Outline for Today

- Objectives:
  - To introduce the critical section problem.
  - To learn how to reason about the correctness of concurrent programs.
  - To present Linux kernel synchronization
- Administrative details:

Reasons for Explicitly Programming with Threads

(User-level Perspective – Birrell)

To capture naturally concurrent activities

- Waiting for slow devices
- Providing human users faster response.
- Shared network servers multiplexing among client requests (each client served by its own server thread)

To gain speedup by exploiting parallelism in hardware

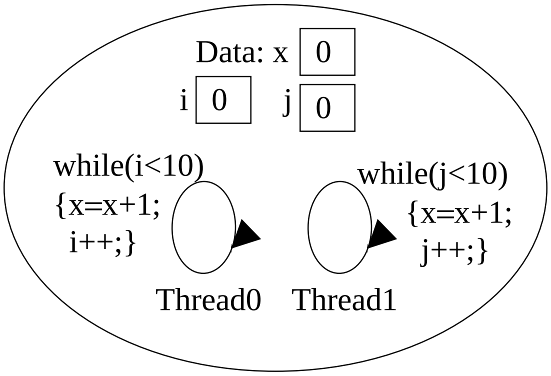
- Maintenance tasks performed "in the background"
- Multiprocessors
- Overlap the asynchronous and independent functioning of devices and users

Within a single user thread – signal handlers cause asynchronous control flow.

Concurrency from the Kernel Perspective

- Kernel preemption – scheduler can preempt task executing in kernel.
- Interrupts occurring – asynchronously invoking handler that disrupts the execution flow.
- Sleeping to wait for events.
- Support for SMP multiprocessors – true concurrency of code executing on shared memory locations.

The Trouble with Concurrency in Threads...



```
while(i<10)
{x=x+1;
 i++;}
```

```
while(j<10)
 {x=x+1;
  j++;}
```

What is the value of x when both threads leave this while loop?

Range of Answers

Process 0	Process1
LD x // x currently 0	
	LD x // x currently 0
	Add 1
	ST x // x now 1
	Do 8 more full loops // x = 9
Add 1	
ST x // x now 1, stored over 9	
	LD x // x now 1
Do 9 more full loops // leaving x at 10	
	Add 1
	ST x // x = 2 stored over 10

Reasoning about Concurrency

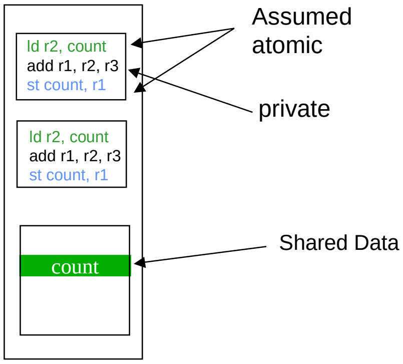
- What unit of work can be performed without interruption? *Indivisible* or *atomic* operations.
- *Interleavings* - possible execution sequences of operations drawn from all threads.
- *Race condition* - final results depend on ordering and may not be “correct”.

The Trouble with Concurrency

- Two threads (T1,T2) in one address space or two processes in the kernel
- One counter (shared)

Time

T1	T2	count
ld (count)		
add		
switch		
	ld (count)	
	add	
	st (count+1)	
		count+1
	switch	
st (count+1)		count+1



Desired: Atomic Sequence of Instructions

Time

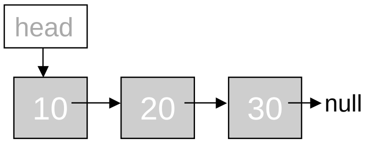
T1	T2	count
begin atomic		
ld (count)		
add		
switch		
	begin atomic	
st (count+1)	*wait*	count+1
end atomic		
switch		
	ld (count+1)	
	add	
	st (count+2)	count+2
	end atomic	

- Atomic Sequence
  - *Appears* to execute to completion without any intervening operations

Unprotected Shared Data

```
void threadcode( )
{
   int i;
   long key;
   for (i=0; i<20; i++){
        key = rand();
        SortedInsert (key);}
   for (i=0; i<20; i++){
        key = SortedRemove();
        print (key); }
}
```

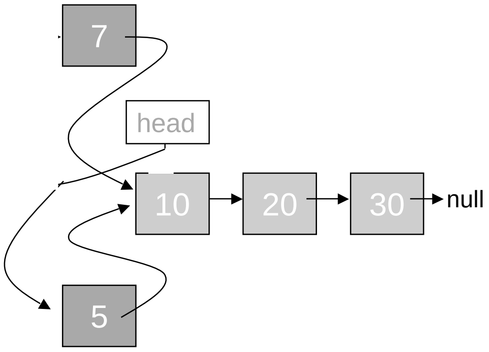
private



What can happen here?

Unprotected Shared Data

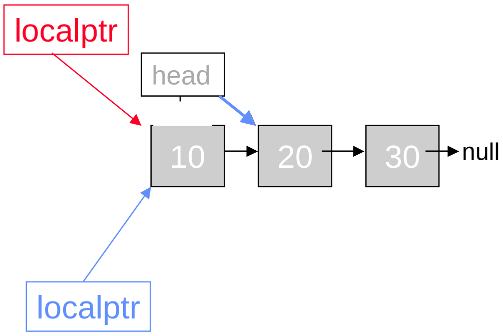
- 2 concurrent SortedInserts with keys 5 and 7.



What can happen here?

Unprotected Shared Data

- 2 concurrent SortedInserts with keys 5 and 7.
- 2 concurrent SortedRemoves



What can happen here?

Critical Sections

- If a sequence of non-atomic operations must be executed *as if* it were atomic in order to be correct, then we need to provide a way to constrain the possible interleavings
  - ***Critical sections*** are defined as code sequences that contribute to “bad” race conditions.
  - Synchronization is needed around such critical sections.
- ***Mutual Exclusion*** - goal is to ensure that critical sections execute atomically w.r.t. related critical sections in other threads or processes.

