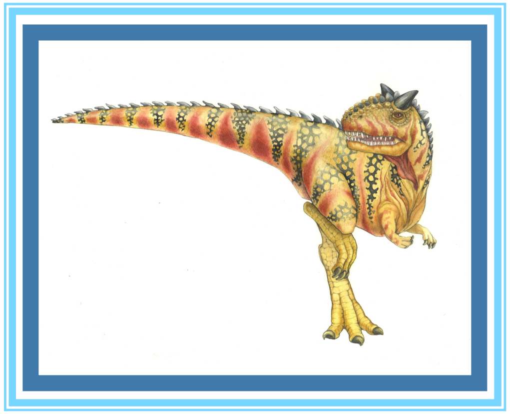
OS161

Synchronization primitives

Chapter 6: Synchronization Tools



Chapter 6: Synchronization Tools

- Background
- The Critical-Section Problem
- Peterson’s Solution
- Hardware Support for Synchronization
- Mutex Locks
- Semaphores
- Monitors
- Liveness
- Evaluation

Background

- Concurrent programming and synchronization
- Coordinating access to shared resources
- Problems
  - Race conditions
  - Deadlocks
  - Resource starvation
- Solutions
  - Synchronization: locks, barriers, semaphores, etc.
  - …

Critical section

- Part of a program that cannot be executed by more than one process/thread at a given time, due to shared resources (variables, tables, files, etc.)

- Solved by
  - Mutual exclusion: Pi and Pj cannot execute their critical sections concurrently
  - Progress: if Pi wish to execute its critical section and no other in its critical section, Pi not waiting indefinitely
  - Bounded wait: if Pi waiting to execute its critical section, bound on number of times other processes enter their critical sections

- Solutions depend on kernel and parallel cores
  - Single core preemprive: preemption allowed when processes in kernel mode
  - Single core non-preemptive: process free of race conditions while in kernel mode
  - Multicore: parallel process always possible, regardless of preemption

Critical Section Problem

- Consider system of ***n*** processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Critical Section

- General structure of process P_i

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

Solution to Critical-Section Problem

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

Critical-Section Handling in OS

Two approaches depending on if kernel is preemptive or non-preemptive

- **Preemptive** – allows preemption of process when running in kernel mode
- **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
  - Essentially free of race conditions in kernel mode

Peterson's Solution

- Not guaranteed to work on modern architectures! (But good algorithmic description of solving the problem)
- Two process solution
- Assume that the **`load`** and **`store`** machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
  - **`int turn;`**
  - **`boolean flag[2]`**
- The variable **`turn`** indicates whose turn it is to enter the critical section
- The **`flag`** array is used to indicate if a process is ready to enter the critical section. **`flag[i] = true`** implies that process P_i is ready!

Algorithm for Process P_i

```
while (true){
        flag[i] = true;
        turn = j;
        while (flag[j] && turn = = j)
                ;

        /* critical section */

        flag[i] = false;

        /* remainder section */

}
```

Peterson's Solution (Cont.)

- Provable that the three CS requirement are met:
  1. Mutual exclusion is preserved

     $\mathbf{P_i}$ enters CS only if:

     either `flag[j] = false` or `turn = i`
  2. Progress requirement is satisfied
  3. Bounded-waiting requirement is met

Peterson's Solution

- Although useful for demonstrating an algorithm, Peterson's Solution is not guaranteed to work on modern architectures.
- Understanding why it will not work is also useful for better understanding race conditions.
- To improve performance, processors and/or compilers may reorder operations that have no dependencies.
- For single-threaded this is ok as the result will always be the same.
- For multithreaded the reordering may produce inconsistent or unexpected results!

