COS 318: Operating Systems

Semaphores, Monitors and Condition Variables

Jaswinder Pal Singh
Computer Science Department
Princeton University

(http://www.cs.princeton.edu/courses/cos318/)

Today's Topics

- Semaphores
- Monitors
- Barriers

Revisit Mutex

- Mutex can solve the critical section problem

  Acquire(lock);

  Critical section

  Release(lock);

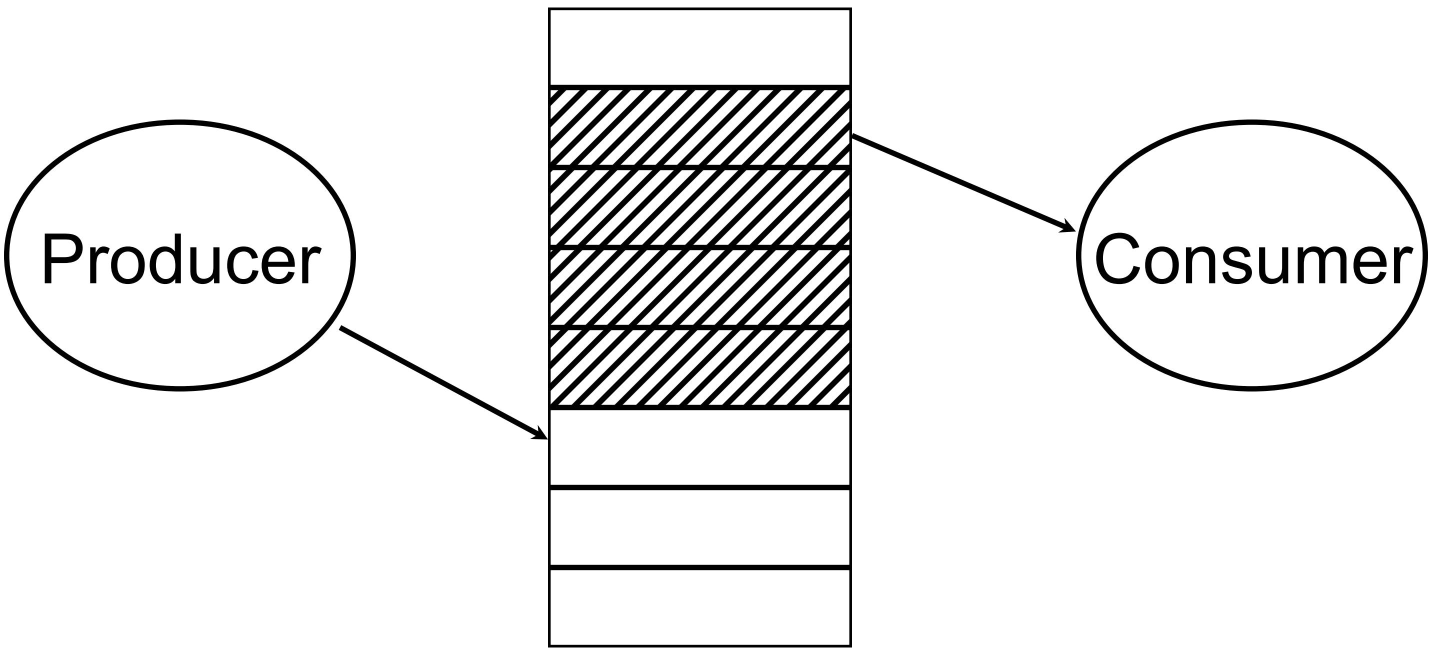
- Always use Mutex primitives when you access shared data structures

  E.g. shared "count" variable

  ```
  Acquire( lock );
  count++;
  Release( lock );
  ```

- Are mutex primitives adequate to solve all problems?

