哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

操作系统

Operating Systems

刘川意　教授

liuchuanyi@hit.edu.cn

哈尔滨工业大学（深圳）

2021年10月

Module 5: Concurrency & Synchronization
并发与同步

1. **Concurrency Introduction**
2. **Locks**
3. **基于Lock的并发数据结构**
4. Condition Variables 条件变量
5. Semaphore 信号量
6. 常见并发问题
7. **基于事件的并发**

Concurrency为什么放到OS中讲？

- **History！**
  - OS Kernel是第一个并发程序，如:`write()`的设计，中断对shared structures的影响（page tables, process lists, file system structures, and virtually every kernel data structure has to be carefully accessed）
  - 很多并发处理技术是在OS中发明和实现的
  - multi-threade进程中，应用程序也需要考虑并发

并发相关的重要术语

- **Critical Section,** a piece of code that accesses a *shared* resource, usually a variable or data structure
- **Race Condition** arises if multiple threads of execution enter the critical section at roughly the same time; both attempt to update the shared data structure, leading to a surprising (and perhaps undesirable) outcome.
- **Indeterminate,** program consists of one or more race conditions; the output of the program varies from run to run, depending on which threads ran when. The outcome is thus not **deterministic**, something we usually expect from computer systems.
- **Mutual Exclusion** primitives, guarantee that only a single thread ever enters a critical section, thus avoiding races, and resulting in deterministic program outputs

4

Review: Thread

- 轻量化执行环境，new abstraction for a single running process

- Multi-threaded 程序的特点：
  - A multi-threaded program has more than one point of execution.
  - Multiple PCs (Program Counter)
  - They share the same address space.

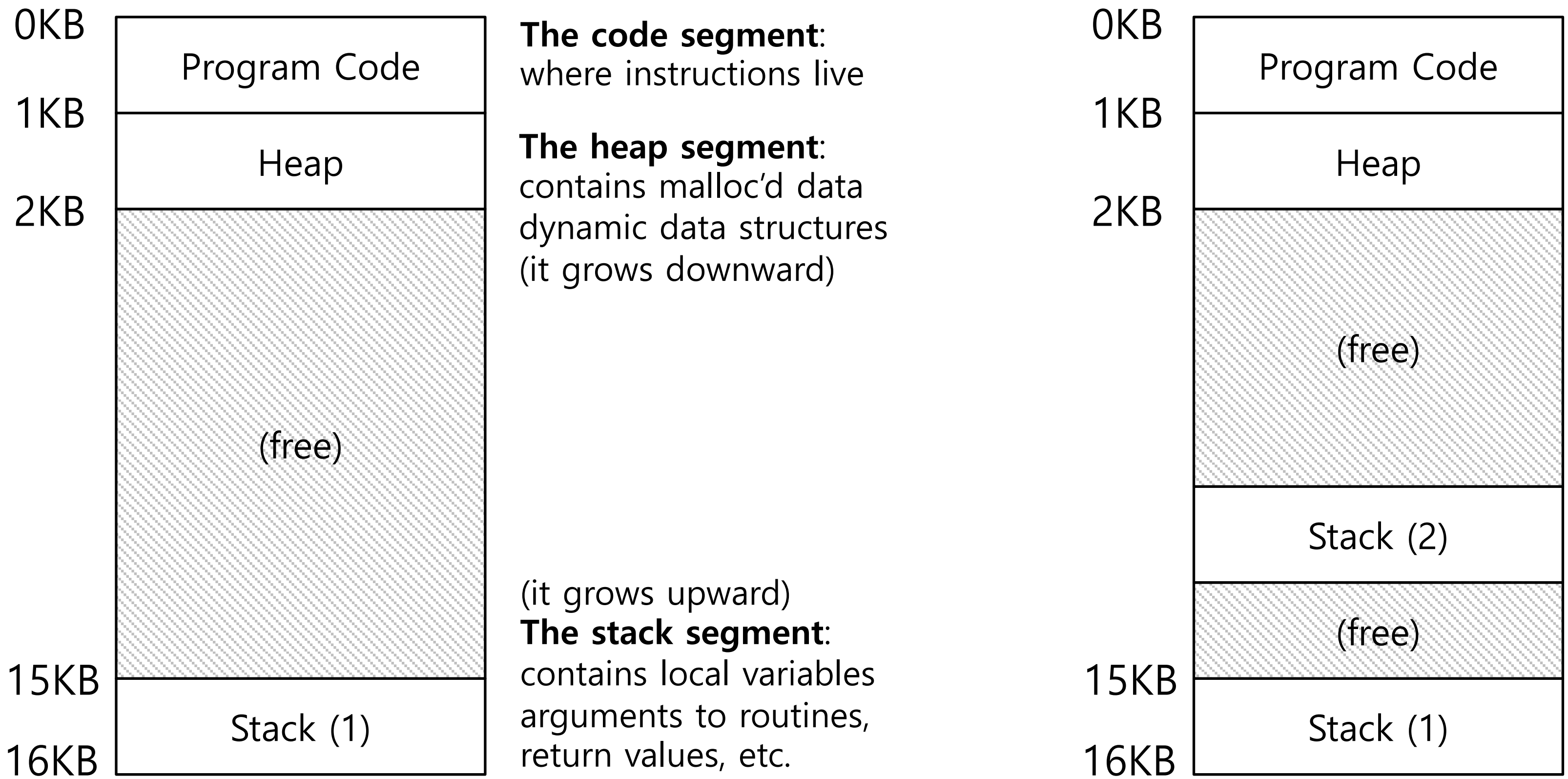
5

Context switch between threads

- Each thread has its own program counter and set of registers.
  - One or more **thread control blocks(TCBs)** are needed to store the state of each thread.
- When switching from running one (T1) to running the other (T2),
  - The register state of T1 be saved.
  - The register state of T2 restored.
  - The address space remains the same.

The stack of the relevant thread

- There will be one stack per thread.



A Single-Threaded Address Space

Two threaded Address Space

badcnt.c: Improper Synchronization

```c
/* Global shared variable */
volatile long cnt = 0; /* Counter */

int main(int argc, char **argv)
{
    long niters;
    pthread_t tid1, tid2;

    niters = atoi(argv[1]);
    Pthread_create(&tid1, NULL,
        thread, &niters);
    Pthread_create(&tid2, NULL,
        thread, &niters);
    Pthread_join(tid1, NULL);
    Pthread_join(tid2, NULL);

    /* Check result */
    if (cnt != (2 * niters))
        printf("BOOM! cnt=%ld\n", cnt);
    else
        printf("OK cnt=%ld\n", cnt);
    exit(0);
}
```

badcnt.c

```c
/* Thread routine */
void *thread(void *vargp)
{
    long i, niters =
                *((long *)vargp);

    for (i = 0; i < niters; i++)
        cnt++;

    return NULL;
}
```

```
[zs_cao@localhost conc]$ ./badcnt 10000
OK cnt=20000
[zs_cao@localhost conc]$ ./badcnt 10000
BOOM! cnt=17302
[zs_cao@localhost conc]$ ./badcnt 10000
OK cnt=20000
```

线程并发执行的问题

Assembly Code for Counter Loop

- 编译:
  - gcc –s badcnt.c –o badcnt.s
  - vim badcnt.s

```
for (i = 0; i < niters; i++)
    cnt++;
```

```
        movq    %rdi, -24(%rbp)
        movq    -24(%rbp), %rax
        movq    (%rax), %rax
        movq    %rax, -8(%rbp)
        movq    $0, -16(%rbp)
        jmp     .L6
.L7:
        movq    cnt(%rip), %rax
        addq    $1, %rax
        movq    %rax, cnt(%rip)
        addq    $1, -16(%rbp)
.L6:
        movq    -16(%rbp), %rax
        cmpq    -8(%rbp), %rax
        jl      .L7
        movl    $0, %eax
        popq    %rbp
```

H_i : Head

L_i : Load `cnt`
U_i : Update `cnt`
S_i : Store `cnt`

T_i : Tail

Assembly Code for Counter Loop

- 汇编:
  - gcc –c badcnt.s –o badcnt.o
  - objdump -dx badcnt.o

```
for (i = 0; i < niters; i++)
    cnt++;
```

```
00000000000000ed <thread>:
  ed:   55                      push   %rbp
  ee:   48 89 e5                mov    %rsp,%rbp
  f1:   48 89 7d e8             mov    %rdi,-0x18(%rbp)
  f5:   48 8b 45 e8             mov    -0x18(%rbp),%rax
  f9:   48 8b 00                mov    (%rax),%rax
  fc:   48 89 45 f8             mov    %rax,-0x8(%rbp)
 100:   48 c7 45 f0 00 00 00    movq   $0x0,-0x10(%rbp)
 107:   00
 108:   eb 17                   jmp    121 <thread+0x34>
 10a:   48 8b 05 00 00 00 00    mov    0x0(%rip),%rax        # 111
<thread+0x24>
                        10d: R_X86_64_PC32      cnt-0x4
 111:   48 83 c0 01             add    $0x1,%rax
 115:   48 89 05 00 00 00 00    mov    %rax,0x0(%rip)        # 11c
<thread+0x2f>
                        118: R_X86_64_PC32      cnt-0x4
 11c:   48 83 45 f0 01          addq   $0x1,-0x10(%rbp)
 121:   48 8b 45 f0             mov    -0x10(%rbp),%rax
 125:   48 3b 45 f8             cmp    -0x8(%rbp),%rax
 129:   7c df                   jl     10a <thread+0x1d>
 12b:   b8 00 00 00 00          mov    $0x0,%eax
 130:   5d                      pop    %rbp
 131:   c3                      retq
```

H_i L_i U_i S_i T_i

Assembly Code for Counter Loop

- 链接:
  - gcc –o badcnt.c –o badcnt -lpthread
  - objdump -d badcnt

```
for (i = 0; i < niters; i++)
    cnt++;
```