Peterson's Solution

- Two threads share the data:

```
boolean flag = false;
int x = 0;
```

- Thread 1 performs

```
while (!flag)
    ;
print x
```

- Thread 2 performs

```
x = 100;
flag = true
```

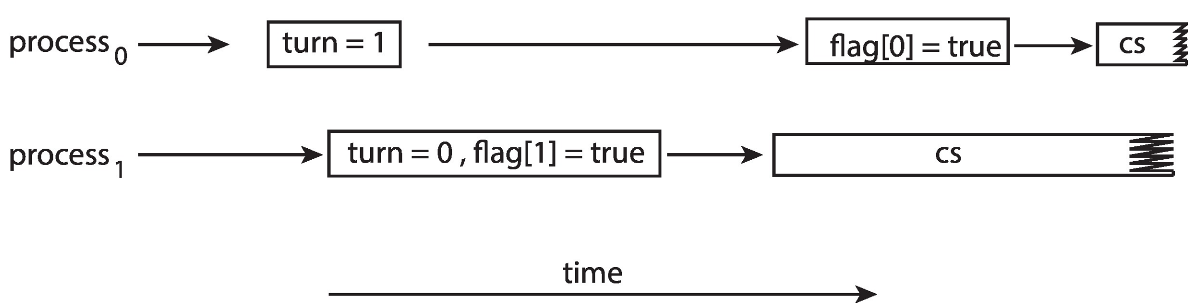
- What is the expected output?

Peterson's Solution

- 100 is the expected output.
- However, the operations for Thread 2 may be reordered:

```
flag = true;
x = 100;
```

- If this occurs, the output may be 0!
- The effects of instruction reordering in Peterson's Solution



- This allows both processes to be in their critical section at the same time!

Solutions: Lock (multiple processes allowed)

```
/* critical section: i,j are 0 or 1 */
do {
    acquire lock
    critical section
    release lock
    remainder section
} while (true);
```

What Should you do if you can't get a lock?

- Keep trying
  - "spin" or "busy-wait"
  - Good if delays are short
- Give up the processor
  - Good if delays are long
  - Always good on uniprocessor

Lock implementation

- Need hardware support

- Uniprocessors – could disable interrupts
  - execute without preemption
  - Cannot be extended to multiprocessor systems
  - Operating systems using this not broadly scalable

- Modern machines provide special atomic hardware instructions
  Atomic = non-interruptible
  - Either test memory word and set value (test-and-set)
  - Or swap contents of two memory words (compare-and-swap)

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- We will look at three forms of hardware support:

  1. Memory barriers

  2. Hardware instructions

  3. Atomic variables

Memory Barriers

- **Memory model** are the memory guarantees a computer architecture makes to application programs.
- Memory models may be either:
  - **Strongly ordered** – where a memory modification of one processor is immediately visible to all other processors.
  - **Weakly ordered** – where a memory modification of one processor may not be immediately visible to all other processors.
- A **memory barrier** is an instruction that forces any change in memory to be propagated (made visible) to all other processors.

Memory Barrier

- We could add a memory barrier to the following instructions to ensure Thread 1 outputs 100:
- Thread 1 now performs

```
while (!flag)
    memory_barrier();
print x
```

- Thread 2 now performs

```
x = 100;
memory_barrier();
flag = true
```

Hardware Instructions

- Special hardware instructions that allow us to either *test-and-modify* the content of a word, or two *swap* the contents of two words atomically (uninterruptibly.)
- **Test-and-Set** instruction
- **Compare-and-Swap** instruction

test_and_set Instruction

Definition:

```
boolean test_and_set (boolean *target)
{
        boolean rv = *target;
        *target = true;
        return rv:
}
```

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to **`true`**

Solution using test_and_set()

- Shared boolean variable **lock**, initialized to **false**
- Solution:

```
do {
      while (test_and_set(&lock))
       ; /* do nothing */

          /* critical section */

   lock = false;
          /* remainder section */
 } while (true);
```

compare_and_swap Instruction

Definition:

```
int compare _and_swap(int *value, int expected, int new_value) {
    int temp = *value;

    if (*value == expected)
       *value = new_value;
 return temp;
}
```

1. Executed atomically
2. Returns the original value of passed parameter **`value`**
3. Set the variable **`value`** the value of the passed parameter **`new_value`** but only if **`*value == expected`** is true. That is, the swap takes place only under this condition.

Solution using compare_and_swap

- Shared integer `lock` initialized to 0;
- Solution:

```
while (true){
    while (compare_and_swap(&lock, 0, 1) != 0)
          ; /* do nothing */

    /* critical section */

    lock = 0;

    /* remainder section */
}
```

Bounded-waiting Mutual Exclusion with compare-and-swap

```
while (true) {
   waiting[i] = true;
   key = 1;
   while (waiting[i] && key == 1)
      key = compare_and_swap(&lock,0,1);
   waiting[i] = false;
   /* critical section */
   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;
   if (j == i)
      lock = 0;
   else
      waiting[j] = false;
   /* remainder section */
}
```