Bounded Buffer Problem

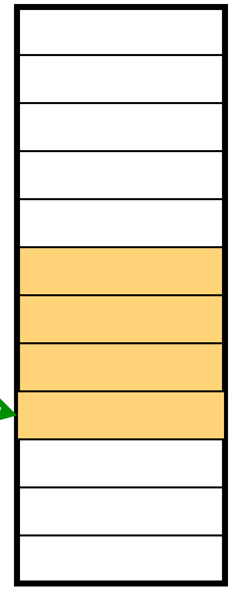


Producer-Consumer (Bounded Buffer) Problem

```
Producer:
  while (1) {
    produce an item

    Insert item in buffer

    count++;
}
```



```
Consumer:
  while (1) {
    remove an item from buffer

    count--;

    consume an item
  }
```

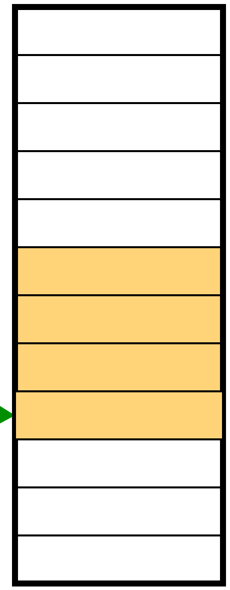
count = 4

N = 12

- Can we solve this problem with Mutex primitives?

Use Mutex, Block and Unblock

```
Producer:
  while (1) {
    produce an item
    if (count == N)
      Block();
    Insert item in buffer
    Acquire(lock);
    count++;
    Release(lock);
    if (count == 1)
      Unblock(Consumer);
  }
```



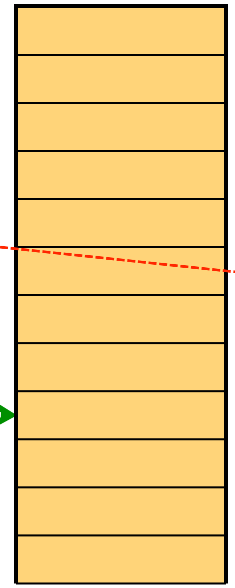
count = 4

N = 12

```
Consumer:
  while (1) {
    if (!count)
      Block();
    remove an item from buffer
    Acquire(lock);
    count--;
    Release(lock);
    if (count == N-1)
      Unblock(Producer);
    consume an item
  }
```

Use Mutex, Block and Unblock

```
Producer:
  while (1) {
    produce an item
    if (count == N)
      Block();
    Insert item in buffer
    Acquire(lock);
    count++;
    Release(lock);
    if (count == 1)
      Unblock(Consumer);
  }
```



count = 12

N = 12

```
Consumer:
  while (1) {
    if (!count)
{context switch}
      Block();
    remove an item from buffer
    Acquire(lock);
    count--;
    Release(lock);
    if (count == N-1)
      Unblock(Producer);
    consume an item
  }
```

- Race condition!
- Any way to make this work?
- These primitives are not enough

Semaphores (Dijkstra, 1965)

- Keep count of number of wakeups saved
- Initialization: initialize value atomically
- P (or Down or Wait) definition
  - Atomic operation
  - If semaphore not positive, block
  - Or wait for semaphore to become positive and then decrement

```
P(s){
  while (s <= 0)
    ;
  s--;
}
```

```
P(s){
    if (--s < 0)
        block(s);
}
```

- V (or Up or Signal) definition
  - Atomic operation
  - Increment semaphore by 1

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

```
V(s){
    if (++s <= 0)
        unblock(s);
}
```

Bounded Buffer with Semaphores

```
Producer:
  while (1) {
    produce an item
    P(emptyCount);

    P(mutex);
    put item in buffer
    V(mutex);

    V(fullCount);
  }
```

```
Consumer:
  while (1) {
    P(fullCount);

    P(mutex);
    take an item from buffer
    V(mutex);

    V(emptyCount);
    consume item
  }
```

