计算机操作系统

1.1.1 操作系统的概念、功能

操作系统(OS)：是指控制和管理整个计算机系统的硬件和软件资源，并合理地组织调度计算机的工作和资源的分配；以提供给用户和其他软件方便的接口和环境；是计算机系统中最基本的系统软件。

功能和目标

- 资源的管理者
  - 处理机管理
  - 存储器管理
  - 文件管理
  - 设备管理
- 向上层提供服务
  - 普通用户
    - GUI 用户图形界面
    - 命令接口
      - 联机命令接口
      - 脱机命令接口
  - 软件级程序员用——程序接口——系统调用

- 对硬件机器的扩展
  - 扩充机器，(虚拟机)

1.1.2 操作系统的特征

1. 并发　2. 共享　3. 虚拟　4. 异步

并发：两个或多个事件在同一时间间隔内发生。这些事件宏观上是同时发生的，但微观上是交替发生的。

并行：两个或多个事件在同一时间发生。

操作系统的并发性指计算机系统中"同时"运行着多个程序。

单核CPU：同一时刻只能执行一个程序，各个程序只能并发地执行。

多核CPU：同一时刻可以执行多个程序，多个程序可以并行地执行。

共享：资源共享，指系统中的资源可供内存中多个并发执行的进程共同使用。

1. 互斥共享方式
   - 一个时间段只允许一个进程访问该资源。(QQ、微信视频使用摄像头)
2. 同时共享方式

   允许一个时间段内由多个进程"同时"对它们进行访问。(QQ、微信发送信息)

虚拟：

把一个物理上的实体变为若干个逻辑上的对应物。

物理实体是实际存在的，而逻辑上的对应物是用户感受到的。

异步：在多道程序环境下，运行走走停停，以不可知的速度向前推进。

1.2 操作系统的发展与分类

1) 手工操作阶段

缺：人机速度矛盾

2) 批处理阶段

- 单道批处理系统

  优：缓解了人机速度矛盾

  缺：资源利用率依然很低。
- 多道批处理系统

  优：多道程序并发执行，资源利用率大大提高。

  缺：不提供人机交互功能。

3) 分时操作系统

优：提供人机交互功能

缺：不能优先处理紧急事务

4) 实时操作系统

- 硬实时系统

  必须在绝对严格的规定时间内处理
- 软实时系统

  能接受偶尔违反时间规定。

优点：能优先处理紧急事务。

1.3.1 操作系统的运行机制

- 普通程序员写的程序是"应用程序"
- 微软、苹果有一帮人负责实现操作系统，他们写的是"内核程序"，由很多内核程序构成了"操作系统内核"，它们"内核"

- 内核是操作系统最重要最核心的部分，也是最接近硬件的部分。
- 在CPU设计和生产的时候就划分了特权指令和非特权指令，因此CPU执行一条指令前就能判断出其类型。
- CPU有两种状态

  内核态、用户态

  处于内核态时，正在运行内核程序，可执行特权指令。

  处于用户态时，正在运行应用程序，可执行非特权指令。
- 内核态、用户态的切换

  内核态→用户态：执行一条特权指令——修改PSW的标志位为"用户态"，这个动作意味着操作系统将主动让出CPU使用权。

  用户态→内核态：由"中断"引发，硬件自动完成变态过程，向系统发中断信号意味着操作系统将强行夺回CPU的使用权。

1.3.2 中断和异常

- 中断的类型

  内中断：与当前执行的指令有关，中断信号来源于CPU内部。

  外中断：与当前执行的指令无关，中断信号来源于CPU外部。
- 内中断

  例1：试图在用户态下执行特权指令

  例2：试图执行非法的信号(除数为0)

  例3：有时间应用程序想请求操作系统内核的服务，此时会执行一条特殊的指令——陷入指令，该指令会引发一个内部中断信号。
- 外中断

  例1：时钟中断：由时钟部件发来的中断信号

  例2，I/O中断：由输入/输出设备发来的中断信号。
- 异常：陷阱、陷入、故障

- 中断机制的基本实现原理

  内中断：CPU在执行指令时，会检查是否有异常。

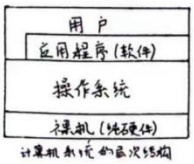
  外中断：每个指令周期末尾，CPU都会检查是否有外中断信号要处理。

1.3.3 系统调用

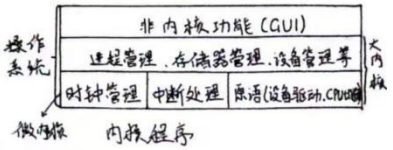
- 系统调用是操作系统提供给应用程序使用的接口，理解为应用程序可以通过系统调用来请求获得操作内核的服务。
- 凡是与共享资源有关的操作(如存储分配、I/O操作、文件管理)都必须通过系统调用的方式向操作系统内核提出服务请求。
  - 设备管理：请求、释放、启动
  - 文件管理：读、写、创建、删除
  - 进程控制：创建、撤销、阻塞、唤醒
  - 进程通信：消息传递、信号传递
  - 内存管理：分配、回收
- 系统调用的过程

