Lecture Note 4. Concurrency: Thread and Lock

April 2, 2020

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 operatic
- ✓ Spin vs. Sleep

Chap 29. Locked Data Structure

- ✓ list, queue, hash, …

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)

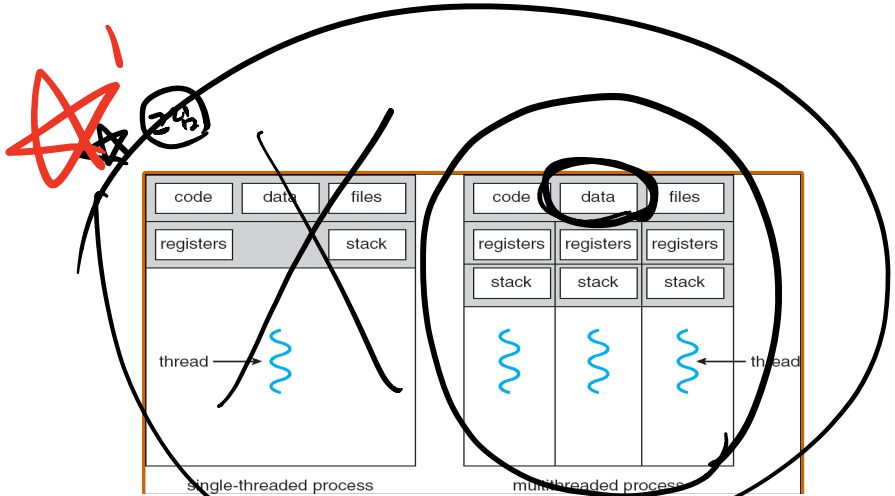
From now on

- ✓ 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

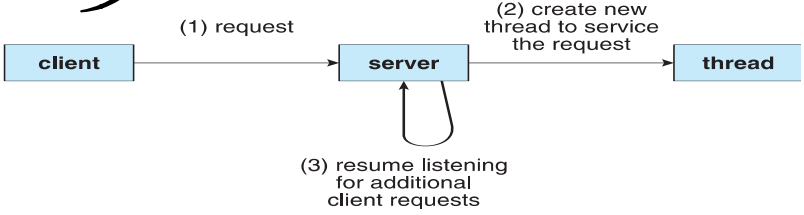


(Source: A. Silberschatz, "Operating system Concept")

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



- 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

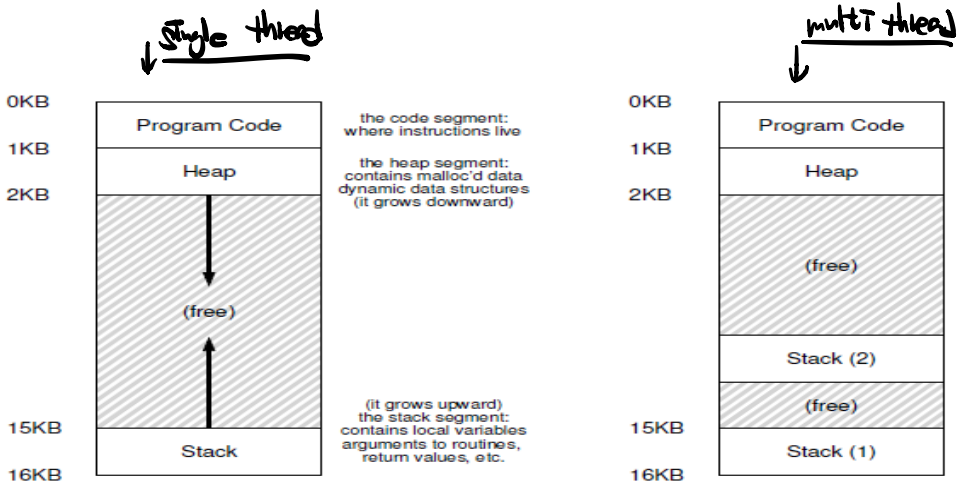


Figure 26.1: Single-Threaded And Multi-Threaded Address Spaces

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

```
1   #include <stdio.h>
2   #include <assert.h>
3   #include <pthread.h>
4
5   void *mythread(void *arg) {
6       printf("%s\n", (char *) arg);
7       return NULL;
8   }
9
10  int
11  main(int argc, char *argv[]) {
12      pthread_t p1, p2;
13      int rc;
14      printf("main: begin\n");
15      rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
16      rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
17      // join waits for the threads to finish
18      rc = pthread_join(p1, NULL); assert(rc == 0);
19      rc = pthread_join(p2, NULL); assert(rc == 0);
20      printf("main: end\n");
21      return 0;
22  }
```

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

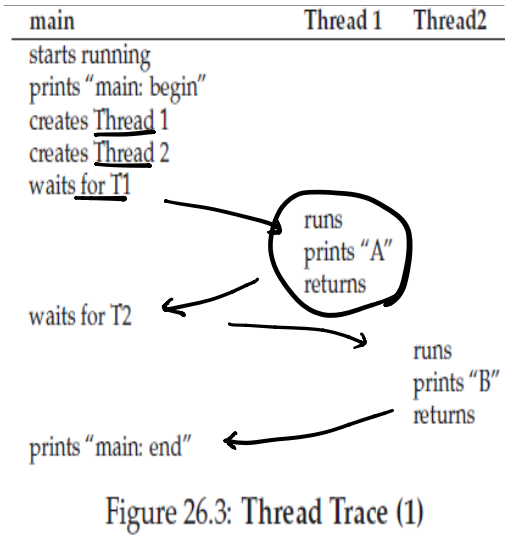


Figure 26.3: Thread Trace (1)

