

- 《现代操作系统》读摘

进程与线程

进程

- **伪并行**：某一时间，CPU是能运行一个进程。但一秒钟内可以运行多个进程，这就会产生并行的错觉
- **多处理器系统**：该系统有两个或多个CPU共享一个物理内存

进程模型

- **进程**：一个进程就是一个正在执行程序的实例
- **多道程序设计**
- 因为先今的芯片是多核的，包含多个CPU
 - 一些帮助理解的视频链接：
 - [进程和线程是什么关系？有什么区别？](#)
 - [CPU的超线程技术到底是什么意思？有啥用？](#)
- 一个进程是某种类型的一个活动，它有程序、输入、输出以及状态。如果一个程序运行了两遍，则算作两个进程。

进程的创建

- 主要事件导致进程的创建：
 - 系统初始化
 - 正在运行的程序执行了创建进程的系统调用
 - 用户请求创建一个新进程
 - 一个批处理作业的初始化
- **守护进程**是一种在后台执行的计算机程序
- **写时复制**其核心思想是，如果有多个调用者同时请求相同资源（如内存或磁盘上的数据存储），他们会共同获取相同的指针指向相同的资源，直到某个调用者试图修改资源的内容时，系统才会真正复制一份专用副本给该调用者，而其他调用者所见到的最初的资源仍然保持不变。这过程对其他的调用者都是透明的。此作法主要的优点是如果调用者没有修改该资源，就不会有副本被建立，因此多个调用者只是读取操作时可以共享同一份资源。

进程的终止

- 正常退出
- 出错退出
- 严重错误
- 被其他进程杀死

进程的层次结构

- 进程只有一个父进程，但是可以有零个、一个或多个子进程
- 所有进程都属于以init为根的一个树
- 在创建进程的时候，父进程得到一个特别的令牌（称为**句柄**），该句柄可以用来控制子进程

进程的状态

- 显示进程的三种状态：
 - 运行态
 - 就绪态
 - 阻塞态

```
graph LR;
    运行状态 --1--> 阻塞状态;
    就绪状态 == <=2-==3=> === 运行状态;
    阻塞状态 --4--> 就绪状态;
```

说明： 1.进程因为等待输入而被阻塞 2.调度程序选择另一个进程 3.调度程序选择这个进程 4.出现有效输入

- 操作系统发现进程不能运行下去时，发生转换1
- 转换2和3是由进程调度程序引起的
- 当进程等待的一个外部事件发生时（如一些输入到达），发生转换4

进程的实现

- 操作系统维护着一张表格（一个结构数组），即**进程表**。每个进程占用一个表项（**进程控制块**）。该表项包含了进程状态的重要信息，如下：
 - **进程管理**
 - 寄存器
 - 程序计数器
 - 程序状态字
 - 堆栈指针
 - 进程状态
 - 优先级
 - 调度参数
 - 进程ID
 - 父进程
 - 信号
 - 进程开始时间
 - 使用的CPU时间
 - 子进程的CPU时间
 - 下次定时器时间
 - **存储管理**
 - 正文段指针
 - 数据段指针
 - 堆栈段指针
 - **文件管理**
 - 根目录
 - 工作目录
 - 文件描述符
 - 用户ID

■ 组ID

- 与每一I/O类管理的是一个称做**中断向量**的位置
- 中段发生后操作系统最底层的工作步骤：
 1. 硬件压入堆栈程序计数器
 2. 硬件从中断向量装入新的程序计数器
 3. 汇编语言过程保存寄存器值
 4. 汇编语言过程设置新的堆栈
 5. C中断服务例程运行（典型地读和缓冲输入）
 6. 调度程序决定下一个将运行的进程
 7. C过程返回至汇编代码
 8. 汇编语言过程开始运行新的进程

多道程序设计模型

- 假设一个进程等待I/O操作时间与其停留在内存中的时间比为 p ，当内存中同时有 n 个进程时，则所有 n 个进程都在等待I/O的概率是 p^n 。CPU利用率公式： $\text{CPU利用率} = 1 - p^n$ 。 n 称为**多道程序设计的道数**