The Critical Section Problem

Each process follows this template:

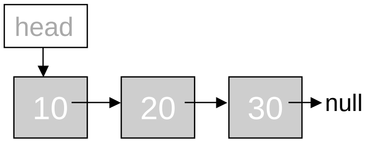
```
while (1)
{ ...other stuff...   //processes in here shouldn't stop others
   enter_region( );
   critical section
   exit_region( );
}
```

Problem with this definition:
It focuses on code
not shared data
that needs protecting!

The problem is to implement enter_region and exit_region to ensure mutual exclusion with some degree of fairness.

Temptation to Protect Critical Sections (Badly)

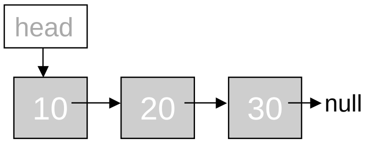
```
void threadcode( )
{
   int i;
   long key;
   for (i=0; i<20; i++){
         key = rand();
         Acquire(insertmutex);
         SortedInsert (key);
         Release(insertmutex);
   }
   for (i=0; i<20; i++){
         Acquire(removemutex);
         key = SortedRemove();
         Release(removemutex);
         print (key); }
}
```



Focus on the data!

Temptation to Protect Critical Sections (Badly)

```
void threadcode( )
{
    int i;
    long key;
    for (i=0; i<20; i++){
           key = rand();
           Acquire(listmutex);
           SortedInsert (key);
           Release(listmutex);
    }
    for (i=0; i<20; i++){
           Acquire(listmutex);
           key = SortedRemove();
           Release(listmutex);
           print (key); }
}
```


head

10 20 30 null

Focus on the data!

Yet Another Example

Problem: Given arrays C[0:x,0:y], A [0:x,0:y], and B [0:x,0:y]. Use n threads to update each element of C to the sum of A and B and then the last thread returns the average value of all C elements.

Design Alternatives

- Static partitioning of arrays

```
for (i = lowi; i < highi; i++)
   for (j = lowj; j < highj; j++)
     {C[i,j] = A[i,j] + B[i,j];
      sum = sum + C[i,j]; }
```

- Static partitioning of arrays

```
for (i = lowi; i < highi; i++)
   for (j = lowj; j < highj; j++)
     {C[i,j] = A[i,j] + B[i,j];
       privatesum = privatesum +
       C[i,j]; }
sum = sum + privatesum;
```

lowi = 0
highi = n/2-1
lowj = 0
highj = n/2-1
	lowi = n/2
	highi = n-1
	lowj = 0
	highj = n/2-1

lowi = 0
highi = n/2-1
lowj = n/2
highj = n-1
	lowi = n/2
	highi = n-1
	lowj = n/2
	highj = n-1

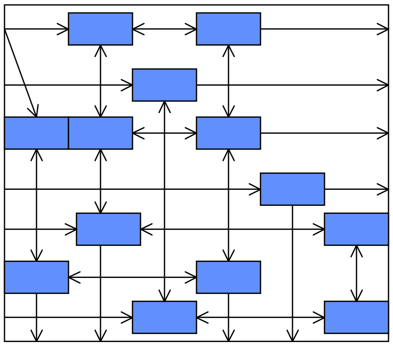
C

sum

Design Alternatives

- Dynamic partitioning of arrays

```
while (elements_remain(&i,&j))
{C[i,j] = A[i,j] + B[i,j];
 sum = sum + C[i,j]; }
```



C

sum

Implementation Options for Mutual Exclusion

- Disable Interrupts
- Use atomic operations (read-mod-write instr.)
- Busywaiting solutions - spinlocks
  - execute a tight loop if critical section is busy
  - benefits from specialized atomic instructions
- Blocking synchronization
  - sleep (enqueued on wait queue) while C.S. is busy

Synchronization primitives (abstractions, such as locks) which are provided by a system may be implemented with some combination of these techniques.

The Critical Section Problem

while (1)

{ **...***other stuff***...**

enter_region( );

critical section – anything that touches a particular set of shared data

exit_region( );

}

Critical Data

- Goal in solving the critical section problem is to build synchronization so that the sequence of instructions that can cause a race condition are executed *AS IF* they were indivisible
  - “Other stuff” code that does not touch the critical data associated with a critical section can be interleaved with the critical section code.
  - Code from a critical section involving data x can be interleaved with code from a critical section associated with data y.

The Critical Section Problem

```
while (1)
{ ...other stuff...
    local_irq_save(flags);
    critical section – anything that touches a particular
    set of shared data
    local_irq_restore(flags);
}
```

Overkill on UP
Insufficient for SMP

Disabling Preemption

while (1)

{ **...***other stuff***...**

preempt_disable();

critical section – per-processor data

preempt_enable();

}

Milder impact on UP

Atomic Operations (Integer)

- Special data type `atomic_t`
  - Prevent misuse and compiler optimizations
  - Only 24 bit values (it's SPARC's fault)
  - `atomic_t u = ATOMIC_INIT (0);`
- Selected operations (see p. 119)

```
o atomic_read     o atomic_sub_and_test
o atomic_set      o atomic_add_negative
o atomic_add      o atomic_inc_and_test
o atomic_inc
```

Atomic Operations (Bitwise)

- No special data type – take pointer and bit number as arguments. Bit 0 is least sign. bit.
- Selected operations

```
o set_bit          o test_and_set_bit
o clear_bit        o test_and_clear_bit
o change_bit       o test_and_change_bit
o test_bit
```

Uses of Atomic Operations

```
static int x = 0;

threadcode()
{
int j = 0;
while(j<10)
//10 times per thread
      {x=x+1;
        j++;}
}
```

```
atomic_t x ATOMIC_INIT (0);

threadcode()
{
int j=0;
while(j<10)
//10 times per thread
    {atomic_inc(&x);
     j++;}
}
```

Uses of Atomic Operations

```
static int x = 0;
static int j = 11;

threadcode()
{
while((--j)!=0)
// 10 times in all
    x=x+1;
}
```

```
atomic_t x ATOMIC_INIT (0);
atomic_t j ATOMIC_INIT (11);

threadcode()
{
while(!atomic_dec_and_test(&j))
//10 times in all
    atomic_inc(&x);
}
```

Uses of Atomic Operations

while (1)

