


- 进程管理
  - 进程与线程
    - 进程概念
    -  进程的状态与转换
    - 进程控制（创建、终止、阻塞、唤醒、挂起、激活）
      - 进程控制块 (Process Control Block, PCB)
      - 进程创建 (Process Creation)
      - 进程终止 (Process Termination)
    - 进程通信（共享存储系统；消息传递系统；管道与套接字通信）
      - 为什么要通信
      - 通信的三种方式
      - 消息传递
        - 概念
        - 直接通信与间接通信
      - 共享内存
        - 概念
        - 生产者-消费者问题
      - 管道
        - 概念
        - 普通管道
        - 命名管道
      - 客户端-服务器系统中的通信
        - Sockets
        - Remote Procedure Calls 远程过程调用
    - 线程概念与多线程模型
      - 概念
      - 多线程模型
        - Many-to-One
        - One-to-One
        - Many-to-Many
    - 用户态线程和内核态线程
    - 其他
  - 进程同步
    - 进程同步的基本概念
      - 背景
      - 临界区问题
      - 解决临界区问题的方法

- 操作系统中临界区的处理
- 锁
- 实现临界区互斥的基本方法（硬件实现方法, 不太重要）
  - 相关硬件
  - 自旋锁
- 🚀 信号量（绝对会考）
  - 概念
  - 实现
  - 使用
  - 示例（P（） and V（））
    - PV 原语
    - 保护的临界区
  - 信号量的问题
    - 信号量操作的不正确使用：
    - 可能出现的问题：
- 管程(考的不多)
  - 概念
  - 条件变量
- 🚀 经典同步问题
  - 有界缓冲（生产者-消费者）
  - 变量
  - 代码
    - 生产者
    - 消费者
  - 读者-写者
    - 概述
    - 数据结构
    - 变体
  - 哲学家进餐
    - 背景
    - 数据结构
    - 死锁处理
- CPU调度
  - 调度的基本概念
    - 突发周期
    - CPU 调度器
    - 分配器
  - 调度的基本准则（cpu利用率、吞吐量、周转时间）

- 🚀 典型（进程）调度算法
  - 先来先服务(FCFS)
  - 短作业（短进程、短线程）优先SJF
    - 概念
    - 计算
    - 如何确定下一个突发的len
    - 抢占版本的最短剩余时间优先调度算法
  - 时间片轮转 RR
  - 优先级调度
    - 概念
    - 例子
  - 多级队列
- 🚀 实时CPU调度(实时操作系统)
  - 概念
  - 基于优先级的调度
  - 单调速率调度 RMS（抢占式）
  - ? 最早截止时间优先 EDF（抢占式）
- Deadlocks死锁
  - 死锁的概念
    - 死锁概念
    - 形成死锁四个必要条件
    - 资源占用分配图
  - 死锁处理策略
  - 死锁预防
  - 🚀 死锁避免
    - 系统安全状态
    - 银行家算法
      - 过程
      - 例子
- 其他
  - 死锁检测算法总结
  - 死锁恢复
    - 进程终止
    - 资源抢占

# 进程管理

---

# 进程与线程

---

## 进程概念

基本定义：

- 进程是 **程序的执行实例**，是程序在执行期间的活动实体。
- 程序是存储在磁盘上的被动实体（如可执行文件），**当程序被加载到内存中并开始执行时，它成为一个进程。**

进程的组成部分：

- **程序代码（文本段/Text section）**：也称为代码段，包含程序的指令。
- **数据段（Data section）**：存储全局变量。
- **当前活动信息**：包括程序计数器和处理器寄存器。
- **堆栈（Stack）**：用于存储临时数据，如函数参数、返回地址和局部变量。
- **堆（Heap）**：存储运行时动态分配的内存。

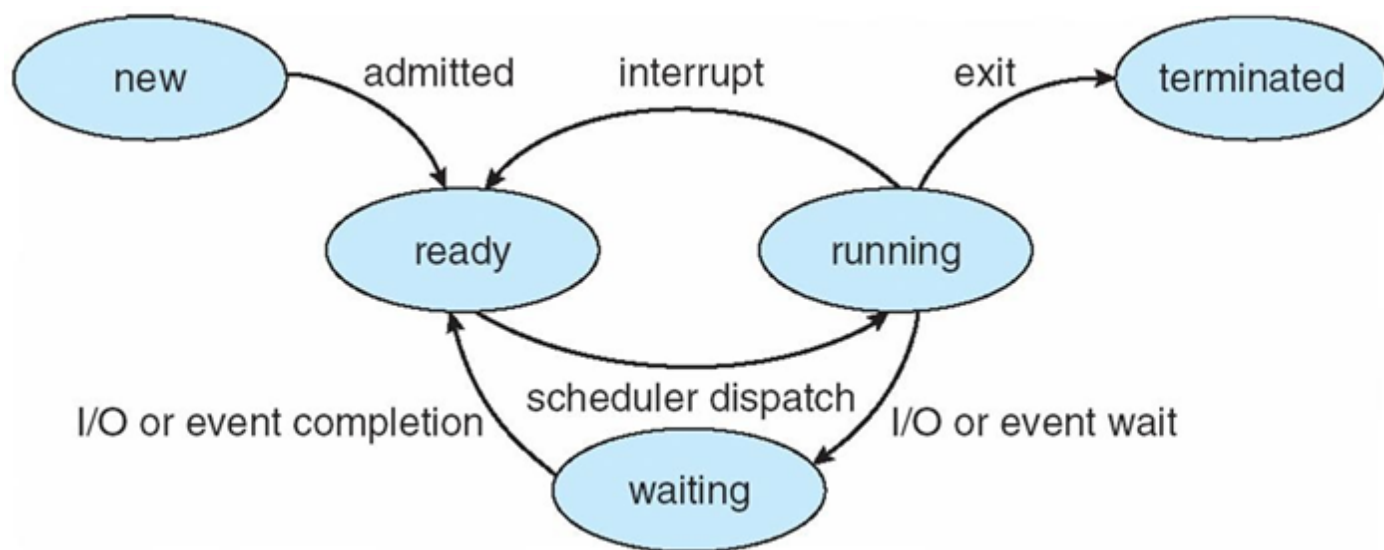
**一个程序可以有多个进程**：例如，多个用户可以同时执行同一个程序，每个执行实例都是一个独立的进程。

**启动方式**：进程的启动可能通过GUI操作（如鼠标点击）或命令行输入程序名称等方式。

**术语使用**：在一些情况下，"作业（job）"和"进程（process）"可以互换使用。

## 进程的状态与转换

- **new（新建）**：进程正在被创建。
- **running（运行）**：指令正在被执行。
- **waiting（等待）**：进程在等待某些事件发生。
- **ready（就绪）**：进程在等待被分配处理器。
- **terminated（终止）**：进程已完成执行。



## 进程控制（创建、终止、阻塞、唤醒、挂起、激活）

### 进程控制块 (Process Control Block, PCB)

每个进程都关联有如下信息（也称为任务控制块 Task Control Block）：

1. **进程状态**：如运行中、等待中等。
2. **程序计数器**：下一条将被执行的指令的位置。
3. **CPU寄存器**：保存与进程相关的所有寄存器内容。
4. **CPU调度信息**：包括优先级和调度队列指针。
5. **内存管理信息**：分配给进程的内存信息。
6. **会计信息**：如CPU使用时间、启动后经过的时间、时间限制等。
7. **I/O状态信息**：包括分配给进程的I/O设备以及打开文件的列表。

PCB是操作系统用来管理和调度进程的关键数据结构。

### 进程创建 (Process Creation)

**父进程 (Parent process) 创建子进程 (Children processes)**，子进程可以进一步创建其他进程，从而形成一个**进程树 (tree of processes)**。

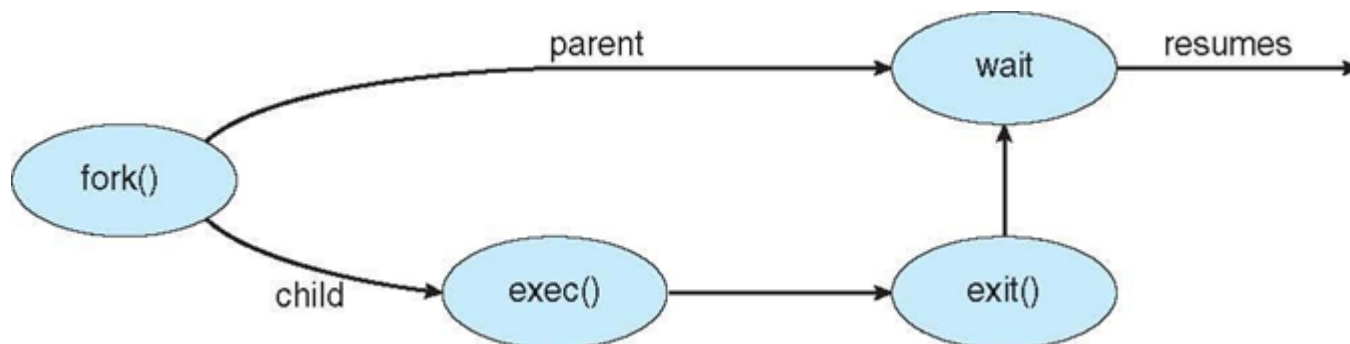
通常，通过进程标识符 (**PID**) 来标识和管理进程。

### 资源共享选项：

1. 父进程与子进程共享所有资源。
2. 子进程共享父进程资源的子集。
3. 父进程与子进程完全不共享资源。

## 执行选项：

1. 父进程和子进程并发执行。
2. 父进程等待子进程终止后再继续执行。



## 地址空间：

- 子进程是父进程的副本（完全复制父进程的地址空间）。
- 子进程可以加载新的程序到其地址空间中。

## UNIX中的系统调用：

- **fork()**：创建一个新进程（子进程）。
- **exec()**：在fork()之后调用，用一个新程序替换当前进程的内存空间。

## 进程终止 (Process Termination)

- **进程的正常终止**：进程执行完最后一条指令后，通过调用exit()系统调用请求操作系统删除进程。
  - 子进程会将状态数据返回给父进程（通过wait()实现）。
  - 进程使用的资源由操作系统回收和释放。
- **父进程终止子进程**：父进程可以通过abort()系统调用终止子进程的执，可能的原因包括：
  - 子进程超出了分配的资源。
  - 子进程完成的任务已不再需要。
  - 父进程正在终止，且操作系统不允许子进程在父进程终止后继续运行。
- **级联终止 (Cascading Termination)**：
  - 某些操作系统不允许子进程在其父进程终止后继续存在。
  - 如果一个进程终止，所有子进程及其后代进程（孙进程等）也必须被终止。
  - 此过程由操作系统发起。

- 父进程等待子进程：

- 父进程可以通过wait()系统调用等待子进程的终止。
- wait()返回被终止子进程的状态信息和进程标识符（PID）。

- 特殊进程状态：

- 如果父进程未调用 wait()等待子进程，子进程变为僵尸进程（zombie），即其终止信息未被回收。
- 如果父进程在未调用 wait()的情况下终止，子进程变为孤儿进程（orphan），由操作系统接管管理。

