R

بسم الله الرحمن الرحیم

سیستم عامل

جلسه هفتم – سمافور و مسائل همزمانی

آنچه گذشت

پیاده‌سازی mutex

Mutex Lock Operations

- Lock (*mutex*)
  - *Acquire the lock if it is free … and continue*
  - *Otherwise wait until it can be acquired*
- Unlock (*mutex*)
  - *Release the lock*
  - *If there are waiting threads wake one up*

Peterson’s Algorithm for Thread *i*

```
while (true){

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

        /* critical section */

    flag[i] = false;

    /* remainder section */
}
```

Test-and-Set-Lock (TSL) Instruction

- Test-and-set-lock does the following *atomically*:
  - *Get the (old) value*
  - *Set the lock to TRUE*
  - *Return the old value*

If *the returned value was FALSE...*

- Then you got the lock!!!

If *the returned value was TRUE...*

- Then someone else has the lock *(so try again later)*

Implementing a Mutex With TSL

```
repeat
 while(TSL(mylock))
    no-op;
 critical section
 mylock = FALSE;
 remainder section
until FALSE
```

Lock (mylock)

Unlock (mylock)

- Note that processes are busy while waiting
  - *this kind of mutex is called a spin lock*

Busy Waiting

- Also called polling or spinning
  - *The thread consumes CPU cycles to evaluate when the lock becomes free !*

- Problem on a single CPU system...
  - *A busy-waiting thread can prevent the lock holder from running & completing its critical section & releasing the lock!*
    - *time spent spinning is wasted on a single CPU system*
  - *Why not block instead of busy wait ?*

Concurrency in the Kernel

Solution 1: Disable interrupts during critical sections

- *Ensures that interrupt handling code will not run*
- *… but what if there are multiple CPUs?*

Solution 2: Use mutex locks based on TSL for critical sections

- *Ensures mutual exclusion for all code that follows that convention*

Disabling interrupts is not enough on MPs…

- Disabling interrupts during critical sections
  - *Ensures that interrupt handling code will not run*
  - *But what if there are multiple CPUs?*
  - *A thread on a different CPU might make a system call which invokes code that manipulates the ready queue*
  - *Disabling interrupts on one CPU didn’t prevent this!*
- Solution: use a mutex lock (based on TSL)
  - *Ensures mutual exclusion for all code that uses it*

Mutex is not enough

- Interrupt inside interrupt handler

جلسه‌ی جدید

سمافور و مسائل کلاسیک هم‌روندی

مسئلەی نویسندە، خوانندە

The Producer-Consumer Problem

- An example of the pipelined model
  - *One thread produces data items*
  - *Another thread consumes them*
- Use a bounded buffer between the threads
- The buffer is a shared resource
  - *Code that manipulates it is a critical section*
- Must suspend the producer thread if the buffer is full
- Must suspend the consumer thread if the buffer is empty

Is This Solution Correct?

```
thread producer {
    while(1){
      // Produce char c
      while (count==n) {
        no_op
      }
      buf[InP] = c
      InP = InP + 1 mod n
      count++
    }
}
```

```
thread consumer {
    while(1){
      while (count==0) {
        no_op
      }
      c = buf[OutP]
      OutP = OutP + 1 mod n
      count--
      // Consume char
    }
}
```

```
Global variables:
    char buf[n]
    int InP = 0   // place to add
    int OutP = 0  // place to get
    int count
```

This Code is Incorrect!

- The “count” variable can be corrupted:
  - *Increments or decrements may be lost!*
  - *Possible Consequences:*
    - Both threads may spin forever
    - Buffer contents may be over-written
- *What is this problem called?*

This Code is Incorrect!

- *What is this problem called?*
- *Race Condition*
- Code that manipulates count must be made into a *???* and protected using *???*

This Code is Incorrect!

- *What is this problem called?*
- *Race Condition*
- Code that manipulates count must be made into a *critical section* and protected using *mutual exclusion*

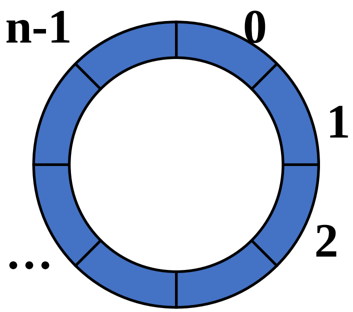
More Problems With This Code

- What if buffer is full?
  - *Producer will busy-wait*
- What if buffer is empty?
  - *Consumer will busy-wait*
- We need a solution based on blocking!

