Operating Systems

Dr. Shu Yin

Part II: Process Management

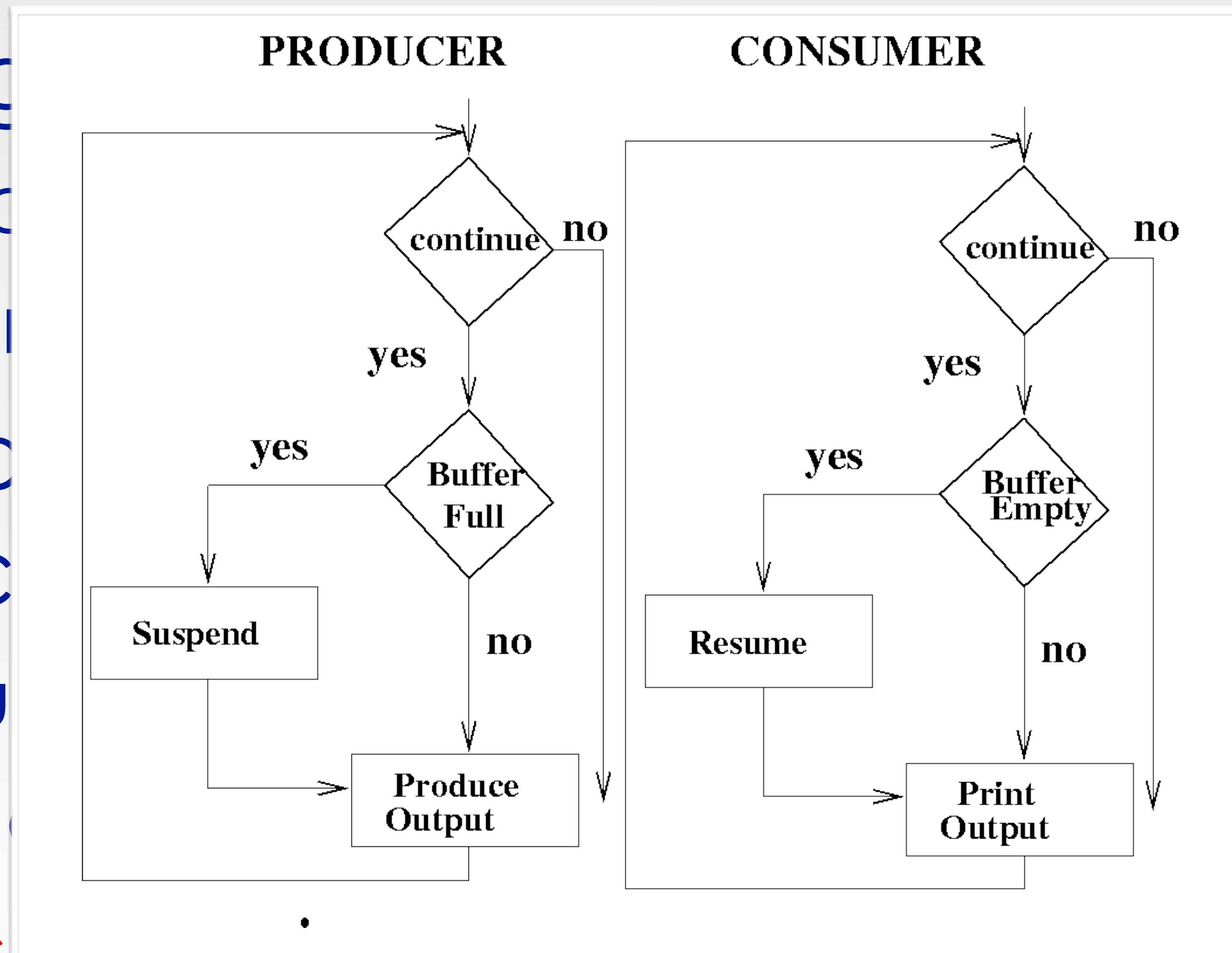
- Processes
- Threads
- Process Synchronization
- CPU Scheduling
- Deadlocks

Goals

- Introduce the Critical Section Problem
- Both SW and HW Solutions of the C-S Problem
- Classical Problems of Synchronization
- Tools to Solve Process Sync. Problems

Recall: Producer-Consumer Problem

- Paradig … ses
  - Produc
  - Consu … ete data
- Need b … filled by produc … ner
  - Unbou
  - Bound

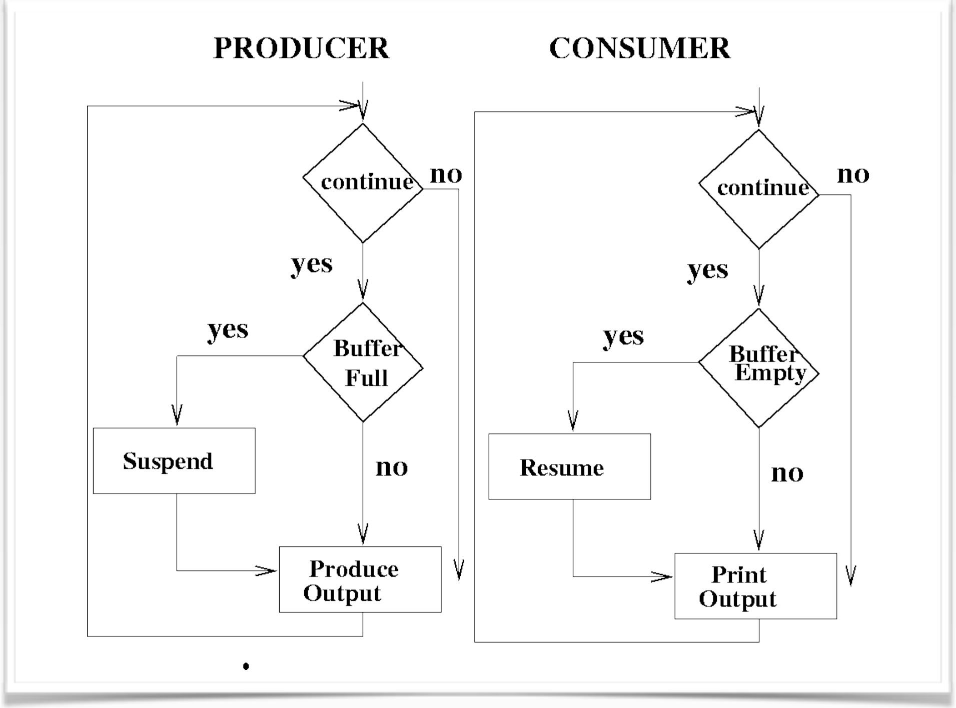


- Produc … chronize

Producer-Consumer Problem (cont.)

