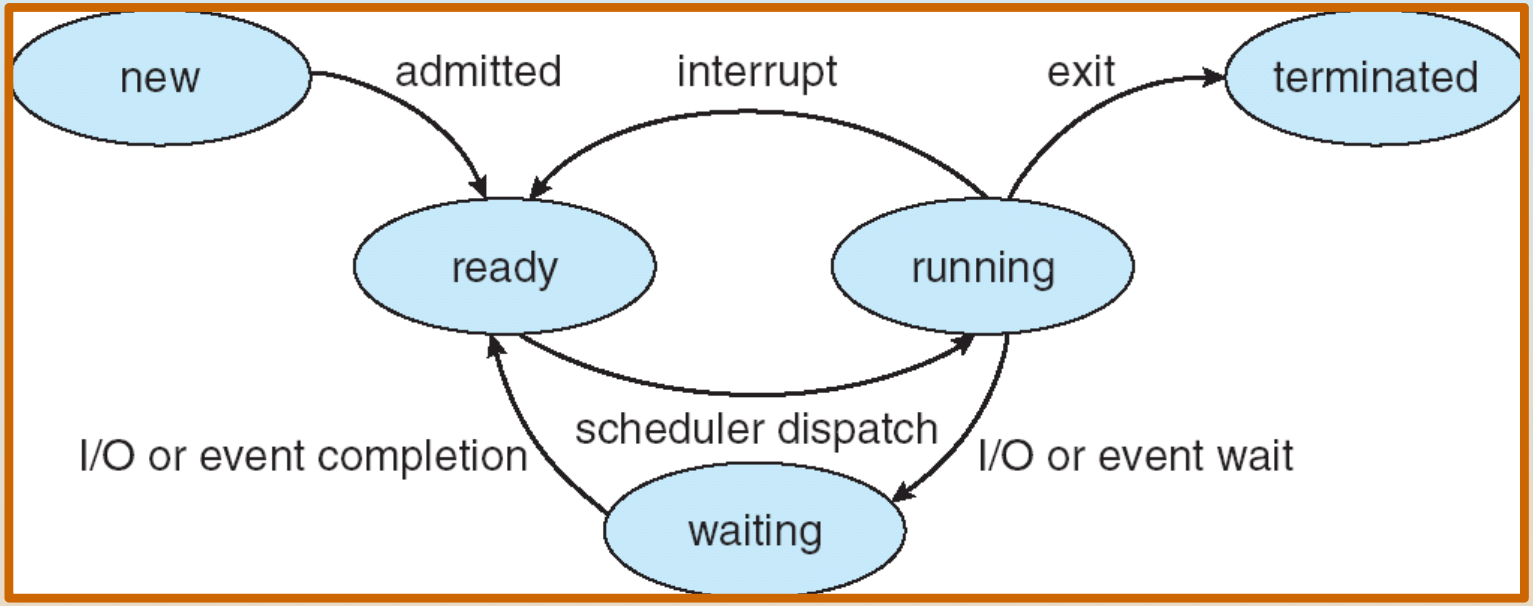
# OS如何运行和管理一个任务?

# 核心角色：Processes

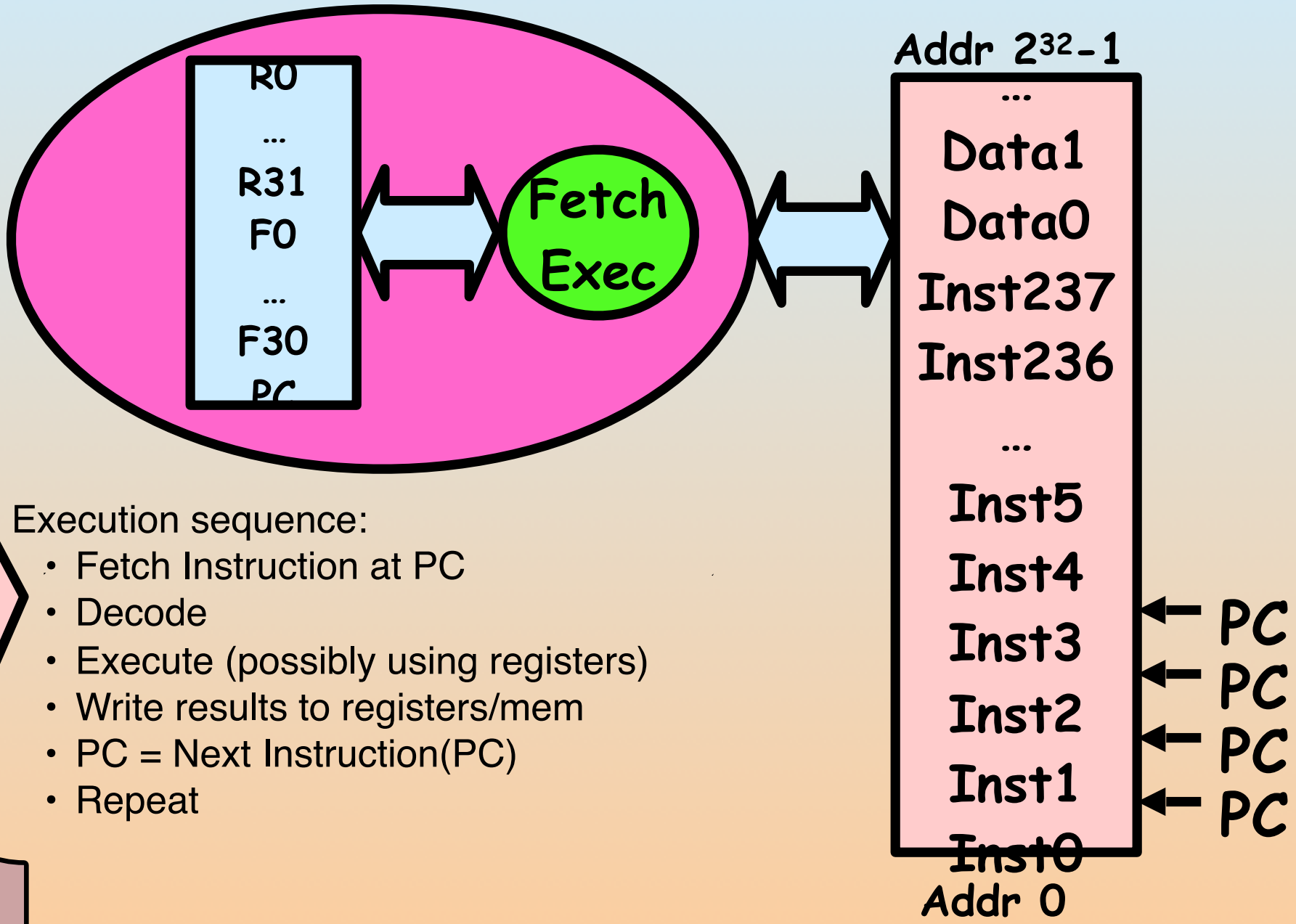
- Process – a program in execution; process execution must progress in sequential fashion
- Textbook uses the terms job and process almost interchangeably
- As a process executes, its state changes:
  - new:  The process is being created
  - running:  Instructions are being executed
  - waiting:  The process is waiting for some event to occur
  - ready:  The process is waiting to be assigned to a process
  - terminated:  The process has finished execution

# Diagram of Process State




- As a process executes, it changes state
  - new: The process is being created
  - ready: The process is waiting to run
  - running: Instructions are being executed
  - waiting: Process waiting for some event to occur
  - terminated: The process has finished execution

