

# 并发

## 第三十一章——信号量

刘玉峰

[Fx\\_yfliu@163.](mailto:Fx_yfliu@163.com)湖

南大学

# 信号量

- 锁和条件变量可以解决广泛的相关和有趣的并发问题。
- **Dijkstra**和他的同事们发明了**信号量**，作为所有与同步相关的事情的单一原语。
- 可以将信号量用作锁和条件变量。

### 关键问题：如何使用信号量？

如何使用信号量代替锁和条件变量？什么是信号量？什么是二值信号量？用锁和条件变量来实现信号量是否简单？不用锁和条件变量，如何实现信号量？

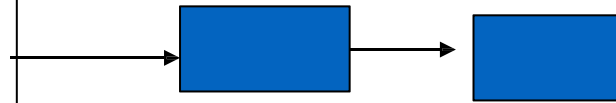
# Semaphores

- 1965年E.W.Dijkstra提出了新的同步工具--信号量和P操作（荷兰语的测试Proberen）、V操作（荷兰语的增量Verhogen）。
- 在许多操作系统的教程中，P操作又被称为wait操作，V操作有被称为signal操作

# 信号量

信号量的值 (-2)

信号量队列指针



- 将信号量定义为记录

```
int value;  
struct process* Q;  
} sample;
```

- 信号量的数据结构:
- 两个分量: 一个是信号量的值, 另一个是信号量队列的队列指针。
- 信号量的值的含义:
- 如果  $S.value \geq 0$ , 则表明系统有多少个临界资源可以分配给请求的进程。
- 如果  $S.value < 0$ , 表示等待使用该临界资源的进程数。

- P和V操作原语定义:
- P(s): 将信号量s.value减1, 若结果小于0, 则调用P(s)的进程被置成等待信号量s的状态, 并把该进程挂到信号量的等待队列上。
- V(s): 将信号量s.value加1, 若结果不大于0, 则释放一个等待信号量s的进程。

对信号量整数值的修改必须是原子操作

# 信号量的定义

- 信号量是一个带有整数值的对象，我们可以用两个例程来操作它;在POSIX标准中，这些例程是`sem_wait()`和`sem_post()`。

```
1  #include <semaphore.h>
2  sem_t s;
3  sem_init(&s, 0, 1);
```

其中申明了一个信号量 `s`，通过第三个参数，将它的值初始化为 1。`sem_init()` 的第二个参数，在我们看到的所有例子中都设置为 0，表示信号量是在同一进程的多个线程共享的。

- `sem_wait()` 要么立刻返回（调用 `sem_wait()` 时，信号量的值大于等于 1），要么会让调用线程挂起，直到之后的一个 `post` 操作。当然，也可能多个调用线程都调用 `sem_wait()`，因此都在队列中等待被唤醒。。
- `sem_post()` 增加信号量的值，如果有等待线程，唤醒其中一个。
- 当信号量的值为负数时，其绝对值就是等待线程的个数。

```
1  int sem_wait(sem_t *s) {
2      decrement the value of semaphore s by one
3      wait if value of semaphore s is negative
4  }
5
6  int sem_post(sem_t *s) {
7      increment the value of semaphore s by one
8      if there are one or more threads waiting, wake one
9  }
```

### **Semaphore: Definitions of Wait and Post**



# 用P、V原语实现进程互斥

通过我们对临界区访问过程的分析，信号量机制中P原语相当于进入区操作，V原语相当于退出区操作。

用P、V原语实现进程互斥就是将**临界区**置于**P**和**V**两个原语操作之间。

进入时执行P操作，使信号量 $S.value$ 减1、如果 $S.value \geq 0$ ，则进入临界区，否则不可进入。进程退出临界区时，执行V操作，使信号量 $S.value + 1$ ，表示释放临界资源。



# 二进制信号量（锁）

```
1  sem_t m;  
2  sem_init(&m, 0, X); // initialize semaphore to X; what should X be?  
3  
4  sem_wait(&m);  
5  // critical section here  
6  sem_post(&m);
```

- 但是，要实现这一点，关键是**信号量****m**的初始值（在图中初始化为**X**）。**X**应该是什么？

表 31.1

追踪线程：单线程使用一个信号量

信号量的值	线程 0	线程 1
1		
1	调用 sem_wait()	
0	sem_wait() 返回	
0	(临界区)	
0	调用 sem_post()	
1	sem_post() 返回	

表 31.2

追踪线程：两个线程使用一个信号量

值	线程 0	状态	线程 1	状态
1		运行		就绪
1	调用 sem_wait()	运行		就绪
0	sem_wait()返回	运行		就绪
0	(临界区：开始)	运行		就绪
0	中断；切换到→T1	就绪		运行
0		就绪	调用 sem_wait()	运行
-1		就绪	sem 减 1	运行
-1		就绪	(sem<0) →睡眠	睡眠
-1		运行	切换到→T0	睡眠
-1	(临界区：结束)	运行		睡眠
-1	调用 sem_post()	运行		睡眠
0	增加 sem	运行		睡眠
0	唤醒 T1	运行		就绪
0	sem_post()返回	运行		就绪
0	中断；切换到→T1	就绪		运行
0		就绪	sem_wait()返回	运行
0		就绪	(临界区)	运行
0		就绪	调用 sem_post()	运行
1		就绪	sem_post()返回	运行

