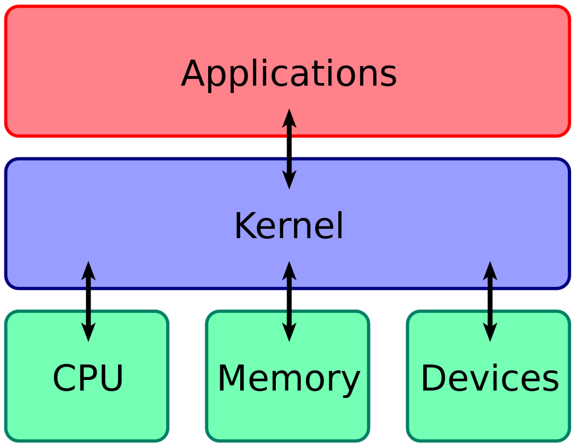
Lecture 01

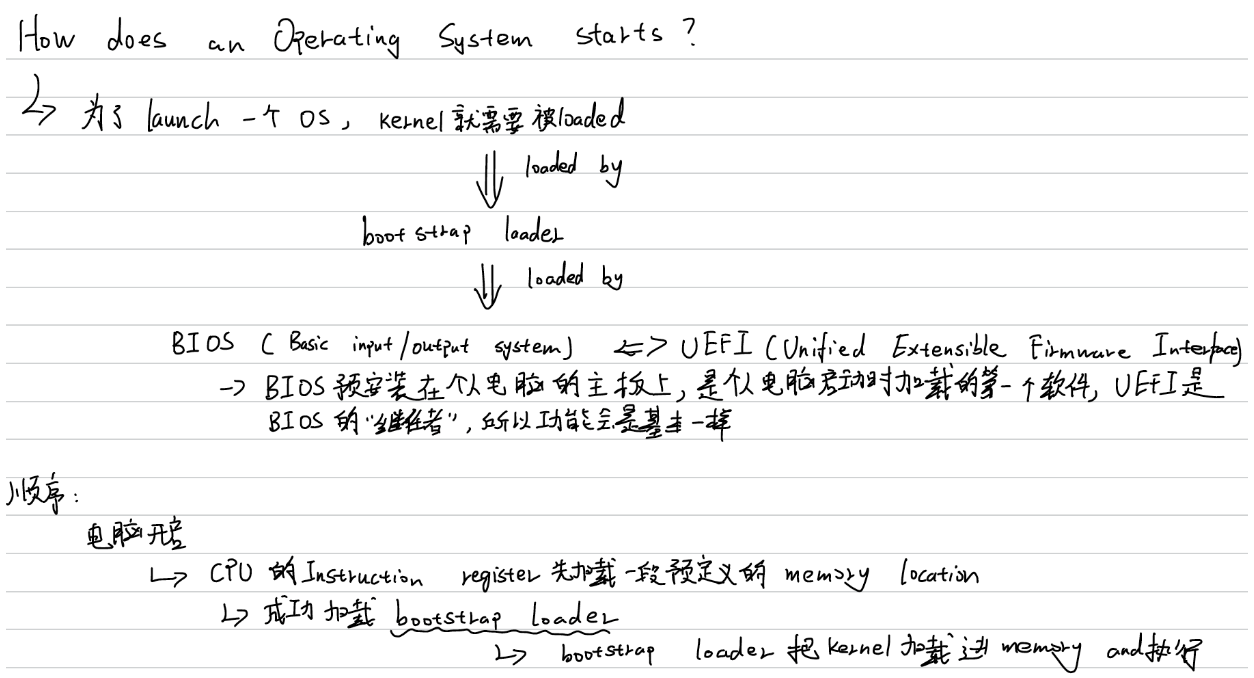
- "Present a virtual machine"
    - Virtual Machine指的是虚拟机，而虚拟机指通过软件模拟的具有完整硬件系统功能的，运行在一个完全隔离环境中的完整计算机系统。可以看作是对computer system的一个模拟，不存在实际的硬件，但却把需要硬件系统的模拟了出来，比如可以在windows上安装linux的虚拟机，不管是什么操作系统，都需要硬件的参与，而在windows上安装的linux虚拟机明显没有适配的硬件参与，所以这时候需要虚拟机去完成模拟
- Kernel是什么（就是OS最主要的一部分）



Kernel内核是现代操作系统中最基本的一个部分，所以他属于操作系统，可以看到他坐落在Application和硬件设备中间，也就是说，他为众多应用程序提供对计算机硬件的访问，他可以决定一个程序可以对硬件的访问程度是多少：比如一个程序可以操作某硬件设备多长的时间。不管是对于什么层面的user来说，直接对硬件进行操作都是很复杂的，所以内核同时提供了一种抽象的方法，来帮助应用程序更清楚简单的完成这些对硬件的操作

最标准的定义：**operating system is (the one program running at all times on the computer—usually called the kernel + system programs(which are associated with the operating system but are not part of the kernel).**

- OS是怎么开始的?


How does an Operating System starts ?

↳ 为了 launch 一个 OS，kernel 就需要被 loaded

⇓ loaded by

boot strap loader

⇓ loaded by

BIOS (Basic input / output system) <=> UEFI (Unified Extensible Firmware Interface)

→ BIOS 预安装在个人电脑的主板上，是个人电脑启动时加载的第一个软件，UEFI 是 BIOS 的"继任者"，所以功能会是基本一样

顺序：

电脑开启

↳ CPU 的 Instruction register 先加载一段预定义的 memory location

↳ 成功加载 bootstrap loader

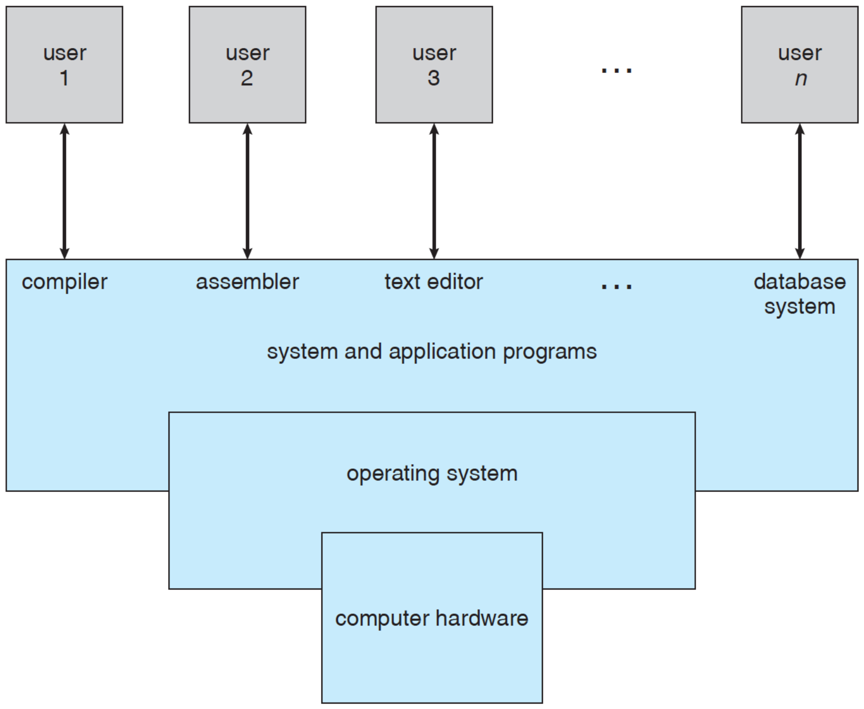
↳ bootstrap loader 把 kernel 加载进 memory and 执行

- 是先加载kernel→再从kernel里面执行第一个进程从而把其他（CPU register，device controller 啥的）给初始化（waits for some event to occur, 而event的发生往往是signalled by an interrupt，interrupt可以看Lecture 03）

Lecture 02

Hardware and software in computer systems

- 一个完整的计算机系统可以分为四个部分：User，Application programs，Operating System，Hardware



- Hardware Component
  - Central Processing Unit

Central Processing Unit: 即中央处理器, 是electronic circuit responsible for executing the instructions of a computer program

↳ 在CPU中, 存在 Register

↳ High speed storage area, 任何事物, data 在处理前都会被暂存于 Register 中

↳ MAR Memory Address Register Holds the memory location of data that needs to be accessed

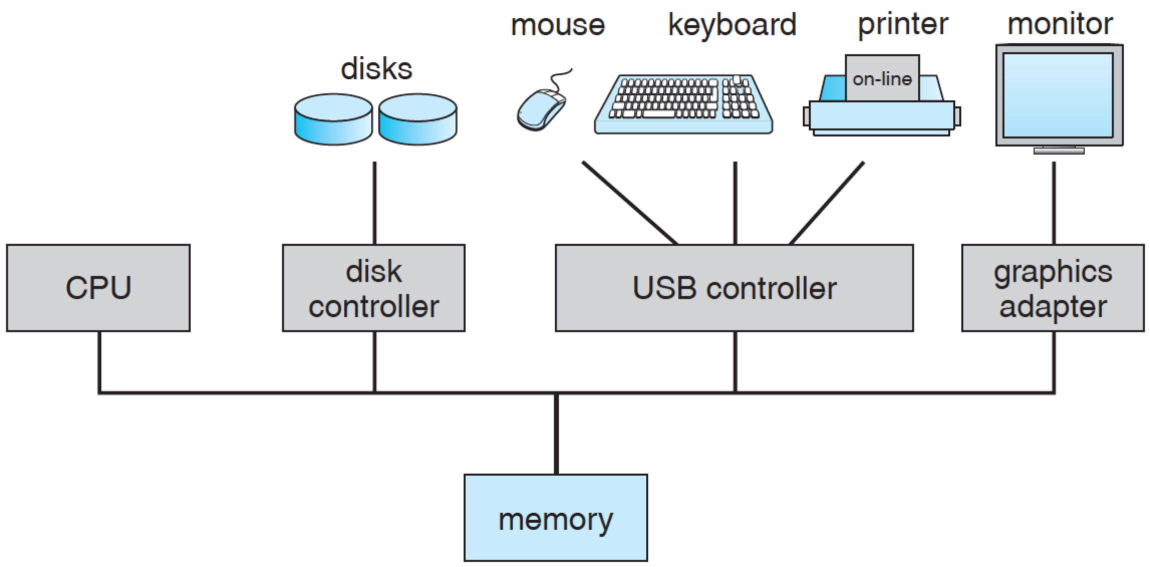
MDR Memory data register Holds data that is being transfered or from memory

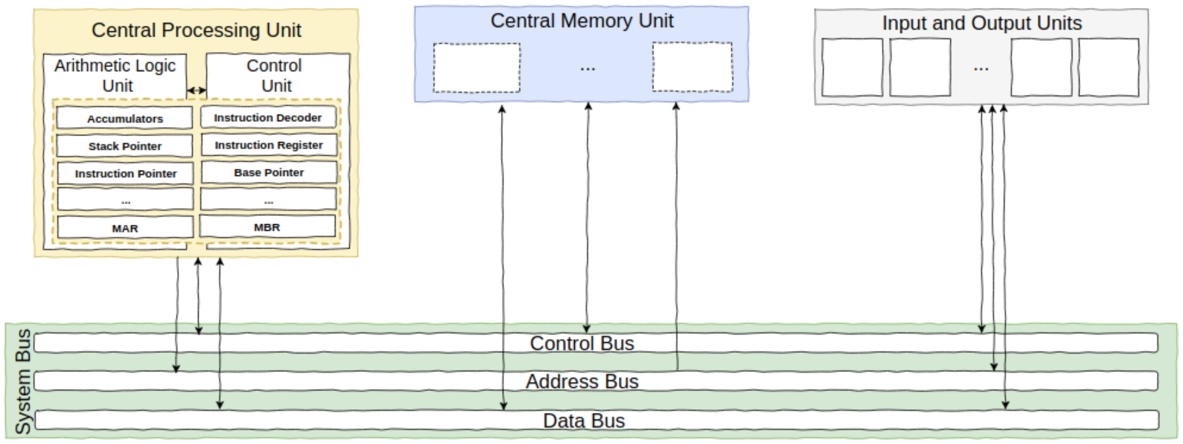
Ac Accumulator 即时arithmetic and logic result被存储的地方

PC Program counter 包含下一个要执行的指令的位置

CIR Current instruction register 包含当前执行的指令

  - Memory
    - Hardware components used to store data
      1. Volatile(关机就删)
      2. Non-volatile(长期保留)
  - I/O Device
    - As the name suggests, input/output devices are capable of sending data output to a computer and receiving data from a computer input.
  - Von Neumann Architecture Model



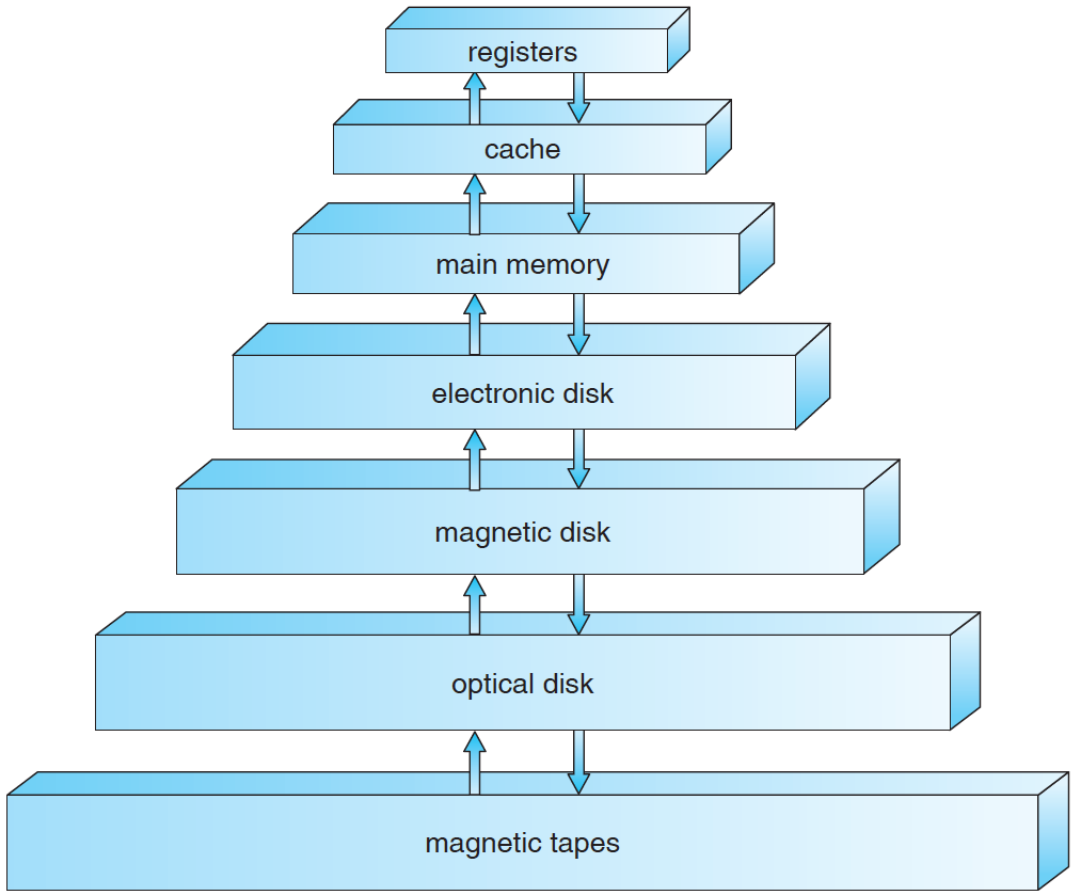


- 不同的硬件需要os的调节，从而令软件在执行的时候并不会因为相同硬件的使用二产生冲突
- 所以bus其实是“Provided access to share memory”，a set of wires that carries information from one part of a computer system to another

- Application Programs:
  - define the ways in which these resources(指的是硬件设备) are used to solve users' computing problems
- Operating System
  - **所以实际上os的作用在硬件和应用中间，他可以控制硬件，并为了应用来协调硬件的使用；而OS的目的是执行User program的同时尽可能让problem easier**
  - 最标准的定义：**operating system is the one program running at all times on the computer—usually called the kernel.**
  - 面向单一用户的计算机系统：In this case, **the operating system is designed mostly for ease of use, with some attention paid to performance and none paid to resource utilization—how various hardware and software resources are shared.**
  - 面向多用户的计算机系统：The operating system in such cases is designed to **maximize resource utilization— to assure that all available CPU time, memory, and I/O are used efficiently and that no individual user takes more than her fair share.**
  - 面对多用户但不同情景的计算机系统：In still other cases, users sit at workstations connected to networks of other workstations and servers. These users have dedicated resources at their disposal, but they also share resources such as networking and servers—file, compute, and print servers. **Therefore, their operating system is designed to compromise between individual usability and resource utilization.**
  - 移动手持计算机：Because of power, speed, and interface limitations, they perform relatively few remote operations. Their operating systems are designed mostly for individual usability, but performance per unit of battery life is important as well.
  - 镶嵌在家电设备的计算机系统：For example, embedded computers in home devices and automobiles may have numeric keypads and may turn indicator lights on or off to show status, but they and their operating systems are designed primarily to run without user intervention
- User
  - 指的就是普通的用户

Memory Unit

- Memory Unit这个小节讲的是硬件设备中的Memory
- CPU只可以从Memory中运行指令，所以所有的程序都需要被储存在Memory中，在那其中，memory又分为两种：
    - RAM，random-access memory，这个又被称为main memory，是rewriteable的
    - ROM，read-only memory，这个memory里面的东西是不可以被改变的，所以里面存着的程序都是static program
- CPU如何与memory交互：
    - `load` ： The load instruction moves a word from main memory to an internal register within the CPU从main memory把东西移入cpu的register中
    - `store` 把内容从cpu的register中移入main memory中
    - 或者是在运行什么程序的时候，cpu会自动把指令从main memory中加载进register中
    - 具体执行指令的顺序在instruction execution cycle
- 对于Memory unit来说，他能看到的只有memory address的流动
- 对于main memory来说，有两个致命的问题：空间太小无法储存所有的程序和数据，他的挥发性会令所有数据在电脑关机之后都消失，在这种前提下，secondary storage（如magnetic disk）诞生了，这种storage可以永久储存大量数据和程序
    - 所以大部分的程序实际上是被储存于disk中的，直到他们被使用时才被加载进main memory中
    - Many programs then use the disk as both the source and the destination of their processing
- 只有main memory，register和disk只是许多种的memory unit中的一种而已，大部分的storage system都尊从下面这张hierachy（根据speed and cost）的图



  - 越上层的阶级cost more but faster，越往下走：cost per bit 下降但是access time会上升
  - magnetic tape and semiconductor memory（半导体）变得越来越快且便宜前四层都可以用semiconductor来制成
  - the storage systems above the electronic disk are volatile，electornic disk既可以是volatile也可以不是，而往下走的就都是nonvolatile的了
- Uniform Memory Access & Non-uniform memory access
  - UMA: 对于多处理器系统来说，不管是哪个处理器去access main memory里面的东西所用时间是一样的
  - NUMA：顾名思义，memory中的有些部分相对access速度慢，这样就造成了表现的下降

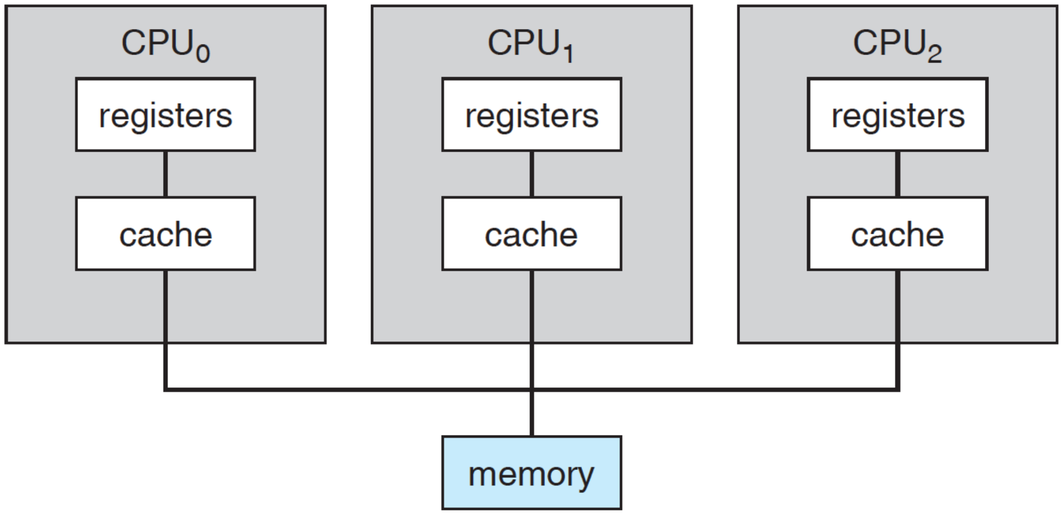
CPU Unit（Processor）*

- is complex electronic circuit designed to execute machine instructions at a very high speed.
- 有很多种不同的register
  - General purpose registers
  - Index registers
  - Segment registers
  - State registers
  - Other registers

        - MDR（Memory Data Register）
        - MBR（Memory Buffer Register）
- Processor Instruction Set（ppt上面可以直接对应）

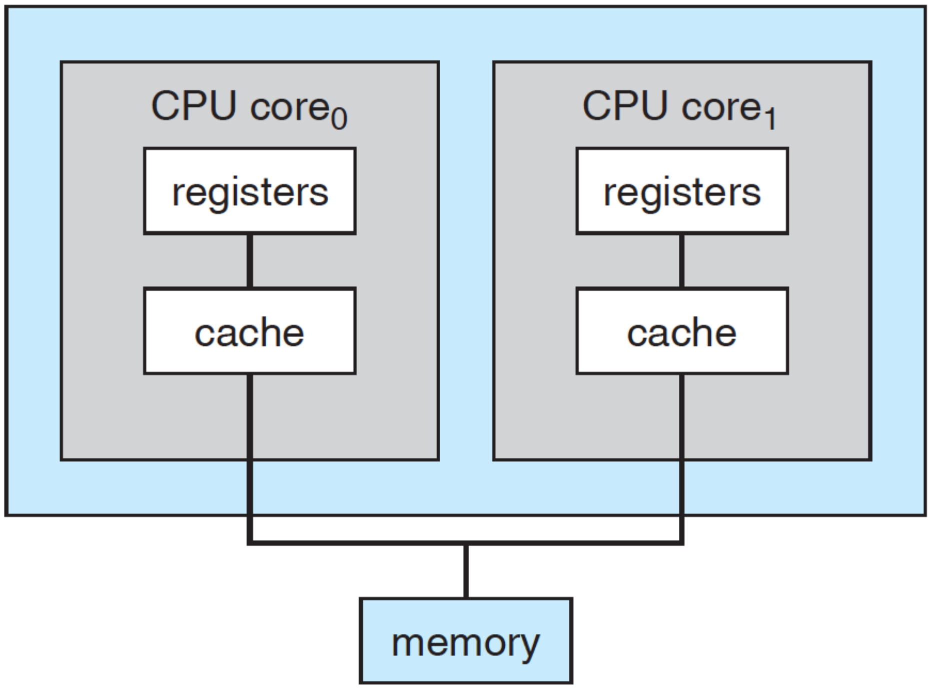
Single and Multiple-processors System

- Single-processor system
    - **If there is only one general-purpose CPU, then the system is a single-processor system.**
    - There is one main CPU capable of executing a general-purpose instruction set, including instructions from user processes.
    - 但是同时，还具备special-purpose processors，这些processors可能是具体到某种设备的处理器，比如说disk，keyboard啥的。这些special-purpose processors并不会运行user processes，一般只会接受从OS传来的指令
        - 比如：For example, a disk-controller microprocessor receives a sequence of requests fromthe main CPU and implements its own disk queue and scheduling algorithm.
        - PCs contain a microprocessor in the keyboard to convert the keystrokes into codes to be sent to the CPU.
    - 甚至有的时候，special-purpose processor是嵌在单独的设备上的，OS不会跟他进行交互，这些单独的processor会自动工作
- Multiprocessor System
    - Two types of multiprocessor system
        - AMP(Asymmetric Multiprocessing)
            - Each processor is assigned a specific task，并且存在一个master processor去control the slave processor，system书上的解释不一样
        - SMP(Symmetric MultiProcessing)



            - Each processor performs all tasks within the operating system
            - 每个处理器都是peer而非从属的关系，这样处理器的好处在于：进程可以被同时进

行，如果有n个cpu，那么n个进程可以被同时进行
        - 但是在处理这样子结构的时候要很小心，所以os要写的很好
    - 也被称为Parallel System 或 Tightly coupled system，have two or more processors in close communication, sharing the computer bus and sometimes the clock, memory, and peripheral devices.
    - 主要有三个优势：
        1. Increased thrughput（增加吞吐量）：但是同时，因为多处理器会带来更多的工作（为了维持多处理器之间正确的调度资源），所以新增N个处理器并不可以提高N倍的速度
        2. Economy of scale：比起单处理器系统，多处理器共享一个Memory，这样子的话储存的成本会变低
        3. Increased Reliability：因为有多处理器，所以如果工作能被正常分配的话，即使其中一个处理器出问题了，也不会导致完全崩溃
            - 比如说有十个处理器，一个处理器崩了，那么这个处理器本身要处理的进程将会被平局分配至剩下的处理器中，这样的技术被称为graceful degradation，或者还有别的技术叫做fault tolerant，"can suffer a failure of any single component and still continue operation"、
                - 比如hp有个很浪费的：每个进程/指令都由两个cpu来执行，并且每次都会对比结果，如果结果不一样的话，说明有一个cpu出问题了，两个cpu都会停止，会将这个指令移至下一对cpu去执行
- 多核处理器（多个处理器嵌在一个chip上）比多个单核处理器速度要快很多，并且成本也低很多，下面是一个例子

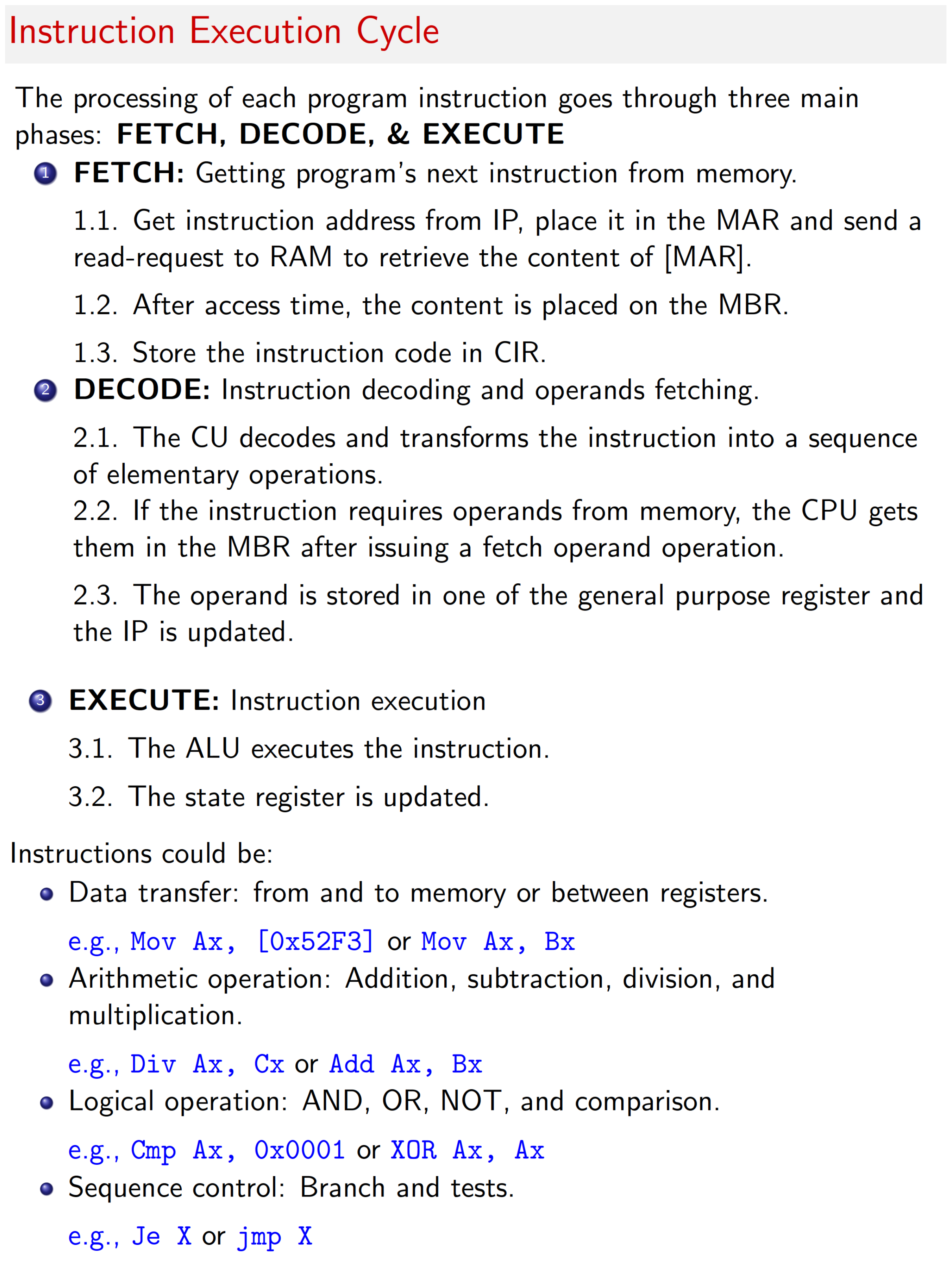


- Clustered System

    - that clustered computers share storage and are closely linked via a local-area network (LAN) (as described in Section 1.10) or a faster interconnect, such as InfiniBand. 许多单独的计算机系统通过本地网络连接
    - 提供high-availability service：即使一个或多个计算机系统塌了，服务仍然会进行
    - 提供high-performance computing environment，这样的系统的计算能力比single/multiple processor系统都要强，因为他可以同时在多个计算机系统运行一个程序
        - Parallelization是一种令程序能够适应这种运行条件的技术：which consists of dividing a program into separate components that run in parallel on individual computers in the cluster
    - asymetric clustering*
    - symmetric mode*
    - parallel cluster*
    - distributed lock manager*
    - storage-area networks*
- blade servers*

Instruction Execution Cycle

- 这是一个von neumann architecture的电脑会执行指令的顺序
    - 比较详细的看ppt
    1. **Fetches** an instruction from memory and stores that instruction in the instruction register
    2. The instruction is then **decoded** and may cause operands to be fetched from memory and stored in some internal register
    3. Instruction on the operands being **executed**
    4. The result may be stored back in memory



Instruction Execution Cycle

The processing of each program instruction goes through three main phases: **FETCH, DECODE, & EXECUTE**

1. **FETCH:** Getting program's next instruction from memory.

