



操作系统

第四章

进程同步与通信、进程死锁

文艳军（教授）

计算机学院

Linux 0.11

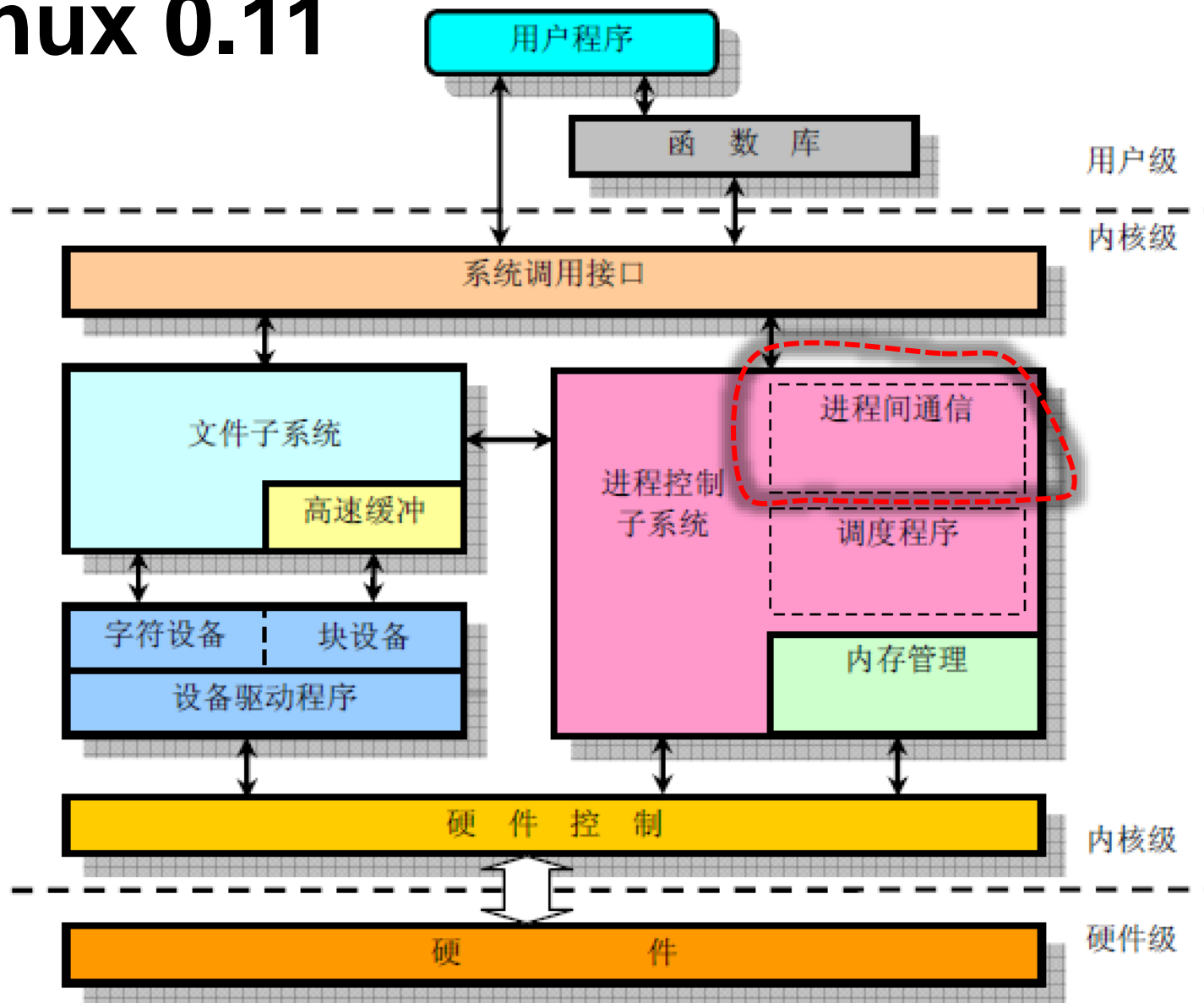


图 2-4 内核结构框图

目录

- 一. 线程的引入
- 二. 并发执行的表示与实现
- 三. 进程的互斥与临界段
- 四. 解决临界段问题的硬件方法
- 五. 并发问题实例: DirtyCOW漏洞

```
#include <stdio.h>  /* 2.c */
#include <unistd.h>
#include <stdlib.h>

int sum1=0, sum2=0;

void p1() {
    int i, tmp=0;
    for(i=1; i<=100; i++)
        tmp += i;
    sum1 += tmp;
}

void p2() {
    int i, tmp=0;
    for(i=101; i<=200; i++)
        tmp += i;
    sum2 += tmp;
}
```

多进程并发的缺陷

```
void p3() {  
    printf("sum: %d\n", sum1+sum2);  
}
```

```
int main() {  
    pid_t  pid;  
    int  stat;  
    pid = fork();  
    if (pid == 0) {  
        p1();  
        exit(0);  
    }  
    p2();  
    pid = wait(&stat);  /* 等待子进程结束 */  
    p3();  
    return 0;  
}
```