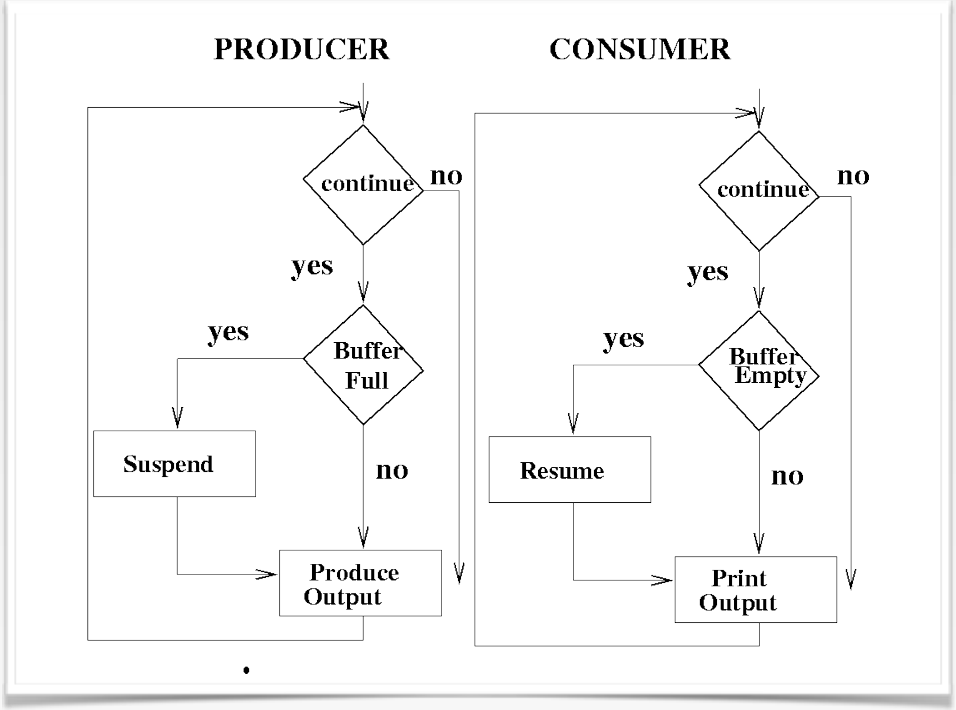
- Bounded-buffer using IPC (MP)

```
Producer
        item next_produced;
        while (true){
             /*produce an item in next
             produced*/
        send (next_produced)
        }
Consumer
        item next_consumed;
        while (true){
            while (in == out)
             receive(next_consumed);
             /*consume the item in the
             next consumed*/
}
```



Producer-Consumer Problem (cont.)

- Bounded-buffer using IPC (shared memory solution)
- Shared data



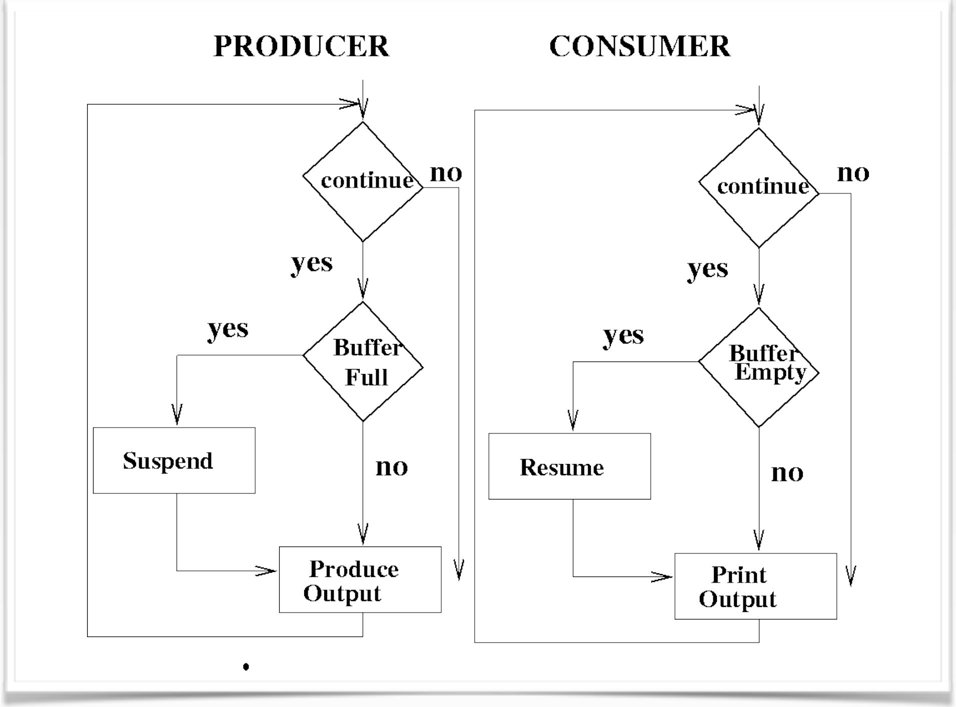

```
#define BUFFER_SIZE 10
typedef struct{
…
}item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Producer-Consumer Problem (cont.)

- Bounded-buffer using IPC (shared memory solution)
- Producer- creates filled buffer

```
item next_produced;
while (true){
  /*produce an item in next produced*/
  while (((in + 1) % BUFFER_SIZE) == out)
      ; /*do nothing*/
  buffer[in] = next_produced;
  in = (in + 1) % BUFFER_SIZE;
}
```

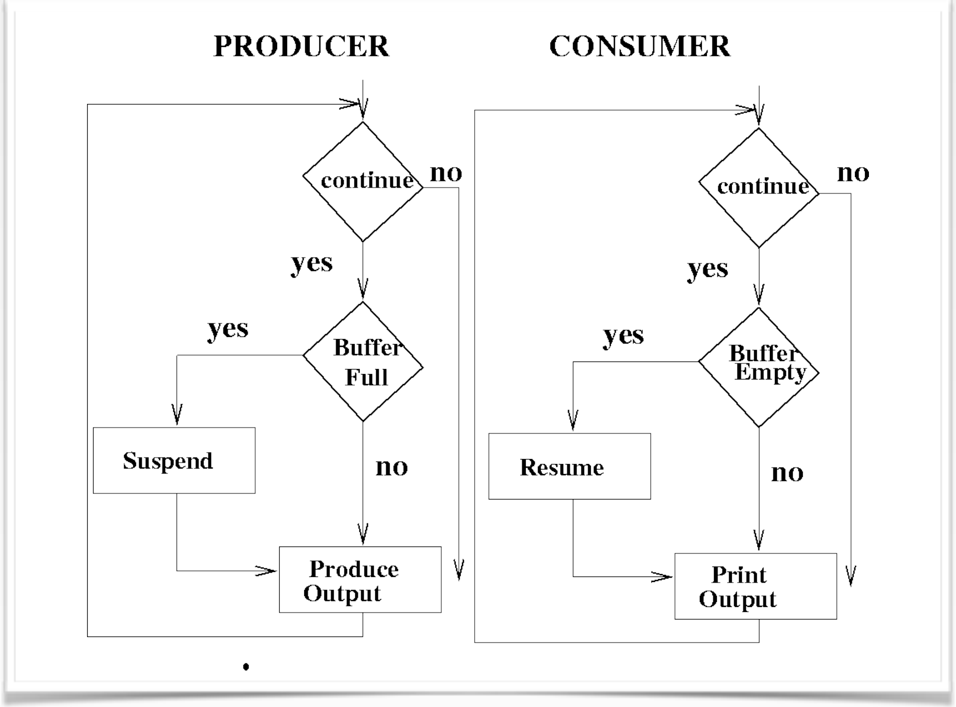


ShanghaiTech University 上海科技大学

Producer-Consumer Problem (cont.)

- Bounded-buffer using IPC (shared memory solution)
- Consumer- empties filled buffer

