OPERATING SYSTEM CONCEPTS

Chapter 6. Process Synchronization

A/Prof. Kai Dong

Warm-up

What is the Output?

```c
/* thread.c */
#include <stdio.h>
#include <stdlib.h>
#include <common.h>

volatile int counter = 0;
int loops;

void *worker(void *arg) {
 int i;
 for (i = 0; i < loops; i++) {
 counter++;
 }
 return NULL;
}
```

```c
int main(int argc, char *argv[]) {
 if (argc != 2) {
 fprintf(stderr, "usage: threads <
      value>\n");
 exit(1);
 }
 loops = atoi(argv[1]);
 pthread_t p1, p2;
 printf("Initial value : %d\n",
      counter);

 Pthread_create(&p1, NULL, worker,
       NULL);
 Pthread_create(&p2, NULL, worker,
       NULL);
 Pthread_join(p1, NULL);
 Pthread_join(p2, NULL);
 printf("Final value : %d\n",
      counter);
 return 0;
}
```

Warm-up

Concurrency

```
prompt> gcc -o thread thread.c -Wall -pthread
prompt> ./thread 1000
Initial value : 0
Final value : 2000

prompt> ./thread 100000
Initial value : 0
Final value : 143012
prompt> ./thread 100000
Initial value : 0
Final value : 137298
```

- **Concurrency** — Many problems arise, and must be addressed, when working on many things at once (i.e., concurrently) in the same program.

Objectives

- To present the concept of process synchronization.
- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem
- To examine several classical process-synchronization problems
- To explore several tools that are used to solve process synchronization problems

Contents

Contents

Background

- Processes (and threads) can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- In our warm-up example, what data is shared?

Background

Result Indeterminate

- One line of C code

```
counter ++
```

- is compiled as

```
mov 0x8049a1c, %eax      ; register1 = counter
add $0x1, %eax           ; register1 = register1 + 1
mov %eax, 0x8049a1c      ; counter = register1
```

- Consider this execution interleaving with *counter* $= 50$ initially:

$S0$:	p1 executes register1 = counter	(register1 = 50)
$S1$:	p1 executes register1 = register1 + 1	(register1 =51)
$S2$:	p2 executes register2 = counter	(register2 = 50)
$S3$:	p2 executes register2 = register2 + 1	(register2 = 51)
$S4$:	p1 executes counter = register1	(counter = 51)
$S5$:	p2 executes counter = register2	(counter = 51 NOT 52)

- Because of multi-processors? Not really.

Background

Uncontrolled Scheduling

OS	Thread 1	Thread 2	PC	%eax	counter
			(after instruction)		
	before critical section		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt					
save T2's state					
restore T1's state			108	51	50
	mov %eax, 0x8049a1c		113	51	51

- The heart of the problem: uncontrolled scheduling
- What if we had a super instruction "memory-add 0x8049a1c, $0x1"?

Background
Race Condition

- **Race condition**
  - Several processes (threads) access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place.
  - Result indeterminate

Contents

The Critical-Section Problem

- Consider system of n processes $P_0, P_1, \cdots, P_{n-1}$
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- **Critical section**
  - Multiple threads executing a segment of code, which can result in a **race condition**.
  - Question: what codes are in critical section in previous examples?
- **Critical section problem** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

The Critical-Section Problem

- General structure of process P_i

```
do {
        [entry section]
                [critical section]
        [exit section]
                [remainder section]
} while (true);
```

- An algorithm for process P_i

```
do {
        while (turn == j);
                critical section
        turn = j;
                remainder section
} while (true);
```

- Is it a correct solution?

The Critical-Section Problem

Solution to Critical-Section Problem

1. **Mutual Exclusion** — If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** — If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** — A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
   - Assume that each process executes at a nonzero speed
   - No assumption concerning relative speed of the n processes

The Critical-Section Problem

Critical Section Handling in OS

- Two approaches depending on if kernel is preemptive or non-preemptive
  - Preemptive — allows preemption of process when running in kernel mode
  - Non-preemptive — runs until exits kernel mode, blocks, or voluntarily yields CPU
    - Essentially free of race conditions in kernel mode
- Why would anyone favor a preemptive kernel over a non-preemptive one?
  - A preemptive kernel may be more responsive, since there is less risk that a kernel-mode process will run for an arbitrarily long period before relinquishing the processor to waiting processes.

Contents

Mutex Locks

Controlling Scheduling

- What we need: Hardware synchronization primitives.
  - **Lock**-related source codes are put around critical sections, thus ensure that any such critical section executes as if it were a single atomic instruction.
- A lock is just a variable.
- How to use a lock:
  - Declare a lock variable (e.g., *mutex*).
  - The lock variable holds the state of the lock.
  - It is either available (or unlocked or free), means that no thread holds the lock;
  - Or acquired (or locked or held), means that exactly one thread holds the lock.

Mutex Locks

Locks

- Algorithm for process P_i

```
do {
        lock();
                critical section
        unlock();
                remainder section
} while (true);
```

- The semantic of the *lock*() and *unlock*() routines?

Mutex Locks

Semantic of lock() and unlock() Routines

- The semantic of the *lock()* routines.
  - Calling the routine *lock*() tries to acquire the lock.
  - If no other thread holds the lock (i.e., it is free), the thread will acquire the lock and enter the critical section; this thread is sometimes said to be the owner of the lock.
  - If another thread then calls *lock*() on that same lock variable, it will not return since the lock is held by its owner.
  - Other threads are prevented from entering the critical section while the first thread that holds the lock is in there.
- The semantic of the *unlock()* routines.
  - Once the owner of the lock calls *unlock*(), the lock is now available (free) again.
  - If no other threads are waiting for the lock, the state of the lock is simply changed to free; otherwise, one of the waiting threads will notice this change of the lock's state, acquire the lock, and enter the critical section.