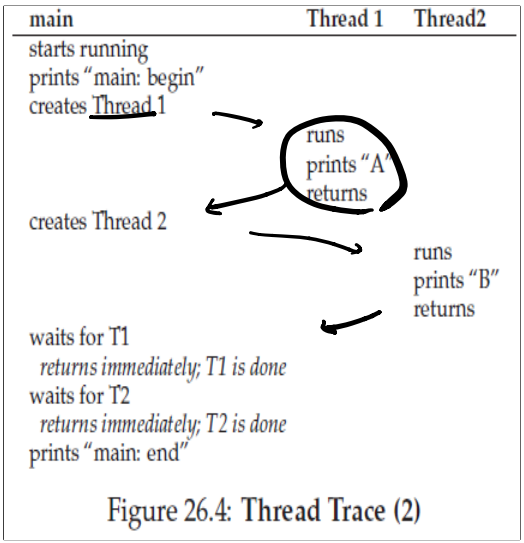


Figure 26.4: Thread Trace (2)

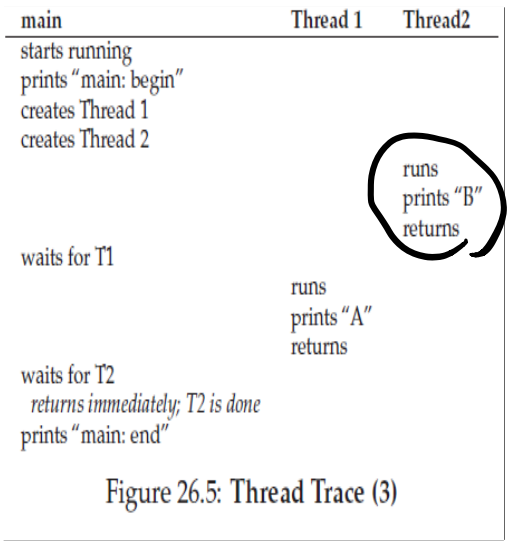


Figure 26.5: Thread Trace (3)

26.3 Why It Gets Worse: Shared Data

Shared data example (see Figure 2.5 in LN 1)

```
1   #include <stdio.h>
2   #include <pthread.h>
3   #include "mythreads.h"
4
5   static volatile int counter = 0;
6
7   //
8   // mythread()
9   //
10  // Simply adds 1 to counter repeatedly, in a loop
11  // No, this is not how you would add 10,000,000 to
12  // a counter, but it shows the problem nicely.
13  //
14  void *
15  mythread(void *arg)
16  {
17      printf("%s: begin\n", (char *) arg);
18      int i;
19      for (i = 0; i < 1e7; i++) {
20          counter = counter + 1;
21      }
22      printf("%s: done\n", (char *) arg);
23      return NULL;
24  }
25
26  //
27  // main()
28  //
29  // Just launches two threads (pthread_create)
30  // and then waits for them (pthread_join)
31  //
32  int
33  main(int argc, char *argv[])
34  {
35      pthread_t p1, p2;
36      printf("main: begin (counter = %d)\n", counter);
37      Pthread_create(&p1, NULL, mythread, "A");
38      Pthread_create(&p2, NULL, mythread, "B");
39
40      // join waits for the threads to finish
41      Pthread_join(p1, NULL);
42      Pthread_join(p2, NULL);
43      printf("main: done with both (counter = %d)\n", counter);
44      return 0;
45  }
```

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

26.3 Why It Gets Worse: Shared Data

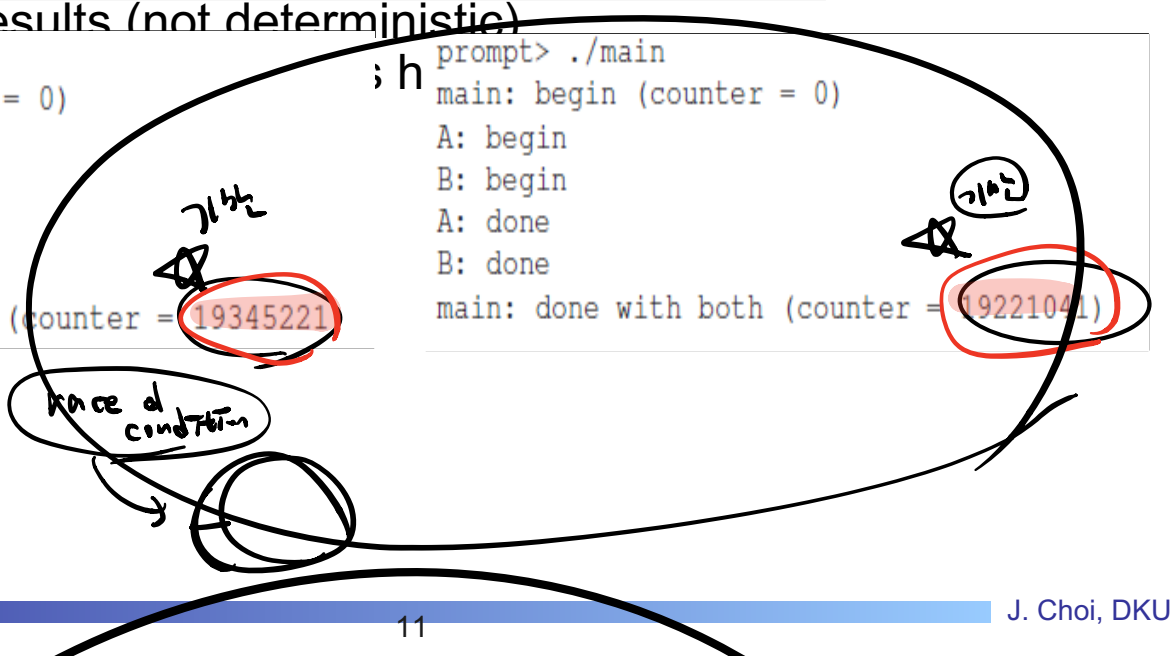
Results of the shared data example

```
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)
```

- Different results (not deterministic)

```
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)
```



26.4 The Heart of Problem: Uncontrolled Scheduling

High level viewpoint

CPU level viewpoint

Scheduling v

```
19        for (i = 0; i < 1e7; i++) {
20            counter = counter + 1;
21        }
```

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

T1 T2

CPU

pc
eax

DRAM

counter
(0x8049a1c)

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

? 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.

