



北京大学  
PEKING UNIVERSITY

# XV6同步机制 源码阅读报告

罗登

2020年10月31日

# 问题回答



## Q1: 什么是临界区?

### 临界资源

多个进程可以共享系统中的各种资源，但其中有些资源，一次只能被一个进程使用。

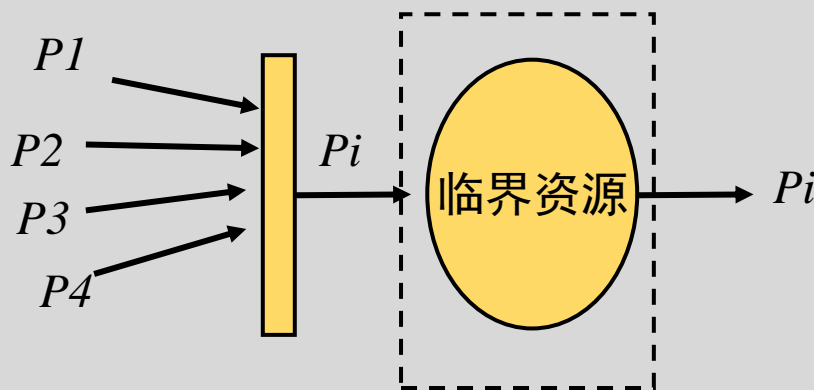
一次只能被一个进程资源称为临界资源。

许多物理设备都属于临界资源，如打印机。内存中的一些变量、数据等可以被若干进程共享，也属于临界资源。

对于临界资源的访问，必须**互斥**（Mutual Exclusion）地进行，访问临界资源的那段代码称为**临界区**。

### 临界区

```
do {  
    entry section;    // 进入区  
    critical section; // 临界区  
    exit section;     // 退出区  
    remainder section; // 剩余区  
} while(true);
```



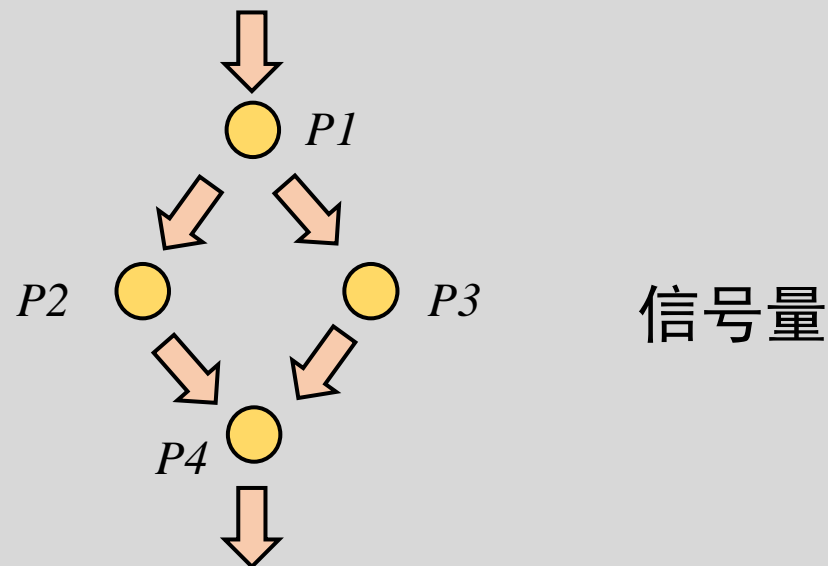
- 进入区。为了进入临界区使用临界资源，在进入区要检查是否可以进入临界区，若能进入临界区，则应设置正在访问临界区的标识，以阻止其他进程同时进入临界区。
- 临界区。进程中访问临界资源的那段代码，又称为临界段。
- 退出区。将正在访问临界区的标志清除。
- 剩余区。代码中的剩余部分。

## Q2: 什么是同步和互斥?

### 同步 synchronization

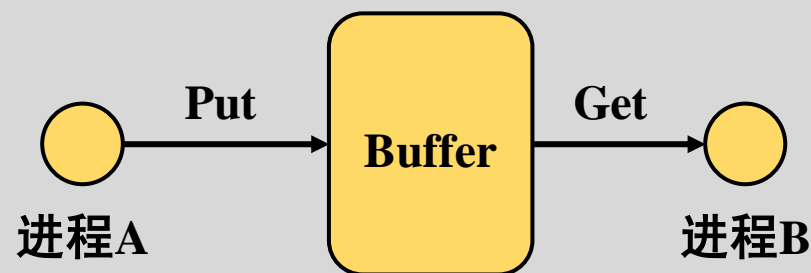
同步也被称为**直接制约关系**，是指为完成某种任务而建立的两个或多个进程，这些进程因为需要在某些位置上协调它们的工作次序而等待、传递消息所产生的制约关系。进程间的直接制约关系源于它们之间的互相合作。

