Condition Variables and Monitors

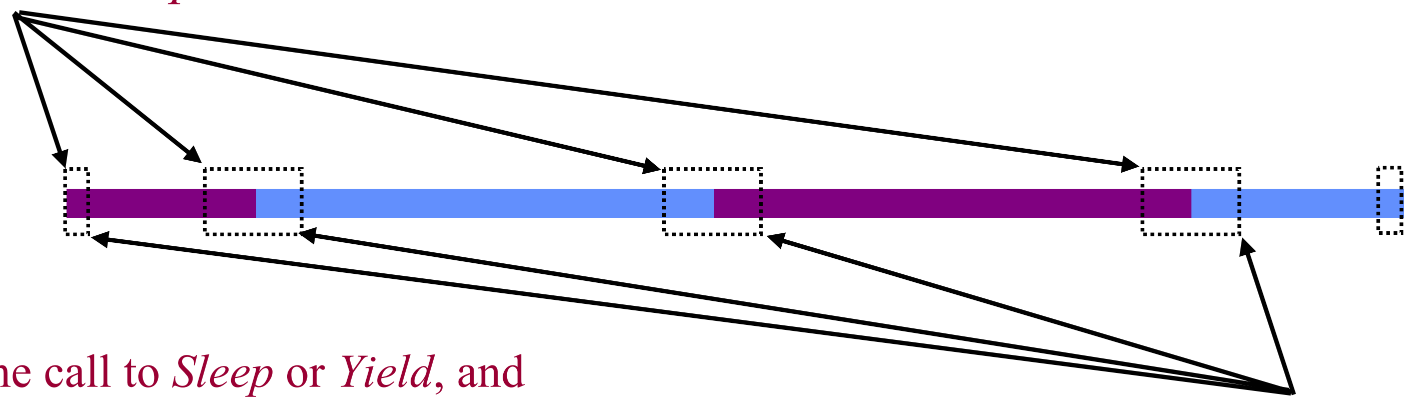
Dr. Daniel Andresen

CIS520

Digression: Sleep and Yield in Nachos

disable interrupts

Context switch itself is a critical section, which we enter only via *Sleep* or *Yield*.



Disable interrupts on the call to *Sleep* or *Yield*, and rely on the "other side" to re-enable on return from its own *Sleep* or *Yield*.

enable interrupts

```
Yield() {
    IntStatus old = SetLevel(IntOff);
    next = scheduler->FindNextToRun();
    if (next != NULL) {
        scheduler->ReadyToRun(this);
        scheduler->Run(next);
    }
    interrupt->SetLevel(old);
}
```

```
Sleep() {
    ASSERT(getLevel = IntOff);
    this->status = BLOCKED;
    next = scheduler->FindNextToRun();
    while(next = NULL) {
        /* idle */
        next = scheduler->FindNextToRun();
    }
    scheduler->Run(next);
}
```

Condition Variables

Condition variables allow *explicit* event notification.

- much like a souped-up *sleep/wakeup*
- associated with a mutex to avoid *sleep/wakeup* races

Condition::Wait(Lock*)

Called with lock held: sleep, atomically releasing lock.
Atomically reacquire lock before returning.

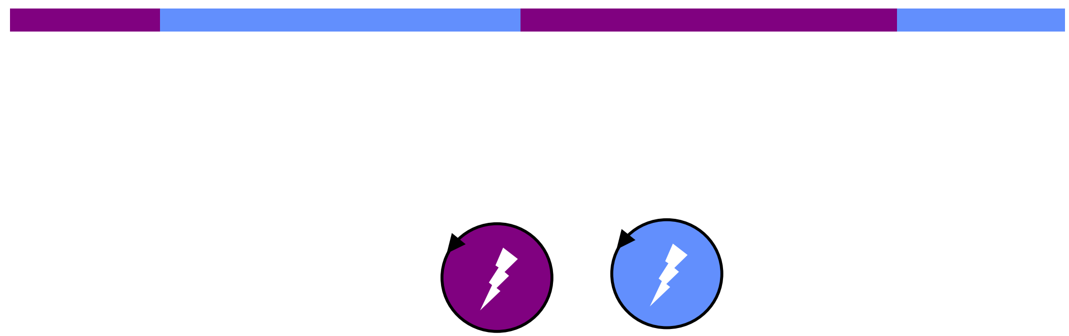
Condition:: Signal(Lock*)

Wake up one waiter, if any.

Condition::Broadcast(Lock*)

Wake up all waiters, if any.