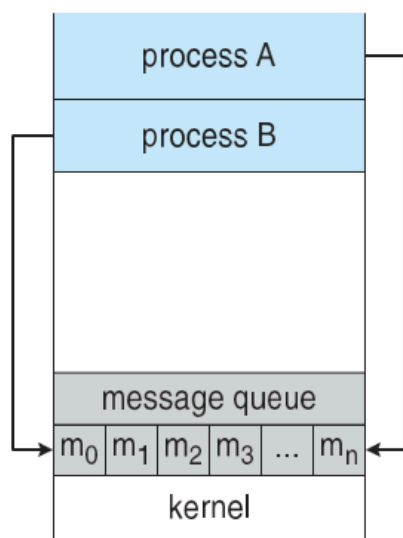
## 进程通信（共享存储系统；消息传递系统；管道与套接字通信）

### 为什么要通信

- Information sharing 信息共享
- Computation speed-up 计算加速
- Modularity 模块化

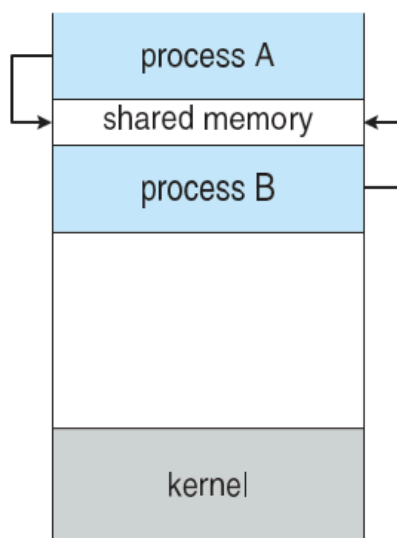
### 通信的三种方式

(a) Message passing.  
通过消息队列通信



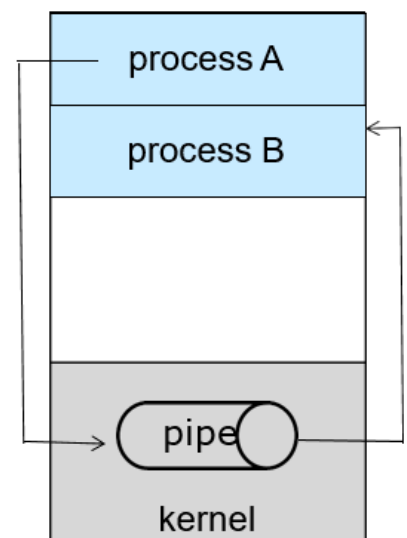
(a)

(b) shared memory.  
共享内存通信



(b)

(c) Pipe  
管道通信（只在内存中）



(c)

- **定义**：提供一种机制，使进程能够相互通信并同步其操作。
- **消息系统**：进程通过消息系统进行通信，而无需使用共享变量。
- **IPC功能**：提供两种基本操作：
  - **receive(message)**：接收消息
  - **send(message)**：发送消息。
- **消息大小**：消息的大小可以是固定的，也可以是可变的。
- **进程通信的基本要求**：如果进程 P 和 Q 想要通信，它们需要：
  - 建立一个 **通信链接 (communication link)** 。
  - 通过**send/receive**操作交换消息。
- **通信链接的实现**：
  - **物理层 (Physical)** ：
    - 共享内存 (Shared memory) 。
    - 硬件总线 (Hardware bus) 。
    - 网络 (Network) 。
  - **逻辑层 (Logical)** ：
    - 直接或间接通信 (Direct or indirect) 。
    - 同步 (Synchronous) 或异步 (Asynchronous) 。
    - 自动缓冲 (Automatic buffering) 或显式缓冲 (Explicit buffering) 。

直接通信与间接通信

特性	直接通信 (Direct Communication)	间接通信 (Indirect Communication)
通信方式	进程必须显式地命名对方 (如: <code>send(P, message)</code> 和 <code>receive(Q, message)</code> ) 。	通过共享的邮箱 (mailbox) 或端口 (port) 传递消息。
通信链接	链接自动建立。	链接仅在共享邮箱时建立。



特性	直接通信 (Direct Communication)	间接通信 (Indirect Communication)
的建立		
链接的唯一性	每个链接只对应一对通信进程。	一个链接可以与多个进程相关联。
多链接支持	每对通信进程之间仅存在一个链接。	每对进程之间可以共享多个链接 (通过多个邮箱) 。
消息接收和发送	直接从发送进程到接收进程。	消息通过邮箱进行中转，发送和接收进程不直接交互。
灵活性	较低：发送和接收进程需明确指向对方。	较高：通过共享邮箱的机制支持多对多通信。

### 邮箱共享 (Mailbox Sharing)

进程P1、P2、P3 共享同一个邮箱 A。P1 发送消息，P2和 P3接收消息。

**问题：** 消息由谁接收？

**解决方案：**

1. 限制一个链接最多只能与两个进程关联。
2. 每次仅允许一个进程执行接收操作。
3. 系统任意选择一个接收进程，并通知发送者消息接收者是谁。

### 共享内存

概念

- 一块内存区域被希望通信的进程共享。
- 通信由用户进程控制，而不是操作系统。
- 主要问题是提供一种机制，使用户进程在访问共享内存时能够同步它们的操作。

- 是共享内存通信的经典问题，用于描述一个生产者进程生产数据，消费者进程消费数据的场景。
- 包括两种模型：
  - **无界缓冲区 (Unbounded Buffer)** ：缓冲区大小无限，生产者不会因缓冲区满而阻塞。
  - **有界缓冲区 (Bounded Buffer)** ：缓冲区大小固定，需考虑同步和阻塞问题。

## 共享缓冲区代码

```
#define BUFFER_SIZE 10 // 缓冲区的大小
typedef struct {
    // 数据结构定义，假设为任意数据类型
    ...
} item;

item buffer[BUFFER_SIZE]; // 缓冲区数组
int in = 0; // 生产者放入数据的下标
int out = 0; // 消费者取出数据的下标
```

Solution is correct, but can only use BUFFER\_SIZE-1 elements

## 生产者代码

```
item next_produced; // 定义生产的下一个数据

while (true) {
    /* 生产一个数据项 */
    next_produced = ...; // 生成数据逻辑

    /* 缓冲区满时，阻塞 */
    while (((in + 1) % BUFFER_SIZE) == out)
        ; // 什么也不做，等待缓冲区有空位

    /* 将数据放入缓冲区 */
    buffer[in] = next_produced;

    /* 更新 in 指针 */
    in = (in + 1) % BUFFER_SIZE;
}
```

In: 放在哪，生产者

Out: 去哪里拿，消费者

## 消费者代码

```
item next_consumed; // 定义要消费的下一个数据

while (true) {
    /* 缓冲区空时，阻塞 */
    while (in == out)
        ; // 什么也不做，等待缓冲区有数据

    /* 从缓冲区取出数据 */
    next_consumed = buffer[out];

    /* 更新 out 指针 */
    out = (out + 1) % BUFFER_SIZE;

    /* 消费数据项 */
    ... // 消费数据逻辑
}
```

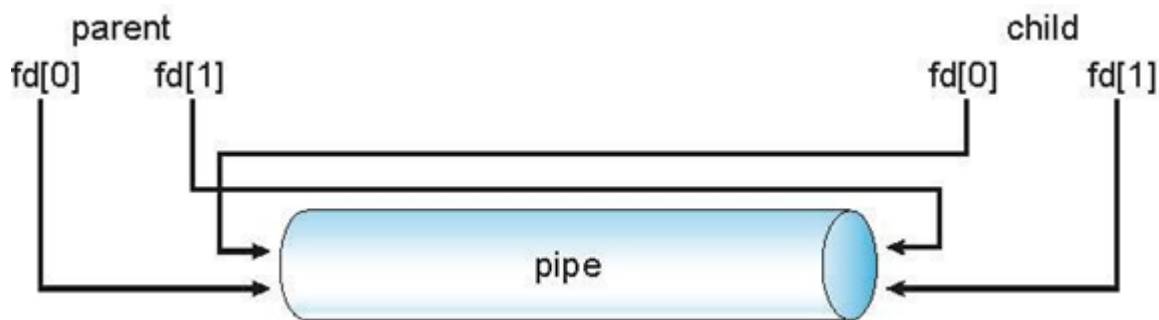
问题：可能in++或者out++执行时间到被timer停了

## 管道

### 概念

- **定义：** 管道充当一个通道，允许两个进程进行通信。
- **问题：**
  - 通信是单向的 (unidirectional) 还是双向的 (bidirectional) ?
  - 如果是双向通信，是半双工 (half-duplex) 还是全双工 (full-duplex) ?
  - 通信的进程之间是否必须存在某种关系 (例如，父子关系) ?
- **类型：**
  - **普通管道 (Ordinary pipes)：**
    - 不能被创建它的进程外部访问。
    - 通常由父进程创建，用于与它创建的子进程通信。
  - **命名管道 (Named pipes)：**
    - 可以在没有父子关系的情况下被访问。

### 普通管道



- 普通管道允许以标准的生产者-消费者模式进行通信。
- **生产者**将数据写入管道的一端（管道的写端）。
- **消费者**从管道的另一端（管道的读端）读取数据。
- 因此，普通管道是 **单向的**（unidirectional）。
- 通信的进程之间必须有 **父子关系**。
- 在Windows系统中，这些管道被称为 **匿名管道（anonymous pipes）**，只能用于父子进程之间的通信。

#### 命名管道

- 命名管道比普通管道更强大。
- 支持 **双向通信**（bidirectional）。
- 通信的进程之间 **不需要父子关系**。
- 多个进程可以使用同一个命名管道进行通信。
- 命名管道在UNIX和Windows系统中都支持。

#### 客户端-服务器系统中的通信

##### Sockets

- **定义**：Socket（套接字）被定义为通信的端点。
- **地址组成**：
  - 由 **IP地址** 和 **端口号（port）** 组合而成。
  - 端口号是消息包开始部分包含的数字，用于区分主机上的网络服务。
  - **示例**：Socket地址 **161.25.19.8:1625** 表示主机 **161.25.19.8** 上的端口 **1625**。
- **通信方式**：通信在一对套接字之间进行。
- **端口号**：端口号低于1024的为 **知名端口**（well-known ports），用于标准服务。

- **特殊地址：**

- **127.0.0.1**（环回地址，loopback）是一个特殊的IP地址，指向运行该进程的本地系统。
- 通过该地址，计算机上的进程可以与同一台计算机上的其他进程通信，而无需通过网络。

Remote Procedure Calls 远程过程调用

**定义：**

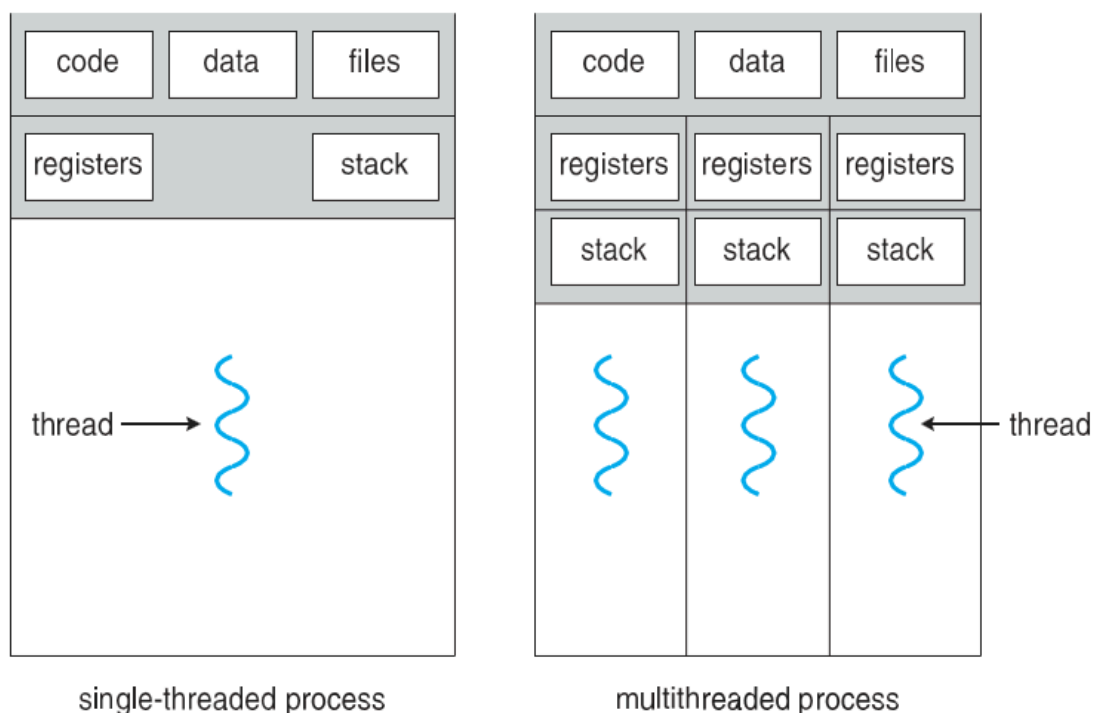
- RPC抽象了网络系统中进程之间的过程调用，使得远程调用看起来像本地函数调用。
- 使用端口号区分服务。

