# Operating Systems Lecture

# Final Exercises

Prof. Mengwei Xu

# 进程线程

1.（单选）进程相较于线程的优势，以下哪一项说法正确？

a.进程创建更快、更轻量化

b.进程间隔离更彻底、安全

c.进程间通信更容易实现

d.进程间切换速度更快

# 进程线程

1.（单选）进程相较于线程的优势，以下哪一项说法正确？

a.进程创建更快、更轻量化

b.进程间隔离更彻底、安全

c.进程间通信更容易实现

d.进程间切换速度更快

# 进程线程

1.加速一个矩阵运算的方法包括

a.在单核处理器上使用多线程

b.在多核处理器上使用多线程

c.通过优化运算顺序提高缓存命中率

# 进程线程

1.加速一个矩阵运算的方法包括

a.在单核处理器上使用多线程

b.在多核处理器上使用多线程

c.通过优化运算顺序提高缓存命中率

# 内存

1.（单选）虚拟地址向物理地址翻译的过程中，哪一项不会被用到

a.CPU中的MMU

b.CPU中的TLB

c.二级存储（磁盘）

d.指向页目录的寄存器（CR3）

e.内存中的页表

# 内存

1.（单选）虚拟地址向物理地址翻译的过程中，哪一项不会被用到

a.CPU中的MMU

b.CPU中的TLB

c.二级存储（磁盘）

d.指向页目录的寄存器（CR3）

e.内存中的页表

# 内存

1.以下哪些不是内存地址翻译的目的/功能？

a.进程间内存保护与隔离

b.提高缓存命中率

c.进程间内存共享

d.为进程提供连续、大块内存的抽象

# 内存

1.以下哪些不是内存地址翻译的目的/功能？

a.进程间内存保护与隔离

b.提高缓存命中率

c.进程间内存共享

d.为进程提供连续、大块内存的抽象

# 内存

1.在一个32-bit、2级页表、页大小为4KB的系统中，以下哪些说法是正确的？

a. 不考虑TLB/Cache的情况下，每次读取内存数据需要进行3次内存访问

b. 页目录中相邻页目录项指向的页表在物理内存中是连续的

c. 页表中相邻页表项指向的物理页在物理内存中是连续的

d. 同一个页表中，相邻页表项的数据在物理内存中是连续的

# 内存

1.在一个32-bit、2级页表、页大小为4KB的系统中，以下哪些说法是正确的？

a. 不考虑TLB/Cache的情况下，每次读取内存数据需要进行3次内存访问

b. 页目录中相邻页目录项指向的页表在物理内存中是连续的

c. 页表中相邻页表项指向的物理页在物理内存中是连续的

d. 同一个页表中，相邻页表项的数据在物理内存中是连续的

# 内存

1. 下列关于分段（segmentation）和分页（paging）叙述中，正确的有？

a. 分段和分页管理的区别之一是内存分配单元是否固定大小

b. 分段内存管理方式不具有权限管理能力

c. 在进程上下文切换时，分段比分页速度更快

d. 采用分页管理方式时，产生缺页中断就会淘汰一个页面

e. 分段管理的物理内存碎片化更为严重

# 内存

1. 下列关于分段（segmentation）和分页（paging）叙述中，正确的有？

a. 分段和分页管理的区别之一是内存分配单元是否固定大小

b. 分段内存管理方式不具有权限管理能力

c. 在进程上下文切换时，分段比分页速度更快

d. 采用分页管理方式时，产生缺页中断就会淘汰一个页面

e. 分段管理的物理内存碎片化更为严重

# 同步

1.线程A和B共享整型x的值，分别执行两行代码A{x=0;x=1;}；B{x=0；x=2;}，程序结束时，x的值可能是

a.0

b.1

c.2

d.3

# 同步

1.线程A和B共享整型x的值，分别执行两行代码A{x=0;x=1;}；B{x=0；x=2;}，程序结束时，x的值可能是

a.0

b.1

c.2

d.3

# 同步

1.（单选）以下那种情况，不需要使用同步机制（锁、信号量等）？

a.线程间没有共享资源

b.资源无限

c.没有并发的程序

d.以上都不需要

# 同步

1.（单选）以下那种情况，不需要使用同步机制（锁、信号量等）？

a.线程间没有共享资源

b.资源无限

c.没有并发的程序

d.以上都不需要

# 概念

1.（单选）以下哪项不是（常见）操作系统的任务/功能？

a.管理硬件资源

b.隔离进程的地址空间

c.处理系统调用、中断、异常

d.防止用户进程进入死锁状态

# 概念

1.（单选）以下哪项不是（常见）操作系统的任务/功能？

a.管理硬件资源

b.隔离进程的地址空间

c.处理系统调用、中断、异常

d.防止用户进程进入死锁状态

# 概念

1.以下哪些关于操作系统的说法是正确的？

a.BIOS是操作系统的一部分

b.用户态与内核态是指CPU运行的状态

c.中断处理程序（包括系统调用）是用户态进入内核态唯一的入口程序。

d.部分中断可以被屏蔽；该屏蔽操作是特权指令。

# 概念

1.以下哪些关于操作系统的说法是正确的？

a.BIOS是操作系统的一部分

b.用户态与内核态是指CPU运行的状态

c.中断处理程序（包括系统调用）是用户态进入内核态唯一的入口程序。

d.部分中断可以被屏蔽；该屏蔽操作是特权指令。

# 堆栈

1.以下关于堆栈的说法，正确的是？

a.用户态进程对用户栈的操作（push，pop）会陷入操作系统内核

b.同一个进程的不同线程共享一个用户态的栈

c.同一个进程的不同线程共享一个中断栈

d.用户态进程陷入内核后，用户态栈指针会被存在中断栈中

e.通过malloc分配的堆，在物理内存上可能是不完全连续的

# 堆栈

1.以下关于堆栈的说法，正确的是？

a.用户态进程对用户栈的操作（push，pop）会陷入操作系统内核

b.同一个进程的不同线程共享一个用户态的栈

c.同一个进程的不同线程共享一个中断栈

d.用户态进程陷入内核后，用户态栈指针会被存在中断栈中

e.通过malloc分配的堆，在物理内存上可能是不完全连续的

# 进程线程

1.（单选）以下关于线程的说法，不正确的是

