进程与进程调度

进程

- 进程与线程的区别：
  - 资源分配：进程是操作系统分配资源的基本单位，每个进程都有独立的内存空间、文件描述符、执行上下文等。而**线程是在进程内部创建的执行单元，多个线程共享同一进程的资源。**
  - 执行方式：进程是独立执行的实体，具有独立的程序计数器（`Program Counter`）、寄存器集合和堆栈空间，可以并行执行。而线程是进程内的一个执行路径，由线程调度器负责调度，线程在进程内部共享进程的资源，**并且线程的执行是并发的，可以利用多核处理器的并行性。**
  - 轻量级：线程相对于进程来说更加轻量级，创建、销毁和切换线程的开销远小于进程。因此，在需要频繁创建、销毁和切换执行单元的场景下，使用线程可以更高效地利用系统资源。
  - 通信与同步：进程间通信（`IPC`）需要使用特定的机制，如管道、信号量、消息队列等。而线程之间共享同一进程的内存空间，因此可以直接读写共享的数据，也可以使用线程同步机制（如互斥锁、条件变量）来实现线程间的通信和同步。
  - 可靠性：一个进程的崩溃不会影响其他进程的执行，每个进程都有独立的地址空间。而线程是在同一个进程内执行的，一个线程的错误可能会导致整个进程崩溃。
- 管程（`Monitor`）：管程是一种并发编程的概念，用于协调多个线程对共享资源的访问。它提供了一种结构化的方法来同步和互斥访问共享数据。
- 用户态和内核态的区别：
  - 用户态（`User Mode`）：在用户态下运行的程序只能访问受限的资源，如自身的内存空间和部分操作系统提供的服务。用户态程序无法直接访问操作系统内核的功能和敏感资源。这是为了确保系统的安全性和稳定性。
  - 内核态（`Kernel Mode`）：内核态是操作系统内核执行的特权级别。在内核态下，程序可以访问和控制系统的所有资源和功能，包括底层硬件设备、系统内存和其他进程。内核态程序拥有更高的特权级别，可以执行特权指令和访问受保护的系统数据结构。
- 并行（parallel）和并发（concurrent）的区别：

  "parallel"强调任务的同时执行，而"concurrent"强调任务的重叠执行。并行执行通常需要硬件支持，而并发执行可以通过合适的软件机制实现。

进程调度

- 常用的进程调度算法
  - 先进先出调度算法（`FCFS`，*先到先服务*）：按照进程到达的顺序进行调度，即先到先执行的原则。适用于长作业时间的进程，但**可能导致短进程等待时间过长。**
  - 短作业优先调度算法（`SJF，Shortest Job First`，*谁快谁先来*）：选择预计执行时间最短的进程进行调度。可以**最大程度地减少平均等待时间，但可能出现饥饿现象。**

    > 最短剩余时间优先：（`SRTF`）：抢占式`SJF`算法。
    >
    > `SJF`可以证明其**平均等待时间最短，算法最优。**

  - 优先级调度算法（`PF,Priority Scheduling`，*谁吊谁先来*）：为每个进程分配一个优先级，根据优先级来决定调度顺序。可以根据进程的特性或重要性进行优先级的划分，但**可能导致低优先级进程永远等待高优先级进程。**

> 最高优先级优先分配，同一优先级按 `FCFS` 顺序调度。
>
> 本书以低数字代表高优先级。
>
> 优先级调度也有抢占式和非抢占式的区别。

- 时间片轮转调度算法（`RR, Round Robin`，*体验时间有限，体验完要开VIP（`bushi`）*）：为每个进程分配一个固定的时间片，每个进程按照时间片的顺序轮流执行，如果时间片用完，则被放到队列的末尾等待下一次调度。适用于**时间要求相对均匀**的场景，但**可能存在上下文切换开销过大的问题。**

  > 一般平均等待时间较长。
  >
  > 时间片太大：退化为 `FCFS` 算法，时间片太小：上下文切换成本过高。80%的CPU执行应该小于时间片。

- 多级反馈队列调度算法（`Multilevel Feedback Queue`，*上面两个的合体*）：将进程分为多个队列，每个队列具有不同的优先级和时间片大小，根据进程的行为动态调整优先级。适用于多种类型的进程，可以兼顾长短作业的特点。

