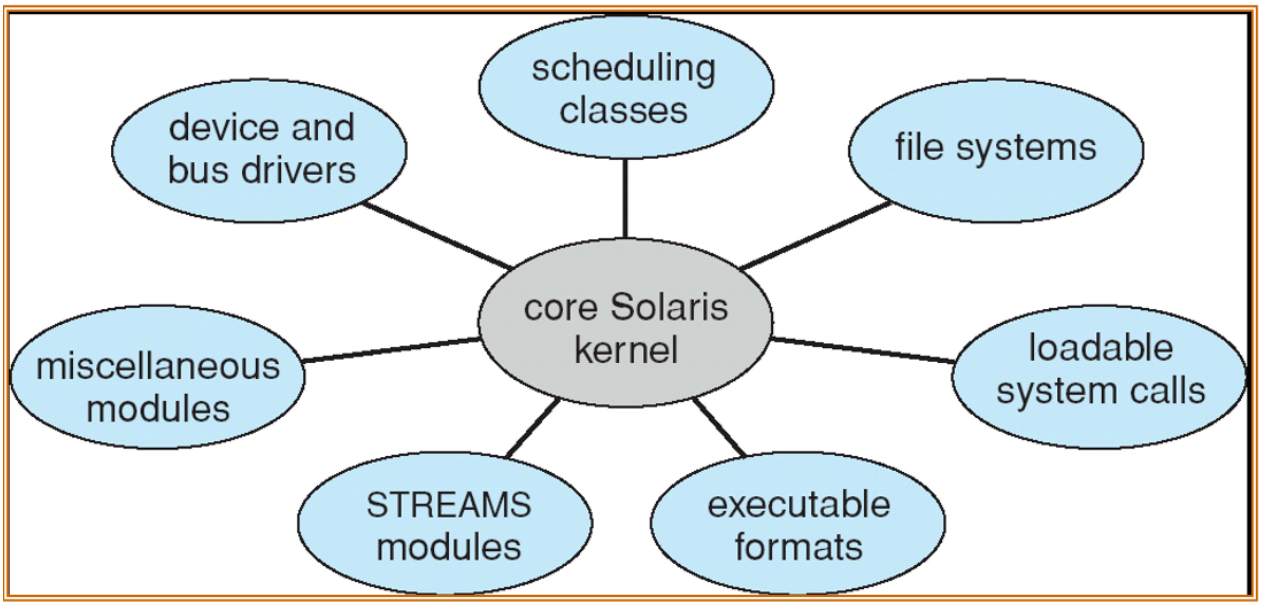


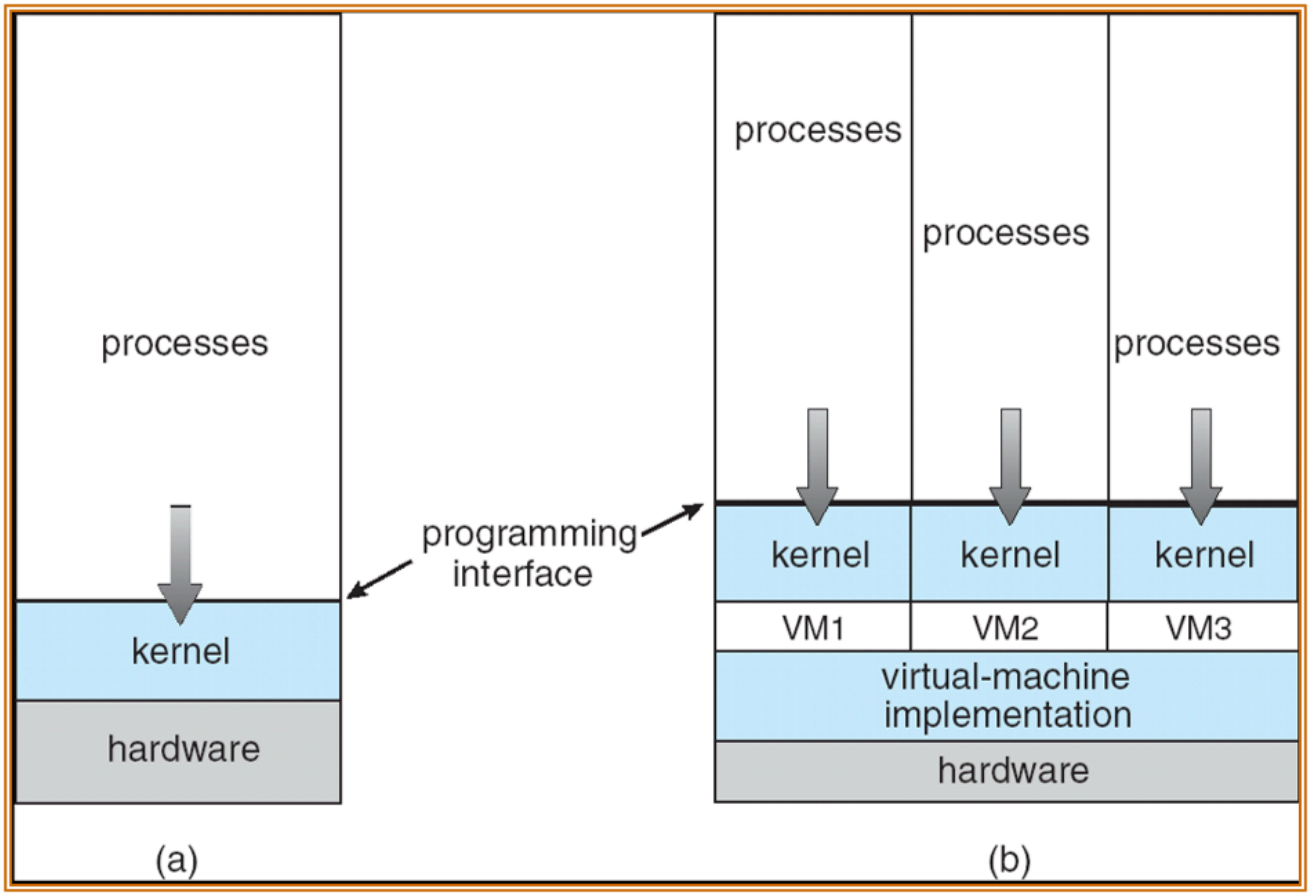

- 层级化结构
  - 操作系统被分为几层，高层基于低层构建
  - 优点：优化的组织和结构、提高可维护度、提高可测试性和可调试性
  - 缺点：性能问题，设计层级间的高效沟通非常的复杂（相对于模块化而言最大的缺点）




- 模块化
  - 用面向对象的方式
  - 核心组件是分离的
  - 组件之间的交流通过已知的接口
  - 优点：灵活度和可扩展度高，更新升级更便捷；每个模块都可以单独开发和测试；更优秀的资源管理和更好的性能

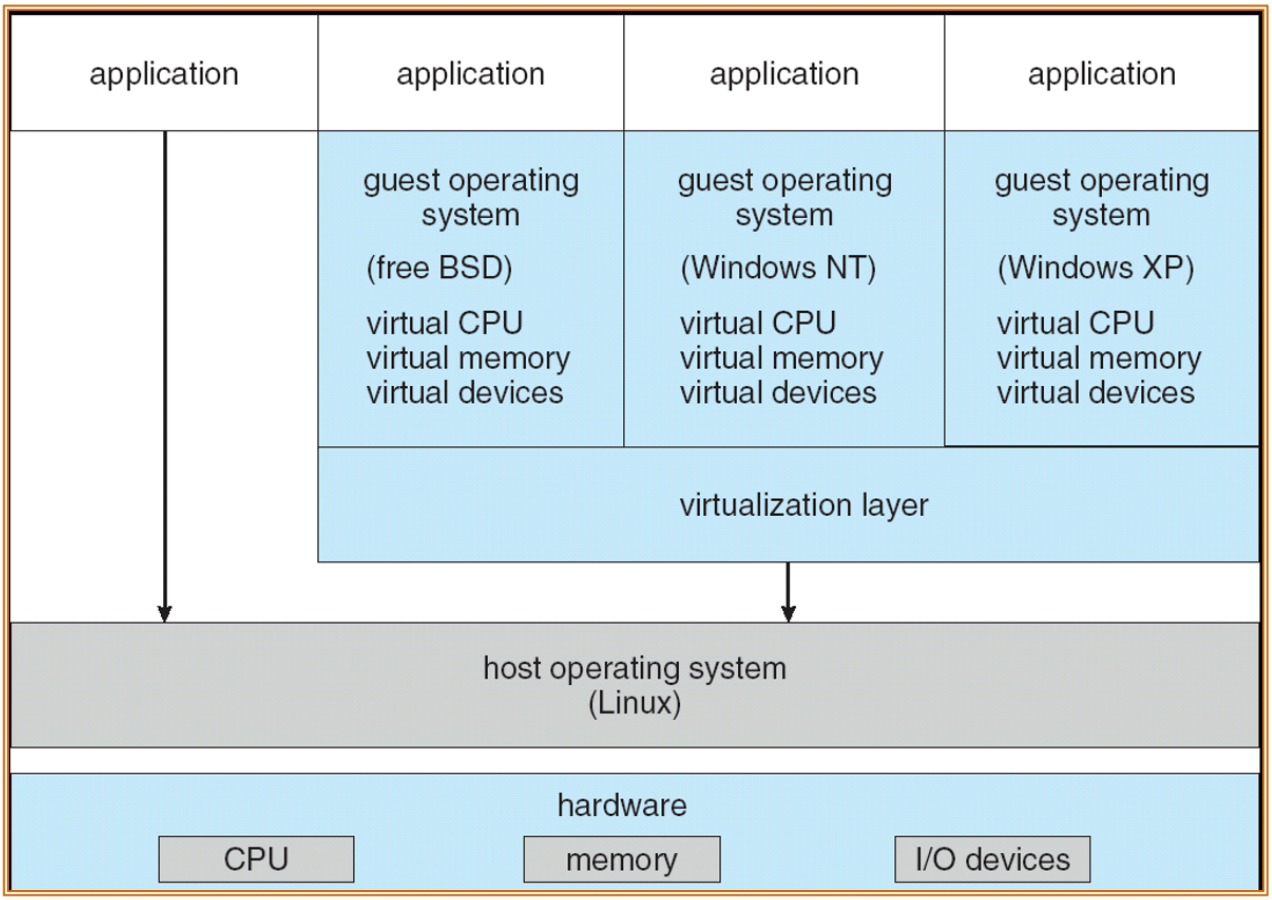



    - 外核
        - 只负责资源分配，提供了低级的硬件操作，必须通过定制library供应用使用
        - 为了提供定制化资源管理的高度灵活性
        - 高性能，定制化library难度大，兼容性差
- 虚拟机
    - Hypervisor从操作系统抽象硬件层
        - Type1
            - bare-metal，直接运行在宿主机硬件上，高效且提供更好的性能表现
            - 优点：高性能表现；直接访问硬件资源；更安全，虚拟机之间独立
        - Type2
            - hosted，运行在常规的操作系统上，易于启动，方便用户管理，但会引入额外的损耗
            - 优点：易于使用和安装；与宿主操作系统兼容；测试开发环境友好
    - 将硬件和操作系统内核视为硬件；提供与底层裸硬件相同的接口
    - 操作系统产生了多个进程的错觉，每个进程都在自己的处理器上执行，并拥有自己的（虚拟）内存
    - 共享物理计算机的资源以创建虚拟机
        - CPU调度可以让用户看起来有自己的处理器；spoling和文件系统可以提供虚拟读卡器和虚拟行式打印机；普通用户分时终端充当虚拟机操作员的控制台
    - 虚拟机概念提供了对系统资源的完全保护，因为每个虚拟机都与所有其他虚拟机隔离。然而，这种隔离不允许直接共享资源。虚拟机系统是操作系统研发的完美载体。系统开发是在虚拟机上完成的，而不是在物理机上，因此不会中断正常的系统操作。由于需要向底层机器提供精确的副本，虚拟机概念很难实现。（例如，虚拟用户模式和内核模式）



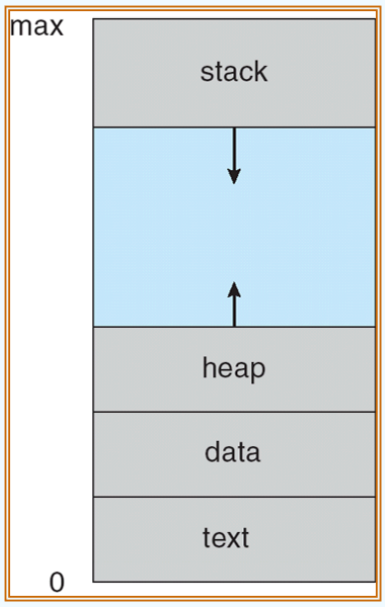

(a) Nonvirtual machine (b) virtual machine

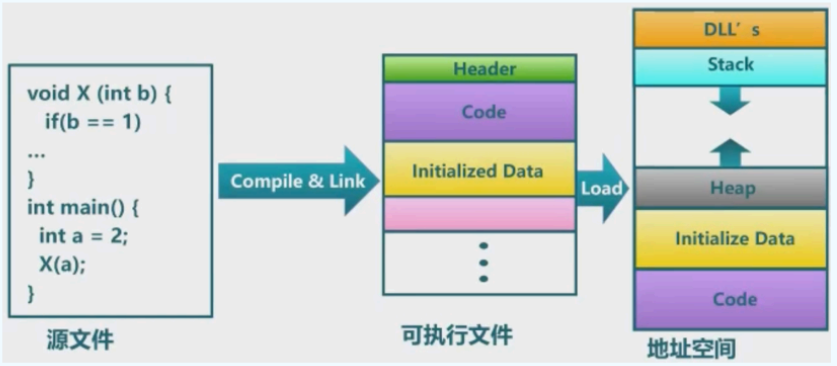
- 虚拟机架构



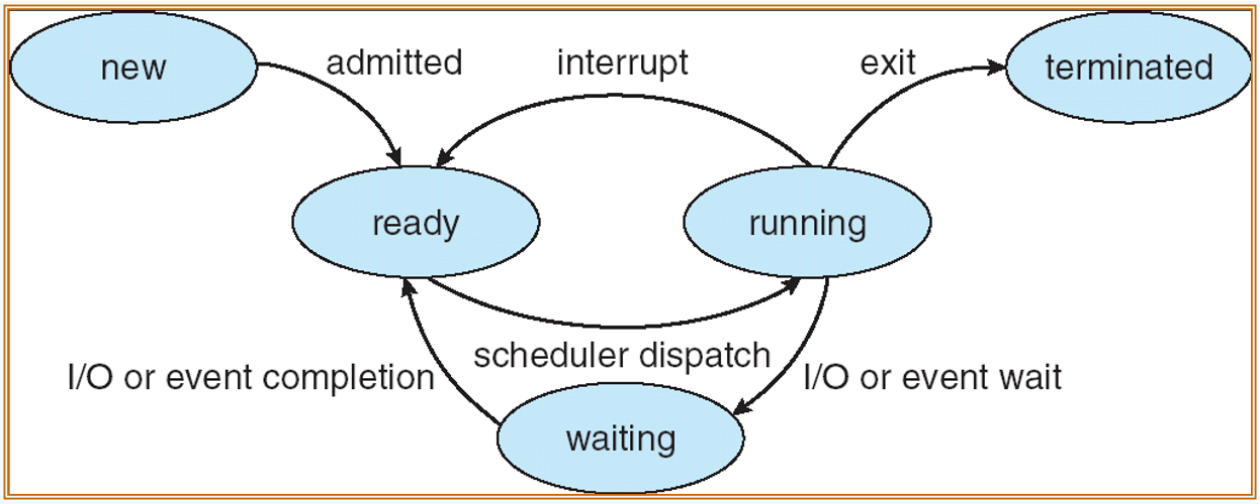

# CH3 Processes

- 进程概念
  - 正在执行的程序，包括：文本区（进程的可执行代码）、数据区（初始化的全局变量）、栈（函数参数、本地变量、返回地址）、堆（动态分配内存）、程序计数器
    - 栈和堆相对的目的：灵活使用内存空间

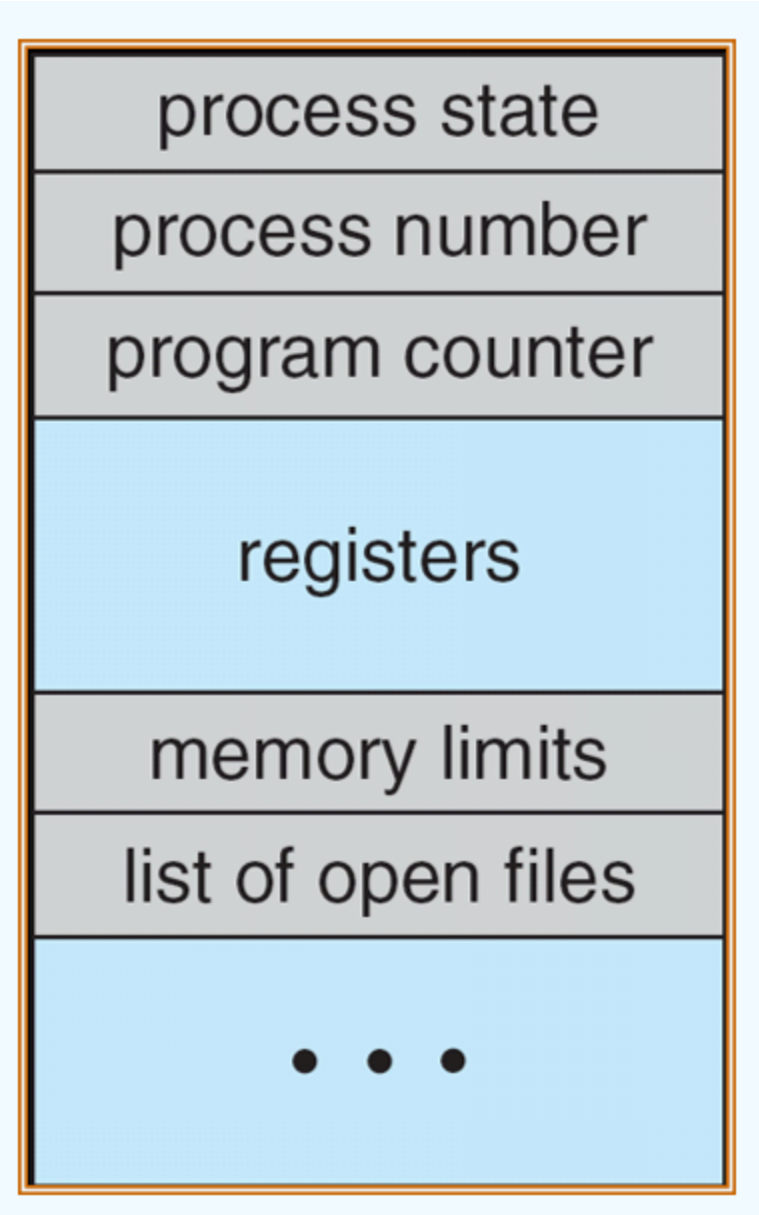


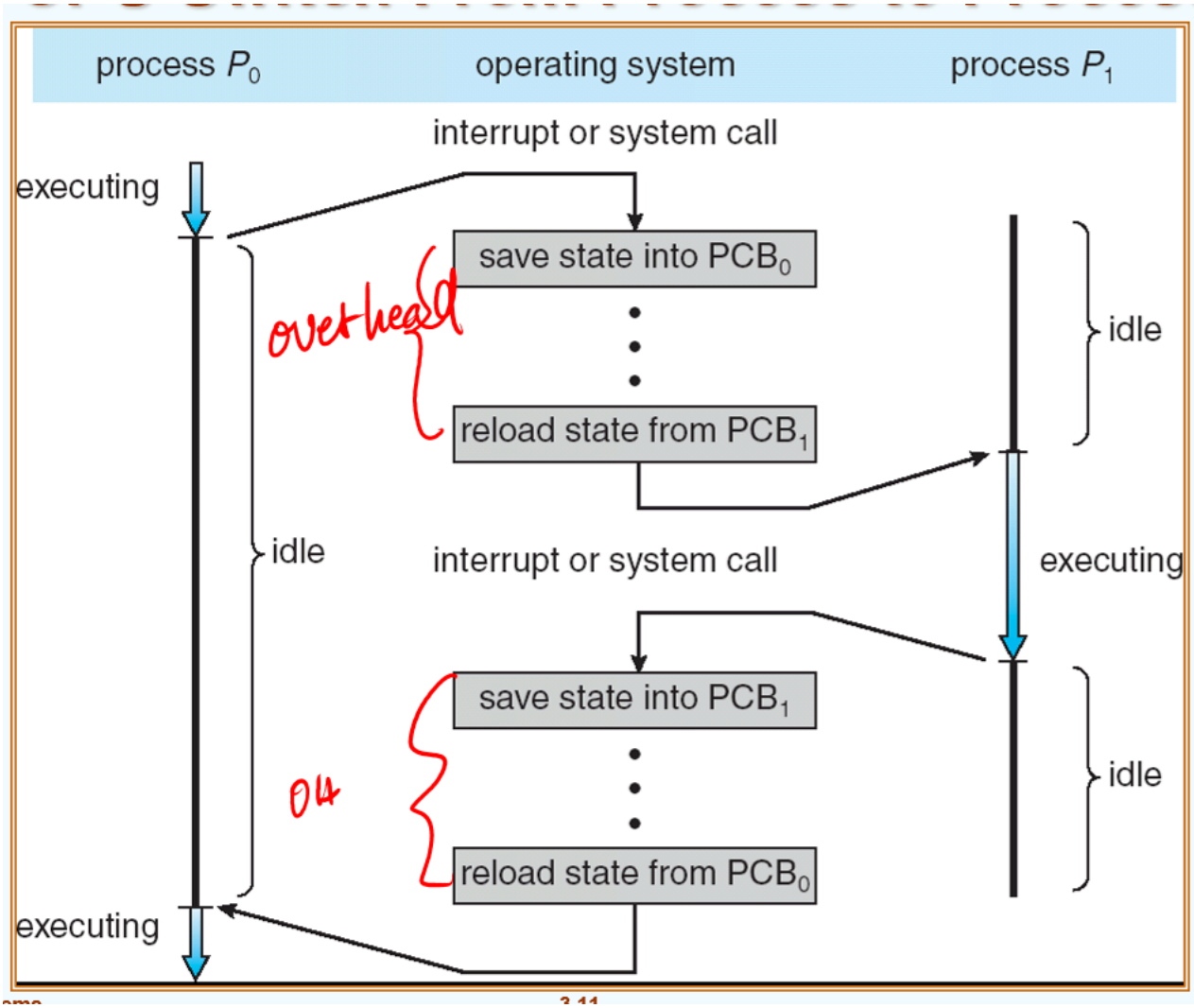
- 进程状态
  - new：正在被创建的进程
  - running：正在被执行的进程
  - ready：正在等待处理器分配的进程
  - waiting/blocked：正在等待某些事件发生的进程
  - terminated：已经执行完毕的进程（未被清除）



