

Operating System - P10

层次式系统 上层软件都是在下一层软件的基础上构建的，如 THE 系统：

0 处理器分配 和多道程序设计	1 存储器和 磁盘管理	2 操作员— 进程通信	3 输入/输出管理	4 用户程序	5 操作员
基本的 CPU 多道 程序设计功能	内存管理 分配进程主存空间	进程与操作员 控制台(用户)通信	进程与良好特性 向抽象 I/O 设备	无需考虑进程、内存 控制台、I/O 的细节	系统操作员 进程

与 THE 的层次式结构相比，MULTICS 采用同心环结构
内环有更高的级别，外环调用内环进程需要等价于 TRAP 的指令，并进行严格的参数检查
THE 系统的各部分最终仍链接成单个目标程序，而 MULTICS 的环形结构由硬件实现
环形结构的一个优点是容易扩展，以构造用户子系统

微内核 将系统划分成小的、良好定义的模块，只有“微内核”运行在内核态，以实现高可靠性，如 MINIX 3

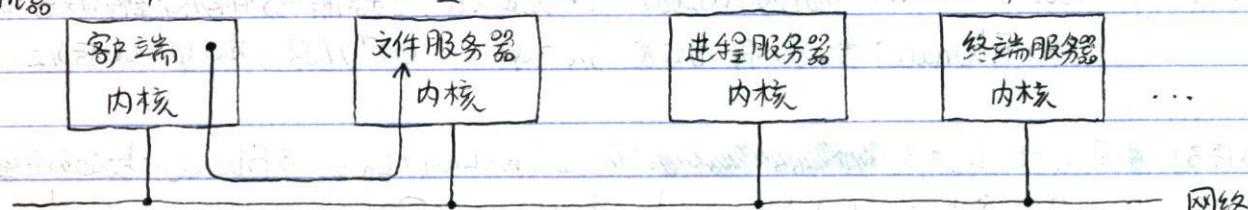
Shell	Make	... ← 进程	用户程序	用户态	设备驱动运行在用户态，所以不能物理地访问 I/O 端口空间，也不能直接发出 I/O 命令，需要生成内接 调用	
FS	Proc.	Reinc.	服务器			
Disk	TTY	New	Print			驱动程序
Clock	Sys	中断处理、进程、调度、进程间通信(IPC)				

再生服务器 (reincarnation server) 检查其他服务器和驱动器是否正常，出现错误则自动取代
无需用户干预，使得系统具有自我修复能力

机制与策略分离 与微内核相关的思想，典型如一个调度算法，为每个进程赋优先级，并让内核执行最高优先级的进程
机制（在内核中）是寻找最高优先级进程并运行，而策略（在用户态中）决定了进程优先级
分离策略与机制，从而使系统内核更小

客户端-服务器 模式 将进程划分为两类：服务器提供某种服务，客户端使用这些服务
通常系统最底层是微内核，但并不必须如此
典型的方式是，客户端和服务器运行在不同的计算机上，通过局域网或广域网连接
对于客户端而言，本地处理和远程处理，两种情形一样，都是发送请求并得到回应

机器 1 2 3 4



Operating System - P12

外核 (exokernel). 与虚拟机克隆真实机器不同，而是对机器进行分区，用户得到资源的一个子集。任务是为虚拟机分配资源，检查使用资源的企图，以确保没有机器使用他人资源。
优点是减少了映像层，并将多道程序(外核内)与用户操作系统代码(用户空间内)加以分离。

指针 (语言有而 Java, Python 没有的是显式指针 (explicit pointer), 指向(即包含对象地址)一个变量或数据结构)

类型安全 (type safety) [C语言中没有包括, 还有内建字符串, 线程, 包, 类, 对象]
垃圾回收 (garbage collection), OS 的“淋浴器塞子”]

分配存储空间 (语言中分配或者是静态的, 或者是程序员明确分配或释放的, 通常使用 malloc 和 free 函数)