Atomic Variables

- Typically, instructions such as compare-and-swap are used as building blocks for other synchronization tools.
- One tool is an **atomic variable** that provides *atomic* (uninterruptible) updates on basic data types such as integers and booleans.
- For example, the `increment()` operation on the atomic variable `sequence` ensures `sequence` is incremented without interruption:

```
increment(&sequence);
```

Atomic Variables

- The `increment()` function can be implemented as follows:

```
void increment(atomic_int *v)
{
     int temp;

     do {
          temp = *v;
     }
     while (temp != (compare_and_swap(v,temp,temp+1));
}
```

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Protect a critical section by first **`acquire()`** a lock then **`release()`** the lock
  - Boolean variable indicating if lock is available or not
- Calls to **`acquire()`** and **`release()`** must be atomic
  - Usually implemented via hardware atomic instructions such as compare-and-swap.
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

Solution to Critical-section Problem Using Locks

```
while (true) {
        acquire lock

        critical section

        release lock

        remainder section
}
```

Mutex Lock Definitions

- ```
  acquire() {
      while (!available)
          ; /* busy wait */
      available = false;;
  }
  ```

- ```
  release() {
      available = true;
  }
  ```

These two functions must be implemented atomically. Both test-and-set and compare-and-swap can be used to implement these functions.

Spinlock

- “spin” or “busy-wait”

- Good if delays are short
  - Fast critical section

- Can be implemented by in many ways
  - Test-and-set
  - Optimized (for performance) versions: test + test-and-set.
    - Loop on test (no bus contention)
    - When (possibly) free do test-and-set (with bus contention)

Test-and-set Spinlock

Package java.util.concurrent.atomic

```
class TASspinlock {
 AtomicBoolean state =
  new AtomicBoolean(false);

 void lock() {
  while (state.getAndSet(true)) {}
 }

 void unlock() {
  state.set(false);
 }}
```

Test-and-test-and-set Spinlock

Package java.util.concurrent.atomic

```
class TTASspinlock {
 AtomicBoolean state =
  new AtomicBoolean(false);

 void lock() {
  while (true) {
   while (state.get()) {}
   if (!state.getAndSet(true))
    return;
 }
}
```

OS/161 Spinlocks
(kern/thread/spinlock.c)

```
spinlock_acquire(struct spinlock *splk) {
  ...
  while (1) {
    /* Do test-test-and-set, that is, read first before
       doing test-and-set, to reduce bus contention.
       Test-and-set is a machine-level atomic operation
     */
    if (spinlock_data_get(&splk->splk_lock) != 0) {
       continue;
    }
    if (spinlock_data_testandset(&splk->splk_lock) != 0) {
       continue;
    }
    break;
  }
  ...
}
```

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 */
```

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore ***S*** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **`wait()`** and **`signal()`**
    - (Originally called **`P()`** and **`V()`**)
- Definition of the **`wait() operation`**

```
wait(S) {
    while (S <= 0)
       ; // busy wait
    S--;
}
```

- Definition of the **`signal() operation`**

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

Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can solve various synchronization problems
- Consider P_1 and P_2 that require S_1 to happen before S_2

  Create a semaphore "`synch`" initialized to 0

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

- Can implement a counting semaphore S as a binary semaphore

Semaphore Implementation

- Must guarantee that no two processes can execute the **`wait()`** and **`signal()`** on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the **`wait`** and **`signal`** code are placed in the critical section
  - Could now have **busy waiting** in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue
-

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

OS/161: 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. This is the way that the OS/161 kernel enforces mutual exclusion. There is a simple interface (splhigh(), spl0(), splx()) for disabling and enabling interrupts. See kern/arch/mips/include/spl.h .

OS/161 Semaphores (1.9x: interrupt based)

```
struct semaphore {
  char *name;
  volatile int count;
};

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

see

- kern/include/synch.h
- kern/thread/synch.c

OS/161 Semaphores: P()
(1.9x: interrupt based)

```
void P(struct semaphore *sem) {
   int spl;
   assert(sem != NULL);

   /* May not block in an interrupt handler.
    * For robustness, always check, even if we can actually
    * complete the P without blocking. */
   assert(in_interrupt==0);

   spl = splhigh();
   while (sem->count==0) {
     thread_sleep(sem);
   }
   assert(sem->count>0);
   sem->count--;
   splx(spl);
}
```

OS/161 Semaphores: V()
(1.9x: interrupt based)

```
void V(struct semaphore *sem)
{
  int spl;
  assert(sem != NULL);
  spl = splhigh();
  sem->count++;
  assert(sem->count>0);
  thread_wakeup(sem);
  splx(spl);
}
```

Implementation with no Busy waiting (Cont.)