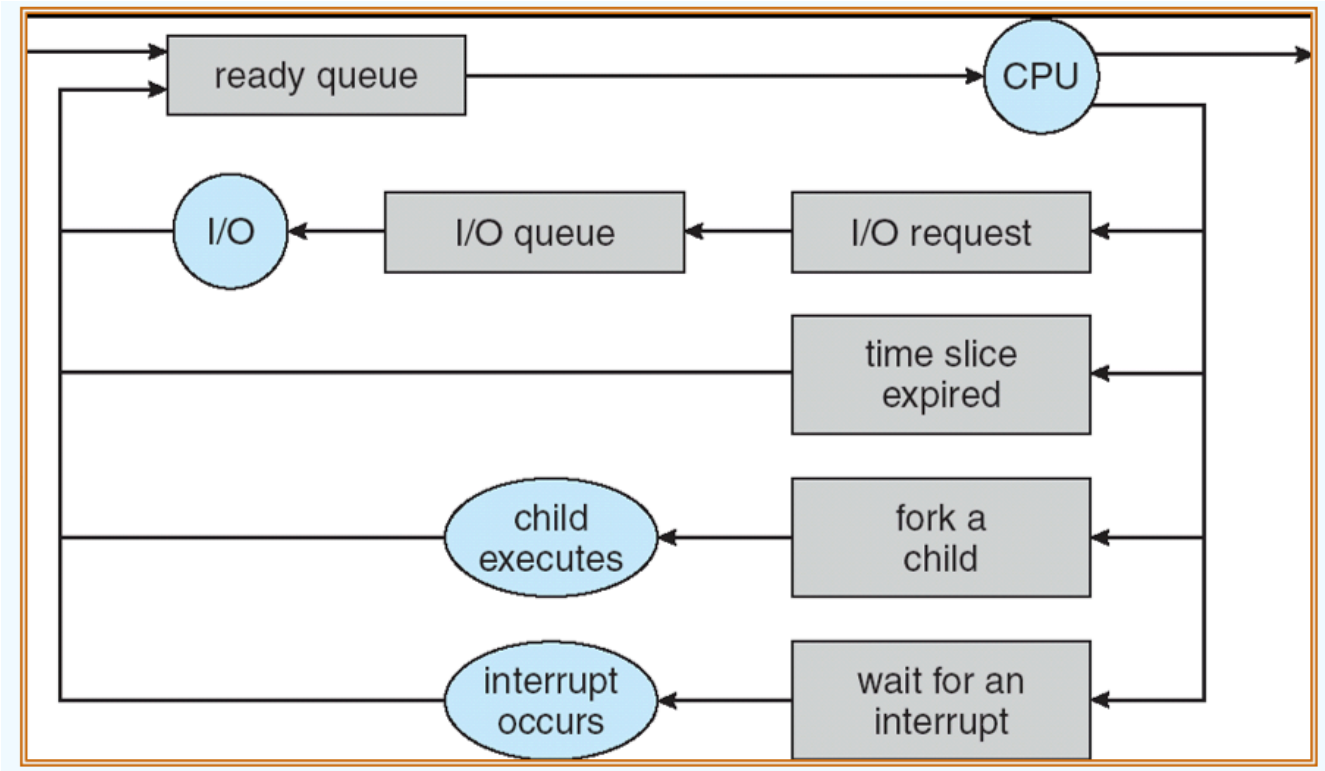

- **进程控制区块（PCB）**
  - 进程状态、程序计数器、CPU寄存器内容、CPU调度信息、内存管理信息、审计信息、I/O状态信息


process state
process number
program counter
registers
memory limits
list of open files

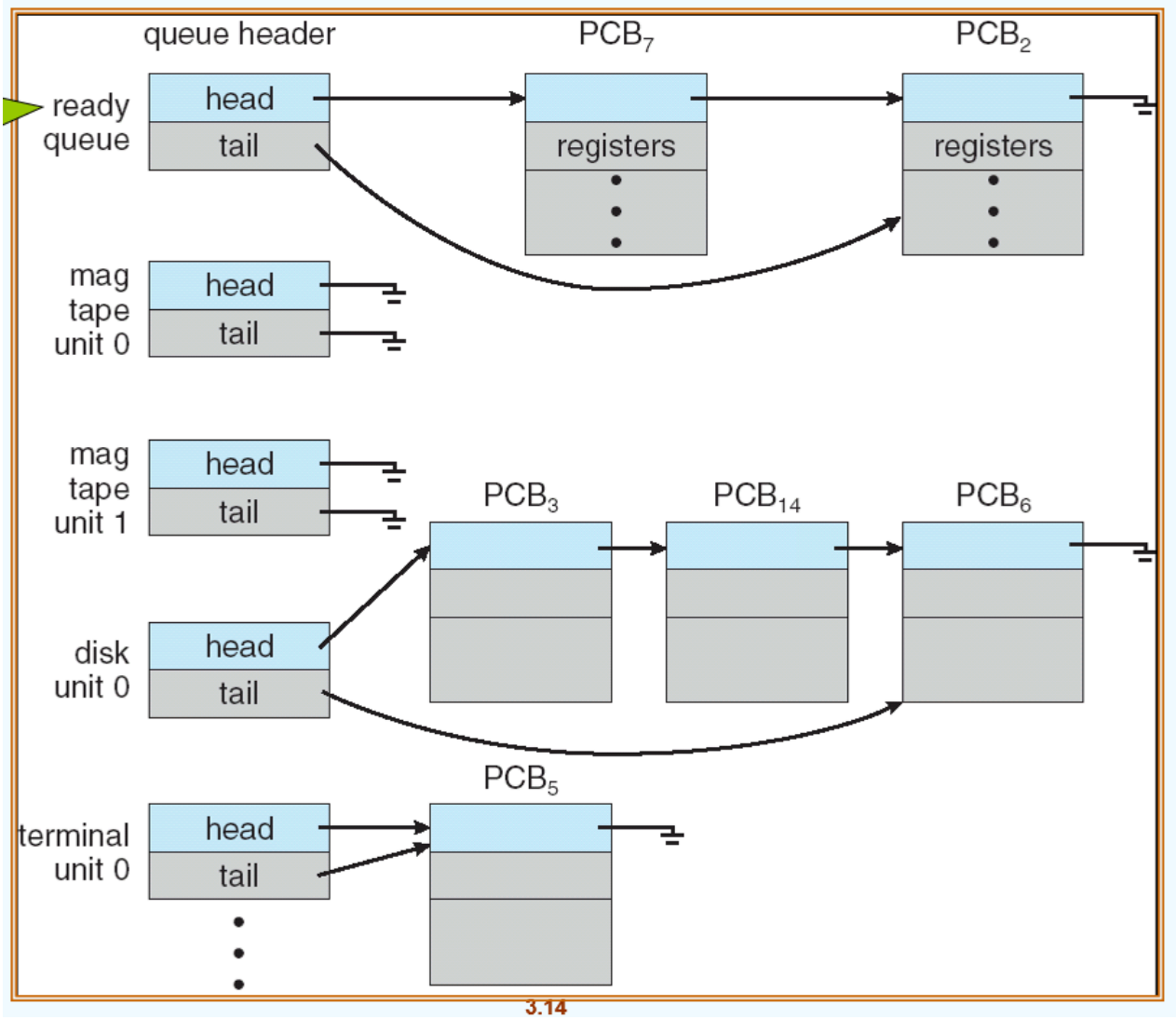
- 进程切换


process $P_0$
operating system
process $P_1$
interrupt or system call
executing
save state into $PCB_0$
overhead
reload state from $PCB_1$
idle
idle
interrupt or system call
executing
save state into $PCB_1$
04
reload state from $PCB_0$
idle
executing

- 进程调度
  - 进程调度队列

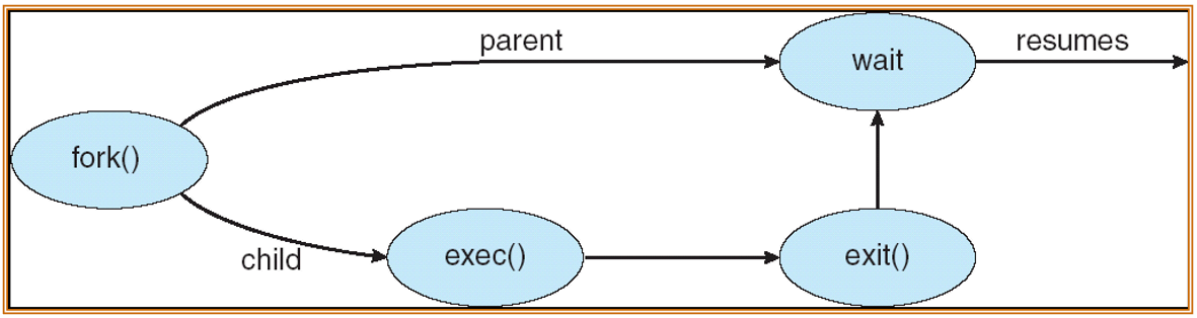

ready queue
CPU
I/O
I/O queue
I/O request
time slice expired
child executes
fork a child
interrupt occurs
wait for an interrupt

        - 进程队列Job queue：系统中的所有进程
        - 就绪队列Ready queue：所有进入内存等待执行的进程
        - 设备队列Device queues：所有等待I/O设备的进程




    - 调度器
        - Long-term调度器（job scheduler）：决定哪些进程应该被带进内存
        - Short-term调度器（cpu scheduler）：决定接下来哪个进程应该执行然后给它分配CPU
    - 上下文切换
        - 当CPU切换到其他进程时，系统必须保存老进程的状态并且加载新进程的保存状态
        - 上下文切换的时间有损耗
- 进程操作
    - 进程创建
        - 父进程创建子进程
        - 资源共享
            - 父进程和子进程资源共享
            - 子进程共享父进程资源的子集
            - 父子不共享资源
        - 执行

        - 父子进程并发执行
        - 父进程等待子进程结束
    - 地址空间
        - 子进程复制父进程的地址空间
        - 有一个程序加载到子进程中
    - UNIX 例子
        - fork系统调用创建新的进程
        - exec系统调用在fork之后，为了用一个新的程序替换进程的内存空间




  - 进程终止
- 进程合作
  - 独立进程无法影响或被正在执行的其他进程影响，而合作进程可以
  - **优势**
    - 信息共享、计算加速、模块性、便利性
  - 生产-消费范式（主要问题：并发进程之间的同步）
    - 生产者进程产生信息被消费者进程消费
    - 无界缓冲器：生产者不需要关注buffer满没满，消费者需要关注buffer空不空
    - 有界缓冲器：生产者需要关注buffer满没满

Producer pseudo-code:

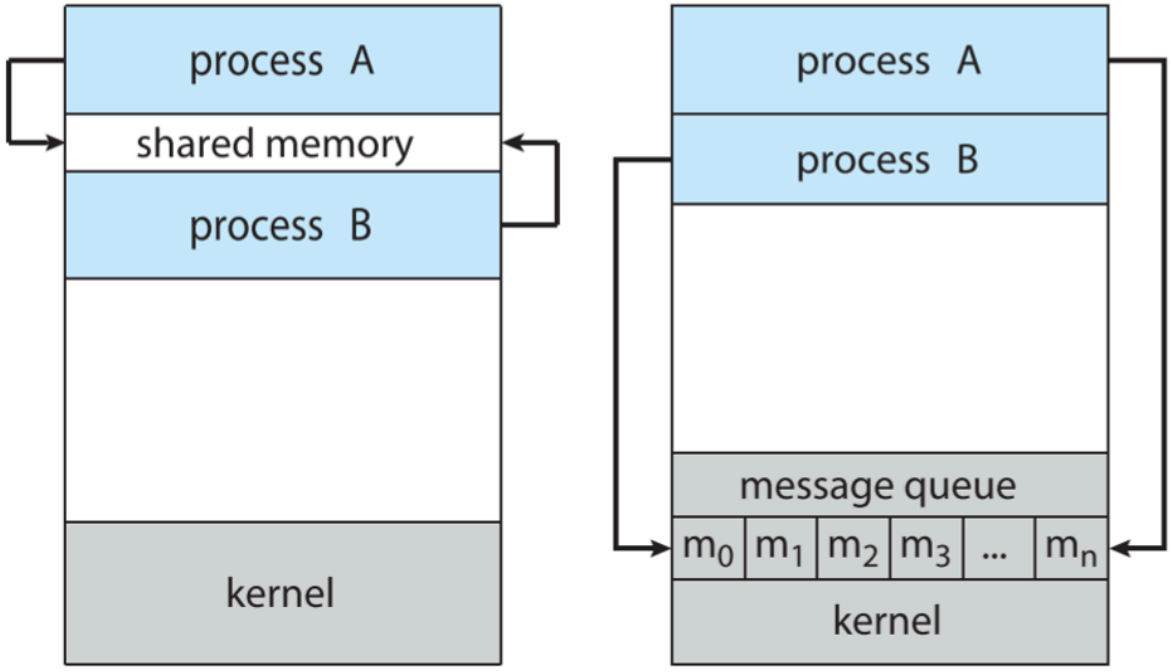
```
while (true) {
        Produce an item;

    while (((in + 1) % BUFFER_SIZE   == out)
          ; /* do nothing -- no free buffers */
        buffer[in] = item;
        in = (in + 1) % BUFFER_SIZE;
 }
```

Consumer pseudo-code:

```
while (true) {
    while (in == out)
      ; //do nothing, nothing to consume

      Remove an item from the buffer;
      item = buffer[out];
      out = (out + 1) % BUFFER SIZE;
      return item;
  }
```

Solution is correct, but can only use BUFFER_SIZE-1 elements

- 进程间通讯
  - 用于进程通信和同步其操作的机制
  - IPC两种方法：消息传递、共享内存
    - 消息传递：进程之间不借助共享变量进行通信
      - 两种操作：send、receive
      - 通信过程：建立通信连接、通过send/receive交换信息
      - 优势：数据一致性
    - 共享内存
      - 优势：访问共享信息更容易、overhead更少、更快




Shared memory　　Message passing

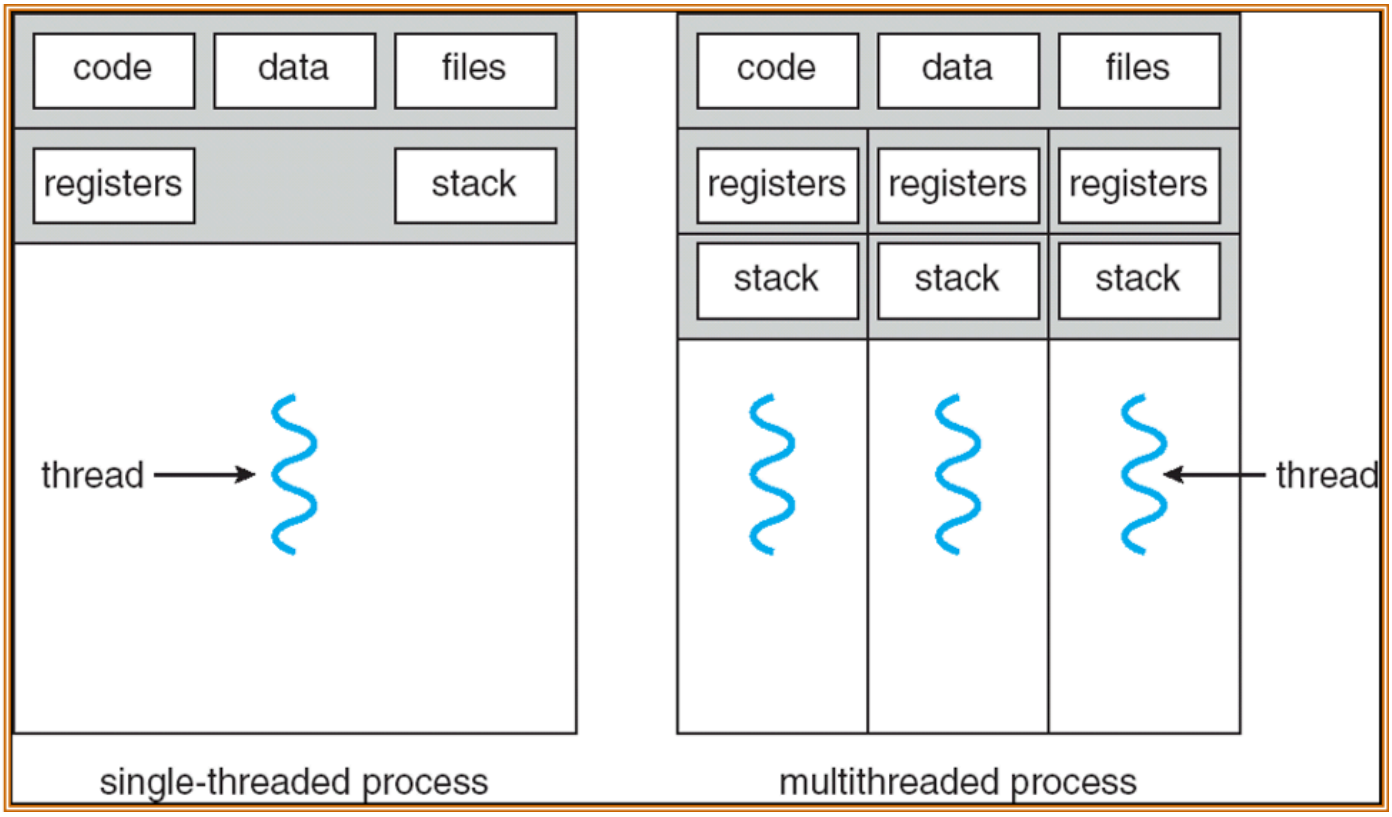
- 直接通信
  - 进程必须显式的标记对方的名字
  - 通信连接特性：
    - 连接自动建立
    - 连接与特定的一对通信进程相关
    - 每对只有特定的一个连接
- 间接通信
  - 消息从mailbox（端口）检测和接收
  - 每个mailbox有一个特定的id
  - 共享一个mailbox的进程可以相互通信
  - 通信连接特性：
    - 只有当进程共享一个mailbox时连接建立
    - 一个连接可能与很多进程有关
    - 每对进程可能共享多个通信连接
    - 单向或双向连接
  - 操作：
    - 创建一个新的mailbox
    - 收发消息通过mailbox
    - 销毁mailbox

- 同步

- 消息传递可能是blocking或non-blocking
  - blocking是同步的
    - 消息发送者blocked直到消息被接收，消息接收者被blocked直到有消息可以接收
  - non-blocking是异步的
    - 和同步相反

# CH4 Threads

- 单线程进程和多线程进程




  - 优势：
    - 响应性：交互性强的应用
    - 资源共享：代码和数据内存可以共享
    - 经济性：创建进程成本更高
    - 对多处理器架构的利用：多线程增加并发
- 并发(Concurrency)和并行(Parallelism)

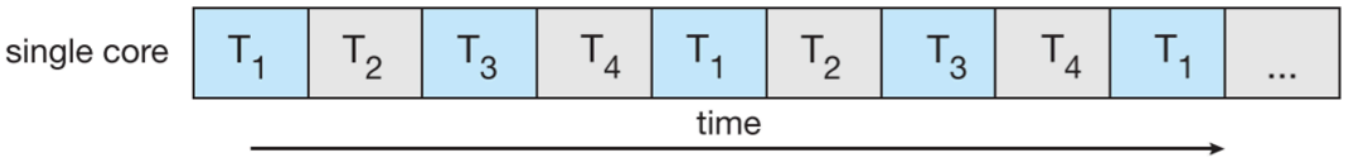



**Figure 4.3** Concurrent execution on a single-core system.

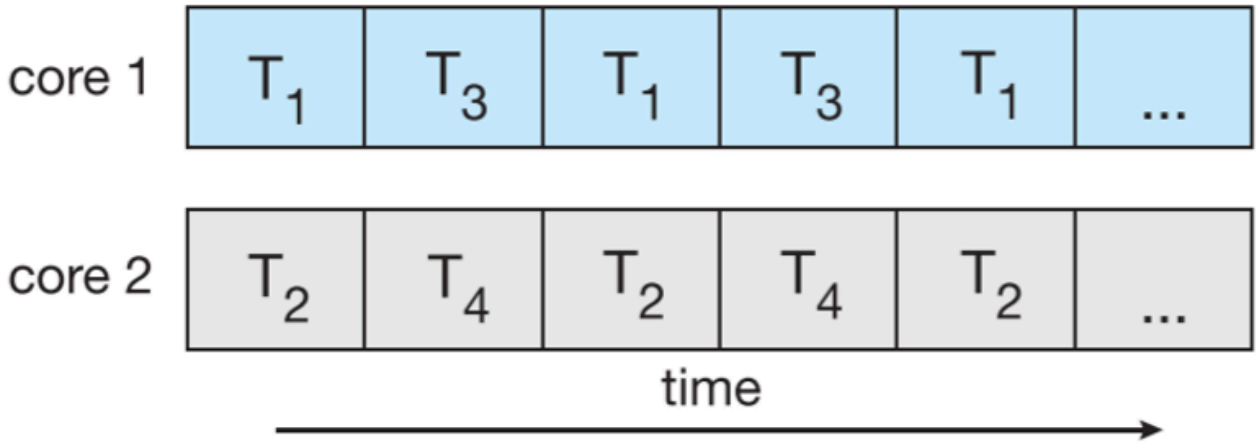
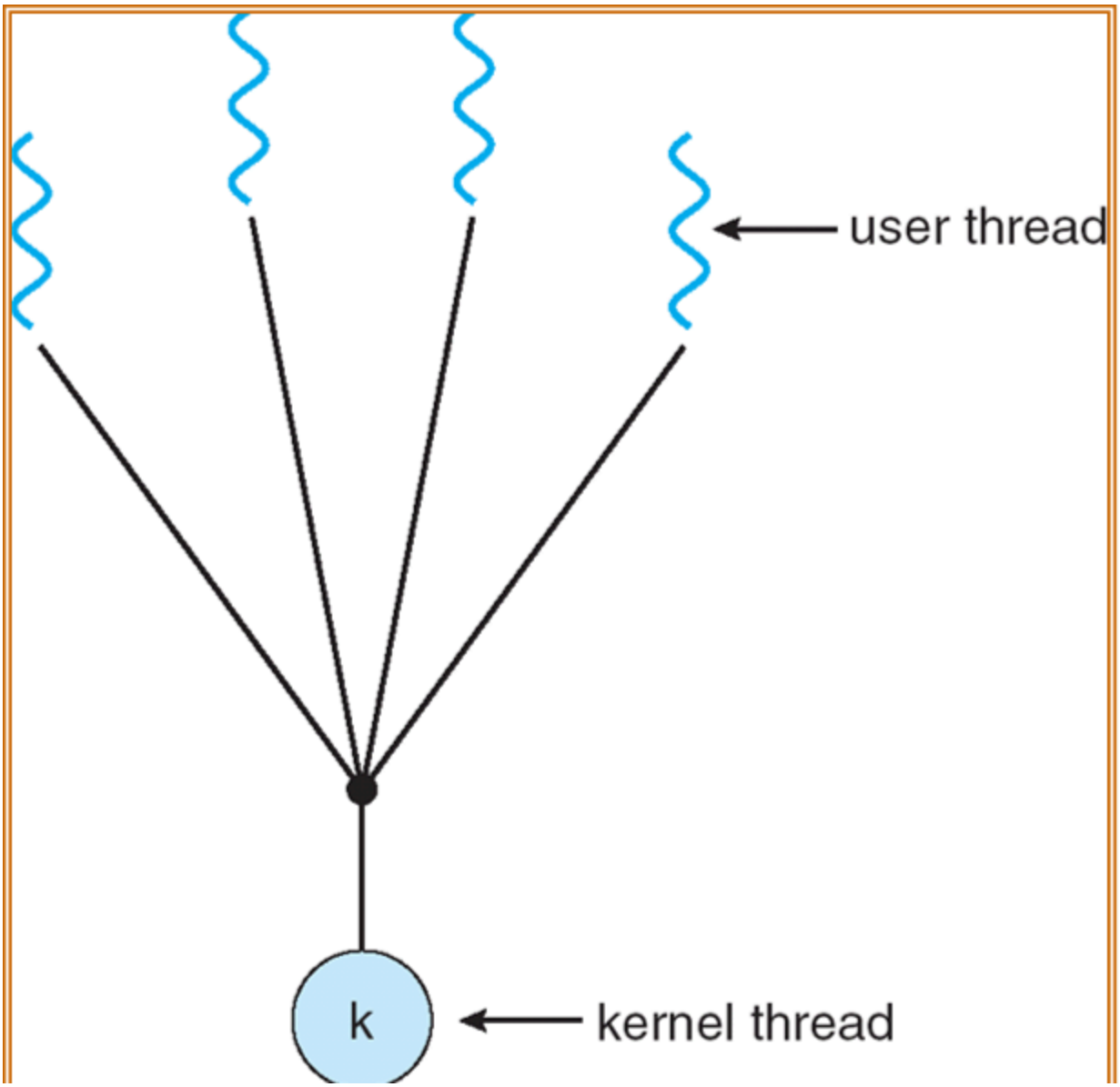
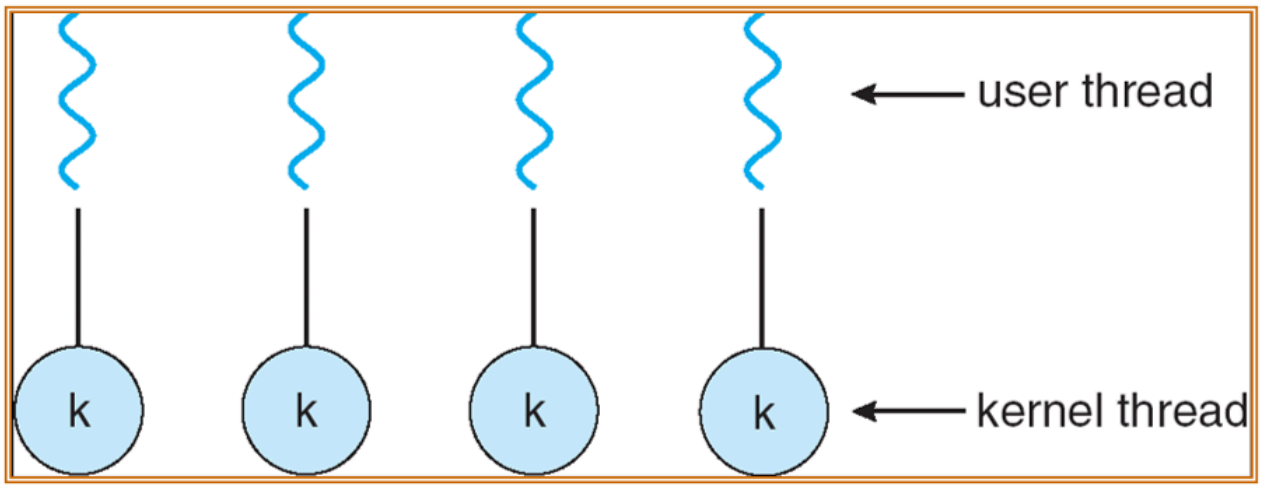


