



操作系统

第4章 进程同步

朱小军, 教授
<https://xzhu.info>
南京航空航天大学
计算机科学与技术学院
2025年春

目录

- 为什么要提供同步机制?

- 基础问题：临界区问题

 - 临界区问题的解决办法

 - 软件同步机制

 - 硬件同步机制

- 高级同步机制：信号量

 - 利用信号量解决同步问题

 - 经典的进程同步问题

 - 其他同步问题

- 管程

网友提问：

银行卡里只有100块钱，在ATM取钱时，只要手够快，是不是可以在取钱的一瞬间把钱同时转到支付宝？

被挤破产的小银行



balance:
共享变量

```
int withdraw(){  
    if(balance>=100) {  
        balance-=100;  
        return 100;  
    }  
    return 0;  
}
```

服务器每收到一个取款、转账请求，创建线程执行
withdraw

网友的愿望能
达成吗？

```
#define INIT_BAL 100*10000 // 初始余额
```

```
int balance = INIT_BAL;  
int current_thread_number=0;
```

```
int withdraw() {  
    if (balance >= 100) {  
        balance -= 100;  
        return 100;  
    }  
    return 0;  
}
```

```
// 线程工作函数
```

```
void* thread_func(void* arg) {  
    current_thread_number++;  
    while(current_thread_number<=1); //waiting for the second thread to start  
    int money = 0;  
    int ret;  
    while ((ret = withdraw()) > 0) {  
        money += ret;  
    }  
    printf("Thread %s withdrew total:\t %d\n", arg=="0?"ATM":"AliPay", money);  
    return (void *)money;  
}
```

```
void main() {  
    pthread_t ATM, AliPay;  
    void * ret_ATM, * ret_AliPay;  
    printf("Initial balance:\t %d\n", balance);  
    pthread_create(&ATM, NULL, thread_func, 0);  
    pthread_create(&AliPay, NULL, thread_func, (void *) 1);  
    pthread_join(ATM, &ret_ATM);  
    pthread_join(AliPay, &ret_AliPay);  
    printf("Total withdrawn:\t %ld\n", (long)ret_ATM + (long)ret_AliPay);  
    printf("Final balance in the Bank:\t %d\n", balance);  
}
```

每次取100

银行服务器的工作
线程：

每接到一个请求，
取一次，直到没钱

创建两个线程，
对应于ATM、支
付宝

显示最终的银行
余额，取走得钱
数

演示

```

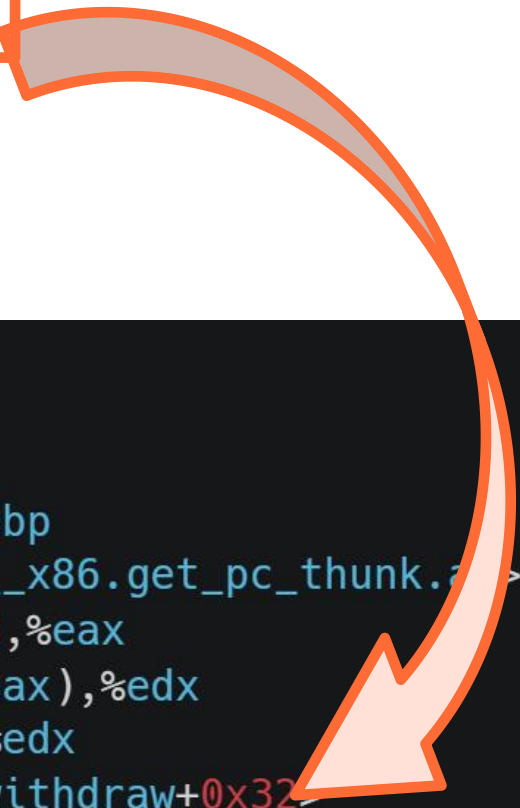
int withdraw() {
    if (balance >= 100) {
        balance -= 100;
        return 100;
    }
    return 0;
}

```

```

0000122d <withdraw>:
122d:  f3 0f 1e fb      endbr32
1231:  55               push    %ebp
1232:  89 e5            mov     %esp,%ebp
1234:  e8 98 01 00 00   call   13d1 <__x86.get_pc_thunk.a>
1239:  05 93 2d 00 00   add     $0x2d93,%eax
123e:  8b 90 3c 00 00 00 mov     0x3c(%eax),%edx
1244:  83 fa 63         cmp     $0x63,%edx
1247:  7e 16           jle     125f <withdraw+0x32>
1249:  8b 90 3c 00 00 00 mov     0x3c(%eax),%edx
124f:  83 ea 64         sub     $0x64,%edx
1252:  89 90 3c 00 00 00 mov     %edx,0x3c(%eax)
1258:  b8 64 00 00 00   mov     $0x64,%eax
125d:  eb 05           jmp     1264 <withdraw+0x37>
125f:  b8 00 00 00 00   mov     $0x0,%eax
1264:  5d             pop     %ebp
1265:  c3             ret

```



计数器

```
#include "pthread.h"
#include "stdio.h"
#define NUM 10000000
long count=0;

void *routine(void *arg){
    for(int i=0;i<NUM;i++)
        count++;
    return 0;
}

void main(){
    pthread_t threads[2];
    pthread_create(&threads[0],NULL,routine,0);
    pthread_create(&threads[1],NULL,routine,0);

    pthread_join(threads[0],NULL);
    pthread_join(threads[1],NULL);
    printf("%ld\n",count);
}
```


上面是用户进程遇到的问题，操作系统内核的设计过程中也有同样的问题

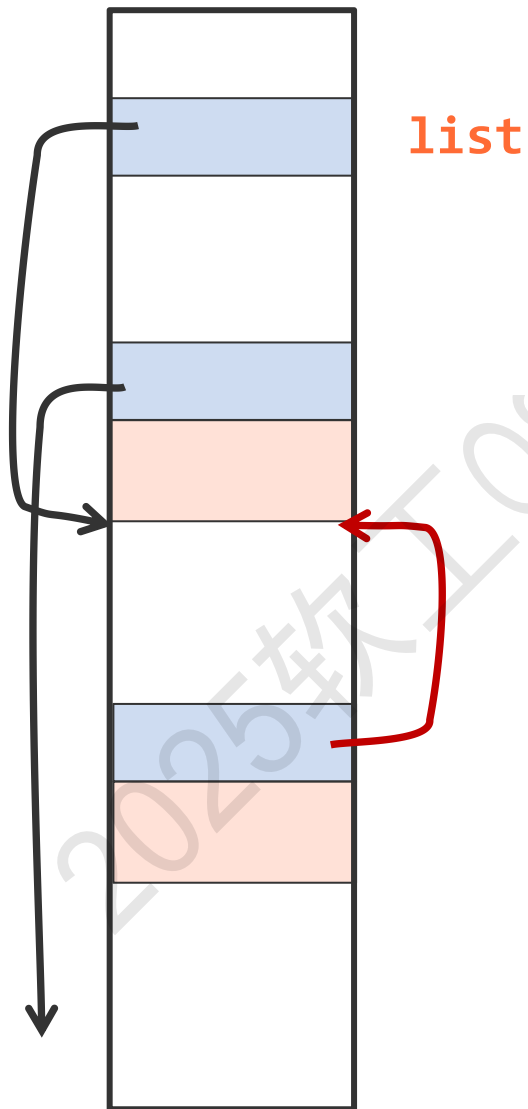
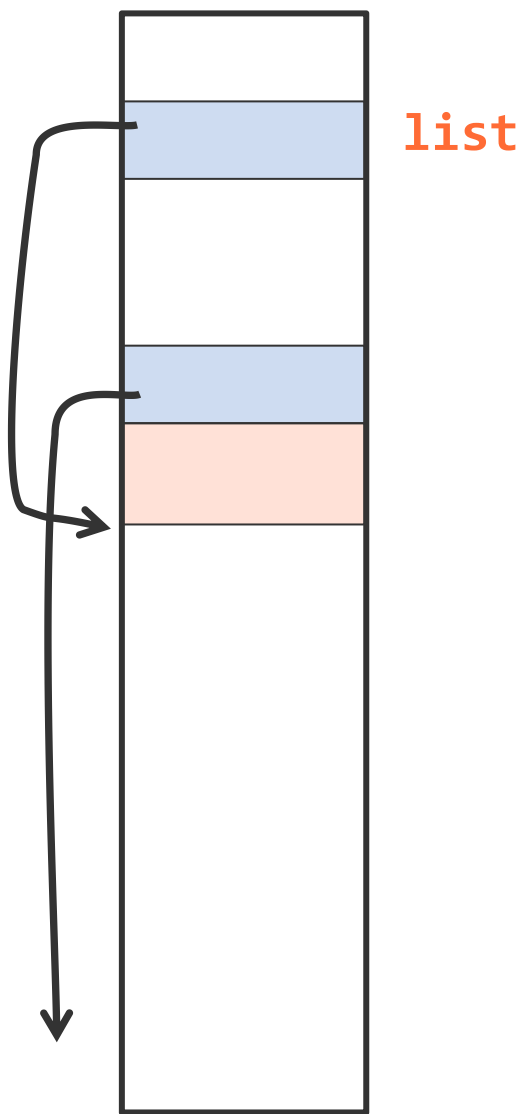
写磁盘与磁盘驱动程序

