

章一. OS

1. 定义: A program {
- 1. 演绎中间人. 在 **用户与硬件之间** *intermediary*
 - 2. 分配已资源者 *resource allocator* {
 - 1. 管理所有资源 *conflicting requests*
 - 2. 在相互冲突的请求中做出决定
以实现有效和公平的资源使用
 - 3. 监督电脑中所有执行 *supervise execution* {
 - 1. 正确执行
 - 2. 不影响其它程序

① CPU 时间
② 内存
③ I/O 磁盘通道 access

2. 作用 {
- 用户角度: OS 是用户与计算机硬件之间的接口.
 - 软件/程序: 提供程序接口. → system call
 - 资源管理: 计算系统的管理者

定义: OS 中资源可由内存中多个执行的进程共同使用

3. 基本特征 {
- 共享 *Share* {
 - 共享方式 {
 - 互斥共享: 打印机 → 监听资源
 - 同时访问: 对磁盘访问
 - 并行及并发 *Parallel Concurrency* {
 - 并发: 多个事件同一时间发生
 - 并行: 多 ~ 同时.

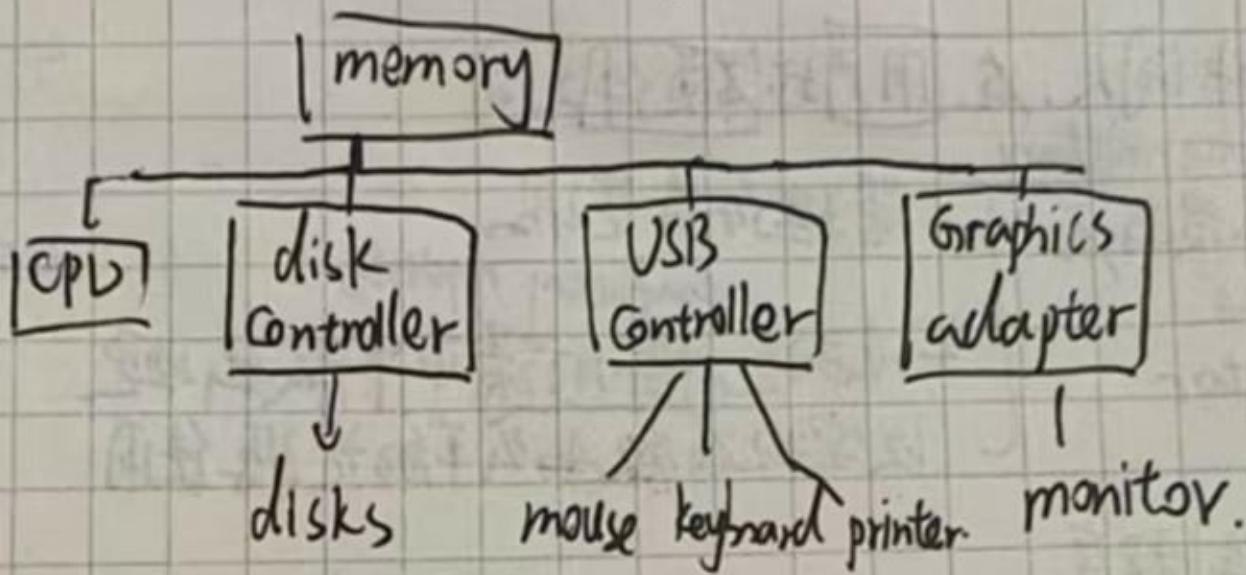
Asynchronous Synchronous

虚拟技术 {

View of Operating System Services



A modern Computer System



章一. 信号与中断

1. 定义：CPU 停止正在执行的程序，在保留现场后自动转去执行该事件的中断处理程序。执行完后返回原程序断点处继续执行

译文：一个信号通知程序，事件发生

包括 { 内核 Kernel
系统 System
应用 Application }

中断
(外中断)

2. 类型：时钟中断，控制控制中断，I/O 中断 { 硬件 | 按键
时钟中断 clock interrupt
软件 | 陷0
时间片中断 timer interrupt.

3. 陷入
(内中断) { 1. 处理器及内存内部中断
2. 网络：地址越界、访管中断
调试指令
3. 主要区别：1. 陷入 CPU 内部
2. 中断 CPU 外部

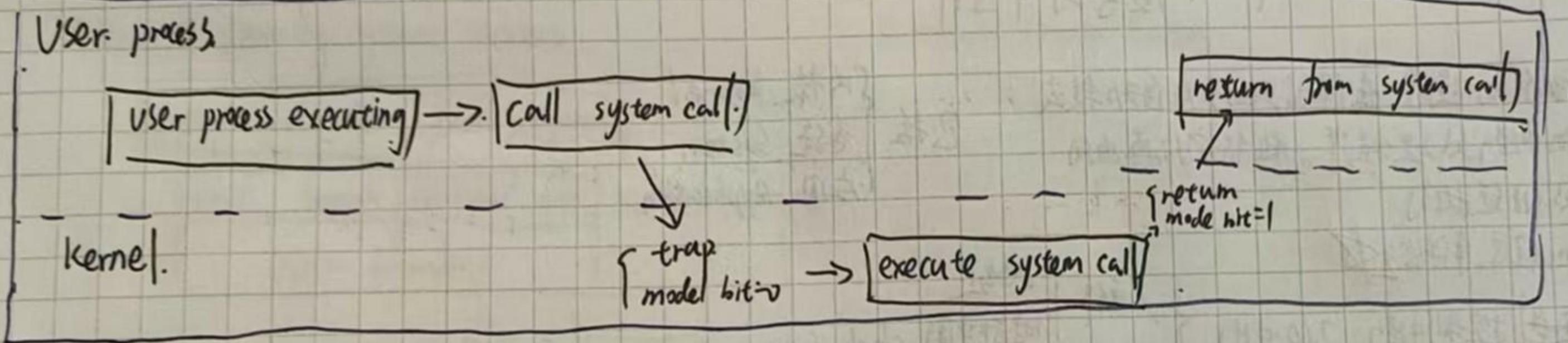
4. 信号
(软中断)