例如，输入进程A通过单缓冲区向进程B提供数据。当该缓冲区为空时，进程B不同获得所需数据而阻塞，一旦进程A将数据送入缓冲区，进程B就被唤醒。反之，当缓冲区满时，进程A被阻塞，仅当进程B取走缓冲数据时，才唤醒进程A。



信号量

同步关系（进程前驱图）



## Q2: 什么是同步和互斥?

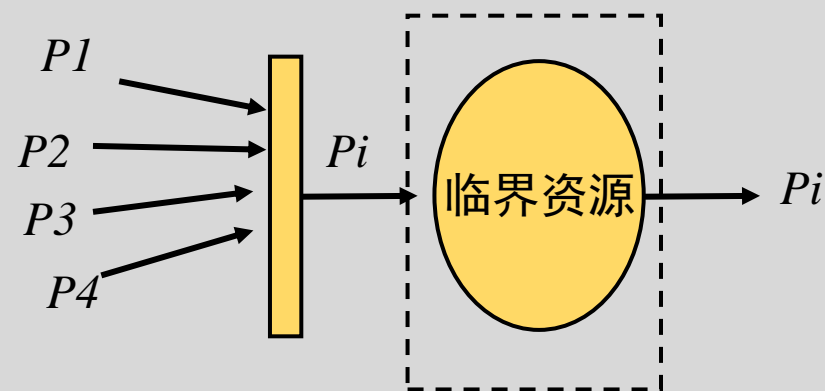
### 互斥 Mutual Exclusion

互斥也被称为间接制约关系。当一个进程进入临界区使用临界资源的时候，另外一个进程必须等待，当占用临界资源的进程退出临界区后，另外一个进程才被允许去访问该临界资源。

互斥是对临界资源来说的，临界资源互斥访问。

锁：将对临界区的争用抽象成对锁的争用。

### 管理临界区的原则



- 空闲让进。临界区空闲时，可以允许一个请求进入临界区的进程立即进入临界区。
- 忙则等待。当已有进程进入临界区时，其他试图进入临界区的进程必须等待。
- 有限等待。对于请求访问的进程，应保证能在有限时间内进入临界区。
- 让权等待。当进程不能进入临界区时，应立即释放处理器，防止进程忙等待。

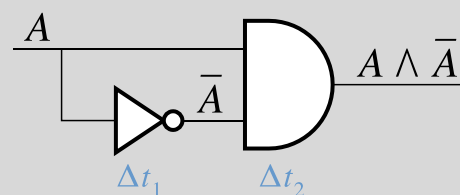
## Q3: 什么竞争状态?

### 竞态 (Race Condition)

竞争冒险/竞态条件，旨在描述一个系统或进程的输出依赖于**不受控制的事件出现顺序**，或者**出现时机**。

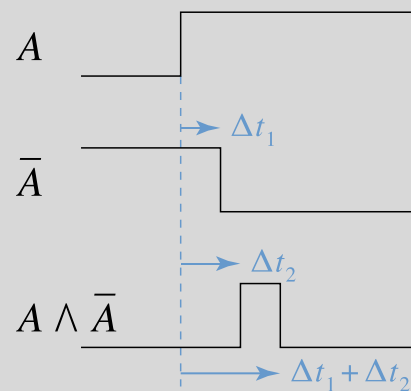
例子，两个进程试图同时修改一个共享内存的内容，在没有并发控制的情况下，最后的结果依赖于两个进程的**执行顺序与时机**，而这是不可控的（异步性）。因此结果可能是不正确的。

竞争冒险常见于不良设计的电子系统中，以及未合理设计并发控制的多线程程序。

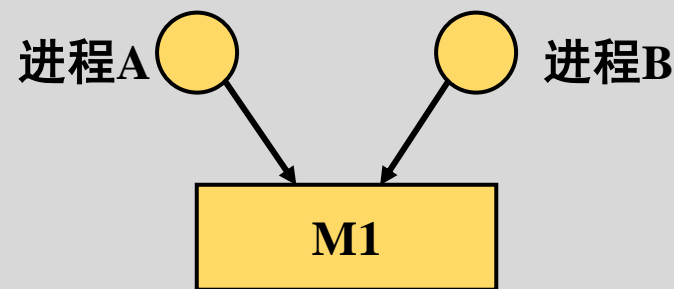


A -> M1

B -> M1



电路上的竞态



- 软件实现方法
  - 单标志法。必须交替进入，违背“空闲让进”
  - 双标志先检查。不必交替进入，可能同时进入，违背“忙着等待”
  - 双标志后检查。互相谦让，造成饥饿
  - *Peterson's Algorithm*
- 硬件实现方法
  - 中断屏蔽方法。进入临界区前关中断，访问结束后开中断。效率不高
  - 硬件指令方法。TestAndSet指令/Swap指令。锁





