操作系统 Operating System

第四章 进程与并发程序设计(1)
——进程与线程

沃天宇

woty@buaa.edu.cn

2020年3月30日

内容提要

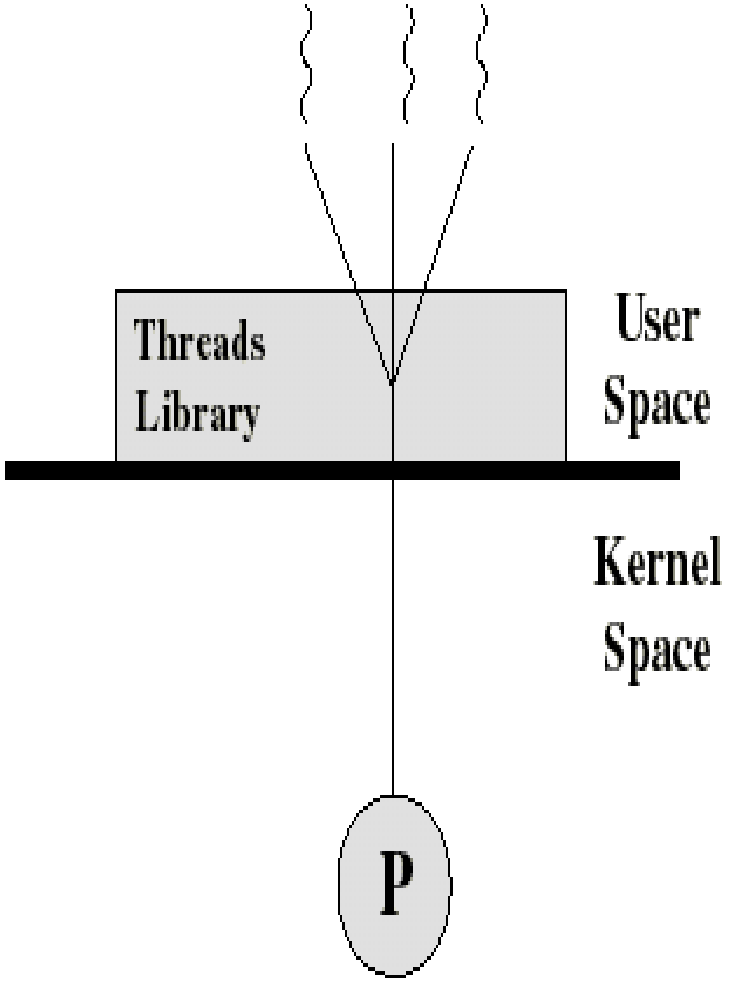
- 进程概念的引入
- 进程状态与控制
- 线程概念的引入
- 线程的实现方式
- 小结

线程的实现方式

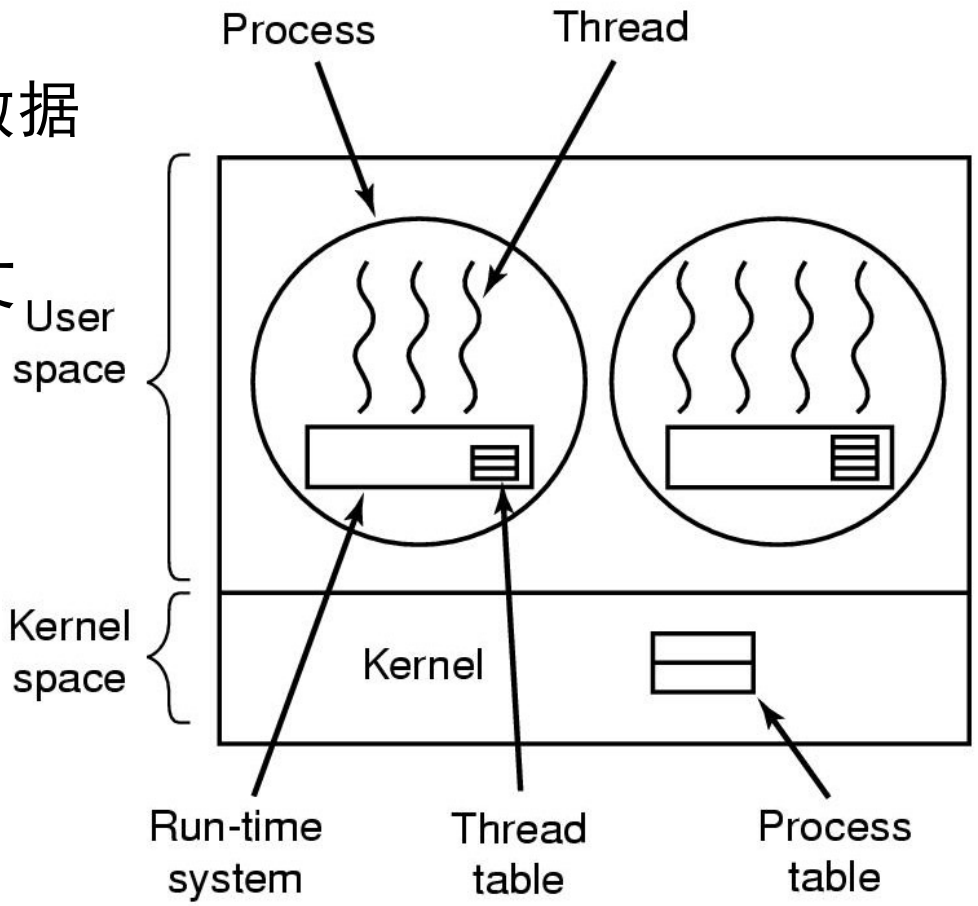
- 用户级线程：**User level threads(ULT)**
- 内核级线程：**Kernel level threads (KLT)**
- 混合实现方式

用户级线程

- 线程在用户空间,通过**library**模拟的**thread**,不需要或仅需要极少的**kernel**支持
- 上下文切换比较快,因为不用更改**page table**等,使用起来较为轻便快速.
- 提供操控视窗系统的较好的解决方案.



- 用户级的线程库的主要功能：
  - 创建和销毁线程
  - 线程之间传递消息和数据
  - 调度线程执行
  - 保存和恢复线程上下文
- 典型的例子
  - POSIX *Pthreads*
  - Mach *C-threads*
  - *Java Threads*



POSIX Pthreads

- 用于线程创建和同步的POSIX 标准API (IEEE 1003.1c).
- 可在用户级或者内核级实现.
- API规定了线程库的行为，但不限定实现方法
- 类Unix操作系统中很常见： Solaris, Linux, Mac OS X.

典型的Pthreads API

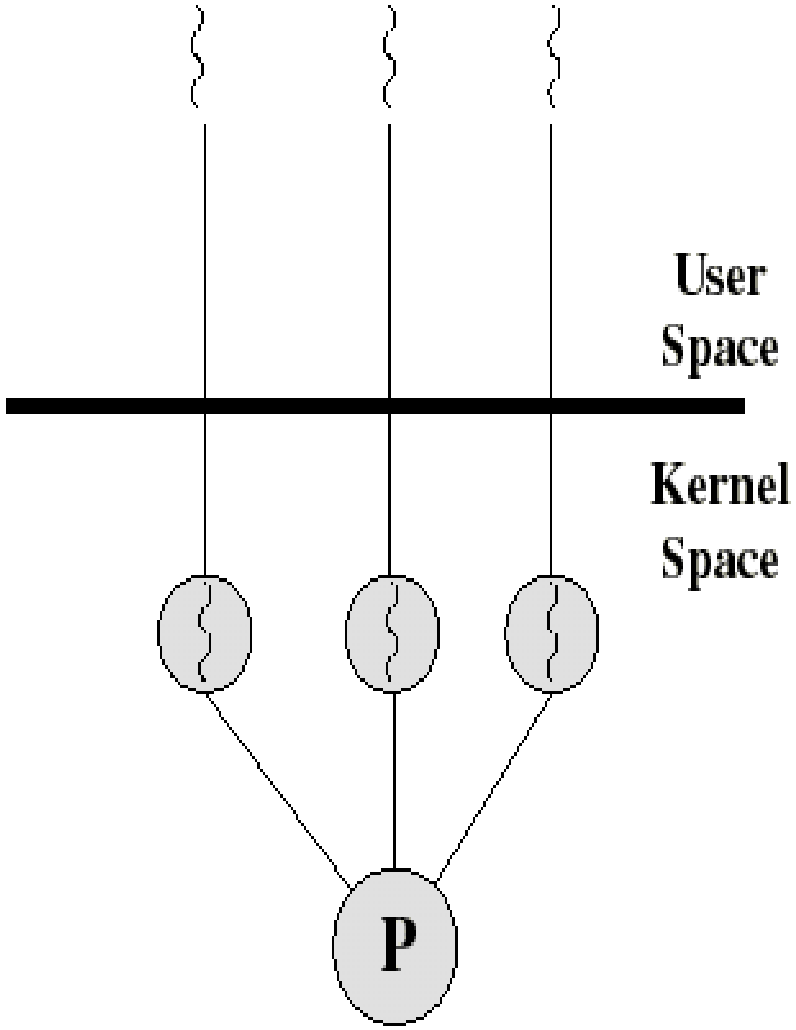
Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

用户级线程的优缺点

- 优点
  - 线程切换与内核无关
  - 线程的调度由应用决定，容易进行优化
  - 可运行在任何操作系统上，只需要线程库的支持
- 不足
  - 很多系统调用会引起阻塞，内核会因此而阻塞所有相关的线程。
  - 内核只能将处理器分配给进程，即使有多个处理器，也无法实现一个进程中的多个线程的并行执行。

