Advanced Operating Systems: Three Easy Pieces

2. Concurrency

Outline

- Concurrency: Introduction
- Interlude (intervening): Threads APIs
- Locks

- Lock-based Concurrent Data Structures
- Condition Variables

- Semaphores
- Common Concurrency Problems including Deadlock
- Advanced Concurrency - Event-based

Concurrency: Introduction

Thread

- A new abstraction for a single running process

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

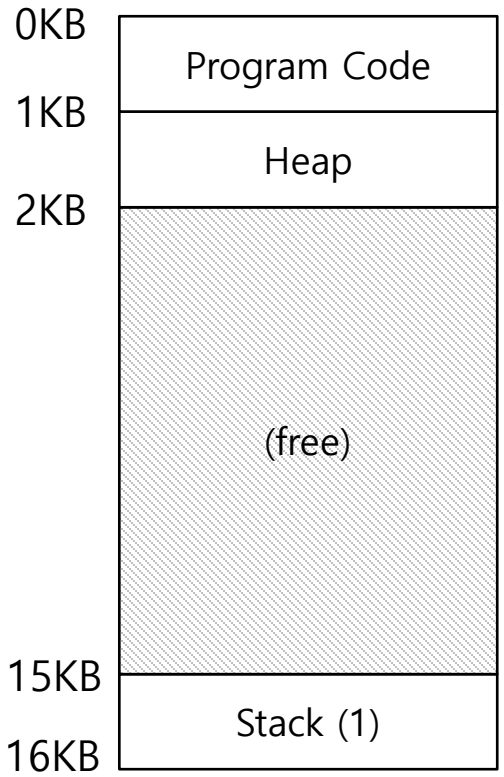
Context switch between threads

- **Each thread** has its own program counter and set of registers including SP:
  - One **thread control blocks (TCBs)** is needed to store the state for each thread.

- **Single CPU:** when switching from running one thread (T1) to running another thread (T2):
  - The registers state of T1 be saved.
  - The registers state of T2 restored.
  - The address space remains the same.

The stack of the relevant thread

- There will be one stack per thread, but threads share the heap

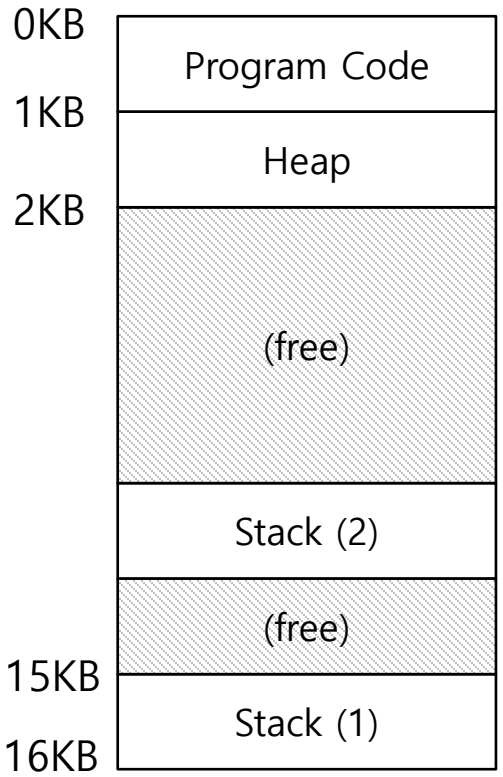


The code segment: where instructions live

The heap segment: contains malloc'd data dynamic data structures (it grows downward)

(it grows upward)
The stack segment: contains local variables arguments to routines, return values, etc.

A Single-Threaded Address Space



Two threaded Address Space

Race condition

- **Example with two threads:**
  - balance = balance + 1 (initial is 50)
  - We expect the result is 52, however…

- 0x8049a1c is the current PC (PC address)
- eax is a register

OS	Thread1	Thread2	PC (after instruction)	%eax	Balance (shared-memory)	
	before critical section		100	0	50	
	mov 0x8049a1c, %eax		105	50	50	
	add $0x1, %eax		108	**51**	**50**	**Did not store the register in the balance yet**
interrupt						
save T1's state						
restore T2's state			100	0	50	
		mov 0x8049a1c, %eax	105	50	50	**Did store the register in the balance**
		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**	**Problem**

Critical section

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

Locks

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

```
1    lock_t mutex;
2    . . .
3    lock(&mutex);
4    balance = balance + 1;
5    unlock(&mutex);
```

Critical section

Interlude*: Threads APIs

* **Interlude**: a break between threads accessing shared data

Thread Creation

- **How to create and control threads?**

```
#include <pthread.h>

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

- **thread:** handle used to interact with this thread (output argument)
- **attr:** Used to specify any attributes this thread might have:
  - Stack size, Scheduling priority, …
- **start_routine**: the function this thread starts running in.
- **arg:** the argument to be passed to the function (start routine)
  - *a void pointer* allows us to pass in *any type of* argument.

Thread Creation (Cont.)

- If start_routine instead required another type argument, the declaration would look like this:
  - **An integer argument**:

```
int
pthread_create(…, // first two args are the same
                  void*  (*start_routine)(int),
                  int    arg);
```

  - **Return an integer**:

```
int
pthread_create(…, // first two args are the same
                  int  (*start_routine)(void*),
                  void*    arg);
```

Example: Creating a Thread

```
#include <pthread.h>

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

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

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

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

```
Int  pthread_create (pthread_t*  t,
     pthread_attr_t*    attr,
     void * (*start-routine) (void*),
     void * arg);
```

Wait for a thread to complete

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

- **thread:** Specify which thread *to wait for*
- **value_ptr:** A pointer to the return value
  - Because **pthread_join**() routine changes the value arg, you need to pass-in a pointer to that value.
- **Pthread_join returns** 0 on success or error number on error

Example: Waiting for Thread Completion

```
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) {                    // arg is an input value
17      myarg_t *m = (myarg_t *) arg;
18      printf("%d %d\n", m->a, m→b);
19      myret_t  *r = malloc(sizeof(myret_t));     // allocated from the Heap
20      r→x = 1;
21      r→y = 2;
22      // on return r is freed but we return the addr of the malloc() returned address
23      return (void *) r;
24  }
```

Example: Waiting for Thread Completion (Cont.)

```
25  int main(int argc, char *argv[]) {
26      int rc;
27      pthread_t p;
28      myret_t *m;
29
30      myarg_t args;
31      args.a = 10;
32      args.b = 20;
33      pthread_create(&p, NULL, mythread, &args);
34      pthread_join(p, (void **) &m);   // - this thread has been
                                         // waiting inside of the
                                         // pthread_join() routine.
                                         // - m is the return value from the
                                         // other thread
35      printf("returned %d %d\n", m->x, m->y);   // print 1, 2.
36      return 0;
37  }
```

Example: Dangerous code

- Be careful with how values are returned from a thread.

```
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;                 // BAD! We need (myret_t *r)
                                    // - address of address
5       r.x = 1;
6       r.y = 2;
7       return (void *) &r;         // (void **) &m
8   }                               // m == *r
```

- When the variable **r** returns, it is automatically de-allocated.
- In the above Example, the created thread allocated the memory for returned value.
- Remember **r** should be **value_ptr; on return r is freed but we are passing pointer to the value address – as a result loosing **r** is not an issue. In other words, m in pthread_join() will be set to content of r which is (r = myret_t *), i.e., m is modified on return and that is why it is passed by address and not by value. Once m is set to content of r, loosing r is OK