- 磁盘驱动程序维护一个磁盘请求链表

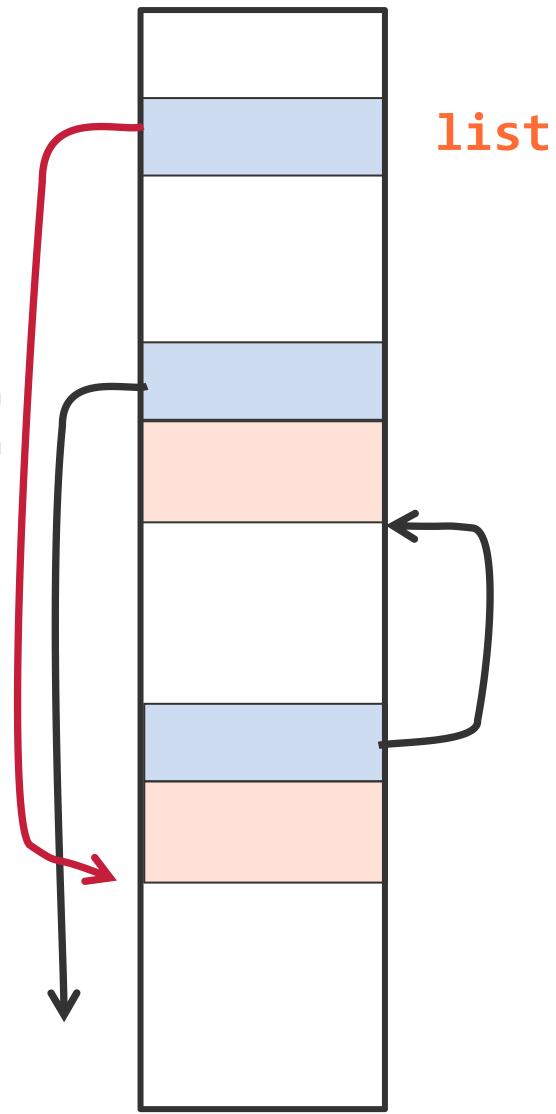
```
struct list *list=0;
struct list{
    int data;
    struct list *next;
}
```

- 用户进程通过系统调用向链表中插入请求

```
void insert(int data){
    struct list *l;
    l=malloc(sizeof *l);
    l->data=data;
    l->next=list;
    list=l;
}
```




...
l->next=list;
list=l;

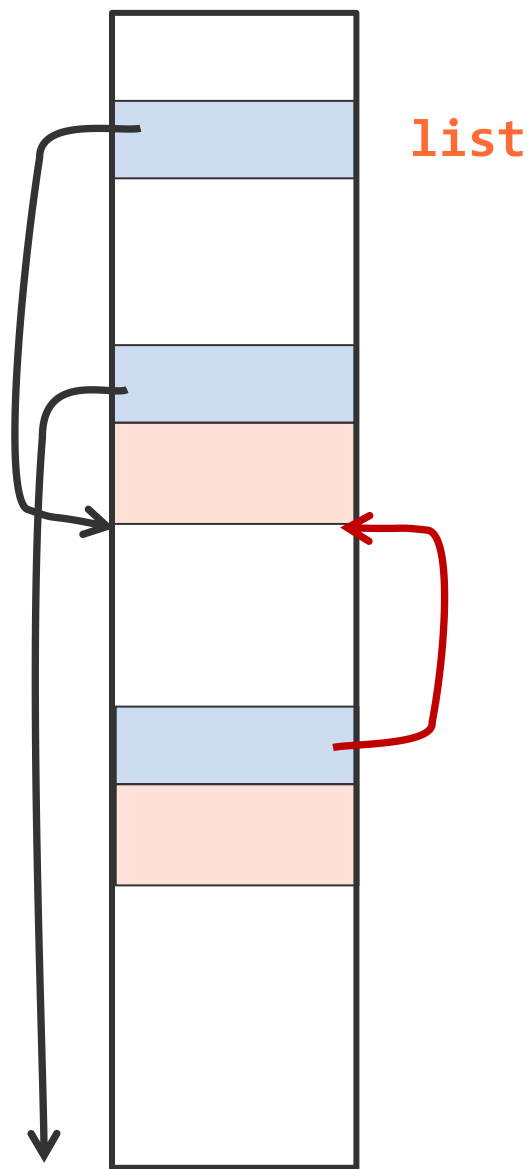


...
l->next=list;
list=l;

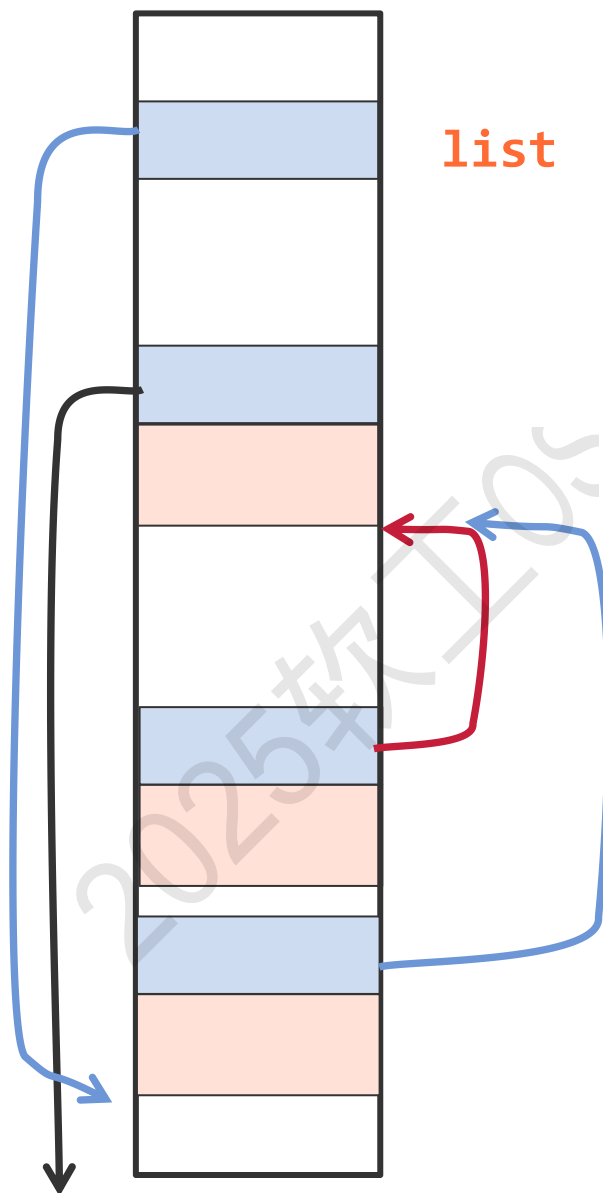
- 如果在执行`list=l`前被打断（时间片到），另一个进程执行系统调用结束，则有何后果？

```
void insert(int data){  
    struct list *l;  
    l=malloc(sizeof *l);  
    l->data=data;  
    l->next=list;  
    list=l;  
}
```

