OPERATING SYSTEMS

Cooperating Processes

Operating Systems

郑贵锋 博士
中山大学计算机学院
zhenggf@mail.sysu.edu.cn
https://gitee.com/code_sysu

中山大學 SUN YAT-SEN UNIVERSITY

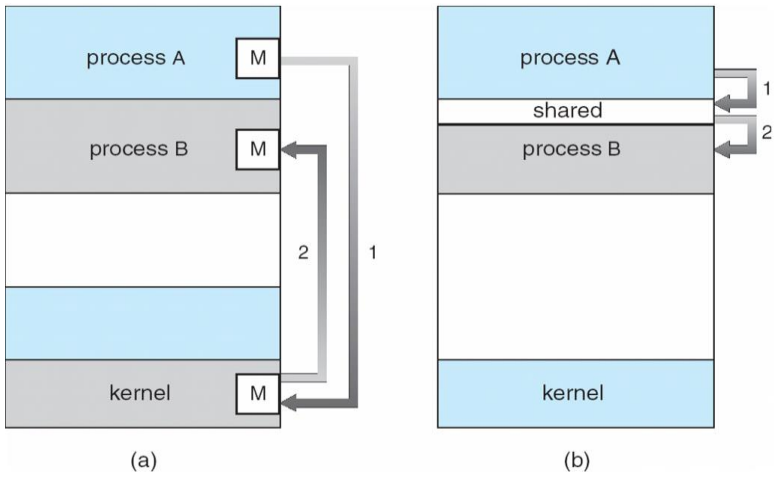
- **目录**
  - **前景**
  - **竞争条件**(Race Condition)
  - **临界区问题**(The Critical-Section Problem)

Introduction

- **协作进程**
  - 系统内的进程可以是独立的，也可以是协作的。
    - 独立进程不能影响或受另一进程的执行影响。
    - 协作进程可以影响或受其他进程的影响，包括共享数据。
  - 协作进程的原因：
    - 信息共享
    - 计算加速
    - 模块化
    - 方便

Introduction

- **协作进程**
  - 协作模式



数据一致性与并发执行

- 并发进程或线程通常需要共享数据（在共享内存或文件中维护）和其他资源。
- 如果没有受控的访问策略，对共享数据的并发访问可能会导致数据不一致。
- 并发进程执行的操作将取决于它们交错执行的顺序。
  - 维护数据一致性需要确保协作进程有序执行的机制。

数据一致性与并发执行

- 例1
  - 共享数据

    int a = 1, b = 2, c = 3;
  - 线程 T_1:

    c = a + b;

    线程 T_2:

    int d = 4;
    a = a − d
    b = b + d;
    c = a + b;
  - 似乎T_1和T_2应该有相同的结果c=3
  - 但如果T_1和T_2同时执行：
    - 假设T_1在T_2完成a=a-d之后，b=b+d之前计算a+b
    - 此时a=-3，T_1将无法获得c=3的正确结果，而得到c=-1

Introduction

- **数据一致性与并发执行**
  - 例2
    - 进程P_1和P_2正在运行相同的过程echo()，并且可以访问相同的变量inchar。进程可以在任何地方中断。
    - 共享数据

```
static char inchar;
```

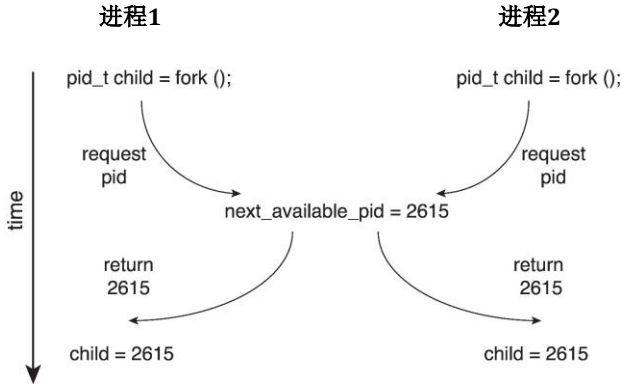
    - 进程P_1、P_2：

```
void echo() {
  cin >> inchar;
  cout << inchar;
}
```

    - 如果P_1在用户输入后先中断，P_2完全执行。然后P_1继续执行，回显的字符将是P_2输入的字符。我们失去了数据的一致性。

Race Condition

- **竞争条件**
  - 示例：分配pid时的竞争条件。



Race Condition

- **竞争条件**
  - 竞争条件是多个进程并发（同时）访问和操作共享数据的情况。共享数据的值取决于应用于数据的处理顺序。
    - 为了防止竞争条件，并发进程必须协调，换句话说，必须同步
  - 示例：更新变量时的竞争条件
    - 共享数据：

```
double balance;
```

Process$_1$:

```
… …
balance += amount;
… …
```

Code for Process$_1$:

```
… …
Load  R1, balance
Load  R2, amount
Add   R1, R2
Store R1, balance
… …
```

Process$_2$:

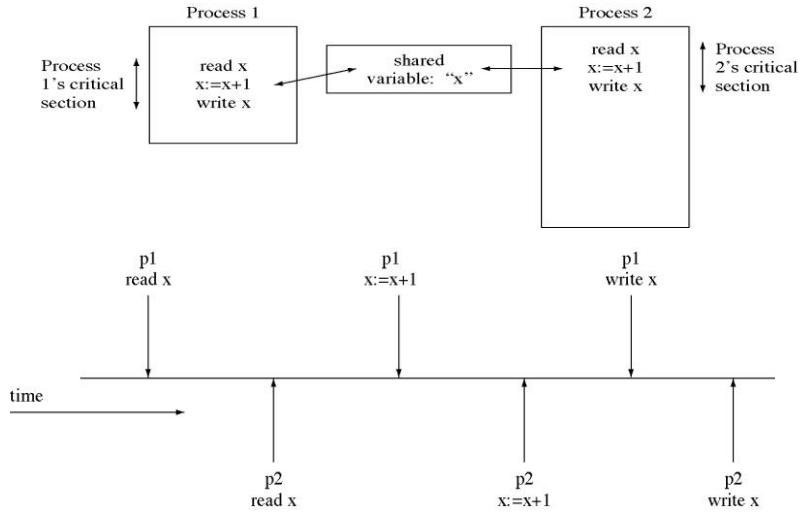
```
… …
balance += amount;
… …
```

Code for Process$_2$:

```
… …
Load  R1, balance
Load  R2, amount
Add   R1, R2
Store R1, balance
… …
```