怎么计算？

```
0000000000000957 <thread>:
 957:    55                      push    %rbp
 958:    48 89 e5                mov     %rsp,%rbp
 95b:    48 89 7d e8             mov     %rdi,-0x18(%rbp)
 95f:    48 8b 45 e8             mov     -0x18(%rbp),%rax
 963:    48 8b 00                mov     (%rax),%rax
 966:    48 89 45 f8             mov     %rax,-0x8(%rbp)
 96a:    48 c7 45 f0 00 00 00    movq    $0x0,-0x10(%rbp)
 971:    00
 972:    eb 17                   jmp     98b <thread+0x34>
 974:    48 8b 05 b5 06 20 00    mov     0x2006b5(%rip),%rax
  # 201030 <cnt>
 97b:    48 83 c0 01             add     $0x1,%rax
 97f:    48 89 05 aa 06 20 00    mov     %rax,0x2006aa(%rip)
  # 201030 <cnt>
 986:    48 83 45 f0 01          addq    $0x1,-0x10(%rbp)
 98b:    48 8b 45 f0             mov     -0x10(%rbp),%rax
 98f:    48 3b 45 f8             cmp     -0x8(%rbp),%rax
 993:    7c df                   jl      974 <thread+0x1d>
 995:    b8 00 00 00 00          mov     $0x0,%eax
 99a:    5d                      pop     %rbp
 99b:    c3                      retq
```

H_i (957–972)

L_i, U_i, S_i (974–97f)

T_i (986–99b)

Race condition

- 把上述示例简化一下:
  - counter = counter + 1 (default is 50)
  - We expect the result is 52. However,

OS	Thread1	Thread2	PC	%eax	counter
			(after instruction)		
	before critical section		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt					
save T2's state					
restore T1's state			108	51	50
	mov %eax, 0x8049a1c		113	51	**51**

Critical section

- A piece of code that accesses a shared variable and must not be concurrently executed by more than one thread.
  - Multiple threads executing critical section can result in a race condition.
  - Need to support **atomicity** for critical sections (**mutual exclusion**)

