Concurrency

- On multiprocessors, several threads can execute simultaneously, one on each processor.
- On uniprocessors, only one thread executes at a time. However, because of preemption and timesharing, threads appear to run concurrently.

Concurrency and synchronization are important even on uniprocessors.

Thread Synchronization

- Concurrent threads can interact with each other in a variety of ways:
  - Threads share access, through the operating system, to system devices (more on this later . . .)
  - Threads may share access to program data, e.g., global variables.
- A common synchronization problem is to enforce *mutual exclusion*, which means making sure that only one thread at a time uses a shared object, e.g., a variable or a device.
- The part of a program in which the shared object is accessed is called a *critical section*.

Critical Section Example (Part 0)

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

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

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

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

Critical Section Example (Part 0)

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

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

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

Critical Section Example (Part 0)

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

One possible order of execution.

Critical Section Example (Part 0)

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

Another possible order of execution. Many interleavings of instructions are possible. Synchronization is required to ensure a correct ordering.

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

The use of volatile

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

void add() {
   loadaddr R8 total
   lw R9 0(R8)
   add R9 N
   sw R9 0(R8)
}

void sub() {
   loadaddr R10 total
   lw R11 0(R10)
   sub R11 N
   sw R11 0(R10)
}
```

The compiler could aggressively optimize the code., Volatile tells the compiler that the object may change for reasons which cannot be determined from the local code (e.g., due to interaction with a device or because of another thread).

The use of volatile

```
/* 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)
   }                                 }
}                                 }
```

The volatile declaration forces the compiler to load and store the value on every use. Using volatile is necessary but not sufficient for correct behaviour. Mutual exclusion is also required to ensure a correct ordering of instructions.

Ensuring Correctness with Multiple Threads

```
/* 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++) {
     Allow one thread to execute and make others wait
        total++;                              total--;
     Permit one waiting thread to continue execution
   }                                    }
}                                    }
```

Threads must enforce mutual exclusion.

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?)

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`

Enforcing Mutual Exclusion

- mutual exclusion algorithms ensure that only one thread at a time executes the code in a critical section
- several techniques for enforcing mutual exclusion
  - exploit special hardware-specific machine instructions, e.g., *test-and-set*, *compare-and-swap*, or *load-link / store-conditional*, that are intended for this purpose
  - use mutual exclusion algorithms, e.g., *Peterson's algorithm*, that rely only on atomic loads and stores
  - control interrupts to ensure that threads are not preempted while they are executing a critical section

Disabling Interrupts

- On a uniprocessor, only one thread at a time is actually running.
- If the running thread is executing a critical section, mutual exclusion may be violated if
  1. the running thread is preempted (or voluntarily yields) while it is in the critical section, and
  2. the scheduler chooses a different thread to run, and this new thread enters the same critical section that the preempted thread was in
- Since preemption is caused by timer interrupts, mutual exclusion can be enforced by disabling timer interrupts before a thread enters the critical section, and re-enabling them when the thread leaves the critical section.

Interrupts in OS/161

This is one way that the OS/161 kernel enforces mutual exclusion on a single processor. There is a simple interface

- `spl0()` sets IPL to 0, enabling all interrupts.
- `splhigh()` sets IPL to the highest value, disabling all interrupts.
- `splx(s)` sets IPL to S, enabling whatever state S represents.

These are used by splx() and by the spinlock code.

- `splraise(int oldipl, int newipl)`
- `spllower(int oldipl, int newipl)`
- For splraise, `NEWIPL > OLDIPL`, and for spllower, `NEWIPL < OLDIPL`.

See `kern/include/spl.h` and `kern/thread/spl.c`

Pros and Cons of Disabling Interrupts

- advantages:
  - does not require any hardware-specific synchronization instructions
  - works for any number of concurrent threads
- disadvantages:
  - indiscriminate: prevents all preemption, not just preemption that would threaten the critical section
  - ignoring timer interrupts has side effects, e.g., kernel unaware of passage of time. (Worse, OS/161's `splhigh()` disables *all* interrupts, not just timer interrupts.) Keep critical sections *short* to minimize these problems.
  - will not enforce mutual exclusion on multiprocessors (why??)

Peterson's Mutual Exclusion Algorithm

