

Lecture 2: 知识 - 进程与线程

进程的基本概念

概念：程序

- 就是一个指令序列，程序本身不是进程；
- 程序只是被动 (`passive`) 实体，如存储在磁盘上包含一系列指令的文件；

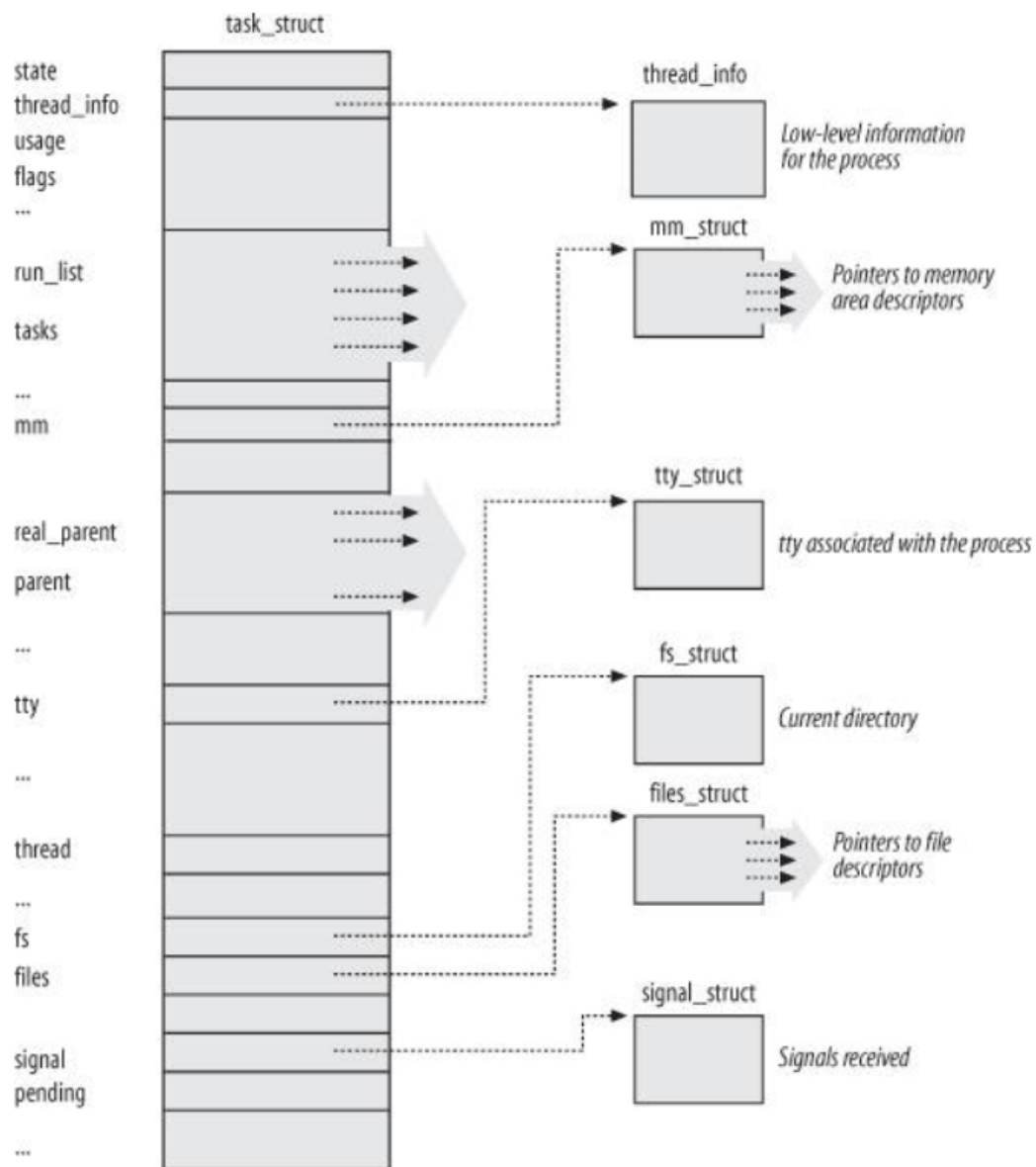
概念：进程

- 概念
 - 1. 进程是程序的一次执行过程；
 - 2. 进程是一个程序及其数据在处理机上顺序执行时所发生的活动；
 - 3. 进程是具有独立功能的程序在数据集合上运行的过程，它是系统进行资源分配和调度的一个独立单位。
 - 4. 进程是活动 (`active`) 实体，具有一个程序计数器用于表示下个执行命令和一组相关资源。当一个可执行文件被加载到内存时，这个程序就成为进程；
- 补充：进程实体下的进程定义
 - 进程是进程实体的运行过程，是系统进行资源分配和调度的一个独立单位。

概念：进程的构成

- 概念
 - 程序段，数据段，PCB 三部分组成了 **进程实体（进程映像）**；
 - 一般情况下，我们把 **进程实体就简称为进程**；
 - PCB 是 **进程存在的唯一标识**，系统是根据 `PCB` 来感知进程的存在，其 **常驻于主存** 当中；
- 组成：
 - `PCB`：
 - 包含进程描述信息、进程控制和管理信息、资源分配信息、处理机相关信息；
 - 程序段（文本段）：
 - 包含程序指令；
 - 数据段：
 - 包含运行过程中产生的各种数据；
 - 堆栈：
 - 包括临时数据，如函数参数、返回地址和局部变量；
- 注意
 - 进程实体和进程并不一样：进程实体是静态的，进程则是动态的；
 - 除非题目专门考察二者区别，否则可以认为进程实体就是进程；
- 分析：进程与程序的关系
 - 多对多；

补充：Linux 中进程



进程控制块

概念: PCB

- 组成
 - 进程状态
 - 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
 - 进程描述信息
 - 进程标识符 `PID`
 - 用户标识符 `UID`
 - 进程控制和管理信息
 - 进程当前状态: 描述进程的状态信息, 作为 CPU 分配调度的依据
 - 进程优先级: 描述进程抢占 CPU 的优先级, 优先级高的进程可优先获得 `CPU`;
 - 资源分配清单
 - 程序段指针

- 数据段指针
- 键盘
- 鼠标
- 处理机相关信息
 - 各种寄存器值
- 图示

进程描述信息	进程控制和管理信息	资源分配清单	处理机相关信息
进程标识符 (PID)	进程当前状态	代码段指针	通用寄存器值
用户标识符 (UID)	进程优先级	数据段指针	地址寄存器值
	代码运行入口地址	堆栈段指针	控制寄存器值
	程序的外存地址	文件描述符	标志寄存器值
	进入内存时间	键盘	状态字
	CPU 占用时间	鼠标	
	信号量使用		

进程状态
进程编号
程序计数器
寄存器
内存界限
打开文件列表
...

进程的组织方式

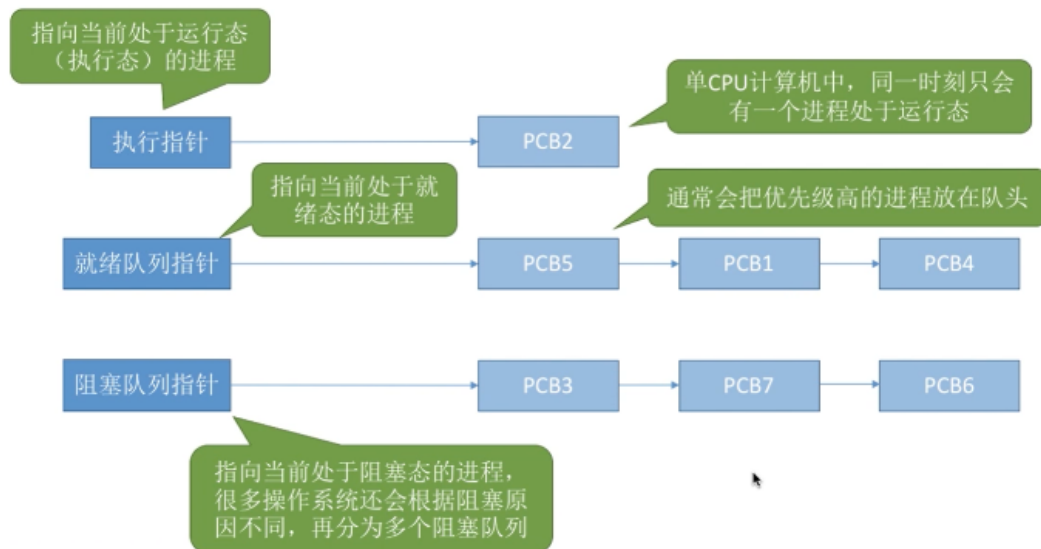
概述：进程的组织方式

- 链接方式
 - 按照进程状态将 PCB 分为多个队列
 - 操作系统持有指向各个队列的指针
- 索引方式
 - 根据进程状态的不同，建立几张索引表
 - 操作系统持有指向各个索引表的指针

概念：链接方式

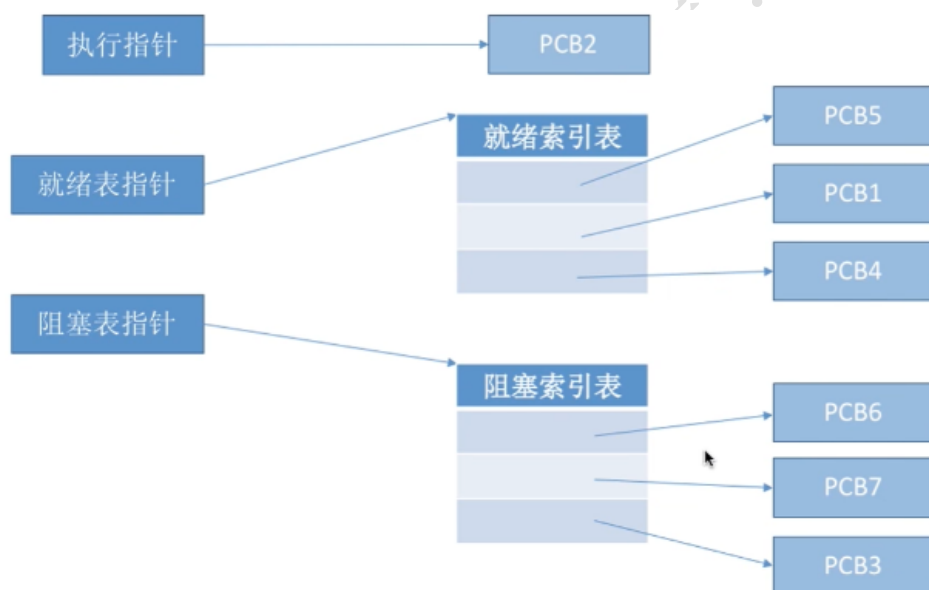
- 概念：
 - 不同状态下的进程，使用不同的队列来链接不同状态的进程；

- 图示：



概念：索引方式

- 概念：
 - 使用索引表，用索引表的表项来获得不同状态的进程；
- 图示：



进程的状态与转换

进程的状态

概念：进程状态

- 基本状态
 - 就绪态 (Ready)
 - 进程获得了除 CPU 外的一切所需资源，一旦得到 CPU，便可立即运行；
 - 系统中处于就绪态的进程可能有多个，通常将它们排成一个队列，称为就绪队列；
 - 运行态 (Running)
 - 占有 CPU，并在 CPU 上运行；
 - 多核 CPU 的情况下，可以有多个进程处于运行态；

