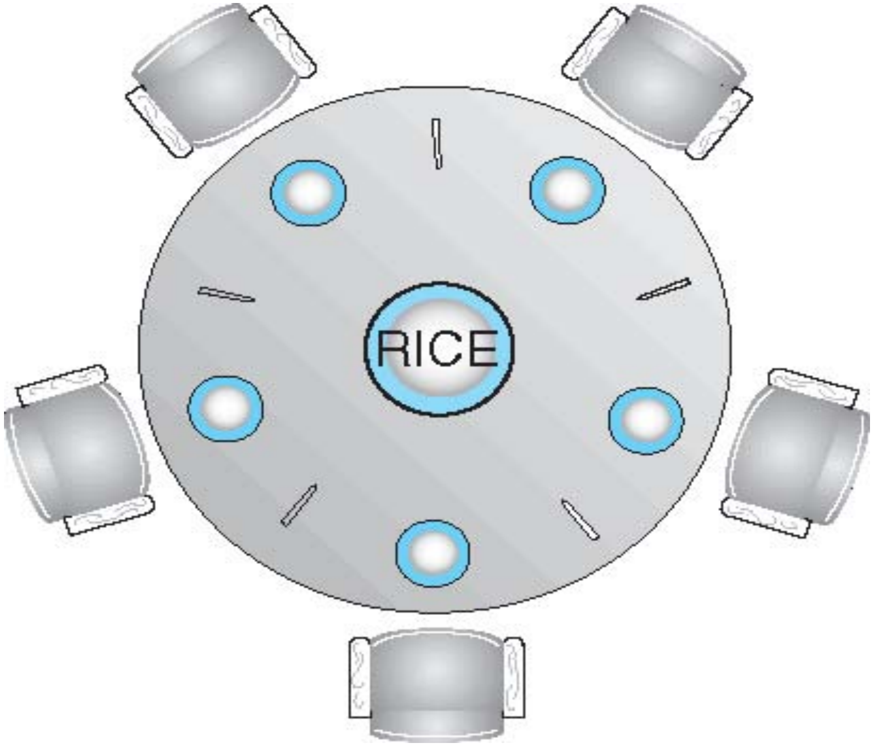
# CS-206 Concurrency

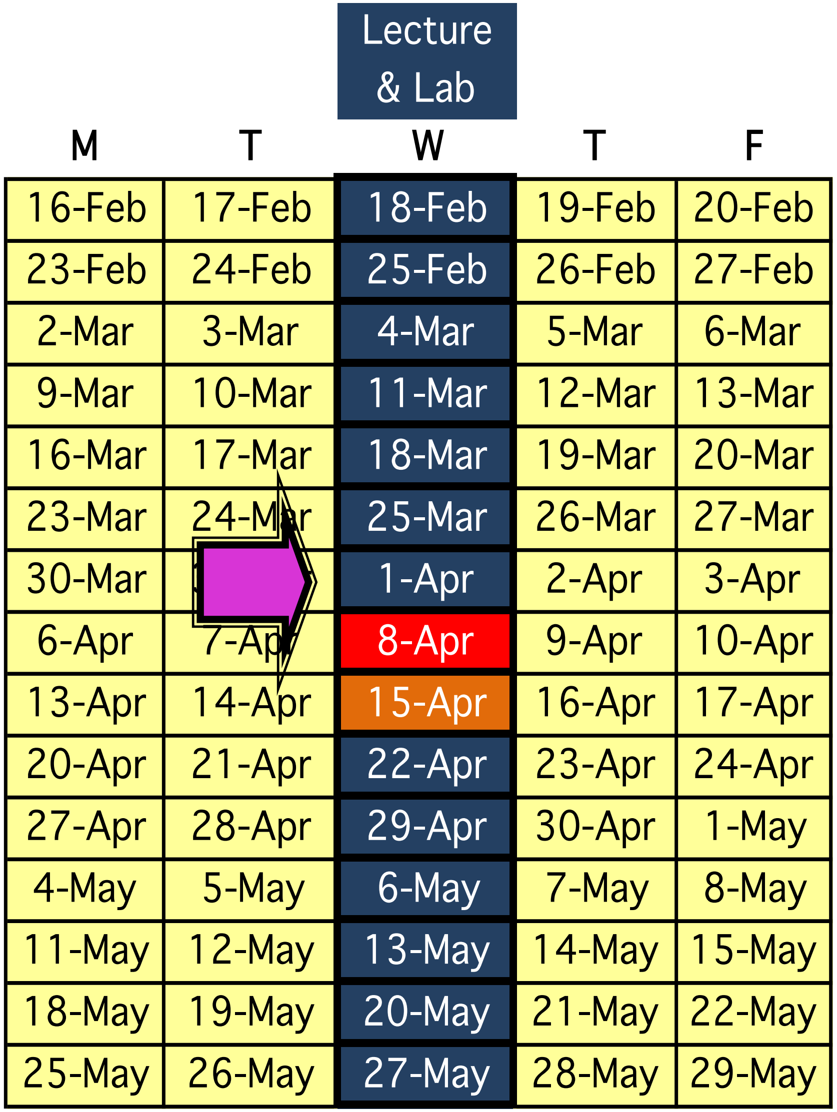
## Lecture 7
## Synchronization Constructs

Spring 2015
Prof. Babak Falsafi
parsa.epfl.ch/courses/cs206/



Adapted from slides originally developed by Silberschatz, Galvin and Gagne

# Where are We?



Lecture & Lab

| M | T | W | T | F |
|---|---|---|---|---|
| 16-Feb | 17-Feb | 18-Feb | 19-Feb | 20-Feb |
| 23-Feb | 24-Feb | 25-Feb | 26-Feb | 27-Feb |
| 2-Mar | 3-Mar | 4-Mar | 5-Mar | 6-Mar |
| 9-Mar | 10-Mar | 11-Mar | 12-Mar | 13-Mar |
| 16-Mar | 17-Mar | 18-Mar | 19-Mar | 20-Mar |
| 23-Mar | 24-Mar | 25-Mar | 26-Mar | 27-Mar |
| 30-Mar | [illegible] | 1-Apr | 2-Apr | 3-Apr |
| 6-Apr | 7-Apr | 8-Apr | 9-Apr | 10-Apr |
| 13-Apr | 14-Apr | 15-Apr | 16-Apr | 17-Apr |
| 20-Apr | 21-Apr | 22-Apr | 23-Apr | 24-Apr |
| 27-Apr | 28-Apr | 29-Apr | 30-Apr | 1-May |
| 4-May | 5-May | 6-May | 7-May | 8-May |
| 11-May | 12-May | 13-May | 14-May | 15-May |
| 18-May | 19-May | 20-May | 21-May | 22-May |
| 25-May | 26-May | 27-May | 28-May | 29-May |