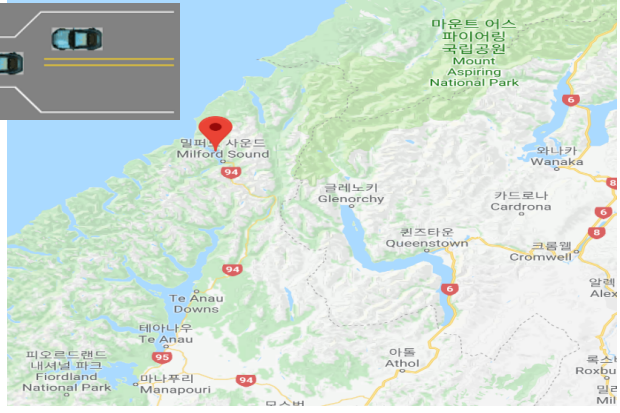
26.4 The Heart of Problem: Uncontrolled Scheduling

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

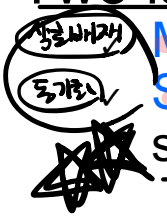
- ✓ Controlled scheduling: Do all or nothing (indivisible) ? atomicity
- ✓ The code that can result in the race condition ? critical section
  Code ranging from 100 to 108 in the example of the previous slide
- ✓ Allow only one thread in the critical section ? mutual exclusion



(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

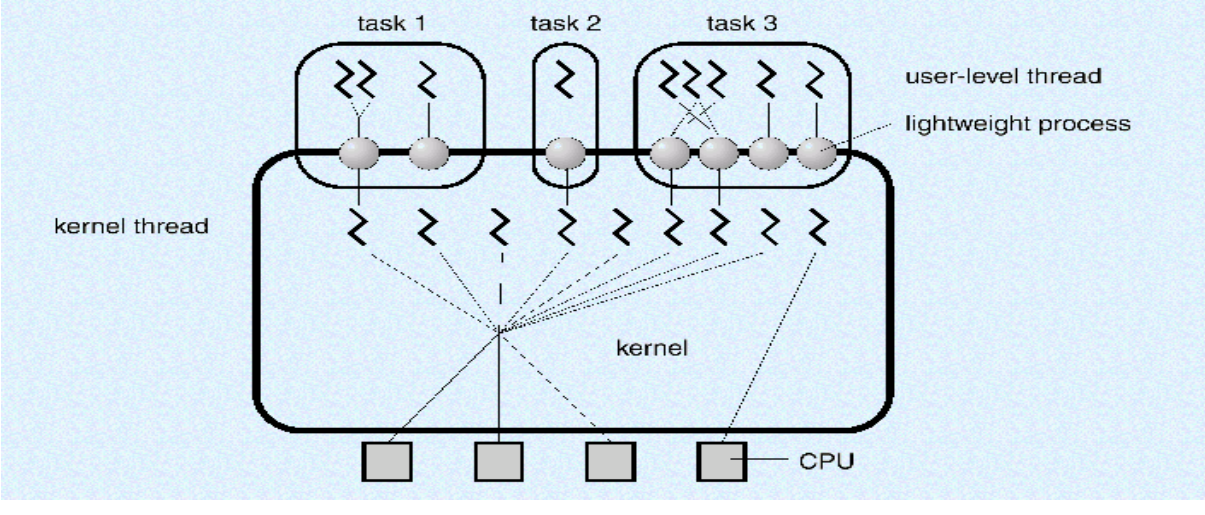
These four terms are so central to concurrent code that we thought it worth while to call them out explicitly. See some of Dijkstra's early work [D65,D68] for more details.

- 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);
```

- ✓ Arg… 2) attr… …ost case it is NULL (use default), 3) function pointer for start routine, 4) arguments

Example

```
1   #include <pthread.h>
2
3   typedef struct __myarg_t {
4       int a;
5       int b;
6   } myarg_t;
7
8   void *mythread(void *arg) {
9       myarg_t *m = (myarg_t *) arg;
10      printf("%d %d\n", m->a, m->b);
11      return NULL;
12  }
13
14  int
15  main(int argc, char *argv[]) {
16      pthread_t p;
17      int rc;
18
19      myarg_t args;
20      args.a = 10;
21      args.b = 20;
22      rc = pthread_create(&p, NULL, mythread, &args);
23      ...
24  }
```

Figure 27.1: Creating a Thread

27.2 Thread Completion

Wait for completion

synchronization

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

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

Example

```
1  #include <stdio.h>
2  #include <pthread.h>
3  #include <assert.h>
4  #include <stdlib.h>
5
6  typedef struct __myarg_t {
7      int a;
8      int b;
9  } myarg_t;
10
11 typedef struct __myret_t {
12     int x;
13     int y;
14 } myret_t;
15
16 void *mythread(void *arg) {
17     myarg_t *m = (myarg_t *) arg;
18     printf("%d %d\n", m->a, m->b);
19     myret_t *r = Malloc(sizeof(myret_t));
20     r->x = 1;
21     r->y = 2;
22     return (void *) r;
23 }
24
25 int
26 main(int argc, char *argv[]) {
27     int rc;
28     pthread_t p;
29     myret_t *m;
30
31     myarg_t args;
32     args.a = 10;
33     args.b = 20;
34     Pthread_create(&p, NULL, mythread, &args);
35     Pthread_join(p, (void **) &m);
36     printf("returned %d %d\n", m->x, m->y);
37     return 0;
38 }
```

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

