



# 第3章 进程线程模型

## ■ 本章内容：

- 1. 进程的概念，进程控制块，进程状态的转换，进程的地址空间
- 2. 进程的创建、撤销、阻塞和唤醒，进程控制命令和API实例
- 3. 线程的概念，线程的组成，线程与进程的关系
- 4. 线程的实现方式，pthread线程库的应用





# 为什么需要引入进程？

- 人们常常希望同时运行多个程序
- 如何提供有多个CPU可用的假象？
  - 时分复用(time sharing)技术





# 什么是进程？

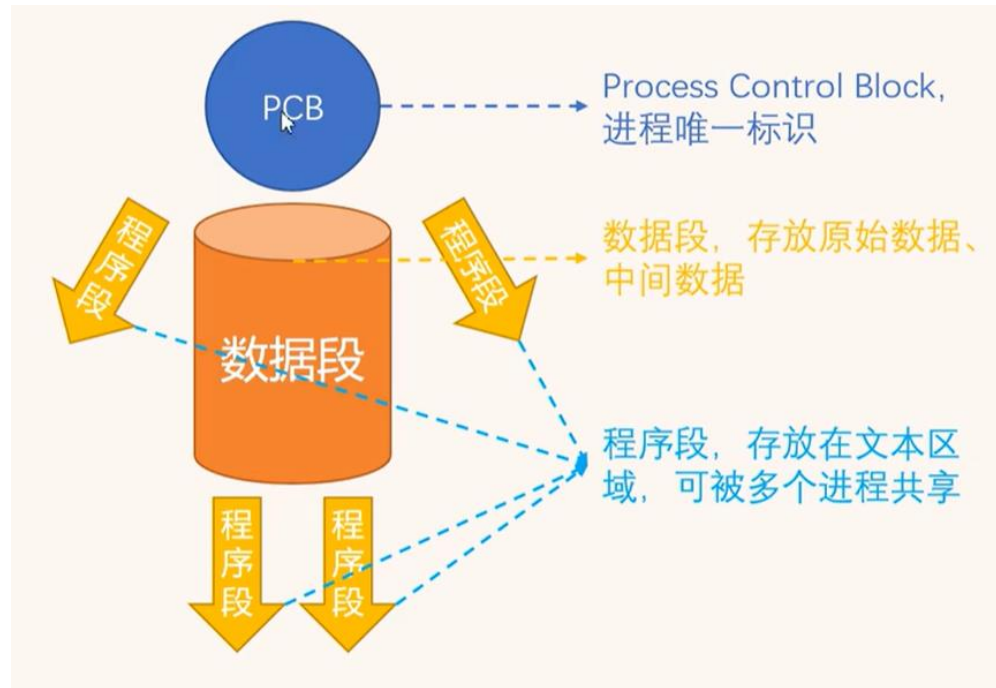
- 操作系统执行应用程序是以进程的方式运行的
- 进程（Process）：一个具有一定**独立功能**的程序在一个**数据集**上的一次**动态执行过程**。也称为**任务(Task)**。
- 几个要点：
  - 进程是『程序』的「一次执行」
  - 进程是一个程序及其数据在处理机上顺序执行时所发生的『活动』
  - 进程是程序在一个『数据集』上运行的过程
  - 进程是系统进行「资源分配和调度」的一个「独立」单位(或者说基本单位)





# 什么是进程

- 进程的构成
  - 控制块（PCB）
  - 程序段，也称为文本段
  - 数据段

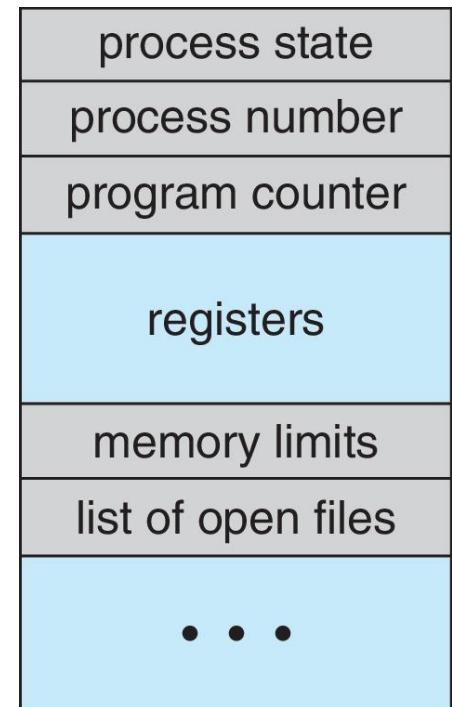




# 如何表示进程？

## ■ 进程控制块 (PCB)——管理程序运行的数据结构

- 进程状态：运行、等待、就绪等
- 程序计数器：下一条指令的位置
- CPU 寄存器：所有进程相关寄存器的内容
- CPU 调度信息：优先级、调度队列指针
- 内存管理信息：分配给进程的内存
- 记账信息：CPU 使用情况、自启动以来经过的时钟时间、时间限制
- I/O 状态信息：分配给进程的 I/O 设备、打开文件列表



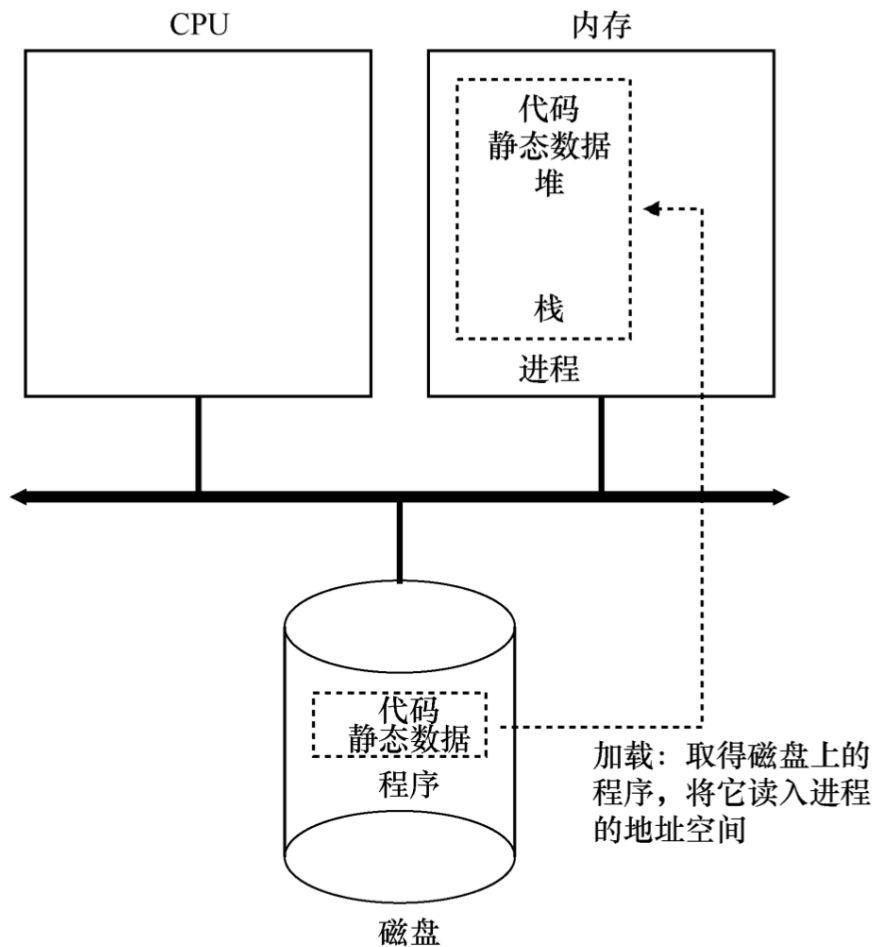


# 进程特征

- 进程是对正在运行程序的抽象
  - **动态性**：进程有生命周期，可以创建、运行、暂停、消亡。
  - **并发性**：进程可以并发执行
  - **独立性**：每个进程有独立的地址空间，不需要了解其它进程的实现细节，一个进程的错误不会影响到其它进程的执行；进程间可以通过通信机制共享资源。
  - **异步性**：每个进程有独立的执行环境，互不干扰。



# 程序如何转化为进程？



加载：从程序到进程

- 将代码和静态数据加载到内存
- 为程序的运行时栈 (run-time stack) 分配内存
- 为程序的堆(heap)分配内存
- 执行与I/O相关的设置
- 启动程序的运行





# 程序如何转化为进程？

- 程序是存储在磁盘上的被动实体（可执行文件）；进程是活动的
- 当可执行文件被加载到内存中时，程序变成进程
- 通过 GUI 鼠标点击或命令行输入其名称等，启动程序的执行
- 一个程序可以有多个进程
  - 考虑多个用户执行相同的程序
    - 文本编辑器
    - 编译器

```
#同时启动两个Hello World程序
```

```
$ ./hello & ./hello
```

```
Hello World!
```

```
Hello World!
```

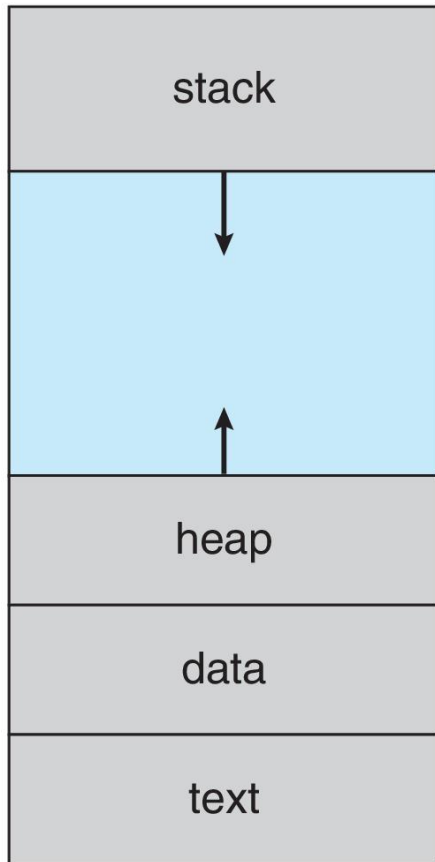






# 进程的内存映像

max



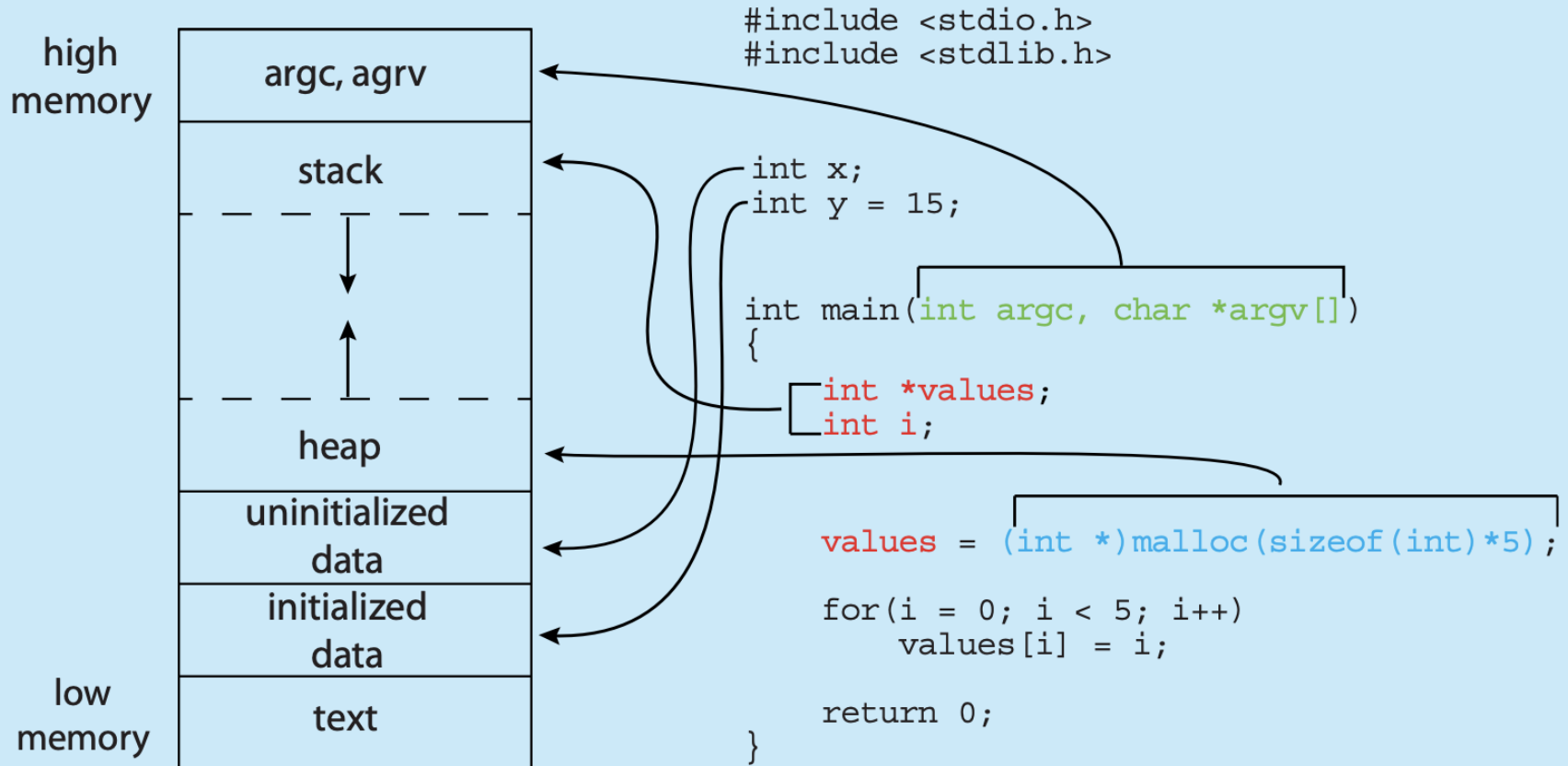
0

- 文本区/代码区：存放程序的机器代码，通常只读，以防止恶意或意外修改
- 数据区：存放全局变量和静态变量
- 堆：用于动态内存分配的区域
- 栈：用于存储函数调用的上下文信息



