Concurrency
Semaphore

Liu yufeng

Fx_yfliu@163.com

Hunan University

Semaphore

- **Locks** and **condition variables** can solve a broad range of relevant and interesting concurrency problems.
- **Dijkstra** and colleagues invented the **semaphore** as a single primitive for all things related to synchronization.
- One can use semaphores as both locks and condition variables.

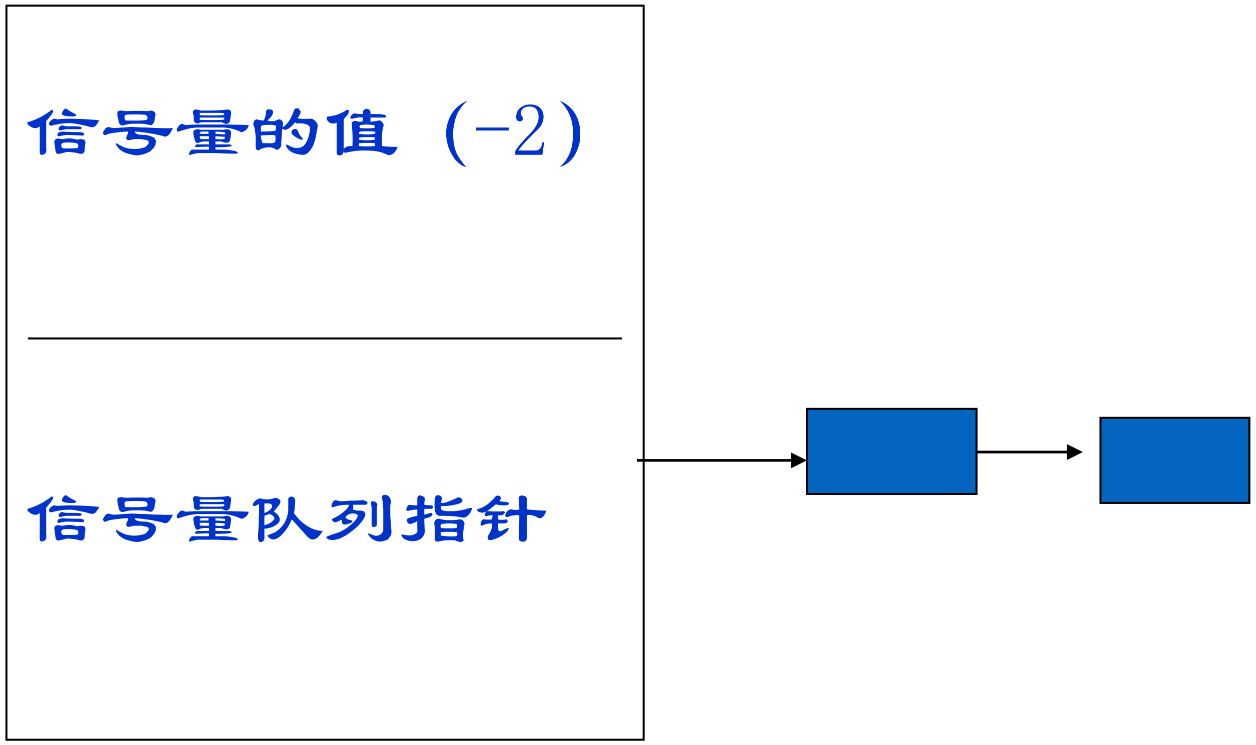
关键问题：如何使用信号量?

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

Semaphores



- **Define a semaphore as a record**

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

- 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≥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

Semaphores: A Definition

- A semaphore is as an object with **an integer value** that we can manipulate with two routines; in the POSIX standard, these routines are **`sem_wait`**`()` and **`sem_post`**`()`.

```
#include <semaphore.h>
sem_t s;
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.