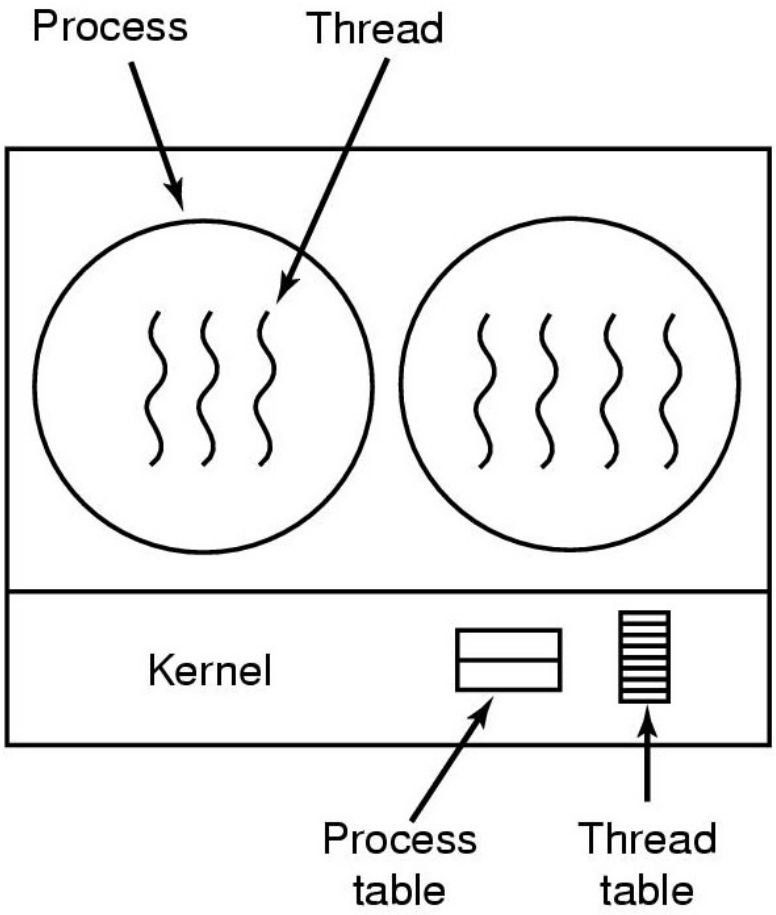
内核级线程

- 内核级线程就是kernel有好几个分身,一个分身可以处理一件事.
- 这用来处理非同步事件很有用,kernel可以对每个非同步事件产生一个分身来处理.
- 支持内核线程的操作系统内核称作*多线程内核*.



典型实现

- Windows 2000/XP
- OS/2
- Linux
- Solaris
- Tru64 UNIX
- Mac OS X



内核级线程的优缺点

- 优点
  - 内核可以在多个处理器上调度一个进程的多个线程实现同步并行执行
  - 阻塞发生在线程级别
  - 内核中的一些处理可以通过多线程实现
- 缺点
  - 一个进程中的线程切换需要内核参与，线程的切换涉及到两个模式的切换（进程-进程、线程-线程）
  - 降低效率

线程操作的延迟

μs

Operation	User-Level Threads	Kernel-Level Threads	Processes
Null Fork	34	948	11,300
Signal Wait	37	441	1,840

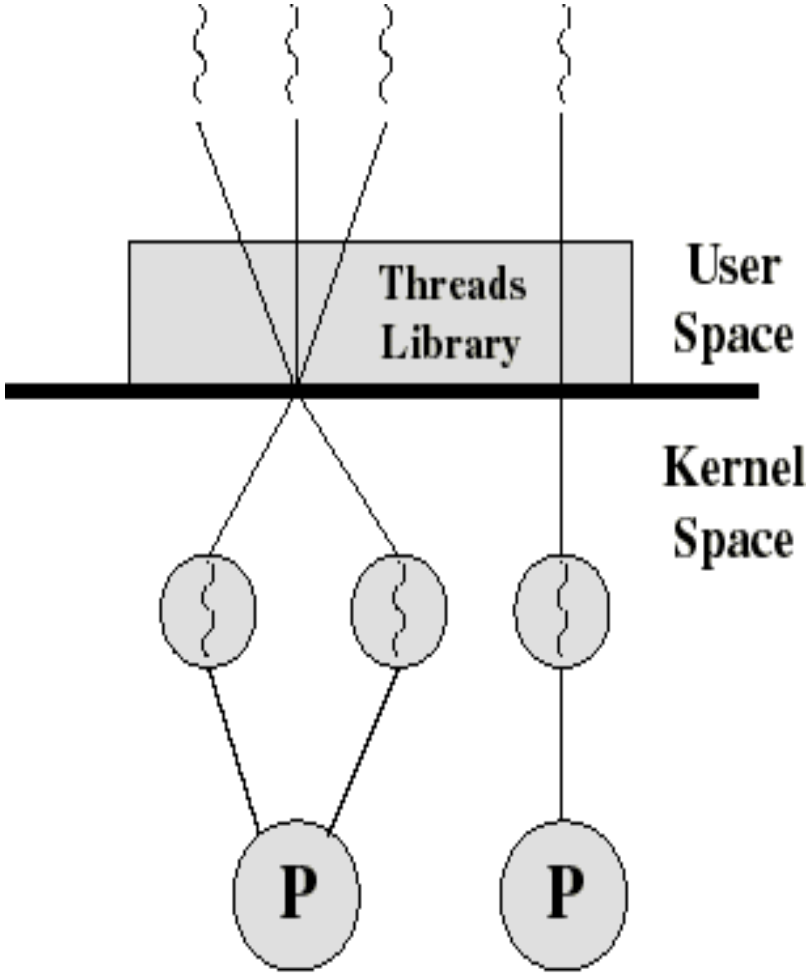
Source: Anderson, T. et al, “Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism”, ACM TOCS, February 1992.

混合的线程实现方式

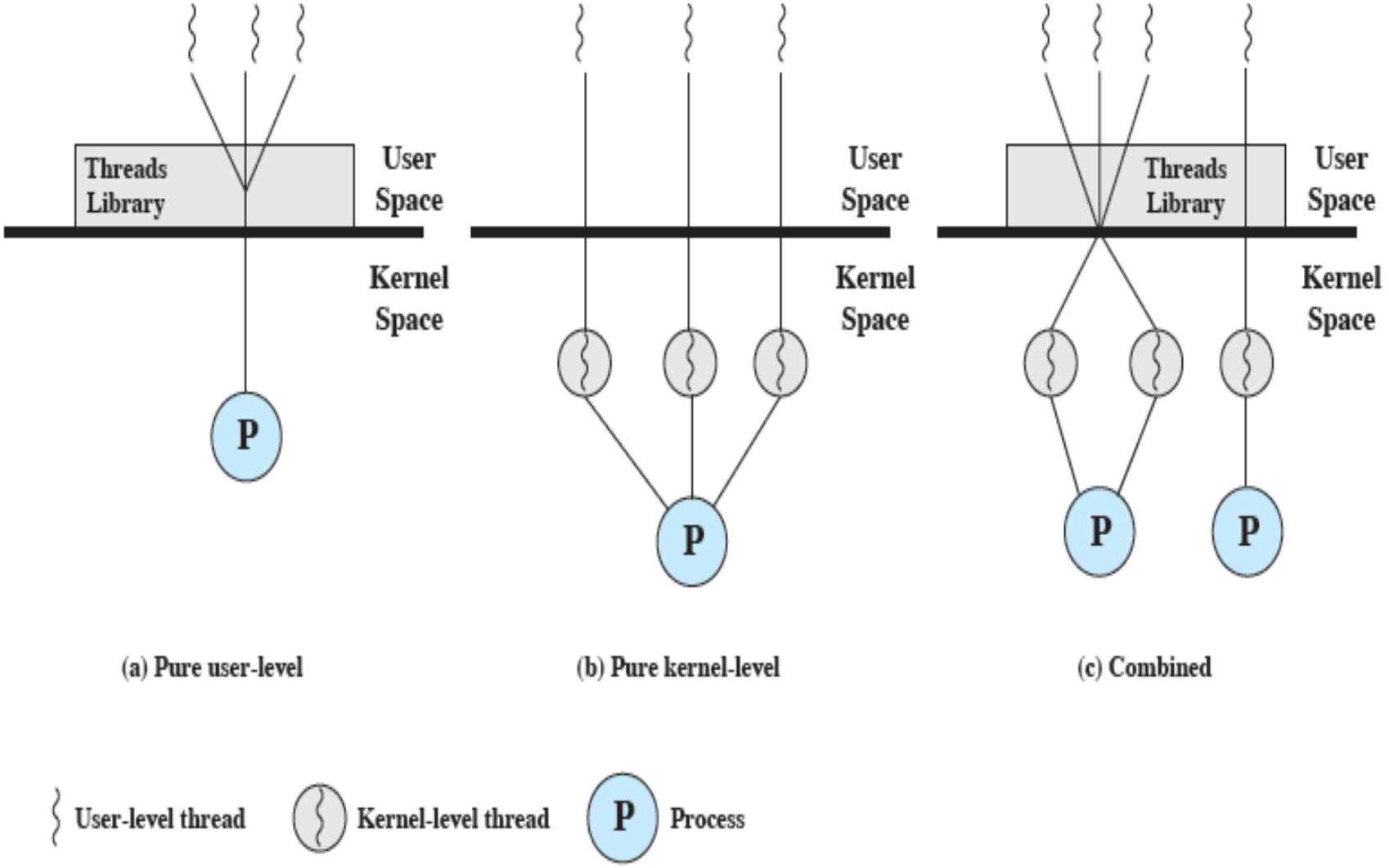
- 使用内核级线程，然后将用户级线程与某些或者全部内核线程多路复用起来，形成混合的线程实现方式。
- 采用这种方法，编程人员可以决定有多少个内核级线程和多少个用户级线程彼此多路复用。这一模型带来最大的灵活度。
- 内核只识别内核级线程，并对其进行调度。其中一些内核级线程会被多个用户级线程多路复用。如同在没有多线程能力操作系统中某个进程中的用户级线程一样，可以创建、撤销和调度这些用户级线程。在这种模型中，每个内核级线程有一个可以轮流使用的用户级线程集合。