1. 停止当前程序：
CPU 返回指令的地址到 stack
2. 查找中断类型，并用这个信息跳转到 IVT
IVT 在低地址中
IVT 的入口指向 ISR
ISR 是一个小程序，也在低地址中。

5. 中断作用 {
1. 中断转移控制到 ISR
2. 中断向量 包含所有服务程序地址
vector
3. 中断机制 必须保存被中断指令的地址
4. trap 或 exception 是由错误/用户请求引起的软件生成的中断
5. OS 是中断驱动 interrupt driven

3. 执行 ISR
4. 执行前程序的下一条指令
6. 中断处理过程

User process

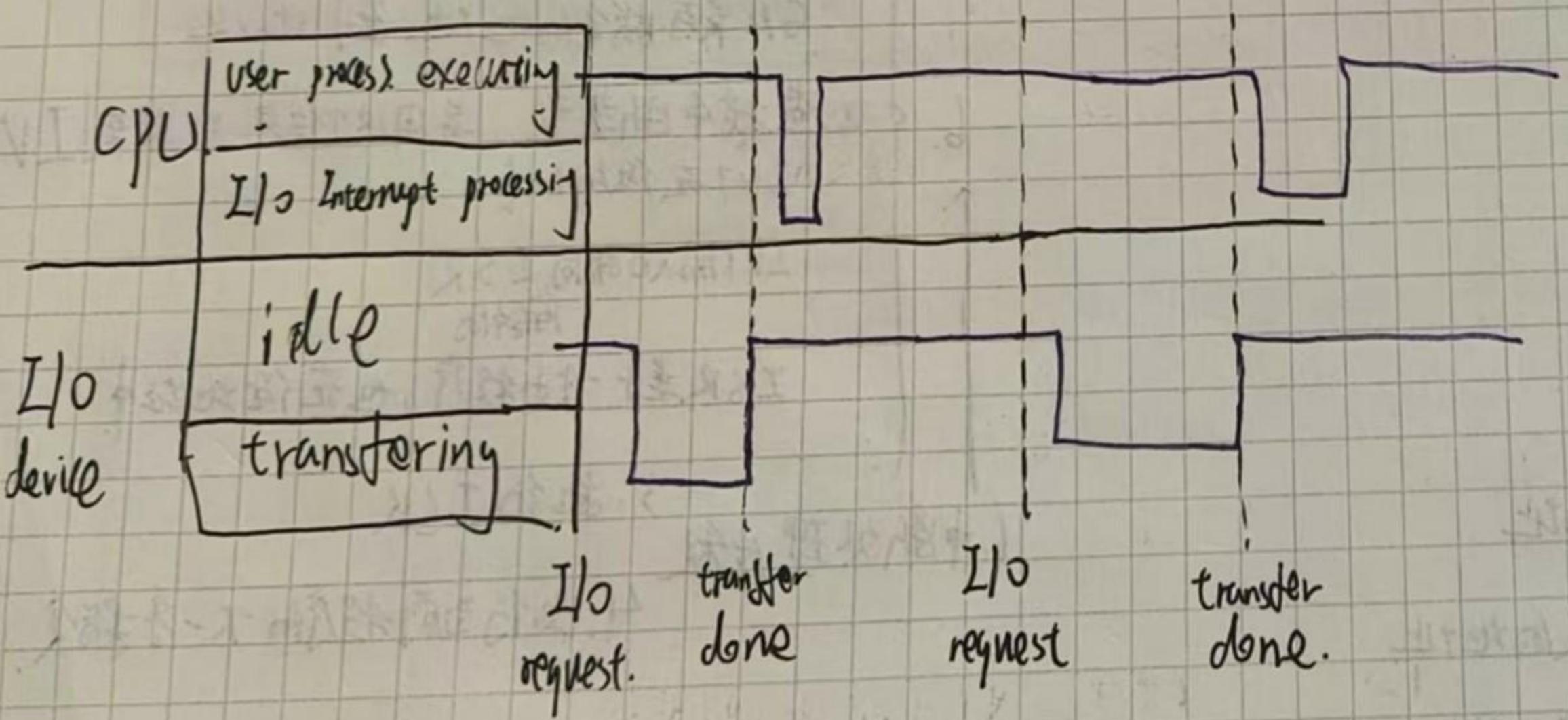


User mode : 只执行用户指令
mode bit = 1 限制内存.