Example: Simpler Argument Passing to a Thread

- **Just passing in a single value**: allocate memory for the returned value in **main**() and then only pass the returned value address.

```
1   void *mythread(void *arg) {
2       int m = (int) arg;
3       printf("%d\n", m);
4       return (void *) (arg + 1);
5   }
6
7   int main(int argc, char *argv[]) {
8       pthread_t p;
9       int rc, m;
10      pthread_create(&p, NULL, mythread, (void *) 100);
11      pthread_join(p, (void **) &m);          // m = arg + 1
12      printf("returned %d\n", m);
13      return 0;
14  }
```

Locks

- **Provide mutual exclusion** to a critical section
  - **Interface**

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

  - **Usage** (w/o *lock initialization* and *error check*)

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_lock(&lock);      // get the lock
x = x + 1;                      // or whatever your critical section is
pthread_mutex_unlock(&lock);    // release the lock
```

    - If no other thread holds the lock → the thread will acquire the lock and enter the critical section.
    - If another thread holds the lock → the thread will not return from the call until it has acquired the lock.

Locks (Cont.)

- **All locks must be properly initialized:**
  - **One way:** using **PTHREAD_MUTEX_INITIALIZER**

```
pthread_mutex_t  lock = PTHREAD_MUTEX_INITIALIZER;
```

  - **The dynamic way:** using **pthread_mutex_init**()

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

Locks (Cont.)

- **Check errors** code when calling lock and unlock:
  - **An example wrapper**

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

- **These two calls** are used in lock acquisition:

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

  - **trylock**: return failure if the lock is already held
  - **timelock:** return after a timeout or after acquiring the lock

Condition Variables

- **Condition variables** are useful when some kind of signaling must take place between threads.

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

  - **pthread_cond_wait:** wait for a condition to be true
    - Put the calling thread to sleep.
    - Wait for some other thread to signal you that cond is now TRUE.
  - **pthread_cond_signal**:
    - Unblock at least one (one or all) of the threads that are blocked on the condition variable

Condition Variables (Cont.)

- **A thread calling wait routine:**

```
int  initialized = 0;          // variable set by calling signal thread and tested by waiting thread
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  init = PTHREAD_COND_INITIALIZER;

pthread_mutex_lock(&lock);     // get the lock to test initialized
while (initialized == 0)       // cond is false
       pthread_cond_wait(&init, &lock);   // release lock  & wait for
                                          // cond to be true, and then get the lock back
pthread_mutex_unlock(&lock);   // release the lock
```

  - The **cond _wait()** call releases the lock before putting the caller to sleep.
  - Before returning after being woken, the **cond wait()** call re-acquire the lock.

- **A thread calling signal routine:**

```
pthread_mutex_lock(&lock);     // get the lock to set initialized
initialized = 1;               // set cond to true
pthread_cond_signal(&init);    // wake up a thread waiting for cond to be true
pthread_mutex_unlock(&lock);   // release the lock
```

Condition Variables (Cont.)

- The waiting thread **re-checks** the condition in a while loop, instead of a simple if statement.

```
int  initialized = 0;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t init = PTHREAD_COND_INITIALIZER;

pthread_mutex_lock(&lock);
while (initialized == 0)
        pthread_cond_wait(&init, &lock);
pthread_mutex_unlock(&lock);
```

  - Without rechecking (you might be awaked by some signal), the waiting thread will think that the condition has changed *even though it has not*.

Condition Variables (Cont.)

- **Don't ever do this:**
  - **A thread calling wait routine:**

```
while(initialized == 0)
        ;                               // spin
```

  - **A thread calling signal routine:**

```
initialized = 1;
```

  - It performs poorly in many cases → just wastes CPU cycles.
  - It is error prone.

Compiling and Running

- To compile the application code, you must include the header pthread.h:
  - Explicitly link with the pthreads library, by adding the –pthread flag. By adding the –Wall flag, we enable all warning messages.

```
prompt> gcc –o main main.c –Wall -pthread
```

  - For more information…

```
man –k pthread
```

Locks

Locks: The Basic Idea

- Ensure that any **critical section** executes as if it were a single atomic instruction.
  - **An example**: the canonical update of a shared variable

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

  - Add some code around the critical section

```
1     lock_t mutex; // some globally-allocated lock 'mutex'
2     …
3     lock(&mutex);
4     balance = balance + 1;
5     unlock(&mutex);
```

  - **Lock variable** holds the lock state: **available** (lock is free) and **acquired** (lock is held – lock can be held only by single thread).

The semantics of the lock()

- **lock()**
  - **Try to** acquire the lock.
  - If no other thread holds the lock, the thread will **acquire** the lock.
  - **Enter** the *critical section*.
    - This thread is said to be the owner of the lock.
  - **Other threads** are *prevented from* entering the critical section while the first thread that holds the lock is still in there.

Pthread Locks - mutex

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

```
1    pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
2
3    Pthread_mutex_lock(&lock);         // acquire the lock
4    balance = balance + 1;
5    Pthread_mutex_unlock(&lock);        // release the lock
```

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

Building A Lock

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

Evaluating locks – Basic criteria

- **Hardware support**
  - Building a lock needs some help from the hardware & OS
- **Mutual exclusion**
  - Does the lock work, preventing multiple threads from entering *a critical section*?
- **Fairness**
  - Does each thread contending for the lock gets a fair shot at acquiring it once it is free? (Starvation!)
- **Performance**
  - The time overheads added by using (acquire/release) the lock

Controlling Interrupts

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

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

  - **Problem:**
    - Require **too much *trust* in applications** for proper functioning
      - Greedy (or malicious) program could monopolize the CPU.
    - Do not work on **multiprocessors**
    - Code that masks or unmasks interrupts be executed in the kernel (expensive context switch)

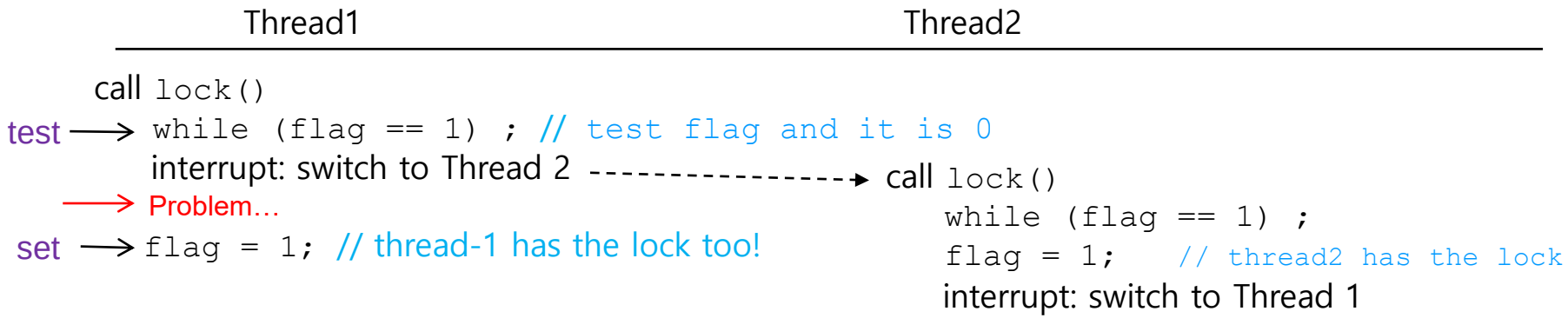
Why hardware support is needed?

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

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

Why hardware support needed? (Cont.)

- **Problem 1**: No Mutual Exclusion (testing & setting flag is not atomic)



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

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

(i) Test And Set (Atomic Exchange)

- **An instruction** to support the creation of simple locks:

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

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

A Simple Spin Lock using test-and-set

