Classical Synchronization Problems

Yi Shi
Fall 2018
Xi'an Jiaotong University

Review: Paradigms for Threads to Share Data

- We've looked at critical sections
  - Really, a form of locking
  - When one thread will access shared data, first it gets a kind of lock
  - This prevents other threads from accessing that data until the first one has finished
- Peterson's, hardware primitives and semaphores
  - We saw that semaphores make it easy to implement critical sections

Review: Semaphores

```
semaphore S;

P(S) {
    while(S ≤ 0)
        ;
    S--;
}

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

```
typedef struct semaphore {
    int value:
    ProcessList L;
} Semaphore;

void P(Semaphore *S) {
    S->value = S->value - 1;
    if (S->value < 0) {
            add this process to S->L;
            block();
    }
}

void V(Semaphore *S) {
    S->value = S->value + 1;
    if (S->value <= 0) {
            remove process P from S->L;
            wakeup(P);
    }
}
```

Semaphore Types

- Counting Semaphores:
  - Any integer
  - Used for applying/releasing resource
- Binary Semaphores
  - Value is limited to 0 or 1
  - Used for mutual exclusion (mutex)
- Synchronization
  - Value is limited to 0 or 1
  - Used for synchronization

```
Shared: semaphore S
Init: S = n;
```

```
Shared: semaphore S
Init: S = 1;
```

```
Shared: semaphore S
Init: S = 0;
```

Process i

```
P(S);
Critical Section
V(S);
```

Process i

```
Si;
V(S);
```

Process j

```
P(S);
Sj
```

Reminder: Critical Section

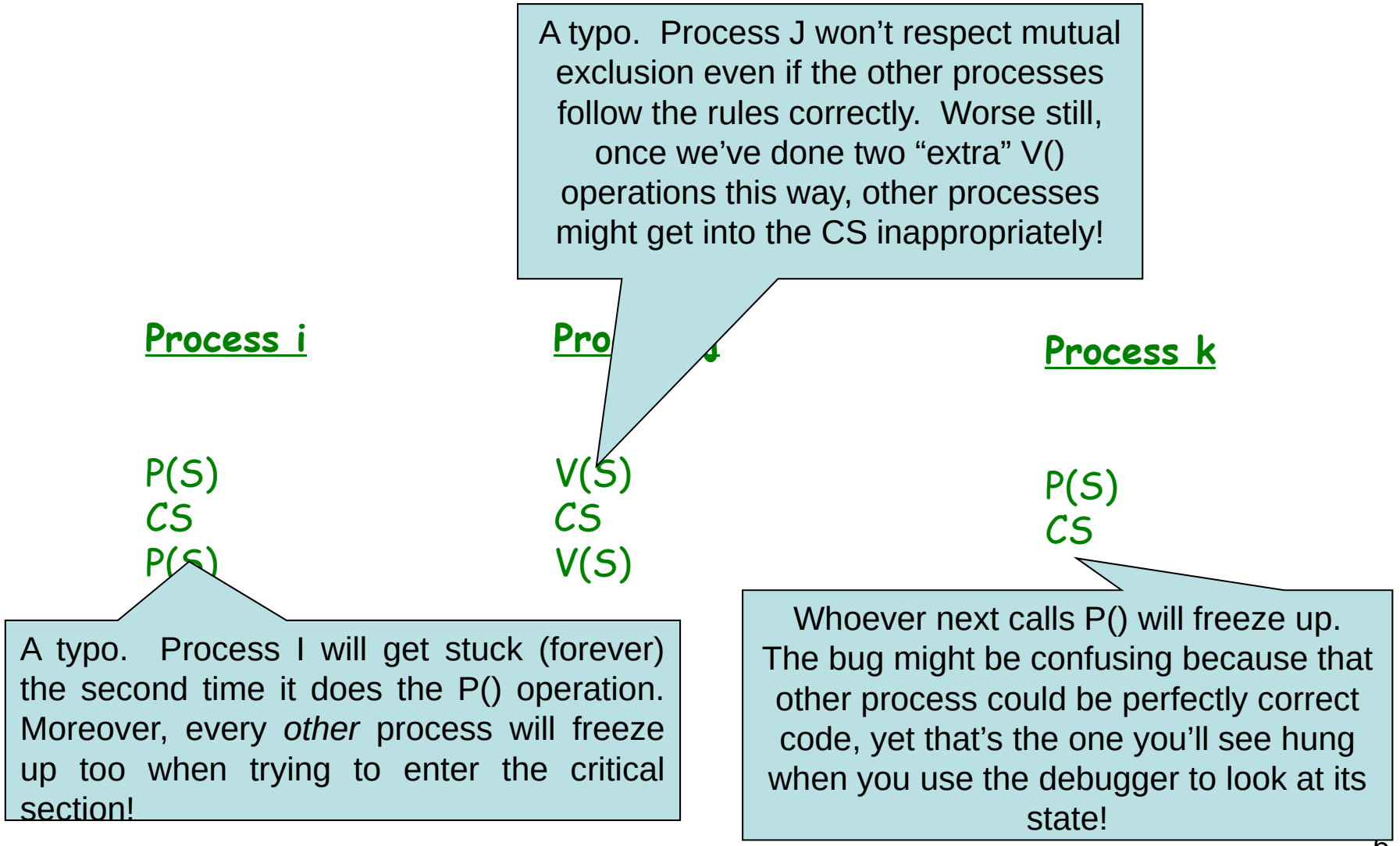
- Classic notation due to Dijkstra:

```
Semaphore mutex = 1;
CSEnter() { P(mutex); }
CSExit()  { V(mutex); }
```

- Other notation (more familiar in Java):

```
CSEnter() { mutex.wait(); }
CSExit() { mutex.signal(); }
```

Review: Common errors



Review: More common mistakes

- Conditional code that can break the normal top-to-bottom flow of code in the critical section
- Often a result of someone trying to maintain a program, e.g. to fix a bug or add functionality in code written by someone else

```
P(S)
if(something or other)
   return;
CS
V(S)
```

Goals for Today

- Classic Synchronization problems
  - Producer/Consumer with bounded buffer
  - Reader/Writer
  - Dining-Philosophers
- Solutions to classic problems via semaphores
- Solutions to classic problems via monitors

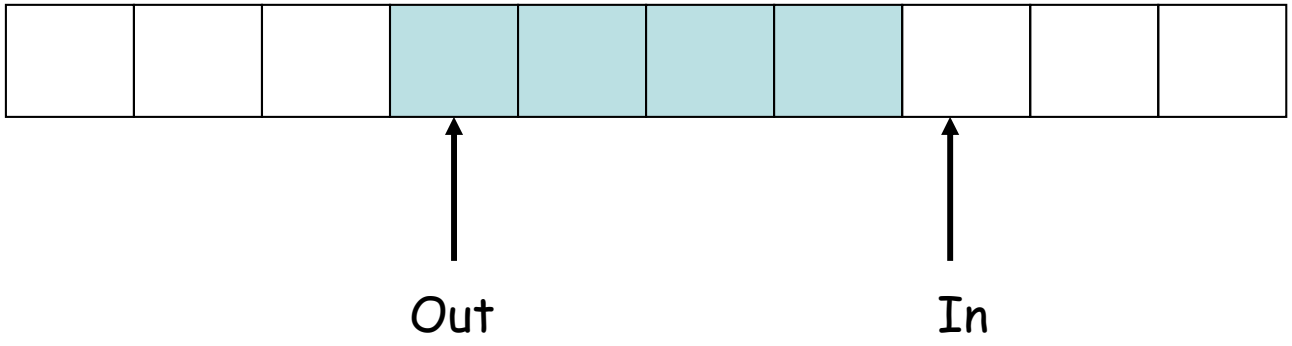
Bounded Buffer

- Arises when two or more threads communicate
  - some threads “produce” data that others “consume”.

- Example: preprocessor for a compiler “produces” a preprocessed source file that the parser of the compiler “consumes”

Producer-Consumer Problem

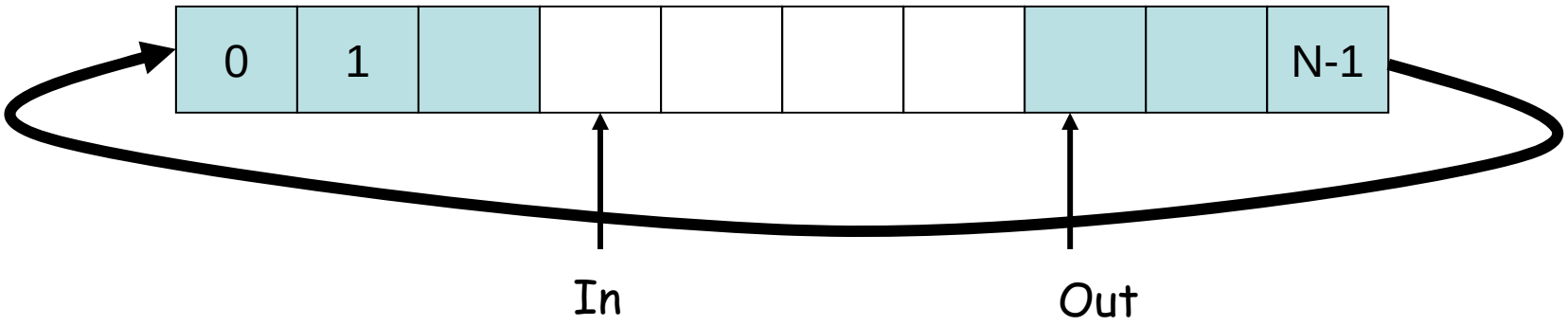
- Start by imagining an unbounded (infinite) buffer
- Producer process writes data to buffer
  - Writes to `In` and moves rightwards
- Consumer process reads data from buffer
  - Reads from *Out* and moves rightwards
  - Should not try to consume if there is no data



Need an infinite buffer

Producer-Consumer Problem

- Bounded buffer: size ‘N’
  - Access entry 0… N-1, then “wrap around” to 0 again
- Producer process writes data to buffer
  - Must not write more than ‘N’ items more than consumer “ate”
- Consumer process reads data from buffer
  - Should not try to consume if there is no data



Producer-Consumer Problem

- A number of applications:
  - Data from bar-code reader consumed by device driver
  - Data in a file you want to print consumed by printer spooler, which produces data consumed by line printer device driver
