操作系统的作用

- 提供容易使用的界面(终端用户及程序员)
- 最大限度地提高资源利用率(CPU,内存)
- 为多用户提供分时服务(time sharing system)
- 在多用户多系统之间实现资源共享(存储、打印机)
- 嵌入式设备：界面问题、电池寿命问题(不只是OS的任务)

操作系统的作用: 提供易于使用的界面

磁盘读写操作:

- 磁头、柱面、磁道、扇区
- 读写前需等待机械运动结束
- 数据存储可能不连续
- 磁盘大小、速度各不相同
- 对程序员而言, 编程读写磁盘数据是非常复杂的任务

操作系统提供的磁盘读写界面:

- open(), close()
- read(), write()
- named files(按名访问)

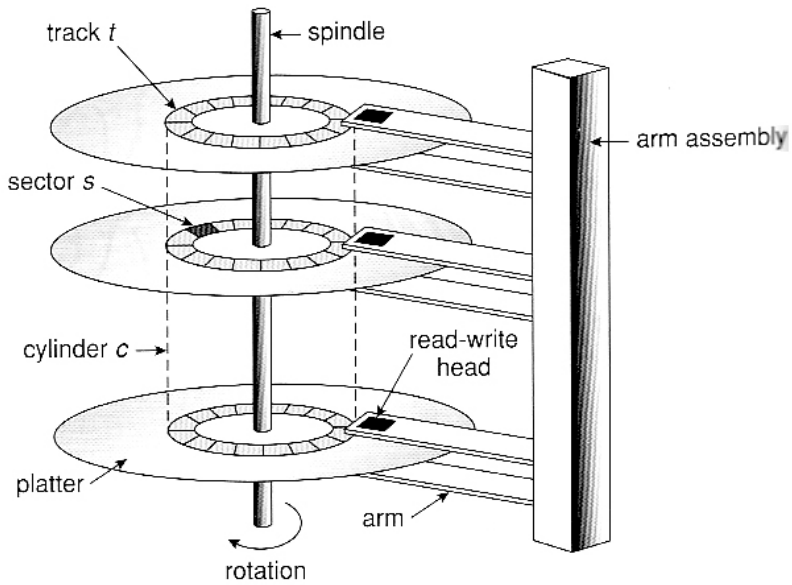
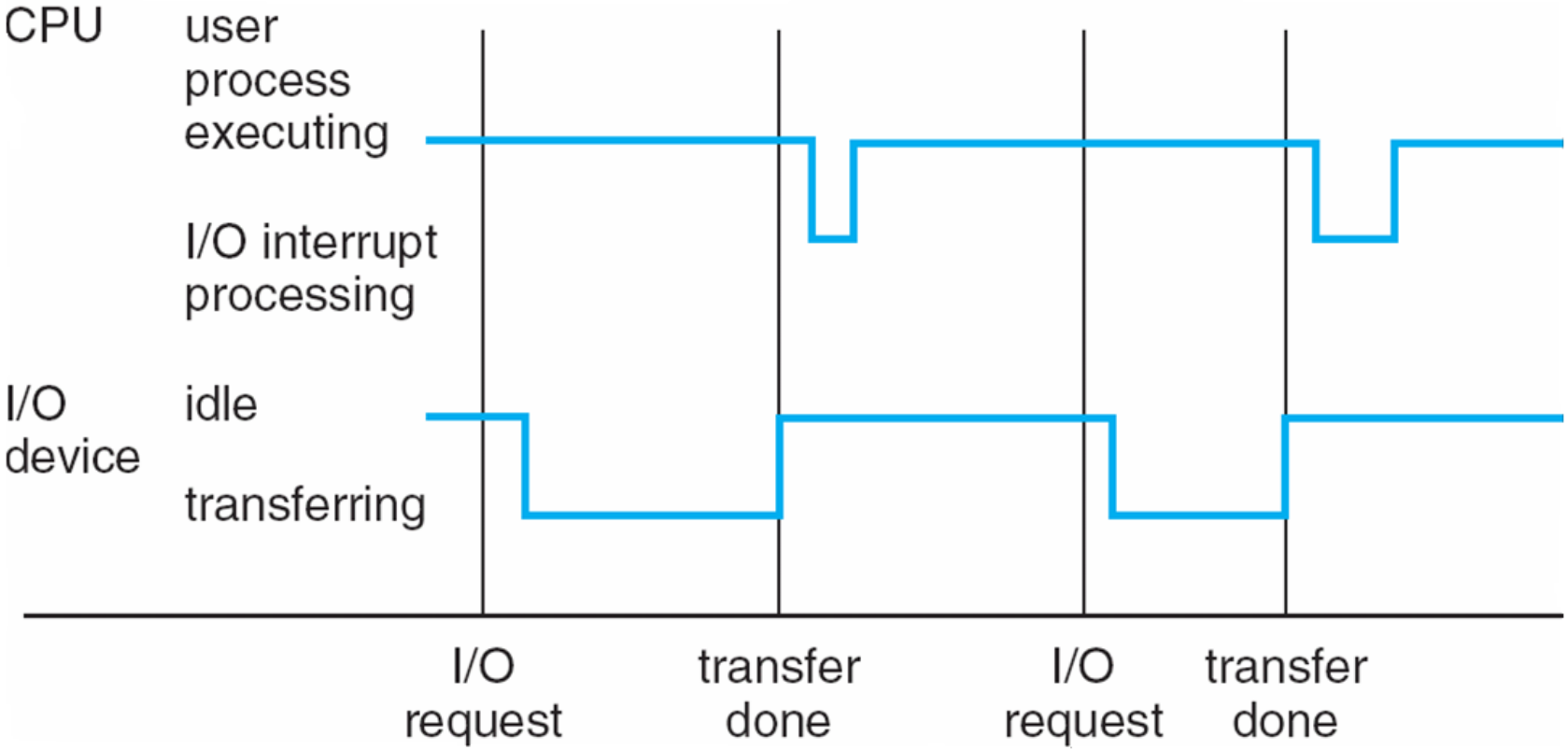


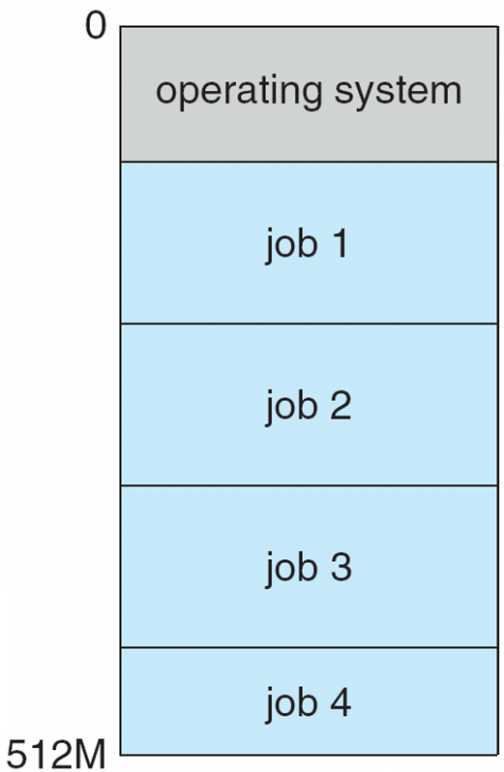
Figure 12.1 Moving-head disk mechanism.

操作系统的中断(interrupt)处理机制

- 操作系统由中断驱动
- 硬件中断与软件中断



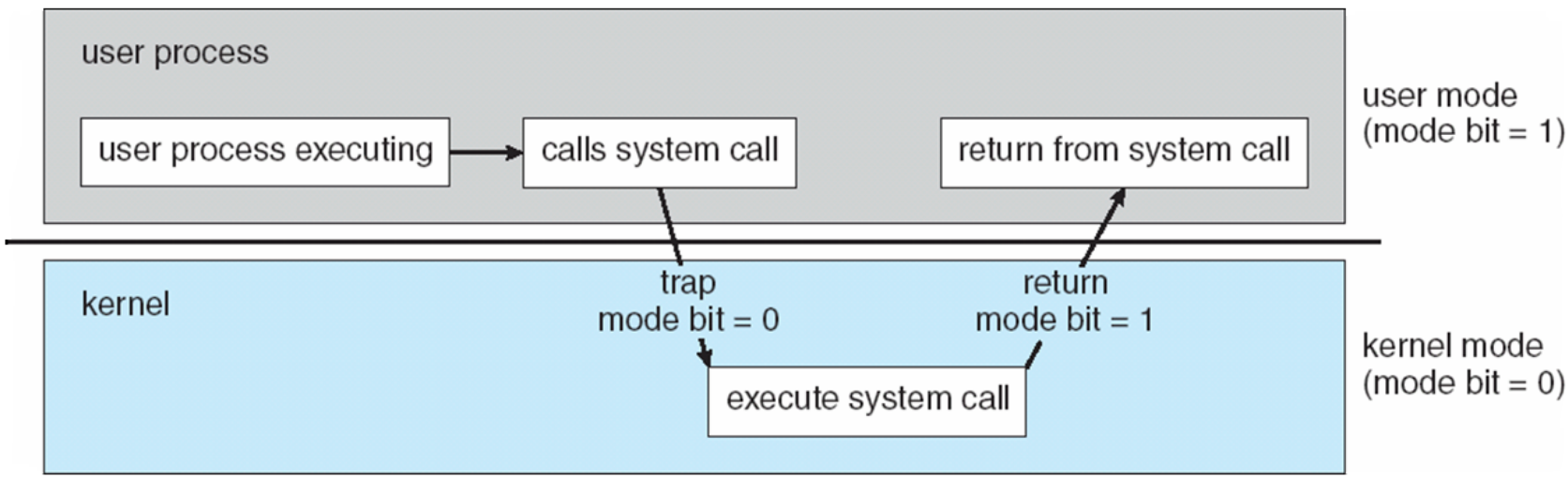
多道程序设计的概念



- 单个用户或程序无法使CPU或外设保持忙碌
- 引入多道程序设计技术
- 多个作业(jobs)驻留内存
- 操作系统需要进行作业调度
- 需要等待I/O时，调入另一作业运行