- Hardware atomics
- Sophisticated primitives
  - Semaphores
  - Monitors
  - Conditional variables
- Common problems
  - Bounded buffer
  - Readers-Writers
  - Dining Philosophers
- Next lecture (after break)
  - Mid-term

# Synchronization Hardware

- Many systems provide hardware support for critical section
- Old days: Uniprocessors disabled interrupts
  - Currently running code executes without preemption
  - Too inefficient on multiprocessors
- Today all machines provide atomic instructions
    - Atomic = non-interruptable
  - Either test memory word and set value
  - Or swap contents of two memory words
- Recent machines provide support for transactions
  - Transaction = atomic instruction sequence
  - All memory changes visible before/after but not during

# Solution to Critical-section Problem Using Locks

```
acquire lock

critical section

release lock
```

# Test&Set Instruction

- Definition

```
boolean Test&Set (boolean *target)
{
   boolean rv = *target;
   *target = TRUE;
   return rv;
}
```

# Solution using Test&Set

- Shared boolean variable lock, initialized to FALSE
- Solution

```
while ( TestAndSet ( &lock ))
   ; // do nothing

// critical section

lock = FALSE;
```

# Swap Instruction

- Definition

```
void Swap (boolean *a, boolean *b)
{
   boolean temp = *a;
   *a = *b;
   *b = temp;
}
```

# Solution using Swap

- Shared Boolean variable lock initialized to FALSE
  - Each process has a local Boolean variable key
- Solution

```
key = TRUE;

while ( key )
      Swap ( &lock, &key );

// critical section

lock = FALSE;
```

# Examples in modern instruction sets

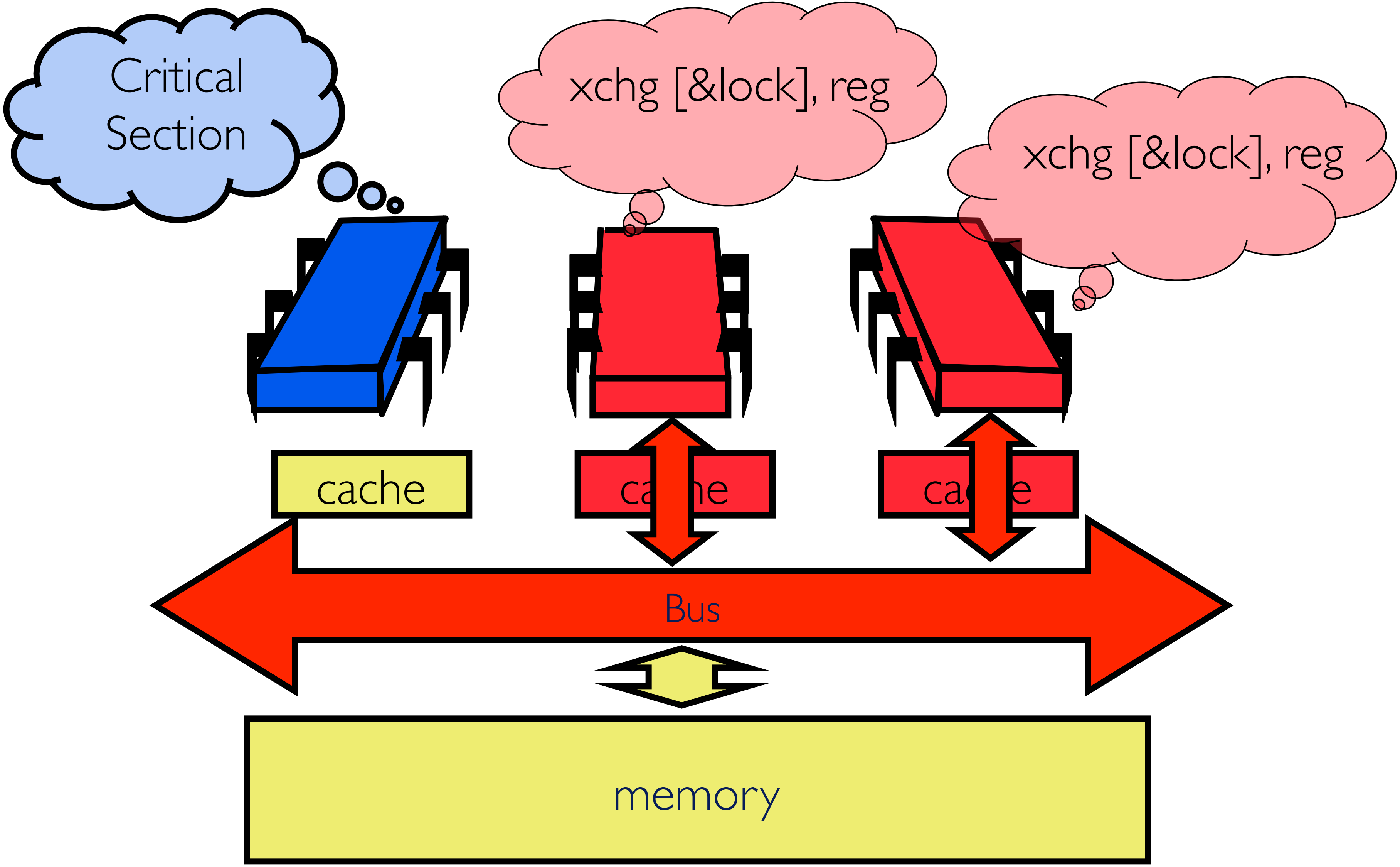
- Oracle SPARC ISA
  - swap [reg1], reg2 // swap contents at address reg1 w/ reg2
- Intel x86
  - xchg [reg1], reg2 // swap contents at address reg1 w/ reg2