输出15050

进程间通信开销大

多进程并发的缺陷

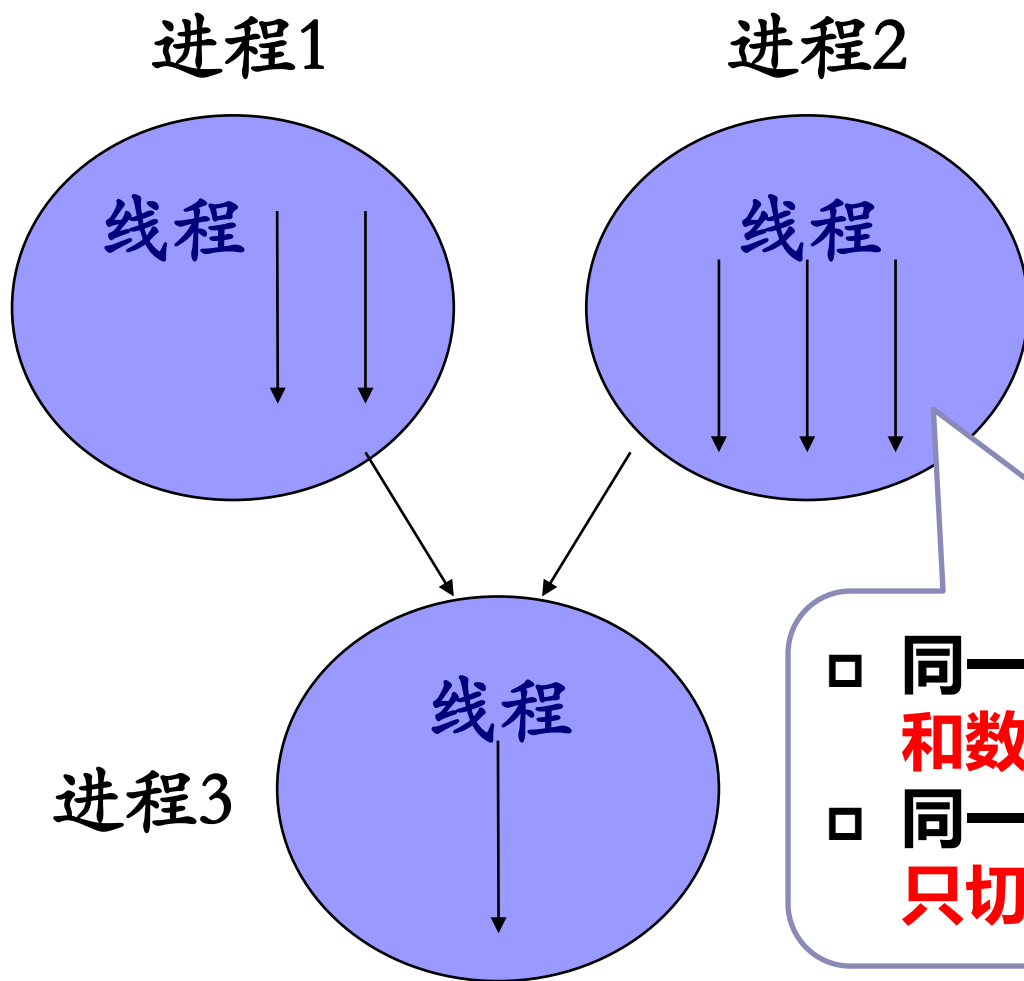
一. 线程的引入

线程：一种实现**进程内并行执行**的机制。同一进程的不同线程之间切换时，**只切换堆栈**，不切换地址空间的其它部分。



进程与线程的关系

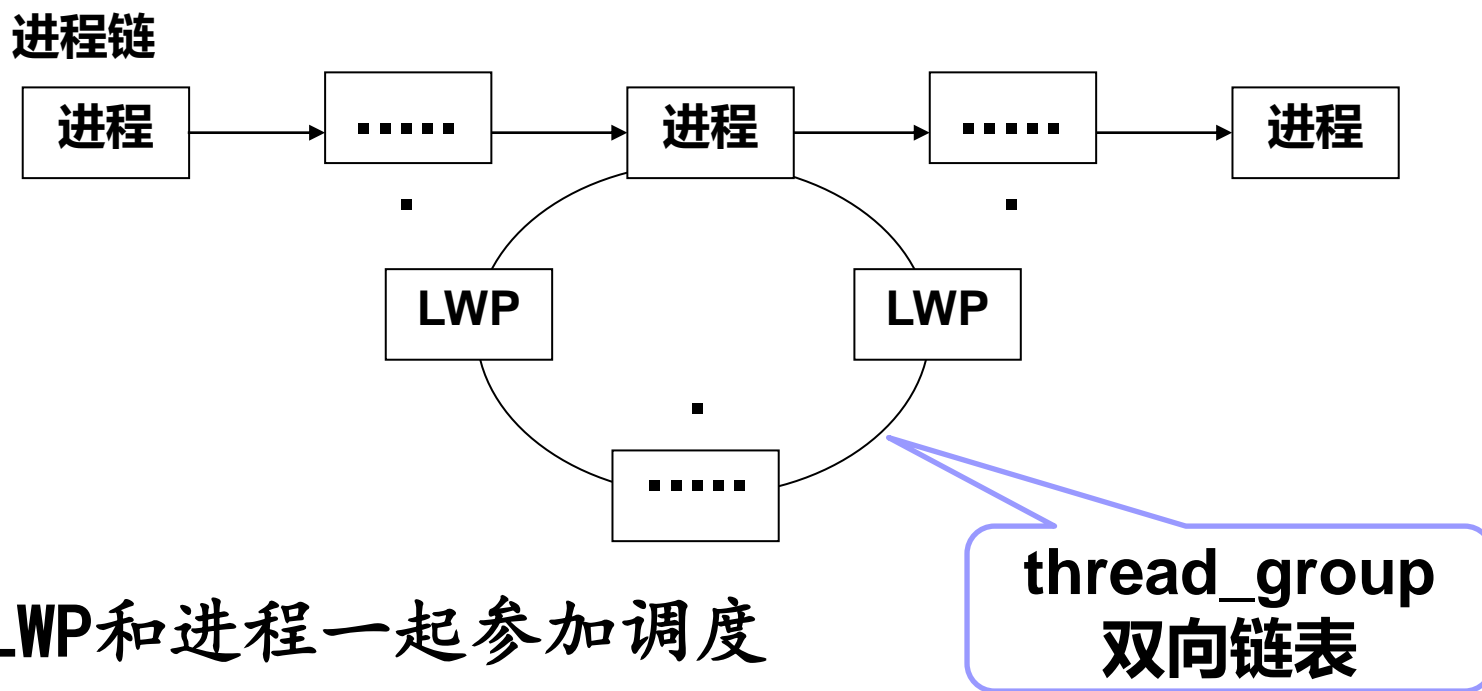
使用线程的优点：可
减少**通信开销**和
任务**切换开销**



- 同一进程的线程**共享代码段和数据段**，但有各自的栈；
- 同一进程的线程之间切换时**只切换地址空间中的栈**

Linux的线程

- Linux在进程机制中实现了线程——轻权进程（Light-Weight Process, LWP）



- LWP和进程一起参加调度
- 在LWP基础上实现了线程库pthread


```
#include <stdio.h> /* 3.c */
```

```
#include <unistd.h>
```

```
#include <stdlib.h>
```

```
#include <pthread.h>
```

```
int sum1=0, sum2=0;
```

```
void * p1() {
```

```
    int i, tmp=0;
```

```
    for(i=1; i<=100; i++)
```

```
        tmp += i;
```

```
    sum1 += tmp;
```

```
}
```

```
void p2() {
```

```
    int i, tmp=0;
```

```
    for(i=101; i<=200; i++)
```

```
        tmp += i;
```

```
    sum2 += tmp;
```

```
}
```

```
void p3() {
```

```
    printf("sum: %d\n", sum1+sum2);
```

```
}
```

全局变量被共享

多线程并发

```
int main() {  
    int res;  
    pthread_t  t1;  
    void *thread_result;  
    res = pthread_create(&t1, NULL, p1, NULL);  
    if (res != 0) {  
        perror("failed to create thread");  
        exit(1);  
    }  
    p2();  
    res = pthread_join(t1, &thread_result);  
    if (res != 0) {  
        perror("failed to join thread");  
        exit(2);  
    }  
    p3();  
    return 0;  
}
```

输出20100

Linux的线程

课内实验：实训5-第1关

- 请将例子程序3.c改为用3个线程实现，功能不变。3个线程分别完成计算任务p1、p2和p3。

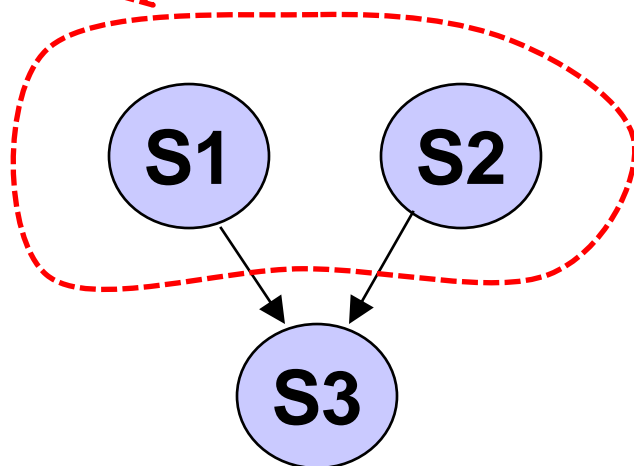
目录

- 一. 线程的引入
- 二. 并发执行的表示与实现
- 三. 进程的互斥与临界段
- 四. 解决临界段问题的硬件方法
- 五. 并发问题实例: DirtyCOW漏洞

二. 并发执行的表示与实现

计算任务存在可并行成分：

可并发执行



```
Parbegin  
  S1;  
  S2;  
Parend  
S3;
```

一般语法:

```
Parbegin S1; S2; ...Sn; Parend;
```

并发执行的实现（多进程）

```
Parbegin  
  S1;  
  S2;  
Parend  
S3;
```

```
pid = fork();  
if (pid == 0) {  
  S1;  
  exit(0);  
}  
S2;  
wait(&status);  
S3;
```

等待子进程结束

并发执行的实现（多线程）

主进程等待线程t1结束
（同步关系）

```
Parbegin  
    S1;  
    S2;  
Parend  
S3;
```

```
pthread_create(&t1, NULL, S1, NULL);  
S2();  
pthread_join(t1, &res);  
S3;
```

目录

- 一. 线程的引入
- 二. 并发执行的表示与实现
- 三. 进程的互斥与临界段
- 四. 解决临界段问题的硬件方法
- 五. 并发问题实例: DirtyCOW漏洞

三. 进程的互斥与临界段

```
int main() {  
    Parbegin  
        s1(100);  
        s2(50);  
    Parend  
    s3();  
}
```

存钱

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

```
double balance = 100;
```

```
void s1(double amount) {
```

```
    double t = balance + amount;  
    balance = t;  
}
```

取钱

```
void s2(double amount) {
```

```
    double t = balance - amount;  
    balance = t;  
}
```

```
void s3() {
```

```
    printf("%f", balance);  
}
```

存取钱问题

```
#include <stdio.h> /* 1.c */
#include <stdlib.h>
#include <pthread.h>
```

初始余额100元

```
double balance = 100;
```

pthread_create
对参数类型有要求

```
void* s1(void * amount) {
    double t = balance + *(double *)amount;
    balance = t;
}
```

```
void s2(double amount) {
    double t = balance - amount;
    balance = t;
}
```

```
void s3() {
    printf("new balance: %f\n", balance);
}
```

多线程实现

```
int main() {  
    int res;  
    pthread_t t1;  
    void *thread_result;  
    double b=100;  
    res = pthread_create(  
        &t1, NULL, s1, (void *)&b);  
    if (res != 0) {  
        perror("failed to create thread");  
        exit(1);  
    }  
    s2(50);  
    res = pthread_join(t1, &thread_result);  
    if (res != 0) {  
        perror("failed to join thread");  
        exit(2);  
    }  
    s3();  
    return 0;  
}
```

存100元

取50元

演示...

三. 进程的互斥与临界段

临界资源

```
double balance = 100;
void s1(double amount) {
    double t = balance + amount;
    balance = t;
}
void s2(double amount) {
    double t = balance - amount;
    balance = t;
}
void pr
}
```

```
Parbegin
    s1(100);
    s2(50);
Parend
s3();
```