```
1    typedef struct __lock_t {          // lock
2        int flag;
3    } lock_t;
4
5    void init(lock_t *lock) {
6        // 0 indicates that lock is available,
7        // 1 that it is held
8        lock→flag = 0;
9    }
10
11   void lock(lock_t *lock) {
12       while (TestAndSet(&lock→flag, 1) == 1)
13               ;          // spin-wait
14   }
15
16   void unlock(lock_t *lock) {
17       lock→flag = 0;
18   }
```

- **Note**: To work correctly on *a single processor*, it requires a preemptive scheduler.

Evaluating Spin Locks

- **Correctness**: yes
  - The spin lock only allows a single thread to enter the critical section.
- **Fairness**: no
  - Spin locks don't provide any fairness guarantees.
  - Indeed, a thread spinning may spin *forever*
- **Performance**:
  - **In the single CPU**, performance overheads can be quite *painful*.
  - **If the number of threads roughly equals the number of CPUs**, spin locks work *reasonably well*.

(ii) Compare-And-Swap (IBM)

- Test whether the lock value at the address(ptr) is equal to expected (i.e., free): if yes (lock is free) set the lock to the new value (busy). If not (lock was busy), leave things as is.
  - *In either case*, **return** the **actual** (old) value of the lock.

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

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

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

Spin lock with compare-and-swap (C-style)

(iii) Load-Linked and Store-Conditional (LL/SC)

https://en.wikipedia.org/wiki/Load-link/store-conditional

```
1    int LoadLinked(int *ptr) {
2         return *ptr;                    // return current value == 0
3    }
4
5    // store new value only if no update happened to that memory location since the LoadLink
6   int StoreConditional(int *ptr, int value) {
7         if (no one has updated *ptr since the LoadLinked to this address) {
8                   *ptr = value;
9                   return 1;             // success!
10        } else {
11                  return 0;             // failed to update
12        }
13   }
```

Load-linked And Store-conditional

- The **store-conditional** *only succeeds* if **no intermittent store** to the address has taken place.
  - **success**: return 1 and update (*ptr) to value.
  - **fail**: the value at ptr is not updated and 0 is returned.
- Together, implements a lock-free atomic **read-modify-write** operation (read + write new value to the same loc atomically)

Load-Linked and Store-Conditional (Cont.)

```
1    void lock(lock_t *lock) {
2        while (1) {
3                while (LoadLinked(&lock→flag) == 1)  // lock is busy
4                        ;   // spin until it's zero
5                if (StoreConditional(&lock→flag, 1) == 1) // 1 = success
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   }
```

Using LL/SC To Build A Lock

```
1    void lock(lock_t *lock) {
2        while (LoadLinked(&lock→flag) || !StoreConditional(&lock->flag, 1))
3                ; // spin
4    }
```

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

(iv) Fetch-And-Add

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

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

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

(v) Ticket Lock

Ticket = turn = n-1

get Lock:
ticket++
ticket = n, turn = n-1

Unlock:
turn++
turn == ticket = n

- **Ticket lock** can be built with fetch-and-add:
  - Ensure progress for all threads. → fairness
  - **Get lock()** increments ticket. Unlock() increments turn
  - Lock is free if turn = ticket…..

```
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); // ticket = 1, myturn = 0
13         while (lock→turn != myturn)        // old ticket ≠ turn (lock is held)
14                   ;          // spin
15   }
16   void unlock(lock_t *lock) {              // turn = ticket - 1
17         FetchAndAdd(&lock→turn);           // increment turn
18   }
```

Lock is held:
myturn (current ticket) = 1
turn = 0
➔ spin

Unlock:
myturn = Turn = 1
➔ out of spinning

So Much Spinning

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

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

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

A Simple Approach: Just Yield

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

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

Using Queues:
Sleeping Instead of Spinning

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

Using Queues: Sleeping Instead of Spinning

```
1    typedef struct __lock_t { int flag; int guard; queue_t *q; } lock_t;
2
3    void lock_init(lock_t *m) {
4        m→flag = 0;          // set flag that the lock is free
5        m→guard = 0;         // lock to protect flag field
6        queue_init(m->q);    // empty queue
7    }
8
9    void lock(lock_t *m) {
10       while (TestAndSet(&m→guard, 1) == 1)
11           ;                    // acquire guard lock by spinning
12       if (m→flag == 0) {   // we have guard + check if flag (lock) is free
13           m→flag = 1;      // lock is acquired
14           m→guard = 0;     // reset/release guard lock to be 0
15       } else {             // somebody else has the lock
16           queue_add(m→q, gettid());
17           m→guard = 0;     // reset guard (release lock) to be 0
18           park();          // this thread goes to sleep till the lock is free
19       }
20   }
21   …
```

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

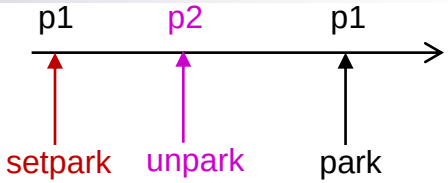
Using Queues:
Sleeping Instead of Spinning

```
22  void unlock(lock_t *m) {
23      while (TestAndSet(&m→guard, 1) == 1)
24          ;                                  // acquire guard lock by spinning
25      if (queue_empty(m→q))                  // nobody is waiting for the lock
26          m→flag = 0;                        // let go of lock; no one wants it
27      else                                   // Queue is not empty
28          unpark(queue_remove(m→q)); // wakeup next thread in the queue

29      m→guard = 0;                           // release guard
30  }
```

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

Wakeup/waiting race



- In case of releasing the lock (*thread A*) just before thread B call to park() → Thread B would sleep forever (potentially) ← race condition.
- **Solaris** solves this problem by adding a third system call: **setpark**().
  - By calling this routine, a thread can indicate it *is about to* park.
  - If it happens to be interrupted and another thread calls **unpark**() before **park**() is actually called, the subsequent **park**() returns immediately instead of sleeping.

```
1          queue_add(m→q, gettid());
2          setpark();          // new code
3          m→guard = 0;        // release guard
4          park();             // return and not sleep
```

Code modification inside of `lock()`

Two-Phase Locks

Spin for a while followed by sleep

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

- Concurrent counters:
  * Sloppy Counters
- Concurrent Linked List
- Concurrent Hash Table

Lock-based Concurrent Data Structures

Lock-based Concurrent Data structure

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

Example: Concurrent Counters without Locks

- Simple but not safe/scalable

```
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++;              /* no locking */
11      }
12
13      void decrement(counter_t *c) {
14              c→value--;              /* no locking */
15      }
16
17      int get(counter_t *c) {
18              return c→value;
19      }
```

Example: Concurrent Counters with Locks

- **Add a single lock**:
  - The lock is acquired when calling a routine that manipulates the data structure.