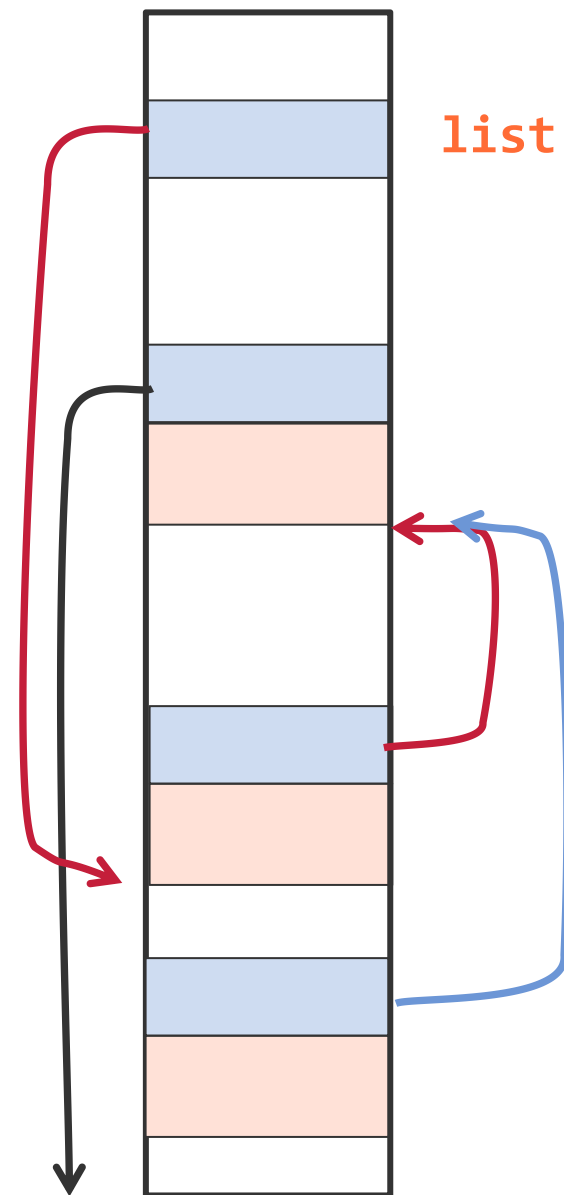




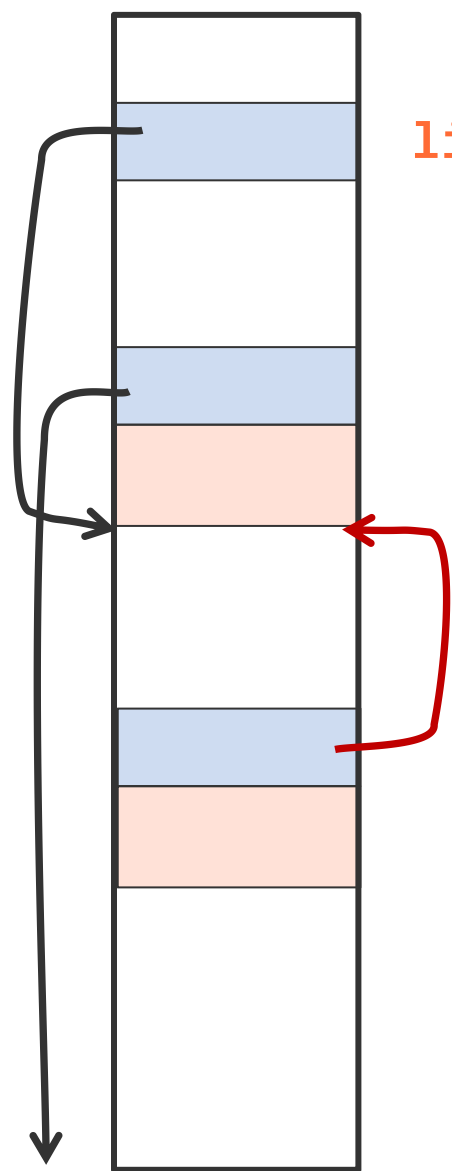
...
`l->next=list;`
`list=l;`



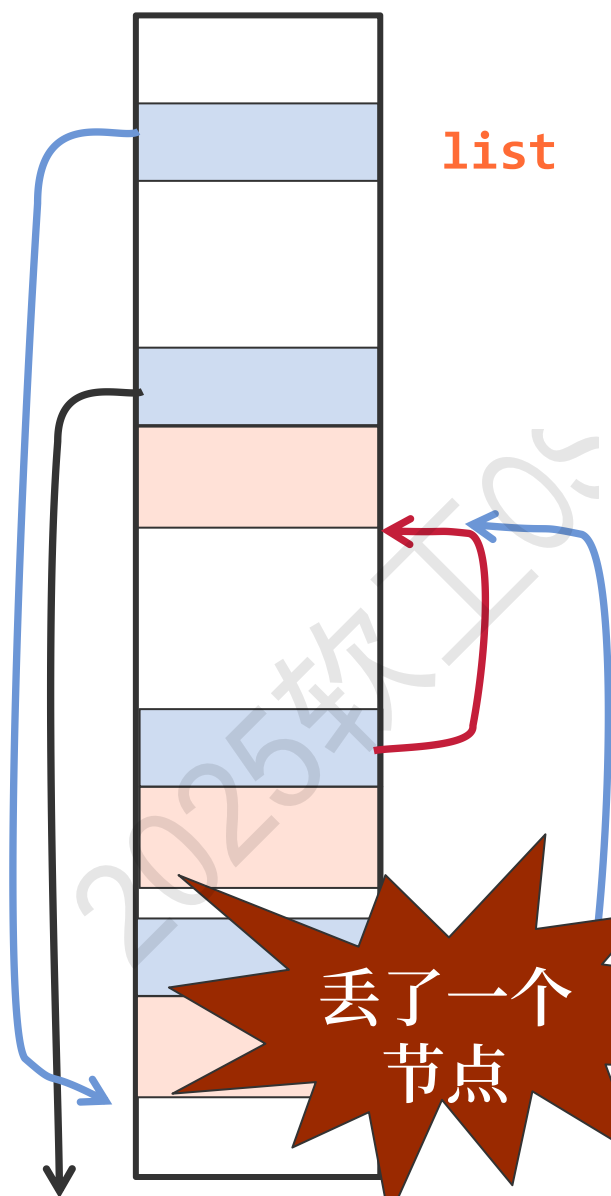
第二个进程修改
 完毕



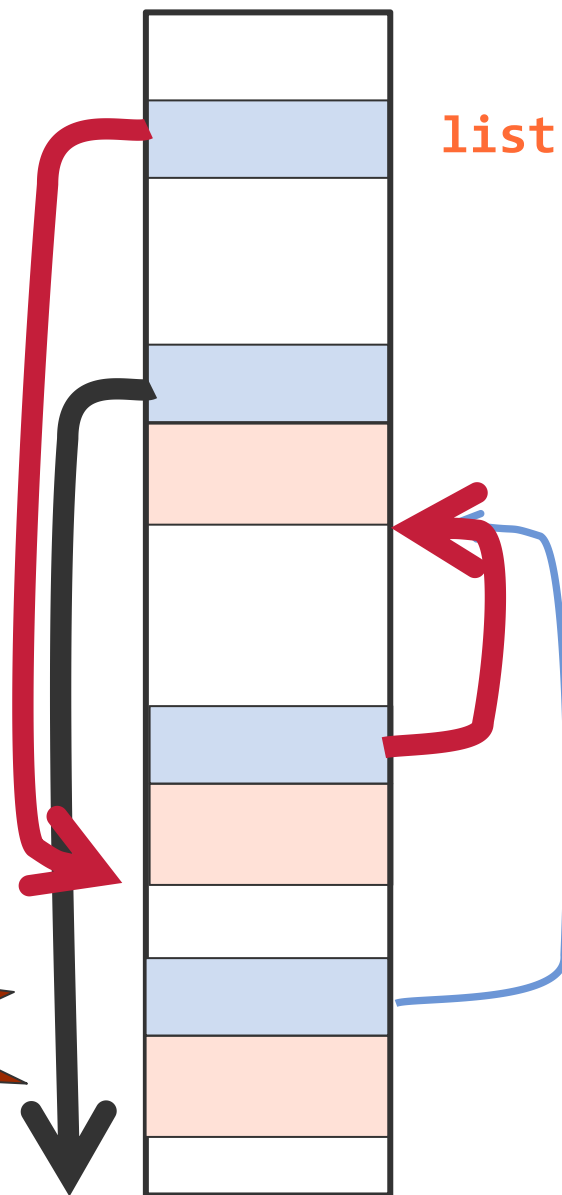
...
`l->next=list;`
`list=l;`



...
`l->next=list;`
`list=l;`



第二个进程修改
完毕



...
`l->next=list;`
`list=l;`

遇到了竞争条件（race condition）

➤ 多个进程（线程）同时访问、修改共享的数据，其最终结果取决于事件发生的次序

需要进程同步方法

目录

- 为什么要提供同步机制？

- 基础问题：临界区问题

 - 临界区问题的解决办法

 - 软件同步机制

 - 硬件同步机制

- 高级同步机制：信号量

 - 利用信号量解决同步问题

 - 经典的进程同步问题

 - 其他同步问题

- 管程

代表性问题：临界区问题

- 考虑一个有 n 个进程/线程的系统

$\{p_0, p_1, \dots, p_{n-1}\}$

- 每个进程/线程有一段**临界区**代码

➤ 比如在临界区内修改共享变量，

写文件

➤ 当1个进程在临界区，所有其

他进程不可在它们的临界区内

- 如何设计一个**协议**让进程正常访问临界区资源，称为**临界区问题**

```
do {  
    entry section  
    critical section  
    exit section  
    remainder section  
} while (true);
```

典型结构

临界区问题的解决方案应满足的条件

- 空闲让进

- 临界区资源**空闲**时，一个想进入临界区的进程应当**可以进入**

- 忙则等待

- 资源被占用（**忙**）时，想进入的进程应**等待**

- 有限等待

- **有限**时间内能进入，不能无限等待（被插队的次数不能无限多）

- 让权等待

- 不能进入时，应当**释放处理机**，尽量不要“忙等”

目录

- 为什么要提供同步机制?
- 基础问题：临界区问题
 - 临界区问题的解决办法
 - 软件同步机制
 - 硬件同步机制
- 高级同步机制：信号量
 - 利用信号量解决同步问题
 - 经典的进程同步问题
 - 其他同步问题
- 管程

尝试

- flag[i], flag[j], 初始值为 false
- flag[i]=true 表示i在临界区内

```
void process_i(){
    do{
        while(flag[j]);
        flag[i]=true;
        critical_section;
        flag[i]=false;
        remainder_section;
    }
}
```

```
void process_j(){
    do{
        while(flag[i]);
        flag[j]=true;
        critical_section;
        flag[j]=false;
        remainder_section;
    }
}
```

Peterson方法

- flag[i], flag[j], turn
- flag[i]=true 表示 i 想进入临界区
- turn=i 表示 i 可以进入临界区

```
void process_i(){
    do{
        flag[i]=true; turn=j;
        while(flag[j]&&turn==j);
        critical_section;
        flag[i]=false;
        remainder_section;
    }
}
```

```
void process_j(){
    do{
        flag[j]=true; turn=i;
        while(flag[i]&&turn==i);