临界段



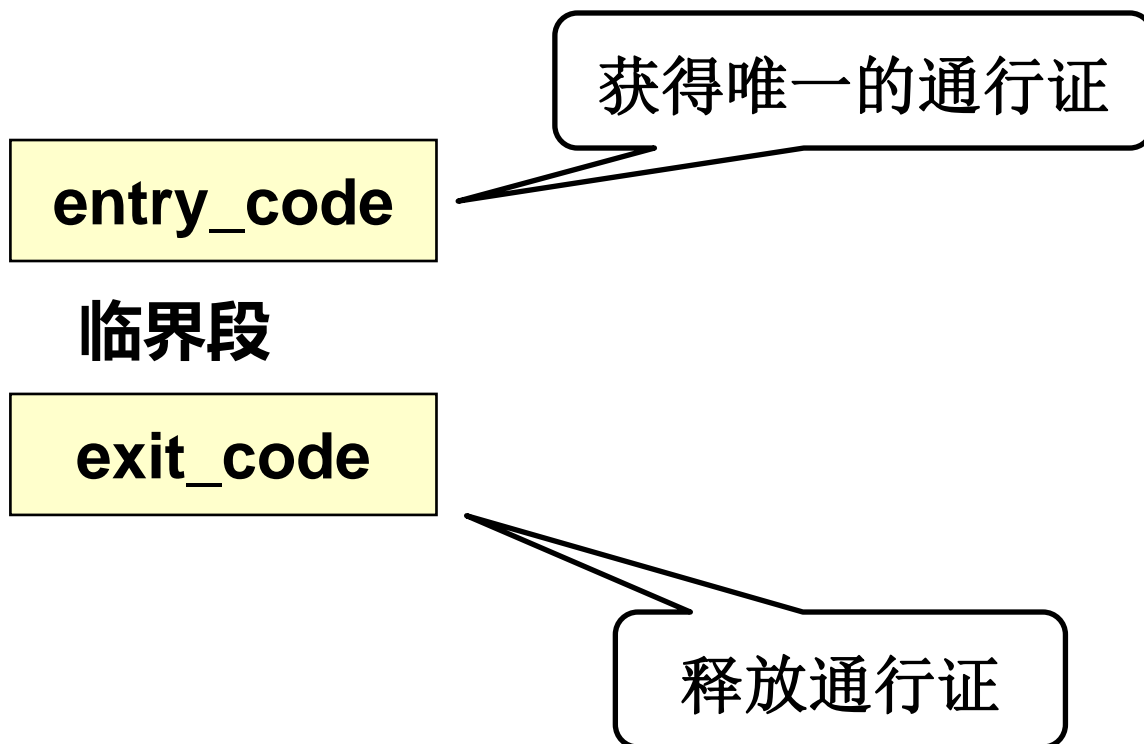
临界段(critical section): 相关进程必须互斥执行的程序段。

目录

- 一. 线程的引入
- 二. 并发执行的表示与实现
- 三. 进程的互斥与临界段
- 四. 解决临界段问题的方法
- 五. 并发问题实例: DirtyCOW漏洞

四. 解决临界段问题的方法

■ 临界段的编程模型



1. 解决临界段问题的软件方法

■ 直观尝试

```
boolean lock=false;
double balance;
void s1(double amount) {
    while(lock)
        ;
    lock = true;
    double t = balance + amount;
    balance = t;
    lock = false;
}
```



1. 解决临界段问题的软件方法

■ Peterson算法

```
boolean flag[2]={false, false};  
int turn;  
double balance;  
void s0(double amount) {  
    flag[0] = true;  
    turn = 1;  
    while(flag[1] && turn==1);  
    double t = balance + amount;  
    balance = t;  
    flag[0] = false;  
}
```


2. 解决临界段问题的硬件方法

- ① 屏蔽中断（单CPU系统）
- ② Test_and_Set指令（多CPU）
- ③ Swap指令

2. 解决临界段问题的硬件方法

① 屏蔽中断（单CPU系统）

```
double balance;  
void s1(double amount) {  
    关中断(cli)  
    double t = balance + amount;  
    balance = t;  
    开中断(sti)  
}  
void s2(double amount) {  
    关中断(cli)  
    double t = balance - amount;  
    balance = t;  
    开中断(sti)  
}
```

② Test_and_Set指令 (多CPU)

一条硬件指令，功能为：

```
boolean Test_and_Set (boolean &target) {  
    boolean rv = target;  
    target = true;  
    return rv;  
}
```

```
double balance;  
boolean lock = false;  
void s1(double amount) {  
    while Test_and_Set(lock) ;  
    double t = balance + amount;  
    balance = t;  
    lock = false;  
}
```



目录

- 一. 线程的引入
- 二. 并发执行的表示与实现
- 三. 进程的互斥与临界段
- 四. 解决临界段问题的方法
- 五. 并发问题实例: DirtyCOW漏洞

互斥与临界段问题

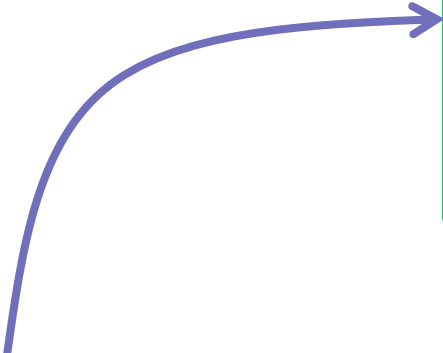
■ DirtyCOW漏洞



DirtyCOW漏洞

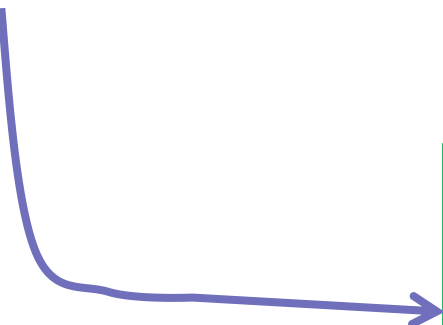
```
$ sudo -s
# echo this is not a test > foo
# chmod 0404 foo
$ ls -lah foo
-r-----r-- 1 root root 19 Oct 20 15:23 foo
$ cat foo
this is not a test
$ gcc -pthread dirtycow.c -o dirtycow
$ ./dirtycow foo m00000000000000000000
mmap 56123000
madvise 0
procselfmem 1800000000
$ cat foo
m0000000000000000000000
```

DirtyCOW



```
f2=open("/proc/self/mem", O_RDWR);  
while(1){  
    lseek(f2, map, ...);  
    write(f2, ...);  
}
```

```
f1 = open("passwd", O_RDONLY);  
map = mmap(..., PROT_READ, MAP_PRIVATE, f1 ...);
```



```
while(1){  
    madvise(map, ..., MADV_DONTNEED);  
}
```

Yanjun Wen, Ji Wang. Analysis and remodeling of the DirtyCOW vulnerability by debugging and abstraction,

小结

- 一. 线程的引入
- 二. 并发执行的表示与实现
- 三. 进程的互斥与临界段
- 四. 解决临界段问题的硬件方法

小结

1. 并发执行的实现

多进程、多线程

2. 进程的同步与互斥

① 互斥与临界段问题

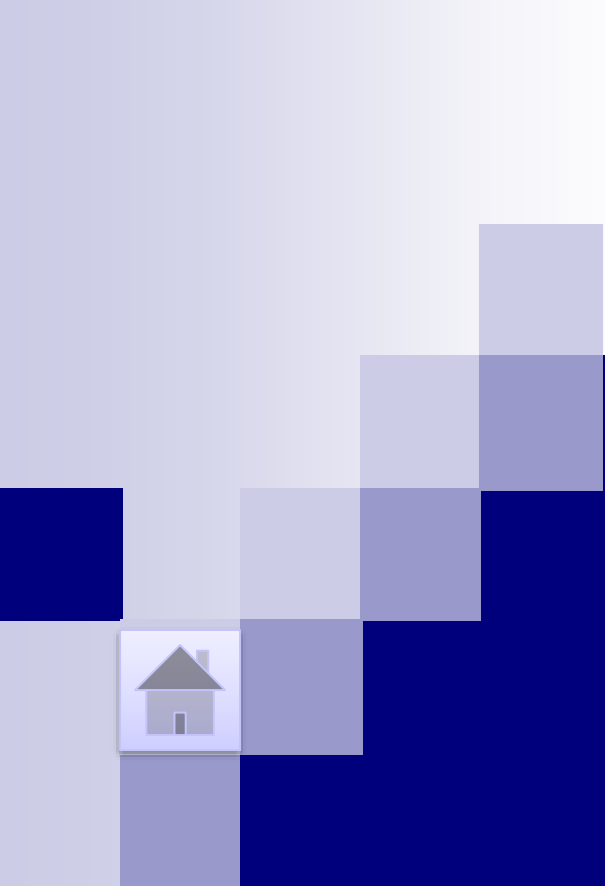
临界段、软件方法

② 解决临界段问题的硬件方法

屏蔽中断、Test_and_Set指令

③ 解决临界段问题的信号量方法

④ 信号量的应用



操作系统

第四章 第2讲

信号量及其应用

文艳军（教授）
计算机学院

回顾

1. 并发执行的实现
多进程、多线程

2. 进程的同步与互斥

① 互斥与临界段问题
临界段、软件方法

② 解决临界段问题的硬件方法
屏蔽中断、Test_and_Set指令

③ 解决临界段问题的信号量方法

④ 信号量的应用

目录

- 一. 信号量的概念
- 二. 用信号量实现互斥和同步
- 三. 用信号量实现受控并发
- 四. 受控并发的其他形式

一. 信号量的概念

信号量 S: 一种特殊变量, 其操作限制为:

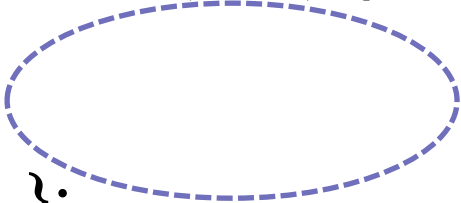
- **赋初值**: 只能初始化一次, 表示资源数量
- **P(S)**: 获得一个资源(数量减一), 若不能则**等待**
- **V(S)**: 释放一个资源(数量加一)

P、V操作的实现
应保证并发场景
下的功能正确性

功能定义:

```
semaphore S;  
P(S): while(S<=0) continue;  
      S = S - 1;  
V(S): S = S + 1;
```