  - Web server produces data consumed by client's web browser
- Thought questions: where's the bounded buffer?

Producer-Consumer Problem

- Solving with semaphores
  - We'll use two kinds of semaphores
  - We'll use *counters* to track how much data is in the buffer
    - One counter counts as we add data and stops the producer if there are N objects in the buffer
    - A second counter counts as we remove data and stops a consumer if there are 0 in the buffer
  - Idea: since general semaphores can count for us, we don't need a separate counter variable
- Why do we need a second kind of semaphore?
  - We'll also need a mutex semaphore

Producer-consumer with a bounded buffer

- Problem Definition
  - Producer puts things into a shared buffer (wait if full)
  - Consumer takes them out (wait if empty)
  - Use a fixed-size buffer between them
    - Need to synchronize access to this buffer
- Correctness Constraints:
  - Consumer must wait for producer to fill buffers, if none full
    - synchronization
  - Producer must wait for consumer to empty buffers, if all full
    - synchronization
  - Only one thread can manipulate buffer queue at a time
    - mutual exclusion
- Remember why we need mutual exclusion
- General rule of thumb:
  Use a separate semaphore for each constraint
  - `Semaphore full;  // consumer's constraint`
  - `Semaphore empty; // producer's constraint`
  - `Semaphore mutex; // mutual exclusion`

Producer-Consumer Problem

```
Init: Semaphore mutex = 1;  /* for mutual exclusion*/
      Semaphore empty = N; /* number empty buf entries */
      Semaphore  full = 0;      /* number full buf entries */
      any_t buf[N];
      int tail = 0, head = 0;
```

Producer

```
void put(char ch)  {

   P(empty);
   P(mutex);

   // add ch to buffer
   buf[head%N] = ch;
   head++;

   V(mutex);
   V(full);
}
```

Consumer

```
char get() {

   P(full);
   P(mutex);

   // remove ch from buffer
   ch = buf[tail%N];
   tail++;

   V(mutex);
   V(empty);

