Lecture 7: Semaphores and monitors

601.418/618 Operating Systems

David Hovemeyer

February 11, 2026

Agenda

- Semaphores
- Monitors
- Condition variables

Acknowledgments: These slides are shamelessly adapted from Prof. Ryan Huang's Fall 2022 slides, which in turn are based on Prof. David Mazières's OS lecture notes.

Higher-Level Synchronization

Last time: We looked at using locks to provide mutual exclusion

Locks work, but they have limited semantics

- Just provide mutual exclusion

Instead, we want synchronization mechanisms that

- Block waiters, leave interrupts enabled in critical sections
- Provide semantics beyond mutual exclusion

Look at two common high-level mechanisms

- *Semaphores*: binary (mutex) and counting
- *Monitors*: mutexes and condition variables

Semaphores

An *abstract data type* to provide synchronization

- Described by Dijkstra in the "THE" system in 1968

Semaphores are "integers" that support two operations:

- `Semaphore::P()` decrements, blocks until semaphore is open, a.k.a `wait()`
  - after the Dutch word "Proberen" (to try)
  - Pintos `sema_down()`, pthreads `sem_wait()`
- `Semaphore::V()` increments, allows another thread to enter, a.k.a `signal()`
  - after the Dutch word "Verhogen" (increment)
  - Pintos `sema_up()`, pthreads `sem_post()`
- **That's it!** No other operations – not even just reading its value

Semaphore safety property: the semaphore value is always greater than or equal to 0

Blocking in Semaphores

Associated with each semaphore is a queue of waiting threads

When `P()` is called by a thread:

- If semaphore is *open*, thread continues
- If semaphore is *closed*, thread blocks on queue

Then `V()` opens the semaphore:

- If a thread is waiting on the queue, the thread is unblocked
  - If multiple threads are waiting, one is chosen to wake up
- *If no threads are waiting on the queue, the signal is remembered for the next thread*
  - In other words, `V()` has "history" (c.f., condition variables)
  - This "history" is a counter

Semaphore Types

Semaphores come in two types

Mutex semaphore (or *binary* semaphore)

- Represents single access to a resource
- Guarantees mutual exclusion to a critical section

Counting semaphore (or *general* semaphore)

- Represents a resource with many units available, or a resource that allows certain kinds of unsynchronized concurrent access (e.g., reading)
- Multiple threads can pass the semaphore
- Number of threads determined by the semaphore "count"
  - Mutex has initial count $= 1$, counting has count $= N$

Using Semaphores

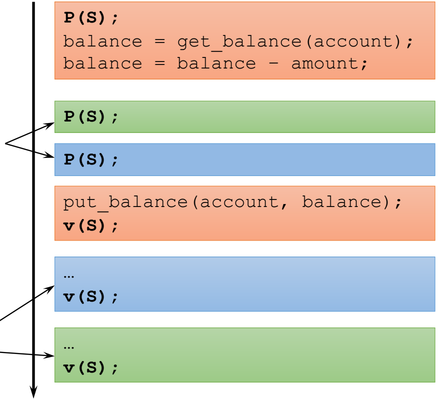
Use is similar to locks (from last time), but semantics are different

```
struct Semaphore {
    int value;
    Queue q;
} S;
withdraw (account, amount) {
    P(S);
    balance = get_balance(account);
    balance = balance - amount;
    put_balance(account, balance);
    v(S);
    return balance;
}
```

Threads block

critical section

It is undefined which thread runs after a signal



Semaphore Questions

Are there any problems that can be solved with counting semaphores that cannot be solved with mutex semaphores?

- If a system only gives you mutex semaphore, can you use it to implement counting semaphores?

Does it matter which thread is unblocked by a signal operation?

Implementing counting semaphores using mutex semaphores

```
// constructor
CSem(K) {                    // counting semaphore initialized to K
  int val := K;              // the value of csem
  BSem gate(min(1,val)); // 1 if val > 0; 0 if val = 0
  BSem mutex(1);             // protects val
}
```

```
// P() operation
CSem::P() {
  P(gate)
  P(mutex);
  val := val - 1;
  if val > 0
    V(gate);
  V(mutex);
}
```

```
// V() operation
CSem::V() {
  P(mutex);
  val := val + 1;
  if val = 1
    V(gate);
  V(mutex);
}
```

See: http://www.cs.umd.edu/~shankar/412-Notes/10x-countingSemUsingBinarySem.pdf

Semaphore Implementation in Pintos

```c
void sema_down(struct semaphore *sema)
{
  enum intr_level old_level;
  old_level = intr_disable();
  while (sema->value == 0) {
    list_push_back(&sema->waiters,
          &thread_current()->elem);
    thread_block();
  }
  sema->value--;
  intr_set_level(old_level);
}
```

```c
void sema_up(struct semaphore *sema)
{
  enum intr_level old_level;
  old_level = intr_disable();
  if (!list_empty (&sema->waiters))
    thread_unblock(list_entry(
          list_pop_front(&sema->waiters),
               struct thread, elem));
  sema->value++;
  intr_set_level(old_level);
}
```

To reference current thread: `thread_current()`

`thread_block()` puts the current thread to sleep

Assignment 1 note:

- leverage semaphore instead of directly using `thread_block()`/`thread_unblock()`

Implementation of thread_block()

pick another thread to run

```
/* Puts the current thread to sleep. This function
must be called with interrupts turned off.*/
void thread_block ()
{
  ASSERT (!intr_context ());
  ASSERT (intr_get_level () == INTR_OFF);
  thread_current ()->status = THREAD_BLOCKED;
  schedule ();
}
```

thread_block() assumes the interrupts are disabled

This means we will have the thread sleep with interrupts disabled

Isn't this bad?