**Stub机制：**

- **Stub**是客户端和服务端之间的代理：
  - **客户端Stub**：定位服务器并封装（marshal）参数。
  - **服务器Stub**：接收消息，解封（unmarshal）参数，并在服务器上执行相应的过程。

## 线程概念与多线程模型

**概念**



如果一个进程fork(), 会把父进程的all空间复制一份、如果不复制code/data/files, 只复制寄存器/stack等, 即创建了一个线程

相比于进程, 简化了公共的部分



## 线程相比于进程的优点：

**响应性：**线程允许在某些部分阻塞时继续执行其他部分，特别适用于用户界面，提高系统响应能力。

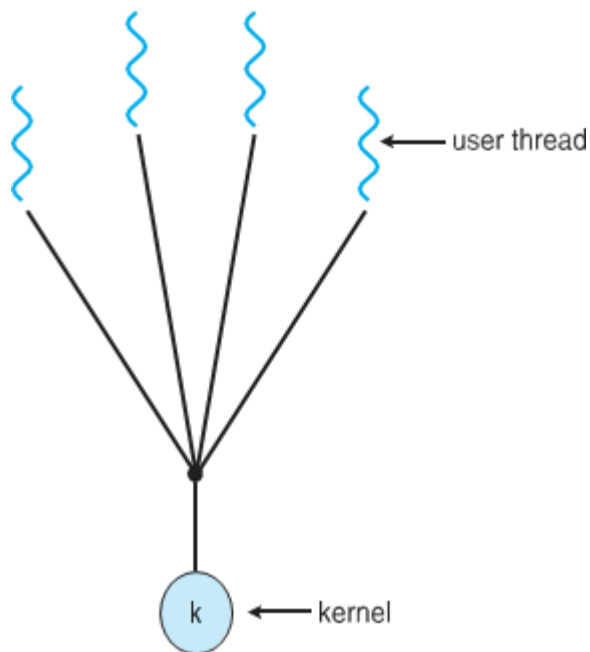
**资源共享：**同一进程内的线程共享资源，比通过共享内存或消息传递的方式更容易实现。

**经济性：**创建线程比创建进程开销更低，线程切换的开销也低于进程上下文切换。

**可伸缩性：**线程可以更好地利用多处理器架构，通过并行执行提高性能。

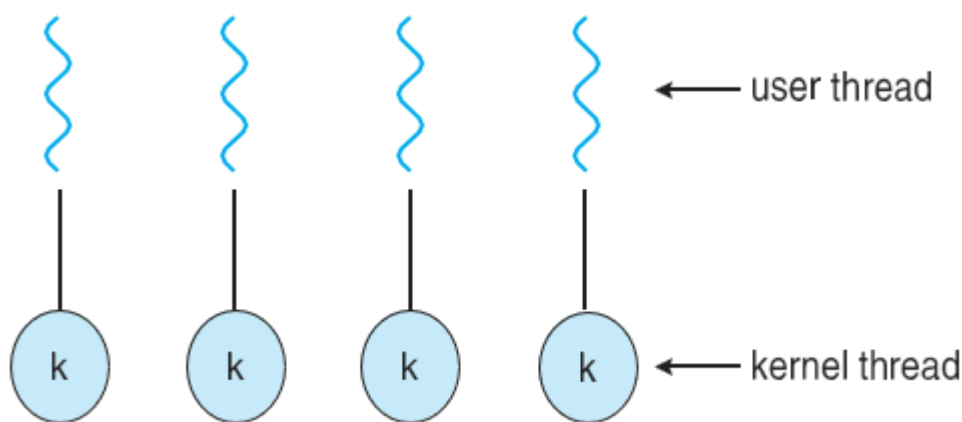
## 多线程模型

### Many-to-One



- 多个用户态线程映射到一个内核线程。
- 用户线程的管理在用户空间完成。
- **优点：** 线程操作（如创建、切换）不需要内核参与，效率较高。
- **缺点：**
  - 一个用户线程阻塞会导致所有线程阻塞。
  - 不能在多核系统中并行执行。
- **应用：** 很少使用。

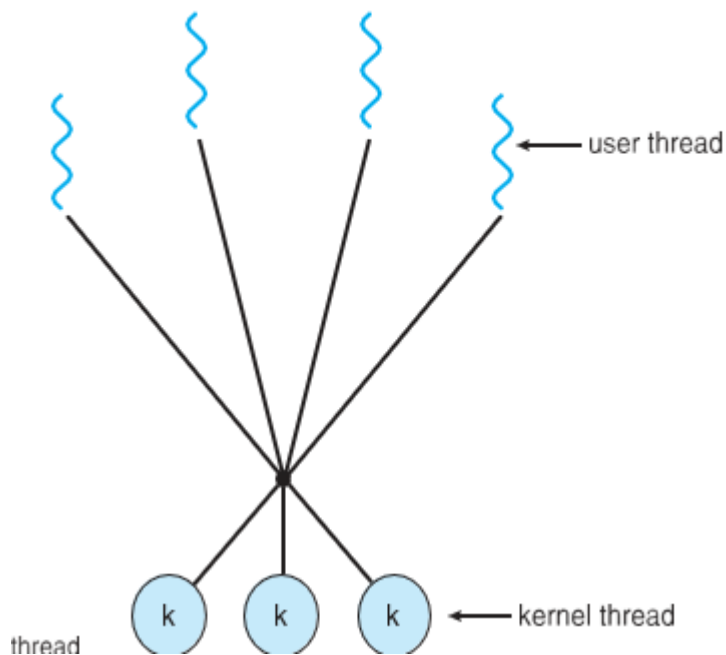
#### One-to-One



- 每个用户线程都映射到一个内核线程。
- 线程的创建会创建一个对应的内核线程。
- **优点：**
  - 支持真正的并行（多核系统）。
  - 一个线程阻塞不会影响其他线程。
- **缺点：**
  - 开销较大：线程数多时可能导致性能下降。

- **应用：**
  - 典型操作系统如 Windows 和 Linux 使用该模型。

#### Many-to-Many



- 多个用户线程映射到多个内核线程。
- 映射关系由操作系统决定，灵活性较高。
- **优点：**
  - 结合了多对一和一对一模型的优点。
  - 支持并行执行，并且用户线程不会因为内核线程限制而阻塞。
- **缺点：**
  - 实现复杂度较高。
- **应用：**
  - 少数系统支持，如一些研究型操作系统。

## 用户态线程和内核态线程

- 用户线程 — 管理由用户级线程库完成
- 内核线程 — 由内核支持



### 1. 用户级线程 (User threads) :

- 用户级线程是由用户空间的线程库管理的，而不是由操作系统内核直接管理。
- 用户级线程的创建、同步、调度和销毁等操作都在用户空间进行，不需要内核的介入。
- 由于这些操作不涉及系统调用，因此用户级线程的开销相对较小，可以创建大量线程而不会显著影响性能。
- 但是，用户级线程的一个主要缺点是它们不受内核支持，这意味着如果一个线程阻塞，整个进程中的所有线程都会阻塞，这种现象称为“线程间死锁”。
- 用户级线程的一个例子是早期的POSIX线程 (pthreads) 实现，它们完全在用户空间运行。

### 2. 内核级线程 (Kernel threads) :

- 内核级线程是由操作系统内核直接支持和管理的。
- 这些线程的创建、调度和销毁等操作需要通过系统调用来完成，因此它们的开销相对较大。
- 内核级线程的优点是每个线程都可以独立于其他线程运行，一个线程的阻塞不会影响到同一进程中的其他线程。
- 内核级线程可以充分利用多处理器系统的优势，因为内核可以在同一时间在不同处理器上调度不同的线程。

## 其他

### 线程管理的三种方法：

#### 1. 线程池 (Thread Pools) :

- 预先创建一组线程，这些线程可以重复使用。
- 优点：
  - 避免频繁创建和销毁线程的开销。
  - 提高资源利用率和性能。
  - 限制线程数量，防止系统资源耗尽。
- 使用场景：服务器应用程序、任务密集型程序。

#### 1. OpenMP :

- 一种支持多线程并行编程的API，用于共享内存系统。
- 编译器和运行时库自动管理线程，程序员只需使用指令（如 `#pragma omp`）指定并行区域。
- 优点：
  - 简化多线程开发，减少程序员负担。
  - 适合科学计算和数据密集型程序。

#### 1. Grand Central Dispatch (GCD) :

- Apple引入的一种任务管理框架，基于任务队列模型。

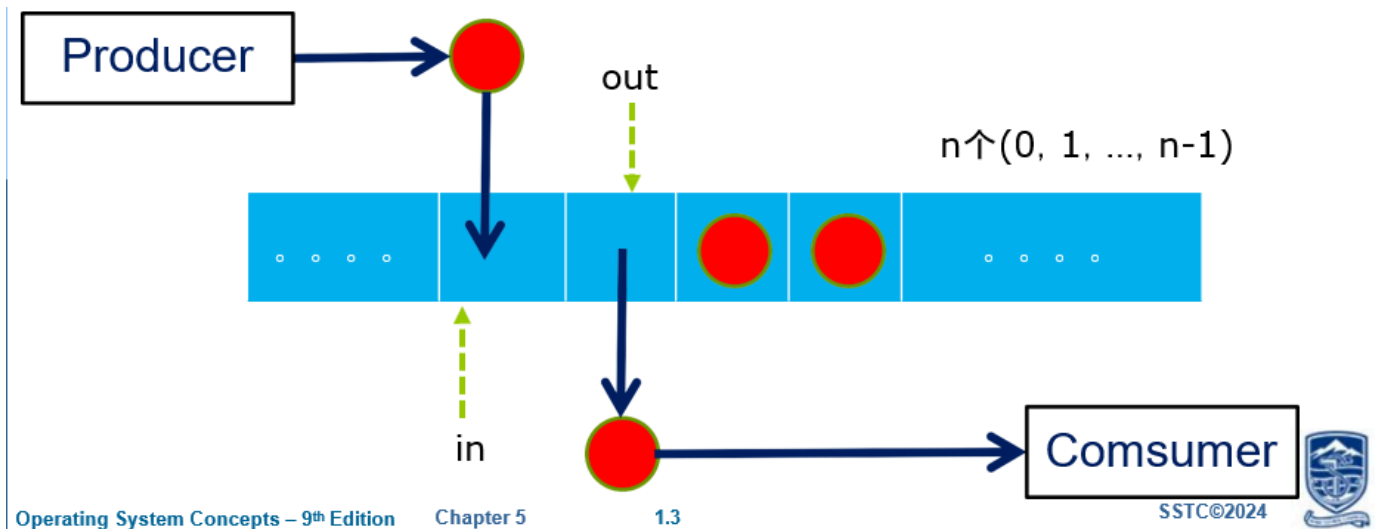
- 程序员将任务提交到系统提供的并发队列，GCD负责分配线程。
- 优点：
  - 自动管理线程，避免死锁和资源竞争。
  - 支持多核处理，优化并行性能。
- 使用场景：iOS和macOS应用程序。