Mutex Locks

Pthread Locks

- Entry & Exit

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

- initialization

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
int rc = pthread_mutex_init(&lock, NULL);
assert(rc == 0);
```

- Destruction

```
pthread_mutex_destroy(&lock);
pthread_mutex_destroy(lock);
```

- Other versions

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

Mutex Locks

Building A Lock

- How can OS build an efficient lock?
  - It depends, on which hardware synchronization primitives are used.
- Ways of Building A Lock
  - Without special hardware support
    - » Dekker's and Peterson's Algorithms
    - » Lamport's Bakery Algorithm
  - With hardware support
    - » Controlling Interrupts
    - » The test-and-set instruction (atomic exchange)
    - » The compare-and-swap instruction (compare-and-exchange)
    - » Load-Linked and Store-Conditional
    - » Fetch-And-Add
- Discussion on spin-waiting

Mutex Locks

Controlling Interrupts

- For single-processor systems:
  - One of the earliest solutions used to provide mutual exclusion was to disable interrupts for critical sections.

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

  - *Disable/EnableInterrupts*() are implemented by using special hardware instructions.
- Question: Is it a solution to the critical section problem?
  - Whether or not it satisfies: Mutual exclusion? Progress? Bounded Waiting?
- And what about Performance?

Mutex Locks

Controlling Interrupts (contd.)

- What are the **negatives**?
- This approach requires us to allow any calling thread to perform a privileged operation (turning interrupts on/off), and trust this facility is not abused.
  - A greedy program could call lock() at the beginning of its execution and thus monopolize the processor; worse, an errant or malicious program could call lock() and go into an endless loop.
- This approach does not work on multiprocessors.
  - Threads will be able to run on other processors, and thus could enter the critical section.
- This approach may lost interrupts.
  - E.g., if the CPU missed the fact that a disk device has finished a read request. How will the OS know to wake the process waiting for said read?
- This approach can be inefficient.
  - Codes that mask or unmask interrupts are executed slowly.

Mutex Locks

Why Failed?

- A failed attempt — why failed?

```c
typedef struct __lock_t { int flags; } lock_t;

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

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;
}
```

Mutex Locks

Why Failed? (contd.)

Mutual exclusion	F
Progress	T
Bounded waiting	F

Thread 1	Thread 2
call *lock*()	
while (flag == 1)	
interrupt: switch to Thread 2	
	call *lock*()
	while (flag == 1)
	flag = 1
	critical section
	interrupt: switch to Thread 1
flag = 1	
critical section	

Mutex Locks

Peterson's Solution

- Good algorithmic description of solving the problem
- Two process solution
- Assume that the load and store machine-language instructions are atomic; that is, cannot be interrupted.
- The two processes share two variables:

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

- The variable *turn* indicates whose turn it is to enter the critical section
- The *flag* array is used to indicate if a process is ready to enter the critical section. $flag[i] = true$ implies that process P_i is ready!

Mutex Locks

Peterson's Solution (contd.)

Algorithm for Process P_i

```
do {
 flag[i] = true;
 turn = j;
 while (flag[j] && turn == j);
  critical section
 flag[i] = false;
 remainder section
} while (true);
```

Algorithm for Process P_j

```
do {
 flag[j] = true;
 turn = i;
 while (flag[i] && turn == i);
  critical section
 flag[j] = false;
 remainder section
} while (true);
```

- Prove that the algorithm satisfies all requirements for the critical section problem

Mutex Locks

Why Failed?

- Failed attempt #1

```
do {     while (flag[j]);
         flag[i] = true;
                 critical section
         flag[i] = false;
                 remainder section        } while (true);
```

- Failed attempt #2

```
do {     flag[i] = true;
         while (flag[j]);
                 critical section
         flag[i] = false;
                 remainder section        } while (true);
```

- Failed attempt #3

```
do {     victim = i;
         while (victim == i);
                 critical section
                 remainder section        } while (true);
```

Mutex Locks

Why Failed? (Failed attemp #1)

Mutual exclusion	F
Progress	T
Bounded waiting	F

Thread 1	Thread 2
while(flag[2]);	
interrupted: switch to Thread 2	
	while(flag[1]);
	flag[2] = true;
	critical section
	interrupted: switch to Thread 1
flag[1] = true;	
critical section	

Mutex Locks

Why Failed? (Failed attemp #2)

Mutual exclusion	T
Progress	F
Bounded waiting	T

Thread 1	**Thread 2**
	flag[2] = true;
	interrupted: switch to Thread 1
flag[1] = true;	
while (flag[2])	
blocked: switch to Thread 2	
	while (flag[1])
	blocked: switch to Thread 1
. . .	

Mutex Locks

Why Failed? (Failed attemp #3)

Mutual exclusion	T
Progress	F
Bounded waiting	T

Thread 1	**Thread 2**
remainder section	
(performing I/O)	
blocked: switch to Thread 2	
	victim = 2;
	while (victim == 2)
	blocked: switch to Thread 1
. . .	
(I/O interrupt)	
victim = 1;	
. . .	

Mutex Locks

Bakery Algorithm

- Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections.
- What if there are more than two cooperating processes?
- Bakery Algorithm — Critical section for *n* processes
- Lamport envisioned a bakery with a numbering machine at its entrance so each customer is given a unique number. Numbers increase by one as customers enter the store. A global counter displays the number of the customer that is currently being served. All other customers must wait in a queue until the baker finishes serving the current customer and the next number is displayed. When the customer is done shopping and has disposed of his or her number, the clerk increments the number, allowing the next customer to be served. That customer must draw another number from the numbering machine in order to shop again.

Mutex Locks

Bakery Algorithm

- Before entering its critical section, process (customer) receives a number. Holder of the smallest number enters the critical section. (wait in a queue)
- If processes P_i and P_j receive the same number, if $i < j$, then P_i is served first; else P_j is served first. (a unique number)
- The numbering scheme always generates numbers in increasing order of enumeration; e.g., 1, 2, 3, 3, 3, 3, 4, 5, $\cdots$
- Notation
  - $(a, b) < (c, d)$ if $a < c$ or if $a = c$ and $b < d$
  - $k = \max(a_0, \cdots, a_{n-1})$, i.e., $k \geq a_i, \forall i \in [0, n]$
- Shared data