- 无论是单 CPU 还是多核 CPU 下，都可能同时所有进程都不处于运行态（发生死锁时，可能出现所有进程都阻塞）
- 阻塞态（Waiting/Blocked）
 - 进程正在等待某一事件而暂停运行，如等待某个资源可用（不包括 CPU）或等待 IO 完成；
 - 即使 CPU 空闲，该进程也不能运行；
 - 系统通常将处于阻塞态的进程也排成一个队列，甚至根据阻塞原因的不同，设置多个阻塞队列；
- 其他状态
 - 创建态（New）
 - 又称新建态
 - 进程正在被创建，尚未转到就绪态；
 - 创建进程需要多个步骤：
 1. 首先申请一个空白 PCB，并向 PCB 中填写用于控制和管理进程的信息；
 2. 然后为该进程分配运行时所必须的资源；
 3. 最后将该进程转入就绪态并插入就绪队列；
 - 在创建进程的步骤当中，如果进程所需的资源尚不能得到满足，如内存不足，则创建工作尚未完成，进程此时所处的状态称为创建态；
 - 终止态（Terminated）
 - 又称结束态
 - 进程正从系统中消失，可能是进程正常结束或其他原因退出运行；
 - 进程需要结束运行时，系统首先将该进程置为终止态，然后进一步处理资源释放和回收等工作；
- 注意：
 - 单核处理机环境下，每一时刻最多只有一个进程处于运行态（双核环境可以两个）；
- 注意：就绪态和阻塞态区别
 - （1）依赖不同：
 - 两者都是缺少某一东西、在等待执行，但就绪态只缺少 CPU，只要获得了 CPU 就立马可以开始执行；而阻塞态是指进程需要其他资源（除了 CPU）或等待某一事件；
 - （2）就绪态和阻塞态的时间：
 - 进程得到 CPU 的时间很短且非常频繁，进程在运行过程中实际上是频繁地转换到就绪态的；
 - 而其他资源（如外设）的使用和分配或某一事件的发生（如 IO 完成）对应的时间相对来说很长，进程转换到阻塞态的次数也相对较少；

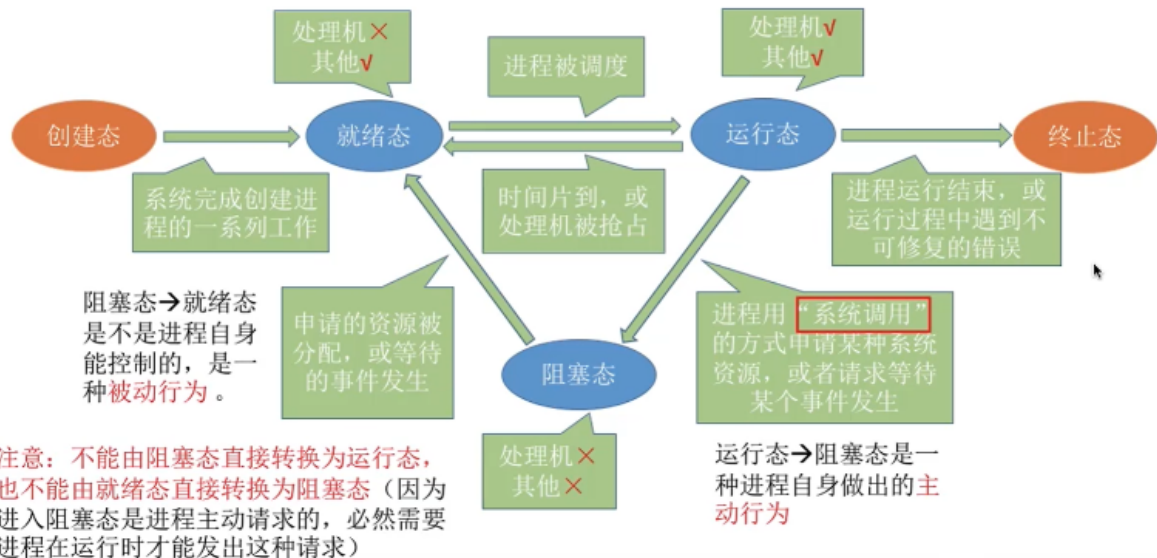
进程状态的转换

分析：进程状态间的转换

- 重点：
 - 就绪态 -> 运行态：
 - 处于就绪态的进程被调度后获得 CPU 资源（分派 CPU 的时间片）

- 运行态 -> 就绪态：
 - 1. 时间片到；
 - 2. CPU 被其他**高优先级**的进程抢占；
 - 真题：2015年
 - 被高优先级进程抢占时，会导致 CPU 从运行态到就绪态，因为被高优先级进程抢占后，进程依然具备运行的所有条件，除了 CPU 的执行，因此应当挂到就绪队列当中，等待 CPU 的调度；
- 运行态 -> 阻塞态：
 - 1. 请求系统某一资源的分配；
 - 真题：2018年
 - 进程申请临界资源；
 - 真题：2022年
 - 进程申请外设；
 - 2. 等待某事件发生；
 - 等待 IO 操作的完成；
 - 申请内存失败；
 - 3. 信号量相关操作
 - 真题：2015年
 - P 操作
 - 执行 P 操作后，可能会出现继续进行或被阻塞两种情况，不会转换为就绪态；
 - 真题：2020年
 - 当 wait() 的值小于 0 时，进程同样进入阻塞状态；
 - 是一种**进程本身的主动行为**；
- 阻塞态 -> 就绪态：
 - 1. 资源分配到位；
 - 真题：2014年
 - 当进程向操作系统提出读磁盘请求后，会进入**阻塞态**，当 IO 操作完成时，必须唤醒该进程，将其状态转换为**就绪态**；
 - 2. 等待的事件发生；
 - 比如所请求的 IO 操作完成后；
 - 阻塞态 -> 就绪态是一种进程受 OS 控制的**被动行为**，需要其他进程的协助；
- 其他：
 - 创建态 -> 就绪态：
 - 系统完成创建进程相关工作；
 - 运行态 -> 终止态：
 - 进程运行结束，或运行过程中遇到不可修复的错误；
- 注意：
 - (1) 不能由阻塞态直接转换为运行态；
 - (2) 不能由就绪态直接转换为阻塞态；
 - 因为**进入阻塞态是进程主动请求**的，必然需要进程在运行时才能发出这种请求

- 图示：



进程控制

概念：进程控制

- 主要功能是对系统中的所有进程实施有效管理，它具有创建新进程，撤销已有进程，实现进程状态转换等功能；

概念：原语

- 概念：
 - 原语的特点是**执行期间不允许中断**，只能一气呵成，这种不可被中断的操作即原子操作；
 - 如果不使用原语，可能会因为中断导致进程状态切换的时间不一致性；
 - 即原语具有原子性；
- 原语的实现：
 - 1. 原语采用关中断指令和开中断指令实现；
 - 2. 关中断和开中断是**只允许在核心态下执行的特权指令**；
 - 3. 原语的指令被放在了关中断和开中断指令之间，因此在原语指令过程当中，程序不会接收到外部中断的影响；
 - 什么是关中断：CPU 执行了关中断指令之后，就不再例行检查中断信号，直到执行开中断指令之后才会恢复检查；
 - 什么是开中断：CPU 执行了开中断指令之后，继续例行检查中断信号；

进程的创建

概念：进程的创建

- 实现：
 - 真题：2021年
 - (1) 分配 PID 与 PCB
 - 为新进程分配一个唯一的进程标识号，并申请一个空白 PCB (PCB 是有限的)；
 - 若 PCB 申请失败，则创建失败；

- (2) 为新进程分配所需资源
 - 为进程分配其运行所需的资源，如内存、文件、I/O 设备和 CPU 时间等 (在 PCB 中体现)；
 - 这些资源或从操作系统获得，或仅从其父进程获得；
 - 如果资源不足 (如内存)，则并不是创建失败，而是处于创建态，等待内存资源；
- (3) 初始化 PCB 及其 CPU 上下文信息
 - 初始化 PCB，主要包括初始化标志信息、初始化 CPU 状态信息和初始化 CPU 控制信息，以及设置进程的优先级等；
- (4) 将 PCB 插入就绪队列
 - 若进程就绪队列能够接纳新进程，则将新进程插入就绪队列，等待被调度运行；
- 分析：引起进程创建的事件
 - 真题：2010年
 - 用户登录：
 - 用户登录成功，系统会为其建立一个新的进程；
 - 启动程序执行：
 - 可以是操作系统启动创建服务进程响应用户，也可以是现有用户进程启动新进程，这些都可能会创建新进程；
 - 作业调度：
 - 多批处理器系统中，有**新的作业放入内存**时，会为其建立一个新的进程
 - 父进程派生：
 - 用户进程主动请求创建一个子进程
 - 易错点：
 - **设备分配不需要创建新进程实现**，仅需要设置相对应的数据结构

父子进程

概念：父子进程与进程数

- 概念
 - 1. 创建进程称为父进程，而新的进程称为子进程；
 - 2. 每个新进程可以再创建其他进程，从而形成进程树；
- 补充：根进程
 - Linux 中的 init 为根进程，其 pid 为 1；
- 补充：进程结束
 - 真题：2024年
 - 在 Linux 下，如果父进程比子进程更先结束，则子进程短期内会变成孤儿进程，但最终会被 init 进程领养，由 init 进程代为管理，并负责释放其资源；

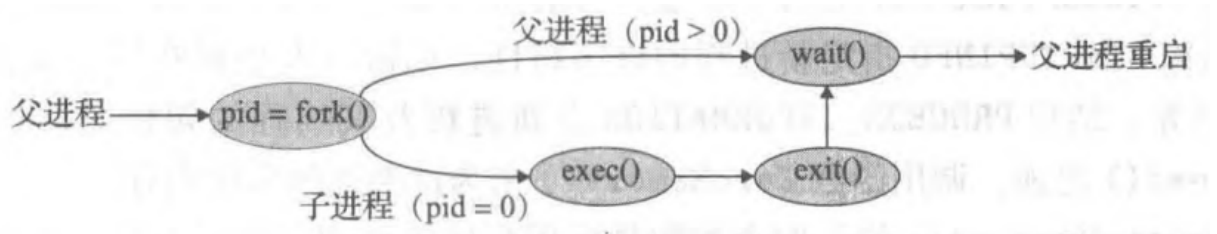
概念：资源子集与子进程

- 当一个进程创建子进程时，该子进程会需要一定的资源，这些资源可以从 OS 分配，也可以从父进程来分配资源子集；
- 真题：2020年

- 父进程可与子进程共享一部分资源，但不能共享虚拟地址空间；
- 在创建子进程时，会为子进程分配资源，如虚拟地址空间等；
- 新进程的地址空间有两种可能：
 - 子进程是父进程的复制品；
 - 子进程加载另一个新程序；

