嵌入式操作系统

李曦

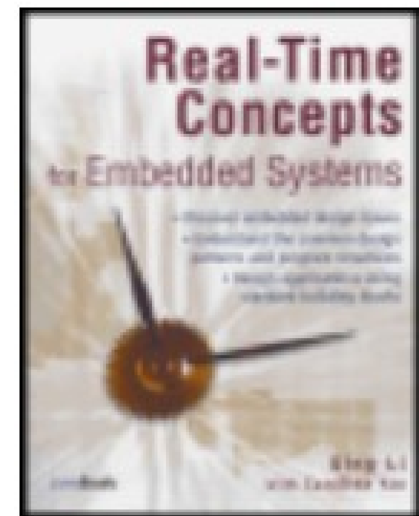
llxx@ustc.edu.cn

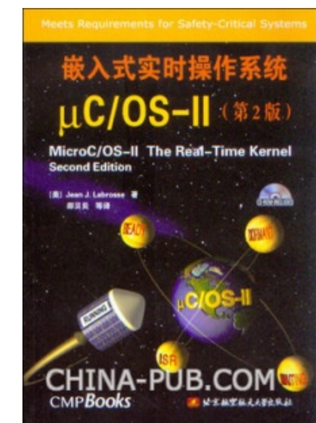
内容提要

- 嵌入式操作系统概述
  - 嵌入式操作系统体系结构
  - 典型的嵌入式操作系统
- RTOS基本概念
  - 编程模型
- RTOS内核功能
- RTOS的性能指标

参考书

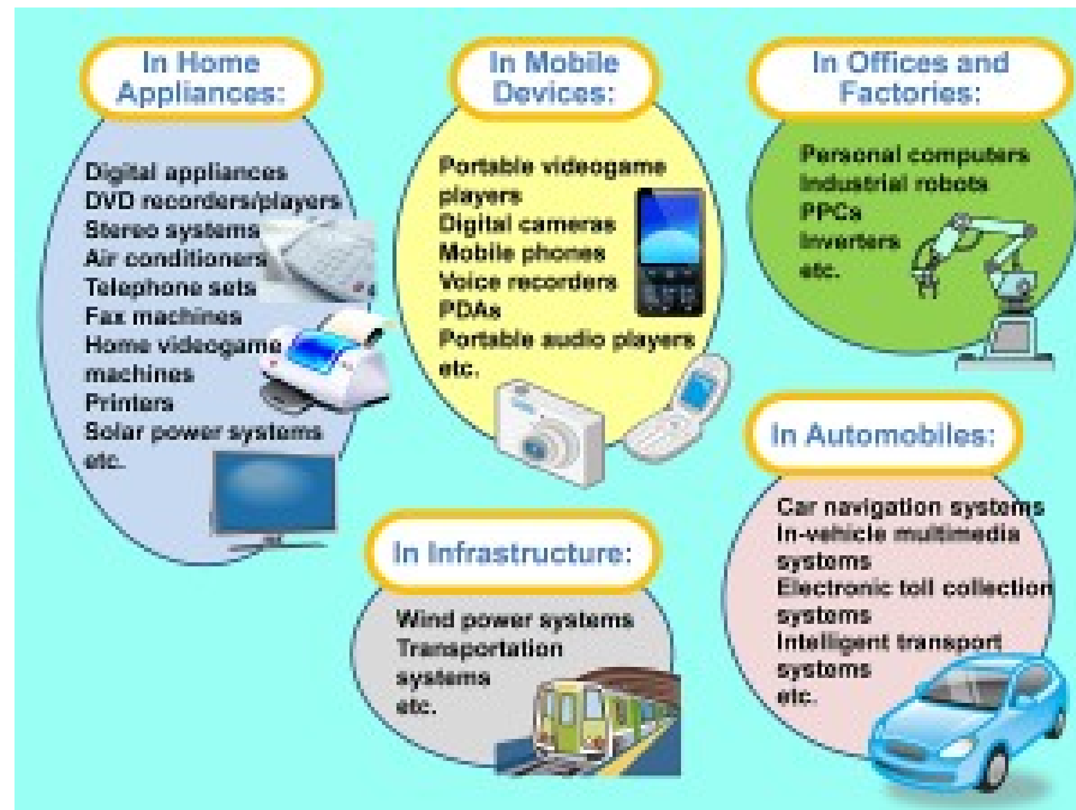
- Real-Time Concepts for Embedded Systems, 2003
  - Qing Li，Wind River Systems, Inc., A lead architect
- 目录
  - 5. 任务
  - 6. 信号量
  - 7. 消息队列
  - 11. 定时器与服务
  - 12. I/O子系统
  - 13. 内存管理
  - 15. 同步与通信机制
  - 16. 常见设计问题
    - 死锁
    - 资源访问控制协议PIP/CPP/PCP
- 嵌入式实时操作系统μC/OS-II，第二版，2002
  - Jean J. Labrosse





Requirements for Embedded SW Development

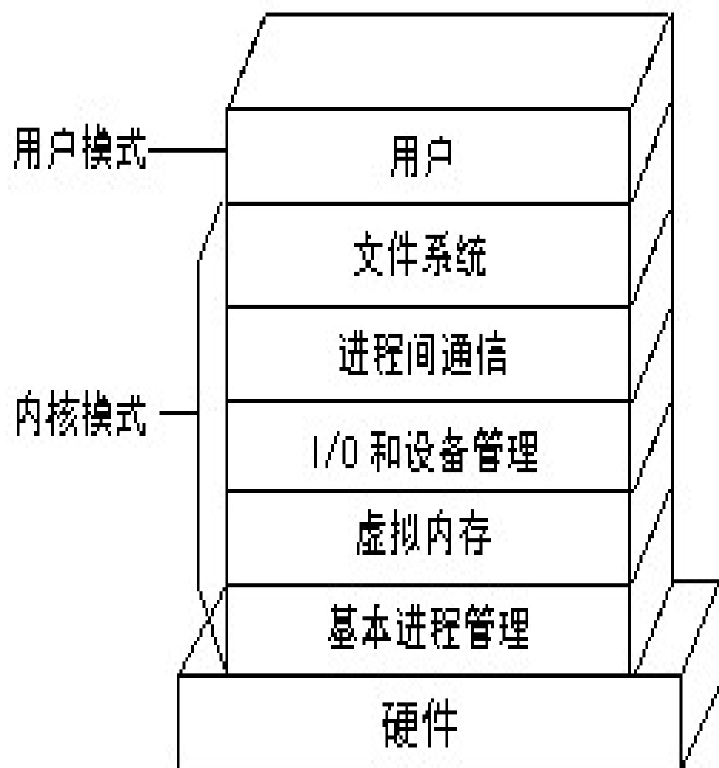
- Increasingly large and complex programs
- Rapid technological progress and shortened development periods
- Demand for reduced development cost

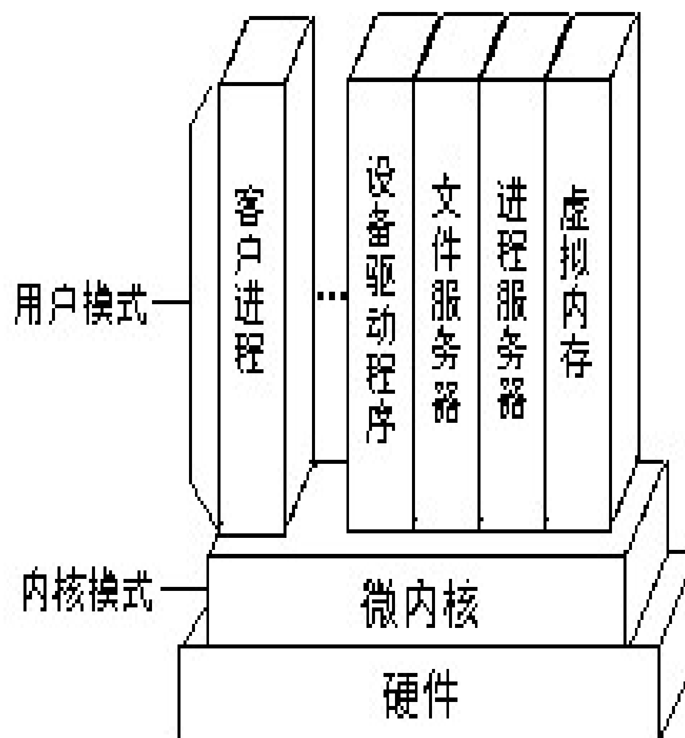


操作系统的分类

- 单任务：顺序执行
  - 系统内只含有一个程序，独占CPU的运行时间
    - 按语句顺序执行该程序。执行完毕，另一程序才启动运行。
    - DOS
- 多任务：并发执行
  - 分时：CPU时间分片，每个时间片执行不同的程序
    - UNIX
  - 优先级：每个程序有不同的优先级，最高优先级者执行
- 通用OS：公平性
- 嵌入式OS：资源(cpu，mem，time。。。)受限
  - RTOS：时间约束
    - FreeRTOS
- 单处理器，多处理器，分布式

OS体系结构：大内核和微内核





OS体系结构由固定变为灵活

- 大内核操作系统（Monolithic Kernel）
  - 将图形、设备驱动、文件系统等全部功能在操作系统内核中实现，运行在内核状态、同一地址空间。
  - 优点：减少进程间通信和状态切换的系统开销，获得较好的运行效率。
  - 缺点：内核庞大，占用资源多，剪裁不易，并且一旦个别驱动程序运行出错，就会导致整个系统崩溃，稳定性、安全性不好。
- 微内核（Micro Kernel）
  - 内核中只实现那些必须由内核实现的基本功能
    - 图形、文件系统、设备驱动、通讯等功能放在内核之外，作为系统服务来提供，这些程序在用户状态下运行。
  - 优点：有一个精炼的内核，便于剪裁、移植。

通用系统架构：Linux

用户程序

函 数 库

用户级

内核级

系统调用接口

文件子系统

高速缓冲

进程控制
子系统

进程间通信

调度程序

内存管理

字符设备

块设备

设备驱动程序

硬 件 控 制

内核级

硬 件

硬件级

嵌入式操作系统

- 在本质上与通用的操作系统没有大的区别
- 对嵌入式系统的硬件有较高的要求
  - 内核本身也增加了系统的额外负荷
    - 代码空间增加ROM用量
    - 内核的数据结构增加了RAM的用量
    - 内核的CPU占用时间为2～5%
  - 许多嵌入式操作系统不划分“系统空间”和“用户空间”
    - 操作系统“内核”与外围应用程序之间不再有物理的边界
    - 系统中所谓“进程”实际上全都是内核线程
    - 应用程序与OS API采用静态编链方式
- 嵌入式OS体系结构采用微内核结构（kernel）
  - 可伸缩、可移植、可剪裁、可配置

嵌入式系统架构

Hardware Independent Software

Applications

I/O System

RTOS libraries

TCP/IP Stack

File System

RTOS Kernel

Hardware Abstraction Layer： BSP&Device Driver

SCSI Driver	Flash Driver	MMU Driver	Cache Driver	Serial Driver	Ethernet Driver	Device Drivers

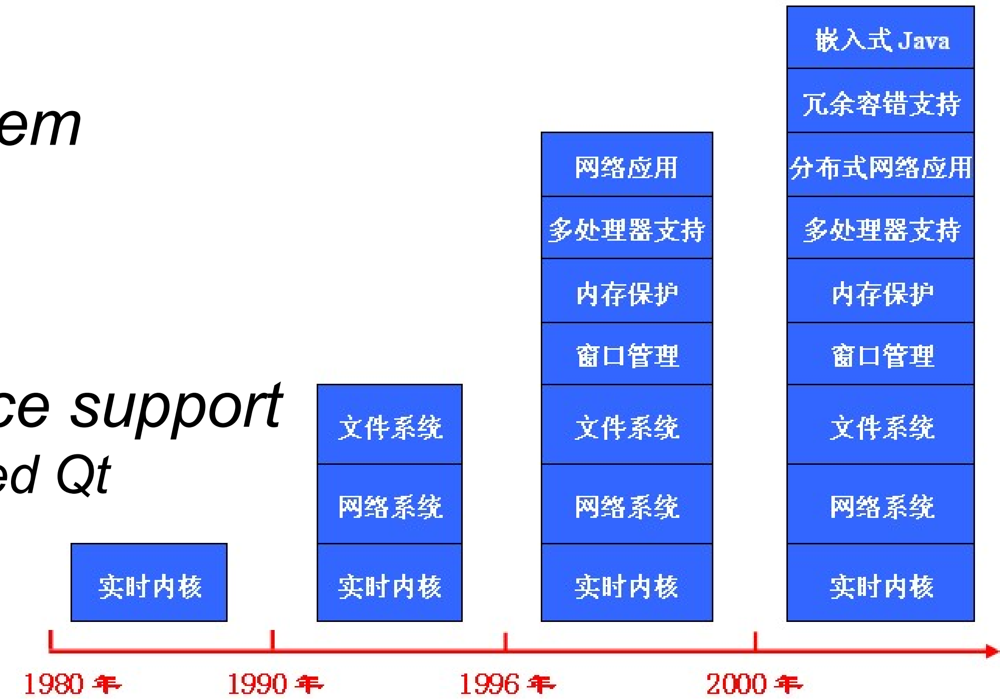
Hardware

实时操作系统

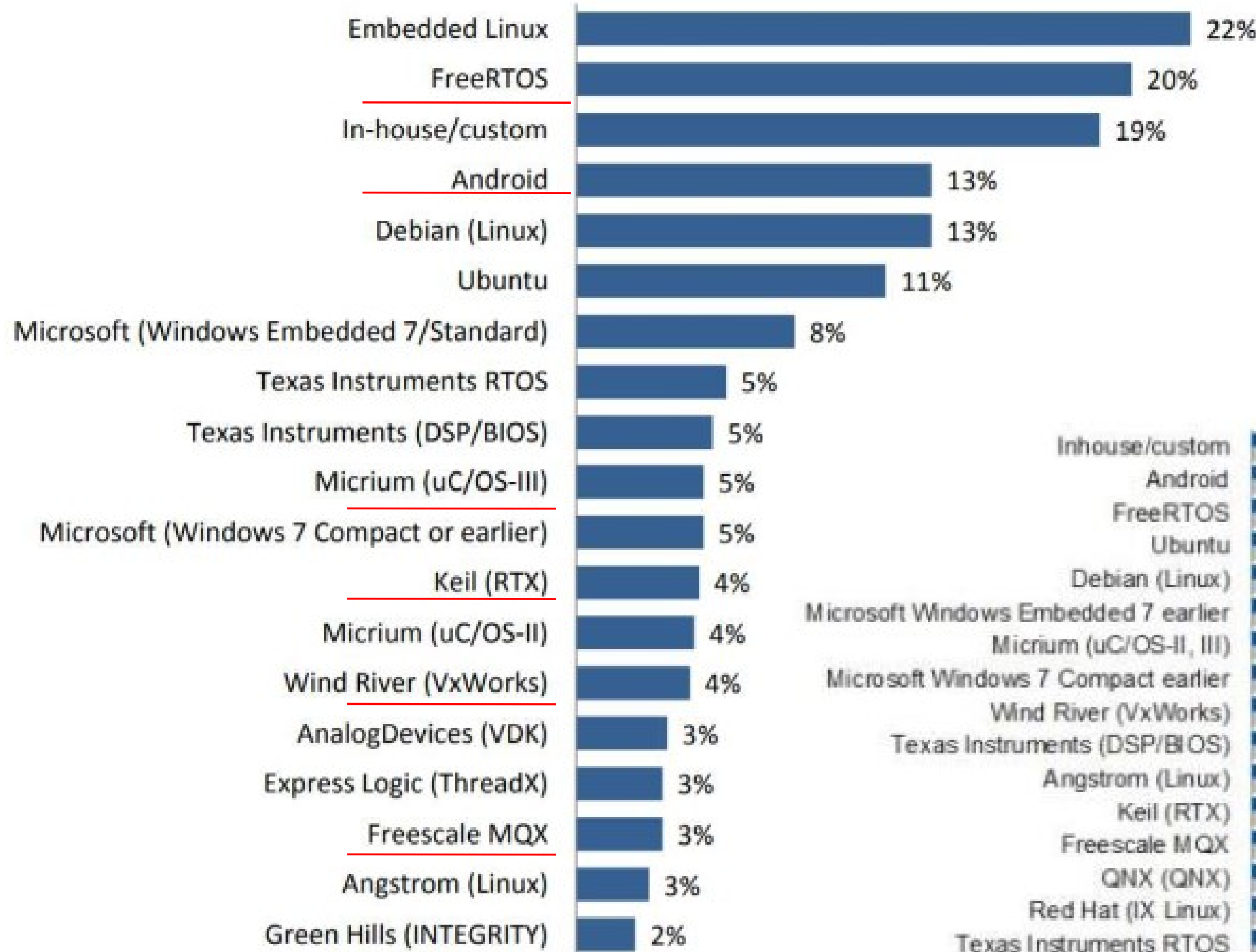
- RTOS的要求
  - 提供任务调度管理机制
    - 基于时间、基于优先级
  - 提供时间管理
  - 时间行为必须可预测
    - 任何服务的执行时间都有界
      - Task dispatch time & switch latency, interrupt latency, synchronization
    - 中断可被禁止
  - 高效：快
- 典型性能指标
  - 内核大小：几K～几百K
  - 调度时间片：1ms
  - 实时进程/线程响应时间：20～40us
  - 普通线程响应时间：20us～几百ms

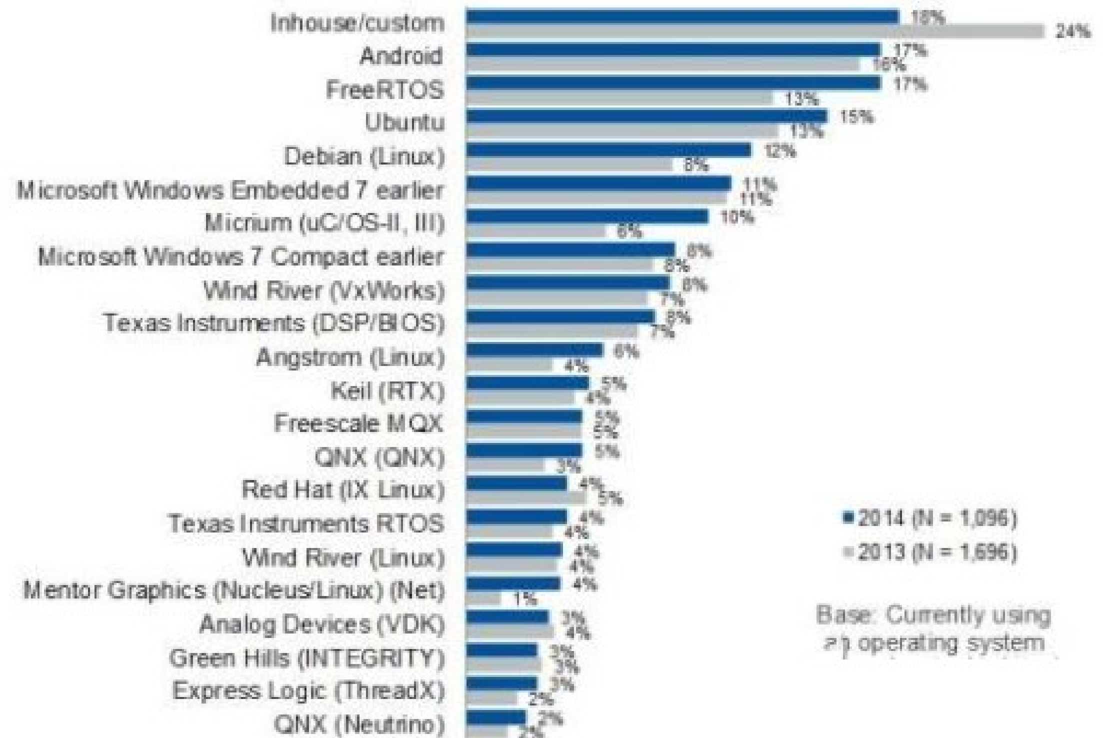
Real-Time Operating System

- Kernel (RTK)
  - Scheduling
  - Mutual exclusion
- Executive (RTE)
  - Inter-task communication & synchronisation
  - Memory management
- *RTOS services*
  - *File management System*
    - *FAT file system*
  - *Networking*
    - *E.g. TCP/IP, CAN*
  - *Graphical User Interface support*
    - *E.g. OpenGL, Embedded Qt*



嵌入式操作系统（近200种）





2017,2014,2013

VxWorks嵌入式实时操作系统

- 支持主流的32位CPU
  – x86、 68K、PowerPC、MIPS、ARM等
- 基于微内核结构
  – 可裁剪性和可配置性相当出色
    - 由400多个相对独立的、短小精悍的目标模块组成
- 主要思想
  – 在嵌入式系统中最大限度地实现内核的时间可预测性
  – 根据用户定义的任务优先级对任务实现调度
    - 给用户最大的控制权
  – 兼容POSIX实时系统标准

VxWorks Architecture

HOST

TORNADO® 2.2 Embedded Development Tools

Core TORNADO Tools

Compiler

Debugger

Project Facility

TORNADO Dynamic Visualization Tools

RTOS Event and Object Analysis

Application Data and Event Analysis

Code Coverage Analysis

Memory Analysis

Source Code Execution Analysis

Third-party Tools

Target Server

Simulation Environment

Virtual

Ethernet

Serial Line

Custom

Hardware Runtime Control JTAG/BDM

TARGET

VxWORKS® 5.5 Scalable Runtime System

Real-time Embedded Application

Graphics

Java Support

POSIX Library

File Systems

Connectivity

I/O Management

Multiprocessing

Virtual Memory

Networking Protocols

Debug Agent

Core OS

WIND® *Microkernel*

BSP

Wind River Reference Hardware

VxWorks基本构成模块

- 高效实时微内核wind
  - 基于优先级的任务调度
  - 任务同步和通信
  - 中断处理
  - 定时器
  - 内存管理
- I/O处理系统
  - 与ANSI C兼容的I/O处理系统，主要包括：
    - UNIX缓冲I/O处理系统，
    - 面向实时的异步I/O处理系统
- 本机文件系统

VxWorks基本构成模块（续）

- 网络处理模块
  - Vxworks网络处理模块能与许多运行其他协议的网络进行通信，如TCP/IP、NFS、UDP、SNMP、FTP等
- 虚拟内存模块VxVMI
  - VxVMI主要用于对指定内存区的保护，如内存块只读等
- 共享内存模块VxMP
  - 主要用于多处理器行运行任务之间的共享信号量、消息队列、内存块管理
- 板级支持包BSP
  - 提供各种硬件的初始化、中断的建立、定时器、内存映象

VxWorks的评价

- 追求实时性，不以通用OS为设计目标。
  - 为保证实时性，去掉了一些OS模块
    - 如：在内存管理中没有采用页面管理模式，而是采用平面式内存。
  - 资源共享和优先级继承机制
  - 采用最优化的上下文切换和中断返回机制
  - 内核从不禁止非屏蔽中断 NMI (non-maskable interrupts)
- 任务调度采用的是基于优先级的抢占式任务调度模式
  - 优先级分256级(0-255)
  - 用户可以动态的改变优先级，但是这种做法不提倡
  - 用户可以锁定一个任务使它不被更高的任务或中断抢占
  - 允许使用固定优先级响应时间来检查任务调度的性能

VxWorks缺点

- 保证时限要求是设计者自己的任务
  - 系统的灵活性带来的弊端
- 不支持很多应用和APIs
  - 只支持部分POSIX标准的函数集
- 尽管采用了平板式内存管理，但是由于动态分配内存，仍然存在内存段，这样仍然存在时间不可预测性
- 应用领域主要局限在对实时性要求较严格的硬实时系统中

RTX，KEIL公司设计

- 基于Cortex-M3/M4架构
  - 支持零延迟中断特性
- 任务切换等性能优于其它RTOS
  - 上下文切换时间< 300 cycles
- 调度：循环、抢先和协作
- 不限数量的任务，*每个任务*都具有254优先级
- 不限数量的邮箱、信号、互斥函数和计时器
- 支持多线程和线程安全运算
- µVision IDE/调试器完全支持RTX

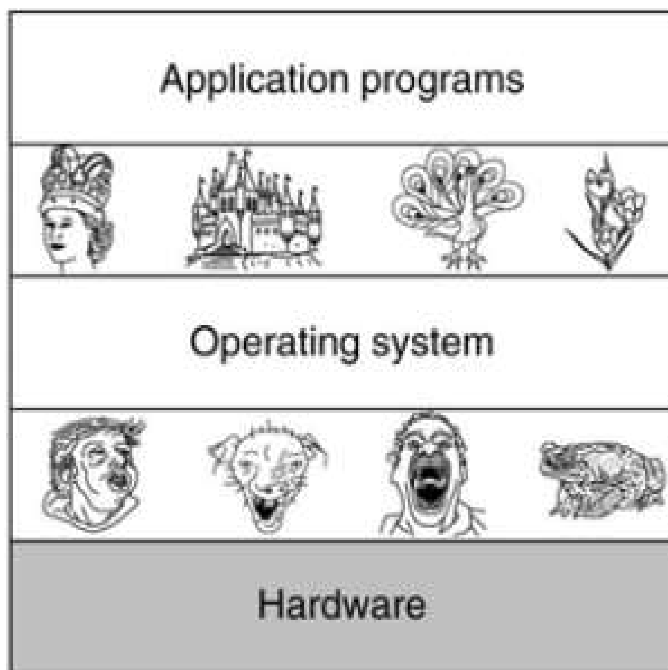
嵌入式实时内核μC/OS

- 90年代初，免费的公开源码实时操作系统
  - 代码尺寸小，结构简明，易学、易移植。源代码的绝大部分是用C语言，汇编语言写的部分只有200行左右。
  - μC/OSII最小可达2KB，最小数据RAM需求10KB。
  - 被移植到众多CPU上
    - Analog 设备公司：AD21xx
    - ARM公司：ARM 6, ARM7
    - 日立公司：64180，H8/3xx，SH系列
    - Intel公司：80x86,Pentium, Pentium II, 8051,8052, MCS-251,80196,8096
    - 三菱公司：M16，M32
    - 摩托罗拉公司：PowerPC，68K,CPU32, CPU32＋,68H11，68HC16
    - 飞利浦公司：XA
    - 西门子公司：80C166，TriCore
    - TI公司：TMS320
    - Zilog公司：Z—80，Z—180
- 应用普遍
  - 照相机、医疗器械、音响设施、发动机控制、网络设备、自动提款机、工业机器人

