Synchronization

林鸿渝
徐晨熠

Contents

- Thread Intro
- **Lock**
  - Spin-based lock
    - Atomic Ops
    - Peterson
  - Sleep-based lock
  - Two-phase lock
- Lock-based Concurrent Data Structures
- Semaphores
  - Extensions of the Producer-Consumer Problem
  - Reader Writer Problem

Lock —— Spin-based: Peterson

```
int flag[2];
int turn;

void init() {
    // indicate you intend to hold the lock w/ 'flag'
    flag[0] = flag[1] = 0;
    // whose turn is it? (thread 0 or 1)
    turn = 0;
}
void lock() {
    // 'self' is the thread ID of caller
    flag[self] = 1;
    // make it other thread's turn
    turn = 1 - self;
    while ((flag[1-self] == 1) && (turn == 1 - self))
        ; // spin-wait while it's not your turn
}
void unlock() {
    // simply undo your intent
    flag[self] = 0;
}
```

requirement of
sequence of read & write

Nowadays CPU...

cache
instruction reordering

Lock —— Spin-based: one variable control

```
1  typedef struct __lock_t { int flag; } lock_t;
2
3  void init(lock_t *mutex) {
4      // 0 -> lock is available, 1 -> held
5      mutex->flag = 0;
6  }
7
8  void lock(lock_t *mutex) {
9      while (mutex->flag == 1)  // TEST the flag
10         ; // spin-wait (do nothing)
11     mutex->flag = 1;          // now SET it!
12 }
13
14 void unlock(lock_t *mutex) {
15     mutex->flag = 0;
16 }
```

Figure 28.1: **First Attempt: A Simple Flag**

Thread 1	**Thread 2**
call `lock()`	
while (flag == 1)	
interrupt: switch to Thread 2	
	call `lock()`
	while (flag == 1)
	flag = 1;
	interrupt: switch to Thread 1
flag = 1; // set flag to 1 (too!)	

Figure 28.2: **Trace: No Mutual Exclusion**

when begin with both flag == 0 at each thread

BOTH Thread1 and Thread2
enter Critical Section!!!

NOT Mutual Exclusion!

Lock —— Spin-based: TestAndSet()

```
1  int TestAndSet(int *old_ptr, int new) {
2      int old = *old_ptr; // fetch old value at old_ptr
3      *old_ptr = new;     // store 'new' into old_ptr
4      return old;         // return the old value
5  }
```

1. old_flag = 0
   new_flag = 1, cs

2. old_flag = 1
   new_flag = 1, spin

```
1  typedef struct __lock_t {
2      int flag;
3  } lock_t;
4
5  void init(lock_t *lock) {
6      // 0: lock is available, 1: lock is held
7      lock->flag = 0;
8  }
9
10 void lock(lock_t *lock) {
11     while (TestAndSet(&lock->flag, 1) == 1)
12         ; // spin-wait (do nothing)
13 }
14
15 void unlock(lock_t *lock) {
16     lock->flag = 0;
17 }
```

Figure 28.3: **A Simple Spin Lock Using Test-and-set**

Lock —— Spin-based: CompareAndSwap()

```
1  int CompareAndSwap(int *ptr, int expected, int new) {
2      int original = *ptr;
3      if (original == expected)
4          *ptr = new;
5      return original;
6  }
```

Figure 28.4: **Compare-and-swap**

```
1  void lock(lock_t *lock) {
2      while (CompareAndSwap(&lock->flag, 0, 1) == 1)
3          ; // spin
4  }
```

add variable expected

Flexible & Certain than tas

Contents

- Thread Intro
- Lock
  - Spin-based lock
    - Atomic Ops
    - Peterson
  - Sleep-based lock
  - Two-phase lock
- Lock-based Concurrent Data Structures
- **Semaphores**
  - Extensions of the Producer-Consumer Problem
  - Reader Writer Problem

Semaphores —— Extensions of the Producer-Consumer Problem

Shared object

```
1 #define N 100
2 semaphore mutex = 1;
3 semaphore avail = N;
4 semaphore fill = 0;
```

wait

```
1 void wait(semaphore *s) {
2     s->value = s->value - 1;
3     if ( s->value < 0 ) {
4         sleep();
5     }
6 }
```