线程

线程的使用

- 需要多线程的理由：
 - 并行实体拥有共享同一个地址空间和所有可用数据的能力
 - 由于线程比进程更轻量级，所以他们比进程更容易（更快）创建，也更容易撤销
 - 存在大量I/O处理，使用多线程可以加快应用程序执行的速度
- **高速缓存**
- 一个称为**分派程序**的线程从网络中读入工作请求，在检查后，分派线程选了一个**工作线程**，提交改请求（类比线程池和进程池原理）
- 每个计算都有一个被保存的状态，存在一个会发生且使得相关状态发生改变的事件集合，把这类设计称为**有限状态机**
- 构造服务器的方法：
 - **多线程** 并行性、阻塞系统调用
 - **单线程进程** 无并行性、阻塞系统调用
 - **有限状态机** 并行性、非阻塞系统调用、中断

经典的线程模型

- **进程**用于把资源集中到一起，而**线程**则是在CPU上被调度执行的实体
- 每个**进程**中的内容：
 - 地址空间
 - 全局变量
 - 打开文件
 - 子进程
 - 即将发生的定时器
 - 信号与信号处理程序
 - 账户信息

- 每个线程的内容：
 - 程序计数器
 - 寄存器
 - 堆栈
 - 状态
- 线程概念试图实现的是，共享一组资源的多线程的执行能力，以便这些线程可以为完成某一任务而共同工作

在用户空间中实现线程

- 有两种方法实现线程包：在用户空间中和在内核中
- 在用户空间中：
 - 在用户空间管理线程时，每个进程需要有其专用的线程表
 - 优点：用户级线程包可以在不支持现成的操作系统上实现；线程切换不需要陷入内核，这比陷入内核要快一个数量级或更多；保存该线程状态的过程和调度程序都只是本地过程，所以启动它们比进行内核调用效率更高；它允许每个进程有自己定制的调度算法
 - 缺点：如何实现阻塞系统调用？包装器

在内核中实现线程

- 在内核中有用来记录系统中所有线程的线程表
- 内核的线程表保存了每个线程的寄存器、状态和其他信息。内核还维护了传统的进程表
- 系统对于线程的创建和撤销采取环保的方式。回收线程时，就把它标志为不可运行的，但其内部的数据结构没有受到影响；创建线程时，就将重新启动某个旧线程。其线程管理代价很小

混合实现

- 将用户级线程的优点和内核级线程的优点结合起来

调度程序激活机制

- 上行调用

弹出式线程

- 一个消息到达导致系统创建一个处理该消息的线程，称为弹出式线程，结果是消息到达与处理开始之间的时间非常短

使单线程代码多线程化

进程间通信

- 进程间通信 IPC问题
 - 进程如何把信息传递给另一个
 - 确保两个或更多的进程在关键活动中不会出现交叉
 - 与正确的顺序有关

竞争条件

- **Murphy法则**：任何可能出错的地方终将出错
- **竞争条件**：两个或多个进程读写某些共享数据，而最后的结果取决于进程运行的精确时序

临界区

- **互斥**：阻止多个进程同时读写共享数据
- **临界区域**：对共享内存进行访问的程序片段
- 需要满足条件：
 - 任何两个进程不能同时处于其临界区
 - 不应CPU的速度和数量做任何假设
 - 临界区外运行的进程不得阻塞其他进程
 - 不得使进程无限期等待进入临界区

忙等待的互斥

- **屏蔽中断**
 - 把屏蔽中断的权利交给用户进程是不明智的
- **锁变量**
 - 设想有一个共享锁变量，其初始值为0。有两个进程，当有一个进程要进入时，将值置为1，另一进程则看到锁值为1就等待
- **严格轮换法**
 - **忙等待**：连续测试一个变量直到某个值出现为止。这种方式浪费CPU时间，所以应该避免。用于忙等待的锁称为**自旋锁**。
 - 示例：

```
// 进程0
while (TRUE){
    while(trun != 0);
    critical_region(); // 临界区域
    turn = 1;
    noncritical_region(); // 非临界区域
}
// 进程1
while (TRUE){
    while(trun != 1);
    critical_region(); // 临界区域
    turn = 0;
    noncritical_region(); // 非临界区域
}
```