    return ch;
}
```

Discussion about Bounded Buffer Solution

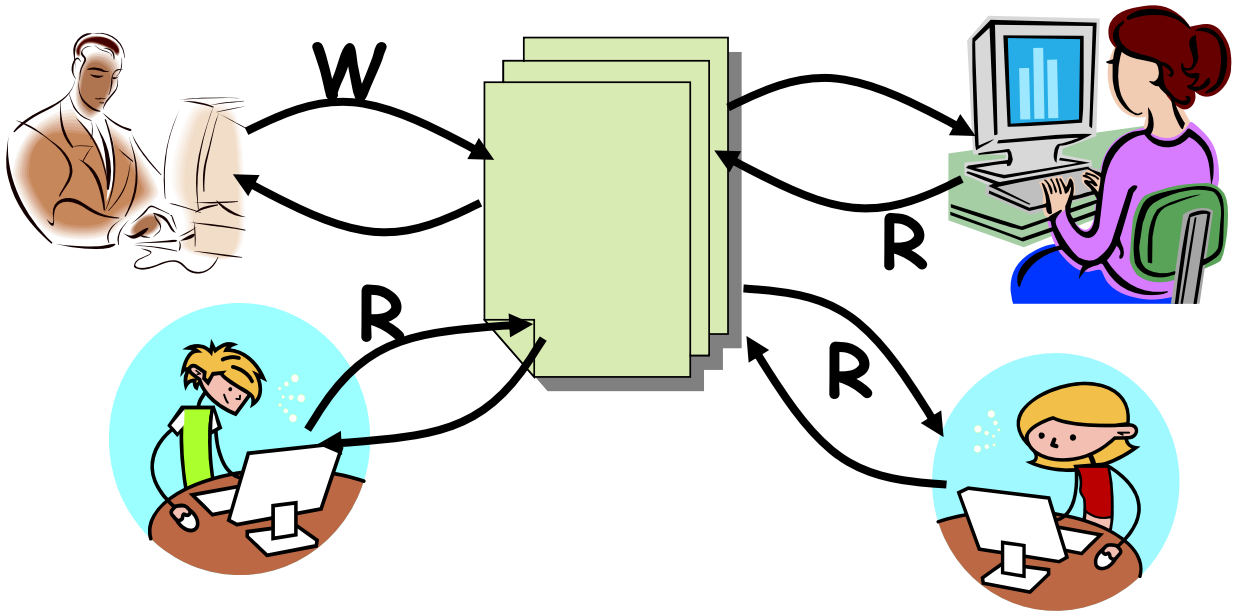
- Why asymmetry?
  - Producer does: `P(empty), V(full)`
  - Consumer does: `P(full), V(empty)`
- Is order of P's important?
  - Yes! Can cause deadlock
- Is order of V's important?
  - No, except that it might affect scheduling efficiency
- What if we have 2 producers or 2 consumers?
  - Do we need to change anything?

Readers and Writers

- In this model, threads share data that
  - some threads “read” and other threads “write”.

- Goal: allow multiple concurrent readers but only a single writer at a time, and if a writer is active, readers wait for it to finish

Readers/Writers Problem



- Motivation: Consider a shared database
  - Two classes of users:
    - Readers – never modify database
    - Writers – read and modify database
  - 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

Readers-Writers Problem

- Courtois et al 1971
- Models access to a database
  - A reader is a thread that needs to look at the database but won't change it.
  - A writer is a thread that modifies the database
- Example: making an airline reservation
  - When you browse to look at flight schedules, the web site is acting as a reader on your behalf
  - When you reserve a seat, the web site has to write into the database to make the reservation

Readers-Writers Problem

- Many threads share an object in memory
  - Some write to it, some only read it
  - Only one writer can be active at a time
  - Any number of readers can be active simultaneously
- Key insight: generalizes the critical section concept

- One issue we need to settle, to clarify problem statement.
  - Suppose that a writer is active and a mixture of readers and writers now shows up. Who should get in next?
  - Or suppose that a reader is active and a writer is waiting and an endless of stream of readers keeps showing up. Is it fair for them to become active?

- Ideally we would like a kind of back-and-forth form of fairness:
  - Once a reader is waiting, readers will get in next.
  - If a writer is waiting, one writer will get in next.

Readers-Writers Problem

- Problem Definition
  - Writers write to a database (wait if there's any readers or writers active)
  - Readers read from a database (wait if there's a writer active, continue if there's any readers active)
- Correctness Constraints:
  - Writers must wait for others to finish, if there's any readers or writers active
    - mutual exclusion
  - Readers must wait for the active writer to finish
    - mutual exclusion
  - If there's a reader active, any readers can read. So we need a counter to count reader's number and only one reader can visit the counter at a time
    - mutual exclusion
- General rule of thumb:
  Use a separate semaphore for each constraint
  - `Semaphore wrl;  // mutual exclusion for writer-writer or writer-reader`
  - `Semaphore mutex; // mutual exclusion for shared counter rcount`

Readers-Writers (Take 1)

```
Shared variables: Semaphore mutex, wrl;
                  integer rcount;

Init: mutex = 1, wrl = 1, rcount = 0;
```

Writer

```
do {

   P(wrl);
   . . .
   /*writing is performed*/
   . . .
   V(wrl);

