✨操作系统

操作系统（Operation System，OS）是管理计算机硬件和软件的计算机程序。

计算机系统的构成：用户、应用程序、操作系统、硬件（裸机）。

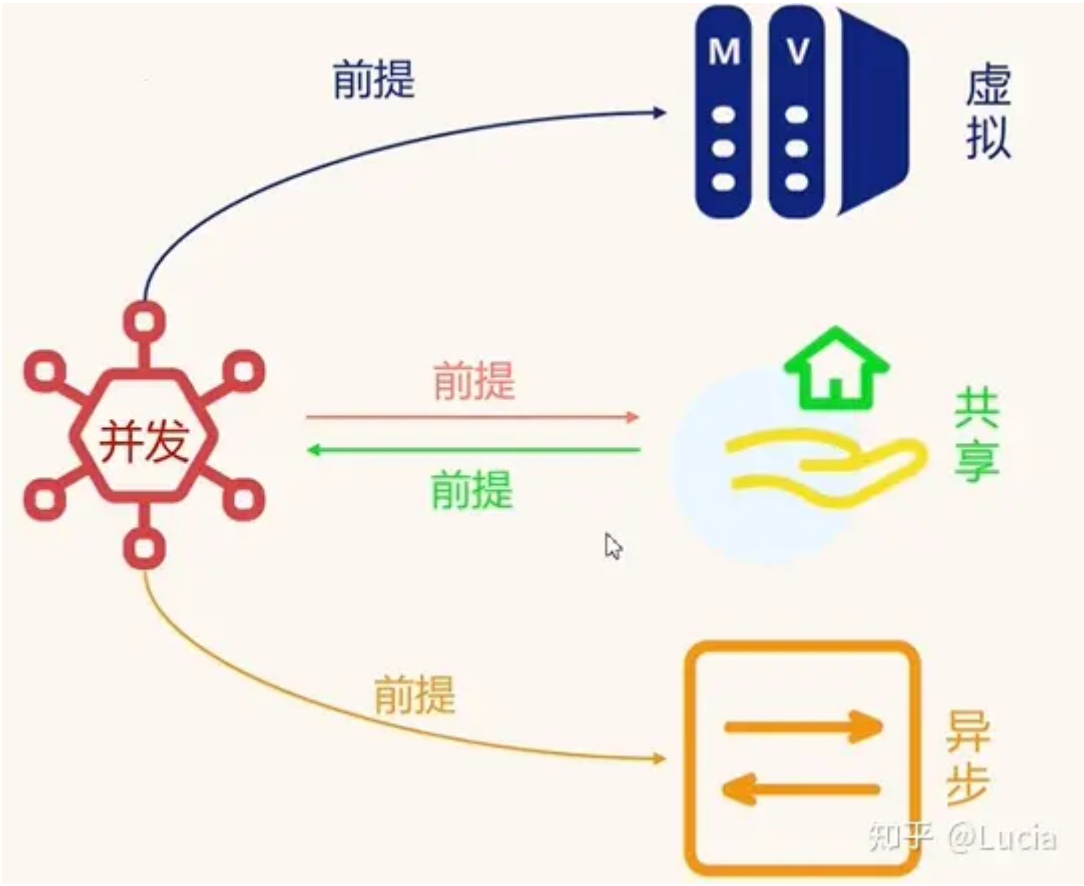
OS是一种系统软件：

- 与硬件交互
- 对资源共享（硬件）进行调度管理
- 解决并发操作处理中存在的协调问题
- 数据结构复杂，外部接口多样化，便于用户反复使用

操作系统的功能

- 作为计算机系统资源的管理者
  处理器管理：进程控制、进程同步、进程通信、调度
  存储器管理：内存分配、内存保护、地址映射、内存扩充
  I/O设备管理：缓冲管理、设备分配、设备处理
  文件管理：文件存储空间的管理、目录管理、文件的读/写管理和保护
- 作为用户与计算机硬件系统之间的接口
  程序接口（系统调用）、命令接口、GUI（图形用户接口）
- 实现了对计算机资源的抽象
  将具体的计算机硬件资源抽象成软件资源，方便用户使用（开放了简单的访问方式，隐藏了实现细节）

操作系统的特征



并发是共享、虚拟、异步的前提，其中并发和共享互为前提

- **并发**：同一时间间隔内执行和调度多个程序的能力
  宏观上，处理机同时执行多道程序
  微观上，处理机在多道程序间高速切换（分时交替执行）
  关注单个处理机同一时间段内处理任务数量的能力
  并行：同一时刻发生的事件数量，关注有多少个CPU可以同时执行任务的能力
- **共享**：即资源共享，系统中的资源供多个并发执行的应用程序共同使用
  同时访问方式：同一时段允许多个程序同时访问共享资源
  互斥共享方式：也叫独占式，允许多个程序在同一个共享资源上独立而互不干扰的工作
- **虚拟**：使用某种技术把一个物理实体变成多个逻辑上的对应物
  时分复用TDM、空分复用SDM
- **异步**：多道程序环境下，允许多个程序并发执行，单处理环境下，多个程序分时交替执行。
  程序执行的不可预知性
  - 获得运行的时机
  - 因何暂停
  - 每道程序需要多少时间
  - 不同程序的性能，比如计算多少，I/O多少

✨线程与进程的比较

- 进程是**资源调度的基本单位**，运行一个可执行程序会创建一个或多个进程，进程就是运行起来的可执行程序
- 线程是**程序执行的基本单位**，是轻量级的进程。**每个进程中都有唯一的主线程，且只能有一个**，主线程和进程是相互依存的关系，主线程结束进程也会结束。**协程是用户态的轻量级线程，线程内部调度的基本单位**

✨并发和并行的区别

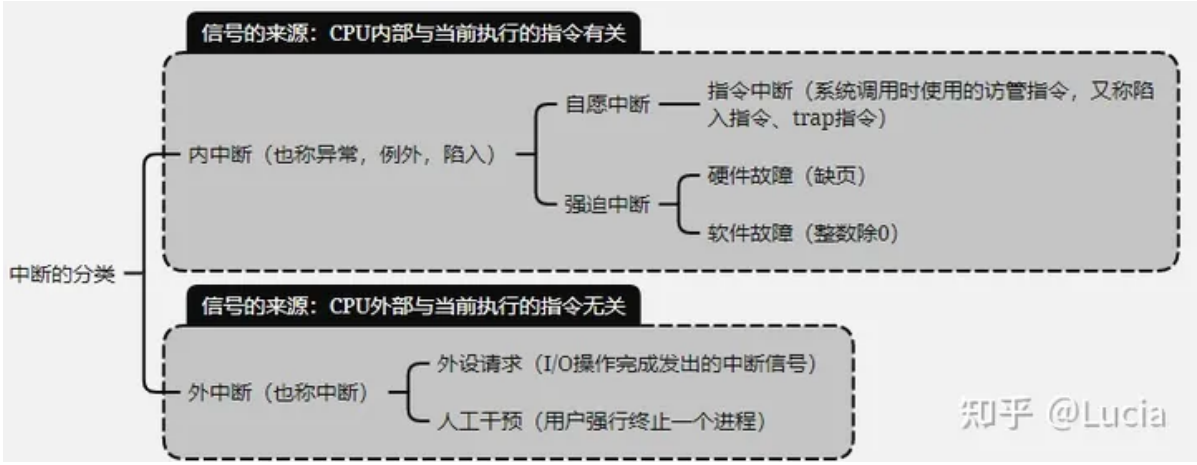
- **并发（concurrency）**：指宏观上看起来两个程序在同时运行，比如说在单核cpu上的多任务。但是从微观上看两个程序的指令是交织着运行的，指令之间交错执行，在单个周期内只运行了一个指令。这种并发并不能提高计算机的性能，只能提高效率（如降低某个进程的响应时间）。
- **并行（parallelism）**：指严格物理意义上的同时运行，比如多核cpu，两个程序分别运行在两个核上，两者之间互不影响，单个周期内每个程序都运行了自己的指令，也就是运行了两条指令。这样说来并行的确提高了计算机的效率。所以现在的cpu都是往多核方面发展。

✨内核态和用户态

为了避免操作系统和关键数据被用户程序破坏，保证系统的安全运行，将处理器的执行状态分为内核态和用户态。

- **内核态**是操作系统管理程序执行时所处的状态，能够执行包含特权指令在内的一切指令，能够访问系统内所有的存储空间。（特权指令和非特权指令都能执行，但不能执行陷入指令）
- **用户态**是用户程序执行时处理器所处的状态，不能执行特权指令，只能访问用户地址空间。（只能执行非特权指令）
- 用户程序运行在用户态，操作系统内核程序运行在内核态。
- 处理器从用户态切换到内核态的方法：**中断**