进程 (process) 计算机上所有可运行的软件被组织成若干顺序进程 (sequential process)

一个进程是一个正在执行的程序的实例, 包括程序计数器、寄存器和变量的当前值。
从根本上说, 每个进程拥有自己的虚拟 CPU, 但实际上 CPU 在各个进程之间切换。
技术上, 一个进程是某种类型的一个活动, 具有程序, 输入, 输出, 状态。

伪并行 (parallelism) 单个CPU在进程之间快速切换, 从而产生进程并行的感觉。
以此区分多处理器系统 (多个CPU共享同一个物理内存, 可能有独立或共享的L2 cache) 的真正硬件。
这种快速切换即多道程序设计。

由于进程运算速度不确定, 且再次运行不可再现, 所以不能对进程时序做任何想当然的假设。

进程创建 (daemon) 系统初始化: 其中与特定用户没有关系, 具有某些专门功能, 停留在后台处理事件的称为守护进程。

正在运行的进程执行: 当工作可以容易地划分成若干相关的但没有相互作用的进程时, 创建。

创建进程的系统调用: 新的进程效果拨打群。

用户请求创建新进程: 交互式系统中, 键入命令或点击图标, 新进程接管它的窗口 (UNIX 中)。

批处理作业初始化: 大型机的批处理系统, OS认为有资源运行另一个作业时, 创建进程并输入到 next

fork-execve UNIX中只有一个系统调用 fork 可以用来创建新进程, 该调用会创建一个与调用进程相同的副本。
父进程与子进程拥有相同的内存映像、环境字符串, 打开文件, 相同的文件句柄。

随后执行 execve 调用修改内存映像, 并执行一个新程序。

用处是在 fork 和 execve 之间, 子进程处理文件描述符, 完成标准输入/输出/错误文件的重定向。

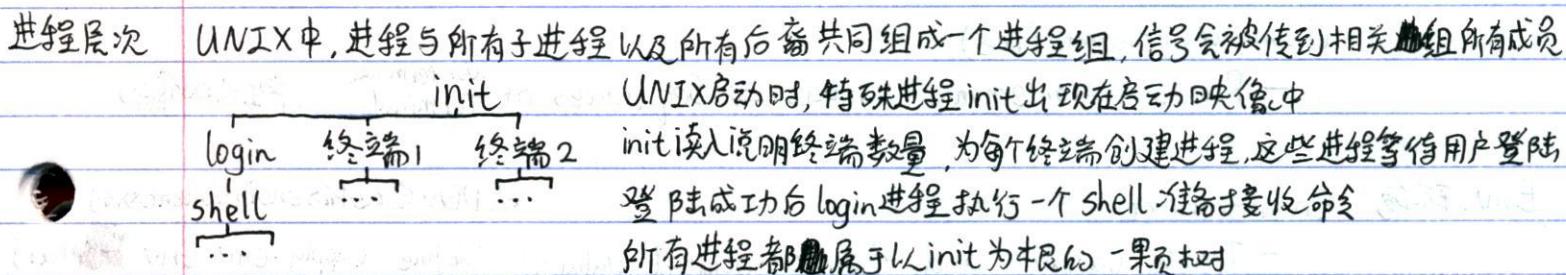
Create Process 10个参数的Win32调用: 执行程序, 命令行参数, 安全属性, 打开文件继承控制位, 优先级信息, 创建窗口根据
指向结构的指针, 结构中新创建进程的信息将返回调用者。

Operating System - P. 13

写时复制 (copy-on-write), 如果子进程共享父进程的所有内存, 修改内容前首先明确地复制, 以确保
修改发生在私有内存区域 (STOLEN)

父进程与子进程有不同的地址空间, 某个进程在地址空间中修改了一个字, 对其他进程不可见
可写的内存不可以共享, 不可写的内存可以共享, 某些 UNIX 中实现父子进程的程序正义共享