```
1       typedef struct __counter_t {
2               int value;
3               pthread_lock_t  lock;
4       } counter_t;
5
6       void init(counter_t *c) {
7               c→value = 0;
8               Pthread_mutex_init(&c→lock, NULL);   // use default attr
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
```

Example: Concurrent Counters with Locks (Cont.)

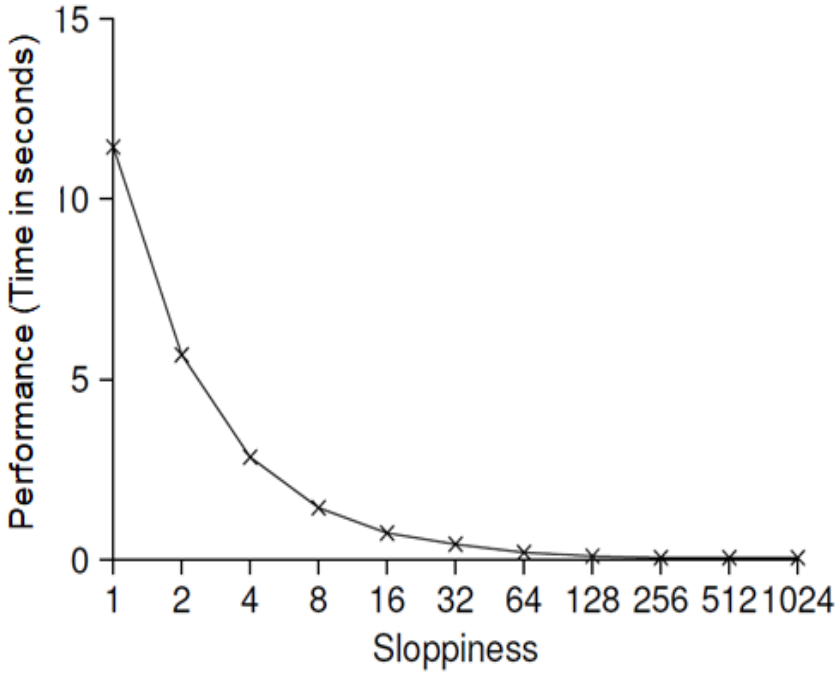
```
(Cont.)
17      void decrement(counter_t *c) {
18              Pthread_mutex_lock(&c->lock);
19              c->value--;
20              Pthread_mutex_unlock(&c->lock);
21      }
22
23      int get(counter_t *c) {
24              Pthread_mutex_lock(&c->lock);
25              int rc = c->value;
26              Pthread_mutex_unlock(&c->lock);
27              return rc;
28      }
```

Sloppy (Concurrent) counter

- **The sloppy counter** works by representing …
  - A single **logical counter** via numerous local physical counters, one per CPU core
  - A single **global counter**
  - There are **locks**:
    - One for each local counter and one for the global counter
- **Example:** on a machine with four CPUs:
  - Four local counters + four local locks
  - One global counter + one global lock
  - Each CPU updates its local counter (own lock) and periodically (every **S** time interval) local counter is propagated to the global counter (using the global lock) ← counter updates are scalable but less accurate
  - **S** time is larger → more efficient but global value is less accurate

Importance of the threshold value *S* *(S: time interval)*

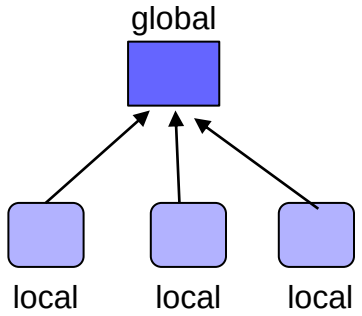
- **Each of the four threads** increment a counter 1 million times on four CPUs system:
- **S:** Sloppiness update time interval
  - **Low S** → Performance is **poor** (high frequent updates) The global count is always quite **accurate**.
  - **High S** → Performance is **excellent (infrequent updates)**, count **lags/inaccurate**.
- **Goal** is to reduce contention on the global lock in shared memory system.



Scaling Sloppy Counters (increase sloppiness reduce time)

Sloppy Counter Implementation

```
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 many local locks
6           int threshold;                      // update frequency counter
7       } counter_t;        // update global when local count reaches threshold
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
14          c→global = 0;
15          pthread_mutex_init(&c→glock, NULL);
16
17          int i;
18          for (i = 0; i < NUMCPUS; i++) {
19              c→local[i] = 0;
20              pthread_mutex_init(&c→llock[i], NULL);
21          }
22      }
23
```



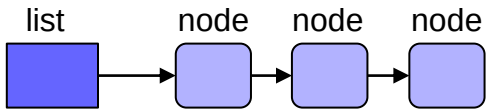
Sloppy Counter Implementation (Cont.)

(Cont.)

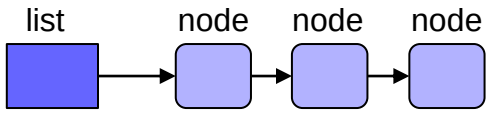
```
24      // update: usually, just grab local lock and update local amount
25      //         once local count has risen by 'threshold', grab global
26      //         lock and transfer/add local values to the global lock
27      void update(counter_t *c, int threadID, int amt) {
28          pthread_mutex_lock(&c→llock[threadID]);
29          c→local[threadID] += amt;                    // assumes amt > 0
30          if (c→local[threadID] >= c→threshold) { // transfer to global
31              pthread_mutex_lock(&c→glock);
32              c→global += c→local[threadID];       // increment global
33              pthread_mutex_unlock(&c→glock);
34              c→local[threadID] = 0;               // local counter = 0
35          }
36          pthread_mutex_unlock(&c->llock[threadID]);
37      }
38
39      // get: just return global amount (which may not be perfect)
40      int get(counter_t *c) {
41          pthread_mutex_lock(&c→glock);
42          int val = c→global;
43          pthread_mutex_unlock(&c→glock);
44          return val;    // only approximate!
45      }
```

Concurrent Linked Lists

```
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 lock is 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
(Cont.)
```



Concurrent Linked Lists (Contd.)




```
(Cont.)
18        int List_Insert(list_t *L, int key) {   // insert node at list front
19                pthread_mutex_lock(&L→lock);              // List 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        }
(Cont.)
```

Concurrent Linked Lists (Contd.)

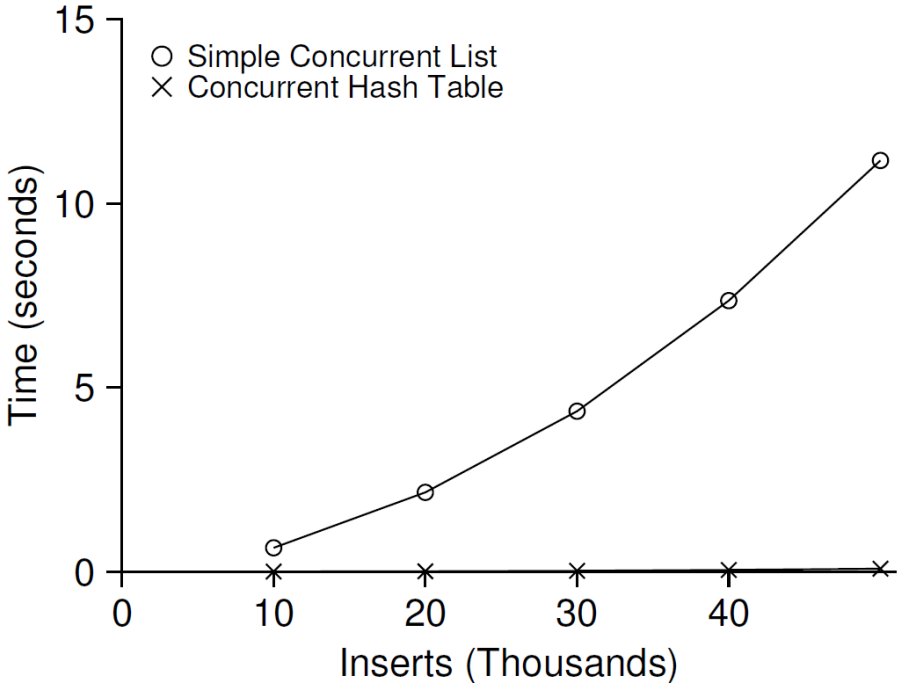
```
(Cont.)
32
32      int List_Lookup(list_t *L, int key) {
33              pthread_mutex_lock(&L→lock);
34              node_t *curr = L→head;
35              while (curr) {
36                      if (curr→key == key) {
37                              pthread_mutex_unlock(&L→lock);
38                              return 0; // success
39                      }
40                      curr = curr→next;
41              }
42              pthread_mutex_unlock(&L→lock);
43              return -1;                  // failure
44      }
```

Concurrent Linked Lists (Contd.)

- **The code acquires a lock** in the insert routine upon entry and release upon exit.
  - **Scaling Solution:**
    - The **lock and release** *only surround* the actual critical section in the insert code
    - Add a **lock per node** of the linked list instead of having a single lock for the entire list.

Concurrent Hash Table

- **Focus on a simple hash table** (Not Dynamic Hash Table):
  - The hash table does not resize.
  - Built using the concurrent lists; each bucket is a list
  - It uses a lock per hash bucket each of which is represented by *a list*.
  - **Performance**: From 10,000 to 50,000 concurrent updates from each of four threads.



The simple concurrent hash table scales magnificently.

Condition Variables

Condition Variables

- There are many cases where a thread wishes to check whether a **condition** is true before continuing its execution.
- **Example:**
  - A parent thread might wish to check whether a child thread has *completed*; this is often called a join().
- **Condition variable**
  - **Waiting** on the condition
    - **An explicit queue** that threads can put themselves on waiting for the condition to happen.
  - **Signaling** on the condition
    - **Some other thread**, *when it turns the condition to be true*, can wake-up one of those waiting threads and allow them to continue/resume execution.

Definition and Routines

- **Declare condition variable**