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

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

Thread Blocking in OS/161

- OS/161 thread library functions:
  - `void thread_sleep(const void *addr)`
  blocks the calling thread on address addr
  - `void thread_wakeup(const void *addr)`
  unblock threads that are sleeping on address addr

- `thread_sleep()` is much like `thread_yield()`. The calling thread voluntarily gives up the CPU, the scheduler chooses a new thread to run, and dispatches the new thread. However
  - after a `thread_yield()`, the calling thread is *ready* to run again as soon as it is chosen by the scheduler
  - after a `thread_sleep()`, the calling thread is blocked, and should not be scheduled to run again until after it has been explicitly unblocked by a call to `thread_wakeup()`.

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 head */
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.

Problems with Semaphores

- Incorrect use of semaphore operations:
  - **`signal (mutex)`** …. **`wait (mutex)`**
  - **`wait (mutex)`** … **`wait (mutex)`**
  - Omitting of **`wait (mutex)`** and/or **`signal (mutex)`**
- These – and others – are examples of what can occur when sempahores and other synchronization tools are used incorrectly.

Semaphore limitation

- Can be hard to reason about synchronization

- Reason for waiting is embedded in P() (wait() operation)
  - Wait on counter
    - If (count == 0) sleep
  - Other waiting conditions not possible
    - E.g. if ((x==0) && (y>0 || z>0)) sleep

- Wait on condition
  - Condition checked outside P()
  - BUT checking needs nutual exclusion (extra lock/mutex/semaphore)
  - When sleeping (as a result of checking condition), extra lock/mutex/semaphore is OWNED
  - POSSIBLE DEADLOCK

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- Pseudocode syntax of a monitor:

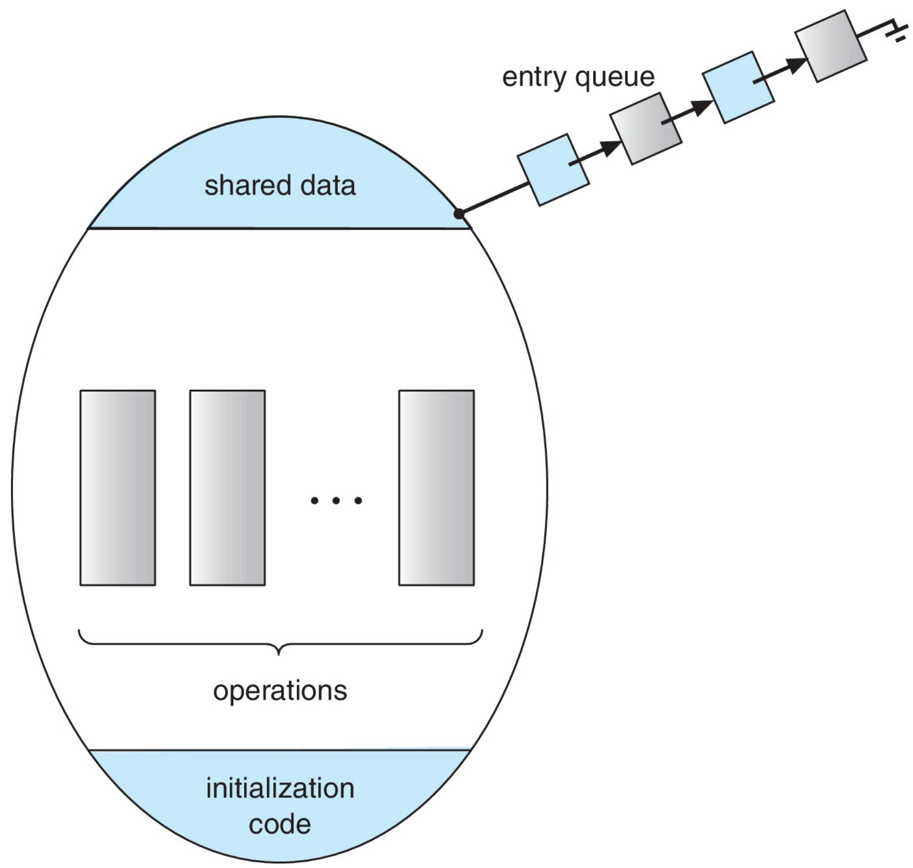
```
monitor monitor-name
{
  // shared variable declarations
  function P1 (…) { …. }

  function P2 (…) { …. }

  function Pn (…) {……}

  initialization code (…) { … }
}
```