        critical_section;
        flag[j]=false;
        remainder_section;
    }
}
```

- 都想进入临界区时，最后设置turn的会等待

评价

● 优点：

- 当CPU严格顺序执行指令时正确（这种CPU不多了）
- 不需要特殊硬件指令的支持
- 可用于内核或者用户进程

● 缺点：

- 证明正确性较复杂（程序员最讨厌复杂）
- 如果CPU跳着执行指令会有问题（指令乱序优化）
- 当一个进程在临界区内时，另一个进程不断测试，消耗CPU，这称为忙等（busy waiting）

● 没有实际系统用这个（deepseek说的）

目录

- 为什么要提供同步机制?
- 基础问题：临界区问题
 - 临界区问题的解决办法
 - 软件同步机制
 - 硬件同步机制
- 高级同步机制：信号量
 - 利用信号量解决同步问题
 - 经典的进程同步问题
 - 其他同步问题
- 管程

硬件支持

- 大多系统提供硬件支持用于解决临界区问题
- 所有解决方案都基于“锁”的思想
 - 进临界区前拿锁
 - 退出临界区释放锁

```
do{  
    acquire_lock();  
    critical_section;  
    release_lock();  
    remainder_section;  
} while(true);
```


关中断

- 进入临界区前关中断
- 退出临界区前开中断
- 为何有效?

➤ 因为不会被打断

- 缺点

➤ 特权指令，内核空间可用，用户空间不可用

➤ 单CPU可用，多CPU不可用（为何？）

■ 关的是执行指令的CPU的中断，关不了其他CPU的中断

■ 其他处理器本来就可以正常访问内存（即使关了中断）

- 评价：单核处理器系统中仍可使用

```
do{  
    asm{"cli"};  
    critical_section;  
    asm{"sti"};  
    remainder_section;  
} while(true);
```

特殊的硬件指令

- 出错的根本原因在于**读和写之间被打断**，或者穿插了其他CPU的操作
 - “读”：将数据从共享区域读到私有区域
 - CPU私有的寄存器
 - “写”：将数据从私有区域写到共享区域
 - 物理内存
- 能否设计一条**读和写不可被打断**的指令？
 - 测试内存并设置值 (test_and_set)
 - 对换两个内存单元的内容 (swap)

test_and_set 指令

```
bool test_and_set(bool *target){  
    bool rv=*target;  
    *target=true;  
    return rv;  
}
```

- 原子操作，执行期间不但不会被打断，而且
- 其他CPU无法访问target指向的内存

用test_and_set指令解决临界区问题

- 共享变量lock初始值为false

```
bool lock=false;

do{
    while(test_and_set(&lock));
        critical_section;
    lock=false;
        remainder_section;
} while(true);
```

swap 指令

```
bool swap(bool *a, bool *b){  
    bool t=*a;  
    *a=*b;  
    *b=t;  
}
```

- 原子操作，执行期间不但不会被打断，而且
- 其他CPU无法访问a和b指向的内存

用swap指令解决临界区问题

- 共享变量lock初始值为false

```
bool lock=false;

do{
    bool key=true;
    while(key) swap(&lock,&key);
        critical_section;
    lock=false;
        remainder_section;
} while(true);
```

xv6的spinlock

- 依赖于硬件指令xchg:
交换寄存器和内存

```
xchgl reg, mem  
xchgl mem, reg  
xchgl reg, reg
```