# What happens during execution?

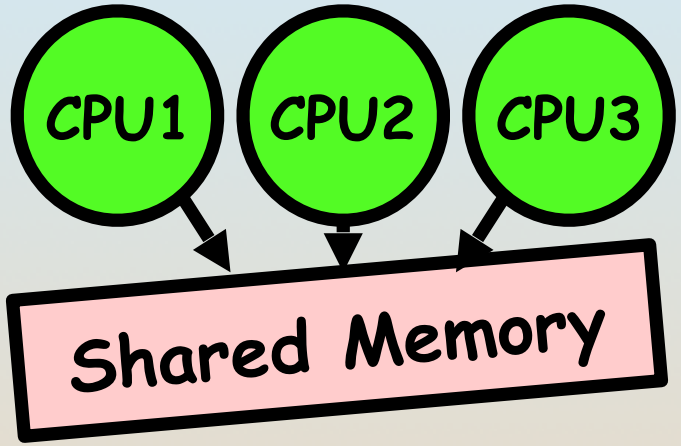


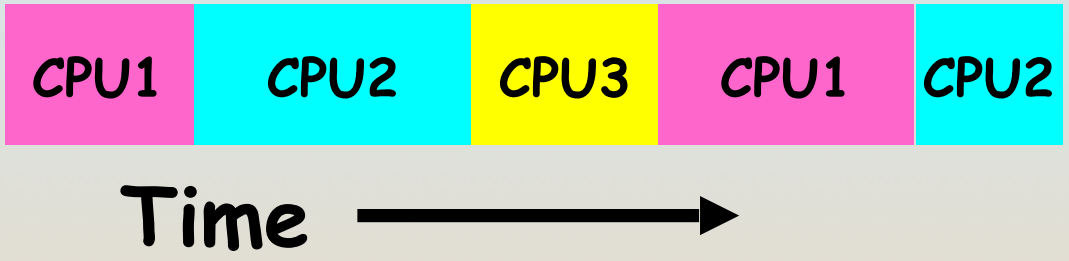

Execution sequence:

- Fetch Instruction at PC
- Decode
- Execute (possibly using registers)
- Write results to registers/mem
- PC = Next Instruction(PC)
- Repeat


专属休息室
对于INBOX(输入栏)中的每两个数:
如果它们符号相同(都是正数或负数)，就把 0 送到OUTBOX(输出栏)里。
如果它们符号不同，就把 1 送到OUTBOX(输出栏)里。
重复，直到清空INBOX(输入栏)为止。
inbox (输入栏)
outbox (输出栏)
copyfrom (拷贝自)
copyto (拷贝至)
add (相加)
sub (相减)
jump (跳转)
jump if zero (0跳转)
jump if negative (负跳转)
...
INBOX (输入栏)
OUTBOX (输出栏)
-3
2
8
-3
5
-3
-2
0
1
撤销
复制
粘贴
清空

# 一个简单示例：OS是如何管理资源的?




- Assume a single processor. How do we provide the illusion of multiple processors?
  - Multiplex in time!
- Each virtual "CPU" needs a structure to hold:
  - Program Counter (PC), Stack Pointer (SP)
  - Registers (Integer, Floating point, others…?)
- How switch from one CPU to the next?
  - Save PC, SP, and registers in current state block
  - Load PC, SP, and registers from new state block
- What triggers switch?
  - Timer, voluntary yield, I/O, other things

# 一个简单示例实现：Simple multiprogramming

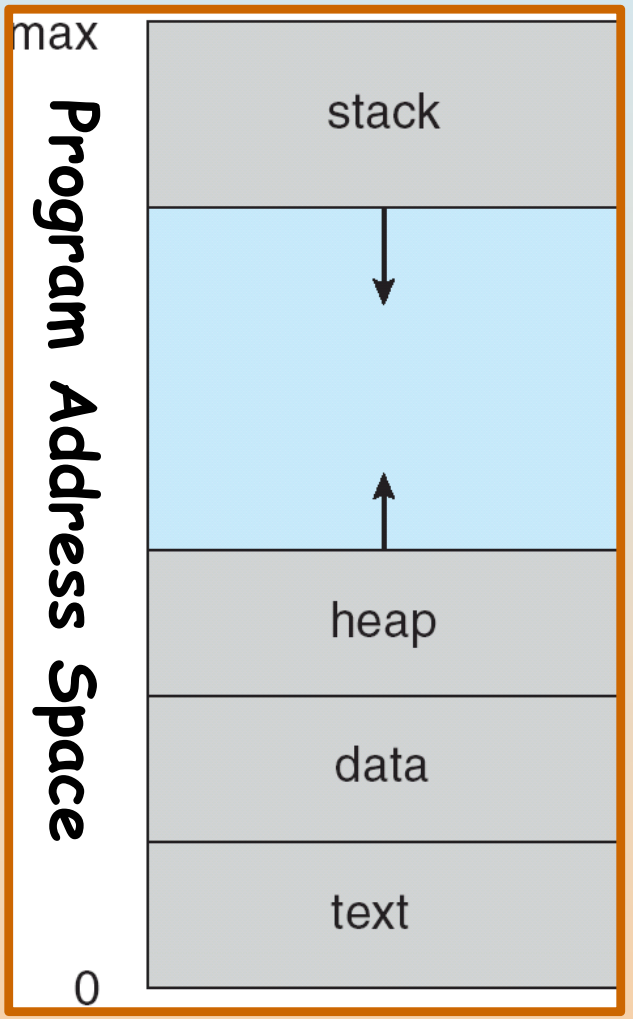
- All virtual CPUs share same non-CPU resources
  - I/O devices the same
  - Memory the same
- Consequence of sharing:
  - Each thread can access the data of every other thread (good for sharing, bad for protection)
  - Threads can share instructions (good for sharing, bad for protection)
  - Can threads overwrite OS functions?
- This (unprotected) model common in:
  - Embedded applications
  - switch with both yield and/or timer