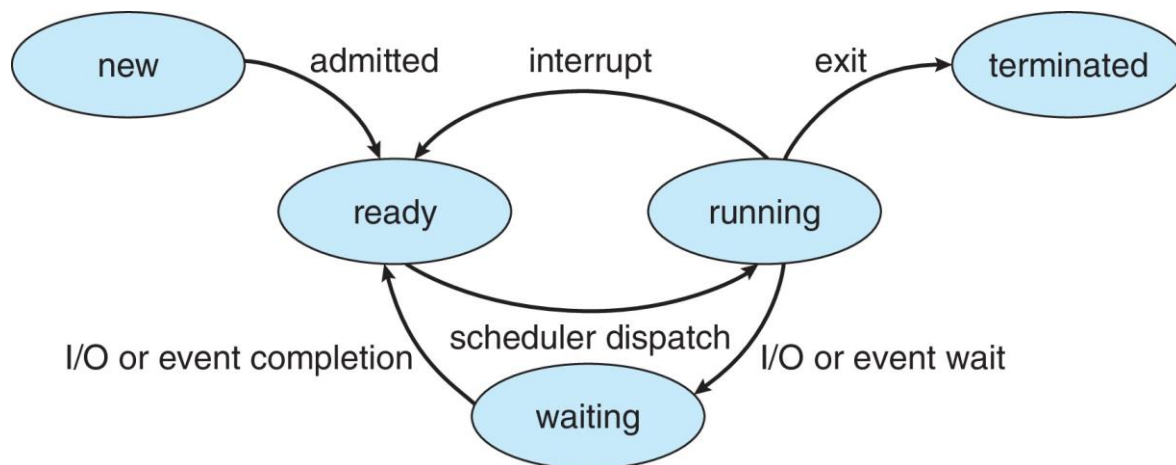


# MEMORY LAYOUT OF A C PROGRAM



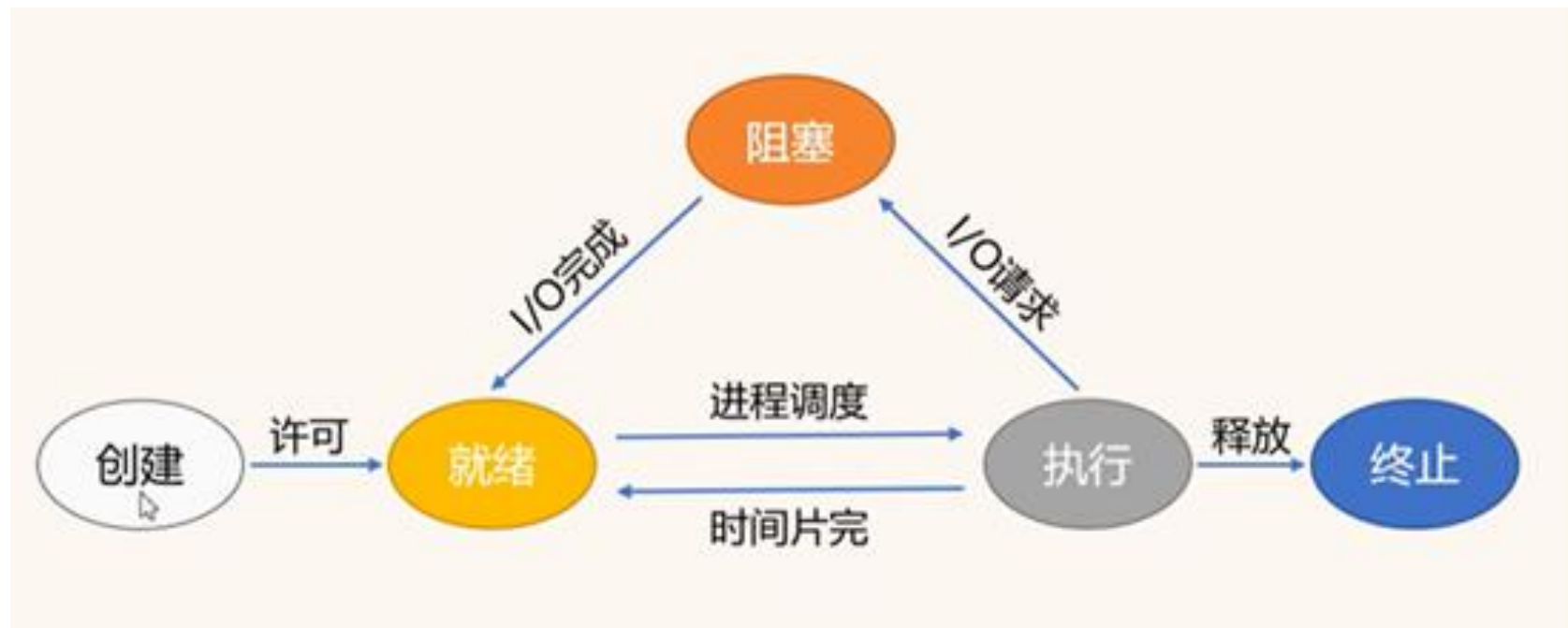
# 进程状态

- 进程运行时，状态会发生变化
  - 新建 **New**: 进程被创建
  - 运行 **Running**: 指令执行时的状态
  - 等待（阻塞） **Waiting**: 进程等待某些事件发生时的状态
  - 就绪 **Ready**: 进程等待获得 CPU
  - 终止 **Terminated**: 进程正常或异常结束时的状态



进程状态变迁图

# 进程状态





# 状态转换的有关说明

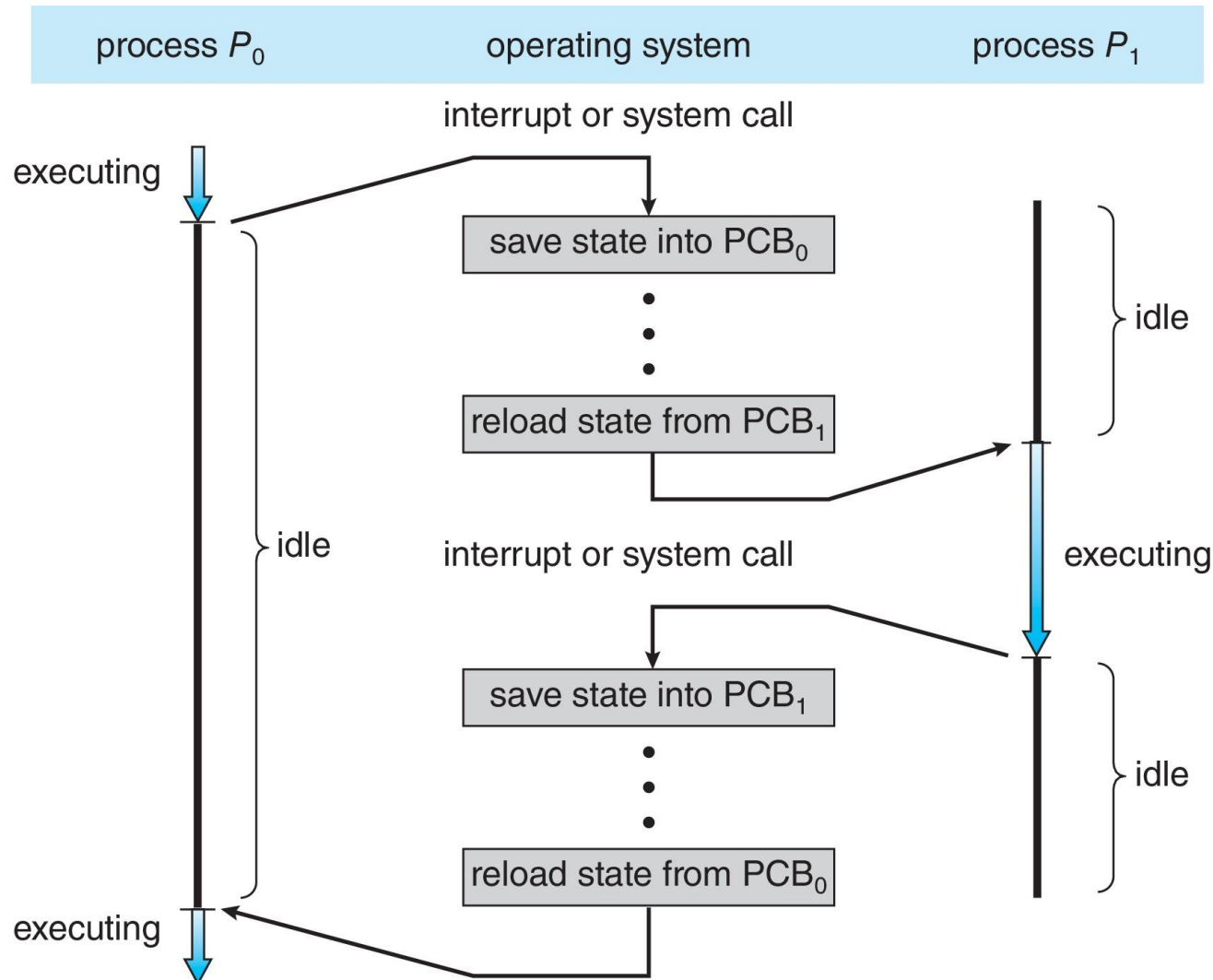
- 大多数状态不可逆转，如等待不能转换为运行
- 状态转换大多为被动进行，但运行→等待是主动的
- 一个进程在一个时刻只能处于上述状态之一





# CPU从一个进程切换给另一个进程

当CPU从一个进程切换到另一个进程时，就会发生上下文切换





# 上下文切换

- 当 CPU 切换到另一个进程时，系统必须通过上下文切换保存旧进程的状态，加载新进程的状态
- 进程的上下文保存在 PCB 中
- 上下文切换时间是纯开销，系统在切换时不执行任何有用的工作
  - 操作系统和 PCB 越复杂 → 上下文切换时间就越长
- 时间开销取决于硬件支持
  - 某些硬件为每个 CPU 提供多组寄存器 → 一次要加载多个上下文





# PCB包含的主要内容

- **进程标识符**：惟一标识进程的一个标识符或整数id号
- **进程当前状态**：说明进程当前所处状态
- **进程队列指针**：用于记录PCB队列中下一个PCB的地址
- **程序和数据地址**：进程的程序和数据在内存或外存中的存放地址
- **进程优先级**：反映进程获得CPU的优先级别







# PCB包含的主要内容

- **CPU现场保护区**：CPU现场信息的存放区域，包括：通用寄存器、程序计数器、程序状态字等
- **通信信息**：进程与其他进程所发生的信息交换时所记录的有关信息
- **家族关系**：指明本进程与家族的关系，如父子进程标识
- **资源清单**：列出进程所需资源及当前已分配资源

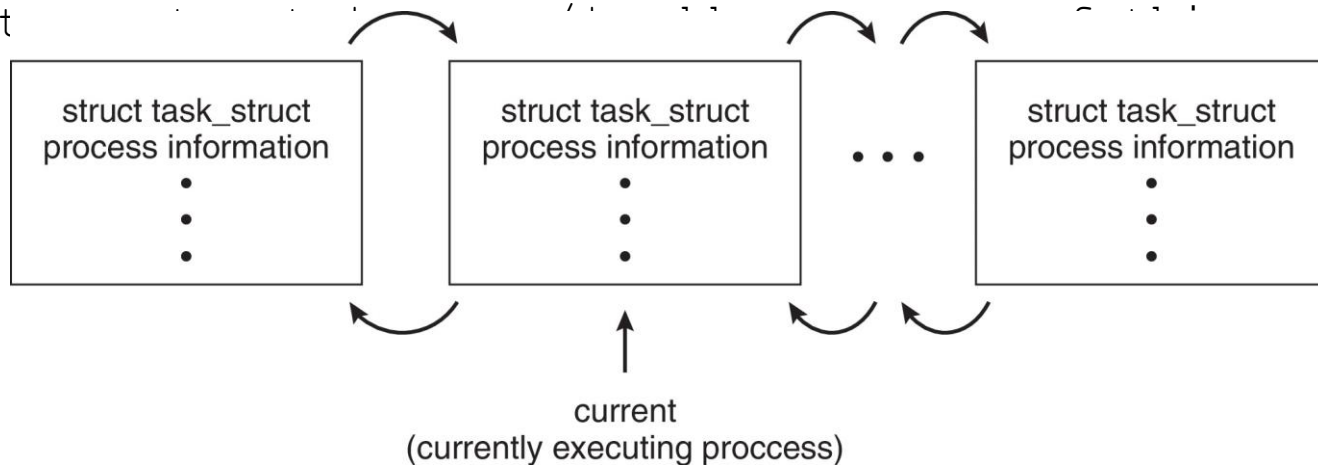




# Process Representation in Linux

Represented by the C structure `task_struct`

```
pid t_pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice /* scheduling information */
struct task_struct *parent; /* this process' s parent */
struct list_head children; /* this process' s children */
struct files_struct *files; /* list of open files */
struct
```





# 一个程序在运行中状态的转换

```
1  #include <stdio.h>
2  #define LEN 10
3
4  int main(int argc, char *argv[]){
5      char name[LEN] = {0};
6      fget(name, LEN, stdin);
7      printf("Hello %s\n", name);
8      return 0;
9  }
```

```
1  $./hello_name
2  xiaoming
3  Hello xiaoming
```





# 进程状态的变化

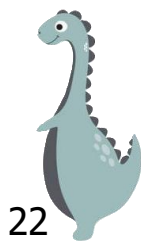
1. 执行./hello\_name，内核创建新进程，且还未完成初始化时，处于**创建状态**
2. 内核对进程需要的数据结构进行初始化，并将其交给调度器，加入就绪队列，使其处于**就绪状态**
3. 若调度器选择该进程执行，则变为**运行状态**，开始执行main函数
4. 进程执行到fget，需接受用户的输入，此时进程变为**阻塞状态**