- x86.h中的xchg封装函数

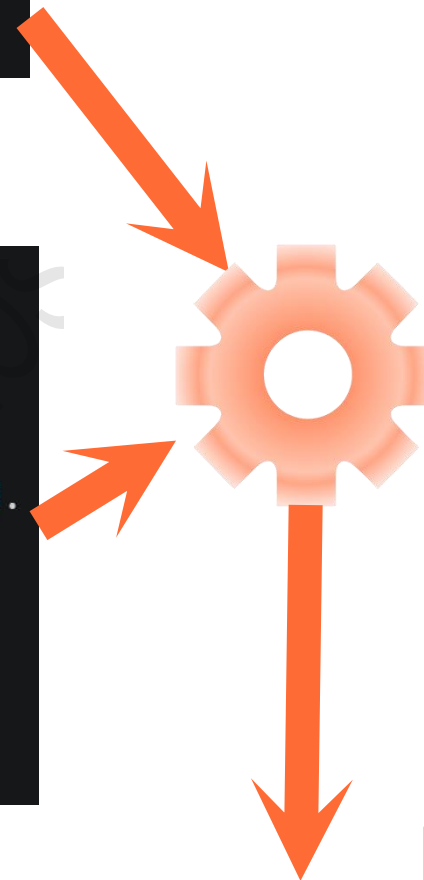
```
static inline uint  
xchg(volatile uint *addr, uint newval)  
{  
    uint result;  
  
    // The + in "+m" denotes a read-modify-write operand.  
    asm volatile("lock; xchgl %0, %1" :  
                  "+m" (*addr), "=a" (result) :  
                  "1" (newval) :  
                  "cc");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

C语言调用代码

```
xchg(&(mycpu()->started), 1);
```

C语言内联函数

```
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;  
}
```



编译器

目标代码

```
mov $0x1,%eax  
lock xchg %eax,0xa0(%edx)
```



```
struct spinlock{
    uint locked;
    ...
}

void acquire(struct spinlock *lk)
{
    ...
    while(xchg(&lk->locked, 1) != 0)
        ;
    ...
}

void release(struct spinlock *lk)
{
    ...
    xchg(&lk->locked, 0);
    ...
}
```

想进入临界区时执行
acquire

退出临界区时执行
release

```
struct spinlock lock;  
initlock(&lock, "name");  
  
do{  
    acquire(&lock);  
        critical_section;  
    release(&lock);  
        remainder_section;  
} while(true);
```

硬件指令方法评价

- 饥饿：有可能出现无穷等待（有限等待×）
- 忙等：当一个进程在临界区内时，其他想进入临界区的进程处于忙等（让权等待×）
- 广泛应用于操作系统内核中，以此为基础可实现后续的所有同步机制
- 历史上，spinlock是为了多处理器同步设计的，那之前用什么？
 - 单核CPU，协作式调度：没有同步问题
 - 单核CPU，抢占式调度：用关中断方法

讨论与实现：有比忙等更好的办法吗？

目录

- 为什么要提供同步机制?
- 基础问题：临界区问题
 - 临界区问题的解决办法
 - 软件同步机制
 - 硬件同步机制
- 高级同步机制：信号量
 - 利用信号量解决同步问题
 - 经典的进程同步问题
 - 其他同步问题
- 管程

信号量 (semaphore)

- 信号量s包含

- 一个整数变量，初始值非负，一般表示资源数量
- 一个阻塞队列，等待的进程阻塞在此队列上

- 信号量的值只能通过两个原子操作访问

要么全执行，要么不执行，不能做一半

- **wait**(s): 若s的值为正，则执行s--后立即返回；
否则，执行s--后将调用进程改为阻塞状态
- **signal**(s): s++，若有进程是阻塞状态则将其唤醒
- 不可直接访问信号量的值

- wait 和 signal以前一直被称为P、V操作

信号量的实现方式：如何解决忙等？

- wait和signal作为系统调用出现
- 讨论：如何保证wait和signal的执行是原子操作？

```
typedef struct{
    int value;
    struct process *list;
} semaphore;

void wait(semaphore *s){
    s->value--;
    if(s->value<0){
        add process to s->list;
        block();
    }
}

void signal(semaphore *s){
    s->value++;
    if(s->value<=0){
        remove process p from s->list;
        wakeup(p);
    }
}
```

课设任务 (proj3)

- 用自旋锁保护

value的读和修改

- 首先定义临界区
- 用自旋锁保护临界区的访问


- 阻塞需要特殊设计

- 不能拿着锁阻塞
- 模仿sleep

```
typedef struct{
    int value;
    struct process *list;
} semaphore;

void wait(semaphore *s){
    s->value--;
    if(s->value<0){
        add process to s->list;
        block();
    }
}

void signal(semaphore *s){
    s->value++;
    if(s->value<=0){
        remove process p from s->list;
        wakeup(p);
    }
}
```



不对

课设任务 (proj3)

● 用自旋锁保护

value的读和修改

- 首先定义临界区
- 用自旋锁保护临界区的访问


● 阻塞需要特殊设计

- 不能拿着锁阻塞
- 模仿sleep

```
typedef struct{
    int value;
    struct process *list;
} semaphore;

void wait(semaphore *s){
    s->value--;
    if(s->value<0){
        add process to s->list;
        block();
    }
}

void signal(semaphore *s){
    s->value++;
    if(s->value<=0){
        remove process p from s->list;
        wakeup(p);
    }
}
```



课设任务 (proj3)

● 用自旋锁保护

value的读和修改

- 首先定义临界区
- 用自旋锁保护临界区的访问


● 阻塞需要特殊设计

- 不能拿着锁阻塞
- 放锁和阻塞需要同时做

```
typedef struct{
    int value;
    struct process *list;
} semaphore;

void wait(semaphore *s){
    s->value--;
    if(s->value<0){
        add process to s->list;
        block();
    }
}

void signal(semaphore *s){
    s->value++;
    if(s->value<=0){
        remove process p from s->list;
        wakeup(p);
    }
}
```



用信号量解决临界区问题

● 创建初始值为1的信号量

- 进入临界区前执行wait、退出临界区前执行signal
- 初始值为1的信号量又名mutex (mutual exclusion)

```
semaphore s=1;
do{
    wait(s);
    critical_section;
    signal(s);
    remainder_section;
} while(true);
```

目录

- 为什么要提供同步机制?
- 基础问题：临界区问题
 - 临界区问题的解决办法
 - 软件同步机制
 - 硬件同步机制
- 高级同步机制：信号量
 - 利用信号量解决同步问题
 - 经典的进程同步问题
 - 其他同步问题
- 管程

经典的同步问题

- 一般用于测试新提出的同步机制

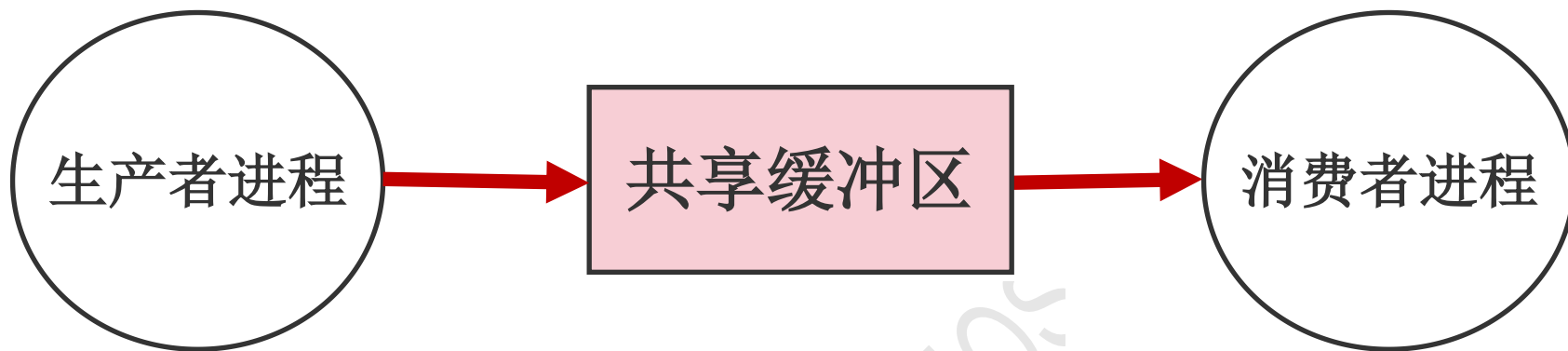
- 生产者消费者问题

- 读者写者问题

- 哲学家进餐问题

2025软工105

生产者消费者问题



●缓冲区包含

- n 个空间，用数组 `buffer` 表示
- `count`: 缓冲区中的产品数
- `in`: 生产者可以放产品的位置, `buffer[in]`
- `out`: 消费者可以取产品的位置, `buffer[out]`