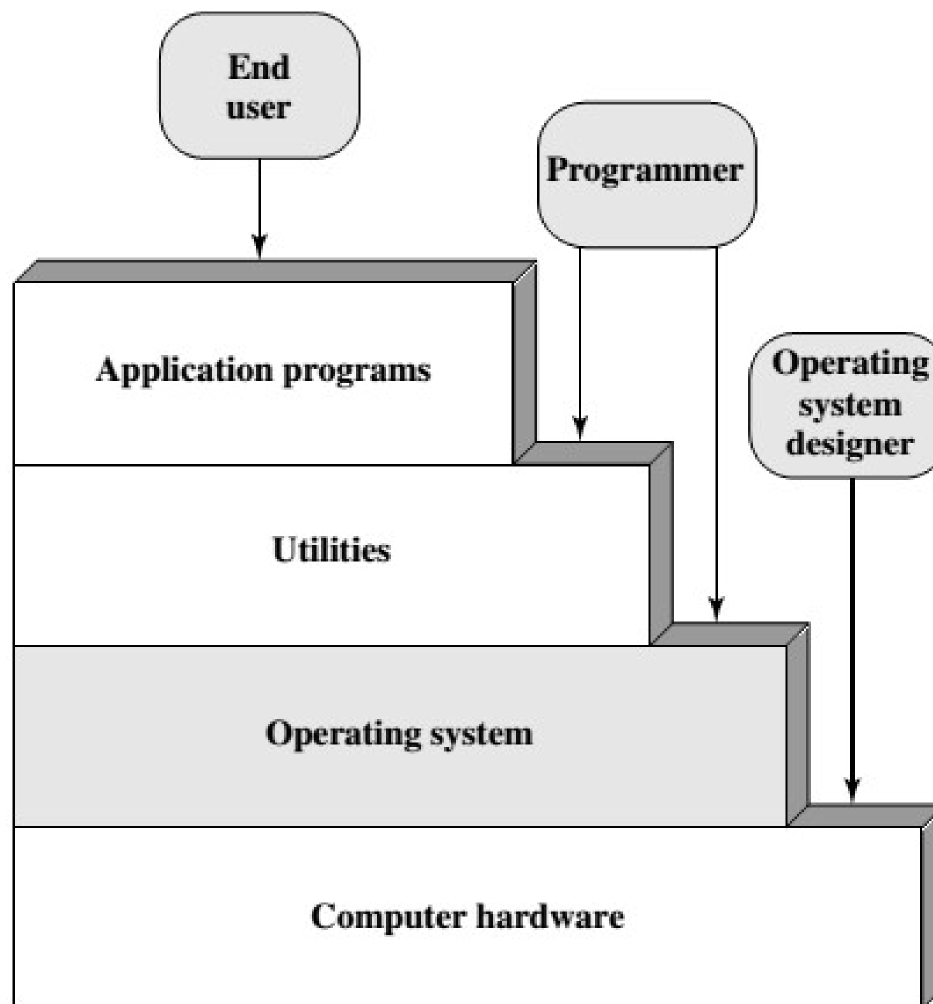
POSIX标准

- POSIX（Portable Operating System Interface for UNIX）
  - 为标准化类UNIX操作系统所必须具有的特征和接口而制定：增强软件的可移植性
- 实时扩展
  - 1003.1b：一个用于实时编程的标准
    - “能够在限定的响应时间内提供所需水平的服务”
  - 1003.5b：一个相当于 1003.1b（实时扩展）的 Ada 语言的 API

OS的服务与接口

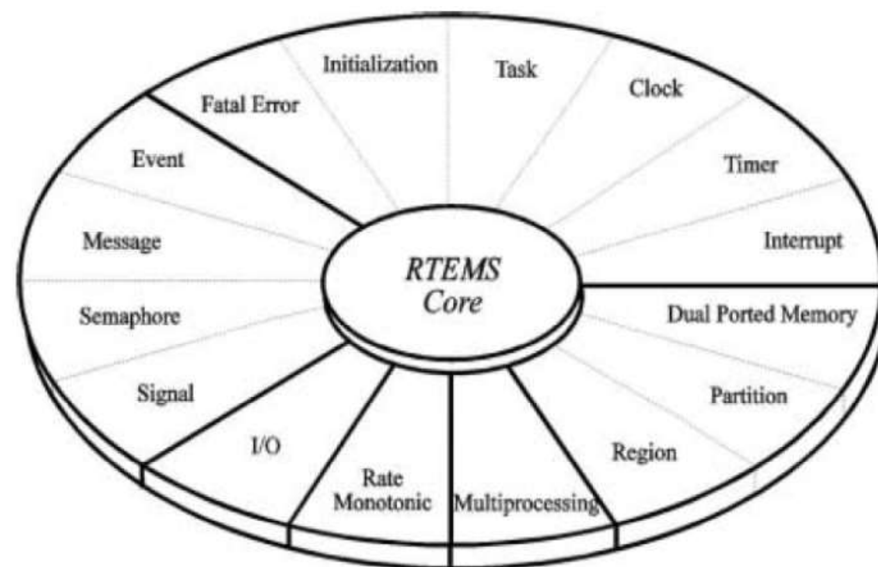


- 命令行接口
- 编程接口



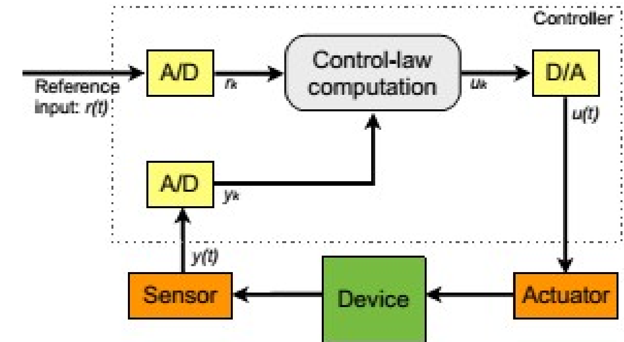
内容提要

- 嵌入式操作系统概述
  - 嵌入式操作系统体系结构
  - 典型的嵌入式操作系统
- RTOS基本概念
  - 单任务模型
  - 多任务编程模型
    - 同步、互斥、通信
  - I/O子系统
  - 存储管理
- RTOS的性能指标

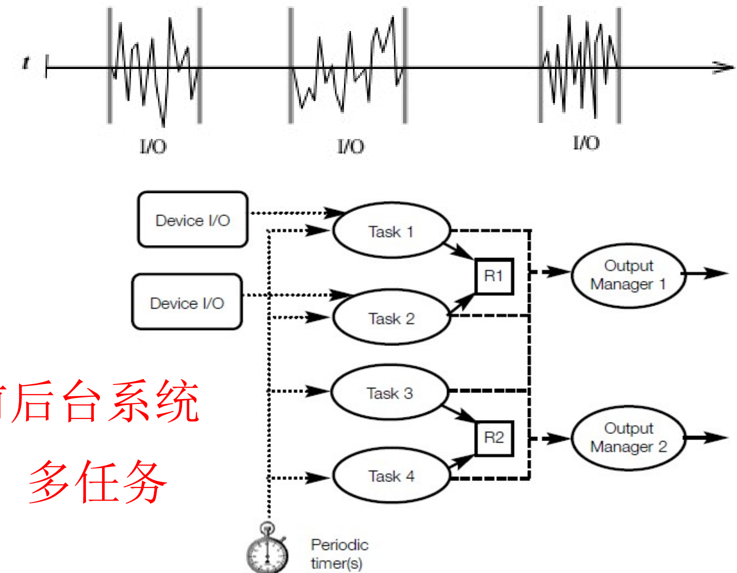


实时嵌入式系统的应用特征

- 控制系统：基于cybernetcs
  - Singal：continuous
    - sampling，I-C-O
    - 周期：period，rate
  - control loops：“sense-and-control-then-delay” cycle

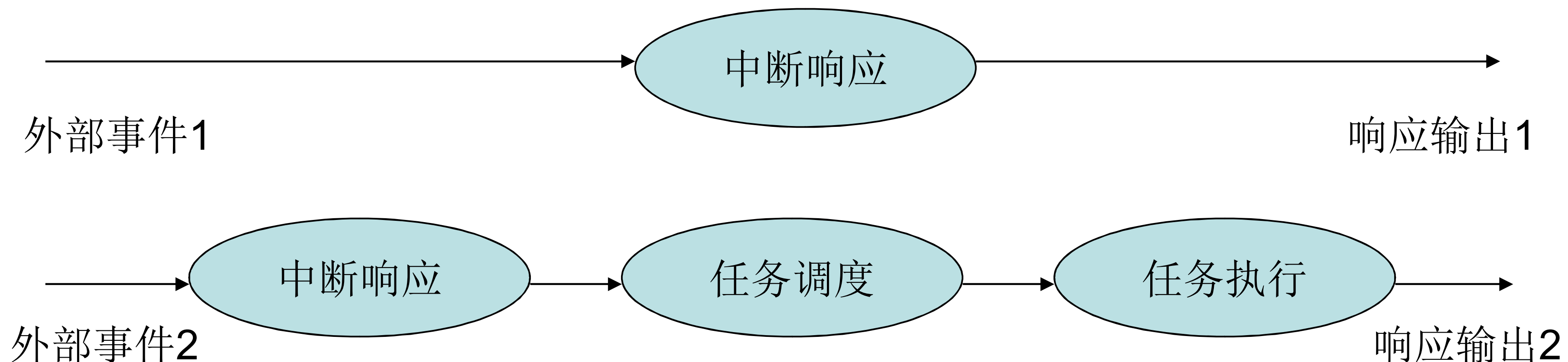


- 嵌入式计算机：基于event-action模型
  - Event：discrete，spontaneous
    - Monitoring
    - sporadic，最小到达间隔
  - ET：中断驱动
    - 模式1：在ISR中完成对事件的响应。前后台系统
    - 模式2：中断触发任务，任务完成响应。多任务
  - TT：时钟驱动，轮询式



外部事件的响应模式：ET/TT

- 模式1：在ISR中完成对事件的服务。前后台系统
  - 响应时间＝中断响应时间＝中断延迟时间＋中断处理时间
- 模式2：中断触发任务，任务完成事件服务。多任务系统
  - 优点：减少中断服务时间，有利于高优先级任务的实时性
  - 响应时间＝中断响应时间＋任务调度时间＋任务执行时间
    - 任务调度算法、任务数、任务通信、资源共享
    - 事件发生频率：频率越低，响应时间越容易满足
    - 任务执行时间：最坏时间分析(WCET)

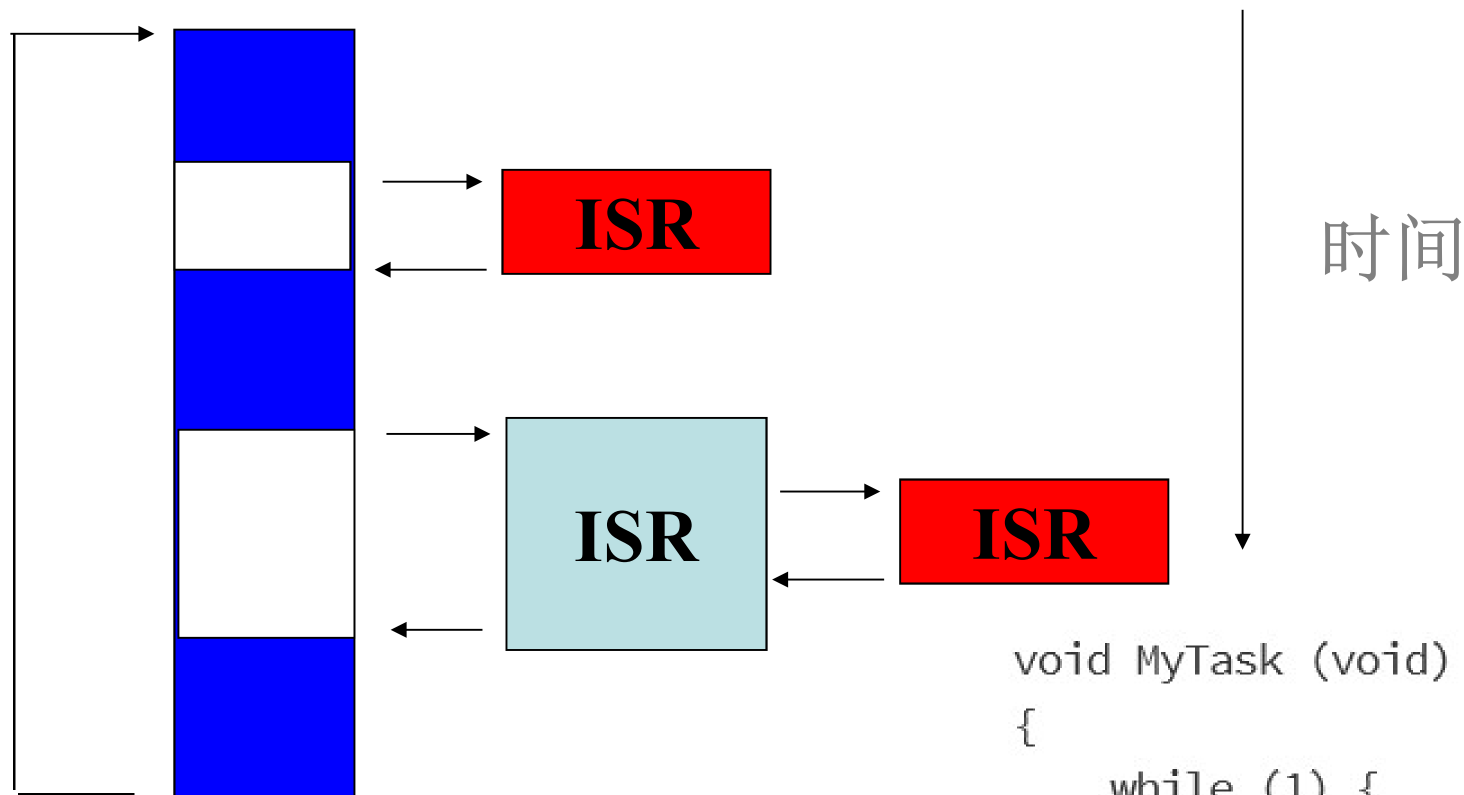


控制软件编程模型/范式

- 简单系统控制：单任务，无需OS的支持
  - Run-to-Completion Tasks
  - Endless-Loop Tasks：non-terminal
    - 直到系统关闭或者出现异常停止运行
  - 前后台系统
  - cyclic executive：cooperative multitasking（activities）
- 复杂系统控制：多任务，Divide and Conquer
  - Threads模型：多任务并发异步执行
    - CPU轮流服务于一系列任务中的某一个
    - 使CPU的利用率得到最大的发挥
    - 使应用程序模块化、层次化。
  - 使用嵌入式操作系统进行任务调度

前后台系统：ET例，多任务（ISR）？

后台循环　　　　　　前台中断




```
void MyTask (void)
{
    while (1) {
        Wait for an event to occur;
        Perform task operation;
    }
}
```

- 实时性要求较低、运行周期较长，且计算复杂的上层控制策略算法任务作为后台
- 实时性要求高的采样等任务作为前台

前后台：EMSBench程序架构

main()
Read sensors
Switch sensor data
Calculate fuel/ignition
Switch control data
COM

RTIISR()
1ms

SecondaryRPMISR()
Calculate RPM

IgnitionFireISR()
Activate Ignition

RPM Data

IgnitionFire Data

Fuel/Ignition Constants

PrimaryRPMISR()
Injection
IgnitionDwell
IgnitionFire

Injection Data

InjectorXISR()
Injection active: set deactivation
Injection not active: (set queued activation)

IgnitionDwell Data

IgnitionDwellISR()
Charge Ignition Coil

Data access
Set IRQ trigger
Set trigger flag

cyclic executive：TT例

```
int main(void) {
    // initialization code here
    while (1) {
        currTemp = tempRead();
        lcdWrite(currTemp);

        // waste CPU cycles until 50 ms

        currTemp = tempRead();
        // do other stuff

        // waste CPU cycles until 100 ms
    }
}
```

Timing：minor cycle，major cycle？

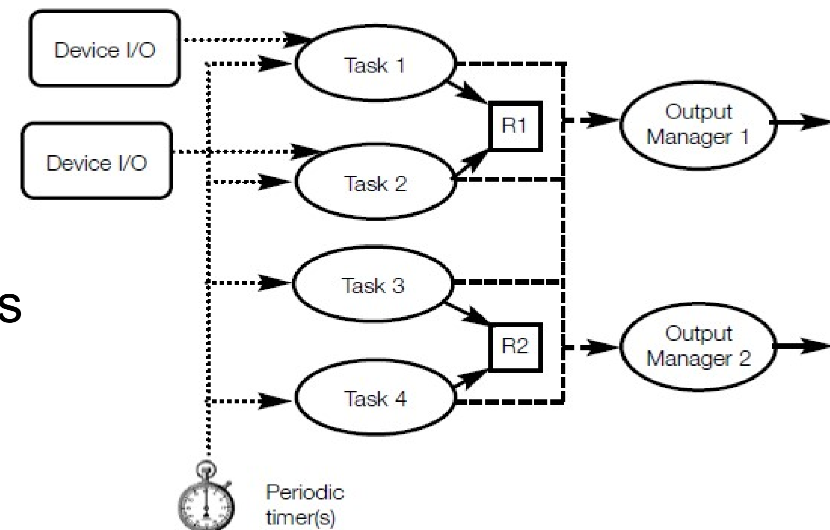
超级循环：轮询式，时间驱动

系统上电，调用main函数

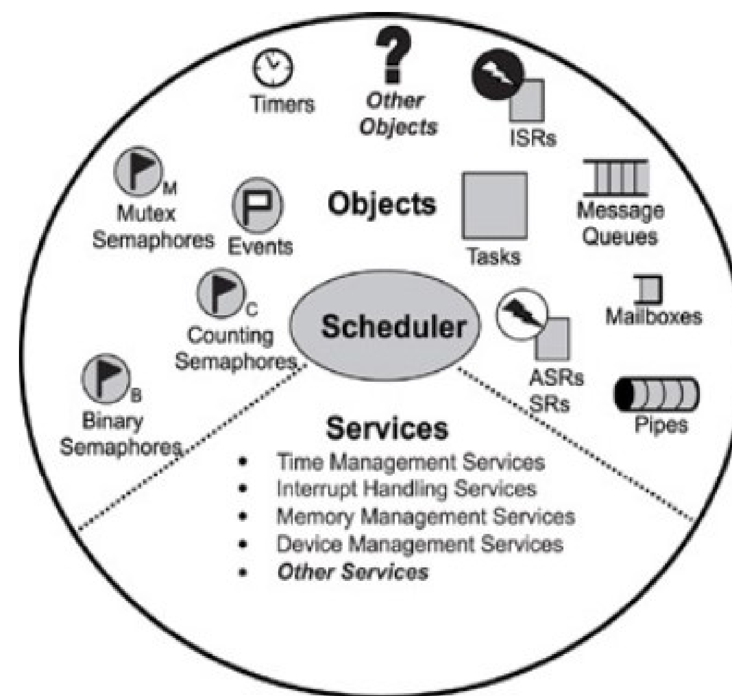
首先初始化

然后进入超级循环，逐一轮询外部事件

直到系统关闭或者出现异常停止运行



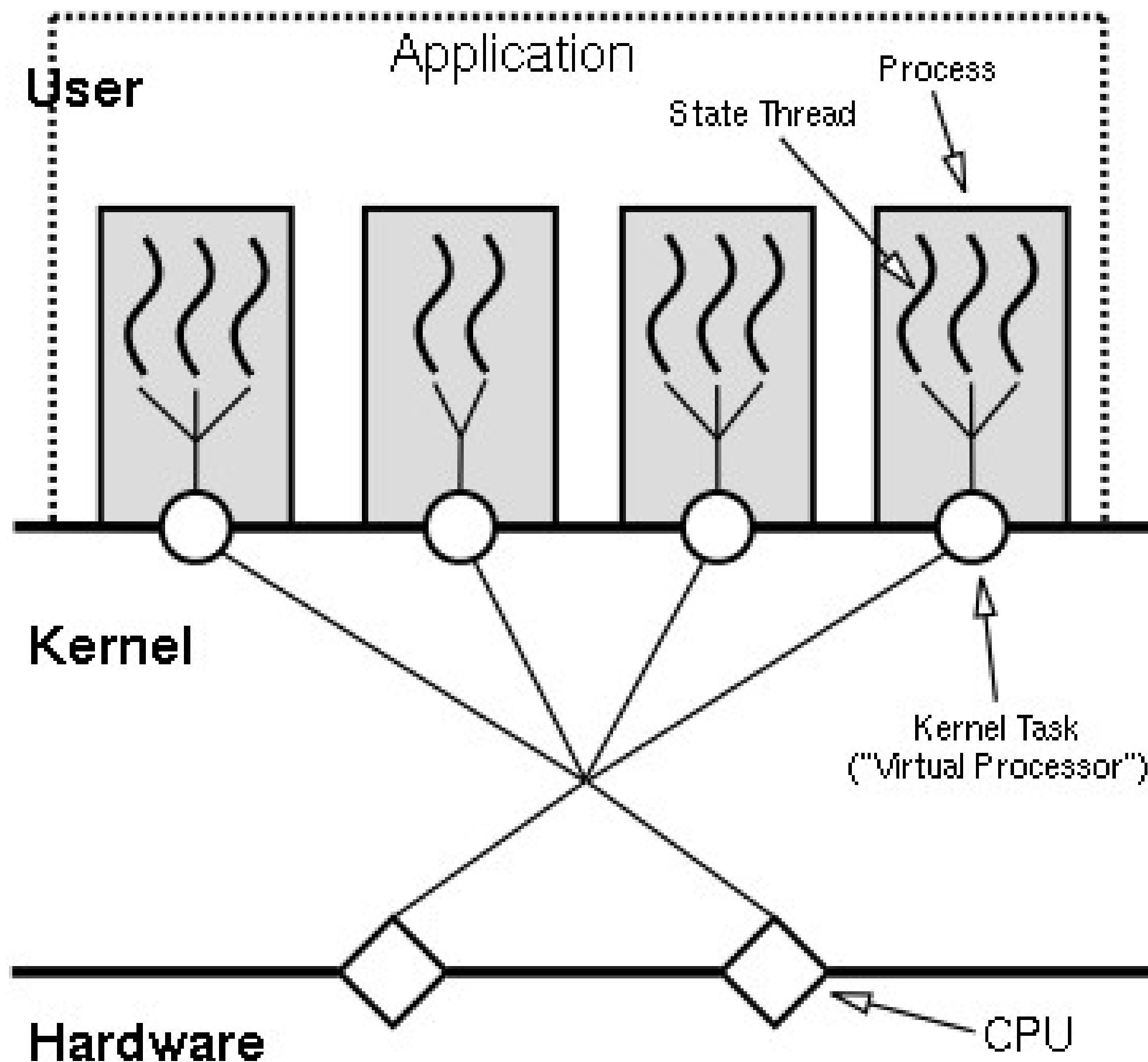
RTOS内核实现



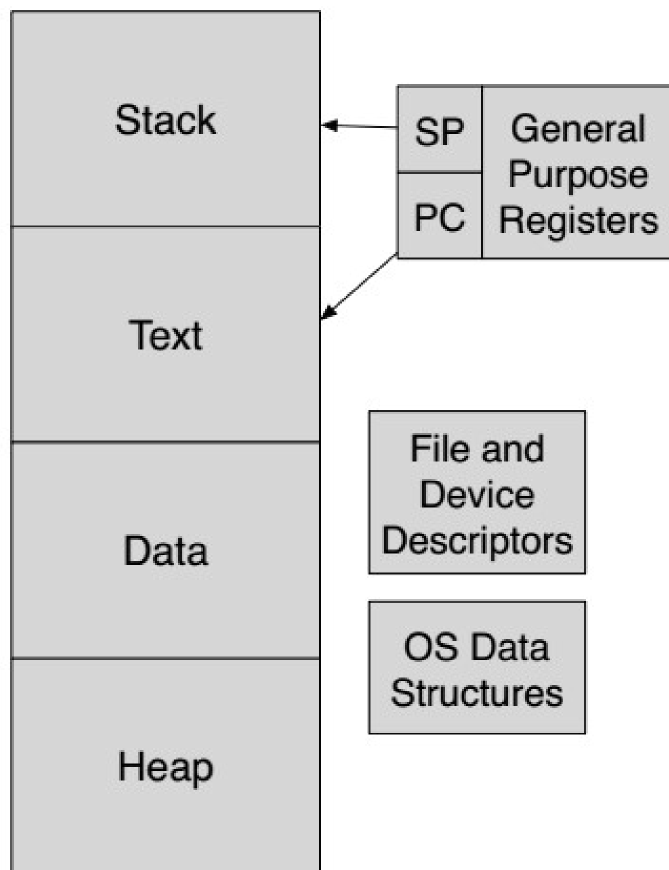
- 组成：
  - 调度器：决定当前执行的任务
  - 对象：支持应用开发的结构
    - 任务、信号量、消息队列。。。
  - 服务：内核完成的操作
    - 定时、中断处理、资源管理。。。
- 功能
  - 任务管理
    - 调度、创建、删除、挂起
  - 中断/异常管理
  - 任务互斥、同步与通信管理
    - 信号量、消息、事件、管道、异步信号、共享内存等
  - 时钟管理：提供高精度（<10ms）系统时钟，定时、计时管理
  - 内存管理：有限地使用虚存技术，减小开销，保证可预测性
- 毫微内核：只包含time和调度