举例：用屏蔽中断方法实现P、V操作在并发场景下的功能正确性

```
P(s) {  
    关中断;  
    while (s ≤ 0) {  
          
    };  
    s = s - 1;  
    开中断;  
}  
V(s) {  
    关中断;  
    s = s + 1;  
    开中断;  
}
```



功能定义：

```
P(S): while(S ≤ 0) continue;  
      S = S - 1;  
V(S): S = S + 1;
```

举例：用阻塞等待方法实现P、V操作的原子性

P(s) {

关中断;

s.value = s.value - 1;

if (s.value < 0) {

开中断;

将当前进程挂入s.L队列然后重新调度

}

开中断;

}

V(s) {

关中断;

s.value = s.value + 1;

if (s.value <= 0)

从s.L队列中解挂一个进程并置为就绪态

开中断;

}

可消除忙等待，
提高CPU效率

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

下列关于信号量的说法正确的有：

- ☒ A 信号量只支持赋初值、P、V三种操作
- ☐ B 对信号量赋初值可以进行多次
- ☒ C P操作表示获取一个资源，V操作表示释放一个资源
- ☐ D 信号量内部有一个资源计数，可以从外部直接读出该值

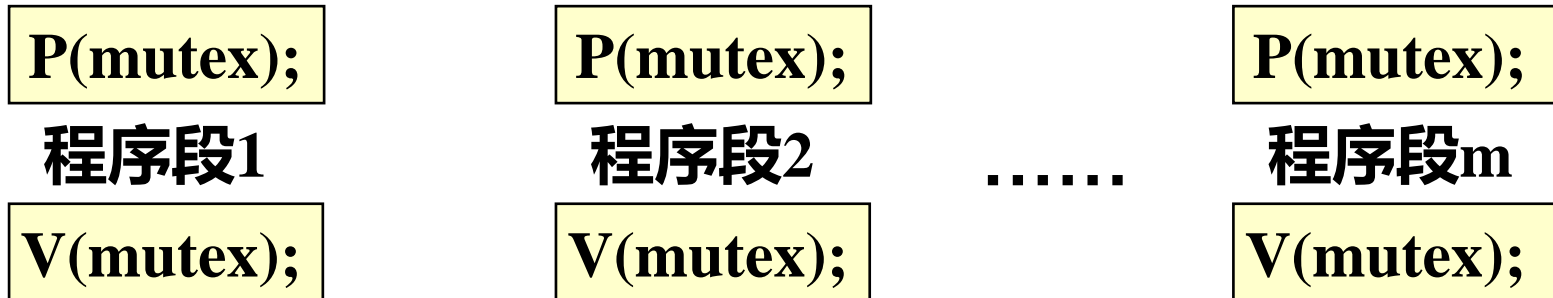
提交

目录

- 一. 信号量的概念
- 二. 用信号量实现互斥和同步
- 三. 用信号量实现受控并发
- 四. 受控并发的其他形式

2. 用信号量实现互斥和同步

- 用信号量实现互斥
 - ➡ 信号量初值设为1



同时进入这些程序段的进程数 ≤ 1

用信号量实现互斥

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

double balance = 100;
semaphore s = 1;
s1(double amount) {
    P(s);
    double t = balance + amount;
    balance = t;
    V(s);
}
s2(double amount) {
    P(s);
    double t = balance - amount;
    balance = t;
    V(s);
}
```

```
int main() {
    Parbegin
        s1(100);
        s2(50);
    Parend
    s3();
}
```

存取钱问题

```
... /* 2.c */  
#include <semaphore.h>  
double balance = 100;  
sem_t s;
```

```
void* s1(void * amount) {  
    sem_wait(&s);  
    double t = balance + *(double *)amount;  
    balance = t;  
    sem_post(&s);  
}
```

P操作

V操作

```
void s2(double amount) {  
    sem_wait(&s);  
    double t = balance - amount;  
    balance = t;  
    sem_post(&s);  
} ...
```

多线程实现

