进程

进程概念

如前所述， 进程是执行的程序， 这是一种非正式的说法。 进程不只是程序代码，程序代码有时称为文本段 (或 代 码段)。 进程还包括当前活动， 如程序计数器 的值和处理器寄存器的内容等。 另外， 进程通常还包括：进程堆栈 (包括临时数据，如函数参数 、 返回地址和局部变量) 和数据段(包括全局变量) 。 进程还可能包括堆 这是在进程运行时动态分配的内存。

进程状态

新 的： 进程正在创建。

• 运 行 ： 指令正在执行。

• 等 待 ： 进程等待发生某个事件 （ 如 I/O 完成或收到信号)。

• 就 绪: 进程等待分配处理器。

• 终 止 ： 进程已经完成执行。

进程控制块PCB

• 进 程 状 态：状态可以包括新的、 就绪、 运行、 等待、 停止等。

• 程 序 计 数 器 ： 计数器表示进程将要执行的下个指令的地址。

• CPU 寄存器: 根据计算机体系结构的不同， 寄存器的类型和数量也会不同。 它们包括累加器、 索引寄存器 、 堆栈指针、 通用寄存器和其他条件码信息寄存器。

• CPU 调度信息: 这类信息包括进程优先级、 调度队列的指针和其他调度参数。

• 内存管理信息 根据操作系统使用的内存系统，这类信息可以包括基地址和界限寄存器的值、 页表或段表

• 记 账 信 息: 这类信息包括 CPU 时间、 实际使用时间、 时间期限、 记账数据、作业或进程数量等。

•I/O 状态信息: 这类信息包括分配给进程的 I/O 设备列表、 打开文件列表等

线程

每个进程是一个只能进行单个执行线程 （ thread) 的程序。

在支持线程的系统中， PCB 被扩展到包括每个线程的信息。 系统还会需要一些其他改变， 以便支持线程。

进程调度

进程调度器 选择一个可用进程 （ 可能从多个可用进程集合中） 到CPU 上执行。 单处理器系统不会具有多个正在运行的进程。 如果有多个进程， 那么余下的需要等待 CPU 空闲并能重新调度。

调度队列

进程在进入系统时， 会被加到作业队列 (job queue), 这个队列包括系统内的所有进程。

驻留在内存中的、 就绪的、 等待运行的进程保存在就绪队列 （ ready queue ）上。

等待特定 I/O 设备的进程列表， 称为设备队列

新进程被加到就绪队列； 它在就绪队列中等待， 直到被选中执行或被分派

进程可能发出 I/O 请求， 并被放到 I/O 队列。

• 进程可能创建一个新的子进程， 并等待其终止。

• 进程可能由于中断而被强制释放 CPU, 并被放回到就绪队列。

调度程序

通常， 对于批处理系统， 提交的进程多于可以立即执行的。 这些进程会被保存到大容量存储设备（通常为磁盘） 的缓冲池， 以便以后执行。 长期调度程序或作业调度程序 从该池中选择进程， 加到内存， 以便执行。 短期调度程序_ 或 CPU 调度程序 从准备执行的进程中选择进程，并分配 CPU。

两种调度程序的主要区别是执行频率

大多数进程可分为： I/O 为主或 CPU为主。 I/O 密集型进程 执行 I/O 比执行计算需要花费更多时间。 相反， CPU 密集型进程 很少产生 I/O 请求， 而是将更多时间用于执行计算。

上下文切换

当中断发生时， 系统需要保存当前运行在 CPU 上的进程的上下文 ，以便在处理后能够恢复上下文， 即先挂起进程， 再恢复进程。 进程上下文采用进程 PCB 表示，包括 CPU 寄存器的值、 进程状态和内存管理信息等，通常通过执行状态保存, 保存 CPU 当前状态 （ 包括内核模式和用户模式）之后，状态恢复重新开始运行

切换 CPU 到另一个进程需要保存当前进程状态和恢复另一个进程的状态， 这个任务称为上下文切换

进程运行

进程创建

进程在执行过程中可能创建多个新的进程。 正如前面所述， 创建进程称为父进程， 而新的进程称为子进程。 每个新进程可以再创建其他进程， 从而形成进程树

大多数的操作系统 （ 包括 UNIX、 Linux 和 Windows) 对进程的识别采用的是唯一的进程标识符, 这通常是一个整数值。 系统内的每个进程都有一个唯一pid， 它可以用作索引， 以便访问内核中的进程的各种属性。

父进程可能要在子进程之间分配资源或共享资源 （ 如内存或文件)。限制子进程只能使用父进程的资源， 可以防止创建过多进程， 导致系统超载。

除了提供各种物理和逻辑资源外， 父进程也可能向子进程传递初始化数据 （ 或输入)。

当进程创建新进程时， 可有两种执行可能：

• 父进程与子进程并发执行。

• 父进程等待， 直到某个或全部子进程执行完。

新进程的地址空间也有两种可能：

子进程是父进程的复制品（它具有与父进程同样的程序和数据)。

• 子进程加载另一个新程序。

进程终止

当进程完成执行最后语句并且通过系统调用 exitO 请求操作系统删除自身时， 进程终止。 这时， 进程可以返回状态值 （ 通常为整数） 到父进程 （ 通过系统调用 wait()h 所有进程资源， 如物理和虚拟内存、 打开文件和 I/O 缓冲区等， 会由操作系统释放。