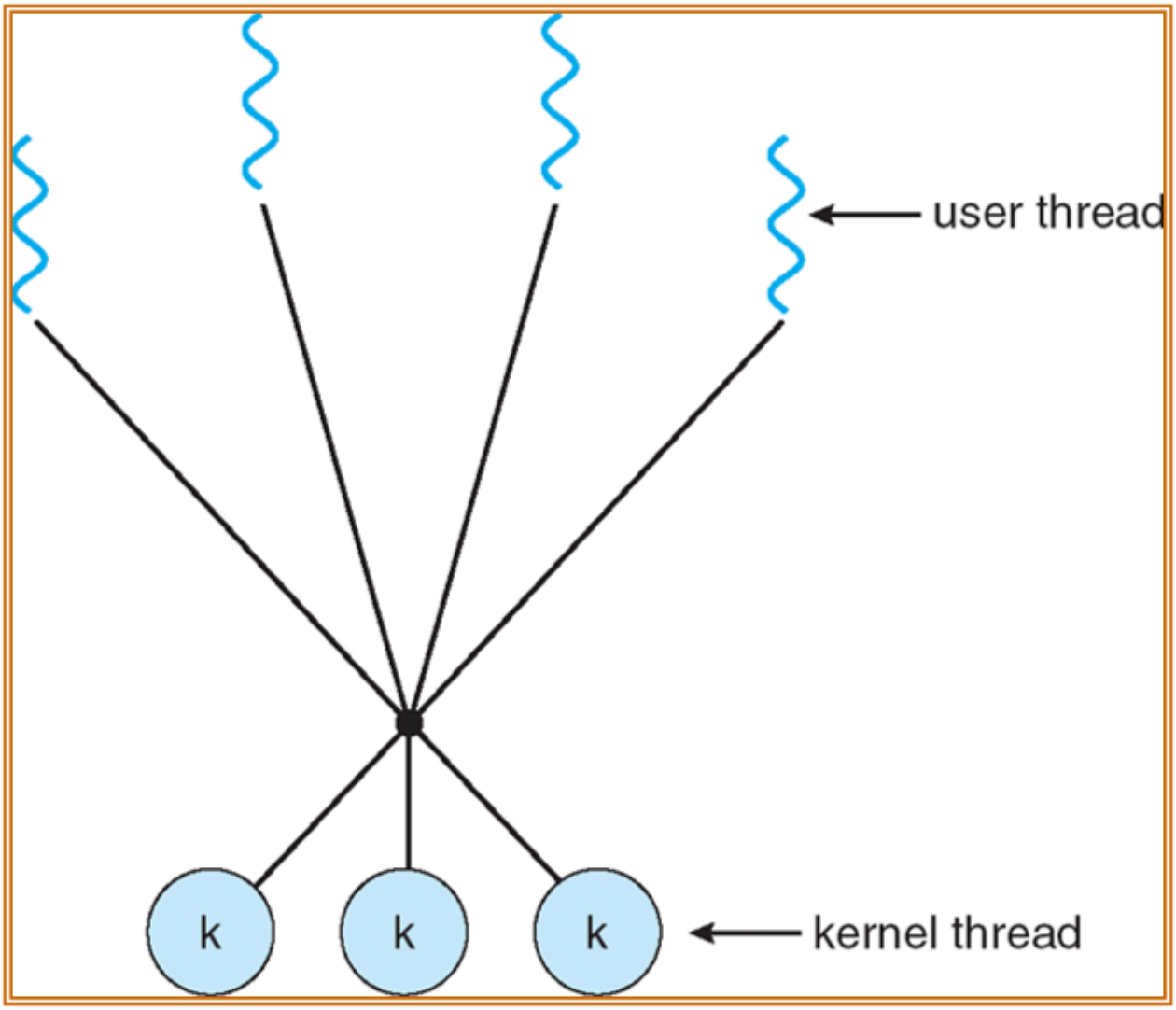

**Figure 4.4** Parallel execution on a multicore system.

- 用户线程：线程管理由用户级线程库进行
  - 三个主要的线程库：POSIX Pthreads、Win32 threads、Java threads
- 内核线程：由内核支持的线程
  - Windows XP/2000、Solaris、Linux、Tru64 UNIX、Mac OS X
- 多线程模型
  - Many-to-One
    - 多个用户级线程映射到一个内核级线程
    - 优劣：线程管理高效；但执行系统调用时会造成阻塞，内核一次只能调度一个线程

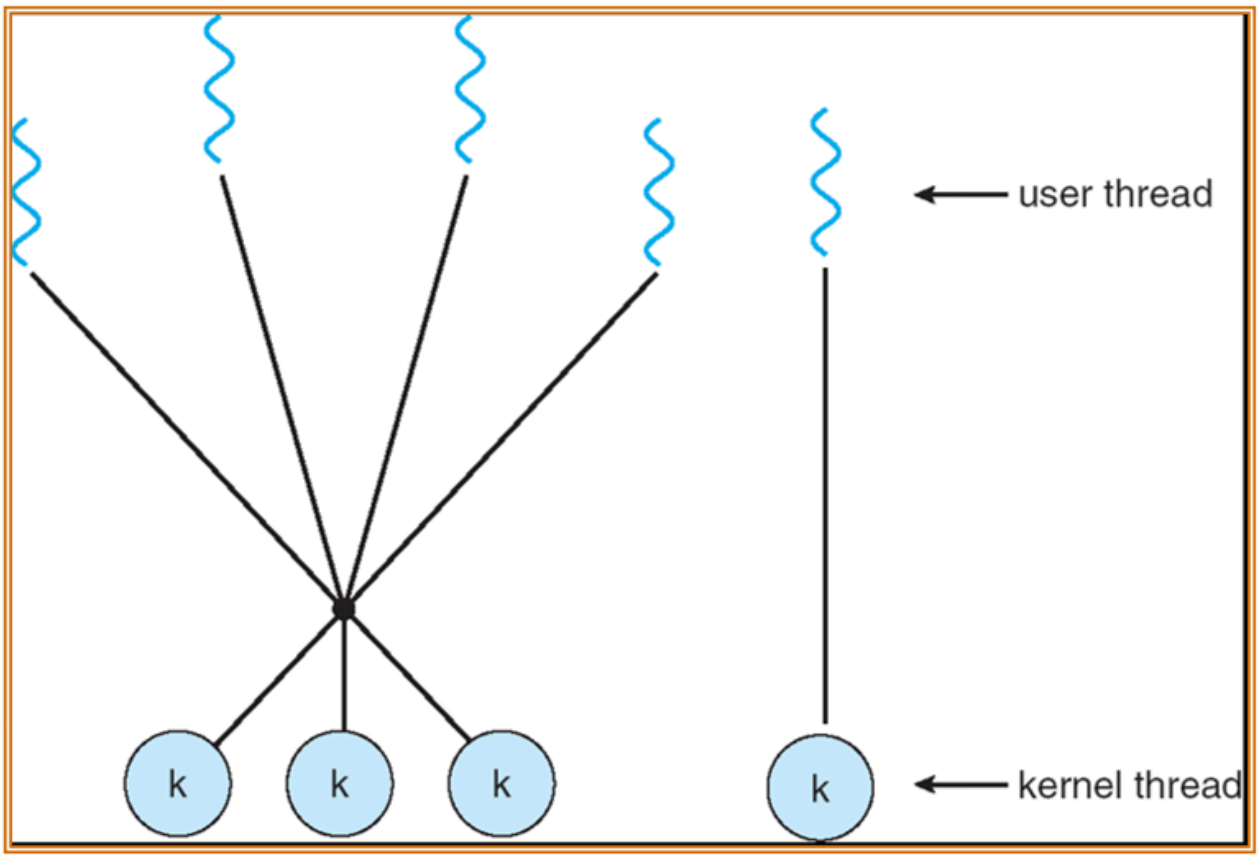
- 


- One-to-One
  - 一个用户级线程映射到一个内核级线程
  - 优劣：更高的并发性；但创建线程成本较高
  - 


- Many-to-Many
  - 允许多个用户级线程映射到多个内核级线程

- 允许操作系统创建足够数量的内核级线程
- 优势：更灵活
- 


- Two-level module
  - 允许多个用户级线程映射到多个内核级线程，同时也允许一个用户级线程映射到一个单独的内核级线程

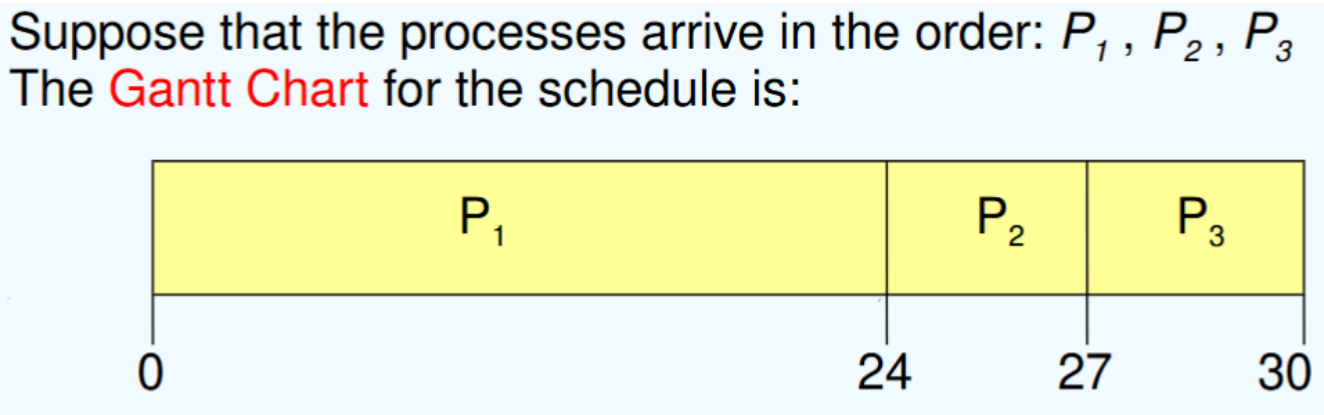
- 


- 线程问题
  - Semantics of fork() and exec() system calls
    - fork() 复制调用fork()的线程的信息还是所有线程的信息?
    - exec() 将替换整个进程
  - Thread cancellation
    - 异步取消：立即终止目标线程
    - 延迟取消：允许目标线程通过检查一个标志来判断是否需要取消
  - Signal handling
    - 在类UNIX系统中，信号被用来通知进程一个特定事件的发生
    - signal handler用来处理信号，有同步和异步方式
      - 由特定事件产生信号->信号被发送至进程->信号被处理
    - 根据信号的不同类型有不同的发送方式
      - 发送至应用信号的线程
      - 发送至所有线程
      - 发送至特定线程
      - 指定一个线程接收进程的所有信号
  - Thread pools
    - 在一个池中创建几个线程，线程在池中等待工作
    - 优点：通常，用现有进程处理请求比创建新进程更快一些；允许应用中存在的线程数被限制在线程池大小
  - Thread-specific data

        - 允许每个线程都有其自己的数据副本，是每个线程特有的，在你无法控制线程创建过程是非常有用
    - Scheduler activations
        - M2M和Two-level Model都需要通过交流来维持分配给应用的正确数量的内核级线程

# CH5 CPU scheduling（10.16课 有总结）

- 基本概念
    - 目标最大化CPU的使用
    - CPU-I/O Burst Circle——进程执行包括CPU执行和I/O等待的一个循环
- CPU调度器
    - 从内存中选择就绪态的进程，并为他们分配CPU
    - 抢占式
        - 当进程从执行态转换为就绪态时
        - 当进程从等待（阻塞）态转换到就绪态时
    - 非抢占式
        - 当进程从执行态转换为等待（阻塞）态时
        - 当进程终止时
- Dispatcher
    - Dispatcher module
    - Dispatcher latency 调度延迟
- Scheduling Criteria 调度准则
    - CPU利用率：$CPU执行任务的时间/总时间 \times 100\%$
    - Throughput吞吐率：$完成进程数/总时间$
    - 周转时间：
    - 等待时间：在ready queue中等待CPU的时间
    - 响应时间：
- 调度算法
    - FCFS Scheduling（先来先服务）（非抢占式）




        - 优点：简单、公平

- 缺点：
  - 护航效应（Convoy effect）{短进程在长进程之后，平均等待时间就变长，导致I/O设备或CPU闲置}；
  - 有利于长作业，不利于短作业；有利于CPU繁忙型，不利于I/O繁忙型

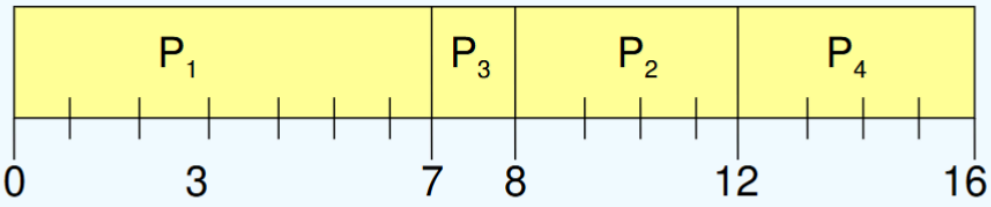
- SJF Scheduling（短作业优先）
  - 以进程下一次CPU burst的时间长短进行排序，时间最短的先执行
    - 如何确定下一次CPU burst的时间长度：指数平均法

1. $t_n$ = actual length of $n^{th}$ CPU burst
2. $\tau_{n+1}$ = predicted value for the next CPU burst
3. $\alpha, 0 \le \alpha \le 1$
4. Define: $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$.

  - 两种模式
    - 非抢占式：CPU分配给一个进程后，进程可以完全执行其CPU burst（注意arrival time）

| Process | Arrival Time | Burst Time |
|---|---|---|
| $P_1$ | 0.0 | 7 |
| $P_2$ | 2.0 | 4 |
| $P_3$ | 4.0 | 1 |
| $P_4$ | 5.0 | 4 |

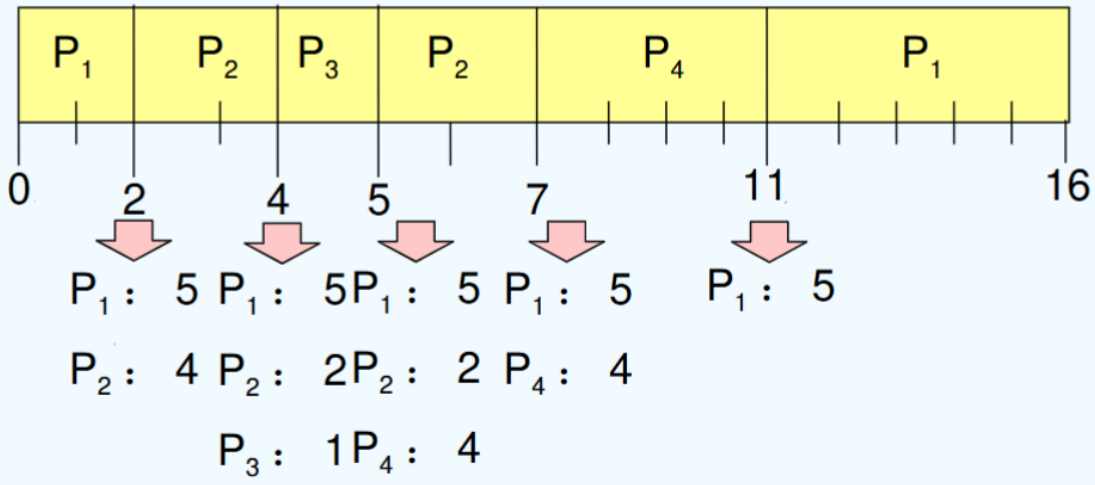
SJF (non-preemptive)




    - 抢占式（SRTF）：根据当前所有进程的剩余运行时间进行排序，如果当前运行进程的remaining time大于某个进程的CPU burst，则进行一次调度

| Process | Arrival Time | Burst Time |
|---|---|---|
| $P_1$ | 0.0 | 7 |
| $P_2$ | 2.0 | 4 |
| $P_3$ | 4.0 | 1 |
| $P_4$ | 5.0 | 4 |

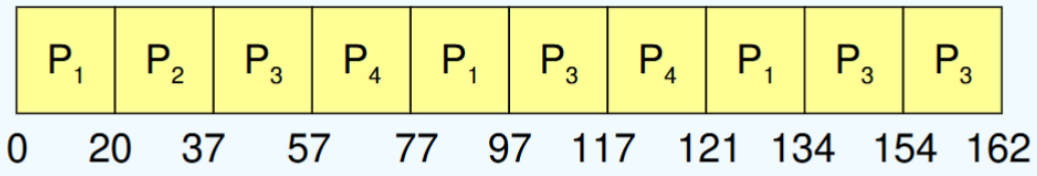
- SJF (preemptive)




- HRRN（高响应比优先）
  - 响应比：$(waiting\ time + CPU\ burst)/CPU\ burst$
  - 非抢占式；FCFS和SJF的折中；提高整个系统的响应性
- RR Round Robin（时间片轮转调度算法）
  - example：Time Quantum = 20

| Process | Burst Time |
|---|---|
| $P_1$ | 53 |
| $P_2$ | 17 |
| $P_3$ | 68 |
| $P_4$ | 24 |

- The Gantt chart is:




  - 优点：响应性好
  - 时间片大小的设置

        - 太大 -> FCFS
        - 太小 -> overhead变高
    - 应用场景：分时系统、多任务系统
  - 多级队列
    - ready queue 分成不同的队列：foreground（interactive）、background（batch/CPU）
    - 每个队列用不同的调度算法：foreground-RR、background-FCFS
    - 不同队列间也需要调度
  - 多级反馈队列
    - 进程可以在不同的队列中进行相应的移动
    - 例子：
      - 设置多个就绪队列，优先级从第一级依次降低
      - 优先级高的队列，进程时间片越短
      - 每个队列都采用FCFS，若在该时间片完成，则撤离系统，未完成，转入下一级级队列
      - 按队列优先级调度，仅当上一级为空时，才运行下一级
  - 多处理器调度
    - 非对称多处理器调度
    - 对称多处理器调度
  - 实时调度
    - 硬实时系统
    - 软实时系统
    - 最早截止时间优先 Earliest Deadline First
    - 最低松弛度优先 Least Laxity First
      - A松弛度=A必须完成的时刻-A的运行时间-当前时刻
    - 速率单调调度 Rate Monotonic Scheduling
      - 基于任务周期来分配优先级，周期越短，优先级越高
- 线程调度
  - Local
  - Global
  - Many-to-one
  - One-to-one

# CH6 Process Synchronization

- 背景
  - 为什么要有进程同步机制：并发访问共享数据时可能会导致数据不一致性
  - 解决方法（进程同步定义）：进程同步是指在多进程或多线程环境中，为了确保多个进程或线程能够按照一定的顺序或条件正确地访问共享资源而采取的各种协调机制

- 协作进程：能被其他进程的执行所影响的进程
- 生产者-消费者问题
  - 通过in和out指针计算item的数量：$(in - out)\%BUFFER\_SIZE$

```
while (true) {
    Produce an item;
    while ((in + 1) % BUFFER_SIZE
      == out)
        ; /* do nothing -- no free
    buffers */
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
 }
```

```
while (true) {
      while (in == out)
        ; //do nothing, nothing to consume
        Remove an item from the buffer;
        item = buffer[out];
        out = (out + 1) % BUFFER SIZE;
        return item;
  }
```

  - 通过一个额外的变量count来统计

```
while (true) {
/*  produce an item and put in
     nextProduced  */
          while (count == BUFFER_SIZE)
                ; // do nothing
           buffer [in] = nextProduced;
           in = (in + 1) % BUFFER_SIZE;
          count++;
}
```

```
while (true)  {
     while (count == 0)
          ; // do nothing
     nextConsumed =  buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     count--;
     /*  consume the item in nextConsumed
}
```

  - 由于调度导致的交错执行会导致count值的错误（竞态条件：内存空间被**并发访问**，并且至少有**一个访问为write**；同时**调度方式**也决定是否会产生竞态条件）
    - 正常情况：

