CS2310 Modern Operating Systems

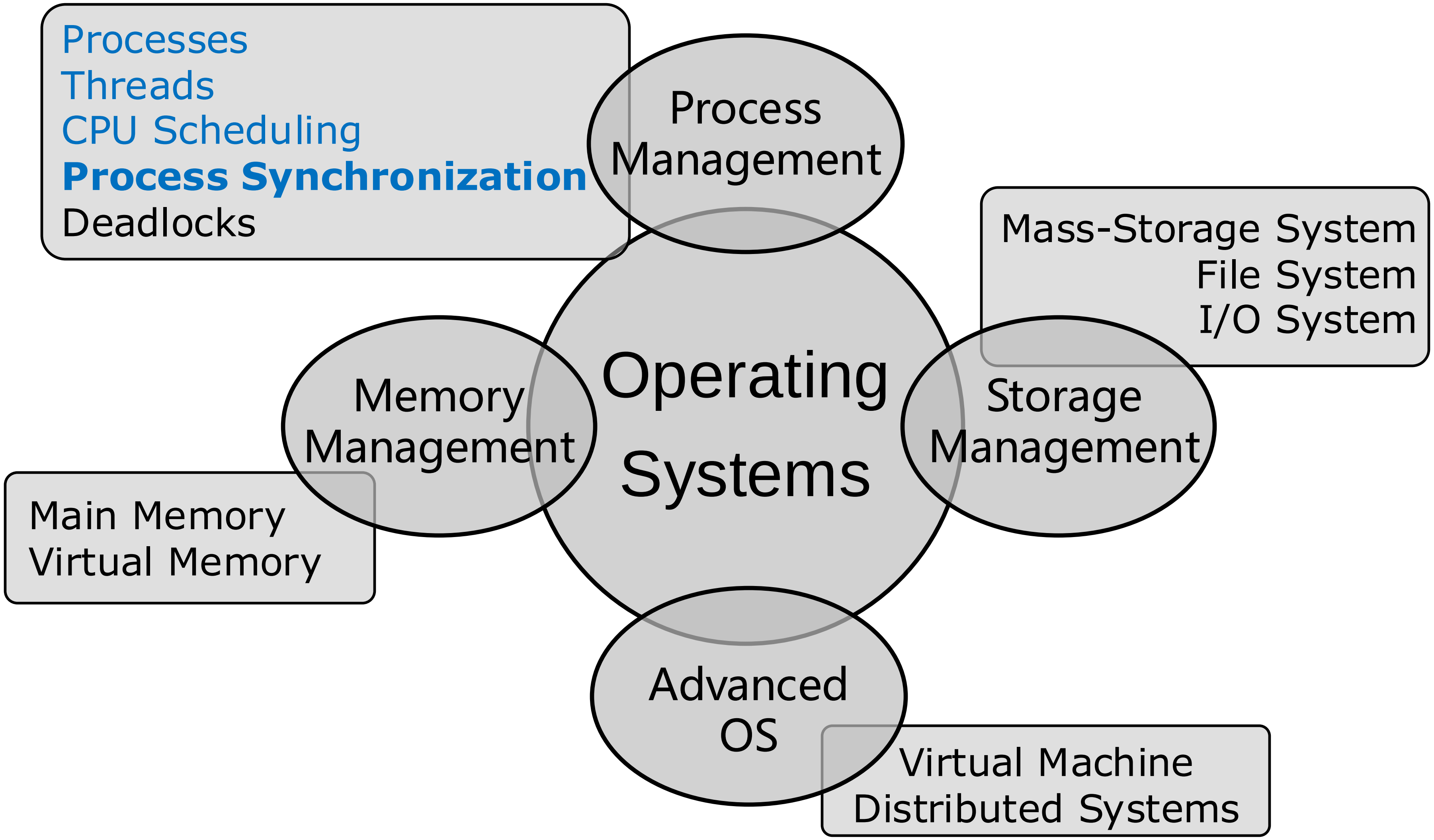
Process Synchronization

Shengzhong Liu

Department of Computer Science and Engineering

Shanghai Jiao Tong University

Operating System Topics



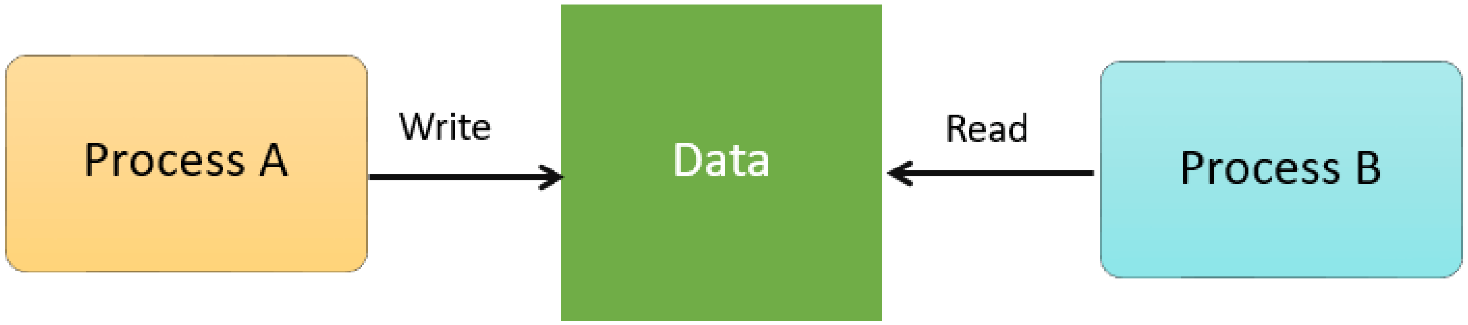
Outline

- Background: Critical-Section Problem
- Synchronization Mechanisms:
  - Peterson’s Solution
  - Hardware Support for Synchronization
  - Mutex Locks and Semaphores
  - Monitors
- Synchronization Problem Formulations