## Q4: 临界区操作时中断是否应该开启?

### 关中断机制

如果使用了关中断机制，当一个进程在访问临界区代码的使用，是关闭了整个系统的中断，这样就确保了在临界区内不会发生进程切换，保证了当前运行的进程可以在临界区执行完毕，确保了互斥的正确实现。

- 效率低下。限制了处理机交替执行程序的能力，因此整个系统的执行效率会显著降低。
- 不安全。对于内核来说，在更新变量或者列表中的几条指令期间，关中断是很方便的，但是将关中断的权利交给用户是很不明智的，如果一个用户进程关中断后不再开启，系统可能会因此终止。

```
...  
关中断;  
临界区;  
开中断;  
...
```

**在xv6中，获取锁的时候关闭了系统中断。**



## 关于自旋锁

这玩意儿跟物理的自旋态没什么关系.....  
可以联想抢厕所.....

- 自旋锁是互斥锁的一种实现，但是普通的互斥锁一般会在尝试失败后阻塞，引起上下文切换。而使用自旋锁的进程不会阻塞自己，而是不断地循环尝试获取
- 自旋锁要求临界区的代码比较短
- 自旋锁只适用于多核情况

获取、释放自旋锁，是通过读写自旋锁的存储内存或寄存器。因此这种读写操作必须是原子的。通常用testAndSet等原子操作来实现。

自旋锁是计算机科学用于多线程同步的一种锁，线程反复检查锁变量是否可用。由于线程在这一过程中保持执行，因此是一种忙等待。一旦获取了自旋锁，线程会一直保持该锁，直至显式释放自旋锁。

自旋锁避免了进程上下文的调度开销，因此对于线程只会阻塞很短时间的场合是有效的。

显然，单核单线程的CPU不适于使用自旋锁，因为，在同一时间只有一个线程是处在运行状态，假设运行线程A发现无法获取锁，只能等待解锁，但因为A自身不挂起，所以那个持有锁的线程B没有办法进入运行状态，只能等到操作系统分给A的时间片用完，才能有机会被调度。这种情况下使用自旋锁的代价很高。

# XV6中的实现



## Q5: xv6中锁的实现

### xv6中实现了自旋锁spinlock

```
// spinlock.h
struct spinlock
{
    uint locked; // Is the lock held? 1表示持有, 0表示未持有, 汇编修改

    // For debugging: 用于调试的信息
    char *name;      // Name of lock.
    struct cpu *cpu; // The cpu holding the lock.
    uint pcs[10];    // The call stack (an array of program counters)
                      // that locked the lock.
};
```

```
// spinlock.c
void initlock(struct spinlock *lk, char *name); // 初始化
void acquire(struct spinlock *lk);              // 获取
void release(struct spinlock *lk);              // 释放
int holding(struct spinlock *lock);             // 查看当前cpu释放持有
```

```
void initlock(struct spinlock *lk, char *name){
    lk->name = name;
    lk->locked = 0;
    lk->cpu = 0;
}
```

```
void pushcli(void){
    int eflags;
    eflags = readeflags();
    cli();
    if (mycpu()->ncli == 0)
        mycpu()->intena = eflags & FL_IF;
    mycpu()->ncli += 1;
}
```

```
void popcli(void) {
    if (readeflags() & FL_IF)
        panic("popcli - interruptible");
    if (--mycpu()->ncli < 0)
        panic("popcli");
    if (mycpu()->ncli == 0 && mycpu()->intena)
        sti();
}
```

```
// x86.h
static inline uint
readeflags(void)
{
    uint eflags;
    asm volatile("pushfl; popl %0" :
        "=r" (eflags));
    return eflags;
}
pushfl; popl eflags;
```

readeflags() & FL\_IF 1表示中断开启

禁止中断

开启中断

```
// x86.h
static inline void
cli(void) {
    asm volatile("cli");
}
```

```
// mmu.h  Eflags register
#define FL_IF  0x00000200
```