Ping-Pong Using Condition Variables



```
void
PingPong() {
        mx->Acquire();
        while(not done) {
                cv->Signal();
                cv->Wait();
        }
        mx->Release();
}
```

See how the associated mutex avoids sleep/wakeup races?

Will your Project 1 condition variables execute this example correctly?

Using Condition Variables

```
Condition *cv;
Lock* cvMx;
int waiter = 0;

void await() {
        cvMx->Lock();
        waiter =  waiter + 1;   /* "I'm sleeping" */
        cv->Wait(cvMx);         /* sleep */
        cvMx->Unlock();
}

void wakeup() {
        cvMx->Lock();
        if (waiter)
                cv->Signal(cvMx);
        waiter = waiter - 1;
        CvMx->Unlock();
}
```

Must hold lock when calling *Wait*.

Wait atomically releases lock and sleeps until next *Signal*.

Wait atomically reacquires lock before returning.

Association with mutex allows threads to safely manage state related to the sleep/wakeup coordination (e.g., *waiters* count).

Another example: Malloc

```
Lock *l;
Condition *c;

char *malloc(int s) {
    l->Acquire();
    while(cannot allocate
          a chunk of size s) {
      c->Wait(l);
    }
    allocate chunk of size
s;
    l->Release();
    return pointer to
          allocated chunk;
}
```

```
void free(char *m) {
  l->Acquire();
  deallocate m.
  c->Broadcast(l);
  l->Release();
}
```

Question: Why do a Broadcast() rather than a Signal()?

Broadcast() vs. Signal()

- Broadcast() wakes up all threads, Signal() wakes up just one thread, so…
- If **any** thread can take advantage of the new status, and **only one**, then use Signal() for efficiency's sake.
  - Ex: Web server thread pool – OK. All are identical.
  - Ex: Malloc – not OK. More than one thread might be able to progress.
- Otherwise, use Broadcast().

An incorrect CV implementation

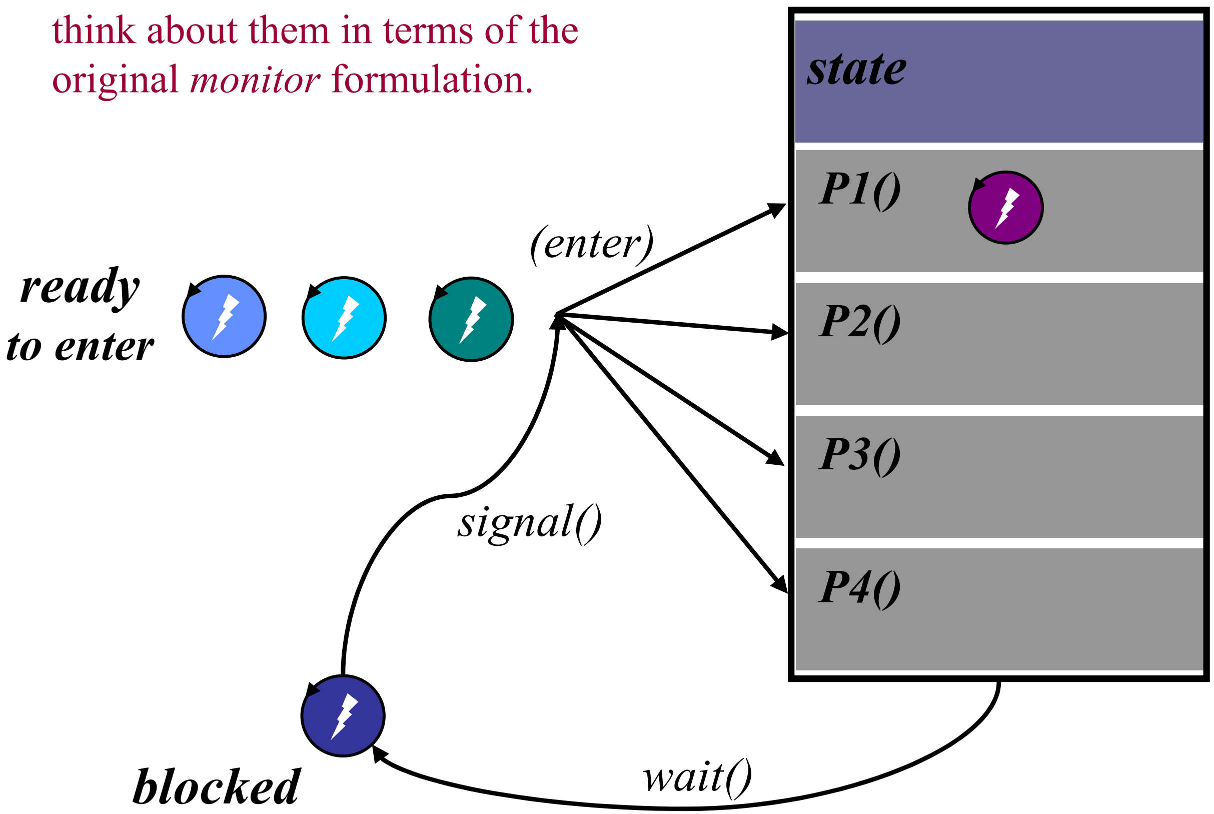
```
class Condition {
    int waiting;
     Semaphore *sema;
}
 void Condition::Wait(Lock* l){
   waiting++;
   l->Release();
   sema->P();
   l->Acquire();
 }
void Condition::Signal(Lock* l){
   if (waiting > 0) {
      sema->V();
     waiting--;
  }
}
```