kernel mode : 可执行所有指令
mode bit = 0

注意 system call 是在用户态使用

CPU & I/O 设备在处理 I/O 请求状态变化



过程：用户发出 I/O 请求

I/O 开始传输

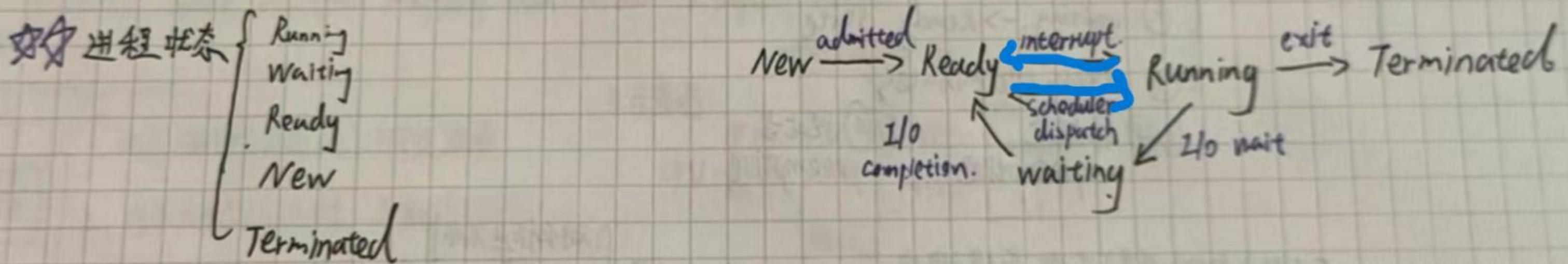
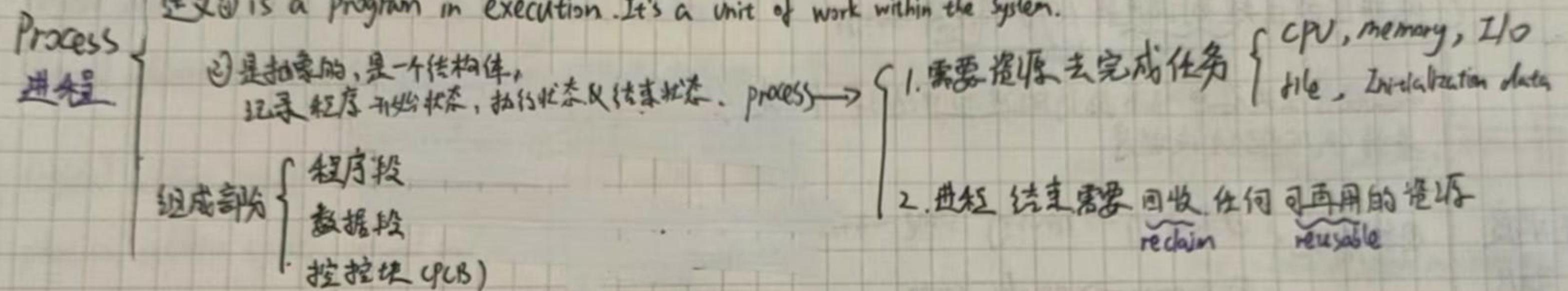
CPU 继续执行用户程序

I/O 设备传输完，并发出中断信号

CPU 接收到中断信号并处理 I/O 中断

CPU 处理完 I/O 中断，继续执行用户程序

第二章 进程管理



基本属性:

- ① 它是一个可拥有资源的独立单位.
- ② 它又是一个可独立调度和分派的基本单位.

进程调度:

- ① 多道程序 (multiprogramming): 任何时候都有些进程在运行, 最大限度提高 CPU 利用率.
- ② time sharing: 在进程之间频繁切换 CPU, 以便用户可以在 many at once with 这个程序
- ③ 为达到这些目标, 进程调度程序选择一个可用的程序在 CPU
- ④ 对单处理器来说, 只能在一个以上的它们运行. 其实叫等

CPU 调度: 当 CPU 空闲时, OS 必须从就绪队列 ready queue 选一个 process 来运行。

选择过程由短期调度器 / CPU 调度器 执行,

short term scheduler

调度程序从内存中准备执行的进程中选择一个进程,
并将 CPU 分配给该进程

CPU 调度
程序

四种情况

- ① Running → Waiting state
- ② Running → Ready state
- ③ Waiting → Ready state
- ④ process terminates.

