OS Review

Deadlocks

Deadlock problem

- **Deadlock:** a set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set
- Examples:
  - a system has 2 disk drives, P1 and P2 each hold one disk drive and each needs another one
  - semaphores A and B, initialized to 1

P_1	P_2
wait (A);	wait(B)
wait (B);	wait(A)

System model

- Resources: $R_1, R_2, \ldots, R_m$
  - each represents a different **resource type**
    - e.g., CPU cycles, memory space, I/O devices
  - each resource type R_i has W_i **instances**.
- Each process utilizes a resource in the following pattern
  - request
  - use
  - release

Four Conditions of Deadlock

- **Mutual exclusion**: only one process at a time can use a resource
- **Hold and wait**: a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption**: a resource can be released only **voluntarily** by the process holding it, after it has completed its task
- **Circular wait**: there exists a set of waiting processes $\{P_0, P_1, \ldots, P_n\}$
  - P_0 is waiting for a resource that is held by P_1
  - P_1 is waiting for a resource that is held by P_2 …
  - P_{n-1} is waiting for a resource that is held by P_n
  - P_n is waiting for a resource that is held by P_0

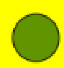
- Mutual exclusion（互斥）：一次只能有一个进程使用一个资源（信号量为1）；
- Hold and wait（等待）：持有至少一个资源的进程正在等待获取其他进程持有的其他资源；
- No preemption（非抢占）：我所获得的资源不能被剥夺，只能自己主动释放；
-

Resource-Allocation Graph

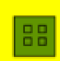
- Two types of nodes:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set of all the **processes** in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set of all **resource** types in the system
- Two types of edges:
  - **request edge**: directed edge $P_i \rightarrow R_j$
  - **assignment edge**: directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph

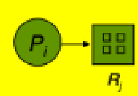
- Process



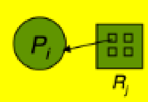
- Resource Type with 4 instances



- Pi requests instance of Rj

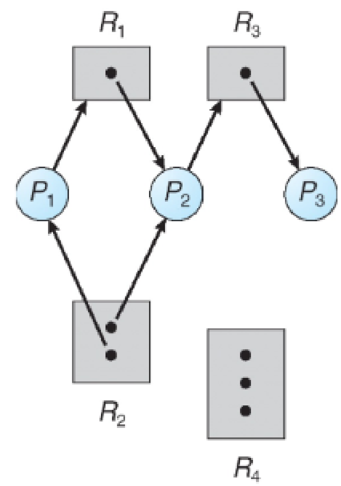


- Pi is holding an instance of Rj



Resource Allocation Graph

- One instance of R1
- Two instances of R2
- One instance of R3
- Three instance of R4
- P1 holds one instance of R2 and is waiting for an instance of R1
- P2 holds one instance of R1, one instance of R2, and is waiting for an instance of R3
- P3 is holds one instance of R3



> 节点和边代表什么关系

Basic Facts

- If graph contains **no cycles** ➡ **no deadlock**
- If graph contains a cycle
  - if only **one instance per resource type**, ➡ **deadlock**
  - if **several instances** per resource type ➡ **possibility** of deadlock

Handling deadlocks

How to Handle Deadlocks

- Ensure that the system will never enter a deadlock state
  - **Prevention**
  - **Avoidance**
- Allow the system to enter a deadlock state and then recover - database
  - **Deadlock detection and recovery**:
- **Ignore the problem** and pretend deadlocks never occur in the system

deadlock prevention

Deadlock Prevention

- How to prevent **mutual exclusion**
  - not required for sharable resources
  - must hold for non-sharable resources
- How to prevent **hold and wait**
  - whenever a process requests a resource, it doesn't hold any other resources
    - require process to request ***all*** its resources before it begins execution
    - allow process to request resources only when the process has none
      - release all resources that are held before it can request any additional resource
  - low resource utilization; starvation possible

Deadlock Prevention

- How to handle **no preemption**
  - if a process requests a resource not available
    - release all resources currently being held
    - preempted resources are added to the list of resources it waits for
    - process will be restarted only when it can get all waiting resources
- How to handle **circular wait**
  - **impose a total ordering of all resource types**
  - require that each process requests resources in an increasing order
  - **Many operating systems adopt this strategy for some locks.**

deadlock avoidance

> avoidance 中、要求 how resources are be requestsed

Deadlock Avoidance

- Dead avoidance: require **extra information** about how resources are to be requested
  - **Is this requirement practical**?
- Each process declares a **max** number of resources it may need
- Deadlock-avoidance algorithm ensure there can **never** be a **circular-wait** condition
- Resource-allocation state:
  - the number of **available** and **allocated** resources
  - the **maximum demands** of the processes

Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a **safe state**:
  - there exists a **sequence <P_1, P_2, ..., P_n>** of all processes in the system
  - for each P_i, resources that P_i can still request can be satisfied by currently **available resources + resources held by all the P_j, with j < i**

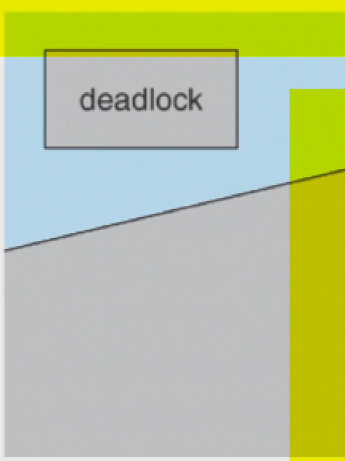
找出这样的序列就能说明 safety

当前 available 的资源加上前面所有 P hold 的资源足够当前 P 使用，称 Safe state

- **Safe state can guarantee no deadlock**
  - if P_i's resource needs are not immediately available:
    - wait until all P_j have finished (j < i)
    - when P_j (j < i) has finished, P_i can obtain needed resources,
  - when P_i terminates, P_{i+1} can obtain its needed resources, and so on

Basic Facts

- If a system is in **safe state ➡ no deadlocks**
- If a system is in **unsafe state ➡ possibility of deadlock**
- **Deadlock avoidance** ➡ ensure a system **never enters an unsafe state**



Example

Current Hold 当前持有，所以 available = 12 - 5 - 2- 2 = 3

- Resources: 12

	Maximum Needs	Current Needs	Available	Extra need
T_0	10	5	3	5
T_1	4	2		2
T_2	9	2		7

- Safe sequences: T1 T0 T2
  - T1 gets and return (5 in total), T0 gets all and returns (10 in total) and then T2
- What if we allocate 1 more for T2?

在多 instance，避免进入死锁状态，具体细节不需要理解，但要会一下题

Deadlock Avoidance Algorithms

- Single instance of each resource type ➡ use **resource-allocation graph**
- Multiple instances of a resource type ➡ use the **banker's algorithm**

Banker's Algorithm

- Banker's algorithm is for **multiple-instance resource deadlock avoidance**
  - each process must a **priori** claim **maximum** use of each resource type
  - when a process requests a resource it may have to wait
  - when a process gets all its resources it must release them in a finite amount of time

Banker's Algorithm: Example

- System state:
  - **5 processes** P_0 through P_4
  - **3 resource types**: A (10 instances), B (5instances), and C (7 instances)
- Snapshot at time T_0:

	allocation	max	available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Data Structures for the Banker's Algorithm

- **n** processes, **m** types of resources
  - **available**: an array of length ***m***, instances of available resource
    - available[j] = k: k instances of resource type R_j available
  - **max**: a ***n x m*** matrix
    - max [i,j] = k: process P_i may request at most k instances of resource R_j
  - **allocation**: ***n x m*** matrix 已经分配的
    - allocation[i,j] = k: P_i is currently allocated k instances of R_j
  - **need**: ***n x m*** matrix 现在还需要多少， max - allocation
    - need[i,j] = k: P_i may need k more instances of R_j to complete its task
    - **need [i,j] = max[i,j] – allocation [i,j]**

Banker's Algorithm: Example

- **need = max – allocation**

	need	available
	A B C	A B C
P_0	7 4 3	3 3 2
P_1	1 2 2	
P_2	6 0 0	
P_3	0 1 1	
P_4	4 3 1	

- The system is in a safe state since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria

> 找到一个序列，是 safe 的

Banker's Algorithm: Example

- Why < P_1, P_3, P_4, P_2, P_0> is in safe state?

	allocation	max	available.	Needed
	A B C	A B C	A B C	
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

1) Finish[1] = true, needed[1] < work -> work = work + allocation = [5 3 2]
2) Finish[3] = true, needed[3]< work -> work = work + allocation = [7 4 3]
3) finish[4] = true, needed[4] < work -> work = work + allocation = [7 4 5]
4) finish[2] = true, needed[2] < work -> work = work + allocation = [10 4 7]
5) Finish[0] = true, needed[0] < work -> work = work + allocation = [10 5 7]

deadlock detection

Deadlock recovery

Deadlock Recovery: Option I

avoidance 是在资源即将分配的时候检查，recovery 是周期性检查系统是否 safe

- Terminate deadlocked processes. options:
  - abort all deadlocked processes
  - abort one process at a time until the deadlock cycle is eliminated
  - In which order should we choose to abort?
    - priority of the process
    - how long process has computed, and how much longer to completion
    - resources the process has used
    - resources process needs to complete
    - how many processes will need to be terminated
    - is process interactive or batch?

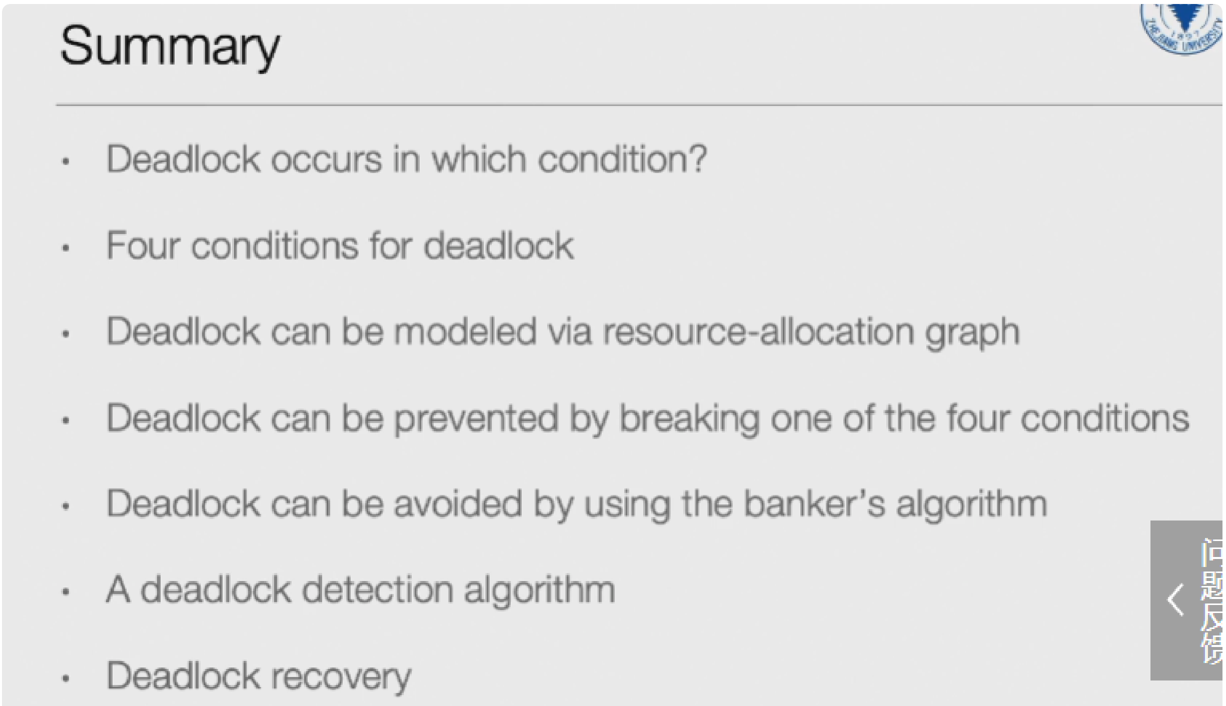
kil所有的死锁进程；

每次kil一个进程，知道不存在死锁现象；

Deadlock Recovery: Option II

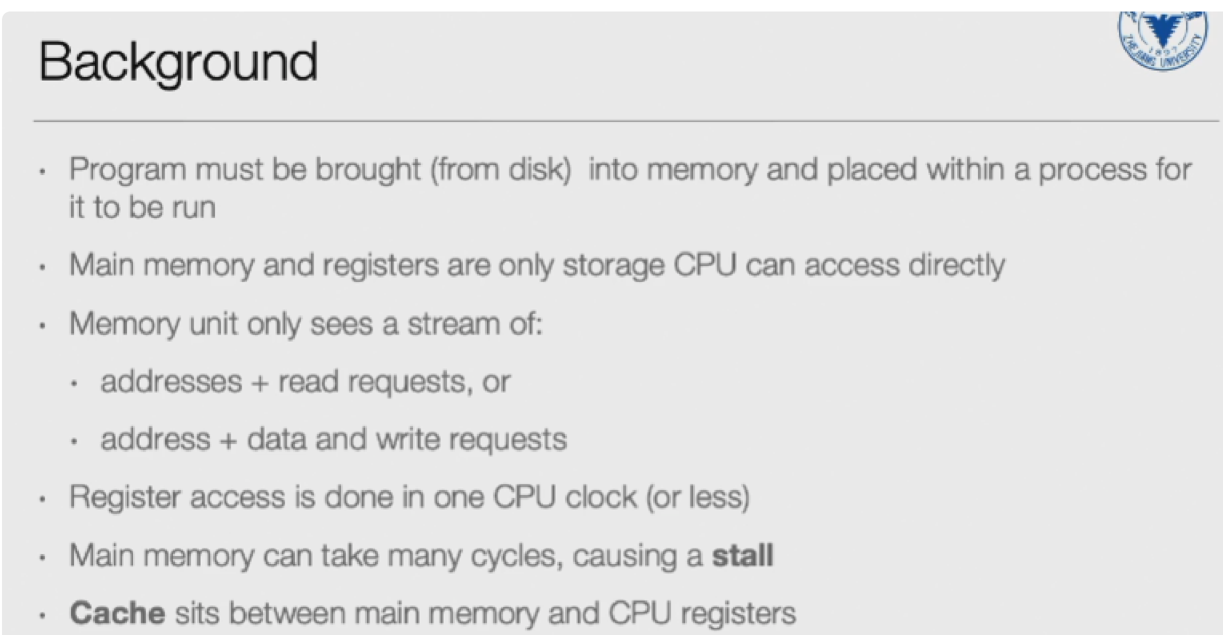
- Resource preemption
  - Select a victim
  - Rollback
  - Starvation
    - How could you ensure that the resources do not preempt from the same process?