- 请注意，当**线程1**试图获取已经持有的锁时，它将进入睡眠状态；只有当**线程0**再次运行时，**线程1**才能被唤醒并可能再次运行。

# n个过程的临界截面

- 共享数据:

semaphore mutex;//初始mutex= 1

- 工艺 $P_i$ :

做什么

public void run ();  
临界截面

public void run ();  
剩余部分

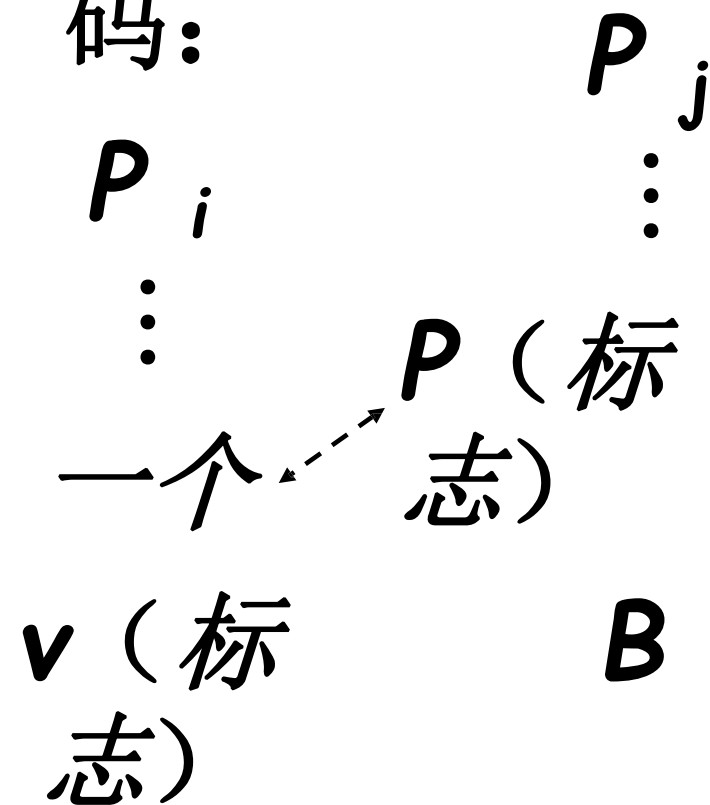
} while (1);

锁, 互斥

# 作为通用同步工具的信号量

- 仅在 **A** 在  $P_i$  中执行之后才在  $P_j$  中执行 **B**

- 验证  
码:



# 信号量作为条件变量

- 当线程希望停止其进程以等待条件变为真时，信号量也很有用。

```
1  sem_t s;
2
3  void *
4  child(void *arg) {
5      printf("child\n");
6      sem_post(&s); // signal here: child is done
7      return NULL;
8  }
9
10 int
11 main(int argc, char *argv[]) {
12     sem_init(&s, 0, X); // what should X be?
13     printf("parent: begin\n");
14     pthread_t c;
15     Pthread_create(c, NULL, child, NULL);
16     sem_wait(&s); // wait here for child
17     printf("parent: end\n");
18     return 0;
19 }
```

## A Parent Waiting For Its Child

- 这个信号量的**初始值**应该是多少？

表 31.3

追踪线程：父线程等待子线程（场景 1）

值	父线程	状态	子线程	状态
0	create(子线程)	运行	(子线程产生)	就绪
0	调用 sem_wait()	运行		就绪
-1	sem 减 1	运行		就绪
-1	(sem<0)→ 睡眠	睡眠		就绪
-1	切换到→子线程	睡眠	子线程运行	运行
-1		睡眠	调用 sem_post()	运行
0		睡眠	sem 增 1	运行
0		就绪	wake(父线程)	运行
0		就绪	sem_post()返回	运行
0		就绪	中断：切换到→父线程	就绪
0	sem_wait()返回	运行		就绪

## Thread Trace: Parent Waiting For Child (Case 1)

```

1  sem_t s;
2
3  void *
4  child(void *arg) {
5      printf("child\n");
6      sem_post(&s); // signal here: child is done
7      return NULL;
8  }
9
10 int
11 main(int argc, char *argv[]) {
12     sem_init(&s, 0, X); // what should X be?
13     printf("parent: begin\n");
14     pthread_t c;
15     Pthread_create(c, NULL, child, NULL);
16     sem_wait(&s); // wait here for child
17     printf("parent: end\n");
18     return 0;
19 }
```



