OPERATING SYSTEMS

Synchronization & Semaphores

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

Process Synchronization

- 目录
  - 同步硬件
  - 互斥锁
  - 信号量
    - 概念
    - 实现
  - 经典问题
    - 有界缓冲区
    - 读者和写者
    - 哲学家就餐
  - 管程
  - 死锁
  - Linux中的同步

Synchronization Hardware

- **同步硬件**
  - 概述
    - 许多系统为实现临界区代码提供硬件支持。
    - 本节讨论的所有解决方案都基于锁的思想
      - 通过锁保护临界区。
    - 单处理器可以禁用中断来保护临界区
      - 当前运行的代码将在没有抢占的情况下执行
      - 通常在多处理器系统上效率太低
    - 现代机器提供特殊的原子硬件指令：
      - *不可中断*
      - 他们要么测试一个内存字并立即设置值，要么交换两个内存字的内容。
      - 通过描述两条指令来抽象这些类型的指令背后的主要概念
        - test_and_set()
        - compare_and_swap()

Synchronization Hardware

- **同步硬件**
  - 中断禁用
    - 进程P_i

```
repeat
    disable interrupts
    critical section
    enable interrupts
    remainder section
forever
```

    - 在单处理器上
      - 互斥被保护，但执行效率降低：在临界区，我们不能将其与其他进程的剩余区代码交错执行。
    - 在多处理器上
      - 互斥不被保护
        - 临界区现在是原子的，但不是互斥的（在其他处理器上中断没有禁用）

Synchronization Hardware

- **同步硬件**
  - 特殊机器指令
    - 通常，对内存位置的访问排除了对同一位置的其他访问。
    - 延伸
      - 设计者已经提出了两个在同一个内存位置上原子（不可分割）地执行动作的机器指令。
        - 例如，读和写
      - 即使在多处理器上，这种指令的执行也是互斥的。

Synchronization Hardware

- **同步硬件**
  - 用锁解决临界区问题
    - 锁解决方案的总体布局为：

```
do {
    acquire lock
    critical section
    release lock
    remainder section
} while (TRUE);
```

Synchronization Hardware

- 同步硬件
  - 测试和设置同步硬件指令Test_and_Set
    - 以原子方式测试和设置（修改）内存字的内容（布尔版本）

```
boolean TestAndSet(boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
```

      - 它是原子执行的
      - 返回传递参数的原始布尔值
      - 将传递参数的新值设置为“TRUE”
    - 该布尔函数表示了相应机器指令的本质

Synchronization Hardware

- 同步硬件
  - 测试和设置同步硬件指令 Test_and_Set
    - 带test_and_set指令的互斥
      - 共享数据

```
boolean lock = FALSE;
```

      - 进程P_i

```
do {
    while (TestAndSet(&lock))
        ;  /* FALSE → TRUE：跳出while循环
              TRUE  → TRUE：忙等待    */
    critical section
    lock = FALSE;
    remainder section
} while (TRUE);
```

Synchronization Hardware

- **同步硬件**
  - 测试和设置同步硬件指令 Test_and_Set
    - 另一个 test_and_set 的例子
      - test_and_set 的机器指令说明：

```
bool testset(int &i)
{
    if (i == 0) {
        i = 1;
        return TRUE;    /* 成功设置返回TRUE */
    } else {
        return FALSE;   /* 否则返回FALSE */
    }
}
```

Synchronization Hardware

- **同步硬件**
  - 测试和设置同步硬件指令 Test_and_Set
    - 另一个 test_and_set 的例子
      - 使用 test_and_set 进行互斥的算法：
        - 共享变量b初始化为0；
        - 只有第一个成功设定b的P_i进入临界区。
      - 共享数据

```
int b = 0;
```

      - 进程P_i

```
repeat
    repeat {
        ; /* 忙等待，直到成功设置时跳出循环 */
    } until testset(b);
    critical section
    b = 0;
    remainder section
forever
```