   1.1. Get instruction address from IP, place it in the MAR and send a read-request to RAM to retrieve the content of [MAR].

   1.2. After access time, the content is placed on the MBR.

   1.3. Store the instruction code in CIR.

2. **DECODE:** Instruction decoding and operands fetching.

   2.1. The CU decodes and transforms the instruction into a sequence of elementary operations.
   2.2. If the instruction requires operands from memory, the CPU gets them in the MBR after issuing a fetch operand operation.

   2.3. The operand is stored in one of the general purpose register and the IP is updated.

3. **EXECUTE:** Instruction execution

   3.1. The ALU executes the instruction.

   3.2. The state register is updated.

Instructions could be:

- Data transfer: from and to memory or between registers.

  e.g., `Mov Ax, [0x52F3]` or `Mov Ax, Bx`
- Arithmetic operation: Addition, subtraction, division, and multiplication.

  e.g., `Div Ax, Cx` or `Add Ax, Bx`
- Logical operation: AND, OR, NOT, and comparison.

  e.g., `Cmp Ax, 0x0001` or `XOR Ax, Ax`
- Sequence control: Branch and tests.

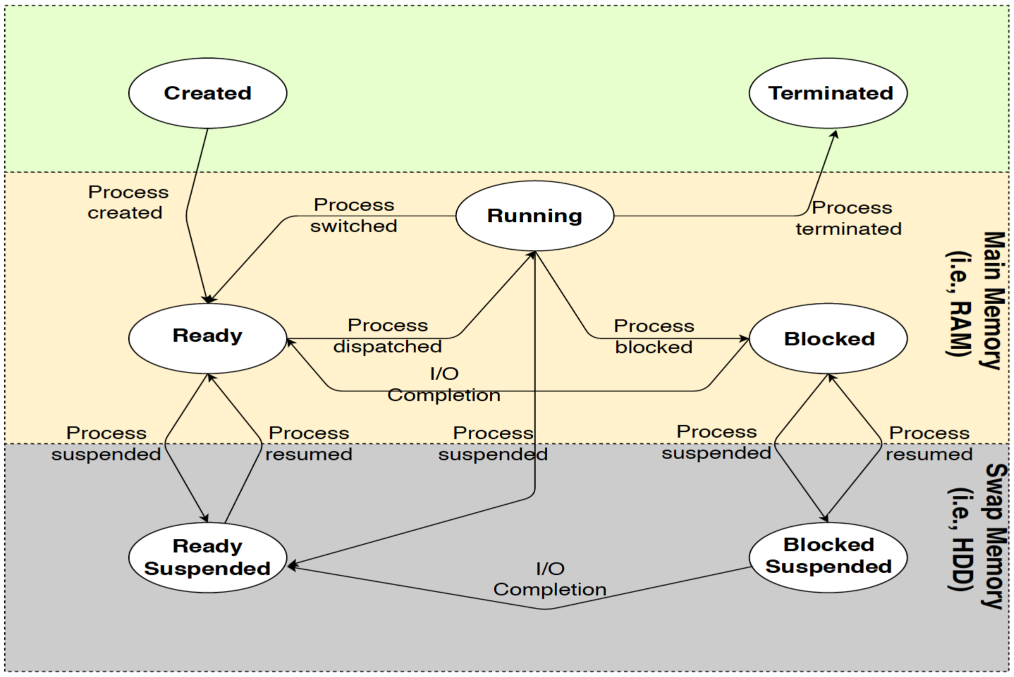
  e.g., `Je X` or `jmp X`

可以补充看看I/O structure pg12*

Lecture 03 & Lecture 04 & Lecture 05

Processes（进程）

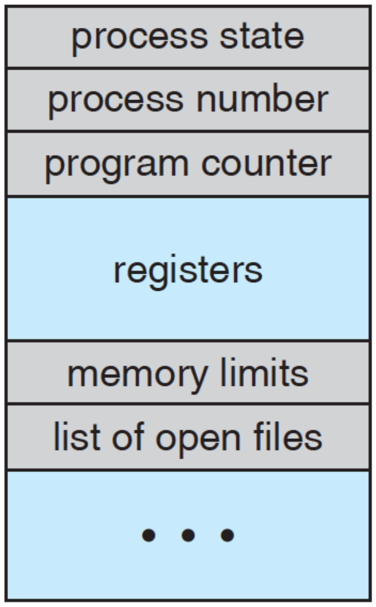
- 什么是process?
    - 现在的电脑系统和以前的电脑系统不一样：曾经的电脑系统每次只允许一个程序运行，而现在的电脑系统允许多个程序同步运行，进程指的其实就是“a program in execution”, a process is the unit of work in a modern time-sharing system
    - process和程序的区别：
        - 程序只是一个passive entity，就是一个存在磁盘上的里面有指令的文件而已
        - Process是一个active entity，他是在活动着的，with a program counter specifying the next instruction to execute and a set of associated resources，当程序被load进memory中的时候，换言之，当程序在运行的时候，这个程序就变成了一个进程
- process里面有什么
    - An address space（containing the program code and data）
        - static
        - heap，is memory that is dynamically allocated during process run time
        - stack，which contains temporary data (such as function parameters, return addresses, and local variables)
        - data section：global variables
    - The cpu state：The value of cpu registers including pc and sp
    - A set of os resources: open files, network connections,...
    - ??A lifetime: created-executed-[interrupted-resumed]-terminated
- 有什么种类的进程：
    1. operating system processes executing system code
        - executed in kernel mode（master mode）
    2. user processes executing user code
        - executed in user mode（slave mode）
- 即使是由同一个程序产生的两个进程，也是完全不一样的哦
- Process state



- 上面这张图罗列了一个进程能拥有的state
  - New：The process is being created
  - Ready：Having all needed resources, The process is waiting to be assigned to a processor
  - Running：Instruction are being executed
  - Blocked：Waiting(blocked queue) for another process to terminate or an event to happen(e.g. signal reception)
  - Suspended：Swapped out of RAM for some reason(具体的reason看ppt)
  - Terminated：The process has finished execution or has been killed
- Process IDentifier（PID）：During the lifetime of a process, it's identified by a unique(temporary) number called PID
- **Zombie process**, it is a process which has terminated (executed exit()), however, its parent process has not yet executed the wait() system call to receive the notification about the termination of its child. It is like the process has terminated (dead) but it still has an existance entry in the system as long as the parent has not yet executed wait() or terminated (in a nutshell, the process is dead but still alive).

PCB(Process Context Blocks)

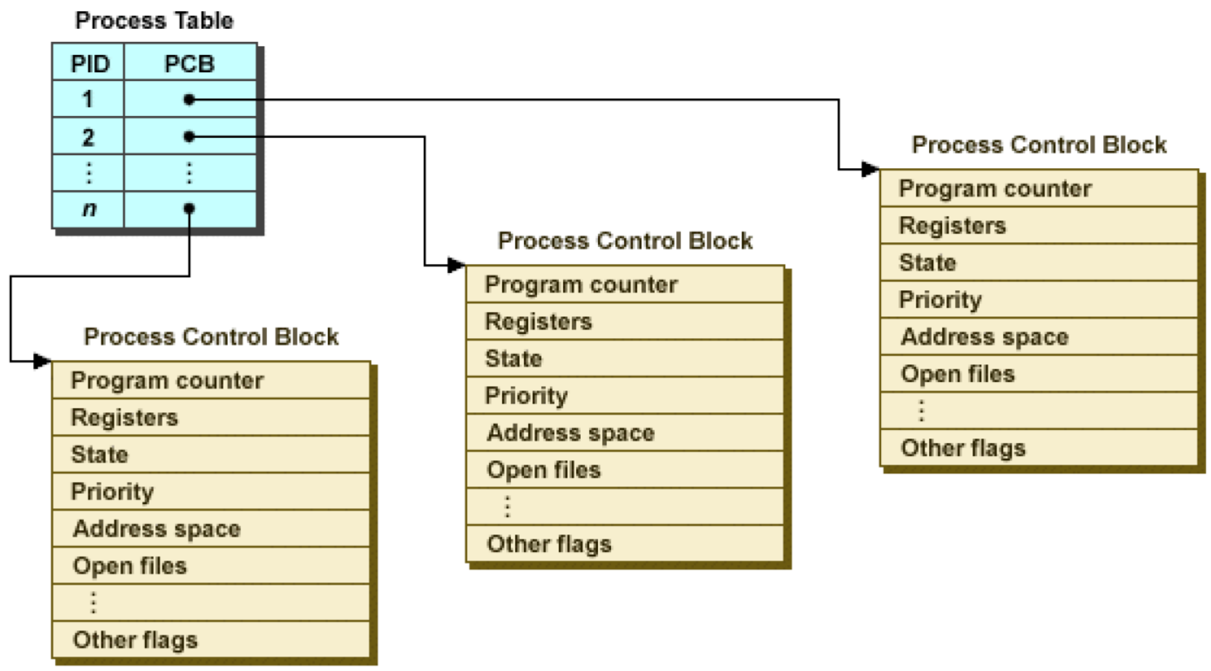
- 这个东西存在的目的之一是可以在interrupt之后回来，追踪进程
- 在操作系统中，每个进程都被PCB（Proces control blocks）所代表，这个也被称为task control block



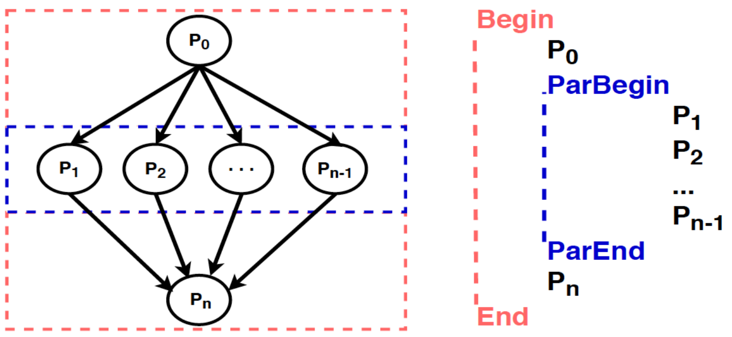
上面是一个PCB，里面包含的值分别是：

- Process state: 这个process的last state？？目前所处于的state
- Program counter：The counter indicates the address of the next instruction to be executed for this process
- Registers：指的是这个进程使用的register，和上面的program counter一样，如果有interrupt出现的话，这个register和pc的值必须要被保存下来以方便从interrupt回来之后可以继续执行刚刚的指令
- CPU-scheduling information：包含了process priority，pointers to scheduling queues，and any other scheduling parameters
- Memory-management information：This information may include such information as the value of the base and limit registers, **the page tables, or the segment tables***, depending on the memory system used by the operating system (Chapter 7).
- Accounting information: 就是关于这个进程占用了多少cpu的资源啊，用了多长时间啊，PID
- I/O status information：储存了诸如什么I/O设备被使用了，打开了什么文件

- 下图是另外一张图，可以看到不同的processes的PCB都会被储存在Process Table中



- 具体的一些代码上的东西看lab_1
- Concurrent Process：
    - 当多个程序被load进main memory的时候，也就是有多个进程同时执行的时候，这个时候他们的运行要么是parallel或者是fake-parallel? ？
    - 他们执行的时间是重叠的，一次执行要么是sequential（serial）或者是parallel的
    - Process Precedence Graph（PPG）：用这样的图可以表示进程的执行



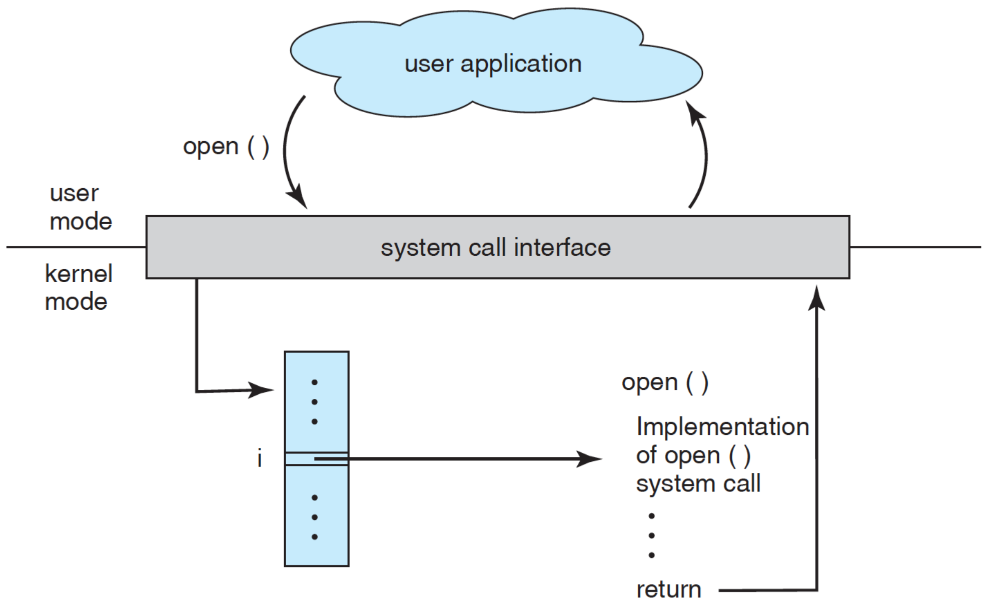
Processes Precedence Graph (Left) and its pseudocode (Right)

- 当出现了多进程之后，进程之间就可以合作，他们的合作有许多的有点，具体看ppt
- Process Scheduling/Scheduler*

Interrupt

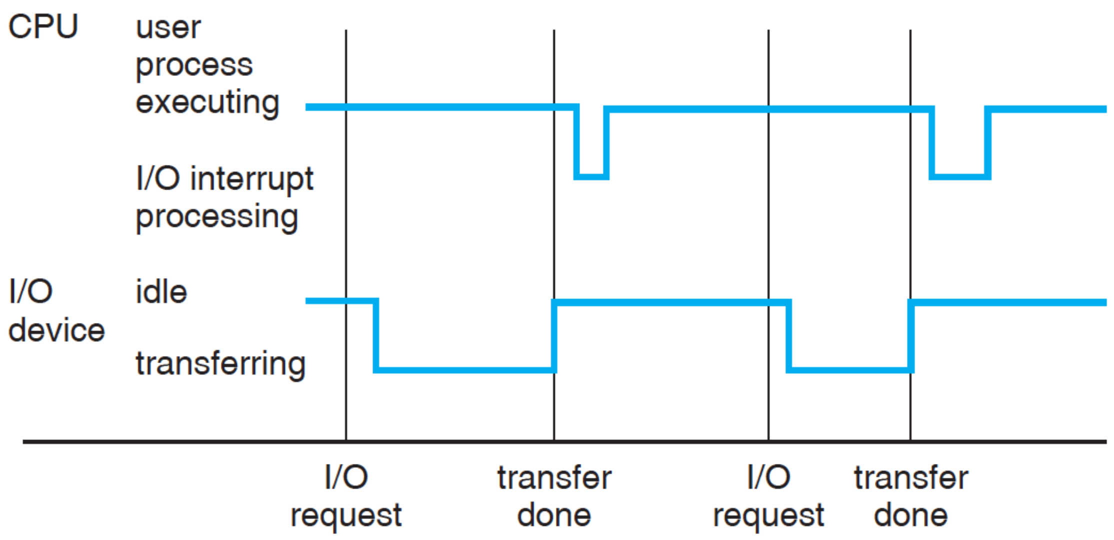
- is an event that alters the sequence of instructions executed by the CPU 改变CPU原来的运行顺序
    - From Hardware（Hardware-based）：

        - Hardware may trigger an interrupt at any time by sending a signal to the CPU, usually by way of the system bus.
    - From Software(Software-based)：
        - Software may trigger an interrupt by executing a special operation called a system call (also called a monitor call).
            - **Exceptions**: An exception happens when the current instruction perform an illegal action such as division by 0
            - **System calls**: User program are executed in user mode. This program doesn't have any access to peripheral(外围), 所以这样子的程序必须要issue一个system call去得到interrupt
                - 什么是system call；System calls provide an interface to the servicesmade available by an operating system
                - 我们为什么需要system call? ：一个例子：当我们写一个程序去从一个文件读取并且复制数据到另一个文件的时候，我们的程序首先需要的第一个input必然是这两个文件的名字，我们可以直接问user，当是在一个交互式的系统中，首先这个程序需要在屏幕上显示message“请输入文件名”，然后获得键盘的使用权限从而读取user在键盘上输入的东西，这些全都得靠system call来完成
                - 什么是**API（Application Program Interface）**：The API specifies a set of functions that are available to an application programmer, including the parameters that are passed to each function and the return values the programmer can expect



                    - 比如说当我们使用Win32的函数`CreateProcess()`的时候，我们实际上是在调用`NTCreateProcess()`的system call
                    - 那么为什么要使用API的指令而不是真正的system call呢?
                        - 首先不同的系统有不同的system call，用API指令可以统筹起不同名字但作用相同的system call

        - 真正的system call比api难用
      - System call的种类：
        1. Process Control
        2. File management
        3. Device Manipulation
        4. Information Maintenance
        5. Communication
        6. Protection
  - Inter-processor interrupts
  - Spurious Interrupts
- 什么是Interrupt storm：就是当os接收到了太多的从hardware过来的interrupt，导致handler运行太长时间，那么这个os就会处于危险当中
- 不管是哪种interrupt，CPU在接收到了之后都会马上停止目前所有的工作，并且把指令的执行迅速转移到另一个固定的位置。这个位置中存在着Service Routine，这个**Interrupt Service Routine**(a low level program (assembly)that is executed in kernel mode to service the cause of the interrupt)会执行，执行完了之后，CPU会回到这个interrupt之前的执行的位置，下图表示的是运行的一个顺序

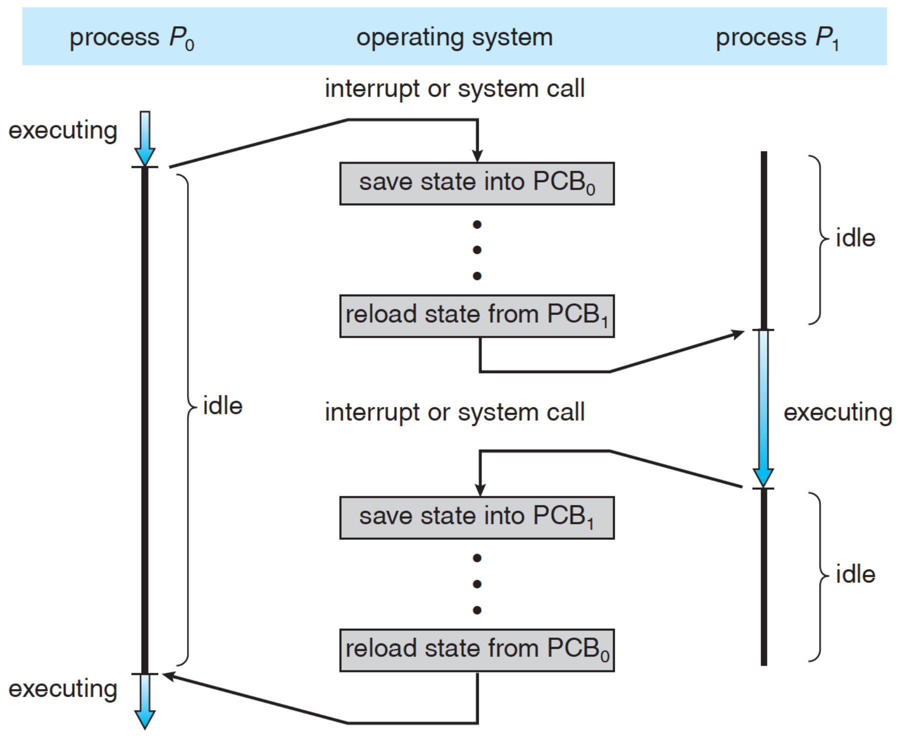


- Interrupt是怎么把控制转移给interrupt service routine?
  - Invoke a generic routine to examine the interrupt information
  - 然后这个routine会根据这个interrupt information去call具体的interrupt handler
- Interrupt怎么保证handle的速度？(**Interrupt Vector Table**)
  - 因为interrupt总共也就那么几种，所以一个table of pointer to interrupt routine can be used
  - 在执行interrupt的时候，会直接找到这个**Interrupt Vector Table**然后从具体的那个pointer那儿去向具体的interrupt service routine
  - 这个table一般存在low memory(the first hundred or so location)，而在这个table里面，每个设备的interrupt routine都会被给予一个具体的值
- Interrupt怎么回到之前运行着的程序?
  - 为了回到之前运行的程序，必须把interrupt之前的执行的地址给保存下来，以前的设计一般是把interrupt之前的地址放在一个固定的位置，但是现在的设计更多的是store the return address

on the system stack
    - After the interrupt is serviced, the saved return address is loaded into the program counter (Lecture 02可以找到), and the interrupted computation resumes as though the interrupt had not occurred.
- 如何将硬件事件(键盘按下、传入网络包、鼠标移动……)升级为正在运行的程序?
    - Polling*：操作系统定期检查每个硬件看看new information是不是available（(simple but high latency and CPU cycle wasting）
    - Interrupts: 设备向Interrupt controller发信号从而引起attention，CPU用进程去处理device request，interrupt到达cpu到cpu执行routine之间的时间间隔被称为interrupt latency
        - 当每一个指令执行完之后，cpu都会检查interrupt controller去看看有没有interrupt，如果有的话，那么现在的进程将会被“suspended”，并且执行handler
- **Interrupt Handlers $\neq$ Processes**

Context Switching

- 什么是context?
    - 当Interrupt出现的时候，the system needs to save the current context of the process running on the cpu so that it can restore that context when the handler is done. 所以context就是一些关于目前的process的信息，需要把这些信息记录(state save to the process PCB)下来从而恢复(state restore)这个process
- context里面有什么东西?
    - the value of the cpu registers
    - the process state
    - memory-management information
    - 以上的内容不知道是什么的话可以参考PCB
- Context switch（这个指的单纯是进程和进程之间的转换，而不是handler和进程之间）：Switching the cpu to another process或者可以认为是以下操作：
    1. state save of the current process(save the context of current process into its PCB)
    2. state restore of a different process(load the context of the newly scheduled process from its PCB)
- Context switch（switch between processes的图示）

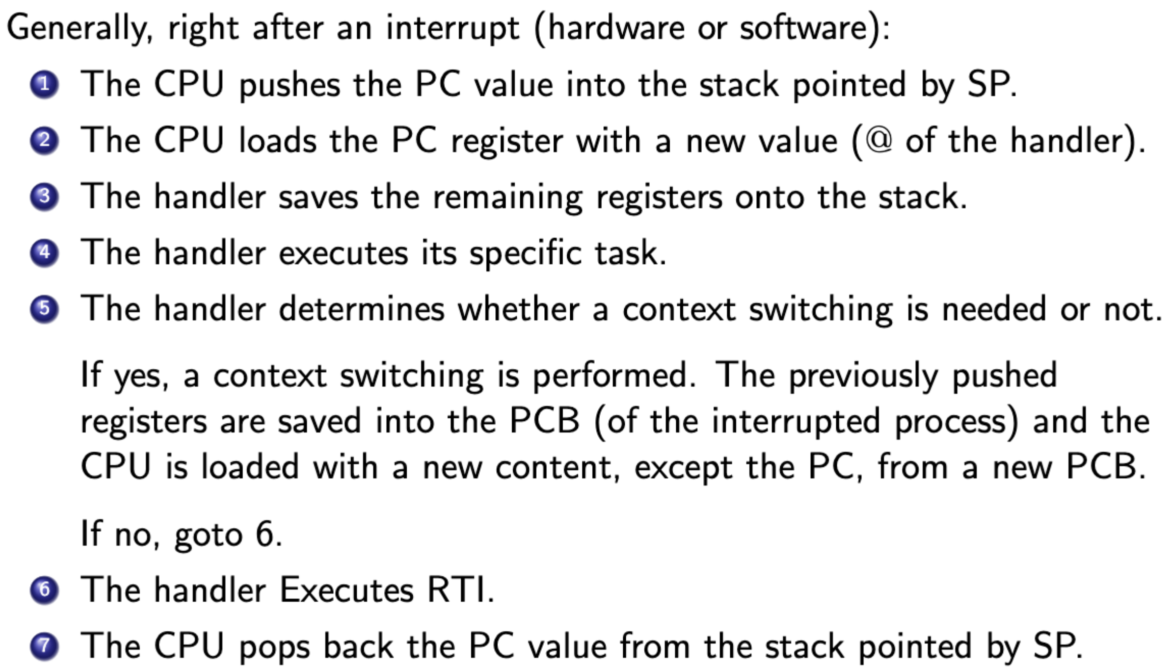


- 当context switch的时候，switch的时间纯粹是白费的，因为这时候系统无法进行其他的工作，一般是few milliseconds
- Interrupt with Context switching（P1是正在执行的进程，P2是下一个进程，这是一个interrupt出现了，会怎么样）

  Assuming that process P_1 is executing and process P_2 is the next process in the ready queue. Then, if an interrupt occurs:

  1. CPU (hardware) pushes the current PC onto the stack and updates the PC-register to contain the address of the 1^{st} instruction in the corresponding interrupt handler.
  2. The interrupt handler starts execution:
     - It pushes all remaining registers onto the stack then performs its specific task.
     - It copies all registers values from the stack into PCB1.
     - It move PCB1 to end of ready/blocked queue, and PCB2 into the head.
     - It copies register values from PCB2 into CPU-registers, except the PC values which get copied to the top of new stack.
  3. The handler executes RTI (ReTurn from Interrupt) which causes the CPU (hardware) to pop the PC value from the new stack to CPU's PC register.
  4. The CPU then, starts/resumes the execution of process P_2.