a.在单核处理器上无法实现并发（concurrency）

b.线程是操作系统最小的独立调度单元

c.同一个进程的不同线程，对同一个指针取值，得到的结果是一样的

d.线程间切换需要保存部分寄存器值和栈指针

# 进程线程

1.（单选）以下关于线程的说法，不正确的是

a.在单核处理器上无法实现并发（concurrency）

b.线程是操作系统最小的独立调度单元

c.同一个进程的不同线程，对同一个指针取值，得到的结果是一样的

d.线程间切换需要保存部分寄存器值和栈指针

# 内存

1.对于一个32-bit操作系统，当页大小从4KB变为8KB后，会导致

a.页表变大

b.页表项用于保存页信息（dirty、r/w、valid/present等）的比特数变多

c.表示页内偏移的比特数变多

d.可以寻址的物理地址变大

# 内存

1.对于一个32-bit操作系统，当页大小从4KB变为8KB后，会导致

a.页表变大

b.页表项用于保存页信息（dirty、r/w、valid/present等）的比特数变多

c.表示页内偏移的比特数变多

d.可以寻址的物理地址变大

# 内存

1.以下关于缺页中断（page fault）的说法正确的是？

a.发生缺页中断的程序会产生异常并崩溃

b.缺页中断可能是由于访问的虚拟地址没有相应的物理页映射

c.缺页中断处理程序中可能会修改页表

# 内存

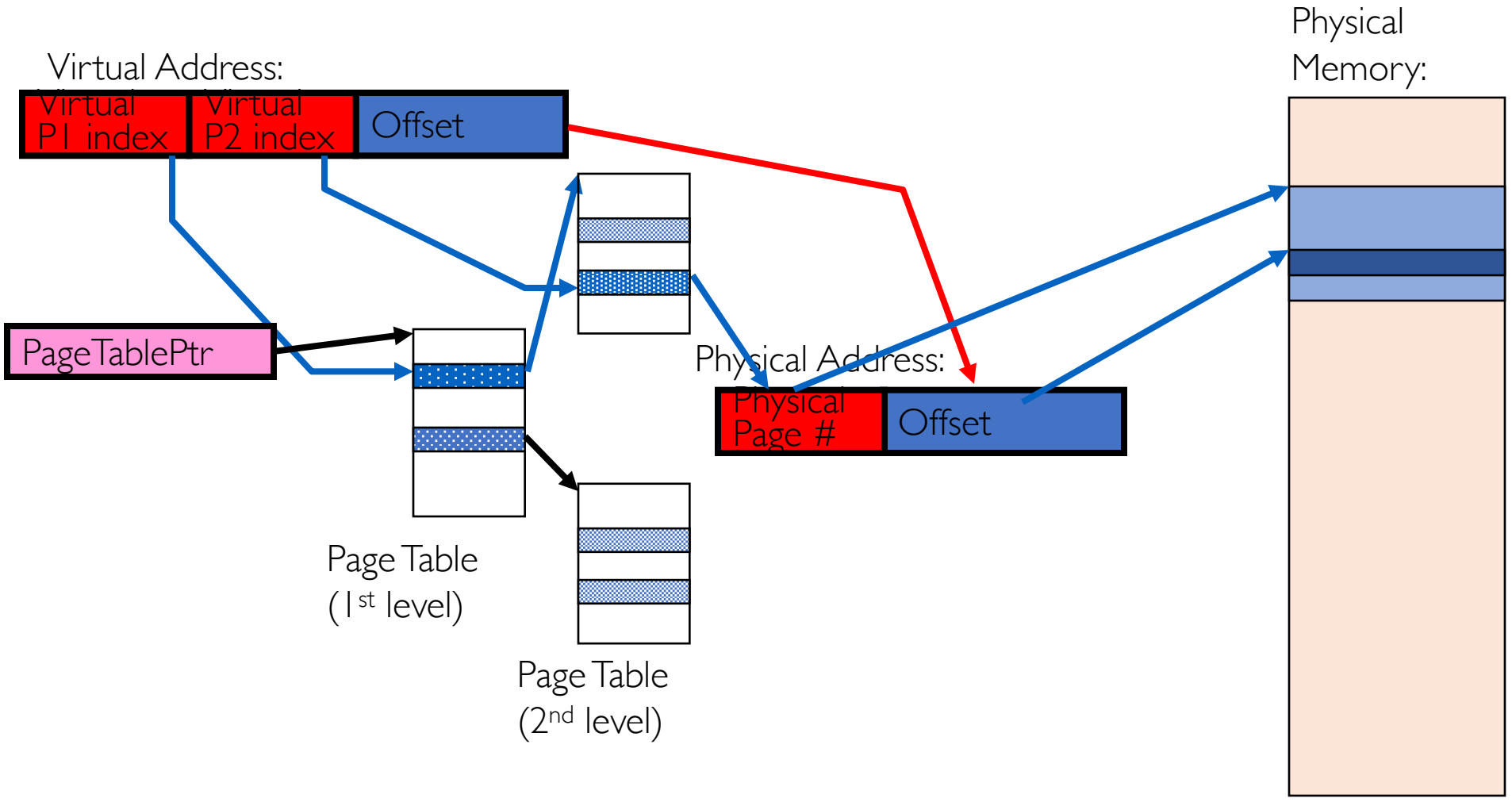
1.以下关于缺页中断（page fault）的说法正确的是？

a.发生缺页中断的程序会产生异常并崩溃

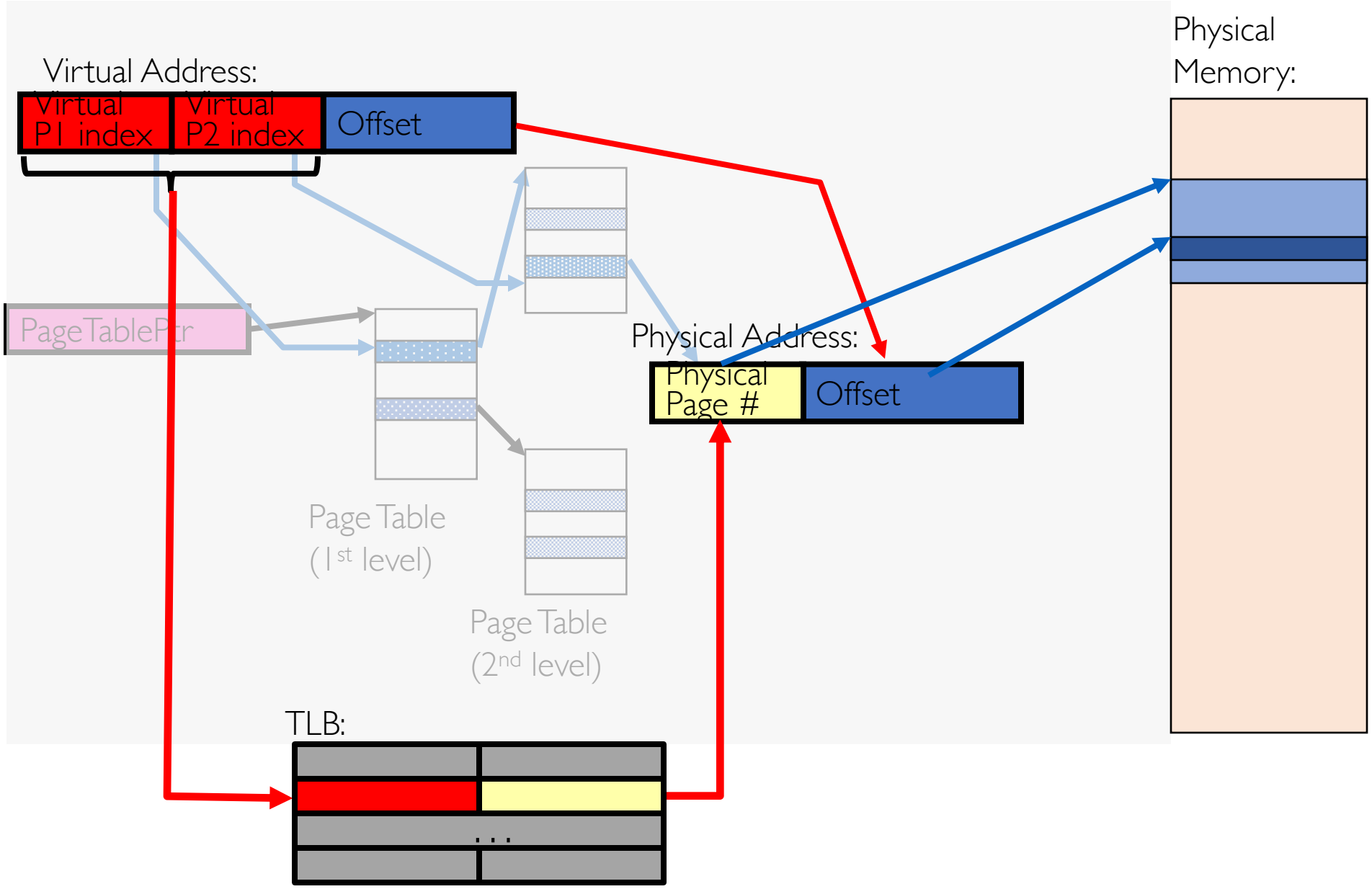
b.缺页中断可能是由于访问的虚拟地址没有相应的物理页映射

c.缺页中断处理程序中可能会修改页表

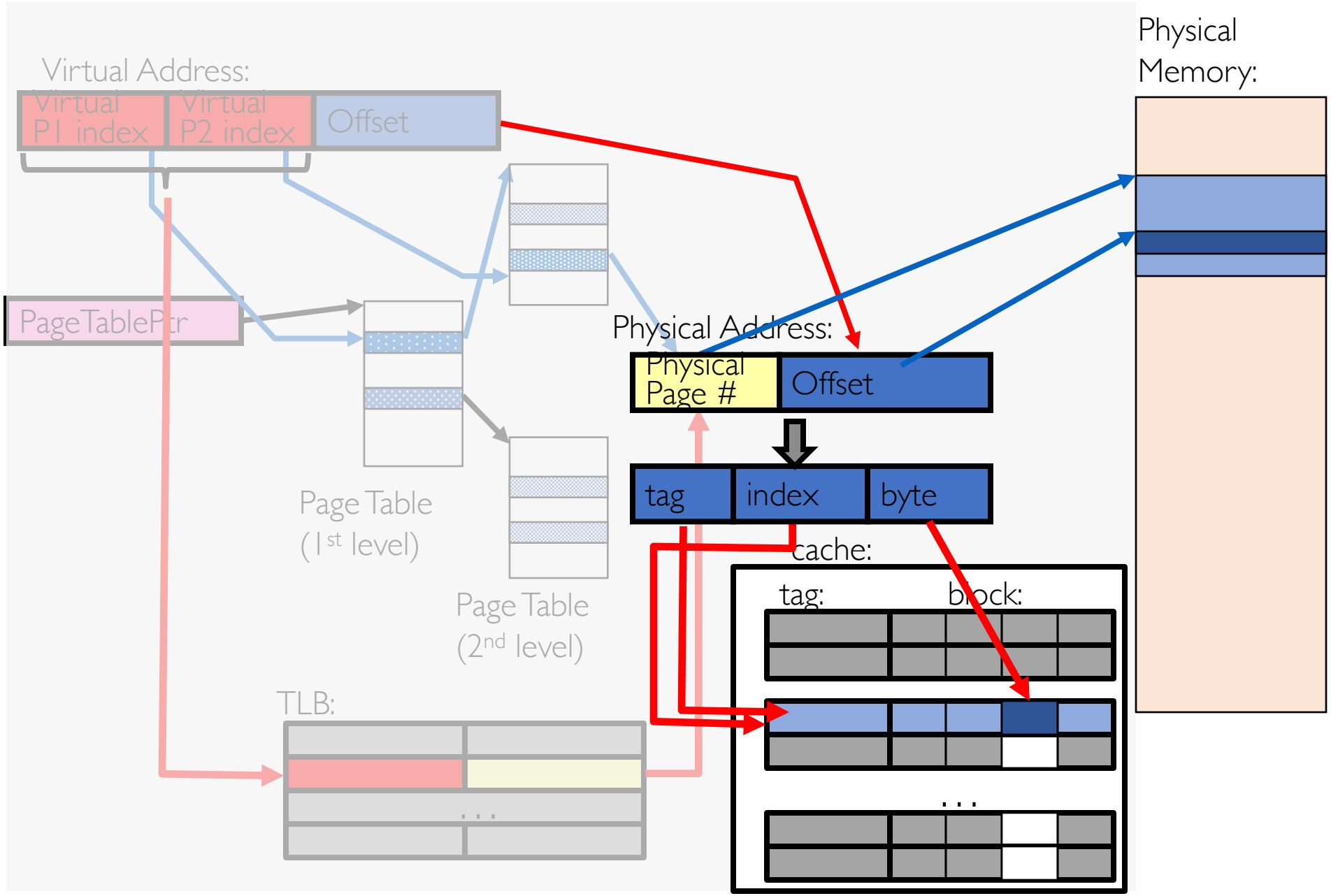
# Recap: Putting Everything Together: Address Translation



# Recap: Putting Everything Together: TLB



# Recap: Putting Everything Together: Cache



# 内存

1.以下关于缺页中断（page fault）的说法正确的是？

a.TLB hit必定不会缺页中断

b.TLB miss必定导致缺页中断

c.缺页不中断的前提是TLB hit

d.缺页中断产生的前提是TLB Miss

e.无

# 内存

1.以下关于缺页中断（page fault）的说法正确的是？

a.TLB hit必定不会缺页中断

b.TLB miss必定导致缺页中断

c.缺页不中断的前提是TLB hit

d.缺页中断产生的前提是TLB Miss

a. Write to read-only page is TLB hit

e.无

# 内存

- Page Fault产生的原因有哪些？请列举一些会导致以及不会导致程序崩溃的原因

# 缓存

1.以下关于缓存Cache（指内存的缓存，非TLB）的说法，正确的是？

a.完全基于虚拟地址查找相应的数据块

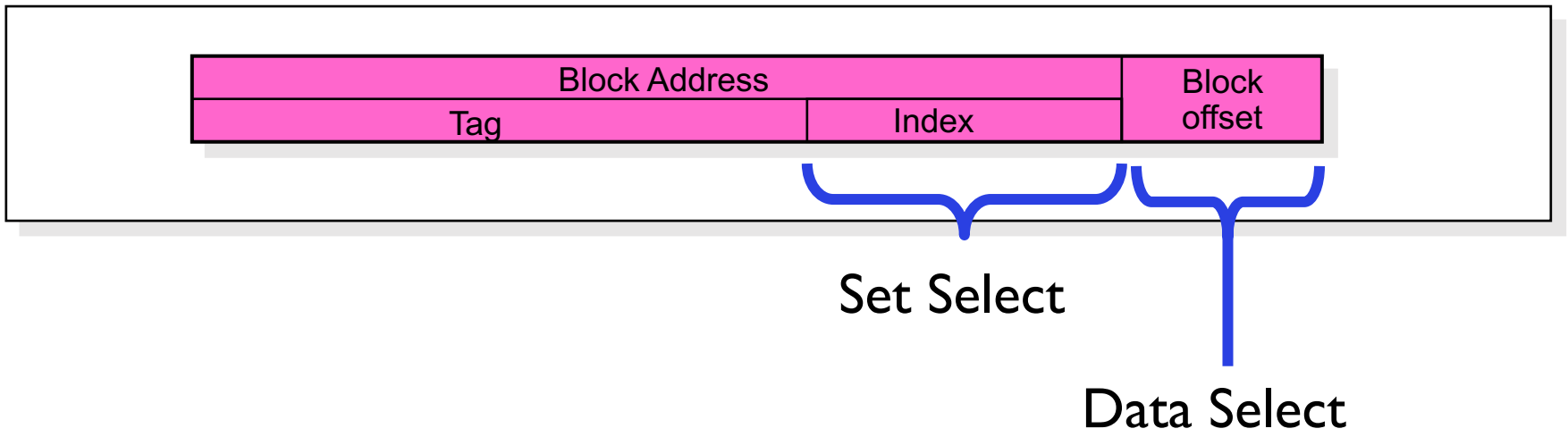
b.计算机中有多级缓存，距离CPU越近，缓存越小

c.进程切换时，需要清空缓存

d.缓存命中率与应用程序的指令顺序相关

# Memory Cache

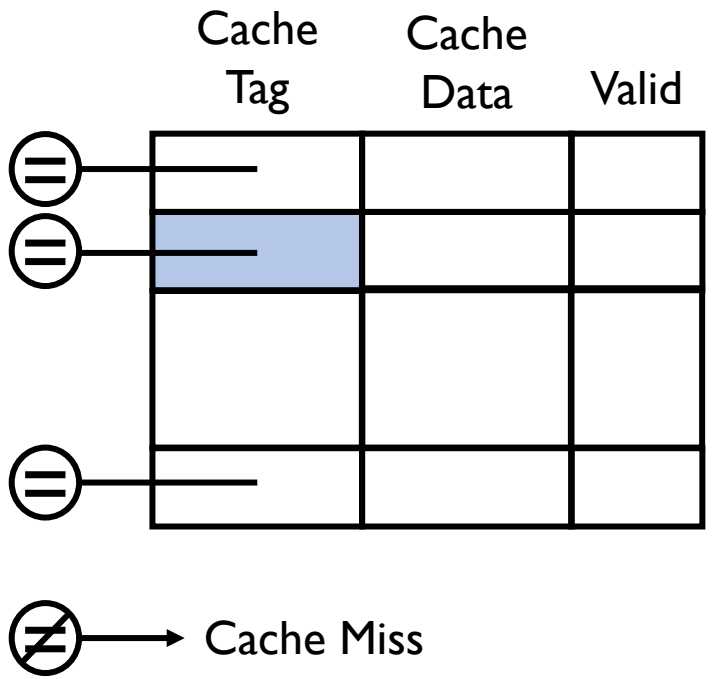
- Block (块) is the minimal unit of caching
  - Often larger than 1 word/byte to exploit the spatial locality
  - Shall not be neither too large or too small. Why?
  - Modern Intel processors use 64B
- Address fields for cache lookup



# Cache Lookup

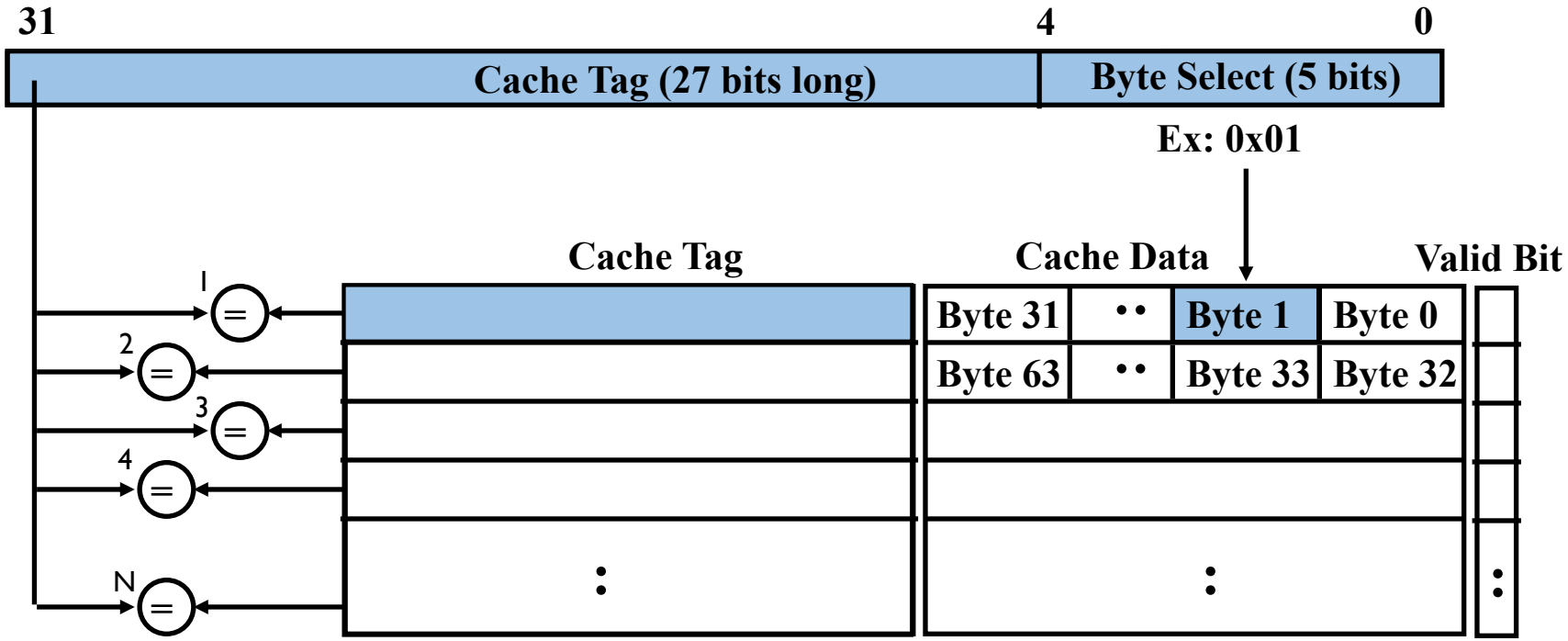
- Fully associative (全关联、完全关联): each address can be stored anywhere in the cache table
- Direct mapped (直接映射): each address can be stored in one location in the cache table
- N-way set associative (N路组关联): each address can be stored in one of N cache sets

- Tradeoffs: lookup speed and cache hit rate



# Fully Associative

- Compare the cache tag on each cache line
- Example: Block Size=32B blocks
  - We need Nx 27-bit comparators



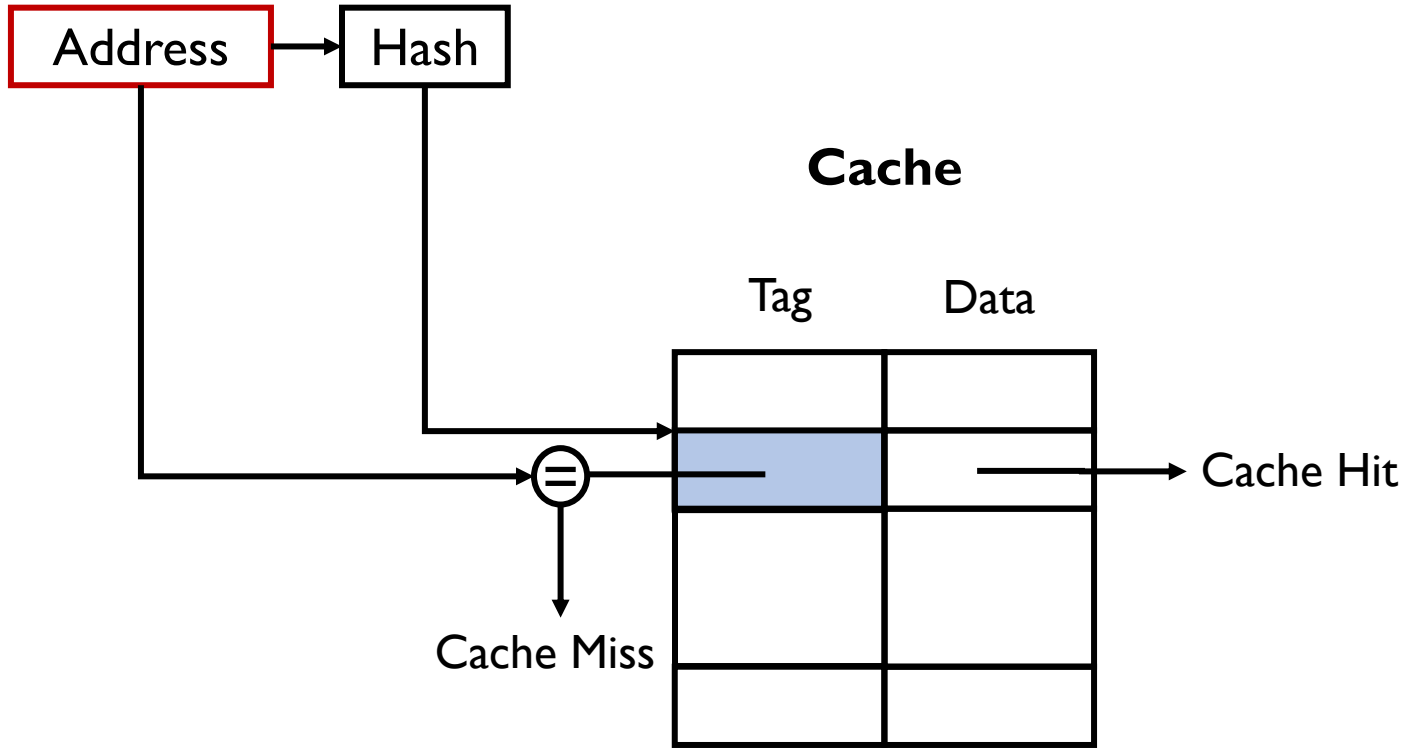
# Fully Associative

- Compare the cache tag on each cache line
- Example: Block Size=32B blocks
  - We need Nx 27-bit comparators

- The drawback: performance degrades with larger cache, because there are more tags to be compared.
  - Solution #1: using larger block, but..