Producer

```
1 void producer(void) {
2     int item;
3     while(TRUE) {
4         item = produce_item();
5         wait(&avail);
6         wait(&mutex);
7         insert_item(item);
8         post(&mutex);
9         post(&fill);
10    }
11 }
```

Consumer

```
1 void consumer(void) {
2     int item;
3     while(TRUE) {
4         wait(&fill);
5         wait(&mutex);
6         item = remove_item();
7         post(&mutex);
8         post(&avail);
9     }
10 }
```

Semaphores —— Extensions of the Producer-Consumer Problem

Consider an interesting situation: 2 Consumers and 1 Producer

Consumer

```
1 void consumer(void) {
2     int item;
3     while(TRUE) {
4         wait(&fill);
5         wait(&mutex);
6         item = remove_item();
7         post(&mutex);
8         post(&avail);
9     }
10 }
```

Null

Null

Producer

```
1 void producer(void) {
2     int item;
3     while(TRUE) {
4         item = produce_item();
5         wait(&avail);
6         wait(&mutex);
7         insert_item(item);
8         post(&mutex);
9         post(&fill);
10    }
11 }
```

Null

1

wait

```
1 void wait(semaphore *s) {
2     s->value = s->value - 1;
3     if ( s->value < 0 ) {
4         sleep();
5     }
6 }
```

Consumer

```
1 void consumer(void) {
2     int item;
3     while(TRUE) {
4         wait(&fill);
5         wait(&mutex);
6         item = remove_item();
7         post(&mutex);
8         post(&avail);
9     }
10 }
```

