操作系统A

Principles of Operating System

北京大学计算机科学技术系 陈向群

Department of computer science and Technology, Peking University

2020 Autumn

Unix、Linux

进程同步、通信实例

IPC

进程同步/通信实例

UNIX

管道、消息队列、共享内存、信号量、信号、套接字

Linux

管道、消息队列、共享内存、信号量、信号、套接字

内核同步机制：原子操作、自旋锁、读写锁、信号量、屏障、BKL、RCU……

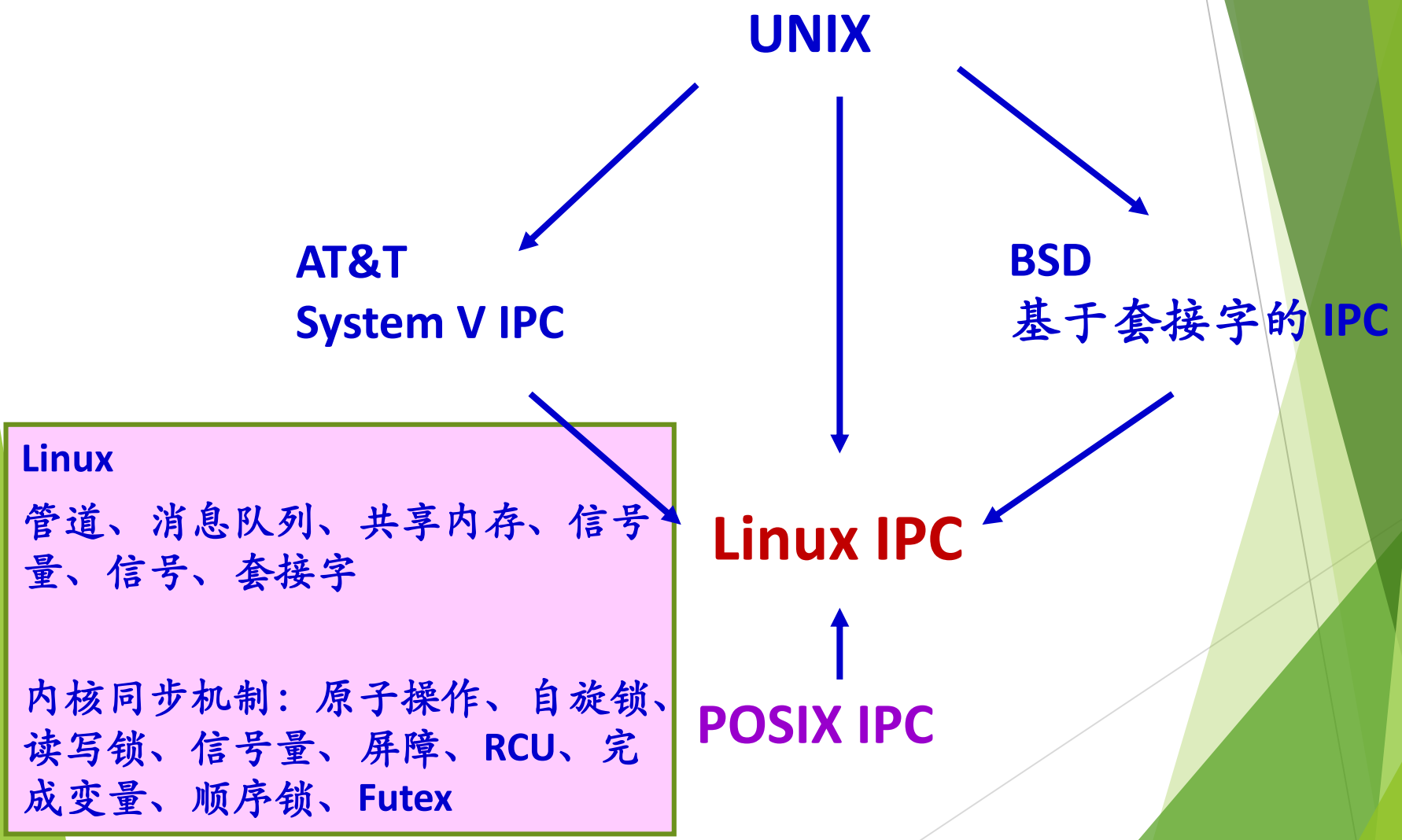
Windows

同步对象：互斥对象、事件对象、信号量对象
临界区对象
互锁变量

套接字、文件映射、管道、命名管道、邮件槽、剪贴板、动态数据交换、对象连接与嵌入、动态链接库、远程过程调用

Linux的进程通信机制

思考题：整理Linux最新版本的IPC机制



Linux

管道、消息队列、共享内存、信号量、信号、套接字

内核同步机制：原子操作、自旋锁、读写锁、信号量、屏障、RCU、完成变量、顺序锁、Futex