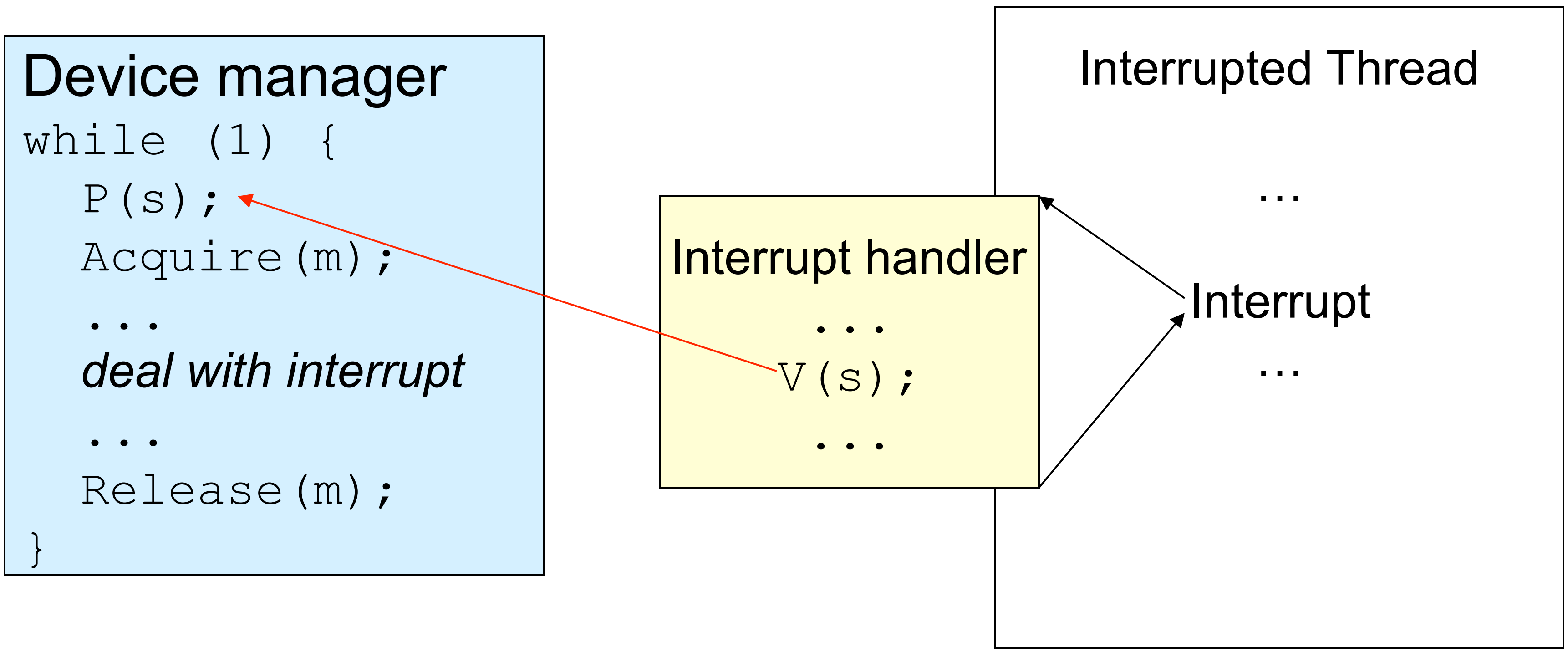
- Initialization: emptyCount = N; fullCount = 0
- Are **`P(mutex)`** and **`V(mutex)`** necessary?