```
Pthread_cond_t c;
```

  - **Proper initialization is required**

    pthread_cond_t c = PTHREAD_COND_INITIALIZER;

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

- **Operation (the POSIX calls)**
  - **The wait() call** takes a mutex as a parameter:
    - Caller need to acquire the lock/mutex to be able to read the condition state. If the condition is not true, the wait() call releases the lock and put the calling thread to sleep.
    - When the thread wakes up, it must re-acquire the lock before returning from sleep.

The Producer / Consumer (Bound Buffer) Problem

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

The Put and Get Routines (Version 1)

```
1       int buffer;                             // one element
2       int count = 0;                          // initially, empty
3
4       void put(int value) {
5               assert(count == 0);
6               count = 1;
7               buffer = value;
8       }
9
10      int get() {
11              assert(count == 1);             // buffer is full
12              count = 0;
13              return buffer;
14      }
```

- **Only put data** into the buffer when count is zero (buff is free/empty).
  - i.e., when the buffer is *empty*.
- **Only get data** from the buffer when count is one (buff is busy/full).
  - i.e., when the buffer is *full*.

Producer/Consumer: Single CV and If Statement

- **A single condition variable** cond and associated lock mutex

```
1         cond_t cond;
2         mutex_t mutex;
3
4         void *producer(void *arg) {                    // *arg is loops
5             int i;
6             for (i = 0; i < loops; i++) {
7                 Pthread_mutex_lock(&mutex);                  // p1
8                 while (count == 1)                           // p2
9                     Pthread_cond_wait(&cond, &mutex);        // p3
10                put(i);                                      // p4
11                Pthread_cond_signal(&cond);                  // p5
12                Pthread_mutex_unlock(&mutex);                // p6
13            }
14
          }
15
```

Producer/Consumer: Single CV and If Statement

```
(Cont.)
16      void *consumer(void *arg) {                    // *arg is loops
17          int i;
18          for (i = 0; i < loops; i++) {
19              Pthread_mutex_lock(&mutex);                 // c1
20              while (count == 0)                          // c2
21                  Pthread_cond_wait(&cond, &mutex);       // c3
22              int tmp = get();                            // c4
23              Pthread_cond_signal(&cond);                 // c5
24              Pthread_mutex_unlock(&mutex);               // c6
25              printf("%d\n", tmp);
26          }
27      }
```

- A simple rule to remember with condition variables is to **always use while loops**.
- **However, this code still has a bug:** consumer should not wakeup other consumers, only wakeup producers, and vice versa, i.e., problem is both producer and consumer use the same CV.

The single Buffer Producer/Consumer Solution

- Use **two** condition variables (empty, fill) and while
  - **Producer** threads wait on the condition **empty**, and signals **fill** (not empty) condition to consumers
  - **Consumer** threads wait on **fill** condition and signals **empty** condition to producers.

```
1         cond_t empty, fill;
2         mutex_t mutex;
3
4         void *producer(void *arg) {
5             int i;
6             for (i = 0; i < loops; i++) {
7                 Pthread_mutex_lock(&mutex);
8                 while (count == 1)
9                     Pthread_cond_wait(&empty, &mutex)     // empty is set by consumer
10                put(i);
11                Pthread_cond_signal(&fill);
12                Pthread_mutex_unlock(&mutex);
13            }
14        }
15
```

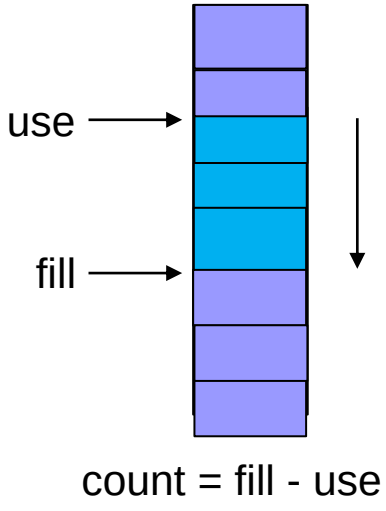
The single Buffer Producer/Consumer Solution

```
(Cont.)
16      void *consumer(void *arg) {
17          int i;
18          for (i = 0; i < loops; i++) {
19              Pthread_mutex_lock(&mutex);
20              while (count == 0)
21                  Pthread_cond_wait(&fill, &mutex); // fill is set by producer
22              int tmp = get();
23              Pthread_cond_signal(&empty);
24              Pthread_mutex_unlock(&mutex);
25              printf("%d\n", tmp);
26          }
27      }
```

The Final Producer/Consumer Solution

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

```
1         # define MAX  10
2         int buffer[MAX];
3         int fill = 0;
4         int use = 0;
5         int count = 0;
6
7         void put(int value) {
8             buffer[fill] = value;
9             fill = (fill + 1) % MAX;
10            count++;
11        }
12
13        int get() {
14            int tmp = buffer[use];
15            use = (use + 1) % MAX;
16            count--;
17            return tmp;
18        }
```



The Final Producer/Consumer Solution (Cont.)

```
1         cond_t empty, fill;
2         mutex_t mutex;
3
4         void *producer(void *arg) {              // *arg is loops
5             int i;
6             for (i = 0; i < loops; i++) {
7                 Pthread_mutex_lock(&mutex);              // p1
8                 while (count == MAX)                     // p2
9                     Pthread_cond_wait(&empty, &mutex);   // p3
10                put(i);                                  // p4
11                Pthread_cond_signal(&fill);              // p5
12                Pthread_mutex_unlock(&mutex);            // p6
13            }
14        }
15
16        void *consumer(void *arg) {              // *arg is loops
17            int i;
18            for (i = 0; i < loops; i++) {
19                Pthread_mutex_lock(&mutex);              // c1
20                while (count == 0)                       // c2
21                    Pthread_cond_wait(&fill, &mutex);    // c3
22                int tmp = get();                         // c4
```

The Final Producer/Consumer Solution (Cont.)