Synchronization Hardware

- **同步硬件**
  - 测试和设置同步硬件指令 Test_and_Set
    - 汇编级 test_and_set 指令

```
enter_region:
TSL REGISTER, LOCK   |将锁复制到寄存器并将锁设置为1
CMP REGISTER, #0     | 锁原值是零吗？
JNE enter_region     | 若非零则锁已被别人设置，继续循环
RET                  | 返回调用方，进入临界区

leave_region:
MOVE LOCK, #0        | 在锁中存储零
RET                  | 返回调用方
```

    - TSL：原子的测试和设置锁的指令
    - 存储总线在执行期间将被锁定，以提供互斥的保证

Synchronization Hardware

- **同步硬件**
  - 交换同步硬件指令Swap
    - 原子swap（交换）两个变量的值：

```
void Swap(boolean *a, boolean *b)
{
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```

    - 本程序表示了相应机器指令的本质。

Synchronization Hardware

- 同步硬件
  - 交换同步硬件指令Swap
    - 带swap指令的互斥
      - 共享数据

```
boolean lock = FALSE;
```

      - 进程P_i

```
do {
      /* 每进程都有一个本地布尔变量 key */
    boolean key = TRUE;
    while (key == TRUE)     /* 忙等待   */
        swap(&lock, &key); /* 交换锁   */
    critical section
    lock = FALSE;
    remainder section
} while (TRUE);
```

Synchronization Hardware

- 同步硬件
  - 交换同步硬件指令Swap
    - 汇编级swap指令

```
enter_region:
MOVE REGISTER, #1          | 将1复制到寄存器
XCHG REGISTER,LOCK         | 交换寄存器和锁变量的内容
CMP REGISTER, #0           | 锁原值是零吗？
JNE enter_region           | 若非零则锁已被设置，循环
RET                        | 返回调用方，进入临界区

leave_region:
MOVE LOCK, #0              | 在锁中存储零
RET                        | 返回调用方
```

    - XCHG：原子的交换指令
    - 存储总线在执行期间将被锁定，以提供互斥的保证。

Synchronization Hardware

- **同步硬件**
  - 比较交换同步硬件指令 Compare_and_Swap
    - compare_and_swap指令对三个操作数进行操作；

```
int compare_and_swap(int *value, int expected, int new_value)
{
    int temp = *value;

    if (*value == expected)
        *value = new_value;
    return temp;
}
```

    - 原子执行
    - 仅当条件“value==expected”为真时，将变量value设置为传递的new_value的值
      - 也就是说，交换仅在这种情况下发生
    - 返回传递的参数value的原始值

Synchronization Hardware

- **同步硬件**
  - 比较交换同步硬件指令 Compare_and_Swap
    - 带 compare_and_swap 指令的互斥
      - 共享数据

```
integer lock = 0;
```

      - 解决方案：

```
do {
    while (compare_and_swap(&lock, 0, 1))
        ;  /* 返回的原值非0则忙等待 */
    critical section
    lock = 0;
    remainder section
} while (true);
```

Synchronization Hardware

- 同步硬件
  - 特殊机器指令的缺点
    - 采用忙等待，因此当进程等待访问临界区时，它将继续消耗处理器时间。
    - 当一个进程离开临界区且有多于一个进程在等待时，可能不能推进（饥饿）。
    - 它们可以用于提供互斥，但需要通过其他机制进行补充，以满足临界区问题的有限等待要求。
    - 参阅下一张幻灯片，了解使用test_and_set指令的满足所有临界区要求的算法。

Synchronization Hardware

- 同步硬件
  - 示例：带test_and_set的有限等待互斥
    - 共享数据初始化为false

