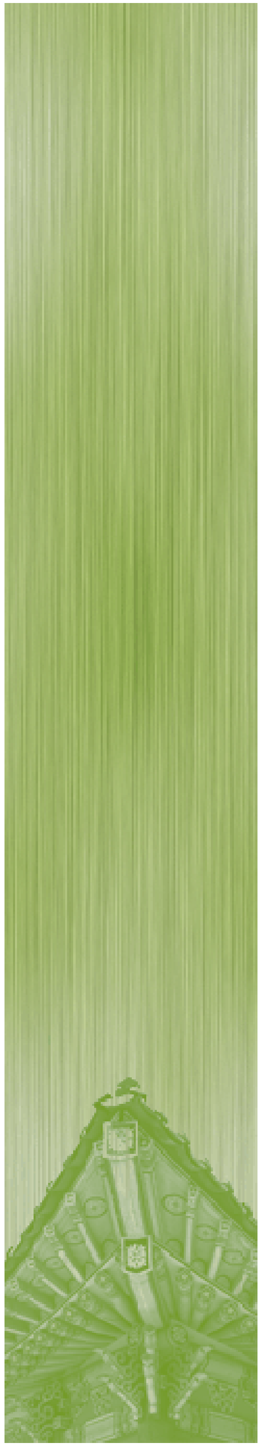


Synchronization II

Jin-Soo Kim (jinsookim@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu

Today's Topics

- **Spinlock is not enough**
  - What if a lock is held by others?
  - What if a condition is not met inside the critical section?
- **Higher-level synchronization mechanisms**
  - Semaphores
  - Monitors
  - Condition variables and mutex

Higher-level Synchronization

- **Motivation**
  - Spinlocks and disabling interrupts are useful only for very short and simple critical sections.
    - Wasteful otherwise
    - These primitives are "primitive" – don't do anything besides mutual exclusion.
  - Need higher-level synchronization primitives that
    - Block waiters
    - Leave interrupts enabled within the critical section
  - Two common high-level primitives:
    - Semaphores: binary (mutex) and counting
    - Monitors: mutexes and condition variables
  - We'll use our "atomic" locks as primitives to implement them.

Semaphores (1)

- **Semaphores**
  - A synchronization primitive higher level than locks.
  - Invented by Dijkstra in 1968, as part of the THE OS.
  - Does not require busy waiting.
  - Manipulated atomically through two operations:
    - Wait (S): decrement, block until semaphore is open
      = P(), after Dutch word for test, also called down()
    - Signal (S): increment, allow another to enter
      = V(), after Dutch word for increment, also called up()

Semaphores (2)

- **Blocking in semaphores**
  - Each semaphore has an associated queue of processes/threads.
  - When wait() is called by a thread,
    - If semaphore is “open”, thread continues.
    - If semaphore is “closed”, thread blocks, waits on queue.
  - Signal() opens the semaphore.
    - If thread(s) are waiting on a queue, one thread is unblocked.
    - If no threads are on the queue, the signal is remembered for next time a wait() is called.
  - In other words, semaphore has history.
    - The history is a counter.
    - If counter falls below 0, then the semaphore is closed.
    - wait() decreases the counter while signal() increases it.

Implementing Semaphores

```
typedef struct {
   int value;
   struct process *L;
} semaphore;
void wait (semaphore S)  {
   S.value--;
   if (S.value < 0)   {
      add this process to S.L;
      block ();
   }
}
void signal (semaphore S)  {
   S.value++;
   if (S.value <= 0)  {
      remove a process P from S.L;
      wakeup (P);
   }
}
```

wait() / signal()
are critical sections!
Hence, they must be
executed atomically
w.r.t.
each other.

HOW??

Types of Semaphores

- **Binary semaphore (a.k.a mutex)**
  - Guarantees mutually exclusive access to resource.
  - Only one thread/process allowed entry at a time.
  - Counter is initialized to 1.
- **Counting semaphore**
  - Represents a resource with many units available.
  - Allows threads/processes to enter as long as more units are available.
  - Counter is initialized to N (=units available).

Bounded Buffer Problem (1)

- **Producer/consumer problem**
  - There is a set of resource buffers shared by producer and consumer.
  - Producer inserts resources into the buffer.
    - Output, disk blocks, memory pages, etc.
  - Consumer removes resources from the buffer.
    - Whatever is generated by the producer
  - Producer and consumer execute in different rates.
    - No serialization of one behind the other
    - Tasks are independent
    - The buffer allows each to run without explicit handoff.