# Direct Mapped

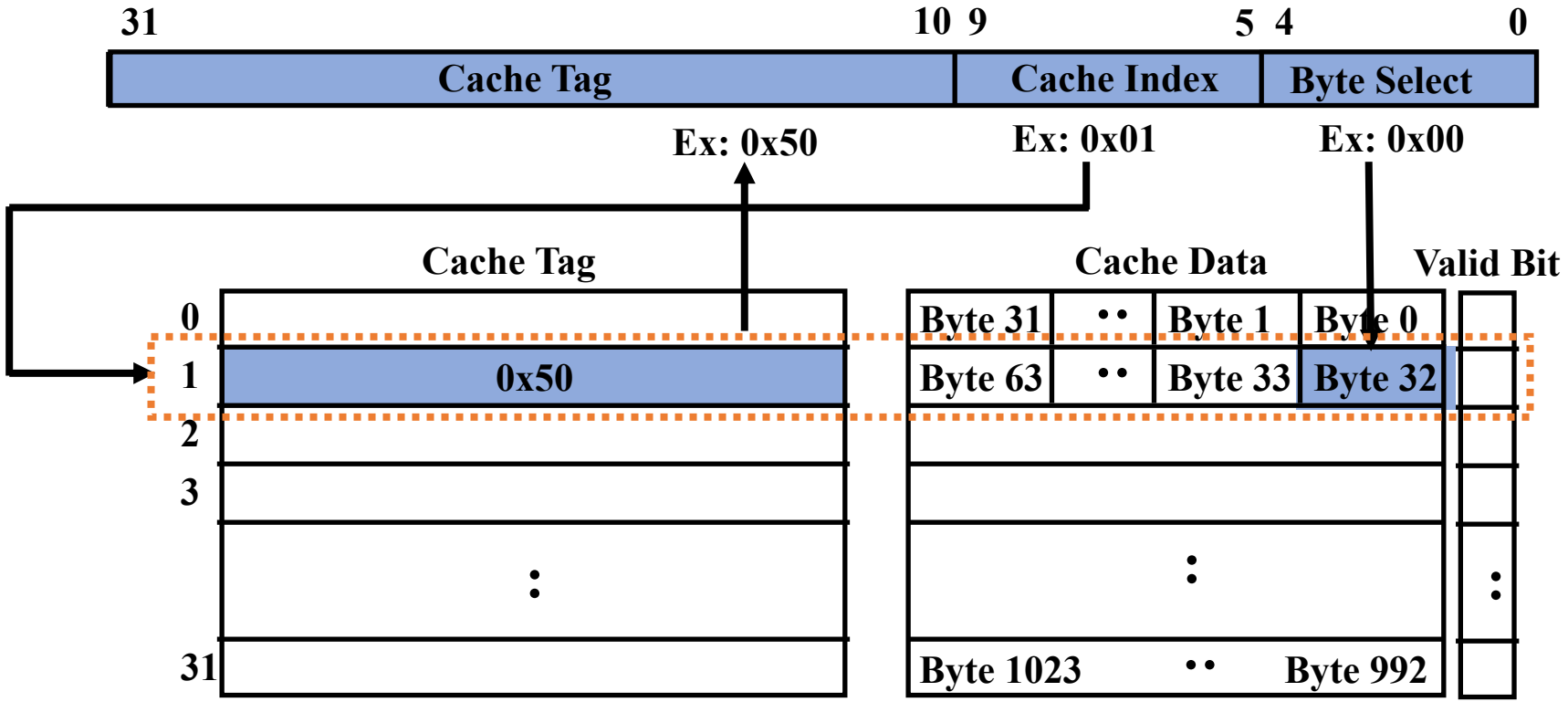
- Map to one specific cache line through a Hash function.
- Verify the address.



# Direct Mapped

- Example: 1 KB Direct Mapped Cache with 32B Blocks
  - Index chooses potential block
  - Tag checked to verify block
  - Byte select chooses byte within block

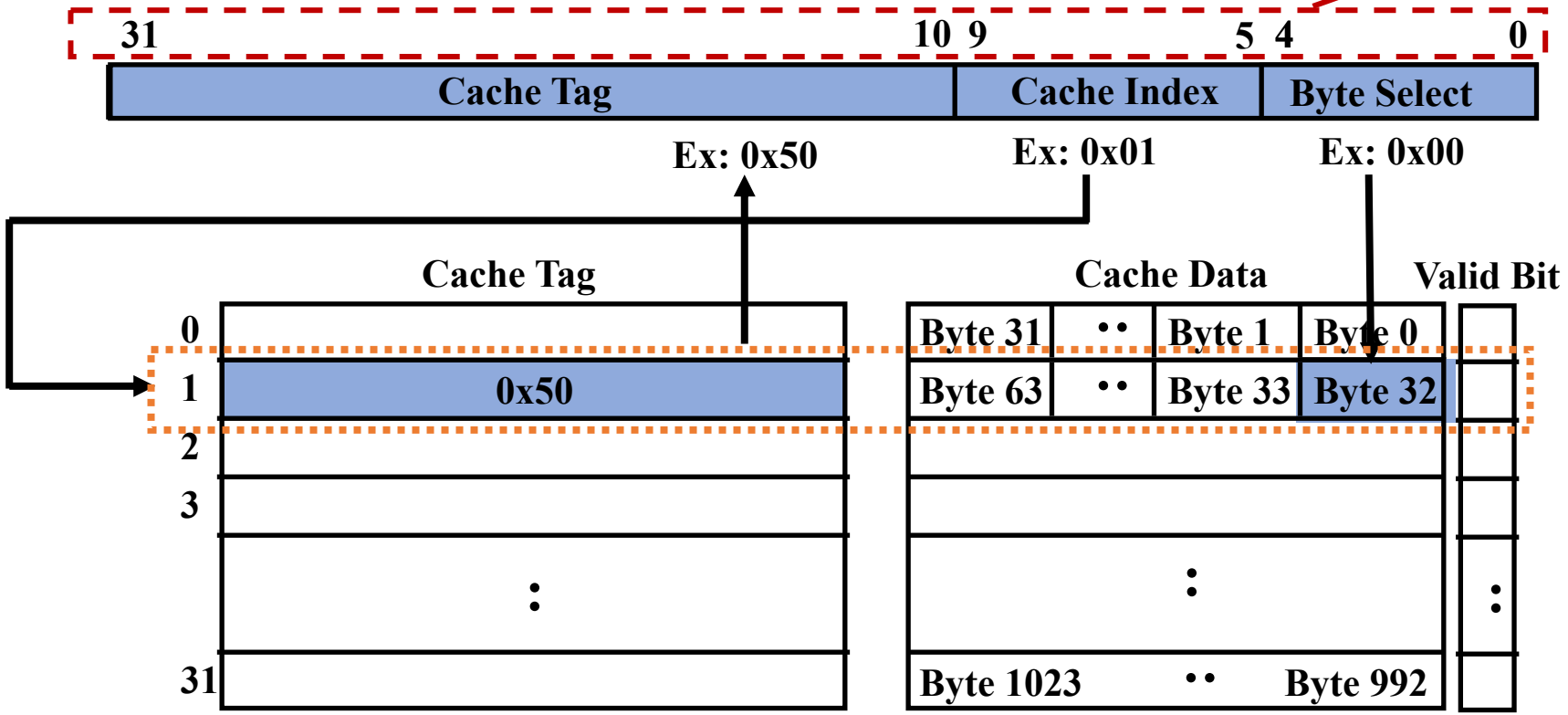
Hash(x) = addr & 1111100000



# Direct Mapped

- Example: 1 KB Direct Mapped Cache with 32B Blocks
  - Index chooses potential block
  - Tag checked to verify block
  - Byte select chooses byte within block

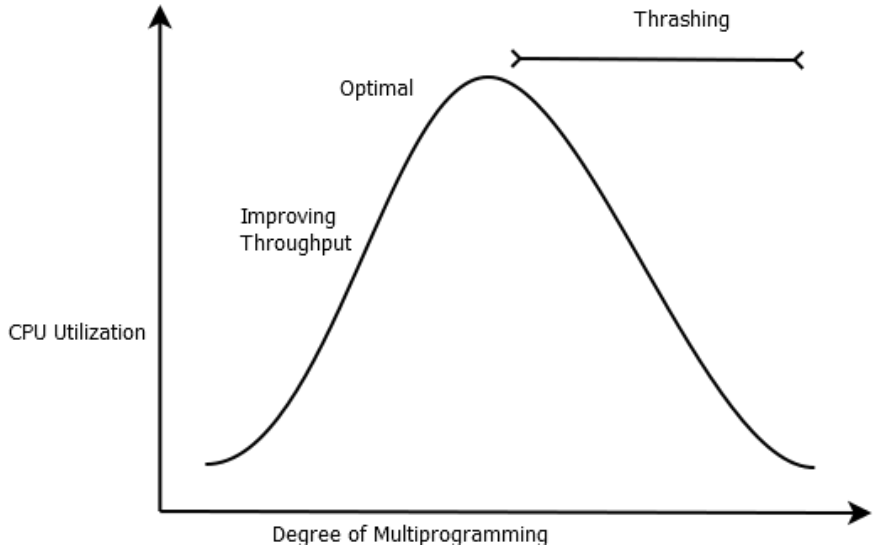
How those numbers are determined?



# Direct Mapped

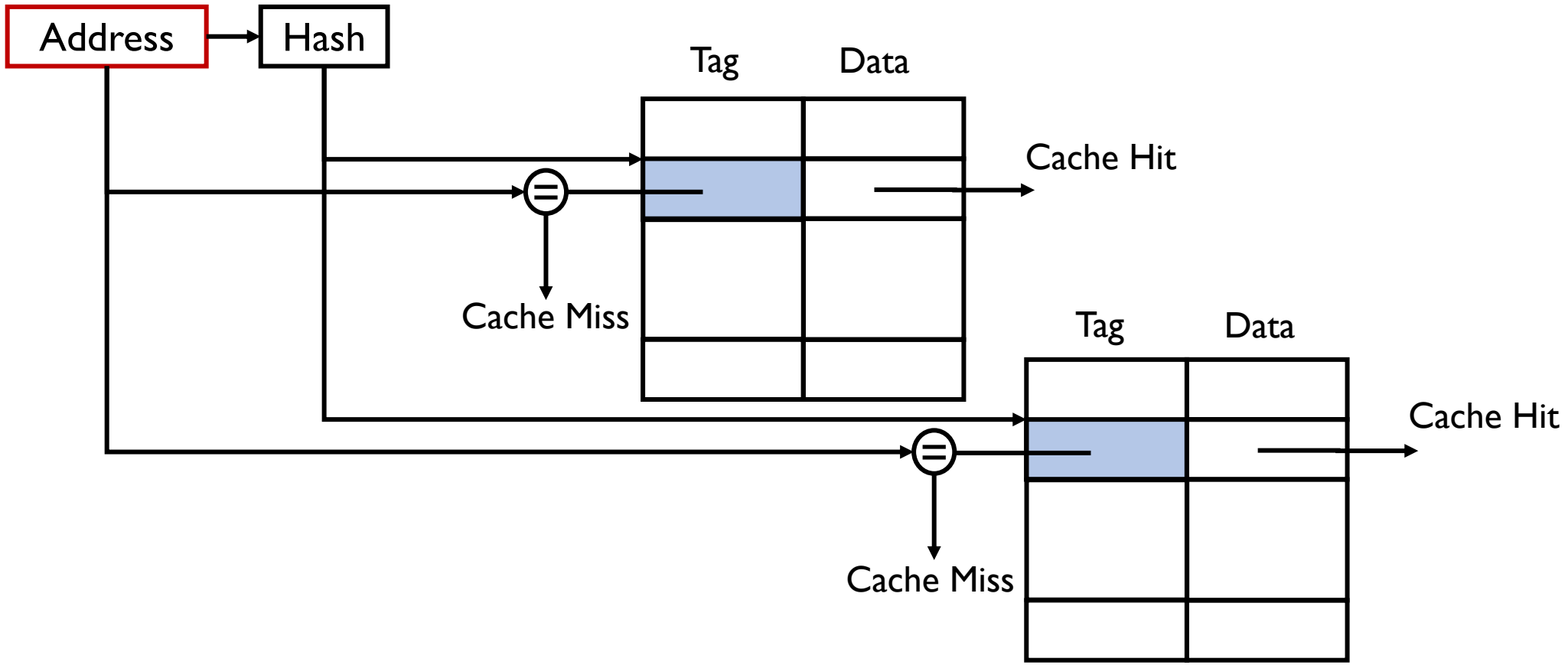
- Example: 1 KB Direct Mapped Cache with 32B Blocks
  - Index chooses potential block
  - Tag checked to verify block
  - Byte select chooses byte within block

- The drawback: low flexibility
  - Thrash (颠簸): frequently using two addresses that map to the same cache entry.