<https://blog.csdn.net/zang141588761/article/details/52325106>

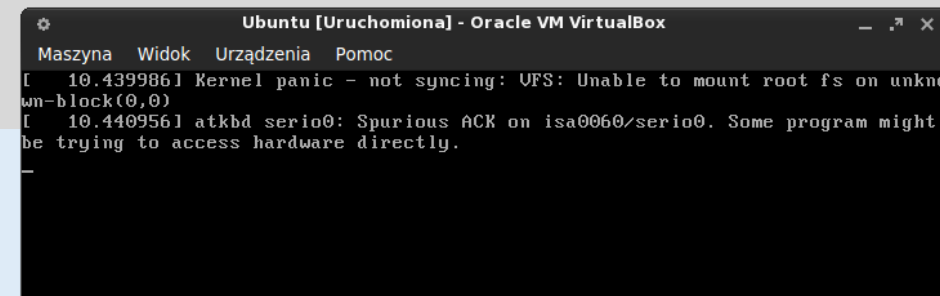
X86 PUSHF/PUSHFD/PUSHFQ 指令详解 <https://blog.csdn.net/ross1206/article/details/72833084>

# Kernel Panic



- 操作系统采用的一种安全措施，来检测内部致命错误（可能导致系统无法恢复或者无法继续运行下去）
- Unix或类Unix系统中，也叫Fatal system error
- Windows叫做Stop error，俗称蓝屏死机

```
// console.c
void panic(char *s) {
    int i;
    uint pcs[10];
    cli();
    cons.locking = 0;
    // use lapiccpunum so that we can call panic from mycpu()
    cprintf("lapicid %d: panic: ", lapicid());
    cprintf(s);
    cprintf("\n");
    getcallerpcs(&s, pcs);
    for(i=0; i<10; i++)
        cprintf(" %p", pcs[i]);
    panicked = 1; // freeze other CPU
    for(;;) ;
}
```



## mycpu()



- xv6支持多CPU，使用mycpu()函数标识当前进程运行所在的CPU
- struct cpu结构体记录当前CPU的状态，cpus为存储CPU状态的数组

```
// proc.h
struct cpu
{
    uchar apicid;           // Local APIC ID --- CUID CUID, 通过lapicid()函数获得
    https://en.wikipedia.org/wiki/CUID
    struct context *scheduler; // swtch() here to enter scheduler
    struct taskstate ts;       // Used by x86 to find stack for interrupt
    struct segdesc gdt[NSEGS]; // x86 global descriptor table
    volatile uint started;     // Has the CPU started?
    int ncli;                  // Depth of pushcli nesting. 中断调用深度
    int intena;                // Were interrupts enabled before pushcli? 调用之前是否开启
    struct proc *proc;        // The process running on this cpu or null
};
extern struct cpu cpus[NCPU]; //当前系统中存在的CPU
extern int ncpu;
```



## mycpu()

- 调用前必须关中断

```
// proc.c
struct cpu* mycpu(void)
{
```

```
    int apicid, i;
```

中断开启，进入panic

APIC(Advanced Programmable Interrupt Controller)

控制多处理器中的中断信息传递

```
    if(readeflags() & FL_IF)
```

```
        panic("mycpu called with interrupts enabled\n");
```

获取CPLID，定义在lapic.c文件中

```
    apicid = lapicid();
```

```
    // APIC IDs are not guaranteed to be contiguous. Maybe we should have
```

```
    // a reverse map, or reserve a register to store &cpus[i].
```

```
    for (i = 0; i < ncpu; ++i) {
```

```
        if (cpus[i].apicid == apicid)
```

遍历cpus数组，返回其值

```
            return &cpus[i];
```

```
    }
```

```
    panic("unknown apicid\n");
```

```
}
```





## xchg函数

- 内部调用xchg指令

```
static inline uint
xchg(volatile uint *addr, uint newval)
{
    uint result;

    // The + in "+m" denotes a read-modify-write operand.
    asm volatile("lock; xchgl %0, %1"
                 : "+m"(*addr), "=a"(result)
                 : "1"(newval)
                 : "cc");
    return result;
}
```

输入的是地址

告诉编译器内存被修改了

调用了lock指令

lock指令锁的是什么？

锁的是内存。在多处理器系统中，各个处理器独立运行，单条指令操作可能会被干扰。锁定一个特定内存地址，防止其他处理器读取或修改该地址的内容。

# xchg指令



- 双操作数指令，交换两个操作数的内容。有以下三种形式

```
XCHG reg, reg ; 寄存器<->寄存器  
XCHG reg, mem ; 寄存器<->内存  
XCHG mem, reg ; 内存寄<->存器
```

- 要求两边的类型（长度）要相同

```
xchg ax,bx      ;交换 16 位寄存器内容  
xchg ah,al      ;交换 8 位寄存器内容  
xchg var1,bx    ;交换 16 位内存操作数与 BX 寄存器内容  
xchg eax,ebx    ;交换 32 位寄存器内容
```

