Lecture Note 4. Concurrency: Thread and Lock

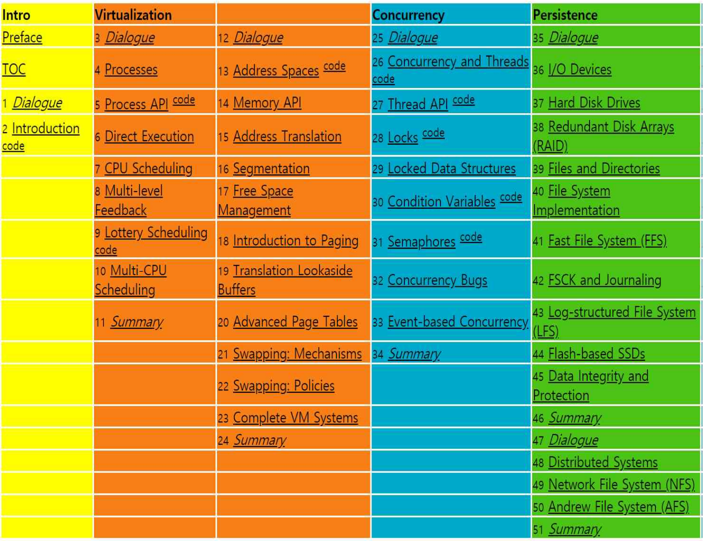
March 23, 2021

Jongmoo Choi

Dept. of software
Dankook University
http://embedded.dankook.ac.kr/~choijm

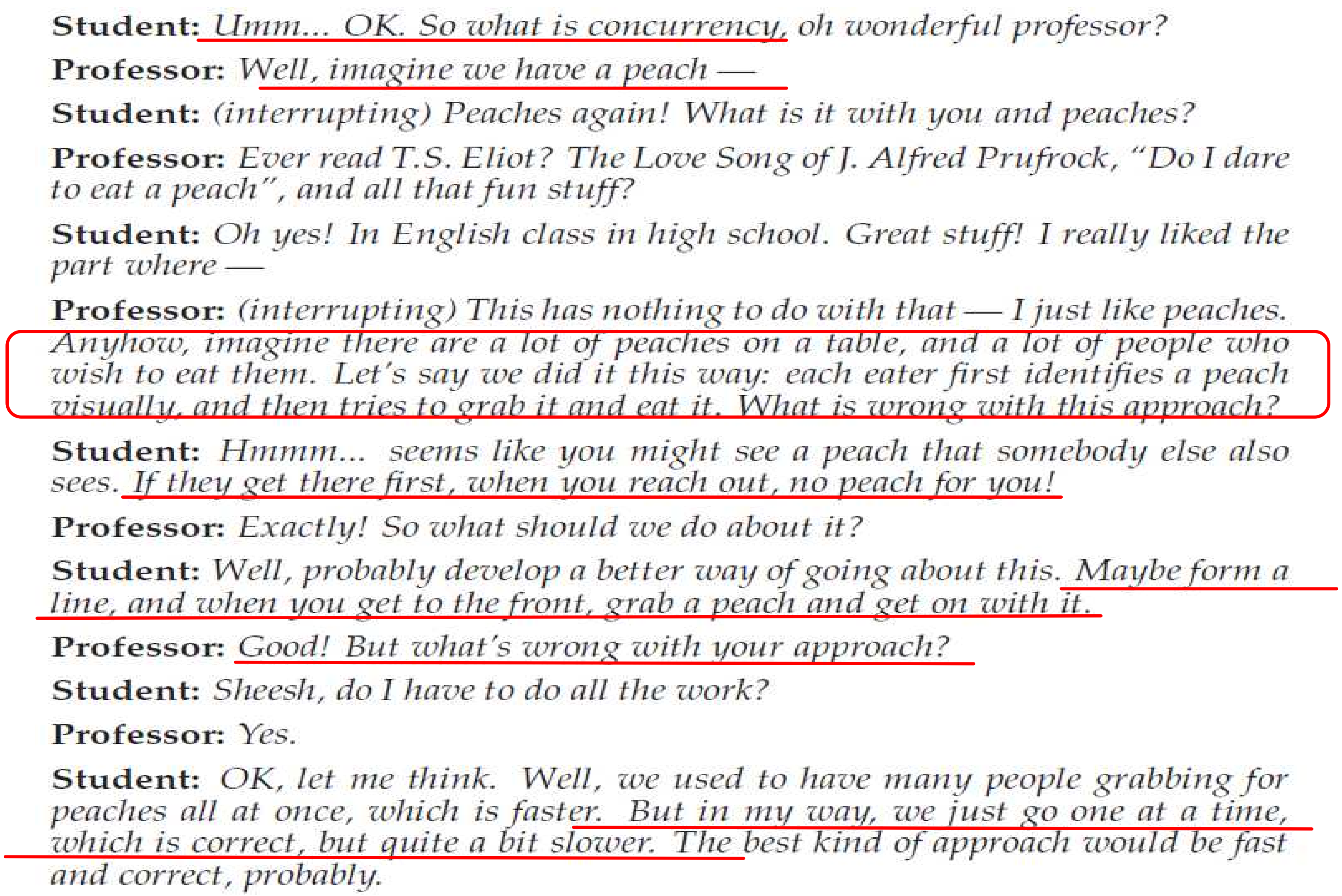
Contents

- From Chap 25~29 of the OSTEP
- Chap 25. A Dialogue on Concurrency
- Chap 26. Concurrency: An Introduction
  - ✓ Heart of problem: un-controlled schedule
  - ✓ Race condition, Mutual exclusion, Atomicity, …
- Chap 27. Interlude: Thread API
  - ✓ Thread vs. Process
  - ✓ Thread manipulation: creation, completion, mutex
- Chap 28. Locks
  - ✓ Evaluation method
  - ✓ Building method: Four atomic operations
  - ✓ Spin vs. Sleep
- Chap 29. Locked Data Structure
  - ✓ list, queue, hash, …



Intro	Virtualization		Concurrency	Persistence
Preface	3 *Dialogue*	12 *Dialogue*	25 *Dialogue*	35 *Dialogue*
TOC	4 Processes	13 Address Spaces code	26 Concurrency and Threads code	36 I/O Devices
1 *Dialogue*	5 Process API code	14 Memory API	27 Thread API code	37 Hard Disk Drives
2 Introduction code	6 Direct Execution	15 Address Translation	28 Locks code	38 Redundant Disk Arrays (RAID)
	7 CPU Scheduling	16 Segmentation	29 Locked Data Structures	39 Files and Directories
	8 Multi-level Feedback	17 Free Space Management	30 Condition Variables code	40 File System Implementation
	9 Lottery Scheduling code	18 Introduction to Paging	31 Semaphores code	41 Fast File System (FFS)
	10 Multi-CPU Scheduling	19 Translation Lookaside Buffers	32 Concurrency Bugs	42 FSCK and Journaling
	11 *Summary*	20 Advanced Page Tables	33 Event-based Concurrency	43 Log-structured File System (LFS)
		21 Swapping: Mechanisms	34 *Summary*	44 Flash-based SSDs
		22 Swapping: Policies		45 Data Integrity and Protection
		23 Complete VM Systems		46 *Summary*
		24 *Summary*		47 *Dialogue*
				48 Distributed Systems
				49 Network File System (NFS)
				50 Andrew File System (AFS)
				51 *Summary*

Chap. 25 A Dialogue on Concurrency



Student: *Umm... OK. So what is concurrency, oh wonderful professor?*

Professor: *Well, imagine we have a peach —*

Student: *(interrupting) Peaches again! What is it with you and peaches?*

Professor: *Ever read T.S. Eliot? The Love Song of J. Alfred Prufrock, "Do I dare to eat a peach", and all that fun stuff?*

Student: *Oh yes! In English class in high school. Great stuff! I really liked the part where —*

Professor: *(interrupting) This has nothing to do with that — I just like peaches. Anyhow, imagine there are a lot of peaches on a table, and a lot of people who wish to eat them. Let's say we did it this way: each eater first identifies a peach visually, and then tries to grab it and eat it. What is wrong with this approach?*

Student: *Hmmm... seems like you might see a peach that somebody else also sees. If they get there first, when you reach out, no peach for you!*

Professor: *Exactly! So what should we do about it?*

Student: *Well, probably develop a better way of going about this. Maybe form a line, and when you get to the front, grab a peach and get on with it.*

Professor: *Good! But what's wrong with your approach?*

Student: *Sheesh, do I have to do all the work?*

Professor: *Yes.*

Student: *OK, let me think. Well, we used to have many people grabbing for peaches all at once, which is faster. But in my way, we just go one at a time, which is correct, but quite a bit slower. The best kind of approach would be fast and correct, probably.*

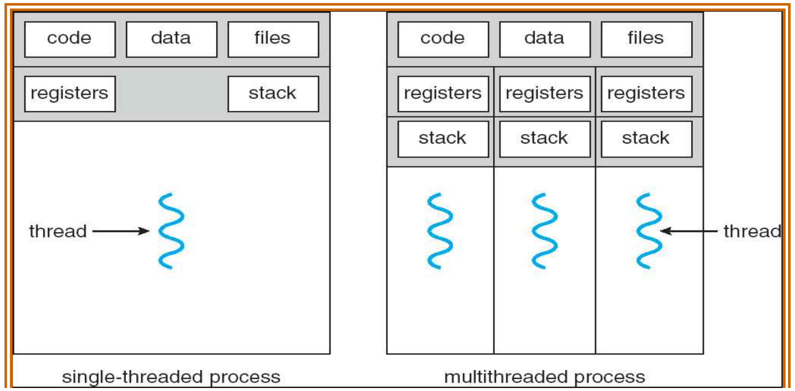
- Shared data, Race condition, Atomicity, Performance , Fine/Coarse-grained locking, …

Chap. 26 Concurrency: An Introduction

- So far
  - ✓ CPU virtualization
    - Goal: Enable multiple programs to be executed (conceptually) in parallel
    - How to: Make an illusion that we have virtual CPUs as many as the # of processes
  - ✓ Memory virtualization
    - Goal: Share physical memory among processes in an isolated manner
    - How to: Create an illusion that each process has a private, large address space (virtual memory)
- From now on
  - ✓ Multi-threaded program
    - Thread: flow of control
    - Process: one flow of control + resources (address space, files)
    - Multi-threaded program (or process): multiple flow of controls + resources (address space, files)
      - Multiple threads share address space
      - cf.) Multiple Processes do not share their address space
  - ✓ Concurrency
    - Shared data ➔ race condition ➔ may generate wrong results
    - Concurrency: enforce to access shared data in a synchronized way

26.1 Why Use Threads?

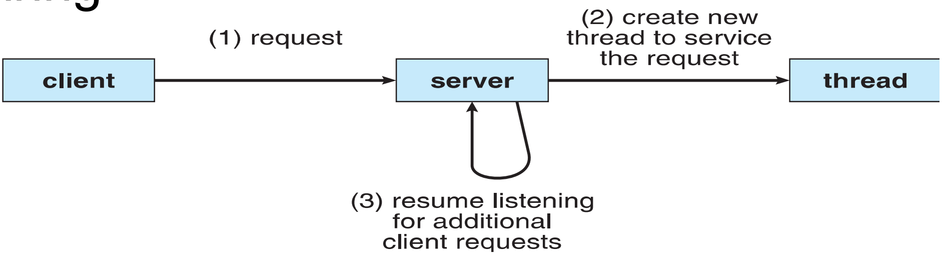
- Thread definition
  - ✓ Computing resources for a program
    - CPU: registers (context), scheduling entity
    - Address space: code, data, heap and stack
    - Files: non-volatile data and I/O devices
  - ✓ Process model
    - Use all resources exclusively
    - fork(): create all resources ➔ better isolation, worse sharing, slow creation
  - ✓ Thread model
    - Share resources among threads: code, data, heap and files
    - Exclusively resources used by a thread: CPU abstraction and stack
    - pthread_create(): create exclusive resources only ➔ fast creation, better sharing, worse isolation



(Source: A. Silberschatz, "Operating system Concept")

26.1 Why Use Threads?

- Benefit of Thread
  - ✓ Fast creation
    - Process: heavyweight, Thread: lightweight
  - ✓ Parallelism
    - Example: sort 100,000 items
      - Single thread ➔ scan all for sorting
      - Multithread: divide and conquer (Google's MapReduce Model)
  - ✓ Can overlap processing with waiting (e.g. I/O waiting)
    - Example: web server
      - Single thread: receive, processing, response
      - Multiple thread: receive thread, processing thread x n, response thread
  - ✓ Data sharing


(1) request
client
server
(2) create new thread to service the request
thread
(3) resume listening for additional client requests

☛ Make SW (e.g. browser, web server): either using multiple processes or multiple threads? (both are independent scheduling entity (can utilize CPUs), but different sharing semantics)

26.1 Why Use Threads?

- Thread management
  - ✓ Several stacks in an address space
    - Stack: called as thread local storage since each thread has its own stack
  - ✓ Scheduling entity
    - State and transition
      - Thread state: Ready, Run, Wait, … (like process)
    - Each thread has its own scheduling priority
    - Context switch at the thread level
    - ➔ TCB (Thread Control Block)
      - for thread-specific information management

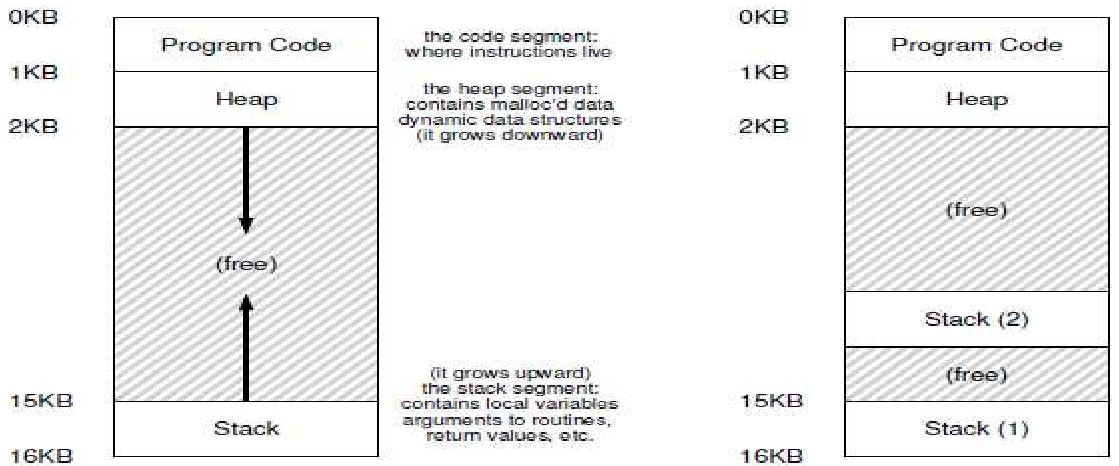
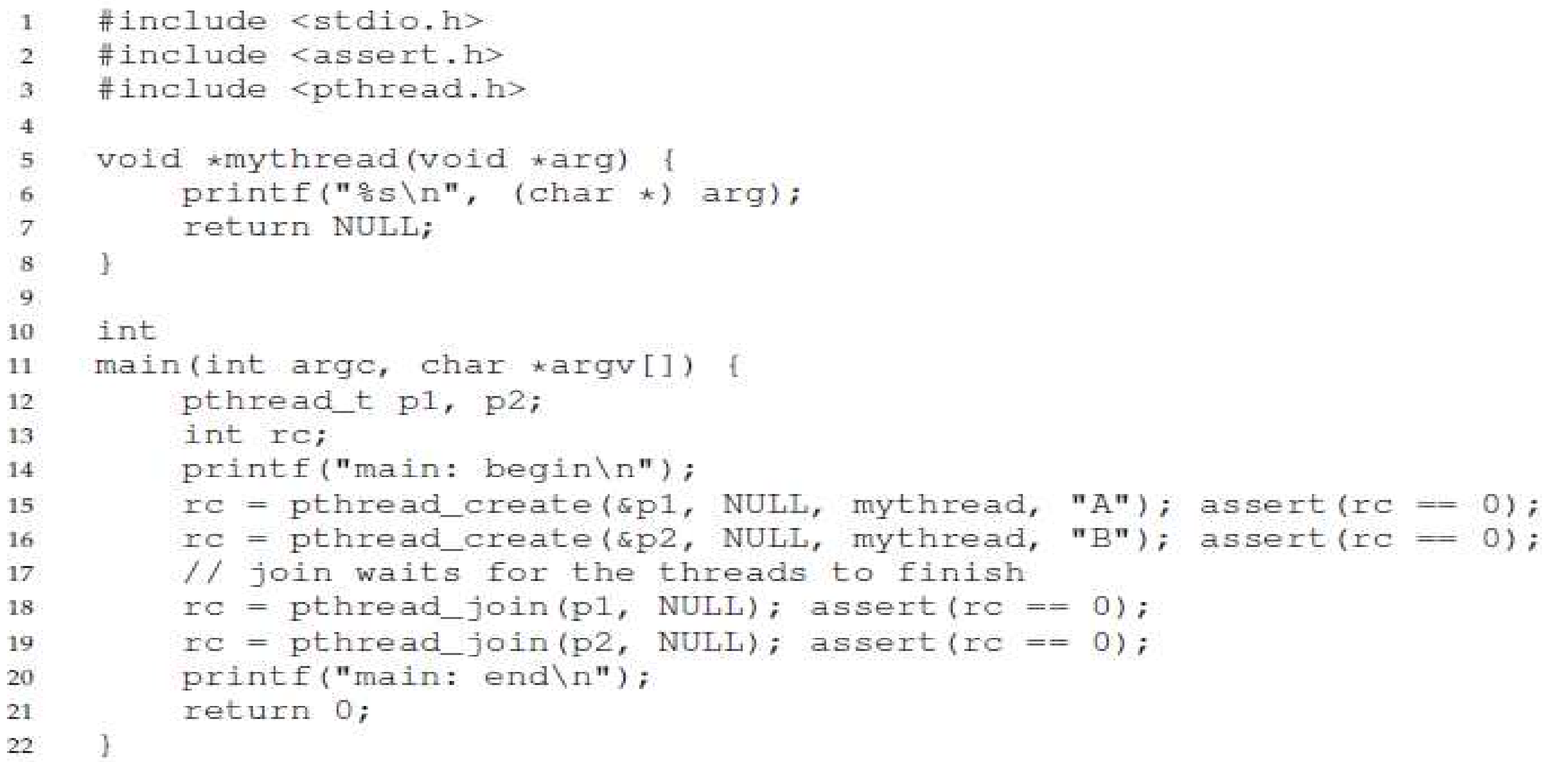