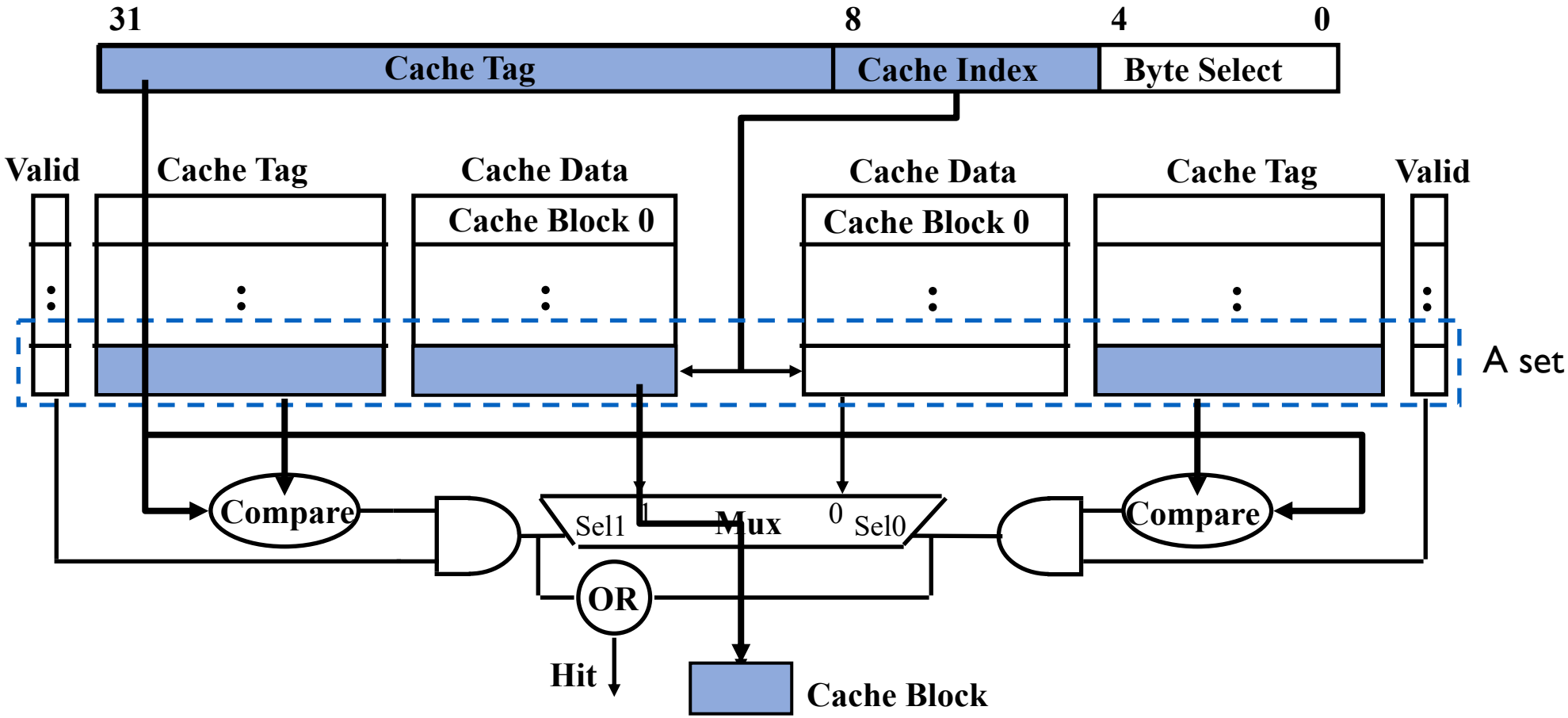
# Set Associative

- N-way Set Associative: N entries per Cache Index
  - N direct mapped caches operates in parallel



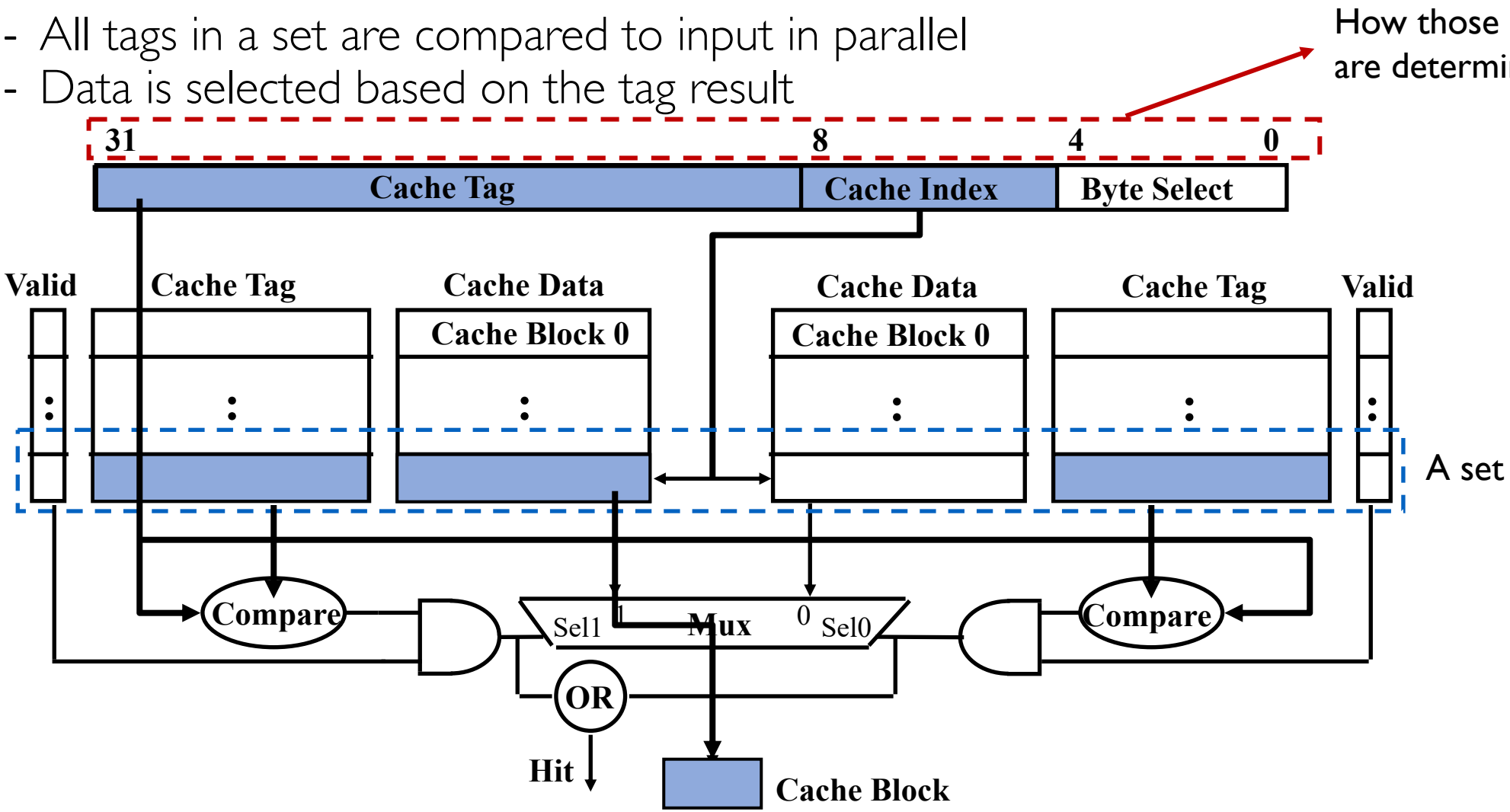
# Set Associative

- Example: two-way set associative cache
  - Cache Index selects a "set" from the cache
  - All tags in a set are compared to input in parallel
  - Data is selected based on the tag result



# Set Associative

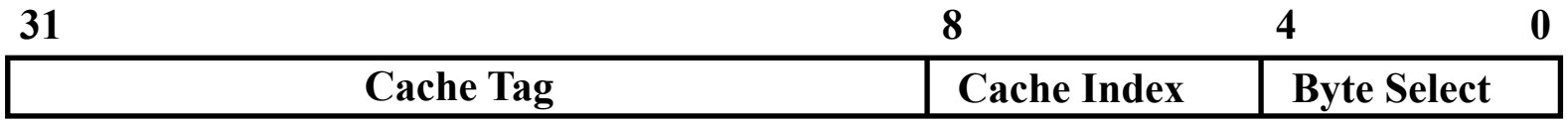
- Example: two-way set associative cache
  - Cache Index selects a "set" from the cache
  - All tags in a set are compared to input in parallel
  - Data is selected based on the tag result



# Set Associative

- Example: two-way set associative cache
  - Cache Index selects a "set" from the cache
  - All tags in a set are compared to input in parallel
  - Data is selected based on the tag result

- N-way set associative is a mix of direct mapped and fully associative
  - When n = ? It becomes directed mapped
  - When n = ? It becomes fully associative

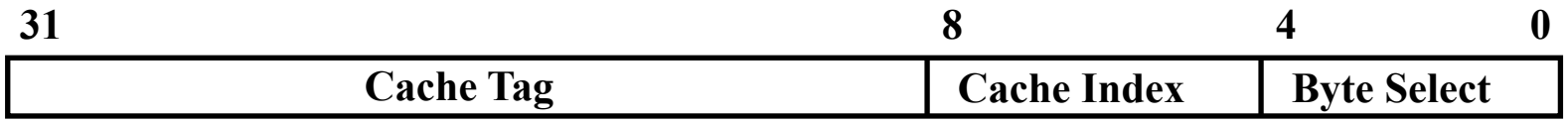


# Set Associative

- Example: two-way set associative cache
  - Cache Index selects a "set" from the cache
  - All tags in a set are compared to input in parallel
  - Data is selected based on the tag result

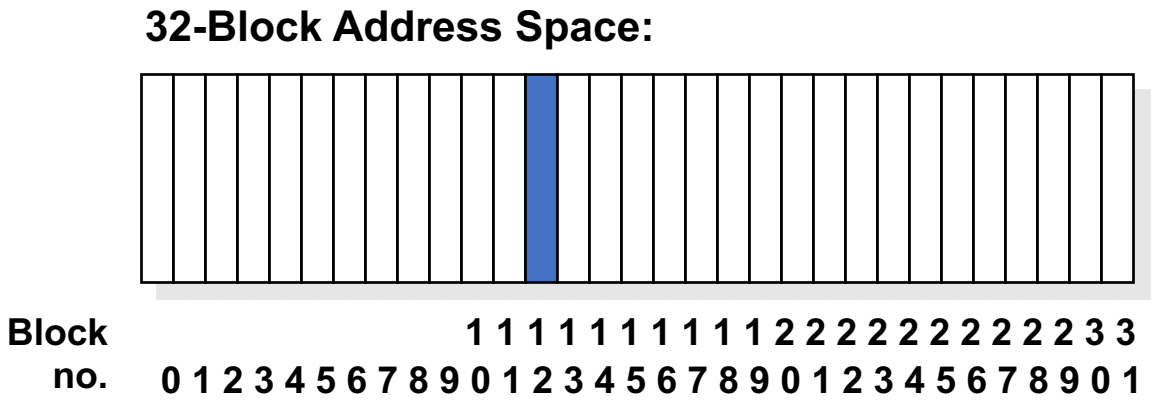
- N-way set associative is a mix of direct mapped and fully associative
  - When n = ? It becomes directed mapped
  - When n = ? It becomes fully associative

- Why use the lower bits for index, higher bits for tag?



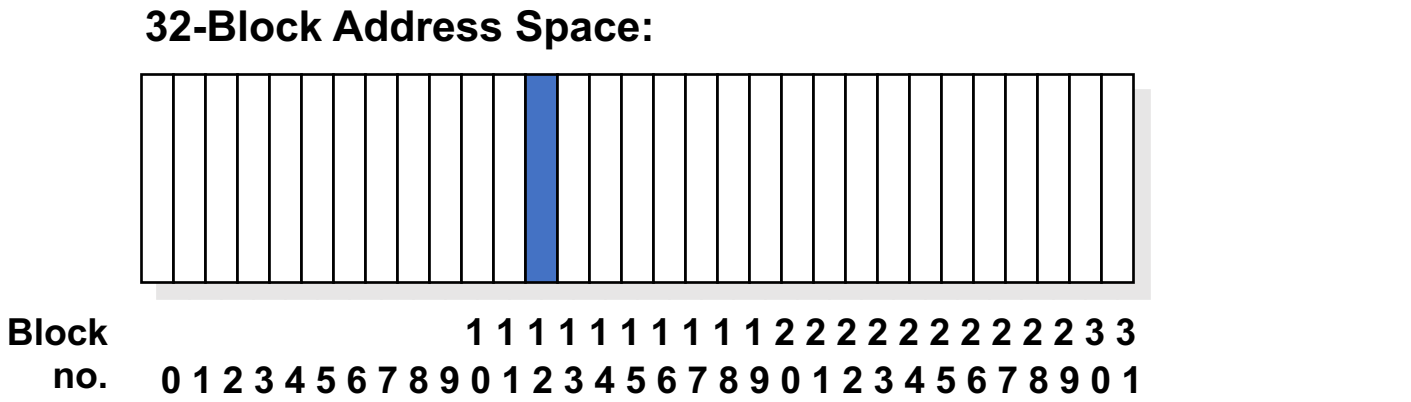
# Where does a Block Get Placed in a Cache?

- Example: Block 12 placed in 8 block cache??



# Where does a Block Get Placed in a Cache?

- Example: Block 12 placed in 8 block cache



# 缓存

1. 以下关于缓存Cache（指内存的缓存，非TLB）的说法，正确的是？

a. 完全基于虚拟地址查找相应的数据块

b. 计算机中有多级缓存，距离CPU越近，缓存越小

c. 进程切换时，需要清空缓存

   a. TLB？

d. 缓存命中率与应用程序的指令顺序相关

# 同步

1.以下关于条件变量（Condition Variable, cv）的说法，正确的是？

a.Wait(&lock)函数必须在持有lock锁的过程中才能调用

b.Wait()函数会释放锁

c.Signal()函数会释放锁

d.Wait()函数返回时，有可能不持有lock锁，程序需要重新获取锁

e.Broadcast()函数至少会成功醒一个等待线程

# Lock

- Suppose we have some sort of implementation of a lock
  - `lock.Acquire()` – wait until lock is free, then grab
  - `lock.Release()` – Unlock, waking up anyone waiting
  - These must be atomic operations – if two threads are waiting for the lock and both see it's free, only one succeeds to grab the lock
- 3 formal properties
  - Mutual exclusion: at most one thread holds the lock
  - Progress: if no thread holds the lock and any thread attempts to acquire the lock, then eventually some thread succeeds in acquiring the lock
  - Bounded waiting: if thread T attempts to acquire a lock, then there exists a bound on the number of times other threads can successfully acquire the lock before T does
    - ❑ Yet, it does not promise that waiting threads acquire the lock in FIFO order.