{ **...***other stuff***...**

```
//homegrown spinlock
while(test_and_set_bit(0, &busy);
```

critical section – anything that touches a particular set of shared data

```
clear_bit(0, &busy );
```

}

Linux Kernel Spinlocks

```
spinlock_t busy = SPIN_LOCK_UNLOCKED;
  while (1)
  { ...other stuff...

      //canned spinlock
      spin_lock(&busy);

    critical section – anything that touches a particular
    set of shared data

      spin_unlock(&busy );

  }
```

Pros and Cons of Busywaiting

- Key characteristic - the “waiting” process is actively executing instructions in the CPU and using memory cycles.
- Appropriate when:
  - High likelihood of finding the critical section unoccupied (don’t take context switch just to find that out) or estimated wait time is very short
  - You have a processor all to yourself
  - In interrupt context
- Disadvantages:
  - Wastes resources (CPU, memory, bus bandwidth)

Spinlock Subtleties

- Using spinlock in interrupt handlers – disable local interrupts before obtaining lock
- Saves (and restores) IRQ-enable state.
  Disables while holding lock

```
spin_lock_irqsave (&lockvar, flags)
spin_unlock_irqrestore (&lockvar, flags)
spin_lock_irq (&lockvar)
spin_unlock_irq(&lockvar)
```

- Disabling bottom halves

`spin_lock_bh()` and `spin_unlock_bh()`

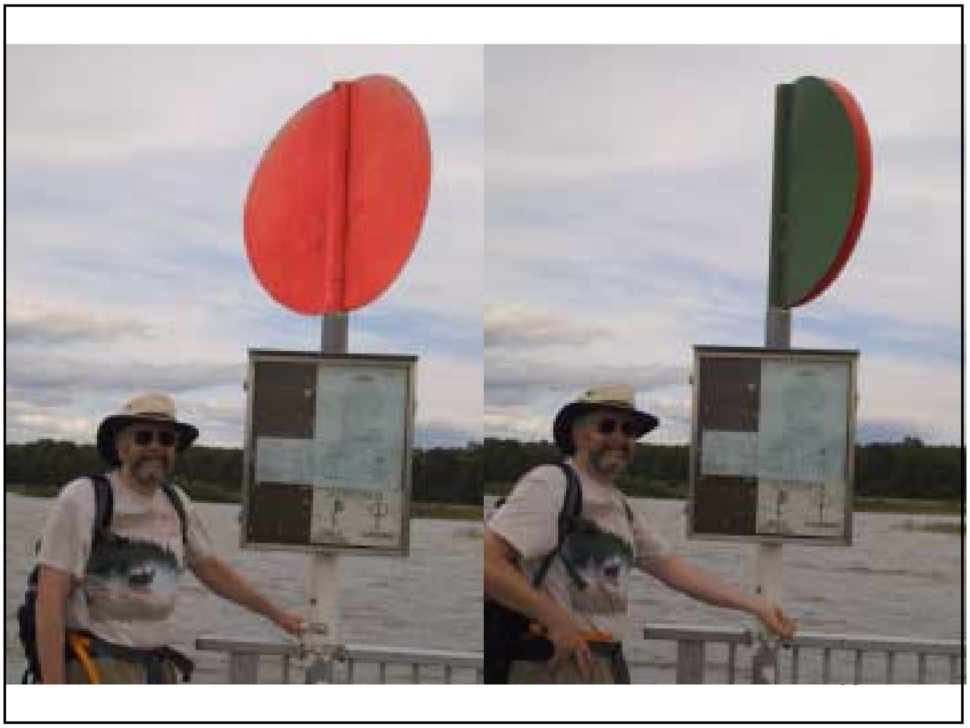
Pros and Cons of Blocking

- Sleeping processes/threads don’t consume CPU cycles
- Appropriate: when the cost of a system call is justified by expected waiting time
  - High likelihood of contention for lock
  - Long critical sections
- Disadvantage: context switch → overhead

Semaphores

- Well-known synchronization abstraction
- Defined as a non-negative integer with two atomic operations

```
P(s) - [wait until s > 0; s--]   or down(s)
V(s) - [s++]                      or up(s)
```



Semaphore Usage

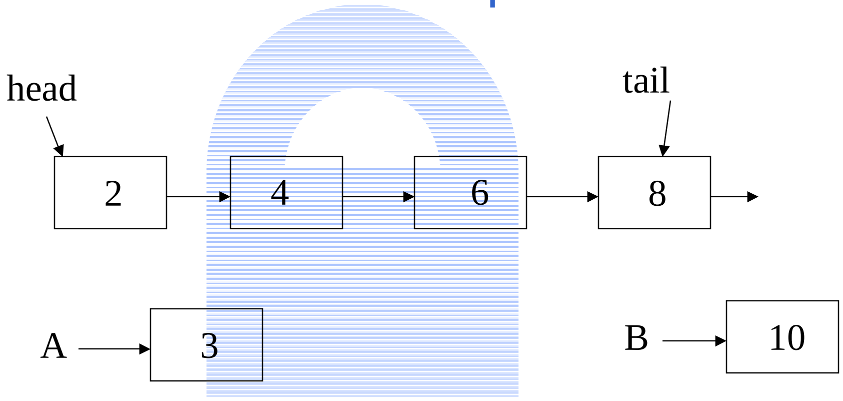
- Binary semaphores can provide mutual exclusion – mutex (solution to critical section problem)
- Counting semaphores can represent a resource with multiple instances (e.g. solving producer/consumer problem)
- Signaling events (persistent events that stay relevant even if nobody listening right now)

The Critical Section Problem

```
static DECLARE_SEMAPHORE_GENERIC(mutex,1)  or
static DECLARE_MUTEX(mutex)
    while (1)
    { ...other stuff...
        down_interruptable(&mutex);

        critical section

        up(&mutex);
    }
```

Fill in the boxes

Lock Granularity – how much should one lock protect?



head

tail

2

4

6

8

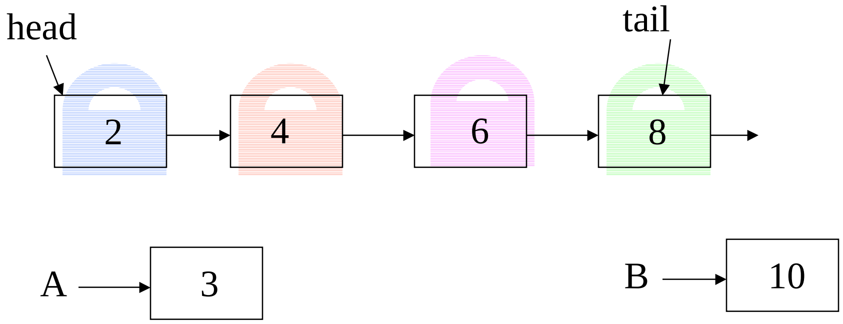
A

3

B

10

Lock Granularity – how much should one lock protect?