  - 以下是一个更加直白general的图示：


Generally, right after an interrupt (hardware or software):

1. The CPU pushes the PC value into the stack pointed by SP.
2. The CPU loads the PC register with a new value (@ of the handler).
3. The handler saves the remaining registers onto the stack.
4. The handler executes its specific task.
5. The handler determines whether a context switching is needed or not.

   If yes, a context switching is performed. The previously pushed registers are saved into the PCB (of the interrupted process) and the CPU is loaded with a new content, except the PC, from a new PCB.

   If no, goto 6.
6. The handler Executes RTI.
7. The CPU pops back the PC value from the stack pointed by SP.

- 为什么context switch对于一个time sharing system来说很重要？看ppt
- Interrupt interrupts interrupt handlers
  - 如果在执行handler的时候一个更高级别的interrupt出现了，那么现在在执行的handler就会被interrupted，具体会怎么搞看ppt

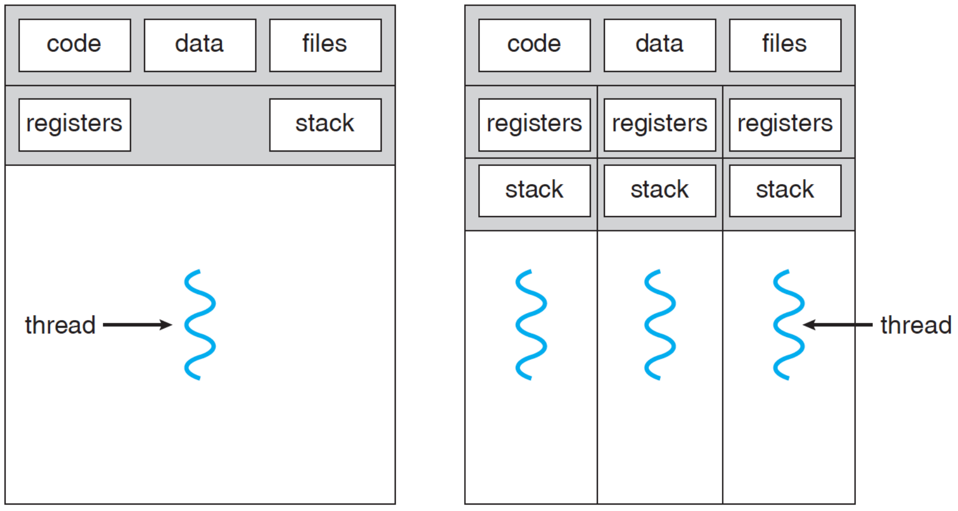
Lecture 06

Threads

- Thread: **A thread is lightweight process**. It is an execution flow of a given program. A programmay have multiple execution flows (multi-threaded program).
- A thread shares with the main process (as well as with other threads from the same process) the code section, global data section, heap section, as well as the resources such as open files. Yet, it has a private stack section and private CPU-register values.线程属于一个进程，共享这个进程中的很多内容，但是他仍然有一个属于自身的private stack section and private cpu-register values
- 和process一样，thread是被TID（thread identifier）给区分开来的，并且有一个跟PCB一样的结构
  - Thread VS Process
    - 如果想要多线工作的话，为什么不直接创建新的进程呢?
      - 每次创建新的进程的时候都会分配一个新的memory location，并且把现在的东西都复制过去，time consuming，resource intensive，所以更高效的办法是在一个主进程里面布置多个线程
    - 当创建新的进程的时候，计算机会复制现有的PCB，并且为新的进程分配新的memory location，把现有的线程的所有东西都复制进新的memory location中，并且update the PCB data fields

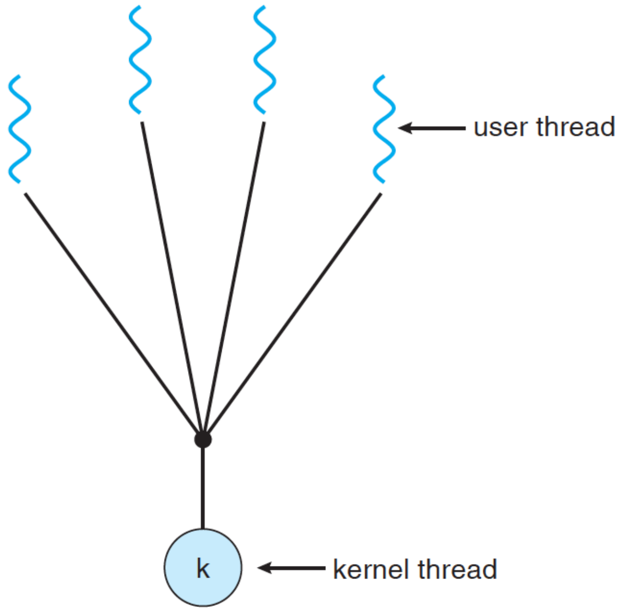
        - 而当创建新的线程的时候，也会有新的memory location，只不过这个memory location中存的只是他自身的private stack section and private cpu register values，对于其他运行需要的内容会直接回主进程里面拿

- single-threaded process VS multithreaded process

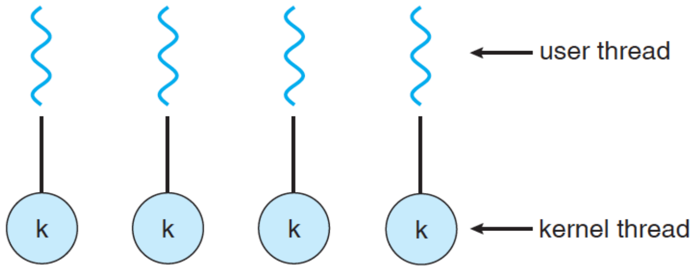


single-threaded process multithreaded process

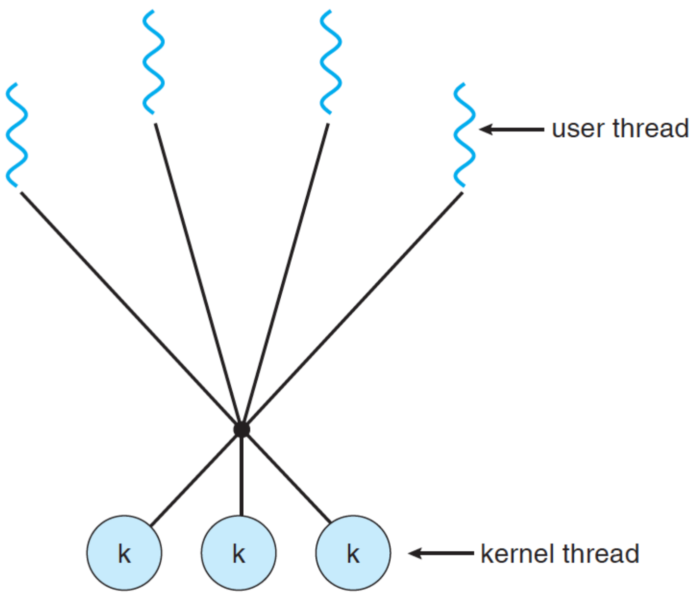
    - 现在大多数的os kernels都是多线程的了
- advantage of thread：
    - responsiveness：即使一个线程被搞了，另一个线程依然可以单独工作
    - resource sharing：
    - economy：不用分配一个完全独立的memory space(Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads.)
    - scalability：对多核处理器的最大利用
- 在单核处理器系统里面运行多线程应用一点意义都没有
- Thread level:
    - User-level thread: supported **above** the kernel and are managed without kernel support
    - Kernel-level thread: supported and managed directly by the operating system
    - 在管理user thread和kernel thread的时候有三个常见的模型
        - Many to one：



    - 多个用户进程连接到一个kernel进程，因为每次只有一个user thread可以access到一个kernel thread，所以它实际上不可以在多核处理器中平行工作
- One to one：



    - 很straight forward：这个的优势在于它可以真正的令多线程在多个处理器上工作，但劣势也是相当明显：每创建一个user thread就必须随着他创建一个kernel thread，而创建kernel thread是会对程序的表现造成影响的
    - linux和微软的操作系统都是实现了这样子的模式
- Many to many：



- 把M个user thread和N个kernel thread相匹配such that$N \leq M$
- 这个东西就很厉害，因为user可以创建他所需要的数量的线程，并且程序的表现不会被太大的影响，并且还可以在多核处理器上面平行运行

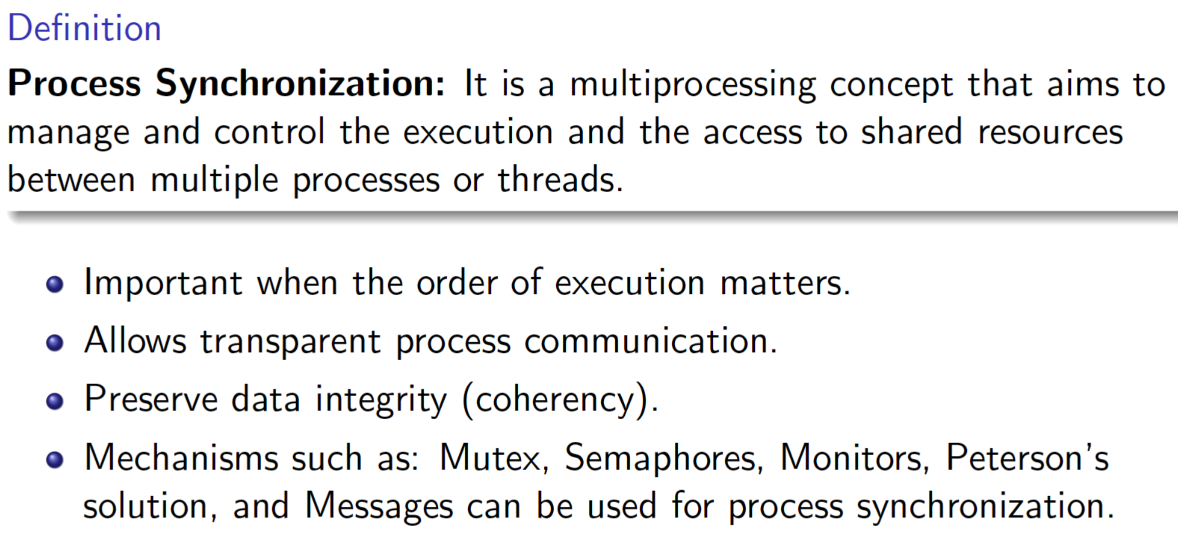
DMA

Lecture 07 & Lecture 08 & Lecture 09

Introduce the critical section problem and process synchronization

- 进程/线程之间如果share同一个data的话可能会带来data inconsistency，这里面要讲的是various mechanism to ensure the orderly execution of cooperating processes that share a logical address space
- 当一个会引发race condition的行为执行的时候，他的结果取决于：
  - **The logic of your program**
  - Depends on which thread is started first.
  - Depends on which thread has the higher priority
  - Depends on the hardware, each thread is running on one CPU core
- 为什么会出现“shared data corruption”：
  - Processes can execute in (fake) concurrency i.e., any process can be interrupted at any point in its instruction flow (e.g.,quantum expires or I/O operation), and the CPU is assigned to another process.
  - Processes can execute in (real concurrency) parallel (e.g., two instruction flows of two different processes are simultaneously executing on separate processing cores).
  - These concurrent or parallel processes are sharing some data.
  - **The concurrent or parallel execution of multiple processes may result in the corruption of shared data by among several processes.**

- **race condition**: where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a race condition.多个进程同时取得和改变同一个数据，而这数据的最终结果则则取决于实际运行时进程获取他的顺序。为了保证不发生这种状况，需要保证每次只有一个进程可以改变这一数据 **To make such a guarantee, we require that the processes be synchronized in some way.** 所以**synchronization**就是一个保证每次只有一个进程可以修改一个shared变量的机制
  - ppt上的解释（和书一模一样只不过paraphrase了一下）：
    1. Multiple processes are being executed
    2. These processes share at least a common variable
    3. The outcome of the execution depends on the order in which the processes modified the shared variable



Definition

Process Synchronization: It is a multiprocessing concept that aims to manage and control the execution and the access to shared resources between multiple processes or threads.

- Important when the order of execution matters.
- Allows transparent process communication.
- Preserve data integrity (coherency).
- Mechanisms such as: Mutex, Semaphores, Monitors, Peterson's solution, and Messages can be used for process synchronization.

- ppt上讲了几个synchronization with boolean的例子
- **the critical section problem**：
  - Assumed such system existed：The system consisting of n processes $\{P_0, P_1, \dots, P_{n-1}\}$. Each processes has a segment of code called ***Critical Section***, critical section里面的代码会干的事情诸如改变common variable，update a table，writing a file等等。在这个系统里面，**最重要的特征是**：当一个进程在运行他自己的critical section的时候，其他剩下的所有进程都不可以运行（自己的critical section）：**<u>no two processes are executing in their critical sections at the same time</u>**
    - **Critical section**的标准定义是part of a program code in which the program requests to use shared resources on which the access is mutually exclusive
    - Note: A process might be interrupted during its critical section
  - critical section problem是为了设计一个protocol从而令进程可以相互合作。每个进程在执行他自己的critical section之前必须先request permission。这样的设计的话每个进程中的代码都可以被分为四部分：
    1. entry section：获取permission
    2. critical section
    3. exit section: the process inform the OS of leaving the critical section to wake up any

   waiting processes
4. remaining section

```
do {

    entry section

        critical section

    exit section

        remainder section

} while (TRUE);
```

- 对于这样子设计的程序必须遵守以下三个要求：
  - Mutual exclusion：当一个进程在执行自身的critical section的时候，其他的进程都不可以执行critical section
  - Progress：如果当前并没有进程在执行critical section，那么就会有其他的进程申请执行critical section，只有没有处于remainder section的进程可以参与决定谁去执行critical section，这个决定的过程是不可以被推迟的
  - Bounded waiting：当一个进程作出了进入critical section的请求但是还没有被批准之前，存在一个次数上限制去限制别的进程被允许运行critical section
- kernel-mode processes也会面临race condition的问题*
  - Preemptive kernels
    - 这样子的模式允许一个在kernel mode执行的进程被优先
  - Nonpreemptive kernels
    - 和上面的相反，“不允许”，a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU
    - 这样子的设计可以避免race condition，因为在kernel里面每次只可以运行一个kernel process

Introduce software and hardware solutions for critical-section problem

Semaphore

- a synchronization tool
- A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait() and signal(). 信号量S是一个int的变量，除了一开始被initialize这个过程之外，这东西只被wait()和signal()这两个atomic operation给access

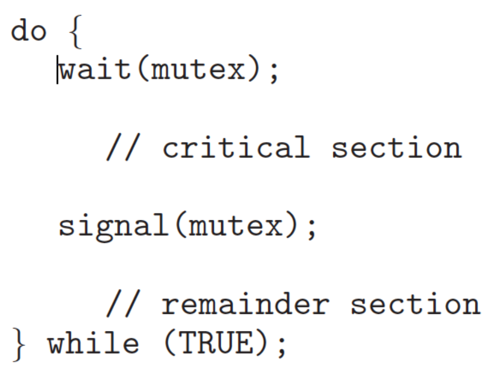
- **Atomic operation**："原子操作(atomic operation)是不需要synchronized"，这是多线程编程的老生常谈了。所谓原子操作是指不会被线程调度机制打断的操作；这种操作一旦开始，就一直运行到结束，中间不会有任何 context switch 。把wait和signal设置成原子操作可以确保semaphore不会被同时access到，如果能被同时access到，那就相当愚蠢了

```
wait(S){
  while S <= 0
    ;
  S--;
}
```

```
signal(S) {
  S++;
}
```

  - when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.
  - in the case of wait(S), the testing of the integer value of S (S ≤ 0), as well as its possible modification (S--), must be executed without interruption.

- **Counting semaphore**
  - 这个的值可以在一个没有限制的值域
  - can be used to control access to a given resource consisting of a finite number of instances. The semaphore is **initialized to the number of resources available**. 也就是说，当一个进程在占用一个资源的时候，这个信号量（available资源的总数）-1，而当他完成了之后，信号量+1，也就是说，当信号量=0的时候，当前没有可用资源，所以检测到信号量为0的进程不会执行critical section
- **Binary semaphore（Mutex Locks）**
  - 值永远处于0-1
  - 前面的板块提到的critical section就可以利用这种技术：



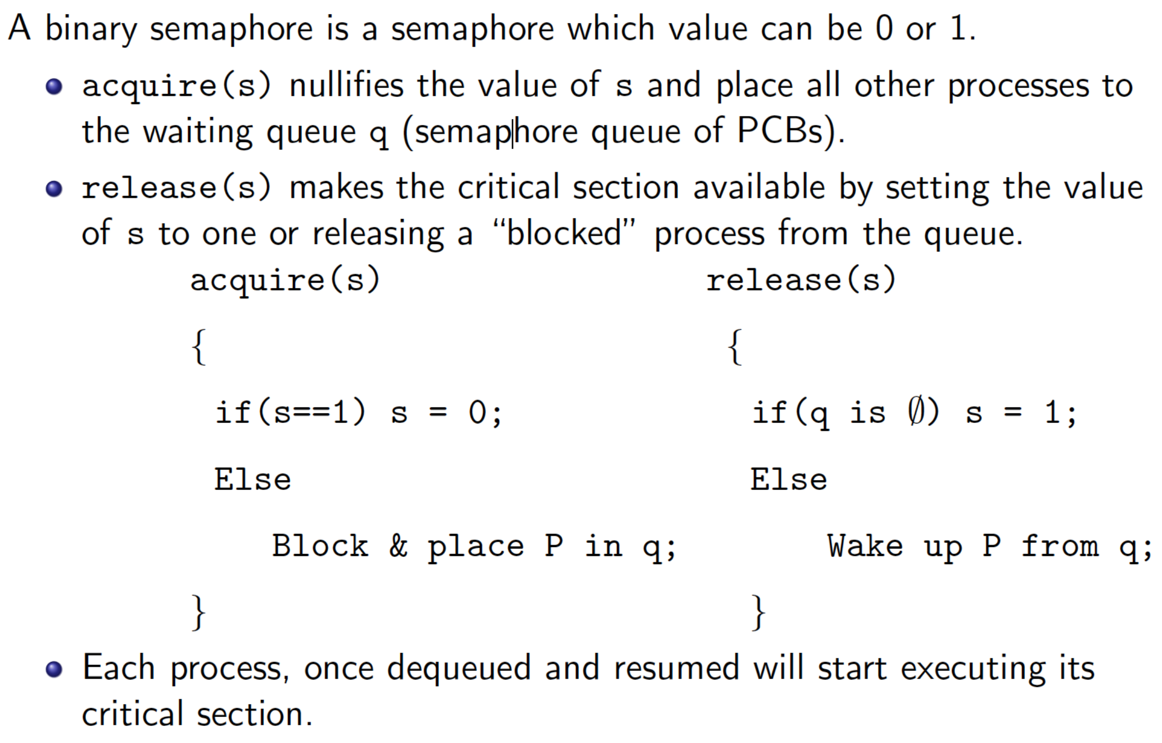
```
do {
   wait(mutex);

      // critical section

   signal(mutex);

      // remainder section
} while (TRUE);
```

  - 也就是说，当一个进程在使用一个资源的时候，这个mutex的大家都可以看到的值是0，而当mutex是0的时候，其他进程无法执行critical selection，当占用资源的进程结束了他的critical section之后，mutex变回1

- 不论是counting semaphore还是binary semaphore，他们的行为都会涉及**busy waiting**，busy waiting（忙碌等待）值得是在运行那段while遍历，意即他的等待阶段，他在等待的时候，实际上是在占用着宝贵的cpu cycle，这些cpu cycle本身是可以拿来更加高效的干些别的。这些“process在等待的时候spin”的semaphore也被称为**Spinlock**。他的优势在于并不需要context switch，while context switch很显然会浪费时间
- ***Mutex和Binary Semaphore的区别***：Binary semaphore的标准定义是范围从0-1的之间的信号量，但是Mutex lock的定义是一个互斥锁，他解决的问题是让两个想占用同一个资源的进程互斥，但是在语义上是有差别的，因为互斥锁管理的是资源的使用权：我有这个互斥锁，所以我可以用，你没有，所以你要等我给你了你才能用；而信号量管理的是资源的数量，我在进入critical section之前，ok现在信号量是10，还有十个可用资源，我拿了一个，信号量对你来说变成了9，你接着拿，当他拿的时候，只剩下0个了，那他去queue等一等，并且先宣布对下一个释放的资源的使用权：信号量变成了-1。所以互斥锁和信号量是有本质语义上的区别的。
- 针对Spinlock的改进：
    - 当进程检测到目前的信号量不为正（不可执行）的时候，之前的做法是让他不停的在while中进行等待并检查；instead of that，现在在检查到不为正的时候，这个进程可以把自己给*block*掉
    - 当执行了block这个指令之后，会把这个被block的进程协同semaphore放入一个waiting queue中，然后控制会回到cpu scheduler并且找到下一个进程
    - 在一个进程完成后（执行了signal指令），被block掉的进程会被重新开始（`wakeup()`），然后这个被block掉的state会变成ready state
    - Note：(The CPU may or may not be switched from the running process to the newly ready process, depending on the CPU-scheduling algorithm.)
    - 在这种情况下的定义：

```
typedef struct{
  int value;
  struct process *list; /*When a process must wait on a semaphore, it is
added   to the list of processes.*/
}semaphore;
/*-------------------------------------------------*/

wait(semaphore *S) {
  S->value--;
  if (S->value < 0) {
    add this process to S->list;
    block();
  }
}
/*也就是说，如果轮到某进程执行的时候，可分配的资源为0，那么这个进程也会先把信号
量-1，“-1”代表了目前有一个进程在等待*/
/*-------------------------------------------------*/
signal(semaphore *S) {
  S->value++;
  if (S->value <= 0) {
    remove a process P from S->list;
    wakeup(P);
  }
/*也就是说，signal会先给信号量+1，这样子的话，表示他已经完成了对这个资源的使用，但是，如果在等待队列中存在着被block了正在等待的进程，这个资源“虽然已经被这个进程搞完了，但是队列中已经有进程排着队要搞我”，那么才会执行把等待队列中等待着的进程唤醒的操作；这样做的目的是为了区分开在等待队列的进程，以及没有在等待队列但是也想要用这个资源的进程，对于没有在等待队列但是也想用这个资源的进程，他看到的信号量依然不大于0，所以他也只能被放入等待队列 */
}
```

- 这个写法对于Binary Semaphore一样适用：

  A binary semaphore is a semaphore which value can be 0 or 1.

  - `acquire(s)` nullifies the value of s and place all other processes to the waiting queue q (semaphore queue of PCBs).
  - `release(s)` makes the critical section available by setting the value of s to one or releasing a "blocked" process from the queue.

  ```
  acquire(s)                      release(s)
  {                               {
   if(s==1) s = 0;                 if(q is ∅) s = 1;
   Else                            Else
      Block & place P in q;            Wake up P from q;
  }                               }
  ```

  - Each process, once dequeued and resumed will start executing its critical section.