# Condition Variable

- Condition Variable (条件变量): a queue of threads waiting for something *inside* a critical section
  - Key idea: allow sleeping inside critical section by atomically releasing lock at time we go to sleep
- Operations:
  - `Wait(&lock)`: Atomically release lock and go to sleep. Re-acquire lock later, before returning.
  - `Signal()`: Wake up one waiter, if any
  - `Broadcast()`: Wake up all waiters
  - Differentiate them from UNIX wait and signal

# Condition Variable Example

- Condition Variable (条件变量): a queue of threads waiting for something *inside* a critical section
  - Key idea: allow sleeping inside critical section by atomically releasing lock at time we go to sleep
- A common pattern:

```
FuncA_wait() {
    lock.acquire();
    // read/write shared state here
    while (!testOnSharedState())
        cv.wait(&lock);
    assert(testOnSharedState());
    lock.release();
}
```

```
FuncB_signal() {
    lock.acquire();
    // read/write shared state here
    // If state has changed that allows another thread to make progress, call signal or broadcast
    cv.signal();
    lock.release();
}
```

# Condition Variable Example

- A concrete example of bounded queue implementation (or producer-consumer, 生产者消费者)

```
class bounded_queue {
    Lock lock;
    CV itemAdded;
    CV itemRemoved;
    void insert(int item);
    int remove();
}
```

```
void bounded_queue::insert(int item) {
    lock.acquire();
    while (queue.full()) {
        itemRemoved.wait(&lock);
    }
    add_item(item);
    itemAdded.signal();
    lock.release();
}
```

How to implement remove()?

# Condition Variable Example

- A concrete example of bounded queue implementation (or producer-consumer, 生产者消费者)

- Two key principles
  - CV is always used with lock acquired
  - CV is put in a while loop. Why?

```
void bounded_queue::insert(int item) {
    lock.acquire();
    while (queue.full()) {
        itemRemoved.wait(&lock);
    }
    add_item(item);
    itemAdded.signal();
    lock.release();
}
```

# 同步

1.以下关于条件变量（Condition Variable, cv）的说法，正确的是？

a.Wait(&lock)函数必须在持有lock锁的过程中才能调用

b.Wait()函数会释放锁

c.Signal()函数会释放锁

d.Wait()函数返回时，有可能不持有lock锁，程序需要重新获取锁

e.Broadcast()函数至少会成功醒一个等待线程

# 文件系统

1. 以下关于三类常见文件系统：FAT、FFS和NTFS的说法，正确的是？

a. FAT文件系统对大文件的随机读取速率较差；

b. FFS文件系统的采用非对称（深度）树状结构索引的目的是为了同时支持小文件和大文件的高效存储和查找；

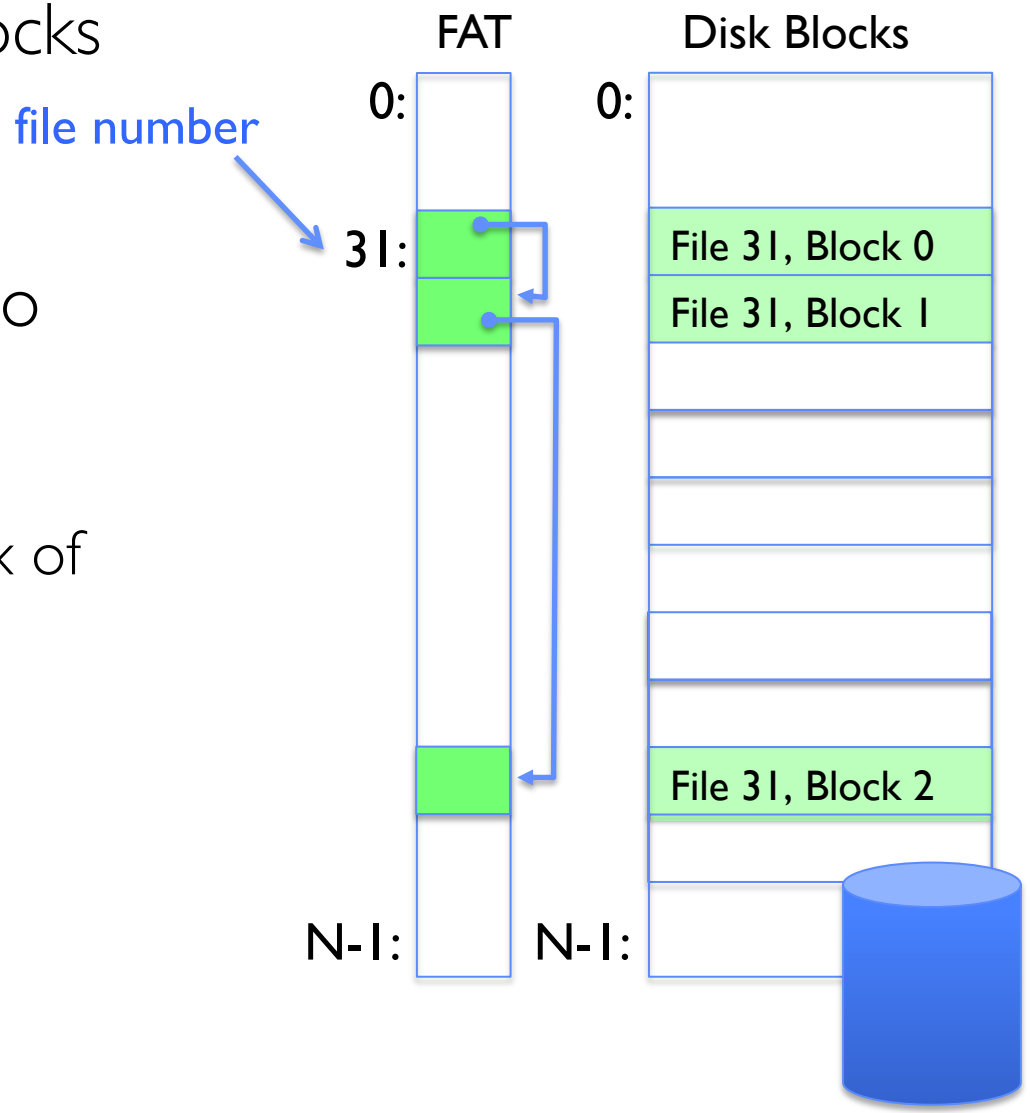
c. NTFS文件系统对小文件最友好，因为可以直接存储数据在MFT中，而其他两个文件系统都需要索引；

d. FAT文件系统可以支持稀疏化文件表示；

e. FAT文件系统使用next fit分配算法；FFS文件系统使用first fit分配算法；NTFS文件系统使用best fit分配算法

# FAT (File Allocation Table)

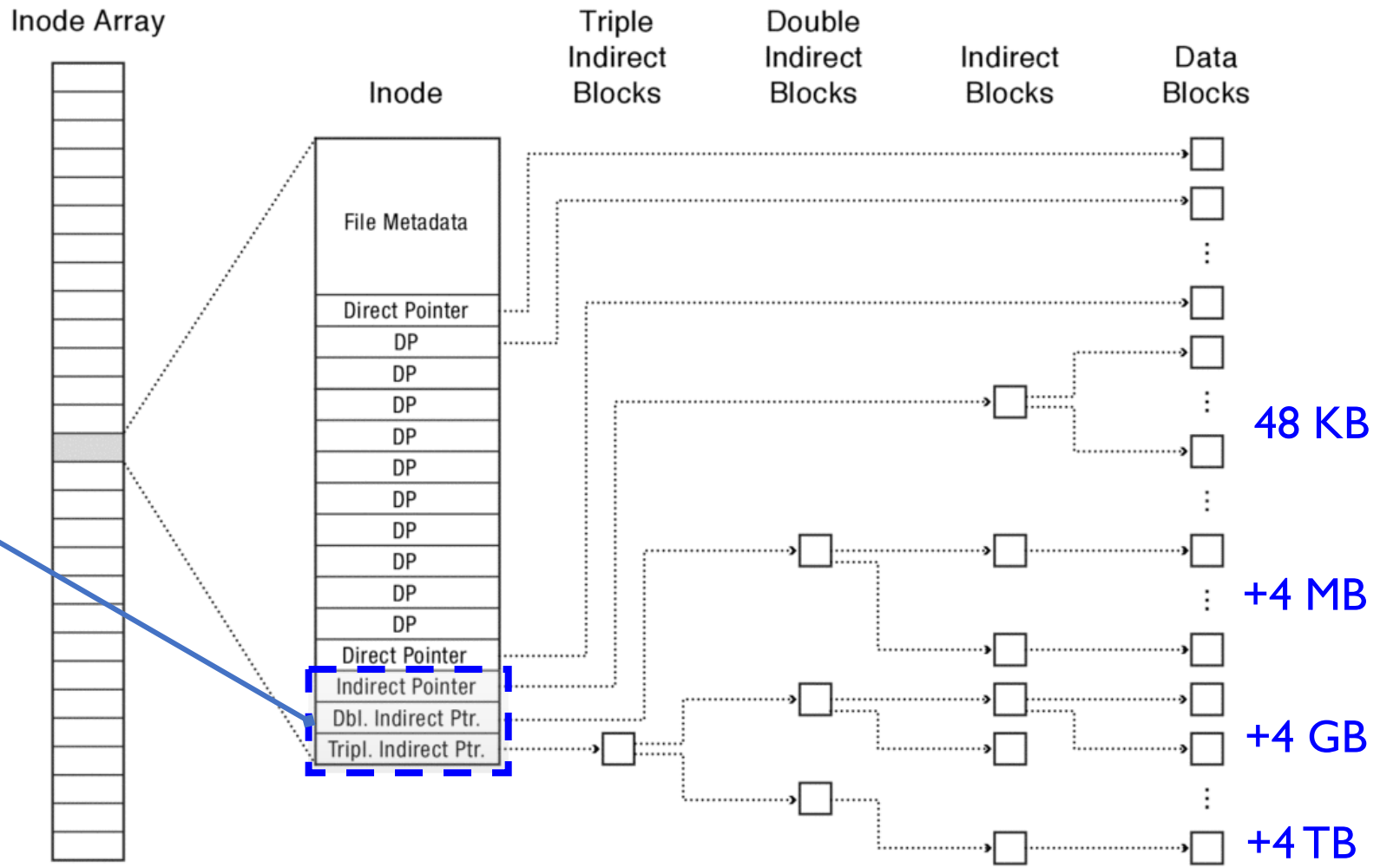
- FAT is a linked list as 1-1 map with blocks
  - Represented as a list of 32-bit entries
  - Older versions use fewer bits
- Each entry in FAT contains a pointer to the next FAT entry of the same file
  - Or a special END_OF_FILE value.
  - The file number is the 1st (or root) index of the block list for the file
- For File No. #i, its
  - 1st data block index: i
  - 2nd data block index: *(FAT[i])
  - 3rd data block index: *(*(FAT[i]))
  - ..