```
int sem_wait(sem_t *s) {
    decrement the value of semaphore s by one
    wait if value of semaphore s is negative
}

int sem_post(sem_t *s) {
    increment the value of semaphore s by one
    if there are one or more threads waiting, wake one
}
```

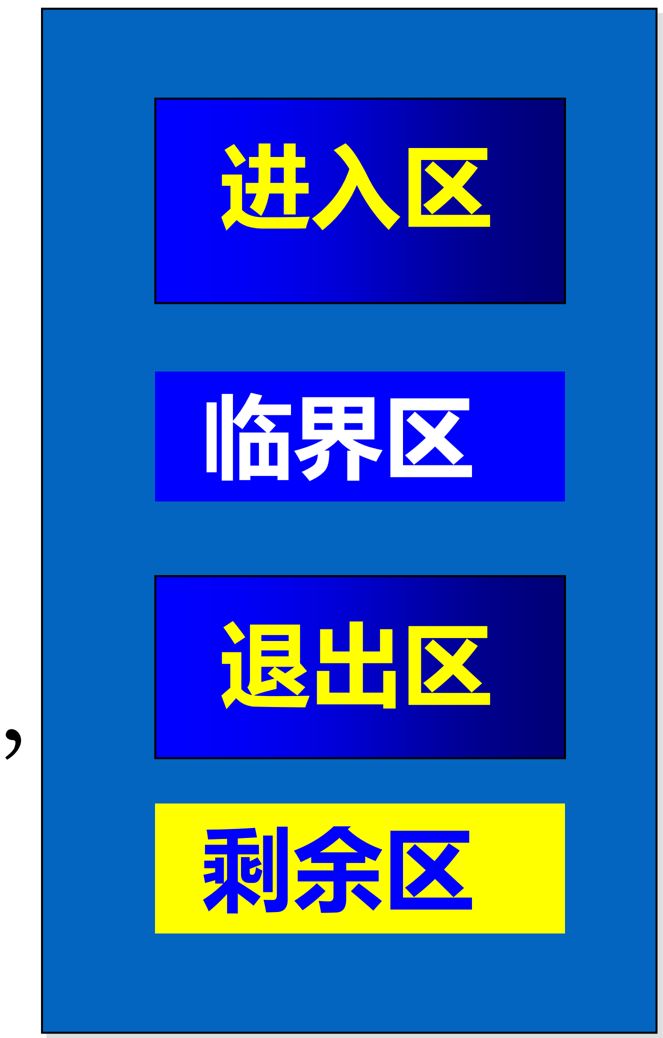
Semaphore: Definitions of Wait and Post

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>=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。



Binary Semaphores (Locks)

```
sem_t m;
sem_init(&m, 0, X); // initialize semaphore to X; what should X be?

sem_wait(&m);
// critical section here
sem_post(&m);
```

- Critical to making this work, though, is the **initial value of the semaphore** **m** (initialized to **X** in the figure). What should **X** be?

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

- Note that **Thread 1** goes into the sleeping state when it tries to acquire the already-held lock; only when **Thread 0** runs again can **Thread 1** be awoken and potentially run again.

Critical Section of *n* Processes

- **Shared data:**

  semaphore mutex; //**initially *mutex* = 1**

- ***Process Pi:***

```
do {
   P(mutex);
      critical section
   V(mutex);
      remainder section
} while (1);
```

锁，互斥

Semaphore as a General Synchronization Tool

- Execute B in P_j only after A executed in P_i

- Code:

P_i	P_j
⋮	⋮
A	$P(flag)$
$V(flag)$	B

Semaphores As Condition Variables

- Semaphores are also useful when a thread wants to halt its progress waiting for a condition to become true.