A Solution Based On Blocking

```
thread producer {
  while(1) {
    // Produce char c
    if (count==n) {
      sleep(full)
    }
    buf[InP] = c;
    InP = InP + 1 mod n
    count++
    if (count == 1)
      wakeup(empty)
    }
}
```

```
thread consumer {
  while(1) {
    if(count==0) {
      sleep(empty)
    }
    c = buf[OutP]
    OutP = OutP + 1 mod n
    count--;
    if (count == n-1)
      wakeup(full)
    // Consume char
  }
}
```

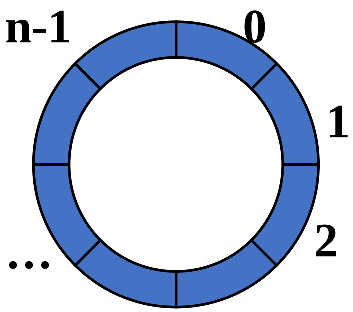



```
Global variables:
    char buf[n]
    int InP = 0    // place to add
    int OutP = 0   // place to get
    int count
```

Use a Mutex to Fix The Problem

```
thread producer {
  while(1) {
    // Produce char c
    if (count==n) {
      sleep(full)
    }
    buf[InP] = c;
    InP = InP + 1 mod n
    count++
    if (count == 1)
      wakeup(empty)
    }
}
```

```
thread consumer {
  while(1) {
    if(count==0) {
      sleep(empty)
    }
    c = buf[OutP]
    OutP = OutP + 1 mod n
    count--;
    if (count == n-1)
      wakeup(full)
    // Consume char
  }
}
```




```
Global variables:
    char buf[n]
    int InP = 0    // place to add
    int OutP = 0   // place to get
    int count
```

Problems

- 1. Sleeping while holding the mutex causes deadlock !
- 2. Releasing the mutex then sleeping opens up a window during which a context switch might occur … again risking deadlock
- 3. How can we release the mutex and sleep in a single atomic operation?

- We need a more powerful synchronization primitive

Semaphores

- An abstract data type that can be used for condition synchronization and mutual exclusion

  What is the difference between mutual exclusion and condition synchronization?

Semaphores

- Condition synchronization
  - *wait until condition holds before proceeding*
  - *signal when condition holds so others may proceed*
- Mutual exclusion
  - *only one at a time in a critical section*

Semaphores

- An abstract data type
  - *containing an integer variable (S)*
  - *Two operations: Wait (S) and Signal (S)*

- Alternative names for the two operations
  - *Wait(S) = Down(S) = P(S)*
  - *Signal(S) = Up(S) = V(S)*

- Blitz names its semaphore operations Down and Up

Classical Definition

```
Down(S)
   {
         while S <= 0 do noop;    /* busy wait!
   */
         S = S - 1;               /* S >= 0 */
   }

Up(S)
   {
         S = S + 1;
   }
```

Problems With The Definition

- Waiting threads hold the CPU
  - *Waste of time in single CPU systems*
  - *Required preemption to avoid deadlock*

Blocking Semaphores

Semaphore S has a value, S.val, and a thread list, S.list.

Down (S)

```
S.val = S.val - 1
If S.val < 0                          /* negative value of S.val */
    {  add calling thread to S.list;  /* is # waiting threads */
            block;                         /* sleep */
    }
```

Up (S)

```
S.val = S.val + 1
If S.val <= 0
    {  remove a thread T from S.list;
            wakeup (T);
    }
```

Implementing Semaphores

- Down () and Up () are assumed to be atomic

  How can we ensure that they are atomic?

Implementing Semaphores

- Implement Down() and Up() as system calls?
  - *How can the kernel ensure Down() and Up() are completed atomically?*
  - *Same solutions as before (disable interrupts, or use TSL-based mutex)*

Semaphores With Disabling

```
struct semaphore {
          int val;
          list L;
       }
```

```
Down(semaphore sem)
  DISABLE_INTS
    sem.val--
    if (sem.val < 0){
      add thread to sem.L
      sleep(thread)
    }
  ENABLE_INTS
```

```
Up(semaphore sem)
  DISABLE_INTS
    sem.val++
    if (sem.val <= 0) {
      th = remove next
         thread from sem.L
      wakeup(th)
    }
  ENABLE_INTS
```