```
boolean choosing[n];     // initialized to false
int number[n];   // initialized to 0
```

Mutex Locks

Bakery Algorithm (contd.)

```
do {
        choosing[i] = true;
        number[i] = max(number[0], number[1], ..., number [n - 1]) + 1;
        choosing[i] = false;
        for (j = 0; j < n; j ++) {
                while (choosing[j]);
                while ((number[j] != 0) && ((number[j], j) < (number[i], i)));
        }
                critical section
        number[i] = 0;
                remainder section
} while (true);
```

- What part are entry section and exit section?
- What codes are in *lock*() and *unlock*()?
- What is the use of *choosing*[]? (why fail without *choosing*[ ]?)

Mutex Locks

Bakery Algorithm (Why fail without choosing[ ]?)

Mutual exclusion	F
Progress	T
Bounded waiting	T

Thread i	**Thread j**
max(· · ·)	
interrupted: switch to Thread j	
	number[j] = max(· · ·)
	while ((number[i]!=0) && (· · ·))
	critical section
	interrupted: switch to Thread i
number[i] = max(· · ·)	
while ((· · ·) && ((number[i],i)<(number[j],j)))	
critical section	

Mutex Locks

Conclusion

- Developing locks that work without special hardware support became all the rage for a while, giving theory-types a lot of problems to work on.
- This line of work became quite useless when people realized it is much easier to assume a little hardware support.
- Further, algorithms like the ones above don't work on modern hardware (due to relaxed memory consistency models), thus making them even less useful than they were before.

Mutex Locks

Test And Set

- Hardware support: Some form of **test-and-set** instruction.

```
int TestAndSet(int *old_ptr, int new) {
        int old = *old_ptr;
        *old_ptr = new;
        return old;
}
```

- It returns the old value pointed to by the *ptr*, and simultaneously updates said value to *new*.
- The key is that this sequence of operations is performed atomically.
- Test-and-set enables you to *test* the old value (which is what is returned) while simultaneously *setting* the memory location to a new value.
- Question: Can you build a lock based on this instruction?

Mutex Locks

Test And Set (contd.)

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

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

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

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

- Evaluating this spin lock:
  - Mutual exclusion? Yes
  - Progress? Yes
  - Bounded waiting? No

Mutex Locks

Compare And Swap

- Hardware support: **compare-and-swap** atomic instruction
- To test whether the value at the address specified by *ptr* is equal to *expected*; if so, update the memory location pointed to by *ptr* with the *new* value. If not, do nothing.

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

- Question: Can you build a lock based on this instruction?

Mutex Locks

Compare And Swap (contd.)

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

- Evaluating this spin lock.
  - Mutual exclusion? Yes
  - Progress? Yes
  - Bounded waiting? No
- Compare-and-swap is more powerful than test-and-set. Can be used to achieve lock-free synchronization.

Mutex Locks

Compare And Swap (contd.)

- Lock-free synchronization
- How to implement a concurrent counter?
  - lock(); update counter; unlock();
- Discussion: Can you build a Class like *AtomicInteger*?

```
int CompareAndSwap(int *ptr, int expected, int new) {
        int actual = *ptr;
        if (actual == expected) {
         *ptr = new;
         return 1;
        }
        return 0;
}
```

Mutex Locks

Compare And Swap (contd.)

- An alternative approach that does not require explicit locking:

```
void AtomicIncrement(int *counter, int amount) {
        do {
                int old = *counter;
        } while (CompareAndSwap(counter, old, old+amount) == 0);
}
```

- Benefits: No deadlock can arise.
- We will detail deadlocks later in Chapter 7.

Mutex Locks

Load-Linked and Store-Conditional

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

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

void lock(lock_t *lock) {
        while (true) {
                while (LoadLinked(&lock->flag) == 1)
                        ;       // spin-waiting
                if (StoreConditional(&lock->flag, 1) == 1)
                        return;
        }
}

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

Mutex Locks

How to Satisfy Bounded Waiting?