# 程序地址空间（Program's Address Space）

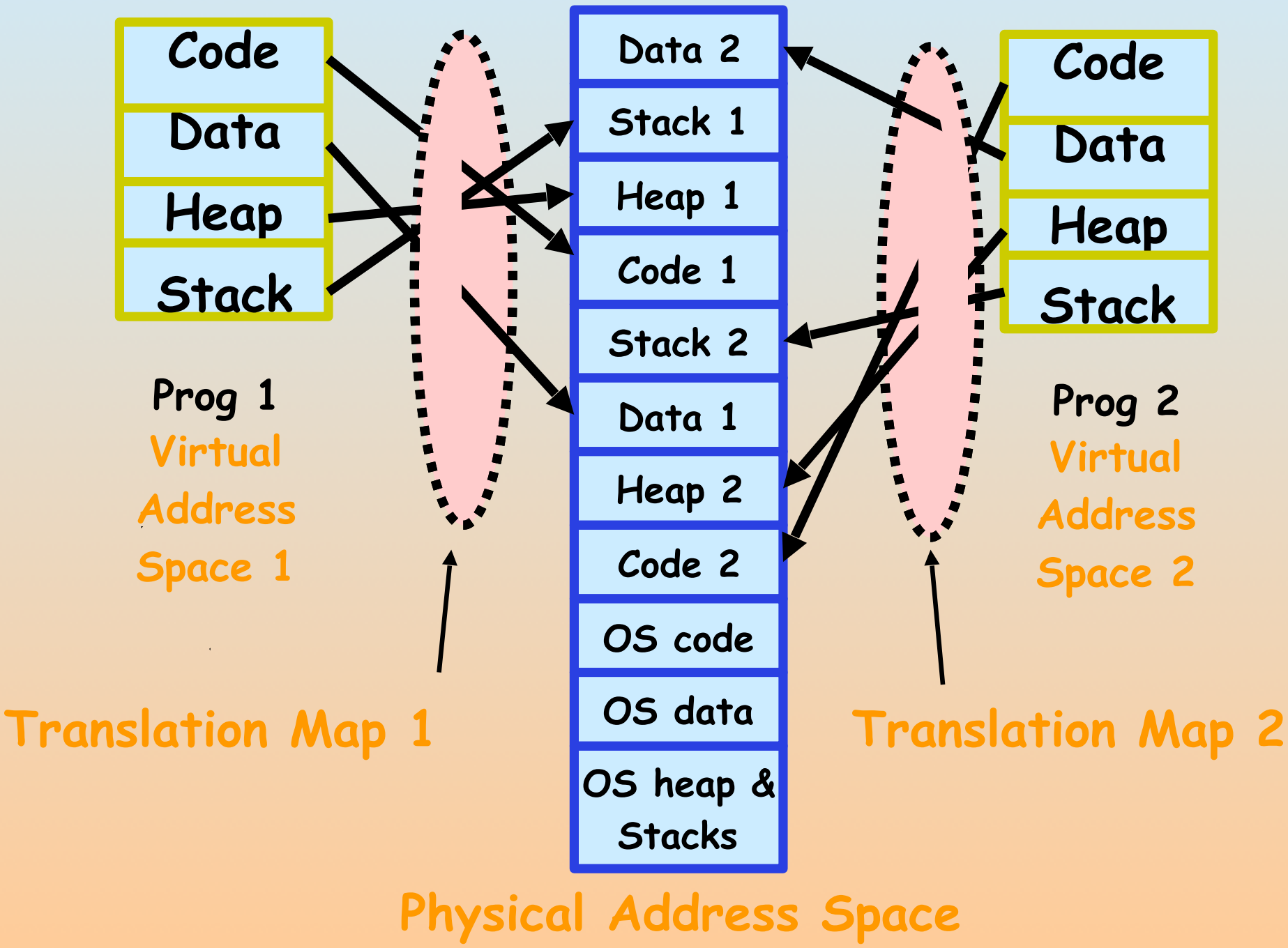
- 兵马未动，粮草先行：

程序代码和数据都需要空间，各类资源也需要可被寻址

- Address space ⇒ the set of accessible addresses + state associated with them:
  - For a 32-bit processor there are 2^32 = 4 billion addresses
- What happens when you read or write to an address?
  - Perhaps Nothing
  - Perhaps acts like regular memory
  - Perhaps ignores writes
  - Perhaps causes I/O operation
    - (Memory-mapped I/O)
  - Perhaps causes exception (fault)

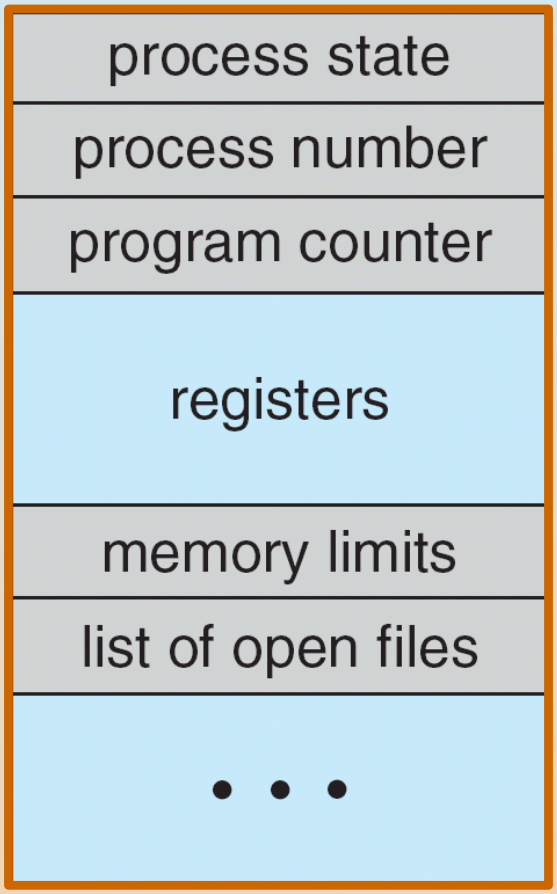


# 一个简单示例实现：地址转换与映射



# How do we multiplex processes?

- The current state of process held in a process control block (PCB):
  - This is a "snapshot" of the execution and protection environment
  - Only one PCB active at a time
- Give out CPU time to different processes (Scheduling):
  - Only one process "running" at a time
  - Give more time to important processes
- Give pieces of resources to different processes (Protection):
  - Controlled access to non-CPU resources
  - Sample mechanisms:
    - Memory Mapping: Give each process their own address space
    - Kernel/User duality: Arbitrary multiplexing of I/O through system calls



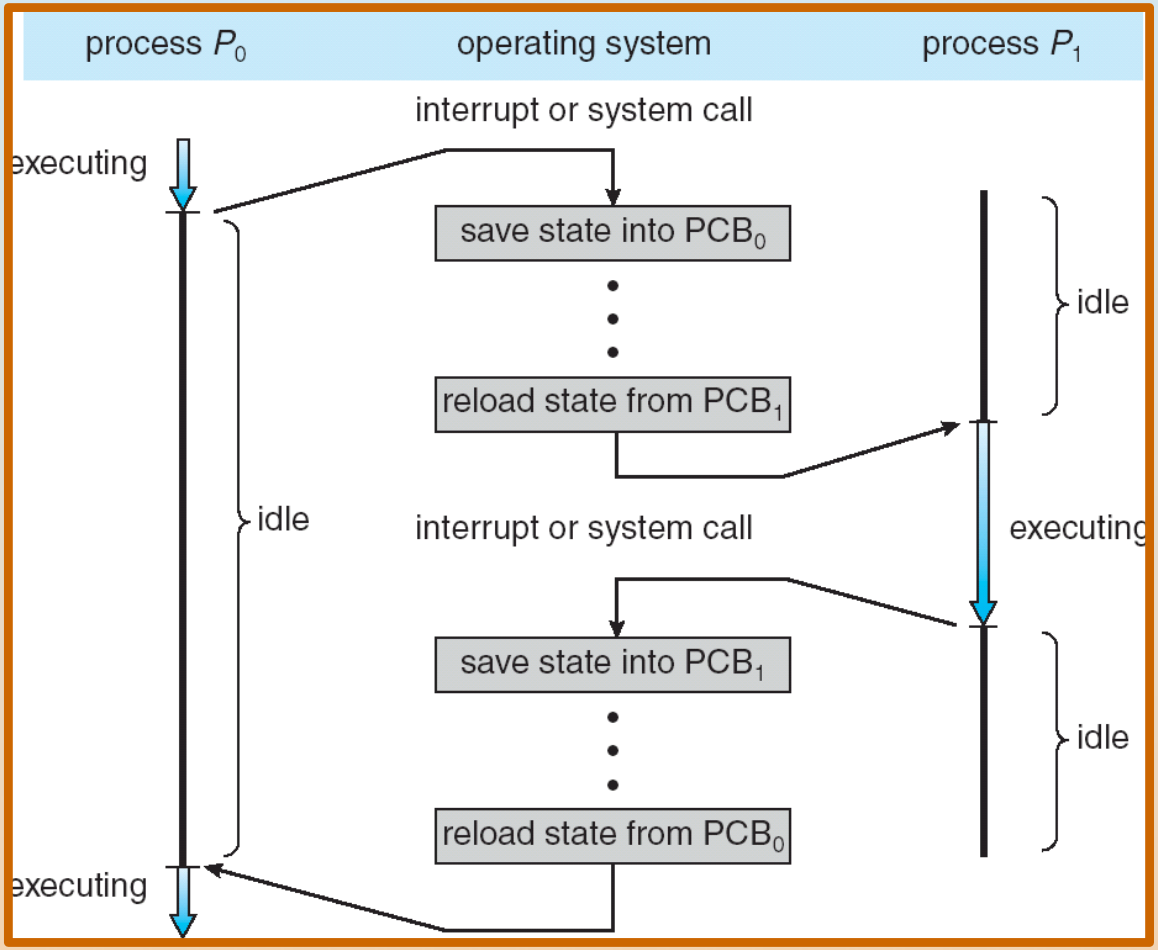

Process Control Block

# Process Control Block (PCB)


兵马未动，粮草先行：

程序代码和数据需要空间，各类资源需要可被寻址

# CPU Switch From Process to Process




- This is also called a “context switch”
- Code executed in kernel above is overhead
  - Overhead sets minimum practical switching time
  - Less overhead with SMT/hyperthreading, but… contention for resources instead