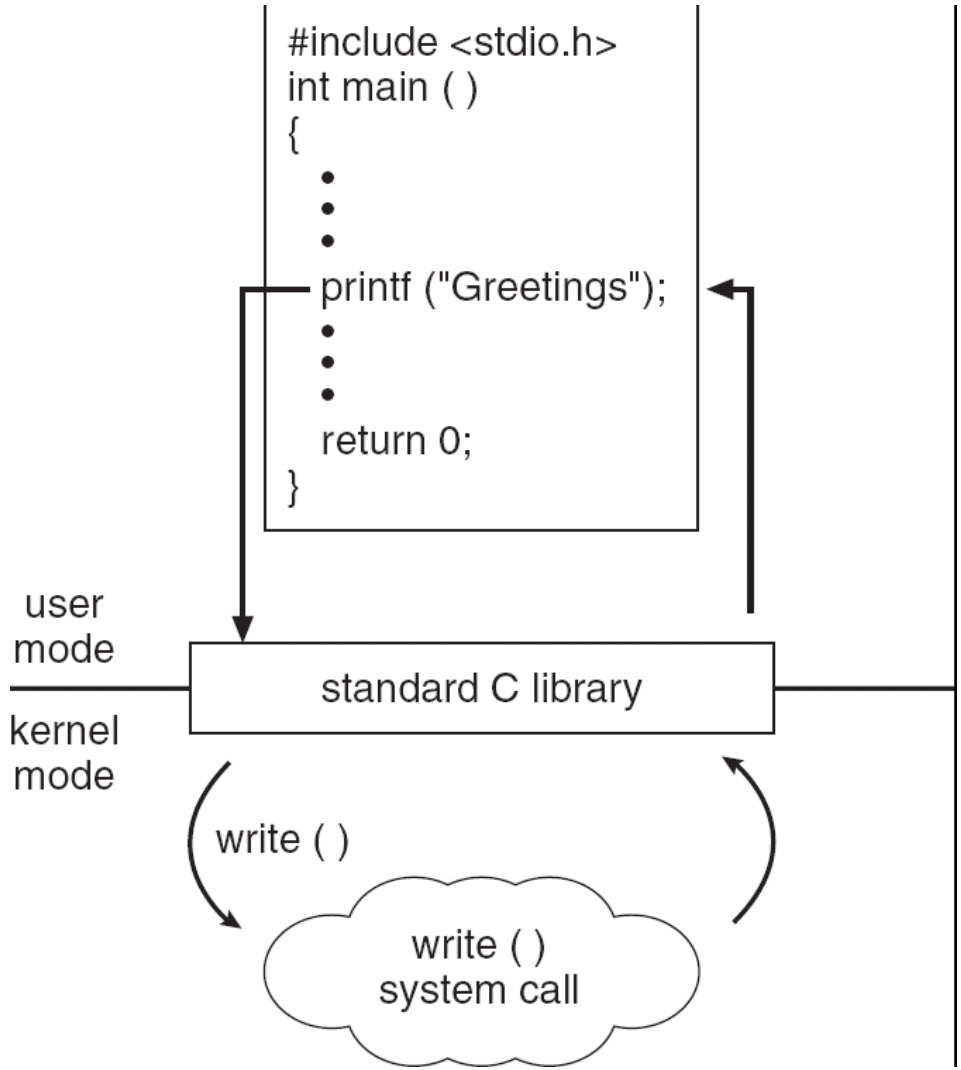
核心态与用户态

- 核心态: 执行操作系统代码
- 用户态: 执行用户程序代码
- 思考：哪些指令需要在核心态下执行?
- 系统调用导致从用户态转入核心态



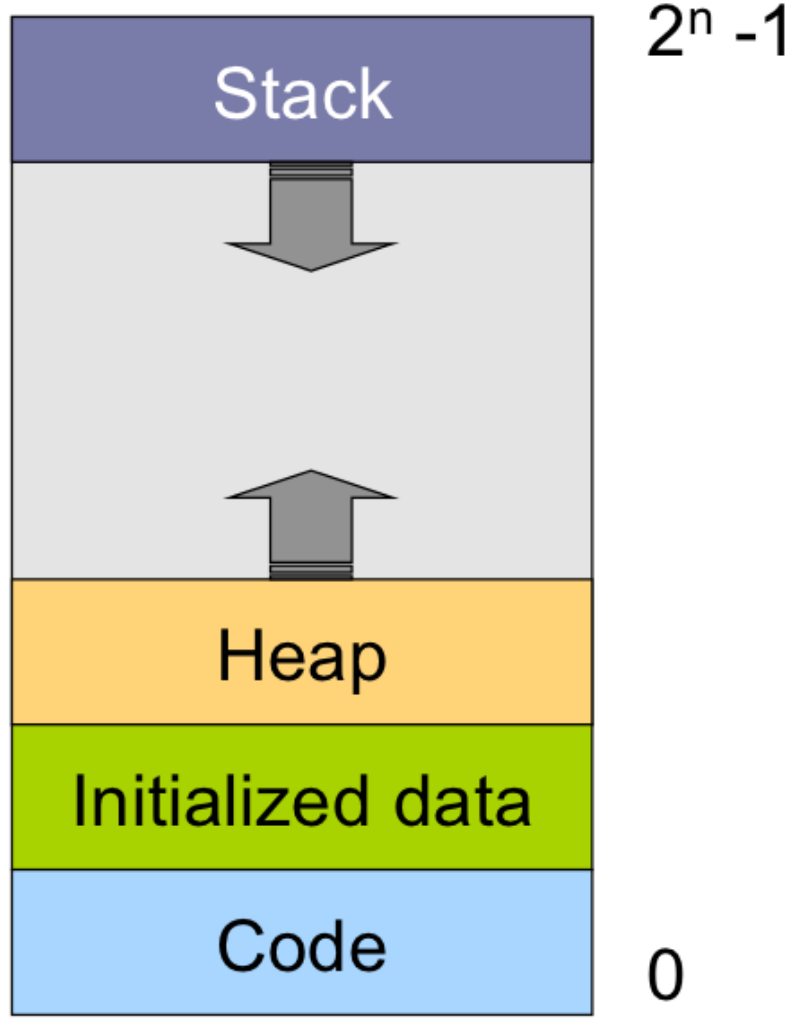
系统调用

- CLI/GUI是人使用操作系统时的界面
- 系统调用可看作获取操作系统服务的编程界面(接口)



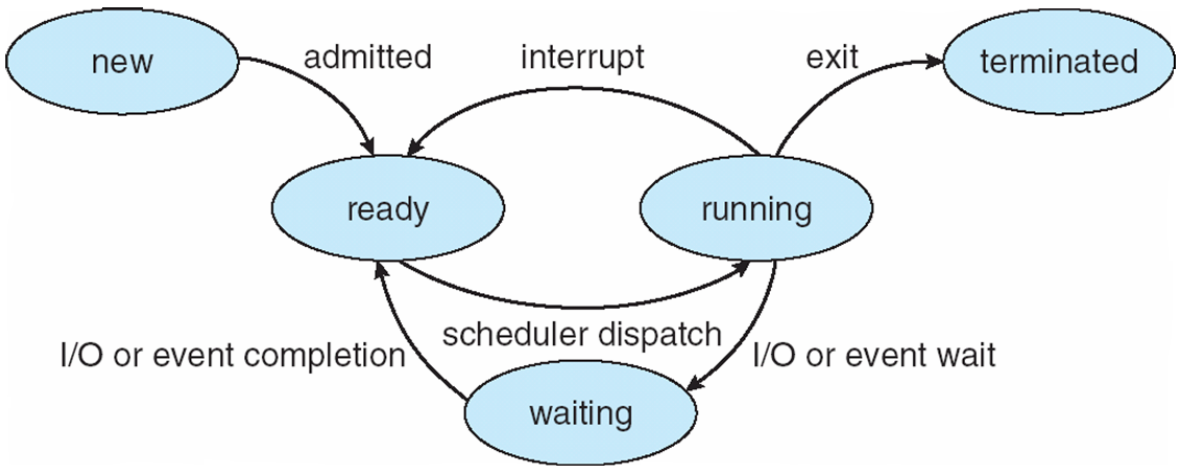
进程概念

- 操作系统执行用户程序
  - 批处理系统— 作业
  - 分时系统— 用户程序、任务
- 进程: 运行中的程序
- 进程包含三部分内容:
  - 程序代码
  - 当前状态: 程序计数器以及寄存器
  - 栈(函数参数, 返回地址, 局部变量)
  - 数据区(全局变量)
  - 堆(动态分配的内存)



进程的状态及其转移

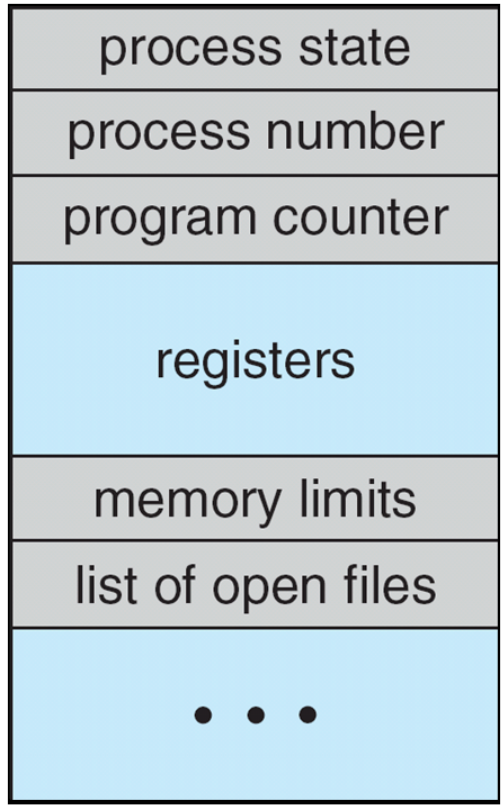
- new: 进程正在被创建
- running: 正在运行
- waiting: 挂起
- ready: 就绪
- terminated: 进程正在被销毁



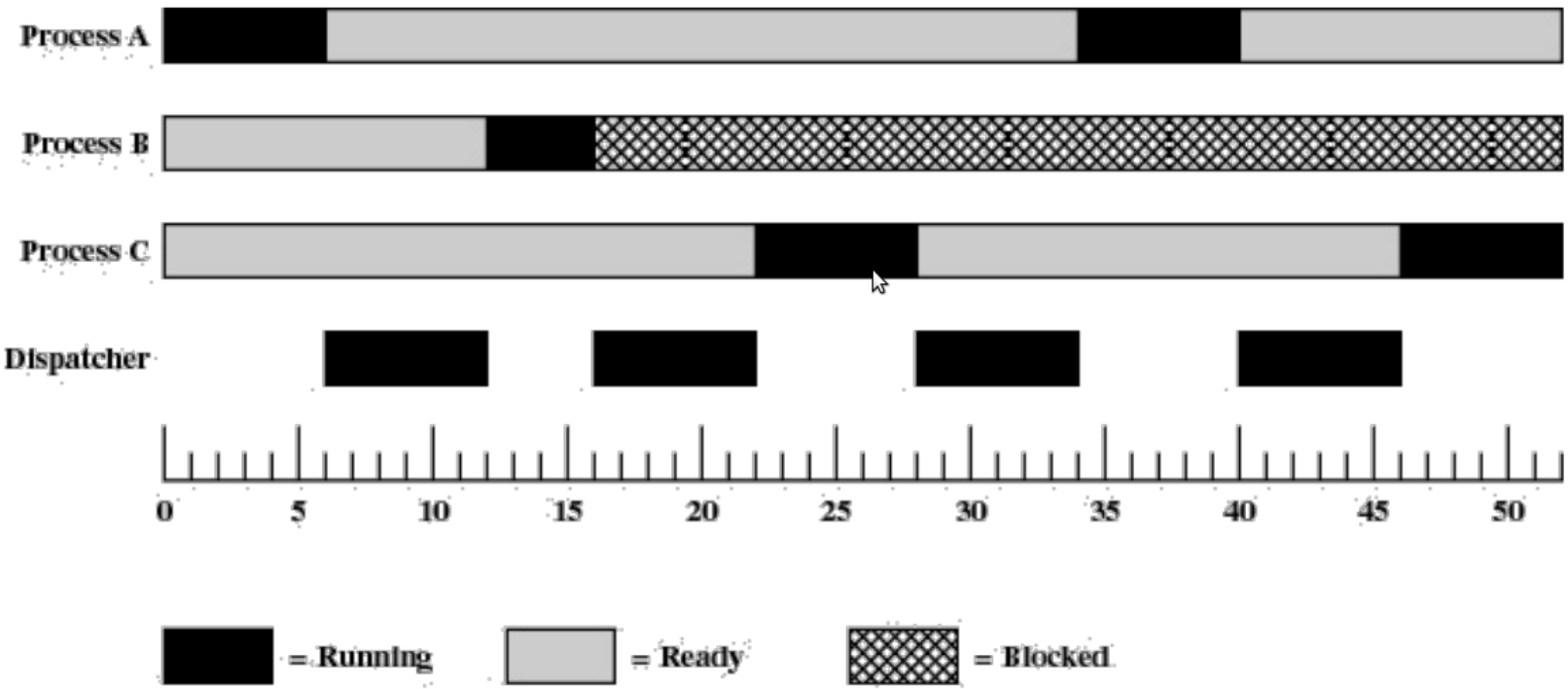
进程控制块(PCB)