Schematic view of a Monitor



Example: wait on condition

```
/* shared state vars with some initial value */
int x,y,z;
/* mutual exclusion for shared vars */
struct lock *mylock = lock_create("Mutex");
/* semaphore to wait if necessary */
struct semaphore *no_go = sem_create("MySem", 0);

compute_a_thing {
  lock_aquire(mylock); /* lock out others */
  /* compute new x, y, z */
  x = f1(x); y = f2(y); z = f3(z);
  if (x != 0 || (y <= 0 && z <= 0)) V(no_go);
  lock_release(mylock); /* enable others */
}
```

Example: wait on condition

```
use_a_thing {
  lock_aquire(mylock); /* lock out others */
  if(x == 0 && (y > 0 || z > 0))
    P(no_go);
  /* Now either x is non-zero or y and z are
     non-positive. In this state, it is safe to run
     "work" on x,y,z, which may also change them */
  work(x,y,z);
  lock_release(mylock); /* enable others */
}
```

Example: wait on condition

```
use_a_thing {
  lock_aquire(mylock); /* lock out others */
  if(x == 0 && (y > 0 || z > 0))
    P(no_go);
  /* Now either x is non-zero or y and z are
     non-positive. In this state, it is safe to run
     "work" on x,y,z, which may also change them */
  work(x,y,z);
  lock_release(mylock); /* enable others */
}
```

When waiting, mylock owned !
Other threads cannot modify x,y,z
and possibly unblock

Solution

- Release lock while waiting => NO DEADLOCK
  - release lock while waiting
  - wait/sleep
  - Get lock again when woken up

- NEED to test condition again (while instead of if) as condition can change between wake-up and re-acquire lock

- Problems:
  - Not clean, hard to read
  - Cannot wake up ALL waiters

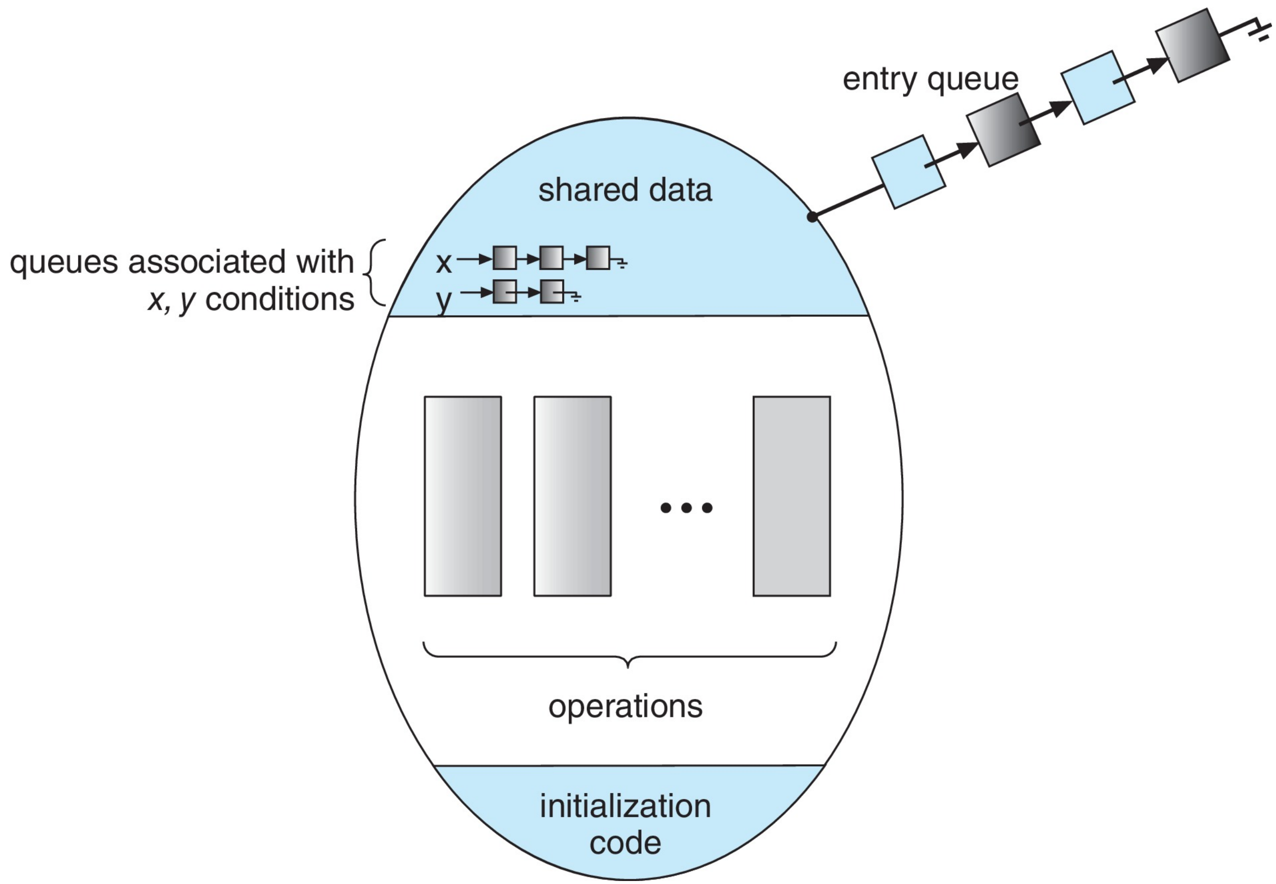
Example: wait on condition fixed

```
use_a_thing {
  lock_aquire(mylock); /* lock out others */
  while (x == 0 && (y > 0 || z > 0)) {
    lock_release(mylock); /* no deadlock */
    P(no_go);
    lock_aquire(mylock); /* lock for next test */
  }
  /* Now either x is non-zero or y and z are
     non-positive. In this state, it is safe to run
     “work” on x,y,z, which may also change them */
  work(x,y,z);
  lock_release(mylock); /* enable others */
}
```