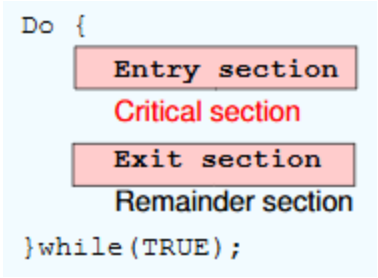
```
S0: producer execute register1 = count   {register1 = 5}
S1: producer execute register1 = register1 + 1   {register1 = 6}
S2: producer execute count = register1 {count = 6}
S3: consumer execute register2 = count   {register2 = 6}
S4: consumer execute register2 = register2 - 1   {register2 = 5}
S5: consumer execute count = register2  {count = 5 }
```

    - 出错情况：

```
S0: producer execute register1 = count   {register1 = 5}
S1: producer execute register1 = register1 + 1   {register1 = 6}
S2: consumer execute register2 = count   {register2 = 5}
S3: consumer execute register2 = register2 - 1   {register2 = 4}
S4: producer execute count = register1   {count = 6 }
S5: consumer execute count = register2   {count = 4}
```

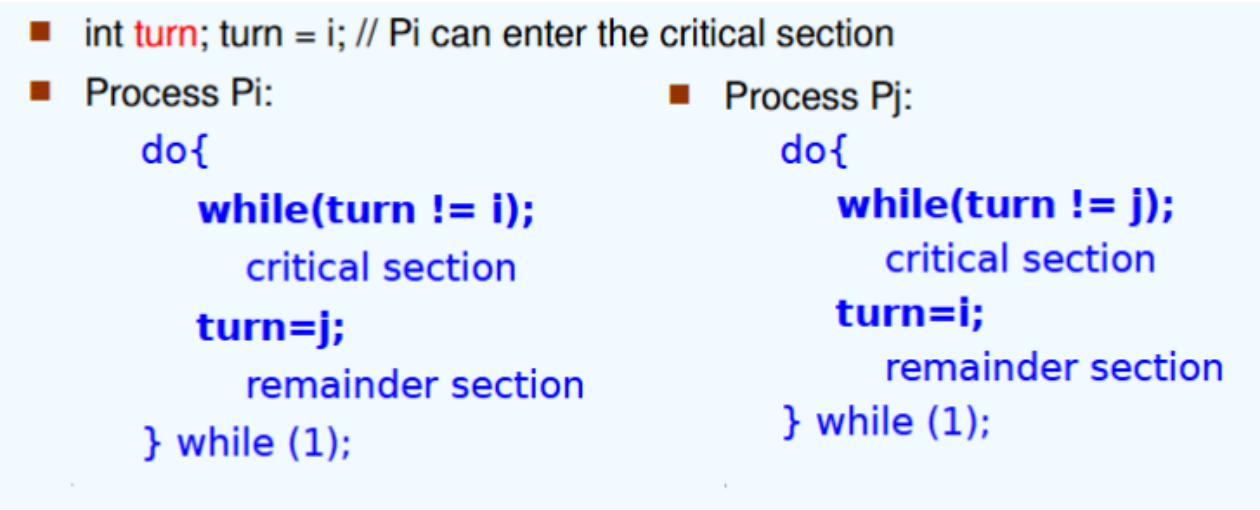
```
S0: producer execute register1 = count   {register1 = 5}
S1: producer execute register1 = register1 + 1   {register1 = 6}
S2: consumer execute register2 = count   {register2 = 5}
S3: consumer execute register2 = register2 - 1   {register2 = 4}
S4: producer execute count = register2  {count = 4 }
S5: consumer execute count = register1 {count = 6}
```

- Critical Section 临界区问题

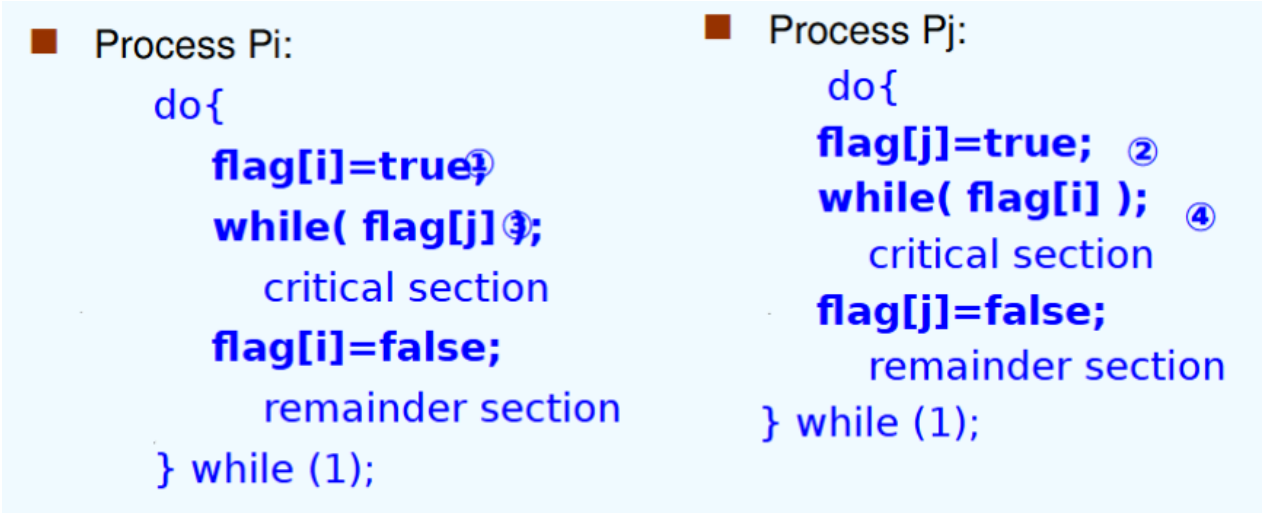



- 临界资源：指的是在多线程或多进程环境中被多个线程或进程共享的资源，这些资源需要通过适当的同步机制来保护，以确保同一时刻只有一个线程或进程可以访问这些资源
- 临界区：指的是程序中一个访问公共资源的程序片段，每个进程中访问临界资源的那段代码称为临界区
- 临界区问题的解决方法
  - Mutual Exclusion 互斥：一个进程在临界区中执行时，其他所有进程都不可以执行它们的临界区
  - Progress 空闲让进：如果没有进程在执行临界区，且存在一些进程想要进入临界区，那么一定会有一个进程被选择进入临界区执行
  - Bounded Waiting 有限等待：在一个进程提交进入临界区的请求到这个请求被授权之间，其他进程被允许进入临界区的次数必须有一个界限
  - （让权等待）：如果一直无法进入临界区，则释放CPU给其他进程

- 同步机制
  - 软件实现方法：单标志法、双标志后检查法、双标志前检查法、Peterson's Solution、面包房算法
    - 单标志法：设置公共整型变量turn，指示允许进入临界区的进程编号，退出临界区时交给另一个进程

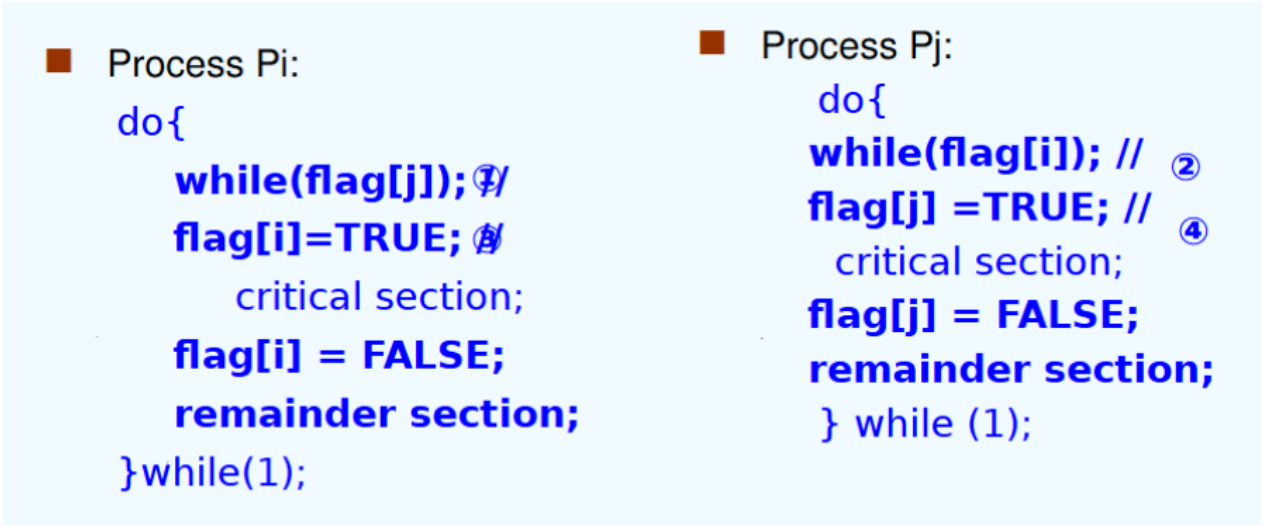



      - 满足互斥，不满足空闲让进，满足有限等待
    - 双标志后检查法：设置布尔数组flag[2]，用来标记各进程进入临界区的意愿， `flag[i] = true` 表示进程Pi想要进入临界区，先表达自己进入临界区意愿，再轮询对方是否想要进入临界区，确认对方不想后再进入临界区，访问结束后设置 `flag[i] = false` ，表示不想进入，允许对方进入



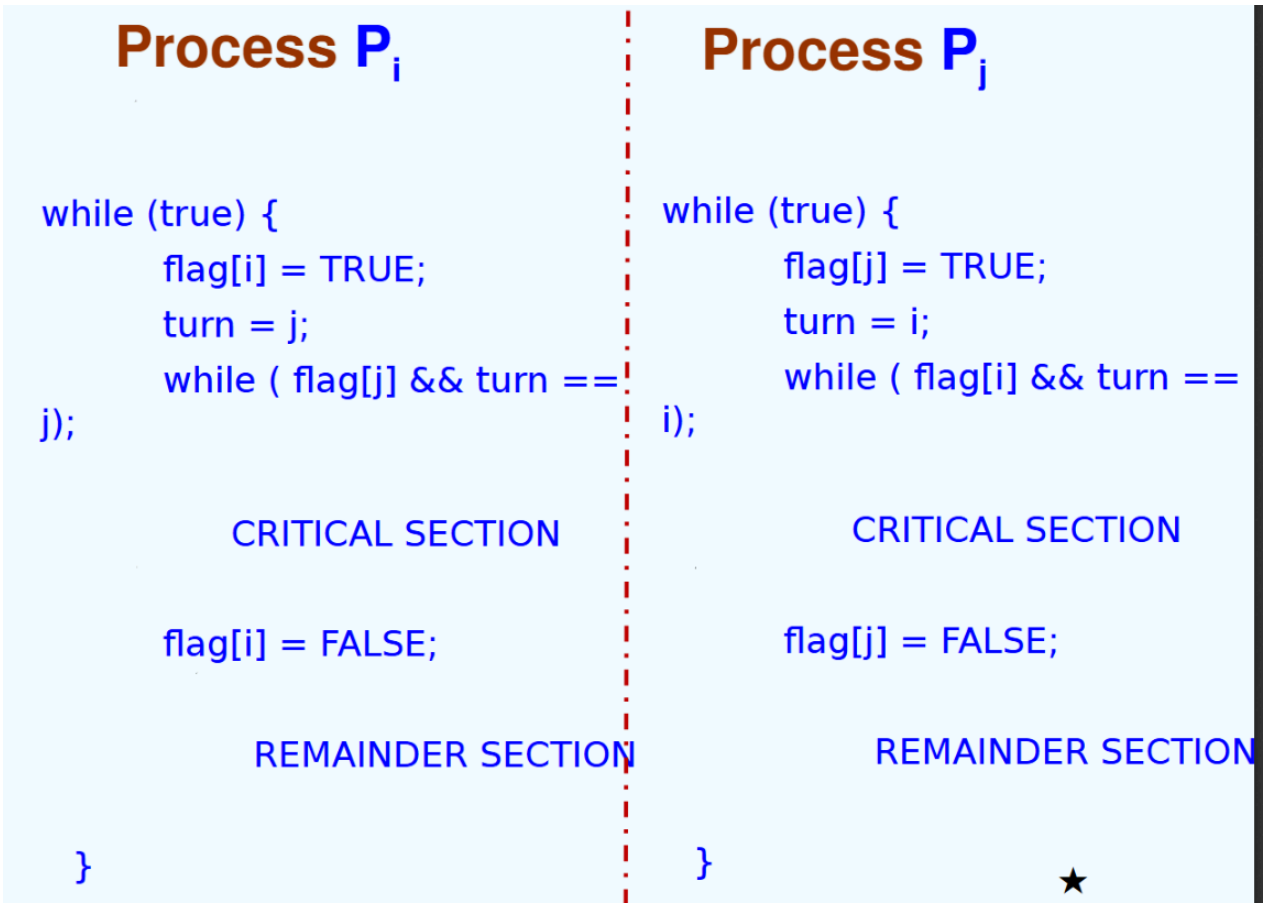

- 满足互斥，不满足空闲让进（由CPU调度引发的双方都进入不了），满足有限等待

- 双标志先检法：先轮询对方是否想要进入，确定对方不想进入后再进入



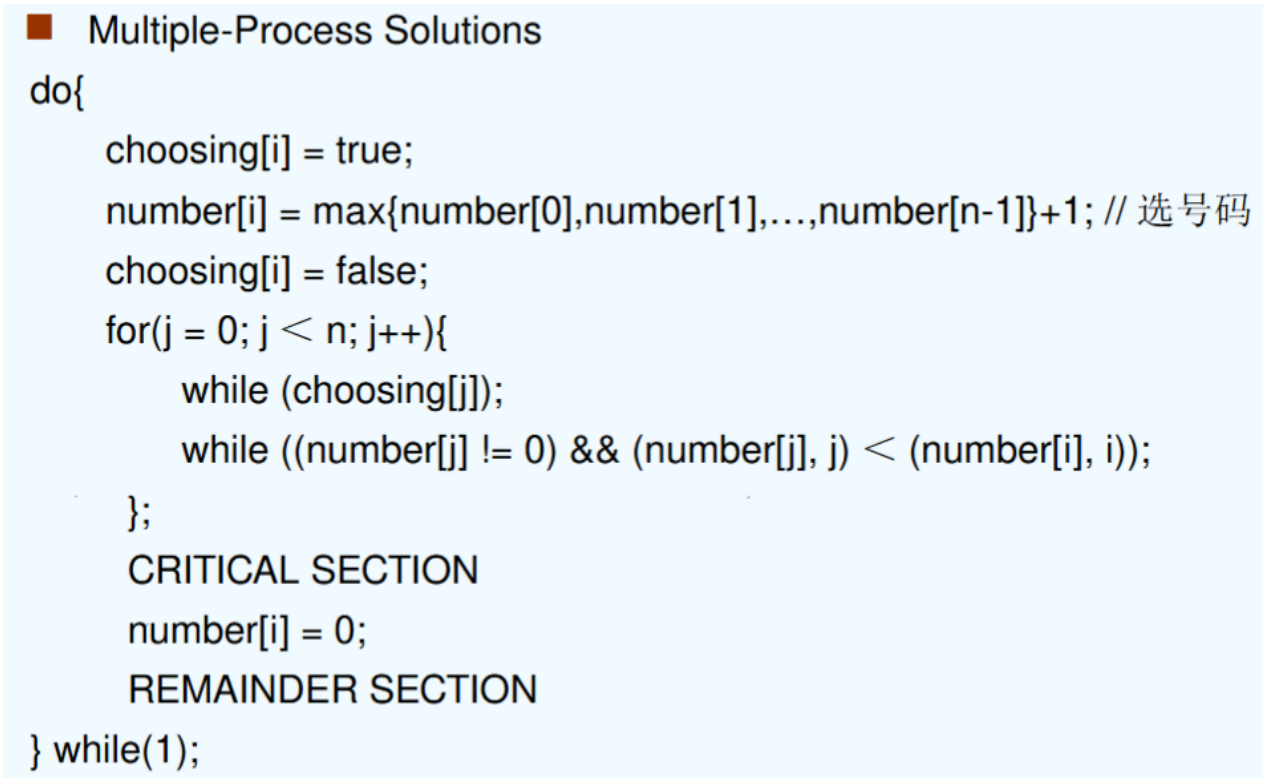

- 不满足互斥

- Peterson's Solution：结合单标志和双标志后检查，首先表达自身意愿（ `flag[i] = true` ）之后设置自身要进入（ `turn = 0/1` ），若双方互相确定对方都想进入时，turn只能等于一个值，因此会谦让对方进入，若一方不想进入，则其 `flag[i] = false` ，对方可直接进入




- 满足互斥，满足空闲让进，满足有限等待

- 面包房算法（Lamport）（类似取号和叫号）：在进入临界区前，进程会收到一个number，数字最小的先进入临界区；如果进程Pi和Pj收到相同的数字，i和j谁小谁先进临界区；生成的数字不会小于上一个数字




- 满足互斥，满足空闲让进，满足有限等待

- 硬件同步方法：关中断法、TestAndSet Instruction、Swap Instruction、Mutex Locks
  - 优点：适用于任意数目进程，单处理器or多处理器；简单，容易验证正确性；支持进程内存在多个临界区，只需为每个临界区设立一个布尔变量
  - 缺点：耗费CPU时间，无法实现让权等待；可能不满足有限等待；可能死锁
  - 关中断法：进入临界区之前直接屏蔽中断，保证临界区资源顺利使用，使用完毕打开中断

```
while (true) {
        Disable Interrupts;
        Critical section;
        Enable Interrupts;
        Remainder section;
}
```

    - 严重影响CPU执行效率，不适用于多CPU系统，安全性问题（错过重要中断请求）
  - TestAndSet Instruction：设置一个共享布尔变量 `lock` 初始化为false，利用 `TestAndSet` 机制进行互斥

```
boolean TestAndSet (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv:
}
```

```
while (true) {
        while ( TestAndSet (&lock ))
                ;   /* do nothing

                //    critical section

        lock = FALSE;

                //      remainder section

}
```

    - 满足互斥，满足空闲让进，不满足有限等待
  - Swap Instruction：

```
void Swap(boolean *a, boolean *b)
{
    boolean temp = *a;
    *a = *b;
    *b = *temp;
}
```

对于每个临界资源， Swap 设置一个全局布尔变量lock（初值为false），每个进程设置局部变量key（初值为true），进程调用 Swap 指令访问临界区，会交换key和lock的值，实现上锁，进入访问，退出时把lock重置为false

```
while (true) {
        key = TRUE;
        while (key = =TRUE)
                Swap(&lock, &key) ;

        //    critical section

        lock = FALSE;

        //      remainder section

}
```

  - 满足互斥，满足空闲让进，不满足有限等待
- compare_and_swap（CAS）：

```
int compare_and_swap(int *value, int expected, int new_value)
{
        int temp = *value;
        if (*value == expected)
           *value = new_value;
        return temp;
}
    while (compare_and_swap(&lock, 0, 1) != 0)
 ; /* do nothing */
```

  - 满足互斥，满足空闲让进，不满足有限等待
- Bounded-waiting with CAS：

```
while (true) {
        waiting[i] = true;
        key = 1;
        while (waiting[i] && key == 1)
            key = compare_and_swap(&lock,0,1);
        waiting[i] = false;
        /* critical section */
        j = (i + 1) % n;
        while ((j != i) && !waiting[j])
            j = (j + 1) % n;
        if (j == i)
            lock = 0;
        else
            waiting[j] = false;
    /* remainder section */
}
```

Lock
Boolean waiting[n]: 排队队列
to be initialize to false

This algorithm satisfies all the critical section requirements !

选择下一个进入临界区的进程

- 互斥锁：通过原子操作 `acquire()` 和 `release()` 锁实现对临界区的保护

```
acquire() {
        while (!available)
            ; /* busy wait */
        available = false;
}

release() {
        available = true;
}
```

```
while (true) {
    acquire lock

    critical section

    release lock

        remainder section
}
```

  - 问题：在获取锁时需要busy waiting，因此这种锁又被称为spinlock

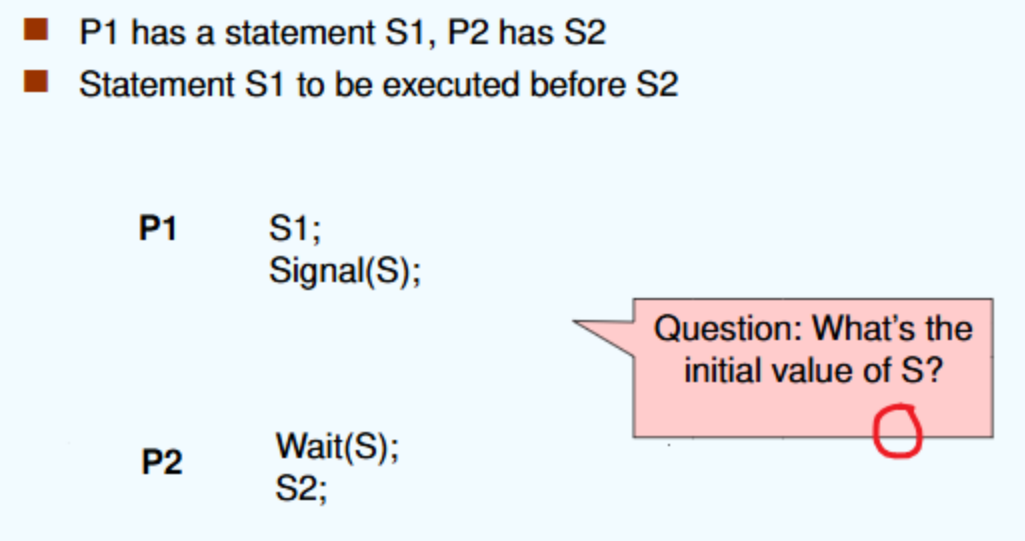
- 信号量 Semaphore：更简单的同步工具，是一个整型变量
  - 两种修改S的原子操作： `wait()` 和 `signal()` （又叫 `P()` 和 `V()` ）

```
wait (S) {
        while S <= 0
                ; // no-op
        S--;
}
signal (S) {
    S++;
}
```

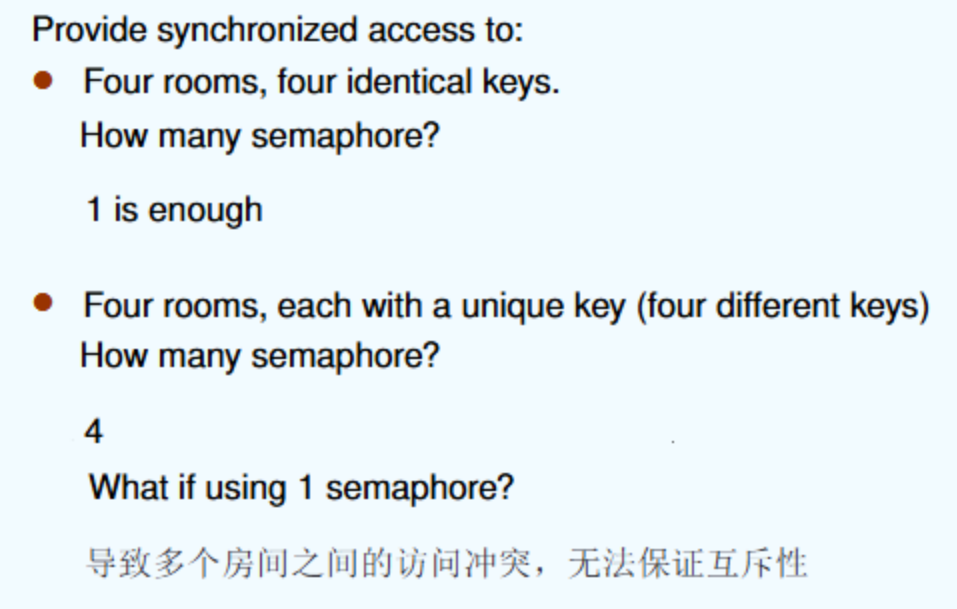
- 信号量类型：计数型（S值范围没有限制）；二进制型（S值范围限制在0和1，也被称为互斥锁）
- 利用信号量S构成互斥---------->

```
Semaphore S;     // initialized to 1
wait (S);
Critical Section
signal (S);
Remainder Section
```

- 利用信号量保持同步




- 如何设置信号量数量




- 信号量同步实现（Busy Waiting）

```
100    struct semaphore {
101      struct spinlock lock;
102      int count;
103    };
104
105    void
106    V(struct semaphore *s)
107    {
108       acquire(&s->lock);
109       s->count += 1;
110       release(&s->lock);
111    }
112
113    void
114    P(struct semaphore *s)
115    {
```

```
116       while(s->count == 0)
117         ;
118       acquire(&s->lock);
119       s->count -= 1;
120       release(&s->lock);
121    }
```

To allow one producer and one consumer, running P() and V() on different CPUs

If producer rarely acts, consumer spins on line 116. Expensive!!