混合的线程实现方式

- 线程在用户空间创建和管理
- 需要实现从用户空间的线程到内核空间线程（轻量级进程）的映射



三种线程实现方式的对比

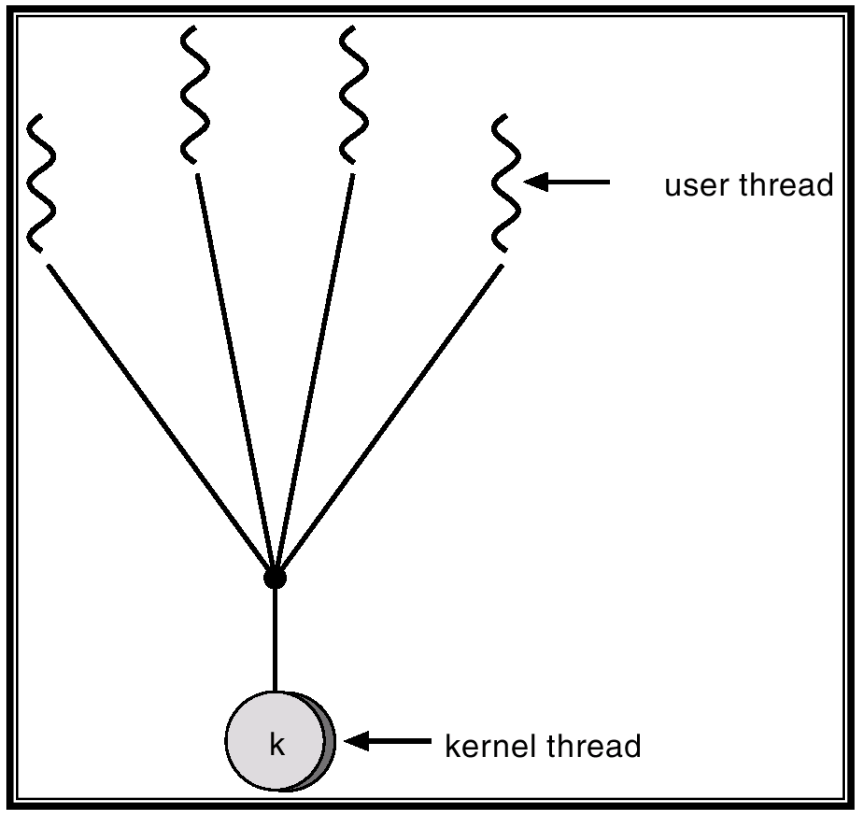


线程模型

- 有些系统同时支持用户线程和内核线程由此产生了不同的多线程模型，即实现用户级线程和内核级线程的连接方式。
  - Many-to-One
  - One-to-One
  - Many-to-Many

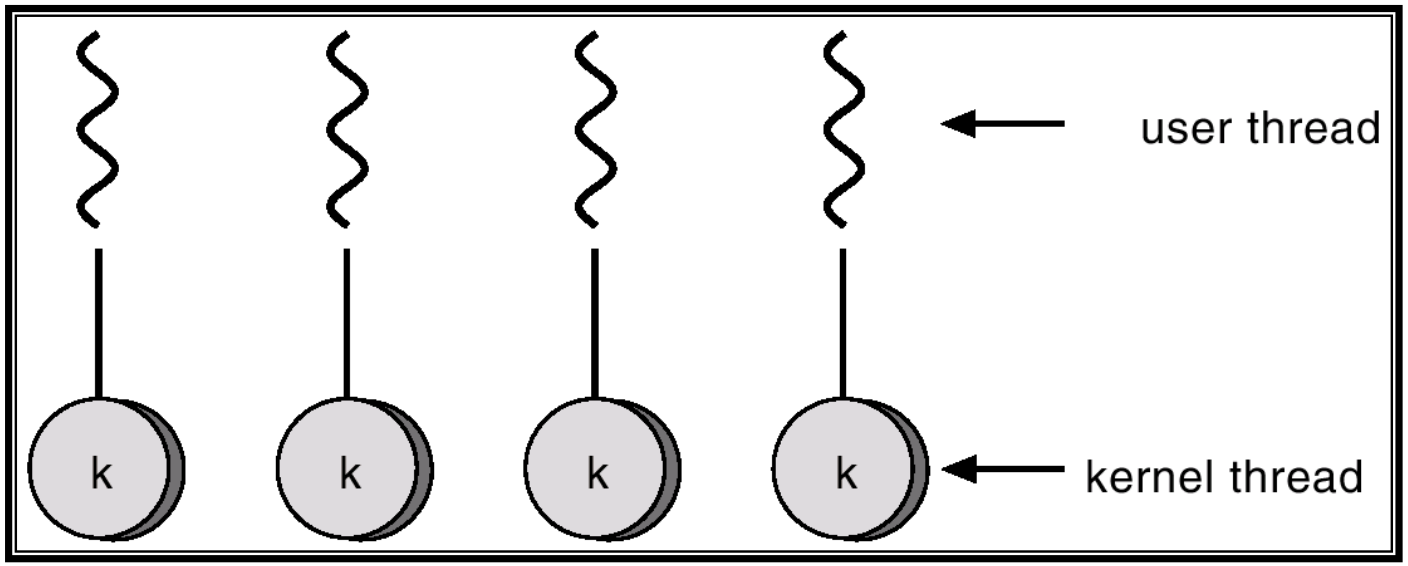
Many-to-One Model

- 将多个用户级线程映射到一个内核级线程，线程管理在用户空间完成。此模式中，用户级线程对操作系统不可见（即透明）。
  - 优点：线程管理是在用户空间进行的，因而效率比较高。
  - 缺点：当一个线程在使用内核服务时被阻塞，那么整个进程都会被阻塞；多个线程不能并行地运行在多处理机上。



One-to-one Model

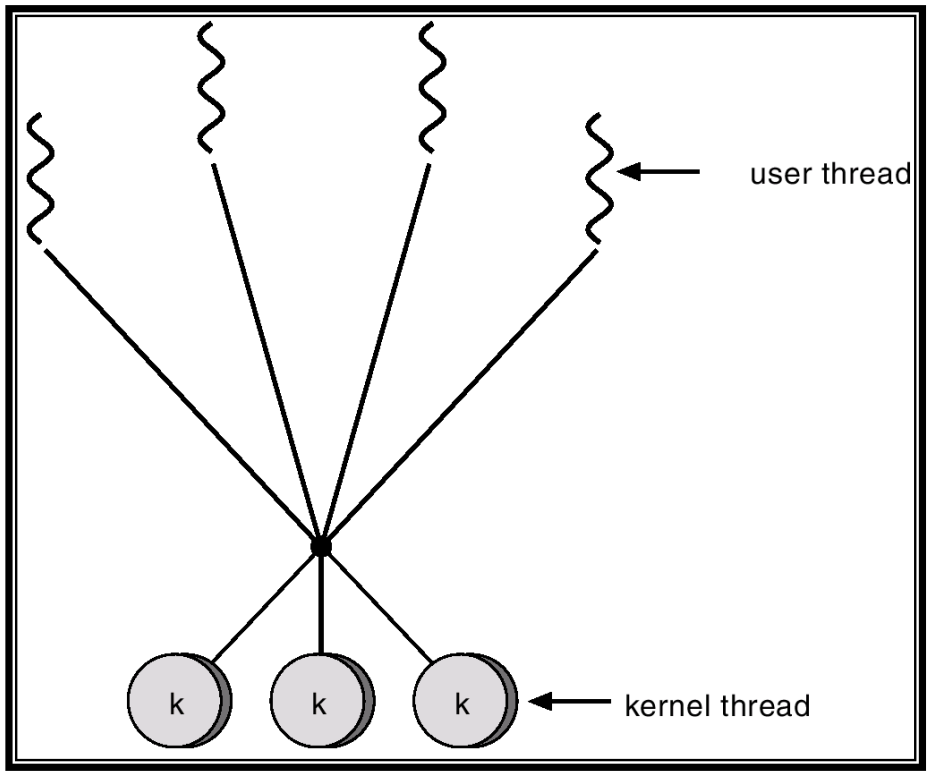
- 将每个用户级线程映射到一个内核级线程。
  - 优点：当一个线程被阻塞后，允许另一个线程继续执行，所以并发能力较强。
  - 缺点：每创建一个用户级线程都需要创建一个内核级线程与其对应，这样创建线程的开销比较大，会影响到应用程序的性能。



Many-to-Many Model

将 n 个用户级线程映射到 m 个内核级线程上，要求 m <= n。

- 特点：在多对一模型和一对一模型中取了个折中，克服了多对一模型的并发度不高的缺点，又克服了一对一模型的一个用户进程占用太多内核级线程，开销太大的缺点。又拥有多对一模型和一对一模型各自的优点，可谓集两者之所长。