Uses of Semaphores in this Example

- Event sequencing
  - Don't consume if buffer empty, wait for something to be added
  - Don't add if buffer full, wait for something to be removed
- Mutual exclusion
  - Avoid race conditions on shared variables

Use Semaphores for Interrupt Handling

```
Init(s,0);
```



Bounded Buffer with Semaphores (again)

```
producer() {
  while (1) {
    produce an item
    P(emptyCount);

    P(mutex);
    put the item in buffer
    V(mutex);

    V(fullCount);
  }
}
```

```
consumer() {
  while (1) {
    P(fullCount);

    P(mutex);
    take an item from buffer
    V(mutex);

    V(emptyCount);
    consume the item
  }
}
```

Does Order Matter?

```
producer() {
  while (1) {
    produce an item
    P(mutex);
    P(emptyCount);

    put the item in buffer
    V(mutex);

    V(fullCount);
  }
}
```

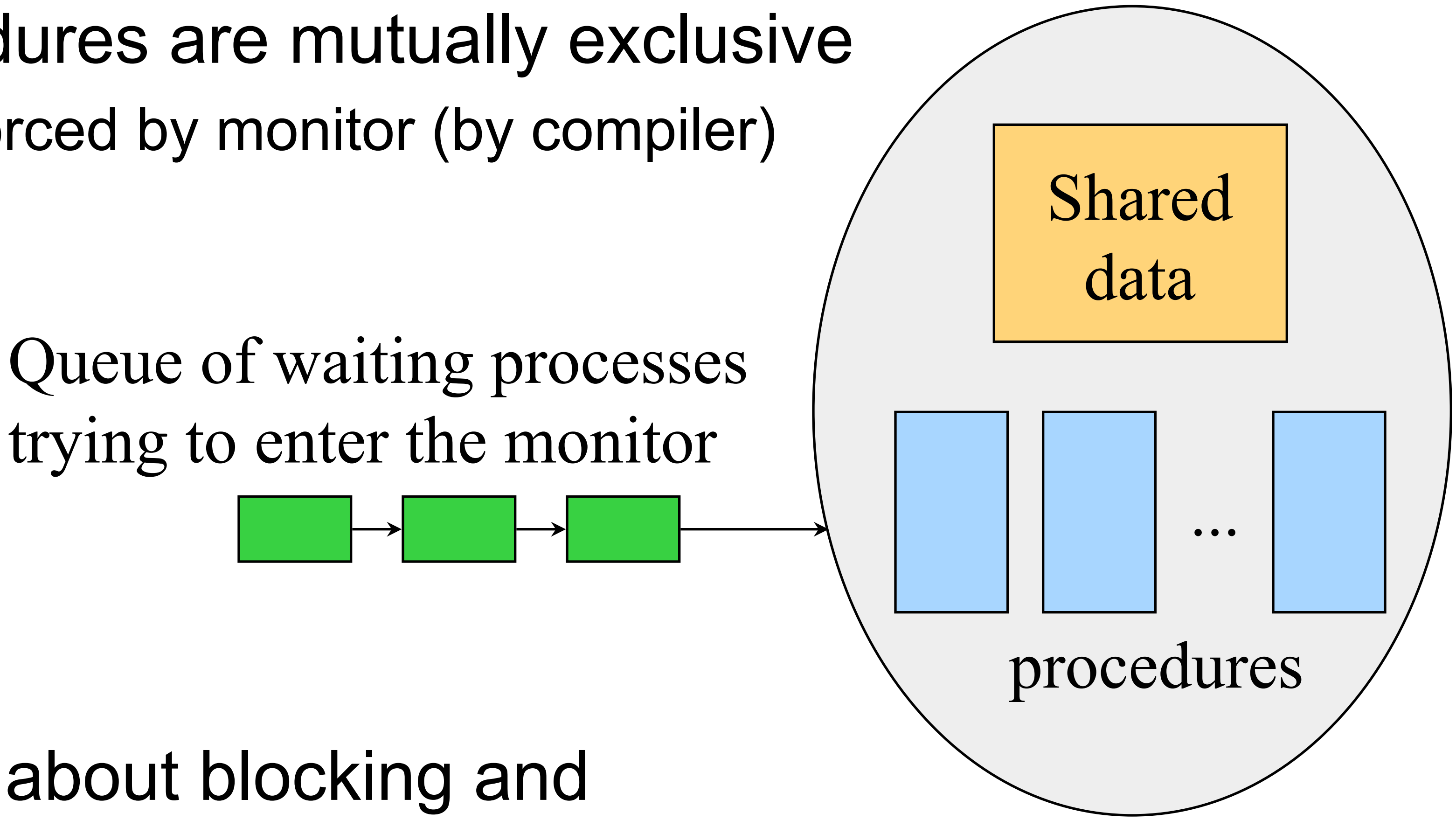
```
consumer() {
  while (1) {
    P(fullCount);

    P(mutex);
    take an item from buffer
    V(mutex);

    V(emptyCount);
    consume the item
  }
}
```