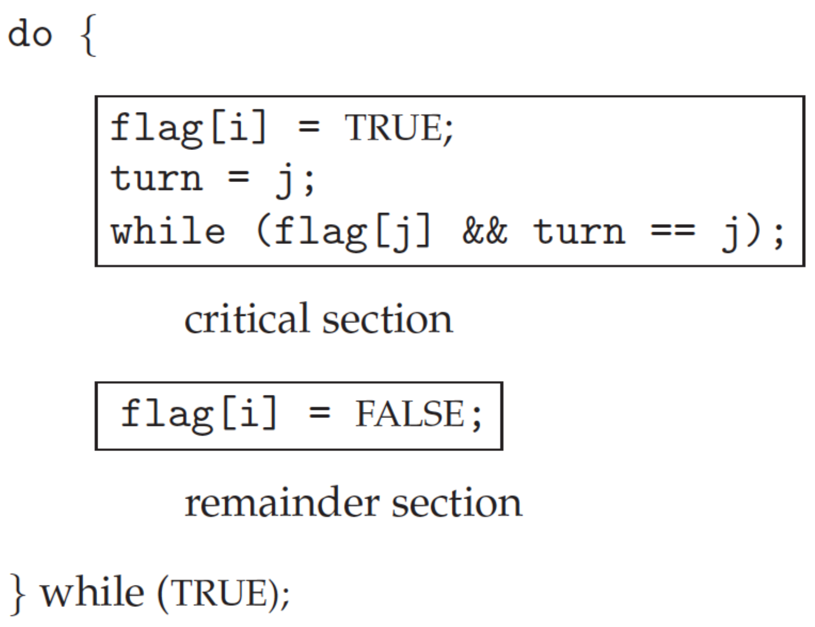
- 现在这样定义了之后，信号量可以是负的（对于counting semaphore）来说，负的定义是在等待的进程的数量，会是负的是因为This fact results from switching the order of the decrement and the test in the implementation of the wait() operation
- 每个信号量（定义在这里的）都存有一个integer value，and a pointer to a list of PCBs，用FIFO（first in first out）的queue可以implement这种list（先进入等待区的进程也会先进入准备区）（虽然理论上来说可以用any queueing strategy）
- wait()和signal()必须要是atomic operation必须保证没有两个进程可以同时执行`wait()`和`signal()`
  - For single-processor：
    - 在wait或者signal被执行的时候，把interrupt的出现给抑制住：once interrupts are inhibited, instructions from different processes cannot be interleaved
  - For multiprocessor：

        - interrupts must be disabled on every processor，不然的话，在不同的处理器上进行的进程可能会interleave
    - 课本229讲了为什么即使这样设计也无法彻底消除忙碌等待

- **Deadlocks**:
    - The implementation of a semaphore with a waiting queue may result in a situation where two or more processes <u>are waiting indefinitely for an event that can be caused only by one of the waiting processes.</u> “我等你走了再吃饭，你等我吃完饭再走”
    - 打个比方，假设有P1和P2两个进程，都需要A和B两个资源，现在P1持有A等待B资源，而P2持有B等待A资源，两个都等待另一个资源而不肯释放资源，就这样无限等待中，这就形成死锁，这也是死锁的一种情况。给死锁下个定义，如果一组进程中每一个进程都在等待仅由该组进程中的其他进程才能引发的事件，那么该组进程是死锁的。
        - 竞争不可抢占资源引起死锁：也就是上面那种，都在等待着对方的不可抢占资源
        - 竞争可消耗资源引起死锁：也就是assignmnet那种死锁，A向B发消息，B向C发消息，C向A发消息，死锁了
        - 进程推进顺序不当引起死锁：有进程p1，p2，都需要资源A，B，本来可以p1运行A --> p1运行B --> p2运行A --> p2运行B，但是顺序换了，p1运行A时p2运行B，容易发生第一种死锁。互相抢占资源。
- **Starvation**：
    - Another problem related to deadlocks is indefinite blocking, or **starvation**, a situation in which processes wait indefinitely within the semaphore. Indefinite blocking may occur if we remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.
    - 这基本上都取决于分配进程的设计，如果分配进程的设计是不够公平的，那么进程就有可能一直拿不到需要critical section的使用权，就会“饥饿”，当饥饿到一定程度，进程被赋予的任务及时完成也不具有实际意义的时候，这样子这个进程被称为饿死

Peterson's Algorithm

- 没有用到atomic operation
- 弊端：there are no guarantees that Peterson’s solution will work correctly on such architectures
- context: 这个人的solution把问题缩小到了两个进程：$P_i \ and \ P_j$，并且require the two processes to share two data item: `int turn; boolean flag[2];`
    - The variable **turn** indicates whose turn it is to enter its critical section. That is, if turn == i, then process Pi is allowed to execute in its critical section. The **flag array** is used to indicate if a process is ready to enter its critical section. For example, **if flag[i] is true, this value indicates that Pi is ready to enter its critical section.**

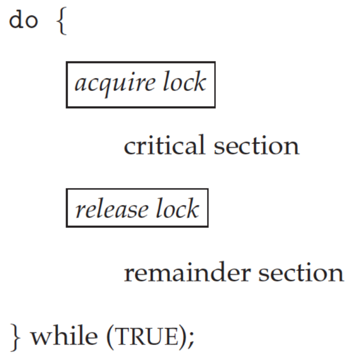


```
do {
        flag[i] = TRUE;
        turn = j;
        while (flag[j] && turn == j);

              critical section

        flag[i] = FALSE;

              remainder section

} while (TRUE);
```

- 也就是说，当下的进程会把自己的flag设为true，这暗示着自己是ready去执行critical selection的，但是同时他把turn设置为j，也就是说如果此时j也ready了，那么j会先运行，当j运行完了flag[j]会变成false，也就在此时，i会运行自己的critical selection，运行完了之后flag[i]会变成false，接下来就轮到j，一直一直搞，但在这个里面是存在busy waiting的
- 证明*

Hardware solutions for Critical section

- 就像前文提到的，像Peterson这种software based solutions并不能保证一定能够在现代的计算机上面正常工作。不过不管怎么样，我们都可以大致的总结出：基本上全部的critical-section problem都需要一个**lock**

  ```
  do {
        acquire lock

              critical section

        release lock

              remainder section

  } while (TRUE);
  ```

  一个进程在进入critical section之前都必须先获得这个lock，然后他的critical section结束之后才可以release这个lock
- Prevent interrupts from occurring while a shared variable was being modified：
  - 对于uniprocessor environment来说，这样子做的话其实效果很不错，This is often the approach taken by nonpreemptive kernels.
  - 但是对于multiprocessor environment的话(直接屏蔽interrupt对multiprocessor的影响)：Disabling interrupts on a multiprocessor can be time consuming, as the message is passed to all the processors. This message passing delays entry into each critical section, and

system efficiency decreases. Also, consider the effect on a system’s clock if the clock is kept updated by interrupts.

- 什么叫做“atomically”？ ： a unit "atomically" 意思就是这是一个“原始的，原子的”单位，也就是说，这个东西，他不会被interrupt打断！！！
    - 这里介绍两个atomic operation

    ```
    boolean TestAndSet(boolean *target){
      boolean rv = *target;
      *target = TRUE;
      return rv;
    }
    /*也就是把一个东西把input变成true，然后return 原来的input*/
    ```

    The important characteristic of this instruction is that it is executed atomically. Thus, if two TestAndSet() instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order

    ```
    do{
      while(TestAndSet(&lock))
        ;
      //critical section
      lock = FALSE;
      //remainder
    }while (TRUE)
    ```

    ```
    void Swap(boolean *a, boolean *b){
      boolean temp = *a;
      *a = *b;
      *b = temp;
    }
    //也就是把两个boolean变量交换位置
    ```

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

      // critical section

   lock = FALSE;

      // remainder section
} while (TRUE);
```

会不停的检查lock，直到出现了一个false之后，就把lock的false的值和key交换，这样lock变成true，别的不运行，key变成false，本体运行

- 上面的都没有满足“bound waiting这个条件”，下面的是满足的

```
      boolean waiting[n];
      boolean lock;

do {
   waiting[i] = TRUE;
   key = TRUE;
   while (waiting[i] && key)
      key = TestAndSet(&lock);
   waiting[i] = FALSE;

      // critical section

   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;

   if (j == i)
      lock = FALSE;
   else
      waiting[j] = FALSE;

      // remainder section
} while (TRUE);
```

        - 也就是说，当运行的时候，i代表了这个进程，只有当waiting[i]和key是True的时候，也就是都不可以运行时，会进入遍历，遍历中会不停的检查lock这个值，当lock是false之后，收到了信号可以执行，那么遍历停止，两个变量都被设置为false，开始执行critical section。wait这个array中只会有一个值会是false，这保证了mutually exclusive
        - To prove that the bounded-waiting requirement is met, we note that, when a process leaves its critical section, it scans the array waiting in the cyclic ordering (i + 1, i + 2, ..., n − 1, 0, ..., i − 1). It designates the first process in this ordering that is in the entry section (waiting[j] == true) as the next one to enter the critical section. Any process waiting to enter its critical section will thus do so within n − 1 turns也就是说他按照那个顺序检查第一个达到waiting[j] =true准备区域的进程并且释放lock
            - 为什么要检查i和j的equality?

Examine some well-know classical process synchronization problems

- The bounded-buffer problem
    - 要知道这是什么，首先要知道一个shared memory system的要求是：two or more processes agree to remove the restriction for preventing one process from accessing another process. 并且进程要同时负责各自不会在同一时刻加载到同一个shared variable
    - buffer：指的是一个缓存器，要保证两个进程能够同时运行，必须要有一个放置东西的缓存器，这个缓存器坐落在两个进程都可以看见的一个地址中
        - unbounded buffer：缓存器的一种，也就是说这个缓存器是没有大小限制的："consumer可能要等新的item，但是producer却可以源源不断的把东西放入缓存器中"
        - bounded buffer：缓存器的一种，这个缓存器是有限制的，当没有东西的时候，consumer一定要等，当东西满了放不进去的时候，作为producer也必须要等这个里面的东西空了之后再生产东西并且放进去
    - 解决办法：
        - 条件：在一个池中有n个buffer，并且每个buffer都可以存一件item，mutex semaphore provides mutual exclusion for access to the buffer pool and is initialized to the value 1. The empty and full semaphores count the number of empty(initialized to the value n) and full buffers(initialized to value 0).

```
//producer
do {
  //produce an item in nextp
  wait(empty);
  wait(mutex);
  //add nextp to buffer
  signal(mutex);
  signal(full);
}while(TRUE)
```

```
//consumer
do {
  wait(full);//没有有东西的buffer的话你去拿什么东西
  wait(mutex);//别的进程在占着的话你去拿什么定西
  //remove an item form buffer to nextc
  signal(mutex);
  signal(empty);
  //consume item in nextc
}
```

- The Readers-Writers Problem
    - 是个相当简单的概念：这样是为了能够允许多个进程读取同一个数据，但允许多个进程读取同一个数据的同时却又不允许多个进程修改同一个数据，因为读取并不会打乱数据本身
    - Require that the writers have exclusive access to the shared database while writing to the database
    - The simplest one: 除非一个写入进程已经获取了对某个共享资源的控制，不然不能让读取进程等待。No reader should wait for other readers to finish simply because a writer is waiting。也就是说，当一个写入进程在等待的时候，读取进程不会因为这个写入进程想进入就停止进入
        - 要解决以上这种，首先，先声明读取进程会分享以下的变量

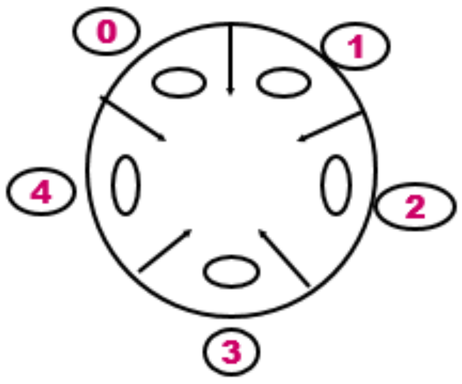
```
semaphore mutex, wrt;
mutex = 1; wrt = 1;
//像之前的各种例子一样，这个mutex的信号量就是用来确保mutual exclusive的，所以这个的唯一的作用其实是在readcount这个变量更新的时候，保证mutually exclusive
//wrt是一个给writer进程的mutually exclusive的信号量，同样的这个信号量也会为第一个进入以及最后一个离开的读取进程所服务
int readcount;
readcount = 0;
//这个readcount的变量是用来记录目前有多少个读取进程在读取这个东西的
```

```
//这是一个writer process
do {
  wait(wrt);
  //writing is performed
  signal(wrt);
}while(TRUE);
```

```
//这是一个reader process
//也就是说，当只有一个读取进程进入到了这里的时候，会让写入进程先执行，然后自己再执行读取任务，如果自己是最后一个执行读取任务的，那么就会signal（wrt），从而让写入进程执行
```

```
//Note that, if a writer is in the critical section and n readers are
waiting, //then one reader is queued on wrt, and n − 1 readers are
queued on mutex.
do {
  wait(mutex);
  readcount++;
  if (readcount == 1)
    wait(wrt);
  signal(mutex);
  //reading is performed
  wait(mutex);
  readcount --;
  if (readcount == 0)
    signal(wrt);
  signal(mutex);
}while(TRUE);
```

        - 当一个写入进程结束之后，可能会恢复到正在等待的读取进程或者是写入进程，具体会去到等待的哪个进程完全由schduler去决定
    - The second one: 只要一个写入进程准备好了，那么没有新的读取进程可以开始，写入进程会尽可能快的开始
    - 上面介绍的这两种不一样的情况，都很可能会导致饥饿问题，要么是写入进程进不去，要么是读取进程进不去
    - 这个问题十分广泛，以至于这种结局方法都已经被generalize到reader-writer lock
        - When a process wishes only to read shared data, it requests the reader–writer lock in read mode; a process wishing to modify the shared data must request the lock in write mode.
    - 一般广泛使用在以下场景：
        - 每个进程被完美的区分成读取进程和写入进程
        - 读取进程的数量要多于写入进程
- The Dining-Philosopher Problem哲学家进餐问题



- 一个桌子上有五个哲学家，哲学家生存的方式是思考和进餐，思考和进餐不能同时进行，桌上只有五只筷子，每次进餐的时候，一个哲学家需要同时拿起左边和右边的筷子开始进餐。当哲学家进餐完毕之后，哲学家便会马上放下手中的另个筷子。如果五个哲学家同时饥饿，同时拿起自己左边的筷子，那么就会陷入无限的等待右边的筷子当中，死锁
- 要解决这个问题似乎相当的直白，就是把五个筷子每个都标记成一个信号量，每次哲学家拿起筷子之前要先形式wait，当他用完筷子之后要使用signal，按照这个思路，五个哲学家之间分享的变量即是：