Locks

- Ensure that any such critical section executes as if it were a single atomic instruction (**execute a series of instructions atomically**).

```
lock_t mutex;
. . .
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

Critical section

14

Concurrent Execution(并发执行)

- *Key idea:* In general, any sequentially consistent interleaving is possible, but some give an unexpected result!
  - I_i denotes that thread i executes instruction I
  - $\%rdx_i$ is the content of %rdx in thread i's context

i (thread)	$instr_i$	$\%rdx_1$	$\%rdx_2$	cnt
1	H_1	-	-	0
1	L_1	0	-	0
1	U_1	1	-	0
1	S_1	1	-	1
2	H_2	-	-	1
2	L_2	-	1	1
2	U_2	-	2	1
2	S_2	-	2	2
2	T_2	-	2	2
1	T_1	1	-	2

OK

Thread 1 critical section

Thread 2 critical section

Concurrent Execution (cont)

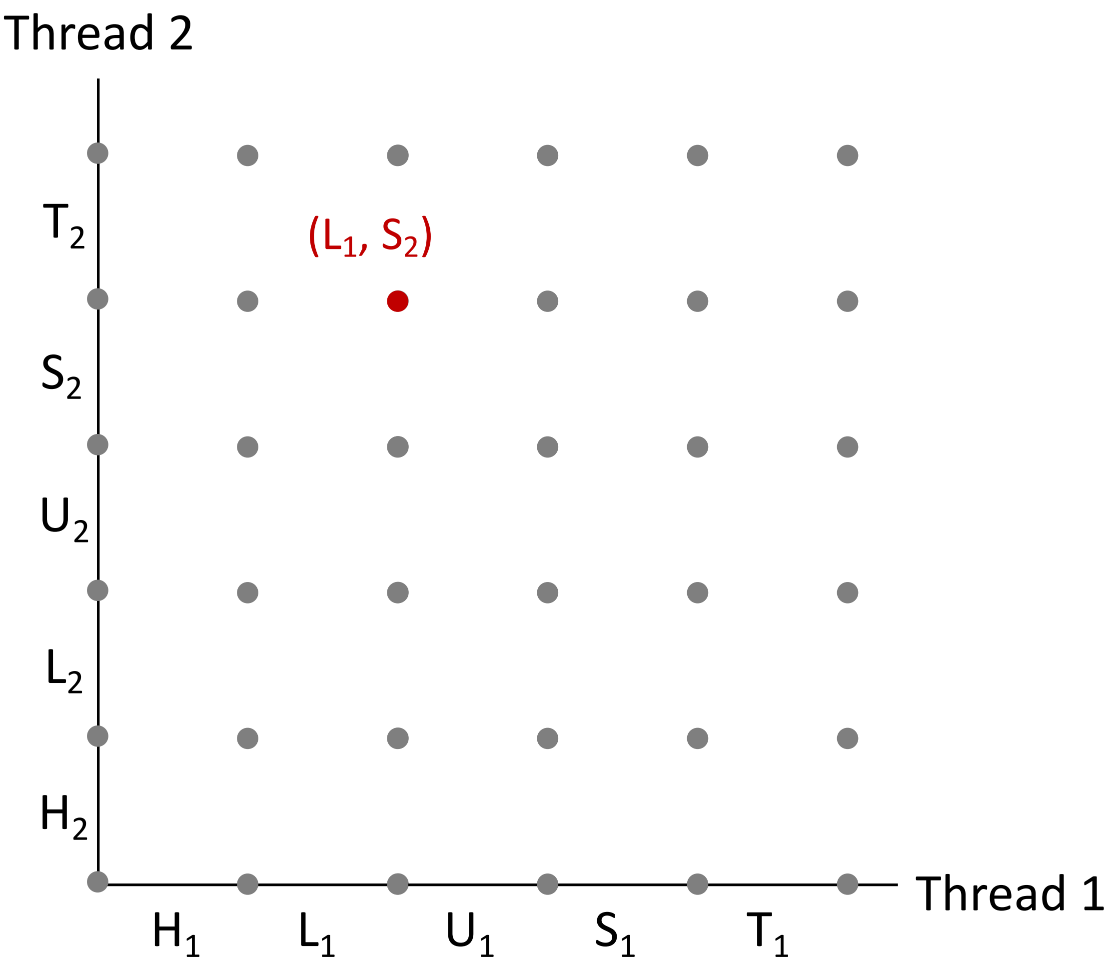
- Incorrect ordering: two threads increment the counter, but the result is 1 instead of 2

i (thread)	$instr_i$	$\%rdx_1$	$\%rdx_2$	cnt
1	H_1	-	-	0
1	L_1	0	-	0
1	U_1	1	-	0
2	H_2			0
2	L_2	-	0	0
1	S_1	1	-	1
1	T_1	1	-	1
2	U_2	-	1	1
2	S_2	-	1	1
2	T_2	-	1	1

S1应该在L2之前执行

Oops!

Progress Graphs（进度图）



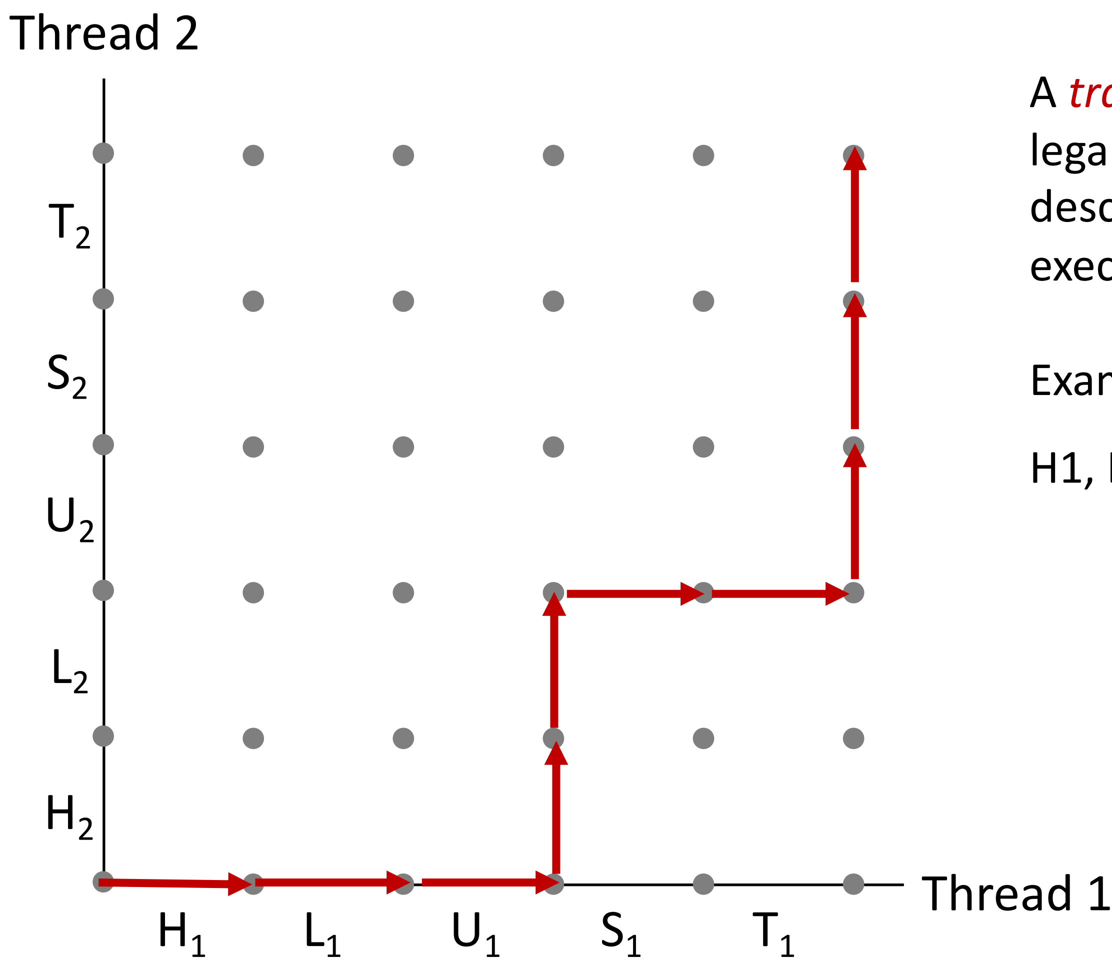
A *progress graph* depicts the discrete *execution state space* of concurrent threads.

Each axis corresponds to the sequential order of instructions in a thread.

Each point corresponds to a possible *execution state* ($Inst_1$, $Inst_2$).

E.g., (L_1, S_2) denotes state where thread 1 has completed L_1 and thread 2 has completed S_2.

Trajectories in Progress Graphs

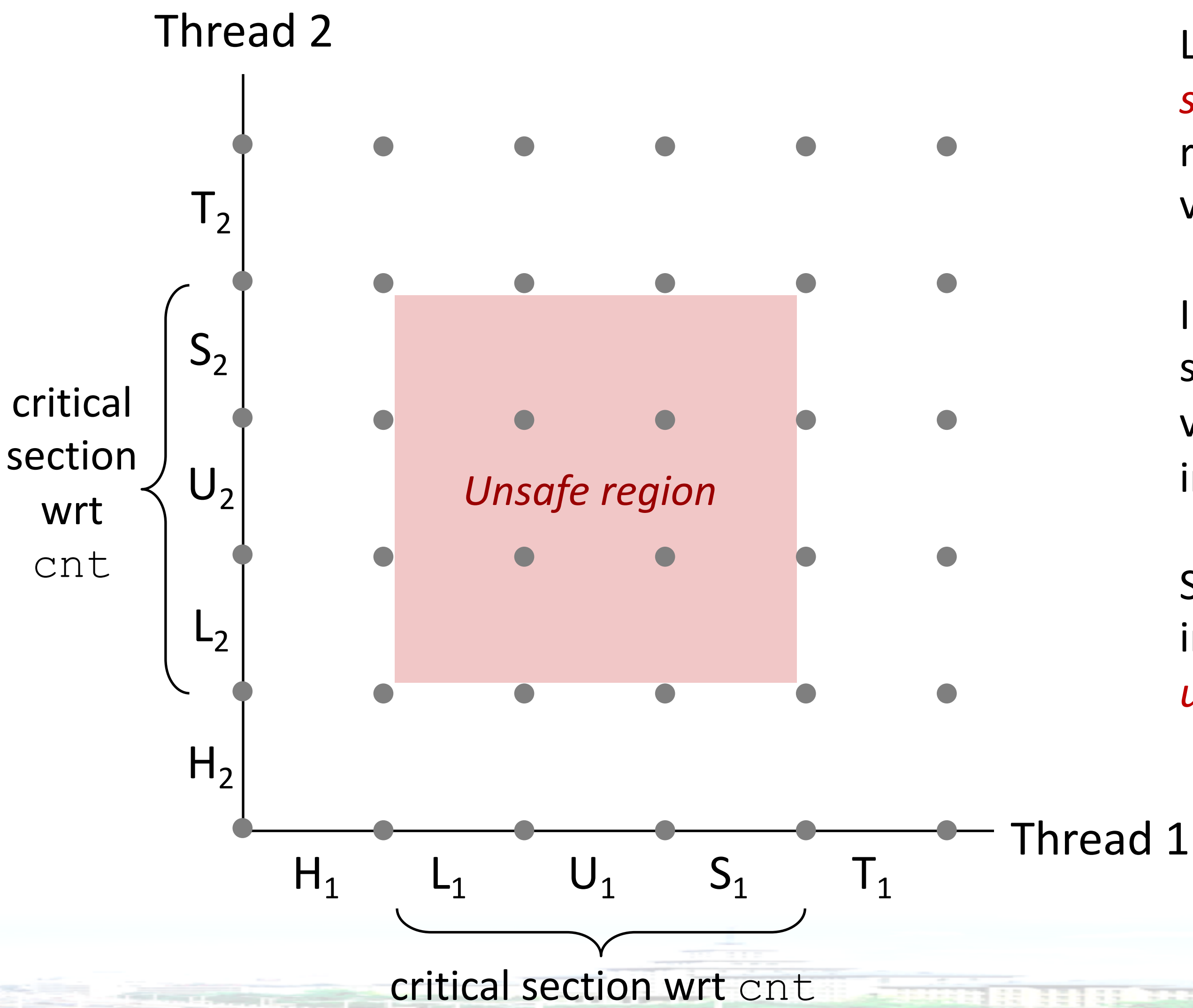


A *trajectory（轨道）* is a sequence of legal state transitions（转换） that describes one possible concurrent execution of the threads.

Example:

H1, L1, U1, H2, L2, S1, T1, U2, S2, T2

Critical Sections and Unsafe Regions

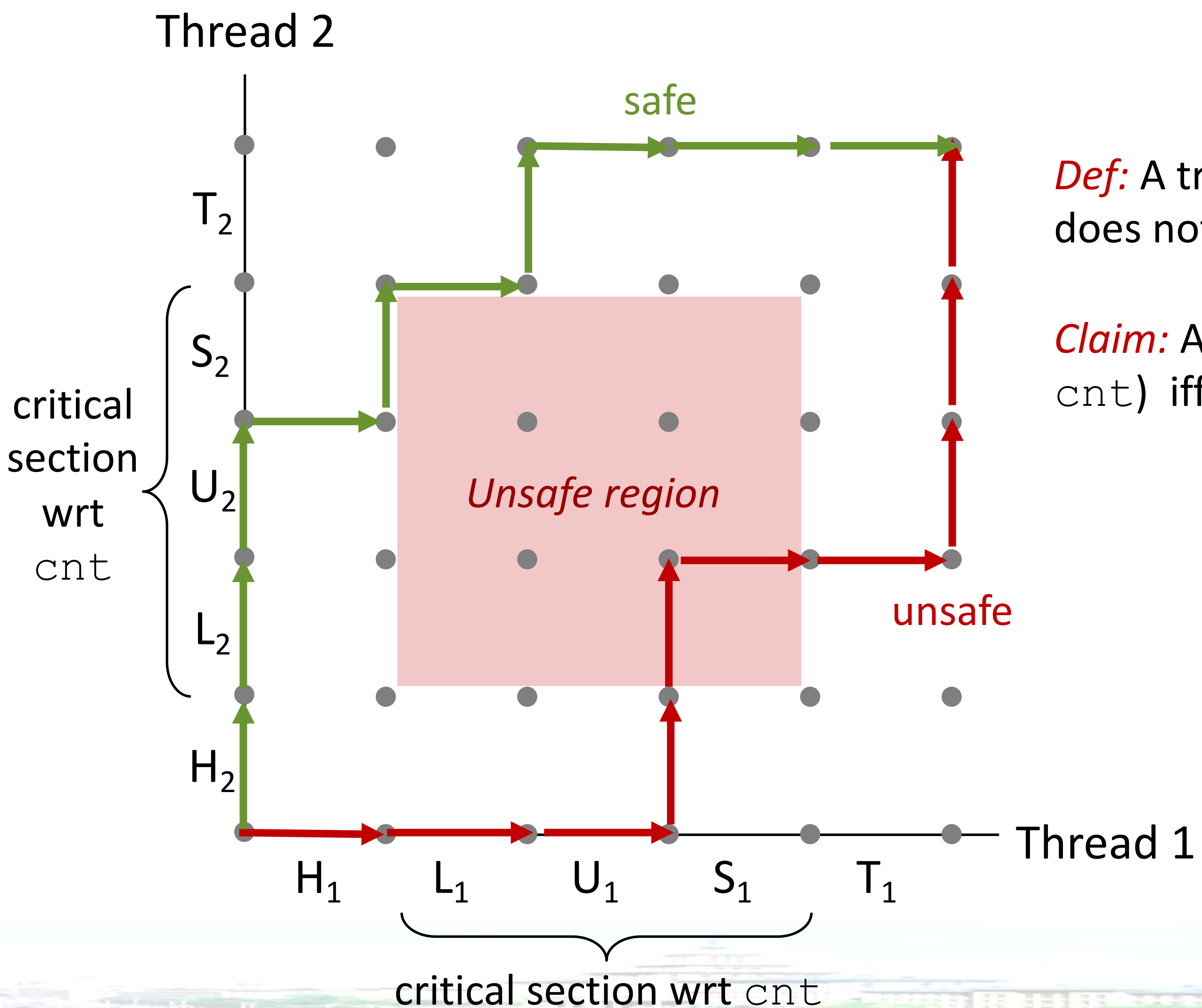


L, U, and S form a *critical section（临界区）*with respect to the shared variable `cnt`

Instructions in critical sections (wrt some shared variable) should not be interleaved

Sets of states where such interleaving occurs form *unsafe regions*

Critical Sections and Unsafe Regions



Def: A trajectory（轨道）is *safe* iff it does not enter any unsafe region

Claim: A trajectory is correct (wrt `cnt`) iff it is safe

Enforcing Mutual Exclusion

- *Question:* How can we guarantee a safe trajectory?

- Answer: We must ***synchronize***（*同步*）the execution of the threads so that they can never have an unsafe trajectory.
  - i.e., need to guarantee ***mutually exclusive access（互斥地访问）*** for each critical section.

- Classic solution:
  - Semaphores（信号量）(Edsger Dijkstra)

- Other approaches (out of our scope)
  - Mutex and condition variables (Pthreads)
  - Monitors (Java)

信号处理

【例题1】子进程运行结束会向父进程发送______信号。

【例题2】异步信号安全的函数要么是可重入的，要么不能被信号处理程序中断，包括I/O函数（）

A.printf　　B.sprintf　　C.write　　D.malloc

答案：**SIGCHLD/17号**　　**C**

1. **Concurrency Introduction**
2. **Locks**
3. **基于Lock的并发数据结构**
4. Condition Variables 条件变量
5. Semaphore 信号量
6. 常见并发问题
7. **基于事件的并发**

Locks: The Basic Idea

- Ensure that any **critical section** executes as if it were a single atomic instruction. “全部或都不”
  - Eg. update of a shared variable

```
balance = balance + 1;
```

  - Add some code around the critical section

```
lock_t mutex; // some globally-allocated lock 'mutex'
…
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```


34

Lock变量

- Lock variable holds the state of the lock.
  - **available** (or **unlocked** or **free**)
    - No thread holds the lock.
  - **acquired** (or **locked** or **held**)
    - Exactly one thread holds the lock and presumably is in a critical section.

35

lock()原语的语义（semantics）

- `lock()`
  - **Try to** acquire the lock.
  - If no other thread holds the lock, the thread will **acquire** the lock.
  - **Enter** the *critical section*.
    - This thread is said to be the owner of the lock.

  - Other threads are *prevented from* entering the critical section while the first thread that holds the lock is in there.

Pthread Locks - mutex

- The name that the POSIX library uses for a lock.
  - Used to provide mutual exclusion between threads.