# 进程同步

## 进程同步的基本概念

### 背景

- **进程可以并发执行**：可能随时被中断，部分完成执行。
- **共享数据的并发访问**：可能导致数据不一致。
- **数据一致性的维护**：需要机制确保协作进程的有序执行。



### 临界区问题

- 假设系统有 $n$ 个进程： $\{p_0, p_1, \dots, p_{n-1}\}$
- 每个进程都有一个临界区（critical section）代码段：
  - 进程可能会更改共享变量（changing common variables）、更新表（updating table）、写文件（writing file）等。
  - 当一个进程处于临界区时，其他进程不能进入其临界区。
- **临界区问题的核心**：设计一个协议来解决多个进程同时访问临界区的问题。
- 进入和退出临界区的步骤：
  - 每个进程必须在进入临界区之前 **请求权限（entry section）**。
  - 可能在临界区代码之后有 **退出区（exit section）**。

- 其余代码在剩余区 (remainder section) 执行。

## 通用结构

```
do {  
    entry section  
    critical section  
    exit section  
    remainder section  
} while (true);
```

## 解决临界区问题的方法

1. **互斥 (Mutual Exclusion)** : 如果进程  $P_i$ 正在执行它的临界区代码, 那么其他进程不能进入它们的临界区。
2. **步进 (Progress)** : 如果没有进程在执行临界区, 并且存在一些希望进入临界区的进程, 则必须选择一个进程进入临界区, 这一选择不能无限期推迟。
3. **有限等待 (Bounded Waiting)** : 一个进程在请求进入临界区后, 等待的时间是有限的, 不会因为其他进程不断抢占而无限期推迟。

## 操作系统中临界区的处理

- 根据内核是否支持抢占模式 (Preemptive) 分为两种方式:
  - **抢占式 (Preemptive)** :
    - 允许进程在内核模式下运行时被抢占 (中断执行)。
  - **非抢占式 (Non-preemptive)** :
    - 进程运行直到退出内核模式、阻塞或主动放弃CPU。
    - 在内核模式下基本不会发生竞争条件 (race condition)。

## 各自的优缺点:

- **抢占式 (Preemptive)** :
  - **优点** : 提高系统的响应速度、更适合实时系统。

- **缺点**：增加了同步复杂性、可能引发竞争条件，需要额外的机制保护数据一致性。

- **非抢占式 (Non-preemptive)**：

- **优点**：简化了同步问题、内核模式下无需担心数据竞争。
- **缺点**：系统可能因为一个长时间运行的进程而变得不响应、不适合实时环境。

## 锁

- **锁的核心概念**：通过锁来保护临界区，防止多个进程或线程同时访问共享资源。
- **计算机中锁的定义**：一种布尔变量 (boolean variable)。
  - 值为 **True** 时表示加锁 (lock)。
  - 值为 **False** 时表示解锁 (unlock)。
- 谁来加锁？谁来解锁？
  - 通常由操作系统或同步机制负责管理锁的状态。

```
do {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
} while (TRUE);
```

## 实现临界区互斥的基本方法（硬件实现方法, 不太重要）

### 相关硬件

### 同步硬件 (Synchronization Hardware)

- 许多系统提供硬件支持用于实现临界区代码的同步。
- **单处理器 (Uniprocessors)**：可以通过**关闭中断**来实现同步。
  - 当前运行的代码在没有抢占的情况下执行。

- **缺点**：在多处理器系统中效率较低，不适用。
- **现代机器**：提供特殊的 **原子硬件指令 (Atomic Instructions)** 。
  - **原子操作 (Atomic)**：不可中断的操作。
    - 例如：测试内存中的值并设置新值。
    - 或者交换两个内存位置的内容。

## 自旋锁

- **特点**：
  - 硬件同步解决方案需要 **忙等待 (busy waiting)** 。
  - 因此，这种锁被称为自旋锁 (spinlock) 。
- **优缺点**：
  1. **无上下文切换 (No context switch)**：
    - 提高性能，因为不会发生线程的上下文切换。
  2. **持续占用CPU (Persistent CPU usage)**：
    - 在锁未释放前会持续消耗CPU资源。
- **适用范围**：通常由操作系统内核和设计者使用，适用于锁等待时间短的场景。

**总结**：自旋锁适用于高性能要求但锁竞争时间较短的环境，尽管会导致忙等待，但避免了上下文切换的开销。

## 信号量（绝对会考）

### 概念

- 之前的解决方案（如自旋锁）复杂且通常 **对应用程序开发者不可用** 。
- 操作系统设计者构建了**软件工具**，提供更精细的同步方式（比互斥锁更高级），以解决临界区问题。
- **信号量定义**：
  - 信号量**S**：一个整数变量（ **integer variable** ）。
- **信号量分类**：
  1. **计数信号量 (Counting Semaphore)**：
    - 整数值可以在无限范围内变化。
    - 适合管理多个共享资源。
  2. **二进制信号量 (Binary Semaphore)**：

- 整数值只能在 0 和 1 之间变化。
- 功能等同于互斥锁（ **mutex lock** ）。

## 实现

### 信号量结构定义：

```
typedef struct {  
    int value;           // 整数值  
    struct process *list; // 指向等待队列中下一个进程的指针  
} semaphore;
```

- 信号量包含的两个数据项：
  - **value**：整数类型，用于表示信号量的当前值。
  - **list**：指向等待队列中下一个记录的指针。
- 信号量的两种基本操作：
  - **block**：将调用该操作的进程放入对应的等待队列中。
  - **wakeup**：从等待队列中移除一个进程，并将其放入就绪队列中。
- 只能通过两个原子操作来accessed
  - **wait()**: 用于进程请求资源，如果资源不足，进程将被挂起 (P原语)
  - **signal()**: 释放资源，如果有等待的进程，将唤醒其中一个 (V原语)

## 使用

- 信号量的作用：可以解决各种同步问题。
- 示例：
  - 假设有两个进程 P1和 P2，需要保证 S1的执行发生在 S2之前：
    1. 创建一个初始值为 0 的信号量 **synch**。
    2. 在 P1中：

```
S_1;  
signal(synch);
```

3. 在 P2中：

```
wait(synch);
S_2;
```

- **互斥锁 (Mutex Locks) :**

- 用于实现对临界区的互斥访问。
- 关键问题：等待 (wait 前) 是否会造成忙等？是否会有锁队列 (lock 队列) ？
- 示例代码：

```
do {
    wait(lock);
    // critical section
    signal(lock);
    // remainder section
} while (TRUE);
```

## 示例 (P () and V () )

// 定义信号量操作函数 down 和 up，实现信号量的 P() 和 V() 操作。

```
void down(struct semaphore *sem) {
    unsigned long flags;

    // 获取自旋锁并保存中断状态，用于保护临界区
    raw_spin_lock_irqsave(&sem->lock, flags);

    // 如果信号量计数大于 0，表示资源可用
    if (likely(sem->count > 0))
        sem->count--; // 直接将信号量计数减一，代表占用一个资源
    else
        __down(sem); // 如果信号量计数小于等于 0，调用阻塞函数进入等待队列

    // 释放自旋锁并恢复中断状态
    raw_spin_unlock_irqrestore(&sem->lock, flags);
}

void up(struct semaphore *sem) {
    unsigned long flags;

    // 获取自旋锁并保存中断状态，用于保护临界区
    raw_spin_lock_irqsave(&sem->lock, flags);

    // 如果等待队列为空，表示没有进程等待
    if (likely(list_empty(&sem->wait_list)))
        sem->count++; // 增加信号量计数，表示释放一个资源
    else
        __up(sem); // 如果有等待进程，唤醒等待队列中的一个进程
```

```
// 释放自旋锁并恢复中断状态
raw_spin_unlock_irqrestore(&sem->lock, flags);
}
```

## PV 原语

- **P 原语 (down() 函数) :**
  - down() 是信号量的 P 操作，用于获取资源。
  - **保护的临界区：** 信号量的值 sem->count 和等待队列 sem->wait\_list。
- **V 原语 (up() 函数) :**
  - up() 是信号量的 V 操作，用于释放资源。
  - **保护的临界区：** 信号量的值 sem->count 和等待队列 sem->wait\_list。

## 保护的临界区

- raw\_spin\_lock\_irqsave() 和 raw\_spin\_unlock\_irqrestore()
- if (likely(sem->count > 0)) 和 else \_\_down(sem) (down() 内)
- if (likely(list\_empty(&sem->wait\_list))) 和 else \_\_up(sem) (up() 内) :

临界区是程序中一段需要独占访问共享资源（如变量、内存、文件等）的代码段。为了确保共享资源的一致性和正确性，在临界区内同一时刻只能有一个线程或进程执行操作。

## 信号量的问题

### 信号量操作的不正确使用：

- **信号和等待顺序错误：** signal(mutex) 在 wait(mutex) 之前调用。
- **重复等待：** 连续调用 wait(mutex) 而没有中间释放。
- **缺少操作：** 忽略了 wait(mutex) 或 signal(mutex)，或者两者都遗漏。

### 可能出现的问题：

- **死锁：** 多个进程因为资源的循环等待而无法继续。
- **饥饿：** 某些进程长时间无法获得资源。

## 管程(考的不多)



## 概念

管程也是用来解决进程同步互斥的一种工具，他的操作比信号量更加简单，管程封装了同步操作，对进程隐藏了同步细节。

- 高级抽象机制，提供了方便且有效的进程同步方式。
- 抽象数据类型：内部变量只能通过管程中的代码访问。
- 同一时刻，仅允许一个进程在管程中活动。

代码结构：

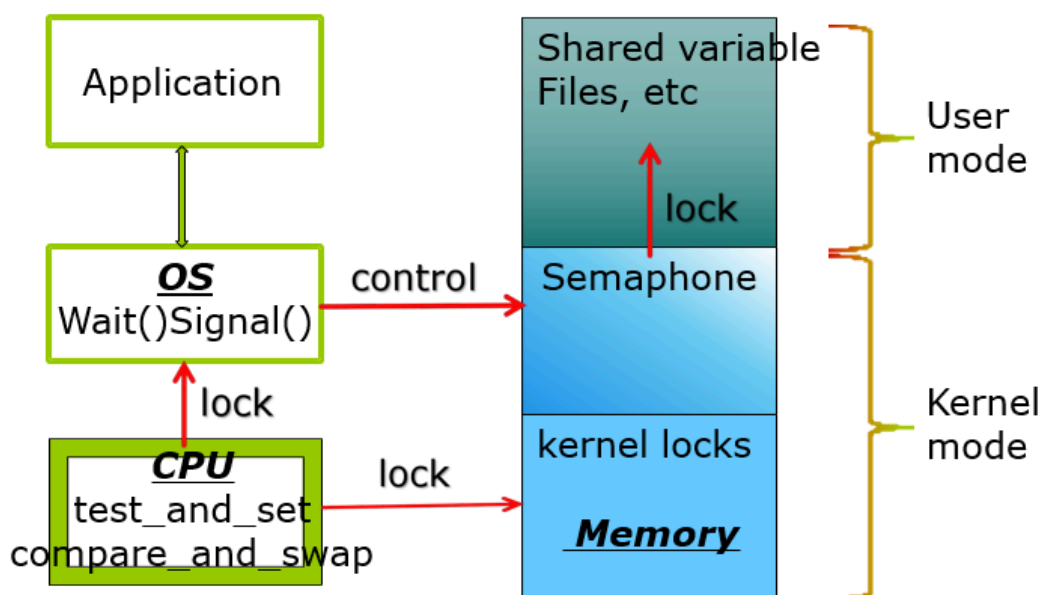
```
monitor monitor-name
{
    // 共享变量声明
    procedure P1 (...) { ... }
    procedure Pn (...) { ... }
    Initialization code (...) { ... }
}
```