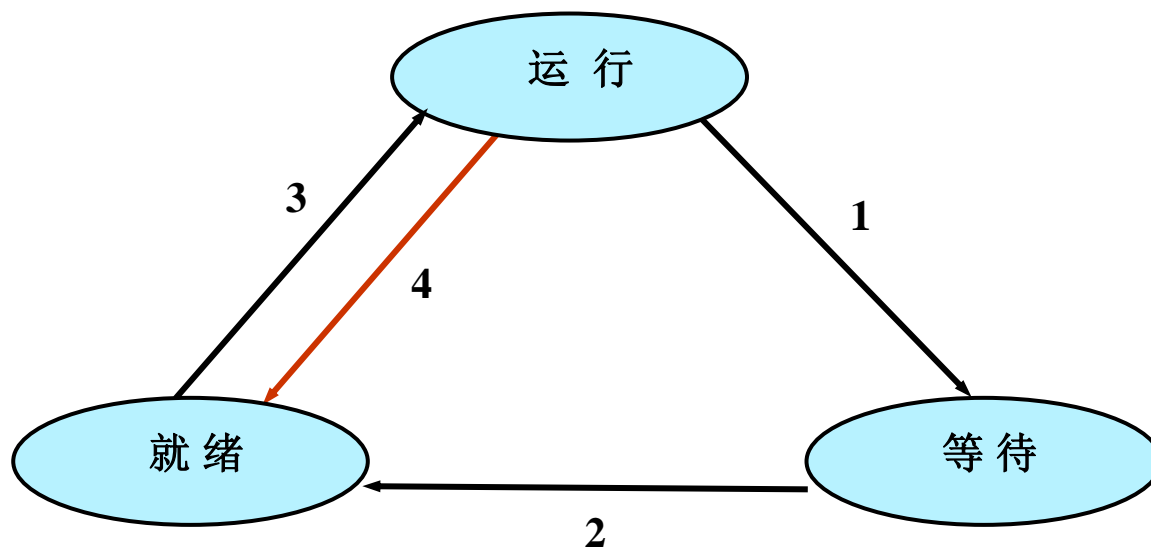
# 进程状态的变化

5. 用户在屏幕上输入xiaoming并回车，则进程会重新回到就绪队列，并在CPU空闲的时候重新回到**运行状态**，并输出“Hello Xiaoming”
6. 进程执行完main函数，回到内核中，进程变为**终止状态**，内核回收该进程的资源





# 进程状态变迁的讨论



进程状态的变迁

- 变迁1——> 变迁3，是否会发生？需要什么条件？
- 变迁4——> 变迁3，是否会发生？需要什么条件？
- 变迁2——> 变迁3，是否会发生？需要什么条件？





# 多进程的并发执行

- 程序A、B、C分别是冒泡排序、堆排序和快速排序算法，它们分别在屏幕的左、中右1/3处开设窗口显示其排序过程；
- 在不支持多进程的OS下如何运行？
- 在支持多进程的OS下如何运行？





# 进程控制

- 即OS对进程实现有效的管理，包括创建新进程、撤销已有进程、挂起、阻塞和唤醒、进程切换等多种操作。
- OS通过原语(Primitive)操作实现进程控制。
- 原语的概念：
  - 由若干条指令组成，完成特定的功能，是一种原子操作(Action Operation)
- 原语的特点：
  - 原子操作，要么全做，要么全不做，执行过程不会被中断。
  - 在管态/系统态/内核态下执行，常驻内存。
  - 是内核三大支撑功能(中断处理/时钟管理/原语操作)之一