用于存放与每个进程相关的信息

- 进程状态
- 程序计数器PC
- CPU寄存器
- CPU调度信息
- 内存管理信息
- I/O状态信息
- 记账信息



CPU在进程间切换


Process A
Process B
Process C
Dispatcher
0
5
10
15
20
25
30
35
40
45
50
= Running
= Ready
= Blocked

上下文切换(context switch)

- CPU分配给新进程时，原进程的状态需要保存，新进程的状态需要载入
- 进程的上下文(context)保存于进程控制块(PCB)中
- 上下文切换时间属无用开销，因此越快越好
- 切换时间取决于硬件支持(e.g, 具有多组寄存器的CPU)

线程的概念

可以发现，进程概念可以分割成两块：

- 资源分配单位
  - 内存空间
  - I/O设备，文件等
- 执行单位
  - 单一执行路径
  - 状态：寄存器、栈等
- 重新定义术语的内涵：
  - 进程：资源分配单位
  - 线程：执行单位（轻量级进程）
  - 多线程技术：支持一个进程中同时运行多个线程

为什么引入多线程技术?

- 应用程序可能同时执行多个动作(例如文字编辑器)
- 线程比进程更容易创建和销毁
- 如果程序部分因I/O阻塞，其余线程可以运行
  - CPU密集型与I/O密集型并行
  - 加快系统速度
- 充分利用多核处理器硬件资源

临界资源、临界区与竞争条件

- 临界资源
- 临界区
- 多个进程并发访问和操作同一数据且执行结果与访问的特定顺序有关，称为竞争条件。

对临界资源的互斥访问

理想的互斥方案需要满足4个条件：

1. 两个进程不能同时进入临界区
2. 不能依赖CPU数目或者运行速度
3. 不在临界区的进程，不能妨碍其他进程进入临界区
4. 任一进程需在有限时间内能够进入临界区

对临界资源的互斥访问: 忙等待

- 需要硬件支持TSL指令
- 进程进入临界区前，调用enter_region
- 离开临界区时，调用leave_region
- 这是一个正确的解决方法，但是...

```
enter_region:
        TSL REGISTER,LOCK
        CMP REGISTER,#0
        JNE enter_region
        RET

leave_region:
        MOVE LOCK,#0
        RET
```

对临界资源的互斥访问: 忙等待

两个缺点：

- 缺点1：忙等待浪费了CPU时间
- 缺点2：优先级反转问题
  - 进程H优先级高于进程L, 二者同时需要某临界资源
  - 假设当进程L在临界区时，进程H可以运行
  - 结局：进程L永远无法离开临界区，H永远忙等待

为了克服这些缺点，增加sleep和wakeup系统调用

生产者–消费者问题

```
#define N 100
int count = 0;

void producer(void)
{
        int item;

        while (TRUE) {
                item = produce_item( );
                if (count == N) sleep( );
                insert_item(item);
                count = count + 1;
                if (count == 1) wakeup(consumer);
        }
}
```

```
void consumer(void)
{
        int item;

        while (TRUE) {
                if (count == 0) sleep( );
                item = remove_item( );
                count = count – 1;
                if (count == N – 1) wakeup(producer);
                consume_item(item);
        }
}
```

竞争条件：消费者进程测试count==0成功后、调用sleep之前，调度生产者进程运行

信号量机制(Semaphores)

为了解决唤醒信号丢失的问题，引入信号量，它是一种特殊的整型变量。在信号量上定义两个原子操作：

down 如果信号量值大于0，则将其减1然后返回；否则，进程在该信号量上进入睡眠

up 如果有进程在该信号量上睡眠，则选择其中一个唤醒；否则，信号量加1

用信号量解决生产者–消费者问题

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void)
{
        int item;

        while (TRUE) {
                item = produce_item( );
                down(&empty);
                down(&mutex);
                insert_item(item);
                up(&mutex);
                up(&full);
        }
}
```

```
void consumer(void)
{
        int item;

        while (TRUE) {
                down(&full);
                down(&mutex);
                item = remove_item( );
                up(&mutex);
                up(&empty);
                consume_item(item);
        }
}
```

该方案中，信号量empty和full具有计数和同步功能，而mutex仅有互斥功能。

专门用来实现互斥的特殊信号量– 互斥锁

互斥锁只有两种状态：locked (1) / unlocked (0)

```
mutex_lock:
        TSL REGISTER,MUTEX
        CMP REGISTER,#0
        JZE ok
        CALL thread_yield
        JMP mutex_lock
ok:     RET


mutex_unlock:
        MOVE MUTEX,#0
        RET
```

信号量的危险情形— 管程机制的引入

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void)
{
        int item;

        while (TRUE) {
                item = produce_item( );
                down(&empty);
                down(&mutex);
                insert_item(item);
                up(&mutex);
                up(&full);
        }
}
```

```
void consumer(void)
{
        int item;

        while (TRUE) {
                down(&full);
                down(&mutex);
                item = remove_item( );
                up(&mutex);
                up(&empty);
                consume_item(item);
        }
}
```

如果把producer中的down(empty)和down(mutex)顺序颠倒，则当缓冲区满时，会发生什么？

管程:解决生产者–消费者问题

```
monitor ProducerConsumer
    condition full, empty;
    integer count;

    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count + 1;
        if count = 1 then signal(empty)
    end;

    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count := count - 1;
        if count = N - 1 then signal(full)
    end;

    count := 0;
end monitor;
```

```
procedure producer;
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item)
    end
end;

procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item)
    end
end;
```

注意概念: 条件变量empty, full以及wait, signal

此外，insert与remove之间的互斥由编译器完成

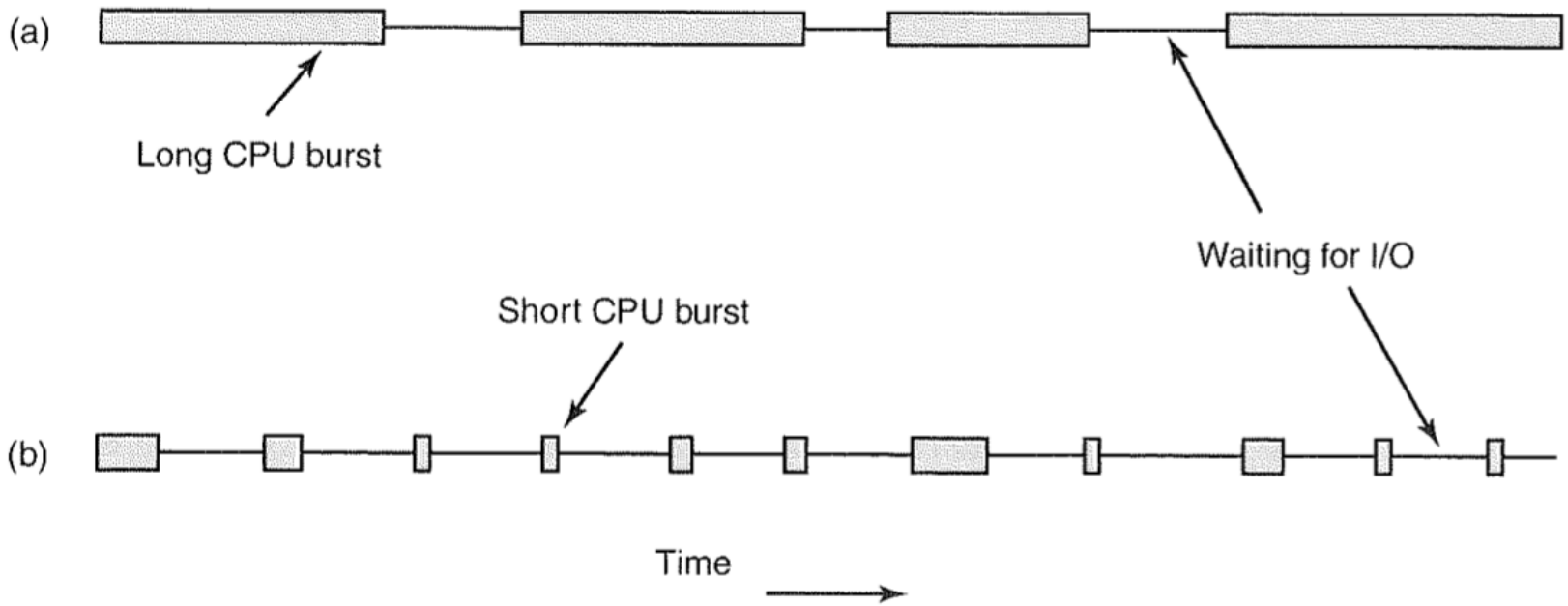
管程:解决生产者–消费者问题

- 管程内程序段之间的互斥（自动）
- 进程同步问题?：条件变量及wait, signal实现
  - wait: 将当前进程阻塞，并允许其他进程进入管程
  - signal: 将被相应条件变量阻塞的进程唤醒
- 上述方法中，signal必须是最后一条指令，为什么?

进程（线程）调度

当系统中有多个进程或线程处于就绪态时，操作系统需要从中选择一个放到CPU上运行。这就是进程调度问题。实现该任务的部件称作调度器。

进程的典型行为: CPU密集型与IO密集型进程



思考：两种进程举例?

二者关键区别：不是I/O时间长度，而是CPU时间长度

进程的典型行为: CPU密集型与IO密集型进程

基本想法：如果某I/O进程处于就绪态，则应该努力优先让其运行。为什么？(这里有个深刻原因)

进程调度的时机：何时调度?