表 31.4

追踪线程：父线程等待子线程（场景 2）

值	父线程	状态	子线程	状态
0	create（子线程）	运行	（子线程产生）	就绪
0	中断：切换到→子线程	就绪	子线程运行	运行
0		就绪	调用 sem_post()	运行
1		睡眠	sem 增 1	运行
1		就绪	wake(没有线程)	运行
1		就绪	sem_post()返回	运行
1	父线程运行	运行	中断：切换到→父线程	就绪
1	调用 sem_wait()	运行		就绪
0	sem 减 1	运行		就绪
0	(sem>=0)→不用睡眠	运行		就绪
0	sem_wait()返回	运行		就绪

```

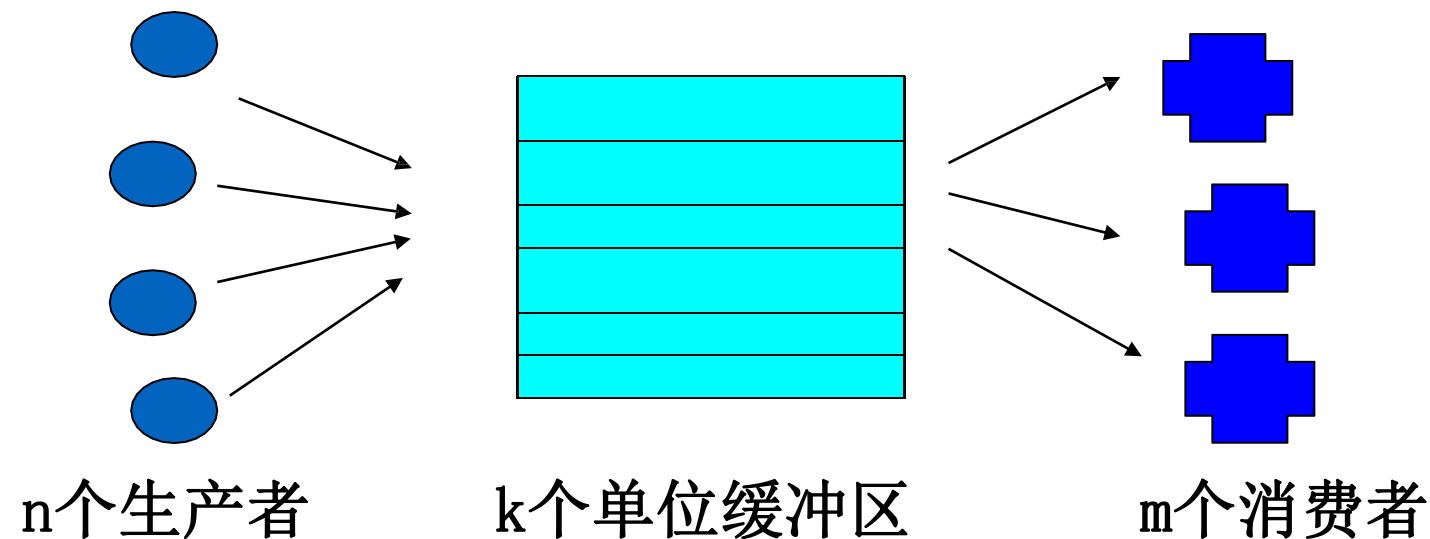
1  sem_t s;
2
3  void *
4  child(void *arg) {
5      printf("child\n");
6      sem_post(&s); // signal here: child is done
7      return NULL;
8  }
9
10 int
11 main(int argc, char *argv[]) {
12     sem_init(&s, 0, X); // what should X be?
13     printf("parent: begin\n");
14     pthread_t c;
15     Pthread_create(c, NULL, child, NULL);
16     sem_wait(&s); // wait here for child
17     printf("parent: end\n");
18     return 0;
19 }

```

# 生产者--消费者问题表述

把并发进程的互斥和同步问题一般化，我们可以得出一个抽象化的一般模型，即生产者-消费者问题。

- 有 $n$ 个生产者和 $m$ 个消费者，连接在一个有 $k$ 个单位缓冲区的有界缓冲上。其中， $p_i$ 和 $c_j$ 都是并发进程，只要缓冲区未满，生产者 $p_i$ 生产的产品就可投入缓冲区;只要缓冲区不空，消费者进程 $c_j$ 就可从缓冲区取走并消耗产品。



# 生产者/消费者（有界缓冲区）问题

- 我们在解决这个问题的第一次尝试引入了两个信号量，**空**和**满**，线程将使用它们来指示缓冲区条目何时被清空或填满。

```

1  int buffer[MAX];
2  int fill = 0;
3  int use  = 0;
4
5  void put(int value) {
6      buffer[fill] = value;    // line f1
7      fill = (fill + 1) % MAX; // line f2
8  }
9
10 int get() {
11     int tmp = buffer[use];    // line g1
12     use = (use + 1) % MAX;    // line g2
13     return tmp;
14 }
15
16 sem_t empty;
17 sem_t full;
18
19 void *producer(void *arg) {
20     int i;
21     for (i = 0; i < loops; i++) {
22         sem_wait(&empty);    // line P1
23         put(i);              // line P2
24         sem_post(&full);     // line P3
25     }
26 }
27
28 void *consumer(void *arg) {
29     int i, tmp = 0;
30     while (tmp != -1) {
31         sem_wait(&full);    // line C1
32         tmp = get();        // line C2
33         sem_post(&empty);   // line C3
34         printf("%d\n", tmp);
35     }
36 }
37
38 int main(int argc, char *argv[]) {
39     // ...
40     sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
41     sem_init(&full, 0, 0);    // ... and 0 are full
42     // ...
43 }

