Threads and Concurrency

DUKE Systems & Architecture

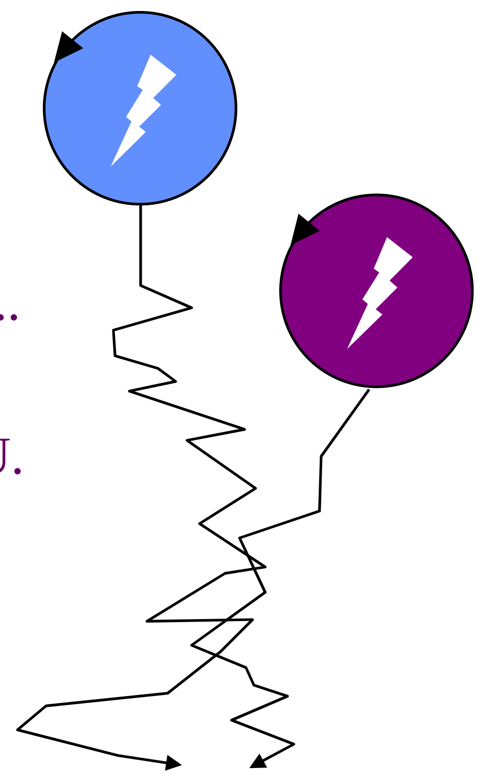
Threads

A *thread* is a schedulable stream of control.

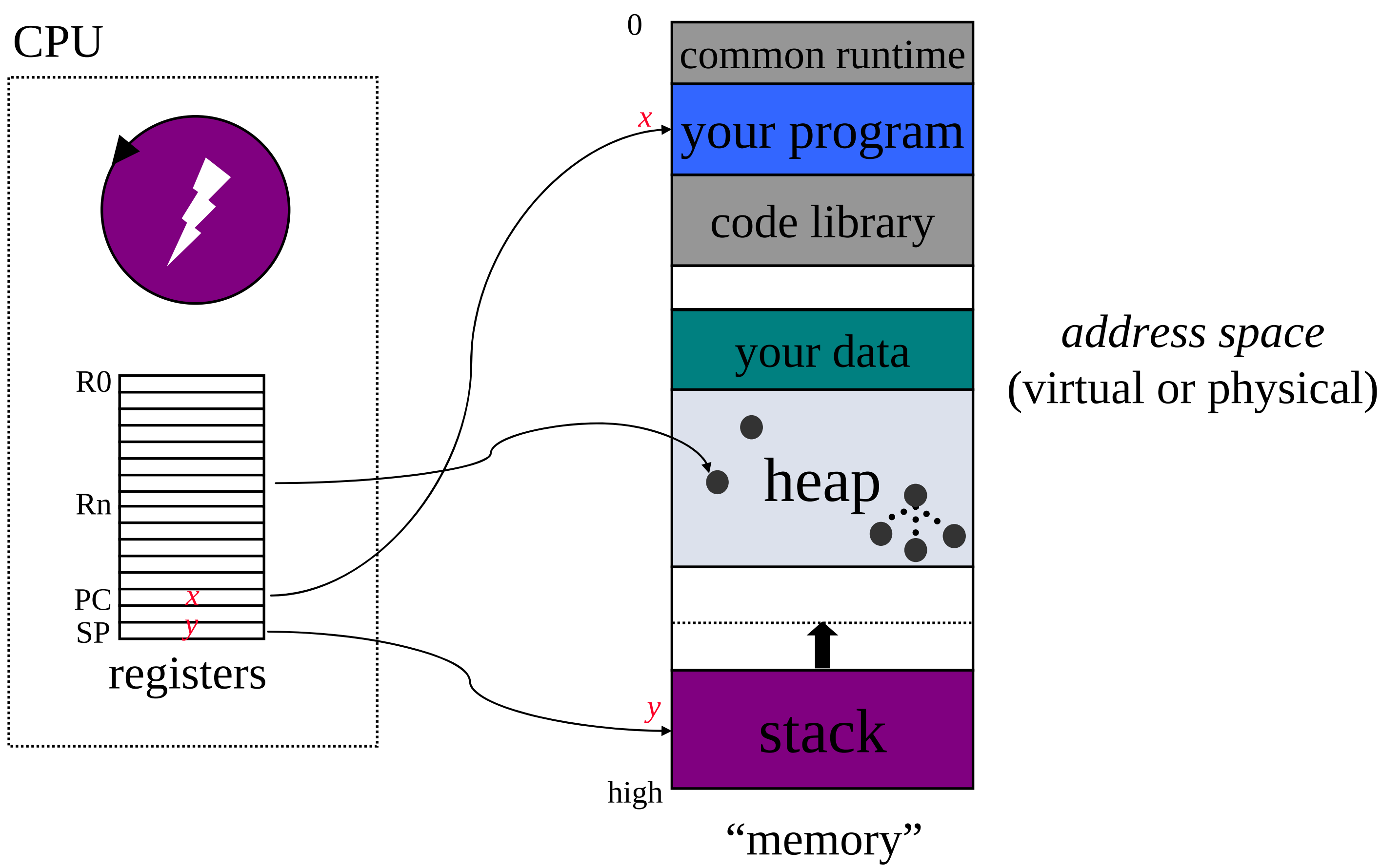
- defined by CPU register values (PC, SP)
- *suspend*: save register values in memory
- *resume*: restore registers from memory

Multiple threads can execute independently:

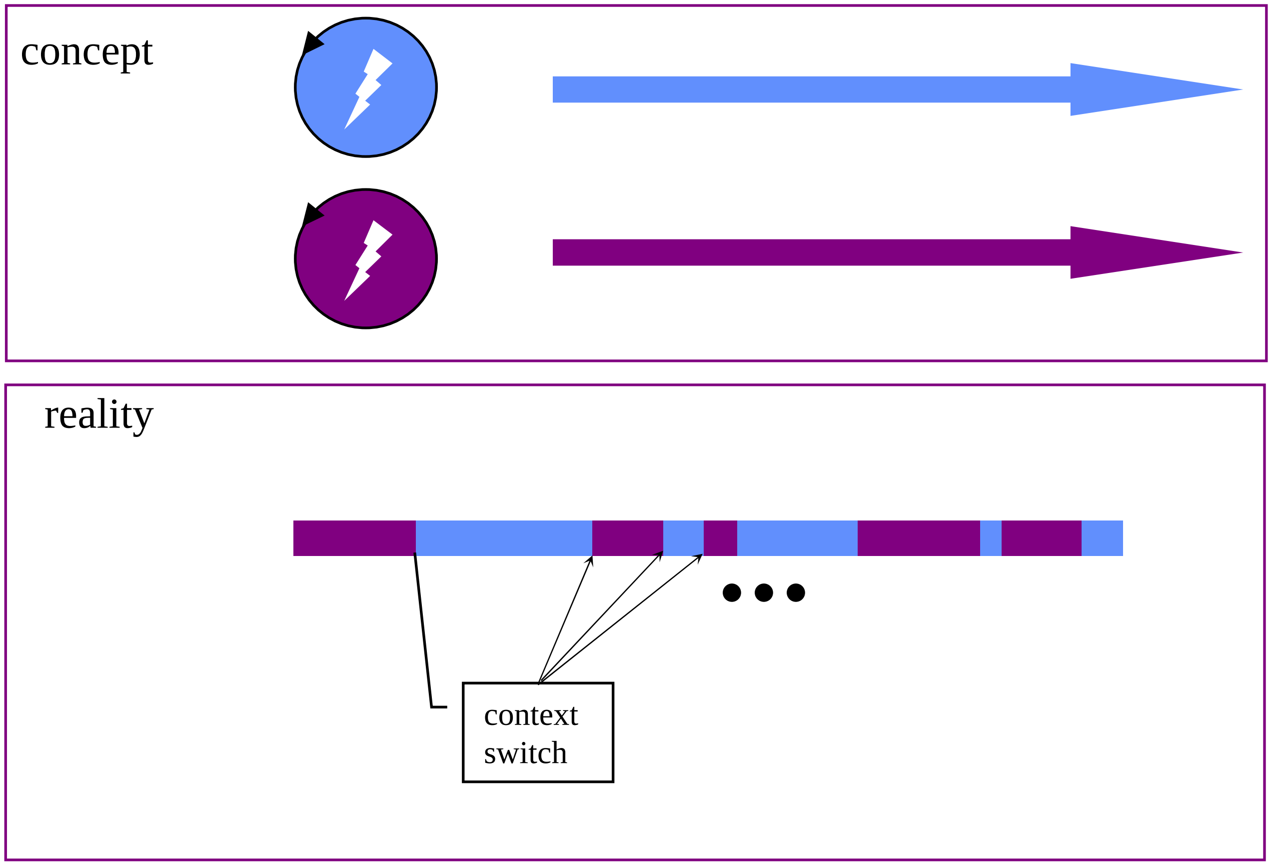
- They can run in parallel on multiple CPUs...
  - ***physical concurrency***
- …or arbitrarily interleaved on a single CPU.
  - ***logical concurrency***
- Each thread must have its own stack.



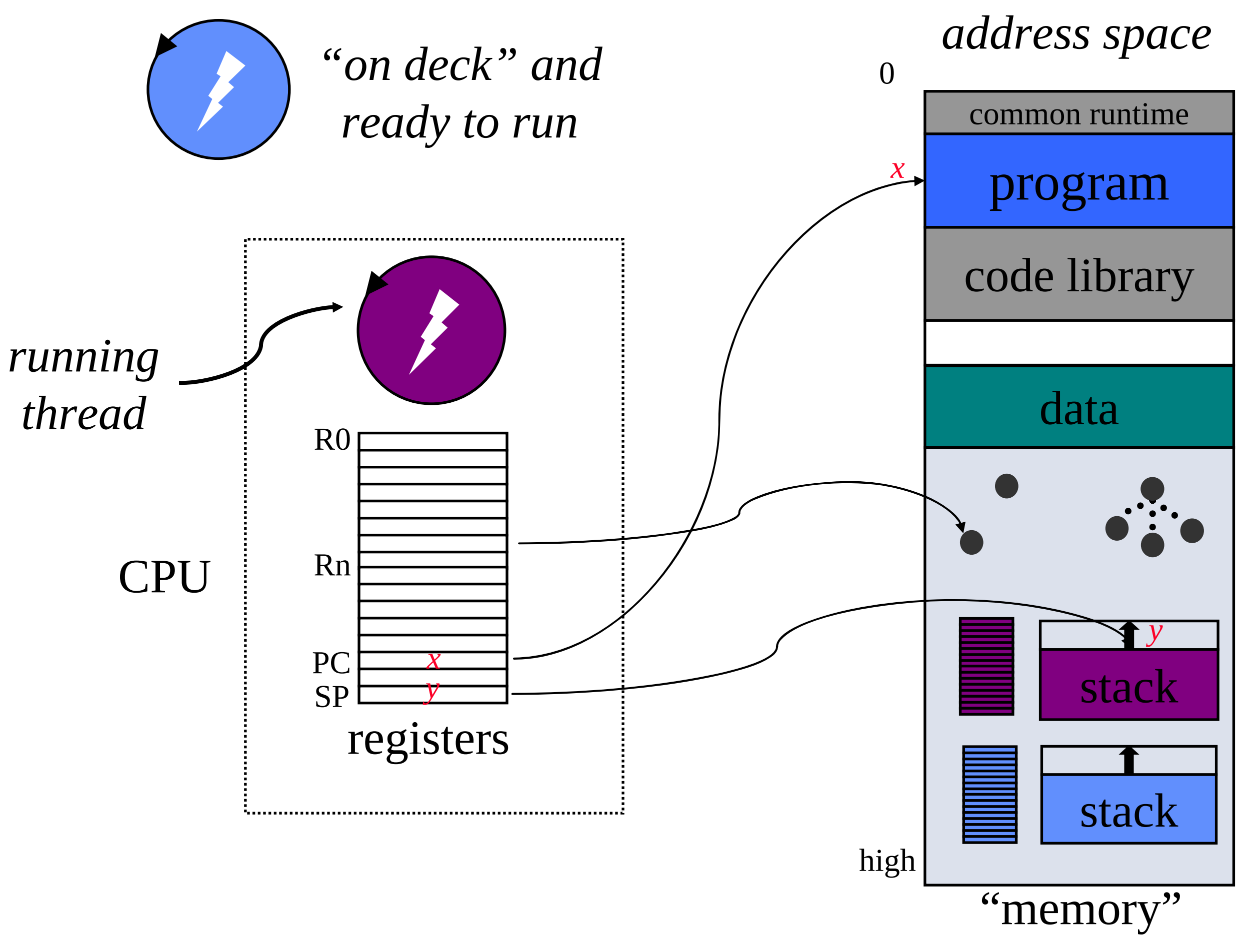
A Peek Inside a Running Program



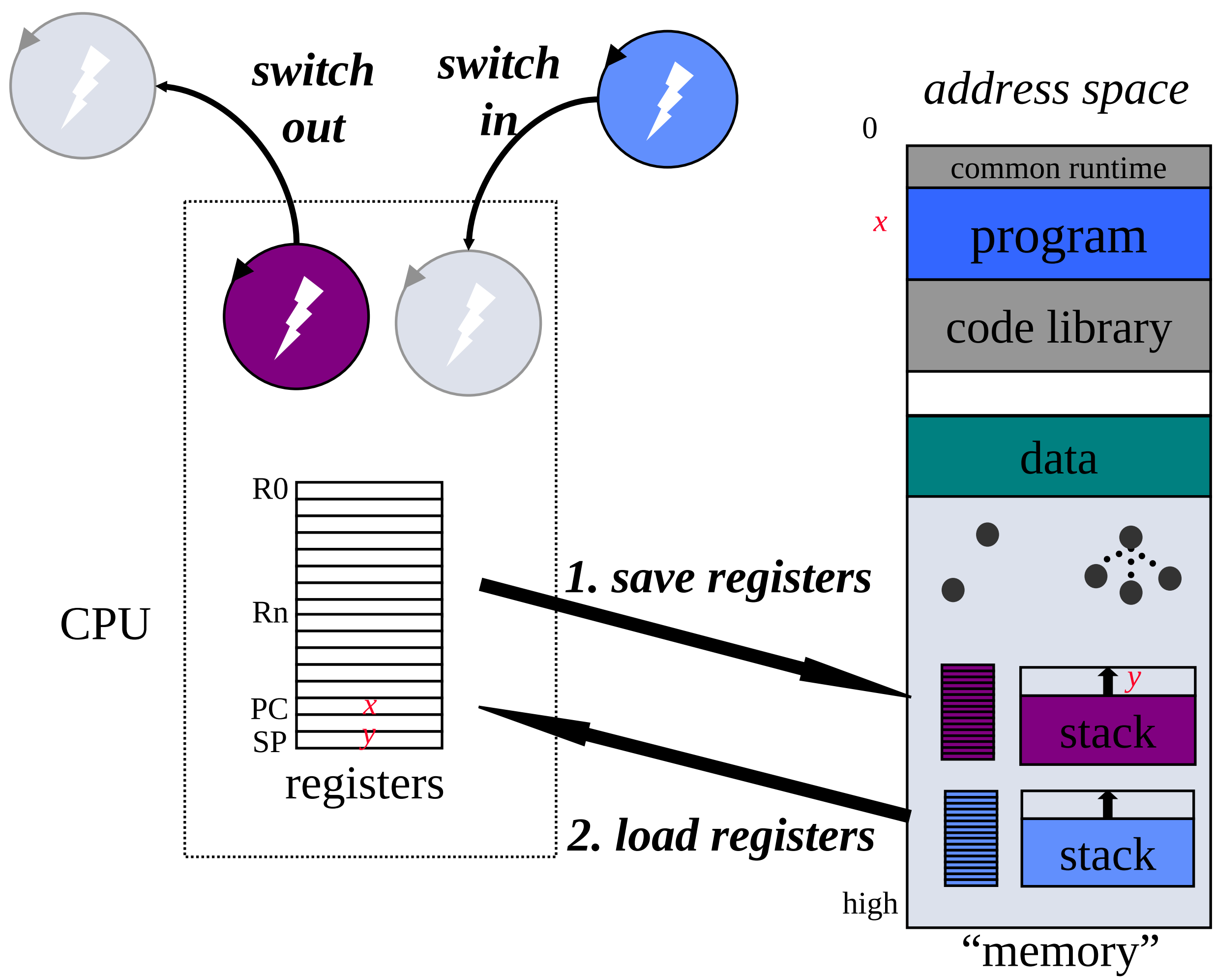
Two Threads Sharing a CPU



A Program With Two Threads

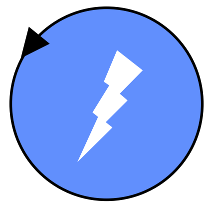


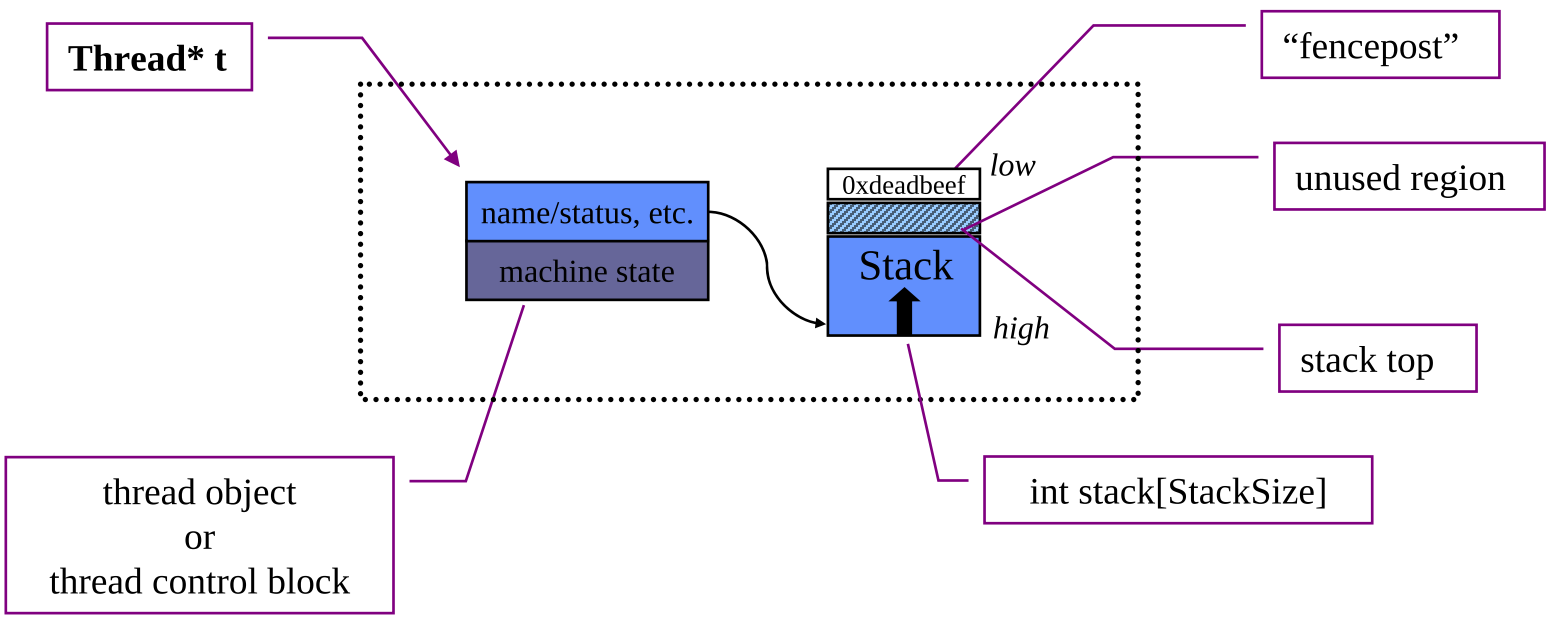
Thread Context Switch



Example: A Nachos Thread

```
t = new Thread(name);
t->Fork(MyFunc, arg);
currentThread->Sleep();
currentThread->Yield();
```