  ```
  semaphore chopstick[5];
  ```

  ```
  do {
    wait(chopstick[i]);
    wait(chopstick[(i + 1)%5]);
    //eat
    signal(chopstick[i]);
    signal(chopstick[(i + 1)%5]);
    //think
  }
  ```

  这样子的设计只能保证相邻的两个哲学家不会同时吃饭，但是如果五个哲学家同时饥饿，每个哲学家都拿起了左边的筷子，那么在chopstick里面存着的信号量会一下子全变成0，死锁了！

  - 有三种解决办法：
    - Allow at most four philosophers to be sitting simultaneously at the table.
    - Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).
    - Use an asymmetric solution; that is, an odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick.
- 不管怎么样，解决这个的办法除了要保证他不会死锁，还要保证哲学家不会挨饿

Lecture 10-Partial

Process Precedence Graph for synchronization

- PPG(Process Precedence Graph) is a directed graph that is used to graphically express the order on which processes or threads are executed with respect to other processes or threads

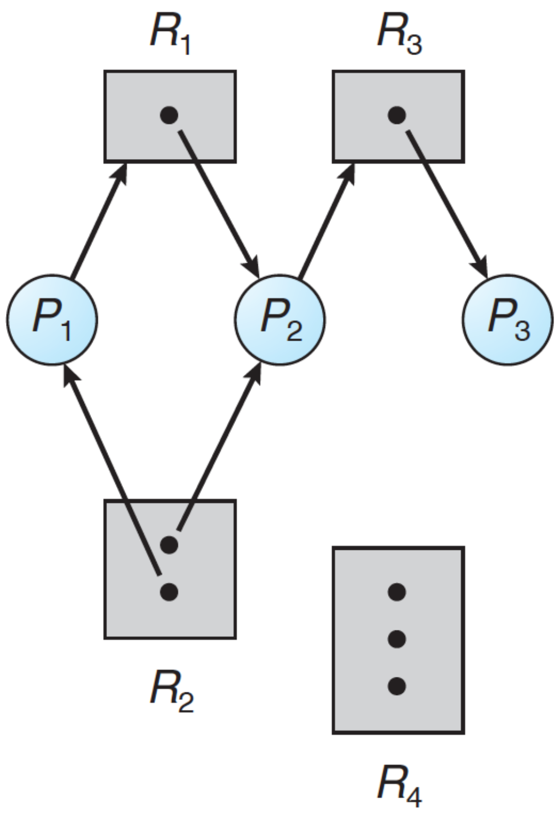
*用ipad来写

Lecture 12&Lecture 13&Lecture 14&Lecture 15

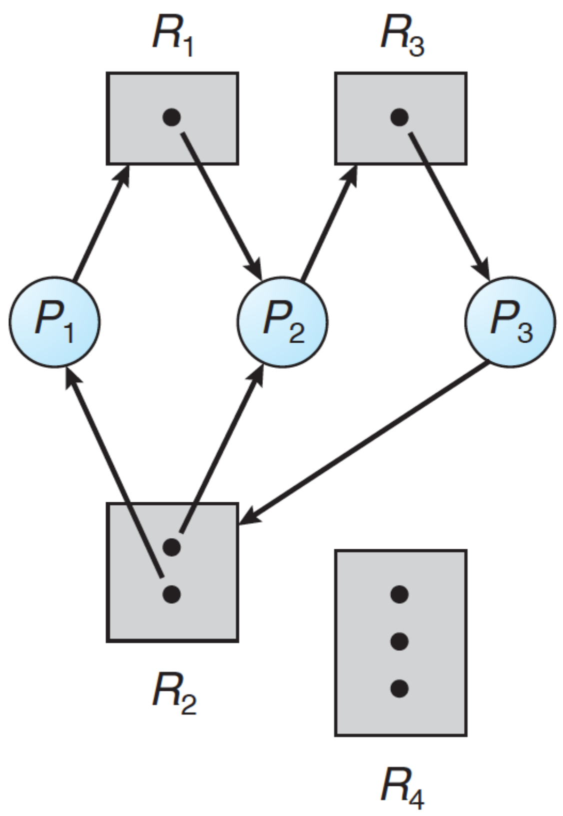
Deadlock

- 当进程进入了一个无止境的等待资源的区间，那么这个进程就是死锁了

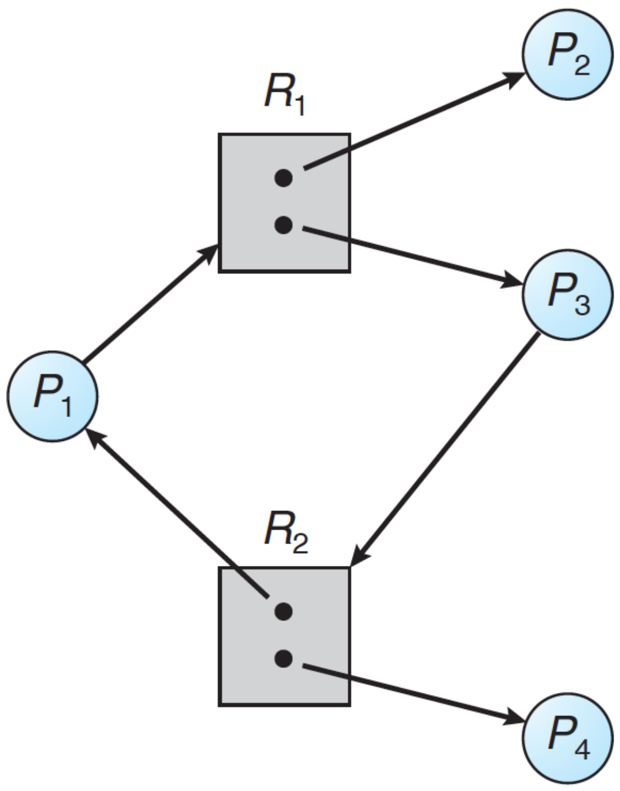
- 标准定义：**A set of processes is in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set.**
- 当以下四种情况同时发生，死锁会产生：
  1. Mutual Exclusion（想使用同一个资源的进程互斥）
  2. Hold and wait：一个进程持有至少一个资源，并且他在等待别的进程释放资源
  3. No preemption：一个资源不可以被抢占，一个资源只可以自愿被进程所释放
  4. Circular Wait：在一个集合$\{P_0, P_1, ..., P_n\}$中，P_i在等待P_{i+1}释放的资源
- System resource-allocation graph



  - P为进程的顶点，R为Resource type，里面的小点代表了他的各个instance，所以把每个小点可以看作是一个具体的资源的实例
  - $P_i \rightarrow R_i$: Request edge，进程P_i已经申请并且在等待资源R_i，这个请求的edge并不会 具体指向每个实例，而是只会指向资源的类型
  - $R_i \rightarrow P_i$：Assignment edge，资源R_i已经被分配给了进程P_i，这个分配的edge是从具体的资源实例出发
  - ***CYCLE***：A directed **cycle** in a directed **graph** is a subset of the graph in which the only repeated are the first and last vertices. 根据这个定义，可以看出，这张图如果存在cycle，并且每个资源类型只有一个资源实例的话，那就代表，最后一个进程在申请第一个进程持有的资源，并且circular wait，也就是说这张图死锁了
  - 存在cycle并且存在死锁：



- 和下面那张图不一样的是：要打破死锁唯一的可能是P3拿到R2的资源，P1拿到R1的资源，P2拿到R3的资源，但这些资源目前所处于的进程都需要依赖其他资源来释放，所以这个必定死锁
- 存在cycle但不存在死锁：



- 所以看图的时候要认真看，因为P4可以释放自己的R2资源，从而打破这个cycle

- Handling Deadlock：
    - Protocol to prevent/avoid deadlock
        - deadlock prevention
            - 一系列的条件去确保至少一个（共四个）死锁条件不会成立
        - deadlock avoidance
            - 给os额外的知识：一个进程会申请什么资源
    - Enter a deadlock state, detect it and recover it
    - Ignore the problem
- 如果死锁出现了，并且没有避开它，没有侦查到他，会怎么办：卡死，一个进程被拖住，全部进程步步维艰，根本搞不起来，万能大法：重启，因为实际上很多系统一年都不会遇到一次死锁的问题，相比起写那么多代码，还不如直接重启

Preemption（抢占）

- 抢占(Preemption)是指内核强行切换正在CPU上运行的进程，在抢占的过程中并不需要得到进程的配合，在随后的某个时刻被抢占的进程还可以恢复运行。发生抢占的原因主要有：进程的时间片用完了，或者优先级更高的进程来争夺CPU了

Lecture 16 & Lecture 19

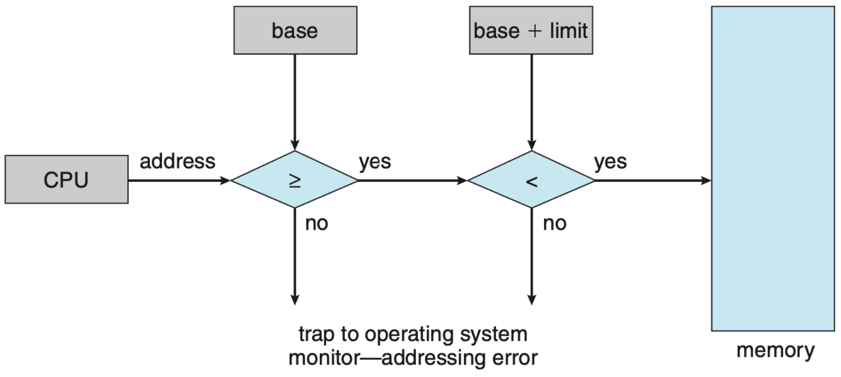
- Discuss various ways to manage memory. The memory management algorithms vary from a primitive bare machine approach to paging and segmentation strategies.
- 关于memory management的选择基于多种因素，很大程度上被这个系统的硬件设计而影响
- 程序调用memory中的东西的时候都是通过typical instruction executation cycle, memory unit只能看到存储地址的流动，但却无法知道这些存储地址是怎么被生成或者调用的
- Word size VS Address size
    - byte addressable指的是一个地址有八个bit，“size of one location is 8 bits”
    - address size指的是size of one address, 这个可以拿来计算总共可以有多少个location

Basic Hardware

- Main Memory和register都属于direct access storage device，因为cpu可以直接从这些设备里面读取信息，所以多难过要执行程序的时候，我们fetch的instruction，data都必须要在这些设备里面，如果这些数据本身并不在这些设备里面，那么就需要在执行操作之前先把这些数据，instruction移动到这些设备中
    - 读取Register:
        - 读取register就很简单，are generally accessible within one cycle of the CPU clock. Most CPUs can decode instructions and perform simple operations on register contents at the rate of one or more operations per clock tick.
    - 读取Main Memory:
        - 但是读取main memory却完全不一样，因为读取main memory需要通过transaction on

the memory bus，这样一次access就需要多个cpu cycle，the processor normally needs to **stall**, since it does not have the data required to complete the instruction that it is executing. 这就会拖慢整个运行的速度$\rightarrow$不能容忍！怎么办？$\rightarrow$ CACHE！

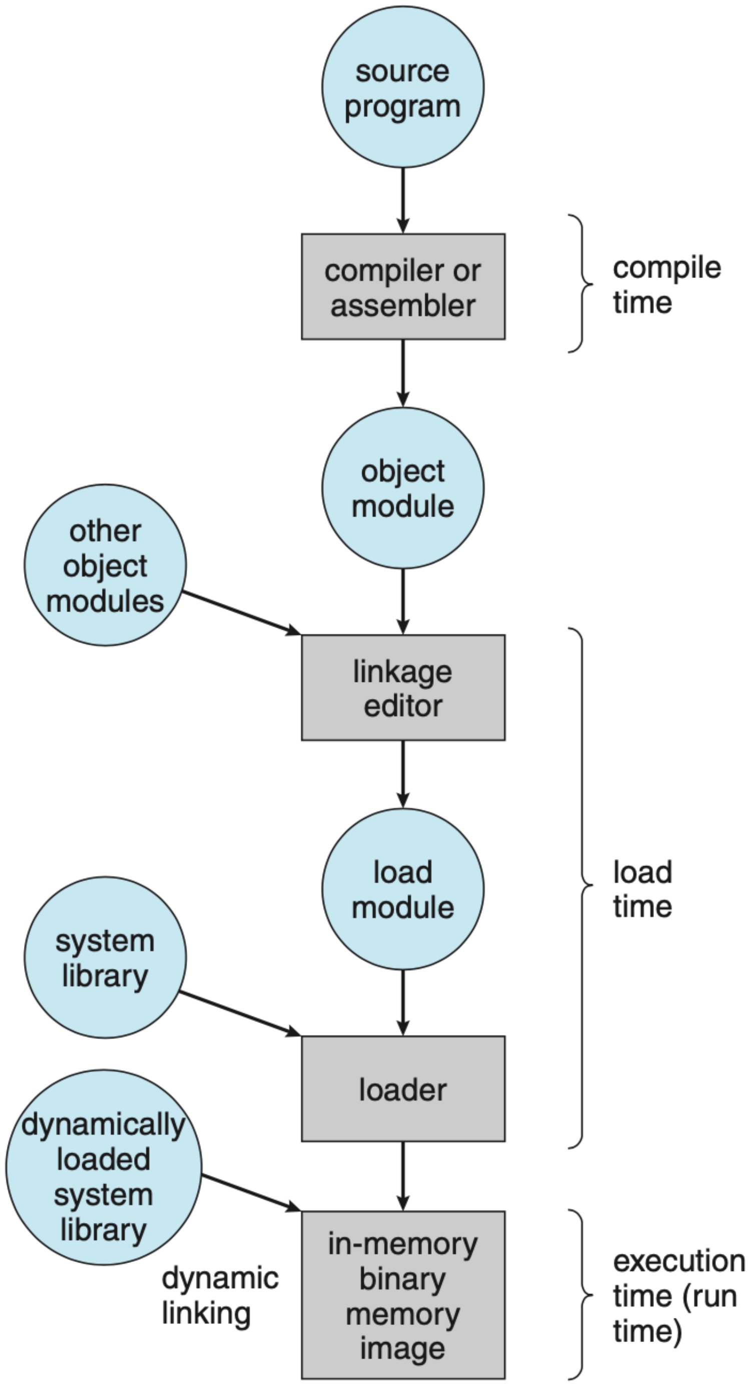
- ***Cache***：坐落于cpu和main memory中间，this is the remedy(add fast memory between cpu and main memory) for main memory accessing time. **A memory buffer used to accommodate a speed differential, called a cache, is described in Section 1.8.3.**
- ***Protection of memory space***: 意即为了防止操作系统的文件被user给access到，或者是一个user的memory被另一个user access到，为了做到防止，我们首先需要确保每个进程有一个独立的储存空间
  - **Base register**: Holds the smallest legal physical memory address
  - **Limit register**: Specifies the size of the range
    - Example: If the **base register** holds 300040 and the **limit register is 120900**, then the program can legally access all addresses from 300040 through 420939 (inclusive). Base和Limit的值对于每个进程都是一样的嘛?
    - 这两个register都只可以被操作系统加载，也就是说只有操作系统自身可以修改这两个值
  - 然后cpu会对user mode里面的register生成的每一条地址都进行检查，只要检查到一条是想要access os或者是别的user的memory的都会令这个程序进入error
  - Having the CPU hardware compare *every* address generated in user mode with the registers.



  - 但是对于os来说，它拥有很高的权限，它能没有限制的访问os memory和user memory，访问user memory是为了把user program加载进user memory中

Address Binding（位址定位）

- **The process may be moved between disk and memory during its execution.**
- 程序最后要在main memory中的那个address执行，那么这个address是怎么产生的呢？这时候就需要位址定位。
- Generally, processes are on the disk that are waiting to be bought into memory for execution from the **input queue**, 一般的流程是这样的: 从input queue中选择一个process, load进memory, 然后执行，这个process使用指令和data，最终当这个process结束的时候，它使用的这个memory会被declared available
- 所以一个user program要执行的时候，需要经过多重障碍：



- 注意操作系统允许user process被储存在any part of physical memory中，虽然一个计算机的第一个地址是00000，但是user processes的首位地址不一定要是00000，这样的设计决定了user process能用到的地址
- 一个user program在最最最初到被执行之间，经历了多种阶段，address会被不同的表示方法来表示，在source program中，address一般是比较symbolic的(such as count). 然后到了compiler的阶段，compiler会bind这些symbolic的地址**to relocatable addresses (such as “14 bytes from the beginning of this module”).** The linkage editor or loader will in turn bind the relocatable addresses to absolute addresses (such as 74014).

    - **Each binding is a mapping from one address space to another.**所以最后才可以使用address去找到这个程序并且执行
    - Classically, the binding of instructions and data to memory addresses can be done at any step along the way，位址定位可以在下面几个阶段中的任意一个完成:
        - Compile time(编译时期): **If you know at compile time where the process will reside in memory, then <u>absolute code</u> can be generated.** For example, if you know that a user process will reside starting at location *R,* then the generated compiler code will start at that location and extend up from there. If, at some later time, the starting location changes, then it will be necessary to recompile this code.
        - Load time(载入时期): **If it is not known at compile time where the process will reside in memory, then the compiler must generate <u>relocatable code</u>.** In this case, final binding is delayed until load time. If the starting address changes, we need only reload the user code to incorporate this changed value.
        - Execution time(执行时期): **If the process can be moved during its execution from one memory segment to another, then <u>binding must be delayed until run time</u>.** Special hardware must be available for this scheme to work, as will be discussed in Section 7.1.3. Most general-purpose operating systems use this method.

Logical Versus Physical Address Space

- ***Logical Address(Virtual address)***: An address generated by the CPU (in the range 0 to *max*)
    - **Logical address space**: The set of all logical addresses generated by a program (in the range *R* + 0 to *R* + *max* for a base value *R*)
- ***Physical address***: An address seen by the memory unit, that is, the one loaded into the memory-address register of the memory
    - **Physical address space**: the set of all physical addresses corresponding to these logical addresses
- 如果位址定位发生在编译时期或载入时期，那么此时的logical address和physical address是一样的
- 但是如果位址定位发生在执行时期，那么此时出来的logical address和physical address是不一样的，换言之此时他们的logical and physical address space differ
- ***Memory-management unit(MMU)***: 这个东西是为了在runtime的时候可以把virtual address映射到相对应的physical address上面去

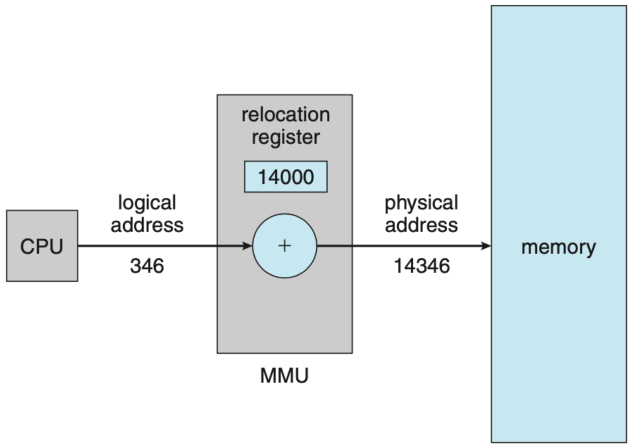


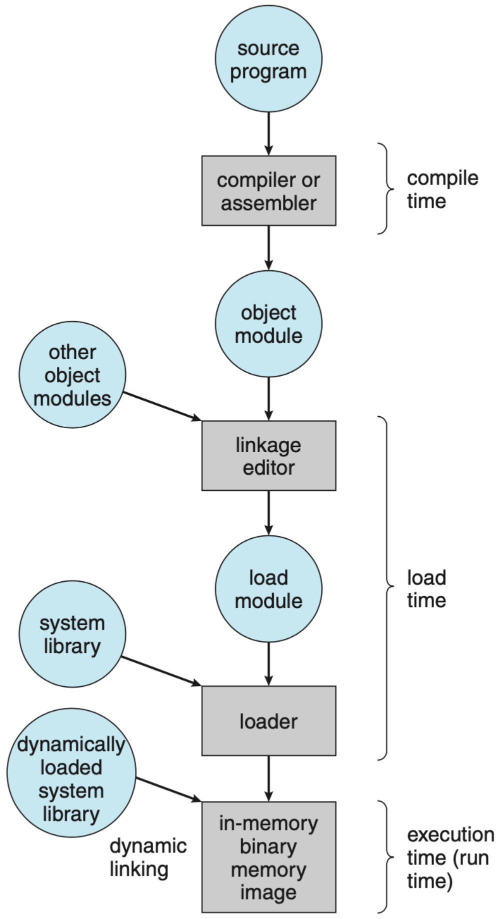
Figure 7.4 Dynamic relocation using a relocation register.

  - （其中一种做法）如上图，中间那个就是**relocation register**，在这个relocation register中存着一个值，每一个address generated by a user process都会先加上这个值再送进内存中。For example, if the base is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location 14000; an access to location 346 is mapped to location 14346.
- User program **only** deal with **logical address**, 从来不知道真正的physical address
- The **memory-mapping hardware** converts logical addresses into physical addresses. This form of **execution-time binding** was discussed in Section 7.1.2. The final location of a referenced memory address is not determined until the reference is made.
- **The concept of a *logical address space* that is bound to a separate *physical address space* is central to proper memory management.**
- 这样做的目的是：这样的做法给os提供了一个保护内存的办法

Dynamic Loading动态加载

- 基于刚刚的讨论，会发现一个问题，就是一个process基本上所有的东西（全部的数据，entire program）都是存在physical 内存里面的，所以一个进程的大小就被physical memory的大小给限制住了，dynamic loading动态加载就是为了解决这个问题的
- 什么是动态加载？→程序会被保存在磁盘上，in a relocatable load format.主程序会首先被加载进内存，并且执行，如果此时需要执行另一个程序，the calling routine要首先检查这个被叫起来的程序有没有已经被加载了，如果没有被加载的话，relocatable linking loader会被执行并且会把那个程序给load进memory，然后更新program的address table。然后控制会给到新进来的程序
- 这个的优点是不需要的程序将永远不会被加载。**This method is particularly useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines.** In this case, although the total program size may be large, the portion that is used (and hence loaded) may be much smaller.
- 但是！这个动态加载并不是操作系统的职责而是设计者的职责。

Dynamic Linking and Shared Libraries



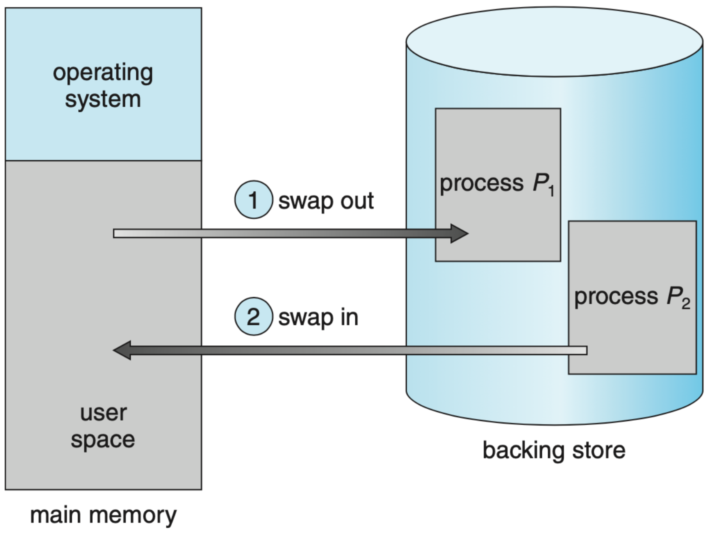
- 这是刚刚那个图，可以看到有一个dynamically loaded system library。因为有许多的程序需要用到system library，那么这个system library会在什么时候加载呢？用两种不同的办法：静态连接和动态连接
    - 静态连接：
        - statically-linked library is a set of routines, external functions and variables which are resolved in a caller at compile-time and copied into a target application by a compiler, linker, or binder, producing an object file and a stand-alone executable
        - 静态连接就是，使用普通的函数库，在程序连接时将库中的代码拷贝到可执行文件中。假设有多个程序同时执行，并且同时调用了同一个库文件，这是内存中就会保留着许多重复的代码副本。造成内存浪费。
    - 动态连接：
        - 动态链接就是，只有程序在执行时才将库中的代码装入内存，对于同一个动态链接库，无论有多少个程序在调用，内存中都只有一个动态库的副本。当动态库不再被任何程序使用，系统就会将它调出内存，这样就减少了应用程序对内存的要求。动态链接库是一种程序模块，不仅可以包含可执行的代码，通常还包含各种类型的预定义的数据和资源，扩大了库文件的使用范围。
        - **With dynamic linking, a *stub* is included in the image for each library- routine reference. The stub is a small piece of code（stub是存在哪里的呢？？ stub是存在程序里面，当这个程序需要用到system library的时候，他对于system library的reference就会包含这个stub）**that indicates how to locate the appropriate memory-resident

library routine or how to load the library if the routine is not already present. **When the stub is executed, it checks to see whether the needed routine is already in memory.** If it is not, the program loads the routine into memory. Either way, the stub replaces itself with the address of the routine and executes the routine. （所以，第一个需要这个system library的程序会把这个system library routine给加载进内存里面，这样下一个需要这个system library的程序就不需要重复加载了）Thus, the next time that particular code segment is reached, the library routine is executed directly, incurring no cost for dynamic linking.

    - Under this scheme, all processes that use a language library execute only one copy of the library code.
    - 动态连接同时还关系到了库的更新，如果一个库更新了，使用dynamic linking的系统不会受到太大的影响，因为每个程序是在run time的时候才连接的库，但是如果static linking，在编译的时候就连接的库的话，会造成一定的影响，因为**all such programs would need to be relinked to gain access to the new library**. Other programs linked before the new library was installed will continue using the older library. This system is also known as **shared libraries（动态连接库）**
  - **链接有两种类型，一种是静态链接，直接将所用到的各个部分拼接起来作为所得到的最终文件。另一种是动态链接，不是直接的进行拼接，最终得到的文件仍然不是完整的程序，而是保留了接口。动态链接所得到的可执行文件在执行的时候按照需要动态的加载用到的部分，与此不同的静态链接没有这个动态加载的过程，因为所有需要的部分已经包含在可执行文件里面了。这里可以看到动态链接生成的文件往往会小一些，因为可重用的部分已经独立出来作为库文件了，这也是构建大型软件所必须的。**（精彩解释）
- 和dynamic loading不一样的是，如果进程之间是相互保护的，只有操作系统可以看到其他进程的情况，那么操作系统就有义务汇报其他的进程有没有把目前需要的system library给加载进来

Swapping

- 回想之前的进程部分，一个进程的状态其中一个是***Suspended***：Swapped out of RAM for some reason. 那么现在正式的定义一下swap这个操作：A process can be **swapped** temporarily out of memory to a **backing store** and then brought back into memory for continued execution



- 就像上图描述的这样，理想状态是： the memory manager can swap processes fast enough that some processes will be in memory, ready to execute, when the CPU scheduler wants to reschedule the CPU
- 并且这样的概念还可以用在priority-based scheduling里面，如果一个优先级别高的进程想要进入，那么memory manager就会把优先级别低的进程给swap out，执行优先级别高的进程，这个执行完了，再让优先级别低的进来。这样子的这样子的用法也叫做***roll out, roll in***
- A process that is swapped out will be swapped back into **the same memory space** it occupied previously. $\rightarrow$ 因为如果位址定位发生在编译时期或者载入时期的话，这个process的location是不可以改变的。但是如果位址定位是发生在执行时期的话，then a process can be swapped into a different memory space, because the physical addresses are computed during execution time.
- The system maintains a **ready queue** consisting of all processes whose memory images are on the backing store or in memory and are ready to run.
- 关于cpu scheduler在运行进程时的一个总结：Whenever the CPU scheduler decides to execute a process, it calls the dispatcher. The dispatcher checks to see whether the next process in the queue is in memory. If it is not, and if there is no free memory region, the dispatcher swaps out a process currently in memory and swaps in the desired process. It then reloads registers and transfers control to the selected process.