```
/* shared variables */
/* note: one flag array and turn variable */
/* for each critical section */
boolean volatile flag[2];  /* shared, initially false */
int volatile turn;         /* shared */

flag[i] = true;   /* for one thread, i = 0 and j = 1 */
turn = j;         /* for the other, i = 1 and j = 0 */
while (flag[j] && turn == j) { }  /* busy wait */

 critical section    /* e.g., call to list_remove_front */

flag[i] = false;
```

Ensures mutual exclusion and avoids starvation, but works only for two threads. (Why?)

Hardware-Specific Synchronization Instructions

- a test-and-set instruction *atomically* sets the value of a specified memory location and either
  - places that memory location's *old* value into a register, or
  - checks a condition against the memory location's old value and records the result of the check in a register
- for presentation purposes, we will abstract such an instruction as a function `TestAndSet(address,value)`, which takes a memory location (`address`) and a value as parameters. It atomically stores `value` at the memory location specified by `address` and returns the previous value stored at that address
- Often only two values are used 0 and 1 so the `value` parameter is not used and a value of 1 is implied (e.g., in OS/161)

A Spin Lock Using Test-And-Set in OS/161

- a test-and-set instruction can be used to enforce mutual exclusion
- for each critical section, define a shared variable

```
volatile spinlock_data_t lk_lock; /* initially 0 */
```

  We will use the lock variable to keep track of whether there is a thread in the critical section, in which case the value of `lk_lock` will be `1`

- before a thread can enter the critical section, it does the following:

```
while (spinlock_data_testandset(&lk->lk_lock) != 0) {
  /* busy wait */
}
```

- if `lk_lock == 0` then it is set to 1 and the thread enters the critical section
- when the thread leaves the critical section, it does:

```
spinlock_data_set(&lk->lk_lock, 0);
```


A Spin Lock Using Test-And-Set

- this enforces mutual exclusion (why?), but starvation is a possibility

---

This construct is sometimes known as a *spin lock*, since a thread “spins” in the while loop until the critical section is free. Spin locks are widely used on multiprocessors.

Spinlocks in OS/161

```
struct spinlock {
  volatile spinlock_data_t lk_lock; /* word for spin */
  struct cpu *lk_holder; /* CPU holding this lock */
};

void spinlock_init(struct spinlock *lk);
void spinlock_cleanup(struct spinlock *lk);
void spinlock_acquire(struct spinlock *lk);
void spinlock_release(struct spinlock *lk);
bool spinlock_do_i_hold(struct spinlock *lk);
```


Spinlocks in OS/161

```
spinlock_init(struct spinlock *lk)
{
  spinlock_data_set(&lk->lk_lock, 0);
  lk->lk_holder = NULL;
}

void spinlock_cleanup(struct spinlock *lk)
{
  KASSERT(lk->lk_holder == NULL);
  KASSERT(spinlock_data_get(&lk->lk_lock) == 0);
}

void spinlock_data_set(volatile spinlock_data_t *sd,
  unsigned val)
{
  *sd = val;
}
```

Spinlocks in OS/161

```
void spinlock_acquire(struct spinlock *lk)
{
  struct cpu *mycpu;
  splraise(IPL_NONE, IPL_HIGH);

  /* this must work before curcpu initialization */
  if (CURCPU_EXISTS()) {
    mycpu = curcpu->c_self;
    if (lk->lk_holder == mycpu) {
      panic("Deadlock on spinlock %p\n", lk);
    }
  } else {
    mycpu = NULL;
  }
```


Spinlocks in OS/161

```
  while (1) {
    /* Do test-test-and-set to reduce bus contention */
    if (spinlock_data_get(&lk->lk_lock) != 0) {
      continue;
    }
    if (spinlock_data_testandset(&lk->lk_lock) != 0) {
      continue;
    }
    break;
  }

  lk->lk_holder = mycpu;
}
```

Spinlocks in OS/161

```
void spinlock_release(struct spinlock *lk)
{
  /* this must work before curcpu initialization */
  if (CURCPU_EXISTS()) {
    KASSERT(lk->lk_holder == curcpu->c_self);
  }

  lk->lk_holder = NULL;
  spinlock_data_set(&lk->lk_lock, 0);
  spllower(IPL_HIGH, IPL_NONE);
}
```


Load-Link / Store-Conditional

Load-link 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-link.