## 条件变量

允许在条件变量上进行的两种操作：

- **x.wait()**：调用该操作的进程将被挂起，直到收到 **x.signal()**。
- **x.signal()**：恢复一个调用了 **x.wait()** 的进程（如果有的话）。
- 如果变量上没有 **x.wait()**，则此操作对变量没有任何影响。

## 🚀 经典同步问题



只要不是原子操作，其他任何指令都能被打断

## 有界缓冲（生产者-消费者）

### 变量

#### 1. mutex 信号量

- **作用：**用于确保生产者和消费者不能同时操作缓冲区中的共享数据，起到互斥锁的作用。它确保对共享缓冲区的访问是互斥的，即同一时刻只有一个生产者或消费者可以访问缓冲区。
- **初始化值：**mutex 初始化为 1，表示缓冲区最初是解锁的，任何生产者或消费者都可以申请进入操作。

#### 2. full 信号量

- **作用：**表示缓冲区中已经被填充的槽位数量。full 信号量用于控制消费者的操作，即确保消费者只有在缓冲区有物品时才能从中取出数据。
- **初始化值：**full 初始化为 0，表示缓冲区最初是空的，没有物品可供消费。

#### 3. empty 信号量

- **作用：**表示缓冲区中空闲的槽位数量。empty 信号量用于控制生产者的操作，即确保生产者只有在缓冲区有空位时才能往其中放入数据。
- **初始化值：**empty 初始化为 n，表示缓冲区最初有 n 个空槽，生产者可以开始放入物品。

### 代码

#### 生产者

```
do {  
    // 第一步：生产一个物品并存储在 next_produced 中  
    // 这里表示生产者的生产逻辑，将生成的物品存储到 next_produced。  
  
    wait(empty); // 第二步：对信号量 empty 执行 wait 操作（减1）。  
                // 确保缓冲区中至少有一个空槽可以存放物品。  
                // 如果缓冲区已满（empty == 0），生产者会在此等待。  
  
    wait(mutex); // 第三步：获取互斥锁（mutex）。  
                // 确保在访问共享缓冲区时，只有一个生产者或消费者可以操作，保证互斥访问。  
  
    // 第四步：将 next_produced 中的物品添加到缓冲区。  
    // 把生成的物品放入共享缓冲区。
```

```
    signal(mutex); // 第五步：释放互斥锁（mutex）。
                    // 允许其他生产者或消费者访问缓冲区。

    signal(full);   // 第六步：对信号量 full 执行 signal 操作（加1）。
                    // 表示缓冲区中有一个新的物品可供消费。
                    // 允许等待的消费者继续执行。
} while (true); // 不断重复以上过程。
```

## 消费者

```
do {
    wait(full); // 第一步：对信号量 full 执行 wait 操作（减1）。
                // 确保缓冲区中至少有一个物品可供消费。
                // 如果缓冲区为空（full == 0），消费者会在此等待。

    wait(mutex); // 第二步：获取互斥锁（mutex）。
                 // 确保在访问共享缓冲区时，只有一个生产者或消费者可以操作，保证互斥访问。

    // 第三步：从缓冲区中取出一个物品并存储到 next_consumed 中。
    // 将物品从共享缓冲区移除，准备消费。

    signal(mutex); // 第四步：释放互斥锁（mutex）。
                   // 允许其他生产者或消费者访问缓冲区。

    signal(empty); // 第五步：对信号量 empty 执行 signal 操作（加1）。
                   // 表示缓冲区中多了一个空槽可以供生产者使用。
                   // 允许等待的生产者继续执行。

    // 第六步：消费 next_consumed 中的物品。
    // 执行消费者的消费逻辑。

} while (true); // 不断重复以上过程。
```

## 读者-写者

### 概述

**读者-写者问题**是一种经典的同步问题，描述了多个线程访问共享数据的场景，涉及以下两类线程：

1. **读者**：只读取共享数据，不修改。
2. **写者**：可以读取并修改共享数据。

### 问题：

- 允许多个读者同时读取共享数据。
- 但在同一时刻只允许一个写者访问共享数据。

## 数据结构

- **rw\_mutex 信号量:**
  - 负责控制写操作，确保同一时刻只有一个写者可以访问共享数据集。
  - 初始化为 1，表示资源最初是可用的。
- **mutex 信号量:**
  - 负责控制对读者计数器 `read_count` 的访问，确保在更新 `read_count` 时是互斥的。
  - 初始化为 1，表示读者计数器可以被独占访问。
- **read\_count 整数变量:**
  - 表示当前正在读取共享数据的读者数量。
  - 初始化为 0，表示初始状态没有读者在读取数据。

## writer process

```
// 写者进程的结构
do {
    wait(rw_mutex); // 等待信号量 rw_mutex，确保没有其他写者或读者正在操作共享数据
    // 此处是执行写操作的代码部分
    /* 执行写操作 */
    signal(rw_mutex); // 写操作完成后释放信号量 rw_mutex，允许其他写者或读者进入
} while (true); // 写者进程持续运行
```

## reader process

```
// 读者进程的结构
do {
    wait(mutex); // 等待信号量 mutex，进入互斥区以修改读者计数器
    read_count++; // 增加读者计数器，表示有一个新的读者开始读取
    if (read_count == 1) { // 如果是第一个读者进入
        wait(rw_mutex); // 等待信号量 rw_mutex，阻止写者进入
    }
    signal(mutex); // 释放信号量 mutex，允许其他读者或写者修改读者计数器

    // 此处是执行读操作的代码部分
    /* 执行读操作 */
}
```

```
wait(mutex); // 再次等待信号量 mutex，进入互斥区以修改读者计数器
read_count--; // 减少读者计数器，表示一个读者离开
if (read_count == 0) { // 如果是最后一个读者离开
    signal(rw_mutex); // 释放信号量 rw_mutex，允许写者进入
}
signal(mutex); // 释放信号量 mutex，允许其他读者或写者修改读者计数器
} while (true); // 读者进程持续运行
```

## 变体

- **第一种变体**：除非写者已经获得使用共享对象的许可，否则不能让任何读者等待。
- **第二种变体**：一旦写者准备好，它应尽快执行写操作。
- **两者问题**：这两种变体可能都会导致饥饿问题，从而引发更多的变体。
- **解决方案**：在某些系统中，通过内核提供的读者-写者锁解决此问题。

## 哲学家进餐

### 背景

- 哲学家在生活中交替进行思考和进餐。
- 他们不会与邻座的哲学家互动，但偶尔会尝试从碗中进食，进食时需要拿起两根筷子（一次拿一根）。
  - 进食时需要两根筷子，进食完毕后需同时放下两根筷子。
- 对于五个哲学家的情况：
  - **共享数据**：设有一个信号量数组 `chopstick[5]`，每根筷子用一个信号量表示，初始值为 1，表示筷子可用。

### 数据结构

```
do {
    // 尝试拿起左边的筷子，使用信号量实现互斥
    wait (chopstick[i]);

    // 尝试拿起右边的筷子
    // 如果右边的筷子不可用，会一直等待，可能导致死锁
    wait (chopstick[(i + 1) % 5]);

    // 此时哲学家已经拿到两根筷子，可以开始进餐
    // 进餐逻辑在此处实现
    // ...

    // 放下左边的筷子，释放资源
    signal (chopstick[i]);
```

```
// 放下右边的筷子，释放资源
signal (chopstick[(i + 1) % 5]);

// 哲学家开始思考，不需要筷子
// ...
} while (TRUE); // 无限循环，哲学家在吃饭和思考之间切换
```

## 该算法的问题

1. **死锁 (Deadlock)** : 如果每个哲学家都同时拿起左边的筷子 (`wait(chopstick[i])`) , 然后等待右边的筷子 (`wait(chopstick[(i + 1) % 5])`) , 整个系统会陷入死锁状态。
2. **资源浪费 (Resource Starvation)** : 如果某些哲学家不断地成功拿到筷子, 其他哲学家可能会无限期地等待, 导致资源饥饿问题。

### 死锁处理

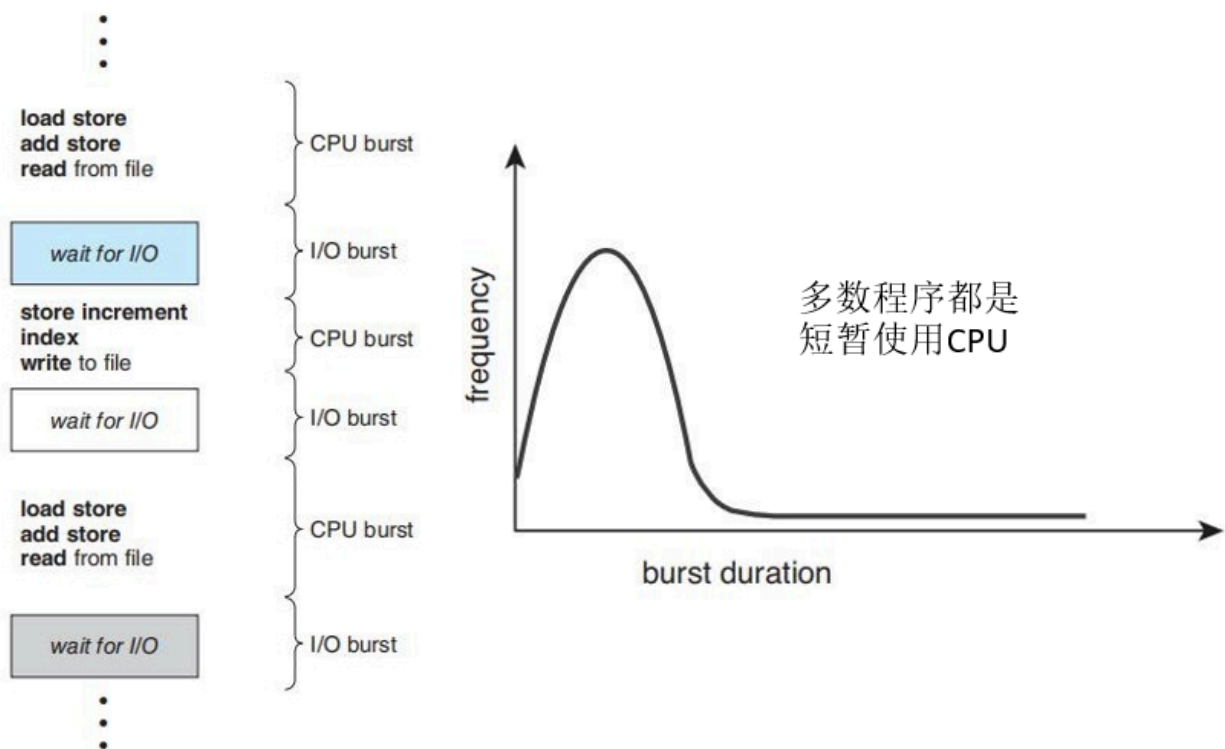
- **限制同时就坐的哲学家人数** : 最多允许4位哲学家同时坐在桌旁。
- **条件拿取筷子** : 允许哲学家仅在两根筷子都可用的情况下拿取 (拿取操作必须在临界区内完成) 。
- **使用非对称解决方案** :
  - 奇数编号的哲学家先拿起左边的筷子, 再拿起右边的筷子。
  - 偶数编号的哲学家先拿起右边的筷子, 再拿起左边的筷子。

# CPU调度

## 调度的基本概念

### 突发周期

- 通过多道程序设计实现最大化的 CPU 利用率 。
- CPU-I/O 突发周期 : 进程执行由 CPU 执行 和 I/O 等待 的周期组成。
- CPU 突发 (CPU burst) 后接 I/O 突发 (I/O burst) 。