1

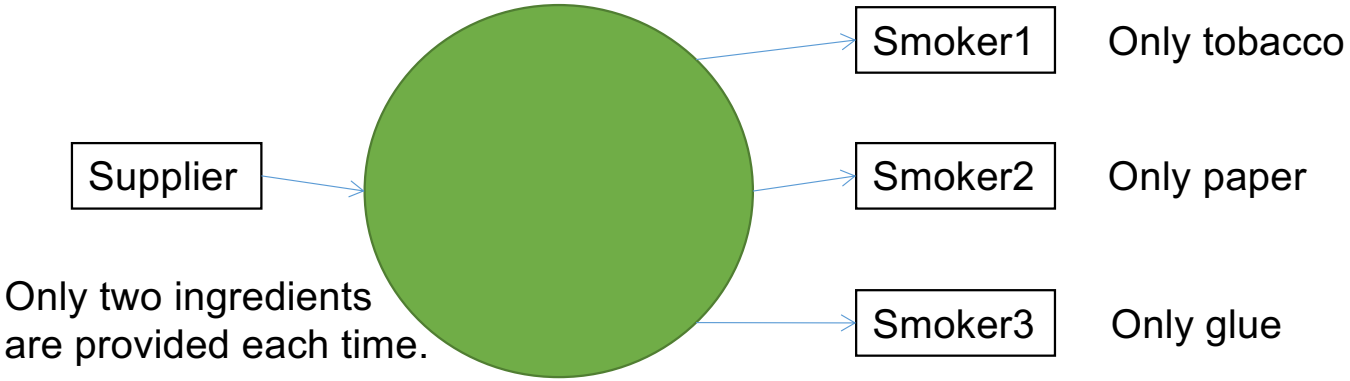
Null

Mesa semantics to Hoare semantics

Semaphores —— Extensions of the Producer-Consumer Problem

Consider an interesting problem: The Smoker's Problem

Tobacco, paper, and glue -> Cigarette

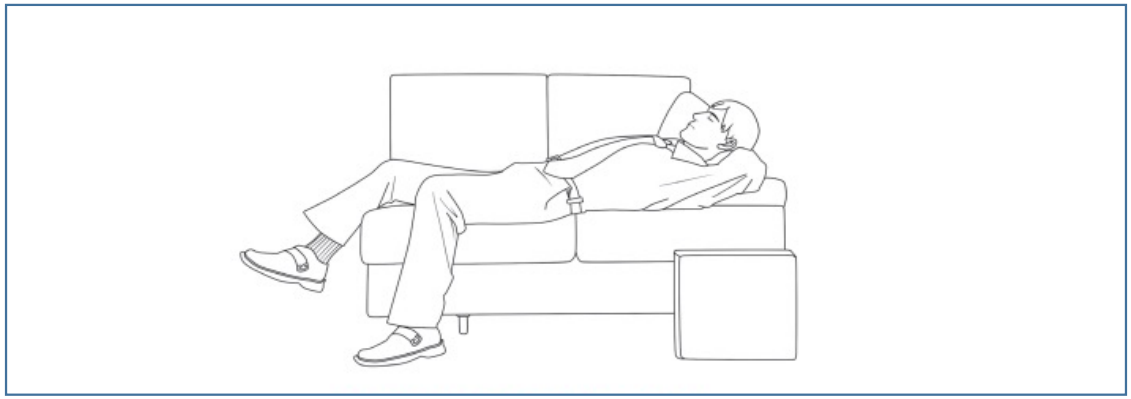


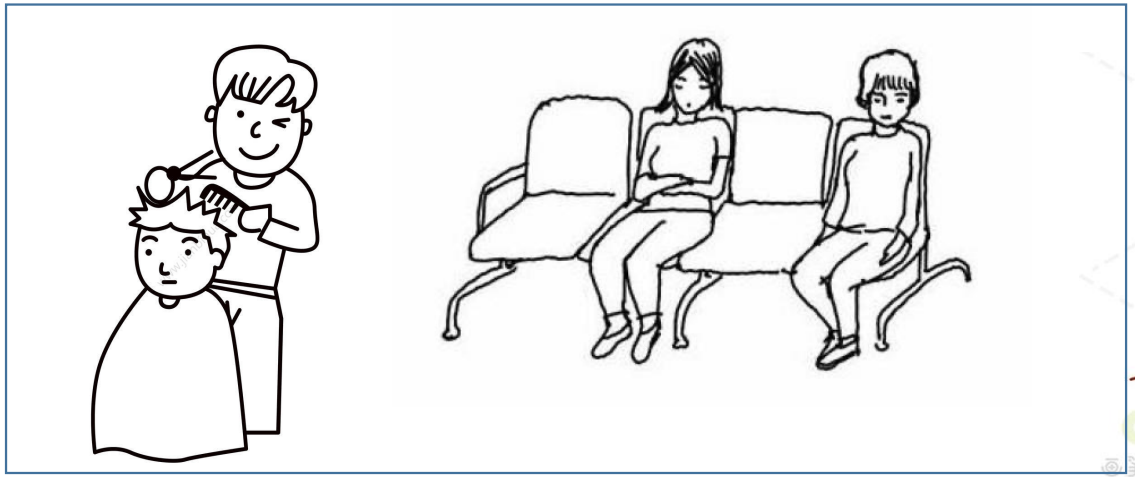
Producer

```
1  void producer(void) {
2      int item;
3      while(TRUE) {
4          item = produce_item();
5          wait(&avail);
6          wait(&mutex);
7          insert_item(item);
8          post(&mutex);
9          post(&fill);
10     }
11 }
```

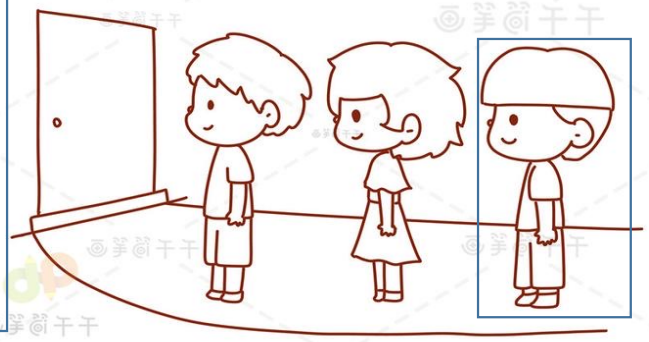
Semaphores —— Extensions of the Producer-Consumer Problem