进程终止	正常退出	出错退出 (自愿) (非自愿)	严重错误	被其他进程杀死
	完成工作而终止	进程发现了严重错误	通常是指程序中的错误	另一个进程执行系统调用杀死
	(编译器完成后执行) 如 cc foo.c 但存在 UNIX 中为 exit	如执行非法指令 (通知 OS)	如执行非法指令 (通知 OS)	UNIX 中为 kill
	Windows 中为 Exit Process	编译器将退出 → 错误参数	进程可以自行处理错误	Win32 为 Terminate Process

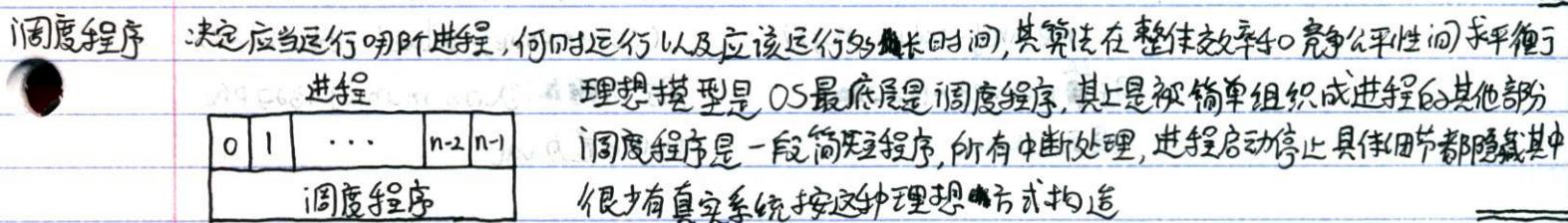
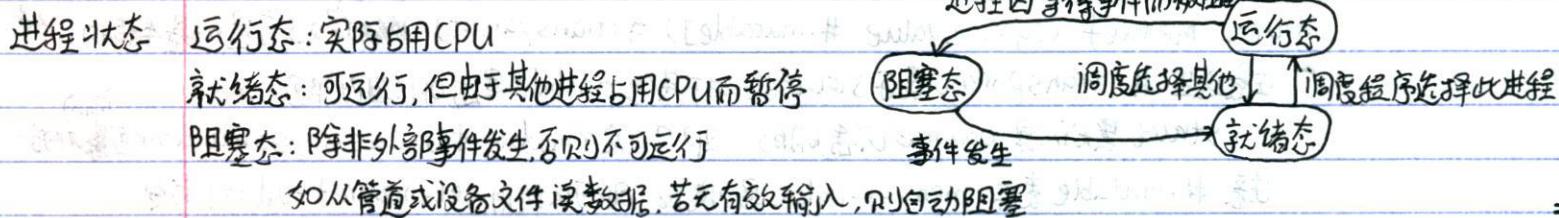


句柄

Windows 中没有进程层次, 而是在创建子进程时, 父进程得到一个特别令牌 (句柄)
句柄可以控制子进程, 也可以由父进程传递给其他进程

阻塞

当一个进程在逻辑上不能继续运行时, 就会被阻塞, 典型如等待可使用的输入
与之相比的是, 极端上能运行的进程被硬停止, 典型如 OS 调度另一个进程占用 CPU
注意前者挂起是程序自身固有原因, 后者是由技术系统的原因引起



Operating

System - P14

进程表 是由OS维护的一个结构数组, 每个进程占用一个进程表项(或称进程控制块) **process table** 包含进程状态的重要信息, 以及切换到阻塞态必须保存的信息。

从而保证该进程随后可以再次启动, 且如同从未中断, 其典型关键字段如下:

进程管理	存储管理	文件管理
寄存器(Reg)、程序计数器(PC)、程序状态字(PSW)	堆栈段	根目录
堆栈指针、进程状态、优先级、调度参数	堆栈	工作目录
进程ID(PID)、父进程、进程组、信号	指针 (地址空间)	文件描述符
进程开始时间、使用CPU时间、子进程CPU时间	数据段	用户ID
下次定时器时间	正文段	组ID