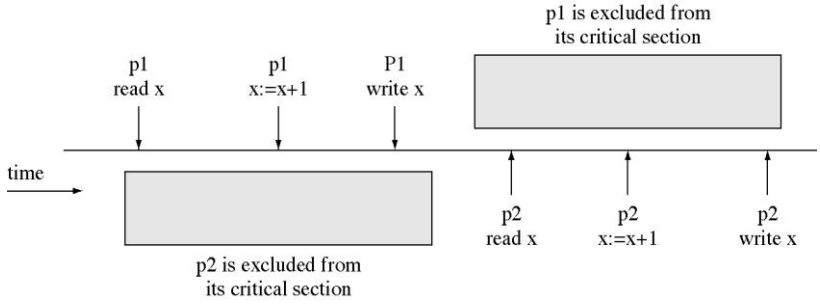
Race Condition

- **竞争条件**
  - 示例：更新变量时的竞争条件（续）



Race Condition

- **竞争条件**
  - 示例：更新变量时的竞争条件（续）
    - 设置临界区(Critical Section) 防止竞争条件



    - 多道程序允许逻辑并行（有效地使用设备），但当存在竞争条件时，我们会失去正确性。
    - 因此，我们禁止在临界区内进行逻辑并行，失去了一些并行性，但恢复了正确性。

Race Condition

- **竞争条件**
  - 回顾：共享内存的生产者-消费者问题（Lecture08）
    - 共享数据

```
#define BUFFER_SIZE 10

typedef struct {
    … …      /* item structure */
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

    - 共享缓冲区实现为带有两个逻辑指针的循环数组：in和out。
      - in指向缓冲区下一可用空间；out指向缓冲区的第一个可用项。
      - 当in == out时，缓冲区为空

      - 当((in+1) % BUFFER_SIZE) == out时，缓冲区已满
      - 此方案最多允许缓冲区中同时包含(BUFFER_SIZE-1)个项目

Race Condition

- **竞争条件**
  - 回顾：共享内存的生产者-消费者问题
    - 生产者：

```
item next_produced;
while (true) {
    …… /* 生产一个项目保存在next_produced结构体 */
    while (((in + 1) % BUFFER_SIZE) == out)
        ; /* buffer满，等待 */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

    - 消费者：

```
item next_consumed;
while (true) {
    while (in == out)
        ;     /* buffer空，等待 */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    …… /* 消费next_consumed项目 */
}
```

Race Condition

- **竞争条件**
  - P/C问题的共享计数器解决方案
    - 使用一个整数*count*来跟踪缓冲区中的项目数，可得到一个填充所有缓冲区的解决方案，而不仅是 BUFFER_SIZE-1 的可用项
    - 最初，*count*设置为0。生产者在生产一个新项目后增加1，消费者在消费一个项目后减少1。
    - 共享数据

```
#define BUFFER_SIZE 10

typedef struct {
    … …    /* item structure */
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int count = 0;
```

Race Condition

- **竞争条件**
  - P/C问题的共享计数器解决方案
    - 生产进程

```
item nextProduced;
while (TRUE) {
        … …     /* produce an item and put in nextProduced */
    while (count == BUFFER_SIZE)
        ;       /* do nothing – no free slots */
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
}
```

    - 消费进程

count为4

```
item nextConsumed;
while (TRUE) {
    while (count == 0)
        ;       /* do nothing – nothing to consume */
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;
                /* consume the item in nextConsumed */
}
```

Race Condition

- **竞争条件**
  - 原子操作
    - 原子的/不可分割的操作(原语) 指完整完成且无中断的操作。
    - 在生产者进程和消费者进程中

```
count++;
```

    - 和

```
count--;
```

    - 必须以原子方式执行。
    - 语句“count++”可以用机器语言实现为：

```
register1 = count
register1 = register1 + 1
count = register1
```

    - 语句“count--”可以实现为：

```
register2 = count
register2 = register2 – 1
count = register2
```

Race Condition

- **竞争条件**
  - 原子操作
    - 如果生产者和消费者都试图同时更新缓冲区，则汇编语言语句可能会交错。
      - 交错取决于生产者和消费者进程的调度方式。
    - 考虑一个初始为"count = 5"的交错执行：

      Producer: `register1 = count` (*register1 = 5*)
      Producer: `register1 = register1 + 1` (*register1 = 6*)
      Consumer: `register2 = count` (*register2 = 5*)
      Consumer: `register2 = register2 - 1` (*register2 = 4*)
      Producer: `count = register1` (*count = 6*)
      Consumer: `count = register2` (*count = 4*)
    - count的值可以是4或6，而正确的结果应该是5。
    - 这也是竞争条件(Race Condition)

Race Condition

- **竞争条件**
  - 通过共享进行协作
    - 协作进程使用和更新共享数据，如共享变量、内存、文件和数据库。
    - 写入必须是互斥的，以防止竞争条件导致数据视图不一致。
      - 临界区用于提供这种数据完整性。
      - 执行临界区的进程不得无限期延迟；没有死锁或饥饿。
  - 通过消息传递进行协作
    - 消息通信提供了一种同步或协调各种活动的方法。
      - 可能出现死锁Deadlock
        - 每个进程都在等待来自另一个进程的消息
      - 可能出现饥饿Starvation
        - 两个进程相互发送消息，而另一个进程等待消息

The Critical-Section Problem

- **临界区问题**
  - 假设
    - n个进程正在竞争使用一些共享数据。
    - 不对速度或CPU数量进行任何假设。
    - 每个进程都有一个访问共享数据的代码段，称为临界区（CS）
  - 当一个进程执行处理共享数据或资源的代码时，我们说该进程处于该共享数据或资源的临界区。
  - 临界区问题
    - 临界区问题是设计一个协议，竞争进程可以使用该协议来同步其活动，以便协作地共享数据。
    - 临界区的执行必须相互排斥(互斥).
      - 确保当一个进程在其临界区执行时，不允许其他进程在其临界区执行（即使有多个处理器）。
        - 也就是说，没有两个进程同时在其临界区执行。

