FDU 操作系统 3. 并发

本文参考以下教材:

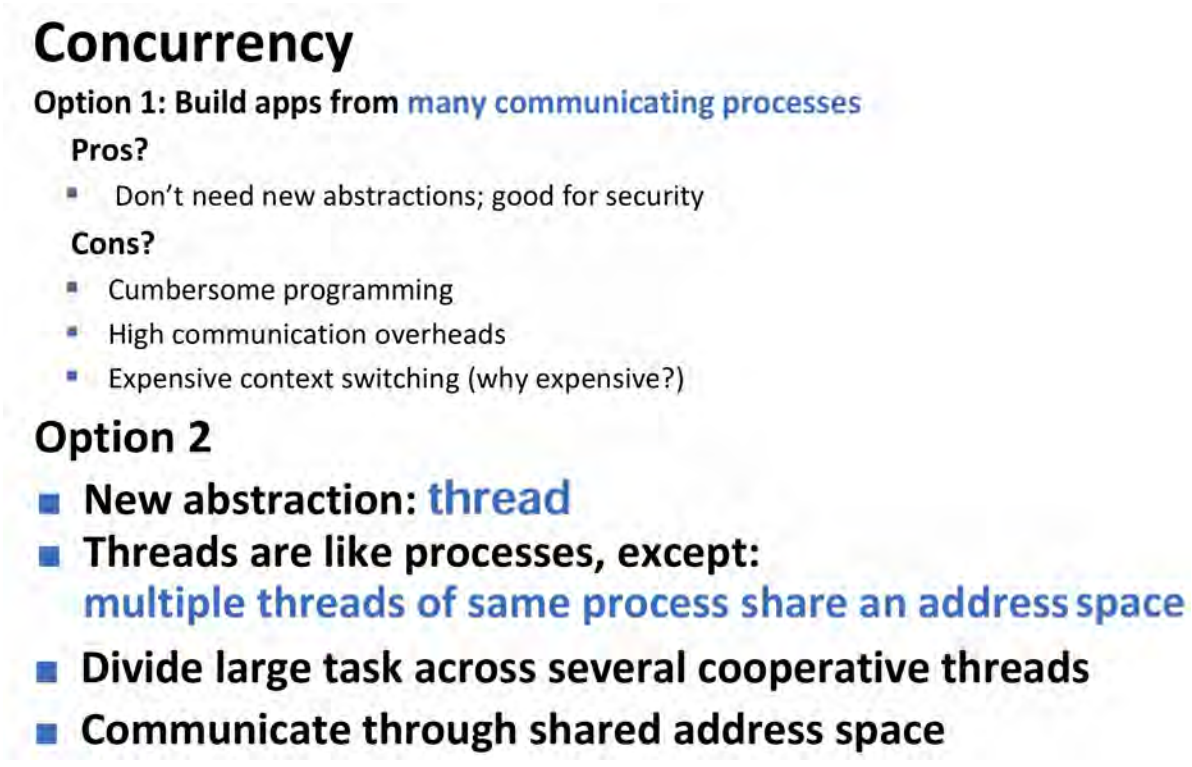
- Operating Systems: Three Easy Pieces (R. H. Arpaci-Dusseau & A. C. Arpaci-Dusseau) Chapter $26 \sim 33$
- 操作系统: 三个简单的部分 (王海鹏 译) 第 $26 \sim 33$ 章

欢迎批评指正!

3.1 An Introduction

3.1.1 抽象: 线程

我们将介绍为单个运行进程提供的新抽象: **线程** (thread)
经典观点是一个程序只有一个执行点 (一个程序计数器，用来指向下一步要执行的指令)，
但**多线程** (multi-threaded) 程序会有多个执行点 (多个程序计数器，每个都用于取指令和执行)
从某种角度来看，每个线程类似于独立的进程，只有一点区别: 它们共享地址空间，从而能够访问相同的数据.
(线程之间可以共享文件描述符 `fd`，进程之间不行，需要使用 `binder` 工具共享文件描述符)



Concurrency

Option 1: Build apps from many communicating processes

Pros?

- Don't need new abstractions; good for security

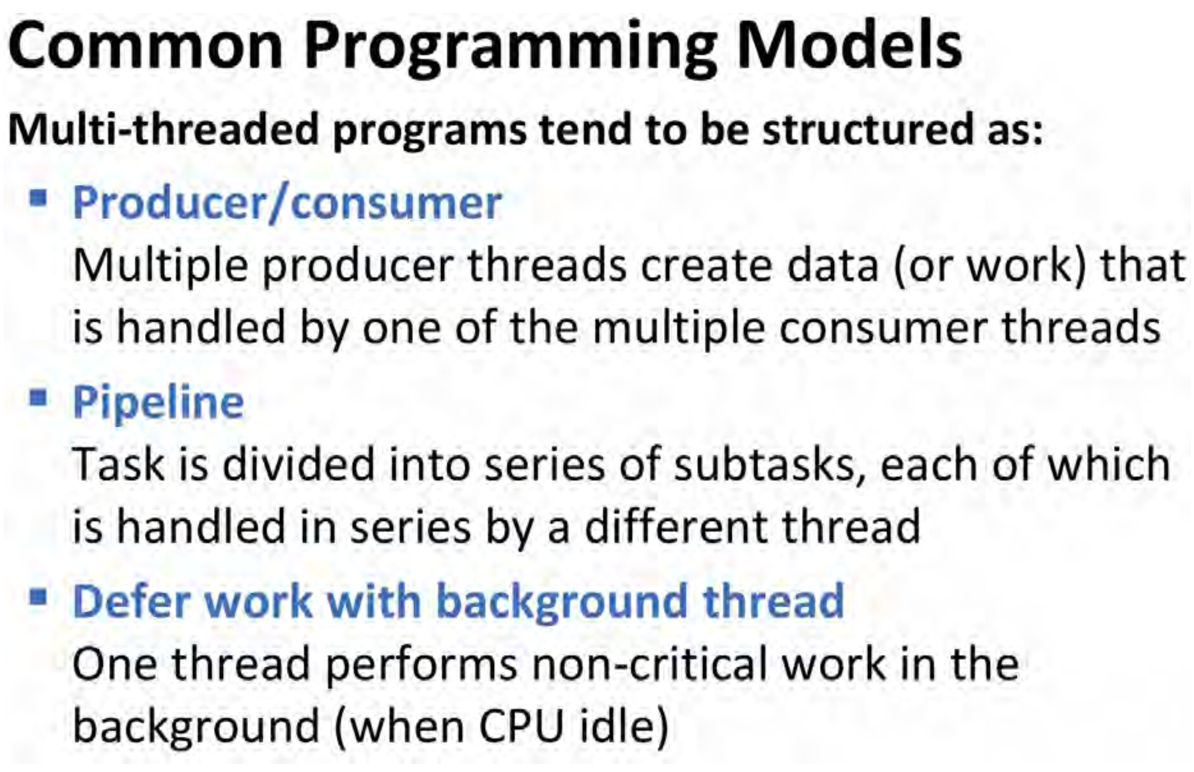
Cons?

- Cumbersome programming
- High communication overheads
- Expensive context switching (why expensive?)

Option 2

- New abstraction: thread
- Threads are like processes, except: multiple threads of same process share an address space
- Divide large task across several cooperative threads
- Communicate through shared address space

因此单个线程的状态与进程状态非常类似 (我们现在可以称之为单线程 (single-threaded) 进程).
线程有一个程序计数器，记录程序从哪里获取指令，每个线程有自己的一组用于计算的寄存器.
所以如果有两个线程运行在一个处理器上，从运行一个线程 T_1 切换到另一个线程 T_2 时，必定发生上下文切换 (context switch)
线程之间的上下文切换类似于进程间的上下文切换.
对于进程，我们将状态保存到进程控制块 (Process Control Block, PCB)
对于多线程，我们需要若干个线程控制块 (Thread Control Block, TCB)，保存每个线程的状态.
但是与进程相比，线程之间的上下文切换有一点主要区别: 地址空间保持不变 (即不需要切换当前使用的页表)



Common Programming Models

Multi-threaded programs tend to be structured as:

- **Producer/consumer**
  Multiple producer threads create data (or work) that is handled by one of the multiple consumer threads
- **Pipeline**
  Task is divided into series of subtasks, each of which is handled in series by a different thread
- **Defer work with background thread**
  One thread performs non-critical work in the background (when CPU idle)

线程和进程之间的另一个主要区别在于栈.
在简单的单线程进程地址空间模型中只有一个栈，通常位于地址空间的底部 (如左图).
然而在多线程的进程中，每个线程独立运行，当然可以调用各种例程来完成正在执行的任何工作.
不是地址空间中只有一个栈，而是每个线程都有一个栈.
假设有一个多线程的进程，它有两个线程，结果地址空间看起来不同 (如右图)

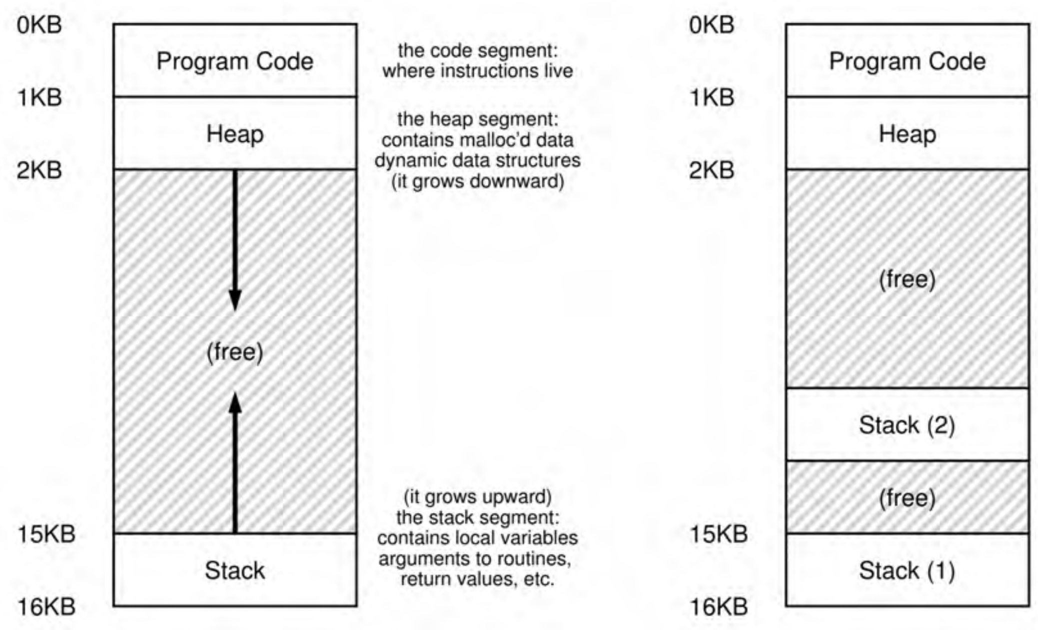
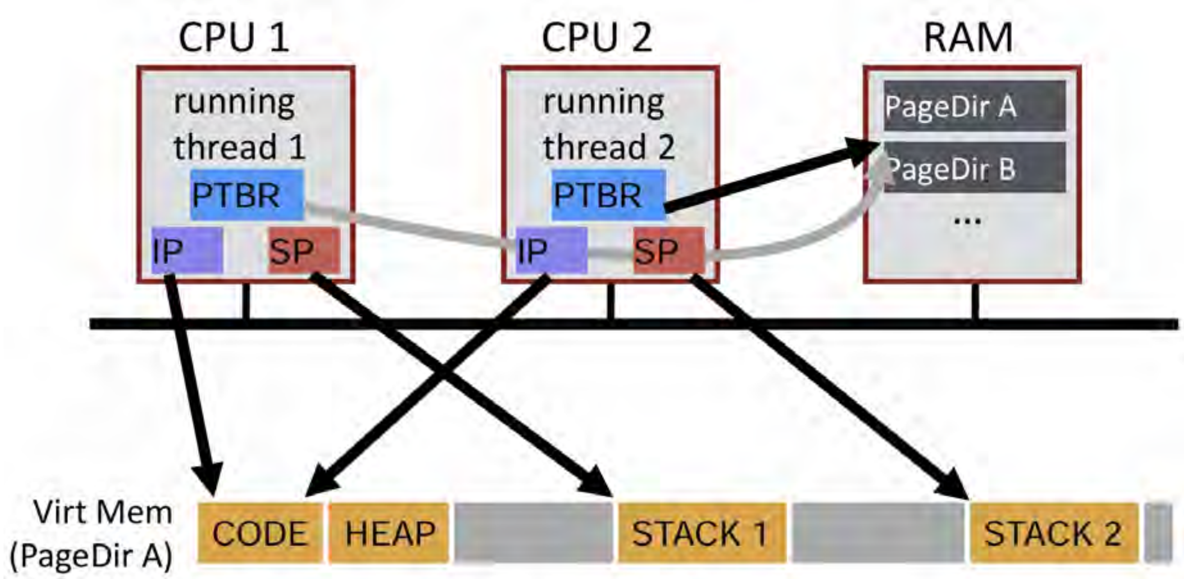


Figure 26.1: **Single-Threaded And Multi-Threaded Address Spaces**

在图中可以看到双线程的进程的两个栈跨越了进程的地址空间.
因此所有位于栈上的变量、参数、返回值和其他放在栈上的东西，
将被放置在有时称为**线程本地** (thread-local) 存储的地方，即相关线程的栈.
多个栈也破坏了地址空间布局的美感，它们既不共享也不隔离.
在单线程进程的地址空间中，堆和栈可以互不影响地增长直到空间耗尽.
多个栈就没有这么简单了，不过幸运的是，栈通常不会很大 (除了大量使用递归的程序)



What threads share page directories? Yes!

Threads belonging to the same process share page directory

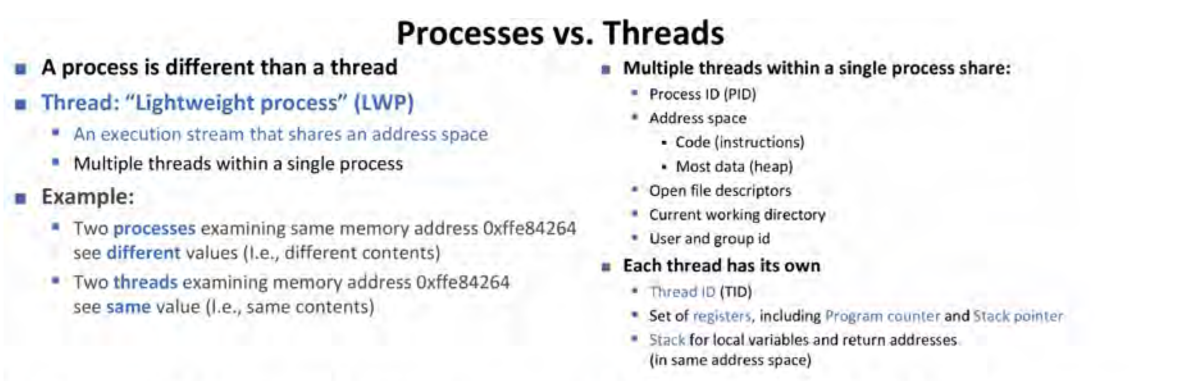
Do threads share Instruction Pointer? No!

Share code, but each thread may be executing different code at the same time → Different Instruction Pointers

Do threads share heap? Yes!

Do threads share stack pointer? No!

Threads executing different functions need different stacks



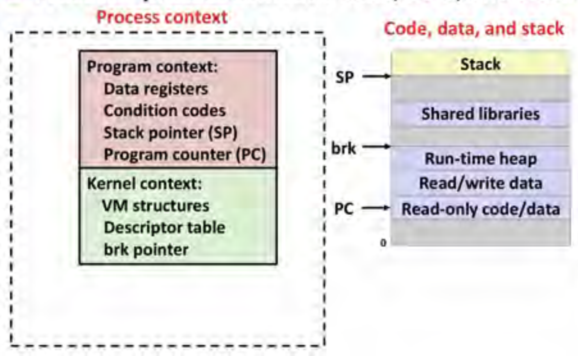
Processes vs. Threads

- A process is different than a thread
- Thread: "Lightweight process" (LWP)
  - An execution stream that shares an address space
  - Multiple threads within a single process
- Example:
  - Two **processes** examining same memory address 0xffe84264 see **different** values (I.e., different contents)
  - Two **threads** examining memory address 0xffe84264 see **same** value (I.e., same contents)

- Multiple threads within a single process share:
  - Process ID (PID)
  - Address space
    - Code (instructions)
    - Most data (heap)
  - Open file descriptors
  - Current working directory
  - User and group id
- Each thread has its own
  - Thread ID (TID)
  - Set of registers, including Program counter and Stack pointer
  - Stack for local variables and return addresses (in same address space)

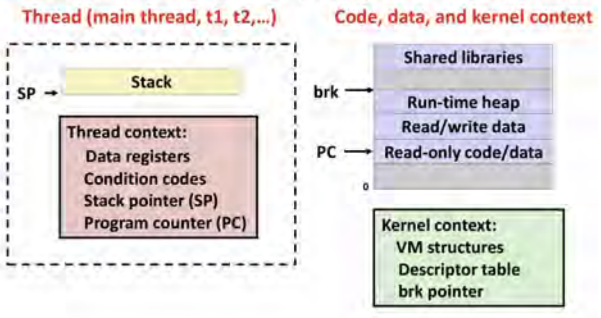
Traditional View of a Process

- Process = process context + code, data, and stack



Alternate View of a Process

- Process = threads + code, data, and kernel context



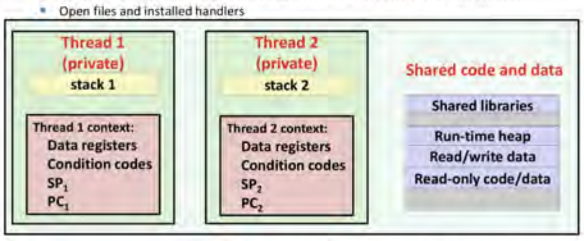
Separation of data is not strictly enforced: (不同线程的数据既不完全共享，也不完全隔离)

A Process With Multiple Threads

- **Multiple threads can be associated with a process**
  - Each thread has its own logical control flow
  - Each thread shares the same code, data, and kernel context
  - Each thread has its own stack for local variables
    - but not protected from other threads
  - Each thread has its own thread id (TID)

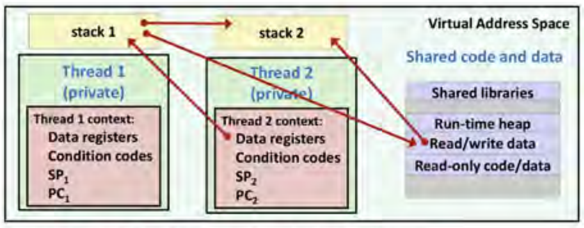
Threads Memory Model: Conceptual

- **Multiple threads run within the context of a single process**
- **Each thread has its own separate thread context**
  - Thread ID, stack, stack pointer, PC, condition codes(eflags), and General Purpose registers
- **All threads share the remaining process context**
  - Code, data, heap, and shared library segments of the process virtual address space
  - Open files and installed handlers



Threads Memory Model: Actual

- **Separation of data is not strictly enforced:**
  - Register values are truly separate and protected, but...
  - Any thread can read and write the stack of any other thread



The mismatch between the conceptual and operation model is a source of confusion and errors

Mapping Variable Instances to Memory

- **Global variables**
  - *Def:* Variable declared outside of a function
  - **Virtual memory contains exactly one instance of any global variable**
- **Local variables**
  - *Def:* Variable declared inside function without `static` attribute
  - **Each thread stack contains one instance of each local variable**
- **Local static variables**
  - *Def:* Variable declared inside function with the `static` attribute
  - **Virtual memory contains exactly one instance of any local static variable**

Example Program to Illustrate Sharing

```c
char **ptr;  /* global var */

int main(int argc, char *argv[])
{
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };

    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
            NULL,
            thread,
            (void *)i);
    Pthread_exit(NULL);
}
```
sharing.c

```c
void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;

    printf("[%ld]:  %s (cnt=%d)\n",
          myid, ptr[myid], ++cnt);
    return NULL;
}
```

Peer threads reference main thread's stack indirectly through global ptr variable

A common, but inelegant way to pass a single argument to a thread routine

Global var: 1 instance (`ptr` [data])

Local vars: 1 instance (`i.m, msgs.m`)

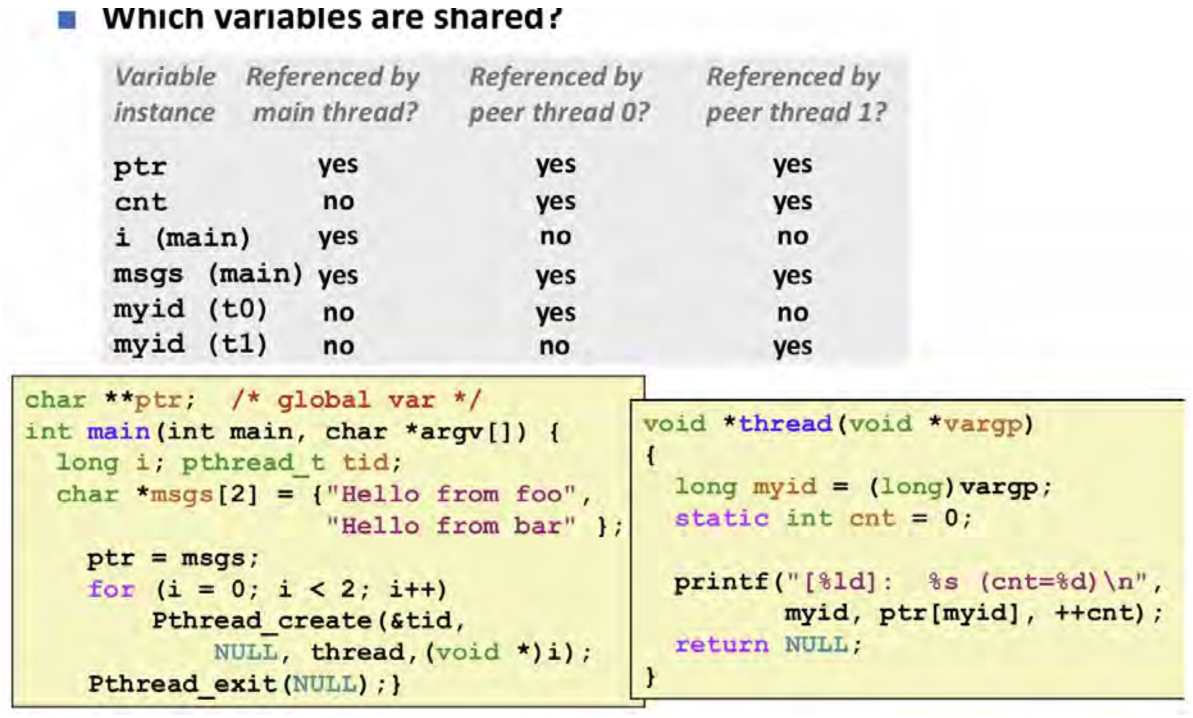
```c
char **ptr;  /* global var */

int main(int main, char *argv[])
{
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };

    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
            NULL,
            thread,
            (void *)i);
    Pthread_exit(NULL);
}
```
sharing.c

Local var: 2 instances (
  `myid.p0` [peer thread 0's stack],
  `myid.p1` [peer thread 1's stack]
)

```c
void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;

    printf("[%ld]:  %s (cnt=%d)\n",
          myid, ptr[myid], ++cnt);
    return NULL;
}
```

Local static var: 1 instance (`cnt` [data])

Shared Variable Analysis



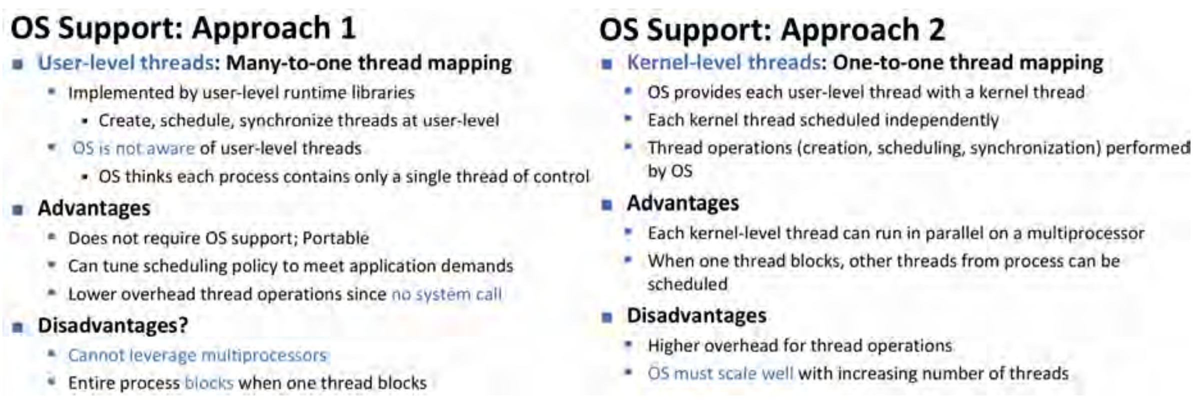
■ Which variables are shared?

Variable instance	Referenced by main thread?	Referenced by peer thread 0?	Referenced by peer thread 1?
`ptr`	yes	yes	yes
`cnt`	no	yes	yes
`i (main)`	yes	no	no
`msgs (main)`	yes	yes	yes
`myid (t0)`	no	yes	no
`myid (t1)`	no	no	yes

```c
char **ptr;  /* global var */
int main(int main, char *argv[]) {
  long i; pthread_t tid;
  char *msgs[2] = {"Hello from foo",
                   "Hello from bar" };
    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
            NULL, thread,(void *)i);
    Pthread_exit(NULL);}
```

```c
void *thread(void *vargp)
{
  long myid = (long)vargp;
  static int cnt = 0;

  printf("[%ld]:  %s (cnt=%d)\n",
         myid, ptr[myid], ++cnt);
  return NULL;
}
```

线程实现的两种方式: (我们默认采用第二种内核级别的线程)



OS Support: Approach 1

- User-level threads: Many-to-one thread mapping
  - Implemented by user-level runtime libraries
    - Create, schedule, synchronize threads at user-level
  - OS is not aware of user-level threads
    - OS thinks each process contains only a single thread of control
- Advantages
  - Does not require OS support; Portable
  - Can tune scheduling policy to meet application demands
  - Lower overhead thread operations since no system call
- Disadvantages?
  - Cannot leverage multiprocessors
  - Entire process blocks when one thread blocks

OS Support: Approach 2

- Kernel-level threads: One-to-one thread mapping
  - OS provides each user-level thread with a kernel thread
  - Each kernel thread scheduled independently
  - Thread operations (creation, scheduling, synchronization) performed by OS
- Advantages
  - Each kernel-level thread can run in parallel on a multiprocessor
  - When one thread blocks, other threads from process can be scheduled
- Disadvantages
  - Higher overhead for thread operations
  - OS must scale well with increasing number of threads

3.1.2 线程创建

我们考虑 t0.c 的例子 (OSTEP 26.2)

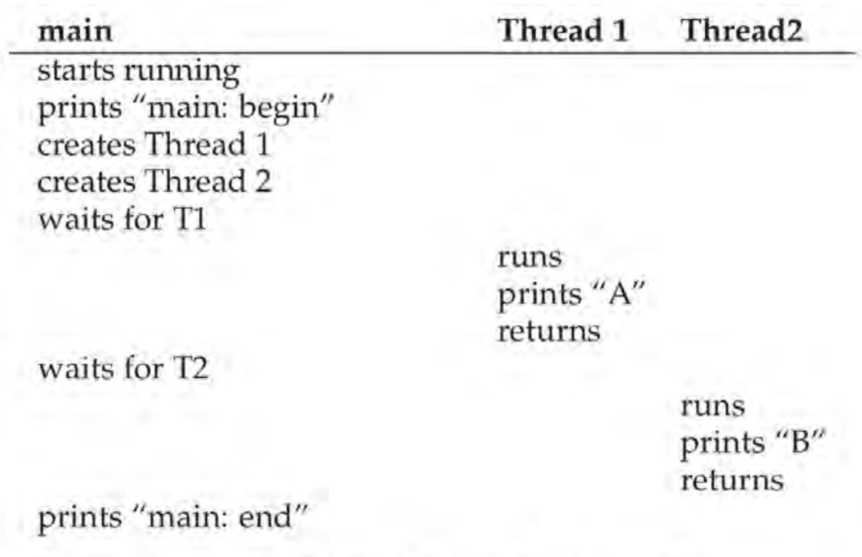
```c
#include <stdio.h>              // Standard I/O library
#include <assert.h>            // Assertion library for debugging
#include <pthread.h>           // POSIX threads library

// Thread function that prints the provided argument
void *mythread(void *arg)
{
    printf("%s\n", (char *) arg);  // Cast argument to string and print
    return NULL;                     // Return NULL to indicate completion
}

// Main function - entry point of the program
int main(int argc, char *argv[])
{
    pthread_t p1, p2;              // Declare thread identifiers
    int rc;                        // Variable for return code
    printf("main: begin\n");      // Indicate the start of main execution

    // Create two threads, each running 'mythread' with different arguments
    rc = pthread_create(&p1, NULL, mythread, "A"); assert (rc == 0);
    rc = pthread_create(&p2, NULL, mythread, "B"); assert (rc == 0);

    // Join waits for the threads to finish execution
    rc = pthread_join(p1, NULL); assert (rc == 0);      // Wait for thread 1 to
finish
    rc = pthread_join(p2, NULL); assert (rc == 0);      // Wait for thread 2 to
finish

    printf("main: end\n");        // Indicate the end of main execution
    return 0;                     // Return 0 to indicate successful completion
}
```

- `mythread` 是线程执行的函数，它接收一个 `void *` 类型的参数，强制转换为字符串并打印出来. 返回 `NULL` (即值为 0 的宏) 表示线程正常结束.
- `pthread_t p1, p2;` 声明了两个 `pthread_t` 变量 (线程标识符)，分别表示两个线程.
- 使用 `pthread_create` 函数创建两个线程，均运行 `mythread` 函数，分别传入参数 `"A"` 和 `"B"` 使用 `assert` 检查 `pthread_create` 的返回值，确保线程创建成功. 如果不成功，程序将终止.
- 使用 `pthread_join` 阻塞主线程，等待两个线程的完成.
  再一次使用 `assert` 检查返回值，确保线程成功结束.

输出结果如下:

```
Linux:~/Desktop/OS/OSTEP/03$ gcc -o t0 t0.c -Wall
Linux:~/Desktop/OS/OSTEP/03$ ./t0
main: begin
A
B
main: end
Linux:~/Desktop/OS/OSTEP/03$ ./t0
main: begin
B
A
main: end
Linux:~/Desktop/OS/OSTEP/03$
```

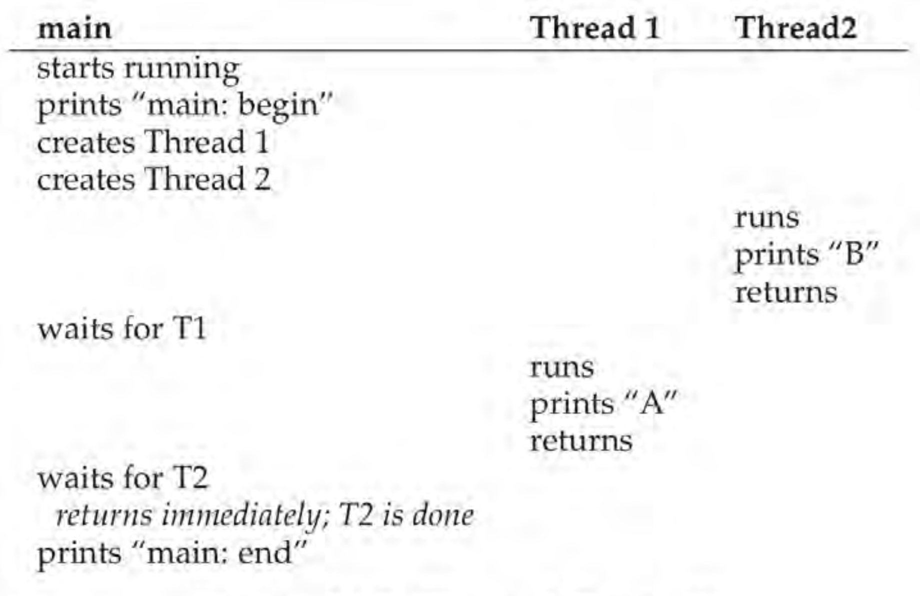
程序的执行流程可能如下:



main	Thread 1	Thread2
starts running		
prints “main: begin”		
creates Thread 1		
creates Thread 2		
waits for T1		
	runs	
	prints “A”	
	returns	
waits for T2		
		runs
		prints “B”
		returns
prints “main: end”		

Figure 26.3: **Thread Trace (1)**

上述执行流程不是唯一的可能.
实际上，给定一系列指令，有很多可能的执行顺序，这取决于调度程序决定在给定时刻运行哪个线程.
例如创建线程 1 后，它可能在创建线程 2 之前立即运行.
我们甚至可以在 `"A"` 之前看到 `"B"`，即使线程 1 的创建要先于线程 2
因此我们没有理由认为先创建的线程一定会先运行.



main	Thread 1	Thread2
starts running		
prints “main: begin”		
creates Thread 1		
creates Thread 2		
		runs
		prints “B”
		returns
waits for T1		
	runs	
	prints “A”	
	returns	
waits for T2		
returns immediately; T2 is done		
prints “main: end”		

Figure 26.5: **Thread Trace (3)**

线程创建有点像函数调用，但区别在于:
它并不像函数那样先执行然后返回给调用者，而是为被调用的例程创建一个新的执行线程，它独立于调用者运行.

3.1.3 核心问题

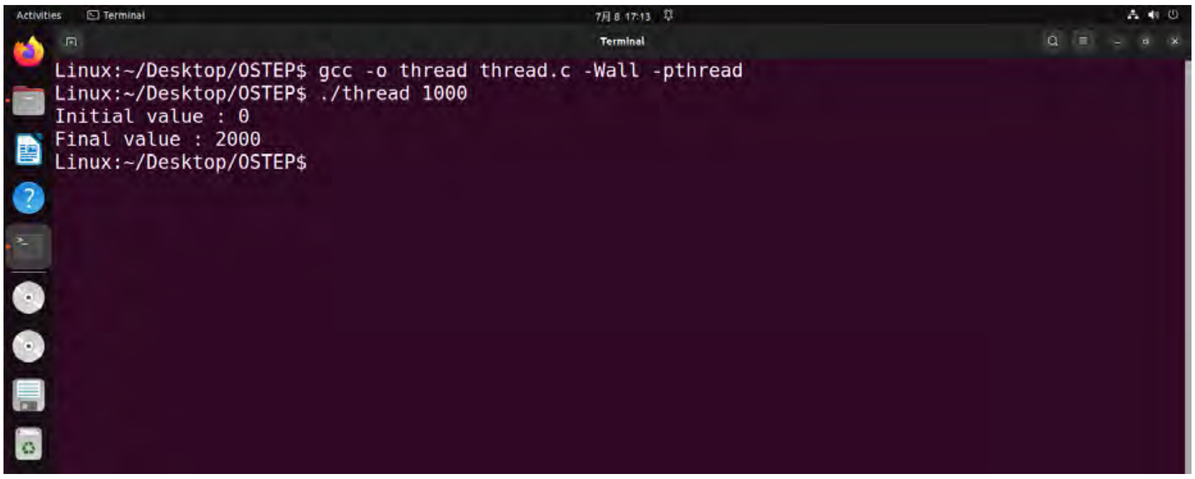
我们来看一个多线程程序 thread.c 的例子:

```
// code for thread.c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
```

```c
// The header file pthread.h includes 'pthread_t', 'pthread_create' and
'pthread_join'
// pthread_t is a data type in C that is used to represent a POSIX thread
identifier,
// which is a standard for creating and managing threads in a multi-threaded
program.
#include "common.h"

volatile int counter = 0;
// The 'volatile' keyword is used to indicate to the compiler
// that the variable may be changed by external factors not known to the
compiler,
// so it should not optimize away reads and writes to it.

int loops;
// Declare a global variable 'loops' without initialization,
// which will be set based on the command-line argument.

// Define a function 'worker' that will be executed by the threads.
void *worker(void *arg)
{
    int i;
    for (i = 0; i < loops; i++)
        counter++;
    return NULL;
}

int main(int argc, char *argv[])
{
    // Check if there is exactly one command-line argument provided
    if (argc != 2)
    {
        fprintf(stderr, "usage: threads <value>\n");
        exit(1);
    }

    // Parse the command-line argument as an integer
    // and store it in the 'loops' variable.
    loops = atoi(argv[1]);

    // Create two pthreads ('p1' and 'p2')
    // and execute the 'worker' function in each of them.
    pthread_t p1, p2;

    // Print the initial value of 'counter'
    printf("Initial value : %d\n", counter);

    pthread_create(&p1, NULL, worker, NULL);
    pthread_create(&p2, NULL, worker, NULL);

    // Wait for both threads to complete using 'pthread_join'.
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);

    // Print the final value of 'counter'.
    printf("Final value : %d\n", counter);
    return 0;
}
```

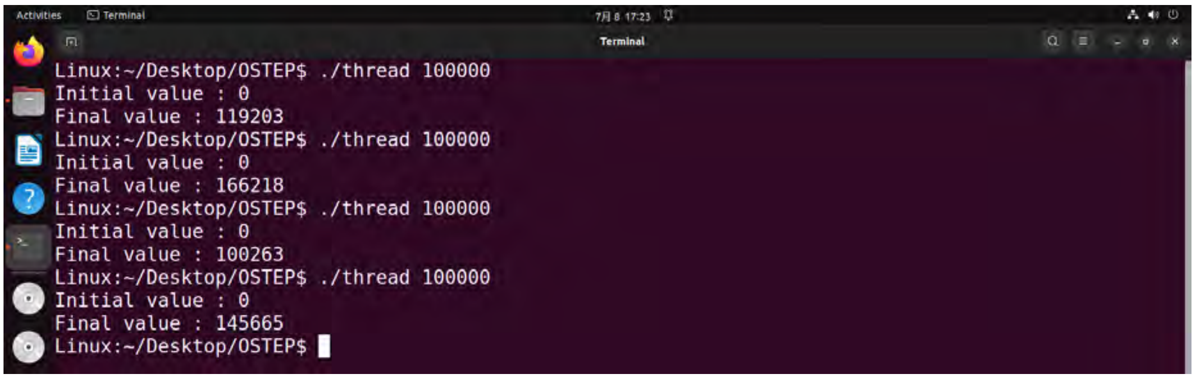
主函数利用 pthread_create() 创建了两个**线程** (thread).
我们可以将线程看作与其他函数在同一内存空间中运行的函数,
并且每次都有多个线程处于活动状态.
在这个例子中, 每个线程开始在一个名为 worker() 的函数中运行,
在该函数中, 它只是递增一个计数器, 循环 loops 次.
(loops 的值决定了两个 worker 各自在循环中递增共享计数器的次数)

我们将变量 loops 设为 1000, 得到的输出结果如下:



```
Linux:~/Desktop/OSTEP$ gcc -o thread thread.c -Wall -pthread
Linux:~/Desktop/OSTEP$ ./thread 1000
Initial value : 0
Final value : 2000
Linux:~/Desktop/OSTEP$
```

我们很容易想当然地认为, 当 loops 的输入值设为 N 时, 程序的最终输出就是 $2N$.
但事实证明, 事情并不是那么简单.
让我们将 loops 设置为更大的值, 看看会发生什么:



```
Linux:~/Desktop/OSTEP$ ./thread 100000
Initial value : 0
Final value : 119203
Linux:~/Desktop/OSTEP$ ./thread 100000
Initial value : 0
Final value : 166218
Linux:~/Desktop/OSTEP$ ./thread 100000
Initial value : 0
Final value : 100263
Linux:~/Desktop/OSTEP$ ./thread 100000
Initial value : 0
Final value : 145665
Linux:~/Desktop/OSTEP$
```

奇怪的是, 当我们将 loops 的值设为 100000 时,
并不一定能得到预期的输出 200000, 而且每次运行的输出也不一定相同!

事实证明, 上述奇怪的结果与指令如何执行有关.
程序每次只执行一条指令,
但上述程序的关键部分 (worker() 中递增共享计数器的部分) 需要 3 条指令:

- 一条将计数器的值从内存加载到寄存器;
- 一条将其递增;
- 一条将其保存回内存;

而这 3 条指令并不是以**原子方式** (atomically) 执行的 (即不是作为一个不可拆分的单元执行的).
假设某一线程这 3 条指令还未执行完时, 进程的时间片恰好用完了 (或者两个线程相互冲突),
那么会造成不确定的 bug (很难 debug, 可能 99% 的情况是对的, 但 1% 的情况是错误的)
这就像一个人分别通过 ATM 机和网银同时向同一银行账户存 1 块钱, 本来能增长 2 块钱, 结果只增加了 1 块钱.
上述例子展示了**多线程共享写**会出现的问题.

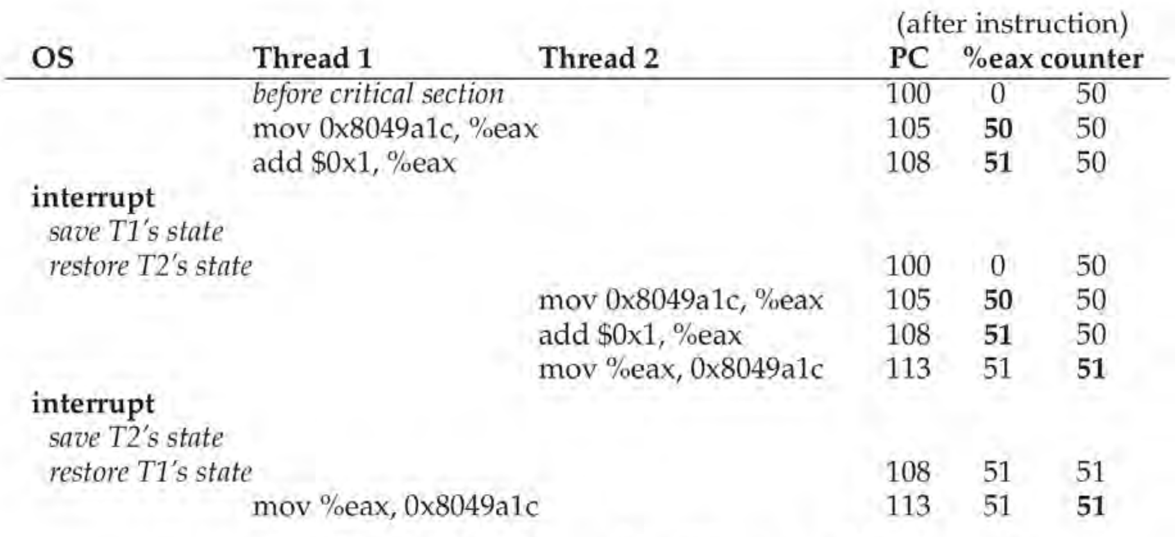
在上述例子里，我们只是想给 `counter` 加上一个数字 1
在 x86 中执行该操作的汇编指令可能长这样:

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

它将 `counter` (假定位于地址 0x8049a1c) 的值 (假设为 `50`) 加载到寄存器 `eax` 中
因此线程 1 的 `eax = 50`，然后它向寄存器加 1，得到 `eax = 51`
假设此时突然发生时钟中断，操作系统会将线程 1 的状态 (程序计数器、寄存器，包括 `eax` 等) 保存到线程的 TCB

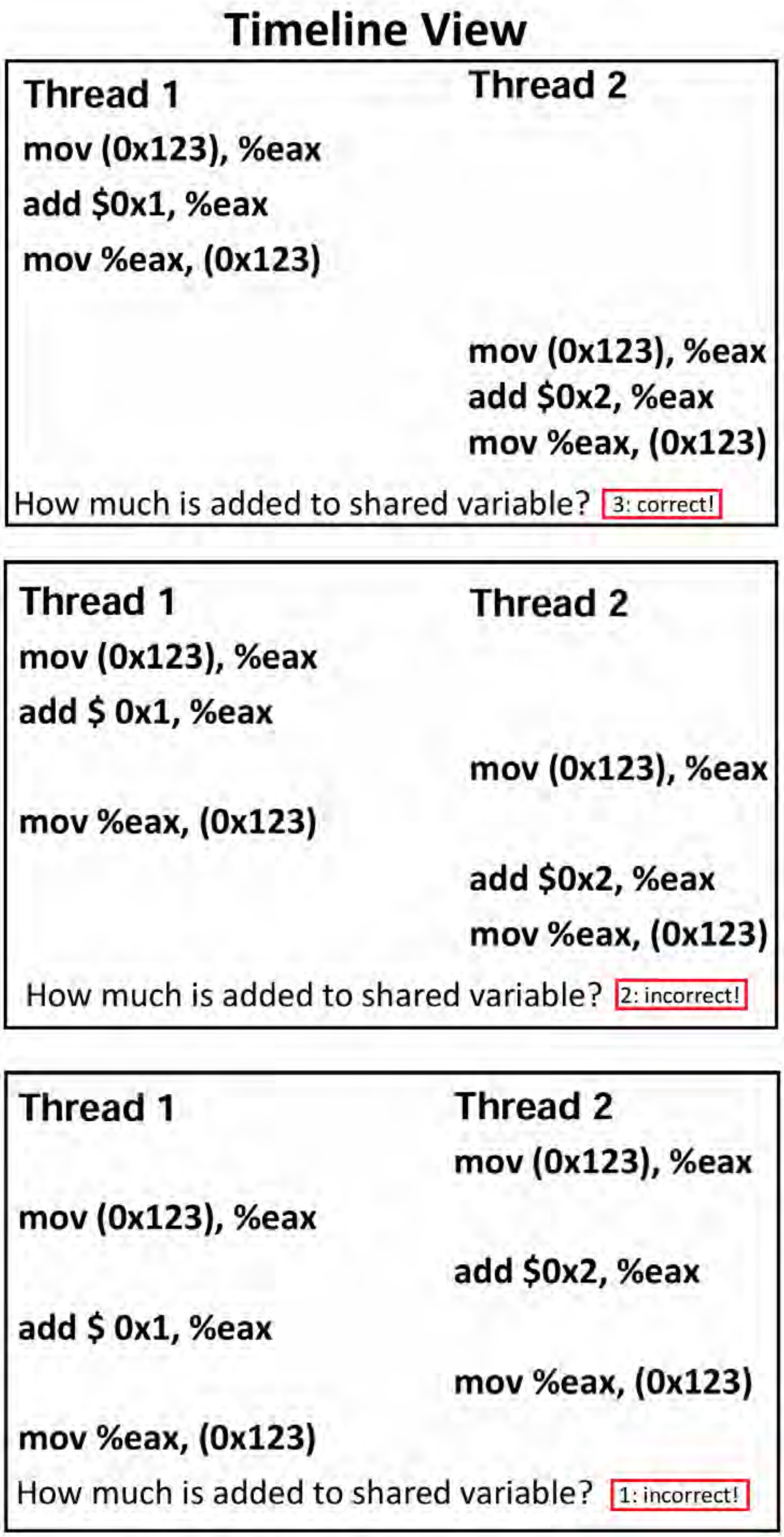
现在更糟的事发生了: 线程 2 被选中运行，并进入同一段代码.
它也执行了第一条指令，获取计数器的值并将其放入其 `eax` 中
(值得注意的是，运行时每个线程都有自己的专用寄存器，上下文切换代码将寄存器虚拟化，保存并恢复它们的值)
此时 `counter` 的值仍为 `50`，因此线程 2 的 `eax = 50`
假设线程 2 执行接下来的两条指令，将 `eax` 递增 1 (因此 `eax = 51`)
然后将 `eax` 的内容保存到 `counter` (地址 `0x8049a1c`) 中.
因此全局变量 `counter` 现在的值是 `51`.