- **Peterson解法**

```
# include <iostream>

using namespace std;

#define FALSE 0
```

```

#define TRUE 1
#define N 2 // 进程数量

int turn; // 轮到谁
int interested[N]; // 所有值初始化为0

void enter_region(int process){ // 进程是0或1
    int other; // 另一进程号

    other = 1 - process; // 另一进程
    interested[process] = TRUE; // 表示感兴趣
    turn = process; // 设置标志
    while (turn == process && interested[other] == TRUE); // 空语句
}

void leave_region(int process){ // 哪个进程离开
    interested[process] = FALSE; // 表示离开临界区
}

```

- TSL指令
 - TSL RX, LOCK称为测试并加锁

睡眠与唤醒

- 优先级反转问题：有三个进程（其优先级从高到低分别为T1、T2、T3），有一个临界资源CS（T1与T3会用到）。这时，T3先执行，获取了临界资源CS。然后T2打断T3。接着T1打断T2，但由于CS已被T3获取，因此T1被阻塞，这样T2获得时间片。直到T2执行完毕后，T3接着执行，其释放CS后，T1才能获取CS并执行。这时，我们看T1与T2，虽然T1优先级比T2高，但实际上T2优先于T1执行。这称之为优先级逆转。
- 生产者-消费者问题
 - 也称为有界缓冲区问题，实例：

```

#include <iostream>

using namespace std;

#define N 100 // 缓冲区中的槽的数目
int count = 0; // 缓冲区中的数据项数目

void producer(void){
    int item;
    extern int count;
    while(true){ // 无限循环
        item = produce_item(); // 产生下一个数据项
        if (count == N) sleep(); // 如果缓冲区满了，就进入休眠
        insert_item(item); // 将新数据项放入缓冲区中
        count = count + 1; // 将缓冲区的数据项计数器增1
        if (count == 1) wakeup(consumer); // 缓冲区空吗?
    }
}

```

```

void consumer(void){
    int item;
    extern int count;
    while(true){ // 无限循环
        if (count == 0) sleep(); // 如果缓冲区空，就进入休眠
        item = remove_item(); // 从缓冲区中取出一个数据项
        count = count - 1; // 将缓冲区的的数据项计数器减1
        if (count == N - 1) wakeup(producer); // 缓冲区满吗？
        consume_item(); // 打印数据项
    }
}

```

~ 唤醒等待位：当发给一个（尚）未睡眠进程的wakeup信号消失，一种快速的弥补办法是修改规则，加上一个唤醒等待位

信号量

- 使用一个整型变量来累计唤醒次数
- 信号量的另一种用途是用于实现同步
- 用信号量解决生产者-消费者问题：
 - 使用三个信号量：
 - full：用来记录充满的缓冲槽数目，初值为0
 - empty：记录空的缓冲槽数目，初值为槽的数目
 - mutex：用来确保生产者和消费者不会同时访问缓冲区，初值为1

```

#include <iostream>

using namespace std;

#define N 100 // 缓冲区中的槽的数目
typedef int semaphore; // 信号量是一种特殊的整型数据
semaphore mutex = 1; // 控制对临界区的访问
semaphore empty = N; // 计数缓冲区的空槽数目
semaphore full = 0; // 计数缓冲区的满槽数目

void producer(void){
    int item;

    while(true){ // true是常量1
        item = produce_item(); // 产生下一个数据项
        down(&empty); // 将空槽数目减一
        down(&mutex); // 进入临界区
        insert_item(item); // 将新数据项放入缓冲区中
        up(&mutex); // 离开临界区
        up(&full); // 将满槽的数目加1
    }
}

void consumer(void){

```

```

int item;

while(true){ // 无限循环
    down(&full); // 将空槽数目减1
    down(&mutex); // 进入临界区
    item = remove_item(); // 从缓冲区中取出一个数据项
    up(&mutex); // 离开临界区
    up(&empty); // 将满槽的数目加1
    consume_item(); // 打印数据项
}
}