- 通过程序状态字（PSW）区分目前运行的是用户应用程序还是内核程序。（1用户程序，0内核程序）



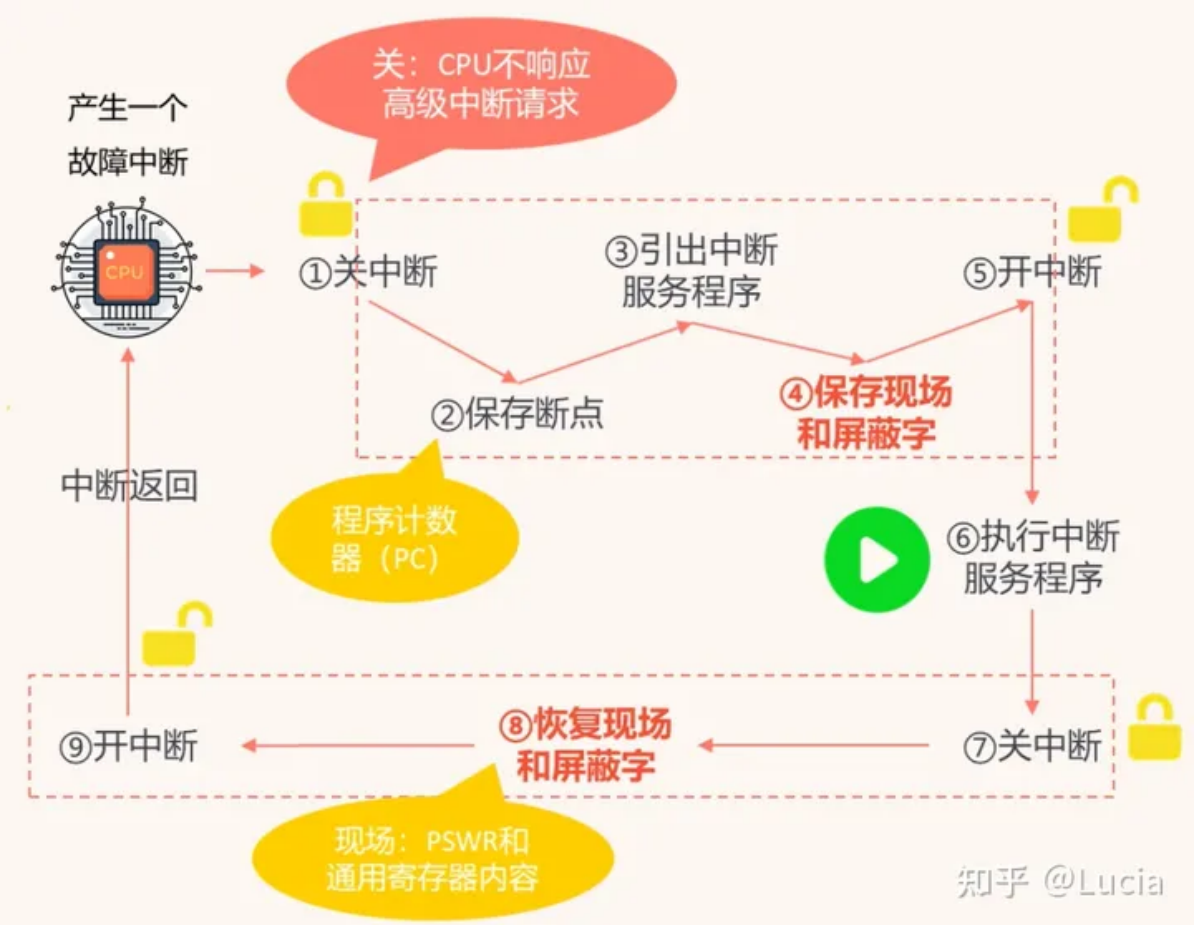
外中断和异常有什么区别?

外中断是指由 CPU 执行指令以外的事件引起，如 I/O 完成中断，表示设备输入/输出处理已经完成，处理器能够发送下一个输入/输出请求。此外还有时钟中断、控制台中断等。

而**异常**是由 CPU 执行指令的内部事件引起，如非法操作码、地址越界、算术溢出等。

中断处理过程

1. 关中断：CPU对当前中断作出的基本响应；
2. 保存断点：保证中断处理完毕后能够返回继续执行程序，保存的是程序计数器（PC）；
3. 引出中断处理程序：读取中断处理程序地址，并没有执行；
4. 保存现场和屏蔽字：保存寄存器中的数据；
5. 开中断：关中断后CPU不响应高级中断请求，开中断后CPU可以开始响应高级中断请求；
6. 执行中断服务程序：可以并发响应其他中断；
7. 关中断
8. 恢复现场和屏蔽字
9. 开中断
10. 中断返回



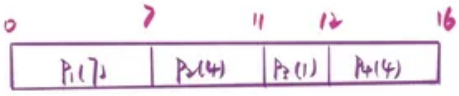
✨调度算法评价指标

1. **CPU利用率**：CPU忙碌时间 / 总时间
2. **系统吞吐量**：总共完成作业的数量 / 总花费时间
3. **周转时间**：作业完成时间 – 作业提交时间
4. **平均周转时间**：（作业1的周转时间 + ... + 作业n的周转时间）/ n
5. **带权周转时间**：作业周转时间 / 作业实际运行时间
6. **平均带权周转时间**：（作业1的带权周转时间 + ... + 作业n的带权周转时间）/ n
7. **等待时间**：进程处于等待处理机状态的时间之和
8. **响应时间**：从用户提交请求到系统首次产生响应所用的时间

相关计算：

例题：各进程到达就绪队列的时间，需要的运行时间如下表所示。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
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4



周转时间 = 完成时间 - 到达时间

P_1：7-0=7　P_2：11-2=9　P_3：12-4=8　P_4：16-5=11

平均周转时间：(7+9+8+11)/4 = 8.75

等待时间 = 周转时间 - 运行时间

P_1：7-7=0　P_2：9-4=5　P_3：8-1=7　P_4：11-4=7

平均等待时间：(0+5+7+7)/4 = 4.75

带权周转时间 = 周转时间 / 运行时间

P_1：7/7=1　P_2：9/4=2.25　P_3：8/1=8　P_4：11/4=2.75

平均带权周转时间：(1+2.25+8+2.75)/4 = 3.5

✨进程调度算法

1. 先来先服务（FCFS，First Come First Served）

- 算法内容：调度作业/就绪队列中最先入队者，等待操作完成或阻塞
- 算法原则：按作业/进程到达顺序服务（执行）
- 调度方式：非抢占式调度
- 适用场景：作业/进程调度
- 优缺点：
  - 有利于CPU繁忙型作业，充分利用CPU资源
  - 不利于I/O繁忙型作业，操作耗时，其他饥饿

2. 短作业优先（SJF，Shortest Job First）

- 算法内容：所需服务时间最短的作业/进程优先服务（执行）
- 算法原则：追求最少的平均（带权）周转时间
- 调度方式：SJF/SPF非抢占式调度，SRTN（最短剩余时间优先）抢占式

- 适用场景：作业/进程调度
- 优缺点：
  - 平均等待/周转时间最少
  - 长作业周转时间会增加或饥饿
  - 估计时间不准确，不能保证紧迫任务及时处理

3. 高响应比优先调度（HRRN，Highest Response Ratio Next）

- 算法内容：结合FCFS和SJF，综合考虑等待时间和服务时间计算响应比，高的优先调度
- 算法原则：综合考虑作业/进程的等待时间和服务时间
- 调度方式：非抢占式
- 适用场景：作业/进程调度
- 响应比计算：
  - 响应比=（等待时间-服务时间）/服务时间，≥1
  - 只有当前进程放弃执行权（完成/阻塞）时，重新计算所有进程响应比
  - 长作业等待越久响应比越高，更容易获得处理机

4. 优先级调度（PSA，Priority-Scheduling Algorithm）

- 算法内容：按作业/进程的优先级（紧迫程度）进行调度
- 算法原则：优先级最高（最紧迫）的作业/进程先调度
- 调度方式：抢占/非抢占式（并不能获得及时执行）
- 适用场景：作业/进程调度
- 优先级设置原则：
  - 静态/动态优先级
  - 系统>用户，交互型>非交互型，I/O型>计算型
  - 低优先级进程可能会产生饥饿