Bounded Buffer Problem (2)

- No synchronization

Producer

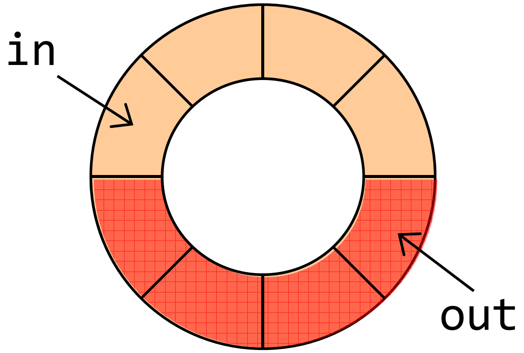
```
void produce(data)
{

  while (count==N);
  buffer[in] = data;
  in = (in+1) % N;
  count++;

}
```

```
int count;
```

```
struct item buffer[N];
int in, out;
```



Consumer

```
void consume(data)
{

  while (counter==0);
  data = buffer[out];
  out = (out+1) % N;
  count--;

}
```

Bounded Buffer Problem (3)

- **Implementation with semaphores**

Producer

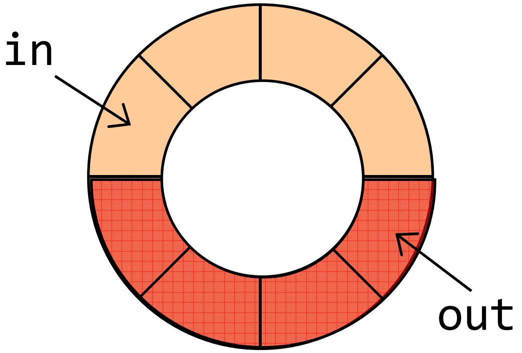
```
void produce(data)
{

  wait (empty);
  wait (mutex);
  buffer[in] = data;
  in = (in+1) % N;
  signal (mutex);
  signal (full);

}
```

```
Semaphore
        mutex = 1;
        empty = N;
        full = 0;
```

```
struct item buffer[N];
int in, out;
```



Consumer

```
void consume(data)
{

  wait (full);
  wait (mutex);
  data = buffer[out];
  out = (out+1) % N;
  signal (mutex);
  signal (empty);

}
```

Readers-Writers Problem (1)

- **Readers-Writers problem**
  - An object is shared among several threads.
  - Some threads only read the object, others only write it.
  - We can allow multiple readers at a time.
  - We can only allow one writer at a time.

- **Implementation with semaphores**
  - readcount – # of threads reading object
  - mutex – control access to readcount
  - rw – exclusive writing or reading

Readers-Writers Problem (2)

```
// number of readers
int readcount = 0;
// mutex for readcount
Semaphore mutex = 1;
// mutex for reading/writing
Semaphore rw = 1;

void Writer ()
{
   wait (rw);
   …
   Write
   …
   signal (rw);
}
```

```
void Reader ()
{
   wait (mutex);
   readcount++;
   if (readcount == 1)
      wait (rw);
   signal (mutex);
   …
   Read
   …
   wait (mutex);
   readcount--;
   if (readcount == 0)
      signal (rw);
   signal (mutex);
}
```

Readers-Writers Problem (3)

- **Note:**
  - If there is a writer
    - The first reader blocks on rw.
    - All other readers will then block on mutex.
  - Once a writer exits, all readers can fall through.
    - Which reader gets to go first?
  - The last reader to exit signals waiting writer.
    - Can new readers get in while writer is waiting?
  - When writers exits, if there is both a reader and writer waiting, which one goes next is up to scheduler.

Dining Philosopher (1)