```

互斥量

- 如果不需要信号量的计数能力，有时可以使用信号量的一个简化版本，**互斥量**
- 互斥量是一个可以处于两态之一的变量：解锁和加锁

```

mutex_lock:
    # 将互斥信号量复制到寄存器，并且将互斥信号量置为1
    # 互斥信号量是0吗？
    # 如果互斥信号量为0，它被解锁，所以返回
    # 互斥信号量忙，调度另一个线程
    # 稍后再试
    # 返回调用者，进入临界区

mutex_lock:
    # 将mutex置为0
    # 返回调用者

```

- 如何共享turn变量：
 - 共享数据结构
 - 让进程和其他进程共享其部分地址空间
- **快速用户区互斥量futex**
 - 一个futex包含两部分：
 - 一个内核服务：它提供了一个等待队列，允许多个进程在一个锁上等待
 - 一个用户库
 - 没有竞争时，futex完全在用户空间工作，避免陷入内核（代价很大的）
- **pthread中的互斥量**
 - pthread还提供了另一种同步机制：**条件变量**：允许线程由于一些未达到的条件而阻塞
 - 利用线程解决生产者-消费者问题，示例：

```

# include <stdio.h>
# include <pthread.h>
# define MAX 1000000000
pthread_mutex_t the_mutex; // 需要生产数量
pthread_cond_t condc,condp;
int buffer = 0; // 生产者消费者使用的缓冲区

```



```

void *producer(void *ptr){ // 生产数据
    int i;
    for (i = 1; i<=MAX; i++){
        pthread_mutex_lock(&the_mutex); // 互斥使用缓冲区
        while (buffer!=0) pthread_cond_wait(&condp, &the_mutex);
        buffer = i; // 将数据放入缓冲区
        pthread_cond_signal(&condc); // 唤醒消费者
        pthread_mutex_unlock(&the_mutex); // 释放缓冲区
    }
    pthread_exit(0);
}

void *consumer(void *ptr){ // 消费数据
    int i;
    for (i=1; i<=MAX; i++){
        pthread_mutex_lock(&the_mutex); // 互斥使用缓冲区
        while (buffer!=0) pthread_cond_wait(&condp, &the_mutex);
        buffer = 0; // 从缓冲区中取数据
        pthread_cond_signal(&condp); // 唤醒生产者
        pthread_mutex_unlock(&the_mutex); // 释放缓冲区
    }
    pthread_exit(0);
}

int main(int argc, char **argv){
    pthread_t pro, con;
    pthread_mutex_init(&the_mutex, 0);
    pthread_cond_init(&condc, 0);
    pthread_cond_init(&condp, 0);
    pthread_create(&con, 0, consumer, 0);
    pthread_create(&pro, 0, producer, 0);
    pthread_join(pro, 0);
    pthread_join(con, 0);
    pthread_cond_destroy(&condc);
    pthread_cond_destroy(&condp);
    pthread_mutex_destroy(&the_mutex);
}

```

管程

- 有了互斥量和信号量后并不能完全解决问题，可能会遇到死锁
- 所以提出一种高级同步原语，称为管程
- 一个管程是由一个过程、变量及数据结构等组成的一个集合，它们组成一个特殊的模块或软件包
- 特性：
 - 任意时刻管程中只能有一个活跃进程
 - 进入管程时的互斥由编译器负责，所以出错的可能性要小很多

```

// 管程
monitor example:

```

```

integer i; // 整数
condition c; // 环境
...
procedure producer(); // 生产程序
...
end;

procedure sonsumer(); // 消费程序
...
end;
end monitor;