```
spinlock_data_testandset(volatile spinlock_data_t *sd)
{
  spinlock_data_t x,y;

  /* Test-and-set using LL/SC.
   * Load the existing value into X, and use Y to store 1.
   * After the SC, Y contains 1 if the store succeeded,
   * 0 if it failed. On failure, return 1 to pretend
   * that the spinlock was already held.
   */

  y = 1;
```

Load-Link / Store-Conditional

```
  __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;
}
```


Pros and Cons of Spinlocks

- Pros:
  - can be efficient for short critical sections
  - using hardware specific synchronization instructions means it works on multiprocessors
- Cons:
  - CPU is busy (nothing else runs) while waiting for lock
  - starvation is possible

If critical section is short prefer spinlock.

If critical section is long prefer blocking lock.

Hybrid locks will spin for a period of time before blocking.

Question: How to determine how long to spin for hybrid lock?

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

- Two kinds of semaphores:

  counting semaphores: can take on any non-negative value

  binary semaphores: take on only the values 0 and 1. (`V` on a binary semaphore with value 1 has no effect.)

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

A Simple Example using Semaphores

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

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

What type of semaphore can be used for `sem`?

OS/161 Semaphores

```
struct semaphore {
  char *sem_name;
  struct wchan *sem_wchan;
  struct spinlock sem_lock;
  volatile int sem_count;
};

struct semaphore *sem_create(const char *name,
  int initial_count);
void P(struct semaphore *s);
void V(struct semaphore *s);
void sem_destroy(struct semaphore *s);
```

see `kern/include/synch.h` and `kern/thread/synch.c`

Mutual Exclusion Using a Semaphore

```
struct semaphore *s;
s = sem_create("MySem1", 1); /* initial value is 1 */

P(s);  /* do this before entering critical section */

   critical section /* e.g., call to list_remove_front */

V(s);  /* do this after leaving critical section */
```

OS/161 Semaphores: P() from kern/thread/synch.c

```
P(struct semaphore *sem)
{
  KASSERT(sem != NULL);
  KASSERT(curthread->t_in_interrupt == false);

  spinlock_acquire(&sem->sem_lock);
    while (sem->sem_count == 0) {
      /* Note: we don't maintain strict FIFO ordering */
      wchan_lock(sem->sem_wchan);
      spinlock_release(&sem->sem_lock);
      wchan_sleep(sem->sem_wchan);
      spinlock_acquire(&sem->sem_lock);
    }
    KASSERT(sem->sem_count > 0);
    sem->sem_count--;
  spinlock_release(&sem->sem_lock);
}
```


OS/161 Semaphores: V() from kern/thread/synch.c

```
V(struct semaphore *sem)
{
  KASSERT(sem != NULL);

  spinlock_acquire(&sem->sem_lock);

    sem->sem_count++;
    KASSERT(sem->sem_count > 0);
    wchan_wakeone(sem->sem_wchan);

  spinlock_release(&sem->sem_lock);
}
```

Thread Blocking

- Sometimes a thread will need to wait for an event. One example is on the previous slide: a thread that attempts a `P()` operation on a zero-valued semaphore must wait until the semaphore's value becomes positive.
- other examples that we will see later on:
  - wait for data from a (relatively) slow device
  - wait for input from a keyboard
  - wait for busy device to become idle
- In these circumstances, we do not want the thread to run, since it cannot do anything useful.
- To handle this, the thread scheduler can *block* threads.

Thread Blocking in OS/161

- OS/161 thread library functions for blocking and unblocking threads:
  - `void wchan_lock(struct wchan *wc);`
  - `void wchan_unlock(struct wchan *wc);`
    - locks/unlocks the wait channel `wc`
  - `void wchan_sleep(struct wchan *wc);`
    - blocks calling thread on wait channel `wc`
    - channel must be locked, will be unlocked upon return
  - `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`

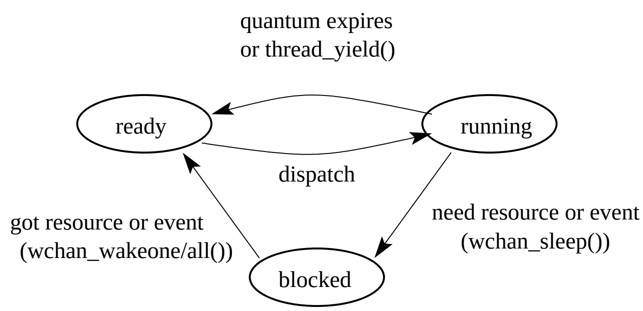
Note: current implementation is FIFO but not promised by the interface

Thread Blocking in OS/161

- `wchan_sleep()` is much like `thread_yield()`. The calling thread is voluntarily giving up the CPU, so the scheduler chooses a new thread to run, the state of the running thread is saved and the new thread is dispatched. However:
  - after a `thread_yield()`, the calling thread is *ready* to run again as soon as it is chosen by the scheduler
  - after a `wchan_sleep()`, the calling thread is *blocked*, and must not be scheduled to run again until after it has been explicitly unblocked by a call to `wchan_wakeone()` or `wchan_wakeall()`.

Thread States

- 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.

Producer/Consumer Synchronization

- suppose we have threads that add items to a list (producers) and threads that remove items from the list (consumers)
- suppose we want to ensure that consumers do not consume if the list is empty - instead they must wait until the list has something in it
- this requires synchronization between consumers and producers
- semaphores can provide the necessary synchronization, as shown on the next slide

Producer/Consumer Synchronization using Semaphores