最后又发生一次上下文切换，线程 1 恢复运行.
注意到它已经执行过 `mov` 和 `add` 指令，得到 `eax = 51`，现在准备执行最后一条 `mov` 指令.
这将 `eax` 的值 `51` 保存到内存，`counter` 再次被设置为 `51`.
换言之，增加 `counter` 的代码被执行两次，初始值为 `50`，理论结果是 `52`，但实际结果为 `51`



			(after instruction)		
OS	**Thread 1**	**Thread 2**	**PC**	**%eax**	**counter**
	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	51
	mov %eax, 0x8049a1c		113	51	**51**

Figure 26.7: **The Problem: Up Close and Personal**



Timeline View

```
Thread 1                      Thread 2
mov (0x123), %eax
add $0x1, %eax
mov %eax, (0x123)

                              mov (0x123), %eax
                              add $0x2, %eax
                              mov %eax, (0x123)
```

How much is added to shared variable? 3: correct!

```
Thread 1                      Thread 2
mov (0x123), %eax
add $ 0x1, %eax
                              mov (0x123), %eax
mov %eax, (0x123)
                              add $0x2, %eax
                              mov %eax, (0x123)
```

How much is added to shared variable? 2: incorrect!

```
Thread 1                      Thread 2
                              mov (0x123), %eax
mov (0x123), %eax
                              add $0x2, %eax
add $ 0x1, %eax
                              mov %eax, (0x123)
mov %eax, (0x123)
```

How much is added to shared variable? 1: incorrect!

临界区更形象的展示:

`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;
}
```

```
linux> ./badcnt 10000
OK cnt=20000
linux> ./badcnt 10000
BOOM! cnt=13051
linux>
```

`cnt` should equal 20,000.

What went wrong?

Assembly Code for Counter Loop

C code for counter loop in thread i

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

Asm code for thread i

```
    movq  (%rdi), %rcx
    testq %rcx,%rcx
    jle   .L2
    movl  $0, %eax
.L3:
    movq  cnt(%rip),%rdx
    addq  $1, %rdx
    movq  %rdx, cnt(%rip)
    addq  $1, %rax
    cmpq  %rcx, %rax
    jne   .L3
.L2:
```

H_i : Head

L_i : Load cnt

U_i : Update cnt

S_i : Store cnt

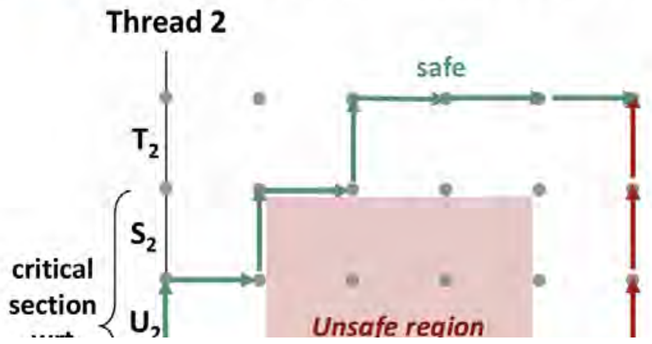
T_i : Tail

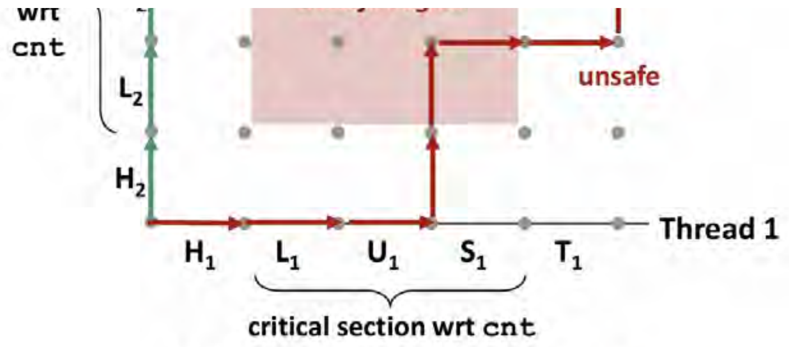
Variable	main	thread1	thread2
cnt	yes*	yes	yes
niters.m	yes	no	no
tid1.m	yes	no	no
i.1	no	yes	no
i.2	no	no	yes
niters.1	no	yes	no
niters.2	no	no	yes

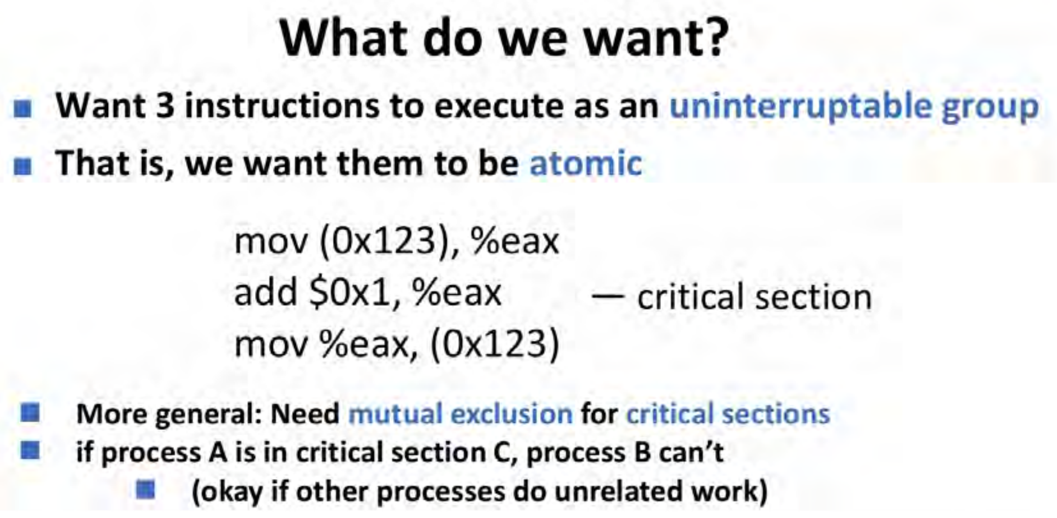
Critical Sections and Unsafe Regions

Def: A trajectory is *safe* if it does not enter any unsafe region

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







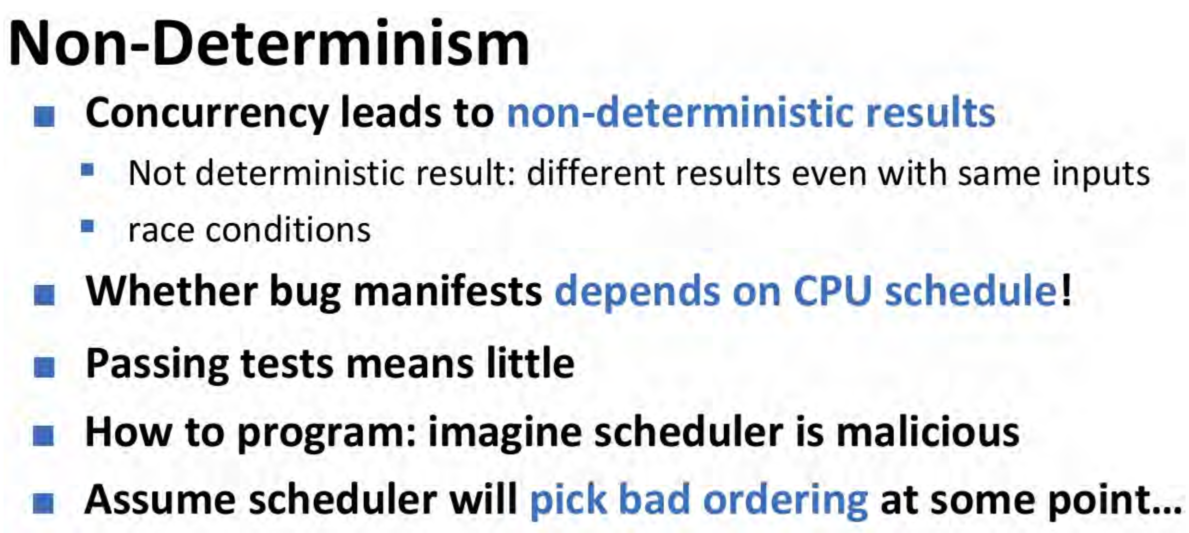
What do we want?

- Want 3 instructions to execute as an uninterruptable group
- That is, we want them to be atomic

```
mov (0x123), %eax
add $0x1, %eax          — critical section
mov %eax, (0x123)
```

- More general: Need mutual exclusion for critical sections
- if process A is in critical section C, process B can't
  - (okay if other processes do unrelated work)

由于执行这段关键代码的多个线程可能导致竞争状态，因此我们将此段代码称为**临界区** (critical section)
多个执行线程同时进入临界区并试图更新共享的数据结构情况称为**竞态条件** (race condition)
由于运气不好 (即在执行关键代码的过程中发生了上下文切换)，我们得到了错误的结果.
事实上每次运行都可能得到不同的结果，因此我们称这个结果是**不确定的** (indeterminate).



Non-Determinism

- Concurrency leads to non-deterministic results
  - Not deterministic result: different results even with same inputs
  - race conditions
- Whether bug manifests depends on CPU schedule!
- Passing tests means little
- How to program: imagine scheduler is malicious
- Assume scheduler will pick bad ordering at some point...

临界区是访问共享变量 (或更一般地说，共享资源) 的代码片段，一定不能由多个线程同时执行.
我们希望上述关键代码具有**互斥性** (mutual exclusion)
以保证只要有一个线程在临界区内执行，其他线程都将被阻止进入临界区.

我们要做的是要求硬件提供一些有用的指令，称为**同步原语** (synchronization primitive)
通过使用这些硬件同步原语，加上操作系统的一些帮助，我们将能够构建多线程代码，
以同步和受控的方式访问临界区，从而可靠地产生正确的结果.

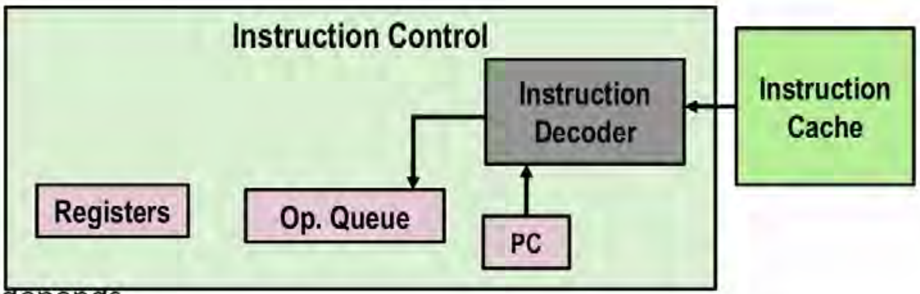
单个汇编语句并不是同步原语:

Single Instruction?

- **INC (0x123)**

Increase balance by one. (尽管只有一条汇编，但 CPU 可能分步执行, 因此不能解决原子性问题)

- **Can inc instruction avoids races?**



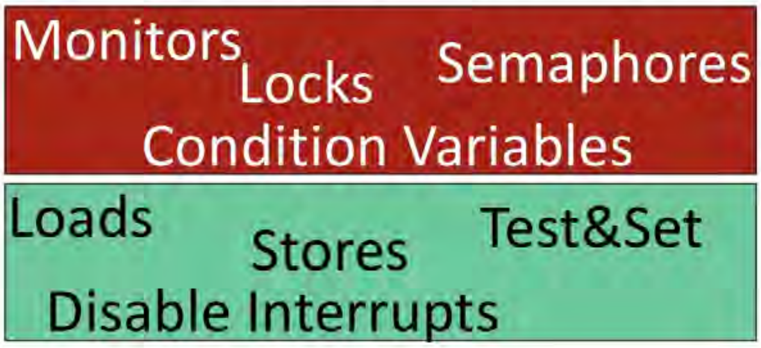
  - It depends.
  - Depends on if the CPU ISA treats single instruction atomic.
- **Lock prefix may help**
  - For x86, adding the lock prefix making sure the atomicity of the instruction.

Correct Concurrency is difficult

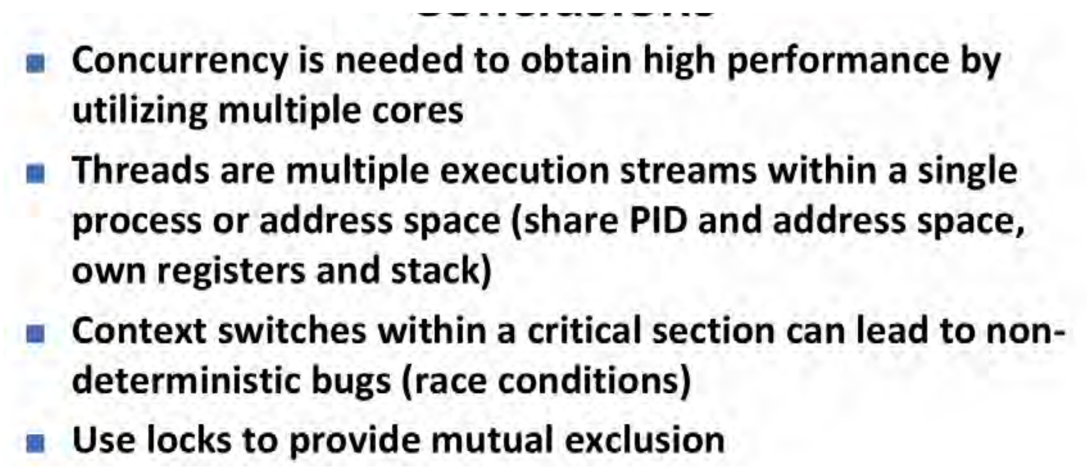
- **Compiler Optimization**
  - Compiler assumes nobody else is modifying memory
  - It may combine, split memory accesses to the same address
  - You may need "volatile" to direct the compiler to not think so
- **Compiler Re-ordering**
  - Compiler may reorder memory accesses to different addresses
  - You may need compiler barrier to prevent compiler reordering
    任何的重排序不会越过 barrier
- **CPU reordering**
  - In runtime, CPU may reorder memory accesses
  - You may need memory fence to restrict re-ordering
    赋值语句的重排序不会越过 fence

Synchronization

- **Build higher-level synchronization primitives in OS**
  - Operations that ensure correct ordering of instructions across threads
- **Motivation: Build them once and get them right**



Conclusions


- Concurrency is needed to obtain high performance by utilizing multiple cores
- Threads are multiple execution streams within a single process or address space (share PID and address space, own registers and stack)
- Context switches within a critical section can lead to non-deterministic bugs (race conditions)
- Use locks to provide mutual exclusion

3.2 线程 API

3.2.1 线程创建

编写多线程程序的第一步就是创建新线程
我们可通过 POSIX 线程 (pthread) 库的 `pthread_create` 函数做到.
其函数原型为:

```
#include <pthread.h>

int pthread_create(
    pthread_t *thread,
    const pthread_attr_t *attr,
    void *(*start_routine)(void*),
    void *arg
);
```

参数说明:

- `pthread_t *thread` 是一个指向 `pthread_t` 类型的指针，用于返回新创建线程的标识符.
  `pthread_t` 是一个无符号整数类型，代表线程的唯一标识.
- `const pthread_attr_t *attr` 是一个指向 `pthread_attr_t` 类型的指针，用于指定新线程的属性.
  如果设置为 `NULL`，则线程将使用默认属性 (诸如栈大小、线程调度优先级等)
- `void *(*start_routine)(void*)` 是一个指向函数的指针，指定新线程要执行的函数 `start_routine`
  该函数必须接受一个 `void*` 类型的参数，并返回 `void*`
  这允许我们传入任何类型的参数，返回任何类型的结果.
- `void *arg` 是传递给 `start_routine` 的参数，类型为 `void*`
  若有多个参数，则我们可以将所需的参数打包成一个我们自己定义的类型 (下例中的 `myarg_t`)
  该线程被创建后便可根据需要将参数解包.

  ```
  #include <stdio.h>
  #include <stdlib.h>
  #include <pthread.h>    // Include pthread header for thread functions

  // Define a structure to hold arguments for the thread function
  typedef struct myarg_t {
  ```

```
    int a;
    int b;
} myarg_t;

// Thread function to be executed by the new thread
void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;  // Cast the argument back to myarg_t
pointer
    printf("%d %d\n", m->a, m->b); // Print the values of a and b
    return NULL;                    // Return NULL to indicate completion
}

int main(int argc, char *argv[]) {
    pthread_t p;          // Thread identifier
    int rc;               // Return code for thread functions

    // Initialize the arguments to be passed to the thread
    myarg_t args;
    args.a = 10;
    args.b = 20;

    // Create a new thread that runs the mythread function
    rc = pthread_create(&p, NULL, mythread, (void *)&args);
    if (rc) {  // Check if pthread_create was successful
        fprintf(stderr, "Error: pthread_create failed with code %d\n", rc);
        exit(EXIT_FAILURE); // Exit if thread creation fails
    }

    // Wait for the created thread to finish
    rc = pthread_join(p, NULL);
    if (rc) {  // Check if pthread_join was successful
        fprintf(stderr, "Error: pthread_join failed with code %d\n", rc);
        exit(EXIT_FAILURE); // Exit if joining fails
    }

    return EXIT_SUCCESS; // Indicate successful execution
}
```

当我们成功创建了一个线程之后，我们就拥有了另一个执行实体.
它有自己的调用栈，与程序中所有当前存在的线程在相同的地址空间内运行.

3.2.2 等待线程完成

`pthread_join` 是 POSIX 线程 (pthread) 库中的一个重要函数，用于等待线程的结束.
它使得主线程能够等待其他线程完成执行，并获取这些线程的返回状态.
这是多线程编程中保证线程同步和资源管理的关键部分.
其函数原型为:

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **value_ptr);
```

参数说明:

- `pthread_t thread` 是线程标识符，指向要等待的线程.

- `void **value_ptr` 是一个**指向指针的指针**，用于获取被等待线程的返回值.
  如果不需要返回值，可以将其设置为 `NULL`
  通过将返回值存储在指向指针的指针中，主线程可以在调用 `pthread_join` 后访问和使用这个值.

  值得注意的是，返回一个指向局部变量 (在栈上分配的变量) 的指针将导致严重的错误.
  ① 栈内存是由编译器自动管理的，通常用于局部变量和函数调用.
  当函数返回时，栈上的所有局部变量 (包括指向这些变量的指针) 都被释放.
  这意味着它们不再有效，访问这些变量将导致未定义行为.
  ② 堆内存由程序员手动管理，使用 `malloc`、`calloc` 或 `realloc` 等函数进行分配.
  在使用后，必须手动释放 (`free`) 分配的内存.
  返回指向堆内存的指针是安全的，因为它在函数返回后仍然有效.

我们来看一个例子:
首先创建单个线程，并通过 `myarg_t` 结构打包并传递参数，同时使用 `myret_t` 结构打包返回值.
当线程完成运行时，主线程已经在 `pthread_join()` 函数内等待了.
我们可以访问线程返回的值，即解包 `myret_t` 结构中的内容.

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>    // Include pthread header for thread functions
#include <assert.h>     // Include assert for debugging

// Define a structure to hold input arguments for the thread
typedef struct myarg_t {
    int a; // First integer
    int b; // Second integer
} myarg_t;

// Define a structure to hold the return values from the thread
typedef struct myret_t {
    int x; // First return value
    int y; // Second return value
} myret_t;

// Thread function that will be executed by the created thread
void *mythread(void *arg) {
    myarg_t *m = (myarg_t *)arg;  // Cast the argument to myarg_t pointer
    printf("%d %d\n", m->a, m->b); // Print the values of a and b

    // Allocate memory for the return structure
    myret_t *r = (myret_t *)malloc(sizeof(myret_t));
    assert(r != NULL); // Check for successful memory allocation

    r->x = 1; // Set return value x
    r->y = 2; // Set return value y
    return (void *)r; // Return the pointer to the allocated memory
}

int main(int argc, char *argv[]) {
    int rc;                      // Variable to store return codes
    pthread_t p;                // Thread identifier
    myret_t *m;                 // Pointer to hold return values from the thread

    myarg_t args;               // Structure to hold arguments for the thread
    args.a = 10;                // Initialize first argument
    args.b = 20;                // Initialize second argument
    // Create a new thread
    rc = pthread_create(&p, NULL, mythread, (void *)&args);
    assert(rc == 0); // Check for successful thread creation

    // Wait for the thread to finish and get the return value
    rc = pthread_join(p, (void **)&m);
    assert(rc == 0); // Check for successful thread join

    // Print the returned values from the thread
    printf("returned %d %d\n", m->x, m->y);

    free(m); // Free the allocated memory for the return structure
    return 0; // Return success
}
```

下面是一个错误的例子:

```
void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg; // Cast the argument to myarg_t pointer
    printf("%d %d\n", m->a, m->b); // Print the values of a and b

    myret_t r; // ALLOCATED ON STACK: BAD!
    // The variable 'r' is allocated on the stack, which means it will only be
valid within this function.

    r.x = 1; // Assign value to x
    r.y = 2; // Assign value to y

    // Return a pointer to a local variable (r)
    return (void *) &r;
    // This is dangerous because 'r' will be deallocated when the function
returns.
    // Accessing this pointer outside this function will lead to undefined
behavior.
}
```

值得注意的是，在上述例子中我们使用 `pthread_create` 创建线程，然后立即调用 `pthread_join`
这就相当于在主线程中等待新线程完成，实际上与普通的**过程调用** (procedure call) 非常相似.
在这种情况下，创建线程似乎没有太大意义，因为我们只是等待它完成，而没有实现并行执行的优势.

多线程的主要目的是为了实现并行处理，充分利用多核处理器的能力.
通过同时运行多个线程，可以提高程序的性能和响应性.
例如在进行大量计算或 I/O 操作时，可以将任务分配给多个线程，从而缩短整体执行时间.
在设计多线程程序时，应考虑如何有效地利用线程.
创建多个线程并让它们并行执行各自的任务，而不是在创建后立即等待每个线程完成，是实现多线程优势的关键.

3.2.3 锁

POSIX 线程 (pthread) 库提供的最重要的功能是通过锁实现线程间的互斥访问.
互斥 (mutual exclusion) 是一种同步机制，旨在防止多个线程同时访问共享资源 (如变量或数据结构)
如果没有适当的同步，多个线程同时读写共享资源可能会导致数据不一致或损坏
临界区 (critical section) 是指一段在任意时刻只能由一个线程执行的代码.

```c
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- `pthread_mutex_lock` 函数用于获取一个**互斥锁** (`mutex`)
  如果锁当前未被其他线程占用，调用该函数的线程将成功获取锁 (返回 `0`)，并可以进入临界区.
  如果锁已被其他线程占用，调用该函数的线程将被阻塞，直到锁变得可用为止.
  (将返回一个错误码，指示失败的原因，例如锁已被占用)
- `pthread_mutex_unlock` 函数用于释放先前获取的互斥锁.
  如果调用该函数时，线程持有该锁，它将解锁 (返回 `0`)，使得其他线程可以获取该锁并访问临界区.
  如果一个线程尝试解锁一个未被它自身锁定的互斥锁，将导致未定义的行为.
  (将返回一个错误码，指示失败的原因，例如锁未被当前线程持有)

所有锁必须正确初始化，以确保它们具有正确的值，并在锁和解锁被调用时按照需要工作.
对于 POSIX 线程，有两种方法来初始化锁:

- ① `pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;`
  当锁的生命周期与它的作用域相同，并且在声明时已经知道要使用这个锁时，这种静态初始化方法是非常方便的.
  这样做会将锁设置为默认值，从而使锁可用.
- ② `int rc = pthread_mutex_init(&lock, NULL);`
  这是初始化的动态方法，也是我们通常使用的方法，它可以控制锁的属性和行为，提供更多灵活性.
  函数 `pthread_mutex_init` 第一个参数是锁本身的地址，而第二个参数是一组可选属性 (若为 `NULL` 则使用默认属性)

当不再需要锁时，应调用 `pthread_mutex_destroy` 来销毁它.
(销毁锁之前，应确保没有线程在持有锁，否则将导致未定义行为)

在获取锁和释放锁时一定要检查错误代码，例如使用如下的包装函数来代替对应的库函数:

```c
// Use this to keep your code clean but check for failures
// Only use if exiting program is OK upon failure
void Pthread_mutex_lock(pthread_mutex_t *mutex) {
  int rc = pthread_mutex_lock(mutex);
  assert(rc == 0);
}
```

其中 `assert` 函数只在调试模式下有效，在发布模式下即使出错也不会有任何行为.
若希望采用更复杂的错误处理机制 (例如重试机制或错误记录)，则应当使用更灵活的错误检查，例如:

```c
// Use this to keep your code clean but check for failures
// Only use if exiting program is OK upon failure
void Pthread_mutex_lock(pthread_mutex_t *mutex) {
  int rc = pthread_mutex_lock(mutex);
  if (rc != 0)
      fprintf(stderr, "Error locking mutex: %s\n", strerror(rc));
}
```

获取锁的 `timedlock` 版本允许线程在指定的超时时间内尝试获取锁，这在研究死锁时会很有用:
(在某些情况下，程序可能希望在一定的时间内尝试获取锁，而不是无限期等待)

```c
int pthread_mutex_timedlock(pthread_mutex_t *mutex, struct timespec
*abs_timeout);
```

参数说明:

- `pthread_mutex_t *mutex` 是指向要锁定的互斥锁的指针，该锁必须在调用前被初始化.
- `struct timespec *abs_timeout` 是指向一个 `timespec` 结构体的指针，代表设定的超时时间 `timespec` 结构体通常包含两个字段: 秒数部分 `tv_sec` 和纳秒部分 `tv_nsec`

当超时时间为 `0` 时，`pthread_mutex_timedlock` 函数可由 `pthread_mutex_trylock` 函数替代，其函数原型为:

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

我们考虑 PPT `lecture10-Locks.pptx` 上的例子:
主函数的代码为:

```
/* Global shared variable */
volatile long cnt = 0;
/* Counter: This variable keeps track of the count and is declared as volatile to
prevent optimization by the compiler that could lead to incorrect behavior in a
multithreaded environment. */

int main(int argc, char **argv)
{
    long niters; /* Variable to hold the number of iterations for the threads */
    pthread_t tid1, tid2; /* Thread identifiers for the two threads being
created */

    /* Check if the correct number of command-line arguments are provided */
    if (argc != 2)
    {
        fprintf(stderr, "Usage: %s <number_of_iterations>\n", argv[0]);
        /* Print usage message if incorrect arguments are given */
        exit(1); /* Exit the program with an error code */
    }

    niters = atoi(argv[1]); /* Convert the second command-line argument to a
long integer */

    /* Create the first thread, passing the address of niters as an argument */
    Pthread_create(&tid1, NULL, thread, &niters);
    /* Pthread_create is a wrapper around pthread_create for error checking */

    /* Create the second thread with the same argument */
    Pthread_create(&tid2, NULL, thread, &niters);
    /* Both threads will execute the same thread function */

    /* Wait for both threads to finish execution */
    Pthread_join(tid1, NULL); /* Wait for the first thread to complete */
    Pthread_join(tid2, NULL); /* Wait for the second thread to complete */

    /* Check the result of the counting operation */
    if (cnt != (2 * niters))
        /* Since both threads increment cnt by niters, the expected value should
be 2 * niters */
        printf("BOOM! cnt=%ld\n", cnt);
        /* Print an error message if the count is not as expected */
    else
        printf("OK cnt=%ld\n", cnt);
        /* Print a success message if the count is correct */

    exit(0); /* Exit the program successfully */
}
```

`thread` 函数的一个坏的实现如下:

```
/* Thread routine */
void *thread(void *vargp) /* Function executed by each thread */
{
    long i, niters = *((long *)vargp);
    /* Dereference the argument to get the number of iterations */

    for (i = 0; i < niters; i++) /* Loop from 0 to niters - 1 */
        cnt++; /* Increment the shared counter; this is not thread-safe! */

    return NULL; /* Return NULL to indicate successful completion of the thread
*/
}
```

我们需要引入锁来保证临界区 `cnt++` 在任意时刻至多有一个线程进入:

```
/* Global shared variable */
volatile long cnt = 0; /* Counter: */
pthread_mutex_t mutex; /* Mutex for synchronizing access to cnt */
pthread_mutex_init(&mutex, NULL); /* Initialize the mutex */
```

`thread` 函数的一个好的实现如下:

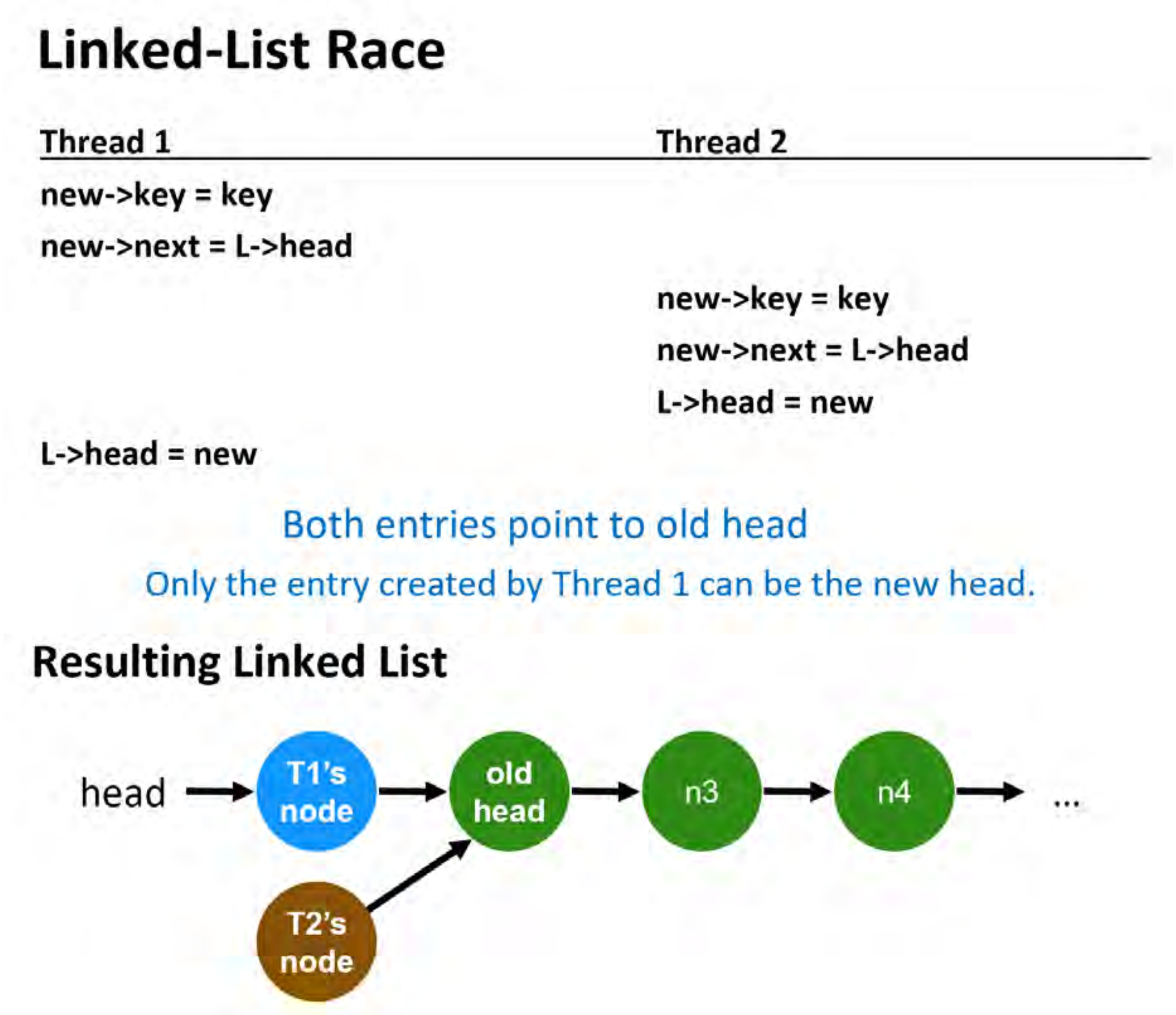
```
void *thread(void *vargp) {
    long i, niters = *((long *)vargp); /* Dereference the argument */

    for (i = 0; i < niters; i++)
    {
        pthread_mutex_lock(&mutex); /* Lock the mutex before accessing cnt */
        cnt++; /* Safely increment the shared counter */
        pthread_mutex_unlock(&mutex); /* Unlock the mutex after the increment */
    }

    return NULL; /* Return from the thread function */
}
```

其他共享数据结构的例子: 共享更新的列表
这是基本的列表实现:

```
// Define a structure for a node in the linked list
typedef struct __node_t {
    int key; // The data value stored in the node
    struct __node_t *next; // Pointer to the next node in the list
} node_t;

// Define a structure for the linked list itself
```

```
typedef struct __list_t {
    node_t *head; // Pointer to the head (first node) of the list
} list_t;

// Function to initialize a linked list
void List_Init(list_t *L) {
    L->head = NULL; // Set the head pointer to NULL, indicating the list is
empty
}

// Function to insert a new node with a given key at the beginning of the list
void List_Insert(list_t *L, int key) {
    // Allocate memory for a new node
    node_t *new = malloc(sizeof(node_t));
    assert(new); // Ensure that memory allocation was successful

    new->key = key; // Set the key of the new node
    new->next = L->head; // Set the new node's next pointer to the current head
    L->head = new; // Update the head pointer to point to the new node
}

// Function to look up a key in the linked list
int List_Lookup(list_t *L, int key) {
    node_t *tmp = L->head; // Start from the head of the list

    // Traverse the list until reaching the end
    while (tmp)
    {
        if (tmp->key == key) // Check if the current node's key matches the
target key
            return 1; // Return 1 if the key is found
        tmp = tmp->next; // Move to the next node
    }
    return 0; // Return 0 if the key is not found in the list
}
```

但上述列表实现可能出现问题:



Linked-List Race

Thread 1	Thread 2
new->key = key	
new->next = L->head	
	new->key = key
	new->next = L->head
	L->head = new
L->head = new	

Both entries point to old head

Only the entry created by Thread 1 can be the new head.

Resulting Linked List

我们应该这样修改 `__list_t` 结构体和 `List_Init` 函数:

```
// Define a structure for the linked list
typedef struct __list_t {
    node_t *head; // Pointer to the head (first node) of the list
    pthread_mutex_t lock; // Mutex for synchronizing access to the list
} list_t;

// Function to initialize a linked list
void List_Init(list_t *L) {
    L->head = NULL; // Set the head pointer to NULL, indicating the list is
empty
    pthread_mutex_init(&L->lock, NULL); // Initialize the mutex for the list
}
```

我们应该这样修改 `List_Insert` 函数:

```
// Function to insert a new node with a given key at the beginning of the list
void List_Insert(list_t *L, int key) {
    // Allocate memory for a new node
    node_t *new = malloc(sizeof(node_t));
    assert(new); // Ensure that memory allocation was successful

    new->key = key; // Set the key of the new node

    // Lock the mutex to ensure exclusive access to the list
    Pthread_mutex_lock(&L->lock);
    new->next = L->head; // Set the new node's next pointer to the current head
    L->head = new; // Update the head pointer to point to the new node

    // Unlock the mutex to allow other threads to access the list
    Pthread_mutex_unlock(&L->lock);
}
```

若没有 `List_Delete` 函数，则 `List_Lookup` 函数无需加锁.
若有 `List_Delete` 函数 (它需要加锁)，则 `List_Lookup` 函数需要加锁:

```
// Function to delete a node with a given key from the list
void List_Delete(list_t *L, int key) {
    // Lock the mutex to ensure exclusive access to the list
    pthread_mutex_lock(&L->lock);

    node_t *current = L->head; // Start from the head of the list
    node_t *previous = NULL; // Pointer to keep track of the previous node

    // Traverse the list to find the node to delete
    while (current != NULL) {
        if (current->key == key) { // Check if the current node has the key
            if (previous == NULL) {
                // The node to delete is the head
                L->head = current->next; // Update head to the next node
            } else {
                // Bypass the current node
                previous->next = current->next; // Link previous node to the
next node
            }
            free(current); // Free the memory of the deleted node
            pthread_mutex_unlock(&L->lock); // Unlock the mutex
            return; // Exit the function after deletion
        }
        // Move to the next node
        previous = current;
        current = current->next;
    }

    pthread_mutex_unlock(&L->lock); // Unlock the mutex if the key is not found
}

// Function to look up a key in the linked list
int List_Lookup(list_t *L, int key) {
    // Lock the mutex to ensure exclusive access to the list
    pthread_mutex_lock(&L->lock);

    node_t *tmp = L->head; // Start from the head of the list

    // Traverse the list until reaching the end
    while (tmp) {
        if (tmp->key == key) { // Check if the current node's key matches the
target key
            pthread_mutex_unlock(&L->lock); // Unlock the mutex before returning
            return 1; // Return 1 if the key is found
        }
        tmp = tmp->next; // Move to the next node
    }

    pthread_mutex_unlock(&L->lock); // Unlock the mutex if the key is not found
    return 0; // Return 0 if the key is not found in the list
}
```

3.2.4 条件变量

条件变量 (condition variable) 在多线程编程中非常有用，特别是当一个线程需要等待另一个线程完成某些操作后才能继续执行时.
它们提供了一种机制，使得线程之间能够以信号的方式进行通信和协调.
POSIX 线程 (pthread) 库中用于处理条件变量的关键函数如下:

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
```

功能说明:

- 当调用 `pthread_cond_wait` 时，线程会释放所持有的互斥锁，并进入等待状态，
  直到另一个线程通过调用 `pthread_cond_signal` 或 `pthread_cond_broadcast` 通知条件变量.
  一旦条件变量被信号通知，等待线程会重新获取互斥锁并继续执行，这确保了对共享数据的安全访问.
  参数 `pthread_cond_t *cond` 是指向条件变量的指针
  参数 `pthread_mutex_t *mutex` 是指向互斥锁的指针
  (等待调用除了使调用线程进入睡眠状态外，还会让调用者睡眠时释放锁)
  典型用法如下:

  ```
  // Initialize a mutex lock using the default initializer.
  // This lock will be used to protect shared resources.
  pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

  // Initialize a condition variable using the default initializer.
  // This condition variable will be used to signal between threads.
  pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

  // Acquire the mutex lock before checking or modifying shared resources.
  // This ensures that only one thread can access the critical section at a
  time.
  Pthread_mutex_lock(&lock);

  // Enter a loop to wait until the 'ready' condition is met.
  // If 'ready' is 0, the thread will wait on the condition variable.
  while (ready == 0)
      // Wait for a signal on the condition variable while releasing the lock.
      // The thread will block until another thread signals 'cond',
      // at which point it will re-acquire the lock and check 'ready' again.
      Pthread_cond_wait(&cond, &lock);

  // Once 'ready' is non-zero, release the mutex lock to allow other threads
  to access the shared resource.
  Pthread_mutex_unlock(&lock);
  ```

  等待线程在 `while` 循环中重新检查条件，而不是简单的 `if` 语句.
  在后续章节中研究条件变量时，我们会详细讨论这个问题
  但是使用 `while` 循环是一件简单而安全的事情 (尽管会增加一点开销)

- 当调用 `pthread_cond_signal` 时，会唤醒一个等待在条件变量 `pthread_cond_t *cond` 上的线程.
  如果没有线程在等待，则该调用没有效果.
  例如当一个线程更新共享数据并希望其他线程知道该数据已更改时，可以使用此函数通知等待的线程.
  典型用法如下:

  ```
  // Acquire the mutex lock before modifying shared resources.
  // This ensures that only this thread can access the critical section.
  Pthread_mutex_lock(&lock);

  // Set the 'ready' flag to indicate that the condition has been met.
  // This variable is shared between threads, so it must be modified while
  holding the lock.
  ready = 1;

  // Signal one thread waiting on the condition variable that the condition
  has changed.
  // This will wake up one of the threads that are blocked on
  Pthread_cond_wait.
  Pthread_cond_signal(&cond);

  // Release the mutex lock to allow other threads to acquire it and access
  shared resources.
  // It's important to unlock the mutex as soon as possible to avoid blocking
  other threads.
  Pthread_mutex_unlock(&lock);
  ```

构建多线程程序需要注意以下几点:

- **① 初始化锁和条件变量，时刻注意检查返回值**
- **② 保持简洁，尽可能减少线程交互:**
  复杂的线程交互容易产生缺陷，每次交互都应该想清楚，并用符合规范的方法来实现.
  (线程之间总是通过条件变量发送信号，切记不要用标记变量来同步)
- **③ 注意传给线程的参数和返回值:**
  切记不要传递在栈上分配的变量的引用.
  每一个线程都有自己的栈，因此线程局部变量应该是线程私有的，其他线程不应该访问.
  线程之间的共享数据要存放在堆或者其他全局可访问的位置.

3.3 锁

并发编程最基本问题是:
我们希望原子式执行一系列指令，但由于单处理器上的中断 (或者多个线程在多处理器上并发执行)，我们做不到.
而锁的引入直接解决这一问题: 程序员在源代码中加锁，放在临界区周围，保证临界区能够像单条原子指令一样执行.

3.3.1 基本思想

典型的更新共享变量: (当然也有对共享结构的复杂更新操作)

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

锁就是一个变量，因此我们需要声明一个某种类型的**锁变量** (lock variable)
这个锁变量保存了锁在某一时刻的状态:
要么是**可用的** (available)，表示没有线程持有锁，要么是**被占用的** (acquired)，表示有一个线程持有锁，正处于临界区.
我们也可以保存其他的信息，例如持有锁的线程，或请求获取锁的线程队列，但这些信息对锁的使用者是不可见的.

```
// Declare a globally allocated lock named 'mutex'.
// This lock will be used to synchronize access to shared resources across
threads.
lock_t mutex; // some globally-allocated lock 'mutex'

... // Additional code may be present here

// Acquire the lock before modifying the shared resource.
// This ensures that only this thread can access the critical section.
lock(&mutex);

// Perform the critical operation: increment the balance.
// This operation is now protected by the mutex, preventing race conditions.
balance = balance + 1;

// Release the lock after the critical operation is complete.
// This allows other threads to acquire the lock and access the shared resource.
unlock(&mutex);
```

`lock()` 和 `unlock()` 函数的语义很简单.

- 调用 `lock()` 尝试获取锁，如果没有其他线程持有锁 (即它是可用的)，则该线程会获得锁，进入临界区.
  这个线程被称为锁的持有者 (owner)
  此时如果另外一个线程对相同的锁变量调用 `lock()`，因为锁被另一线程持有，该调用不会返回.
  这样当持有锁的线程在临界区时，其他线程就无法进入临界区.
- 锁的持有者一旦调用 `unlock()`，锁就变成可用了.
  如果没有其他等待线程 (即没有其他线程调用过 `lock()` 并卡在那里)，锁的状态就变成可用了.
  如果有等待线程 (卡在 `lock()` 里)，则其中一个会获取该锁，进入临界区.

锁为程序员提供了最小程度的调度控制.
我们把线程视为程序员创建的实体，但是被操作系统调度，具体方式由操作系统选择.
锁让程序员获得一些控制权: 通过给临界区加锁，可以保证临界区内任意时刻至多有一个线程活跃.
锁将原本由操作系统调度的混乱状态变得更为可控.

POSIX 线程库 (pthread) 将锁称为**互斥量** (mutex)
当一个线程在临界区时，它能够阻止其他线程进入直到当前线程离开临界区.

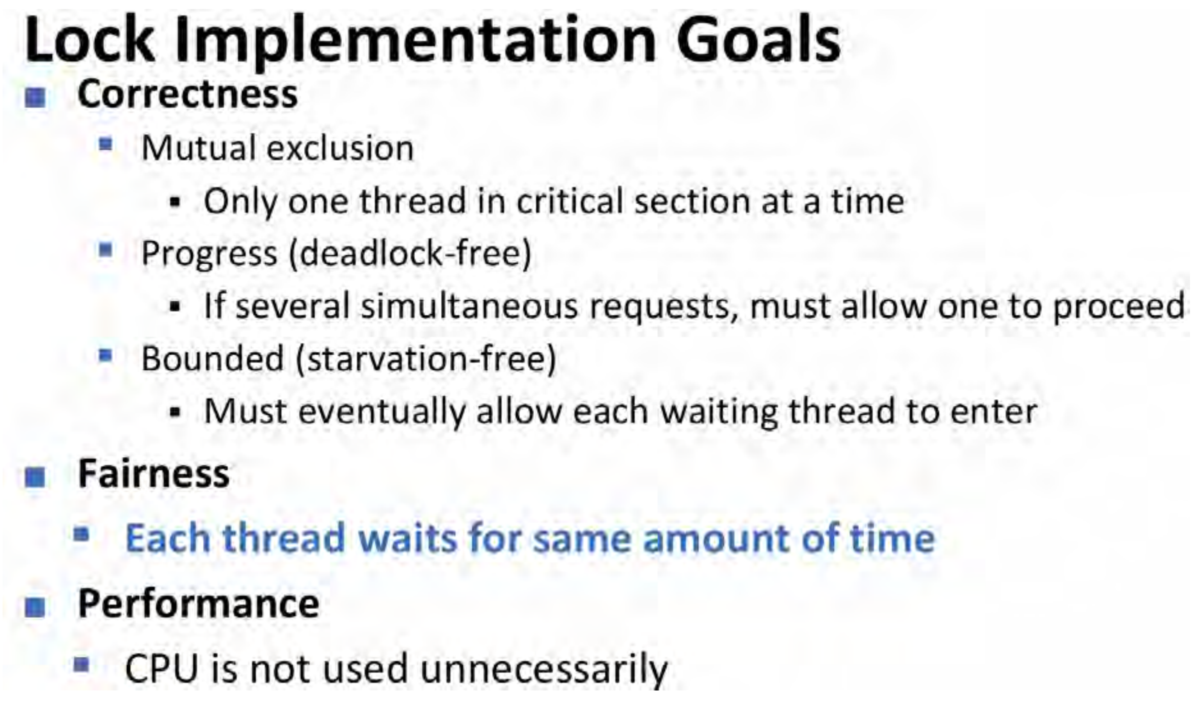
```
// Initialize a mutex lock using the default initializer.
// This mutex will be used to synchronize access to shared resources.
```

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

// Acquire the mutex lock before modifying the shared resource.
// This ensures that only this thread can access the critical section at this
time.
// The lock is wrapped in a custom function to handle potential errors.
Pthread_mutex_lock(&lock); // wrapper for pthread_mutex_lock()

// Perform the critical operation: increment the balance.
// This operation is now protected by the mutex, preventing race conditions.
// Only one thread can execute this section at a time, ensuring data integrity.
balance = balance + 1;

// Release the mutex lock after the critical operation is complete.
// This allows other threads to acquire the lock and access the shared resource.
Pthread_mutex_unlock(&lock);
```

我们会用不同的锁来保护不同的数据结构，从而允许更多的线程进入临界区.