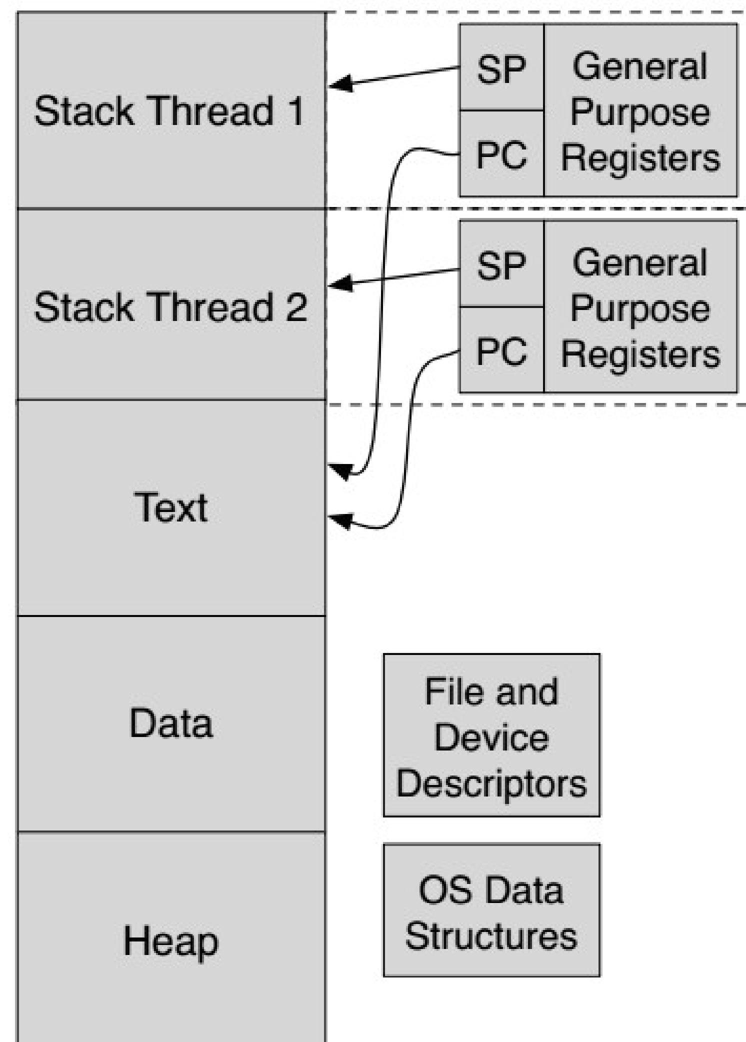
通用计算：应用、process、thread

- 进程
  - 资源分配
- 线程
  - 调度执行



进程与线程模型





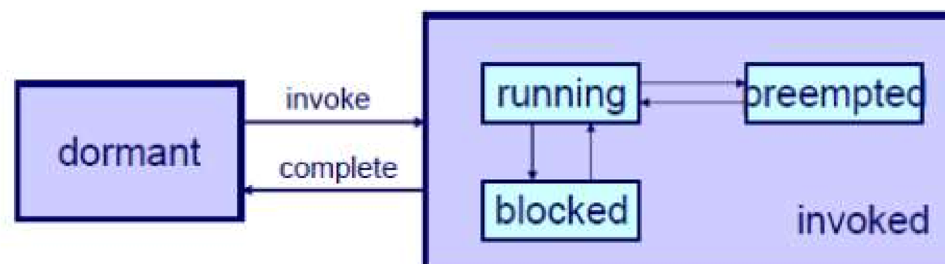
嵌入式计算：任务、thread

- 任务：逻辑概念（程序块，线程）
- 基于线程的任务模型：扁平化
  - 一个任务一个线程
    - 是OS可分派的一个执行单元
    - 有自己的一套CPU寄存器（上下文）和栈
    - 由任务控制块 (TCB)进行定义
  - *每个任务被赋予一定的优先级*
- 任务间关系
  - 资源共享型：独立，互斥，同步
  - 相互合作型：precedence关系
- 多任务执行：并发，异步

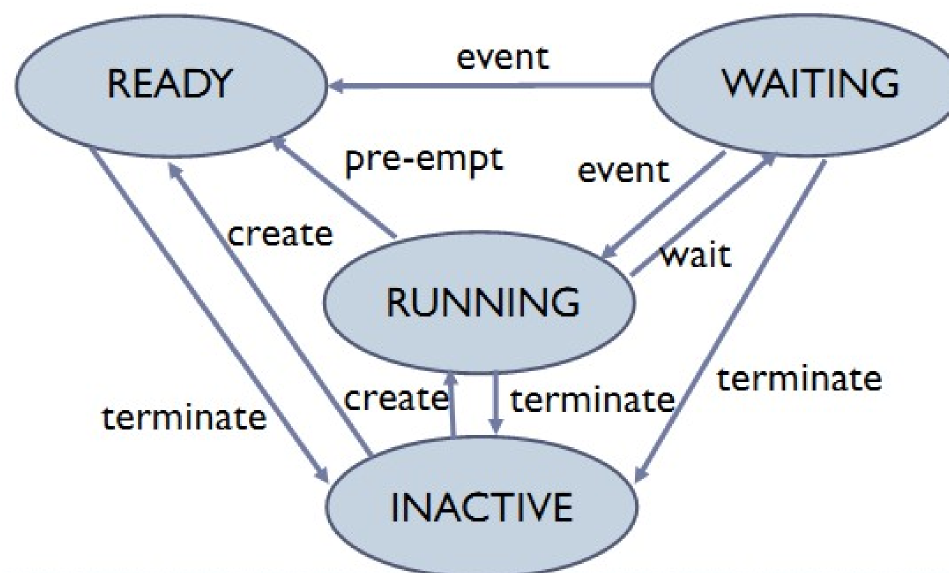
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
Stack Pointer (R13)
Link Register (R14)
Program Counter (R15)
CPSR
SPSR

任务管理的基本概念

- 任务管理
  - TCB
  - 任务状态
  - 并发与任务切换
- 资源管理
  - 可重入型函数
  - 临界资源
  - 代码的临界区
  - 互斥
  - PIP和PCP
- 任务间通信
  - 共享内存
  - 消息传递



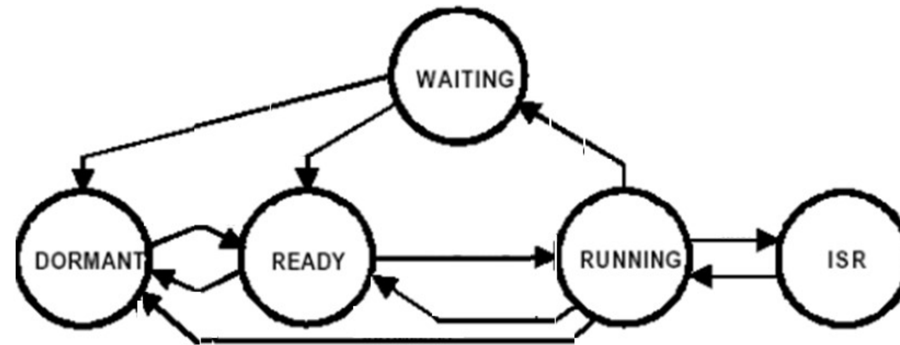
Tasks are invoked periodically or by events

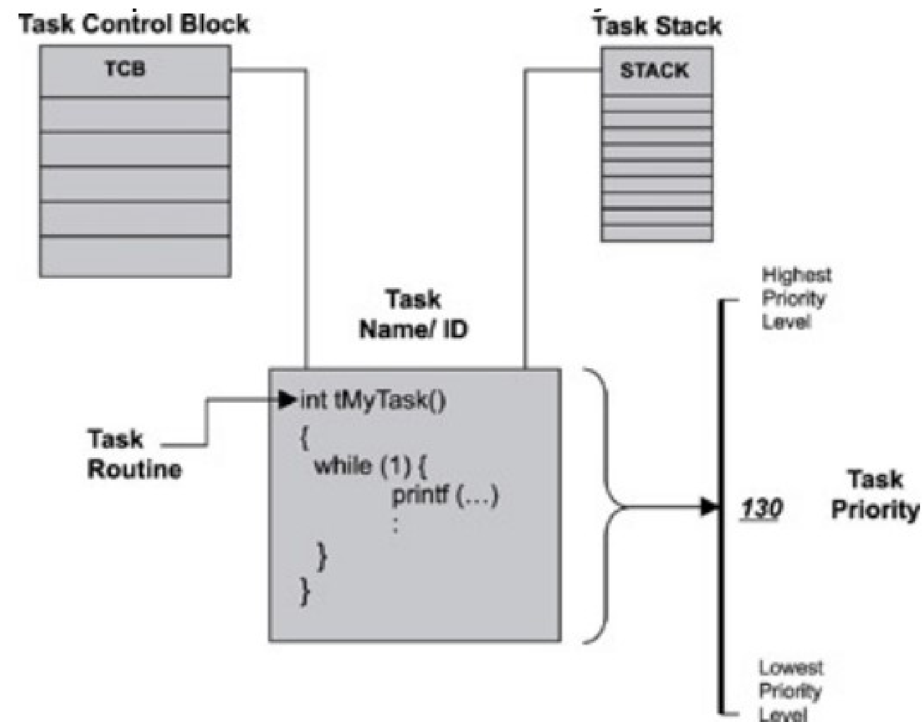


任务TCB（Task control block）

Task Control Block

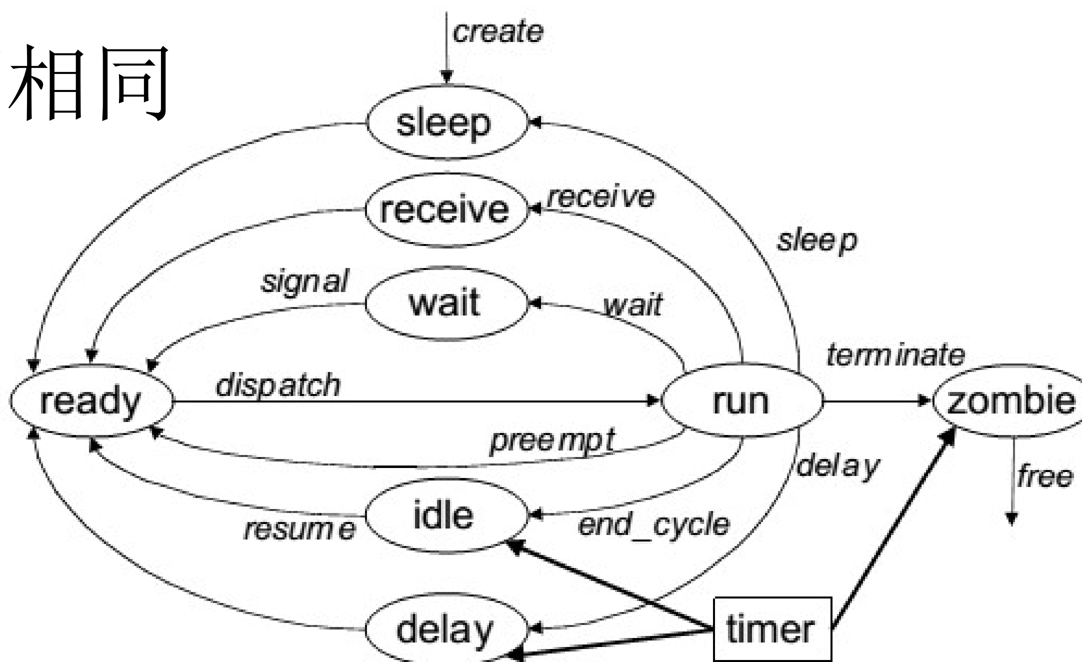
task identifier
task address
task type
criticalness
priority
state
computation time
period
relative deadline
absolute deadline
utilization factor
context pointer
precedence pointer
resource pointer
pointer to the next TCB

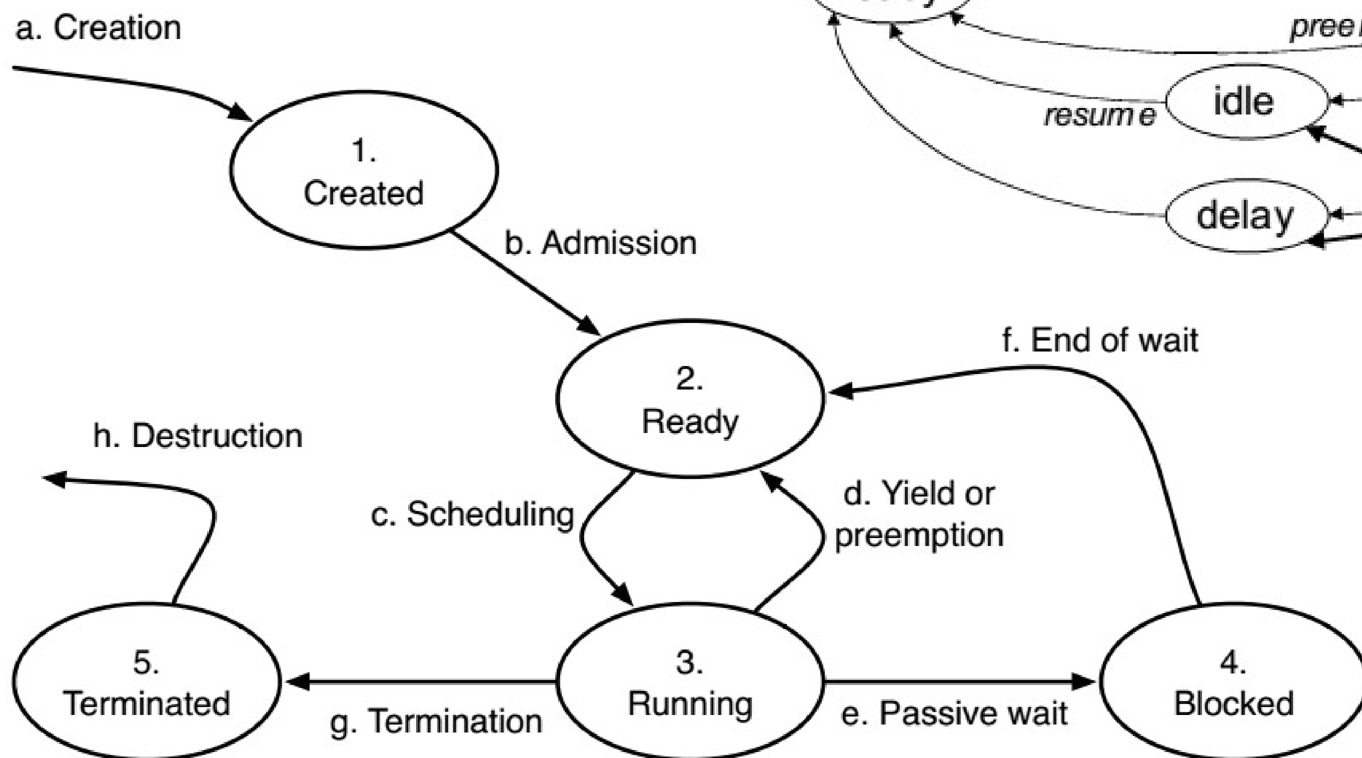




任务状态

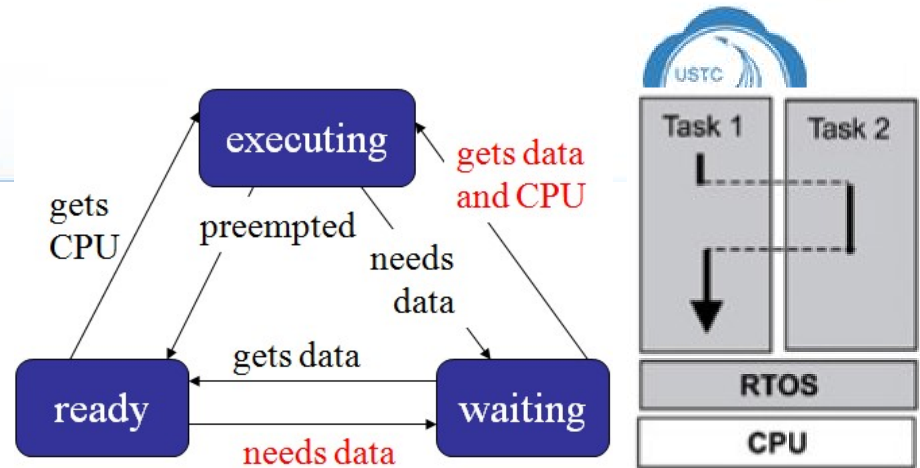
- 状态机：OS各不相同
- 就绪队列：唯一
- 等待队列：多个

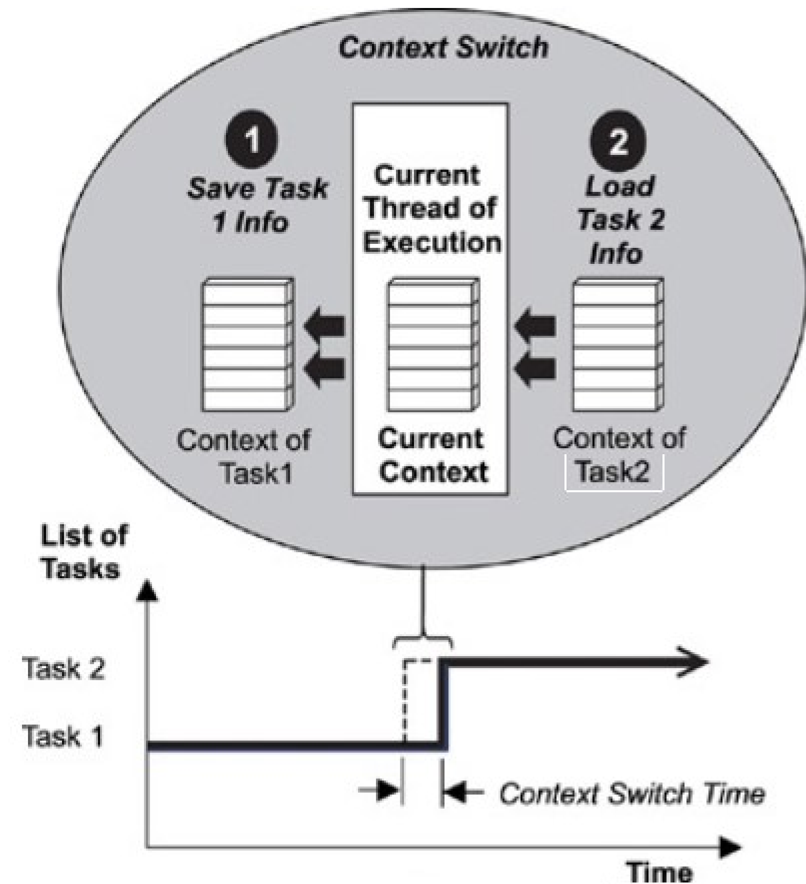




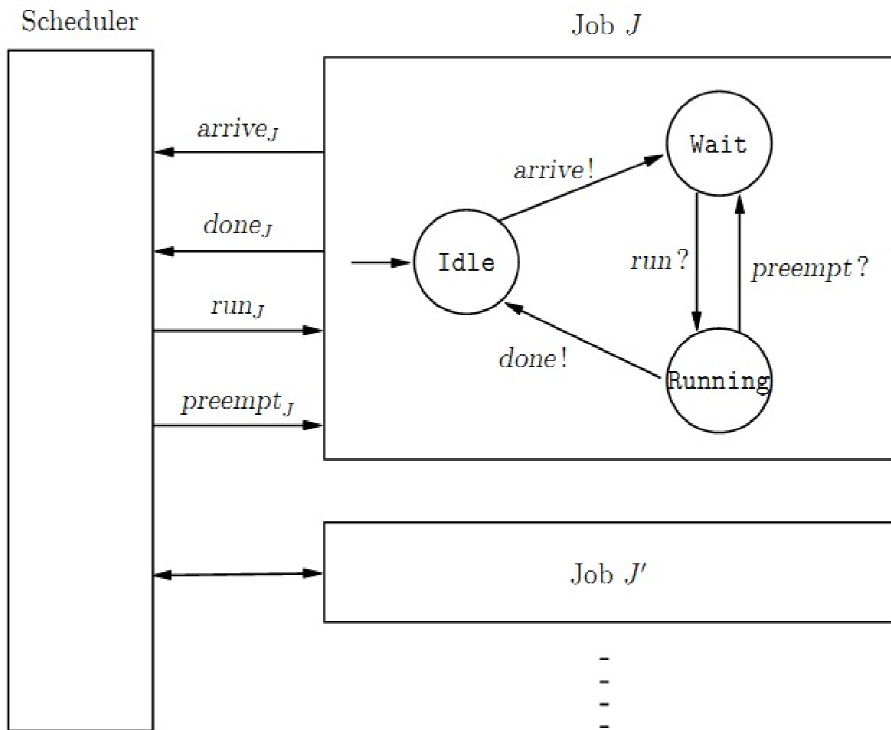
多任务切换

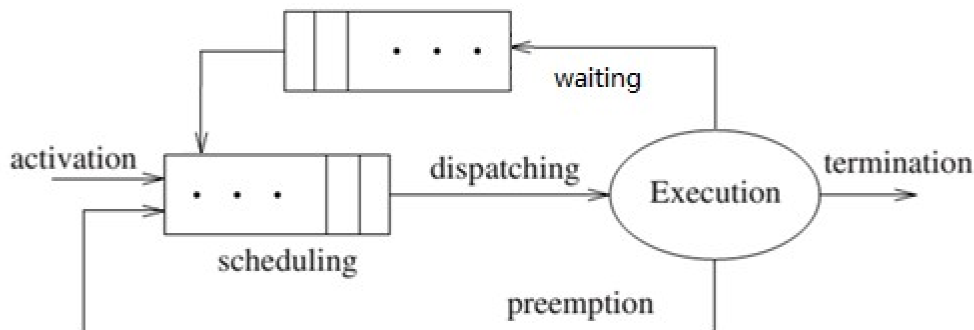
- 任务Context
  - 任务的CPU寄存器状态等
  - 所需硬件支持？
- 任务调度Scheduling
  - 选择下一个任务TCB
    - 调度策略
    - *资源访问控制*
  - 任务调度时机？
    - done：run to completion
    - preempted：ET/TT
    - blocked：互斥、同步
    - yield：延时
- 任务分发Dispatch
  - context switch
  - 执行流切换（nPC）
    - 启动被调度的任务执行





Scheduler：scheduling and dispatching

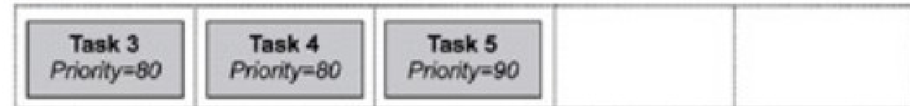




1 First-Step: State of Task-Ready List

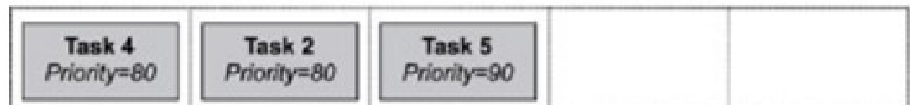
2 Second-Step: State of Task-Ready List

3 Third-Step: State of Task-Ready List



4 Fourth-Step: State of Task- Ready List

5 Fifth-Step: State of Task-Ready List



Task-ready-list的变化情况

设：T3占用T2的资源，故（5）

任务调度（Scheduler）

- 调度策略
  - 任务排序、任务开始时间、分配CPU
  - 评价：利用率、延迟、完成时间（makespan）
- 基于时间片：按时间额度*quantum*占有CPU
  - 任务切换条件
    - 当前任务的时间片结束
    - 当前任务所需的资源被其他任务占用
    - 当前任务在时间片还没结束时已经无事可做
- 基于优先级：高优先级任务占有CPU
  - 合作式（non-preemptive，非抢占、非剥夺）
    - 任一时刻只有一个任务是活动的，可预测
    - 程序员参与调度控制：APP代码中设置调度点
  - 抢占式（preemptive，占先式、剥夺式）

优先级调度理论(priority)

- 每个任务都有其优先级
  - 允许同优先级？
- 静态优先级
  - 应用程序执行过程中诸任务优先级不变
    - 诸任务以及它们的时间约束在程序编译时是已知的。
- 动态优先级
  - 应用程序执行过程中，任务的优先级是可变的
- 如何确定优先级？

可抢占式调度（preemptive）

可剥夺型，占先式型

- 当系统响应时间很重要时，要使用占先式（preemptive）内核。
  - 最高优先级的任务一旦就绪，总能得到CPU的控制权。
  - 当一个运行着的任务使一个比它优先级高的任务进入了就绪态，当前任务的CPU使用权就被剥夺了（被挂起），那个高优先级的任务立刻得到了CPU的控制权。
- 使用占先式内核时，应用程序不可使用不可重入型函数。
  - 如果调入可重入型函数时，低优先级的任务CPU的使用权被高优先级任务剥夺，不可重入型函数中的数据结构有可能被破坏。

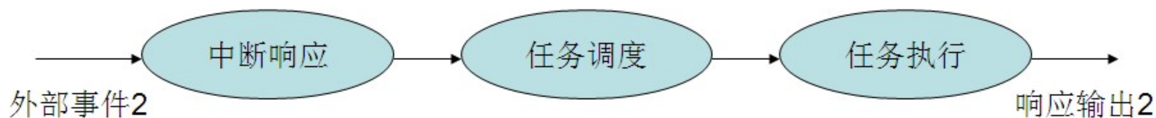
不可抢占式（non-preemptive）

协作式（cooperative multitasking）

- 中断服务使一个高优先级的任务由挂起变为就绪
  - 中断服务结束后控制权回到被中断的任务，直到该任务主动放弃CPU的使用权，高优先级的任务才能获得CPU的使用权。
- 运行着的任务占有CPU，不必担心被别的任务抢占。
  - 几乎不需要使用信号量保护共享数据。

- 问题：高优先级任务的响应时间不确定
  - 实时性取决于最长任务的执行时间
    - 任务已经进入就绪态，但还不能运行，也许要等很长时间，直到当前运行着的任务释放CPU。

不可抢占式 vs. 可抢占式时序图



低优先级任务

（1） （2）

ISR （3）

（4）

中断服务程序使
高优先级任务就绪

（5）

（6）

高优先级任务

（7）

低优先级任务释放
CPU使用权

低优先级任务

中断服务程序使
高优先级任务就绪

（1） （2）

ISR （3） 高优先级任务

TIME

（4）

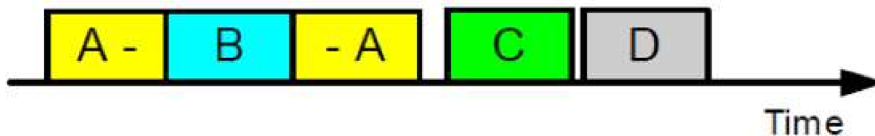
（5）

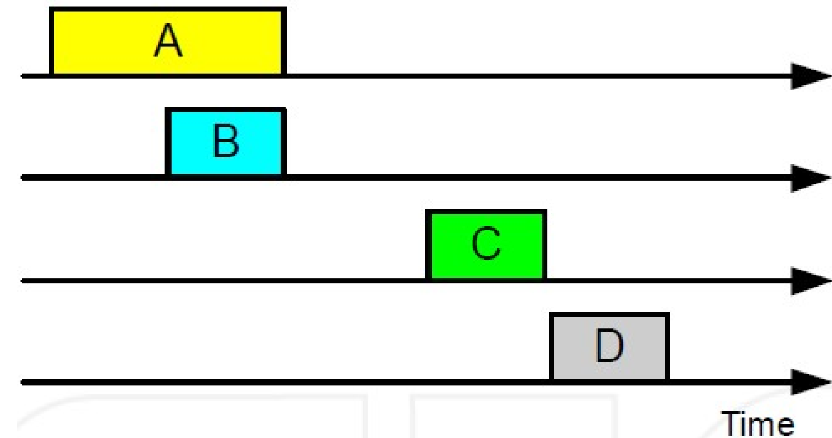
（6）

高优先级任务得到
CPU使用权

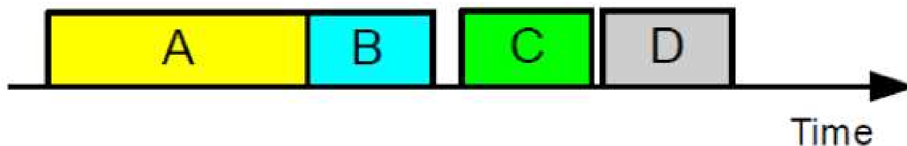
Preemptive or Non-Preemptive

- Preemptive



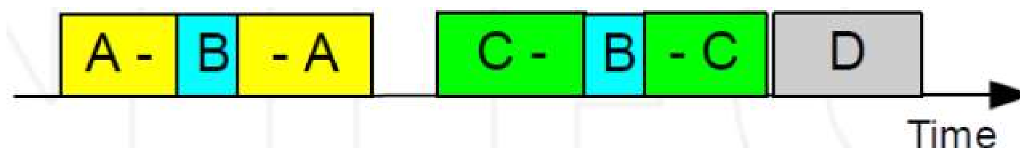


- Non-Preemptive， co-operative



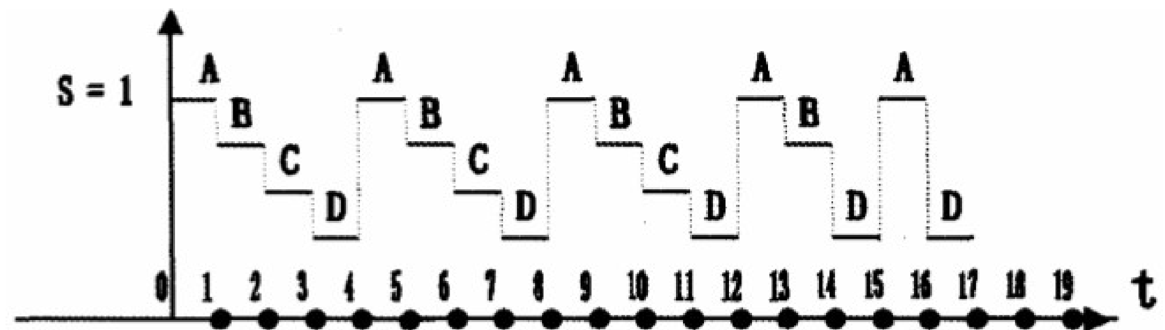
WCETs may increase up to 35% in the presence of preemptions!
In general, NP scheduling reduces schedulability introducing blocking delays in high priority tasks!