  传递系统调用参数 → 执行陷入指令(用户态) → 执行相应的内核程序处理系统调用(核心态) → 返回应用程序
- 陷入指令 = trap指令 = 访管指令

1.4 操作系统体系结构



计算机系统的层次结构



- 大内核

  将操作系统的主要功能模块都作为系统内核，运行在核心态。

  优点：高性能

  缺点：内核代码庞大，结构混乱
- 微内核

  只把最基本的功能保留在内核。

  优点：内核功能少，结构清晰。

  缺点：频繁在核心态和用户态之间切换，性能低。
- 分层结构

  最底层是硬件，最高层是用户接口。

  每一层可单向调用更低一层提供的接口。

  优点：便于调试和验证。

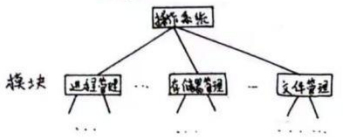
  缺点：效率低，不可跨层调用
- 模块化

  将内核划分为多个模块，各模块之间相互协作。

  内核=主模块+可加载内核模块

  主模块：负责核心功能，如进程调度、内存管理。

  可加载内核模块：动态加载新模块到内核。



- 外核

  内核负责进程调度、进程通信等。

  外核负责为用户进程分配未经抽象的硬件资源，且由外核负责保证资源使用安全。

1.5 操作系统引导

1.6 虚拟机

2.1.1 进程的概念、组成、特征

- 进程的概念

  程序：是静态的，就是个存放在磁盘里的可执行文件，就是一系列的指令集合。

  进程：是动态的，是程序的一次执行过程。
- 进程的组成——PCB

  当进程被创建时，操作系统会为该进程分配一个唯一的、不重复的"身份证号"——PID(Process ID)进程ID

  操作系统要记录PID、进程所属用户ID、分配资源、对实现操作系统对资源的管理，可用于实现操作系统对进程的控制、调度。这些信息被保存在一个数据结构PCB中，即进程控制块。

PCB
- 进程描述信息
  - 进程标识符PID
  - 用户标识符UID
- 进程控制和管理信息
  - CPU、磁盘、网络流量使用情况
  - 进程当前状态：就绪/阻塞/运行
- 资源分配清单
  - 正在使用"打开"文件
  - 正在使用"申请"内存区域
  - 正在使用"申请"I/O设备
- 处理机相关信息：如PSW、PC等，用于实现进程的切换

进程的组成
- PCB
- 程序段：程序的代码(指令序列)
- 数据段：运行时产生的各种数据

PCB是给操作系统用的。

程序段和数据段是给进程自己用的。

进程实体(进程映像)是静态的。

- 进程的特征

  动态性：进程是程序的一次执行过程，是动态产生、变化、消亡的。

  并发性：内存中有多个进程实体，各进程可并发执行。

  独立性：进程是能独立运行、独立获得资源、独立接受调度的基本单位。

异步性：按各自独立的、不可预知的速度向前推进，操作系统提供"进程同步机制"来解决异步问题。

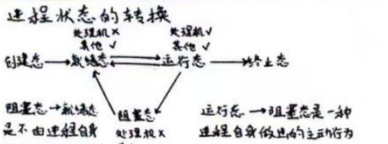
结构性：每个进程配置一个PCB，结构上看，进程由程序段、数据段、PCB组成。

2.1.2 进程的状态与转换

进程的状态

- 创建态：进程正在被创建时，它的状态是"创建态"，在这个阶段操作系统会为进程分配已资源、初始化PCB。
- 就绪态：当进程创建完成后，便进入"就绪态"，处于就绪态的进程已经具备运行条件，但由于没有空闲的CPU，就暂时不能运行。
- 运行态：如果一个进程此时在CPU上运行，那么这个进程处于"运行态"。
- 阻塞态：在进程运行过程中，可能会请求等待某个事件的发生。在进程无法继续往下执行，操作系统会让这个进程下CPU，进入"阻塞态"。
- 终止态：执行exit系统调用，请求操作系统终止该进程。此时进程会进入"终止态"。将进程下CPU回收内存空间，回收该进程PCB

进程状态的转换



不能由阻塞态直接转换为运行态。

也不能由就绪态直接转换为阻塞态。

- 进程状态间的转换

  就绪态→运行态：进程被调度

  运行态→就绪态：时间片到，CPU被其他高优先级的进程抢占。

  运行态→阻塞态：等待系统资源分配，等待某事件发生(主动行为)

  阻塞态→就绪态：资源分配到位，等待的事件发生。

  创建态→就绪态：系统完成创建进程相关的工作。

  运行态→终止态：进程运行结束，运行过程中遇到不可修复的错误。

2.1.3 进程控制

- 进程控制就是实现进程状态转换

进程控制：原语，具有"原子性"，一气呵成。

通过用"关中断指令"和"开中断指令"这两个特权指令实现原子性。

- 进程的创建：

  创建原语

  事件：用户登录、作业调度、提供服务、应用请求
- 进程的终止：

  撤销原语

  事件：正常结束、异常结束、外界干预
- 进程的阻塞：

  阻塞原语

  事件：等待系统分配某种资源、等待相互合作的其他进程完成工作
- 进程的唤醒原语

  事件：等待的事件发生

- 进程的切换

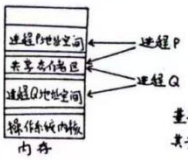
切换原语：运行态→就绪态
就绪态→运行态

事件：当前进程时间片到
有更高优高级的进程到达
当前进程主动阻塞
当前进程终止.

2.1.4 进程通信

- 进程通信：两个进程之间产生数据交互.
- 通信方式：

1. 共享存储



基于数据结构：低级通信
基于存储区：高级通信

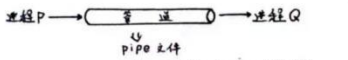
2. 消息传递

进程间的数据交换以格式化的消息为单位.
进程通过操作系统提供的"发送消息/接收消息"两个原语进行数据交换.

