

Operating Systems

Chapter 5 Mutual Exclusion(互斥) and Synchronization(同步)

Agenda

- 5.1 Principles of Concurrency 并发
- 5.2 Mutual Exclusion: Hardware Support
- 5.3 Semaphores
- 5.4 Monitors
- 5.5 Message Passing
- 5.6 Readers/Writers Problem
- 5.7 Summary

5.1 Principles of Concurrency

- 5.1.1 A Simple Example of Concurrency
- 5.1.2 Race Condition
- 5.1.3 Different Concerns
- 5.1.4 Requirements for Mutual Exclusion

5.1.1 A Simple Example of Concurrency(1/18)

- Concurrency 并发 arises in three different contexts:
 - Multiple applications(多应用程序)
 - Multiprogramming
 - Structured application(结构化应用程序)
 - Some application can be designed as a set of concurrent processes
 - Operating-system structure(操作系统结构)
 - Operating system is a set of processes or threads

5.1.1 A Simple Example of Concurrency(2/18)

- Example : allocate pid for two new processes when call fork()
 - next_pid is a global variable, whose value is now 100
 - new_pid = next_pid++
 - Machine code
 - LOAD next_pid Reg1 #next_pid 源 — > Reg1 目标
 - STORE Reg1 new_pid
 - INC Reg1
 - STORE Reg1 next_pid
- Q: pid1 = ? and pid2 = ?

5.1.1 A Simple Example of Concurrency(3/18)

- process1

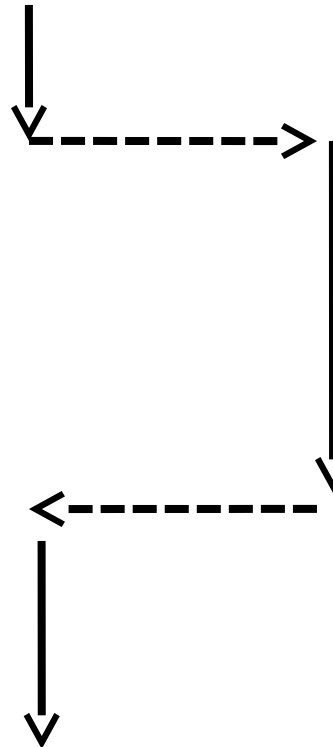
-
- LOAD next_pid Reg1
- STORE Reg1 new_pid
-
- INC Reg1
- STORE Reg1 next_pid

pid1=100
nexp_pid is 101 now

- process2

- LOAD next_pid Reg1
- STORE Reg1 new_pid
- INC Reg1
- STORE Reg1 next_pid

pid1=100
nexp_pid is 101 now



5.1.1 A Simple Example of Concurrency(4/18)

- Q : possible output?

- 全局变量 `int i=0;`

- `while(i<10){`

`i++;`

`printf("A wins\n");`

`}`

`while(i<10){`

`i--;`

`printf("B wins\n");`

`}`

运行程序 `increasei.c` 观察可能的输出

5.1.1 A Simple Example of Concurrency(5/18)

- `int main()`
- `{`
- `pthread_t t1, t2;`
- `pthread_create(&t1, NULL, P1, NULL/*(void*)1*/);`
- `pthread_create(&t2, NULL, P2, NULL/*(void*)2*/);`
- `pthread_join(t1, NULL);`
- `pthread_join(t2, NULL);`
- `return 0;`
- `}`

5.1.1 A Simple Example of Concurrency(6/18)

```
int a=1,b=1;
void* P1(void *arg)
{
    int i=0;
    while(i<20){
        i++;
        a=a+1;
        sleep(1);
        b=b+1;    printf("P1:a=
%d,b=%d\n",a,b);
    }
}
```

```
void* P2(void *arg)
{
    int i=0;
    while(i<20){
        i++;
        b=2*b;
        sleep(1);
        a=2*a;    printf("P2:a=
%d,b=%d\n",a,b);
    }
}
```

5.1.1 A Simple Example of Concurrency(7/18)

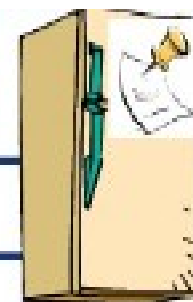
- 为啥代码先 create 线程 p1, 后创建 p2 , 但先执行的是 p2 ?
 - 1. 线程创建后会进入就绪态等待执行
 - 2. 可能存在多核 , 不能肯定那个线程先执行
- 为啥 p2 全部执行完了才是 p1
- 主要看 p2 的全部执行是否会用完一次时间片 , 另外当有更高优先级线程来时 , 也会发生抢占

5.1.1 A Simple Example of Concurrency(8/18)

- 出现下列输出的原因：后输出的 b 反而更小
- $a=3, b=3$
- $a=3, b=2$
- P2 先执行一次，得到 $a=2, b=2$ ，随后执行打印指令 `printf(...,a,b);` 由于压入堆栈的时间是先 b, 后 a。当 b 压入堆栈后发生切换
- P1 执行 +1 操作并打印输出，得到 $a=3, b=3$ ，此时再切换回 P2
- P2 压入此时的 $a=3$ ，再输出就得到 $a=3, b=2$