Condition Variables

- `condition x, y;`
- Two operations are allowed on a condition variable:
  - **`x.wait()`** – a process that invokes the operation is suspended until **`x.signal()`**
  - **`x.signal()`** – resumes one of processes (if any) that invoked **`x.wait()`**
    - If no **`x.wait()`** on the variable, then it has no effect on the variable

Monitor with Condition Variables



Condition variables

- Abstract data type that encapsulates pattern of
  - release lock, sleep, re-acquire lock
- Each condition variable works together with a lock: condition variables are only used *from within the critical section that is protected by the lock*
- Internal data is just a queue of waiting threads
- Operations are (each of these is atomic) – in pseudocode:
  - cv_wait(struct cv *cv, struct lock *lock)
    Releases lock, waits, re-acquires lock before return
  - cv_signal(struct cv *cv)
    Wake one enqueued thread
  - cv_broadcast(struct cv *cv)
    Wakes all enqueued threads
- If no one waiting, cv_signal, cv_broadcast have no effect (different behaviour lock/semaphore/mutex)
- OS/161 signal and broadcast get lock parameter just to ckeck ownership:
  - cv_signal(struct cv *cv, struct lock *lock)
  - cv_broadcast(struct cv *cv, struct lock *lock)

Condition Variables Choices

- If process P invokes **`x.signal()`**, and process Q is suspended in **`x.wait()`**, what should happen next?
  - Both Q and P cannot execute in paralel. If Q is resumed, then P must wait
- Options include
  - **Signal and wait** – P waits until Q either leaves the monitor or it waits for another condition
  - **Signal and continue** – Q waits until P either leaves the monitor or it waits for another condition
  - Both have pros and cons – language implementer can decide
  - Monitors implemented in Concurrent Pascal compromise
    - P executing signal immediately leaves the monitor, Q is resumed
  - Implemented in other languages including Mesa, C#, Java

Using Condition variables with cv_signal

- **Always** used together with locks
  - The lock protects the shared data that is modified and tested when deciding whether to wait or signal/broadcast
- General Usage:

P_i

```
lock_acquire(lock);
while (condition not true)
{
  cv_wait(cond, lock);
}
... // do stuff
lock_release(lock);
```

P_j

```
lock_acquire(lock);
... // modify condition
cv_signal(cond);
lock_release(lock);
```

