OS 笔记

卢雨轩 19071125

2021 年 9 月 23 日

目录

第一部分　操作系统简介

1.1　什么是操作系统

系统观点 操作系统作为资源管理器。记录、协调各个程序对资源的请求。

- 硬件资源：处理器资源、内存、IO 设备…
- 信息资源：文件管理、锁…

用户观点 作为机器的拓展。

1.2　操作系统的历史

- 无操作系统
- 单道批处理系统
- 多道批处理系统
- 分时系统、抢占式调度
- 现代操作系统

1.3　操作系统的基本特征

- 并发 Concurrence
- 共享 Sharing
- 虚拟 Virtual
- 异步性 Asynchronism

1.4　操作系统的结构

- *整体式结构*如 MS-DOS，Unix。是一系列过程的集合，可以互相调用。
- *层次式结构*层次式系统的各种功能可以划分为几个层次，每个层次建立在下面的层次之上。
  优点：模块化
  缺点：对层的定义；相对效率差
  例子：OS/2
- *微内核结构*把部分属于操作系统的功能放到内核的外面，使内核更小，称为微内核。
  - 操作系统微内核之外的进程都是服务进程
  - 用户进程是客户进程
  - 微内核中只提供进程管理、内存管理和通讯功能
  - 系统效率较低（信息传递性能损耗）优点：
  - 易于维护
  - 易于扩充

1.5　操作系统的运行环境

1.5.1　程序状态字

程序状态字（Program State Word, PSW）处于 CPU，用于包含状态信息。

- Flags（OF，CR，ZERO，…）
- *指令优先级*
- *模式（用户、内核）*
- 其他控制位

1.5.2　双重模式

- 监督程序模式（Monitor mode, M Mode）：执行 OS 任务
  - Kernel / System / Privileged/ Supervisor mode
  - 内核模式、系统模式、特权模式、管态
- 用户模式（User mode, U mode）：执行用户程序
  - 目态
- 区分两种模式的原因：
  - 提供保护操作系统和其他用户程序的首段
  - 特权指令：可以引起损害的指令

1.5.3　中断

- *现代操作系统是中断驱动的*
- 定义：由外部事件引起的暂停过程。
- 中断与陷阱：
  - 中断（Interrupt）指硬中断，如外设、事件
  - 陷入（Trap）指软中断

第二部分　进程管理

2.1　进程的概念

2.1.1　什么是进程

进程（Process）是一个正在执行的程序，除了程序代码段等之外包括堆栈段。
*可重入*指能被多个程序同时调用的程序。纯代码，执行过程中不会改变。

2.1.2　进程的状态及其转换

- *两状态进程模型*
  - 进程要么正在被处理器执行，要么没有被处理器执行。
  - 只有两种状态：
    - 运行状态
    - 非运行状态
  - 无法区分等待与就绪

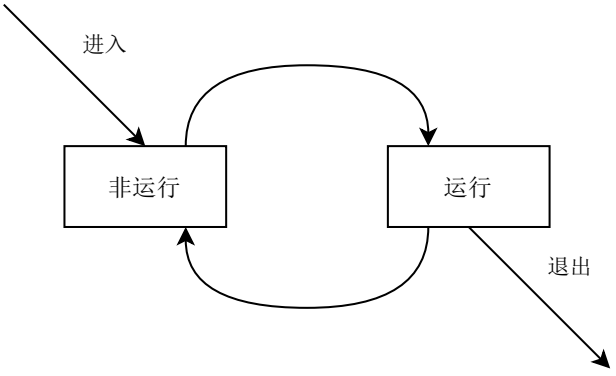

图 1: 两状态进程模型

- 三状态进程模型
  - 三种状态
    - 就绪（Ready）
    - 运行（Running）
    - 等待（Waiting）

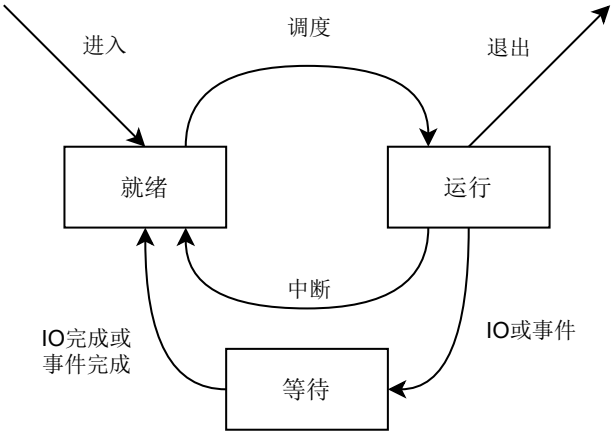

图 2: 三状态进程模型

- 五状态进程模型
  - 五种状态
    - 新（New）：进程正在被创建
    - 就绪（Ready）
    - 运行（Running）
    - 等待（Waiting）
    - 终止（Stopped）：进程已经停止
  - 缺点：如果所有进程都在等待，CPU 利用率低