中断向量 与I/O类关联的靠近内存底部的固定区域, 包括中断服务程序的入口地址
interrupt vector

中断发生后的系统操作步骤:

硬件 { 1. 硬件压入堆栈程序计数器等: 中断硬件将 PC, PSW, 一个或多个Reg压入堆栈

2. 硬件从中断向量装入新的程序计数器: 计算机跳转到中断向量指定的地址

汇编语言 { 3. 汇编语言过程保存寄存器值: 所有的中断都从保存寄存器开始, 通常存在进程表项中

C语言高级语句描述 { 4. 汇编语言过程设置新的堆栈: 从堆栈中删除由中断硬件压入的PC信息

堆栈指针指向进程处理程序使用的临时堆栈

5. C中断服务例程运行(典型地读和缓冲输入)

6. 调度程序决定下一个将运行的进程

7. C过程返回至汇编代码

汇编语言 { 8. 汇编语言过程开始运行新的当前进程: 为当前进程装入Reg值以及内存映射并启动

一个进程执行中可能中断数千次, 关键是中断后进程都返回到中断发生前完全相同的状态

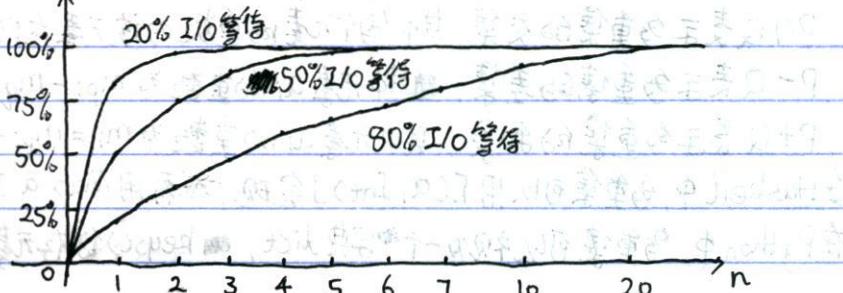
道数

多道程序设计的道数(degree of multiprogramming), 令表示为n

并令一个进程等待I/O操作时间与停留在内存中时间比为P

CPU利用率 可表示为: CPU利用率 = $1 - P^n$ (注意此处假设进程间相互独立)

注意: 由此前推断
更精确的模型
用排队论构建



Operating

System - P15

线程

进程的定义近似于拥有一个地址空间和一个控制线程，但是存在着在一个地址空间中能并行运行多个控制线程的情形，差不多像分离的进程。

多线程：1. 并行实体拥有共享同一个地址空间和所有可用处理器的能力。

2. 比进程更容易、更快创建或撤消，大约快 10~100 倍，对大量线程动态快速修改效果拔群。

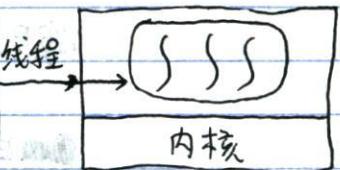
3. 提高性能，对非 CPU 密集型，即允许大量计算和大量 I/O 处理彼此重叠进行。

4. 在多 CPU 系统中，真正的并行有 3 实现的可能。

如在一个文字编辑进程中，重新计算文档格式

可以有多个线程负责不同工作。

但共享同一个文件和地址空间



进程

内核

Web 服务器 单线程：主循环获得请求并检查，并且在取下一个请求前完成当前工作。

等待磁盘操作时空转，不处理任何到来请求。

(高速缓存 cache：服务器保存获得大量访问页面的页面集合)