# Problems with Test&Set/Swap?

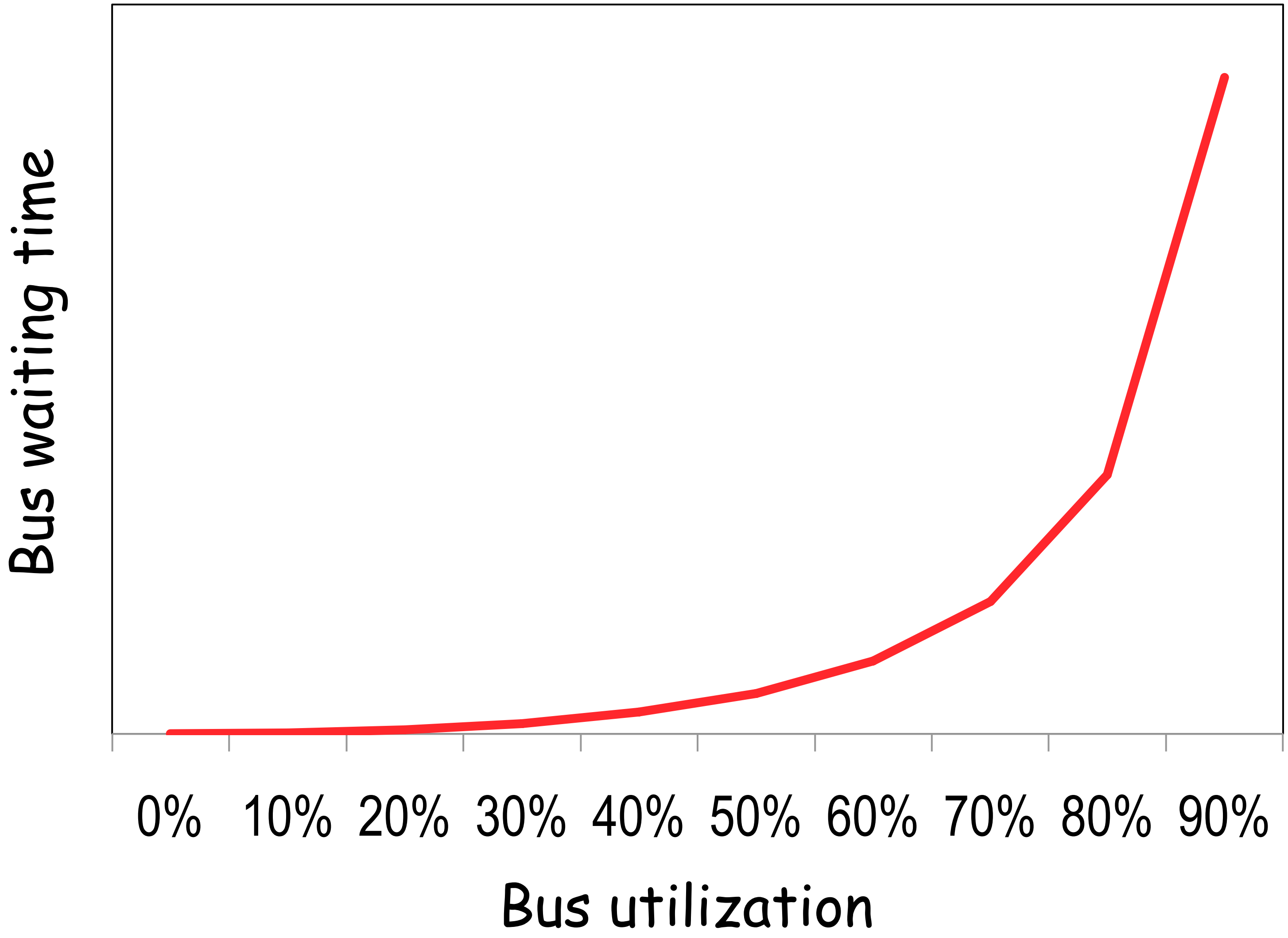
- Threads wait spinning
  - Constantly reading/writing to/from the lock
  - Traffic out of the caches through the bus
  - Bus is a queue: > 50% utilization -> response time exponential

- Not fair
  - There is no queue
  - Any thread can be next independent of waiting

# Traffic


Critical Section
xchg [&lock], reg
xchg [&lock], reg
cache
cache
cache
Bus
memory

# Bus Traffic vs. Waiting Time: M/M/1 Queue



# Test&Test&Set

```
do {
   while ( lock )
        ; // test spinning in cache

          // lock is 0
} while ( TestAndSet ( &lock ));

// critical section

lock = FALSE;
```