```
(Cont.)
23                Pthread_cond_signal(&empty);                    // c5
24                Pthread_mutex_unlock(&mutex);                   // c6
25                printf("%d\n", tmp);
26            }
27        }
```

The Final Working Solution (Cont.)

- **p2: A producer** only sleeps if all buffers are currently filled.
- **c2: A consumer** only sleeps if all buffers are currently empty.

Semaphores

Semaphore: Why

- **Semaphore** is a lock that is built based on mutex. It is more flexible but more expensive:
  - **It is meant to be used between different processes on the same OS** while mutexes are meant to be used between thread within a single process.
  - **A semaphore can be deleted by another thread other than the one that create it**, while mutex can only be deleted by the thread which create it.

Semaphore: A definition

- Is an object with an integer value (semaphore value)
  - We can manipulate a semaphore with two routines; sem_wait() and sem_post().
  - Initialization

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

    - Declare a semaphore **s** and initialize it to the value 1 (free).
    - The second argument, 0, indicates that the semaphore is shared between *threads in the same process* (i.e., not shared between processes)
    - The third argument is the initial value assigned to the semaphore
    - Return value = 0 means success, and -1 means failure

Semaphore: Interact with semaphore

- **sem_wait()**

```
1  int sem_wait(sem_t *s) {              // Sem value = 1 is free
2       // decrement the value of semaphore s by one
3       // wait if value of semaphore s ≤ 0
4  }
```

- Semaphore value = 1 means it is free
- Decrement (lock) the semaphore value by 1
- If semaphore value > 0, i.e., sem is free, decrement the sem value and **return right away with caller having the lock**.
- If sem value is 0, the **caller blocks** (suspend execution) and go to sleep waiting for a subsequent post; sem value = -1
- When negative, the value of the semaphore is equal to the number of waiting threads asking for the semaphore.
- Return value = 0 on success, and -1 on failure

Semaphore: Interact with semaphore (Cont.)

- `sem_post()`

```
1  int sem_post(sem_t *s) {
2       // increment the value of semaphore s by one
3       // if there are one or more threads waiting, wake up one
4  }
```

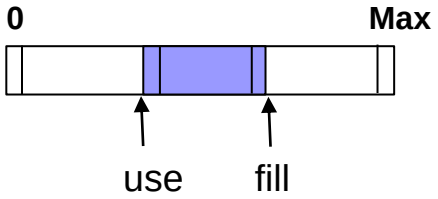
- Simply **increments** the value of the semaphore.
  - If the value after increment > 0 then there are no waiting threads
  - If the value after increment = 0 then wake up a waiting thread and give them the semaphore
- If there is multiple threads waiting to be woken, **wakes** one of them and give them the semaphore.
- Return value = 0 on success, and -1 on failure

Binary Semaphores (Locks)

- What should **X** (i.e., Sem value) be?
  - The initial value should be **1 (free)**.

```
1    sem_t m;
2    sem_init(&m, 0, X); // initialize semaphore to X; what should X be?
3
4    sem_wait(&m);
5    //critical section here
6    sem_post(&m);
```

The Producer/Consumer (Bounded-Buffer) Problem



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

```
1   int buffer[MAX];
2   int fill = 0;                          // pointer to putting
3   int use = 0;                           // pointer to getting
4
5   void put(int value) {
6         buffer[fill] = value;            // line f1
7         fill = (fill + 1) % MAX;         // line f2
8   }
9
10  int get() {
11        int tmp = buffer[use];           // line g1
12        use = (use + 1) % MAX;           // line g2
13        return tmp;
14  }
```

- Filling the buffer & incrementing the index into the buffer is a critical section – multiple producers is a race condition

A Solution: Adding Mutual Exclusion

```
1    sem_t empty;                          // buffer is empty
2    sem_t full;                           // buffer is full
3    sem_t mutex;                          // protect buffer[max] entries
4
5    void *producer(void *arg) {
6        int i;
7        for (i = 0; i < loops; i++) {
8            sem_wait(&mutex);             // line p0 (NEW LINE)
9            sem_wait(&empty);             // line p1 - block waiting for consumer
10           put(i);                       // line p2
11           sem_post(&full);              // line p3 - wakeup consumer
12           sem_post(&mutex);             // line p4 (NEW LINE)
13       }
14   }
15
(Cont.)
```

Adding Mutual Exclusion (Incorrectly – hold-and-wait)

A Solution: Adding Mutual Exclusion

```
(Cont.)
16  void *consumer(void *arg) {
17      int i;
18      for (i = 0; i < loops; i++) {
19          sem_wait(&mutex);          // line c0 (NEW LINE)
20          sem_wait(&full);           // line c1 - wait till buff is full
21          int tmp = get();           // line c2
22          sem_post(&empty);          // line c3 - wakeup producer
23          sem_post(&mutex);          // line c4 (NEW LINE)
24          printf("%d\n", tmp);
25      }
26  }
```

Adding Mutual Exclusion (Incorrectly)

- **The above code can lead to a deadlock ↓**
  - **Hold-and-wait:** The consumer **acquire** the mutex (line c0). The consumer **calls** sem_wait() on the full semaphore (line c1). The consumer is **blocked** waiting for the producer and **yield** the CPU (the consumer still holds the mutex!)
  - The producer **calls** sem_wait() on the binary mutex semaphore (line p0) ➔ deadlock

Finally, A Working Solution

```
1    sem_t empty;
2    sem_t full;
3    sem_t mutex;
4
5    void *producer(void *arg) {
6        int i;
7        for (i = 0; i < loops; i++) {
8                sem_wait(&empty);          // line p1 (no hold-and-wait)
9                sem_wait(&mutex);          // line p1.5 (MOVED MUTEX HERE…)
10               put(i);                    // line p2
11               sem_post(&mutex);          // line p2.5 (… AND HERE)
12               sem_post(&full);           // line p3
13       }
14   }
15
(Cont.)
```

Adding Mutual Exclusion (Correctly)
On return from sem_wait(&empty) the mutex was released by the consumer

Finally, A Working Solution