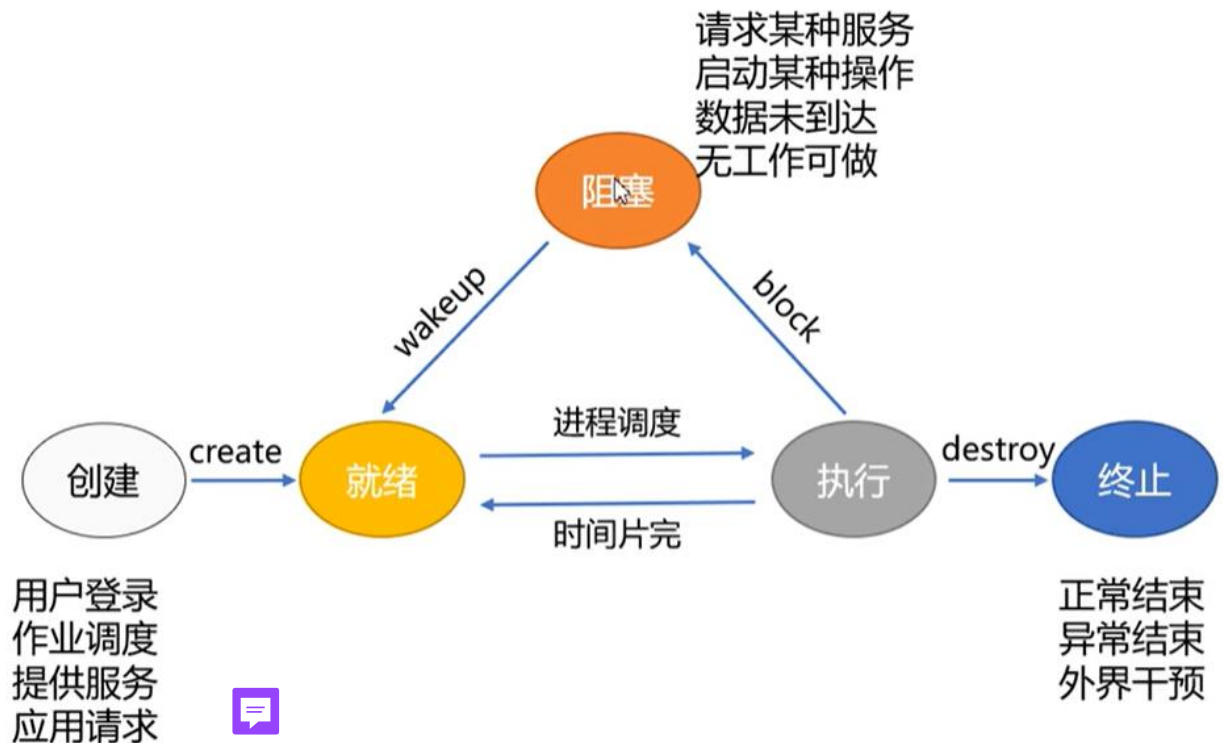




# 进程控制

## ■ 常用系统调用

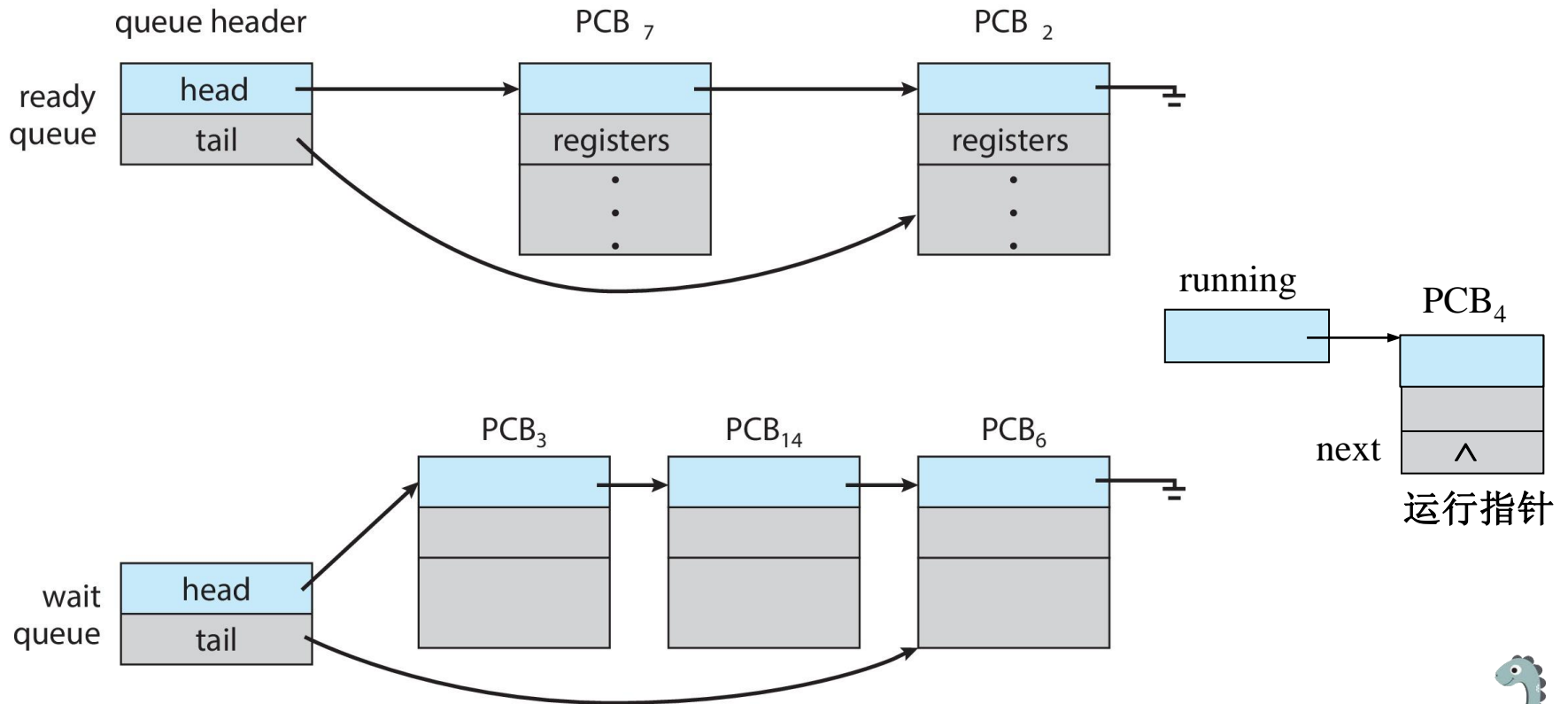
- 创建
- 终止
- 阻塞（等待）
- 唤醒：





# 进程的组织方式

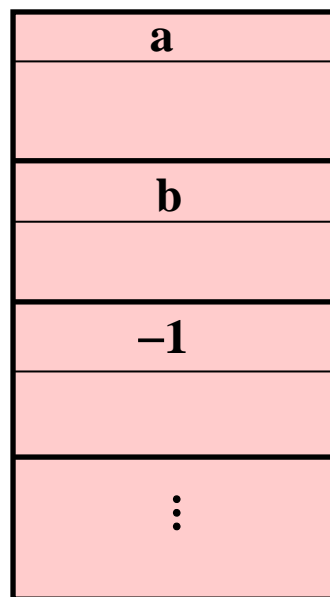
## ■ 大量进程如何组织？



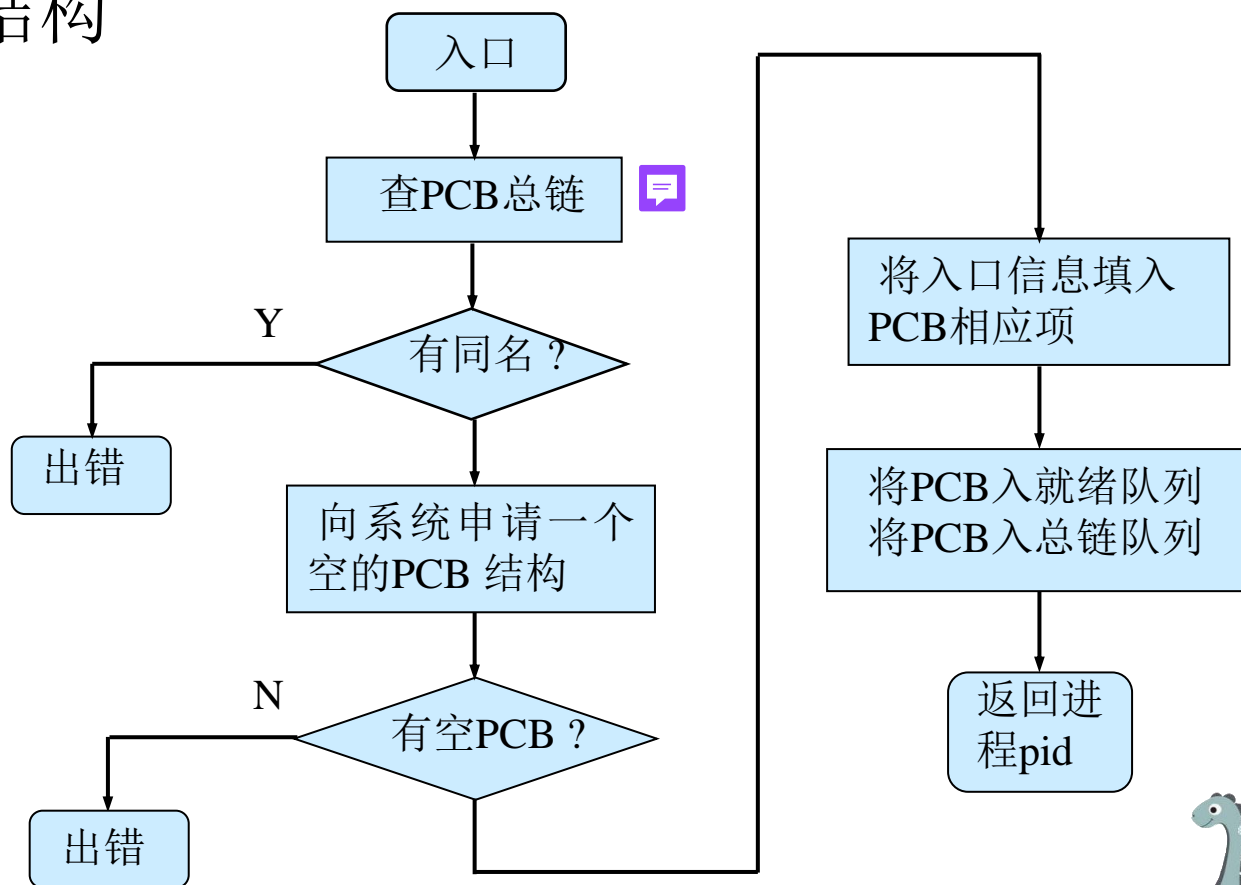


# 进程创建

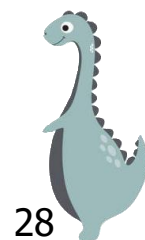
- 功能：创建一个具有指定标识符的进程，建立进程的PCB结构



PCB池示意图



进程创建原语流程图





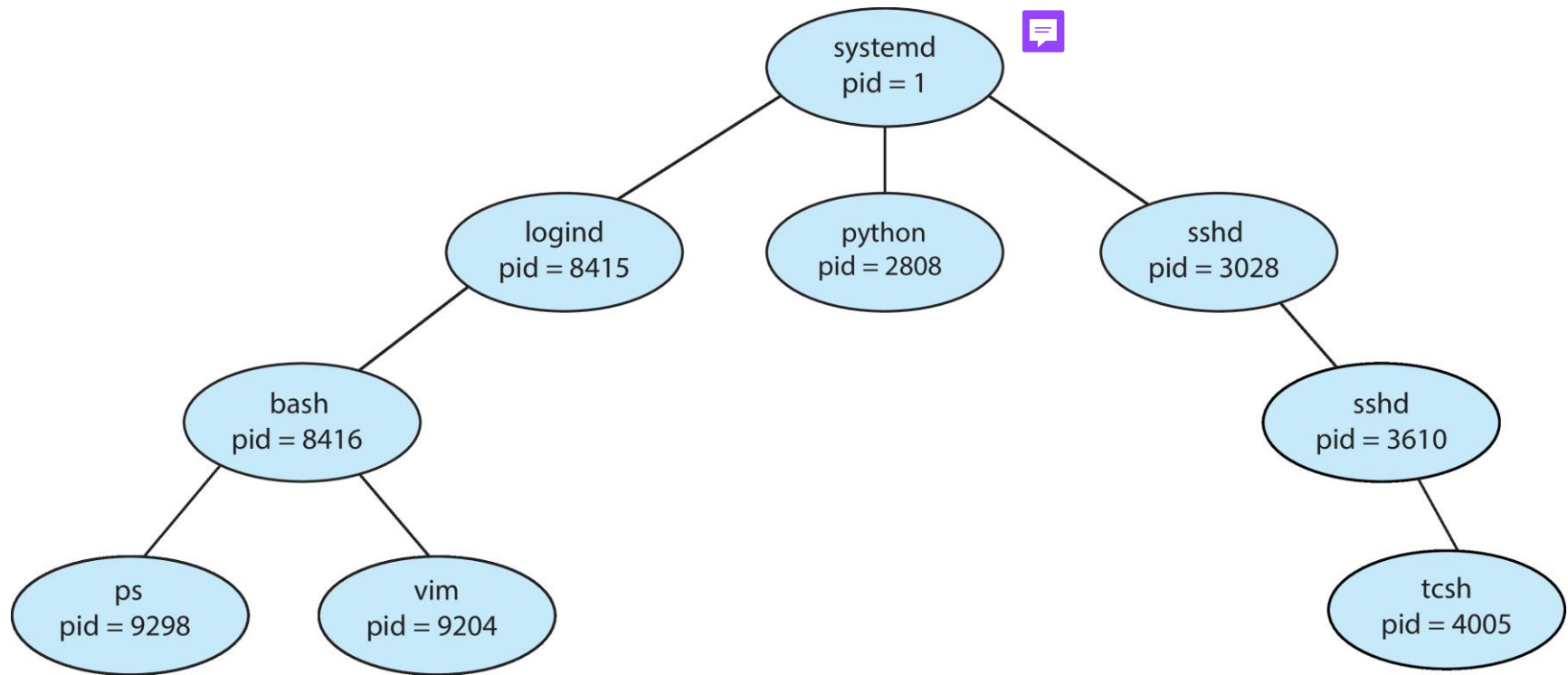
# 进程创建

- 父进程创建子进程，子进程又创建子进程，从而形成进程树
- 通常进程通过进程标识符（pid）进行标识和管理
- 父子进程的资源共享
  - 父进程和子进程共享所有资源
  - 子进程共享父进程资源的子集
  - 父进程和子进程不共享任何资源
- 父子进程的执行：
  - 父进程和子进程并发执行
  - 父进程等待子进程终止





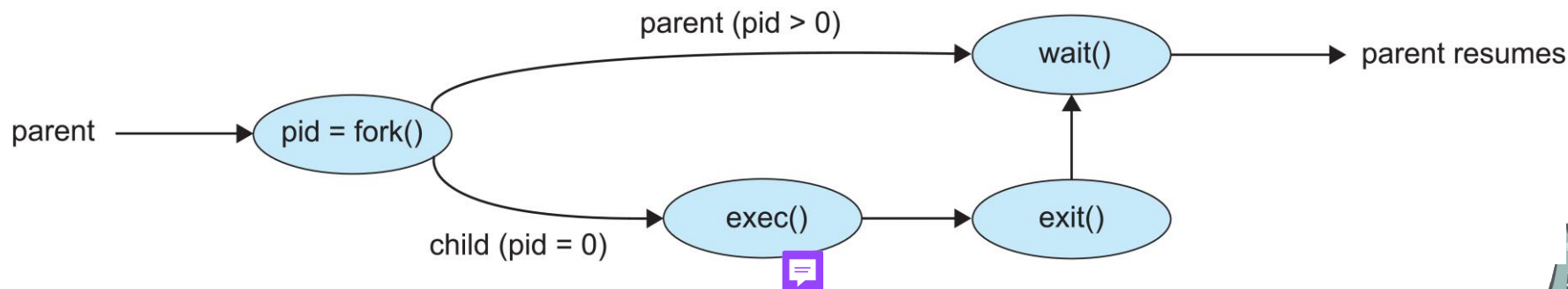
# A Tree of Processes in Linux





# 进程创建

- 地址空间
  - 子进程是父进程的副本
  - 子进程中加载了一个程序
- UNIX 示例
  - `fork()` 系统调用创建新进程
  - `exec()` 系统调用在 `fork()` 之后使用，用于将进程的内存空间替换为新程序
  - 父进程调用 `wait()` 等待子进程终止





# wait函数的作用

`wait(&status);`

`waitpid(pid, &status, options);`

- `wait()`: 该函数暂停调用进程，直至其子进程结束。如果子进程已经结束（成为僵尸进程），则该函数立即返回。子进程的结束状态会被返回并存储在 `status` 所指向的位置。
- `waitpid()`: 与 `wait()` 类似，但它添加了更多的控制选项。你可以使用它来等待一个特定的子进程，或者是满足特定条件的任何子进程。





# C Program Forking Separate Process

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

int main()
{
    pid_t pid;

    /* fork a child process */
    pid = fork();

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

    return 0;
}
```







# Creating a Separate Process via Windows API

```
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    /* allocate memory */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));


    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
        "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
        NULL, /* don't inherit process handle */
        NULL, /* don't inherit thread handle */
        FALSE, /* disable handle inheritance */
        0, /* no creation flags */
        NULL, /* use parent's environment block */
        NULL, /* use parent's existing directory */
        &si,
        &pi))
    {
        fprintf(stderr, "Create Process Failed");
        return -1;
    }
    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```





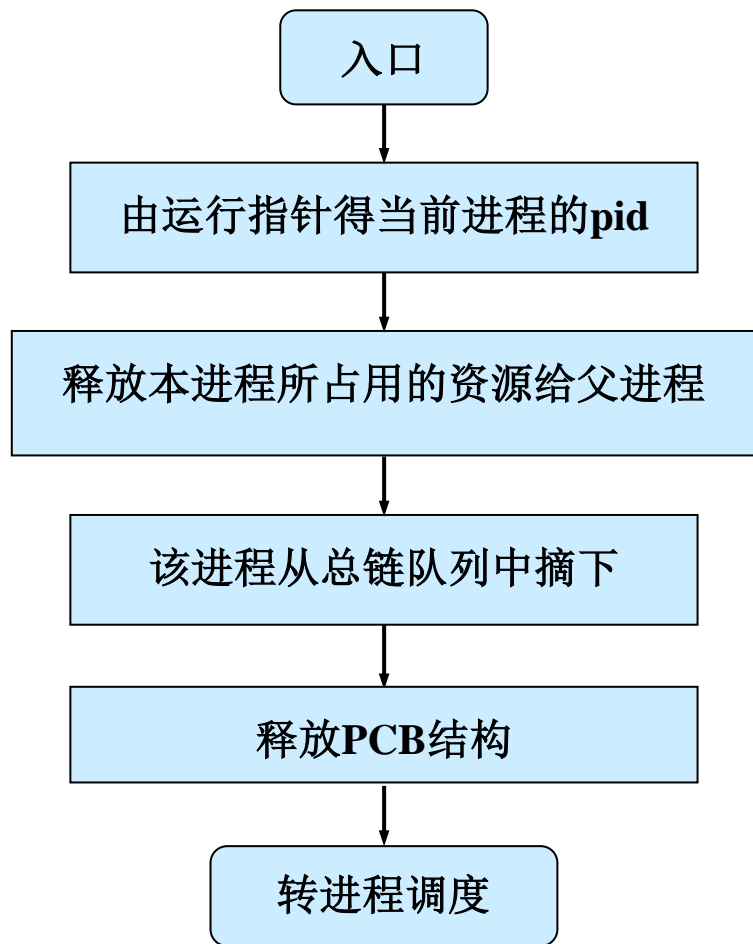
# 导致进程创建的原因

- 用户登录：用户登录后，若合法则为用户创建一个进程。
- 作业调度：为调度到的作业分配资源并创建进程。
- OS服务：创建服务进程。
- 应用需要：应用程序根据需要创建子进程。





# 进程终止



进程终止原语流程图

- 功能：终止当前运行的进程。将该进程的PCB结构归还到PCB资源池，所占用的资源归还给父进程，从总链队列中摘除它，然后转进程调度程序。



# 进程终止

- 进程执行最后一条语句，然后通过`exit()`系统调用请求操作系统删除它
  - 通过`wait()`将状态数据从子进程返回给父进程
  - 进程的资源由操作系统释放
- 父进程可以使用`abort()`系统调用终止子进程的执行。一些终止子进程的原因包括：
  - 子进程超过了分配的资源
  - 分配给子进程的任务不再需要
  - 父进程正在退出，如果父进程终止，操作系统不允许子进程继续执行





# 进程终止

- 如果父进程终止，则某些操作系统不允许子进程存在。如果一个进程终止，则其所有子进程也必须终止。
  - 级联终止。所有子代进程都将被终止
  - 终止由操作系统发起
- 父进程可以使用`wait()`系统调用等待子进程的终止。该调用返回状态信息和已终止进程的pid。
- `pid = wait(&status);`
- 如果没有父进程等待（未调用`wait()`），则该进程是僵尸进程
- 如果父进程终止而没有调用`wait()`，则该进程是孤儿进程





# 引发进程终止的原因

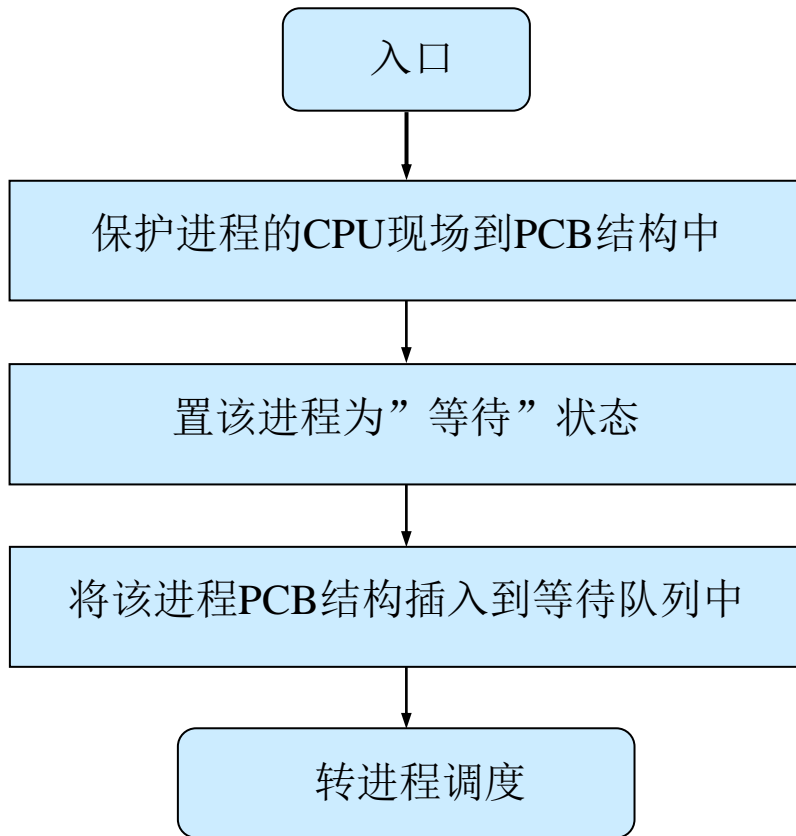
- 1. 正常退出：进程完成了它的任务并正常退出，通过调用退出系统调用（如 Linux 的 `exit()`）实现
- 2. 错误退出：如果进程遇到无法处理的运行时错误，如除以零或访问无效内存，它可能会被操作系统终止
- 3. 致命错误：某些严重的错误，如硬件错误或操作系统错误，可能导致进程被终止
- 4. 由其他进程终止：一个进程可以请求操作系统终止另一个进程。这通常通过发送一个信号（如 Unix 或 Linux 的 `kill`）或调用一个系统调用（如 Windows 的 `TerminateProcess()`）实现
- 5. 用户请求：用户可以请求操作系统终止一个进程。这可以通过命令行（如 Unix 或 Linux 的 `kill` 命令）或图形用户界面（如任务管理器）实现
- 6. 操作系统干预：如果进程使用了过多的系统资源（如 CPU 时间或内存），或者它的行为违反了操作系统的策略（如优先级太低而 CPU 时间不足），操作系统可能会终止它
- 7. 父进程终止：在某些操作系统中，如果父进程被终止，它的子进程可能也会被终止
- 8. 系统关机或重启：当操作系统关机或重启时，所有正在运行的进程都会被终止