```

让我们首先假设**MAX=1**  
（数组中只有一个缓冲区），看看是否有效。

假设有**两个线程**，一个生产者和一个消费者。

```

1  int buffer[MAX];
2  int fill = 0;
3  int use  = 0;
4
5  void put(int value) {
6      buffer[fill] = value;    // line f1
7      fill = (fill + 1) % MAX; // line f2
8  }
9
10 int get() {
11     int tmp = buffer[use];    // line g1
12     use = (use + 1) % MAX;    // line g2
13     return tmp;
14 }
15
16 sem_t empty;
17 sem_t full;
18
19 void *producer(void *arg) {
20     int i;
21     for (i = 0; i < loops; i++) {
22         sem_wait(&empty);    // line P1
23         put(i);              // line P2
24         sem_post(&full);     // line P3
25     }
26 }
27
28 void *consumer(void *arg) {
29     int i, tmp = 0;
30     while (tmp != -1) {
31         sem_wait(&full);    // line C1
32         tmp = get();        // line C2
33         sem_post(&empty);   // line C3
34         printf("%d\n", tmp);
35     }
36 }
37
38 int main(int argc, char *argv[]) {
39     // ...
40     sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
41     sem_init(&full, 0, 0);    // ... and 0 are full
42     // ...
43 }

```

假设 **consumer** 先运行. 阻塞在C1 (full从0 变为-1), 等待另一个线程调用 `sem_post(&full)`.

假设**producer**随后执行. 执行到P1 (empty 为 1), 继续执行到 P2, 向缓冲区放入数据, 然后执行P3调用 `sem_post(&full)`, 唤醒consumer.

这时可能有两种情况. 第一, producer继续执行多次循环到P1并阻塞(empty为0); 第二, consumer开始执行调用`sem_wait(&full)`(C1),取数据. 这两种情况都**正常** (包括多个**consumer**和**producer**).

```

1  int buffer[MAX];
2  int fill = 0;
3  int use  = 0;
4
5  void put(int value) {
6      buffer[fill] = value;    // line f1
7      fill = (fill + 1) % MAX; // line f2
8  }
9
10 int get() {
11     int tmp = buffer[use];    // line g1
12     use = (use + 1) % MAX;    // line g2
13     return tmp;
14 }

1  sem_t empty;
2  sem_t full;
3
4  void *producer(void *arg) {
5      int i;
6      for (i = 0; i < loops; i++) {
7          sem_wait(&empty);    // line P1
8          put(i);              // line P2
9          sem_post(&full);      // line P3
10     }
11 }

12
13 void *consumer(void *arg) {
14     int i, tmp = 0;
15     while (tmp != -1) {
16         sem_wait(&full);      // line C1
17         tmp = get();          // line C2
18         sem_post(&empty);     // line C3
19         printf("%d\n", tmp);
20     }
21 }

22
23 int main(int argc, char *argv[]) {
24     // ...
25     sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
26     sem_init(&full, 0, 0);    // ... and 0 are full
27     // ...
28 }

```

假设**MAX**大于1, 并且假设  
 有多个producer和  
 consumer. 问题来了: **a**  
**race condition.** 哪里?

假设两个producer( $P_a$ 和 $P_b$ )同  
 时调用put(). 假设 $P_a$ 先执  
 行f1, 但在执行f2之前被中  
 断,  $P_b$ 开始执行, 也在f1处写  
 入数据, 这将覆盖 $P_a$ 写入的  
 数据.

# 解决方案： 添加互斥

- 我们在这里忘记了互相排斥。
- 填充缓冲区和将索引递增到缓冲区中是一个关键部分，因此必须小心防范。



```

1  sem_t empty;
2  sem_t full;
3  sem_t mutex;
4
5  void *producer(void *arg) {
6      int i;
7      for (i = 0; i < loops; i++) {
8          sem_wait(&mutex);          // line p0 (NEW LINE)
9          sem_wait(&empty);          // line p1
10         put(i);                    // line p2
11         sem_post(&full);           // line p3
12         sem_post(&mutex);          // line p4 (NEW LINE)
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         sem_wait(&mutex);          // line c0 (NEW LINE)
20         sem_wait(&full);           // line c1
21         int tmp = get();           // line c2
22         sem_post(&empty);          // line c3
23         sem_post(&mutex);          // line c4 (NEW LINE)
24         printf("%d\n", tmp);
25     }
26 }
27
28 int main(int argc, char *argv[]) {
29     // ...
30     sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
31     sem_init(&full, 0, 0);    // ... and 0 are full
32     sem_init(&mutex, 0, 1);   // mutex=1 because it is a lock (NEW LINE)
33     // ...
34 }

```

### Adding Mutual Exclusion (Incorrectly)

- 在代码的整个**put()** / **get()** 部分添加一些锁，如**NEW LINE**注释所示。看起来是个正确的主意，但也**不起作用**。为什么？
- 为什么会出现死锁？程序必须按照什么顺序执行死锁？