Concurrency vs. overhead
Complexity threatens correctness

Optimistic Locking – Seqlocks

- Sequence counter incremented on write
- Compare counter before and after a read
- Even counter value means data is stable
- Odd counter value means write in progress

Writes

```
write_seqlock(&lock);
// do write, lock is odd
write_sequnlock(&lock);
// write complete,
// lock is even
```

Reads

```
do {
    old=read_seqbegin(&lock);
     //reading data
}
while (read_seqretry(&lock, old));
```

Peterson's Algorithm for 2 Process Mutual Exclusion

- enter_region:

```
needin [me] = true;
turn = you;
while (needin [you] && turn == you) {no_op};
```

- exit_region:

```
needin [me] = false;
```

Based on the assumption of atomic ld/st operations

Interleaving of Execution of 2 Threads (blue and green)

```
enter_region:
    needin [me] = true;
    turn = you;
    while (needin [you] &&
       turn == you) {no_op};
Critical Section
exit_region:
    needin [me] = false;
```

```
enter_region:
    needin [me] = true;
    turn = you;
    while (needin [you] &&
       turn == you) {no_op};
Critical Section
exit_region:
    needin [me] = false;
```

needin [blue] = true;

needin [green] = true;

turn = green;

turn = blue;

while (needin [green] && turn == green)

Critical Section

while (needin [blue] && turn == blue){no_op};

while (needin [blue] && turn == blue){no_op};

needin [blue] = false;

while (needin [blue] && turn == blue)

Critical Section

needin [green] = false;

Peterson’s Algorithm for 2 Process Mutual Exclusion

- enter_region:

```
needin [me] = true;          mb();
turn = you;
while (needin [you] && turn == you) {no_op};
```

- exit_region:

```
needin [me] = false;
```

Barriers

- rmb – prevents loads being reordered across barrier
- wmb – prevents reordering stores
- mb – both loads and stores
- read_barrier_depends – data-dependent loads
- SMP versions of above – compiles to barrier on UP
- barrier – prevents compiler optimizations from causing the reordering

Classic Synchronization Problems

There are a number of “classic” problems that represent a class of synchronization situations

- Critical Section problem
- Producer/Consumer problem
- Reader/Writer problem
- 5 Dining Philosophers

Why? Once you know the “generic” solutions, you can recognize other special cases in which to apply them (e.g., this is just a version of the reader/writer problem)

Producer / Consumer

Producer:

```
while(whatever)
{   locally generate item

    fill empty buffer with item

}
```

Consumer:

```
while(whatever)
{
    get item from full buffer

    use item
}
```

Producer / Consumer

(with Counting Semaphores*)

Producer:

```
while(whatever)
{   locally generate item
      P(emptybuf);
    fill empty buffer with item
      V(fullbuf);
}
```

Consumer:

```
while(whatever)
{     P(fullbuf);
    get item from full buffer
      V(emptybuf);
    use item
}
```

Semaphores: emptybuf initially N; fullbuf initially 0;

*not Linux syntax

What does it mean that Semaphores have *persistence*?

Tweedledum and Tweedledee Problem

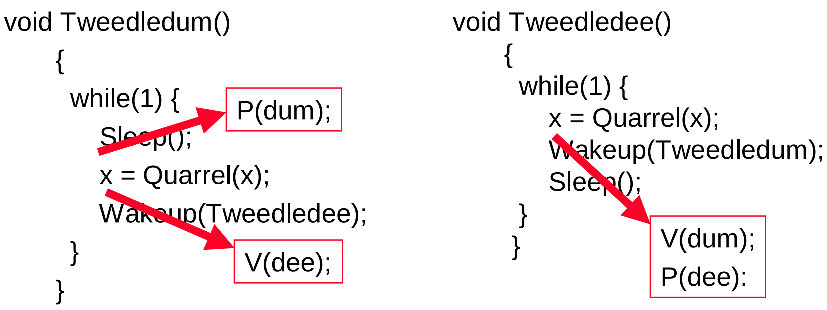
- Separate threads executing their respective procedures. The code below is intended to cause them to forever take turns exchanging insults through the shared variable X in strict alternation.
- The Sleep() and Wakeup() routines operate as follows:
  - Sleep blocks the calling thread,
  - Wakeup unblocks a specific thread if that thread is blocked, otherwise its behavior is unpredictable
- Linux: wait_for_completion() and complete()

The code shown above exhibits a well-known synchronization flaw. Outline a scenario in which this code would fail, and the outcome of that scenario

```
void Tweedledum()                    void Tweedledee()
    {                                    {
      while(1) {                           while(1) {
         Sleep();                             x = Quarrel(x);
         x = Quarrel(x);                      Wakeup(Tweedledum);
         Wakeup(Tweedledee);                  Sleep();
      }                                    }
    }                                     }
```

Lost Wakeup:
If dee goes first to sleep, the wakeup is lost (since dum isn’t sleeping yet). Both sleep forever.