①④ 非抢占式 ②③ 抢占式
nonpreemptive preemptive

调度

目标

- 1. 防止 process 失期不能获得调度
- 2. 提高处理器利用率
- 3. 提高系统吞吐量
- 4. 减少进程响应时间

FCFS

可用于作业调度和进程调度

利于 CPU 复杂型作业, 长作业
不利 I/O ~ ~ , 短作业

SJF

对作业不公

- 1. 未考虑作业的紧迫程度
- 2. 有意无意缩短其估计执行时间
- 3. 优先级作业运行时间

HPFS

RR

第四章：内存管理

{ logical address: 由 CPU 生成的, 也称 virtual address

physical address 硬件内存单元看到的地址 即 加载到内存地址寄存器中的地址

{ logical address space: 程序生成的所有 logical address's 集合

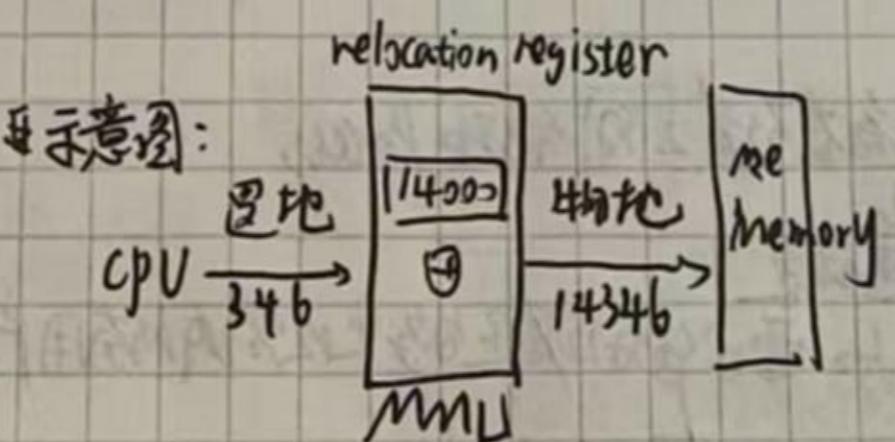
physical address space: logical addresses \rightarrow to. \sim

MMU: a device, make run-time mapping from virtual \rightarrow physical address
hardware 从 vir-add 到 phy-add 的运行时映射

内存管理单元

Memory manager: Unit.

1. 内存哪部分正在用，哪些没用
 2. 内存分配给进程，并取消分配
 3. 管理主存和磁盘间的交换



地址绑定: 重新定义: 将指令与数据绑定到内存

绑定时期 { ① Compile time: 生成逻辑地址, 如果你对 processor 在 compile time 存在哪儿没有感知, absolute address

② Load time: 生成物理地址

- ② Load time: 生成物理地址
- ③ Execution time: 如果进程可以在执行期从一个内存 $\xrightarrow{\text{物理}} \rightarrow$ 另一个内存，则使用

Swapping

定义：将进程从主存移到磁盘并移回来

时期：步骤聚
Swap in
Swap out

不同时间下做交换操作

- ① 编译期/加载时地址绑定：
进程必须被交换回它们换出时所在 相同内存位置
- ② 执行时绑定：Process 可被交换回 任何可用的内存位置

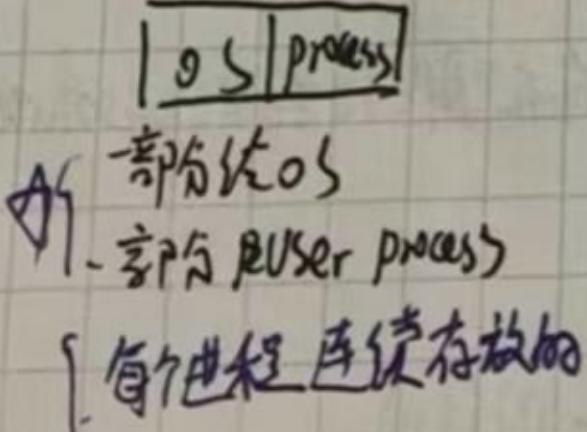
Standard swapping：主存与外存存储器之间移动 Process

↓
高速磁盘
→ 要求内存足够大以容纳所有用户的所有 内存镜像
③ 需要快。

三种内存分配方式

① 连续内存分配

memory 分配方式



Warning: 内存碎片

② 可变分区内存分配

定义：内存被划分为可变大小分区

OS会维护一个表（记录每个分区）

特殊：会合并

① 固定大小内存分配

定义：分区固定

Fragmentation

- external ~: 由于 process 的分配与释放导致内存中出现许多不连续的小空闲块，存在满足请求的总内存空间，但它不能运行。
- internal ~: 内存分配过程中，由于采用了固定大小的内存分区，导致内存块中有未被利用的使用而浪费。
解决方法：ex: Compaction (回收) 或 ex-tray