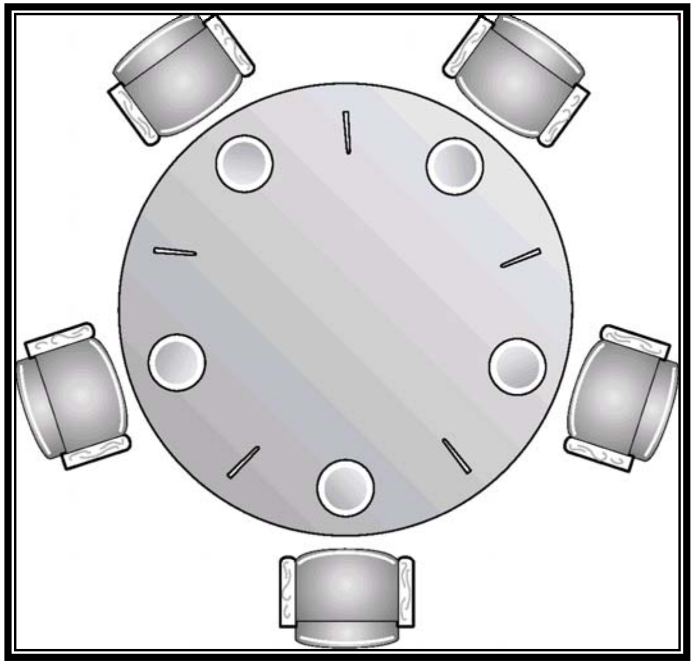
- **Dining philosopher problem**
  - Dijkstra, 1965.
  - Life of a philosopher
    - Repeat forever:

      Thinking

      Getting hungry

      Getting two chopsticks

      Eating



Dining Philosopher (2)

- **A simple solution**

```
Semaphore chopstick[N];  // initialized to 1
void philosopher (int i)
{
    while (1)  {
        think ();
        wait (chopstick[i]);
        wait (chopstick[(i+1) % N];
        eat ();
        signal (chopstick[i]);
        signal (chopstick[(i+1) % N];
    }
}
```

Dining Philosopher (3)

- **Deadlock-free version: starvation?**

```
#define N          5
#define L(i)       ((i+N-1)%N)
#define R(i)       ((i+1)%N)
void philosopher (int i)  {
  while (1)  {
    think ();
    pickup (i);
    eat();
    putdown (i);
  }
}
void test (int i)  {
  if (state[i]==HUNGRY &&
      state[L(i)]!=EATING &&
      state[R(i)]!=EATING)  {
     state[i] = EATING;
     signal (s[i]);
 }
```

```
Semaphore mutex = 1;
Semaphore s[N];
int state[N];

void pickup (int i)  {
  wait (mutex);
  state[i] = HUNGRY;
  test (i);
  signal (mutex);
  wait (s[i]);
}
void putdown (int i)  {
  wait (mutex);
  state[i] = THINKING;
  test (L(i));
  test (R(i));
  signal (mutex);
}
```

Problems with Semaphores

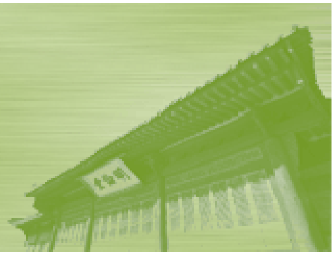
- **Drawbacks**
  - They are essentially shared global variables.
    - Can be accessed from anywhere (bad software engineering)
  - There is no connection between the semaphore and the data being controlled by it.
  - Used for both critical sections (mutual exclusion) and for coordination (scheduling).
  - No control over their use, no guarantee of proper usage.

- **Thus, hard to use and prone to bugs**
  - Another approach: use programming language support

Monitors (1)

- **Monitor**
  - A programming language construct that supports controlled access to shared data.
    - Synchronization code added by compiler, enforced at runtime.
    - Allows the safe sharing of an abstract data type among concurrent processes.
  - A monitor is a software module that encapsulates.
    - shared data structures
    - procedures that operate on the shared data.
    - synchronization between concurrent processes that invoke those procedures.
  - Monitor protects the data from unstructured access.
    - guarantees only access data through procedures, hence in legitimate ways.