Example: Context Switch on MIPS

```
/*
 * Save context of the calling thread (old), restore registers of
 *  the next thread to run (new), and return in context of new.
 */
switch/MIPS (old, new) {
        old->stackTop = SP;
        save RA in old->MachineState[PC];
        save callee registers in old->MachineState

        restore callee registers from new->MachineState
        RA = new->MachineState[PC];
        SP = new->stackTop;

        return (to RA)
}
```

*Save current stack pointer and caller's return address in **old** thread object.*

Caller-saved registers (if needed) are already saved on the thread's stack.

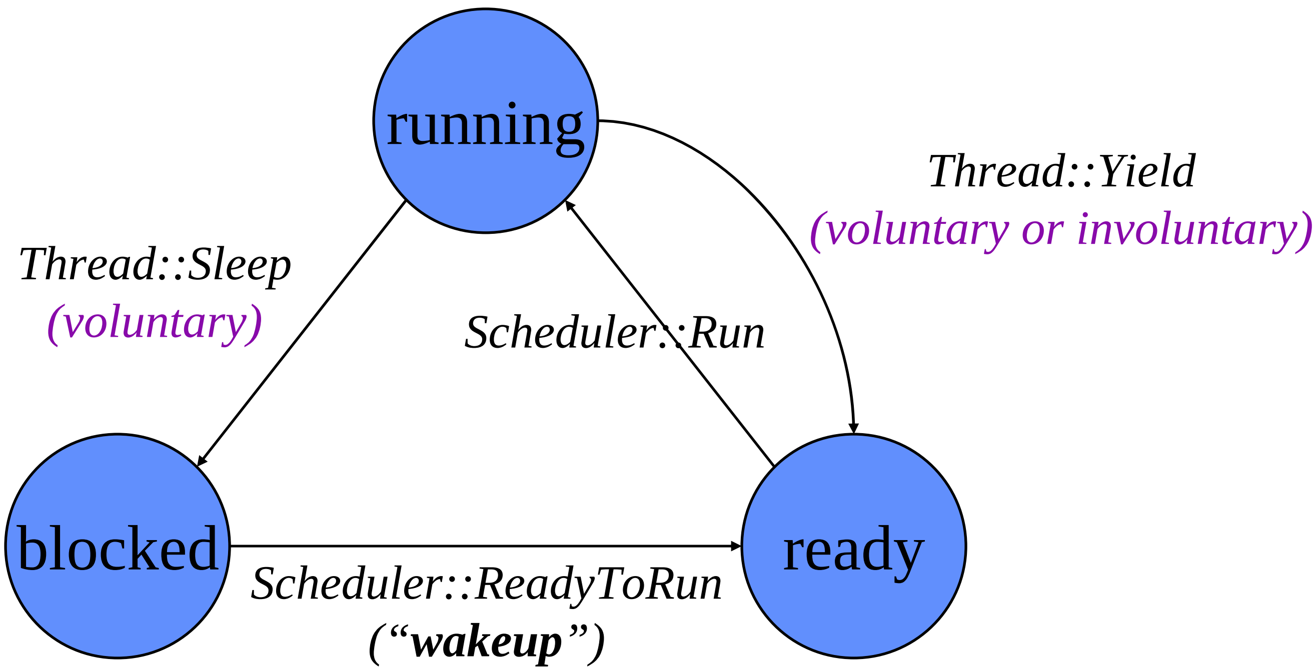
Caller-saved regs restored automatically on return.

*Switch off of **old** stack and back to **new** stack.*

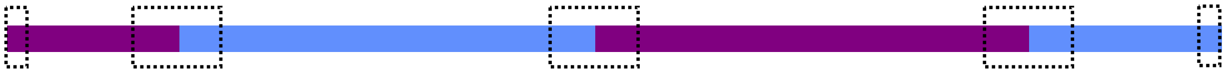
*Return to procedure that called switch in **new** thread.*

```
/*
 *  Save context of the calling thread (old), restore registers of
 *  the next thread to run (new), and return in context of new.
 */
switch/MIPS (old, new) {
        old->stackTop = SP;
        save RA in old->MachineState[PC];
        save callee registers in old->MachineState

        restore callee registers from new->MachineState
        RA = new->MachineState[PC];
        SP = new->stackTop;

        return (to RA)
}
```

Thread States and Transitions



Example: Sleep and Yield (Nachos)

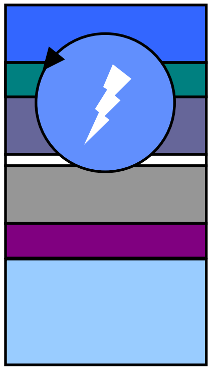


```
Yield() {
    next = scheduler->FindNextToRun();
    if (next != NULL) {
        scheduler->ReadyToRun(this);
        scheduler->Run(next);
    }
}
```

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

Threads vs. Processes

1. The *process* is a *kernel abstraction* for an independent executing program.



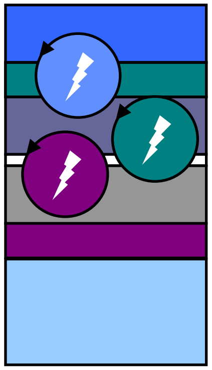
includes at least one "thread of control"

also includes a private address space (VAS)

- requires OS kernel support

(but some use *process* to mean what we call *thread)*

2. Threads may share an address space



threads have "context" just like vanilla processes

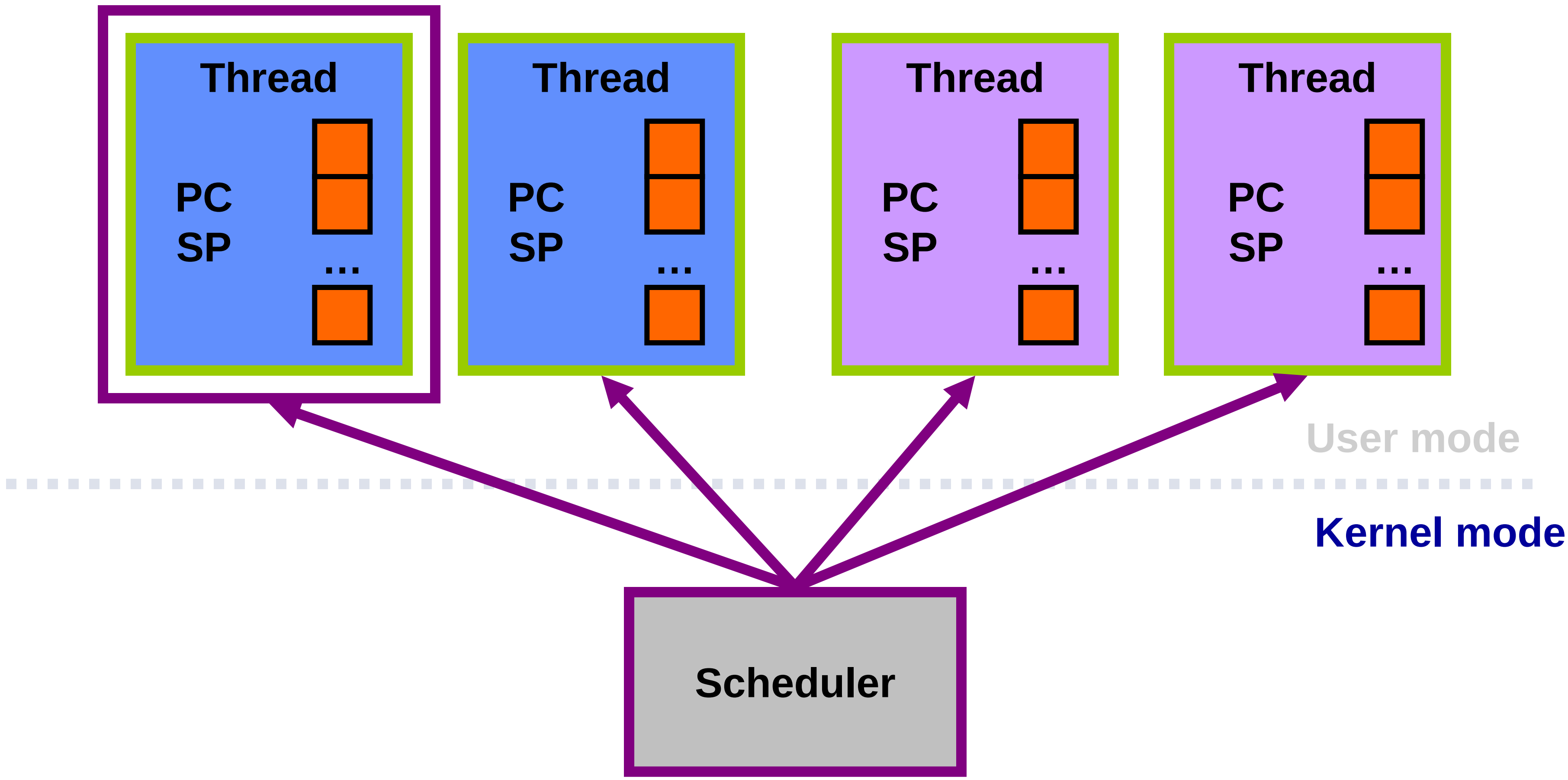
- *thread context switch* vs. *process context switch*

every thread must exist within some process VAS

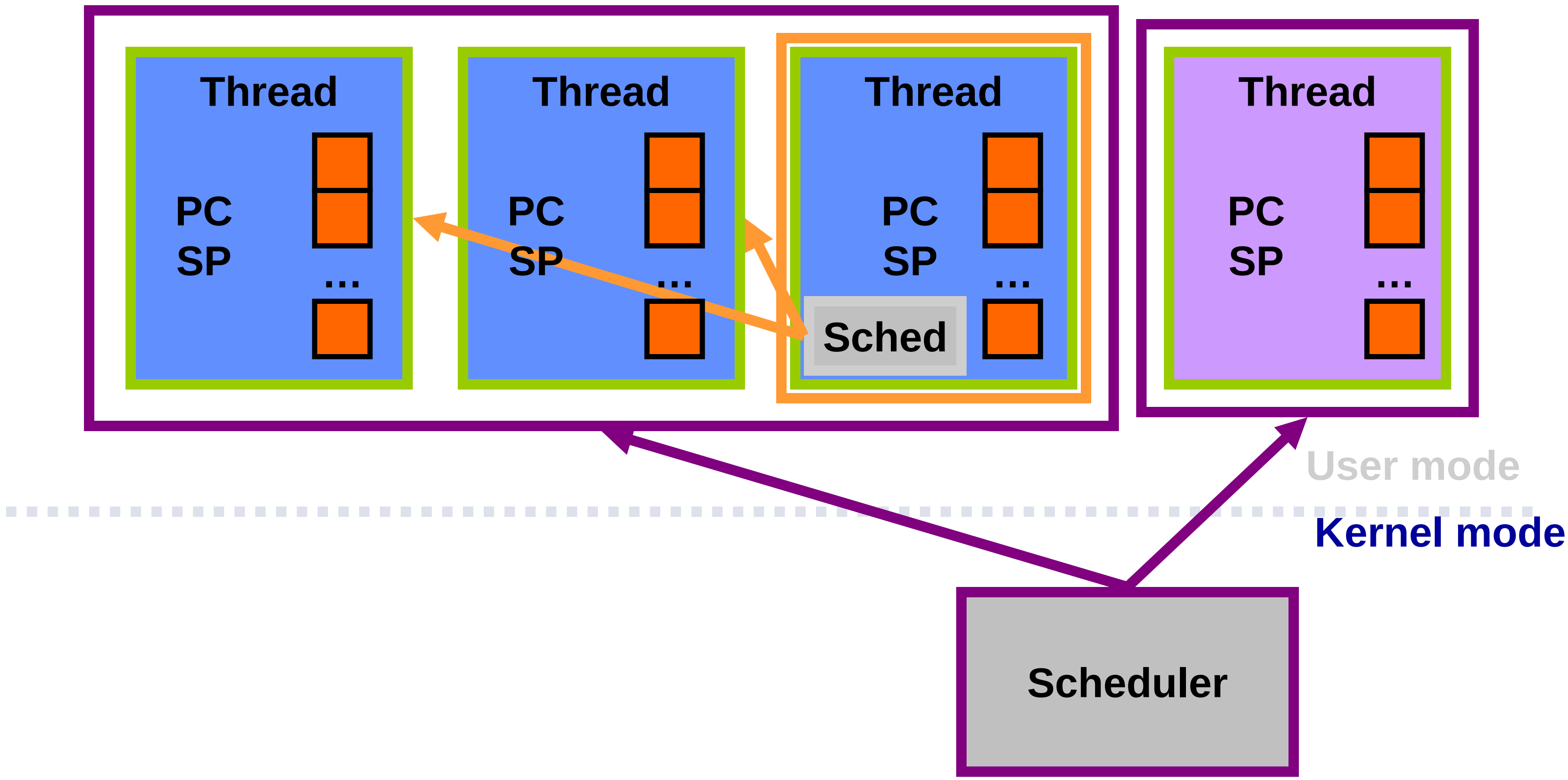
processes may be "multithreaded"

Thread::Fork

Kernel threads



User threads



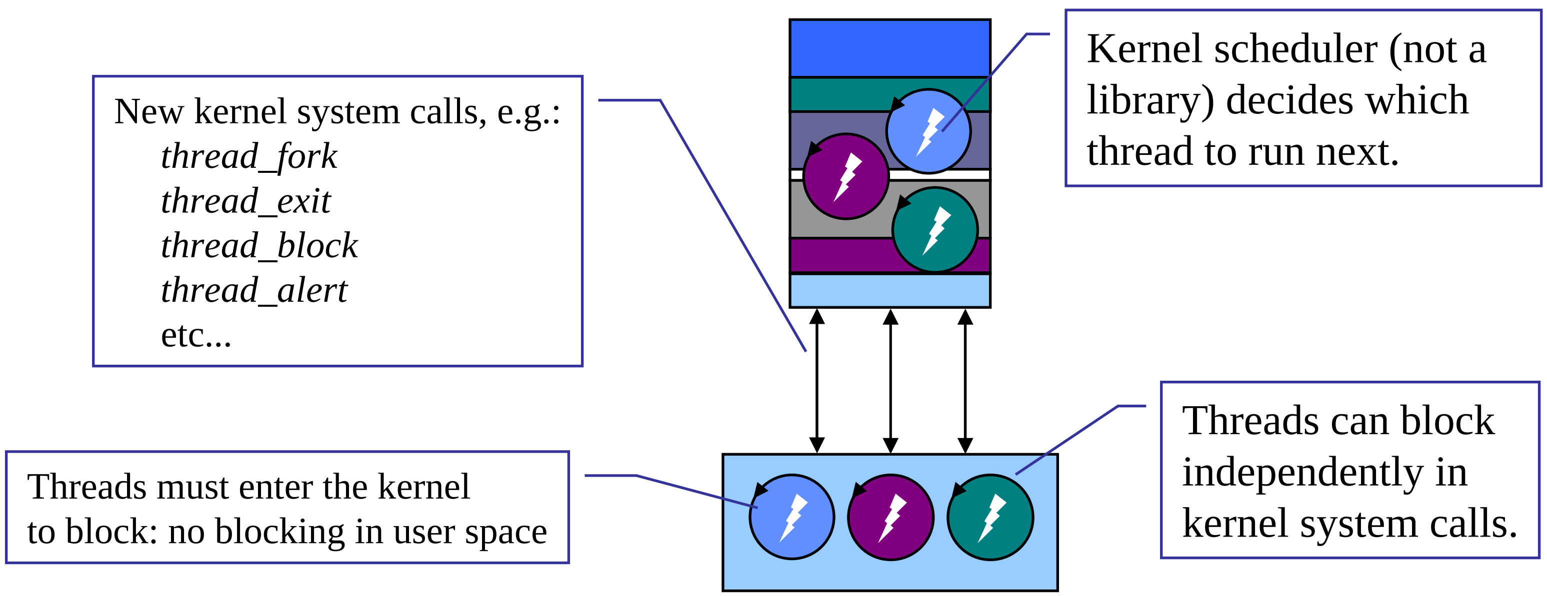
Kernel-Supported Threads

Most newer OS kernels have *kernel-supported threads*.