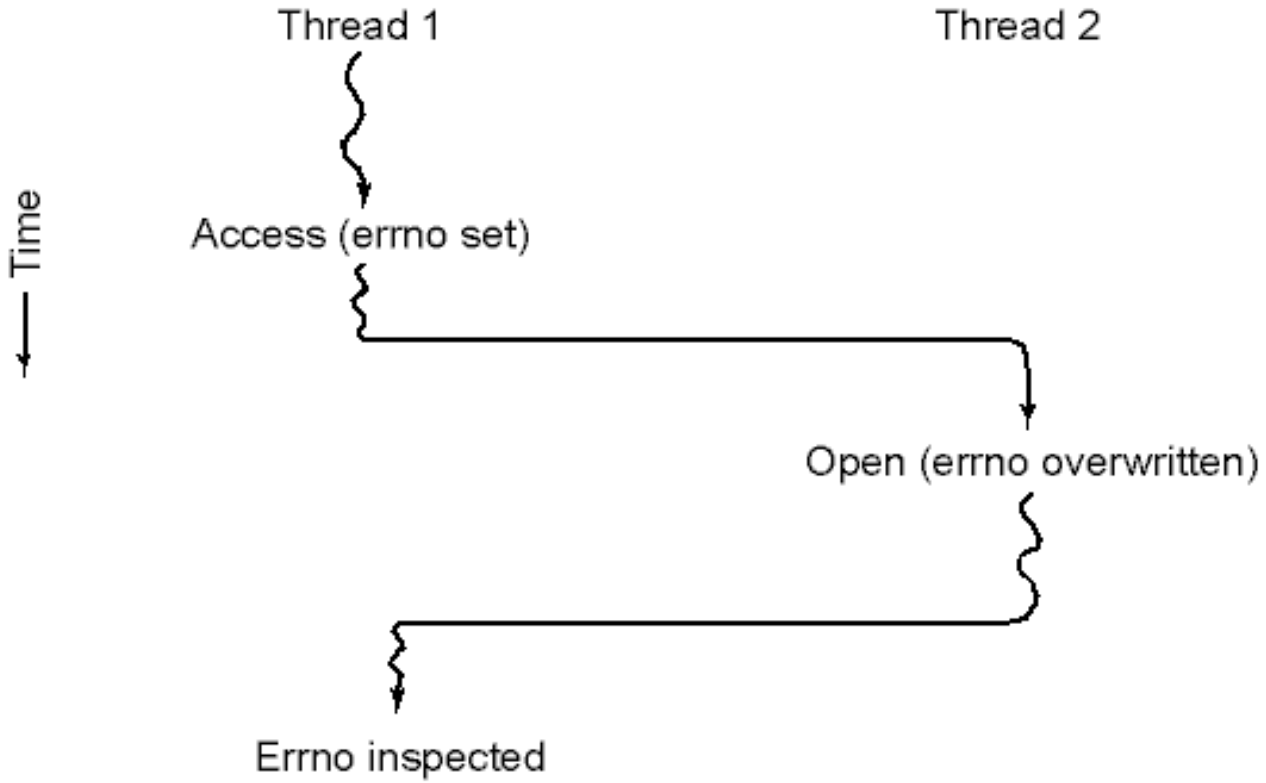


用户线程和内核线程的映射模型

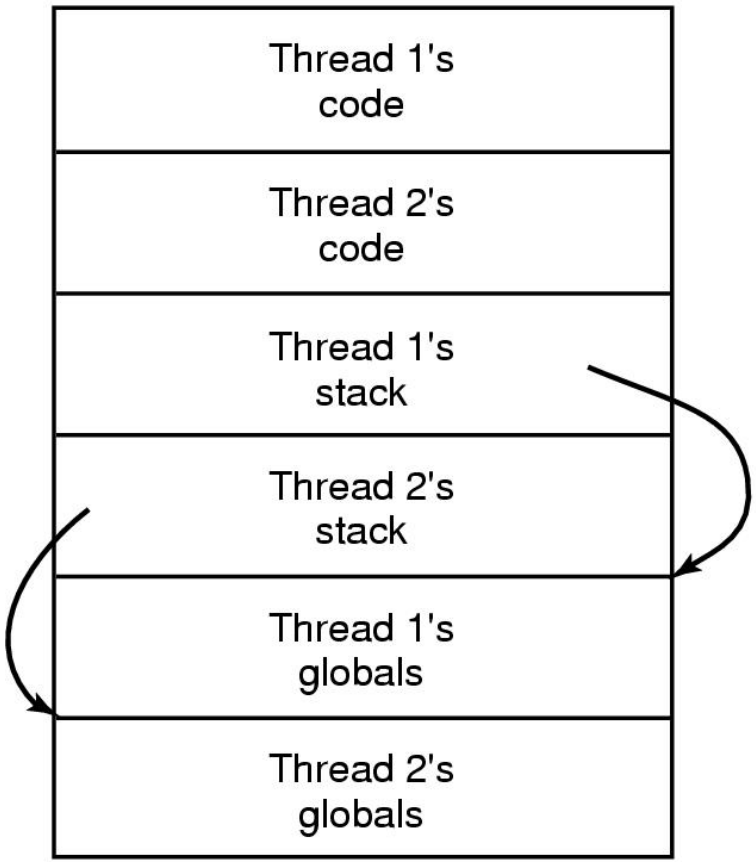
Threads:Processes	Description	Example Systems
1:1	Each thread of execution is a unique process with its own address space and resources.	Traditional UNIX implementations
M:1	A process defines an address space and dynamic resource ownership. Multiple threads may be created and executed within that process.	Windows NT, Solaris, Linux, OS/2, OS/390, MACH
1:M	A thread may migrate from one process environment to another. This allows a thread to be easily moved among distinct systems.	Ra (Clouds), Emerald
M:N	Combines attributes of M:1 and 1:M cases.	TRIX

线程编程问题—线程安全

- 全局变量



线程局部变量




```
int errno;

int main() {
        pthread_t pa, pb;
        pthread_create(&pa, NULL, access);
        pthread_create(&pb, NULL, open);
        pthread_join(pa,NULL);
        pthread_join(pb, NULL);
}

int access() {
        errno = 1;
}

int open() {
        errno = 0;
}
```

线程安全

- 多个线程访问同一个对象时，如果不用考虑这些线程在运行时环境下的调度和交替执行，也不需要进行额外的同步，或者在**调用方**进行任何其他操作，调用这个对象的行为都可以获得正确的结果，那么这个对象就是线程安全的。

- 可重入 vs 线程安全
  - 可重入不一定线程安全
  - 线程安全不一定可重入

```
// 可重入，
// 线程不安全
int tmp;
int add10(int a) {
    tmp = a;
    return a + 10;
}
```

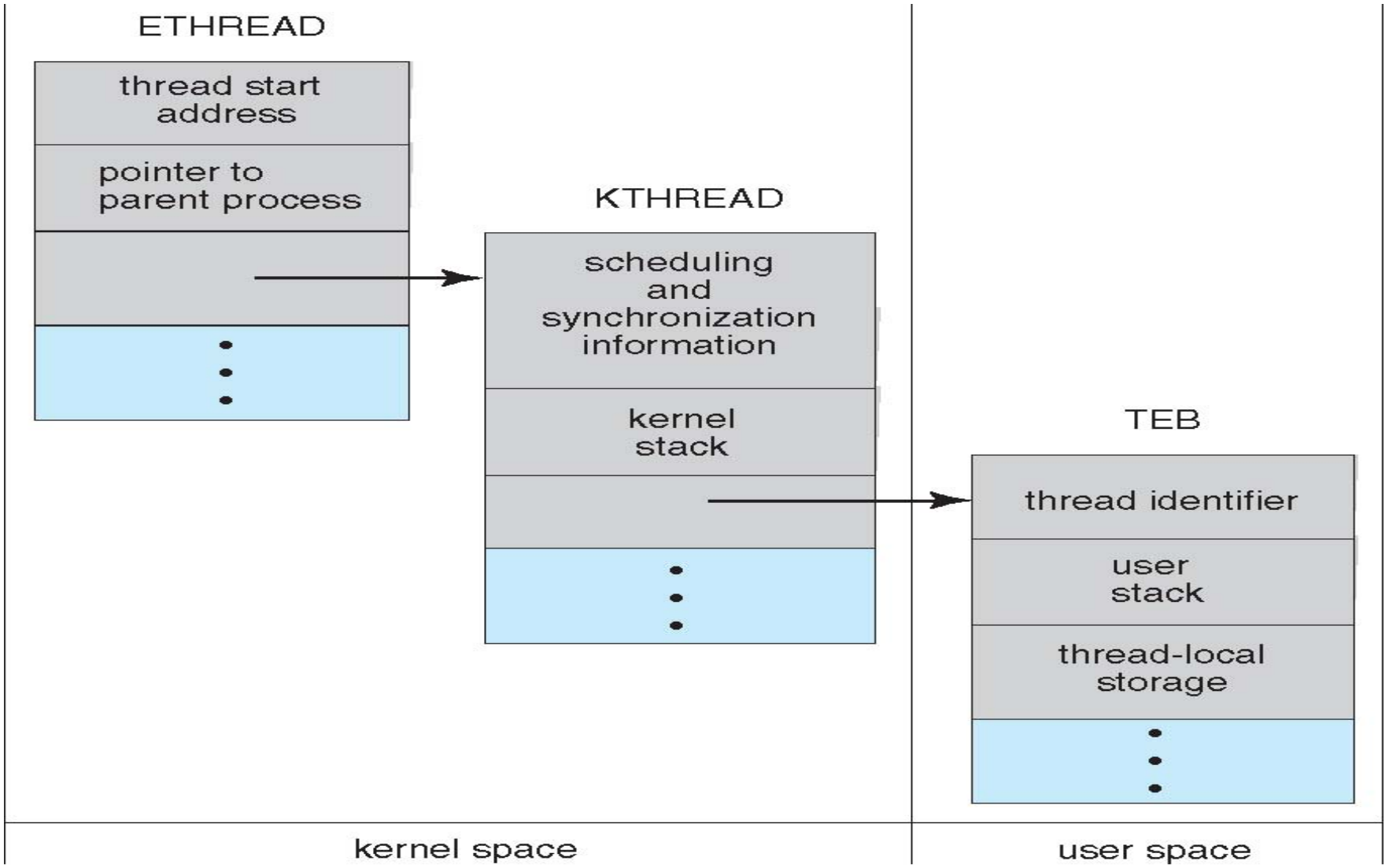
```
// 不可重入，
// 线程安全
thread_local int tmp;
int add10(int a) {
    tmp = a;
    return tmp + 10;
}
```

Windows XP 线程

- 实现one-to-one模型
- 每个线程包括的内容
  - 线程id
  - 若干寄存器
  - 用户和内核线程的堆栈
  - 私有的数据区

  } 线程的上下文
- 线程的主要数据结构包括
  - ETHREAD (executive thread block)
  - KTHREAD (kernel thread block)
  - TEB (thread environment block)