```
struct semaphore *s;
s = sem_create("Items", 0); /* initial value is 0 */

Producer's Pseudo-code:
  add item to the list (call list_append())
  V(s);

Consumer's Pseudo-code:
  P(s);
  remove item from the list (call list_remove_front())
```

The Items semaphore does not enforce mutual exclusion on the list. If we want mutual exclusion, we can also use semaphores to enforce it. (How?)

Bounded Buffer Producer/Consumer Synchronization

- suppose we add one more requirement: the number of items in the list should not exceed `N`
- producers that try to add items when the list is full should be made to wait until the list is no longer full
- We can use an additional semaphore to enforce this new constraint:
  - semaphore `Full` is used to count the number of full (occupied) entries in the list (to ensure nothing is produced if the list is full)
  - semaphore `Empty` is used to count the number of empty (unoccupied) entries in the list (to ensure nothing is consumed if the list is empty)

```
struct semaphore *full;
struct semaphore *empty;
full = sem_create("Full", 0);   /* initial value = 0 */
empty = sem_create("Empty", N); /* initial value = N */
```


Bounded Buffer Producer/Consumer Synchronization with Semaphores

```
Producer's Pseudo-code:
  P(empty);
  add item to the list (call list_append())
  V(full);

Consumer's Pseudo-code:
  P(full);
  remove item from the list (call list_remove_front())
  V(empty);
```

OS/161 Locks

- OS/161 also uses a synchronization primitive called a *lock*. Locks are intended to be 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);
```

- A lock is similar to a binary semaphore with an initial value of 1. However, locks also enforce an additional constraint: the thread that releases a lock must be the same thread that most recently acquired it.
- The system enforces this additional constraint to help ensure that locks are used as intended.

Reader/Writer Locks

- Reader/Writer (or a shared) locks can be acquired in either of read (shared) or write (exclusive) mode
- In OS/161 reader/writer locks might look like this:

```
struct rwlock *rwlock = rw_lock_create("RWLock");

rwlock_aquire(rwlock, READ_MODE);
    can only read shared resources
   /* access is shared by readers */
rwlock_release(rwlock);

rwlock_aquire(rwlock, WRITE_MODE);
    can read and write shared resources
   /* access is exclusive to only one writer */
rwlock_release(rwlock);
```

Critical Section Requirements

- **Mutual exclusion**: While one thread is executing in the critical section no other thread can execute in that critical section.
- **Progress**: The thread in the critical section will eventually leave the critical section.
- **Bounded waiting**: Any thread will wait for a bounded amount of time before it is able to enter the critical section.

Performance Issues

- **Overhead**: the memory and CPU resources used when acquiring and releasing access to critical sections
- **Contention**: competition for access to the critical section
- **Granularity**: the amount of code executed while in a critical section

Why are these important issues?

Lock Overhead, Contention and Granularity (Option 1)

```
void add() {
   int i;
     for (i=0; i<N; i++) {
       P / Acquire
         total++;
       V / Release
     }
}
```

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

Should one use `P()/V()`, `spinlock_acquire()/spinlock_release()` or `lock_acquire()/lock_release()`?

Lock Overhead, Contention and Granularity (Option 2)

```
void add() {
   int i;
   P / Acquire
     for (i=0; i<N; i++) {
         total++;
     }
   V / Release
}
```

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

Which option is better Option 1 (previous slide) or 2 (this slide)? Why?

Does the choice of where to do synchronization influence the choice of which mechanism to use for synchronization?