- thread model and scheduling defined by OS

  NT, advanced Unix, etc.

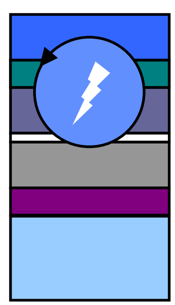
- Linux: threads are “lightweight processes”



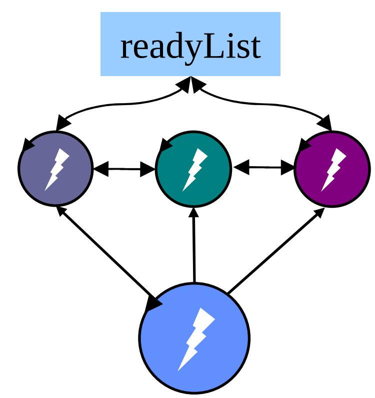
User-level Threads

Can also implement *user-level threads* in a library.

- no special support needed from the kernel (use any Unix)
- thread creation and context switch are fast (no syscall)
- defines its own thread model and scheduling policies



```
while(1) {
      t = get next ready thread;
      scheduler->Run(t);
}
```



Threads in Java

All Java implementations support threads:

- *Thread* class implements *Runnable* interface
- *Thread t = new Thread(); t.run();*
- Typical: create subclasses of *Thread* and *run* them.

If the underlying OS supports native threads (kernel threads), then Java maps its threads onto kernel threads.

- If one thread blocks on a system call, others keep going.
- If no native threads, then a "user-level" implementation

  Threads are not known to the OS kernel.

  System calls by the program/process/JVM are single-threaded.

Concurrency

Working with multiple threads (or processes) introduces *concurrency*: several things are happening "at once".

How can I know the order in which operations will occur?

- ***physical concurrency***

  On a **multiprocessor**, thread executions may be arbitrarily interleaved at the granularity of individual instructions.

- ***logical concurrency***

  On a **uniprocessor**, thread executions may be interleaved as the system switches from one thread to another.

  context switch (suspend/resume)

The Dark Side of Concurrency

With interleaved executions, the order in which threads or processes execute at runtime is *nondeterministic*.

- depends on the exact order and timing of process arrivals
- depends on exact timing of asynchronous devices (disk, clock)
- depends on scheduling policies

Some schedule interleavings may lead to incorrect behavior.

- Open the bay doors *before* you release the bomb.
- Two people can't wash dishes in the same sink at the same time.

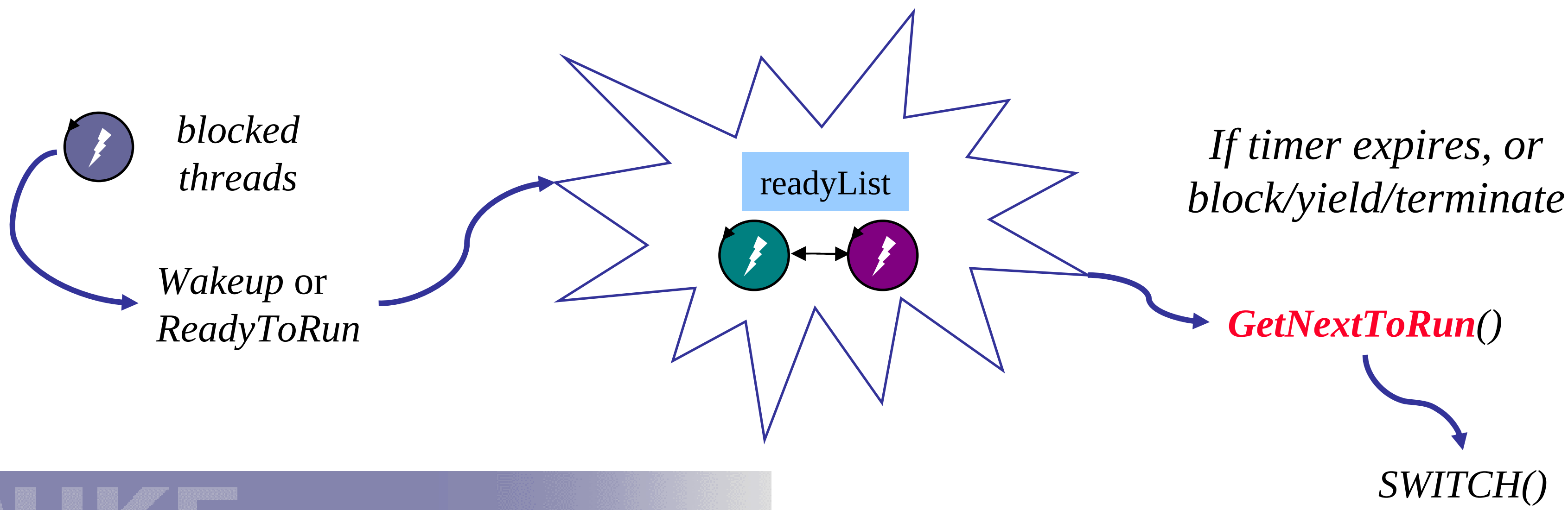
The system must provide a way to coordinate concurrent activities to avoid incorrect interleavings.

CPU Scheduling 101

The CPU scheduler makes a sequence of “moves” that determines the interleaving of threads.

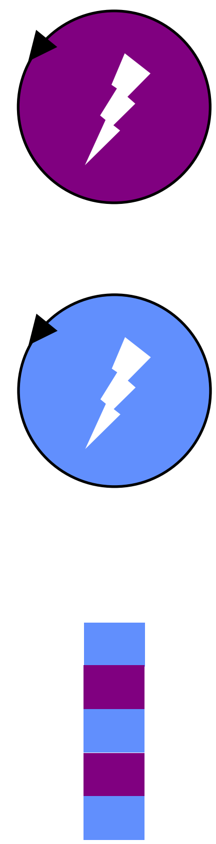
- Programs use synchronization to prevent “bad moves”.
- …but otherwise scheduling choices appear (to the program) to be *nondeterministic*.

The scheduler’s moves are dictated by a *scheduling policy*.

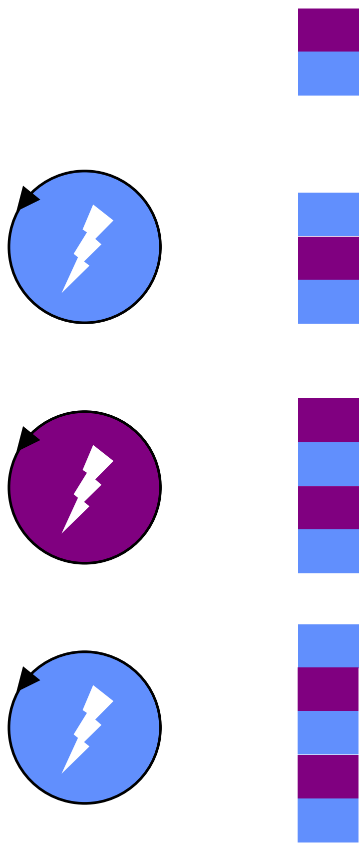


Example: A Concurrent Color Stack

```
InitColorStack() {
        push(blue);
        push(purple);
}

PushColor() {
        if (s[top] == purple) {
                ASSERT(s[top-1] == blue);
                push(blue);
        } else {
                ASSERT(s[top] == blue);
                ASSERT(s[top-1] == purple);
                push(purple);
        }
}
```

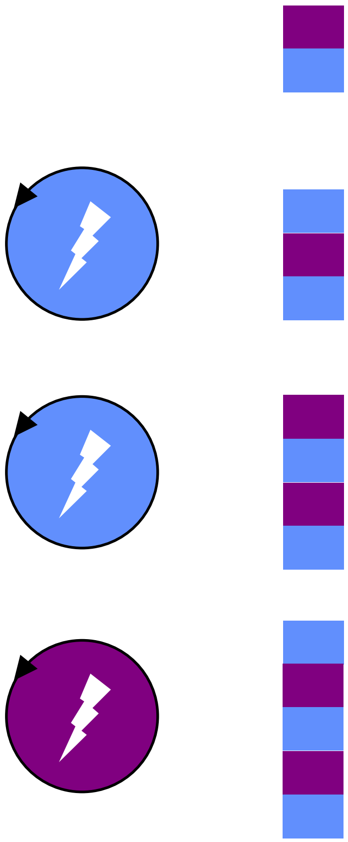


Interleaving the Color Stack #1



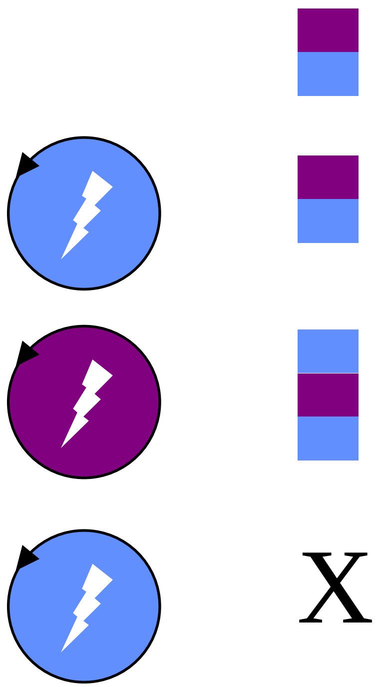
```
PushColor() {
        if (s[top] ==  purple) {
                ASSERT(s[top-1] ==  blue);
                push(blue);
        } else {
                ASSERT(s[top] == blue);
                ASSERT(s[top-1] == purple);
                push(purple);
        }
}

ThreadBody() {
        while(true)
                PushColor();
}
```

Interleaving the Color Stack #2



```
if (s[top] == purple) {
        ASSERT(s[top-1] == blue);
        push(blue);
} else {
        ASSERT(s[top] == blue);
        ASSERT(s[top-1] == purple);
        push(purple);
}
```

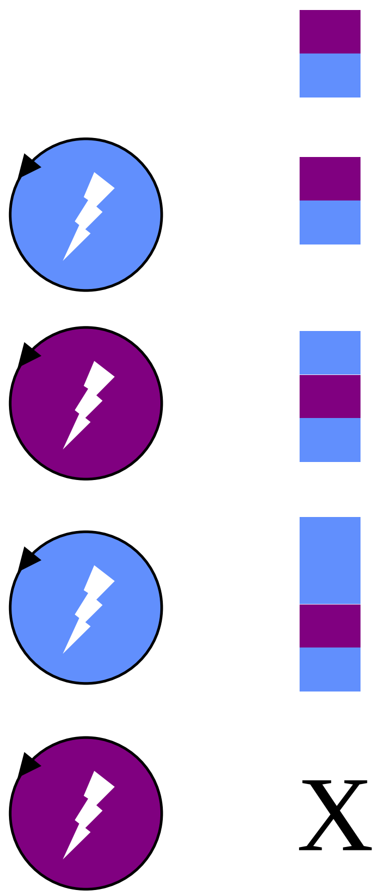
Interleaving the Color Stack #3



Consider a yield here on blue's first call to PushColor().

```
if (s[top] == purple) {
        ASSERT(s[top-1] == blue);
        push(blue);
} else {
        ASSERT(s[top] == blue);
        ASSERT(s[top-1] == purple);
        push(purple);
}
```

Interleaving the Color Stack #4



Consider yield here on blue's first call to PushColor().

```
if (s[top] ==  purple) {
        ASSERT(s[top-1] ==  blue);
        push(blue);
} else {
        ASSERT(s[top] == blue);
        ASSERT(s[top-1] == purple);
        push(purple);
}
```

Race Conditions Defined

1. Every data structure defines *invariant* conditions.

    defines the space of possible *legal* states of the structure

    defines what it means for the structure to be "well-formed"

2. Operations depend on and preserve the invariants.

    The invariant must hold when the operation begins.

    The operation may temporarily violate the invariant.

    The operation restores the invariant before it completes.

3. Arbitrarily interleaved operations violate invariants.

    Rudely interrupted operations leave a mess behind for others.

4. Therefore we must constrain the set of possible schedules.

Avoiding Races #1

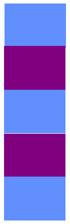
1. Identify *critical sections*, code sequences that:
    - rely on an invariant condition being true;
    - temporarily violate the invariant;
    - transform the data structure from one legal state to another;
    - or make a sequence of actions that assume the data structure will not "change underneath them".
2. *Never sleep or yield in a critical section.*

    Voluntarily relinquishing control may allow another thread to run and "trip over your mess" or modify the structure while the operation is in progress.

3. Prevent another thread/process from entering a mutually critical section, which would result in a race.

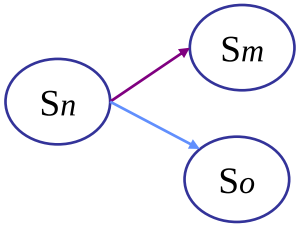
Critical Sections in the Color Stack

```
InitColorStack() {
        push(blue);
        push(purple);
}

PushColor() {
        if (s[top] ==  purple) {
                ASSERT(s[top-1] ==  blue);
                push(blue);
        } else {
                ASSERT(s[top] == blue);
                ASSERT(s[top-1] == purple);
                push(purple);
        }
}
```



Resource Trajectory Graphs

Resource trajectory graphs (RTG) depict the thread scheduler's "random walk" through the space of possible system states.



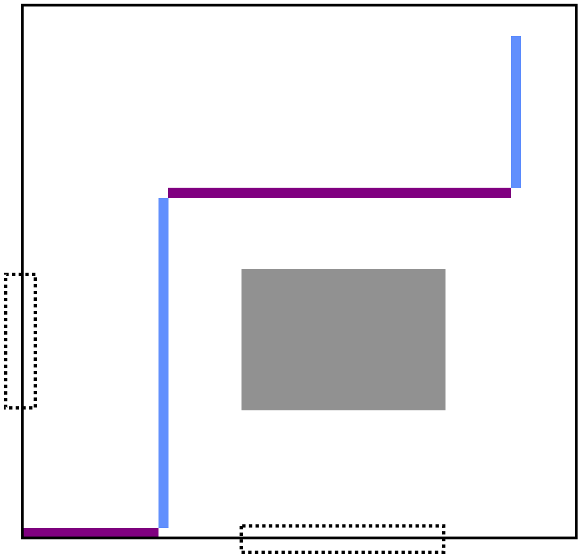
RTG for N threads is N-dimensional.

Thread *i* advances along axis *I*.

Each point represents one state in the set of all possible system states.

cross-product of the possible states of all threads in the system

(But not all states in the cross-product are legally reachable.)



Relativity of Critical Sections

1. If a thread is executing a critical section, never permit another thread to enter the same critical section.

    Two executions of the same critical section on the same data are *always* "mutually conflicting" (assuming it modifies the data).

2. If a thread is executing a critical section, never permit another thread to enter a *related* critical section.

    Two different critical sections may be mutually conflicting.

    E.g., if they access the same data, and at least one is a writer.

    E.g., *List::Add* and *List::Remove* on the same list.

3. Two threads may safely enter *unrelated* critical sections.

    If they access different data or are reader-only.

Mutual Exclusion

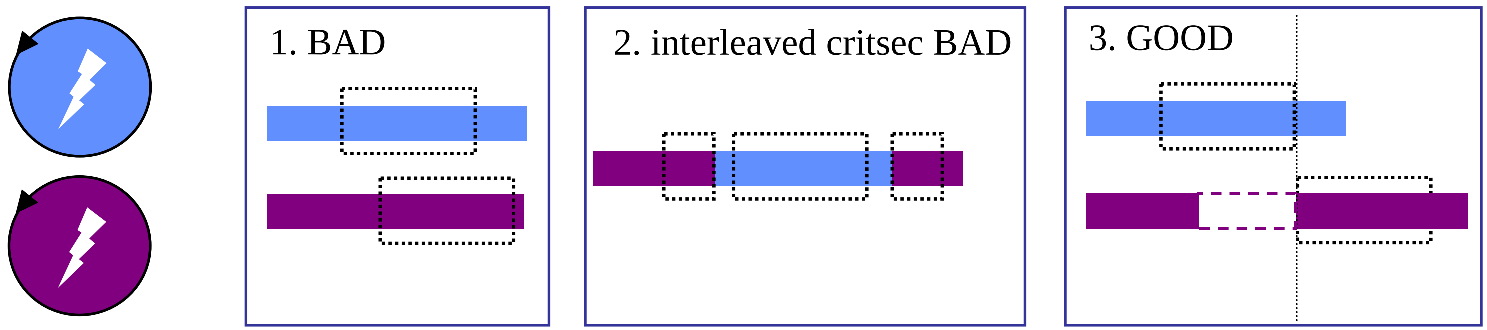
Race conditions can be avoiding by ensuring *mutual exclusion* in critical sections.