# 进程阻塞（等待）

- 功能：暂停进程的  
执行，并将其加入  
到等待某事件的等  
待队列中；将控制  
转向进程调度



进程等待原语流程图





# 引发进程阻塞的事件

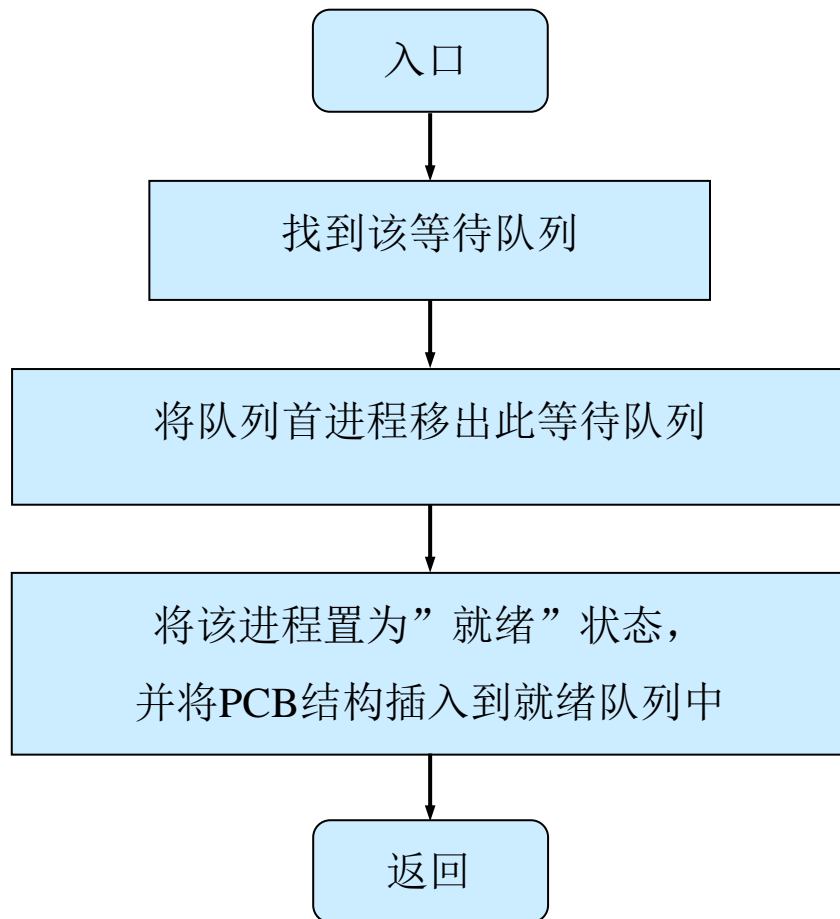
1. **I/O 请求**: 进程发出 I/O 请求并等待其完成时，例如读写磁盘文件或网络数据。
2. **等待子进程**: 父进程使用如 `wait()` 这样的系统调用等待一个或多个子进程终止。
3. **信号或消息**: 进程等待接收特定的信号或消息。
4. **资源争用**: 进程等待获取互斥锁、信号量或其他同步原语。
5. **内存页错误**: 进程访问的内存页不在物理内存中，需要从磁盘中调入。





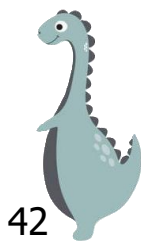


# 进程唤醒



进程唤醒原语流程图

- 功能：当进程等待的事件发生时，由事件发现者唤醒等待该事件的进程。






# 引发进程唤醒的事件

1. **I/O 完成**: 进程发出的 I/O 请求已完成。
2. **子进程终止**: 进程等待的子进程已终止。
3. **接收到信号或消息**: 进程接收到了它正在等待的信号或消息。
4. **获取资源**: 进程等待的资源（如互斥锁、信号量）已经可用。
5. **内存页调入完成**: 进程等待的内存页已经被调入物理内存。





# 进程的挂起状态

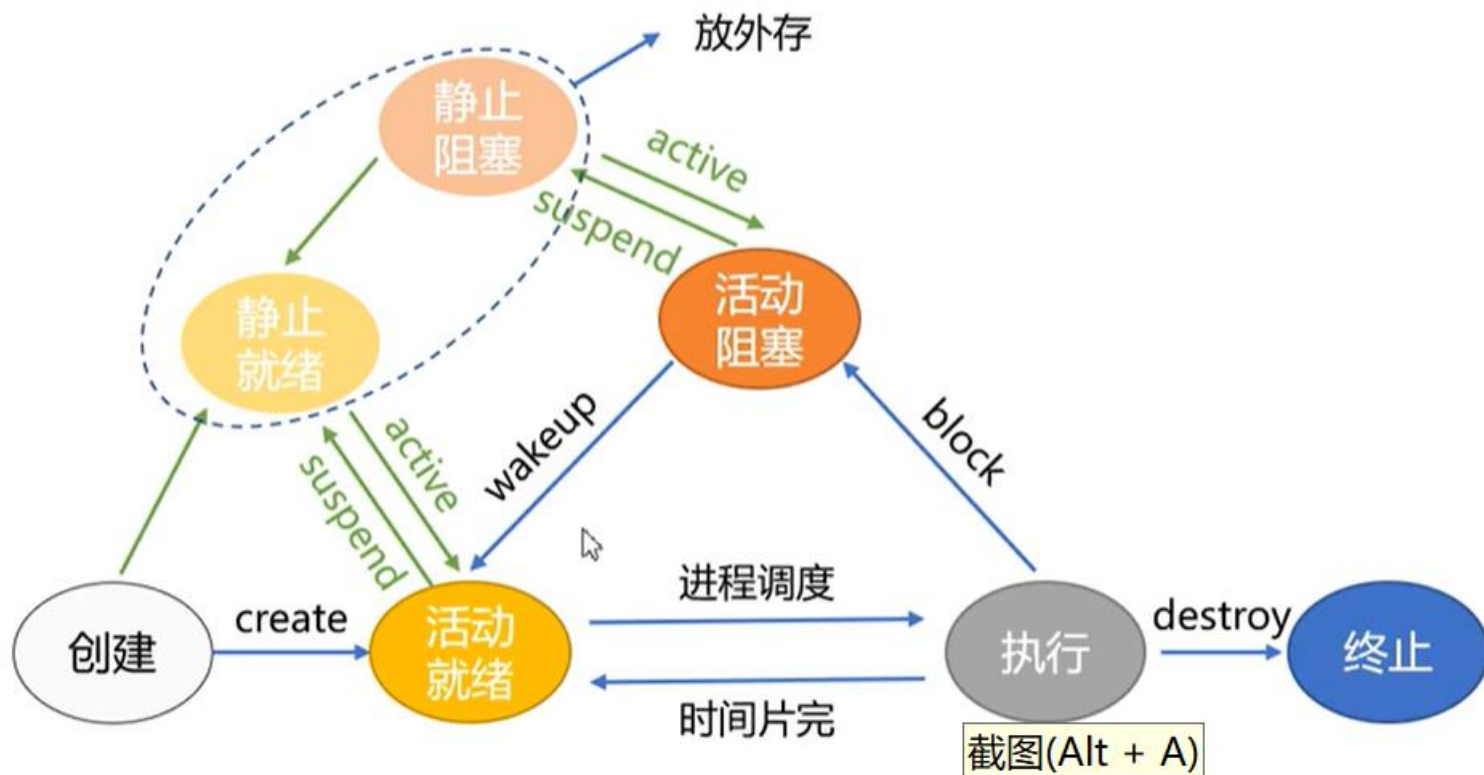
- 进程的挂起指的是进程因为某种原因暂时不能继续执行，需要等待某个事件的发生或者满足某种条件后才能恢复执行。
- 在挂起状态下，进程会被保存在磁盘上，不占用内存资源，也不占用CPU时间。
- 进程挂起的原因有：
  - 用户请求
  - 系统资源不足
  - 进程间通信
  - I/O操作
  - 错误或异常





# 有挂起状态的进程状态转换图

- 引入挂起状态





# 挂起原语的功能实现

- 将进程的状态设置为挂起状态
- 如果进程在CPU中运行，那么保存进程的上下文信息，包括程序计数器、寄存器值、内存映射等
- 将进程的上下文信息和其他相关数据结构（如打开文件列表、信号处理状态等）写入磁盘。
- 将进程从调度队列中移除





# 挂起原语的主要功能

- 挂起原语的主要功能是将指定进程挂起，算法思想如下：
  - 到PCB表中查找该进程的PCB；
  - 检查进程状态：如果该进程已经处于挂起状态或者由于其他原因（如等待I/O操作）无法被挂起，那么挂起原语会返回一个错误；
  - 更改进程状态：如果目标进程可以被挂起，那么挂起原语会将其状态从当前状态（如运行或就绪）更改为挂起状态；
  - 保存进程上下文：这样当进程被重新激活时，它可以从中断的地方恢复执行；
  - 更新PCB：挂起原语会更新PCB，反映进程的新状态和上下文信息
  - 调整调度队列：将目标进程从就绪队列中移除，并将其添加到一个专门的挂起队列中。这样，操作系统的调度器就不会再选择这个进程进行执行，从而实现了进程的挂起。





# 激活原语的主要功能




- 激活原语的主要功能是将指定进程激活，算法思想如下：
  - 到PCB表中查找该进程的PCB
  - 检查进程状态：如果该进程不处于挂起状态，那么激活原语会返回一个错误
  - 更改进程状态：如果目标进程处于挂起状态，那么激活原语会将其状态从挂起状态更改为就绪状态
  - 恢复进程上下文：激活原语会恢复进程的上下文信息，这样，当进程被重新调度执行时，它可以从中断的地方恢复执行
  - 更新PCB：激活原语会更新PCB，反映进程的新状态和上下文信息
  - 调整调度队列：将目标进程从挂起队列中移除，并将其添加到就绪队列中。这样，操作系统的调度器就可以再次选择这个进程进行执行，从而实现了进程的激活。





# 什么是线程

- 
- Thread，进程的轻型实体，也叫“轻量级进程”，是一系列活动按事先设定好的顺序依次执行的过程，是一系列指令的集合。
  - 是一条执行路径，不能单独存在，必须包含在进程中线程。
  - 是OS中运算调度的最小单位



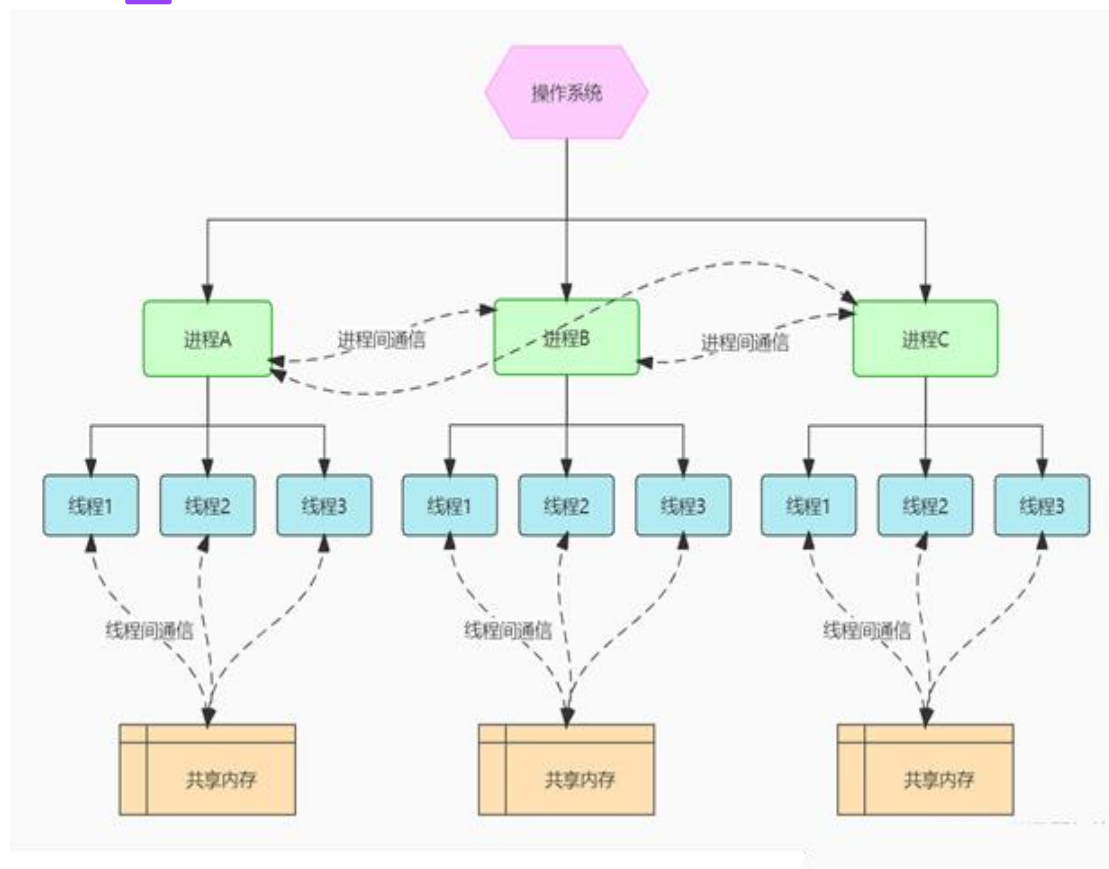


# 什么是线程




## ■ 线程的属性

- 轻型实体
- 独立调度和分派的基本单位
- 可并发执行
- 共享进程资源





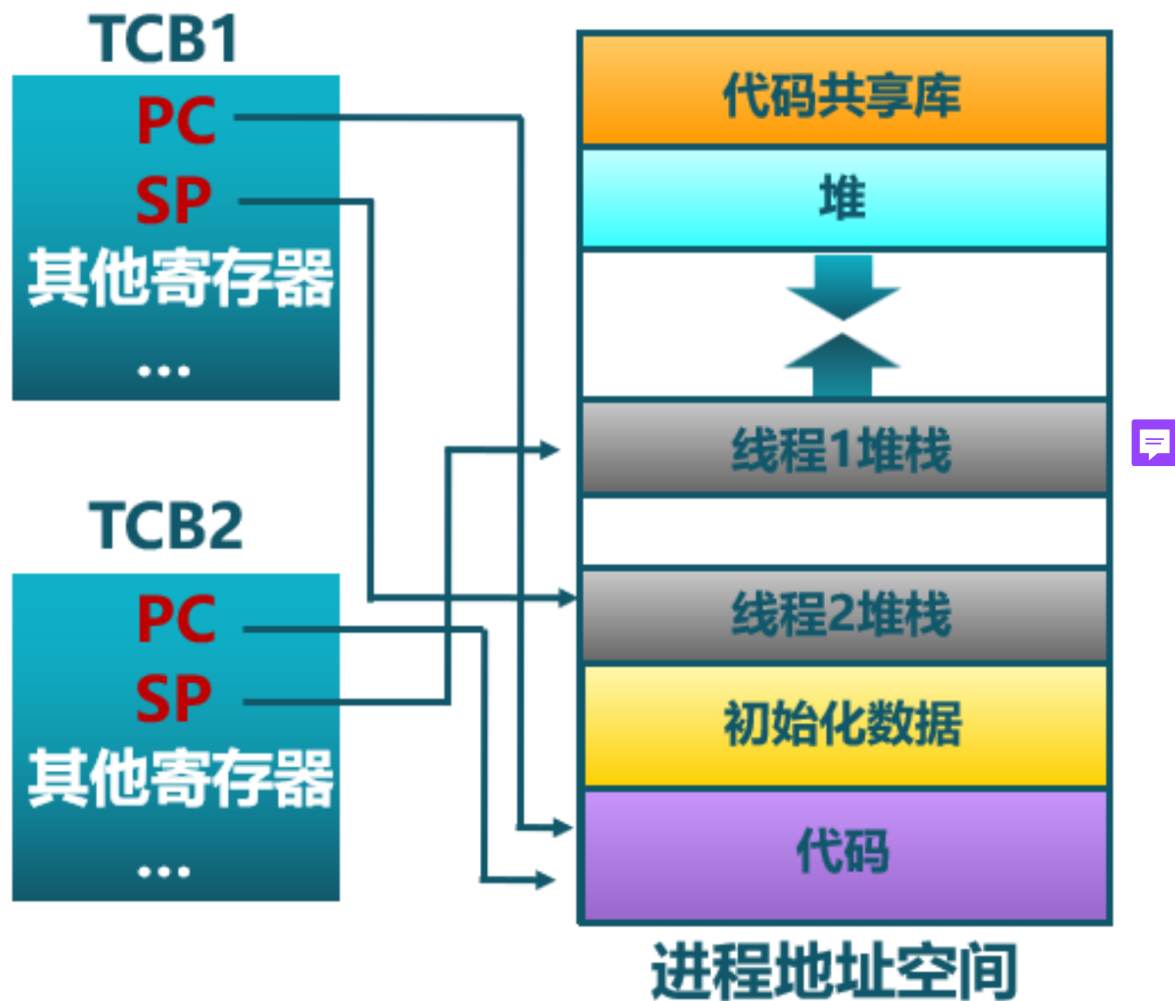
# 线程的组成

- 线程控制块（TCB）：TCB是线程的大脑，它存储了操作系统需要管理和调度线程所需的所有信息。包括：
  - 线程标识符
  - 线程状态（如运行、就绪、阻塞等）
  - CPU寄存器值（PC、PS和通用寄存器）
  - 线程优先级
  - 所属进程的引用