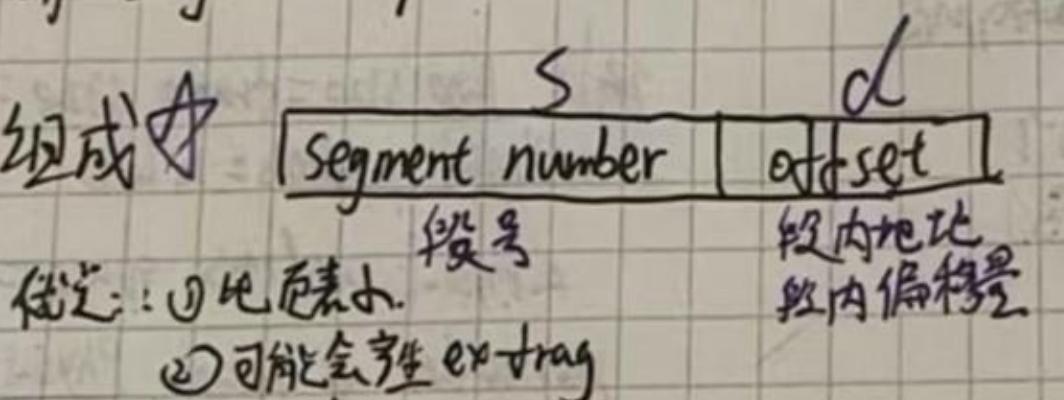
上空置
外部 ok. 但已分隔

解复用
剩余未用部分

Segmentation: 一种内存分配技术，将内存分成大小可变的块 → 给进程

分段

tip: Log-add-space 是分段的集合



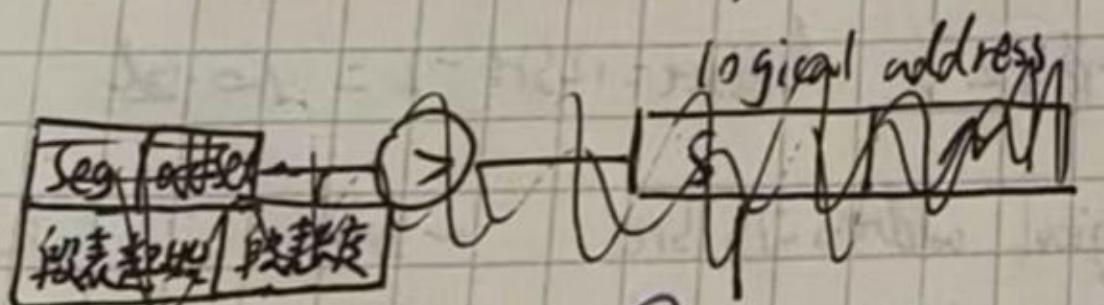
例

seg	offset
31	16 15 0

→ 每个段的最大长度 $2^6 \Rightarrow 64KB$

一个作业界长有 $2^6 \Rightarrow 64K$ 个段

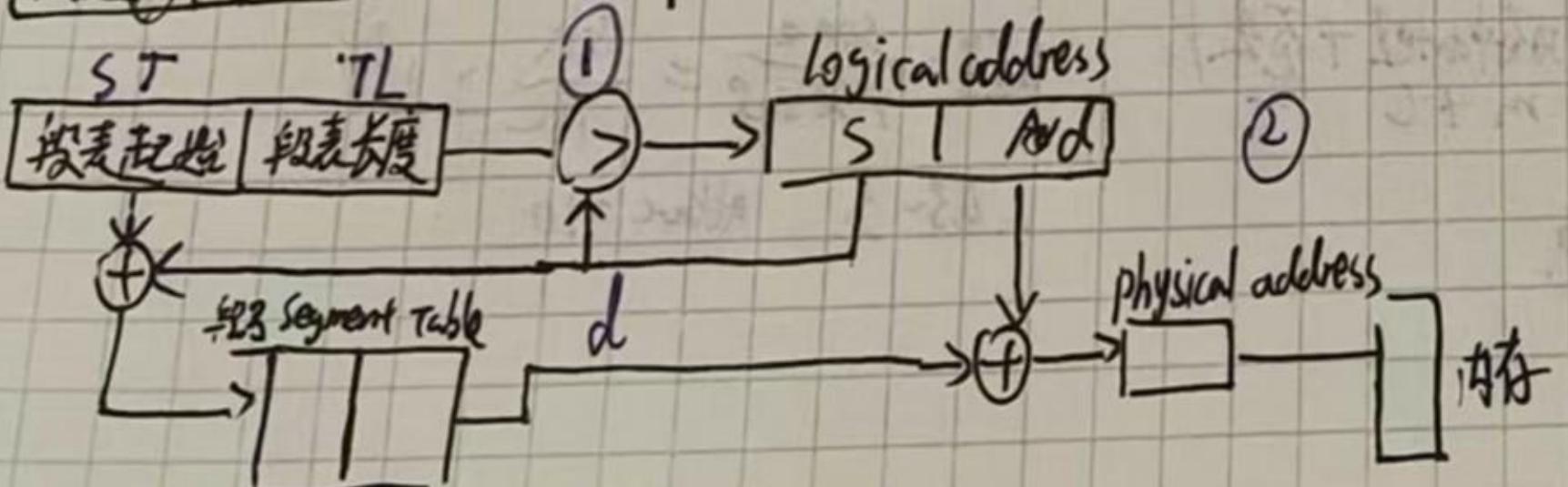
Address Translation (mapping)



① 段表长度 if > 段号(s) T: s+段表起始地址 \Rightarrow 起始地址(d) if > SL { T → 起始
if 越界中断 }

找到

{ T → 起始
if d + offset > physical address }

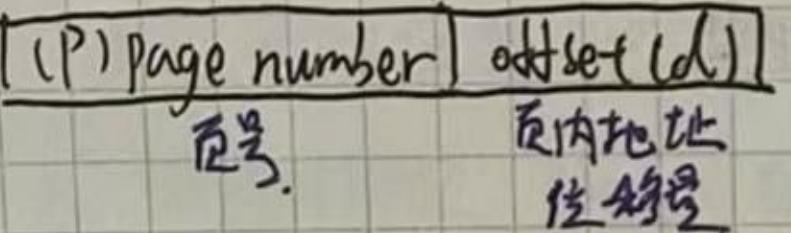


Paging 定内存分配方案，消除了对连续物理内存的需要。

- 好处 | ① 避免了外部碎片化和压缩的需要
- ② 允许物理地址不连续。

怎么搞的：将用户程序的地址空间分成页
内存空间 \rightarrow 块。页 = 块大小。

组成：



$\star P = \text{int}$ $\frac{\text{逻辑空}}{\text{页面大小}}$ 把一个空间能整分成大小为 X 的 $\frac{1}{X}$ 个

$X = \frac{\text{逻辑空}}{\text{页面大小}}$
 $\frac{1}{X}$ 个 $\frac{\text{逻辑空}}{\text{页面大小}}$

TLB:
1. 逻辑地址大小 = 页面大小。
2. 逻辑地址 = 页面大小 $\rightarrow 2^X$

EAT: $X(C+m) + (1-X)(C+m+t)$

Effective Access Time

X : TLB命中率

C : TLB access time

m : Main memory access time.