```

- 条件变量：
 - 用来解决进程无法在运行时被阻塞，有两个操作wait和signal
- Java是一种支持用户级线程的语言，只要将关键字synchronized加入到方法声明中，Java保证一旦某个线程执行该方法，就不允许其他线程执行该对象中的任何synchronized方法。
- 管程是一种编程语言概念
- 通过将信号量放在共享内存中并用TSL或SCHG指令来保护他们，可以避免竞争
- 用Java语言实现的生产者-消费者问题的办法：

```

public class ProducerConsumer{
    static final int N= 100; // 定义缓冲区大小的常量
    static producer p = new producer(); // 初始化一个新的生产者线程
    static consumer c = new consumer(); // 初始化一个新的消费者线程
    static our_monitor mon = new our_monitor(); // 初始化一个新的管程

    public class producer void main(String args[]){
        p.start(); // 开始生产者线程
        c.start(); // 开始消费者线程
    }

    public class producer extends Thread{
        public void run(){ // run方法包含了线程代码
            int item;
            while (true) { // 生产者循环
                item = produce_item();
                mon.insert(item);
            }
        }
        private int produce_item(); // 实际生产
    }

    public class consumer extends Thread{
        public void run(){ // run方法包含了线程代码
            int item;
            while (true) { // 消费者循环
                item = mon.remove();
                consume_item(item);
            }
        }
        private int produce_item(); // 实际消费
    }
}

```

```

static class our_monitor{ // 这是一个管程
    private int buffer[] = new int [N];
    private int count = 0, lo = 0, hi = 0; // 计数器和索引

    public synchronized void insert(int val){
        if (count == N) go_to_sleep(); // 如果缓冲区满，则进入休眠
        buffer[hi] = val; // 向缓冲区中插入一个新的数据项
        hi = (hi + 1) % N; // 设置下一个数据项的槽
        count = count + 1; // 缓冲区中的数据项又多了一项
        if (count == N - 1) notify(); // 如果消费者在休眠，则将其唤醒
    }
    static synchronized int remove(){
        int val;
        if (count == 0) go_to_sleep(); // 如果缓冲区空，进入休眠
        val = buffer[lo]; // 从缓冲区中取出一个数据项
        lo = (lo + 1) % N; // 设置代取数据项的槽
        count = count - 1; // 缓冲区中的数据项数目减少1
        if (count == N - 1) notify(); // 如果生产者在休眠，则将其唤醒
        return val;
    }
    private void go_to_sleep(){
        try{
            wait();
        } catch (InterruptedException exc) {
        };
    }
}
}

```

消息传递

- `send(destination, &message);`
- `receive(source, &message);`
- 消息传递系统的设计要点：参考TCP协议，思想相似
- 信箱：是一个用来对一定数量的消息进行缓冲的地方
- 用N条消息实现生产者-消费者问题：

```

# include <iostream>

using namespace std;

# define N 100 // 缓冲区中的槽数目

void producer(void){
    int item;
    messages m; // 消息缓冲区

    while(true){
        item = produce_item(); // 产生放入缓冲区的一些数据
    }
}

```

```

        receive(consumer, &m); // 等待消费者发送空缓冲区
        bulid_message(&m, item); // 建立一个待发送的消息
        send(consumer, &m); // 发送数据项给消费者
    }
}

void consumer(void){
    int item, i;
    messages m;

    for(i = 0; i < N; i++) send(producer, &m); // 发送N个空缓冲区
    while(true){
        receive(producer, &m); // 接收包含数据项的消息
        item = extract_item(&m); // 将数据项从消息中提取出来
        send(producer, &m); // 将缓冲区发送回生产者
        consume_item(item); // 处理数据项
    }
}

```

- 如果先执行receive，则接收者会被阻塞，直到send发生，这种方法称为**会合**
- **消息传递接口**

屏障

- 除非所有的进程都就绪准备着手下一个阶段，否则任何进程都不能进入下一个阶段

避免锁：读-复制-更新

- **读-复制-更新**，**RCU**：将更新过程中的移除和再分配过程分析开来。直白点是“随意读，但更新数据的时候，需要先复制一份副本，在副本上完成修改，再一次性地替换旧数据”。
- **读端临界区**访问数据结构，它可以包含任何代码，只要该代码不阻塞或者休眠
- 定义一个任意时间段的**宽限期**：在这个时期内，每个线程至少有一次在读端临界区之外

调度

- **调度程序**和**调度算法**：操作系统管理了系统的有限资源，当有多个进程（或多个进程发出的请求）要使用这些资源时，因为资源的有限性，必须按照一定的原则选择进程（请求）来占用资源。