The Roots of Condition Variables: Monitors

A *monitor* is a module (a collection of procedures) in which execution is serialized.

[Brinch Hansen 1973, C.A.R. Hoare 1974]

CVs are easier to understand if we think about them in terms of the original *monitor* formulation.



At most one thread may be active in the monitor at a time.

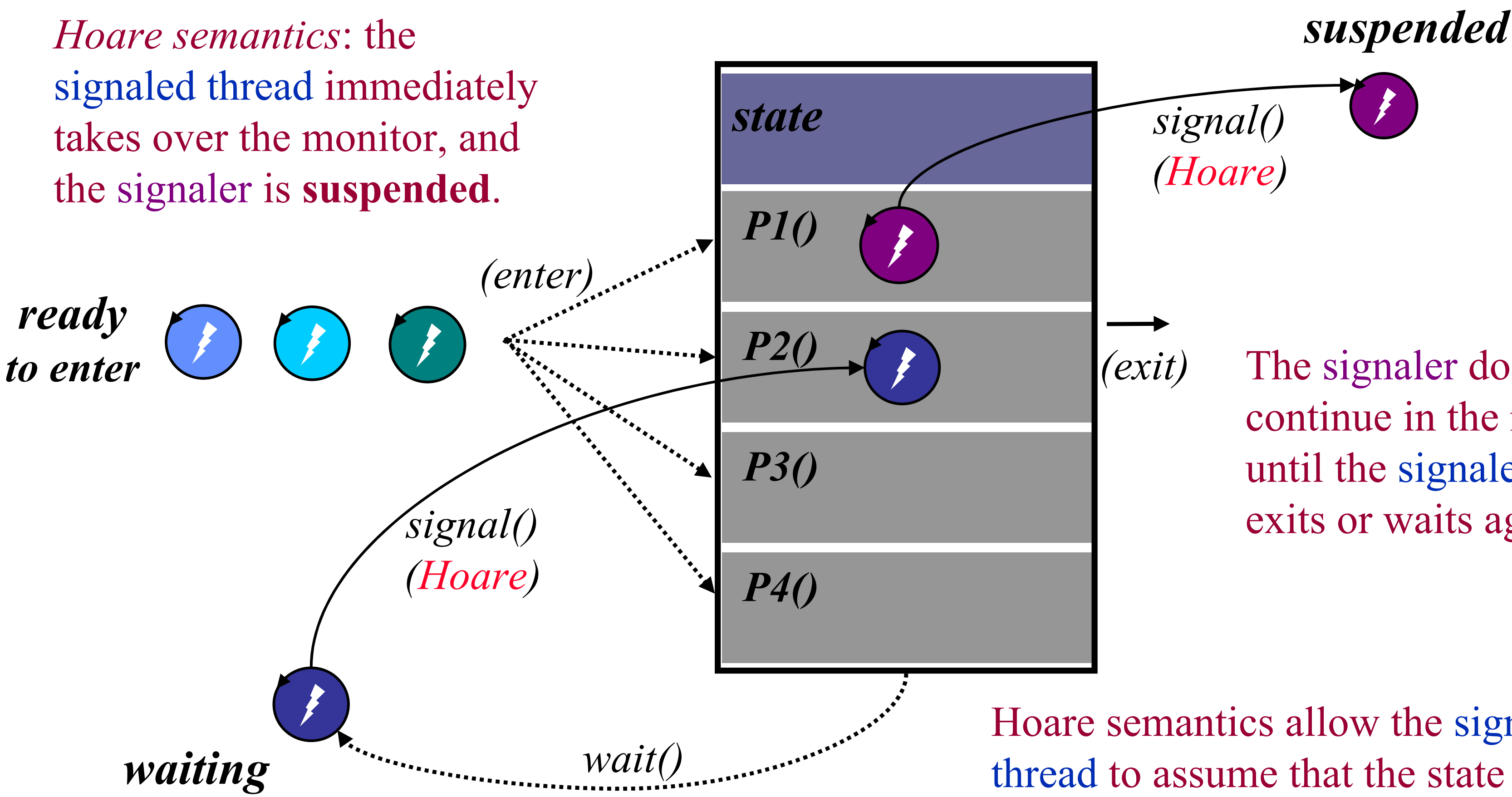
A thread may *wait* in the monitor, allowing another thread to enter.