- Critical sections are code sequences that are vulnerable to races.

  Every race (possible incorrect interleaving) involves two or more threads executing related critical sections concurrently.

- To avoid races, we must *serialize* related critical sections.

  Never allow more than one thread in a critical section at a time.



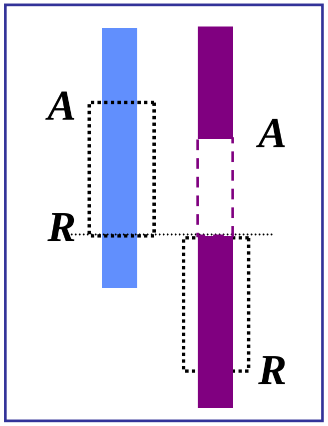
Locks

Locks can be used to ensure mutual exclusion in conflicting critical sections.

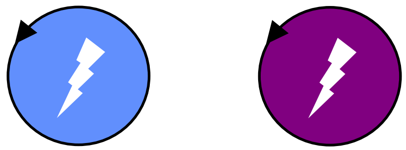
- A lock is an object, a data item in memory.

  Methods: *Lock::Acquire* and *Lock::Release.*

- Threads pair calls to *Acquire* and *Release*.
- *Acquire* before entering a critical section.
- *Release* after leaving a critical section.
- Between *Acquire*/*Release*, the lock is *held*.
- *Acquire* does not return until any previous holder releases.
- Waiting locks can spin (a *spinlock*) or block (a *mutex*).

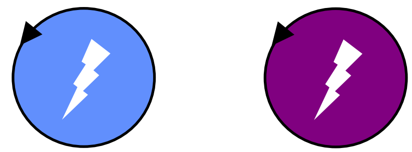


Example: Per-Thread Counts and Total



```
/* shared by all threads */
int counters[N];
int total;

/*
 * Increment a counter by a specified value, and keep a running sum.
 * This is called repeatedly by each of N threads.
 * tid is an integer thread identifier for the current thread.
 * value is just some arbitrary number.
 */
void
TouchCount(int tid, int value)
{
        counters[tid] += value;
        total += value;
}
```

Using Locks: An Example



```
int counters[N];
int total;
Lock *lock;

/*
 * Increment a counter by a specified value, and keep a running sum.
 */
void
TouchCount(int tid, int value)
{
        lock->Acquire();
        counters[tid] += value;        /* critical section code is atomic...*/
        total += value;                /* …as long as the lock is held */
        lock->Release();
}
```

Reading Between the Lines of C

load add store	load add store

```
/*
   counters[tid] += value;
   total += value;
*/

load      counters, R1     ; load counters base
load      8(SP), R2        ; load tid index
shl       R2, #2, R2       ; index = index * sizeof(int)
add       R1, R2, R1       ; compute index to array
load      4(SP), R3        ; load value
load      (R1), R2         ; load counters[tid]
add       R2, R3, R2       ; counters[tid] += value
store     R2, (R1)         ; store back to counters[tid]
load      total, R2        ; load total
add       R2, R3, R2       ; total += value
store     R2, total        ; store total
```

vulnerable between **load** and **store** of *counters[tid]*...but it's non-shared.

vulnerable between **load** and **store** of *total*, which is shared.

Lesson: never assume that some line of code "executes atomically": it may compile into a sequence of instructions that does not execute atomically on the machine.

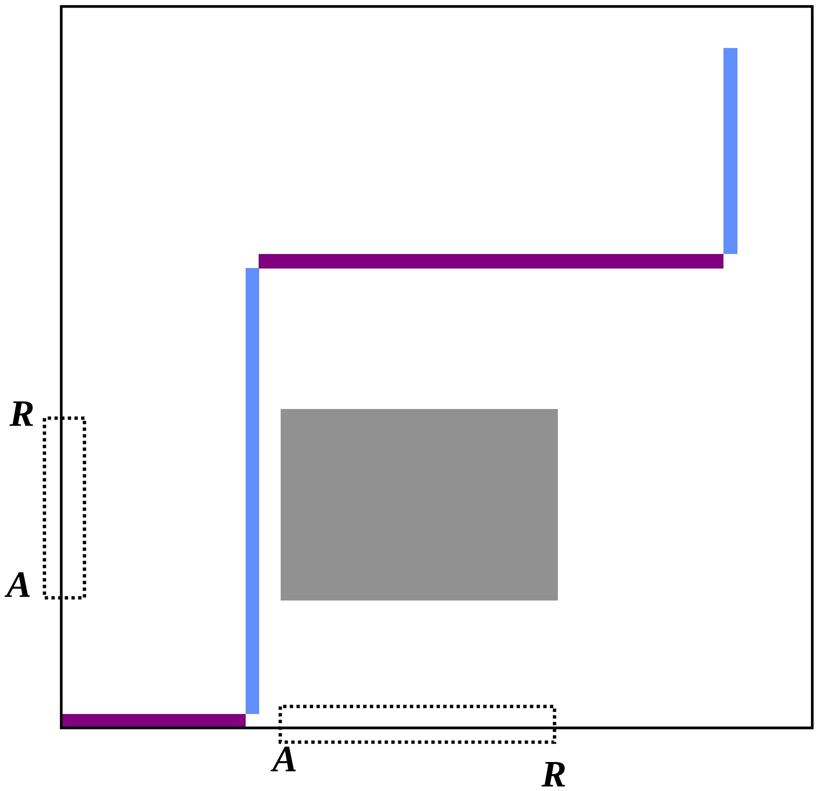
Things Your Mother Warned You About #1

```
Lock dirtyLock;
List dirtyList;
Lock wiredLock;
List wiredList;

struct buffer {
	unsigned int flags;
	struct OtherStuff etc;
};

void MarkDirty(buffer* b) {
	dirtyLock.Acquire();
	b->flags |= DIRTY;
	dirtyList.Append(b);
	dirtyLock.Release();
}
```

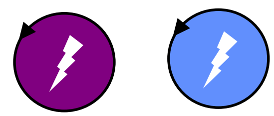
```
#define WIRED    0x1
#define  DIRTY    0x2
#define  FREE      0x4

void MarkWired(buffer *b) {
	wiredLock.Acquire();
	b->flags |= WIRED;
	wiredList.Append(b);
	wiredLock.Release();
}
```

Lesson?

Portrait of a Lock in Motion



A New Synchronization Problem: Ping-Pong



```
void
PingPong() {
        while(not done) {
                if (blue)
                  switch to purple;
                if (purple)
                  switch to blue;
        }
}
```

How to do this correctly using sleep/wakeup?

How to do it without using sleep/wakeup?

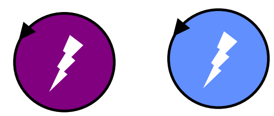
Ping-Pong with Sleep/Wakeup?

```
void
PingPong() {
    while(not done) {
        blue->Sleep();
        purple->Wakeup();
    }
}
```

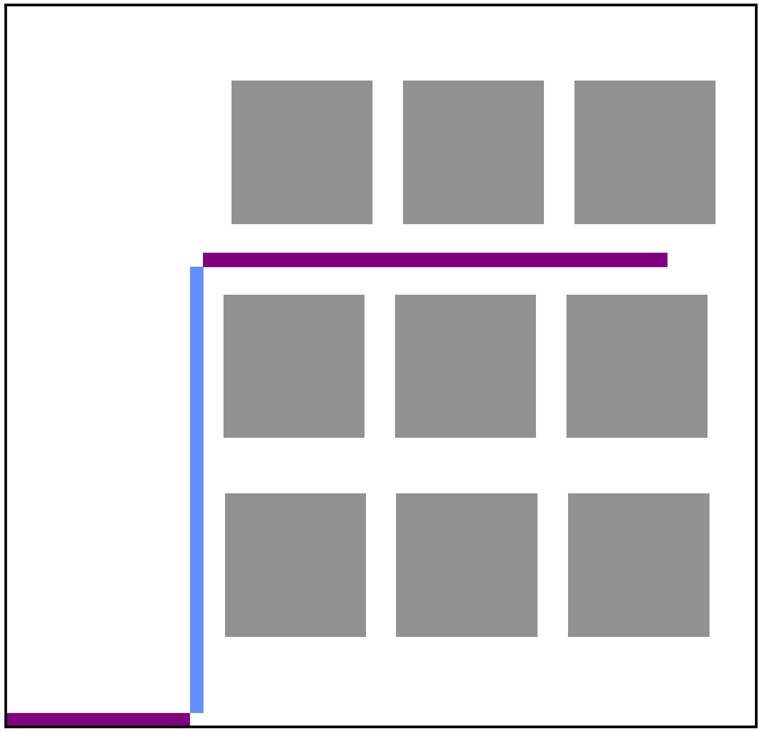
```
void
PingPong() {
    while(not done) {
        blue->Wakeup();
        purple->Sleep();
    }
}
```

Ping-Pong with Mutexes?

```
void
PingPong() {
        while(not done) {
                Mx->Acquire();
                Mx->Release();
        }
}
```



Mutexes Don't Work for Ping-Pong



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.

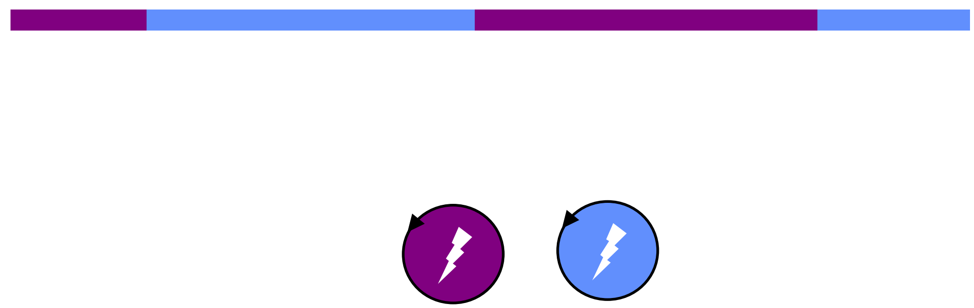
Condition:: Signal(Lock*)