```
sem_t s;

void *
child(void *arg) {
    printf("child\n");
    sem_post(&s); // signal here: child is done
    return NULL;
}

int
main(int argc, char *argv[]) {
    sem_init(&s, 0, X); // what should X be?
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(c, NULL, child, NULL);
    sem_wait(&s); // wait here for child
    printf("parent: end\n");
    return 0;
}
```

A Parent Waiting For Its Child

- What should the **initial value** of this semaphore be?

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)

```
sem_t s;

void *
child(void *arg) {
    printf("child\n");
    sem_post(&s); // signal here: child is done
    return NULL;
}

int
main(int argc, char *argv[]) {
    sem_init(&s, 0, X); // what should X be?
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(c, NULL, child, NULL);
    sem_wait(&s); // wait here for child
    printf("parent: end\n");
    return 0;
}
```

表 31.4 追踪线程：父线程等待子线程（场景 2）

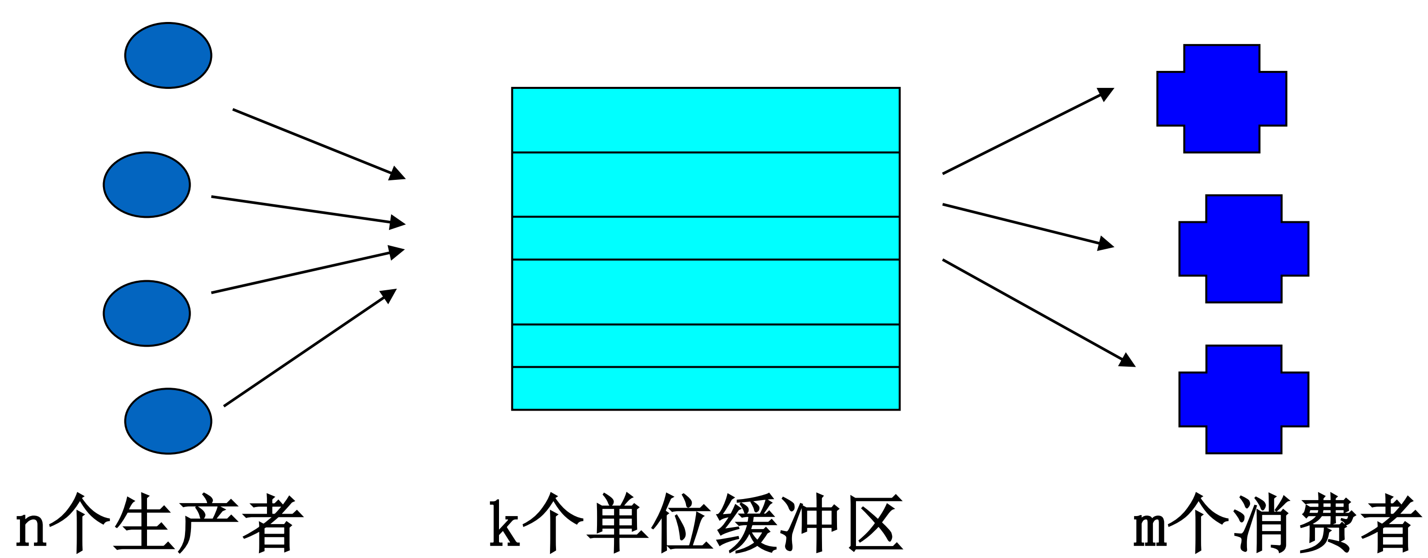
值	父线程	状态	子线程	状态
0	create（子线程）	运行	(子线程产生)	就绪
0	中断：切换到→子线程	就绪	子线程运行	运行
0		就绪	调用 sem_post()	运行
1		睡眠	sem 增 1	运行
1		就绪	wake(没有线程)	运行
1		就绪	sem_post()返回	运行
1	父线程运行	运行	中断：切换到→父线程	就绪
1	调用 sem_wait()	运行		就绪
0	sem 减 1	运行		就绪
0	(sem>=0)→不用睡眠	运行		就绪
0	sem_wait()返回	运行		就绪

```
sem_t s;

void *
child(void *arg) {
    printf("child\n");
    sem_post(&s); // signal here: child is done
    return NULL;
}

int
main(int argc, char *argv[]) {
    sem_init(&s, 0, X); // what should X be?
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(c, NULL, child, NULL);
    sem_wait(&s); // wait here for child
    printf("parent: end\n");
    return 0;
}
```

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，pi和cj都是并发进程，只要缓冲区未满，生产者pi生产的产品就可投入缓冲区；只要缓冲区不空，消费者进程cj就可从缓冲区取走并消耗产品。



The Producer/Consumer (Bounded-Buffer) Problem

- Our first attempt at solving the problem introduces **two semaphores**, **empty** and **full**, which the threads will use to indicate when a buffer entry has been emptied or filled, respectively.