The Critical-Section Problem

- **临界区问题**
  - 临界区问题的动态
    - 进入区Entry Section(ES)
      - 每个进程必须首先请求进入其临界区的许可。实现此请求的代码段称为进入区。
    - 退出区Leave/Exit Section(LS)
      - 临界区之后可能会有一个退出区
    - 剩余区Remainder Section(RS)
      - 之后其他代码为剩余区
    - 进程P_i的一般结构：

```
while (TRUE) {
    entry section进入区
    critical section临界区
    leave section退出区
    remainder section剩余区
};
```

The Critical-Section Problem

- **临界区问题**
  - 临界区问题的正确解决方案必须符合三个基本标准：
    - 互斥Mutual Exclusion
    - 推进Progress
    - 有限等待Bounded Waiting
  - 互斥
    - 如果进程P_i在其临界区执行，则其他进程不能在其临界区执行
    - 启示：
      - 临界区最好集中和简短
      - 最好不要在临界区陷入无限循环
      - 如果某个进程在其临界区以某种方式停止/等待，则它不得干扰其他进程。

The Critical-Section Problem

- **临界区问题**
  - 推进
    - 如果没有进程在其临界区执行，并且存在一些希望进入其临界区的进程，则不能无限期推迟选择下一个将进入临界区的进程
      - 如果只有一个进程想要进入，它应能进入。
      - 如果两个或以上的进程想进入，其中一个应能进入。
  - 有限等待
    - 在进程发出进入其临界区的请求后，且该请求被批准之前（即在请求进程的等待期间），其他并发进程进入临界区的次数必须受到限制。
      - 假设每个进程以非零速度执行
      - 没有关于所有进程的相对速度的假设
    - 注意到这并不意味着该进程是有限等待的，比如在死锁发生的情况下。

The Critical-Section Problem

- **临界区问题**
  - 抢占式和非抢占式内核
    - 许多内核模式进程可能同时在操作系统中运行。实现操作系统的内核代码可能会受到几种可能的竞争条件的影响。
      - 考虑容易出现竞争条件的内核数据结构包括维护打开文件列表的结构、维护内存分配、维护进程列表和中断处理。
    - 非抢占式内核和抢占式内核是处理操作系统中临界区的两种常用方法。
    - 非抢占内核
      - 非抢占式内核不允许抢占在内核模式下运行的进程；内核模式进程将一直运行，直到退出内核模式、阻塞或自动放弃对CPU的控制。
      - 非抢占式内核基本上不受内核数据结构上的竞争条件的影响，因为一次只有一个进程处于活动状态。

The Critical-Section Problem

- **临界区问题**
  - 抢占式和非抢占式内核
    - 抢占式内核
      - 抢占式内核允许进程在内核模式下运行时被抢占。
      - 抢占式内核必须确保共享的内核数据不受竞争条件的影响。对于SMP体系结构来说尤其困难，因为两个内核模式进程可能同时在不同的处理器上运行。
      - 抢占式内核的响应速度更快。内核模式进程在放弃处理器之前运行任意长时间的风险较小，处理器会自动分配给等待的进程。
      - 抢占式内核更适合于实时编程。它将允许实时进程抢占当前在内核中运行的进程。

The Critical-Section Problem

- **临界区问题**
  - 临界区问题的几种解决方案
    - 基于软件的解决方案
      - 算法的正确性不依赖于任何其他假设
    - 基于硬件的解决方案
      - 同步硬件
        - 依靠一些特殊的机器指令
    - 操作系统解决方案
      - 通过系统/库调用向程序员提供函数和数据结构
    - 编程语言解决方案
      - 作为语言的一部分提供的语言结构

The Critical-Section Problem

- **临界区问题的软件解决方案**
  - 我们首先考虑只有两个进程的情况。
    - 算法1、2和3不能满足三个基本标准。
    - 算法4是正确的。
      - 这是Peterson算法（*G. L. Peterson*, 1981）
  - 然后我们推广到N个进程。
    - N进程的Peterson算法
    - 兰波特的烘焙算法（*Leslie Lamport*, 1974）
    - 艾森伯格-麦奎尔算法（*Murray A. Eisenberg* & *Michael R. McGuire*, 1972）
  - 只有两个进程时的初始表示法
    - 它们被编号为P_0和P_1
    - 当呈现进程P_i(Larry, l, i)时，使用P_j(Jim, J,j)表示另一个进程
      - 在只有两个进程的情况下，j = i – 1

The Critical-Section Problem

- **临界区问题的软件解决方案**
  - 初步尝试
    - 进程P_i的一般结构（另一个是P_j）

```
do {
    entry section
    critical section
    leave section
    remainder section
} while (TRUE);
```

    - 进程可以共享一些公共数据以同步其操作。
      - 这些共享数据已初始化，无法从任何剩余区访问。

The Critical-Section Problem

- **临界区问题的软件解决方案**
  - 算法1 – Larry/Jim版本
    - 共享变量

```
string turn = "Larry"; /* 初始值无所谓 */
```

Process Larry:

```
do {
    while (turn != "Larry")
        sleep(1); /* busy waiting */
    Larry's critical section
    turn = "Jim";
      /* Jim can enter its CS */
    Larry's remainder section
} while (TRUE);
```

Process Jim:

```
do {
    while (turn != "Jim")
        sleep(1); /* busy waiting */
    Jim's critical section
    turn = "Larry";
      /* Larry can enter its CS */
    Jim's remainder section
} while (TRUE);
```

    - 互斥：只有当turn为"Larry"时，Larry才能进入他的CS。
    - 没有推进：如果Jim继续在剩余区工作，Larry无法第二次进入CS
    - 有限等待：Larry在离开CS时将turn改为"Jim"，睡着的Jim可以醒来