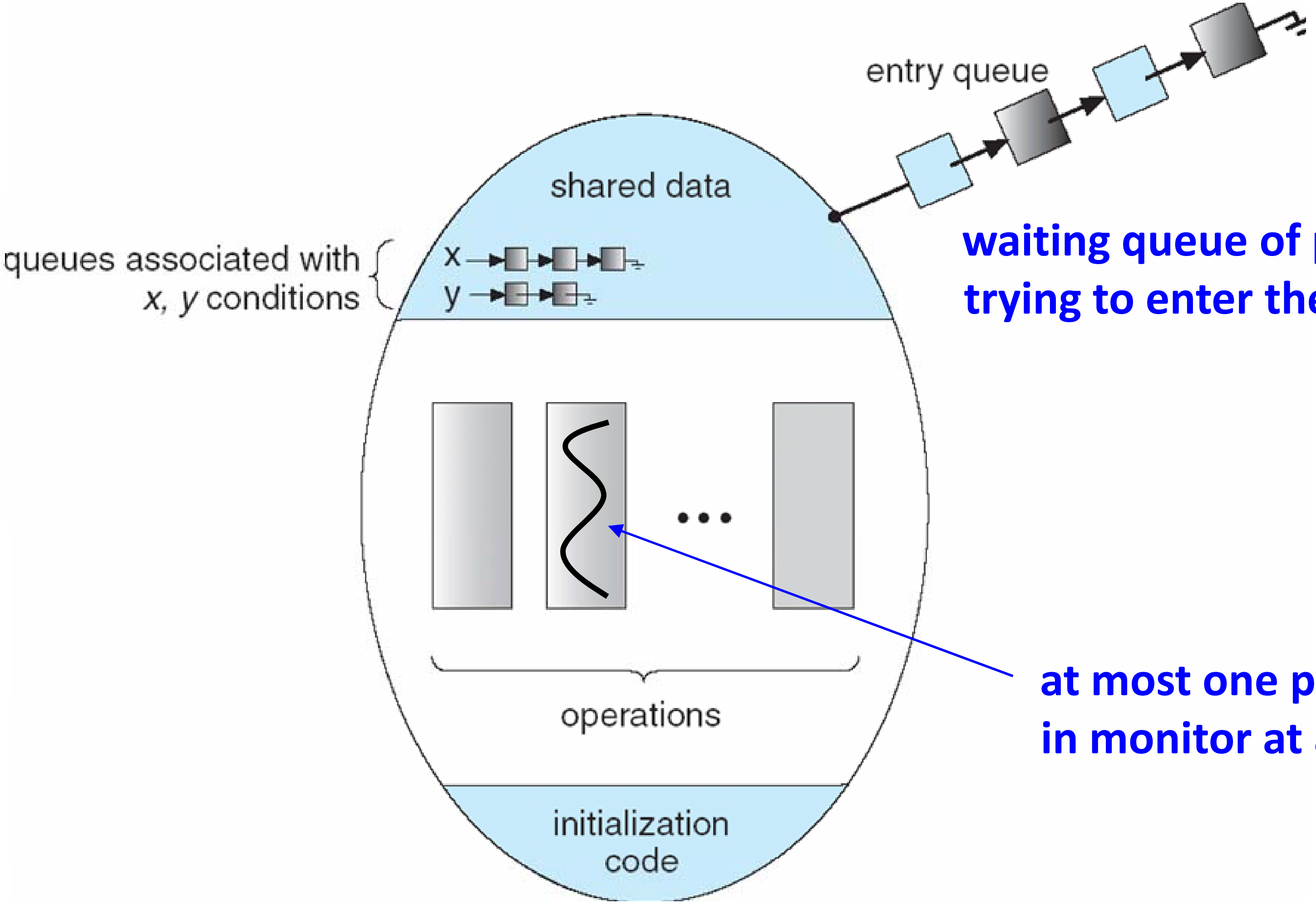
Monitors (2)



- **Mutual exclusion**
  - Only one process can be executing inside at any time.
    - Thus, synchronization implicitly associated with monitor
  - If a second process tries to enter a monitor procedure, it blocks until the first has left the monitor.
    - More restrictive than semaphores.
    - But easier to use most of the time.

- **Condition variables**
  - Once inside, a process may discover it can't continue, and may wish to sleep, or allow some other waiting process to continue.
  - Condition variables are provided within monitor.
    - Processes can wait or signal others to continue.
    - Can only be accessed from inside monitor.

Monitors (3)



waiting queue of processes trying to enter the monitor

at most one process in monitor at a time

Condition Variables

- **Purpose**
  - provides a mechanism to wait for events.
    (a “rendezvous point”)
- **Three operations:**
  - wait (c)
    - release monitor lock, so somebody else can get in.
    - wait for somebody else to signal condition.
    - thus, condition variables have wait queues.
  - signal (c)
    - wake up at most one waiting process.
    - if no waiting processes, signal is lost.
    - this is different from semaphores: no history!
  - broadcast (c)
    - wake up all waiting processes.