- swap花费的时间相当的长：
    - 假设现在一个user process是100MB，然后hard disk作为backing store，他的transfer time是50MB/s，一次transfer需要2000milliseconds，然后延迟8milliseconds，那么swap一次（不管是in还是out）需要2008milliseconds，总共的swap time for one process is 4016 milliseconds
- swap和context switch
- 因为swap花费的时间相当的长，所以最好能够只swap那些真正用到的，dynamic memory

Memory Mapping and Protection

- In **contiguous memory allocation**, each process is contained in a single contiguous section of memory.
- 具体的解释了cpu scheduler是怎么通过**relocation**和**limit register**去保护os和其他用户的内存的：
    - When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch. Because every address generated by a CPU is checked against these registers, we can protect both the operating system and other users’ programs and data from being modified by this running process.

Memory Allocation

- 其中一个最简单的分配内存的方法就是divide memory into several fixed-sized partitions(划分). 划分好了之后可能的做法是每一个划分都包含着一个进程，所以这样的话划分的数量就决定了multiprogramming的上限。

- ***Multiple-partition method***: when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process.
    - ***Fixed-partition scheme***
        - In this partitioning, number of partitions (non-overlapping) in RAM are **fixed but size** of each partition may or **may not be same**. Here partition are made before execution or during system configure.
    - ***Variable-partition scheme***:
        - **Initially RAM is empty and partitions are made during the run-time according to process's need instead of partitioning during system configure.**
        - The size of partition will be equal to incoming process.
        - The partition size varies according to the need of the process so that the internal fragmentation can be avoided to ensure efficient utilisation of RAM.
        - Number of partitions in RAM is not fixed and depends on the number of incoming process and Main Memory's size.
        - Memory会不停的被分配给进程直到内存无法满足下一个进程的要求
        - textbook上面举的例子是把整个内存想象成很多的洞，一个进程进来了，系统搜索到一个过大的洞，那么这个系统把这个洞分成两半，一半给这个进程，一半回到洞的集合，如果两个相邻的洞都是available的，那么他们可以merge在一起形成一个更大的洞。This procedure is a particular instance of the general **dynamic storage- allocation problem**, which concerns how to satisfy a request of size *n* from a list of free holes。然后一般搜索洞的过程又可以使用三种不同的策略：
            1. *first-fit*: Allocate the *first* hole that is big enough
            2. *best-fit*: Allocate the *smallest* hole that is big enough.
            3. *worst-fit*: Allocate the *largest* hole.

Fragmentation

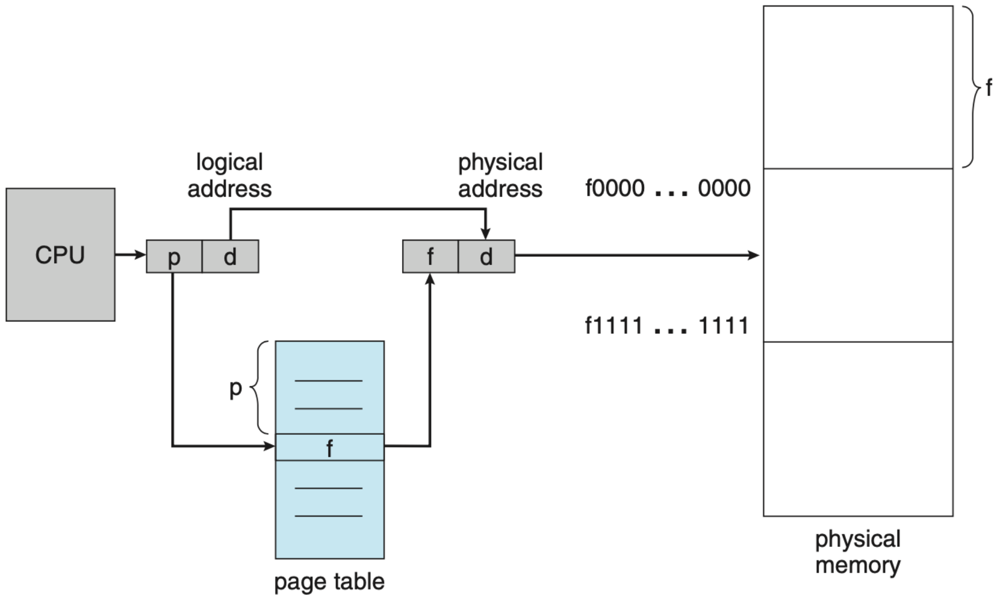
- 如果是使用了first/best fit的策略的话，不难发现，这样会带来许多的存储碎片，也就是说，现在可能内存中拥有足够的空间，但这些空间都被分成了不相邻的几个部分，各个部分无法满足程序的要求，这样子的话就叫做**External Fragmentation**. ***External fragmentation*** exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous; storage is fragmented into a large number of small holes.
    - 其中一个解决external的办法是**compaction**：The goal is to shuffle the memory contents so as to place all free memory together in one large block.
        - 可见如果位址定位发生在编译时期或载入时期（relocation是不possible的）的话，这个就是不可能实现的
    - 外部碎片问题的另一个可能的解决方案是允许进程的逻辑地址空间是非连续的，从而允许在任何可用的物理内存中分配进程。（和Paging很像）

- Statistical analysis of first fit, for instance, reveals that, even with some optimization, given *N* allocated blocks, another 0.5 *N* blocks will be lost to fragmentation. That is, one-third of memory may be unusable! This property is known as the **50-percent rule**.
- ***Internal fragmentation***: Break the physical memory into fixed-sized blocks and allocate memory in units based on block size.
    - 为什么要这样子做呢？因为如果现在这个大洞是18464bytes，一个进程需要18462bytes，把这个大洞分给这个进程之后只剩下2byte的洞，那么一直keep track of this hole的cost会比合格hole本身还要高。With this approach, the memory allocated to a process may be slightly larger than the requested memory.
    - 当一个进程装入到固定大小的分割块（比如页）时，假如进程所需空间小于分割块，则分割块的剩余的空间将无法被系统使用，称为内部碎片（**internal fragmentation**）

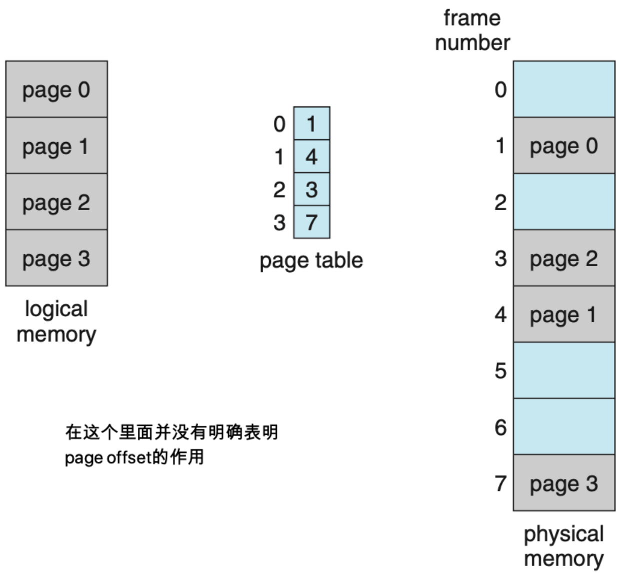
Lecture 20 & Lecture 21

Paging（分页）

- Paging is a memory-management scheme that permits the physical address space of a process to be noncontiguous. (Paging技术允许一个进程的物理地址是不连续的)
- **Paging的好处**：
    - Avoids external fragmentation（and the need for compaction）
    - Solves the considerable problem of fitting memory chunks of varying sizes onto the **backing store**，这个问题产生的原因是当一个在内存中的进程被swapped out要存进backing store的时候，backing store同样面临着上面memory allocation的问题，并且因为在backing store里面，读取速度相当的慢，所以compaction几乎是不可能的
- 传统的，paging技术都是由硬件来完成的，最近的设计通过整合硬件和操作系统来完成paging
- ***Basic Method***：
    - The basic method for implementing paging involves breaking *physical memory* into fixed-sized blocks（fixed size，但没有说是same size） called **frames** and breaking *logical memory* into blocks of the same size（same size，但没有说是fixed size）called **pages**.
    - ***Page***：page的大小是由硬件来决定的，一般是2的幂（这是为了在把逻辑地址切成page number和page offset的时候能够比较简单），在512bytes和16mb per page之间徘徊
        - If the size of the logical address space is 2^m and a page size is 2^n addressing units (bytes or words), then the high-order *m* − *n* bits of a logical address designate the page number and the *n* low-order bits designate the page offset.

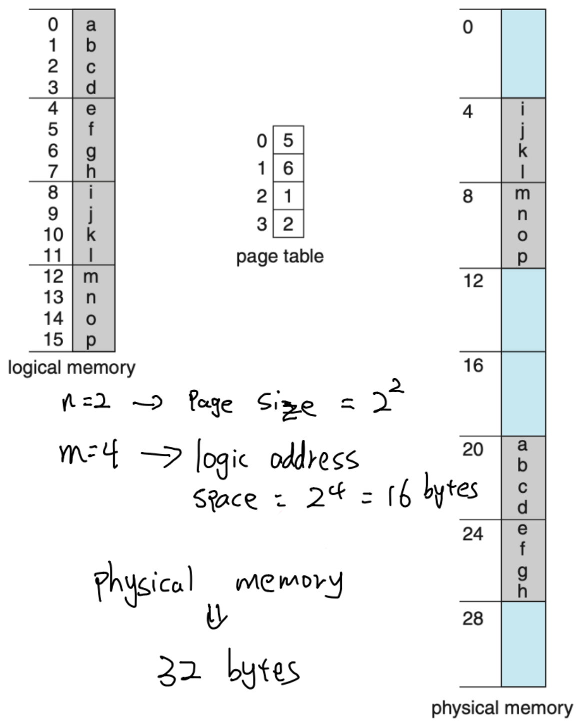


  - 一个逻辑地址（address generated by cpu）会被分成两个部分:
    - ***page number(p)***：作为一个***Page table***的index，这个table包含了the base address of each page in physical memory
    - ***page offset(d)***：通过上面的page number找到了具体的page table里面对应的东西了之后，page offset和page table里面定义的可以结合成一个physical memory address，那么就找到了！(displacement within the page)比较简单的例子是下面的



在这个里面并没有明确表明
page offset的作用

- 当一个进程准备运行时，这个进程的page会被加载进一个available的memory frame（要么是file system的，要么是backing store的）
- 一个更加concrete的例子



logical memory
page table
physical memory
n=2 → page size = 2^2
m=4 → logic address space = 2^4 = 16 bytes
physical memory ⇓ 32 bytes

- For logical address 0: page num = 0, page offset = 0, 从page num找到page table中的值是5，那么

$$According\ physical\ address: (5*4)+0=20$$

- For logical address 3: page num = 0, page offset = 3, 从page num找到page table中的值是5，那么

$$According\ physical\ address: (5*4)+3=23$$

- For logical address 4: page num = 1, page offset = 0, 从page num找到page table中的值是6，那么

$$According\ physical\ address: (6*4)+0=24$$

- 以此类推，因为现在page size是4 byte，所以把逻辑地址分成4份（每份4 bytes），物理地址大大小是32 bytes，所以物理地址总共能够容纳8个pages，在这种情况下，比如要转换第一个逻辑地址0，他属于逻辑地址中的第一个page，所以page num是0，然后他在这个这个page中所占的位置也是第一个，所以page offset是0，从page num找到page table中的值是5，然后进行$(5*4)+0=20$，那么这个逻辑地址为0的东西映射到物理地址就是20
- Logical Address = 24 bits

        - Logical Address space = 2 ^ 24 bytes
        - Let's say, Page size = 4 KB = 2 ^ 12 Bytes
        - Page offset = 12
        - Number of bits in a page = Logical Address - Page Offset = 24 - 12 = 12 bits
        - Number of pages = 2 ^ 12 = 2 X 2 X 10 ^ 10 = 4 KB
        - Let's say, Page table entry = 1 Byte
        - Therefore, the size of the page table = 4 KB X 1 Byte = 4 KB
    - Page和Frame的区别："page" means "virtual page" (i.e. a chunk of virtual address space) and "page frame" means "physical page" (i.e. a chunk of physical memory). 也就是说他们的大小都是一样的，只不过一个是logical的一个是physical的
    - Using paging is similar to using a table of base (or relocation) registers, one for each frame of memory
- 当我们使用paging的时候，external fragmentation会被彻底消除，frame会被分配给有需要的进程。
- 但是internal fragmentation却没有完全消除（就是有些分配给进程的内存大于进程实际需要的），在最坏的情况下，一个进程需要n pages + 1 byte，这样申请下来的第n + 1个 frame里面会有internal fragmentation of almost an entire frame
- **If process size is independent of page size, we expect internal fragmentation to average one-half page per process**：这样也就是说，如果page size小一点的话，似乎会更好，但是如果page size小的话，page table也需要存更多的东西，这会造成overhead
- Today, pages typically are between **4 KB and 8 KB** in size, and some systems support even larger page sizes. 有些cpu或者内核甚至需要多个不一样的page size。
- **Usually, each page-table entry is 4 bytes long, but that size can vary as well. A 32-bit entry can point to one of 2^{32} physical page frames. If frame size is 4 KB, then a system with 4-byte entries can address 2^{44} bytes (or 16 TB) of physical memory.** 4 byte entry可以指向2^{32}个physical page frames，一个系统的frame size，一个frame的大小是4kb，所以这个总共可以address $2^{32} * 4KB$个bytes

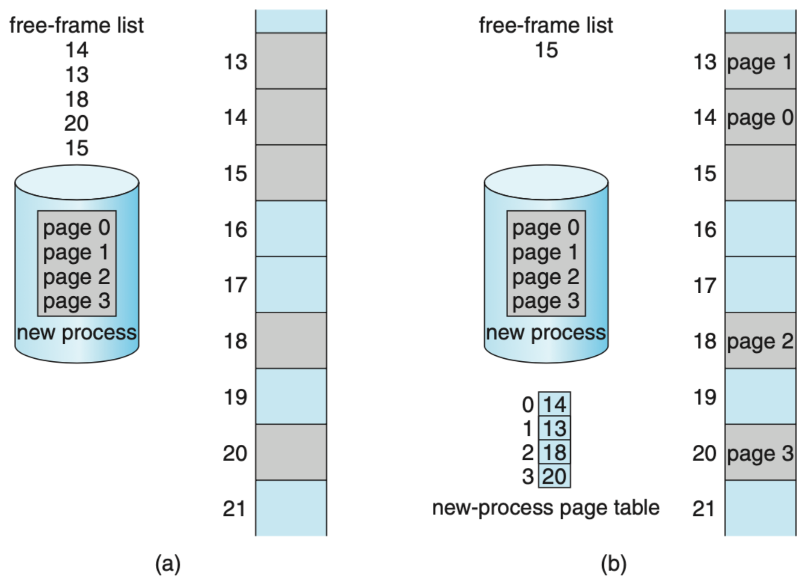


Figure 7.10 Free frames (a) before allocation and (b) after allocation.

- 关于paging很重要的一个概念是用户对于内存的了解跟实际的物理内存是完全不一样的
- 一个进程has no way of addressing memory outside of its page table, and the table includes only those pages that the process owns.
- 因为是操作系统负责的翻译（把逻辑地址map到物理地址上面去），所以操作系统必须要知道一些信息，比如哪些frame已经被使用了，哪些frame还没被使用，总共还剩下多少个frame等等，这些信息，都被储存在一个叫做***frame table***的data structure里面
  - The **frame table** has **one entry for each physical page frame**, indicating whether the latter is free or allocated and, if it is allocated, to which page of which process or processes.
- 并且，os还必须要保留copy of page table，这样是为了在未来可能得手动把逻辑地址翻译成物理地址。并且，It is also used by the CPU dispatcher to define the hardware page table when a process is to be allocated the CPU. 所以，paging会增加context-switch time？？

Hardware Support

- 大部分的操作系统为每个进程整一个page table，然后在PCB里面储存指向这个page table的pointer，当一个dispatcher要执行一个进程的时候，他必须首先要加载register的值，然后**define the correct hardware page-table values from the stored user page table**（所以是在hardware中也有一个page table的？）
- 所以这里讲的就是硬件是怎么implement page table的：
- ***Case 1***：The page table is implemented as a set of dedicated **registers**. 所以很显然，这些拿来做page table的寄存器需要用到相当好的硬件，这样才能保证paging-address的翻译是高效的
  - 举个例子，如果一个地址是16bits，page size 是8kb，那么page table就包含了八个告诉处理器去代表8个entries，这个其实问的是page的数量，一个地址是16bits，那么logical address space就是2^{16}，page size是8KB，也就是2^{13}bytes，那么所以对于这一个地址需要$\frac{2^{16}}{2^{13}}$个page
  - 有一些操作系统允许page table拥有非常多个entry，这说明需要有很多个寄存器，很明显这样子的设计是极其不合理的

    - Rather, the page table is kept in main memory, and a **page-table base register (PTBR)** points to the page table. 改变page table只需要在内存中改变这一个寄存器的值就可以了。
    - 弊端！弊端就是，当我们需要access location i，那么我们需要先用PTBR的值去找到page table，然后用page table的值去找到真正的physical memory，那么可以看到的是，我们access一个byte需要两次memory access，memory access is slowed by a factor of 2，怎么办呢? **TLB!**
- 只有操作系统可以改变memory map（page table）

TLB（Translation Look-aside buffer（TLB））

- 属于缓存
- Associative, high-speed memory
- Each entry in the TLB consists of two parts: a **key** (or tag) and a **value**
- 怎么运作的呢? →当一个item进入TLB的时候，TLB同时把这个item和自己的所有的key来做比较，如果找到了对应的key，就会return对应的值。用TLB搜索相当的快，TLB一般包含64和1024个entries
- TLB里面只存部分的page table entries

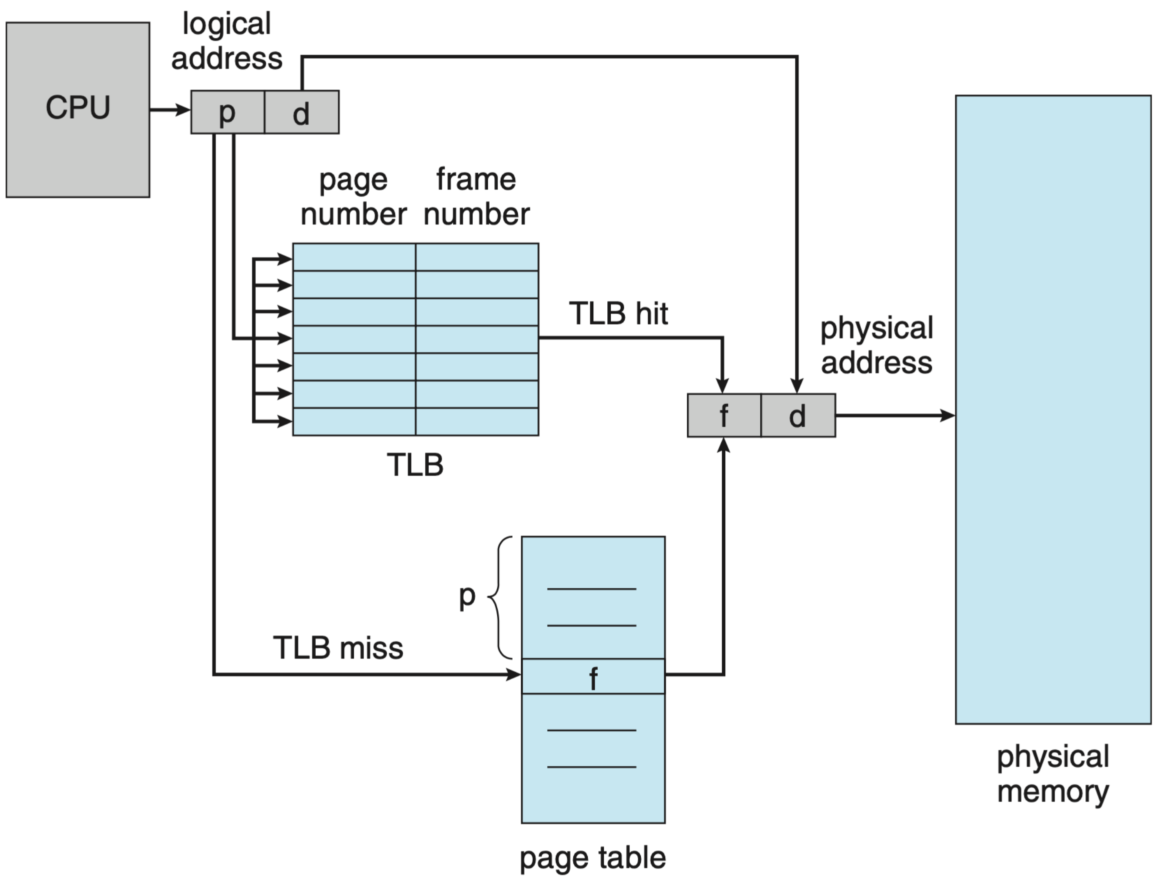
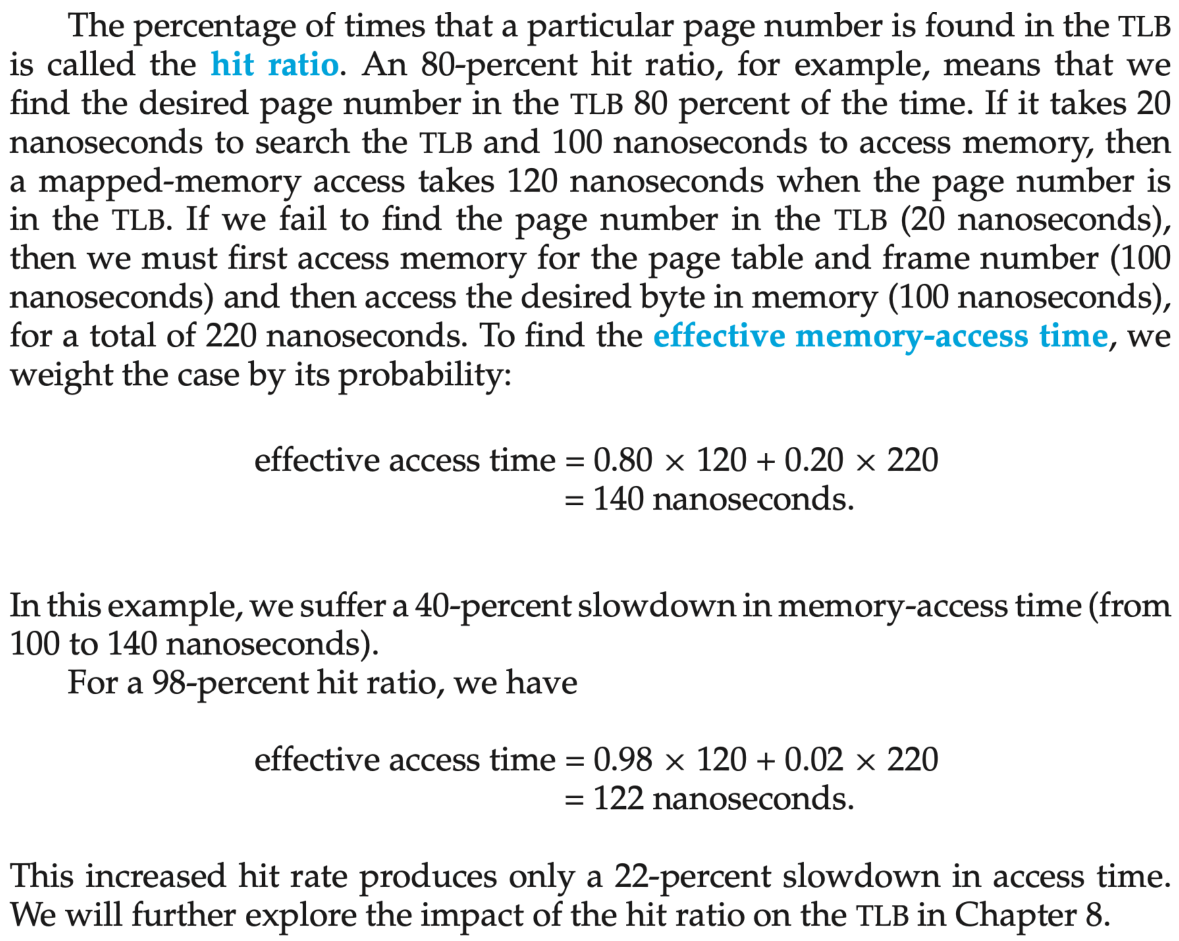


Figure 7.11 Paging hardware with TLB.

    - 当cpu搞出来了一个logical address之后，这个logical address的page number会首先拿去TLB里面，如果中了，那么TLB里面找到frame number，就直接去physical address了，这叫做**TLB Hit**；如果没中的话，就是**TLB Miss**，那么就要像之前一样，有两个memory access

- If the TLB is already full of entries, the operating system **must select one for replacement**. Replacement policies range from least recently used (LRU) to random.
    - 还有一种可能是，有一些tlb是允许一些entry是**wire down**的，这说明了他们不可以从tlb中被删掉
    - Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry. An ASID uniquely identifies each process and is used to provide address-space protection for that process.？？？？ 这个要重新看的
- ***Hit ratio***：The percentage of times that a particular page number is found in the TLB



The percentage of times that a particular page number is found in the TLB is called the **hit ratio**. An 80-percent hit ratio, for example, means that we find the desired page number in the TLB 80 percent of the time. If it takes 20 nanoseconds to search the TLB and 100 nanoseconds to access memory, then a mapped-memory access takes 120 nanoseconds when the page number is in the TLB. If we fail to find the page number in the TLB (20 nanoseconds), then we must first access memory for the page table and frame number (100 nanoseconds) and then access the desired byte in memory (100 nanoseconds), for a total of 220 nanoseconds. To find the **effective memory-access time**, we weight the case by its probability:

$$\begin{aligned}\text{effective access time} &= 0.80 \times 120 + 0.20 \times 220\\ &= 140 \text{ nanoseconds.}\end{aligned}$$

In this example, we suffer a 40-percent slowdown in memory-access time (from 100 to 140 nanoseconds).

For a 98-percent hit ratio, we have

$$\begin{aligned}\text{effective access time} &= 0.98 \times 120 + 0.02 \times 220\\ &= 122 \text{ nanoseconds.}\end{aligned}$$

This increased hit rate produces only a 22-percent slowdown in access time. We will further explore the impact of the hit ratio on the TLB in Chapter 8.

Lecture 22

Protection

- 即使是在运用了paged的机制中，也需要对内存的保护。而这个protection是通过***protection bits（associated with each frame）***来完成的。There bits are kept in page table。
    - 这个bit的作用包括定义一个page是只读的还是可以读写的。当物理地址被计算的时候，the protection bits can be checked to verify that no writes are being made to a read-only page. 如果尝试去写的话→memory-protection violation。在这个基础上，当然可以加入更多的限制：execute-only, ...
- 除了这个bit，在每个page table都有一个叫做valid-invalid bit。valid代表的是当前page是在进程的逻辑地址空间里面，所以是valid的；invalid就是他不在

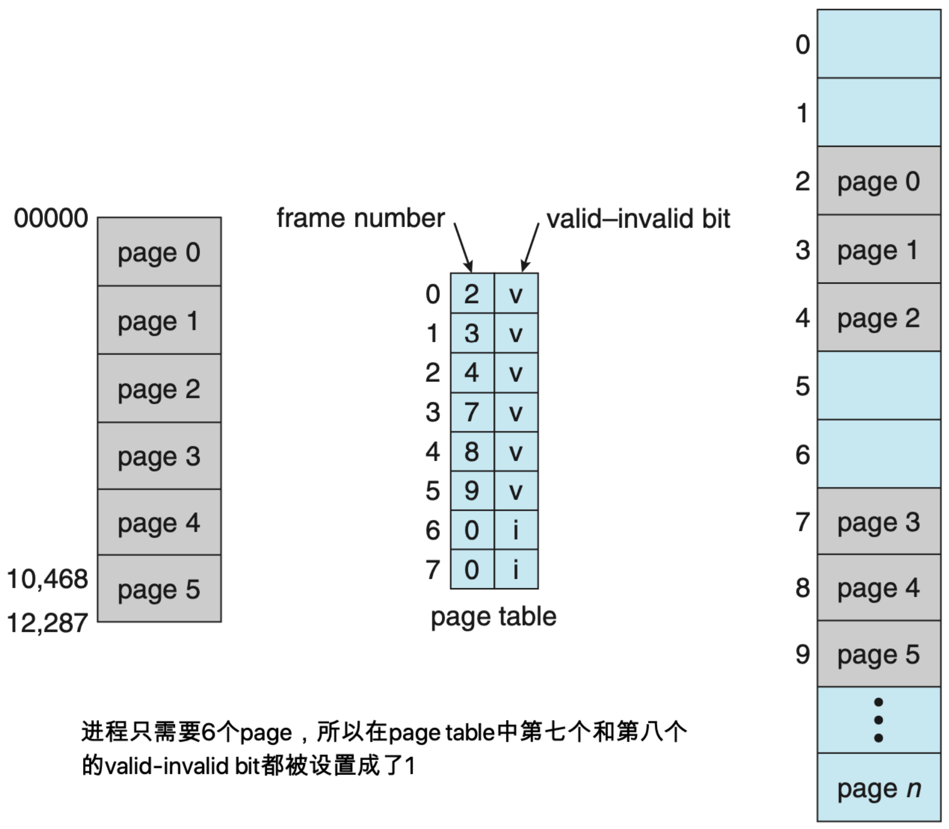
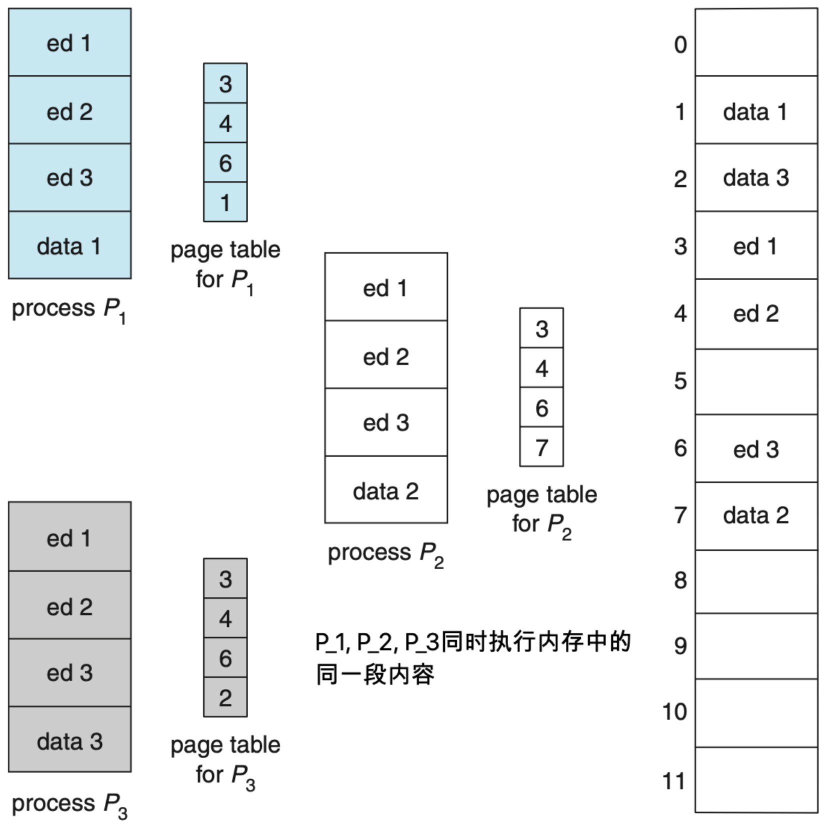


Figure 7.12 Valid (v) or invalid (i) bit in a page table.

    - umm然后这个举例子还搞出了一个internal fragmentation（详情参照page 298）

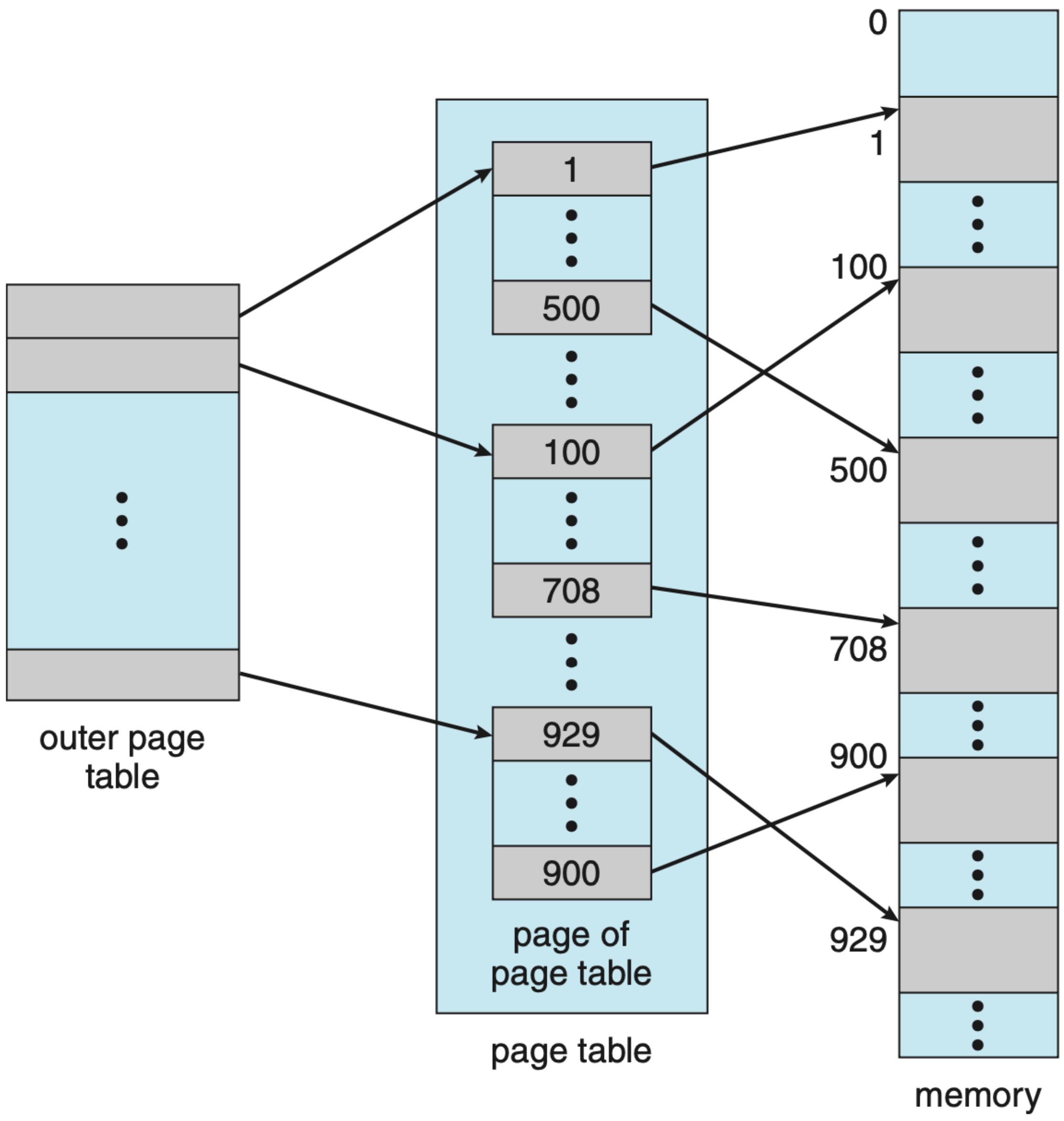
Shared Pages

- 如果当前有很多个进程但执行的都是同一个代码，并且这个代码是**reentrant code**：non-self-modifying code，指的是这段代码在执行的过程中不会有变化。如果符合这种情况的话，那么几个不同的进程可以执行同一段在内存中的代码

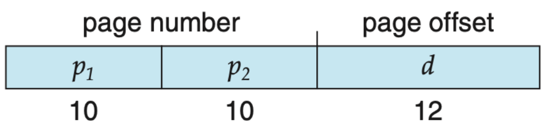


Hierarchical Paging