课堂展示——Linux的IPC机制

- 原子操作
- 管道
- 信号
- 套接字
- 信号量
- 消息队列
- 共享内存
- 自旋锁
- 读写锁
- 屏障
- 完成变量
- 顺序锁
- RCU机制
- Futex

要求：
（1）描述概念；
（2）说明作用；
（3）简单讲解在Linux中的实现；
（4）应用举例（包括代码实现）

制作ppt，发送邮件

管道（Pipeline）

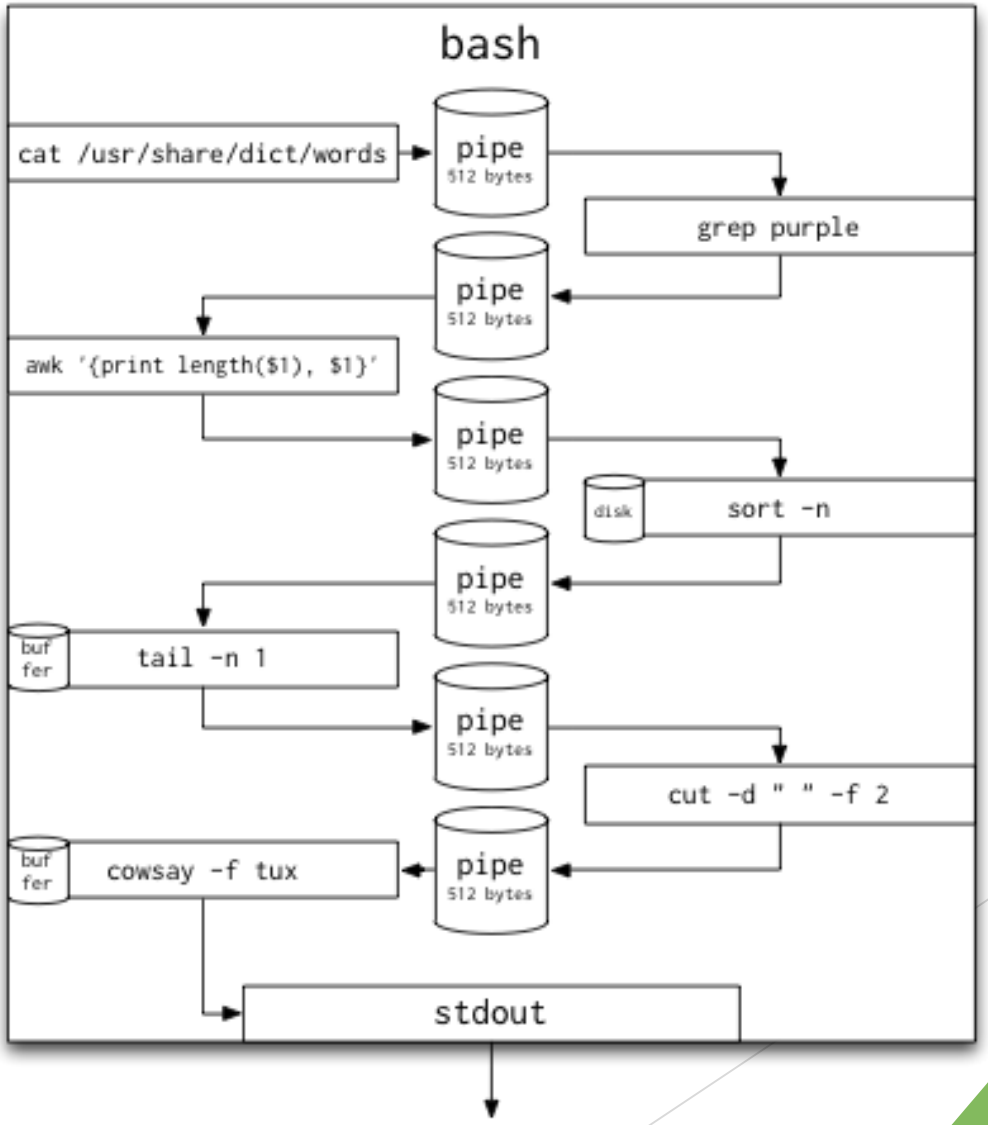
- Pipelines are an extremely useful (and surprisingly underused) architectural pattern in modern software engineering（现代软件工程中一个非常有用架构模型）
- 如果说Unix是计算机文明中最伟大的发明，那么，Unix下的Pipeline就是跟随Unix所带来的另一个伟大的发明
- 管道所要解决的问题是软件设计中的设计目标——高内聚、低耦合；它以一种“链式模型”来串接不同的程序或者不同的组件，让它们组成一条工作流；给定一个完整的输入，经过各个组件的先后协同处理，得到唯一的最终输出