The Critical-Section Problem

- **临界区问题的软件解决方案**
  - **算法.1–P_i/P_j版本**
    - **共享变量**

```
int turn = 0;
  /* 初始时turn = 0. turn = i 表示进程i可以进入
     其临界区 */
```

    - **进程P_i**

```
do {
    while (turn != i)
        sleep(1);  /* 忙等待 */
    critical section of process i
    turn = j;
    remainder section of process i
} while (TRUE);
```

The Critical-Section Problem

- **临界区问题的软件解决方案**
  - **算法.2–Larry/Jim版本**
    - **共享变量**

```
boolean flag_larry = TRUE;
Boolean flag_jim = FALSE;
  /* Larry准备进入他的临界区 */
```

Process Larry:

```
do {
    while (flag_jim)
        sleep(1); /* busy waiting */
    flag_larry = TRUE;
    Larry's critical section
    flag_larry = FALSE;
      /* Jim can enter its CS */
    Larry's remainder section
} while (TRUE);
```

Process Jim:

```
do {
    while (flag_larry)
        sleep(1); /* busy waiting */
    flag_jim = TRUE;
    Jim's critical section
    flag_jim = FALSE;
      /* Larry can enter its CS */
    Jim's remainder section
} while (TRUE);
```

    - **不互斥：**flag_larry在第二次开始迭代时是FALSE。如果Jim在此时开始他的迭代，两个进程将同时进入临界区。

The Critical-Section Problem

- **临界区问题的软件解决方案**
  - **算法.2–P_i/P_j版本**
    - **共享变量**

```
boolean flag[2];
  /* 初始化 flag[0] = flag[1] = FALSE */
flag [i] = TRUE;
  /* Pi已准备好进入其临界区*/
```

    - **进程P_i**

```
do {
    while (flag[j]);
        sleep(1); /* busy waiting */
    flag[i] = TRUE;
    critical section of process i
    flag[i] = FALSE;
    remainder section of process i
} while (TRUE);
```

The Critical-Section Problem

- **临界区问题的软件解决方案**
  - **算法.3–Larry/Jim版本**
    - **共享变量**

```
boolean flag_larry = TRUE;
Boolean flag_jim = FALSE;
  /* Larry拉里准备进入它的临界区 */
```

Process Larry:

```
do {
    flag_larry = TRUE;
    while (flag_jim)
        sleep(1); /* busy waiting */
    Larry's critical section
    flag_larry = FALSE;
    Larry's remainder section
} while (TRUE);
```

Process Jim:

```
do {
    flag_jim = TRUE;
    while (flag_larry)
        sleep(1); /* busy waiting */
    Jim's critical section
    flag_jim = FALSE;
    Jim's remainder section
} while (TRUE);
```

    - **无推进：Larry和Jim在同一时间开始，都会进入循环等待**

The Critical-Section Problem

- **临界区问题的软件解决方案**
  - **算法.3–P_i/P_j版本**
    - **共享变量：**

```
boolean flag[2];
  /* 初始化 flag[0] = flag[1] = FALSE */
flag[i] = TRUE;
  /* Pi已准备好进入其临界区 */
```

    - **进程P_i**

```
do {
     flag[i] = TRUE;
     while (flag[j])
        sleep(1); /* busy waiting */
     critical section of process i
     flag [i] = FALSE;
     remainder section of process i
} while (TRUE);
```

The Critical-Section Problem

- **临界区问题的软件解决方案**
  - **算法.4–Larry/Jim版本（Peterson的解决方案）**
    - **算法1和2/3的组合共享变量：**

```
string turn = "Larry";
boolean flag_larry = TRUE;
Boolean flag_jim = FALSE;
```

Process Larry:

```
do {
    flag_larry = TRUE;
    turn = "Jim";
    while (flag_jim && turn=="Jim");
        sleep(1); /* busy waiting */
    Larry's critical section
    flag_larry = FALSE;
    Larry's remainder section
} while (TRUE);
```

Process Jim:

```
do {
    flag_jim = TRUE;
    turn = "Larry";
    while (flag_larry && turn=="Larry");
        sleep(1); /* busy waiting */
    Jim's critical section
    flag_jim = FALSE;
    Jim's remainder section
} while (TRUE);
```

    - **算法4符合所有基本标准：互斥、推进和有限等待；解决两个进程的临界区问题。**

The Critical-Section Problem

- **临界区问题的软件解决方案**
  - **算法.4–P_i/P_j版本（Peterson解决方案）**
    - **算法1和2/3的组合共享变量**

```
int turn;
turn = i;
boolean flag[2];
flag[i] = TRUE;
```

    - **进程P_i**

```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);
        sleep(1); /* busy waiting */
    critical section of process i
    flag[i] = FALSE;
    remainder section of process i
} while (TRUE);
```

The Critical-Section Problem

- **临界区问题的软件解决方案**
  - Peterson解决方案
    - **仅限于在其临界区和剩余区之间交替执行的两个进程。算法4被称为Peterson解决方案（*Gary.L.Peterson*，1981）。**
    - Peterson算法为解决临界区问题提供了良好的算法描述。为了证明Peterson的算法满足互斥、推进和有限等待三个基本标准，请参阅课本第6.3章。
      - 条件是变量turn, flag[0]和flag[1]发生变化，则会立即以原子方式传播

```
__sync_lock_test_and_set(&turn, i)
```

      - do-while甚至可以被抢占

The Critical-Section Problem

- **临界区问题的软件解决方案**
  - Peterson解决方案
    - 由于现代计算机体系结构执行基本机器语言指令（如加载和存储）的方式，无法保证Peterson的解决方案能够在此类体系结构上正确工作。
      - 为了提高系统性能，处理器和/或编译器可以对没有依赖关系的读写操作**重新排序**。
    - 对于单线程应用程序， 就程序正确性而言这种重新排序是非实质性的，最终值与预期值一致。
    - 但对于具有共享数据的多线程应用程序，指令的重新排序可能会导致不一致或意外的结果。
    - 可以使用一些**内存屏障**机制来防止这种混淆。
      - 比如Linux中的 __syn_synchronize()，或者

```
#define barrier() __asm__ __volatile__("": : :"memory")
```