调度简介

- 进程行为
 - **计算密集型**：具有较长的CPU集中使用和较小频度的I/O等待
 - **I/O密集型**：具有较短的CPU集中使用和较长频度的I/O等待
 - 随着CPU变得越来越快，更多的及进程倾向为I/O密集型
- 何时调度
 - 各种情形：
 1. 在创建y一个新进程之后，需要决定是运行父进程还是子进程
 2. 在一个进程退出时必须做出调度决策
 3. 当一个进程阻塞在I/O和信号量上或由于其他原因阻塞时，必须选择另一个进程运行

4. 在一个I/O中断发生时，必须做出调度决策

- **非抢占式**：调度算法挑选一个进程，然后让该进程运行直至被阻塞，或者直到该进程自动释放CPU
 - 缺点：在时钟中断发生时，不会进程调度，在处理完时钟中断后，如果没有更高优先级的进程等待到时，则被中断的进程会继续运行
- **抢占式**：调度算法挑选一个进程，并且让该进程运行某个固定时段的最大值
- 调度算法分类
 - **批处理**：商业领域应用广泛，例：账目收入、利息计算
 - **交互式**：服务器要服务多个突发的（远程）用户
 - **实时**
- 调度算法的目标
 - 公平：
 - 系统策略的强制执行
 - 保持系统的所有部分尽可能忙碌
 - 批处理系统指标：
 - **吞吐量**：是系统每小时完成的作业数量
 - **周转时间**：指从一个批处理作业提交时刻开始直到该作业完成时刻位置的统计平均时间
 - **CPU利用率**：用于对批处理系统的度量。系统每小时可完成多少作业（吞吐量）；完成作业需要多长时间（周转时间）。
 - 把CPU利用率作为度量依据，就像用引擎每小时转动了多少次来比较汽车好坏一样。
 - 交互式系统指标：
 - **最小响应时间**：从发出命令到得到响应之间的时间
 - 均衡性

批处理系统中的调度

- **先来先服务**
 - **FCFS算法**（first come first service），非抢占式算法
 - 如果早就绪的进程排在就绪队列的前面，迟就绪的进程排在就绪队列的后面
 - 优缺点：
 - 有利于长作业以及CPU繁忙的作业
 - 简单，易于编码实现
 - 不利于短作业以及I/O繁忙的作业
- **最短作业优先**
 - **SJF算法**，（Shortest Job First），非抢占式算法，其目标是减少平均周转时间。
 - 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；
 - 对长作业非常不利，可能长时间得不到执行；
 - 缺点：
 - 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。
- **最短剩余时间优先**
 - **SRT算法**，（Shortest Remaining Time），允许比当前进程剩余时间更短的进程来抢占

交互式系统中的调度

- **轮转调度**

- 每个进程被分配一个时间段，称为**时间片**，即允许该进程在该时间段中运行
- 从一个进程切换到另一个进程是需要一定时间进行管理事务处理的--保存和装入寄存器值及内存映像、更新各种表格和列表、清除和重新调入内存高速缓存等。进程切换、上下文切换。
- 时间片设得太短会导致过多的进程切换，降低了CPU效率；而设得太长又可能引起对短的交互请求的响应时间变长。将时间片设为20~50ms通常是一个比较合理的折中。
- **优先级调度**
 - 每个进程被赋予一个优先级，允许优先级最高的可运行进程先运行
 - 低优先级可能会长时间处于饥饿状态：给每个进程赋予一个允许运行的最大时间片，当用完这个时间片时，次高优先级的进程便获得运行机会
- **多级队列**
 - 属于最高优先级类的进程运行一个时间片，属于次高优先级类的进程运行2个时间片，再次一级运行4个时间片，以此类推。当一个进程用完分配的时间片，它被移到下一类。
- **最短进程优先**
 - 根据进程过去的行为进行推测，并执行估计运行时间最短的那一个
 - 有时把这种通过当前测量值和先前估计值进行加权平均而得到下一个估计值的技术称为**老化**
- **保证调度**
 - 向用户做出明确的性能保证，然后去实现它
 - 例如：若用户工作时时有n个用户登录，则用户将获得CPU处理能力的1/n
- **彩票调度**
 - 给每个过程至少一个彩票保证它在每个调度操作中具有非零概率被选中。
 - 在此调度中，每个进程都有一些票证，并且调度程序选择随机票证和进程，该票证是赢家，并且执行时间片，然后调度程序选择另一张票证。这些票证代表了流程的份额。具有更多票数的进程使其有更多机会被选择执行。
- **公平共享调度**