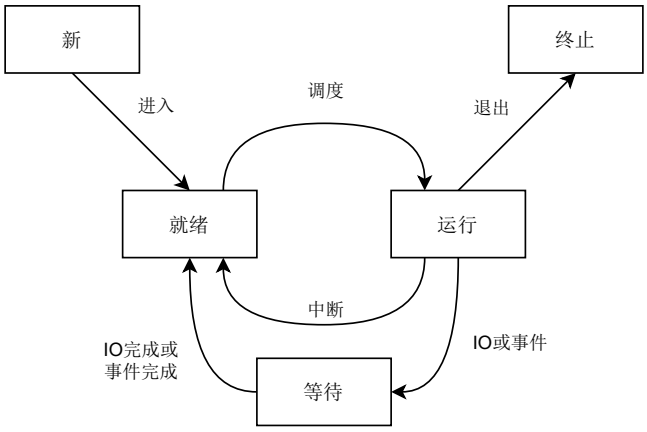

图 3: 五状态进程模型

- 七状态进程模型
  - 七种状态
    - 新（New）：进程正在被创建
    - 就绪（Ready）
    - 运行（Running）
    - 等待（Waiting）
    - 终止（Stopped）：进程已经停止
    - 就绪挂起：进程在外存中，等待的事情已经发生
    - 等待挂起：进程在外存中，等待的事情还未发生
  - 提高 CPU、内存利用率
  - 挂起等待中的进程

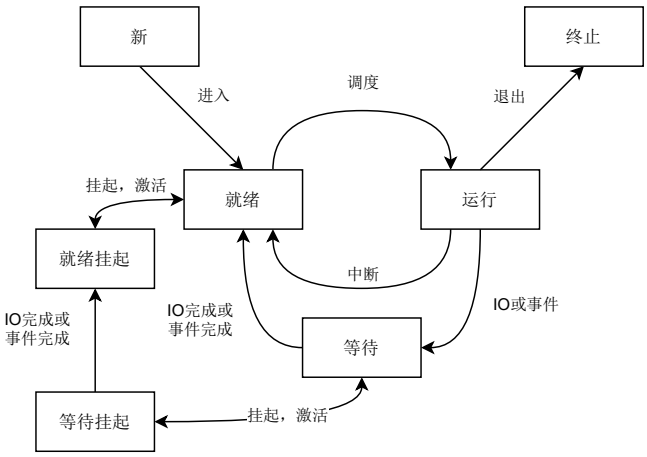

图 4: 七状态进程模型

2.1.3　进程的实体与特征

- 进程实体
  - 程序代码

  - 当前的活动
  - 数据段（data, bss, etc.）
  - 栈
  - 堆
- **进程映像**：进程在内存中的组成
  - 进程控制块（PCB, Process Control Block）
  - 程序
  - 数据
  - 堆栈
- **进程控制块的内容**
  - ID
    - 进程 ID
    - 父进程 ID
    - 用户 ID
  - 当前状态
    - 寄存器
    - 栈指针
  - 常规信息（调度信息，IPC 信息，FD，…）

2.1.4　进程的调度

- **长期调度**
  - 从进程池中选择进程进入内存
    - 控制内存中进程的数量
  - 搭配选择 I/O bound 和 CPU bound 程序
  - 频率低（几分钟一次），有些系统不用
- **中期调度**
  - 从换出到外存中挂起的进程选择进程进入内存
- **短期调度**
  - 从就绪队列中选择进程到 CPU 上执行
  - 频繁（100ms）

2.1.5　进程的特点

- **动态性**：状态在变化
- **并发性**：多个进程可以同时运行
- **独立性**：是独立运行的基本单位
- **异步性**：可以独立的、以不可知的速度运行

2.2　进程的控制

2.2.1　进程的创建

- 进程通过系统调用创建进程，前者称为父进程，后者称为子进程
  - 进程树
- 系统调用:
  - fork
    - 共享地址空间，Copy On Write
  - execve
    - 替代地址空间
  - spawn
  - CreateProcess

2.2.2　进程终止

- 进程终止自己 exec 系统调用
  - 父进程使用 wait
- 父进程终止子进程
  - SIGTERM, SIGKILL
  - TerminateProcess
- 操作系统终止进程

2.2.3　进程的阻塞与唤醒

- 阻塞操作阻塞的系统调用（调用资源）
- 进程唤醒等待的事件到来

2.3　进程间通信（IPC）

- 管道通讯
  - 管道是一个环形缓冲区
  - Producer - Consumer Model
- 共享内存
  - 最快
  - 共享一个内存块
- 消息传递
  - 直接通讯：给指定进程发送消息
  - 间接通讯:『邮箱』
- 同步性问题：同步、异步

2.4　线程

- 什么是线程
  - 线程是调度的单位
  - 进程是资源的拥有者
  - 轻量级线程，是进程内部的一条运行线
  - 共享地址空间，拥有线程 ID、PC、寄存器和堆栈
- 线程的优点
  - 响应度高：不会阻塞整个进程
  - 资源共享
  - 通信简单
  - 经济
- 线程的分类
  - 用户级线程
  - 内核级线程
  - 混和