Figure 26.1: Single-Threaded And Multi-Threaded Address Spaces

26.2 An Example: Thread Creation

- Thread API
  - ✓ pthread_create(): similar to fork(), thread exits when the passed function reach the end.
  - ✓ pthread_join(): similar to wait(), for synchronization

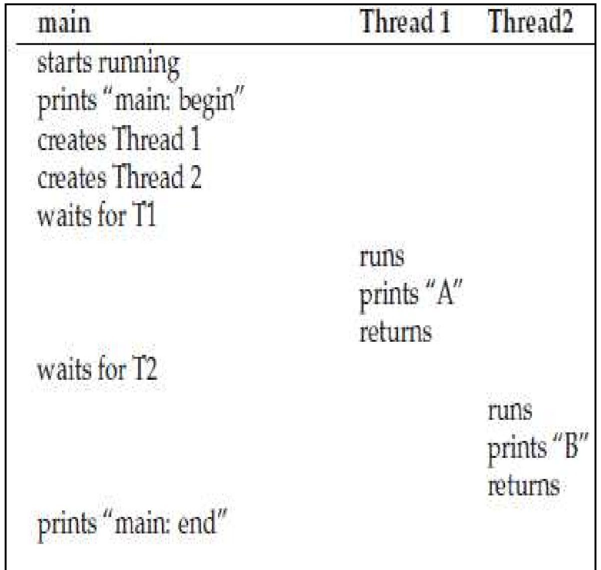


```
#include <stdio.h>
#include <assert.h>
#include <pthread.h>

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int
main(int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf("main: begin\n");
    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
    rc = pthread_join(p1, NULL); assert(rc == 0);
    rc = pthread_join(p2, NULL); assert(rc == 0);
    printf("main: end\n");
    return 0;
}
```

Figure 26.2: Simple Thread Creation Code (t0.c)

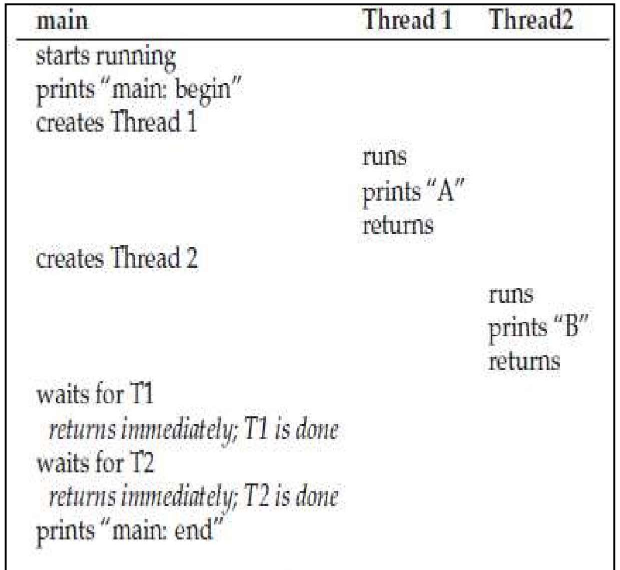
26.2 An Example: Thread Creation

- Thread trace
  - ✓ Threads: main, thread1, thread2
  - ✓ Scheduling order: depend on the whims of scheduler
    - Main ➔ create t1 ➔ create t2 ➔ wait ➔ run t1 ➔ run t2 ➔ main: Fig. 3
    - Main ➔ create t1 ➔ run t1 ➔ create t2 ➔ run t2 ➔ wait ➔ main: Fig. 4
    - Main ➔ create t1 ➔ create t2 ➔ run t2 ➔ run t1 ➔ wait ➔ main: Fig. 5



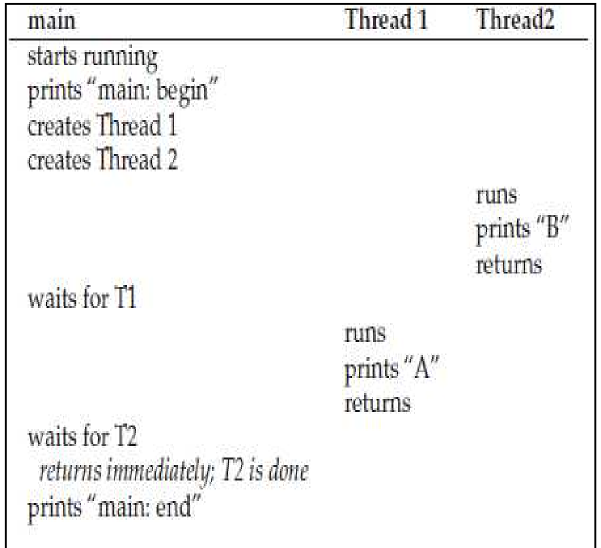
main	Thread 1	Thread2
starts running		
prints “main: begin”		
creates Thread 1		
creates Thread 2		
waits for T1		
	runs	
	prints “A”	
	returns	
waits for T2		
		runs
		prints “B”
		returns
prints “main: end”		

Figure 26.3: Thread Trace (1)



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

Figure 26.4: Thread Trace (2)

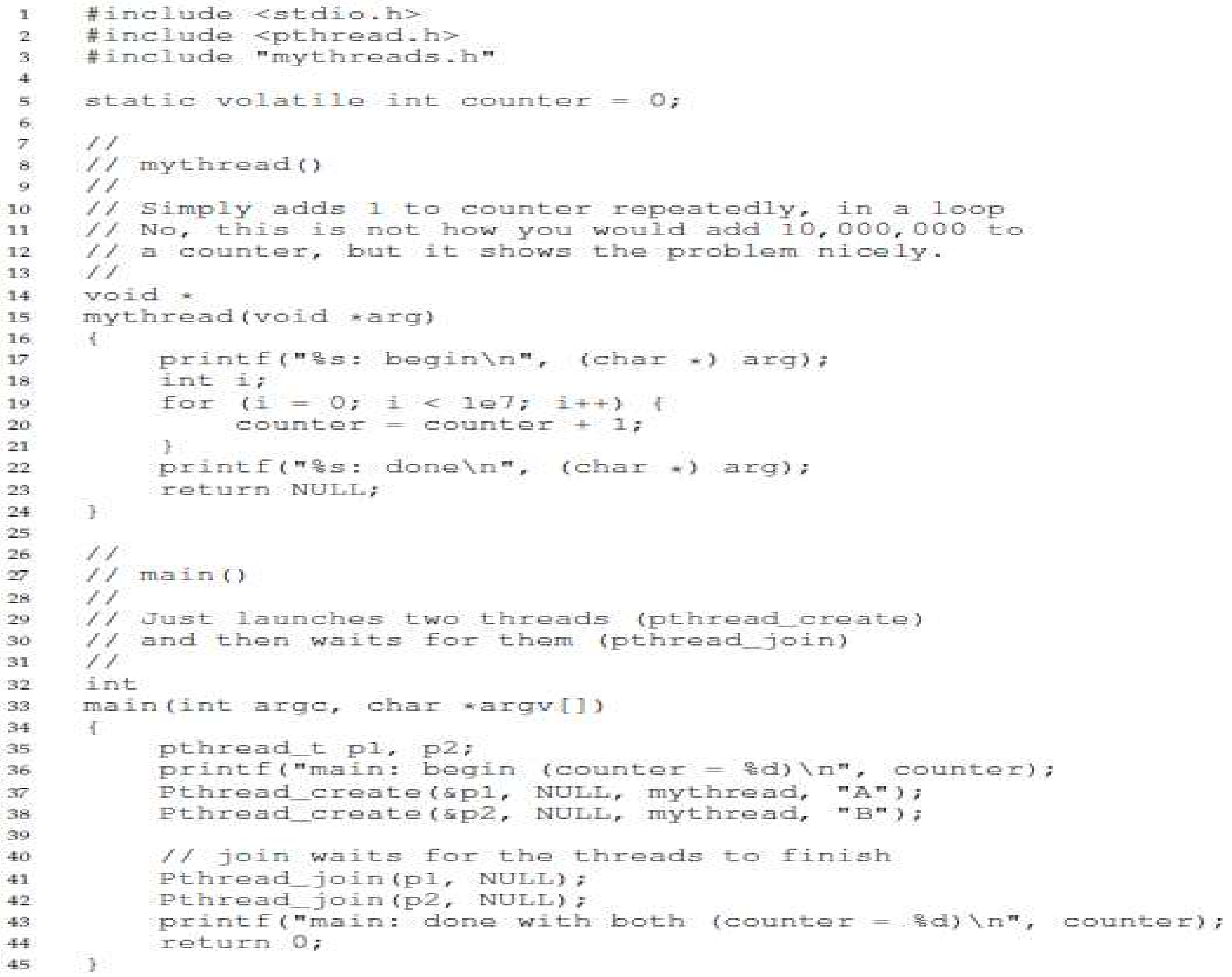


main	Thread 1	Thread2
starts running		
prints “main: begin”		
creates Thread 1		
creates Thread 2		
		runs
		prints “B”
		returns
waits for T1		
	runs	
	prints “A”	
	returns	
waits for T2		
returns immediately; T2 is done		
prints “main: end”		

Figure 26.5: Thread Trace (3)

26.3 Why It Gets Worse: Shared Data

- Shared data example (see Figure 2.5 in LN 1)

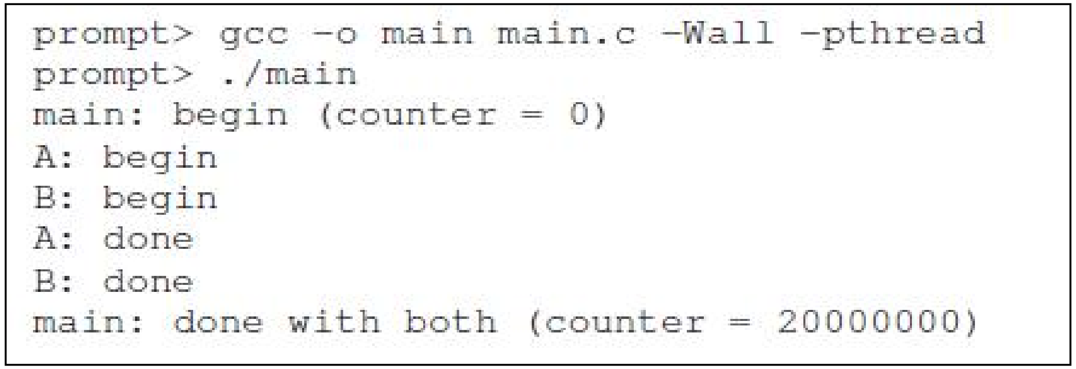


```c
#include <stdio.h>
#include <pthread.h>
#include "mythreads.h"

static volatile int counter = 0;

//
// mythread()
//
// Simply adds 1 to counter repeatedly, in a loop
// No, this is not how you would add 10,000,000 to
// a counter, but it shows the problem nicely.
//
void *
mythread(void *arg)
{
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}

//
// main()
//
// Just launches two threads (pthread_create)
// and then waits for them (pthread_join)
//
int
main(int argc, char *argv[])
{
    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", counter);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");

    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done with both (counter = %d)\n", counter);
    return 0;
}
```

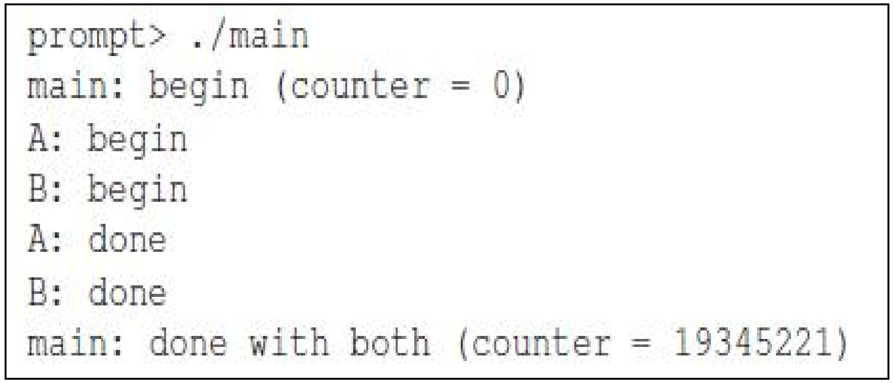
Figure 26.6: Sharing Data: Uh Oh (t1.c)

26.3 Why It Gets Worse: Shared Data

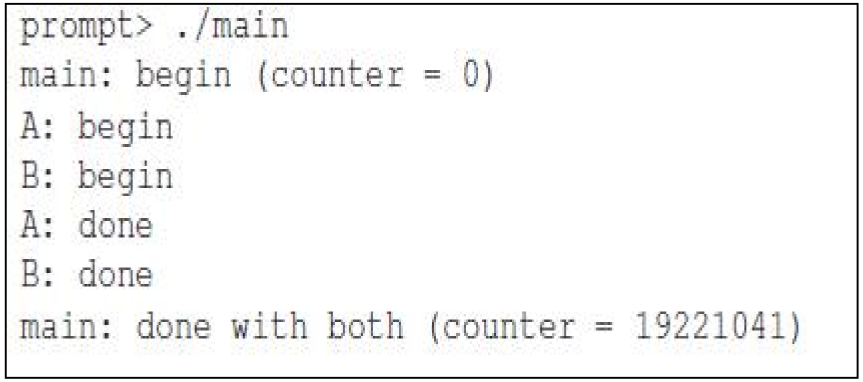
- Results of the shared data example



```
prompt> gcc -o main main.c -Wall -pthread
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 20000000)
```



```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19345221)
```

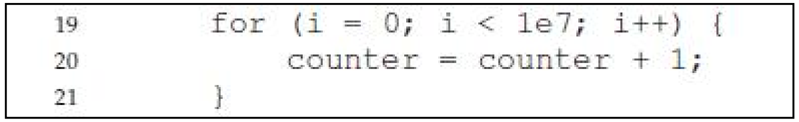


```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19221041)
```

- ✓ Different results (not deterministic)
- ✓ Big question? Why does this happen?

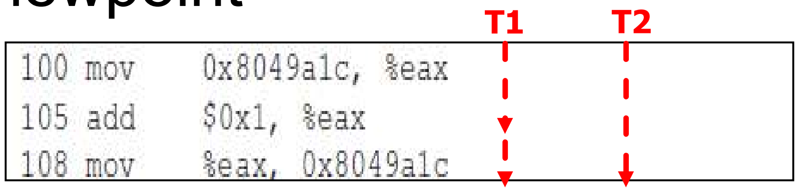
26.4 The Heart of Problem: Uncontrolled Scheduling

- High level viewpoint

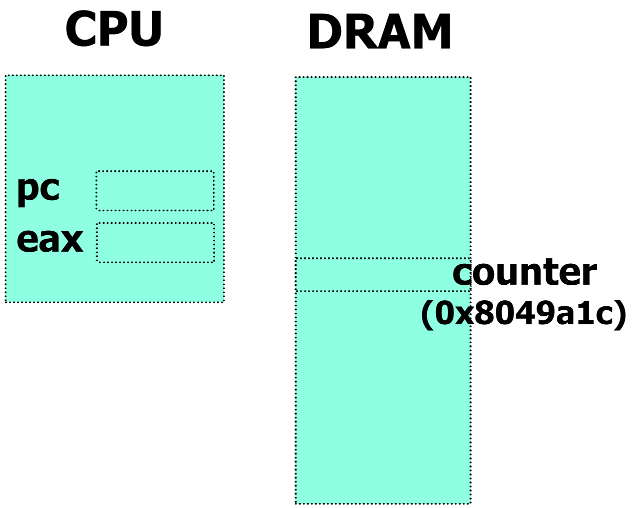


```
        for (i = 0; i < 1e7; i++) {
            counter = counter + 1;
        }
```

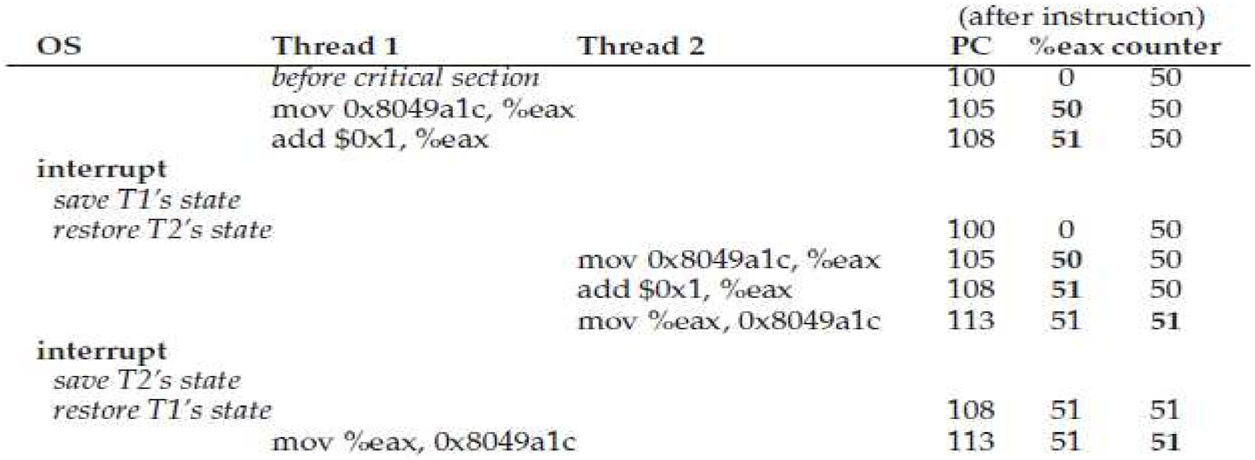
- CPU level viewpoint



```
100 mov     0x8049a1c, %eax
105 add     $0x1, %eax
108 mov     %eax, 0x8049a1c
```



- Scheduling viewpoint



OS	Thread 1	Thread 2	(after instruction) 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

- ☛ The counter value increases only 1, even though two additions are performed
- ☛ Some students show their capability while using pthread_mutex_lock() in the 1st homework.