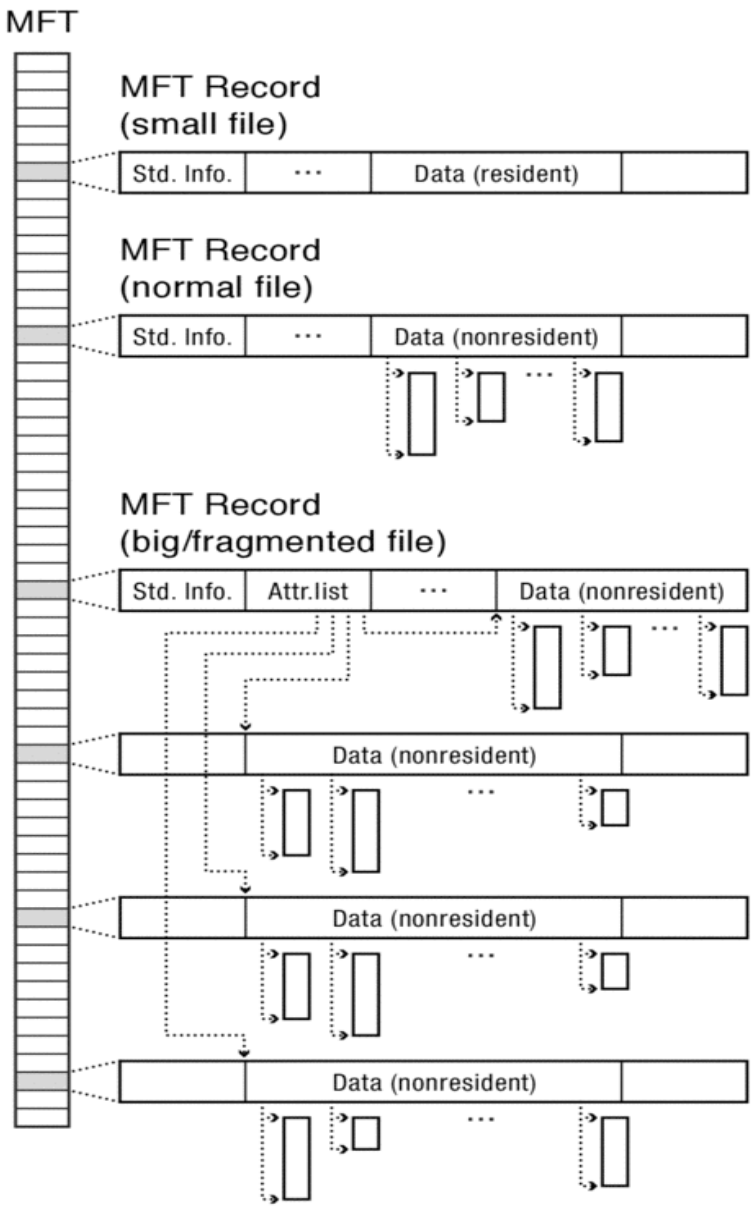
# FFS

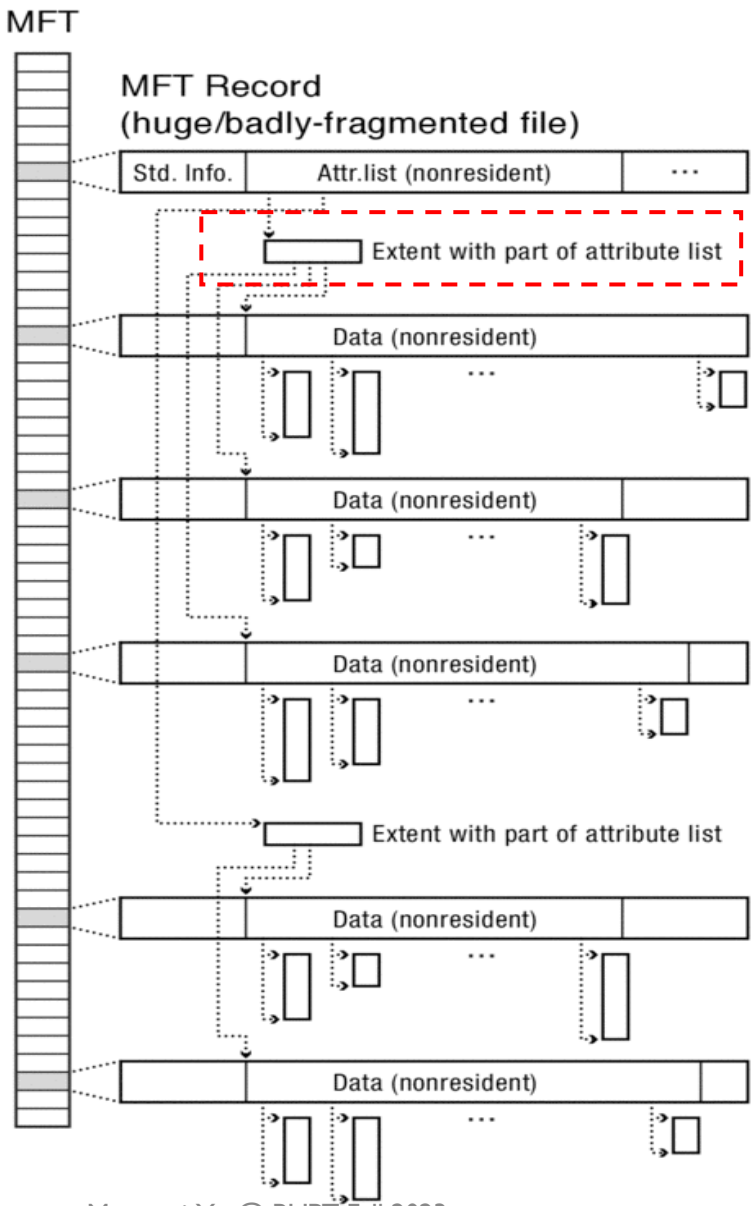
Indirect pointers
- point to a disk block containing only pointers

4 kB blocks => 1024 ptrs
Indirect Pointer (一级间接索引)
=> 4 MB
Double Indirect Pointer (二级..)
=> 4 GB
Triple Indirect Pointer (三级..)
=> 4 TB



# NTFS




Even the attribute list becomes nonresident!

Why it is possible??

# 文件系统

1. 以下关于三类常见文件系统：FAT、FFS和NTFS的说法，正确的是？

a. FAT文件系统对大文件的随机读取速率较差；

b. FFS文件系统的采用非对称（深度）树状结构索引的目的是为了同时支持小文件和大文件的高效存储和查找；

c. NTFS文件系统对小文件最友好，因为可以直接存储数据在MFT中，而其他两个文件系统都需要索引；

d. FAT文件系统可以支持稀疏化文件表示；

e. FAT文件系统使用next fit分配算法；FFS文件系统使用first fit分配算法；NTFS文件系统使用best fit分配算法

# 文件系统

1.以下关于使用奇偶校验方法实现的RAID 5磁盘冗余技术的说法，正确的是？

a.相较于RAID 1完全镜像的方式，节省了磁盘空间

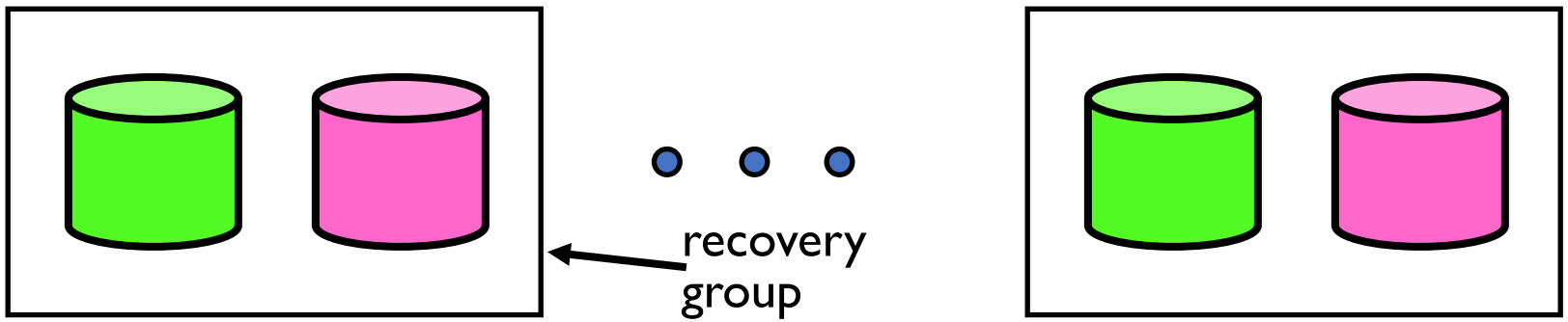
b.当超过1个磁盘损坏时，无法恢复数据

c.当只有1个磁盘损坏，但未知是哪个磁盘时，无法恢复数据

d.奇偶校验值不放在同一个磁盘上可以防止该磁盘成为I/O瓶颈

# RAID 1: Disk Mirroring/Shadowing

- Each disk is fully duplicated onto its "shadow"
  - For high I/O rate, high availability environments
  - Most expensive solution: 100% capacity overhead
- Bandwidth sacrificed on write:
  - Logical write = two physical writes
  - Highest bandwidth when disk heads and rotation fully synchronized (hard to do)
- Reads may be optimized
  - Can have two independent reads to same data
- Recovery:
  - Disk failure ⇒ replace disk and copy data to new disk
  - Hot Spare: idle disk already attached to system to be used for immediate replacement

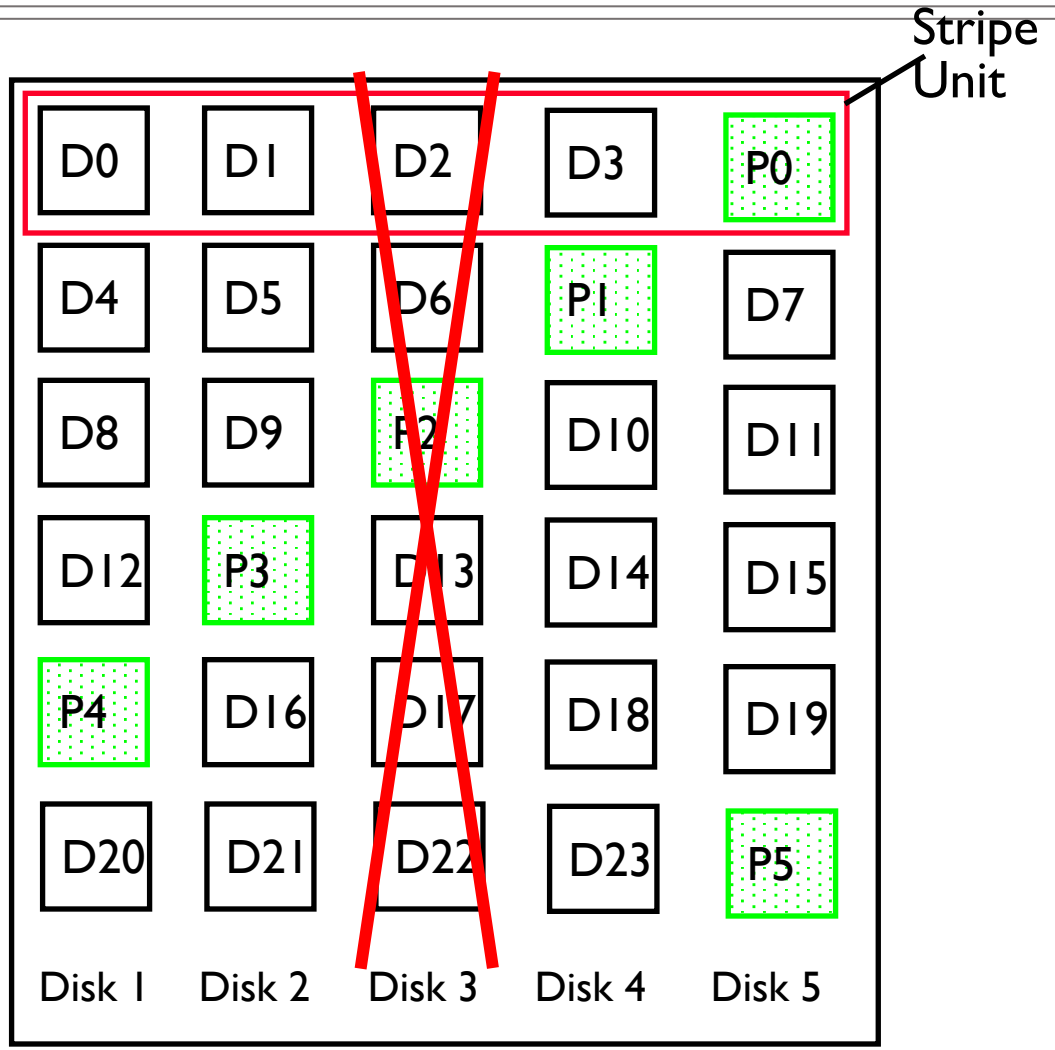


# Magic XOR (异或)

- XOR (^), or eXclusive OR, is a bitwise operator that returns true (1) for odd frequencies of 1. The XOR truth table is as follows:
  - 1 ^ 1 = 0
  - 1 ^ 0 = 1
  - 0 ^ 1 = 1
  - 0 ^ 0 = 0
- XOR is commutative.
  - a^b = b^a.
- XOR is associative.
  - a^(b^c) = (a^b)^c = (a^c)^b.
- XOR is self-inverse.
  - Any number XOR'ed with itself evaluates to 0.
- a^a = 0.
  - 0 is the identity element for XOR.
- This means, any number XOR'ed with 0 remains unchanged.
  - a^0 = a.

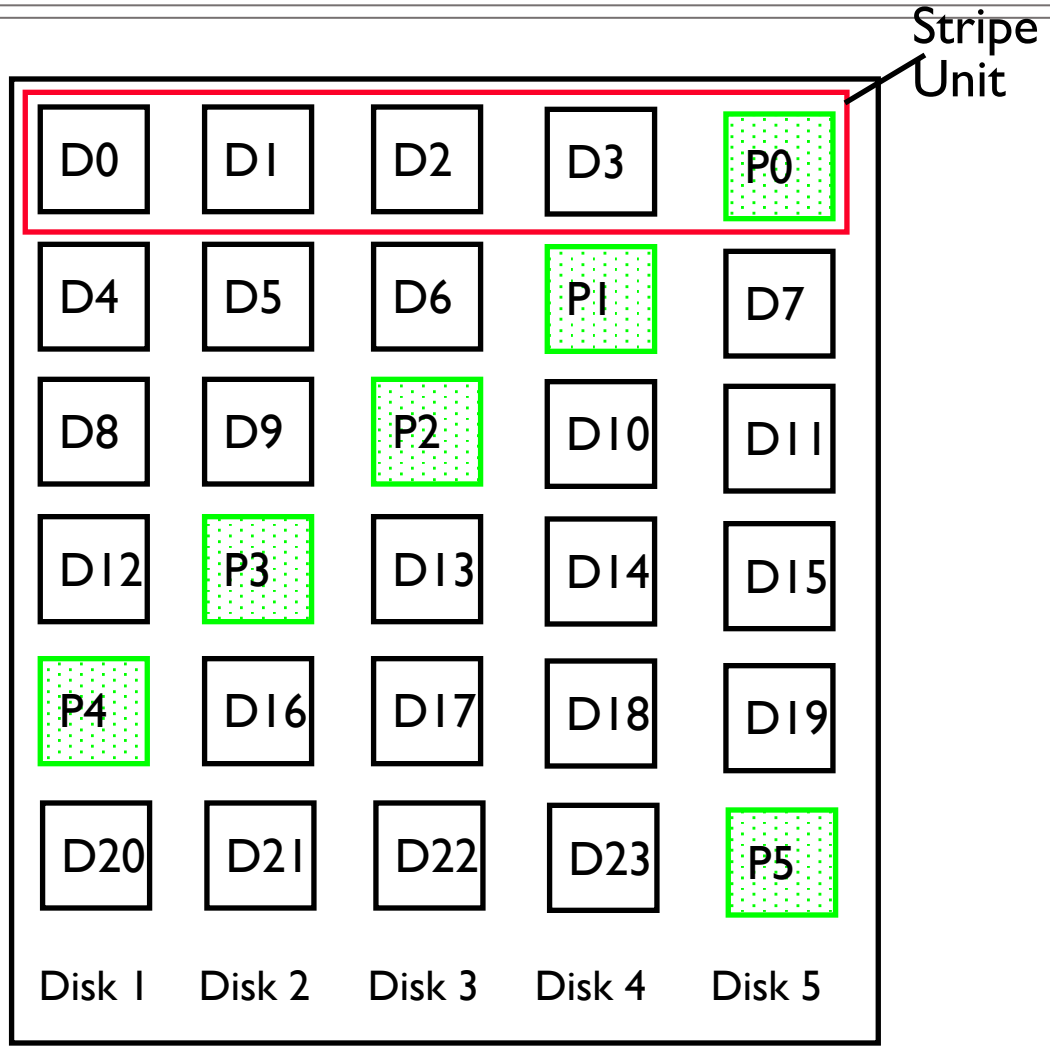
# RAID 5+: High I/O Rate Parity

- Data stripped across multiple disks
  - Successive blocks stored on successive (non-parity) disks
  - Increased bandwidth over single disk
- Parity block (in green) constructed by XORing (异或) data blocks in stripe
  - P0=D0⊕D1⊕D2⊕D3
  - Can destroy any one disk and still reconstruct data
  - Suppose Disk 3 fails, then can reconstruct: D2=D0⊕D1⊕D3⊕P0