一个管道应用的例子

```
cat /usr/share/dict/words |     # Read in the system's dictionary.
grep purple |                    # Find words containing 'purple'
awk '{print length($1), $1}' |  # Count the letters in each word
sort -n |                        # Sort lines ("${length} ${word}")
tail -n 1 |                     # Take the last line of the input
cut -d " " -f 2 |               # Take the second part of each line
cowsay -f tux                   # Put the resulting word into Tux's
                                 # mouth
```

```
 _____________
< unimpurpled >
 -------------
   \
    \
        .----.
       |o_o |
       | :_/  |
      //     \ \
     (|       | )
    /'\_     _/`\
    \___)=(___/
```

流程展示


bash
cat /usr/share/dict/words
pipe
512 bytes
grep purple
pipe
512 bytes
awk '{print length($1), $1}'
pipe
512 bytes
disk
sort -n
pipe
512 bytes
buf fer
tail -n 1
pipe
512 bytes
cut -d " " -f 2
buf fer
cowsay -f tux
pipe
512 bytes
stdout

管道

	无名管道	命名管道
服务对象	进程及其子孙进程	任意关系的进程
实现机制	逻辑上：管道文件 物理上：利用高速缓冲区，与外设无关	逻辑上：管道文件 物理上：依赖于文件系统实现，作为特殊文件
永久性	创建在内存中 临时存在（没有磁盘映像）	存在于文件系统中 可供以后使用（有一个磁盘i-节点）
名称	无文件名，用文件描述符存取	有文件名
主要操作	pipe() write() read()	mkfifo fopen() write() read()

信号 Signal

- 软中断信号（简称为信号）

  用来通知进程发生了异步事件

- 信号本质上是在软件层次上对中断机制的一种模拟
  - 进程之间可以互相通过系统调用发送软中断信号
  - 内核也可以因为内部事件而给进程发送信号，通知进程发生了某个事件