- 不kill进程，只是强制回收资源；



> 死锁这一部分的核心是知道死锁的四个条件，处理死锁的几个策略（prevention. avoidance, detection, ignore problem）,一些 fact 资源分配图是不是有环，和死锁什么关系 ；safe/unsafe stage 和死锁的关系；银行家算法会判断是否在 safe stage

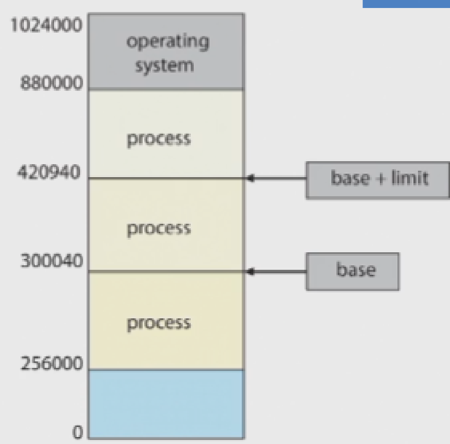
*虚拟内存



> 保护不同的进程间memory 是互不干扰的，base+limit 其实就是分段思想

Protection

- Need to censure that a process can access only access those addresses in it address space.
- We can provide this protection by using a pair of **base** and **limit** registers define the logical address space of a process



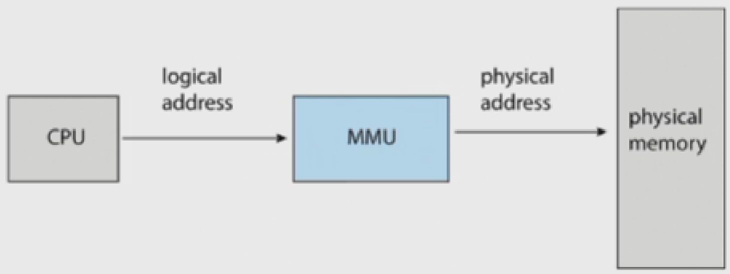
Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a **separate physical address** space is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as virtual address
  - **Physical address** – address seen by the memory unit
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme
- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses generated by a program

> 逻辑地址到物理地址的转换由 MMU 完成

Memory-Management Unit (MMU)

- Hardware device that at run time maps virtual to physical address



- Many methods possible, covered in the rest of this chapter

Memory-Management Unit

- Consider simple scheme. which is a generalization of the base-register scheme.
- The base register now called **relocation register**
- The value in the relocation register is added to every address generated by a user process at the time it is sent to memory
- The user program deals with logical addresses; it never sees the real physical addresses
  - Execution-time binding occurs when reference is made to location in memory
  - Logical address bound to physical addresses

Dynamic Linking

- **Static linking** – system libraries and program code combined by the loader into the binary program image
- Dynamic linking –linking postponed until **execution time**
  - Small piece of code, **stub**, used to locate the appropriate memory-resident library routine
  - Stub replaces itself with the address of the routine, and executes the routine
- Dynamic linking is particularly useful for libraries
  - System also known as **shared libraries**
  - Consider applicability to patching system libraries
    - Versioning may be needed
    - What will happen without dynamic linking?
- Help from OS: share libraries between processes

Contiguous Allocation

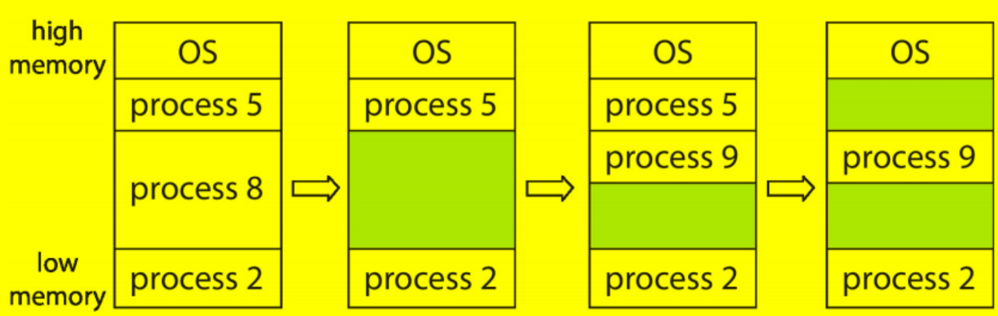
- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method
- Main memory usually into two **partitions**:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
  - Each process contained in single contiguous section of memory

把main memory分成两部分，一部分给os用，另一部分给用户态的进程用；

Protection：因为是连续的，所以通过base和limit两个register就可以进行检测；

First-fit：首次匹配（first fit）策略就是找到第一个足够大的块，将请求的空间返回给用户。同样，剩余的空闲空间留给后续请求。 Best-fit：首先遍历整个空闲列表，找到和请求大小一样或更大的空闲块，然后返回这组候选者中最小的一块。 Worst-fit：最差匹配（worst fit）方法与最优匹配相反，它尝试找最大的空闲块，分割并满足用户需求后，将剩余的块（很大）加入空闲列表。

弊端：内存逐渐碎片化（外部碎片）

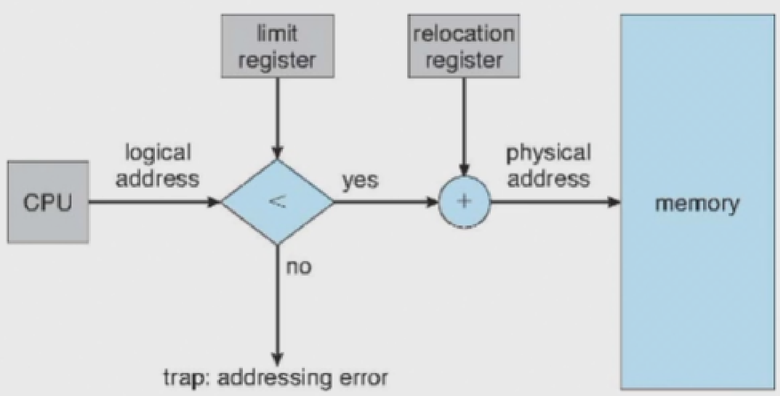


Contiguous Allocation: protection

- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Base register contains value of smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address dynamically
  - Can then allow actions such as kernel code being **transient** and kernel changing size

> 在连续分配的策略下如何用 MMU 做 protection

Hardware Support for Relocation and Limit Registers



> 连续分配带来的问题：外部碎片 & 内部碎片

Fragmentation

- **External fragmentation**
  - unusable memory between allocated memory blocks
    - **total amount** of free memory space **is larger than a request**
    - the request cannot be fulfilled because the free memory is not **contiguous**
  - external fragmentation can be reduced by **compaction**
    - shuffle memory contents to place all free memory in one large block
    - program needs to be **relocatable** at runtime
    - Performance overhead, timing to do this operation
  - Another solution: **paging**
  - 50-percent rule: N allocated blocks, 0.5N will be lost due to fragmentation. 1/3 is unusable!

> 外部碎片解决方法 —— 压缩 compaction（但不实用），第二种方法是 Paging

- **Internal fragmentation**
  - memory allocated may be larger than the requested size
  - this size difference is memory *internal to a partition*, but not being used
  - Example: free space 18464 bytes, request 18462 bytes
- Sophisticated algorithms are designed to avoid fragmentation
  - none of the first-/best-/worst-fit can be considered sophisticated

Paging

一个进程的物理地址空间可以不连续，可以分成很多虚拟页映射到很多物理页，从而实现不连续的地址空间。

问题：page也有一个最小的粒度(比如4KB)，仍然存在内部碎片的问题

- 内部碎片：如果分配程序给出的内存块超出请求的大小，在这种块中超出请求的空间（因此而未使用）就被认为是内部碎片（因为浪费发生在已分配单元的内部）。

方法：

- 把物理内存切分成固定大小的block，称为frame，但是size必须是\(2^n\)，一般在512B到16MB之间；
- 把逻辑内存也分成同样大小的block，成为pages；
- 当需要N个page，就需要有N个frame与之对应（当然同时可能不需要N个frame）；
- 需要初始化一张页表（CPU内部没有这么大的空间去存页表，所以要把页表存储在内存中，而CPU有一个专门的寄存器去存储这个页表的物理地址）去把page对应到frame；

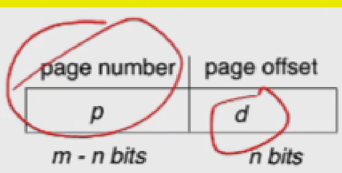
Paging

- Physical address space of a process can be **noncontiguous**; process is allocated physical memory whenever the latter is available
  - Avoids **external fragmentation -> avoid for compacting**
  - Avoids problem of varying sized memory chunks
- Basic methods
  - Divide physical memory into fixed-sized blocks called **frames**
    - Size is power of 2, between 512 bytes and 16 Mbytes
  - Divide logical memory into blocks of same size called **pages**
  - Keep track of all free frames
  - To run a program of size **N** pages, need to find **N** free frames and load program
  - Set up a **page table** to translate logical to physical addresses
  - Backing store likewise split into pages
  - Still have Internal fragmentation

> 最简单的一级分页

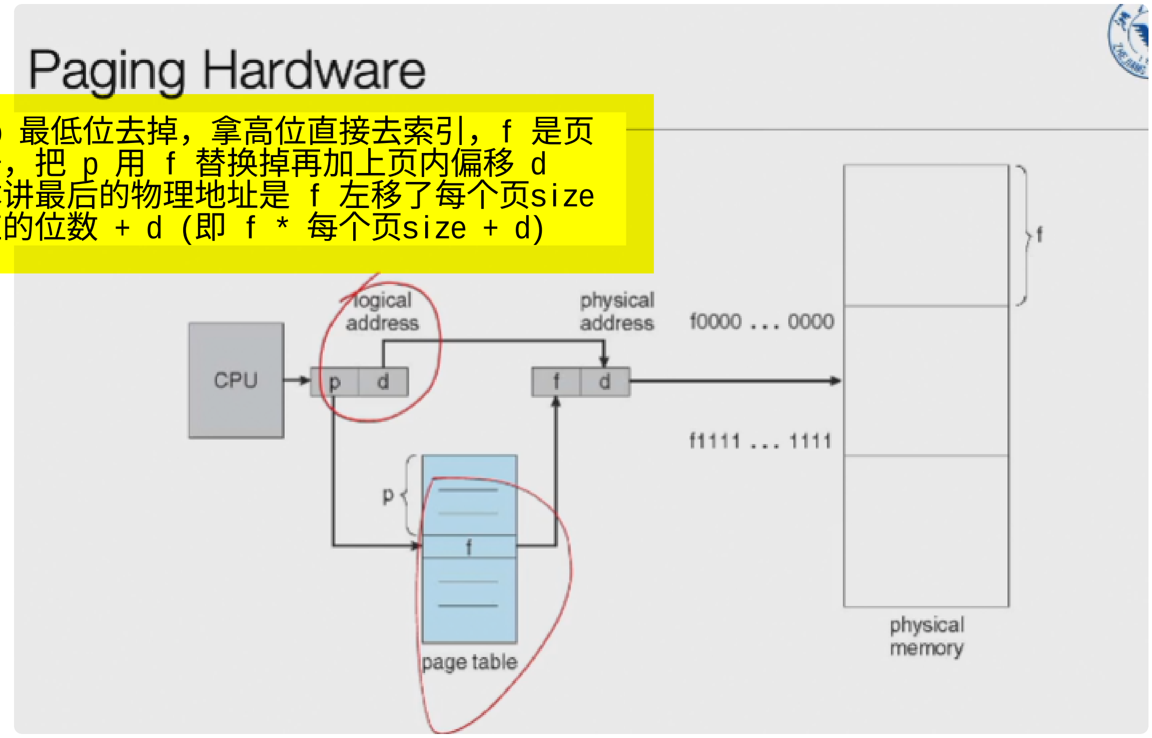
Paging: Address Translation

- A logical address is divided into:
  - **page number** (p)
    - used as an index into a page table
    - page table entry contains the corresponding **physical** frame number
  - **page offset** (d)
    - offset within the page/frame
    - combined with frame number to get the physical address

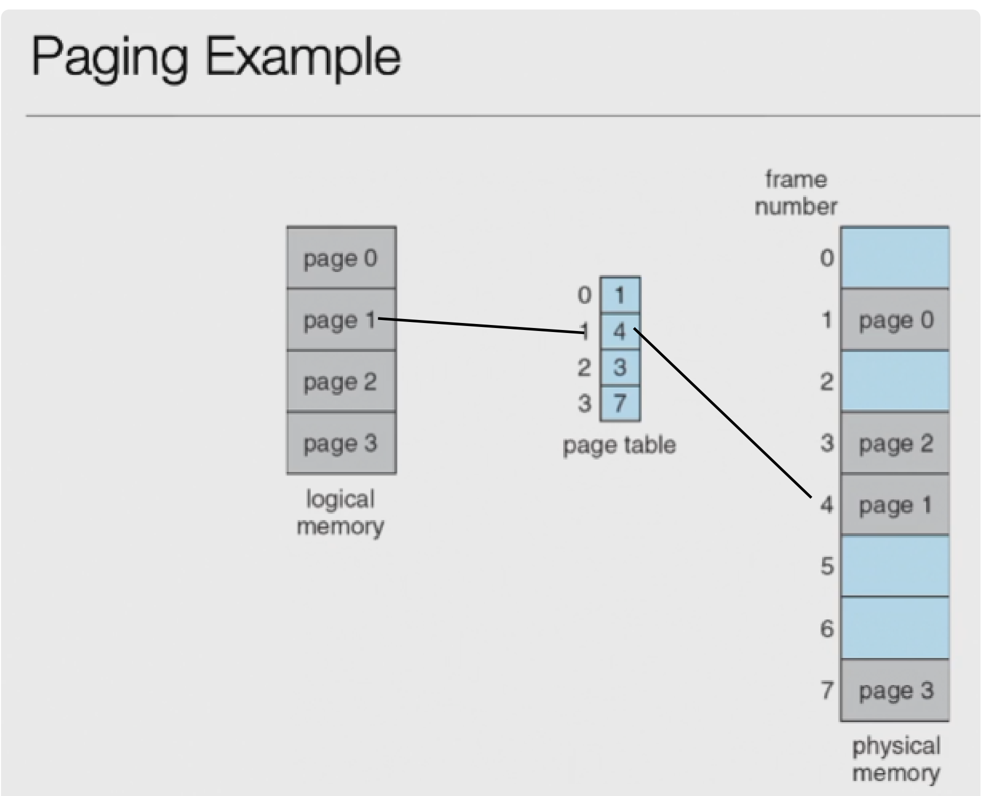


m bit logical address space, n bit page size

把 p 最低位去掉，拿高位直接去索引，f 是页框号，把 p 用 f 替换掉再加上页内偏移 d
具体讲最后的物理地址是 f 左移了每个页size对应的位数 + d (即 f * 每个页size + d)

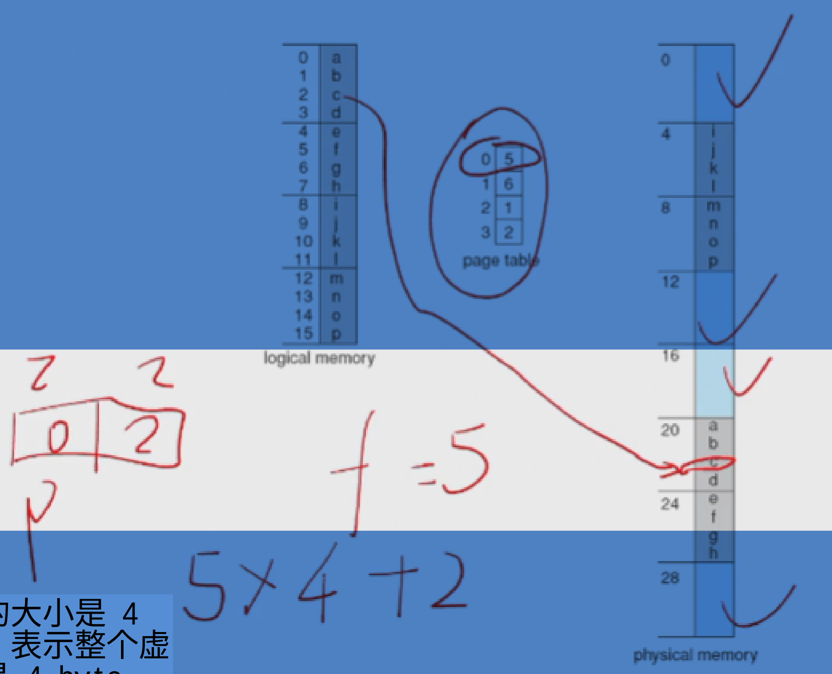


有时间听一下这个例子



虚拟地址为 2 的地方，页号是 0， 页框号为 5，物理地址应该为 5 * 4 + 2 = 22

Paging Example II



$m = 4$ and $n = 2$ 32-byte memory and 4-byte pages

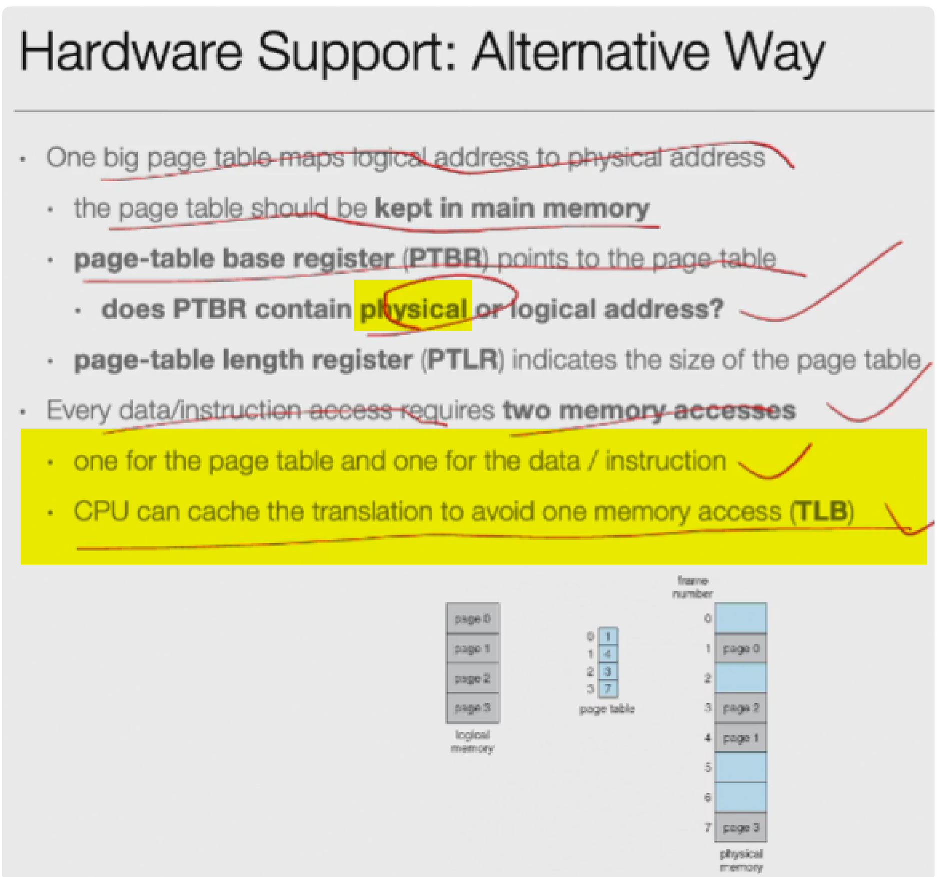
n = 2 表示页的大小是 4 个字节，m = 4 表示整个虚拟地址的大小是 4 byte
2 的 m - n 次方为 4 表示 4 个 Page