```
/* C-like pseudo code */
Initially Boolean waiting[i] = false; lock = false;

lock() {
        waiting[i] = true;
        while (waiting[i] && (TestAndSet(lock, 1) == 1));
        waiting[i] = false;
}

unlock() {
        j = (i + 1) % n;
        while ((j != i) && !waiting[j])
                j = (j + 1) % n;
        if (j == i)
                lock = false;
        else
                waiting[j] = false;
}
```

Mutex Locks

Fetch And Add

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

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

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

Mutex Locks

Spin-Waiting

- When a thread waits to acquire a lock that is already held, it endlessly checks the value of flag, a technique known as spin-waiting.
- Hardware support for locks — Spin locks are simple and they work, and can be fair (as with the case of the ticket lock), but also can be quite inefficient.
- Think about N threads contending for a lock; $N - 1$ time slices may be wasted.
- How to solve this problem?
- Hints: some OS support like de-scheduling.

Mutex Locks

Yield

- Assuming an OS primitive *yield*().
- *yield*() is simply a system call that moves the caller from the running state to the ready state. Process de-schedules itself.

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

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

- Better than spin, but still *inefficient*.
- Think about N threads contending for a lock; $N - 1$ threads may execute the run-and-yield pattern.
- Starvation

Mutex Locks

Using Queues

- The real problem:
  - The scheduler determines which thread runs next.
  - Solution: exert some control over scheduling.
- A queue can be used to keep track of which threads are waiting to acquire the lock.
- E.g. two calls provided by Solaris.
  - *park*() puts a calling thread to sleep.
  - *unpark*(*threadID*) wakes a particular thread as designated by threadID.

Mutex Locks

park() and unpark()

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

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

```c
void_lock(lock_t *m) {
  while (TestAndSet(&m->guard, 1)
        == 1);
  if (m->flag == 0) {
  m-flag = 1;
  m->guard = 0;
  } else {
  queue_add(m->q, gettid());
  m->guard = 0;
  park();
  }
}

void unlock(lock_t *m) {
  while (TestAndSet(&m-guard, 1) ==
         1);
  if (queue_empty(m->q))
  m->flag = 0;
  else
  unpark(queue_remove(m->q));
  m->guard = 0;
}
```

Mutex Locks

setpark()

- Race condition before the call to *park*(): With just a wrong timing switch, the subsequent park by the first thread would then sleep forever (potentially).
- A third system call: *setpark*()
- A thread indicates it is about to park.
- If it then happens to be interrupted and another thread calls *unpark*() before *park*() is actually called, the subsequent *park*() returns immediately instead of sleeping.

```
queue_add(m->q, gettid());
setpark();
m->guard = 0;
```

- Is spin avoided? No, but the time spent spinning is quite limited.

Mutex Locks

Spin-Waiting (contd.)

- when can spin-waiting be useful?
  - No context switch is required
  - On multi processor systems, one thread can spin on one processor while another thread performs its critical section on another processor.
- Two-phase locks in Linux
  - In the first phase, the lock spins for a while, hoping that it can acquire the lock.
  - If the lock is not acquired during the first spin phase, a second phase is entered, where the caller is put to sleep, and only woken up when the lock becomes free later.

Mutex Locks

Spin-Waiting (contd.)

- Correctness reason to avoid spinning: **priority inversion** — A higher-priority thread waiting for a lock held by lower-priority thread.
- If the lock is a spin lock, the higher priority thread spins forever, and the system is hung.
- With more threads and priority levels, the problem becomes more complicated.
  - Imagine three threads, T_1, T_2, and T_3, with T_3 at the highest priority, and T_1 the lowest.
  - T_1 grabs a lock.
  - T_3 starts and tries to acquire the lock that T1 holds, and gets stuck waiting.
  - T_2 starts. Now T_3, which is at higher priority than T_2, is stuck waiting for T_1, which may never run now that T_2 is running.
- **priority inheritance** — 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.

Contents

Locked Data Structures

- How to add locks to a data structure to make it usable by threads makes the structure thread safe.
- Consider both correctness and Performance
  - Concurrent Counter
  - Concurrent Linked List
  - Concurrent Queue
  - Concurrent Hash Table

Locked Data Structures

Non-concurrent Counter

```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;
}
```

Locked Data Structures

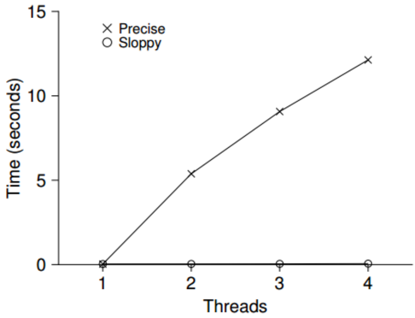
Concurrent Counter

```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;
}
```

Locked Data Structures

Concurrent Counter (contd.)

- Performance is still a problem
  - A single thread completes in 0.03 seconds, where two threads complete in more than 5 seconds!
- Sloppy counter



Locked Data Structures

Sloppy Counter

- local counter + local lock
- global counter + global lock

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)

Locked Data Structures

Sloppy Counter (contd.)

```c
typedef struct __counter_t {
 int global;
 pthread_mutex_t glock;
 int local[NUMCPUS];
 pthread_mutex_t llock[NUMCPUS];
 int threshold;
} counter_t;

void init(counter_t *c, int threshold) {
 c->threshold = threshold;
 c->global = 0;
 Pthread_mutex_init(&c->glock, NULL);
 for (int i = 0; i < NUMCPUS; i++) {
 c->local[i] = 0;
 pthread_mutex_init(&c->llock[i], NULL);
 }
}
```

```c
void update(counter_t *c, int threadID, int amt) {
 int cpu = threadID % NUMCPUS;
 pthread_mutex_lock(&c->llock[cpu]);
 c->local[cpu] += amt;
 if (c->local[cpu] >= c->threshold) {
 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]);
}

int get(counter_t *c) {
 pthread_mutex_lock(&c->glock);
 int rc = c->global;
 pthread_mutex_unlock(&c->glock);
 return rc;
}
```

Locked Data Structures

A Simple Concurrent Linked List