```
1   void *mythread(void *arg) {
2       myarg_t *m = (myarg_t *) arg;
3       printf("%d %d\n", m->a, m->b);
4       myret_t r; // ALLOCATED ON STACK: BAD!
5       r.x = 1;
6       r.y = 2;
7       return (void *) &r;
8   }
```

```
choijm@choijm-VirtualBox: ~/2017_OS
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

Mutual exclusion API (mutex_***)

- ✓ Basic

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

- ✓ Argument: lock variable
- ✓ Example

```
pthread_mutex_t lock;
     ...
int rc = pthread_mutex_init(&lock, NULL);
assert(rc == 0); // always check success!
...
pthread_mutex_lock(&lock);
x = x + 1; // or whatever your critical section is
pthread_mutex_unlock(&lock);
```

Lock already hold case ? will not return from the call until acquiring the lock

- ✓ Extended

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

27.4 Condition Variables

Condition Variables

- ✓ Useful when some kind of signaling must take place between threads
- ✓ API

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

- ✓ Example

thread1 thread2

```
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);
```

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

- 9th Simple question to take attendance: What happen when we disable interrupt in the Figure 26.7 in 12 page? What is the concern when we disable it? (until 6 PM, April 15th)
- Interactive Q&A announcement
  - April 21th(Tuesday) at class time (10:30 am for Section 2, 2:30 pm for Section 3)
  - Zoom Meeting ID: 375-898-1997 (PW: To be announced)

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;` e possible such as adding a node to a linked list, hash update or more complex updates to shared data structures

Mutual exclusion using lock

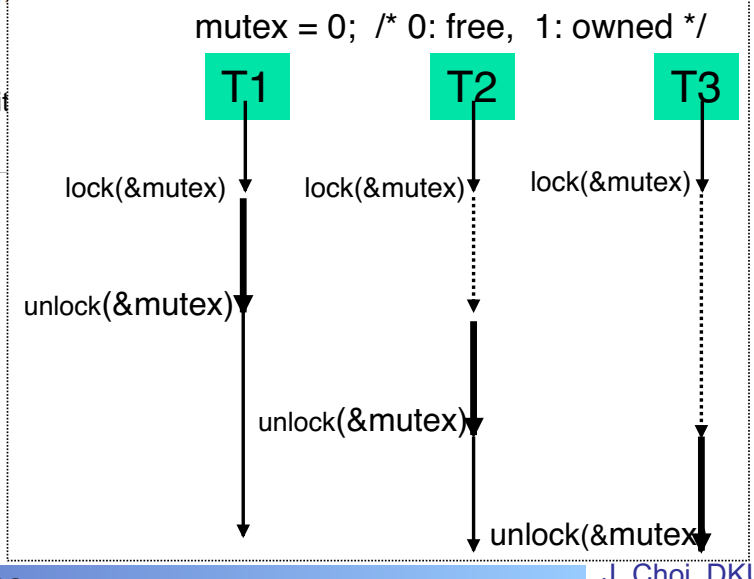
- Using lock/unlock before/after critical section (generic description)

```
1   lock_t mutex; // some globally
2   ...
3   lock(&mutex);
4
5   unlock(&mutex)
```

Guarantee that only one thread can enter the crit

- Real example: pthread

pthread_mutex_lock()
pthread_mutex_unlock()

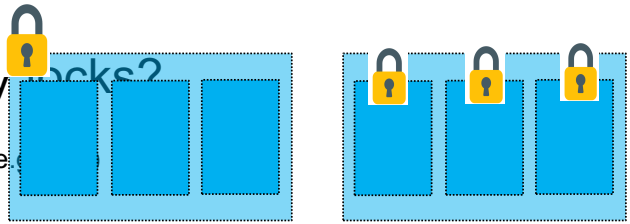


28.3 Building A Lock / 28.4 Evaluating Locks

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 locks?
- ✓ 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

```
1  void lock() {
2      DisableInterrupts();
3  }
4  void unlock() {
5      EnableInterrupts();
6  }
```

Pros)

- Sim

Cons)

- Dis… …ight 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

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

Figure 28.1: First Attempt: A Simple Flag

Is it correct?

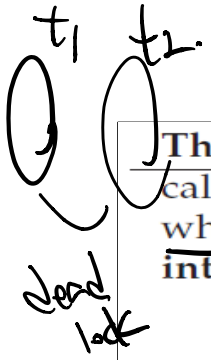
28.6 Test-and-Set (Atomic Exchange)

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**

28.6 Test-and-Set (Atomic Exchange)

How to build the lock()/unlock() APIs?

- ✓ There are many SW-only approach: 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
}
```

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

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

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

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

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 Compare-and-Swap

Another atomic operation (example of the third solution)

- ✓ Compare-and-Swap instruction

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

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

- ✓ Lo

- ✓ Cf) Real implementation of CompareAndSwap()

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

```
1   char CompareAndSwap(int *ptr, int old, int new) {
2       unsigned char ret;
3
4       // Note that sete sets a 'byte' not the word
5       __asm__ __volatile__ (
6               "  lock\n"
7               "  cmpxchgl %2,%1\n"
8               "  sete %0\n"
9               : "=q" (ret), "=m" (*ptr)
10              : "r" (new), "m" (*ptr), "a" (old)
11              : "memory");
12      return ret;
13  }
```

28.10 Load-Linked and Store-Conditional (Optional)

Another atomic operation, again

- ✓ Load-Linked and Store-Conditional supported by MIPS, ARM, …
- ✓ Loc

```
1   int LoadLinked(int *ptr) {
2       return *ptr;
3   }
4
5   int StoreConditional(int *ptr, int value) {
6       if (no one has updated *ptr since the LoadLinked to this address) {
7           *ptr = value;
8           return 1; // success!
9       } else {
10          return 0; // failed to update
11      }
12  }
```

Figure 28.5: **Load-linked And Store-conditional**