Paging: Internal Fragmentation

- Paging has **no external fragmentation**, but **internal fragmentation**
  - e.g., page size: 2,048, program size: 72,766 (35 pages + 1,086 bytes)
    - internal fragmentation: 2,048 - 1,086 = 962
  - **worst** case internal fragmentation: **1 frame – 1 byte**
  - **average** internal fragmentation: 1 / 2 frame size
- Small frame sizes more desirable than large frame size?
  - memory becomes larger, and page table takes memory
  - page sizes actually grow over time
    - 4KB ➔ 2MB ➔ 4MB ➔ 1GB ➔ 2GB

将表放到物理内存

- **寄存器**：page table放到一系列专门的寄存器里面，这样**查找**会非常快，但缺点是table的**size**会非常小（寄存器的资源有限）而且**上下文切换**的时候也要去save和restore寄存器；



页表的 cache

- **内存中的表格**：用一个寄存器（page-table base register (PTBR)）来存放当前进程页表的物理地址，然后去内存中去查找页表项，这样空间大小可以保证。但是这样就需要进行至少两次的**内存访问**；
- 通过TLB作为页表的cache（存储着几个虚拟地址到物理地址的映射）；

TLB

- TLB (**translation look-aside buffer**) caches the address translation
  - if page number is in the TLB, no need to access the page table
  - if page number is not in the TLB, need to replace one TLB entry
  - TLB usually use a fast-lookup hardware cache called **associative memory**
  - TLB is usually small, 64 to 1024 entries
- Use with page table
  - TLB contains a few page table entries
  - Check whether page number is in TLB
    - If -> frame number is available and used
    - If not -> **TLB miss**. access page table and then fetch into TLB
      - TLB flush: TLB entries are full
      - TLB wire down: TLB entries should not be flushed

TLB

- TLB and context switch
  - Each process has its own page table
    - switching process needs to switch page table
  - **TLB must be consistent with page table**
    - Option I: Flush TLB at every context switch, **or**,
    - Option II: Tag TLB entries with **address-space identifier** (ASID) that uniquely identifies a process
  - some TLB entries can be **shared** by processes, and fixed in the TLB
    - e.g., TLB entries for the kernel
- TLB and operating system
  - MIPS: OS should deal with TLB miss exception
  - X86: TLB miss is handled by hardware

> 会算 EAT

TLB：页表的cache，TLB命中一次就能减少一次内存访问的开销。

- 进程切换的时候，把TLB给flush掉来保持TLB和页表的相关性。但这样就会造成冷启动。
- 利用地址空间标识符ASID来标记该项地址属于哪个进程的地址空间，Tag TLB entries with **address-space identifier** (ASID) that uniquely identifies a process。
- 有些映射可以固定，some TLB entries can be **shared** by processes, and fixed in the TLB。

Effective Access Time

页表起到 memory protection 的作用

- **Hit ratio** – percentage of times that a page number is found in the TLB
- An 80% hit ratio means that we find the desired page number in the TLB 80% of the time.
- Suppose that 10 nanoseconds to access memory.
  - If we find the desired page in TLB then a mapped-memory access take 10 ns
  - Otherwise we need **two memory access** so it is 20 ns: page table + memory access
- Effective Access Time (EAT)
- EAT = 0.80 x 10 + 0.20 x 20 = 12 nanoseconds
- implying 20% slowdown in access time
- Consider a more realistic hit ratio of 99%,
- EAT = 0.99 x 10 + 0.01 x 20 = 10.1ns
- implying only 1% slowdown in access time.

就算有了TLB，分页后还是会变慢

Memory Protection

- Accomplished by protection bits with each frame
- Each page table entry has a **present** (aka. valid) bit
  - present: the page has a valid physical frame, thus can be accessed
- Each page table entry contains some protection bits
  - **kernel/user**, **read/write**, **execution**?, **kernel-execution**?
  - why do we need them?
- Any violations of memory protection result in a trap to the kernel

比如多个进程共享libc的一段代码，这就需要进行内存的共享，所以libc的代码在物理内存中仅放一份，然后把各个进程的虚拟地址都指向同一个物理地址，这样就实现了共享

Page Sharing

不同 VA 指向相同 PA

- Paging allows to share memory between processes
  - e.g., one copy of **code** shared by **all processes of the same program**
    - text editors, compilers, browser..
  - shared memory can be used for **inter-process communication**
  - shared libraries
- Reentrant code: non-self-modifying code: never changes between execution
- Each process can, of course, have its private code and data

Structure of Page Table

- One-level page table can consume lots of memory for page table
  - e.g., 32-bit logical address space and 4KB page size
    12 位
    - page table would have 1 million entries ($2^{32} / 2^{12}$)
    - if each entry is 4 bytes ➔ 4 MB of memory for page table alone
  - each process requires its own page table
  - page table must be **physically contiguous**
- To reduce memory consumption of page tables:
  - **hierarchical page table**
  - **hashed page table**
  - **inverted page table**

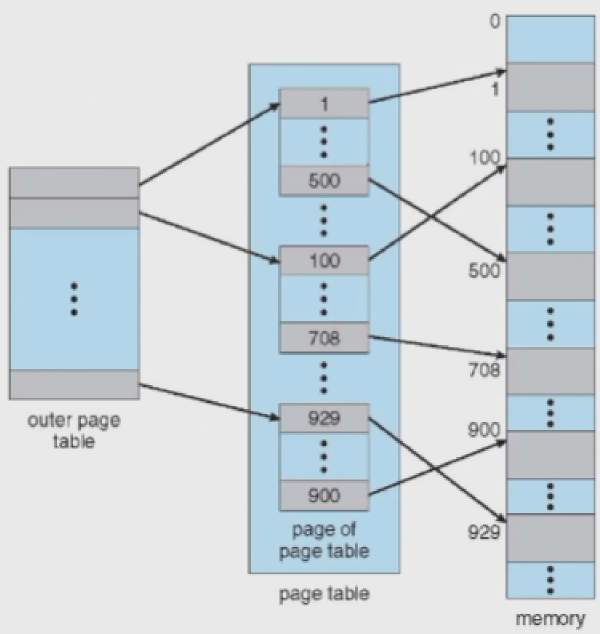
> 要知道三种页表的含义，优缺点，组织方式。重点理解 Hierarchical
> 给一个 VA 需要清楚给出一个 VA –> PA 的转换，第一次用什么查表，第二次怎么查表，最后查出来的页框号怎么和页内偏移拼接成物理地址

Hierarchical Page Tables

- Break up the logical address space into **multiple-level** of page tables
  - e.g., two-level page table
  - first-level page table contains the frame# for second-level page tables
    - “page” the page table
- Why hierarchical page table can save memory for page table?

把页表分为多级 --- 为了减少页表本身所需要的内存空间

Two-Level Page Table



Two-Level Paging

- A logical address is divided into:
  - a **page directory number** (first level page table)
  - a **page table number** (2nd level page table)
  - a **page offset**
- Example: 2-level paging in 32-bit Intel CPUs
  - 32-bit address space, 4KB page size
  - 10-bit page directory number, 10-bit page table number
  - each page table entry is 4 bytes, one frame contains 1024 entries (2^{10})

页表的空间是 4K（第一层空间） + N（第二层合法映射的二级页表个数） * 4K（二级页表空间） （N << 2^10）

$2^{10} \times 4 = 4K$

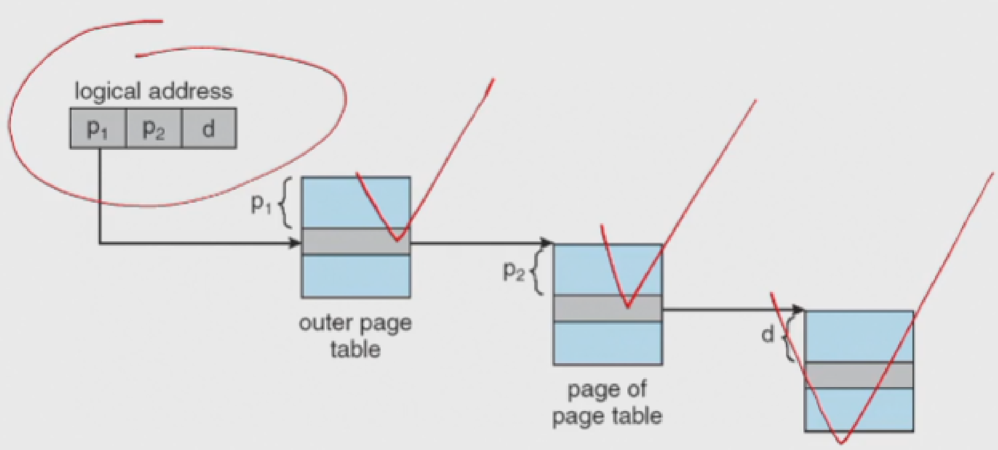
4K + N * 4'

p_1	p_2	d
10	10	12

缺点是使得访问页表这件事情变慢

大部分虚拟内存根本不是合法地址，不需要建立映射，从而减少开销

Address-Translation Scheme



64-bit Logical Address Space

- 64-bit logical address space requires more levels of paging
  - two-level paging is not sufficient for 64-bit logical address space
    - if page size is 4 KB (2^{12}), outer page table has 2^{42} entries, inner page tables have 2^{10} 4-byte entries
  - one solution is to add more levels of page tables
    - e.g., three levels of paging: 1st level page table is 2^{34} bytes in size
    - and possibly 4 memory accesses to get to one physical memory location
  - usually **not support full 64-bit virtual address space**
    - AMD-64 supports 48-bit
    - canonical form: 48 through 63 of valid virtual address must be copies of bit 47

64 位的分页机制中，一个 page_table_entry 的大小是 8 字节

64-bit Logical Address Space

4K 8 512 9

outer page	inner page	offset
p_1	p_2	d
42	10	12

2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

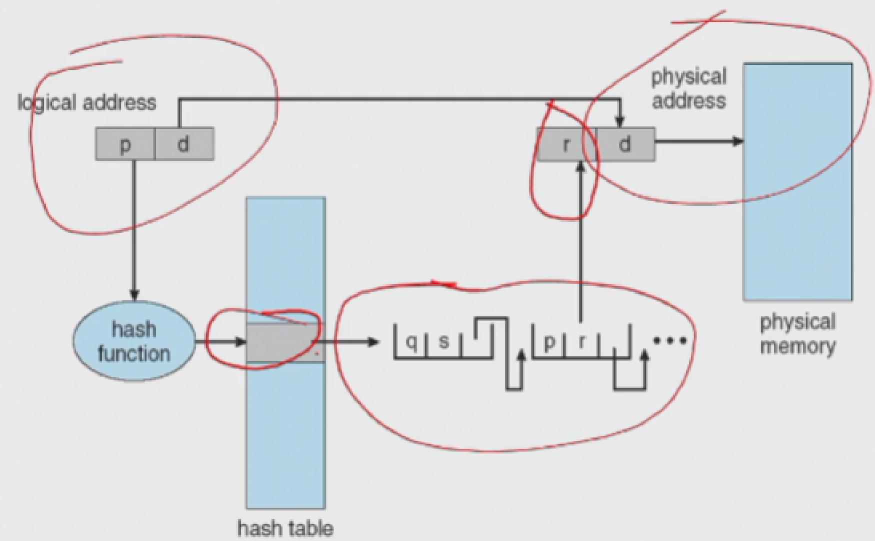
缺点：页表变成了多层，就会导致访问内存的次数又会增加，虽然可以通过TLB来减少，但如果TLBmiss了，那么开销会急剧增加。

通过哈希碰撞来减少内存的占用，hash table中存着一个链表，记录着映射到相同位置的地址表项，要去遍历链表来查表项。

Hashed Page Tables

- In hashed page table, virtual page# is hashed into a frame#
  - the page table contains a chain of elements hashing to the same location
  - each element contains: **page#**, **frame#**, and a **pointer to the next element**
    - virtual page numbers are compared in this chain searching for a match
    - if a match is found, the corresponding frame# is returned
- Hashed page table is common in address spaces > 32 bits
- **Clustered page tables**
  - Each entry refers to several pages

Hashed Page Table



可以把page table中的几个表项进行合并，一个表项可以指向很多的物理页，增大了表项（或者说是页的大小）的长度，减少了表项的个数，从而减少了碰撞次数。

既然虚拟地址空间通常比物理地址空间大很多，所以反过来，通过物理页框号去寻找虚拟内存的页号（这样多个进程都共用了转置页表，而不是一个进程一张页表），要去遍历搜索表项（这是不切实际的）。

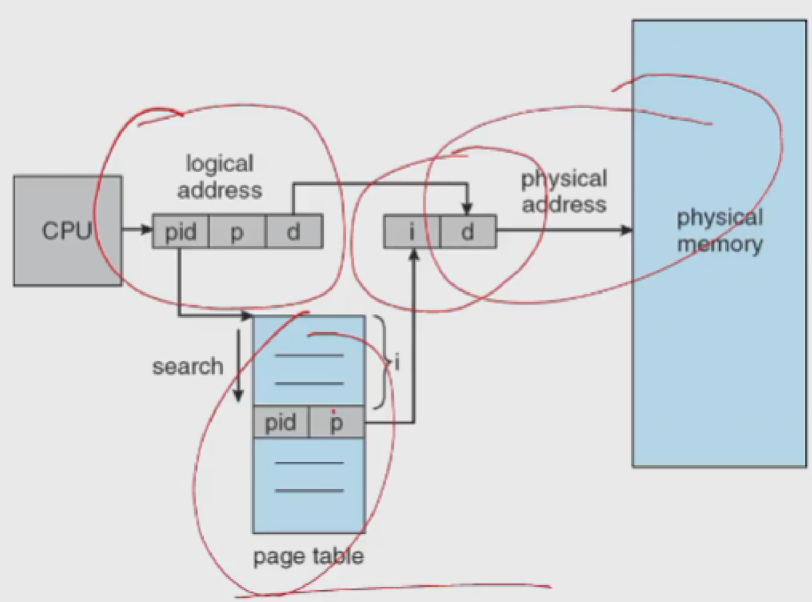
转置页表的内容：

- 页号
- 进程ID
- 控制位——包括valid位，dirty位，reference位，protection位和locking位。
- 链接指针——如果出现进程共享内存的情况，就会用到链接指针。

Inverted Page Table

- Inverted page table tracks allocation of physical frame to a process
  - one entry for each physical frame ➔ fixed amount of memory for page table
  - each entry has the **process id** and the **page#** (virtual address)
- Sounds like a brilliant idea?
  - to translate a virtual address, it is necessary to search the (**whole**) page table
    - can use TLB to accelerate access, TLB miss could be very expensive
  - how to implement shared memory?
    - a physical frame can only be mapped into one process!
    - Because one physical memory page cannot have multiple virtual page entry!