```c
typedef struct __node_t {
        int key;
        struct __node_t *next;
} node_t;

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);
}
```

Locked Data Structures

A Simple Concurrent Linked List (contd.)

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

Locked Data Structures

Scaling Linked Lists

- Hand-over-hand locking (a.k.a. lock coupling)
- Instead of having a single lock for the entire list, you instead add a lock per node of the list.
- When traversing the list, the code first grabs the next node's lock and then releases the current node's lock
- High degree of concurrency in list operations. In practice, it is hard to make such a structure faster than the simple single lock approach. (Surprise? Guess why.)
- A hybrid would be worth investigating.

Locked Data Structures

Concurrent Queue

- Instead of adding a big lock, any approach more concurrently?

```
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);
}
```

Locked Data Structures

Concurrent Queue (contd.)

```
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;
        }
        *value = newHead->value;
        q->head = newHead;
        pthread_mutex_unlock(&q->headLock);
        free(tmp);
        return 0;
}
```

Locked Data Structures

Concurrent Hash Table

- Instead of adding a big lock, any approach more concurrently?

```
#define BUCKETS (101)

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

void Hash_Init(hash_t *h) {
        for (int 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[buckets], key);
}

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

Contents

Condition Variables

Implementing pthread_join()

- Two problems:

```c
void *child(void *arg) {
        printf("child\n");
        // Problem #1: how to indicate we are done?
        return NULL;
}

int main(int argc, char *argv[]) {
        printf("parent: begin\n");
        pthread_t c;
        pthread_create(&c, NULL, child, NULL);
        // Problem #2: how to wait for child?
        printf("parent: end\n");
        return 0;
}
```

Condition Variables

Implementing pthread_join() (contd.)

- Using a shared variable:

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

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

  - Correct, but
  - Waste of CPU time — spin-waiting.
  - Do we need a lock here? Not a critical section.

Condition Variables

Implementing pthread_join() (contd.)

- A condition variable is an explicit queue
- The POSIX calls:
  - *wait*(): put itself to sleep;
  - *signal*(): wake a sleeping thread waiting on this condition.

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

Condition Variables

Implementing pthread_join() (contd.)

```c
int done = 0;
pthread_mutex_m =
        PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c =
        PTHREAD_COND_INITIALIZER;

void thr_exit() {
 pthread_mutex_lock(&m);
 done = 1;
 pthread_cond_signal(&c);
 pthread_mutex_unlock(&m);
}

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

```c
void thr_join() {
 pthread_mutex_lock(&m);
 while (done == 0)
 pthread_cond_wait(&c, &m);
 pthread_mutex_unlock(&m);
}

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

Condition Variables

Implementing pthread_join() (contd.)

- Condition variable is always with a lock and a flag
- Why are the following codes broken?

```c
void thr_exit() {
 pthread_mutex_lock(&m);
 pthread_cond_signal(&c);
 pthread_mutex_unlock(&m);
}

void thr_join() {
 pthread_mutex_lock(&m);
 pthread_cond_wait(&c, &m);
 pthread_mutex_unlock(&m);
}

/* broken code #1 */
```

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


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


/* broken code #2 */
```

Condition Variables

The Producer-Consumer Problem

```c
/* Let's begin with only 1 producer, 1 consumer, buffer_size = 1 */

cond_t cond;
mutex_t mutex
void *producer(void *arg) {
        for (int i = 0; i < loops; i ++) {
                pthread_mutex_lock(&mutex);
                if (count == 1)
                        pthread_cond_wait(&cond, &mutex);
                put();                      // produce
                pthread_cond_signal(&cond);
                pthread_mutex_unlock(&mutex);
        }
}
void *consumer(void *arg) {
        for (int i = 0; i < loops; i ++) {
                pthread_mutex_lock(&mutex);
                if (count == 0)
                        pthread_cond_wait(&cond, &mutex);
                get();                      // consume
                pthread_cond_signal(&cond);
                pthread_mutex_unlock(&mutex);
        }
}
```

Condition Variables

The Producer-Consumer Problem (contd.)

- Problematic with more threads

T_{c1}	State	T_{c2}	State	T_p	State	**Count**	*Comment*
lock	Running		Ready		Ready	0	
wait	Waiting		Ready		Ready	0	Nothing to get
	Waiting		Ready	*lock*	Running	0	
	Waiting		Ready	*put*	Running	1	Buffer now full
	Ready		Ready	*signal*	Running	1	T_{c1} awoken
	Ready		Ready	*unlock*	Running	1	
	Ready		Ready	*lock*	Running	1	
	Ready		Ready	*wait*	Waiting	1	Buffer full
	Ready	*lock*	Running		Waiting	1	T_{c2} sneaks in
	Ready	*get*	Running		Waiting	0	T_{c2} grabs data
	Ready	*signal*	Running		Ready	0	$T_p$awoken
	Ready	*unlock*	Running		Ready	0	
get	Running		Ready		Ready	0	No data!

Condition Variables

The Producer-Consumer Problem (contd.)