# Test&Swap

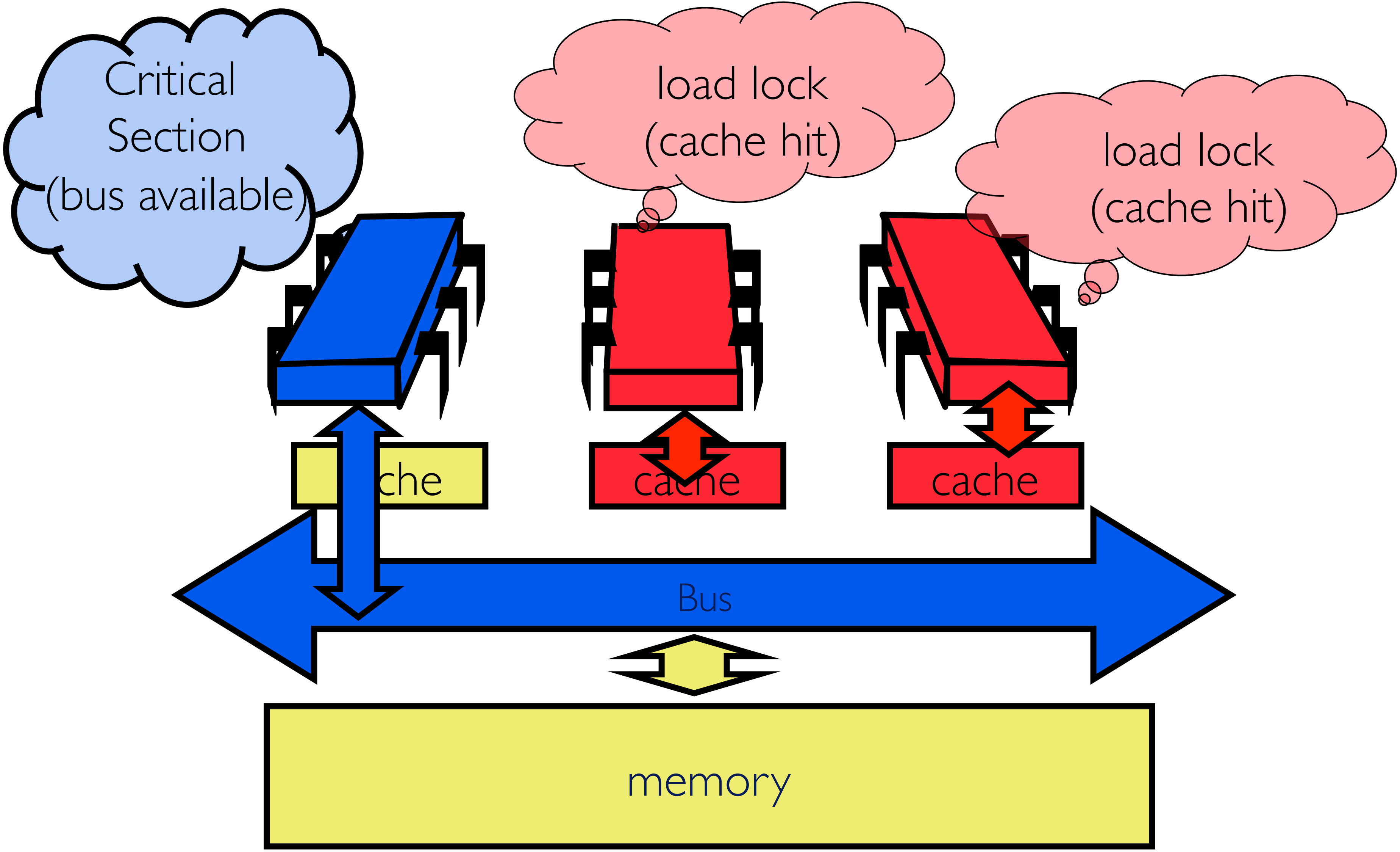
```
do {
   key = TRUE;
   while ( lock )
         ; // test spinning in cache

           // lock is FALSE, quick!
   Swap ( &lock, &key );
} while ( key );

// critical section

lock = FALSE;
```

# Traffic


Critical Section (bus available)
load lock (cache hit)
load lock (cache hit)
che
cache
cache
Bus
memory

# Semaphore

- A high-level abstraction
- Semaphore S: an integer variable
- Two standard operations modify S
  - `wait()` & `signal()`
  - Originally called `P()` & `V()`
- Can only be accessed via two indivisible (atomic) operations

```
wait (S) {
   while (S <= 0)
      ; // no-op
   S--;
}
```

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

# Example Implementation with Test&Set

```
wait(semaphore s){
 done = FALSE; //done is a local variable
 do {
  while(s <= 0 || TestAndSet(&lock))
      ; // do nothing
  if (s > 0){
   done = TRUE;
   s--;
  }
  lock=FALSE;
 } while (!done);
}
```

```
signal(semaphore s){
 while(TestAndSet(&lock))
       ; /* do nothing */
 s++;
 lock=FALSE;
}
```

# Semaphore implementation

- Old days on uniprocessors: disabling/enabling interrupts
- Modern systems:
  - Variety of ways including hardware primitives
  - Test&Set, Swap, Transactional Memory (Intel Haswell)
- From now on, assume wait & signal are atomic
  - All of the operation is performed indivisibly

# Binary Semaphore

- Counting semaphore: integer ranging over unrestricted domain
- Binary semaphore: integer values of 0 or 1; simpler to implement
  - Also known as mutex locks
- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion

```
Semaphore mutex; // initialized to 1

   wait (mutex);
   // Critical Section
   signal (mutex);
```

# Simple Use of Semaphores: Rendez-Vous

Semaphore rendezvous;  // initialized to 0

| Thread 1 | Thread 2 |
|---|---|
| // critical section 1 | wait (rendezvous); |
| signal (rendezvous); | // critical section 2 |

# Semaphore with Busy Waiting

- Busy waiting is not the best use of resources
  - Operating system (OS) can run other threads

- For each wait, there has to be signal
  - To satisfy liveness

- Counting semaphores also suffer from fairness
  - No notion when a thread arrived

# Semaphore without Busy Waiting

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

# Semaphore with Queues (atomic Wait & Signal)

- Wait (and queue):

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

Atomic

- Signal (and wakeup):

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

Atomic

# Deadlock & Starvation

- Let S and Q be two semaphores initialized to 1

| P0 | P1 |
|---|---|
| wait(S); | wait(Q); |
| wait(Q); | wait(S); |
| - | - |
| - | - |
| - | - |
| signal(S); | signal(Q); |
| signal(Q); | signal(S); |

# Deadlock and Starvation

- Starvation
  - Indefinite blocking
  - Thread may never be removed from the semaphore queue in which it is suspended
- Priority Inversion
  - Scheduling problem when lower-priority thread holds a lock needed by higher-priority thread
  - Solved via priority (inheritance) protocol

# Classical Problems of Synchronization

- Classical problems solved via semaphores
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

# Bounded-Buffer Problem

- 1 buffer that holds N items
- Semaphore mutex initialized to value 1
- Semaphore full initialized to value 0
- Semaphore empty initialized to value N

# Bounded-Buffer Problem (Cont.)

- The structure of the producer thread

```
do {
   // produce an item in nextp
   wait (empty);
   wait (mutex);
   // add the item to the buffer
   signal (mutex);
   signal (full);
} while (TRUE);
```

# Bounded-Buffer Problem (Cont.)

- The structure of the producer thread

```
do {
    // produce an item in nextp
    wait (empty);
    wait (mutex);
    // add the item to the buffer
    signal (mutex);
    signal (full);
} while (TRUE);
```

Wait for space in the buffer

# Bounded-Buffer Problem (Cont.)

- The structure of the producer thread

```
do {
   // produce an item in nextp
   wait (empty);
   wait (mutex);
   // add the item to the buffer
   signal (mutex);
   signal (full);
} while (TRUE);
```

Wait for permission to access

# Bounded-Buffer Problem (Cont.)

- The structure of the producer thread

```
do {
   // produce an item in nextp
   wait (empty);
   wait (mutex);
   // add the item to the buffer
   signal (mutex);
   signal (full);
} while (TRUE);
```