Semaphores With Disabling

```
struct semaphore {
         int val;
         list L;
       }
```

```
Down(semaphore sem)
  DISABLE_INTS
    sem.val--
    if (sem.val < 0){
      add thread to sem.L
      sleep(thread)
    }
  ENABLE_INTS
```

```
Up(semaphore sem)
  DISABLE_INTS
    sem.val++
    if (sem.val <= 0) {
      th = remove next
         thread from sem.L
      wakeup(th)
    }
  ENABLE_INTS
```

Semaphore.down in Blitz

```
method Down()
   var oldIntStat: int
   oldIntStat = SetInterruptsTo (DISABLED)
   if count == 0x80000000
        FatalError ("Semaphore count underflowed during 'Wait‘operation")
   EndIf
   count = count – 1
   if count < 0 waitingThreads.AddToEnd (currentThread)
        currentThread.Sleep ()
   endIf
   oldIntStat = SetInterruptsTo (oldIntStat)
endMethod
```

Semaphore.down in Blitz

```
method Down()
    var oldIntStat: int
    oldIntStat = SetInterruptsTo (DISABLED)
    if count == 0x80000000
         FatalError ("Semaphore count underflowed during 'Wait‘      operation")
    EndIf
    count = count – 1
    if count < 0 waitingThreads.AddToEnd (currentThread)
         currentThread.Sleep ()
    endIf
    oldIntStat = SetInterruptsTo (oldIntStat)
endMethod
```

Semaphore.down in Blitz

```
method Down()
    var oldIntStat: int
    oldIntStat = SetInterruptsTo (DISABLED)
    if count == 0x80000000
         FatalError ("Semaphore count underflowed during 'Wait'      operation")
    EndIf
    count = count - 1
    if count < 0 waitingThreads.AddToEnd (currentThread)
         currentThread.Sleep ()
    endIf
    oldIntStat = SetInterruptsTo (oldIntStat)
endMethod
```

Semaphore.down in Blitz

```
method Down()
    var oldIntStat: int
    oldIntStat = SetInterruptsTo (DISABLED)
    if count == 0x80000000
        FatalError ("Semaphore count underflowed during 'Wait‘     operation")
    EndIf
    count = count – 1
    if count < 0 waitingThreads.AddToEnd (currentThread)
        currentThread.Sleep ()
    endIf
    oldIntStat = SetInterruptsTo (oldIntStat)
endMethod
```

What is `currentThread.Sleep ()?`

- If sleep stops a thread from executing, how, where, and when does it return?
  - *which thread enables interrupts following sleep?*
  - *the thread that called sleep shouldn't return until another thread has called signal !*
  - *… but how does that other thread get to run?*
  - *… where exactly does the thread switch occur?*

- Trace down through the Blitz code until you find a call to switch()
  - *Switch is called in one thread but returns in another!*
  - *See where registers are saved and restored*

Study The Blitz Code

- Thread.c
  - *Thread.Sleep ()*
  - *Run (nextThread)*

- Switch.s
  - *Switch (prevThread, nextThread)*

Blitz Code For Semaphore.up

```
method Up ()
    var oldIntStat: int
    t: ptr to Thread
    oldIntStat = SetInterruptsTo (DISABLED)
    if count == 0x7fffffff
        FatalError ("Semaphore count overflowed during 'Signal' operation")
    endIf
    count = count + 1
    if count <= 0
        t = waitingThreads.Remove ()
        t.status = READY
        readyList.AddToEnd (t)
    endIf
    oldIntStat = SetInterruptsTo (oldIntStat)
endMethod
```

Using Atomic Instructions

- Implementing semaphores with interrupt disabling only works on uni-processors
  - *What should we do on a multiprocessor?*
- Special (hardware) atomic instructions for synchronization
  - *test and set lock (TSL)*
  - *compare and swap (CAS)*
- Semaphore can be built using atomic instructions
  1. *build mutex locks from atomic instructions*
  2. *build semaphores from mutex locks*

پیاده‌سازی سمافور با میوتکس!؟

How about this solution?

```
var cnt: int = 0              -- Signal count
var m1: Mutex = unlocked -- Protects access to “cnt”
    m2: Mutex = locked      -- Locked when waiting
```

Down ():

```
  Lock(m1)
  cnt = cnt - 1
  if cnt<0
    Lock(m2)
    Unlock(m1)
  else
    Unlock(m1)
  endIf
```

Up():

```
  Lock(m1)
  cnt = cnt + 1
  if cnt<=0
    Unlock(m2)
  endIf
  Unlock(m1)
```

How about this solution?

```
var cnt: int = 0             -- Signal count
var m1: Mutex = unlocked -- Protects access to “cnt”
    m2: Mutex = locked       -- Locked when waiting
```

Down ():

```
  Lock(m1)
  cnt = cnt - 1
  if cnt<0
    Lock(m2)
    Unlock(m1)
  else
    Unlock(m1)
  endIf
```

Up():

```
  Lock(m1)
  cnt = cnt + 1
  if cnt<=0
    Unlock(m2)
  endIf
  Unlock(m1)
```

Contains a Deadlock!