Background:
Critical Section Problem

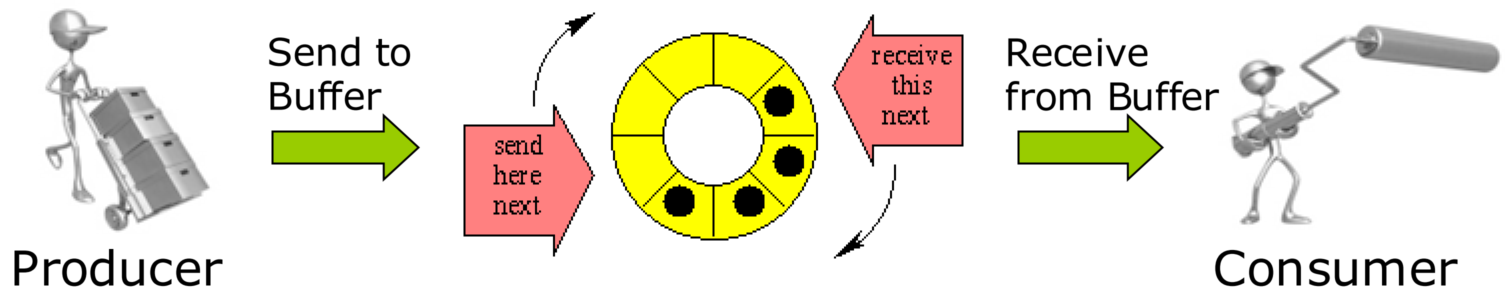
Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the **orderly execution of cooperating processes**



Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  - **Unbounded-buffer** places no practical limit on the buffer size
  - **Bounded-buffer** assumes that there is a fixed buffer size



Improved Solution of Producer-Consumer Problem

Producer

```
while (true) {
   /* produce an item and put in nextProduced*/
   while (counter == BUFFER_SIZE); // do nothing
   buffer [in] = nextProduced;
   in = (in + 1) % BUFFER_SIZE;
   counter++;
}
```

Consumer

```
 while (true)  {
   while (counter == 0) ; // do nothing
   nextConsumed =  buffer[out];
   out = (out + 1) % BUFFER_SIZE;
   counter--;
   /*  consume the item */
}
```

This solution allows us to utilize all available buffer slots.

Race Condition: Example 1

- counter++ could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- counter-- could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

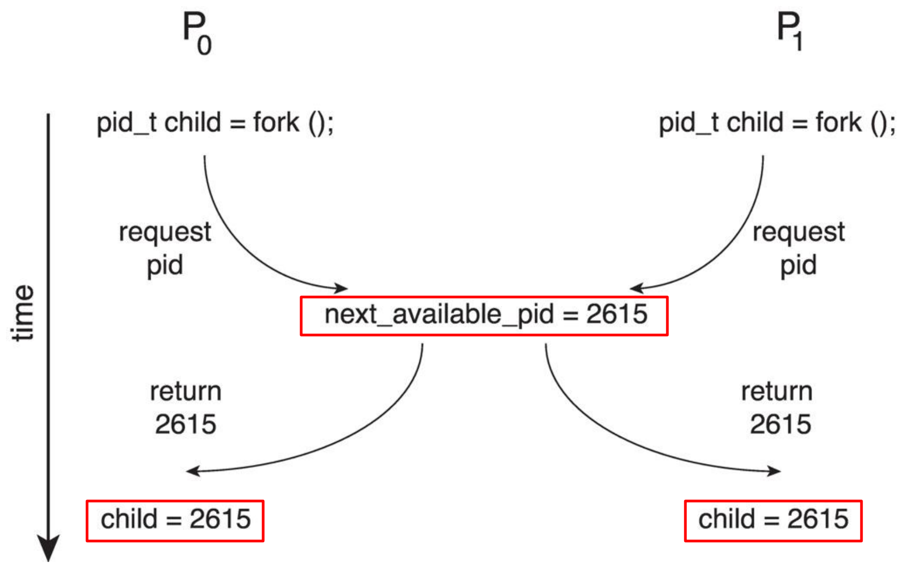
- Consider this execution interleaving with "**counter = 5**" initially:

S0: producer execute	register1 = counter	{register1 = 5}
S1: producer execute	register1 = register1 + 1	{register1 = 6}
S2: consumer execute	register2 = counter	{register2 = 5}
S3: consumer execute	register2 = register2 - 1	{register2 = 4}
S4: producer execute	counter = register1	{counter = 6}
S5: consumer execute	counter = register2	{counter = 4}

- **Race condition**:
  - Several processes access and manipulate the same data concurrently
  - The execution outcome depends on access order

Race Condition: Example 2

- Processes P_0 and P_1 are creating child processes using the **`fork()`** system call
- Race condition on kernel variable **`next_available_pid`** which represents the next available process identifier (`pid`)



- Unless there is a mechanism to prevent P_0 and P_1 from accessing the variable `next_available_pid`, the same pid could be assigned to two different processes!

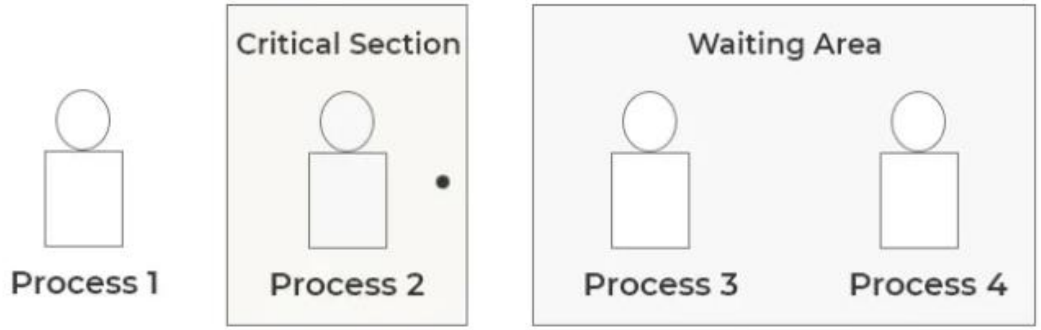
Critical Section Problem

- Consider system of *n* processes *{p_0, p_1, … , p_{n-1}}*
  - Each process has a **critical section** segment of codes
  - Process may be changing common variables, updating table, writing file, etc
  - When one process is in critical section, no others could in its critical section
- Critical section problem is to design protocols to solve this
- Each process must:
  - ask permission to enter the critical section in **entry section**
  - may follow critical section with **exit section**
  - then **remainder section**
- Especially challenging with preemptive kernels

```
while (true) {

        entry section

            critical section

        exit section

            remainder section

}
```