Give permission to other threads

# Bounded-Buffer Problem (Cont.)

- The structure of the producer thread

```
do {
   // produce an item in nextp
   wait (empty);
   wait (mutex);
   // add the item to the buffer
   signal (mutex);
   signal (full);
} while (TRUE);
```

Announce an item was added

# Bounded-Buffer Problem (Cont.)

- The structure of the consumer thread

```
do {
   wait (full);
   wait (mutex);
   // remove an item from buffer to
     nextc
   signal (mutex);
   signal (empty);
   // consume the item in nextc
} while (TRUE);
```

# Bounded-Buffer Problem (Cont.)

- The structure of the consumer thread

```
do {
   wait (full);
   wait (mutex);
   // remove an item from buffer to
     nextc
   signal (mutex);
   signal (empty);
   // consume the item in nextc
} while (TRUE);
```

Wait for an item
in the buffer

# Bounded-Buffer Problem (Cont.)

- The structure of the consumer thread

```
do {
   wait (full);
   wait (mutex);
   // remove an item from buffer to
     nextc
   signal (mutex);
   signal (empty);
   // consume the item in nextc
} while (TRUE);
```

Wait for permission to access

# Bounded-Buffer Problem (Cont.)

- The structure of the consumer thread

```
do {
   wait (full);
   wait (mutex);
   // remove an item from buffer to
     nextc
   signal (mutex);
   signal (empty);
   // consume the item in nextc
} while (TRUE);
```

Give permission to other threads

# Bounded-Buffer Problem (Cont.)

- The structure of the consumer thread

```
do {
   wait (full);
   wait (mutex);
   // remove an item from buffer to
     nextc
   signal (mutex);
   signal (empty);
   // consume the item in nextc
} while (TRUE);
```

Announce an item was removed

# Readers-Writers Problem

- Data set shared among concurrent threads
  - Readers only read the data set – no updates
  - Writers can both read and write
- Multiple readers can read at the same time
  - Only single writer can access shared data at same time
- Shared Data
  - Data set
  - Semaphore mutex initialized to 1
  - Semaphore wrt initialized to 1
  - Integer readcount initialized to 0

# Readers-Writers Problem (Cont.)

Reader-Writer decisions:

- When is the writer done?
- When are the readers done?

Must do book-keeping:

- The first reader waits on *wrt* to allow the writer to finish
  - Other readers go through
- The last reader signals on *wrt* to allow the writer to start
  - Other readers go through

# Readers-Writers Problem (Cont.)

- The structure of a writer thread

```
do {
    wait(wrt);
    // writing is performed
    signal (wrt) ;
} while (TRUE);
```

# Readers-Writers Problem (Cont.)

- The structure of a writer thread

```
do {
  wait(wrt);
  // writing is performed
  signal (wrt) ;
} while (TRUE);
```

Wait for permission to access the data set

# Readers-Writers Problem (Cont.)

- The structure of a writer thread

```
do {
   wait(wrt);
   // writing is performed
   signal (wrt) ;
} while (TRUE);
```

Give permission to other threads

# Readers-Writers Problem (Cont.)

- The structure of a reader thread

```
do {
   wait (mutex) ;
   readcount ++ ;
   if (readcount == 1)
      wait (wrt) ;
   signal (mutex)
   // reading is performed
   wait (mutex) ;
   readcount -- ;
   if (readcount == 0)
      signal (wrt) ;
   signal (mutex) ;
} while (TRUE);
```

Wait for permission to increase the readcount

# Readers-Writers Problem (Cont.)

- The structure of a reader thread

```
do {
   wait (mutex) ;
   readcount ++ ;
   if (readcount == 1)
      wait (wrt) ;
   signal (mutex)
   // reading is performed
   wait (mutex) ;
   readcount -- ;
   if (readcount == 0)
      signal (wrt) ;
   signal (mutex) ;
} while (TRUE);
```

Wait for permission to increase the readcount

# Readers-Writers Problem (Cont.)

- The structure of a reader thread

```
do {
   wait (mutex) ;
   readcount ++ ;
   if (readcount == 1)
      wait (wrt) ;
   signal (mutex)
   // reading is performed
   wait (mutex) ;
   readcount -- ;
   if (readcount == 0)
      signal (wrt) ;
   signal (mutex) ;
} while (TRUE);
```

If you are the first reader, wait for permission to access the data set

# Readers-Writers Problem (Cont.)

- The structure of a reader thread

```
do {
   wait (mutex) ;
   readcount ++ ;
   if (readcount == 1)
      wait (wrt) ;
   signal (mutex)
   // reading is performed
   wait (mutex) ;
   readcount -- ;
   if (readcount == 0)
      signal (wrt) ;
   signal (mutex) ;
} while (TRUE);
```

Give permission to other readers to access the readcount

# Readers-Writers Problem (Cont.)

- The structure of a reader thread

```
do {
   wait (mutex) ;
   readcount ++ ;
   if (readcount == 1)
      wait (wrt) ;
   signal (mutex)
   // reading is performed
   wait (mutex) ;
   readcount -- ;
   if (readcount == 0)
      signal (wrt) ;
   signal (mutex) ;
} while (TRUE);
```

Wait for permission to decrease the readcount

# Readers-Writers Problem (Cont.)

- The structure of a reader thread

```
do {
   wait (mutex) ;
   readcount ++ ;
   if (readcount == 1)
      wait (wrt) ;
   signal (mutex)
   // reading is performed
   wait (mutex) ;
   readcount -- ;
   if (readcount == 0)
      signal (wrt) ;
   signal (mutex) ;
} while (TRUE);
```

If you are the last reader give permission to other threads

# Readers-Writers Problem (Cont.)

- The structure of a reader thread

```
do {
   wait (mutex) ;
   readcount ++ ;
   if (readcount == 1)
      wait (wrt) ;
   signal (mutex)
   // reading is performed
   wait (mutex) ;
   readcount -- ;
   if (readcount == 0)
      signal (wrt) ;
   signal (mutex) ;
} while (TRUE);
```