```c
int buffer[MAX];
int fill = 0;
int use  = 0;

void put(int value) {
    buffer[fill] = value;     // line f1
    fill = (fill + 1) % MAX; // line f2
}

int get() {
    int tmp = buffer[use];    // line g1
    use = (use + 1) % MAX;    // line g2
    return tmp;
}
```

```c
sem_t empty;
sem_t full;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);           // line P1
        put(i);                     // line P2
        sem_post(&full);            // line P3
    }
}

void *consumer(void *arg) {
    int i, tmp = 0;
    while (tmp != -1) {
        sem_wait(&full);            // line C1
        tmp = get();                // line C2
        sem_post(&empty);           // line C3
        printf("%d\n", tmp);
    }
}

int main(int argc, char *argv[]) {
    // ...
    sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
    sem_init(&full, 0, 0);    // ... and 0 are full
    // ...
}
```

Let us first imagine that `MAX=1` (there is only one buffer in the array), and see if this works.

Imagine again there are **two threads**, a producer and a consumer.

```c
int buffer[MAX];
int fill = 0;
int use  = 0;

void put(int value) {
    buffer[fill] = value;     // line f1
    fill = (fill + 1) % MAX; // line f2
}

int get() {
    int tmp = buffer[use];    // line g1
    use = (use + 1) % MAX;    // line g2
    return tmp;
}
```

```c
sem_t empty;
sem_t full;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);            // line P1
        put(i);                      // line P2
        sem_post(&full);             // line P3
    }
}

void *consumer(void *arg) {
    int i, tmp = 0;
    while (tmp != -1) {
        sem_wait(&full);             // line C1
        tmp = get();                 // line C2
        sem_post(&empty);            // line C3
        printf("%d\n", tmp);
    }
}

int main(int argc, char *argv[]) {
    // ...
    sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
    sem_init(&full, 0, 0);    // ... and 0 are full
    // ...
}
```

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**).

```c
int buffer[MAX];
int fill = 0;
int use  = 0;

void put(int value) {
    buffer[fill] = value;     // line f1
    fill = (fill + 1) % MAX; // line f2
}

int get() {
    int tmp = buffer[use];    // line g1
    use = (use + 1) % MAX;    // line g2
    return tmp;
}
```

```c
sem_t empty;
sem_t full;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);           // line P1
        put(i);                     // line P2
        sem_post(&full);            // line P3
    }
}

void *consumer(void *arg) {
    int i, tmp = 0;
    while (tmp != -1) {
        sem_wait(&full);            // line C1
        tmp = get();                // line C2
        sem_post(&empty);           // line C3
        printf("%d\n", tmp);
    }
}

int main(int argc, char *argv[]) {
    // ...
    sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
    sem_init(&full, 0, 0);    // ... and 0 are full
    // ...
}
```

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**.

A Solution: Adding Mutual Exclusion

- What we’ve forgotten here is **mutual exclusion**.
- The filling of a buffer and incrementing of the index into the buffer is a **critical section**, and thus must be guarded carefully.

```
sem_t empty;
sem_t full;
sem_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);           // line p0 (NEW LINE)
        sem_wait(&empty);           // line p1
        put(i);                     // line p2
        sem_post(&full);            // line p3
        sem_post(&mutex);           // line p4 (NEW LINE)
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);           // line c0 (NEW LINE)
        sem_wait(&full);            // line c1
        int tmp = get();            // line c2
        sem_post(&empty);           // line c3
        sem_post(&mutex);           // line c4 (NEW LINE)
        printf("%d\n", tmp);
    }
}