The Critical-Section Problem

- **临界区问题的软件解决方案**
  - 算法5–Larry/Jim版本
    - 类似算法4，但与条目部分的前2条指令**交换**。
    - 问题：这仍然是一个正确的解决方案吗？这是互斥的吗？
    - 共享变量：

```
string turn = “Larry”;
boolean flag_larry = TRUE;
Boolean flag_jim = FALSE;
```

Process Larry:

```
do {
   turn = “Jim”;
   flag_larry = TRUE;
   while (flag_jim && turn==“Jim”);
       sleep(1); /* busy waiting */
   Larry’s critical section
   flag_larry = FALSE;
   Larry’s remainder section
} while (TRUE);
```

Process Jimy:

```
do {
   turn = “Larry”;
   flag_jim = TRUE;
   while (flag_larry && turn==“Larry”);
       sleep(1); /* busy waiting */
   Jim’s critical section
   flag_jim = FALSE;
   Jim’s remainder section
} while (TRUE);
```

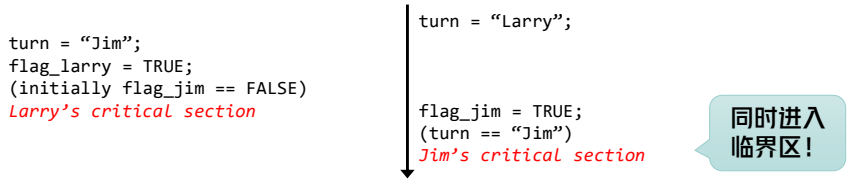
The Critical-Section Problem

- **临界区问题的软件解决方案**
  - **算法.5–Larry/Jim版本**

```
string turn = "Larry";
boolean flag_larry = TRUE;
Boolean flag_jim = FALSE;
```

Process Larry:	Process Jimy:

`do {`
`  turn = "Jim";`
`  flag_larry = TRUE;`
`  while (flag_jim && turn=="Jim");`
`  sleep(1); /* busy waiting */`
`  Larry's critical section`
`  flag_larry = FALSE;`
`  Larry's remainder section`
`} while (TRUE);`
	`do {`
	`  turn = "Larry";`
	`  flag_jim = TRUE;`
	`  while (flag_larry && turn=="Larry");`
	`  sleep(1); /* busy waiting */`
	`  Jim's critical section`
	`  flag_jim = FALSE;`
	`  Jim's remainder section`
	`} while (TRUE);`



The Critical-Section Problem

- **临界区问题的软件解决方案**
  - N进程的Peterson算法.
    - Peterson算法可以推广到两个以上的进程。下面的算法是Peterson算法对N个进程的推广。
    - 它使用N个不同的级别
      - 每一级代表临界区前的一个"等候室"。
      - 每个级别将允许至少一个进程前进，同时保持一个进程处于等待状态。

The Critical-Section Problem

- **临界区问题的软件解决方案**
  - N进程的Peterson算法.
    - 共享数据

```
int level[N]; /* 进程 0..N-1的当前级数 */
int waiting[N-1]; /* 级数 0..N-2中每一级的一个等待进程 */
memset(level,(-1),sizeof(level));
memset(waiting,(-1),sizeof(waiting));
```

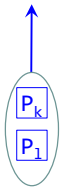
    - 进程P_i

```
for (lev = 0; lev < N-1; ++lev) { /* lev遍历 0..N-2级 */
    level[i] = lev;    /* 进程 i的当前级数 lev */
    waiting[lev] = i;  /* 级数 lev中进程 i在等待 */
    while (waiting[lev] == i && (存在k≠i, 使得level[k]>=lev))
        sleep(1); /* 当前面级别有其他进程k在等时，等待*/
}
/* Pi升级到达 N-1时，可以进入临界区*/
critical section of process i
level[i] = -1;
  /* 允许 n-2 级的另一进程退出while循环，进入它的临界区 */
remainder section of process i
```

The Critical-Section Problem

- **临界区问题的软件解决方案**
  - N进程的Peterson算法.
    - 到达N-1级 (level(i)==N-1) 的进程P_i将退出for循环并进入其临界区
    - 任何进程P_i都会将其级别lev升级为lev+1(即退出while循环)，当
      - 其他一些进程P_j将其级别升级到P_i的级别（在 level(j)==lev 和 waiting[lev]==j后）
      - 或任何其他进程的级别都低于lev

级别	0	…	q	q+1	q+2	…	N-2	N-1
等候室	P_m	…	P_j	P_i	P_t	…		



等待室中的进程不能退出while循环并升级，即使已调度，除非它是当前级别最高的唯一进程。
任何不在等候室中的进程如被调度，都将退出while循环并升级。
注：图中P_k、P_l与P_j都级别都是q，但等候室里只有P_j

The Critical-Section Problem

- **临界区问题的软件解决方案**
  - N进程的Peterson算法.
    - 到达N-1级 (level(i)==N-1) 的进程P_i将退出for循环并进入其临界区
    - 任何进程P_i都会将其级别lev升级为lev+1(即退出while循环)，当
      - 其他一些进程P_j将其级别升级到P_i的级别（在 level(j)==lev 和 waiting[lev]==j后）
      - 或任何其他进程的级别都低于lev

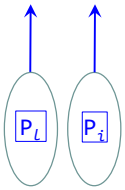
级别	0	...	q	q+1	q+2	...	N-2	N-1
等候室	P_m	...	P_j	P_i	P_t	...		

因此，任何级别低于当前最高级别的等候室都必须被占用。

The Critical-Section Problem

- **临界区问题的软件解决方案**
  - N进程的Peterson算法.
    - 到达N-1级 (level(i)==N-1) 的进程P_i将退出for循环并进入其临界区
    - 任何进程P_i都会将其级别lev升级为lev+1(即退出while循环)，当
      - 其他一些进程P_j将其级别升级到P_i的级别（在 level(j)==lev 和 waiting[lev]==j后）
      - 或任何其他进程的级别都低于lev

级别	0	...	q	q+1	q+2	...	N-2	N-1
等候室	P_m	...	P_j	P_k	P_t	...		