在实现锁之前，我们首先明确锁的基本任务:

- ① **有效性:**
  提供互斥——当一个线程在临界区时，它能够阻止其他线程进入直到当前线程离开临界区.
  且不发生**死锁** (deadlock): 进展 (progress) 是无死锁系统的基本特性.
  死锁发生在两个或多个进程互相等待对方持有的资源，导致它们都无法继续.
  通过保证至少一个请求能够进行，系统可以避免这种情况.
- ② **公平性:**
  当锁可用时，每一个竞争线程应该有公平的机会抢到锁，不会出现某个竞争锁的线程会饿死的情况发生.
- ③ **高效性:**
  尽可能降低使用锁之后增加的时间开销 (例如 CPU 不能空转, spinning: `while(1);`).
  有几种场景需要考虑:
  一种是没有竞争的情况，即只有一个线程抢锁、释放锁的开支;
  另一种是一个 CPU 上多个线程竞争的情况;
  最后一种是多个 CPU 上多个线程竞争时的性能.


Lock Implementation Goals

- Correctness
  - Mutual exclusion
    - Only one thread in critical section at a time
  - Progress (deadlock-free)
    - If several simultaneous requests, must allow one to proceed
  - Bounded (starvation-free)
    - Must eventually allow each waiting thread to enter
- Fairness
  - Each thread waits for same amount of time
- Performance
  - CPU is not used unnecessarily

3.3.2 控制中断

最早的互斥实现方案就是在临界区关闭中断.
这个解决方案是为单处理器 (uniprocessor) 系统开发的，其代码如下:

```
// Function to acquire a lock by disabling interrupts.
// This prevents context switches and ensures that the critical section
// can be executed without interruption from other threads or processes.
void lock()
{
    DisableInterrupts(); // Disable interrupts to protect critical section.
}

// Function to release the lock by enabling interrupts.
// This allows other threads or processes to be scheduled and interrupts to be
handled again.
void unlock()
{
    EnableInterrupts(); // Re-enable interrupts to allow normal operation.
}
```

这个方法的主要优点就是简单:
没有中断，线程便可以确信它的代码会继续执行下去，不会被其他线程干扰.
遗憾的是，这个方法的缺点也很多:

- 首先，这种方法要求我们允许所有调用线程执行特权操作 (打开或关闭中断)，即信任这种机制不会被滥用.
  但一个贪婪的程序可能在它开始时就调用 `lock()`，从而独占处理器.
  更糟的情况是，恶意程序可以在调用 `lock()` 后陷入死循环，从而使系统无法重新获得控制.
- 其次，这种方案不支持多处理器.
  如果多个线程运行在不同的 CPU 上，每个线程都试图进入同一个临界区，
  那么关闭中断也没有不能阻止它们同时进入临界区.
- 最后，关闭中断会导致中断丢失.
  例如 CPU 会丢失 $\mathrm{I/O}$ 操作完成的信息，从而不会去唤醒等待读取的进程.

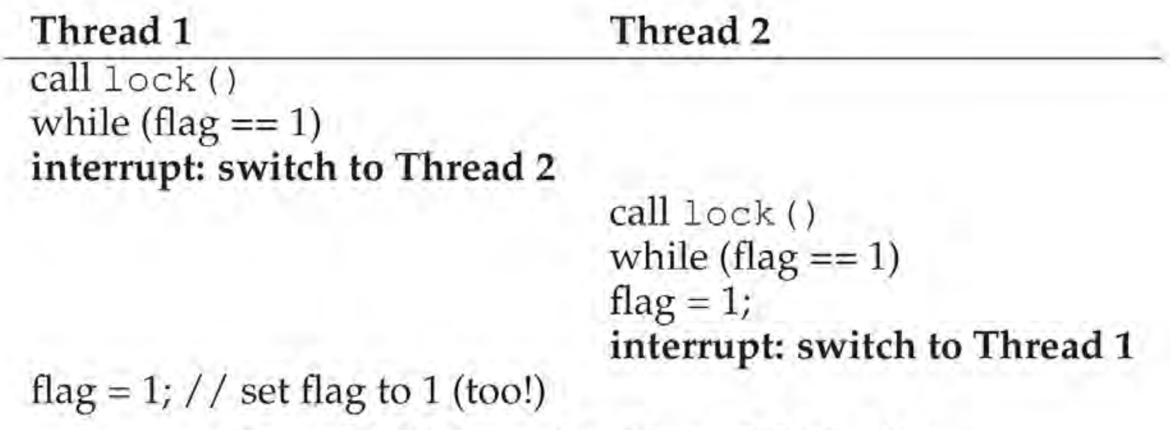
3.3.3 软件实现

因为关闭中断的方法无法工作在多处理器上，所以系统设计者开始让硬件支持锁.
最简单的硬件支持是**测试-设置指令** (test-and-set instruction)，也称为**原子交换** (atomic exchange)

为了理解 test-and-set 指令如何工作，我们首先实现一个不依赖它的锁，用一个变量标记锁是否被持有:

```
// Define a structure for a lock mechanism
typedef struct lock_t {
    int flag; // A flag to indicate the lock state (0 for unlocked, 1 for
locked)
} lock_t;

// Initialize the lock by setting the flag to 0 (unlocked)
void init(lock_t *mutex) {
    // 0 -> lock is available, 1 -> held
    mutex->flag = 0; // Set the initial state of the lock to unlocked
}

// Acquire the lock
void lock(lock_t *mutex) {
    // Spin-wait until the lock is available
    while (mutex->flag == 1) // TEST the flag
        ; // spin-wait (do nothing) - this is busy waiting
    mutex->flag = 1;         // now SET it! - lock is acquired
}

// Release the lock
void unlock(lock_t *mutex) {
    mutex->flag = 0; // Set the flag to 0 to indicate the lock is now unlocked
}
```

当第一个线程正处于临界区时，
如果另一个线程调用 `lock()`，它会在 `while` 循环中自旋等待 (spin-wait)，直到第一个线程调用 `unlock()` 清空标志.
然后等待的线程会退出 `while` 循环，设置标志，执行临界区代码.



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**

从上述交替执行可以看出，通过适时的中断，我们很容易构造出两个线程都将标志设置为 1，都能进入临界区的场景.
这个问题来源于 `flag` 本身也是先读后写的，它本身也是一个临界区，因此并不能实现锁的功能.

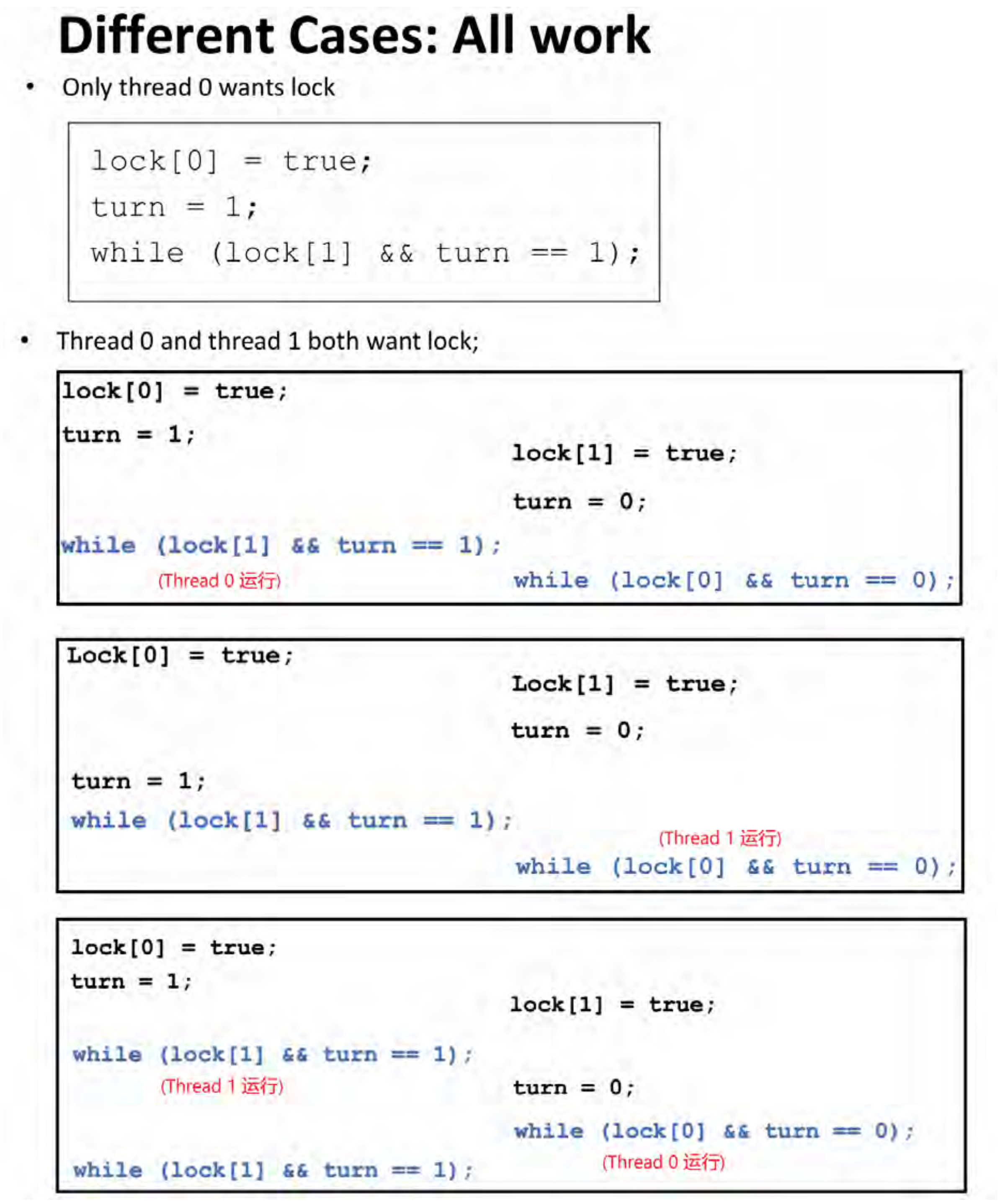
理论上，我们可以不依赖硬件只通过软件实现锁——**Peterson 算法**:

```
// Shared variables
int turn = 0; // Variable indicating whose turn it is to enter the critical
section
Boolean lock[2] = {false, false}; // Array indicating whether each thread wants
to enter the critical section

// Function to acquire the lock for the current thread
void acquire() {
    // Indicate that the current thread (identified by 'tid') wants to enter the
critical section
    lock[tid] = true;

    // Set turn to the other thread, signaling it to yield if it also wants to
enter
    turn = 1 - tid;
    // Wait while the other thread is in its critical section
    // This condition ensures mutual exclusion
    while (lock[1 - tid] && turn == 1 - tid) /* wait */ ;
}

// Function to release the lock for the current thread
void release() {
    // Indicate that the current thread is leaving the critical section
    lock[tid] = false;
}
```

Peterson 算法的基本原理:

- Peterson 算法利用两个主要变量来控制对临界区的访问:
  `turn` 表示当前是哪个线程的回合
  `lock[2]` 是一个布尔数组，指示每个线程是否希望进入临界区
- **工作机制:**
  当一个线程想要进入临界区时，它将 `lock[tid]` 设置为 `true`，表示它想要进入.
  它将 `turn` 设置为另一个线程的 ID，表明现在是另一个线程的回合.
  然后它会进入一个循环，检查另一个线程是否也想进入临界区.
  如果另一个线程的 `lock` 是 `true`，并且 `turn` 是指向另一个线程的 ID，那么当前线程将会等待.
  当线程完成其操作后，它将 `lock[tid]` 设置为 `false`，表明它不再想进入临界区.
  (Enter the critical section if and only if the other thread does not have lock or it's your turn)
- **特点:**
  满足互斥性: 能够确保同一时间至多有一个线程进入临界区.
  进程同步: 能够确保线程之间的适当协调.
  无死锁: 不可能出现死锁的情况，因为线程会在条件不满足时主动退出.

(图中红色注释存疑)



Different Cases: All work

- Only thread 0 wants lock

```
lock[0] = true;
turn = 1;
while (lock[1] && turn == 1);
```

- Thread 0 and thread 1 both want lock;

```
lock[0] = true;
turn = 1;
                                   lock[1] = true;
                                   turn = 0;
while (lock[1] && turn == 1);
       (Thread 0 运行)              while (lock[0] && turn == 0);
```

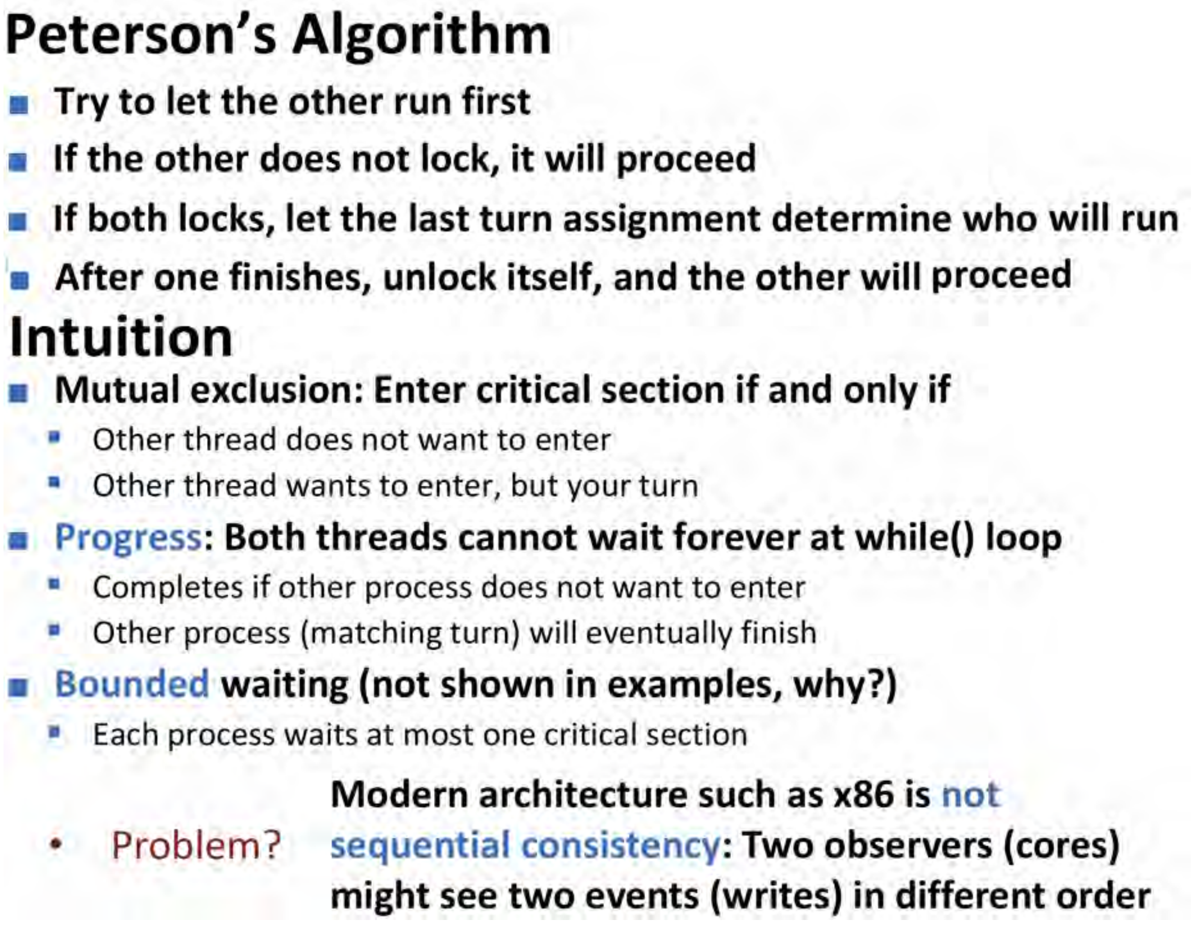
```
Lock[0] = true;
                                   Lock[1] = true;
                                   turn = 0;
turn = 1;
while (lock[1] && turn == 1);
                                         (Thread 1 运行)
                                   while (lock[0] && turn == 0);
```

```
lock[0] = true;
turn = 1;
                                   lock[1] = true;
while (lock[1] && turn == 1);
       (Thread 1 运行)              turn = 0;
                                   while (lock[0] && turn == 0);
                                        (Thread 0 运行)
while (lock[1] && turn == 1);
```

但上述算法在现代架构 (例如 x86) 上是错误的:
它们不满足顺序一致性，即两个观察者可能以不同的顺序看到两个事件 (写操作)

- **顺序一致性 (sequential consistency) 模型**
  这是一个理论模型，要求在多线程或多核心环境中，所有操作的执行结果应当看起来像是按某种顺序执行的，
  并且这个顺序在所有线程中是一致的.
  换句话说，如果线程 1 先写入一个变量，然后线程 2 读取该变量，那么线程 2 应该看到线程 1 更新后的值.
- 许多现代处理器 (如 x86 架构) 采用了更为放宽的**内存一致性模型**.
  这种模型允许内部优化，可能导致不同线程看到的操作顺序不同.
  例如一个核的写操作可能不会立即被另一个核看到，因为存在缓存、指令重排序等优化机制.
  这会导致 Peterson 算法中的两个线程可能会同时认为对方把锁让给了自己 (信息传播的延迟)，从而不满足互斥性.

因此在现代架构 (例如 x86) 不可能不依赖硬件而只通过软件实现锁.



Peterson's Algorithm

- Try to let the other run first
- If the other does not lock, it will proceed
- If both locks, let the last turn assignment determine who will run
- After one finishes, unlock itself, and the other will proceed

Intuition

- Mutual exclusion: Enter critical section if and only if
  - Other thread does not want to enter
  - Other thread wants to enter, but your turn
- Progress: Both threads cannot wait forever at while() loop
  - Completes if other process does not want to enter
  - Other process (matching turn) will eventually finish
- Bounded waiting (not shown in examples, why?)
  - Each process waits at most one critical section

- Problem? Modern architecture such as x86 is not sequential consistency: Two observers (cores) might see two events (writes) in different order

3.3.4 测试-设置指令

尽管 3.3.3 的想法很好，但没有硬件的支持是无法实现的.
幸运的是，一些系统提供了这一指令，支持基于这种概念创建简单的锁.
这个更强大的指令有不同的名字:
在 SPARC 上，这个指令叫 `ldstub` (load/store unsigned byte, 加载/保存无符号字节);
在 x86 上，是 `xchg` (atomic exchange, **原子交换**) 指令.
但它们基本上做的是同样的事，称为**测试-设置** (test-and-set) 指令，其功能的 C 语言描述 (伪代码) 如下:

```
// Function to perform a Test-and-Set operation
int TestAndSet(int *old_ptr, int new) {
    // Fetch the old value pointed to by old_ptr
    int old = *old_ptr; // Store the current value at old_ptr in 'old'

    // Store the new value into the location pointed to by old_ptr
    *old_ptr = new;     // Update the value at old_ptr with 'new'

    // Return the old value that was replaced
    return old;         // Return the previous value before the update
}
```

测试-设置指令做了下述事情:
它返回 `old_ptr` 指向的旧值，同时更新为 `new` 的新值.
这些代码原子地 (atomically) 执行，因而可以实现一个简单的**自旋锁** (spin lock):

```
// Define a structure to represent a lock
typedef struct lock_t {
    int flag; // Flag to indicate the state of the lock (0: available, 1: held)
} lock_t;

// Function to initialize the lock
void init(lock_t *lock) {
    // Set the initial state of the lock to available (0)
    lock->flag = 0;
}

// Function to acquire the lock
void lock(lock_t *lock) {
    // Continuously attempt to set the flag to 1 (indicating the lock is held)
    // using a TestAndSet operation, which atomically sets the flag and returns
the previous value.
    // The loop will keep spinning (busy-waiting) until the lock can be
acquired.
    while (TestAndSet(&lock->flag, 1) == 1)
        ; // spin-wait (do nothing)
}

// Function to release the lock
void unlock(lock_t *lock) {
    // Set the flag back to 0 (indicating the lock is now available)
    lock->flag = 0;
}
```

我们来理解为什么这个锁能工作:

- 首先假设一个线程调用了 `lock()`，且没有其他线程持有锁 (即 `flag` 为 `0`)
  当调用 `TestAndSet(&lock->flag,1)` 时会返回 `0`，因此线程会跳出 `while` 循环获取锁，
  同时也会原子地设置 `flag` 为 `1`，表明锁已经被持有.
  当线程离开临界区时，它调用 `unlock()` 将 `flag` 清理为 `0`
- 第二种场景: 假设某个线程已经持有锁 (即 `flag` 为 `1`)
  本线程调用 `lock()`，然后调用 `TestAndSet(&lock->flag,1)` 时会返回 `1`
  只要另一个线程一直持有锁，`TestAndSet(&lock->flag,1)` 就会一直返回 `1`，本线程会一直自旋.
  当 `flag` 终于被改为 `0` 时，本线程会调用 `TestAndSet(&lock->flag,1)`，
  返回 `0` 并且原子地设置 `flag` 为 `1`，从而获得锁，进入临界区.

将测试 (test 旧的锁值) 和设置 (set 新的值) 合并为一个原子操作之后，
我们就能保证同一时间至多有一个线程能获取锁，从而实现一个有效的互斥原语.

这是一种**自旋锁** (spin lock)，即不停自旋直到锁可用.
在单处理器上，需要**抢占式的调度器** (preemptive scheduler，即不断通过时钟中断一个线程，运行其他线程)
否则该自旋锁在单 CPU 上无法使用，因为一个自旋的线程永远不会放弃 CPU，从而浪费整个 CPU 时间片.

3.3.5 比较-交换指令

某些系统提供了另一个硬件原语，即**比较-交换指令**
在 SPARC 系统中是 `compare-and-swap`，而 `x86` 系统是 `compare-and-exchange`
其功能的 C 语言描述 (伪代码) 如下:

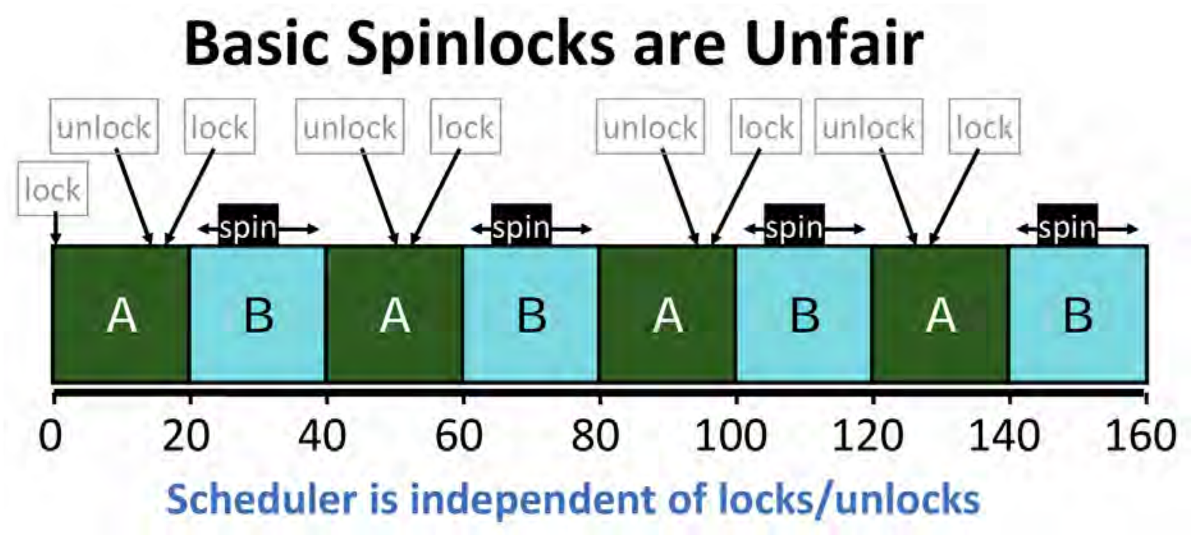
```c
// Function to perform an atomic compare-and-swap operation
int CompareAndSwap(int *ptr, int expected, int new) {
    // Retrieve the current value pointed to by ptr
    int actual = *ptr;

    // Check if the current value is equal to the expected value
    if (actual == expected)
        // If they are equal, update the value at ptr to the new value
        *ptr = new;

    // Return the actual value that was present at ptr before the operation
    return actual;
}
```

比较-交换的基本思路是检测 `ptr` 指向的值是否和 `expected` 相等.
如果是，则更新 `ptr` 所指的值为新值 `new`; 否则，什么也不做.
不论哪种情况，都返回该内存地址的实际值，让调用者能够知道执行是否成功.

有了比较-交换指令，就可以实现一个锁.
只需将 `while` 中的条件 `TestAndSet(&lock->flag, 1) == 1` 换为 `CompareAndSwap(&lock->flag, 0, 1) == 1` 即可:

```c
// Define a structure to represent a lock
typedef struct lock_t {
    int flag; // Flag to indicate the state of the lock (0: available, 1: held)
} lock_t;

// Function to initialize the lock
void init(lock_t *lock) {
    // Set the initial state of the lock to available (0)
    lock->flag = 0;
}

// Function to acquire the lock
void lock(lock_t *lock) {
    // Continuously attempt to set the flag to 1 (indicating the lock is held)
    // using a CompareAndSwap operation, which atomically checks and sets the
flag.
    // The loop will keep spinning (busy-waiting) until the lock can be
successfully acquired.
    while (CompareAndSwap(&lock->flag, 0, 1) == 1)
        ; // spin-wait (do nothing)
}

// Function to release the lock
void unlock(lock_t *lock) {
    // Set the flag back to 0 (indicating the lock is now available)
    lock->flag = 0;
}
```

它的行为等价于测试-设置指令构建的自旋锁.
这种简单的自旋锁是不公平的 (例如一个线程有可能一直自旋，即使其他线程在不断获取和释放锁)



3.3.6 获取-增加指令

最后一个硬件原语是**获取-增加** (fetch-and-add) 指令
它能返回特定地址的旧值，并原子地让该值自增一.
其功能的 C 语言描述 (伪代码) 如下:

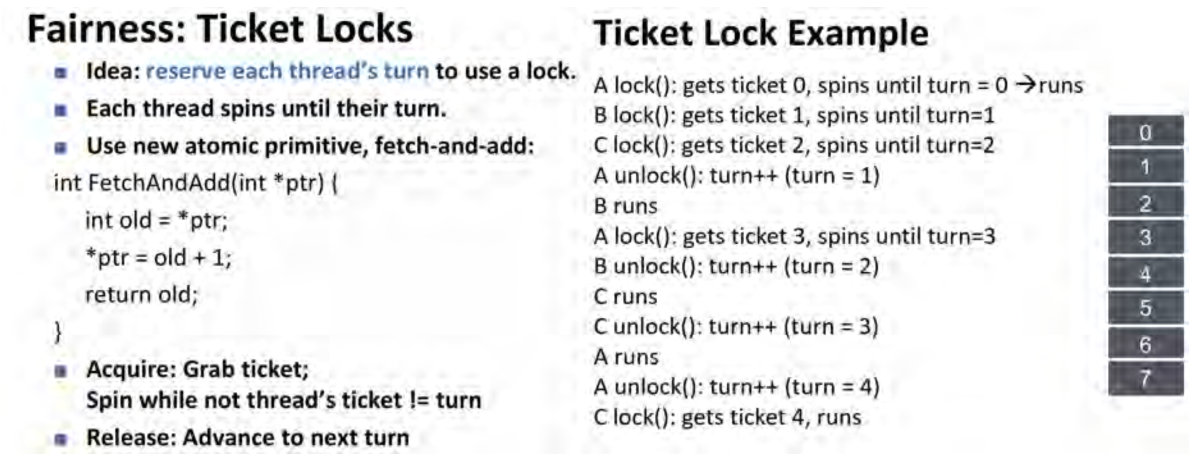
```c
// Function to perform an atomic fetch-and-add operation
int FetchAndAdd(int *ptr) {
    // Store the current value at the pointer's address in old
    int old = *ptr;

    // Increment the value at *ptr by 1
    *ptr = old + 1;

    // Return the old value (the value before the increment)
    return old;
}
```

我们可以使用获取-增加指令，实现一个更有趣的 ticket 锁:

```c
// Define a structure to represent a lock using a ticket-based locking mechanism
typedef struct lock_t {
    int ticket; // Next ticket number to be issued
    int turn;   // Current turn number for lock acquisition
} lock_t;

// Function to initialize the lock
void lock_init(lock_t *lock) {
    // Initialize the ticket and turn values to 0
    lock->ticket = 0;
    lock->turn = 0;
}

// Function to acquire the lock
void lock(lock_t *lock) {
    // Get the next ticket number by incrementing the ticket
    int myturn = FetchAndAdd(&lock->ticket);

    // Spin until it's this thread's turn to acquire the lock
    while (lock->turn != myturn)
        ; // spin (busy-waiting)
}

// Function to release the lock
void unlock(lock_t *lock) {
    // Increment the turn to allow the next thread to acquire the lock
    FetchAndAdd(&lock->turn);
}
```

这个解决方案使用了 `ticket` 和 `turn` 变量来构建锁.
基本操作也很简单:
如果线程希望获取锁，首先对一个 `ticket` 值执行一个原子的获取-相加指令.
这个值作为该线程的 `turn` (顺位, 即 `myturn`)
根据全局共享的 `lock->turn` 变量，当某一个线程的 `myturn == turn` 时，则轮到这个线程进入临界区.
`unlock` 则是增加 `turn`，从而下一个等待线程可以进入临界区.

不同于之前的方法，本方法能够保证所有线程都能抢到锁.
只要一个线程获得了 `ticket` 值，它就最终会被调度.
而基于测试-设置指令或比较-交换指令构建的自旋锁是不公平的.
(例如一个线程有可能一直自旋，即使其他线程在不断获取和释放锁)


Fairness: Ticket Locks

- Idea: reserve each thread's turn to use a lock.
- Each thread spins until their turn.
- Use new atomic primitive, fetch-and-add:

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

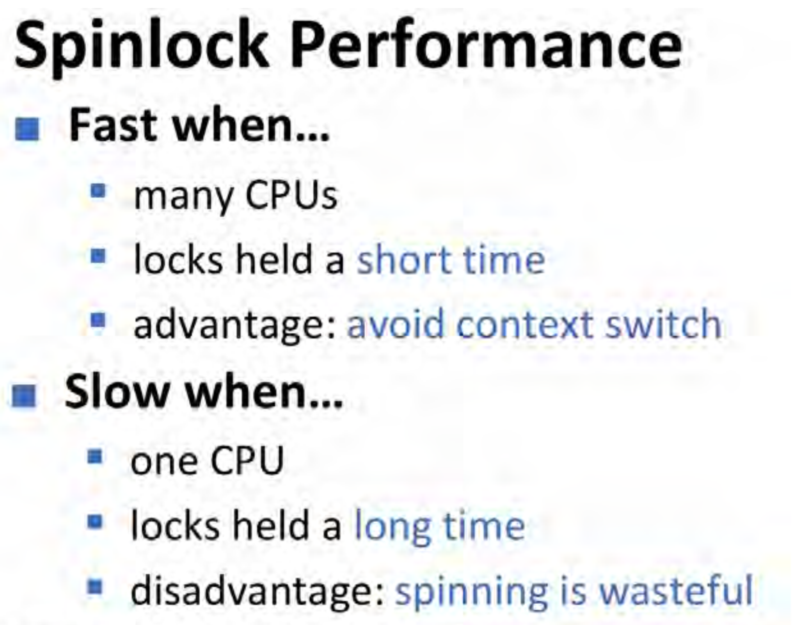
- Acquire: Grab ticket;
  Spin while not thread's ticket != turn
- Release: Advance to next turn

Ticket Lock Example

A lock(): gets ticket 0, spins until turn = 0 →runs
B lock(): gets ticket 1, spins until turn=1
C lock(): gets ticket 2, spins until turn=2
A unlock(): turn++ (turn = 1)
B runs
A lock(): gets ticket 3, spins until turn=3
B unlock(): turn++ (turn = 2)
C runs
C unlock(): turn++ (turn = 3)
A runs
A unlock(): turn++ (turn = 4)
C lock(): gets ticket 4, runs

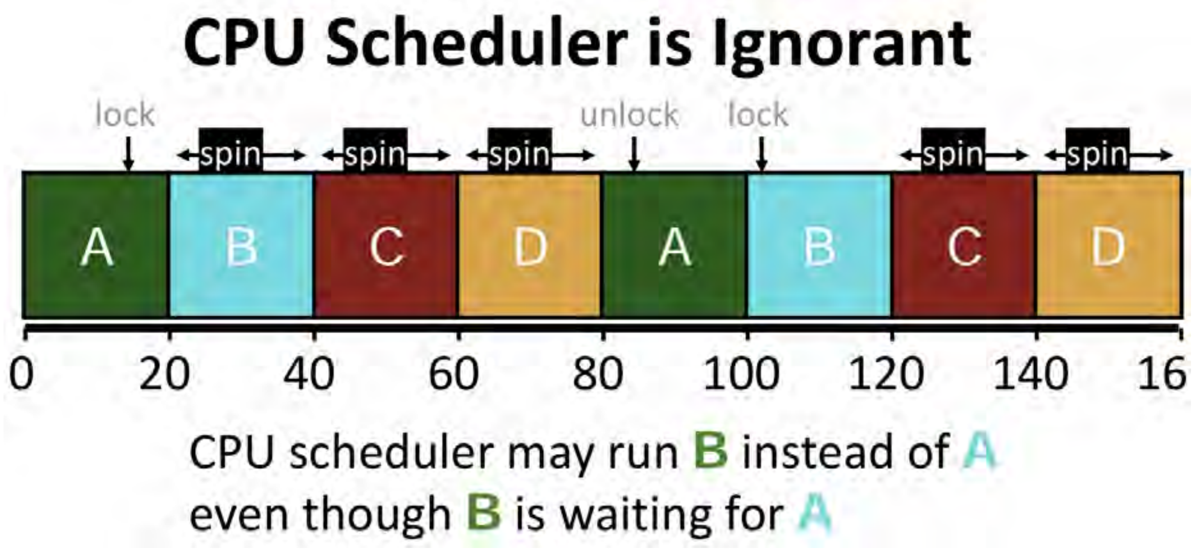
3.3.7 自旋的替代方案

基于硬件的锁简单而且有效，但在某些场景下，这些解决方案会效率低下.
以两个线程运行在单处理器上为例，假设一个线程 0 在持有锁时被中断.
此时若线程 1 去获取锁，则会发现锁已经被持有，于是它就开始自旋，不断自旋，直到时钟中断切换到线程 0 继续运行.
在上述场景中，线程 1 一直自旋检查一个不会改变的值，浪费掉整个时间片!
如果有 N 个线程去竞争一个锁，则情况会更糟糕:
同样的场景下，会浪费 $N-1$ 个时间片用于自旋并等待一个线程释放锁.
如何阻止锁不必要地自旋从而减少 CPU 时间的浪费?
仅靠硬件支持是不够的，我们还需要操作系统支持!



Spinlock Performance

- **Fast when…**
  - many CPUs
  - locks held a short time
  - advantage: avoid context switch
- **Slow when…**
  - one CPU
  - locks held a long time
  - disadvantage: spinning is wasteful



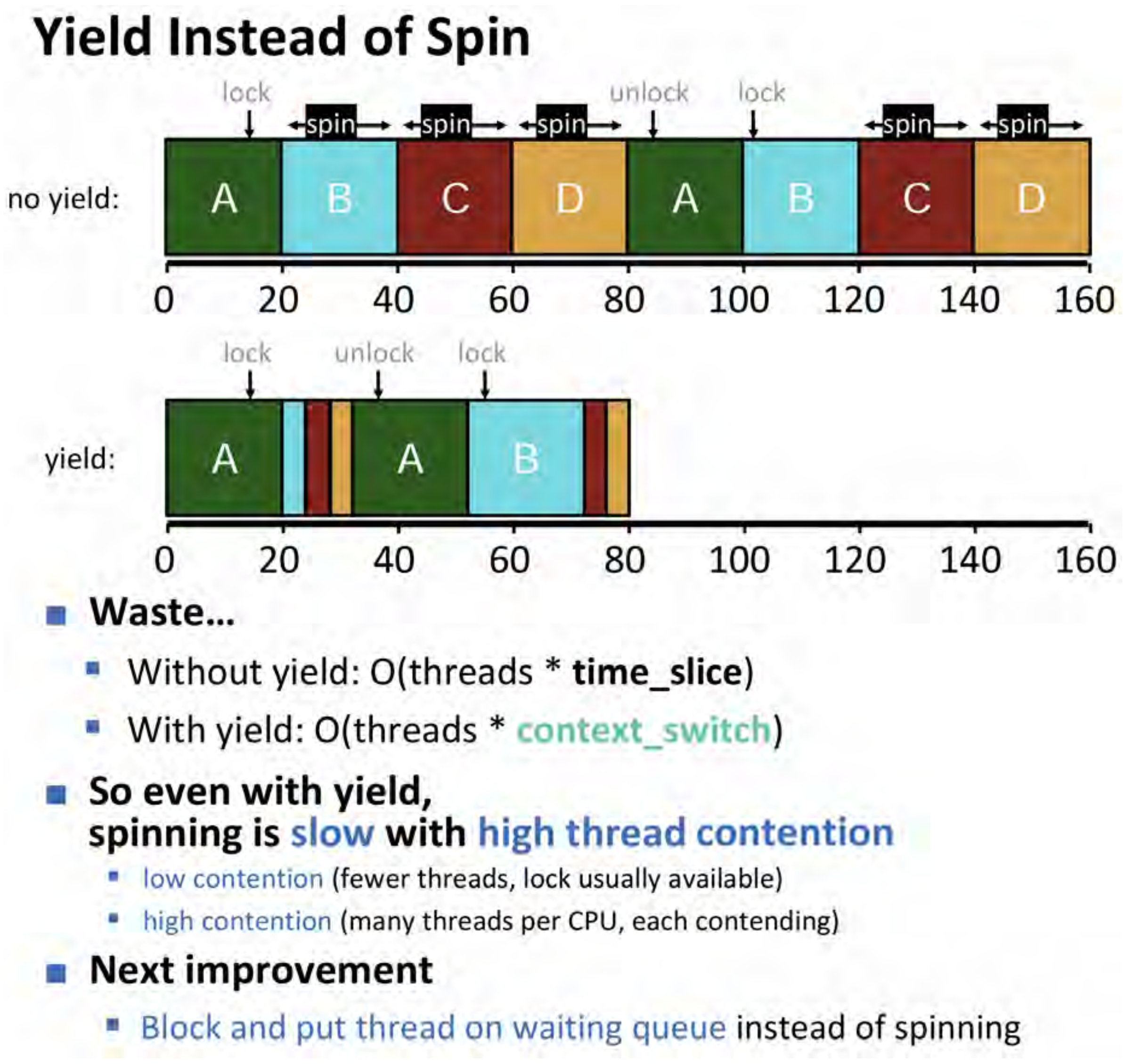
CPU Scheduler is Ignorant

CPU scheduler may run **B** instead of **A**
even though **B** is waiting for **A**

第一种简单友好的方法就是在要自旋的时候主动放弃 CPU:

```
// Function to initialize the lock
void init() {
    // Set the flag to 0, indicating that the lock is available
    flag = 0;
}

// Function to acquire the lock
void lock() {
    // Continuously attempt to set the flag to 1 (indicating the lock is held)
    // using a TestAndSet operation. If the flag is already 1, it means the lock
is held.
    while (TestAndSet(&flag, 1) == 1)
        yield(); // Give up the CPU to allow other threads to run
}

// Function to release the lock
void unlock() {
    // Set the flag back to 0, indicating the lock is now available
    flag = 0;
}
```

在这种方法中，我们假定操作系统提供了原语 `yield()` (The word brings **Game of Thrones** to my mind)
线程可以调用 `yield()` 主动放弃 CPU，让其他线程运行.
于是线程可以处于 3 种状态之一 (运行、就绪和阻塞)
`yield()` 系统调用能够让运行态变为就绪态，即**取消调度** (deschedule) 了自己，从而允许其他线程运行.



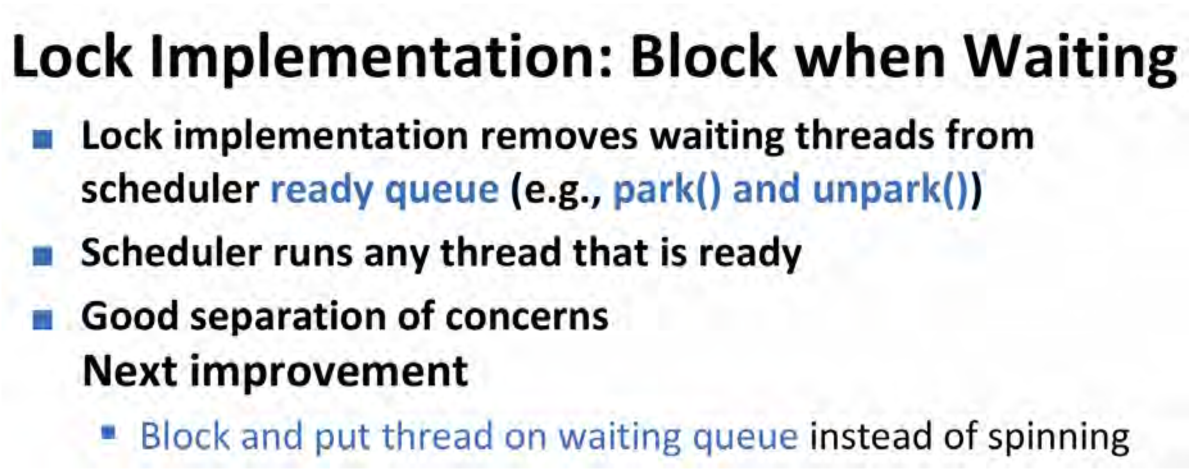
Yield Instead of Spin

- **Waste...**
  - Without yield: O(threads * **time_slice**)
  - With yield: O(threads * context_switch)
- **So even with yield, spinning is slow with high thread contention**
  - low contention (fewer threads, lock usually available)
  - high contention (many threads per CPU, each contending)
- **Next improvement**
  - Block and put thread on waiting queue instead of spinning

尽管这种方法能够避免自旋浪费 CPU 时间，
但当许多线程反复竞争一把锁且轮转调度时上下文切换的成本依然很高.
更糟的是，一个线程可能一直处于让出的循环 (从而被饿死)，而其他线程反复进出临界区.

使用队列: 休眠替代自旋
前面一些方法的真正问题是存在太多的偶然性.
调度程序决定如何调度.
如果调度不合理，那么线程或者一直自旋 (第一种方法)，或者立刻让出 CPU (第二种方法)
无论哪种方法，都可能造成 CPU 的浪费，以及线程饿死的问题.

因此我们必须显式地施加某种控制，决定锁释放时谁能抢到锁.
为做到这一点， 我们需要操作系统的支持，利用一个队列来保存等待锁的线程.
简单起见，我们利用 Solaris 提供的两个调用:
`park()` 能够让调用线程休眠，而 `unpark(threadID)` 则会唤醒 `threadID` 标识的线程.
我们可以用这两个调用来实现锁，让调用者在获取不到锁时休眠，在锁可用且轮到它时被唤醒.



Lock Implementation: Block when Waiting

- Lock implementation removes waiting threads from scheduler ready queue (e.g., park() and unpark())
- Scheduler runs any thread that is ready
- Good separation of concerns
  Next improvement
  - Block and put thread on waiting queue instead of spinning

```
// Define a structure to represent a lock with additional fields for queuing
typedef struct lock_t {
    int flag;      // Indicates if the lock is acquired (1) or available (0)
    int guard;     // Guard variable for protecting access to the lock structure
    queue_t *q;    // Queue to hold threads waiting for the lock
} lock_t;

// Function to initialize the lock
void lock_init(lock_t *m) {
    // Set the flag to 0 (lock is available) and guard to 0
    m->flag = 0;
    m->guard = 0;
    // Initialize the waiting queue
    queue_init(m->q);
}

// Function to acquire the lock
void lock(lock_t *m) {
    // Spin to acquire the guard lock to ensure exclusive access to the lock
structure
    while (TestAndSet(&m->guard, 1) == 1)
        ; // Acquire guard lock by spinning (only take a short time, so spinning
is acceptable)

    // Check if the lock is currently available
    if (m->flag == 0)
    {
        // If available, acquire the lock by setting the flag to 1
        m->flag = 1; // Lock is acquired
        m->guard = 0; // Release guard lock
    }
    else
    {
        // If the lock is not available, add the current thread to the queue
        queue_add(m->q, gettid());
        m->guard = 0; // Release guard lock
        park(); // Put the current thread to sleep until it is unparked
    }
}

// Function to release the lock
void unlock(lock_t *m) {
    // Spin to acquire the guard lock to ensure exclusive access to the lock
structure
    while (TestAndSet(&m->guard, 1) == 1)
        ; // Acquire guard lock by spinning

    // Check if there are any threads waiting in the queue
    if (queue_empty(m->q))
        // If no one is waiting, release the lock by setting the flag back to 0
        m->flag = 0; // Let go of lock; no one wants it
    else
        // If there are waiting threads, unpark the next thread in the queue
        unpark(queue_remove(m->q)); // Hold lock (for next thread!)

    // Release the guard lock
    m->guard = 0;
}
```

在这个例子中，我们做了两件有趣的事:
首先，我们将之前的测试-设置指令和等待队列结合，实现了一个更高性能的锁.
其次，我们通过等待队列来控制谁会获得锁，避免饿死.
值得注意的是:

- ① `guard` 起到了自旋锁的作用，围绕着 `flag` 和等待队列操作.
  因此这个方法并没有完全避免自旋等待，但这个自旋等待时间是很有限的
  (不是用户定义的临界区，而是在 `lock` 和 `unlock` 代码中的几条指令)
  因此这种方法是合理的.
- ② 你可能注意到在 `lock()` 函数中，如果线程不能获取锁 (它已被持有)，
  那么线程会把自己加入等待队列 (通过调用 `gettid()` 获得当前的线程ID)，将 `guard` 设置为 `0`，然后让出 CPU
  如果我们在 `park()` 之后，才把 `guard` 设置为 `0` 释放锁，会发生什么呢?
  当前线程执行 `park()` 时，它会进入等待状态，并不会执行之后的 `m->guard = 0;`
  因为 `guard` 未释放，其他等待的线程会一直在 `while (TestAndSet(&m->guard, 1) == 1)` 处自旋等待，
  无法获取到 `guard`，这会导致**死锁**，因为该锁会一直被当前线程持有，但当前线程已经进入睡眠状态，
  且无法释放 `guard` 供其他线程使用.
- ③ 当要唤醒另一个线程时，`flag` 并没有设置为 `0`，这是正确的.
  当一个线程执行 `unlock()` 时，如果发现有其他线程在等待队列中，
  它就会直接调用 `unpark()` 唤醒队列中的下一个线程，让其获得执行权限.
  锁的持有状态直接从当前线程传递给下一个线程.
  由于 `flag` 一直保持为 `1`，唤醒的线程在执行 `park()` 后返回时，实际上已经获得了锁的持有权.