- 不能都为内存操作数
- 任何一个操作数都不能为段寄存器
- 任何一个操作数都不能为立即数
- 两个操作数的长度不能不相等

是否有原子性？

<https://juejin.im/post/6844904147129466893>

<https://baike.baidu.com/item/XCHG/1589577>

<https://blog.csdn.net/ross1206/article/details/72852609>

# 获取锁

```
void acquire(struct spinlock *lk) {  
    pushcli();  
    if (holding(lk))  
        panic("acquire");  
    while (xchg(&lk->locked, 1) != 0);  
  
    __sync_synchronize();  
  
    lk->cpu = mycpu();  
    getcallerpcs(&lk, lk->pcs);  
}
```

关中断

如果当前已经持有，系统panic

自旋等待

内存屏障。告诉C编译器和处理器不要在此时访问内存（load/store），确保临界区的内存引用是在锁获取之后的

记录锁获取信息，方便调试

**xchg**函数不断让lk->locked与1交换，result值存储的是lk->locked的原始值，如果原始为1，说明已被上锁，循环继续。  
如果result为0，说明之前没有上锁，成功获取到锁，向下执行

# 释放锁

```
void release(struct spinlock *lk)
{
    if (!holding(lk))
        panic("release");
    lk->pcs[0] = 0;
    lk->cpu = 0;
    __sync_synchronize();
    asm volatile("movl $0, %0"
                  : "+m"(lk->locked)
                  :);
    popcli();
}
```

如果当前未持有，系统panic


恢复锁的状态

内存屏障。告诉C编译器和处理器不要在此时访问内存（load/store），确保临界区的内存引用是在锁获取之后的

内联汇编实现赋值，确保原子性

开启中断

## getcallerpcs

```
void getcallerpcs(void *v, uint pcs[])
{
    uint *ebp;  基地址寄存器
    int i;

    ebp = (uint *)v - 2;
    for (i = 0; i < 10; i++)
    {
        if (ebp == 0 || ebp < (uint *)KERNBASE || ebp == (uint *)0xffffffff)
            break;
        pcs[i] = ebp[1]; // saved %eip    eip保存到pcs[i]中, eip就是pc
        ebp = (uint *)ebp[0]; // saved %ebp    更新ebp
    }
    for (; i < 10; i++)
        pcs[i] = 0;
}
```

memlayout.h  
0x80000000。内核的虚拟地址

# 使用自旋锁设计实现 信号量、读写锁、信号机制

# 信号量



- 信号量是一种较强的进程同步机制，可以用来解决同步和互斥的问题。
- 考虑有多种临界资源的情况，信号量的核心是维护一个计数器，表示当前可用的资源数量。
- 提供两个操作P操作和V操作，如果当前信号量为正，表示资源可用，P操作将信号量减一。如果进程使用临界资源完毕，使用V操作将信号量加一。
- 与自旋等待不同，信号量机制中当资源不够的情况，进程一般选择阻塞自身，因此还要维护一个等待队列。

```
struct semaphore {  
    int value;  
    struct spinlock lock;  
    struct proc *queue[NPROC];  
    int end;  
    int start;  
};
```

```
void sem_init(struct semaphore *s, int value) {
    s->value = value;
    initlock(&s->lock, "semaphore_lock");
    end = start = 0;
}
```

初始化自旋锁

```
void sem_wait(struct semaphore *s) {
    acquire(&s->lock);
    s->value--;
    if (s->value < 0) {
        s->queue[s->end] = myproc();
        s->end = (s->end + 1) % NPROC;
        sleep(myproc(), &s->lock);
    }
    release(&s->lock);
}
```

入队阻塞

s->lock的作用是什么?  
像是保护了value的互斥访问,  
有没有什么问题?

出队

```
void sem_signal(struct semaphore *s) {
    acquire(&s->lock);
    s->value++;
    if (s->value <= 0) { // 还有等待
        wakeup(s->queue[s->start]);
        s->queue[s->start] = 0;
        s->start = (s->start + 1) % NPROC;
    }
    release(&s->lock);
}
```