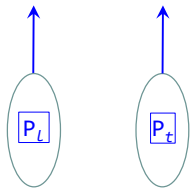


当q级的P_k被调度时，它退出while循环，升级到q+1级，占用q+1级等候室。
原 q+1等候室的P_i被移出。

临界区问题的软件解决方案

- N进程的Peterson算法.
  - 到达N-1级 (level(i)==N-1) 的进程P_i将退出for循环并进入其临界区
  - 任何进程P_i都会将其级别lev升级为lev+1(即退出while循环)，当
    - 其他一些进程P_j将其级别升级到P_i的级别（在 `level(j)==lev` 和 `waiting[lev]==j`后）
    - 或任何其他进程的级别都低于lev

级别	0	…	q	q+1	q+2	…	N-2	N-1
等候室	P_m	…	P_j	P_k	P_i	…		



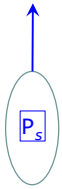
当q+1级的P_i被调度时，它退出while循环，升级到q+2级，占用q+2级等候室。
原 q+2等候室的P_t被移出。

临界区问题的软件解决方案

- N进程的Peterson算法.
  - 到达N-1级 (level(i)==N-1) 的进程P_i将退出for循环并进入其临界区
  - 任何进程P_i都会将其级别lev升级为lev+1(即退出while循环)，当
    - 其他一些进程P_j将其级别升级到P_i的级别（在 `level(j)==lev` 和 `waiting[lev]==j`后）
    - 或任何其他进程的级别都低于lev

级别	0	…	q	q+1	q+2	…	N-2	N-1
等候室	P_m	…	P_j	P_k	P_i	…	P_t	

一种极端情况：0至N-2级的房间都被占用，且P_t和P_s在同一层N-2中。P_t无法退出while循环，即使已调度，因为存在P_s与它具有相同的级别。

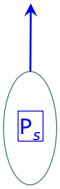


The Critical-Section Problem

- **临界区问题的软件解决方案**
  - N进程的Peterson算法.
    - 到达N-1级 (level(i)==N-1) 的进程P_i将退出for循环并进入其临界区
    - 任何进程P_i都会将其级别lev升级为lev+1(即退出while循环)，当
      - 其他一些进程P_j将其级别升级到P_i的级别（在 level(j)==lev 和 waiting[lev]==j后）
      - 或任何其他进程的级别都低于lev

级别	0	…	q	q+1	q+2	…	N-2	N-1
等候室	P_m	…	P_j	P_k	P_i	…	P_t	

当P_s被调度时，它退出while循环，因为它不在等候室中，并立即结束for循环，进入其临界区。



The Critical-Section Problem

- **临界区问题的软件解决方案**
  - N进程的Peterson算法.
    - 到达N-1级 (level(i)==N-1) 的进程P_i将退出for循环并进入其临界区
    - 任何进程P_i都会将其级别lev升级为lev+1(即退出while循环)，当
      - 其他一些进程P_j将其级别升级到P_i的级别（在 level(j)==lev 和 waiting[lev]==j后）
      - 或任何其他进程的级别都低于lev

级别	0	…	q	q+1	q+2	…	N-2	N-1
等候室	P_m	…	P_j	P_i	P_t	…		

进入临界区的进程将按照其等候室编号N-2、N-3、…、2、1和0的顺序排列。

想象一下刚开始时的情形：一批进程等着被调度……

The Critical-Section Problem

- **临界区问题的软件解决方案**
  - N进程的Peterson算法.
    - 不难证明Peterson算法满足互斥、进步和有限等待三个基本标准。
    - 编程练习
      - 编写一个包含N个竞争线程的程序，使用Peterson算法解决临界区问题。

The Critical-Section Problem

- **临界区问题的软件解决方案**
  - 算法.15-1-peterson-counter.c (1)

```
static int counter = 0;  /* number of process(s) in the critical section */
int level[MAX_N];  /* level of processes 0 .. MAX_N-1 */
int waiting[MAX_N-1];  /* waiting process of each level number 0 .. MAX_N-2 */
int max_num = 20; /* default max thread number */

static void *ftn(void *arg)
{
    int *numptr = (int *)arg;
    int thread_num = *numptr;
    int lev, k, j;

    printf("thread-%3d, ptid = %lu working\n", thread_num, pthread_self( ));
    for (lev = 0; lev < max_num-1; ++lev) { /* at least max_num-1 waiting rooms */
        level[thread_num] = lev;
        waiting[lev] = thread_num;
        while (waiting[lev] == thread_num) { /* busy waiting */
            for (k = 0; k < max_num; k++) {
                if(level[k] >= lev && k != thread_num)
                    break;
                if(waiting[lev] != thread_num)  /* check again */
                    break;
            }
            if(k == max_num) { /* lev greater than any other processes */
                break;
            }
        }
    }
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <signal.h>
#define MAX_N 1024
```

The Critical-Section Problem

- **临界区问题的软件解决方案**
  - **算法**.15-1-peterson-counter.c (2)

```
        /* critical section of process thread_num */
    printf("thread-%3d, ptid = %lu entering the critical section\n", thread_num,
pthread_self( ));
    counter++;
    if (counter > 1) {
        printf("ERROR! more than one processes in their critical sections\n");
        kill(getpid(), SIGKILL);
    }
    counter--;
      /* end of critical section */

    level[thread_num] = -1;
      /* allow other process of level max_num-2 to exit the while loop
         and enter his critical section */
    pthread_exit(0);
}

int main(int argc, char *argv[])
{
    printf("Usage: ./a.out total_thread_num\n");
    if(argc > 1) {
        max_num = atoi(argv[1]);
    }
    if (max_num < 0 || max_num > MAX_N) {
        printf("invalid max_num\n");
        exit(1);
    }
```