# 线程的组成





# 为什么要引入线程？

- 单线程应用的瓶颈问题
  - CPU利用率低
  - 响应性能差
  - 无法充分利用多核CPU
  - 不适合并发处理
- 应用程序中的多个任务可以通过多个线程来实现
  - 更新显示
  - 获取数据
  - 拼写检查
  - 响应网络请求
- 线程的创建是轻量级的，而进程的创建是重量级的
- 使用多线程可以简化代码，提高效率
- 内核通常也是多线程的





# 多线程的优势

- 响应性：进程在某个部分被阻塞的情况下被允许继续执行，对于用户界面尤其重要
- 资源共享：线程共享进程的资源，比共享内存或消息传递更容易
- 经济性：比进程创建的开销小，线程切换的开销比上下文切换低
- 可扩展性：进程可以利用多核架构的优势





# 进程和线程的关系

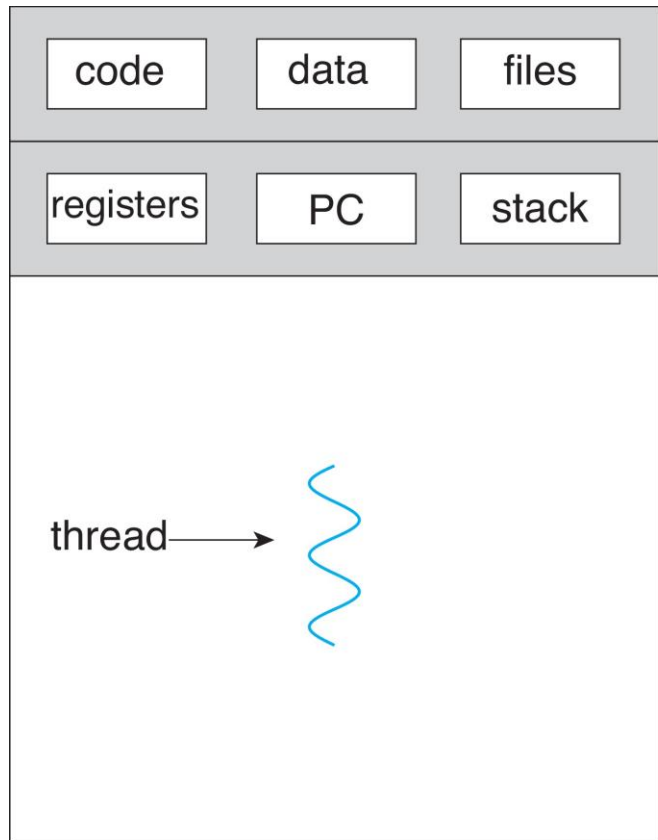
- 进程
  - 是系统进行资源分配和调度的一个独立单位；
  - 有自己的独立内存空间，包括代码段、数据段和堆栈段等；
  - 进程间通信需要使用进程间通信（IPC）机制，如管道、消息队列等。
- 线程
  - 是CPU调度和分派的基本单位，是比进程更小的能独立运行的基本单位；
  - 一个进程可以包含多个线程，线程运行在同一内存空间，共享相同的运行环境；
  - 线程之间可以直接读写同一进程中的数据，线程间的通信更便捷
- 关系
  - 线程是属于进程的，它们可以被看作是在进程内部的一个个独立的执行路径；
  - 一个进程内的多个线程之间共享该进程的资源，如内存空间、文件等；
  - 线程间的通信比进程间的通信更简单，线程的切换开销也比进程切换要小。



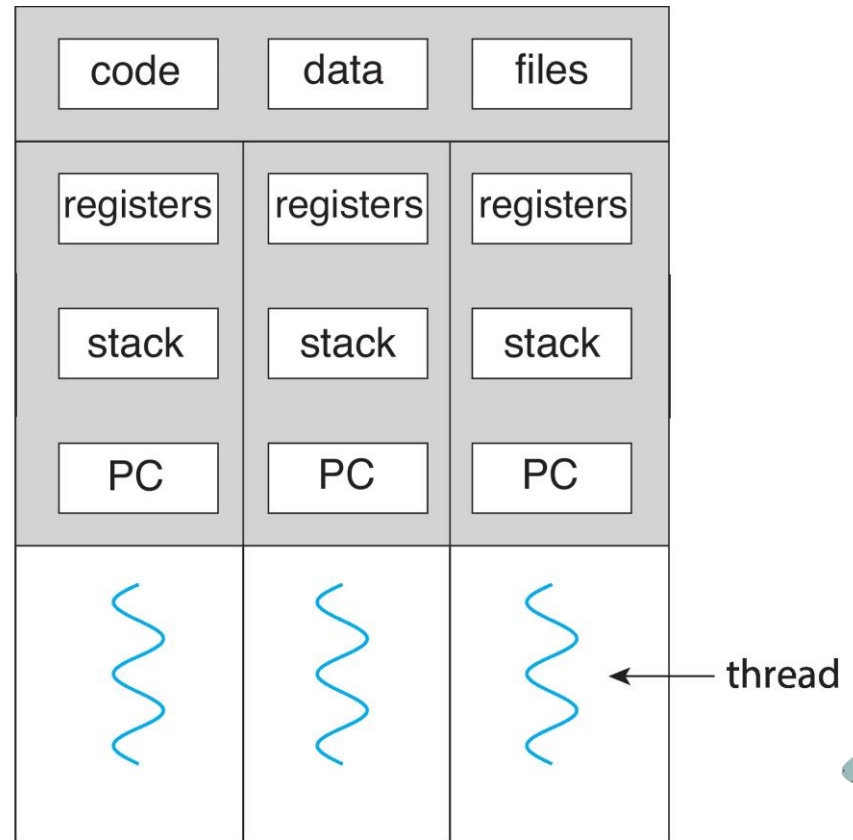


# 单线程进程和多线程进程

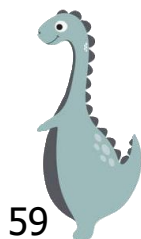
- 进程 = 线程 + 资源



single-threaded process



multithreaded process



# 多线程进程的地址空间布局

- 分离的内核栈和用户栈
  - 每个线程都有自己的栈，用于存放临时数据
  - 用户线程切换到内核中执行时，栈指针则切换到对应的内核栈
- 共享其它区域
  - 除栈以外的其它区域，进程内的所有栈共享
  - 一个进程的多个线程需要动态分配内存时，将在同一个堆上完成



包含三个线程的进程的地址空间布局





# 进程与线程比较

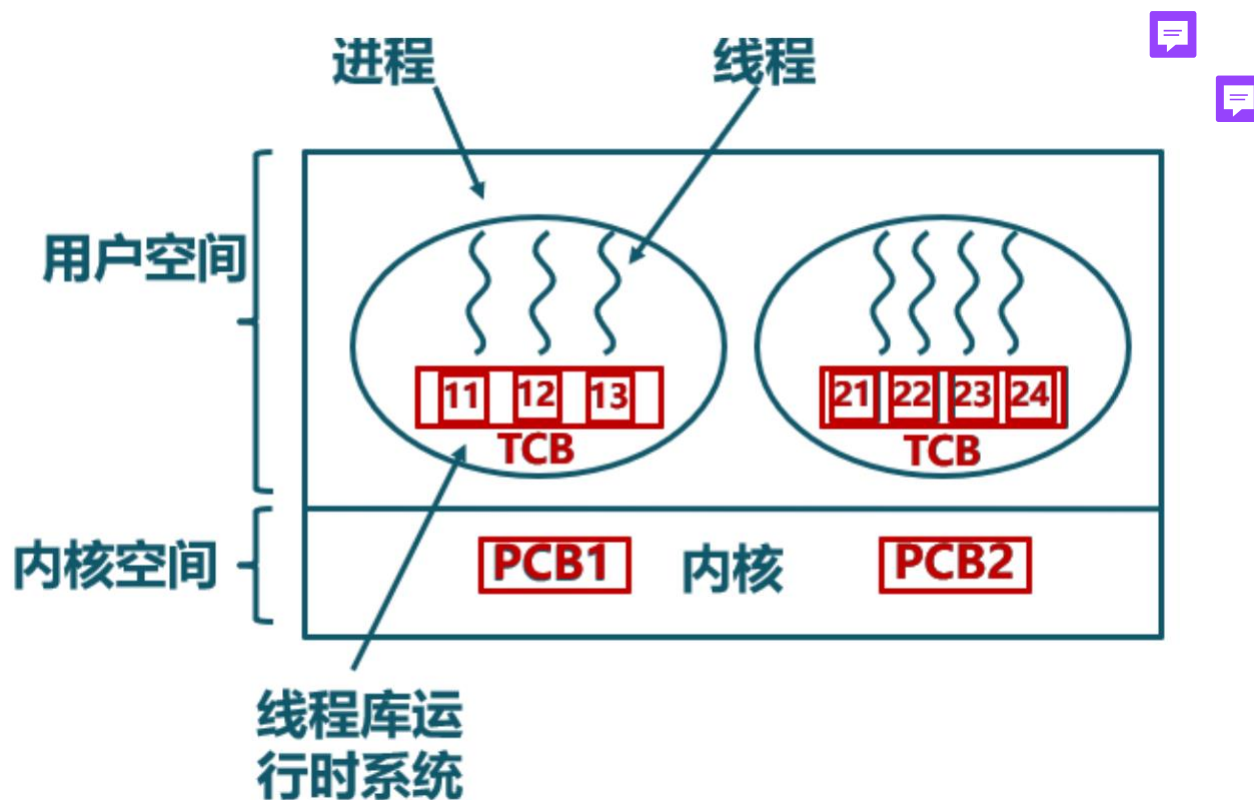
- 进程是资源分配单位，线程是CPU调度单位
- 进程拥有一个完整的资源平台，而线程只独享指令流执行的必要资源，如寄存器和栈。
- 线程具有就绪、等待和运行三种基本状态和状态间的转换关系。
- 线程能减少并发执行的时间和空间开销
  - 线程的创建/终止/切换时间比进程短
  - 同一进程的各线程间共享内存和文件资源，可不通过内核进行直接通信



# 线程的实现方式

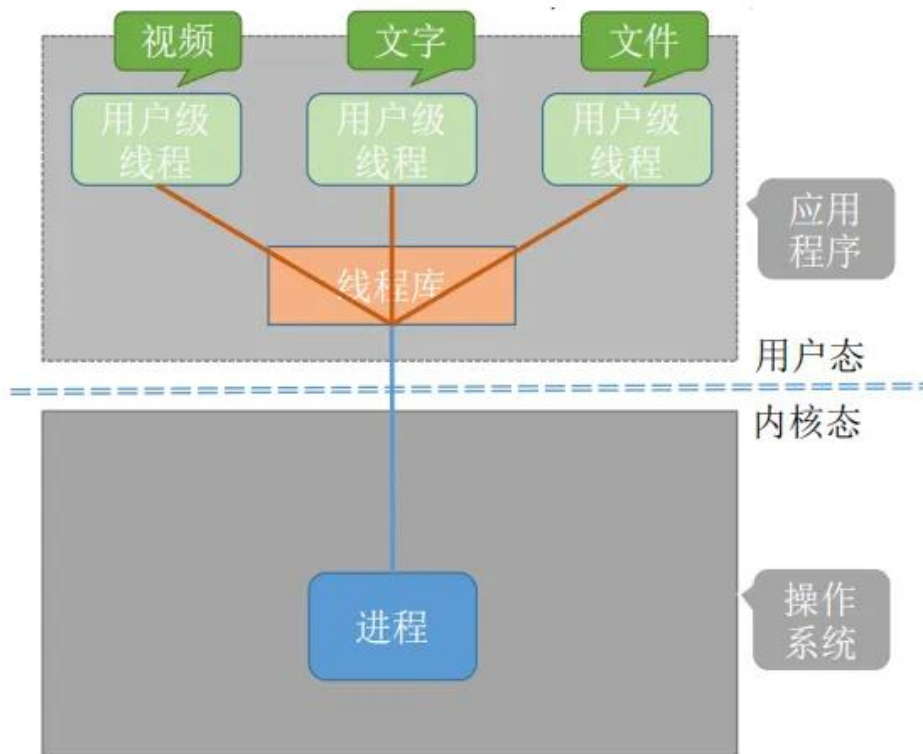
## 用户级线程（User-Level Threads）

- 由一组用户级的线程库函数来完成线程的管理，包括线程的创建、终止、同步和调度等





# 例子



```
int main() {  
    int i = 0;  
    while (true) {  
        if (i==0){处理视频聊天的代码;}  
        if (i==1){处理文字聊天的代码;}  
        if (i==2){处理文件传输的代码;}  
        i = (i+1)%3; //i的值为 0,1,2,0,1,2...  
    }  
}
```

QQ进程

从代码的角度看，线程其实就是一段代码逻辑。上述三段代码逻辑上可以看作三个“线程”。while 循环就是一个最弱智的“线程库”，线程库完成了对线程的管理工作（如调度）。