Show how to fix the problem by replacing the Sleep and Wakeup calls with semaphore P (down) and V (up) operations.



```
void Tweedledum()
    {
      while(1) {
         Sleep();
         x = Quarrel(x);
         Wakeup(Tweedledee);
      }
    }
```

P(dum);

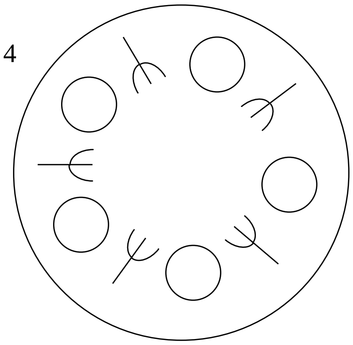
V(dee);

```
void Tweedledee()
    {
      while(1) {
         x = Quarrel(x);
         Wakeup(Tweedledum);
         Sleep();
      }
    }
```

V(dum);
P(dee):

```
semaphore dee = 0;
semaphore dum = 0;
```

5 Dining Philosophers



```
while(food available)
{pick up 2 adj. forks;
  eat;
  put down forks;
  think awhile;
}
```

Template for Philosopher

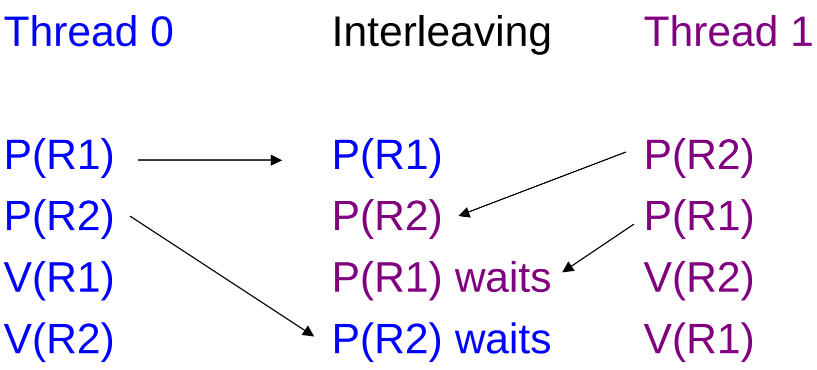
```
while (food available)
{                                    /*pick up forks*/

   eat;
                                     /*put down forks*/

   think awhile;
}
```

Naive Solution

```
while (food available)
{                                    /*pick up forks*/
      P(fork[left(me)]);
      P(fork[right(me)]);
   eat;
                                     /*put down forks*/
       V(fork[left(me)]);
       V(fork[right(me)]);
   think awhile;
}
```

Does this work?

Simplest Example of Deadlock



Thread 0	Interleaving	Thread 1
P(R1)	P(R1)	P(R2)
P(R2)	P(R2)	P(R1)
V(R1)	P(R1) waits	V(R2)
V(R2)	P(R2) waits	V(R1)

R1 and R2 initially 1 (binary semaphore)

Conditions for Deadlock

- Mutually exclusive use of resources
  - Binary semaphores R1 and R2
- Circular waiting
  - Thread 0 waits for Thread 1 to V(R2) and Thread 1 waits for Thread 0 to V(R1)
- Hold and wait
  - Holding either R1 or R2 while waiting on other
- No pre-emption
  - Neither R1 nor R2 are removed from their respective holding Threads.

Philosophy 101
(or why 5DP is interesting)

- How to eat with your Fellows without causing ***Deadlock.***
  - Circular arguments (the circular wait condition)
  - Not giving up on firmly held things (no preemption)
  - Infinite patience with Half-baked schemes (hold some & wait for more)
- Why ***Starvation*** exists and what we can do about it.

Dealing with Deadlock

It can be ***prevented*** by breaking one of the prerequisite conditions:

- Mutually exclusive use of resources
  - Example: Allowing shared access to read-only files (readers/writers problem)
- circular waiting
  - Example: Define an ***ordering*** on resources and acquire them in order
- hold and wait
- no pre-emption

Circular Wait Condition

```
while (food available)
{   if (me == 0) {P(fork[left(me)]); P(fork[right(me)]);}
    else {(P(fork[right(me)]); P(fork[left(me)]); }
    eat;
    V(fork[left(me)]); V(fork[right(me)]);
    think awhile;
}
```

Hold and Wait Condition

```
while (food available)
{   P(mutex);
    while (forks [me] != 2)
            {blocking[me] = true; V(mutex); P(sleepy[me]); P(mutex);}
    forks [leftneighbor(me)] --;  forks [rightneighbor(me)]--;
    V(mutex):
    eat;
    P(mutex); forks [leftneighbor(me)] ++;  forks [rightneighbor(me)]++;
    if (blocking[leftneighbor(me)]) {blocking [leftneighbor(me)] = false;
    V(sleepy[leftneighbor(me)]); }
    if (blocking[rightneighbor(me)]) {blocking[rightneighbor(me)] = false;
    V(sleepy[rightneighbor(me)]); }     V(mutex);
    think awhile;
}
```

Starvation

The difference between deadlock and starvation is subtle:

- Once a set of processes are deadlocked, there is no future execution sequence that can get them out of it.
- In starvation, there does exist some execution sequence that is favorable to the starving process although there is no guarantee it will ever occur.
- Rollback and Retry solutions are prone to starvation.
- Continuous arrival of higher priority processes is another common starvation situation.

Readers/Writers Problem

Synchronizing access to a file or data record in a database such that any number of threads requesting read-only access are allowed but only one thread requesting write access is allowed, excluding all readers.

Template for Readers/Writers

```
Reader()
{while (true)
   {
      /*request r access*/
   read
      /*release r access*/
   }
}
```

```
Writer()
{while (true)
   {
      /*request w access*/
   write
      /*release w access*/
   }
}
```

Reader/Writer Spinlocks

- Class of reader/writer problems
- Multiple readers OK
- Mutual exclusion for writers
- No upgrade from reader lock to writer lock
- Favors readers – starvation of writers possible

```
rwlock_t
read_lock,read_unlock
read_lock_irq  // also unlock
read_lock_irqsave
read_unlock_irqrestore
write_lock,write_unlock
//_irq,_irqsave,_irqrestore
write_trylock
rw_is_locked
```

Reader/Writer Semaphores

- All reader / writer semaphores are mutexes (usage count 1)
- Multiple readers, solo writer
- Uninterruptible sleep
- Possible to downgrade writer to reader