信号的产生

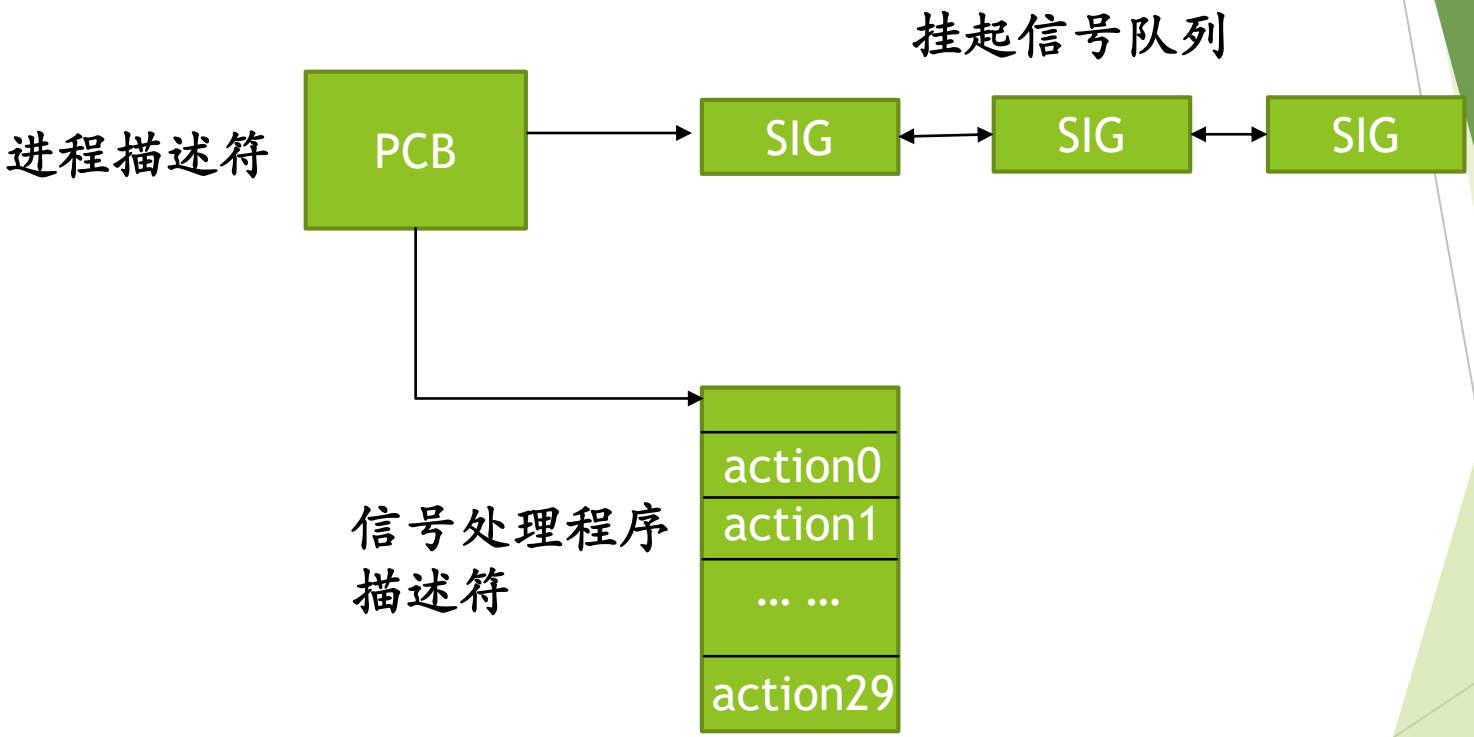
- 异常
  - 当一个进程出现异常（比如试图执行一个非法指令，除0，浮点溢出等），内核通过向进程发送一个信号来通知进程异常的发生
- 其他进程
  - 一个进程可以通过kill或是sigsend系统调用向另一个进程或一个进出组发送信号。一个进程也可以向自身发送信号
- 作业控制
  - 发送信号给那些想要读或写终端的后台进程。比如shell使用信号来管理前台和后台进程
- 通知
  - 一个进程也许要求能被通知某些事件的发生。比如设备已经就绪等待I/O操作
- 闹钟
  - 定时器产生的信号，由内核发送给进程

等等……

信号的处理

- 收到信号的进程对各种信号有三类处理方法
  - 对于需要处理的信号，进程可以指定处理函数，由该函数来处理
  - 忽略某个信号，对该信号不做任何处理，就象未发生过一样
  - 对该信号的处理保留系统的默认处理方式
- 检测并响应信号的时机
  - 进程由于系统调用、中断或异常进入内核，当该进程从内核返回用户空间前
  - 例子：进程被调度程序选中，由于检测到信号从而提前返回到用户空间

关键数据结构


挂起信号队列
进程描述符
PCB
SIG
SIG
SIG
action0
action1
… …
action29
信号处理程序
描述符

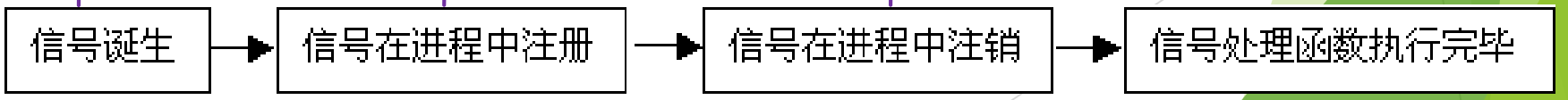
信号举例：异常

- 当程序发生除0错误或是有非法指令时，将引起一个内核态的trap
- 内核trap处理程序识别出这个异常并发送合适的信号到当前进程
  - 发送信号的本质就是将信号加入进程的挂起信号队列中
- 当trap处理程序将要返回到用户态时，内核会检查并发现信号，进而执行信号处理函数
  - 内核会检查进程中该信号的处理程序描述符，进而决定是忽略该信号、或执行默认处理程序、或执行进程自己注册的处理程序
  - 进程自己注册处理程序，实际就是修改信号处理程序描述符的内容

信号生命周期

何时?