```
/* Still broken */

cond_t cond;
mutex_t mutex
void *producer(void *arg) {
        for (int i = 0; i < loops; i ++) {
                pthread_mutex_lock(&mutex);
                while (count == 1)      // use "while" instead of "if"
                        pthread_cond_wait(&cond, &mutex);
                put();                  // produce
                pthread_cond_signal(&cond);
                pthread_mutex_unlock(&mutex);
        }
}
void *consumer(void *arg) {
        for (int i = 0; i < loops; i ++) {
                pthread_mutex_lock(&mutex);
                while (count == 0)      // use "while" instead of "if"
                        pthread_cond_wait(&cond, &mutex);
                get();                  // consume
                pthread_cond_signal(&cond);
                pthread_mutex_unlock(&mutex);
        }
}
```

Condition Variables

The Producer-Consumer Problem (contd.)

T_{c1}	State	T_{c2}	State	T_p	State	**Count**	*Comment*
lock	Running		Ready		Ready	0	
wait	Waiting		Ready		Ready	0	Nothing to get
	Waiting	*lock*	Running		Ready	0	
	Waiting	*wait*	Waiting		Ready	0	Nothing to get
	Waiting		Waiting	*lock*	Running	0	
	Waiting		Waiting	*put*	Running	1	Buffer now full
	Ready		Waiting	*signal*	Running	1	T_{c1} awoken
	Ready		Waiting	*unlock*	Running	1	
	Ready		Waiting	*lock*	Running	1	
	Ready		Waiting	*wait*	Waiting	1	Buffer full
while	Running		Waiting		Waiting	1	Recheck condition
get	Running		Waiting		Waiting	0	T_{c1} grabs data
signal	Running		Ready		Waiting	0	Oops, T_{c2} awoken
unlock	Running		Ready		Waiting	0	
lock	Running		Ready		Waiting	0	
wait	Waiting		Ready		Waiting	0	Nothing to get
	Waiting	*while*	Ready		Waiting	0	Recheck condition
	Waiting	*wait*	Waiting		Waiting	0	Everyone waiting!

Condition Variables

The Producer-Consumer Problem (contd.)

```
/* buffer_size = 1 */

cond_t empty, fill;                  // two condition variables
mutex_t mutex
void *producer(void *arg) {
        for (int i = 0; i < loops; i ++) {
                pthread_mutex_lock(&mutex);
                while (count == 1)
                        pthread_cond_wait(&empty, &mutex);
                put();                   // produce
                pthread_cond_signal(&fill);
                pthread_mutex_unlock(&mutex);
        }
}
void *consumer(void *arg) {
        for (int i = 0; i < loops; i ++) {
                pthread_mutex_lock(&mutex);
                while (count == 0)
                        pthread_cond_wait(&fill, &mutex);
                get();                   // consume
                pthread_cond_signal(&empty);
                pthread_mutex_unlock(&mutex);
        }
}
```

Condition Variables

The Producer-Consumer Problem (contd.)

```
/* dealing with bounded buffer instead of single buffer */

cond_t empty, fill;                    // still two condition variables
mutex_t mutex
void *producer(void *arg) {
        for (int i = 0; i < loops; i ++) {
                pthread_mutex_lock(&mutex);
                while (count == MAX)   // remember: always use "while"
                        pthread_cond_wait(&empty, &mutex);
                put();                 // produce
                pthread_cond_signal(&fill);
                pthread_mutex_unlock(&mutex);
        }
}
void *consumer(void *arg) {
        for (int i = 0; i < loops; i ++) {
                pthread_mutex_lock(&mutex);
                while (count == 0)
                        pthread_cond_wait(&fill, &mutex);
                get();                 // consume
                pthread_cond_signal(&empty);
                pthread_mutex_unlock(&mutex);
        }
}
```

Condition Variables

Covering Condition

- Consider a memory allocation scenario:
  - Assume there are 0 bytes free;
  - Thread T_a calls *allocate*(100)
  - Thread T_b calls *allocate*(10)
  - Thread T_c calls *free*(50)
- Which thread to signal?
  - What if T_a is awoken?
- *pthread_cond_broadcast*(), to wake up all waiting threads.

Contents

Semaphores

- A semaphore is an object with an integer value that we can manipulate with two routines: *wait*() and *signal*().
  - In the POSIX standard, these routines are *sem_wait*() and *sem_post*().
  - Historically, Dijkstra uses *P*() and *V*(). *P*() comes from “prolaag”, a contraction of “probeer” (Dutch for “try”) and “verlaag” (“decrease”); *V*() comes from the Dutch word “verhoog” which means “increase”.

```
wait(s) {
        while (s <= 0) ;
        s --;
}

signal(s) {
        s ++;
}
```

```
wait(s) {
        s --;
        while (s < 0) ;
}

signal(s) {
        s ++;
}
```

Semaphores

Semaphore Implementation

$$Semaphore = Mutex + Integer + ConditionVariable$$

```c
typedef struct {
        int value;
        struct process *list;
} semaphore;

wait(semaphore *s) {
        s->value --;
        if (s->value < 0) {
                add this process to s->list;
                block();
        }
}

signal(semaphore *s) {
        s->value ++;
        if (s->value <= 0) {
                remove a process P from s->list;
                wakeup(P);
        }
}
```

Semaphores

Why Semaphores

- Functionality
  - A single primitive for all things related to synchronization
  - Both locks and condition variables
- Correctness and Convenience
  - Avoid errors — can *signal*() first then *wait*().
  - Clean and organized — making it easy to demonstrate their correctness.
  - Efficient.

Semaphores

Basic Synchronization Patterns

- Signaling
- Rendezvous
- Mutex
- Multiplex
- Barrier
- Reusable barrier
- Pairing

Semaphores

Signaling

```
/* Thread A */
statement a;
```

```
/* Thread B */
statement b;
```

- How to guarantee that a happens before b?