Solution Requirements of CS Problem

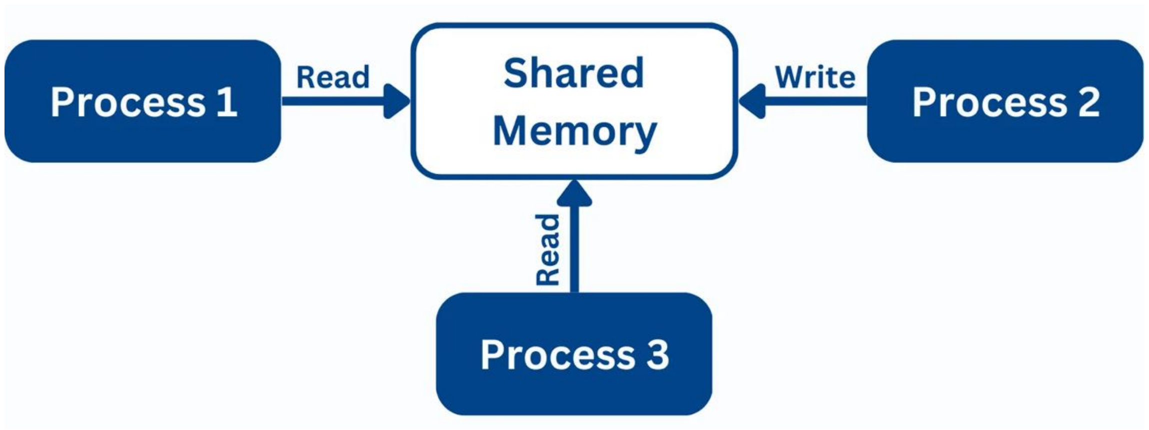
1. **Mutual Exclusion**
   - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress**
   - If no process is executing in its critical section and there exist processes waiting to enter, the selection of entering processes cannot be postponed indefinitely
3. **Bounded Waiting**
   - A bound must exist on #times that other processes enter their critical sections after a process requests to enter its critical section but before the request is granted
   - **Sequential Access** – The sequence of accessing the critical section follows the order of requests raised by the processes



Synchronization Mechanisms

Process Synchronization Mechanisms

- Peterson’s Solution – A classical and general algorithm
- Hardware Synchronization
- Mutex and Semaphores
- Monitors



Peterson's Solution

- We have two processes: P_i and P_j
- Assume the **load** and **store** machine-language instructions are atomic;
  - They 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!

```
while (true){

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

      /* critical section */

    flag[i] = false;

    /* remainder section */

}
```

Algorithm for **process** P_i

Correctness of Peterson's Solution

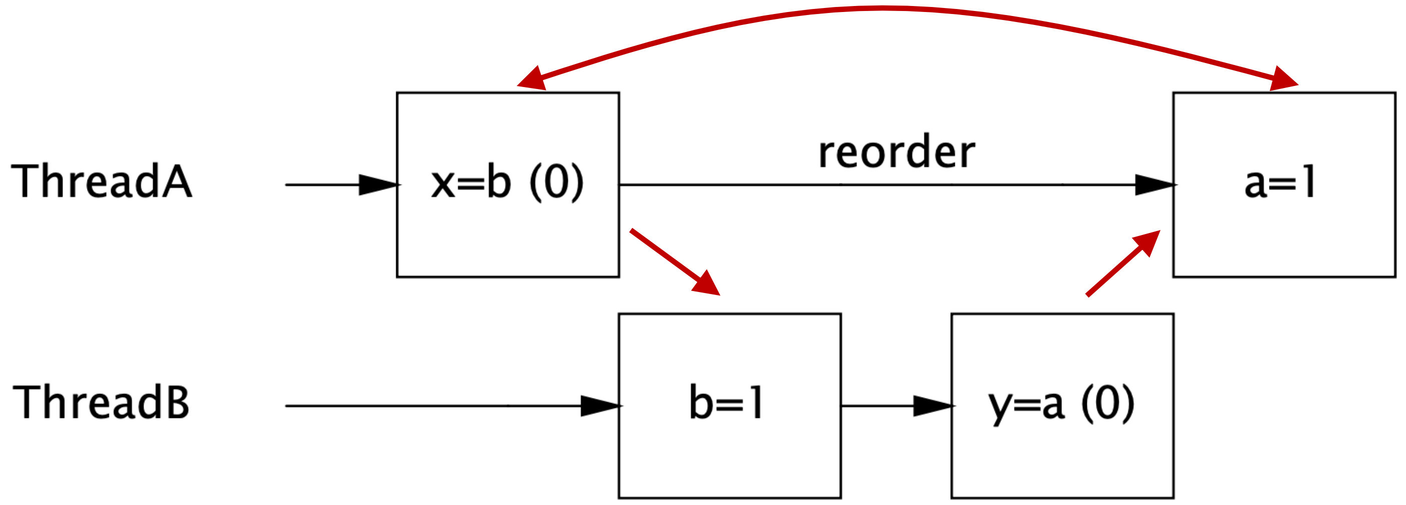
- Provable that the three CS requirement are met:
    1. Mutual exclusion is preserved

       P_i enters CS only if: either `flag[j] = false` or `turn = i`
    2. Progress requirement is satisfied
    3. Bounded-waiting requirement is met

```
while (true){

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

         /* critical section */

     flag[i] = false;

     /* remainder section */

}
```