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:
  - $count > 0$ (condition variable `notempty`)
  - $count < N$ (condition variable `notfull`)
- 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 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);
  }
  add item to buffer (call list_append())
  count = count + 1;
  cv_signal(notempty, mutex);
  lock_release(mutex);
}
```

Bounded Buffer Consumer Using Condition Variables

```
itemType Consume() {
  lock_acquire(mutex);
  while (count == 0) {
     cv_wait(notempty, mutex);
  }
  remove item from buffer (call list_remove_front())
  count = count - 1;
  cv_signal(notfull, mutex);
  lock_release(mutex);
  return(item);
}
```

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

Monitors

- Condition variables are derived from *monitors*. A monitor is a programming language construct that provides synchronized access to shared data. Monitors have appeared in many languages, e.g., Ada, Mesa, Java.
- a monitor is essentially an object with special concurrency semantics
- it is an object, meaning
  - it has data elements
  - the data elements are encapsulated by a set of methods, which are the only functions that directly access the object's data elements
- only *one* monitor method may be active at a time, i.e., the monitor methods (together) form a critical section
  - if two threads attempt to execute methods at the same time, one will be blocked until the other finishes
- inside a monitor, condition variables can be declared and used

Monitors in OS/161

- The C language, in which OS/161 is written, does not support monitors.
- However, programming convention and OS/161 locks and condition variables can be used to provide monitor-like behavior for shared kernel data structures:
  - define a C structure to implement the object's data elements
  - define a set of C functions to manipulate that structure (these are the object "methods")
  - ensure that only those functions directly manipulate the structure
  - create an OS/161 lock to enforce mutual exclusion
  - ensure that each access method acquires the lock when it starts and releases the lock when it finishes
  - if desired, define one or more condition variables and use them within the methods.

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.

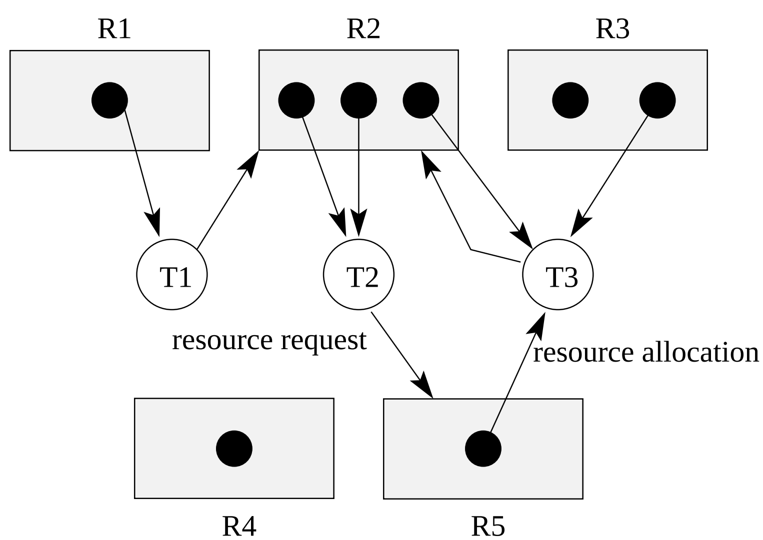
These two threads are *deadlocked* - neither thread can make progress. Waiting will not resolve the deadlock. The threads are permanently stuck.

Deadlocks (Another Simple Example)

- Suppose a machine has 64 MB of memory. The following sequence of events occurs.
  1. Thread A starts, requests 30 MB of memory.
  2. Thread B starts, also requests 30 MB of memory.