5.1.1 A Simple Example of Concurrency(9/18)

- 似乎没有任何头绪... 可借鉴生活中的道理



时间	丈夫	妻子
3:00	打开冰箱, 没有牛奶了	
3:05	离开家去商店	
3:10	到达商店	打开冰箱, 没有牛奶了
3:15	买牛奶	离开家去商店
3:20	回到家里, 牛奶放进冰箱	到达商店
3:25	看电视 or 洗碗	买牛奶
3:30		回到家里, 牛奶放进冰箱

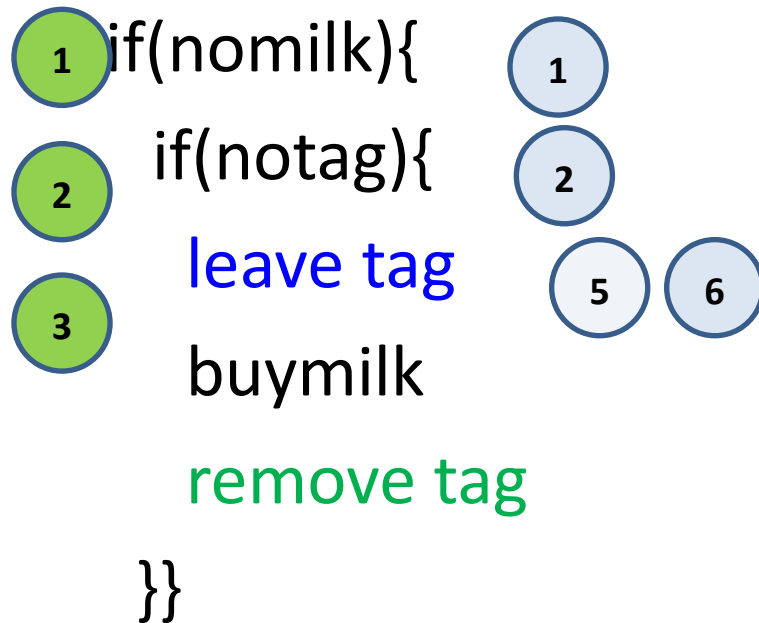


5.1.1 A Simple Example of Concurrency(10/18)

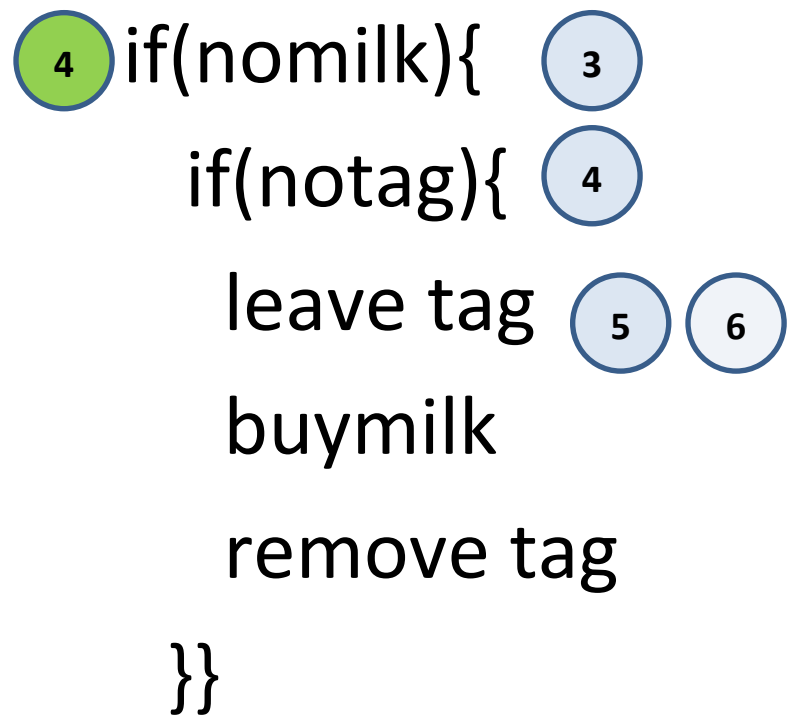
- Basic problem: buy milk
- if(nomilk)
- buymilk
- if(nomilk)
- buymilk

5.1.1 A Simple Example of Concurrency(11/18)

- Using tag (lock – unlock) buy milk: improvement 1



```
1 if(nomilk){
2   if(notag){
3     leave tag
      buymilk
      remove tag
  }
}
```



```
4 if(nomilk){
      if(notag){
        leave tag
        buymilk
        remove tag
      }
}
```

5.1.1 A Simple Example of Concurrency(12/18)

- Using tag (lock – unlock) buy milk: improvement 2

```
1 if(notag) {  
3   leave tag  
•  
4   if(nomilk){  
       buymilk}  
       remove tag  
}
```

```
2 if(notag) {  
5   leave tag  
•   if(nomilk){  
       buymilk}  
       remove tag  
}
```