Monitor: Hide Mutual Exclusion

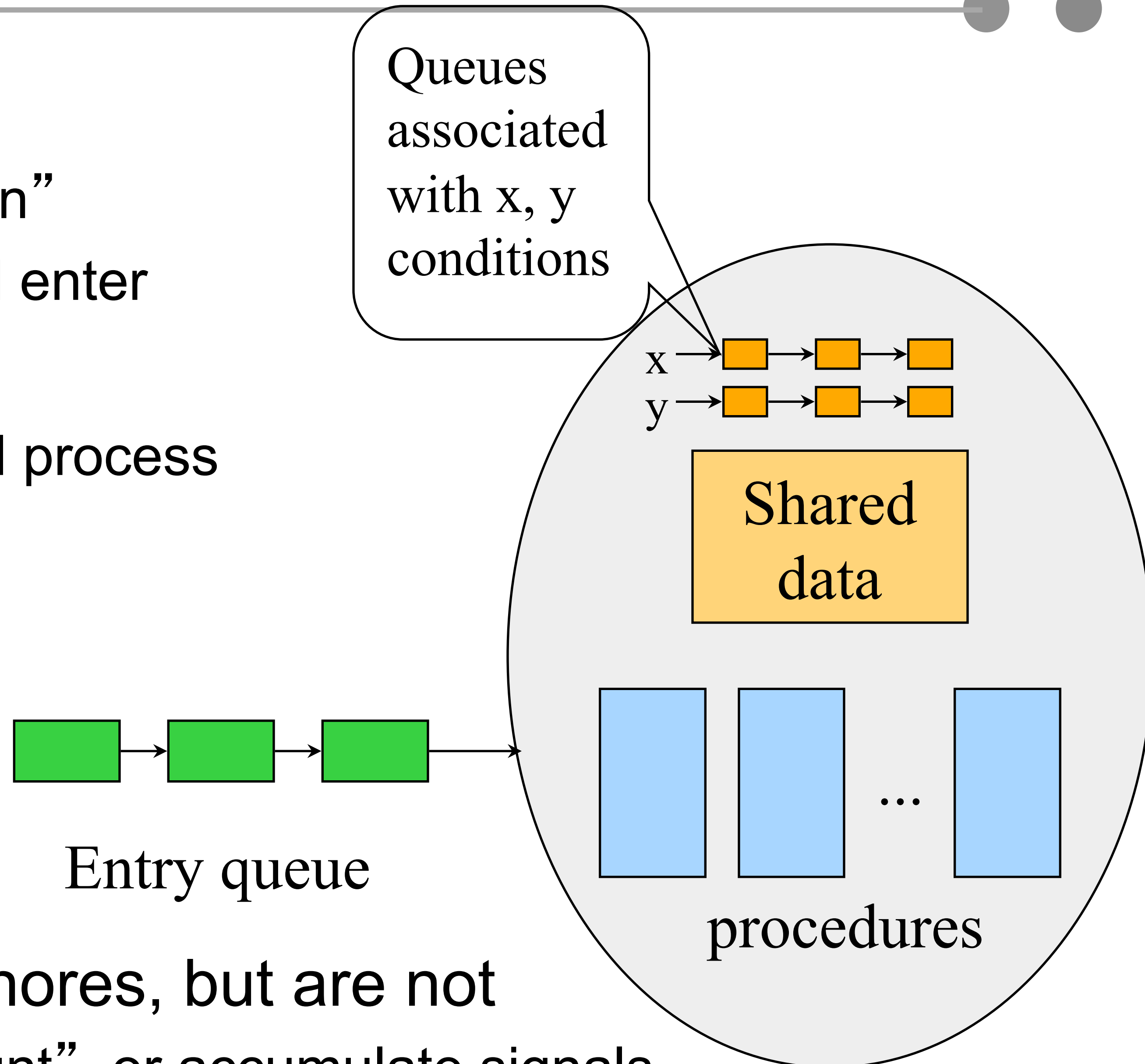
- Brinch-Hansen (73), Hoare (74)
- Procedures are mutually exclusive
  - Enforced by monitor (by compiler)