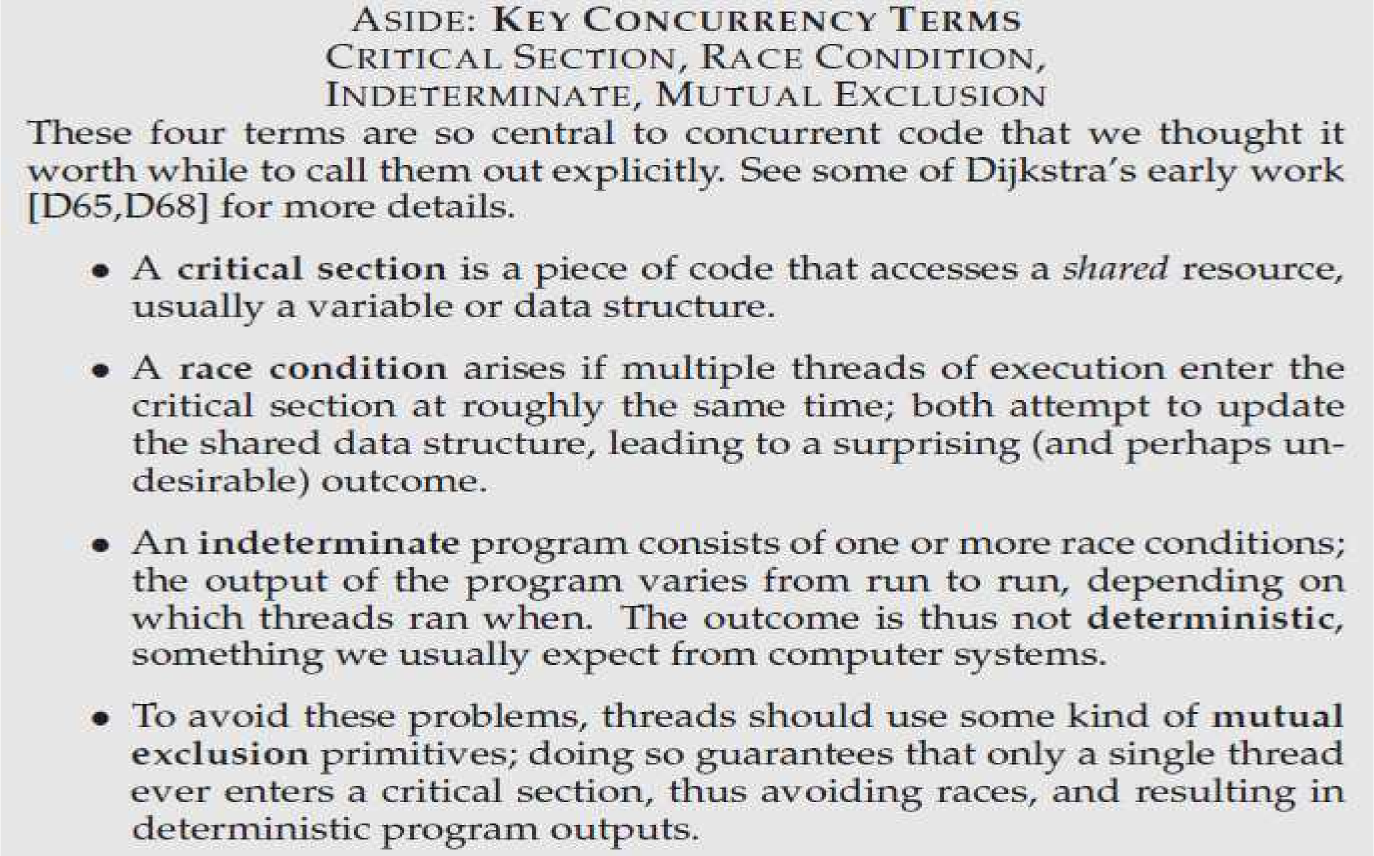
26.4 The Heart of Problem: Uncontrolled Scheduling

- Reason
  - ✓ Two threads access shared data at the same time ➔ race condition
  - ✓ Uncontrolled scheduling ➔ Results are different at each execution depending on scheduling order
- Solution
  - ✓ Controlled scheduling: Do all or nothing (indivisible) ➔ atomicity
  - ✓ The code that can result in the race condition ➔ critical section
    - Code ranging from 100 to 108 in the example of the previous slide
  - ✓ Allow only one thread in the critical section ➔ mutual exclusion

(One-lane Tunnel to Milford Sound in New Zealand)

26.6 One More Problem: Waiting for Another

- Two issues related to concurrency
  - ✓ Mutual exclusion: only one thread can enter a critical section
  - ✓ Synchronization: one thread must wait for another to complete some action before it continues



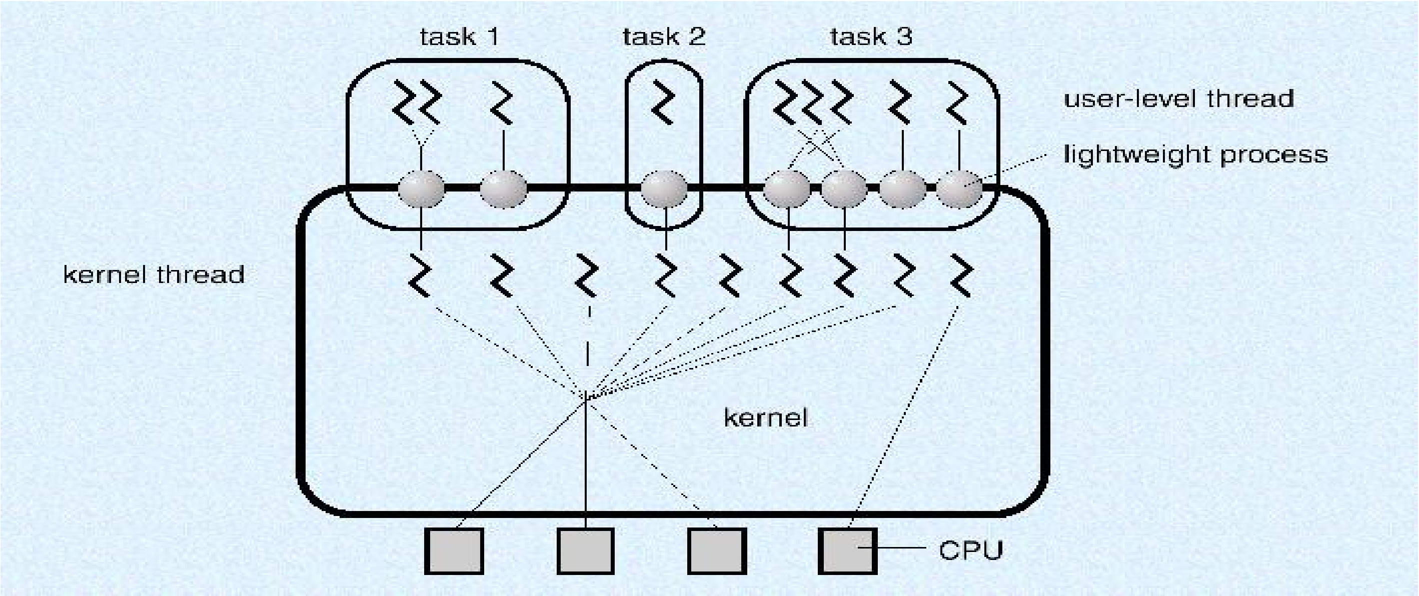
ASIDE: KEY CONCURRENCY TERMS
CRITICAL SECTION, RACE CONDITION,
INDETERMINATE, MUTUAL EXCLUSION

These four terms are so central to concurrent code that we thought it worth while to call them out explicitly. See some of Dijkstra's early work [D65,D68] for more details.

- A **critical section** is a piece of code that accesses a *shared* resource, usually a variable or data structure.
- A **race condition** arises if multiple threads of execution enter the critical section at roughly the same time; both attempt to update the shared data structure, leading to a surprising (and perhaps undesirable) outcome.
- An **indeterminate** program consists of one or more race conditions; the output of the program varies from run to run, depending on which threads ran when. The outcome is thus not **deterministic**, something we usually expect from computer systems.
- To avoid these problems, threads should use some kind of **mutual exclusion** primitives; doing so guarantees that only a single thread ever enters a critical section, thus avoiding races, and resulting in deterministic program outputs.

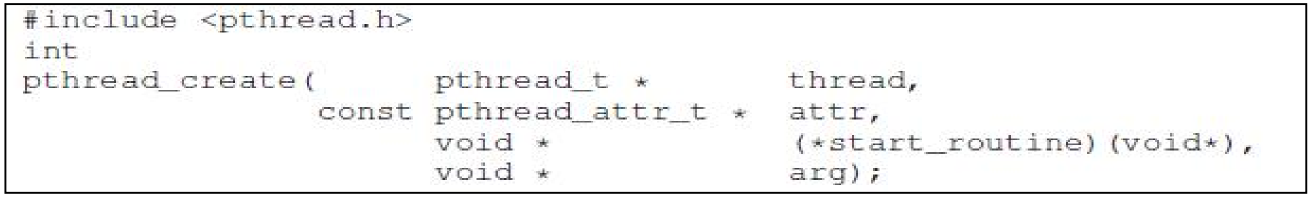
Chap. 27 Interlude: Thread API

- Thread classification
  - ✓ User-level thread
    - Thread managements are done by user-level threads library including user-level scheduler
  - ✓ Kernel-level thread
    - Thread managements are supported by the Kernel (Most operating systems)
  - ✓ Three representative libraries: pthread, Windows thread, Java thread
    - In this class, we focus on the pthread in Linux which is implemented using clone() system call with sharing options (pthread based on kernel thread)



27.1 Thread Creation

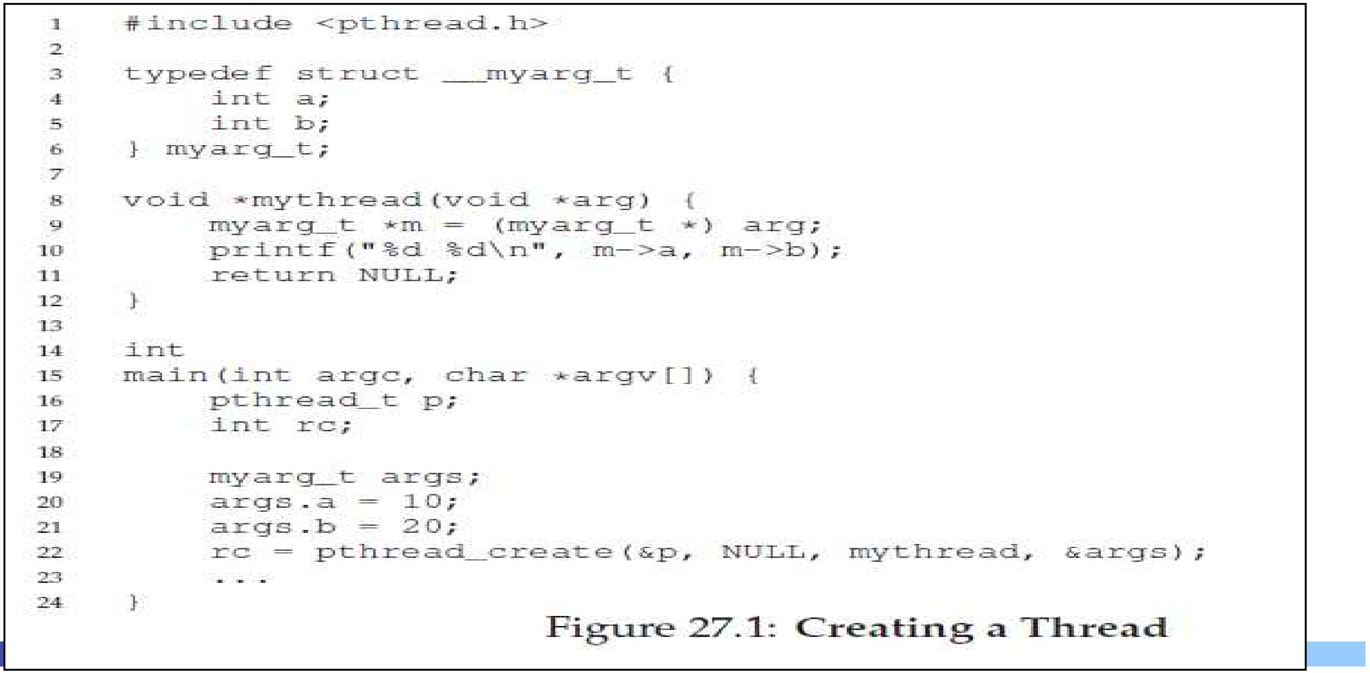
- Thread creation API



```
#include <pthread.h>
int
pthread_create(      pthread_t *         thread,
               const pthread_attr_t *    attr,
                     void *              (*start_routine)(void*),
                     void *              arg);
```

  - ✓ Arguments: 1) thread structure to interact with this thread, 2) attribute of the thread such as priority and stack size, in most case it is NULL (use default), 3) function pointer for start routine, 4) arguments

- Example

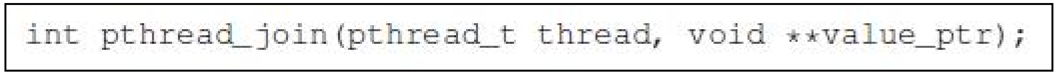


```
#include <pthread.h>

typedef struct __myarg_t {
    int a;
    int b;
} myarg_t;

void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf("%d %d\n", m->a, m->b);
    return NULL;
}

int
main(int argc, char *argv[]) {
    pthread_t p;
    int rc;

    myarg_t args;
    args.a = 10;
    args.b = 20;
    rc = pthread_create(&p, NULL, mythread, &args);
    ...
}
```

Figure 27.1: Creating a Thread

27.2 Thread Completion

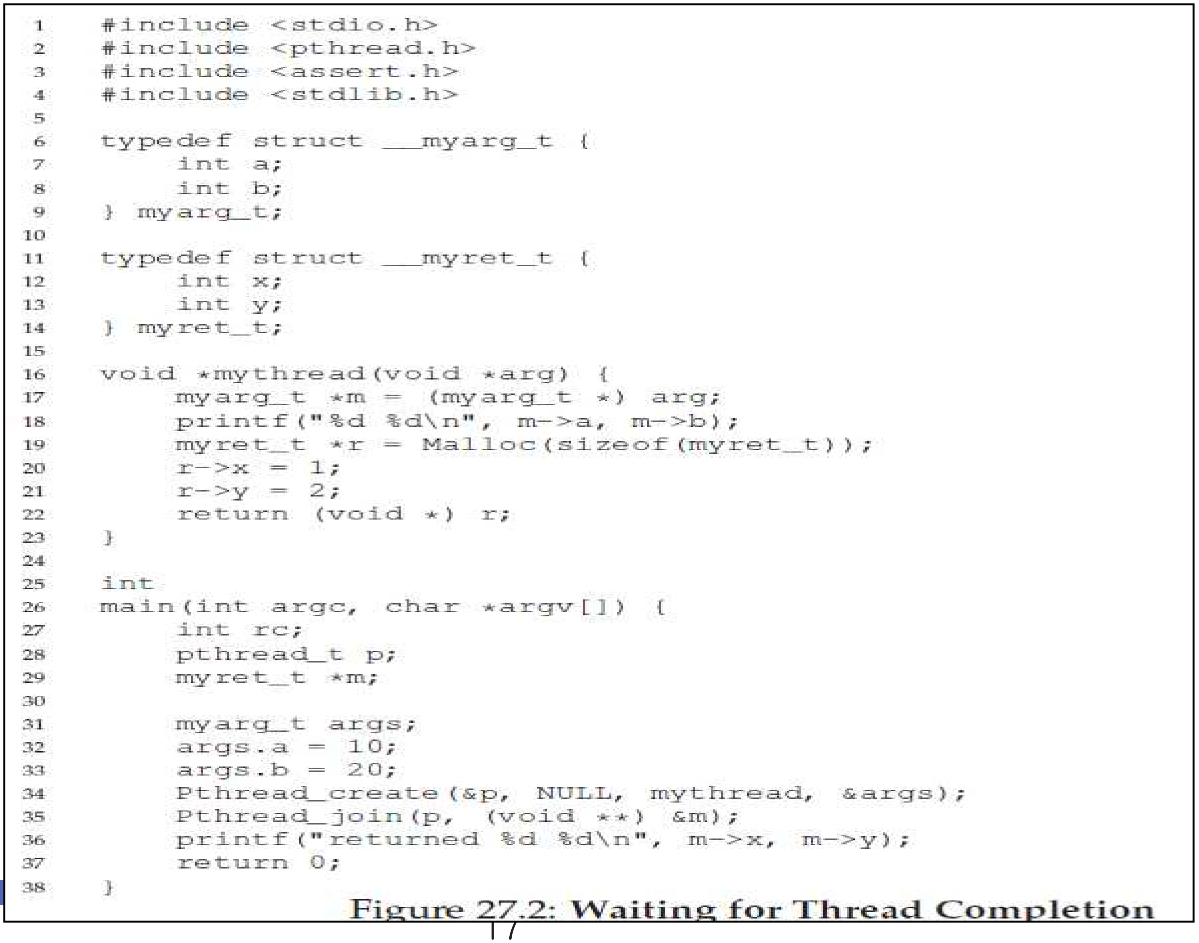
- Wait for completion



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

  - ✓ Arguments: 1) thread structure, which is initialized by the thread creation routine, 2) a pointer to the return value (NULL means “don’t care”)