```
int main() {  
    int res;  
    pthread_t  t1;  
    void *thread_result;  
    double  b=100;  
    sem_init(&s, 0, 1);  
    res = pthread_create(  
                                &t1, NULL, s1, (void *)&b);  
    if (res != 0) {  
        perror("failed to create thread");  
        exit(1);  
    }  
    s2(50);  
    res = pthread_join(t1, &thread_result);  
    if (res != 0) {  
        perror("failed to join thread");  
        exit(2);  
    }  
    s3();  
    return 0; }
```

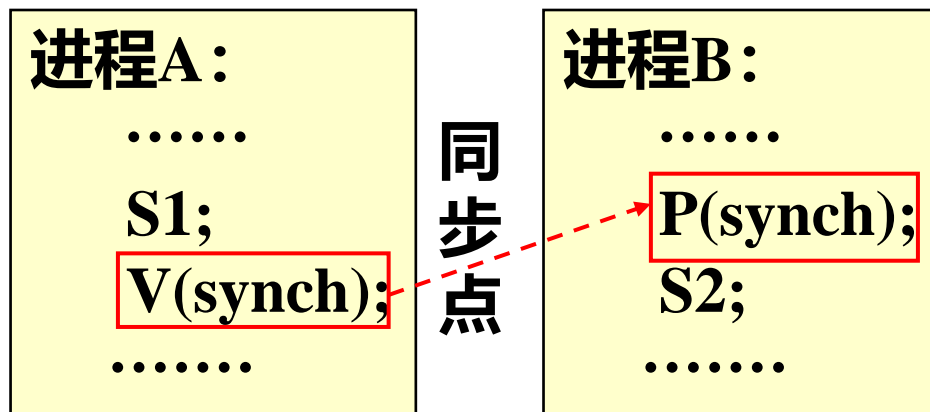
信号量的初始化,
初值为1, 其中的0
表示线程间共享

演示...

2. 用信号量实现互斥和同步

● 实现同步

- ➡ S1执行完后S2才能执行
- ➡ 信号量**初值为0**



解决临界段问题的信号量方法

■ **管程 (Monitor)**：一种用来**集中管理临界段**的抽象数据类型

- 局部于该管程的共享数据（临界资源）
- 局部于该管程的一组操作过程（临界段）
- 对局部于该管程的数据的初始化函数

解决临界段问题的信号量方法

■ Java的synchronized函数：

- 可以防止多个线程同时访问这个对象的synchronized方法
- 如果一个对象有多个synchronized方法，只要一个线程访问了其中的一个synchronized方法，其它线程不能同时访问这个对象中任何一个synchronized方法

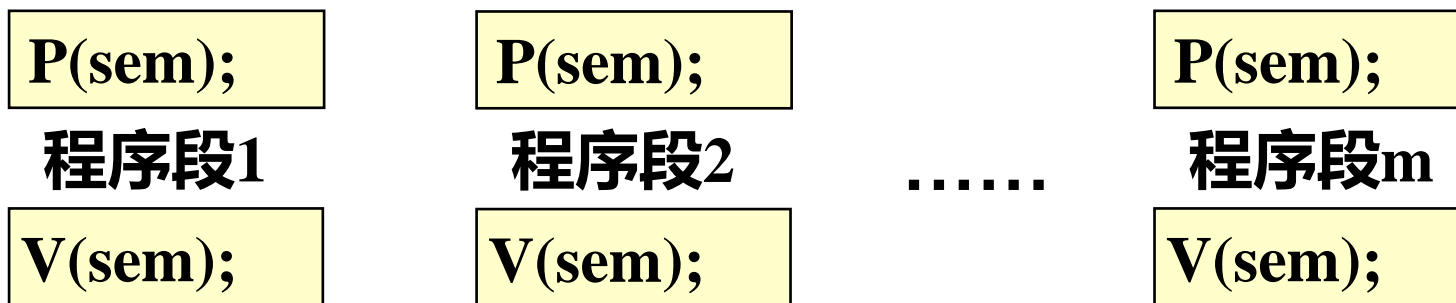
目录

- 一. 信号量的概念
- 二. 用信号量实现互斥和同步
- 三. 用信号量实现受控并发
- 四. 受控并发的其他形式

3. 用信号量实现受控并发

- 实现受控并发

➡ 信号量sem的初值为n

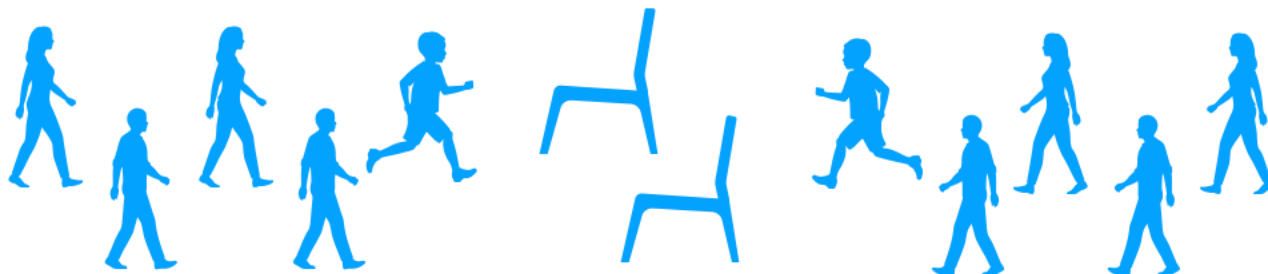


进入这些程序段的进程数 $\leq n$

3. 用信号量实现受控并发

■ 在线课堂练习：实训5.1

程序4.c模拟了如下场景：某休息厅里有足够多（10把以上）的椅子，10位顾客先后进入休息厅寻找空椅子，找到后开始在椅子上休息，休息完后让出空椅子、退出休息厅。请只在该程序中插入一些代码，来将上述场景调整为休息厅里只有2把椅子。



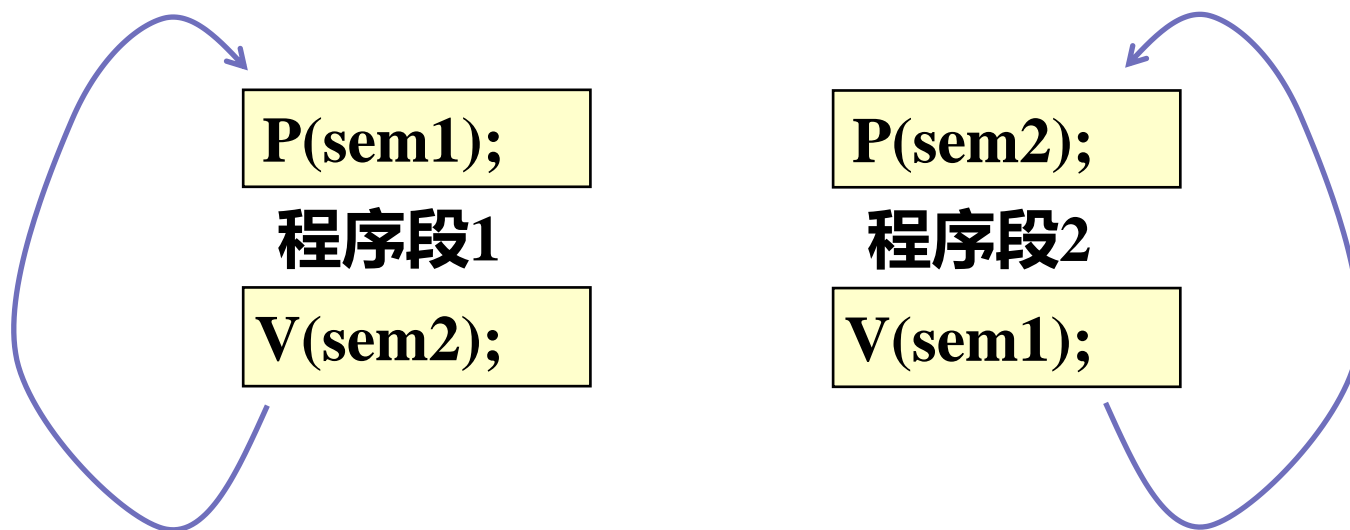
目录

- 一. 信号量的概念
- 二. 用信号量实现互斥和同步
- 三. 用信号量实现受控并发
- 四. 受控并发的其他形式

4. 受控并发的其他形式

● 受控并发的其他形式

➡ semaphore sem1=1, sem2=0;

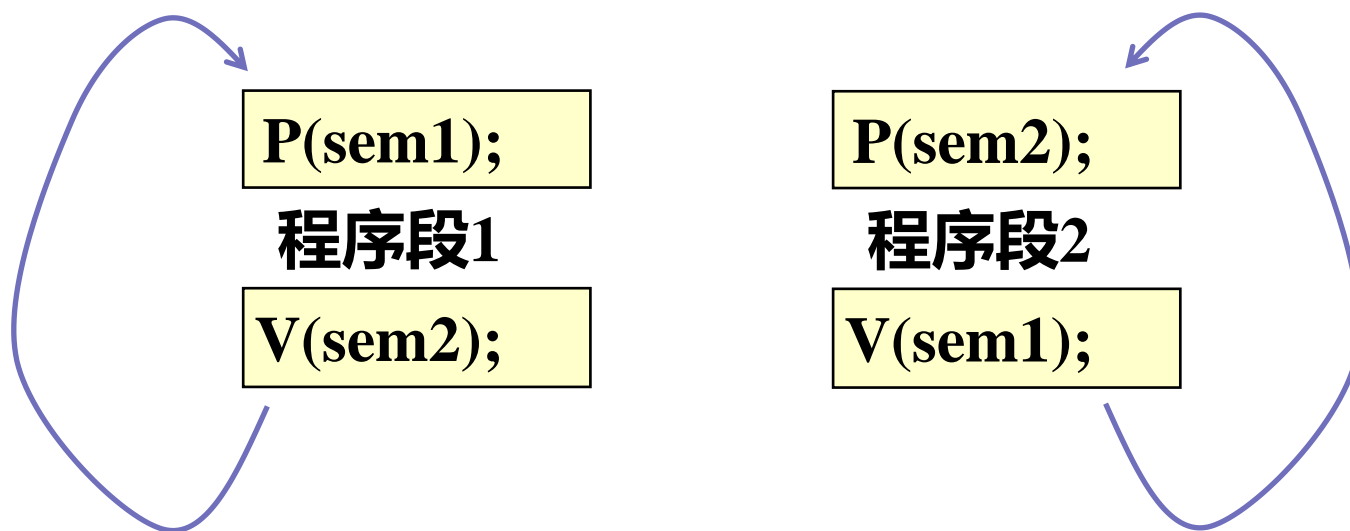


程序段1, 程序段2, 程序段1, 程序段2, 程序段1, 程序段2, ...

4. 受控并发的其他形式

● 受控并发的其他形式

➡ semaphore sem1=3, sem2=0;

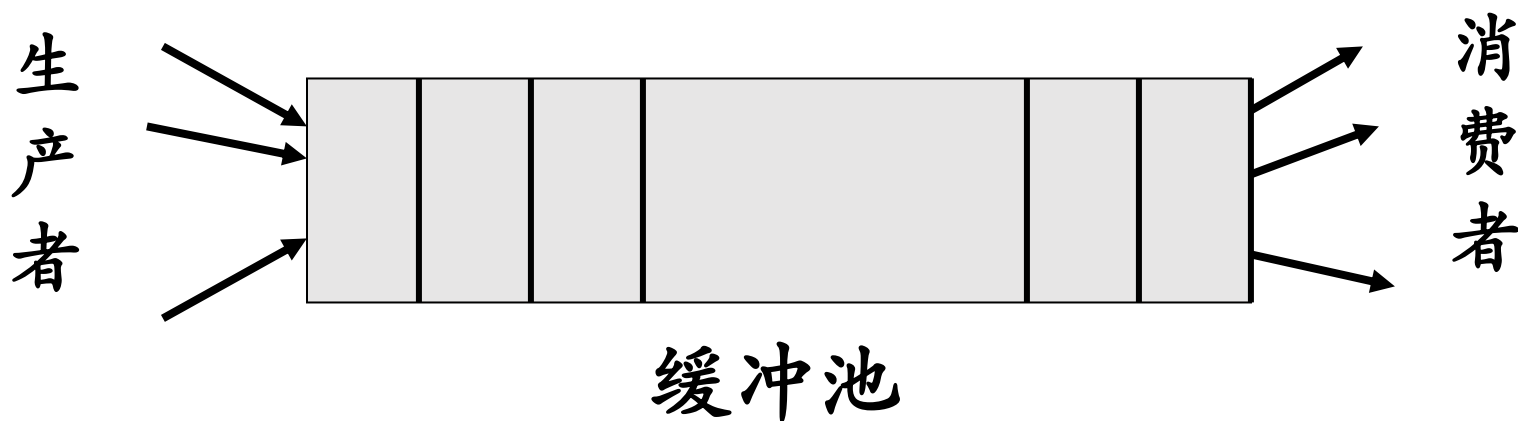


程序段1, 程序段1, 程序段1, 程序段1, 程序段1, 程序段1, ...
程序段2, 程序段2, 程序段2, 程序段2, 程序段2, ...

$0 \leq \text{程序段1运行的次数} - \text{程序段2运行的次数} \leq 3$

例1：生产者/消费者问题

- 生产者和消费者共享一个产品缓冲池，其中包含N个缓冲块。



- 如果N个缓冲区全满，生产者进程必须等待。
- 如果缓冲区全空，消费者进程必须等待。

```

... /* 6.c */
#include <semaphore.h>
#define N 10
#define PRODUCT_NUM 15
int buffer[N], readpos=0, writepos=0;
semaphore full=0, empty=N;

void produce() {
    for (int i=0; i<PRODUCT_NUM; i++) {
        P(empty);
        buffer[writepos++] = i + 1;
        if (writepos >= N) writepos = 0;
        V(full); } }

void consume() {
    for (int i=0; i<PRODUCT_NUM; i++) {
        P(full);
        printf("consume:%d\n", buffer[readpos]);
        buffer[readpos++] = -1;
        if (readpos >= N) readpos = 0;
        V(empty); } }

```

生产者\消费者问题


```

... /* 6.c */
#include <semaphore.h>
#define N 10
#define PRODUCT_NUM 10
int buffer[N], readpos=0;
semaphore full=0, empty=N;

void produce() {
    for (int i=0; i<PRODUCT_NUM; i++) {
        P(empty);
        buffer[writepos++] = i + 1;
        if (writepos >= N) writepos = 0;
        V(full); } }

void consume() {
    for (int i=0; i<PRODUCT_NUM; i++) {
        P(full);
        printf("consume:%d\n", buffer[readpos]);
        buffer[readpos++] = -1;
        if (readpos >= N) readpos = 0;
        V(empty); } }

```

```

int main() {
    int i;
    for (i=0; i<NUM; i++)
        buffer[i] = -1;
    Parbegin
        produce();
        consume();
    Parend
}

```

一个生产者/一个消费者

演示...