- Hybrid scheduling
  – B is pre-emptive, A, C and D run co-operatively



Preemptive or Non-Preemptive

- Non-Preemptive，co-operative
  - State-space needed is smaller：Lower impl cost (mem)
  - No race：保证独占访问任何共享资源或数据！！！
  - Less overhead at run-time
    - Preemptive：Context switching is significant
    - Cache pollution，memory size
  - Certification authorities tend to support this form of scheduling.
    - The scheduler is simpler.
    - Testing is easier.
    - 时间可预测：no race, no jitter
  - Liu： 当任务不可抢占时，没有最优的online调度算法
  - 长任务会影响系统响应时间
- Preemptive
  - 高优先级任务响应快
  - Makespan小
  - 上下文切换开销大



内核的可抢占性

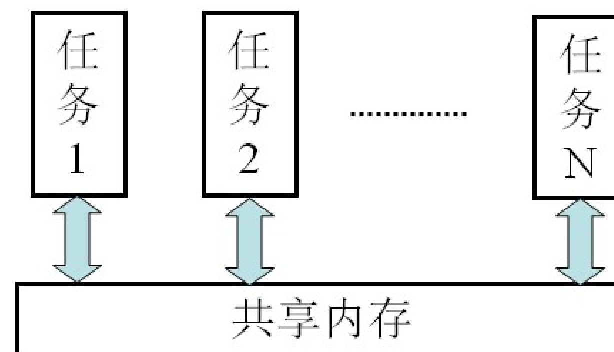
- 可抢占内核（preemptable kernel）
  - 即使正在执行内核服务，也能响应中断，并且中断服务退出时能够进行任务重新调度。
    - 此时，如果有高优先级任务就绪，就立即调度其执行，不要求回到被中断的任务。
- 不可抢占内核
  - 内核服务不能被中断，或
    - 操作系统执行时关闭所有中断！
  - 内核服务允许中断，但不能进行任务重新调度

Concurrency（异步，抢占）问题

- Concurrency Can occur at many levels：conflicts
  - *non-deterministic interleaving* of concurrent activities
    - Events（任务）that occur out of expected or desired order
    - are a major source of defects in real-time software
      - Difficult to identify，Difficult to detect，Difficult to fix
- Edsgar Wybe Dijkstra (1930–2002)
  - 【EWD 1303】[The interrupt] was a great invention, but also a Pandora's Box…essentially, for the sake of efficiency, concurrency [became] visible… and then, all hell broke loose
    - race condition：竞争临界资源（互斥）
      - 互斥—>死锁deadlock
    - timing anomaly（jitter）：优先级反转（资源访问协议）
  - 哲学家就餐问题：死锁
  - 银行家算法（EWD1965）：避免死锁
    - 动态申请，需求审查，及时/限时发放，限时使用

资源与临界资源

- 资源：任务运行时使用的软硬件实体
  - 被任何任务占用的对象：物理设备、数据结构等
- 临界资源：多个任务需同时访问的共享资源
  - 也称“资源的临界区”
- Race Condition
  - 多个任务同时访问共享数据，造成数据的不一致性。
- 临界资源访问原则：禁止抢占
  - 互斥（mutually exclusive）
  - 阻塞（blocked）



llxx@ustc.edu.cn

并发的不确定性：临界资源，临界区

低优先级任务

```
while (1) {
    x = 1;
    y = 2;

    swap (&x, &y);
     {
          Temp = *x;

          *x = *y;
          *y = Temp;
     }
   .
   .
   OSTimeDly(1);
}
```

高优先级任务

```
while (1) {
    z = 3;
    t = 4;

    swap(&z, &t);
     {
          Temp = *z;
          *z   = *t;
          *t   = Temp;
     }
    .
    .
    OSTimeDly();
    .
    .
}
```

Temp == 1

OSIntExit()

(1) ISR (2) O.S. (3)

(5) O.S. (4)

Temp == 3!

Temp == 3

Critical region/ Critical Section ， Race Condition，代码的可重入问题，解决方案？

代码的临界区

- 临界区：访问临界资源的代码
  - 延伸：处理时不可分割的代码
    - 一旦这部分代码开始执行，则不允许被中断。
- OS系统中会有哪些临界区？
  - 如：任务上下文切换时，进行地址、指令、数据等寄存器堆栈保护操作的代码。

- “可重入函数”是临界区特例？
- 线程安全(Thread safe)问题？

可重入型函数(reentrant)

- 可以供多个任务并发使用而不必担心数据错误。
  - 任何时候都可以被中断，重新运行后相应数据不会被破坏
  - 只能使用局部变量，即变量保存在CPU寄存器中或堆栈中
- 代码共享：用户代码，OS服务

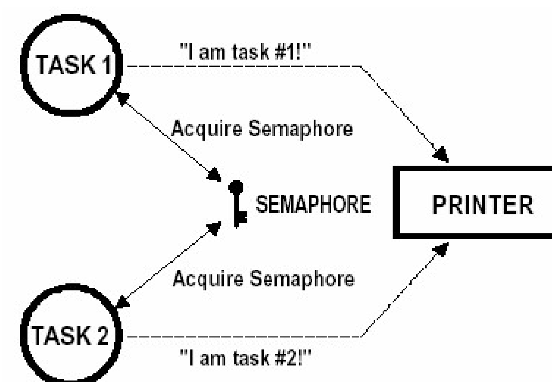
```
int Temp;
Void swap (int *x,int *y)
{
    Temp=*x;
    *x=*y;
    *y=Temp;
}
```

```
Void swap (int *x,int *y)
{
    int Temp;

    Temp=*x;
    *x=*y;
    *y=Temp;
}
```

对共享/临界资源的互斥管理

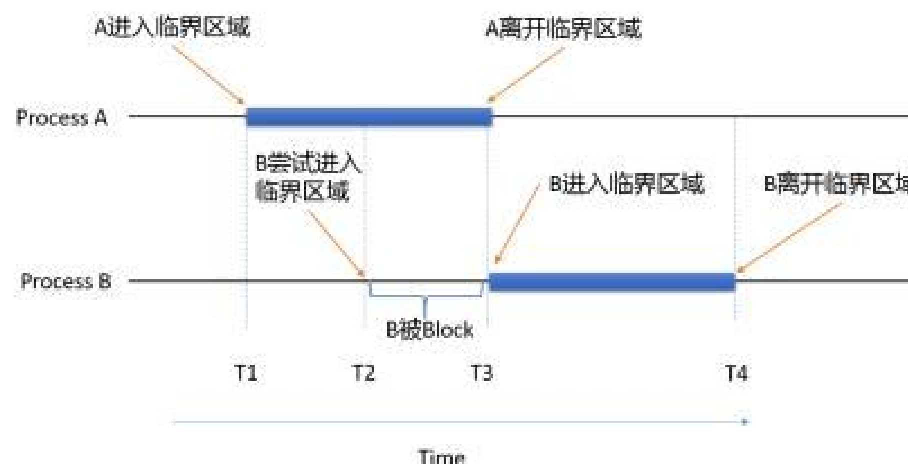
- 关中断：在进入临界区之前关中断
  - 对CPU上锁，互斥粒度最强
  - 可能降低实时性
- 使用“测试并置位”指令：体系结构层
  - 利用某个全局变量判断资源互斥条件
- 禁止任务抢占
  - 禁止正在执行临界区代码的任务切换
  - 对任务调度器上锁，但不禁止中断
  - 可能降低实时性
- 使用信号量（锁）：OS层
  - 提供互斥的最优选择
  - 对共享资源上锁，比关中断和禁止任务切换精确



互斥机制比较

	关中断	测试并置位	禁止抢占	信号量
锁定范围	所有可屏蔽中断事件	所有使用该指令的代码	所有任务（对任务调度器上锁）	竞争共享资源的任务
响应时间影响	如果时间长，影响较大	较小	如果时间长，影响较大	可能导致优先级反转
系统开销	小	小	小	较大
注意事项	时间尽量短	可能影响可移植性	时间尽量短	避免过度使用

- Blocking代价
- 方法易用性
  - 实施成本
- 影响：全局 vs. 局部

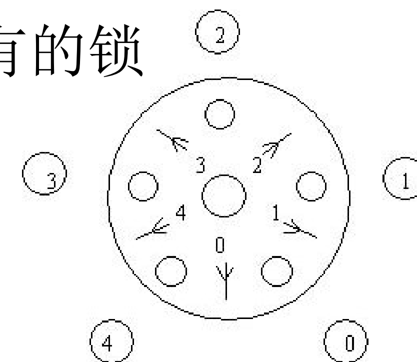


锁的类型

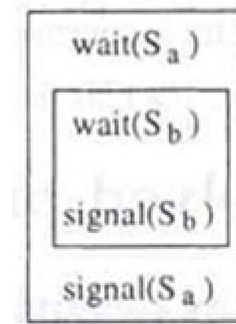
- 共享锁（读锁、S锁）
  - 事务T对数据A加上共享锁后，其他事务只能对A再加共享锁，不能加互斥锁，直到已释放所有共享锁。
  - 获得共享锁的事务只能读数据，不能修改数据。
- 互斥锁（写锁、X锁、排它锁、独占锁）
  - 事务T对数据A加上排他锁后，其他事务不能再对A加任何类型的锁，直到在事务的末尾将锁释放为止。
  - 获得排他锁的事务既能读数据，又能修改数据。
- 自旋锁spinlock
  - 与互斥锁类似，效率比互斥锁高
    - 调用者不会睡眠，而是一直循环等待（“自旋”）该锁被释放
  - 适用于锁使用者保持锁时间比较短的情况
    - 内核可抢占式或SMP
    - 等待者不被切换？

死锁Deadlock

- 任务执行需要两个以上资源
- 定义：多个任务互相等待对方控制的资源
  - 错序死锁（lock—ordering deadlock）：死锁的基本定义
  - 自死锁（self—deadlock）：申请自己已经占有的锁
  - 递归死锁（recursive—deadlock）



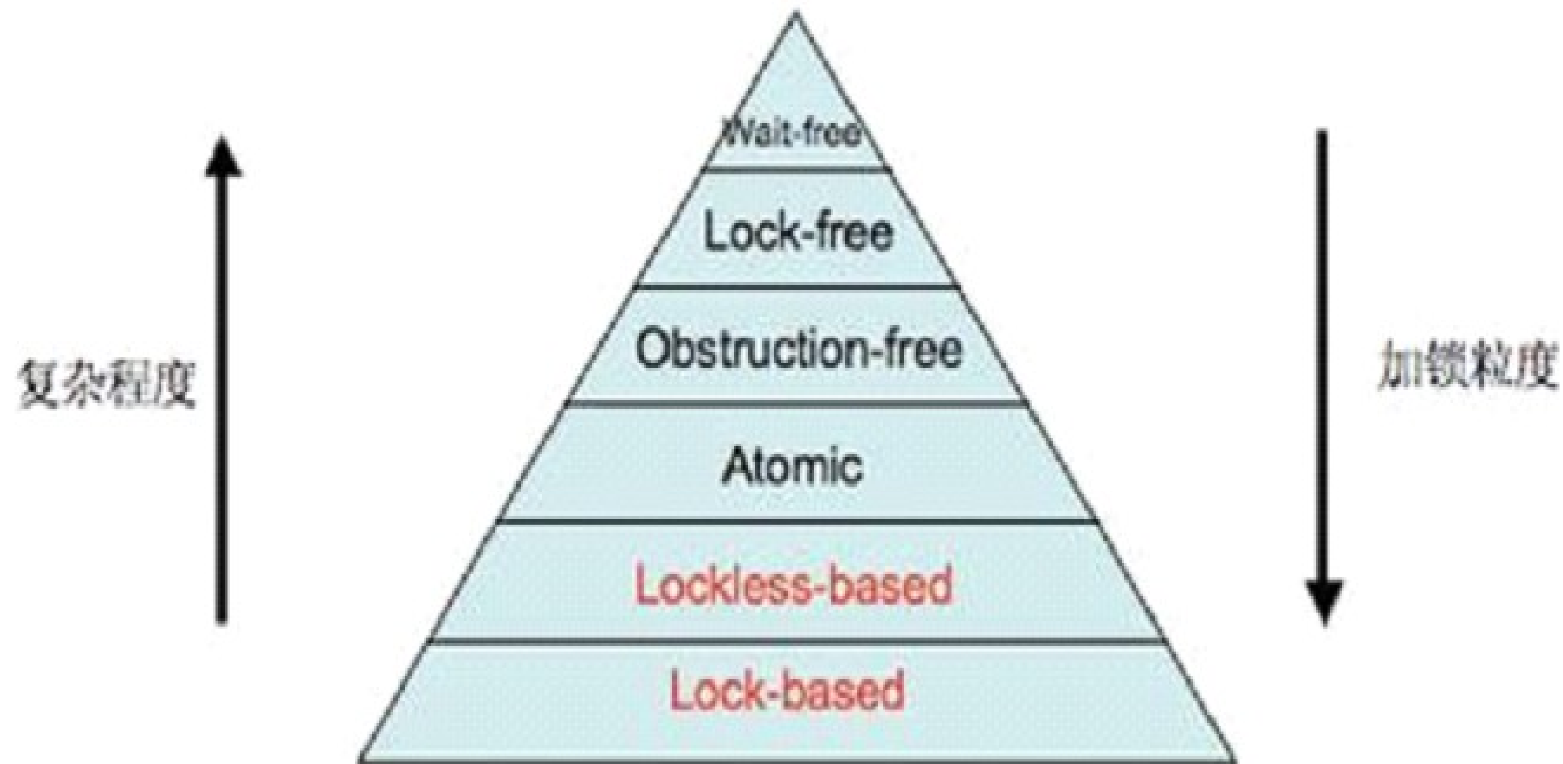
- 防止发生死锁的方法
  - 保证访问顺序：“按序访问”
    - 所有任务按相同顺序申请多个资源，按相反的顺序释放
  - 使用“尝试加锁”操作：“先占后用”
    - 不成功则放弃自己的锁。随机重试。
    - 活锁（live lock）问题：任务间不断冲突，不断退避。
      - 指数退避技术：重试间隔时间为上一次的两倍
  - 内核大多允许申请信号量时定义等待超时。



共享数据的互斥访问编程模型

- 阻塞型（Blocking Synchronization）
  - 当一个线程到达临界区时被阻塞，等待其他线程释放锁
    - 死锁（deadlock），活锁（livelock）
    - 护航问题（convoying）
      - 持有锁的线程被挂起造成其他申请者也无法执行
    - 优先级反转（priority inversion）
- 非阻塞型（Non-blocking Synchronization）
  - 无锁编程：为每个任务复制一个共享资源副本
  - 非阻塞的三个层次
    - Wait-free：任意线程的任何操作都可以在有限步之内结束
    - Lock-Free：能够确保执行它的所有线程中至少有一个能够继续往下执行
    - Obstruction-free：在任何时间点，一个孤立运行线程的每一个操作可以在有限步之内结束
  - Linux内核实现了Lock-Free

互斥访问编程模型



- 红色：Blocking synchronization
  - Lock-based 和 Lockless-based 的区别：锁粒度不同
  - Lock-based如：mutex，semaphore
- 黑色：Non-blocking synchronization

任务优先级反转/阻塞（谁与谁反转?）

Priority Inversion

Task 1 (H)

(4)

(12)

Task 2 (M)

(8)

Task 3 (L)

(1)

(6)

(10)

Task 3 Get Semaphore
(2)

Task 1 Preempts Task 3
(3)

Task 1 Tries to get Semaphore
(5)

Task 2 Preempts Task 3
(7)

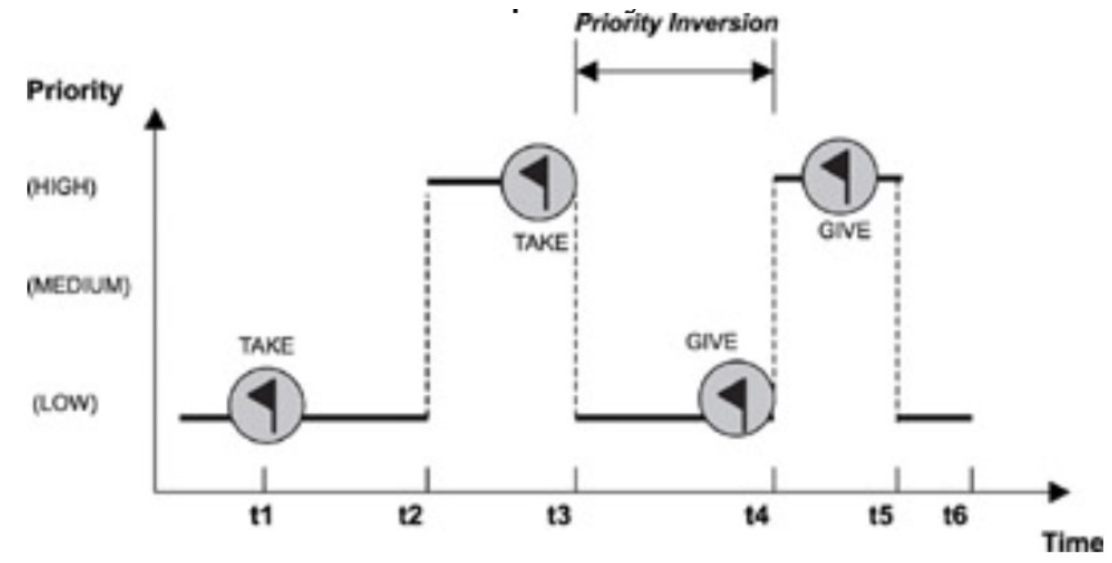
Task 3 Resumes
(9)

Task 3 Releases the Semaphore
(11)

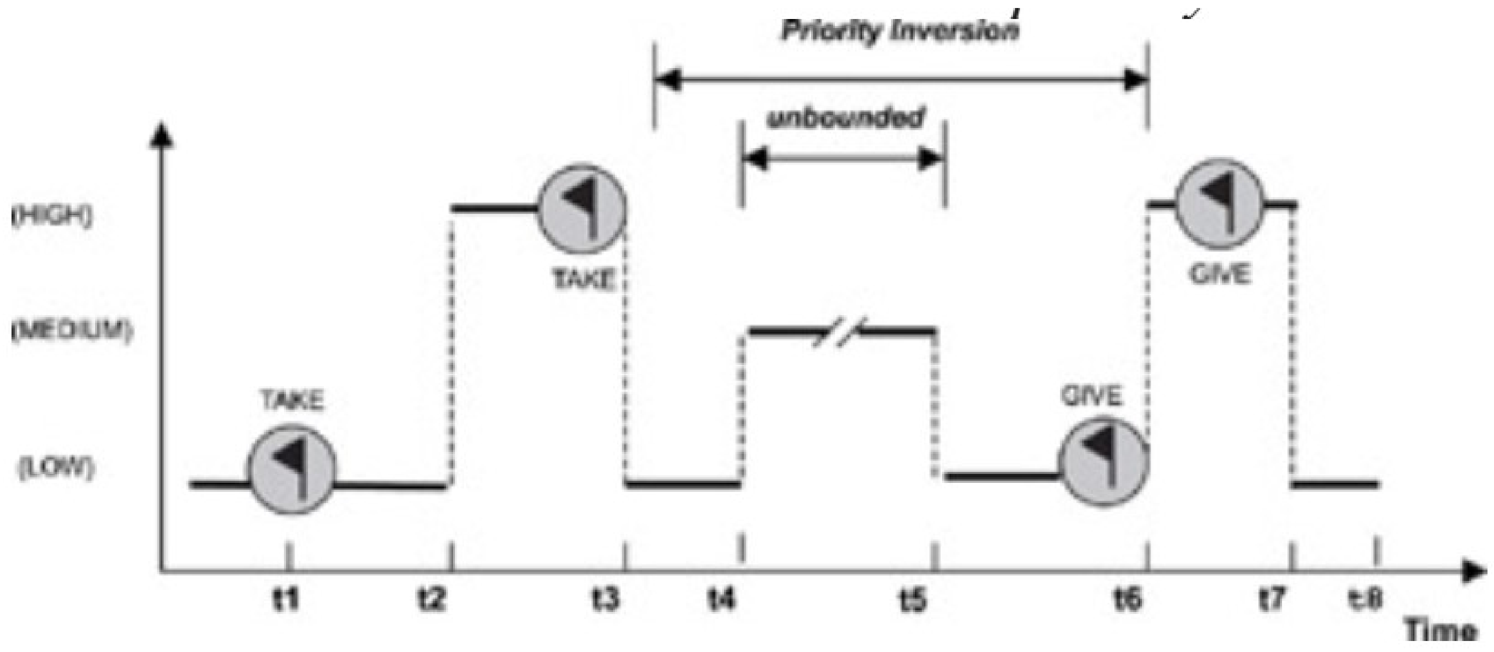
priority (H) > priority (M) > priority (L)

Blocking time for H：Timing anonaly

bounded

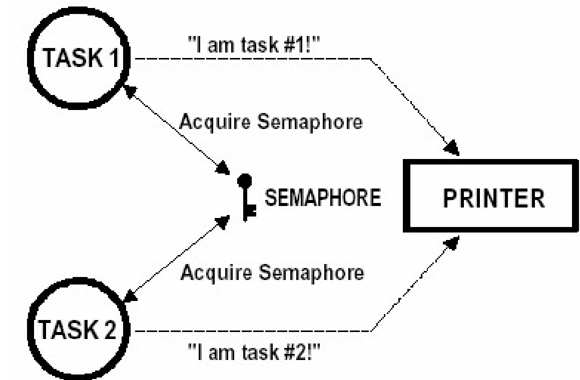


unbounded
critical region
Blocking chain
Timing anomaly



Resource access control protocols

- Resource sharing
  - 反转：禁止抢占（阻塞时间）
    - cannot be avoided，can be nininized
    - Priority inversion is modeled by blocking time
  - 死锁：共享多个资源
- 协议：尽快执行临界区，限制反转，防止死锁
  - Under static priorities
    - Non-Preemptive Protocol (NPP)
    - Priority Inheritance Protocol(PIP)
    - Ceiling Priority Protocol(CPP)
    - Priority Ceiling Protocol(PCP)
  - Under dynamic priorities
    - Stack Resource Policy (SRP)
      - 针对“多单元资源multi-unit resources”，extends PCP



Nonpreemption Protocol

Nonpreemptible critical sections

τ_2:{…P(S1)…V(S1)…}

τ_4:{…P(S1)…V(S1)…}

ready

blocked

τ_1(H)

ready

τ_2

ready

τ_3

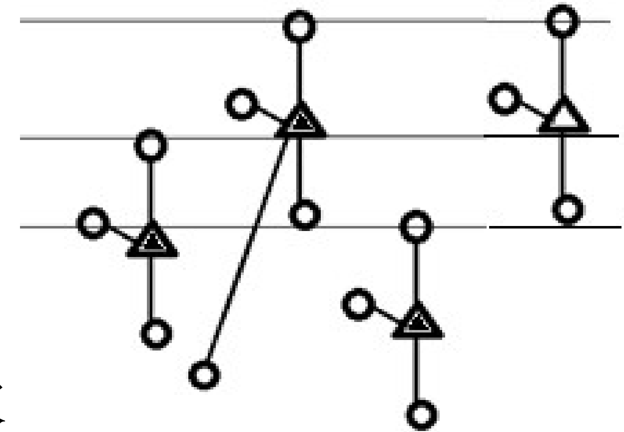
S1 locked

S1 unlocked

τ_4(L)

Time

相关概念Some Notations



- 任务的分配优先级
  - 可静（RM），可动（EDF/LLF）
- 任务的当前优先级：执行优先级
  - 动态
- 优先级继承：任务临时提升优先级
- 资源的优先级天花板priority ceiling
  - 所有使用该资源的任务的（基本）优先级最高者
  - 静态，预设
- 系统的当前优先级天花板current priority ceiling
  - 某时刻，所有被使用资源的天花板最高者。
  - 无资源分配时，CPC比所有任务的优先级都低（=Ω）
    - 如系统初始，所有资源空闲
  - 动态

优先级继承(PIP)协议：Sha et al., 1990

- 将占有者的优先级抬升至请求者的优先级
  - 需检测：占有者的优先级是否低于请求者的优先级
  - 阻塞链：如果有多个高优先级任务，则需要多次抬升。Timing anonaly
  - 优先级传递：A阻塞B，B阻塞C，则A继承C。
- 存在死锁问题。（场景示例？）