- 信号量同步实现（no Busy waiting）：多一个waiting queue（S包含一个value和一个指向PCB链表的指针）
  - 两个新操作：`block/sleep`（将调用该操作的进程放入合适的waiting queue）和 `wakeup`（将waiting queue中的一个进程移除并放到ready queue）

```
wait (S){
        value--;
        if (value < 0) {
        // add this process to waiting queue
         block(); }
    }
        Signal (S){
                value++;
                 if (value <= 0) {
                // remove a process P from the waiting
queue
                  wakeup(P); }
           }
```

```
400     void
401     V(struct semaphore *s)
402     {
403       acquire(&s->lock);
404       s->count += 1;
405       wakeup(s);
406       release(&s->lock);
407     }
408
409     void
410     P(struct semaphore *s)
411     {
412       acquire(&s->lock);
413       while(s->count == 0)
414         sleep(s, &s->lock);
415       s->count -= 1;
416       release(&s->lock);
```

- `sleep(s, &s->lock)` can release the lock after the calling process is marked as asleep and waiting on the wait queue s.

    - 死锁和饥饿
        - Deadlock---------->

Let S and Q be two semaphores initialized to 1

| $P_0$ | $P_1$ |
|---|---|
| wait (S); | wait (Q); |
| wait (Q); | wait (S); |
| . | . |
| . | . |
| . | . |
| signal (S); | signal (Q); |
| signal (Q); | signal (S); |

            - P0和P1互相等待对方释放S和Q
        - Starvation：进程可能永远不被移出waiting queue
- 经典同步问题：有界缓冲区问题、读者写者问题、哲学家就餐问题
    - 基本定义
        - 通常用信号量表示资源或临界区
        - S.value > 0 表示有 S.value 个资源可用
        - S.value = 0 表示无资源可用或表示不允许进程再进入临界区
        - S.value < 0 则|S.value|表示再等待队列中进程的个数或表示等待进入临界区的进程个数
        - wait和signal操作必须成对出现：互斥操作处于同一进程；同步操作处于不同进程
        - 相邻的wait操作顺序需要格外注意：同步wait操作需要在互斥wait操作之前（相邻时）
    - 有界缓冲区问题
        - 三个信号量：mutex(init = 1), full(init = 0), empty(init = N)

- The structure of the producer process

```
while (true) {
        // produce an item
    wait (empty);
    wait (mutex);
        // add the item to the buffer
    signal (mutex);
    signal (full);
}
```

The structure of the consumer process

```
while (true) {
    wait (full);
    wait (mutex);
        // remove an item from buffer
    signal (mutex);
    signal (empty);
        // consume the removed item
}
```

- 读者写者问题
  - 一些并发进程共享一个数据集，读者只能读，写者可以读写；允许多个读者同时读，只能一个写着在写；写者需要等待所有读者读完再写
  - 两个信号量：mutex(init = 1), wrt(init = 1)
  - 一个变量：readcount(init = 0)

The structure of a writer process

```
while (true) {
    wait (wrt) ;
        // writing is performed
    signal (wrt) ;
}
```

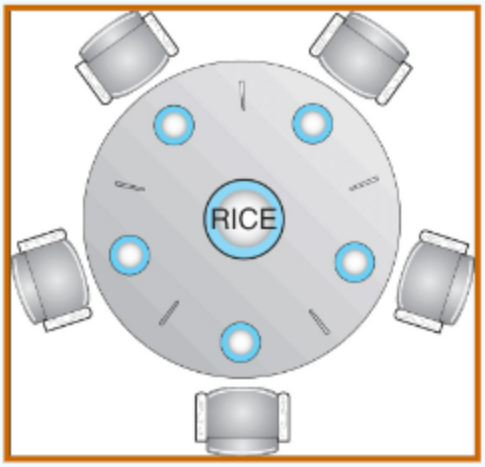
- The structure of a reader process

```
while (true) {
        wait (mutex) ;
        readcount ++ ;
        if (readcount == 1)  wait (wrt) ;
        signal (mutex)

            // reading is performed

        wait (mutex) ;
        readcount  - - ;
        if (readcount  == 0)  signal (wrt) ;
        signal (mutex) ;
}
```

"Locking" the wrt semaphore, rather than "waiting"

Reason is that wrt is initialized to "1"

"Unlocking" the wrt semaphore, rather than "signaling"

- 哲学家就餐问题
  - 

  - 五个信号量：chopstick[5](init = 1)

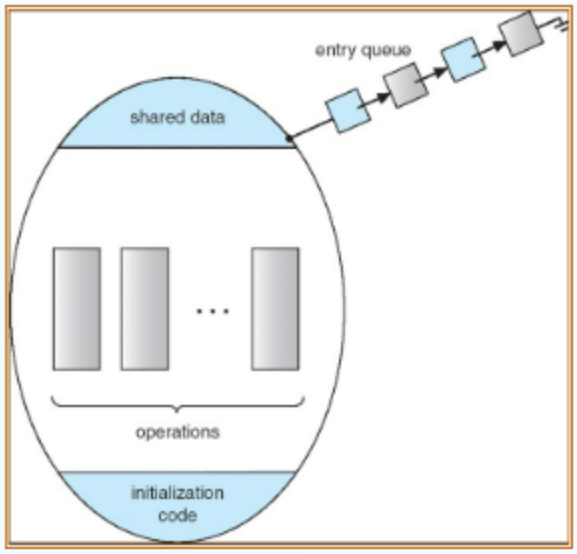
The structure of Philosopher *i*:

```
While (true)  {
        wait ( chopstick[i] );
        wait ( chopStick[ (i + 1) % 5] );

            //  eat

        signal ( chopstick[i] );
        signal ( chopstick[ (i + 1) % 5] );

            //  think

}
```

这种实现方式会导致死锁

- Monitors：一次只能有一个进程在monitor中活跃

```
monitor monitor-name
{
   // shared variable declarations
   procedure P1 (…) { …. }
         …

   procedure Pn (…) {……}

   Initialization code ( ….) { … }
         …

}
```




  - 条件变量：x, y
    - 每个变量的两个操作： `x.wait()` 挂起进程； `x.signal()` 激活一个正在wait的进程
  - 使用monitor解决哲学家就餐问题

```
monitor DP
  {
    enum { THINKING; HUNGRY, EATING) state [5] ;
    condition self [5];  //philosopher i can delay herself
    when unable to get chopsticks

    void pickup (int i) {
          state[i] = HUNGRY;
          test(i);
          if (state[i] != EATING) self [i].wait;
    }

     void putdown (int i) {
          state[i] = THINKING;
              // test left and right neighbors
           test((i + 4) % 5);
           test((i + 1) % 5);
     }

    void test (int i) {
          if ( (state[(i + 4) % 5] != EATING) &&
          (state[i] == HUNGRY) &&
          (state[(i + 1) % 5] != EATING) ) {
              state[i] = EATING ;
             self[i].signal () ;
           }
     }

     initialization_code() {
          for (int i = 0; i < 5; i++)
          state[i] = THINKING;
     }
}
```

```
dp.pickup (i)

    EAT

dp.putdown (i)
```

    - 当左右哲学家交替用餐时，会导致饥饿

- 进程同步例子：Solaris、Windows XP、Linux、Pthreads
  - Pthreads：mutex locks；condition variables
    - Pthread互斥锁

```
void reader_function ( void );
void writer_function ( void );

char buffer;
int buffer_has_item=0;
pthread_mutex_t mutex;
struct timespec delay;
void main ( void ){
   pthread_t reader;

   delay.tv_sec = 2;
   delay.tv_nec = 0;

   pthread_mutex_init (&mutex,NULL);
   pthread_create(&reader, pthread_attr_default, (void *)&reader_function),
NULL);
   writer_function( );
}
```

```
void writer_function (void){
   while(1){

      pthread_mutex_lock (&mutex);
      if (buffer_has_item==0){
         buffer=make_new_item( );
         buffer_has_item=1;
      }

      pthread_mutex_unlock(&mutex);
      pthread_delay_np(&delay);
   }
}
```

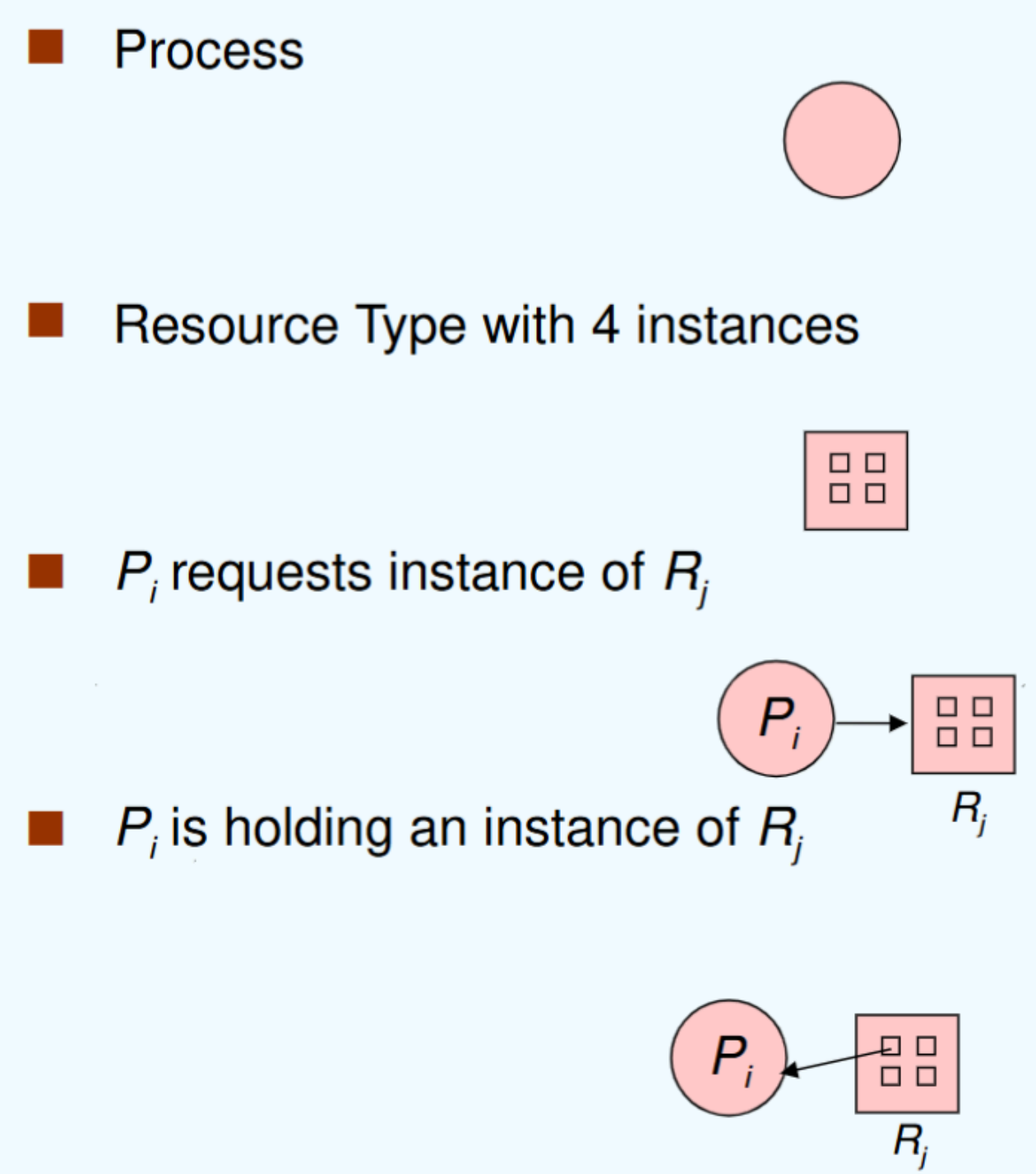
```
void reader_function(void){
    while(1){
        pthread_mutex_lock(&mutex);
        if(buffer_has_item==1){
            consume_item(buffer);
            buffer_has_item=0;
        }
        pthread_mutex_unlock(&mutex);
        pthread_delay_np(&delay);
    }
}
```

- Pthread条件变量

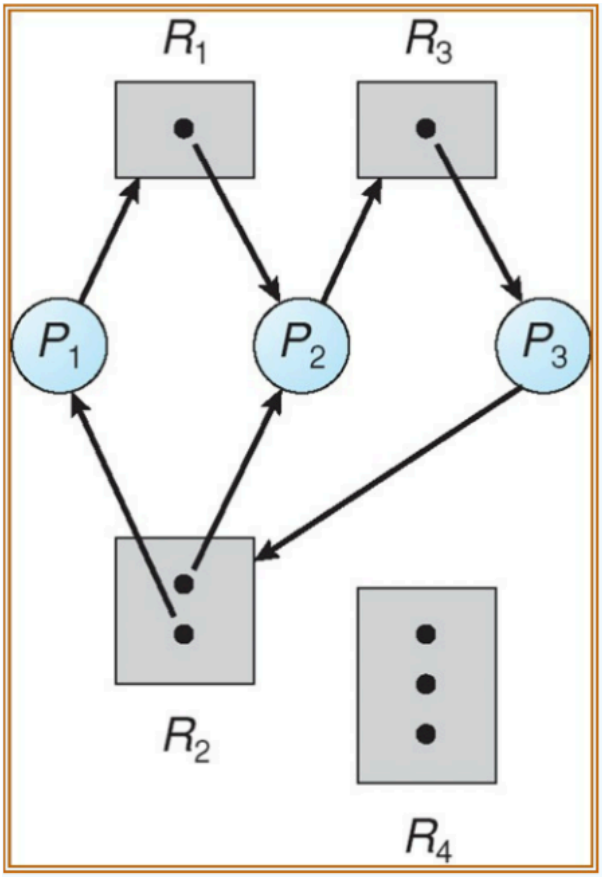
```
pthread_mutex_t count_lock;
pthread_cond_t count_nonzero;
unsigned count;

decrement_count()
{
    pthread_mutex_lock(&count_lock);
    while (count == 0)
        pthread_cond_wait(&count_nonzero, &count_lock);
    count = count - 1;
    pthread_mutex_unlock(&count_lock);
}

increment_count()
{
    pthread_mutex_lock(&count_lock);
    if (count == 0)
        pthread_cond_signal(&count_nonzero);
    count = count + 1;
    pthread_mutex_unlock(&count_lock);
}
```

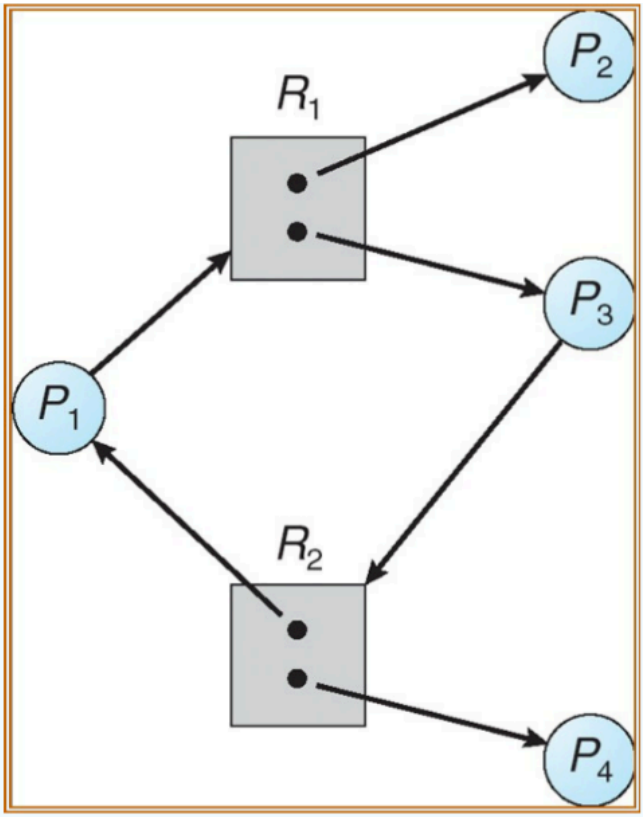
# CH7 Deadlocks 死锁

- 死锁问题：指多个进程因竞争共享资源而造成相互等待的一种僵局，若无外力作用，这些进程都将永远不能再向前推进
  - 产生死锁的条件（四个条件同时存在）
    - Mutual exclusion 互斥条件
    - Hold and wait 请求并保持条件
    - No preemption 不剥夺条件
    - Circular wait 环路等待条件
- System Model
  - Resource Types 资源类型
  - 每种资源类型有instances，例如：资源类型I/O devices有打印机、键盘、鼠标等实例
  - 每个进程通过 `request` 、 `use` 、 `release` 的流程使用资源

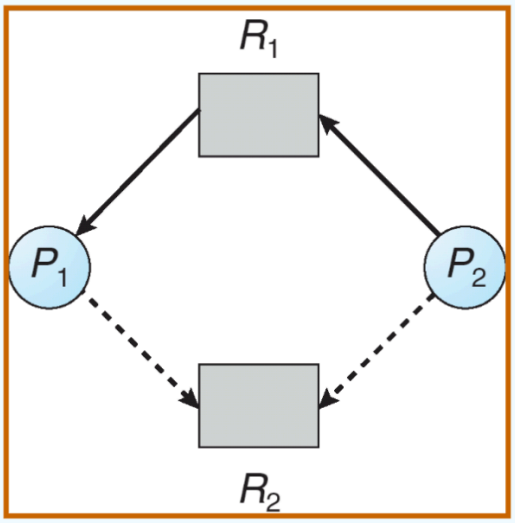
- 资源分配图 Resource-Allocation Graph：一系列点V和边E的合集
  - 两种顶点：P表示系统中的所有进程；R表示系统中的所有资源类型
  - request Edge：有向边 $P_i \rightarrow R_j$
  - assignment Edge：有向边 $R_j \rightarrow P_i$
  - 

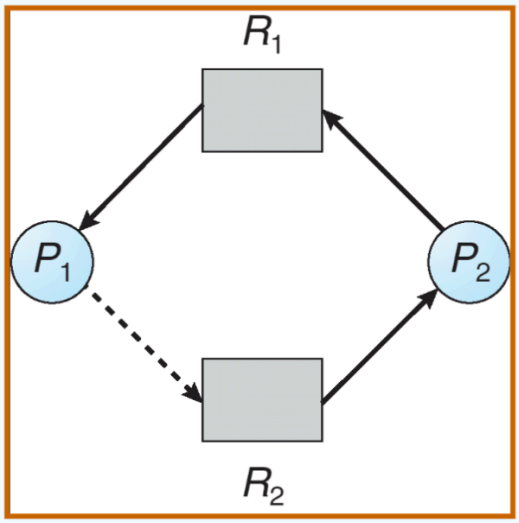
  - 图中无环→没有死锁
  - 图中有环→（1）每个资源类型只有一个实例，则死锁（2）如果每个资源类型有多个实例，则可能死锁
  - example




- 死锁的解决方法
  - Prevention 死锁预防：通过破坏产生死锁的必要条件之一来防止死锁的发生
    - Prevent Mutual Exclusion：给每个进程分配独占的资源
    - Prevent Hold and Wait：在进程执行之前获取所有需要的资源；或者只有当不持有资源时才能请求资源
    - Prevent No Preemption：直接去抢占
    - Prevent Circular Wait：给资源编号，进程请求资源只能按照资源编号的一定顺序进行
  - Avoidance 死锁避免：通过动态检测资源分配的安全性，确保系统不会进入不安全状态
    - 安全状态：对于进程序列$< P_1, P_2, ..., P_n >$，包含了系统中的所有进程，对于每一个进程$P_i$，其所请求的资源可以被$现有的资源 + P_j(j < i)所持有的资源$所满足。此时进程序列$< P_1, P_2, ..., P_n >$被称为安全序列
      - 系统处于安全状态$\rightarrow$没有死锁
      - 系统处于不安全状态$\rightarrow$可能有死锁
    - 避免算法
      - 每个资源类型只有一个实例：资源分配图算法
        - 只有当请求边转换为分配边时图中不会出现环，请求才会被授权




- 每个资源类型有多个实例：银行家算法
  - 数据结构

Let $n$ = number of processes, and $m$ = number of resources types.

- ***Available***: Vector of length $m$. If available [$j$] = $k$, there are $k$ instances of resource type $R_j$ available.
- ***Max***: $n$ x $m$ matrix. If $Max$ [$i,j$] = $k$, then process $P_i$ may request at most $k$ instances of resource type $R_j$.
- ***Allocation***: $n$ x $m$ matrix. If Allocation[$i,j$] = $k$ then $P_i$ is currently allocated $k$ instances of $R_j$.
- ***Need***: $n$ x $m$ matrix. If $Need$[$i,j$] = $k$, then $P_i$ may need $k$ more instances of $R_j$ to complete its task.

  - 安全算法

1. Let ***Work*** and ***Finish*** be vectors of length $m$ and $n$, respectively. Initialize:

   $Work = Available$

   $Finish\ [i] = false$ for $i = 0, 1, \ldots, n-1$.
2. Find an $i$ such that both:

   (a) $Finish\ [i] = false$

   (b) $Need_i \leq Work$

   If no such $i$ exists, go to step 4.
3. $Work = Work + Allocation_i$

   $Finish[i] = true$

   go to step 2.
4. If $Finish\ [i]$ == true for all $i$, then the system is in a safe state.

  - 资源请求算法

*Request* = request vector for process $P_i$. If $Request_i[j] = k$ then process $P_i$ wants $k$ instances of resource type $R_j$.