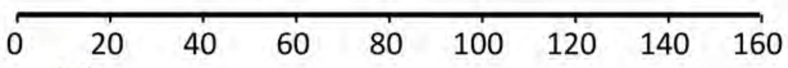
  如果在唤醒等待线程前将 `flag` 设置为 `0`，那么在这段时间内，锁会处于一个未被持有的状态.
  这可能导致其他非等待队列的线程误认为锁可用，进而尝试获取锁，从而破坏了队列的顺序性.
- ④ **唤醒/等待竞争 (wakeup/waiting race)**
  假设线程 A 正在尝试获取锁，但此时锁被线程 B 持有.
  线程 A 进入 `lock()` 函数并判断锁不可用，因此将调用 `park()` 函数以加入等待队列.
  然而在 A 执行 `park()` 前的瞬间可能发生线程切换，导致以下情景:
  1. **线程切换:** 在 A 执行 `park()` 之前，系统切换到持有锁的线程 B
  2. **锁释放:** 假设线程 B 即将退出，它发现等待队列为空 (此时线程 A 并不在等待队列中)，因此直接释放锁.

3. **再切换到线程 A:** 当系统再次切换回线程 A 时，A 继续执行接下来的 `park()` 调用，进入休眠.
   由于线程 B 释放锁的操作已经发生在 `park()` 调用之前 (甚至没有 `unpark()`，因为等待队列为空)，
   因此 A 错过了本该唤醒它的信号，可能会永久地睡眠下去 (如果后续没有执行其他需要该锁的线程的话)

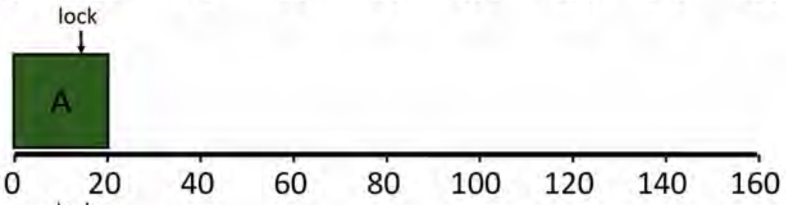
Solaris 通过增加了第三个系统调用 `setpark()` 来解决这一问题.
一个线程可以调用 `setpark()` 表明自己马上要 `park()`
如果刚好另一个线程被调度，并且调用了 `unpark()`，
那么后续的 `park()` 调用就会直接返回，而不是进入睡眠.
于是我们可将 `lock()` 函数修改为:

```c
// Function to acquire the lock
void lock(lock_t *m) {
    // Spin to acquire the guard lock to ensure exclusive access to the lock structure
    while (TestAndSet(&m->guard, 1) == 1)
        ; // Acquire guard lock by spinning (only take a short time, so spinning is acceptable)

    // Check if the lock is currently available
    if (m->flag == 0)
    {
        // If available, acquire the lock by setting the flag to 1
        m->flag = 1; // Lock is acquired
        m->guard = 0; // Release guard lock
    }
    else
    {
        // If the lock is not available, add the current thread to the queue
        queue_add(m->q, gettid());

        // Indicate that the thread is about to enter sleep mode
        // `setpark()` is called to mark the thread as "waiting," which lets the system know
        // that it is about to call `park()` and enter a sleep state.
        // If another thread calls `unpark()` after `setpark()` but before the `park()` call,
        // then the subsequent `park()` will return immediately, without putting the thread to sleep.
        // This prevents the wakeup/waiting race by ensuring that any signal to wake the thread
        // is not lost between `setpark()` and `park()`.
        setpark(); // notify of plan

        m->guard = 0; // Release guard lock

        // Put the current thread to sleep until it is unparked
        // `park()` will either sleep if no `unpark()` has been called, or it will
        // return immediately if an `unpark()` has occurred between `setpark()` and `park()`.
        park(); // Put the current thread to sleep until it is unparked
    }
}
```

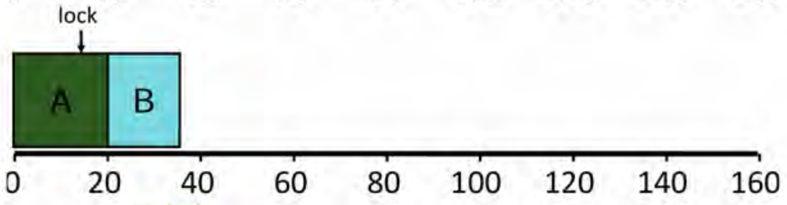
RUNNABLE: A, B, C, D
RUNNING: <empty>
WAITING: <empty>



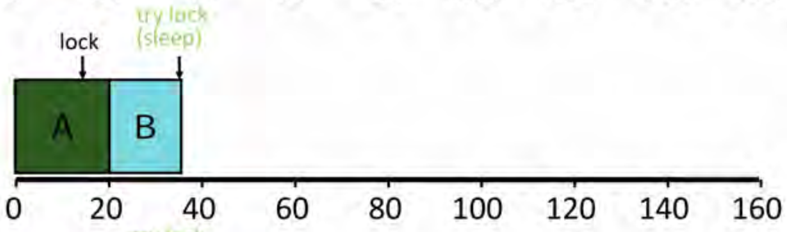
RUNNABLE: B, C, D
RUNNING: A
WAITING: <empty>



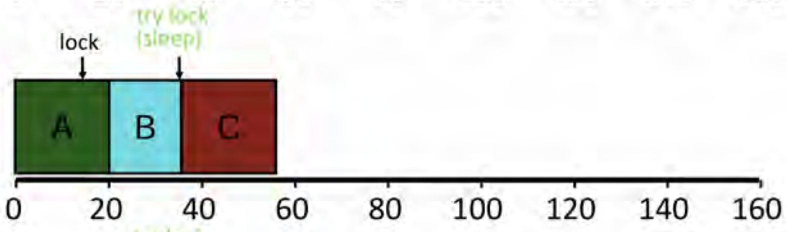
RUNNABLE: C, D, A
RUNNING: B
WAITING: <empty>



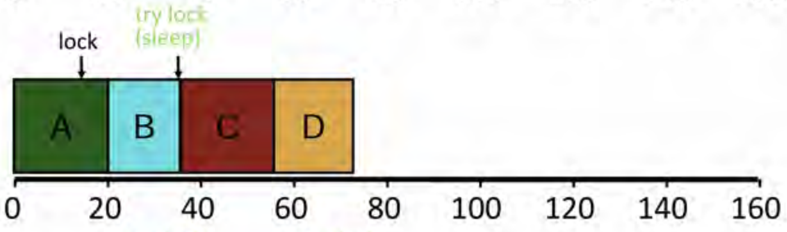
RUNNABLE: C, D, A
RUNNING:
WAITING: B



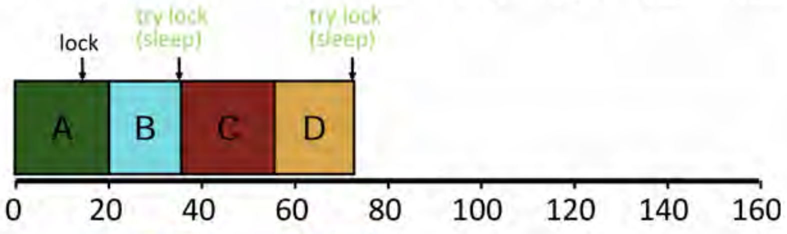
RUNNABLE: D, A
RUNNING: C
WAITING: B



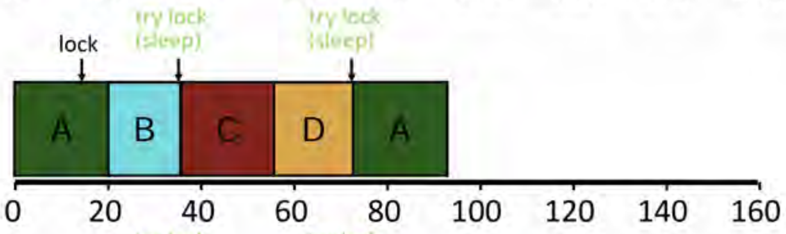
RUNNABLE: A, C
RUNNING: D
WAITING: B



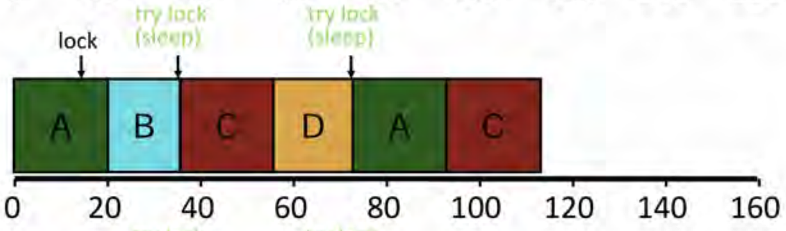
RUNNABLE: A, C
RUNNING:
WAITING: B, D



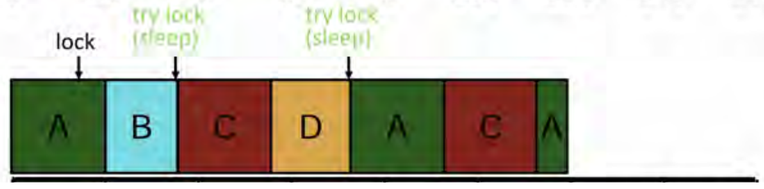
RUNNABLE: C
RUNNING: A
WAITING: B, D

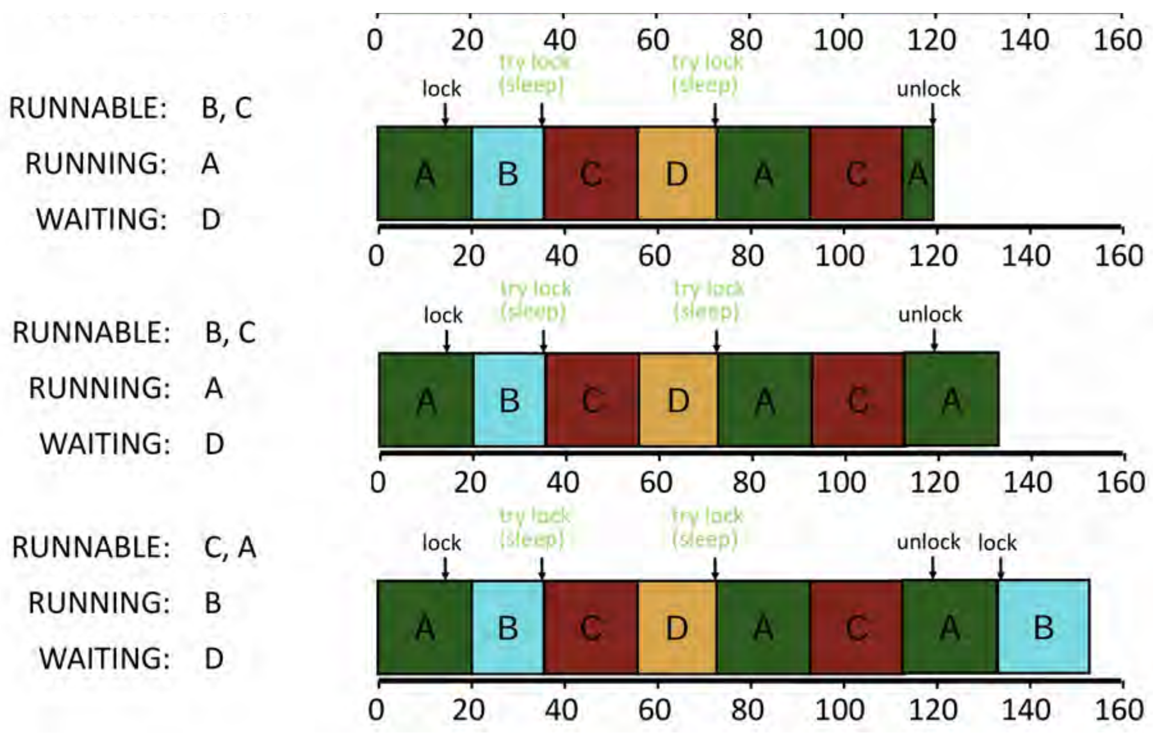


RUNNABLE: A
RUNNING: C
WAITING: B, D



RUNNABLE: C
RUNNING: A
WAITING: B, D

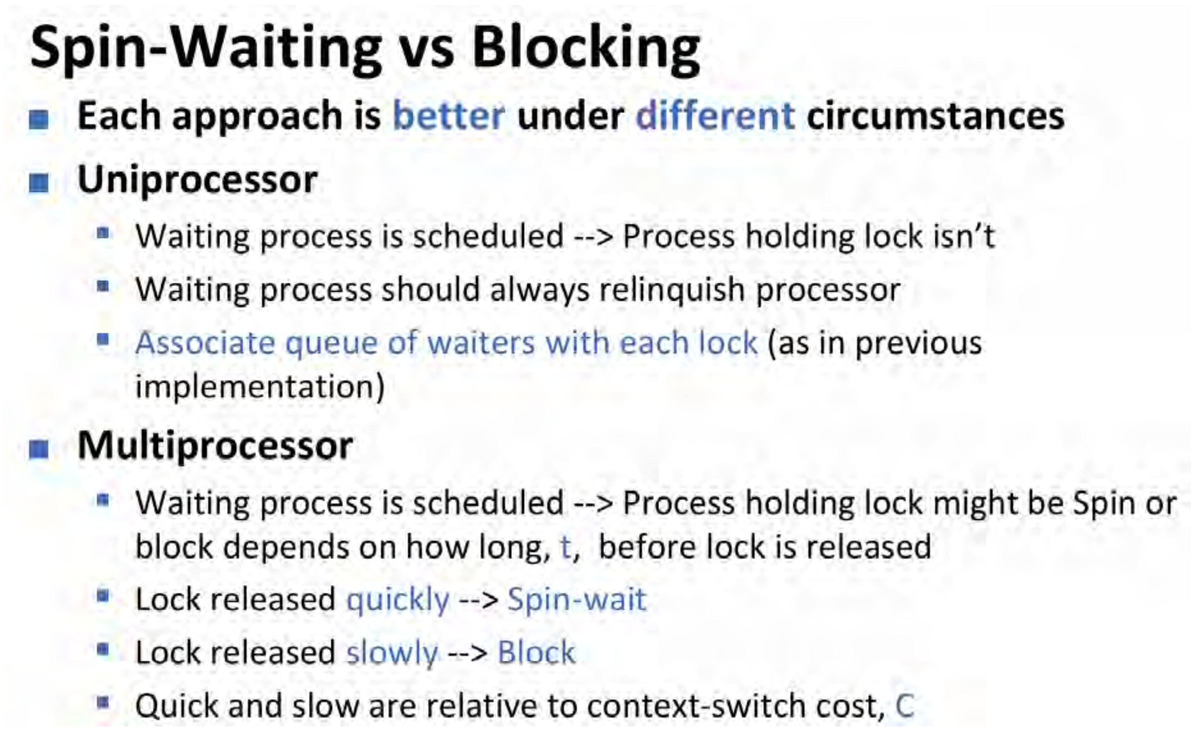




3.3.8 自旋与休眠的比较

在不同的情况下，自旋等待 (Spin-Waiting) 和阻塞等待 (Blocking) 各有其优势.
它们的适用场景取决于系统的硬件架构和锁的释放速度.

- **单处理器 (Uniprocessor) 情况:**
  由于单核 CPU 只有一个处理器核心，处理器不能同时执行多个线程.
  此时自旋等待不是最优选择，为了避免浪费 CPU 资源，等待线程需要主动放弃 CPU 时间片.
- **多处理器 (Multiprocessor) 情况:**
  多核处理器可以同时执行多个线程，因此是采用自旋等待还是采用阻塞等待，
  取决于锁被释放的时间 `t` 和上下文切换的开销 `c` 哪个更大.
  - 如果锁能迅速释放 (例如持有锁的进程只做非常短的工作)，
    那么等待线程可以选择自旋等待，即反复检查锁的状态，直到锁变为可用.
    这种方法适用于锁竞争较轻的情况.
  - 如果锁释放较慢，等待线程可以选择阻塞等待，
    即将线程挂起，并在锁可用时被唤醒.
    阻塞等待适用于锁竞争激烈或者持有锁的线程执行时间较长的场景.
  - 如果上下文切换的成本非常高，那么自旋等待可能会更有效，因为它避免了过多的上下文切换开销.
    如果上下文切换的成本相对较低，阻塞等待则更合适，因为它能避免浪费 CPU 资源.
    但做上述决策需要了解未来的情况，进行任何特殊预测可能会带来更多的开销.



Spin-Waiting vs Blocking

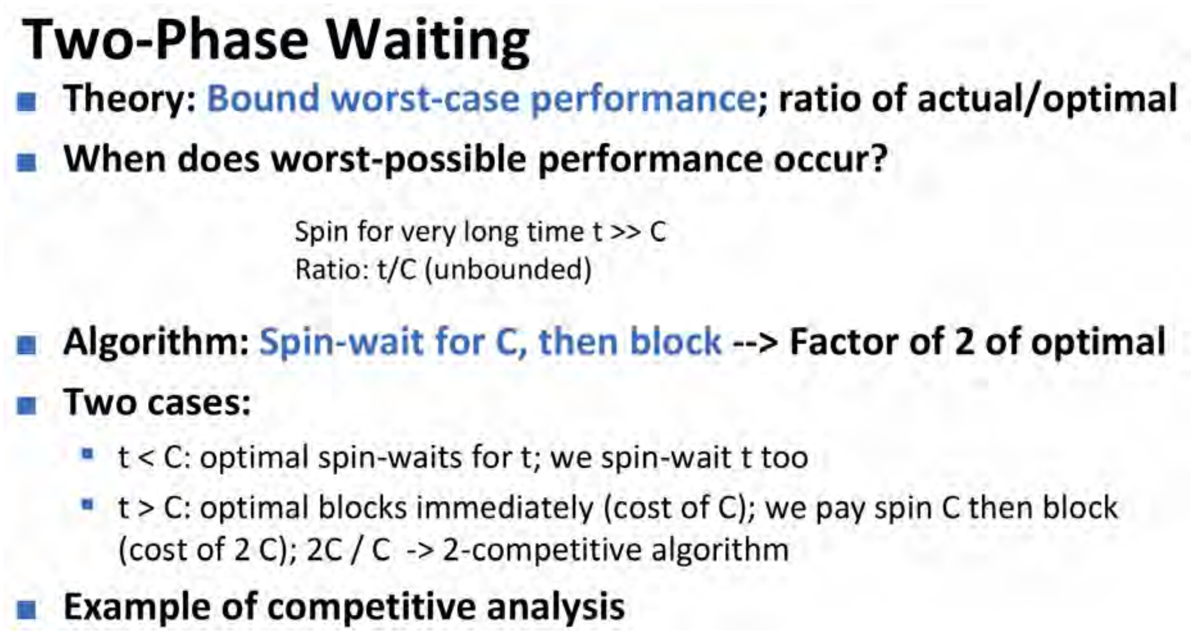
- Each approach is better under different circumstances
- Uniprocessor
  - Waiting process is scheduled --> Process holding lock isn't
  - Waiting process should always relinquish processor
  - Associate queue of waiters with each lock (as in previous implementation)
- Multiprocessor
  - Waiting process is scheduled --> Process holding lock might be Spin or block depends on how long, t, before lock is released
  - Lock released quickly --> Spin-wait
  - Lock released slowly --> Block
  - Quick and slow are relative to context-switch cost, C

解决方案: 二阶段等待 (Two-Phase Waiting)
二阶段等待的目标是限定最坏情况的性能，并通过引入一个实际与最优性能的比率来确保较好的性能.
该方法的关键在于在不同的情况下调整等待策略，从而在最坏情况下避免性能过于糟糕.

- **① 自旋等待阶段:**
  在锁释放之前，我们先进行自旋等待，持续时间为 `c` (上下文切换的开销)
  这意味着，如果锁在 `c` 时间内释放，那么线程会一直在自旋等待，而不会进入阻塞.
- **② 阻塞等待阶段:**
  如果在 `c` 时间内锁没有被释放，则转为阻塞等待.
  此时线程会选择放弃处理器，直到锁释放并唤醒.

该算法的关键在于通过这种自旋等待和阻塞等待的组合，
能够使得最坏情况下的性能与最优情况相差不超过一个常数倍，即保证一个接近最优的性能.

- **当 `t < c` 时:**
  在这种情况下，最优算法会在 `t` 时间内完成自旋等待，完全没有必要进入阻塞等待.
  根据二阶段等待的算法，我们同样会进行自旋等待 `t` 时间，所以这时我们的算法表现得与最优算法相同.
- **当 `t > c` 时:**
  最优算法会立刻进入阻塞状态 (并付出 `c` 的开销),
  而我们的算法则会先进行 `c` 时间的自旋等待，然后再进入阻塞状态.
  因此我们的算法总共会支付 `2c` 的开销，相比于最优算法的开销 `c`，我们多花了一个倍数的代价.
  所以算法的竞争比率为 `2c / c = 2`，意味着我们算法在最坏情况下是一个 2-竞争算法.
  这意味着，即使在最坏情况下，二阶段等待的算法性能也不会比最优算法差两倍以上.



Two-Phase Waiting

- **Theory: Bound worst-case performance; ratio of actual/optimal**
- **When does worst-possible performance occur?**

  Spin for very long time t >> C
  Ratio: t/C (unbounded)

- **Algorithm: Spin-wait for C, then block --> Factor of 2 of optimal**
- **Two cases:**
  - t < C: optimal spin-waits for t; we spin-wait t too
  - t > C: optimal blocks immediately (cost of C); we pay spin C then block (cost of 2 C); 2C / C -> 2-competitive algorithm
- **Example of competitive analysis**

3.4 基于锁的并发数据结构

3.4.1 并发计数器

(1) 非并发的实现

首先考虑一个非并发的计数器:

```c
// Definition of a structure to represent a simple counter.
typedef struct counter_t {
    int value; // Stores the current value of the counter.
} counter_t;

// Initializes the counter to 0.
void init(counter_t *c) {
    c->value = 0; // Set the counter's value to 0.
}

// Increments the counter's value by 1.
void increment(counter_t *c) {
    c->value++; // Increase the counter's value by 1.
}

// Decrements the counter's value by 1.
void decrement(counter_t *c) {
    c->value--; // Decrease the counter's value by 1.
}

// Retrieves the current value of the counter.
int get(counter_t *c) {
    return c->value; // Return the current value of the counter.
}
```

(2) 并发化

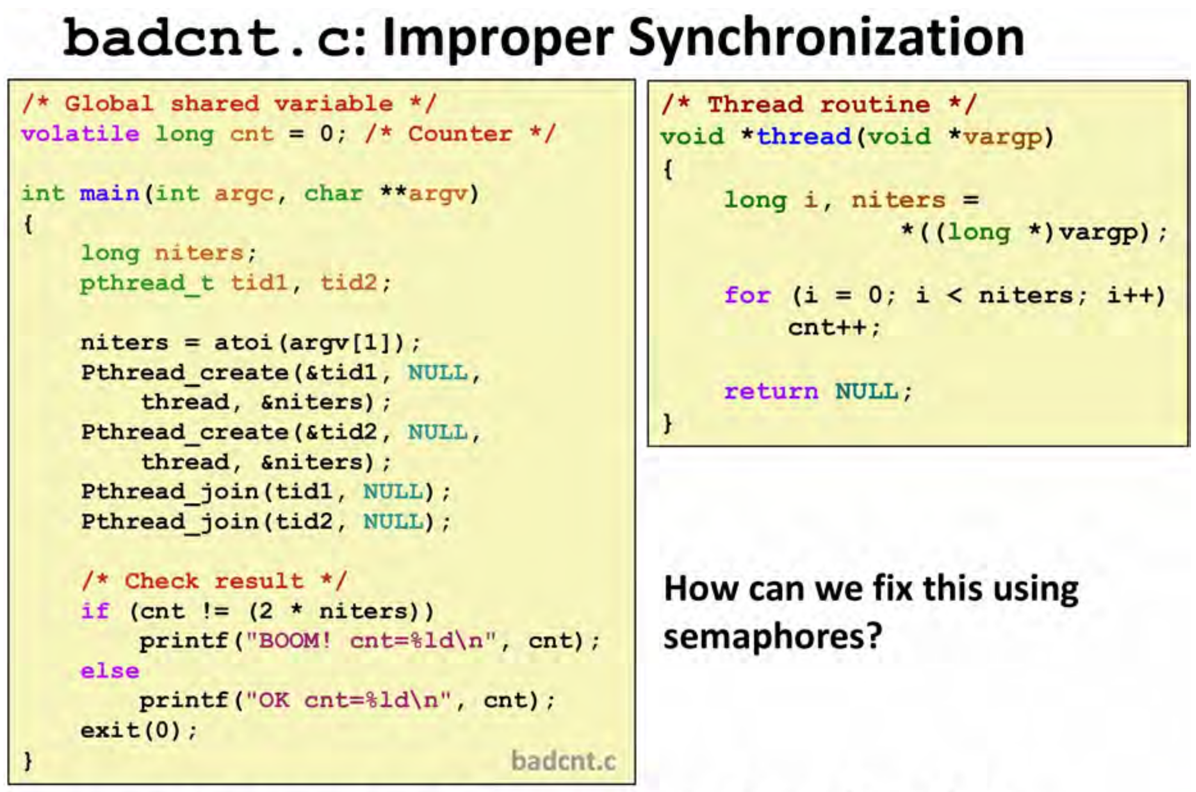
我们很容易通过加一把锁来将其变为并发计数器，
只需在调用函数操作该数据结构时获取锁，从调用返回时释放锁即可.

```
// Definition of a structure to represent a thread-safe counter.
typedef struct counter_t {
    int value;                // Stores the current value of the counter.
    pthread_mutex_t lock;     // A mutex lock to ensure thread-safe operations.
} counter_t;

// Initializes the counter and its associated lock.
void init(counter_t *c) {
    c->value = 0;                              // Set the counter's value to 0.
    Pthread_mutex_init(&c->lock, NULL);        // Initialize the mutex lock.
}

// Increments the counter's value by 1 in a thread-safe manner.
void increment(counter_t *c) {
    Pthread_mutex_lock(&c->lock);              // Acquire the lock before
modifying the value.
    c->value++;                                // Increase the counter's value
by 1.
    Pthread_mutex_unlock(&c->lock);            // Release the lock after
modification.
}

// Decrements the counter's value by 1 in a thread-safe manner.
void decrement(counter_t *c) {
    Pthread_mutex_lock(&c->lock);              // Acquire the lock before
modifying the value.
    c->value--;                                // Decrease the counter's value
by 1.
    Pthread_mutex_unlock(&c->lock);            // Release the lock after
modification.
}

// Retrieves the current value of the counter in a thread-safe manner.
int get(counter_t *c) {
    Pthread_mutex_lock(&c->lock);              // Acquire the lock before
accessing the value.
    int rc = c->value;                         // Read the current value of the
counter.
    Pthread_mutex_unlock(&c->lock);            // Release the lock after
accessing the value.
    return rc;                                 // Return the retrieved value.
}
```

或者使用**二值信号量** (参见后文) 来实现锁:

```
// Definition of a structure to represent a thread-safe counter.
typedef struct counter_t {
    int value;          // Stores the current value of the counter.
    sem_t lock;         // A binary semaphore to ensure thread-safe operations.
} counter_t;
```

```
// Initializes the counter and its associated binary semaphore.
void init(counter_t *c) {
    c->value = 0;                        // Set the counter's value to 0.
    sem_init(&c->lock, 0, 1);            // Initialize the binary semaphore
with value 1.
}

// Increments the counter's value by 1 in a thread-safe manner.
void increment(counter_t *c) {
    sem_wait(&c->lock);                  // Acquire the semaphore (decrement
it to 0).
    c->value++;                          // Increase the counter's value by 1.
    sem_post(&c->lock);                  // Release the semaphore (increment
it to 1).
}

// Decrements the counter's value by 1 in a thread-safe manner.
void decrement(counter_t *c) {
    sem_wait(&c->lock);                  // Acquire the semaphore (decrement
it to 0).
    c->value--;                          // Decrease the counter's value by 1.
    sem_post(&c->lock);                  // Release the semaphore (increment
it to 1).
}

// Retrieves the current value of the counter in a thread-safe manner.
int get(counter_t *c) {
    sem_wait(&c->lock);                  // Acquire the semaphore (decrement
it to 0).
    int rc = c->value;                   // Read the current value of the
counter.
    sem_post(&c->lock);                  // Release the semaphore (increment
it to 1).
    return rc;                           // Return the retrieved value.
}
```



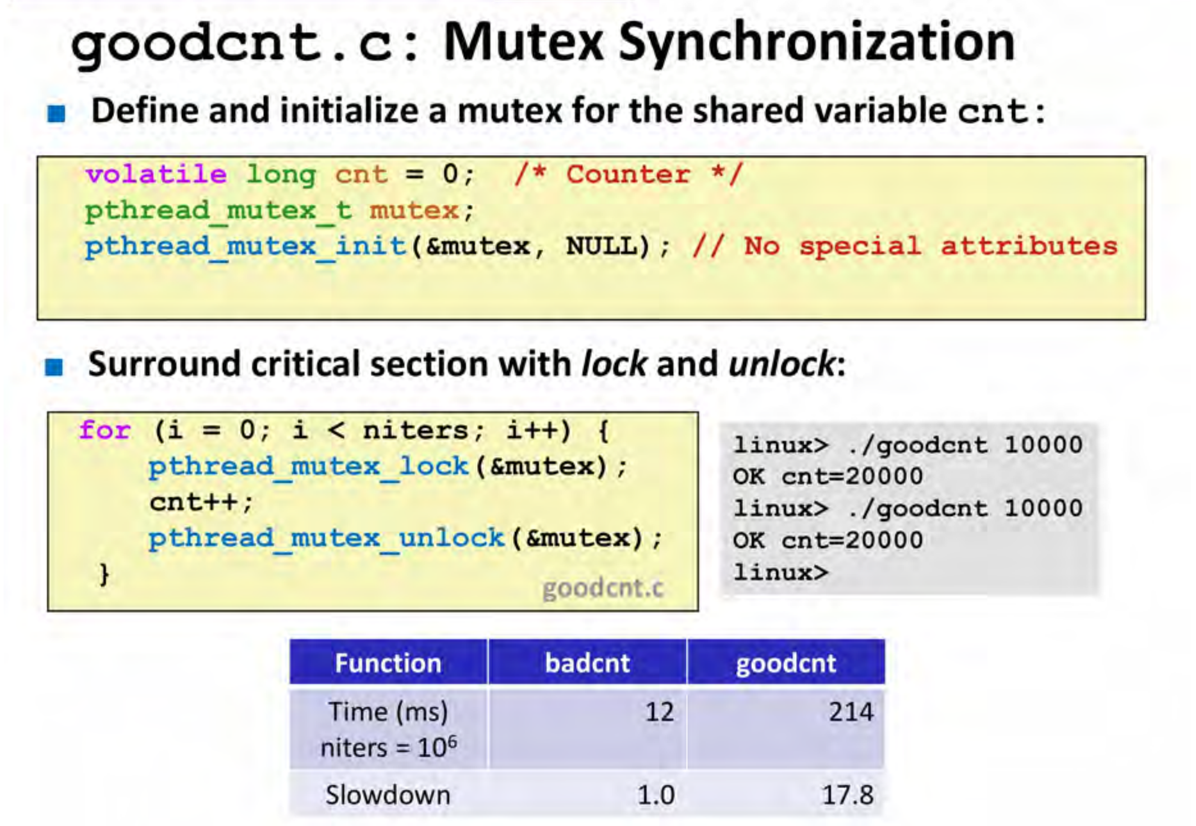
badcnt.c: Improper Synchronization

```
/* 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

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

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

    return NULL;
}
```

How can we fix this using semaphores?



goodcnt.c: Mutex Synchronization

- Define and initialize a mutex for the shared variable cnt:

```
volatile long cnt = 0;  /* Counter */
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL); // No special attributes
```

- Surround critical section with *lock* and *unlock*:

```
for (i = 0; i < niters; i++) {
    pthread_mutex_lock(&mutex);
    cnt++;
    pthread_mutex_unlock(&mutex);
 }
```

goodcnt.c

```
linux> ./goodcnt 10000
OK cnt=20000
linux> ./goodcnt 10000
OK cnt=20000
linux>
```

Function	badcnt	goodcnt
Time (ms) niters = 10^6	12	214
Slowdown	1.0	17.8

(3) 近似计数器

现在我们有了一个并发数据结构，接下来的问题就是性能了.
理想情况下，并发执行的多线程应该和单线程一样快，这种状态就称为**完美扩展** (perfect scaling)
换言之，虽然总工作量增多，但是并行执行后，完成任务的时间并没有增加.
但在基准程序上的测试表明我们实现的并发计数器的可扩展性不好——并发执行时性能下降很多.
这说明更多并发不一定更快:
如果方案带来了大量的开销 (例如，频繁地获取和释放锁)，那么高并发就没有什么意义.

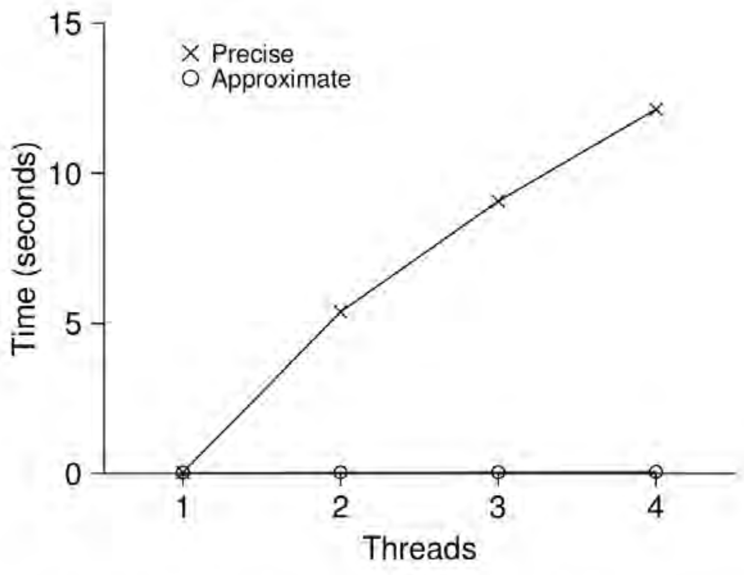


Figure 29.3: **Performance of Traditional vs. Approximate Counters**

一种改进版本是**近似计数器** (approximate counters)
它通过多个局部计数器和一个全局计数器来实现一个逻辑计数器.
其中每个 CPU 都有一个局部计数器和一把对应的锁，全局计数器也有一把锁.
如果一个 CPU 上的线程想增加计数器，那就增加它的局部计数器.
访问这个局部计数器是通过对应的局部锁同步的.
因为每个 CPU 有自己的局部计数器，不同 CPU 上的线程不会竞争，所以计数器的更新操作可扩展性好.
为了保持全局计数器更新 (以防某个线程要读取该值)，局部值会定期转移给全局计数器.
其方法是获取全局锁，让全局计数器加上局部计数器的值，然后将局部计数器置零.
这种局部转全局的频率取决于阈值 S (for sloppiness)
局部计数器的值达到 S 就转移到全局计数器，然后被置零.
阈值 S 越大，可拓展性越强，但是全局计数器与实际计数之间的误差越大.

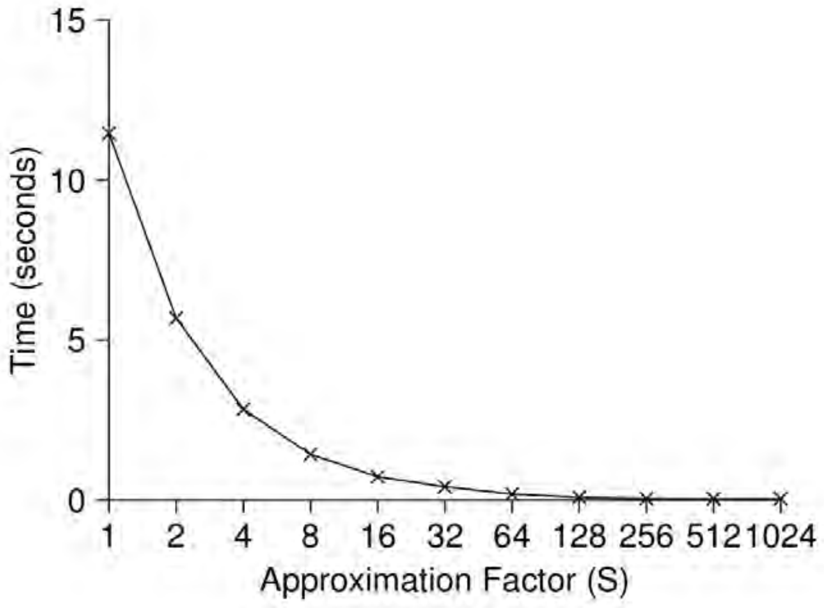


Figure 29.6: **Scaling Approximate Counters**

近似计数器的简单实现如下:

```
// Definition of a structure to represent an approximate counter.
```

```
typedef struct counter_t {
    int global;                     // Global counter to hold the approximate
total count.
    pthread_mutex_t glock;          // Mutex lock to protect access to the
global counter.
    int local[NUMCPUS];             // Array of local counters, one for each
CPU/thread.
    pthread_mutex_t llock[NUMCPUS];   // Array of mutex locks, one for each
local counter.
    int threshold;        // Threshold value to determine when local counters
update the global counter.
} counter_t;

// Initialization function for the approximate counter.
// Parameters:
// - c: Pointer to the counter structure to initialize.
// - threshold: The threshold value for local-to-global updates.
void init(counter_t *c, int threshold) {
    c->threshold = threshold;         // Store the threshold value.
    c->global = 0;                  // Initialize the global counter to 0.
    pthread_mutex_init(&c->glock, NULL); // Initialize the mutex lock for the
global counter.

    int i;                          // Loop variable for initializing local
counters and locks.
    for (i = 0; i < NUMCPUS; i++) {
        c->local[i] = 0;            // Initialize each local counter to 0.
        pthread_mutex_init(&c->llock[i], NULL); // Initialize the mutex lock for
each local counter.
    }
}

// Update function to increment the counter by a specified amount.
// Parameters:
// - c: Pointer to the counter structure.
// - threadID: The ID of the thread calling the function (used to access the
correct local counter).
// - amt: The amount to increment the counter by (assumes amt > 0).
void update(counter_t *c, int threadID, int amt) {
    pthread_mutex_lock(&c->llock[threadID]);   // Acquire the lock for the
thread's local counter.
    c->local[threadID] += amt;                // Increment the local counter by
the specified amount.

    // If the local counter exceeds the threshold, transfer its value to the
global counter.
    if (c->local[threadID] >= c->threshold) {
        pthread_mutex_lock(&c->glock);        // Acquire the lock for the global
counter.
        c->global += c->local[threadID];      // Add the local counter value to
the global counter.
        pthread_mutex_unlock(&c->glock);      // Release the global counter
lock.
        c->local[threadID] = 0;               // Reset the local counter to 0.
    }
    pthread_mutex_unlock(&c->llock[threadID]); // Release the lock for the local
counter.
}
```

```
// Get function to retrieve the current value of the global counter.
// This value may not be perfectly accurate due to delays in local-to-global
updates.
// Parameters:
// - c: Pointer to the counter structure.
// Returns:
// - The approximate value of the global counter.
int get(counter_t *c) {
    pthread_mutex_lock(&c->glock);    // Acquire the lock for the global
counter.
    int val = c->global;              // Read the value of the global counter.
    pthread_mutex_unlock(&c->glock);  // Release the lock for the global
counter.
    return val;                       // Return the approximate value.
}
```

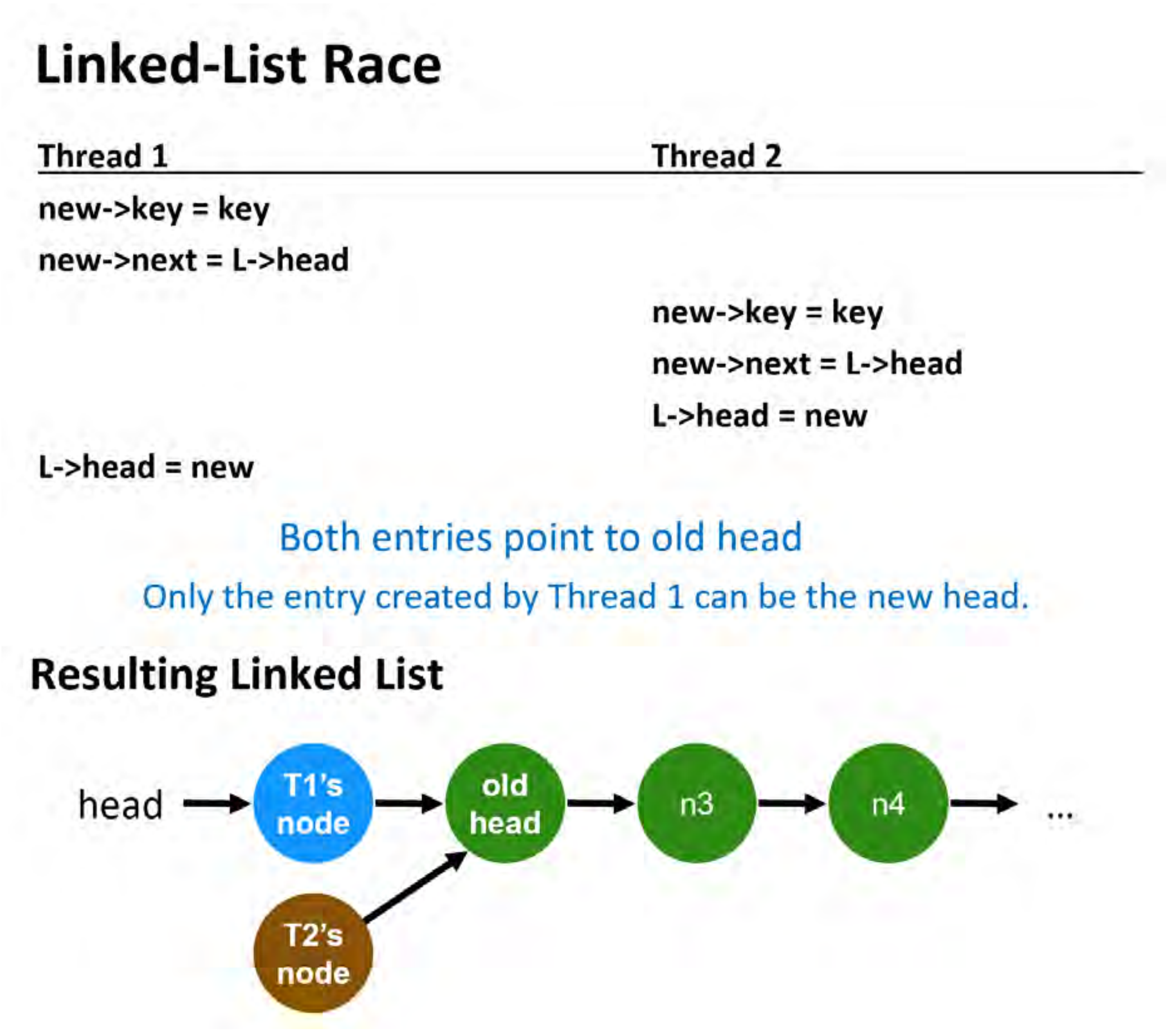
3.4.2 并发链表

(1) 非并发的实现

首先考虑一个非并发的链表:

```
// Definition of a node structure representing an element in the linked list.
typedef struct __node_t {
    int key;                  // The data stored in the node (integer key).
    struct __node_t *next;    // Pointer to the next node in the list.
} node_t;

// Definition of the linked list structure.
typedef struct __list_t {
    node_t *head;             // Pointer to the head (first node) of the list.
} list_t;

// Function to initialize the list.
// Sets the head of the list to NULL, indicating the list is empty.
void List_Init(list_t *L) {
    L->head = NULL;           // Empty list has no nodes, so head points to
NULL.
}

// Function to insert a new node with the given key at the beginning of the
list.
// Parameters:
// - L: Pointer to the list structure.
// - key: The integer key to be inserted.
void List_Insert(list_t *L, int key) {
    node_t *new = malloc(sizeof(node_t));  // Allocate memory for a new node.
    assert(new);                           // Ensure memory allocation was
successful.
    new->key = key;                        // Set the key value for the new
node.
    new->next = L->head;                   // Point the new node to the current
head.
    L->head = new;                         // Update the head to the new node.
```

```
}

// Function to look up a node with the given key in the list.
// Parameters:
// - L: Pointer to the list structure.
// - key: The integer key to search for.
// Returns:
// - 1 if the key is found, 0 otherwise.
int List_Lookup(list_t *L, int key) {
    node_t *tmp = L->head;                  // Start from the head of the list.
    while (tmp) {                           // Traverse the list until the end.
        if (tmp->key == key)                // If the key is found, return 1.
            return 1;
        tmp = tmp->next;                    // Move to the next node.
    }
    return 0;                               // Key not found, return 0.
}

// Function to delete a node with the given key from the list.
// Parameters:
// - L: Pointer to the list structure.
// - key: The integer key of the node to delete.
// Returns:
// - 1 if the key is found and deleted, 0 otherwise.
int List_Delete(list_t *L, int key) {
    node_t *tmp = L->head;                  // Start from the head of the list.
    node_t *prev = NULL;                    // Pointer to track the previous
node.

    while (tmp) {                           // Traverse the list to find the key.
        if (tmp->key == key) {              // If the key is found:
            if (prev) {                     // If it's not the first node:
                prev->next = tmp->next;     // Link the previous node to the next
node.
            } else {                        // If it's the first node:
                L->head = tmp->next;        // Update the head of the list.
            }
            free(tmp);                      // Free the memory for the deleted
node.
            return 1;                       // Return 1 to indicate successful
deletion.
        }
        prev = tmp;                         // Update the previous node pointer.
        tmp = tmp->next;                    // Move to the next node.
    }
    return 0;                               // Key not found, return 0.
}

// Function to update a node with the given key in the list.
// Parameters:
// - L: Pointer to the list structure.
// - old_key: The integer key to search for (the key to be updated).
// - new_key: The new integer key to replace the old one.
// Returns:
// - 1 if the key is found and updated, 0 otherwise.
int List_Update(list_t *L, int old_key, int new_key) {
    node_t *tmp = L->head;  // Start from the head of the list.
    while (tmp) {  // Traverse the list until the end.
        if (tmp->key == old_key) {  // If the key matches:
            tmp->key = new_key;  // Update the node's key to the new value.
            return 1;  // Return 1 to indicate successful update.
        }
        tmp = tmp->next;  // Move to the next node.
    }

    return 0;  // Return 0 if the key is not found.
}
```



Linked-List Race

Thread 1	Thread 2
new->key = key	
new->next = L->head	
	new->key = key
	new->next = L->head
	L->head = new
L->head = new	

Both entries point to old head

Only the entry created by Thread 1 can be the new head.