# 中断

- 程序执行过程中，当发生某个事件时，中止CPU上现行程序的运行，引出处理该事件的程序执行的过程。
  - 能充分发挥处理机的使用效率
  - 提高系统的实时处理能力

# 中断的分类

- 从中断事件的性质和激活的手段分类：
  - 强迫中断：强迫性中断事件不是正在运行的程序所期待的，而是由于某种事故或外部请求信息所引起的，分为：机器故障中断事件；程序性中断事件；外部中断事件；输入输出中断事件。
  - 自愿中断：自愿性中断事件是正在运行的程序所期待的事件。正在运行的程序对操作系统有某种需求，一旦机器执行到一条访管指令时，便自愿停止现行程序的执行而转入访管中断处理程序处理。

- 按照中断信号的来源分类：
  - 外中断(又称中断)：指来自处理器和主存之外的中断。外中断包括电源故障中断、时钟中断、控制台中断、它机中断和I/O中断等。不同的中断具有不同的中断优先级，处理高一级中断时，往往会屏蔽部分或全部低级中断。
  - 内中断(又称异常)：指来自处理器和主存内部的中断。内中断包括：通路校验错、主存奇偶错、非法操作码、地址越界、页面失效、调试指令、访管中断、算术操作溢出等各种程序性中断。异常是不能被屏蔽的，一旦出现应立即响应并加以处理。

# 中断

- 中断是由与现行指令无关的中断信号触发的(异步的)，且中断的发生与CPU处在用户模式或内核模式无关，在两条机器指令之间才可响应中断，一般来说，中断处理程序提供的服务不是为当前进程所需的，如时钟中断、硬盘读写服务请求中断；
- 异常是由处理器正在执行现行指令而引起的，一条指令执行期间允许响应异常，异常处理程序提供的服务是为当前进程所用的。异常包括很多方面，有出错(fault)，也有陷入(trap)。

- 硬中断和软中断：
  - 中断和异常要通过硬件设施来产生中断请求，可看作硬中断。
  - 不必由硬件发信号而能引发的中断称软中断，软中断是利用硬件中断的概念，用软件方式进行模拟，实现宏观上的异步执行效果。
    - 软中断是由内核或进程对某个进程发出的中断信号，可看作内核与进程或进程与进程之间用来模拟硬中断的一种信号通信方式。

# 中断处理

- 机器故障中断事件的处理:事件是由硬件故障(例如，电源故障、主存储器故障)产生。中断处理能做的工作是：保护现场，防止故障蔓延，报告给操作员并提供故障信息以便维修和校正，对程序中所造成的破坏进行估价和恢复。
- 程序性中断事件的处理:采用中断续元处理来进行程序性中断事件的处理
- 外部中断事件的处理:时钟是操作系统进行调度工作的重要工具，例如让分时进程作时间片轮转、让实时进程
- 控制台中断事件的处理：操作员可以利用控制台开关请求操作系统工作，当使用控制台开关后，就产生一个控制台中断事件通知操作系统。操作系统处理这种中断就如同接受一条操作命令一样，转向处理操作命令的程序执行。
- I/O中断的处理

# 中断优先级和多重中断

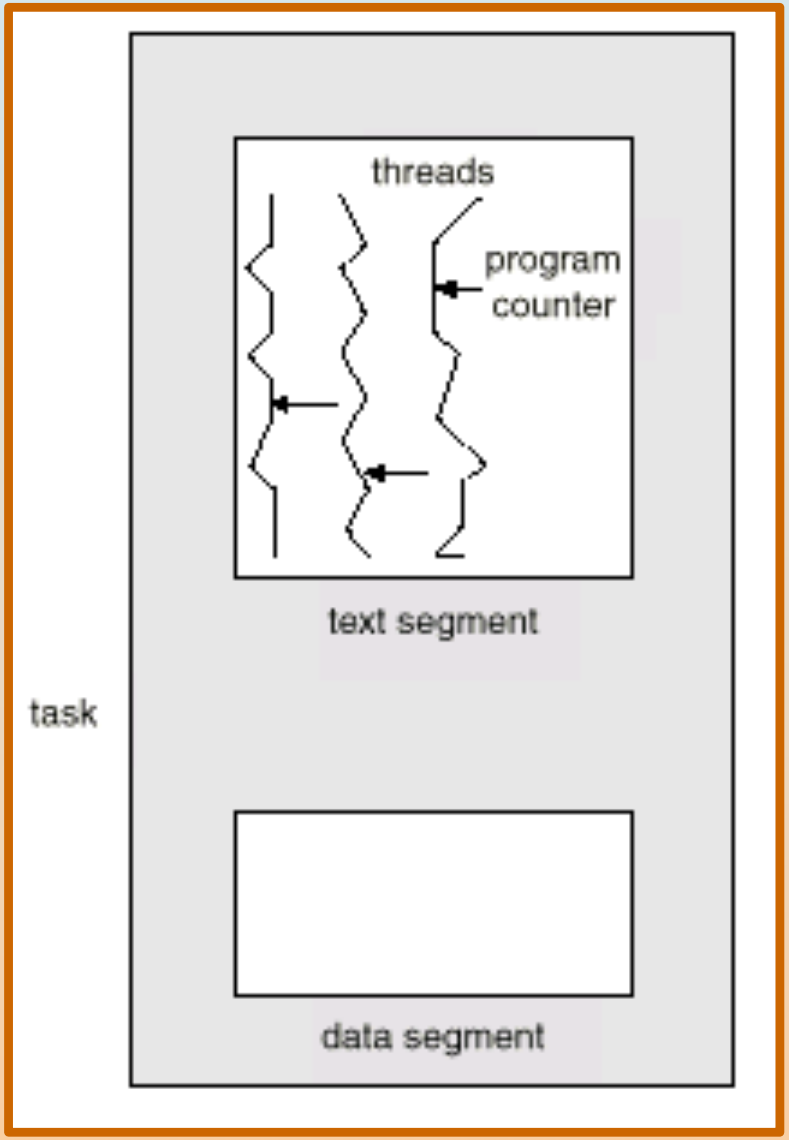
- 中断的屏蔽：主机可允许或禁止某类中断的响应，如允许或禁止所有的I/O中断、外部中断、及某些程序性中断。有些中断是不能被禁止的，例如，计算机中的自愿性访管中断就不能被禁止。

- 多重中断事件的处理：中断正在进行处理期间，这时CPU又响应了新的中断事件，于是暂时停止正在运行的中断处理程序，转去执行新的中断处理程序，这就叫多重中断（又称中断嵌套）
  - 禁止再发生中断
  - 定义中断优先级
  - 响应并进行中断处理:

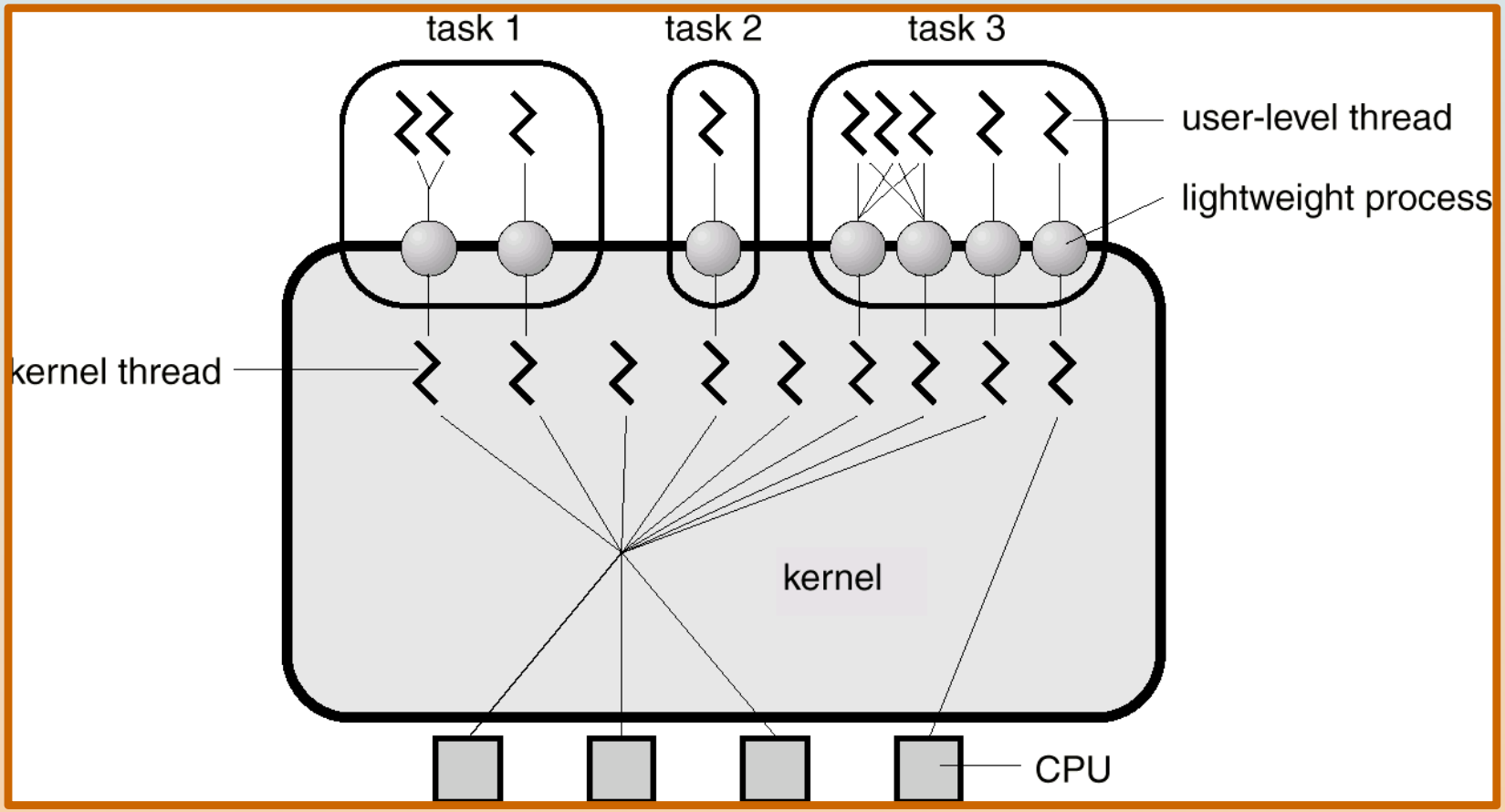
# Threads

- A thread (or lightweight process) is a basic unit of CPU utilization; it consists of:
  - program counter
  - register set
  - stack space
- A thread shares with its peer threads its:
  - code section
  - data section
  - operating-system resources
  - collectively known as a job.
- A traditional or heavyweight process is equal to a job with one thread

# Multiple Threads within a job



# Solaris 2 Threads



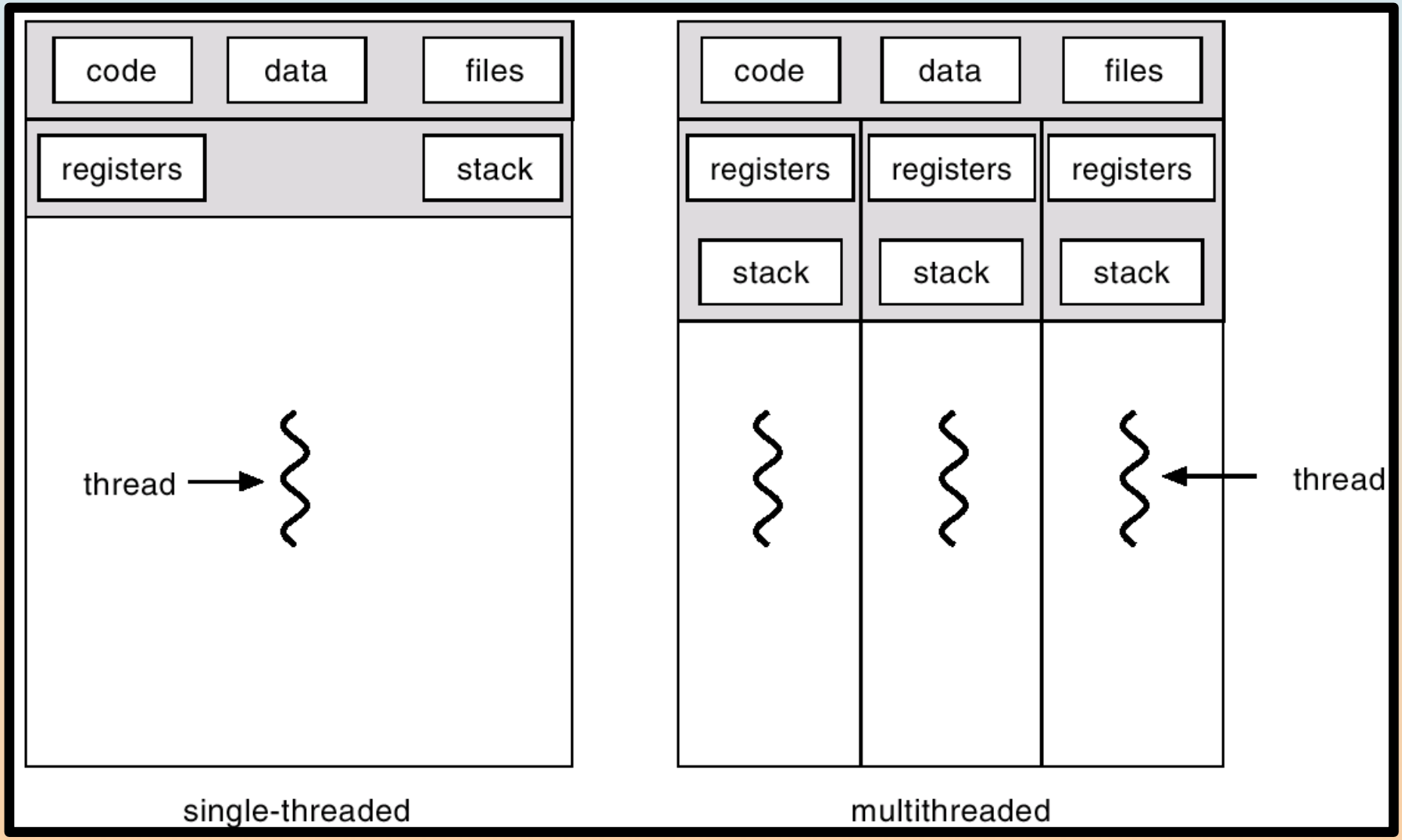
# Threads (Cont.)