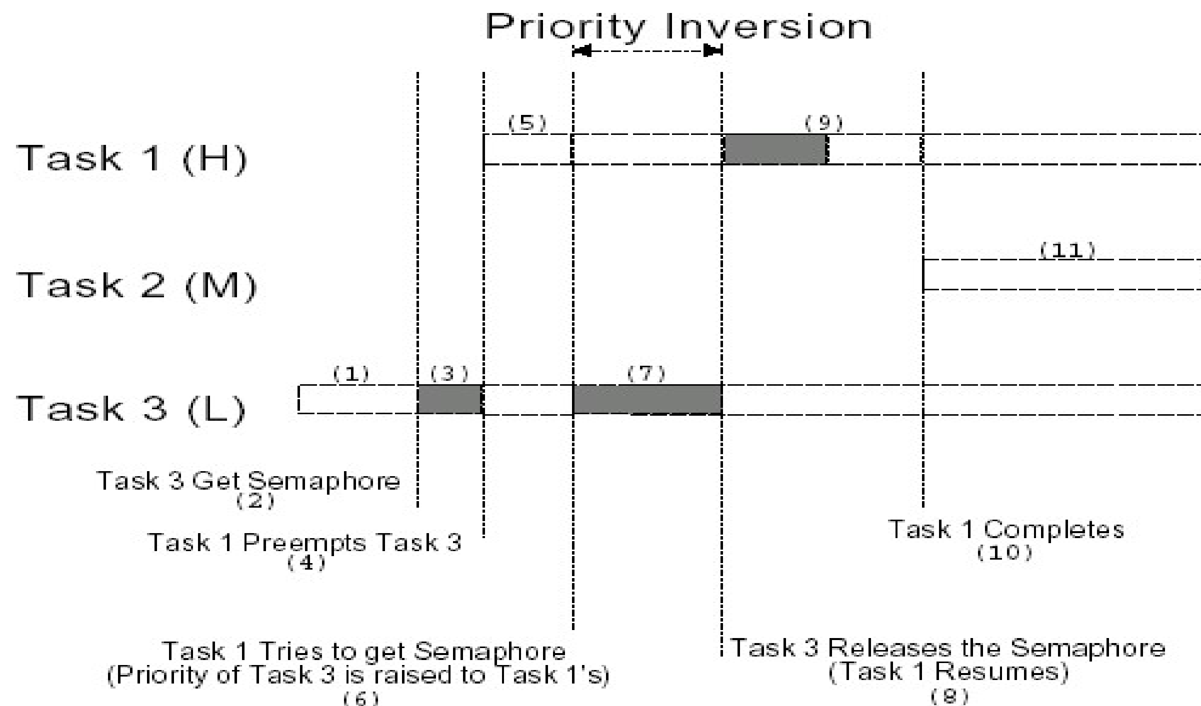
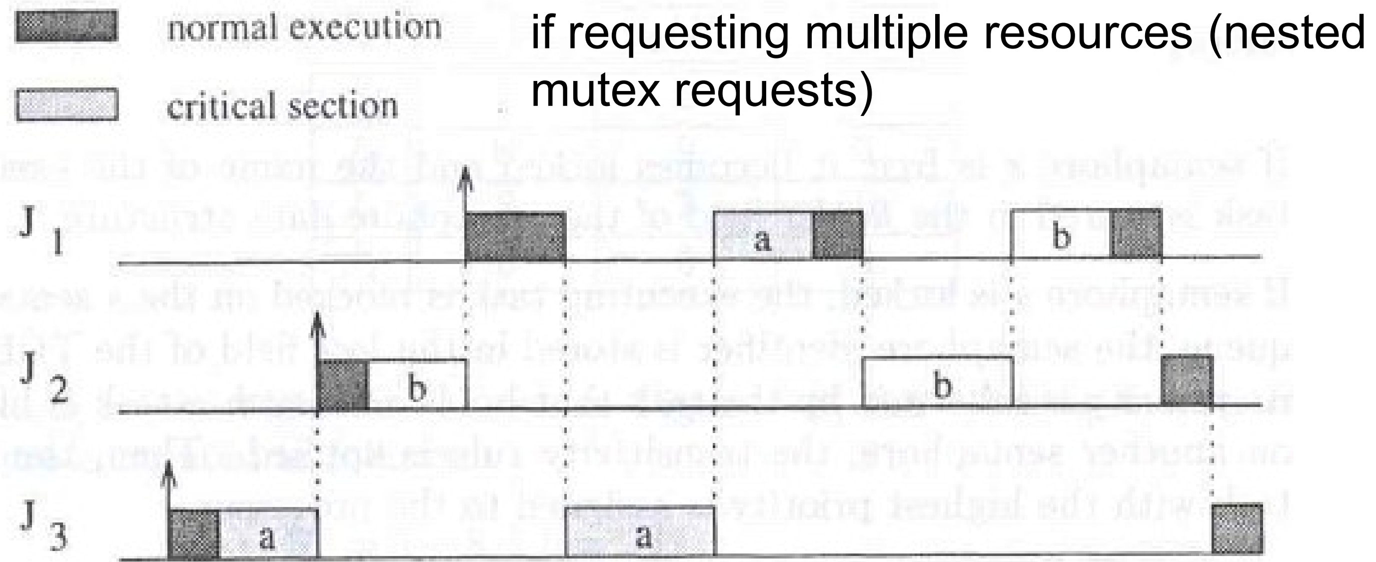


Figure 2-8, Kernel that supports priority inheritance.

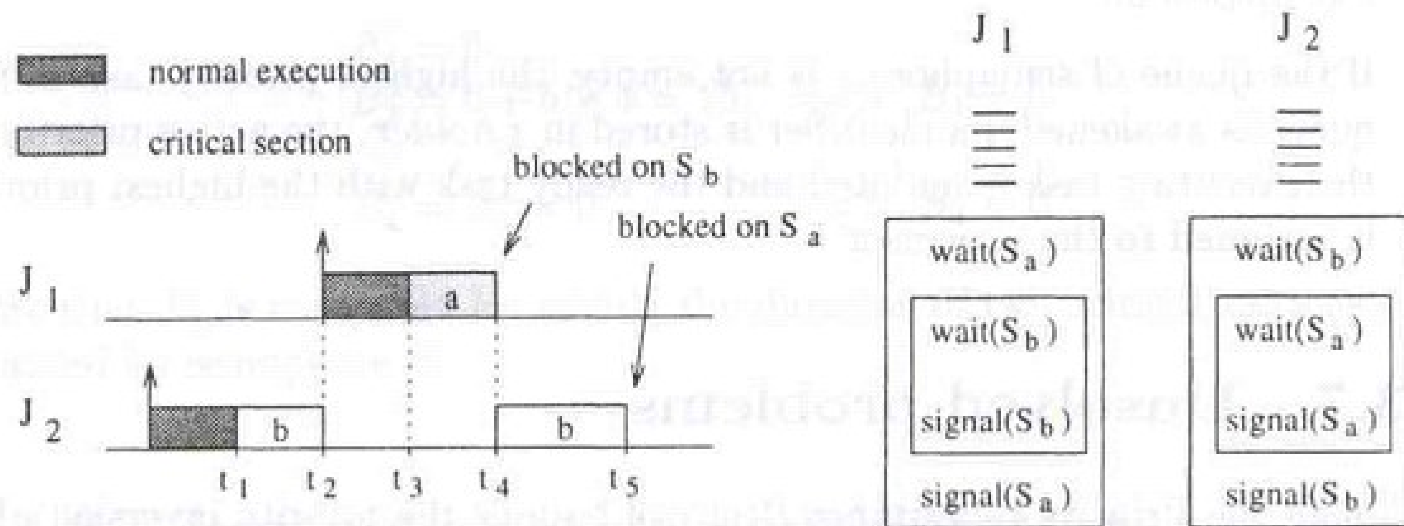
PIP：Chained Blocking

if requesting multiple resources (nested mutex requests)

优先级嵌套

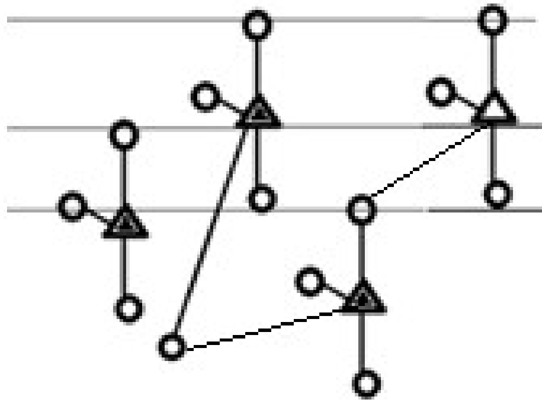


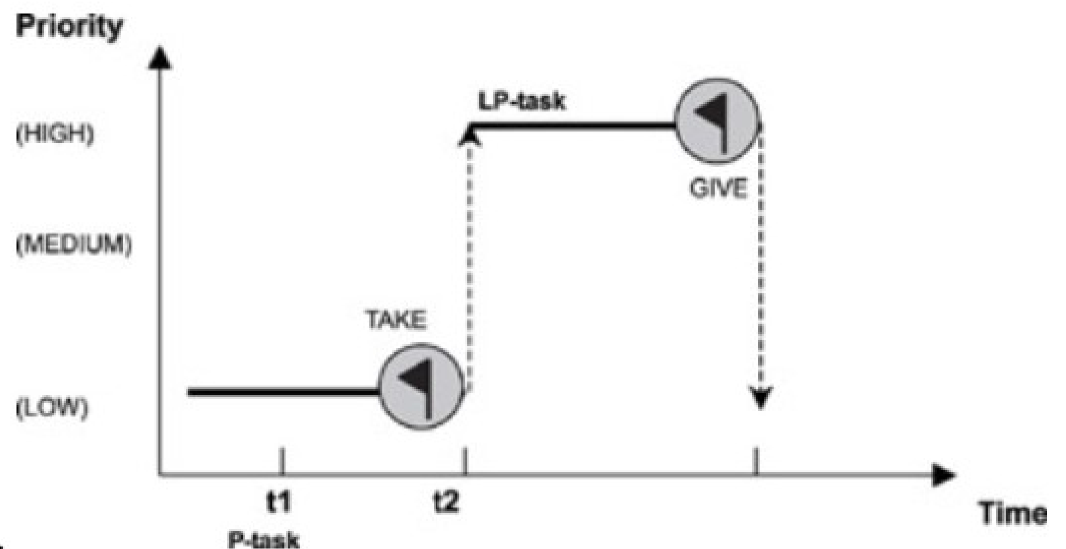
死锁
nested locks



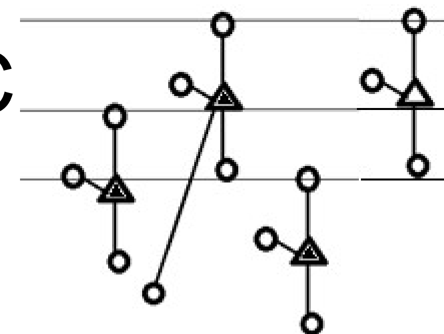
Ceiling Priority Protocol(CPP)

- Immediate Priority Ceiling (IPC)/Highest Locker Priority (HLP)
  - 一旦任务进入临界区，则继承资源的优先级天花板
    - 避免按任务逐级抬升
    - 无论是否有冲突都抬升，可能浪费
  - 避免了低优先级任务阻塞当前任务
  - 缓解PIP阻塞链：至多只有一个低优先级任务可以阻塞高优先级任务
- 无防止死锁？
- 实现简单：不检测，但需先验知识





优先级天花板（PCP）协议



- 基于系统的当前优先级天花板CPC
- 任务T申请资源R时的规则
  - R空闲时：——不一定分配！
    - 规则1：如果T的优先级高于CPC，则T得到R
      - 防止死锁："保证资源按序访问"！
    - 规则2：如果T的优先级不高于CPC，则仅当T拥有天花板等于CPC的资源时，才能分配
      - 防止自己阻塞自己
- 阻塞T的任务继承T的优先级执行
  - 直至他们释放所有天花板高于等于T的资源

PCP协议：Sha/Rajkumar/Lehoczky，1990

priority (H) > priority (M) > priority (L)

H and L share resource R

normal execution

critical region

L receives Rís ceiling priority (= Hís priority)

H blocked

L receives original priority

H

M

L

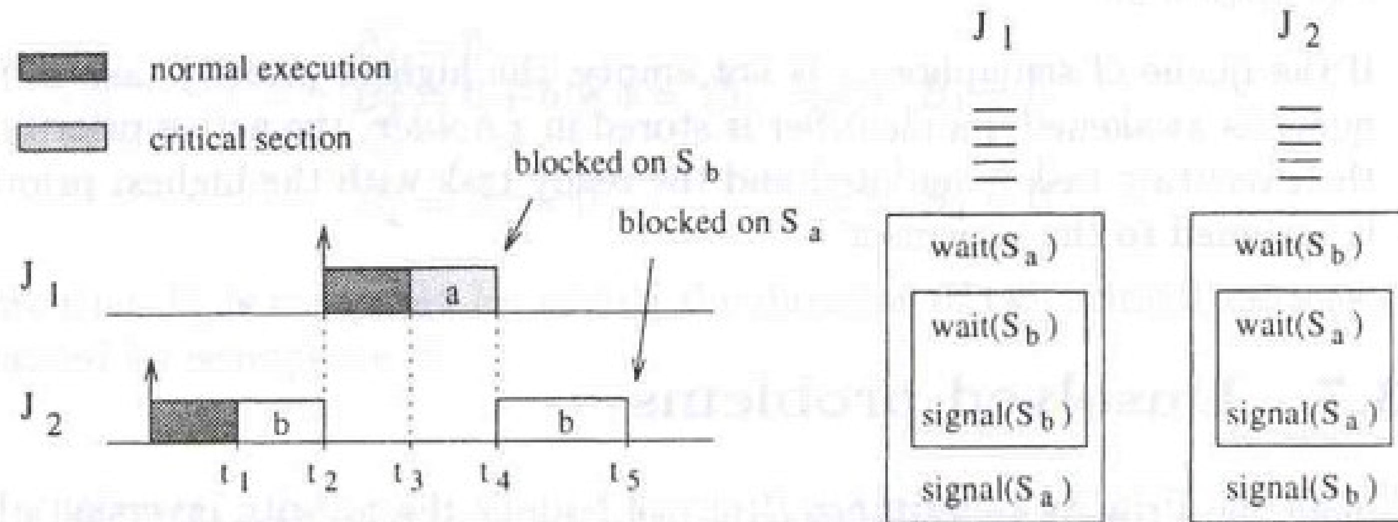
t

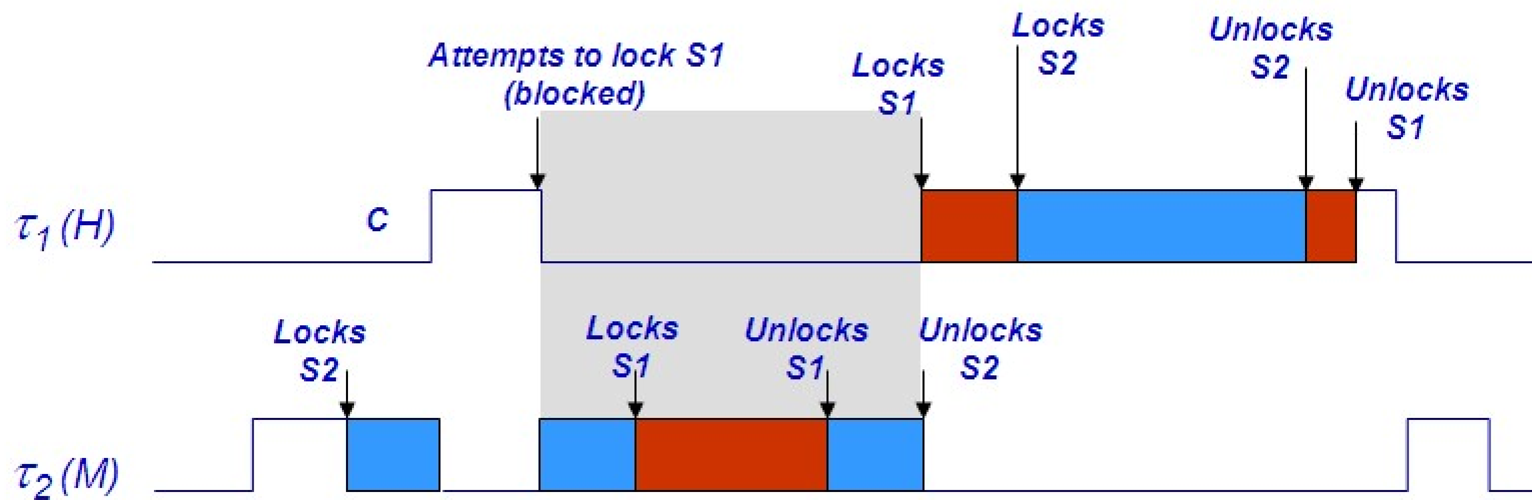
t

t

- “高于当前优先级天花板的任务不会申请已被占用的资源，正在占用资源的任务也不会申请高于当前优先级天花板的任务所需要的资源”

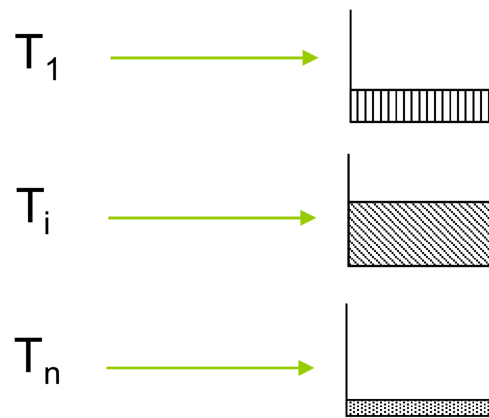
Deadlock Avoidance: Using PCP



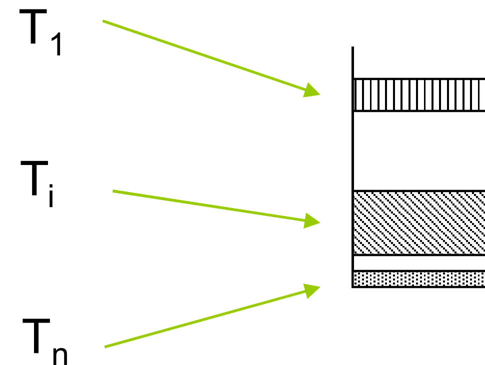


Stack Sharing

- Sharing of the stack among tasks eliminates stack space fragmentation and so allows for memory savings:



no stack sharing



stack sharing

- However:
  - Once job is preempted, it can only resume when it returns to be on top of stack.
  - Otherwise, it may cause a deadlock.
  - Stack becomes a resource that allows for “one-way preemption”.

栈资源策略(Stack Resource Policy)

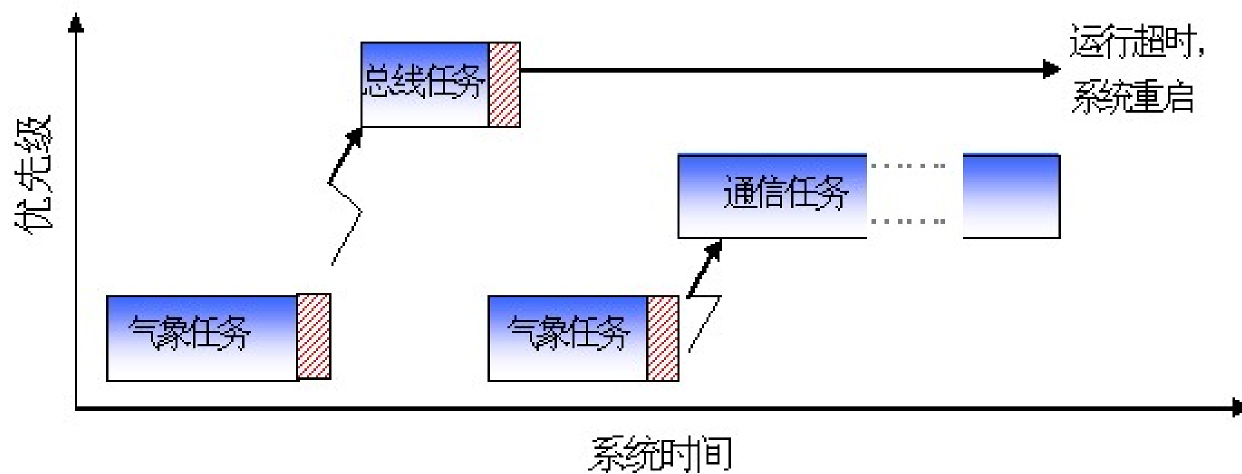
- Stack-Based PCP
  - 允许对所有的进程使用一个单独的栈
  - 一旦一个作业开始执行，它可以被抢占，但不会被阻塞。
  - allow preemption only if the priority is higher than the ceiling of the resources in use
  - Whenever a job is preempted, all the resources needed by the preempting job are free.
- 栈资源策略是优先级天花板协议的改进
  - 既适用于静态优先级调度算法又适用于动态优先级调度算法。
    - 优先级天花板协议只适用于静态优先级算法。
  - 减少了上下文转换的最大数量。
  - 如果使用EDF调度，唯一可以使用的策略就是栈资源策略。

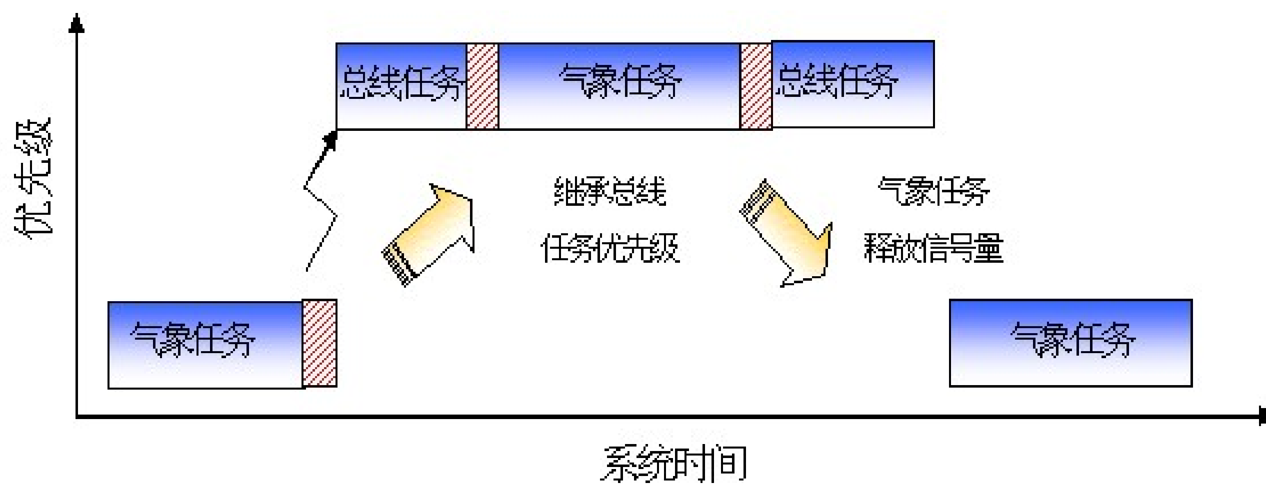
Mars Rover Pathfinder

- The Mars Rover Pathfinder landed on Mars on July 4th, 1997.
- A few days into the mission, the Pathfinder began sporadically missing deadlines, causing total system resets, each with loss of data.
- The problem was diagnosed on the ground as priority inversion, where a low priority meteorological task was holding a lock blocking a high-priority task while medium priority tasks executed.
  - 看门狗监测到“总线”没有活动，reset！
- VxWorks中采用了优先级继承协议，默认情况下该功能被关闭

- Source: RISKS-19.49 on the comp.programming.threads newsgroup, December 07, 1997, by Mike Jones (mbj@MICROSOFT.com).

meteorological task





并发：Pandora‘s Box

- 临界资源：数据一致性问题，互斥访问
  - 抢占：锁，优先级反转（阻塞，执行时间异常miss deadline），死锁
    - 失控：优先级反转的持续时间无限长
  - Disallow preemption：无须锁
- 资源访问控制协议：临界区尽快执行，使阻塞时间有界，并防止死锁
  - PIP “按需提升”：不需要预知任务请求资源的信息。提升占有者！
    - 存在阻塞链（无界？），没有避免死锁
    - 贪心：一旦资源空闲就抢占
  - PCP“按序访问”：需要预知所有资源请求。提升申请者！
    - 最多一个阻塞，避免死锁（）
    - 非贪心：不追求资源利用率最优，即使资源空闲，也可能拒绝任务的资源访问请求
  - 与调度算法一起使用
- PIP和PCP存在的问题：为啥需要两个协议？
  - 都只能缓解阻塞时间（最小化），无法消除，中间优先级任务可能被反转
  - 系统需要支持同优先级和动态优先级
  - 单处理器？

Summary of Sync Protocols

Protocol	Bounded Priority Inversion	Blocked at Most Once	Deadlock Avoidance
NPP	Yes	Yes[1]	Yes[1]
PIP	Yes	No	No
CPP	Yes	Yes[1]	Yes[1]
PCP	Yes	Yes[2]	Yes[1]

[1] Only if tasks do not suspend within critical sections

[2] PCP is not affected if tasks suspend within critical sections.