Resulting Linked List

(2) 并发化

现在我们引入锁:
(总有一个标准的方法来创建一个并发数据结构: 添加一把大锁)

- ① 插入操作显然是需要锁的，否则会出现不同插入操作之间表头连接的竞争 (如 (1) 中图片所示)
- ② 查找操作本身是只读的，不会改变链表中的数据结构，因此它本身不会引起竞态条件.
  插入操作只会影响表头，因此不会改变已查找的节点内容.
  删除操作会改变链表的结构，更新操作会改变节点的值，可能影响查找操作正在遍历的节点.
  如果没有删除操作和更新操作，那么我们不需要给查找操作上锁.
- ③ 删除或更新节点时，必须保证没有其他线程正在遍历该节点.
  对删除操作进行加锁可以防止其他线程在删除节点时访问无效内存.
  对更新操作进行加锁可以防止其他线程读出旧数据.

```
// Node structure representing an element in the linked list.
```

```c
typedef struct __node_t {
    int key;                    // The key stored in the node.
    struct __node_t *next;      // Pointer to the next node in the list.
} node_t;

// Linked-list structure.
typedef struct __list_t {
    node_t *head;               // Pointer to the head (first node) of the list.
    pthread_mutex_t lock;       // Mutex lock to synchronize access to the list.
} list_t;

// Initializes the list.
// Sets the head to NULL and initializes the mutex lock.
void List_Init(list_t *L) {
    L->head = NULL;                             // Initialize the head pointer to
NULL.
    pthread_mutex_init(&L->lock, NULL);         // Initialize the mutex lock.
}

// Function to insert a new node with the given key at the beginning of the
list.
// Parameters:
// - L: Pointer to the list structure.
// - key: The integer key to be inserted.
int List_Insert(list_t *L, int key) {
    pthread_mutex_lock(&L->lock);               // Acquire the lock.

    node_t *new = malloc(sizeof(node_t));       // Allocate memory for the new
node.
    if (new == NULL) {                          // Handle memory allocation
failure.
        perror("malloc");
        pthread_mutex_unlock(&L->lock);         // Release the lock before
returning.
        return -1;                              // Return -1 to indicate failure.
    }

    new->key = key;                             // Set the key for the new node.
    new->next = L->head;                        // Point the new node to the
current head.
    L->head = new;                              // Update the head to the new
node.

    pthread_mutex_unlock(&L->lock);             // Release the lock.
    return 0;                                   // Return 0 to indicate success.
}

// Function to look up a node with the given key in the list.
// Parameters:
// - L: Pointer to the list structure.
// - key: The integer key to search for.
// Returns:
// - 1 if the key is found, 0 otherwise.
int List_Lookup(list_t *L, int key) {
    pthread_mutex_lock(&L->lock);               // Acquire the lock.

    node_t *curr = L->head;                     // Start from the head of the
list.
```

```
    while (curr) {                              // Traverse the list.
        if (curr->key == key) {                 // If the key is found:
            pthread_mutex_unlock(&L->lock);     // Release the lock before
returning.
            return 0;                           // Return 0 to indicate success.
        }
        curr = curr->next;                      // Move to the next node.
    }

    pthread_mutex_unlock(&L->lock);             // Release the lock.
    return -1;                                  // Return -1 to indicate failure
(key not found).
}

// Function to delete a node with the given key from the list.
// Parameters:
// - L: Pointer to the list structure.
// - key: The integer key of the node to delete.
// Returns:
// - 1 if the key is found and deleted, 0 otherwise.
int List_Delete(list_t *L, int key) {
    pthread_mutex_lock(&L->lock);               // Acquire the lock.

    node_t *curr = L->head;                     // Start from the head of the
list.
    node_t *prev = NULL;                        // Pointer to track the previous
node.

    while (curr) {                              // Traverse the list to find the
key.
        if (curr->key == key) {                 // If the key is found:
            if (prev) {                         // If it's not the first node:
                prev->next = curr->next;        // Link the previous node to the
next node.
            } else {                            // If it's the first node:
                L->head = curr->next;           // Update the head to the next
node.
            }
            free(curr);                         // Free the memory of the deleted
node.
            pthread_mutex_unlock(&L->lock);     // Release the lock.
            return 0;                           // Return 0 to indicate success.
        }
        prev = curr;                            // Update the previous node
pointer.
        curr = curr->next;                      // Move to the next node.
    }

    pthread_mutex_unlock(&L->lock);             // Release the lock.
    return -1;                                  // Return -1 if the key is not
found.
}

// Function to update a node with the given key in the list.
// Parameters:
// - L: Pointer to the list structure.
// - old_key: The integer key to search for (the key to be updated).
// - new_key: The new integer key to replace the old one.
// Returns:
// - 1 if the key is found and updated, -1 otherwise.
int List_Update(list_t *L, int old_key, int new_key) {
    pthread_mutex_lock(&L->lock);  // Acquire the lock to protect the list
during the update operation.

    node_t *curr = L->head;        // Start from the head of the list.

    while (curr) {                 // Traverse the list.
        if (curr->key == old_key) {  // If the key matches:
            curr->key = new_key;     // Update the key of the node.
            pthread_mutex_unlock(&L->lock);  // Release the lock after updating.
            return 1;                        // Return 1 to indicate the key
was updated.
        }
        curr = curr->next;           // Move to the next node.
    }

    pthread_mutex_unlock(&L->lock);  // Release the lock if key was not found.
    return -1;                        // Return -1 if the key was not found.
}
```

(当心锁和控制流)
控制流的变化或其他错误情况可能导致函数退出或报错.
代码需要在返回或报错前恢复各种状态 (例如上述代码对 `malloc()` 失败情况的处理)
实际编程时，我们需要尽量避免这种模式.

(3) 调整

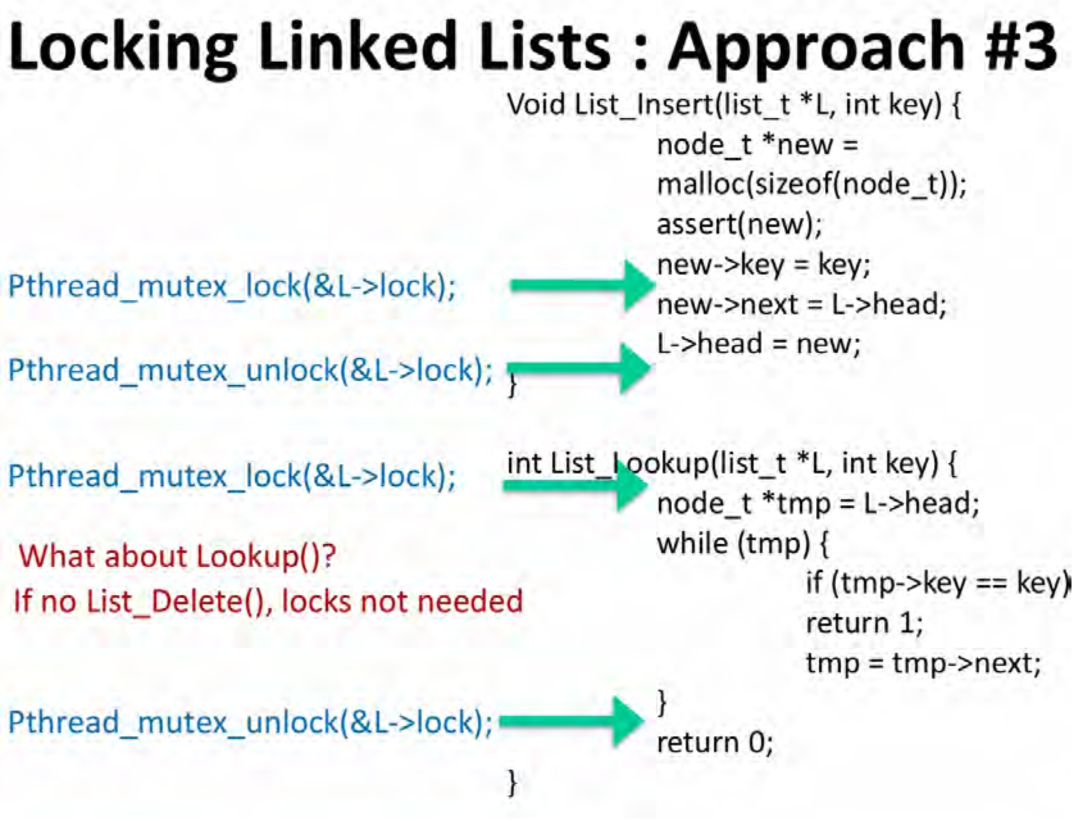
我们可以调整代码，让获取锁和释放锁的操作只环绕插入函数、查找函数和删除函数的真正临界区.
因为部分工作实际上不需要锁，例如 `malloc()` 函数是线程安全的,
每个线程都可以调用它，不需要担心竞争条件和其他并发缺陷.
我们只需在更新共享链表时需要持有锁即可.

- ① 插入函数调整如下:

```
// Function to insert a new node with the given key at the beginning of the
list.
// Parameters:
// - L: Pointer to the list structure.
// - key: The integer key to be inserted.
int List_Insert(list_t *L, int key) {
    node_t *new = malloc(sizeof(node_t));     // Allocate memory for the new
node.
    if (new == NULL) {                        // Handle memory allocation
failure.
        perror("malloc");
        return -1;                            // Return -1 to indicate
failure.
    }
    new->key = key;                           // Set the key for the new
node.

    // Just lock critical section
    pthread_mutex_lock(&L->lock);             // Acquire the lock.
    new->next = L->head;                      // Point the new node to the
current head.
```

```
    L->head = new;                          // Update the head to the new
node.
    pthread_mutex_unlock(&L->lock);         // Release the lock.
    return 0;                               // Return 0 to indicate
success.
}
```

- ② 查找函数、删除函数和更新函数不变，这是 "整个链表只有一把锁" 所造成的局限性 **(存疑)**



Locking Linked Lists : Approach #3

```
Void List_Insert(list_t *L, int key) {
        node_t *new =
        malloc(sizeof(node_t));
        assert(new);
        new->key = key;
        new->next = L->head;
        L->head = new;
}

int List_Lookup(list_t *L, int key) {
        node_t *tmp = L->head;
        while (tmp) {
                if (tmp->key == key)
                return 1;
                tmp = tmp->next;
        }
        return 0;
}
```

Pthread_mutex_lock(&L->lock);

Pthread_mutex_unlock(&L->lock);

Pthread_mutex_lock(&L->lock);

What about Lookup()?

If no List_Delete(), locks not needed

Pthread_mutex_unlock(&L->lock);

(4) 交替锁定

尽管我们有了基本的并发链表，但其在高并发场景下的性能很不好.
一种增加链表并发的技术称为**交替锁定** (hand-over-hand locking)
其基本思想很简单:
为每个节点都设置一把锁，而不是整个链表一把锁.
在遍历链表的时候， 首先抢占下一个节点的锁，然后释放当前节点的锁.
这种方案理论上可以增加并发:
线程不会因为其他线程在操作链表的某一部分而被阻塞.
它可以继续操作自己感兴趣的节点部分，而不需要等待对整个链表的锁定.
但实际应用中会带来获取和释放锁的巨大开销.
也许某种折中的方案 (即给一定数量的节点加一把锁) 是可行的.

3.4.3 并发队列

简单起见，我们直接来看并发队列的实现.

- 我们一共引入了两把锁，一个负责队列头，另一个负责队列尾.
  这两个锁使得入队列操作和出队列操作可以并发执行,
  因为入队列只访问 `tailLock` 锁，而出队列只访问 `headLock` 锁.

- 我们还在队列初始化中引入了一个假节点 `tmp`，它分开了头和尾操作.

```
// Node structure representing an element in the queue
typedef struct node_t {
    int value;                // The value stored in the node
    struct node_t *next;      // Pointer to the next node in the queue
} node_t;

// Queue structure with head, tail, and locks for synchronization
typedef struct queue_t {
    node_t *head;            // Pointer to the head (front) of the queue
    node_t *tail;            // Pointer to the tail (back) of the queue
    pthread_mutex_t headLock;  // Mutex lock to synchronize access to the head
    pthread_mutex_t tailLock;  // Mutex lock to synchronize access to the tail
} queue_t;

// Initializes the queue by allocating memory for the first node (head and tail)

// and setting up mutex locks for the head and tail.
void Queue_Init(queue_t *q) {
    node_t *tmp = malloc(sizeof(node_t));  // Allocate memory for a new node
    tmp->next = NULL;                      // Set the next pointer of the node
to NULL
    q->head = q->tail = tmp;               // Set both head and tail to point to
the new node
    pthread_mutex_init(&q->headLock, NULL); // Initialize mutex for head lock
    pthread_mutex_init(&q->tailLock, NULL); // Initialize mutex for tail lock
}

// Enqueues a new value to the back of the queue by creating a new node and
// updating the tail pointer. Uses tailLock to ensure synchronization.
void Queue_Enqueue(queue_t *q, int value) {
    node_t *tmp = malloc(sizeof(node_t));  // Allocate memory for the new node
    assert(tmp != NULL);                   // Ensure memory allocation was
successful
    tmp->value = value;                    // Set the value for the new node
    tmp->next = NULL;                      // Set the next pointer of the new
node to NULL

    pthread_mutex_lock(&q->tailLock);      // Acquire the tail lock to ensure
thread-safe access to the tail
    q->tail->next = tmp;                   // Link the current tail node to the
new node
    q->tail = tmp;                         // Update the tail pointer to the new
node
    pthread_mutex_unlock(&q->tailLock);    // Release the tail lock
}

// Dequeues the front value from the queue. The value is returned via the
// passed pointer, and the head pointer is updated. The headLock is used for
synchronization.
int Queue_Dequeue(queue_t *q, int *value) {
    pthread_mutex_lock(&q->headLock);      // Acquire the head lock to ensure
thread-safe access to the head
    node_t *tmp = q->head;                 // Store the current head node
    node_t *newHead = tmp->next;           // Store the next node after the
current head
```

```
    if (newHead == NULL) {                 // If the queue is empty (next node
is NULL)
        pthread_mutex_unlock(&q->headLock); // Release the head lock
        return -1;                          // Return -1 to indicate that the
queue was empty
    }

    *value = newHead->value;               // Get the value from the new head
node
    q->head = newHead;                     // Update the head pointer to the new
head node
    pthread_mutex_unlock(&q->headLock);    // Release the head lock

    free(tmp);                             // Free the memory of the old head
node
    return 0;                              // Return 0 to indicate success
}
```

3.4.4 并发 Hash 表

简单起见，考虑一个不需要调整大小的并发 Hash 表:
(借助 3.4.2 的并发链表)

```
// Define the number of buckets in the hash table
#define BUCKETS (101)

// Define the hash table structure
// It consists of an array of lists, each representing a bucket in the hash
table.
typedef struct hash_t {
    list_t lists[BUCKETS];  // Array of list_t structures, each representing a
bucket in the hash table
} hash_t;

// Initialize the hash table by initializing each bucket's list
void Hash_Init(hash_t *H) {
    int i;
    // Loop through each bucket and initialize the corresponding list
    for (i = 0; i < BUCKETS; i++) {
        List_Init(&H->lists[i]);  // Initialize the list for each bucket
    }
}

// Insert a key into the hash table
// The key is hashed to determine the appropriate bucket, and then inserted into
the corresponding list
int Hash_Insert(hash_t *H, int key) {
    int bucket = key % BUCKETS;  // Compute the bucket index using modulo
operation
    return List_Insert(&H->lists[bucket], key);  // Insert the key into the
appropriate bucket's list
}

// Lookup a key in the hash table
// The key is hashed to determine the appropriate bucket, and then searched in the corresponding list
int Hash_Lookup(hash_t *H, int key) {
    int bucket = key % BUCKETS;  // Compute the bucket index using modulo operation
    return List_Lookup(&H->lists[bucket], key);  // Search for the key in the appropriate bucket's list
}
```

更多内容参考 Homework 2.

3.5 条件变量

3.5.1 定义

到目前为止，我们已经形成了锁的概念，看到了如何通过硬件和操作系统支持的正确组合来实现锁.
然而锁并不是并发程序设计所需的唯一原语.
在很多情况下，线程直到某一条件满足之后才会继续运行.
例如，父线程需要检查子线程是否执行完毕 (这常被称为 `join()`)
这种等待如何实现呢?

我们可以尝试用一个共享变量 `done`:

```
// Declare 'done' as a volatile integer, initialized to 0.
// The volatile keyword is used here to indicate to the compiler that 'done' may be changed unexpectedly (e.g., by a different thread).
volatile int done = 0;

void *child(void *arg) {
    // Print "child" to the console, indicating that the child thread has started executing.
    printf("child\n");
    done = 1;          // Set 'done' to 1, signaling that the child thread is finished.
    return NULL;       // Exit the child thread.
}

int main(int argc, char *argv[]) {
    // Print "parent: begin" to indicate that the main function (parent thread) has started.
    printf("parent: begin\n");

    // Declare a pthread variable 'c' to hold the child thread's identifier.
    pthread_t c;

    // Create the child thread, running the 'child' function with NULL as an argument.
    Pthread_create(&c, NULL, child, NULL);

    while (done == 0) // Loop until 'done' is set to 1 by the child thread.
        ;             // Busy-wait (spinning) while 'done' is 0.

    // Print "parent: end" to indicate that the parent thread has detected 'done' is now 1 and will end.
    printf("parent: end\n");
    return 0; // Exit the program successfully.
}
```

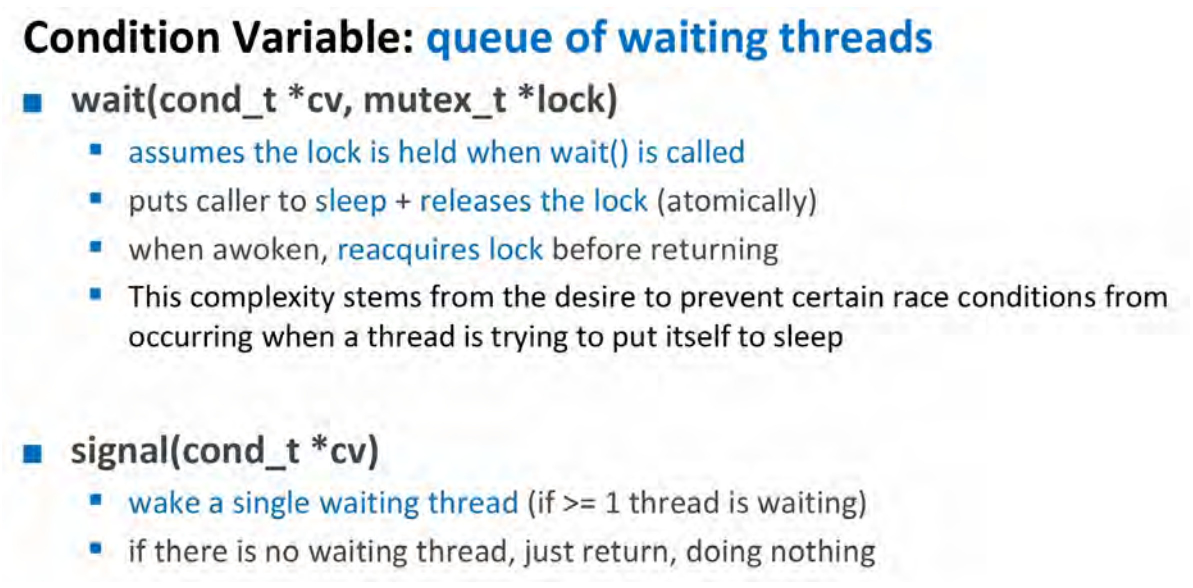
预期的运行结果为:

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

这种解决方案一般能工作，但是效率低下，因为父线程会自旋检查，浪费 CPU 时间.
我们希望有某种方式让父线程休眠，直到等待的条件满足 (即子线程完成执行)

线程可以使用**条件变量** (condition variable)，来等待一个条件变成真.
条件变量是一个显式队列，当某些执行状态 (即条件) 不满足时，线程可以把自己加入队列，等待该条件.
当某个线程改变了上述状态时，就可以通过在该条件上发信号唤醒一个或者多个等待线程，让它们继续执行.

我们可以使用 `pthread_cond_t c` 来声明一个条件变量 `c` (还需要适当的初始化)
条件变量有两种相关操作: `pthread_cond_wait()` (简称 `wait()`) 和 `pthread_cond_signal()` (简称 `signal()`)
当线程要睡眠的时候，就调用 `wait()`
当线程想唤醒等待在某个条件变量上的睡眠线程时，就调用 `signal()`



Condition Variable: queue of waiting threads

- wait(cond_t *cv, mutex_t *lock)
  - assumes the lock is held when wait() is called
  - puts caller to sleep + releases the lock (atomically)
  - when awoken, reacquires lock before returning
  - This complexity stems from the desire to prevent certain race conditions from occurring when a thread is trying to put itself to sleep
- signal(cond_t *cv)
  - wake a single waiting thread (if >= 1 thread is waiting)
  - if there is no waiting thread, just return, doing nothing

---

下面我们使用条件变量 `c` 和共享变量 `done` 让父进程等待子进程执行完成:

```
// Declare a condition variable and a mutex for thread synchronization
pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
pthread_cond_signal(pthread_cond_t  *c);

// Initialize a shared variable that indicates task completion status
int done = 0;

// Initialize a mutex to guard access to the shared variable
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

// Initialize a condition variable to allow threads to signal each other
```

```
pthread_cond_t c = PTHREAD_COND_INITIALIZER;

// Function to signal that the task is complete
void thr_exit() {
    Pthread_mutex_lock(&m);      // Lock the mutex to ensure exclusive access to
'done'
    done = 1;                     // Set 'done' to 1, indicating the task is
complete
    Pthread_cond_signal(&c);      // Signal any waiting thread that the
condition has changed
    Pthread_mutex_unlock(&m);     // Unlock the mutex after updating the shared
variable
}

// Child thread function, executes task, then signals completion
void *child(void *arg) {
    printf("child\n");            // Print a message to indicate the child
thread has started
    thr_exit();                   // Call thr_exit() to signal task completion
    return NULL;                  // Exit the child thread
}

// Function for the parent thread to wait until the child thread signals
completion
void thr_join() {
    Pthread_mutex_lock(&m);       // Lock the mutex to access 'done' safely
    while (done == 0)             // Loop to wait until 'done' is set to 1
        Pthread_cond_wait(&c, &m);// Wait for the condition variable signal,
releasing the mutex temporarily
    Pthread_mutex_unlock(&m);     // Unlock the mutex after the condition is met
}

// Main function to create the child thread and wait for its completion
int main(int argc, char *argv[]) {
    printf("parent: begin\n");    // Print message indicating the parent thread
has started
    pthread_t p;                  // Declare a thread identifier
    Pthread_create(&p, NULL, child, NULL); // Create the child thread, running
'child' function
    thr_join();                   // Wait for the child thread to complete by
calling thr_join
    printf("parent: end\n");      // Print message indicating the parent thread
is ending
    return 0;                     // End the main program
}
```

我们发现:

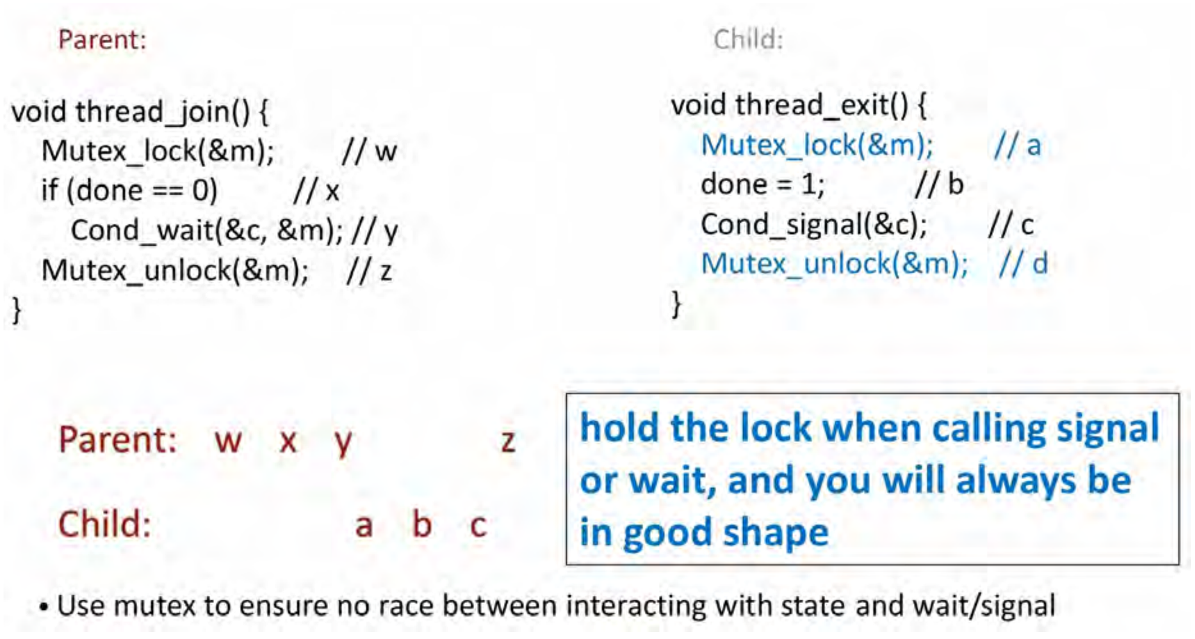
- `pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);`
  其调用有一个互斥量参数，它假定在 `wait()` 调用时，这个互斥量是已上锁状态 (这是强制要求的)
  `wait()` 的职责是 (原子地) 释放锁，并让调用线程休眠.
  当线程被唤醒时 (在另外某个线程发信号给它后)，它必须重新获取锁，再返回到调用它的线程.

  对应地，请在调用 `signal()` 时持有锁.
  尽管并不是所有情况下都严格需要，但最有效且简单的做法，还是在使用条件变量发送信号时持有锁.
- 假设只有一个处理器，则上述代码执行时会出现两种情况:

① 第一种情况是，父线程创建出子线程后自己继续运行，然后马上调用 `thr_join()` 等待子线程.
在这种情况下，它会先获取锁，检查子进程是否完成 (还没有完成)，然后调用 `wait()`，让自己休眠.
子线程最终得以运行，打印出 `"child"`，并调用 `thr_exit()` 函数唤醒父进程.
其中 `thr_exit()` 函数会在获得锁后设置状态变量 `done` 为 `1`， 然后向父线程发信号唤醒它.
最后父线程从 `wait()` 调用返回并继续运行，打印出 `"parent: end"` 后返回 `0`.

② 第二种情况是，父线程创建出子线程后直接运行子线程.
子线程打印出 `"child"`，并调用 `thr_exit()` 函数唤醒等待在条件变量 `c` 上的线程 (但没有等待的线程)，于是结束.
最后父线程恢复运行后，调用 `thr_join()` 时会发现 `done` 已经是 `1` 了，
因此无需等待，直接打印 `"parent: end"` 后返回 `0`

- 值得注意的是，父线程使用 `while` 循环而不是 `if` 语句来判断是否需要等待.
  虽然从逻辑上来说没有必要使用循环语句，但这样做总是好的 (后面我们会加以说明)



Parent:

```
void thread_join() {
   Mutex_lock(&m);        // w
   if (done == 0)        // x
      Cond_wait(&c, &m); // y
   Mutex_unlock(&m);     // z
}
```

Child:

```
void thread_exit() {
   Mutex_lock(&m);       // a
   done = 1;          // b
   Cond_signal(&c);       // c
   Mutex_unlock(&m);   // d
}
```

```
Parent:   w   x   y          z

Child:               a   b   c
```

hold the lock when calling signal or wait, and you will always be in good shape

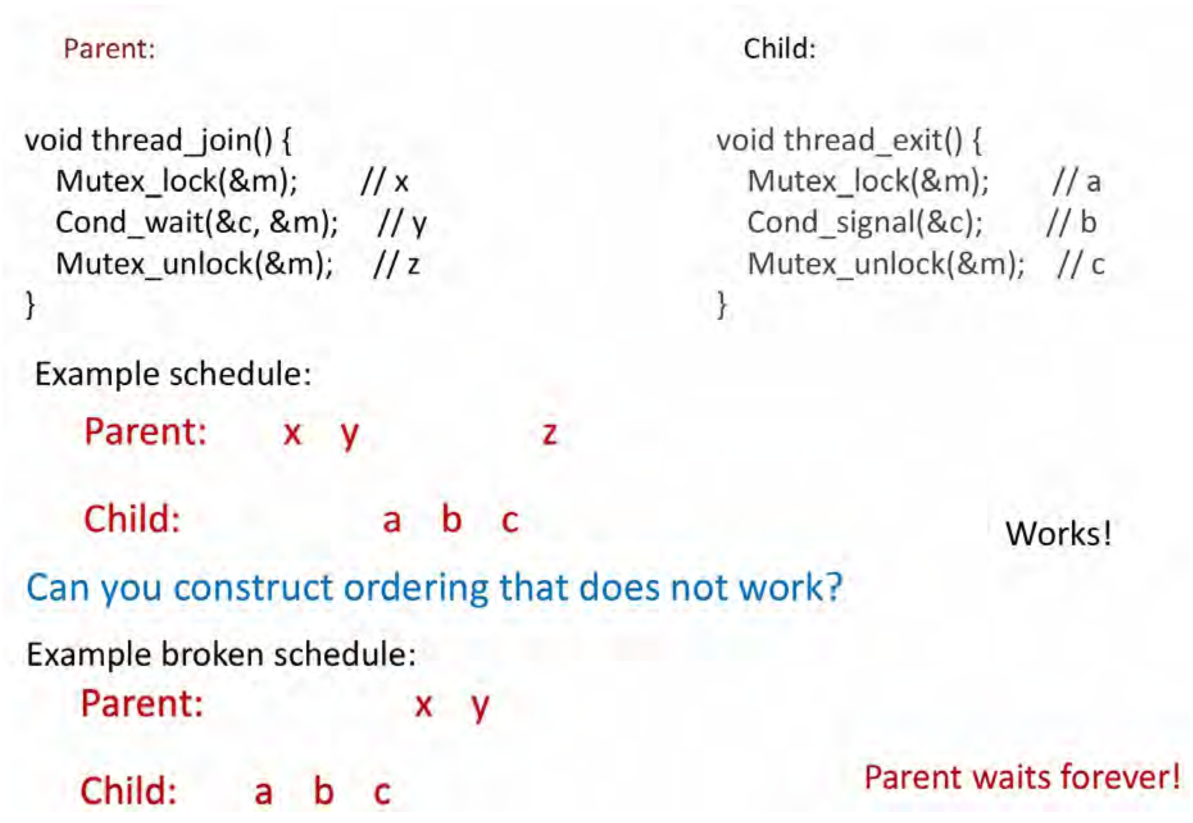
- Use mutex to ensure no race between interacting with state and wait/signal

---

共享变量 `done` 是不可缺少的，它记录了线程感兴趣的值，线程的睡眠、唤醒和锁都离不开它.
如果删去共享变量 `done`，则代码变为:

```
void thr_exit() {
    Pthread_mutex_lock(&m);       // Lock the mutex to synchronize access
    Pthread_cond_signal(&c);      // Signal the condition variable, notifying
waiting threads
    Pthread_mutex_unlock(&m);     // Unlock the mutex after signaling
}

void thr_join() {
    Pthread_mutex_lock(&m);       // Lock the mutex before waiting
    // Wait for the signal on the condition variable and release the mutex while
waiting
    Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);     // Unlock the mutex once signaled (after
waking up)
}
```

假设父线程创建出子线程后直接运行子线程，并且调用 `thr_exit()`
在这种情况下，子线程调用 `signal()` 发送信号，但此时却没有在条件变量上睡眠等待的线程.
父线程运行时，就会调用 `wait()` 并卡在那里，没有其他线程会唤醒它 (相当于唤醒信号丢失了)



Parent:

```
void thread_join() {
    Mutex_lock(&m);       // x
    Cond_wait(&c, &m);    // y
    Mutex_unlock(&m);     // z
}
```

Child:

```
void thread_exit() {
    Mutex_lock(&m);       // a
    Cond_signal(&c);      // b
    Mutex_unlock(&m);     // c
}
```

Example schedule:

```
Parent:   x  y        z

Child:         a  b  c              Works!
```

Can you construct ordering that does not work?

Example broken schedule:

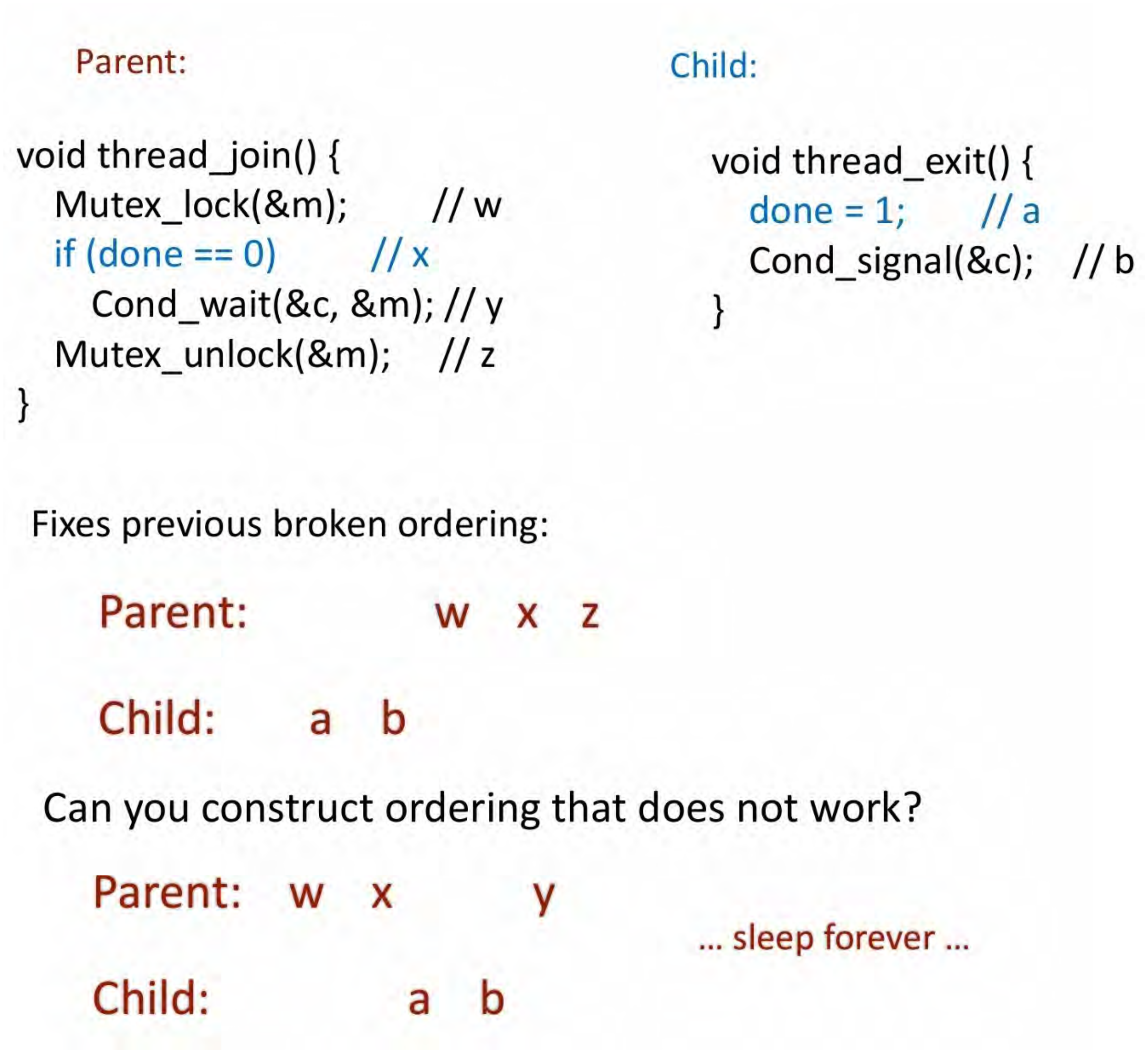
```
Parent:            x  y

Child:     a  b  c                  Parent waits forever!
```

锁 `m` 也是不可缺少的，它用于处理线程睡眠和唤醒的竞态条件.
假设线程在发信号和等待时都不加锁:

```
void thr_exit() {
    done = 1;                  // Set 'done' to 1 to indicate task completion,
but without mutex protection
    Pthread_cond_signal(&c);   // Signal the condition variable, notifying any
waiting thread
}

void thr_join() {
    if (done == 0)             // Check 'done' without holding a mutex lock,
making this check unreliable
        Pthread_cond_wait(&c);// Wait for the signal, but no mutex is passed,
which will lead to an error
}
```

假设父线程创建出子线程后自己继续运行，调用 `thr_join()` 检查发现 `done` 的值为 `0`，然后试图睡眠.
假设在调用 `wait()` 进入睡眠之前，父进程被中断了.
随后子线程开始运行，并修改变量 `done` 为 `1`，调用 `signal()` 发出唤醒等待在条件变量 `c` 上的线程 (但没有等待的线程)
等到父线程再次运行时，就会长眠不醒，这就惨了.

只给父进程加锁也是不行的:



Parent:

```
void thread_join() {
    Mutex_lock(&m);         // w
    if (done == 0)          // x
        Cond_wait(&c, &m); // y
    Mutex_unlock(&m);      // z
}
```

Child:

```
void thread_exit() {
    done = 1;        // a
    Cond_signal(&c);    // b
}
```

Fixes previous broken ordering:

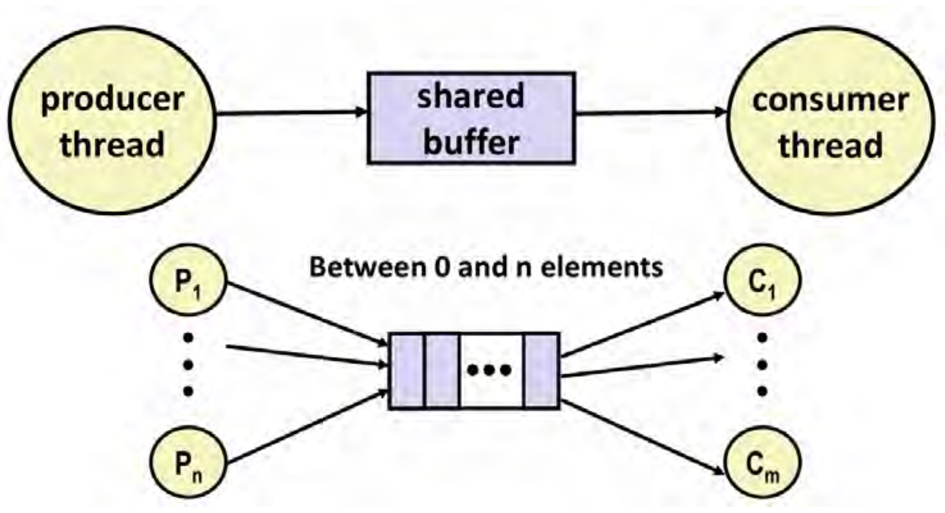
```
Parent:          w   x   z

Child:     a   b
```

Can you construct ordering that does not work?

```
Parent:   w   x        y
                                  ... sleep forever ...
Child:              a   b
```

3.5.2 生产者/消费者问题

我们要面对的下一个问题，是**生产者/消费者** (producer/consumer) 问题，也称为**有界缓冲区** (bounded buffer) 问题.
假设有若干个生产者线程和若干个消费者线程.
生产者把生成的数据放入缓冲区，而消费者从缓冲区取走数据，以某种方式消费.



但它可以用无锁的方式实现 **(期末可能会考?)**
[基于无锁队列的单生产者单消费者模型]()

首先我们需要一个共享缓冲区，让生产者放入数据，消费者取出数据.
简单起见，我们就拿一个整型数据的存储单元来做缓冲区 (稍后我们会一般化，用队列保存更多数据项)
内部函数 `put()` 将值放入缓冲区，内部函数 `get()` 从缓冲区取值.
由于有界缓冲区是共享资源，故我们必须通过同步机制来访问它，以免产生竞态条件.

(1) 首次尝试

首次尝试不可避免地会出现问题.
假设 `put()` 和 `get()` 函数定义如下:

```
int buffer;           // Shared buffer to hold a single integer value
int count = 0;        // Buffer state indicator; 0 means empty, 1 means full

// Function to add a value to the buffer
void put(int value) {
    assert(count == 0); // Ensure the buffer is empty before putting a value; if
not, the program will halt
    count = 1;          // Set count to 1, indicating the buffer is now full
    buffer = value;     // Store the input value in the buffer
}

// Function to retrieve the value from the buffer
int get() {
    assert(count == 1); // Ensure the buffer is full before getting a value; if
not, the program will halt
    count = 0;          // Set count to 0, indicating the buffer is now empty
    return buffer;      // Return the value stored in the buffer
}
```

`put()` 函数会假设缓冲区是空的，把一个值存在缓冲区，然后把 `count` 设置为 `1` 表示缓冲区满了.
`get()` 函数刚好相反，把缓冲区清空后 (即将 `count` 设置为 `0`)，并返回缓冲区的值.

现在我们需要编写一些函数，知道何时可以访问缓冲区，以便将数据放入缓冲区或从缓冲区取出数据.
条件是显而易见的:
仅在 `count` 为 `0` 时 (即缓冲器为空时)，才将数据放入缓冲器中;
仅在 `count` 为 `1` 时 (即缓冲器已满时)，才从缓冲器取出数据.
上述任务将由两种类型的线程完成，其中一类称为**生产者线程**，另一类称为**消费者线程**.

显然 `put()` 和 `get()` 函数之中会有临界区，因为 `put()` 更新缓冲区而 `get()` 读取缓冲区.
但是单纯给 `put()` 和 `get()` 函数加锁是没有用的，我们需要引入某些条件变量.

```
cond_t cond;                // Condition variable for signaling between producer
and consumer
mutex_t mutex;              // Mutex to synchronize access to the shared buffer
and condition variable

// Producer function that adds items to the buffer
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++)
    {
        Pthread_mutex_lock(&mutex); // p1: Lock the mutex to ensure exclusive
access to shared resources

        if (count == 1)             // p2: Check if the buffer is full (assuming
a single-item buffer)
```

```
            // p3: If full, wait for the condition signal, releasing the mutex
            Pthread_cond_wait(&cond, &mutex);

        put(i);                             // p4: Add an item to the buffer
(producing an item)
        Pthread_cond_signal(&cond);         // p5: Signal the consumer that an
item is available
        Pthread_mutex_unlock(&mutex);       // p6: Unlock the mutex to allow
consumer access
    }
}

// Consumer function that retrieves items from the buffer
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++)
    {
        Pthread_mutex_lock(&mutex);  // c1: Lock the mutex to ensure exclusive
access to shared resources

        if (count == 0)                     // c2: Check if the buffer is
empty
            // c3: If empty, wait for the producer's signal, releasing the mutex
            Pthread_cond_wait(&cond, &mutex);

        int tmp = get();                    // c4: Retrieve an item from the
buffer (consuming an item)
        Pthread_cond_signal(&cond);         // c5: Signal the producer that
the buffer has space
        Pthread_mutex_unlock(&mutex);       // c6: Unlock the mutex to allow
producer access

        printf("%d\n", tmp);         // Print the consumed item
    }
}
```

生产者和消费者之间的信号逻辑如下:

- 当生产者想要填充缓冲区时，它等待缓冲区变空 (`p1~p3`)
- 消费者具有完全相同的逻辑，但等待不同的条件——它等待缓冲区变满 (`c1~c3`)

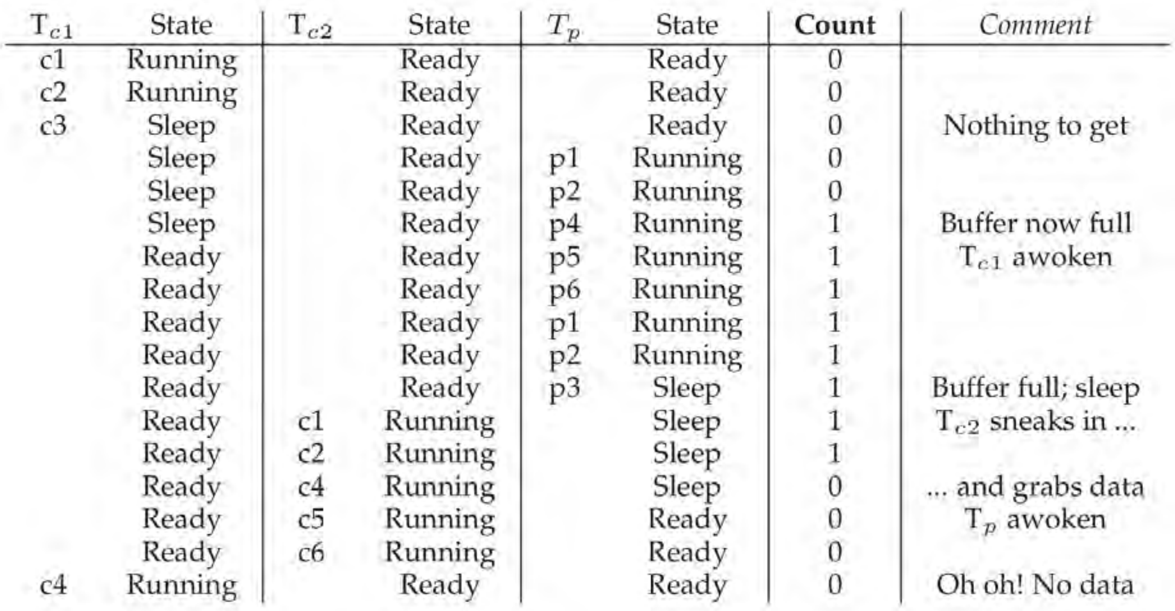
当只有一个生产者和一个消费者时上述代码能够正常运行.
但如果有多个生产者或多个消费者，那么上述代码就会出现两个严重的问题.
我们首先来理解第一个问题，它与 `wait()` 所在的 `if` 语句有关.