5.1.1 A Simple Example of Concurrency(13/18)



- Using tag (lock – unlock) buy milk: improvement 3





Dead lock

5.1.1 A Simple Example of Concurrency(14/18)

- Using tag (lock – unlock) buy milk: improvement 4

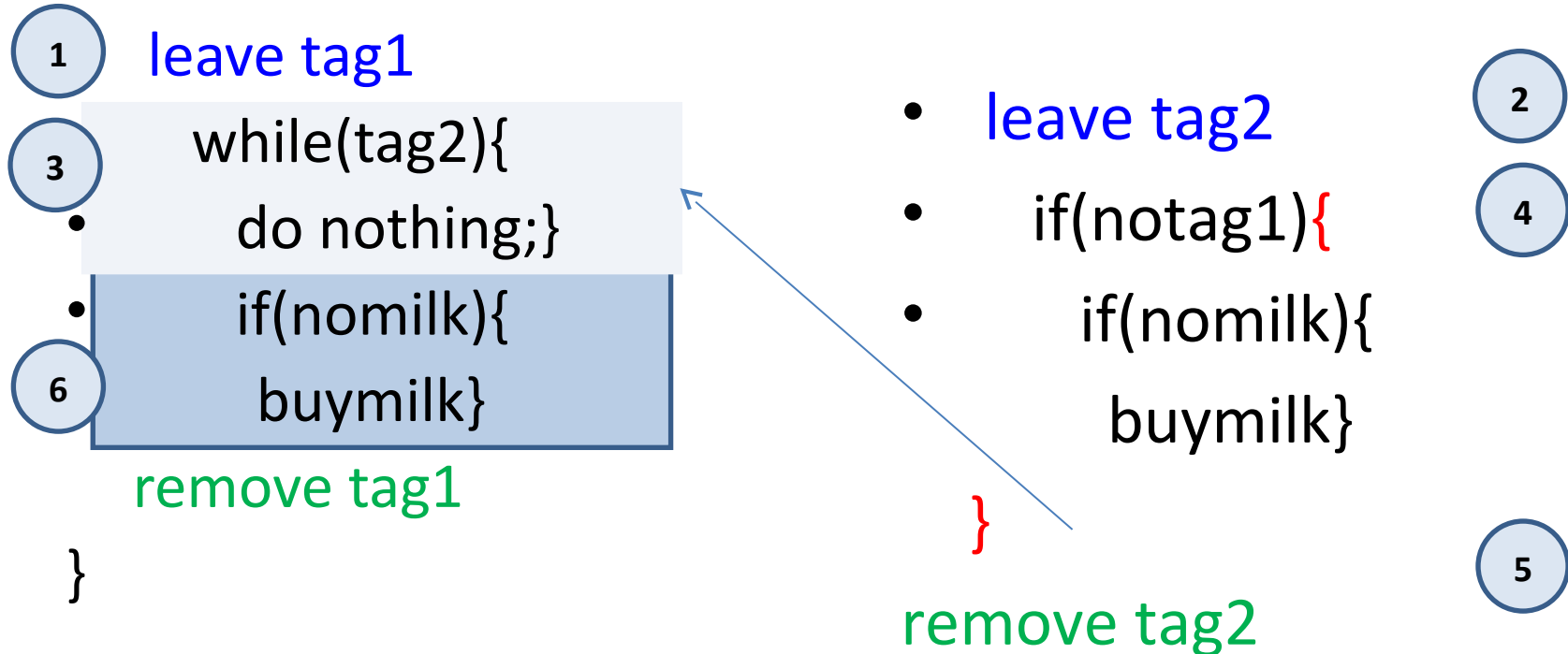
- leave tag1 
- if(notag2){ 
- if(nomilk){
 buymilk}
- remove tag1
- }

- leave tag2 
- if(notag1){ 
- if(nomilk){
 buymilk}
- remove tag2

Dead lock

5.1.1 A Simple Example of Concurrency(15/18)

- Using tag (lock – unlock) buy milk: improvement 5
- 退让等待



5.1.1 A Simple Example of Concurrency(16/18)

- 问题的本质：
 - Competition Among Processes for Resources(进程间的资源争用)
 - 线程的切换不确定，导致需要连续执行的几条指令被分开执行
- 问题的现象
 - Deadlock(死锁)
 - Starvation(饥饿)
 - 执行结果不确定

5.1.1 A Simple Example of Concurrency(17/18)

- 解决方法：
 - Mutual Exclusion(互斥)
 - Critical sections(临界区)
 - Only one program at a time is allowed in its critical section 某时只允许一个程序访问临界区
 - E.g. only one process at a time is allowed to send command to the printer

5.1.1 A Simple Example of Concurrency(18/18)

Table 5.1 Some Key Terms Related to Concurrency

临界区	critical section	A section of code within a process that requires access to shared resources and which may not be executed while another process is in a corresponding section of code.
死锁	deadlock	A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.
活锁	livelock	A situation in which two or more processes continuously change their state in response to changes in the other process(es) without doing any useful work.
互斥	mutual exclusion	The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.
竞争条件	race condition	A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution.
饥饿	starvation	A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.