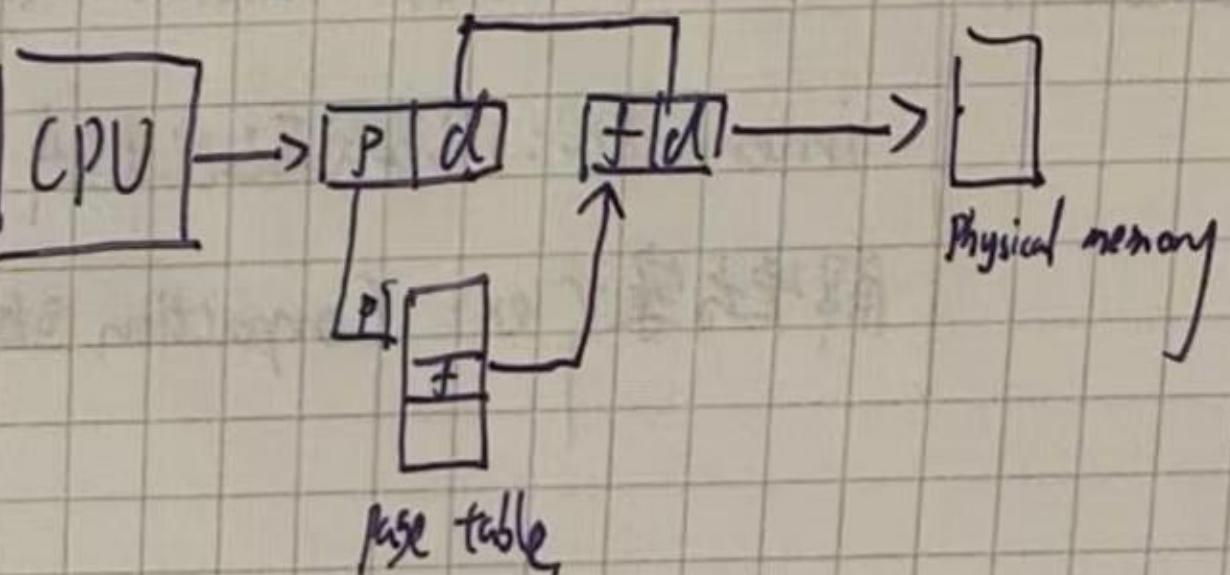
t : Page Table access time

* 有题目不给 t , 报不缺 t 用 m .

解注：有效访问时间 = 命中(快表+内存) + 命中(内存+快表)
 $\times (C+m)(1-X) m + C t$

页表常驻内存，而快表存放快存中。

logical address



例 31. 1111 0
 $\frac{31}{P} \frac{1111}{d} 0$

地址空间是 2^{32} 一个页大小为 $2^{12} = 4\text{KB}$
~~物理空间~~ $2^{32} / 4\text{KB} = 2^{20}\text{MB}$

例：Page size = frame size = 1K words
 \Rightarrow Physical address = 12 bits

\sim Physical address space = $2^{12} = 4\text{K words}$
 $\text{page number of page} = \frac{\text{Physical address}}{\text{frame page size}} = \frac{4\text{K}}{1\text{K}} = 2^{20}$

\sim 页号 $= 2^{12} - 1 = 4096$

logical address: 13 bit \sim space: $2^{13} = 8\text{K words}$

页号 = $\frac{\text{space}}{\text{page size}} = \frac{8\text{K}}{1\text{K}} = 2^3$

$\therefore 2^3 - 1 = 7$ other = 10

虚拟内存管理

virtual memory management.

Demand Paging

为什么只执行部分代码?

- ① 简单处理, 不经常发生
- ② 故障空间多余
- ③ 废弃功能

好处:

- ① 让程序不再受物理内存数量限制
用户能够为极大的虚拟地址空间
写程序, 简化了编程任务。
- ② 每个用户程序, 可占更少物理内存.
同时运行更多程序。
- ③ CPU利用率与吞吐量增加, 但响应时间 & 周转时间不变
- ④ ~~减少了? 2倍~~ 加载交互少, 运行更快

因此引出虚拟内存 定义: 内存管理技术, 允许执行可能不完全
在主存且不需要连续内存分配的过程

优点

- ① 程序不再受物理内存量限制
multitasking degree
- ② 增加了 多路多道程序的程度.
- ③ 减少由于 swapping 的 overhead
可忽略