补充：父进程使用 `fork` 创建子进程

- 图示：



- 代码：

```

#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    pid_t pid;
    /* fork a child process */
    pid = fork();

    /*error occurred */
    if(pid<0)
    {
        fprintf(stderr,"Fork Failed");
        return 1;
    }
    else if(pid==0)
    {
        /* child process */
        execlp("/bin/ls","ls",NULL);
    }
    else /*parent process */
    {
        wait(NULL);
        printf("child Complete");
    }
}

```

进程终止

概念：进程的终止

- 实现
 - 从 `PCB` 集合中找到终止进程的 `PCB`
 - 若进程正在运行，立即剥夺 `CPU`，将 `CPU` 分配给其他进程
 - 终止其所有子进程；
 - 将该进程拥有的所有资源归还给父进程或操作系统；
- 分析：引起进程终止的事件
 - (1) 正常结束
 - 当进程完成执行最后语句并且通过系统调用 `exit()` 请求操作系统删除自身；
 - (2) 异常结束
 - 若被终止进程处于运行状态，立即终止该进程的执行，将 `CPU` 资源分配给其他进程；
 - (3) 外界干预：
 - 1. 进程通过适当系统调用 (如 `Windows` 的 `Terminate-Process ()`)，可以终止另一进程；
 - 2. 通常只有终止进程的父进程才能执行这一系统调用；
 - (4) 子进程与父进程：
 - 若该进程还有子孙进程，则通常需将其所有子孙进程终止 (有些系统无此要求)；

概念：级联进程

- 概念
 - 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；
- 实现：
 - 级联终止 (`cascade termination`) 通常由操作系统来启动；

进程的阻塞、唤醒和切换

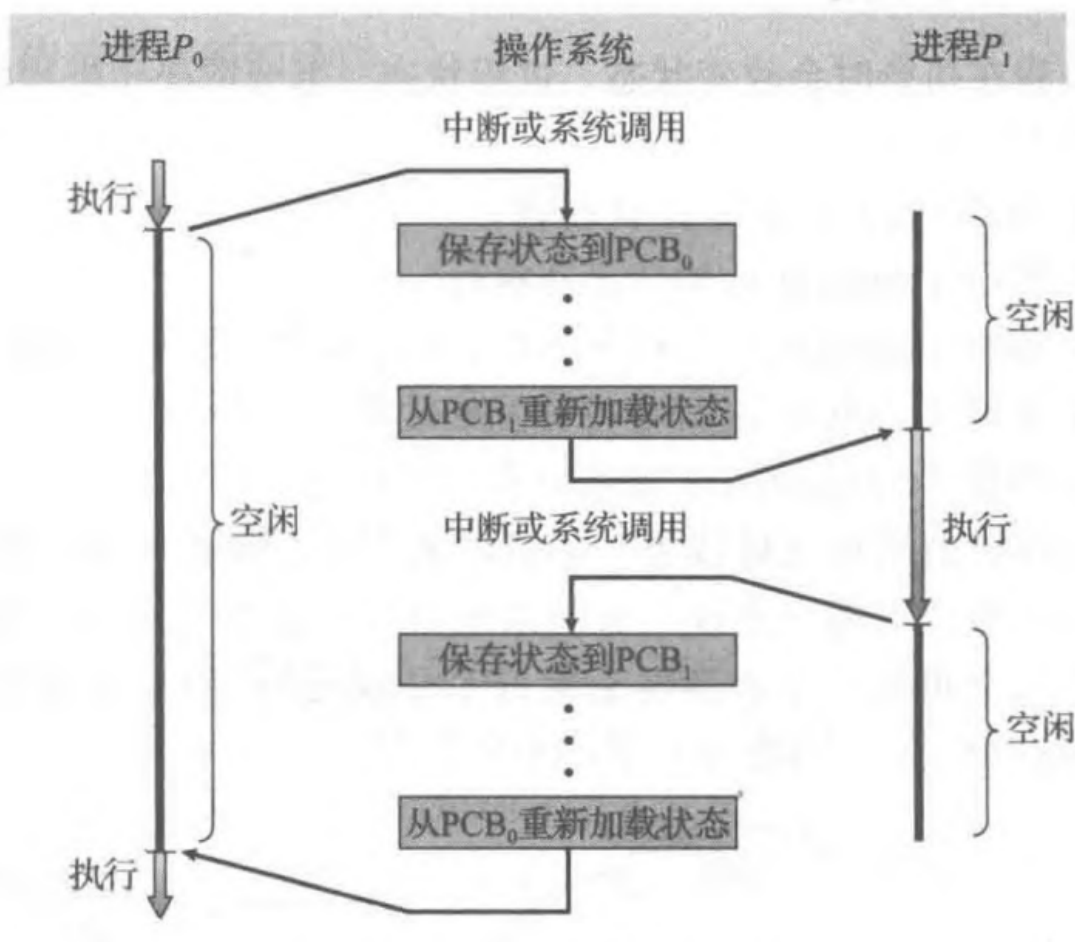
概念：进程的阻塞和唤醒

- 实现：进程的阻塞
 - 概念：
 - 运行态 $\rightarrow$ 阻塞态
 - 阻塞原语：
 - 找到要阻塞的进程对应的 `PCB`
 - 保护进程运行现场，将 `PCB` 状态信息设置为 **阻塞态**，暂时停止进程运行
 - 将 `PCB` 插入相应事件的等待队列；
 - 引起进程阻塞的事件：
 - 需要等待系统分配某种资源
 - 需要等待相互合作的其他进程完成工作
- 实现：进程的唤醒
 - 概念：
 - 阻塞态 $\rightarrow$ 就绪态

- 注意**唤醒只能是从阻塞变为就绪**，而不是就绪变为执行或者阻塞变为执行；
- 唤醒原语：
 - 在该事件的等待队列中找到 PCB；
 - 将 PCB 从等待队列移除，设置进程为就绪态
 - 将 PCB 插入就绪队列，等待被调度
- 引起进程唤醒的事件：
 - 等待的事件发生；
 - 真题：2019年
 - 比如：IO 结束、某进程退出临界区；
- 易错点：当前进程的时间片用完，必会引起唤醒事件；
 - 这是错误的。
 - 因为进程时间片用完后，会从执行态变成就绪态。而唤醒是要求从阻塞态变为执行态；
- 注意：阻塞和唤醒需要成对存在
 - 应当注意，Block 原语和 Wakeup 原语是一对作用刚好相反的原语，必须成对使用；
 - 如果在某个进程中调用了 Block 原语，则必须在与之合作的或其他相关的进程中安排一条相应的 Wakeup 原语，以便唤醒阻塞进程；
 - 否则，阻塞进程将因不能被唤醒而永久地处于阻塞态；

概念：进程的切换

- 概述
 - 在 PCB 保存当前进程作为状态机时的状态（状态=指令的执行+寄存器），将 PCB 移入对应队列后、加载另外一个进程的 PCB 状态；
- 实现
 - (1) 将运行环境信息（上下文环境）存入 PCB
 - (2) 将 PCB 移入相应队列
 - (3) 选择另一个进程执行，并更新其 PCB
 - (4) 根据 PCB，恢复新进程所需的运行环境；
- 图示：进程间的 CPU 切换



进程的优先级

概念：进程的优先级

- 概念：优先级与就绪队列
 - 概念：
 - 就绪队列未必只有一个，可以按照不同优先级来组织。另外，也可以把优先级高的进程排在更靠近对头的位置；
 - 分类：
 - 根据优先级是否可以动态改变分类，分成两种；
 - 静态优先级
 - 创建进程时确定，之后一直不变；
 - 动态优先级
 - 创建进程时有一个初始值，之后会根据具体情况动态地调整优先级；
- 概念：进程的优先级判定
 - 1. 系统进程的优先级高于用户进程；
 - 2. **前台进程**的优先级高于后台进程；
 - 3. 操作系统**更偏好 I/O 型进程**
 - 或称 **I/O 繁忙型进程**，**I/O** 设备可以和 **CPU** 并行运行，所以**优先 I/O 可以提升资源利用率**；
 - 与 **I/O** 型进程相对的是计算型进程（或 **CPU** 繁忙型进程）；
 - 在两个进程一个 **CPU** 繁忙、一个 **I/O** 繁忙时，**I/O** 繁忙的优先级通常更高；

- 真题：2013年

【2013 统考真题】某系统正在执行三个进程 P_1 , P_2 和 P_3 , 各进程的计算 (CPU) 时间和 I/O 时间比例如下表所示。

进程	计算时间	I/O 时间
P_1	90%	10%
P_2	50%	50%
P_3	15%	85%

- 为提高系统资源利用率, 合理的进程优先级设置应为 ()。
- 优先级为 $P_3 > P_2 > P_1$, 因为 P_3 进程的 IO 占比最大, 先开始可以提高并发度;

- 概念: 进程优先级的动态调整

- 分析:

- 可以从追求公平, 提升资源利用率等角度考虑;

- 提升优先级:

- 如果某进程在就绪队列中等待了很长时间, 则可以适当提升其优先级
- 如果发现一个进程频繁地进行 I/O 操作, 则可适当提升其优先级

- 降低优先级:

- 真题: 2010年

- 在进程的**时间片用完**后, 进程由运行态变为就绪态, 重新等待分配处理机, 此时应该降低其优先级, 保证其他就绪进程使用处理机;
- 如果某进程占用处理机运行了很长时间, 则可适当降低其优先级;

- 图示:

- 就是进程的优先级: NI 的数字越大, 就证明这个人约好, 就越愿意把 CPU 的资源让给别人; 当 NI 的值非常低, 只要 CPU 有空就抢, 即最自私的策略;
- 如果有一个 NI 值极低的需要执行, 则此时其他进程无法执行, 必须得等这个 NI 值低的进程执行完之后才可执行;

优先级

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
1	root	20	0	167504	11564	8372	S	0.0	0.1	0:02.35	systemd
2	root	20	0	0	0	0	S	0.0	0.0	0:00.02	kthreadd
3	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	rcu_gp
4	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	rcu_par+
6	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	kworker+
9	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	mm_perc+
10	root	20	0	0	0	0	S	0.0	0.0	0:03.72	ksoftir+
11	root	20	0	0	0	0	I	0.0	0.0	0:39.13	rcu_sch+
12	root	rt	0	0	0	0	S	0.0	0.0	0:01.08	migrati+
13	root	-51	0	0	0	0	S	0.0	0.0	0:00.00	idle_in+
14	root	20	0	0	0	0	S	0.0	0.0	0:00.00	cpuhp/0

进程通信

简述: 进程间通信

- 别名:

- Inter-Process Communication, IPC

- 概念:

- 是指两个进程之间产生数据交互，指进程之间的信息交换;

- 原因:

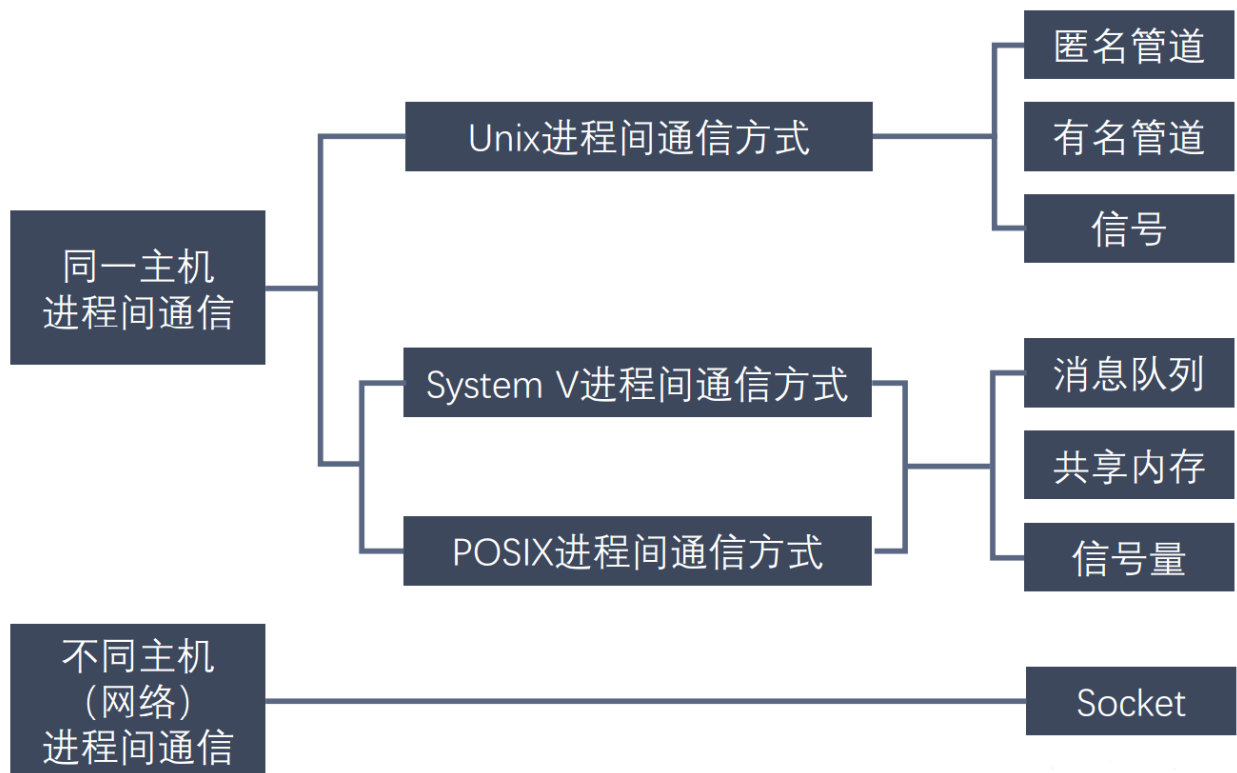
- 进程是分配系统资源的单位（包括内存地址空间），因此各进程拥有的内存地址空间相互独立;
- 因此为了保证安全，一个进程不能直接访问另一个进程的地址空间;

- 方式:

- 1. 共享存储;
- 2. 消息传递;
- 3. 管道通信;

面试题：进程间通信有哪些方式？

- 进程间通信的方式有 6 种，管道、信号、消息队列、共享内存、信号量和套接字。



- 管道:

- 管道可以理解成不同进程之间的传话筒，一方发声，一方接收，声音的介质可以是空气或者电缆;
- **进程间的管道就是内核中的一串缓存**，从管道的一端写入数据，另一端读取。数据只能单向流动，遵循先进先出（FIFO）的原则;
- **匿名管道**：允许具有亲缘关系的进程（如父子进程）进行通信
- **命名管道**：允许无亲缘关系的进程通信，通过在文件系统中创建一个特殊类型的文件来实现;

- 信号:

- 信号可以理解成以前的 BB 机，用于通知接收进程某件事情发生了，是一种较为简单的通信方式，主要用于处理异步事件。

- Linux 中常用的信号：
 - **SIGHUP**：当我们退出终端 (Terminal) 时，由该终端启动的所有进程都会接收到这个信号，默认动作为终止进程。
 - **SIGINT**：程序终止 (interrupt) 信号。按 **Ctrl+C** 时发出，大家应该在操作终端时有过这种操作。
 - **SIGQUIT**：和 SIGINT 类似，按 **Ctrl+** 键将发出该信号。它会产生核心转储文件，将内存映像和程序运行时的状态记录下来。
 - **SIGKILL**：强制杀死进程，本信号不能被阻塞和忽略。
 - **SIGTERM**：与 SIGKILL 不同的是该信号可以被阻塞和处理。通常用来要求程序自己正常退出；

- 共享内存：

- 允许两个或多个进程共享一个给定的内存区，一个进程写入的东西，其他进程马上就能看到；
- 共享内存是最快的进程间通信方式，它是针对其他进程间通信方式运行效率低而专门设计的；

- 信号量：

- 它常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。
- 控制信号量的方式有两种原子操作：
 - 一个是 **P 操作** (wait, 减操作)，当进程希望获取资源时，它会执行 P 操作。如果信号量的值大于 0，表示有资源可用，信号量的值减 1，进程继续执行。如果信号量的值为 0，表示没有可用资源，进程进入等待状态，直到信号量的值变为大于 0。
 - 另一个是 **V 操作** (signal, 加操作)，当进程释放资源时，它会执行 V 操作，信号量的值加 1。如果有其他进程因为等待该资源而被阻塞，这时会唤醒其中一个进程。

- 套接字：

- 提供网络通信的端点，可以让不同机器上运行的进程之间进行双向通信。

补充：代码示例 - 使用 C 语言在 **Unix/Linux** 环境下通过匿名管道实现两个进程（通常是父子进程）之间通信的示例

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int main() {
    int pipefd[2];
    pid_t cpid;
    char buf;

    // 创建管道
    if (pipe(pipefd) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }
}
```

```

// 创建子进程
cpid = fork();
if (cpid == -1) {
    perror("fork");
    exit(EXIT_FAILURE);
}

if (cpid == 0) {    /* 子进程 */
    close(pipefd[1]); // 关闭写端

    // 从管道读取数据
    while (read(pipefd[0], &buf, 1) > 0)
        write(STDOUT_FILENO, &buf, 1);

    write(STDOUT_FILENO, "\n", 1);
    close(pipefd[0]);
    exit(EXIT_SUCCESS);
} else {           /* 父进程 */
    close(pipefd[0]); // 关闭读端