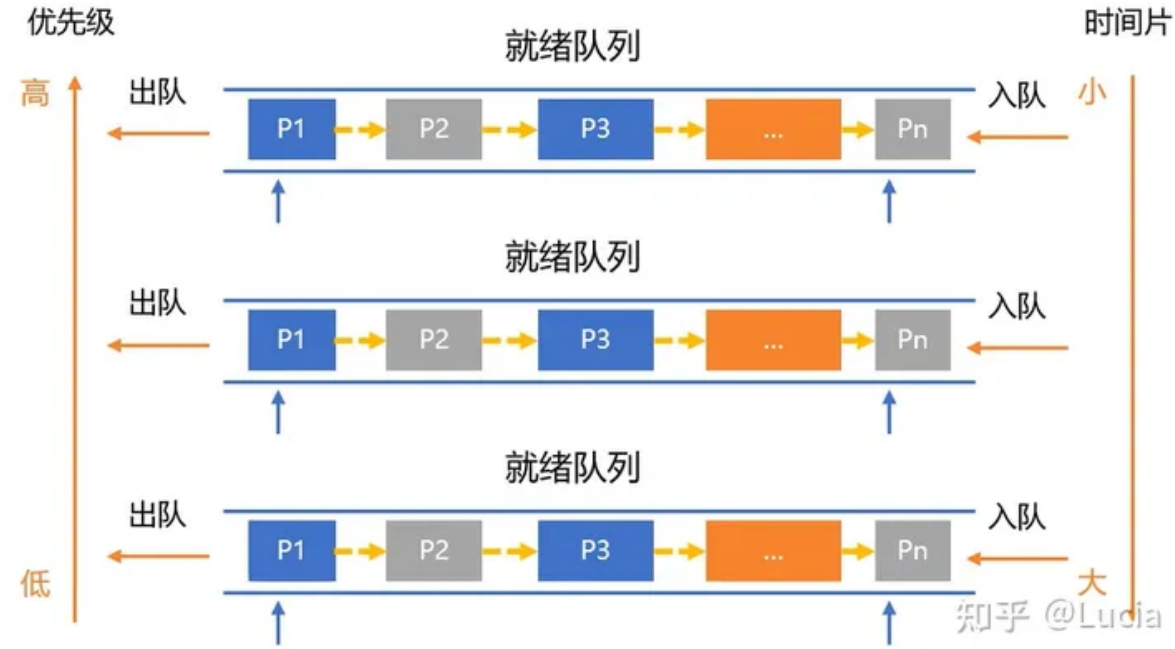
5. 时间片轮转调度（RR，Round-Robin）

- 算法内容：按进程到达就绪队列的顺序，轮流分配一个时间片去执行，时间用完则剥夺
- 算法原则：公平、轮流为每个进程服务，进程在一定时间内都能得到响应
- 调度方式：抢占式，由时钟中断确定时间到
- 适用场景：进程调度
- 优缺点：
  - 公平、响应快，适用于分时系统
  - 时间片决定因素：系统响应时间，就绪队列进程数量，系统处理能力
  - 时间片太大，相当于FCFS，太小，处理机切换频繁，开销增大

6. 多级反馈队列调度（MFQ，Multileveled Feedback Queue）

- 算法内容：设置多个按优先级排序的就绪队列，优先级从高到低，时间片从小到大，新进程采用队列降级法：进入第一级队列，按FCFS分时间片，没有执行完，移到第二级、第三级...，前面队列不为空，不执行后续队列进程
- 算法原则：集前几种算法优点，相当于PSA+RR
- 调度方式：抢占式

- 适用场景：进程调度
- 优缺点：
  - 对各类型相对公平，快速响应
  - 终端型作业用户：短作业优先
  - 批处理作业用户：周转时间短
  - 长批处理作业用户：在前几个队列部分执行

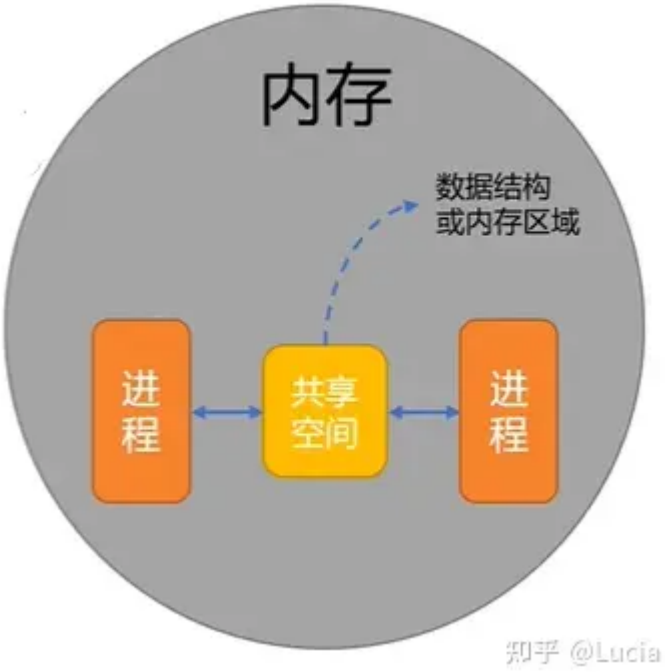


✨进程通信

- 进程通信即进程间的信息交换
- 进程是资源分配的基本单位，各进程内存空间彼此独立
- 一个进程不能随意访问其他进程的地址空间

共享存储（Shared-Memory）

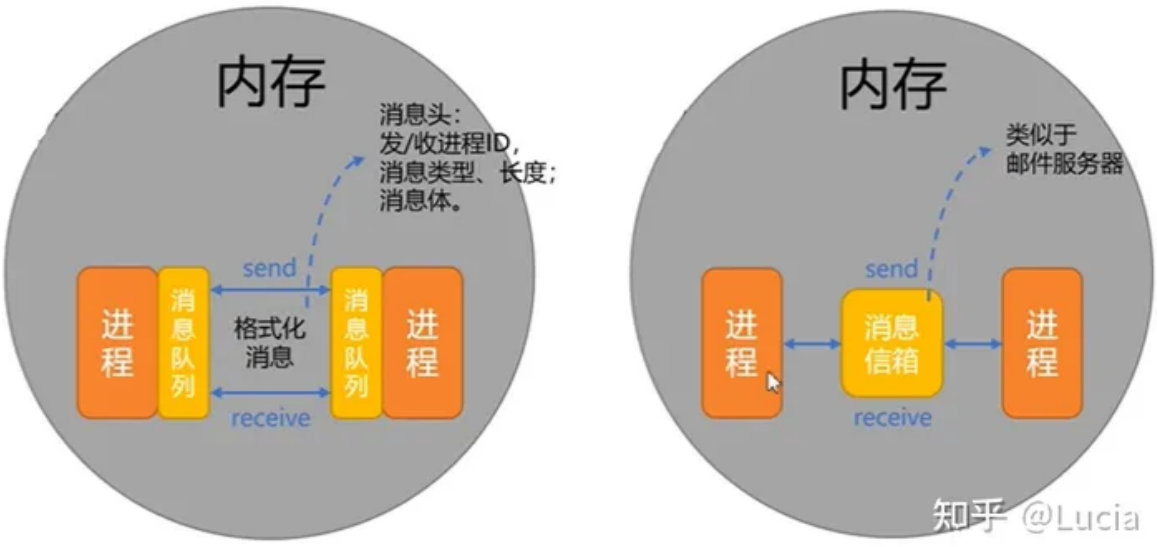
- 基于**共享数据结构**的通信方式
  - 多个进程共用某个数据结构（OS提供并控制）
  - 由用户（程序员）负责同步处理
  - 低级通信：可以传递少量数据，效率低
- 基于**共享存储区**的通信方式
  - 多个进程共用内存中的一块存储区域
  - 由进程控制数据的形式和方式
  - 高级通信：可以传递大量数据，效率高
- 数据收发双方不可见，存在安全隐患