- 信号的诞生指的是触发信号的事件发生
- 信号在进程中注册指的就是信号值加入到进程的未决信号集中
- **在目标进程执行过程中，会检测是否有信号等待处理。**如果存在未决信号等待处理且该信号没有被进程阻塞，则在运行相应的信号处理函数前，进程会把信号在未决信号链中占有的结构卸掉
- 进程注销信号后，立即执行相应的信号处理函数，执行完毕后，信号的本次发送对进程的影响彻底结束



应用实例

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

int beeps = 0;

/* SIGALRM handler */
void handler(int sig) {
  safe_printf("BEEP\n");

  if (++beeps < 5)
    alarm(1);
  else {
    safe_printf("BOOM!\n");
    exit(0);
  }
}
```

```
main() {
  signal(SIGALRM, handler);
  alarm(1); /* send SIGALRM in
               1 second */

  while (1) {
    /* handler returns here */
  }
}
```

号码	名字	默认行为	相应事件
1	SIGHUP	终止	终端线挂断
2	SIGINT	终止	来自键盘的中断
3	SIGQUIT	终止	来自键盘的退出
4	SIGILL	终止	非法指令
5	SIGTRAP	终止并转储存储器（1）	跟踪陷阱
6	SIGABRT	终止并转储存储器（1）	来自 abort 函数的终止信号
7	SIGBUS	终止	总线错误
8	SIGPFE	终止并转储存储器（1）	浮点异常
9	SIGKILL	终止（2）	杀死程序
10	SIGUSR1	终止	用户定义的信号 1
11	SIGSEGV	终止并转储存储器（1）	无效的存储器引用（段故障）
12	SIGUSR2	终止	用户定义的信号 2
13	SIGPIPE	终止	向一个没有读用户的管道做写操作
14	SIGALRM	终止	来自 alarm 函数的定时器信号
15	SIGTERM	终止	软件终止信号
16	SIGSTKFLT	终止	协处理器上的栈故障
17	SIGCHLD	忽略	一个子进程停止或者终止
18	SIGCONT	忽略	继续进程如果该进程停止
19	SIGSTOP	停止直到下一个 SIGCONT（2）	不来自终端的停止信号
20	SIGTSTP	停止直到下一个 SIGCONT	来自终端的停止信号
21	SIGTTIN	停止直到下一个 SIGCONT	后台进程从终端读
22	SIGTTOU	停止直到下一个 SIGCONT	后台进程向终端写
23	SIGURG	忽略	套接字上的紧急情况
24	SIGXCPU	终止	CPU 时间限制超出
25	SIGXFSZ	终止	文件大小限制超出
26	SIGVTALRM	终止	虚拟定时器期满
27	SIGPROF	终止	剖析定时器期满
28	SIGWINCH	忽略	窗口大小变化
29	SIGIO	终止	在某个描述符上可执行 I/O 操作
30	SIGPWR	终止	电源故障

Linux信号的实现

- signal
  - __bsd_signal
    - 函数别名，glibc/sysdeps/posix/signal.c
  - __sigaction
    - 函数调用，glibc/nptl/sigaction.c
  - __libc_sigaction
    - 函数调用，glibc/sysdeps/unix/sysv/linux/sigaction.c
  - rt_sigaction
    - 系统调用，linux/kernel/signal.c
  - do_sigaction
    - 函数调用，linux/kernel/signal.c

用户态

内核态

Linux信号的实现

- alarm
  - alarm
    - 函数定义，glibc/sysdeps/posix/alarm.c
  - setitimer
    - 系统调用，linux/kernel/compat.c
  - do_setitimer
    - 函数调用，linux/kernel/time/itimer.c
  - 开始计时……
  - it_real_fn
    - 函数回调，linux/kernel/time/itimer.c
  - kill_pid_info …… __send_signal
    - 函数调用，linux/kernel/signal.c

用户态

内核态

Linux信号的实现

- 信号处理（x86）
  - prepare_exit_to_usermode, exit_to_usermode_loop
    - 系统调用返回，linux/arch/x86/entry/common.c
  - do_signal, handle_signal
    - 处理信号并设置栈帧，linux/arch/x86/kernel/signal.c
  - iret, sysexit, sysret指令
    - 返回用户态
  - 执行signal handler
  - rt_sigreturn
    - 系统调用

内核态

用户态

内核态

Linux内核同步机制

- 原子操作
- 自旋锁
- 读写自旋锁
- 信号量
- 读写信号量
- 互斥体
- 完成变量
- 顺序锁
- 屏障
- RCU

原子操作(1)

- 不可分割，在执行完之前不会被其他任务或事件中断
- 常用于实现资源的引用计数
- atomic_t