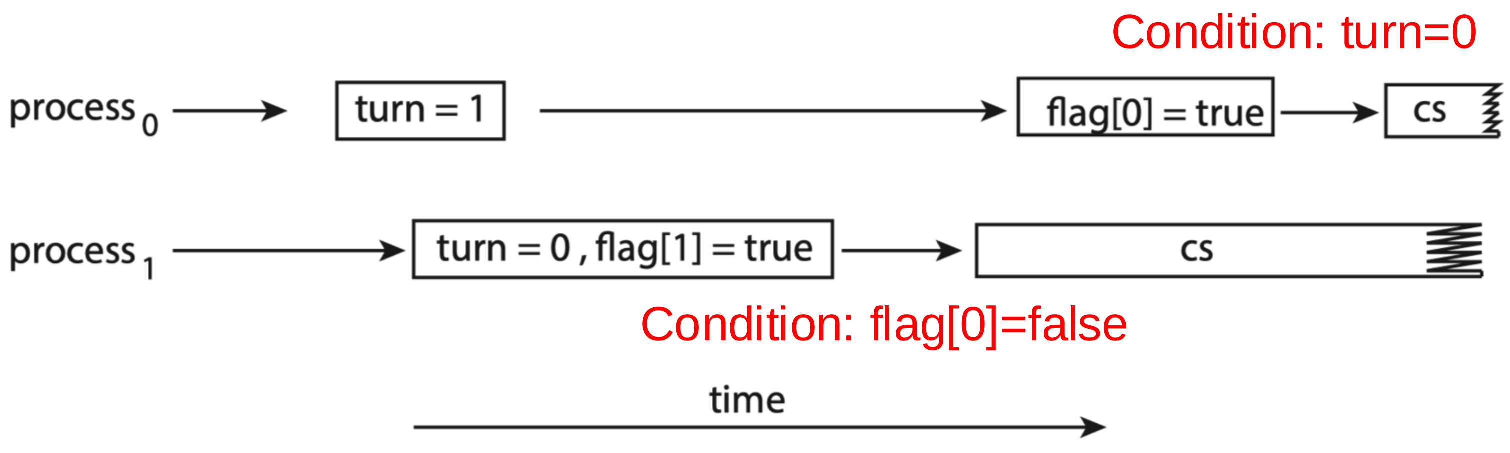
Peterson's Solution and Modern Architecture

- Although useful for demonstrating an algorithm, Peterson's Solution is not guaranteed to work on modern architectures.
  - To improve performance, processors and/or compilers may **reorder read and write operations that have no dependencies**
  - For single-threaded program, this is ok as the result will always be the same.
  - For multi-threaded program, the reordering may produce inconsistent or unexpected results!



Peterson’s Solution

- Peterson’s Solution is not guaranteed to work on modern architectures with reordered execution of instructions.
- We need proper synchronization tools to preserve mutual exclusion.
- The example below allows both processes to be in their critical section at the same time!



Synchronization Mechanisms: Hardware Synchronization

Hardware Support for Synchronization

- Many systems provide hardware support for critical section code
  - Memory barriers
  - Hardware instructions
  - Atomic variables

Hardware Sync: (1) Memory Barrier

- **Memory models** are the memory guarantees a computer architecture makes to application programs, including two categories:
  - **Strongly ordered** – a memory modification of one processor is **immediately visible** to all other processors.
  - **Weakly ordered** – a memory modification of one processor **may not be immediately visible** to all other processors.
- A **memory barrier (内存屏障)** is an instruction that forces any change in memory to be propagated (visible) to all other processors.
  - When a memory barrier instruction is performed, **all previous loads and stores are completed before any subsequent load / store operations are performed (within the same process)**.
- Even if instructions were reordered, the memory barrier ensures that the store operations are
  - Completed in memory
  - Visible to other processors before future load / store operations

Reordered Execution Example

- 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?

  100

- However, since the variables `flag` and `x` are independent of each other, the instructions:

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

  for Thread 2 may be reordered

- If this occurs, the output may be **0**!

Hardware Sync: (1) Memory Barrier

- Memory barrier:
  - All loads and stores are completed before any subsequent load or store operations are performed.
  - We could add a memory barrier 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
```

- For Thread 1 we guarantee that
  - The value of `flag` is loaded before the value of `x`.

- For Thread 2 we guarantee that
  - The assignment to `x` occurs before the assignment `flag`.

Hardware Sync: (2) Hardware Instructions

- Atomic instruction:
  - The instruction can not be interrupted by other instructions.
  - Appears to occur instantaneously from the view of other threads or processes executing concurrently.
  - Either completes entirely or has no effect at all
- Two abstract atomic instruction types:
  - **Test-and-Set** instruction
  - **Compare-and-Swap** instruction

The `test_and_set` Instruction

- Definition of `test_and_set` instruction:

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

- Properties
  - Executed atomically
  - Set the new value of passed parameter to `true`
  - Returns the original value of passed parameter

The `compare_and_swap` Instruction

- Definition of **`compare_and_swap`** instruction:

```
int compare_and_swap(int *value, int expected, int new_value)
  {
    int temp = *value;
    if (*value == expected)
        *value = new_value;
    return temp;
 }
```

- Properties
  - Executed atomically
  - Returns the original value of the passed parameter **`value`**
  - Set the variable **`value`** as the passed parameter **`new_value`** but only if **`*value == expected`** is true.
    - The swap happens only under this condition.

Hardware Instructions for Mutual Exclusion

`test_and_set` solution

- 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` solution

- 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 */
    }
```

Do they solve the critical-section problem?

Bounded-waiting with `compare_and_swap`

```
while (true) {
    /* entry section - acquire the lock */
    waiting[i] = true;
    key = 1;
    while (waiting[i] && key == 1) // use key to exit
        key = compare_and_swap(&lock, 0, 1);
    waiting[i] = false;
    /* critical section - release one process */
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = 0;
    else
        waiting[j] = false;
    /* remainder section */
}
```

Hardware Sync: (3) Atomic Variables

- Typically, instructions such as compare-and-swap are used as building blocks for other synchronization tools.