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

Pthread_mutex_lock(&lock); // wrapper for pthread_mutex_lock()
balance = balance + 1;
Pthread_mutex_unlock(&lock);
```

  - We may be using *different locks* to protect *different variables* → Increase concurrency (a more **fine-grained** approach).

37

Lock如何实现？

- Efficient locks provided mutual exclusion at low cost.
- Building a lock need some help from the **hardware** and the **OS**.

38

如何评价lock原语？

- **Mutual exclusion 正确性**
  - Does the lock work, preventing multiple threads from entering *a critical section*?
- **Fairness 公平性**
  - Does each thread contending for the lock get a fair shot at acquiring it once it is free? (Starvation)
- **Performance 性能**
  - The time overheads added by using the lock

Controlling Interrupts 基于中断控制的锁实现

- **Disable Interrupts** for critical sections
  - One of the earliest solutions used to provide mutual exclusion
  - Invented for single-processor systems.

```
void lock() {
    DisableInterrupts();
}
void unlock() {
    EnableInterrupts();
}
```

  - Problem:
    - Require too much *trust* in applications
      - Greedy (or malicious) program could monopolize the processor.
    - Do not work on multiprocessors 多处理器体系结构这种方式不work
    - Code that masks or unmasks interrupts be executed *slowly* by modern CPUs

40

Why hardware support needed?

- **First attempt**: Using a *flag* denoting whether the lock is held or not.
  - The code below has problems.

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    // 0 → lock is available, 1 → held
    mutex->flag = 0;
}

void lock(lock_t *mutex) {
    while (mutex->flag == 1)  // TEST the flag
            ;  // spin-wait (do nothing)
    mutex->flag = 1;  // now SET it !
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```


41

Why hardware support needed? (Cont.)

- **Problem 1**: No Mutual Exclusion (assume `flag=0` to begin)

Thread1	Thread2
call `lock()` `while (flag == 1)` interrupt: switch to Thread 2	

	call `lock()`
	`while (flag == 1)`
	`flag = 1;`
	interrupt: switch to Thread 1

- **Problem 2**: Spin-waiting wastes time waiting for another thread.

```
flag = 1; // set flag to 1 (too!)
```

- So, we need an atomic instruction supported by Hardware!
  - ***test-and-set* instruction**, also known as ***atomic exchange***

42

基于Test-and-set硬件指令实现

- An instruction to support the creation of simple locks

```
int TestAndSet(int *ptr, int new) {
    int old = *ptr;         // fetch old value at ptr
    *ptr = new;     // store 'new' into ptr
    return old;     // return the old value
}
```

- **return**(testing) old value pointed to by the `ptr`.
- *Simultaneously* **update**(setting) said value to `new`.
- This sequence of operations is performed atomically.

43

A Simple Spin Lock using test-and-set

```
typedef struct __lock_t {
    int flag;
} lock_t;

void init(lock_t *lock) {
    // 0 indicates that lock is available,
    // 1 that it is held
    lock->flag = 0;
}

void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
        ;      // spin-wait
}

void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

- **Note**: To work correctly on *a single processor*, it requires a preemptive scheduler.
- 在单处理器体系结构中，需要OS kernel实现抢占式调度策略来支持

44

Evaluating Spin Locks

- **Correctness**: yes
  - The spin lock only allows a single thread to entry the critical section.

- **Fairness**: no
  - Spin locks <u>don't provide any fairness</u> guarantees.
  - Indeed, a thread spinning may spin *forever*.

- **Performance**:
  - In the single CPU, performance overheads can be quire *painful*.
  - If the number of threads roughly equals the number of CPUs, spin locks work *reasonably well*.

45

基于Compare-And-Swap硬件指令实现

- Test whether the value at the address(`ptr`) is equal to `expected`.
  - *If so*, update the memory location pointed to by `ptr` with the `new` value.
  - *In either case*, return the actual value at that memory location.

```c
int CompareAndSwap(int *ptr, int expected, int new) {
    int actual = *ptr;
    if (actual == expected)
        *ptr = new;
    return actual;
}
```

Compare-and-Swap hardware atomic instruction (C-style)

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

Spin lock with compare-and-swap

46

Compare-And-Swap (Cont.)

- C-callable x86-version of compare-and-swap

```
char CompareAndSwap(int *ptr, int old, int new) {
    unsigned char ret;

    // Note that sete sets a 'byte' not the word
    __asm__ __volatile__ (
            " lock\n"
            " cmpxchgl %2,%1\n"
            " sete %0\n"
            : "=q" (ret), "=m" (*ptr)
            : "r" (new), "m" (*ptr), "a" (old)
            : "memory");
    return ret;
}
```


47

Load-Linked and Store-Conditional

```
int LoadLinked(int *ptr) {
    return *ptr;
}

int StoreConditional(int *ptr, int value) {
  if (no one has updated *ptr since the LoadLinked to this address) {
    *ptr = value;
    return 1; // success!
  } else {
    return 0; // failed to update
  }
}
```

Load-linked And Store-conditional

- The store-conditional *only succeeds* if no intermittent store to the address has taken place.
  - **success**: return 1 and <u>update</u> the value at `ptr` to `value`.
  - **fail**: the value at `ptr` is <u>not updates</u> and 0 is returned.

48

Load-Linked and Store-Conditional (Cont.)

```
void lock(lock_t *lock) {
    while (1) {
        while (LoadLinked(&lock->flag) == 1)
            ; // spin until it's zero
        if (StoreConditional(&lock->flag, 1) == 1)
            return; // if set-it-to-1 was a success: all done
                    otherwise: try it all over again
    }
}

void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

Using LL/SC To Build A Lock

```
void lock(lock_t *lock) {
    while (LoadLinked(&lock->flag)||!StoreConditional(&lock->flag, 1))
        ; // spin
}
```

A more concise form of the `lock()` using LL/SC

49

Fetch-And-Add

- Atomically increment a value while returning the old value at a particular address.

```c
int FetchAndAdd(int *ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}
```

Fetch-And-Add Hardware atomic instruction (C-style)

50

Ticket Lock

- **Ticket lock** can be built with fetch-and add.
  - Ensure progress for all threads. → fairness

```
typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn = 0;
}

void lock(lock_t *lock) {
    int myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn != myturn)
        ; // spin
}
void unlock(lock_t *lock) {
    FetchAndAdd(&lock->turn);
}
```


51

So Much Spinning

- Hardware-based spin locks are simple and they work.

- In some cases, these solutions can be quite inefficient.
  - Any time a thread gets caught *spinning*, it **wastes an entire time slice** doing nothing but checking a value.

How To Avoid *Spinning*?
We'll need OS Support too!

如何解决“自旋空转”，办法1：Just Yield

- When you are going to spin, give up the CPU to another thread.
  - OS system call moves the caller from the *running state* to the *ready state*.
  - The cost of a **context switch** can be substantial and the **starvation** problem still exists.

```
void init() {
    flag = 0;
}

void lock() {
    while (TestAndSet(&flag, 1) == 1)
        yield(); // give up the CPU
}

void unlock() {
    flag = 0;
}
```

Lock with Test-and-set and Yield

53

办法2：Using Queues: Sleeping, not Spinning

- **Queue** to keep track of which threads are waiting to enter the lock.
- `park()`
  - Put a calling thread to sleep
- `unpark(threadID)`
  - Wake a particular thread as designated by `threadID`.

Using Queues: Sleeping Instead of Spinning

```c
typedef struct __lock_t { int flag; int guard; queue_t *q; } lock_t;

void lock_init(lock_t *m) {
    m->flag = 0;
    m->guard = 0;
    queue_init(m->q);
}

void lock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; // acquire guard lock by spinning
    if (m->flag == 0) {
        m->flag = 1; // lock is acquired
        m->guard = 0;
    } else {
        queue_add(m->q, gettid());
        m->guard = 0;
        park();
    }
}
…
```

Lock With Queues, Test-and-set, Yield, And Wakeup

55

Using Queues: Sleeping Instead of Spinning

```c
void unlock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; // acquire guard lock by spinning
    if (queue_empty(m->q))
        m->flag = 0; // let go of lock; no one wants it
    else
        unpark(queue_remove(m->q)); // hold lock (for next thread!)
    m->guard = 0;
}
```

Lock With Queues, Test-and-set, Yield, And Wakeup (Cont.)

56

Futex (Cont.)

```
            if (v >= 0)
                continue;
            futex_wait(mutex, v);
        }
}

void mutex_unlock(int *mutex) {
    /* Adding 0x80000000 to the counter results in 0 if and only if
       there are not other interested threads */
    if (atomic_add_zero(mutex, 0x80000000))
        return;
    /* There are other threads waiting for this mutex,
       wake one of them up */
    futex_wake(mutex);
}
```

Linux-based Futex Locks (Cont.)

60

Two-Phase Locks

- A two-phase lock realizes that spinning can be useful if the lock *is about to* be released.
  - **First phase**
    - The lock spins for a while, *hoping that* it can acquire the lock.
    - If the lock is not acquired during the first spin phase, a second phase is entered,
  - **Second phase**
    - The caller is put to sleep.
    - The caller is only woken up when the lock becomes free later.

61

1. **Concurrency Introduction**
2. **Locks**
3. **基于Lock的并发数据结构**
4. Condition Variables 条件变量
5. Semaphore 信号量
6. 常见并发问题
7. 基于事件的并发

Lock-based Concurrent Data structure

- Adding locks to a data structure makes the structure **thread safe**.
  - How locks are added determine both the correctness and performance of the data structure.

Example: Concurrent Counter without Lock

- Simple but not scalable

```c
typedef struct __counter_t {
        int value;
} counter_t;

void init(counter_t *c) {
        c->value = 0;
}

void increment(counter_t *c) {
        c->value++;
}

void decrement(counter_t *c) {
        c->value--;
}

int get(counter_t *c) {
        return c->value;
}
```


64

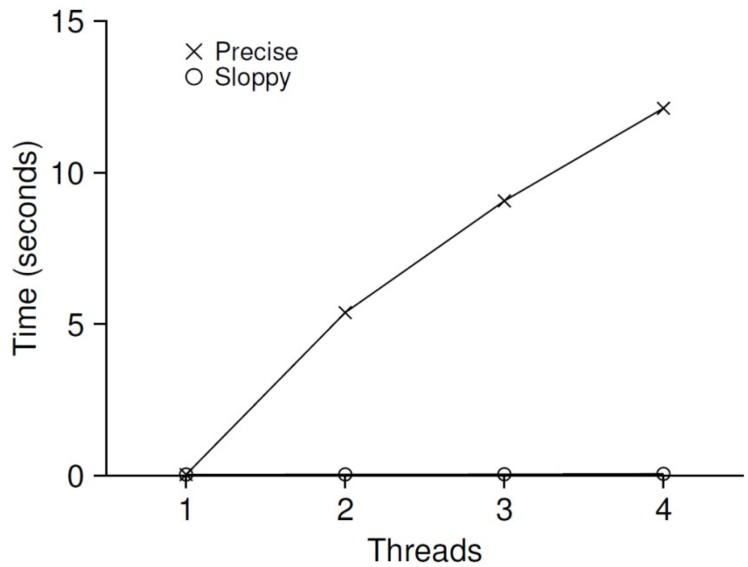
Add a single lock

- acquired when calling a routine manipulating the data structure.