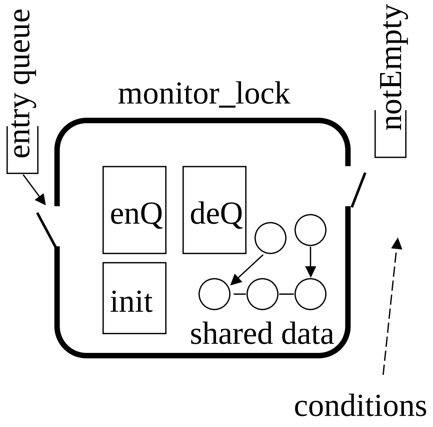
```
down_read
down_write
up_read
up_write
downgrade_writer
down_read_trylock
down_write_trylock
```

Birrell paper: SRC Thread Primitives

- SRC thread primitives
  - Thread = Fork (procedure, args)
  - result = Join (thread)
  - LOCK mutex DO critical section END
  - Wait (mutex, condition)
  - Signal (condition)
  - Broadcast (condition)
  - Acquire (mutex), Release (mutex) //more dangerous

Monitor Abstraction

- Encapsulates shared data and operations with mutual exclusive use of the object (an associated *lock*).
- Associated *Condition Variables* with operations of *Wait* and *Signal.*

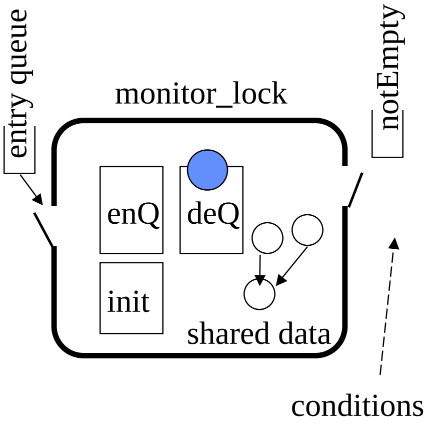


Condition Variables

- We build the monitor abstraction out of a lock (for the mutual exclusion) and a set of associated condition variables.
- *Wait on condition*: releases lock held by caller, caller goes to sleep on condition's queue.
  When awakened, it must reacquire lock.
- *Signal condition*: wakes up one waiting thread.
- *Broadcast*: wakes up all threads waiting on this condition.

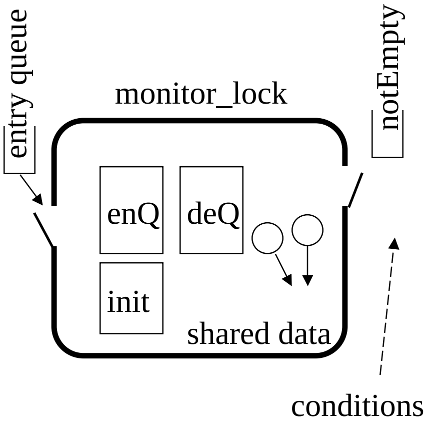
Monitor Abstraction

```
EnQ:{acquire (lock);
    if (head == null)
        {head = item;
        signal (lock, notEmpty);}
    else tail->next = item;
    tail = item;
    release(lock);}
deQ:{acquire (lock);
    if (head == null)
        wait (lock, notEmpty);
     item = head;
     if (tail == head) tail = null;
     head=item->next;
    release(lock);}
```



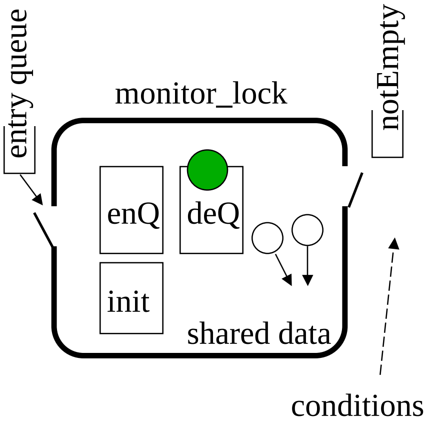
Monitor Abstraction

```
EnQ:{acquire (lock);
    if (head == null)
        {head = item;
        signal (lock, notEmpty);}
    else tail->next = item;
    tail = item;
    release(lock);}
deQ:{acquire (lock);
    if (head == null)
        wait (lock, notEmpty);
     item = head;
     if (tail == head) tail = null;
     head=item->next;
    release(lock);}
```



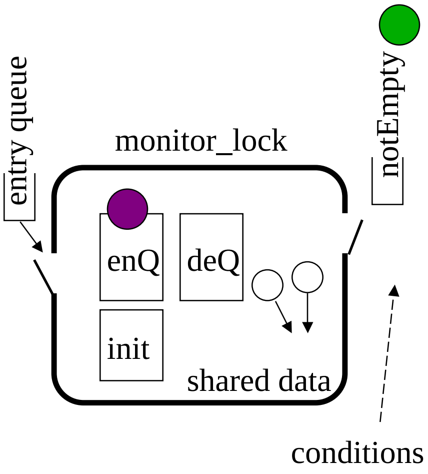
Monitor Abstraction

```
EnQ:{acquire (lock);
     if (head == null)
          {head = item;
          signal (lock, notEmpty);}
     else tail->next = item;
     tail = item;
     release(lock);}
deQ:{acquire (lock);
     if (head == null)
          wait (lock, notEmpty);
      item = head;
      if (tail == head) tail = null;
      head=item->next;
     release(lock);}
```



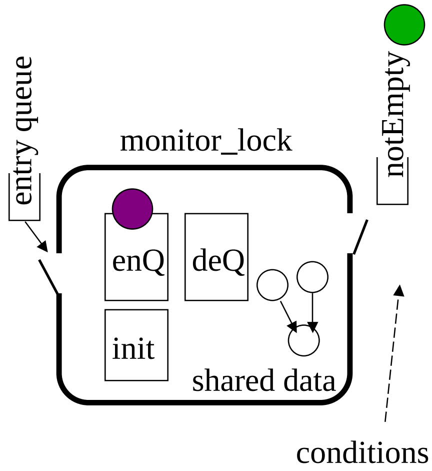
Monitor Abstraction

```
EnQ:{acquire (lock);
     if (head == null)
          {head = item;
          signal (lock, notEmpty);}
     else tail->next = item;
     tail = item;
     release(lock);}
deQ:{acquire (lock);
     if (head == null)
          wait (lock, notEmpty);
      item = head;
      if (tail == head) tail = null;
      head=item->next;
     release(lock);}
```