- One way to use is defining **atomic variable** that provides atomic updates (uninterruptible) on basic data types such as integers and booleans.

- For example:
  - Let **`sequence`** be an atomic variable
  - Let **`increment()`** be operation on the atomic variable **`sequence`**
  - The command:

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

ensures **`sequence`** is incremented without interruption:

Hardware Sync: (3) Atomic Variables

- The **`increment()`** function can be implemented with **`compare_and_swap`** instruction as follows:

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

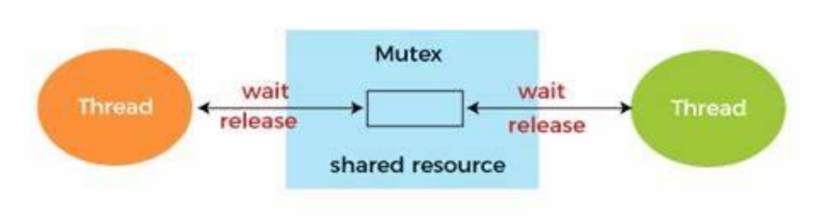
Goal: v → v+1

Synchronization Mechanisms: Mutex and Semaphore

Mutex Locks (互斥锁)

- OS designers build higher-level software tools to solve critical section problem, where the simplest is **`mutex`** lock
  - Mutex is a boolean variable indicating if lock is available or not
- Protect a critical section by
  - First **`acquire()`** a lock
  - Then **`release()`** the lock
- 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 (自旋锁)**

```
while (true) {
          acquire lock

              critical section

          release lock

remainder section
}
```



Semaphore (信号量)

- Semaphore is a synchronization tool that provides more sophisticated ways (than Mutex locks) for processes to synchronize their activities.
  - Semaphore ***S*** – integer variable

- Can only be accessed via two **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 (Cont.)

- Two types of semaphore:
  - **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 applying a counting semaphore ***S*** as a binary semaphore
- We can use semaphores to solve various synchronization problems

Semaphore Usage Examples

- Example 1: Solution to the CS Problem
  - Create a semaphore "**mutex**" initialized to 1

```
wait(mutex);
   CS
signal(mutex);
```

- Example 2: Consider P_1 and P_2 that with two statements S_1 and S_2 and the requirement that S_1 to happen before S_2
  - Create a semaphore "**synch**" initialized to 0

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

Semaphore Implementation

- Must guarantee that no two processes can execute the **`wait()`** and **`signal()`** on the same semaphore at the same time
  - 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
  - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections, so this is not a good solution

Semaphore Implementation with no Busy Waiting

- Implementation of **wait**:

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

- Implementation of **signal**:

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

- Semaphore stucture

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

Synchronization Mechanisms: Monitors

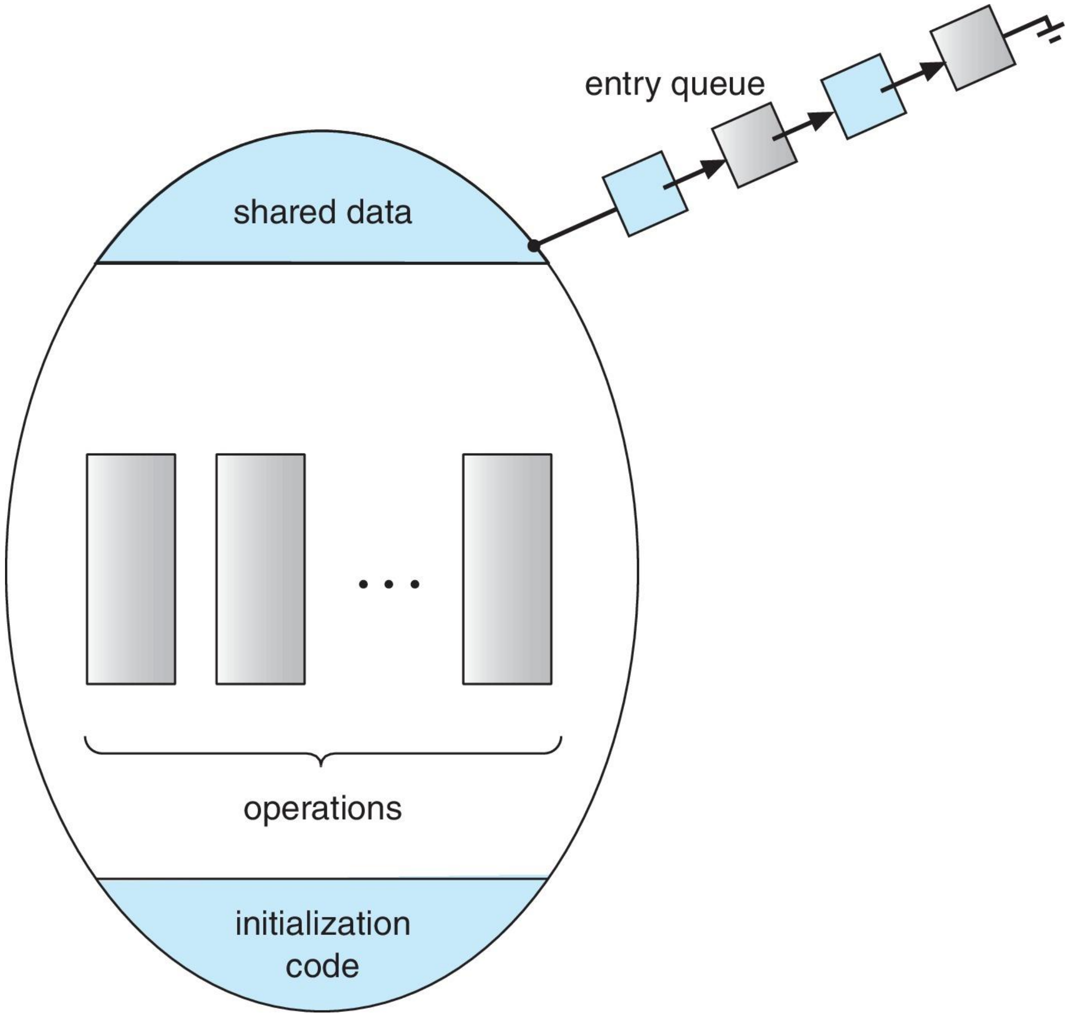
Monitors（管程）

- **Monitor**: A high-level abstraction that provides a convenient and effective mechanism for process synchronization
  - **Only one process may be active within the monitor at a time**