```
typedef struct { volatile int counter; } atomic_t;

原子操作API包括:
atomic_read(atomic_t * v);
atomic_set(atomic_t * v, int i);
void atomic_add(int i, atomic_t *v);
int atomic_sub_and_test(int i, atomic_t *v);
void atomic_inc(atomic_t *v);
int atomic_add_return(int i, atomic_t *v);
```

原子操作(2)

- 例子

```
atomic_t   v = atomic_init(0);

atomic_set(&v, 4);
atomic_add(2, &v);
atomic_inc(&v);
printk("%d\n", atomic_read(&v));


int atomic_dec_and_test(atomic_t  *v)
```

Linux同步机制 原子操作

原子操作演示

```
atomic_t my_counter ATOMIC_INIT(0);
atomic_add( 1, &my_counter );
atomic_inc( &my_counter );
atomic_sub( 1, &my_counter );
atomic_dec( &my_counter );
```

1 include/linux/types.h

```
typedef struct {
    int counter;
} atomic_t;
```

2 设置 加减 等实现 与具体的arch相关 arch/x86/include/asm/atomic.h

```
static inline void atomic_add(int i, atomic_t *v)
{
    asm volatile(LOCK_PREFIX "addl %1,%0"
            : "+m" (v->counter)
            : "ir" (i));
}
```

Linux同步机制 自旋锁

- 为防止多处理器并发而引入的一种锁
- 在内核中大量应用于中断处理等部分

  ？在单CPU上如何解决中断处理过程中的并发?
- 自旋锁最多只能由一个内核任务持有，如果一个内核任务试图请求一个已被占用的自旋锁，这个任务会一直忙等待（旋转）等待锁重新可用
- 可以在任何时刻防止多于一个的内核任务同时进入临界区，因此这种锁可有效地避免多处理器上并发运行的内核任务竞争共享资源

自旋锁使用例子

```
#include <linux/spinlock.h>   //自旋锁声明
spinlock_t mylock = SPIN_LOCK_UNLOCKED; /* 初始化 */

/* 获取自旋锁 */
spin_lock(&mylock);

/* ... 临界区代码 ... */

spin_unlock(&mylock); /* 释放锁 */
```

传统自旋锁的实现

spinlock_t类型定义

```
typedef struct arch_spinlock {
    unsigned int slock;
} arch_spinlock_t;
```

```
static inline void
__raw_spin_unlock(raw_spinlock_t *lock)
{
asm volatile("movb $1,%0" : "+m" (lock-
>slock) :: "memory");
}
```

```
static inline void
__raw_spin_lock(raw_spinlock_t *lock)
{
asm volatile("\n1:\t"
             LOCK_PREFIX " ; decb %0\n\t"
             "jns 3f\n"
             "2:\t"
             "rep;nop\n\t"
             "cmpb $0,%0\n\t"
             "jle 2b\n\t"
             "jmp 1b\n"
             "3:\n\t"
             : "+m" (lock->slock) : : "memory");

}
```

这样实现有什么问题?

排队自旋锁

类似银行营业厅排队票号系统

排队机票号

柜台票号

客户

客户票号

由 Linux 内核开发者
Nick Piggin 实现

排队机票号	柜台票号

```
typedef struct arch_spinlock {
    unsigned int slock;
} arch_spinlock_t;
```

```
static inline void __raw_spin_lock(raw_spinlock_t *lock)
{
 short inc = 0x0100;  //客户票据 高8位表示自己票号，
         // 低8位表示查看到的柜台票号，低8位在忙等待过程中会持续更新
 __asm__ __volatile__ (
 LOCK_PREFIX "xaddw %w0, %1\n"//客户获得票据，以及当前柜台的票号；
                              //排队机票号加1
 "1:\t"
 "cmpb %h0, %b0\n\t" //客户比较自己票号与柜台票号
 "je 2f\n\t"  //如果相等，说明空闲可用，就去办理业务
 "rep ; nop\n\t" //如果不相等，则空等一个指令
 "movb %1, %b0\n\t" //获取柜台最新票号
 "jmp 1b\n" //跳转到标签为1的位置，准备再一次检查能否获得锁
  "2:"
 :"+Q" (inc), "+m" (lock->slock)
 :
 :"memory", "cc");
}
```