Inverted Page Table



一个进程可以被临时地存放到disk中，然后再从disk中回到memory。程序里面访问的是虚拟地址，而我们只需要把page table的映射关系改一下，所以swap前后物理地址不需要相同。

因为今天物理内存很大，所以平时swap的场景比较少。而在Mobile System里面，没有swap这件事情，因为他空间更小，CPU的吞吐量更小，如果没有空间了直接kill进程。

我们可以swap页，而不是swap整个进程，这样粒度就变小了，可以减少文件传输量，从而减少性能上的开销。

比如进程A的某个page被swap out到disk中，进程B的某些page被swap in进来，当B结束后，又要运行进程A，首先把page out的load进来，再进行运行，运行一段时间后又开始运行进程B，然后又发现进程B的某些page也要去load进来。所以在某些极端的情况下，CPU在不停的swap，而要执行的任务没有什么实际的进展，这种叫做操作系统的颠簸（抖动），主要的原因是系统在运行的进程数量很多，但是内存又很紧张。

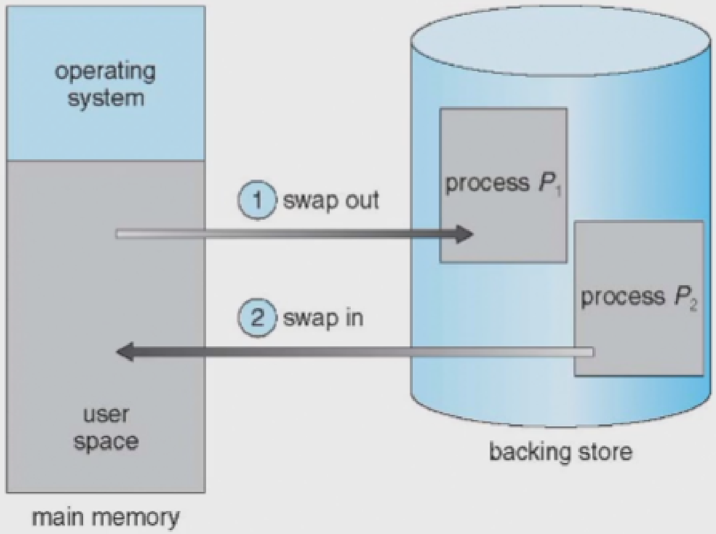
Swapping

- **Swapping** extends physical memory with backing disks
  - a process can be swapped temporarily out of memory to a backing store
    - backing store is usually a (fast) disk
  - the process will be brought back into memory for continued execution
    - does the process need to be swapped back in to same physical address?
- Swapping is usually only initiated under memory pressure
- Context switch time can become very high due to swapping
  - if the next process to be run is not in memory, need to swap it in
  - disk I/O has high latency

Context Switch Time including Swapping

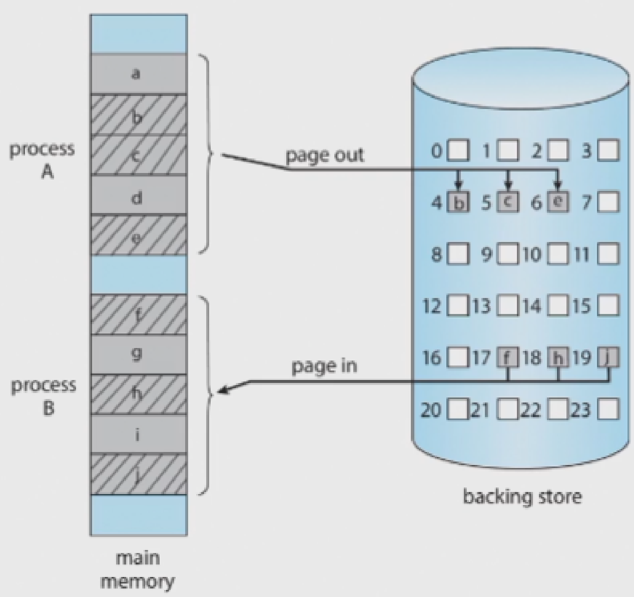
- If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
- Context switch time can then be **very high**
- 100MB process swapping to hard disk with transfer rate of 50MB/sec
  - Swap out time of 2,000 **ms**
  - Plus swap in of same sized process
  - Total context switch swapping component time of 4000ms (4 seconds)
- Can reduce if reduce size of memory swapped – by knowing how much memory really being used

Swapping



Swapping with Paging

- Swap pages instead of entire process



> Swap 通常是在内存压力较大时做

> 理解 32， 64 位系统如何做分页

在 RISC-V 中做分页

SV39

- Effective address : 64 bit
- [39:63] == bit 38
- vpn: virtual page number

页框号 44 位 + 12 位 偏移 == 56 位物理地址

```
|38      30|29      21|20      12|11            0|
+----------+----------+----------+---------------+
| VPN[2]   | VPN[1]   | VPN[0]   | page offset   |
+----------+----------+----------+---------------+
```

```
|63      54|53      28|27      19|18      10|9   8|7|6|5|4|3|2|1|0|
+----------+----------+----------+----------+-----+-+-+-+-+-+-+-+-+
| reserve  | PPN[2]   | PPN[1]   | PPN[0]   | RSW |D|A|G|U|X|W|R|V|
+----------+----------+----------+----------+-----+-+-+-+-+-+-+-+-+
```

44 + 12

Sv39

每个page的大小是\(2^{12}\)字节

每个page table entry的大小是8字节，用9位去索引这8个字节

页表项存的是下一级页表的物理地址

每个 9 位可以索引的是 512 个entry，每个 entry 8 字节，总共还是 4K，保证了一个页 --- 每一级页表项用一个页记录所有 entry 刚好

- Effective address : 64 bit
- [39:63] == bit 38
- vpn: virtual page number

```
|38      30|29      21|20      12|11            0|
+----------+----------+----------+---------------+
| VPN[2]   | VPN[1]   | VPN[0]   | page offset   |
+----------+----------+----------+---------------+
```

```
|63      54|53      28|27      19|18      10|9   8|7|6|5|4|3|2|1|0|
+----------+----------+----------+----------+-----+-+-+-+-+-+-+-+-+
| reserve  | PPN[2]   | PPN[1]   | PPN[0]   | RSW |D|A|G|U|X|W|R|V|
+----------+----------+----------+----------+-----+-+-+-+-+-+-+-+-+
```

An Example

- VA: 0xffff ffc0 1357 9bdf
- PA: 0x9377 9bdf
- The physical address of page table is 0x8010 0000
- Question: How to setup the satp and page tables?

1. satp 设定

SATP

- Mode: 8 (sv39)
- Asid = 0
- PPN = 0x8010 0000/4K = 0x80100 >> 12

```
|63  60|59  44|43                 0|
+------+------+--------------------+
| mode | ASID | physical page number |
+------+------+--------------------+
```

First Level Page Table

- Page size: 4K, page table entry: 8 bytes -> 512 entry
- 0xffff ffc0 1357 9bdf
- 1**100 0000 00**01 [38:30]
- b'1 0000 0000 x 8 bytes = 0x800 每一个 entry 8 byte 所以还要乘8得到偏移量
- So the physical address of first level page table is 0x8010 0000 + 0x800 = 0x8010 0800

First Level Page Table

- So the physical address of first level page table is 0x80100000 + 0x800 = 0x8010 0800
- Suppose its value is 0x00000000_20040401

开始解码

```
|63      54|53     28|27     19|18     10|9    8|7|6|5|4|3|2|1|0|
+----------+---------+---------+---------+------+-+-+-+-+-+-+-+-+
| reserve  | PPN[2]  | PPN[1]  | PPN[0]  | RSW  |D|A|G|U|X|W|R|V|
+----------+---------+---------+---------+------+-+-+-+-+-+-+-+-+
```

First Level Page Table

- Suppose its value is 0x00000000_20040401

```
|63      54|53     28|27     19|18     10|9    8|7|6|5|4|3|2|1|0|
+----------+---------+---------+---------+------+-+-+-+-+-+-+-+-+
| reserve  | PPN[2]  | PPN[1]  | PPN[0]  | RSW  |D|A|G|U|X|W|R|V|
+----------+---------+---------+---------+------+-+-+-+-+-+-+-+-+
```

- V = 1, R =0, W = 0, X = 0 (rwx = 0??)
- PPN [53:10] = 0x80101

riscv 特殊设计，在 page table entry 中当 RWX 都是 0 的时候，表示其指向的是下一级的页表，不是物理页

PPN << 12 表示第二级页表的起始地址，得到 second-page table address

Second Level Page Table

- VPN[1] (bit 29 : bit 21) -> b'0 1001 1010 (0x96)
- The second-page table address is 0x8010 1000 + 0x96x8 = 0x801014b0
- Suppose its value is 0x00000000_20040801
  - Third-level page table address: 200408 >> 2 << 12 = 0x8010 2000

Third Level Page Table

- VPN[0] (bit 20 : bit 12) -> b'1 0111 1001 (0x179)
- The page table entry address is 0x80102000 + 0x179x8 = 0x80102bc8
- Suppose its value is 0x00000000_24dde40f

```
|63      54|53     28|27     19|18     10|9    8|7|6|5|4|3|2|1|0|
+----------+---------+---------+---------+------+-+-+-+-+-+-+-+-+
| reserve  | PPN[2]  | PPN[1]  | PPN[0]  | RSW  |D|A|G|U|X|W|R|V|
+----------+---------+---------+---------+------+-+-+-+-+-+-+-+-+
```

- V =1 RWX = 111
- PPN = 0x24dde4 >> 2 = 0x93779

PPN -> 物理页框号

- -> pa = PPN << 12 + page offset = 0x93779bdf

- VA: 0xffff ffc0 1357 9bdf
- PA: 0x9377 9bdf

得到页内偏移 bdf，最低 12 位

Background

核心 --- demand paging

- Code needs to be in memory to execute, but entire program **rarely** needed or used at the same time
  - error handling code, unusual routines, large data structures
- Consider ability to execute **partially-loaded program**
  - program no longer constrained by limits of physical memory
  - programs could be larger than physical memory

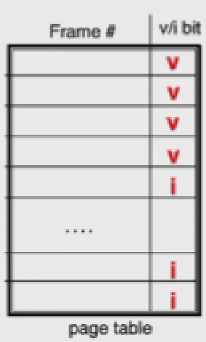
- Virtual memory: separation of **logical memory** from **physical memory**
  - **only part of the program needs to be in memory for execution**
    - logical address space can be much larger than physical address space
    - more programs can run concurrently
    - less I/O needed to load or swap processes (part of it)
  - allows memory (e.g., shared library) to be shared by several processes: better IPC performance
  - allows for more efficient process forking (**copy-on-write**)
- Virtual memory can be implemented via:
  - **demand paging**

Demand Paging

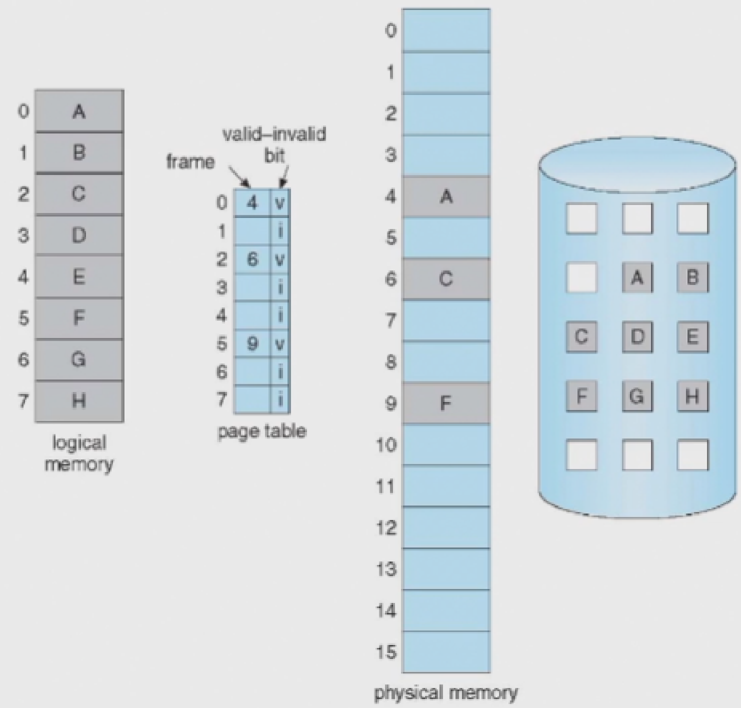
- **Demand paging** brings a page into memory **only when it is accessed**
  - if page is invalid ➡ abort the operation
  - if page is valid but not in memory ➡ bring it to memory via swapping
  - no unnecessary I/O, less memory needed, faster response, more apps
- **Lazy swapper**: never **swaps a page in** memory unless it will be needed
  - the swapper that deals with pages is also caller a pager
- **Pre-Paging**: pre-page all or some of pages a process will need, before they are referenced
  - it can reduce the number of page faults during execution
  - if pre-paged pages **are unused**, I/O and memory was wasted
    - although it reduces page faults, total I/O# likely is higher

Valid-Invalid Bit

- Each page table entry has a valid–invalid (present) bit
  - *V* ➡ in memory (memory is resident), *I* ➡ not-in-memory
  - initially, valid–invalid bit is set to *i* on all entries
  - during address translation, if the entry is invalid, it will trigger a **page fault**
- Example of a page table snapshot:



Page Table (Some Pages Are Not in Memory)



Page Fault

PC → LW

- First reference to a non-present page will trap to kernel: **page fault**
- Operating system looks at memory mapping to decide:
  - **invalid reference** ➡ deliver an exception to the process
  - **valid but not in memory** ➡ swap in
    - get an empty physical frame
    - swap page into frame via disk operation
    - set page table entry to indicate the page is now in memory
    - restart the instruction that caused the page fault

在操作系统中还要维护 Free–Frame List

Free-Frame List

- When a page fault occurs, the operating system must bring the desired page from secondary storage into main memory.
- Most operating systems maintain a free-frame list -- a pool of free frames for satisfying such requests.

head → 7 → 97 → 15 → 126 ··· → 75

- Operating system typically allocate free frames using a technique known as **zero-fill-on-demand** -- the content of the frames zeroed-out before being allocated.
- When a system starts up, all available memory is placed on the free-frame list.

理解 12 步

Stages in Demand Paging – Worse Case

- 1. Trap to the operating system
- 2. Save the user registers and process state
- 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:
  - 5.1 Wait in a queue for this device until the read request is serviced
  - 5.2 Wait for the device seek and/or latency time
  - 5.3 Begin the transfer of the page to a free frame

- 6. While waiting, allocate the CPU to some other user
- 7. Receive an interrupt from the disk I/O subsystem (I/O completed)
- 8. Save the registers and process state for the other user
- 9. Determine that the interrupt was from the disk
- 10. Correct the page table and other tables to show 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

Demand Paging: EAT

- Page fault rate: $0 \le p \le 1$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault
- Effective Access Time (EAT):

  $(1 - p)$ x memory access + p x (

  page fault overhead +

  swap page out + swap page in +

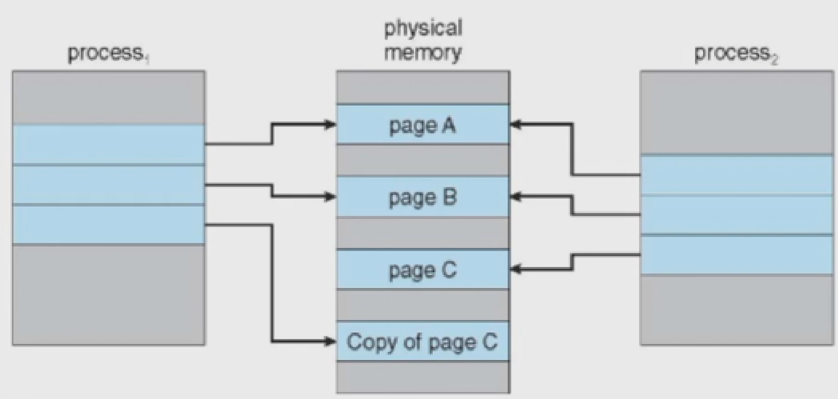
  instruction restart overhead)

> 理解 C–O–W 本质，细节，为什么这么做

Copy-on-Write

- **Copy-on-write** (COW) allows parent and child processes to initially share the same pages in memory
  - the page is shared as long as no process modifies it
  - if either process modifies a shared page, only then is the page copied
- COW allows more efficient **process creation**
  - no need to copy the parent memory during fork
  - only changed memory will be copied later
- vfork syscall optimizes the case that child calls **exec** immediately after fork
  - parent is suspend until child exits or calls exec
  - child shares the parent resource, including the heap and the stack
    - child cannot return from the function or call exit
  - vfork could be fragile, **it is invented when COW has not been implemented**

After Process 1 Modifies Page C



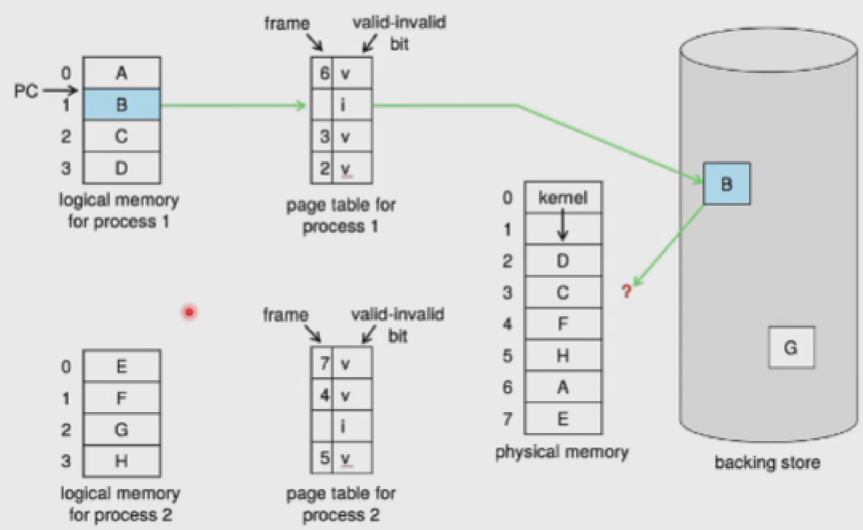
What Happens if There is no Free Frame?