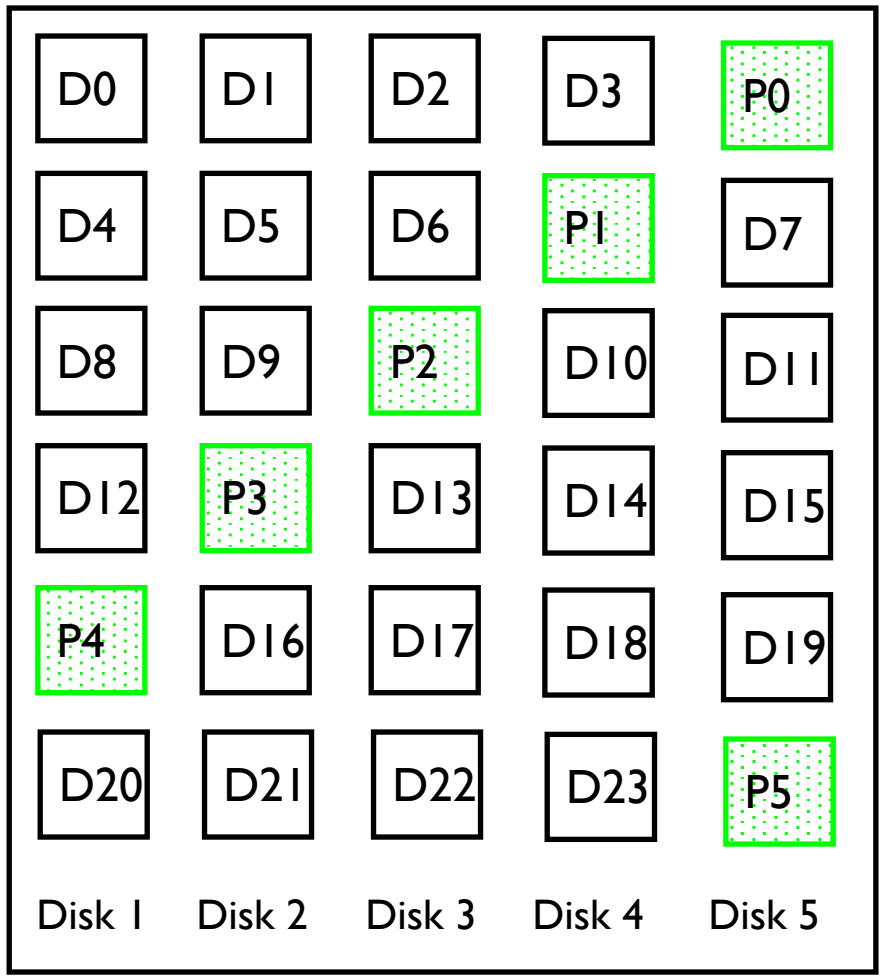
# RAID 5+: High I/O Rate Parity

- Rotating parity (奇偶校验)
  - The parity needs to be updated more often than normal data blocks.
- Striping data
  - Balance parallelism vs. sequential access efficiency
- RAID 5 can recover the failed disk only if (i) only one disk fails and (ii) the failed disk is known.



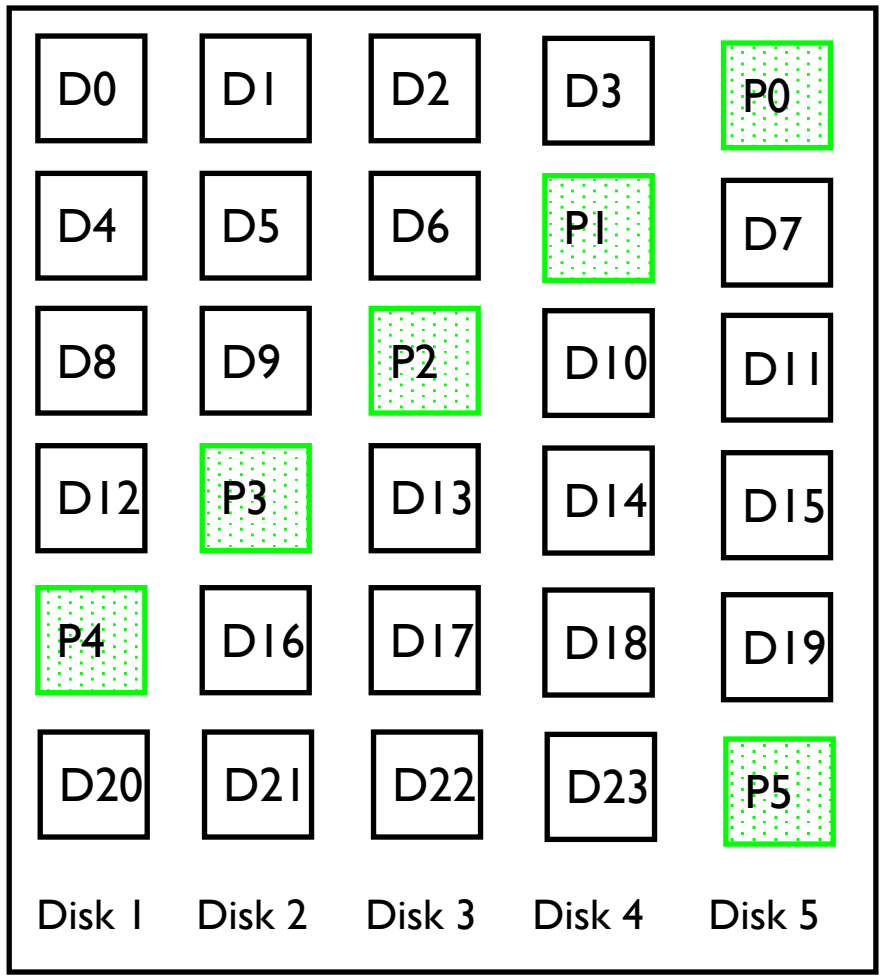
# RAID 5+: High I/O Rate Parity

- What I/O operations would occur if we want to update D21 in this figure?



# RAID 5+: High I/O Rate Parity

- What I/O operations would occur if we want to update D21 in this figure?
  - Read D21(old)
  - Read P5(old)
  - Compute tmp=P5(old)⊕D21(old)
  - Compute P5(new)=tmp ⊕D21(new)
  - Write D21(new)
  - Write P5(new)



# 文件系统

1.以下关于使用奇偶校验方法实现的RAID 5磁盘冗余技术的说法，正确的是？

a.相较于RAID 1完全镜像的方式，节省了磁盘空间

b.当超过1个磁盘损坏时，无法恢复数据

c.当只有1个磁盘损坏，但未知是哪个磁盘时，无法恢复数据

d.奇偶校验值不放在同一个磁盘上可以防止该磁盘成为I/O瓶颈

# 虚拟机

1. 以下关于虚拟机（virtual machine）的说法，正确的是？

a. 虚拟机中的操作系统内核（guest kernel）运行在内核态，可以执行特权指令

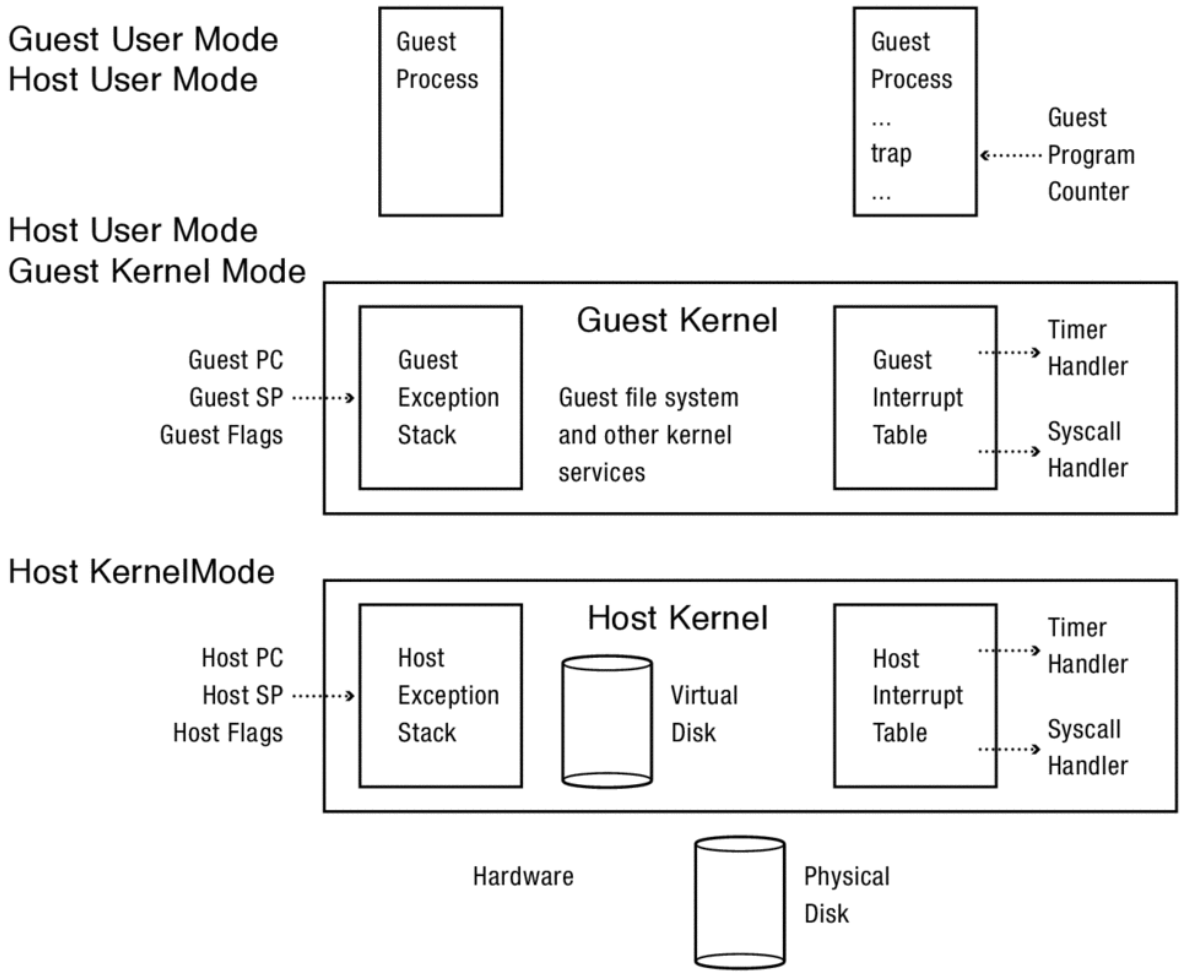
b. 当中断发生时，硬件决定将它发送给host kernel还是guest kernel

c. guest kernel通过iret指令从内核中返回到guest用户程序时，会陷入到host kernel中

d. 使用shadow page tables实现虚拟机中的内存映射，需要host kernel追踪guest kernel中对页表的修改

# Case Study: Mode Transfer

- When guest user process issues a syscall
  1. Traps into the host kernel's syscall handler (why?)
  2. The host kernel saves the $PC, $FLAGS, and user stack pointer on the interrupt stack of the guest kernel.
  3. The host kernel transfers control to the guest kernel, which runs with user-mode privilege.
  4. The guest kernel performs the system call — saving user state and checking arguments.
  5. When the guest kernel attempts to return from the system call back to guest user process (iret), this causes a processor exception, dropping back into the host kernel.
  6. The host kernel can then restore the state of the user process, running at user level, as if the guest OS had been able to return there directly.



# 虚拟机

1. 以下关于虚拟机（virtual machine）的说法，正确的是？

a. 虚拟机中的操作系统内核（guest kernel）运行在内核态，可以执行特权指令

b. 当中断发生时，硬件决定将它发送给host kernel还是guest kernel

c. guest kernel通过iret指令从内核中返回到guest用户程序时，会陷入到host kernel中

d. 使用shadow page tables实现虚拟机中的内存映射，需要host kernel追踪guest kernel中对页表的修改

# 用户态内核态

1.（单选）下面哪个操作可能不会导致用户态切换到内核态？

a.缺页异常Page fault

b.调用libc中的字符串函数

c.除零

d.用户程序打开磁盘上的一个文件

# 用户态内核态

1.（单选）下面哪个操作可能不会导致用户态切换到内核态？

a.缺页异常Page fault

b.调用libc中的字符串函数

c.除零

d.用户程序打开磁盘上的一个文件

# 简答题

# 简答题

1. 请列举3个操作系统。

2. 请简述异常（Exception）和中断（Interrupt）的区别，并分别举一例具体的异常和中断。

3. 以下代码会打印多少次OS

```
if (fork() || fork())
fork();
printf("OS ");
```

# 简答题

1. 操作系统在处理系统调用时，必须将传入的参数拷贝到内核空间后再使用，为什么？请举例说明不这样做的后果。

2. TLB作为一种缓存，请简述其利用了怎样的局部性（locality）？

# 简答题

1. 在32-bit操作系统中，使用4-路组相连缓存，缓存中共有128个数据块空间，块（block）大小为64 bytes
   a. Cache Tag、Cache Index、Byte Select（块内偏移）分别占据多少个比特数？
   b. 地址x和y满足怎样的关系时，他们可能会导致缓存冲突？

# 简答题

1. 在32-bit操作系统中，使用4-路组相连缓存，缓存中共有128个数据块空间，块（block）大小为64 bytes
   a. Cache Tag、Cache Index、Byte Select（块内偏移）分别占据多少个比特数？
   b. 地址x和y满足怎样的关系时，他们可能会导致缓存冲突？

正确答案：分别占据21，5，6个比特。|x/64-y/64|%32==0。

# 简答题

1. 线程生命周期中，有5种状态：Init，Runnable（Ready）、Waiting、Running、Finished（dead），请分别举一例操作/事件可能导致以下状态之间的转换：
   a. Runnable -> Running
   b. Running -> Waiting
   c. Waiting -> Runnable

# 简答题

1. 线程生命周期中，有5种状态：Init，Runnable（Ready）、Waiting、Running、Finished（dead），请分别举一例操作/事件可能导致以下状态之间的转换：

   a.Runnable -> Running

   b.Running -> Waiting

   c.Waiting -> Runnable

参考答案：a可能是调度器将该线程调度到cpu上运行；b可能是等待某个锁、等待I/O完成、pthread_join()等；c可能是获得了锁、收到了信号、I/O结束等。

# 简答题

1. 某全关联缓存大小为4个blocks，程序依次访问内存中第0，1，2，3，4，2，4，1，3，2，0号block数据，在使用先入先出（FIFO）和最近未被访问（LRU）缓存替换策略下，分别产生了多少次cache miss？

# 简答题

1. 某全关联缓存大小为4个blocks，程序依次访问内存中第0，1，2，3，4，2，4，1，3，2，0号block数据，在使用先入先出（FIFO）和最近未被访问（LRU）缓存替换策略下，分别产生了多少次cache miss？

参考答案：FIFO：6 LRU：6

# 简答题

1. 在多核线程调度算法中，有一种重要的设计思想是亲和性调度（affinity scheduling），即尽量将同一个线程调度在同一个核上。请简述这样做的目的。

# 简答题

1. 在多核线程调度算法中，有一种重要的设计思想是亲和性调度（affinity scheduling），即尽量将同一个线程调度在同一个核上。请简述这样做的目的。

正确答案：提升线程的TLB和L1 Cache命中率。

# 简答题

1. 在一个物理磁盘中，不考虑queuing和controller的时延，平均寻道时间（seek time）为3ms，磁盘转速为10000RPM（转每分钟），I/O读取速度为8MB/s，扇区大小为1KB。请分别计算（1）平均延迟时间。注意，磁头只可以单向转动。（2）完全随机读取速率（单位KB/s，不考虑调度器优化）；（3）完全顺序读取速率（单位KB/s）。

# 简答题

1. 在一个物理磁盘中，不考虑queuing和controller的时延，平均寻道时间（seek time）为3ms，磁盘转速为10000RPM（转每分钟），I/O读取速度为8MB/s，扇区大小为1KB。请分别计算（1）平均延迟时间。注意，磁头只可以单向转动。（2）完全随机读取速率（单位KB/s，不考虑调度器优化）；（3）完全顺序读取速率（单位KB/s）。

正确答案：

（1）平均延迟时间为60000(ms/min)/10000(rev/min)/2=3ms；（2）随机读取下，每个扇区的读取时间为1(KB)/8（MB/s）=0.128ms。整体，每个扇区读取时间为3ms+3ms+0.128ms=6.128ms，因此读取速率约为163KB/s.（3）顺序读取下的速率即为磁盘IO速率8MB/s。

# 简答题

5. Define three types of user-mode to kernel-mode transfers.

6. Define four types of kernel-mode to user-mode transfers.

7. Most hardware architectures provide an instruction to return from an interrupt, such as iret. This instruction switches the mode of operation from kernel-mode to user-mode.

   a. Explain where in the operating system this instruction would be used.
   b. Explain what happens if an application program executes this instruction.