```
static inline void
__raw_spin_unlock(raw_spinlock_t *lock)
{
 __asm__ __volatile__(

//办理完业务后 将柜台票号加1
   UNLOCK_LOCK_PREFIX "incb %0"
  :"+m" (lock->slock)
 :
 :"memory", "cc");
}
```

Linux同步机制 信号量

semaphore 类型定义

```
struct semaphore {
        spinlock_t              lock;
        unsigned int            count;
        struct list_head        wait_list;
};
```

semaphore 主要操作

```
extern void down(struct semaphore *sem);
extern int __must_check down_interruptible(struct semaphore *sem);
extern int __must_check down_killable(struct semaphore *sem);
extern int __must_check down_trylock(struct semaphore *sem);
extern int __must_check down_timeout(struct semaphore *sem, long jiffies);
extern void up(struct semaphore *sem);
```

Linux同步机制 读写锁

- 应用场景

  如果每个执行实体对临界区的访问或者是读或者是写共享数据结构，但是它们都不会同时进行读和写操作，读写锁是最好的选择

- 实例：Linux的IPX路由代码使用了读-写锁，用ipx_routes_lock的读-写锁保护IPX路由表的并发访问

要通过查找路由表实现包转发的执行单元需要请求读锁；需要添加和删除路由表中入口的执行单元必须获取写锁（由于通过读路由表的情况比更新路由表的情况多得多，使用读-写锁提高了性能）

读写锁使用

```
rwlock_t myrwlock = RW_LOCK_UNLOCKED;

read_lock(&myrwlock);     /* 获取读锁 */
/* ... 临界区代码 ... */
read_unlock(&myrwlock);   /* 释放读锁 */
```

```
rwlock_t myrwlock = RW_LOCK_UNLOCKED;

write_lock(&myrwlock);    /* 获取写锁 */
/* ... 临界区代码 ... */
write_unlock(&myrwlock);  /* 释放写锁 */
```

RCU机制(Read-Copy Update)

- Linux 2.6之后，一种数据一致性访问机制
- 对于读操作
  - ✓ 直接对共享资源进行访问（需要CPU支持访存操作的原子化）
  - ✓ 采用read_rcu_lock()，RCU的读操作上下文是不可抢占的
- 对于写操作
  - ✓ 将原来的老数据作一次备份（copy），然后对备份数据进行修改，修改完毕之后再用新数据更新老数据，更新老数据时采用了rcu_assign_pointer()宏，之后，需要进行老数据资源的回收
  - ✓ 采用数据备份的方法可以实现读者与写者之间的并发操作，但是不能解决多个写者之间的同步，所以当存在多个写者时，需要通过锁机制对其进行互斥，也就是在同一时刻只能存在一个写者
- 读操作几乎没有开销，写操作开销比较大，适用于读操作很多、写操作极少的场景

Linux同步机制 完成变量

Completion variable

如果内核中一个进程需要发出信号通知另一进程发生了某特定事件，利用完成变量是使两进程得以同步的简单方法

与信号量思想类似

方法	描述
init_completion(struct completion *)	初始化指定的动态创建的完成变量
wait_for_completion(struct completion *)	等待指定的完成变量接收信号
complete(struct completion *)	发信号唤醒等待进程

完成变量在vfork实现中的应用

```
//当定义了CLONE_VFORK，会实现vfork的功能
  do_fork(...){
    struct task_struct *p; //声明子进程
    ...
    p = copy_process(...);//复制父进程，初始化子进程基本结构
    ...
    struct completion vfork;//声明完成变量vfork
    if(clone_flags & CLONE_VFORK){
        init_completion(&vfork);//初始化vfork
        p->vfork_done = &vfork; //设置子进程vfork_done
    }
    ...
    wake_up_new_task(p); //
    ...
    if(clone_flags & CLONE_VFORK){
        wait_for_completion(&vfork); //等待子进程完成
        ...
    }
}
```

Linux3.0
源代码