}while(TRUE);
```

Reader

```
do {
   P(mutex);
   rcount++;
   if (rcount == 1)
      P(wrl);
   V(mutex);
   . . .
   /*reading is performed*/
   . . .
   P(mutex);
   rcount--;
   if (rcount == 0)
      V(wrl);
   V(mutex);
}while(TRUE);
```

Readers-Writers Notes

- If there is a writer
  - First reader blocks on **wrl**
  - Other readers block on **mutex**
- Once a reader is active, all readers get to go through
  - Trick question: Which reader gets in first?
- The last reader to exit signals a writer
  - If no writer, then readers can continue
- If readers and writers waiting on **wrl**, and writer exits
  - Who gets to go in first?
- Why doesn't a writer need to use **mutex**?

Does this work as we hoped?

- If readers are active, no writer can enter
  - The writers wait doing a P(wrl)
- While writer is active, nobody can enter
  - Any other reader or writer will wait
- But back-and-forth switching is buggy:
  - Any number of readers can enter in a row
  - Readers can “starve” writers
- With semaphores, building a solution that has the desired back-and-forth behavior is really, really tricky!
  - We recommend that you try, but not too hard…

Readers-Writers (Take 2)

Shared variables: Semaphore rmutex, wrl,wmutex,S;
integer rcount,wcount;
Init: rmutex = 1, wrl = 1, wmutex=1, S=1, rcount = 0, wcount=0;

Writer

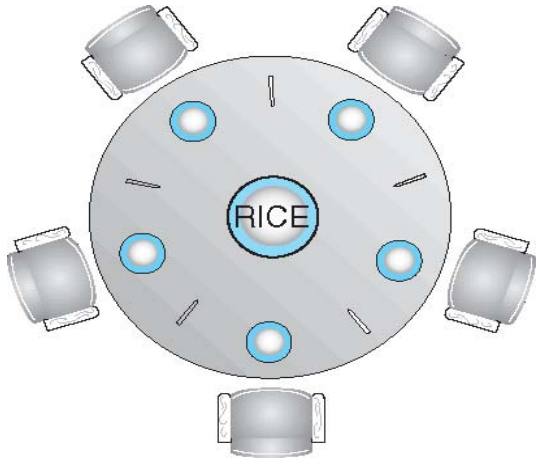
```
do {
    P(wmutex);
    wcount++;
    if (wcount ==1)
        P(S);
    V(wmutex);
    P(wrl);
    . . .
    /*writing is performed*/
    . . .
    V(wrl);
    P(wmutex);
    wcount--;
    if (wcount == 0)
        V(S);
    V(wmutex);
}while(TRUE);
```

Reader

```
do {
    P(S);
    P(rmutex);
    rcount++;
    if (rcount == 1)
        P(wrl);
    V(rmutex);
    V(S);
    . . .
    /*reading is performed*/
    . . .
    P(rmutex);
    rcount--;
    if (rcount == 0)
        V(wrl);
    V(rmutex);
}while(TRUE);
```

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 the case of 5 philosophers
  - Need both to eat, then release both when done
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1



Dining-Philosophers Problem Algorithm

Shared variables: chopstick[5];
Init: chopstick[i] = 1;

```
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?

postscript semaphores

- We seem to be using them in two ways
  - For mutual exclusion, the “real” abstraction is a critical section
  - But the bounded buffer example illustrates something different, where threads “communicate” using semaphores

- Semaphores are very “low-level” primitives
  - Users could easily make small errors
  - Similar to programming in assembly language
    - Small error brings system to grinding halt
  - Very difficult to debug

postscript 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
- Simplification: Provide concurrency support in compiler
  - Monitors

Monitors

- Hoare 1974
- Abstract Data Type for handling/defining shared resources
- Comprises:
  - Shared Private Data
    - The resource
    - Cannot be accessed from outside
  - Procedures that operate on the data
    - Gateway to the resource
    - Can only act on data local to the monitor
  - Synchronization primitives
    - Among threads that access the procedures

Structure of a Monitor

```
Monitor monitor_name
{
    // shared variable declarations

    procedure P1(. . . .) {
        . . . .
    }

    procedure P2(. . . .) {
        . . . .
    }
    .
    .
    procedure PN(. . . .) {
        . . . .
    }

    initialization_code(. . . .) {
        . . . .
    }
}
```

For example:

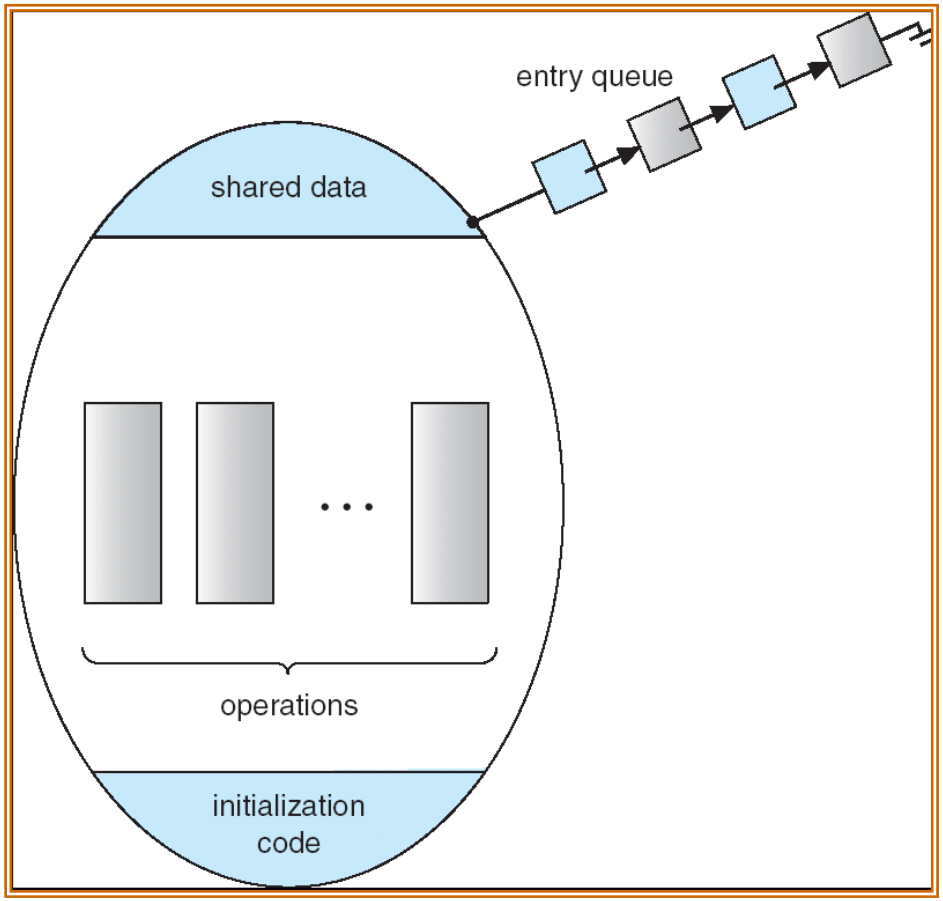
```
Monitor stack
{
    int top;
    void push(any_t *) {
        . . . .
    }

    any_t * pop() {
        . . . .
    }

    initialization_code() {
        . . . .
    }
}
```

only one instance of stack can be modified at a time

Monitor Semantics

- Monitors guarantee mutual exclusion
  - Only one thread can execute monitor procedure at any time
    - "*in the monitor*"
  - If second thread invokes monitor procedure at that time
    - It will block and wait for entry to the monitor
      - ⇒ Need for a wait queue
  - If thread within a monitor blocks, another can enter
- Effect on parallelism?