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();
}
```

Mutual Exclusion in Java

Mutexes and condition variables are built in to every object.

- no classes for mutexes and condition variables

Every Java 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*
  - by executing the body of a *synchronized* statement
  - Entry to a synchronized block is an "acquire"; exit is "release"
- Built-in condition variable

Java wait/notify*

Monitors provide condition variables with two operations which can be called when the lock is held

- wait: an unconditional suspension of the calling thread (the thread is placed on a queue associated with the condition variable). The thread is *sleeping*, *blocked*, *waiting*.
- notify: one thread is taken from the queue and made runnable
- notifyAll: all suspended threads are made runnable
- **notify and notifyAll have no effect if no threads are waiting on the condition variable**
- **Each notified thread reacquires the monitor before returning from wait().**

Example: 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*.

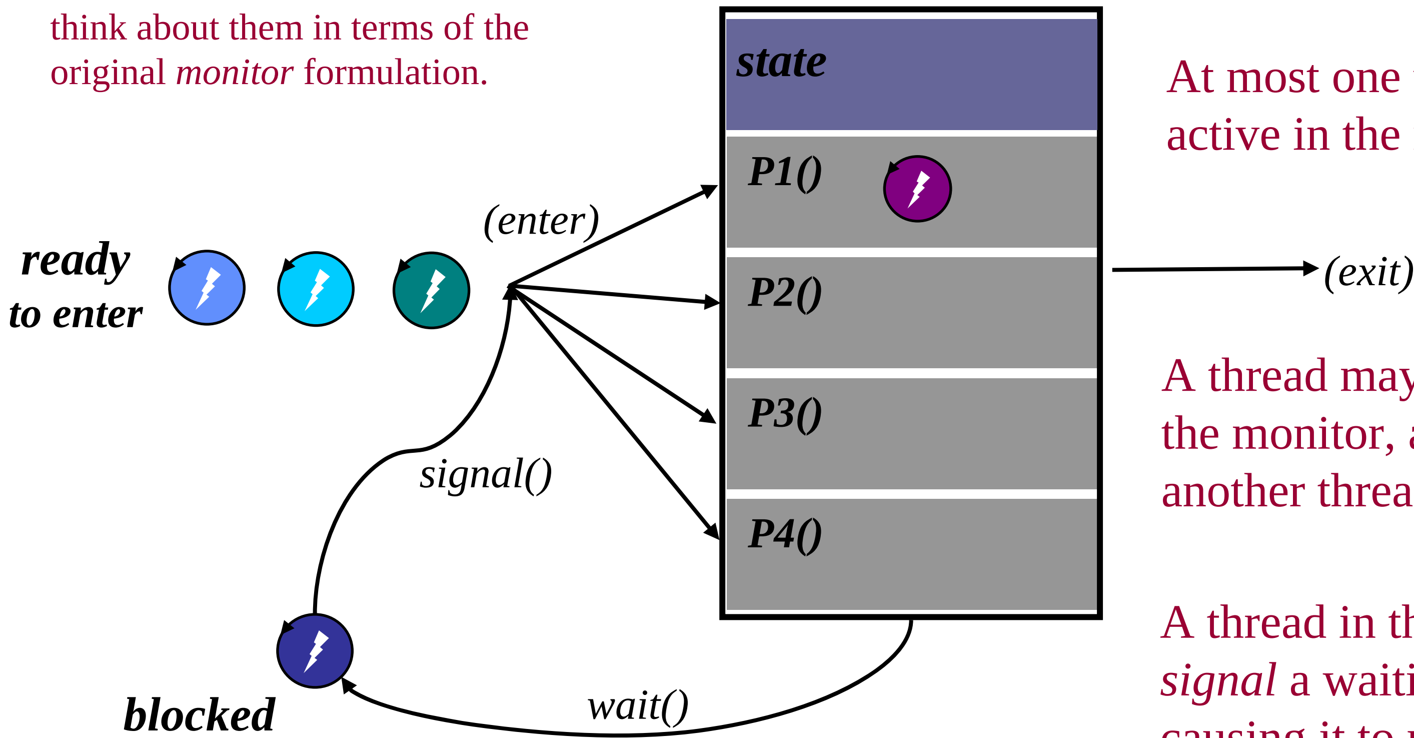
Wait(*) waits until the timeout elapses or another thread notifies.

Back to the Roots: 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.

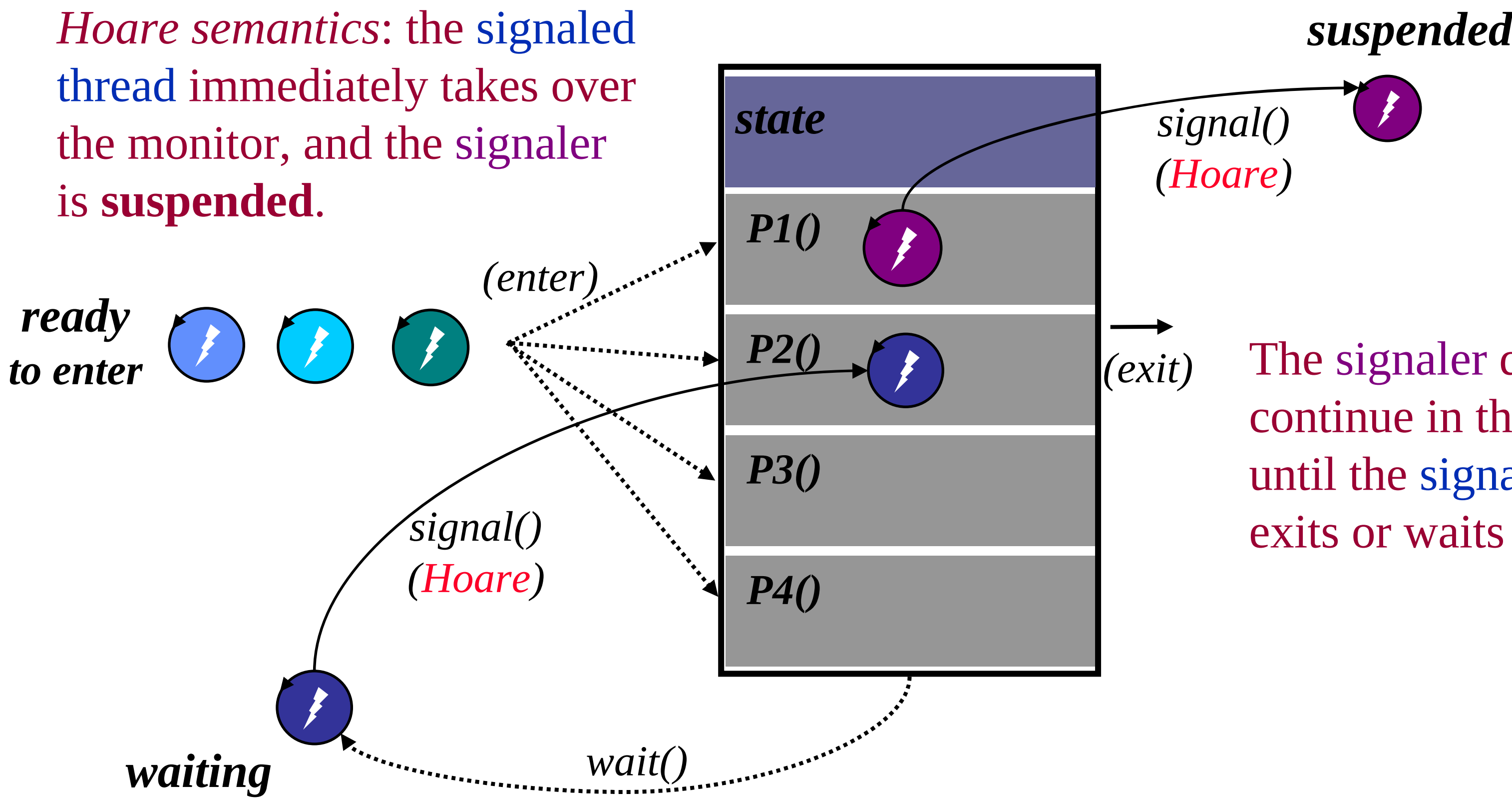
A thread may *wait* in the monitor, allowing another thread to enter.

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**.

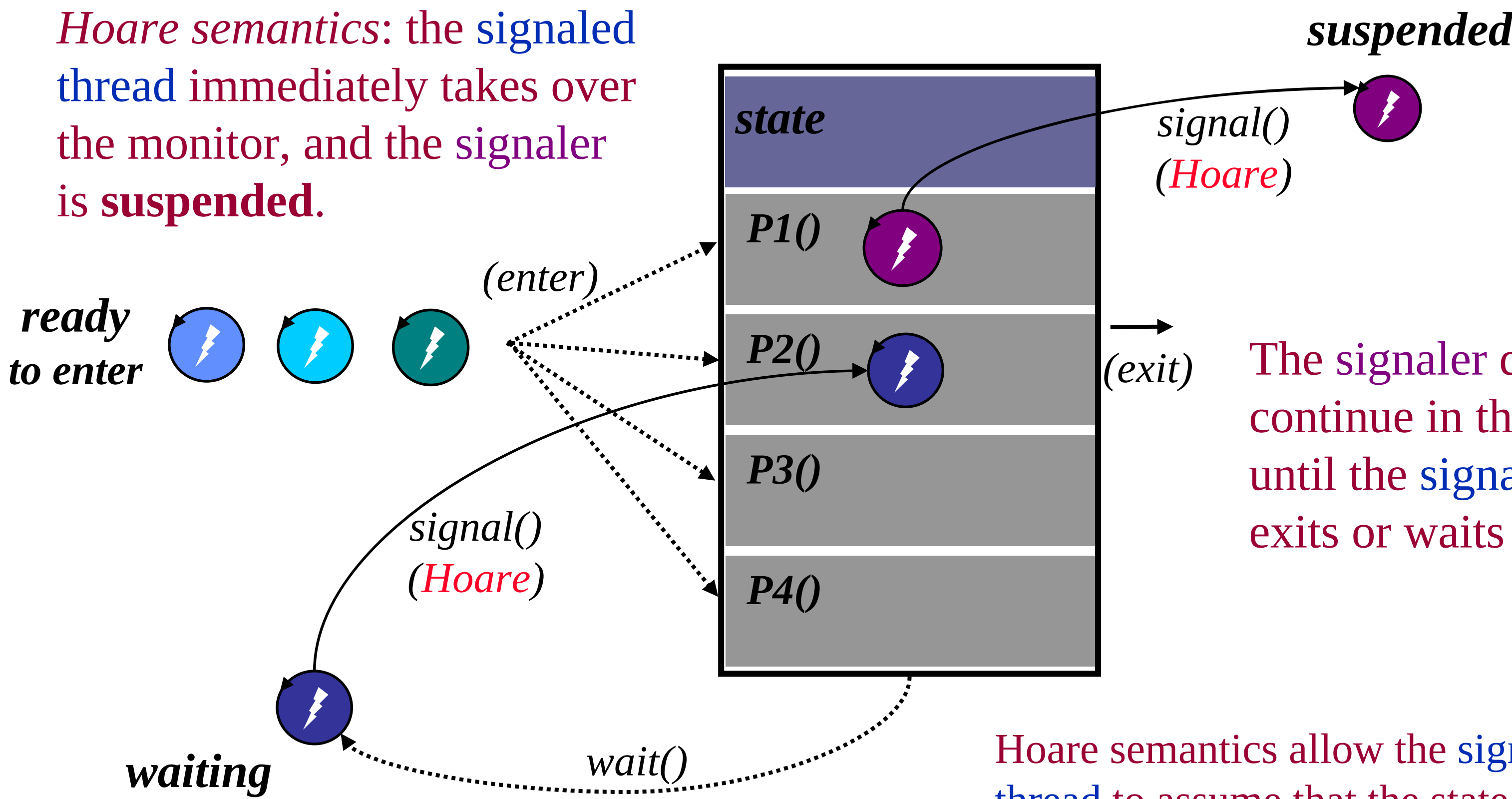


The signaler does not continue in the monitor until the signaled thread exits or waits again.

Hoare Semantics

Suppose purple signals blue in the previous example.

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



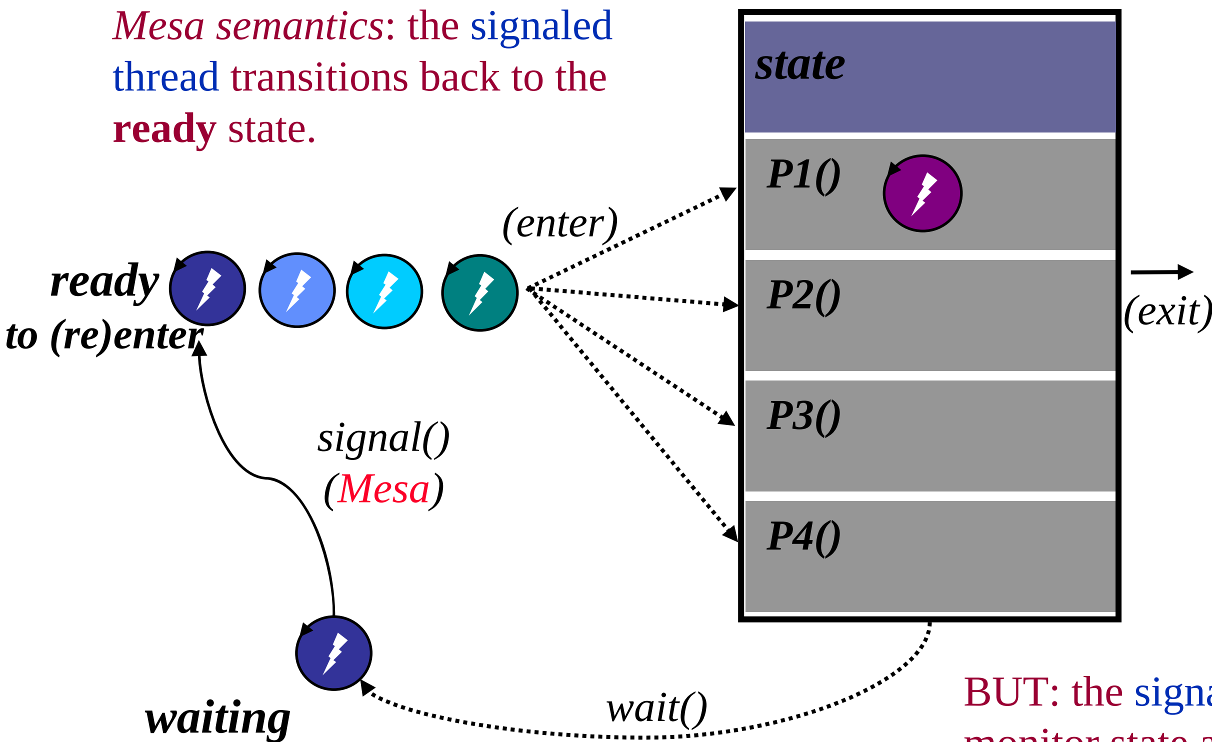
The signaler does not continue in the monitor until the signaled thread exits or waits again.

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 Java) are based on monitors, but they are more flexible.

- A object with its 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: the critical (synchronized) 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.
- Java uses *Mesa semantics* for its condition variables: *loop before you leap*!

Annotated Condition Variable Example

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

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

void awake() {
        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 lock/mutex allows threads to safely manage state related to the sleep/wakeup coordination (e.g., *waiters* count).

SharedLock: Reader/Writer Lock

A reader/write lock or *SharedLock* is a new kind of "lock" that is similar to our old definition:

- supports *Acquire* and *Release* primitives
- guarantees mutual exclusion when a writer is present

<u>But</u>: a *SharedLock* provides better concurrency for readers when no writer is present.

often used in database systems

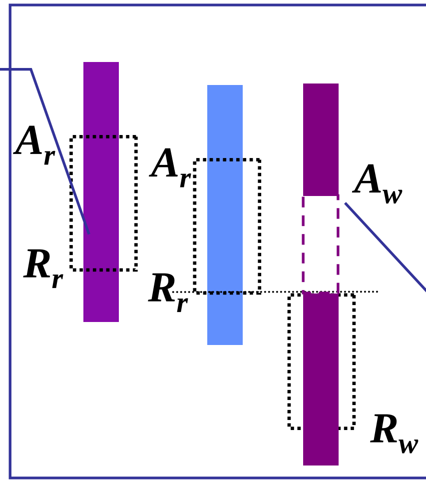
easy to implement using mutexes and condition variables

a classic synchronization problem

```
class SharedLock {
	AcquireRead();   /* shared mode */
	AcquireWrite();  /* exclusive mode */
	ReleaseRead();
	ReleaseWrite();
}
```

Reader/Writer Lock Illustrated

Multiple readers may hold the lock concurrently in **shared** mode.



If each thread acquires the lock in **exclusive** (*write) mode, *SharedLock* functions exactly as an ordinary mutex.

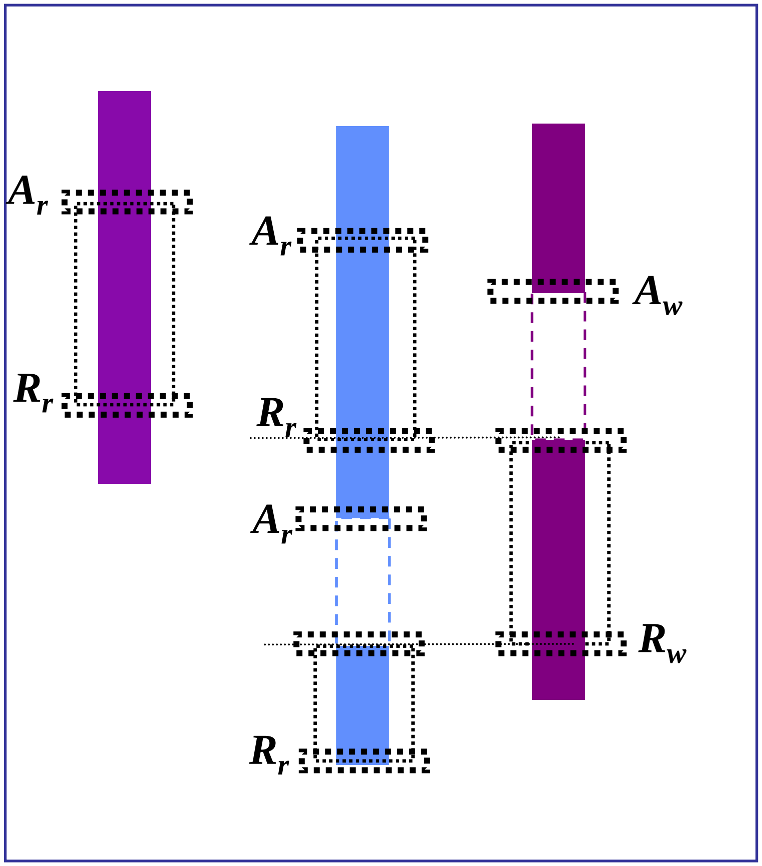
Writers always hold the lock in **exclusive** mode, and must wait for all readers or writer to exit.

mode	read	write	max allowed
shared	yes	no	many
exclusive	yes	yes	one
not holder	no	no	many

Reader/Writer Lock: First Cut

```
int i;          /* # active readers, or -1 if writer */
Lock rwMx;
Condition rwCv;

SharedLock::AcquireWrite() {
    rwMx.Acquire();
    while (i != 0)
        rwCv.Wait(&rwMx);
    i = -1;
    rwMx.Release();
}
SharedLock::AcquireRead() {
    rwMx.Acquire();
    while (i < 0)
        rwCv.Wait(&rwMx);
    i += 1;
    rwMx.Release();
}

SharedLock::ReleaseWrite() {
    rwMx.Acquire();
    i = 0;
    rwCv.Broadcast();
    rwMx.Release();
}

SharedLock::ReleaseRead() {
    rwMx.Acquire();
    i -= 1;
    if (i == 0)
        rwCv.Signal();
    rwMx.Release();
}
```

The Little Mutex Inside SharedLock



Limitations of the SharedLock Implementation

This implementation has weaknesses discussed in [Birrell89].

- *spurious lock conflicts* (on a multiprocessor): multiple waiters contend for the mutex after a signal or broadcast.

  Solution: drop the mutex before signaling.

  (If the signal primitive permits it.)

- *spurious wakeups*

  ReleaseWrite awakens writers as well as readers.

  Solution: add a separate condition variable for writers.

- *starvation*

  How can we be sure that a waiting writer will *ever* pass its acquire if faced with a continuous stream of arriving readers?

Reader/Writer Lock: Second Try

```
SharedLock::AcquireWrite() {
    rwMx.Acquire();
    while (i != 0)
        wCv.Wait(&rwMx);
    i = -1;
    rwMx.Release();
}

SharedLock::AcquireRead() {
    rwMx.Acquire();
    while (i < 0)
        ...rCv.Wait(&rwMx);...
    i += 1;
    rwMx.Release();
}
```

```
SharedLock::ReleaseWrite() {
    rwMx.Acquire();
    i = 0;
    if (readersWaiting)
        rCv.Broadcast();
    else
        wcv.Signal();
    rwMx.Release();
}
SharedLock::ReleaseRead() {
    rwMx.Acquire();
    i -= 1;
    if (i == 0)
        wCv.Signal();
    rwMx.Release();
}
```

Starvation

The reader/writer lock example illustrates *starvation*: under load, a writer will be stalled forever by a stream of readers.

- **Example**: a **one-lane bridge or tunnel**.

  Wait for oncoming car to exit the bridge before entering.

  Repeat as necessary.

- **Problem**: a “writer” may never be able to cross if faced with a continuous stream of oncoming “readers”.
- **Solution**: some reader must politely stop before entering, even though it is not forced to wait by oncoming traffic.

  Use extra synchronization to control the lock scheduling policy.

  Complicates the implementation: optimize only if necessary.

Deadlock

Deadlock is closely related to starvation.

- Processes wait forever for each other to wake up and/or release resources.
- *Example: **traffic gridlock**.*

The difference between deadlock and starvation is subtle.

- With starvation, there always exists a schedule that feeds the starving party.

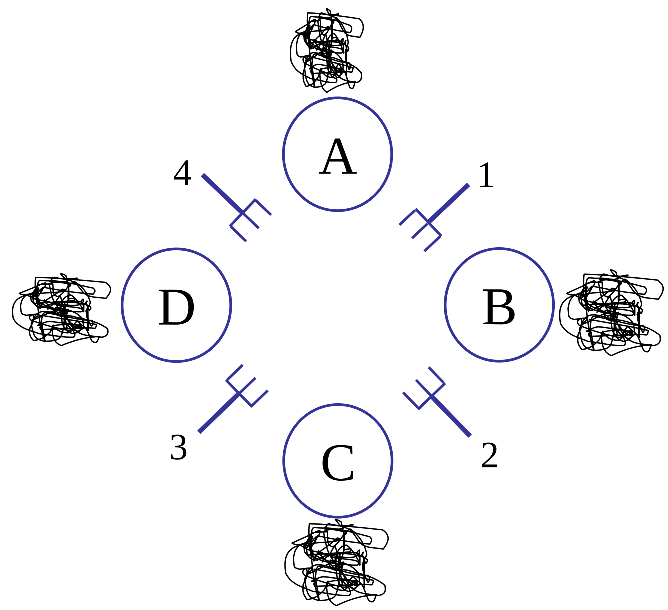
  The situation may resolve itself…if you're lucky.

- Once deadlock occurs, it cannot be resolved by any possible future schedule.

  …though there may exist schedules that *avoid* deadlock.

Dining Philosophers

- *N* processes share *N* resources
- resource requests occur in pairs
- random think times
- hungry philosopher grabs a fork
- ...and doesn't let go
- ...until the other fork is free
- ...and the linguine is eaten