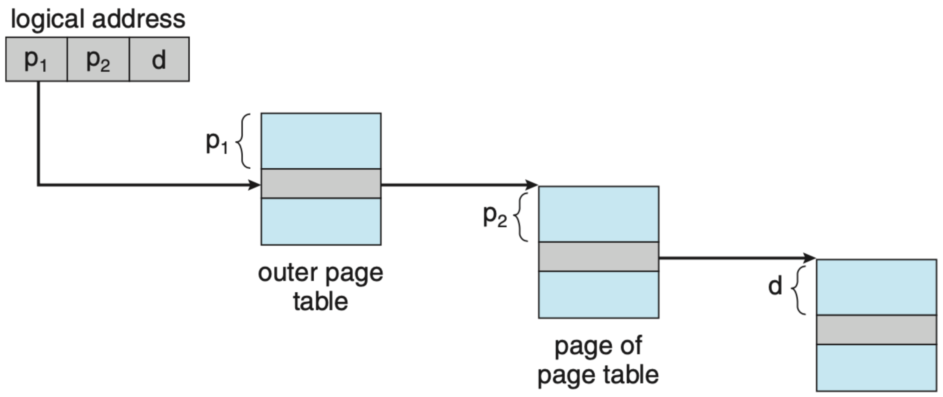
- 现代计算机中，逻辑地址空间非常大，在这种情况下，往往光是page table本身就占用了相当多的内存。 **One simple solution to this problem is to divide the page table into smaller pieces. We can accomplish this division in several ways.**



- 以上是一种办法，**the page table itself is also paged**，以上的条件是一个32-bit的logical address space and a page size of 4KB，那么总共有$\frac{2^{32}}{2^{12}} = 2^{20}$个page
  - If the size of the logical address space is 2^m and a page size is 2^n addressing units (bytes or words), then the high-order *m* − *n* bits of a logical address designate the page number and the *n* low-order bits designate the page offset.
  - 所以现在page number包含20个bits，而page offset则是12bits，又因为现在在本来的基础上page了page table，所以，further divided：



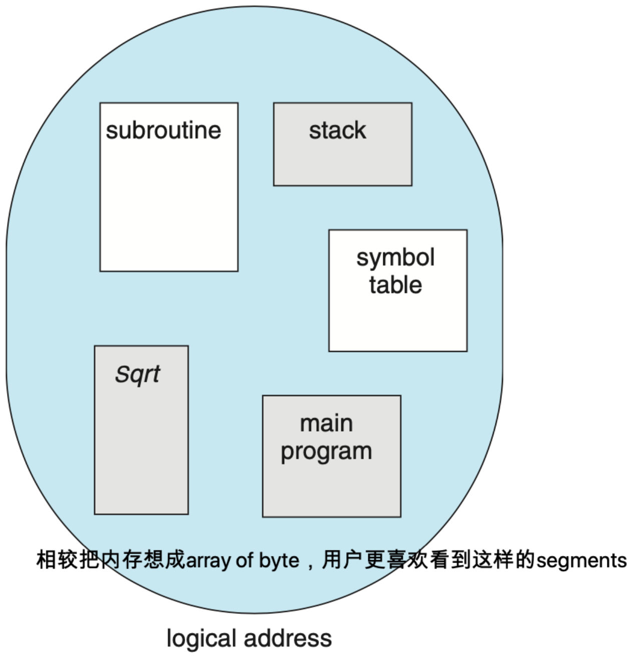
        - 在这个里面，p1是index into the outer page table，and p2 is the displacement within the page of the inner page table. 所以是用p1找到page table的page table（outer page table）中对应的实际的page table中的page，然后p2是inner page table的offset，找到了对应的page之后，再用page offset算physical address中的offset，具体的可以看下图



        - 这种scheme被称为forward-mapped page table
    - 通过计算发现，这样子的操作会让page table的内存大大降低，比如刚刚那个例子，现在有分别两个page table，一个page table有2^{10}个entries，所以现在需要的内存是$2 * 2^{10} * 2^2 = 8KB$ for two page table

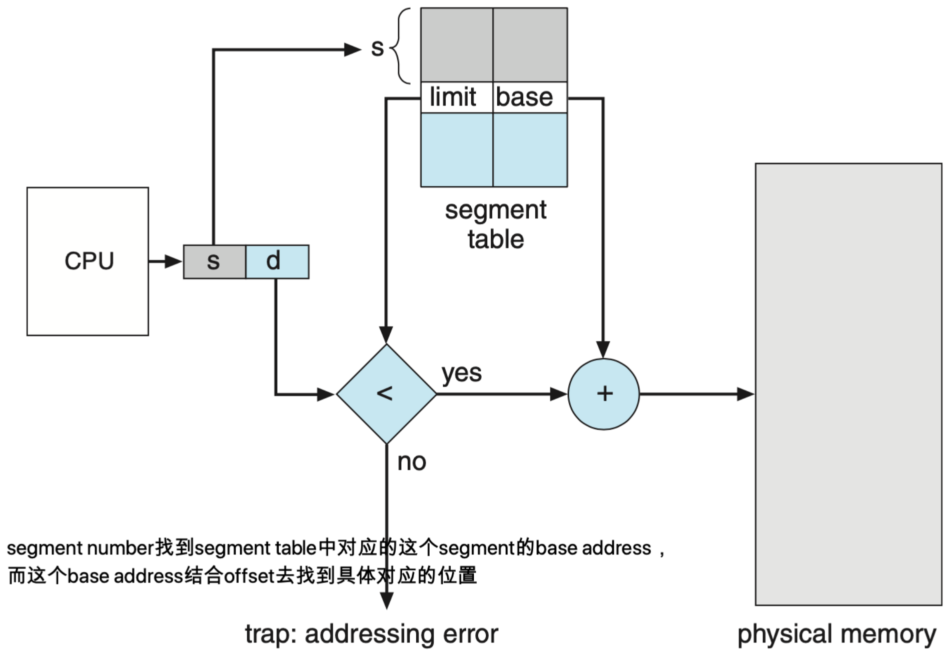
Segmentation

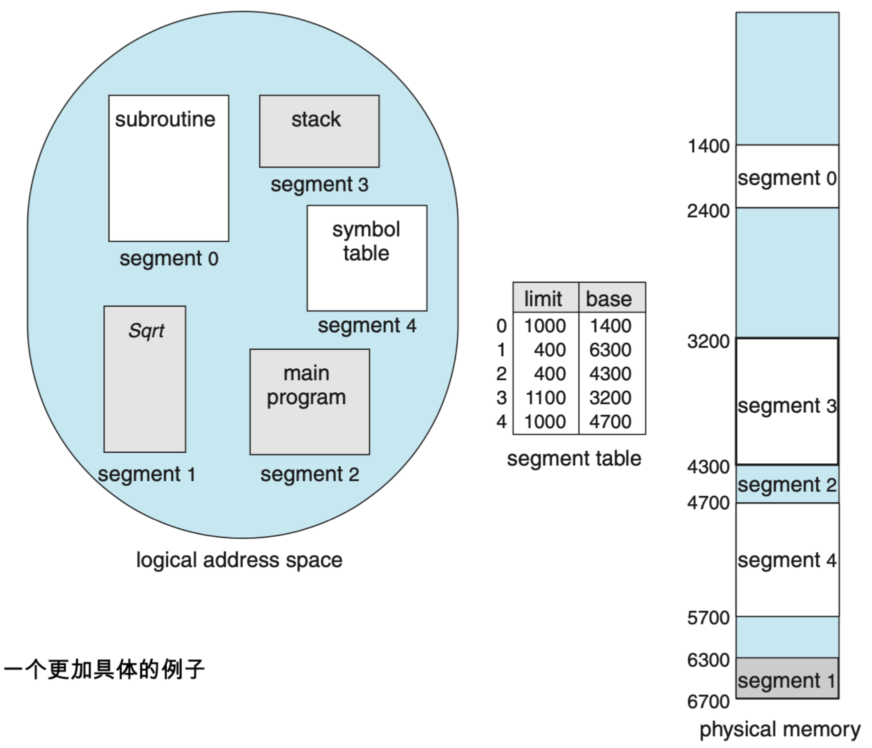
- **Segmentation** is a memory-management scheme that supports this user view of memory.



- Segmentation会把逻辑地址空间看作是segments的集合，每个segment有自己的名字和长度。地址标明了segment的名字，以及offset within the segment。所以**The user therefore specifies each address by two quantities: a <u>segment name</u> and an <u>offset</u>.** 一般为了简单而言，segment name会被segment number给替换掉，所以一般是：**a <u>segment number</u> and an <u>offset</u>.**

- 和paging scheme不同的是：在paging scheme中，系统会把一个完整的逻辑地址拆分成page number和page offset，用以指向真实的物理地址。
- 一般来说，当一个user program被编译的时候，编译器就会为这个input program创建segment：
  - A C compiler might create separate segments for the following:
    1. The code
    2. Global variables
    3. The heap, from which memory is allocated
    4. The stacks used by each thread
    5. The standard C library
- 分段和分页似乎是两种可以单独存在的储存管理，但是当然他们可以一起存在，**分段和分页的区别是**：
  1. 页是信息的物理单位，分页是为实现离散分配方式，以消减内存的外零头，提高内存的利用率。段则是信息的逻辑单位，它含有一组其意义相对完整的信息。分段的目的是为了能更好地满足用户的需要。
  2. 页的大小固定且由系统决定；而段的长度却不固定，决定于用户所编写的程序。
  3. 分页的地址空间是一维的，程序员只需利用一个记忆符，即可表示一个地址；而分段的作业地址空间是二维的，程序员在标识一个地址时，既需给出段名，又需给出段内地址。
- ***Segment table***：Map two- dimensional user-defined addresses into one-dimensional physical addresses. 和之前的很像，segment table中的每一个entry都定义了这个segment存在的base物理地址位置，还有segment limit就是这个segment的长度


s
limit
base
segment table
CPU
s
d
<
yes
+
no
segment number找到segment table中对应的这个segment的base address，
而这个base address结合offset去找到具体对应的位置
trap: addressing error
physical memory



一个更加具体的例子

Lecture 23

Segmentation with paging

- 先把逻辑地址的内容分段，然后再把段给分成页
- 所以assignment 4的最后一题显得非常具有代表性
- 这样子分段之后，segemnt table里面本来存着的base物理地址位置, 现在变成了对应的page table，然后还有segment limit
- 所以在使用这样子的scheme的时候，逻辑地址对于用户来说是这样子的：
  - Segment#: Segment_offset
  - 然后这个东西被操作系统解释释成: Segment#:Page#:Page_Offset
- 也就是说，在追寻一个逻辑地址的时候，先根据segment table base register来追寻到segment table的位置，然后根据这个逻辑地址追寻到对应的segment entry, it would be something like this:

Segment#	Segment size	Page table base address
0	512kb	011
1	512kb	045

- 根据segment table来找到对应的page table base address，然后找到page table，it would be something like this:

Page#	Page base address
0	0742
1	0693

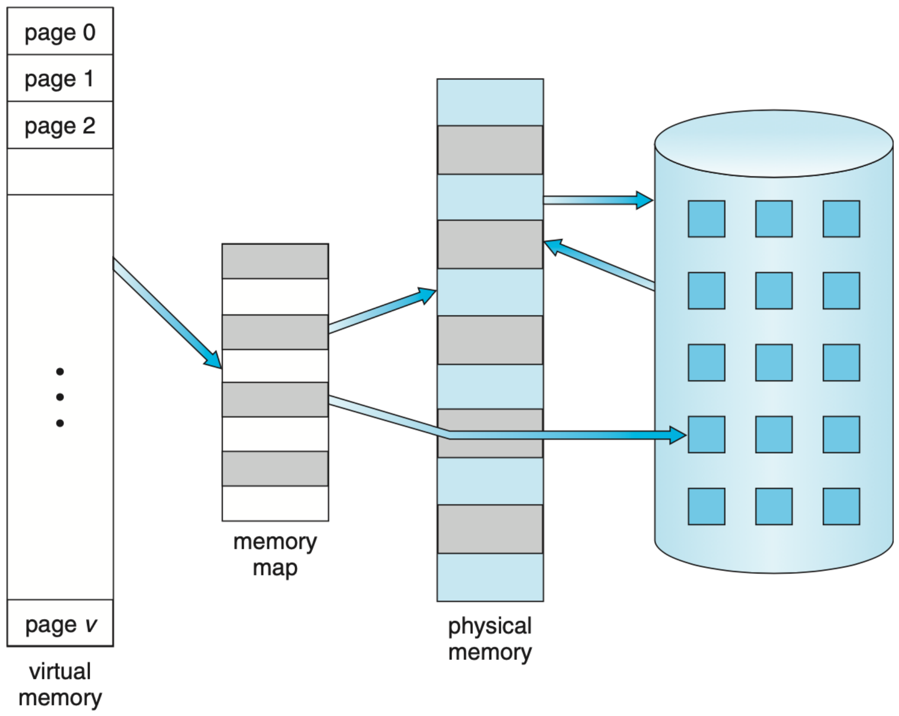
- 然后根据page#找到对应的entry，找到page base address，然后根据page offset锁定最终位置
-**Segmentation with paging and TLB**:
    - 一个tlb里面存着的是：
        - Segment#:Page#:limit: Frame#

Lecture 24

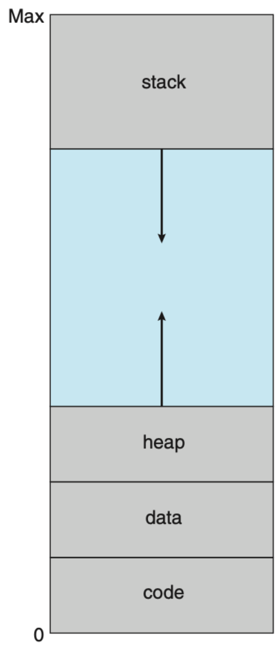
Virtual Memory

- 之前讨论的东西，都是建立在：如果运行一个进程的话，要求进程必须全部都处于main memory之间。而***virtual memory: A technique that allows the execution of processes that are not completely in memory***
- 所以当一个程序进来的时候，他的逻辑地址首先会被分成page，这是毋庸置疑的，知道了哪个部分分成哪个page，但是却不实际上上allocate一个内存里面的frame给这个page
    - **Advantage**:
        1. program can be larger than physical memory
        2. virtual memory abstracts main memory into an extremely large, uniform array of storage, separating logical memory as viewed by the user from physical memory
        3. 允许更简单的文件共享，并且支持实现*shared memory*
        4. 提供了一种有效率的创建进程的方式

- 就是说一般一个完整的程序是不必要的，因为很多时候，一个完整的程序还包括了一些处理error condition的代码，所以一般实际上来说，如果这个error不常见的话，这段代码是不会用到的；或者是array，一般会请求多余自己实际需要的空间；又或者是一些没怎么用到的功能



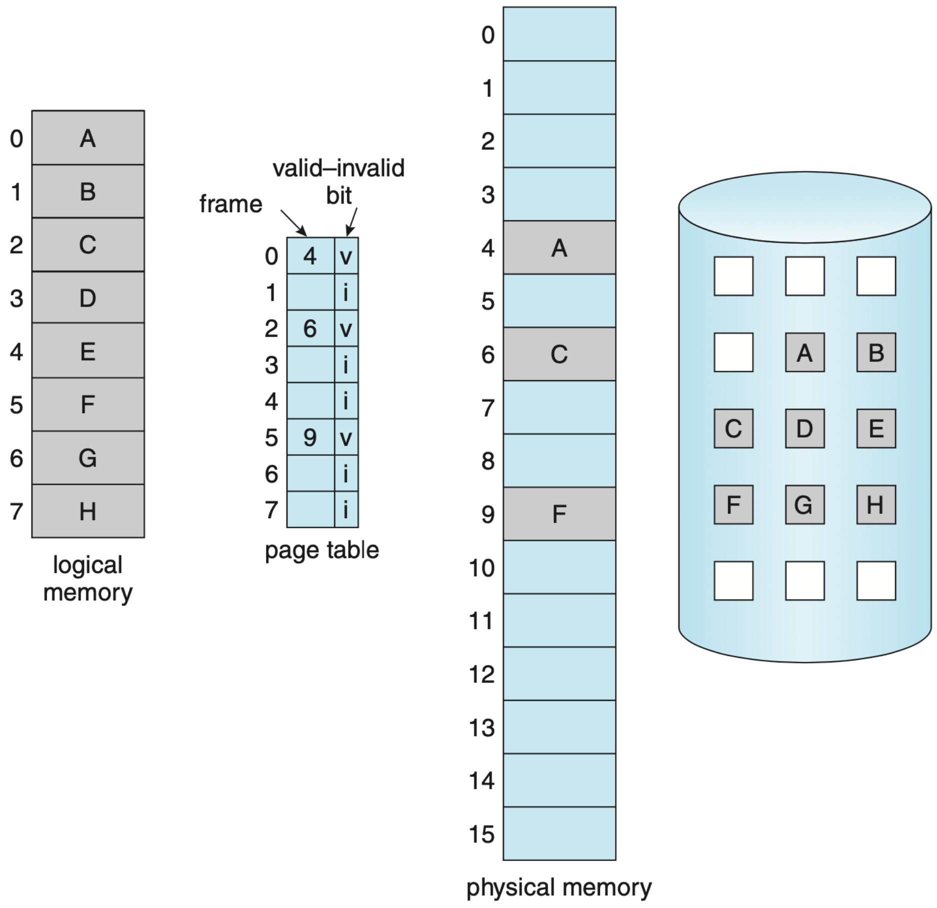
- 以上这个就是一个运用到了virtual memory的示意图



- 以上这张图则展示了一种比较基础的使用方法，这张图就是一个了virtual view of how a process is stored in memory，然后中间那蓝色的空档，则被称为***sparse address space***，这种东西will require actual physical pages only if the heap or stack grows
- **Demand paging**:
  - Load pages only as they are needed，用了这个机制之后，也就是说只有page被需要的时候才

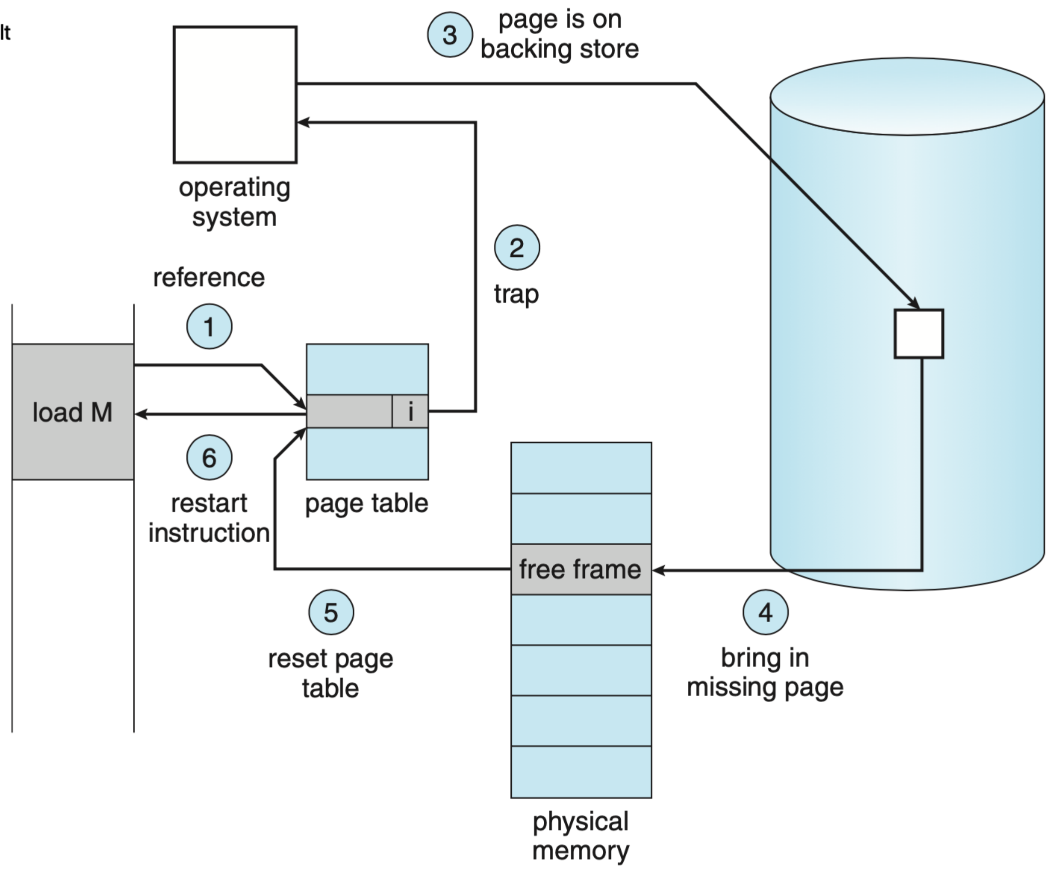
会被loaded，这个和之前讲过的一个概念很像：就是说进程一般都呆在磁盘里面，只有在用到这个进程之后才会把这个进程从磁盘从放到内存中；但是这个deman paging使用到了一个叫做**lazy swapper**的东西：A lazy swapper never swaps a page into memory unless that page will be needed.

- 接下来也会使用**pager**而不是用swapper，因为swap指的是替换整个完整的进程
  - When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again. 所以为了实现这一点，需要一些硬件上的支持，也就是需要区分开哪些分页已经在内存里了，哪些分页还在磁盘里，可以通过之前的***valid-invalid bit***去区分，只不过现在语义上面有细微的差别：valid指的是这个page是legal的兵器，这个page在内存中；invalid指的是这个page要么是illegal的要么不在内存里面



  - 所以如果猜的很准的话，那些invalid分页是不会被用到的，分页不会被用到的话，这样子的操作就是没有问题的
  - 但是如果用到了那些标志了invalid的分页的话→**Page fault**

处理page fault

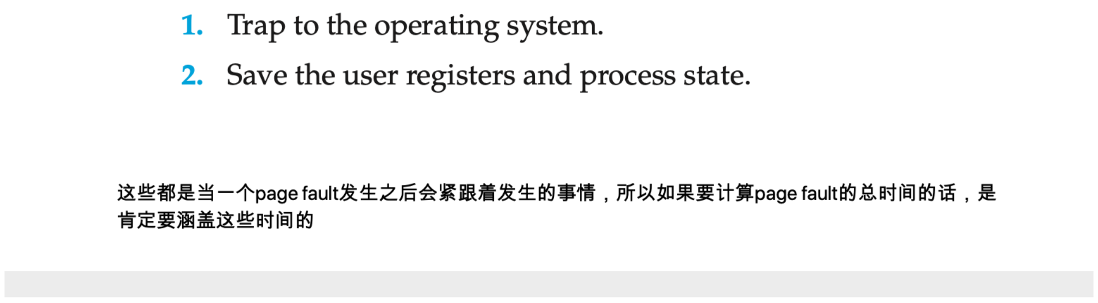


- 首先会看一下这属于什么样式的invalid
  - 如果属于会造成不安全的memory access的话，就根本直接把进程给结束掉
  - 如果属于不会造成不安全，但是是属于还没有放入内存话的分页的话：
    - 内存中找到一个free frame，然后把分页从磁盘读出来放入这个free frame中
    - 然后更新internal table，然后现在对于这个page来说，他的valid-invalid bit已经变成了valid
  - 最后回到刚刚被打断的进程里面
- We check an internal table (usually kept with the process control block) for this process to determine whether the reference was a valid or an invalid memory access.
  - If the reference was invalid, we terminate the process. If it was valid, but we have not yet brought in that page, we now page it in.
  - We find a free frame (by taking one from the free-frame list, for example).
  - We schedule a disk operation to read the desired page into the newly

    allocated frame.
  - When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.
  - We restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory

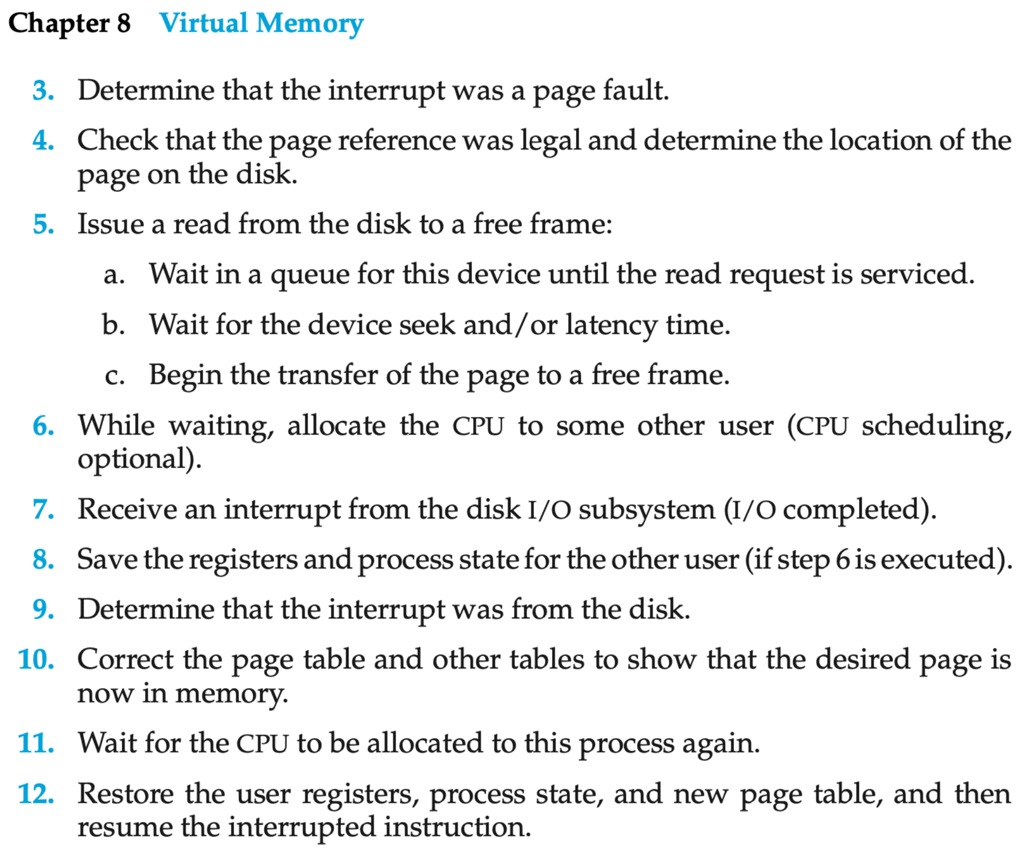
- ***Pure Demand Paging***: **never bring a page into memory until it is required.**
    - We can start executing a process with no pages in memory. 所以当执行第一个指令的时候，啪，就已经是一个invalid bit了，已经构成了一个page fault了，然后找空的frame，这样子的话他所有的需要的page都会经历一次page fault，直到所需要的page被全部放进了内存中
- 比如说现在要计算一个简单的加减法，那么A + B的值从A和B中被fetch出来之后，就会计算，然后存入C中，但是如果C此时出现了page fault，那么全部之前的操作都得重新开始；有些更傻逼的情况是，一条指令会涉及到多个地址，所以很有可能指令到一半就fault掉了
- ***Effective Access time***:
    - 如果一直没有page fault出现的话，那么他的effective access time = memory access time
    - 如果有page fault出现的话：
        - $p\ (probability\ of\ a\ page\ fault) \in [0, 1]$
        - $Effective\ access\ time = (1-p) * ma(memory\ access\ time) + p * page\ fault\ time$
    - 可见如果要计算出这个的话，就必须知道page fault需要多少时间，当一个page fault发生，以下的事情会发生：

- ...ppt上面的page fault handler 35步，傻逼的，主要是它其中涉及到了很多细节，他这个描述的是三个进程在运行，当第一个进程遇到了一个page fault了之后会怎么样，然后还涉及到进程之间的转换之类的



1. Trap to the operating system.
2. Save the user registers and process state.

这些都是当一个page fault发生之后会紧跟着发生的事情，所以如果要计算page fault的总时间的话，是肯定要涵盖这些时间的



Chapter 8 Virtual Memory

3. Determine that the interrupt was a page fault.
4. Check that the page reference was legal and determine the location of the page on the disk.
5. Issue a read from the disk to a free frame:
   a. Wait in a queue for this device until the read request is serviced.
   b. Wait for the device seek and/or latency time.
   c. Begin the transfer of the page to a free frame.
6. While waiting, allocate the CPU to some other user (CPU scheduling, optional).
7. Receive an interrupt from the disk I/O subsystem (I/O completed).
8. Save the registers and process state for the other user (if step 6 is executed).
9. Determine that the interrupt was from the disk.
10. Correct the page table and other tables to show that the desired page is now in memory.
11. Wait for the CPU to be allocated to this process again.
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction.

- 但是通常，我们只需要考虑三个主要的部分：
  1. Service the page-fault interrupt
  2. Read in the page
  3. Restart the process

- The first and third tasks can be reduced, with careful coding, to several hundred instructions. These tasks may take **from 1 to 100 microseconds each**. The **page-switch time, however, will probably be close to 8 milliseconds**. (A typical hard disk has an average latency of 3 milliseconds, a seek of 5 milliseconds, and a transfer time of 0.05 milliseconds. Thus, the total paging time is **about 8 milliseconds, including hardware and software time**.) 这里根据推算，得出了paging time的时间：8ms
- 所以依据刚刚的公式：

$$\begin{aligned} Effective\ access\ time &= (1-p)\backslash{*}200 + p\backslash{*}8ms \\ &= 200 + 7999800\backslash{*}p \end{aligned}$$

- 可以看到，这个时间和page-fault rate是成正比的
- 补坑：**具体怎么计算这些rate之类的**

Page replacement

- Is a mechanism that allows the OS to select a page among many in the RAM and swap it out to HDD to free a frame to host a new page from the HDD
- 所以这里探讨的主要是，如果没有available的frame的话，cpu总得做些什么：把现在霸占着frame的page给换出去，然后把当下需要frame的page给换进来
- **Reference Page**：First, for a given page size (and the page size is generally fixed by the hardware or system), we need to consider only the page number, rather than the entire address. Second, if we have a reference to a page *p*, then any references to page *p* that *immediately* follow will never cause a page fault. Page *p* will be in memory after the first reference, so the immediately following references will not fault.
- 所谓的**Reference string**就是其对应的page number
- 然后重点介绍了三个page replacement algorithm
  - FIFO
  - OPT: Clairvoyant replacement algorithm, replaces the pages that will not be used for the longest period of time. 因为要实现这个的话必须要知道additional infromation about reference string所以很难implemented
  - LRU: replaces the pages that had not been referenced for the longest period of time.
  - MFU: Most frequently ussed,
- 因为在有些时候，不同的程序设计导致了用系统原先设计好的page replacement算法会比较低效率，所以操作系统也提供了另外一种服务：Raw I/O, 让程序自己specify自己的special-purpose storage

Side topic: BUFFER(缓冲) vs CACHE(缓存)

- 简单说，BUFFER的核心作用是用来缓冲，缓和冲击。比如你每秒要写100次硬盘，对系统冲击很大，浪费了大量时间在忙着处理开始写和结束写这两件事嘛。用个BUFFER暂存起来，变成每10秒写一次硬盘，对系统的冲击就很小，写入效率高了，日子过得爽了。极大缓和了冲击。
- CACHE的核心作用是加快取用的速度。比如你一个很复杂的计算做完了，下次还要用结果，就把结果放手边一个好拿的地方存着，下次不用再算了。加快了数据取用的速度。
- 还存在一些别的情况需要我们自定义minimum number of frame，一些比较极端的情况是：因为如果一个指令在执行到一半的时候page fault出现了，那么这个指令就必须得重新执行，所以如果一条指令需要是三个frame，而此时只有一个free frame，那么这个指令将永远不会完成
- 所以怎么把适当数量的frame划分给process就需要适当的思考(Allocation of frame), 其中有几个scheme

   1. equal allocation
   2. proportional allocation
- Global Allocation: Global replacement allows a process to select a replacement frame from the set of all frames, even if that frame is currently allocated to some other process; that is, one process can take a frame from another.
- Local Allocation: Local replacement requires that each process select from only its own set of allocated frames.
- 就是在有些系统里，纵使内存还是那个内存，cpu还是那个cpu，但是cpu在加载某一部分内存的时候会比较快，这样的系统被称为Non-Uniform-memory access system

Thrashing

- 有时候一些可能出现的事情是如果一个进程没有足够的frame，那么他在运行的时候，很快就要会陷入page fault, 那么他就要去处理，所以他花在paging上的时间比他实际执行指令的时间都还要长，这样就会造成**thrashing(颠簸)**, a process is thrashing if it's spending more time paging than executing
- 在课本Pg370有描述早期的paging system的一个会导致系统效率变得很低的现象
  - 基本上就是：CPU利用率变低 -> degree of multiprograming上升(global allocation used here) -> available frame随着进程数量变多而变少 -> 对于进程来说page fault rate上升 -> 进程被卡在等待page的过程中 -> CPU利用率变低然后不停循环
- 用local scheme可以解决这个问题(不会抢走别的进程的frame), 但不能彻底解决，只能把这个问题引向另外一个问题
- 要真正的解决颠簸问题，我们只可以**为进程提供他所需要的全部frame**
  - **Locality**: A locality is a set of pages that are actively used together (Figure 8.19). A program is generally composed of several different localities that may overlap, locality是由一个进程的程序结构和他的数据结构来决定的.
  - 所以*Locality model*需要进程把自己的basic memory reference给展示出来
  - 所以系统会根据这个locality model去为他分配frame
  - ***Working set的概念***

$$\Delta: \ defines\ how\ many\ page\ reference\ will\ be\ in\ one\ working\ set$$
$$WORKING\ SET\ : The\ set\ which\ contains\ most\ recent \Delta page$$

  - 然后还可以得出以下的公式：

$$D = \sum WSS_i$$
$$D\ is\ the\ total\ number\ of\ demands\ for\ frames$$

  - 计算出来之后就可以通过这个来判断了

- 同时还可以根据PFF(page fault frequency)来判断给一个process的frame是多了还是少了，所以也可以根据这个值来控制thrashing(有thrashing的时候，page fault rate会明显变多)
- https://blog.csdn.net/qq_28602957/article/details/53821061
- 以上这篇文章讲的非常好，包括一些工作集的变化
    - 当进程在工作的时候，一般会经历几个阶段
        - 根据局部性原理，进程会在一段时间内相对稳定在某些页面构成的工作集上
        - 当局部性区域的为之改变时，工作集大小快速变化
        - 当工作集划过这些页面之后，工作集再次稳定在一个局部性稳定阶段
- 工作集用进程过去某段时间内的行为作为未来某段时间内行为的近似。

Monitors for synchronization

- Timing error still occur when semaphores are used(rarely tho)
- Various types of errors can be generated easily when programmers use semaphores incorrectly
- 所以为了尽可能的避免这种失误的出现，发明了high level language constructs: monitor
- **Abstract data type(ADT)**: encapsulates(压缩) private data with public methods to operate on that data
- 然后具体来说什么是MONITOR(管程/监视器)
    - Monitor type是ADT的一种，提供一系列的由programmer定义的操作，这些操作为处于管程里面的子程序提供互斥操作，下面是管程的wiki定义
    - 管程 (英语：Monitors，也称为监视器) 是一种程序结构，结构内的多个子程序（对象或模块）形成的多个工作线程互斥访问共享资源。这些共享资源一般是硬件或一群变量。管程实现了在一个时间点，最多只有一个线程在执行管程的某个子程序。与那些通过修改数据结构实现互斥访问的并发程序设计相比，管程实现很大程度上简化了程序设计。
- 以下是管程的syntax