```
typedef struct __counter_t {
        int value;
        pthread_lock_t lock;
} counter_t;

void init(counter_t *c) {
        c->value = 0;
        Pthread_mutex_init(&c->lock, NULL);
}

void increment(counter_t *c) {
        Pthread_mutex_lock(&c->lock);
        c->value++;
        Pthread_mutex_unlock(&c->lock);
}
void decrement(counter_t *c) {
        Pthread_mutex_lock(&c->lock);
        c->value--;
        Pthread_mutex_unlock(&c->lock);
}

int get(counter_t *c) {
        Pthread_mutex_lock(&c->lock);
        int rc = c->value;
        Pthread_mutex_unlock(&c->lock);
        return rc;
}
```

The performance cost of the simple approach

- Each thread updates a single shared counter.
  - Each thread updates the counter one million times.
  - iMac with four Intel 2.7GHz i5 CPUs.



Performance of
Traditional vs. Sloppy Counters
(Threshold of Sloppy, S, is set to 1024)

Synchronized counter scales poorly.

66

Perfect Scaling

- Even though more work is done, it is **done in parallel**.
- The time taken to complete the task is *not increased*.

Sloppy counter

- The sloppy counter works by representing …
  - A single **logical counter** via numerous local physical counters, on per CPU core
  - A single **global counter**
  - There are **lock**s:
    - One fore each local counter and one for the global counter

- Example: on a machine with four CPUs
  - Four local counters
  - One global counter

The basic idea of sloppy counting

- When a thread running on a core wishes to increment the counter.
  - It increment its local counter.
  - Each CPU has its own local counter:
    - Threads across CPUs can update local counters *without contention*.
    - Thus counter updates are scalable.
  - The local values are periodically transferred to the global counter.
    - Acquire the global lock
    - Increment it by the local counter's value
    - The local counter is then reset to zero.

The basic idea of sloppy counting (Cont.)

- How often the local-to-global transfer occurs is determined by a threshold, S (sloppiness).
  - The smaller S:
    - The more the counter behaves like the *non-scalable counter*.
  - The bigger S:
    - The more scalable the counter.
    - The further off the global value might be from the *actual count*.

Sloppy counter example

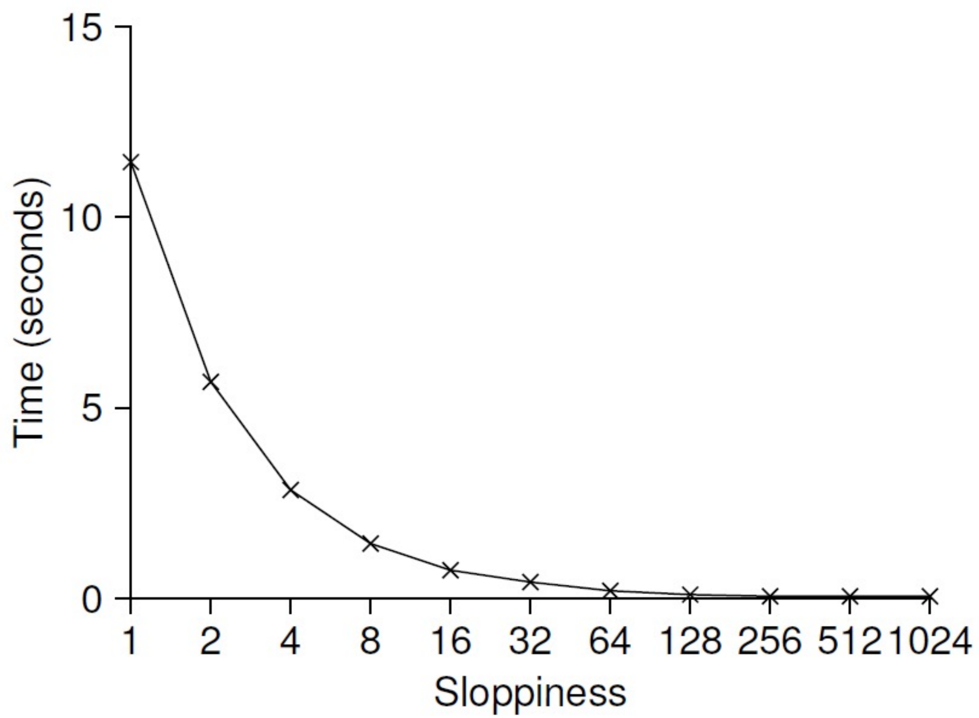
- Tracing the Sloppy Counters
  - The threshold S is set to 5.
  - There are threads on each of 4 CPUs
  - Each thread updates their local counters $L_1 \ldots L_4$.

Time	L_1	L_2	L_3	L_4	G
0	0	0	0	0	0
1	0	0	1	1	0
2	1	0	2	1	0
3	2	0	3	1	0
4	3	0	3	2	0
5	4	1	3	3	0
6	5 → 0	1	3	4	5 (from L_1)
7	0	2	4	5 → 0	10 (from L_4)

71

Importance of the threshold value *S*

- Each four threads increments a counter 1 million times on four CPUs.
  - Low S → Performance is **poor**, The global count is always quire **accurate**.
  - High S → Performance is **excellent**, The global count **lags**.



Scaling Sloppy Counters

72

Sloppy Counter Implementation

HARBIN INSTITUTE OF TECHNOLOGY

```
typedef struct __counter_t {
    int global;                      // global count
    pthread_mutex_t glock;      // global lock
    int local[NUMCPUS];           // local count (per cpu)
    pthread_mutex_t llock[NUMCPUS]; // ... and locks
    int threshold;       // update frequency
} counter_t;

// init: record threshold, init locks, init values
//       of all local counts and global count
void init(counter_t *c, int threshold) {
    c->thres hold = threshold;

    c->global = 0;
    pthread_mutex_init(&c->glock, NULL);

    int i;
    for (i = 0; i < NUMCPUS; i++) {
        c->local[i] = 0;
        pthread_mutex_init(&c->llock[i], NULL);
    }
}
```


73

Sloppy Counter Implementation (Cont.)

```
(Cont.)
    // update: usually, just grab local lock and update local amount
    //         once local count has risen by 'threshold', grab global
    //         lock and transfer local values to it
    void update(counter_t *c, int threadID, int amt) {
        pthread_mutex_lock(&c->llock[threadID]);
        c->local[threadID] += amt;          // assumes amt > 0
        if (c->local[threadID] >= c->threshold) { // transfer to global
            pthread_mutex_lock(&c->glock);
            c->global += c->local[threadID];
            pthread_mutex_unlock(&c->glock);
            c->local[threadID] = 0;
        }
        pthread_mutex_unlock(&c->llock[threadID]);
    }

    // get: just return global amount (which may not be perfect)
    int get(counter_t *c) {
        pthread_mutex_lock(&c->glock);
        int val = c->global;
        pthread_mutex_unlock(&c->glock);
        return val;         // only approximate!
    }
```

Concurrent Linked Lists

```
// basic node structure
typedef struct __node_t {
        int key;
        struct __node_t *next;
} node_t;

// basic list structure (one used per list)
typedef struct __list_t {
        node_t *head;
        pthread_mutex_t lock;
} list_t;

void List_Init(list_t *L) {
        L->head = NULL;
        pthread_mutex_init(&L->lock, NULL);
}

```

(Cont.)

75

Concurrent Linked Lists

(Cont.)

```
    int List_Insert(list_t *L, int key) {
        pthread_mutex_lock(&L->lock);
        node_t *new = malloc(sizeof(node_t));
        if (new == NULL) {
            perror("malloc");
            pthread_mutex_unlock(&L->lock);
        return -1; // fail
        new->key = key;
        new->next = L->head;
        L->head = new;
        pthread_mutex_unlock(&L->lock);
        return 0; // success
}
    int List_Lookup(list_t *L, int key) {
        pthread_mutex_lock(&L->lock);
        node_t *curr = L->head;
        while (curr) {
            if (curr->key == key) {
                pthread_mutex_unlock(&L->lock);
                return 0; // success
            }
            curr = curr->next;
        }
        pthread_mutex_unlock(&L->lock);
        return -1; // failure
    }
```

Concurrent Linked Lists (Cont.)

- The code **acquires** a lock in the insert routine upon entry.
- The code **releases** the lock upon exit.
  - If `malloc()` happens to *fail*, the code must also release the lock before failing the insert.
  - This kind of exceptional control flow has been shown to be quite error prone.
  - **Solution**: The lock and release *only surround* the actual critical section in the insert code

Concurrent Linked List: Rewritten

```
void List_Init(list_t *L) {
        L->head = NULL;
        pthread_mutex_init(&L->lock, NULL);
}

void List_Insert(list_t *L, int key) {
        // synchronization not needed
        node_t *new = malloc(sizeof(node_t));
        if (new == NULL) {
                perror("malloc");
                return;
        }
        new->key = key;

        // just lock critical section
        pthread_mutex_lock(&L->lock);
        new->next = L->head;
        L->head = new;
        pthread_mutex_unlock(&L->lock);
}

```


78

Concurrent Linked List: Rewritten (Cont.)

(Cont.)

```
int List_Lookup(list_t *L, int key) {
        int rv = -1;
        pthread_mutex_lock(&L->lock);
        node_t *curr = L->head;
        while (curr) {
                if (curr->key == key) {
                        rv = 0;
                        break;
                }
                curr = curr->next;
        }
        pthread_mutex_unlock(&L->lock);
        return rv; // now both success and failure
}
```


79

Scaling Linked List

- Hand-over-hand locking (lock coupling)
  - Add **a lock per node** of the list instead of having a single lock for the entire list.
  - When traversing the list,
    - First grabs the next node's lock.
    - And then releases the current node's lock.
  - Enable a high degree of concurrency in list operations.
    - However, in practice, the overheads of acquiring and releasing locks for each node of a list traversal is *prohibitive*.

80

Michael and Scott Concurrent Queues

- There are two locks.
  - One for the **head** of the queue.
  - One for the **tail**.
  - The goal of these two locks is to enable concurrency of *enqueue* and *dequeue* operations.

- Add a dummy node
  - Allocated in the queue initialization code
  - Enable the separation of head and tail operations

Concurrent Queues (Cont.)