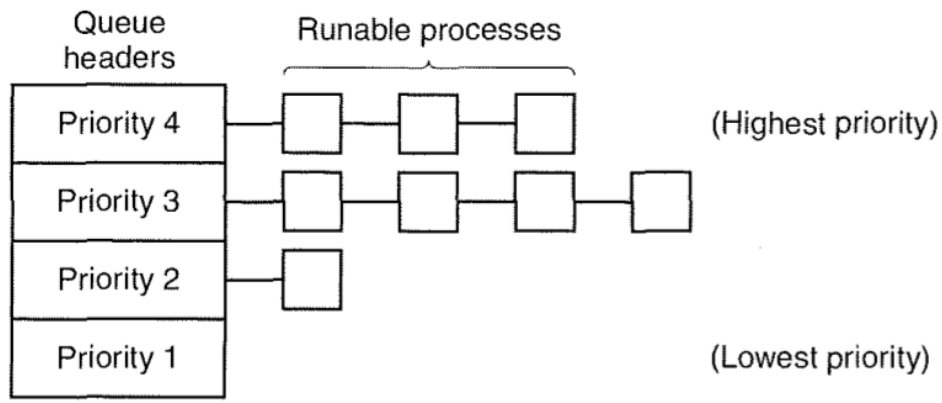
1. 新进程创建时，是继续运行父进程，还是运行新创建的子进程?
2. 当前进程退出时，CPU空闲，此时需从就绪态进程集合中选择一个运行
3. 当前进程阻塞时（I/O或者信号量引起）
4. 当发生I/O中断时，由此I/O信号导致阻塞的进程进入就绪态

抢占式调度与非抢占式调度

- 时钟硬件中断信号的频率大约为50～60Hz
- 在1个或者K个时钟中断信号处，强迫终止当前运行的进程。这类调度称为抢占式调度
- 非抢占式调度：进程一旦运行，则除非它阻塞或者自愿放弃CPU，不剥夺其CPU使用权。
- 抢占式调度用于分时系统；需要时钟硬件的支持

进程调度算法

- 时间片轮转调度
  - 公平、简单
  - 关键数据结构：队列
- 优先级调度
  - 基本假定是有些进程比其他进程重要
  - 关键数据结构：优先队列
- 优先级与时间片轮转相结合



内存管理器(Memory Manager)的任务

- 提供内存抽象界面
- 分配物理内存，回收物理内存
- 记录内存使用情况等

没有内存抽象：程序员直接操作物理内存

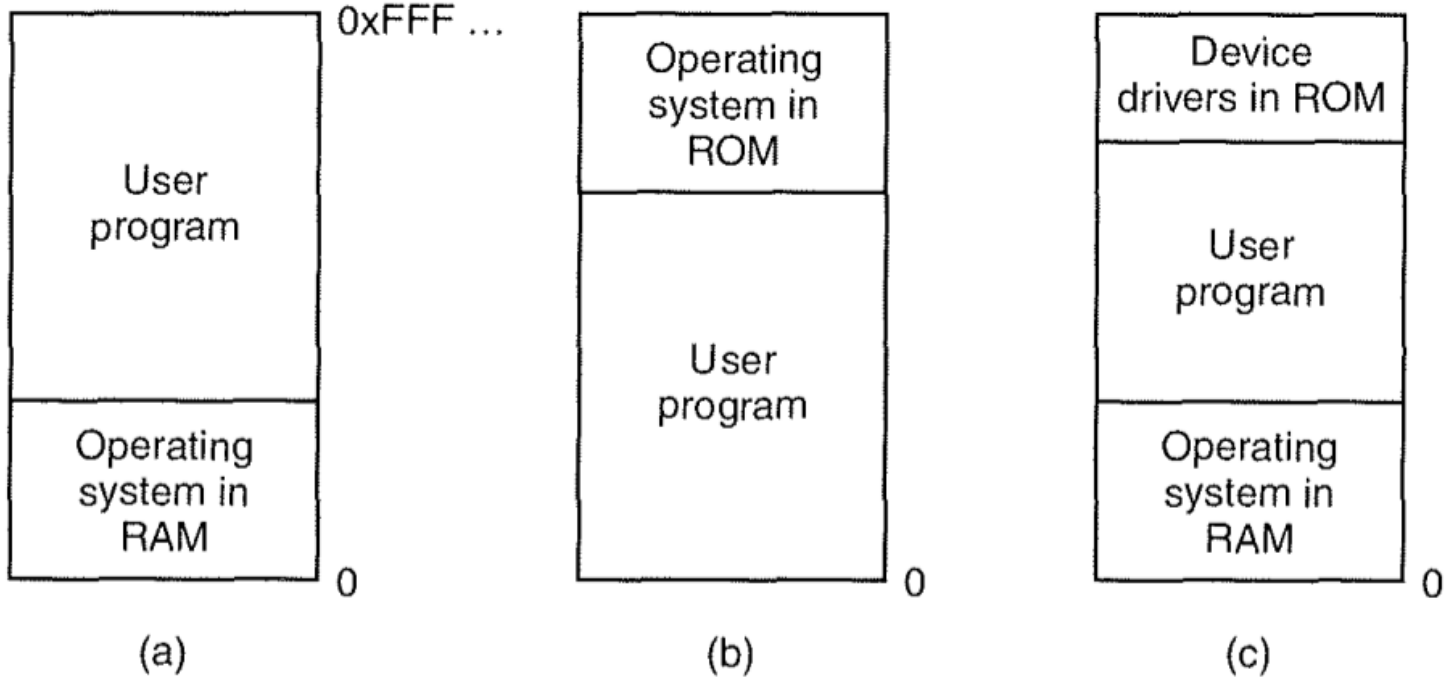


Figure 3-1. Three simple ways of organizing memory with an operating system and one user process. Other possibilities also exist.

a. 内存中一次只能驻留一个程序：MOV REGISTER1, 1000
b. 操作系统自身代码难以保护(没有地址空间概念)

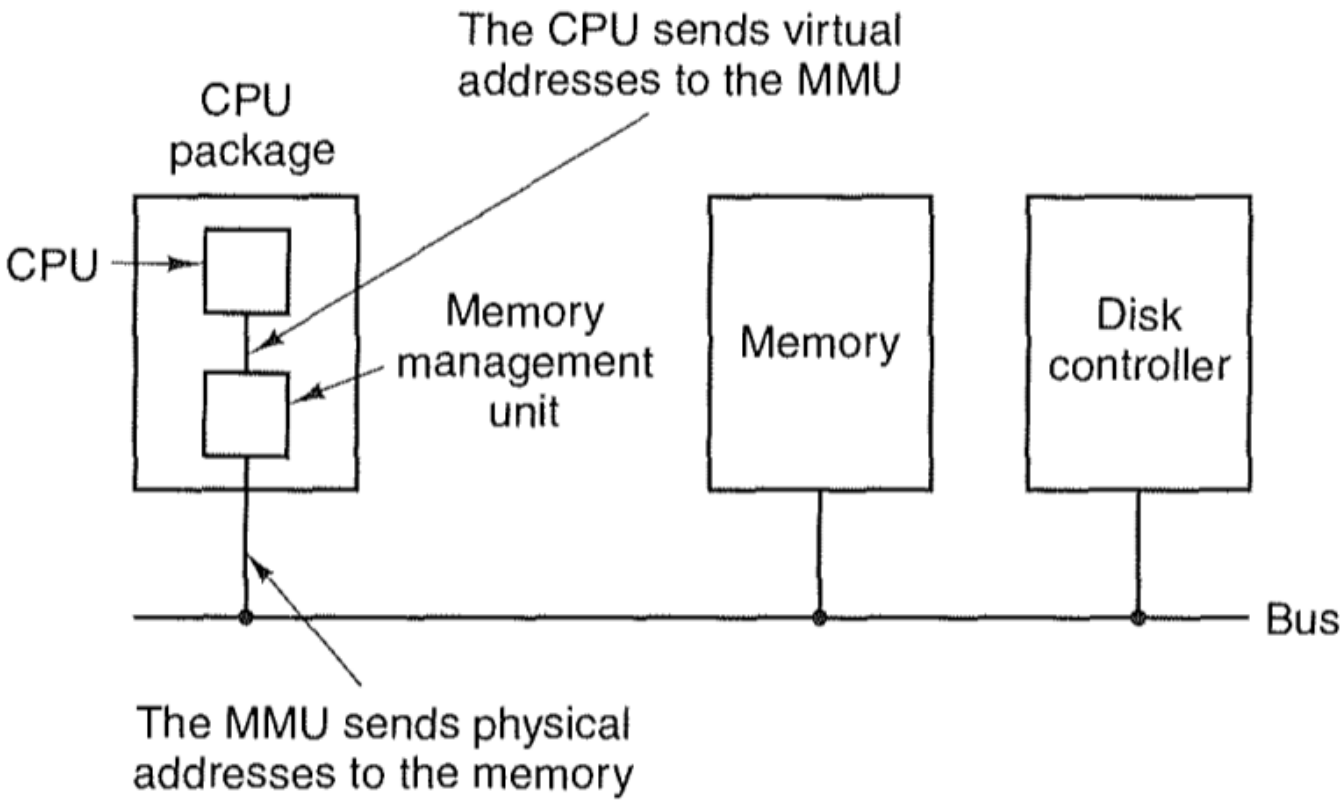
虚拟内存技术

- 问题一：如何让多个程序驻留内存?
  - 每个程序有专属地址空间
  - 程序不能非法访问其他程序的地址空间
- 问题二：如何满足程序对内存的无限需求?
  - 地址空间分成若干页面
  - 地址空间的页面映射到物理内存的页框内
  - 利用页表实现页面号到页框号的映射
  - 不是所有的页面都需要放到物理内存中
  - 缺页中断技术