消息头	消息头包括：发送进程ID、接受进程ID、消息长度
消息体	

直接通信方式：指明进程的ID
间接通信方式：通过"信箱"间接的通信.

3. 管道通信



大小固定的内存缓冲区（队列）

管道采用半双工通信，某一时段内只能实现单向的传输
各进程互斥地访问管道.
管道写满时，写进程将阻塞
管道读空时，读进程将阻塞

2.1.5 线程的概念

线程

线程是一个基本的CPU执行单元，也是程序执行流的最小单位.

引入线程之后，不仅是进程之间可以并发，进程内的各线程之间也可以并发，从而进一步提升了系统的并发度.

引入线程后，进程只作为除CPU之外的系统资源的分配单元.

- 引入线程机制后，有什么变化？

1. 资源分配、调度

传统进程机制中，进程是资源分配、调度的基本单位.

引入线程后，进程是资源分配的基本单位，线程是调度的基本单位.

2. 并发性

传统进程机制中，只能进程间并发

引入线程后，各线程间也能并发，提升了并发度.

3. 系统开销

传统的进程间并发，需要切换进程的运行环境，系统开销很大.

线程间并发，如果是同一进程内的线程切换，则不需要切换进程环境，系统开销小.

引入线程后，并发所带来的系统开销减小.

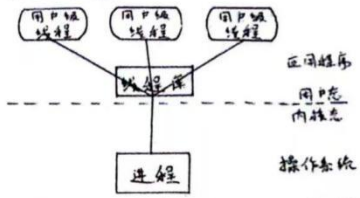
线程的属性

1. 线程是处理机调度的单位.
2. 多CPU计算机中，各个线程可占用不同的CPU.
3. 每个线程都有一个线程ID、线程控制块（TCB）.
4. 线程也有就绪、阻塞、运行三种基本状态.

5. 线程几乎不拥有系统资源
6. 同一进程的不同线程间共享进程的资源.
7. 由于共享内存地址空间，同一进程间通信甚至无需系统干预
8. 同一进程中的线程切换，不会引起进程切换.
9. 不同进程中的线程切换，会引起进程切换.
10. 切换同进程内的线程，系统开销很小.
11. 切换进程，系统开销较大.

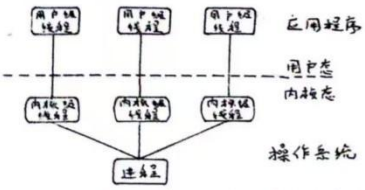
2.1.6 线程的实现方式/多线程模型

用户级线程



1. 所有线程管理工作都由应用程序实现，所有线程切换在用户态下完成.

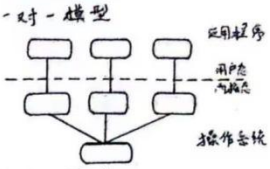
内核级线程



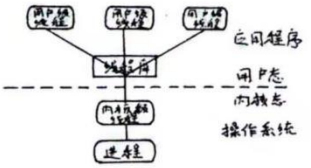
1. 内核级线程的管理工作由操作系统内核完成.

多线程模型

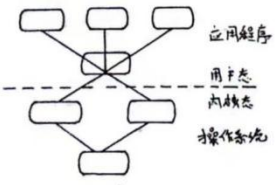
- 一对一模型



- 多对一模型



- 多对多模型



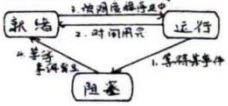
- 优缺点

一对一：优：各个线程可分配到多核处理机并行执行，并发度高.
缺：线程管理都需操作系统，开销大

多对一：优：线程管理开销小，效率高.
缺：一个线程阻塞会导致整个进程都被阻塞

多对多：集二者之所长.

- 线程的状态与转换



3.1 调度的概念、层次

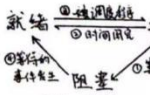
- 调度的层次

1. 高级调度（作业调度）：
按一定的原则从外存的作业后备队列中挑选一个作业调入内存并创建进程.
2. 低级调度（进程调度）：
按照某种策略从就绪队列中选取一个进程，将处理机分配给它.
3. 中级调度（内存调度）
按照某种策略决定将哪个处于挂起状态的进程重新调入内存.
内存不够时，可将某些进程的数据调出外存.

3.2 进程调度的方式

- 非抢占方式：在运行进程中即使有更紧迫的任务到达，当前进程仍然会继续使用处理机，直到该进程终止.
- 抢占式：当一个进程正在处理机上执行时，如果有一个更重要或更紧迫的进程需要处理机，则立即暂停正在执行的进程，将处理机分配给更重要紧迫的那个进程.

3.3 调度器/调度程序



②③由调度程序引起，调度程序决定：
让谁运行？——调度算法
运行多长时间？——时间片大小

调度时机
- 创建新进程
- 进程退出
- 运行进程阻塞
- I/O中断发生

3.4 调度算法的评价指标

- CPU的利用率

CPU"忙碌"的时间占总时间的比例

$$\text{利用率} = \frac{\text{忙碌的时间}}{\text{总时间}}$$

- 周转时间

周转时间，指从作业被提交给系统开始，到作业完成为止的这段时间间隔.