消息传递（Message-Passing）

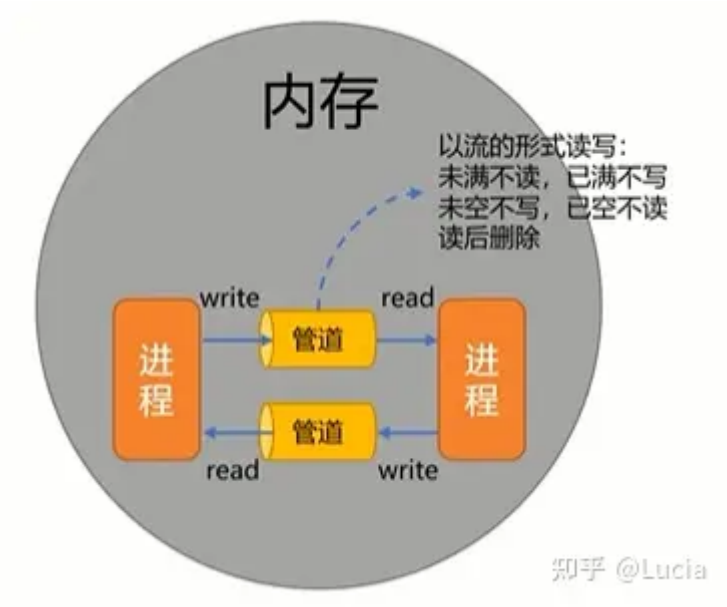
- 直接通信：点到点发送
  - 发送和接收时指明双方进程的ID
  - 每个进程维护一个消息缓冲队列
- 间接通信：广播信箱
  - 以信箱为媒介，作为中间实体
  - 发进程将消息发送到信箱，收进程从信箱读取
  - 可以广播，容易建立双向通信链

和共享内存的区别：消息传递基于send和receive原语进行操作，共享存储直接操作数据结构或某一块内存。



管道通信（Pipe）

- 管道
  - 用于连续读/写进程的共享文件，pipe文件
  - 本质是内存中固定大小的缓冲区
- 半双工通信
  - 同一时段只能单向通信，双工通信需要两个管道
  - 以先进先出（FIFO）方式组织数据传输
  - 通过系统调用read()/write()函数进行读写操作



信号量（Semaphore）机制

进程同步

- 协调进程间的相互制约关系，使它们按照预期的方式执行的过程
- 前提：
- 进程是并发执行的，进程间存在着相互制约的关系
- 并发的进程对系统共享资源进行竞争
- 进程通信，过程中相互发送的信号称为消息或事件
- 两种相互制约形式
- 间接相互制约关系（互斥）：进程排他性地访问共享资源
- 直接相互制约关系（同步）：进程间的合作，比如管道通信

互斥地访问临界资源

访问过程：

1. 进入区：尝试进入临界区，成功则加锁（lock）
2. 临界区：访问共享资源
3. 退出区：解锁（unlock），唤醒其他阻塞进程
4. 剩余区：其他代码

访问原则：

- 空闲让进：临界区空闲，允许一个进程进入
- 忙则等待：临界区已有进程，其他进程等待（阻塞状态）
- 有限等待：处于等待的进程，等待时间有限
- 让权等待：等待时应让出CPU执行权，防止忙等待

PV操作：

- P操作：wait原语，进程等待
- V操作：signal原语，唤醒等待进程

整型信号量：违背让权等待，会发生忙等

```
// 整型信号量，表示可用资源数
int S = 1;
// wait原语，相当于进入区
void wait(int S) {
    // 资源不够，循环等待
    while (S <= 0) {
        S = S - 1;
    }
}
// signal原语，相当于退出区
void signal(int S) {
    S = S + 1;
}
```

记录型信号量：进程进入阻塞状态，不会忙等

```
// 记录型信号量定义
typeof struct {
    int value;  // 剩余资源数量
    struct process *L;  // 进程等待队列
} semaphore;

void wait(semaphore S) {  // 申请资源
    S.value--;
    if(S.value < 0) {
        // block原语阻塞进程
        block(S.L);
    }
}

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

管程（Monitor）

- 管程是用于实现进程同步的工具，是由代表共享资源的数据结构和一组过程（进行PV操作的函数）组成的管理程序（封装）。

- 管程的组成：
  - 管程名称
  - 局部于管程内部的共享数据结构
  - 对改数据结构操作的一组过程（函数）
  - 管程内共享数据的初始化语句
- 管程的基本特性：
  - 是一个模块化的基本程序单位，可以单独编译
  - 是一种抽象的数据类型，包含数据和操作
  - 信息掩蔽，共享数据只能被管程内的过程访问
- 条件变量/条件对象
  - 进入管程的进程可能由于条件不满足而阻塞
  - 此时进程应释放管程以便其他进程调用管程
  - 进程被阻塞的条件有多个，移入不同的条件队列
  - 进程被移入条件队列后，应释放管程

✨信号量（生产者消费者、读者写者、哲学家进餐问题）

- 用户进程可以通过使用操作系统提供的一对原语来对信号量进行操作，从而方便的实现进程互斥、进程同步。
- 信号量其实就是一个变量（可以是一个整数，也可以是更复杂的记录型变量）可以使用一个信号量表示系统中某种资源的数量。
- 原语是一种特殊的程序段，其执行只能一气呵成，不可被中断。原语是由关中断/开中断指令实现的。
- 一对原语：`wait(S)`原语和`signal(S)`原语，可以把原语理解为我们自己写的函数，函数名分别为wait和signal，括号里的信号量S其实就是函数调用时传入的一个参数。wait、signal原语常简称为P、V操作。

实现进程互斥	实现进程同步
设置互斥信号量，初始值为1 临界区之前对信号量执行P操作 临界区之后对信号量执行V操作	设置同步信号量，初始值为0 在前操作之后执行V操作 在后操作之前执行P操作

- 互斥访问临界资源原则：空闲让进、忙则等待、有限等待、让权等待
- 管程：用于实现进程同步的工具，是由代表共享资源的数据结构和一组过程（进行P、V操作的函数）组成的管理程序（封装）

生产者消费者问题

系统中有一组生产者进程和一组消费者进程，生产者进程每次生产一个产品放入缓冲区，消费者进程每次从缓冲区中取出一个产品并使用（产品可理解为某种数据）。

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（同步关系）；只有缓冲区不为空时，消费者才能从中取出产品，否则必须等待（同步关系）；缓冲区是临界资源，各进程必须互斥的访问（互斥关系）。

生产者每次要消耗(P)一个空闲缓冲区，并生产(V)一个产品。消费者每次要消耗(P)一个产品，并释放一个空闲缓冲区(V)。往缓冲区放入/取走产品需要互斥。

```
semaphore mutex = 1;  // 互斥信号量，实现对缓冲区的互斥访问
semaphore empty = n;  // 同步信号量，表示空闲缓冲区的数量
semaphore full = 0;   // 同步信号量，表示产品的数量，也即非空缓冲区的数量

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

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

多生产者多消费者问题

桌子上有一只盘子，每次只能向其中放入一个水果，爸爸专门向盘子中放苹果，妈妈专门向盘子中放橘子，儿子专等着吃盘子中的橘子，女儿专等着吃盘子中的苹果。只有盘子空时，爸爸或妈妈才可向盘子中放入一个水果，仅当盘子中有自己需要的水果时，儿子或女儿可以从盘子中取出水果。

对缓冲区（盘子）的访问要互斥的进行（互斥关系），父亲将苹果放入盘子后，女儿才能取苹果（同步关系），母亲将橘子放入盘子后，儿子才能取橘子（同步关系），只有盘子为空时，父亲或母亲才能放入水果（同步关系）。

```
semaphore mutex = 1;   // 实现互斥访问盘子（缓冲区）
semaphore apple = 0;   // 盘子中有几个苹果
semaphore orange = 0;  // 盘子中有几个橘子
semaphore plate = 1;   // 盘子中还可以放多少个水果

dad () {
  while (1) {
    准备一个苹果;
    P(plate);
    P(mutex);
    把苹果放入盘子;
    V(mutex);
    V(apple);
  }
}

mom () {
  while (1) {
```

```
    准备一个橘子;
    P(plate);
    P(mutex);
    把橘子放入盘子;
    V(mutex);
    V(orange);
  }
}

daughter () {
  while (1) {
    P(apple);
    P(mutex);
    从盘子中取出苹果;
    V(mutex);
    V(plate);
    吃掉苹果;
  }
}

son () {
  while (1) {
    P(orange);
    P(mutex);
    从盘子中取出橘子;
    V(mutex);
    V(plate);
    吃掉橘子;
  }
}
```

读者写者问题

有读者和写者两组并发进程，共享一个文件，当两个或两个以上的读进程同时访问共享数据时不会产生副作用，但若某个写进程和其他进程（读进程或写进程）同时访问共享数据时，则可能会导致数据不一致的错误。允许多个读者可以同时对文件执行读操作，只允许一个写者往文件中写信息，任一写者在完成写操作之前不允许其他读者或写者工作，写者执行写操作前，应让已有的读者和写者全部退出。