5.1 Principles of Concurrency

- 5.1.1 A Simple Example of Concurrency
- 5.1.2 Race Condition
- 5.1.3 Different Concerns
- 5.1.4 Requirements for Mutual Exclusion

5.1.2 Race Condition(竞争条件)(1/3)

A race condition occurs when multiple processes or threads read and write data items so that the final result depends on the order of execution of instructions in the multiple processes or thread.

(竞争条件发生在当多个进程或者线程在读写数据时，其最终结果依赖于多个进程或者线程的指令执行顺序)

5.1.2 Race Condition(竞争条件) (2/3)

The image displays two side-by-side terminal windows on a Linux system. Both windows show the execution of a C program that creates two threads, thread1 and thread2. In the left window, thread2 prints 'wowowowowo!' and thread1 prints 'hahaha!'. In the right window, thread1 prints 'hahaha!' and thread2 prints 'wowowowowo!'. The output demonstrates that the threads execute concurrently. The terminal windows have a dark background with light-colored text. The left window's title bar includes icons for a terminal, a file manager, a text editor, and a web browser. The right window's title bar includes icons for a terminal, a file manager, a text editor, a web browser, and a stack of documents.

• 5.1.2 Race Condition(竞争条件)(3/3)

• Process Interaction(进程交互)

• 3 类进程间感知程度

Table 5.2 Process Interaction

Degree of Awareness	Relationship	Influence that one Process has on the Other	Potential Control Problems
Processes unaware of each other	Competition	<ul style="list-style-type: none"> •Results of one process independent of the action of others •Timing of process may be affected 	<ul style="list-style-type: none"> •Mutual exclusion •Deadlock (renewable resource) •Starvation
Processes indirectly aware of each other (e.g., shared object)	Cooperation by sharing	<ul style="list-style-type: none"> •Results of one process may depend on information obtained from others •Timing of process may be affected 	<ul style="list-style-type: none"> •Mutual exclusion •Deadlock (renewable resource) •Starvation •Data coherence
Processes directly aware of each other (have communication primitives available to them)	Cooperation by communication	<ul style="list-style-type: none"> •Results of one process may depend on information obtained from others •Timing of process may be affected 	<ul style="list-style-type: none"> •Deadlock (consumable resource) •Starvation

5.1 Principles of Concurrency

- 5.1.1 A Simple Example of Concurrency
- 5.1.2 Race Condition
- 5.1.3 Different Concerns
- 5.1.4 Requirements for Mutual Exclusion

5.1.3 Different Concerns(1/2)

- Programmer Concerns:
 - Difficulties of Concurrency
 - Sharing of global resources
 - Operating system managing the allocation of resources optimally
 - Difficult to locate programming errors

5.1.3 Different Concerns(2/2)

- Operating System Concerns
 - Keep track of various processes(through PCB)
 - Allocate and deallocate resources
 - Processor time/Memory/Files/ I/O devices
 - Protect data and resources
 - Output of process must be independent of the speed of execution of other concurrent processes

5.1 Principles of Concurrency

- 5.1.1 A Simple Example of Concurrency
- 5.1.2 Race Condition
- 5.1.3 Different Concerns
- 5.1.4 Requirements for Mutual Exclusion

5.1.4 Requirements for Mutual Exclusion(1/3)

互斥的要求

1. Only one process at a time is allowed in the critical section for a resource (一次只允许一个进程进入临界区，忙则等待)
2. A process that halts in its noncritical section must do so without interfering with other processes (阻塞于临界区外的进程不能干涉其它进程)
3. No deadlock or starvation (不会发生饥饿和死锁，有限等待)

5.1.4 Requirements for Mutual Exclusion(2/3)

4. A process must not be delayed access to a critical section when there is no other process using it (闲则让进)
5. No assumptions are made about relative process speeds or number of processors (对相关进程的**执行速度**和 CPU 数目没有要求)
6. A process remains inside its critical section for a finite time only (**有限占用**)