- Shouldn't we only disable interrupts when entering/leaving critical sections but keep interrupts enabled during critical section?

Interrupts Re-enabled Right After Context Switch

```
thread_yield() {
  Disable interrupts;
  add current thread to ready_list;
  schedule(); // context switch
  Enable interrupts;
}
```

```
sema_down(){
 Disable interrupts;
 while(value == 0) {
   add current thread to waiters;
   thread_block();
 }
 value--;
 Enable interrupts;
}
```

Thread 1

```
[thread_yield]
Disable interrupts;
add current thread to ready_list;
schedule();
```

Thread 2

```
[thread_yield]
(Returns from schedule())
Enable interrupts;
```

...

Thread 2

```
[sema_down]
Disable interrupts;
while(value == 0) {
  add current thread to waiters;
  thread_block();
}
```

Thread 1

```
[thread_yield]
(Returns from schedule())
Enable interrupts;
```

Semaphore Summary

Semaphores can be used to solve any traditional synchronization problem

However, they have some drawbacks

- They are essentially shared global variables
  - Can potentially be accessed anywhere in program
- No connection between the semaphore and the data controlled by the semaphore
- Used both for critical sections (mutual exclusion) and coordination (scheduling)
  - Note that I had to use comments in the code to distinguish
- No control or guarantee of proper usage

Sometimes hard to use and prone to bugs

- Another approach: **Use programming language support**

Monitors

A programming language construct that controls access to shared data

- Synchronization code added by compiler, enforced at runtime
- **Why is this an advantage?**

A monitor is a module that encapsulates

- *Shared data structures*
- *Procedures* that operate on the shared data structures
- *Synchronization* between concurrent threads that invoke the procedures

A monitor protects its data from unstructured access

It guarantees that threads accessing its data through its procedures interact only in legitimate ways

Monitor Semantics

A monitor guarantees mutual exclusion

- Only one thread can execute any monitor procedure at any time
  - The thread is “in the monitor”
- If a second thread invokes a monitor procedure when a first thread is already executing one, it blocks
  - So the monitor has to have a wait queue...
- If a thread within a monitor blocks, another one can enter

What are the implications in terms of parallelism in a monitor?

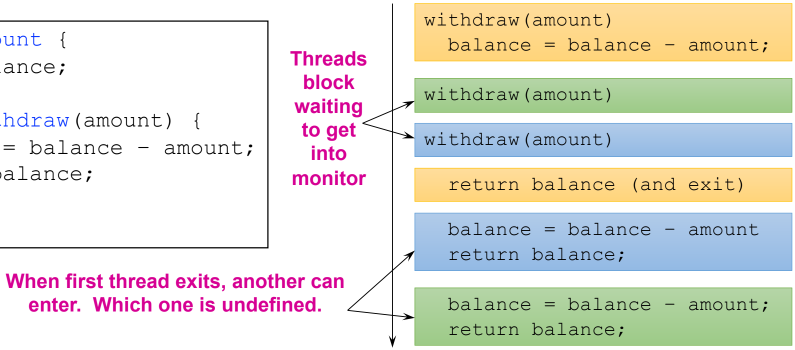
A *monitor invariant* is a *safety property* associated with the monitor

- It’s expressed over the monitored variables.
- It holds whenever a thread enters or exits the monitor.

Account Example

```
Monitor account {
  double balance;

  double withdraw(amount) {
    balance = balance - amount;
    return balance;
  }
}
```



Hey, that was easy!

Monitor invariant: $balance \geq 0$

Condition Variables

But what if a thread wants to wait for something inside the monitor?

- If we busy wait, it's bad
- Even worse, no one can get in the monitor to make changes now!

A *condition variable* is associated with a *condition* needed for a thread to make progress once it is in the monitor.

```
Monitor M {
  ... monitored variables
  Condition c;

  void enterMonitor (...) {
    if (extra property not true) wait(c);  waits outside of the monitor's mutex
    do what you have to do
    if (extra property true) signal(c);   brings in one thread waiting on condition
  }
```

Condition Variables

Condition variables support three operations:

- *Wait*: **release monitor lock**, wait for condition variable to be signaled
  - So condition variables have wait queues, too
- *Signal*: wake up one waiting thread
- *Broadcast*: wake up all waiting threads

Condition variables are not boolean objects

- ✗ `if (condition_variable) then ...` does not make sense
- ✓ `if (num_resources == 0) then wait(resources_available)` does
- An example later will make this more clear