虚拟内存技术

分页技术的精髓：不是所有页面都需要同时调入内存

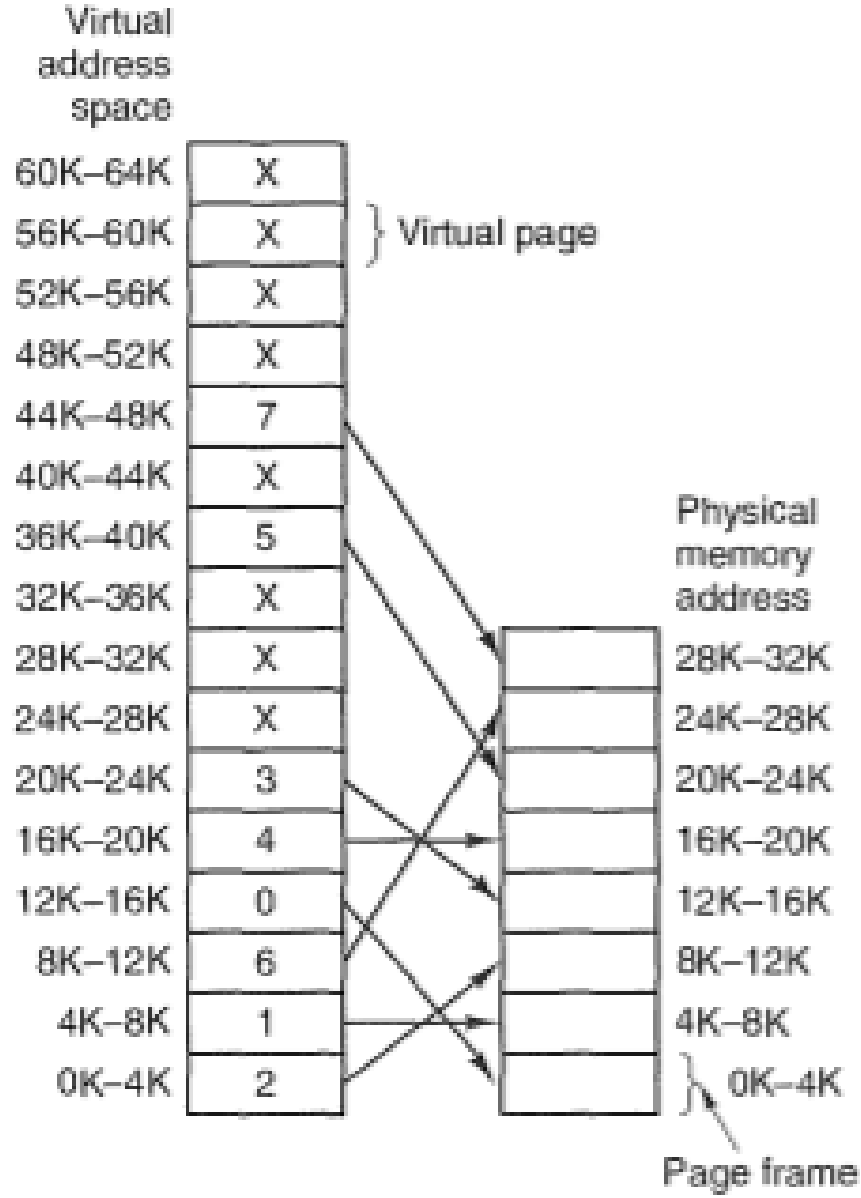
虚拟地址、物理地址及内存管理单元



注意:程序中的地址全都是虚拟地址

虚拟地址、物理地址及内存管理单元

- 虚拟地址空间：64K (16 bit)
- 物理地址空间：32K (15 bit)
- 页面大小：4K
- 共16个（虚拟）页面，8个（物理）页框
- 对于大于32K的程序，只能有32K驻留物理内存(右图数字部分)



虚拟地址、物理地址及内存管理单元

MOV REG, 20500

20500 = 5 * 4K + 20

页面5 → 页框3

12K + 20 = 12308（物理地址）

MOV REG, 32780

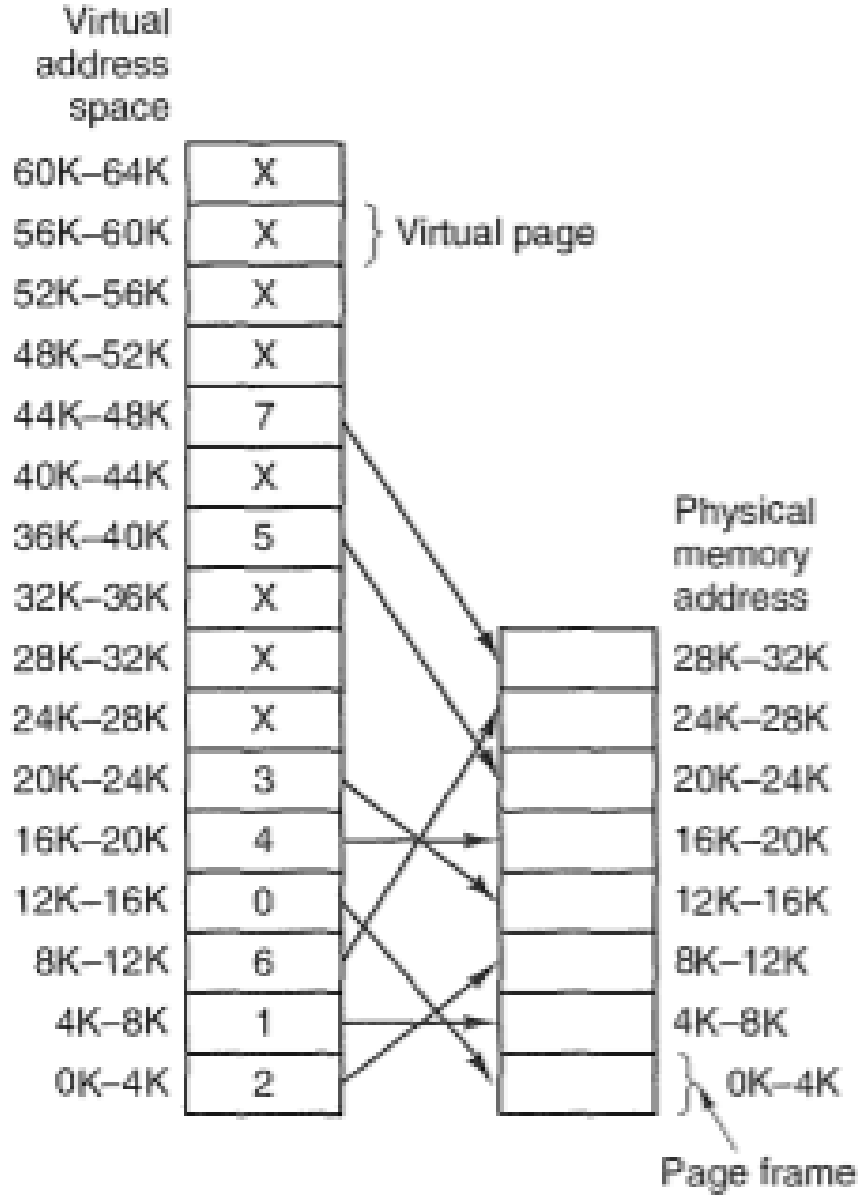
32780 = 8 * 4K + 12

对应虚拟页面8（缺页）

what next?

页表内容

需要由操作系统维护



虚拟地址、物理地址及内存管理单元

虚拟地址映射到物理地址，考虑右图例子：

- 页面大小：4K
- 页内地址为12位
- 对于16位机器而言，有4位用于页表索引
- 因此共有16个虚拟页面
- 8个物理页框（需3位)

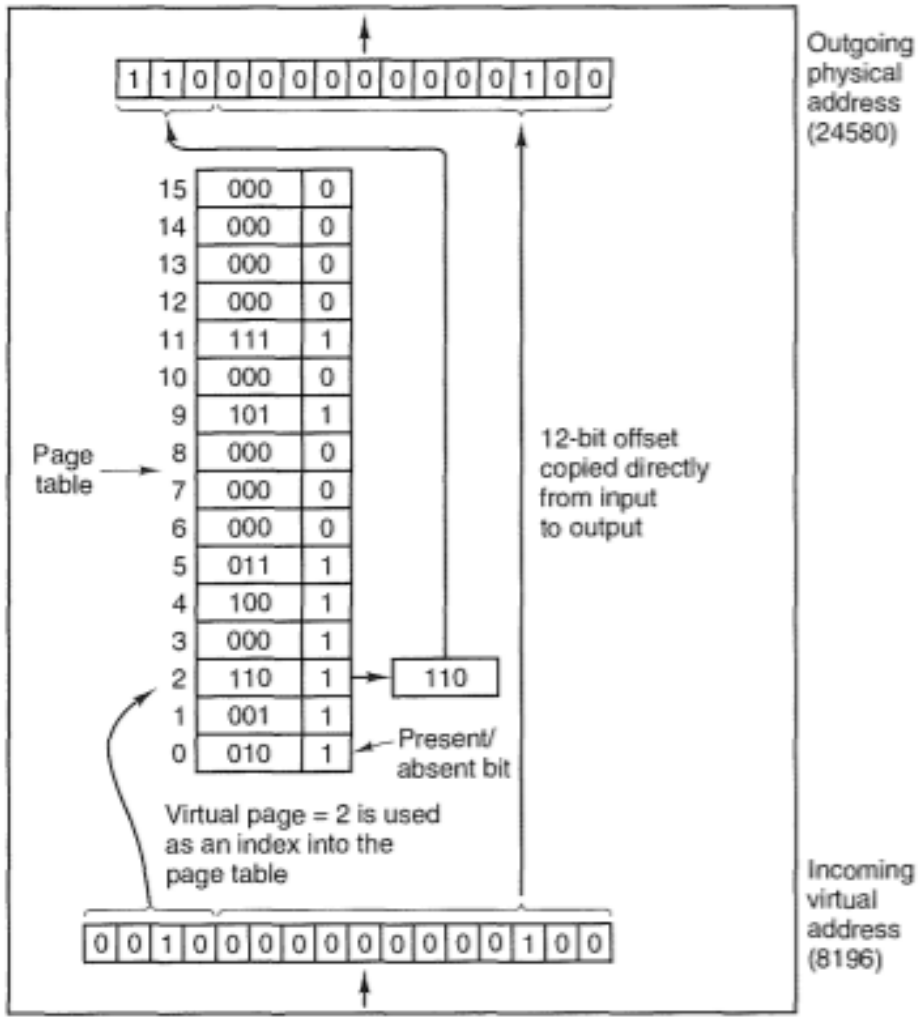
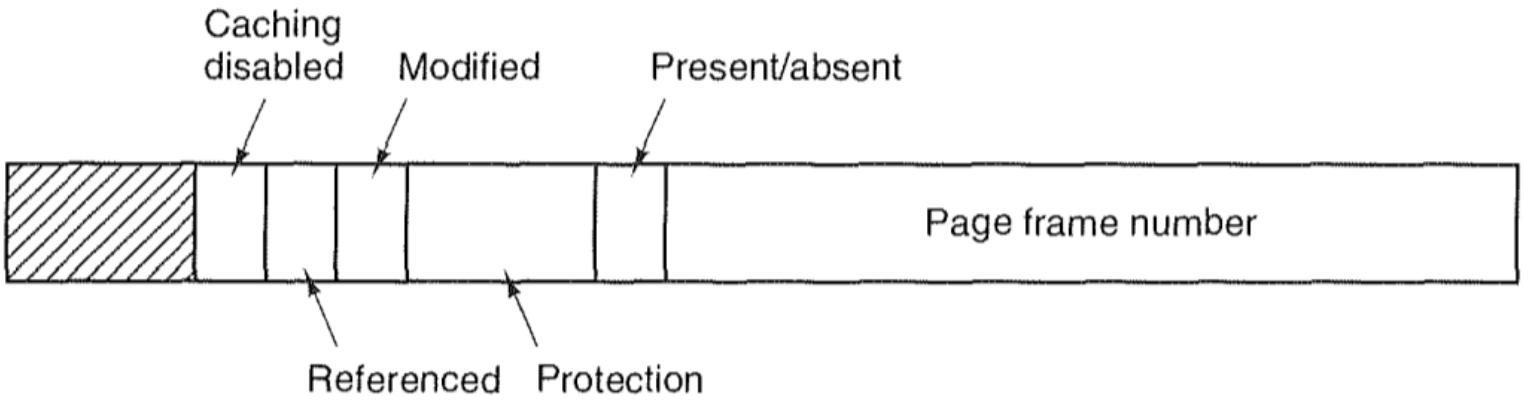


Figure 3-10. The internal operation of the MMU with 16 4-KB pages.