- In a multiple threaded job, while one server thread is blocked and waiting, a second thread in the same job can run.
  - Cooperation of multiple threads in same job confers higher throughput and improved performance.
  - Applications that require sharing a common buffer (i.e., producer-consumer) benefit from thread utilization.
- Threads provide a mechanism that allows sequential processes to make blocking system calls while also achieving parallelism.
- Kernel-supported threads (Mach and OS/2).
- User-level threads; supported above the kernel, via a set of library calls at the user level (Project Andrew from CMU).
- Hybrid approach implements both user-level and kernel-supported threads (Solaris 2).

# 线程

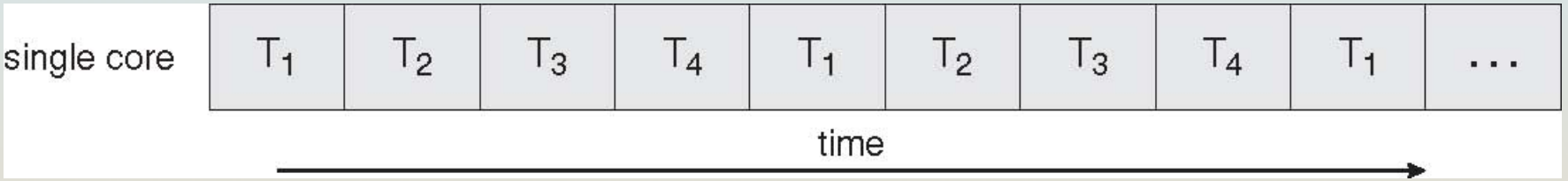
- 线程(*thread*),有时也被称为轻量级进程(lightweight process , LWP)
  - 线程是CPU使用的基本单元
  - 线程由如下部分组成:
    - 线程ID
    - PC指针
    - 一组寄存器
    - 调用栈
  - 同一进程内的所有线程共享代码段、数据段和其它操作系统资源 （如文件、信号等）。

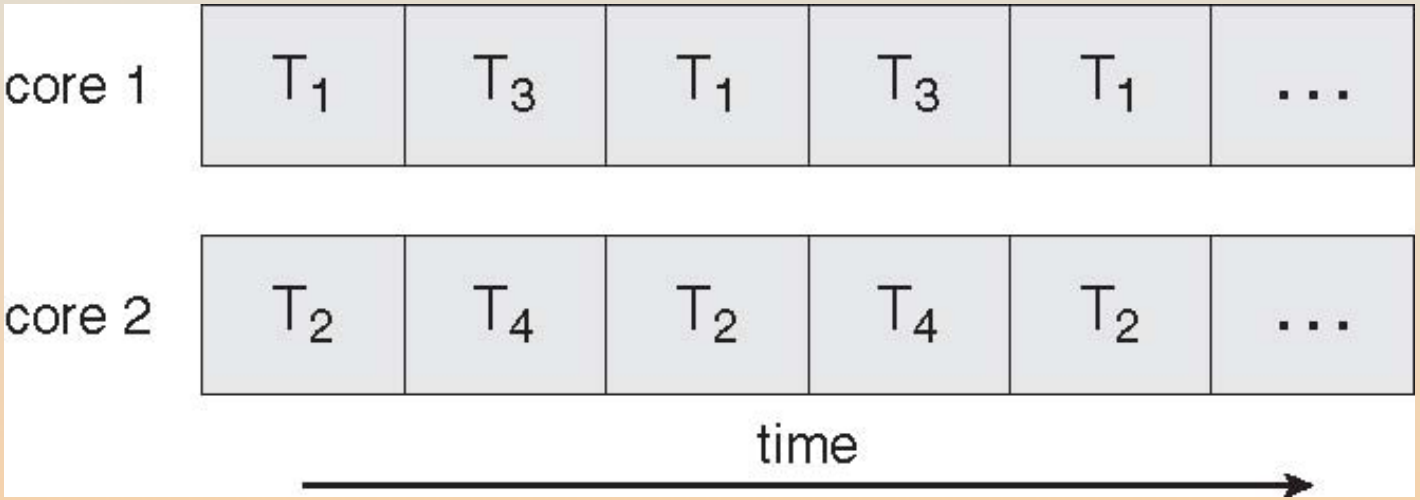
# 单线程进程和多线程进程



# 调度

## 线程


single core
T1
T2
T3
T4
T1
T2
T3
T4
T1
...
time


core 1
T1
T3
T1
T3
T1
...
core 2
T2
T4
T2
T4
T2
...
time

# 动机

- 应用程序通常实现得像一个具有多个控制线程的独立进程，例如：
  - 网页浏览器可能包括：
    - 显示图片和文字的线程
    - 从网络接收数据的线程
  - 字处理器可能包括：
    - 格式化显示文本和图像的线程
    - 读入用户键盘输入的线程
    - 拼写和语法检查的线程
  - 网页服务器可能包括：
    - 许多完成相似或者不同功能的不同线程

# 优点

- 响应度高: 如果对一个交互式应用程序采用多线程，即使其部分阻塞或执行冗长的操作，那么该程序仍能够继续执行，从而增加了对用户的响应度。

- 资源共享: 线程默认共享它们所属进程的内存和资源。

- 经济:创建和切换代价小
  - 例如在Solaris 2中
    - 进程创建要比线程创建慢30倍
    - 进程上下文切换要比线程上下文切换慢5倍

- 多处理器体系结构的利用
  - 单线程的进程只能运行在一个CPU上，无论系统中有多少个CPU。

## 2. 线程的属性

(1) 轻型实体。

(2) 独立调度和分派的基本单位。

(3) 可并发执行。

(4) 共享进程资源。

- 线程的性质：

（1）线程是进程内的一个相对独立的可执行单元。

（2）线程是操作系统中的基本调度单元，它含有调度所需的必要信息。

（3）每个进程在创建时，至少需要同时为该进程创建一个线程。也就是说进程中至少要有一个或一个以上线程，否则该进程无法被调度执行。

（4）线程可以创建其他线程。

（5）进程是资源分配的基本单元，同一进程内的多个线程共享该进程的资源。但线程并不拥有资源，只是使用它们。

（6）由于共享资源(包括数据和文件)，所以，线程间需要通信和同步机制。

（7）线程有生命期，在生命期中有状态的变化。

- 进程的两个基本属性：

（1）资源的拥有者：给每个进程分配一虚拟地址空间，保存进程映像；控制一些资源（文件，I/O设备）；有状态、优先级、调度。

（2）调度单位：进程是一个执行轨迹。

- 线程的引入：首先，不同应用中的并发任务要共享一个公共的地址空间和其他资源，只能将这些任务串行化，效率很低。其次，进程的创建、撤消和切换需要需要很大的开销，限制并发度的提高。
- 线程的定义：线程是进程内一个相对独立的、可调度的执行单元。它是进程中的一个实体，有时称轻量级进程。但资源的拥有者还是它的进程。线程将原来进程的两个属性分开处理。

- 线程与进程的比较：

（1）调度；

（2）并发性；

（3）拥有资源；

（4）系统开销。

## 3. 线程的状态

(1) 状态参数。

在OS中的每一个线程都可以利用线程标识符和一组状态参数进行描述。状态参数通常有这样几项:

① 寄存器状态，它包括程序计数器PC和堆栈指针中的内容;
② 堆栈，在堆栈中通常保存有局部变量和返回地址;
③ 线程运行状态，用于描述线程正处于何种运行状态;
④ 优先级，描述线程执行的优先程度;
⑤ 线程专有存储器，用于保存线程自己的局部变量拷贝;
⑥ 信号屏蔽，即对某些信号加以屏蔽。

(2) 线程运行状态。

如同传统的进程一样，在各线程之间也存在着共享资源和相互合作的制约关系，致使线程在运行时也具有间断性。　　相应地，线程在运行时，也具有下述三种基本状态:

① 执行状态，表示线程正获得处理机而运行;

② 就绪状态，指线程已具备执行条件，一旦获得CPU便可执行的状态;