```
Boolean waiting[n];
integer lock;
```

    - 进程P_i

```
do {
    waiting[i] = TRUE;
    key = TRUE;
    while (waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;
    critical section
    j = (i + 1) % n;
    while ((j != i) && !waiting[j]) /* 依序查找下一个 */
        j = (j + 1) % n;            /* 等待中的进程j */
    if (j == i)     /* 没有找到 */
        lock = FALSE;
    else            /* 找到j */
        waiting[j] = FALSE;
    remainder section
} while (TRUE);
```

Synchronization Hardware

- **同步硬件**
  - 示例：带test_and_set的有限等待互斥
    - 互斥
      - 只有当 waiting[i] == FALSE 或 key == FALSE 时，进程 P_i才能进入其临界区。
      - 只有在执行key = TestAndSet(&lock)时，key的值才能变为FALSE. 成功执行的第一个进程将发现key==FALSE，跳出循环；所有其他进程都必须等待。
      - 只有当另一个进程离开其临界区时，变量waiting[i]才会变为FALSE；只有一个waiting[i]被设置为FALSE，以保持互斥要求。
    - 进步
      - 与互斥类似，因为退出临界区的进程要么将lock设置为FALSE，要么将waiting[j]设置为FALSE。两者都允许等待进入其临界区的进程继续进行。

Synchronization Hardware

- **同步硬件**
  - 示例：带test_and_set的有限等待互斥
    - 有限等待
      - 当一个进程离开其临界区时，它按循环顺序 (i + 1, i + 2, ..., n - 1, 0, ..., i - 1) 扫描waiting数组。它将此顺序中位于进入区的第一个进程（ waiting[j] == TRUE ）指定为进入临界区的下一个进程。因此，任何等待进入其临界区的进程都将在n-1轮内完成。

Mutex Lock

- **互斥锁**
  - 基于硬件的临界区问题解决方案非常复杂，程序员一般无法访问。
  - 操作系统设计者构建软件工具来解决临界区问题。
    - 互斥锁(*Mutex Lock*)是最简单的一个。
    - 它通过先 acquire() 锁，然后 release() 锁来保护临界区。
    - 布尔变量用于指示锁是否可用。
  - 对acquire()和release()的调用必须是原子的。
    - 通常通过硬件原子指令实现。

```
acquire() {
    while (!available)
        ; /* busy waiting */
    available = FALSE;
}

release() {
    available = TRUE;
}
```

Mutex Lock

- **互斥锁**
  - 进程P_i:

```
do {
    acquire lock
    critical section
    release lock
    remainder section
 } while (true);
```

  - 互斥锁解决方案需要忙等待
    - 当一个进程处于其临界区时，试图进入其临界区的任何其他进程都必须在acquire()调用中不断循环。
    - 因此，该锁称为自旋锁(spinlock).

Semaphores

- **信号量概念**
  - 信号量(Semaphore) 是一种更健壮的同步工具，其行为类似于互斥锁，但也可以为进程提供更复杂的方式来同步其活动。
  - 从逻辑上讲，信号量S是一个有符号整数变量，除初始化外，只能通过两个原子且互斥的操作进行更改：
    - wait(S)；
      - 也表示为P(S)或down(S)
    - signal(S)；
      - 也表示为V(S)或up(S)
  - 信号量是由荷兰计算机科学家Edsger Dijkstra引入的，因此wait()操作最初被称为P（荷兰语*proberen*，“to test”）；signal()最初被称为V（*verhogen*，“to increment”）。

Semaphores

- **信号量概念**
  - n进程的临界区
    - 共享数据

```
semaphore mutex; /*初始化为 1 */
```

    - 进程P_i

```
do {
    wait(mutex);
    critical section
    signal(mutex);
    remainder section
} while (TRUE);
```

    - 通过两个原子操作进行访问，定义如下：