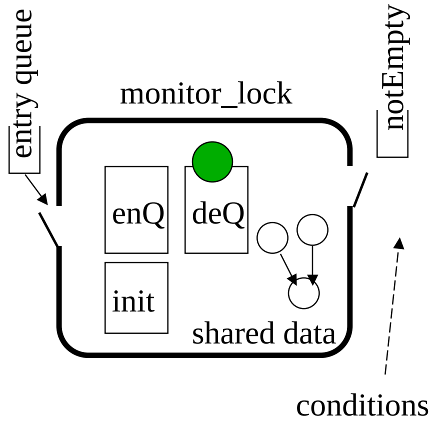
Monitor Abstraction

```
EnQ:{acquire (lock);
     if (head == null)
          {head = item;
          signal (lock, notEmpty);}
     else tail->next = item;
     tail = item;
     release(lock);}
deQ:{acquire (lock);
     if (head == null)
          wait (lock, notEmpty);
      item = head;
      if (tail == head) tail = null;
      head=item->next;
     release(lock);}
```



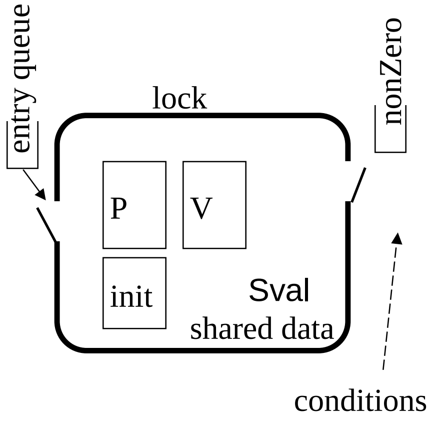
Monitor Abstraction

```
EnQ:{acquire (lock);
     if (head == null)
          {head = item;
          signal (lock, notEmpty);}
     else tail->next = item;
     tail = item;
     release(lock);}
deQ:{acquire (lock);
     while (head == null)
          wait (lock, notEmpty);
      item = head;
      if (tail == head) tail = null;
      head=item->next;
     release(lock);}
```



P&V using Locks & CV (Monitor)

```
P: {acquire (lock);
     while (Sval == 0)
          wait (lock, nonZero);
     Sval = Sval –1;
     release(lock);}

V: {acquire (lock);
     Sval = Sval + 1;
     signal (lock, nonZero);
     release(lock);}
```

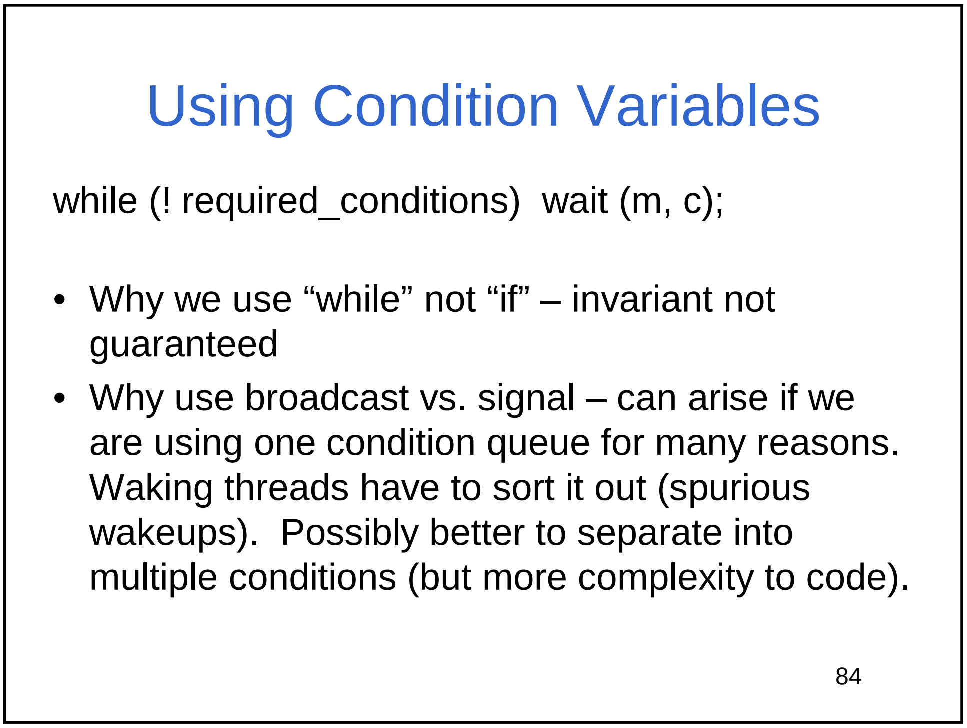


Design Decisions / Issues

- Locking overhead (granularity)
- Broadcast vs. Signal
- Nested lock/condition variable problem

```
            LOCK a DO
Unseen  ⎧       LOCK b DO
in      ⎨           while (not_ready) wait (b, c) //releases b not a
call    ⎩       END
            END
```

