进程通信与同步、进程调度

林祥，PB16020923

1. 进程间通信

进程在执行过程中，如果不被其他进程影响或影响其他进程，则称该进程是独立的，否则称为协作的。进程协作可能是出于以下几个目的：信息共享、提高计算速度、模块化设计…进程间要实现协作就需要一套**通信机制**（IPC），目前主要有两种模式：共享内存和消息传递。

消息传递模型通过在进程间交换消息来实现通信，它易于实现，不需要避免冲突，但是需要内核介入，时间消耗大，且只能传递少量信息。消息传递可以分为直接通信和间接通信，直接通信明确指定了接收者和发送者，而间接通信在多个进程间共享了一个虚拟邮箱，消息经过邮箱中转。另外，消息传递的实现可以是阻塞（同步）或非阻塞（异步）的，阻塞指发送者（接收者）一直等待直至消息被接收（有消息可用），非阻塞则不等待，如果二者皆阻塞，则它们之间就有一个“集合点”。

共享内存模型通过建立一块允许多个进程读写的内存区域实现通信，它能以较快的速度通信，可以传递较多信息，但实现较为复杂。共享内存有一个典型的生产者-消费者问题，生产者产生信息以供消费者使用，为了使它们能并发执行，在共享内存中设置一个缓冲队列，生产者可以加入一项信息，消费者可以取走一项信息。如果缓冲为空，则消费者必须等待，如果缓冲有限且已满，则生产者必须等待。

进程间通信还可以通过**套接字**（Socket）和**管道**（Pipe）实现。套接字由一个 IP 地址和端口组成，它必须独一无二，确保服务器和客户端的连接，这种方法通常用在不同计算机之间。管道可以实现父子进程之间的通信，可以是匿名或者命名的，但是只能单向传递信息。

进程间协作的一个关键问题是出现**竞争**（Race），即同时操作一个变量，这将导致执行结果与访问的特定顺序有关。

2. 进程同步

为了解决进程的竞争问题，**进程同步**是十分必要的。同步的一个基本要求是**互斥**，即两个进程不能同时操作一个共享的资源，为此，需要定义一个**临界区**，多个进程不能同时处在临界区中，并且等待进入临界区的时间应该是有限的，即临界区应该尽可能地短。此外，还要实现进入临界区和退出临界区的操作，并且能够选择一个合适的进程进入临界区。

目前有几种实现方式，一方面是从硬件来考虑。当进程要进入临界区时，关闭中断，退出临界区时，开启中断，这种方式使得其他进程不能运行，并且在多处理器环境中开销巨大。

另一方面是在软件层面来解决。一个简单的方法是**严格的轮换法**，通过设置一个变量 turn 标识，仅取值 0/1，代表允许哪个进程进入临界区，进程要进入临界区时检查 turn，如果不是自己则等待，退出临界区时，将 turn 置为对方标识，这种方法会导致 CPU 资源的浪费，并且一个进程不能连续两次进入临界区。**Peterson 方法**则对此进行了改进，增加了一个布尔型数组表示每个进程是否希望进入临界区，每个进程要进入临界区时检查该数组，如果其他进程也想进入则该进程做出让步。这种方法不会形成严格的轮换，相反地，有可能一个进程一直占着临界区导致一个更高优先级的进程无法进入。还有一种实现方法是**互斥锁**，通过共享一个新的变量 lock，进程要进入临界区时请求锁并等待至锁被释放，退出临界区时释放锁。这种方法要求请求和释放锁的操作是一种**原子操作**（不可中断地），但这种硬件层面的实现并不简单。以上几种方法都存在的问题是等待进入临界区时 CPU 资源的浪费。

实际上，一个较好的方法是通过**信号量**（Semaphore）实现，这类似一个交通指挥者。它定义一个共享的变量 Semaphore，可以是二值的（0/1，类似于互斥锁）或者是多值/计数的（表示资源数量），还实现了两个对信号量的操作 up()和 down()，进程请求资源时 down()，释放资源

时 up()。当 down()操作不能被满足时，该进程不是 wait 而是通过特殊的 sleep 使自己进入等待队列，这样就不会浪费 CPU 资源，并且其他进程 up()时会唤醒队列中等待的进程（只能是一个），该进程的 down()继续执行，并进入临界区。