```
1   void lock(lock_t *lock) {
2       while (1) {
3           while (LoadLinked(&lock->flag) == 1)
4               ; // spin until it's zero
5           if (StoreConditional(&lock->flag, 1) == 1)
6               return; // if set-it-to-1 was a success: all done
7                       // otherwise: try it all over again
8       }
9   }
10
11  void unlock(lock_t *lock) {
12      lock->flag = 0;
13  }
```

Figure 28.6: **Using LL/SC To Build A Lock**

28.11 Fetch-and-Add (Optional)

Final atomic operation

- ✓ Fetch-and-Add

```
1    int FetchAndAdd(int *ptr) {
2        int old = *ptr;
3        *ptr = old + 1;
4        return old;
5    }
```

- ✓ Loc

```
1    typedef struct __lock_t {
2        int ticket;
3        int turn;
4    } lock_t;
5
6    void lock_init(lock_t *lock) {
7        lock->ticket = 0;
8        lock->turn   = 0;
9    }
10
11   void lock(lock_t *lock) {
12       int myturn = FetchAndAdd(&lock->ticket);
13       while (lock->turn != myturn)
14           ; // spin
15   }
16
17   void unlock(lock_t *lock) {
18       lock->turn = lock->turn + 1;
19   }
```

Figure 28.7: **Ticket Locks**

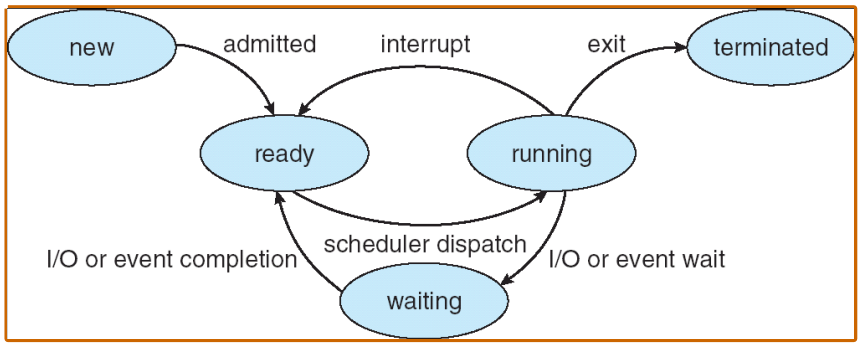
) enter the

ls that have

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 and just yield for giving a chance to schedule the thread that holds the 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

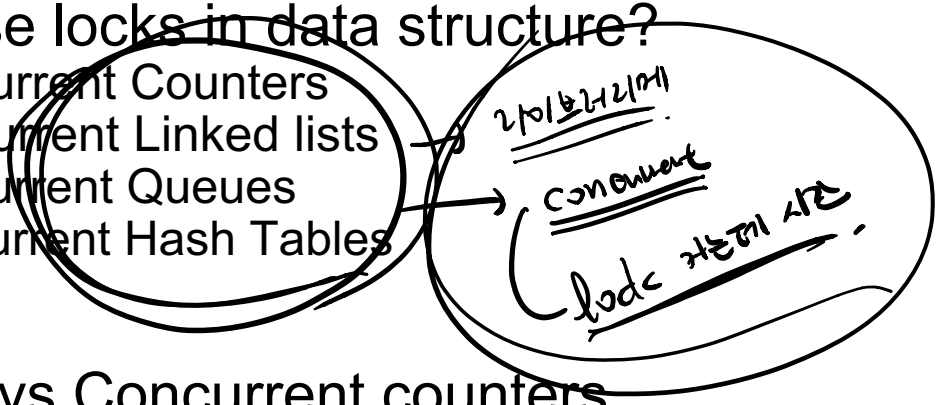
```
1   typedef struct __lock_t {
2       int flag;
3       int guard;
4       queue_t *q;
5   } lock_t;
6
7   void lock_init(lock_t *m) {
8       m->flag  = 0;
9       m->guard = 0;
10      queue_init(m->q);
11  }
12
13  void lock(lock_t *m) {
14      while (TestAndSet(&m->guard, 1) == 1)
15          ; //acquire guard lock by spinning
16      if (m->flag == 0) {
17          m->flag = 1; // lock is acquired
18          m->guard = 0;
19      } else {
20          queue_add(m->q, gettid());
21          m->guard = 0;
22          park();
23      }
24  }
25
26  void unlock(lock_t *m) {
27      while (TestAndSet(&m->guard, 1) == 1)
28          ; //acquire guard lock by spinning
29      if (queue_empty(m->q))
30          m->flag = 0; // let go of lock; no one wants it
31      else
32          unpark(queue_remove(m->q)); // hold lock (for next thread!)
33      m->guard = 0;
34  }
```

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

Chap. 29 Lock-based Concurrent Data Structure

How to use locks in data structure?

- ✓ Concurrent Counters
- ✓ Concurrent Linked lists
- ✓ Concurrent Queues
- ✓ Concurrent Hash Tables
- ✓ …



Counters vs Concurrent counters

- ✓ Thread safe
- ✓ Issue: Correctness and 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?