```
item next_consumed;
while (true){
 while (in == out)
  ;/*do nothing*/
 next_consumed = buffer[out];
 out = (out + 1) % BUFFER_SIZE;
 /*consume the item in the next consumed*/
}
```



Shared Data

- Concurrent access to shared data may result in data inconsistency
- Data consistency requires orderly execution
- Shared memory solution allows at most (n-1) items in the buffer at the same item

Bounded Buffer

- A solution that uses all N buffers is not that simple
  - Adding a variable *counter*
  - Initialized to 0
  - Incremented each time a new item is added

Bounded Buffer

- Shared data

```
#define BUFFER_SIZE 1-
typedef struct{
…
}item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```

上海科技大学
ShanghaiTech University

Bounded Buffer

- Producer process

```
while (true){
  /*produce an item in the next
  produced*/
  while (counter == BUFFER_SIZE)
       ; /*do nothing*/
  buffer[in] = next_produced;
  in = (in+1) % BUFFER_SIZE;
  counter ++;
until false;
}
```

- Consumer process

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

The Statements must be executed ATOMICALLY

上海科技大学
ShanghaiTech University

Problems is at the lowest level

- If threads are working on separate data, scheduling doesn't matter:

Thread A: `x = 1;` Thread B: `y = 2;`

- However, what about (initially, y=12)

Thread A: `x = 1; x = y+1;`

Thread B: `y = 2; y = y*2;`

- Or, what are the possible values of x

Thread A: `x = 1;` Thread B: `x = 2;`

上海科技大学
ShanghaiTech University

The Critical-Section Problem

- N processes all competing to use shared data
  - Structure of process P_i
  - Each process has a code segment (or critical section)
  - Shared Data is accessed in the critical section

```
repeat
  entry section /*enter critical section*/
      critical section /*access shared variables*/
  exit section /*leave critical section*/
      remainder section /*do other work*/
until false;
```


上海科技大学
ShanghaiTech University

Critical-Section Problem (cont.)

- Problem: Have to ensure that...
  - One process is executing in its critical section
  - No other process is allowed to execute in its critical section

Critical-Section Problem (cont.)

- Solutions - 3 requirements
  - Mutual Exclusion
  - Progress
  - Bounded Waiting
    - Assume that each process executes at a non-zero speed
    - No assumption concerning relative speed of n processes

Critical-Section Problem (cont.)

- Solutions - Initial Attempt
  - Only 2 processes, P_0 and P_1
  - General structure of process P_i

```
repeat
  entry section /*enter critical section*/
        critical section /*access shared variables*/
  exit section /*leave critical section*/
        remainder section /*do other work*/
until false;
```

  - Processes may share some common variables to synchronize their actions

上海科技大学 ShanghaiTech University

Critical-Section Handling in OS

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

上海科技大学 ShanghaiTech University

Critical-Section Problem: Algorithm 1

- Shared variables

```
var turn = 0; /*indicating whose turn it is to enter
               its critical section*/
turn = i; /*implies that process  Pi is allowed to enter
           its critical section*/
```

- Process P_0

```
repeat
  while turn ≠ 0 do noop;
      critical section
      turn = 1; /*leave critical section*/
      remainder section /*do other work*/
until false;
```

Satisfies mutual exclusion, but not progress

上海科技大学 ShanghaiTech University

Critical-Section Problem: Algorithm 2

- Shared variables

```
boolean flag[2]; /*initially flag[0]=flag[1]=false*/
flag[i] = true; /*implies that process Pi is ready to enter it critical section*/
```

- Process P_i

Can block indefinitely, but progress requirement not met.

```
repeat
  flag[i] = true;
  while (flag[j]) do noop;
       critical section
  flag[i] = false;
       remainder section
until false
```


上海科技大学 ShanghaiTech University

Critical-Section Problem: Algorithm 3

- Shared variables

```
boolean flag[2]; /*initially flag[0]=flag[1]=false*/
flag[i] = true; /*implies that process Pi is ready to enter it critical
section*/
```

- Process P_i

Does not satisfy mutual exclusion requirement.

```
repeat
  while (flag[j]) do noop;
  flag[i] = true;
        critical section
  flag[i] = false;
        remainder section
until false
```

Critical-Section Problem: Algorithm 4

- Shared variables

```
int turn;
boolean flag[2]; /*initially flag[0]=flag[1]=false*/
flag[i] = true; /*implies that process Pi is ready to enter it critical
section*/
```

- Process P_i

Meets all three requirements, solves the critical section problem for 2 processes

```
repeat
  flag[i] = true;
  turn = j;
  while (flag[j] && turn ==j) do noop;
       critical section
  flag[i] = false;
       remainder section
until false
```


上海科技大学 ShanghaiTech University

Peterson's solution

- Provable that the 3 critical-section requirements are met:
  - Mutual exclusion is preserved
    - P_i enters critical section only if:
      - either **`flag[j] = false`** or **`turn == i`**
  - Progress requirement is satisfied
  - Bounded-waiting requirements is met

```
repeat
   flag[i] = true;
   turn = j;
   while (flag[j] && turn ==j) do noop;
         critical section
   flag[i] = false;
         remainder section
until false
```


上海科技大学 ShanghaiTech University

Bakery Algorithm

- Critical section for n processes
  - Before entering its critical section, process receives a number. (Holder of the smallest number enters critical section)
  - If process P_i and P_j receive the same number
    - if $i \leq j$, then P_i is served first
    - else P_j is served first
  - The numbering scheme always generates number in increasing order of enumeration
    - i.e. 1,2,3,3,3,4,4,5,5

Bakery Algorithm (cont.)