- Used up by process pages
- Also in demand from the kernel, I/O buffers, etc
- How much to allocate to each?
- **Page replacement** – find some page in memory, but not really in use, page it out
  - Algorithm – terminate? swap out? replace the page?
  - Performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times

Page Replacement

- Memory is an important resource, system may run out of memory
- To prevent out-of-memory, swap out some pages
  - page replacement usually is a part of the page fault handler
  - policies to select victim page require careful design
    - need to reduce overhead and avoid **thrashing**
  - use modified (dirty) bit to reduce number of pages to swap out
    - only modified pages are written to disk
  - select some processes to kill (last resort)
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory

Need For Page Replacement



Page Fault Handler (with Page Replacement)

- To page in a page:
  - find the location of the desired page on disk
  - find a free frame:
    - if there is a free frame, use it
    - if there is none, use a page replacement policy to pick a victim frame, write victim frame to disk if dirty
  - bring the desired page into the free frame; update the page tables
  - restart the instruction that caused the trap
- Note now potentially **2 page I/O** for **one page fault** ➡ increase EAT

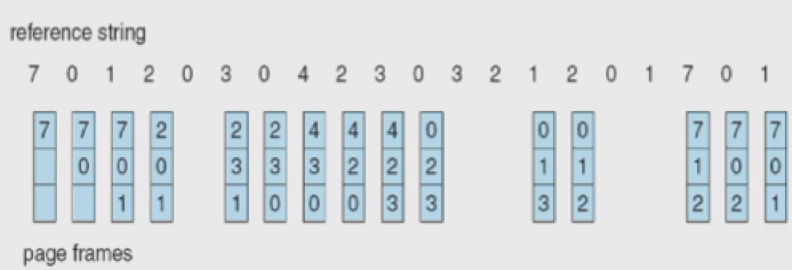
> **Importatnt** and **reference string，知道算法含义，以及给定 reference string 怎么计算缺页次数**

Page Replacement Algorithms

- Page-replacement algorithm should have lowest page-fault rate on both first access and re-access
  - **FIFO**, **optimal**, **LRU**, **LFU**, **MFU**…
- To evaluate a page replacement algorithm:
  - run it on a particular string of memory references (reference string)
    - string is just page numbers, not full addresses
  - compute the number of page faults on that string
    - repeated access to the same page does not cause a page fault
  - in all our examples, the reference string is 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

First-In-First-Out (FIFO)

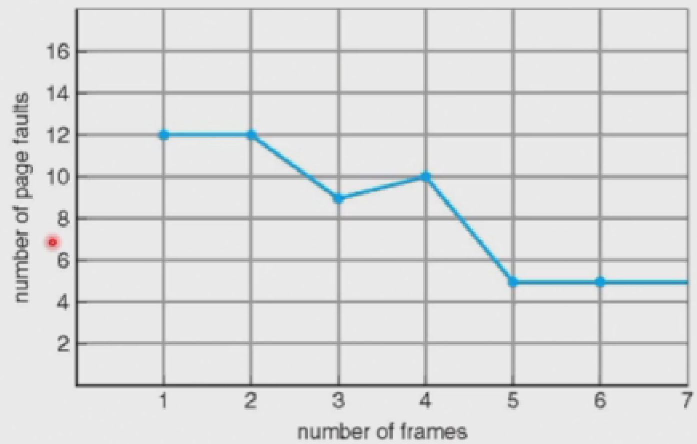
- **FIFO**: replace the first page loaded
  - similar to sliding a window of n in the reference string
  - our reference string will cause 15 page faults with 3 frames
  - how about reference string of 1,2,3,4,1,2,5,1,2,3,4,5 /w 3 or 4 frames?
- For FIFO, adding **more frames** can cause **more page faults**!
  - **Belady's Anomaly**



15 page faults

> belady 异常现象 --- 当系统可用物理页框数量增加，在某些情况下，反而引发缺页的数量也增加（理解为什么，以及哪些算法有这样的异常）

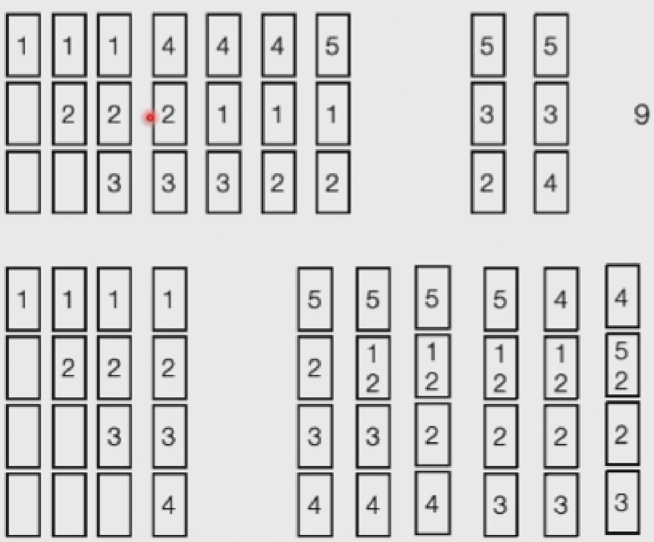
FIFO Illustrating Belady's Anomaly



1 2 3 4 1 2 5 1 2 3 4 5

Belady's Anomaly

1 2 3 4 1 2 5 1 2 3 4 5

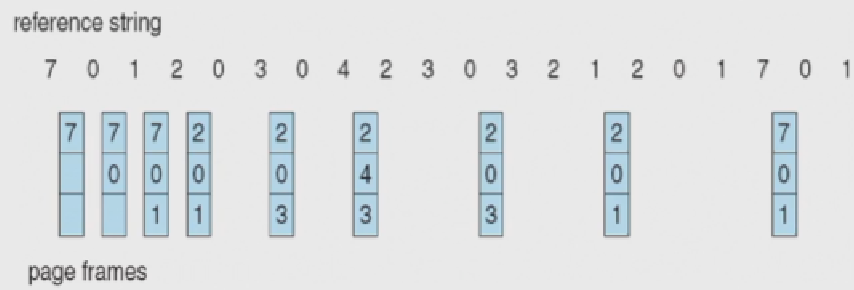


9 page faults

10 page faults!

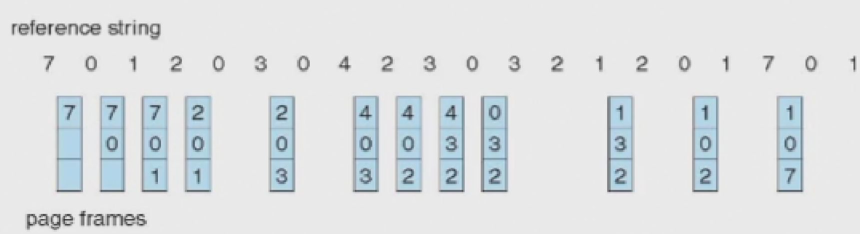
Optimal Algorithm

- **Optimal** : replace page that will not be used for the longest time
  - 9 page fault is optimal for the example on the next slide
- How do you know which page will not be used for the longest time?
  - can't read the future
  - used for measuring how well your algorithm performs



Least Recently Used (LRU)

- **LRU** replaces pages that have not been used for the longest time
  - associate time of last use with each page, select pages w/ oldest timestamp
  - generally good algorithm and frequently used
  - 12 faults for our example, better than FIFO but worse than OPT
- LRU and OPT do **NOT** have Belady's Anomaly
- How to implement LRU?
  - **counter-based**
  - **stack-based**



> LRU 实现算法不要求；

LRU Approximation Implementation

- Counter-based and stack-based LRU have high performance overhead
- Hardware provides a **reference bit**
- LRU approximation with a **reference bit**
  - associate with each page a reference bit, initially set to 0
  - when page is referenced, set the bit to 1 (done by the hardware)
  - replace any page with reference bit = 0 (if one exists)
    - We do not know the **order**, however

> 不要求，知道里面原理就行了
>
> # Enhanced Second-Chance Algorithm
>
> - Improve algorithm by using **reference bit** and modify bit (if available) in concert
> - Take ordered pair (reference, modify):
>   - (0, 0) neither recently used not modified – best page to replace
>   - (0, 1) not recently used but modified – not quite as good, must write out before replacement
>   - (1, 0) recently used but clean – probably will be used again soon
>   - (1, 1) recently used and modified – probably will be used again soon and need to write out before replacement
> - When page replacement called for, use the clock scheme but use the four classes replace page in lowest non-empty class
>   - Might need to search circular queue several times

Allocation of Frames

- Each process needs **minimum number** of frames -according to instructions semantics
- Example: IBM 370 – **6 pages to handle SS MOVE** instruction:
  - instruction is 6 bytes, might span 2 pages
  - 2 pages to handle from
  - 2 pages to handle to
- **Maximum** of course is total frames in the system
- Two major allocation schemes
  - fixed allocation
  - priority allocation
- Many variations

> 抖动

Thrashing

- If a process doesn't have "enough" pages, page-fault rate may be high
  - page fault to get page, replace some existing frame
  - but quickly need replaced frame back
  - this leads to:

    low CPU utilization ➡

    kernel thinks it needs to increase the degree of

    multiprogramming to maximize CPU utilization ➡

    another process added to the system
- **Thrashing**: a process is busy swapping pages in and out

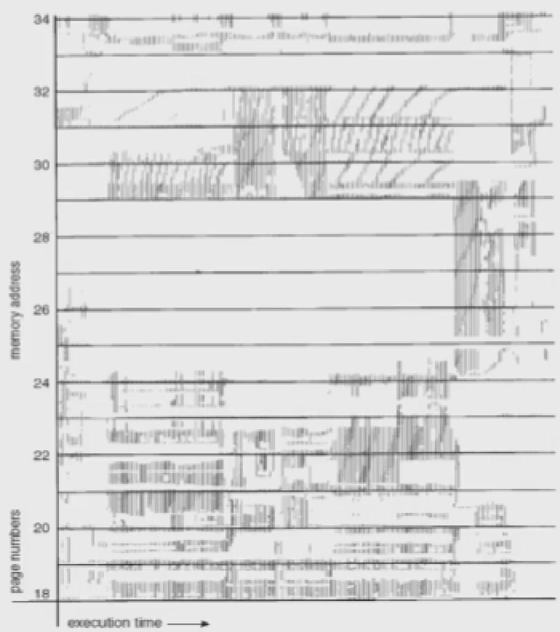
> 解决抖动

Option I

- Limit thrashing effects by **using local or priority page** replacement
  - One process starts thrashing does not affect others -> it cannot cause other processes thrashing

Option II

Provide a process with **as many frames as it needs**. How?



> 第二种方法使用 工作集的模型

Working-Set Model

- **Working-set window**(Δ): a fixed number of page references
  - if Δ too small ➡ will not encompass entire locality
  - if Δ too large ➡ will encompass several localities
  - if Δ = ∞ ➡ will encompass entire program
- **Working set** of process p_i (WSS_i): total number of pages referenced in the most recent Δ (varies in time)
- **Total working sets**: $D = \sum WSS_i$
  - approximation of total locality
  - if D > m ➡ possibility of thrashing
  - to avoid thrashing: if D > m, suspend or swap out some processes

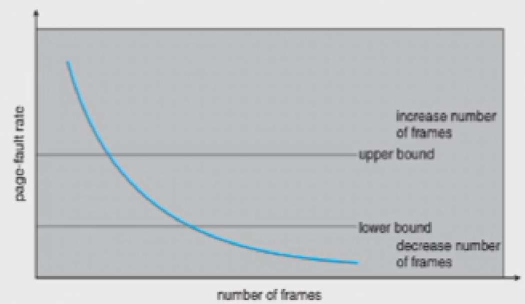
Challenge: Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example: Δ = 10,000
  - Timer interrupts after every 5,000 time units
  - Keep in **memory 2 bits for each page**
  - Whenever a timer interrupts copy and sets the values of all reference bits to 0
  - If one of the bits in memory = 1 ➡ page in working set
- Why is this not completely accurate? - we can not tell **when (in 5000 time unites)** the access occurs
- Improvement = 10 bits and interrupt every **1000** time units

> 工作集难以实现，往往会再通过其他方式知道系统压力状况

Page-Fault Frequency

- More direct approach than WSS
- Establish “acceptable” page-fault frequency (PFF) rate
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame
- Need to swap out a process if no free fames are available



> 内核中的内存分配

Kernel Memory Allocation

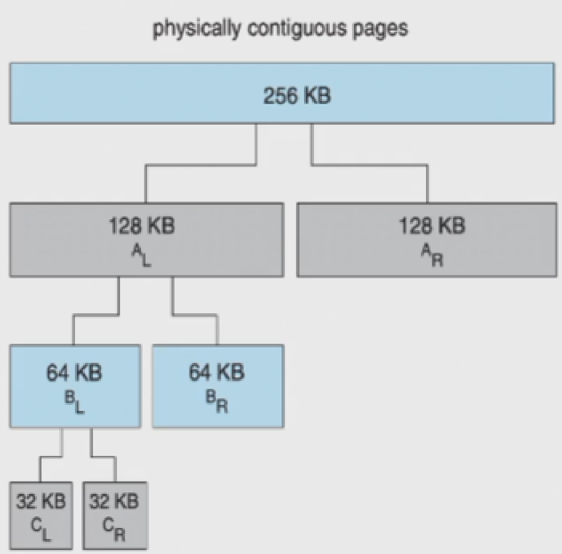
- Kernel memory allocation is treated differently from user memory, it is often allocated from a free-memory pool
  - kernel requests memory for structures of varying sizes -> minimize waste due to fragmentation
  - Some kernel memory needs to be **physically contiguous**
    - e.g., for device I/O

> 几种内存分配算法

Buddy System

- Memory allocated using power-of-2 allocator
  - memory is allocated in units of the size of **power of 2**
    - round up a request to the closest allocation unit
    - split the unit into two **"buddies"** until a proper sized chunk is available
  - e.g., assume only 256KB chunk is available, kernel requests 21KB
    - split it into A_l and A_r of 128KB each
    - further split an 128KB chunk into B_l and B_r of 64KB
    - again, split a 64KB chunk into C_l and C_r of 32KB each
    - give one chunk for the request
- advantage: it can quickly coalesce unused chunks into larger chunk
- disadvantage: **internal fragmentation**
  - 33k request -> 64k segment