- Example



```
#include <stdio.h>
#include <pthread.h>
#include <assert.h>
#include <stdlib.h>

typedef struct __myarg_t {
    int a;
    int b;
} myarg_t;

typedef struct __myret_t {
    int x;
    int y;
} myret_t;

void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf("%d %d\n", m->a, m->b);
    myret_t *r = Malloc(sizeof(myret_t));
    r->x = 1;
    r->y = 2;
    return (void *) r;
}

int
main(int argc, char *argv[]) {
    int rc;
    pthread_t p;
    myret_t *m;

    myarg_t args;
    args.a = 10;
    args.b = 20;
    Pthread_create(&p, NULL, mythread, &args);
    Pthread_join(p, (void **) &m);
    printf("returned %d %d\n", m->x, m->y);
    return 0;
}
```

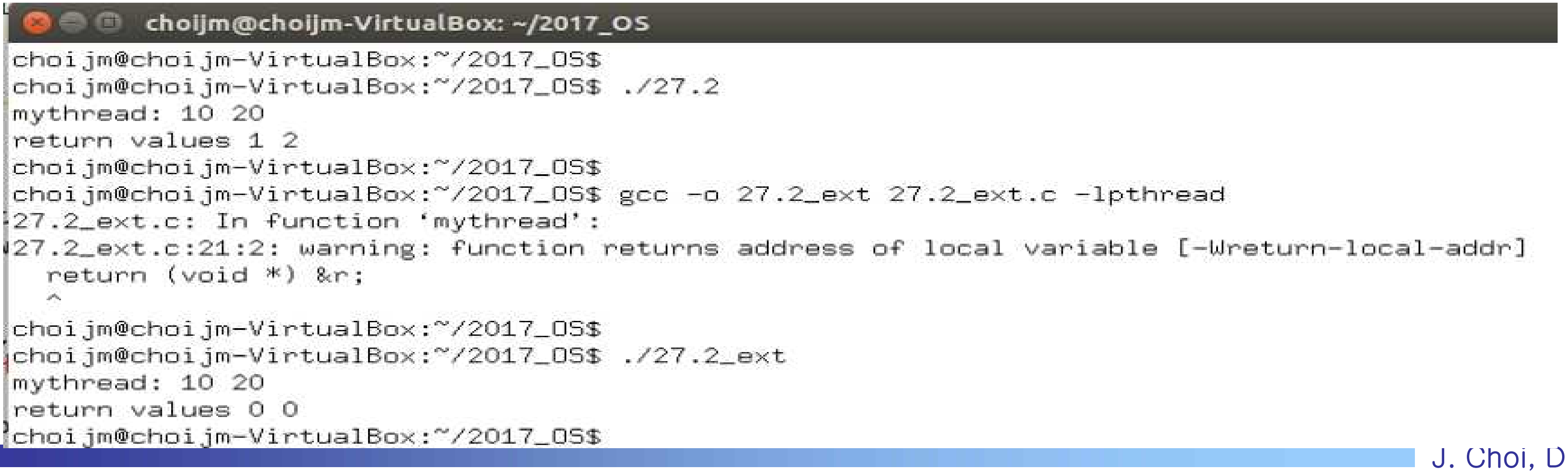
Figure 27.2: Waiting for Thread Completion

27.2 Thread Completion

- Be careful: do not return a pointer allocated on the stack
  - ✓ Modified version of Figure 27.2

```
void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf("%d %d\n", m->a, m->b);
    myret_t r; // ALLOCATED ON STACK: BAD!
    r.x = 1;
    r.y = 2;
    return (void *) &r;
}
```

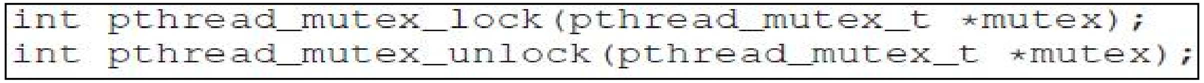
  - Variable r is allocated on the stack of mythread
  - Note that when a thread returns, stack is automatically deallocated



```
choijm@choijm-VirtualBox:~/2017_OS$
choijm@choijm-VirtualBox:~/2017_OS$ ./27.2
mythread: 10 20
return values 1 2
choijm@choijm-VirtualBox:~/2017_OS$
choijm@choijm-VirtualBox:~/2017_OS$ gcc -o 27.2_ext 27.2_ext.c -lpthread
27.2_ext.c: In function ‘mythread’:
27.2_ext.c:21:2: warning: function returns address of local variable [-Wreturn-local-addr]
  return (void *) &r;
  ^
choijm@choijm-VirtualBox:~/2017_OS$
choijm@choijm-VirtualBox:~/2017_OS$ ./27.2_ext
mythread: 10 20
return values 0 0
choijm@choijm-VirtualBox:~/2017_OS$
```

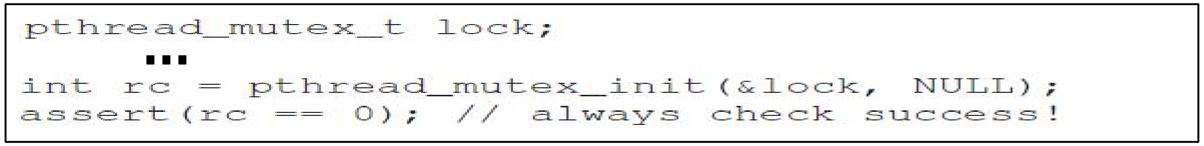
27.3 Locks / 27.4 Condition Variables

- Concurrency mechanisms
  - ✓ Mutual exclusion API (mutex_***): for mutual exclusion
    - API



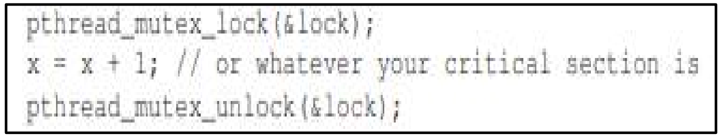
```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

    - Example



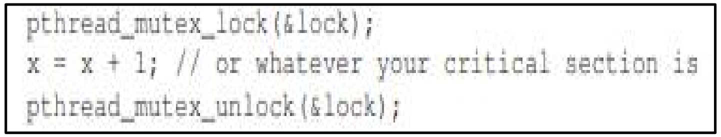
```
pthread_mutex_t lock;
    ...
int rc = pthread_mutex_init(&lock, NULL);
assert(rc == 0); // always check success!
```

Thread 1



```
pthread_mutex_lock(&lock);
x = x + 1; // or whatever your critical section is
pthread_mutex_unlock(&lock);
```

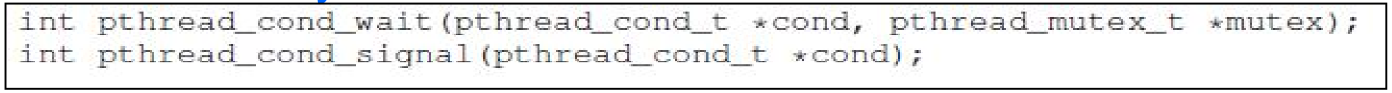
Thread 2



```
pthread_mutex_lock(&lock);
x = x + 1; // or whatever your critical section is
pthread_mutex_unlock(&lock);
```

- 1) Lock free ➔ entering CS. 2) Lock already hold ➔ not return from the call

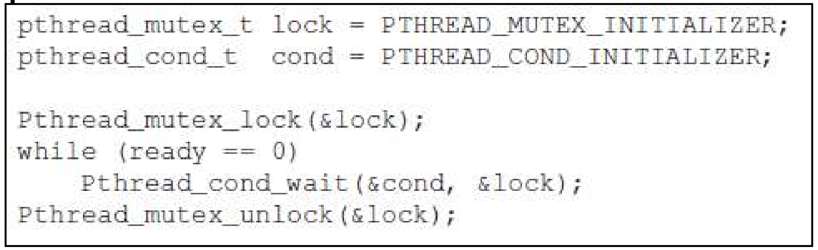
  - ✓ Condition Variables: for synchronization
    - API



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

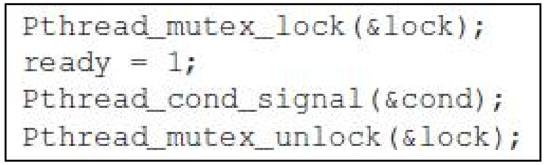
    - Example

Thread 1



```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cond = PTHREAD_COND_INITIALIZER;

Pthread_mutex_lock(&lock);
while (ready == 0)
    Pthread_cond_wait(&cond, &lock);
Pthread_mutex_unlock(&lock);
```

Thread 2

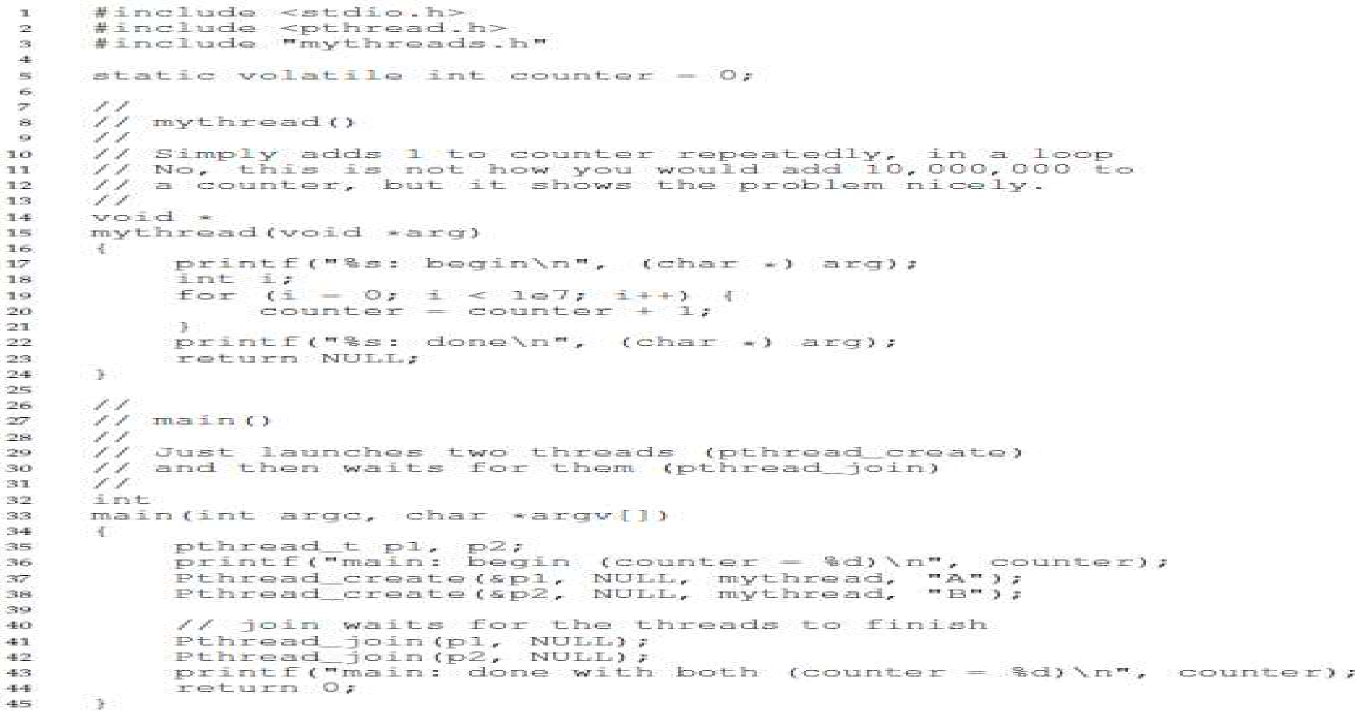


```
Pthread_mutex_lock(&lock);
ready = 1;
Pthread_cond_signal(&cond);
Pthread_mutex_unlock(&lock);
```

- Guarantee that some part (e.g. initialization) will execute before others (service)

Quiz for 5th-Week 1st-Lesson

- Quiz
  - ✓ 1. Discuss the differences between fork() and pthread_create() using the term of computing resources (CPU, address space, file).
  - ✓ 2. Explain the "shared resource“, “race condition“, “atomicity”, “critical section” and “mutual exclusion“ using the program in Figure 26.6.
  - ✓ Due: until 6 PM Friday of this week (2nd, April)



```c
#include <stdio.h>
#include <pthread.h>
#include "mythreads.h"

static volatile int counter = 0;

//
// mythread()
//
// Simply adds 1 to counter repeatedly, in a loop
// No, this is not how you would add 10,000,000 to
// a counter, but it shows the problem nicely.
//
void *
mythread(void *arg)
{
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}

//
// main()
//
// Just launches two threads (pthread_create)
// and then waits for them (pthread_join)
//
int
main(int argc, char *argv[])
{
    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", counter);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");