- Notation
  - Lexicographic order (ticket #, process id #)
    - $(a, b) < (c, d)$ if $(a<c)$ or if $(a == c)$ and $(b < d)$
    - $\max(a_0, \ldots, a_{n-1}) = k$,
    - such that $k \geq a_i$ (for $i = 0, \ldots, n-1$, there is k,)
- Shared Data
  - boolean array: **`choosing[n]`** (initialized to false)
  - int array: **`turn[n]`** (initialized to 0)

上海科技大学
ShanghaiTech University

Bakery Algorithm (cont.)

```
int turn[n];
boolean choosing[n];
  int j;
  while(1){
    choosing[i] = true;
    turn[i] = 1 + max(turn[0], turn[1], … turn[n-1]);
    choosing[i] = false;
    for (j = 0; j < n; j++)
      if (j != i){
      //Wait until thread j receives its number:
        while (choosing[j]);
        //Wait until all threads with smaller numbers or with the same number
        //but with higher priority, finish their work:
        while (turn[j] != 0 && ((turn[j], j) < (turn[i],i)));
      }
    critical section;
    turn[i] = 0;
    non-critical section;
  }
```

/* notation: `(a, b) < (c,d)` is equivalent to `(a < c) || (a = c && b < d)*/`

上海科技大学 ShanghaiTech University

Supporting Synchronization

Programs	Shared Programs
High-Level API	Locks, Semaphores, Monitors, Send/ Receive, CCregions
Hardware	Load/Store, Disable Ints, Test/Set,Comp/Swap

上海科技大学 ShanghaiTech University

HW Solutions for Sync.

- Load/Store
  - Atomic operations required for synchronization
  - Shows how to protect a critical section with only atomic load and store

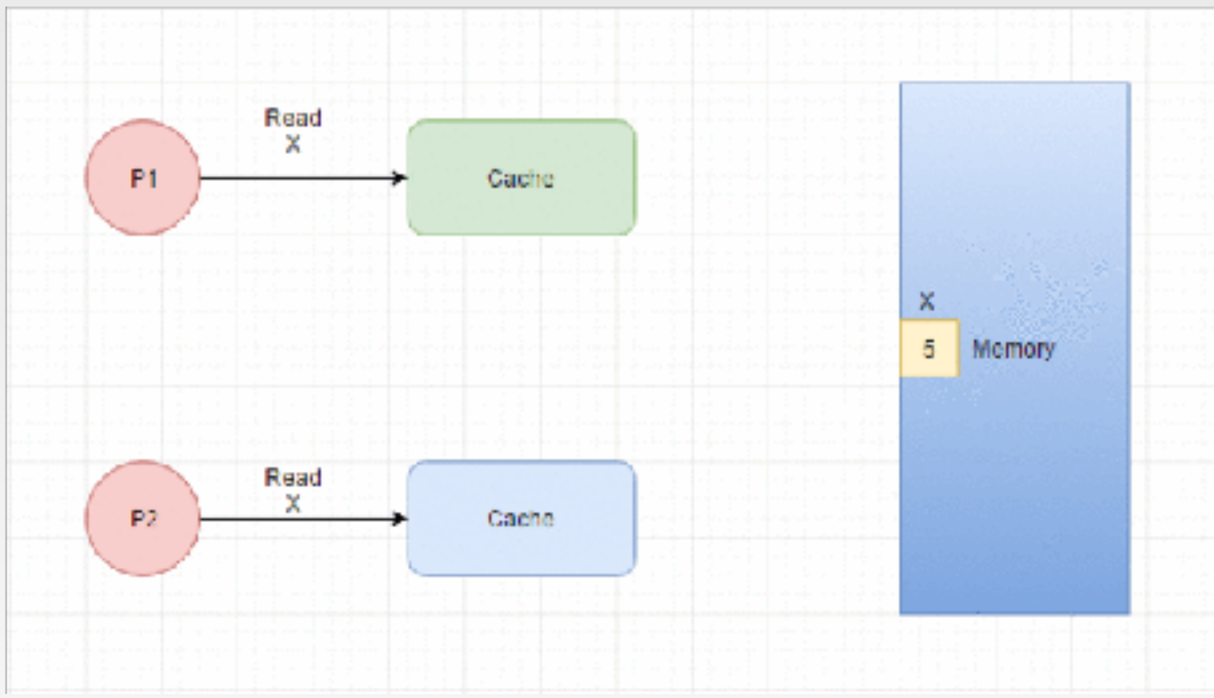
HW Solutions for Sync. (cont.)

- Mutual exclusion solutions presented depend on memory HW having R/W cycle
  - multiple R/W + same location ne time
  - Processors with caches but no cache coherency can NOT use the solutions

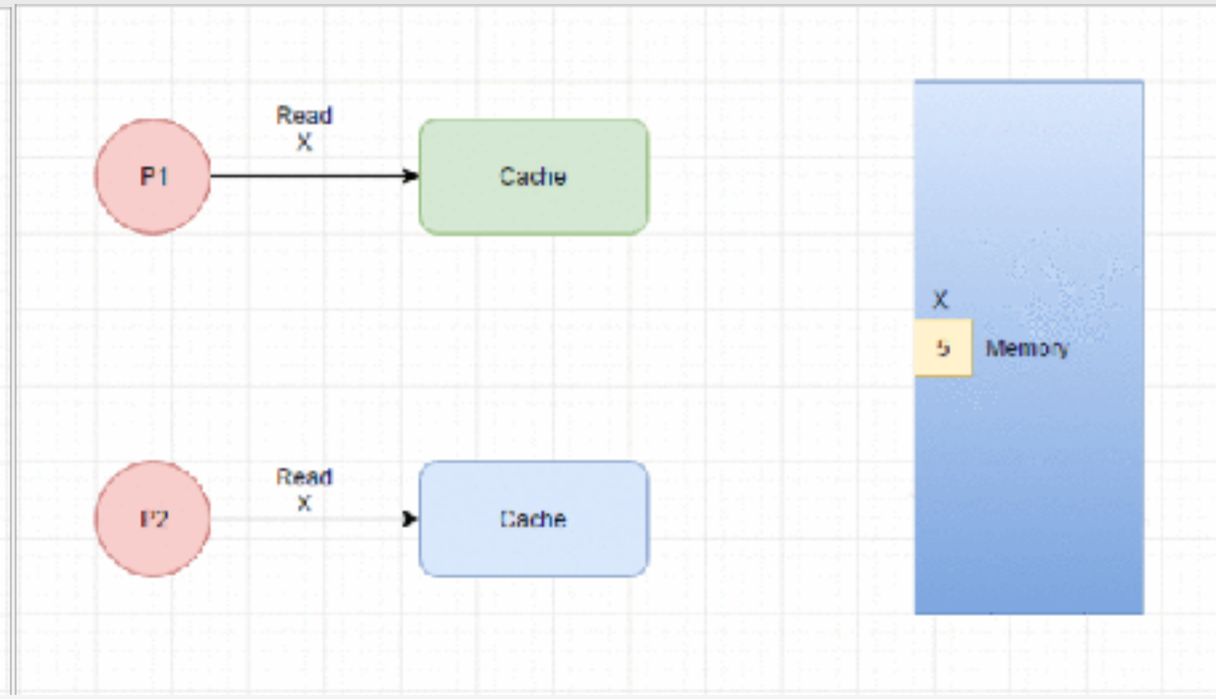
Would Not Work

上海科技大学
ShanghaiTech University

Recall: cache coherence



No Cache Coherence



Cache Coherence

Synchronization Hardware

- Based on idea of locking
  - Protecting critical regions via locks
- Uniprocessors - could disable interrupts
  - Currently running code→execute w/o preemption
  - Too inefficient on multiprocessor systems
- Modern machine→atomic instructions
  - Either test memory word and set value
  - Or swap contents of two memory words