```
wait(S) {
    while (S <= 0)
        ; /* busy waiting; do nothing */
    S--;
}

signal(S) {
    S++;
}
```

Semaphores

- **信号量概念**
  - 须保证没有两个进程可同时在同一信号量上执行wait()和signal()
  - 信号量的实现成为临界区问题：wait和signal的代码需设临界区
    - 在临界区实现中可能有忙等待：
      - 但是实现代码很短
      - 如果临界区很少有人占用，则等待时间很短
    - 注意，应用程序可能会在临界区花费大量时间，因此这不是一个好的解决方案。
  - 当进程必须等待时，为了避免忙等待，它将被放入等待相同事件的进程阻塞队列中。
    - 每个信号量都有一个相关的等待队列。
    - 等待队列中的每个条目都有两个数据项：
      - value（整型）
      - 指向列表中下一条记录的指针

Semaphores

- **信号量的实现**
  - 将信号量定义为C结构：

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

  - 假设有两种操作：
    - block()：将调用的进程放在相应的等待队列上（挂起进程）
    - wakeup()：删除等待队列中的一个进程，并将其放入就绪队列（恢复被阻塞进程的执行）
  - 这两个操作由操作系统作为基本系统调用提供。

Semaphores

- **信号量的实现**
  - 信号量操作现在定义为

```
wait (semaphore *S) {
    S->value--;
    if (S->value < 0) {
        block(); /* no busy waiting */
        add this process to S->list;
    }
}

signal (semaphore *S) {
    S->value++;
    if (S->value <= 0) { /* S.list not empty */
        remove a process P from S->list;
        wakeup(P);
    }
}
```

  - 在此实现中，信号量值可能为负值，而其大小（等待列表的长度）是等待该信号量的进程数。

Semaphores

- **信号量的实现**
  - 以原子方式执行信号量操作至关重要。
    - 回想一下，我们必须保证没有两个进程可以在同一信号量上同时执行wait()和signal()操作。这是一个临界区问题，可通过以下方式解决：
      - 在单处理器环境中抑制中断
      - 在多处理器环境中使用 compare_and_swap() 或自旋锁
  - 两种类型的信号量
    - 二值信号量
      - 其整数值value只能取0或1（实际上是布尔值）；
      - 可以更简单地实现（使用waitb()和signalb()操作）；
      - 与互斥锁相同
    - 计数信号量
      - 它的整数值value可以在不受限制的范围上变化。

Semaphores

- **信号量的实现**
  - 二值信号量的实现
    - 实现二值信号量S是很容易的。
    - waitb(S):

```
if (S.value == 1) { /* 锁空闲 */
    S.value = 0;                /* 占用之 */
} else {            /* 否则锁被占用 */
    block();                    /* 阻塞自己 */
    add this process to S.list  /* 加入队列 */
}
```

    - signalb(S):

```
if (S.list is empty) {  /* 无人等待 */
    S.value = 1;                /* 释放锁 */
} else {                /* 有人等待 */
    remove a process P from S.list  /* 把一个进程出队,      */
    wakeup(P);                      /* 唤醒它, 进入其临界区 */
}
```

Semaphores

- **信号量的实现**
  - 计数信号量的实现
    - 用两个二值信号量S1和S2实现一个计数信号量S
      - S1用于互斥，S2用于阻塞
      - S.value用于计数，记录up减去down的次数
      - 链表S.list来记录阻塞在这个计数信号量上的进程。
    - wait(S)的实现：进程先waitb(S1)来获得S的独占访问权，并把S.value减1。如果S.value大于等于0，直接signalb(S1)即可；否则，记录在链表再signalb(S1)，然后waitb(S2)阻塞这个进程
    - signal(S)的实现：进程同样先waitb(S1)，S.value加1，若其大于0，没有阻塞进程，直接signalb(S1)即可；否则S.value小于等于0，把链表中一个进程移出，然后依次signalb(S2)、signalb(S1)

Semaphores

- **信号量的实现**
  - 计数信号量的实现
    - 用两个二值信号量S1和S2实现一个计数信号量S
      - S1用于互斥，S2用于阻塞
      - S.value用于计数，记录up减去down的次数
      - 链表S.list来记录阻塞在这个计数信号量上的进程。
    - 数据结构

```
semaphore S;                /*计数与列表*/
binary-semaphore S1, S2; /*分别用于互斥与阻塞*/
```

    - 初始化：