周转时间 = 作业完成时间 − 作业提交时间

$$\text{平均周转时间} = \frac{\text{各作业周转时间之和}}{\text{作业数}}$$

$$\text{带权周转时间} = \frac{\text{作业周转时间}}{\text{作业实际运行时间}} = \frac{\text{作业完成时间} - \text{作业提交时间}}{\text{作业实际运行时间}}$$

$$\text{平均带权周转时间} = \frac{\text{各作业带权周转时间之和}}{\text{作业数}}$$

- 等待时间

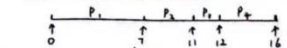
进程处于等待处理机状态时间之和，等待时间越长，用户满意度越低.

3.5 调度算法

- 先来先服务（FCFS）

使用先来先服务调度算法，计算各进程的等待时间、平均等待时间、周转时间、平均周转时间、带权周转时间、平均带权周转时间.

进程	到达时间	运行时间
P1	0	7
P2	2	4
P3	4	1
P4	5	4



周转时间 = 完成时间 − 到达时间
带权周转时间 = 周转时间 / 运行时间

周转时间：P1 = 7 − 0 = 7
P2 = 11 − 2 = 9
P3 = 12 − 4 = 8
P4 = 16 − 5 = 11

带权周转时间：P1 = 7/7 = 1
P2 = 9/4 = 2.25
P3 = 8/1 = 8
P4 = 11/4 = 2.75

等待时间：P1 = 7 − 7 = 0
P2 = 9 − 4 = 5
P3 = 8 − 1 = 7
P4 = 11 − 4 = 7

先来先服务 优缺点：

优点：公平、算法实现简单

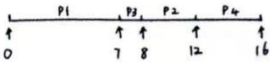
缺点：对长作业有利，对短作业不利

- 短作业优先

eg：使用非抢占式的短作业优先调度算法，计算各进程的等待时间、平均等待时间、周转时间、平均周转时间、带权周转时间、平均带权周转时间.

进程	到达时间	运行时间
P1	0	7
P2	2	4
P3	4	1
P4	5	4

调度顺序：P1→P3→P2→P4



周转时间：P1 = 7 − 0 = 7
P2 = 12 − 2 = 10
P3 = 8 − 4 = 4
P4 = 16 − 5 = 11

带权周转时间：P1 = 7/7 = 1
P2 = 10/4 = 2.5
P3 = 4/1 = 4
P4 = 11/4 = 2.75

等待时间：P1 = 7 − 7 = 0
P2 = 10 − 4 = 6
P3 = 4 − 1 = 3
P4 = 11 − 4 = 7

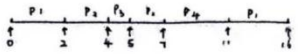
平均周转时间：
(7 + 4 + 1 + 11)/4 = 8

平均带权周转时间：
(1 + 2.5 + 4 + 2.75)/4 = 2.56

平均等待时间：
(0 + 6 + 3 + 7)/4 = 4

时：使用抢占式的短作业优先的调度算法

进程	到达时间	运行时间
P1	0	7
P2	2	4
P3	4	1
P4	5	4



0：P1(7)
2：P1(5) P2(4)
4：P1(5) P2(2) P3(1)
5：P1(5) P2(2) P3(1) P4(4)
7：P1(5) P4(4)
11：P1(5)

周转时间：
P1：16 − 0 = 16
P2：7 − 2 = 5
P3：5 − 4 = 1
P4：11 − 5 = 6

带权周转时间：
P1：16/7 = 2.28
P2：5/4 = 1.25
P3：1/1 = 1
P4：6/4 = 1.5

等待时间：
P1：16 − 7 = 9
P2：5 − 4 = 1
P3：1 − 1 = 0
P4：6 − 4 = 2

平均周转时间：(16 + 5 + 1 + 6)/4 = 7

平均带权周转时间：(2.28 + 1.25 + 1 + 1.5)/4 = 1.5

平均等待时间：(9 + 1 + 0 + 2)/4 = 3

短作业优先 优缺点：

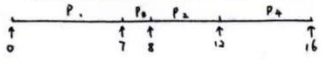
优点："最短的"平均等待时间，周转时间.

缺点：不公平，可能产生饥饿现象.

- 优先级调度算法

非抢占式的优先级调度算法

进程	到达时间	运行时间	优先级
P1	0	7	1
P2	2	4	2
P3	4	1	3
P4	5	4	2



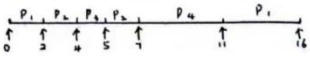
周转：P_1：7-7=0

P_2：8-4=4

P_3：12-2=10

P_4：16-5=11

抢占式的优先级调度算法



- 高响应比优先

非抢占式的调度算法，只有当前正行的进程主动放弃CPU时，才需要进行调度，调度时计算就绪队列中每个就绪进程的响应比，选响应比最高的进程上处理机.

进程	到达时间	运行时间
P1	0	7
P2	2	4
P3	4	1
P4	5	4

响应比 = 等待时间+需要服务时间 / 需要服务时间

0时刻：只有P1到达就绪队列，P1上处理机

7时刻（就绪的P1主动放弃P1）：队列中有

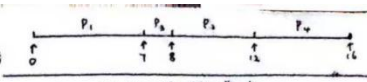
P2(响应比=(3+4)/4=2.25)

P3(响应比=(3+1)/1=4)

P4(响应比=(2+4)/4=1.5)

8时刻(P3完成)：P_2(2.5)、P_4(1.75)

12时刻(P2完成)：队列中只剩下P_4



死锁的概念

- 死锁：各进程互相等待对方手里的资源，导致进程都阻塞，无法向前推进的现象.

饥饿：由于长期得不到想要的资源，某进程无法向前推进的现象.

死循环：某进程执行过程中一直跳不出某个循环的现象.