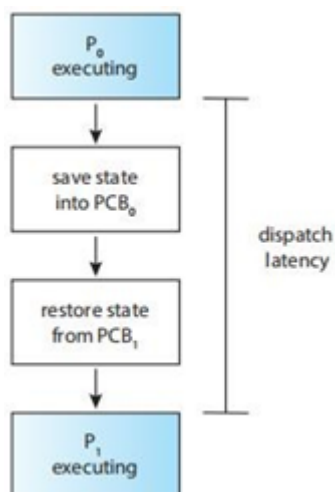
## CPU 调度器

- **短程调度器** 或 **CPU 调度**：从就绪队列中的进程选择一个，并将 CPU 分配给它。
  - 队列可以以不同方式排序。
- **CPU 调度决策可能发生的场景**：
  1. 进程从运行状态切换到等待状态。
  2. 进程从运行状态切换到就绪状态。
  3. 进程从等待状态切换到就绪状态。
  4. 进程终止。
- **场景 1 和 4 的调度为非抢占式调度 (nonpreemptive)。**
- **其他调度为抢占式调度 (preemptive)：**
  - 考虑对共享数据的访问。
  - 考虑内核模式下的抢占。
  - 考虑在操作系统关键活动中发生的中断。

## 分配器

- 分配器模块将 CPU 的控制权交给短程调度器选择的进程；这涉及：
  - 切换上下文 (switching context)
  - 切换到用户模式 (switching to user mode)
  - 跳转到程序的正确位置 (不一定是用户程序) 以重新启动该程序

- **分配延迟 (Dispatch latency)** : 分配器从停止一个进程到开始另一个进程所需的时间。



## 调度的基本准则 (cpu利用率、吞吐量、周转时间)

- **(MAX) CPU 利用率** : 保持 CPU 尽可能忙碌。
- **(MAX) 吞吐量** : 每单位时间内完成其执行的进程数量。
- **(MIN) 周转时间** : 从提交一个进程到完成该进程的时间间隔。
- **(MIN) 等待时间** : 一个进程在就绪队列中等待的时间总量。
- **(MIN) 响应时间** : 从提交请求到产生第一个响应所需的时间 (不包括输出的完成时间, 主要针对分时环境) 。

## 🚀 典型 (进程) 调度算法

先来先服务(FCFS)

<u>Process</u>	<u>Burst Time</u>
$P_1$	24
$P_2$	3
$P_3$	3

假设进程到达的顺序为:  $P_1$ ,  $P_2$ ,  $P_3$ 。时间表的甘特图为:





- 等待时间;  $P_1 = 0$ ;  $P_2 = 24$ ;  $P_3 = 27$
- 平均等待时间:  $(0 + 24 + 27)/3 = 17$

**护航效应 (Convoy effect)** : 当一个短进程被一个长进程阻塞时, 系统整体性能可能受到影响。例如, 一个计算密集型任务可能会导致大量 I/O 密集型任务被延迟处理。

### 短作业 (短进程、短线程) 优先SJF

#### 概念

#### 定义 :

- 每个进程与其下一次 CPU 运行所需时间 (CPU burst) 的长度相关联。
- 使用这些长度来调度具有最短运行时间的进程。

#### 特性 :

- SJF 是一种优化算法, 能够在给定的一组进程中实现最小的平均等待时间。

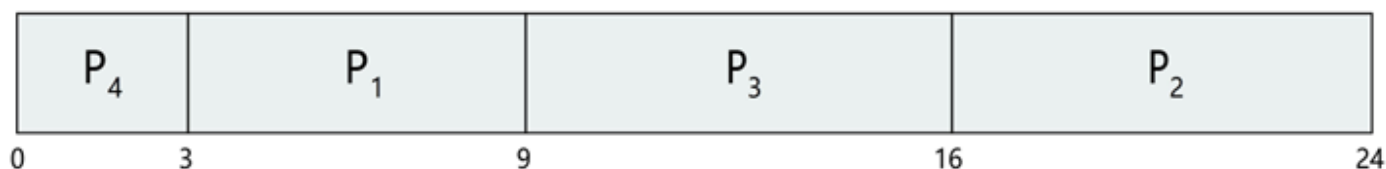
#### 困难 :

- 难点在于如何提前知道进程下一次 CPU 请求的长度。
- 可能需要通过用户提供此信息, 但实际实现中并不常见。

#### 计算


<u>Process</u>	<u>Burst Time</u>
$P_1$	6
$P_2$	8
$P_3$	7
$P_4$	3

#### 调度顺序



- 平均等待时间 =  $(3 + 16 + 9 + 0) / 4 = 7$
- FIFS平均等待时间 =  $(0 + 6 + 14 + 21) / 4 = 10.25$

如何确定下一个突发的len

 1735802235835

**只能估计长度：**

- 长度的估计值应与之前的 CPU burst 类似。
- 然后选择预测的下一次 CPU burst 最短的进程。

**估计方法：**

- 使用之前 CPU burst 的长度，通过 **指数平均** (Exponential Averaging) 计算。

**公式：**

- $t_n$ : 第  $n$  次 CPU burst 的实际长度。
- $\tau_{n+1}$ : 下一次 CPU burst 的预测值。
- $\alpha$ : 权重系数，满足  $0 \leq \alpha \leq 1$ 。
- 定义公式:  $\tau_{n+1} = \alpha \cdot t_n + (1 - \alpha) \cdot \tau_n$

**权重值的选择：**

- 通常， $\alpha$  设置为  $\frac{1}{2}$ 。

抢占版本的最短剩余时间优先调度算法

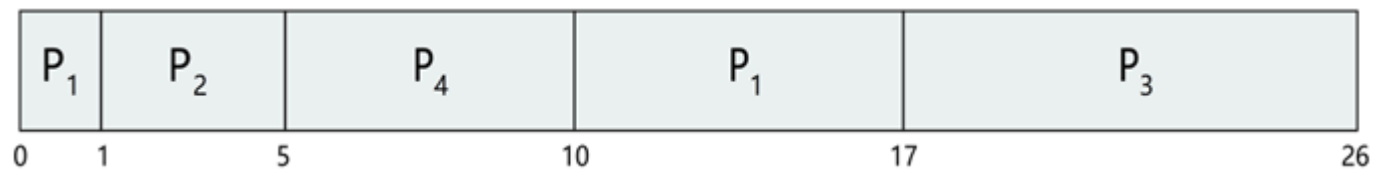
**抢占式版本被称为 Shortest-Remaining-Time-First (SRTF)。**

现在我们将引入**到达时间**和**抢占**的概念进行分析。

- 在进程执行时，如果新到达的进程的剩余执行时间更短，当前进程会被抢占，切换到新进程执行。
- 此算法动态调整调度顺序，优先处理剩余执行时间最短的进程。

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
$P_1$	0	8
$P_2$	1	4
$P_3$	2	9
$P_4$	3	5

抢占SJF调度的甘特图：



- Average waiting time抢占 =  $[(10-1)+(1-1)+(17-2)+ (5-3)]/4 = 26/4 = 6.5$  msec
- Average waiting time非抢占 =  $[0+(8-1)+(17-2)+ (12-3)]/4 = 31/4 = 7.75$  msec

调度过程解析：

- **0时刻**： $P_1$  到达，开始执行。
- **1时刻**： $P_2$  到达，剩余时间比  $P_1$  短，抢占  $P_1$ ，执行  $P_2$ 。
- **5时刻**： $P_4$  到达，剩余时间比  $P_3$  短，抢占  $P_2$ ，执行  $P_4$ 。
- **10时刻**： $P_1$  剩余时间最短，继续执行。
- **17时刻**： $P_3$  是最后的进程，执行完成。

时间片轮转 RR

- 每个进程获得一小段 CPU 时间（**时间量 quantum** 或 **时间片 time slice**），通常为 10-100 毫秒。当时间片耗尽后，进程会被抢占并添加到就绪队列的末尾。
- 定时器在每个时间量结束时触发中断，以调度下一个进程。
- **性能分析**：
  - 如果  $q$  较大，则表现为先到先服务（FIFO）。
  - 如果  $q$  较小，则  $q$  必须足够大以相较于上下文切换时间，否则开销过高。
  - 通常，时间片  $q$  为 10 毫秒到 100 毫秒，上下文切换时间小于 10 微秒 ( $\mu s$ )。

优先级调度

## 概念

- 每个进程关联有一个优先级数字（整数）。
- CPU 分配给优先级最高的进程（整数越小，优先级越高）：
  - **抢占式调度** (Preemptive)
  - **非抢占式调度** (Nonpreemptive)
- **SJF (Shortest Job First)** 是一种优先级调度，其中优先级是下一个 CPU 突发时间预测值的倒数。
- **问题：饥饿 (Starvation)** - 优先级低的进程可能永远得不到执行。
  - **解决方案：老化 (Aging)** - 随着时间的推移，逐步提高进程的优先级。

## 例子

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
$P_1$	10	3
$P_2$	1	1
$P_3$	2	4
$P_4$	1	5
$P_5$	5	2

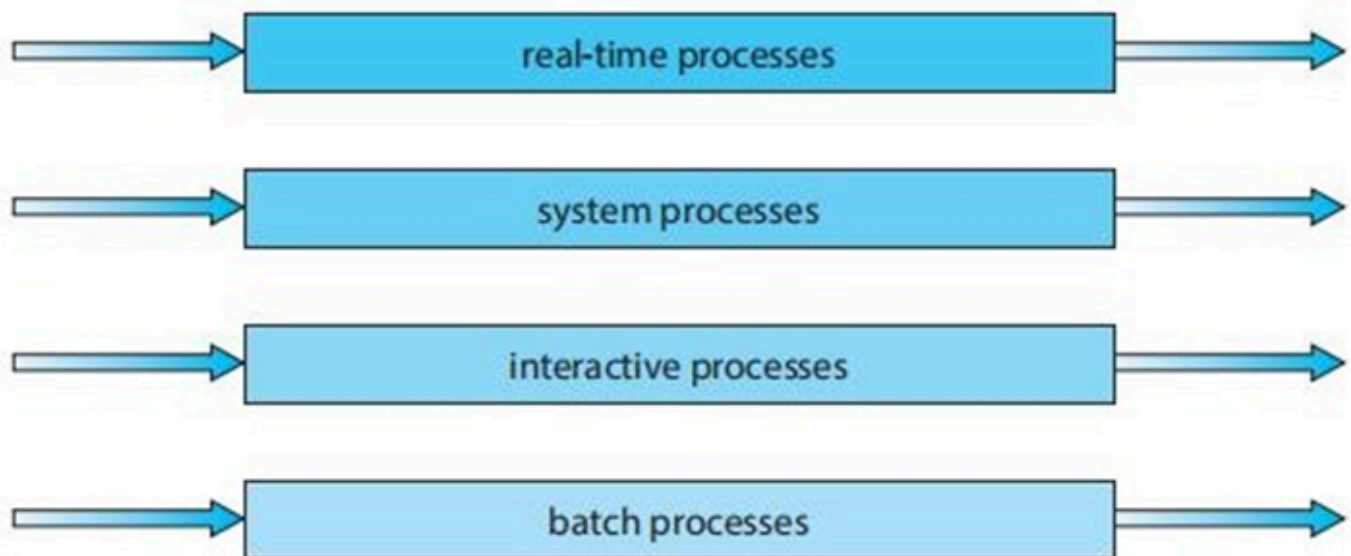
## Priority scheduling Gantt Chart



Average waiting time = 8.2 msec

## 多级队列

highest priority



lowest priority

- 就绪队列被划分为独立的多个队列，例如：
  - 前台 (foreground) : 交互式 (interactive)
  - 后台 (background) : 批处理 (batch)
- 进程永久属于某一个队列。
- 每个队列有其自己的调度算法：
  - 前台队列: RR (Round Robin, 轮转调度)
  - 后台队列: FCFS (First-Come-First-Served, 先来先服务)
- 队列之间的调度必须满足以下条件：
  - 固定优先级调度 (Fixed priority scheduling) : 例如, 先处理前台的所有进程, 再处理后台的进程。可能会导致 **饥饿问题**。
  - 时间片 (Time slice) : 每个队列分配一定的 CPU 时间。例如, 80% 时间分配给前台的 RR 调度, 20% 时间分配给后台的 FCFS 调度。