```
1  typedef struct __counter_t {
2      int value;
3  } counter_t;
4
5  void init(counter_t *c) {
6      c->value = 0;
7  }
8
9  void increment(counter_t *c) {
10     c->value++;
11 }
12
13 void decrement(counter_t *c) {
14     c->value--;
15 }
16
17 int get(counter_t *c) {
18     return c->value;
19 }
```

Figure 29.1: A Counter Without Locks

```
1  typedef struct __counter_t {
2      int             value;
3      pthread_mutex_t lock;
4  } counter_t;
5
6  void init(counter_t *c) {
7      c->value = 0;
8      Pthread_mutex_init(&c->lock, NULL);
9  }
10
11 void increment(counter_t *c) {
12     Pthread_mutex_lock(&c->lock);
13     c->value++;
14     Pthread_mutex_unlock(&c->lock);
15 }
16
17 void decrement(counter_t *c) {
18     Pthread_mutex_lock(&c->lock);
19     c->value--;
20     Pthread_mutex_unlock(&c->lock);
21 }
22
23 int get(counter_t *c) {
24     Pthread_mutex_lock(&c->lock);
25     int rc = c->value;
26     Pthread_mutex_unlock(&c->lock);
27     return rc;
28 }
```

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

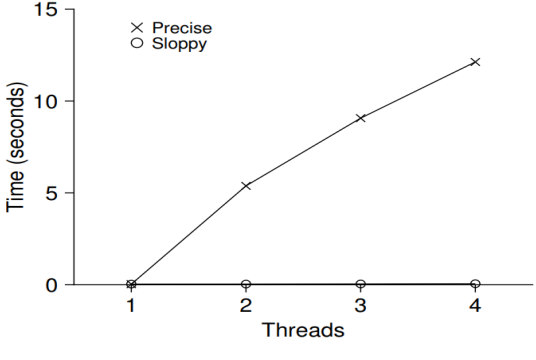


Figure 29.3: **Performance of Traditional vs. Sloppy Counters**

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	$5 \rightarrow 0$	1	3	4	5 (from L_1)
7	0	2	4	$5 \rightarrow 0$	10 (from L_4)

Figure 29.4: **Tracing the Sloppy Counters**

29.1 Concurrent Counters

Implementation of Sloppy Counter

```
1   typedef struct __counter_t {
2       int             global;          // global count
3       pthread_mutex_t glock;           // global lock
4       int             local[NUMCPUS];  // local count (per cpu)
5       pthread_mutex_t llock[NUMCPUS];  // ... and locks
6       int             threshold;       // update frequency
7   } counter_t;
8
9   // init: record threshold, init locks, init values
10  //       of all local counts and global count
11  void init(counter_t *c, int threshold) {
12      c->threshold = threshold;
13      c->global = 0;
14      pthread_mutex_init(&c->glock, NULL);
15      int i;
16      for (i = 0; i < NUMCPUS; i++) {
17          c->local[i] = 0;
18          pthread_mutex_init(&c->llock[i], NULL);
19      }
20  }
21
22  // update: usually, just grab local lock and update local amount
23  //         once local count has risen by 'threshold', grab global
24  //         lock and transfer local values to it
25  void update(counter_t *c, int threadID, int amt) {
26      int cpu = threadID % NUMCPUS;
27      pthread_mutex_lock(&c->llock[cpu]);
28      c->local[cpu] += amt;                    // assumes amt > 0
29      if (c->local[cpu] >= c->threshold) {   // transfer to global
30          pthread_mutex_lock(&c->glock);
31          c->global += c->local[cpu];
32          pthread_mutex_unlock(&c->glock);
33          c->local[cpu] = 0;
34      }
35      pthread_mutex_unlock(&c->llock[cpu]);
36  }
37
38  // get: just return global amount (which may not be perfect)
39  int get(counter_t *c) {
40      pthread_mutex_lock(&c->glock);
41      int val = c->global;
42      pthread_mutex_unlock(&c->glock);
43      return val; // only approximate!
44  }
```

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

```
1  // basic node structure
2  typedef struct __node_t {
3      int                 key;
4      struct __node_t     *next;
5  } node_t;
6
7  // basic list structure (one used per list)
8  typedef struct __list_t {
9      node_t              *head;
10     pthread_mutex_t     lock;
11 } list_t;
12
13 void List_Init(list_t *L) {
14     L->head = NULL;
15     pthread_mutex_init(&L->lock, NULL);
16 }
17
18 int List_Insert(list_t *L, int key) {
19     pthread_mutex_lock(&L->lock);
20     node_t *new = malloc(sizeof(node_t));
21     if (new == NULL) {
22         perror("malloc");
23         pthread_mutex_unlock(&L->lock);
24         return -1; // fail
25     }
26     new->key  = key;
27     new->next = L->head;
28     L->head   = new;
29     pthread_mutex_unlock(&L->lock);
30     return 0; // success
31 }
32
33 int List_Lookup(list_t *L, int key) {
34     pthread_mutex_lock(&L->lock);
35     node_t *curr = L->head;
36     while (curr) {
37         if (curr->key == key) {
38             pthread_mutex_unlock(&L->lock);
39             return 0; // success
40         }
41         curr = curr->next;
42     }
43     pthread_mutex_unlock(&L->lock);
44     return -1; // failure
45 }
```

Figure 29.7: **Concurrent Linked List**

```
1  void List_Init(list_t *L) {
2      L->head = NULL;
3      pthread_mutex_init(&L->lock, NULL);
4  }
5
6  void List_Insert(list_t *L, int key) {
7      // synchronization not needed
8      node_t *new = malloc(sizeof(node_t));
9      if (new == NULL) {
10         perror("malloc");
11         return;
12     }
13     new->key = key;
14
15     // just lock critical section
16     pthread_mutex_lock(&L->lock);
17     new->next = L->head;
18     L->head   = new;
19     pthread_mutex_unlock(&L->lock);
20 }
21
22 int List_Lookup(list_t *L, int key) {
23     int rv = -1;
24     pthread_mutex_lock(&L->lock);
25     node_t *curr = L->head;
26     while (curr) {
27         if (curr->key == key) {
28             rv = 0;
29             break;
30         }
31         curr = curr->next;
32     }
33     pthread_mutex_unlock(&L->lock);
34     return rv; // now both success and failure
35 }
```

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

29.2 Concurrent Queues

Implementation

- ✓ How to enhance scalability? Multiple locks