Windows XP Threads

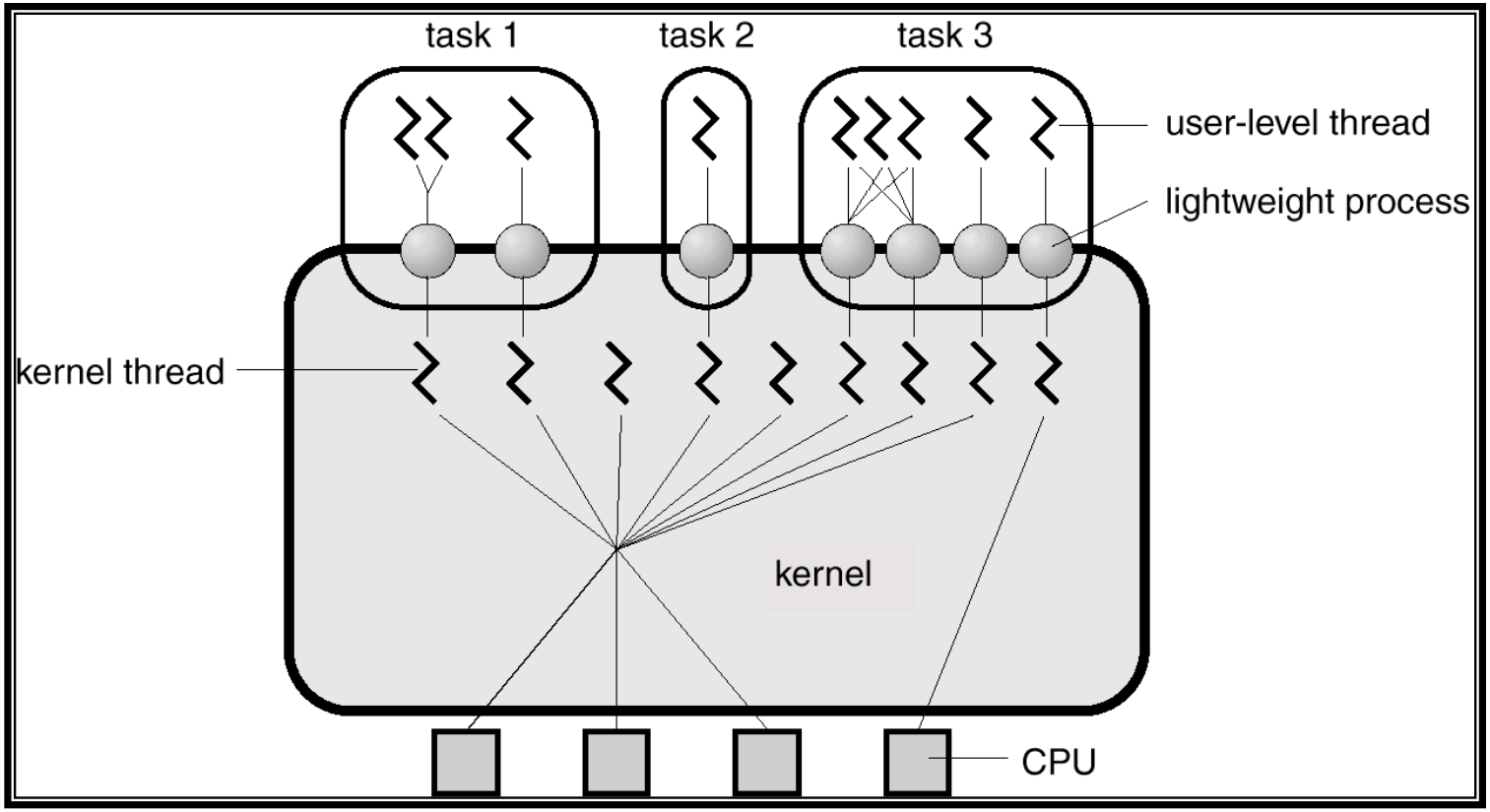


Solaris 2 Threads

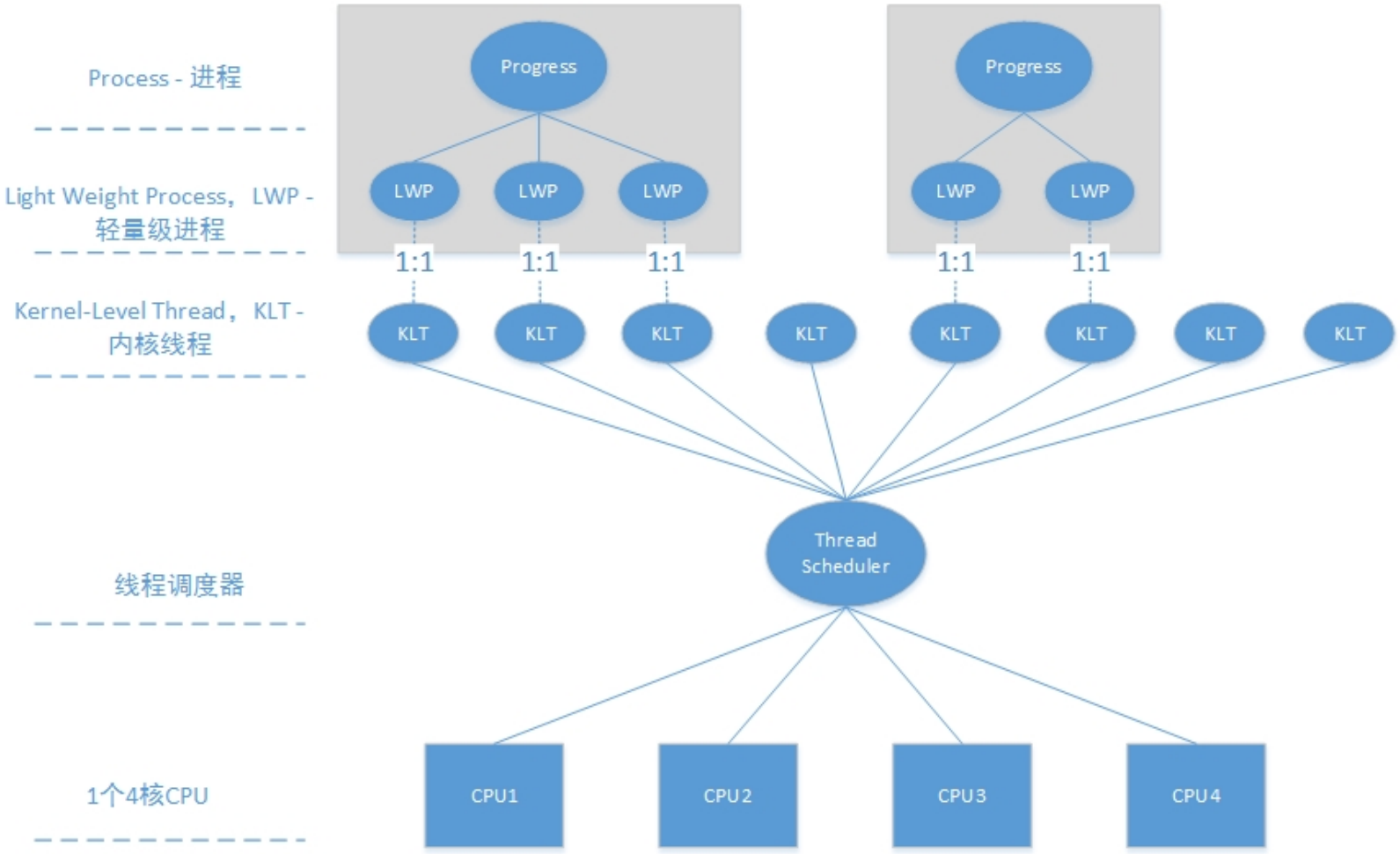
Solaris2 引入了四个线程相关的概念：

- 进程（process）：这是直接引用原来的UNIX的进程概念，包括用户地址空间、堆栈和进程控制块；
- 用户级线程（User-level threads）用户级线程是通过在进程地址空间内的线程库实现的，是内核不可见的。
- 轻量进程（Lightweight processes）一个轻量进程可以看成是用户级线程到内核线程之间的映射。每个轻量进程可以把一个或多个用户级线程映射为一个内核线程。轻量进程被内核单独调度并且可在多处理器间并行运行。
- 内核线程（Kernel thread）内核线程是基本被调度实体，并且也是被指派（dispatch）到多处理器之中一个上运行的独立实体。

Solaris 2 Threads

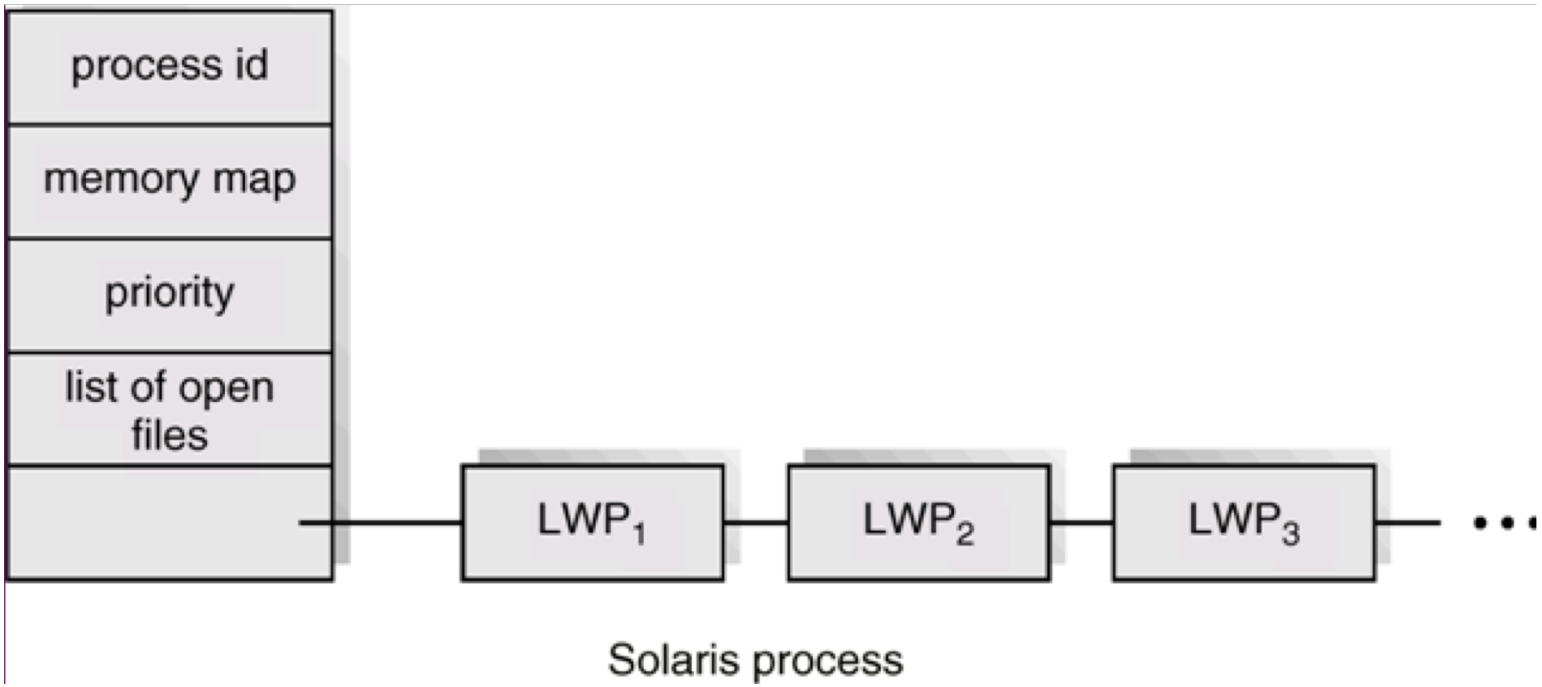


Solaris 2 Threads



进程、轻量级进程、内核线程
以及LWP和KLT的1:1关系

Solaris 2 Threads



Linux

- Linux并不确切区分进程与线程，而将线程定义为“执行上下文”，它实际只是同一个进程的另外一个执行上下文而已。对于调度，仍然可以使用进程的调度程序。Linux的内核进程，使用kernel_thread创建，一般被称作线程。

```
$ ps -fL
UID        PID  PPID   LWP  C NLWP STIME TTY          TIME CMD
wty      36076 36075 36076  0    1 13:29 pts/0    00:00:00 -bash
wty      54532 36076 54532  0    1 14:53 pts/0    00:00:00 ps -fL
```

Linux下的线程创建

- 有两个系统调用可用以建立新的进程：**fork**与**clone**。
- **fork**一般用以创建普通进程，而**clone** 可用以创建线程，**kernel_thread**便是通过**sys_clone**来创建新的内核线程。
- **fork**与**clone**都调用**do_fork**函数执行创建进程的操作。
- **fork**并不指定克隆标志，而**clone**可由用户指定克隆标志。克隆标志有**CLONE_VM**、**CLONE_FS**、**CLONE_FILES**、**CLONE_SIGHAND** 与**CLONE_PID**等，这些克隆标志分别对应相应的进程共享机制。而**fork**创建普通进程则使用**SIGCHLD**标志。

Linux下的线程创建

- CLONE_VM。父子进程共享同一个mm_struct结构，这个克隆标志用以创建一个线程。由于两个进程都使用同一个mm_struct结构，于是这两个进程的指令、数据都共享，也就是将线程视为同一个进程的不同执行上下文。
- CLONE_FS：父子进程共享同一个文件系统。
- CLONE_FILES：父子进程共享打开的文件。
- CLONE_SIGHAND。父子进程共享信号处理句柄。
- CLONE_PID。父子进程共享pid

思考题

- 什么情况下不适合用多线程?