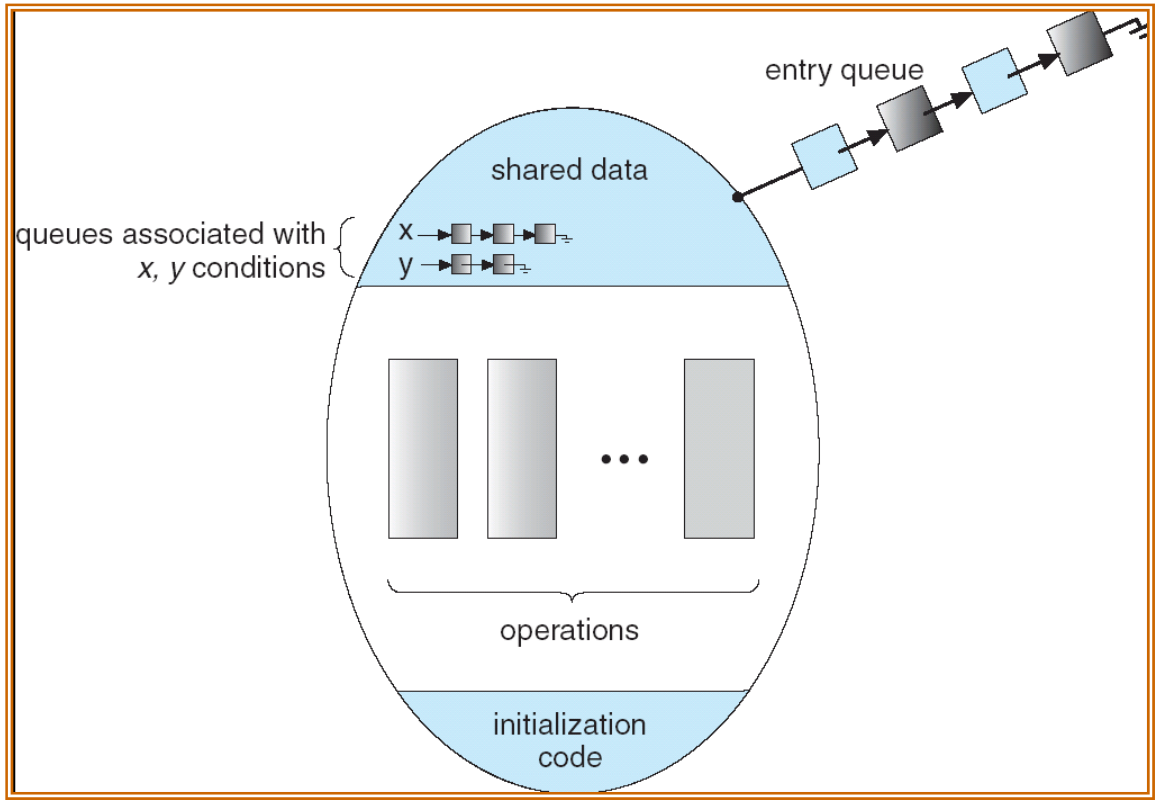
Schematic view of a Monitor



Synchronization Using Monitors

- Defines Condition Variables:
  - condition x;
  - Provides a mechanism to wait for events
    - Resources available, any writers
- 3 atomic operations on *Condition Variables*
  - x.wait(): release monitor lock, sleep until woken up
    ⇒ condition variables have waiting queues too
  - x.notify(): wake one process waiting on condition (if there is one)
    - No history associated with signal
  - x.broadcast(): wake all processes waiting on condition
    - Useful for resource manager
- Condition variables are not Boolean
  - If(x) then { } does not make sense

Monitor with Condition Variables



Monitor with Condition Variables

- If process P invokes **`x.signal(),`** with Q in **`x.wait()`** state, what should happen next?
  - If Q is resumed, then P must wait

- Options include
  - **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
  - Both have pros and cons – language implementer can decide
  - Monitors implemented in Concurrent Pascal compromise
    - P executing signal immediately leaves the monitor, Q is resumed
  - Implemented in other languages including Mesa, C#, Java

Dining Philosophers using Monitors

```
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 left and right neighbors
                   test((i + 4) % 5);
                   test((i + 1) % 5);
            }
```

Dining Philosophers using Monitors

```
        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;
        }
}
```

Producer Consumer using Monitors

```
Monitor Producer_Consumer {
   any_t buf[N];
   int n = 0, tail = 0, head = 0;
   condition not_empty, not_full;
   void put(char ch) {
            if(n == N)
                 wait(not_full);
            buf[head%N] = ch;
            head++;
            n++;
            signal(not_empty);
   }
   char get()  {
            if(n == 0)
                 wait(not_empty);
            ch = buf[tail%N];
            tail++;
            n--;
            signal(not_full);
            return ch;
   }
}
```

What if no thread is waiting when signal is called?