只有终止进程的父进程才能执行这一系统调用。 否 则， 用户可以任意终止彼此的作业。 记住， 如果终止子进程， 则父进程需要知道这些子进程的标识符。 因此，当一个进程创建新进程时， 新创建进程的标识符要传递到父进程。

父进程终止子进程的原因有很多， 如：

子进程使用了超过它所分配的资源。 （ 为判定是否发生这种情况， 父进程应有一个机制， 以检查子进程的状态）。

• 分配给子进程的任务， 不再需要。

• 父进程正在退出， 而且操作系统不允许无父进程的子进程继续执行。

有些系统不允许子进程在父进程已终止的情况下存在。 对于这类系统， 如果一个进程终止 （ 正常或不正常)， 那么它的所有子进程也应终止。 这种现象， 称为级联终止

当一个进程终止时， 操作系统会释放其资源。 不过， 它位于进程表中的条目还是在的，直到它的父进程调用 wait(); 这是因为进程表包含了进程的退出状态。

当进程已经终止，但是其父进程尚未调用 wait()， 这样的进程称为**僵尸进程**。所有进程终止时都会过渡到这种状态， 但是一般而言僵尸只是短暂存在。 一旦父进程调用了 waitO,僵尸进程的进程标识符和它在进程表中的条目就会释放。

如果父进程没有调用 waitO 就终止， 以致于子进程成为**孤儿进程** （Linux 和 UNIX 对这种情况的处理是： 将 init 进程作为孤儿进程的父 进程。

进 程 间 通 信

操作系统内的并发执行进程可以是独立的或也可以是协作的。

提供环境允许进程协作，

信息共享

计算加速

模块化

方便

进程间通信有两种基本模型

共享内存

共享内存模型会建立起一块供协作进程共享的内存区域， 进程通过向此共享区域读出或写入数据来交换信息

消息传递

消息传递模型通过在协作进程间交换消息来实现通信。

优缺点

消息传递对于交换较少数量的数据很有用， 因为无需避免冲突。 对于分布式系统， 消息传递也比共享内存更易实现。

共享内存可以快于消息传递，这是因为消息传递的实现经常采用系统调用，因此需要消耗更多时间以便内核介入。 与此相反， 共享内存系统仅在建立共享内存区域时需要系统调用； 一旦建立共享内存， 所有访问都可作为常规内存访问， 无需借助内核。

共享内存系统

一片共享内存区域驻留在创建共享内存段的进程地址空间内。 其他希望使用这个共享内存段进行通信的进程应将其附加到自己的地址空间。 回忆一下， 通常操作系统试图阻止一个进程访问另一进程的内存。 共享内存需要两个或更多的进程同意取消这一限制； 这样它们通过在共享区域内读出或写入来交换信息。 数据的类型或位置取决于这些进程， 而不是受控于操作系统。 另外， 进程负责确保， 它们不向同一位置同时写入数据

消息传递系统

操作系统提供机制， 以便协作进程通过消息传递功能进行通信。

直接或间接的通信

对于直接通信, 需要通信的每个进程必须明确指定通信的接收者或发送者

在间接通信 中， 通过邮箱或端口来发送和接收消息。

同步或异步的通信

消息传递可以是阻塞或非阻塞, 也称为同步 或异步。

阻塞发送: 发送进程阻塞， 直到消息由接收进程或邮箱所接收

• 非阻塞发送 ： 发送进程发送消息， 并且恢复操作 ，

• 阻塞接收 ： 接收进程阻塞， 直到有消息可用。

• 非阻塞接收 ： 接收进程收到一个有效消息或空消息。

自动或显式的缓冲

通信进程交换的消息总是驻留在临时队列中

零容量: 队列的最大长度为 0; 因此， 链路中不能有任何消息处于等待。 对于这种情况， 发送者应阻塞， 直到接收者接收到消息。

有限容量 (: 队列长度为有限的《; 因此， 最多只能有《 个消息驻留其中。 如果在发送新消息时队列未满， 那么该消息可以放在队列中

无 限 容 置: 队列长度可以无限， 因此， 不管多少消息都可在其中等待。 发送者从不阻塞。

客户机 / 服务器通信

套接字为通信的端点。 通过网络通信的每对进程需要使用一对套接字， 即每个进程各有一个。由一个 IP 地址和一个端口号组成。

远程过程调用： 与 IPC 的消息不一样， RPC 通信交换的消息具有明确结构， 因此不再仅仅是数据包。消息传到 RPC 服务， RPC 服务监听远程系统的端口号； 消息包含用于指定： 执行函数的一个标识符以及传递给函数的一些参数。

管道 (pipe) 允许两个进程进行通信。

普通管道允许两个进程按标准的生产者 - 消费者方式进行通信： 生产者向管道的一端(写入端） 写， 消费者从管道的另一端 （读出端） 读。 因此， 普通管道是单向的， 只允许单向通信。

命名管道提供了一个更强大的通信工具，通信可以是双向的，并且父子关系不是必需的，当建立了一个命名管道后多个进程都可用它通信