使用信号量能较好地解决几个经典问题：**生产者-消费者问题、哲学家进餐问题**（哲学家围绕而坐，相邻两人共享一根筷子）、**读者-写者问题**（读者与写者以及两个写者之间不能同时操作）。如对于生产者-消费者问题，设置一个二值信号量 mutex（确保互斥）和两个计数信号量 empty/full（监视缓存的情况），生产者生产一个资源之前要先 down(empty)，再 down(mutex)，完成后先 up(mutex)再 up(full)，消费者的过程类似。

进程同步经常遇到的一个问题是**死锁**（Deadlock）。死锁的出现的原因是一个资源只能同时被一个进程使用，而进程占用一个资源时又在等待其他资源，并且不能抢占，这就形成了一个循环等待的情况。比如生产者-消费者问题中将两个 down()的顺序弄反了，生产者获得 mutex 等待 empty，而消费者等待 mutex，这就形成了死锁。

3. 进程调度

多道程序设计允许同时运行多个进程，通过在进程间切换以达到该效果。一般进程执行过程中包括了多个 CPU 区间和 I/O 等待区间，在 I/O 区间时，CPU 空闲，为了不浪费 CPU 这个宝贵的资源，因此需要进程调度，当一个进程等待时（如 I/O 等待），操作系统将 CPU 使用权交给其他进程，确保 CPU 不会空闲。

进程调度包括**抢占**的和**非抢占**的。如果调度只发生在进程必须等待或者结束时，则为非抢占调度，否则称为抢占调度。非抢占调度允许进程更连续地执行，但可能发生 CPU 占用时间过长，抢占调度则使得每个进程能更快得到 CPU 的响应。现代操作系统几乎都是抢占调度。

进程调度的**准则**主要有 CPU 使用率、吞吐量、周转时间、等待时间和响应时间等。CPU 使用率应该尽可能高，实际情况在 40%~90%较好。吞吐量指单位时间完成进程的数量，也是越高越好。周转时间指从进程提交到完成的所有时间段之和（包括等待进入内存、在就绪队列等待、执行和 I/O），等待时间即在就绪队列花费的时间，响应时间即进程提交到开始响应的时间，这三者都是越低越好。但是，这些准则之间是有冲突的，在不同环境下各有所侧重。

进程调度的**算法**主要包括先到先服务（FCFS）、最短作业优先（SJF）、轮转法（RR）、优先级调度和多级队列调度。**先到先服务**顾名思义，由一个 FIFO 队列实现，并且是非抢占的，进程执行直到等待或结束，如果一个长进程先到达，这将导致平均等待时间过长。**最短作业调度**每次选择一个 CPU 区间最短的进程执行，并且可以分为抢占式和非抢占式的，这种调度方法最大的困难在于不知道下一个 CPU 区间的长度，一种解决方法是通过公式$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$来预测。**轮转法**是定义了一个较小的时间单元（时间片），就绪队列作为循环队列，被调度执行的进程在达到一个时间片的时间后被重新加入队尾，显然，这是抢占的（除非进程长度小于时间片）。时间片的大小很大程度上决定了 RR 调度的性能，过长将变成 FCFS，过短将导致进程切换开销过大，根据经验，80%的 CPU 区间应该小于时间片。**优先级调度**为每个进程设置了优先级，优先级高（数值不一定是高）的进程先执行，同样也可以分为抢占和非抢占的，这种调度的主要问题是无穷阻塞（饥饿），即低优先级的进程可能很久都不能得到执行。**多级队列调度**则是优先级调度的一种拓展，进程被划分为多个队列，每个队列有不同的优先级，队列内部则可以根据不同属性采取以上几种不同的调度方法。进一步地，如果允许进程在不同队列之间转移，这就形成了**多级反馈队列调度**，根据进程的执行和等待时间动态调整优先级，可以防止饥饿发生，当然这种调度算法也最为复杂。

参考文献:

[1] Abraham Silberschatz. 操作系统概念. 高等教育出版社, 2010.1.