Buddy System Allocator



> 在伙伴系统上面抽象出其他的分配器

Slab Allocator

- Slab allocator is a **cache of objects**
  - a **cache** in a slab allocator consists of one or more slabs
  - a Slab contains **one or more pages**, divided into **equal-sized objects**
  - kernel uses one cache for each unique kernel data structure
    - when cache created, allocate a slab, divided the slab into free objects
    - objects for the data structure is allocated from free objects in the slab
    - if a slab is full of used objects, next object comes from an empty/new slab
- Benefits: **no fragmentation** and fast memory allocation
  - some of the object fields may be reusable; no need to initialize again

> Slab 的状态

Slab Allocator in Linux

- For example process descriptor is of type ***struct task_struct***
- Approx 1.7KB of memory
- New task -> allocate new struct from cache
  - Will use existing free ***struct task_struct***
- *A* Slab can be in three possible states
  - **Full** – all used
  - **Empty** – all free
  - **Partial** – mix of free and used
- Upon request, slab allocator
  - Uses free struct in **partial** slab
  - If none, takes one from **empty** slab
  - If no empty slab, create new empty

> 最后知道一些概念

Page Size

- Sometimes OS designers have a choice
  - Especially if running on custom-built CPU
- Page size selection must take into consideration:
  - Fragmentation -> small page size
  - Page table size -> large page size
  - Resolution -> small page size
  - I/O overhead -> large page size
  - Number of page faults -> large page size
  - Locality -> small page size
  - TLB size and effectiveness -> large page size
- Always power of 2, usually in the range 212 (4,096 bytes) to 222 (4,194,304 bytes)
- On average, **growing over time**

TLB Reach

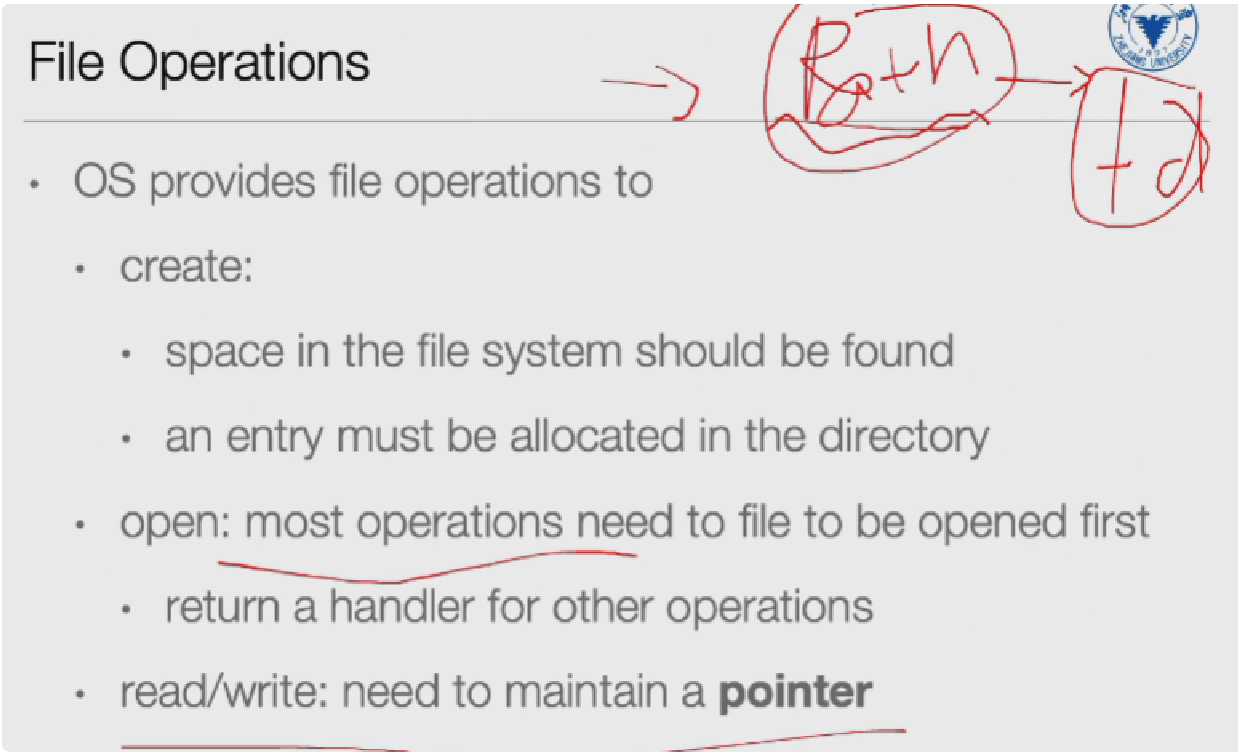
- **TLB reach**: the amount of memory accessible from the TLB
  - TLB reach = (TLB size) X (page size)
- Ideally, the working set of each process is stored in the TLB
  - otherwise there is a high degree of page faults
- **Increase the page size** to reduce **TLB pressure**
  - it may increase fragmentation as not all applications require large page sizes
  - multiple page sizes allow applications that require larger page sizes to use them without an increase in fragmentation

> 理解 demand paging 的过程，处理流程，vbyte, ibyte 是做什么的
> 怎么评估页替换算法，page size 怎么评定好坏

*文件系统

> 需要整体掌握的不是太多

文件目录



Open Files

- Most of file operations need to search the directory to find the named file
    - To avoid the searching, OS maintains a table - **open-file table** contains information about all open files
    - Then following operation is specified via an index to the table - no searching is required
- For os that several processes may open the file simultaneously
    - **Per-process table:** current location pointer, access rights
    - **System-wide table:** location on the disk …

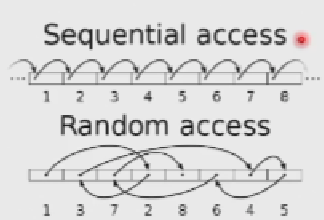
> 两张表的内容有什么，根据内容应该属于哪个表

File Structure

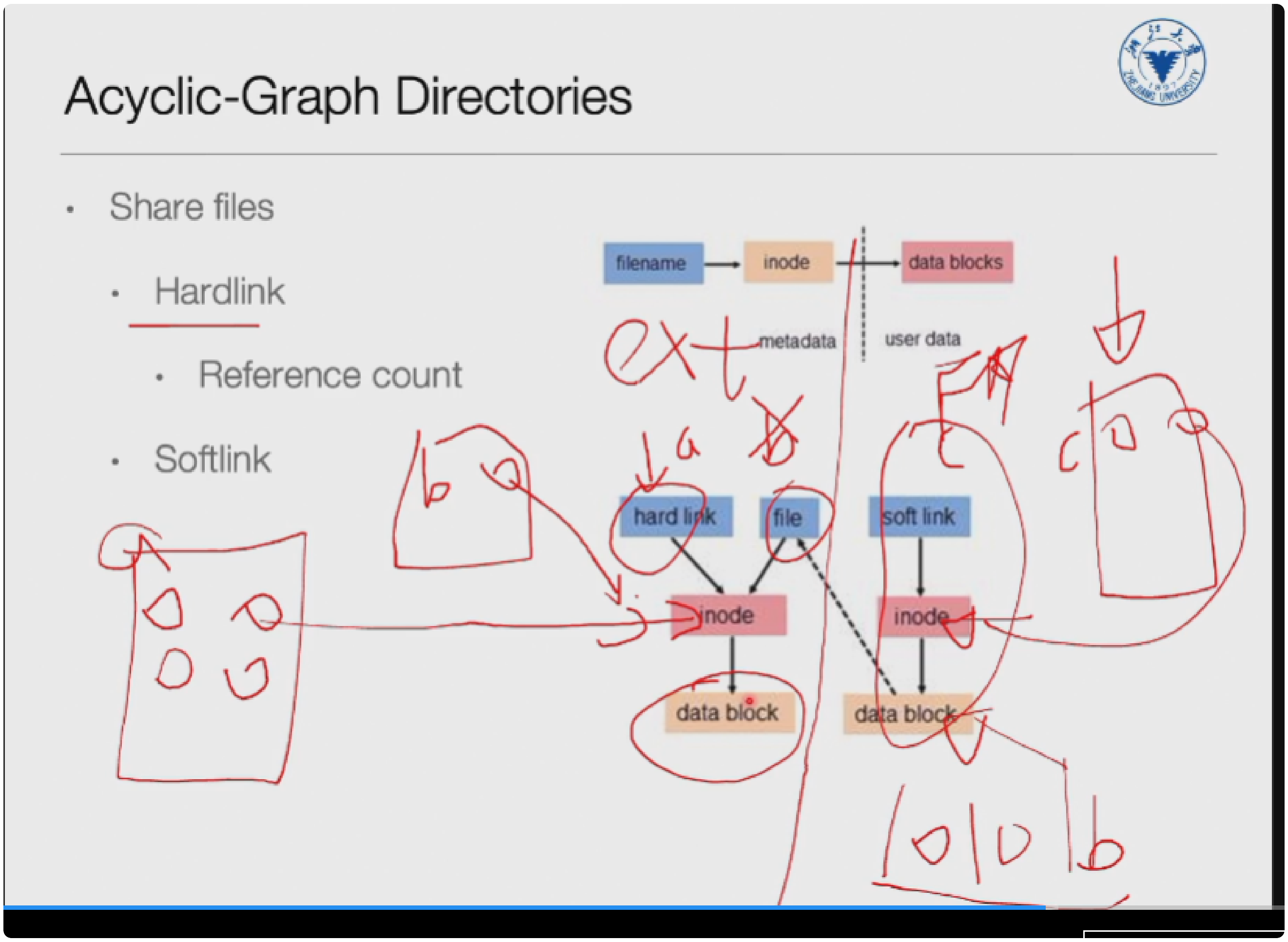
- A file can have different structures, determined by OS or program
  - **no structure**: a stream of bytes or words
    - linux files
  - **simple record structure**
    - lines of records, fixed length or variable length
    - e.g., database
  - **complex structures**
    - e.g., word document, relocatable program file
  - simple and complex structure can be encoded in the first method
- Usually user programs are responsible for identifying file structure

Access Methods

- Sequential access
  - a group of elements is access **in a predetermined order**
  - for some media types, the only access mode (e.g., **tape**)
- Direct access
  - access an element at an **arbitrary position** in a sequence in (roughly) **equal time**, independent of sequence size
    - it is possible to emulate random access in a tape, but access time varies
  - sometime called random access



硬链接和软链接的概念；硬链接不能跨文件系统，删除一个文件的时候，需要删除 data block 和 inode（只有当没有 inode 指向它的时候才会删除）
软链接 data block 较特殊



文件系统的实现

> 掌握文件系统实现的基本思想

File-System Structure

- File is a logical storage unit for a collection of related information
- There are many file systems; OS may support several **simultaneously**
  - Linux has Ext2/3/4, Reiser FS/4, Btrfs…
  - Windows has FAT, FAT32, NTFS…
  - new ones still arriving – ZFS, GoogleFS, Oracle ASM, FUSE
- File system resides on **secondary storage** (disks)
  - disk driver provides interfaces to read/write disk blocks
  - **fs provides user/program interface to storage, mapping logical to physical**
    - file control block – storage structure consisting of information about a file
- File system is usually implemented and organized into **layers**

File-System Implementation

- partition == volume == file system storage space
- File-system needs to maintain **on-disk** and **in-memory** structures
  - on-disk for data storage, in-memory for data access
- **On-disk structure** has several control blocks
  - **boot control block** contains info to boot OS from that volume - per volume
    - only needed if volume contains OS image, usually first block of volume
  - **volume control block** (e.g., *superblock*) contains volume details - per volume
    - total # of blocks, # of free blocks, block size, free block pointers, free FCB count, free FCB pointers
  - **directory structure** organizes the directories and files - per file system
    - A list of **(file names and associated inode numbers)**
  - **per-file file control block** contains many **details about the file - per file**
    - permissions, size, dates, data blocks or pointer to data blocks

In-Memory File System Structures

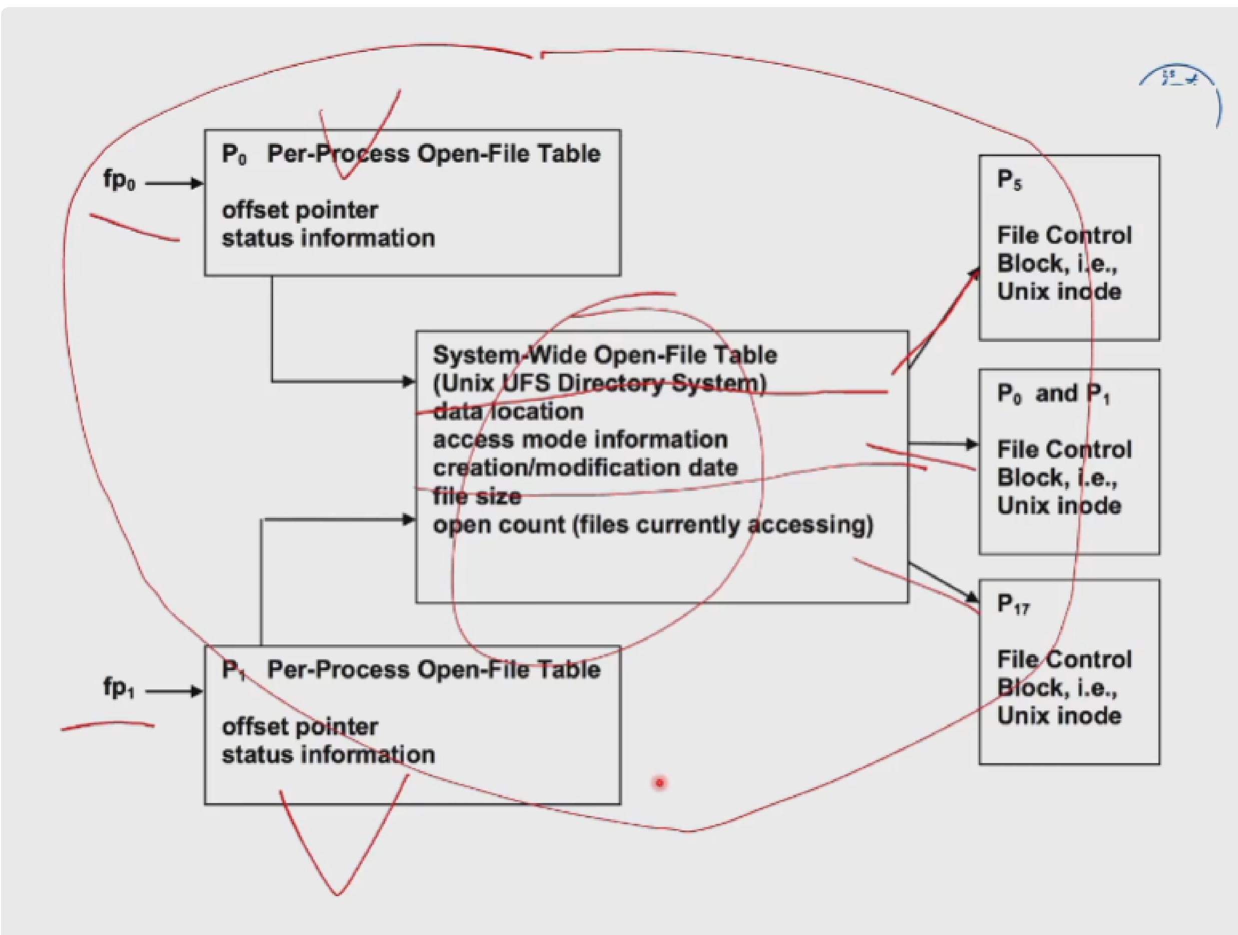
- **In-memory structures** reflects and extends **on-disk structures**
  - **Mount table** storing file system mounts, mount points, file system types
  - In-memory directory-structure cache: holds the directory information about recently accessed directories
  - **system-wide open-file table** contains a copy of the FCB of each file and other info
  - **per-process open-file table** contains pointers to appropriate entries in system-wide open-file table as well as other info
  - **I/O Memory Buffers**: hold file-system blocks while they are being read from or written to disk