Give permission to other readers to access the readcount

# Readers-Writers Problem Variations

Many variations possible

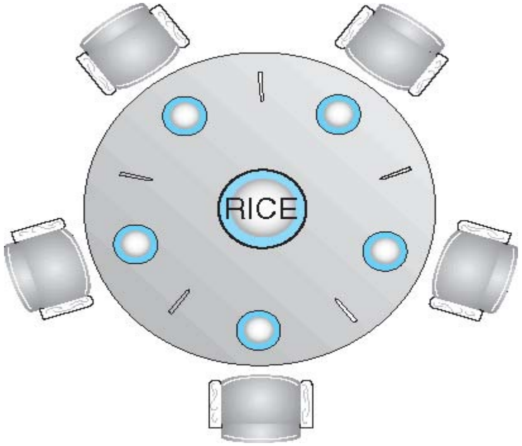
E.g.,

1. No reader kept waiting unless writer has permission to use shared object
2. Or, once writer is ready, it performs write asap

- These variations may suffer from starvation
- Problem can be solved through reader-writer locks

# Dining-Philosophers Problem

- Philosophers spend their lives thinking and eating
- Don't interact with their neighbors
- Occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In case of 5 philosophers, shared data
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1



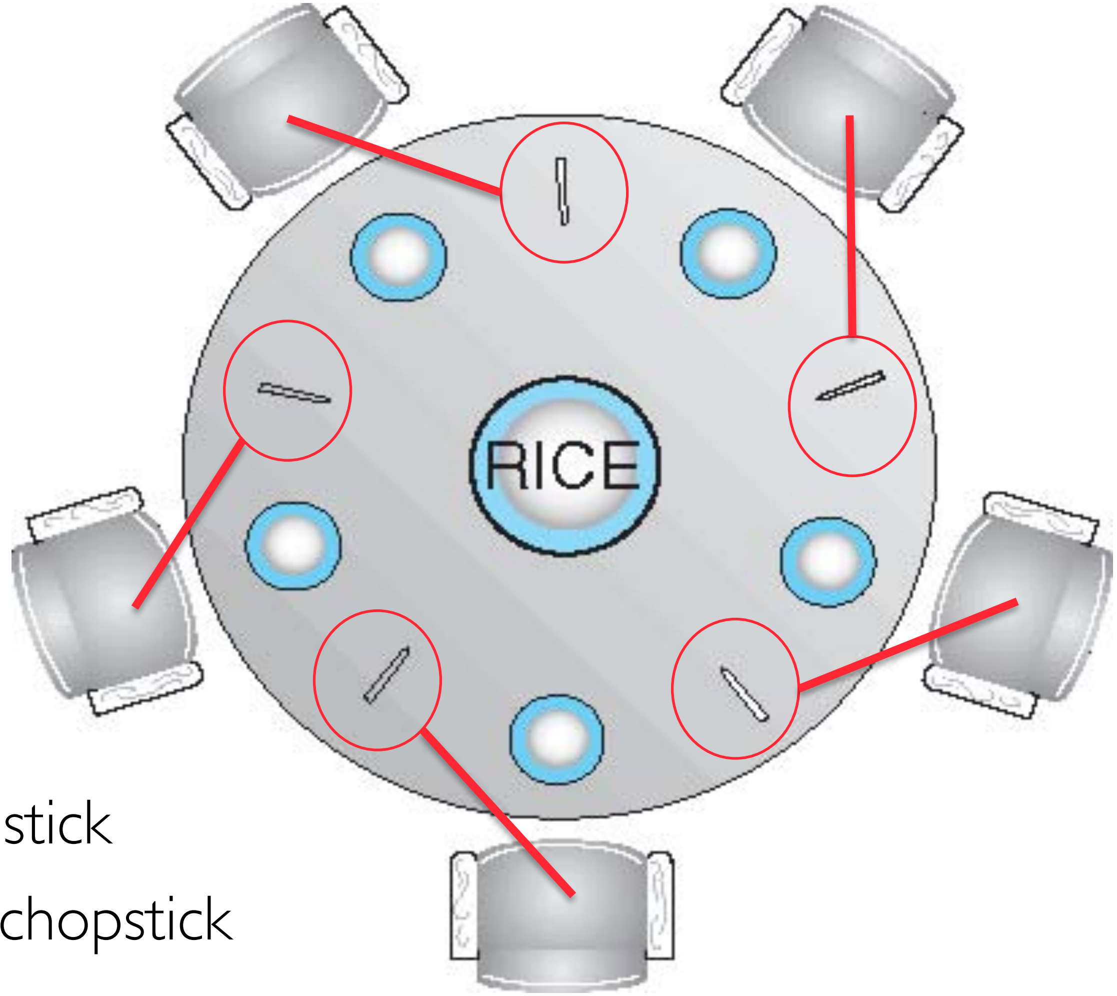
# Dining-Philosophers Problem Algorithm

- The structure of Philosopher i:

```
do {
   wait ( chopstick[i] );
   wait ( chopStick[ (i + 1) % 5] );
   // eat
   signal ( chopstick[i] );
   signal ( chopstick[ (i + 1) % 5] );
   // think
} while (TRUE);
```

- What is the problem with this algorithm?

# Dining-Philosophers: Deadlock!



- Each philosopher
  - Grabs their left chopstick
  - Waits for their right chopstick
- Deadlock!