第六章 Synchronization Tools 同步

Race condition: 定义：多个进程/线程并发访问和操作共享资源时
如果执行顺序不同会导致不同的结果。

临界资源 critical resource CR : 进程使用它们的时候，需要采用互斥方式的资源。 { 硬件：打印机
软件：共享变量，文件

临界区 critical section CS : 每个进程中运行的那段代码

CS solution 的三种要求：

- ① Mutual exclusion: 如果有 Process 正在执行 CS，其它 Process 都不能执行它们的 CS
- ② Progress Progress: 如果没有 Process 执行 CS，且有些有在的 Process 想进入 CS，那么下一个进入 CS 的 Process 不能无限期地推迟
- ③ Bounded waiting: 在一个 P 发出进入 CS 的请求后，并且在 request 被允许前，其它进程被先叫进入它们的 CS 体有一个上界。
次数 限制

人类征服同步的历史

① 先提出软件思想

while(true)

```
{ while(turn==i){  
    if(CS)  
        turn=i;  
    RS  
}
```

解决了互斥 ✓

Progress X

有限等待 X

② 提出 Peterson

基于①

```
while(true)  
    flag[i]=true;  
    turn = i;  
    while(flag[i] && turn==i)  
    {  
        if(CS)  
            flag[i]=false;  
        RS  
    }
```

int turn => 是轮到 P_i 了

boolean flag[i] 判断 P_i 是否准备进入 CS

Peterson 解决了互斥问题。

但是在现代电脑中，可能没有 Memory barrier

而导致多个 P 进入 CS 中

图解 $P_0 \rightarrow turn=0 \rightarrow flag[0]=true \Rightarrow CS$
 $P_1 \rightarrow turn=1 \rightarrow flag[1]=true \Rightarrow CS$

③ 因此有 Memory Barrier

在 memory model { Strong ordered : 改后立刻看到 \Rightarrow Memory Barrier.
weakly ordered : 改后不是立刻

定义：是一种指令，强制内存中的任何更改
传播给其它处理器

④ 在硬件方面

Hardware Instruction { Test-and-Set
Compare-and-Swap
Atomic variables

Atomic variables (-般基于 Compare-and-Swap 实现)

void increment (atomic_int *V)

{ int temp;
do { temp = *V }

while (temp != compare_and_swap (V, temp, temp+1))

作用：确保 V 连续而不中断 - 从用法 increment (&sequence)

Test-and-Set 源代码

boolean T-A-S (boolean *target)
{ boolean rV = *target; // 用法：返回参数
if (*target == true); // 并
return rV; } // 将参数改为true.

Compare-and-Swap 代码

int compare_and_swap (int *value, int expected, int new_value)
{ int temp = *value; // 自动执行
if (*value == expected) *value = new_value; // 用法：返回 value 的原值
return temp; }

⑤ 如果 $V=0 \Rightarrow V=New$

Mutex Lock / spinLock
以前，太慢了，为了使人用，发明了这个

{ acquire() 申请
release() 释放锁.

问题：出现忙等待

liveness 性质。

定义：系统必须满足的性质，以确保P能够取得进展

deadlock:

两个P正在无限期等待一件事，而这件事
可能是由其中一个等待进程引起。
原因：优先级P拥有高优先级
{ starvation
P永远不会起始运行队列中执行
block();]
priority Inversion }

Monitors 管理。

定义：一种抽象数据类型，内部变量只能由 Monitor 的代码访问

规则：任何时候 Monitor 内只能有一个进程处于活跃状态

P220 另一个定义

代表共享资源的数据体，以及由对该共享数据结构实施 资源管理操作
操作的一组函数组成向资源管理程序 共同构成一个对象类化

Semaphore 信号量

具体实现 { wait() / P() → while(s <= 0) busy wait;
signal() / V() s--;]
S 为 Semaphore. → signal(s) / s++;

实现 信号量 实现无忙等 ☆

{ block() → P 放在等待队列上
wake up() → 从等待队列中移除一个 P，放入就绪队列

具体方法 wait(Semaphore *s)

{ s->value --;
if(s->value < 0)

{ add this process to S->List;

signal (Semaphore *s)

{ s->value ++;
if(s->value <= 0)

{ remove this process from S->List;
wake up(p);] }

Condition Variables

存在理由：因为进程因不同的原因被挂起，所以需要 Condition Variable

第七章 同步与线程

① 生产者-消费者问题

也称：有界缓冲区问题

基本概念

mutex	：初始为1， \leftarrow 一段时间只能一人取/存				
Semaphore	<table border="0"> <tr> <td>full</td> <td>：始值为0 \leftarrow限制 producer 有 \rightarrow 有资源车辆</td> </tr> <tr> <td>empty</td> <td>：始值为n, \leftarrow Pn 个 consumer 取</td> </tr> </table>	full	：始值为0 \leftarrow 限制 producer 有 \rightarrow 有资源车辆	empty	：始值为n, \leftarrow Pn 个 consumer 取
full	：始值为0 \leftarrow 限制 producer 有 \rightarrow 有资源车辆				
empty	：始值为n, \leftarrow Pn 个 consumer 取				
n	：固定大小的缓冲区，缓冲区可用数量，翻转为“空” empty(0) = 满了，有没有空				

具体实现：生产者

```

while(true)
{
    // 生产者做事
    wait(empty);
    wait(mutex);
}

```

顺序不同

```

// 消费者吃
signal(mutex);
signal(empty);
}

```

消费者

```

while(true)
{
    wait(full);
    wait(mutex);
}

```

顺序不同

```

// 生产者放
signal(mutex);
signal(full);
}

```

② 读写问题

Semaphore

rw-mutex	：1 \rightarrow 控制对 Shared data 的读写互斥访问
mutex	：1 保证它变量操作时的互斥访问
read_count	：0：读者数量（正在）