Consider an interesting problem: The Barber Problem



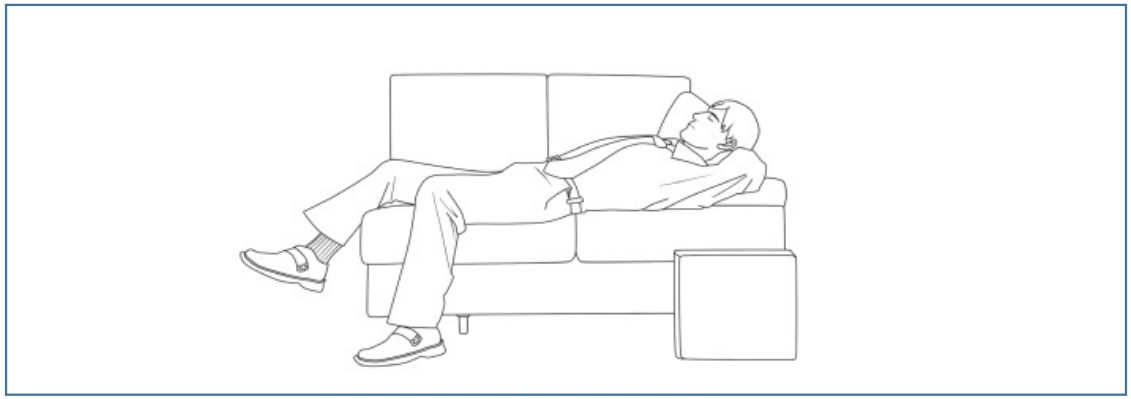


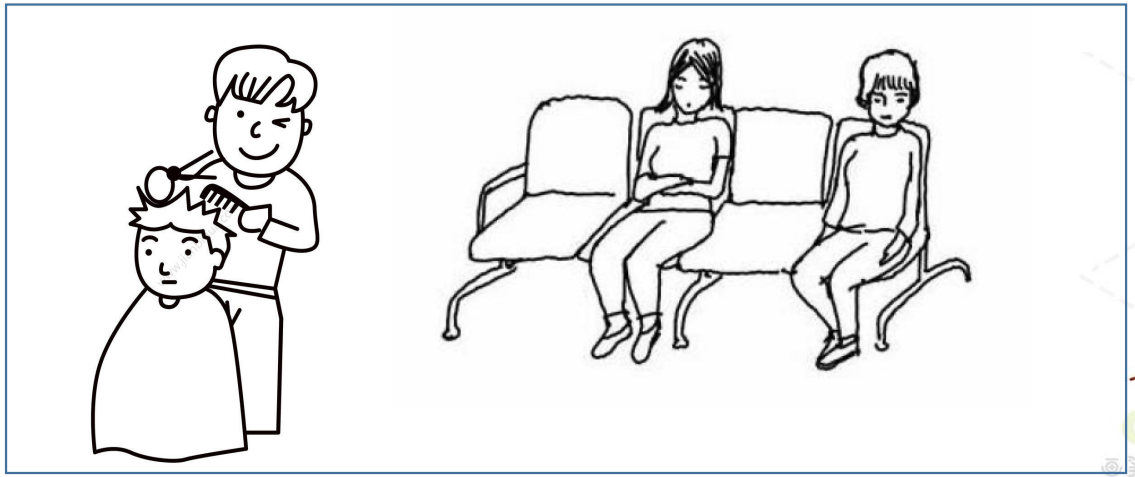
He will leave



Semaphores —— Extensions of the Producer-Consumer Problem

Consider an interesting problem: The Barber Problem





Producer

```
1 void producer(void) {
2     int item;
3     while(TRUE) {
4         item = produce_item();
5         wait(&avail);
6         wait(&mutex);
7         insert_item(item);
8         post(&mutex);
9         post(&fill);
10    }
11 }
```

Semaphores —— Reader Writer Problem

Requirement

1. When a writer is writing, readers are not allowed to read.
2. When a reader is reading, writers are not allowed to write.
3. When a writer is writing, other writers are not allowed to write.
4. When a reader is reading, other readers are allowed to read.