- 死锁产生的必要条件（四条必须都满足）

1. 互斥条件：只有对必须互斥使用的资源的争抢才会导致死锁.
2. 不可抢夺占条件：进程所获得的资源在未使用完之前，不能由其他进程强行夺走，只能主动释放
3. 请求和保持条件：进程已经保持了至少一个资源，但又提出了新的资源请求.
4. 循环等待条件：存在一种进程资源的循环等待链，链中每一个进程已获得的资源同时被下一个进程所请求.

- 死锁的处理策略

1. 预防死锁，破坏死锁产生的四个必要条件中的一个或几个.
2. 避免死锁，用某种方式防止系统进入不安全状态，从而避免死锁.
3. 检测死锁：允许死锁发生，不过操作系统会负责检测出死锁的发生，并采取某种措施.
4. 解除死锁

死锁的处理策略

- 预防死锁

1. 破坏互斥条件

互斥条件：只有对必须互斥使用的资源的争抢才会导致死锁.

把只能互斥使用的资源改造为允许共享使用，则系统不会进入死锁状态.

2. 破坏不剥夺条件

不剥夺：进程所获得的资源在未使用完之前，不能由其他进程强行夺走，只能主动释放

方案一：当某个进程请求新的资源，得不到满足时，它必须主动释放保持的所有资源.

方案二：考虑各进程的优先级.

3. 破坏请求和保持条件

请求和保持条件：进程已经保持了至少一个资源，但又提出了新的资源请求，而该资源又被其他进程占有，此时请求进程被阻塞，但又对自己已有的资源保持不放

用静态分配已方法

4. 破坏循环等待条件

给资源编号，必须按编号大小顺序申请资源.

死锁的处理策略——避免死锁

安全序列：如果系统按照这种序列分配已资源，则每个进程都能顺利完成，只要能找到一个安全序列，系统就是安全状态.

系统处于安全状态，就一定不会发生死锁.

系统进入不安全状态，就可能发生死锁.

银行家算法

假设系统中有n个进程，m种资源.

每个进程在运行前先声明对各种资源的最大需求数，用n×m的矩阵表示为最大需求矩阵Max.

系统用一个n×m的分配已矩阵，表示对所有进程的资源分配已情况，为一个Allocation矩阵.

Max - Allocation = Need矩阵，表示各进程最多还需要多少各类资源.

长度为m的一维数组Available表示当前系统中还有多少可用资源.

某进程P_i向系统申请资源，可用一个长度为m的一维数组Request，表示本次申请的各种资源量.

用银行家算法预判本次分配已是否会导致进入不安全状态：

① 如果Request$_i$[j] ≤ Need[i,j]，转向到②，否则出错.

② 如果Request$_i$[j] ≤ Available[j]，转向到③，否则表示尚无足够资源，P_i必须等待

③ 系统试探着把资源分配已给P_i，并修改相应数据

available = available - Request

Allocation[i,j] = Allocation[i,j] + Request[i,j]

Need[i,j] = Need[i,j] - Request[i,j]

④ 操作系统执行安全算法，检查此次资源分配已后，系统是否处于安全状态.

若安全，才正式分配已.

否则，恢复相应数据，让进程阻塞等待

死锁的处理策略

- 死锁的检测

为了能对系统是否已发生死锁进行检测，必须：

① 用某种数据结构来保存资源的请求和分配已信息.

② 提供一种算法，利用上述信息来检测系统是否已进入死锁状态.

1）两种结点：

① 进程结点：对应一个进程

② 资源结点：对应一类资源.

一类资源可能有多个.

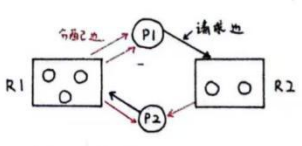
2）两种边：

① 进程结点 → 资源结点

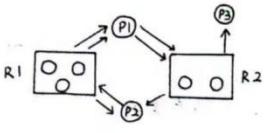
表示进程想申请几个资源

② 资源节点 → 进程结点

表示已经为进程分配了几个资源



能消除所有边，称可完全简化，称一定没有发生死锁.



不能消除所有边，就是发生了死锁.

- 死锁的解除

1. 资源剥夺法
2. 撤销有进程法（终止进程法）
3. 进程回退法.

进程同步、进程互斥

- 同步亦称直接制约关系，它是指为完成某种任务而建立的两个或多个进程，这些进程因为需要在某些位置上协调它们的工作次序而产生的制约关系.

- 把一个时间段内只允许一个进程使用的资源称为临界资源.

对于临界资源的访问，必须互斥地进行.进程互斥指当一个进程访问某临界资源时，另一个想要访问该临界资源的进程必须等待.

- 对于进程互斥.

```
do{
    entry section;  //进入区
    critical section;  // 临界区
    exit section;  // 退出区
    remainder section;  // 剩余区
}while(true)
```

进入区

```
acquire(){
    while(!available)
        ;                    //忙等待
    available = false;  // 获得锁
}
```

退出区

```
release(){
    available = true;  // 释放锁
}
```

信号量机制

- 对原语

wait(S)原语　P操作

signal(S)原语　V操作

- int S=1;　//表示打印机资源数

```
void wait(int S){
    while(S<=0);
    S=S-1;
}
```

```
void signal(int S){
    S=S+1;
}
```

进程P0：

```
...
wait(S);          // 进入区，申请资源
使用打印机资源...  // 临界区，访问资源
signal(S);        // 退出区，释放资源
...
```