小结

- 并发与并行的区别
- 引入进程的目的
- 进程与程序的区别
- 进程的状态与控制
- 引入线程的目的
- 线程与进程的区别
- 线程的实现方法

操作系统 Operating System

第四章 进程与并发程序设计(2)
——调度

沃天宇

woty@buaa.edu.cn

2020年3月30日

内容提要

- 基本概念
- 设计调度算法要考虑的问题
- 批处理系统的调度算法
- 交互式系统的调度算法
- 实时系统的调度算法
- 多处理机调度

CPU调度

什么是CPU调度?

CPU 调度的任务是控制、 协调 多个进程对 CPU 的竞争。 也就是按照一定的策略（调度算法），从就绪队列中 选择一个进程，并把 CPU 的控制权交给被选中的进程。

CPU调度的场景

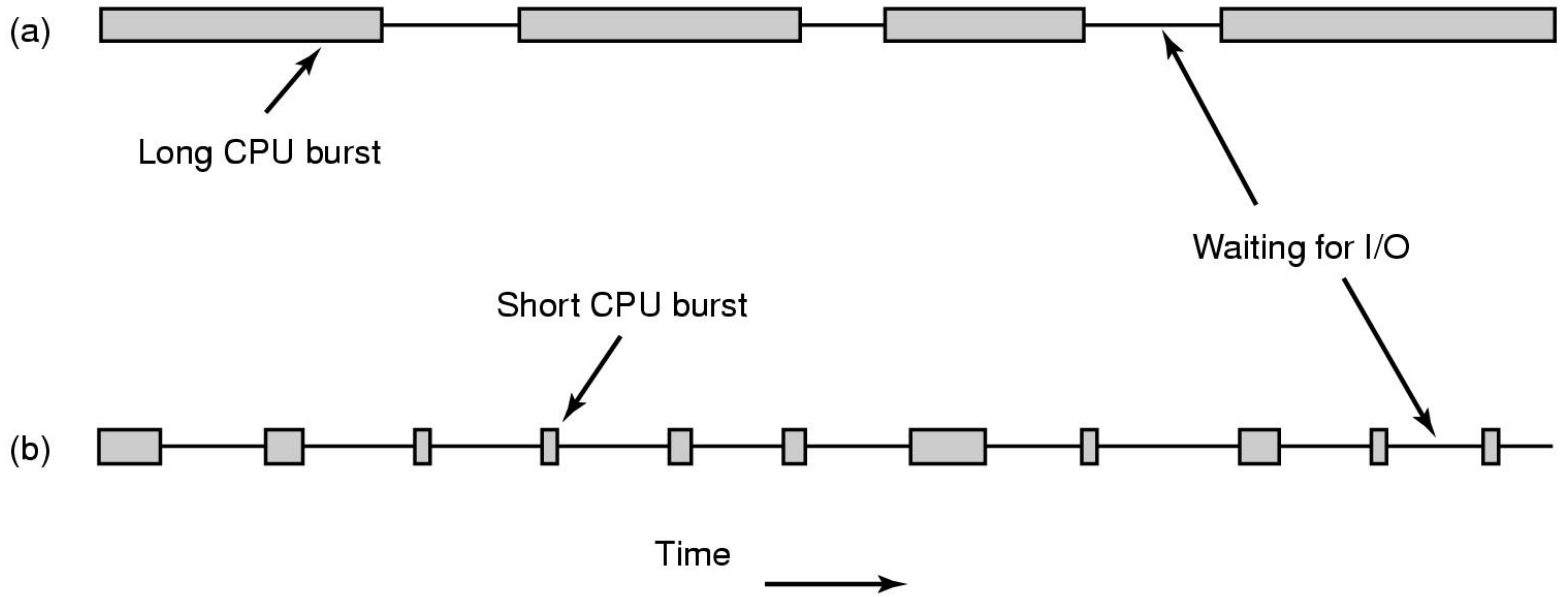
- N 个进程就绪，等待上 CPU 运行
- M个CPU，M≥1
- OS需要决策，给哪个进程分配哪个 CPU 。

要解决的问题

- WHAT:
  - 按什么原则分配CPU—进程调度算法
- WHEN:
  - 何时分配CPU—进程调度的时机
- HOW:
  - 如何分配CPU—CPU切换过程（进程的上下文切换）

问题

- 处理机管理的工作是对CPU资源进行合理的分配使用，以提高处理机利用率，并使各用户公平地得到处理机资源。这里的主要问题是处理机调度算法和调度算法特征分析。



调度的类型

- 高级调度
- 中级调度
- 低级调度

高级调度

- 高级调度：又称为“宏观调度”、“作业调度”。从用户工作流程的角度，一次提交的若干个作业，对每个作业进行调度。时间上通常是分钟、小时或天。

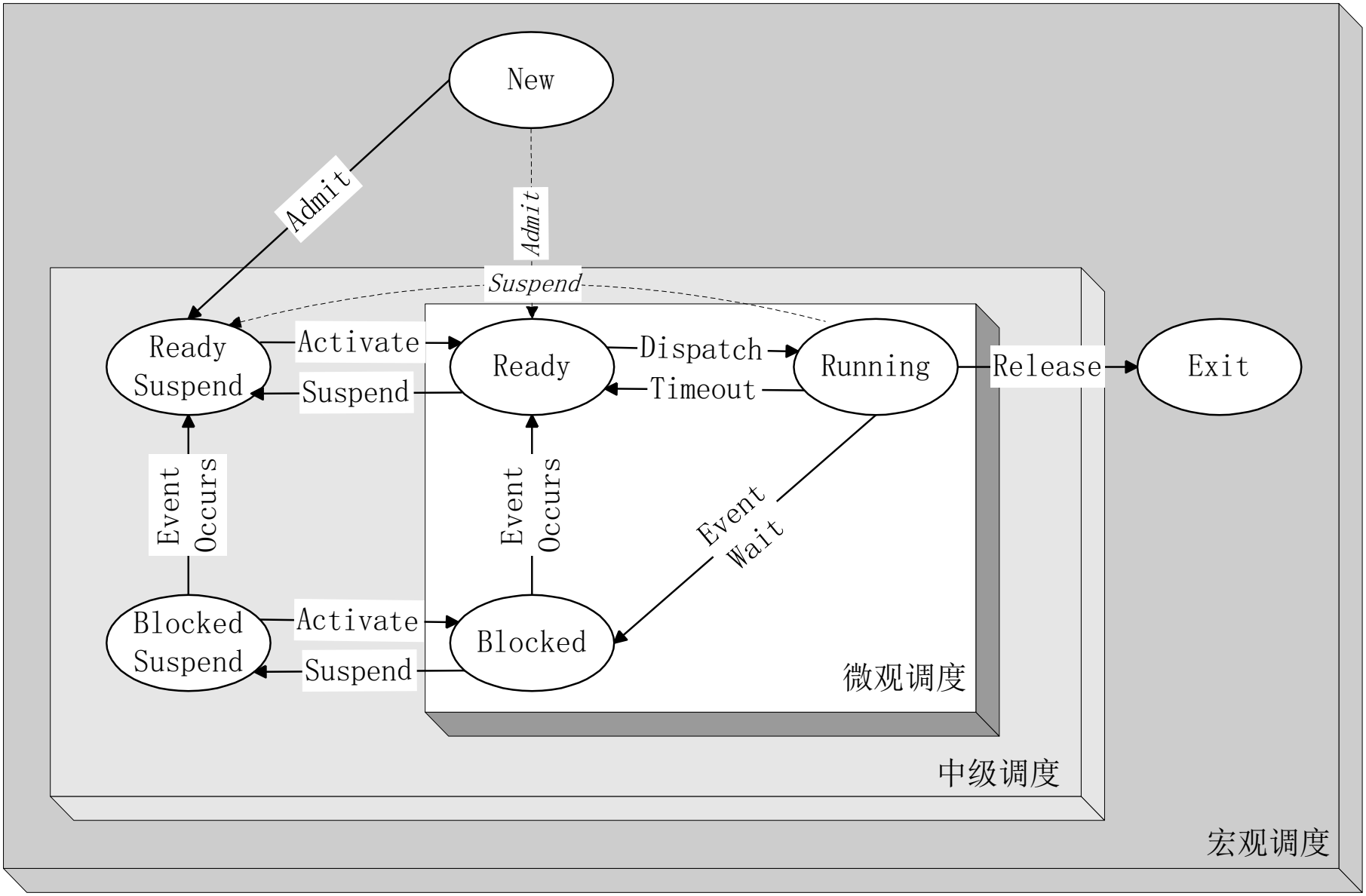
- 接纳多少个作业
- 接纳哪些作业

中级调度

- 内外存交换：又称为“中级调度”。
- 指令和数据必须在内存里才能被CPU直接访问。
- 从存储器资源的角度，将进程的部分或全部换出到外存上，将当前所需部分换入到内存。

低级调度

- 低级调度：又称为“微观调度”、“进程或线程调度”。从CPU资源的角度，执行的单位，时间上通常是毫秒。因为执行频繁，要求在实现时达到高效率。
- 非抢占式
- 抢占式
  - 时间片原则
  - 优先权原则
  - 短作业（进程）优先


New
Admit
Admit
Suspend
Ready
Suspend
Activate
Suspend
Ready
Dispatch
Timeout
Running
Release
Exit
Event
Occurs
Event
Occurs
Event
Wait
Blocked
Suspend
Activate
Suspend
Blocked
微观调度
中级调度
宏观调度