int main(int argc, char *argv[]) {
    // ...
    sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
    sem_init(&full, 0, 0);    // ... and 0 are full
    sem_init(&mutex, 0, 1);   // mutex=1 because it is a lock (NEW LINE)
    // ...
}
```

Adding Mutual Exclusion (Incorrectly)

- Add some **locks around the entire put()/get()** parts of the code, as indicated by the **`NEW LINE`** comments. Seems like the right idea, but it also **doesn't work**. **Why?**
- Why does deadlock occur? What sequence of steps must happen for the program to deadlock?

Avoiding Deadlock

- 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**。

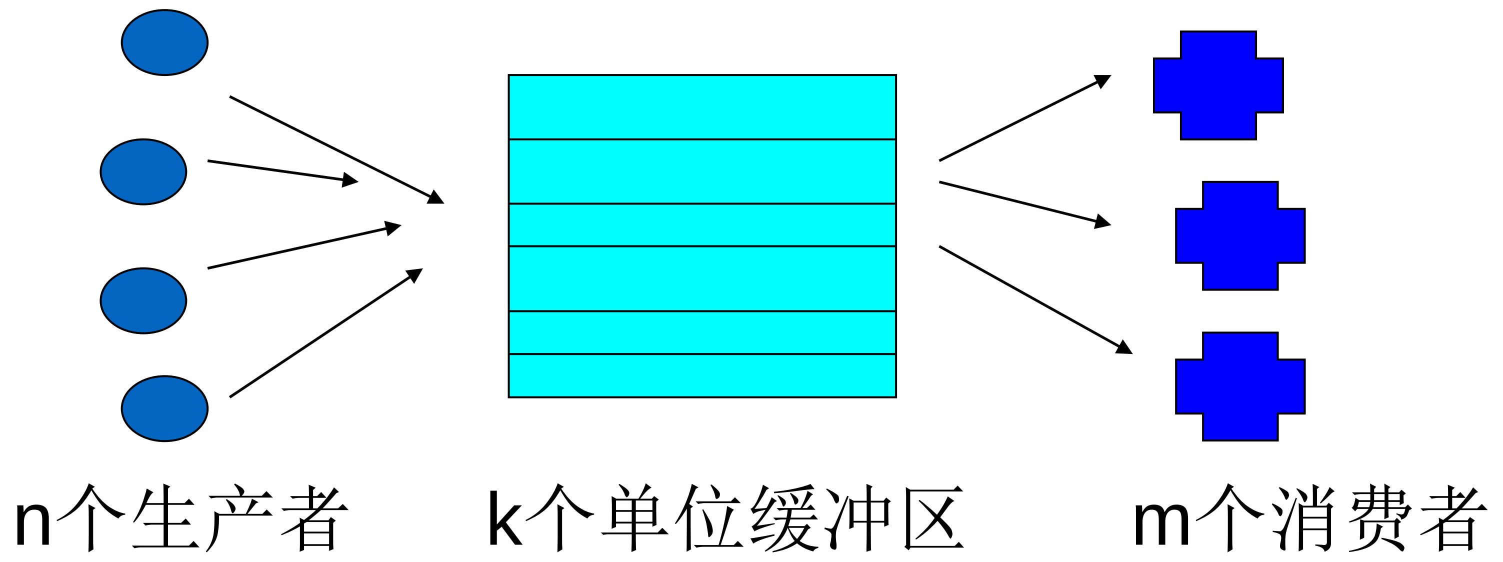
- 这里出现了一个**循环等待**。消费者持有互斥量，等待在 full 信号量上。生产者可以发送 full 信号，却在等待互斥量。因此，**生产者和消费者互相等待对方**——**典型的死锁**。

```
sem_t empty;
sem_t full;
sem_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);           // line p1
        sem_wait(&mutex);           // line p1.5 (MOVED MUTEX HERE...)
        put(i);                     // line p2
        sem_post(&mutex);           // line p2.5 (... AND HERE)
        sem_post(&full);            // line p3
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&full);            // line c1
        sem_wait(&mutex);           // line c1.5 (MOVED MUTEX HERE...)
        int tmp = get();            // line c2
        sem_post(&mutex);           // line c2.5 (... AND HERE)
        sem_post(&empty);           // line c3
        printf("%d\n", tmp);
    }
}

int main(int argc, char *argv[]) {
    // ...
    sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
    sem_init(&full, 0, 0);    // ... and 0 are full
    sem_init(&mutex, 0, 1);   // mutex=1 because it is a lock
    // ...
}
```

Adding Mutual Exclusion (Correctly)

- To solve this problem, we must **reduce the scope of the lock**. We simply move the mutex to be just around the critical section; the full and empty wait and signal code is left outside.

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.

Bounded-Buffer Problem

Shared data

semaphore full, empty, mutex;

Initially: full = 0, empty = n, mutex = 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?

Producer