# 避免死锁

- 假设有两个线程，一个生产者和一个消费者。消费者先运行获得锁(c0)，然后对 full 信号量执行 sem\_wait() (c1)。因为还没有数据，所以消费者阻塞。但是，重要的是，此时消费者仍然持有锁。
- 然后生产者运行，它首先对二值互斥信号量调用 sem\_wait ( ) (p0)。锁已经被持有，因此生产者也被卡住。
- 这里出现了一个循环等待。消费者持有互斥量，等待在 full 信号量上。生产者可以发送 full 信号，却在等待互斥量。因此，生产者和消费者互相等待对方——典型的死锁。

```

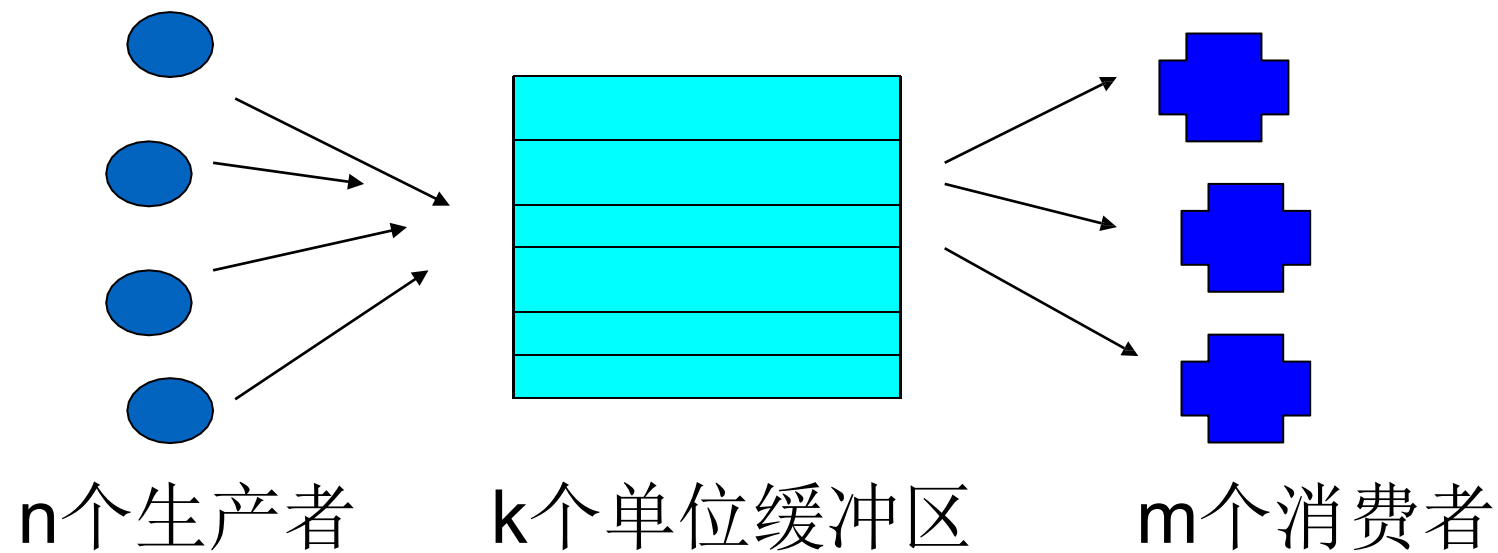
1  sem_t empty;
2  sem_t full;
3  sem_t mutex;
4
5  void *producer(void *arg) {
6      int i;
7      for (i = 0; i < loops; i++) {
8          sem_wait(&empty);          // line p1
9          sem_wait(&mutex);          // line p1.5 (MOVED MUTEX HERE...)
10         put(i);                    // line p2
11         sem_post(&mutex);          // line p2.5 (... AND HERE)
12         sem_post(&full);           // line p3
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         sem_wait(&full);            // line c1
20         sem_wait(&mutex);          // line c1.5 (MOVED MUTEX HERE...)
21         int tmp = get();            // line c2
22         sem_post(&mutex);          // line c2.5 (... AND HERE)
23         sem_post(&empty);          // line c3
24         printf("%d\n", tmp);
25     }
26 }
27
28 int main(int argc, char *argv[]) {
29     // ...
30     sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
31     sem_init(&full, 0, 0);    // ... and 0 are full
32     sem_init(&mutex, 0, 1);   // mutex=1 because it is a lock
33     // ...
34 }

```

### Adding Mutual Exclusion (Correctly)

- 要解决这个问题，我们必须**缩小锁的范围**。我们简单地将互斥体移动到临界区附近；满和空等待和信号码被留在外面。

# 多个生产者、多个消费者、共享多个缓冲区的解



- a. 设置公有信号量**mutex**，以实现互斥；
- b. 设置私有信号量**empty**和**full**，以实现同步；
- c. 赋初值：**empty=k, full=0, mutex=1**；
- d. 实现**P**、**V**操作。

# 有界缓冲区问题

## 共享数据

信号量满、空、互斥;

初始化:  $\text{full} = 0$ ,  $\text{empty} = n$ ,  $\text{mutex} = 1$

## 生产者

做什么

```
...
下一个生产项目
    ...wait
    (
empty) ;intt
    (int x) ;
    ...
将nextp添加到缓冲区
    .signal
    (
mutex) ;signa
    l (full) ;
} while (1) ;
```

## 消费者

做什么

```
wait (full)
wait (mut e
x) .;
从缓冲区移到下一个
    .signal
    (mutex) ;
signal
    (empty) ;
    ...
下一次消费该项目
    ...
} while (1) ;
```

`wait (empty)` 操作和  
`wait (mutex)` 操作的  
顺序互换, 以及  
`wait (full)` 操作和  
`wait (mutex)` 操作的  
顺序互换, 可能会怎样?