何时进行调度

- 当一个新的进程被创建时，是执行新进程还是继续执行父进程？ 例：fork()
- 当一个进程运行完毕时；
- 当一个进程由于I/O、信号量或其他的某个原因被阻塞时；
- 当一个I/O中断发生时，表明某个I/O操作已经完成，而等待该I/O操作的进程转入就绪状态；
- 在分时系统中，当一个时钟中断发生时。

何时进行切换

- 只要OS取得对CPU的控制，进程切换就可能发生:
  - 用户系统调用：来自程序的显式请求(如：打开文件)，该进程多半会被阻塞
  - 陷阱：最末一条指令导致出错，会引起进程移至退出状态
  - 中断：外部因素影响当前指令的执行，控制被转移至中断处理程序

进程切换

- 进程（上下文）切换的步骤
  - 保存处理器的上下文，包括程序计数器和其它寄存器
  - 用新状态和其它相关信息更新正在运行进程的PCB
  - 把进程移至合适的队列（就绪、阻塞）
  - 选择另一个要执行的进程（调度）
  - 更新被选中进程的PCB
  - 从被选中进程PCB中重新装入CPU 上下文

调度的性能准则

- 从不同的角度来判断处理机调度算法的性能，如用户的角度、处理机的角度和算法实现的角度。实际的处理机调度算法选择是一个综合的判断结果。

面向用户的调度性能准则1

- **周转时间：** 作业从提交到完成（得到结果）所经历的时间。包括：在收容队列中等待，CPU上执行，就绪队列和阻塞队列中等待，结果输出等待——批处理系统（外存等待时间、就绪等待时间、CPU执行时间、I/O操作时间）
  - 平均周转时间
  - 带权平均周转时间（T/Ts)：总周转时间/总服务时间
- **响应时间：** 用户输入一个请求（如击键）到系统给出首次响应（如屏幕显示）的时间——分时系统

面向用户的调度性能准则2

- 截止时间：开始截止时间和完成截止时间——实时系统，与周转时间有些相似。
- 优先级：可以使关键任务达到更好的指标。
- 公平性：不因作业或进程本身的特性而使上述指标过分恶化，如长作业等待很长时间。

面向系统的调度性能准则

- 吞吐量：单位时间内所完成的作业数，跟作业本身特性和调度算法都有关系——批处理系统
  - 平均周转时间不是吞吐量的倒数，因为并发执行的作业在时间上可以重叠。如：在2小时内完成4个作业，则吞吐量是2个作业/小时，而平均周转时间可能是0.5小时、1小时、1.25小时、2小时、…
- 处理机利用率：——大中型主机
- 各种资源的均衡利用：如CPU繁忙的作业和I/O繁忙（指次数多，每次时间短）的作业搭配——大中型主机

调度算法本身的调度性能准则

- 易于实现
- 执行开销比

内容提要

- 基本概念
- 设计调度算法要考虑的问题
- 批处理系统的调度算法
- 交互式系统的调度算法
- 实时系统的调度算法
- 多处理机调度

设计调度算法要点

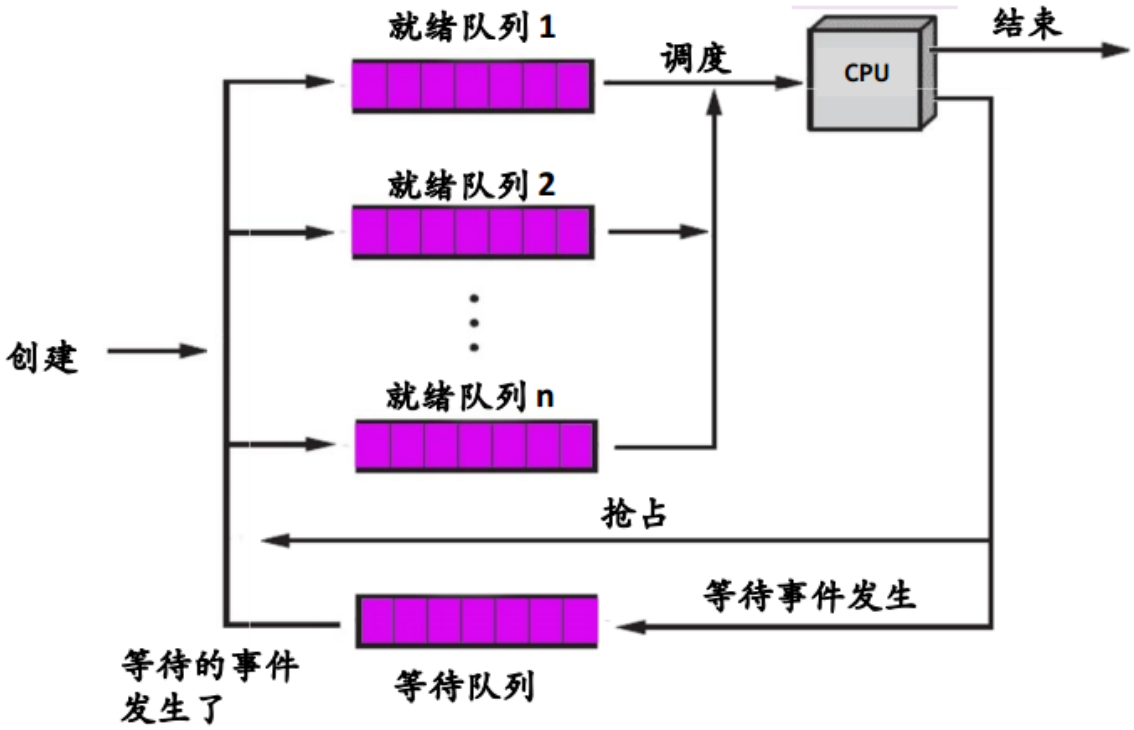
- 进程优先级（数）
- 进程优先级就绪队列的组织
- 抢占式调度与非抢占式调度
- 进程的分类
- 时间片

进程优先级（数）

- 优先级和优先数是不同的，优先级表现了进程的重要性和紧迫性，优先数实际上是一个数值，反映了某个优先级。
- 静态优先级
  - 进程创建时指定，运行过程中不再改变
- 动态优先级
  - 进程创建时指定了一个优先级，运行过程中可以动态变化。如：等待时间较长的进程可提升其优先级。

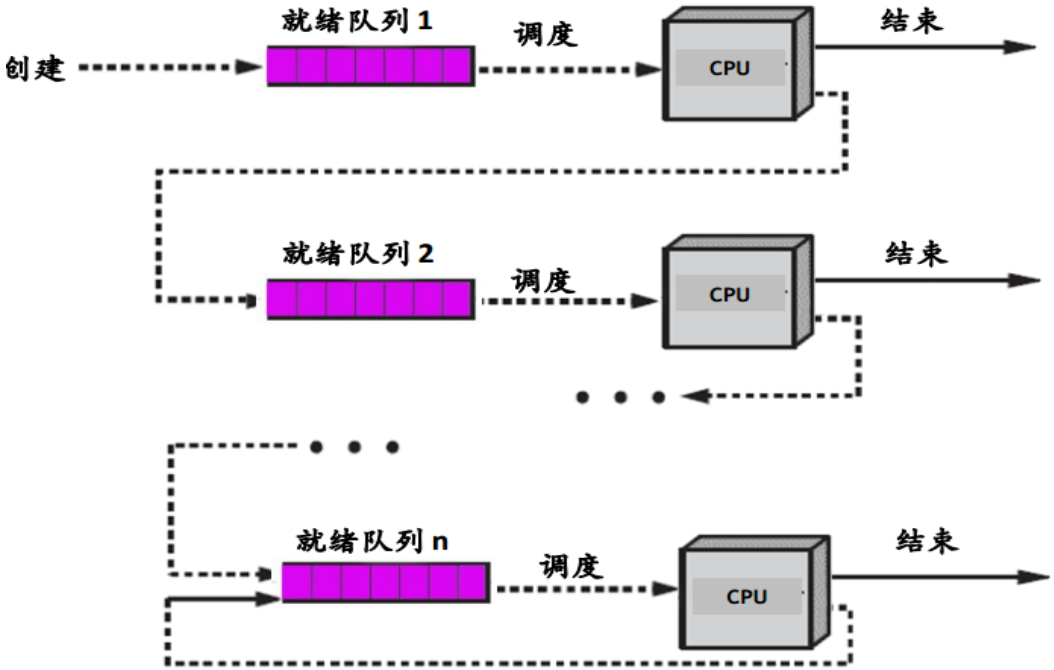
进程就绪队列组织

- 按优先级排队方式
  - 创建多个进程后按照不同的优先级进行排列，CPU调度优先级较高的进程进行执行。



进程就绪队列组织

- 另一种方式
  - 所有进程创建之后都进入到第一级就绪队列，随着进程的运行，可能会降低某些进程的优先级，如某些进程的时间片用完了，那么就会将其降级。

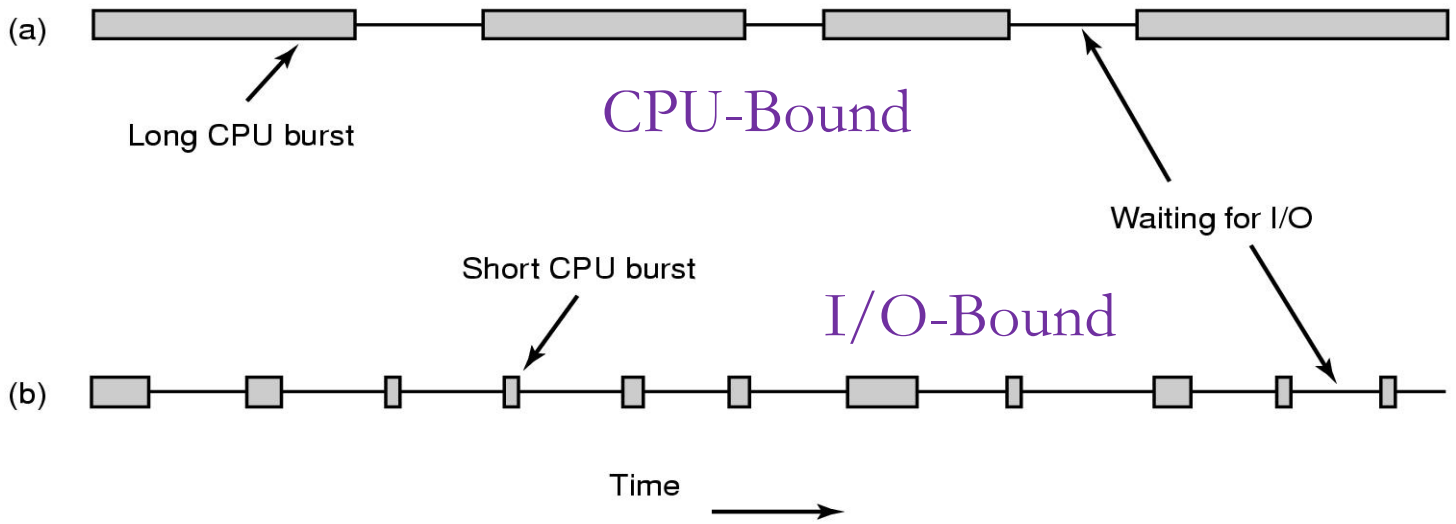


占用CPU的方式

- 不可抢占式方式
  - 一旦处理器分配给一个进程，它就一直占用处理器，直到该进程自己因调用原语操作或等待I/O等原因而进入阻塞状态，或时间片用完时才让出处理器，重新进行
- 抢占式方式
  - 就绪队列中一旦有优先级高于当前运行进程优先级的进程存在时，便立即进行进程调度，把处理器转给优先级高的进程