```
do {
    …
  produce an item in nextp
    …
  wait(empty);
  wait(mutex);
    …
  add nextp to buffer
    …
  signal(mutex);
  signal(full);
} while (1);
```

Consumer

```
do {
  wait(full)
  wait(mutex);
    …
  remove an item from buffer to nextc
    …
  signal(mutex);
  signal(empty);
    …
  consume the item in nextc
    …
} while (1);
```

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**

死锁！

交换**signal**的顺序不会有任何影响

Producer

```
do {
    …
  produce an item in nextp
    …
  wait(mutex);
  wait(empty);
    …
  add nextp to buffer
    …
  signal(mutex);
  signal(full);
} while (1);
```

Consumer

```
do {
  wait(mutex)
  wait(full);
    …
  remove an item from buffer to nextc
    …
  signal(mutex);
  signal(empty);
    …
  consume the item in nextc
    …
} while (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：

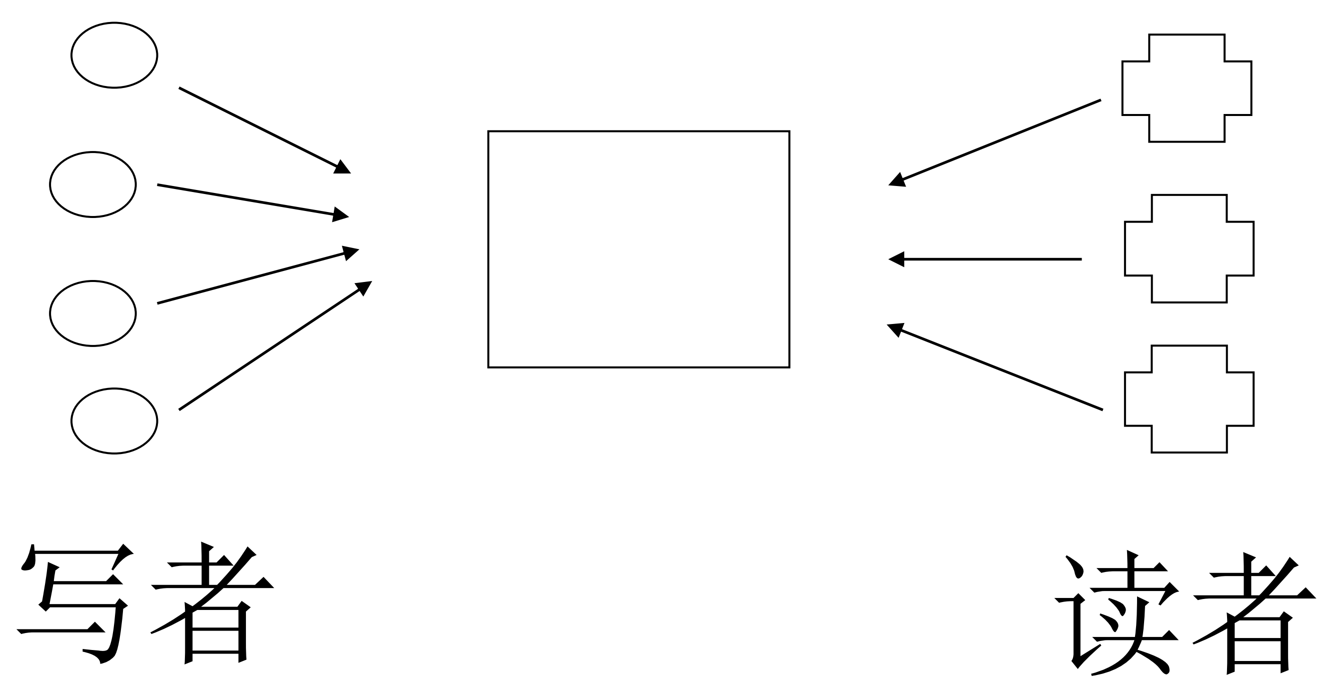
简单，而且表达能力强（用P.V操作可解决任何同步互斥问题）

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（读一读允许）；

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。

Shared data:
semaphore mutex, wrt;
Initially
mutex = 1, wrt = 1, readcount = 0

Reader:

```
wait(mutex);
readcount++;
if (readcount == 1)
    wait(wrt);
signal(mutex);
    …
  reading is performed
    …
wait(mutex);
readcount--;
if (readcount == 0)
    signal(wrt);
signal(mutex):
```