```
while(true) {
    Think();
    AcquireForks();
    Eat();
    ReleaseForks();
}
```

Resource Graphs

Deadlock is easily seen with a *resource graph* or *wait-for graph.*

The graph has a vertex for each process and each resource.

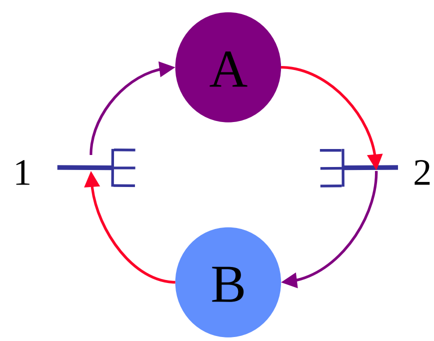
If process ***A*** holds resource ***R***, add an arc from ***R*** to ***A***.

If process ***A*** is waiting for resource ***R***, add an arc from ***A*** to ***R***.

The system is deadlocked iff the wait-for graph has at least one cycle.

S*n*

A grabs fork ***1*** and
waits for fork ***2***.



B grabs fork ***2*** and
waits for fork ***1***.

assign
request

Not All Schedules Lead to Collisions

The scheduler chooses a path of the executions of the threads/processes competing for resources.

Synchronization constrains the schedule to avoid illegal states.

Some paths “just happen” to dodge dangerous states as well.

What is the probability that philosophers will deadlock?

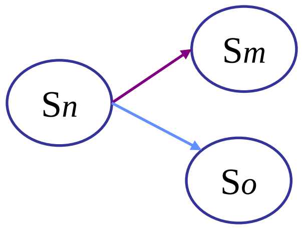
- How does the probability change as:

  think times increase?

  number of philosophers increases?

Resource Trajectory Graphs

Resource trajectory graphs (RTG) depict the scheduler's "random walk" through the space of possible system states.



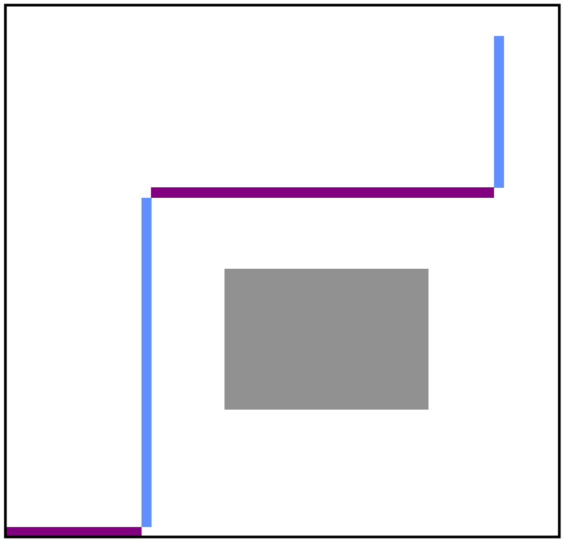
RTG for N processes is N-dimensional.

Process *i* advances along axis *I*.

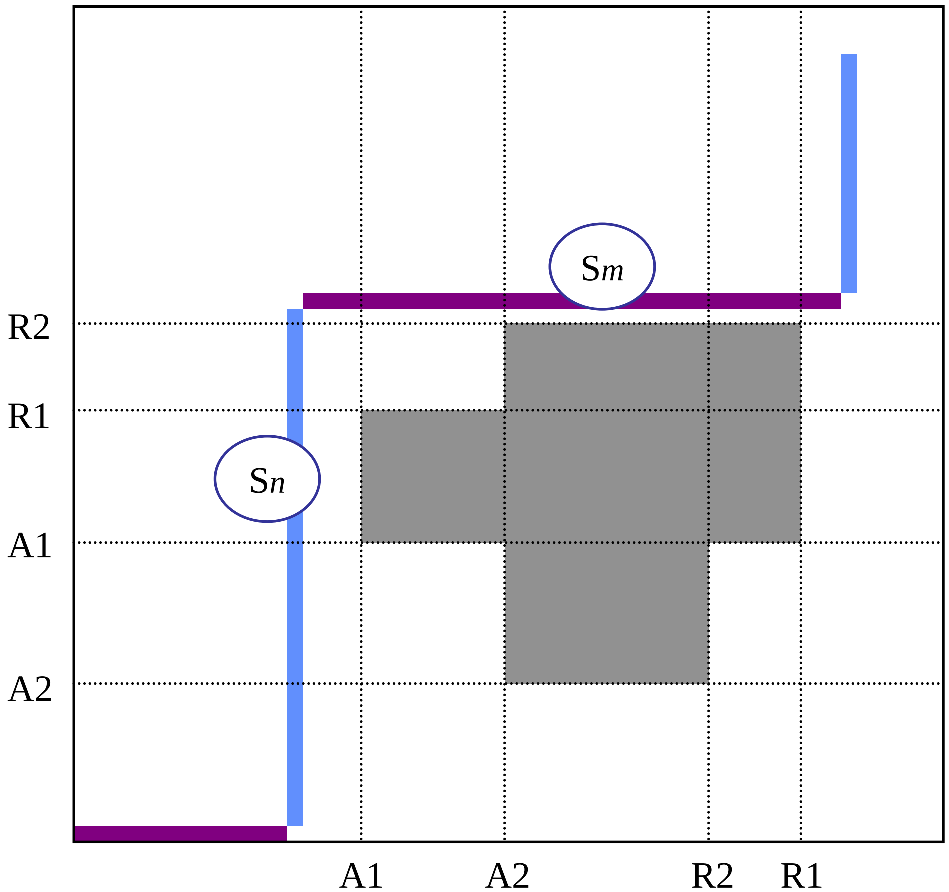
Each point represents one state in the set of all possible system states.

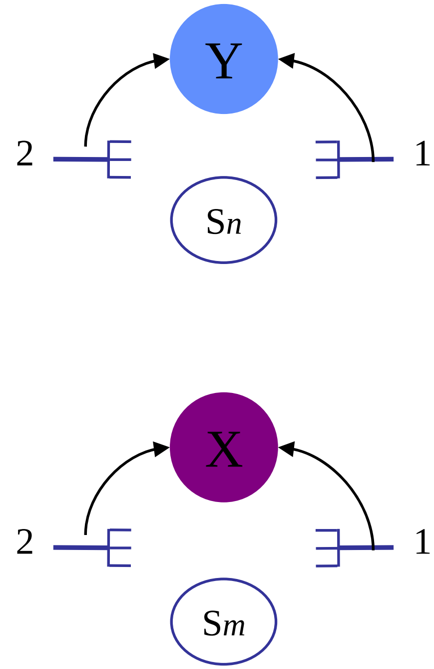
cross-product of the possible states of all processes in the system

(But not all states in the cross-product are legally reachable.)



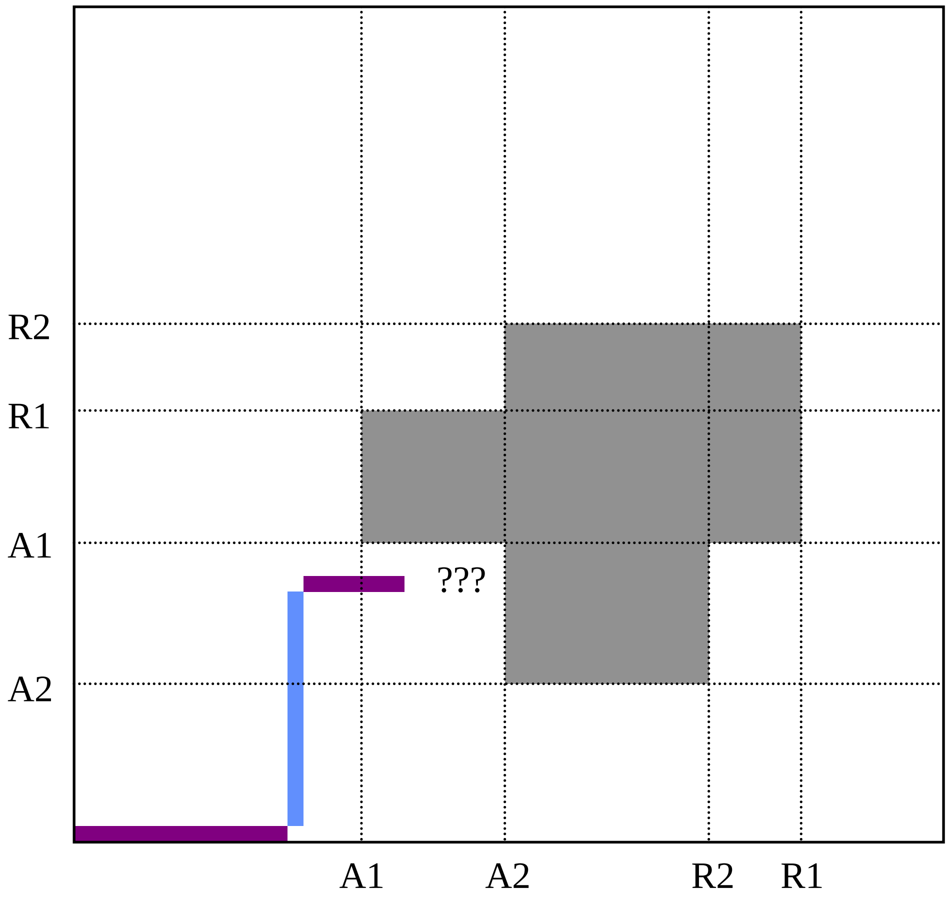
RTG for Two Philosophers

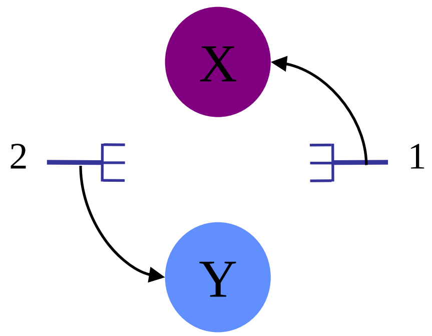




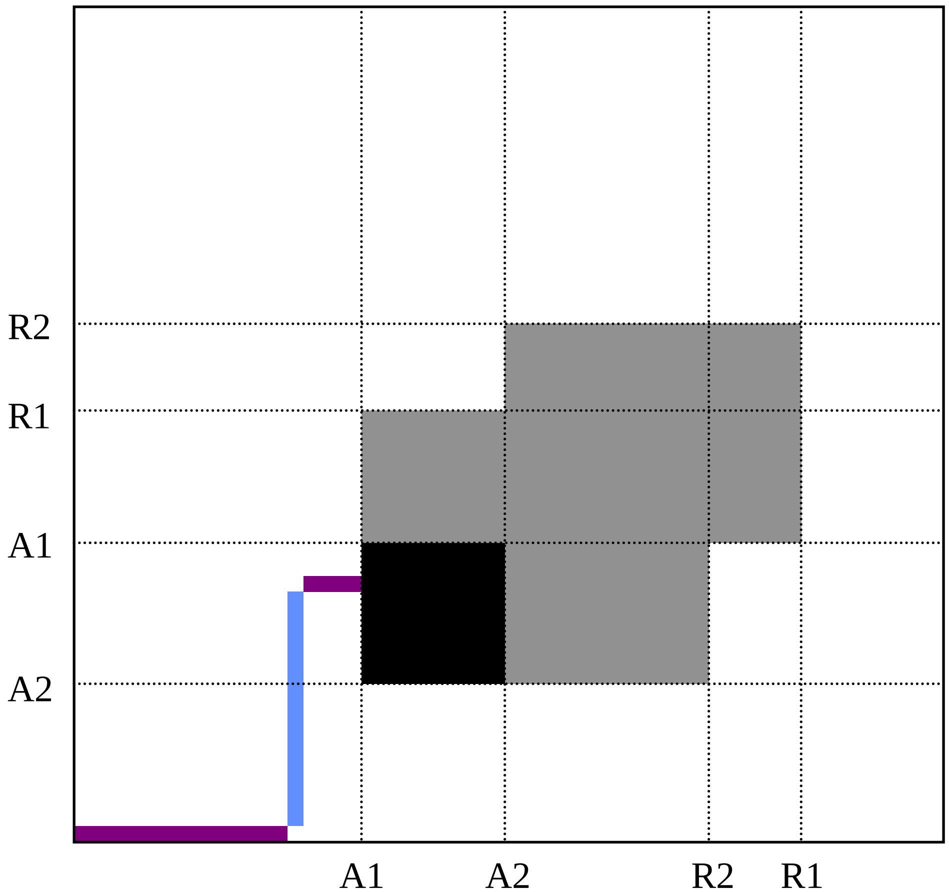
(There are really only 9 states we care about: the important transitions are allocate and release events.)

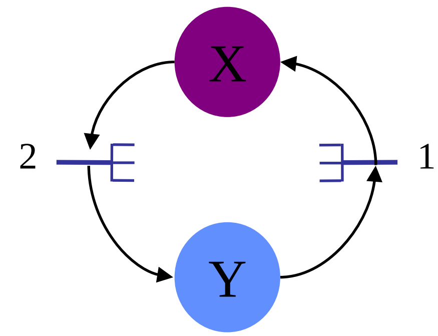
Two Philosophers Living Dangerously





The Inevitable Result





no legal transitions out of this *deadlock state*

Four Preconditions for Deadlock

Four conditions must be present for *deadlock* to occur:

1. *Non-preemption*. Resource ownership (e.g., by threads) is *non-preemptable*.

   Resources are never taken away from the holder.

2. *Exclusion*. Some thread cannot acquire a resource that is held by another thread.

3. *Hold-and-wait*. Holder blocks awaiting another resource.

4. *Circular waiting*. Threads acquire resources out of order.

Dealing with Deadlock

1. *Ignore it*. "How big can those black boxes be anyway?"
2. *Detect it and recover*. Traverse the resource graph looking for cycles before blocking any customer.
   - If a cycle is found, **preempt**: force one party to release and restart.
3. *Prevent it* statically by breaking one of the preconditions.
   - Assign a fixed *partial ordering* to resources; acquire in order.
   - Use locks to reduce multiple resources to a single resource.
   - Acquire resources in advance of need; release all to retry.
4. *Avoid it* dynamically by denying some resource requests.

   Banker's algorithm

Extending the Resource Graph Model

Reasoning about deadlock in real systems is more complex than the simple resource graph model allows.

- Resources may have multiple instances (e.g., memory).
  - Cycles are necessary but not sufficient for deadlock.
  - For deadlock, each resource node with a request arc in the cycle must be fully allocated and unavailable.
- Processes may block to await *events* as well as resources.
  - E.g., ***A*** and ***B*** each rely on the other to wake them up for class.
  - These "logical" producer/consumer resources can be considered to be available as long as the producer is still active.
    - Of course, the producer may not produce as expected.

Reconsidering Threads

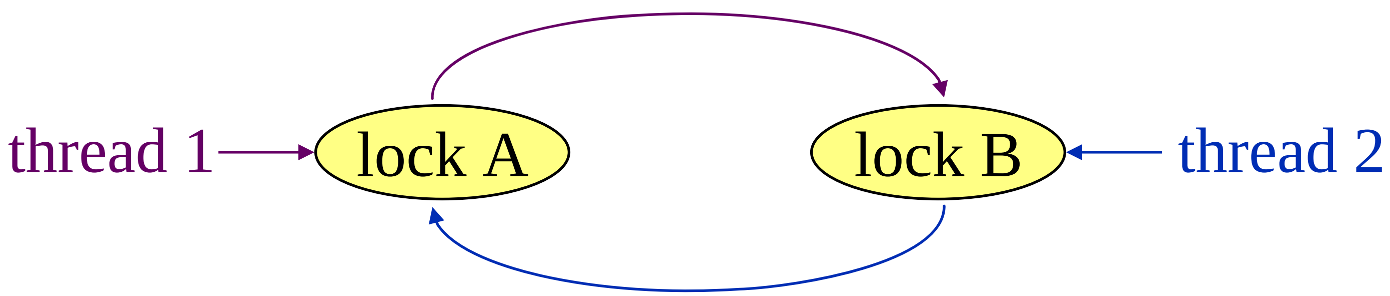
Why Threads Are Hard

Synchronization:

- Must coordinate access to shared data with locks.
- Forget a lock? Corrupted data.

Deadlock:

- Circular dependencies among locks.
- Each process waits for some other process: system hangs.



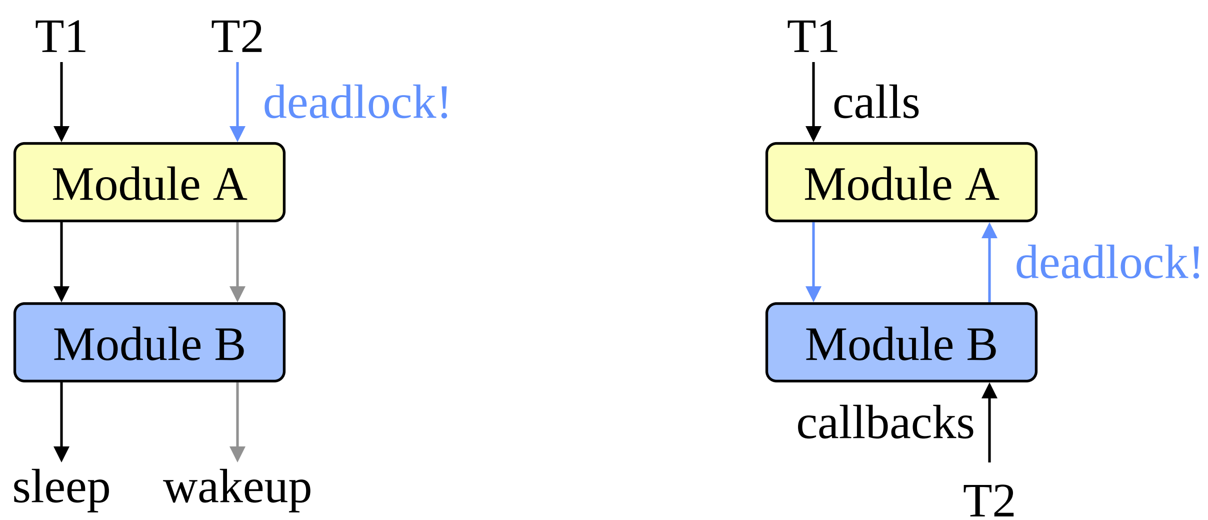
[Ousterhout 1995]

Why Threads Are Hard, cont'd

Hard to debug: **data dependencies, timing dependencies**.

Threads break abstraction: **can't design modules independently**.

Callbacks don't work with locks.



[Ousterhout 1995]

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?

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*.

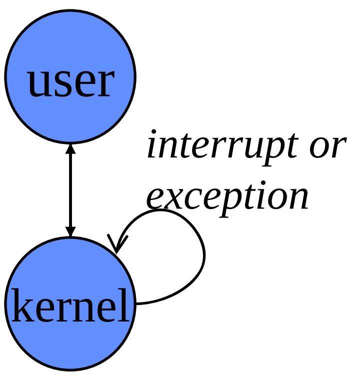
Kernel Concurrency Control 101

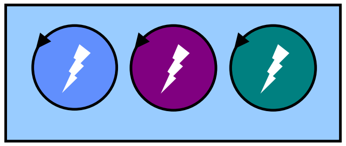
Processes/threads running in kernel mode share access to system data structures in the kernel address space.

- *Sleep/wakeup* (or equivalent) are the basis for:

  coordination, e.g., join (*exit/wait*), timed waits (*pause*), bounded buffer (pipe *read/write*), message *send/receive*

  synchronization, e.g., long-term mutual exclusion for atomic *read*/write** syscalls



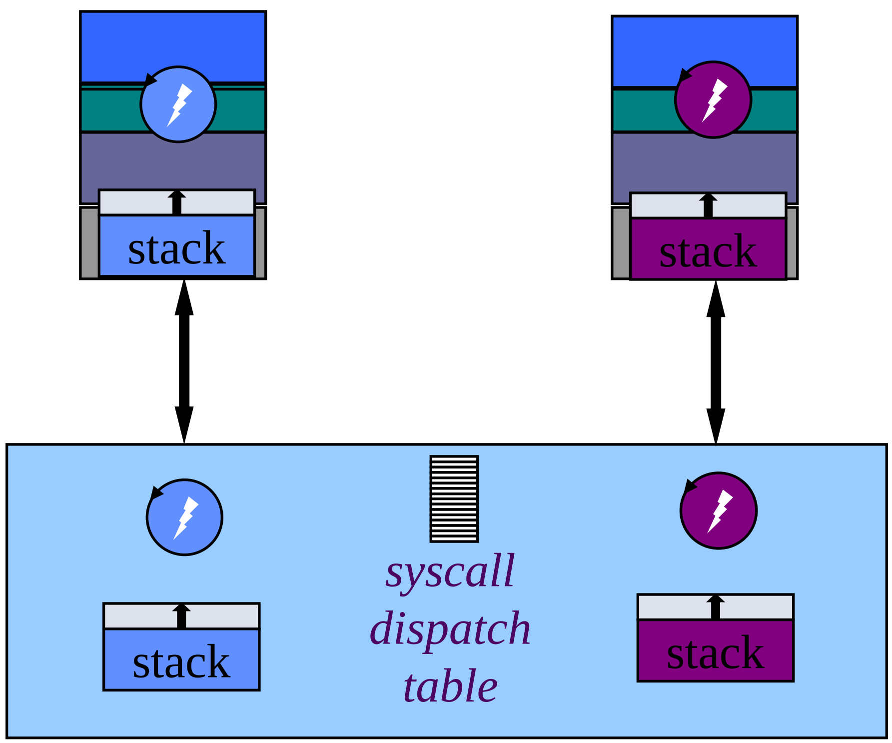


Sleep/wakeup is sufficient for concurrency control among kernel-mode threads on uniprocessors: problems arise from *interrupts* and *multiprocessors*.

Kernel Stacks and Trap/Fault Handling

Processes execute user code on a ***user stack*** in the user portion of the process virtual address space.

Each process has a second ***kernel stack*** in kernel space (the kernel portion of the address space).