  > 需要很多参数，是最复杂的算法。

> 如何解决优先级调度低优先级可能永远等待的潜在问题？
>
> `Aging` 机制：在每个时间片或特定时间间隔后，检查所有低优先级进程的等待时间。对于等待时间达到时间阈值的进程，增加其优先级。

- 进程调度中常用的参量（时间一般取平均值）
  - CPU使用率
  - 吞吐量：单位时间内完成的进程数量
  - 周转时间 = 进程完成的时间点 - 进程提交的时间点
  - 等待时间 = 进程在等待就绪队列中所花时间之和
- 进程调度的状态机，书本 `P73`

线程

CPU使用的基本单元，同一进程内所有线程共享代码段，数据段和其它系统资源（文件，信号等）

- 用户线程：第五个 `PPT` `P33`，知道概念和优缺点
- 内核线程：`P35`，更好地支持多CPU体系结构

> 并发性：某个线程地阻塞不一定导致进程地阻塞

用户线程和内核线程地对应关系形成了三种多线程模型：多对一、一对一、一对多。

同步和死锁

同步

- 并发（`Concurrent`）：对共享数据的并发访问会导致数据的不一致性。

  竞争有条件：多个进程并发访问操作同一数据并且执行结果与特定访问顺序无关，为了确保一次仅有一个进程可以操作同一变量，需要**进程同步**和**进程协调**。

- 临界区

  临界区问题：n个进程竞争使用一些共享的数据。

  临界资源：一次只允许一个进程使用或访问的资源。

  临界区问题的解决方案需要满足以下要求：互斥，进步，有限等待。

- 经典的同步问题：

  - 生产者-消费者问题（实际使用了**记录型信号量**）

    使用缓冲区机制，使用两个信号指示缓冲区是否满或者空。生产者在缓冲区满时必须等待，直到缓冲区有空间才继续生产；消费者在缓冲区空时必须等待，直到缓冲区中有产品才能继续读取。

  - 哲学家就餐问题

    解释死锁问题，如果同时拿起左边的筷子，会发生死锁。
    解决方法：

    - 最多允许4个哲学家坐在桌子上
    - 两个筷子都能用时才能拿起（AND型信号量）
    - 非对称解决：奇数哲学家先拿左边后拿右边，偶数哲学家先拿右边后拿左边

    没有死锁不保证没有饿死

  - 读者写者问题

    允许多位读者，但同一时间只允许一位写者，并且执行写操作前应该让已有的读者和写者全部退出。

- 信号量与管程

  两个关键函数：`wait()` 和 `signal()`。

  - 整形信号量：用一个整形变量记录信号量

    影响效率，资源利用率不高。

  - 记录型信号量：为了解决“忙等”的问题，除了需要一个用于代表资源数目的整型变量 value 外，又增加一个进程链表 L，用于链接上述的所有等待进程。

    > 如果 `S.value` 的初值为1，表示只允许一个进程访问临界资源，此时的信号量转化为互斥信号量

  - AND型信号量：将进程在整个运行过程中需要的所有资源，一次性全部地分配给进程，待进程使用完后
    再一起释放。只要尚有一个资源未能分配给进程，其它所有可能为之分配的资源，也不分配给他。亦
    即，对若干个临界资源的分配，采取原子操作方式：要么全部分配到进程，要么一个也不分配。

  管程的引入：

  信号量 `PV` 操作的缺点：

- 易读性差：因为要了解对于一组共享变量及信号量的操作是否正确，则必须通读整个系统或者并发
  程序。
- 不利于修改和维护：因为程序的**局部性很差**，所以任一组变量或一段代码的修改都可能影响全局。
- 正确性难以保证：因为操作系统或并发程序通常很大，而 `PV` 操作代码都是由用户编写的，系统无
  法有效地控制和管理这些 `PV` 操作，**要保证这样一个复杂的系统没有逻辑错误是很难的**，它将导致死锁
  现象的产生。

管程的特征：

- 模块化：一个管程是一个基本程序单位，可以**单独编译**；
- 抽象数据类型：**管程是一种特殊的数据类型**，其中不仅有数据，而且有对数据进行操作的代码；
- 信息掩蔽：管程是半透明的，管程中的外部过程（函数）实现了某些功能，而这些功能是怎样实
  现的，在其外部则是不可见的。