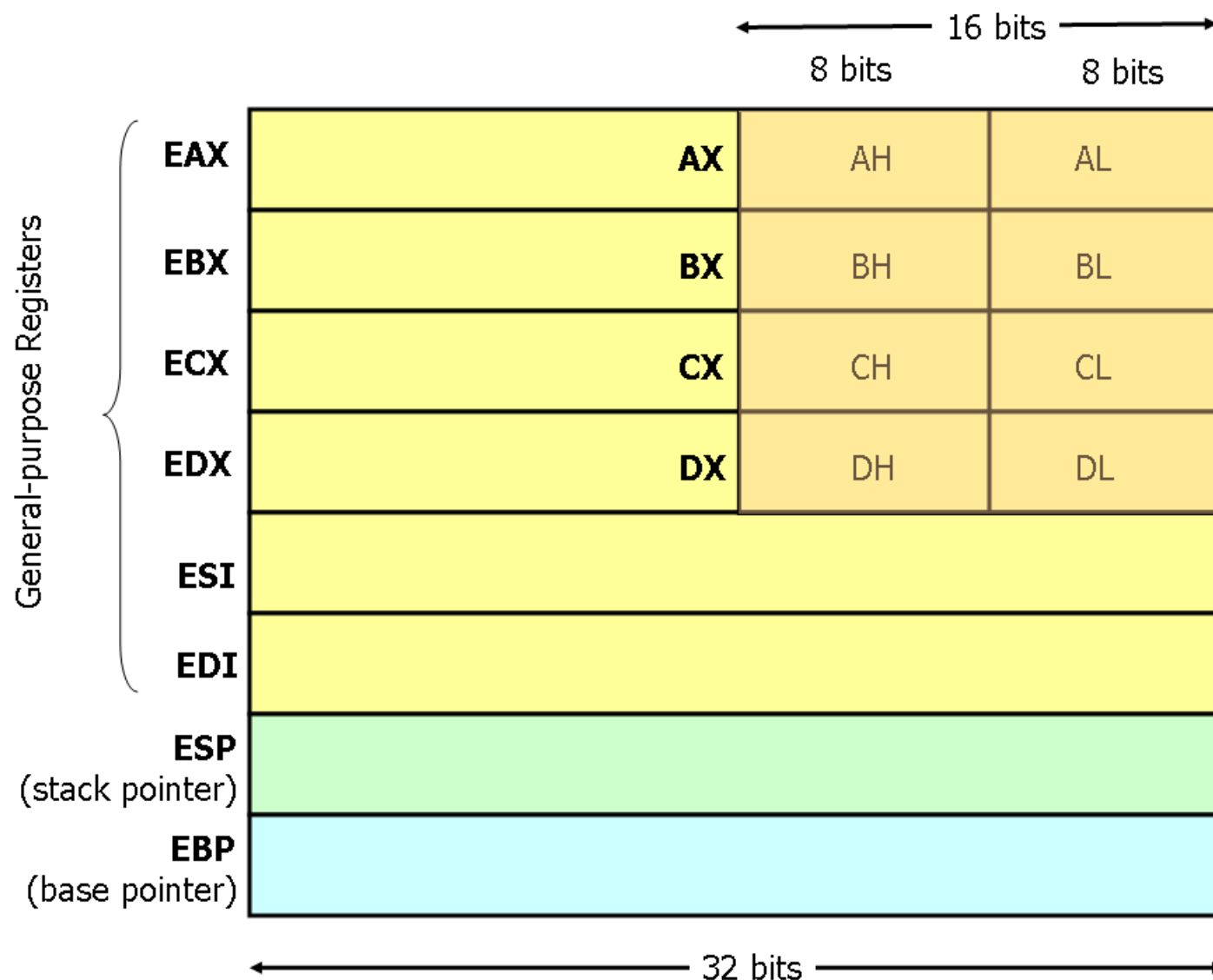
# 读写锁



- 计算机程序中并发控制的一种同步机制，也称共享-互斥锁、多读者-单写者锁
- 用于解决读者-写者问题
  - 多个读者进程和多个写者进程共享同一个文件
  - 允许多个读者进程对文件进行读操作
  - 只允许一个写者进程向文件中写信息
  - 任一写者在完成操作之前都不运行其他读者进程或写者进程访问该文件
  - 写者执行写操作前，应该等待已有的读者进程和写者进程全部退出
- 通常用互斥锁、条件变量、信号量实现
- 读写锁可以有不同的操作模式优先级
  - 读操作优先：允许最大并发，但写操作可能饿死
  - 写操作优先：一旦所有已经开始的读操作完成，等待的写者立即获得锁。内部实现需要两把互斥锁
  - 未指定优先级（公平？）