A thread in the monitor may *signal* a waiting thread, causing it to return from its *wait* and reenter the monitor.

Hoare Semantics

Suppose purple signals blue in the previous example.

Hoare semantics: the signaled thread immediately takes over the monitor, and the signaler is **suspended**.



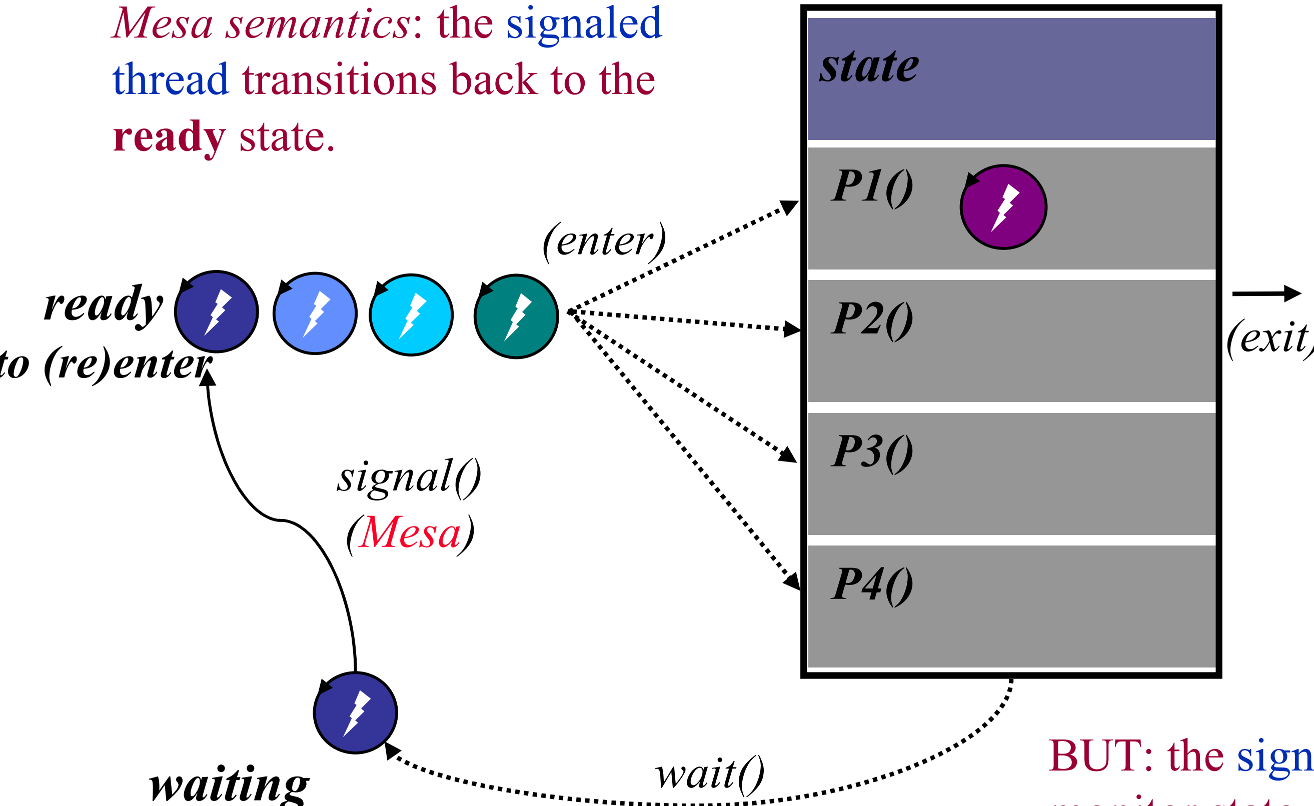
The signaler does not continue in the monitor until the signaled thread exits or waits again.