  3. Thread A requests an additional 8 MB of memory. The kernel blocks thread A since there is only 4 MB of available memory.
  4. Thread B requests an additional 5 MB of memory. The kernel blocks thread B since there is not enough memory available.

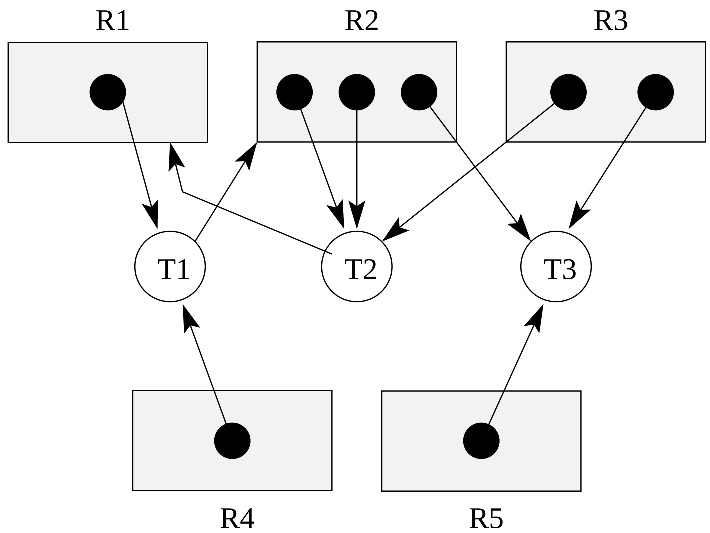
These two threads are deadlocked.

Resource Allocation Graph (Example)



Is there a deadlock in this system?

Resource Allocation Graph (Another Example)



Is there a deadlock in this system?

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.

Preemption: take resources away from a thread and give them to another (usually not possible). Thread is restarted when it can acquire all the resources it needs.

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.

Deadlock Detection and Recovery

- main idea: the system maintains the resource allocation graph and tests it to determine whether there is a deadlock. If there is, the system must recover from the deadlock situation.
- deadlock recovery is usually accomplished by terminating one or more of the threads involved in the deadlock
- when to test for deadlocks? Can test on every blocked resource request, or can simply test periodically. Deadlocks persist, so periodic detection will not "miss" them.

Deadlock detection and deadlock recovery are both costly. This approach makes sense only if deadlocks are expected to be infrequent.

Detecting Deadlock in a Resource Allocation Graph

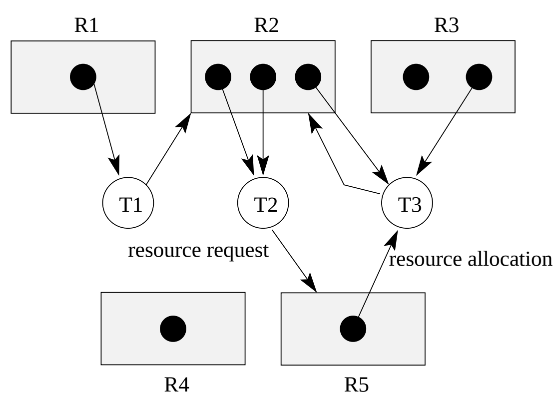
- System State Notation:
  - D_i: demand vector for thread T_i
  - A_i: current allocation vector for thread T_i
  - U: unallocated (available) resource vector
- Additional Algorithm Notation:
  - R: scratch resource vector
  - f_i: algorithm is finished with thread T_i? (boolean)

Detecting Deadlock (cont'd)

/* initialization */
$R = U$
for all i, $f_i =$ false
/* can each thread finish? */
while $\exists\ i\ (\ \neg\ f_i\ \wedge\ (D_i\ \leq\ R)\ )\ \{$
    $R\ =\ R\ +\ A_i$
    $f_i\ =$ true
$\}$
/* if not, there is a deadlock */
if $\exists\ i\ (\ \neg\ f_i\ )$ then report deadlock
else report no deadlock

Deadlock Detection, Positive Example

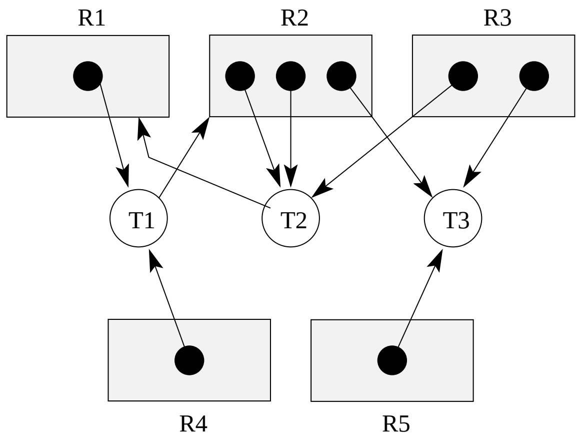
- $D_1 = (0, 1, 0, 0, 0)$
- $D_2 = (0, 0, 0, 0, 1)$
- $D_3 = (0, 1, 0, 0, 0)$
- $A_1 = (1, 0, 0, 0, 0)$
- $A_2 = (0, 2, 0, 0, 0)$
- $A_3 = (0, 1, 1, 0, 1)$
- $U = (0, 0, 1, 1, 0)$



The deadlock detection algorithm will terminate with $f_1 == f_2 == f_3 ==$ false, so this system is deadlocked.

Deadlock Detection, Negative Example

- $D_1 = (0, 1, 0, 0, 0)$
- $D_2 = (1, 0, 0, 0, 0)$
- $D_3 = (0, 0, 0, 0, 0)$
- $A_1 = (1, 0, 0, 1, 0)$
- $A_2 = (0, 2, 1, 0, 0)$
- $A_3 = (0, 1, 1, 0, 1)$
- $U = (0, 0, 0, 0, 0)$



This system is not in deadlock. It is possible that the threads will run to completion in the order T_3, T_1, T_2.