    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done with both (counter = %d)\n", counter);
    return 0;
}
```

Figure 26.6: Sharing Data: Uh Oh (t1.c)

Chap. 28 Locks

- Locks
  - ✓ Basic idea
  - ✓ How to Evaluate?
- Realization
  - ✓ 1) Controlling interrupt
  - ✓ 2) SW approach
  - ✓ 3) HW approach: using atomic operations
    - Test-and-Set, Compare-and-Swap, Load-Linked and Store-Conditional, Fetch-and-Add, …
- Building and Evaluating spin locks
- Sleeping instead of Spinning: using Queues
- Different OS, Different Support

28.1 Locks: Basic Idea / 28.2 Pthread Locks

- Critical section example

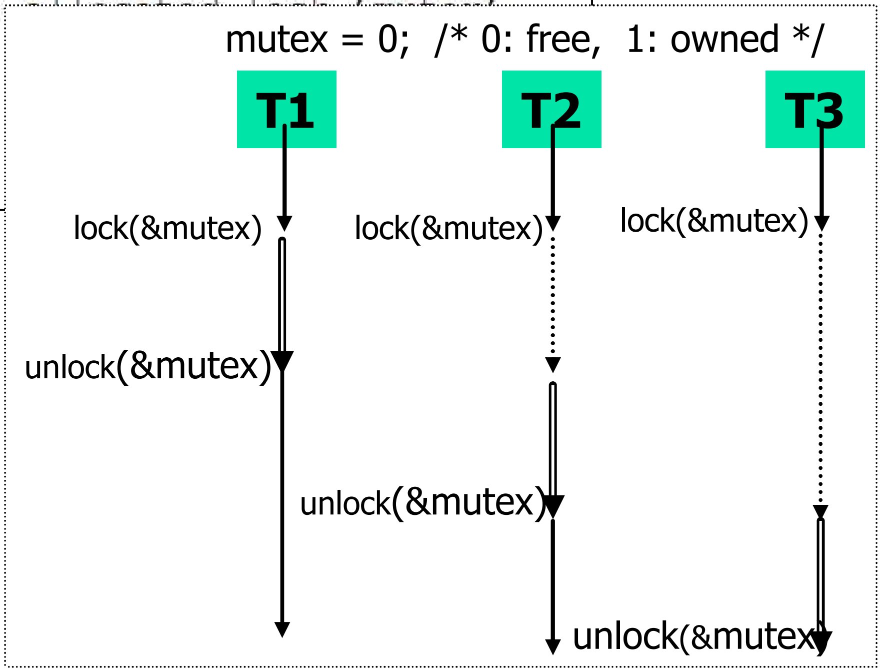
```
balance = balance + 1;
```

  - ✓ Other critical sections are possible such as adding a node to a linked list, hash update or more complex updates to shared data structures

- Mutual exclusion using lock
  - ✓ Using lock/unlock before/after critical section (generic description)

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

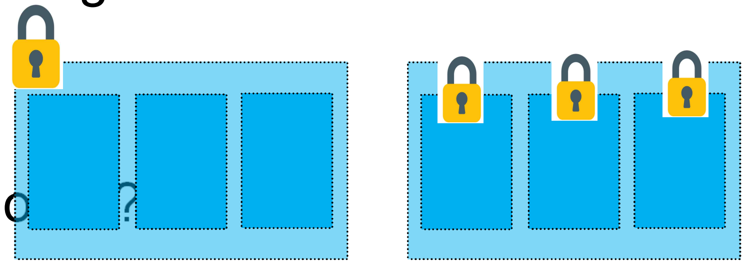
    - Guarantee that only one thread
  - ✓ Real example: pthread
    - pthread_mutex_lock()
    - pthread_mutex_unlock()



mutex = 0; /* 0: free, 1: owned */

28.3 Building A Lock / 28.4 Evaluating Locks

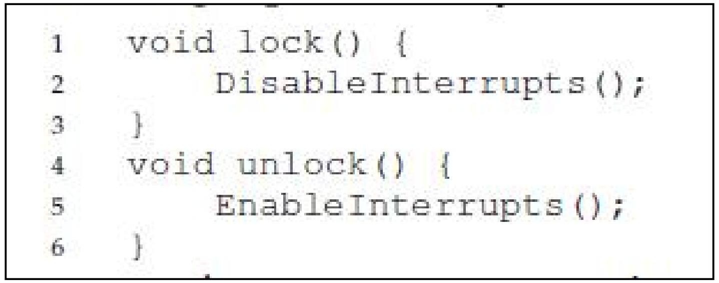
- How to build the lock()/unlock() APIs?
  - ✓ Collaboration between HW and OS supports
- How to evaluate a lock()/unlock()? ➔ Three properties
  - ✓ 1) Correctness: Does it guarantee mutual exclusion?
  - ✓ 2) Fairness: Does any thread starve (or being treated unfairly)?
  - ✓ 3) Performance: Overhead added by using the lock



- One issue: lock size
  - ✓ Three shared variables ➔ how many lo ?
  - ✓ Coarse-grained lock
    - Prefer to big critical section with smaller number of locks (e.g. one)
    - Pros) simple, Cons) parallelism
  - ✓ Fine-grained lock
    - Prefer to small critical section with larger number of locks (e.g. three)
    - Pros) parallelism, Cons) simple

28.5 Controlling Interrupts

- How to build the lock()/unlock() APIs?
  - ✓ First solution: Disable interrupt
    - No interrupt ➔ No context switch ➔ No intervention in critical section

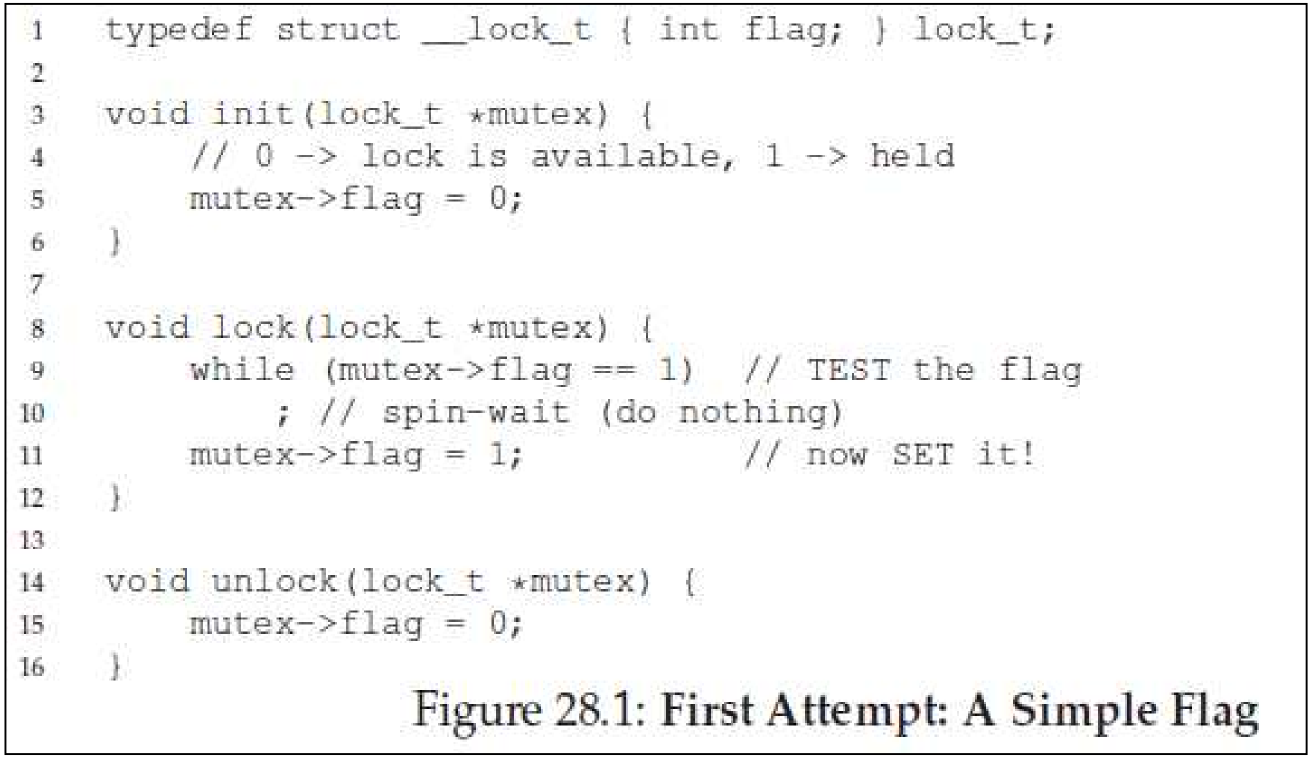


```
void lock() {
    DisableInterrupts();
}
void unlock() {
    EnableInterrupts();
}
```

- Pros)
  - Simplicity (earliest used solution)
- Cons)
  - Disable interrupt for a long period ➔ might lead to lost interrupt
  - Abuse or misuse ➔ monopolize, endless loop (no handling mechanism only reboot)
  - Work only on a single processor (Not work on multiprocessors) ➔ Can tackle the race condition due to the context switch, not due to the concurrent execution
  - ➔ used inside the OS (or trusty world)

28.6 Test-and-Set (Atomic Exchange)

- How to build the lock()/unlock() APIs?
  - ✓ Second solution: SW-only approach



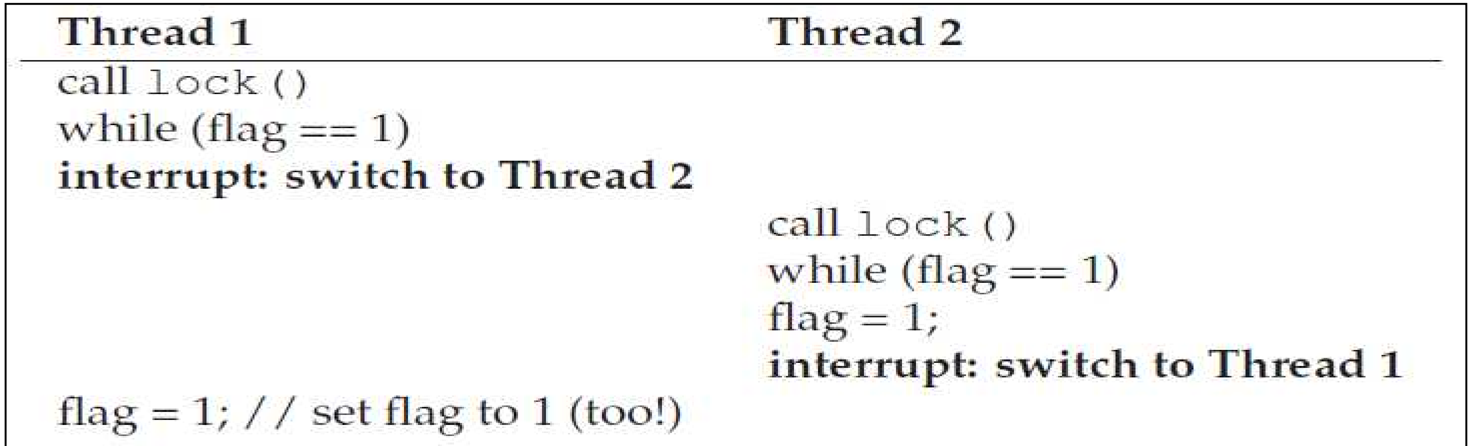
```
typedef struct __lock_t { int flag; } lock_t;

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

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

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

Figure 28.1: First Attempt: A Simple Flag

☛ Is it correct?

28.6 Test-and-Set (Atomic Exchange)

- How to build the lock()/unlock() APIs?
  - ✓ Problems of the SW-only approach
    - Correctness: fail to provide the mutual exclusion
      - Both thread can enter the critical section
      - Test and Set are done separately (not indivisible)



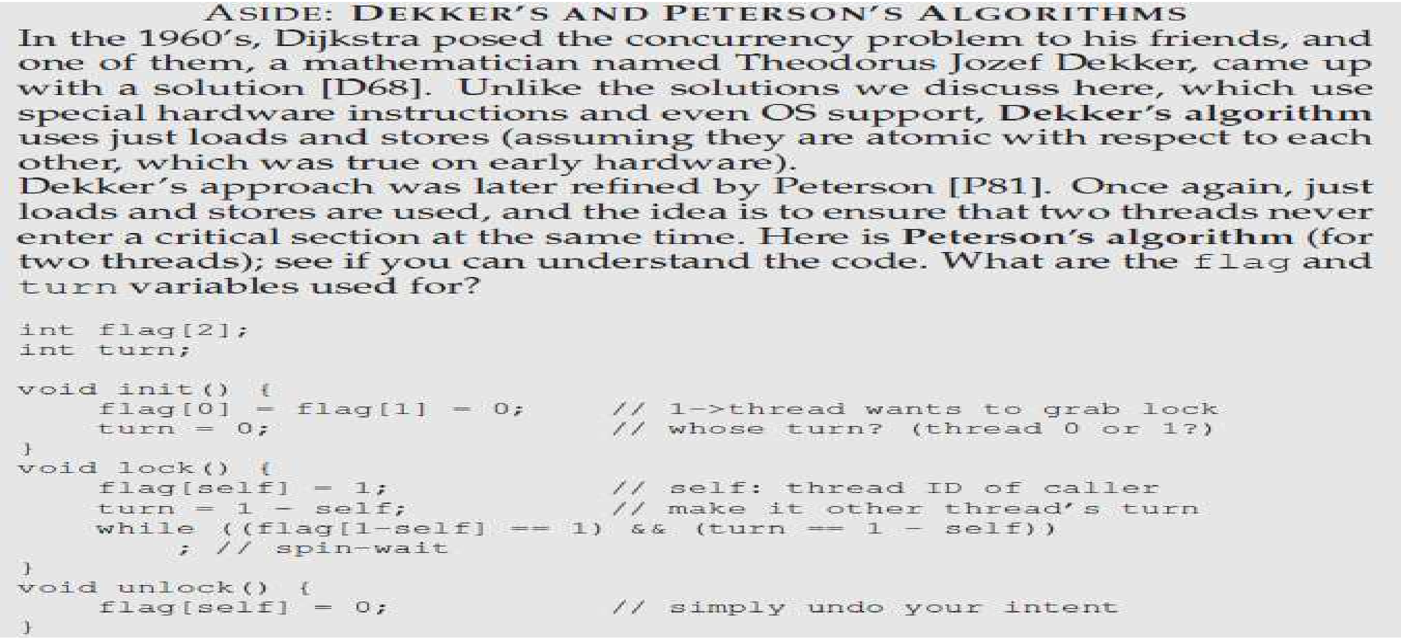
Thread 1	Thread 2
call `lock()`	
while (flag == 1)	
interrupt: switch to Thread 2	
	call `lock()`
	while (flag == 1)
	flag = 1;
	interrupt: switch to Thread 1
flag = 1; // set flag to 1 (too!)	

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

- Performance
  - Spinning (Spin-waiting): endlessly check the value of flag
  - CPU is busy, but doing useless work

28.6 Test-and-Set (Atomic Exchange)

- How to build the lock()/unlock() APIs?
  - ✓ There are many SW-only approaches: Such as Dekker's algorithm, Peterson's algorithm, …



ASIDE: DEKKER'S AND PETERSON'S ALGORITHMS

In the 1960's, Dijkstra posed the concurrency problem to his friends, and one of them, a mathematician named Theodorus Jozef Dekker, came up with a solution [D68]. Unlike the solutions we discuss here, which use special hardware instructions and even OS support, **Dekker's algorithm** uses just loads and stores (assuming they are atomic with respect to each other, which was true on early hardware).

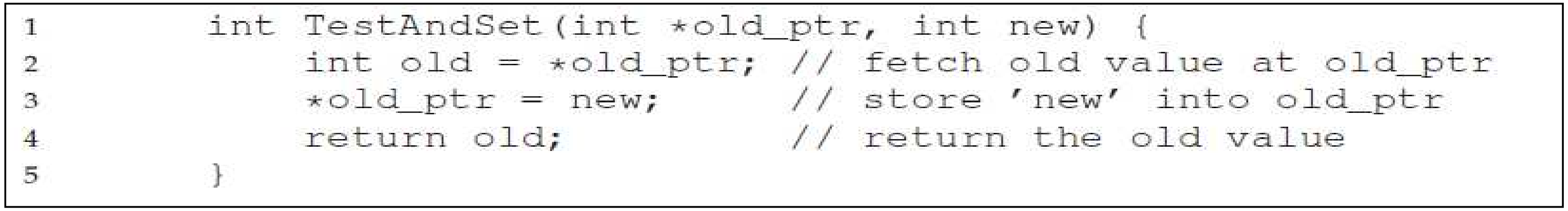
Dekker's approach was later refined by Peterson [P81]. Once again, just loads and stores are used, and the idea is to ensure that two threads never enter a critical section at the same time. Here is **Peterson's algorithm** (for two threads); see if you can understand the code. What are the `flag` and `turn` variables used for?