写者

```

while(true)
{
    wait(rw-mutex);
}

```

写者写... CS

```

signal(rw-mutex);
}

```

问题：会导致饿死

读者

```

while(true)
{
    wait(mutex); // 为了互斥使用 read_count
    read_count++; // 第一个读者
    if (read_count == 1) // 第一个读者
        wait(rw-mutex); // 读者是第一个
    // 多读者读取  $\rightarrow$  signal(mutex);  $\leftarrow$  基本锁
}

```

读者读... CS

如果是最
后一个
读者

```

wait(mutex); // 同上
read_count--;
if(read_count == 0) // 非常数据
    signal(rw-mutex);
    signal(mutex);
}

```

哲学家用餐问题

用管程来解决哲学家用餐

```
monitor Diningphilosophers  
{  
    enum {Thinking Hungry Eating} state [5];  
    condition self [5];  
    void pickup (int i)  
    { state [i] = Hungry;  
        test (i);  
        if (state [i] != Eating) self [i].wait; }  
  
    void putdown (int i)  
    { state [i] = Thinking;  
        test ((i+4)%5);  
        test ((i+1)%5); } }
```

Void test (int i)

{ id (state [(i+4)%5] != Eating) \wedge state [i] = Hungry \wedge state [(i+1)%5] != Eating)
 | state [i] = Eating \rightarrow 去吃
 self [i].signal(); } ← ?
}

判断左边边人是否吃饭

右边边人

用法简单

Diningphilosophers . pickup (i);

吃饭

~ . putdown (i); 不会死锁，但可能 starvation,

章八. 死锁

什么是死锁：两个P希望其它的P能释放自己所需的Res，但它又因不能获得所需的Res，而无法释放自己占有Res

~~活锁~~：两个P在执行过程中，因争夺资源而造成的互相等待的现象

活锁：P在未被 blocked 的情况下，仍然重复执行某个相同动作，从而无法继续执行后面内容

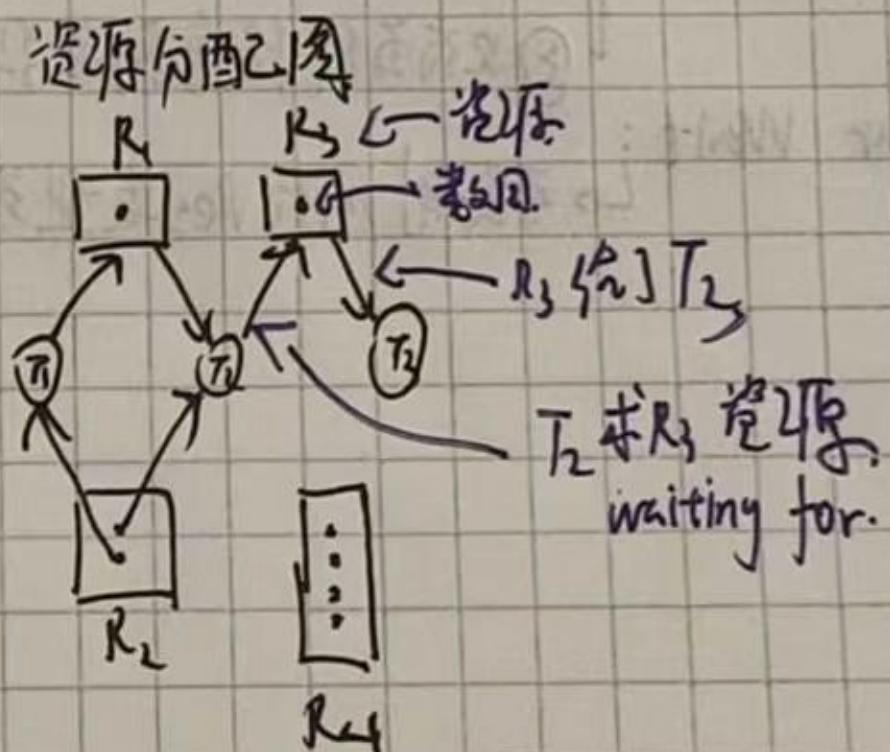
死锁产生四个条件

① ~~互斥~~: 只有一个线程可以使用一个资源

② Hold and Wait: 一个线程拥有至少一个Res 并且等待
其它线程拥有的其它 Res

③ No-preemption: 一个Res 只能被它所占有的线程，在完成
任务后，主动释放

④ 紧 Circular wait: To be Res 被 T₁占. T₁向 Res 申请. T₁向 Res 申请



① 无循环不死锁

② 有循环 { 一个资源一个空闲 } → 死锁

一个资源多个空闲 → 可能死锁

- 预防死锁
- Mutual Exclusion: 共享资源不需要互斥，非共享Res必须持有
 - Hold and Wait: 保证当一个T执行一个操作时，它不能有其它的Res
 - No preemption:
 - ①如果T拥有Res，并求其它Res且求不到，应释放Res.
 - ②被抢占的Res被添加到 waiting line
 - ③只有前线程重新获得它的旧资源并且它请求新Res，T才重新启动。
 - Circular Wait:
 - 强制所有Res类型总排序，并要求每个T以递增的枚举顺序请求Res.