一些基础知识

# x86中的常用寄存器



带E是指32位机器上的，  
不带E是16位机器上使用，  
基本功能和用法类似。

# x86中的常用寄存器



- 通用寄存器

- EAX(Extend Add)。累加器，在乘法和除法指令中自动使用；Win32中，一般用在函数的返回值中。  
EBX(Extend Base)。基地址寄存器，DS（数据段）中的数据指针。
- ECX (Extend Count)。计数器，CPU自动使用ECX作为循环计数器，在循环指令（LOOP）或串操作中，ECX用来进行循环计数，每执行一次循环，ECX都会被CPU自动减一。
- EDX(Extend Data)。数据寄存器。
- ESI (Extend Source Index)。源变址寄存器，字符串操作源指针。
- EDI (Extend Destination Index)。目的变址寄存器，字符串操作目标指针。
- EBP (Extend Base Pointer)。基地址指针寄存器，SS（堆栈段）中数据指针。
- ESP (Extend Stack Pointer)。堆栈指针寄存器，SS（堆栈段）中堆栈指针，ESP用来寻址堆栈上的数据，E一般不参与算数运算。

通用寄存器是通用了，可以不限于上述用途，用来存储别的东西。

# x86中的常用寄存器



北京大学  
PEKING UNIVERSITY

- 指令指针寄存器
  - **EIP (Instruction Pointer)**。32位宽，CPU指令寻址，引导CPU的执行；CPU调用后自动修改；不能够直接修改这个寄存器的值，通过影响EIP的指令间接修改，跳转或分支指令（JMP, JE, LOOP, CALL等）。
- 段寄存器（保护模式下为段选择器）
  - CS（Code Segment） 代码段，或代码选择器。used for **IP/EIP**，CS:IP存储了CPU要执行的下一条指令
  - DS（Data Segment） 数据段，或数据选择器。used for MOV
  - ES（Extra Segment） 附加段，或附加选择器。used for MOVS, etc.
  - SS（Stack Segment） 堆栈段或堆栈选择器。used for SP
  - FS
  - GS

带E是指在32位机器上的，不带E是16位机器上使用，基本功能和用法类似。

[https://blog.csdn.net/qq\\_24423085/article/details/103835522](https://blog.csdn.net/qq_24423085/article/details/103835522)

<https://stackoverflow.com/questions/10810203/what-is-the-fs-gs-register-intended-for>



## x86中的常用寄存器

- 标志寄存器
  - 16位CPU中，标志寄存器称为 FLAGS 或者 PSW (Program Status Word)
  - 32位CPU中，标志寄存器被扩展成32位，称为 **EFLAGS** (Extend FLAGS)。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				OF	DF	IF	TF	SF	ZF		AF		PF		CF
				溢出	方向	中断	陷阱	符号	零		辅助进位		奇偶		进位

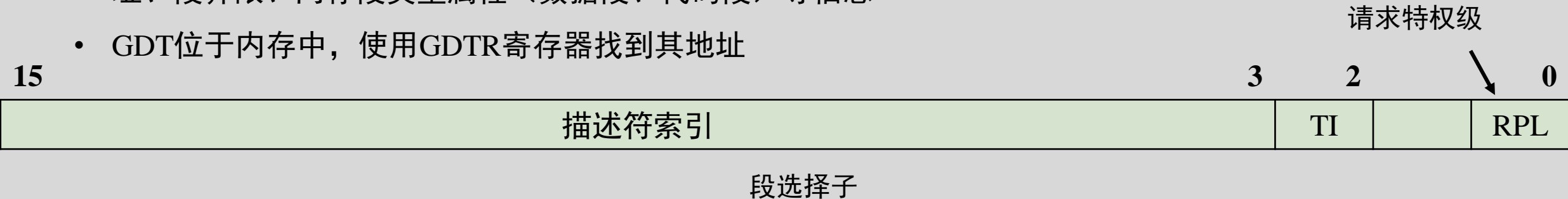
条件标志寄存器

- 控制寄存器。控制CPU的一些重要特性
  - CR0。Protected Enable/Moniter Coprocessor/Emulate Coprocessor/Processor Extension Type/Paging Enable
  - CR1。未使用
  - CR2。页故障线性地址寄存器
  - CR3。页目录基址寄存器，保存页目录物理地址**

控制标志寄存器

# 实模式和保护模式

- CPU的工作模式指的是CPU的寻址方式、寄存器大小等来反映CPU在当前环境下如何工作的概念
- 实模式
  - 早期8086CPU。CPU性能有限，只有20位地址线。
  - 访存格式<段基址: 段内偏移>，物理地址 = 段基址 << 4 + 段内偏移
- 保护模式
  - CPU地址线从20发展为32，寄存器也变为32位。保护模式引入更大空间、更安全的内存访问
  - 段内偏移不变。段寄存器不再存放段基址，而变成段选择器，类似数组索引
  - 关于保护模式下内存访问限制等信息存储在GDT全局描述符表中，每个表项为一个段描述符，存放段基址、段界限、内存段类型属性（数据段、代码段）等信息
  - GDT位于内存中，使用GDTR寄存器找到其地址