```

...    /* 7.c */
#include <semaphore.h>
#define N 10
#define PRODUCT_NUM 15
int buffer[N], readpos=0, writepos=0;
semaphore full=0, empty=N;

void produce(int id) {
    for (int i=0; i<PRODUCT_NUM; i++) {
        P(empty);
        buffer[writepos++] = 1000*(id-1)+i+1;
        if (writepos >= N) writepos = 0;
        V(full); } }

void consume() {
    for (int i=0; i<2*PRODUCT_NUM; i++) {
        P(full);
        printf("consume:%d\n", buffer[readpos]);
        buffer[readpos++] = -1;
        if (readpos >= N) readpos = 0;
        V(empty); } }

```

两个生产者 / 一个消费者

```

...    /* 7.c */
#include <semaphore.h>
#define N 10
#define PRODUCT_NUM 1
int buffer[N], readpos=0;
semaphore full=0, empty=N;

void produce(int id) {
    for (int i=0; i<PRODUCT_NUM; i++) {
        P(empty);
        buffer[writepos++] = id;
        if (writepos >= N) writepos = 0;
        V(full); } }

void consume() {
    for (int i=0; i<2*PRODUCT_NUM; i++) {
        P(full);
        printf("consume:%d\n", buffer[readpos]);
        buffer[readpos++] = -1;
        if (readpos >= N) readpos = 0;
        V(empty); } }

```

```

int main() {
    int i;
    for (i=0; i<NUM; i++)
        buffer[i] = -1;
    Parbegin
        produce(1);
        produce(2);
        consume();
    Parend
}

```

两个生产者/一个消费者

演示...

```
...    /* 8.c */  
int buffer[N], readpos=0, writepos=0;  
semaphore full=0, empty=N, mutex=1;  
  
void produce(int id) {  
    for (int i=0; i<PRODUCT_NUM; i++) {  
        P(empty);  
        P(mutex);  
        buffer[writepos++] = 1000*(id-1)+i+1;  
        if (writepos >= N) writepos = 0;  
        V(mutex);  
        V(full); } }  
  
void consume() {  
    for (int i=0; i<2*PRODUCT_NUM; i++) {  
        P(full);  
        printf("consume:%d\n", buffer[readpos]);  
        buffer[readpos++] = -1;  
        if (readpos >= N) readpos = 0;  
        V(empty); } }
```

生产者间互斥访问

两个生产者、一个消费者

演示...

小结

一. 信号量的概念

P、V操作

二. 用信号量实现互斥和同步

三. 用信号量实现受控并发

休息厅问题

四. 受控并发的其他形式

生产者/消费者问题

后续

1. 并发执行的实现
2. 进程的同步与互斥
 - ① 互斥与临界段问题
 - ② 解决临界段问题的硬件方法
 - ③ 解决临界段问题的信号量方法
 - ④ 信号量的应用: 生产者/消费者问题
3. 死锁
4. 消息传递原理



操作系统

第四章 第3讲

死锁与消息传递原理

文艳军（教授）
计算机学院

回顾

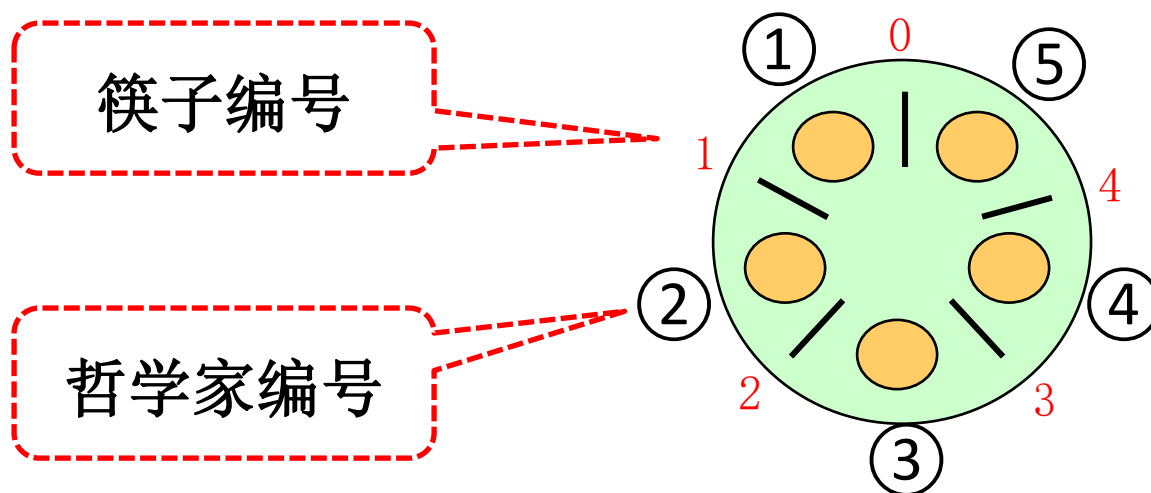
1. 并发执行的实现
2. 进程的同步与互斥
 - ① 互斥与临界段问题
 - ② 解决临界段问题的硬件方法
 - ③ 解决临界段问题的信号量方法
 - ④ 信号量的应用：生产者/消费者问题
3. 死锁
4. 消息传递原理

目录

- 一. 死锁的概念
- 二. 死锁防止
- 三. 死锁避免
- 四. 消息传递原理

一. 死锁的概念——哲学家就餐问题

- 五个哲学家，五只筷子
- 哲学家不停地思考和吃饭
- 只有拿到左右两支筷子才能吃饭



```
...    /* 9.c */
#define N 5
semaphore chopstick[N];

void phi(int id) { /* id从1开始 */
    int i, left, right;
    left = id - 1;
    right = (id < N)? id : 0;
    for (i=0; i<3; i++) {
        printf("phi #%d: thinking\n", id);

        P(chopstick[left]);
        P(chopstick[right]);
        printf("phi #%d: eating\n", id);
        V(chopstick[left]);
        V(chopstick[right]);
    } }
```

计算筷子下标

```

...    /* 9.c */
#define N 5
semaphore chopstick[N];

void phi(int id) { /* i
    int i, left, right;
    left = id - 1;
    right = (id < N)? id
    for (i=0; i<3; i++) {
        printf("phi #%d: t

```

```

int main() {
    for(int i=0;i<N;i++)
        chopstick[i]=1;
    Parbegin
        phi(1);
        phi(2);
        phi(3);
        phi(4);
        phi(5);
    Parend
}

```

```

P(chopstick[left]);

```

```

P(chopstick[right]);

```

```

printf("phi #%d: eating\n", id);

```

```

V(chopstick[left]);

```

```

V(chopstick[right]);

```

```

} }

```

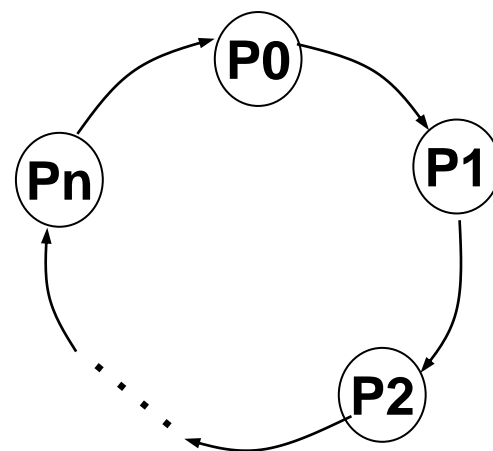
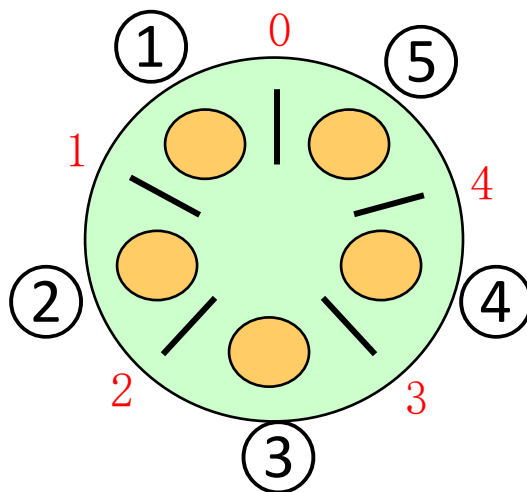
可能死锁

演示...

死锁的定义

- 死锁：在一个**进程集合**中，若**每个进程**都在**等待**某些**资源**，而这些资源又必须由该进程集合中的进程来**释放**，导致存在无法解决的**资源等待环**而相互锁定。

当哲学家同时拿起左边的筷子、等待右边的筷子时...



死锁的四个必要条件

- 互斥占用：存在必须互斥使用的资源
- 占有等待：存在占有资源而又等待其它资源的进程
- 非剥夺：进程占有的资源未主动释放时不可以被剥夺
- 循环等待

死锁的应对方法

- ① 死锁防止
 - ② 死锁避免
 - ③ 死锁检测
 - ④ 死锁恢复
- 无死锁系统**
- 允许死锁发生但能够及时排除死锁**

目录

- 一. 死锁的概念
- 二. 死锁防止
- 三. 死锁避免
- 四. 消息传递原理

二. 死锁防止

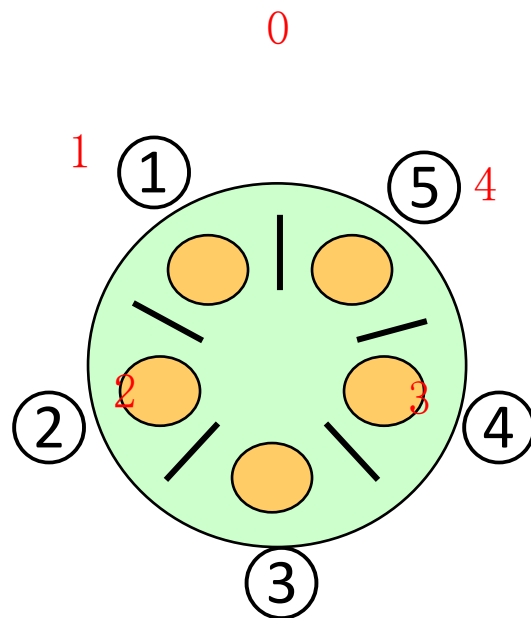
■ 破坏死锁的四个必要条件之一

□ 互斥占用

□ 占有等待

□ 非剥夺

□ 循环等待



破坏互斥等待条件：
多元信号量