进程的分类（第一种）

- I/O Bound（I/O密集型）
  - 频繁的进行I/O，通常会花费很多时间等待I/O操作完成
- CPU bound（CPU密集型）
  - 计算量大，需要大量的CPU时间。



进程的分类（第二种）

- 批处理进程（Batch Process)
  - 无需与用户交互，通常在后台运行
  - 不需很快的响应
  - 典型的批处理程序：编译器、科学计算
- 交互式进程（Interactive Process)
  - 与用户交互频繁，因此要花很多时间等待用户输入
  - 响应时间要快，平均延迟要低于50~150ms
  - 典型的交互式进程：Word、触控型GUI
- 实时进程（Real-time Process)
  - 有实时要求，不能被低优先级进程阻塞
  - 响应时间要短且要稳定
  - 典型的实时进程：视频/音频、控制类

时间片（Time slice或quantum）

- 一个时间段，分配给调度上CPU的进程，确定了允许该进程运行的时间长度。那么如何选择时间片？有一下需要考虑的因素：
  - 进程切换的开销
  - 对响应时间的要求
  - 就绪进程个数
  - CPU能力
  - 进程的行为

内容提要

- 基本概念
- 设计调度算法要考虑的问题
- 批处理系统的调度算法
- 交互式系统的调度算法
- 实时系统的调度算法
- 多处理机调度

吞吐量、平均等待时间和平均周转时间

- 吞吐量 $=\frac{\text{作业数}}{\text{总执行时间}}$，即单位时间CPU完成的作业数量
- 周转时间(Turnover Time) = 完成时刻 – 提交时刻
- 带权周转时间=周转时间/服务时间（执行时间）
- 平均周转时间 $=\frac{\text{作业周转时间之和}}{\text{作业数}}$
- 平均带权周转时间 $=\frac{\text{作业带权周转时间之和}}{\text{作业数}}$

批处理系统中常用的调度算法

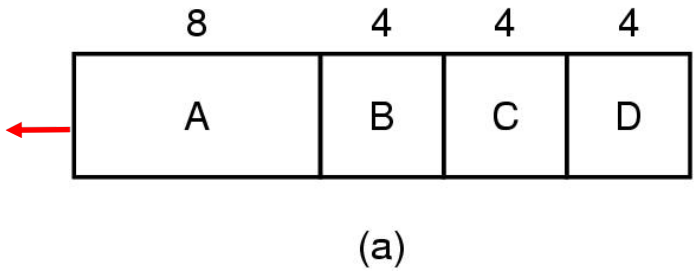
- 先来先服务（FCFS：First Come First Serve)
- 最短作业优先（SJF：Shortest Job First)
- 最短剩余时间优先（SRTF：Shortest Remaining Time First)
- 最高响应比优先（HRRF：Highest Response Ratio First)

先来先服务 (FCFS, First Come First Served)

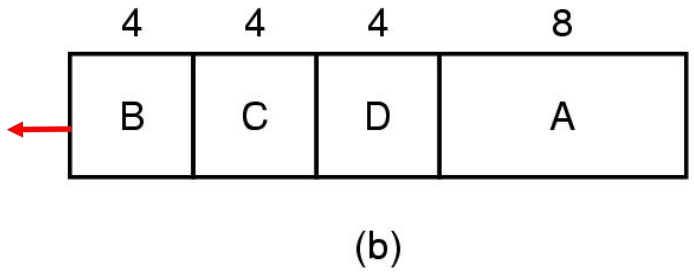
- 这是最简单的调度算法，按先后顺序调度。
  - 按照作业提交或进程变为就绪状态的先后次序，分派CPU；
  - 当前作业或进程占用CPU，直到执行完或阻塞，才出让CPU（非抢占方式）。
  - 在作业或进程唤醒后（如I/O完成），并不立即恢复执行，通常等到当前作业或进程出让CPU。
- FCFS的特点
  - 比较有利于长作业，而不利于短作业。
  - 有利于CPU繁忙的作业，不利于I/O繁忙的作业。

短作业优先(SJF, Shortest Job First)

- 又称为“短进程优先”SPN(Shortest Process Next)；这是对FCFS算法的改进，其目标是减少平均周转时间。
  - 对预计执行时间短的作业（进程）优先分派处理机。通常后来的短作业不抢先正在执行的作业。



(a)



(b)

SJF的特点

- 优点：
  - 比FCFS改善平均周转时间和平均带权周转时间，缩短作业的等待时间；
  - 提高系统的吞吐量；
- 缺点：
  - 对长作业非常不利，可能长时间得不到执行；
  - 未能依据作业的紧迫程度来划分执行的优先级；
  - 难以准确估计作业（进程）的执行时间，从而影响调度性能。

示例

有三道作业，它们的提交时间和运行时间见下表

作业号	提交时刻	运行时间/h
1	10:00	2
2	10:10	1
3	10:25	0.25

试给出在下面两种调度算法下，作业的执行顺序、平均周转时间和平均带权周转时间。

（1）先来先服务FCFS调度算法；

（2）短作业优先SJF调度算法。

示例

采用FCFS调度算法时，作业的执行顺序是作业1 ->作业2 ->作业3。由此可得到运行表

作业号	提交时刻	运行时间/h	开始时刻	完成时刻
1	10:00	2	10:00	12:00
2	10:10	1	12:00	13:00
3	10:25	0.25	13:00	13:15

那么，平均周转时间为

$T=(\sum Ti)/3=[(12-10)+(13-10:10)+(13:15-10:25)]/3$
$=[2+2.83+2.83]/3 =2.55h$

带权平均周转时间为

$W=[\sum(Ti/Tir)]/3=(2/2+2.83/1+2.83/0.25)/3=5.05h$

示例

在SJF调度算法下，作业的执行顺序是作业1 -> 作业3-> 作业2；由此得运行表

作业号	提交时刻	运行时间/h	开始时刻	完成时刻
1	10:00	2	10:00	12.00
2	10:10	1	12:15	13:15
3	10:25	0.25	12:00	12:15

那么，平均周转时间为

$T=(\sum Ti)/3=[(12-10)+(13:15-10:10)+(12:15-10:25)]/3$
$=[2+3.08+1.83]/3=2.3h$

带权平均周转时间为

$W=[\sum(Ti/Tir)]/3=(2/2+3.08/1+1.83/0.25)/3=3.8h$

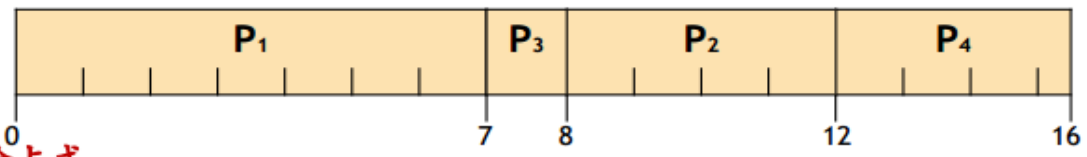
最短剩余时间优先SRTF

- 将短作业优先进行改进，改进为抢占式，这就是最短剩余时间优先算法了，即一个新就绪的进程比当前运行进程具有更短的完成时间，系统抢占当前进程，选择新就绪的进程执行。

进程	到达时刻	运行时间
P1	0	7
P2	2	4
P3	4	1
P4	5	4

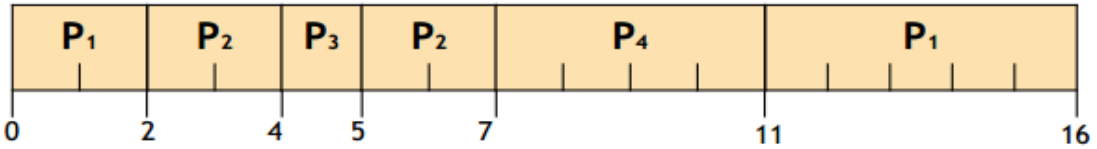
SJF

非抢占式



SRTF

抢占式



缺点：源源不断的短任务到来，可能使长的任务长时间得不到运行，导致产生“饥饿”现象。

最高响应比优先HRRF

- HRRF算法实际上是FCFS算法和SJF算法的折衷既考虑作业等待时间，又考虑作业的运行时间，既照顾短作业又不使长作业的等待时间过长，改善了调度性能。

- 在每次选择作业投入运行时，先计算后备作业队列中每个作业的响应比RP(相应优先级)，然后选择其值最大的作业投入运行。

- RP定义为：$RP = \dfrac{\text{已等待时间} + \text{要求运行时间}}{\text{要求运行时间}}$

$$= 1 + \frac{\text{已等待时间}}{\text{要求运行时间}}。$$

最高响应比优先HRRF

- 响应比的计算时机：
  - 每当调度一个作业运行时，都要计算后备作业队列中每个作业的响应比，选择响应比最高者投入运行。
- 响应比最高优先（HRRF）算法效果：
  - 短作业容易得到较高的响应比
  - 长作业等待时间足够长后，也将获得足够高的响应比
  - 饥饿现象不会发生
- 缺点：
  - 每次计算各道作业的响应比会有一定的时间开销，性能比SJF略差。

谢谢！

沃天宇

woty@buaa.edu.cn