    // 向管道写入数据
    write(pipefd[1], "Hello, Child!", 13);
    close(pipefd[1]); // 关闭写端，触发EOF
    wait(NULL);       // 等待子进程退出
    exit(EXIT_SUCCESS);
}
}

```

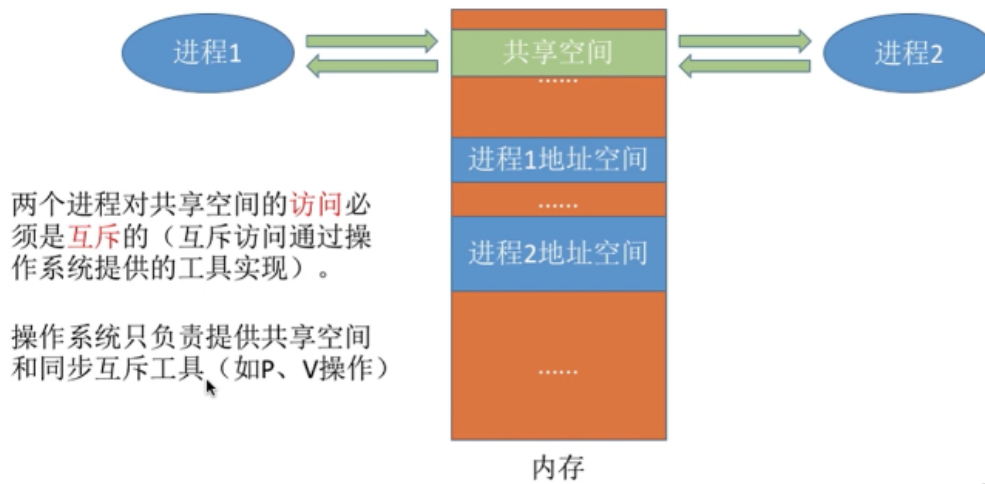
共享存储

概念：基于存储区的共享

- 概念：
 - 基于数据结构的共享；
 - 共享空间可以被多个进程进行使用；
 - 比如 Linux 中的 `shmap_open()` 接口可以用于访问共享存储区；
- 特点：
 - 1. 共享存储中的访问需要时互斥的（比如用 PV 操作、上锁）；
 - 2. 共享存储区中的存放位置由进程控制，而不是操作系统；
 - 3. 共享内存**速度通常快于消息传递**；
 - 这是因为消息传递的实现需要**频繁采用系统调用**，因此需要消耗更多时间以便内核介入；
 - 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；
- 补充：
 - 对具有多个处理核系统的最新研究表明：在这类系统上，消息传递的性能要优于共享内存。共享内存会有高速缓存一致性问题，这是由共享数据在多个高速缓存之间迁移而引起

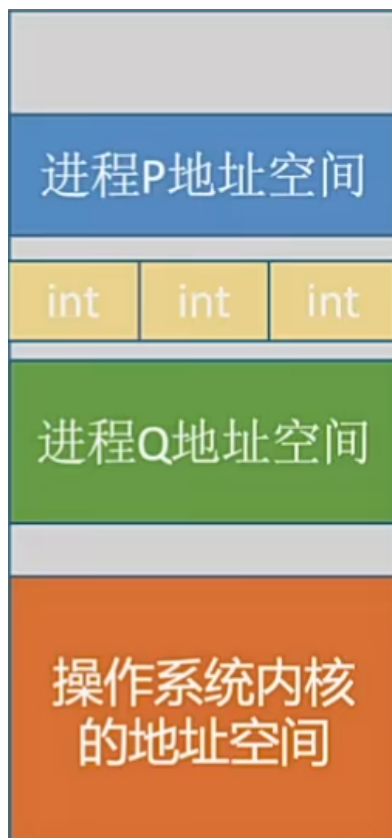
的。随着系统的处理核的数量的日益增加，可能导致消息传递作为 **IPC** 的首选机制；

- 图示：



概念：基于数据结构的共享

- 概念：
 - 比如共享空间里只能放一个长度为 **10** 的数组；
- 特点：
 - 这种共享方式速度慢、限制多，是一种**低级通信**方式；
- 图示：

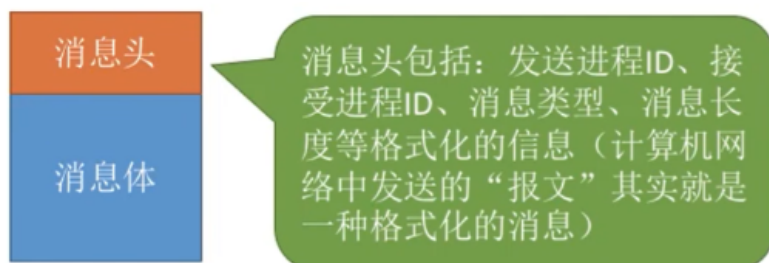


消息传递

概念：消息

- 概念：
 - 进程间的数据交换以格式化的消息（**Message**）为单位；

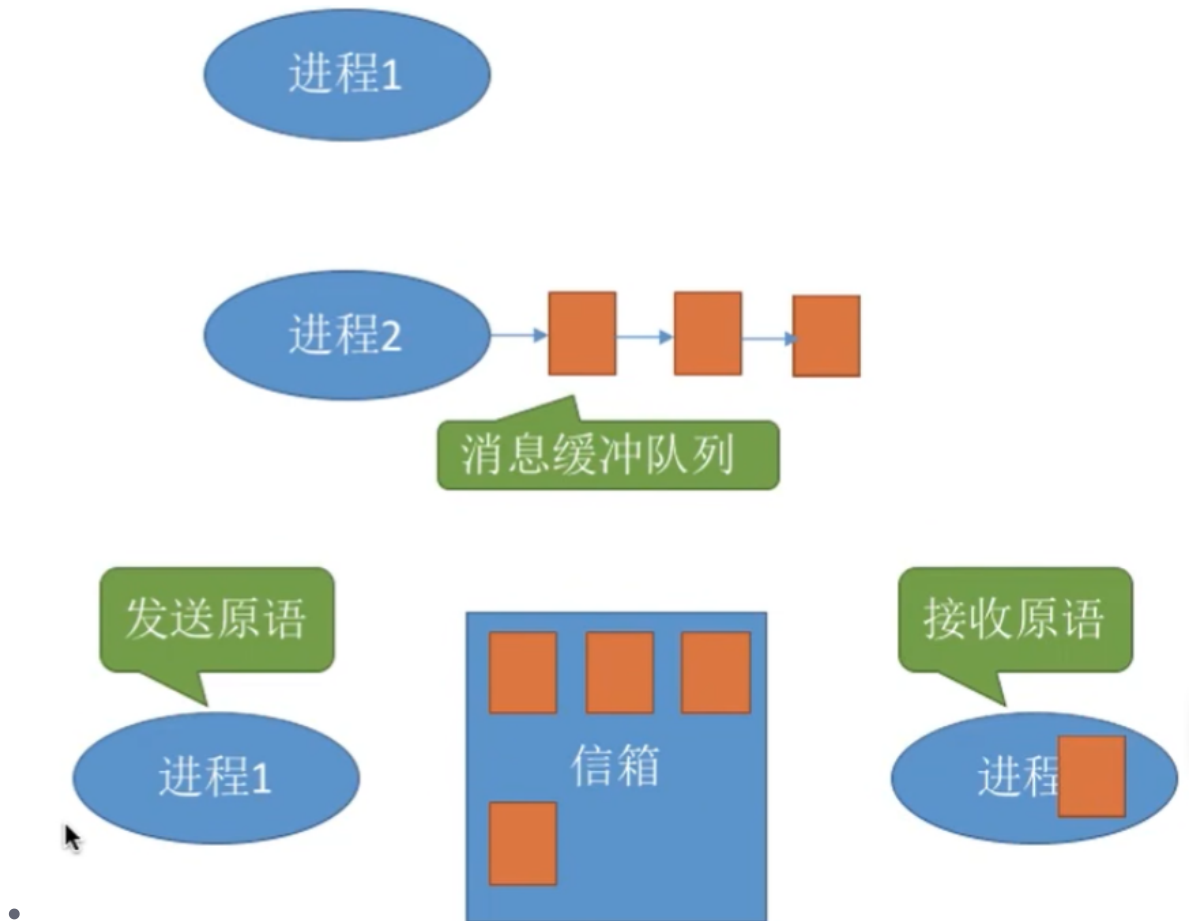
- 进程通过操作系统提供的**发送消息/接受消息**两个原语进行数据交换；
- 特点：
 - 消息传递对于交换较少数量的数据很有用，因为无需避免冲突；
 - 对于分布式系统，消息传递比共享内存更易实现；
- 原语：
 - 发送：`send(Q,msg);`
 - 接受：`receive(P,&msg);`
- 图示：什么是消息



概念：两种消息传递的方式

- 方式一：直接通信方式
 - 概念：
 - 发送进程需要指明接受进程的 ID，消息直接挂到接收进程的**消息缓冲队列**上；
 - 原语：
 - `send (P, message)`：向进程 P 发送 message；
 - `receive (Q, message)`：从进程 Q 接收 message；
 - 方式：
 - 1. 发送方进程 P 使用 `send` 原语将消息发送到内核当中的进程 Q 的消息队列上；
 - 2. 从消息队列中根据进程 Q 的 PCB，匹配到进程 Q 的地址，调用接受原语 `receive`；
 - 3. 进程 Q 接收到内核的消息；
 - 缺点：硬编码
 - 更改进程的标识符可能需要分析所有其他进程定义；
 - 所有旧的标识符的引用都应找到，以便修改成为新标识符；
- 方式二：间接通信方式
 - 概念：
 - 消息要先发送到中间实体（信箱）中，因此也称**信箱通信方式**；
 - 原语：
 - `send (A, message)`：向邮箱 A 发送 message；
 - `receive (A, message)`：从邮箱 A 接收 message；
 - 方式：
 - 1. 进程 P 申请一个 A 邮箱和 B 邮箱；
 - 2. 进程 P 在自己的地址空间中实例化信箱内容；
 - 3. 使用发送原语，指定 `send` 要发送的信箱 A；

- 4. 进程 Q 使用接收原语，接受信箱 A 的信息；
- 图示：



管道通信

概念：管道

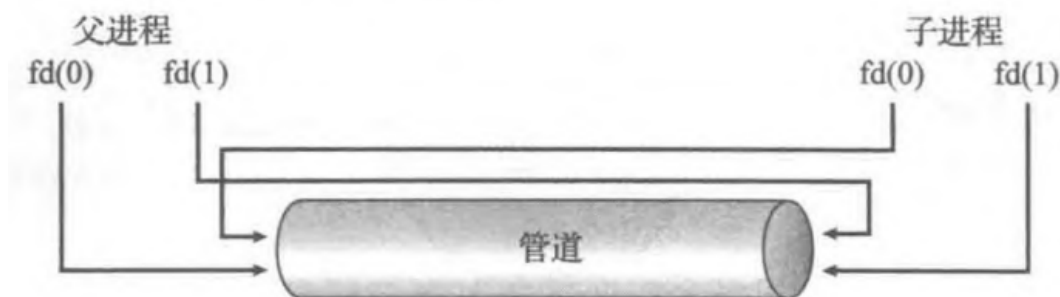
- 概念：
 - 用于连接读写进程的一个共享文件，又名 `pipe` 文件；
 - 数据在管道中是先进先出的，管道通信允许两个进程按 `生产者-消费者` 方式进行通信；
 - 只要管道不满，写进程就能向管道的一端写入数据；
 - 只要管道非空，读进程就能从管道的一端读出数据；
 - 管道是早期 `UNIX` 系统最早使用的一种 `IPC` 机制；
- 要求：
 - 为了协调双方的通信，管道机制必须提供三方面的协调能力；
 - (1) 互斥
 - 指当一个进程对管道进行读/写操作时，其他进程必须等待；
 - 真题：2014年
 - 可以有多个读进程或写进程对管道进行操作，但是同一时刻只能有一个进程操作；
 - 同样的，不可以同时两个进程对管道操作，即使一个是读进程、一个是写进程也不行；
 - (2) 同步

- 指写进程向管道写入一定数量的数据后，写进程阻塞，直到读进程取走数据后再将它唤醒；
- 读进程将管道中的数据取空后，读进程阻塞，直到写进程将数据写入管道后才将其唤醒；
- (3) 确定对方的存在