```
...
#define N 5
semaphore chopstick[N];
semaphore mutex;
void phi(int id) { /* id从1开始 */
    int i, left, right;
    left = id - 1;
    right = (id < N)? id : 0;
    for (i=0; i<3; i++) {
        printf("phi #%d: thinking\n", id);
        P(chopstick[left], chopstick[right]);
        printf("phi #%d: eating\n", id);
        V(chopstick[left], V(chopstick[right]);
    }
}
```

```
...    /* 10.c */
#define N 5
semaphore chopstick[N];
semaphore mutex;
void phi(int id) { /* id从1开始 */
    int i, left, right;
    left = id - 1;
    right = (id < N)? id : 0;
    for (i=0; i<3; i++) {
        printf("phi #%d: thinking\n", id);
        P(mutex);
        P(chopstick[left]);
        P(chopstick[right]);
        V(mutex);
        printf("phi #%d: eating\n", id);
        V(chopstick[left]);
        V(chopstick[right]);    } }
```



二. 死锁防止

■ 破坏死锁的四

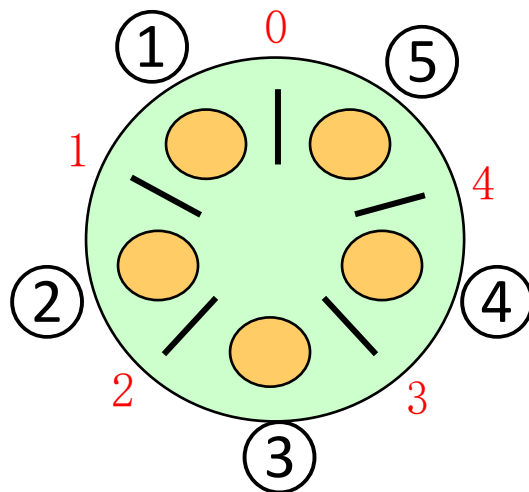
□ 互斥占用

□ 占有等待

□ 非剥夺

□ 循环等待

资源顺序分配法：给每类资源编号，进程**只能按序号由小到大的顺序**申请资源，若不满足则拒绝分配



```
...    /* 9.c */  
#define N 5  
semaphore chopstick[N]
```

```
void phi(int id) {  
    int i, left, right;  
    left = id - 1;  
    right = (id < N)? id : 0;  
    for (i=0; i<3; i++) {  
        printf("phi #%d: thinking\n", id);
```