1. If $Request_i \le Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
2. If $Request_i \le Available$, go to step 3. Otherwise $P_i$ must wait, since resources are not available.
3. Pretend to allocate requested resources to $P_i$ by modifying the state as follows:

$$Available = Available - Request;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

- Call safety algorithm
- *If safe ⇒ the resources are allocated to Pi.*
- *If unsafe ⇒ Pi must wait, and the old resource-allocation state is restored*

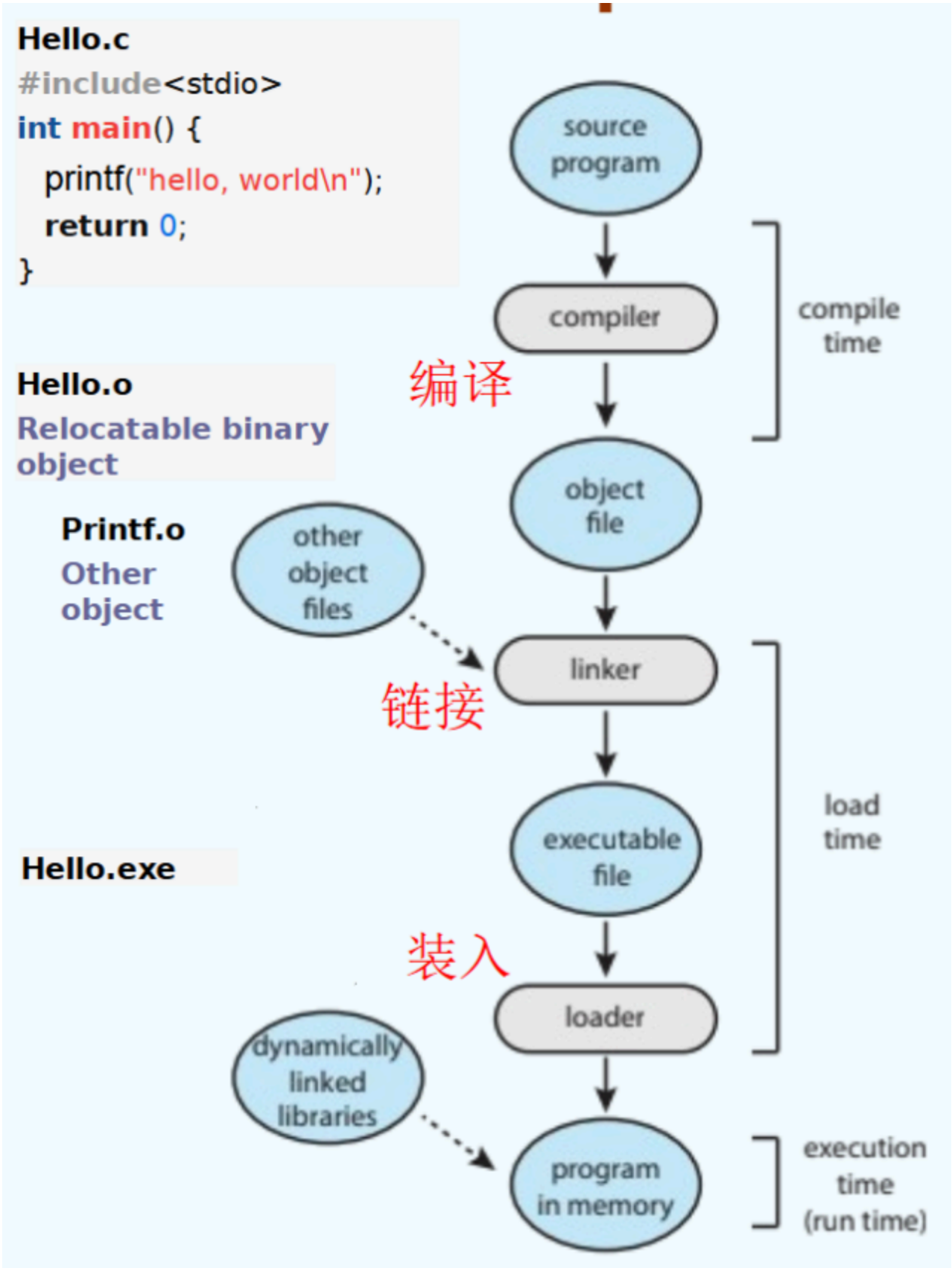
- Detection 死锁检测
  - Completely Reducible Graph 可完全简化图
    - 能消去图中所有边，能则称为可完全化简图
    - 找出一个既不阻塞又非独立的进程结点 Pi
    - 在顺利的情况下，分配给其资源让其完成，消去所有边变成孤立点
    - 循环上述两步操作，直至消去所有边，代表无死锁。
- Recovery 死锁恢复
  - 资源剥夺法
  - 撤销进程法
  - 进程回退法

# CH8 Main Memory 内存（12.2总结）

- 背景
  - 程序必须被带入内存并被放在一个进程中运行
  - CPU能直接访问的存储：内存和寄存器
  - 内存和寄存器之间有高速缓存
  - 内存层级图

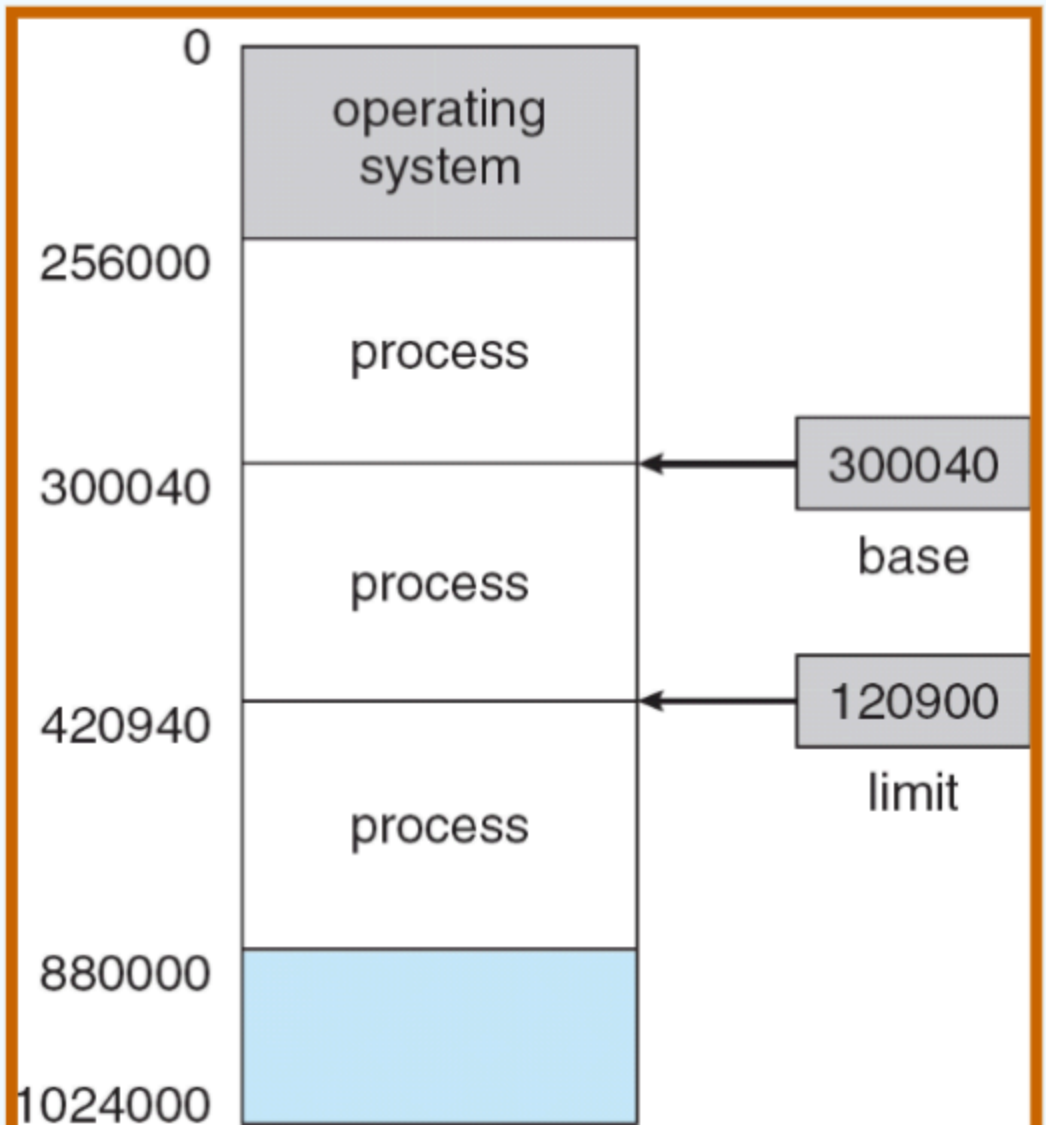
| | L1 Cache | L2 Cache | Main memory |
|---|---|---|---|
| **Block size** | 16--32 bytes | 32--64 bytes | 4--16 KB |
| **Size** | 16--64 KB | 256KB—8MB | |
| **Hit time** | 1 Clock Cycle | 1—4 Clock Cycles | 10—40 Clock Cycles |
| **Backing Store** | L2 Cache | Main memory | Disk |
| **Block replacement** | Random | Random | Replacement strategies |
| **Miss penalty** | 4-20 clock cycles | 40-200 clock cycles | ~6M clock cycles |

- 用户程序的处理步骤

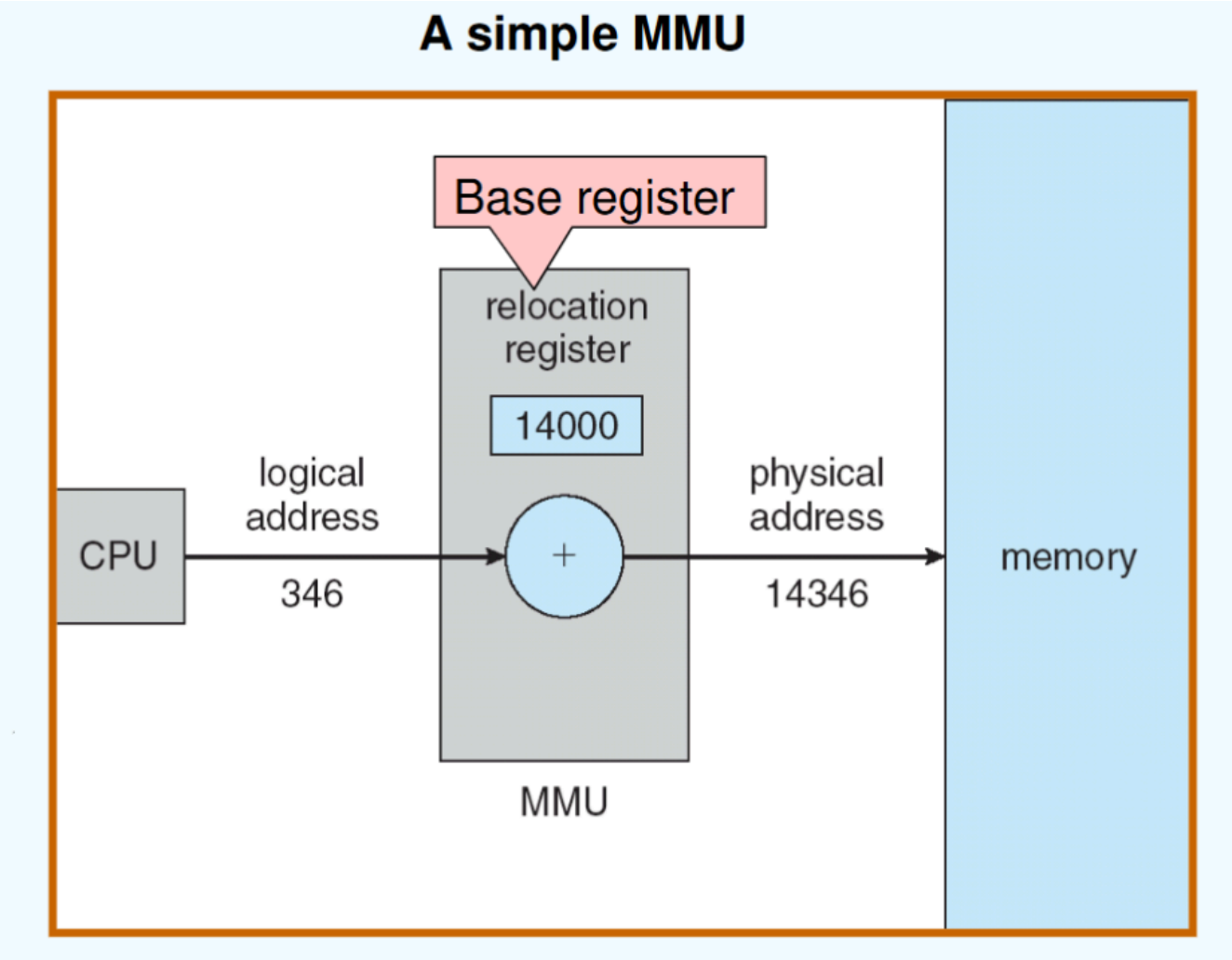



- 地址的不同形式
  - 符号地址：源程序中的地址大部分都是符号，例如程序中的变量 `count`
  - 可重定位地址：编译器会将符号地址绑定到可重定位地址，例如距离 moduleA 的开头14个bytes
  - 绝对地址：链接器或加载器将可重定位地址绑定到绝对地址，例如74014

- 操作和数据绑定到内存的三个时刻（地址映射）
  - 编译时刻
  - 装入时刻：静态
  - 执行时刻：动态
- Base and Limit Register 基址寄存器和限长寄存器




- 逻辑地址和物理地址
  - 逻辑地址：由CPU生成，也被称为虚拟地址
  - 物理地址：内存单元看到的地址
  - 在编译时刻和装入时刻绑定地址，逻辑地址和物理地址相同；在执行时刻绑定地址，逻辑地址和物理地址不同
- 内存管理单元MMU：映射虚拟地址到物理地址的硬件设备




    - 动态装入：部分装入，直到用到才装入
        - 更好的内存空间利用
        - 并行/并发友好
        - 运行时绑定
    - 动态链接
        - 运行时链接
        - 动态链接库：节省内存空间、减小可执行映像文件大小、不需要重新链接到新的库
- Contiguous Allocation 连续分配

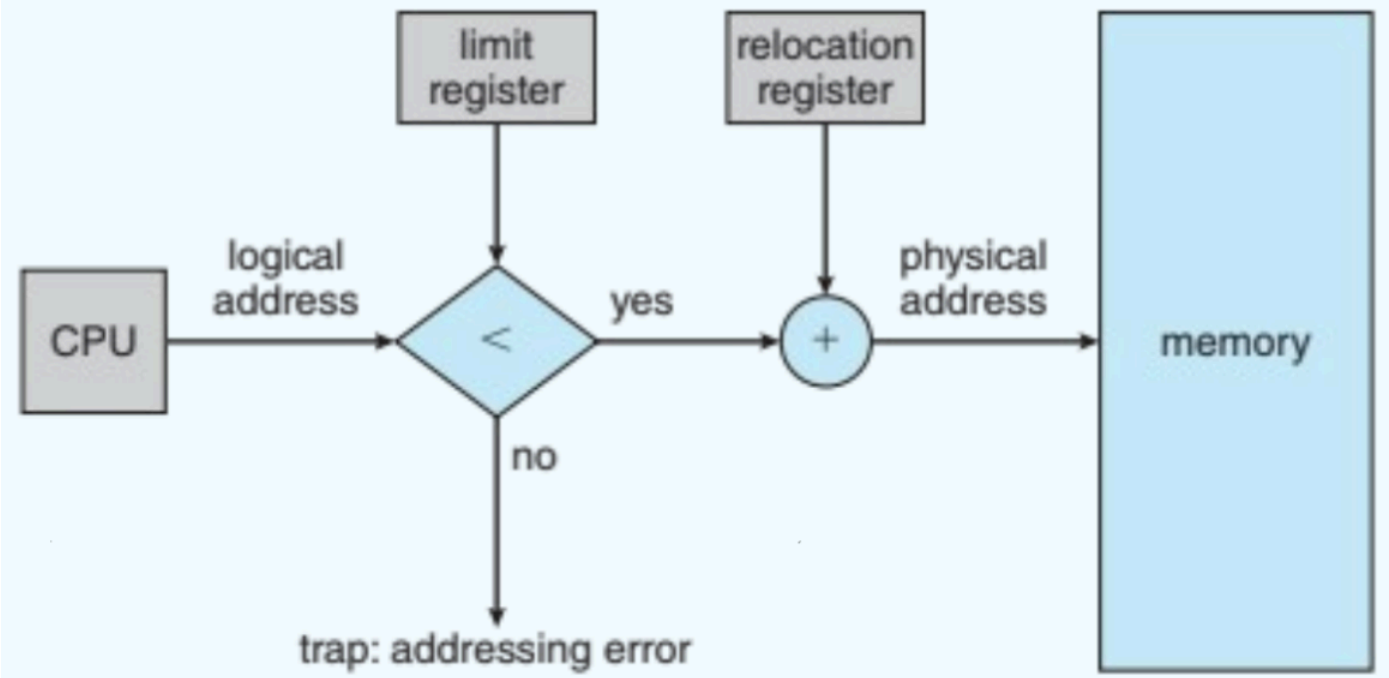
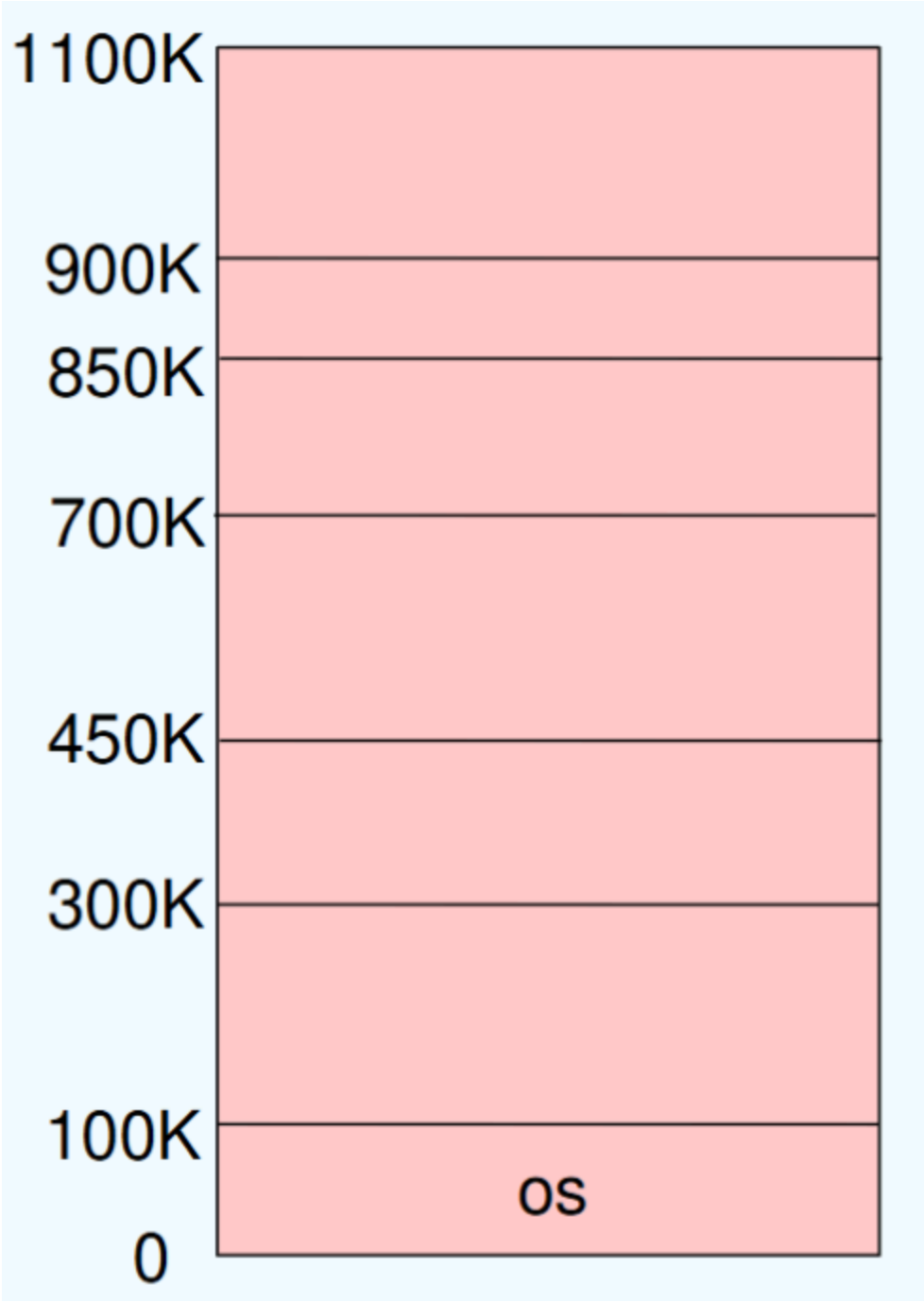



**Figure 9.6** Hardware support for relocation and limit registers.

- 单一连续分配：内存分为系统区和用户区，用户区每次只调入一道程序运行，无并发
- 多分区分配：将内存划分成若干个连续区域，称为分区，每个分区只能存放一个进程
  - 固定分区

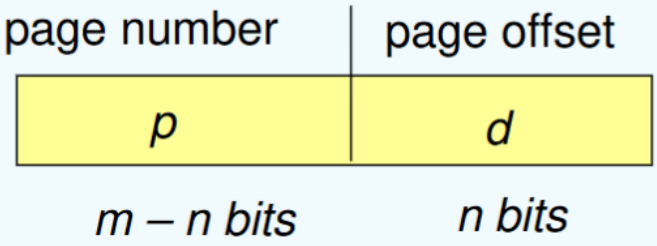



- 动态分区或可变分区：动态划分分区，在程序装入内存时把可用内存切出一个连续的区域分配给该进程，且分区大小正好适合进程的需要。
  - Hole：可用内存块，各种大小的hole散布在内存中
  - 操作系统维护两个信息：已分配分区 和 空闲分区（hole）
  - 动态存储分配算法
    - First-fit：分配第一个足够大的hole（Next-fit）
    - Best-fit：分配足够大的hole里面最小的那个
    - Worst-fit：分配最大的hole
    - 基于索引
      - 快速适应算法：将空闲分区按客量大小进行分类，设置素引表项，每一个空闲分区类型对应一项，挂成链(把原来一根变成多根)，根据进程长度，从素