System calls and faults run in kernel mode on the process kernel stack.

System calls run in the process space, so *copyin* and *copyout* can access user memory.

The syscall trap handler makes an indirect call through the *system call dispatch* table to the handler for the specific system call.

Mode, Space, and Context

At any time, the state of each processor is defined by:

1. *mode*: given by the mode bit

   Is the CPU executing in the protected kernel or a user program?

2. *space*: defined by V->P translations currently in effect

   What address space is the CPU running in? Once the system is booted, it always runs in some virtual address space.

3. *context*: given by register state and execution stream

   Is the CPU executing a thread/process, or an interrupt handler?

   Where is the stack?

These are important because the mode/space/context determines the meaning and validity of key operations.

Common Mode/Space/Context Combinations

1. *User code* executes in a process/thread context in a process address space, in user mode.

   Can address only user code/data defined for the process, with no access to privileged instructions.

2. *System services* execute in a process/thread context in a process address space, in kernel mode.

   Can address kernel memory or user process code/data, with access to protected operations: may sleep in the kernel.

3. *Interrupts* execute in a system interrupt context in the address space of the interrupted process, in kernel mode.

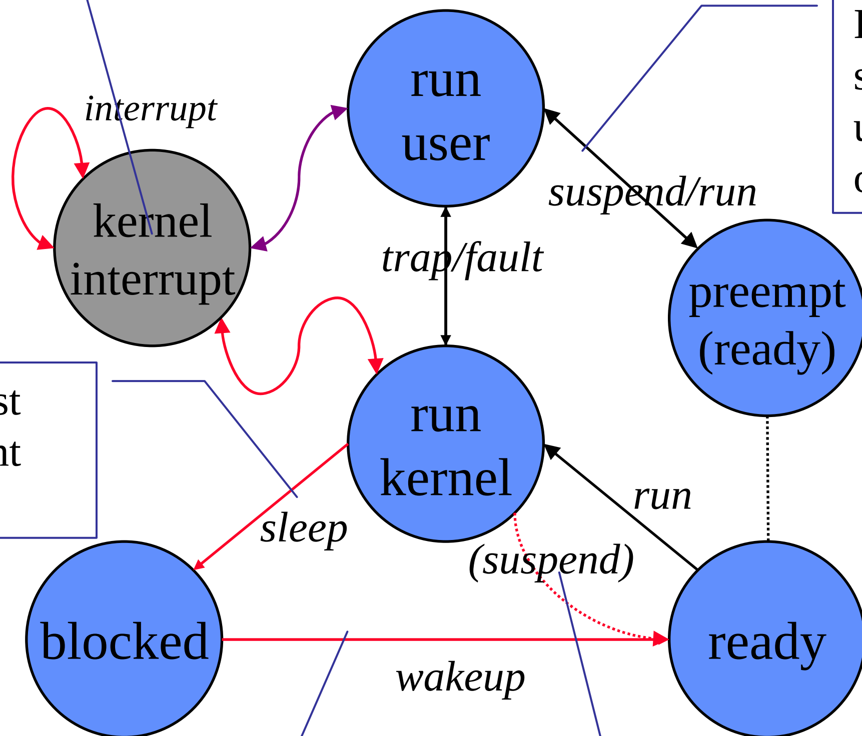
   Can access kernel memory and use protected operations.

   no sleeping!

Dangerous Transitions

Interrupt handlers may share data with syscall code, or with other handlers.

Involuntary context switches of threads in user mode have no effect on kernel data.

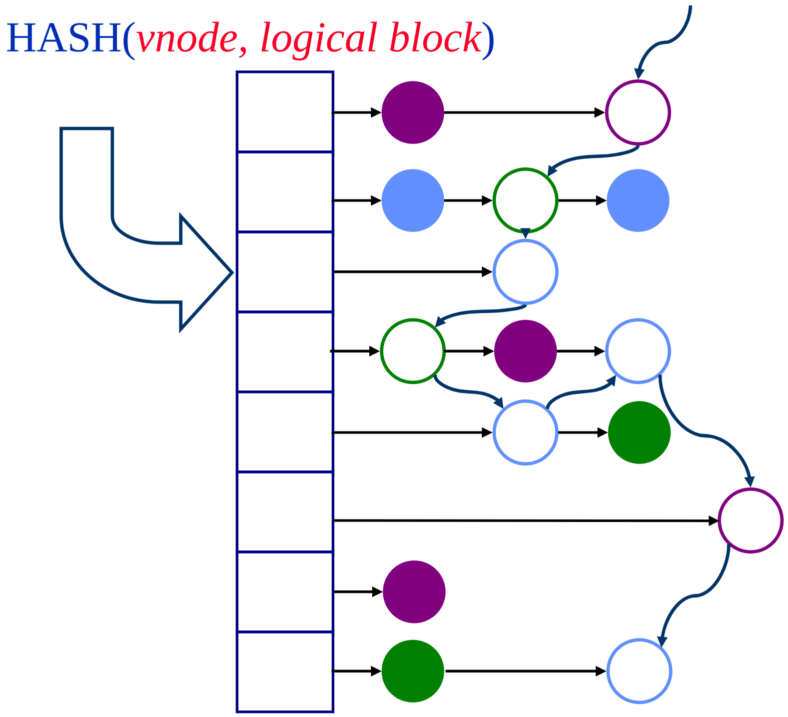


Kernel-mode threads must restore data to a consistent state before blocking.

The shared data states observed by an awakening thread may have changed while sleeping.

Thread scheduling in kernel mode is *non-preemptive* as a policy in classical kernels (but not Linux).

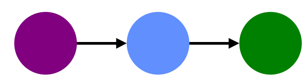
Concurrency Example: Block/Page Buffer Cache



Buffers with valid data are retained in memory in a *buffer cache* or *file cache*.

Each item in the cache is a *buffer header* pointing at a buffer .

Blocks from different files may be intermingled in the hash chains.



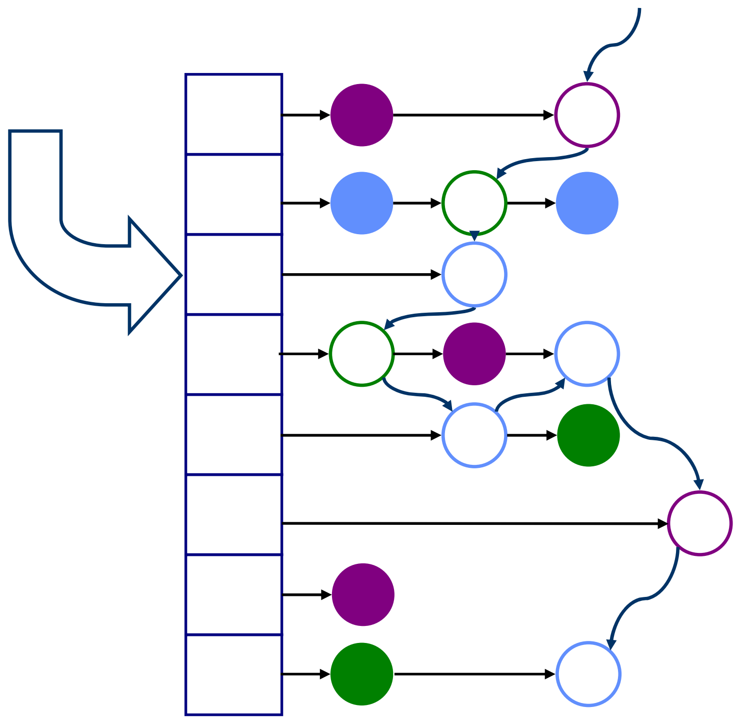
System data structures hold pointers to buffers only when I/O is pending or imminent.

- *busy bit* instead of refcount
- most buffers are “free”

Most systems use a pool of buffers in kernel memory as a staging area for memory<->disk transfers.

VM Page Cache Internals

HASH(*memory object/segment, logical block*)



1. Pages in active use are mapped through the page table of one or more processes.

2. On a fault, the global object/offset hash table in kernel finds pages brought into memory by other processes.

3. Several page queues wind through the set of active frames, keeping track of usage.

4. Pages selected for eviction are removed from all page tables first.

Kernel Object Handles

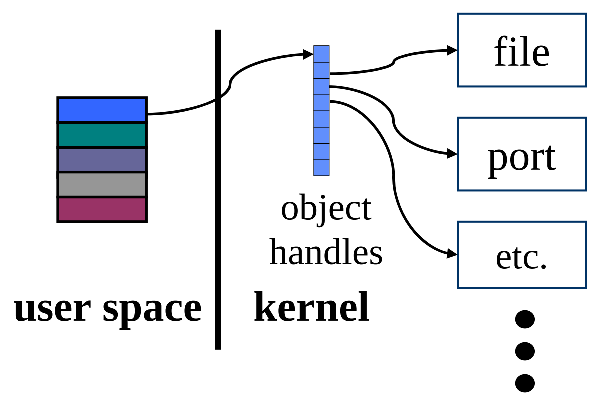
Instances of kernel abstractions may be viewed as "objects" named by protected *handles* held by processes.

- Handles are obtained by *create/open* calls, subject to security policies that grant specific rights for each handle.
- Any process with a handle for an object may operate on the object using operations (system calls).

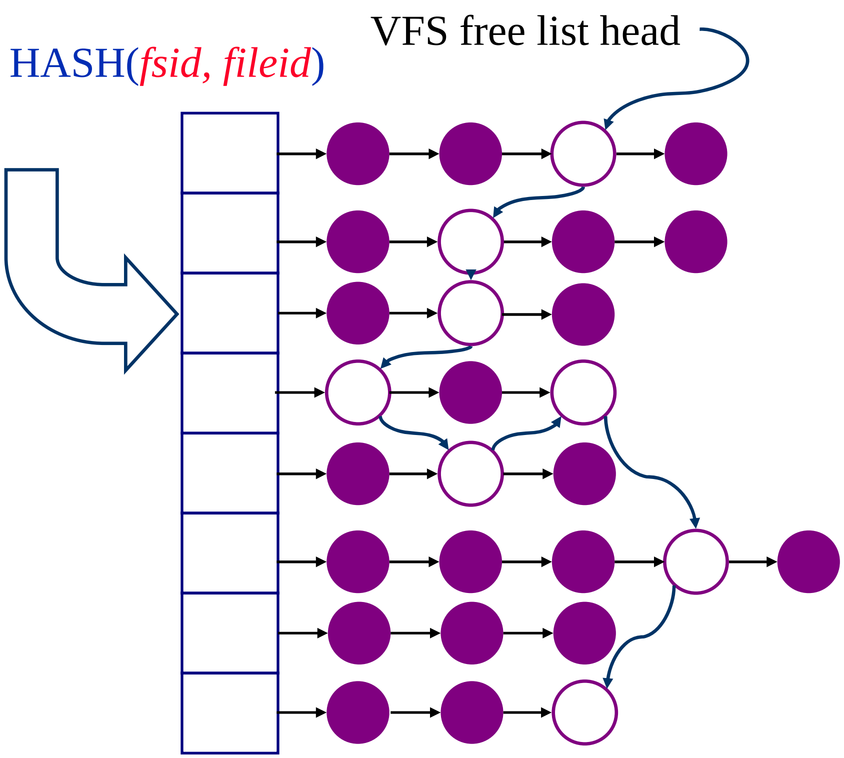
  Specific operations are defined by the object's type.

- The handle is an integer index to a kernel table.

Microsoft NT object handles
Unix file descriptors



V/Inode Cache



vget(vp): reclaim cached inactive vnode from VFS free list
vref(vp): increment reference count on an active vnode
vrele(vp): release reference count on a vnode
vgone(vp): vnode is no longer valid (file is removed)

Active vnodes are *reference- counted* by the structures that hold pointers to them.

- system open file table
- process current directory
- file system mount points
- etc.

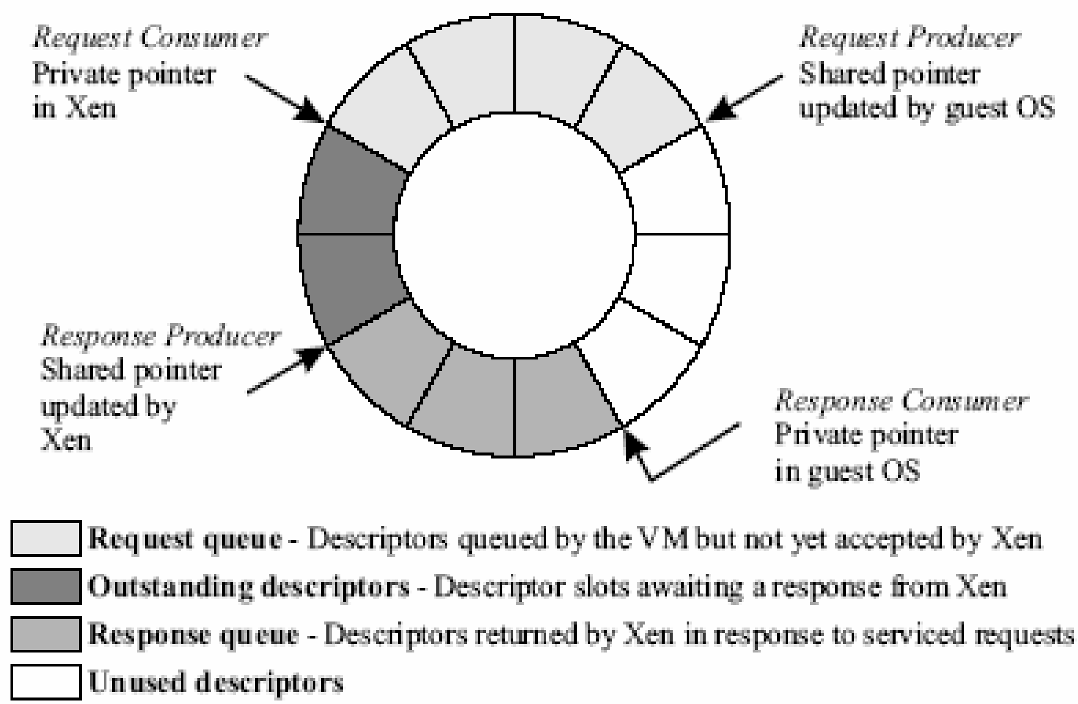
Each specific file system maintains its own hash of vnodes (BSD).

- specific FS handles initialization
- free list is maintained by VFS

Device I/O Management in Xen

Data transfer to and from domains through buffer descriptor ring

- producer/consumer
- decouples data transfer an event notification
- Reordering allowed



The Problem of Interrupts

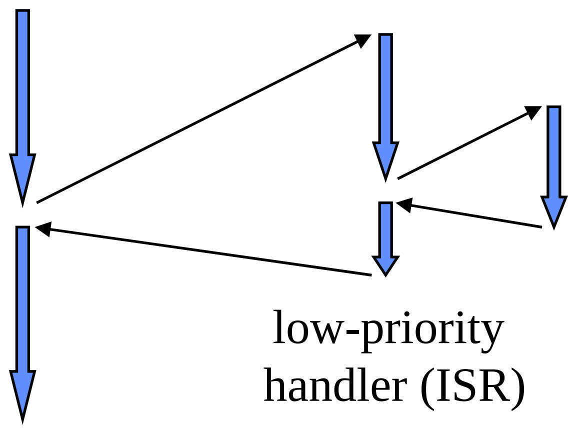
Interrupts can cause races if the handler (ISR) shares data with the interrupted code.

e.g., *wakeup* call from an ISR may corrupt the sleep queue.

Interrupts may be nested.

ISRs may race with each other.

kernel code
(e.g., syscall)



high-priority
ISR

low-priority
handler (ISR)

Interrupt Priority

Classical Unix kernels illustrate the basic approach to avoiding interrupt races.

- Rank interrupt types in *N priority classes.*
- When an ISR at priority *p* runs, CPU blocks interrupts of priority *p* or lower.

  How big must the interrupt stack be?

- Kernel software can query/raise/lower the CPU *interrupt priority level* (IPL).

  Avoid races with an ISR of higher priority by raising CPU IPL to that priority.

  Unix *spl*/splx* primitives (may need software support on some architectures).

spl0	low
splnet	
splbio	
splimp	
clock	high

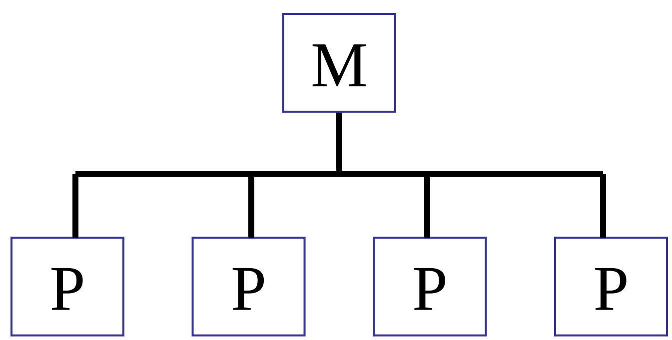
splx(s)