假设有两个消费者 (记为 T_{c_1} 和 T_{c_2}) 和一个生产者 (记为 T_p)
假设第一个消费者 T_{c_1} 首先执行，它会获取锁 (`c1`)，检查缓冲区是否已满 (`c2`)，发现缓冲区为空后开始等待 (`c3`) (会释放锁)
假设生产者 T_p 接着执行，它会获取锁 (`p1`)，检查缓冲区是否为空 (`p2`)，
发现缓冲区为空后调用 `put()` 填入一个整数 (`p4`) (并设置 `count` 为 `1`)
然后发出信号，说缓冲区已满 (`p5`)，最后释放锁 (`p6`)
此时第一个消费者 T_{c_1} 不再睡在条件变量上，而是进入就绪队列
生产者 T_p 进入下一个循环，它会获取锁 (`p1`)，检查缓冲区是否为空 (`p2`)，发现缓冲区已满后开始等待 (`p3`) (会释放锁)

这时问题发生了:
假设第二个消费者 T_{c_2} 抢先执行，它会获取锁 (`c1`)，检查缓冲区是否已满 (`c2`)，
发现缓冲区已满后取出一个整数并清空缓冲区 (`c4`) (即设置 `count` 为 `0`)
然后发出信号，说缓冲区变空 (`c5`)，最后释放锁 (`c6`)
此时生产者 T_p 不再睡在条件变量上，而是进入就绪队列.

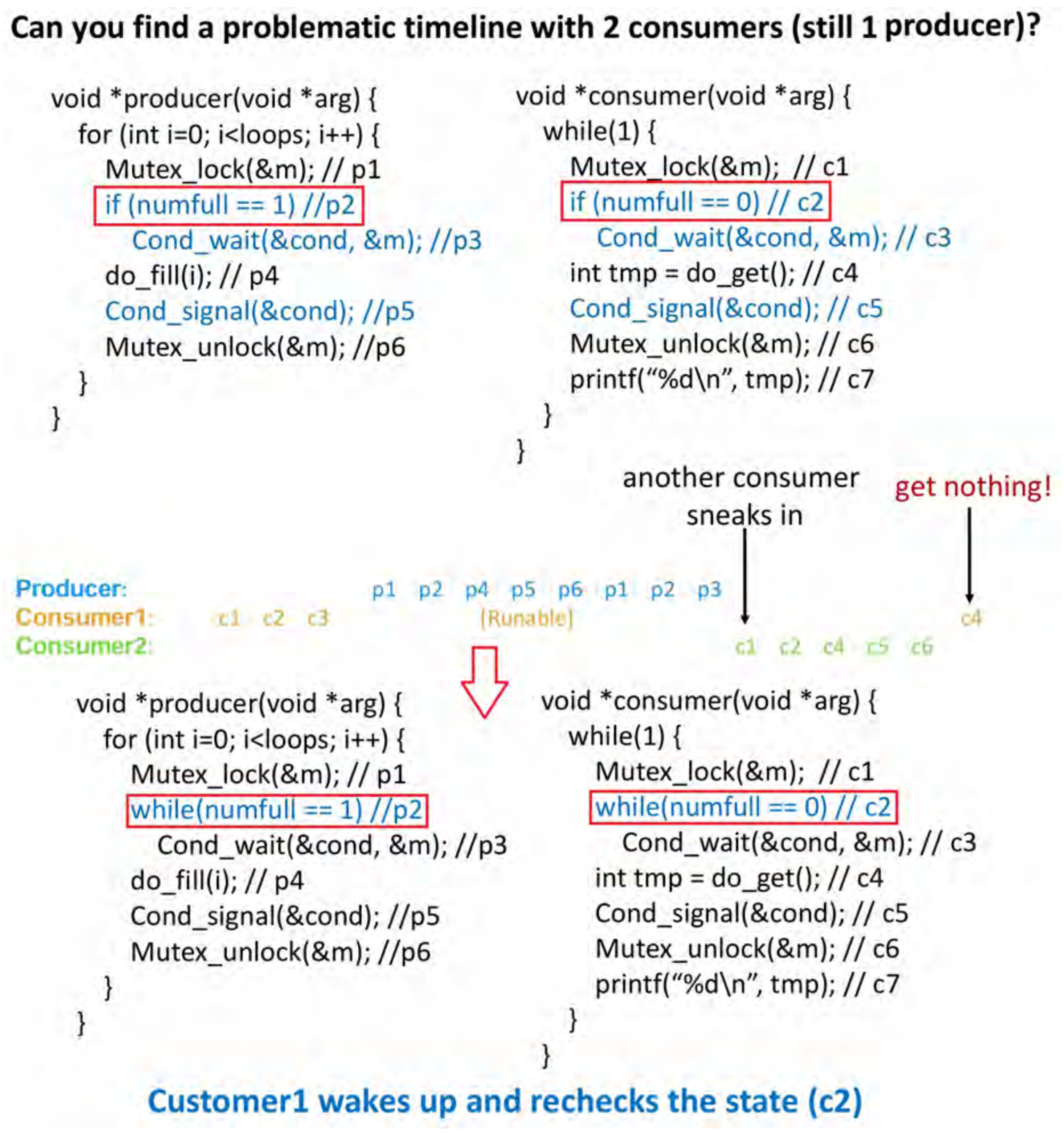
假设第一个消费者 T_{c_1} 继续执行，它会从 `c3` 的 `wait()` 返回 (重新持有锁)，
然后调用 `get()` 从缓冲区取一个整数 (`c4`)，但缓冲区已无法消费!
此时 `get()` 函数的断言 `assert(count == 1)` 触发，程序中止.

问题的原因很简单:
第一个消费者 T_{c_1} 在被生产者 T_p 唤醒后，它的确进入了就绪队列等待执行.
但在它运行之前，缓冲区的状态发生了改变 (由于第二个消费者 T_{c_2} 插队，抢先消费了本该由第一个消费者 T_{c_1} 消费的数据!)
发信号给线程只是唤醒它们，暗示状态发生了变化 (在这个例子中，就是值已被放入缓冲区)，
但并不会保证在它运行之前状态一直是期望的情况.
信号的这种释义常称为 **Mesa 语义**，几乎所有系统都采用了 Mesa 语义.
信号的另一种释义是 **Hoare 语义**，虽然实现难度大，但是会保证被唤醒的线程立刻执行.



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

Figure 30.7: **Thread Trace: Broken Solution (Version 1)**


Can you find a problematic timeline with 2 consumers (still 1 producer)?

```
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m); // p1
        if (numfull == 1) //p2
            Cond_wait(&cond, &m); //p3
        do_fill(i); // p4
        Cond_signal(&cond); //p5
        Mutex_unlock(&m); //p6
    }
}
```

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);  // c1
        if (numfull == 0) // c2
            Cond_wait(&cond, &m); // c3
        int tmp = do_get(); // c4
        Cond_signal(&cond); // c5
        Mutex_unlock(&m); // c6
        printf("%d\n", tmp); // c7
    }
}
```

```
Producer:                    p1  p2  p4  p5  p6  p1  p2  p3
Consumer1:   c1  c2  c3                (Runable)                              c4
Consumer2:                                           c1  c2  c4  c5  c6
```

```
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m); // p1
        while(numfull == 1) //p2
            Cond_wait(&cond, &m); //p3
        do_fill(i); // p4
        Cond_signal(&cond); //p5
        Mutex_unlock(&m); //p6
    }
}
```

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);  // c1
        while(numfull == 0) // c2
            Cond_wait(&cond, &m); // c3
        int tmp = do_get(); // c4
        Cond_signal(&cond); // c5
        Mutex_unlock(&m); // c6
        printf("%d\n", tmp); // c7
    }
}
```

Customer1 wakes up and rechecks the state (c2)

(2) 再次尝试

修复第一个问题的方法很简单: 把 `if` 语句改为 `while` 语句
在 Mesa 语义下，我们要记住一条关于条件变量的简单规则: **总是使用 `while` 循环**
虽然多线程程序有时候不需要重新检查条件，但使用 `while` 循环总是安全的.
这也解决了**假唤醒** (spurious wakeup) 的情况.
(某些线程库中，由于实现细节的缘故，有可能出现一个信号唤醒两个线程的情况)

当消费者 T_{c_1} 被唤醒后，立刻再次检查缓存区是否已满 (`c2`)
如果缓冲区此时为空，消费者就会回去继续睡眠 (`c3`)
生产者也相应地将 `if` 语句改为 `while` 语句 (`p2`)

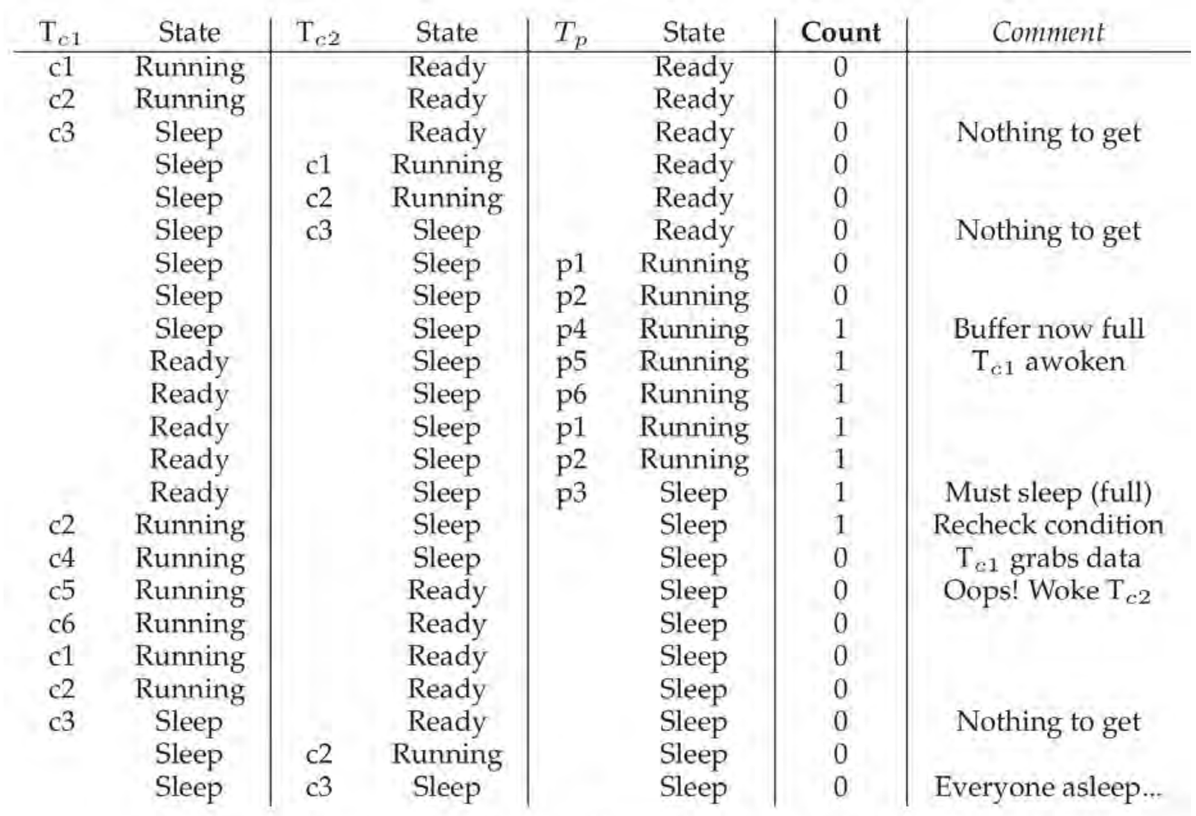
```
cond_t cond;                // Condition variable for signaling between producer
and consumer
mutex_t mutex;              // Mutex to synchronize access to the shared buffer
and condition variable

// Producer function that adds items to the buffer
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++)
    {
        Pthread_mutex_lock(&mutex); // p1: Lock the mutex to ensure exclusive
access to shared resources

        while (count == 1)             // p2: Check if the buffer is full
(assuming a single-item buffer)
            // p3: If full, wait for the condition signal, releasing the mutex
            Pthread_cond_wait(&cond, &mutex);

        put(i);                               // p4: Add an item to the buffer
(producing an item)
        Pthread_cond_signal(&cond);           // p5: Signal the consumer that an
item is available
        Pthread_mutex_unlock(&mutex);         // p6: Unlock the mutex to allow
consumer access
    }
}

// Consumer function that retrieves items from the buffer
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++)
    {
        Pthread_mutex_lock(&mutex);  // c1: Lock the mutex to ensure exclusive
access to shared resources

        while (count == 0)                      // c2: Check if the buffer is
empty
            // c3: If empty, wait for the producer's signal, releasing the mutex
            Pthread_cond_wait(&cond, &mutex);

        int tmp = get();                      // c4: Retrieve an item from the
buffer (consuming an item)
        Pthread_cond_signal(&cond);           // c5: Signal the producer that
the buffer has space
        Pthread_mutex_unlock(&mutex);         // c6: Unlock the mutex to allow
producer access

        printf("%d\n", tmp);         // Print the consumed item
    }
}
```

但上述代码仍然有一个问题，它和我们只用了一个条件变量有关.
假设两个消费者 T_{c_1} 和 T_{c_2} 先运行，都因为缓存区为空而睡眠了 (`c3`)
生产者开始运行，在缓冲区放入一个值，并唤醒了一个消费者 (假定是 T_{c_1})，然后开始睡眠.
现在消费者 T_{c_1} 即将运行，线程 T_{c_2} 和 T_p 都等待在同一个条件变量上.

消费者 T_{c_1} 醒过来并从 `wait()` 调用返回 (`c3`)，重新检查缓存区是否已满 (`c2`)，
发现缓冲区是满的，于是消费了缓冲区中的值 (`c4`) 并将缓存区清空 (设置 `count` 为 `1`)
然后在该条件上发信号 (`c5`)，唤醒一个正在睡眠的线程 (T_{c_2} 或 T_p).
但是应该唤醒哪个线程呢?

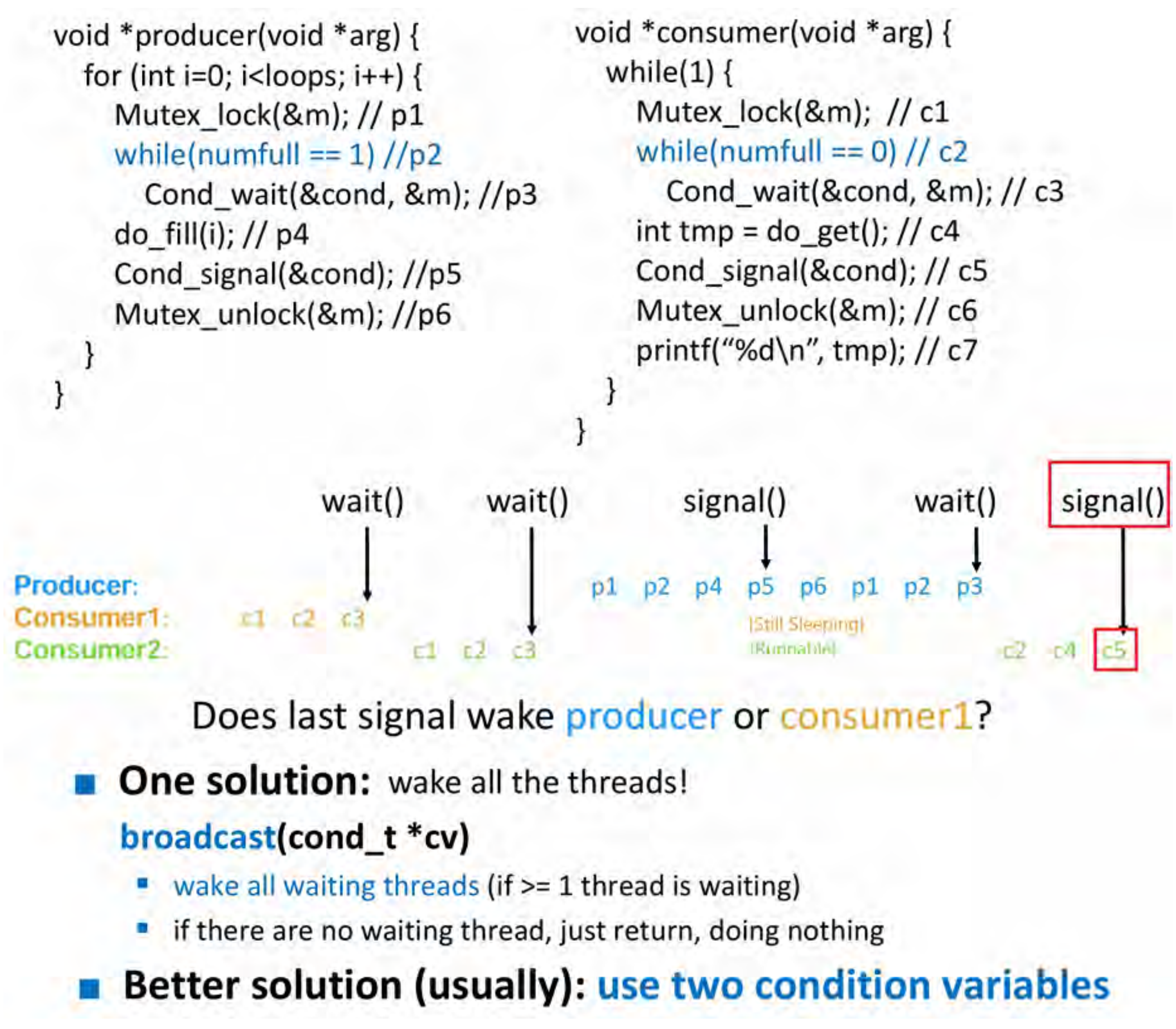
显然，因为消费者 T_{c_1} 已经清空了缓冲区，所以应该唤醒生产者 T_p
但如果它唤醒了 T_{c_2} (这是可能的，取决于操作系统如何管理等待队列)，那么问题就出现了.
消费者 T_{c_2} 会醒过来，重新检查缓存区是否已满 (`c2`)，发现缓冲区为空又继续睡眠 (`c3`)
生产者 T_p 正在睡眠，另一个消费者 T_{c_1} 也在睡眠.
三个线程都在睡眠，完蛋!

因此**信号需要有指向性**:
消费者不应该唤醒消费者，而应该只唤醒生产者，反之亦然.



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}
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...

Figure 30.9: **Thread Trace: Broken Solution (Version 2)**



```
void *producer(void *arg) {
  for (int i=0; i<loops; i++) {
    Mutex_lock(&m); // p1
    while(numfull == 1) //p2
      Cond_wait(&cond, &m); //p3
    do_fill(i); // p4
    Cond_signal(&cond); //p5
    Mutex_unlock(&m); //p6
  }
}
```

```
void *consumer(void *arg) {
  while(1) {
    Mutex_lock(&m);  // c1
    while(numfull == 0) // c2
      Cond_wait(&cond, &m); // c3
    int tmp = do_get(); // c4
    Cond_signal(&cond); // c5
    Mutex_unlock(&m); // c6
    printf("%d\n", tmp); // c7
  }
}
```

Does last signal wake producer or consumer1?

- **One solution:** wake all the threads!

  broadcast(cond_t *cv)
  - wake all waiting threads (if >= 1 thread is waiting)
  - if there are no waiting thread, just return, doing nothing
- **Better solution (usually): use two condition variables**

(3) 正确方案

第二个问题的解决方案也很简单:
使用两个条件变量，而不是一个，以便指向性地发出信号，唤醒特定类别的进程.

我们使用两个条件变量: `empty` 和 `fill`
生产者线程等待条件变量 `empty`，发信号给条件变量 `fill`
相应地，消费者线程等待条件变量 `fill`，发信号给条件变量 `empty`
这样就从设计上避免了第二个问题: 消费者不会唤醒消费者，生产者也不会唤醒生产者.

```
// Condition variables:
// 'empty' signals when there's space in the buffer
// 'fill' signals when there are items to consume
cond_t empty, fill;

// Mutex for synchronizing access to shared resources (e.g., buffer and count)
mutex_t mutex;

// Producer function that produces items and adds them to the buffer
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++)
    {
        Pthread_mutex_lock(&mutex); // p1: Lock the mutex to ensure exclusive
access to shared resources
```

```
        while (count == 1)          // p2: Check if the buffer is full
            // p3: Wait for 'empty' condition signal (i.e., space in buffer),
releasing the mutex
            Pthread_cond_wait(&empty, &mutex);

        put(i);                      // p4: Add an item to the buffer
(producing an item)
        Pthread_cond_signal(&fill);   // p5: Signal the consumer that an item is
available in the buffer
        Pthread_mutex_unlock(&mutex); // p6: Unlock the mutex to allow consumer
access to the buffer
    }
}

// Consumer function that consumes items from the buffer
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex); // c1: Lock the mutex to ensure exclusive
access to shared resources

        while (count == 0)   // c2: Check if the buffer is empty
            // c3: Wait for 'fill' condition signal (i.e., item available),
releasing the mutex
            Pthread_cond_wait(&fill, &mutex);

        int tmp = get();             // c4: Retrieve an item from the buffer
(consuming an item)
        Pthread_cond_signal(&empty); // c5: Signal the producer that the buffer
has space available
        Pthread_mutex_unlock(&mutex);// c6: Unlock the mutex to allow producer
access to the buffer

        printf("%d\n", tmp);  // Print the consumed item
    }
}
```

(4) 最终方案

我们现在有了正确的生产者/消费者方案，但不太通用.
我们最后的修改是增加更多缓冲区槽位，以提高并发效率:

- 单个生产者和消费者时，生产者在睡眠之前可以生产多个值，消费者在睡眠之前可以消费多个值. 这减少了大量的上下文切换，从而提高了效率.
- 多个生产者和消费者时，它支持并发生产和消费，从而提高了效率.

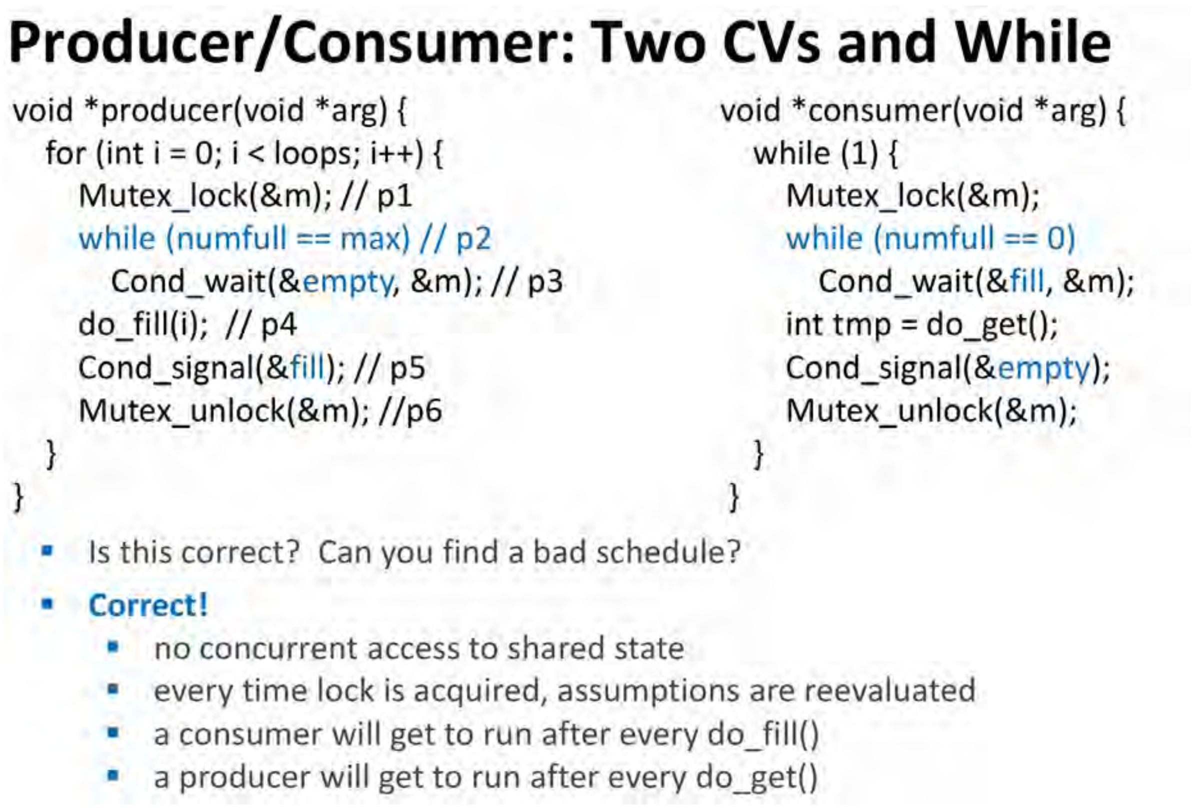
第一处修改是缓冲区结构本身 (推广为环形缓冲区)，以及对应的 `put()` 和 `get()` 方法.

```
int buffer[MAX];  // Circular buffer array of size MAX to hold produced items
int fill = 0;     // Index indicating the next position to fill in the buffer
int use = 0;      // Index indicating the next position to use (consume) from
the buffer
int count = 0;    // Counter to track the number of items currently in the
buffer

// Function to add an item to the buffer
void put(int value) {
    buffer[fill] = value; // Place the provided value at the current 'fill'
index in the buffer
    // Increment 'fill' index circularly, wrapping around to 0 if it reaches MAX
    fill = (fill + 1) % MAX;
    count++;              // Increment the count of items in the buffer to
reflect the addition
}

// Function to remove an item from the buffer
int get() {
    int tmp = buffer[use]; // Retrieve the item at the current 'use' index from
the buffer
    use = (use + 1) % MAX; // Increment 'use' index circularly, wrapping around
to 0 if it reaches MAX
    count--;               // Decrement the count of items in the buffer to
reflect the removal
    return tmp;            // Return the retrieved item to the caller
}
```

第二处修改是生产者和消费者的检查条件.
生产者只有在缓冲区满了的时候才会睡眠(`p2`), 消费者也只有在缓冲区为空的时候睡眠(`c2`)

```
// Condition variables:
// 'empty' signals when there's space in the buffer
// 'fill' signals when there are items to consume
cond_t empty, fill;

// Mutex to ensure mutual exclusion when accessing shared resources (the
buffer).
mutex_t mutex;

// Function for the producer thread
void *producer(void *arg) {
    int i; // Loop counter
    // Loop for a specified number of iterations (controlled by 'loops')
    for (i = 0; i < loops; i++)
    {
        // p1: Lock the mutex to ensure exclusive access to the buffer
        Pthread_mutex_lock(&mutex);

        // p2: Wait while the buffer is full (when 'count' equals 'MAX')
        while (count == MAX)

            // p3: Wait for the condition variable 'empty' to be signaled,
releasing the mutex while waiting
            Pthread_cond_wait(&empty, &mutex);

        // p4: Call the 'put' function to add the item (index 'i') to the buffer
        put(i);

        // p5: Signal the consumer that an item has been added to the buffer
        Pthread_cond_signal(&fill);

        // p6: Unlock the mutex to allow other threads to access the buffer
        Pthread_mutex_unlock(&mutex);
    }
}
```

```
// Function for the consumer thread
void *consumer(void *arg) {
    int i; // Loop counter
    // Loop for a specified number of iterations (controlled by 'loops')
    for (i = 0; i < loops; i++)
    {
        // c1: Lock the mutex to ensure exclusive access to the buffer
        Pthread_mutex_lock(&mutex);

        // c2: Wait while the buffer is empty (when 'count' equals 0)
        while (count == 0)

            // c3: Wait for the condition variable 'fill' to be signaled,
releasing the mutex while waiting
            Pthread_cond_wait(&fill, &mutex);

        // c4: Call the 'get' function to retrieve an item from the buffer
        int tmp = get();

        // c5: Signal the producer that an item has been consumed from the
buffer
        Pthread_cond_signal(&empty);

        // c6: Unlock the mutex to allow other threads to access the buffer
        Pthread_mutex_unlock(&mutex);

        printf("%d\n", tmp); // Print the consumed item
    }
}
```



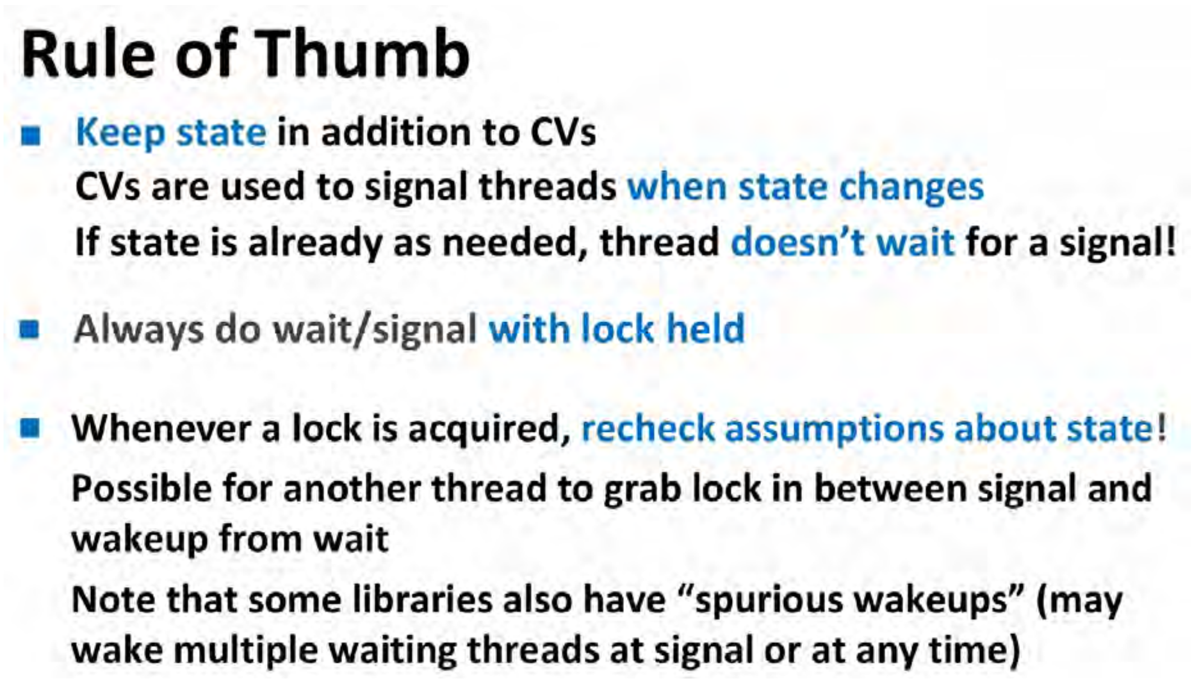
Producer/Consumer: Two CVs and While

```
void *producer(void *arg) {
  for (int i = 0; i < loops; i++) {
    Mutex_lock(&m); // p1
    while (numfull == max) // p2
      Cond_wait(&empty, &m); // p3
    do_fill(i);  // p4
    Cond_signal(&fill); // p5
    Mutex_unlock(&m); //p6
  }
}
```

```
void *consumer(void *arg) {
  while (1) {
    Mutex_lock(&m);
    while (numfull == 0)
      Cond_wait(&fill, &m);
    int tmp = do_get();
    Cond_signal(&empty);
    Mutex_unlock(&m);
  }
}
```

- Is this correct? Can you find a bad schedule?
- Correct!
  - no concurrent access to shared state
  - every time lock is acquired, assumptions are reevaluated
  - a consumer will get to run after every do_fill()
  - a producer will get to run after every do_get()

条件变量的使用准则如下:



Rule of Thumb

- Keep state in addition to CVs
  CVs are used to signal threads when state changes
  If state is already as needed, thread doesn't wait for a signal!
- Always do wait/signal with lock held
- Whenever a lock is acquired, recheck assumptions about state!
  Possible for another thread to grab lock in between signal and wakeup from wait
  Note that some libraries also have "spurious wakeups" (may wake multiple waiting threads at signal or at any time)

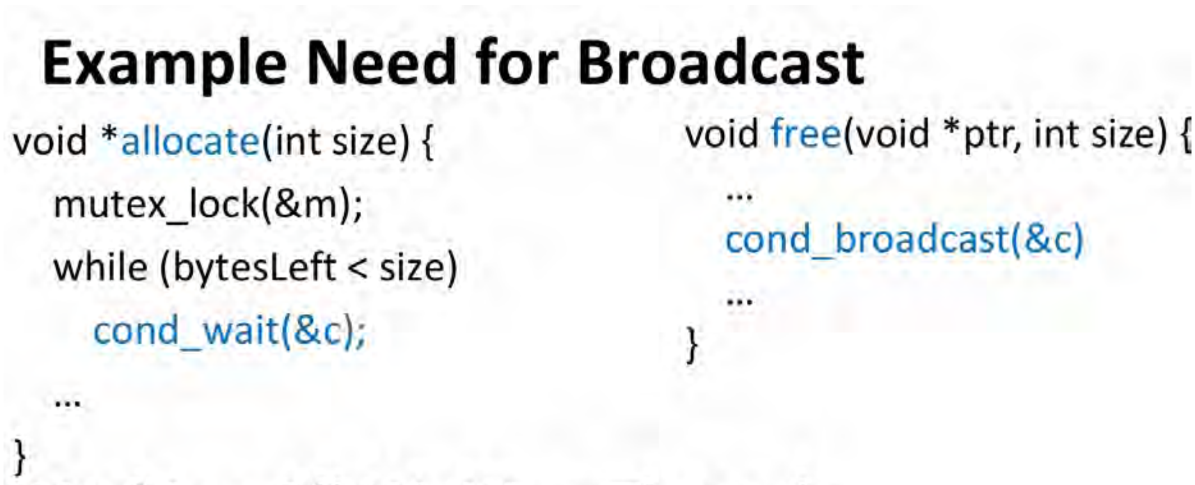
3.5.3 覆盖条件

考虑一个简单的多线程内存分配库:
当线程调用进入内存分配代码时，它可能会因为内存不足而等待.
当线程调用释放内存时，会发信号说有更多内存空闲，唤醒等待队列中调用内存分配代码的线程.

```
// Variable to keep track of the amount of free memory on the heap
// Initialize 'bytesLeft' to the maximum heap size, representing total free
memory available
int bytesLeft = MAX_HEAP_SIZE;

// Condition variable to signal when memory is available
cond_t c;

// Mutex to ensure mutual exclusion when accessing shared resources
mutex_t m;

// Function to allocate memory of a specified size from the heap
void * allocate(int size) {
    // Lock the mutex to ensure exclusive access to the shared memory state
    Pthread_mutex_lock(&m);

    // Wait until there is enough free memory to allocate the requested size
    while (bytesLeft < size)
        // Release the mutex while waiting, and re-acquire it when signaled
        Pthread_cond_wait(&c, &m);

    // Placeholder: Get memory from the heap (this line should implement the
actual allocation logic)
    void *ptr = ...;

    // Decrease the free memory count by the allocated size
    bytesLeft -= size;

    // Unlock the mutex to allow other threads to access the memory
    Pthread_mutex_unlock(&m);
    // Return the pointer to the allocated memory
    return ptr;
}

// Function to free memory previously allocated to 'ptr' of specified size
void free(void *ptr, int size) {
    // Lock the mutex to ensure exclusive access to the shared memory state
    Pthread_mutex_lock(&m);

    // Increase the free memory count by the size of the memory being freed
    bytesLeft += size;

    // Signal waiting threads (if any) that memory is now available
    Pthread_cond_signal(&c);

    // Unlock the mutex to allow other threads to access the memory
    Pthread_mutex_unlock(&m);
}
```

上述代码存在一个问题，这与多个等待线程该唤醒哪一个有关.
假设目前没有空闲内存.
线程 T_a 调用 `allocate(100)`，因空闲内存不足而在条件上睡眠.
接着线程 T_b 调用 `allocate(10)`，也因空闲内存不足而在条件上睡眠.
然后线程 T_c 调用了 `free(50)`，并发信号唤醒等待线程 T_a 和 T_b 中的一个.
不幸的是，如果 T_c 唤醒的是线程 T_a，那么 T_a 仍会因内存不足而睡眠，不会唤醒其他进程.
这导致两个线程 T_a, T_b 都处于睡眠状态.

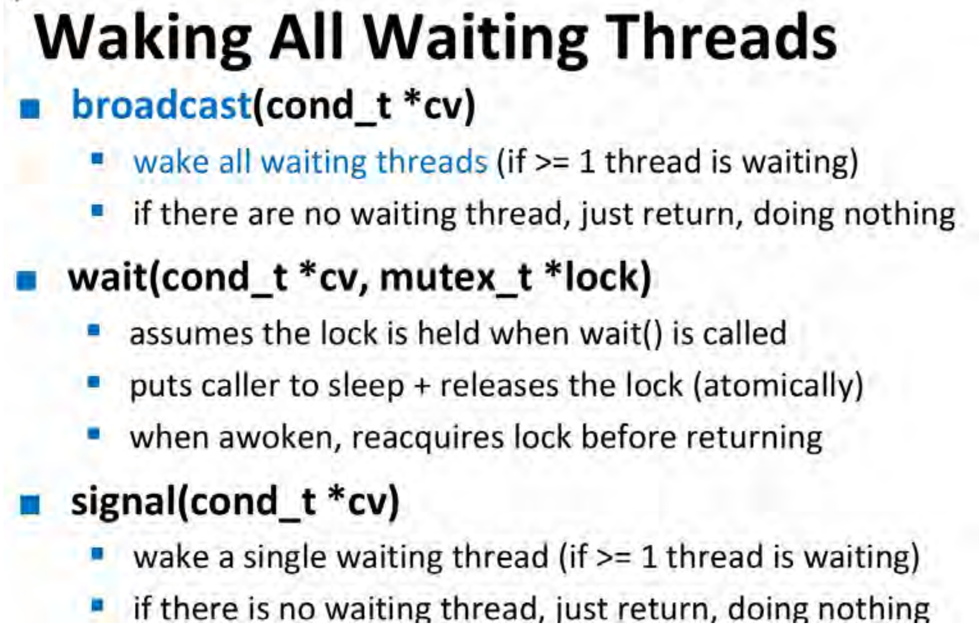
我们的解决方法也很简单:
用 `pthread_cond_broadcast()` 代替上述代码中的 `pthread_cond_signal()`，唤醒所有应该唤醒的线程.
这样当线程 T_c 调用了 `free(50)` 后，它会唤醒所有等待在条件变量 `c` 上的线程 (T_a 和 T_b)
于是 T_a 和 T_b 都会进入就绪状态，只要空闲内存足够，它们就终将完成执行.



Example Need for Broadcast

```
void *allocate(int size) {
    mutex_lock(&m);
    while (bytesLeft < size)
        cond_wait(&c);
    ...
}
```

```
void free(void *ptr, int size) {
    ...
    cond_broadcast(&c)
    ...
}
```



Waking All Waiting Threads

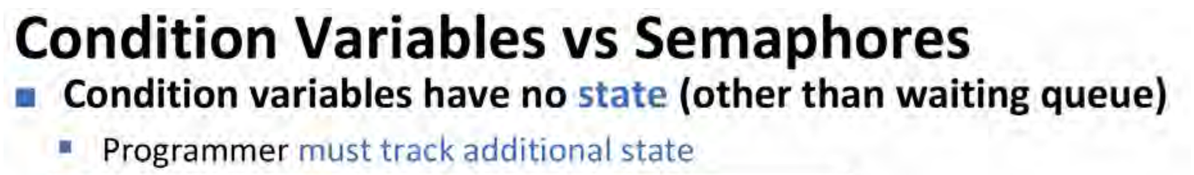
- **broadcast(cond_t *cv)**
  - wake all waiting threads (if >= 1 thread is waiting)
  - if there are no waiting thread, just return, doing nothing
- **wait(cond_t *cv, mutex_t *lock)**
  - assumes the lock is held when wait() is called
  - puts caller to sleep + releases the lock (atomically)
  - when awoken, reacquires lock before returning
- **signal(cond_t *cv)**
  - wake a single waiting thread (if >= 1 thread is waiting)
  - if there is no waiting thread, just return, doing nothing

当然，上述做法也是有代价的，
因为它可能会唤醒了不需要被唤醒 (即其条件并没有真正得到满足) 的等待线程 (例如上例中的 T_a)
这些线程被唤醒后，会重新检查条件并再次睡眠，造成时间浪费.

这种条件变量叫作**覆盖条件** (covering condition)，对应的信号称为**广播信号**，
因为它能覆盖所有需要唤醒线程的场景 (保守策略)
一般来说，如果我们发现某个条件信号只有改成覆盖条件时才能工作，可能因为是程序有缺陷.
但在上述内存分配的例子中，覆盖条件可能是最直接有效的解决方案.

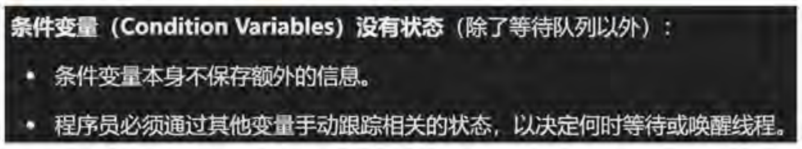
3.6 信号量

Dijkstra 及其同事发明了**信号量** (semaphores)，作为与同步有关的所有工作的唯一原语.
我们可以使用信号量作为锁和条件变量.



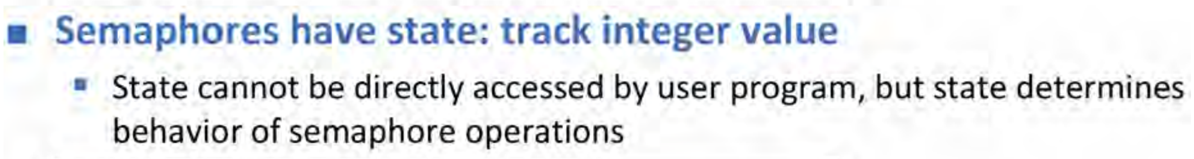
Condition Variables vs Semaphores

- Condition variables have no state (other than waiting queue)
  - Programmer must track additional state

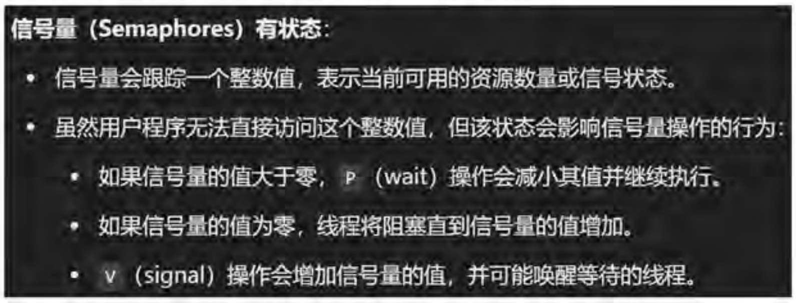


条件变量（Condition Variables）没有状态（除了等待队列以外）：

- 条件变量本身不保存额外的信息。
- 程序员必须通过其他变量手动跟踪相关的状态，以决定何时等待或唤醒线程。



- Semaphores have state: track integer value
  - State cannot be directly accessed by user program, but state determines behavior of semaphore operations

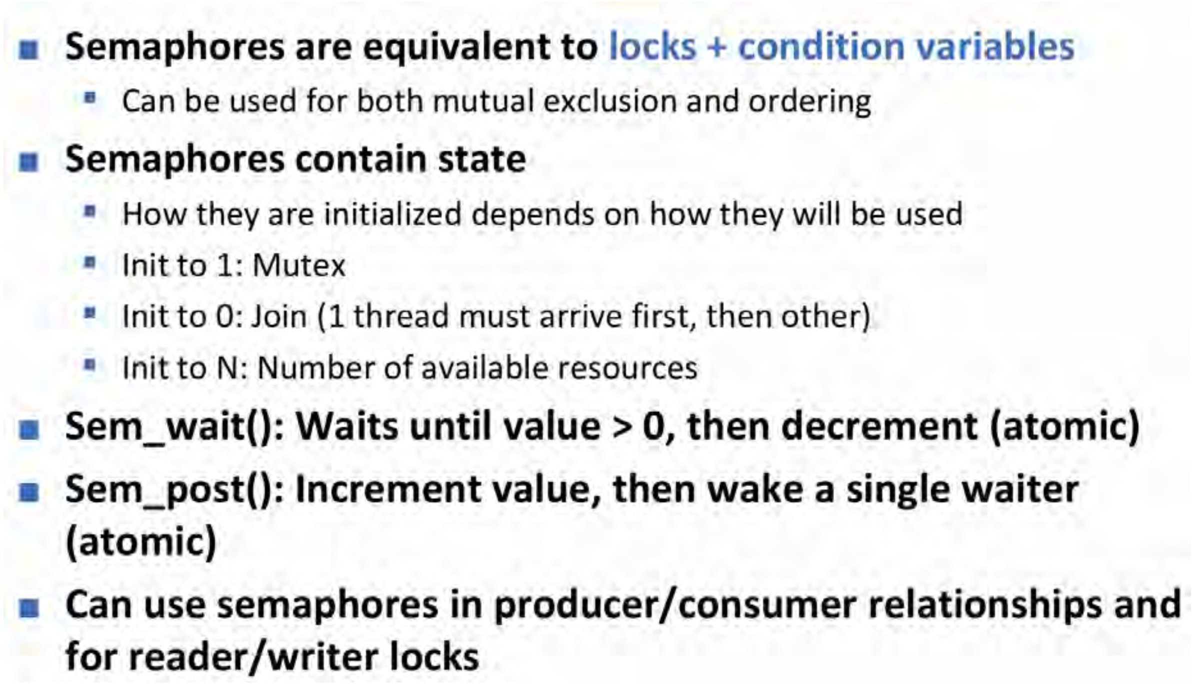


信号量（Semaphores）有状态:

- 信号量会跟踪一个整数值，表示当前可用的资源数量或信号状态。
- 虽然用户程序无法直接访问这个整数值，但该状态会影响信号量操作的行为:
  - 如果信号量的值大于零，`P`（wait）操作会减小其值并继续执行。
  - 如果信号量的值为零，线程将阻塞直到信号量的值增加。
  - `V`（signal）操作会增加信号量的值，并可能唤醒等待的线程。

等价性:

- 信号量和 "锁 + 条件变量" 的组合具有相同的表达能力.
  这意味着任何使用 "锁 + 条件变量" 实现的并发控制逻辑，都可以用信号量实现; 反之亦然.
- 这种等价性强调的是功能上的可替代性，而不是使用上的便利性或实现的简单性.
  在某些情况下，信号量可能更简洁.
  而在另一些情况下，"锁 + 条件变量" 可能更直观
- 信号量可以被用来构造锁和条件变量的机制.
  例如可通过**二值信号量** (binary semaphore) 来实现锁，
  通过**计数信号量** (counting semaphore) 来实现条件等待.