The Critical-Section Problem

- **临界区问题的软件解决方案**
  - **算法**.15-1-peterson-counter.c (3)

```
    memset(level, (-1), sizeof(level));
    memset(waiting, (-1), sizeof(waiting));

    int i, ret;
    int thread_num[max_num];
    pthread_t ptid[max_num];

    for (i = 0; i < max_num; i++) {
        thread_num[i] = i;
    }
    printf("total thread number = %d\n", max_num);
    printf("main(): pid = %d, ptid = %lu.\n", getpid( ), pthread_self( ));

    for (i = 0; i < max_num; i++) {
        ret = pthread_create(&ptid[i], NULL, &ftn, (void *)&thread_num[i]);
        if(ret != 0)
            fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
    }

    for (i = 0; i < max_num; i++) {
        ret = pthread_join(ptid[i], NULL);
        if(ret != 0)
           perror("pthread_join()");
    }

    return 0;
}
```

```
isscgy@ubuntu:/mnt/os-2020$ gcc alg.15-1-peterson-counter.c -pthread
isscgy@ubuntu:/mnt/os-2020$ ./a.out 10
Usage: ./a.out total_thread_num
total thread number = 10
main(): pid = 113819, ptid = 140496837703488.
thread-  0, ptid = 140496829191936 working
thread-  0, ptid = 140496829191936 entering the critical section
thread-  4, ptid = 140496795621120 working
thread-  3, ptid = 140496804013824 working
thread-  5, ptid = 140496787228416 working
thread-  2, ptid = 140496812406528 working
thread-  8, ptid = 140496627832576 working
thread-  9, ptid = 140496762050304 working
thread-  1, ptid = 140496820799232 working
thread-  6, ptid = 140496778835712 working
thread-  4, ptid = 140496795621120 entering the critical section
thread-  3, ptid = 140496804013824 entering the critical section
thread-  9, ptid = 140496762050304 entering the critical section
thread-  1, ptid = 140496820799232 entering the critical section
thread-  2, ptid = 140496812406528 entering the critical section
thread-  8, ptid = 140496627832576 entering the critical section
thread-  5, ptid = 140496787228416 entering the critical section
thread-  6, ptid = 140496778835712 entering the critical section
thread-  7, ptid = 140496770443008 working
thread-  7, ptid = 140496770443008 entering the critical section
isscgy@ubuntu:/mnt/os-2020$
```

The Critical-Section Problem

- **临界区问题的软件解决方案**
  - Lamport烘焙算法
    - 由*Leslie Lamport*于1974年提出
      - LaTex的发明者，Paxos算法，2013年图灵奖得主。
    - n进程的临界区：
      - 在进入其临界区之前，每个进程都会收到一个编号（或一张票，就像在面包房中一样）。持有最小编号的进程进入临界区。
      - 编号方案始终按枚举的递增顺序生成数字，无上界。如：

        1, 2, 3, 3, 3, 3, 4, 5, ...

      - 假设进程P_i和P_j（假定PID唯一）接收到相同的数字。
        - 如果$i<j$，则P_i将较P_j优先进入临界区。

The Critical-Section Problem

- **临界区问题的软件解决方案**
  - Lamport烘焙算法
    - 从 $\max(a_0, \ldots, a_{n-1}) + 1$ 中选择一个数字：
      - 函数 $\max(a_0, \ldots, a_{n-1})$ 返回一个数字k，使得
        $k \geq a_i,\ 0 \leq i \leq n-1$
    - 按字典顺序排序 (ticket #, PID #)
      先按ticket#，再按PID#排序
    - 共享数据：

```
boolean choosing[n];
int number[n];
```

    - 数据结构分别初始化为FALSE和0
    - choosing[i] == TRUE 意味着进程P_i正在获取其编号
    - number[i] == 0 表示进程P_i从等待列表中删除
    - 如果进程P_i执行失败，则 number[i] 必须为0

The Critical-Section Problem

- **临界区问题的软件解决方案**
  - Lamport烘焙算法
    - 进程P_i

```
do {
    choosing[i] = TRUE;
    number[i] = max(number[0], …, number[n – 1]) + 1;
    choosing[i] = FALSE;
    for (j = 0; j < n; j++) {
        /* 确保 n个进程中,每个进程都有一个编号 */
        while (choosing[j]);/*等待进程j收到其编号*/
        while ((number[j] != 0) &&
            ((number[j], j) < (number[i], i)));
          /* 等待所有有较小的编号或编号相同但优先级更高的
             进程离开他们的临界区: */
    }
    critical section of process i
    number[i] = 0;
    remainder section of process i
} while (TRUE);
```

The Critical-Section Problem

- **临界区问题的软件解决方案**
  - Lamport烘焙算法——另一种描述

```
Lock(int i) {
    choosing[i] = TRUE;
    number[i] = max(number[0], …, number[n – 1]) + 1;
    choosing[i] = FALSE;
    for (j = 0; j < n; j++) {
        while (choosing[j]);
        while ((number[j] != 0) &&
               ((number[j], j) < (number[i], i)));
}

Unlock(int i) {
    number[i] = 0;
}
```

```
Process (int i) {
    while (TRUE) {
        Lock(i);
        critical section of process i
        Unlock(i);
        remainder section of process i
    }
}
```

The Critical-Section Problem

- **临界区问题的软件解决方案**
  - Lamport烘焙算法
    - 烘焙算法满足所有三个基本标准
      - 互斥
      - 进步
      - 有限等待
    - 它解决了具有共享内存的更多进程的临界区问题，不需要更多的支持，如原子指令 set-and-test 或信号量。
      - choosing[i] 和 number[i] 仅由P_i修改。
    - 算法无死锁，无饥饿
      - 必须有一个进程，在等待列表中编号最小，以获得进入其临界区的权限。
      - 任何FIFO和无死锁算法都必须是无饥饿的。

The Critical-Section Problem

- **临界区问题的软件解决方案**
  - Eisenberg-McGuire**算法**
    - **该算法**(Murray A. Eisenberg, Michael R. McGuire. 1972)**是解决**N**进程临界区问题的解决方案。也是哲学家就餐问题的一般形式**
  - **共享数据：**

```
enum pstates {IDLE, WAITING, ACTIVE};
pstates flags[n];
int turn;
```