上海科技大学
ShanghaiTech University

Locks

- Solutions to critical-section problem using Locks

```
do {
 acquire lock
    critical section
 release lock
    remainder section
}
```

`test_and_set` Instruction

- Test and modify the content of a word atomically - (`test_and_set` instruction)
- Similarly "SWAP" instruction

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

- Executed atomically
- Returns the original value of passed parameter
- Set the new value of passed parameter of "TRUE"

上海科技大学 ShanghaiTech University

Mutual Exclusion with **`test_and_set`**

- Shared boolean "lock", initialized to False
- Solution:

```
do {
 while (test_and_set(&lock))
     ; /*do nothing*/
          /*critical section*/
 lock = false;
          /*remainder section*/
}while (true)
```

`compare_and_swap` Instruction

- Definition

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

- Executed atomically
- Returns the original value of passed "value"
- Swap takes place only if "value == expected"

上海科技大学 ShanghaiTech University

Mutual Exclusion with **compare_and_swap**

- Shared int "lock" initialized to 0
- Solution:

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


上海科技大学
ShanghaiTech University

Bounded Mutual Exclusion with `test_and_set`

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

Mutex Locks

- SW tool to solve Critical-Section problem
- Simplest is Mutex Lock
  - First **`acquire()`** a lock
  - Then **`release()`** the lock
  - Boolean indicating if lock is on or not
- Calls must be Atomic
  - Usually via HW atomic instructions
- This solution requires busy waiting
  - spinlock

Mutex Locks (cont.)

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

release {
 available = true;
}

do {
 acquire lock;
    /*critical section*/
 release lock;
    /*remainder section*/
}while (true)
```

HW Support: Other Examples

```
swap (&address, register){ /*86*/
   temp = M[address];
   M[address] = register;
   register = temp;
}while (true)
```

```
compare&swap (&address, reg1, reg2){
/*68000*/
   if (reg1 == M[address]){
        M[address] = reg2;
        return success;
   } else{
        return failure;
   }
}
```

```
load-linked&store conditional
(&address){ /*R4000, alpha*/
   loop:
        ll r1, M[address];
        /*Can do arbitrary comp */
        move r2, 1;
        sc r2, M[address];
        beqz r2, loop;
   }
}
```

上海科技大学 ShanghaiTech University

Semaphore

- Synchronization tool
- More sophisticated way than Mutex Lock
- Semaphore S: int variable
- Can only be accessed via two atomic operations
  - `wait()` and `signal()`
  - Originally called `P()` and `V()`

上海科技大学
ShanghaiTech University

Semaphore (cont.)

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

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

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

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


上海科技大学 ShanghaiTech University

Semaphore (cont.)

- Usage
  - Counting semaphore
    - int range over an unrestricted domain
  - Binary semaphore
    - int range only between 0 and 1
    - Same as a Mutex Lock

Semaphore (cont.)

- Usage (cont.)
  - Consider P_1 and P_2 that require S_1 to happen before S_2
  - Create a semaphore “synch” initialized to 0

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

  - Can implement a counting semaphore S as a binary semaphore

Semaphore: Problem…

- Locks prevent conflicting actions on shared data
  - Lock before entering critical section
  - Lock before accessing shared data
  - Unlock when leaving, after accessing shared data
  - Wait if locked

Semaphore: Problem... (cont.)

- All synchronization involves waiting
  - Busy Waiting (spinlock)
  - Waiting thread may take cycles away from thread holding lock
  - OK for short time (prevents context switch)
  - Priority Inversion
- For longer runtimes, need to modify P and V so that processes can *block* an *resume*

上海科技大学 ShanghaiTech University

Semaphore: Implementation

- Must guarantee…NOT on the same semaphore at the same time
- Thus, **`wait`** and **`signal`** code are in placed in the critical section
  - Can have busy waiting in C-S implementation
  - But implementation code is short
  - Little busy waiting if C-S rarely occupied
- Apps may spend lot of time in C-S,

so no a good solution

上海科技大学 ShanghaiTech University

Semaphore: Implementation (cont.)

- Two operations
  - *block* - suspends the process that invokes it
  - *wakeup* - resumes the execution of a blocked process P

Semaphore: Implementation (cont.)

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - Value (type: integer)
  - Pointer to next record in the list

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

Semaphore: Implementation (cont.)

- Semaphore operations are now defined as:

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

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

Block/Resume Semaphore

- If process is blocked, enqueue PCB of process and call scheduler to run a different process

Block/Resume Semaphore (cont.)

- Semaphores are executed atomically
  - No two processes execute *wait* and *signal* at the same time
  - Mutex can be used to make sure that two processes do not change count at the same time
    - If an interrupt occurs while mutex is held, it will result in a long delay
    - Solution: Turn off interrupts during critical section

上海科技大学 ShanghaiTech University

Deadlock and Starvation

- Deadlock
  - Two or more processes are waiting indefinitely for an event that can be caused by only only of the waiting processes

