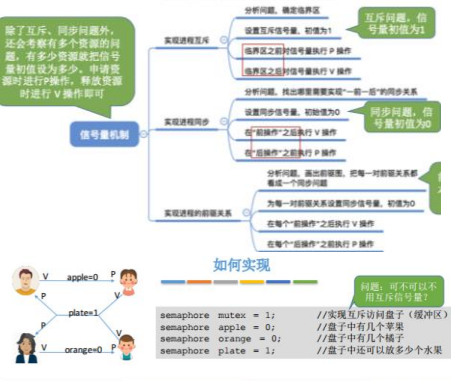


Deadlock： A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
Deadlock的条件: Mutual Exclusion, Hold and wait, No preemption, Circular wait
死锁定理：S为死锁状态的充分条件是：尚且仅当S状态的资源分配图是不可完全简化的。If graph contains a cycle ⇒ if only one instance per resource type, then deadlock; if several instances per resource type, possibility of deadlock.

Ensure that the system will *never* enter a deadlock state. (Prevention) Have a priori information on how each process will be utilizing resources, and try to avoid a deadlock state (Avoidance) A detection algorithm is ready to be invoked to determine whether a deadlock has occurred (Detection) Allow the system to enter a deadlock state and then recover. (Recovery)

安全状态 Safe State
If a system is in safe state ⇒ no deadlocks. unsafe state ⇒ possibility of deadlock.

Avoidance algorithms: Banker's algorithm

```
n:进程个数
m:资源类型的种类
```

- `Available`：长度为m的向量，表示每种资源现有的实例，如果`Available[j] =k`，说明资源类型Rj有k个实例
- `Max`：n*m矩阵，定义为每个进程的最大需求。如果Max[i][j]=k, 那么进程Pi最多可以申请k个Rj资源。行代表进程，列代表资源
- `Allocation`：n*m矩阵，定义为每个进程现在所分配的各种资源类型的实例数量。如果Allocation[i][j]=k,说明进程Pi现在已经分配了k个Rj资源
- `Need`：n*m矩阵，表示每个进程还需要的剩余的资源. `Need[i][j]=Max[i][j]-Allocation[i][j]`

i. 设Work和Finish分别为m和n的向量。按如下方式进行初始化，`Work=Available` 且对于 `i=0,1,...,n-1, Finish[i]=false`
ii. 查找这样的i使其满足
a. `Finish[i]=false`
b. $Need_i \leq Work$
如果没有这样的i,跳转到iv步
iii. `Work=Work+Allocation_i`

```
Finish[i]=true
```

相当于把资源分配给符合要求的线程，等这个线程执行完，将其资源重新加回资源池

iv. 如果对于所有的i, `Finish[i]=true`,那么系统处于安全状态

分页+分段

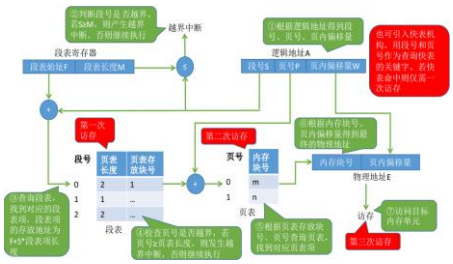


Figure 9.21 Logical to physical address translation in IA-32.

The Critical-Section Problem

Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.
Mutual Exclusion - If process Pi is executing in its critical section, then no other processes can be executing in their critical sections
Progress - If no process is executing in its critical section and some processes wish to enter their critical section, then only those processes that are not executing in their remainder sections can participate in the decision on which will enter its critical section next, and the selection of the processes that will enter the critical section next cannot be postponed indefinitely.
Bounded Waiting - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted. Assume that each process executes at a nonzero speed. No assumption concerning relative speed of the N processes.