```
S1.value = 1;        /* 初始空闲 */
S1.list = NULL;
S2.value = 0;        /* 初始不可用 */
S2.list = NULL;
S.value = initial value of S;
```

Semaphores

- **信号量的实现**
  - 计数信号量的实现
    - wait(S):

```
waitb(S1);
S.value--;
if (S.value < 0) {
    block();
    add to S.list;
    signalb(S1);
    waitb(S2);
}
signalb(S1);
```

    - signal(S):

```
waitb(S1);
S.value++;
if (S.value <= 0) {
    remove a process P from S.list;
    wakeup(P);
    signalb(S2);
} else
    signalb(S1);
```

回顾二值信号量的实现

waitb(S):

```
if (S.value == 1) {
    S.value = 0;
} else {
    block();
    add this process to S.list;
}
```

signalb(S):

```
if (S.list is empty) {
    S.value = 1;
} else {
    remove a process P from S.list;
    wakeup(P);
}
```

Semaphores

- **信号量的实现**
  - 计数信号量的实现
    - wait(S):

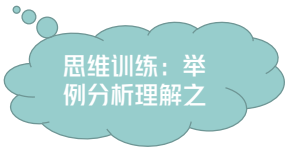
```
waitb(S1);
S.value--;
if (S.value < 0) {
    block();
    add to S.list;
    signalb(S1);
    waitb(S2);
}
signalb(S1);
```

    - signal(S):

```
waitb(S1);
S.value++;
if (S.value <= 0) {
    remove a process P from S.list;
    wakeup(P);
    signalb(S2);
} else
    signalb(S1);
```

wait(S)的实现
进程先waitb(S1)来获得S的独占访问权，并把S.value减1。如果S.value大于等于0，直接signalb(S1)即可；否则，记录在链表再signalb(S1)，然后waitb(S2)阻塞这个进程

signal(S)的实现
进程同样先waitb(S1)，S.value加1，若其大于0，没有阻塞进程，直接signalb(S1)即可；否则S.value小于等于0，把链表中一个进程移出，然后依次signalb(S2)、signalb(S1)



Semaphores

- **作为通用同步工具的信号量**
  - 假设想在进程P_i中执行代码块A之后，在进程P_j中执行代码块B
    - 共享数据：

```
semaphore flag;
```

    - 初始化：

```
flag.value = 0;
flag.list = NULL;
```

    - P_i

```
…
A
signal(flag);
…
```

    - P_j

```
…
wait(flag);
B
…
```

Semaphores

- **死锁与饥饿**
  - 死锁
    - 两个或多个进程正在无限期地等待一个事件，而该事件只能由一个等待进程引起。（我们稍后再讨论）
      - 例如，S和Q是初始化为1的两个信号量。

P_0:	P_1:
`wait(S); /* S-- */`	`wait(Q); /* Q-- */`
`wait(Q); /* Q-- */`	`wait(S); /* S-- */`
… …	… …
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

这里可能会发生死锁。

  - 饥饿
    - 进程可能永远不会从其挂起的信号量队列（例如LIFO）中删除
  - 优先级反转
    - 低优先级进程持有高优先级进程所需锁时的调度问题

Semaphores

- **信号量问题**
  - 信号量为实现互斥和协作进程提供了强大的工具。
  - 但wait()和signal()分散在多个进程中，很难理解它们的影响。
    - 所有进程中的用法必须正确（顺序正确、变量正确、无遗漏）
  - 信号量操作的不正确使用：
    - signal (mutex) … wait (mutex)
    - wait (mutex) … wait (mutex)
    - 缺少signal (mutex)或wait (mutex)（或两者兼有）
  - 一个坏的（或恶意的）进程可以使整个进程集失败。

Classical Problems

- **经典问题**
  - 三个著名的问题被用作一大类并发控制问题的示例。
    - 有界缓冲区
    - 读者和写者
    - 哲学家就餐
  - 这些问题几乎可用于测试每一个新提出的同步方案。
  - 使用信号量进行同步，是展示此类解决方案的传统方式。然而，这些解决方案的实际实现可以使用互斥锁代替二值信号量。

Classical Problems

- **有界缓冲区问题**
  - 回顾：竞争条件的生产者消费者问题

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count – 1;                    /* decrement count of items in buffer */
          if (count == N – 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

Classical Problems

- **有界缓冲区问题**
  - 有界缓冲区生产者消费者问题
    - 需要三个信号量：
      - 一个信号量mutex（初始化为1），用于缓冲区访问互斥
      - 一个信号量full（初始化为0），用于同步商品数量。
      - 一个信号量empty（初始化为n），用于同步可用空间数
    - 共享数据