③ 阻塞状态，指线程在执行中因某事件而受阻，处于暂停执行时的状态。

# 用户线程与内核线程

- 用户级线程（ULT）：

（1）由应用程序通过线程库完成所有线程的管理。线程库包括线程的创建、撤消、消息和数据传递、调度执行以及上下文保护和恢复。

（2）内核不知道线程的存在。

（3）线程切换不需要核心态特权。

（4）调度是应用程序特定的。

- 用户级线程的内核活动：

（1）内核不知道线程的活动，但仍然管理线程的进程的活动。

（2）当线程调用系统调用时，整个进程阻塞。

（3）线程状态是与进程状态独立。

# 用户线程与内核线程

- 用户级线程的优点和缺点：
  （1）线程切换不调用内核。
  （2）调度是应用程序特定的：可以选择最好的算法。
  （3）ULT可运行在任何操作系统上（只需要线程库）。
  （4）大多数系统调用是阻塞的，因此内核阻塞进程，故进程中所有线程将被阻塞。
  （5）内核只将处理器分配给进程，同一进程中的两个线程不能同时运行于两个处理器上。

# 用户线程与内核线程

- 内核级线程（KLT）：
  （1）所有线程管理由内核完成。
  （2）没有线程库，但对内核线程工具提供API。
  （3）内核维护进程和线程的上下文。
  （4）线程之间的切换需要内核支持。
  （5）以线程为基础进行调度。

- 内核级线程的优点和缺点：
  （1）对多处理器，内核可以同时调度同一进程的多个线程。
  （2）阻塞是在线程一级完成。
  （3）内核例程是多线程的。
  （4）在同一进程内的线程切换调用内核，导致速度下降。

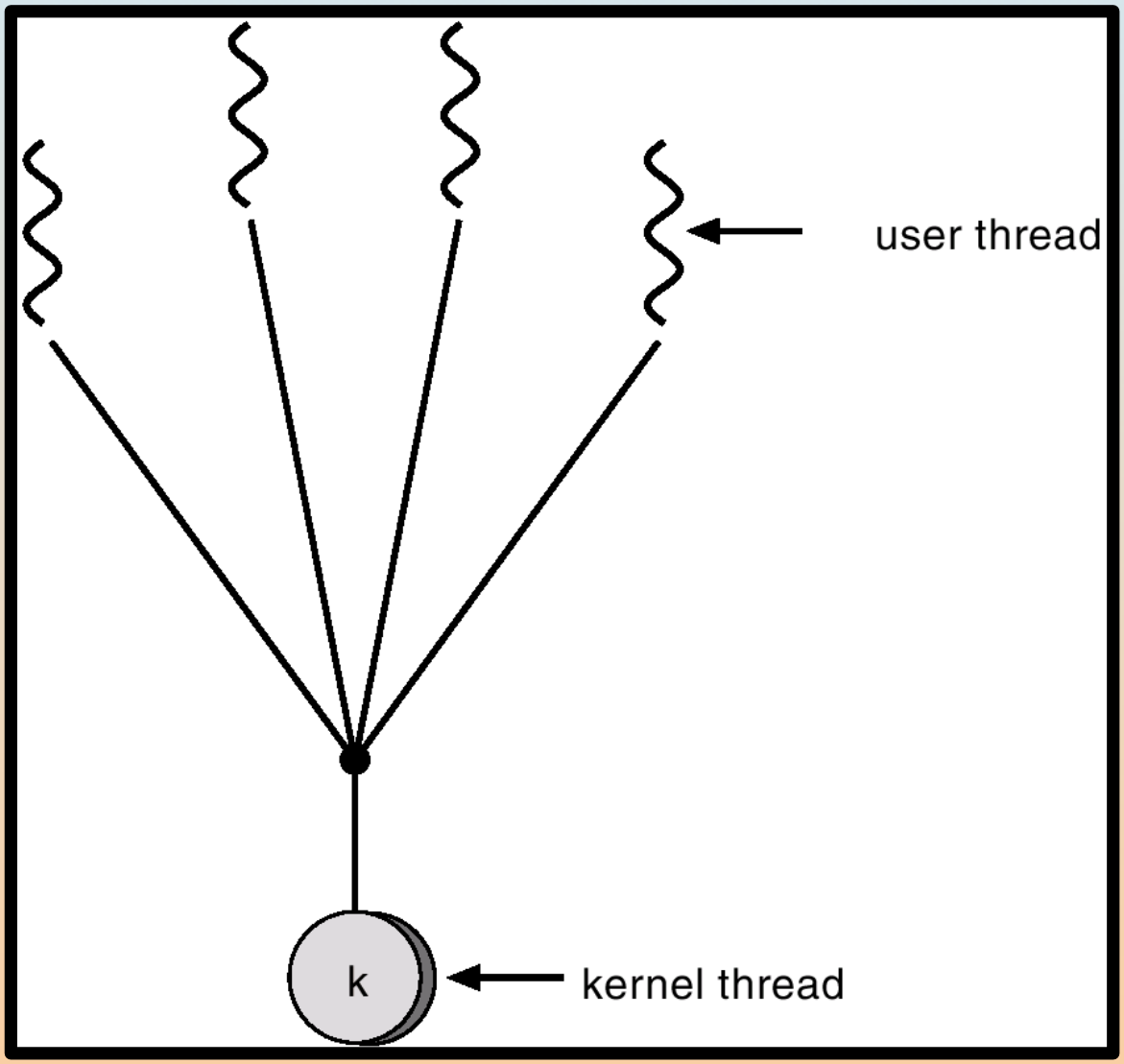
# 用户线程与内核线程

- 内核线程
  - 由操作系统直接支持：内核负责在内核空间执行线程创建、调度和管理；
  - 创建和管理比用户线程慢；
  - 线程之间的阻塞相互独立（既一个线程执行阻塞系统调用时，内核可以调度进程里的其它线程执行）；
  - 更好的支持多**CPU**体系结构；

# 多线程模型

- 用户线程和内核线程的对应关系，形成了三种多线程模型：
  - 多对一模型**(m:1)**
  - 一对一模型**(1:1)**
  - 多对多模型 **(m:n)**