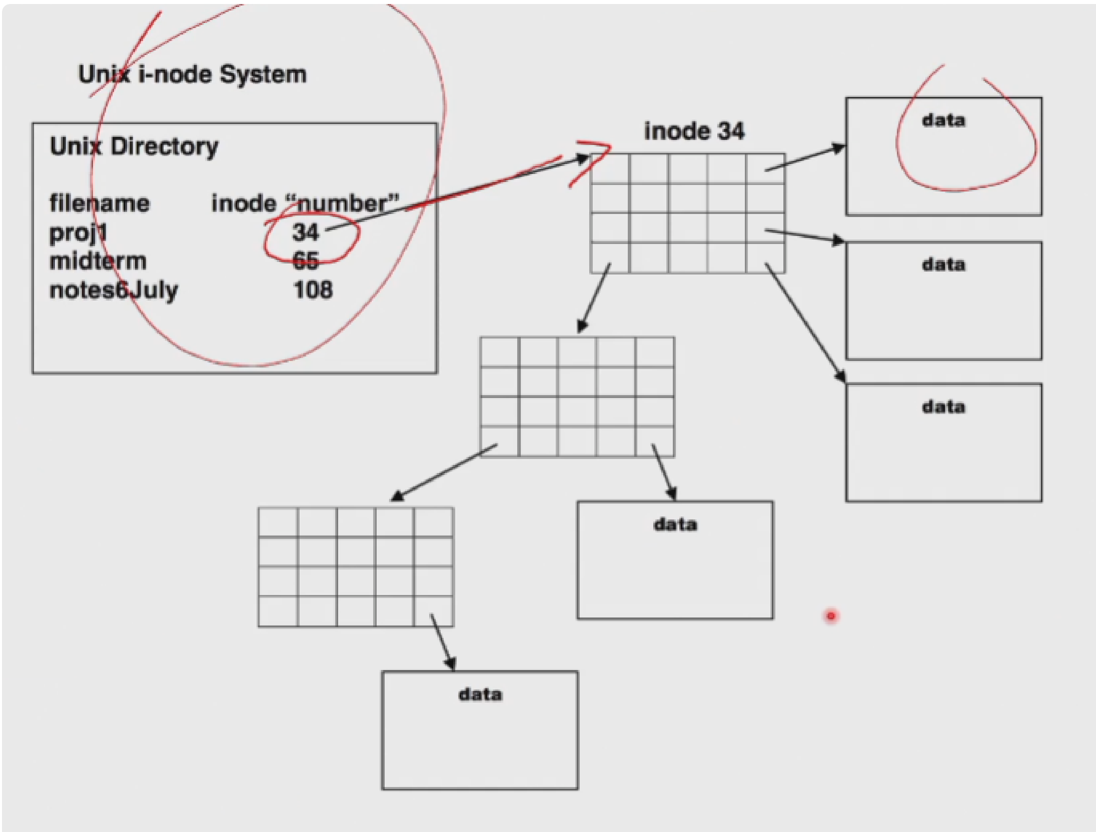
Operations - open()

- search **System-Wide Open-File Table** to see if file is currently in use
  - if it is, create a Per-Process Open-File table entry pointing to the existing System-Wide Open-File Table
  - if it is not, search the directory for the file name; once found, place the **FCB** in the System-Wide Open-File Table
- make an entry, i.e., Unix file descriptor, Windows file handle in the **Per-Process Open-File Table**, with pointers to the entry in the **System-Wide Open-File Table** and other fields which include a pointer to the current location in the file and the access mode in which the file is open

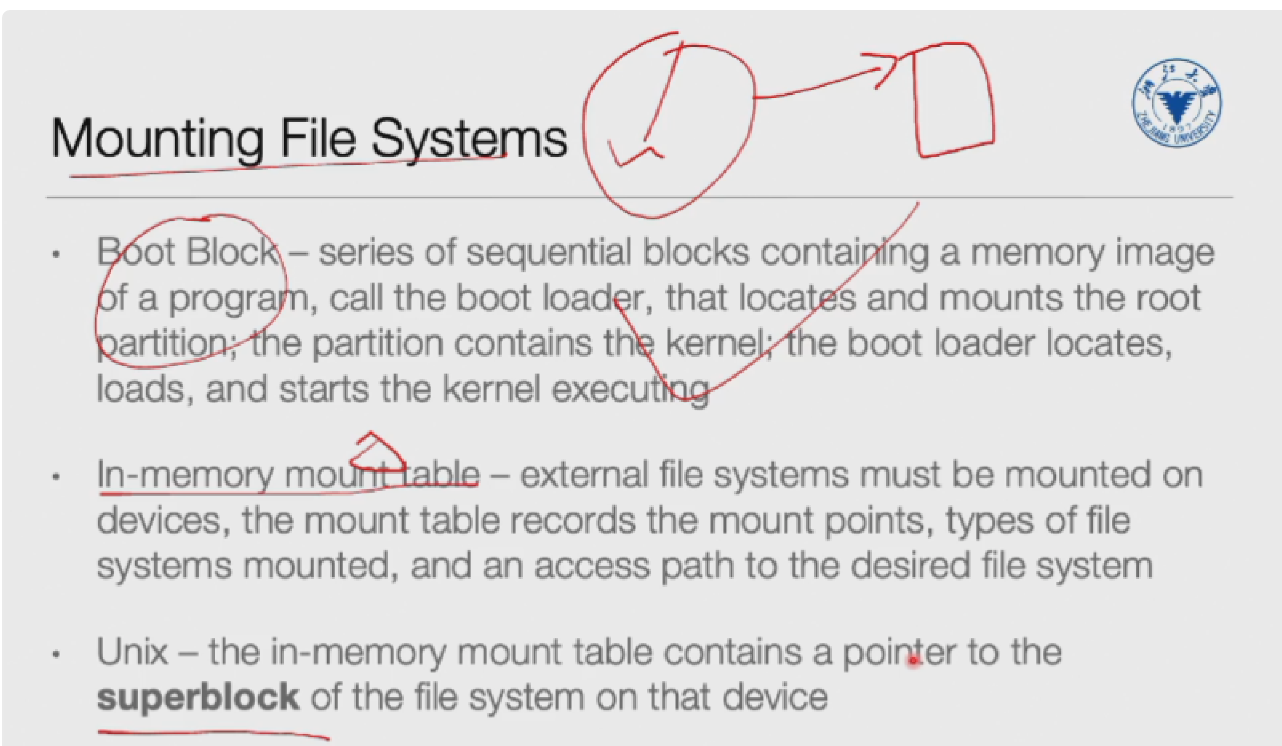
Operations - open()

- increment the open count in the System-Wide Open-File Table
- returns a pointer to the appropriate entry in the Per-Process Open-File Table
- all subsequent operations are performed with **this pointer**
- process closes file -> Per-Process Open-File Table entry is removed; **open count decremented**
- all processes close file -> copy in-memory directory information to disk and System-Wide Open-File Table is removed from memory





> 使用文件系统的时候



> 目录下放的到底是什么

Directory Implementation

- **Linear list of file names** with pointer to the file metadata
  - simple to program, but **time-consuming to search** (e.g., linear search)
    - could keep files ordered alphabetically via linked list or use B+ tree
- **Hash table**: linear list with hash data structure to reduce search time
  - collisions are possible: two or more file names hash to the same location

> 文件系统中的 data 在文件系统中是怎么存储的

Disk Block Allocation

- **Files** need to be allocated with disk blocks to store data
  - different allocation strategies have different complexity and performance
- Many allocation strategies:
  - contiguous
  - linked
  - indexed
  - …

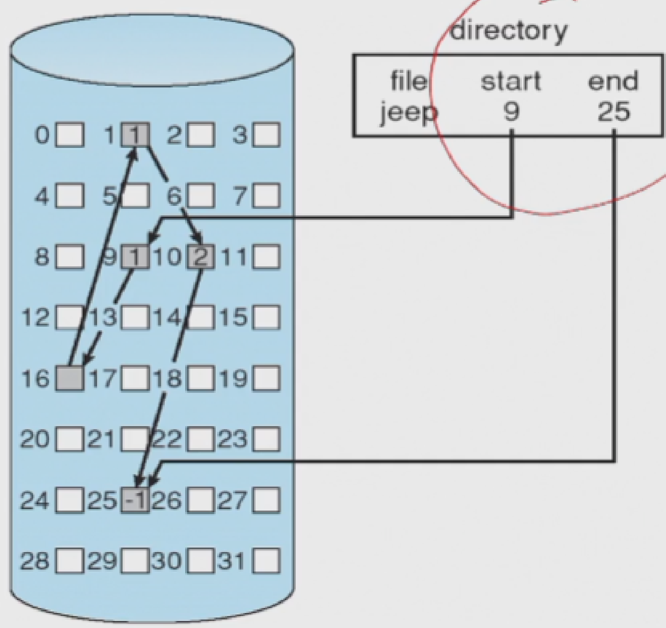
Contiguous Allocation

- Contiguous allocation: each file occupies set of **contiguous blocks**
  - best performance in most cases
  - simple to implement: only starting location and length are required
- Contiguous allocation is not flexible
  - how to *increase/decrease* file size?
    - need to know file size at the file creation?
  - **external fragmentation**
    - how to compact files offline or online to reduce external fragmentation
  - need for **compaction** off-line (downtime) or on-line
  - appropriate for sequential disks like **tape**
- Some file systems use **extent-based contiguous allocation**
  - extent is a set of contiguous blocks
  - a file consists of extents, extents are not necessarily adjacent to each other

Linked Allocation

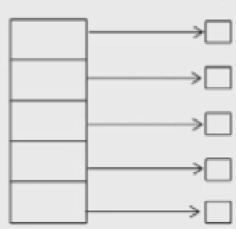
- Linked allocation: each file is a **linked list of disk blocks**
  - each block contains pointer to **next block**, file ends at nil pointer
  - blocks may be scattered anywhere on the disk (no **external fragmentation, no compaction**)
  - *Disadvantages*
    - *locating a file block can take many I/Os and disk seeks*
    - *Pointer size: 4 of 512 bytes are used for pointer - 0.78% space is wasted*
    - *Reliability: what about the pointer has corrupted!*
  - *Improvements: cluster the blocks - like 4 blocks*
    - *however, has internal fragmentation*

Linked Allocation



Indexed Allocation

- Indexed allocation: each file has its own **index blocks of pointers to its data blocks**
  - index table provides **random access** to file data blocks
  - no **external fragmentation**, but overhead of index blocks
  - allows **holes** in the file
  - Index block needs space - waste for small files



index table

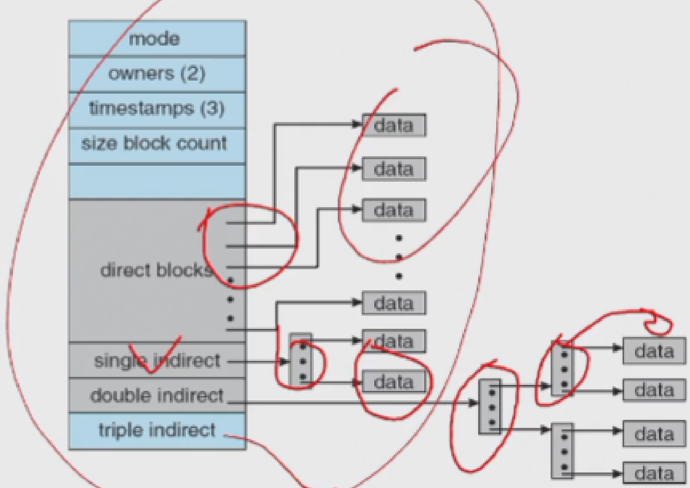
Indexed Allocation

- Need a method to allocate index blocks - cannot too big or too small
  - linked index blocks: link index blocks to support huge file
  - multiple-level index blocks (e.g., 2-level)

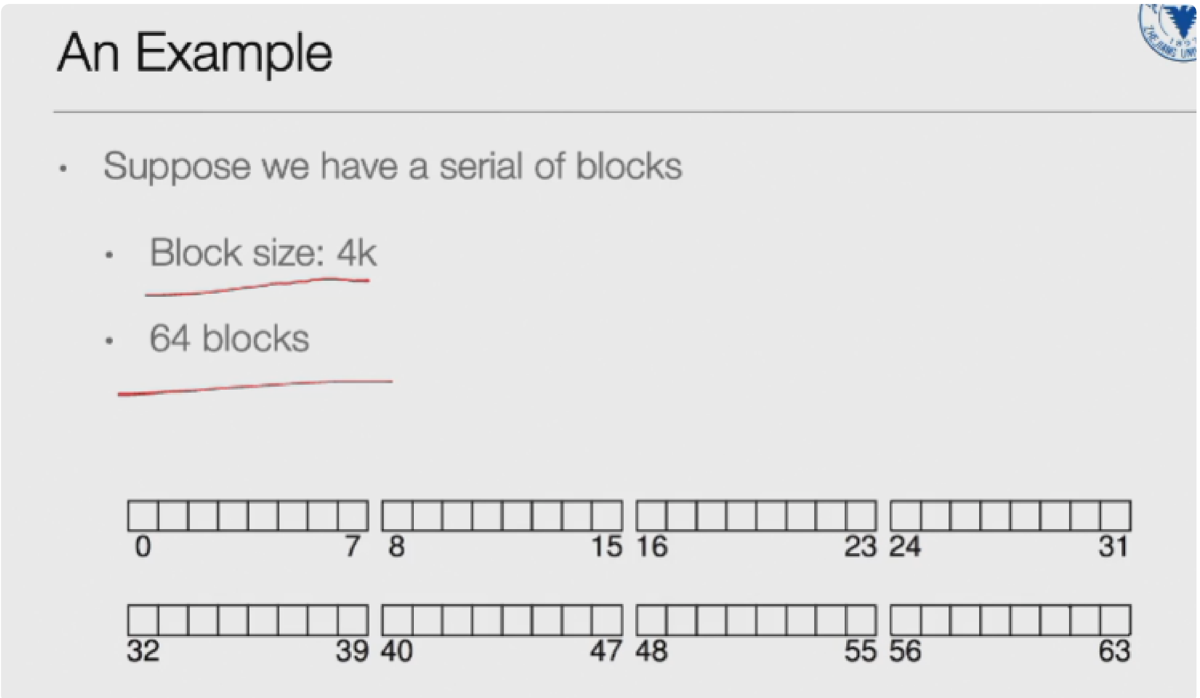
Indexed Allocation

$12\times D + b/4\times$

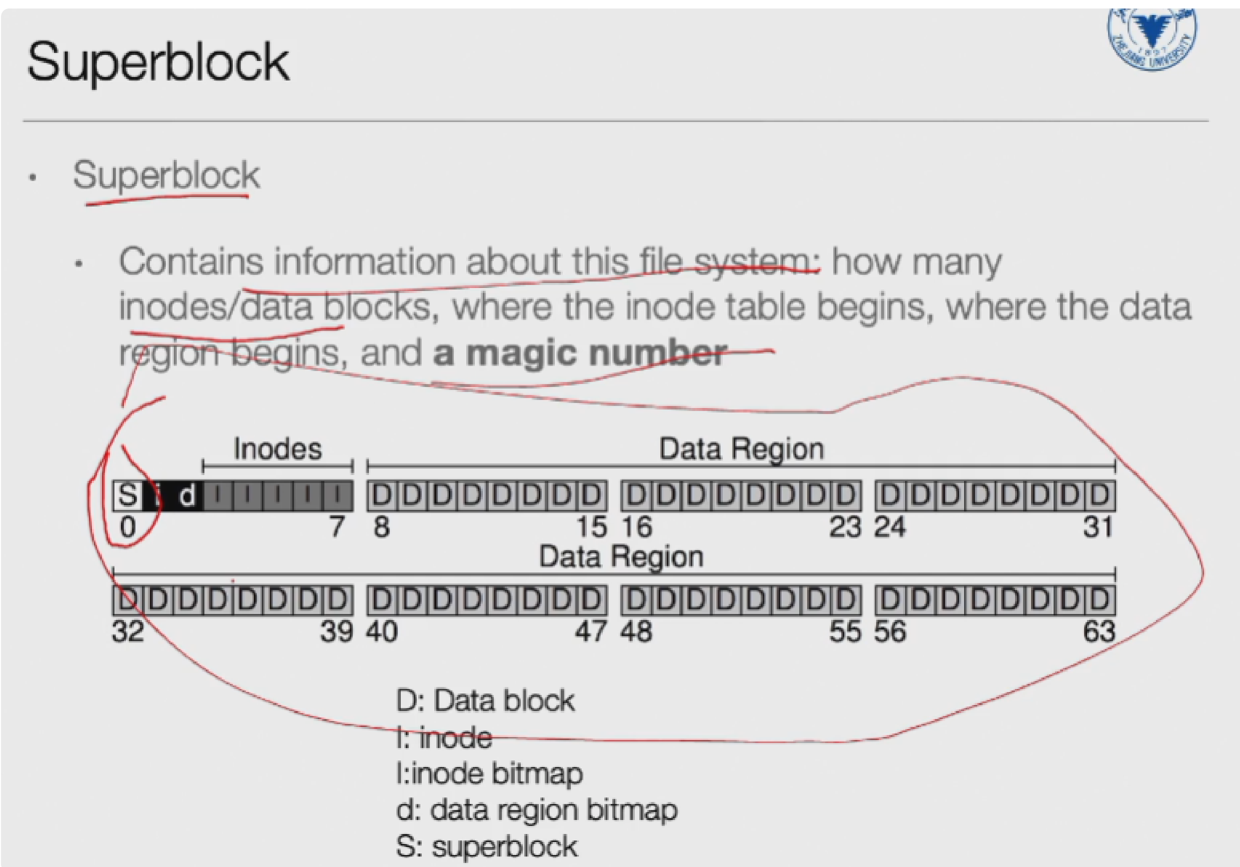
- combined scheme
  - First 15 pointers are in inode
    - Direct block: first 12 pointers
    - Indirect block: next 3 pointers