互斥关系：写进程和写进程、写进程与读进程。

```
semaphore rw = 1;  // 读写互斥
int count = 0;  // 当前读进程数量
semaphore mutex = 1;  // 互斥访问缓冲区
semaphore w = 1;  // 写写互斥

writer () {
  while (1) {
    P(w);
    P(rw);
    写文件;
    V(rw);
    V(w);
  }
}

```

```
reader() {
  while (1) {
    P(w);
    P(mutex);
    if (count == 0)
      P(rw);
    count++;
    V(mutex);
    V(w);
    读文件;
    P(mutex);
    count--;
    if (count == 0)
      V(rw);
    V(mutex);
  }
}
```

哲学家进餐问题

一张圆桌上坐着5名哲学家，每两个哲学家之间的桌上摆一根筷子，桌子的中间是一碗米饭，哲学家们倾注毕生的精力用于思考和进餐，哲学家在思考时，并不影响他人。只有当哲学家饥饿时，才试图拿起左右两根筷子（一根一根拿起）。如果筷子已在他人手上，则需等待。饥饿的哲学家只有同时拿起两根筷子才可以开始进餐，当进餐完毕，放下筷子继续思考。

五位哲学家与左右邻居对其中间筷子的访问是互斥关系。

```
semaphore chopstick[5] = { 1, 1, 1, 1, 1};  // 对5根筷子的互斥访问
semaphore mutex = 1;  // 互斥的取筷子

Pi () {
  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;
  }
}
```

✨死锁

- 死锁：多个进程由于竞争资源而造成的阻塞现象，若无外力作用，这些进程将无继续推进。
- 相似概念：饥饿：等待时间过长以至于给进程推进和响应带来明显影响，饿而不死

死锁产生的原因

- 系统资源的竞争
- 进程推进顺序非法

死锁产生的必要条件

- 互斥条件：共享资源的排他性访问
- 不剥夺条件：访问时该共享资源不会被剥夺
- 请求并保持条件：保持当前资源时请求另一个资源
- 循环等待条件：存在共享资源的循环等待链

死锁的预防

- 破坏互斥条件
    - 将只能互斥访问的资源改为同时共享访问
    - 将独占锁改为共享锁
    - 不是所有资源都能改成可共享的
- 破坏不剥夺条件
    - 请求新资源无法满足时，必须释放已有资源
    - 由OS协助强制剥夺某进程持有的资源
    - 实现复杂，代价高
    - 此操作过多导致原进程任务无法推进
- 破坏请求和保持条件
    - 进程开始运行时一次性申请所需资源（资源浪费、进程饥饿）
    - 阶段性请求和释放资源
- 破坏循环等待条件
    - 对所有资源进行排序，按序号请求资源（请求时先低再高，释放时先高再低）
    - 对资源的编号应相对稳定，限制了新设备增加
    - 进程使用资源的顺序可能与系统编号顺序不同
    - 限制了用户编程

死锁的避免

- 安全性算法
    - 系统安全状态
        - 安全状态一定不会出现死锁
        - 不安全状态可能出现死锁
    - 银行家算法
        - 系统预判进程请求是否导致不安全状态
        - 是则拒绝请求，否则答应请求

（2）银行家算法描述

设 $Request_i$ 是进程 P_i 的请求向量，$Request_i[j] = K$ 表示进程 P_i 需要 j 类资源 K 个。当 P_i 发出资源请求后，系统按下述步骤进行检查：

① 若 $Request_i[j] \le Need[i,j]$，则转向步骤②；否则认为出错，因为它所需要的资源数已超过它所宣布的最大值。

② 若 $Request_i[j] \le Available[j]$，则转向步骤③；否则，表示尚无足够资源，$P_i$ 须等待。

③ 系统试探着把资源分配给进程 P_i，并修改下面数据结构中的数值：

$$Available = Available - Request_i;$$
$$Allocation[i,j] = Allocation[i,j] + Request_i[j];$$
$$Need[i,j] = Need[i,j] - Request_i[j];$$

④ 系统执行安全性算法，检查此次资源分配后，系统是否处于安全状态。若安全，才正式将资源分配给进程 P_i，以完成本次分配；否则，将本次的试探分配作废，恢复原来的资源分配状态，让进程 P_i 等待。

（3）安全性算法

设置工作向量 Work，有 m 个元素，表示系统中的剩余可用资源数目。在执行安全性算法开始时，Work = Available。

① 初始时安全序列为空。

② 从 Need 矩阵中找出符合下面条件的行：该行对应的进程不在安全序列中，而且该行小于等于 Work 向量，找到后，把对应的进程加入安全序列；若找不到，则执行步骤④。

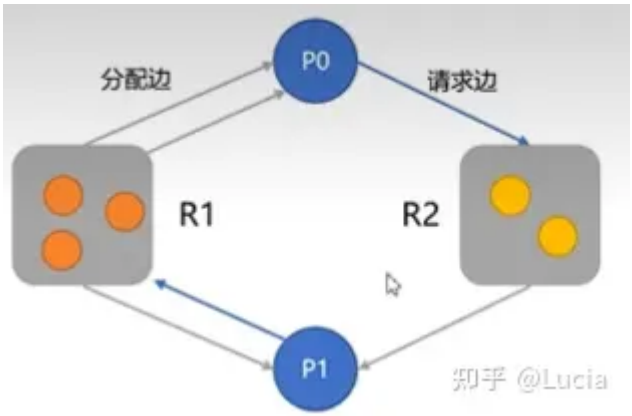
③ 进程 P_i 进入安全序列后，可顺利执行，直至完成，并释放分配给它的资源，因此应执行 Work = Work + Allocation[i]，其中 Allocation[i]表示进程 P_i 代表的在 Allocation 矩阵中对应的行，返回步骤②。

④ 若此时安全序列中已有所有进程，则系统处于安全状态，否则系统处于不安全状态。

看完上面对银行家算法的过程描述后，可能会有眼花缭乱的感觉，现在通过举例来加深理解。

死锁的检测

- 资源分配图
  - 两种资源
  - 两种节点
- 死锁定理（死锁状态的充分条件）
  - 当且仅当此状态下资源分配图是不可完全简化的
  - 简化过程类似于拓扑排序算法



依次消除与不阻塞进程相连的边，直到无边可消，所谓不阻塞进程是指其申请的资源数还足够的进程，若资源分配图是不可完全简化的，说明发生了死锁。

死锁的解除

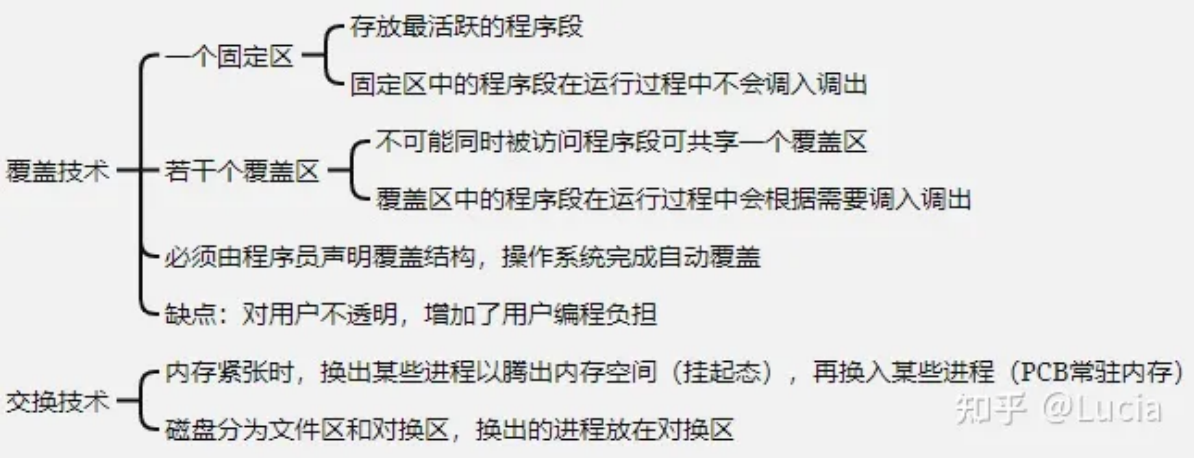
- 资源剥夺
  - 挂起死锁进程
  - 剥夺其资源
  - 将资源分配给其他（死锁）进程
- 撤销进程
- 进程回退
  - 回退到足以避免死锁的地步
  - 需要记录进程历史信息，设置还原点

