Synchronization

key concepts

critical sections, mutual exclusion, test-and-set, spinlocks, blocking and blocking locks, semaphores, condition variables, deadlocks

reading

Three Easy Pieces: Chapters 28-32

Thread Synchronization

- All threads in a concurrent program *share access* to the program's global variables and the heap.
- The part of a concurrent program in which a shared object is accessed is called a *critical section*.
- What happens if several threads try to access the same global variable or heap object at the same time?

Critical Section Example

```
/* Note the use of volatile */
int volatile total = 0;

void add() {                     void sub() {
   int i;                           int i;
   for (i=0; i<N; i++) {            for (i=0; i<N; i++) {
      total++;                         total--;
   }                                }
}                                }
```

If one thread executes `add` and another executes `sub` what is the value of `total` when they have finished?

Critical Section Example (assembly detail)

```
/* Note the use of volatile */
int volatile total = 0;

void add() {                     void sub() {
   loadaddr R8 total                loadaddr R10 total
   for (i=0; i<N; i++) {            for (i=0; i<N; i++) {
      lw R9 0(R8)                      lw R11 0(R10)
      add R9 1                         sub R11 1
      sw R9 0(R8)                      sw R11 0(R10)
   }                                }
}                                }
```

Critical Section Example (Trace 1)

```
Thread 1                        Thread 2
loadaddr R8 total
lw R9 0(R8)   R9=0
add R9 1      R9=1
sw R9 0(R8) total=1
               <INTERRUPT>
                                loadaddr R10 total
                                lw R11 0(R10)   R11=0
                                sub R11 1       R11=-1
                                sw R11 0(R10)   total=-1
```

One possible order of execution. Final value of `total` is 0.

Critical Section Example (Trace 2)

```
Thread 1                        Thread 2
loadaddr R8 total
lw R9 0(R8)   R9=0
add R9 1      R9=1
         <INTERRUPT and context switch>
                                loadaddr R10 total
                                lw R11 0(R10)   R11=0
                                sub R11 1       R11=-1
                                sw R11 0(R10)   total=-1
                  ...
         <INTERRUPT and context switch>
sw R9 0(R8) total=1
```

One possible order of execution. Final value of `total` is 1.

Critical Section Example (Trace 3)

Thread 1		Thread 2	
`loadaddr R8 total`		`loadaddr R10 total`	
`lw R9 0(R8)`	`R9=0`	`lw R11 0(R10)`	`R11=0`
`add R9 1`	`R9=1`	`sub R11 1`	`R11=-1`
`sw R9 0(R8)`	`total=1`		
		`sw R11 0(R10)`	`total=-1`

Another possible order of execution, this time on two processors. Final value of `total` is -1.

About `volatile`

```
/* What if we DO NOT use volatile */
int ~~volatile~~ total = 0;
```

```
void add() {
   loadaddr R8 total
   lw R9 0(R8)
   for (i=0; i<N; i++) {
      add R9 1
   }
   sw R9 0(R8)
}
```

```
void sub() {
   loadaddr R10 total
   lw R11 0(R10)
   for (i=0; i<N; i++) {
      sub R11 1
   }
   sw R11 0(R10)
}
```

Without `volatile` the compiler could optimize the code. `volatile` forces the compiler to load and store the value on every use.

Another Critical Section Example (Part 1)

```
int list_remove_front(list *lp) {
   int num;
   list_element *element;
   assert(!is_empty(lp));
   element = lp->first;
   num = lp->first->item;
   if (lp->first == lp->last) {
     lp->first = lp->last = NULL;
   } else {
     lp->first = element->next;
   }
   lp->num_in_list--;
   free(element);
   return num;
}
```

The `list_remove_front` function is a critical section. It may not work properly if two threads call it at the same time on the same `list`. (Why?)