- 为什么需要?
  - 效率、使用特殊指令、使用或实现系统调用
- GCC内联汇编格式, AT&T汇编语法
  - 源/目的操作数方向
  - 寄存器命名
  - 立即数
  - 操作数大小
  - 内存操作数
- 基本内联汇编
  - `asm("assembly code");`

OP-code dst src //Intel语法  
Op-code src dst //AT&T语法

`%eax,%ebx`

`$0x12`

`movl foo, %al`

`section:disp(base, index, scale)`

```
asm ( "movl %eax, %ebx\n\t"  
      "movl $56, %esi\n\t"  
      "movl %ecx, $label(%edx,%ebx,$4)\n\t"  
      "movb %ah, (%ebx)");
```



# 扩展内联汇编 Extended Inline Assembly



北京大学  
PEKING UNIVERSITY

- 可以指定操作数，C语言变量
- 可以选择输入输出寄存器，指明要修改的寄存器列表
- GCC优化

```
asm [ volatile ] (  
    assembler template  
    [ : output operands ]           /* optional */  
    [ : input operands ]           /* optional */  
    [ : list of clobbered registers ] /* optional */  
);
```

- `asm/__asm__`。表示汇编的开始
- `volatile/__volatile__`。表示不需要编译器对汇编代码做任何优化
- 汇编模板
- 输出操作数
- 输入操作数
- 一个寄存器列表（可能被修改的寄存器）

# 例子



```
int a=10, b;  
asm ( "movl %1, %%eax;  
      movl %%eax, %0;"  
      : "=r"(b)           /* output */  
      : "r"(a)             /* input */  
      : "%eax"            /* clobbered register */  
);  
b = a; // in C code
```

- b是输出操作数，用%0访问，0表示第1个位置，a是输入操作数，%1访问
- r是一个constraint。让GCC自己选择一个寄存器去存储变量a。=表示write only，用于输出操作数
- %%与%，帮助编译器区分寄存器和操作数
- 告诉编译器eax可能被修改，这样GCC在调用内联汇编之前就不会报错eax中的内容

- 可以指明一个操作数是否在寄存器中，在哪些寄存器中；
- 可以指明操作数是否是内存引用，如何寻址；
- 可以说明操作数是否是立即数常量，以及其可能的值（或值的范围）
- 常用的constraints
  - 寄存器操作数constraint: r/a/b/c/d/S/D。GCC会在通用寄存器中进行选择
  - 内存操作数constraint: m。当操作数在内存中时，任何对其操作会直接在内存中进行。
  - 匹配constraint: 0。限定使用哪个寄存器，既用来输入又用来输出。可以更高效地使用寄存器。
- 其他：
  - o。内存操作数，要求内存地址在一个段中。
  - V。内存操作数，不在一个段中，即除了m和o的情况。
  - i。立即数操作数，包括在编译器能够确定的符号常量。
  - n。一个确定值的立即数。
  - g。除了通用寄存器外的任何寄存器。
  - .....

# 总结与问题

# 总结



北京大学  
PEKING UNIVERSITY

- 一次只能被一个进程访问的资源被称为临界资源
- 访问临界资源的那段代码被称为临界区
- 同步是指进程间的直接制约关系，多个进程完成合作需要遵循的基本工作次序。可通过信号量或缓冲区等实现
- 互斥是指进程间的间接制约关系。进程之间争用锁来实现对临界区的互斥访问
- 竞争状态是指一个系统的输出依赖于不受控制的事件发生的顺序。例如两个进程要同时写一块内存
- 临界区操作时关闭了中断，保证了临界区操作的原子性。会影响性能，甚至导致巨大的安全问题
- spinlock自旋锁，核心是维护locked标志位，为0表示未占用，为1表示占用
- xchg函数和xchg指令。交换两个寄存器或在寄存器与内存之间交换值
- 学习（复习）了一下汇编相关的知识，主要是x86中的常用寄存器以及GCC内联汇编的用法

# 思考



北京大学  
PEKING UNIVERSITY

- 在acquire锁和release锁之间，系统中断是一直关闭的吗？
- 进程在临界区内可以被中断吗？开启中断会发生什么？Linux/Windows/xv6如何设计的
- 如何用自旋锁实现信号量、读写锁、信号机制，有什么改进，Linux/Windows/Nachos.....
- xv6是支持多级中断的，这会有什么问题？
- 自旋锁能否递归？如何实现？
- acquire关闭系统中断时，只调用了cli这个指令，其他CPU上中断会受影响吗？
- 进程在临界区内，是否可以被调度？
- 扩展内容，xv6中的sleeplock的实现
- 共享文档是如何实现的？石墨、飞书、腾讯文档.....

谢谢