```
1  typedef struct __node_t {
2      int                 value;
3      struct __node_t     *next;
4  } node_t;
5
6  typedef struct __queue_t {
7      node_t              *head;
8      node_t              *tail;
9      pthread_mutex_t      headLock;
10     pthread_mutex_t      tailLock;
11 } queue_t;
12
13 void Queue_Init(queue_t *q) {
14     node_t *tmp = malloc(sizeof(node_t));
15     tmp->next = NULL;
16     q->head = q->tail = tmp;
17     pthread_mutex_init(&q->headLock, NULL);
18     pthread_mutex_init(&q->tailLock, NULL);
19 }
20
21 void Queue_Enqueue(queue_t *q, int value) {
22     node_t *tmp = malloc(sizeof(node_t));
23     assert(tmp != NULL);
24     tmp->value = value;
25     tmp->next  = NULL;
26
27     pthread_mutex_lock(&q->tailLock);
28     q->tail->next = tmp;
29     q->tail = tmp;
30     pthread_mutex_unlock(&q->tailLock);
31 }
32
33 int Queue_Dequeue(queue_t *q, int *value) {
34     pthread_mutex_lock(&q->headLock);
35     node_t *tmp = q->head;
36     node_t *newHead = tmp->next;
37     if (newHead == NULL) {
38         pthread_mutex_unlock(&q->headLock);
39         return -1; // queue was empty
40     }
41     *value = newHead->value;
42     q->head = newHead;
43     pthread_mutex_unlock(&q->headLock);
44     free(tmp);
45     return 0;
46 }
```

Figure 29.9: **Michael and Scott Concurrent Queue**

29.2 Concurrent Hash Table

Implementation

```
1   #define BUCKETS (101)
2
3   typedef struct __hash_t {
4       list_t lists[BUCKETS];
5   } hash_t;
6
7   void Hash_Init(hash_t *H) {
8       int i;
9       for (i = 0; i < BUCKETS; i++) {
10          List_Init(&H->lists[i]);
11      }
12  }
13
14  int Hash_Insert(hash_t *H, int key) {
15      int bucket = key % BUCKETS;
16      return List_Insert(&H->lists[bucket], key);
17  }
18
19  int Hash_Lookup(hash_t *H, int key) {
20      int bucket = key % BUCKETS;
21      return List_Lookup(&H->lists[bucket], key);
22  }
```

Figure 29.10: **A Concurrent Hash Table**

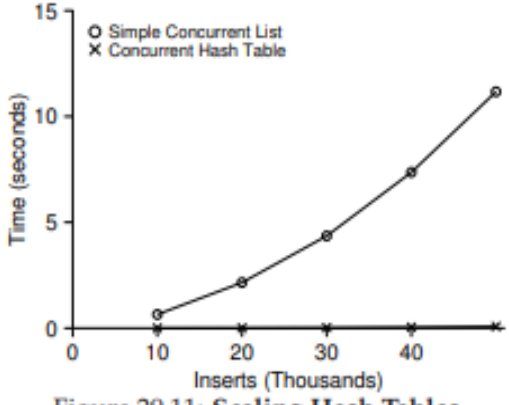


Figure 29.11: **Scaling Hash Tables**

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

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

- 10th Simple question to take attendance : Why the name of the atomic operation discussed in 28 page is “test-and-set”. Explain it using 26 page? (until 6 PM, April 22th)

- Lab 2: Concurrent BST (Binary Search Tree)
  - Requirement: 1) team (2 persons), 2) report (discussion and result snapshot with/without mutex) → print out, 3) Source code and report → email to TA
  - Environment → See Lab. 2 (hint, do the concurrent list (or queue) exercise, first)
  - Due: until May 5th , 6 PM
  - Bonus: Comparison with coarse-grained vs. fine-grained lock
  - Suggestion: If BST is rather complicated, do Lab2 with Hash or Counter (see 38 ~41 pages)

Appendix

27.5 Compiling and Running thread

- ✓ Compiling
  Header files (pthread.h) + library (libpthread.a, libpthread.so)
- ✓ Guidelines for thread programming (for secure coding)

ASIDE: THREAD API GUIDELINES

There are a number of small but important things to remember when you use the POSIX thread library (or really, any thread library) to build a multi-threaded program. They are:

- **Keep it simple.** Above all else, any code to lock or signal between threads should be as simple as possible. Tricky thread interactions lead to bugs.
- **Minimize thread interactions.** Try to keep the number of ways in which threads interact to a minimum. Each interaction should be carefully thought out and constructed with tried and true approaches (many of which we will learn about in the coming chapters).
- **Initialize locks and condition variables.** Failure to do so will lead to code that sometimes works and sometimes fails in very strange ways.
- **Check your return codes.** Of course, in any C and UNIX programming you do, you should be checking each and every return code, and it's true here as well. Failure to do so will lead to bizarre and hard to understand behavior, making you likely to (a) scream, (b) pull some of your hair out, or (c) both.
- **Be careful with how you pass arguments to, and return values from, threads.** In particular, any time you are passing a reference to a variable allocated on the stack, you are probably doing something wrong.
- **Each thread has its own stack.** As related to the point above, please remember that each thread has its own stack. Thus, if you have a locally-allocated variable inside of some function a thread is executing, it is essentially *private* to that thread; no other thread can (easily) access it. To share data between threads, the values must be in the heap or otherwise some locale that is globally accessible.
- **Always use condition variables to signal between threads.** While it is often tempting to use a simple flag, don't do it.
- **Use the manual pages.** On Linux, in particular, the pthread man pages are highly informative and discuss much of the nuances presented here, often in even more detail. Read them carefully!

Appendix

28.10 Less code is better code

28.

TIP: LESS CODE IS BETTER CODE (LAUER'S LAW)

Programmers tend to brag about how much code they wrote to do something. Doing so is fundamentally broken. What one should brag about, rather, is how *little* code one wrote to accomplish a given task. Short, concise code is always preferred; it is likely easier to understand and has fewer bugs. As Hugh Lauer said, when discussing the construction of the Pilot operating system: "If the same people had twice as much time, they could produce as good of a system in half the code." [L81] We'll call this **Lauer's Law**, and it is well worth remembering. So next time you're bragging about how much code you wrote to finish the assignment, think again, or better yet, go back, rewrite, and make the code as clear and concise as possible.

ASIDE: MORE REASON TO AVOID SPINNING: PRIORITY INVERSION

One good reason to avoid spin locks is performance: as described in the main text, if a thread is interrupted while holding a lock, other threads that use spin locks will spend a large amount of CPU time just waiting for the lock to become available. However, it turns out there is another interesting reason to avoid spin locks on some systems: correctness. The problem to be wary of is known as **priority inversion**, which unfortunately is an intergalactic scourge, occurring on Earth [M15] and Mars [R97]!

Let's assume there are two threads in a system. Thread 2 (T2) has a high scheduling priority, and Thread 1 (T1) has lower priority. In this example, let's assume that the CPU scheduler will always run T2 over T1, if indeed both are runnable; T1 only runs when T2 is not able to do so (e.g., when T2 is blocked on I/O).

Now, the problem. Assume T2 is blocked for some reason. So T1 runs, grabs a spin lock, and enters a critical section. T2 now becomes unblocked (perhaps because an I/O completed), and the CPU scheduler immediately schedules it (thus descheduling T1). T2 now tries to acquire the lock, and because it can't (T1 holds the lock), it just keeps spinning. Because the lock is a spin lock, T2 spins forever, and the system is hung.

Just avoiding the use of spin locks, unfortunately, does not avoid the problem of inversion (alas). Imagine three threads, T1, T2, and T3, with T3 at the highest priority, and T1 the lowest. Imagine now that T1 grabs a lock. T3 then starts, and because it is higher priority than T1, runs immediately (preempting T1). T3 tries to acquire the lock that T1 holds, but gets stuck waiting, because T1 still holds it. If T2 starts to run, it will have higher priority than T1, and thus it will run. T3, which is higher priority than T2, is stuck waiting for T1, which may never run now that T2 is running. Isn't it sad that the mighty T3 can't run, while lowly T2 controls the CPU? Having high priority just ain't what it used to be.

You can address the priority inversion problem in a number of ways. In the specific case where spin locks cause the problem, you can avoid using spin locks (described more below). More generally, a higher-priority thread waiting for a lower-priority thread can temporarily boost the lower thread's priority, thus enabling it to run and overcoming the inversion, a technique known as **priority inheritance**. A last solution is simplest: ensure all threads have the same priority.

Appendix

26.5 The Wish for Atomicity: Alternatives for atomicity

- ✓ 1. Make a super instruction

  Do `memory-add 0x8049a1c, $0x1`
  It could not be interrupted in the middle ? has run to completion or not run at all ? atomic operation (or indivisible operation)
  Not feasible: Too many case (memory-mul, list-update, B-tree update, …) ? need more generic form (Test-and-Set or Compare-and-Swap)

- ✓ 2. Disable interrupt

  Disable context switch
  Simple, but long interrupt disable may cause problems

- ✓ 3. Use Mutual exclusion APIs

  Lock, Semaphore, Conditional variable, …
  Based on HW atomic operations: Test-and-Set, Compare-and-Swap, …
  Will be discussed in chapter 28

Appendix

28.15 Different OS / 28.16 Two-Phase Locks

- ✓ Case study: Linux implementation for sleep lock
- ✓ Futex (fast userspace mutex)
  - A system call for basic locking or building higher-level locking abstraction
  - futex_wait(address, expected): put the calling thread to sleep if the value of the address is equal to the expected. Otherwise, return immediately
  - futex_wake(address): wake up one thread that sleeps in the queue related to the address
- ✓ Use case (gnu libc library)
  - Dual meaning: 1) highest bit in the mutex: to track the lock state (held or not), 2) other bits: to count the number of waiters
  - Fastpath: optimize common case ? if (no contention), only do atomic_bit_test for lock() and atomic_add_zero for unlock()