# Problems with Semaphores

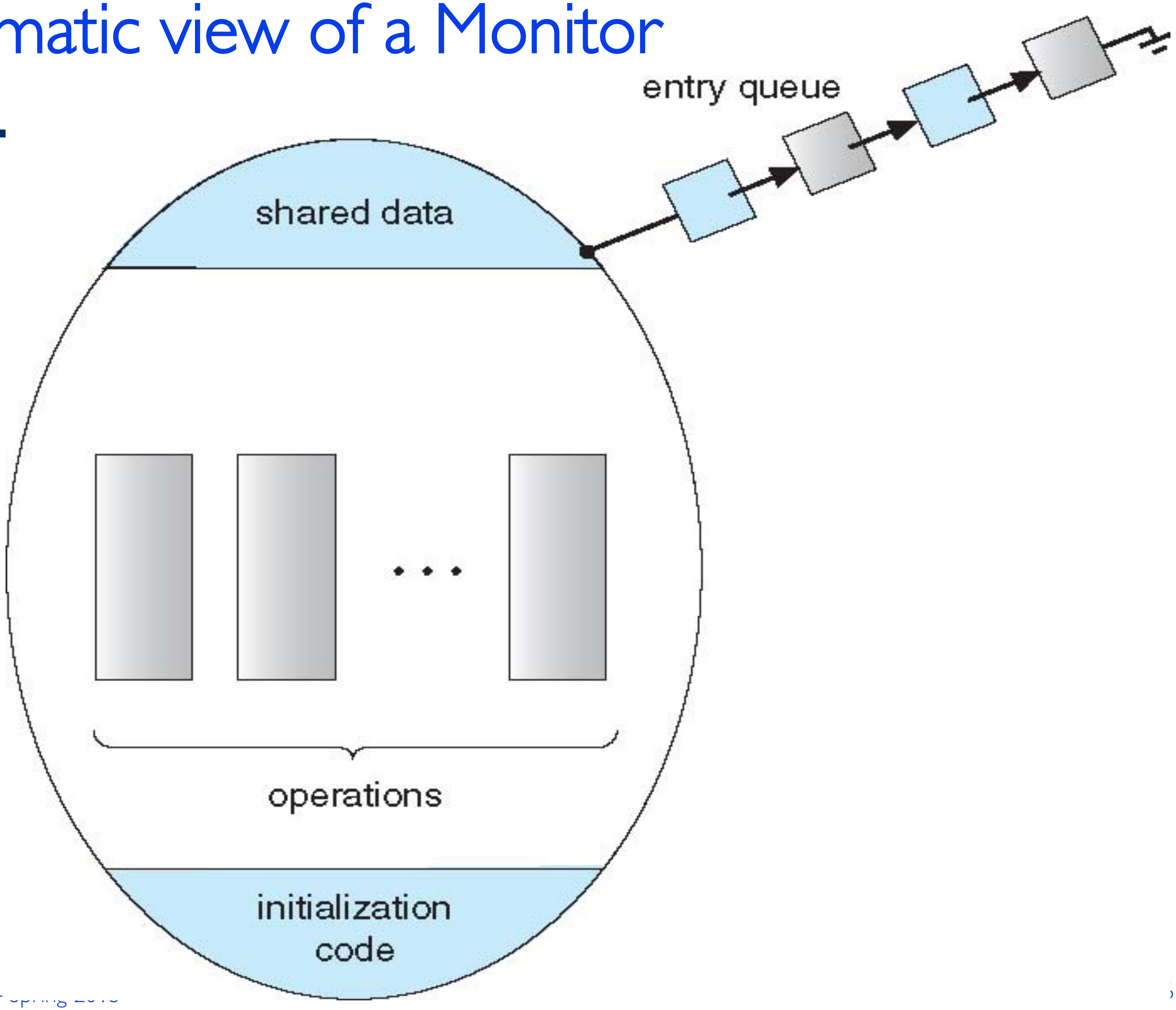
- Incorrect use of semaphore operations:
  - signal(mutex) wait(mutex)
  - wait(mutex) wait(mutex)
  - Omitting of wait(mutex) or signal(mutex) (or both)
- Deadlock and starvation

# Monitors

- Abstraction providing convenient/effective synchronization
- Abstract data type, internal variables only accessible by code within the procedure
- Only one thread may be active within the monitor at a time

```
monitor monitor-name
{
   // shared variable declarations
   procedure P1 (…) { …. }
   …
   procedure Pn (…) {……}
   Initialization code (…) { … }
}
```

# Schematic view of a Monitor

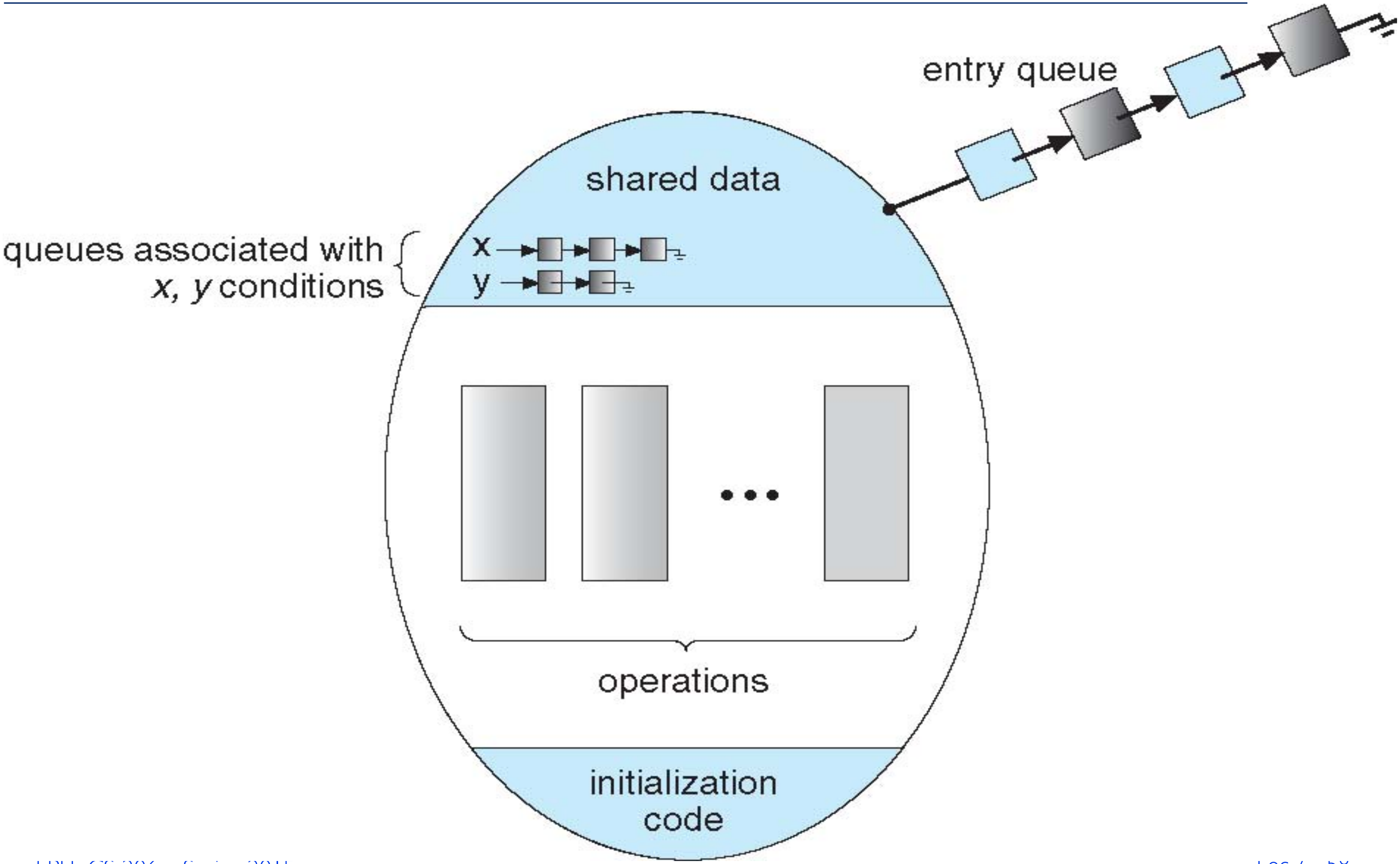


# Condition Variables

condition x, y;