为什么需要`wait (mutex)`

假定：此时`mutex=1`，`full=n`，`empty=0`，并且生产者先占用CPU



交换`signal`的顺序不会有任何影响

生产者  
做什么

```
...  
下一个生产项目  
...wait  
(mutex);  
wait  
(empty);  
...  
将nextp添加到缓冲区  
.signal  
(  
mutex);  
signal  
(full);  
} while (1);
```

消费者  
做什么

```
wait i t  
(mutex)  
wait (full)  
...  
从缓冲区移到下一个  
.signal  
(mutex);  
signal  
(empty);  
...  
} while 次消费  
(1);
```

还有一种死锁的可能性是什么？

# P、V操作小结

## 1) 信号量的物理含义:

$S > 0$ : 表示有  $S$  个资源可用

$S = 0$ : 表示无资源可用

$S < 0$ : 则  $|S|$  表示  $S$  等待队列中的进程个数

$P(S)$ : 表示申请一个资源  $V(S)$ : 表示释放一个资源。

# P、V操作小结

2) P、V操作必须成对出现，有一个P操作就一定有一个V操作

当为互斥操作时，它们同处于同一进程

当为同步操作时，则不在同一进程中出现

如果P(S1)和P(S2)两个操作在一起，那么P操作的顺序至关重要，一个同步P操作与一个互斥P操作在一起时，同步P操作在互斥P操作前

而两个V操作无关紧要

# P、V操作小结

## (3) P、V操作的优

缺点 优点:

简单, 而且表达能力强 (用P、V操作可  
解 决任何同步互斥问题)

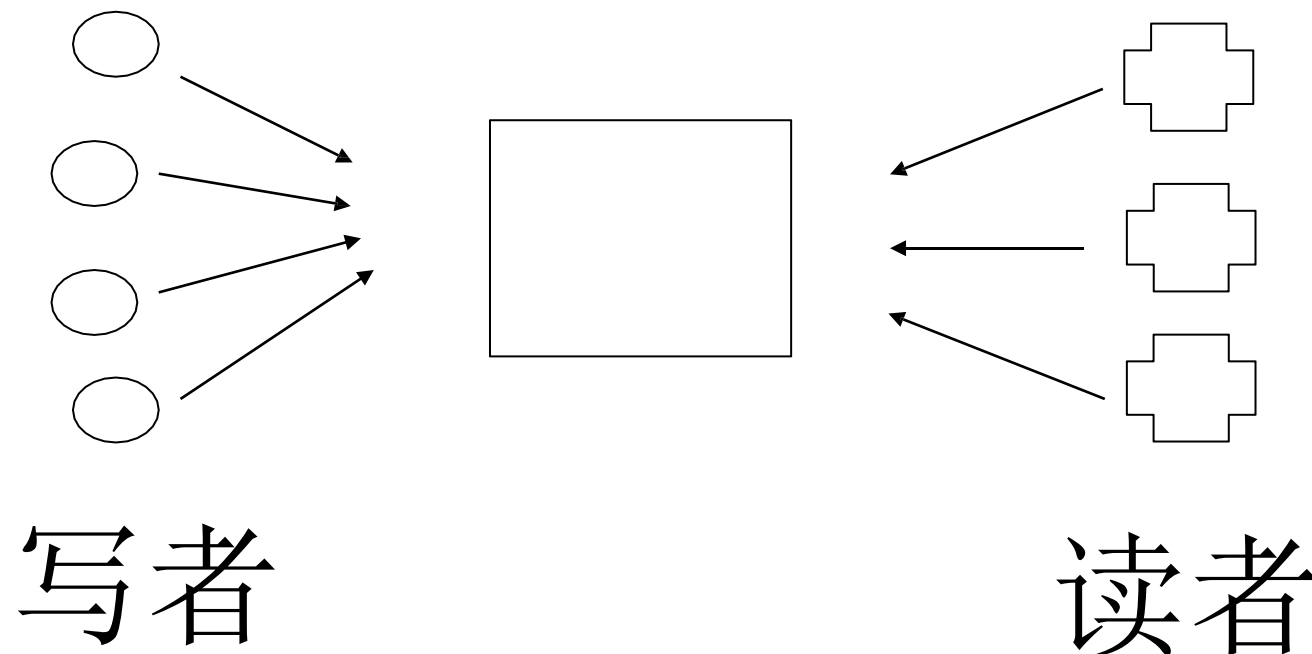
缺点:

不够安全;P、V操作使用不当会出现死锁  
;遇到复杂同步互斥问题时实现复杂



# 读者-写者问题

- 多个并发进程共享一个数据对象。其中有些进程只是读取这些共享数据的内容，而其它进程可能想要更新共享对象。我们称只是要读取共享对象的进程为读者，对共享对象进行更新操作的进程为写者，以此来区别对待这两种进程类型。



# 读者-写者问题

## 制约条件分析:

- 1、允许多个进程同时读文件（读-读允许）；
- 2、不允许在进程读文件时让另外一进程去写文件；  
有进程在写文件时不让另外一个进程去读该文件  
（“读-写”互斥）；
- 3、不允许多个写进程同时写同一文件（“写-写”互斥）。

# 读者-写者问题

因为允许多个进程同时读，系统应记录读进程的个数，而每个读进程去读文件或读文件结束后都要修改读者个数，因此，读者个数又是若干读进程的共享变量，即软件临界资源，它们也必须互斥地修改这个变量。

综上，我们定义：

- 1、**wrt**：写互斥信号量，用于“读-写”和“写-写”互斥；初值为1；
- 2、公共变量**readcount**用于记录当前正在读文件的读者数目，初值为0；
- 3、**mutex**：用于若干读进程对读者个数修改的互斥，初值为1。

共享数据:

信号量互斥体;

最初

$\text{mutex} = 1, \text{wrt} = 1, \text{readcount} = 0$

Reader:

wait

(

mutex); readcount ++

if (readcount == 1)

wait (wrt);

sort (sort);

...