- Semaphores are equivalent to locks + condition variables
  - Can be used for both mutual exclusion and ordering
- Semaphores contain state
  - How they are initialized depends on how they will be used
  - Init to 1: Mutex
  - Init to 0: Join (1 thread must arrive first, then other)
  - Init to N: Number of available resources
- Sem_wait(): Waits until value > 0, then decrement (atomic)
- Sem_post(): Increment value, then wake a single waiter (atomic)
- Can use semaphores in producer/consumer relationships and for reader/writer locks

3.6.1 定义

信号量是有一个整数值的对象，可以用两个函数来操作它 (在 POSIX 标准中，是 `sem_wait()` 和 `sem_post()`)
因为信号量的初始值能够决定其行为，所以首先要初始化信号量才能调用其他函数与之交互 (在 POSIX 标准中，是 `sem_init()`)
在 POSIX 标准中，我们使用 `sem_init()` 初始化信号量，其函数原型为:

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

- `sem_t *sem`: 指向信号量对象的指针
- `int pshared`: 整数值，指示信号量是否可以在不同进程之间共享.
  如果为 `0`，则表示信号量仅在线程间共享; 如果为非零值 (例如 `1`)，则表示信号量可以在进程间共享.
- `unsigned int value`: 初始信号量的值.

信号量初始化之后，我们可以调用 `sem_wait()` 或 `sem_post()` 与之交互:
(它们都是原子操作)

- `sem_wait()` 用于请求获取信号量，其函数原型为:

  ```
  int sem_wait(sem_t *sem);
  ```

  其中 `sem_t *sem` 是指向信号量对象的指针.
  若信号量的值大于 `0`，函数将信号量的值减 `1` 并返回，表示成功获取信号量.
  若信号量的值等于 `0`，调用此函数的线程将被阻塞，直到信号量的值大于 `0` (即有其他线程调用 `sem_post()` 来释放信号量)
  (在某些实现中，当信号量的值为负时，表示有线程正在等待获取信号量，此时绝对值代表正在等待的线程数量)
- `sem_post()` 用于释放信号量，其函数原型为:

  ```
  int sem_post(sem_t *sem);
  ```

  其中 `sem_t *sem` 是指向信号量对象的指针.
  `sem_post()` 将信号量的值加 `1`，表示该线程已完成对共享资源的访问，并允许其他线程获取该资源.
  如果有线程正在等待此信号量 (由于先前调用了 `sem_wait()` 并被阻塞)，那么 `sem_post()` 将会唤醒其中一个等待的线程.

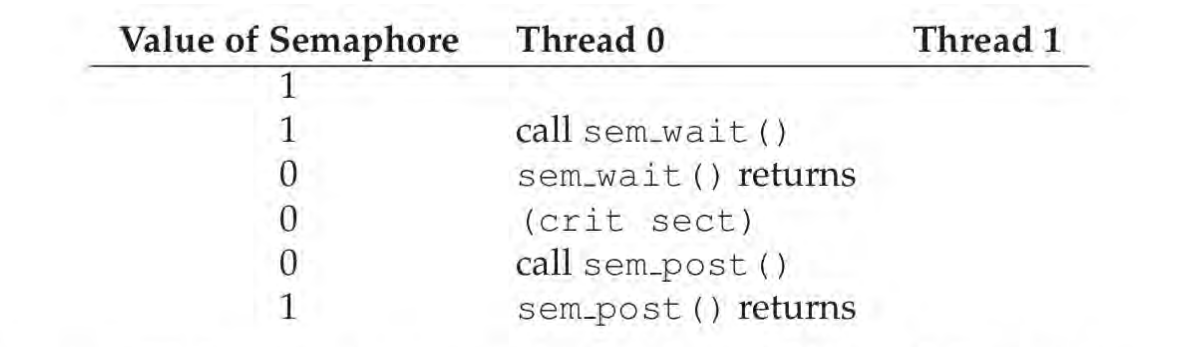
3.6.2 信号量实现锁

我们可以使用**二值信号量** (binary semaphore) 来实现锁.

```
// Declare a semaphore variable 'sem' to represent the lock
typedef struct __lock_t {
    sem_t sem;  // 'sem' is the semaphore used for locking
} lock_t;

// sem_init(&lock->sem, 0, X) initialize the semaphore 'sem' with an initial
value 'X'.
```

```
// The second argument (0) indicates that the semaphore is shared between threads of the same process.
// 'X' should be set to the number of available resources that the semaphore will control.
// Typically, 'X' can be:
// - 1: for mutual exclusion (binary semaphore)
// - N (a positive integer): for a counting semaphore that controls access to N resources.
void init(lock_t *lock) {
    sem_init(&lock->sem, 0, 1);
}

// Acquire the lock by waiting (decrementing) the semaphore.
// If the semaphore's value is greater than 0, decrement it and proceed.
// If the semaphore's value is 0, block the thread until the semaphore is signaled (incremented).
void acquire(lock_t *lock) {
    sem_wait(&lock->sem);  // Decrease the semaphore's value, blocking if it's 0
}

// critical section here
// Only one thread should be in critical section at a time if X was initialized to 1.

// Release the lock by signaling (incrementing) the semaphore.
// This increases the semaphore's value.
// If there were threads waiting for the semaphore, one of them will be unblocked.
void release(lock_t *lock) {
    sem_post(&lock->sem);  // Increase the semaphore's value, potentially waking a blocked thread
}
```

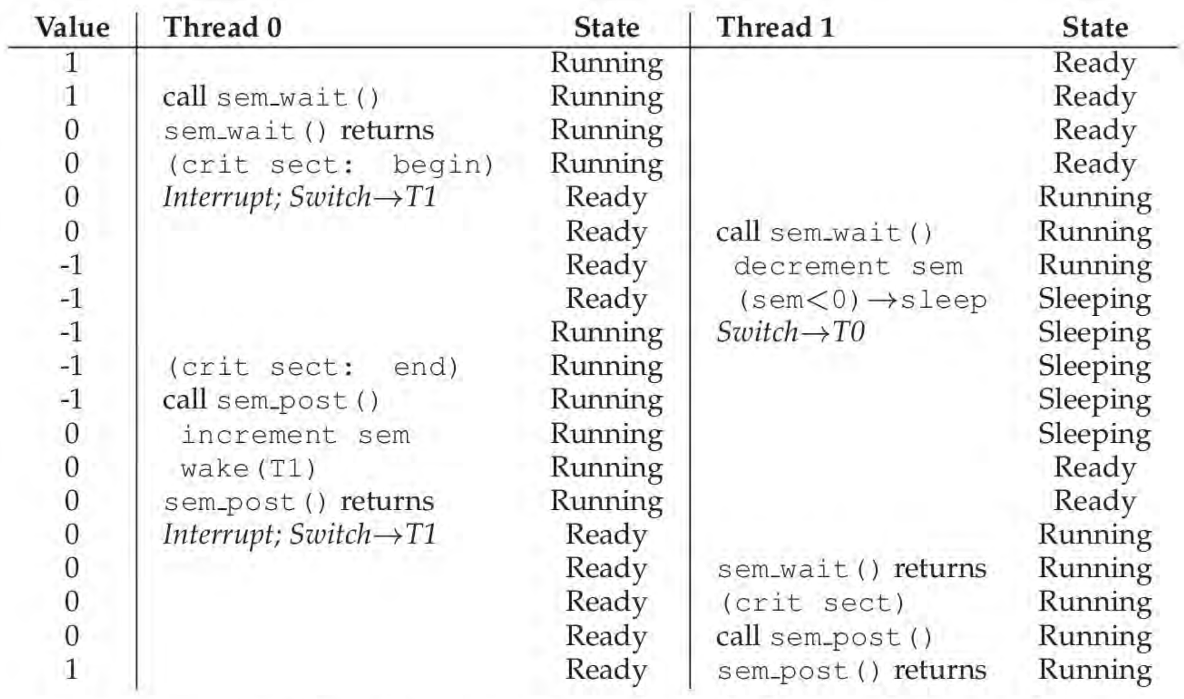
假设有两个线程的场景.
假设线程 0 调用了 `sem_wait()`，把信号量的值减为 `0`
现在线程 0 可以自由进入临界区.
如果没有其他线程尝试获取锁，那么当它调用 `sem_post()` 时就会将信号量重置为 `1`
(因为没有等待线程，不会唤醒其他线程)



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

Figure 31.4: **Thread Trace: Single Thread Using A Semaphore**

如果线程 0 持有锁 (即调用了 `sem_wait()` 之后，调用 `sem_post()` 之前)，
线程 1 调用 `sem_wait()` 尝试进入临界区，那么更有趣的情况就发生了.
这种情况下，线程 1 把信号量减为 `-1`，然后等待.
线程 0 再次运行，最终调用 `sem_post()`，将信号量的值增加到 `0`，唤醒等待的线程 1.
此时线程 1 就可以获取锁，执行结束时再次调用 `sem_post()` 增加信号量的值，将它恢复为 `1`



Value	Thread 0	State	Thread 1	State
1		Running		Ready
1	call sem_wait()	Running		Ready
0	sem_wait() returns	Running		Ready
0	(crit sect: 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() returns	Running
0		Ready	(crit sect)	Running
0		Ready	call sem_post()	Running
1		Ready	sem_post() returns	Running

Figure 31.5: **Thread Trace: Two Threads Using A Semaphore**

使用信号量来实现条件变量也是可行的，但很难实现正确.
(本课程不涉及，详情可参考 Microsoft Research)

3.6.3 锁和条件变量实现信号量

```
// Semaphore structure
typedef struct sem_t {
    int value;              // Semaphore value
    pthread_mutex_t lock;   // Mutex to protect access to the semaphore's value
    pthread_cond_t cond;    // Condition variable to block threads
} sem_t;

// sem_init function - Initializes the semaphore with an initial value
int sem_init(sem_t *s, int pshared, unsigned int value) {
    // Initialize the mutex and condition variable for synchronization
    int ret;

    ret = pthread_mutex_init(&s->lock, NULL); // Initialize the mutex
    if (ret != 0) {
        return ret;  // Return an error code if mutex initialization fails
    }

    ret = pthread_cond_init(&s->cond, NULL); // Initialize the condition
variable
    if (ret != 0) {
        pthread_mutex_destroy(&s->lock);  // Cleanup the mutex if condition
variable initialization fails
        return ret;
    }

    s->value = value;  // Set the initial value of the semaphore
    return 0;  // Return 0 on success
}
```

```
// sem_wait function - Decreases the semaphore value and waits if the value is
zero or less
void sem_wait(sem_t *s) {
    // Acquire the lock to ensure mutual exclusion while modifying the
semaphore's state
    lock_acquire(&s->lock);

    // If the semaphore value is less than or equal to 0, block the thread until
signaled
    while (s->value <= 0) {
        // Wait on the condition variable. The thread will be blocked here
        // until the condition is signaled (i.e., the value of the semaphore
becomes > 0).
        cond_wait(&s->cond);
    }

    // Decrease the semaphore value to reflect that a resource has been acquired
    s->value--;

    // Release the lock after modifying the semaphore value
    lock_release(&s->lock);
}

// sem_post function - Increments the semaphore value and signals a waiting
thread
void sem_post(sem_t *s) {
    // Acquire the lock to ensure mutual exclusion while modifying the
semaphore's state
    lock_acquire(&s->lock);

    // Increase the semaphore value to indicate that a resource is now available
    s->value++;

    // Signal the condition variable to wake up a waiting thread, if any
    cond_signal(&s->cond);

    // Release the lock after modifying the semaphore value and signaling the
condition variable
    lock_release(&s->lock);
}
```

3.6.4 信号量实现条件等待

我们可以使用**计数信号量** (counting semaphore) 来实现条件等待.
考虑一个简单的例子，假设父线程创建子线程，并且等待它结束.

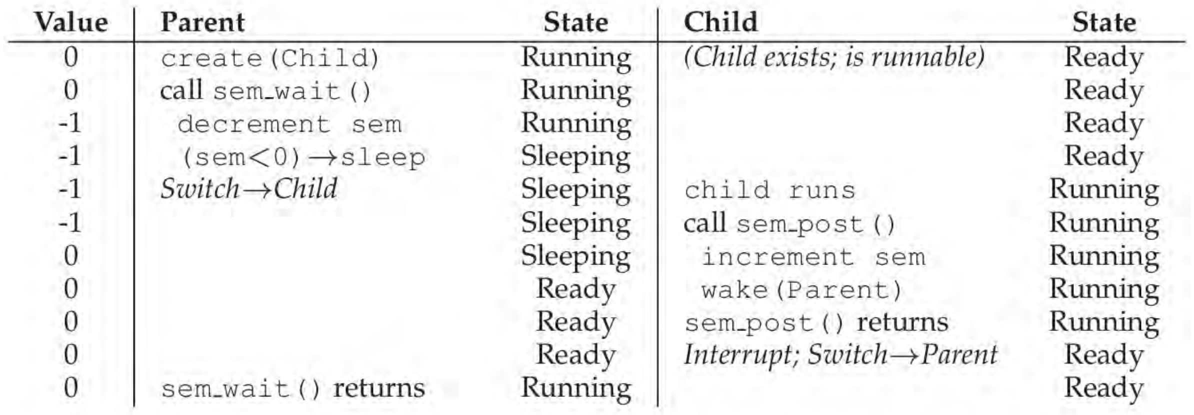
```
// Declare a semaphore variable 's' to manage synchronization between parent and
child threads
sem_t s;

void *child(void *arg) {
    // Print message to indicate the child thread has started its execution
    printf("child\n");
    // Signal by incrementing the semaphore 's', indicating that the child thread
is done
    sem_post(&s);
    return NULL;
}

int main(int argc, char *argv[]) {
    // sem_init(&s, 0, X) initialize the semaphore 's' with an initial value 'X'
    // The second argument (0) indicates that the semaphore is shared between
threads of the same process.
    // 'X' should be 0 so that the parent waits until the child signals it's
done.
    // This makes `sem_wait(&s)` in the parent block until `sem_post(&s)` is
called by the child.
    sem_init(&s, 0, 0);

    // Indicate that the parent has begun execution
    printf("parent: begin\n");

    // Declare a thread identifier for the child thread
    pthread_t c;

    // Create the child thread and start executing the `child` function
    pthread_create(&c, NULL, child, NULL);

    // Parent waits here until the semaphore 's' is incremented by the child
thread's `sem_post(&s)`
    sem_wait(&s);

    // Print message indicating the parent has finished waiting and is about to
exit
    printf("parent: end\n");

    return 0;
}
```

预期的运行结果为:

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

父线程调用 `sem_wait()`，子线程调用 `sem_post()`，父线程等待子线程执行完成.
信号量初始值应该是 0，有两种情况需要考虑:

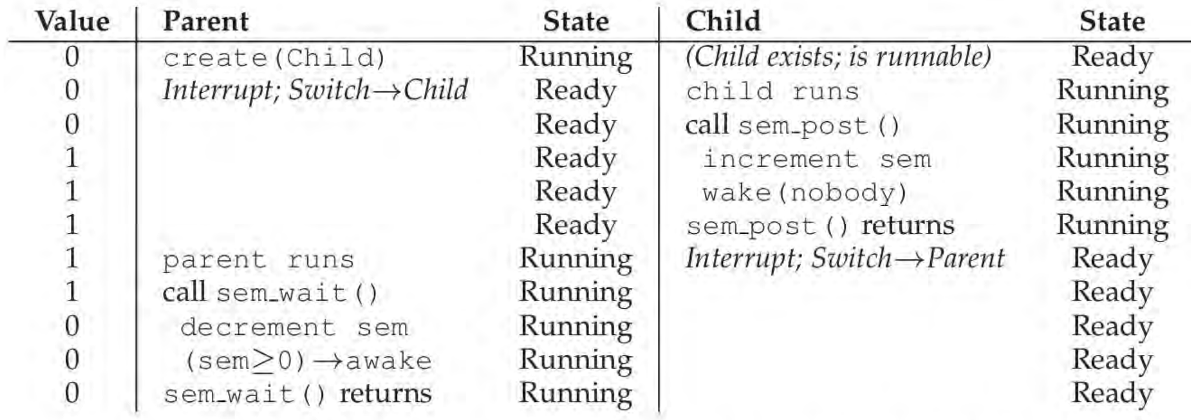
第一种情况是父线程创建子线程后仍然继续运行父线程.
此时父线程调用 `sem_wait()` 会先于子线程调用 `sem_post()`
我们希望父线程等待子线程运行，唯一的办法是让信号量的值不大于 `0`
父线程运行，将信号量减为 `-1`，然后睡眠等待.
接着子线程运行，调用 `sem_post()`，信号量增加为 `0`，唤醒父线程.
然后父线程从 `sem_wait()` 返回，完成该程序.



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()` returns	Running		Ready

Figure 31.7: **Thread Trace: Parent Waiting For Child (Case 1)**

第二种情况是父线程创建子线程后直接运行子线程.
此时子线程会先调用 `sem_post()`，将信号量从 `0` 增加到 `1`
然后父线程运行时，会调用 `sem_wait()`，
将信号量从 `1` 减为 `0`，没有等待，直接从 `sem_wait()` 返回，也达到了预期效果.



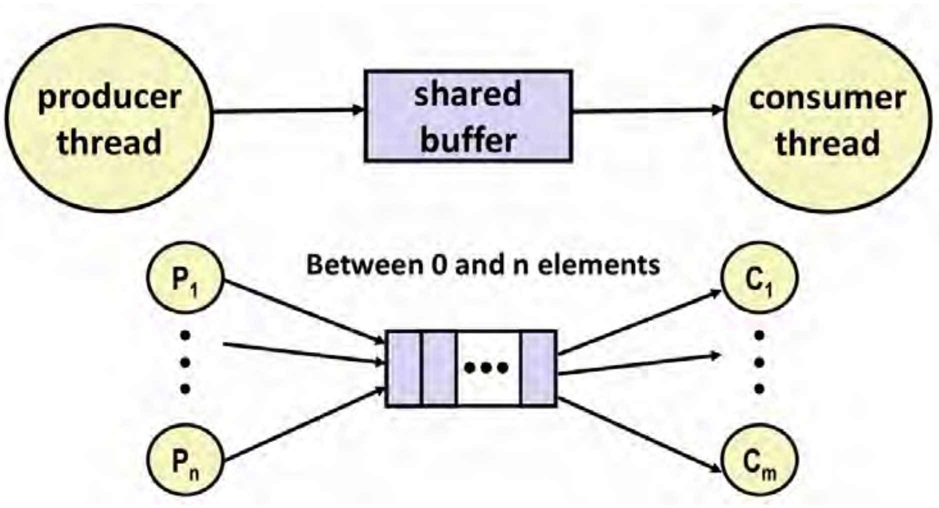
Value	Parent	State	Child	State
0	`create(Child)`	Running	*(Child exists; is runnable)*	Ready
0	*Interrupt; Switch→Child*	Ready	`child runs`	Running
0		Ready	call `sem_post()`	Running
1		Ready	`increment sem`	Running
1		Ready	`wake(nobody)`	Running
1		Ready	`sem_post()` returns	Running
1	`parent runs`	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()` returns	Running		Ready

Figure 31.8: **Thread Trace: Parent Waiting For Child (Case 2)**

使用信号量协调对共享资源的访问:

- **基本思想**: 线程使用信号量操作来通知另一个线程某个条件已经成立.
- **二值信号量 (Binary Semaphores)**
  用于通知其他线程.
  二值信号量的值通常为 0 或 1，类似于锁的机制，可以用来同步线程的执行，通知某个条件发生了变化.
- **计数信号量 (Counting Semaphores)**
  用于跟踪资源的状态.
  计数信号量的值表示可用资源的数量，线程可以通过减少信号量的值来获取资源.
  若信号量值为零，则表示资源不足，线程需要等待资源可用.

3.6.5 生产者/消费者问题



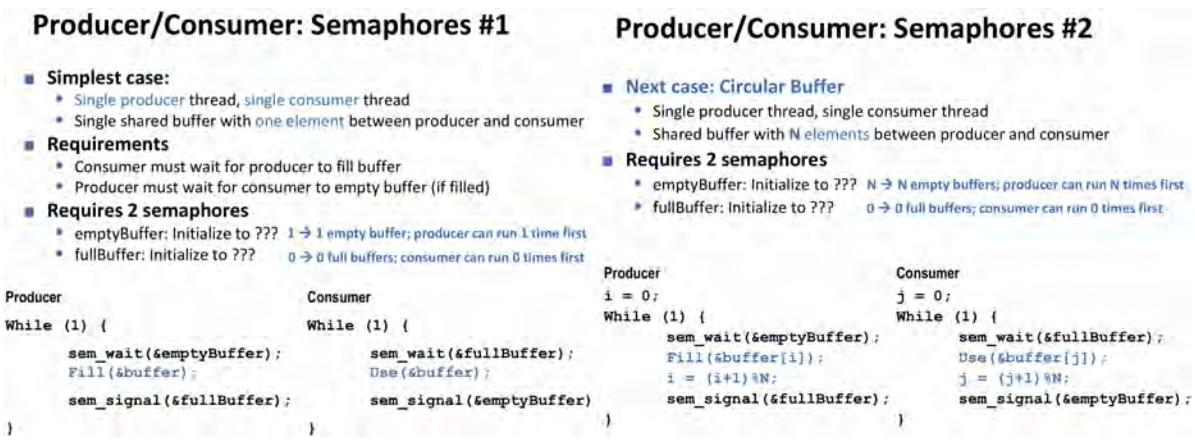
生产者等待缓冲区变为有空闲的状态，然后加入数据，通知消费者.
消费者等待缓冲区变为有数据的状态，然后取走数据，通知生产者.

生产者线程等待条件变量 `empty`，发信号给条件变量 `full`
相应地，消费者线程等待条件变量 `full`，发信号给条件变量 `empty`
此外，向缓冲区增减元素是临界区，需要保护起来.
所以我们使用二值信号量 `mutex` 来实现锁.
注意 `mutex` 不能放在 `empty` 和 `full` 的外部，否则会引发死锁.
例如当先运行消费者线程时，它会持有 `mutex` 并因 `full` 等于 0 而阻塞，
而随后运行的生产者线程会因 `mutex` 被 `consumer` 持有而阻塞，造成死锁.

```
int buffer[MAX];

// Declare semaphores to synchronize the producer and consumer.
sem_t empty;  // Semaphore to count available empty slots in the buffer.
sem_t full;   // Semaphore to count the number of full slots in the buffer.
sem_t mutex;  // Mutex semaphore to ensure mutual exclusion when accessing the
shared buffer.

void *producer(void *arg) {
    // Loop to continuously produce items.
    while(1) {
        // Wait (decrement) the 'empty' semaphore before putting an item in the
buffer.
        // If 'empty' is 0, the buffer is full, and the producer will wait until
space becomes available.
        sem_wait(&empty);

        // Acquire the mutex lock to ensure mutual exclusion when accessing the
buffer.
        sem_wait(&mutex);

        // Find an empty slot in the buffer using 'findempty'.
        myi = findempty(&buffer);

        // Add an item to the buffer (this is done by the 'fill' function).
        fill(&buffer[myi]);  // The producer places an item in the buffer at
position 'myi'.

        // Release the mutex lock after updating the buffer.
        sem_post(&mutex);
```

```c
        // Post (increment) the 'full' semaphore to signal that there is one more item in the buffer.
        // This signals the consumer that there is a new item to consume.
        sem_post(&full);
    }
}

void *consumer(void *arg) {
    while(1) {
        // Wait (decrement) the 'full' semaphore before consuming an item.
        // If 'full' is 0, the buffer is empty
        // and the consumer will block until there is something to consume.
        sem_wait(&full);

        // Acquire the mutex lock to ensure mutual exclusion when accessing the buffer.
        sem_wait(&mutex);

        // Find a filled slot in the buffer using 'findfull'.
        myj = findfull(&buffer);

        // Get an item from the buffer (this is done by the 'use' function).
        use(&buffer[myj]);  // The consumer consumes the item from the buffer at position 'myj'.

        // Release the mutex lock after accessing the buffer.
        sem_post(&mutex);

        // Post (increment) the 'empty' semaphore, signaling that there is space available in the buffer.
        sem_post(&empty);
    }
}

int main(int argc, char *argv[]) {
    // Initialization of semaphores.
    // 'empty' represents the count of empty slots in the buffer.
    // The buffer is full at the start, so it's initialized to MAX.
    sem_init(&empty, 0, MAX);

    // 'full' represents the count of filled slots in the buffer.
    // Initially, the buffer is empty, so it's initialized to 0.
    sem_init(&full, 0, 0);

    // 'mutex' is initialized to 1, allowing only one thread (either producer or consumer)
    // to access the shared buffer at a time.
    sem_init(&mutex, 0, 1);

    // Additional setup for threads, etc.
}
```

值得注意的是，若缓冲区大小为 1，则无需加锁 `mutex`，即使有多个生产者和消费者也是正确的:



Producer/Consumer: Semaphores #1

- **Simplest case:**
  - Single producer thread, single consumer thread
  - Single shared buffer with one element between producer and consumer
- **Requirements**
  - Consumer must wait for producer to fill buffer
  - Producer must wait for consumer to empty buffer (if filled)
- **Requires 2 semaphores**
  - emptyBuffer: Initialize to ??? 1 → 1 empty buffer; producer can run 1 time first
  - fullBuffer: Initialize to ??? 0 → 0 full buffers; consumer can run 0 times first

Producer

```
While (1) {

        sem_wait(&emptyBuffer);
        Fill(&buffer);

        sem_signal(&fullBuffer);

}
```

Consumer

```
While (1) {

        sem_wait(&fullBuffer);
        Use(&buffer);

        sem_signal(&emptyBuffer)

}
```

Producer/Consumer: Semaphores #2

- Next case: Circular Buffer
  - Single producer thread, single consumer thread
  - Shared buffer with N elements between producer and consumer
- **Requires 2 semaphores**
  - emptyBuffer: Initialize to ??? N → N empty buffers; producer can run N times first
  - fullBuffer: Initialize to ??? 0 → 0 full buffers; consumer can run 0 times first

Producer

```
i = 0;
While (1) {
        sem_wait(&emptyBuffer);
        Fill(&buffer[i]);
        i = (i+1)%N;
        sem_signal(&fullBuffer);
}
```

Consumer

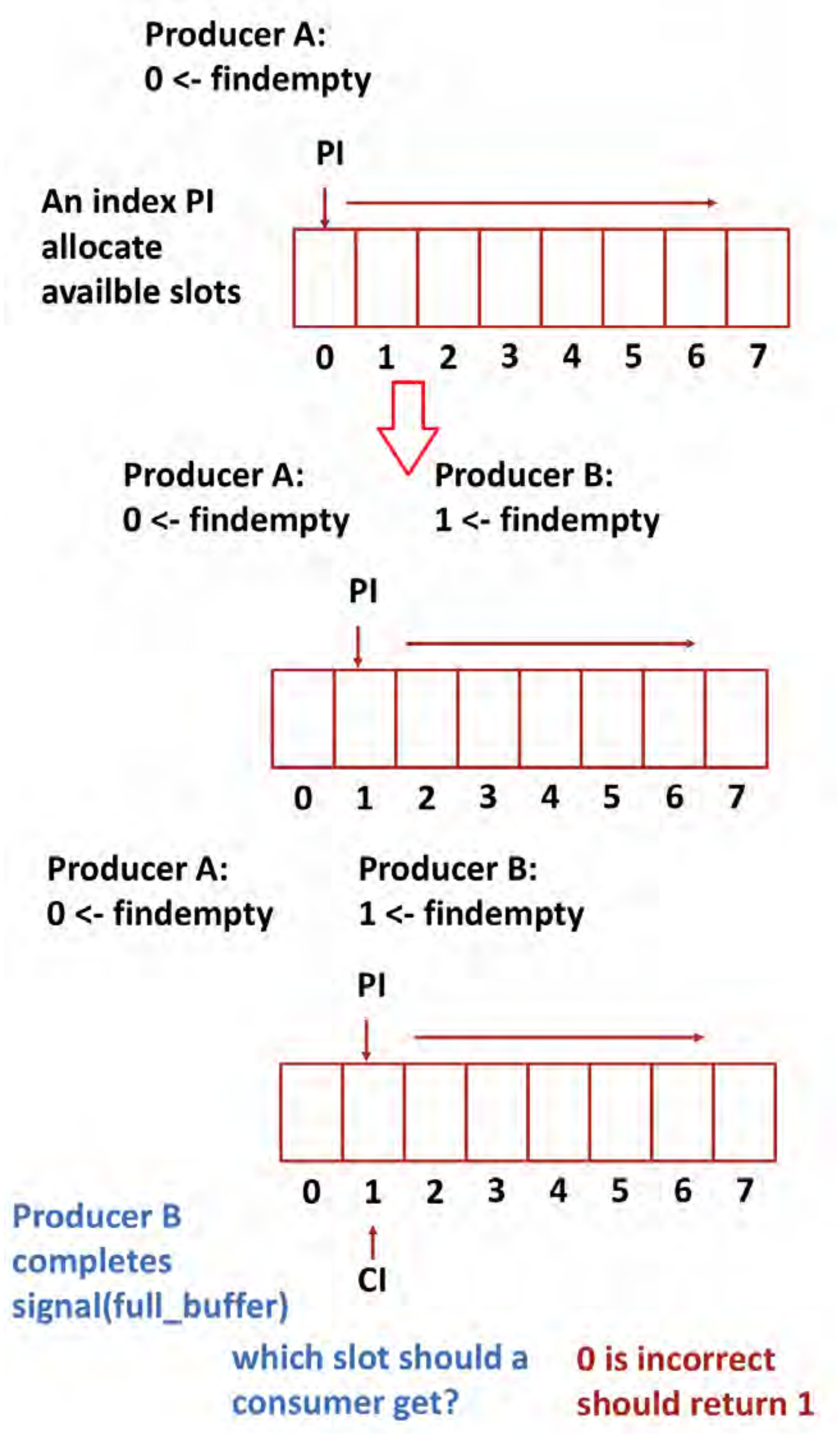
```
j = 0;
While (1) {
        sem_wait(&fullBuffer);
        Use(&buffer[j]);
        j = (j+1)%N;
        sem_signal(&emptyBuffer);
}
```

但是上述实现的效率太低了，同一时间只能有一个线程 (无论是生产者还是消费者) 读/写缓冲区.
我们可以只把 `findempty()` 和 `findfull()` 做为临界区保护起来
(因为我们只需保证不同生产者和不同消费者使用缓冲区的不同位置即可)

```
void *producer(void *arg) {
    while(1) {
        sem_wait(&empty);
        sem_wait(&mutex);
        myi = findempty(&buffer);
        sem_post(&mutex);
        fill(&buffer[myi]);
        sem_post(&full);
    }
}

void *consumer(void *arg) {
    while(1) {
        sem_wait(&full);
        sem_wait(&mutex);
        myj = findfull(&buffer);
        sem_post(&mutex);
        use(&buffer[myj]);
        sem_post(&empty);
    }
}
```

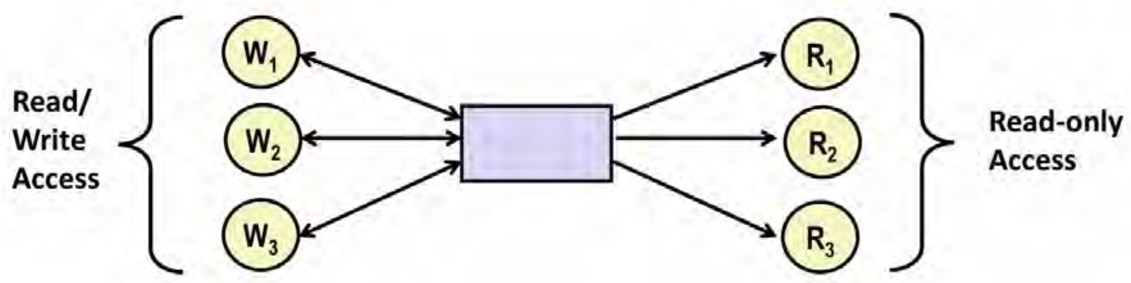
实现 `fullempty()` 和 `findfull()` 的策略:

- ① 扫描缓冲区来找到可用项.
  `findempty` 扫描缓冲区，找到第一个空的槽;
  `findfull` 扫描缓冲区，找到第一个满的槽.
- ② 将环形缓冲区更改成共享链表.
  生产者将新的项添加到链表的末尾，而消费者从链表的开头获取已有的项.

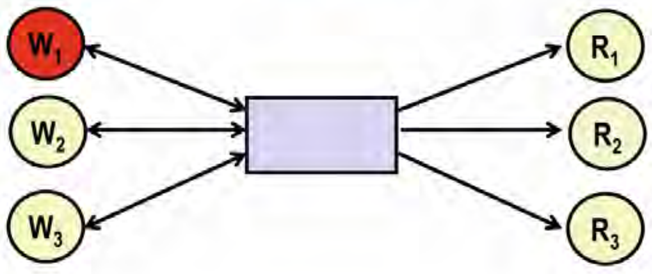


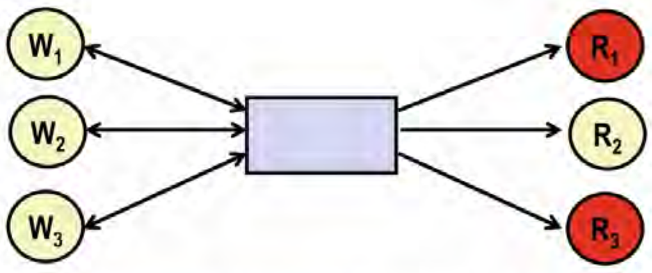
3.6.6 读者/写者问题

访问不同的数据结构，可能需要不同类型的锁.
例如一个并发链表有很多插入和查找操作.
插入操作会修改链表的状态 (因此传统的临界区有用)，而查找操作只是读取该结构.
只要没有进行插入操作，我们就可以并发的执行多个查找操作.
读者/写者锁 (reader-writer lock) 就是用来完成这种操作的.



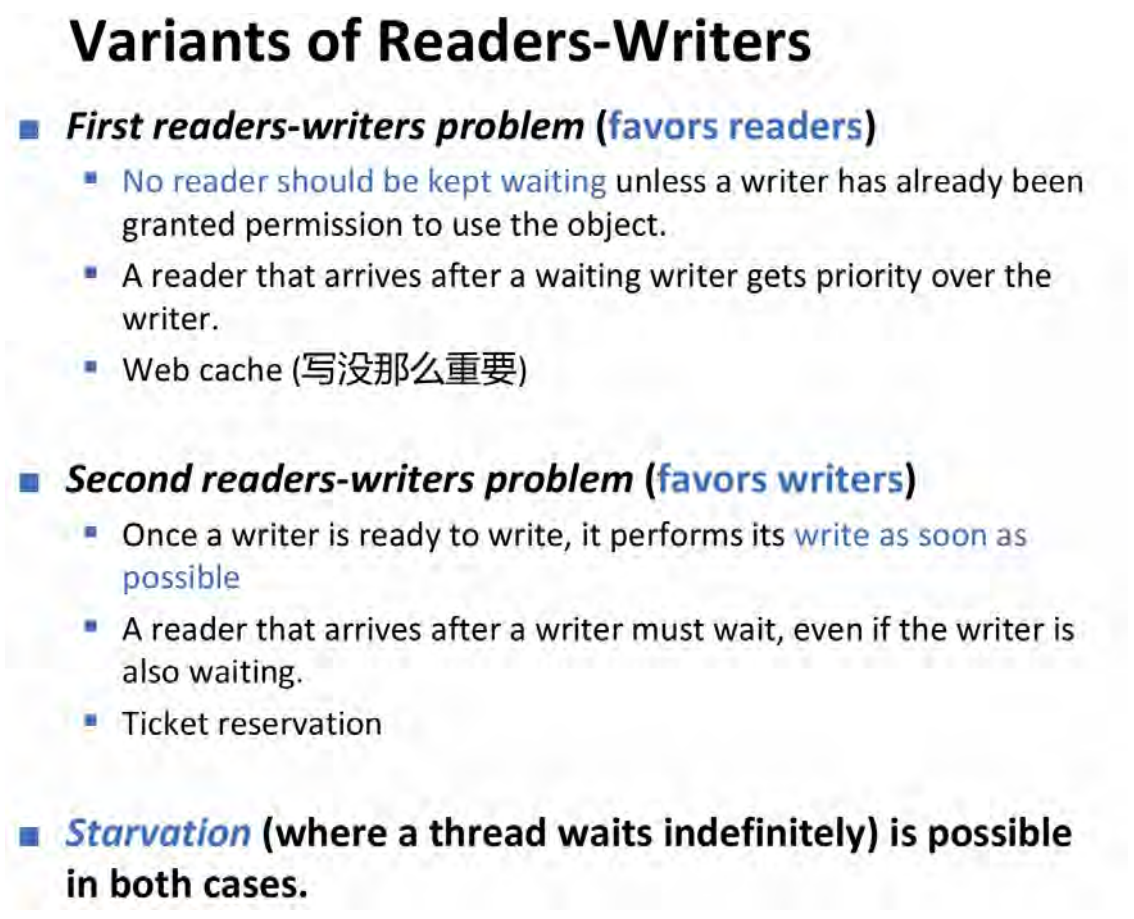
- **Problem statement:**
  - *Reader* threads only read the object
  - *Writer* threads modify the object (read/write access)
  - Writers must have exclusive access to the object
  - Unlimited number of readers can access the object





- **Goal: Let multiple reader threads grab lock (shared)**
- **Only one writer thread can grab lock (exclusive)**
  - No reader threads
  - No other writer threads
- **How to implement reader/writer lock with semaphores**
  - `rwlock_acquire_readlock`
  - `rwlock_release_readlock`
  - `rwlock_acquire_writelock`
  - `rwlock_release_writelock`

读者-写者问题的两种变体:



Variants of Readers-Writers

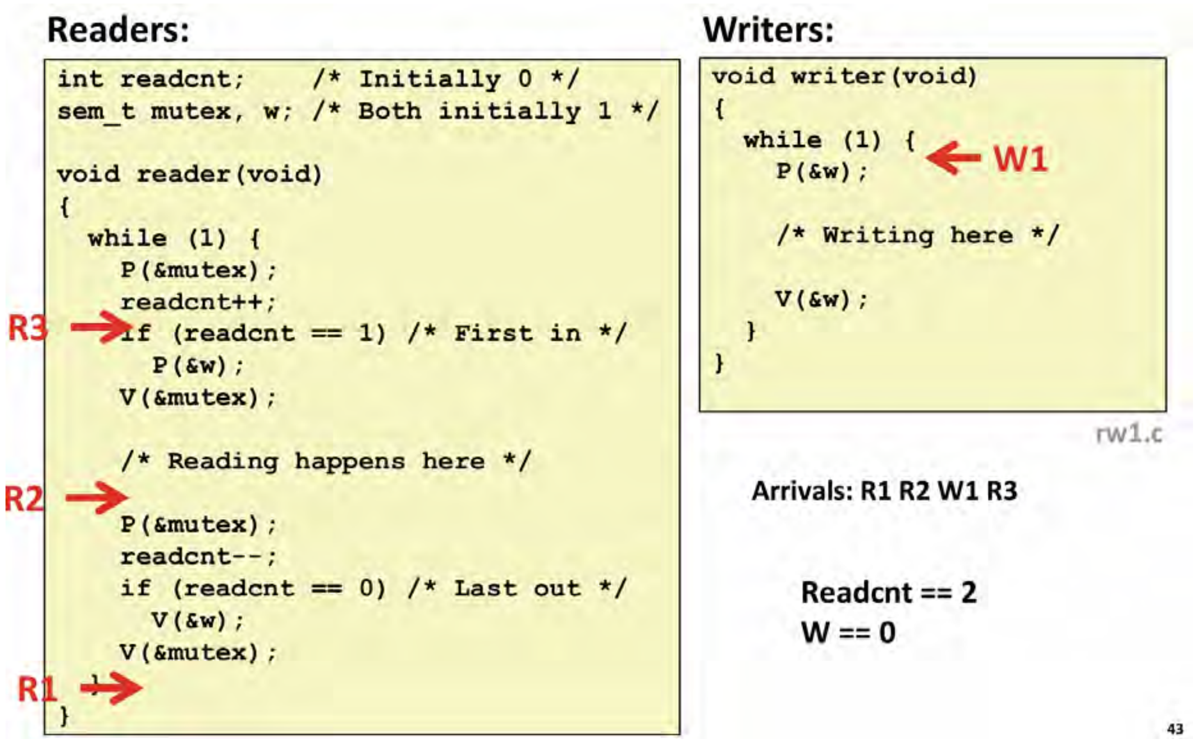
- ***First readers-writers problem*** **(favors readers)**
  - No reader should be kept waiting unless a writer has already been granted permission to use the object.
  - A reader that arrives after a waiting writer gets priority over the writer.
  - Web cache (写没那么重要)
- ***Second readers-writers problem*** **(favors writers)**
  - Once a writer is ready to write, it performs its write as soon as possible
  - A reader that arrives after a writer must wait, even if the writer is also waiting.
  - Ticket reservation
- ***Starvation*** **(where a thread waits indefinitely) is possible in both cases.**

第一读者-写者问题的一个简单的实现如下:

```c
// Define the structure for a reader-writer lock
typedef struct_rwlock_t {
    sem_t lock;       // Binary semaphore to protect access to the `readers` counter
    sem_t writelock;  // Semaphore to ensure mutual exclusion for writers or grant shared access for readers
    int readers;      // Counter for the number of readers currently accessing the critical section
} rwlock_t;

// Initialize the reader-writer lock
void rwlock_init(rwlock_t *rw) {
    rw->readers = 0;                    // Start with zero readers
    sem_init(&rw->lock, 0, 1);          // Initialize the `lock` semaphore as a binary semaphore
    sem_init(&rw->writelock, 0, 1);     // Initialize the `writelock` semaphore as a binary semaphore
}

// Acquire a read lock
void rwlock_acquire_readlock(rwlock_t *rw) {
    sem_wait(&rw->lock);                // Lock access to the `readers` counter
    rw->readers++;                      // Increment the number of readers
    if (rw->readers == 1) {             // If this is the first reader
        sem_wait(&rw->writelock);       // Block writers by acquiring the `writelock`
    }
    sem_post(&rw->lock);                // Unlock access to the `readers` counter
}
```

```
// Release a read lock
void rwlock_release_readlock(rwlock_t *rw) {
    sem_wait(&rw->lock);                 // Lock access to the `readers` counter
    rw->readers--;                       // Decrement the number of readers
    if (rw->readers == 0) {              // If this was the last reader
        sem_post(&rw->writelock);        // Allow writers by releasing the
`writelock`
    }
    sem_post(&rw->lock);                 // Unlock access to the `readers`
counter
}

// Acquire a write lock
void rwlock_acquire_writelock(rwlock_t *rw) {
    sem_wait(&rw->writelock);    // Acquire the `writelock` to ensure exclusive
access for writing
}

// Release a write lock
void rwlock_release_writelock(rwlock_t *rw) {
    sem_post(&rw->writelock);    // Release the `writelock` to allow others
(readers or writers) access
}
```

- 写者需要调用 `rwlock_acquire_lock()` 获得写锁，更新完成后调用 `rwlock_release_writelock()` 释放写锁.
  内部通过一个 `writelock` 的信号量保证只有一个写者能获得锁进入临界区，从而更新数据结构.
- 读者需要调用 `rwlock_acquire_readlock()` 获取读锁 `lock`，
  并增加 `reader` 变量，以追踪目前有多少个读者在访问该数据结构.
  当第一个读者获取读锁 `lock` 时，读者也会获取写锁 `writelock`
  因此只要一个读者获得了读锁，其他读者也可以获取这个读锁.
  此时写者要想获取写锁，就必须等到所有读者都结束.
  最后一个退出的读者会释放写锁，从而让等待的写者能够获取写锁.



Readers:

```
int readcnt;     /* Initially 0 */
sem_t mutex, w; /* Both initially 1 */

void reader(void)
{
  while (1) {
    P(&mutex);
    readcnt++;
    if (readcnt == 1) /* First in */
      P(&w);
    V(&mutex);

    /* Reading happens here */

    P(&mutex);
    readcnt--;
    if (readcnt == 0) /* Last out */
      V(&w);
    V(&mutex);
  }
}
```

Writers:

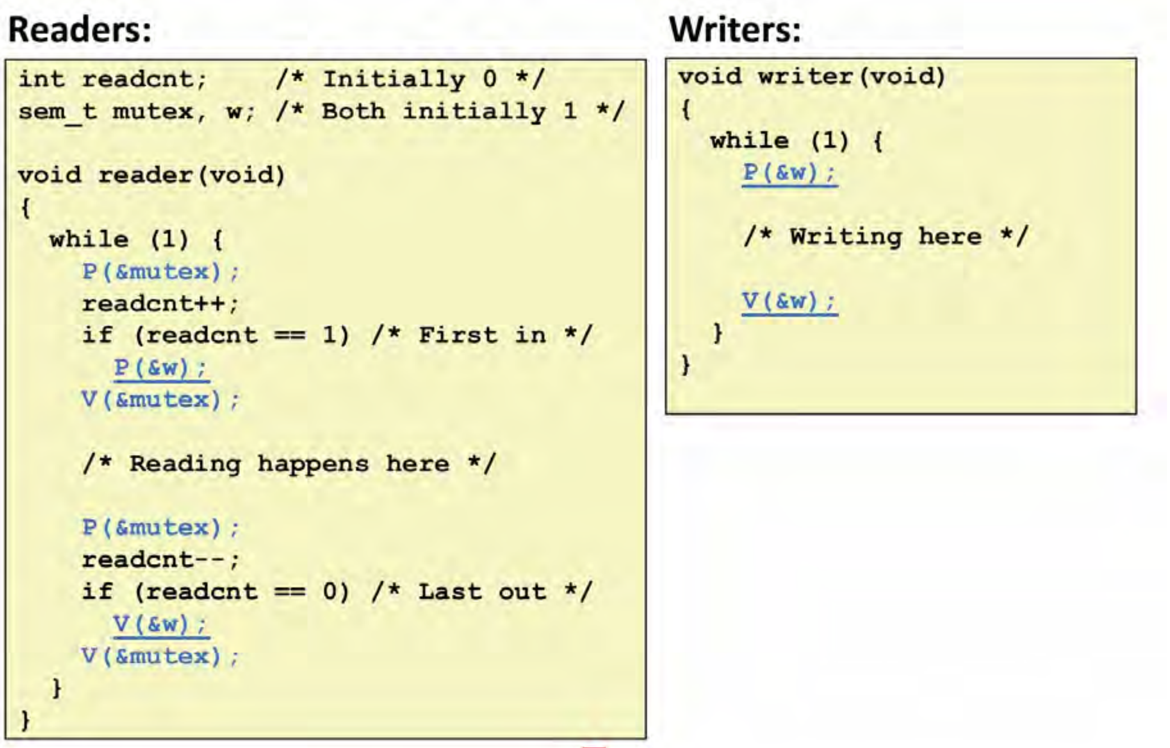
```
void writer(void)
{
  while (1) {
    P(&w);

    /* Writing here */

    V(&w);
  }
}
```

rw1.c

Arrivals: R1 R2 W1 R3

Readcnt == 2
W == 0

上述方案是可行的，但缺乏公平性，读者很容易饿死写者
(如果有连续的读者请求，那么写者会被无限期地阻塞)
一种解决方案是: 当有写者等待时，应避免更多的读者获取读锁.