5.1.4 Requirements for Mutual Exclusion(3/3)

.entercritical : 进入临界区 .exitcritical : 退出临界区

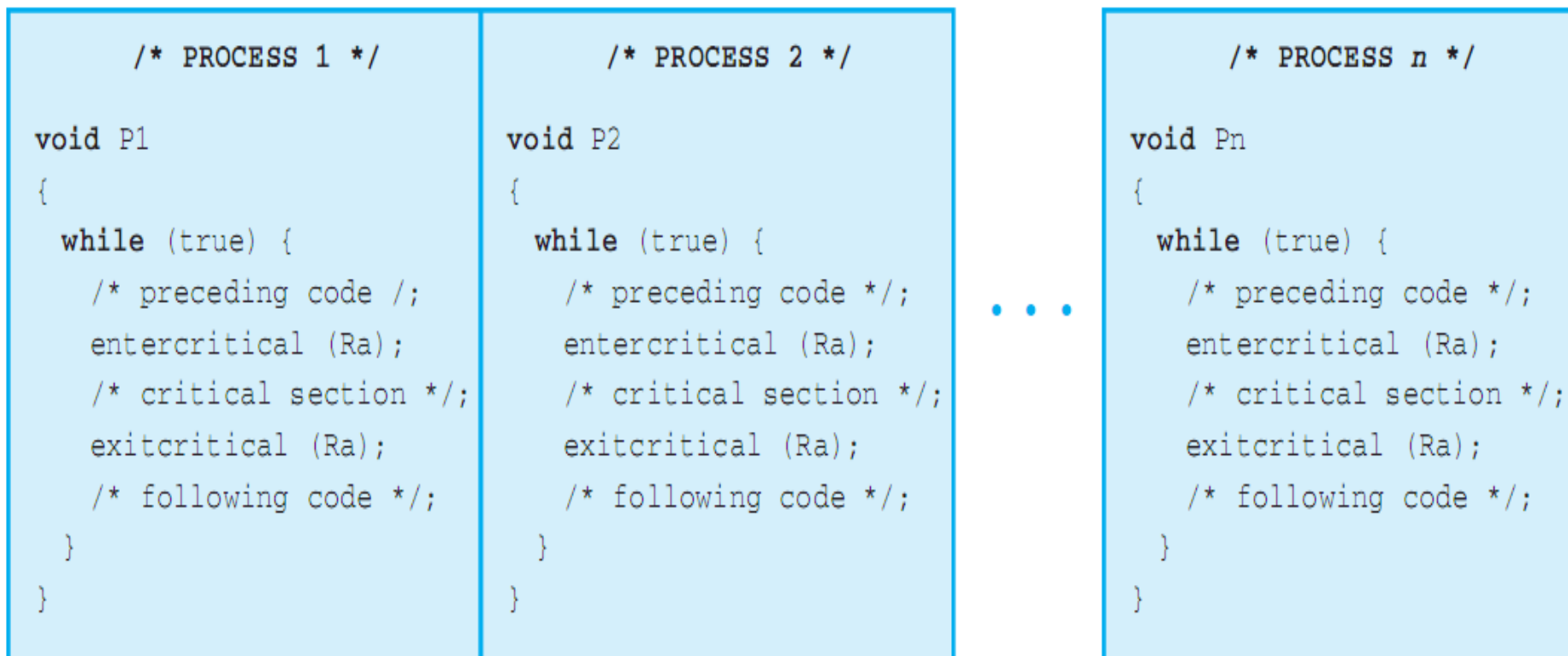


Figure 5.1 Illustration of Mutual Exclusion

互斥机制示例

Agenda

- 5.1 Principles of Concurrency
- 5.2 Mutual Exclusion: Hardware Support
- 5.3 Semaphores
- 5.4 Monitors
- 5.5 Message Passing
- 5.6 Readers/Writers Problem
- 5.7 Summary

5.2 Mutual Exclusion: Hardware Support

- 5.2.1 Hardware approaches
 - 5.2.1 Interrupt Disabling
 - 5.2.2 Special Machine Instructions
- 5.2.2 Software approaches

5.2.1 Interrupt Disabling(1/2)

```
while (true) {  
    /* disable interrupts */;  
    /* critical section */;  
    /* enable interrupts */;  
    /* remainder */;  
}
```

- CPU 内部 PSW 寄存器，使用：
STI 使 IF 置 "1"，即开放中断。
CLI 使 IF 清 "0"，即关闭中断
- Linux 内核使用 cli() 和 sti()

5.2.1 Interrupt Disabling(2/2)

- A process runs until it invokes an operating system service or until it is interrupted
- Disabling interrupts guarantees mutual exclusion on **uniprocessor system**
- Disadvantage:
 - Processor is limited in its ability to interleave programs
 - disabling interrupts on one processor will **not guarantee** mutual exclusion in **multi-processors** environment.

5.2 Mutual Exclusion: Hardware Support

- 5.2.1 Hardware approaches
 - 5.2.1 Interrupt Disabling
 - 5.2.2 Special Machine Instructions
- 5.2.2 Software approaches

5.2.2 Special Machine Instructions(1/12)

- Atomic (原子) Operation
 - Performed in a single instruction cycle
 - access to the memory location is blocked for any other instructions
 - carried out atomically

5.2.2 Special Machine Instructions(2/12)

- Test and Set Instruction (text edition 5)
- **bolt** 插销 是全局变量

```
boolean testset (int &bolt) {  
    if (bolt == 0) {  
        bolt = 1;  
        return true;  
    }  
    else {  
        return false;  
    }  
}
```

5.2.2 Special Machine Instructions(3/12)

```
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
    while (true)
    {
        while (!testset (bolt))
            /* do nothing */;
        /* critical section */
        bolt = 0;
        /* remainder */
    }
}
void main()
{
    bolt = 0;
    parbegin (P(1), P(2), . . . ,P(n));
}
}
```

(a) Test and set instruction

5.2.2 Special Machine Instructions(4/12)

- Compare and swap : 思想

```
int compare_and_swap(int *bolt, int testval, int newval)
{
    int oldval;
    oldval = *bolt;
    if ( oldval == testval) *bolt= newval;
    return oldval;
}
```