How about this solution then?

```
var cnt: int = 0              -- Signal count
var m1: Mutex = unlocked -- Protects access to "cnt"
    m2: Mutex = locked    -- Locked when waiting
```

Down ():

```
  Lock(m1)
  cnt = cnt - 1
  if cnt<0
    Unlock(m1)
    Lock(m2)
  else
    Unlock(m1)
  endIf
```

Up():

```
  Lock(m1)
  cnt = cnt + 1
  if cnt<=0
    Unlock(m2)
  endIf
  Unlock(m1)
```

How about this solution then?

```
var cnt: int = 0            -- Signal count
var m1: Mutex = unlocked -- Protects access to "cnt"
    m2: Mutex = locked      -- Locked when waiting
```

Down ():

```
  Lock(m1)
  cnt = cnt - 1
  if cnt<0
    Unlock(m1)
    Lock(m2)
  else
    Unlock(m1)
  endIf
```

Up():

```
  Lock(m1)
  cnt = cnt + 1
  if cnt<=0
    Unlock(m2)
  endIf
  Unlock(m1)
```

Contains a Race-Condition!

Another solution?

```
var cnt: int = 0            -- Signal count
var m1: Mutex = ulocked -- Protects access to "cnt"
    m2: Mutex = locked    -- Locked when waiting
```

Down ():

```
  Lock(m1)
  cnt = cnt - 1
  if cnt<0
    Unlock(m1)
    Lock(m2)
  endIf
  Unlock(m1)
```

Up():

```
  Lock(m1)
  cnt = cnt + 1
  if cnt<=0
    Unlock(m2)
  else
    Unlock(m1)
  endIf
```