执行读取

.wait

(mutex

x); readcount--;

if (readcount == 0)

信号 (wrt);

作者:

return ();

...

执行写入

...

return ();

满足写写互斥吗?

满足读读允许吗?

满足读写互斥吗?

假定先有写进程, 然后读进程想要读。

假定先有读进程, 然后写进程想要写。

现在是完美的解决方案了吗?

- 读者优先
- 只要不断的有读者来读，那么readcount就一直会大于0，那么永远不会触发signal(wrt)条件，从而导致写者饥饿。
- 该问题被称为**第一读者优先问题**

### **写者优先：**（作业）

1. 写者线程的优先级高于读者线程。
2. 当写者到来时，只有那些已经获得授权的读进程才被允许完成 它们的操作，写者之后到来的读者将被推迟，直到写者完成。
3. 当没有写者进程时读者进程应该能够同时读取文件。

### **公平竞争：**（作业）

1. 优先级相同。
2. 写者、读者互斥访问。
3. 只能有一个写者访问临界区。
4. 可以有多个读者同时访问临界资源。

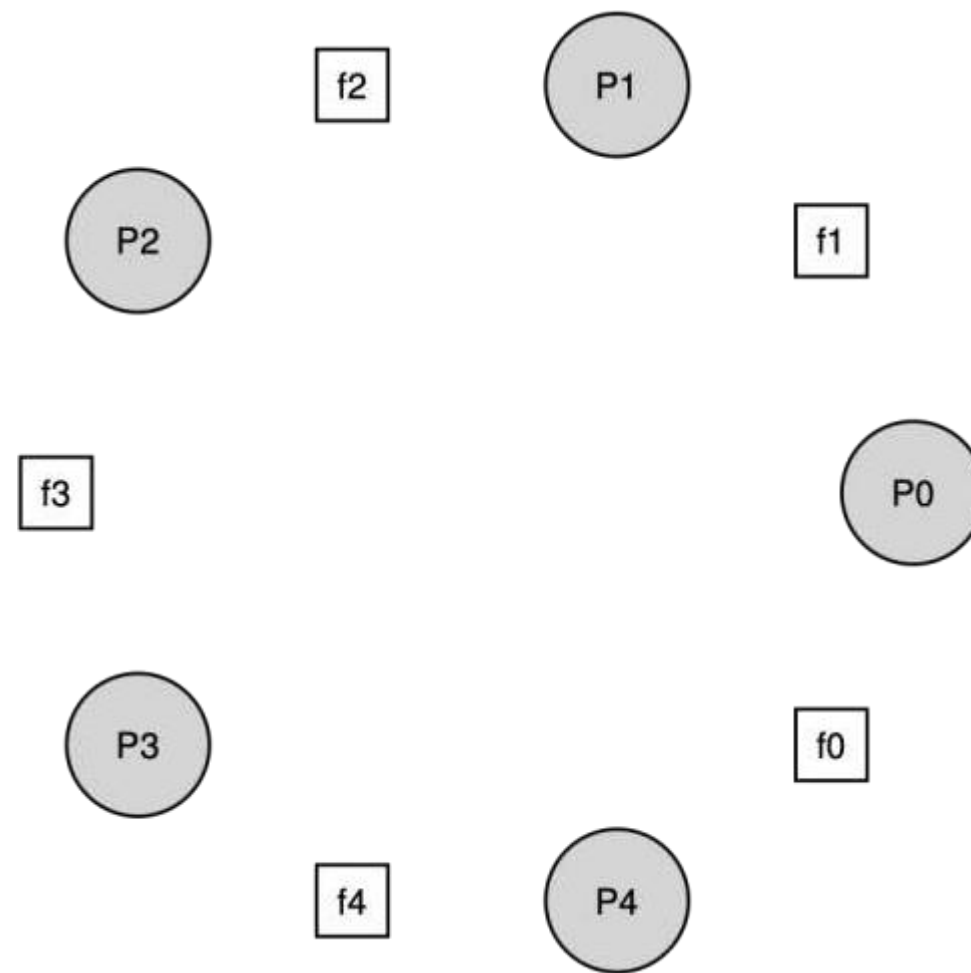
# 读写器锁

```
1  typedef struct _rwlock_t {
2      sem_t lock;          // binary semaphore (basic lock)
3      sem_t writelock;     // allow ONE writer/MANY readers
4      int    readers;      // #readers in critical section
5  } rwlock_t;
6
7  void rwlock_init(rwlock_t *rw) {
8      rw->readers = 0;
9      sem_init(&rw->lock, 0, 1);
10     sem_init(&rw->writelock, 0, 1);
11 }
12
13 void rwlock_acquire_readlock(rwlock_t *rw) {
14     sem_wait(&rw->lock);
15     rw->readers++;
16     if (rw->readers == 1) // first reader gets writelock
17         sem_wait(&rw->writelock);
18     sem_post(&rw->lock);
19 }
20
21 void rwlock_release_readlock(rwlock_t *rw) {
22     sem_wait(&rw->lock);
23     rw->readers--;
24     if (rw->readers == 0) // last reader lets it go
25         sem_post(&rw->writelock);
26     sem_post(&rw->lock);
27 }
28
29 void rwlock_acquire_writelock(rwlock_t *rw) {
30     sem_wait(&rw->writelock);
31 }
32
33 void rwlock_release_writelock(rwlock_t *rw) {
34     sem_post(&rw->writelock);
35 }
```