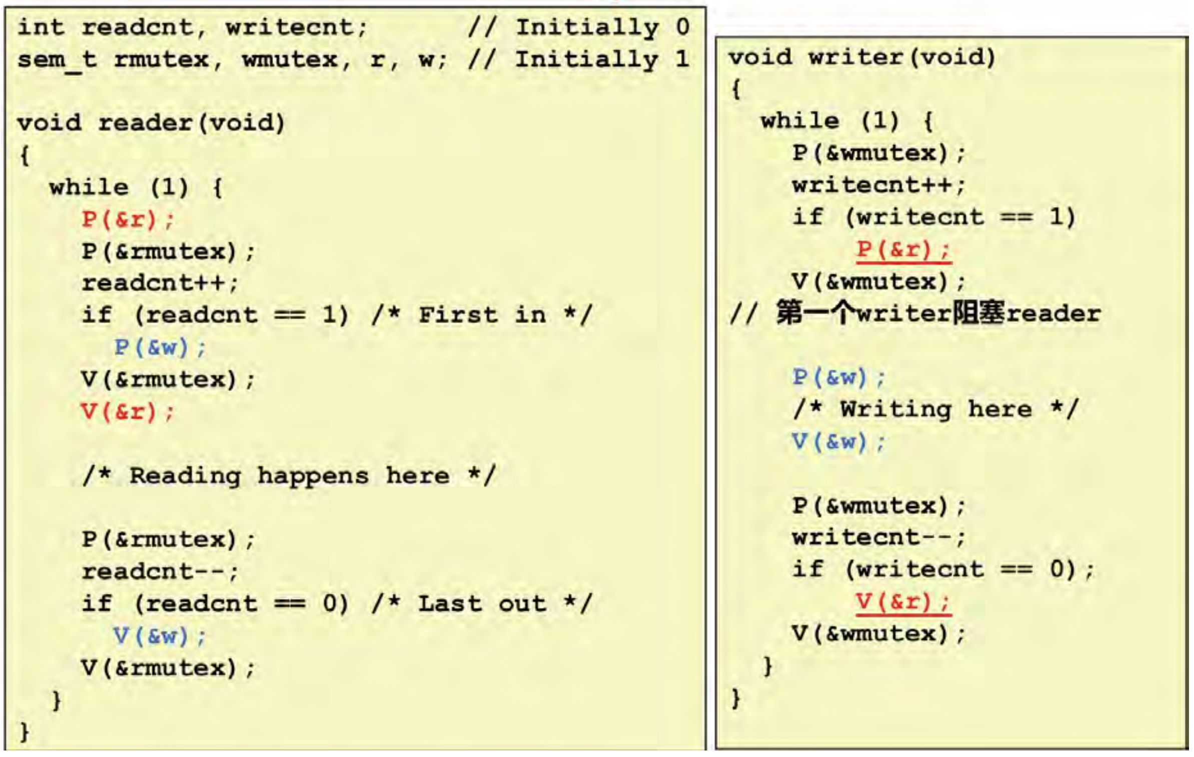


Readers:

```
int readcnt;     /* Initially 0 */
sem_t mutex, w; /* Both initially 1 */

void reader(void)
{
  while (1) {
    P(&mutex);
    readcnt++;
    if (readcnt == 1) /* First in */
      P(&w);
    V(&mutex);

    /* Reading happens here */

    P(&mutex);
    readcnt--;
    if (readcnt == 0) /* Last out */
      V(&w);
    V(&mutex);
  }
}
```

Writers:

```
void writer(void)
{
  while (1) {
    P(&w);

    /* Writing here */

    V(&w);
  }
}
```



```
int readcnt, writecnt;        // Initially 0
sem_t rmutex, wmutex, r, w; // Initially 1

void reader(void)
{
  while (1) {
    P(&r);
    P(&rmutex);
    readcnt++;
    if (readcnt == 1) /* First in */
      P(&w);
    V(&rmutex);
    V(&r);

    /* Reading happens here */

    P(&rmutex);
    readcnt--;
    if (readcnt == 0) /* Last out */
      V(&w);
    V(&rmutex);
  }
}
```

```
void writer(void)
{
  while (1) {
    P(&wmutex);
    writecnt++;
    if (writecnt == 1)
      P(&r);
    V(&wmutex);
// 第一个writer阻塞reader

    P(&w);
    /* Writing here */
    V(&w);

    P(&wmutex);
    writecnt--;
    if (writecnt == 0);
      V(&r);
    V(&wmutex);
  }
}
```

更多内容参考 Homework 3

3.6.7 哲学家就餐问题

哲学家就餐问题 (dining philosopher’s problem) 名气很大，也很有趣，但实用性却不强.

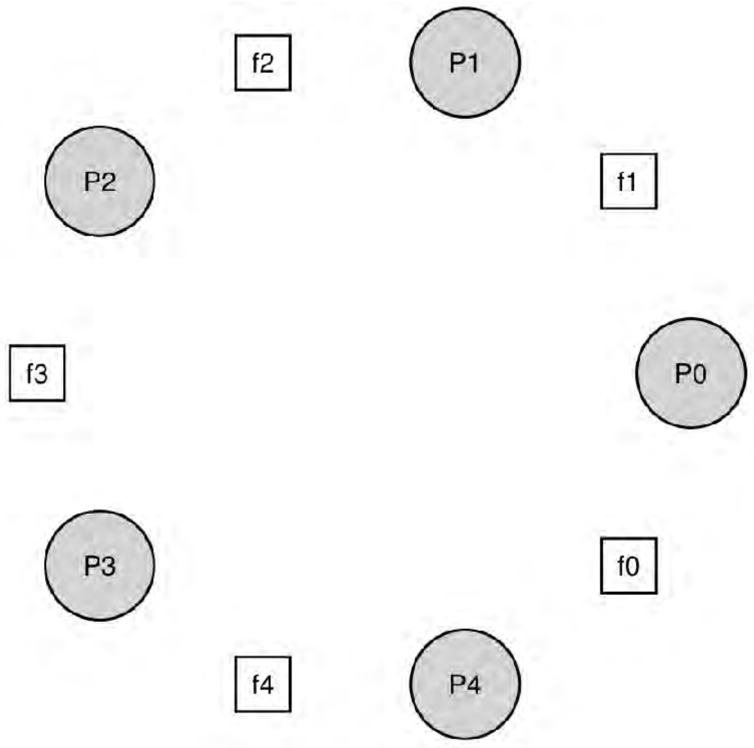
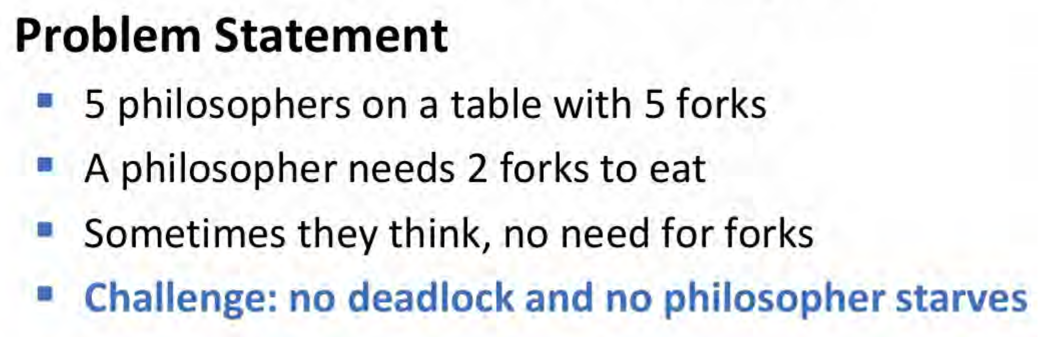


Figure 31.14: **The Dining Philosophers**



Problem Statement

- 5 philosophers on a table with 5 forks
- A philosopher needs 2 forks to eat
- Sometimes they think, no need for forks
- **Challenge: no deadlock and no philosopher starves**

假定有 5 位哲学家围着一个圆桌，每 2 位哲学家之间有 1 把餐叉 (一共 5 把)
哲学家有时要思考，不需要餐叉; 有时又要就餐——只有同时拿到左右两把餐叉才能吃到东西.
下面是每个哲学家的基本循环:

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

问题的关键在于如何实现 `getforks()` 和 `putforks()` 函数.
我们需要确保没有死锁，没有哲学家饿死，并且并发度高 (尽可能让更多哲学家同时吃东西)
我们首先定义辅助函数，来帮助构建解决方案:

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

假设为 5 把餐叉分别设置一个信号量: `sem_t forks[5]`，并初始化为 1
假设每个哲学家都知道自己的编号 `p`
为了拿到餐叉，哲学家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.

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

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

这个简单的实现会造成**死锁** (deadlock) 的问题.
如果每个哲学家同时拿到了左手边的餐叉，那么他们都会阻塞住 (一直等待右手边的餐叉)

解决上述问题最简单的方法，就是修改某个或者某些哲学家的取餐叉顺序.
我们不妨让第 4 号哲学家 (偏好最大的那个) 按照相反的顺序取餐叉——先是右手边的，然后是左手边的.
这样就可以**打破等待循环**.

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

在此实现下，最多可有两名哲学家同时进餐.

3.7 常见并发问题

3.7.1 非死锁缺陷

非死锁问题占了并发问题的大多数.
我们现在主要讨论其中两种: **违反原子性缺陷** (atomicity violation bugs) 和**违反顺序缺陷** (order violation bugs)

(1) 违反原子性缺陷

违反原子性的定义是:
违反了多次内存访问中预期的可串行性 (即代码段本意是原子的，但在执行中并没有强制实现原子性)

考虑 MYSQL 中的一个例子:

```
// Thread 1:
// Check if `thd->proc_info` is not NULL before accessing it
if (thd->proc_info) {
    // Potential concurrency bug here:
```

```
    // Between this check and the actual access in fputs,
    // another thread could modify `thd->proc_info`, making it NULL,
    // leading to a segmentation fault or unintended behavior.

    // Attempt to use `thd->proc_info`, assuming it is valid
    fputs(thd->proc_info, ...);  // If another thread modifies `thd->proc_info`
to NULL here,
                                  // this line may cause a segmentation fault.
    ...
}

// Thread 2:
// Set `thd->proc_info` to NULL, possibly while Thread 1 is using it
thd->proc_info = NULL;  // This action is unsynchronized and can affect Thread
1's operations,
// creating a race condition.
```

上述例子中，两个线程都要访问 `thd` 结构中的成员 `proc_info`
第一个线程检查 `proc_info` 是否非空，如果非空就打印出值;
第二个线程设置 `proc_info` 为空.
假设 `proc_info` 非空，当第一个线程检查之后，在 `fputs()` 调用之前被中断，
第二个线程抢先运行把指针 `proc_info` 置为空.
当第一个线程恢复执行时，由于引用空指针，会触发段错误 (segmentation fault) 导致程序奔溃.

问题的根源在于 `proc_info` 的非空检查和 `fputs()` 调用打印 `proc_info` 是假设原子的.
当假设不成立时，代码就出问题了.
我们要给共享变量的访问加锁，确保每个线程访问 `proc_info` 字段时， 都应先获取并持有锁 `proc_info_lock`

```
// Initialize a mutex for synchronizing access to `thd->proc_info`.
pthread_mutex_t proc_info_lock = PTHREAD_MUTEX_INITIALIZER;

// Thread 1:
// Acquire the mutex lock before checking and using `thd->proc_info`.
// This ensures that Thread 2 cannot modify `thd->proc_info` while Thread 1 is
accessing it.
pthread_mutex_lock(&proc_info_lock);

// Check if `thd->proc_info` is not NULL before proceeding.
// The lock ensures this check and the subsequent access to `thd->proc_info` are
atomic,
// preventing Thread 2 from setting it to NULL in between.
if (thd->proc_info) {
    // Safe to use `thd->proc_info` here since the lock prevents concurrent
modifications.
    fputs(thd->proc_info, ...);  // Write the content of `thd->proc_info`
without risk of it being NULL.
    ...
}

// Release the mutex lock, allowing other threads to access `thd->proc_info`.
pthread_mutex_unlock(&proc_info_lock);


// Thread 2:
// Acquire the same mutex lock before modifying `thd->proc_info`.
// This ensures Thread 1 will not access `thd->proc_info` while Thread 2 modifies
it.
pthread_mutex_lock(&proc_info_lock);

// Set `thd->proc_info` to NULL safely, knowing that Thread 1 cannot be using it
concurrently.
thd->proc_info = NULL;

// Release the mutex lock, allowing other threads to proceed with their
operations on `thd->proc_info`.
pthread_mutex_unlock(&proc_info_lock);
```

(2) 违反顺序缺陷

违反顺序的定义是:
两个内存访问的预期顺序被打破了 (即 A 应该在 B 之前执行，但是实际运行中却不是这个顺序)
考虑如下代码:

```
// Thread 1
void init() {
    // Initialize some resources or states.
    ...

    // Create a new thread `mThread` that will execute the function `mMain`.
    // This call starts Thread 2, which begins executing `mMain` almost
immediately.
    mThread = PR_CreateThread(mMain, ...);

    // Continue with other initialization after creating the thread.
    ...
}

// Thread 2 (started by Thread 1):
void mMain(...) {
    ...

    // Concurrently access `mThread`'s state.
    // BUG: There is a potential race condition here, as `mThread` might not be
fully
    // initialized by Thread 1 when Thread 2 attempts to access `mThread-
>State`.
    // If `init` hasn't fully completed in Thread 1, `mThread` may be in an
undefined state.
    mState = mThread->State;

    ...
}
```

线程 2 的代码似乎假定变量 `mThread` 已经被初始化了.
然而如果线程 1 没有先于线程 2 执行，那么线程 2 就可能因为引用空指针而崩溃.
我们通过强制顺序来修复这种缺陷，其中条件变量就是一种简单可靠的方式.
(当线程之间的执行顺序很重要时，条件变量 (或信号量) 能够用于解决问题)

```
// Define a mutex and a condition variable for thread synchronization.
```

```
pthread_mutex_t mtLock = PTHREAD_MUTEX_INITIALIZER; // Mutex to protect access
to `mtInit`.
pthread_cond_t mtCond = PTHREAD_COND_INITIALIZER;  // Condition variable to
signal initialization.
int mtInit = 0;                                    // Flag to indicate if
initialization is complete.

// Thread 1: Performs initialization and signals completion.
void init() {
    ...

    // Create a new thread that will execute `mMain`.
    mThread = PR_CreateThread(mMain, ...);

    // Signal that the thread has been created and is ready.
    pthread_mutex_lock(&mtLock);      // Lock the mutex to safely modify
`mtInit`.
    mtInit = 1;                       // Set the initialization flag to indicate
readiness.
    pthread_cond_signal(&mtCond);     // Signal the condition variable to wake
up Thread 2.
    pthread_mutex_unlock(&mtLock);    // Unlock the mutex to allow other threads
to proceed.

    ...
}

// Thread 2: Waits for initialization to complete before proceeding.
void mMain(...) {
    ...

    // Wait for the initialization flag to be set.
    pthread_mutex_lock(&mtLock);      // Lock the mutex to check the value of
`mtInit`.
    while (mtInit == 0) {             // While initialization is not complete,
        pthread_cond_wait(&mtCond, &mtLock); // Wait on the condition variable.
    }
    pthread_mutex_unlock(&mtLock);    // Unlock the mutex after initialization
is confirmed.

    // Safely access the state of `mThread`.
    mState = mThread->State;

    ...
}
```

在修正的代码中，我们增加了一个锁 (`mtLock`)、一个条件变量 (`mtCond`) 以及状态的变量 (`mtInit`)
初始化代码运行时，会将 `mtInit` 设置为 1，并发出信号表明它已做了这件事.
如果线程 2 先运行，就会一直等待信号和对应的状态变化;
如果后运行，则线程 2 会检查是否初始化 (即 `mtInit` 被设置为 1)，然后正常运行.

我们还可以用 `mThread` 本身作为状态变量:

```
// Define a mutex and a condition variable for thread synchronization.
pthread_mutex_t mtLock = PTHREAD_MUTEX_INITIALIZER; // Mutex to protect access
to `mThread`.
```

```
pthread_cond_t mtCond = PTHREAD_COND_INITIALIZER;  // Condition variable to
signal initialization.
PR_Thread *mThread = NULL;                         // Pointer to the created
thread, used as the state variable.

// Thread 1: Performs initialization and signals completion.
void init() {
    ...

    // Create a new thread that will execute `mMain`.
    pthread_mutex_lock(&mtLock);      // Lock the mutex to ensure thread-safe
modification of `mThread`.
    mThread = PR_CreateThread(mMain, ...); // Initialize `mThread` with the
created thread pointer.
    pthread_cond_signal(&mtCond);     // Signal the condition variable to wake
up Thread 2.
    pthread_mutex_unlock(&mtLock);    // Unlock the mutex to allow other threads
to proceed.

    ...
}

// Thread 2: Waits for initialization to complete before proceeding.
void mMain(...) {
    ...

    // Wait for `mThread` to be initialized.
    pthread_mutex_lock(&mtLock);      // Lock the mutex to check the value of
`mThread`.
    while (mThread == NULL) {         // While `mThread` has not been
initialized,
        pthread_cond_wait(&mtCond, &mtLock); // Wait on the condition variable.
    }
    pthread_mutex_unlock(&mtLock);    // Unlock the mutex after initialization
is confirmed.

    // Safely access the state of `mThread`.
    mState = mThread->State;

    ...
}
```

(3) 竞争

当程序的正确性取决于一个线程在另一个线程到达点 y 之前到达点 x 时，就会发生**竞争条件** (race condition)

```
/* A threaded program with a race condition */
int main(int argc, char** argv) {
    pthread_t tid[N];  // Array to store thread IDs
    int i;

    // Loop to create N threads
    for (i = 0; i < N; i++)
        // BUG: Passing the address of `i` to each thread.
        // This introduces a race condition because `i` is shared across all
threads.
        // Each thread will read the value of `i` at an unpredictable time, so
they may all print
        // the same or incorrect values depending on when they read `i`.
        Pthread_create(&tid[i], NULL, thread, &i);

    // Wait for all threads to complete
    for (i = 0; i < N; i++)
        Pthread_join(tid[i], NULL);

    return 0;
}

/* Thread routine */
void *thread(void *vargp) {
    // Dereference the shared `i` variable passed to each thread.
    // BUG: All threads access the same address (&i), so the value of `myid`
depends on when
    // the thread reads `i`, which may change due to the main loop's increments.
    int myid = *((int *)vargp);

    // Print message with the thread ID.
    // This may print duplicate or unexpected IDs due to the race condition.
    printf("Hello from thread %d\n", myid);

    return NULL;
}
```

Make sure the threads don't have unintended sharing of state:

```
/* A threaded program without race conditions */
int main(int argc, char** argv) {
    pthread_t tid[N];
    int i;
    // Create each thread with a unique copy of 'i' to avoid race conditions
    for (i = 0; i < N; i++) {
        // Allocate memory for each thread's 'i' value
        int *valp = Malloc(sizeof(int));

        // Store the current value of 'i' in the allocated memory
        *valp = i;

        // Pass the unique pointer 'valp' to the new thread, ensuring each
thread has its own copy
        Pthread_create(&tid[i], NULL, thread, valp);
    }

    // Wait for each thread to finish
    for (i = 0; i < N; i++)
        Pthread_join(tid[i], NULL);

    return 0;
}

/* Thread routine */
void *thread(void *vargp) {
    // Dereference the pointer to get the thread-specific ID
    int myid = *((int *)vargp);

    // Free the dynamically allocated memory after retrieving the ID
    Free(vargp);

    // Print a message including the unique thread ID
    printf("Hello from thread %d\n", myid);

    return NULL;
}
```

This version is also correct:

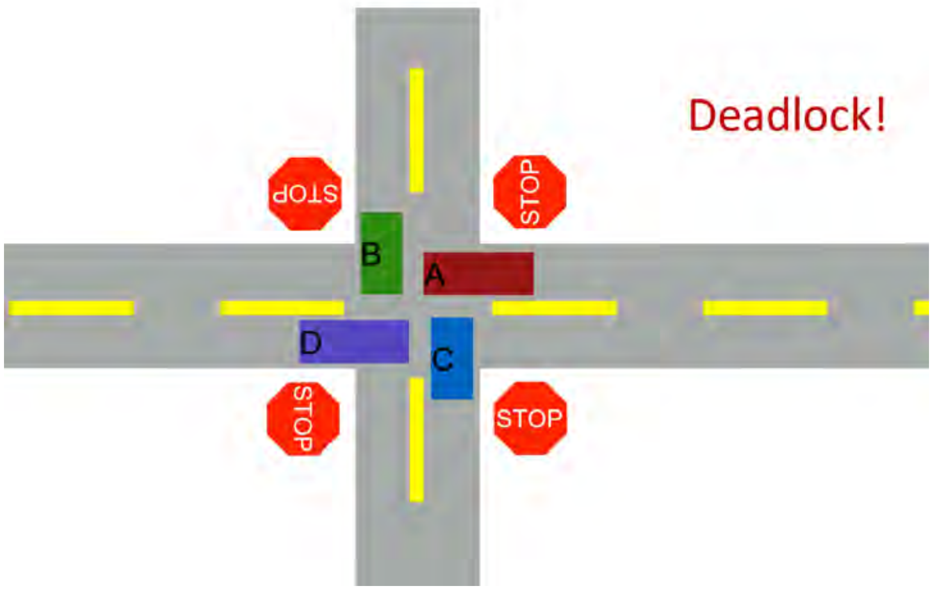
```
/* Main function */
int main(int argc, char** argv) {
    pthread_t tid[N];  // Array to store thread IDs
    uint64_t i;        // Loop variable and thread argument

    // Create N threads
    for (i = 0; i < N; i++) {
        // Create a new thread, passing `i` as a `void *` argument to the thread
routine
        // WARNING: Casting an integer to a pointer directly is platform-
dependent and
        // may lead to issues on systems where pointers and integers have
different sizes.
        Pthread_create(&tid[i], NULL, thread, (void *)i);
    }

    // Wait for all threads to complete
    for (i = 0; i < N; i++) {
        // Join each thread to ensure it has finished execution before
continuing
        Pthread_join(tid[i], NULL);
    }

    return 0;  // Exit the program
}

/* Thread routine */
void *thread(void *vargp) {
    // Retrieve the thread-specific ID by casting the `void *` argument back to
`uint64_t`.
    // WARNING: This relies on the assumption that pointers and `uint64_t` are
the same size.
    uint64_t myid = (uint64_t)vargp;

    // Print a message with the thread-specific ID.
    // Each thread should correctly print its unique ID, as the loop variable `i`
was passed by value.
    printf("Hello from thread %lu\n", myid);

    return NULL;  // Exit the thread
}
```

3.7.2 死锁缺陷

(1) 定义

死锁 (Deadlock):
当两个或多个线程相互等待对方采取某个操作时，程序无法继续执行，导致各线程都无法完成任务.
这种现象被称为死亡拥抱 (deadly embrace)
如果一个进程正在等待一个永远不会成立的条件，那么这个进程就进入了死锁状态.

考虑一个形象的例子:
A 等 B 先走，B 等 C 先走，C 等 D 先走，D 等 A 先走，结果谁也走不了.



死锁产生的一个主要原因是在大型的代码库里，组件之间会有复杂的依赖.
在设计大型系统的锁机制时，我们要注意避免循环依赖导致的死锁.
考虑一个死锁的例子:

```
// Thread 1 and Thread 2 each attempt to acquire locks in a different order, leading to potential deadlock.
Thread 1:                   Thread 2:
    lock(L1);                   lock(L2);       // Thread 1 locks L1, while Thread 2 locks L2.
    lock(L2);                   lock(L1);       // Thread 1 tries to lock L2, but it's held by Thread 2.
                                                // Thread 2 tries to lock L1, but it's held by Thread 1.
```

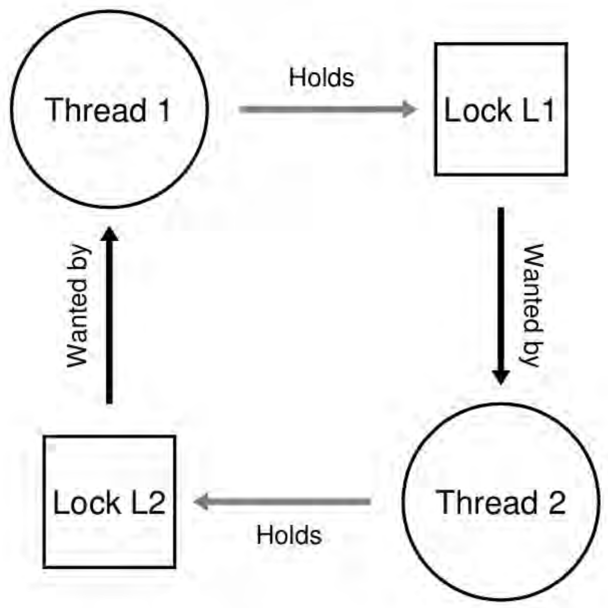


Figure 32.2: **The Deadlock Dependency Graph**

死锁产生的另一个主要原因是**封装** (encapsulation)
软件开发者一直倾向于隐藏实现细节，以模块化的方式让软件开发更容易.
然而模块化和锁不是很契合，某些看起来毫无关系的接口可能会导致一些难以发现的死锁.
考虑以下例子:

```
set_t *set_intersection (set_t *s1, set_t *s2) {
    // Allocate memory for the resulting set `rv` to store the intersection of s1
and s2.
    set_t *rv = Malloc(sizeof(*rv));

    // Attempt to lock both s1 and s2 to ensure thread-safe access to both sets.
    // Bug: The order of locking here may lead to deadlock if another thread
    // tries to lock s2 first and then s1.
    Mutex_lock(&s1->lock);   // Lock set s1.
    Mutex_lock(&s2->lock);   // Lock set s2.

    // Iterate over elements in s1 to find common elements in s2.
    for(int i = 0; i < s1->len; i++)
    {
        if(set_contains(s2, s1->items[i]))    // Check if s1's item exists in
s2.
            set_add(rv, s1->items[i]);        // Add common items to the result
set `rv`.
    }

    // Unlock both sets after processing.
    // Bug: If an error occurs in processing, locks might not be released.
    Mutex_unlock(&s2->lock);   // Unlock set s2.
    Mutex_unlock(&s1->lock);   // Unlock set s1.
}
```

如果一个线程调用 `rv = set_intersection(setA, setB)` (试图先锁定 `setA->lock` 再锁定 `setB->lock`),
同时另一个线程调用 `rv = set_intersection(setB, setA)` (试图先锁定 `setB->lock` 再锁定 `setA->lock`),
就会造成死锁.

假设结构体 `set_t` 具有某种可以提供排序的属性 (记为 `order`),
则我们可以将 `set_intersection()` 函数修改为:

```
set_t *set_intersection(set_t *s1, set_t *s2) {
    set_t *rv = Malloc(sizeof(*rv));  // Allocate memory for the result set.

    // Check if the two sets are the same.
    if (s1->order == s2->order)
    {
        // If the sets are the same, we only need to lock one of them.
        Mutex_lock(&s1->lock);  // Lock the set once.
    }
    else
    {
        // Otherwise, lock the sets in a consistent order.
        if (s1->order < s2->order)
        {
            Mutex_lock(&s1->lock);  // Lock the first set.
            Mutex_lock(&s2->lock);  // Lock the second set.
        }
        else
        {
            Mutex_lock(&s2->lock);  // Lock the second set first.
            Mutex_lock(&s1->lock);  // Lock the first set second.
        }
    }

    // Iterate over elements in s1 to find common elements in s2.
    for (int i = 0; i < s1->len; i++) {
        if (set_contains(s2, s1->items[i])) {  // Check if s1's item exists in
s2.
            set_add(rv, s1->items[i]);  // Add common items to the result set
rv.
        }
    }

    // Unlock both sets after processing.
    if (s1->order == s2->order)
    {
        Mutex_unlock(&s1->lock);  // Unlock the set once.
    }
    else
    {
        // Otherwise, lock the sets in a consistent order.
        if (s1->order < s2->order)
        {
            Mutex_unlock(&s1->lock);  // Unlock the first set.
            Mutex_unlock(&s2->lock);  // Unlock the second set.
        }
        else
        {
            Mutex_unlock(&s2->lock);  // Unlock the second set first.
            Mutex_unlock(&s1->lock);  // Unlock the first set second.
        }
    }

    return rv;  // Return the result set containing the intersection.
}
```

死锁产生的四个条件:

- **互斥 (mutual exclusion):** 在任何时刻，资源只能被一个线程占用
- **持有并等待 (hold-and-wait):** 一个线程在等待其他资源的同时，已经持有了一部分资源
  这个条件不好解决，只能加一把大锁原子地获取所有的锁，
  但这无疑会造成加锁的粒度过粗，而且不适用于封装.
- **不可抢占 (no preemption):** 已经分配给线程的资源，在线程使用完之前不能被强行抢占
- **循环等待 (circular wait):** 存在一个线程等待的闭环，其中的每个线程都持有其他线程需要的资源

(2) Wait-free

首先考虑 "互斥" 条件的解决.
Herlihy 提出了**无等待** (wait-free) 数据结构的思想:
通过强大的硬件指令，我们可以构造出不需要锁的数据结构.
例如通过比较-交换指令 (`CompareAndSwap`)，我们可以原子地给某个值增加特定的数量:

```
void AtomicIncrement(int *value, int amount) {
    int old;
    do {
        // Step 1: Store the current value in a local variable
        int old = *value;

        // Step 2: Attempt to atomically update the value by adding `amount`
        // The CompareAndSwap operation ensures that the update happens only
        // if the current value matches the value in `old`. If successful,
        // it updates the value to `old + amount`. If not, it retries.
    } while (CompareAndSwap(value, old, old + amount) == 0);
}
```

上述实现不会出现死锁 (因为它没有锁使用)，但可能出现活锁 (如果多个线程频繁地竞争同一个变量的话)
更复杂的例子: 考虑使用锁来进行链表插入操作

```
void insert(int value) {
    // Step 1: Allocate memory for a new node
    node_t *n = malloc(sizeof(node_t));
    // Step 2: Assert that memory allocation was successful
    assert(n != NULL);
    // Step 3: Assign the value to the new node's `value` field
    n->value = value;

    // Step 4: Begin critical section by acquiring the lock
    lock(listlock);
    // Critical section starts here. The following operations
    // modify the shared data structure (linked list).
    // Step 5: Insert the new node at the head of the list
    // The current head is assigned to the `next` pointer of the new node.
    n->next = head;
    // Step 6: Update the head of the list to point to the new node
    head = n;

    // Step 7: End critical section by releasing the lock
    unlock(listlock);
}
```

我们可以使用比较-交换指令 (`CompareAndSwap`) 来实现它:

```
void insert(int value) {
    // Step 1: Allocate memory for a new node
    node_t *n = malloc(sizeof(node_t));
    // Step 2: Assert that memory allocation was successful
    assert(n != NULL);
    // Step 3: Assign the given value to the new node's `value` field
    n->value = value;

    // Step 4: Begin atomic insertion into the linked list
    do {
        // Step 4.1: Set the `next` pointer of the new node to the current head
of the list
        n->next = head;
        // Step 4.2: Attempt to atomically update the head pointer
        // CompareAndSwap checks if `head` is still equal to `n->next`.
        // If true, it updates `head` to point to `n` (the new node).
        // If false, it means another thread has modified `head`, and the loop
retries.
    } while (CompareAndSwap(&head, n->next, n) == 0);
}
```

这段代码首先把 `n->next` 指针指向当前的链表头 `head`，然后试着把新节点 `n` 交换到链表头 `head`. 但如果此时其他的线程成功地修改了 `head` 的值，这里的交换就会失败，导致这个线程根据新的 `head` 值重试.

无锁算法虽高效，但在高并发场景下容易出现活锁.
某些情况下可以考虑引入少量锁或条件变量，以减少无效重试.

(3) Exponential back-off

其次考虑 "不可抢占" 条件的解决.
我们可以使用 `trylock()` 函数来实现无死锁的加锁方式:
(即重复尝试获取所有资源，如果有一步出错，则释放已经获取的资源，再重新尝试)

```
top:
    lock(L1);  // Lock the first mutex (L1)

    // Try to lock the second mutex (L2)
    if (trylock(L2) == -1) {  // If trylock fails (returns -1), it means L2 is
already locked
```

```
        unlock(L1);  // Unlock the first mutex (L1) if unable to lock the second
one
        goto top;    // Retry the operation by going back to the top (re-lock L1
and try L2 again)
    }

    // If both mutexes are successfully locked, proceed with the work

    // Unlock both mutexes after completing the work
    unlock(L1);
    unlock(L2);
```

另一个线程可以使用相同的加锁顺序 (先 `L1` 后 `L2`) 或者不同的加锁顺序 (先 `L2` 后 `L1`)，程序都不会发生死锁.
不过这种方法的一个问题便是**封装**:
如果一个锁是封装在函数内部的，则上述代码中的 `goto` 步骤是很难实现的.

这种方法也引入了一个新的问题: **活锁** (livelock)
两个线程有可能一直重复上述序列，又同时都抢锁失败.
此时尽管系统一直在运行这段代码，但又没有任何实质性的进展.
我们可以在 `trylock()` 失败后随机等待一个时间，然后再重复 `trylock()`，这样可以降低线程之间的重复互相干扰.
一个经典的策略是**指数退避** (exponential back-off):
使每次失败后等待的时间增加，直到成功获取锁或达到最大重试次数.
(达到最大重试次数的线程可以放入队列，等到系统资源可用时重新尝试)

```
i = 1;  // Initialize a variable i, which will be used for the backoff time.

// Start of the loop
top:
lock(A);  // Lock mutex A.

if (trylock(B) == -1) {  // Try to lock mutex B. If it fails (trylock returns
-1),
    unlock(A);  // Unlock mutex A to allow other threads to access it.

    sleep(i);  // Sleep for 'i' milliseconds to perform a backoff. This allows
other threads to acquire locks.
    i *= 2;  // Double the backoff time for the next attempt.

    goto top;  // Jump back to the top of the loop to try locking A and B again.
}
```

为避免活锁问题，我们可以在冲突解决机制中引入随机性.
我们还可为线程设置优先级，以避免某些线程无限制地让出资源，确保系统中总有线程能够取得进展.
可采用**动态优先级调整**，如线程重试次数越多，优先级越高.

(4) Lock Ordering

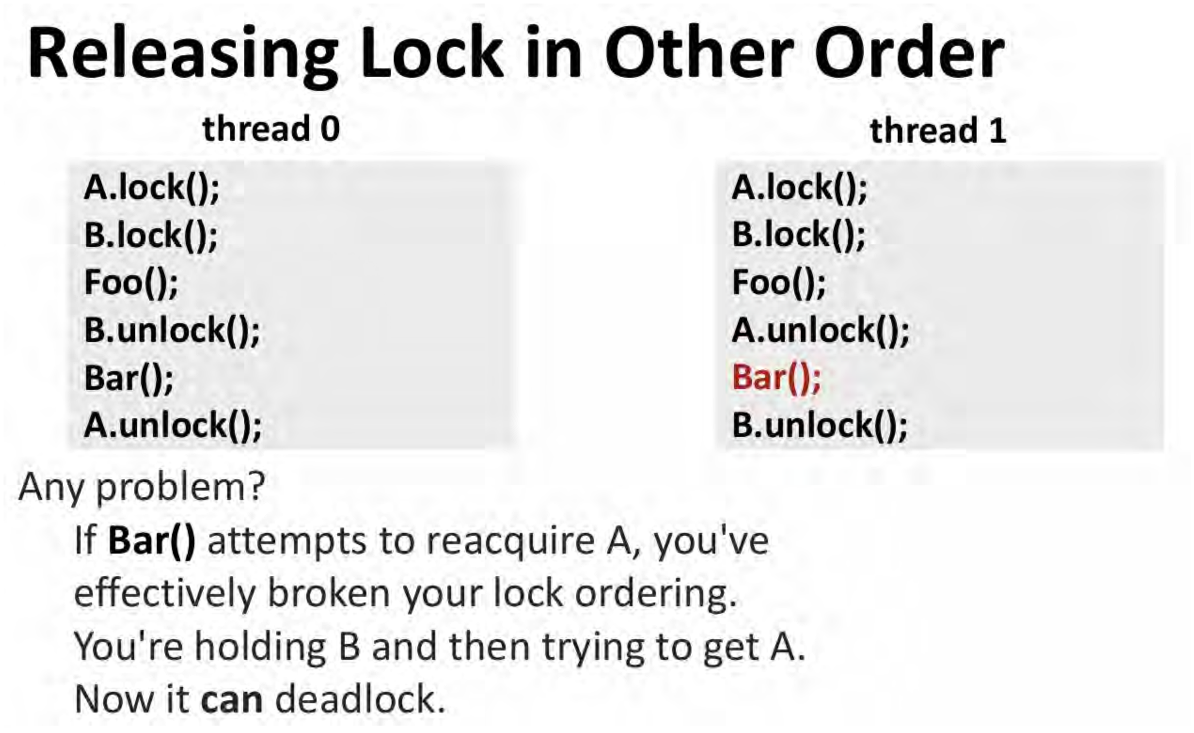
最后考虑 "循环等待" 条件的解决，这也许是最实用的预防死锁的技术.
最直接的方法就是获取锁时提供一个**全序** (total ordering)
假如系统共有两个锁 (`L1` 和 `L2`)， 那么我们每次都先申请 `L1` 然后申请 `L2`，就可以避免死锁.
这样严格的顺序避免了循环等待，也就不会产生死锁.

复杂系统中不会只有两个锁，因此锁的全序可能很难做到，**偏序** (partial ordering) 可能是一种有用的方法.
例如 Linux 的内存映射代码就是偏序锁的一个好例子，代码开头的注释表明了 10 组不同的加锁顺序.
显然全序和偏序都需要细致的锁策略的设计和实现，
而且它们只是一种约定，粗心的程序员很容易忽略，导致死锁.

当一个函数要抢多个锁时，我们需要注意死锁.
假设函数 `do_something(mutex_t *m1, mutex_t *m2)` 总是先抢 `m1`，然后 `m2`，
那么当一个线程调用 `do_something(L1, L2)`，而另一个线程调用 `do_something(L2, L1)` 时，就可能会产生死锁.
为了避免这种问题，我们可以根据锁的地址作为获取锁的顺序.
按照地址从高到低，或者从低到高的顺序加锁，就可以保证不论传入参数是什么顺序，函数都会以固定的顺序加锁.
具体的代码如下:

```
// Function to perform some operation with two mutexes
void do_something(pthread_mutex_t *m1, pthread_mutex_t *m2) {
    // Ensure that the mutexes are locked in a fixed order based on their memory addresses
    if (m1 > m2) {  // If m1's address is higher than m2's, lock m1 first, then m2
        pthread_mutex_lock(m1);  // Lock the first mutex
        pthread_mutex_lock(m2);  // Lock the second mutex
    } else {  // If m2's address is higher or they are the same, lock m2 first, then m1
        pthread_mutex_lock(m2);  // Lock the first mutex
        pthread_mutex_lock(m1);  // Lock the second mutex
    }

    // Perform the actual work here, operating with both mutexes locked

    // Unlock the mutexes after the work is done
    pthread_mutex_unlock(m1);
    pthread_mutex_unlock(m2);
}
```

释放锁的顺序也有讲究:



Releasing Lock in Other Order

thread 0

```
A.lock();
B.lock();
Foo();
B.unlock();
Bar();
A.unlock();
```

thread 1

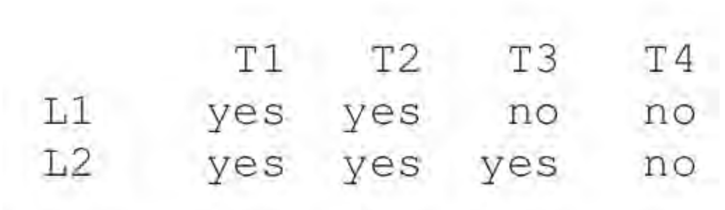
```
A.lock();
B.lock();
Foo();
A.unlock();
Bar();
B.unlock();
```

Any problem?

If **Bar()** attempts to reacquire A, you've effectively broken your lock ordering.
You're holding B and then trying to get A.
Now it **can** deadlock.

(5) 银行家算法

如果我们已知不同线程在运行中对锁的需求情况，那么我们就能通过调度来避免死锁.
一个简单的例子:
假设我们需要在 2 个处理器上调度 4 个线程，它们对锁的需求情况如下:

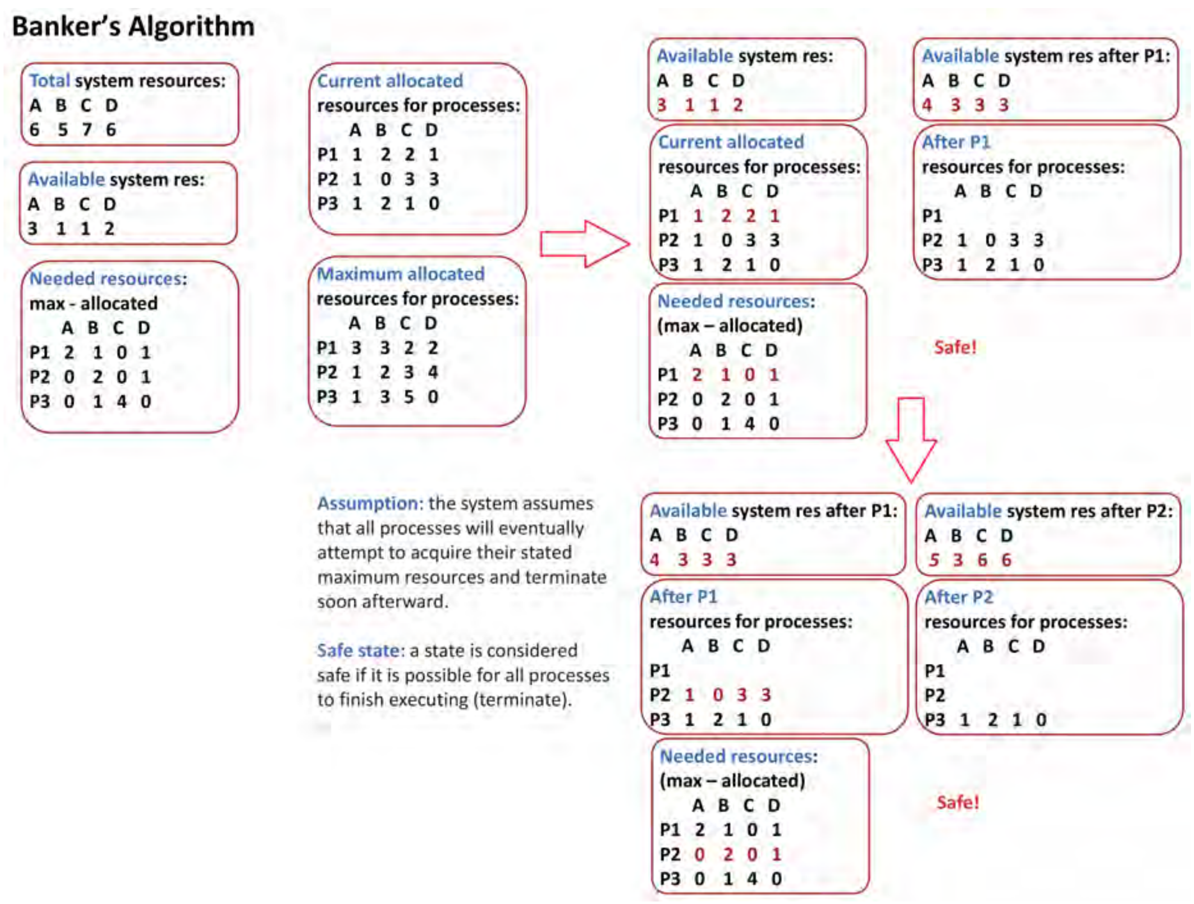


	T1	T2	T3	T4
L1	yes	yes	no	no
L2	yes	yes	yes	no

显然只要 T_1 和 T_2 不同时运行，就不会产生死锁.

Dijkstra 提出的**银行家算法**通过合理分配资源来避免死锁.
它通过模拟分配所有资源的预定最大可能数量来测试系统的安全性，
并在此过程中进行安全状态检查，以测试所有其他待执行活动是否可能导致死锁，
然后再决定是否允许继续进行资源分配.
因此它需要了解以下信息:

- 每个进程可能请求的每种资源的最大数量 (MAX)
- 每个进程当前持有的每种资源的数量 (ALLOCATED)
- 系统当前可用的每种资源的数量 (AVAILABLE)



Banker's Algorithm

Total system resources:

A	B	C	D
6	5	7	6

Available system res:

A	B	C	D
3	1	1	2

Needed resources:
max - allocated

	A	B	C	D
P1	2	1	0	1
P2	0	2	0	1
P3	0	1	4	0

Current allocated resources for processes:

	A	B	C	D
P1	1	2	2	1
P2	1	0	3	3
P3	1	2	1	0

Maximum allocated resources for processes:

	A	B	C	D
P1	3	3	2	2
P2	1	2	3	4
P3	1	3	5	0

Assumption: the system assumes that all processes will eventually attempt to acquire their stated maximum resources and terminate soon afterward.

Safe state: a state is considered safe if it is possible for all processes to finish executing (terminate).

Available system res:

A	B	C	D
3	1	1	2

Current allocated resources for processes:

	A	B	C	D
P1	1	2	2	1
P2	1	0	3	3
P3	1	2	1	0

Needed resources:
(max – allocated)

	A	B	C	D
P1	2	1	0	1
P2	0	2	0	1
P3	0	1	4	0

Available system res after P1:

A	B	C	D
4	3	3	3

After P1 resources for processes:

	A	B	C	D
P1				
P2	1	0	3	3
P3	1	2	1	0

Safe!

Available system res after P1:

A	B	C	D
4	3	3	3

After P1 resources for processes:

	A	B	C	D
P1				
P2	1	0	3	3
P3	1	2	1	0

Needed resources:
(max – allocated)

	A	B	C	D
P1	2	1	0	1
P2	0	2	0	1
P3	0	1	4	0

Available system res after P2:

A	B	C	D
5	3	6	6

After P2 resources for processes:

	A	B	C	D
P1				
P2				
P3	1	2	1	0

Safe!

总结:

- 如果对正确性存疑，宁愿限制并发 (即添加不必要的锁)
- 并发编程本身就很难，而封装 (encapsulation) 会让问题更复杂!
- 制定一个避免死锁的策略，并严格遵守
- 约定锁的获取顺序可能是最实用的方法

The End