管程与进程的异同

- 管程定义的是**公用数据结构**（成员变量被声明为 `public` ），而进程定义的是**私有数据结构**（成员变量被声明为 `private` ）；
- 管程把共享变量上的同步操作集中起来，而临界区却分散在每个进程中；
- 管程是为管理共享资源而建立的，进程主要是为占有系统资源和实现系统并发性而引入的；

条件变量（condition）与 `PV` 操作中信号量的区别：使用 `signal` 释放等待进程时，可能出现两个进程同时停留在管程内。解决方法：

1. 执行 signal 的进程等待，直到被释放进程退出管程或等待另一个条件。
2. 被释放进程等待，直到执行 signal 的进程退出管程或等待另一个条件。
3. 霍尔采用了第一种办法，汉森选择了两者的折衷，规定管程中的过程所执行的 signal 操作是过程
   体的最后一个操作。

死锁

- **死锁的四个必要条件**
  - 互斥条件(Mutual exclusion)：进程互斥使用资源。至少有一个资源处于非共享模式。
  - 持有且等待(Hold and wait)：一个进程应占有至少一个资源，并等待另一个资源，而该资源为其他进程所占用，申请新资源时不释放已占有资源。
  - 不剥夺条件(No preemption)：一个进程不能抢夺其他进程占有的资源。只能在进程完成任务后资源**自愿**释放。
  - 环路条件(Circular wait)：存在一组进程循环等待资源。
- 资源分配图
  - 如果每个资源类型刚好有一个实例，那么存在环是死锁存在的充分且必要条件
  - 如果每个资源类型有多个实例，那么存在环不意味着存在死锁
- 死锁**处理方法**
  - 通过协议来预防或避免死锁，确保系统不进入死锁
  - 允许死锁，检测并加以修复
  - 忽视这个问题（Linux和Windows均在采用），需要程序员自己编写程序避免死锁。
- 死锁**预防方法**

- 不满足互斥条件：通常**互斥条件必须成立**，至少有一个资源是非共享的。
- 不满足持有且等待条件：
  - 一个进程申请之前获得所有资源
  - 一个进程在申请其他资源之前应该释放现在已分配的所有资源（仅在没有资源的时候才可以申请资源）

    > 缺点：资源使用率可能较低，可能发生饥饿
- 不满足无抢占条件：
  一个进程持有资源并申请另一个不能立即分配的资源，那么现在所分配的资源都可被抢占。

  当一个进程处于等待时，如果其他进程**申请其拥有资源，那么该进程的部分资源可以被抢占。**

  > 一般用于状态可以保存和恢复的资源，如CPU寄存器和内存。一般不适用于互斥锁和信号量
- 不满足循环等待条件：

  对所有资源类型进行完全排序，要求每一个进程按递增顺序来申请资源。

  函数F应当根据系统内资源使用的正常顺序来定义。

- 安全序列的计算

  不是所有的非安全状态都能导致死锁
  - 资源分配图算法
  - **银行家算法**

    Allocation

    MAX

    Need

    Available（独立的，是总资源数 - 当前已分配的资源数总和）

内存管理策略

内存管理策略

- 分段（`Segmentation`）
  - 内存被划分为若干段，每一段表示一个逻辑上独立的程序模块或数据结构。
  - **段的大小可以不同**，每个段都有自己的起始地址和长度。
  - 分段存储适用于具有**不同大小和可变大小的程序和数据**，如编译器、操作系统和数据库系统。
  - 分段存储可以提供**更好的逻辑地址空间管理和保护**，允许程序的各个部分被独立地加载和卸载。
  - 分段存储可能会导致**外部碎片**，即空闲内存块不连续，需要使用一定的分配策略来解决。
- 分页（`Paging`）
  - 内存被划分为**固定大小**的页面（页），每个页面的大小相同。
  - 程序和数据被划分为相应大小的页，存储时按页为单位进行管理。

- 分页存储适用于**统一大小的程序和数据**，如操作系统和虚拟内存系统。
- 分页存储可以提供**更好的内存利用率**，减少内部碎片，且页面可以随意调度和置换。
- 分页存储可能会导致**内部碎片**，即每个页面中未被完全利用的部分。