```
      P0                   P1
wait (S);            wait (Q);
wait (Q);            wait (S);
…                    …
signal (S);          signal (Q);
signal (Q);          signal (S);
```

Deadlock and Starvation (cont.)

- Starvation
  - Indefinite blocking
  - A process may never be removed from the semaphore queue in which it is suspended
- Priority Inversion
  - Scheduling problem
  - Lower-priority process holds a lock needed by higher-priority process
  - Solved via priority-inheritance protocol

Classical Problems of Sync.

- Bounded-Buffer problem
- Readers and Writers problem
- Dining-Philosophers problem

Bounded-Buffer Problem

- n buffers, one item each
- Semaphore mutex (initialized to 1)
- Semaphore full (initialized to 0)
- Semaphore empty (initialized to n)

Bounded-Buffer Problem (cont.)

- Producer process (creates filled buffers)

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


上海科技大学
ShanghaiTech University

Bounded-Buffer Problem (cont.)

- Consumer process (empties filled buffers)

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

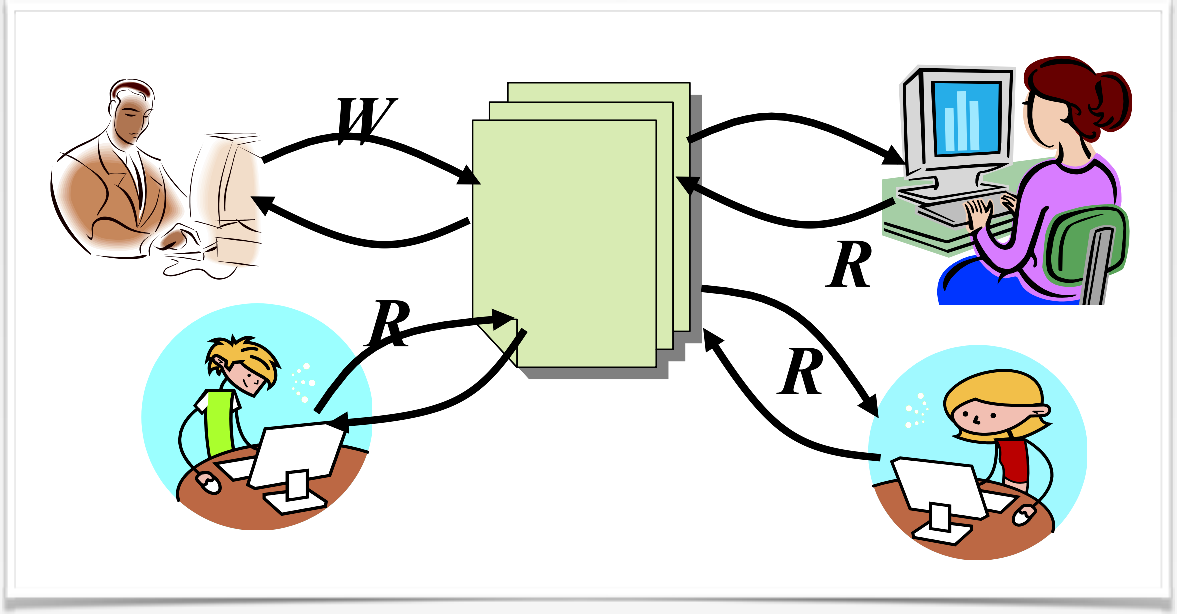
Discussion

- Asymmetry?
  - Producer does: **`wait(empty), signal(full)`**
  - Consumer does: **`wait(full), signal(empty)`**
- Is order of wait()'s important?
  - Yes! Can cause deadlock
- Is order of signal()'s important?
  - No, except that it might affect scheduling efficiency

上海科技大学 ShanghaiTech University

Readers/Writers Problem

- Motivation: shared database
  - Two classes of users
    - Readers - no modify
    - Writers - read&modify
  - Is using a single lock on the whole database sufficient?
    - Like to have many readers at the same time
    - Only one writer at a time



上海科技大学 ShanghaiTech University

Readers-Writers Problem (cont.)

- Shared Data
  - Data set
    - Semaphore **`rw_mutex`** initialized to 1
    - Semaphore **`mutex`** initialized to 1
    - Int **`read_count`** initialized to 0

Readers-Writers Problem (cont.)

- The structure of a writer process

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

Readers-Writers Problem (cont.)

- The structure of a reader process

```
do {
     wait (mutex) ;
     read_count ++;
     if (read_count == 1)
     wait (rw_mutex) ;
    signal(mutex) ;
    …
     /*reading is performed*/
    …
    wait (mutex) ;
     read count --;
     if (read_count == 0)
    signal (rw_mutex) ;
    signal (mutex) ;
} while (true) ;
```

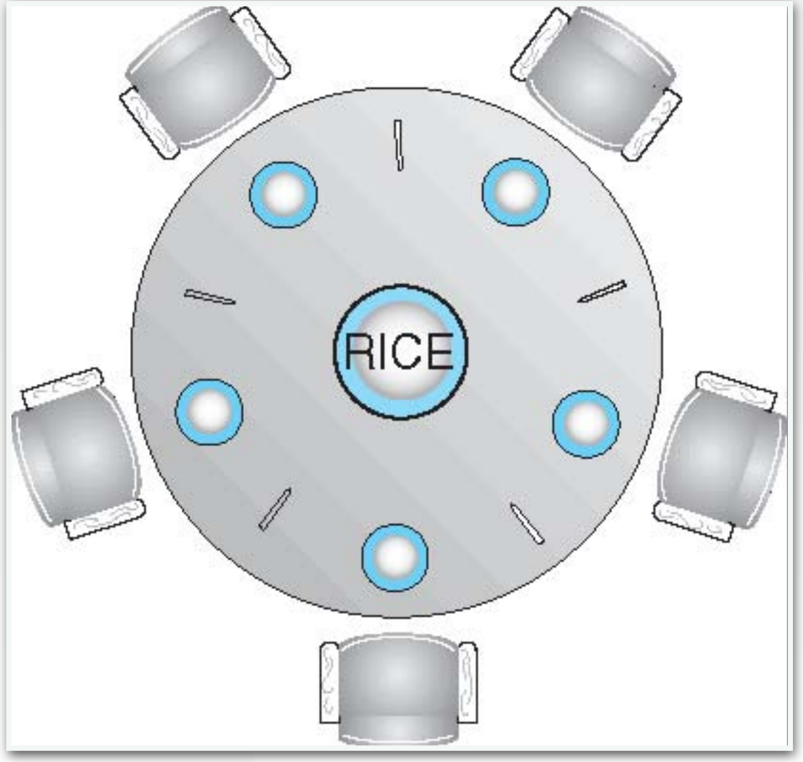
上海科技大学 ShanghaiTech University

Readers-Writers Problem Variations

1. No reader kept waiting unless writer has permission to use shared object
2. Once writer is ready, it performs the write ASAP

Dining-Philosophers Problem

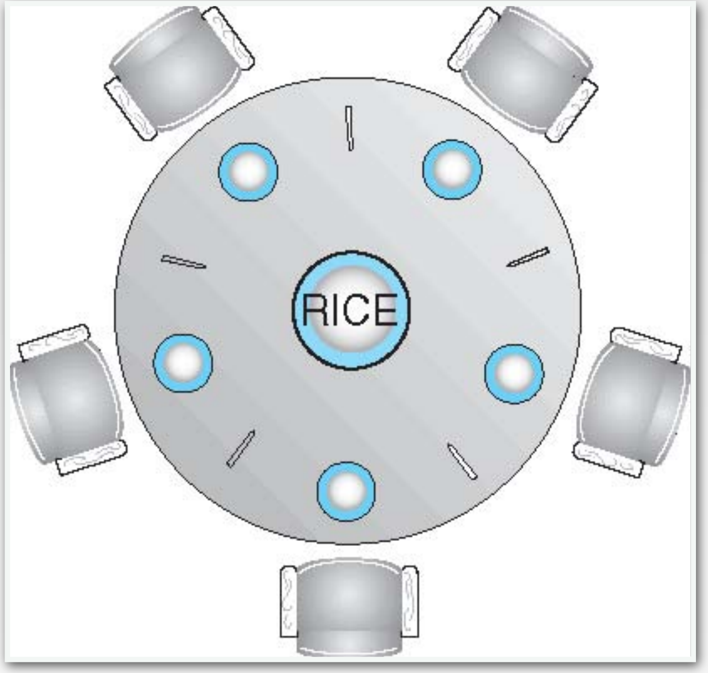
- Either thinking or eating
- Don't interact with neighbors
- Pick up 2 chopsticks to eat
  - One at a time
  - Need both to eat
  - Then release both when done
- Shared Data
  - Bowl of rice (data set)
  - Semaphore **`chopstick[5]`** (init to 1)



上海科技大学
ShanghaiTech University

Dining-Philosopher Problem (cont.)

- 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's the problem with this algorithm?

上海科技大学
ShanghaiTech University

Dining-Philosopher Problem (cont.)

- What's the problem?
  - Deadlock handling
    - Allow at most 4 philosophers to be sitting simultaneously at the table
    - Allow a philosopher to pick up the chopsticks only if both are available (picking must be done in a critical section)
    - Use an asymmetric solution

Higher Level Synchronization

- Timing errors are still possible with semaphores
  - **`signal(mutex)`** … **`wait(mutex)`**
  - **`wait(mutex)`**…**`wait(mutex)`**
  - **`wait(mutex)`**… **`/*forget to signal*/`**
- Deadlock and starvation are possible

上海科技大学
ShanghaiTech University

Motivation for Other Sync. Constructs

- Semaphores are a huge step up form loads and stores
  - Problem is that semaphores are dual purpose
    - Used for both mutex and scheduling constraints
    - E.g.: The fact that flipping of wait()'s in bounded buffer gives deadlock is not immediately obvious. How do you prove the correctness to someone?

Motivation for Other Sync. (cont.)

- Idea: allow manipulation of a shared variable only when condition is met
  - Conditional critical region
- Idea: use locks for mutual exclusion and condition variable for scheduling constraints
  - Monitor

Conditional Critical Regions

- High-level synchronization construct
- Shared variable v of type T
  - **`var v: shared T`**
- Variable v is accessed only inside statement
  - **`region v when B do S`**
  - B: boolean expression
  - No other process can access v while S is being executed

Critical Regions (cont.)

- Region referring to the same shared variable exclude each other in time
- When a process tries to execute the region statement
  - If B is true, S is executed
  - If B is false, the process is delayed until B becomes true, and no other process is in the region associated with v