```
while (true) {  
    produce_an_item;  
    while(count==n);  
    buffer[in]=item;  
    in=(in+1)%n;  
    count++;  
}
```

```
while (true) {  
    while(count==0);  
    item=buffer[out];  
    out=(out+1)%n;  
    count--;  
    consume_the_item;  
}
```

会出现错误吗？

用信号量加以修改

```
semaphore mutex=1;
void producer(){
    while (true) {
        produce_an_item;
        while(count==n);
        buffer[in]=item;
        in=(in+1)%n;
        wait(mutex);
        count++;
        signal(mutex);
    }
}
```

```
void consumer(){
    while (true) {
        while(count==0);
        item=buffer[out];
        out=(out+1)%n;
        wait(mutex);
        count--;
        signal(mutex);
        consume_the_item;
    }
}
```

如何消除忙等？

消除“忙等”

- 对生产者，“资源”是什么？
 - 空盒子数
 - 定义信号量empty，初始为n
 - 每次放入一个新产品时，需要wait(empty)
- 对消费者，“资源”是什么？
 - 产品数
 - 定义信号量 full，初始为0
 - 每次消耗一个产品时，需要wait(full)
- 生产者和消费者还需要做什么操作？
 - 生产者需要signal(full)
 - 消费者需要signal(empty)

```
semaphore empty=n,full=0;
void producer(){
    while (true) {
        produce_an_item;
        wait(empty);
        buffer[in]=item;
        in=(in+1)%n;
        signal(full);
    }
}
```

```
void consumer(){
    while (true) {
        wait(full);
        item=buffer[out];
        out=(out+1)%n;
        signal(empty);
        consume_the_item;
    }
}
```

如果有两个生产者，会有什么问题？

```
semaphore mutex=1,empty=n,full=0;
void producer(){
    while (true) {
        produce_an_item;
        wait(empty);
        wait(mutex);
        buffer[in]=item;
        in=(in+1)%n;
        signal(mutex);
        signal(full);
    }
}
```

```
void consumer(){
    while (true) {
        wait(full);
        item=buffer[out];
        out=(out+1)%n;
        signal(empty);
        consume_the_item;
    }
}
```

课堂作业：如果有两个生产者，两个消费者，请改写上面的程序，消除竞争条件

● item变量需要保护吗？

```
semaphore mutex=1,empty=n,full=0;
void producer(){
    while (true) {
        produce_an_item;
        wait(empty);
        wait(mutex);
        buffer[in]=item;
        in=(in+1)%n;
        signal(mutex);
        signal(full);
    }
}
```

生产者消费者的实例

- 进程通信中的管道
- 磁盘驱动程序与用户请求
- 键盘输入、屏幕输出
- 打印机
- 。 。 。

经典的同步问题

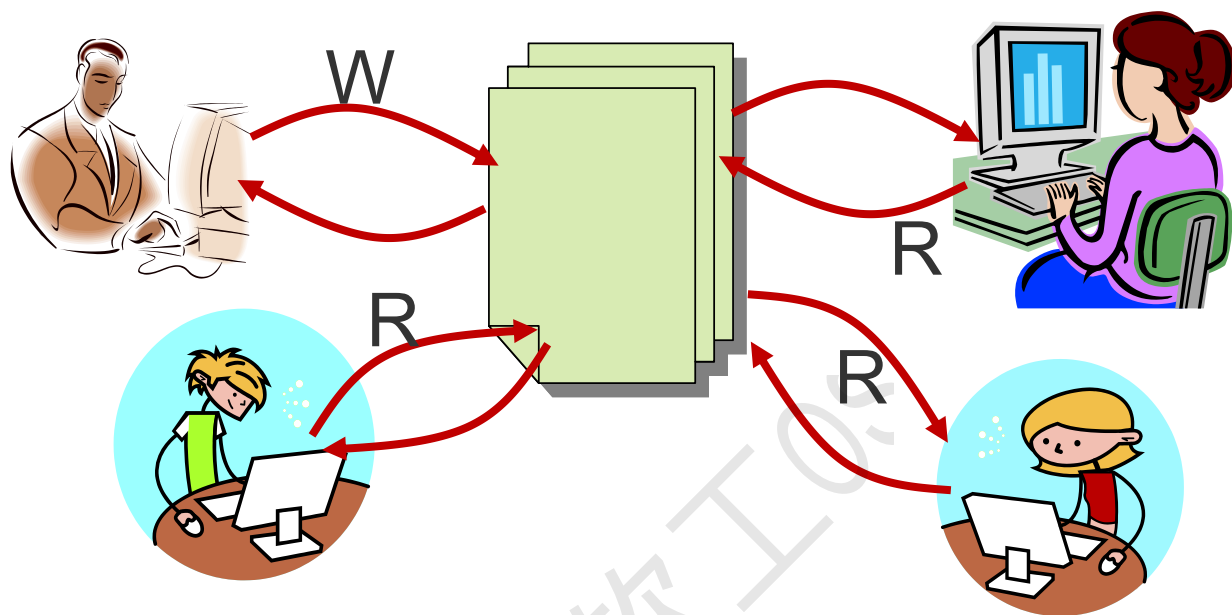
- 一般用于测试新提出的同步机制

- 生产者消费者问题

- 读者写者问题

- 哲学家进餐问题

2025软工10



- 写者：可以读，也可以写
- 读者：只读数据，不可写
- 多个读者可以同时读
- 写者修改数据时，其他写者和读者不可访问数据

```
semaphore mutex=1;
void writer() {
    while(true){
        wait(mutex);
        write_something;
        signal(mutex);
    }
}
void reader(){
    while(true){
        wait(mutex);
        read_something;
        signal(mutex);
    }
}
```

不能同时读！


```
semaphore wmutex=1;
void writer() {
    while(true){
        wait(wmutex);
        write_something;
        signal(wmutex);
    }
}
```

```
semaphore rmutex=1;
int readcount=0;
void reader() {
    while(true){
        wait(rmutex);
        if(readcount==0)
            wait(wmutex);
        readcount++;
        signal(rmutex);
        read_something;
        wait(rmutex);
        readcount--;
        if(readcount==0)
            signal(wmutex);
        signal(rmutex);
    }
}
```

写者有可能被饿死！

经典的同步问题

- 一般用于测试新提出的同步机制

- 生产者消费者问题

- 读者写者问题

- 哲学家进餐问题

2025软工105

哲学家进餐问题

- 哲学家的一生都在两种状态中：吃或者思考
- 当他吃饭时，需要拿起两根筷子，但一次只能拿一只
- 吃完后放回筷子，继续思考
- 哲学家之间不允许抢筷子



尝试

- 资源有五个，所以定义五个信号量