页表项



page frame number 描述该页表项对应的页框编号

present / absent 表示该页表是否在内存中

protection 含读、写、执行等权限信息

modified 该页表内容是否被修改过

referenced 该页表内容是否被用过（读写）

caching disabled 用于memory mapped I/O（后续）

虚拟内存技术与分页技术面临的两大问题

1. 从虚拟地址到物理地址的映射必须快
   - 每条指令都需从内存取出
   - 大量指令涉及读写内存(CISC机器)
2. 如果虚拟地址空间很大，则页表规模会特别大
   考虑页面大小4K的虚拟内存系统：
   - 32位虚拟地址: 100万个页表项
   - 64位虚拟地址：45035996亿个页表项
   - 注意：每个进程都需要单独的页表！！

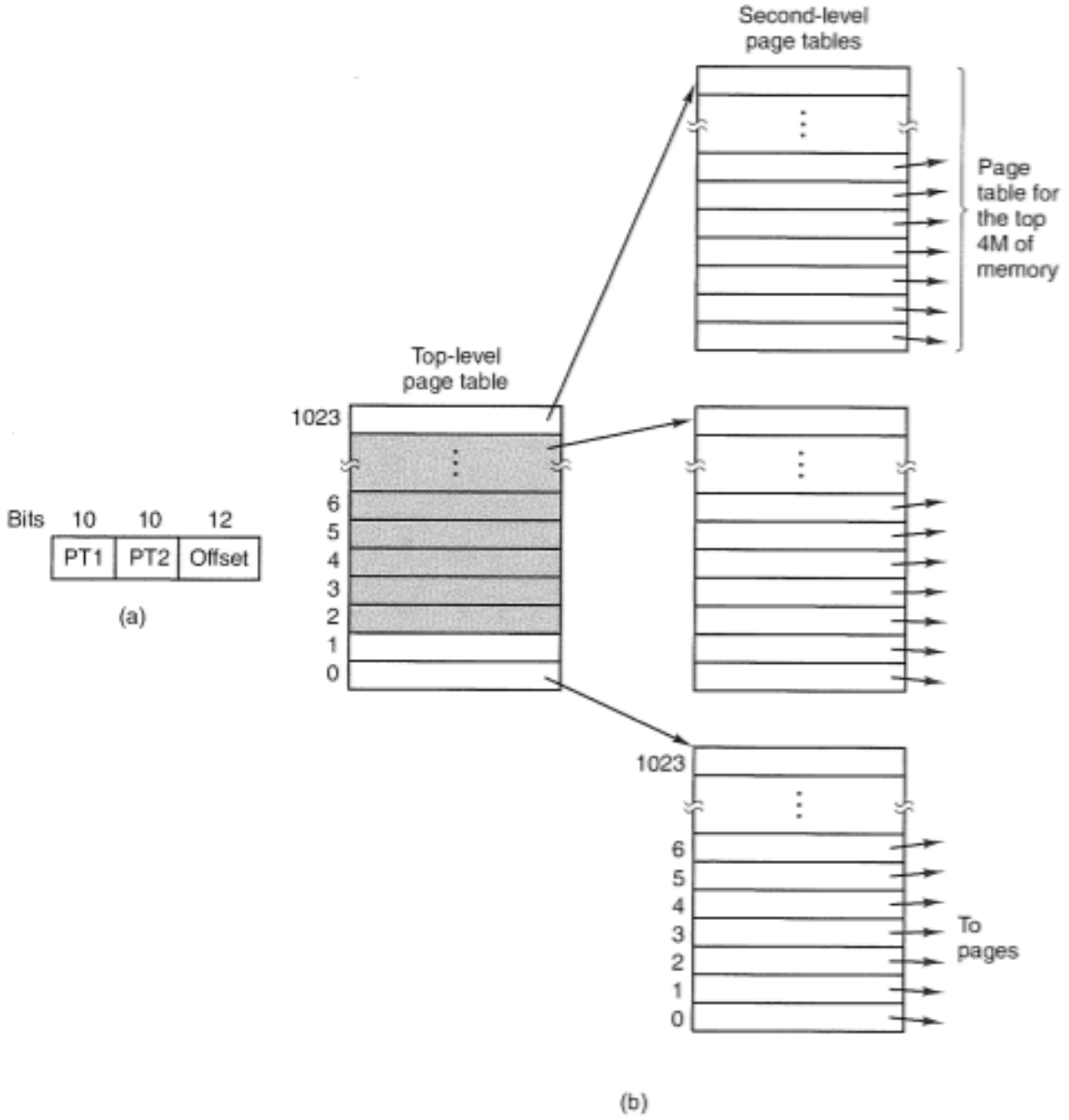
从虚拟地址到物理地址的快速映射：TLB(联想式存储)

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

放置在MMU里面。来虚拟地址，先查TLB。

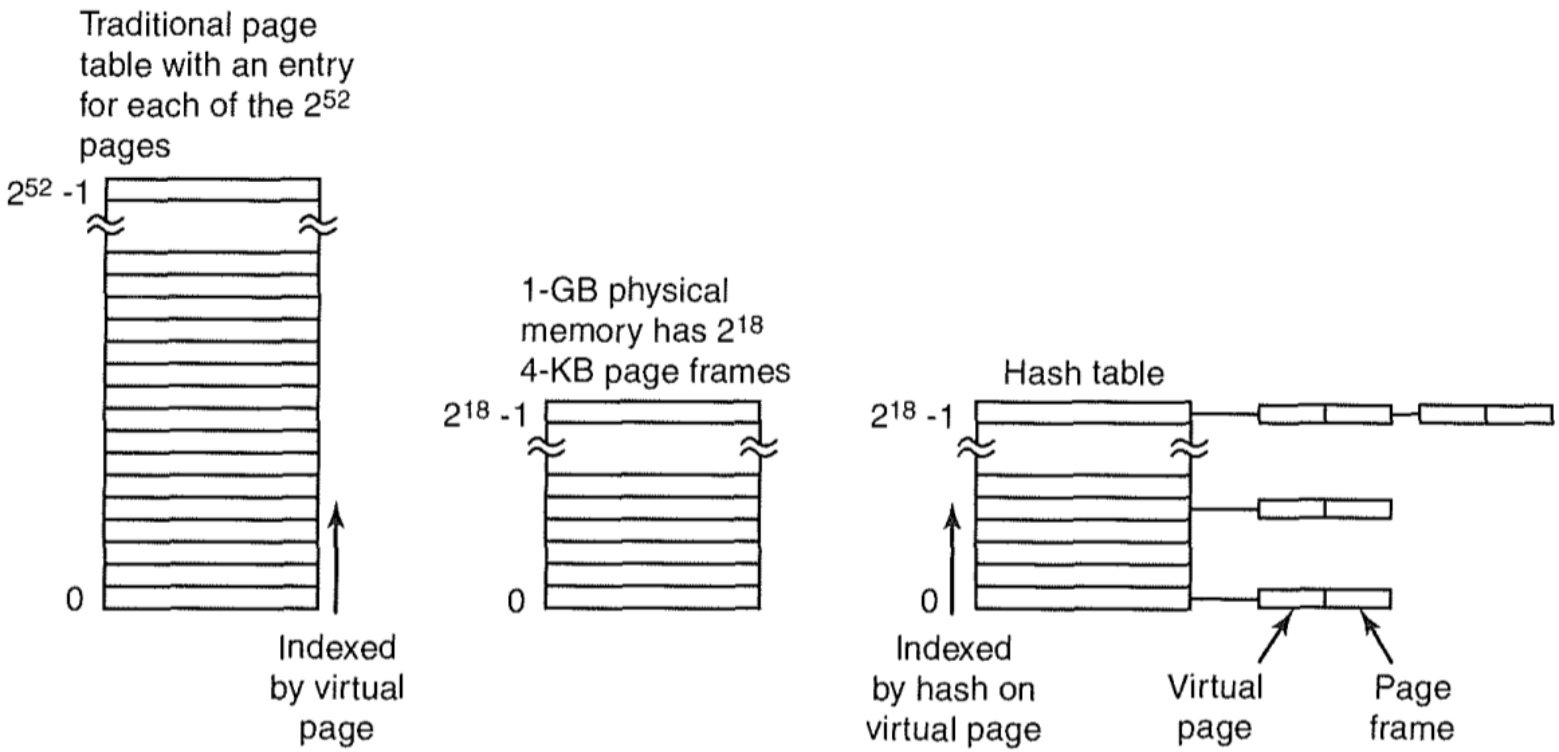
- 若能在TLB中查到该虚拟地址，则直接输出物理页框号码
- 否则，去内存中的页表中查找对应页框号码，并将其调入TLB