- Pseudocode syntax of a monitor:

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

  procedure P2 (…) { …. }

  procedure Pn (…) {……}

   initialization code (…) { … }
}
```

Schematic view of a Monitor

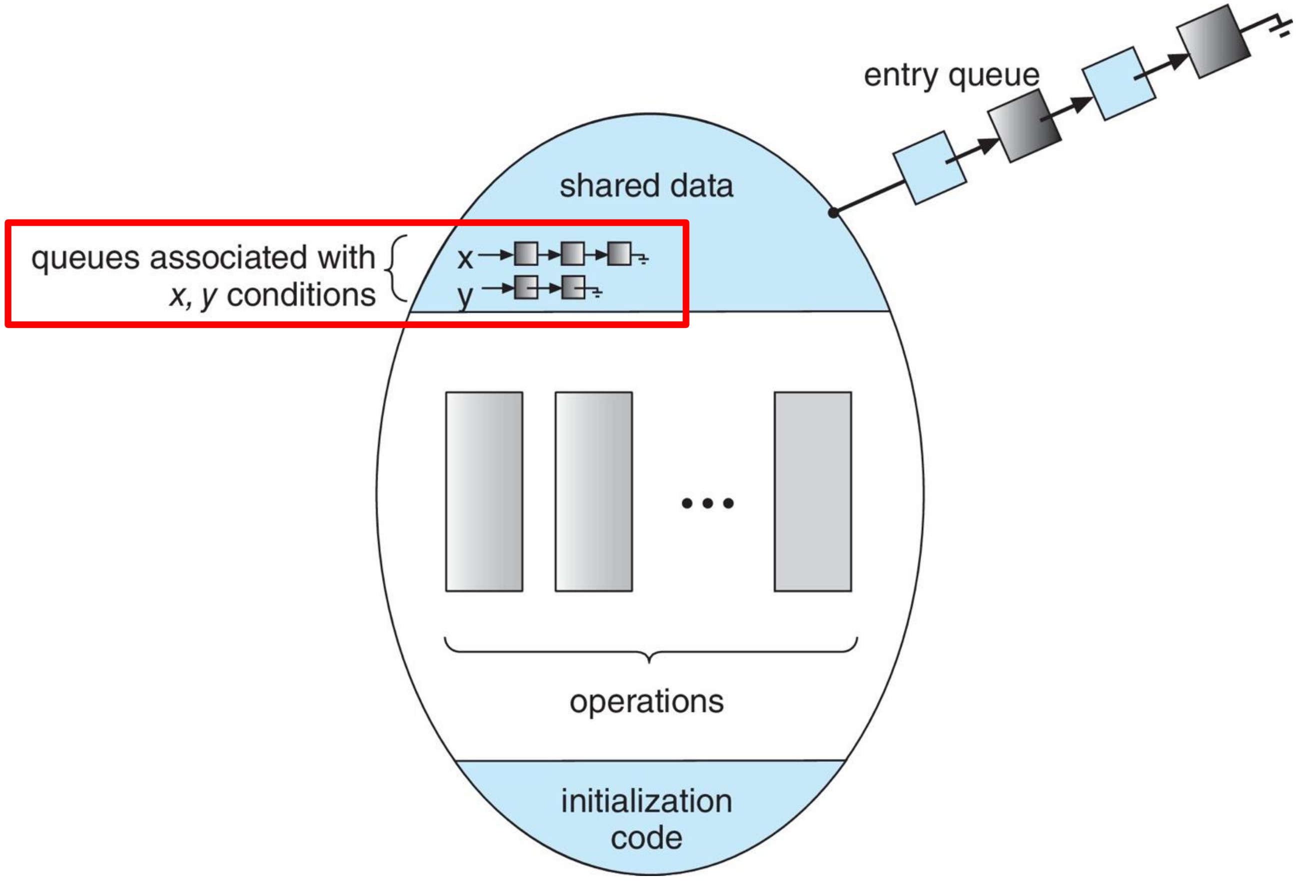


Condition Variables

- Monitors provide condition variables to allow threads to wait for specific conditions to become true before proceeding.
  - When a thread **waits** on a condition variable, **it releases the mutex**, allowing other threads to enter the monitor and potentially change the condition.
  - When the condition changes and another thread **signals** the condition variable, the waiting thread is **awakened and attempts to reacquire the mutex** before continuing its execution.

- 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 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 Choices

- If process P invokes `x.signal()`, with Q in `x.wait()` state, what should happen next?
  - At most one process is allowed active within the monitor.

- Condition variable options:
  - **Signal and wait** – P waits until Q leaves monitor or waits for another condition
  - **Signal and continue** – Q waits until P leaves the monitor or waits for another condition
  - **Compromise** - P leaves the monitor immediately after executing signal, Q is resumed

Implementation of Condition Variables

- For each condition variable ***x***, we have:

```
semaphore x_sem; // condition variable semaphore
int x_count = 0; // #threads that are waiting for "x"
semaphore next; // hanging semaphore
int next_count = 0; // number of hanging processes
semaphore mutex; // for mutual exclusion
```

Outer loop:

```
wait(mutex);
      ...
   body of F
      ...
if (next_count > 0)
   signal(next);
else
   signal(mutex);
```

- The operation **`x.wait()`** can be implemented as:

```
x_count++;
if (next_count > 0)
     signal(next);
 else
     signal(mutex);
wait(x_sem);
x_count--;
```

- The operation **`x.signal()`** can be implemented as:

```
if (x_count > 0) {
     next_count++;
     signal(x_sem);
     wait(next);
     next_count--;
}
```

Monitor for Single-Resource Allocation

`x.wait()` definition:

```
x_count++;
if (next_count > 0)
   signal(next);
else
   signal(mutex);
wait(x_sem);
x_count--;
```

`x.signal()` definition:

```
if (x_count > 0) {
   next_count++;
   signal(x_sem);
   wait(next);
   next_count--;
}
```

```
monitor ResourceAllocator
{
   boolean busy;
   condition x;

   void acquire(int time) {
      if (busy)
         x.wait(time);
      busy = true;
   }

   void release() {
      busy = false;
      x.signal();
   }

   initialization_code() {
      busy = false;
   }
}
```

signal and wait

Resuming Processes within a Monitor

- If several processes queued on condition variable **`x`**, and **`x.signal()`** is executed, which process should be resumed?
  - FCFS frequently not adequate
- Use the **conditional-wait** construct of the form

```
x.wait(c)
```

where:

  - **`c`** is an integer (called the priority number)
  - The process with lowest number (highest priority) is scheduled next

Synchronization Problem Formulations

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Sync Problems: (1) Bounded-Buffer Problem

- ***n*** buffers, each can hold one item
- Semaphore **`mutex`** initialized to the value 1: Control access to the shared buffer.
- Semaphore **`full`** initialized to the value 0: Produced item number.
- Semaphore **`empty`** initialized to the value n: Idle slot number.

- Producer process

```
while (true) {
   /* produce an item in
    next_produced */
   wait(empty);
   wait(mutex);
    /* add next produced to
    the buffer */
   signal(mutex);
   signal(full);
}
```

- Consumer process

```
while (true) {
   wait(full);
   wait(mutex);
   /* remove an item from buffer
    to next_consumed */
   signal(mutex);
   signal(empty);
   /* consume the item in next
    consumed */
}
```

Sync Problems: (2) Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - **Readers** – only read the data set; they do **not** perform any updates
  - **Writers** – can both read and write
- Problem defined:
  - **Allow multiple readers to read at the same time**
  - Only one single writer can access the shared data at each point of time
- Shared Data
  - Semaphore **`rw_mutex`** initialized to 1:
    - Ensure mutual modification to the data set and mutual-exclusion of reading and writing
  - Integer **`read_count`** initialized to 0
  - Semaphore **`mutex`** initialized to 1
    - Ensure mutual access to readcount

Sync Problems: (2) Readers-Writers Problem

- Writer process

```
while (true) {
 wait(rw_mutex);
 ...
 /* writing is performed*/
 ...
 signal(rw_mutex);
}
```

Goal: Allow multiple readers to read at the same time, but one writer each time.

- Reader process

```
while (true){
    wait(mutex);
    read_count++;
    if (read_count == 1) /* first reader */
        wait(rw_mutex);
    signal(mutex);
    ...
    /* reading is performed */
    ...
    wait(mutex);
    read_count--;
    if (read_count == 0) /* last reader */
       signal(rw_mutex);
    signal(mutex);
}
```

Sync Problems: (2) Readers-Writers Problem Variations

- The solution in previous slide can result in a starvation situation where a writer process may never writes.
  - It is a reader-preferred solution
- ***Reader-Preferred Solution*** – no reader kept waiting unless writer has permission to use shared object
  - No reader should wait for other readers to finish simply because a writer is waiting
- ***Writer-Preferred Solution*** – once writer is ready, it performs write asap
  - If a writer is waiting to access the object, no new readers may start reading

Sync Problems: (2) Writer-Preferred Solution

```
int read_count = 0, write_count = 0;
semaphore mutexrc = 1, mutexwc = 1, wrt = 1, rd = 1;
```

- Writer process

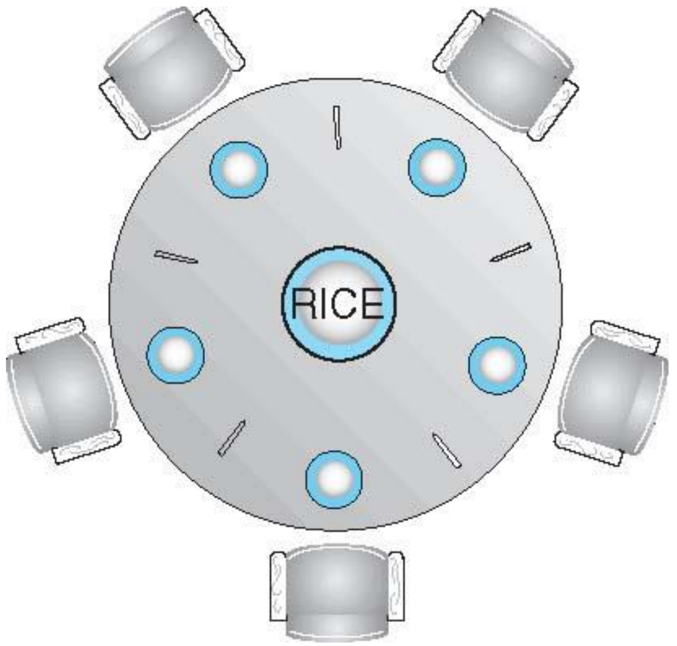
```
do {
    wait(mutexwc);
    writecount ++;
    if (writecount == 1)
            wait(rd);
    signal(mutexwc);

    wait(wrt);
    // writing is performed
    signal(wrt);

    wait(mutexwc);
    writecount--;
    if (writecount == 0)
            signal(rd);
    signal(mutexwc);
} while(TRUE);
```

- Reader process

```
do {
    wait(rd);
    wait(mutexrc);
    readcount ++;
    if (readcount == 1)
            wait(wrt);
    signal(mutexrc);
    signal(rd);