- 记录型信号量

```
/* 记录型信号量的定义 */
typedef struct{
    int value;              // 剩余资源数
    struct process *L;  // 等待队列
}semaphore;
```

```
void wait(semaphore S){
    S.value--;
    if(S.value<0){
        block(S.L);  // 可能进入阻塞
    }
}
```

```
void signal(semaphore S){
    S.value++;
    if(S.value<=0){
        wakeup(S.L);  // 唤醒进程
    }
}
```

P0进程：

```
...
wait(S);
使用打印机;
signal(S);
...
```

P1进程：

```
...
wait(S);
使用打印机;
signal(S);
...
```

S.value+1后<=0，表进程在等待该资源

S.value+1后>0，说明已没有进程等待

S.value初值为某类资源的数目

S.value=-2，表2个进程在等待

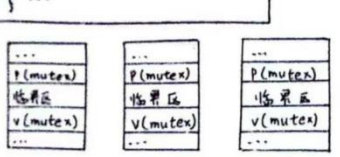
用信号量实现进程互斥、同步

1. 分析并发进程的关键活动，划定临界区.
2. 设置互斥信号量mutex，初值为1
3. 在进入区P(mutex)——申请资源
4. 在退出区V(mutex)——释放资源.

```
typedef struct{
    int value;
    struct process *L;
}semaphore;

semaphore mutex=1;

P1(){
    ...
    P(mutex);
    临界区代码段...
    V(mutex);
}...
```



用信号量实现进程同步

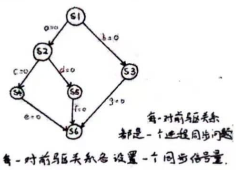
1. 分析什么地方需要实现"同步关系"，即必须保证"一前一后"执行的两个操作.
2. 设置同步信号量S，初始为0
3. 在"前操作"之后执行V(S)
4. 在"后操作"之前执行P(S).

```
semaphore S=0;
P1(){
    代码1;
    代码2;
    V(S);
}
P2(){
    代码3;
    P(S);
    代码4;
}
```

代码3在代码4之前执行

· 信号量机制实现前驱关系

进程P1中有代码S1、P2有S2，…，P6有S6。这些代码要求按如下前驱图所示顺序执行：



每一对前驱关系都是一个进程同步问题

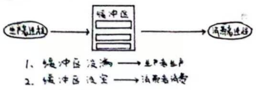
1. 要为每一对前驱关系各设置一个同步信号量。
2. 在"前操作"之后对相应的同步信号量执行 V操作。
3. 在"后操作"之前对相应的同步信号量执行 P操作。

解答：

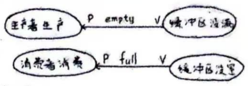
```
P1(){        P2(){        P3(){        P4(){
  ...          ...          ...          ...
  S1;          P(a);        P(b);        P(c);
  V(a);        S2;          S3;          S4;
  V(b);        V(c);        V(g);        V(e);
  ...          V(d);        ...          ...
}              ...        }            }
             }
P5(){        P6(){
  ...          ...
  P(d);        P(e);
  S5;          P(f);
  V(f);        P(g);
  ...          S6;
}              ...
             }
```

生产者消费者问题

· 问题描述



1. 缓冲区没满 ——→ 生产者生产
2. 缓冲区没空 ——→ 消费者消费
3. 互斥关系。



· 实现

生产者、消费者共享一个初始为空、大小为n的缓冲区。

只有缓冲区没满时，生产者才能把产品放入缓冲区，否则必须等待。

只有缓冲区不空时，消费者才能从中取出产品，否则必须等待。

```
semaphore  mutex  = 1;  //互斥
semaphore  empty  = n;  //同步
semaphore  full   = 0;  //同步

producer(){                         生产者
    while(1){
        生产一个产品;
        P(empty);
        P(mutex);               } 互斥
        把产品放入缓冲区;
        V(mutex);
        V(full);
    }
}

consumer(){                         消费者
    while(1){
        P(full);
        P(mutex);
        从缓冲区取出一个产品;
        V(mutex);
        V(empty);
        使用产品;
    }
}
```

实现互斥的P操作一定要在实现同步的P操作之后。

两个V操作顺序可以交换。

在实现生产者、消费者问题实现"一前一后"需要"前V后P"。

· 分段VS分页

1. 分页对用户不可见，分段对用户可见
2. 分页的地址空间是一维的，分段的地址空间是二维的。
3. 分段更容易实现信息的共享和保护
4. 分页、分段访问一个逻辑地址都需要两次访存，分段存储中也可以引入快表机构

虚拟内存的基本概念

· 虚拟内存的定义和特征

寄存器 / 高速缓存 / 内存 / 外存(磁盘、磁带)：↑容量小、速度快、成本高；↓容量大、速度慢、成本低

基于局部性原理，在程序装入后，可以将程序中很快会用到的部分装入内存，暂时用不到的部分留在外存，就可以让程序开始执行。

在程序执行过程后，当所访问的信息不在内存时，由操作系统负责将所需信息从外存调入内存，然后继续执行程序。

在操作系统的管理中，在用户看来似乎有一个比实际内存大得多的内存，这就是虚拟内存。

页面置换算法

· 页面置换算法

最佳置换算法(OPT)

先进先出置换算法(FIFO)

最近最久未使用置换算法(LRU)

时钟置换算法(CLOCK)

改进型的时钟置换算法

追求更少的缺页率

· 最佳置换算法(OPT)