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

void init() {
    flag[0] = flag[1] = 0;       // 1->thread wants to grab lock
    turn = 0;                    // whose turn? (thread 0 or 1?)
}
void lock() {
    flag[self] = 1;              // self: thread ID of caller
    turn = 1 - self;             // make it other thread's turn
    while ((flag[1-self] == 1) && (turn == 1 - self))
        ; // spin-wait
}
void unlock() {
    flag[self] = 0;              // simply undo your intent
}
```

  - Pros) SW solution
  - Cons) 1) not easy to understand, 2) Inefficient (a little HW support can provide the same capability efficiently), 3) incorrect in modern systems that use the relaxed memory consistency model ➔ not used any more

28.7 Building A Working Spin Lock

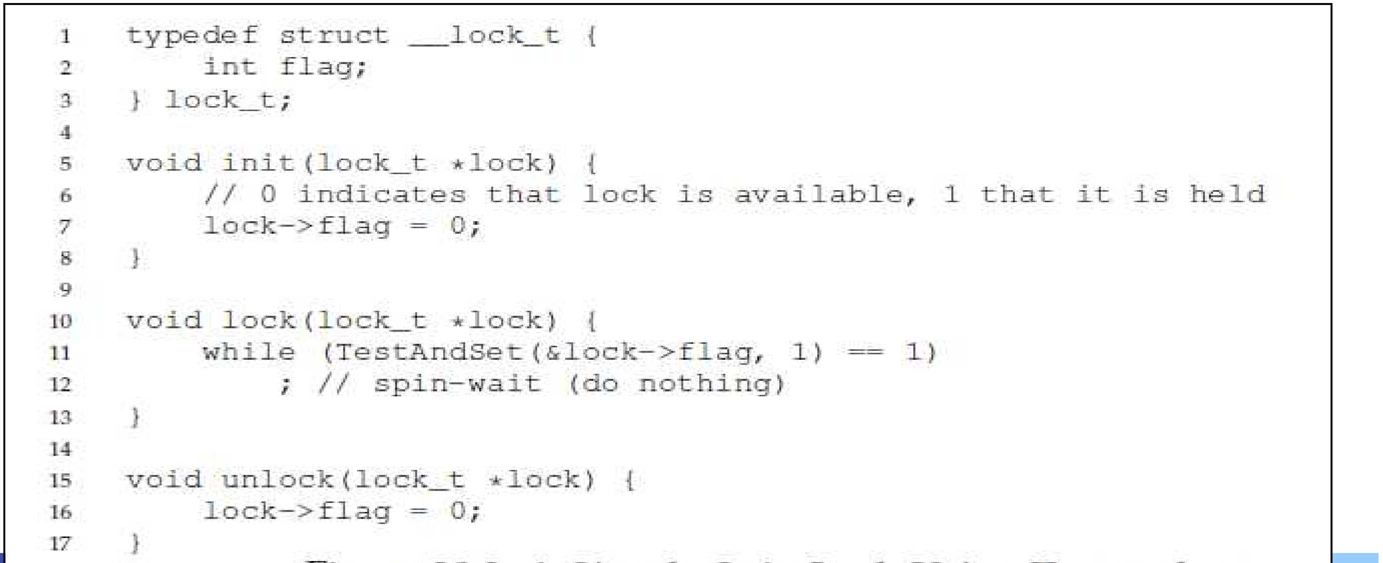
- How to build the lock()/unlock() APIs?
  - ✓ Third solution: Using HW atomic operations
    - Test-and-Set instruction (a.k.a atomic exchange) in this section



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

- All (both test the old value and set a new value) are performed atomically
- Instruction in real systems: xchg in Intel, ldstub in SPARC

- ✓ Implement lock using the Test-and-Set instruction



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

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

void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
        ; // spin-wait (do nothing)
}

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

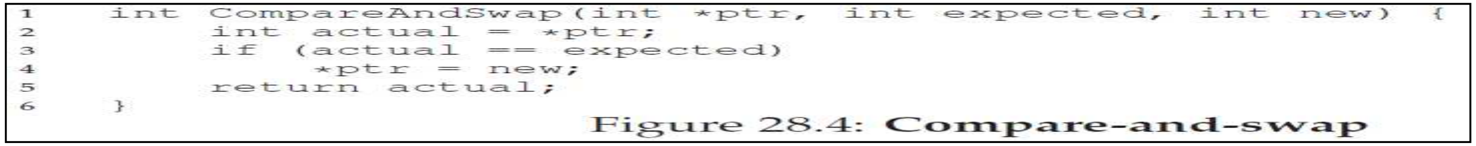
Figure 28.3: A Simple Spin Lock Using Test-and-set

28.8 Evaluating Spin Locks

- How to build the lock()/unlock() APIs?
  - ✓ Third solution: Using HW atomic operations
  - ✓ Evaluating of the Third solution
    - Correctness
      - Does it provide mutual exclusion? ➔ yes
      - Guarantee that only one thread enters the critical section
    - Fairness
      - Can it guarantee that a waiting thread will enter the critical section? ➔ unfortunately no.
      - E.g.) 10 higher priority threads and one low priority thread ➔ the latter one may spin forever, leading to starvation
    - Performance
      - In the single CPU case: Overhead can be quite painful: waste CPU cycles (until a context switch occurs!)
      - In the multiple CPUs case
        - Spin locks work relatively well when the critical section is short ➔ do not waste another CPU cycles that much
        - Usually spin lock is employed for the short critical section situation

28.9/10 Compare&Swap/Load-Linked&Store-Conditional

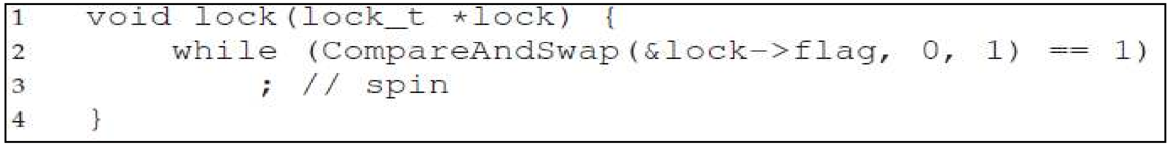
- Another atomic operation (example of the third solution)
  - ✓ Compare-and-Swap instruction
    - Compare the value specified by *ptr* with the expected one. If matched, set the new value. Then, return the previous value. All are done atomically



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

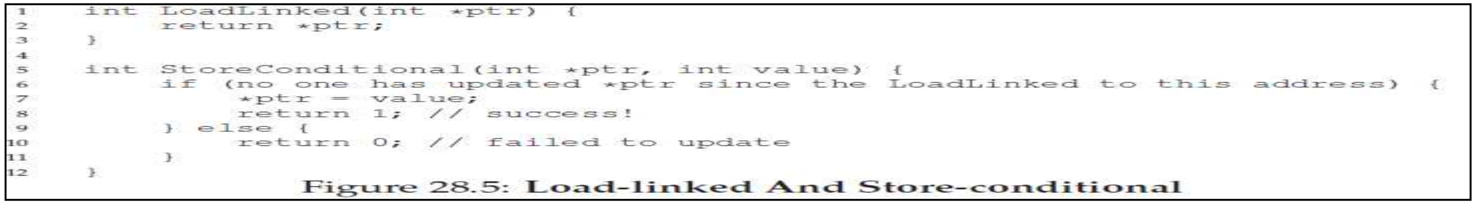
Figure 28.4: Compare-and-swap

    - How to use Lock



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

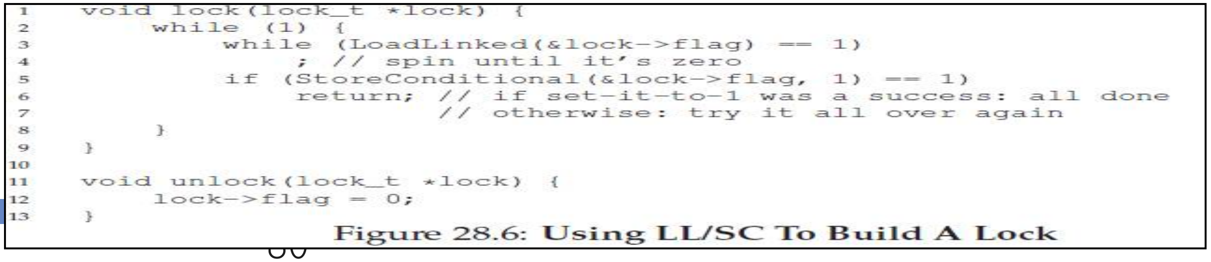
  - ✓ Load-Linked and Store-Conditional supported by MIPS, ARM, …
    - Prevent between load and store.



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

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

Figure 28.5: Load-linked And Store-conditional

    - How to use Lock

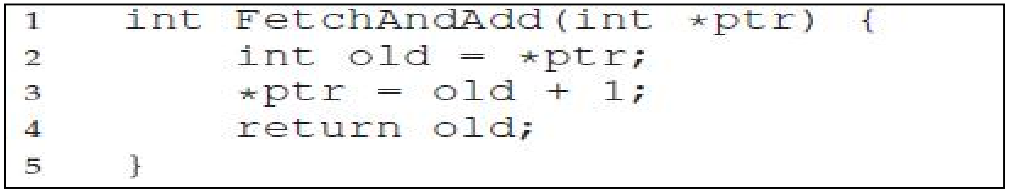


```
void lock(lock_t *lock) {
    while (1) {
        while (LoadLinked(&lock->flag) == 1)
            ; // spin until it's zero
        if (StoreConditional(&lock->flag, 1) == 1)
            return; // if set-it-to-1 was a success: all done
                    // otherwise: try it all over again
  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

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

Figure 28.6: Using LL/SC To Build A Lock

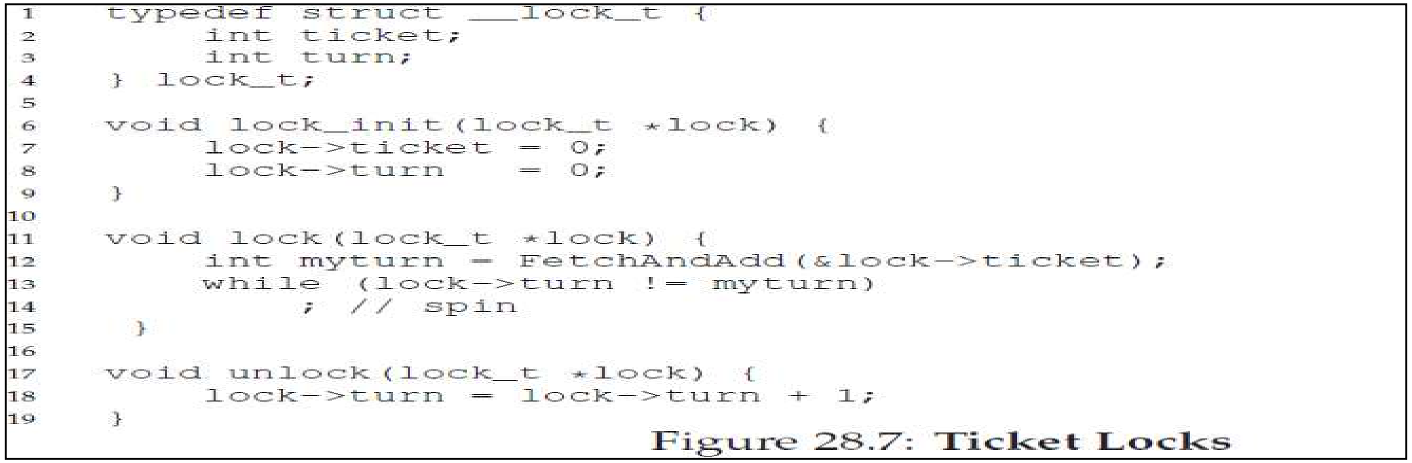
28.11 Fetch-and-Add

- Final atomic operation
  - ✓ Fetch-and-Add



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

    - Atomically increment a value while returning the old value
  - ✓ Lock APIs



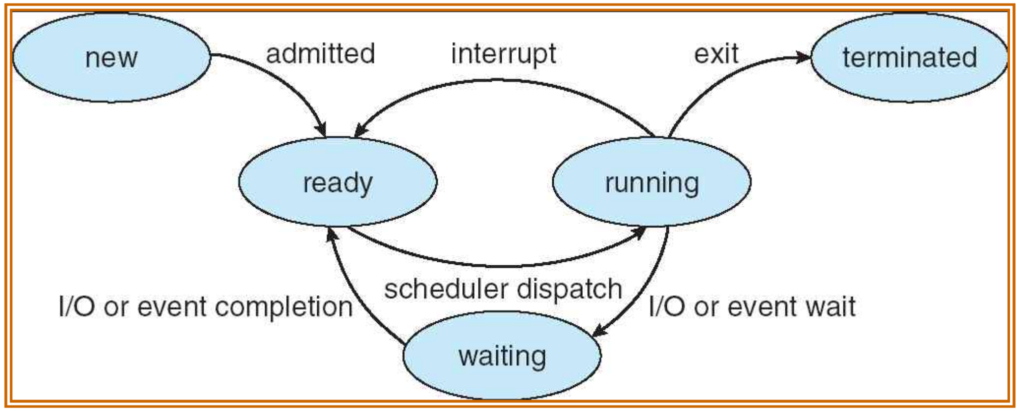
```
typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;

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

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

void unlock(lock_t *lock) {
    lock->turn = lock->turn + 1;
}
```

Figure 28.7: Ticket Locks

    - Ticket lock: 1) wish to acquire lock ➔ call fetchandadd() with lock->ticket, 2) if (myturn == lock->turn) enter the CS, 3) unlock ➔ add turn
    - Ensure progress for all threads (once a thread gets a ticket, it will be scheduled before other threads that have the tickets issued later)

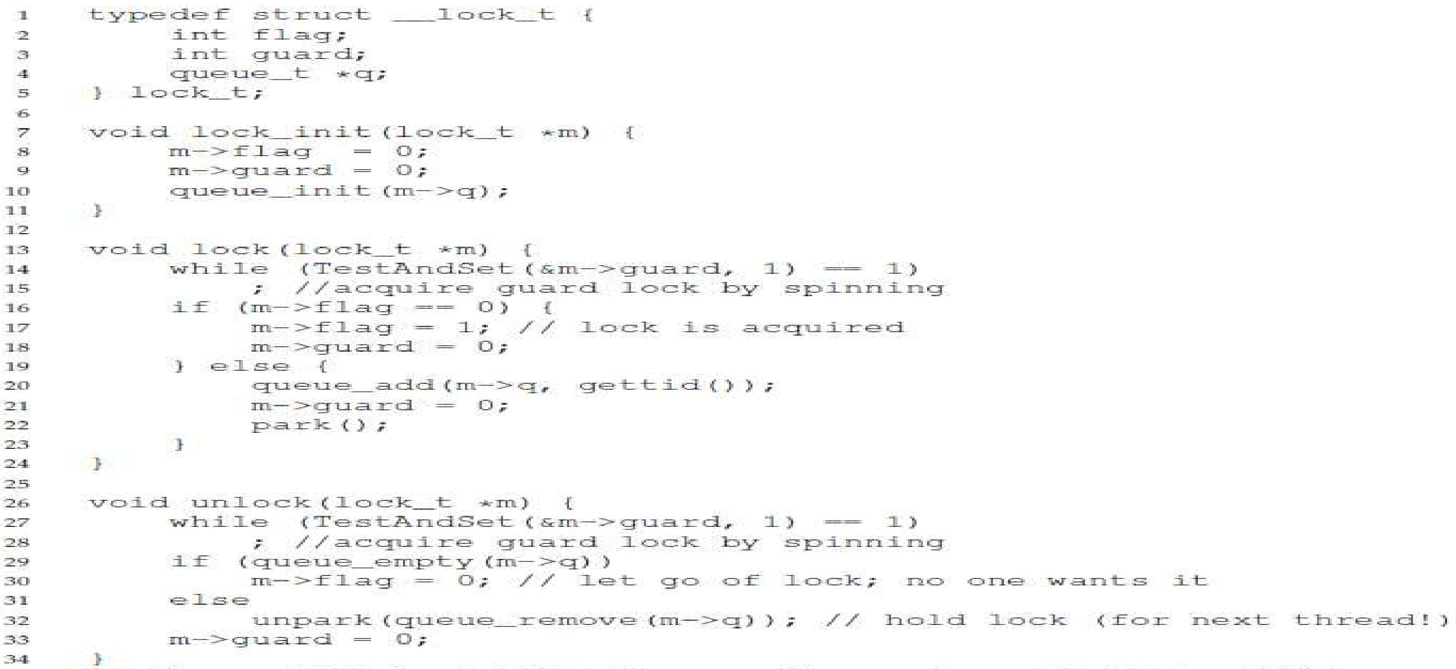
28.12 Too Much Spinning: What Now?

- Lock mechanisms
  - ✓ Spin lock
    - Busy waiting (endless check while using CPU)
    - Simple but inefficient (especially for the long critical section)
    - E.g.) N threads, RR scheduling, 1 thread acquires locks during the period of 1.5 time slice ➔ N -1 time slices are wasted
  - ✓ Sleep lock
    - Preempt and enter into the waiting (block) state, wakeup when the lock is released.
    - 1) Can utilize CPUs for more useful work, but 2) context switch for sleep is expensive (especially for the short critical section)
    - Need OS supports



28.14 Using Queues: Sleeping instead of Spinning

- Sleep
  - ✓ Better than spin since it gives a chance to schedule the thread that holds the lock (A lot of mutexes are implemented using sleep lock)
- Issues
  - ✓ Where to sleep? ➔ Using queue
  - ✓ How to wake up ➔ OS supports
    - E.g) Solaris supports park() to sleep and unpark() to wakeup a thread
    - Flag for lock variable, Guard for mutual exclusion of the flag, Queue for sleep