- PIP
  - no penalty when the locks are not contended（有race时大！）, which covers the vast majority of time-constrained
  - many extra context switches are avoided, excellent average performance
- PCP
  - handles nested locks well, and can avoid deadlock in some cases
  - static analysis of the system to find the priority ceiling

Real-time executive components

调度策略？

Scheduling information

Real-time clock

Scheduler

Interrupt handler

Process resource requirements

等待/挂起队列

Processes awaiting resources

Resource manager

Available resource list

Ready processes

Released resources

Ready list

Despatcher

Processor list

就绪队列

上下文切换，改变执行流

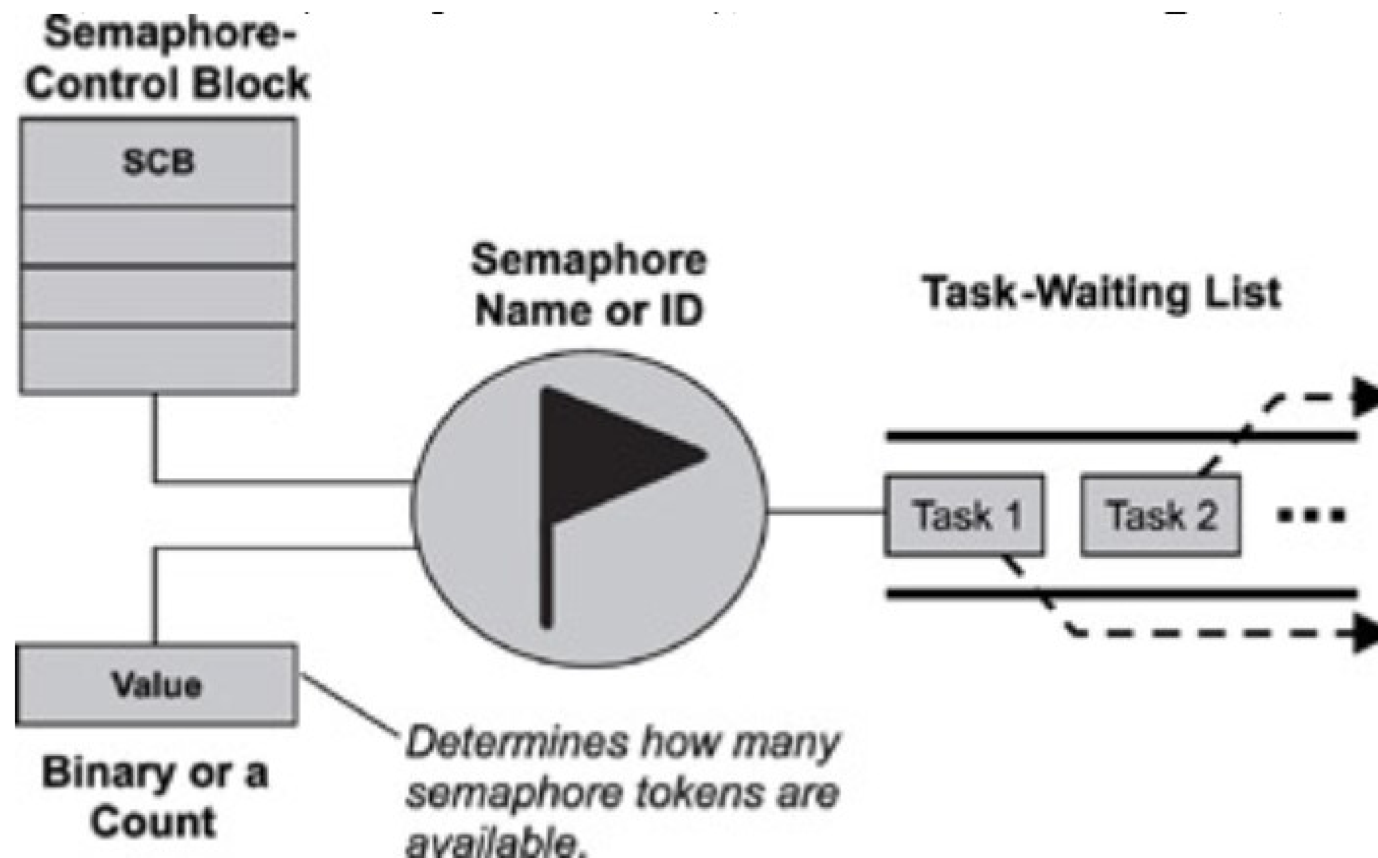
Executing process

并发（同步、互斥）、通信

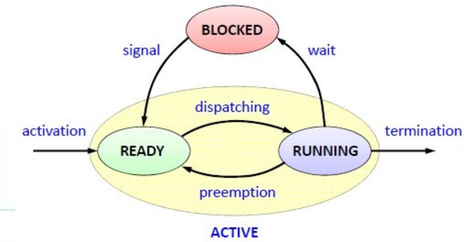
- 互斥：资源共享
  - Shared-Resource-Access Sync
- 同步：同步点相互等待（低级通信），无数据交换
  - Wait-and-Signal Sync
  - Rendezvous sync（*simple rendezvous*）
  - Credit-tracking synchronization：rate sync（见QingLi）
  - barrier sync：三个以上任务
- 通信：高级通信（生产者-消费者）
- Semaphores
  - P、V操作原语：wait()和signal()
    - P = -1，wait，申请；V = + 1，signal，释放
  - 二值信号量（0，1）
    - 互斥mutex/lock：赋初值1；获取时减一，表示已占用
    - 同步semaphore ：赋初值0；获取时加一，表示发出了同步请求
  - 计数信号量：资源计数，rate sync

Semaphores实现

- binary sem， counting sem，Mutex
  - Mutual Exclusion Sem

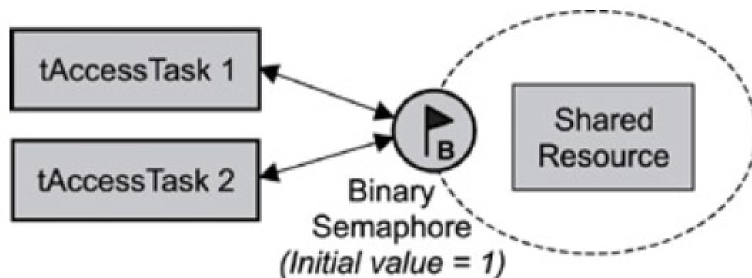


互斥与同步编程模型

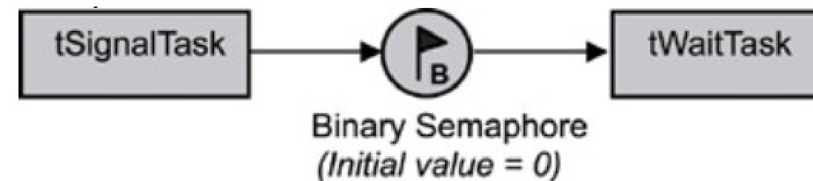



```
1   semaphore S=1;   // 初始化信号量
2
3   T1() {
4       ...
5       P(S);
6       线程T1的临界区;  // 访问临界资源
7       V(S);
8       ...
9   }
10
11  T2() {
12      ...
13      P(S);
14      线程T2的临界区;  // 访问临界资源
15      V(S);
16      ...
17  }
```

```
1   semaphore S=0;   // 初始化信号量
2
3   T1() {
4       ...
5       x;               // 语句x
6       V(S);            // 告诉线程T2，语句x已经完成
7       ...
8   }
9
10  T2() {
11      ...
12      P(S);            // 检查语句x是否运行完成
13      y;               // 检查无误，运行y语句
14      ...
15  }
```

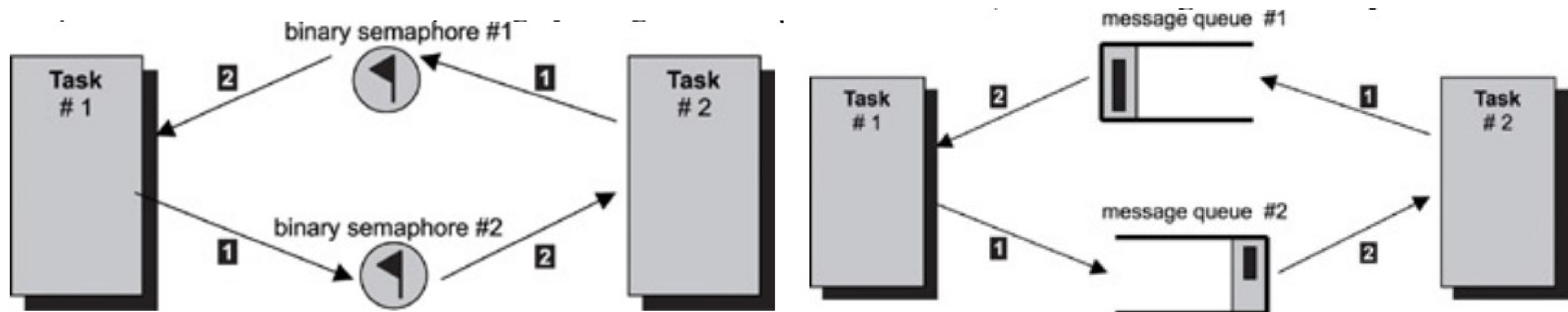


Wait-and-Signal Sync



Rendezvous sync

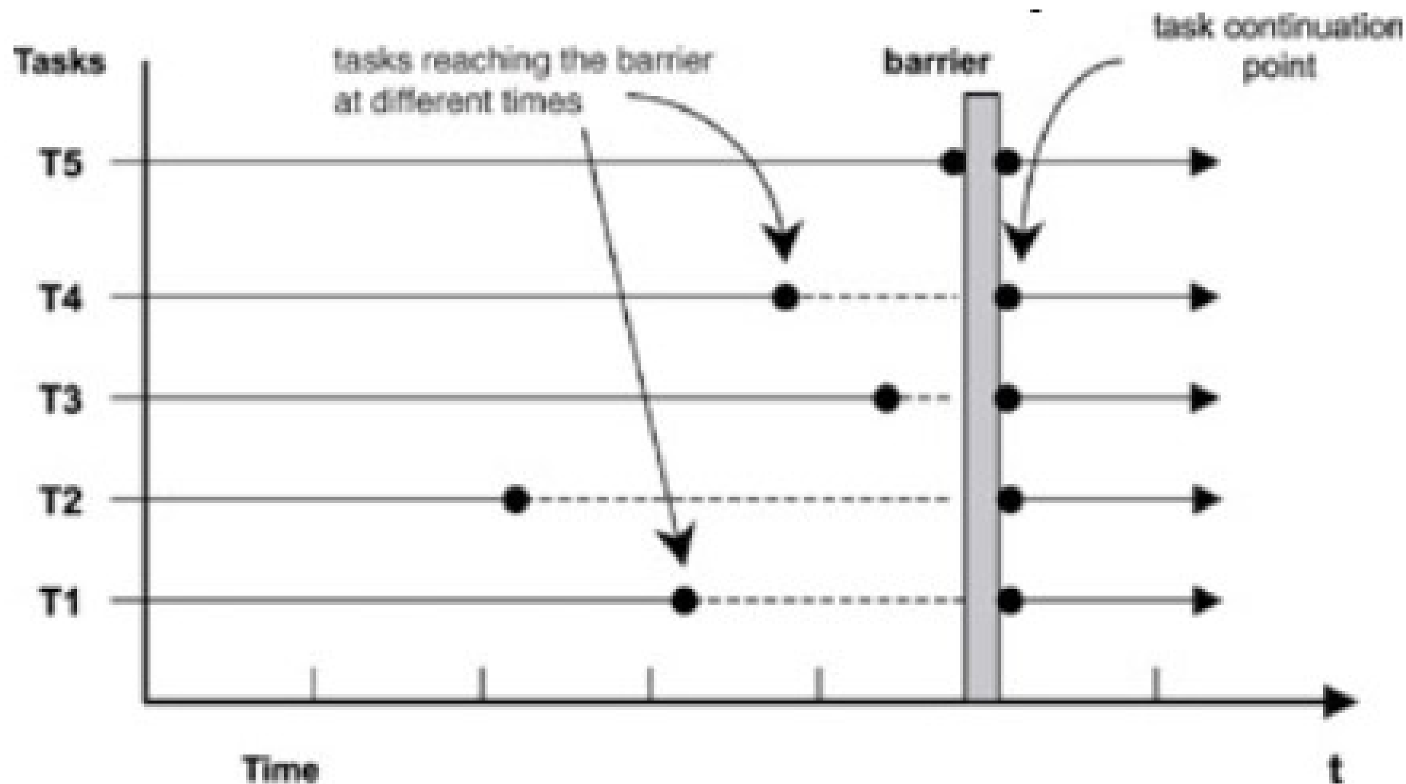
- with/without data passing
- uses kernel primitives
  - such as semaphores or message queues
    - MQ(见后)：a maximum of one message



并发任务的barrier sync模式

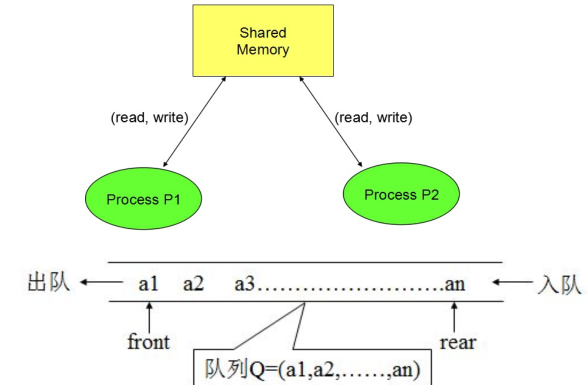
- 针对两个以上任务
- 可用信号量和条件变量实现
- 不是所有OS都提供

- 三个动作
  - 任务声明到达同步点
  - 早到任务等待后续任务
  - 接收*继续前进*的通知



Comm models: Shared Memory

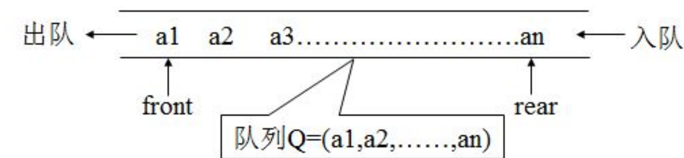
- Shared address space(variables)
  - Communication primitives
    - Data transfer via load, store, atomic swap
  - No time overhead, easy to implement
  - Mistakes are common： synchronize
- Example: Producer/consumer with a mistake
  - Share *buffer*[*N*], *count*
    - *count* = # of valid data items in *buffer*
  - *processA* produces data items and stores in *buffer*
    - If *buffer* is full, must wait
  - *processB* consumes data items from *buffer*
    - If *buffer* is empty, must wait
  - Error when both processes try to update *count* concurrently (lines 10 and 19) and the following execution sequence occurs. Say “count” is 3.
    - *A* loads *count* (*count* = 3) from memory into register R1 (R1 = 3)
    - *A* increments R1 (R1 = 4)
    - *B* loads *count* (*count* = 3) from memory into register R2 (R2 = 3)
    - *B* decrements R2 (R2 = 2)
    - *A* stores R1 back to *count* in memory (*count* = 4)
    - *B* stores R2 back to *count* in memory (*count* = 2)
  - *count* now has incorrect value of 2




```
01: data_type buffer[N];
02: int count = 0;
03: void processA() {
04:   int i;
05:   while( 1 ) {
06:     produce(&data);
07:     while( count == N );/*loop*/
08:     buffer[i] = data;
09:     i = (i + 1) % N;
10:     count = count + 1;
11:   }
12: }
13: void processB() {
14:   int i;
15:   while( 1 ) {
16:     while( count == 0 );/*loop*/
17:     data = buffer[i];
18:     i = (i + 1) % N;
19:     count = count - 1;
20:     consume(&data);
21:   }
22: }
23: void main() {
24:   create_process(processA);
25:   create_process(processB);
26: }
```

Correct Solution of Consumer-Producer Problem

- The primitive *mutex* is used to ensure critical sections are executed in mutual exclusion of each other
- Following the same execution sequence as before:
  - *A*/*B* execute *lock* operation on *count_mutex*
  - Either *A* **or** *B* will acquire *lock*
    - Say *B* acquires it，*A* will be put in blocked state
  - *B* loads *count* (*count* = 3) from memory into register R2 (R2 = 3)
  - *B* decrements R2 (R2 = 2)
  - *B* stores R2 back to *count* in memory (*count* = 2)
  - *B* executes *unlock* operation
    - *A* is placed in runnable state again
  - *A* loads *count* (*count* = 2) from memory into register R1 (R1 = 2)
  - *A* increments R1 (R1 = 3)
  - *A* stores R1 back to *count* in memory (*count* = 3)
- *Count* now has correct value of 3



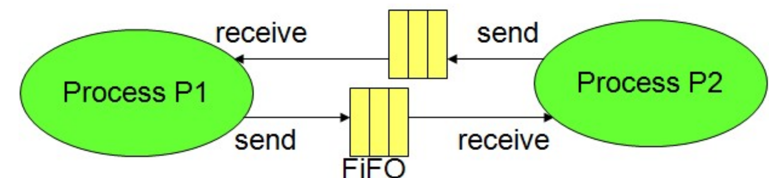

```
01: data_type buffer[N];
02: int count = 0;
03: mutex count_mutex;
04: void processA() {
05:   int i;
06:   while( 1 ) {
07:     produce(&data);
08:     while( count == N );/*loop*/
09:     buffer[i] = data;
10:     i = (i + 1) % N;
11:     count_mutex.lock();
12:     count = count + 1;
13:     count_mutex.unlock();
14:   }
15: }
16: void processB() {
17:   int i;
18:   while( 1 ) {
19:     while( count == 0 );/*loop*/
20:     data = buffer[i];
21:     i = (i + 1) % N;
22:     count_mutex.lock();
23:     count = count - 1;
24:     count_mutex.unlock();
25:     consume(&data);
26:   }
27: }
28: void main() {
29:   create_process(processA);
30:   create_process(processB);
31: }
```

Comm models: Message Passing

- Communication primitives
  - e.g., send, receive library calls
  - Data explicitly sent from one process to another
    - Sending process performs special operation, *send*
    - Receiving process must perform special operation, *receive*, to receive the data
    - Both operations must explicitly specify which process it is sending to or receiving from
- Safer model, but less flexible
  - Why safer？

```
void processA() {
  while( 1 ) {
    produce(&data)
    send(B, &data);
    /* region 1 */
    receive(B, &data);
    consume(&data);
  }
}
```

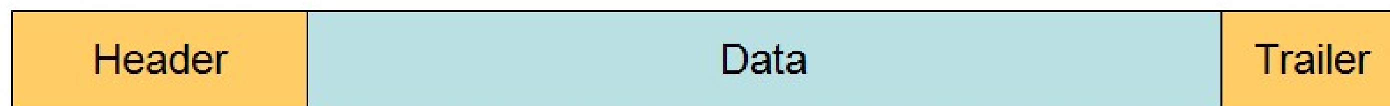
```
void processB() {
  while( 1 ) {
    receive(A, &data);
    transform(&data)
    send(A, &data);
    /* region 2 */
  }
}
```

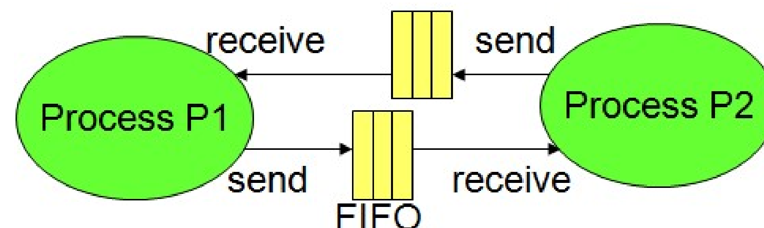


Message Passing：blocking？

- Receive is blocking（？）
- Send may or may not be blocking
  - Non-blocking/Asynchronous
    - Sender does not have to wait until message has arrived;
    - Potential problem: buffer overflow
  - Blocking/Synchronous/Rendezvous
    - Sender will wait until receiver has received message
    - No buffer overflow, but reduced performance.

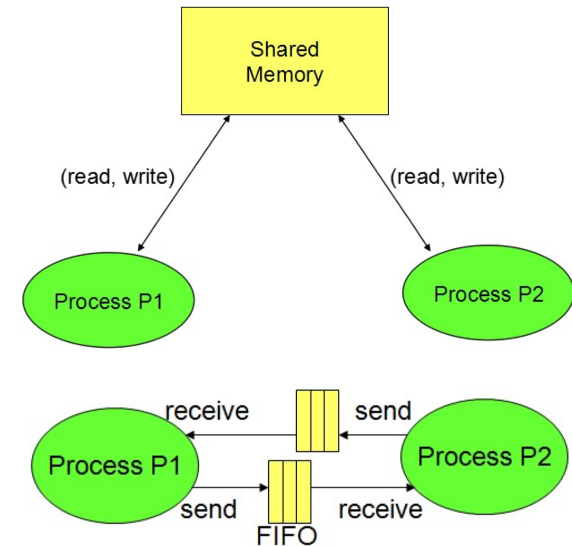
Message structure





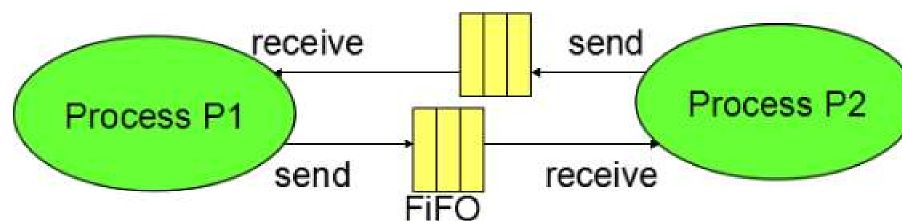
Communication Models: Comparison

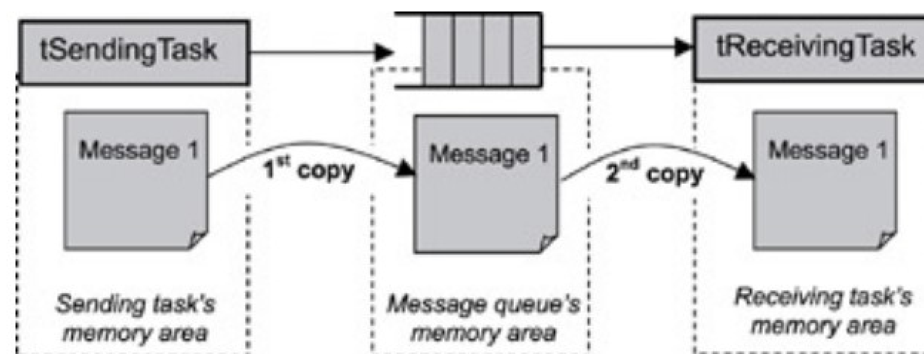
- Shared-Memory
  - Compatibility with well-understood (language) mechanisms
  - Ease of programming for complex or dynamic communications patterns
  - sharing of large data structures
  - Efficient for small items
  - Supports hardware caching
- Messaging Passing
  - Simpler hardware
  - Explicit communication
  - Implicit synchronization (with any communication)
    - OS服务：安全，执行开销大
- 两种方式可以实现同样的功能



消息队列实现

- 任务之间、ISR与任务之间的通信与同步
  - ISR可以写队列，但不能读
- 消息队列：一个可变长缓冲区
  - 数据
  - 指针
  - 空队列：用于同步
- 基本操作
  - 发送普通消息
  - 发送紧急消息
  - 广播消息
  - 接收消息
    - 任务可以选择是否等待
      - 按照FIFO方式等待
      - 按照优先级等待
  - 获得消息队列中未决消息数
  - 清空队列
  - 获得消息队列标识符



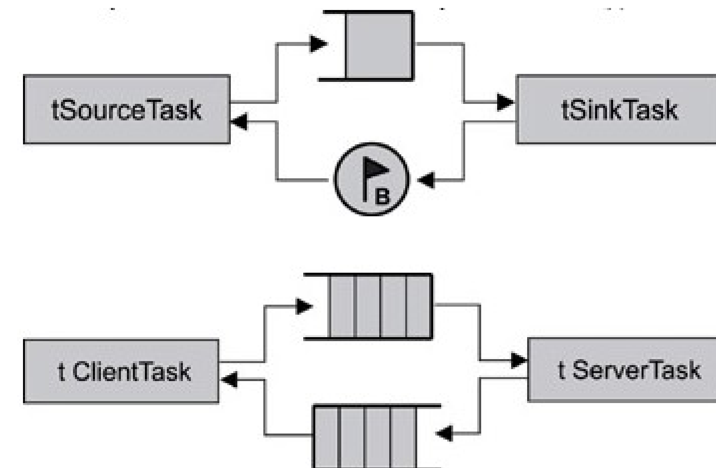


message queue

- 数据传输模式
  - Non-Interlocked, One-Way
  - Interlocked, One-Way
  - Interlocked, Two-Way
  - Broadcast Comm

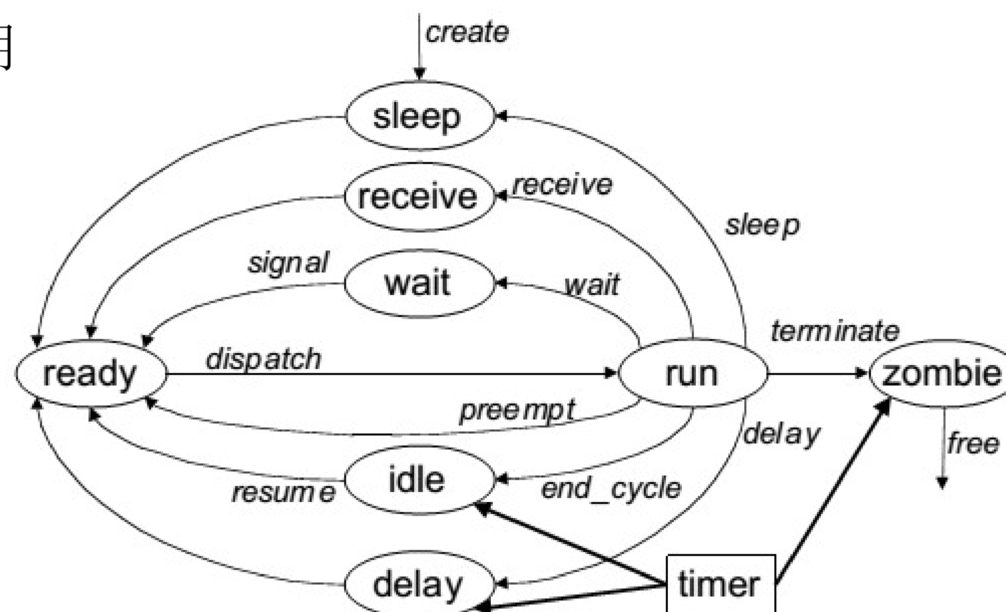
Queue Control Block
QCB
Memory
(System Pool or Private Buffers)
Sending Task Waiting List
Task
Task
Queue Name/ID
Maximum Message Length
Receiving Task Waiting List
Task
Task
Queue Length
Tail
Head
Queue Element

Message Delivered (msgs = msgs –1)
Queue Created (msgs = 0)
Message Delivered (msgs = 0)
Message Delivered (msgs = msgs –1)
Empty
Not Empty
Full
Message Arrived (msgs = 1)
Message Arrived (msgs = Queue Length)
Message Arrived (msgs = msgs +1)

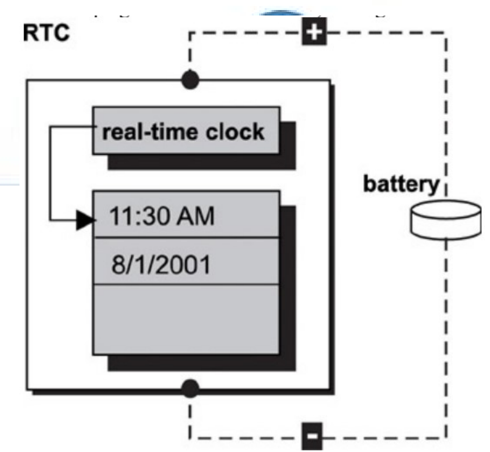


时间模型与定时器的用途

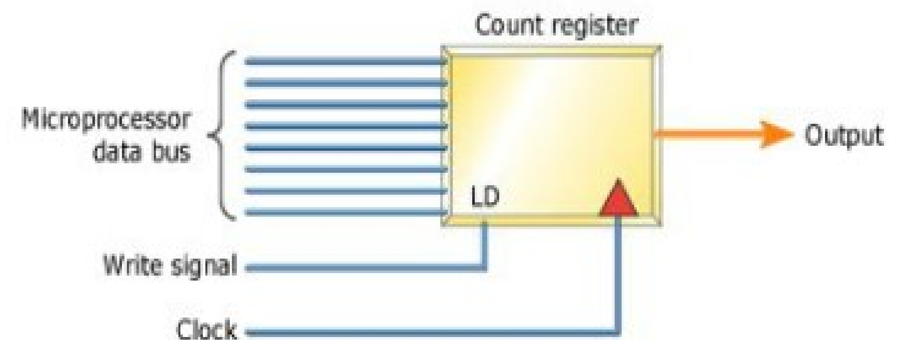
- 时间模型：timeline
  - 系统0时刻、时间单元（unit）、instant、interval、timestamp
- Timing constrains
  - delay（two events，latency？），deadline，duration（action）
- 定时器的用途
  - 处理定时事件：单次、周期
  - 任务延时
    - 更新时间片
    - 提供等待超时计时
    - 更新就绪队列
  - 时钟触发调度
    - 唤醒挂起的任务
  - 将消息放入消息队列
  - 更新统计信息