# 简答题

9. With virtual machines, the host kernel runs in privileged mode to create a virtual machine that runs in user mode. The virtual machine provides the illusion that the guest kernel runs on its own machine in privileged mode, even though it is actually running in user mode.

   Early versions of the x86 architecture (pre-2006) were not *completely virtualizable* — these systems could not guarantee to run unmodified guest operating systems properly. One problem was the popf "pop flags" instruction that restores the processor status word. When popf was run in privileged mode, it changed both the ALU flags (e.g., the condition codes) and the systems flags (e.g., the interrupt mask). When popf was run in unprivileged mode, it changed just the ALU flags.

   a. Why do instructions like popf prevent transparent virtualization of the (old) x86 architecture?

   b. How would you change the (old) x86 hardware to fix this problem?

# 简答题

13. Suppose you have to implement an operating system on hardware that supports interrupts and exceptions but does not have a trap instruction. Can you devise a satisfactory substitute for traps using interrupts and/or exceptions? If so, explain how. If not, explain why.

14. Suppose you have to implement an operating system on hardware that supports exceptions and traps but does not have interrupts. Can you devise a satisfactory substitute for interrupts using exceptions and/or traps? If so, explain how. If not, explain why.

15. Explain the steps that an operating system goes through when the CPU receives an interrupt.

# 简答题

3. What happens if we run the following program on UNIX?

```
main() {
    while (fork() >= 0)
        ;
}
```

# 简答题

9. Consider the following program:

```
main (int argc, char ** argv) {
    int child = fork();
    int x = 5;

    if (child == 0) {
        x += 5;
    } else {
        child = fork();
        x += 10;
        if(child) {
            x += 5;
        }
    }
}
```

How many different copies of the variable x are there? What are their values when their process finishes?

# 简答题

```
// Program 1
main() {
    int val = 5;
    int pid;

    if (pid = fork())
        wait(pid);
    val++;
    printf("%d\n", val);
    return val;
}
```

```
// Program 2:
main() {
    int val = 5;
    int pid;
    if (pid = fork())
        wait(pid);
    else
        exit(val);
    val++;
    printf("%d\n", val);
    return val;
}
```

Output?

# 简答题

1. True or False: If a multi-threaded program runs correctly in all cases on a single time-sliced processor, then it will run correctly if each thread is run on a separate processor of a shared-memory multiprocessor. Justify your answer.

# 简答题

4. Suppose that you mistakenly create an automatic (local) variable v in one thread t1 and pass a pointer to v to another thread t2. Is it possible that a write by t1 to some variable other than v will change the state of v as observed by t2? If so, explain how this can happen and give an example. If not, explain why not.

# 简答题

4. Suppose tha
pass a point
other than v
happen and

```c
void *thread_func(void *arg) {
    // Thread t2 simply prints the value of v passed to it
    int *v_ptr = (int *)arg;
    printf("t2 sees the value of v as: %d\n", *v_ptr);
    return NULL;
}
int main() {
    pthread_t t1, t2;
    int v = 42; // Automatic variable in main thread (t1)
    // Create t2 and pass pointer to v
    pthread_create(&t2, NULL, thread_func, &v);
    // Simulate some work in t1
    char buffer[10];
    // Erroneous write beyond the bounds of buffer
    // This might overwrite the memory where v is stored
    strcpy(buffer, "This string is way too long for buffer");
    // Wait for t2 to finish
    pthread_join(t2, NULL);
    return 0;
}

```

in one thread t1 and
t1 to some variable
ain how this can

“stack corruption”

# 简答题

```
#include <stdio.h>
#include "thread.h"

static void go(int n);

#define NTHREADS 10
static thread_t threads[NTHREADS];

int main(int argc, char **argv) {
    int i;
    long exitValue;

    for (i = 0; i < NTHREADS; i++){
        thread_create(&(threads[i]), &go, i);
    }
    for (i = 0; i < NTHREADS; i++){
        exitValue = thread_join(threads[i]);
        printf("Thread %d returned with %ld\n",
                        i, exitValue);
    }
    printf("Main thread done.\n");
    return 0;
}

void go(int n) {
    printf("Hello from thread %d\n", n);
    thread_exit(100 + n);
```

**EXAMPLE:** Why might the “Hello” message from thread 2 print *after* the “Hello” message for thread 5, even though thread 2 was created before thread 5?

**EXAMPLE:** Why must the “Thread returned” message from thread 2 print *before* the Thread returned message from thread 5?

**EXAMPLE:** What is the *minimum* and *maximum* number of threads that could exist when thread 5 prints “Hello?”

# 简答题

```
#include <stdio.h>
#include "thread.h"

static void go(int n);

#define NTHREADS 10
static thread_t threads[NTHREADS];

int main(int argc, char **argv) {
    int i;
    long exitValue;

    for (i = 0; i < NTHREADS; i++){
        thread_create(&(threads[i]), &go, i);
    }
    for (i = 0; i < NTHREADS; i++){
        exitValue = thread_join(threads[i]);
        printf("Thread %d returned with %ld\n",
                        i, exitValue);
    }
    printf("Main thread done.\n");
    return 0;
}

void go(int n) {
    printf("Hello from thread %d\n", n);
    thread_exit(100 + n);
```

**EXAMPLE:** Why might the "Hello" message from thread 2 print *after* the "Hello" message for thread 5, even though thread 2 was created before thread 5?

**ANSWER: Creating and scheduling threads are separate operations.** Although threads are usually scheduled in the order that they are created, there is no guarantee. Further, even if thread 2 started running before thread 5, it might be preempted before it reaches the printf call.

Rather, the only assumption the programmer can make is that each of the threads runs on its own virtual processor with unpredictable speed. Any interleaving is possible. □

**EXAMPLE:** Why must the "Thread returned" message from thread 2 print *before* the Thread returned message from thread 5?

**ANSWER:** Since the threads run on virtual processors with unpredictable speeds, the order in which the threads finish is indeterminate. However, **the main thread checks for thread completion in the order they were created.** It calls thread_join for thread i +1 only after thread_join for thread i has returned. □

**EXAMPLE:** What is the *minimum* and *maximum* number of threads that could exist when thread 5 prints "Hello?"

**ANSWER:** When the program starts, a main thread begins running main. That thread creates NTHREADS = 10 threads. All of those could run and complete before thread 5 prints "Hello." Thus, **the minimum is two threads** — the main thread and thread 5. On the other hand, all 10 threads could have been created, while 5 was the first to run. Thus, **the maximum is 11 threads.** □

# 简答题

1. True or false. A virtual memory system that uses paging is vulnerable to external fragmentation. Why or why not?

# 简答题

12. In an architecture with paged segmentation, the 32-bit virtual address is divided into fields as follows:

| 4 bit segment number | 12 bit page number | 16 bit offset |
|---|---|---|

The segment and page tables are as follows (all values in hexadecimal):

| Segment Table | Page Table A | Page Table B |
|---|---|---|
| 0 Page Table A | 0 CAFE | 0 F000 |
| 1 Page Table B | 1 DEAD | 1 D8BF |
| x (rest invalid) | 2 BEEF | 2 3333 |
| | 3 BA11 | x (rest invalid) |
| | x (rest invalid) | |

Find the physical address corresponding to each of the following virtual addresses (answer "invalid virtual address" if the virtual address is invalid):

a. 00000000
b. 20022002
c. 10015555

# 简答题

2. Most modern computer systems choose a page size of 4 KB.
    a. Give a set of reasons why doubling the page size might increase performance.
    b. Give a set of reasons why doubling the page size might decrease performance.

3. For each of the following statements, indicate whether the statement is true or false, and explain why.

    a. A direct mapped cache can sometimes have a higher hit rate than a fully associative cache (on the same reference pattern).

    b. Adding a cache never hurts performance.

# 简答题

9. Consider a computer system running a general-purpose workload with demand paging. The system has two disks, one for demand paging and one for file system operations. Measured utilizations (in terms of time, not space) are given in Figure 9.23.

| | |
|---|---|
| Processor utilization | 20.0% |
| Paging Disk | 99.7% |
| File Disk | 10.0% |
| Network | 5.0% |

**Figure 9.23:** Measured utilizations for sample system under consideration.

For each of the following changes, say what its likely impact will be on processor utilization, and explain why. Is it likely to significantly increase, marginally increase, significantly decrease, marginally decrease, or have no effect on the processor utilization?

a. Get a faster CPU

b. Get a faster paging disk

c. Increase the degree of multiprogramming

# 简单题

- FAT、FFS和NTFS中，哪些文件系统的支持最大文件大小有限制（仅从文件系统设计的角度出发考虑）。

# 解答题

# 解答题

- 操作系统中陆续到达以下5个任务（单位：分钟）：
  - 任务A（到达时刻0，运行时间10）
  - 任务B（到达时刻0但晚于A，运行时间5）
  - 任务C（到底时刻0但晚于B，运行时间6）
  - 任务D（到达时刻2，运行时间5）
  - 任务E（到达时刻6，运行时间8）
- 请计算以下三种调度算法下的任务平均等待时间与平均完成时间。

| 调度算法 | 任务平均等待时间 | 任务平均完成时间 |
| --- | --- | --- |
| 先到先服务（FCFS） | 12.8 | 21.2 |
| 轮询调度（Round Robin，时间片长度=2，任务到达时置于等待队列最前） | 20 | 20 |
| 最短剩余任务优先（SRTF） | 9.4 | 17.8 |

# 解答题

**实现长度最大为N的生产者消费者队列可以使用信号量（Semaphore）。**

```
----------------------begin----------------------
Semaphore fullSlots = A?;
Semaphore emptySlots = B?;
Semaphore mutex = C?;
Producer(item) {
    emptySlots.P();
    mutex.P();
    Enqueue(item);
    mutex.V();
    fullSlots.V();
}
   Consumer() {
    fullSlots.P();
    mutex.P();
    item = Dequeue();
    mutex.V();
    emptySlots.V();
    return item;
}
```

问题一：A B C的值分别应该是多少？

问题二：Producer函数的前两行代码（emptySlots.P和mutex.P）是否可以调换顺序？如果不可以，可能会导致什么问题（请举例具体情况）？

# 解答题

**实现长度最大为N的生产者消费者队列可以使用信号量（Semaphore）。**

```
----------begin----------
Semaphore fullSlots = A?;
Semaphore emptySlots = B?;
Semaphore mutex = C?;
Producer(item) {
    emptySlots.P();
    mutex.P();
    Enqueue(item);
    mutex.V();
    fullSlots.V();
}
   Consumer() {
    fullSlots.P();
    mutex.P();
    item = Dequeue();
    mutex.V();
    emptySlots.V();
    return item;
}
```

问题一：A B C的值分别应该是多少？

问题二：Producer函数的前两行代码（emptySlots.P和mutex.P）是否可以调换顺序？如果不可以，可能会导致什么问题（请举例具体情况）？

正确答案：A=0，B=N，C=1.不能调换顺序，当Producer获得mutex，但是由于队列已满需要等待，而consumer由于无法获得mutex不能进行dequeue操作，产生死锁。

# 解答题

在一个Unix FFS文件系统中，块（block）大小为8KB，每个指针大小为64 bits。每一个inode中由10个直接索引（direct pointers），2个一级间接索引，1个二级间接索引，1个三级间接索引，请计算：（1）每个文件最多可以存储多少内容；（2）顺序读取一个大小为90KB的文件所有数据，需要读多少个磁盘block（inode已经在内存中，其余数据都不在内存中）。

# 解答题

在一个Unix FFS文件系统中，块（block）大小为8KB，每个指针大小为64 bits。每一个inode中由10个直接索引（direct pointers），2个一级间接索引，1个二级间接索引，1个三级间接索引，请计算：（1）每个文件最多可以存储多少内容；（2）顺序读取一个大小为90KB的文件所有数据，需要读多少个磁盘block（inode已经在内存中，其余数据都不在内存中）。

正确答案：（1）每个块可以存放8KB/64bit=1024个指针索引。10个直接索引可以存储80KB，2个一级间接索引可以存储2*1024*8KB=16MB，1个二级间接索引可以存储1024*1024*8KB=8GB，1个三级间接索引可以存储8TB，因此最多可以存储8TB+8GB+16MB+80KB大小数据。

（2）13个。

# 解答题

在一个一级分页管理、页大小为8字节、使用8 bit地址的系统中，页表项的格式如下：

| 物理页框号（page frame number，5 bits） | 脏位（dirty，1bit，1代表dirty） | 读写权限位（r/w，1bit，1代表可写） | 有效位（valid/present，1bit，1代表有效） |
|---|---|---|---|

其中，页表的前5项值分别为0x09，0x10，0x0c，0x52, 0x00（注意其中每一项的最低3位为权限位），请回答虚拟地址0x05和0x23是否可访问？如果可访问的话，请计算其对应的物理地址。

正确答案：最后3个bit表示页内偏移（下划线）
0x05（虚拟）: 0000 0101 -> 0000 1101，即0x0d（物理）
0x23（虚拟）: 0010 0011 -> 0000 0011，即0x03（物理），不能访问

| 页表项号 | 页表项（低3位为标记位） |
|---|---|
| 0 | 0000 1001 (0x08) |
| 1 | 0001 0000 (0x10) |
| 2 | 0000 1100 (0x0c) |
| 3 | 0101 0010 (0x52) |
| 4 | 0000 0000 (0x00) |