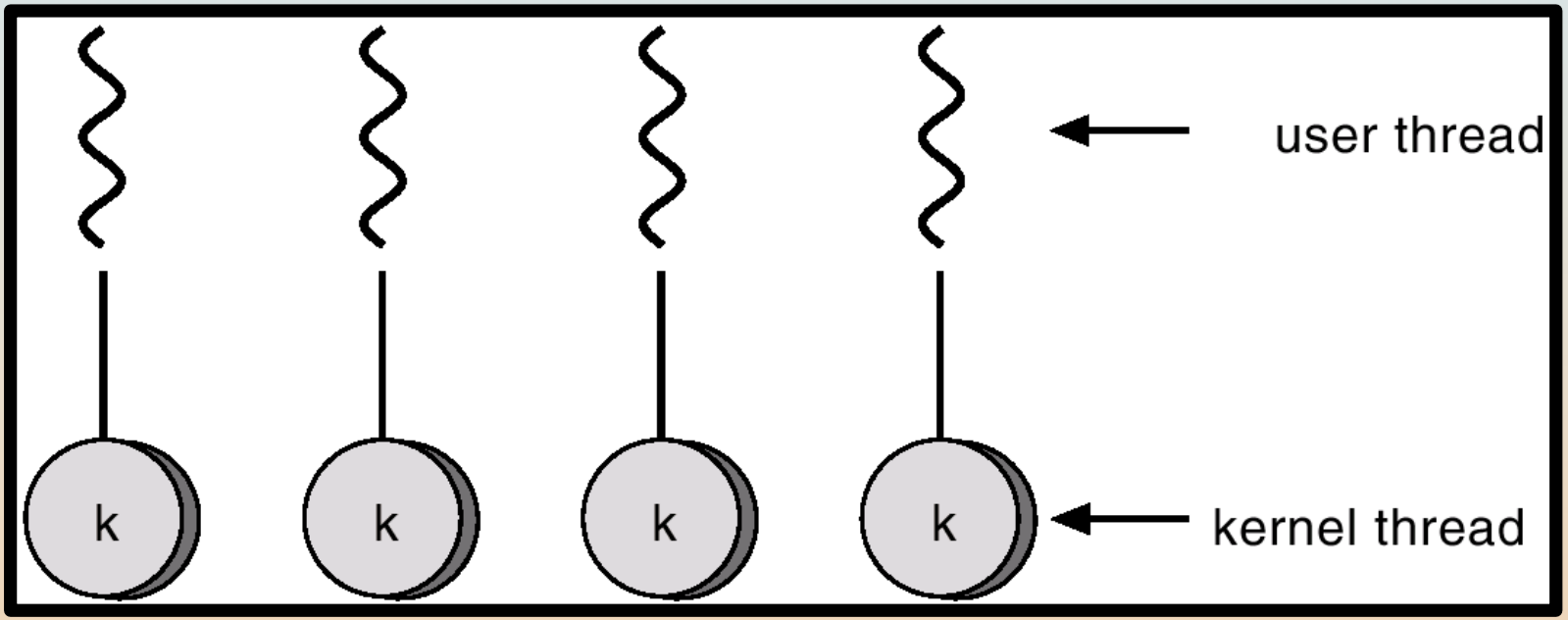
# 多对一模型



# 多对一模型

- 多个用户线程映射到一个内核线程
- 线程的管理在用户空间内完成，效率比较高
  - 通常用于不支持多内核线程的系统上
- 缺点
  - 一个线程阻塞将导致整个进程阻塞
  - 多个线程不能并行运行在多处理器上
- 例如
  - Solaris 2系统的*Green threads*
  - Unix的*Pthreads*

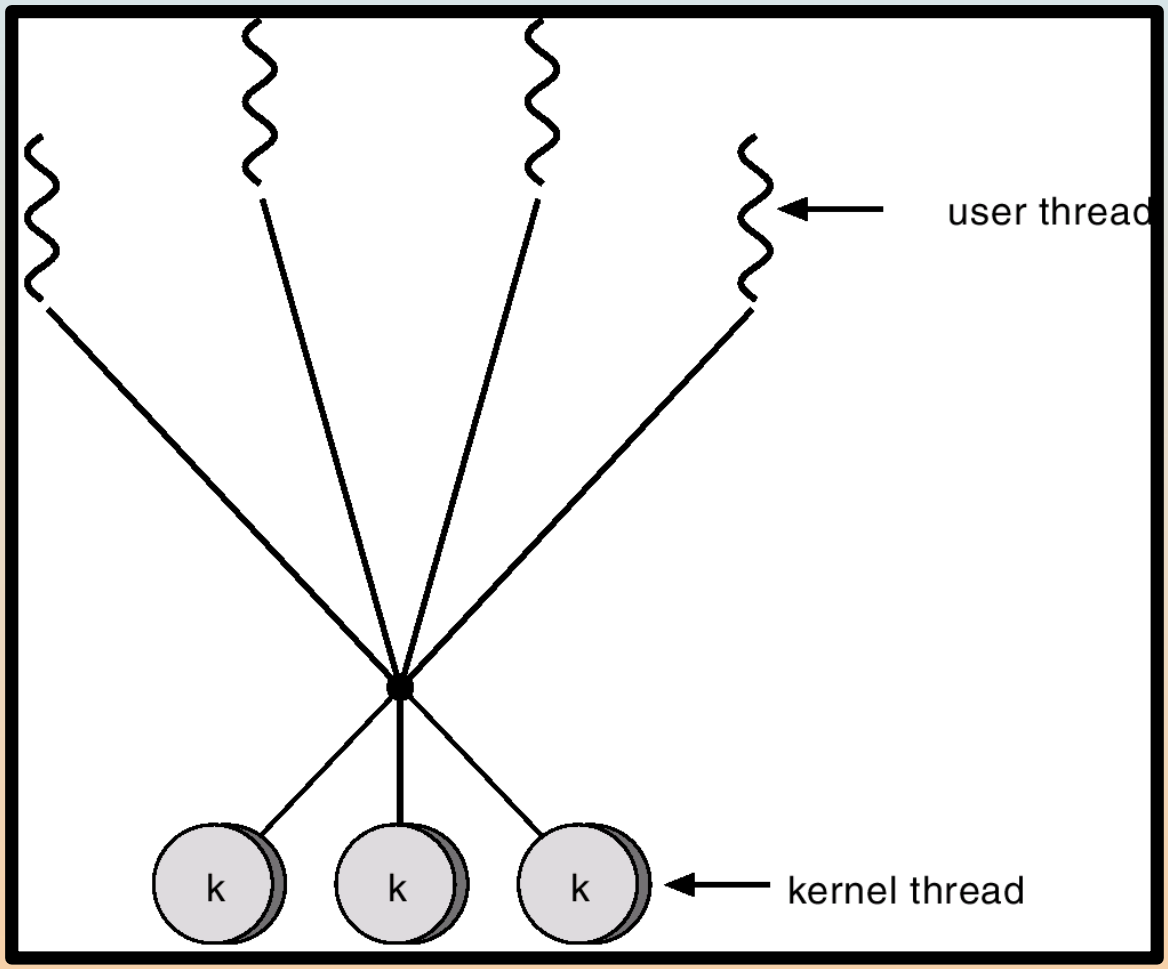
# 一对一模型



# 一对一模型

- 每个用户线程都映射到一个内核线程上
  - 提供了更好的并发性（某个线程的阻塞不一定导致进程的阻塞）
  - 支持多处理器
- 缺点
  - 创建用户线程造成的高开销
- 例如
  - Windows NT
  - OS/2

# 多对多模型



# 多对多模型

- 多路复用了许多用户线程到同样数量或者更小数量的内核线程上。 m:n (m>=n)
- 允许编程人员创建足够多的用户线程
- 避免线程阻塞引起进程阻塞、支持多处理器
- 例如
  - Solaris 2
  - IRIX
  - HP-UX
  - Tru64 UNIX

# 若干多线程的问题

- 线程取消
- 线程特定数据

# 线程取消

- 线程取消 是在线程完成之前来终止进程的任务

- 例如:
  - 多个线程并发执行以搜索数据库并且一个线程返回了结果
  - 用户按下网页浏览器上的停止按钮，装入网页的线程就被取消

# 两种线程取消的情况

- 目标线程: 要被取消的线程
- 异步取消: 一个线程立即终止目标线程
- 延迟取消: 目标线程不断的检查它是否应终止，允许一个线程检查它是否是在安全的点被取消。

# 线程特定数据

- 同属一个进程的线程共享进程数据
- 线程特定数据： 每个线程需要一定数据的拷贝
- 例如
  - 对于事务处理系统，需要通过独立线程以处理各个请求
  - 每个事务都有一个唯一标识符，为了让每个线程与其唯一标识符相关联，可以使用线程特定数据。
- 绝大多数线程库提供了对线程特定数据的支持，如 Win32, Pthreads, Java

# 小结

- 线程 是进程内的控制流
- 多线程 进程在同一地址空间内包括多个不同的控制流
- 用户级线程
  - 对程序员来说是可见的，对内核来说是未知的
  - 用户空间的线程库通常管理用户级线程
  - 优缺点
- 内核线程
  - 操作系统内核支持和管理内核级的线程
  - 利和弊
- 三种模型
  - 多对一 m:1
  - 一对一, 1:1
  - 多对多, m:n (m>=n)