Condition Vars $\neq$ Semaphores

Condition variables $\neq$ semaphores

- Although their operations have similar names, they have entirely different semantics (such is life, worse yet to come)
- **However, they each can be used to implement the other**

Access to the monitor is controlled by a lock

- `wait()` blocks the calling thread, and **gives up the lock**
  - To call `wait`, the thread has to be in the monitor (hence has lock)
  - `Semaphore::wait` just blocks the thread on the queue
- `signal()` causes a waiting thread to wake up
  - **If there is no waiting thread, the signal is lost**
  - `Semaphore::signal` increases the semaphore count, allowing future entry even if no thread is waiting
- Condition variables have no history

Signal Semantics

Two flavors of monitors that differ in the scheduling semantics of `signal()`

- Hoare monitors (original)
  - `signal()` **immediately switches from the caller to a waiting thread**
  - The condition that the waiter was anticipating is guaranteed to hold when waiter executes
  - Signaler must restore monitor invariants before signaling
- Mesa monitors (Mesa, Java)
  - `signal()` places a waiter on the ready queue, but **signaler continues inside monitor**
  - Condition is not necessarily true when waiter runs again
  - Returning from `wait()` is only a *hint* that something changed
  - Must recheck conditional case

Hoare vs. Mesa Monitors

Hoare monitor semantics:

```
if (!condition)
  wait(cond_var); // <-- condition definitely holds when wait() returns
```

Mesa/Java monitor semantics:

```
while (!condition)
  wait(cond_var); // <-- condition *might* hold when wait() returns
// <-- condition definitely holds when loop finishes
```

Tradeoffs:

- Mesa monitors easier to use, more efficient
  - Fewer context switches, easy to support broadcast
- Hoare monitors leave less to chance
  - Easier to reason about the program

Condition variables and locks

Condition variables are also used without monitors in conjunction with locks (e.g., pthreads)

```
void cond_init (cond_t *, ...);
```

```
void cond_wait (cond_t *c, mutex_t *m);
```

- Atomically unlock `m` and sleep until `c` signaled
- Then re-acquire `m` and resume executing

```
void cond_signal (cond_t *c);
void cond_broadcast (cond_t *c);
```

- Wake one/all threads waiting on `c`

Condition variables and locks

A monitor $\approx$ a module whose state includes condition variable(s) and a lock

- Difference is syntactic; with monitors, compiler adds the code

It is "just as if" each procedure in the module calls acquire() on entry and release() on exit

- But can be done anywhere in procedure, at finer granularity

With condition variables, the module methods may wait and signal on independent conditions

Condition variables and locks

Why must `cond_wait` both release `mutex_t` and block the caller?

- `void cond_wait(cond_t *c, mutex_t *m);`

Why not separate mutexes and condition variables?

```
while (count == BUFFER_SIZE) {
    mutex_unlock(&mutex);
    cond_wait(&not_full);
    mutex_lock(&mutex);
}
```

Condition variables and locks

Why must `cond_wait` both release `mutex_t` and block the caller?

- `void cond_wait(cond_t *c, mutex_t *m);`

Why not separate mutexes and condition variables?

Producer

```
while (count == BUFFER_SIZE) {
    mutex_unlock(&mutex);



    cond_wait(&not_full);
    mutex_lock(&mutex);
}
```

Consumer

```
mutex_lock(&mutex);
... count--;
cond_signal(&not_full);
mutex_unlock(&mutex);
```

Using condition variables and locks

Alternation of two threads (ping-pong)

Each executes the following:

```
Lock lock;
Condition cond;

void ping_pong () {
  acquire(lock);
  while (1) {
      printf("ping or pong\n");
      signal(cond);
      wait(cond, lock);
  }
  release(lock);
}
```

Must acquire lock before you can wait (similar to needing interrupts disabled to call `thread_block` in Pintos)

Wait atomically releases lock and blocks until signal()

After signal(), wait re-acquires lock before returning

Monitors and Java

A lock and condition variable are in every Java object

- No explicit classes for locks or condition variables

Every object is/has a monitor

- At most one thread can be inside an object's monitor
- A thread enters an object's monitor by
  - Executing a method declared "`synchronized`"
  - Executing the body of a "`synchronized`" statement
- The compiler generates code to acquire the object's lock at the start of the method and release it just before returning
  - The lock itself is implicit, programmers do not worry about it

Monitors and Java

Every object can be treated as a condition variable

- Half of `Object`'s methods are for synchronization!

Take a look at the Java Object class:

- `Object.wait(*)` is `Condition::wait()`
- `Object.notify()` is `Condition::signal()`
- `Object.notifyAll()` is `Condition::broadcast()`

Summary

Semaphores

- `wait()`/`signal()` implement blocking mutual exclusion
- Also used as atomic counters (counting semaphores)
  - Often used to count availability of units of a resource
- Can be inconvenient to use

Monitors

- Synchronizes execution within procedures that manipulate encapsulated data shared among procedures
  - Only one thread can execute within a monitor at a time
- Relies upon high-level language support

Condition variables

- Used by threads as a synchronization point to wait for events
- Inside monitors, or outside with locks

Next time

Synchronization in practice