每次选择淘汰的页面将是以后永不使用或者在最长时间内不再被访问的页面。

例：分配了三个内存块，

访问页面	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
内存1	7	7	7	2		2		2			2			2				7		
内存2		0	0	0		0		4			0			0				0		
内存3			1	1		3		3			3			1				1		
是否缺页	√	√	√	√		√		√			√			√				√		

发生9次缺页，页面置换发生6次

缺页率 = 9/20 = 45%

· 先进先出置换算法(FIFO)

每次选择淘汰页面是最早进入内存的页面。

访问页面	3	2	1	0	3	2	4	3	2	1	0	4
内存块1	3	3	3	3			4	4	4	4	0	0
内存块2		2	2	2			2	3	3	3	3	4
内存块3			1	1			1	1	2	2	2	2
内存块4				0			0	0	0	1	1	1
是否缺页	√	√	√	√			√	√	√	√	√	√

缺页次数：10次

分配三个内存时，缺页次数9次

缺页率(4个内存) = 10/12 = 83.3%

· 最近最久使用置换算法(LRU)

每次淘汰的页面是最近最久未使用的页面。

访问页面	1	8	1	7	8	2	7	2	1	8	3	8	2	1	3	1	7	1	3	7
内存1	1	1		1		1					1						1			
内存2		8		8		8					8						7			
内存3				7		7					3						3			
内存4						2					2						2			
缺页否	√	√		√		√					√						√			

算法性能好，实现困难，开销大。

缺页率 = 6/20 = 30%

· 时钟置换算法

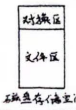
为每个页面设置一个访问位，再将内存中的页面通过链接指针链接成一个循环队列。

当某页被访问时，其访问位置为1。当淘汰一个页面时，只需检查页的访问位。如果是0，换出，如果是1，将其置为0，暂不换出，继续检查下一个页面。(最多两轮扫描)

· 交换技术

交换技术的设计思想：内存空间紧张时，系统将内存中某些进程暂时换出外存，把外存中已具备运行条件的进程换入内存。

暂时换出外存等待的进程状态为挂起状态。



文件区：追求存储空间的利用率，采用离散分配方式。

对换区：追求换入换出速度，采用连续分配方式。

内存空间的分配与回收

连续分配管理方式

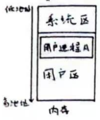
连续分配：指用户进程分配必须是一个连续的内存空间。

· 单一连续分配

内存被分为系统区和用户区。

系统区位于内存的低地址部分，用于存放操作系统相关的数据。

用户区用于存放用户进程相关的数据。

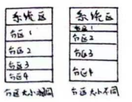


优点：实现简单；无外部碎片。

缺点：只能用于单用户、单任务的操作系统中；有内部碎片，存储器利用率极低。

· 固定分区分配

将整个用户区划分为固定大小的分区，每个分区只装一道作业。



有分区说明表

优点：实现简单，无外部碎片

缺点：1.用户程序太大时，不能满足，不得不采用覆盖技术。2.会产生内部碎片，利用率低

· 动态分区分配

动态分区分配又被分区分配。

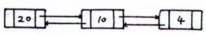
不会预先划分内存分区，根据进程大小动态建立分区。

1. 记录内存的使用情况

① 空闲分区表

分区号	分区大小(MB)	起始地址(M)	状态
1	20	8	空闲
2	10	32	空闲
3	4	60	空闲

② 空闲分区链



动态分配没有内部碎片，但是有外部碎片

内部碎片：分配给某进程的内存区域中，如果有些部分没有用上。

外部碎片：内存中的某些空闲分区由于太小而难以利用。

回收内存分区时，可能遇到四种情况：

相邻分区(空闲)要合并

动态分区分配算法

· 首次适应算法

每次都从低地址开始查找，找到第一个能满足大小的空闲分区。

空闲分区以地址递增的次序排列。每次分配内存时顺序查找空闲分区链，找到大小能满足要求的第一个空闲分区。

· 最佳适应算法

优先使用更小的空闲区。

空闲分区按容量递增次序排列。每次分配内存时顺序查找空闲分区链，找到大小能满足要求的第一个空闲分区

会留下许多很小的难以利用的内存块。

· 最坏适应算法

优先使用更大的空闲区

较大的连续空闲区被迅速用完。有"大进程"到达，就没有内存分存可用。

· 改进型的时钟置换算法

只有被淘汰的页面被修改时，才需要写回外存。

在其他条件都相同时，优先淘汰没有修改过的页面。

修改位=0，页面没有被修改过。

修改位=1，页面被修改过。

(访问位，修改位)