```
semaphore chopstick[5]={1,1,1,1,1}
void process(int i) {
    while(true){
        wait(chopstick[i]);
        wait(chopstick[(i+1)%5]);
        eat;
        signal(chopstick[i]);
        signal(chopstick[(i+1)%5]);
    }
}
```

死锁！

几种解决方案

- 限制同时进餐人数

- 只允许1个人进餐：看黑板
- 2个人？
- 3个人？
- 4个人？
- 5个人？

- 将筷子分等级

- 必须先请求低等级的筷子，然后才可以请求高等级的

- 课堂作业：请写出这种方案

目录

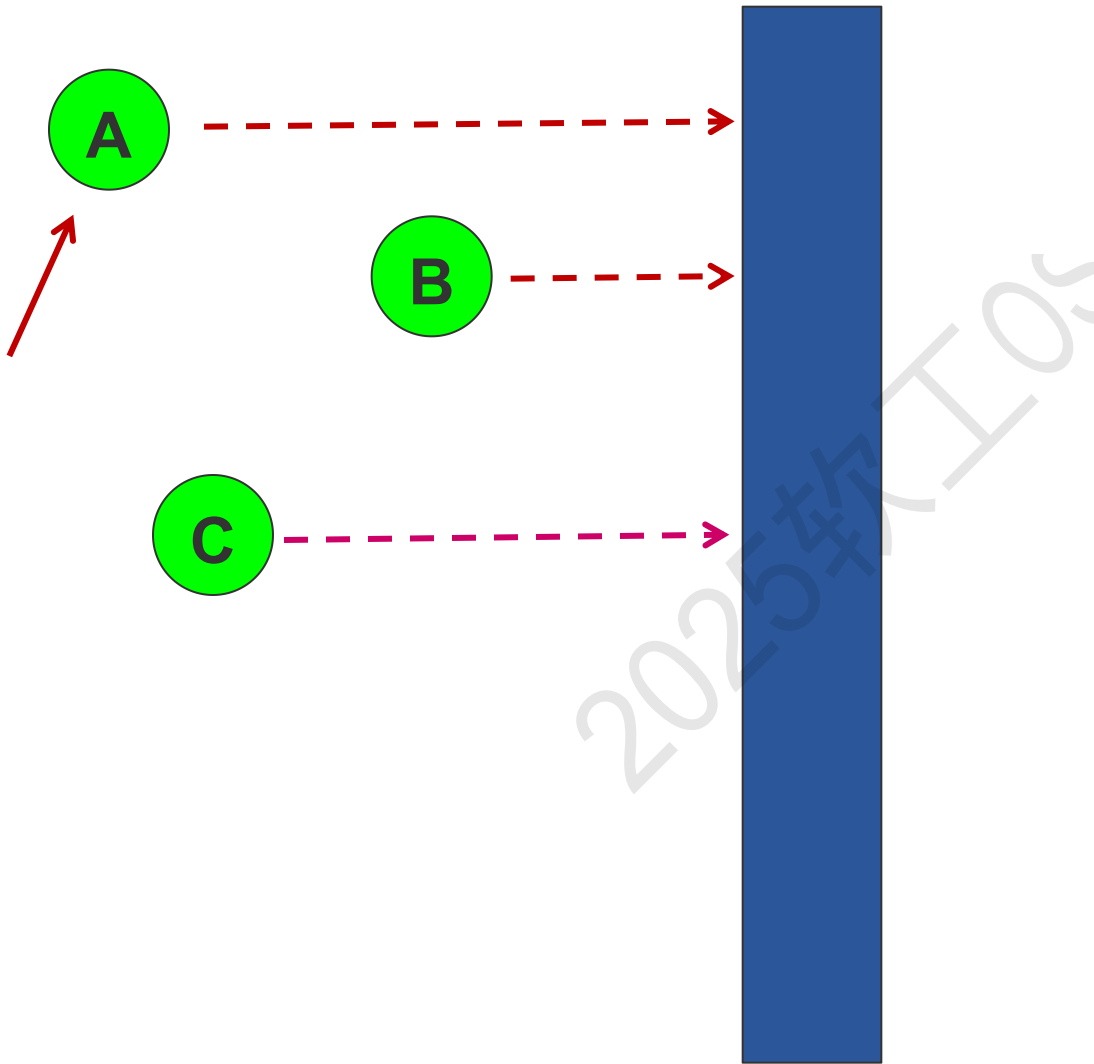
- 为什么要提供同步机制?
- 基础问题：临界区问题
 - 临界区问题的解决办法
 - 软件同步机制
 - 硬件同步机制
- 高级同步机制：信号量
 - 利用信号量解决同步问题
 - 经典的进程同步问题
 - 其他同步问题
- 管程

其他同步问题

蒙特卡洛算法

- 需要用蒙特卡洛法计算 π 的值
- 有全局变量 n_1 , n_2
 - 随机生成 $(x, y) \sim [0,1]^2$ 的点: 个数为 n_1
 - 计算点落到单位圆内的个数: n_2
 - 则: $\pi/4 = n_2/n_1$
- 如何设计一个多线程的程序?
- 是否有出错的可能? 如何消除?

Barriers



Barrier



2025软工105

课堂作业：实现barrier函数

- 实现可以同步3个线程的函数

`void barrier ()`

- 若有3个线程调用了barrier，则只有当最后一个进程执行到barrier时，其他进程才可以继续执行
 - 可以使用全局变量：例如“定义整型变量i；定义信号量i，初始值为x；”
 - barrier无输入参数

前驱图

- 能否将任意的程序并发执行？假如数据共享

- 不行，可能有依赖性！

- 如何描述依赖性？

- 前驱图

节点：表示一条语句，程序段，进程。

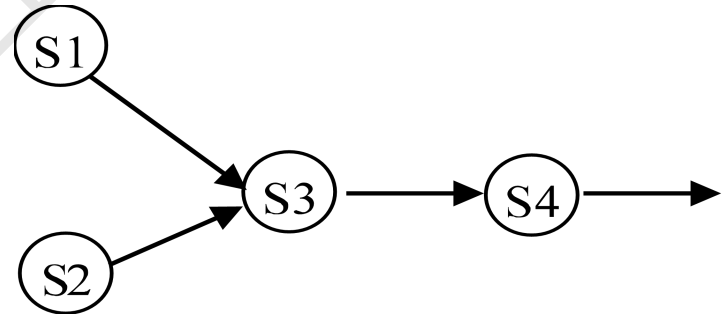
边：表示两结点间的偏序(或前趋)关系。

S1: $a = x + 2$

S2: $b = y + 4$

S3: $c = a + b$

S4: $d = c + 6$



- 如何编写程序，保证按前驱图执行？

- 每条边定义一个初始值为0的信号量

- 左端signal，右端wait

练习

- 用信号量，保证两个进程A执行完后C才可以执行(A->C) 【看黑板】

P1(){ A; B;}

P2(){ C; D;}

- 保证C->A, B->D 【自己练习】

P1(){ A; B;}

P2(){ C; D;}

小银行回顾



balance:
共享变量

```
void withdraw(){  
    if(balance >= 100) {  
        wait(mutex);  
        balance -= 100;  
        signal(mutex);  
        return 100;  
    }  
    return 0;  
}
```

这个修改是否
可以？

服务器每收到一个取款、转账请求，即创建一个线程执行withdraw

目录

- 为什么要提供进程同步机制?

- 基础问题：临界区问题

 - 临界区问题的解决办法

 - 软件同步机制

 - 硬件同步机制

- 高级同步机制：信号量

 - 利用信号量解决同步问题

 - 经典的进程同步问题

 - 其他同步问题

- 管程

管程 (Monitor)

- 信号量功能很强，但是管理分散
 - 对资源的管理分散在不同的进程中，易造成死锁
- 管程含一个互斥锁（隐含）、一个或多个条件变量
 - 管程提出的年代正好是面向对象思想发展的年代
 - 将资源封装为私有、提供操作方法
 - Java中的synchronized关键词、posix标准中的pthread_cond_t（条件变量）
- 互斥锁可以保证任意时刻只有一个进程可以访问管程的共享数据
 - 访问数据前获得锁
 - 访问结束后释放锁

条件变量（可以脱离管程独立存在）

`condition c;`

- 每个条件变量有一个关联的阻塞队列
- 可以执行wait和signal操作
 - wait(c): 释放管程的互斥权，执行此操作的进程的PCB入c链尾部
 - signal(c): 如果c链为空，则相当于空操作，执行此操作的进程继续；否则唤醒第一个等待者
- 怎么跟信号量这么像？
 - 什么是条件？需要额外定义
 - 此处的wait、signal操作没有对“资源数”减1或加1
 - posix标准中有条件变量，没有管程
 - 锁+条件变量可以实现信号量

用管程解决生产者消费者问题