```
monitor monitor name
{
  Condition variables: x1

  Local variables: v1

  procedure P1(...) {
  //
  }

  procedure P2(...) {
  //
  }

  procedure Pn(...) {
  //
```

```
    }

    initialization code(...){
    //
  }
  }
```

- 所以上面的syntax的意思是：如果想要执行定义在这个monitor中的子程序（procedure），则必须互斥地执

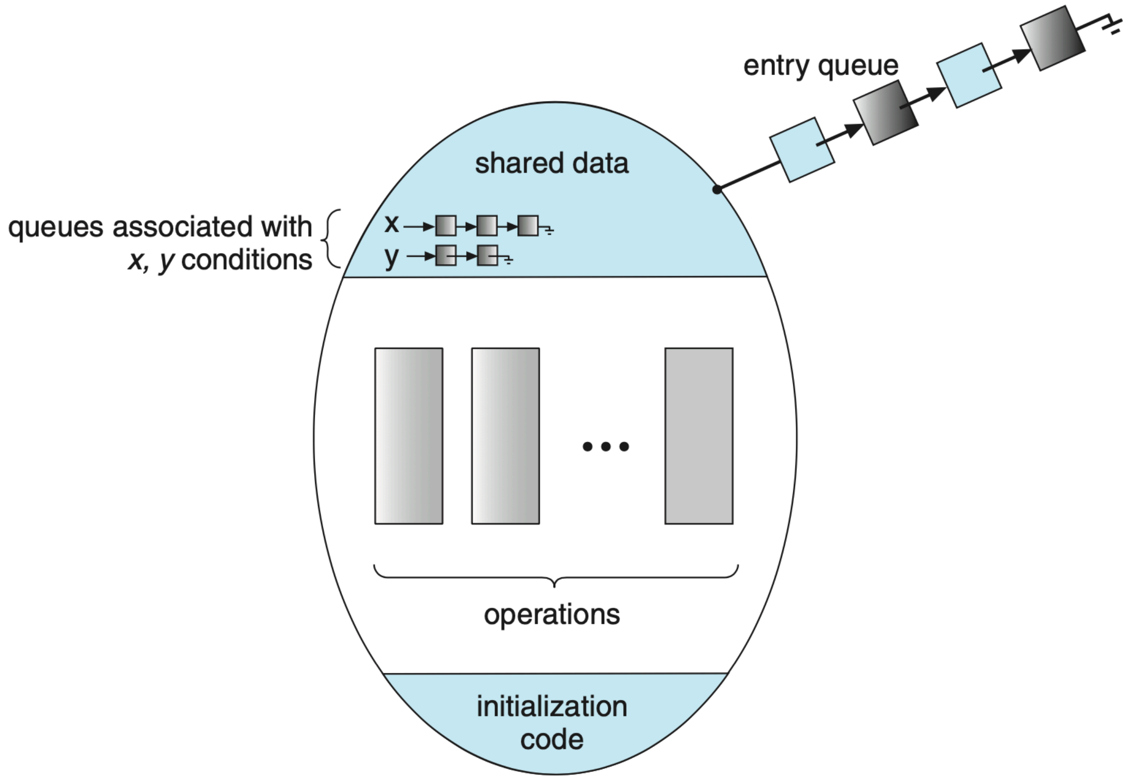


Figure 6.18 Monitor with condition variables.

- **Condition type**
  '''C
  condition x, y;

x.wait();

x.signasignasignal();
'''
 - condition type的变量和信号量非常像，`x.wait()`指的是：执行了这个指令的进程会被suspended until another process invokes`x.signal()`
  - A process without any condition variables is called a *pure monitor*
 - 值得注意的一点是：当某个进程执行`x.signal`的时候，理论上来讲，他不会离开自己的criticla setion，因为`x.signal()'的定义式：The x.signal() operation resumes exactly one suspended process.那么为了达到互斥的效果，不同的系统会采取不同的机制：

1. Signal and wait: Process which execute the x.signal exit its critical section immediately and

yield the right to the process which has been resumed
    2. Signal and continue: Process which execute x.signal continue its execution, the process which has been resumed must wait until this process finish

- Java Monitor != Traditional critical section
  - 传统的cs要求只要有一个进程进入了这段代码，另一个进程就不允许进入这段代码了。但是如果是java monitor的话：
    - Declaring a method synchronized implies that only one thread may invoke that method on a particular object at any given time.
- Notify VS Notifyall
  - 首先，要知道等待池和锁池的区别：
    - **等待池**：处于等待池中的进程，不会去竞争锁。执行了wait的进程会进入等待池中
    - **锁池**：处于锁池中的进程，会去竞争锁，也就是说，只要有锁被wait释放出来了，处于锁池中的线程就都可以去竞争
  - 就要先搞清楚java monitor中几个function的作用是什么：
    - 首先`x.wait()`这个函数，虽然他的名字跟信号量中的wait是一样的，但他们的作用完全不一样，这个wait()只干两件事：
      1. Suspend the thread which called. 把执行了这个函数的线程放入等待池中
      2. Unlock monitor. 把这个线程本来持有的锁释放出来
    - 其次`x.notify()`和`x.notifyAll()`这两个函数的基础功能是一样的
      - 唤醒一个或多个在等待池中的线程，并且把一个或多个在等待池中的线程放入**锁池**中
      - `x.notify()`: 唤醒一个线程，把这一个线程放入锁池中
      - `x.notify()`: 唤醒全部处于等待池中的线程，并且把全部处于等待池中的线程放入**锁池**中
  - 有了以上的基础，就知道为什么`x.notify()`可能会造成死锁了。想象一下，当一个拿着锁的线程执行`x.notify()`, 这时，有一个处于等待池中的线程(A)被移到了锁池中, 那么当目前的线程执行了wait之后，他把锁释放出来，并且把自己放入等待池中，这个时候，位于锁池的线程A就抢到了这个锁，并且开始运行自己的东西，但是如果他在执行wait的时候，并没有执行notify/notifyall，此时，虽然锁被释放出来了，但全部的线程都处于等待池中，他们想要锁，但是他们不处于锁池中，所以他们没有资格抢锁，于是就死锁了。但是只要执行一次notifyall，全部的线程都会进入锁池中抢这个锁。再加上在使用notify的时候，如果有多个线程在等，把哪个线程移入锁池也是完全不由程序员控制的。

CPU Scheduling

- CPU scheduling: Switchin the CPU among processes, by doing this, os can make the computer more productive
- 整个的目的是提高CPU的利用率，其实CPU Scheduling系统的idea是相当简单的：A process is executed until it must wait, 所以如果没有适当的scheduling system的话，当一个进程在wait的时候，比如在执行I/O request的时候，CPU就等在那里，浪费掉了很多的时间，所以最基础的概念就是，当一个进程必须要等待的时候，os把cpu夺回来，并且分配给另一个进程，不浪费时间

CPU Scheduler

- CPU scheduler是负责把一个进程从很多个进程里面选出来，并且分配到闲置的CPU中， 这个进程的队列（就是供CPU scheduler选择进程的地方）可以被很多种不同的算法implement
- Dispatcher: The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler.

Preemptive Scheduling

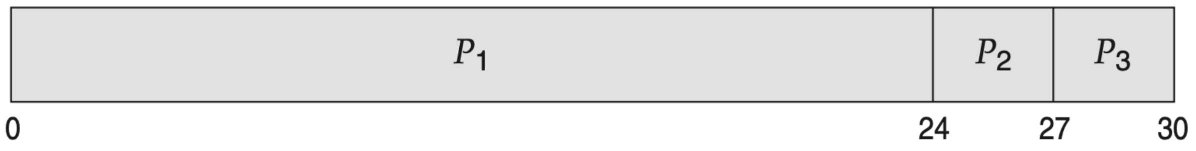
- CPU Scheduling Decision may take place under the following four circumstances:
  1. When a process switch from the running state to the waiting state
  2. When a process switch from the running state to the ready state
  3. When a process switch from the waiting state to the ready state
  4. When a process terminates
- 当scheduling只在1，4发生，这样子的scheduling scheme被称为*nonpreemptive/cooperative*
  - nonpreemptive scheduling scheme发生的时候，被分配到cpu的进程会一直占用着cpu直到下一次scheduling发生
- 当scheduling也在2，3发生，这样子的scheduling scheme被称为*preemptive*
  - 强占scheme还需要额外的防止race condition的软硬件
  - 在这种机制下面，有一种会造成混乱的可能：当一个进程在调整内核的数据的时候，他被preemptive了

Scheduling Criteria

- 这个指的是一些拿来判断scheduling algorithm的标准
- CPU utilization
- Throughput: One measure of work is the number of processes that are completed per time unit, called throughput
- Turnaround Time: The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O. 一个进程从提交到完成的所用时间。
- Waiting Time: Waiting time is the sum of the periods spent waiting in the ready queue. 进程在就绪队列中等待所用时间之和。
- **CPU burst**: CPU burst is when the process is being executed in the CPU
- Response Time: This measure, called response time, is the time it takes to start responding, not the time it takes to output the response.某一进程从发出调度请求，到其开始得到CPU调度器响应，其间所经历的时间

Scheduling Algorithms

- First-come, First-Served Scheduling(FCFS): 和FIFO很像：The process that requests the CPU first is allocated the CPU first. 并且要implement这个真的就是用一个FIFO, 缺点是waiting time会比较长，比如，如果先进来的那个进程要消耗很长的时间的话，后面就要等很长时间, 下面是一张**Gantt chart**, illustrate了用FCFS的scheme的话他的等待时间是由多长：



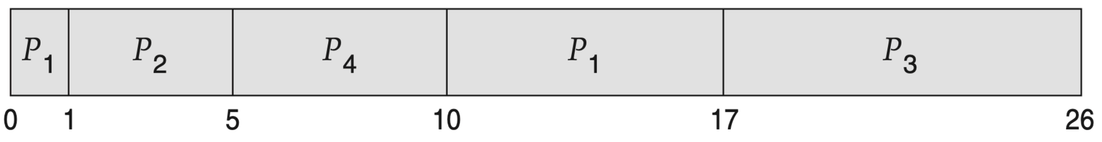
- 但是如果p1, p2, p3的执行顺序是反着的话，他们wait time的均值就会比原来低很多，所以对于FCFS的scheme来说，他们的wait time的表现很大程度上会被进程的burst time所影响
- 并且还会产生convoy effect（护送效应）：就是许多进程一直一个大进程去完成. 并且要注意的是FCFS是个nonpreemptive的scheme，当一个进场占着cpu了，别的进程就不可以从他手上夺走CPU
- Shortest-Job-First Scheduling(SJF)/Shortest-next-cpu-burst-algorithm: 这个algorithm也非常好理解：当CPU available的时候，他衡量所有进程，然后会把cpu assign给那个CPU burst最短的进程, 如果cpu burst of两个进程是一样的话，就会直接使用刚刚的FCFS scheme. 使用SJF被证明了是optimal的，这个optimal指的是他在wait time上面，it gives the minimum average waiting time for a given set of processes.
    - 真正难的地方在于怎么才可以知道进程的cpu burst time。-> 在short-term scheduling里面里面是根本不可以完美的实现sjf的，所以如果想要实现类sjf的算法，他对于cpu burst的评估只有预测下一个cpu burst。怎么估计的：

$$\tau_{n+1}(exponential\ average) = \alpha\tau_n + (1-\alpha)\tau_n$$

- sjf既可以是preemptive也可以是nonpreemptive的，如果是preemptive的sjf scheduling的话，有的时候会被称为：shortest-remaining-time-first scheduling，以下这张图介绍了preemptive的sjf的工作方式：

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

If the processes arrive at the ready queue at the times shown and need the indicated burst times, then the resulting preemptive SJF schedule is as depicted in the following Gantt chart:



如果新进来的进程的burst time < 现在正在运行的进程剩下的时间，现在的进程被新进来的进程抢先

Process P_1 is started at time 0, since it is the only process in the queue. Process P_2 arrives at time 1. The remaining time for process P_1 (7 milliseconds) is larger than the time required by process P_2 (4 milliseconds), so process P_1 is preempted, and process P_2 is scheduled. The average waiting time for this example is $[(10 - 1) + (1 - 1) + (17 - 2) + (5 - 3)]/4 = 26/4 = 6.5$ milliseconds. Nonpreemptive SJF scheduling would result in an average waiting time of 7.75 milliseconds.

- Priority Scheduling: This algorithm indicates that a priority is associated with each process, and the CPU us allocated to the process with the highest priority. Two processes with the same priority will be allocated base on FCFS. Therefore, observe that sjf is a special case of priority scheduling in which the "priority" it indicates is the CPU burst time. **Some system use lower number for high priority while some system use low number for low priority, it really depends**
  - **Internal Priority**: Compute inside os, therefore priority value can only be something that is knowable to OS such as the cpu burst time, memory requirement, number of open files...
  - **External Priority**: Compute outside os, set by criteria outside the knowledge of os such as how important this process is , or how much did someone pay for thi sprocess to be exeucted, etc.
- Can be either preemptive or nonpreemptive. For priority scheduling, one obvious problem is **Starvation**, some process with low priority might be leave indefinately.
  - There is a solution for that: **aging**, gradually increasing the priority of processes that wait in the system for a long time.
- Round-Robin Scheduling(RR):
  - Designed especially for time sharing system
  - Somewhat like FCFS but with preemption
  - Defined a time unit: **time quantum**: generally from 10 to 100 milliseconds
  - Ready queue is treated as a circular queue

- How does it work?: The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.(Therefore, every process gets no more than 1 time quantum)

- If a process has a cpu burst time less than 1 time quantum: let it end; If a process has a cpu burst time larger than 1 time quantum: it gets preempted
- wait time is often long, really depends on length of time quantum it choose: if time quantum is long, the rr = fcfs; if time quantum is extreme small, then the RR approach is called processor sharing and (in theory) creates the appearance that each of n processes has its own processor running at 1/n the speed of the real processor.
- Another problem to consider is the time of context switch. If time quantum is small, we may need more time to perform context switch
- Multilevel Queue Scheduling: This scheme partition processes into different class. Each queue has its own scheduling algorithm. Also, there must be scheduling among the queues and it's often being implemented as fixed-priority preemptive scheduling, following are queues for an example scheme:
  1. System Processes
  2. Interactive Processes
  3. Interactive editing Processes
  4. Batch Processes
  5. Student Processes
  - In above case, each queue has **absolutely priority** over lower priority queue.
  - Absolute Priority? -> No process in the batch queue, for example, could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty. If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted.

Message passing: Interprocess Communication(IPC)

Lecture note

- How processes communicate each other for collaboration
- One thing that need to know is that synchornization method like semaphore, monitors are solution for processes/threads which executed on the same motherboard(share CPU and memory)
- What if we want processes to communicate throgh a *network* (not on the same computer) -> Message Passing
  - `Send(): just to send information through network`
  - `Receive(): receive information from network`
- issues that can happen during communication(message passing)
  - message loss: we can deal with this by receiver sending acknowledgement/reassure
  - message duplication

    - transfer delay
    - error detection: some bits inside the message may get flipped
    - process identification: processes in network need to be identified individually
    - security
        - authentication: message need to be sent by authented sender
        - confidentiality: only receiver and sender know the message
        - availability: server need to be available
        - non-repudiation: need to have evidence to show the message is sent by the sender
- direct communication: Directly send message from one process to another
- Indirect communication: Process send message to an address in between; Process get message from the address in between
    - use a mailbox, a address in between
- Type of Message Passing:
    - Blocking:
        - Blocking send: Sender process is blocked while the message that it has sent has not been received
        - Blocking receive: Receiver process is blocked till it receives a message
    - Nonblocking:
        - Nonblocking send: Sender process continue executing while the message has been send
        - Nonblocking receive: Receiver process receives a valid message or null
- rendezvous(Both sender and receiver are blocking)
- **Sockets**: Endpoint communication objects used to establish connections between two processes in order to communicate over network or within localhost
    - `(192.168.1.10:1255) <-> (130.15.126.132:80)`
    - Above code is somewhat like a client and a server, server socket **waits** for incoming connections from clients on a specific port; While a client socket **initiates a request** for a connection with a server
- **Remote Procedure Calls**: A communication concept that consists of allowing a process running on a computer A, to remotely execute a procedure by a process running on another computer B, in the same way as if it occurred. This is just lile RMI
- So a server implements procedure and waits for clients to call this procedure, and then server returns what it returns to clients which calls this procedure
- More specific way about this can be seen in lab6

Operating System Security

- Basic Idea: Computer resources must be guarded against **Unauthorized access, malicious destruction or alteration, and accidental introduction of inconsistency**
- Notice that the secure is for the whole **Infromation System**

- Some terms that needed to be noted:
    - **Intruder/Cracker**: Those attempting to breach security
    - **Threat**: The <u>potential</u> for a security violation such as the discovery of a vulnerability
    - **Attact**: The <u>attempt</u> to break security
        - Cyberattack: Use specifically software/computer to attack
    - **Intrusion**: Sucessful attack
    - **Cracker**: People who just want to bypass the system, unlike hacker who actually try to attack
- Several forms of accidental and malicious security violation:
    - Breach of confidentiality: Unauthorized reading of data
    - Breach of integrity: Unauthorized modification of data
    - Breach of availability: Unauthorized desturction of data
    - Theft of service: Unauthorized use of resource
    - Denial of service: Preventing legitimate use of the system, attack occurs when legitimate users are unable to access information system
- Masquerading: One participant in a communictaion pretends to be someone else
- Replay Attack: The malicious or fraudulent repeat of a valid data transmission such as repeat of a request to transfer money
- Man-in-the-middle attack: An attacker sits in the data flow of a communication, masquerading as the sender to the receiver, and <u>vide versa</u>

- Safety VS Security;
    - A system is safe if the system operates as it was intended to operate under the circumstances for which it was designed(normal circumstances)
    - A system is secure if the system operates as it was intended to operate under all circumstances
- Two types of threats: Accidental/Intentional
    - Notice that software bugs is counted as accidental
    - There are some definition on lecture state the author of intention
- Security can be established from:
    - Physical aspect: people don't have easy access to the actual device, and device is kept in safe environment
    - Social aspect: User should know the consequences of being attack: which means user don't put password in their sticky note, i.e. User should at least protect their infromation
    - Application aspect: Consists of securing applications running on the system
    - Software aspect: Do not install untrusted software
    - Network aspect: Secure network traffic
    - OS aspect: Keep system secure: plan for backups before updates, consider update, install anti-malware software...

- **A chain is as strong as the weakest link, and the weakest link is always the user**
- Fundamental security services
    - authentication: message need to be sent by authented sender
    - confidentiality: only receiver and sender know the message, protect information from being disclosed to unauthorized parties
        - Symmetric Encryption: receiver use the same key(key that sender used to encrypt) to decrypt
        - Asymmetric Encription: sender use public key to encrypt, receiver use private key to decrypt
        - No-key protocol
    - integrity: Protect information from being modified
        - So using hash functio can see whether a file has been modified
    - availability: server need to be available, guarantee that information system resources are available at all time
    - non-repudiation: Nobody denies having done(executed, sent, received...) something in the system
- **Operating systems threats**
    - Configuration & programming errors: -> buffer overflow, integer overflow
    - Denial of Service: flodding, mail-bombing(send a email to someone, a very large one, but zip the file, the receiver download the zip file thought it's small then unzip the file. BOMB)
    - Squatting: illegal hosting(use your computer to store illegal file)
    - Malware: Viruses, worms, trojan, spyware...(病毒)
    - Spoofing: Use another IP address to do bad things, DNS spoofing
    - Social engineering: Too much confident, basically is just to trick you into some website
        - `x@y`: This url takes you to y instead of x!
    - Boring: Spams, pop-ups, pop-unders, hoax...
    - Reconnaissance: Port scanning, OS fingerprinting, footprinting...(Like scan to see which port of your computer is open)
- **Malware**: a program designed for causing harm to a computer system(Very concrete)
    - There are several difference between malware regard how they are propagated and how they infect system in slide
    - Viruses: attach itself to a file/program enabling it to spread from one computer to another
    - Worms: Does not infect files but has the ability to duplicate itself and send a copy of itself through the network
    - Trojan Horse: Comes in the form of a nicely useful software but once installed, it destroyed your system, used to setup backdoors
    - Bots: run on user os(zombie), and *waits* for commands from a remote attacker which controls it

- Rootkit: Installed after taking over a computer. It's used for erasing evidences to avoid being backtracked
    - Ransomware: Threatens victims to publish their private data or perpetually block access to it unless a ransome is paid
    - Spyware: Used to gather information about users activity over the internet
    - Keylogger: Used to intercept what users type on their keyboard
    - Backdoor: Create vulnerability on the victim system so that the attacker can come back anytime
    - Pop-up: Unsolicited web windows that pops up to display an advertisement or a warning
    - Hoax: Jokes, false news, or bad advertisement about company
- *Trap door*: Piece of code inserted into a program by an insider to bypass some normal checks
- **Protection tools**:
    - Anti-virus: Detect and eliminate different types of malware: signature-based, sandbox, behavior-based
    - Firewall: Barrier to filter traffic coming from and going to the Internet
    - Anti-spyware
    - Integrity Control: Compare system's intergrity with a previous state
    - IDS(Intrusion Detection Systems): Detect ongoing external/internal attacks and *notify* system administrator
    - IPS(Intrusion Prevent Systems): Detect ongoing external/internal attacks and *prevent* it
    - Honypots: Collect traffic of attacker to learn from then
- **Protection**: a security term used to refer to the set of mechanisms used for controlling the access of processes or users to the computer resources
    - Data confidentiality: The owner of a file should be able to specify who can read/copy/execute the file
    - Data integrity: The owner of a file should be able to specify who can modify the file
    - Data availbility: The OS should apply mechanism to make sure that nobody can disturb the system to make it unusable
    - User privacy: The OS protects individuals from misuse of information about them
    - User authentication
    - System reliability: The OS should apply mechanisms to provide fault tolerance an high reliability.

Buffer Overflow

- Is an attack that exploits a bug in a program to cause the bugged program <u>jump</u> to an arbitary program for execution(arbitary program is usually generated by malicious program)
- Two types of buffer overflow
    - Overflow on the stack
        - *Sending more data than the program expects and force it to **overwrite** the return address on the stack with the address of an exploit code(a.k.a shellcode), so that the PC register will get this address, the wrong one and start its execution*

        - This is happening because C compiler does bound checking
        - Stack is used to handle interrupt, function call, when we call a function, there will be a "function stack frame" fro storing local variables in that function. and there is a return address reerence pointer. Base pointer is used to refer stuff in this frame. *There is a concrete definition on slide.*
        - 所以bp的作用是，让程序在运行的时候，能回到上一个function的stack frame，因为经常要处理handler之类的东西，当一个正在执行的函数被打断的时候，执行下一个函数，下一个函数执行完之后，往往需要回到之前执行的函数，那么为了回到之前执行的函数，就必须得有个pointer指向之前执行的函数，这个pointer就是bp，每个function call的function stack frame中都有一个bp，而bp里面的值存的是这个function call之前的function的stack frame的base address(old frame pointer)
    - Overflow on the heap
        - Consists of overwriting a command on the heap to execute an arbitary malicious command
        - *Heap* is a memory segment allocated by the OS to each program in order to perform dynamic allocation during runtime
- **Integer Overflow**:
    - Occurs when the variable type int is used instead of unsigned_int. It allows an attacker to use values that are not allowed but still getting access to some resources
- *Shellcode*: a string that express a binary program that starts a shell interpreter. It's given as an input to a program to be copied into a buffer in such a way it overwrites what is beyond the buffer bound, in particular thet return address from a function call.
- *payload*: 指的是整个的暴露vulunerable的程序
    - shellcode in payload
- Mechanism for preventing buffer overflow:
    - stack samshing prevention
        - 就是在stack中放入检查的东西，每一次都会先检这个东西有没有被覆盖，如果他被覆盖了说明有buffer overflow要发生 -> ABORT
    - disallow execution of code in a stack section of memory
    - *detect a sequence of NOP*
    - detect sequence such as /bin/bash or /bin/sh