```
/* Thread A */
statement a;
signal(s);
```

```
/* Thread B */
wait(s);
statement b;
```

- What should semaphore *s* initially be?

Semaphores

Rendezvous

```
/* Thread A */
statement a1;
statement a2;
```

```
/* Thread B */
statement b1;
statement b2;
```

- How to guarantee that a1 happens before b2, and b1 before a2?

```
/* Thread A */
statement a1;
signal(a);
wait(b);
statement a2;
```

```
/* Thread B */
statement b1;
signal(b);
wait(a);
statement b2;
```

- What should semaphores *a*, *b* initially be?

Semaphores

Rendezvous (contd.)

- Is it correct? — Yes, but probably less efficient.

```
/* Thread A */
statement a1;
signal(a);
wait(b);
statement a2;
```

```
/* Thread B */
statement b1;
wait(a);
signal(b);
statement b2;
```

- Is it correct? — No, deadlock.

```
/* Thread A */
statement a1;
wait(b);
signal(a);
statement a2;
```

```
/* Thread B */
statement b1;
wait(a);
signal(b);
statement b2;
```

Semaphores

Mutex

```
/* Thread A */
count = count + 1;
```

```
/* Thread B */
count = count + 1;
```

- Add semaphores to the following example to enforce mutual exclusion to the shared variable *count*.

```
/* Thread A */
wait(mutex);
count = count + 1;
signal(mutex);
```

```
/* Thread B */
wait(mutex);
count = count + 1;
signal(mutex);
```

- What should semaphores *mutex* initially be?

Semaphores

Multiplex

- Generalize the previous solution so that it allows multiple threads to run in the critical section at the same time, but it enforces an upper limit (*MAX*) on the number of concurrent threads.

```
wait(multiplex);
count = count + 1;        // critical section
signal(multiplex);
```

- What should semaphores *mutex* initially be?

Semaphores

Barrier

- Generalize the rendezvous solution. *n* threads should run the following code:

```
rendezvous;
critical point;
```

- How to ensure that no thread executes critical point until after all threads have executed rendezvous?

```
/* initialization */
int count = 0;
semaphore mutex = 1;
semaphore barrier = 0;
```

```
rendezvous;
wait(mutex);
count = count + 1;
signal(mutex);
if (count == n)
        signal(barrier);
wait(barrier);
signal(barrier);
critical point;
```

Semaphores

Barrier (contd.)

```
/* Bad barrier solution */
rendezvous;
wait(mutex);
count = count + 1;
if (count == n)
        signal(barrier);
wait(barrier);
signal(barrier);
signal(mutex);
critical point;
```

- Common source of deadlocks: blocking on a semaphore while holding a mutex.

Semaphores

Reusable Barrier (contd.)

```
semaphore barrier1 =0, barrier2 = 0, mutex =1;

rendezvous;
wait(mutex);
count += 1;
if (count == n)
        for (int i = 0; i < n; i ++)
                signal(barrier1);
signal(mutex);
wait(barrier1);
critical point;
wait(mutex);
count -= 1;
if (count == 0)
        for (int i = 0; i < n; i ++)
                signal(barrier2);
signal(mutex);
wait(barrier2);
```

Semaphores

Pairing

- Imagine that threads represent ballroom dancers and that two kinds of dancers, leaders and followers, wait in two queues before entering the dance floor. When a leader arrives, it checks to see if there is a follower waiting. If so, they can both proceed. Otherwise it waits. Similarly, when a follower arrives, it checks for a leader and either proceeds or waits, accordingly.

```
semaphore leader = 0, follower = 0;
```

```
/* leader */
signal(leader);
wait(follower);
dance();
```

```
/* follower */
signal(follower);
wait(leader);
dance();
```

- How to ensure *dance*() are executed in pairs?

Semaphores

Pairing (contd.)

```
/* initialization */
int num_l = 0, num_f = 0;
semaphore leader = 0, follower = 0, pairing = 0, mutex = 1;
```

```
/* leader */
wait(mutex);
if (num_f > 0) {
        num_f--;
        signal(leader);
}
else {
        num_l++;
        signal(mutex);
        wait(follower);
}
dance();
wait(pairing);
signal(mutex);
```

```
/* follower */
wait(mutex);
if (num_l > 0) {
        num_l--;
        signal(follower);
}
else {
        num_f++;
        signal(mutex);
        wait(leader);
}
dance();
signal(pairing);
        // no signal(mutex);
```

Semaphore

The Producer-Consumer Problem

```
semaphore empty = MAX, full = 0, mutex = 1;

void *producer(void *args) {
        for (int i = 0; i < loops; i ++) {
                wait(empty);
                wait(mutex);
                put();
                signal(mutex);
                signal(full);
        }
}

void *consumer(void *args) {
        for (int i = 0; i < loops; i ++) {
                wait(full);
                wait(mutex);
                get();
                signal(mutex);
                signal(empty);
        }
}
```

Semaphore

The Dining Philosophers

- There are five “philosophers” sitting around a table. Between each pair of philosophers is a single chopstick (and thus, five total). The philosophers each have times where they think, and don’t need any chopsticks, and times where they eat. In order to eat, a philosopher needs two chopsticks, both the one on their left and the one on their right. The basic loop of each philosopher is as follows, assuming each has a unique identifier $p \in [0, 4]$:

```
while (true) {
        think();
        getchopsticks();
        eat();
        putchopsticks();
}
```

Semaphore

The Dining Philosophers (contd.)