Another Critical Section Example (Part 2)

```
void list_append(list *lp, int new_item) {
   list_element *element = malloc(sizeof(list_element));
   element->item = new_item
   assert(!is_in_list(lp, new_item));
   if (is_empty(lp)) {
     lp->first = element; lp->last = element;
   } else {
     lp->last->next = element; lp->last = element;
   }
   lp->num_in_list++;
}
```

The `list_append` function is part of the same critical section as `list_remove_front`. It may not work properly if two threads call it at the same time, or if a thread calls it while another has called `list_remove_front`

Mutual Exclusion

```
int volatile total = 0;

void add() {                        void sub() {
   int i;                              int i;
   for (i=0; i<N; i++) {               for (i=0; i<N; i++) {
     ----------- mutual exclusion start -------------
        total++;                             total--;
     ----------- mutual exclusion end ---------------
   }                                   }
}                                   }
```

To prevent race conditions, we can enforce *mutual exclusion* on critical sections in the code.

Enforcing Mutual Exclusion With Locks

```
int volatile total = 0;
/* lock for total: false => free, true => locked */
bool volatile total_lock = false;

void add() {                        void sub() {
   int i;                              int i;
   for (i=0; i<N; i++) {               for (i=0; i<N; i++) {
   Acquire(&total_lock);                Acquire(&total_lock);
        total++;                             total--;
   Release(&total_lock);                Release(&total_lock);
   }                                    }
}                                   }
```

Acquire/Release must ensure that only one thread at a time can hold the lock, even if both attempt to Acquire at the same time. If a thread cannot Acquire the lock immediately, it must wait until the lock is available.

Lock Aquire and Release - First Try

```
Acquire(bool *lock) {
  while (*lock == true) ; /* spin until lock is free */
  *lock = true;             /* grab the lock */
}

Release(book *lock) {
  *lock = false;            /* give up the lock */
}
```

This simple approach does not work! (Why?)

Hardware-Specific Synchronization Instructions

- used to implement synchronization primitives like locks
- provide a way to *test and set* a lock in a single *atomic* (indivisible) operation
- example: x86 `xchg` instruction:

  ```
  xchg src,addr
  ```

  where `src` is typically a register, and `addr` is a memory address. Value in register `src` is written to memory at address `addr`, and the old value at `addr` is placed into `src`.
- logical behavior of `xchg` can be thought of as an *atomic* function that behaves like this:

  ```
  Xchg(value,addr) {
    old = *addr;
    *addr = value;
    return(old);
  }
  ```

Lock Aquire and Release with Xchg

```
Acquire(bool *lock) {
  while (Xchg(true,lock) == true) ;
}

Release(book *lock) {
  *lock = false;              /* give up the lock */
}
```

If `Xchg` returns `true`, the lock was already set, and we must continue to loop. If `Xchg` returns `false`, then the lock was free, and we have now acquired it.

This construct is known as a *spin lock*, since a thread busy-waits (loops) in Acquire until the lock is free.

Other Synchronization Instructions

- SPARC `cas` instruction

  ```
  cas addr,R1,R2
  ```

  if value at `addr` matches value in `R1` then swap contents of `addr` and `R2`

- Compare-And-Swap

  ```
  CompareAndSwap(addr,expectedval,newval)
    old = *addr;   // get old value at addr
    if (old == expectedval) *addr = newval;
    return old;
  ```

- MIPS load-linked and store-conditional
  - Load-linked returns the current value of a memory location, while a subsequent store-conditional to the same memory location will store a new value only if no updates have occurred to that location since the load-linked.

Spinlocks in OS/161

```
struct spinlock {
  volatile spinlock_data_t lk_lock;
  struct cpu *lk_holder;
};

void spinlock_init(struct spinlock *lk}
void spinlock_acquire(struct spinlock *lk);
void spinlock_release(struct spinlock *lk);
```

`spinlock_acquire` calls `spinlock_data_testandset` in a loop until the lock is acquired.

Using Load-Linked / Store-Conditional

```
/* return value 0 indicates lock was acquired */
spinlock_data_testandset(volatile spinlock_data_t *sd)
{
  spinlock_data_t x,y;
  y = 1;
  __asm volatile(
    ".set push;"      /* save assembler mode */
    ".set mips32;"    /* allow MIPS32 instructions */
    ".set volatile;"  /* avoid unwanted optimization */
    "ll %0, 0(%2);"   /*   x = *sd */
    "sc %1, 0(%2);"   /*   *sd = y; y = success? */
    ".set pop"        /* restore assembler mode */
    : "=r" (x), "+r" (y) : "r" (sd));
  if (y == 0) { return 1; }
  return x;
}
```