- 特点：
 - (1) 两个进程只能采用**半双工通信**，某一时间段内只能实现单向的传输。如果要实现双向同时通信，则需要设置两个管道；
 - 注意管道是半双工，不是单工通信。单工通信是只能单方向传输，而半双工是方向任意，但同一时间只能一个进程使用；
 - (2) 限制管道的大小：
 - 管道文件是一个固定大小的缓冲区，在 `Linux` 中该缓冲区的大小为 `4KB`，这使得它的大小不像普通文件那样不加检验地增长；
 - 使用单个固定缓冲区也会带来问题，比如在写管道时可能变满，这种情况发生时，随后对管道的写调用将默认地被阻塞，等待某些数据被读取，以便腾出足够的空间供 `write()` 调用；
 - (3) 数据以**字符流**的形式写入管道，当管道写满时，写进程的 `write()` 系统调用将被阻塞，等待读进程将数据取走。当读进程将数据全部取走后，管道变空，此时读进程的 `read()` 系统调用将被阻塞；
- 实现：
 - `UNIX` 下的普通管道创建： `pipe(int fd[])`
 - 通过文件描述符 `int fd[]` 来访问：
 - `fd[0]` 为管道的读出端；
 - `fd[1]` 为管道的写入端；
 - `UNIX` 将管道作为一种特殊类型的文件，因此访问管道可以采用普通的系统调用 `read()` 和 `write()`

- 图示：



- 管道的访问图示：



线程

线程基本概念

概念：线程的基本概念

- 原因：
 - 引入进程的目的是更好地使多道程序并执行，提高资源利用率和系统吞吐量；
 - 而引入线程（Threads）的目的则是减小程序在并发执行时所付出的**时空开销**，提高操作系统的并发性能；
- 概念：什么是线程
 - 线程最直接的理解就是轻量级进程，它是一个基本的 CPU 执行单元，也是程序执行流的最小单元，线程是一个轻型实体，它不拥有系统资源，但每个线程都应有一个唯一的标识符和一个线程控制块，线程控制块记录线程执行的寄存器和栈等现场状态；
 - 线程是进程中的一个实体，是被系统独立调度和分派的基本单位；
 - 线程自己**不拥有**系统资源，只拥有在运行中必不可少的资源，但它可与同属一个进程的其他线程共享进程所拥有的全部资源；
 - 真题：2012年
 - 进程之间拥有相互独立的地址空间，但同一进程的各个线程间共享一个地址空间；
 - 一个线程可以创建和撤销另一个线程，同一进程中的多个线程之间可以并发执行；
 - 由于线程之间的相互制约，致使线程在运行中呈现出间断性；
- 真题：2024年
 - 题目：

28. 若进程 P 中的线程 T 先打开文件，得到文件描述符 fd，再创建两个线程 Ta 和 Tb，则下列资源中，Ta 与 Tb 可共享的是()。

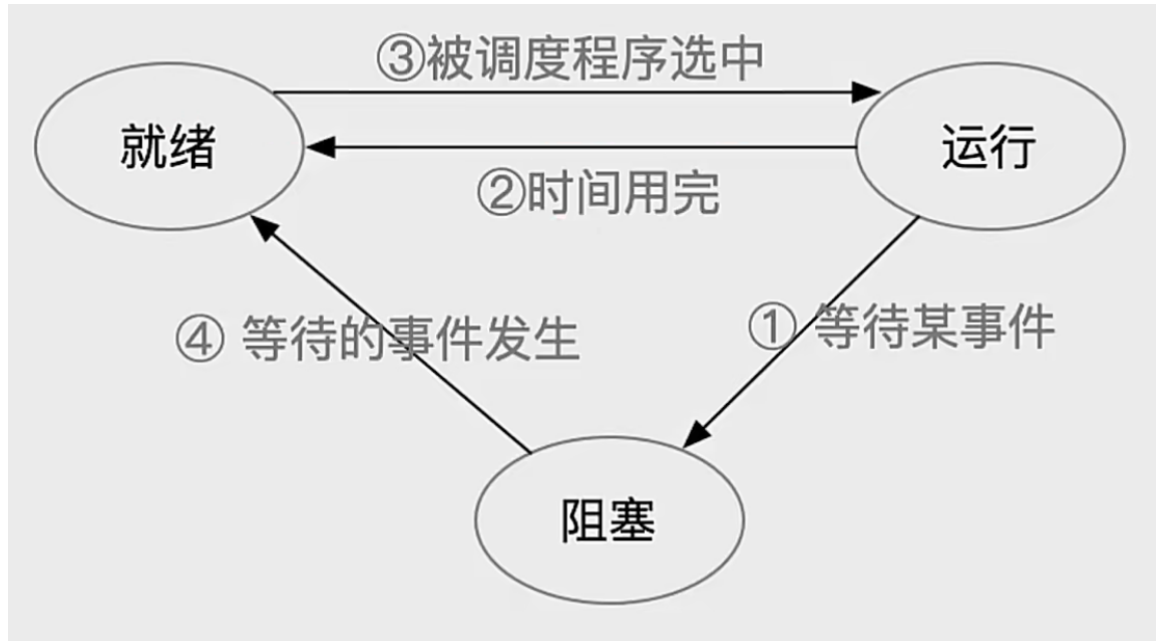
I. 进程 P 的地址空间	II. 线程 T 的栈	III. 文件描述符 fd	
A. 仅 I	B. 仅 I、III	C. 仅 II、III	D. I、II、III
 - 答案
 - B
 - 分析：
 - 每个线程都有自己的栈，这属于线程的私有资源，因此不可以共享；
 - 进程是资源分配的基本单位，而属于同一进程的不同线程共享进程的物理地址空间和进程的资源，而文件描述符属于进程的资源。当打开文件时，文件描述符返回给进程，并且应该是属于进程 P 的用户打开文件表，因此 Ta 和 Tb 可以共享使用此文件描述符；

线程状态

概念：线程的状态

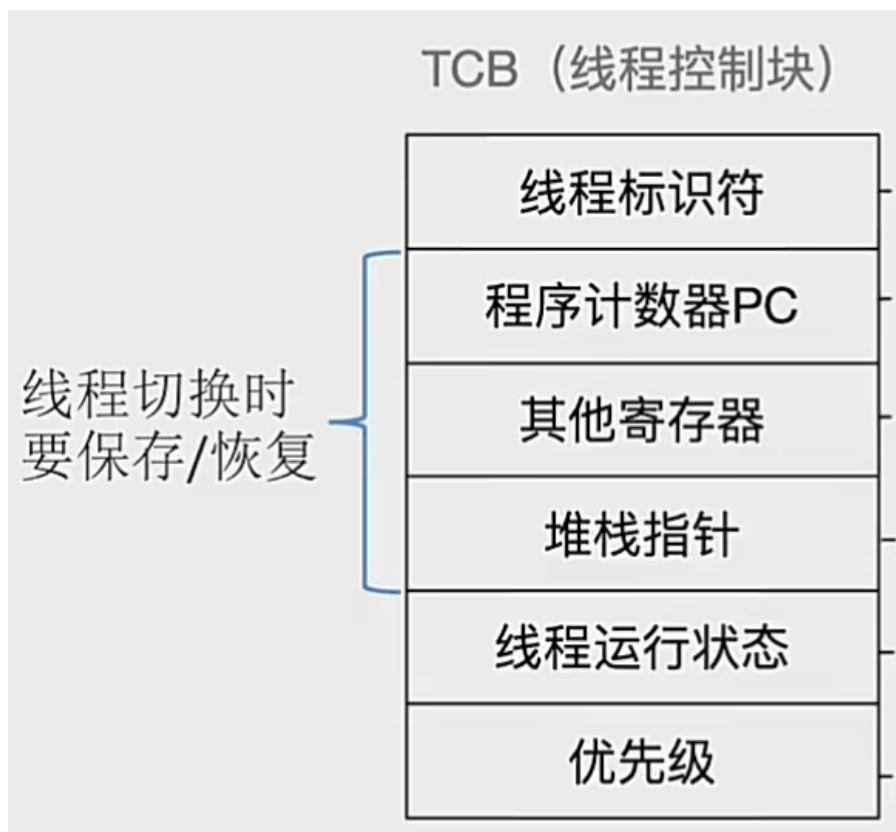
- 概念：线程的状态
 - 概念：
 - 线程也有就绪、阻塞和运行三种基本状态；
 - 执行态：线程已获得 CPU 而正在运行；
 - 就绪态：线程已具备各种执行条件，只需再获得 CPU 便可立即执行；

- 阻塞态：线程在执行中因某事件受阻而处于暂停状态；
- 线程这三种基本状态之间的转换和进程基本状态之间的转换是一样的；
- 图示：



- 概念：线程控制块 TCB

- 概念：
 - 与进程类似，系统也为每个线程配置一个线程控制块 TCB，用于记录控制和管理线程的信息；
 - 线程控制块通常包括：
 - 1. 线程标识符；
 - 2. 一组寄存器，包括：程序计数器、状态寄存器、通用寄存器；
 - 3. 线程运行状态，用于描述线程正处于何种状态；
 - 4. 优先级；
 - 5. 线程专有存储区，线程切换时用于保存现场等；
 - 6. 堆栈指针，用于过程调用时保存局部变量及返回地址等；
 - 同一进程中的所有线程都完全共享进程的地址空间和全局变量。各个线程都可以访问进程地址空间的每个单元，所以一个线程可以读、写或甚至清除另一个线程的堆栈。
- 图示：

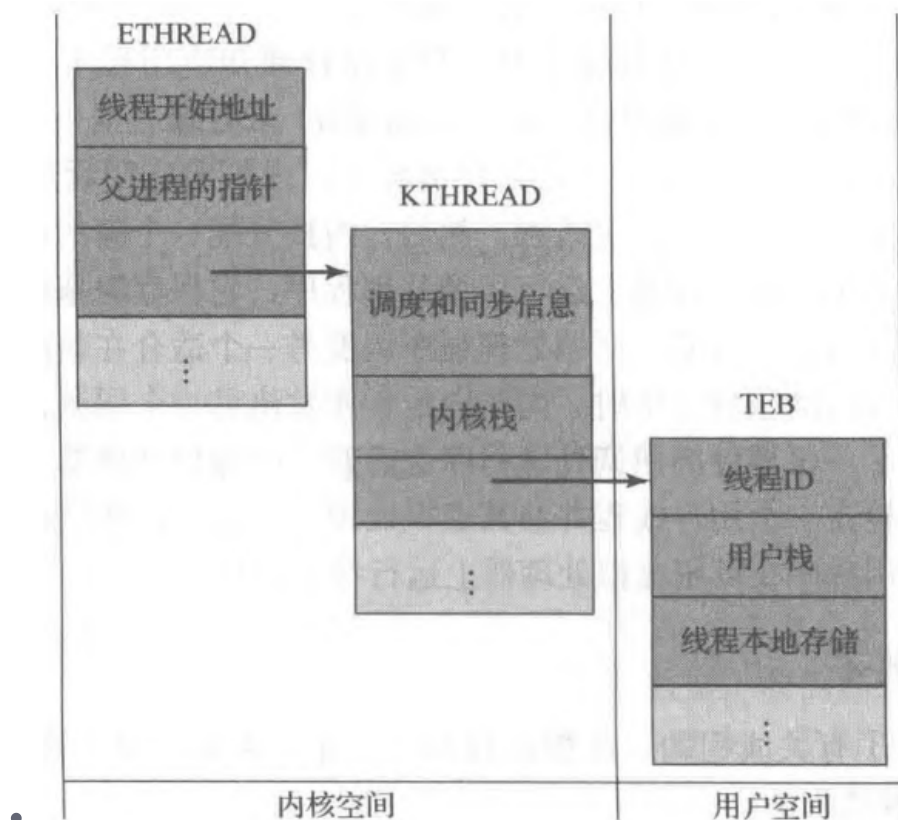


- 特点:

- (1) 线程作为调度的基本单位
 - 传统进程机制中，进程是资源分配、调度的基本单位；
 - 引入线程后，**进程是资源分配的基本单位，线程是调度的基本单位**；
 - 在同一个进程内、切换线程，不会引起进程切换；
 - 在不同进程间、切换线程，将会导致进程切换；
- (2) 并发性增加
 - 传统进程机制中，只能进程间并发
 - 引入线程后，各线程间也能并发，提升了并发度