✨内存交换和覆盖有什么区别

交换技术主要是在不同进程（或作业）之间进行，而覆盖则用于同一程序或进程中。

交换：内存空间紧张时，系统将内存中某些进程暂时换出外存，把外存中某些已具备运行条件的进程换入内存(进程在内存与磁盘间动态调度)。

覆盖：由于程序运行时并非任何时候都要访问程序及数据的各个部分（尤其是大程序），因此可以把用户空间分成为一个固定区和若干个覆盖区。将经常活跃的部分放在固定区，其余部分按照调用关系分段，首先将那些即将要访问的段放入覆盖区，其他段放在外存中，在需要调用前，系统将其调入覆盖区，替换覆盖区中原有的段。



✨内存保护

内存中的进程既不能修改操作系统的数据，也不能修改其他进程的数据，即各个进程只能访问自己的内存空间。

内存保护的方式：

1. 上下限寄存器：存放进程的上下限地址，进程的指令要访问某个地址时，CPU检查是否越界。
2. 界地址寄存器（限长寄存器）+重定位寄存器（基址寄存器）：重定位寄存器中存放进程的起始物理地址，界地址寄存器中存放进程的最大逻辑地址。

✨动态分区分配算法

连续内存分配管理方式

- **单一连续分配：**内存被分为两个部分：系统区和用户区，系统区通常位于内存的低地址部分，用于存放操作系统的相关数据。用户区存放用户进程相关数据，用户区内只能有一道用户程序，用户程序独占用户区。
  - 优点：实现简单，无外部碎片，不一定需要内存保护。
  - 缺点：只能用于单用户，单任务OS，有内部碎片，存储器利用率低。
- **固定分区分配：**将整个用户空间划分为若干个固定大小的分区。（这里的固定大小，指划分完后分区的大小不再改变。而不是内存大小都一样的分区）。每个分区中只能装入一道作业。
  - 优点：实现简单，无外部碎片。
  - 缺点：较大用户程序时，需要采用覆盖技术，降低了性能；会产生内部碎片，利用率低。
- **动态分区分配：**不会预先划分内存分区，而是在进程装入内存时根据进程的大小动态建立分区，并使分区的大小正好适合进程的需要。

动态分区分配方法

1. **首次适应算法**
   算法思想：每次都从低地址开始查找，找到第-个能满足大小的空闲分区。
   如何实现：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(或空闲分区表)，找到大小能满足要求的第一个空闲分区。
2. **最佳适应算法**
   算法思想：由于动态分区分配是一种连续分配方式，为各进程分配的空间必须是连续的一整片区域。因此为了保证当“大进程”到来时能有连续的大片空间，可以尽可能多地留下大片的空闲区，即优先使用更小的空闲区。
   如何实现：空闲分区按容量递增次序链接。每次分配内存时顺序查找空闲分区链(或空闲分区表)，找到大小能满足要求的第一个空闲分区。
3. **最坏适应算法**
   又称最大适应算法(Largest Fit)
   算法思想：为了解决最佳适应算法的问题——即留下太多难以利用的小碎片，可以在每次分配时优先使用最大的连续空闲区，这样分配后剩余的空闲区就不会太小，更方便使用。
   如何实现：空闲分区按容量递减次序链接。每次分配内存时顺序查找空闲分区链(或空闲分区表)，找到大小能满足要求的第-一个空闲分区。
4. **邻近适应算法**
   算法思想：首次适应算法每次都从链头开始查找的。这可能会导致低地址部分出现很多小的空闲分区，而每次分配查找时，都要经过这些分区，因此也增加了查找的开销。如果每次都从上次查找结束的位置开始检索，就能解决上述问题。
   如何实现：空闲分区以地址递增的顺序排列(可排成一个循环链表)。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(或空闲分区表)，找到大小能满足要求的第一个空闲分区。

算法	算法思想	分区排序	优缺点
首次适应	从低地址查找合适空间	地址递增排列	综合性能最好，开销小，不需要对空间空闲分区重排序
最佳适应	优先使用最小空闲空间	容量递增排列	更容易满足大进程需求，小碎片多，开销大，需要重排序

算法	算法思想	分区排序	优缺点
最坏适应	优先使用最大连续空间	容量递减排序	小碎片少，不利于大进程，开销大
邻近适应	从上次查找处向后查找	地址递增排列（循环链表）	不用每次从链表头查找，开销小，会使高地址大分区被用完

✨内存非连续内存分配管理方式

基本分页存储管理方式

将内存空间划分为一个个大小相等的分区，每个分区称为一个页框（物理块），每个页框有一个编号，即页框号，页框号从0开始。之后把一个进程也分为与页框大小相等的一个个部分，每个部分称为一个页，每个页也有一个编号称为页号，页号也是从0开始。

进程的每一个页面分别放入到一个页框中，即进程的页面与内存的页框有一一对应关系，各个页面不需要连续存放，可以放到不相邻的各个页框中。

进程的最后一个页面可能没有一个页框那么大，也就是说，分页存储可能产生内部碎片。

如果要访问逻辑地址A，则：

1. 确定逻辑地址A对应的页号P（页号=逻辑地址/页面大小）
2. 找到P号页面在内存中的起始地址（查页表）
3. 确定逻辑地址A的页内偏移量W（页内偏移量=逻辑地址%页面大小）

逻辑地址A对应的物理地址=P号页面在内存中的起始地址+页内偏移量W

如果每个页面大小为 2K2$\square$ B，用二进制表示逻辑地址，则末尾 K$\square$ 位即为页内偏移量，其余部分就是页号。如果有 K$\square$ 位表示“页内偏移量”，则说明该系统中一个页面的大小是 2K2$\square$ 个内存单元；如果有 M$\square$ 位表示“页号”，则说明在该系统中，一个进程最多允许有 2M2$\square$ 个页面。

页表：操作系统为每个进程建立一张页表，进程的每个页面对应一个页表项，每个页表项由“页号”和“块号”组成，页表记录进程页面和实际存放的内存块之间的映射关系。

地址变化：

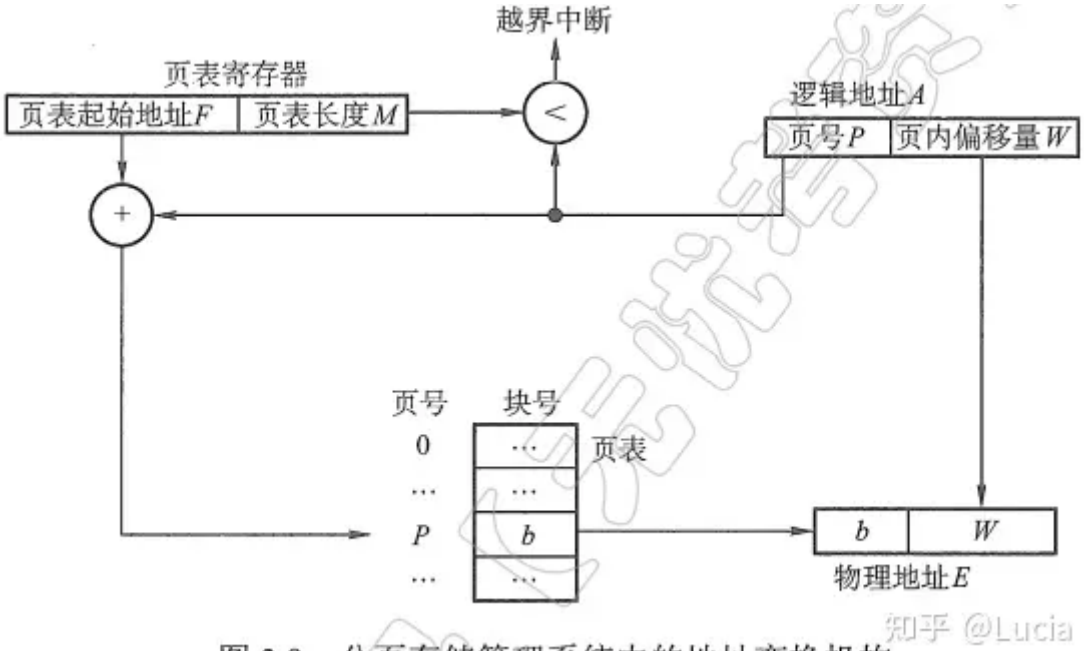


图 3.9　分页存储管理系统中的地址变换机构

1. 根据逻辑地址计算出页号、页内偏移量
2. 判断页号是否越界
3. 查询页表，找到页号对应的页表项，确定页面存放的内存块号
4. 用内存块号和页内偏移量得到物理地址
5. 访问目标内存单元