```
Monitor ProducerConsumer{
```

```
    item buffer[N];
```

```
    int in, out, count;
```

```
    condition notfull, notempty;
```

```
public:
```

```
    void put(item x){
```

```
        if(count>=N) cwait(notfull);
```

```
        buffer[in]=x; in=(in+1)%N; count++;
```

```
        csignal(notempty);    }
```

```
    item get(){
```

```
        if(count<=0) cwait(notempty);
```

```
        item x=buffer[out]; out=(out+1)%N;    count--;
```

```
        csignal(notfull);
```

```
        return x; }
```

```
}
```

总结（2025考研大纲）

- 同步与互斥的基本概念

- 基本的实现方法

 - 软件方法，硬件方法

- 锁

- 信号量

- 条件变量

- 经典同步问题

 - 生产者消费者问题

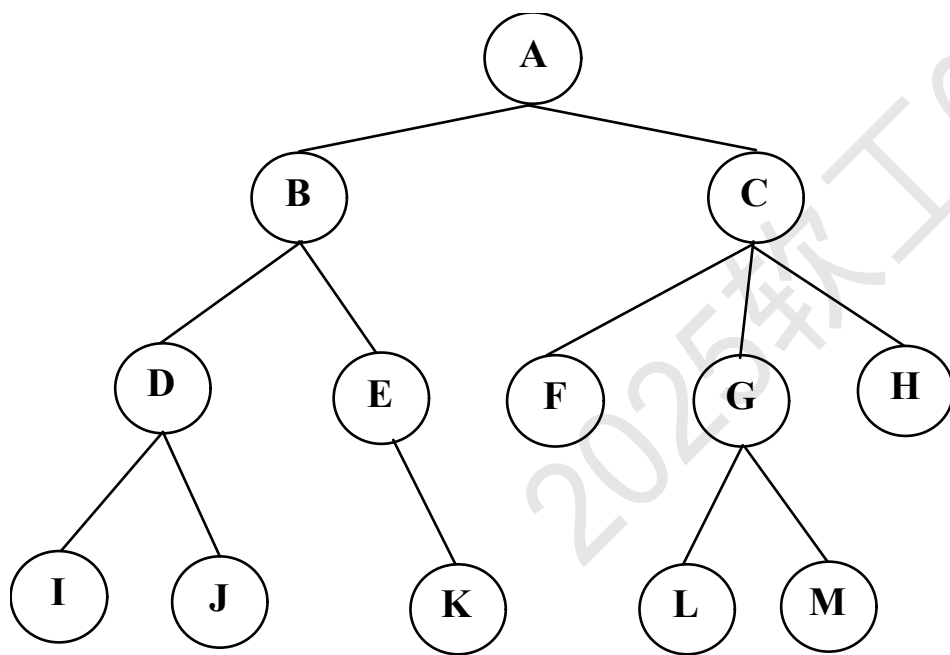
 - 读者写者问题

 - 哲学家进餐问题

阅读教材

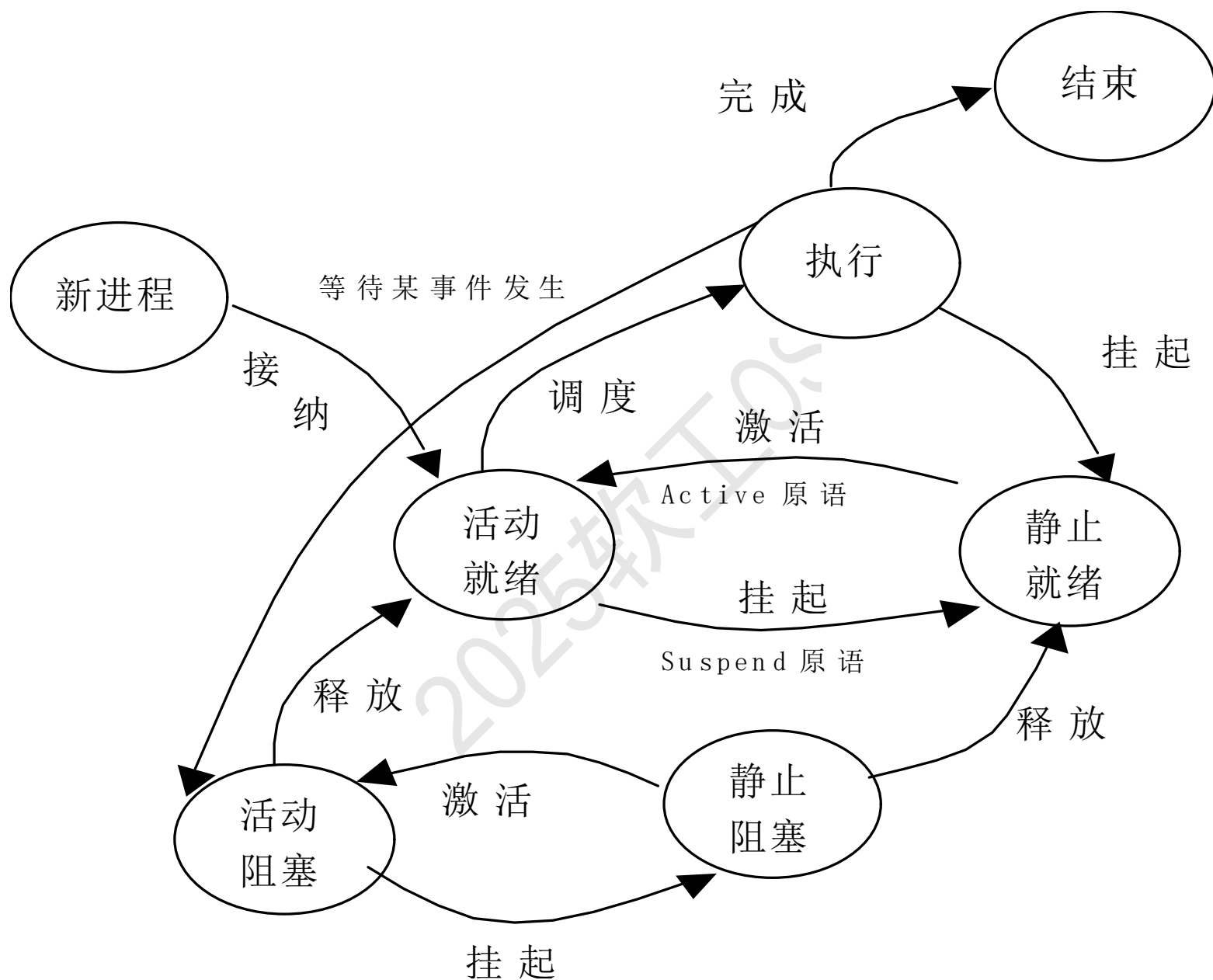
进程图

● 描述进程之间的创建关系



注意：
进程图和前趋图之间的区别。

“挂起”操作



具有挂起状态的进程状态图

延伸知识：Linux的seqlock，写者优先

- 读者检测是否有写者进入，如果是，则读失败，重新读
- 写者之间竞争，均优先于读者
- 如何实现？
 - 用一个 spinlock+sequence number

读者端

```
1 u64 get_jiffies_64(void)
2 {
3
4     do {
5         seq = read_seqbegin(&jiffies_lock);
6         ret = jiffies_64;
7     } while (read_seqretry(&jiffies_lock, seq));
8 }
```

写者端

```
1 static void tick_do_update_jiffies64(ktime_t now)
2 {
3     write_seqlock(&jiffies_lock);
4
5     //临界区会修改jiffies_64
6
7     write_sequnlock(&jiffies_lock);
8 }
```


课程回顾与提问

- 进程同步需求：不同步会出什么错误？
- 硬件同步方法中，关中断适用于什么场景？
- 硬件同步方法中，特殊的硬件指令指什么？举个例子？
- 定义信号量s，哪些操作不可以执行？
 - ① 在定义时初始化为0
 - ② 在定义时初始化为-1
 - ③ 在使用中判断s是否为1
 - ④ 对s执行wait操作
 - ⑤ 对s执行signal操作