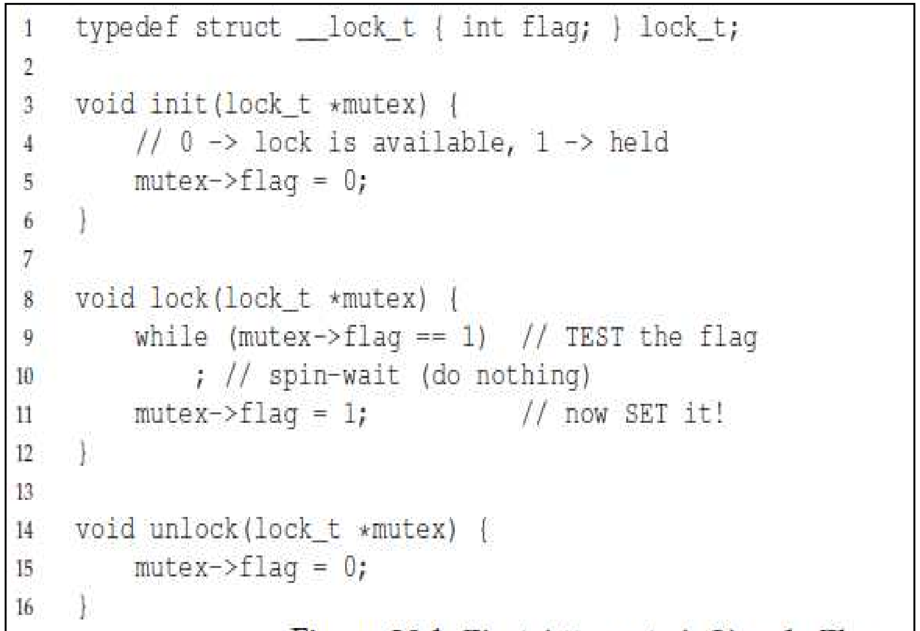


```
typedef struct __lock_t {
    int flag;
    int guard;
    queue_t *q;
} lock_t;

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

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

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

Figure 28.9: Lock With Queues, Test-and-set, Yield, And Wakeup

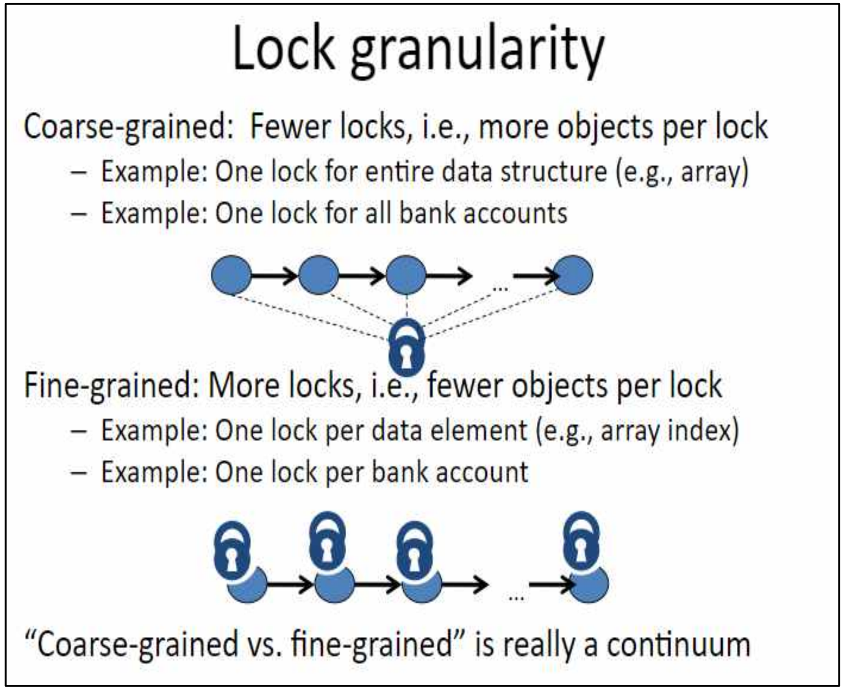
Quiz for 5th-Week 2nd-Lesson

- Quiz
  - ✓ 1. Explain the correctness of the lock mechanism shown in Figure 28.1 (if it is incorrect, explain why).
  - ✓ 2. Discuss the tradeoff between the coarse-grained lock and fine-grained lock.
  - ✓ Due: until 6 PM Friday of this week (2nd, April)



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

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

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

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

Figure 28.1: First Attempt: A Simple Flag



Lock granularity

Coarse-grained: Fewer locks, i.e., more objects per lock

- Example: One lock for entire data structure (e.g., array)
- Example: One lock for all bank accounts

Fine-grained: More locks, i.e., fewer objects per lock

- Example: One lock per data element (e.g., array index)
- Example: One lock per bank account

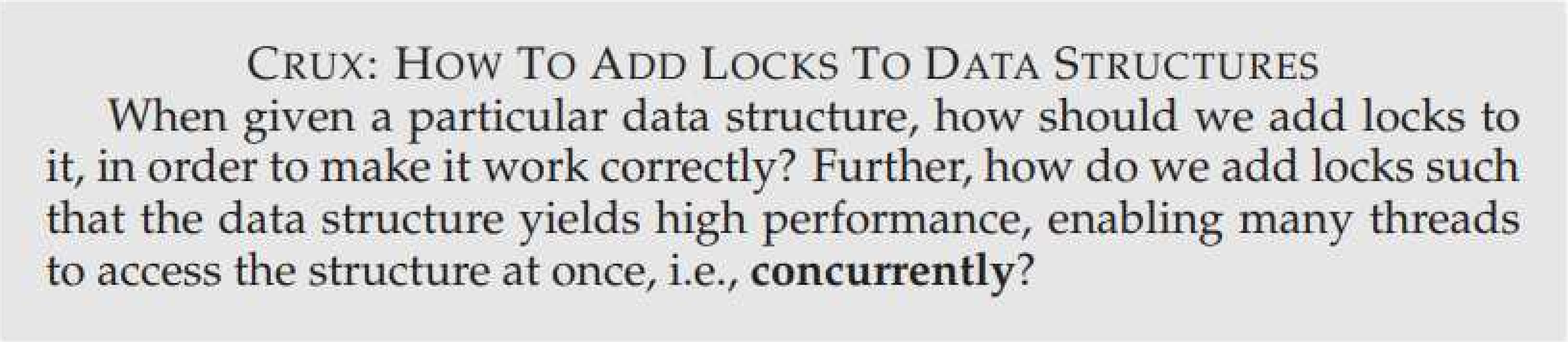
"Coarse-grained vs. fine-grained" is really a continuum

(Source: https://slideplayer.com/slide/4167835/)

Chap. 29 Lock-based Concurrent Data Structure

- How to use locks in data structure?
  - ✓ Concurrent Counters
  - ✓ Concurrent Linked lists
  - ✓ Concurrent Queues
  - ✓ Concurrent Hash Tables
  - ✓ …

- Data structure vs Concurrent data structure
  - ✓ Thread safe (support mutual exclusion)
  - ✓ Two Issues: 1) Correctness and 2) Performance

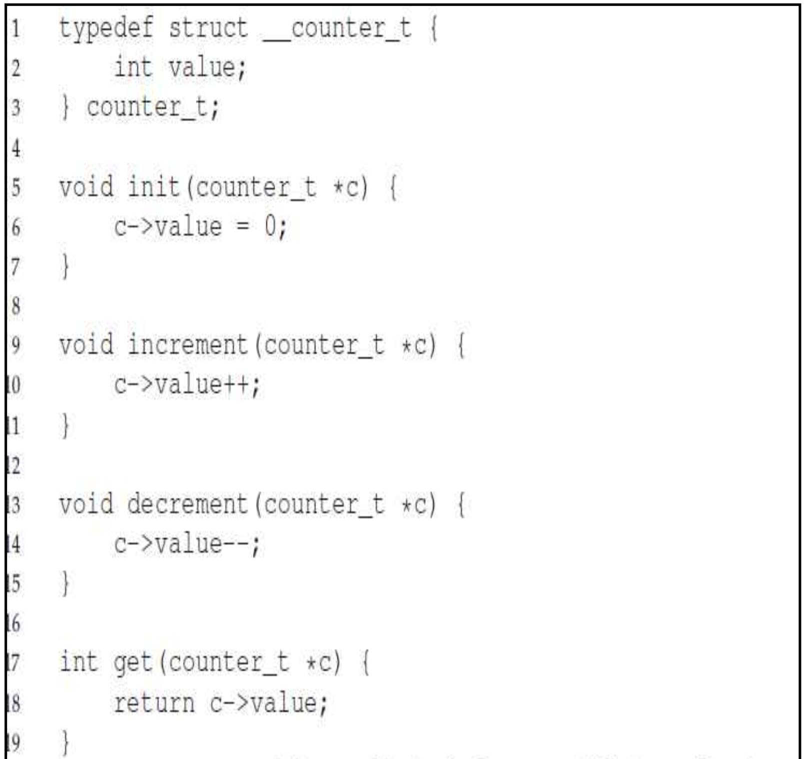


CRUX: HOW TO ADD LOCKS TO DATA STRUCTURES

When given a particular data structure, how should we add locks to it, in order to make it work correctly? Further, how do we add locks such that the data structure yields high performance, enabling many threads to access the structure at once, i.e., **concurrently**?

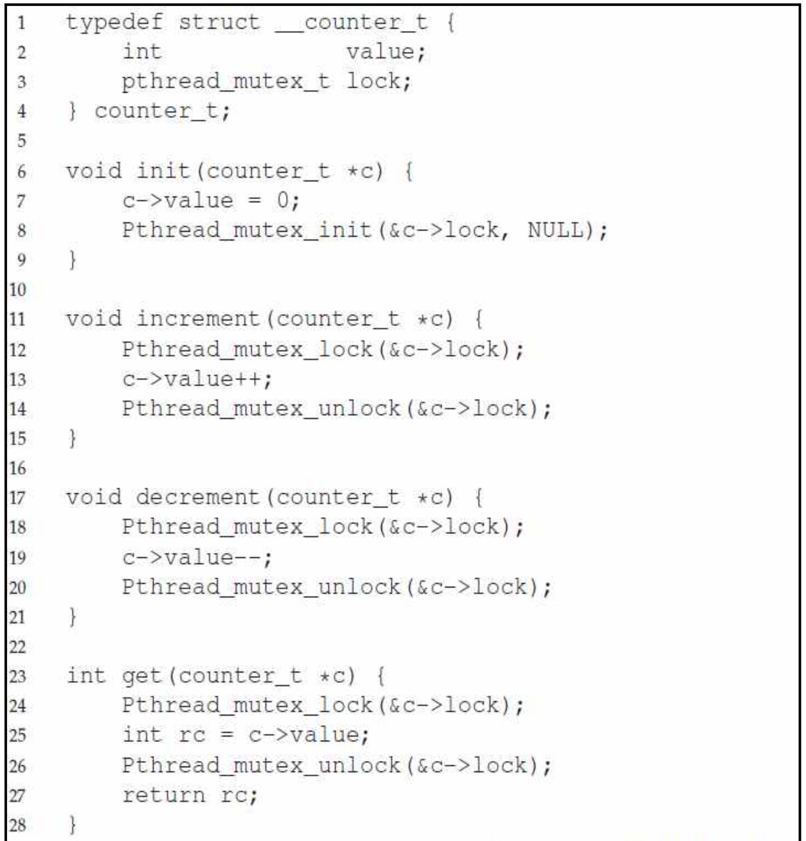
29.1 Concurrent Counters

- A Counter without locks: Figure 29.1
  - ✓ Incorrect under race condition
- A Counter with locks: Figure 29.2
  - ✓ Mutual exclusion using locks
  - ✓ Correct? How about performance?



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

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

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

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

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

Figure 29.1: **A Counter Without Locks**



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

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

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

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

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

Figure 29.2: **A Counter With Locks**

29.1 Concurrent Counters

- Traditional vs. Sloppy
  - ✓ Figure 29.3: total elapsed time when a thread (ranging one to four) updates the counter one million times
  - ✓ Precise (previous slide): poor scalable
  - ✓ Sloppy counter: Quite higher performance (a.k.a Scalable counter or Approximate counter)
    - A single global counter + Several local counters (usually one per CPU core) ➔ e.g. 4 core system: 1 global and 4 local counters
    - Lock for each counter for concurrency
    - Update local counter ➔ periodically update global counter (sloppiness, 5 in figure 29.4) ➔ Less contention ➔ Scalable

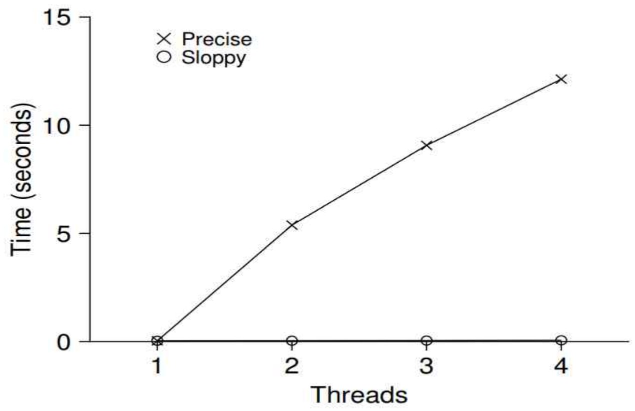
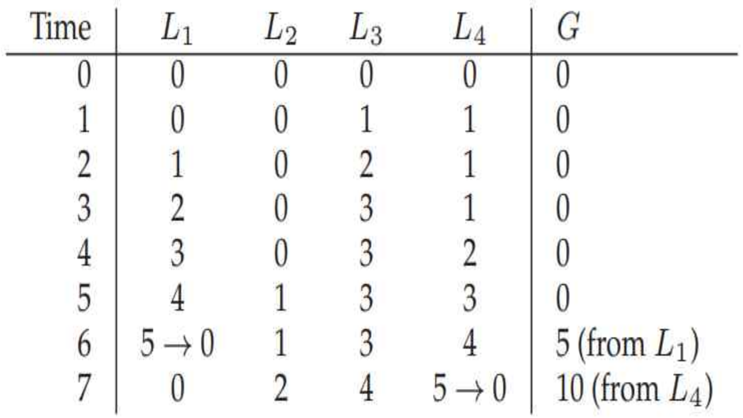


Figure 29.3: Performance of Traditional vs. Sloppy Counters

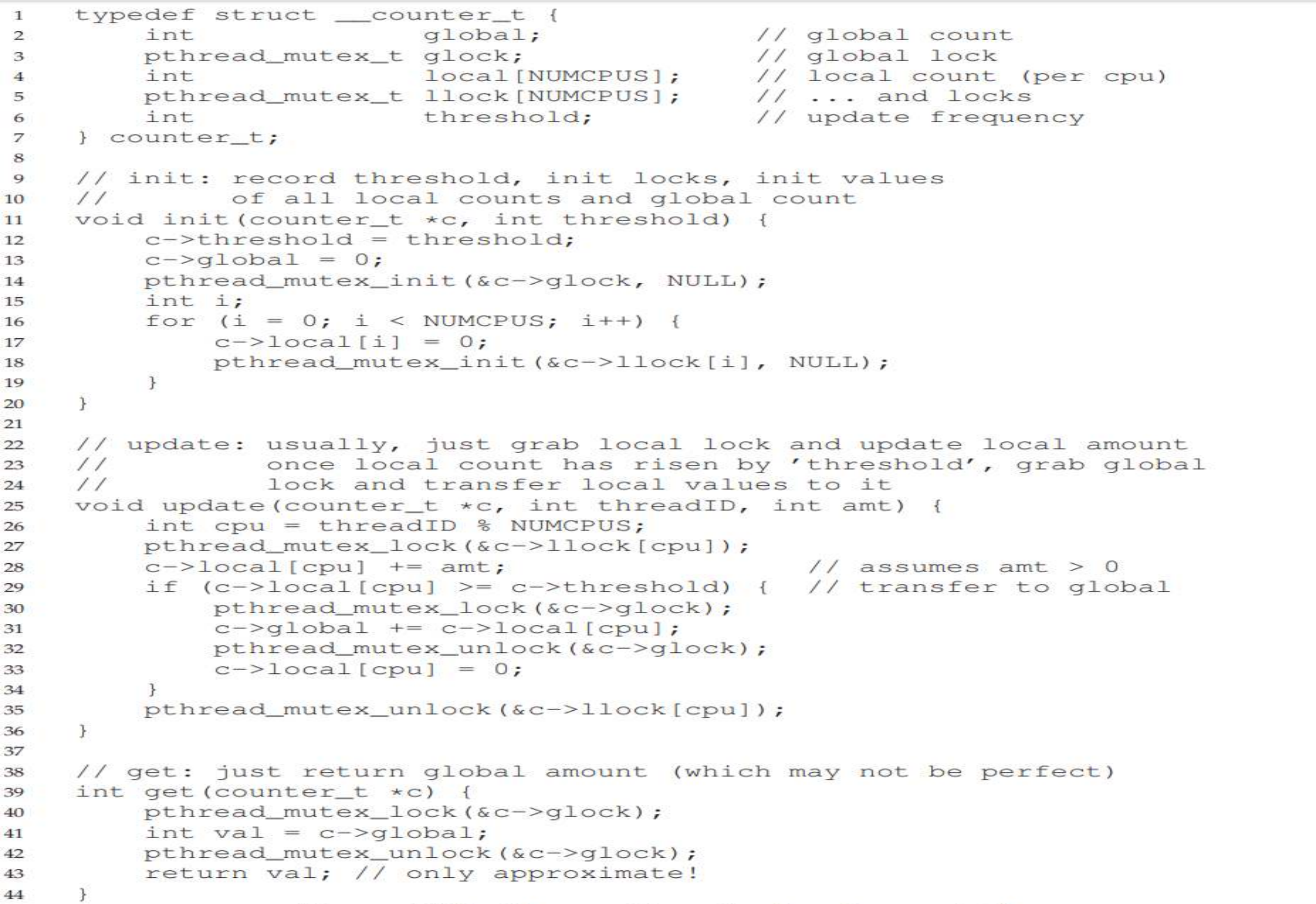


Time	L_1	L_2	L_3	L_4	G
0	0	0	0	0	0
1	0	0	1	1	0
2	1	0	2	1	0
3	2	0	3	1	0
4	3	0	3	2	0
5	4	1	3	3	0
6	5 → 0	1	3	4	5 (from L_1)
7	0	2	4	5 → 0	10 (from L_4)

Figure 29.4: Tracing the Sloppy Counters

29.1 Concurrent Counters

- Implementation of Sloppy Counter

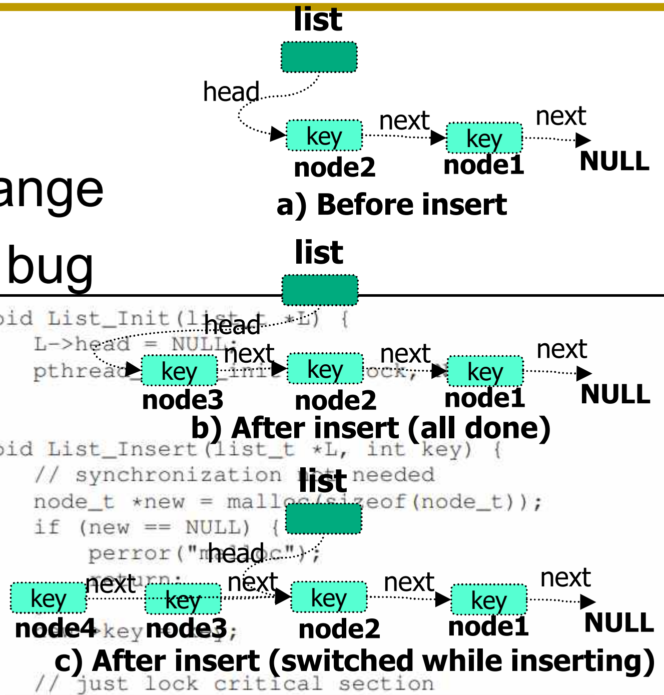


```
typedef struct __counter_t {
    int             global;            // global count
    pthread_mutex_t glock;             // global lock
    int             local[NUMCPUS];    // local count (per cpu)
    pthread_mutex_t llock[NUMCPUS];    // ... and locks
    int             threshold;        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

// init: record threshold, init locks, init values
//       of all local counts and global count
void init(counter_t *c, int threshold) {
    c->threshold = threshold;
    c->global = 0;
    pthread_mutex_init(&c->glock, NULL);
    int i;
    for (i = 0; i < NUMCPUS; i++) {
        c->local[i] = 0;
        pthread_mutex_init(&c->llock[i], NULL);
    }
}

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

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

Figure 29.5: Sloppy Counter Implementation

29.2 Concurrent Linked Lists

- Implementation
  - ✓ How to enhance scalability? lock range
  - ✓ Single return path in lookup(): less bug