- (3) 系统开销减少
 - 传统的进程间并发，需要切换进程的运行环境，系统开销很大
 - 线程间并发，如果是同一进程内的线程切换，则不需要切换进程环境。系统开销小
 - 引入线程后，并发所带来系统开销减小
- (4) 阻塞性
 - 如果一个交互程序采用多线程，那么即使部分阻塞或者执行冗长操作，它仍可以继续执行，从而增加对用户的响应程度；
 - 这对于用户界面设计尤其有用；
- (5) 支持多处理器系统
 - 对于传统单线程进程，不管有多少个 CPU，进程只能运行在一个 CPU 上；
 - 对于多线程进程，可将进程中的多个线程分配到多个 CPU 上执行；
- (6) 线程与程序
 - 不同的线程可以执行相同的程序，即同一个服务程序被不同的用户调用时，操作系统将它们创建成不同的线程；

- 补充: Windows 线程

- 基本概念：
 - **ETHREAD** 和 **KTHREAD** 完全位于内核空间：这意味着只有内核可以访问它们；
 - **TEB** 是个用户空间数据结构，以供线程在用户模式下运行时访问；
 - **TEB** 除了包括许多其他域外，还包括线程标识符、用户模式堆栈以及用于线程本地存储的数组等；
- 图示：



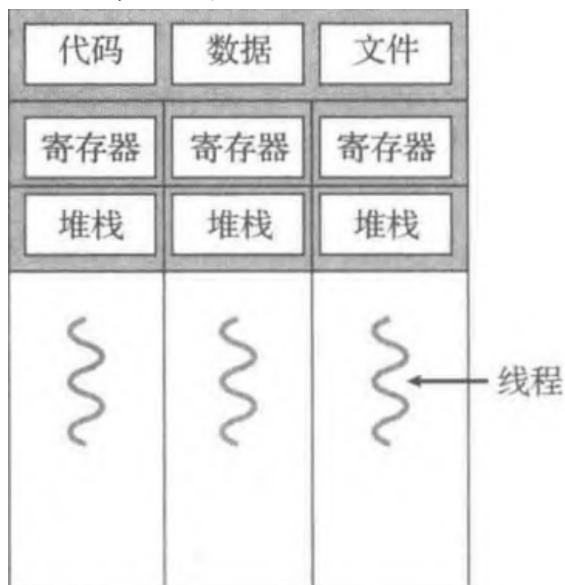
进程和线程的区别与关系

概念：进程和线程的区别

- (1) 进程是系统进行资源分配调度的基本单位，线程是 **CPU** 调度分派的基本单位；
- (2) 线程依赖于进程而存在，一个进程至少有一个线程；
- (3) 进程有自己的独立地址空间，线程共享所属进程的地址空间；
- (4) 进程是拥有系统资源的一个独立单位，而线程自己基本上不拥有系统资源，只拥有一点在运行中必不可少的资源，如程序计数器，一组寄存器和栈，和其他线程共享本进程的相关资源如内存等；
- (5) 在进程切换时，涉及到整个当前进程 **CPU** 环境的保存环境的设置，以及新被调度运行的 **CPU** 环境的设置，而线程切换只需保存和设置少量的寄存器的内容，并不涉及存储器管理方面的操作，可见进程切换的开销远大于线程切换的开销；
- (6) 线程之间的通信更方便，同一进程下的线程共享全局变量等数据，而进程之间的通信需要以进程间通信（**IPC**）的方式进行；
- (7) 多线程程序只要有一个线程崩溃，整个程序就崩溃了，但多进程程序中一个进程崩溃并不会对其它进程造成影响，因为进程有自己的独立地址空间，因此多进程更加健壮；

概念：多线程与进程的关系

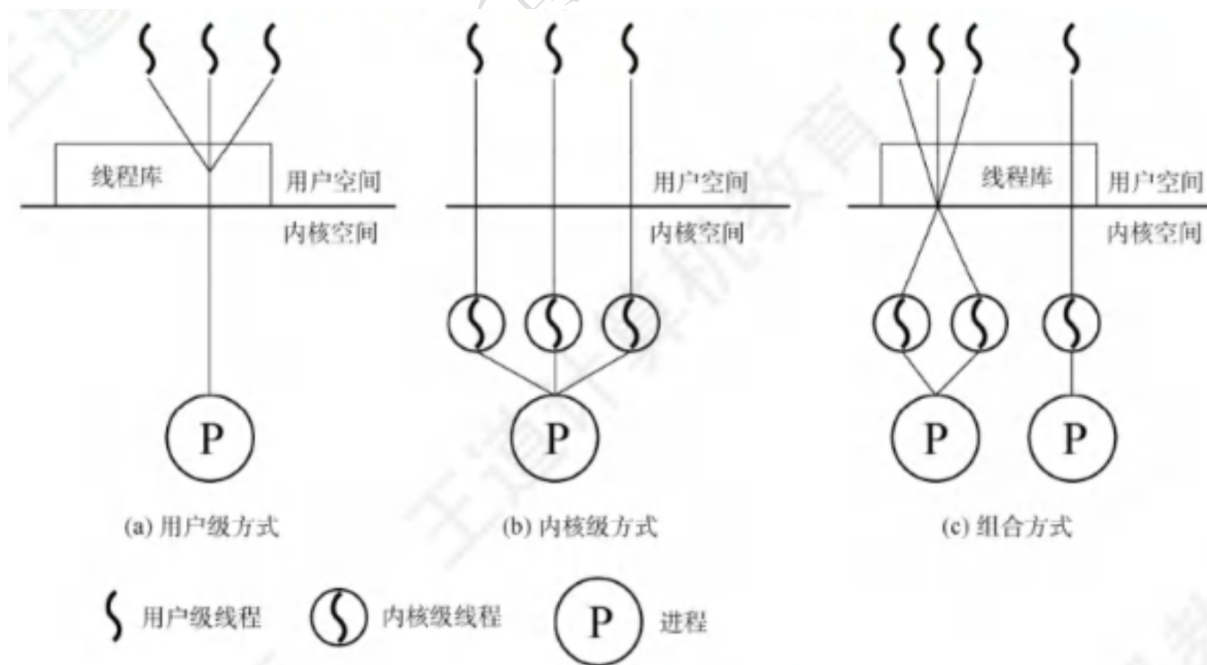
- 共享部分：
 - 它与同一进程的其他线程共享**代码段、数据段和其他操作系统资源**；
- 单独部分：
 - 包括**线程 ID、程序计数器、寄存器组和堆栈**；
- 图示：进程中的多个线程



线程实现方式

概述：三种线程实现方式

- 图示：



概念：用户级线程

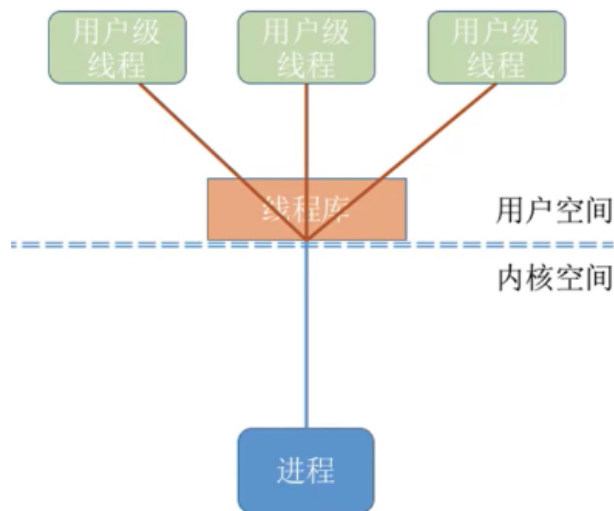
- 概念：
 - 由应用程序通过线程库实现，所有线程管理工作都由应用程序在用户态内负责，包括线程切换、创建、撤销等；
 - 操作系统对于线程一无所知，只是面向进程进行调度与分配；

- **内核的事件不会影响到用户态线程的切换**，比如异常处理、IO 请求、系统调用等；
- **真题：** 2019年
- 用户级线程的管理依靠进程，只需要引入线程库即可支持，不需要操作系统内核的支持，因此即使 OS 不支持内核级线程，用户级线程也依然可以使用；

- 特点：

- (1) 用户级线程中，线程切换可以在用户态下即可完成，无需操作系统干预；
- (2) 在用户视角看来、系统中存在多个线程，线程的切换工作由进程进行。但是在操作系统内核看来，并意识不到线程的存在，操作系统只能感知到进程；

- 图示：



- 特点：

- 优点：

- (1) 用户级线程的切换在用户空间即可完成，不需要切换到核心态，线程管理的系统开销小，效率高；
- (2) 调度算法可以是进程专用的，不同的进程可根据自身的需要，对自己的线程选择不同的调度算法；
- (3) 用户级线程的实现与操作系统平台无关，对线程管理的代码是属于用户程序的一部分；

- 缺点：

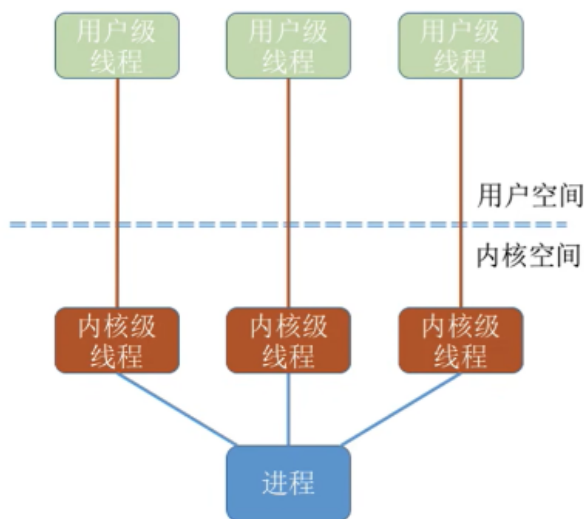
- (1) 如果其中的一个线程被阻塞，整个进程都会被阻塞 -> **进程的并发度不高**；
- (2) 不能发挥多 CPU 的优势，内核每次分配给一个进程的仅有一个 CPU，因此进程中仅有一个线程能执行；

概念：内核级线程

- 概念：

- 其线程的管理工作由操作系统内核完成，线程的切换都需要操作系统接入；
- 内核级线程的切换需要在核心态下才能完成，操作系统也为每个内核级线程设置一个线程控制块 TCB，内核根据该控制块感知某线程的存在，并对其加以控制；

- 图示：



- 优缺点：
 - 优点：当一个线程被阻塞后，别的线程还可以继续执行，并发能力强。**多线程可在多核处理机上并行执行**；
 - 缺点：一个用户进程会占用多个内核级线程，线程的切换由操作系统内核完成，需要切换到核心态，因此线程管理的成本高，开销大；

概念：组合方式

- 概念：
 - 有些系统使用组合方式的多线程实现；