```
semaphore mutex = 1;
semaphore empty = BUFFER_SIZE;
semaphore full = 0;
```

Classical Problems

- **有界缓冲区问题**
  - 有界缓冲区生产者消费者问题
    - 生产者进程

```
do {
        …
    produce an item in next_produced
        …
      /* 必须先 wait(empty) 再 wait(mutex) */
    wait(empty);
    wait(mutex);
        …
    add next_produced to buffer
        …
    signal(mutex);
    signal(full);
} while (TRUE);
```

Classical Problems

- **有界缓冲区问题**
  - 有界缓冲区生产者消费者问题
    - 消费者进程

```
do {
        /* 必须先 wait(full) 再 wait(mutex) */
    wait(full);
    wait(mutex);
        …
    remove an item from buffer to next_consumed
        …
    signal(mutex);
    signal(empty);
        …
    consume the item in next_consumed
        …
} while (TRUE);
```

Classical Problems

- **有界缓冲区问题**
  - 有界缓冲区生产者消费者问题
    - 从消费者的角度：将signal(empty)放在消费者的临界区内部（而不是外部）没有影响，因为生产者必须总是等待两个信号量才能继续执行
    - 消费者必须先wait(full)再wait(mutex)，否则，如果消费者在缓冲区为空（full.value==0）时进入临界区（持有mutex并等待full，而生产者需要mutex才能生产），就会发生死锁
    - 生产者必须先wait(empty)再wait(mutex)，否则，如果生产者在缓冲区已满（empty.value==0）时进入临界区（持有mutex并等待empty），就会发生死锁
    - 结论：使用信号量是一门很难的艺术。。。

Classical Problems

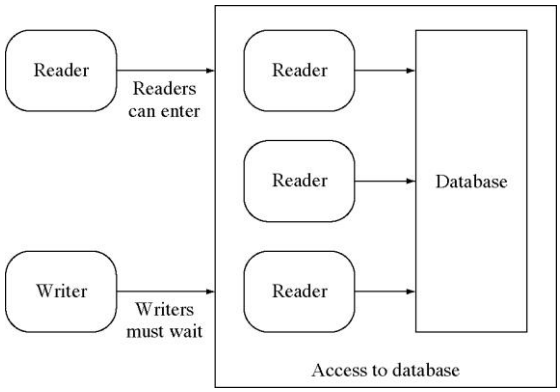
- **读者写者问题**
  - 数据集（数据存储库）在多个并发进程之间共享：
    - 读者
      - 只读取数据集；它们不执行任何更新。
    - 写者
      - 既能读又能写。
  - 问题
    - 允许多个读者同时读取。
    - 同时只有一个写者可以访问共享数据。

Classical Problems

- **读者写者问题**
  - 读者写者动态
    - 正在运行任意数量的读者活动和写者活动。
    - 任何时候，读者活动都可能希望读取数据。
    - 任何时候，写者活动都可能需要修改数据。
    - 任意数量的读者都可以同时访问数据。
    - 在写者写入期间，其他读者或写者都不能访问共享数据。

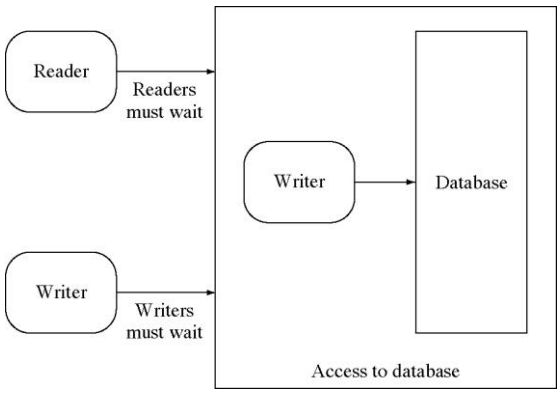
Classical Problems

- **读者写者问题**
  - 有活跃读者时，写者必须等待



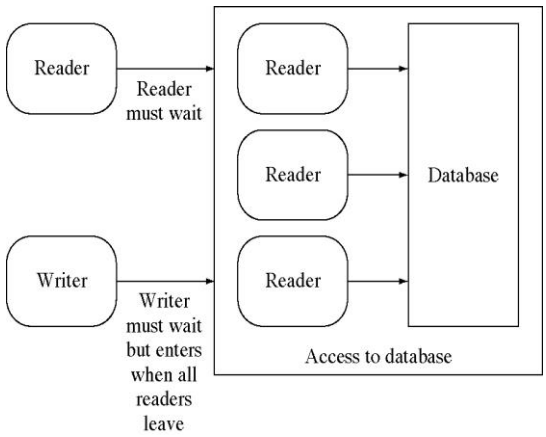
Classical Problems

- **读者写者问题**
  - 有活跃写者时，读者和其他写者必须等待



Classical Problems

- **读者写者问题**
  - 有等待的写者时，其他读者应该等待吗？
    - 不等会怎样？



Classical Problems

- **读者写者问题**
  - 具有不同的读者和写者偏好的不同版本：
    - 第一种读者写者问题
      - 它要求除非写者已获得对共享数据的访问权，否则任何读者都不会一直等待。
      - 这种情况的解决方案里，写者们可能会饥饿。
    - 第二种读者写者问题
      - 它要求一旦写者准备好了，新的读者就不能开始阅读了。
      - 在这种情况下，读者可能会饥饿。
  - 回顾：饥饿的概念
    - 进程可能永远不会从其挂起的信号量队列中删除。

Classical Problems

- **读者写者问题**
  - 第一种读者写者问题的解决方案
    - read_count（初始化为0）计数器跟踪当前正在读取的进程数
    - mutex信号量（初始化为1）为更新read_count提供互斥。
    - rw_mutex信号量（初始化为1）为写者提供互斥；它也被第一个进入或最后一个退出临界区的读者使用。

Classical Problems

- **读者写者问题**
  - 第一种读者写者问题的解决方案
    - 共享数据