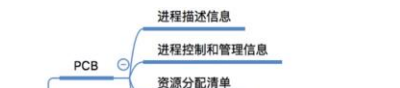
CPU Schedule

First-Come, First-Served (FCFS) Scheduling：根据就绪状态的先后来分配CPU，非抢占，最简单，利于长进程，不利于短进程，利于CPU型不利于IO型。
Shortest-Job-First (SJF) Scheduling：对预计执行时间短的作业优先分派CPU，分为抢占式：如果新来的进程时间比当前的还短，抢占，这种SJF叫做Shortest-Remaining-Time-First (SRTF)。非抢占：允许当前进程运行完再运行最短的。SJF被证明是最佳算法，它能给出最小平均等待时间。SJF是无法实现的，因为不知道下一个CPU脉冲burst时长。
HRRN(Highest Response Ratio Next): 最高响应比优先：SJF的变种，响应比R = (等待时间 + 要求执行时间) / 要求执行时间，是FCFS和SJF的折中。满足短任务优先且不会发生饥饿现象
时间片轮转调度Round Robin(RR)
通过时间片轮转提高并发性和响应时间，提高资源利用率。算法：将就绪中的进程按照FCFS排队；每次调度时将CPU分给队首进程，让其执行一个时间片；时间片结束时发生时钟中断；调度程序暂停当前进程执行将其送至就绪的队尾，然后上下文切换至新的队首；对称没用完一个时间片的话就让出CPU(如阻塞)如果就绪队列中有n个进程时间片为q，那么每个进程得到1/n的CPU时间，长度不超过q。每个进程必须等待的时间不超过(n-1)q，知道下一个时间片位置。例如5个进程，200ms时间片，那么每个进程每100ms不会得到超过20ms的时间。RR算法性能依赖于时间片大小，如果q很大，就和FCFS一样了；如果q很小，那么q也要足够大来保证上下文切换。否则开销过大。时间片长度的影响因素：响应时间一定时，就绪进程越多，时间片越小；应当使用户输入通常能在一个时间片内完成，否则相应平均周转和平均带权周转都会延长。一般来说RR比SJF有更高的平均周转，但是响应时间更好。时间片固定时，用户越多响应时间越长。

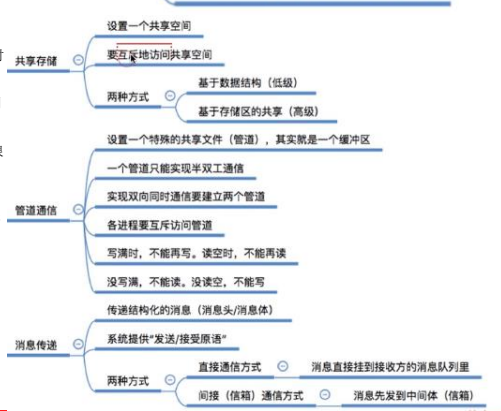
CPU utilization – keep the CPU as busy as possible
Throughput – # of processes that complete their execution per time unit
Turnaround time – amount of time to execute a particular process
Waiting time – amount of time a process has been waiting in the ready queue
Response time – amount of time it takes from when a request was submitted until the first response is produced, **not** output (for time-sharing environment)
Max CPU utilization Max throughput Min turnaround time
Min waiting time Min response time

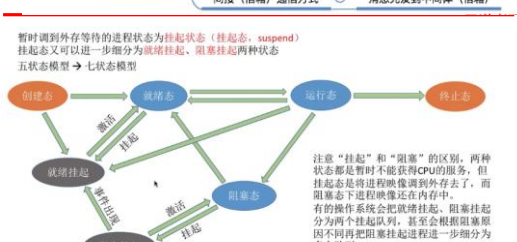
Multithreading models

Many to one：
1.many user-level threads mapped to single kernel thread
2.thread management is done in user space，used on systems that do not support kernel threads. 3.drawbacks：whole process block if one thread blocks. unable to run parallel on multiprocessors.
One to one：
1.each user-level thread maps to a kernel thread：more concurrency than many-to-one，support multiprocessors
2.drawback：overhead of creating kernel threads
3.examples：Windows 95/98/NT/2000，OS/2
Many to many：
1.multiplexes many user level threads to a smaller or equal number of kernel threads 2. allows the programmer to create a sufficient number of user threads 3. avoid bad blocking, support multiprocessors
Examples：Solaris 2，Windows NT/2000



	要做什么	调度发生在..	发生频率	对进程状态的影响
高级调度（作业调度）	按照某种规则，从后备队列中选择合适的作业将其调入内存，并为其创建进程	外存→内存（面向作业）	最低	无→创建态→就绪态
中级调度（内存调度）	按照某种规则，从挂起队列中选择合适的进程将其数据调回内存	外存→内存（面向进程）	中等	挂起态→就绪态（阻塞挂起→阻塞态）
低级调度（进程调度）	按照某种规则，从就绪队列中选择一个进程为其分配处理机	内存→CPU	最高	就绪态→运行态





硬件同步
单处理机很简单：临界区禁止中断，这样就 OK 了。

```
boolean test_and_set(boolean *target) {
    boolean rv = *target;
    *target = true;

    return rv;
}
```

```
while(1) {
    key = TRUE;
    while (key == TRUE)
        Swap(&lock,&key);
    critical section
    lock = false;
    remainder section
}
```

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

    /* critical section */

    lock = false;

    /* remainder section */
} while (true);
void Swap (boolean *a, boolean *b)
{
    boolean temp = *a;
    *a = *b;
    *b = temp;
```