```
typedef struct __node_t {
        int value;
        struct __node_t *next;
} node_t;

typedef struct __queue_t {
        node_t *head;
        node_t *tail;
        pthread_mutex_t headLock;
        pthread_mutex_t tailLock;
} queue_t;

void Queue_Init(queue_t *q) {
        node_t *tmp = malloc(sizeof(node_t));
        tmp->next = NULL;
        q->head = q->tail = tmp;
        pthread_mutex_init(&q->headLock, NULL);
        pthread_mutex_init(&q->tailLock, NULL);
}

(Cont.)
```


82

Concurrent Queues (Cont.)

```
(Cont.)
void Queue_Enqueue(queue_t *q, int value) {
        node_t *tmp = malloc(sizeof(node_t));
        assert(tmp != NULL);

        tmp->value = value;
        tmp->next = NULL;

        pthread_mutex_lock(&q->tailLock);
        q->tail->next = tmp;
        q->tail = tmp;
         pthread_mutex_unlock(&q->tailLock);
}
int Queue_Dequeue(queue_t *q, int *value) {
        pthread_mutex_lock(&q->headLock);
        node_t *tmp = q->head;
        node_t *newHead = tmp->next;
        if (newHead == NULL) {
                pthread_mutex_unlock(&q->headLock);
                return -1; // queue was empty
        }
        *value = newHead->value;
        q->head = newHead;
        pthread_mutex_unlock(&q->headLock);
        free(tmp);
        return 0;
}
```

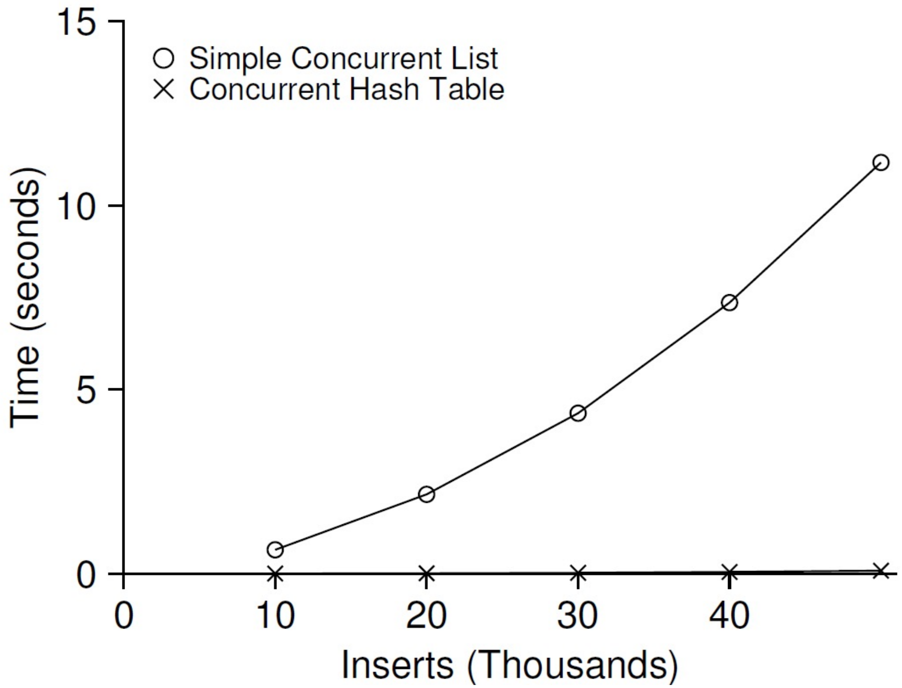
Concurrent Hash Table

- Focus on a simple hash table
  - The hash table does not resize.
  - Built using the concurrent lists
  - It uses a lock per hash bucket each of which is represented by *a list*.

84

Performance of Concurrent Hash Table

- From 10,000 to 50,000 concurrent updates from each of four threads.
  - iMac with four Intel 2.7GHz i5 CPUs.



The simple concurrent hash table scales magnificently.

85

Concurrent Hash Table

哈爾濱工業大學 HARBIN INSTITUTE OF TECHNOLOGY

```c
#define BUCKETS (101)

typedef struct __hash_t {
        list_t lists[BUCKETS];
} hash_t;

void Hash_Init(hash_t *H) {
        int i;
        for (i = 0; i < BUCKETS; i++) {
                List_Init(&H->lists[i]);
        }
}

int Hash_Insert(hash_t *H, int key) {
        int bucket = key % BUCKETS;
        return List_Insert(&H->lists[bucket], key);
}

int Hash_Lookup(hash_t *H, int key) {
        int bucket = key % BUCKETS;
        return List_Lookup(&H->lists[bucket], key);
}
```

Module 5: Concurrency & Synchronization
并发与同步

1. **Concurrency Introduction**
2. **Locks**
3. 基于Lock的并发数据结构
4. **Condition Variables 条件变量**
5. Semaphore 信号量
6. 常见并发问题
7. 基于事件的并发

Condition Variables 条件变量的引入

- There are many cases where a thread wishes to <u>check</u> whether a **condition** is true before continuing its execution. 一个线程需要检查另一个的状态，并据此决定自己是否继续执行
- Example:
  - A parent thread might wish to check whether a child thread has *completed*.
  - This is often called a `join()`.

Condition Variables (Cont.)

HARBIN INSTITUTE OF TECHNOLOGY

A Parent Waiting For Its Child

```
void *child(void *arg) {
    printf("child\n");
    // XXX how to indicate we are done?
    return NULL;
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(&c, NULL, child, NULL); // create child
    // XXX how to wait for child?
    printf("parent: end\n");
    return 0;
}
```

What we would like to see here is:

```
parent: begin
child
parent: end
```


89

Parent waiting fore child: Spin-based Approach

```c
volatile int done = 0;

void *child(void *arg) {
    printf("child\n");
    done = 1;
    return NULL;
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(&c, NULL, child, NULL); // create child
    while (done == 0)
        ; // spin
    printf("parent: end\n");
    return 0;
}
```

- This is hugely inefficient as the parent spins and **wastes CPU time**.

90

How to wait for a condition

- Condition variable 本质上是一个队列及对该队列的操作原语
  - **Waiting** on the condition
    - An explicit queue that threads can put themselves on when some state of execution is not as desired.
  - **Signaling** on the condition
    - Some other thread, *when it changes said state*, can wake one of those waiting threads and allow them to continue.

91

Definition and Routines

- Declare condition variable

```
pthread_cond_t c;
```

  - Proper initialization is required.

- Operation (the POSIX calls)

```
pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);    // wait()
pthread_cond_signal(pthread_cond_t *c);                      // signal()
```

  - The wait() call takes a <u>mutex</u> as a parameter.
    - The wait() call release the lock and put the calling thread to sleep.
    - When the thread wakes up, it must re-acquire the lock.

92

Parent waiting for Child: Use a condition variable

```
int done = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;

void thr_exit() {
        Pthread_mutex_lock(&m);
        done = 1;
        Pthread_cond_signal(&c);
        Pthread_mutex_unlock(&m);
}

void *child(void *arg) {
        printf("child\n");
        thr_exit();
        return NULL;
}

void thr_join() {
        Pthread_mutex_lock(&m);
        while (done == 0)
                Pthread_cond_wait(&c, &m);
        Pthread_mutex_unlock(&m);
}
```


93

Parent waiting for Child: Use a condition variable

```
(cont.)
    int main(int argc, char *argv[]) {
        printf("parent: begin\n");
        pthread_t p;
        Pthread_create(&p, NULL, child, NULL);
        thr_join();
        printf("parent: end\n");
        return 0;
    }
```

Create the child thread and continues running itself.

94

Parent waiting for Child: Use a condition variable

- **Parent:**
  - Create the child thread and continues running itself.
  - Call into `thr_join()` to wait for the child thread to complete.
    - Acquire the lock
    - Check if the child is done
    - Put itself to sleep by calling `wait()`
    - Release the lock
- **Child:**
  - Print the message “child”
  - Call `thr_exit()` to wake the parent thread
    - Grab the lock
    - Set the state variable `done`
    - Signal the parent thus waking it.

The importance of the state variable done

```
void thr_exit() {
        Pthread_mutex_lock(&m);
        Pthread_cond_signal(&c);
        Pthread_mutex_unlock(&m);
}

void thr_join() {
        Pthread_mutex_lock(&m);
        Pthread_cond_wait(&c, &m);
        Pthread_mutex_unlock(&m);
}
```

`thr_exit()` and `thr_join()` without variable `done`

- Imagine the case where the *child runs immediately*.
  - The child will signal, but there is <u>no thread asleep</u> on the condition.
  - When the parent runs, it will call wait and be stuck.
  - No thread will ever wake it.

96

Another poor implementation

```
void thr_exit() {
        done = 1;
        Pthread_cond_signal(&c);
}

void thr_join() {
        if (done == 0)
                Pthread_cond_wait(&c);
}
```

- The issue here is a subtle **race condition**.
  - The parent calls `thr_join()`.
    - The parent checks the value of `done`.
    - It will see that it is 0 and try to go to sleep.
    - *Just before* it calls wait to go to sleep, the parent is <u>interrupted</u> and the child runs.
  - The child changes the state variable `done` to 1 and signals.
    - But no thread is waiting and thus no thread is woken.
    - When the parent runs again, it sleeps forever.

97

The Producer/Consumer (Bound Buffer) Problem

- **Producer**
  - Produce data items
  - Wish to place data items in a buffer
- **Consumer**
  - Grab data items out of the buffer consume them in some way
- Example: Multi-threaded web server
  - *A producer* puts HTTP requests in to a work queue
  - *Consumer threads* take requests out of this queue and process them

Bounded buffer

- A bounded buffer is used when you <u>pipe the output</u> of one program into another.
  - Example: `grep foo file.txt | wc -l`
    - The `grep` process is the producer.
    - The wc process is the consumer.
    - Between them is an in-kernel <u>bounded buffer</u>.
  - Bounded buffer is Shared resource → **Synchronized access** is required.

The Put and Get Routines (Version 1)