快表：一种访问速度比内存快很多的高速缓存，用来存放最近访问的页表项的副本，可以加快地址变换速度。若快表命中，就不需要访问内存了。若快表中没有目标页表项，则需要查询内存中的页表。

两级页表：将页表再次进行划分。在一段时间内并非所有页面都用得到，因此没必要让整个页表常驻内存，将长长的页表再分页，减少页表在内存中占用空间。

优点：内存空间利用率高，不会产生外部碎片，只会有少量页内碎片。

缺点：不方便按照逻辑模块实现信息的共享和保护。

基本分段存储管理方式

进程的地址空间，按照程序自身的逻辑关系划分为若干个段，每个段都有一个段名，每段从0开始编址。

内存分配规则：以段为单位进行分配，每个段在内存中占据连续空间，但各段之间可以不相邻。

逻辑地址结构：分段系统的逻辑地址由段号（段名）和段内地址（段内偏移量）组成。段号的位数决定每个进程最多可分为几个段，段内地址位数决定了每个段的最大长度是多少。

段表：每个进程建立一张段表，每个段对应一个段表项，记录该段在内存中的起始位置。

地址变换：

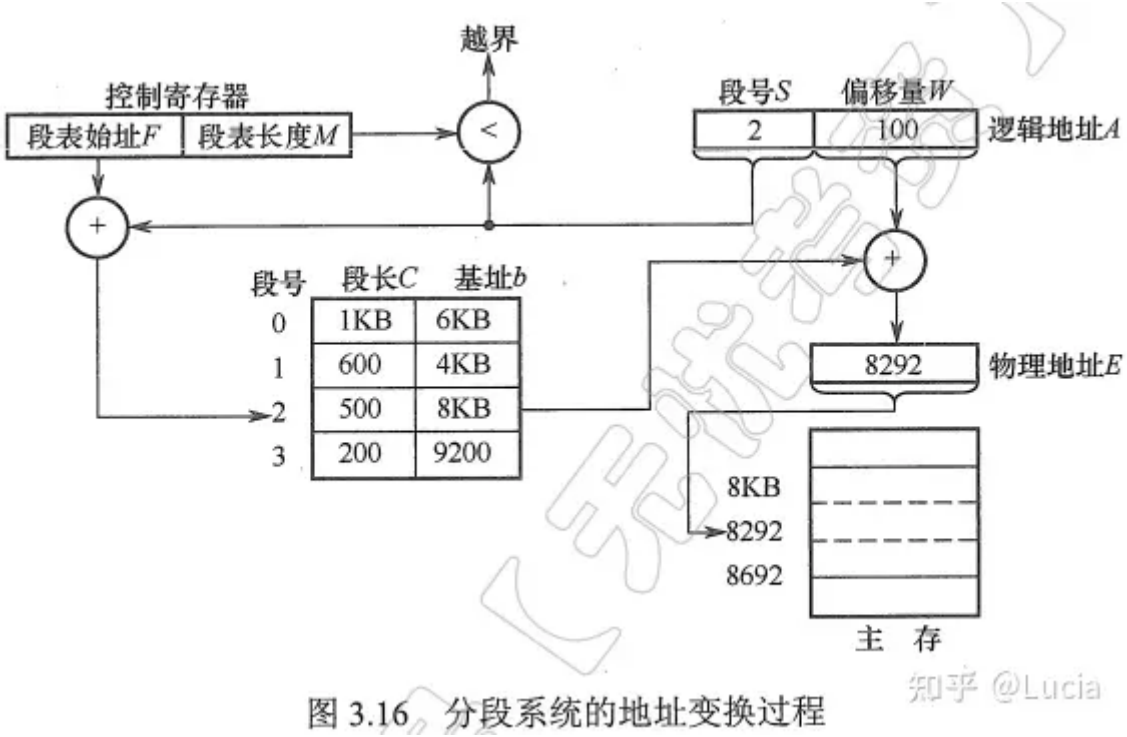


图 3.16 分段系统的地址变换过程

1. 根据逻辑地址得到段号和段内地址
2. 判断段号是否越界，若 S≥M时 则产生越界
3. 查询段表，找到对应段表项
4. 检查段内地址是否超过段长，若 W≥C时 则产生越界中断
5. 计算得到物理地址

6. 访问目标内存单元

优点：很方便按照逻辑模块实现信息的共享和保护。

缺点：如果段长过大，为其分配很大的连续空间会很不方便，另外，段式管理会产生外部碎片。

分段和分页管理方式对比

1. 分页的主要目的是为了实现离散分配，提高内存利用率，分页仅仅是系统管理上的需要，完全是系统行为，对用户是不可见的。
   分段的主要目的是更好的满足用户需求，一段通常包含一组属于一个逻辑模块的信息，分段对用户是可见的，用户编程时需要显示的给出段名。
2. 页的大小固定且由系统决定，段的长度不固定，取决于用户编写的程序。
3. 分页的用户地址空间是一维的，程序员是需要给出一个记忆符即可表示一个地址。分段的用户进程地址空间是二维的，程序员在标识一个地址时，既要给出段名，也要给出段内地址。
4. 分段比分页更容易实现信息的共享和保护，不能被修改的代码称为纯代码或可重入代码，这样的代码是可以共享的，只需要让各个进程的段表项指向同一个段即可实现共享。
5. 分页和分段都需要两次访存（查内存中的页表/段表、访问目标内存单元）。分页和分段系统都可以引入快表机制，将近期访问过的页表项或段表项放到快表中，可以少一次访存，加快地址变换速度。
6. 分页会产生内部碎片，分段会产生外部碎片。

段页式管理方式

进程的地址空间首先被分为若干段，每段有自己的段号，然后将每段分成若干大小固定的页，内存空间和分页存储管理方式一样，将其分为若干和页面大小相等的物理块。

地址结构：段页式系统的逻辑地址由段号、页号、页内地址（页内偏移量）组成。段号的位数决定每个进程最多可以分多少段，页号的位数决定每个段最多有多少页，页内偏移量决定了页面大小、内存块大小是多少。

段表 + 页表：每个段对应一个段表项，每个段表项由段号、页表长度、页表存放块号（页表起始地址）组成。每个页面对应一个页表项，每个页表项由页号、页面存放的内存块号组成。每个页表项长度相等。

地址变换：

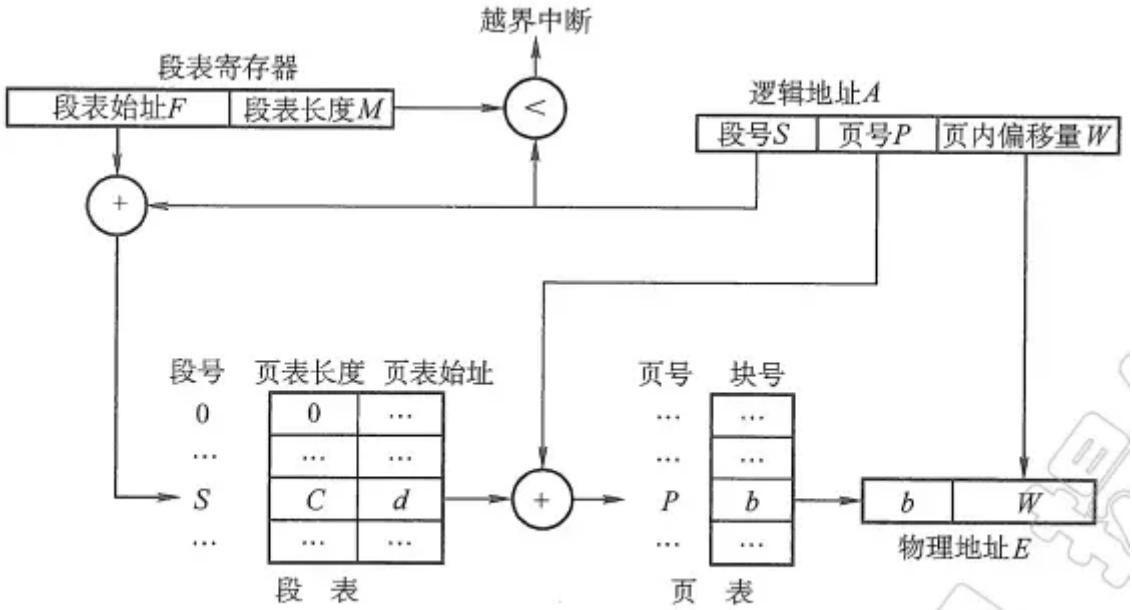


图 3.19 段页式系统的地址变换机构

1. 根据逻辑地址得到段号、页号、页内偏移量
2. 判断段号是否越界
3. 查找段表，找到对应段表项（第一次访存）
4. 检查页号是否越界
5. 根据页表存放块号、页号查询页表，找到对应的页表项（第二次访存）
6. 根据内存块号、页内偏移量得到最终的物理地址
7. 访问目标内存单元（第三次访存）