# 用户级线程特点

- 线程完全在用户空间中实现，不需要内核的支持；
- 用户级线程的创建、销毁、同步和切换都是由相应的用户级线程库来完成的，这些操作通常比内核级线程的对应操作要快得多；
- 操作系统只看到进程而不是用户级线程，所以它不能直接调度用户级线程，也不能在多处理器系统上并行运行用户级线程；
- 如果一个用户级线程阻塞（例如等待I/O操作完成），整个进程（包括所有其他用户级线程）都会被阻塞。





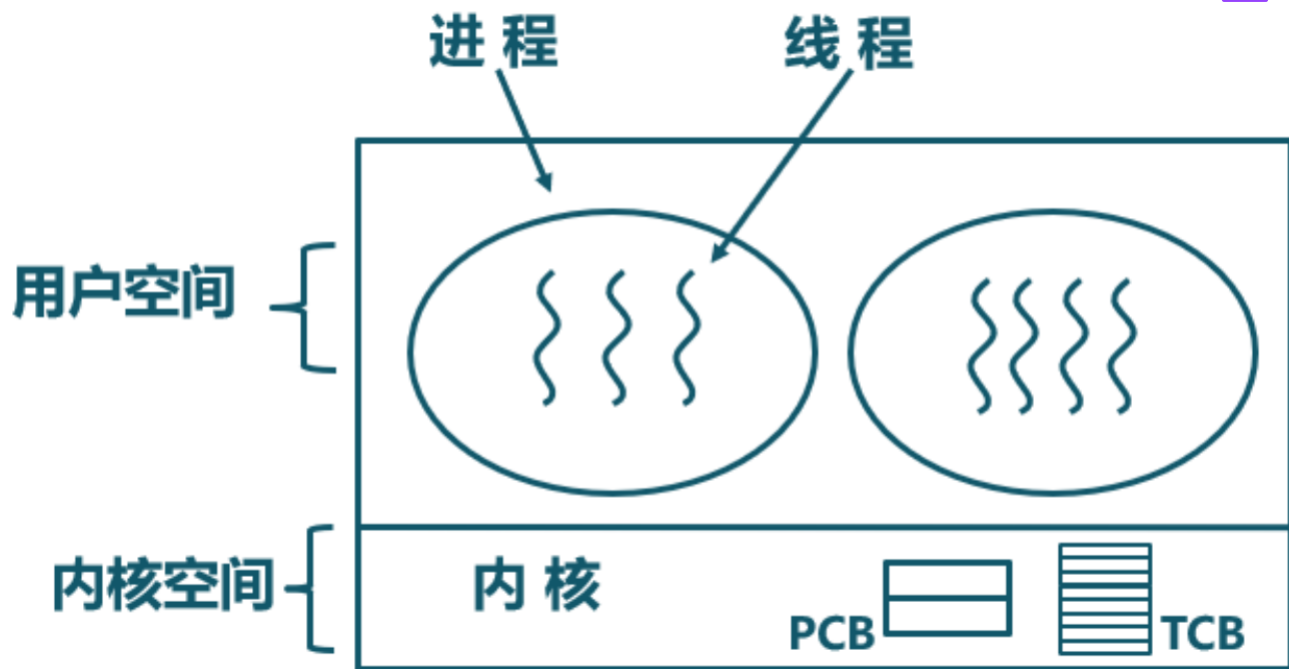
# 支持用户级线程的线程库

- **POSIX Threads (Pthreads):** Pthreads是一种基于POSIX标准的线程库，它在许多UNIX-like系统（如Linux、Mac OS X等）上都有实现。Pthreads提供了一套丰富的API来创建和管理线程。
- **GNU Portable Threads (GNU Pth):** GNU Pth是一种为POSIX/UNIX系统设计的用户级线程库。它提供了一套非抢占式的线程调度机制，这意味着线程间的切换只会在明确的调度点发生。
- **Green threads:** Green threads是Java虚拟机（JVM）在早期版本中使用的一种用户级线程模型。在这种模型中，所有的线程都是在JVM中模拟出来的，而不是直接映射到操作系统的原生线程。
- **Fibers in Windows:** Windows操作系统提供了一种名为Fibers的用户级线程模型。Fibers提供了一种协同式的线程模型，线程切换只会在程序明确请求时发生。
- **Boost.Coroutine:** Boost.Coroutine 是 C++ Boost 库中提供的一个用户级线程（协程）实现。
- **Go routines in Go language:** Go语言提供了一种名为goroutines的轻量级线程模型，尽管goroutines在语言层面被实现，但其行为类似于用户级线程。

# 线程的实现方式

## 内核级线程(kernel-level thread)

- 由内核通过系统调用实现的线程机制，由内核完成线程的创建、终止和管理。
- 由内核维护线程控制块TCB, 在内核实现。





# 内核级线程的特点

- 线程由操作系统内核直接支持和管理；
- 每个内核级线程都有自己在内核中的数据结构（例如线程控制块），其中包含了如线程状态、优先级、调度信息等重要数据；
- 内核级线程可以直接由操作系统调度，因此它们可以在多处理器系统上并行运行
- 如果一个内核级线程阻塞（例如等待I/O操作完成），操作系统可以立即调度同一进程中的另一个线程运行；
- 内核级线程的创建、销毁、同步和切换都需要进行系统调用，因此这些操作的开销相对较大。





# 支持内核级线程的OS

- **Linux:** Linux操作系统从一开始就支持内核级线程。在现代的Linux中，通过Native POSIX Thread Library (NPTL) 或者早期的 LinuxThreads 库，都能实现POSIX标准的线程模型。
- **Windows:** Windows操作系统也支持内核级线程。Windows API提供了一套丰富的线程管理函数，如CreateThread、ExitThread等。
- **Solaris:** Solaris操作系统是早期支持多线程的UNIX操作系统之一。它提供了一种被称为“轻量级进程”（LWP）的机制，这实际上就是内核级线程。
- **FreeBSD:** FreeBSD通过KSE (Kernel Scheduled Entities) 实现内核级线程。
- **Mac OS X:** Mac OS X使用了一种被称为XNU的混合内核支持内核级线程，该内核结合了Mach和FreeBSD的特性。







# 线程的实现方式

- 混合线程模型
  - Java虚拟机（JVM）采用了一种混合线程模型，可以兼顾内核级线程和用户级线程的优点
  - JVM可以创建多个内核级线程，并在每个内核级线程上运行多个用户级线程
  - JVM就可以利用多处理器系统的并行能力，同时也可以通过在用户空间进行线程切换来减小开销





# 多线程模型

- 多对一模型
- 一对一模型
- 多对对模型





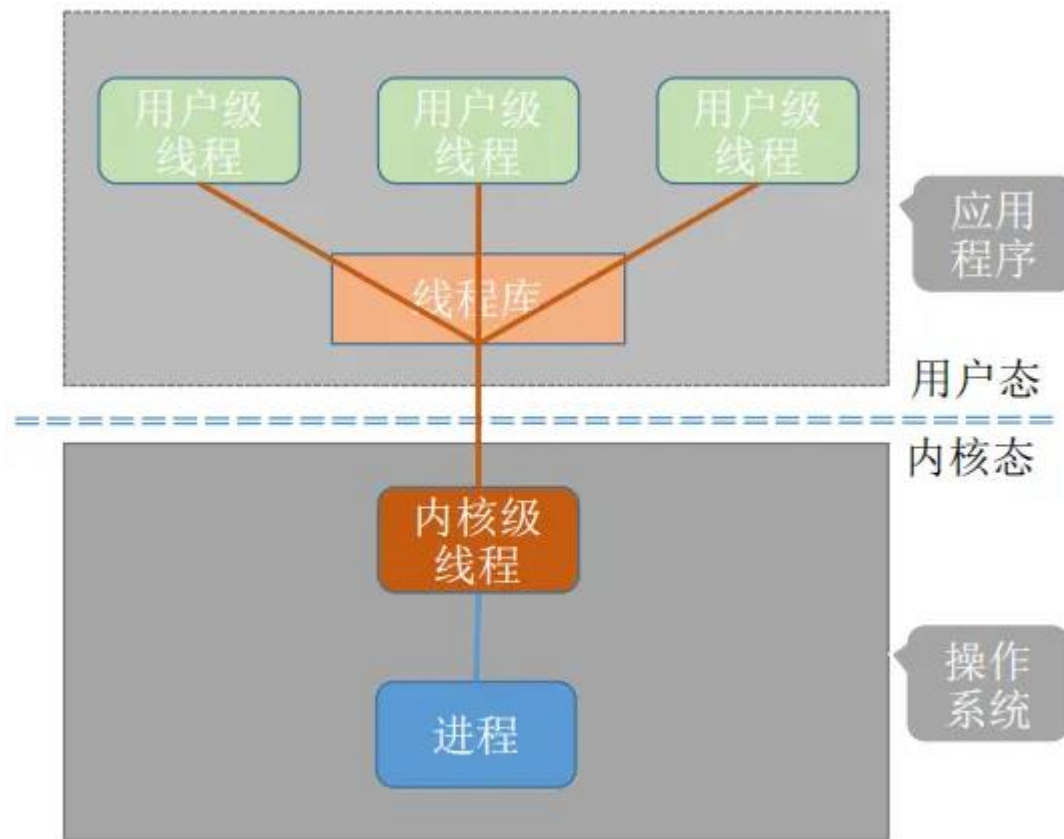
# 多对一模型

- 多个用户级线程映射到单个内核线程
- 一个线程的阻塞会导致所有线程都被阻塞
- 在多核系统上，多个线程可能无法并行运行，因为一次只能有一个线程在内核中运行
- 目前很少有系统使用这种模型
- 示例
  - Solaris Green Threads
  - GNU Portable Threads





# 多对一模型





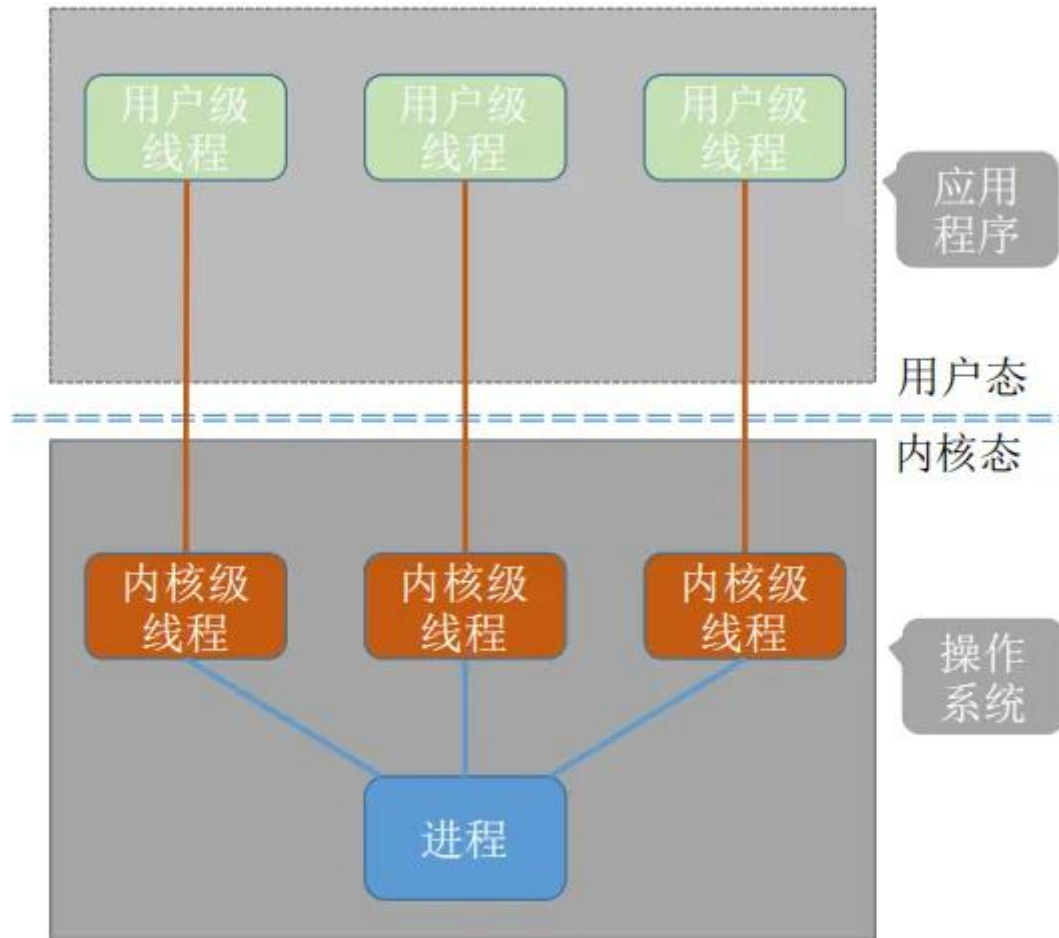
# 一对一模型

- 每个用户级线程映射到内核线程
- 创建一个用户级线程会创建一个内核线程
- 比多对一模型具有更多的并发性
- 由于开销的原因，每个进程中的线程数量有时会受到限制
- 示例
  - Windows
  - Linux