2.5　进程的同步

2.5.1　基础理论

- 进程之间的关系
  - 独立的多个进程异步执行，仿佛没有关系
    - ~~独立进程不影响其他进程，也不被其他进程影响?~~
    - 资源有限!
  - 协作的多个进程需要同步
    - 进程之间可以相互影响
    - 原因：信息共享；加跨计算；模块化；方便
    - 例子：Producer-Consumer Model
- 进程同步问题的提出

  打印机问题。
- 竞争条件

  这种两个以上的进程共享数据，而最终的执行结果是根据执行次序决定的，称为竞争条件 *(Data Race)*。

  如何解决 Data Race? 控制对资源的访问。
- 临界资源和临界区

  为了避免竞争条件，必须找到一种方法来阻止多个进程同时读写共享的数据。
  - 这种共享的数据称为临界资源 *(Critical Resource)*
  - 程序中访问临界资源的部分称为临界区 *(Critical Section)*

  - *互斥 (mutual exclusion)*:
    * 如果有进程在临界区中执行，那么其他进程都不能在临界区中执行。
    * 可以避免 data race 的产生。
  - 有空让进
  - 有限等待
  - 不对进程的相对执行速度进行任何假设

- 如何解决临界区互斥的问题?
  - 软件的解决方案（Raft 等一致性算法?）
  - 硬件的解决方案
  - 信号量的解决方案
  - 管程的解决方案

2.5.2　硬件方法

- 硬件的解决方案之一：关中断

  进入临界区之前关中断，离开之后开中断
  - 线代操作系统是中断驱动的，没有了中断操作系统就失去了控制系统的能力
  - 只有一个 CPU 时有效

- 硬件的解决方案之二：原子指令

  系统提供了特殊的硬件指令，原子的检查和修改字的内容或者交换两个字。
  - TestAndSet
    IBM370 中称为 TS 指令
  - Swap
    在 Intel8086 中称为 XCHG 指令

- *TestAndSet*

```
bool TestAndSet(bool *target){
    bool value = *target;
    *target = True;
    return value;
}
.....
bool lock;
do {
    // Try to acquire the lock.
    // If can't, busy spin.
    while(TestAndSet(&lock));
    // Lock acquired.
    // Do CRITICAL STUFF.
    lock = False;
    // Release lock for other process.
    // Do REMAINING STUFF.
}
```

- *Swap*

```
void Swap(bool *a, bool *b){
    bool value = *a;
    *a = *b;
    *b = value;
}
.....
bool lock;
do {
    bool key = True;
    // Try to acquire the lock.
    // If can't, busy spin.
    while(Swap(&lock, &key));
    // Lock acquired.
    // Do CRITICAL STUFF.
    lock = False;
    // Release lock for other process.
    // Do REMAINING STUFF.
}
```

- 互斥锁（*Mutex Locks*）
  - 底层用 TestAndSet 或者 Swap 实现
  - acquire()
  - release()
- 软件和硬件解决方案的问题：
  - 忙等待（busy waiting）浪费 CPU
    - 解决方案：让出 CPU
  - 优先级反转
    - 解决方案：优先级捐献
- 信号量方法
  - 两个或多个进程可以用信号进行协作
  - 进程可以在任何地方停下来等信号
  - 信号通过 Semaphore 特殊变量传递信号
  - Semaphore 操作：
    - 有一个整形
    - wait（P）用来接受信号
    - signal（V）用来发送信号
    - 初值设置为资源数量
  - 错误使用：
    - 死锁
    - 饥饿

- 生产者消费者问题
  - 生产者将任务放入缓冲区
  - 消费者从缓冲区取出任务
  - 缓冲区要互斥
    - 否则会出错
  - 生产者消费者问题的解决

```
semaphore n=0; // 防止 consumer 消费空队列
semaphore s=1; // 保证 put 和 take 操作原子性
semaphore e=N; // 防止 producer 写入满队列
void producer(){
    while(true){
        a=produce();
        wait(e); // 等待“队列中剩余空位数量”
        wait(s); // 获取队列锁
        put(a);
        signal(s); // 释放队列锁
        signal(n); // 增加“等待处理的任务数量”
    }
}
void consumer(){
    while(true){
        wait(n); // 等待“等待处理的任务数量”
        wait(s); // 获取队列锁
        a=take();
        signal(s); // 释放队列锁
        signal(e); // 增加“队列中剩余空位数量”
        consume(a);
    }
}
```

- 哲学家进餐问题
  - 5 个哲学家住在一起，每个人的生活由吃饭和思考组成
  - 桌子上有一盘菜，每人一个盘子一支叉子
  - 想吃饭的哲学家会走到桌子变的位置，拿起左右的叉子，从中间的盘子中去菜放到自己的盘子中
  - 要求：
    - 保证叉子互斥
    - 防止死锁和饥饿