实时系统中的调度

- **实时系统**是一种时间起着主导作用的系统。分为：
 - **硬实力**：必须满足绝对的截止时间
 - **软实力**：虽然不希望偶尔错失截止时间，但是可以容忍
- 按照响应方式进一步分类为**周期性**（以规则的时间间隔发生）事件或**非周期性**（发生时间不可预知）事件
- 一个系统可能要响应多个周期性事件流。根据每个事件需要处理时间的长短，系统甚至有可能无法处理完所有的事件。例如：如果有m个周期事件，事件i以周期 P_i 发生，并需要 C_i 秒CPU时间处理一个事件，那么可以处理负载的条件是 $\sum_{i=1}^m \frac{C_i}{P_i} \leq 1$ ，满足这个条件的实时系统称为是**可调度的**
- 只有在可以提前掌握所完成工作以及必须满足的截止时间等全部信息时，**静态调度**才能工作；而**动态调度算法**不需要这些限制

策略和机制

- **调度机制与调度策略**

线程调度

- 常用的是轮转调度和优先级调度

- 用户级线程和内核级线程之间的性能差别在于性能。用户级线程的线程切换需要少量的机器指令，而内核级线程需要完整的上下文切换，修改内存影像，使高速缓存失效，这导致了若干数量级的延迟。另一方面，在使用内核级线程时，一旦线程阻塞在I/O上就不需要像在用户级线程中那样将整个进程挂起。

经典的IPC问题

哲学家进餐问题

- **哲学家就餐的同步问题**：假设有五位哲学家围坐在一张圆形餐桌旁，做以下两件事情之一：吃饭，或者思考。吃东西的时候，他们就停止思考，思考的时候也停止吃东西。餐桌中间有一大碗意大利面，每两个哲学家之间有一只餐叉。因为用一只餐叉很难吃到意大利面，所以假设哲学家必须用两只餐叉吃东西。他们只能使用自己左右手边的那两只餐叉。哲学家就餐问题有时也用米饭和筷子而不是意大利面和餐叉来描述，因为很明显，吃米饭必须用两根筷子。
- **饥饿**：所有的程序都在不停的运行，但都无法取得进展
- 哲学家就餐问题的一种错误解法

```
# include <iostream>

# define N 5 // 哲学家的数目

void philosopher(int i){ // i: 哲学家编号, 从0到4
    while(true){
        think(); // 哲学家在思考
        take_fork(i); // 拿起左边叉子
        take_fork((i + 1) % N); // 拿起右边叉子,%是模运算
        eat(); // 进食
        put_fork(i); // 将左叉放回桌子上
        put_fork((i + 1) % N); // 将右叉放回桌上
    }
}
```

- 哲学家就餐问题的一个解法

```
# include <iostream>

# define N 5 // 哲学家的数目
# define LEFT (i+N-1)%N // i的左邻居的编号
# define RIGHT (i+1)%N // i的右邻居的编号
# define THINKING 0 // 哲学家在思考
# define HUNGRY 1 // 哲学家试图拿起叉子
# define EATING 2 // 哲学家进餐

typedef int semaphore; // 信号量是一种特殊的整型数据
int state[N]; // 数组用来跟踪记录每位哲学家的状态
semaphore mutex = 1; // 临界区的互斥
semaphore s[N]; // 每个哲学家一个信号量

void philosopher(int i){ // i: 哲学家编号, 从1到N-1
    while(true){ // 无限循环
```



```

    }
}
void take_forks(int i){ // i: 哲学家编号, 从1到N-1
    down(&mutex); // 进入临界区
    state[i] = HUNGRY; // 记录哲学家i处于饥饿处于饥饿状态
    test(i); // 尝试获取2把叉子
    up(&mutex); // 离开临界区
    down(&s[i]); // 如果得不到需要的叉子则阻塞
}
void put_forks(i){ // i: 哲学家编号, 从1到N-1
    down(&mutex); // 进入临界区
    state[i] = THINKING; // 哲学家已经就餐完毕
    test(LEFT); // 检查左边的邻居现在可以吃吗?
    test(RIGHT); // 检查右边的邻居现在可以吃吗?
    up(&mutex); // 离开临界区
}
void test(i){ // i: 哲学家编号, 从1到N-1
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] !=
EATING){
        state[i] = EATING;
        up(&s);
    }
}
}