```
semaphore mutex = 1;
semaphore rw_mutex = 1;
int read_count = 0;
```

    - 写者进程：

```
do {
    wait(rw_mutex);
        …
    writing is performed
        …
    signal(rw_mutex);
} while(TRUE);
```

Classical Problems

- **读者写者问题**
  - 第一种读者写者问题的解决方案
    - 读者进程：

```
do {
    wait(mutex);
    read_count++;
    if (read_count == 1)
        wait(rw_mutex);
    signal(mutex);
        …
    reading is performed
        …
    wait(mutex);
    read_count--;
    if (read_count == 0)
        signal(rw_mutex);
    signal(mutex);
} while(TRUE);
```

第一个读者需要写锁，确保没有写者在写；

最后一个读者释放写锁，让写者可写；

Classical Problems

- **读者写者问题**
  - 读者写者锁
    - 在某些系统上，读者写者问题及其解决方案已被概括为提供读者写者锁。
    - 获取读者写者锁需要指定锁的模式：读或写访问。
      - 仅读取共享数据请求读模式锁；而在写模式锁下修改共享数据。
      - 多个进程可以同时获取读模式锁，但只有一个进程可以获取写模式锁，因为写者需要独占访问。
  - 读者写者锁在以下情况下最有用：
    - 容易确定哪些进程只读取共享数据，哪些进程只写入共享数据
    - 读者比写者多
      - 读者写者锁通常需要比信号量或互斥锁更多的开销来建立
      - 多个读者并发性的增加弥补了设置读者写者锁的开销

Classical Problems

- **哲学家就餐问题**
  - 哲学家一生都在思考和饮食之间交替。五位哲学家围坐在一张圆桌旁，共享一碗饭。每人前面都有一个碟子，相邻两位哲学家之间只有一根筷子，所以共有五根筷子。
  - 当哲学家思考时，她不会与同事互动。有时，哲学家会感到饥饿，并试图拿起离她最近的两根筷子。
  - 哲学家一次只能拿起一根筷子，但她不能拿已在别人手中的筷子。
  - 当一个饥饿的哲学家同时拥有两根筷子时，她就吃饭，且不会松开筷子。吃完后，她放下两根筷子，开始重新思考。

Classical Problems

- **哲学家就餐问题**
  - 每位哲学家的用餐动态