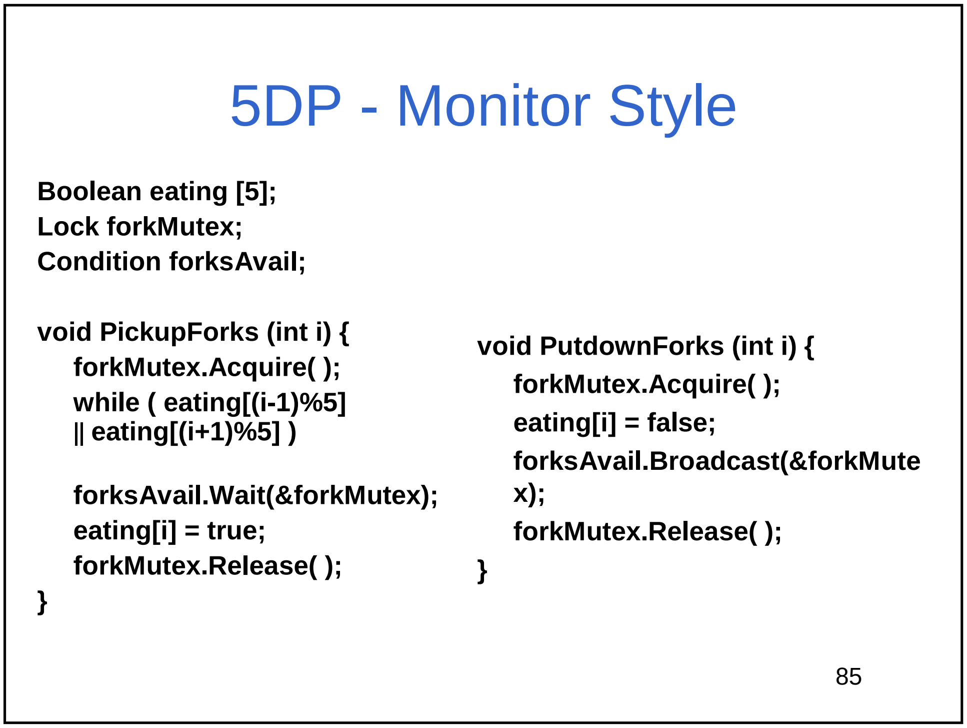
- My advice – correctness first!



Using Condition Variables

while (! required_conditions) wait (m, c);

- Why we use “while” not “if” – invariant not guaranteed
- Why use broadcast vs. signal – can arise if we are using one condition queue for many reasons. Waking threads have to sort it out (spurious wakeups). Possibly better to separate into multiple conditions (but more complexity to code).



5DP - Monitor Style

```
Boolean eating [5];
Lock forkMutex;
Condition forksAvail;

void PickupForks (int i) {
    forkMutex.Acquire( );
    while ( eating[(i-1)%5]
    || eating[(i+1)%5] )

    forksAvail.Wait(&forkMutex);
    eating[i] = true;
    forkMutex.Release( );
}
```

```
void PutdownForks (int i) {
    forkMutex.Acquire( );
    eating[i] = false;
    forksAvail.Broadcast(&forkMute
    x);
    forkMutex.Release( );
}
```

What about this?

```
while (food available)
{  forkMutex.Acquire( );
   while (forks [me] != 2) {blocking[me]=true;
           forkMutex.Release( ); sleep( ); forkMutex.Acquire( );}
   forks [leftneighbor(me)]--;  forks [rightneighbor(me)]--;
    forkMutex.Release( ):
   eat;
    forkMutex.Acquire( );
   forks[leftneighbor(me)] ++;  forks [rightneighbor(me)]++;
   if (blocking[leftneighbor(me)] || blocking[rightneighbor(me)])
           wakeup ( );  forkMutex.Release( );
   think awhile;
}
```

Template for Readers/Writers

```
Reader()                          Writer()
{while (true)                     {while (true)
   {                                 {
      startRead();                      startWrite();
   read                              write
      endRead();                        endWrite();
   }                                 }
}                                 }
```

R/W - Monitor Style

```
Boolean busy = false;
int numReaders = 0;
Lock filesMutex;
Condition OKtoWrite, OKtoRead;

void startRead () {
    filesMutex.Acquire( );
    while ( busy )
        OKtoRead.Wait(&filesMutex);
    numReaders++;
    filesMutex.Release( );}

void endRead () {
    filesMutex.Acquire( );
    numReaders--;
    if (numReaders == 0)
      OKtoWrite.Signal(&filesMutex);
    filesMutex.Release( );}

void startWrite() {
    filesMutex.Acquire( );
    while (busy || numReaders != 0)
          OKtoWrite.Wait(&filesMutex);
    busy = true;
    filesMutex.Release( );}

void endWrite() {
    filesMutex.Acquire( );
    busy = false;
    OKtoRead.Broadcast(&filesMutex);
    OKtoWrite.Signal(&filesMutex);
    filesMutex.Release( );}
```

Issues

- Locking overhead (granularity)
- Broadcast vs. Signal and other causes of spurious wakeups
- Nested lock/condition variable problem

```
            LOCK a DO
Unseen          LOCK b DO
in                  while (not_ready) wait (b, c) //releases b not a
call            END
            END
```

- Priority inversions

Spurious Wakeups

```
while (! required_conditions)  wait (m, c);
```

- Why we use “while” not “if” – invariant not guaranteed
- Why use broadcast – using one condition queue for many reasons. Waking threads have to sort it out. Possibly better to separate into multiple conditions (more complexity to code)

Tricks (mixed syntax)

```
if (some_condition) // as a hint
{
   LOCK m DO
        if (some_condition) //the truth
        {stuff}
   END
}
```

Cheap to get info but must check for correctness; always a slow way

More Tricks

General pattern:

```
while (! required_conditions)  wait (m, c);
```

Broadcast works because waking up too many is OK (correctness-wise) although a performance impact.

```
LOCK m DO
     …
     deferred_signal = true;
END
if (deferred_signal) signal (c);
```

Spurious lock conflicts caused by signals inside critical section and threads waking up to test mutex before it gets released.

Alerts

Thread state contains flag, alert-pending

Exception alerted

Alert (thread)

- alert-pending to true, wakeup a waiting thread

AlertWait (mutex, condition)

- if alert-pending set to false and raise exception
- else wait as usual

Boolean b = TestAlert()

- tests and clear alert-pending

```
TRY
   while (empty)
     AlertWait (m,
   nonempty); return
   (nextchar());
EXCEPT
   Thread.Alerted:
        return (eof);
```

Using Alerts

```
sibling = Fork (proc, arg);
while (!done)
{ done = longComp();
   if (done) Alert (sibling);
   else done = TestAlert();
}
```

Wisdom

Do s

- Reserve using alerts for when you don’t know what is going on
- Only use if you forked the thread
- Impose an ordering on lock acquisition
- Write down invariants that should be true when locks aren’t being held

Don’t s

- Call into a different abstraction level while holding a lock
- Move the “last” signal beyond scope of Lock
- Acquire lock, fork, and let child release lock
- Expect priority inheritance since few implementations
- Pack data and expect fine grain locking to work