```
(Cont.)
16   void *consumer(void *arg) {
17       int i;
18       for (i = 0; i < loops; i++) {
19               sem_wait(&full);          // line c1
20               sem_wait(&mutex);         // line c1.5 (MOVED MUTEX HERE…)
21               int tmp = get();          // line c2
22               sem_post(&mutex);         // line c2.5 (… AND HERE)
23               sem_post(&empty);         // line c3
24               printf("%d\n", tmp);
25       }
26   }
27
28   int main(int argc, char *argv[]) {
29       // …
30       sem_init(&empty, 0, MAX); // MAX entries are empty to begin with …
31       sem_init(&full, 0, 0);    // ... and 0 entries are full
32       sem_init(&mutex, 0, 1);   // mutex=1 means the lock is free
33       // …
34   }
```

Adding Mutual Exclusion (Correctly)

Reader-Writer Locks

- **Problem:** semaphore is a **lock** that does not take advantage of the operation semantics to be performed on the semaphore, i.e., to allow multiple readers but only one writer
- **Imagine a number** of concurrent list operations, including **inserts** and simple **lookups**.
  - **insert:**
    - Change the state of the list
    - A traditional critical section makes sense.
  - **lookup:**
    - Simply *read* the data structure.
    - As long as we can guarantee that no insert is on-going, we can allow many lookups to proceed **concurrently**.

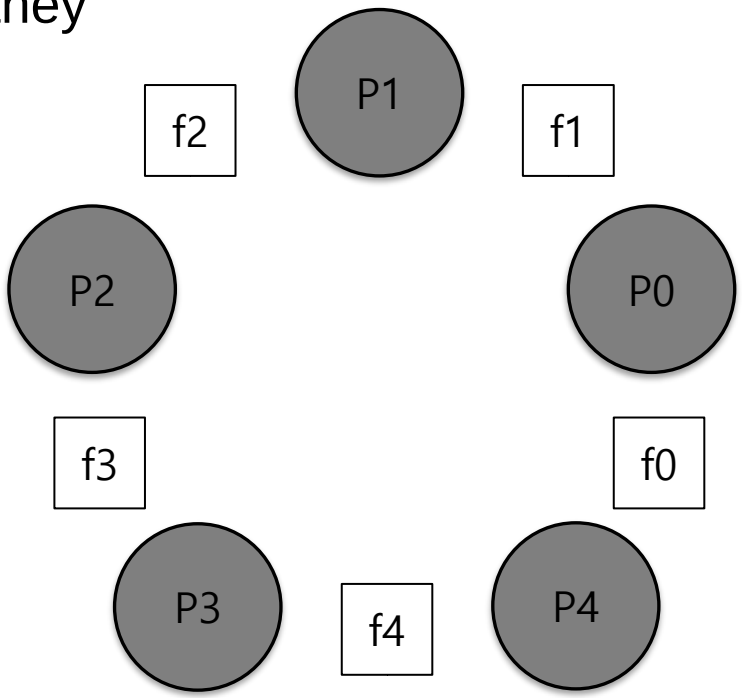
This special type of lock is known as a reader-write lock.

A Reader-Writer Locks (Cont.)

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

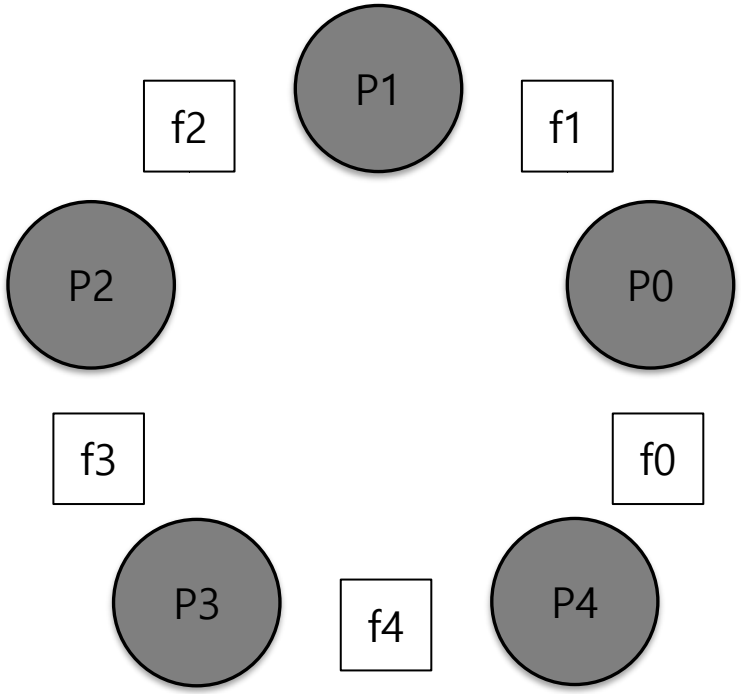
The Dining Philosophers

- **Assume there are five "philosophers"** sitting around a table:
  - Between each pair of philosophers is a single fork (five total).
  - The philosophers each have times where they **think** (and don't need any forks), and times where they **eat** (and need 2 forks).
  - In order to *eat*, a philosopher needs **two forks**, both the one on their *left* and the one on their *right*.
  - **There is contention for these forks.**
  - Clearly you can have deadlock



The Dining Philosophers

- **Solution:**
  - **All philosophers try to get left fork then right fork in this order:**
    - You acquire the lock for the left fork with wait
    - After getting left fork lock, you acquire the right fork lock with no-wait
    - If was able to get the left fork lock and could not get right fork lock, you need to release the left fork and try later, i.e., after some random delay.



Common Concurrency Problems including Deadlock

Common Concurrency Problems

- More recent work focuses on studying other types of **common concurrency bugs**:
  - **Two major types of non deadlock bugs:**
    - Atomicity violation
    - Order violation
  - **Focus on four major open-source applications:**
    - MySQL, Apache, Mozilla, OpenOffice.

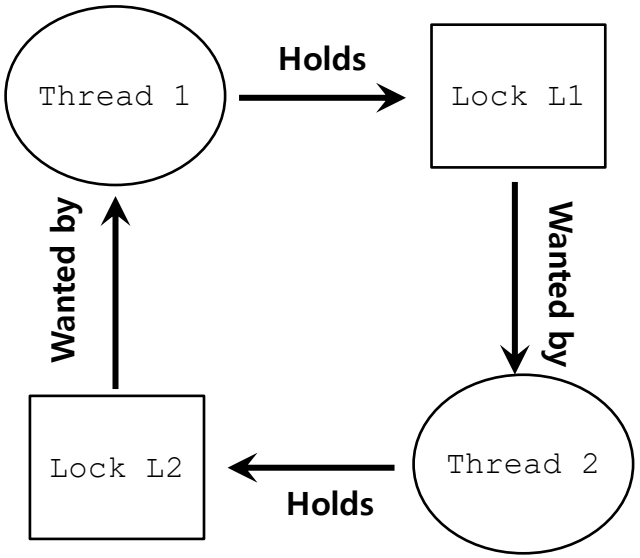
Application	What it does	Non-Deadlock	Deadlock
MySQL	Database Server	14	9
Apache	Web Server	13	4
Mozilla	Web Browser	41	16
Open Office	Office Suite	6	2
Total		**74**	**31**

Bugs In Modern Applications

Deadlock Bugs

Thread 1:	Thread 2:
`lock(L1);`	`lock(L2);`
`lock(L2);`	`lock(L1);`

- **The presence of a cycle**
  - Thread1 is holding a lock L1 and waiting for another one, L2.
  - Thread2 that holds lock L2 is waiting for L1 to be release.



Conditions for Deadlock

- **Four conditions** need to hold for a deadlock to occur:

Condition	Description
Mutual Exclusion	Threads claim exclusive control of resources that they require.
Hold-and-wait	Threads hold resources allocated to them while waiting for additional resources
No preemption	Resources cannot be forcibly taken away from threads that are holding them.
Circular wait	There exists a circular chain of threads such that each thread holds one more resources that are being requested by the next thread in the chain

  - If any of these four conditions are not met, **deadlock cannot occur**.

Prevention – Mutual Exclusion

- **wait-free**
  - Using powerful hardware instruction.
  - You can build data structures in a manner that *does not require* explicit locking.