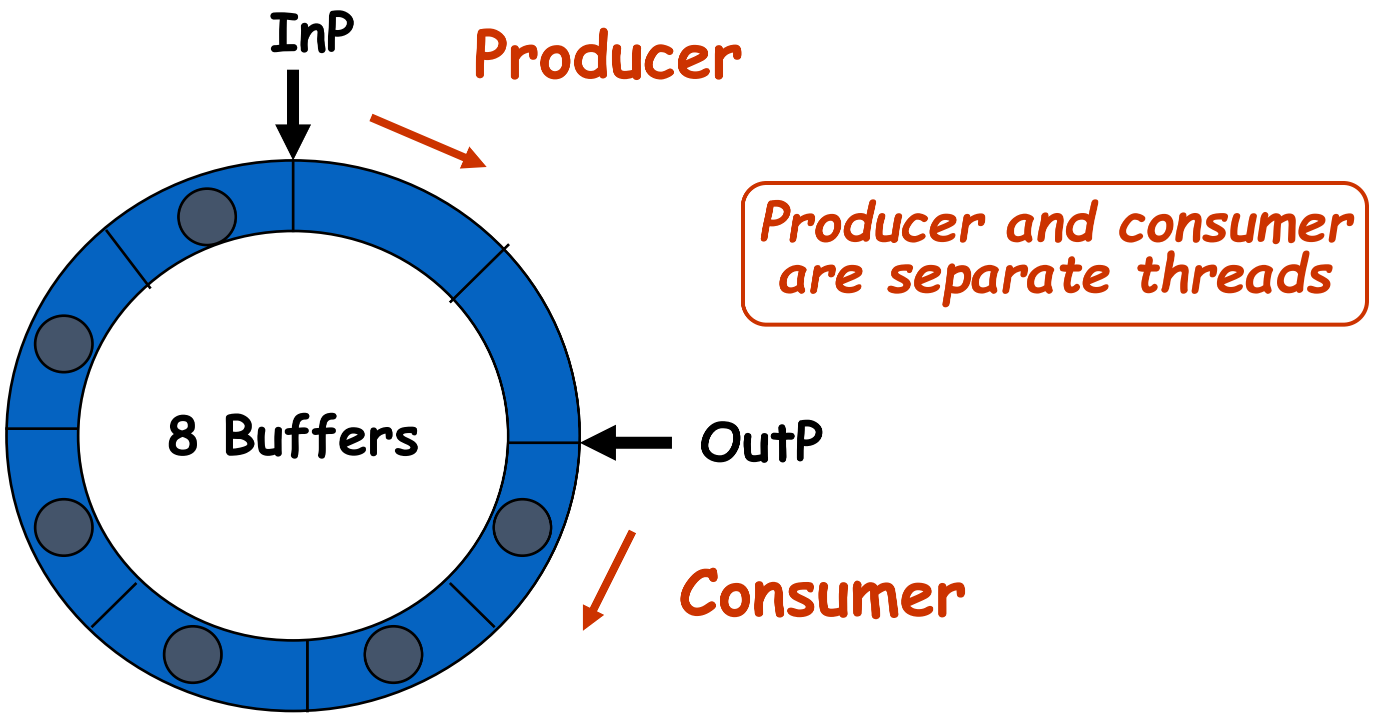
مسئلەی کلاسیک همزمانی

Classical Synchronization Problems

- Producer Consumer (bounded buffer)
- Dining philosophers
- Sleeping barber
- Readers and writers

Producer Consumer Problem

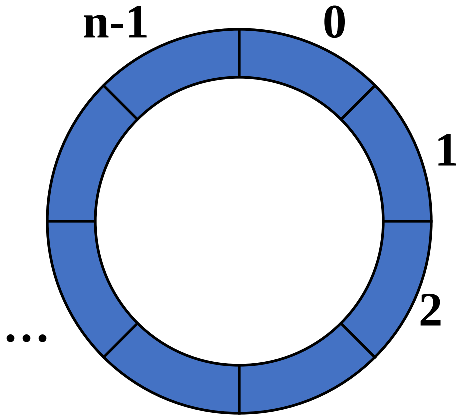
- Also known as the bounded buffer problem



Is This a Valid Solution?

```
thread producer {
     while(1){
       // Produce char c
       while (count==n) {
         no_op
       }
       buf[InP] = c
       InP = InP + 1 mod n
       count++
     }
}
```

```
thread consumer {
     while(1){
       while (count==0) {
         no_op
       }
       c = buf[OutP]
       OutP = OutP + 1 mod n
       count--
       // Consume char
     }
}
```




```
Global variables:
     char buf[n]
     int InP = 0    // place to add
     int OutP = 0   // place to get
     int count
```

Does This Solution Work?

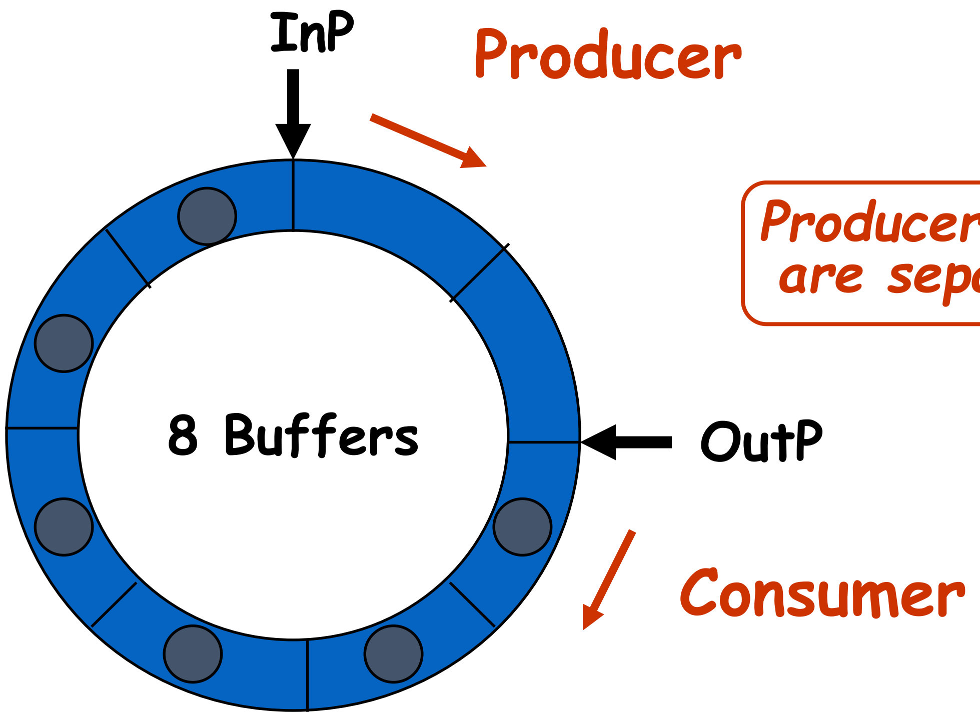
```
Global variables
  semaphore full_buffs = 0;
  semaphore empty_buffs = n;
  char buff[n];
  int InP, OutP;
```

```
thread producer {
   while(1){
      // Produce char c...
      down(empty_buffs)
      buf[InP] = c
      InP = InP + 1 mod n
      up(full_buffs)
   }
}
```

```
thread consumer {
   while(1){
      down(full_buffs)
      c = buf[OutP]
      OutP = OutP + 1 mod n
      up(empty_buffs)
      // Consume char...
   }
}
```