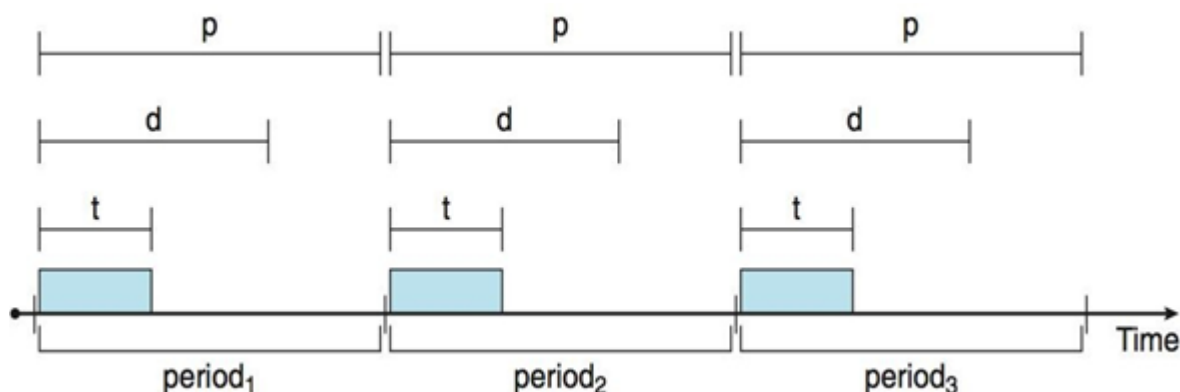
## 实时CPU调度(实时操作系统)

概念

- 实时CPU调度 (Real-Time CPU Scheduling)
  - 软实时系统 (Soft real-time systems)

- 软实时系统只提供优先级支持，没有明确保证关键实时进程会在特定时间内被调度。
- 适用于响应时间要求不严格的场景，例如视频流处理、音频播放等。
- **硬实时系统 (Hard real-time systems)**
  - 硬实时系统要求任务必须在其截止时间之前完成。
  - 常用于需要精确时间控制的场景，例如航空控制系统、工业自动化等。
- **两种延迟 (Latencies) 影响性能**
  1. **中断延迟 (Interrupt latency)**
    - 从中断到达服务中断的例程开始执行的时间。
  2. **调度延迟 (Dispatch latency)**
    - 将当前进程从CPU切换到另一个进程所需的时间。
- **调度延迟 (Dispatch latency) 的冲突阶段：**
  - **抢占 (Preemption)**
    - 系统必须能够强制抢占正在运行的低优先级进程，尤其是内核模式下的进程。
    - 抢占是确保高优先级实时任务能够尽快得到处理的关键机制。
  - **资源释放**
    - 当高优先级进程需要资源时，低优先级进程必须快速释放这些资源，以避免拖延实时任务的执行。

## 基于优先级的调度



- **实时调度要求调度程序支持抢占式和基于优先级的调度**
  - 对于 **软实时系统 (Soft real-time systems)**，只保证优先级，没有明确的完成时间要求。
- **对于硬实时系统 (Hard real-time systems)**
  - 必须具备满足**截止时间 (deadlines)** 的能力。

- 这是硬实时系统的核心要求，确保任务在规定的时间内完成，否则可能导致系统失败。

### • 进程具有新特性：周期性 (Periodic)

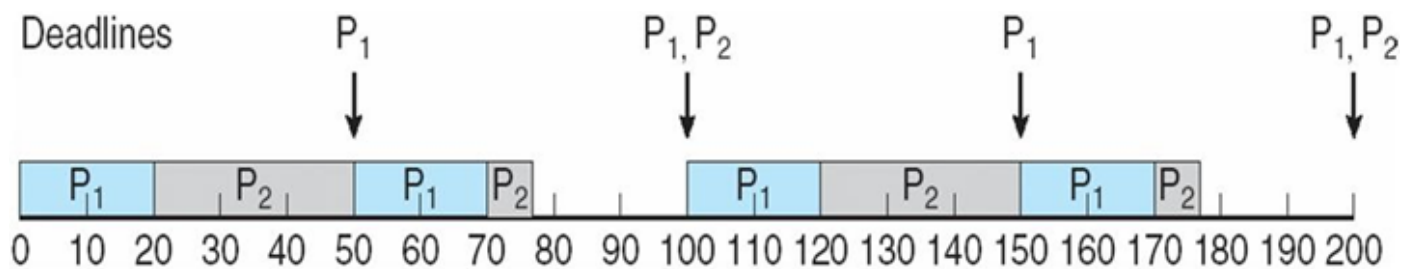
- 某些实时任务需要在固定时间间隔内使用CPU，形成周期性行为。
- 每个任务具有：
  - 处理时间 ( $t$ )：任务实际需要的CPU时间。
  - 截止时间 ( $d$ )：任务必须完成的时间点。
  - 周期 ( $p$ )：周期性任务的间隔时间。
- 周期性任务的执行频率： $1/p$

### 单调速率调度 RMS (抢占式)

#### 优先级分配原则：

- 优先级与任务周期成反比。
  - 周期短 → 优先级高
  - 周期长 → 优先级低
- 在图中，任务  $P_1$  的周期为 50，因此其优先级高于  $P_2$  (周期为 100)。

$p_1 = 50$  ,  $t_1=20$ ;  $p_2 = 100$  ,  $t_2=35$



- 高优先级任务  $P_1$  每个周期有 **20** 的执行时间，且优先级高，因此会抢占 CPU。
- 低优先级任务  $P_2$  每个周期有 **35** 的执行时间，但它只能在  $P_1$  执行完毕后利用剩余时间。

**Missed Deadlines:** 单调速率调度 被认为是 **静态优先级调度中最优的算法**，即如果一个进程集合无法通过该算法调度完成，那么任何其他基于静态优先级的算法也无法对其进行调度。

### ? 最早截止时间优先 EDF (抢占式)

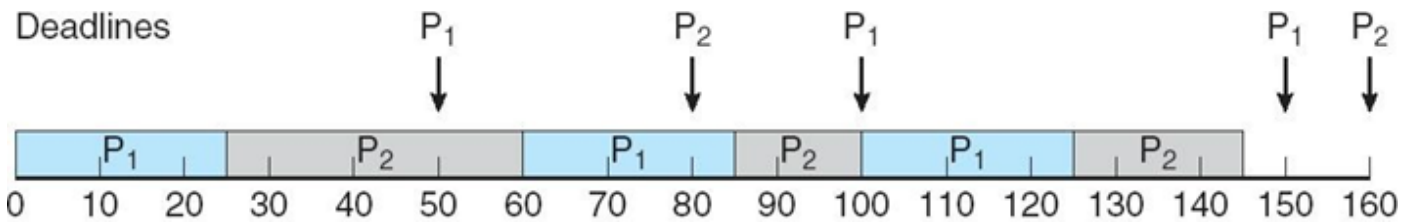
#### 最早截止时间优先 (EDF) \_edf算法-CSDN博客

优先级根据任务的截止时间动态分配：

- 截止时间越早，优先级越高
- 截止时间越晚，优先级越低

$$p_1 = 50, t_1 = 25$$

$$p_2 = 80, t_2 = 35$$



最早截止时间优先(EDF)是动态优先级

### 对图的解释：

进程 P1 的截止期限为最早，所以它的初始优先级比进程 P2 的要高。当 P1 的 CPU 执行结束时，进程 P2 开始运行。不过，虽然单调速率调度允许 P1 在时间 50（即下一周期开始之际）抢占 P2，但是 EDF 调度允许进程 P2 继续运行。进程 P2 的优先级比 P1 的更高，因为它的下一个截止期限（时间 80）比 P1 的（时间 100）要早。因此，P1 和 P2 都能满足它们的第一个截止期限。

进程 P1 在时间 60 再次开始运行，在时间 85 完成第二个 CPU 执行，也满足第二个截止期限（在时间 100）。这时，进程 P2 开始运行，只是在时间 100 被 P1 抢占。P2 之所以被 P1 抢占是因为 P1 的截止期限（时间 150）要比 P2 的（160）更早。在时间 125，P1 完成 CPU 执行，P2 恢复执行；在时间 145，P2 完成，并满足它的截止期限。然后，系统空闲直到时间 150；在时间 150 进程 P1 开始再次被调度。

与单调速率调度不一样，EDF 调度不要求进程应是周期的，也不要求进程的 CPU 执行的长度是固定的。唯一的要求是，进程在变成可运行时，应宣布它的截止期限。

## Deadlocks死锁

### 死锁的概念

#### 死锁概念

**死锁**是指一组进程在计算机系统中由于相互竞争资源而陷入一种无限等待的状态，每个进程都在等待其他进程释放它所需要的资源，但这些资源永远不会被释放，导致所有相关进程无法继续执行。



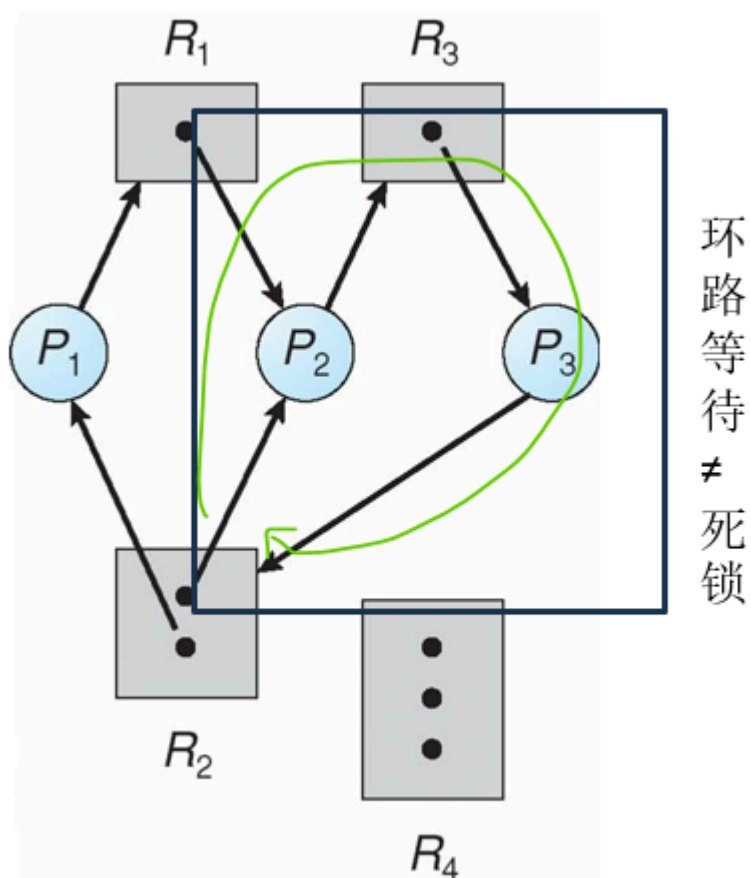
## 形成死锁四个必要条件

死锁发生的条件是以下四个条件 **同时满足**：

1. **互斥 (Mutual Exclusion)**: 在任何时间点, 仅有一个进程能够使用某个资源。
2. **占有与等待 (Hold and Wait)**: 一个进程已经占有至少一个资源, 并正在等待获取其他由其他进程占有的资源。
3. **非抢占 (No Preemption)**: 资源只能由占有它的进程 **自愿释放**, 即在该进程完成任务后释放资源。
4. **循环等待 (Circular Wait)**: 存在一个进程集合  $P_0, P_1, \dots, P_n$ , 使得:
  - $P_0$  正在等待被  $P_1$  占有的资源,
  - $P_1$  正在等待被  $P_2$  占有的资源,
  - .....,
  - $P_n$  正在等待被  $P_0$  占有的资源。