> 逻辑内存的页大小和物理内存的帧大小是否一样？（书上说的一样，但是不一定）
>
> 在分页存储中，逻辑内存被划分为固定大小的页面（页），而物理内存被划分为相同大小的帧（frame）。每个页的大小和每个帧的大小是相同的。在这种情况下，逻辑内存的页大小和物理内存的帧大小是一样的，这样可以简化内存管理，让页面和帧之间一一对应。在逻辑地址到物理地址的转换过程中，只需简单地通过页号找到对应的帧号，再加上页内偏移量即可得到物理地址。
>
> 然而，在某些系统中，为了更好地灵活使用内存，逻辑内存的页大小和物理内存的帧大小可以是不同的。这种情况下，逻辑页和物理帧之间的映射关系可能不再是一对一的关系，而是通过页表来管理。

- `TLB` 以及有效内存访问时间

  类似 `Cache`，有 `TLB hit` 和 `TLB miss` 两种情况，如果 `TLB miss` 则会导致一些较为严重的后果，需要访问内存页表，得到帧码后访问内存，另外将页码和帧码添加到 `TLB`，如果 `TLB` 条目已满，那么会选择一个来使用替换算法进行替换。

- 页表的结构：分层分页式结构

磁盘

- `RAID` 磁盘冗余阵列

  优点：

  - 分布存储，提高单个 `I/O` 请求的处理性能
  - 数据的冗余，提高了系统的可靠性

  讨论：`P380`

  当讨论 RAID 0 到 RAID 6 这七个级别时，我们可以对它们进行重新简单介绍并讨论它们的优缺点：

  1. RAID 0：
  - 介绍：RAID 0将数据分散存储在多个硬盘上，提供了更高的性能，因为**数据可以并行读取和写入。**
  - 优点：卓越的读写性能和较高的数据传输速度。
  - 缺点：没有冗余功能，**单个硬盘故障会导致整个阵列的数据丢失**，可靠性较低。
  2. RAID 1：
  - 介绍：RAID 1使用镜像技术，将相同的数据同时写入两个硬盘中，提**供数据冗余。**
  - 优点：较高的可靠性，即使一个硬盘故障，另一个硬盘仍然包含完整的数据副本。
  - 缺点：**较低的容量利用率，一半的存储容量用于数据冗余。**
  3. RAID 2：
  - 介绍：RAID 2使用位级同步奇偶校验技术，将数据分散存储在多个硬盘上，并通过**奇偶校验**进行纠错。
  - 优点：高度的数据冗余和错误纠正能力。
  - 缺点：**不常用，存储冗余开销较大，性能与其他RAID级别相比较低。**
  4. RAID 3：
  - 介绍：RAID 3使用**字节级同步奇偶校验技**术，将数据分散存储在多个硬盘上，并通过奇偶校验进行纠错。
  - 优点：较高的数据传输速度和错误纠正能力。

- 缺点：单个硬盘的故障会导致整个阵列性能下降，**不适用于随机读取操作。**

5. RAID 4：
- 介绍：RAID 4使用块级同步奇偶校验技术，将数据分散存储在多个硬盘上，并通过奇偶校验进行纠错。
- 优点：较高的数据传输速度和错误纠正能力，**更高的IO总体速率。**
- 缺点：单个硬盘的故障会导致整个阵列性能下降，不适用于随机写入操作。

6. RAID 5：（最常用）
- 介绍：RAID 5使用分布式奇偶校验技术，将数据和奇偶校验位**分散存储在多个硬盘上。**
- 优点：良好的性能和较高的容量利用率，能够容忍单个硬盘故障，通过计算奇偶校验位恢复数据。
- 缺点：在重建过程中，如果有**多于两个硬盘故障，数据可能无法完全恢复。**

7. RAID 6：
- 介绍：RAID 6在RAID 5的基础上引入了**双重奇偶校验**，将数据和两个奇偶校验位分散存储在多个硬盘上。
- 优点：较高的容错能力，能够容忍同时故障两个硬盘，能够恢复数据。
- 缺点：相对于RAID 5级别，需要更多的硬盘用于奇偶校验，容量利用率较低。