5.2.2 Special Machine Instructions(5/12)

- 应用

```
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
    while (true) {
        while (compare and swap(&bolt, 0, 1) == 1)
            /* do nothing */;
        /* critical section */;
        bolt = 0;
        /* remainder */;
    }
}
void main()
{
    bolt = 0;
    parbegin (P(1), P(2), . . . ,P(n));
}
```

(a) Compare and swap instruction

Figure 5.2 Hardware Support for Mutual Exclusion

5.2.2 Special Machine Instructions(6/12)

- 指令实现：CAS 指令：**CMPXCHG/CMPXCHGL**

```
#ifndef CAS_H_INCLUDED
#define CAS_H_INCLUDED

char CAS(int* addr, int old_val, int new_val) {
    char result;
    asm volatile (
        "lock;\n"
        "cmpxchgl %3, %1;\n"
        "sete %0\n"
        : "=q"(result) /*out*/
        : "m"(*addr), "a"(old_val), "r"(new_val) /*in*/
        : "memory"
    );
    return result;
}

#endif // CAS_H_INCLUDED
```

5.2.2 Special Machine Instructions(7/12)

- OpenEuler CAS 指令及加锁 函数

- 处理器设计特殊指令，在硬件重名为加锁和解锁提供原子性支持
- ARM v8 支持 CAS
- OpenEuler 利用 CAS，编写 cmpxchg_case 函数，并提供可打印 API

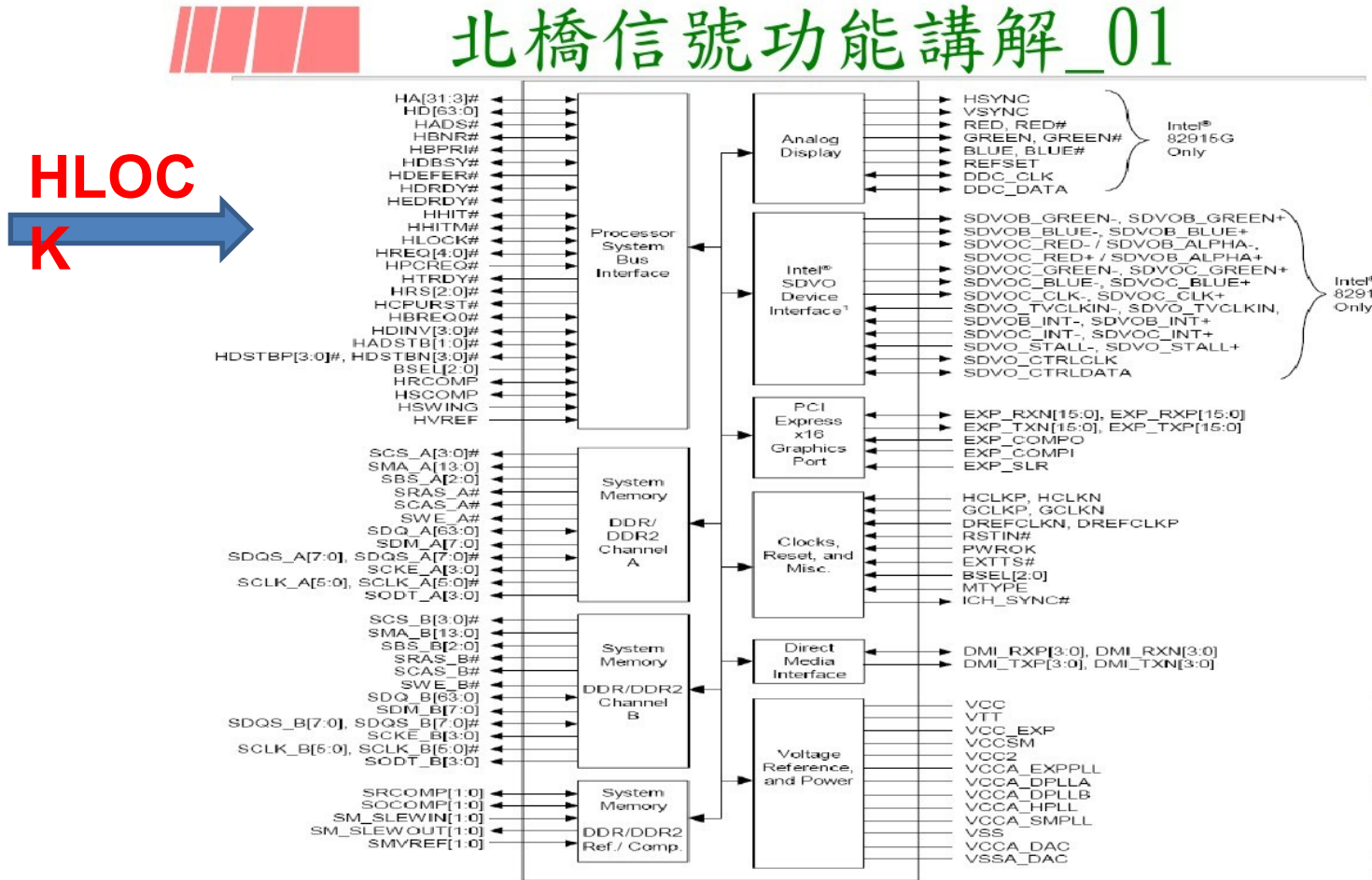
```
1. //源文件: arch/arm64/include/asm/atomic_lse.h:cmpxchg_case_# #name()  
2. asm volatile(ARM64_LSE_ATOMIC_INSN(  
3. ...  
4. /* LSE atomics */  
5.     " mov " #w "30,  %" #w "[old]\n"          \ //将锁的期望值存入寄存器中  
6.     " cas " #mb #sz "\t" #w "30,  %" #w "[new], %[v]\n" \ //检测加锁操作  
7.     " mov  %" #w "[ret],  " #w "30")          \ //返回被更改之前的锁状态  
8.     : [ret] "+r" (x0), [v] "+Q" ( * (unsigned long * )ptr)  \  
9.     : [old] "r" (x1), [new] "r" (x2)          \  
10.    ...  
11.    return x0;  
12.    }
```