在 index Allocation 的情况下算出文件系统最大支持的文件的大小



这个例子要学会这样的文件系统中有哪几样东西需要分配（这个中有五种结构体），最终支持的文件数量多少（inode 支持多少个）

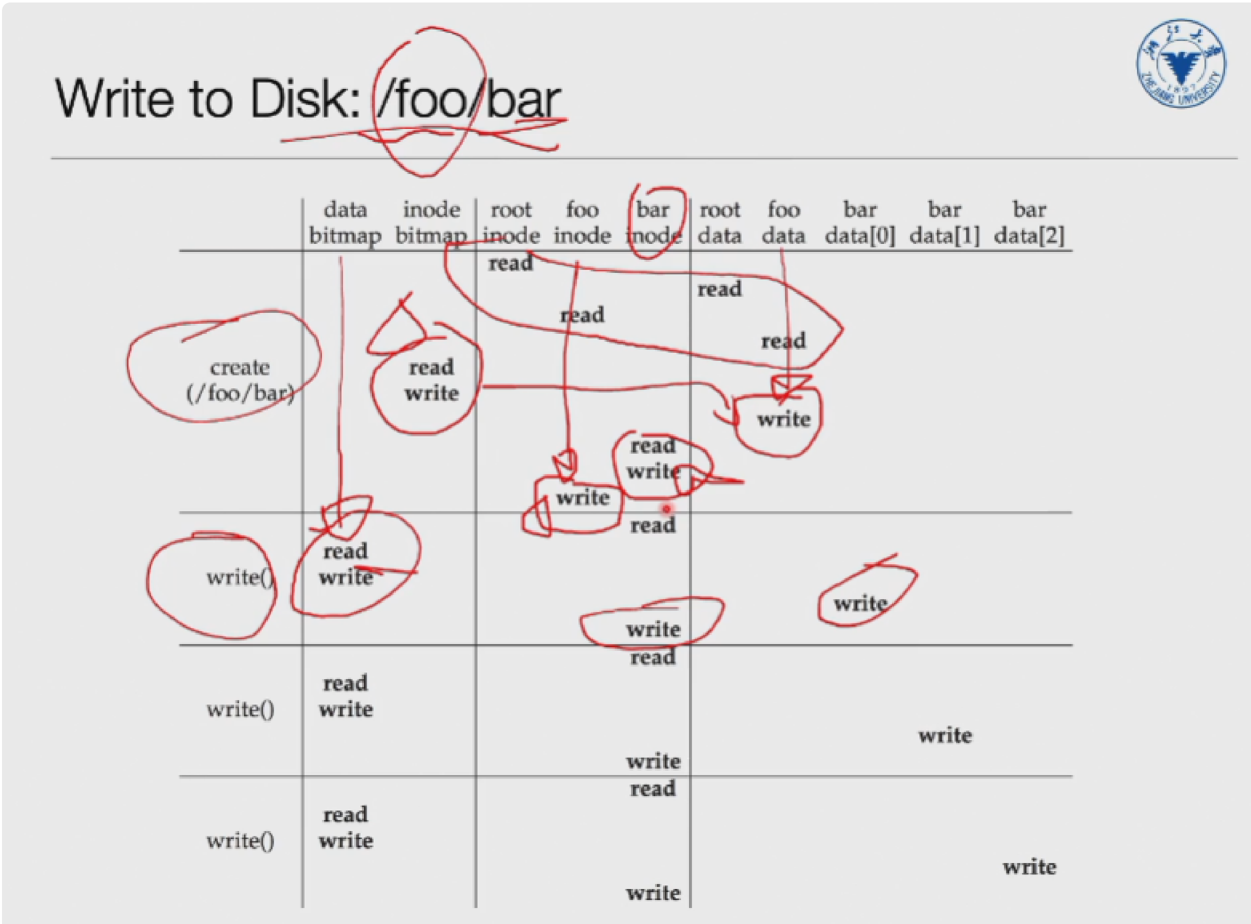


在此基础是，读写的时候对这些结构体做了什么操作

Read /foo/bar

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
open(bar)			read	read	read	read	read			
read()					read write			read		
read()					read write				read	
read()					read write					read

What about the system-wide/per-process open file table?



Storage & I/O

Storage

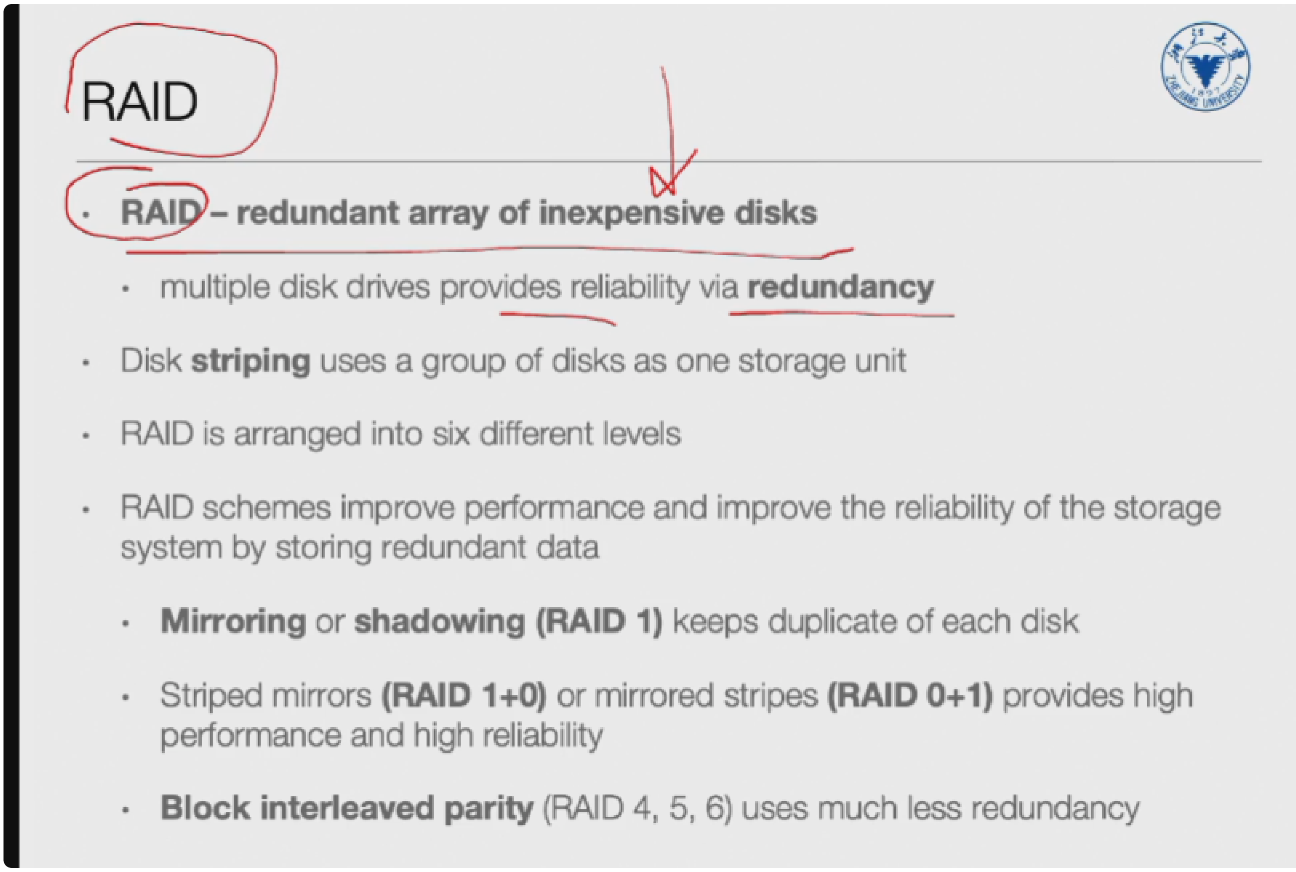
Disk Scheduling

- Disk scheduling usually tries to minimize **seek time**
  - rotational latency is difficult for OS to calculate
- There are many disk scheduling algorithms
  - FCFS
  - SSTF
  - SCAN
  - C-SCAN
  - C-LOOK
- We use a request queue of "**98, 183, 37, 122, 14, 124, 65, 67**" (**[0, 199]**), and initial head position **53** as the example

> 给出 queue 算出磁头移动的距离

掌握这几种算法

> 用冗余性换取磁盘的可靠性


RAID
RAID – redundant array of inexpensive disks
multiple disk drives provides reliability via redundancy
Disk striping uses a group of disks as one storage unit
RAID is arranged into six different levels
RAID schemes improve performance and improve the reliability of the storage system by storing redundant data
Mirroring or shadowing (RAID 1) keeps duplicate of each disk
Striped mirrors (RAID 1+0) or mirrored stripes (RAID 0+1) provides high performance and high reliability
Block interleaved parity (RAID 4, 5, 6) uses much less redundancy

I/O

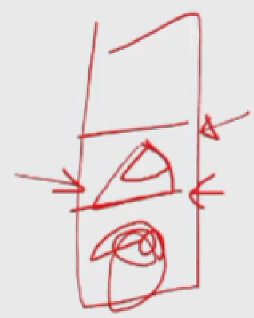
I/O Hardware

- Incredible variety of I/O devices
  - storage, communication, human-interface
- Common concepts: signals from I/O devices interface with computer
  - **bus**: an interconnection between components (including CPU)
  - **port**: connection point for device
  - **controller**: component that control the device
    - can be integrated to device or separate circuit board
    - usually contains processor, microcode, private memory, bus controller, etc
- I/O access can use **polling** or **interrupt**

> 中断处理方式是不是一定比 polling 好，什么时候比 polling 好，为什么

I/O Hardware

- Devices are assigned addresses for registers or on-device memory
  - **direct I/O instructions**
    - to access (mostly) registers
  - **memory-mapped I/O**
    - data and command registers mapped to processor address space
    - to access (large) on-device memory (graphics)



Direct Memory Access

- DMA transfer data directly between I/O device and memory
  - OS only need to issue commands, data transfers bypass the CPU
  - no programmed I/O (one byte at a time), data transferred in large blocks
  - it requires DMA controller in the device or system
- OS issues commands to the DMA controller
  - a command includes: operation, memory address for data, count of bytes…
  - usually it is the pointer of the command written into the command register
  - when done, device interrupts CPU to signal completion

Characteristics of I/O Devices

- Broadly, I/O devices can be grouped by the OS into
  - **block I/O: read, write, seek**
  - **character I/O** (Stream)
  - **memory-mapped file access**
  - **network sockets**
- Direct manipulation of I/O device: usually an escape / back door
  - Linux's **ioctl** call to send commands to a device driver

read( )

Synchronous/Asynchronous I/O

- **Synchronous I/O** includes blocking and non-blocking I/O
  - **blocking I/O**: process suspended until I/O completed
    - easy to use and understand, but may be less efficient
    - insufficient for some needs
  - **non-blocking I/O**: I/O calls return **as much data as available**
    - process does not block, returns whatever existing data (read or write)
    - use **select to find if data is ready**, then **use read or write to transfer data** (blocking during this process)
- **Asynchronous I/O**: process runs while I/O executes,
  - I/O subsystem signals process when I/O completed via signal or callback - **data is already in the buffer, no need to use read() to get the data**
  - difficult to use but very efficient

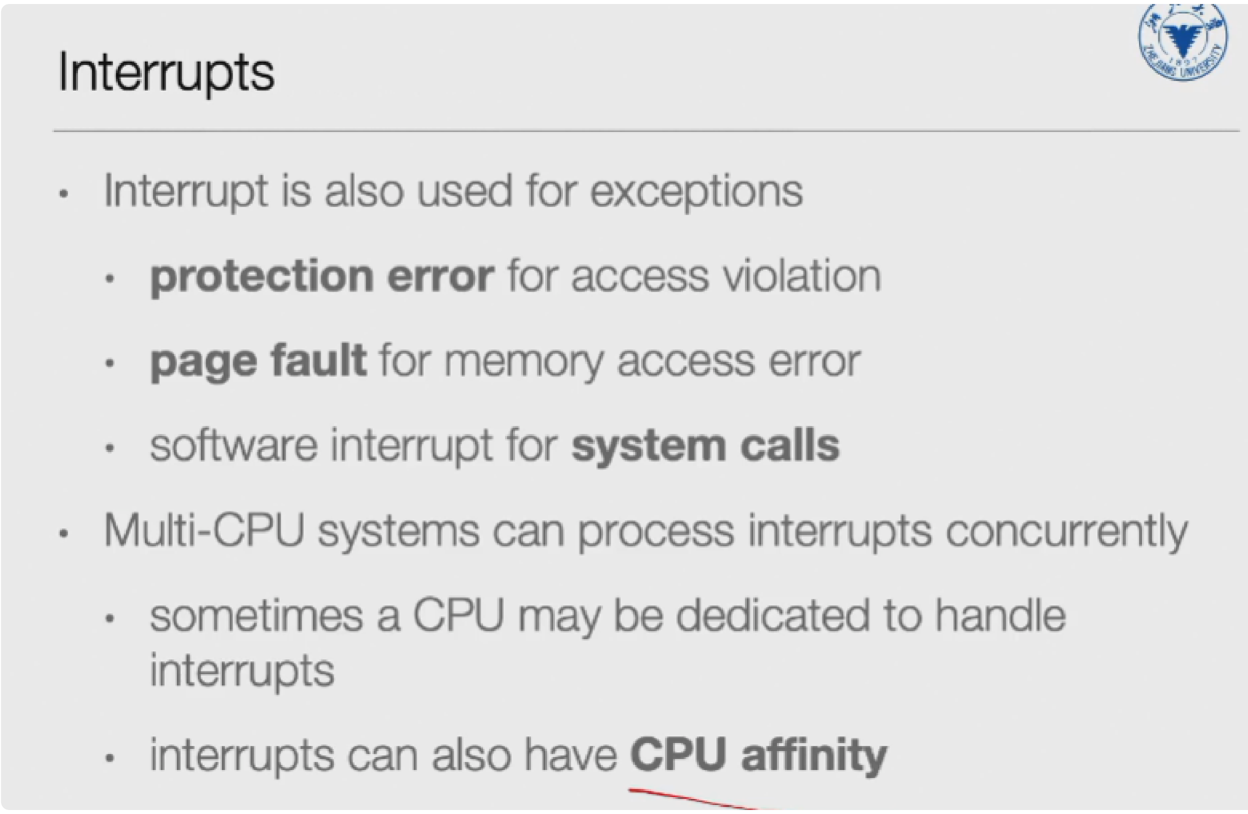
> 同步是指由进程本身来完成，异步是由操作系统就完成了

I/O Protection

- OS need to protect I/O devices
  - e.g., keystrokes can be stolen by a **keylogger** if keyboard is not protected
  - always assume user may attempt to obtain illegal I/O access
- To protect I/O devices:
  - define all I/O instructions to be privileged
    - I/O must be performed via system calls
  - memory-mapped I/O and I/O ports must be protected too

Improve Performance

- Reduce number of context switches
- Reduce data copying
- Reduce interrupts by using large transfers, smart controllers, polling
- Use DMA
- Use smarter hardware devices
- Balance CPU, memory, bus, and I/O performance for highest throughput
- Move user-mode processes / daemons to kernel threads



> I/O 要掌握中断、polling、DMA、一些特性、同步异步、阻塞非阻塞