Maintain two things:

1. Read and write operations on the article
2. The number of readers

Semaphores —— Reader Writer Problem

Shared object

```
1 semaphore mutex = 1;
2 semaphore lock_readers = 1;
3 int readers = 0;
```

Writer

```
1 void writer(void) {
2     wait(mutex);
3     write;
4     post(mutex);
5 }
```

Reader

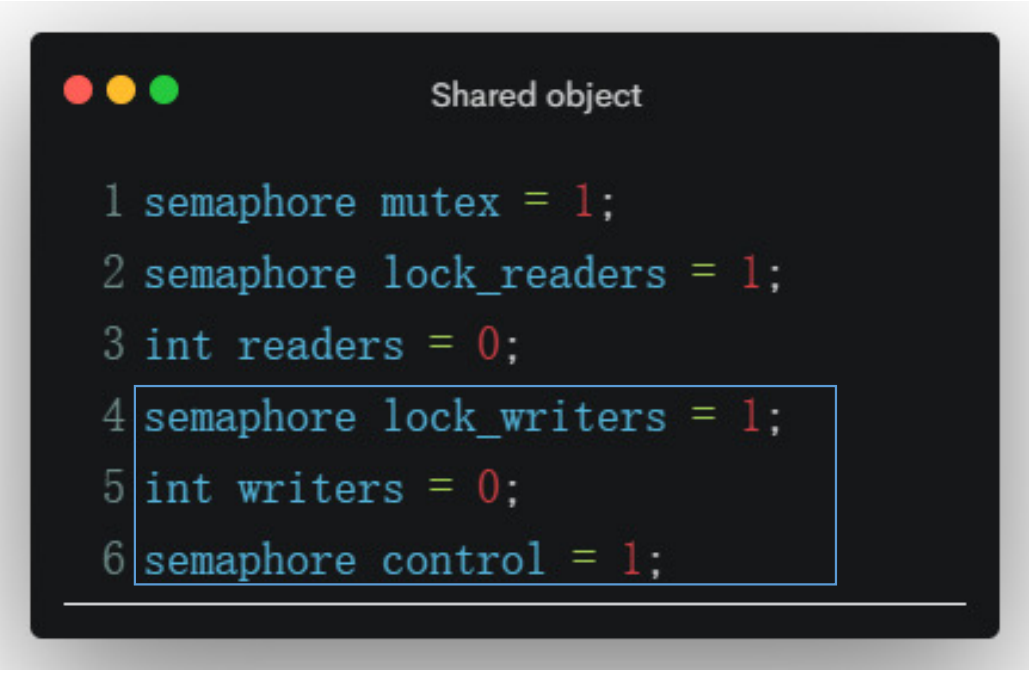
```
1 void reader(void) {
2       wait(lock_readers);
3       if(readers == 0)
4             wait(mutex);
5       readers++;
6       post(lock_readers);
7
8       read;
9
10      wait(lock_readers);
11      readers--;
12      if(readers == 0)
13            post(mutex);
14      post(lock_readers);
15 }
```

Semaphores —— Reader Writer Problem

Thus, the issue is resolved.

However, in reality, a new problem arises:
If there is a continuous stream of readers,
writers may face starvation.

New Requirement
5. If a writer requests to write,
new readers are not permitted to read.



Semaphores —— Reader Writer Problem

Reader

```
1 void reader(void) {
2       wait(control);
3       wait(lock_readers);
4       if(readers == 0)
5             wait(mutex);
6       readers++;
7       post(lock_readers);
8       post(control);
9
10      read;
11
12      wait(lock_readers);
13      readers--;
14      if(readers == 0)
15            post(mutex);
16      post(lock_readers);
17 }
```

Writer

```
1 void writer(void) {
2     wait(lock_writers)
3     writers++;
4     if(writers == 1)
5           wait(control);
6     post(lock_writers)
7     wait(mutex);
8     write;
9     post(mutex);
10    wait(lock_writers)
11    writers--;
12    if(writers == 0)
13          post(control);
14    post(lock_writers)
15 }
```

Thanks