 - 在组合实现方式中，内核支持多个**内核级线程**的建立、调度和管理，同时允许用户程序建立、调度和管理用户级线程；
 - 一些内核级线程对应多个用户级线程，这是用户级线程通过时分多路复用内核级线程实现的；
- 特点：
 - 同一进程中的多个线程可以同时多 CPU 上并行执行，且在阻塞一个线程时不需要将整个进程阻塞，所以组合方式能结合 KLT 和 ULT 的优点，并且克服各自的不足；

线程池

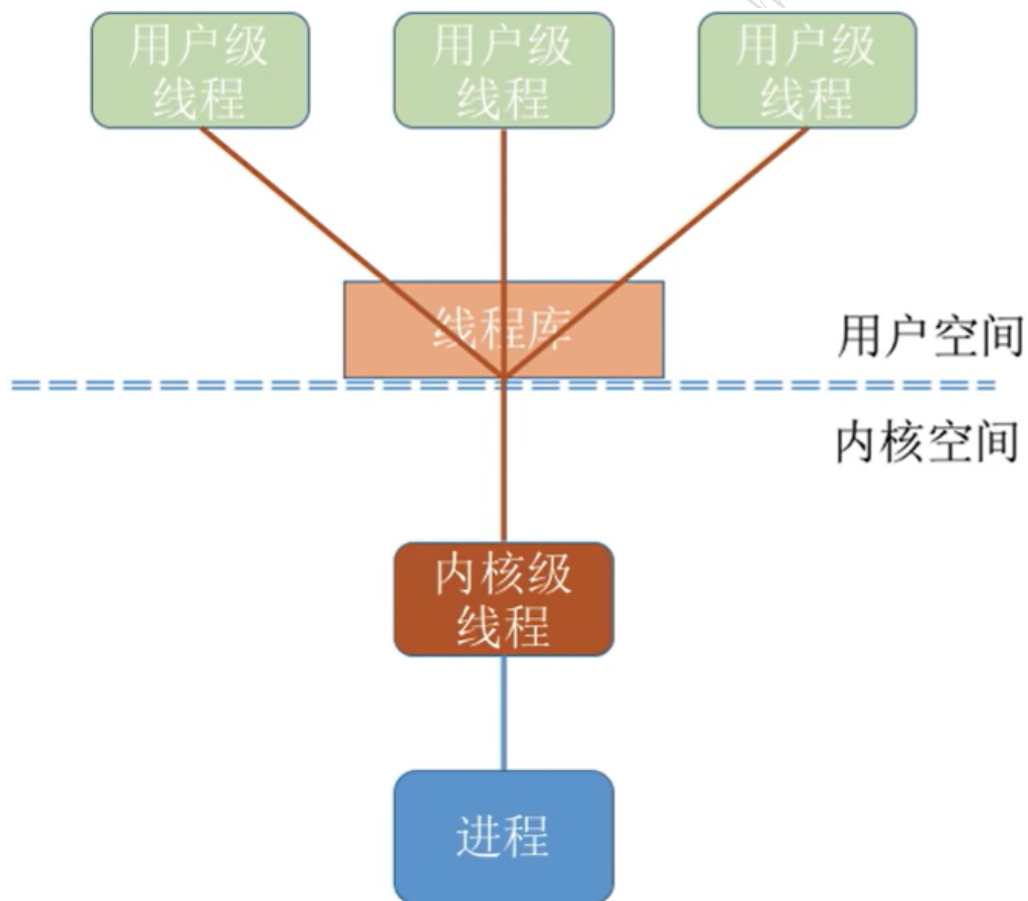
概念：线程池

- 概念：
 - 在进程开始时创建一定数量的线程，并加到池中以等待工作；
 - 当服务器收到请求时，它会唤醒池内的一个线程 (如果有可用线程)，并将需要服务的请求传递给它；
 - 一旦线程完成了服务，它会返回到池中再等待工作。如果池内没有可用线程，那么服务器会等待，直到有空线程为止；
- 优点
 - 速度快：用现有线程服务请求比等待创建一个线程更快；
 - 限制性：线程池限制了任何时候可用线程的数量。这对那些不能支持大量并发线程的系统非常重要；
 - 不同策略：将要执行任务从创建任务的机制中分离出来，允许我们采用不同策略运行任务。例如，任务可以被安排在某一个时间延迟后执行，或定期执行；

多线程模型

概念：多对一模型

- 定义：
 - 多个用户及线程映射到一个内核级线程；
 - 每个用户进程只对应一个内核级线程。
 - 补充：Green threads 的多对一线程库为 Solaris 所采用；
- 特点
 - 优点：
 - 用户级线程的切换在用户空间即可完成，不需要切换到核心态，线程管理的系统开销小，效率高；
 - 缺点：
 - 1. 当一个用户级线程被阻塞后，整个进程都会被阻塞，**并发度不高**；
 - 2. 由于内核只能一次调度一个线程，所以**并未增加并发性**，多个线程不可在多核处理机上并行运行；
- 图示：

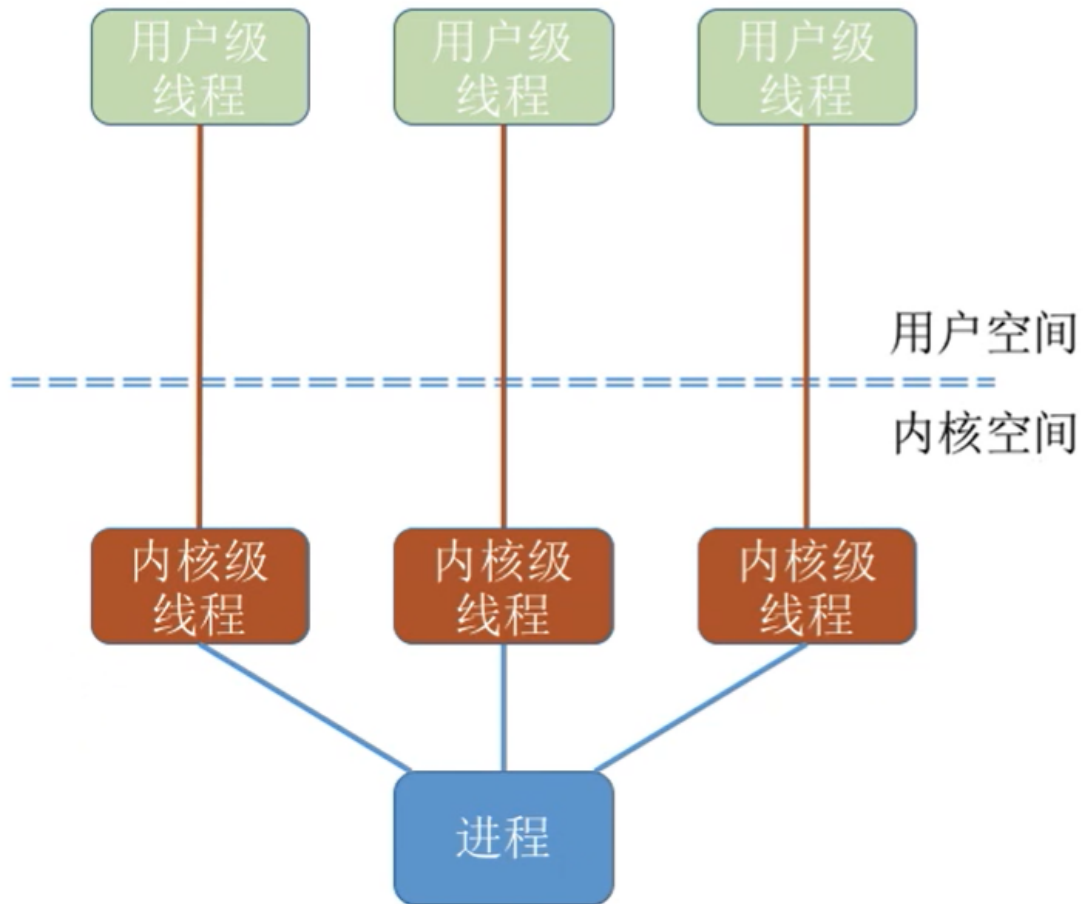


概念：一对一模型

- 定义
 - 一个用户及线程映射到一个内核级线程，每个用户进程有与用户级线程同数量的内核级线程；
- 特点
 - 优点：

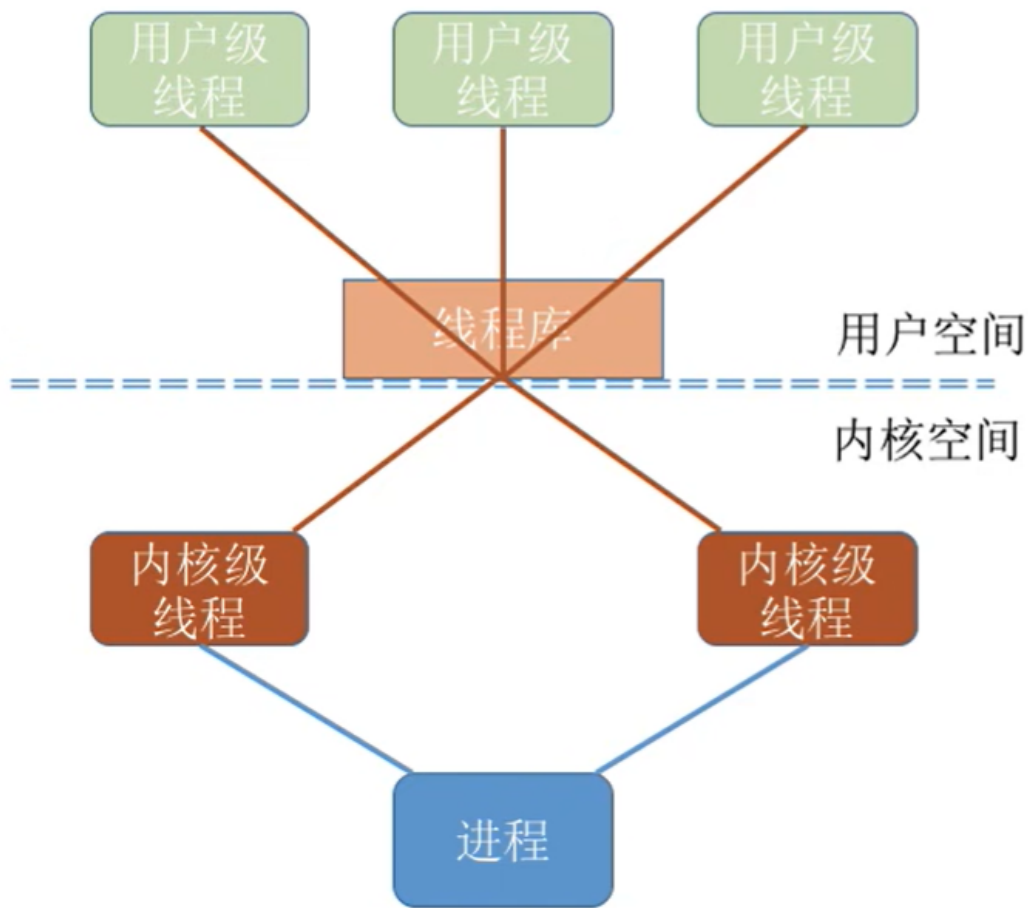
- 当一个线程被阻塞后，别的线程还可以继续执行，并发能力强
- 一对一模型提供了更大的并发性；
- 缺点：
 - 一个进程会占用多个内核级线程，线程切换由操作系统内核完成，需要切换到核心态，因此线程管理的成本高，开销大；

• 图示：



概念：多对多模型

- 概念
 - n 用户及线程映射到 m 个内核线程 ($n \geq m$)
 - 每个用户进程对应 m 个内核级线程；
- 特点
 - 克服了对一模型并发度不高的缺点，又克服了一对一模型中一个用户进程占用太多内核级线程，开销太大的缺点；
- 图示：多核 CPU 环境下，图中进程最多能被分配两个核；



面试题：线程间如何同步？

- 同步解决的是多线程操作共享资源的问题，不管线程之间是如何穿插执行的，最后的结果都是正确的；
- 在操作系统层面，保证线程同步的方式有很多，比如锁、信号量等。临界区是：对共享资源访问的程序片段，我们希望这段代码是 **互斥** 的，可以保证在某个时刻只能被一个线程执行，也就是说一个线程在临界区执行时，其它线程应该被阻止进入临界区；
- 临界区不仅针对线程，同样针对进程。同步的实现方式有：
 1. 互斥锁：
 - 使用加锁操作和解锁操作可以解决并发线程/进程的互斥问题。
 - 任何想进入临界区的线程，必须先执行加锁操作。若加锁操作顺利通过，则线程可进入临界区；在完成对临界资源的访问后再执行解锁操作，以释放该临界资源。
 - 加锁和解锁锁住的是什么呢？可以是 **临界区对象**，也可以只是一个简单的 **互斥量**，例如互斥量是 **0** 无锁，**1** 表示加锁。
 - 根据锁的实现不同，可以分为 **忙等待锁** 和 **无忙等待锁**。
 - 忙等待锁（也称为自旋锁，**Spinlock**）是指当一个线程试图获取锁时，如果该锁已经被其他线程持有，当前线程不会立即进入休眠或阻塞，而是不断地检查锁的状态，直到该锁可用为止。这个过程被称为忙等待（**busy waiting**），因为线程在等待锁时仍然占用 **CPU** 资源，处于活跃状态。优点是避免了线程的上下文切换。
 - 无忙等待锁是指当一个线程尝试获取锁时，如果锁已经被其他线程持有，当前线程不会忙等待，而是主动让出 **CPU**，进入阻塞状态或休眠状态，等待锁释放。当锁被释放时，线程被唤醒并重新尝试获取锁。这类锁的主要目的是避免忙等待带来的 **CPU** 资源浪费。
 2. 信号量：

- 信号量是操作系统提供了一种协调共享资源访问的方法。通常表示资源的数量，对应的变量是一个整型（sem）变量。
- 另外，还有两个原子操作的系统调用函数来控制信号量，分别是：
- P 操作：当线程想要进入临界区时，会尝试执行 P 操作。如果信号量的值大于 0，信号量值减 1，线程可以进入临界区；否则，线程会被阻塞，直到信号量大于 0。
- V 操作：当线程退出临界区时，执行 V 操作，信号量的值加 1，释放一个被阻塞的线程。

获取笔记持续更新可加微信：WandD2277