- What about blocking and sequencing?

Condition Variables in A Monitor

- Wait(condition)
  - Block on "condition"
  - Let another thread enter
- Signal(condition)
  - Wakeup a blocked process on "condition"



- Look like semaphores, but are not
  - They don't "count", or accumulate signals
- Like sleep/wakeup, but with mutual exclusion at monitor level

Producer-Consumer with Monitors

```
procedure Producer
begin
  while true do
  begin
    produce an item
    ProdCons.Enter();
  end;
end;

procedure Consumer
begin
  while true do
  begin
    ProdCons.Remove();
    consume an item;
  end;
end;
```

```
monitor ProdCons
  condition full, empty;

  procedure Enter;
  begin
    if (buffer is full)
      wait(full);
    put item into buffer;
    if (only one item)
      signal(empty);
  end;

  procedure Remove;
  begin
    if (buffer is empty)
      wait(empty);
    remove an item;
    if (buffer was full)
      signal(full);
  end;
```

What happens after a signal?

- Run the signaled thread immediately and suspend the current one (Hoare)
  - If the signaler has other work to do, life is complex
  - It is difficult to make sure there is nothing to do, because the signal implementation is not aware of how it is used
  - It is easy to prove things
- Current thread exits the monitor (Hansen)
  - Signal must be the last statement of a monitor procedure
- Current thread continues its execution (Mesa)
  - Easy to implement
  - But, the condition may not be true when the awakened process actually gets a chance to run

More on Mesa-Style Monitor

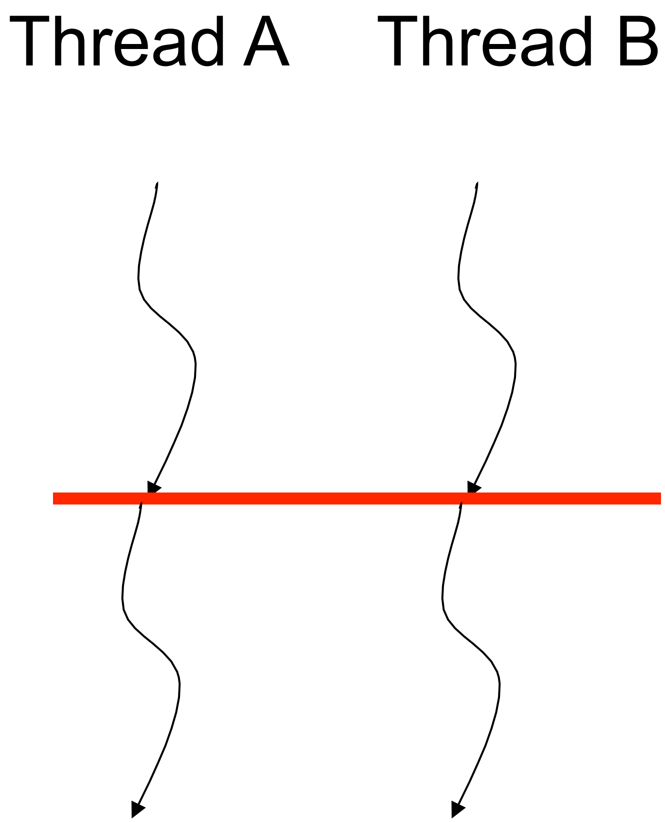
- Signaler continues execution
- Waiters simply put on ready queue, with no special priority
  - Must reevaluate the condition
- No constraints on when the waiting thread/process must run after a "signal"
- Simple to introduce a broadcast: wake up all
- No constraints on signaler
  - Can execute after signal call (Hansen's cannot)
  - Does not need to relinquish control to awaken thread/process