```
int buffer;
int count = 0;          // initially, empty

void put(int value) {
        assert(count == 0);
        count = 1;
        buffer = value;
}

int get() {
        assert(count == 1);
        count = 0;
        return buffer;
}
```

- Only put data into the buffer when `count` is zero.
  - i.e., when the buffer is *empty*.
- Only get data from the buffer when `count` is one.
  - i.e., when the buffer is *full*.

100

Producer/Consumer Threads (Version 1)

```
void *producer(void *arg) {
    int i;
    int loops = (int) arg;
    for (i = 0; i < loops; i++) {
        put(i);
    }
}

void *consumer(void *arg) {
    int i;
    while (1) {
        int tmp = get();
        printf("%d\n", tmp);
    }
}
```

- **Producer** puts an integer into the shared buffer loops number of times.
- **Consumer** gets the data out of that shared buffer.

101

Producer/Consumer: Single CV and If Statement

```
cond_t cond;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);                 // p1
        if (count == 1)                             // p2
            Pthread_cond_wait(&cond, &mutex);       // p3
        put(i);                                     // p4
        Pthread_cond_signal(&cond);                 // p5
        Pthread_mutex_unlock(&mutex);               // p6
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);                 // c1
        if (count == 0)                             // c2
            Pthread_cond_wait(&cond, &mutex);       // c3
        int tmp = get();                            // c4
        Pthread_cond_signal(&cond);                 // c5
        Pthread_mutex_unlock(&mutex);               // c6
        printf("%d\n", tmp);
    }
}
```

If we have more than one of producer and consumer?

Thread Trace: Broken Solution (Version 1)

T_{c1}	State	T_{c2}	State	T_p	State	Count	Comment
c1	Running		Ready		Ready	0	
c2	Running		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep		Ready	p1	Running	0	
	Sleep		Ready	p2	Running	0	
	Sleep		Ready	p4	Running	1	Buffer now full
	Ready		Ready	p5	Running	1	T_{c1} awoken
	Ready		Ready	p6	Running	1	
	Ready		Ready	p1	Running	1	
	Ready		Ready	p2	Running	1	
	Ready		Ready	p3	Sleep	1	Buffer full; sleep
	Ready	c1	Running		Sleep	1	T_{c2} sneaks in ...
	Ready	c2	Running		Sleep	1	
	Ready	c4	Running		Sleep	0	... and grabs data
	Ready	c5	Running		Ready	0	T_p awoken
	Ready	c6	Running		Ready	0	
c4	Running		Ready		Ready	0	**Oh oh! No data**

Thread Trace: Broken Solution (Version 1)

- The problem arises for a simple reason:
  - After the producer woke T_{c1}, but before T_{c1} ever ran, the state of the bounded buffer *changed by* T_{c2}.
  - There is no guarantee that when the woken thread runs, the state will still be as desired → Mesa semantics.
    - Virtually every system ever built employs *Mesa semantics*.
  - Hoare semantics provides a stronger guarantee that the woken thread will run immediately upon being woken.

Producer/Consumer: Single CV and While

```
cond_t cond;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);                  // p1
        while (count == 1)                           // p2
            Pthread_cond_wait(&cond, &mutex);        // p3
        put(i);                                      // p4
        Pthread_cond_signal(&cond);                  // p5
        Pthread_mutex_unlock(&mutex);                // p6
    }
}
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);                  // c1
        while (count == 0)                           // c2
            Pthread_cond_wait(&cond, &mutex);        // c3
        int tmp = get();                             // c4
        Pthread_cond_signal(&cond);                  // c5
        Pthread_mutex_unlock(&mutex);                // c6
        printf("%d\n", tmp);
    }
}
```

Thread Trace: Broken Solution (Version 2)

T_{c1}	State	T_{c2}	State	T_p	State	Count	Comment
c1	Running		Ready		Ready	0	
c2	Running		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep	c1	Running		Ready	0	
	Sleep	c2	Running		Ready	0	
	Sleep	c3	Sleep		Ready	0	Nothing to get
	Sleep		Sleep	p1	Running	0	
	Sleep		Sleep	p2	Running	0	
	Sleep		Sleep	p4	Running	1	Buffer now full
	Ready		Sleep	p5	Running	1	T_{c1} awoken
	Ready		Sleep	p6	Running	1	
	Ready		Sleep	p1	Running	1	
	Ready		Sleep	p2	Running	1	
	Ready		Sleep	p3	Sleep	1	Must sleep (full)
c2	Running		Sleep		Sleep	1	Recheck condition
c4	Running		Sleep		Sleep	0	T_{c1} grabs data
c5	Running		Ready		Sleep	0	**Oops! Woke T_{c2}**

106

Thread Trace: Broken Solution (Version 2) (Cont.)

T_{c1}	State	T_{c2}	State	T_p	State	Count	Comment
…	…	…	…	…	…	…	(*cont.*)
c6	Running		Ready		Sleep	0	
c1	Running		Ready		Sleep	0	
c2	Running		Ready		Sleep	0	
c3	Sleep		Ready		Sleep	0	Nothing to get
	Sleep	c2	Running		Sleep	0	
	Sleep	c3	Sleep		Sleep	0	**Everyone asleep …**

- A consumer should not wake other consumers, only producers, and vice-versa.

The single Buffer Producer/Consumer Solution

- Use two condition variables and while
  - **Producer** threads wait on the condition `empty`, and signals `fill`.
  - **Consumer** threads wait on `fill` and signal `empty`.

```
cond_t empty, fill;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 1)
            Pthread_cond_wait(&empty, &mutex);
        put(i);
        Pthread_cond_signal(&fill);
        Pthread_mutex_unlock(&mutex);
    }
}
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 0)
            Pthread_cond_wait(&fill, &mutex);
        int tmp = get();
        Pthread_cond_signal(&empty);
        Pthread_mutex_unlock(&mutex);
        printf("%d\n", tmp);
    }
```

The Final Producer/Consumer Solution

- More **concurrency** and **efficiency** → Add more buffer slots.
  - Allow concurrent production or consuming to take place.
  - Reduce context switches.

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

void put(int value) {
    buffer[fill] = value;
    fill = (fill + 1) % MAX;
    count++;
}

int get() {
    int tmp = buffer[use];
    use = (use + 1) % MAX;
    count--;
    return tmp;
}
```

The Final Put and Get Routines

The Final Producer/Consumer Solution (Cont.)

```
cond_t empty, fill;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);                 // p1
        while (count == MAX)                        // p2
            Pthread_cond_wait(&empty, &mutex);      // p3
        put(i);                                     // p4
        Pthread_cond_signal(&fill);                 // p5
        Pthread_mutex_unlock(&mutex);               // p6
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);                 // c1
        while (count == 0)                          // c2
            Pthread_cond_wait(&fill, &mutex);       // c3
        int tmp = get();                            // c4
        Pthread_cond_signal(&empty);                // c5
        Pthread_mutex_unlock(&mutex);               // c6
        printf("%d\n", tmp);
    }
}
```

Covering Conditions

- Assume there are zero bytes free
  - Thread T_a calls `allocate(100)`.
  - Thread T_b calls `allocate(10)`.
  - Both T_a and T_b wait on the condition and go to sleep.
  - Thread T_c calls `free(50)`.

Which waiting thread should be woken up?

111

Covering Conditions (Cont.)

```
// how many bytes of the heap are free?
int bytesLeft = MAX_HEAP_SIZE;

// need lock and condition too
cond_t c;
mutex_t m;

void *
allocate(int size) {
    Pthread_mutex_lock(&m);
    while (bytesLeft < size)
        Pthread_cond_wait(&c, &m);
    void *ptr = ...;                  // get mem from heap
    bytesLeft -= size;
    Pthread_mutex_unlock(&m);
    return ptr;
}

void free(void *ptr, int size) {
    Pthread_mutex_lock(&m);
    bytesLeft += size;
    Pthread_cond_signal(&c);          // whom to signal??
    Pthread_mutex_unlock(&m);
}
```


112

Covering Conditions (Cont.)

- Solution (Suggested by Lampson and Redell)
  - Replace `pthread_cond_signal()` with `pthread_cond_broadcast()`
  - `pthread_cond_broadcast()`
    - Wake up **all waiting threads.**
    - Cost: too many threads might be woken.
    - Threads that shouldn't be awake will simply wake up, re-check the condition, and then go back to sleep.

113

Module 5: Concurrency & Synchronization 并发与同步

1. **Concurrency Introduction**
2. **Locks**
3. 基于Lock的并发数据结构
4. Condition Variables 条件变量
5. **Semaphore 信号量**
6. 常见并发问题
7. 基于事件的并发

Semaphore: A definition

- An object with an integer value
  - We can manipulate with two routines; `sem_wait()` and `sem_post()`.
  - Initialization

```
#include <semaphore.h>
sem_t s;
sem_init(&s, 0, 1); // initialize s to the value 1
```

- Declare a semaphore `s` and initialize it to the value 1
- The second argument, 0, indicates that the semaphore is <u>shared</u> between *threads in the same process*.

115

Semaphore: wait原语

- sem_wait()

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

- 减1，若大于0，则返回，否则挂起等待被post唤醒
- If the value of the semaphore was *one* or *higher* when called `sem_wait()`, **return right away**.
- It will cause the caller to suspend execution waiting for a subsequent post.
- When negative, the value of the semaphore is equal to the number of waiting threads.

116

Semaphore: Interact with semaphore (Cont.)

- sem_post()

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

- 加1，若有等待线程，则唤醒一个
- Simply **increments** the value of the semaphore.
- If there is a thread waiting to be woken, **wakes** one of them up.