```
region v when B do S() {
  semaphore xMutex, // mutual exclusion
            xDelay; //
        int xCount, // number of processes waiting for access to region v
            xTemp;  // number of processes allowed to check condition B

        p(xMutex);
        if(!B) { // not B? we have to wait until it is B
                xCount++;  // we're also waiting for not B to become B
                v(xMutex); // wait until you can go further
                p(xDelay); //
                while(!B) {     //
                        xTemp++;        //
                        if(xTemp < xCount) v(xDelay);   //
                        else v(xMutex); //
                        v(xDelay);      //
                }
                xCount--; // got out of while(!B), it means B is true so we're not waiting anymore
        }
        S();    // execute sequence of instructions
        if(xCount > 0) { // if someone's waiting for B...
                xTemp = 0; //
                v(xDelay); //
        } else v(xMutex); // if noone wants B, simply give up the right to the critical region
}
```

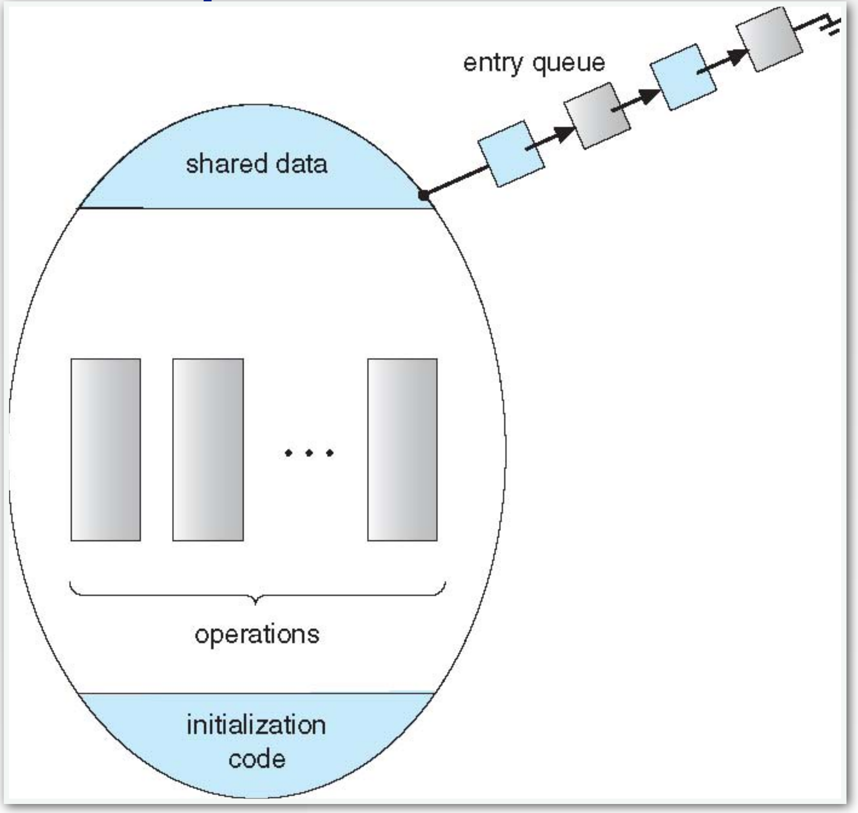
Monitors

- High-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes

```
monitor monitor_name {
      /* shared variable declare*/
       procedure P1(…){…};

       procedure Pn(…){…};