✨页面置换算法

- **最佳置换算法OPT**
  - 被淘汰的页面是以后永不使用的页面，或在最长时内不再被访问的页面
  - 优缺点：缺页率最小，性能最好，但无法实现
- **先进先出置换算法FIFO**
  - 优先淘汰最早进入内存的页面
  - 优缺点：实现简单，但性能很差，可能出现Belady异常（当为进程分配的物理块数增大时，缺页次数不减反增的异常现象）
- **最近最久置换算法LRU**
  - 选择最近最长时间未访问过的页面予以淘汰
  - 优缺点：性能很好，但需要硬件支持，算法开销大
- **时钟置换算法NRU**
  - 为每一个页面设置一个访问位，再将内存中的页面都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。当需要淘汰一个页面时，只需要检查页的访问位，如果是0就选择该页换出，如果是1则将它置0，暂不换出，继续检查下一个页面。若第一轮扫描中所有页面都是1，则将这些页面的访问位依次置0后再进行第二轮扫描，第二轮扫描中一定会有访问位为0的页面，因此简单的时钟置换算法选择一个淘汰页面最多会经过两轮扫描。
  - 优缺点：实现简单，算法开销小，但未考虑页面是否被修改过

- **改进型时钟置换算法**
  - 除了考虑一个页面最近是否被访问过之外，操作系统还考虑页面是否被修改过，在其他条件都相同时，应优先淘汰没有修改过的页面，避免I/O操作。
  - 增加修改位，修改位为0表示页面没有被修改过，修改位为1表示页面被修改过。用（访问位，修改位）的形式表示各页面状态，将所有可能被置换的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）页面用于替换，本轮扫描不修改任何标志位（最近没有被访问，且没有被修改）
  - 第二轮：若第一轮扫描失败，则重新扫描，查找第一个（0,1）页面用于替换，本轮将所有扫描过的页访问位设为0（最近没有被访问，但修改过的页面）
  - 第三轮：若第二轮扫描失败，则重新扫描，查找第一个（0,0）页面用于替换，本轮不修改任何标志位（最近访问过，但没有修改过的页面）
  - 第四轮：若第三轮扫描失败，则重新扫描，查找第一个（0,1）页面用于替换（最近访问过，且修改过的页面）
  - 优缺点：算法开销小，性能好

页面分配策略

- 驻留集（驻留在主存中页面数）大小
  - 分配空间小，进程数量多，CPU时间利用率高
  - 进程在主存中页数少，错页率就高
  - 进程在主存页数多，错页率并无明显改善
- 页面分配策略
  - 固定分配局部置换
  - 可变分配全局置换
  - 可变分配局部置换
- 固定分配 vs 可变分配：区别在于进程运行期间驻留集大小是否可变
  局部置换 vs 全局置换：区别在于发生缺页时是否只能从进程自己的页面中选择一个换出

调入页面的时机

- 预调页策略
  - 一次性调入若干相邻页面
  - 多用于进程首次调入
- 请求调页策略
  - 运行时发现缺页时调入
  - I/O开销大

✨磁盘调度算法

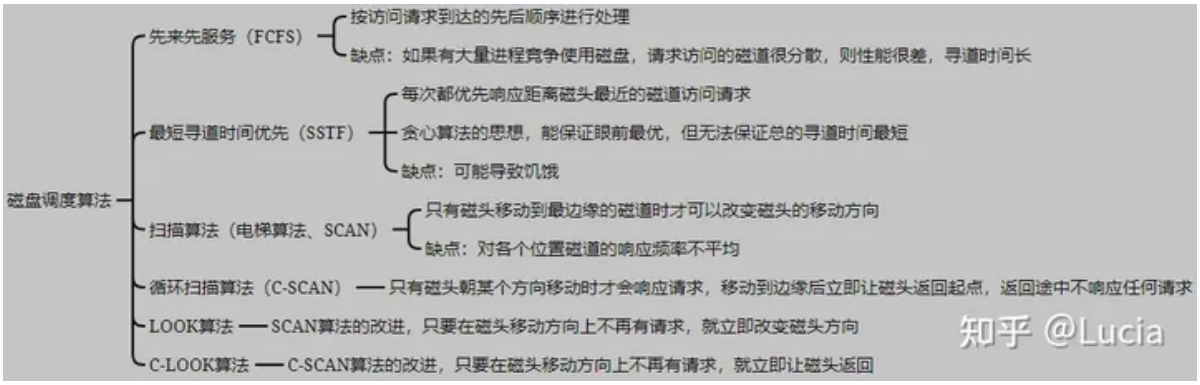
- 磁盘：磁盘的表面有一些磁性物质组成，可以用这些磁性物质来记录二进制数据
- 磁道：磁盘的盘面被划分为一个个磁道，一个圈就是一个磁道
- 扇区：一个磁道又被划分为一个个扇区，每个扇区就是一个磁盘块，各扇区存放的数据量相同，最内侧磁道上的扇区面积最小，数据密度最大
- 盘面：磁盘有多个盘片摞起来，每个盘片可能有两个盘面
- 柱面：所有盘面中相对位置相同的磁道组成柱面

- 磁盘的物理地址：（柱面号，盘面号，扇区号）

读写一个磁盘块的时间的影响因素有：

- 旋转时间（主轴转动盘面，使得磁头移动到适当的扇区上）
- 寻道时间（制动手臂移动，使得磁头移动到适当的磁道上）
- 实际的数据传输时间

其中，寻道时间最长，因此磁盘调度的主要目标是使磁盘的平均寻道时间最短。



✨文件存储空间管理（对空闲磁盘块的管理）

- **空闲表法**
  - 操作系统为磁盘外存上所有空闲区建立一张空闲表，每个表项对应一个空闲区。空闲表中记录每个连续空闲区的起始盘块号、盘块数。
  - 空闲表法适用于连续分配方式。
- **空闲链表法**
  - 空闲盘块链：以盘块为单位组成一条空闲链。
  - 空闲盘区链：以盘区为单位组成一条空闲链。
  - 空闲盘块链适用于离散分配的物理结构，空闲盘区链离散分配、连续分配都适用。
- **位示图法**
  - 每个二进制位对应一个盘块，当其值为0时，表示对应的盘块空闲，当其值为1时，表示对应的盘块已经分配。
  - 离散分配、连续分配都适用。
- **成组链接法**
  - 空闲表法、空闲链表法不适用于大型文件系统，因为空闲表或空闲链表可能过大。
  - 将空闲块分成若干组，每100个空闲块为一组，每组的第一空闲块登记了下一组空闲块的物理盘块号和空闲块总数。如果一个组的第二个空闲块号等于0，则有特殊的含义，意味着该组是最后一组，即无下一个空闲块。

✨I/O控制方式

程序直接控制方式

- CPU发出I/O命令后需要不断轮询，程序等待I/O操作完成后再继续往后执行。
- 实现简单，读/写后轮询状态寄存器，CPU和I/O设备串行，忙等。
- 每次读/写一个字。

中断驱动方式

- 在CPU发出读/写命令后，可将等待I/O的进程阻塞，先切换到别的进程执行，但I/O完成之后，控制器会向CPU发出一个中断信号主动报告I/O已完成，CPU不再需要不停的轮询。
- CPU和I/O设备可以并行，频繁中断耗时。
- 每次读/写一个字。

DMA方式

- 在I/O设备和内存之间开辟直接的数据交换通路，彻底解放CPU。所传输的数据是从设备直接送入内存的，或者相反；仅在传送一个或多个数据块开始和结束时才需要CPU干预，整块数据的传送是在DMA控制器的控制下完成的。
- 传输单位是数据块。
- CPU和I/O设备可以并行，每个I/O指令操作一个块，离散块需要频繁中断。

通道控制方式

- 通道是专门负责I/O的处理机。
- 把对一个数据块的读（或写）为单位的干预减少为对一组数据块的读（或写）及有关控制和管理为单位的干预。