Evolution of Monitors

- Brinch-Hansen (73) and Hoare Monitor (74)
  - Concept, but no implementation
  - Requires Signal to be the last statement (Hansen)
  - Requires relinquishing CPU to signaler (Hoare)
- Mesa Language (77)
  - Monitor in language, but signaler keeps mutex and CPU
  - Waiter simply put on ready queue, with no special priority
- Modula-2+ (84) and Modula-3 (88)
  - Explicit LOCK primitive
  - Mesa-style monitor
- Pthreads (95)
  - Started standard effort around 1989
  - Defined by ANSI/IEEE POSIX 1003.1 Runtime library
- Java threads
  - James Gosling in early 1990s without threads
  - Use most of the Pthreads primitives

Example: A Simple Barrier

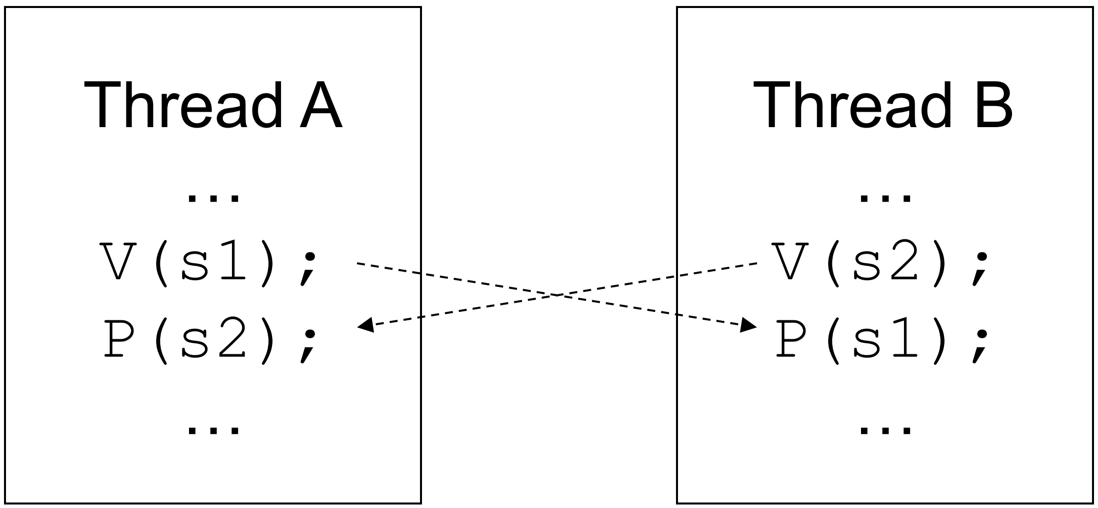
- Thread A and Thread B want to meet at a particular point and then go on
- How would you program this with a monitor?



Using Semaphores as A Barrier

- Use two semaphores?