  - **最初，变量**turn**被设置为**0**到**n-1**之间的任意数字，表示被选择进入其临界区的进程。**
  - **每个进程的**flags**初始化为**IDLE**，并当它想进入临界区时就设置为** WAITING
  - flags[i]**的值**
    - WAITING：**进程** P_i **正在等待资源**
    - ACTIVE：P_i**暂时声明资源（尚未分配）**
    - IDEL：**其他情况**

The Critical-Section Problem

- **临界区问题的软件解决方案**
  - Eisenberg-McGuire**算法**
    - **初始化：**

```
int index; /* index is local, not shared! */
  ...
turn = 0;
  ...
for (index = 0; index < n; index++) {
    flags[index] = IDLE;
}
```

The Critical-Section Problem

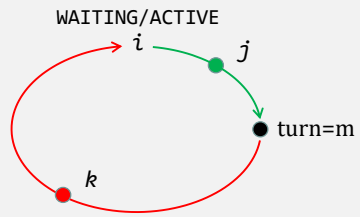
- **临界区问题的软件解决方案**
  - Eisenberg-McGuire算法
    - **进入协议（用于进程P_i）：**

```
repeat {
    /* 进程i宣布需要资源 */
    flags[i] = WAITING;

    /* 不断重复从turn到i扫描进程，直至
       所有扫描的进程的flag均为IDLE为止 */
    index = turn;
    while (index != i) {
        /* 顺时针从turn到i的所有进程的flag均为IDLE时退出循环 */
        if (flags[index] != IDLE)
            index = turn;
        else
            index = (index + 1) mod n;
    }

    /* 现在暂时声明资源*/
    flags[i] = ACTIVE;
```

The Critical-Section Problem

- **临界区问题的软件解决方案**
  - Eisenberg-McGuire算法
    - **进入协议（用于进程P_i）：**

```
repeat {
    /* 进程i宣布需要资源 */
    flags[i] = WAITING;

    /* 不断重复从turn到i扫描进程
       所有扫描的进程的flag均为I
    index = turn;
    while (index != i) {
        /* 顺时针从turn到i的所有进程的flag均为IDLE时退出循环 */
        if (flags[index] != IDLE)
            index = turn;
        else
            index = (index + 1) mod n;
    }
    /* 根据协议，在退出协议中flags[i]变为IDLE之前，从i到turn处的其他
       flags[j]（尽管它顺时针向前移动）不能从WAITING变为ACTIVE */
    /* 现在暂时声明资源*/
    flags[i] = ACTIVE;
```



flags[j]无法更改为ACTIVE，因为从m到j的顺时针扫描将经过i，而flags[i]!=IDLE.
对于从turn到i的flags[k]，则不能保证不会变为WAITING/ACTIVE。

The Critical-Section Problem

- **临界区问题的软件解决方案**
  - Eisenberg-McGuire**算法**
    - **进入协议（用于进程P_i）：**

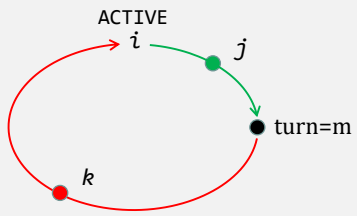
```
/* 查找Pi之外的第一个活动进程（如果有） */
index = 0; /* scan from 0 to n-1 */
while ((index < n) &&
       ((index == i) || (flags[index] != ACTIVE))) {
    index = index + 1;
}
```

```
/* 如果除Pi之外没有其他活动进程，并且如果Pi拥有turn，或者拥有turn的进程IDLE，则退出重复循环并继续。否则，重复整个过程 */
} until ((index >= n) &&
         ((turn == i) || (flags[turn] == IDLE)));

/* 声明轮到进程i执行 */
/* 资源分配给Pi */
turn = i;
```

The Critical-Section Problem

- **临界区问题的软件解决方案**
  - Eisenberg-McGuire**算法**
    - **进入协议（用于进程P_i）：**

```
/* 查找Pi之外的第一个活动
index = 0; /* scan from
while ((index < n) &&
       ((index == i) ||
    index = index + 1;
}
```



假设flags[k]处于ACTIVE状态的最小进程是k.
当进程m返回时，turn向前移动并到达k.

```
/* 当前从turn到i的进程的flags可能已经改变为ACTIVE。在这种情况下，
当进程m返回资源时，turn向前移动，并到达在i之前的第一个非IDLE进程。
再次使用flag[i]= waiting重复整个过程 */
} until ((index >= n) &&
         ((turn == i) || (flags[turn] == IDLE)));

/* 声明轮到进程i执行 */
/* 资源分配给Pi */
turn = i;
```

The Critical-Section Problem

- **临界区问题的软件解决方案**
  - Eisenberg-McGuire算法
    - 退出协议（用于进程P_i）：

```
  /* turn == i */
  /* 查找非IDLE的进程，如果没有其他进程非IDLE，将找到Pi */
index = (turn + 1) mod n;
while (flags[index] == IDLE) {
    index = (index + 1) mod n;
}

  /* 把turn给需要它的进程，否则保留它（给回自己） */
turn = index;

  /* 进程i结束了 */
flags[i] = IDLE;
```

  - *Eisenberg-McGuire算法仍然存在忙等待问题。*

The Critical-Section Problem

- **临界区问题的软件解决方案**
  - 进程执行失败怎么办？
    - 如果所有三个标准
      - 互斥
      - 进步
      - 有限等待
    - 都满足，则有效的解决方案将提供对进程在剩余区中失败的鲁棒性。
      - 因为剩余区的失败就像有一个无限长的剩余区。
    - 然而，任何有效的解决方案都无法提供针对进程在临界区中失败的健壮性。
      - 在其临界区失败的进程P_i不会向其他进程发出故障信号
        - 对于其他进程来说，P_i仍然处于其临界区中。

The Critical-Section Problem

- **临界区问题的软件解决方案**
  - 临界区问题软件解决方案的缺陷
    - 软件解决方案非常脆弱。
    - 请求进入其临界区的进程不得不处于忙等待。
      - 不必要地消耗处理器时间。
      - 如果临界区很长，那么阻塞正在等待的进程将更有效。