Producer Consumer Problem

- What is the shared state in the last solution?
- Does it apply mutual exclusion? If so, how?



Producer and consumer are separate threads

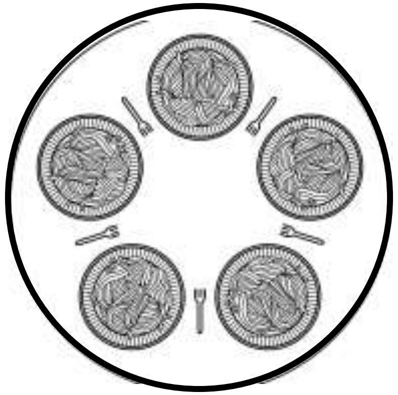
Problems With Solution

- What if we have multiple producers and multiple consumers?
  - *Producer-specific and consumer-specific data becomes shared*
  - *We need to define and protect critical sections*
- *You'll do this in the next parts of the current Blitz project, using the mutex locks you built!*

Dining Philosophers Problem

- Five philosophers sit at a table
- One chopstick between each philosopher

  (need two to eat)



Each philosopher is modeled with a thread

```
while(TRUE) {
  Think();
  Grab first chopstick;
  Grab second chopstick;
  Eat();
  Put down first chopstick;
  Put down second chopstick;
}
```

- *Why do they need to synchronize? How should they do it?*

Is This a Valid Solution?

```
#define N 5

Philosopher() {
  while(TRUE) {
    Think();
    take_chopstick(i);
    take_chopstick((i+1)% N);
    Eat();
    put_chopstick(i);
    put_chopstick((i+1)% N);
  }
}
```

Problems

- Potential for deadlock !

Working Towards a Solution

```
#define N 5

Philosopher() {
  while(TRUE) {
    Think();
    take_chopstick(i);
    take_chopstick((i+1)% N
    Eat();
    put_chopstick(i);
    put_chopstick((i+1)% N);
  }
}
```

take_chopsticks(i)

put_chopsticks(i)

Working Towards a Solution

```
#define N 5

Philosopher() {
  while(TRUE) {
    Think();
    take_chopsticks(i);
    Eat();
    put_chopsticks(i);
  }
}
```

Taking Chopsticks

```
int state[N]
semaphore mutex = 1
semaphore sem[i]
```

```
take_chopsticks(int i) {
   down(mutex);
   state [i] = HUNGRY;
   test(i);
   up(mutex);
   down(sem[i]);
}
```

```
// only called with mutex set!

test(int i) {
 if (state[i] == HUNGRY &&
      state[LEFT] != EATING &&
      state[RIGHT] != EATING){
   state[i] = EATING;
   up(sem[i]);
 }
}
```

Putting Down Chopsticks

```
int state[N]
semaphore mutex = 1
semaphore sem[i]
```

```
put_chopsticks(int i) {
  down(mutex);
  state [i] = THINKING;
  test(LEFT);
  test(RIGHT);
  up(mutex);
}
```

```
// only called with mutex set!

test(int i) {
 if (state[i] == HUNGRY &&
     state[LEFT] != EATING &&
     state[RIGHT] != EATING){
   state[i] = EATING;
   up(sem[i]);
 }
}
```

Dining Philosophers

- Is the previous solution correct?
- What does it mean for it to be correct?
- Is there an easier way?

Recap

- What is a race condition?
- How can we protect against race conditions?
- Can locks be implemented simply by reading and writing to a binary variable in memory?
- How can a kernel make synchronization-related system calls atomic on a uniprocessor?
  - *Why wouldn't this work on a multiprocessor?*
- Why is it better to block rather than spin on a uniprocessor?
- Why is it sometimes better to spin rather than block on a multiprocessor?

Recap

- When faced with a concurrent programming problem, what strategy would you follow in designing a solution?
- What does all of this have to do with Operating Systems?