    //reading is performed

    wait(mutexrc);
    readcount--;
    if (readcount == 0)
            signal(wrt);
    signal(mutexrc);
} while(TRUE);
```

Sync Problems: (3) Dining-Philosophers Problem

- N philosophers (哲学家) sit at a round table with a bowel of rice in the middle.
  - They 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 the case of 5 philosophers
  - Shared data
    - Chopsticks
    - Semaphore chopstick [5] initialized to 1



Sync Problems: (3) Dining-Philosophers with Semaphores

- Semaphore Solution
- The structure of Philosopher i :

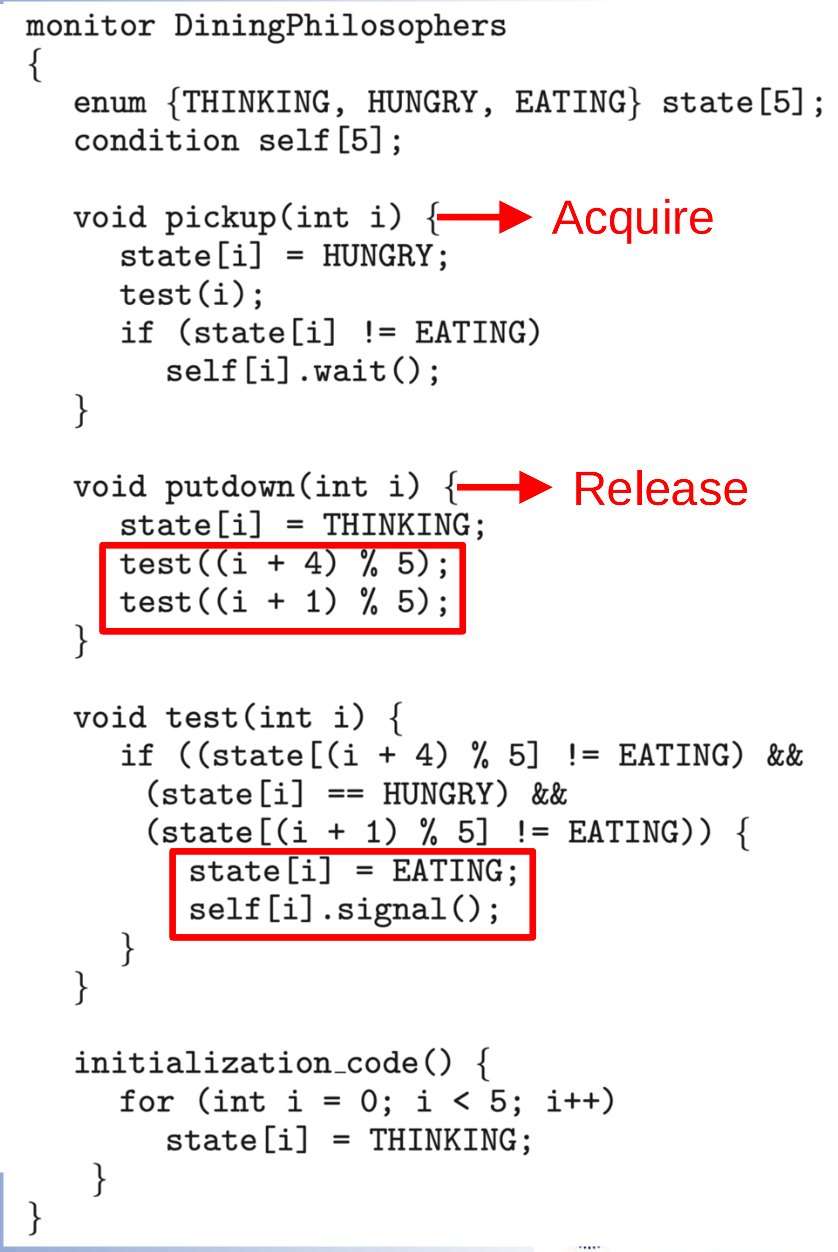
```
while(true){
    wait(chopstick[i] );
    wait(chopStick[(i + 1)%5]);

     /* eat for awhile */

    signal(chopstick[i] );
    signal(chopstick[(i + 1)%5]);

     /* think for awhile */

}
```

- What is the problem with this algorithm?
  - Deadlocks!

Sync Problems: (3) Dining-Philosophers with Monitors

- Deadlock-free solution with monitors.
- Three states for each philosopher:
  - `enum {THINKING, HUNGRY, EATING} state[5];`
- Condition variables for all philosophers:
  - `condition self[5];`
- Eat condition for a philosopher:
  - Two neighbors are not eating

  `(state[(i+4) % 5] != EATING)&`
  `(state[(i+1) % 5] != EATING)`
- Limitation:
  - One philosopher may stare to death



```
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();
   }

   void putdown(int i) {
      state[i] = THINKING;
      test((i + 4) % 5);
      test((i + 1) % 5);
   }

   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;
   }
}
```

Acquire

Release

Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

```
        P0                      P1
  ① wait (S);             ② wait (Q);
  ③ wait (Q);             ④ wait (S);
        .                       .
        .                       .
    signal (S);             signal (Q);
    signal (Q);             signal (S);
```

- **Starvation** – indefinite blocking
  - A process may never be removed from the semaphore queue in which it is suspended

Summary

- A critical section is a code section where shared data may be manipulated and a possible race condition may occur.
- A solution to the critical-section problem must satisfy: (1) mutual exclusion, (2) progress, and (3) bounded waiting.
- Peterson’s solution does not work well on modern computer architectures.
- Hardware solutions to the critical-section problems: (1) memory barriers, (2) hardware instruction (e.g., compare-and-swap), and (3) atomic variables.
- Mutex and semaphores can be used to provide mutual exclusion
- A monitor uses condition variables to allow processes to wait for certain conditions and to signal one another when conditions are true.
- Classic problems of process synchronization include the bounded-buffer, readers–writers, and dining-philosophers problems.
  - Solutions can be developed using mutex locks, semaphores, monitors, and condition variables.

Homework

- Reading
  - Chapter 6
  - Chapter 7
- HW1 due on Mar 12 at 23:59!