117

Binary Semaphores (Locks)

- What should **x** be?
  - The initial value should be **1**.

```
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);
```


118

Thread Trace: Single Thread Using A Semaphore

Value of Semaphore	Thread 0	Thread 1
1		
1	`call sema_wait()`	
0	`sem_wait() returns`	
0	`(crit sect)`	
0	`call sem_post()`	
1	`sem_post() returns`	

Thread Trace: Two Threads Using A Semaphore

Value	Thread 0	State	Thread 1	State
1		Running		Ready
1	call sem_wait()	Running		Ready
0	sem_wait() retruns	Running		Ready
0	(crit set: begin)	Running		Ready
0	*Interrupt; Switch → T1*	Ready		Running
0		Ready	call sem_wait()	Running
-1		Ready	decrement sem	Running
-1		Ready	(sem < 0)→sleep	sleeping
-1		Running	*Switch → T0*	sleeping
-1	(crit sect: end)	Running		sleeping
-1	call sem_post()	Running		sleeping
0	increment sem	Running		sleeping
0	wake(T1)	Running		Ready
0	sem_post() returns	Running		Ready
0	*Interrupt; Switch → T1*	Ready		Running
0		Ready	sem_wait() retruns	Running
0		Ready	(crit sect)	Running
0		Ready	call sem_post()	Running
1		Ready	sem_post() returns	Running

120

Semaphores As Condition Variables

```c
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

```
parent: begin
child
parent: end
```

The execution result

- What should **x** be?
    - The value of semaphore should be set to is **0**.

121

Thread Trace: Parent Waiting For Child (Case 1)

- The parent call `sem_wait()` before the child has called `sem_post()`.

Value	Parent	State	Child	State
0	`Create(Child)`	Running	*(Child exists; is runnable)*	Ready
0	`call sem_wait()`	Running		Ready
-1	`decrement sem`	Running		Ready
-1	`(sem < 0)→sleep`	sleeping		Ready
-1	*Switch→Child*	sleeping	`child runs`	Running
-1		sleeping	`call sem_post()`	Running
0		sleeping	`increment sem`	Running
0		Ready	`wake(Parent)`	Running
0		Ready	`sem_post() returns`	Running
0		Ready	*Interrupt; Switch→Parent*	Ready
0	`sem_wait() retruns`	Running		Ready

Thread Trace: Parent Waiting For Child (Case 2)

- The child runs to completion before the parent call `sem_wait()`.

Value	Parent	State	Child	State
0	`Create(Child)`	Running	*(Child exists; is runnable)*	Ready
0	*`Interrupt; switch→Child`*	Ready	`child runs`	Running
0		Ready	`call sem_post()`	Running
1		Ready	`increment sem`	Running
1		Ready	`wake(nobody)`	Running
1		Ready	`sem_post() returns`	Running
1	`parent runs`	Running	*`Interrupt; Switch→Parent`*	Ready
1	`call sem_wait()`	Running		Ready
0	`decrement sem`	Running		Ready
0	`(sem<0)→awake`	Running		Ready
0	`sem_wait() retruns`	Running		Ready

The Producer/Consumer (Bounded-Buffer) Problem

- **Producer**: `put()` interface
  - Wait for a buffer to become *empty* in order to put data into it.
- **Consumer**: `get()` interface
  - Wait for a buffer to become *filled* before using it.

```
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;
}
```


124

The Producer/Consumer (Bounded-Buffer) Problem

```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);
    }
}
…
```

First Attempt: Adding the Full and Empty Conditions

125

The Producer/Consumer (Bounded-Buffer) Problem

```
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
    // …
}
```

First Attempt: Adding the Full and Empty Conditions (Cont.)

- Imagine that `MAX` is greater than 1 .
  - If there are multiple producers, race condition can happen at line *f1*.
  - It means that the old data there is overwritten.
- We’ve forgotten here is **mutual exclusion**.
  - The filling of a buffer and incrementing of the index into the buffer is a critical section.

126

A Solution: Adding Mutual Exclusion

```
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)
     }
}

(Cont.)
```

Adding Mutual Exclusion (Incorrectly)

127

A Solution: Adding Mutual Exclusion

```
(Cont.)
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);
	}
}
```

Adding Mutual Exclusion (Incorrectly)

128

A Solution: Adding Mutual Exclusion (Cont.)

- Imagine two thread: one producer and one consumer.
  - The consumer **acquire** the `mutex` (line c0).
  - The consumer **calls** `sem_wait()` on the full semaphore (line c1).
  - The consumer is **blocked** and **yield** the CPU.
    - The consumer still holds the mutex!
  - The producer **calls** `sem_wait()` on the binary `mutex` semaphore (line p0).
  - The producer is now **stuck** waiting too. a classic deadlock.

Finally, A Working Solution

```
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
    }
}

(Cont.)
```

Adding Mutual Exclusion (Correctly)

130

Finally, A Working Solution

```
(Cont.)
 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)

131

Reader-Writer Locks

- Imagine a number of concurrent list operations, including **inserts** and simple **lookups**.
  - **insert:**
    - Change the state of the list
    - A traditional critical section makes sense.
  - **lookup:**
    - Simply *read* the data structure.
    - As long as we can guarantee that no insert is on-going, we can allow many lookups to proceed concurrently.

This special type of lock is known as a reader-write lock.

A Reader-Writer Locks

- Only **a single writer** can acquire the lock.
- Once a reader has acquired a read lock,
  - **More readers** will be allowed to acquire the read lock too.
  - A writer will have to wait until all readers are finished.

```
typedef struct _rwlock_t {
    sem_t lock;      // binary semaphore (basic lock)
    sem_t writelock; // used to allow ONE writer or MANY readers
    int readers;     // count of readers reading 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);
    …
```


133

A Reader-Writer Locks (Cont.)

```
    rw->readers++;
    if (rw->readers == 1)
        sem_wait(&rw->writelock); // first reader acquires writelock
    sem_post(&rw->lock);
}

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

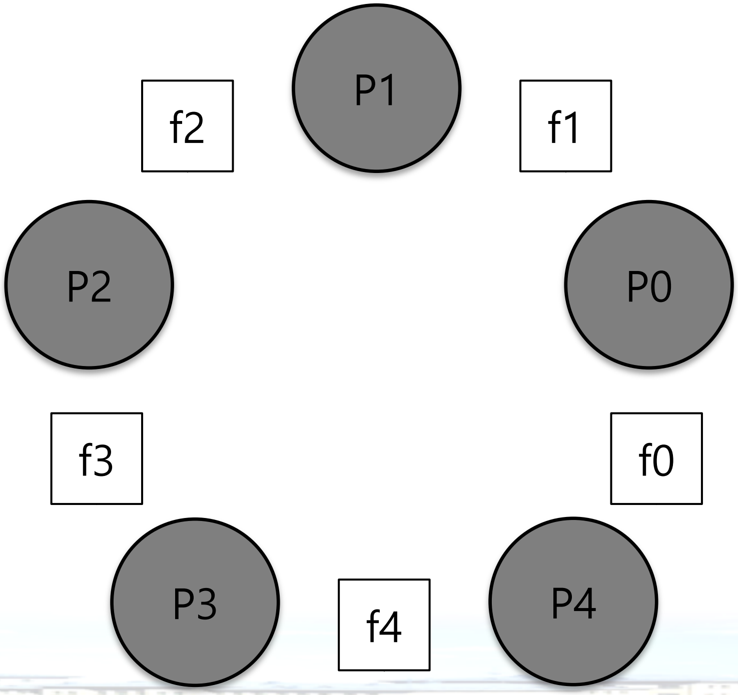

134

A Reader-Writer Locks (Cont.)

- The reader-writer locks have fairness problem.
  - It would be relatively easy for reader to **starve writer**.
  - How to prevent more readers from entering the lock once a writer is waiting?

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*.
  - **The contention for these forks.**



The Dining Philosophers (Cont.)

- Key challenge
  - There is **no deadlock**.
  - **No** philosopher **starves** and never gets to eat.
  - **Concurrency** is high.

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

Basic loop of each philosopher

```
// helper functions
int left(int p) { return p; }

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

Helper functions (Downey's solutions)

- Philosopher `p` wishes to refer to the fork on their left → call `left(p)`.
- Philosopher `p` wishes to refer to the fork on their right → call `right(p)`.

The Dining Philosophers (Cont.)

- We need some **semaphore**, one for each fork: `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)]);
}
```

The `getforks()` and `putforks()` Routines (Broken Solution)

- Deadlock occur!
  - 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*.

138

A Solution: Breaking The Dependency

- Change how forks are acquired.
  - Let's assume that philosopher 4 acquire the forks in a *different order*.

```
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)]);
      }
}
```

- There is no situation where each philosopher grabs one fork and is stuck waiting for another. **The cycle of waiting is broken**.

139

How To Implement Semaphores

- Build our own version of semaphores called Zemaphores

```
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);
}
…
```


140

How To Implement Semaphores (Cont.)

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

- Zemaphore don’t maintain the invariant that *the value of* the semaphore.
  - The value never be lower than zero.
  - This behavior is **easier** to implement and **matches** the current Linux implementation.

141

例题

【例题】当 V 操作唤醒一个等待进程时，被唤醒进程变为（ ）态。

A. 运行

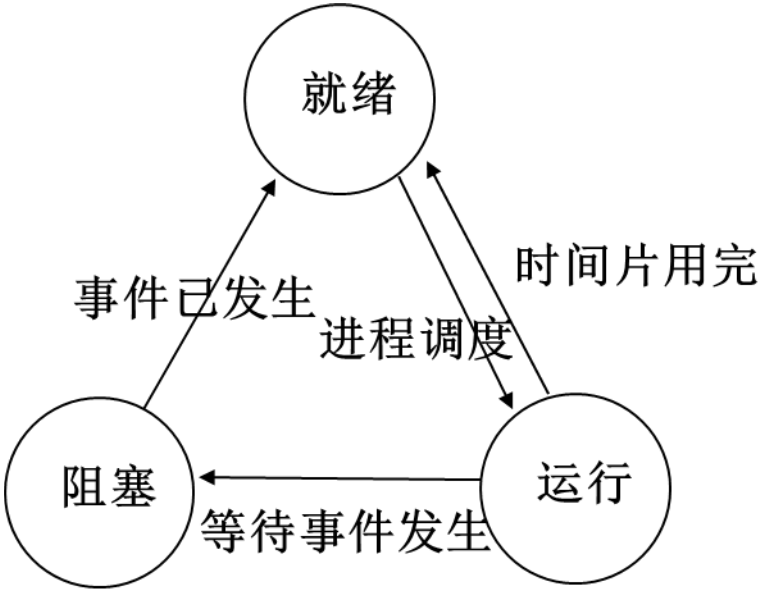
B. 阻塞

C. 就绪

D. 完成

答案：C

等待唤醒的进程处于阻塞态，被唤醒后进入就绪态。只有就绪进程能获得处理器资源，被唤醒的进程并不能直接转化为运行态。



进程:状态转换

例题

【例题】有三个进程共享同一程序段，而每次只允许两个进程进入该程序段，若用 PV 操作同步机制，则信号量 S 的取值范围是（ ）

A. 2,1,0,-1

B. 3,2,1,0

C. 2,1,0,-1,-2

D. 1,0,-1,-2

答案：A

因为每次允许两个进程进入该程序段，信号量最大值取2（否则三个进程可以同时进入程序段）。至多有三个进程申请，则信号量最小为-1。