```
/* a failed solution, why failed? */

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

void putchopsticks() {
        signal(chopsticks[left(p)]);
        signal(chopsticks[right(p)]);
}

void getchopsticks() {
        wait(chopsticks[left(p)]);
        wait(chopsticks[right(p)]);
}
```

Semaphore

The Dining Philosophers (contd.)

```c
/* a correct solution */

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

void putchopsticks() {
        signal(chopsticks[left(p)]);
        signal(chopsticks[right(p)]);
}

void getchopsticks() {
        if (p == 4) {
                wait(chopsticks[right(p)]);
                wait(chopsticks[left(p)]);
        }
        else {
                wait(chopsticks[left(p)]);
                wait(chopsticks[right(p)]);
        }
}
```

Semaphore

The Readers-Writers Problem

- Imagine a number of concurrent operations, including reads and writes.
  - Writes change the state of the data
  - Reads do not — Many reads can proceed concurrently, as long as we can guarantee that no write is on-going.
- The first readers–writers problem — requires that, no reader be kept waiting unless a writer has already obtained permission to use the shared object.
- The second readers-writers problem — requires that, once a writer is ready, that writer perform its write as soon as possible.

Semaphore

The First Readers-Writers Problem

```
semaphore rw_mutex = 1;
semaphore mutex = 1;
int read_count = 0;
```

```
void write() {
 do {
  wait(rw_mutex);
  /* writing */
  signal(rw_mutex);
  }
  while (true);
}
```

```
void read() {
 do {
  wait(mutex);
  read_count ++;
  if (read_count == 1)
  wait(rw_mutex);
  signal(mutex);
  /* reading */
  wait(mutex);
  read_count --;
  if (read_count == 0)
  signal(rw_mutex);
  signal(mutex);
  }
  while (true);
}
```

Semaphore

The No-starve Readers-Writers Problem

```
semaphore mutex= 1;
semaphore writemutex=1;
int read_count = 0;
semaphore wfmutex =1;
```

```
void write() {
 do {
 wait(wfmutex);
 wait(writemutex);
 /* writing */
 signal(writemutex);
 signal(wfmutex);
 }
 while (true);
}
```

```
void read() {
 do {
 wait(wfmutex);
 signal(wfmutex);
 wait(mutex);
 read_count ++;
 if (read_count == 1)
 wait(writemutex);
 signal(mutex);
 /* reading */
 wait(mutex);
 read_count --;
 if (read_count == 0)
 signal(writemutex);
 signal(mutex);
 }
 while (true);
}
```

Semaphore

The Second Readers-Writers Problem

```
semaphore readcount_mutex= 1;
semaphore writemutex=1;
int write_count = read_count = 0;
semaphore writecount_mutex= 1;
semaphore readmutex=1;
```

```
void write() {
 do {
 wait(writecount_mutex);
 write_count ++;
 if (write_count == 1)
 wait(readmutex);
 signal(writecount_mutex);
 wait(writemutex);
 /* writing */
 signal(writemutex);
 wait(writecount_mutex);
 write_count --;
 if (write_count == 0)
 signal(readmutex);
 signal(writecount_mutex);
 }
 while (true);
}
```

```
void read() {
 do {
 wait(readmutex);
 wait(readcount_mutex);
 read_count ++;
 if (read_count == 1)
 wait(writemutex);
 signal(readcount_mutex);
 signal(readmutex);
 /* reading */
 wait(readcount_mutex);
 read_count --;
 if (read_count == 0)
 signal(writemutex);
 signal(readcount_mutex);
 }
 while (true);
}
```

Semaphores

Why and Why Not Semaphores

- Why semaphores?
  - Avoid errors/Clean and organized/Efficient.
  - Synchronization between Processes.
  - How about performance? Better than you can imagine.
- Why not semaphores?
  - Signaling all (broadcast) in CV.
  - Priority inheritance in mutex locks.

Contents

Monitors

- Why **monitors**?
  - As object-oriented programming was gaining ground, people started to think about ways to merge synchronization into a more structured programming environment.
- With a monitor class — the monitor guarantees that only one thread can be active within the monitor at a time.
- A Java Monitor — add the keyword synchronized to the method or set of methods that you wish to use as a monitor

```
public class SynchronizaedCounter {
        private int c = 0;
        public synchronized void increment() {
                c ++;
        }
        public synchronized void decrement() {
                c --;
        }
        public synchronized int value() {
                return c;
        }
}
```

Monitors

Hoare Vs. Mesa

- Hoare Semantics:
- The *signal*() immediately wakes one waiting thread.
- Guess which one is more popular?
- Mesa semantics:
- The *signal*() move a single waiting thread to ready state.

Monitors

Hoare Vs. Mesa (contd.)

```
/* correct with Hoare semantic, but incorrect with Mesa semantic */
monitor class BoundedBuffer {
private:int buffer[MAX];
        int fill, use, fullEntires = 0;
        cond_t empty, full;
public: void produce(int element) {
                if (fullEntires == MAX)
                // correct with Mesa changing "if" to "while"
                        wait(&empty);
                buffer[fill] = element;
                fill = (fill + 1) % MAX;
                fullEntries ++;
                signal(&full);
        }
        int consume() {
                if (fullEntries == 0)
                // correct with Mesa changing "if" to "while"
                        wait(&full);
                int tmp = buffer[use];
                use = (use + 1) % MAX;
                fullEntries --;
                signal(&empty);
                return tmp;
        }
}
```