读者很容易  
饿死写者。

# 用餐哲学家

- Dijkstra提出并解决了最著名的并发问题之一。
- 这个问题很有名，因为它很有趣，而且在智力上也很有趣;然而，它的实际效用很低。



### The Dining Philosophers

- 假设有五个“哲学家”围坐在一张桌子旁。
- 每对哲学家之间有一个叉子（总共五个）。
- 哲学家们都有思考的时间，不需要叉子，也有吃饭的时间。
- 为了吃饭，哲学家需要两把叉子，一把在左边，一把在右边。



- 以下是每个哲学家的基本循环：

```
while (1) {  
    think();  
    getforks();  
    eat();  
    putforks();  
}
```

关键的挑战就是如何实现 `getforks()` 和 `putforks()` 函数，保证没有死锁，没有哲学家饿死，并且并发度更高

- 我们将使用一些辅助函数来帮助我们找到解决方案：

```
int left(int p)    { return p; }  
int right(int p)   { return (p + 1) % 5; }
```

- 我们需要一些信号量来解决这个问题。假设需要 5 个，每个餐叉一个：[`sem\_t forks\[5\]`](#).

# 破碎溶液

```
while (1) {  
    think();  
    getforks();  
    eat();  
    putforks();  
}
```

```
int left(int p) { return p; }  
int right(int p) { return (p + 1) % 5; }
```

```
sem_t forks[5]
```

```
1 void getforks() {  
2     sem_wait(forks[left(p)]);  
3     sem_wait(forks[right(p)]);  
4 }
```

```
5  
6 void putforks() {  
7     sem_post(forks[left(p)]);  
8     sem_post(forks[right(p)]);  
9 }
```

- 我们把每个信号量（在 fork 数组中）都用 1 初始化。同时假设每个哲学家知道自己的编号（p）。我们可以写出 getforks() 和 putforks() 函数。
- 为了拿到餐叉，我们依次获取每把餐叉的锁——先是左手边的，然后是右手边的。结束就餐时，释放掉锁
- 很简单，不是吗？不幸的是，在这种情况下，简单意味着破碎。你能看到出现的问题吗？

- 问题是死锁。如果每个哲学家碰巧在任何一个哲学家抓住右边的叉子之前抓住左边的叉子，每个人都会永远拿着一个叉子等待另一个叉子。

# 死锁预防策略

死锁解决方法：

至多只允许四位哲学家同时去拿左边的筷子；

规定奇数号哲学家先拿起他左边的筷子，而偶数号哲学家先拿起他右边的筷子。

仅当哲学家左右两边的筷子均可用时才允许他拿起筷子；

# 解决方案：打破依赖

- 解决这个问题的最简单的方法是**改变**至少一个哲学家获得分叉的方式;事实上，这就是**Dijkstra**自己解决问题的方法。

```
1 void getforks() {
2     if (p == 4) {
3         sem_wait(forks[right(p)]);
4         sem_wait(forks[left(p)]);
5     } else {
6         sem_wait(forks[left(p)]);
7         sem_wait(forks[right(p)]);
8     }
9 }
```

- 因为最后一个哲学家试图在左之前抓住右，所以不存在每个哲学家抓住一个叉子而被困在等待另一个叉子的情况;等待**的循环被打破了**。

# 如何实现信号量

```
1  typedef struct __Zem_t {
2      int value;
3      pthread_cond_t cond;
4      pthread_mutex_t lock;
5  } Zem_t;
6
7  // only one thread can call this
8  void Zem_init(Zem_t *s, int value) {
9      s->value = value;
10     Cond_init(&s->cond);
11     Mutex_init(&s->lock);
12 }
13
14 void Zem_wait(Zem_t *s) {
15     Mutex_lock(&s->lock);
16     while (s->value <= 0)
17         Cond_wait(&s->cond, &s->lock);
18     s->value--;
19     Mutex_unlock(&s->lock);
20 }
21
22 void Zem_post(Zem_t *s) {
23     Mutex_lock(&s->lock);
24     s->value++;
25     Cond_signal(&s->cond);
26     Mutex_unlock(&s->lock);
27 }
```

实现的 Zemaphore 和 Dijkstra 定义的信号量有一点细微区别，就是我们没有保持当信号量的值为负数时，让它反映出等待的线程数

- 奇怪的是，从信号量构建锁和条件变量是一个更棘手的命题。一些非常有经验的并发程序员尝试在Windows环境中这样做，许多不同的bug随之而来[B04]。

[B04] “Implementing Condition Variables with Semaphores”

Andrew Birrell

December 2004

*An interesting read on how difficult implementing CVs on top of semaphores really is, and the mistakes the author and co-workers made along the way. Particularly relevant because the group had done a ton of concurrent programming; Birrell, for example, is known for (among other things) writing various thread-programming guides.*

结束