Writer:

```
wait(wrt);
    …
    writing is performed
    …
signal(wrt);
```

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。

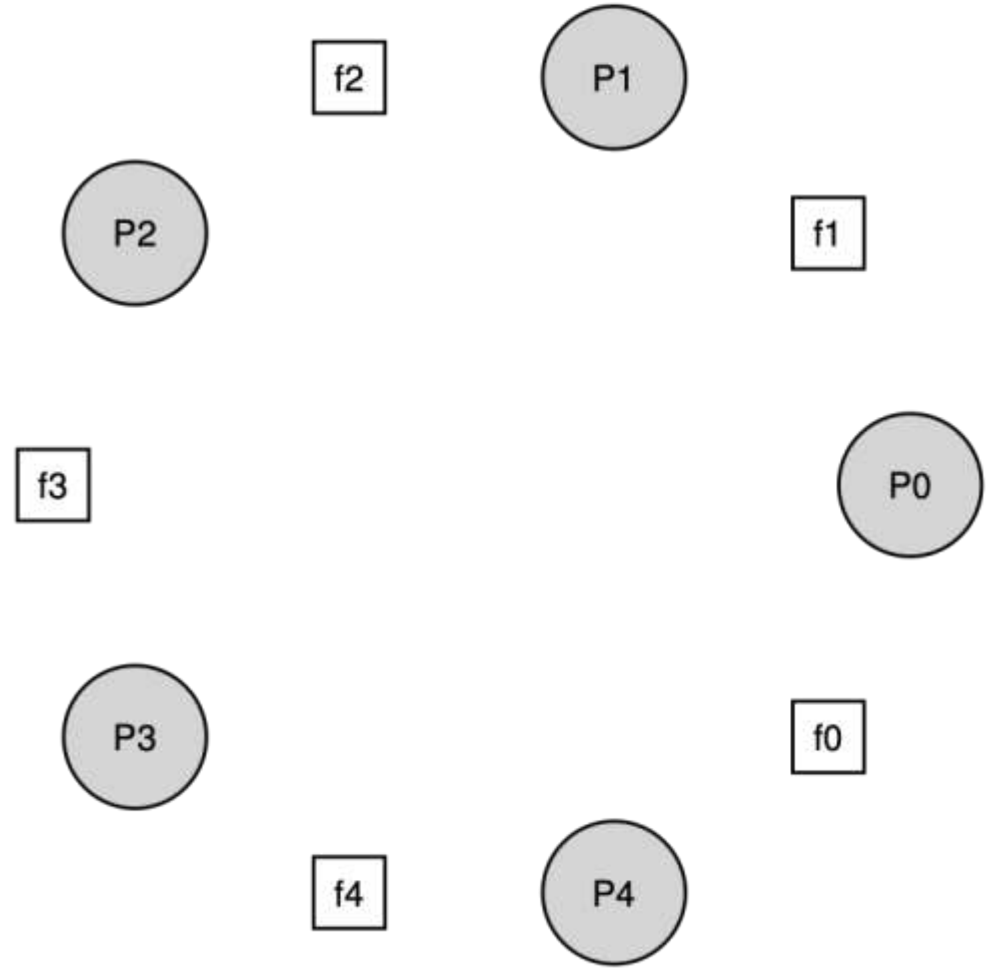
Reader-Writer Locks

读者很容易饿死写者。

```c
typedef struct _rwlock_t {
  sem_t lock;       // binary semaphore (basic lock)
  sem_t writelock;  // allow ONE writer/MANY readers
  int   readers;    // #readers in critical section
} rwlock_t;

void rwlock_init(rwlock_t *rw) {
  rw->readers = 0;
  sem_init(&rw->lock, 0, 1);
  sem_init(&rw->writelock, 0, 1);
}

void rwlock_acquire_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers++;
  if (rw->readers == 1) // first reader gets writelock
    sem_wait(&rw->writelock);
  sem_post(&rw->lock);
}

void rwlock_release_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers--;
  if (rw->readers == 0) // last reader lets it go
    sem_post(&rw->writelock);
  sem_post(&rw->lock);
}

void rwlock_acquire_writelock(rwlock_t *rw) {
  sem_wait(&rw->writelock);
}

void rwlock_release_writelock(rwlock_t *rw) {
  sem_post(&rw->writelock);
}
```

The Dining Philosophers

- One of the most famous concurrency problems posed, and solved, by **Dijkstra**.
- The problem is famous because it is fun and somewhat **intellectually interesting**; however, **its practical utility is low**.



The Dining Philosophers

- Assume there are five “philosophers” sitting around a table.
- Between each pair of philosophers is a single fork (**five total**).
- The philosophers each have times where they **think**, and don’t need any forks, and times where they **eat**.
- In order to eat, a philosopher **needs two forks**, **both** the one on their **left** and the one on their **right**.

- Here is the basic loop of each philosopher:

```
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

- We’ll use a few helper functions to get us towards a solution:

```
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].

Broken Solution

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

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

sem_t forks[5]

void getforks() {
  sem_wait(forks[left(p)]);