处理大规模地址空间的方法一：多级页表



多级页表：页表的全部内容不必都放在内存中

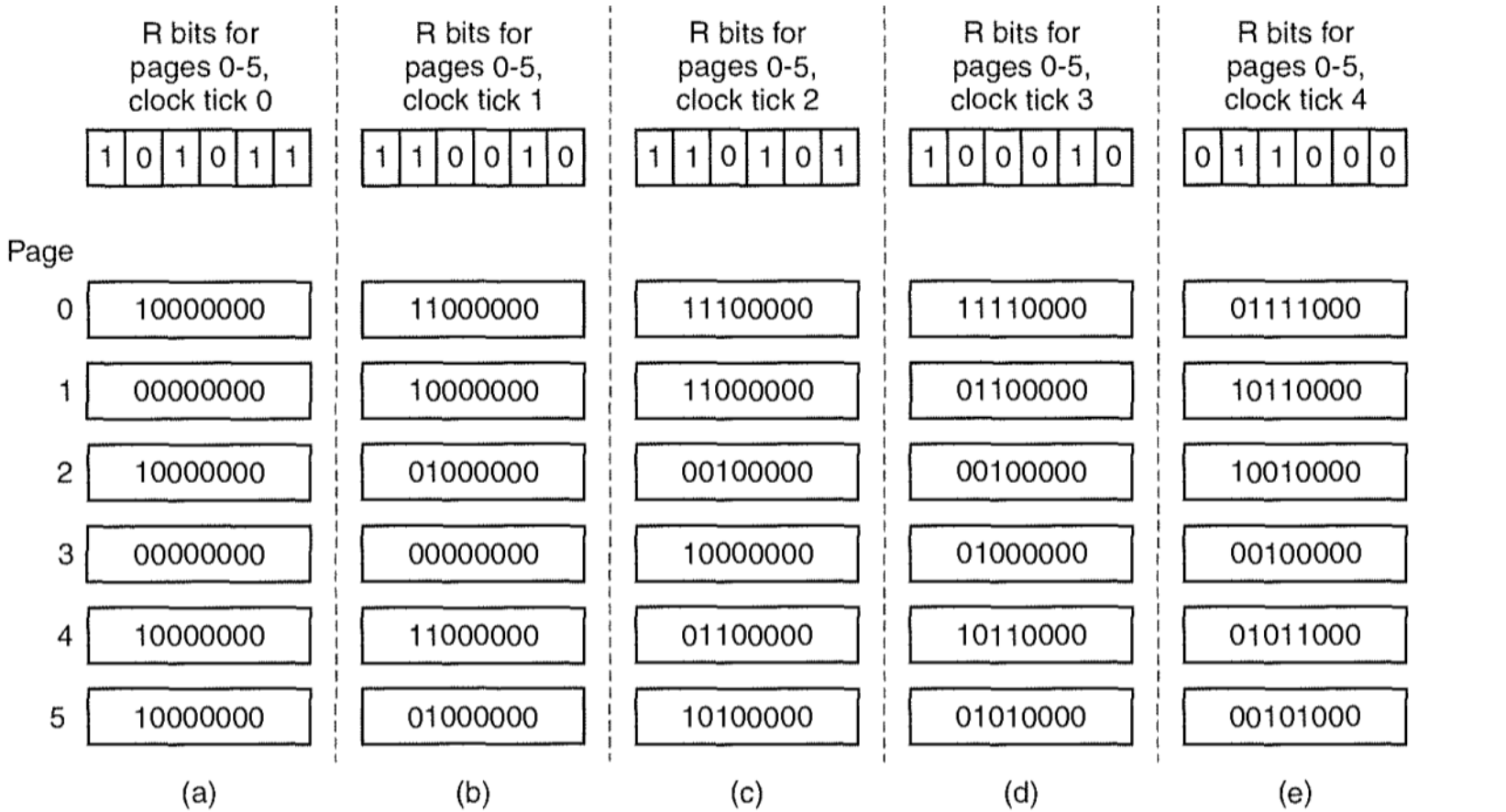
处理大规模地址空间的方法二：倒排页表



页面置换算法

当发生缺页时，OS需要选择一个页面将其从物理内存转移至硬盘，然后从硬盘调入所缺页面。

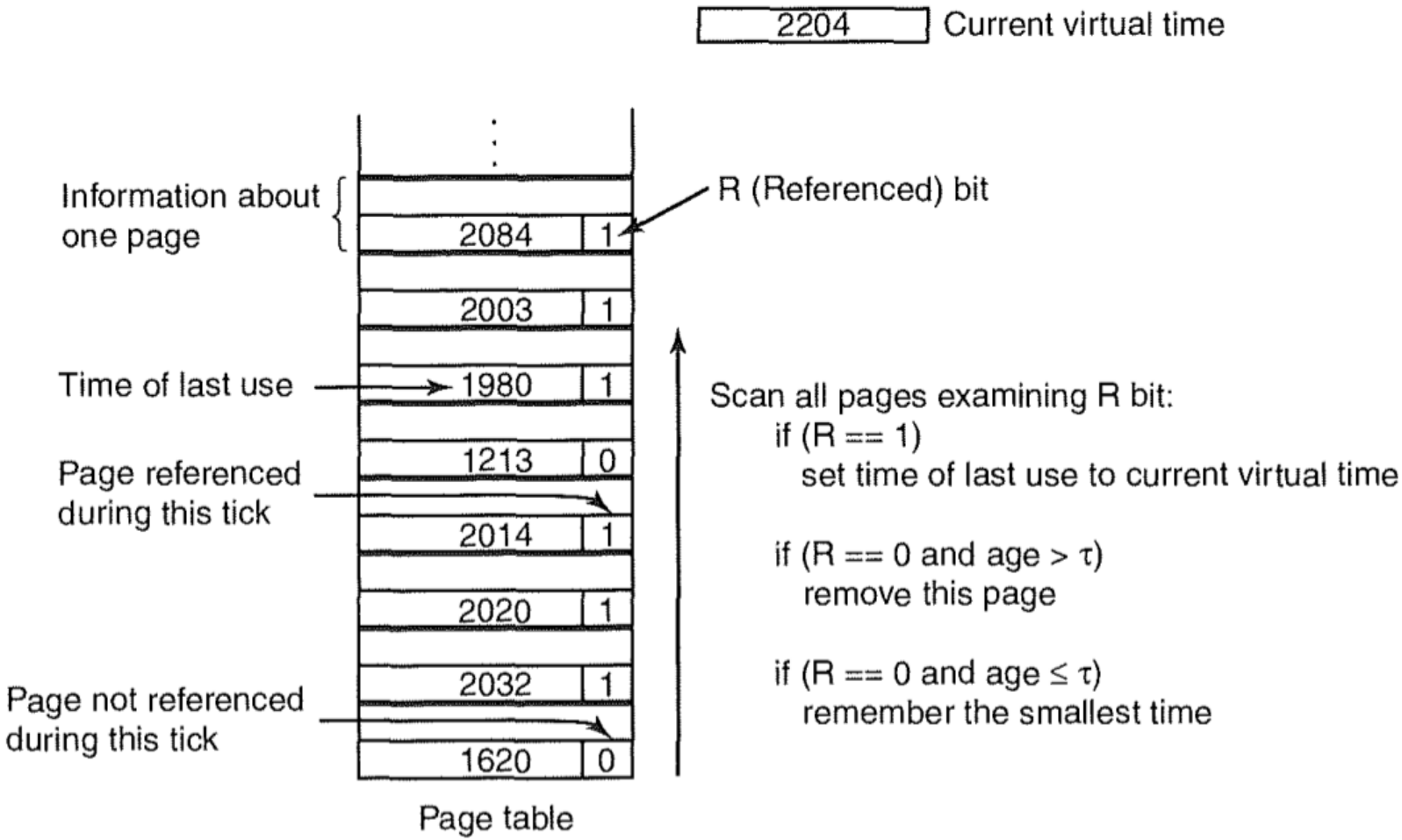
老化(aging)页面置换算法



工作集(working set)页面置换算法

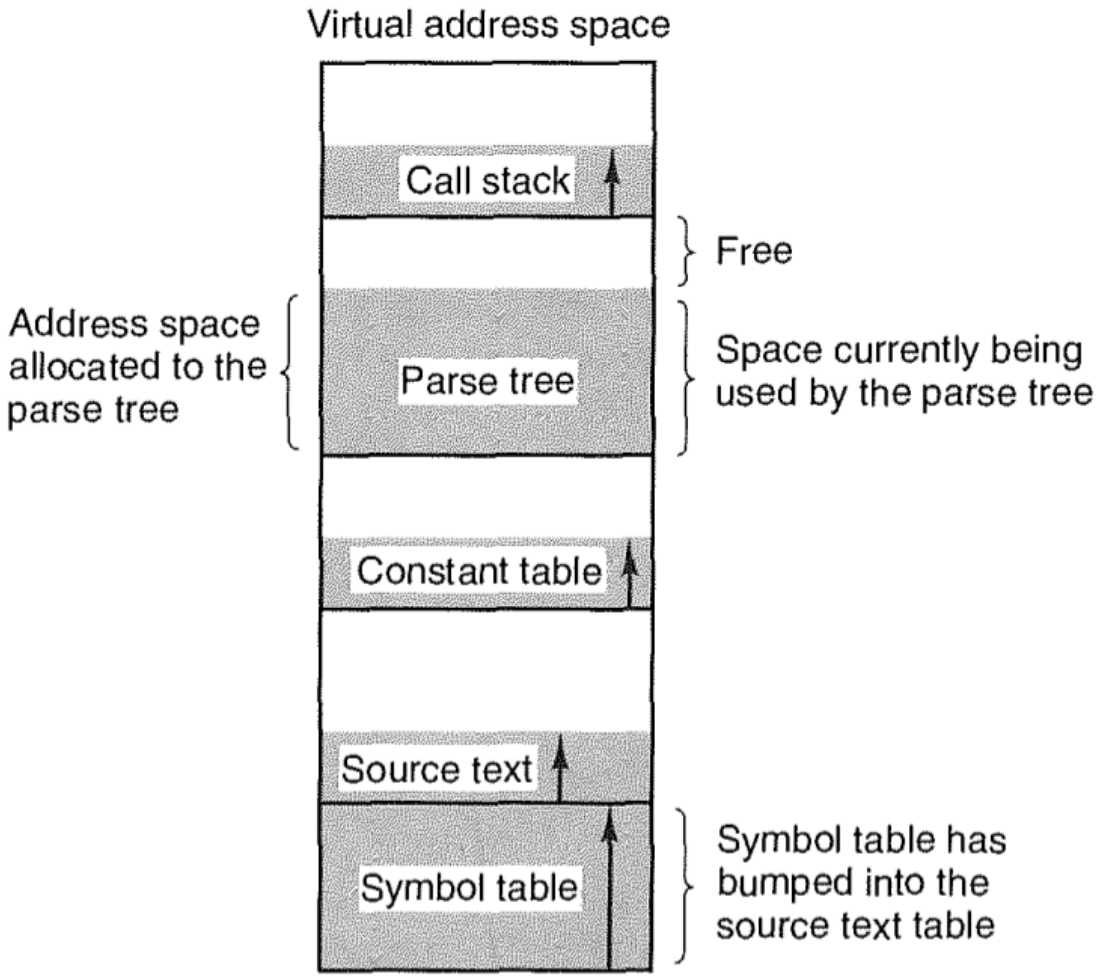
- 访问的局部性：在任一时间段内，程序仅仅访问其所有页面的一小部分。
- 我们将程序在某时间段内密集访问的页面集合成为工作集

工作集(working set)页面置换算法

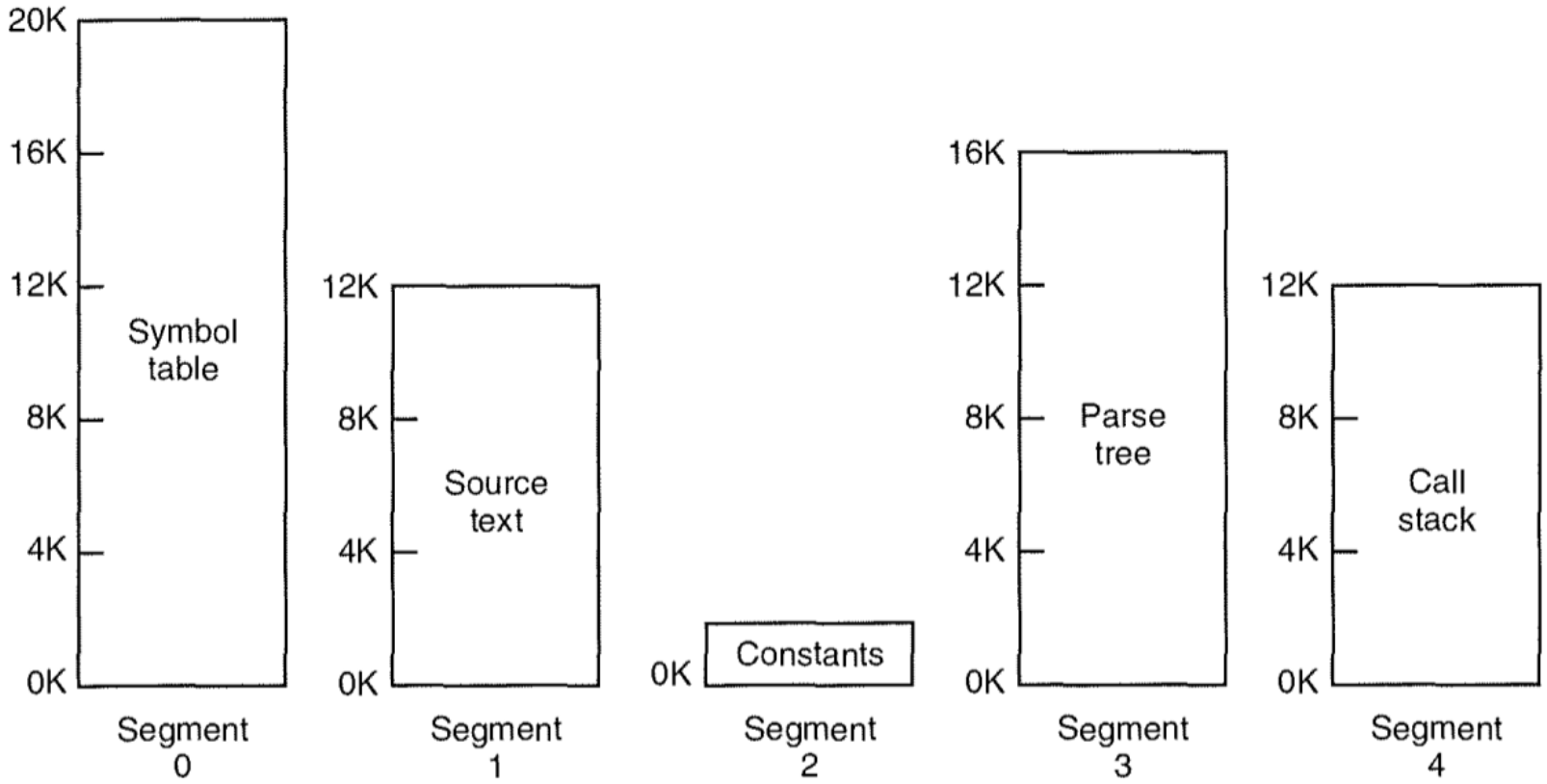


思考：如果找不到不在工作集的页面，怎么办?