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



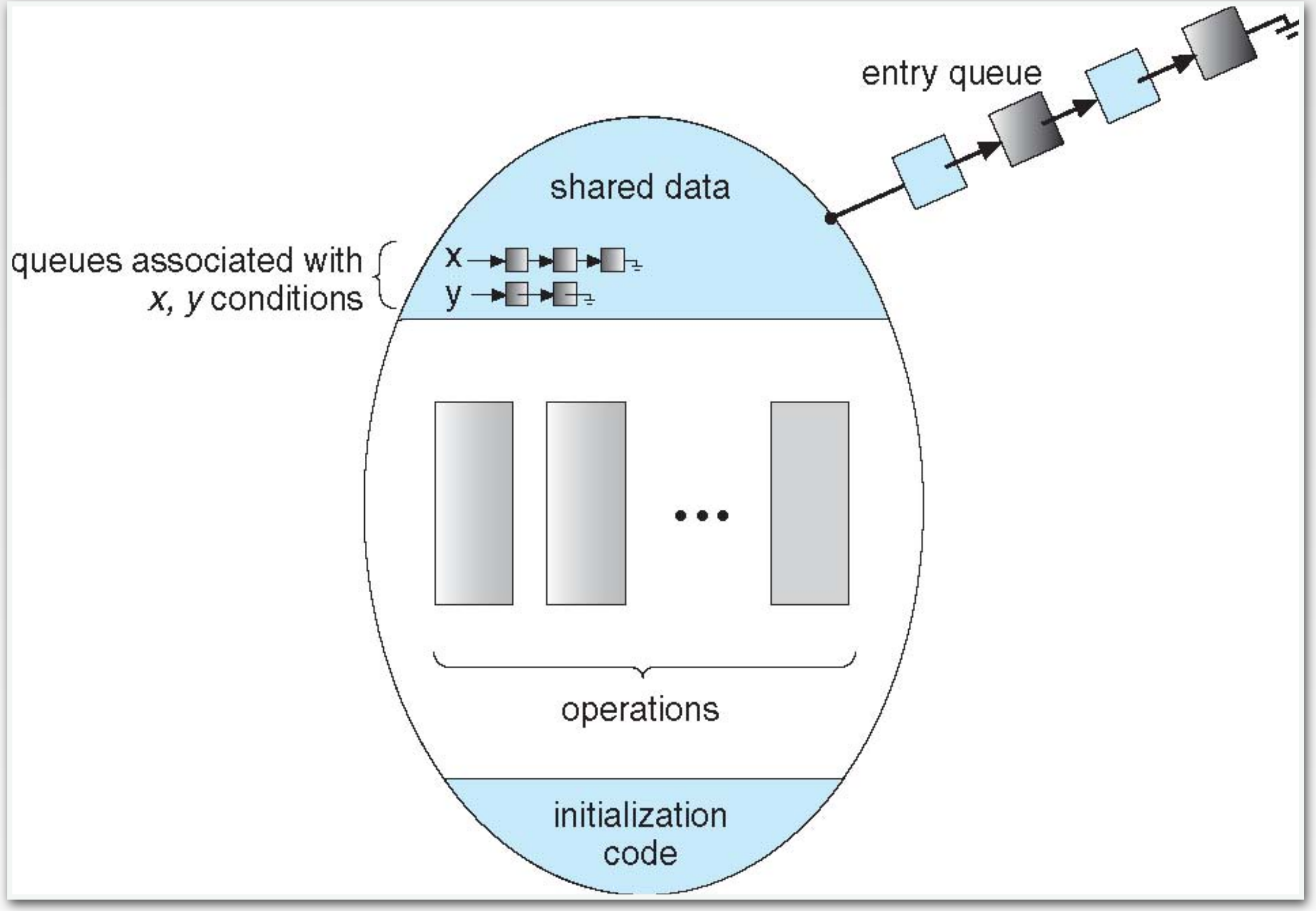
Monitor with Condition Variables

- Lock: provides mutual exclusion to share data
  - Always require before accessing shared data structure
  - Always release after finishing with shared data
  - Lock initially free
- Condition Variable: threads waiting for something inside a critical section
  - Key idea: go to sleep, automatically releasing lock at time going to sleep

Condition Variables

- **`condition x, y;`**
- Two operations are allowed on a condition variable:
  - **`x.wait()`**
    - that invoking this operation is suspended until another process invokes **`x.signal()`**
  - **`x.signal()`**
    - resume exactly one suspended process
    - If no process is suspended, then the signal operation has no effect

Monitor with Condition Variables



上海科技大学 ShanghaiTech University

Monitor with Condition Variables

- Condition variables Choices
  - If P invokes **`x.signal()`**, and Q is suspended in **`x.wait()`**, what should happen next?
    - Both Q and P cannot execute in parallel.
    - If Q is resumed, the P must wait
  - Operation includes
    - Signal and wait - P waits until Q either leaves the monitor or it waits for another condition
    - Signal and continue - Q waits until P leaves the monitor or it waits for another condition

Monitor Solution to Dining Philosophers

```
monitor DiningPhilosophers {
        enum {THINKING, HUNGRY, EATING} state[5];
        condition self[5];
        void pickup (int i){
             state[i] = HUNGRY;
             test(i); /*test left and right are not eating*/
             if (state[i] != EATING) self[i].wait;
        }
        void putdown (int i){
             state[i] = THINKING;
             /*test left and right neighbors*/
               test ((i + 4) % 5); /*signal on neighbor*/
               test ((i + 1) % 5); /*signal other neighbor*/
        }
        void test (int i) {
             if ((state[(i + 4) % 5] != EATING)) &&
             (state[i] == HUNGRY)&&
             (state[(i + 1) % 5] != EATING){
                     state[i] = EATING;
             self.signal();
             }
        }
        initialization_code() {
             for (int i = 0; i < 5; i++)
             state[i] = THINKING;
        }
}
```

上海科技大学 ShanghaiTech University

Monitor Solution to Dining Philosophers (cont.)

- Each philosopher *i* invokes the operation **`pickup()`** and **`putdown()`** in the following sequence

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

- No deadlock, but starvation is possible

上海科技大学 ShanghaiTech University

Monitor Implementation using Semaphore

- Variables

```
semaphore mutex; /*initially = 1*/
semaphore next;  /*initially = 0*/
int next_count = 0;
```

- Each procedure F will be replaced by

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

- Mutual Exclusion is ensured

上海科技大学 ShanghaiTech University

Monitor Implementation using Conditional Variables

- For each condition variable x, we have

```
semaphore x_sem; /*initially = 0*/
int x_count = 0;
```

- The operation **x.wait** can be implemented as

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

Monitor Implementation using Conditional Variables (cont.)

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

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


上海科技大学 ShanghaiTech University

Resuming Processes within Monitor

- If several processes queued on condition **`x`**, and **`x.signal()`** executed, which should be resumed?
- FCFS frequently not adequate
- *conditional-wait* construct of the form **`x.wait(c)`**
  - **`c`** is priority number
  - Processes with lowest number (highest priority) is scheduled next

上海科技大学
ShanghaiTech University

Single Resource Allocation

- a single resource
- among competing processes
- using priority numbers

```
R.acquire (t);
  …
  access the resource;
  …
R.release;
```

- R: an instance of type **`ResouceAllocator`**

上海科技大学
ShanghaiTech University

Monitor to Allocate Single Resource

```
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;
    }
}
```

Summary

- Why synchronization
  - cooperating sequential processes
  - share data
  - must provide mutual exclusion
  - critical section code used by only one precess/ thread at a time
- Synchronization problems
  - bounded-buffer/ reader-writer/ dining-philosopher
- Synchronization implementation
  - Monitor