- Two operations on a condition variable:
    - x.wait() suspends the thread until x.signal()
    - x.signal() resumes a thread (if any) that invoked x.wait()
        - If no x.wait() on variable, then it has no effect

# Monitor with Condition Variables



# Condition Variables Choices

- If P invokes x.signal(), with Q in x.wait(), what happens next?
  - If Q is resumed, then P must wait

- Options include
  - Signal & wait: P waits until Q leaves or waits for another condition
  - Signal & continue: Q waits until P leaves the monitor or waits for another condition
  - Both have pros and cons, language implementer can decide
  - Implemented in many languages including Mesa, C#, Java

# Solution to Dining Philosophers

```
monitor DiningPhilosophers
{
 enum { THINKING; HUNGRY, EATING) state[5];
 condition self[5];

 void pickup (int i) {
     state[i] = HUNGRY;
     test(i);
     if (state[i] != EATING) self[i].wait();
 }
```

# Solution to Dining Philosophers

```
monitor DiningPhilosophers
{
 enum { THINKING; HUNGRY, EATING) state[5];
 condition self[5];

 void pickup (int i) {
      state[i] = HUNGRY;
      test(i);
      if (state[i] != EATING) self[i].wait();
 }
```

Philosopher i is hungry

# Solution to Dining Philosophers

```
monitor DiningPhilosophers
{
 enum { THINKING; HUNGRY, EATING) state[5];
 condition self[5];

 void pickup (int i) {
      state[i] = HUNGRY;
      test(i);
      if (state[i] != EATING) self[i].wait();
 }
```

i tries to eat

# Solution to Dining Philosophers

```
monitor DiningPhilosophers
{
 enum { THINKING; HUNGRY, EATING) state[5];
 condition self[5];

 void pickup (int i) {
      state[i] = HUNGRY;
      test(i);
      if (state[i] != EATING) self[i].wait();
 }
```

If i can't eat,
i goes to sleep

# Solution to Dining Philosophers

```
void putdown (int i) {
    state[i] = THINKING;
    // test left and right neighbors
    test((i + 4) % 5);
    test((i + 1) % 5);
}
```

# Solution to Dining Philosophers

```
void putdown (int i) {
    state[i] = THINKING;
    // test left and right neighbors
    test((i + 4) % 5);
    test((i + 1) % 5);
}
```

i starts thinking

# Solution to Dining Philosophers

```
void putdown (int i) {
    state[i] = THINKING;
    // test left and right neighbors
    test((i + 4) % 5);
    test((i + 1) % 5);
}
```

Puts chopsticks down and lets neighbors use them

# Solution to Dining Philosophers (Cont.)

```
 void test (int i) {
     if ( (state[(i + 4) % 5] != EATING) &&
     (state[i] == HUNGRY) &&
     (state[(i + 1) % 5] != EATING) ) {
          state[i] = EATING ;
          self[i].signal () ;
     }
 }

 initialization_code() {
     for (int i = 0; i < 5; i++)
          state[i] = THINKING;
 }
}
```

# Solution to Dining Philosophers (Cont.)

```
 void test (int i) {
     if ( (state[(i + 4) % 5] != EATING) &&
     (state[i] == HUNGRY) &&
     (state[(i + 1) % 5] != EATING) ) {
          state[i] = EATING ;
          self[i].signal () ;
     }
 }

 initialization_code() {
     for (int i = 0; i < 5; i++)
          state[i] = THINKING;
 }
}
```

Checks if i's left neighbor is eating

# Solution to Dining Philosophers (Cont.)

```
 void test (int i) {
     if ( (state[(i + 4) % 5] != EATING) &&
     (state[i] == HUNGRY) &&
     (state[(i + 1) % 5] != EATING) ) {
          state[i] = EATING ;
          self[i].signal () ;
     }
 }

 initialization_code() {
     for (int i = 0; i < 5; i++)
          state[i] = THINKING;
 }
}
```

Checks if i is hungry

# Solution to Dining Philosophers (Cont.)

```
  void test (int i) {
      if ( (state[(i + 4) % 5] != EATING) &&
      (state[i] == HUNGRY) &&
      (state[(i + 1) % 5] != EATING) ) {
           state[i] = EATING ;
           self[i].signal () ;
      }
  }

  initialization_code() {
      for (int i = 0; i < 5; i++)
           state[i] = THINKING;
  }
}
```

Checks if i’s right neighbor is eating

# Solution to Dining Philosophers (Cont.)

```
 void test (int i) {
     if ( (state[(i + 4) % 5] != EATING) &&
     (state[i] == HUNGRY) &&
     (state[(i + 1) % 5] != EATING) ) {
          state[i] = EATING ;
          self[i].signal () ;
     }
 }

 initialization_code() {
     for (int i = 0; i < 5; i++)
          state[i] = THINKING;
 }
}
```

If i's neighbors are not eating and i is hungry, i starts eating

# Solution to Dining Philosophers (Cont.)

- Each philosopher i invokes the operations pickup() and putdown() in the following sequence:

```
DiningPhilosophers.pickup (i);
EAT
DiningPhilosophers.putdown (i);
```

- No deadlock, but starvation is possible
  - Why?
  - Can you address it?

# Summary

- Need simple, efficient atomic ops
  - Hardware primitives: test&set, swap, transactional memory
  - Think about traffic while busy waiting
- Need higher level abstractions for programmability
  - Semaphores, Monitors & Condition Variables
  - Support in the OS for waiting/sleeping and waking up
- A few classical problems
  - Bounded buffer
  - Readers/writer
  - Dining philosophers