系统时钟

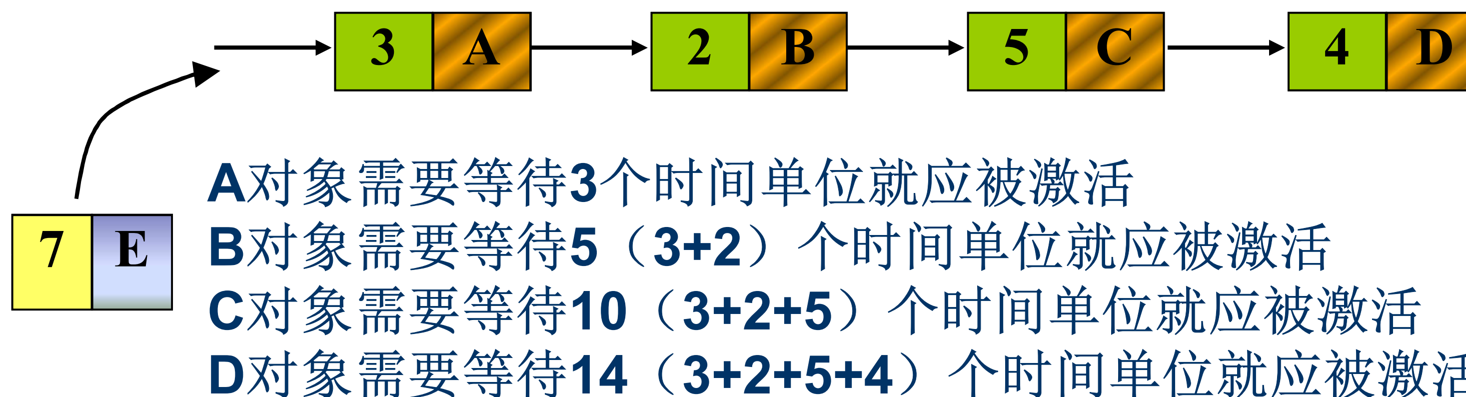


- 硬时钟：实时时钟（RTC）
- 软时钟：时钟节拍(Clock Tick)
  - 由PIT(programmable interval timer)产生
  - 含：计数器、定时器队列和ISR
    - 队列：保存与这个时钟绑定的所有定时器的挂起过期时间
  - 系统中可以有多个clock
  - Tick周期决定系统的时间分辨率
    - 如几百us～1ms（ucOS）～500ms
    - 如何得到更高分辨率？
- 时间管理
  - 定时器创建、删除、启停
  - 系统定时器、用户定时器



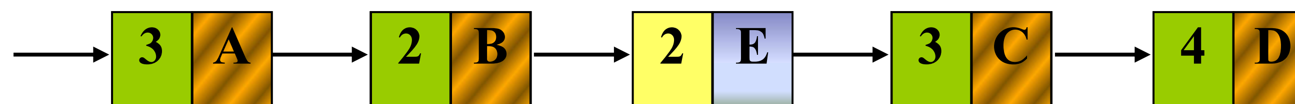
时间等待链处理：差分链

存放需要延迟处理的对象。
每个表项的**计时值**：并非当前时刻到表项激活时刻的绝对计数，而是该表项和先于它的所有表项的计数值之和。

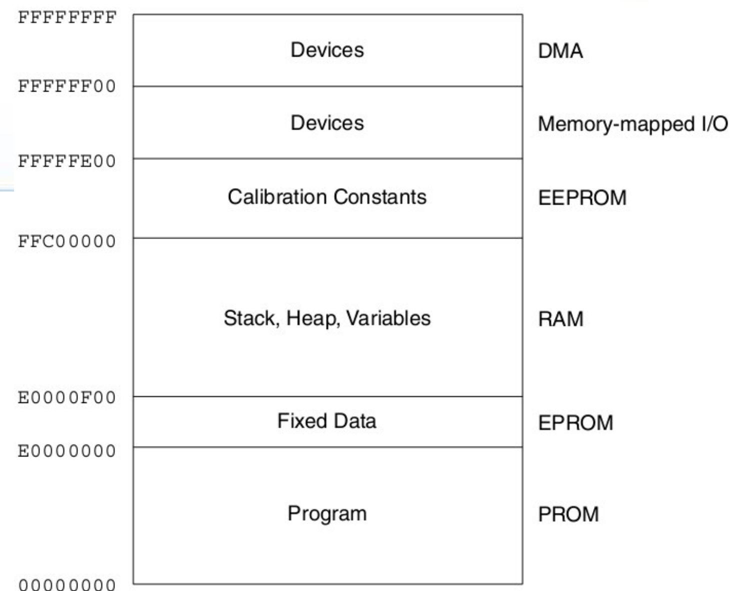


A对象需要等待3个时间单位就应被激活
B对象需要等待5（3+2）个时间单位就应被激活
C对象需要等待10（3+2+5）个时间单位就应被激活
D对象需要等待14（3+2+5+4）个时间单位就应被激活

由于7-3-2=2，而7-3-2-5=-3，
因此E对象需要插入到差分链中介于对象B和对象C之间的位置。



对存储器的需求



- 全局变量、堆变量、栈变量
- 前后台系统：取决于应用程序
- 多任务内核：
  - 小：1～3K
  - 大：100K
- 栈空间
  - 任务栈：每个任务需要独立的栈空间
    - 固定式：固定大小
    - 非固定式：μC/OS-II允许每个任务有不同大小的栈空间
  - 系统栈：ISR使用，考虑中断嵌套
- 堆空间：按可变大小的动态内存块分配
- RAM需求：应用程序代码量＋任务栈＋系统栈
- ROM需求：应用程序代码量＋常量

内存分配策略

- 大小
  - 固定尺寸分配策略
    - 申请内存时, 系统总是返回一个固定大小内存
      - 通常是2的指数倍
    - 延迟时间确定，分配和回收速度快
    - 不会产生太多的细小碎片
  - 可变大小分配策略
    - 按需分配
    - 须进行分区排序、分区合并等管理，开销大
    - 内存碎片多
- 地址连续性
  - 连续内存分配
    - 每个程序在内存中只占有一块单独的连续内存区域
  - 离散内存分配（分页、分段、段页式）
    - 解决内存碎片问题而提出的一种分配模式

动态内存分配

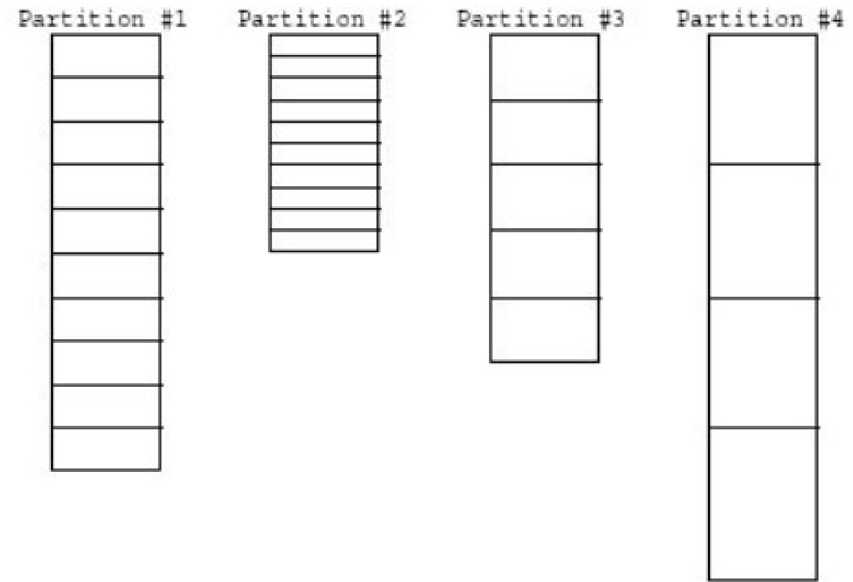
- 常规malloc()和free()：“按需分配”
  - 内存按固定大小分块
  - 找到大于用户申请大小的内存块，一分为二
    - 大小与请求的大小相等的一块给用户，另一块为碎片
    - 连续内存被分割成碎片，造成无内存可用
      - case4：空闲256-230=26，但无法使用
      - case6：两块不相邻，无法响应大于32的请求
    - “小块合并”造成算法时间不确定

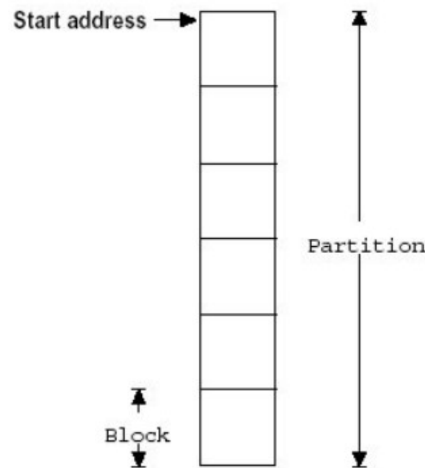
Fixed-Size Memory Management

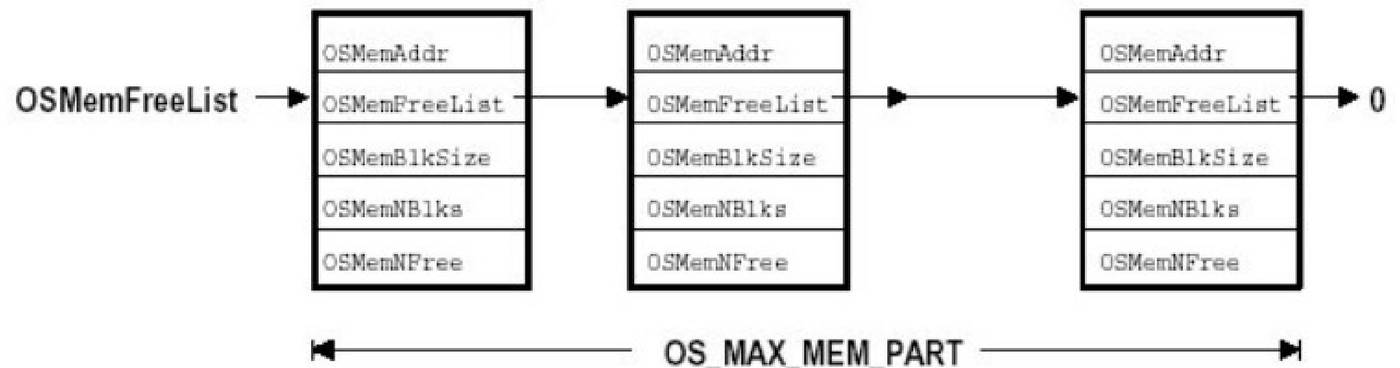
- 目标
  - minimal fragmentation
  - minimal management overhead
  - deterministic allocation time
- uC/OS内存管理策略：执行时间确定
  - “按块分配”
    - 应用程序可以从多个分区中请求多个块
    - 释放时须放回内存块所属的分区中，解决碎片问题
  - 提供的函数
    - OSMemCreat()、OSMemGet()、OSMemPut()、OSMemQuery()

内存控制块MCB-每个分区一个

- 初始化空闲链表
- OSMemCreat()
  - 建立分区







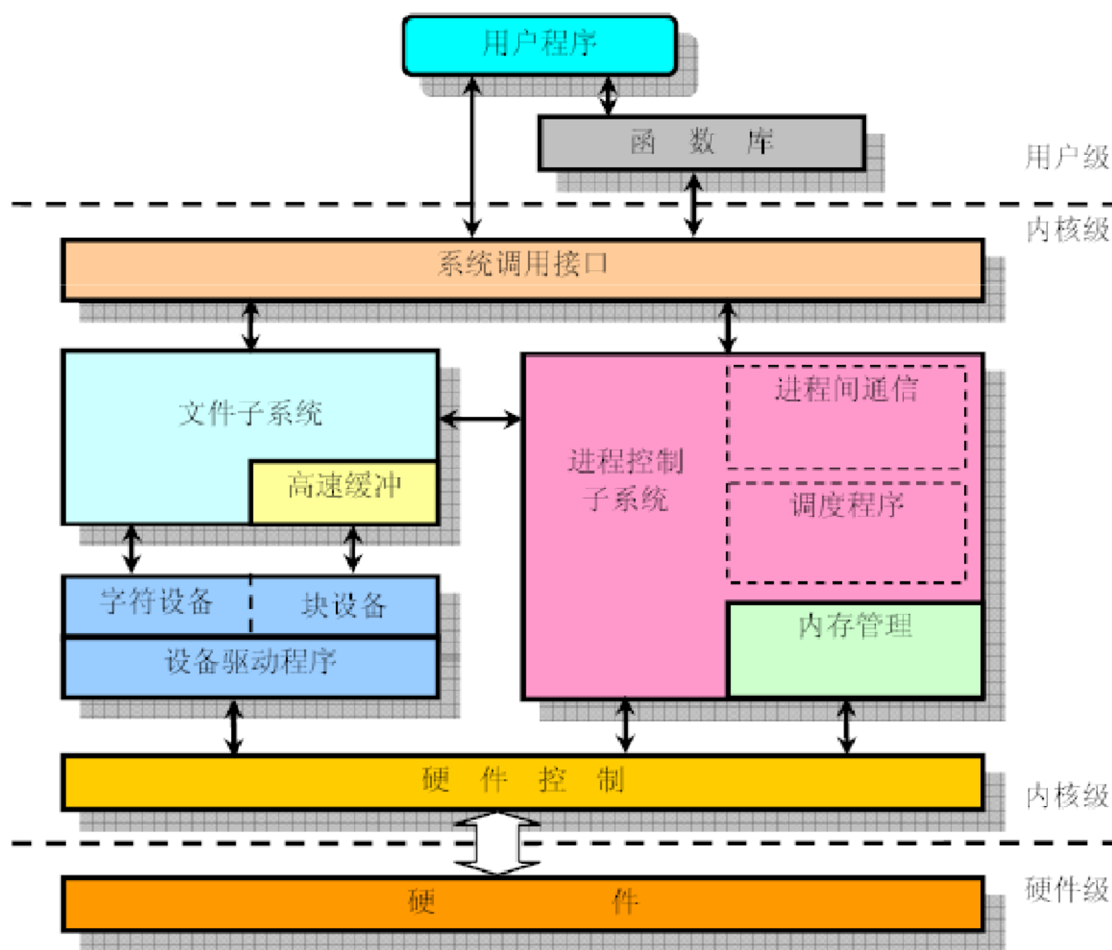
List of free memory control blocks.

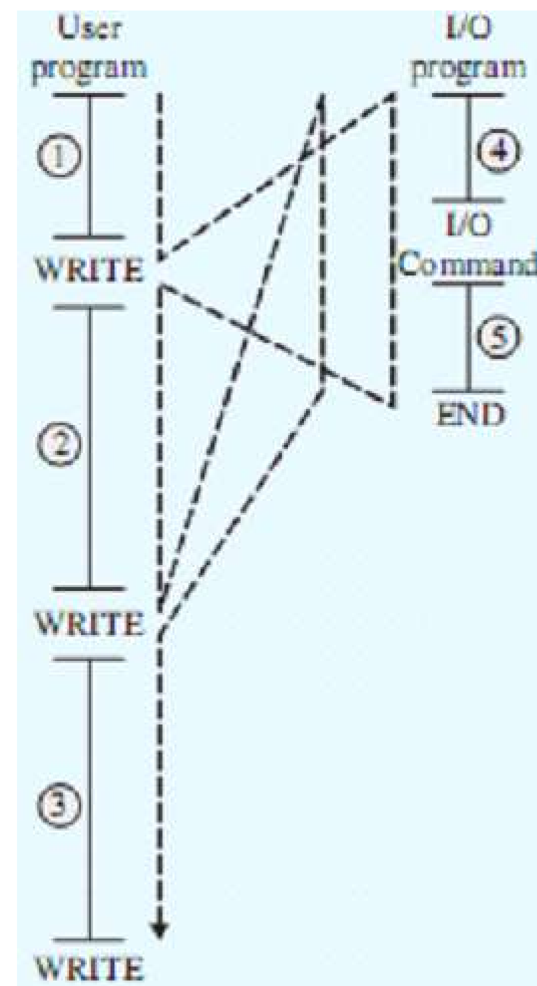
I/O管理

- 提供结构化访问设备的方法
- 设备驱动程序表(I/O driver table)
  - 设备驱动程序入口
- 设备驱动程序
  - 完成设备的初始化、打开、关闭、读写、并发控制等操作
- 设备驱动程序是“临界区”？

设备驱动程序

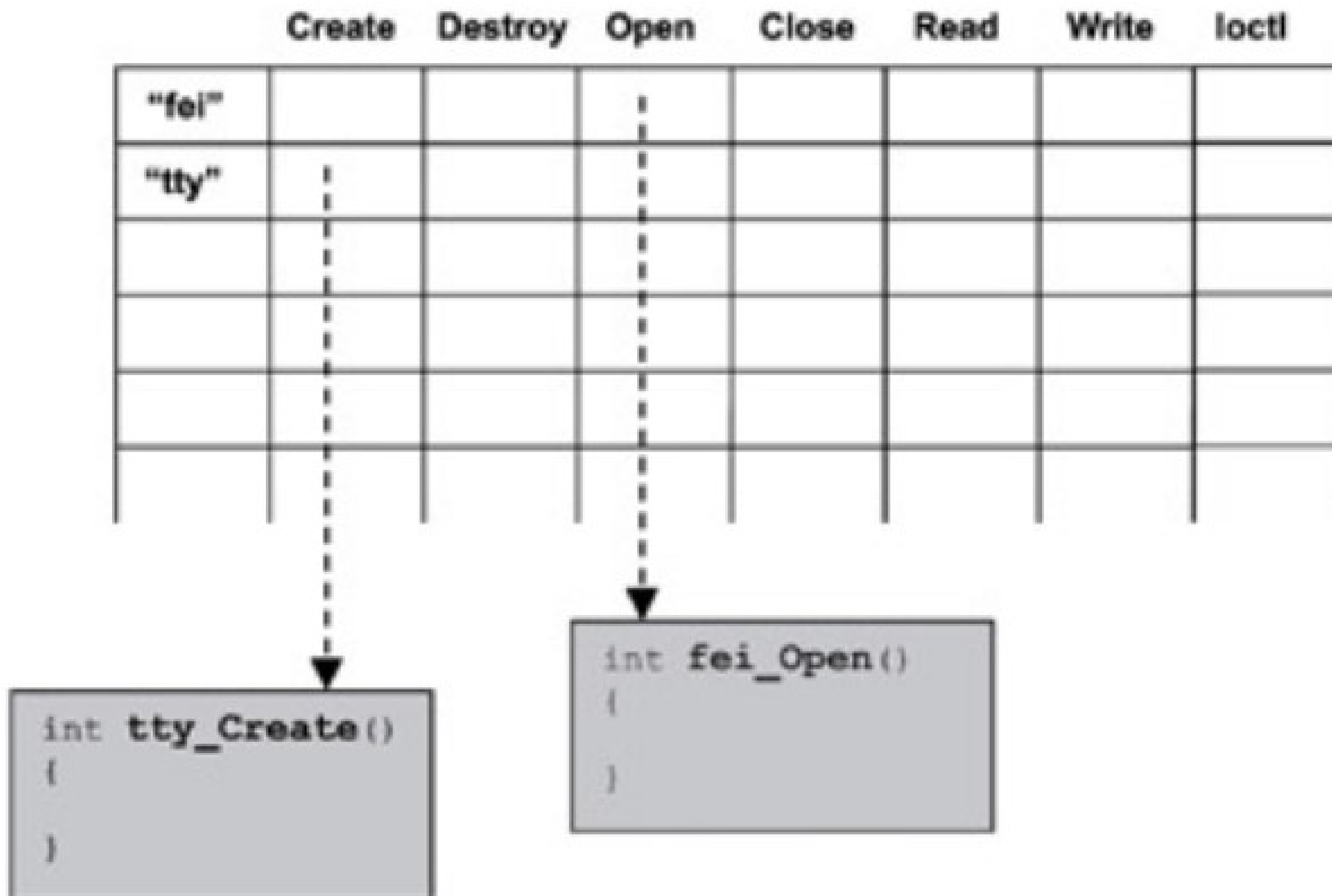
- 操作特定设备的程序：字符设备、块设备
- 将所有设备映射成“文件”





I/O driver table

- Any driver can be installed into or removed from this driver table



device driver的功能

- 设备管理
  - Hardware Startup, initialization of the hardware upon power-on or reset.
  - Hardware Shutdown, configuring hardware into its power-off state.
  - Hardware Install, allowing other software to install new hardware on-the-fly.
  - Hardware Uninstall, allowing other software to remove installed hardware on-the-fly.
  - Hardware Disable, allowing other software to disable hardware on-the-fly.
  - Hardware Enable, allowing other software to enable hardware on-the-fly.
- 读写操作
  - Hardware Read, allowing other software to read data from hardware.
  - Hardware Write, allowing other software to write data to hardware.
- 并发控制
  - Hardware Acquire, allowing other software to gain singular (locking) access to hardware.
  - Hardware Release, allowing other software to free (unlock) hardware.

驱动程序和设备

驱动程序对象

功能码1 → 读

功能码2 → 写

功能码n → 设备控制

→ 启动I/O

→ 卸载

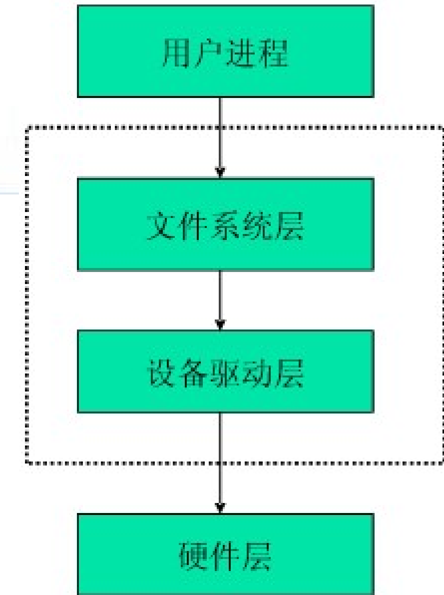
→ 取消

→ 设备对象（磁盘） → 设备对象（磁盘分区） → 设备对象（磁盘分区）

该驱动程序操作的设备

```
struct file-operation {
int （*lseek） （）；
int （*read） （）；
int （*write） （）；
int （*readdir） （）；
int （*select） （）；
int （*ioctl） （）；
int （*mmap） （）；
int （*open） （）；
int （*close） （）；
int （*release） （）；
int （*fsync） （）；
int （*fasync） （）；
int （*check-media-change） （）；
int （*revalidate） （）；
}
```

例：LED应用程序




```
int main(void)
{    int fd;
     char led_on = 0x01;            //待显示的数据

     fd = open("/dev/led/0", O_RDWR);  //打开led设备
     if(fd==-1)  {
        printf("can not open device\n");
        exit(1);
     }
     write(fd, &led_on, 1);                //LED开
     close(fd);                            //关闭设备文件
     return 0;
}
```

RTOS内核的运行时机和过程

External Interrupts
Hardware/ Software exceptions
System Calls
Clock Interrupts
Trap
Immediate Interrupt Service
case of
create_thread
suspend_thread
destroy_thread
⋮
create_timer
timer_sleep
timer_notify
⋮
open
read
⋮
other system calls
Time Services and Scheduling
Scheduling
Kernel
Return_from_exception

- 内核运行时机
  - 初始化
  - 系统调用
    - 任务创建
    - 互斥
    - 同步
    - 通信
    - 内存分配
  - 任务调度
    - 时钟驱动
    - 事件驱动
  - 中断和异常
  - 模式切换

API

Number of system APIs

14
12
10
8
6
4
2
0

μITRON μTKernel μC-OS/II FreeRTOS Salvo Echidna EmbOS Erika Hartik

- System management
- Interrupt management
- Task management
- Task-dependent synchronization
- Semaphore
- Eventflag
- Mailbox
- Data queue
- Mutex
- Message buffer
- Fixed size memory pool
- Variable size memory pool
- System time management
- Trace API

实时性评价指标：可预测性

- 响应时间（Response Time）
  - 识别一个外部事件到作出响应的时间
    - 控制应用中是最重要的指标，如果事件不能及时的处理，系统可能就会崩溃。
    - 中断响应
    - 任务响应
- 任务切换时间
- 系统调用的执行时间
  - 最坏时间分析
  - 考虑任务切换影响
- 同步时间
  - 信号量获取/释放
- 通信时间
  - 消息队列传输

中断响应的过程

中断嵌套时间、关中断时间、指令最长执行时间。。。

中断响应
- 保存断点（PC、PSW）
- 设置工作模式
- 获得中断向量
- 转向ISR入口

保存现场（系统寄存器、用户寄存器）

对抢占式，要执行“中断入口函数”

用户服务

对抢占式，要执行“中断出口函数”

恢复现场

中断返回（PSW、PC）

中断延迟：中断嵌套时间、关中断时间、指令最长执行时间 + 中断响应 + 保存现场

中断响应 | 中断处理 | 中断恢复

硬件 | ISR | 硬件/软件

中断等待、响应、服务、恢复

时间

中断请求

任务

保存的CPU环境

ISR

用户ISR代码

恢复的

CPU环境

任务

中断等待

中断响应

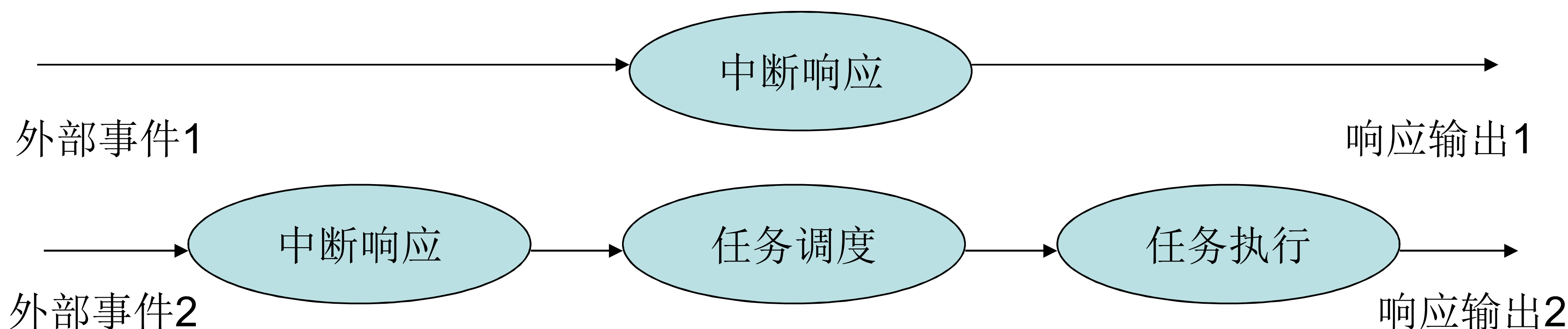
中断恢复

中断处理时间

- 中断等待（延迟）时间：执行第一条ISR指令
  - 最大关中断时间
    - ＝max[max（内核关中断时间），max（应用关中断时间）]
  - 中断嵌套时间
    - 与应用相关
- 中断响应时间：执行第一条用户中断服务指令
  - 非抢占调度：立即执行ISR
    - ＝中断延迟＋保存现场
  - 抢占调度：进行用户服务前要执行“ISR入口函数”，以便ISR结束后进行任务调度
    - ＝中断延迟＋保存现场＋入口函数
- 中断服务时间
- 测量
  - 定时器中断
  - 第一条ISR指令读计数值