OS/161 Locks

- In addition to spinlocks, OS/161 also has *locks*.
- Like spinlocks, locks are used to enforce mutual exclusion.

```
struct lock *mylock = lock_create("LockName");

lock_aquire(mylock);
  critical section /* e.g., call to list_remove_front */
lock_release(mylock);
```

- spinlocks spin, locks *block*:
  - a thread that calls `spinlock_acquire` spins until the lock can be acquired
  - a thread that calls `lock_acquire` *blocks* until the lock can be acquired

Thread Blocking

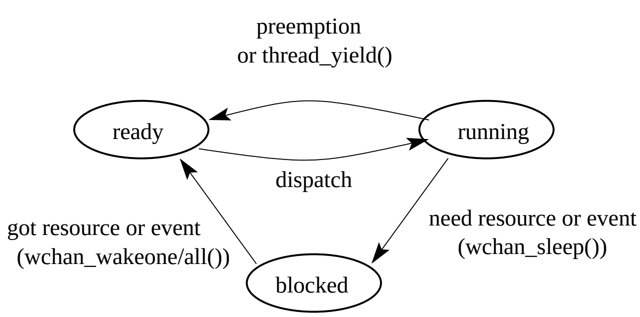
- Sometimes a thread will need to wait for something, e.g.:
  - wait for a lock to be released by another thread
  - wait for data from a (relatively) slow device
  - wait for input from a keyboard
  - wait for busy device to become idle
- When a thread blocks, it stops running:
  - the scheduler chooses a new thread to run
  - a context switch from the blocking thread to the new thread occurs,
  - the blocking thread is queued in a *wait queue* (not on the ready list)
- Eventually, a blocked thread is signaled and awakened by another thread.

Wait Channels in OS/161

- wait channels are used to implement thread blocking in OS/161
  - `void wchan_sleep(struct wchan *wc);`
    - blocks calling thread on wait channel `wc`
    - causes a context switch, like `thread_yield`
  - `void wchan_wakeall(struct wchan *wc);`
    - unblock all threads sleeping on wait channel `wc`
  - `void wchan_wakeone(struct wchan *wc);`
    - unblock one thread sleeping on wait channel `wc`
  - `void wchan_lock(struct wchan *wc);`
    - prevent operations on wait channel `wc`
    - more on this later!
- there can be many different wait channels, holding threads that are blocked for different reasons.

Thread States, Revisited

- a very simple thread state transition diagram



- the states:

  running: currently executing

  ready: ready to execute

  blocked: waiting for something, so not ready to execute.
- ready threads are queued on the ready queue, blocked threads are queued on wait channels

Semaphores

- A semaphore is a synchronization primitive that can be used to enforce mutual exclusion requirements. It can also be used to solve other kinds of synchronization problems.
- A semaphore is an object that has an integer value, and that supports two operations:

  P: if the semaphore value is greater than 0, decrement the value. Otherwise, wait until the value is greater than 0 and then decrement it.

  V: increment the value of the semaphore

By definition, the `P` and `V` operations of a semaphore are *atomic*.

Mutual Exclusion Using a Semaphore

```
volatile int total = 0;
struct semaphore *total_sem;
total_sem = sem_create("total mutex",1);  /* initial value is

void add() {                          void sub() {
   int i;                                int i;
   for (i=0; i<N; i++) {                 for (i=0; i<N; i++) {
     P(sem);                               P(sem);
       total++;                                total--;
     V(sem);                               V(sem);
   }                                     }
}                                     }
```

Producer/Consumer Synchronization with Bounded Buffer

- suppose we have threads (producers) that add items to a buffer and threads (consumers) that remove items from the buffer
- suppose we want to ensure that consumers do not consume if the buffer is empty - instead they must wait until the buffer has something in it
- similarly, suppose the buffer has a finite capacity (N), and we need to ensure that producers must wait if the buffer is full
- this requires synchronization between consumers and producers
- semaphores can provide the necessary synchronization

Bounded Buffer Producer/Consumer Synchronization with Semaphores

```
struct semaphore *Items,*Spaces;
Items = sem_create("Buffer Items", 0);  /* initially = 0 */
Spaces = sem_create("Buffer Spaces", N);/* initially = N */

Producer's Pseudo-code:
  P(Spaces);
  add item to the buffer
  V(Items);

Consumer's Pseudo-code:
  P(Items);
  remove item from the buffer
  V(Spaces);
```