例：(1,1)表示一个页面近期被访问过，且被修改过

算法规则：将所有页面排成一个循环队列

第一轮：从当前位置开始扫描到第一个(0,0)的帧用于替换，本轮不修改

第二轮：若第一轮失败，查找第一个(0,1)的帧用于替换，本轮将所有扫描过的帧访问位设为0。

第三轮：若第二轮失败，查找第一个(0,0)用于替换。

第四轮：若第三轮失败，查找第一个(0,1)用于替换。

改进型CLOCK置换算法选择一个淘汰页面最多会进行四轮扫描。

页面分配、置换策略

· 驻留集

请求分页存储管理中给进程分配的物理块的集合。

固定分配：操作系统为每个进程分配一组固定数目的物理块。

可变分配：在进程运行期间，根据情况做适当的增加或减少

局部置换：进程只能选自己的物理块进行置换。

全局置换：可以自己的进程，也可以为别的进程。

· 抖动(颠簸)现象

刚刚换出的页面马上又要换入内存，刚刚换入的页面马上又要换出外存，这种频繁的页面调度行为称为抖动

I/O 设备

· 特性分类

人机交互类外部设备

存储设备

网络通信设备

· CPU无法直接控制I/O设备的机械部件，因此I/O设备还要有一个电子部件作为CPU和I/O设备机械部件之间的"中介"，用于实现CPU对设备的控制。

这个电子部件为I/O控制器。

I/O控制器功能：
- 接受和识别CPU发出的命令
- 向CPU报告设备的状态
- 数据交换
- 地址识别

· I/O 控制方式

· 邻近适应算法

空闲分区以地址增的顺序排列。

每次分配内存时，从上次查找结束的位置开始查找空闲分区链，找到大小能满足要求的第一个空闲分区。

基本分页存储管理的概念

· 页表

（图：逻辑地址 0页、1页、2页、……、n页；页表 页号/块号 0-3、1-6、2-2、…、n-5；内存 内存块号 0-7）

为了能知道进程的每个页面在内存中存放的位置，操作系统要为每个进程建立一张页表。

页表通常在PCB(进程控制块)中。

· 计算

页号 = 逻辑地址 / 页面长度

页内偏移量 = 逻辑地址 % 页面长度

物理地址 = 页面始址 + 页内偏移量

· 图

（图：页表寄存器 页表始址F、页表长度M；越界中断；逻辑地址 页号P、页内偏移量W；页表；b、W；内存 PCB 系统区、页表、P号页面）

· 地址变换过程

1. 根据逻辑地址算页号、页内偏移量。
2. 页号的合法性检查
3. 若页号合法，再根据页表起始地址、页号找到对应页表项
4. 根据页表项中记录的内存块号、页内偏移量得到的物理地址。
5. 访问物理内存对应的内存单元。

具有快表的地址变换机构

· 基本地址变换机构

两次访存。

· 具有快表的地址变换机构

① 算页号、页内偏移量。

② 检查页号合法性

③ 查快表。若命中，到⑤；未命中，到④。

④ 查页表。找到页面存放的内存块号，并且将页表复制到快表中。

⑤ 根据内存块号与页内偏移量得到物理地址。

⑥ 访问目标内存单元。

快表命中，一次访存。

快表未命中，两次访存。

两级页表

（图：内存块号；0-3、1-…、1023；0-2、1-4、1023；0-762；内存）

将长页表再分页。

N级页表访问一个逻辑地址需要N+1次访存

基本分段存储管理方式

（图：段名A main函数 0号段(7KB)；段名X 子函数 1号段(3KB)；段名D 库函数 2号段(6KB)；段表 段号/段长/基址 0-7K-80K、1-3K-120K、2-6K-40K；内存 40k、80k、120k、(A)单元）

· 哲学家进餐问题

一张圆桌上有5名哲学家，每两个哲学家摆一根筷子，中间是一碗米饭。

1. 关系分析，系统中有5个哲学家进程，5位哲学家与左右邻居对其中间筷子的访问是互斥关系。
2. 每个哲学家进程需要同时持有两个临界资源才能开始吃饭，需避免死锁。
3. 信号量设置，定义互斥信号数组 chopstick[5]={1,1,1,1,1} 用于实现对5个筷子的互斥访问，并对哲学家按0~4编号，哲学家i左边的筷子编号为i，右边筷子编号为(i+1)%5。

```
semaphore  chopstick[5]={1,1,1,1,1}
semaphore  mutex = 1;   //互斥取筷子
Pi(){                           //i号哲学家的进程
    while(1){
        P(mutex);
        P(chopstick[i]);          //拿左
        P(chopstick[(i+1)%5]);    //拿右
        V(mutex);
        吃饭...
        V(chopstick[i]);          //放左
        V(chopstick[(i+1)%5]);    //放右
        思考...
    }
}
```

5.1 内存的基础知识

· 内存

内存可存放数据，程序执行前需要先放到内存中才能被CPU处理——缓和CPU与硬盘之间的速度矛盾。

· 数量单位

$1K = 2^{10}$ (千)

$1M = 2^{20}$ (兆、百万)

$1G = 2^{30}$ (十亿、千兆)

5.2 程序的装入与链接

· 从写程序到程序运行

1. 编译：由源代码文件生成目标模块
2. 链接：由目标模块生成装入模块，链接后形成完整的逻辑地址。
3. 装入：将装入模块装入内存，装入后形成物理地址。

· 三种装入方式

1. 绝对装入：编译时产生绝对地址。
2. 可重定位装入：装入时将逻辑地址转换为物理地址。
3. 动态运行时装入：运行时将逻辑地址转换为物理地址，需设置重定位寄存器。

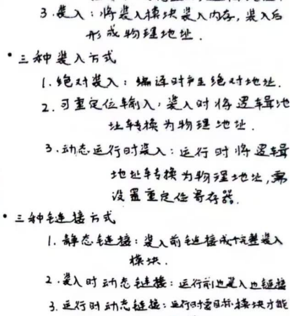
· 三种链接方式

1. 静态链接：装入前链接成完整装入模块。
2. 装入时动态链接：运行前边装入边链接
3. 运行时动态链接：运行时需目标模块才能装入并链接。

5.3 覆盖与交换

· 覆盖技术

解决"程序大小超过物理内存总和"问题



内存分为一个"固定区"和若干个"覆盖区"。

"固定区"中调入后不再调出。

"覆盖区"需调入内存，用不到时调出。

必须由程序员声明覆盖结构，操作系统完成自动覆盖。对用户不透明