Hoare semantics allow the signaled thread to assume that the state has not changed since the signal that woke it up.

Mesa Semantics

Suppose again that purple signals blue in the original example.

Mesa semantics: the signaled thread transitions back to the **ready** state.



There is no **suspended** state: the signaler continues until it exits the monitor or waits.

The signaled thread contends with other ready threads to (re)enter the monitor and return from *wait*.

Mesa semantics are easier to understand and implement...

BUT: the signaled thread must examine the monitor state again after the *wait*, as the state may have changed since the *signal*.

Loop before you leap!

From Monitors to Mx/Cv Pairs

Mutexes and condition variables (as in Nachos) are based on monitors, but they are more flexible.

- A monitor is “just like” a module whose state includes a mutex and a condition variable.
- It’s “just as if” the module’s methods *Acquire* the mutex on entry and *Release* the mutex before returning.
- But with *mutexes*, the critical regions within the methods can be defined at a finer grain, to allow more concurrency.
- With *condition variables*, the module methods may wait and signal on multiple independent conditions.
- Nachos (and Topaz and Java) use *Mesa semantics* for their condition variables: *loop before you leap*!

Mutual Exclusion in Java

Mutexes and condition variables are built in to every Java object.

- no explicit classes for mutuxes and condition variables

Every object is/has a "*monitor*".

- At most one thread may "own" any given object's monitor.
- A thread becomes the owner of an object's monitor by
  - executing a method declared as *synchronized*
    - some methods may choose not to enforce mutual exclusion (unsynchronized)
  - by executing the body of a *synchronized* statement
    - supports finer-grained locking than "pure monitors" allow
    - exactly identical to the Modula-2 "LOCK(m) DO" construct in Birrell

Wait/Notify in Java

Every Java object may be treated as a condition variable for threads using its monitor.

```
public class Object {
        void notify();       /* signal */
        void notifyAll(); /* broadcast */
        void wait();
        void wait(long timeout);
}
```

```
public class PingPong (extends Object) {
        public synchronized void PingPong() {
                while(true) {
                        notify();
                        wait();
                }
        }
}
```

A thread must own an object's monitor to call wait/notify, else the method raises an *IllegalMonitorStateException*.

Wait(*) waits until the timeout elapses or another thread notifies, then it waits some more until it can re-obtain ownership of the monitor: *Mesa semantics*.

Loop before you leap!

Semaphores vs. Condition Variables

1. *V()* differs from *Signal* in that:
   - *Signal* has no effect if no thread is waiting on the condition.

     Condition variables are not variables! They have no value!
   - *V()* has the same effect whether or not a thread is waiting.

     Semaphores retain a "memory" of calls to *V()* .
2. *P()* differs from *Wait* in that:
   - *P()* checks the condition and blocks only if necessary.

     no need to recheck the condition after returning from *P()*

     wait condition is defined internally, but is limited to a counter
   - *Wait* is explicit: it does not check the condition, ever.

     condition is defined externally and protected by integrated mutex

Guidelines for Condition Variables

1. Understand/document the condition(s) associated with each CV.

    What are the waiters waiting for?

    When can a waiter expect a *signal*?

2. Always check the condition to detect spurious wakeups after returning from a *wait*: “loop before you leap”!

    Another thread may beat you to the mutex.

    The signaler may be careless.

    A single condition variable may have multiple conditions.

3. Don’t forget: *signals on condition variables do not stack!*

    A signal will be lost if nobody is waiting: always check the wait condition before calling *wait*.

Guidelines for Choosing Lock Granularity

1. *Keep critical sections short*. Push "noncritical" statements outside of critical sections to reduce contention.

2. *Limit lock overhead*. Keep to a minimum the number of times mutexes are acquired and released.

   Note tradeoff between contention and lock overhead.

3. *Use as few mutexes as possible, but no fewer*.

   Choose lock scope carefully: if the operations on two different data structures can be separated, it **may** be more efficient to synchronize those structures with separate locks.

   Add new locks only as needed to reduce contention. "Correctness first, performance second!"

More Locking Guidelines

1. Write code whose correctness is obvious.
2. Strive for symmetry.
   - Show the Acquire/Release pairs.
   - Factor locking out of interfaces.
   - Acquire and Release at the same layer in your "layer cake" of abstractions and functions.
3. Hide locks behind interfaces.
4. Avoid nested locks.
   - If you must have them, try to impose a strict order.
5. Sleep high; lock low.
   - Design choice: where in the layer cake should you put your locks?