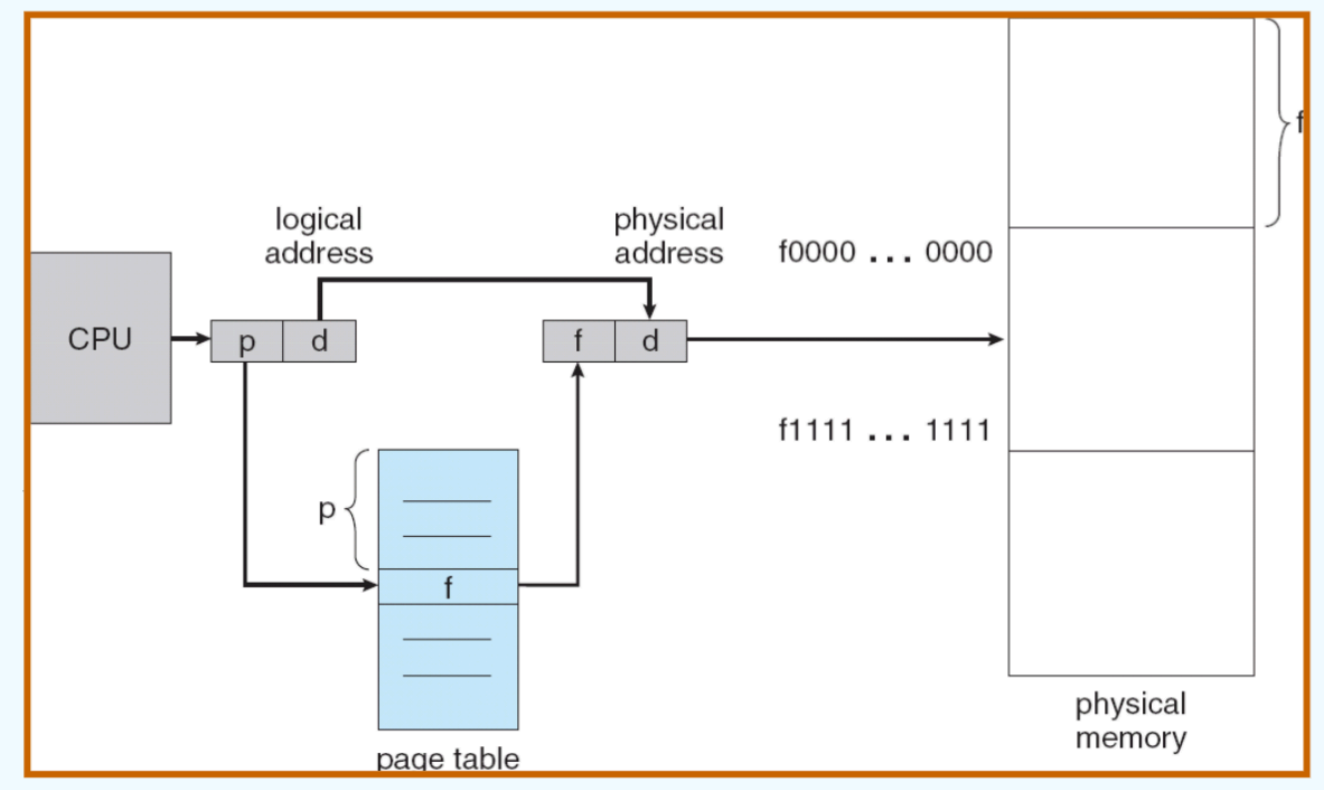
引表项中找到能容纳他的最小空闲区链表；从链表中取下第一块进行分配。
        - 伙伴系统：每个空闲分区大小必须是2的n次幂字节；对进程占用空间n计算一个i值使得$2^i > n$，从剩余空闲分区找最适合的;若无则将分区逐层拆分;释放时则逐层合并
        - 哈希算法:根据空闲分区链表的分布规律，建立哈希函数，构建一张以空闲分区大小为关键字的哈希表，根据所需空间大小通过计算得到哈希表的位置。
    - Fragmentation 碎片
        - 外部碎片：总共的内存空间满足一个请求，但是这些空间是不连续的碎片
        - 内部碎片：分配了比请求的内存略大的内存空间，在分配的内存空间内部有一小部分没有被使用
- Paging 分页
    - 地址转换

Address generated by CPU is divided into:

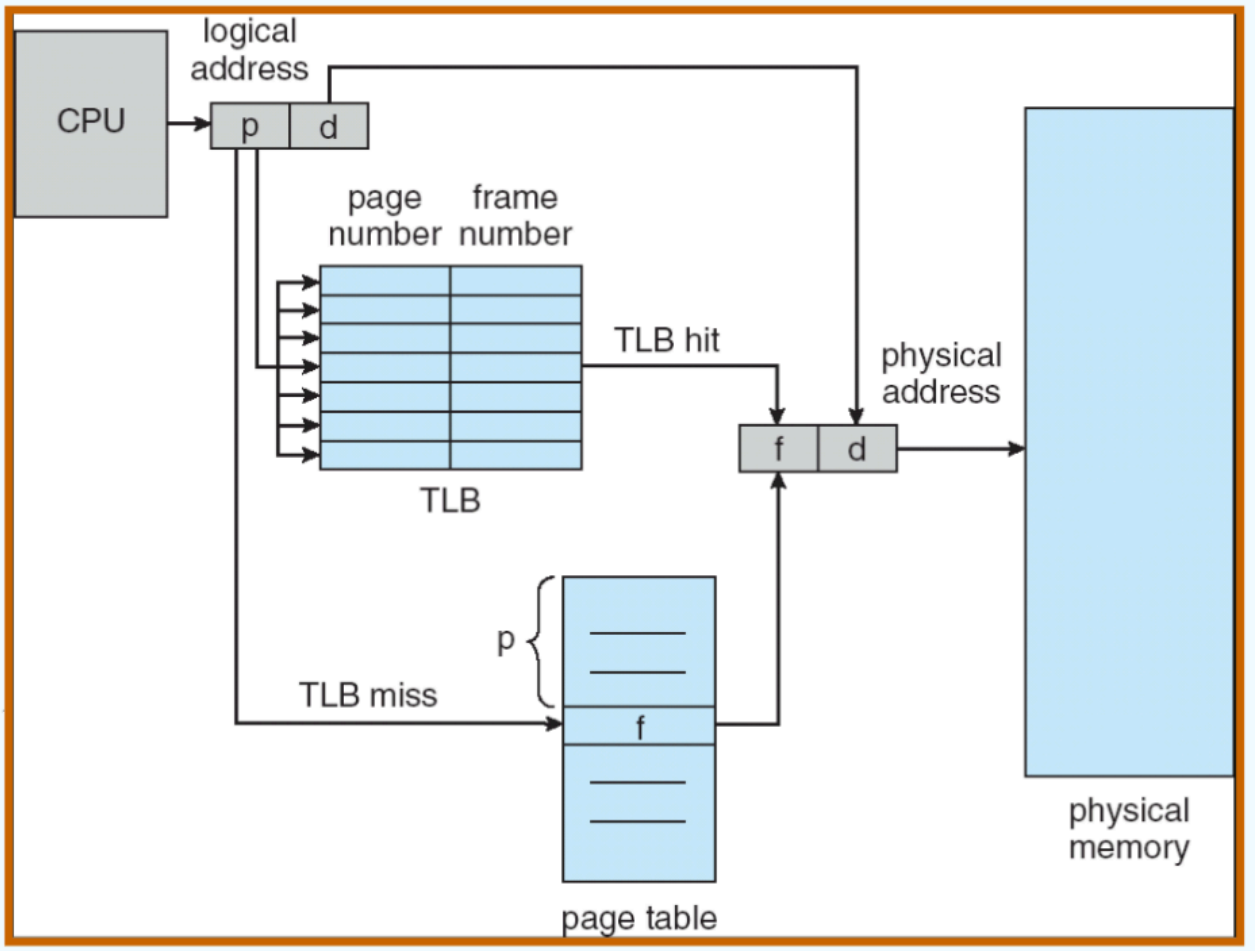
- **Page number (*p*)** – used as an index into a *page table* which contains base address of each page in physical memory
- **Page offset (d)** – combined with base address to define the physical memory address that is sent to the memory unit




- For given logical address space $2^m$ *and page size* $2^n$




- 页表硬件实施
  - PTBR 页表基址寄存器：指向页表基地址
  - PTLR 页表长度寄存器：确定页表的大小
  - TLBs 快表



- EAT 有效访问时间

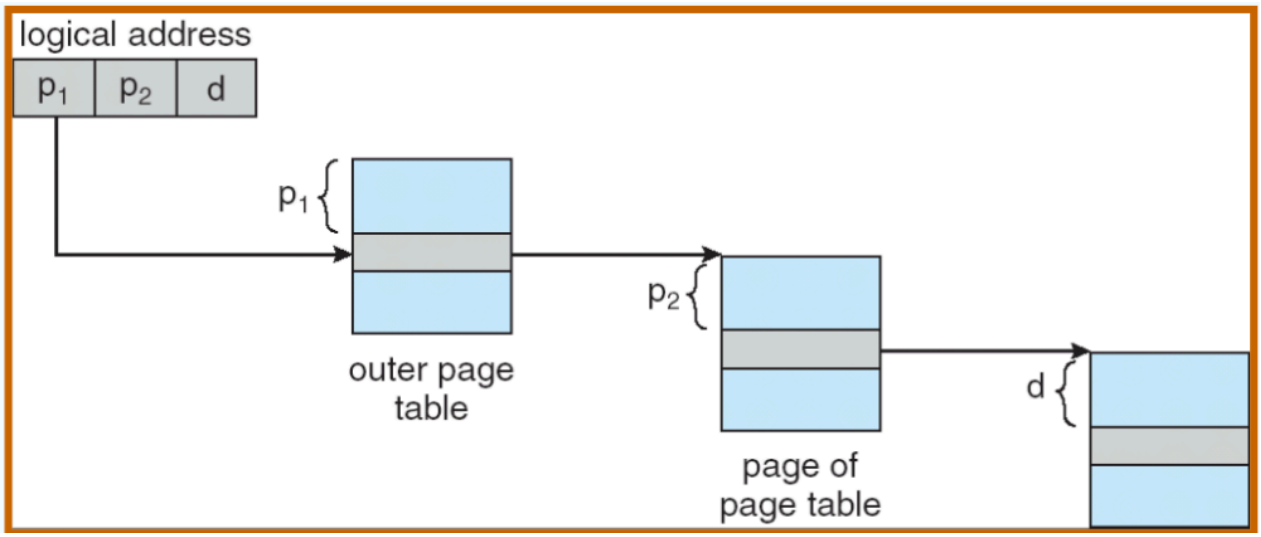
**Effective Access Time** (EAT)

$$EAT = (1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha)$$
$$= 2 + \varepsilon - \alpha$$

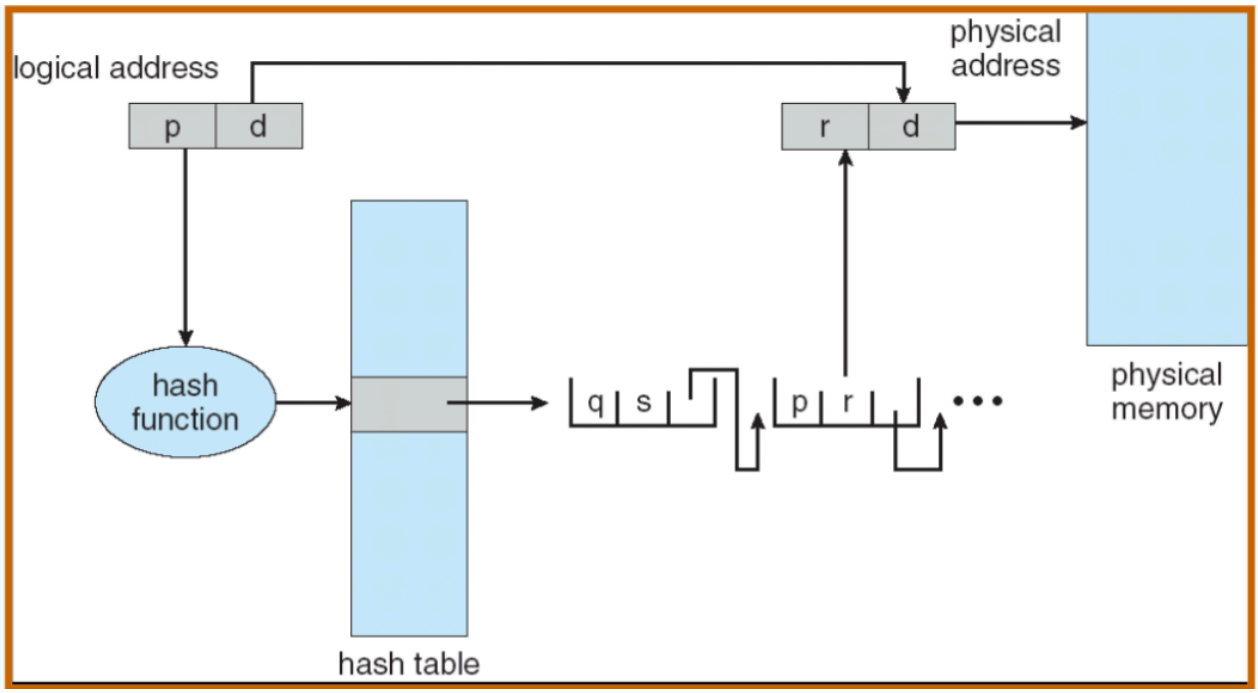
$\alpha$ = 80%, $\varepsilon$ = 2ns, memory access time = 100 ns

EAT=(100+2)*0.8+(100+100+2)*(1-0.8)=122 ns

- 页表中的有效位
- 共享页
- 页表结构
  - 多级页表

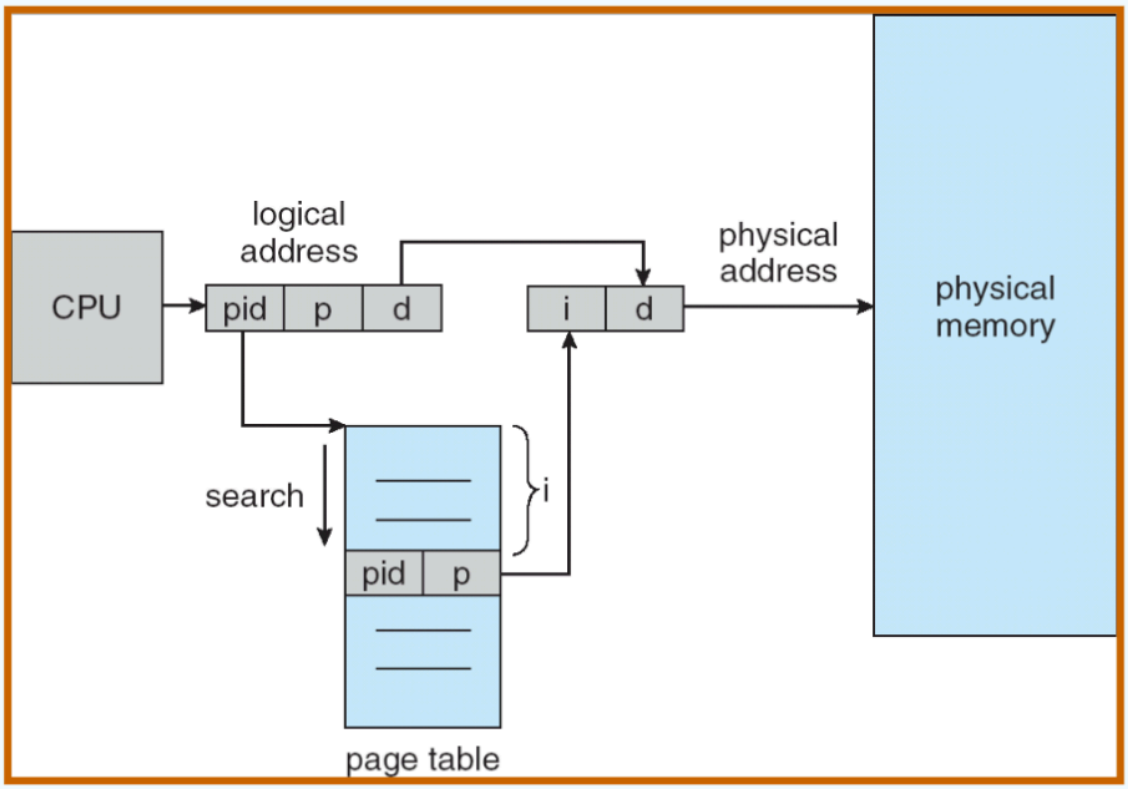



    - 优点：节省内存（根据运行的进程数来分配）
  - 哈希页表




- 优点：快速查找

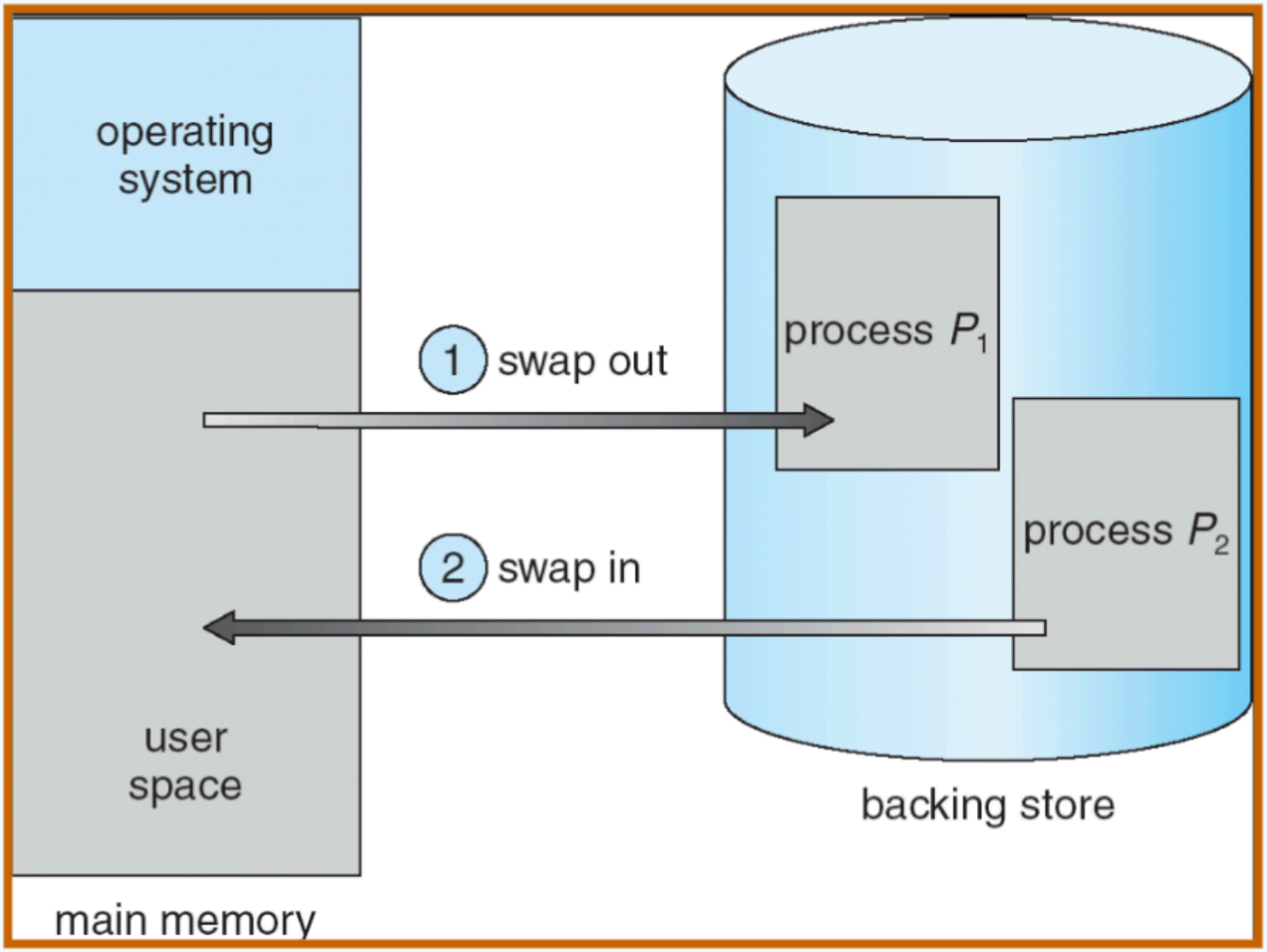
- 倒排页表



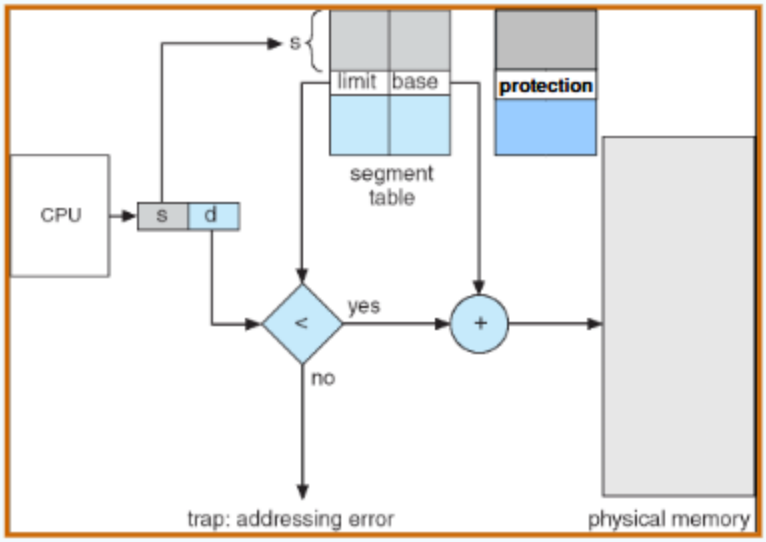

Search is slow, so put page table entries into a hash table. TLB can be used to speed up hash-table reference.