图 6-13 CAS 实现加锁操作

```
1. lock():  
2.     while(atomic_cmpxchg_acquire(locked,0,1) == 1)  
3.     ;
```

图 6-14 基于 CAS 实现的加锁操作示例

5.2.2 Special Machine Instructions(8/12)



Compal Confidential



Page 2

5.2.2 Special Machine Instructions(9/12)

- Exchange Instruction

```
void exchange(int register,  
              int memory) {  
    int temp;  
    temp = memory;  
    memory = register;  
    register = temp;  
}
```

- **XCHG** instruction:
 - XCHG OPRD1, OPRD2

5.2.2 Special Machine Instructions(10/12)

```
/* program mutualexclusion */
int const n = /* number of processes**/;
int bolt;
void P(int i)
{
    int keyi = 1;
    while (true) {
        do exchange (&keyi, &bolt)
        while (keyi != 0);
        /* critical section */;
        bolt = 0;
        /* remainder */;
    }
}
void main()
{
    bolt = 0;
    parbegin (P(1), P(2), . . . , P(n));
}
```

(b) Exchange instruction

Figure 5.2 Hardware Support for Mutual Exclusion

5.2.2 Special Machine Instructions(11/12)

- Advantages
 - By sharing main memory , it is applicable to any number of processes
 - single processor
 - multiple processors
 - It is simple and therefore easy to verify
 - It can be used to support multiple critical sections

5.2.2 Special Machine Instructions(12/12)

- Disadvantages
 - Busy-waiting(忙等待) consumes processor time
 - Starvation(饥饿) is possible when a process leaves a critical section and selection policy may cause some process indefinitely 无限期的 be denied access.
 - Deadlock (死锁) is possible
 - If a low priority process has the critical region and a higher priority process needs, the higher priority process will obtain the processor to wait for the critical region

5.2 Mutual Exclusion: Hardware Support

- 5.2.1 Hardware approaches
 - 5.2.1 Interrupt Disabling
 - 5.2.2 Special Machine Instructions
- 5.2.2 Software approaches
 - Try Dekker in Appendices A.1
 - Peterson A.2

5.2.2 Software approaches(1/6)

- First attempt : `int turn =0; //turn0 P0 进入 , 1 则 P1 进入`

- `while(1){`
- `while(turn != 0) ;`
- Critical section
- `turn = 1;`
- `...`
- `}`

- `while(1){`
- `while(turn != 1) ;`
- Critical section
- `turn = 0;`
- `...`
- `}`

5.2.2 Software approaches(2/6)

- Second attempt `int flag[2]={0,0}; //flag[i] = 1; 进程 i 意图进入`

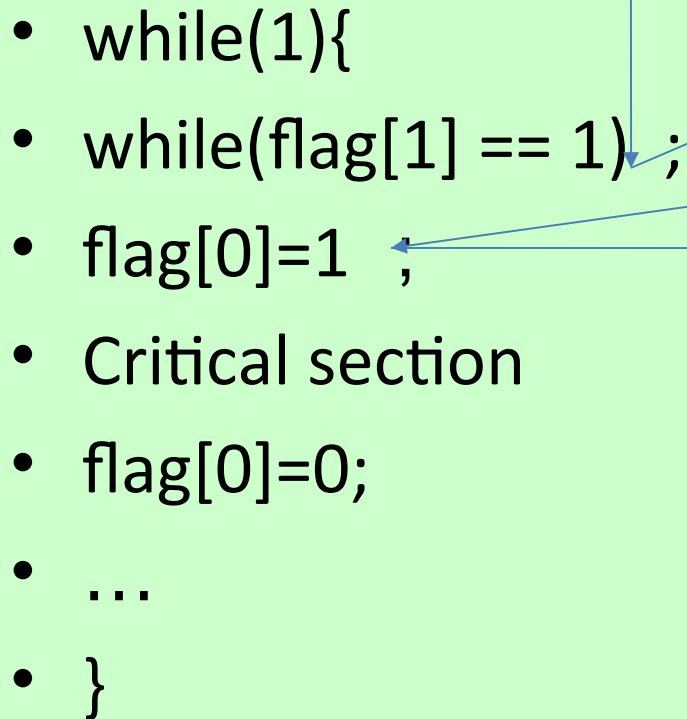


Diagram illustrating the transformation of a code snippet. A vertical blue arrow points from the first code block to the second. Two diagonal blue arrows point from the second code block back to the first, indicating a mapping between the two versions of the code.