Using Condition variables with cv_broadcast

P_i

```
lock_acquire(lock);
while (condition_i not true){
  cv_wait(cond, lock);
}
... // do stuff
lock_release(lock);
```

P_j

```
lock_acquire(lock);
...
// modify conditions
// either for Pi or Pk
cv_broadcast(cond);
lock_release(lock);
```

P_k

```
lock_acquire(lock);
while (condition_k not true){
  cv_wait(cond, lock);
}
... // do stuff
lock_release(lock);
```

Example: wait on condition (condition variables)

```
/* shared state vars with some initial value */
int x,y,z;
/* mutual exclusion for shared vars */
struct lock *mylock = lock create("Mutex");
/* condition variable to wait if necessary */
struct cv *no_go = cv_create("CondV");

compute_a_thing {
  lock_aquire(mylock); /* lock out others */
  /* compute new x, y, z */
  x = f1(x); y = f2(y); z = f3(z);
  if (x != 0 || (y <= 0 && z <= 0)) cv_signal(no_go);
  lock_release(mylock); /* enable others */
}
```

Example: wait on condition (condition variables)

```
use_a_thing {
  lock_aquire(mylock); /* lock out others */
  while (x == 0 && (y > 0 || z > 0)) {
    cv_wait(no_go, mylock);
  }
  /* Now either x is non-zero or y and z are
     non-positive. In this state, it is safe to run
     “work” on x,y,z, which may also change them */
  work(x,y,z);
  lock_release(mylock); /* enable others */
}
```

OS/161 Wait Channels

- Same as condition variables with spinlocks.
  - Spinlock owned for short time
  - Nested or multiple spinlocks not allowed
- Kernel level synchronization objects
- Integrated with thread scheduling
  - Spinlock handled within thread_switch

OS/161 Wait Channels

```
wchan_sleep(struct wchan *wc, struct spinlock *lk) {

    /* may not sleep in an interrupt handler */
    KASSERT(!curthread->t_in_interrupt);

    /* must hold the spinlock */
    KASSERT(spinlock_do_i_hold(lk));

    /* must not hold other spinlocks */
    KASSERT(curcpu->c_spinlocks == 1);

    thread_switch(S_SLEEP, wc, lk);
    spinlock_acquire(lk);
}
```

OS/161 Wait Channels

```
wchan_wakeone(struct wchan *wc, struct spinlock *lk) {
  struct thread *target;
  KASSERT(spinlock_do_i_hold(lk));

  /* Grab a thread from the channel */
  target = threadlist_remhead(&wc->wc_threads);

  /* Note that thread_make_runnable acquires a runqueue
     lock while we're holding LK. This is ok; all
     spinlocks associated with wchans must come before the
     runqueue locks, as we also bridge from the wchan lock
     to the runqueue lock in thread_switch. */

  thread_make_runnable(target, false);
}
```

OS/161 Semaphores (2.0x: multicore)

```
struct semaphore {
  char *name;
  struct wchan *sem_wchan;
  struct spinlock sem_lock;
  volatile int count;
};

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

see

- kern/include/synch.h
- kern/thread/synch.c

OS/161 Semaphores: P()
(2.0x: multicore)

```
void P(struct semaphore *sem) {
  /* May not block in an interrupt handler. For robustness,
     always check, even if we can actually complete the
     without blocking. */
  KASSERT(curthread->t_in_interrupt == false);

  /* Use the semaphore spinlock to protect the wchan as well */
  spinlock_acquire(&sem->sem_lock);
  while (sem->sem_count == 0) {
    /* Note that we don't maintain strict FIFO ordering of
       threads going through the semaphore; */
    wchan_sleep(sem->sem_wchan, &sem->sem_lock);
  }
  KASSERT(sem->sem_count > 0);
  sem->sem_count--;
  spinlock_release(&sem->sem_lock);
}
```

OS/161 Semaphores: V() (2.0x: multicore)

```
void 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, &sem->sem_lock);

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

OS/161 TODO

- Kernel level synchronization objects to be implemented:
    - Locks
    - Condition Variables
- Strategy:
    - look at semaphores
    - Use spinlocks and wait channels

OS/161 TODO

- Kernel level synchronization objects to be implemented:
  - Locks
  - Condition Variables

- Strategy:
  - look at semaphores
  - Use spinlocks and wait channels

LAB 3!