Condition Variables

- OS/161 supports another common synchronization primitive: *condition variables*
- each condition variable is intended to work together with a lock: condition variables are only used *from within the critical section that is protected by the lock*
- three operations are possible on a condition variable:

  wait: This causes the calling thread to block, and it releases the lock associated with the condition variable. Once the thread is unblocked it reacquires the lock.

  signal: If threads are blocked on the signaled condition variable, then one of those threads is unblocked.

  broadcast: Like signal, but unblocks all threads that are blocked on the condition variable.

Using Condition Variables

- Condition variables get their name because they allow threads to wait for arbitrary conditions to become true inside of a critical section.
- Normally, each condition variable corresponds to a particular condition that is of interest to an application. For example, in the bounded buffer producer/consumer example on the following slides, the two conditions are:
  - $count > 0$ (there are items in the buffer)
  - $count < N$ (there is free space in the buffer)
- when a condition is not true, a thread can `wait` on the corresponding condition variable until it becomes true
- when a thread detects that a condition is true, it uses `signal` or `broadcast` to notify any threads that may be waiting

Note that signalling (or broadcasting to) a condition variable that has no waiters has *no effect*. Signals do not accumulate.

Waiting on Condition Variables

- when a blocked thread is unblocked (by `signal` or `broadcast`), it reacquires the lock before returning from the `wait` call
- a thread is in the critical section when it calls `wait`, and it will be in the critical section when `wait` returns. However, in between the call and the return, while the caller is blocked, the caller is out of the critical section, and other threads may enter.
- In particular, the thread that calls `signal` (or `broadcast`) to wake up the waiting thread will itself be in the critical section when it signals. The waiting thread will have to wait (at least) until the signaller releases the lock before it can unblock and return from the `wait` call.

This describes Mesa-style condition variables, which are used in OS/161. There are alternative condition variable semantics (Hoare semantics), which differ from the semantics described here.

Bounded Buffer Producer Using Locks and Condition Variables

```
int volatile count = 0;  /* must initially be 0 */
struct lock *mutex;      /* for mutual exclusion */
struct cv *notfull, *notempty; /* condition variables */

/* Initialization Note: the lock and cv's must be created
 * using lock_create() and cv_create() before Produce()
 * and Consume() are called */

Produce(itemType item) {
  lock_acquire(mutex);
  while (count == N) {
     cv_wait(notfull, mutex); /* wait until buffer is not ful
  }
  add item to buffer (call list_append())
  count = count + 1;
  cv_signal(notempty, mutex); /* signal that buffer is not em
  lock_release(mutex);
}
```

Bounded Buffer Consumer Using Locks and Condition Variables

```
itemType Consume() {
  lock_acquire(mutex);
  while (count == 0) {
     cv_wait(notempty, mutex); /* wait until buffer is not em
  }
  remove item from buffer (call list_remove_front())
  count = count - 1;
  cv_signal(notfull, mutex); /* signal that buffer is not ful
  lock_release(mutex);
  return(item);
}
```

Both `Produce()` and `Consume()` call `cv_wait()` inside of a `while` loop. Why?

Deadlocks

- Suppose there are two threads and two locks, `lockA` and `lockB`, both initially unlocked.
- Suppose the following sequence of events occurs
  1. Thread 1 does `lock_acquire(lockA)`.
  2. Thread 2 does `lock_acquire(lockB)`.
  3. Thread 1 does `lock_acquire(lockB)` and blocks, because `lockB` is held by thread 2.
  4. Thread 2 does `lock_acquire(lockA)` and blocks, because `lockA` is held by thread 1.

These two threads are *deadlocked* - neither thread can make progress. Waiting will not resolve the deadlock. The threads are permanently stuck.

Two Techniques for Deadlock Prevention

No Hold and Wait: prevent a thread from requesting resources if it currently has resources allocated to it. A thread may hold several resources, but to do so it must make a single request for all of them.

Resource Ordering: Order (e.g., number) the resource types, and require that each thread acquire resources in increasing resource type order. That is, a thread may make no requests for resources of type less than or equal to i if it is holding resources of type i.