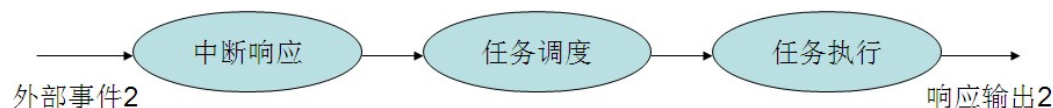
操作系统对外部事件响应的模式

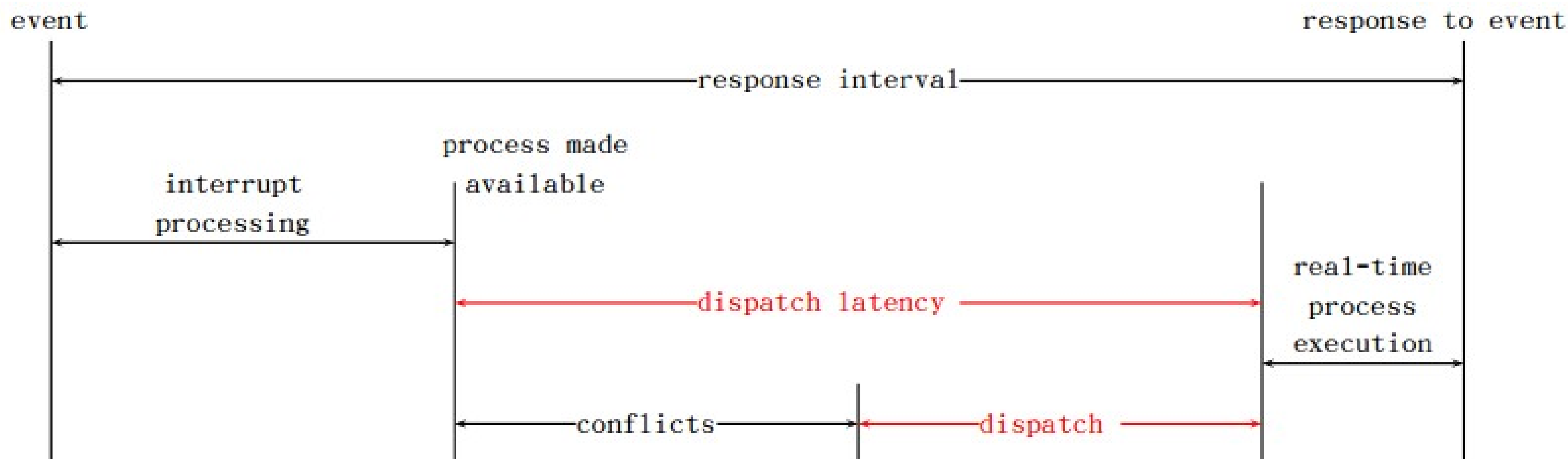
- 模式1：在ISR中完成事件处理
  - 响应时间＝中断响应时间 ＋ 中断处理时间
- 模式2：中断响应＋任务执行，在“任务”中进行事件处理
  - 优点：减少中断服务时间，有利于高优先级任务的实时性
  - 响应时间＝中断响应时间＋任务调度时间＋任务执行时间
    - 任务调度算法、任务数、任务通信、资源共享
    - 事件发生频率：频率越低，响应时间越容易满足
    - 中断响应时间：与模式1中不同
    - 任务执行时间：最坏时间分析(WCET)



事件的响应时间response time

- 从事件到达到系统给出应答信号的时间。
  - 中断处理：中断延迟、响应、服务、恢复
  - 任务响应时间：又称“调度延迟”
    - 从任务对应的中断产生到任务真正开始执行的时间
  - 任务切换时间（Context-switching time）
  - 系统调用执行时间
- WCRT分析与测试？





任务响应时间

Time

Task Response

Interrupt Request (1)

µC/OS-II or your application has interrupts disabled.
(2)

Interrupt Recovery

TASK

TASK

No New HPT or,
OSLockNesting > 0

Vectoring
(3)

Return from interrupt
(9)

Saving Context
(4)

Restore context
(8)

Notify kernel:
OSIntEnter() or,
OSIntNesting++
(5)

Notify kernel: OSIntExit()
(7)

User ISR code

(6)

Interrupt Response

Notify kernel: OSIntExit()
(10)

Restore context
(11)

ISR signals a task

Return from interrupt
(12)

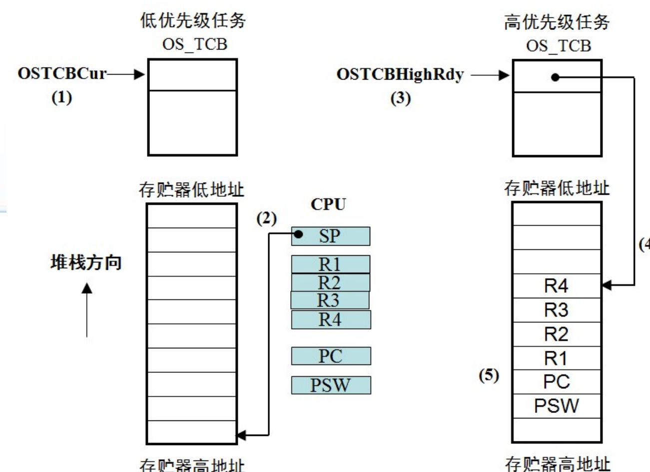
New HPT

TASK

Interrupt Recovery

Task Response

任务切换时间



- 切换过程
  - 保存上下文
  - 任务调度
    - 强实时系统：要求调度时间确定，即不随任务负载而变
  - 恢复上下文
- 测试
  - 第一步：交替的挂起/恢复任务的时间测试
    - 用一个低优先级任务恢复一个挂起的高优先级任务，高优先级任务又立即挂起
      - ＝两次切换时间＋（一次挂起时间＋一次恢复时间）
  - 第二步：挂起/恢复时间的测试
    - 用一个高优先级任务反复挂起和恢复一个低优先级任务
      - ＝一次挂起时间＋一次恢复时间

切换时间＝（交替的挂起/恢复任务的时间－挂起/恢复时间）/2

Rhealstone,1989

- 用于测试实时多任务操作系统的基准程序
  - 在Rhealstone中，采用了简化的WhetStone测试程序作为测试任务，以屏蔽不同体系结构CPU 的差异.
- 测量ERTOS中六个关键操作的时间，并将它们的加权和称为Rhealstone数
  - 任务切换时间、任务抢占时间、中断延迟、信号量阻塞时间、死锁解除时间、邮箱传输延迟
  - 平均值
  - 没有考虑任务调度的性能（如deadline）

$$Rhealstone \quad Metric \quad = \quad a_1 * t_{ts} + a_2 * t_{tp} + a_3 * t_{il} + a_4 * t_{ss} + a_5 * t_{up} + a_6 * t_{dt}$$

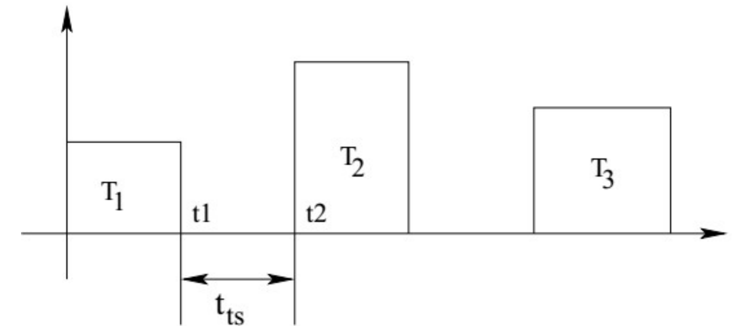
Rhealstone方法：

1、任务切换时间(task switching time)：

– 即系统在两个独立的、处于就绪态并具有相同优先级的任务之间切换所需要的时间。

– 包括三个部分：

- 保存当前任务上下文的时间
- 调度程序选中新任务的时间
- 恢复新任务上下文的时间。



– 此时间取决于

1）保存任务上下文所用的数据结构

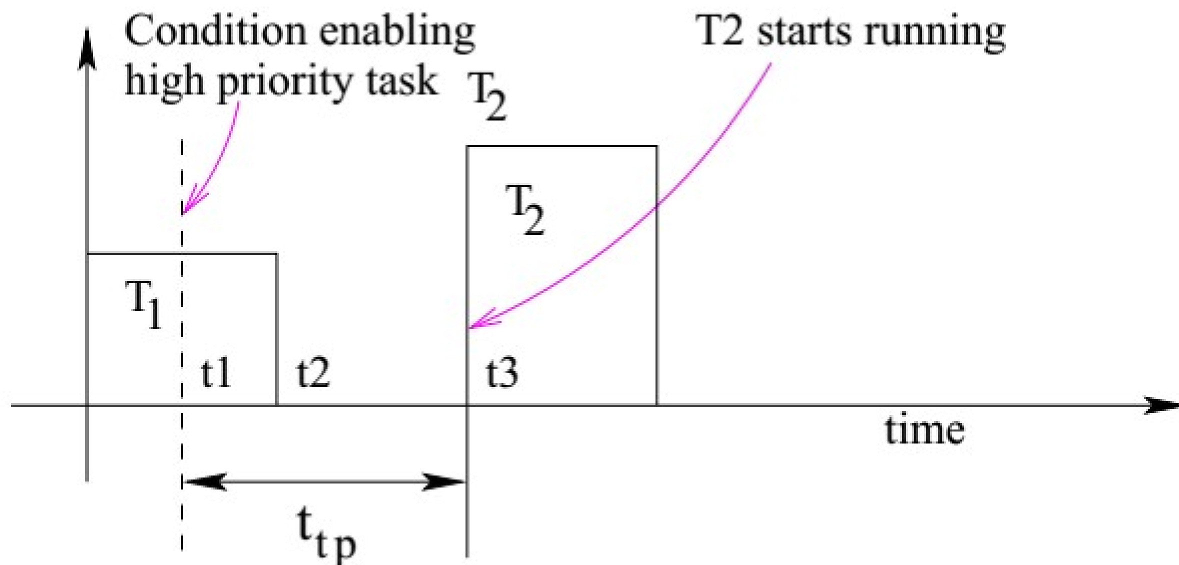
2）操作系统采用的调度算法的效率。

Rhealstone方法：

2、抢占时间（preemption time）

即系统将控制从低优先级的任务转移到高优先级任务所花费的时间。

- 系统必须首先识别引起高优先级任务就绪的事件，比较两个任务的优先级，确定发生抢占
- 抢占时间中包括了任务切换时间。

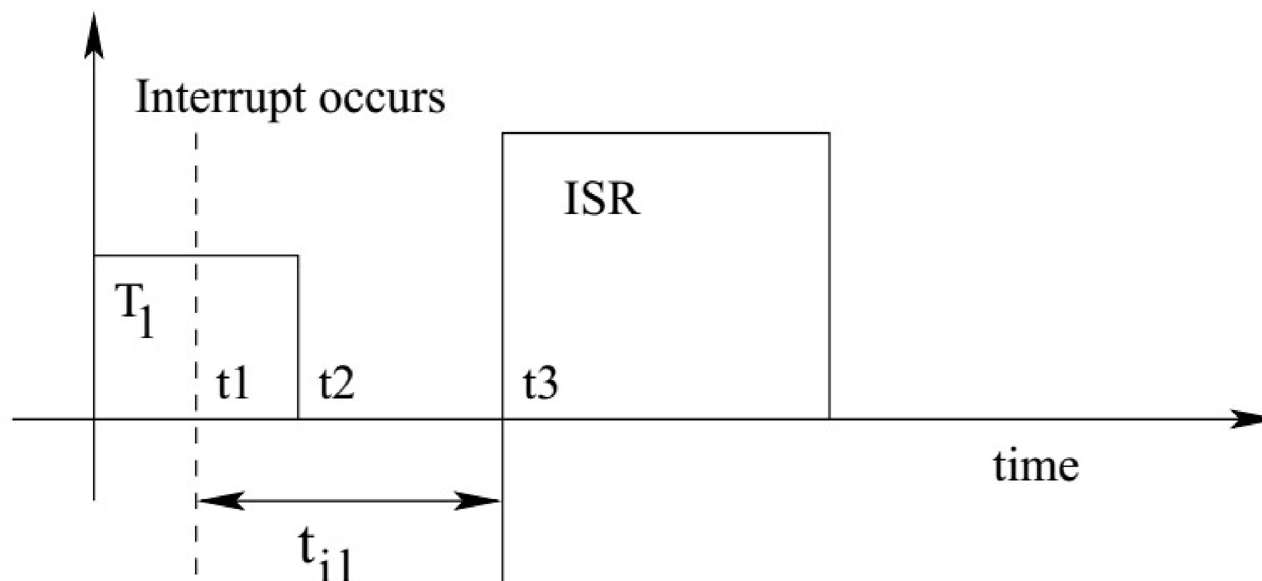


Rhealstone方法：

3、中断延迟时间（interrupt latency time）

即从中断到ISR第一条指令所持续的时间间隔

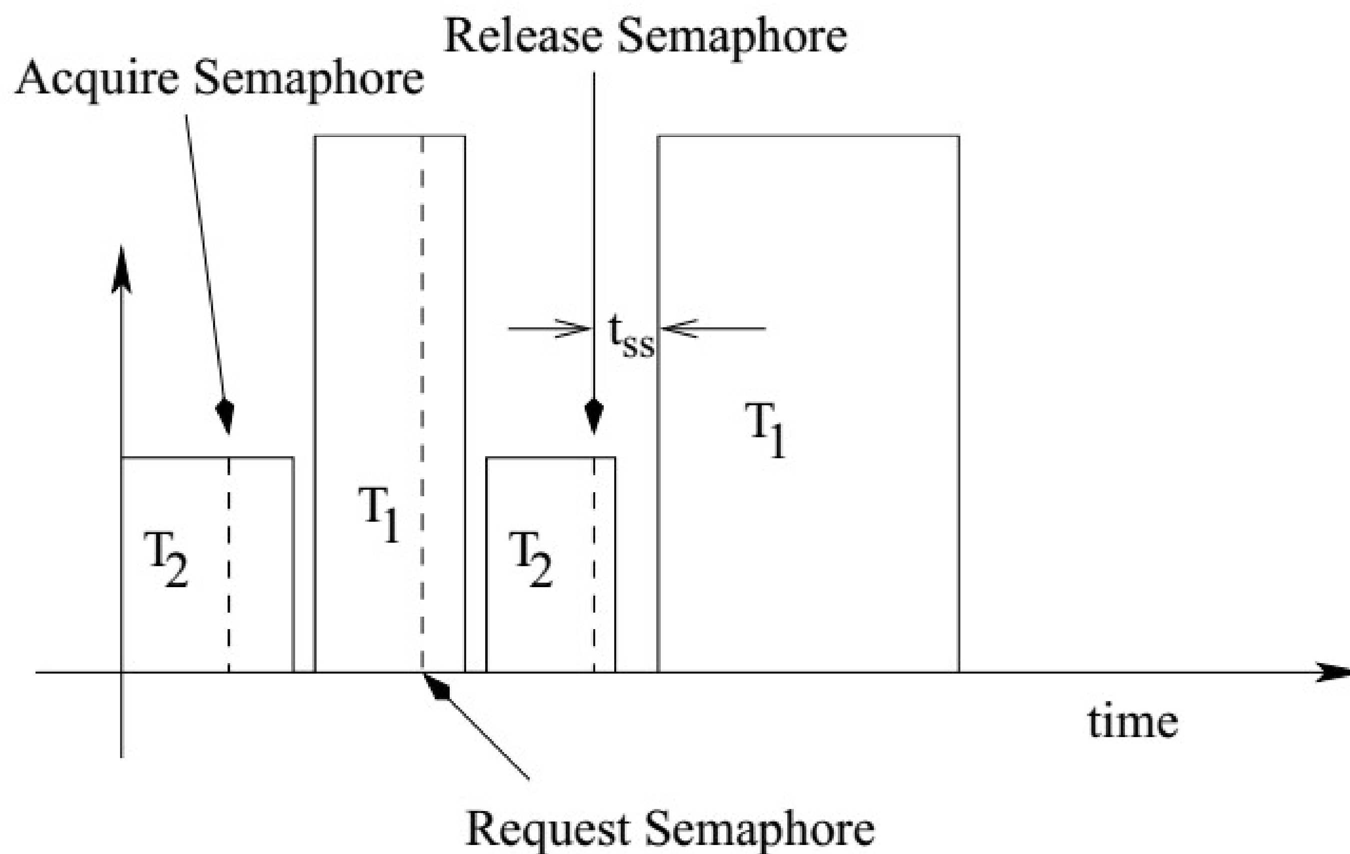
- 由四部分组成：
  - 硬件延迟部分(通常可忽略不计)
  - 关中断时间
  - 处理器完成当前指令的时间
  - 中断响应周期的时间



Rhealstone方法：

4、信号量混洗时间(semaphore shuffling time)

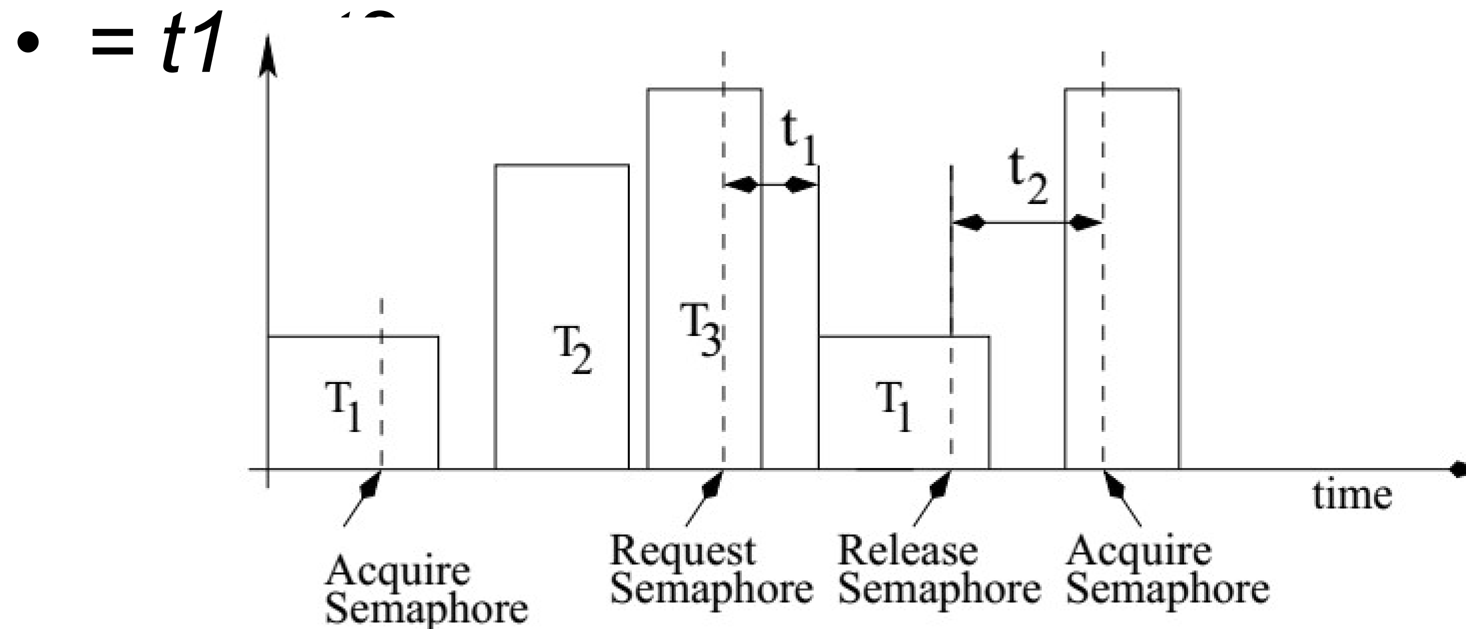
从一个任务释放信号量到另一个等待该信号量的任务被激活的时间延迟。



Rhealstone方法：

5、优先级翻转上界

- Unbounded priority inversion time
  - *t1*：OS发现优先级反转，执行资源占有者T1
  - *t2*：T1释放资源，T3得到资源开始执行
- = *t1*



Rhealstone方法：

5、死锁解除时间(deadlock breaking time)

即系统解开处于死锁状态的多个任务所需花费的时间。

– 死锁解除时间反映了RTOS解决死锁的算法的效率。

6、数据包吞吐率(datagram throughput time)

指一个任务通过调用RTOS的原语，把数据传送到另一个任务去时，每秒可以传送的字节数。度量消息传递的效率

进程分派延迟时间法PDLT

- 另一个常用的测量ERTOS性能的方法
  - 实时系统中，任务总是等待外部事件引发的中断来激活它。
    - 当一个中断产生后，系统必须迅速停止当前运行的低优先级任务，将控制权交给被激活的实时任务。
  - PDLT：**Process Dispatch Latency Time**
    - 从中断的产生到由中断激活的实时任务开始执行之间的时间间隔。

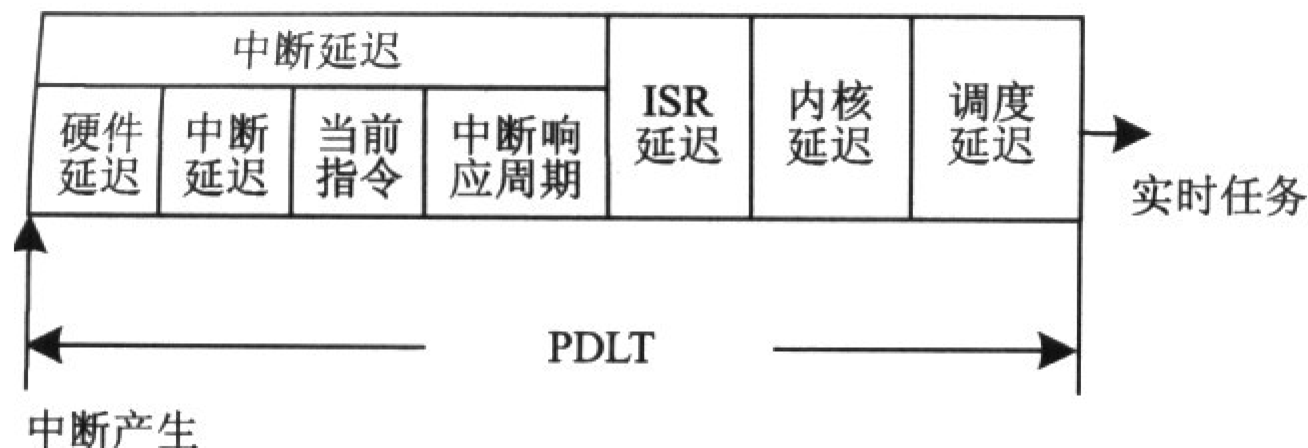


图 1 进程分派延迟时间

RTEMS的实时性(us)

	中断时延		Context转换	
	系统负载中等			
	最大	平均	最大	平均
RtLinux	13.5	1.7	33.1	8.7
RTEMS	15.1	1.3	16.4	2.2
RTEMS[1]	14.9	1.3	16.9	2.3
VxWorks	13.1	2.0	19.0	3.1
	重负载			
RtLinux	196.8	2.1	193.9	11.2
RTEMS	20.5	2.9	51.3	3.7
RTEMS[1]	19.2	2.4	213	10.4
VxWorks	25.2	2.9	38.8	9.5

Execution time for four RTOSes

45
40
35
30
25
20
15
10
5
0

Task switch time
Get semaphore
Release semaphore
Pass semaphore (from 1 to another task)
Pass message to queue
Retrieve message from queue
Pass message (from 1 to another task)
Acquire fixed-size memory block
Release fixed-size memory block
Task activation from interrupt

µC-OS/II
µTKernel
EmbOS
µITRON

嵌入式操作系统认证标准

- RTCA/DO-178B
  - “Software Considerations in Airborne Systems and Equipment Certification”
  - RTCA(航空无线电委员会)
    - 美国航空航天管理局(FAA)采用它的规范/标准。
    - “安全标准”，针对软件开发流程的规范。
  - VxWorks，uC/OS-II，LynxOS-178等通过。
- UL/IEC-61508
  - IEC（International Electrotechnical Commission）
  - 提出了Functional Safety的概念，并且制定了安全完整性等级SIL作为评估的标准。
  - 交通、核系统等领域
- 面向医用系统的标准FDA 510(k)
- uC/OS-II通过了以上三个标准的认证

小结

- RTOS操作系统结构与功能： *可抢占内核*
- 任务，事件
  - 线程=轻量级任务，降低调度延迟
- 任务间的协同：通信、同步（协作）、互斥（竞争）
  - 例：生产者-消费者
- 可重入、可抢占、互斥、锁、优先级反转、资源访问控制
  - *非抢占“几乎不需要”使用信号量保护共享数据？*
  - *PIP/PCP并没有完全消除优先级反转！*
- 操作系统实时性能分析
  - 强实时系统，时间性能指标应与系统状态和负载无关。
  - 实时内核的运行时间可预测：测试方案
    - 中断延迟
    - 任务切换
    - 任务通信与同步
    - 内存管理
    - 系统调用

	可抢占	非抢占	ET	TT
响应时间	快	不确定	快	慢
服务时间	不确定	确定		
互斥	需要	无需		

作业（选二）

- 如何定义某任务的优先级？
- ISR、事件、ASR的比较？
- 任务执行需要哪些“资源”？
- 如何得到更高分辨率（小于几百us）的定时器？
- 互斥锁与信号量的区别？
- PIP、PCP、SRP的原理与区别
- 举例典型的任务间及ISR与任务间进行同步和通信的应用场景，说明何时应采用哪种机制。
- 调研典型的嵌入式操作系统的性能指标。

Thank you