  sem_wait(forks[right(p)]);
}

void putforks() {
  sem_post(forks[left(p)]);
  sem_post(forks[right(p)]);
}
```

- 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
- Simple, no? Unfortunately, in this case, simple means broken. Can you see the problem that arises?

- The problem is deadlock. If each philosopher happens to **grab the fork on their left** before any philosopher can grab the fork on their right, each will be stuck holding one fork and waiting for another, forever.

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;

A Solution: Breaking The Dependency

- The simplest way to attack this problem is to **change how forks are acquired** by at least one of the philosophers; indeed, **this is how Dijkstra himself solved the problem**.

```
void getforks() {
  if (p == 4) {
    sem_wait(forks[right(p)]);
    sem_wait(forks[left(p)]);
  } else {
    sem_wait(forks[left(p)]);
    sem_wait(forks[right(p)]);
  }
}
```

- Because the last philosopher tries to grab right before left, there is no situation where each philosopher grabs one fork and is stuck waiting for another; **the cycle of waiting is broken**.

How To Implement Semaphores

```
typedef struct __Zem_t {
    int value;
    pthread_cond_t cond;
    pthread_mutex_t lock;
} Zem_t;

// only one thread can call this
void Zem_init(Zem_t *s, int value) {
    s->value = value;
    Cond_init(&s->cond);
    Mutex_init(&s->lock);
}

void Zem_wait(Zem_t *s) {
    Mutex_lock(&s->lock);
    while (s->value <= 0)
        Cond_wait(&s->cond, &s->lock);
    s->value--;
    Mutex_unlock(&s->lock);
}

void Zem_post(Zem_t *s) {
    Mutex_lock(&s->lock);
    s->value++;
    Cond_signal(&s->cond);
    Mutex_unlock(&s->lock);
}
```

Implementing Zemaphores with Locks and CVs

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

- **Curiously**, building locks and condition variables out of semaphores is **a much trickier proposition**. Some highly experienced concurrent programmers tried to do this in the Windows environment, and **many different bugs ensued** [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.

End