若这四个条件全部成立, 则系统可能陷入死锁。

## 资源占用分配图



# 死锁处理策略

## 死锁预防

限制请求方式：

- **互斥 (Mutual Exclusion) :**

- 对于可共享资源（如只读文件），不需要互斥；但对于不可共享资源，必须保持互斥。
- 某些资源在同一时间只能被一个进程访问或使用

- **占有并等待 (Hold and Wait) :**

- 必须保证当一个进程请求资源时，它不持有其他任何资源。
  - 要求进程在开始执行前请求并分配所有资源，或者仅允许在没有分配到任何资源时请求资源。
  - 缺点：资源利用率低，可能导致饥饿。

- **无抢占 (No Preemption) :**

- 如果一个持有某些资源的进程请求另一个无法立即分配的资源，那么它当前持有的所有资源将被释放。
- 被抢占的资源会加入该进程等待的资源列表中。
- 进程只有在能够重新获得其旧资源以及所请求的新资源时才会重新启动。

- **循环等待 (Circular Wait) :**

- 对所有资源类型强制施加一个总的顺序，并要求每个进程按照递增顺序请求资源。

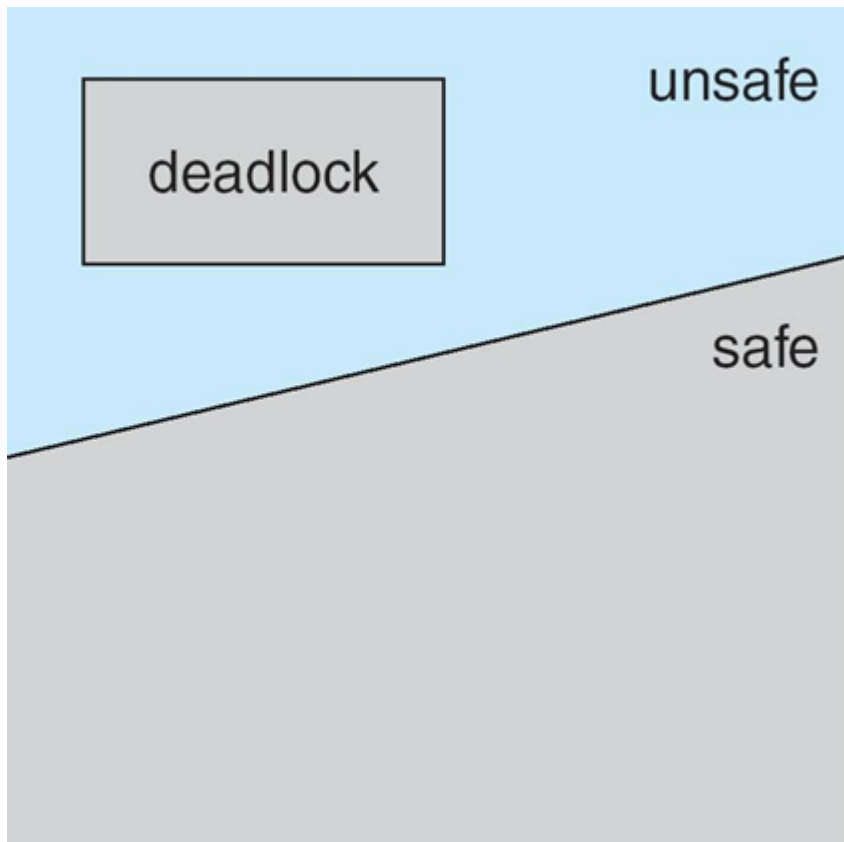
## 死锁避免

- 要求系统有一些**先验信息**可用。
- 最简单和最有用的模型是每个进程声明它可能需要的 **每种资源的最大数量**。
- 死锁避免算法动态检查资源分配状态，以确保 **永远不会存在循环等待条件**。
- **资源分配状态**由以下内容定义：
  1. 可用资源的数量。
  2. 已分配资源的数量。

### 3. 各进程的最大资源需求。

#### 系统安全状态

即能够找到一个序列使每个进程都可以执行完成。

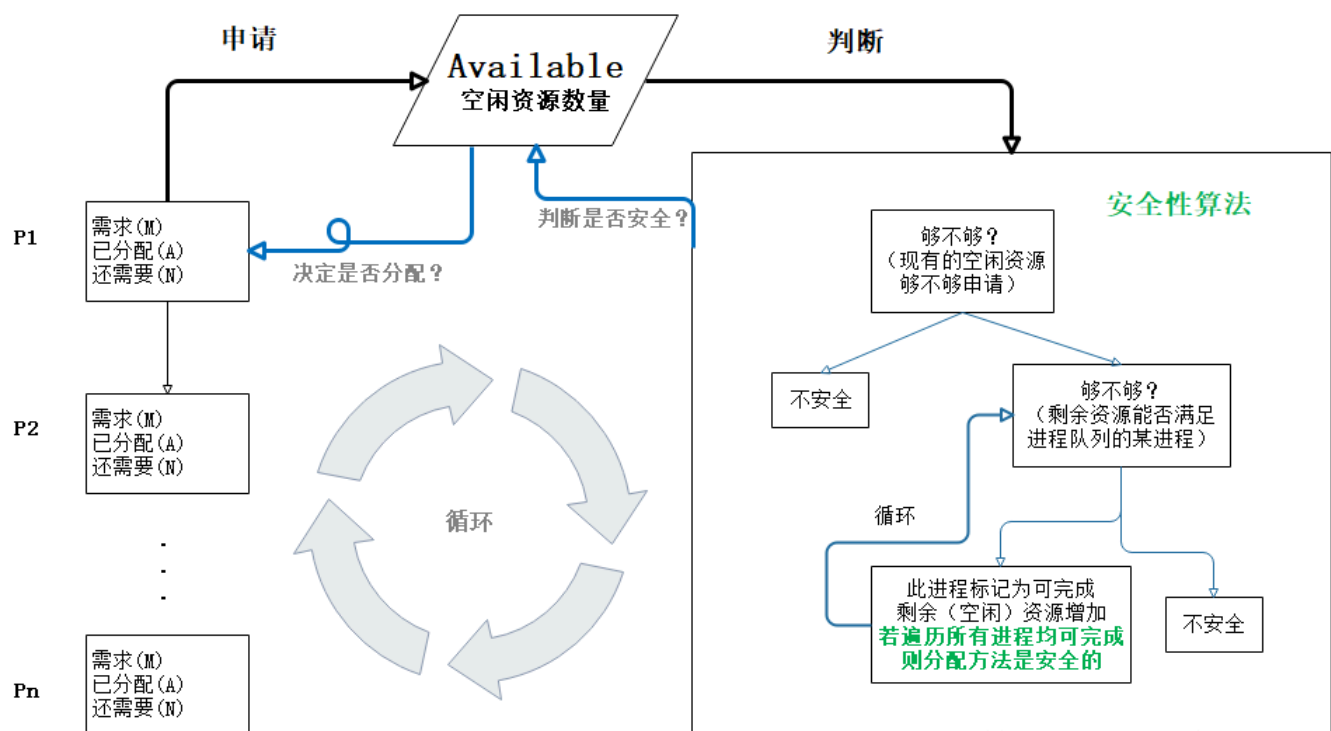


- 当一个进程请求可用资源时，系统必须决定立即分配是否会让系统进入一个安全状态。
- 如果系统中存在一个所有进程可以完成其任务的序列  $\langle P_1, P_2, \dots, P_n \rangle$ ，则系统处于 **安全状态**。
- **系统处于安全状态时**：无死锁。
- **系统处于不安全状态时**：可能会发生死锁。
- **避免死锁 (Avoidance)**：确保系统永远不会进入不安全状态。

#### 银行家算法

[一句话+一张图说清楚——银行家算法-CSDN博客](#)

#### 过程



[https://blog.csdn.net/qq\\_33414271](https://blog.csdn.net/qq_33414271)

## 例子

题中共四种资源，P0的Allocation为 (0, 0, 3, 2) 表示已分配给P0的第一种资源和第二种资源为0个，第三种资源3个，第四种资源2个。

Process	Allocation	Need	Available
P0	0 0 3 2	0 0 1 2	1 6 2 2
P1	1 0 0 0	1 7 5 0	
P2	1 3 5 4	2 3 5 6	
P3	0 3 3 2	0 6 5 2	
P4	0 0 1 4	0 6 5 6	

(1) 该状态是否安全？ (2) 若进程P2提出请求Request (1, 2, 2, 2) 后，系统能否将资源分配给它？

## 解答

(1) 利用安全性算法对上面的状态进行分析（见下表），找到了一个安全序列 {P0,P3,P4,P1,P2}，故系统是安全的。

	Work	Need	Allocation	Work+Allocation	Finish
P0	1 6 2 2	0 0 1 2	0 0 3 2	1 6 5 4	true
P3	1 6 5 4	0 6 5 2	0 3 3 2	1 9 8 6	true
P4	1 9 8 6	0 6 5 6	0 0 1 4	1 9 9 10	true
P1	1 9 9 10	1 7 5 0	1 0 0 0	2 9 9 10	true
P2	2 9 9 10	2 3 5 6	1 3 5 4	3 12 14 14	true

(2) P2发出请求向量Request(1,2,2,2),系统按银行家算法进行检查:

①Request2(1,2,2,2) ≤ Need2(2,3,5,6) ②Request2(1,2,2,2) ≤ Available(1,6,2,2) ③

系统先假定可为P2分配资源,并修改Available, Allocation2和Need2向量:

Available=(0,4,0,0) Allocation2=(2,5,7,6) Need2=(1,1,3,4) 此时再进行安全性检查,发现 Available=(0,4,0,0) 不能满足任何一个进程,所以判定系统进入不安全状态,即不能分配给P2相应的Request(1,2,2,2)。

## 其他

## 死锁检测算法总结

### 1. 算法描述

- 死锁检测算法用于判断系统中是否存在死锁。
- 算法通过检查资源分配状态,找到未完成任务(即未能释放资源)的进程集合。

### 2. 步骤概述

- 初始化 **Work**和 **Finish**两个向量,分别表示当前可用资源和进程完成状态。
- 根据资源分配和请求,动态检查是否可以满足某些进程的资源需求,从而推进进程执行。
- 如果所有进程都能完成,则无死锁;否则存在死锁。

### 3. 示例分析

- 根据资源快照计算资源分配情况和是否满足安全序列。
- 如果一个进程资源请求无法被满足且所有可用资源耗尽,则系统进入死锁状态。

### 4. 使用条件

- 需系统提供资源分配、请求等信息以运行检测算法。
- 根据死锁可能性和系统性能,决定何时以及多频繁地运行检测算法。

### 5. 限制

- 算法复杂度为  $O(m \times n^2)$ ，其中  $m$  是资源种类， $n$  是进程数量。
- 若检测算法过于频繁执行，会增加系统开销。

## 优化策略

- 仅在资源不足或频繁分配失败时运行检测算法。
- 对于大规模系统，需结合日志分析等手段优化检测效率。

# 死锁恢复

## 进程终止

1. **终止所有发生死锁的进程**
  - 一次性终止所有受影响的进程。
2. **逐个终止进程，直到死锁循环结束**
  - 根据策略选择终止的顺序，避免无谓损失

## 资源抢占

1. **选择牺牲者 (Selecting a victim)**
  - 目标是最小化代价，选择适合释放资源的进程。
2. **回滚 (Rollback)**
  - 将系统返回到某个安全状态，并从该状态重新启动相关进程。
3. **饥饿 (Starvation)**
  - 避免某个进程反复被选为牺牲者，可以将回滚次数纳入选择的成本因素中。