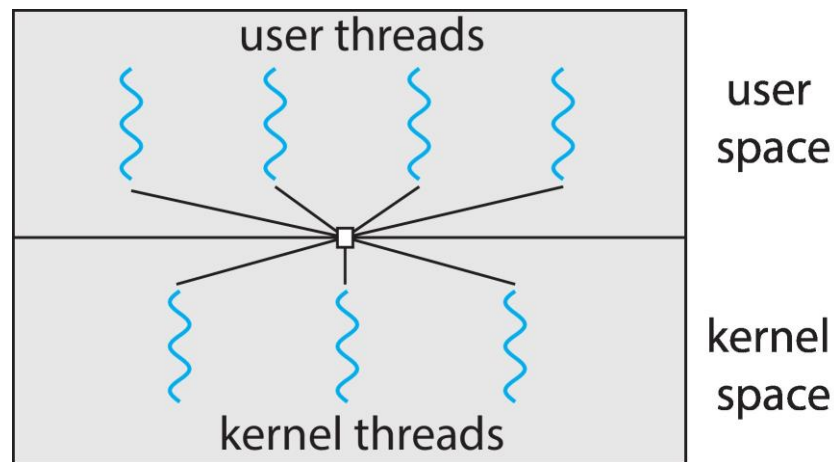
# 一对一模型





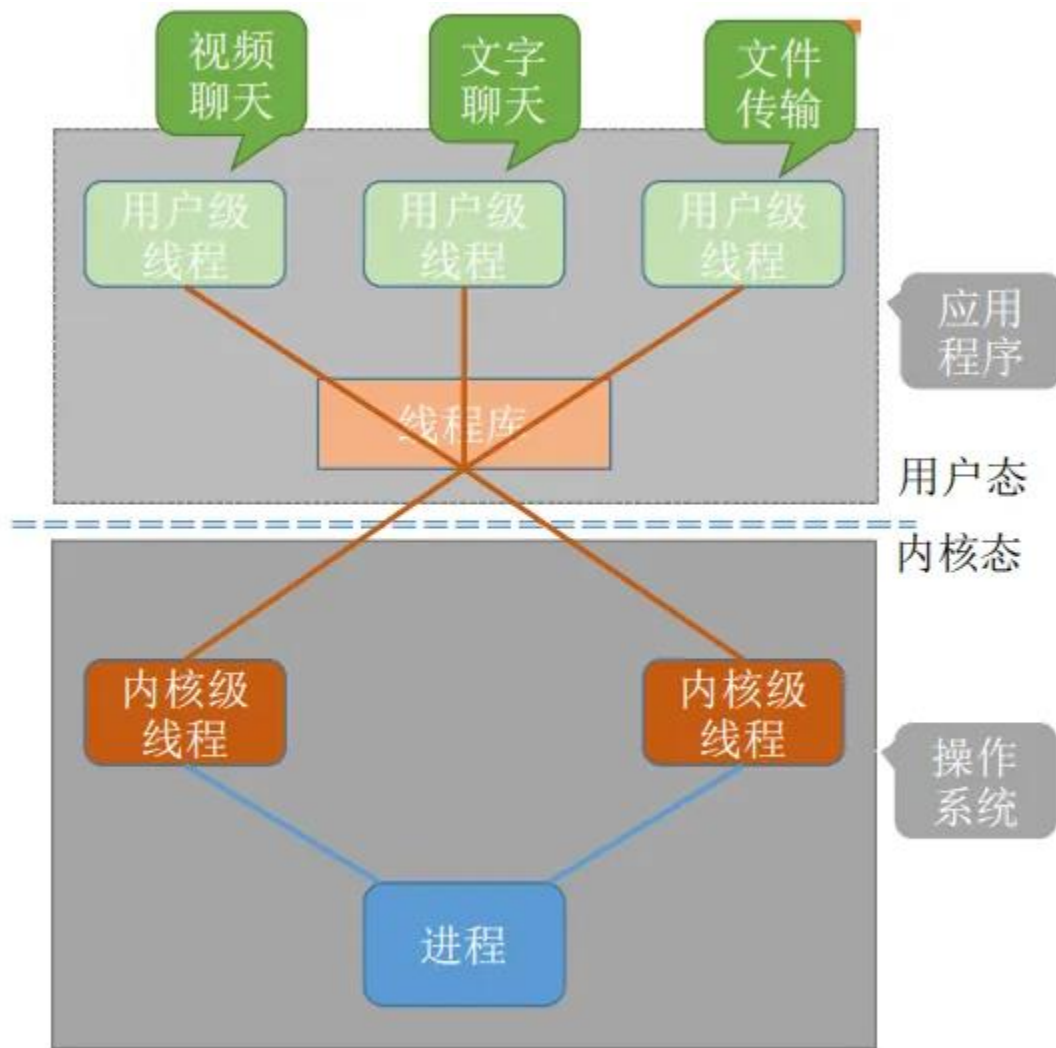
# 多对多模型

- 允许多个用户级线程映射到多个内核线程
- 允许操作系统创建足够数量的内核线程
- Windows使用的ThreadFiber包
- 否则不太常见





# 多对多模型

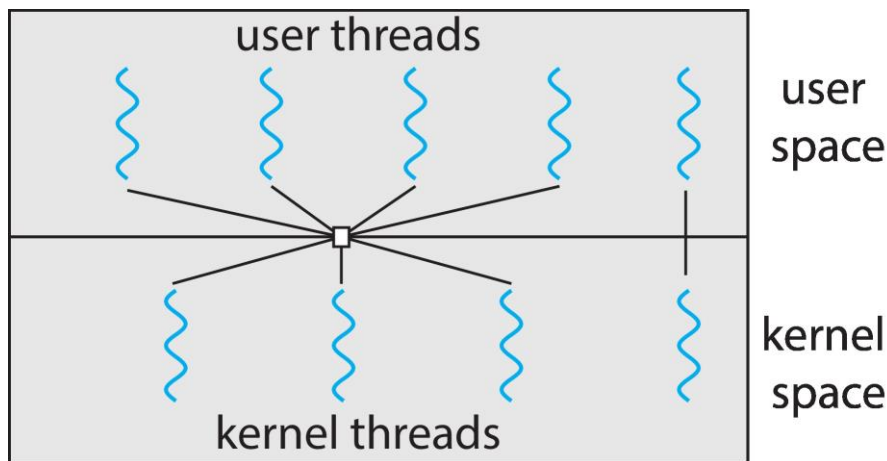






# 两级模型

- 类似于多对多模型
- 用户级线程可以被映射到任意数量的内核级线程上。这意味着一个用户级线程可以对应一个内核级线程，也可以对应多个内核级线程
- 该模型提供了更大的灵活性，可以根据应用程序的需要进行优化，可以利用到多处理器的并行性





# 线程的上下文切换

- 线程是调度的基本单位，而进程则是资源拥有的基本单位。
- 不同进程中的线程切换：进程上下文切换
- 相同进程中的线程切换：虚拟内存等进程资源保持不动，只需要切换线程的私有数据、寄存器等不共享的数据





# 线程库

- 线程库为程序员提供了用于创建和管理线程的API
- 两种主要的实现线程库的方式
  - 完全在用户空间运行的库
  - 由操作系统支持的内核级库





# pthread

- 可能作为用户级或内核级提供
- 遵照POSIX标准(IEEE 1003.1c)提出的一套线程接口
- 只是一种规范，而非实现
- pthreads只对接口进行了定义，不同的OS根据自己的需求提供了实现
  - 在UNIX操作系统（Linux和Mac OS X）中实现了pthreads规范
  - Windows下有第三方实现



# pthread基本接口

- 线程创建

`pthread_create(thread, attr, start_routine, arg);`



- 线程退出

`pthread_exit(retval);`

- 出让资源

`pthread_yield();`

- 合并操作

`pthread_join(thread, retval);`

- 挂起与唤醒

`sleep(seconds);`

`pthread_cond_wait(cond, mutex);`





# Pthreads 示例

```
#include <pthread.h>
#include <stdio.h>

#include <stdlib.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
    pthread_t tid; /* the thread identifier */
    pthread_attr_t attr; /* set of thread attributes */

    /* set the default attributes of the thread */
    pthread_attr_init(&attr);
    /* create the thread */
    pthread_create(&tid, &attr, runner, argv[1]);
    /* wait for the thread to exit */
    pthread_join(tid, NULL);

    printf("sum = %d\n", sum);
}
```





# Pthreads 示例

```
/* The thread will execute in this function */  
void *runner(void *param)  
{  
    int i, upper = atoi(param);  
    sum = 0;  
  
    for (i = 1; i <= upper; i++)  
        sum += i;  
  
    pthread_exit(0);  
}
```





# Pthreads代码

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
    pthread_join(workers[i], NULL);
```







# Windows下的多线程代码

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */

/* The thread will execute in this function */
DWORD WINAPI Summation(LPVOID Param)
{
    DWORD Upper = *(DWORD*)Param;
    for (DWORD i = 1; i <= Upper; i++)
        Sum += i;
    return 0;
}
```





# Windows下的多线程代码

```
int main(int argc, char *argv[])
{
    DWORD ThreadId;
    HANDLE ThreadHandle;
    int Param;

    Param = atoi(argv[1]);
    /* create the thread */
    ThreadHandle = CreateThread(
        NULL, /* default security attributes */
        0, /* default stack size */
        Summation, /* thread function */
        &Param, /* parameter to thread function */
        0, /* default creation flags */
        &ThreadId); /* returns the thread identifier */

    /* now wait for the thread to finish */
    WaitForSingleObject(ThreadHandle, INFINITE);

    /* close the thread handle */
    CloseHandle(ThreadHandle);

    printf("sum = %d\n", Sum);
}
```





# 练习题

- 3.1 Using the program shown in below, explain what the output will be at LINE A.

---

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

int value = 5;

int main()
{
    pid_t pid;

    pid = fork();

    if (pid == 0) { /* child process */
        value += 15;
        return 0;
    }
    else if (pid > 0) { /* parent process */
        wait(NULL);
        printf("PARENT: value = %d",value); /* LINE A */
        return 0;
    }
}
```

---

**Figure 3.30** What output will be at Line A?





# 练习题

- 3.2 Including the initial parent process, how many processes are created by the program shown in Figure 3.31?

---

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

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

    /* fork another child process */
    fork();

    /* and fork another */
    fork();

    return 0;
}
```

---

**Figure 3.31** How many processes are created?





# 练习题

---

- 3.5 When a process creates a new process using the `fork()` operation, which of the following states is shared between the parent process and the child process?
  1. Stack
  2. Heap
  3. Shared memory segments





# 练习题

---

- Please Describe the actions taken by a kernel to context-switch between processes.
- What are two differences between user-level threads and kernel-level threads? Under what circumstances is one type better than the other?





# 选择题1

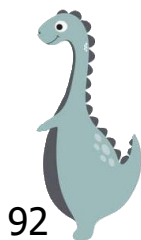
- 对进程的管理和控制使用\_\_\_\_\_。
  - A. 指令
  - B. 信号量
  - C. 原语
  - D. 信箱
- 分配到必要的资源并获得处理机时的进程状态是\_\_\_\_\_。
  - A. 就绪状态
  - B. 撤消状态
  - C. 执行状态
  - D. 阻塞状态





## 选择题2

- 下列进程状态变化中，\_\_\_\_\_变化是不可能发生的。
  - A.等待→就绪      B.等待→运行
  - C.运行→等待      D.运行→就绪
- 当\_\_\_\_\_时，进程从执行状态转变为就绪状态。
  - A.等待的事件发生      B.时间片到
  - C.等待某一事件      D.进程被调度程序选中







## 选择题3

- 下面对进程的描述中，错误的是\_\_\_\_\_。
  - A. 进程是有生命期的    B. 进程执行需要处理机
  - C. 进程是指令的集合    D. 进程是动态的概念
  
- 如果系统中有 $n$ 个进程，则就绪队列中进程的个数最多为\_\_\_\_\_。
  - A.  $n$                       B. 1                      C.  $n-1$                       D.  $n+1$





# 选择题4

- 操作系统通过\_\_\_\_\_对进程进行管理。
  - A. JCB      B. PCB
  - C. DCT      D. CHCT
- 下面所述步骤中，\_\_\_\_\_不是创建进程所必需的。
  - A. 建立一个进程控制块
  - B. 为进程分配内存
  - C. 将进程控制块链入就绪队列
  - D. 由调度程序为进程分配CPU



# 选择题5

- 下述哪一个选项，体现了原语的主要特点\_\_\_\_\_。
  - A. 并发性                  B. 异步性
  - C. 不可分割性          D. 共享性
- 下面对父进程和子进程的叙述不正确的是\_\_\_\_\_。
  - A. 撤消父进程之时，可以同时撤消其子进程
  - B. 父进程和子进程之间可以并发
  - C. 父进程可以等待所有子进程结束后再执行
  - D. 父进程创建了子进程，因此父进程执行完了子进程才能运行





# 选择题6

- 下列几种关于进程的叙述中，最不符合操作系统对进程理解的是\_\_\_\_\_。
  - A. 进程是在多程序并行环境中的完整的程序
  - B. 进程可以由程序，数据和进程控制块描述
  - C. 线程(Thread)是一种特殊的进程
  - D. 进程是程序在一个数据集合上运行的过程，是系统进行资源分配和调度的一个独立单位





# 选择题7

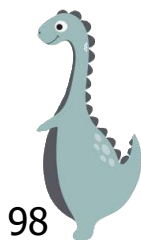
- 当一个进程处于\_\_\_\_\_的状态时，称其为等待状态
  - A. 它正等待调度
  - B. 它正等着协作进程的一个消息
  - C. 它正等分给它一个时间片
  - D. 它正等进入内存
- 进程从执行状态到阻塞状态可能是由于\_\_\_\_\_。
  - A. 进程调度程序的调度
  - B. 现运行进程的时间片用完
  - C. 现运行进程执行了P操作
  - D. 现运行进程执行了V操作





# 选择题8

- 一个进程被唤醒意味着\_\_\_\_\_。
  - A. 该进程重新占有了CPU
  - B. 进程状态变为就绪
  - C. 它的优先权变为最大
  - D. 其PCB移至就绪队列的队首
- 一个进程基本状态可以从其他两种基本状态转变过来，这个基本状态是\_\_\_\_\_。
  - A. 执行状态      B. 阻塞状态
  - C. 就绪状态      D. 撤销状态





# 选择题9

- 设系统中有 $n$  ( $n > 2$ ) 个进程，且当前不在执行进程调度程序，试考虑下述4种情况：
  1. 有1个运行进程， $n-1$ 个就绪进程，没有进程处于等待状态。
  2. 有1个运行进程，没有就绪进程， $n-1$ 进程处于等待状态。
  3. 有1个运行进程，有1个就绪进程， $n-2$ 进程处于等待状态。
  4. 没有运行进程，有2个就绪进程， $n$ 个进程处于等待状态上述情况中，不可能发生的情况是\_\_\_\_\_。





# 选择题10

- 下面关于进程的叙述中，不正确的有 \_\_\_\_\_ 条。
  - ① 进程申请CPU得不到满足时，其状态变为等待状态。
  - ② 在单CPU系统中，任一时刻都有一个进程处于运行状态。
  - ③ 优先级是进行进程调度的重要依据，一旦确定不能改变。
  - ④ 进程获得处理机而运行是通过调度而实现的。







# 考研题1

- 下列选项中，导致创建新进程的操作是\_\_\_\_\_。

I 用户登录成功      II 设备分配

III 启动程序执行

A. 仅 I 和 II

B. 仅 II 和 III

C. 仅 I 和 III

D. I、II、III

- 下列选项中，在用户态执行的是\_\_\_\_\_。

A、命令解释程序    B、缺页处理程序

C、进程调度程序    D、时钟中断处理程序





## 考研题2

- 下列选项中，不可能在用户态发生的事件是（ ）。
- A. 系统调用                      B. 外部中断
- C. 进程切换                      D. 缺页