- 优点：只有一个页表，物理内存的缩影

- Swapping：一个进程可以暂时从内存中交换出去，存放到后备存储中，然后再次被调入内存以继续执行。

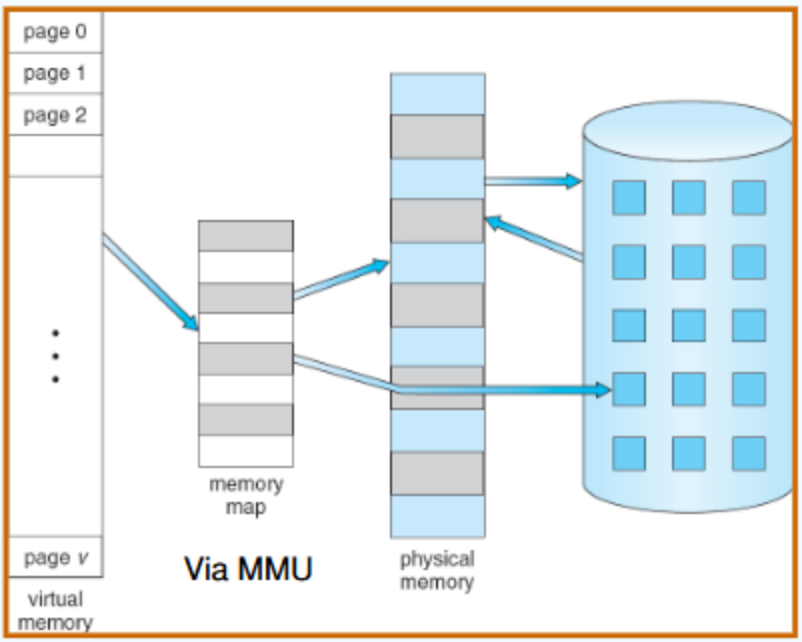



  - Backing store：足够大的快速磁盘，能够容纳所有用户的所有内存映像的副本；必须提供对这些内存映像的直接访问
  - Roll in，Roll out：用于基于优先级的调度算法的交换变体；低优先级进程被交换出去，以便高优先级进程可以被加载并执行
- Segmentation：一个程序是段的集合；段是一个逻辑单元，例如：主程序、函数、方法、对象、局部变量、全局变量、栈
  - 段架构：段表、段表基寄存器、段表限长寄存器

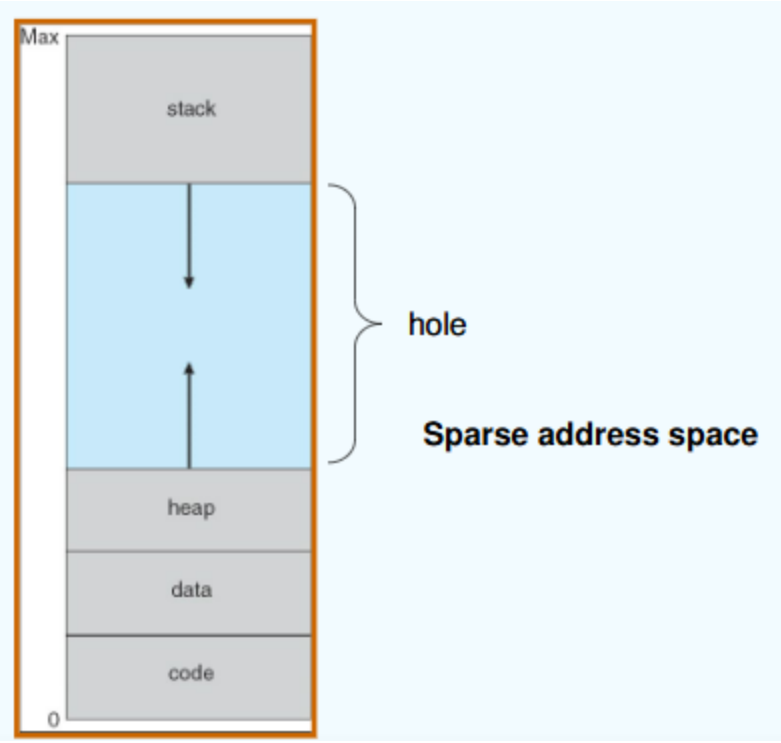


# CH9 Virtual Memory 虚拟内存

- 背景
  - 将用户的逻辑内存和物理内存分离开来：在运行时只需要部分程序在内存中即可，逻辑地址空间因此可以比物理地址空间更大；允许地址空间在多个进程间共享；允许更高效的进程创建
  - 局部性原理：指程序在执行过程中的一个较短时期所执行的指令地址和指令的操作数地址，分别局限于一定区域，表现为：
    - 时间局部性：一条指令的一次执行和下次执行，一个数据的一次访问和下次访问都集中在一个较短时期内
    - 空间局部性：当前指令和临近几条指令，当前访问的数据和临近的数据都集中在一个较小区域内
  - 虚拟内存和物理内存




  - 虚拟地址空间

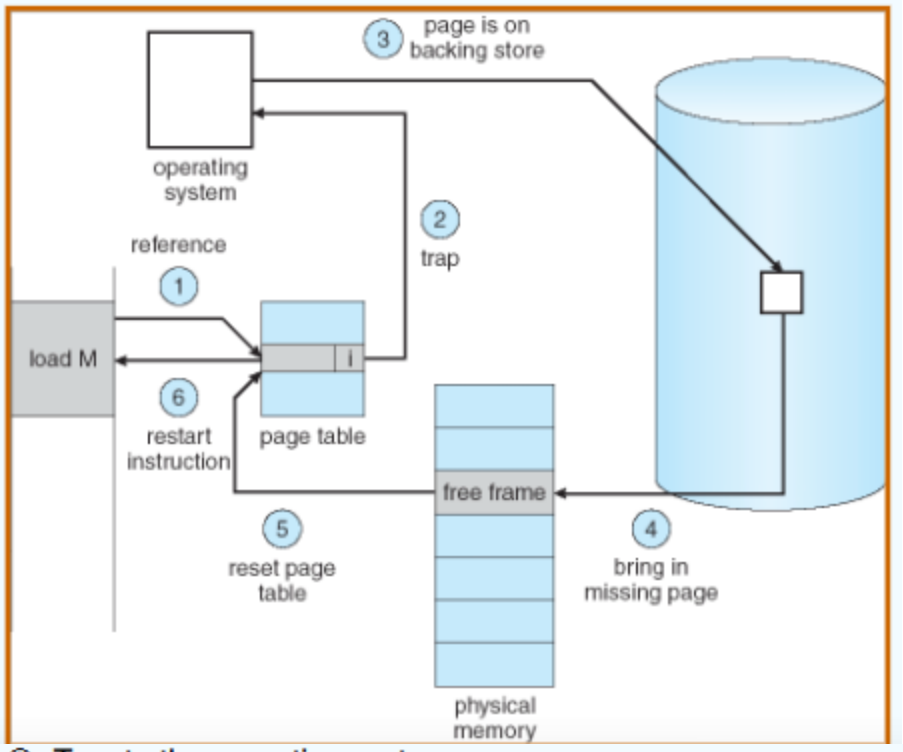



- 虚拟内存优化进程创建
  - Copy-on-Write：创建子进程时，不会立马复制父进程的内存地址空间，子进程会共享父进程的内存页面，并标记这些页面为只读，当父进程或子进程仅读取共享页面时，操作系统不会触发缺页中断；当父进程或子进程尝试写入某个共享页面时，操作系统会检测到写操作并触发一个缺页中断，操作系统处理缺页中断，为写入的进程分配一个新的物理页面，并将原始页面的内容复制到新的页面中，新页面可写，旧页面只读，继续被其他进程共享。
  - Memory-Mapped Files：将文件内容映射到进程的地址空间的技术。通过内存映射文件，可以像访问内存一样访问文件内容，而无需显式的进行读写操作。这种技术在处理大文件、提高文件访问性能以及实现进程间通信等方面非常有用。
- Demand Paging 请求式分页
  - 将页面带入内存仅当需要它时：更少的I/O，更少的内存，更快的响应，更多用户
  - Lazy swapper：除非页面被需要，否则不会将其转入内存
  - Valid-Invalid bit：v -> in-memory、i -> not-in-memory
  - 页表结构

| 物理块号 | 状态位 P | 访问字段 A | 修改位 M | 外存地址 |
|---|---|---|---|---|

- 状态位 P( 存在位 ): 用于指示该页是否已调入内存，供程序访问时参考。
- 访问字段 A: 用于记录本页在一段时间内被访问的次数，或最近已有多长时叠加间未被访问，提供给置换算法选择换出页时参考。
- 标记修改位 R/W: 表示该页在调入内存后是否被修改过。
- 外存地址：用于指出该页在外存上的地址，供调入该页时使用。

- Page Fault 处理




② Trap to the operating system.
- Save the registers and process state.
- Determine that the interrupt was a page fault.

③ Check that the page reference was legal, and determine the location of the page in secondary storage.

④ Issue a read from the storage to a free frame:
- Wait in a queue until the read request is serviced.
- Wait for the device seek and/or latency time.
- Begin the transfer of the page to a free frame.
- While waiting, allocate the CPU core to some other process.
- Receive an interrupt from the storage I/O subsystem (I/O completed).
- Save the registers and process state for the other process.
- Determine that the interrupt was from the secondary storage device.

⑤ Correct the page table and other tables to show that the desired page is now in memory.
- Wait for the CPU core to be allocated to this process again.

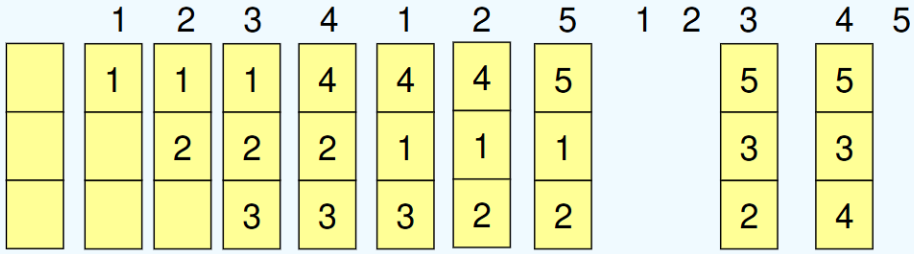
⑥ Restore the registers, process state, and new page table, and then resume the interrupted instruction.

- Page Replacement：找到内存中并不在实际使用的页，将他换出内存
  - 防止内存过度分配
  - 只有被修改过的页才会重新写回到磁盘
  - 完善了逻辑内存和物理内存的分离
  - 替换流程

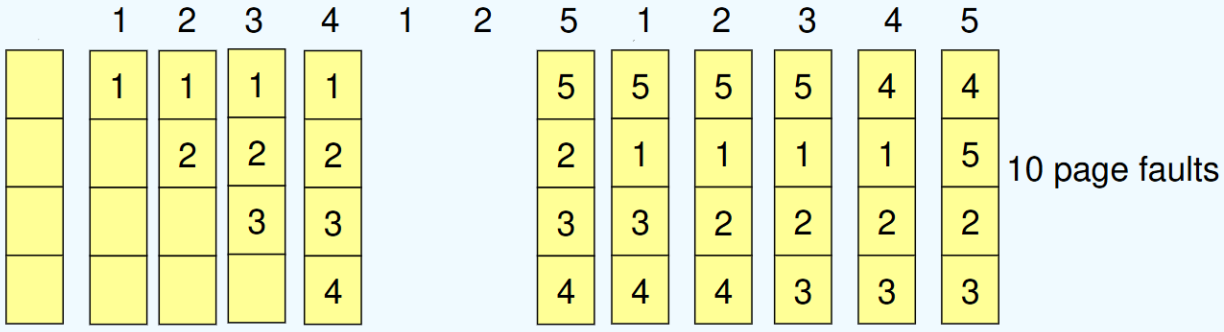
1. **Find the location of the desired page on secondary storage.**
2. **Find a free frame:**
   - **If there is a free frame, use it.**
   - **If there is no free frame, use a page-replacement algorithm to select a victim frame.**
   - **Write the victim frame to secondary storage (if necessary); change the page and frame tables accordingly.**

   Use dirty bit
3. **Read the desired page into the newly freed frame; change the page and frame tables.**
4. **Continue the process from where the page fault occurred.**

- 替换算法
  - FIFO 先进先出
    - Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
    - 3 frames (3 pages can be in memory at a time per process)

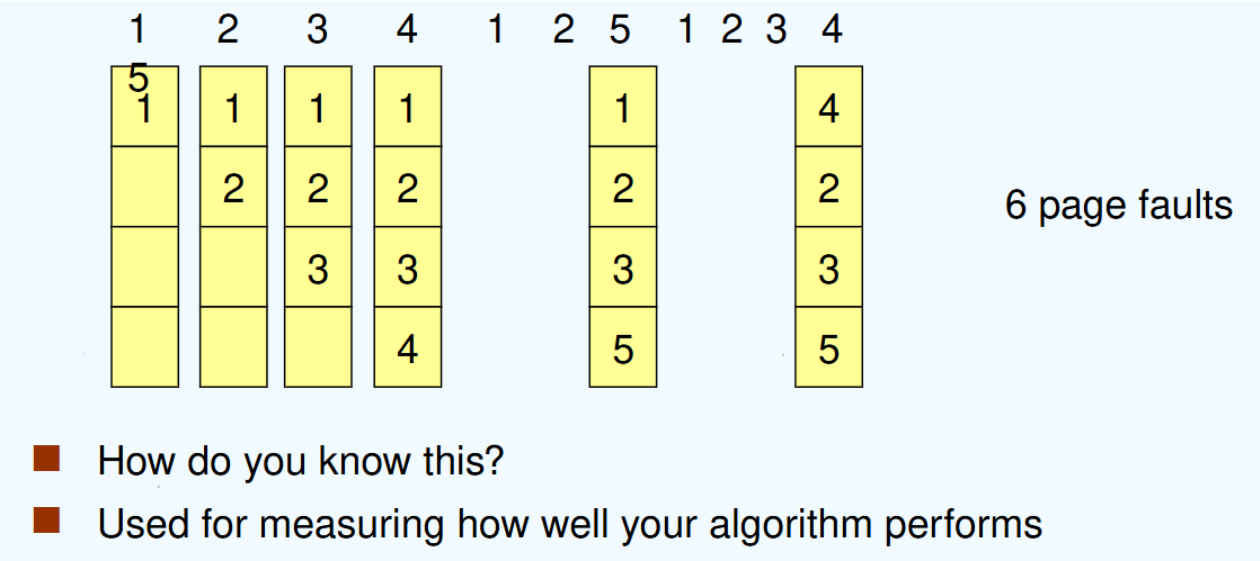



    - 4 frames

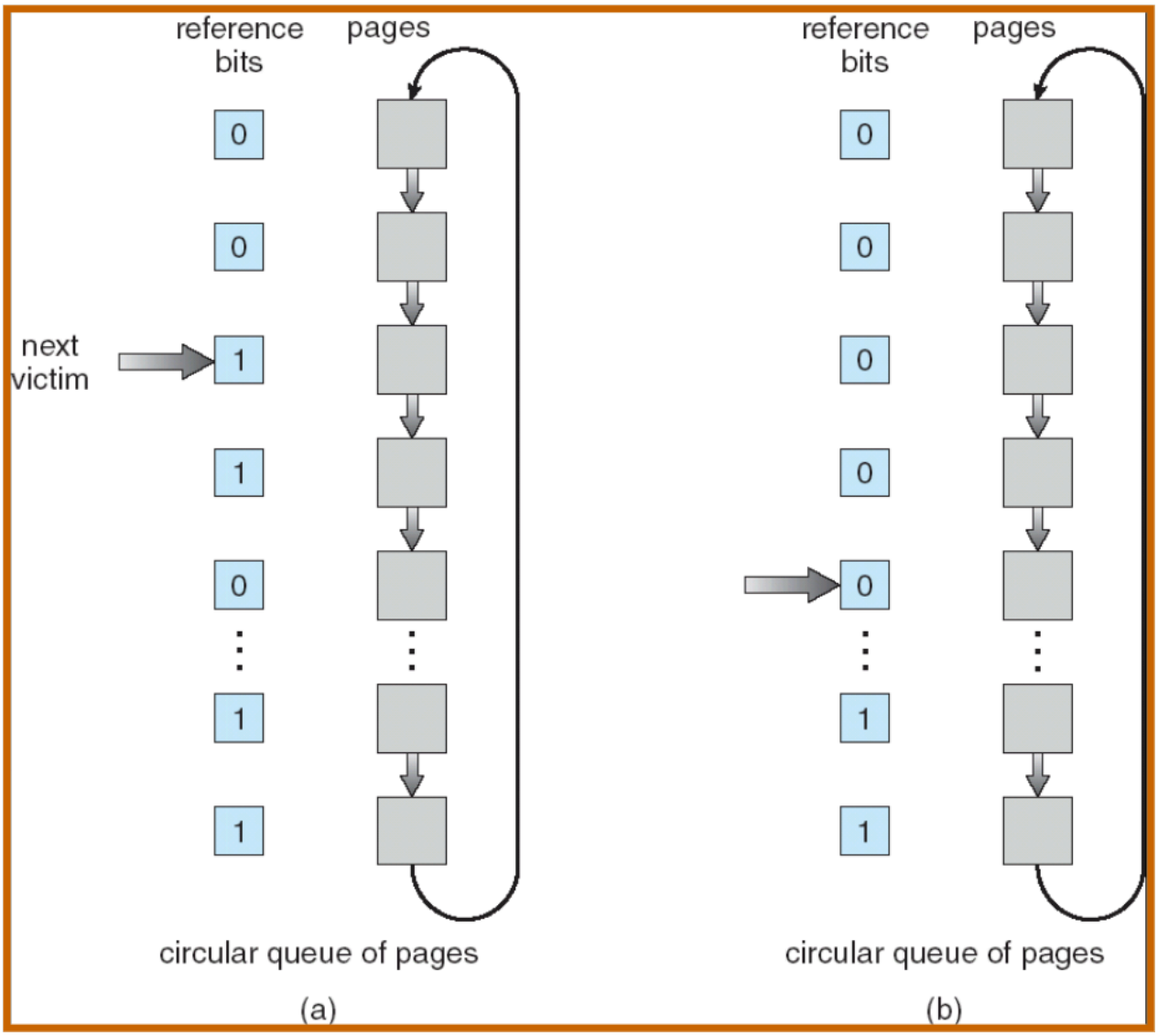



    - Belady's Anomaly: more frames ⇒ more page faults

      Why ?
  - Optimal Algorithm：将未来一段时间内最常不使用的页面替换出来



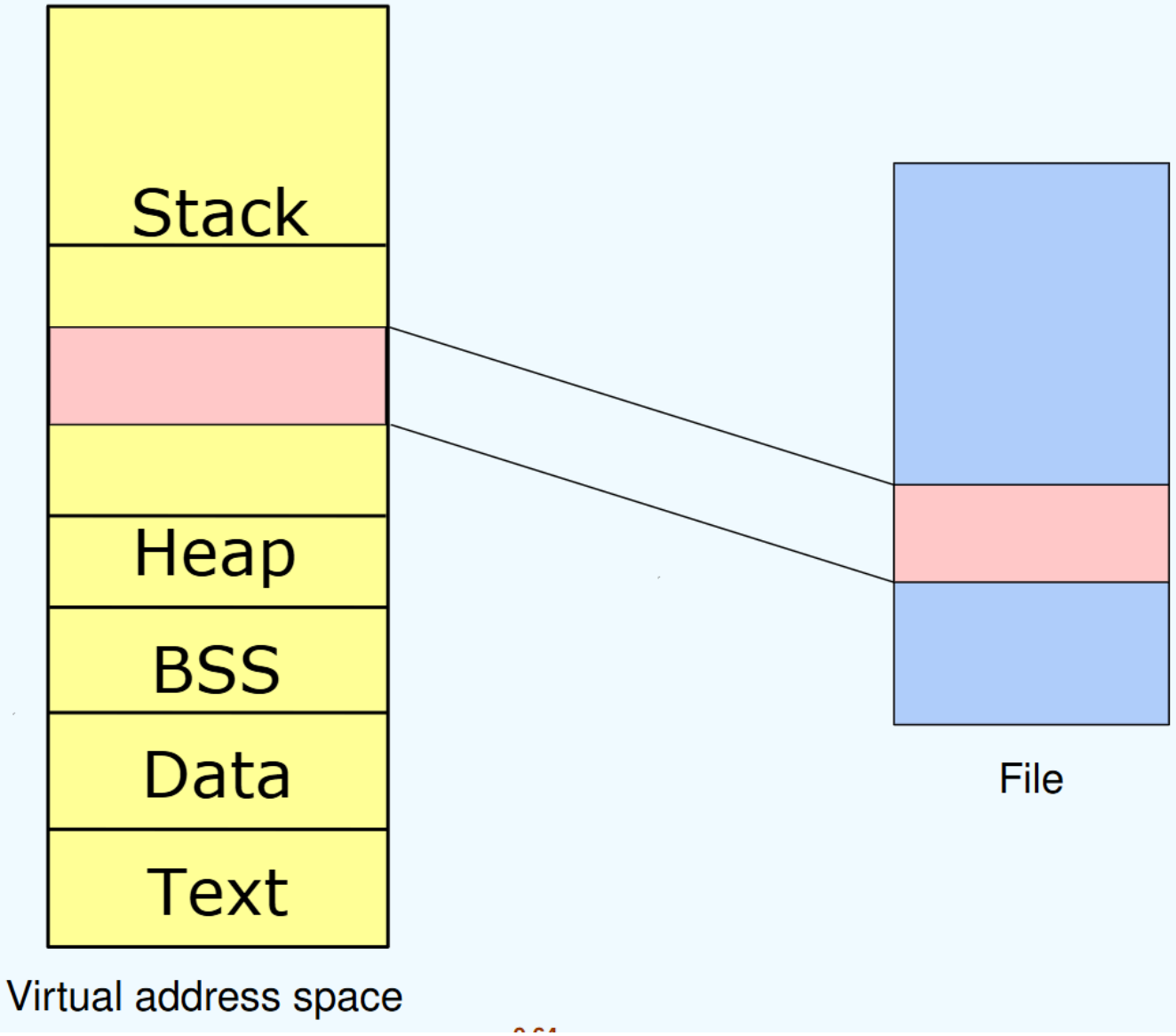

- LRU Algorithm 最近最久未使用：选择内存中最久没有引用的页面被替换（实现方法：counter或stack或移位寄存器）
- LRU近似算法（Second-chance、clock算法）：reference bit，若替换的页reference bit为1，则置0，不替换，找下一个




- 增强clock算法：(引用位，修改位)，淘汰次序$(0,0) > (0,1) > (1,0) > (1,1)$

    - Counting-based算法：LFU、MFU
    - Page Buffering Algorithm 页面缓冲算法
        - 通过被置换页面的缓冲，有机会找回刚被置换的页面
        - 被置换页面的选择和处理
            - 用 FIFO 算法选择被置换页，把被置换的页面放入两个链表之一：如果页面未被修改，就将其归入到空闲页面链表的末尾，否则将其归入到已修改页面链表。
        - 需要调入新的页面时：将新页面内容读入到空闲页面链表的第一项所指的页面，然后将第一项删除。
        - 空闲页面和已修改页面，仍停留在内存中一段时间，如果这些页面被再次访问，这些页面还在内存中。
        - 当已修改页面达到一定数目后，再将它们一起调出到外存，然后将它们归入空闲页面链表。
- Allocation of Frames：每个进程都有所需的最小的页面数量
    - Fixed Allocation
        - Equal Allocation：所有进程都分配相同的frames
        - Proportional Allocation：计算$a_i = (s_i / \sum_{i=1}^{n}(s_i)) \times m$
    - Priority Allocation
    - 全局替换vs局部替换：全局替换从所有帧中选择替换帧，局部替换从进程所有的帧中选择替换帧
- Thrashing 抖动
    - 如果进程没有足够的页面，缺页率就会很高，会导致：
        - low CPU utilization
        - Queuing at paging device, the ready queue becomes empty
        - operating system thinks that it needs to increase the degree of multiprogramming
        - another process added to the system
    - 解决方法：增加物理内存、优化页面置换算法、在CPU调度中引入工作集算法、动态调整进程的内存分配、限制并发进程数、内存压缩、……
    - 工作集模型：
        - $\Delta \equiv$ 工作集窗口 $\equiv$ 一个固定的页面访问数量
        - $WSS_i$(进程$P_i$的工作集大小) = 最近$\Delta$内访问过的页面数量
        - $D = \sum WSS_i$ = 系统中所有进程需要的$frames$总数
        - $m$ = 所有可获得的$frame$
        - $D > m \Rightarrow Thrashing$，此时暂停一个进程
        - 如何track工作集状态：间隔定时器

- Memory-mapped Files
  - 将文件从磁盘中的一个block载入到物理内存中的一个页，减少了I/O操作的消耗，增加了访问效率




  - mmap，unmap
- 分配内核内存
  - Buddy System
  - Slab Allocator
- Other Issues
  - Prepaging
  - Page Size
  - TLB Reach
  - Program Structure
  - I/O inerlock

# CH10 File-System Interface 文件系统接口

- 文件概念
  - 文件系统：控制数据在存储介质中存储和检索的方法
  - 文件：Contiguous logical address space；一系列bits、bytes、lines或records，其含义由创建者和用户定义。
    - 类型：Data、Program
    - 结构
      - None 无结构文件：words，bytes
      - Simple record structure：Lines、Fixed length 定长、Variable length 变长
      - Complex Structures：Formatted document 格式化的文件、Relocatable load file
    - 属性
      - Name：唯一人类可读形式的信息
      - Identifier：在文件系统唯一识别文件的特殊tag（数字）
      - Type：
      - Location：指向文件在设备上的位置的指针
      - Size：文件的大小
      - Protection：控制权限
      - Time，date，and user identification
      - 文件信息保存在目录结构中
    - 操作：创建、写、读、定位、删除、Truncate
  - Open-file table
    - 系统调用 `Open()` 返回一个指向open-file table中的一条记录的指针
      - Per-process table：Current file pointer, Access rights
      - System-wide table：Open count
  - Open Files
    - File pointer
    - File-open count
    - Disk location of the file
    - Access rights
- 访问方法
- 目录结构
- 文件系统挂载
- 文件共享
- 文件保护