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

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

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

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

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

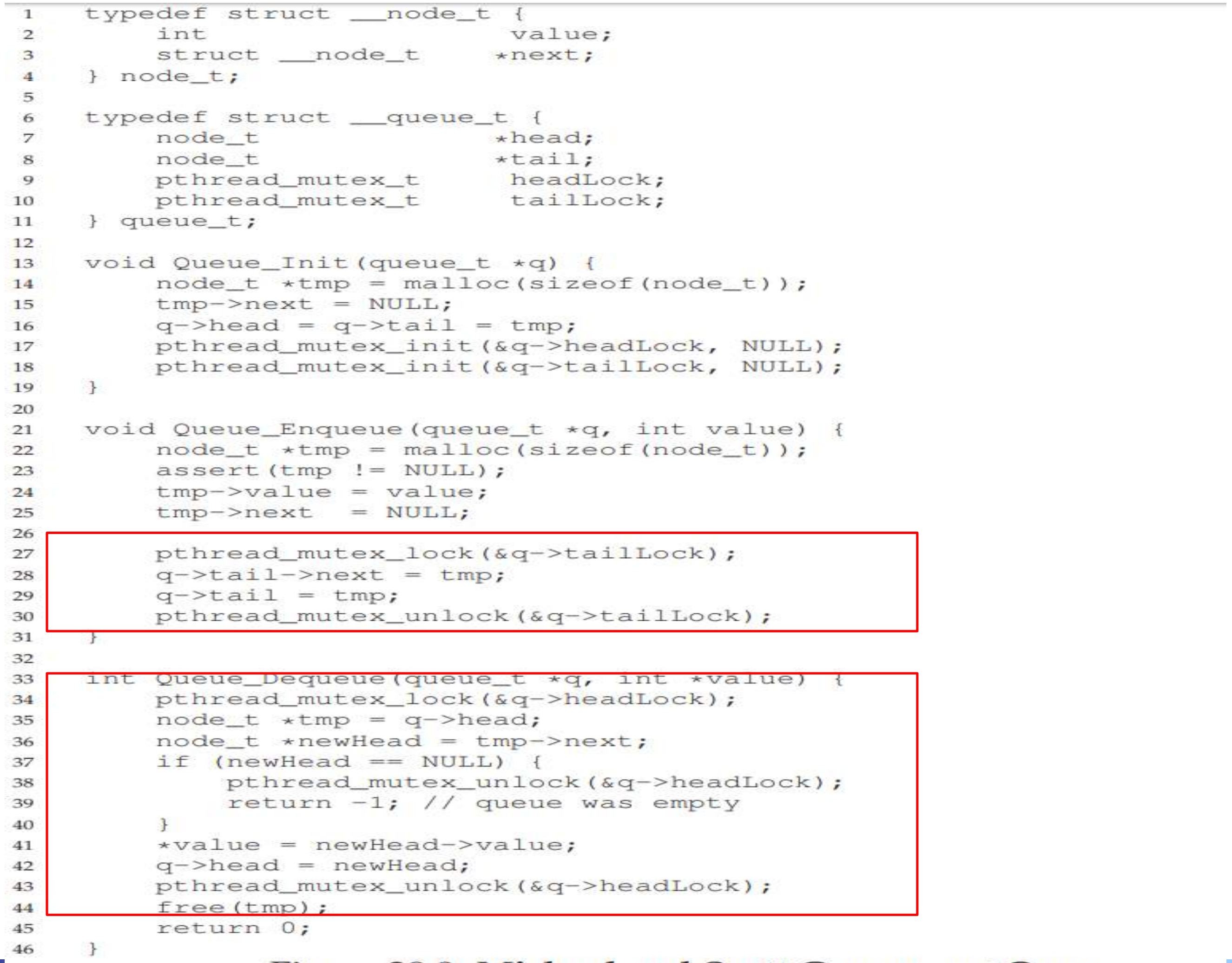
Figure 29.7: **Concurrent Linked List**

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

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

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

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

Figure 29.8: **Concurrent Linked List: Rewritten**

29.2 Concurrent Queues

- Implementation
  - ✓ How to enhance scalability? Multiple locks

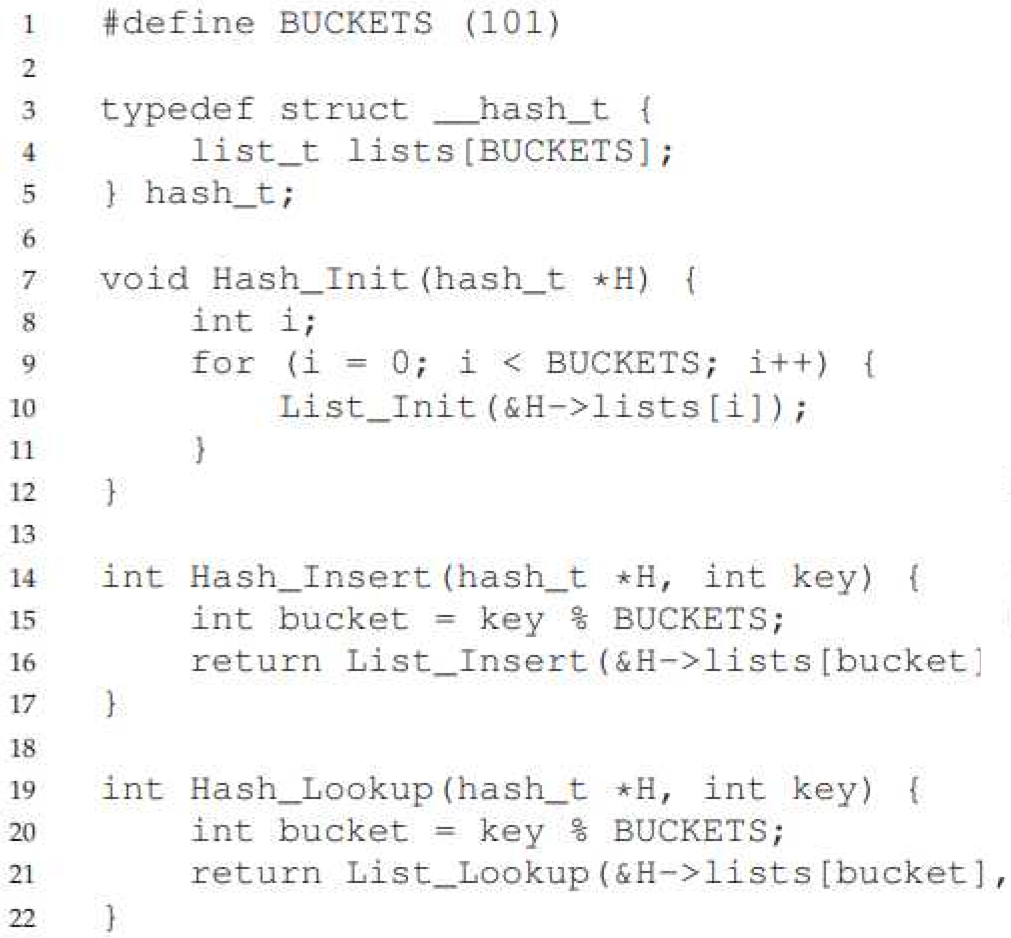


```
typedef struct __node_t {
    int                     value;
    struct __node_t        *next;
} node_t;

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

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

void Queue_Enqueue(queue_t *q, int value) {
    node_t *tmp = malloc(sizeof(node_t));
    assert(tmp != NULL);
    tmp->value = value;
    tmp->next  = NULL;

    pthread_mutex_lock(&q->tailLock);
    q->tail->next = tmp;
    q->tail = tmp;
    pthread_mutex_unlock(&q->tailLock);
}

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

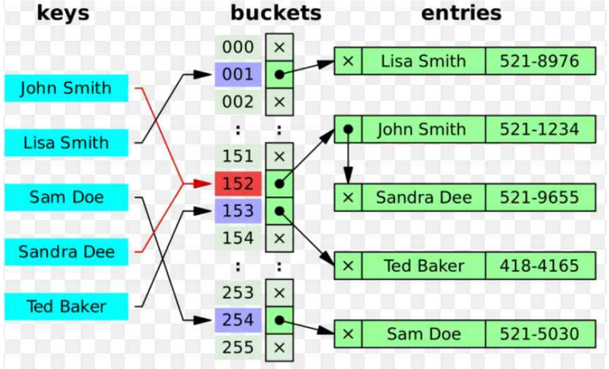
Figure 29.9: Michael and Scott Concurrent Queue

29.2 Concurrent Hash Table

- Implementation



```
#define BUCKETS (101)

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

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

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

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



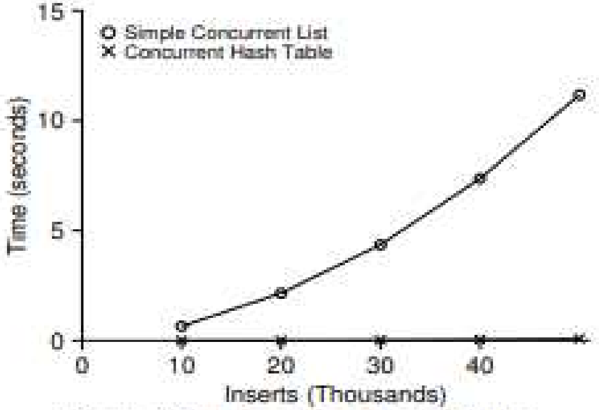


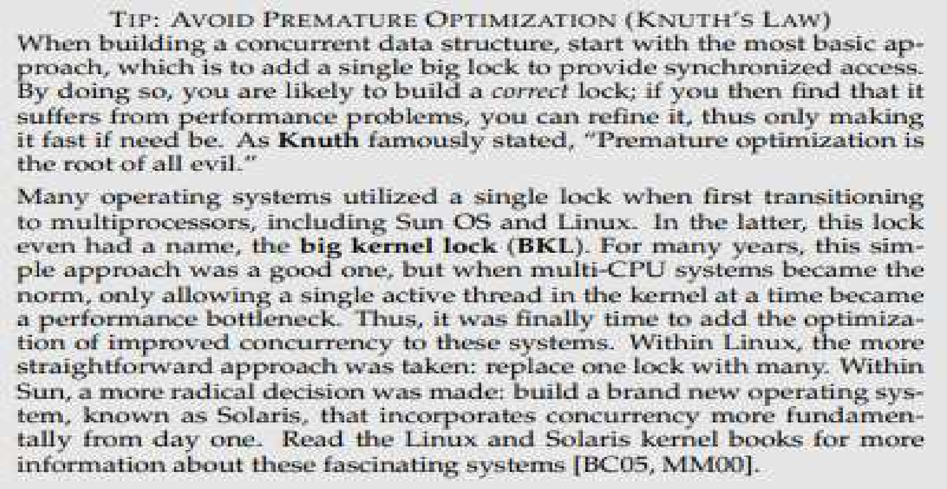
Figure 29.11: **Scaling Hash Tables**

Insert using concurrent hash vs list with a single lock (Fine-grained vs course-grained lock)

Figure 29.10: **A Concurrent Hash Table**

29.5 Summary

- Concurrency terms
  - ✓ Shared data, race condition, mutual exclusion
  - ✓ Lock before/after critical section
- Lock implementation
  - ✓ HW + OS cooperation
    - HW: atomic operations
    - OS: queue management
  - ✓ Spin lock and Sleep lock: Rule of thumb
    - Short critical section ➔ spin lock
    - Long critical section ➔ sleep lock
    - How about hybrid? ➔ Two-phase locks (spin at first, then sleep)
- Concurrent data structure



TIP: AVOID PREMATURE OPTIMIZATION (KNUTH'S LAW)

When building a concurrent data structure, start with the most basic approach, which is to add a single big lock to provide synchronized access. By doing so, you are likely to build a *correct* lock; if you then find that it suffers from performance problems, you can refine it, thus only making it fast if need be. As **Knuth** famously stated, "Premature optimization is the root of all evil."

Many operating systems utilized a single lock when first transitioning to multiprocessors, including Sun OS and Linux. In the latter, this lock even had a name, the **big kernel lock** (**BKL**). For many years, this simple approach was a good one, but when multi-CPU systems became the norm, only allowing a single active thread in the kernel at a time became a performance bottleneck. Thus, it was finally time to add the optimization of improved concurrency to these systems. Within Linux, the more straightforward approach was taken: replace one lock with many. Within Sun, a more radical decision was made: build a brand new operating system, known as Solaris, that incorporates concurrency more fundamentally from day one. Read the Linux and Solaris kernel books for more information about these fascinating systems [BC05, MM00].

DONALD ("DON") ERVIN KNUTH

United States – 1974

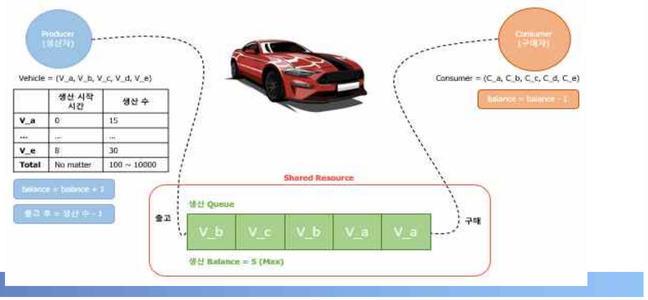
CITATION

For his major contributions to the analysis of algorithms and the design of programming languages, and in particular for his contributions to the "art of computer programming" through his well-known books in a continuous series by this title.

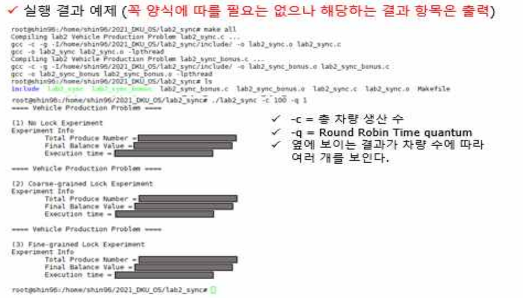
Lab 2: Concurrent Data Structure

- What to do?
  - ✓ Make a concurrent queue used by producers and consumers
    - See Lab2 document at https://github.com/DKU-EmbeddedSystem-Lab/2021_DKU_OS)
    - Refer to Figure 29.9 Concurrent Queue
  - ✓ Requirements
    - Three comparisons: 1) with/without locks, 2) fine-grained/coarse grained lock, 3) Performance under different number of threads
    - Reports: 1) Definition, 2) Design/Implementation, 3) Evaluation (including three comparisons and more), 4) Discussion (observation, new finding, …)
    - Submit: 1) Report ➔ e-learning campus (pdf), 2) report and source codes ➔ TA (with the name of “lab2_sync_32XXXXXX.tar”)
    - Environment: Linux (on virtual box or wsl or bare metal)
    - Deadline: 6 PM, 16th April (Friday)
  - ✓ Suggestion
    - If a concurrent queue is rather complex, making a sloppy counter is okay
    - Three comparisons: 1) with/without locks (Figure 29.1 vs 29.2), 2) fined-grained/coarse grained lock (Figure 29.2 vs 29.5) 3) Performance under different number of threads

Operating System
- OS Lab2 sync -
2021.
Dankook University
Hojin Shin



	생산 시작 시간	생산 수
V_a	0	15
...	...	...
V_e	8	30
Total	No matter	100 ~ 10000



✓ 실행 결과 예제 (꼭 양식에 따를 필요는 없으나 해당하는 결과 항목은 출력)

```
root@shin96:/home/shin96/2021_DKU_OS/lab2_sync# make all
Compiling lab2 Vehicle Production Problem lab2_sync.c ...
gcc -c -g -I/home/shin96/2021_DKU_OS/lab2_sync/include/ -o lab2_sync.o lab2_sync.c
gcc -o lab2_sync lab2_sync.o -lpthread
Compiling lab2 Vehicle Production Problem lab2_sync_bonus.c ...
gcc -c -g -I/home/shin96/2021_DKU_OS/lab2_sync/include/ -o lab2_sync_bonus.o lab2_sync_bonus.c
gcc -o lab2_sync_bonus lab2_sync_bonus.o -lpthread
root@shin96:/home/shin96/2021_DKU_OS/lab2_sync# ls
include  lab2_sync  lab2_sync_bonus  lab2_sync_bonus.c  lab2_sync_bonus.o  lab2_sync.c  lab2_sync.o  Makefile
root@shin96:/home/shin96/2021_DKU_OS/lab2_sync# ./lab2_sync -c 100 -q 1
==== Vehicle Production Problem ====

(1) No Lock Experiment
Experiment Info
        Total Produce Number = [redacted]
        Final Balance Value = [redacted]
        Execution time = [redacted]

==== Vehicle Production Problem ====

(2) Coarse-grained Lock Experiment
Experiment Info
        Total Produce Number = [redacted]
        Final Balance Value = [redacted]
        Execution time = [redacted]

==== Vehicle Production Problem ====

(3) Fine-grained Lock Experiment
Experiment Info
        Total Produce Number = [redacted]
        Final Balance Value = [redacted]
        Execution time = [redacted]

root@shin96:/home/shin96/2021_DKU_OS/lab2_sync#
```

✓ -c = 총 차량 생산 수
✓ -q = Round Robin Time quantum
✓ 옆에 보이는 결과가 차량 수에 따라 여러 개를 보인다.

Quiz for 6th-Week 1st-Lesson

- Quiz
  - ✓ 1. Discuss two differences between Figure 29.7 (Concurrent Linked List) and 29.8 (Concurrent Linked List: rewritten)
  - ✓ 2. Why the sloppy counter learned in Figure 29.5 is also called the approximate counter?
  - ✓ Due: until 6 PM Friday of this week (9th, April)

How can three people paint three walls? No problem if the walls are of equal size and the workers are of equal skills, but quite problematic when synchronization among them is required.

(Source: https://perso.telecom-paristech.fr/kuznetso/projects/Concur/concur/)