- `while(1){`
- `while(flag[1] == 1) ;`
- `flag[0]=1 ;`
- Critical section
- `flag[0]=0;`
- ...
- `}`

- `while(1){`
- `while(flag[0]= 1) ;`
- `flag[1]=1 ;`
- Critical section
- `flag[1] = 0;`
- ...
- `}`

5.2.2 Software approaches(3/6)

- Third attempt `int flag[2]={0,0}; //flag[i] =1; 进程 i 意图进入`

- `while(1){`
- `flag[0]=1 ;`
- `while(flag[1] == 1) ;`
- Critical section
- `flag[0]=0;`
- ...
- `}`

- `while(1){`
- `flag[1]=1 ;`
- `while(flag[0] == 1) ;`
- Critical section
- `flag[1] = 0;`
- ...
- `}`

5.2.2 Software approaches(4/6)

```
/* PROCESS 0 */
-
-
flag[0] = true;
while (flag[1]) {
    flag[0] = false;
    /*delay */;
    flag[0] = true;
}
/*critical section*/;
flag[0] = false;
-

/* PROCESS 1 */
-
-
flag[1] = true;
while (flag[0]) {
    flag[1] = false;
    /*delay */;
    flag[1] = true;
}
/* critical section*/;
flag[1] = false;
```

(d) Fourth attempt



Note

Figure A.1 Mutual Exclusion Attempts

下载 `dekker4textbookab-stu.c` , 加入适当 `sleep`, 演示活锁状况

5.2.2 Software approaches(5/6)

- flag[0],flag[1] 起到的作用是表明线程希望进入临界区的意愿
- 如果两个线程都有意愿，那么将通过对 turn 的状态判断决定谁进入临界区，谁让权等待。
- main 初始化 turn=1, 则争抢时第一次执行为 p1; 争抢情况下如果 p1 第二次进入，则会因为 turn=0 而让权给 p0
- 如果没有争抢，则 p1 p0 均可多次进入

```
boolean flag [2];
int turn;
void P0()
{
    while (true) {
        flag [0] = true;
        while (flag [1]) {
            if (turn == 1) {
                flag [0] = false;
                while (turn == 1) /* do nothing */;
                flag [0] = true;
            }
        }
        /* critical section */;
        turn = 1;
        flag [0] = false;
        /* remainder */;
    }
}
void P1( )
{
    while (true) {
        flag [1] = true;
        while (flag [0]) {
            if (turn == 0) {
                flag [1] = false;
                while (turn == 0) /* do nothing */;
                flag [1] = true;
            }
        }
        /* critical section */;
        turn = 0;
        flag [1] = false;
        /* remainder */;
    }
}
void main ()
{
    flag [0] = false;
    flag [1] = false;
    turn = 1;
    parbegin (P0, P1);
}
```

Figure A.2 Dekker's Algorithm

5.2.2 Software approaches(6/6)

- 1981 Peterson
- 最终谁让权的决定权取决于谁的 turn 赋值指令最后执行
- 争抢情况下，不允许同一个线程多次执行，会交叉执行

```
boolean flag [2];
int turn;
void P0()
{
    while (true) {
        flag [0] = true;
        → turn = 1;
        while (flag [1] && turn == 1) /* do nothing */;
        /* critical section */;
        flag [0] = false;
        /* remainder */;
    }
}
void P1()
{
    while (true) {
        flag [1] = true;
        → turn = 0;
        while (flag [0] && turn == 0) /* do nothing */;
        /* critical section */;
        flag [1] = false;
        /* remainder */;
    }
}
void main()
{
    flag [0] = false;
    flag [1] = false;
    parbegin (P0, P1);
}
```

Figure A.3 Peterson's Algorithm for Two Processes

Linux support

- Linux support :
 - 互斥 mutex
 - 信号量 Semaphore
 - 管程 Monitors
 - 消息 Message Passing

Linux support

- POSIX 标准 (可移植操作系统接口 (Portable Operating System Interface)) 下互斥锁是 pthread_mutex_t
 - 1 int pthread_mutex_init(pthread_mutex_t * mutex , pthread_mutexattr_t * attr);
 - 2 int pthread_mutex_destroy (pthread_mutex_t * mutex);
 - 3 int pthread_mutex_lock (pthread_mutex_t * mutex);
 - 4 int pthread_mutex_unlock (pthread_mutex_t * mutex);
 - 5 int pthread_mutex_trylock (pthread_mutex_t * mutex);