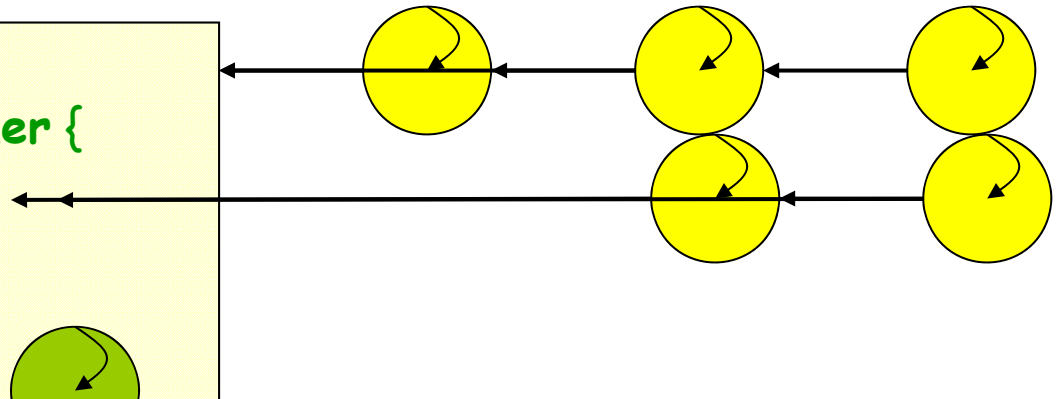
Signal is a “no-op” if nobody is waiting. This is very different from what happens when you call V() on a semaphore – semaphores have a “memory” of how many times V() was called!

Types of wait queues

- Monitors have several kinds of “wait” queues
  - Condition variable: has a queue of threads waiting on the associated condition
    - Thread goes to the end of the queue
  - Entry to the monitor: has a queue of threads waiting to obtain mutual exclusion so they can enter
    - Again, a new arrival goes to the end of the queue

Producer Consumer using Monitors

```
Monitor Producer_Consumer {
    condition not_full;
    /* other vars */
    condition not_empty;
    void put(char ch) {
            wait(not_full);
            . . .
            signal(not_empty);
    }
    char get()  {
            wait(not_empty);
            . . .
            signal(not_full);
    }
}
```



Types of Monitors

What happens on signal():

- Hoare: signaler immediately gives lock to waiter (theory)
  - Condition definitely holds when waiter returns
  - Easy to reason about the program
- Mesa: signaler keeps lock and processor (practice)
  - Condition might not hold when waiter returns
  - Fewer context switches, easy to support broadcast
  - Consider harder to work with this style of monitor
- Brinch Hansen: signaller must immediately exit monitor
  - So, notify should be last statement of monitor procedure
  - We recommend this approach!

Mesa-style monitor subtleties

```
char buf[N];
int n = 0, tail = 0, head = 0;
condition not_empty, not_full;
void put(char ch)
        if(n == N)
                wait(not_full);
        buf[head%N] = ch;
        head++;
        n++;
      signal(not_empty);
char get()
        if(n == 0)
                wait(not_empty);
        ch = buf[tail%N];
        tail++;
        n--;
        signal(not_full);
        return ch;
```

// producer/consumer with monitors

Consider the following time line:

0. initial condition: n = 0
1. c0 tries to take char, blocks on not_empty (releasing monitor lock)
2. p0 puts a char (n = 1), signals not_empty
3. c0 is put on run queue
4. Before c0 runs, another consumer thread c1 enters and takes character (n = 0)
5. c0 runs.

Condition Variables & Semaphores

- Condition Variables != semaphores
- Access to monitor is controlled by a lock
  - Wait: blocks on thread and gives up the lock
    - To call wait, thread has to be in monitor, hence the lock
    - Semaphore P() blocks thread only if value less than 0
  - Signal: causes waiting thread to wake up
    - If there is no waiting thread, the signal has no effect
    - V() increments value, so future threads need not wait on P()
    - Condition variables have no history
- However they can be used to implement each other

Hoare Monitors using Semaphores

For each procedure F:

```
P(mutex);

/* body of F */

if(next_count > 0)
    V(next);
else
    V(mutex);
```

Condition Var Wait: x.wait:

```
x_count++;
if(next_count > 0)
    V(next);
else
    V(mutex);
P(x_sem);
x.count--;
```

Condition Var Notify: x.notify:

```
If(x_count > 0) {
    next_count++;
    V(x_sem);
    P(next);
    next_count--;
}
```

Language Support

- Can be embedded in programming language:
  - Synchronization code added by compiler, enforced at runtime
  - Mesa/Cedar from Xerox PARC
  - Java: **synchronized, wait, notify, notifyall**
  - C#: **lock, wait (with timeouts) , pulse, pulseall**
- Monitors easier and safer than semaphores
  - Compiler can check, lock implicit (cannot be forgotten)

Summary

- Classic Synchronization problems
  - Producer/Consumer with bounded buffer
  - Reader/Writer
  - Dining-Philosophers
- Solutions to classic problems via semaphores
- Solutions to classic problems via monitors