工作线程 (worker thread)：唤醒后检查页面是否在 cache，不在则执行

从磁盘 read 的操作，阻塞直到完成

分派程序 (dispatcher)：读入请求，检查后分派给一个阻塞的工作线程

并唤醒，或者在被 read 阻塞时选择其他进程。

多线程模型允许把服务器编写为顺序线程的集合，以改善性能。

在没有多线程时还可以选择非阻塞版本的 read 系统调用，其回答多以信号/中断形式出现。

主循环在请求到来时，检查是否在 cache 中满足，如不满足则启动一个非阻塞的系统调用 (磁盘操作)。

在表中记录请求状态后处理下一请求，可能是新请求，也可能是磁盘对之前操作的回答。

这种设计没有“顺序进程”，每次服务器为请求切换状态都必须显式地保存/装入相应的计算状态。

称为有限状态机 (finite-state machine)，其中每计算一个保存的状态，存在一个会改变状态的事件。

模型 并行性 系统调用 顺序进程 编程设计 应用

单线程	无	阻塞	是	简单	通常在唯一机器上
-----	---	----	---	----	----------

多线程进程	有	阻塞	是	简单	多线程可用时
-------	---	----	---	----	--------

有限状态机	有	非阻塞 (中断/信号)	否	复杂	多线程不可接受
-------	---	-------------	---	----	---------

单线程不可接受

对于处理极大量数据的应用，多线程提供一种方案：(只能在系统调用阻塞线程而非整个进程时有效)

输入线程：把数据读入到输入缓冲区中。

处理线程：从输入缓冲区取出数据，处理后结果放到输出缓冲区。

输出线程：把结果写到磁盘上。

Operating

System - P16

线程
thread

首先，进程模型基于两个独立概念：资源分组处理与执行

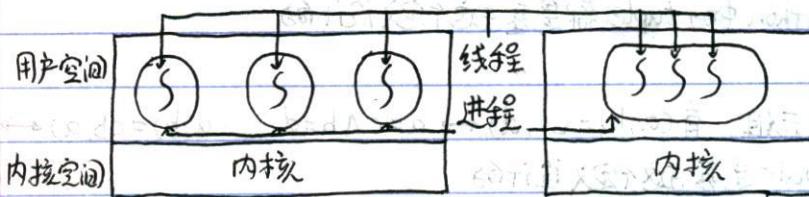
进程将相关资源集中在一起，并存放程序正文、数据以及其他资源的地址空间

线程是CPU上调度的实体。进程所拥有的执行的线程，也称轻量级进程(lightweight process)

包括一个程序计数器(CPC)，用来记录下一条执行的指令

寄存器(Register)，用来保存线程当前的工作变量

堆栈(Stack)，用来记录执行历史，每帧保存一个已调用但未从中返回的过程



同样是拥有3个线程，但区别在于是否有相同的地址空间(是否在同一进程中)

线程(同一进程)间既不可能，也没有必要保护，即线程可以修改另一线程的数据

内容 进程中的内容(线程间共享) 线程中的内容(线程间独立)

地址空间，全局变量，打开文件，子进程

程序计数器，寄存器，堆栈，状态

即将发生的定时器，信号与处理程序，账户信息

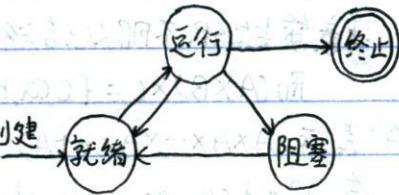
(注意：虽然独立，但相互之间是开放透明的)