```
1   void mutex_lock (int *mutex) {
2     int v;
3     /* Bit 31 was clear, we got the mutex (this is the fastpath)   */
4     if (atomic_bit_test_set (mutex, 31) == 0)
5       return;
6     atomic_increment (mutex);
7     while (1) {
8         if (atomic_bit_test_set (mutex, 31) == 0) {
9             atomic_decrement (mutex);
10            return;
11        }
12        /* We have to wait now. First make sure the futex value
13           we are monitoring is truly negative (i.e. locked). */
14        v = *mutex;
15        if (v >= 0)
16          continue;
17        futex_wait (mutex, v);
18    }
19  }
20
21  void mutex_unlock (int *mutex) {
22    /* Adding 0x80000000 to the counter results in 0 if and only if
23       there are not other interested threads */
24    if (atomic_add_zero (mutex, 0x80000000))
25      return;
26
27    /* There are other threads waiting for this mutex,
28       wake one of them up.   */
29    futex_wake (mutex);
30  }
```

Figure 28.10: Linux-based Futex Locks

28.13 A Simple Approach: Just Yield

- ✓ Before sleeping, let us think about yield()
- ✓ Preempt and enter into the ready state instead of the sleep state

  De-schedule itself
  Give a chance for the lock-held thread to be scheduled
  Consideration
  - Two threads: probably work well
  - More than two threads (say 10): possibly run -> ready and ready -> run transition repeatedly ? sleep is better

```
1   void init() {
2       flag = 0;
3   }
4
5   void lock() {
6       while (TestAndSet(&flag, 1) == 1)
7           yield(); // give up the CPU
8   }
9
10  void unlock() {
11      flag = 0;
12  }
```

Figure 28.8: **Lock With Test-and-set And Yield**