```
        P(chopstick[left]);
```

```
        P(chopstick[right]);
```

```
        printf("phi #%d: eating\n", id);
```

```
        V(chopstick[left]);
```

```
        V(chopstick[right]);
```

```
    } }
```

资源顺序分配法：给每类资源编号，进程只能按序号由小到大的顺序申请资源，若不满足则拒绝分配



二. 死锁防止

■ 课内实验：实训5-第2关

请用**资源顺序分配法**解决
哲学家就餐问题。

目录

- 一. 死锁的概念
- 二. 死锁防止
- 三. 死锁避免
- 四. 消息传递原理

三. 死锁避免

- 前提：预先知道所有进程的资源总需求
- 在处理资源申请时，检查系统是否在满足申请后仍然处于安全状态？是则满足本次资源申请，否则拒绝。
- 安全状态：存在一种资源分配顺序，保证每个进程能获得足够的资源完成运行。

三. 死锁避免

● 银行家算法

- 例子：设银行家有10万贷款，P, Q, R分别需要8, 3, 9万元做项目（假设任何人获得资金总额后都会归还所有贷款），P已获得了4万贷款。这时，Q申请2万，R申请4万。

	总需求	已分配								
P	8	4	4	8	0	0	0	0	0	0
Q	3	0	2	2	2	3	0	0	0	0
R	9	0	0	0	0	0	0	9	0	0
	剩余	6	4	0	8	7	10	1	10	

安全序列：P→Q→R



死锁避免

● 扩展的银行家算法

安全序列: $R \rightarrow P \rightarrow Q$

	总需求	已分配				
P	2, 5, 8	1, 2, 4	1, 2, 4	1, 2, 4	1, 2, 4	2, 5, 8
Q	4, 4, 4	0, 3, 3	0, 3, 3	0, 3, 3	0, 3, 3	0, 3, 3
R	5, 4, 4	4, 1, 1	5, 3, 3	5, 4, 4	0, 0, 0	0, 0, 0
剩余		1, 3, 3	0, 1, 1	0, 0, 0	5, 4, 4	4, 1, 0

R请求 1, 2, 2

P请求 1, 2, 1



死锁避免

● 扩展的银行家算法

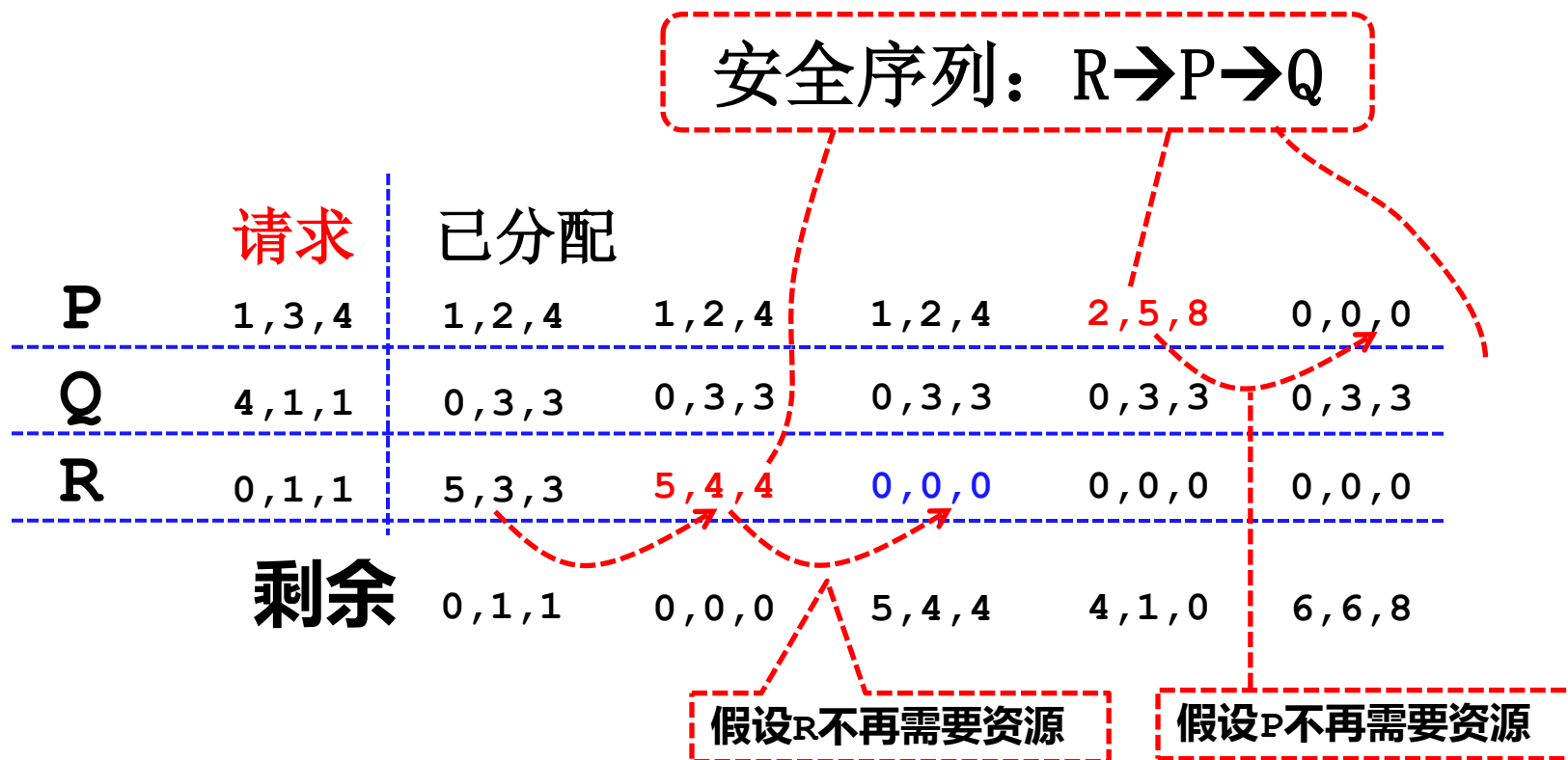
	总需求	已分配	
P	2, 5, 8	1, 2, 4	2, 4, 5
Q	4, 4, 4	0, 3, 3	0, 3, 3
R	5, 4, 4	4, 1, 1	4, 1, 1
剩余		1, 3, 3	0, 1, 2

P请求 1, 2, 1



死锁检测

● 算法类似于死锁避免



死锁的综合处理

把系统中的资源分成几大类，**整体上采用资源顺序分配法**，再对每类资源根据其特点选择最适合的方法。

例如：

- (1) 主存、处理机 -- 剥夺法
- (2) 辅存 -- 预分配法
- (3) 其他 -- 人工检测后处理

实用预防死锁方法：设立资源阈值，当资源少于阈值时限制进程申请，减少申请不到资源的概率。

死锁的必要条件有：

- ☒ A 互斥占用
- ☒ B 占有等待
- ☐ C 循环占用
- ☒ D 非剥夺
- ☒ E 循环等待

提交

目录

- 一. 死锁的概念
- 二. 死锁防止
- 三. 死锁避免
- 四. 消息传递原理

四. 消息传递原理

进程通信方法

共享存储（共享内存）

消息传递（消息队列）

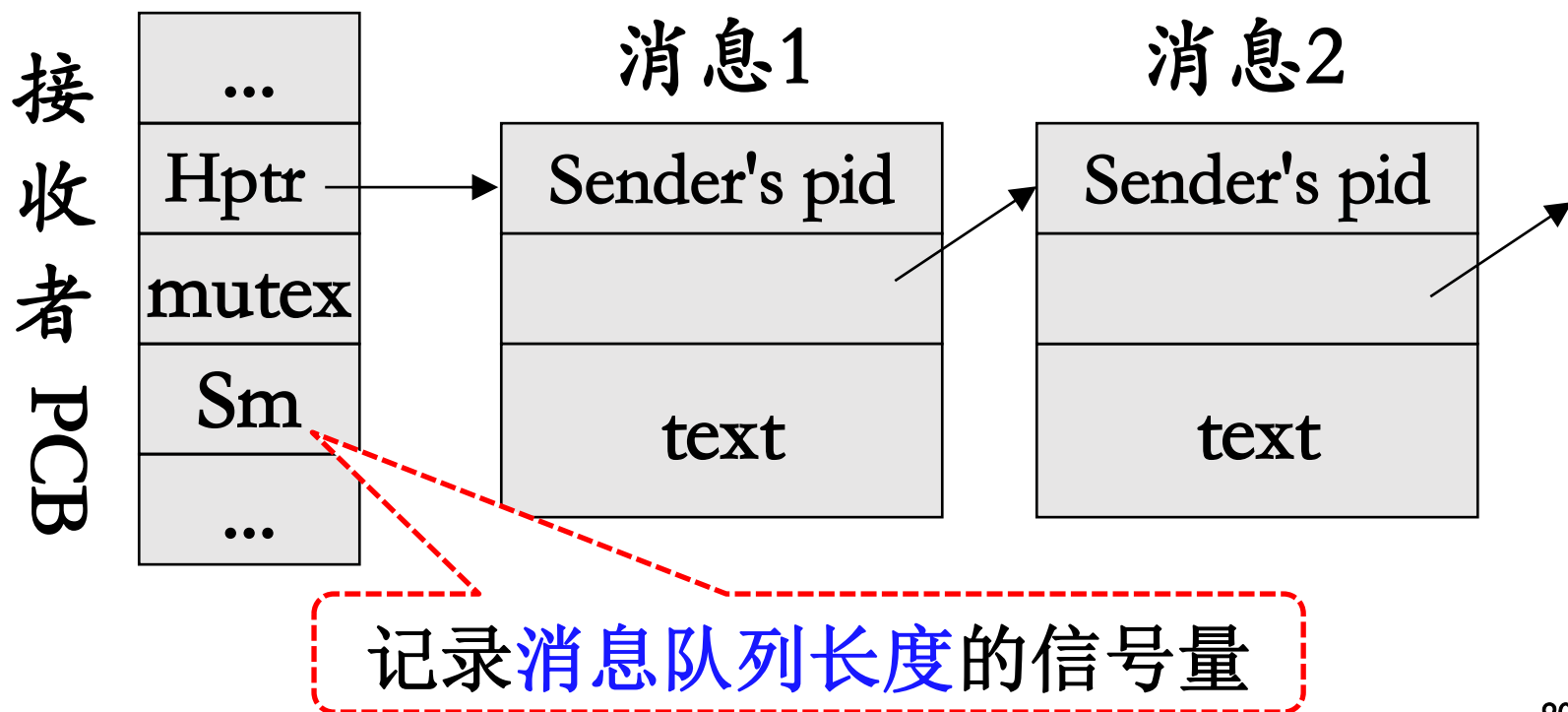
信号量（Semaphore）

信号（Signal）

管道（Pipe）

消息队列

- 两个基本操作：send()、receive()，以消息包为单位
- 相关的进程数据结构：



管道

- 通信方式：FIFO、**字节流**
- 管道缓冲区空间有限，管道的读写可能**阻塞**进程



Linux 0.11

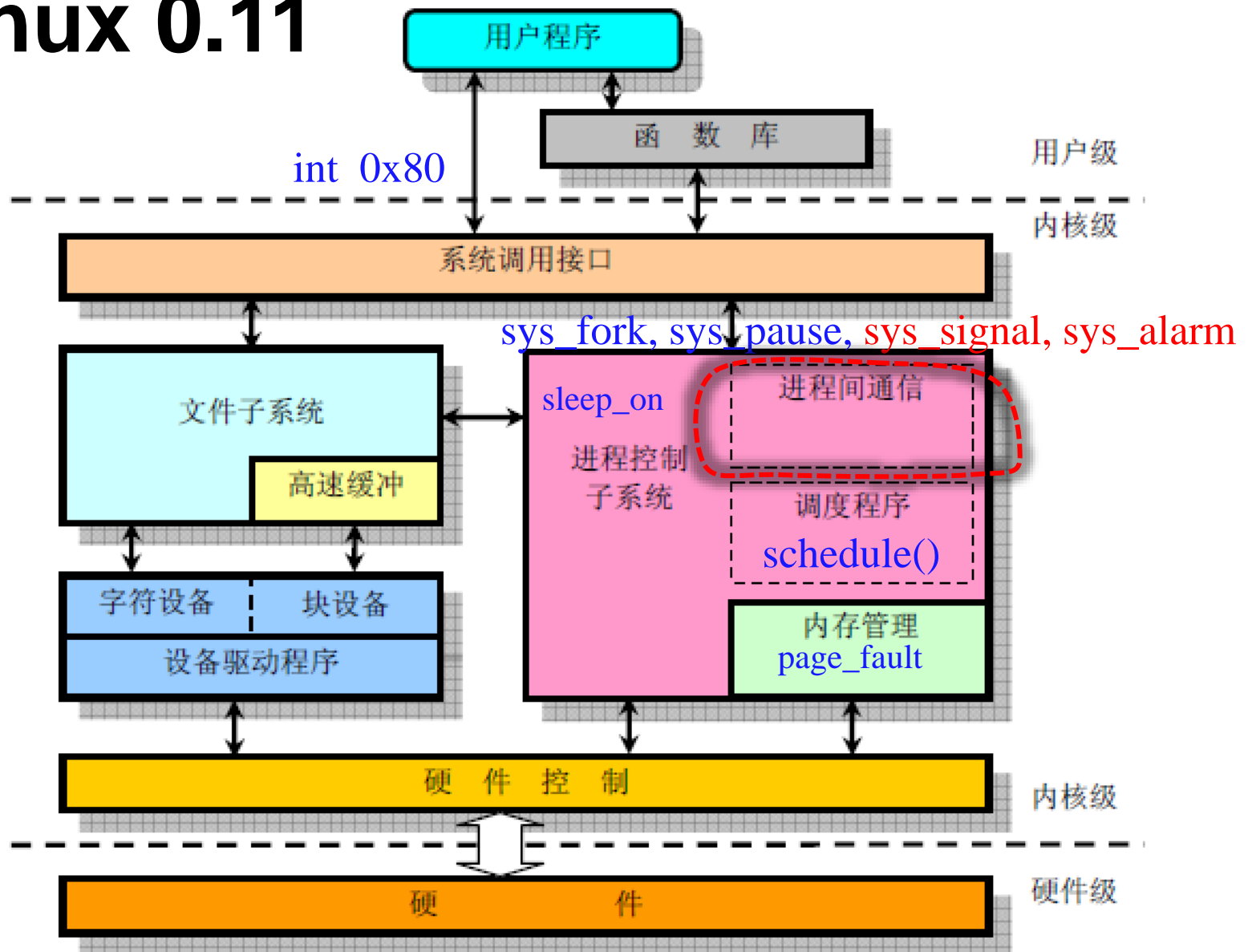


图 2-4 内核结构框图

在银行家算法中，什么叫安全状态？ [填空1]

正常使用填空题需3.0以上版本雨课堂

作答

小结

1. 并发执行的实现

2. 进程的同步与互斥

3. 死锁

死锁防止（资源顺序分配法）、
死锁避免、死锁检测、死锁恢复

4. 消息传递原理

消息队列、管道