```
1    int CompareAndSwap(int *address, int expected, int new){
2         if(*address == expected){
3                   *address = new;
4                   return 1;          // success – got the lock
5         }
6         return 0;                    // Returning w/o the lock
7    }
```

Prevention – Mutual Exclusion (Cont.)

- We now want to atomically increment a value by a certain amount:

```
1    void AtomicIncrement(int *value, int amount){
2         do{
3                   int old = *value;
4         } while( CompareAndSwap(*value, old, old+amount)==0);
5    }
```

- **Repeatedly tries** to update the value to *the new amount* and uses the compare-and-swap to do so.
- **No lock** is acquired
- **No deadlock** can arise
- **livelock** is still a possibility.

Compare *value and old:
If equal, set *value = old + amount and returns 1
If not equal, retuns 0

Prevention – Mutual Exclusion (Cont.)

- **More complex example**: inserting element into a list

```
1    void insert(int value){
2         node_t * n = malloc(sizeof(node_t));
3         assert( n != NULL );
4         n→value = value ;
5         n→next  = head;
6         head    = n;
7    }
```

- If called by multiple threads at the "*same time*", this code has a race condition→ surround critical section with acquire and release lock

```
1    void insert(int value){
2         node_t * n = malloc(sizeof(node_t));
3         assert( n != NULL );
4         n→value = value ;
5         lock(listlock);            // begin critical section
6         n->next  = head;
7         head     = n;
8         unlock(listlock) ;         // end critical section
9    }
```

Prevention – Hold-and-wait

- **Acquire all locks** at once, atomically:

```
1     lock(prevention);
2     lock(L1);
3     lock(L2);
4     …
5     lock(N)
5     unlock(prevention);
```

  - **This code guarantees** that **no un-timely thread switch can occur** *in the midst of* lock acquisition:
  - **Problem**:
    - Require us to know when calling a routine exactly which locks must be held and to acquire them ahead of time.
    - Decrease *concurrency*

Prevention – No Preemption

- **Multiple lock acquisition** often gets us into trouble because when waiting for one lock we are holding another.
- **trylock**()
  - Used to build a *deadlock-free*, *ordering-robust* lock acquisition protocol.
  - Grab the lock (if it is available).
  - Or, return -1: you should try again later.

```
1    top:
2         lock(L1);
3         if( tryLock(L2) == -1 ){
4                   unlock(L1);
5                   sleep(random time)
6                   goto top;
7         }
```

Prevention – No Preemption (Cont.)

- **livelock**
  - Both users are running through the code sequence (trylock()) *over and over again (virtual deadlock)*.
  - Progress is not being made.
  - **Solution:**
    - Add **a random delay** before looping back and trying the entire thing over again.

```
1    top:
2         lock(L1);
3         if( tryLock(L2) == -1 ){
4                   unlock(L1);
5                   random_delay()
6                   goto top;
7         }
```

Prevention – Circular Wait

- **Provide a total ordering** on lock acquisition
  - This approach requires *careful design* of global locking strategies.
- **Example**:
  - There are two locks in the system (L1 and L2)
  - We can prevent deadlock by always acquiring L1 before L2.

Deadlock Avoidance via Scheduling

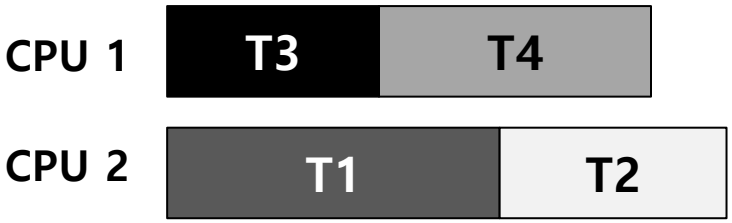
- **In some scenarios** deadlock avoidance is preferable:
  - **Global knowledge** is required:
    - Which locks various threads might grab during their execution.
    - Subsequently schedules threads in a way as to guarantee no deadlock can occur.

Example of Deadlock Avoidance via Scheduling (1)

- **We have two processors and four threads:**
  - **Lock (L1, L2) acquisition demands of the threads (T1 – T4)**:

	T1	T2	T3	T4
L1	yes	yes	no	no
L2	yes	yes	yes	no

  - **A smart scheduler (assuming no preemption)** could compute that as long as T1 and T2 are not run at the same time, no deadlock could ever arise ↑.

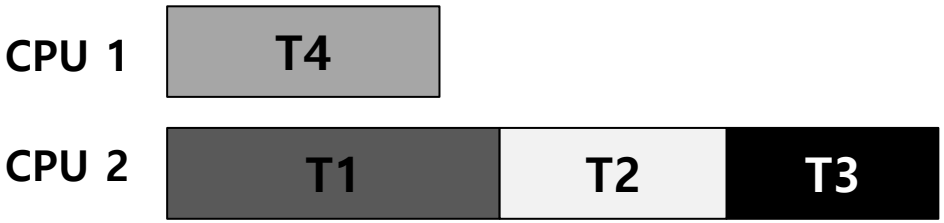


Example of Deadlock Avoidance via Scheduling (2)

- **More contention for the same resources:**

	T1	T2	T3	T4
L1	yes	yes	yes	no
L2	yes	yes	yes	no

  - **A possible schedule** that guarantees that *no deadlock* could ever occur.



    - The total time to complete the jobs is lengthened considerably ↓. Also, fairness ↓

Detect and Recover (not always viable)

- **Allow deadlock** to occasionally occur and then *take some action*:
  - **Example**: if an OS froze, you would reboot it.

- **Many database systems employ *deadlock detection* and *recovery technique*:**
  - A deadlock detector **runs periodically**.
  - Building a **resource (dependency) graph** and checking it for cycles.
  - In deadlock, the system **needs to be restarted ↓**.

Advanced Concurrency - Event-based

Event-based Concurrency

- **A different style** of **concurrent programming**:
  - **Event-based** is Used in *GUI-based applications*, some types of *internet servers*.
- **The problem** that event-based concurrency addresses is two-fold:
  - **Managing concurrency correctly** in multi-threaded applications.
    - **Missing locks**: deadlock, and other nasty problems can arise.
  - **The developer** has little or no control over what is scheduled at a given moment in time.

The Basic Idea: An Event Loop

- **The approach:**
  - **Wait** for something (i.e., an "*event*") to occur.
  - When it does, **check** what type of event it is?
  - **Do** the small amount of work it requires.
- **Example:**

```
1    while(1){
2         events = getEvents();
3         for( e in events )
4                   processEvent(e);    // event handler
5    }
```

A canonical event-based server (Pseudo code)

How exactly does an event-based server determine which events are taking place.

Why Simpler (No Locks Needed)?

- **The event-based server cannot be interrupted** by another thread:
  - When you have a single CPU and an event-based application.
  - It is decidedly **single threaded**.
  - Thus, *concurrency bugs* common in threaded programs **do not manifest** in the basic event-based approach.

A Problem: Blocking System Calls

- What if an event requires that you issue **a system call** that might block?
  - There are no other threads to run: *just the main event loop*
  - **The entire server will do just that**: block until the call completes. **Huge potential waste of resources**

In event-based systems: no blocking calls are allowed.

A Solution: Asynchronous I/O

- **Enable an application** to issue an I/O request and **return control immediately** to the caller, before the I/O has completed:
  - **Instead of blocking:** periodically check if I/O is completed, in Linux you can set it up so when the I/O is completed you receive a signal

What is still difficult with Events.

- **Systems moved** from a single CPU to **multiple CPUs**:
  - Some of the simplicity of the event-based approach disappears.
- **It does not integrate well** with certain kinds of systems activity.
  - **Ex. Paging**: A server will not make progress (single thread) until page fault completes (implicit blocking).
- **Hard to manage overtime:** The exact semantics of various routines changes.
- **Asynchronous disk I/O never quite integrates in with asynchronous network I/O** as simple and uniform manner as you might think.

END