```
void mm_release(struct task_struct *tsk,
                struct mm_struct **m )
 {
  struct completion *vfork_done = tsk->vfork_done;
  ...
 if(vfork_done){
    tsk->vfork_done = NULL;
    complete(vfork_done);//调用complete()以唤醒父进程
}
```

Linux同步机制 顺序锁

- 顺序锁用于读写共享数据
  - 这种锁依靠一个序列计数器
  - 写时：会得到一个锁，序列值增加
  - 读时：读前读后都读取序列值，若相同，则成功读
- 使用场景
  - 读者很多，写者很少
  - 虽然写者少，但希望写优先于读，而且不允许读者让写者饥饿
  - 操作的数据很简单

顺序锁 在jiffies_64上的应用

jiffies_64用来存储Linux机器启动到当前的总时间

```
//读取jiffies_64
 unsigned long long get_jiffies_64(void)
 {    unsigned long seq;
      unsigned long long ret;
      do {
           seq = read_seqbegin(&xtime_lock); //准备读，返回顺序锁的顺序号
           ret = jiffies_64;          //读取数据
      } while (read_seqretry(&xime_lock, seq)); //检查读的过程中有没有写者访问了共享资源
                          //如果有 则进入循环重新读
                          //如果成功完成读操作
 return ret; }
```

```
//修改jiffies_64
 write_seqlock(&xtime_lock );//加顺序锁
 ++jiffies_64;  //修改数据
write_sequnlock( &xtime_lock); //解锁
```

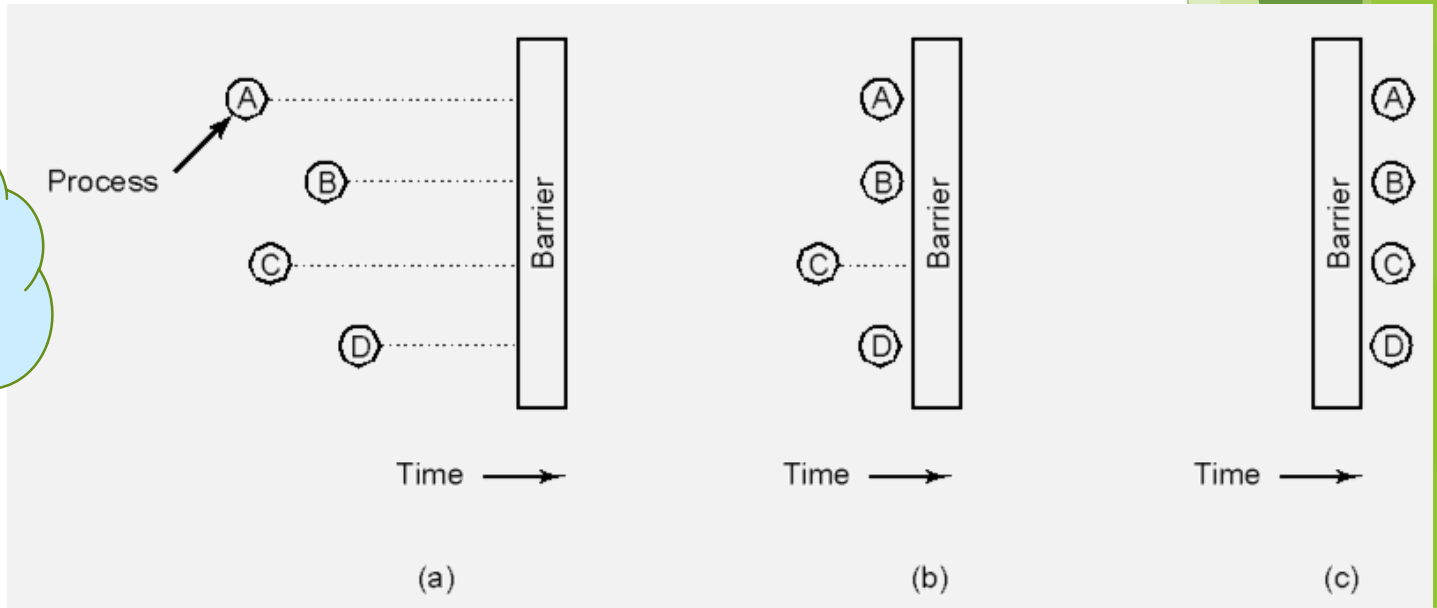
Linux同步机制 屏障(Barrier)

矩阵运算

- 另一种同步机制(又称栅栏、关卡)
- 用于对一组线程进行协调
- 应用场景

  一组线程协同完成一项任务，需要所有线程都到达一个汇合点后再一起向前推进

思考题：
如何用P、V操作实现?



Thanks

The End