状态：运行：拥有CPU并且是活跃的

阻塞：正在等待某个释放它的事件

就绪：可被调度运行，且轮到时可以很快运行

终止：运行结束



堆栈：每个线程的堆栈中有一帧，供各个被调用但尚未返回的过程使用，存放其局部变量和返回地址

通常每个线程调用不同过程，从而有一个各自不同的执行历史，所以线程需要有自己的堆栈

thread_create 库函数用于创建新进程，并专门指定新线程运行的过程名，新线程自动在创建其的地址空间运行

线程是平等的，通常不存在父子关系，只是创建时返回一个线程标识符，即新线程的名字

thread_exit 用于退出线程，执行后线程消失，不再可以调度

thread_join 用于使一个线程等待一个特定线程退出，阻塞调用此过程的线程直到特定线程退出

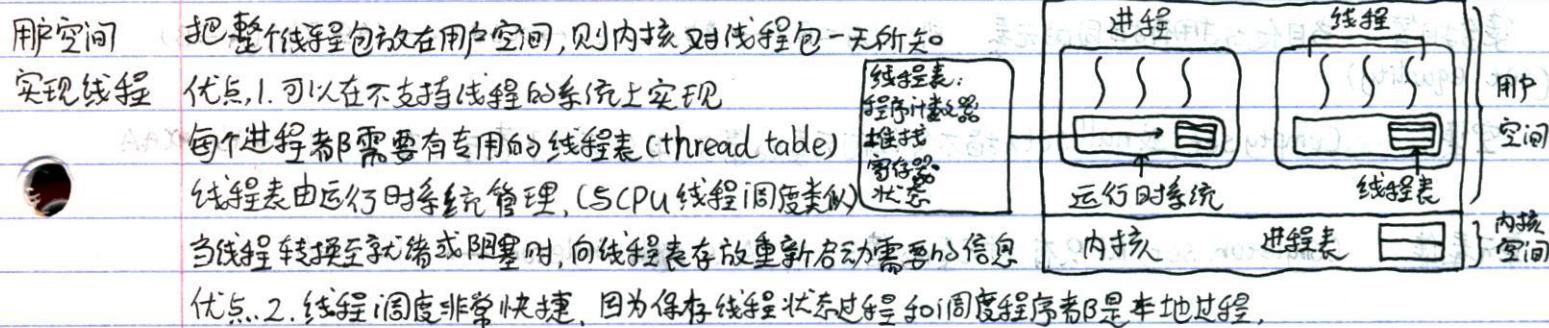
thread_yield 允许线程自动放弃CPU从而让另一个线程运行，因为线程库无法利用时钟中断让线程让出CPU

复杂性：子进程与父进程间是否公用相同线程，如果一个阻塞(thread)收到信号时，应该由谁接收信号

或者一个线程关闭了另一个线程使用的文件，或者两个线程先后分配内存导致同一内存分配两次

Operating System - P17

- pthread: POSIX中定义的线程包, IEEE标准1003.1C中定义了线程的标准, 以及调用
具有一个标识符, 一组寄存器(包括程序计数器PC), 一组存储在结构(struct)中的属性
包括堆栈大小, 调度参数及其他线程需要的项目
- 部分调用:
- pthread_create: 创建一个新线程, 通过创建的线程标识符作为函数结果返回
 - pthread_exit: 当线程完成工作时, 调用终止该线程并释放栈
 - pthread_join: 调用等待特定线程终止, 等待的线程标识符作为一个参数给出
 - pthread_yield: 遇到上没有阻塞, 而假设该线程已运行足够长时间, 并希望给其他线程机会运行
进程没有这种调用因为假设进程间有激烈竞争性, 且每个进程都希望独占CPU
 - pthread_attr_init: 建立关联一个线程的属性结构并初始化成默认值
 - pthread_attr_destroy: 删除一个线程的属性结构并释放内存, 并不影响调用线程的存在



- 包装器 (jacket or wrapper) 即在系统调用周围从事检查的代码
- 问题2. 错误中断, 如果调用跳转到一条不在内存的指令, OS将从磁盘上取回, 从而阻塞整个进程
- 问题3. 无法用轮转调度的方式调度线程, 因为在进程内部没有时钟中断
- 内核中实现 由内核支持和管理线程, 不再需要运行时系统, 内核线程表保存信息
- 阻塞线程的调用以系统调用实现, 相比运行时系统代价可观
- 方案为捕获时“回收线程”, 线程为不可运行, 但内部数据结构不受影响
- 不需要新的, 非阻塞的系统调用, 只是系统调用的代价比较大
- 同样存在进程创建是否复制线程以及信号由谁接收的复杂性问题
- 