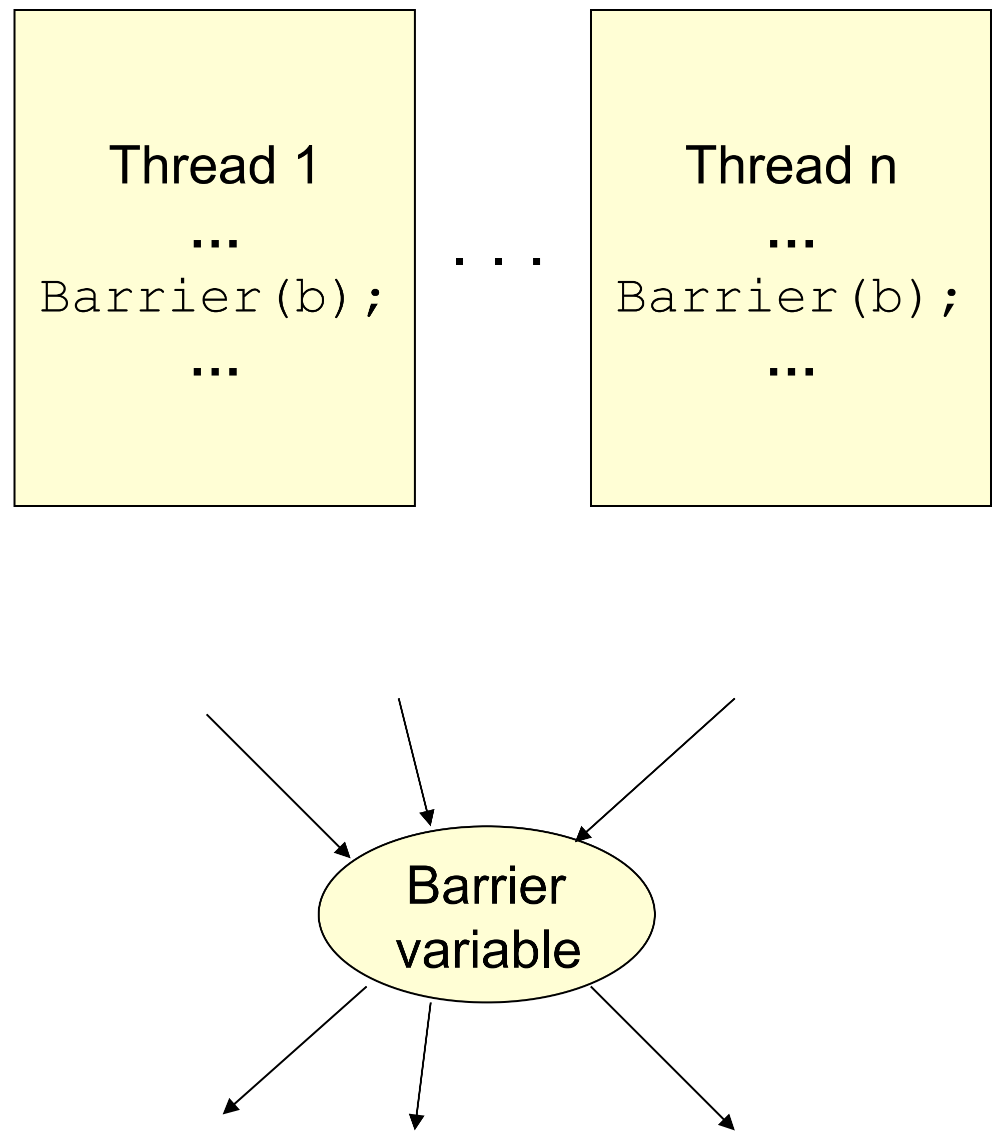
```
init(s1, 0);
init(s2, 0);
```



- What about more than two threads?

Barrier Primitive

- Functions
  - Take a barrier variable
  - Broadcast to n-1 threads
  - When barrier variable has reached n, go forward
- Hardware support on some parallel machines
  - Multicast network
  - Counting logic
  - User-level barrier variables



Equivalence

- Semaphores
  - Good for signaling
  - Not good for mutex because it is easy to introduce a bug
- Monitors
  - Good for scheduling and mutex
  - Maybe costly for a simple signaling

Summary

- Semaphores
- Monitors
- Barriers