单纯分页技术的缺点–分段技术的引入



单纯分页技术的缺点–分段技术的引入



每个段(segment)是独立的地址空间，互不干涉，可自如伸缩

分段技术

- 程序员（编译器）可见
- 每个段可以对应子函数、栈、数组、其它类型变量中的一种(但一般不是多种)
- 分段以后，更有利于保护（代码段–执行、数据段–读写）
- 分段以后，更便于在进程间共享代码与数据

分段与分页技术相结合— 融合二者优点

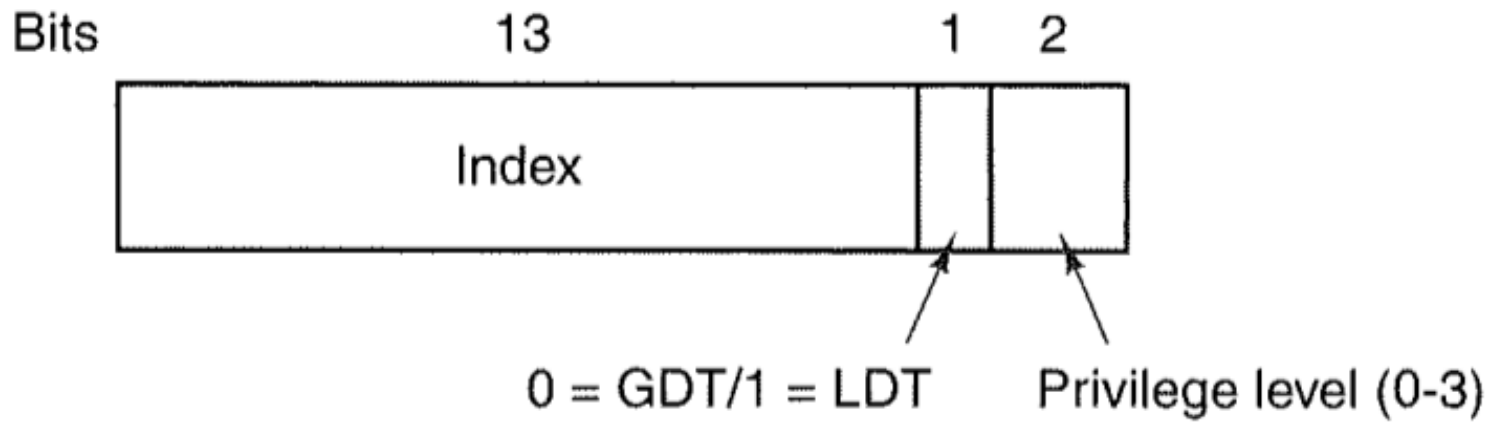
如果某个段(segment)特别大，无法全放入内存，怎么办?
答案:针对每个段，采用分页技术。即只把每段中部分页面放入物理内存

Intel平台上的分段分页技术: GDT与LDT

Intel平台上，所有段表分成两类：

1. GDT (Global Descriptor Table) – 系统段(OS), 仅1个
2. LDT (Local Descriptor Table) – 每个进程都有自己的LDT

Intel平台上的分段分页技术: 段选择符(Segment Selector)

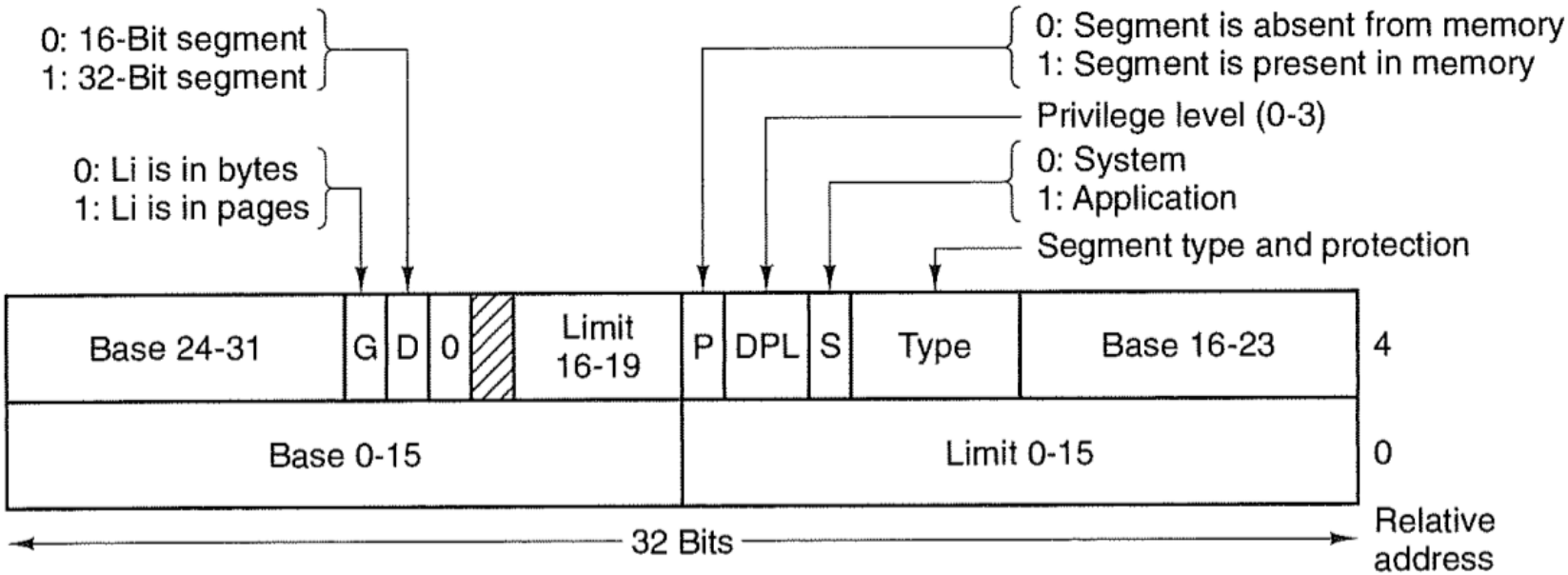


CS寄存器 存放代码段的段选择符

DS寄存器 存放数据段的段选择符

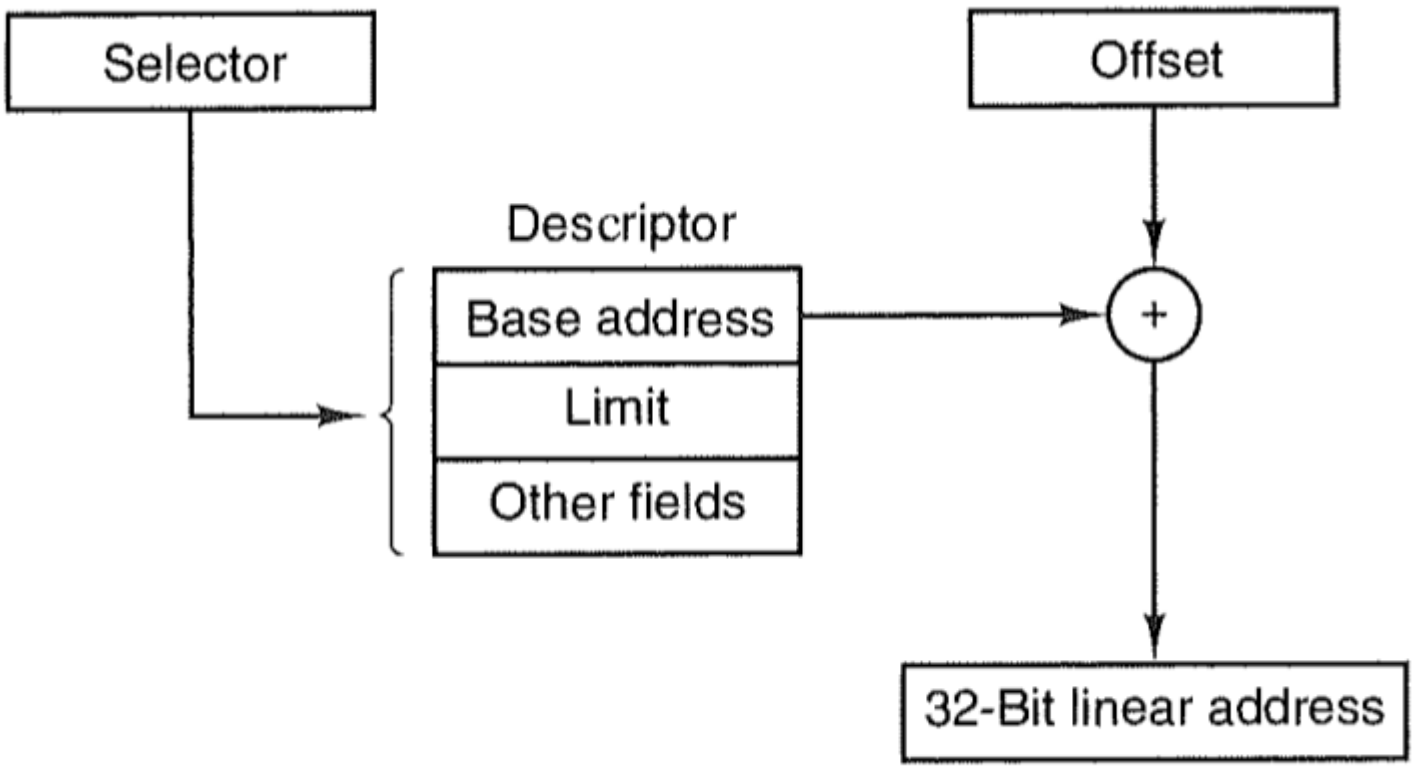
其他 四个段寄存器

Intel平台上的分段分页技术: 段描述符(Segment Descriptor) – 段表项



三个关键字段：BASE，LIMIT，G

Intel平台上的分段分页技术: 从（段选择符，偏移）到线性地址的映射



Intel平台上的分段分页技术: 从线性地址到物理地址的映射