```
do {
    thinks for a while;
    gets the left chopstick;
    gets the right chopstick;
    eats for a while;
    puts the two chopsticks down;
} while (TRUE)
```

  - 挑战在于在满足筷子需求的同时避免死锁和饥饿。这说明了在没有死锁和饥饿的情况下在进程之间分配资源的困难。
  - 如果每个人都试图马上拿到筷子，就会出现死锁：每个人都有一根左筷子，并且被卡住了，因为每个右筷子都是别人的左筷子。
  - 一个简单的解决方案是每根筷子用一个信号量表示。每个哲学家都是一个试图通过对信号量执行wait()操作来获取筷子的进程。她通过对对当的信号量执行signal()操作来释放筷子。

Classical Problems

- **哲学家就餐问题**
  - 哲学家就餐问题的解决方案
    - 共享数据

```
semaphore chopstick[5];
```

    - 初始化

```
for (int i = 0; i < 5; i++) {
    chopstick[i].value = 1;
}
```

    - 进程P_i

```
do {
    think for a while
    wait(chopstick[i]);   /* 拿起左筷子 */
    wait(chopstick[(i + 1) mod 5]); /*拿起右筷子，逆时针方向*/
    eat for a while
    signal(chopstick[(i + 1) mod 5]); /* 放下右筷子 */
    signal(chopstick[i]); /* 放下左筷子 */
} While (TRUE)
```

Classical Problems

- **哲学家就餐问题**
  - 尽管此解决方案保证没有两个邻座同时进食，但可能会造成死锁
    - 假设五位哲学家中的每一位都同时抓住她的左筷子。筷子的所有信号量现在都等于0。对右筷子的任何请求都将永远推迟
  - 避免死锁的可能解决方案：
    - 方案1. 允许最多四位哲学家同时坐在桌子旁吃饭。那么，当其他三个人拿着一根筷子时，一个哲学家总是可以吃东西。
    - 方案2. 只有在两种筷子都可用的情况下，才允许哲学家拿起筷子（必须在临界区拿）。
    - 方案3. 使用非对称解决方案：
      - 奇数的哲学家先拿起左筷子，然后拿起右筷子。偶数的哲学家先拿起右筷子，然后拿起左筷子。
  - 任何解决方案都必须防止饥饿的可能性。无死锁的解决方案不一定能消除饥饿的可能性。

Classical Problems

- **哲学家就餐问题**
  - **解决方案1的实现**
    - **使用另一信号量T，表示“坐在桌子旁”的哲学家数，限制为4**
    - **共享数据**

```
semaphore chopstick[5], T;
```

    - **初始化**

```
for (int i = 0; i < 5; i++)
    chopstick[i].value = 1;
T.value = 4;
```

    - **进程P_i**

```
do {
      /* think for a while */
    wait(T);
    wait(chopstick[i]);
    wait(chopstick[(i + 1) mod 5]);
      /* eat for a while */
    signal(chopstick[(i + 1) mod 5]);
    signal(chopstick[i]);
    signal(T);
} While (TRUE);
```