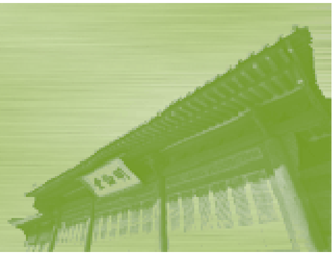
Bounded Buffer using Monitors

```
Monitor bounded_buffer {
   buffer resources[N];
   condition not_full, not_empty;

   procedure add_entry (resource x)  {
       while (array “resources” is full)
           wait (not_full);
       add “x” to array “resources”;
       signal (not_empty);
   }

   procedure remove_entry (resource *x)  {
       while (array “resources” is empty)
           wait (not_empty);
       *x = get resources from array “resources”
       signal (not_full);
   }
}
```

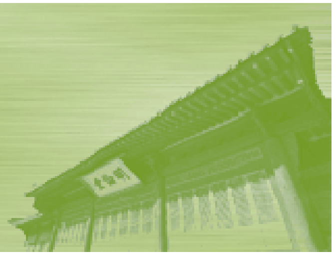
Monitors Semantics (1)



- **Hoare monitors:**
  - signal(c) immediately switches from the caller to a waiting thread, blocking the caller.
    - The condition that the waiter was anticipating is guaranteed to hold when waiter executes.
    - Signaler must restore monitor invariants before signaling.

- **Mesa monitors:**
  - signal(c) places a waiter on the ready queue, but signaler continues inside monitor.
    - Condition is not necessarily true when waiter runs again.
    - Being woken up is only a hint that something has changed.
    - Must recheck conditional case.

Monitors Semantics (2)



Comparison

- Usage:

Hoare monitors

```
if (notReady)
      wait (c);
```

Mesa monitors

```
while (notReady)
      wait (c);
```

- Mesa monitors easier to use.
  - more efficient
  - fewer switches
  - directly supports broadcast()
- Hoare monitors leave less to chance.
  - when wake up, condition guaranteed to be what you expect.

Monitors using Semaphores

- Hoare monitors

```
Semaphore mutex = 1;
Semaphore next = 0;
int next_count = 0;
struct condition {
   Semaphore sem;
   int count;
} x = {0, 0};

procedure F ()  {
   wait (mutex);
   …
   Body of F
   …
   if (next_count)
       signal (next);
   else
       signal (mutex);
}
```

```
procedure cond_wait (x)  {
   x.count++;
   if (next_count)
       signal (next);
   else
       signal (mutex);
   wait (x.sem);
   x.count--;
}

procedure cond_signal (x)  {
   if (x.count)  {
       next_count++;
       signal (x.sem);
       wait (next);
       next_count--;
   }
}
```

Monitors and Semaphores

- **Comparison**
  - Condition variables do not have any history, but semaphores do.
    - On a condition variable signal(), if no one is waiting , the signal is a no-op.

      (If a thread then does a condition variable wait(), it waits.)
    - On a semaphore signal(), if no one is waiting, the value of the semaphore is increased.

      (If a thread then does a semaphore wait(), the value is decreased and the thread continues.)

Condition Variables and Mutex

- **Yet another construct:**
  - Condition variables can be also used without monitors in conjunction with mutexes.
  - Think of a monitor as a language feature
    - Under the covers, compiler knows about monitors.
    - Compiler inserts a mutex to control entry and exit of processes to the monitor's procedures.
    - But can be done anywhere in procedure, at finer granularity.
  - With condition variables, the module methods may wait and signal on independent conditions.

Synchronization in Pthreads

```
pthread_mutex_t mutex;
pthread_cond_t not_full, not_empty;
buffer resources[N];
void add_entry (resource x)  {
    pthread_mutex_lock (&mutex);
    while (array “resources” is full)
        pthread_cond_wait (&not_full, &mutex);
    add “x” to array “resources”;
    pthread_cond_signal (&not_empty);
    pthread_mutex_unlock (&mutex);
}
void remove_entry (resource *x)  {
    pthread_mutex_lock (&mutex);
    while (array “resources” is empty)
        pthread_cond_wait (&not_empty, &mutex);
    *x = get resource from array “resources”
    pthread_cond_signal (&not_full);
    pthread_mutex_unlock (&mutex);
}
```

Synchronization Mechanisms

- **Disabling interrupts**
- **Spinlocks**
  - Busy waiting
- **Semaphores**
  - Binary semaphore = mutex (≅ lock)
  - Counting semaphore
- **Monitors**
  - Language construct with condition variables
- **Mutex + Condition variables**
  - Pthreads