```

- 注意：最多可同时有两个哲学家进餐，不是一个。

读者-写者问题

- 读者-写者问题的一种解法

```

#include <iostream>

typedef int semaphore; // 运用你的想象
semaphore mutex = 1; // 控制对rc的访问
semaphore db = 1; // 控制对数据库的访问
int rc = 0; // 正在读或者即将读的进程数目

void reader(void){
    while(true){ // 无限循环
        down(&mutex); // 获得对rc的互斥访问权
        rc = rc + 1; // 现在有多了一个读者
        if (rc == 1) down(&db); // 如果这是第一个读者
        up(&mutex); // 释放对rc的访问
        read_data_base(); // 访问数据
        down(&mutex); // 获取对rc的互斥访问
        rc = rc - 1; // 现在减少了一个读者
        if (rc == 1) up(&db); // 如果这是最后一个读者
        up(&mutex); // 释放对rc的互斥访问
        use_data_read(); // 非临界区
    }
}

```



```

}
void write(void){
    while(true){ // 无限循环
        think_up_data(); // 非临界区
        down(&db); // 获取互斥访问
        write_data_base(); // 更新数据
        up(&db); // 释放互斥访问
    }
}

```

有关进程与线程的研究

- 几乎所有系统都把进程视为一个容器，用以管理相关资源，如地址空间、线程、打开的文件、权限保护等。
- 事实上，进程、线程与调度已经不再是研究的热点，功耗管理、虚拟化、云计算、和安全问题成为了新的热点主题。

小结

- **原语**是由若干条指令组成的，用于完成一定功能的一个过程。是由若干个机器指令构成的完成某种特定功能的一段程序，具有不可分割性·即原语的执行必须是连续的，在执行过程中不允许被中断。
- 进程间通信原语可以用来解决诸如**生产者-消费者问题**、**哲学家进餐问题**、**读者-写者问题**和**睡眠理发师问题**等。
- 目前已有大量成熟调度算法：
 - 轮转调度
 - 优先级调度
 - 多级队列调度
 - 有保证调度
 - 彩票调度
 - 公平分享调度
 - ...

进程状态补充

- Linux内核源码（内核状态是7种）


```

/*
 * The task state array is a strange "bitmap" of
 * reasons to sleep. Thus "running" is zero, and
 * you can test for combinations of others with
 * simple bit tests.
 */
static const char * const task_state_array[] = {
    "R (running)",          /* 0 */
    "S (sleeping)",         /* 1 */
    "D (disk sleep)",       /* 2 */
    "T (stopped)",          /* 4 */
    "t (tracing stop)",     /* 8 */

```

```
        "X (dead)",           /* 16 */
        "Z (zombie)",         /* 32 */
};
```

- 通过ps aux可以看到进程的状态。
 - O: 进程正在处理器运行,这个状态从来没有见过.
 - S: 休眠状态 (sleeping)
 - R: 等待运行 (runable) R Running or runnable (on run queue) 进程处于运行或就绪状态
 - I: 空闲状态 (idle)
 - Z: 僵尸状态 (zombie)
 - T: 跟踪状态 (Traced)
 - B: 进程正在等待更多的内存页
 - D: 不可中断的深度睡眠, 一般由IO引起, 同步IO在做读或写操作时, cpu不能做其它事情, 只能等待, 这时进程处于这种状态, 如果程序采用异步IO, 这种状态应该就很少见到了

 进程状态转换