```
int s;
s = splhigh();
/* touch sleep queues */
splx(s);
```

Multiprocessor Kernels

On a shared memory multiprocessor, non-preemptive kernel code and *spl*()* are no longer sufficient to prevent races.

- **Option 1**, *asymmetric multiprocessing*: limit all handling of traps and interrupts to a single processor.
  - slow and boring
- **Option 2**, *symmetric multiprocessing* (“SMP”): supplement existing synchronization primitives.
  - any CPU may execute kernel code
  - synchronize with spin-waiting
  - requires atomic instructions
  - use *spinlocks…*
  - *…but still must disable interrupts*



Example: Unix Sleep (BSD)

```
sleep (void* event, int sleep_priority)
{
        struct proc *p = curproc;
        int s;

        s = splhigh();                    /* disable all interrupts */
        p->p_wchan = event;               /* what are we waiting for */
        p->p_priority -> priority;        /* wakeup scheduler priority */
        p->p_stat = SSLEEP;               /* transition curproc to sleep state */
        INSERTQ(&slpque[HASH(event)], p); /* fiddle sleep queue */
        splx(s);                          /* enable interrupts */
        mi_switch();                      /* context switch */
        /* we're back... */
}
```

Illustration Only

Stuff to Know

- Know how to use mutexes, CVs, and semaphores. It is a craft. Learn to think like Birrell: write concurrent code that is clean and obviously correct, and balances performance with simplicity.
- Understand why these abstractions are needed: sleep/wakeup races, missed wakeup, double wakeup, interleavings, critical sections, the adversarial scheduler, multiprocessors, thread interactions, ping-pong.
- Understand the variants of the abstractions: Mesa vs. Hoare semantics, monitors vs. mutexes, binary semaphores vs. counting semaphores, spinlocks vs. blocking locks.
- Understand the contexts in which these primitives are needed, and how those contexts are different: processes or threads in the kernel, interrupts, threads in a user program, servers, architectural assumptions.
- Where should we define/implement synchronization abstractions? Kernel? Library? Language/compiler?
- Reflect on scheduling issues associated with synchronization abstractions: how much should a good program constrain the scheduler? How much should it assume about the scheduling semantics of the primitives?

Note for CPS 196, Spring 2006

In this class we did not talk about semaphores, and the presentation of kernel synchronization was confused enough that I do not plan to test it.

So the remaining slides are provided for completeness.

Implementing Sleep on a Multiprocessor

```
sleep (void* event, int sleep_priority)
{
        struct proc *p = curproc;
        int s;

        s = splhigh();                          /* disable all interrupts */
        p->p_wchan = event;                     /* what are we waiting for */
        p->p_priority -> priority;              /* wakeup scheduler priority */
        p->p_stat = SSLEEP;                     /* transition curproc to sleep state */
        INSERTQ(&slpque[HASH(event)], p);       /* fiddle sleep queue */
        splx(s);                                /* enable interrupts */
        mi_switch();                            /* context switch */
        /* we're back... */
}
```

What if another CPU takes an interrupt and calls *wakeup*?

What if another CPU is handling a syscall and calls *sleep* or *wakeup*?

What if another CPU tries to *wakeup* *curproc* before it has completed *mi_switch*?

Illustration Only

Using Spinlocks in *Sleep*: First Try

```
sleep (void* event, int sleep_priority)
{
        struct proc *p = curproc;
        int s;

        lock spinlock;
        p->p_wchan = event;              /* what are we waiting for */
        p->p_priority -> priority;       /* wakeup scheduler priority */
        p->p_stat = SSLEEP;              /* transition curproc to sleep state */
        INSERTQ(&slpque[HASH(event)], p);        /* fiddle sleep queue */
        unlock spinlock;
        mi_switch();                     /* context switch */
        /* we're back */
}
```

Grab spinlock to prevent another CPU from racing with us.

Wakeup (or any other related critical section code) will use the same spinlock, guaranteeing mutual exclusion.

Illustration Only

Sleep with Spinlocks: What Went Wrong

```
sleep (void* event, int sleep_priority)
{
        struct proc *p = curproc;
        int s;

        lock spinlock;
        p->p_wchan = event;             /* what are we waiting for */
        p->p_priority -> priority;      /* wakeup scheduler priority */
        p->p_stat = SSLEEP;             /* transition curproc to sleep state */
        INSERTQ(&slpque[HASH(event)], p);       /* fiddle sleep queue */
        unlock spinlock;
        mi_switch();                    /* context switch */
        /* we're back */
}
```

Potential *deadlock*: what if we take an interrupt on this processor, and call *wakeup* while the lock is held?

Potential doubly scheduled thread: what if another CPU calls *wakeup* to wake us up before we're finished with *mi_switch* on this CPU?

Illustration Only

Using Spinlocks in *Sleep*: Second Try

```
sleep (void* event, int sleep_priority)
{
        struct proc *p = curproc;
        int s;

        s = splhigh();
        lock spinlock;

        p->p_wchan = event;                 /* what are we waiting for */
        p->p_priority -> priority;          /* wakeup scheduler priority */
        p->p_stat = SSLEEP;                 /* transition curproc to sleep state */
        INSERTQ(&slpque[HASH(event)], p);          /* fiddle sleep queue */

        unlock spinlock;
        splx(s);

        mi_switch();            /* context switch */
        /* we're back */
}
```

Grab spinlock *and* disable interrupts.

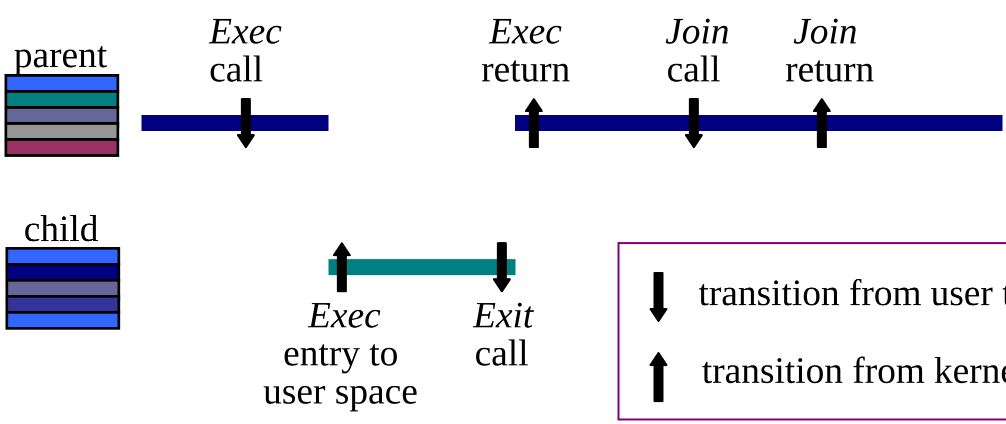
Illustration Only

Mode Changes for Exec/Exit

Syscall traps and “returns” are not always paired.

- *Exec* “returns” (to child) from a trap that “never happened”
- *Exit* system call trap never returns
- system may switch processes between trap and return

In contrast, interrupts and returns are strictly paired.

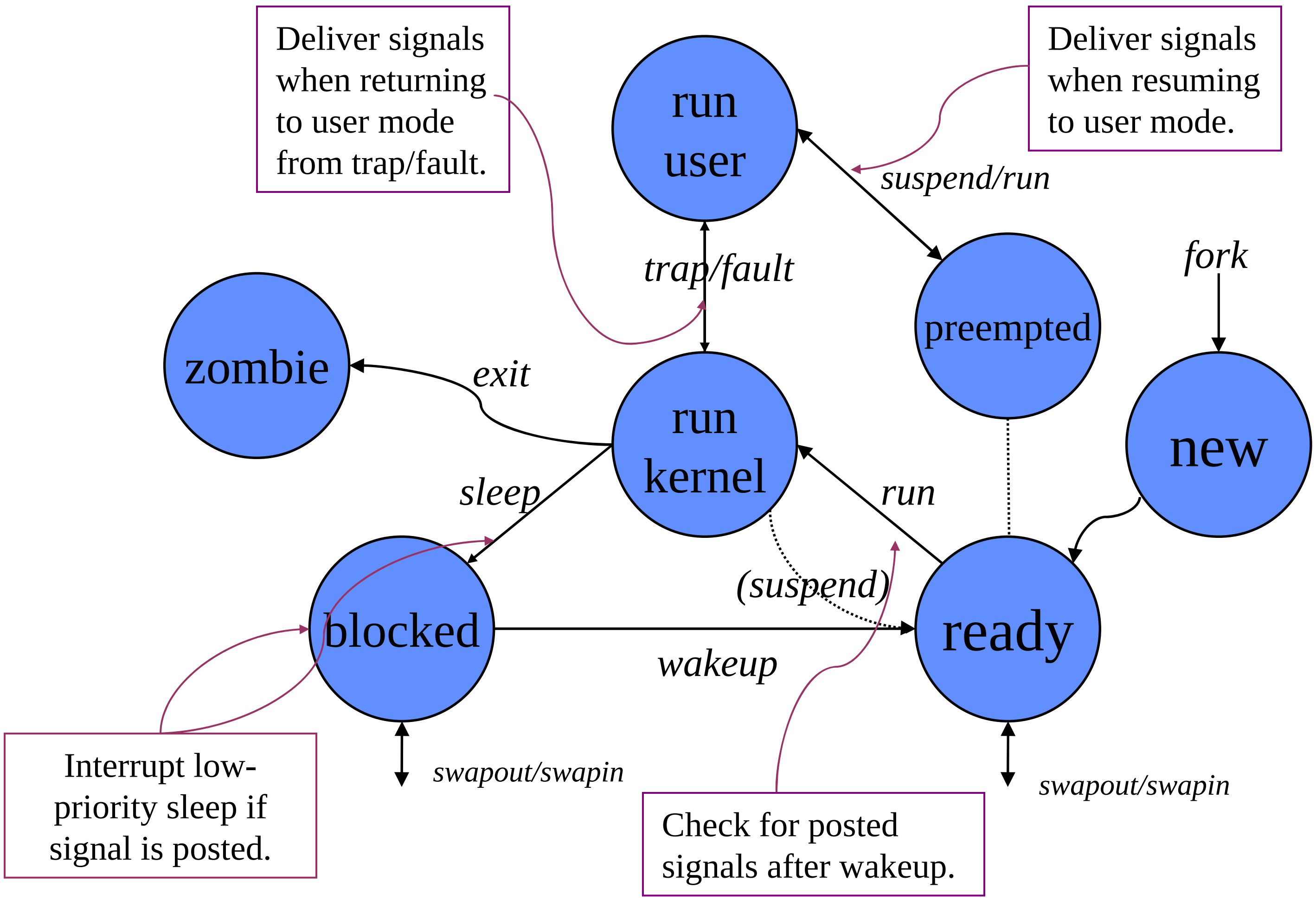


Exec *enters the child by doctoring up a saved user context to “return” through.*

↓ transition from user to kernel mode (*callsys*)

↑ transition from kernel to user mode (*retsys*)

When to Deliver Signals?



Implementing Spinlocks: First Cut

```
class Lock {
            int held;
}

void Lock::Acquire() {
            while (held);     "busy-wait" for lock holder to release
            held = 1;
}

void Lock::Release() {
            held = 0;
}
```

Spinlocks: What Went Wrong

Race to acquire: two threads could observe *held == 0* concurrently, and think they both can acquire the lock.

```
void Lock::Acquire() {
        while (held);          /* test */
        held = 1;              /* set */
}

void Lock::Release() {
        held = 0;
}
```

What Are We Afraid Of?

Potential problems with the "rough" spinlock implementation:

(1) races that violate mutual exclusion

- involuntary context switch between **test** and **set**
- on a multiprocessor, race between **test** and **set** on two CPUs

(2) wasteful spinning

- lock holder calls **sleep** or **yield**
- interrupt handler acquires a busy lock
- involuntary context switch for lock holder

Which are implementation issues, and which are problems with spinlocks themselves?

The Need for an Atomic "Toehold"

To implement safe mutual exclusion, we need support for some sort of "magic toehold" for synchronization.

- The lock primitives themselves have critical sections to test and/or set the lock flags.
- These primitives must somehow be made *atomic*.

  uninterruptible

  a sequence of instructions that executes "all or nothing"

- Two solutions:

  (1) hardware support: *atomic instructions* (**test-and-set**)

  (2) scheduler control: *disable timeslicing* (**disable interrupts**)

Atomic Instructions: Test-and-Set

load
test
store

load
test
store

Problem: interleaved load/test/store.

Solution: TSL atomically sets the flag and leaves the old value in a register.

```
Spinlock::Acquire () {
        while(held);
        held = 1;
}
```

Wrong

```
      load    4(SP), R2             ; load "this"
busywait:
      load    4(R2), R3             ; load "held" flag
      bnz     R3, busywait          ; spin if held wasn't zero
      store   #1, 4(R2)             ; held = 1
```

Right

```
      load    4(SP), R2             ; load "this"
busywait:
      tsl     4(R2), R3             ; test-and-set this->held
      bnz     R3,busywait           ; spin if held wasn't zero
```

Implementing Locks: Another Try

```
class Lock {
}

void Lock::Acquire() {
            disable interrupts;
}

void Lock::Release() {
            enable interrupts;
}
```

Problems?

Implementing Mutexes: Rough Sketch

```
class Lock {
        int held;
        Thread* waiting;
}

void Lock::Acquire() {
        if (held) {
                waiting = currentThread;
                currentThread->Sleep();
        }
        held = 1;
}

void Lock::Release() {
        held = 0;
        if (waiting)        /* somebody's waiting: wake up */
                scheduler->ReadyToRun(waiting);
}
```

Implementing Mutexes: A First Cut

```
class Lock {
        int held;
        List sleepers;
}

void Lock::Acquire() {
        while (held) {                         Why the while loop?
                sleepers.Append((void*)currentThread);
                currentThread->Sleep();
        }
        held = 1;                              Is this safe?
}

void Lock::Release() {
        held = 0;
        if (!sleepers->IsEmpty())        /* somebody's waiting: wake up */
                scheduler->ReadyToRun((Thread*)sleepers->Remove());
}
```

Mutexes: What Went Wrong

Potential *missed wakeup*: holder could *Release* before thread is on sleepers list.

Potential *corruption* of *sleepers* list in a race between two *Acquires* or an *Acquire* and a *Release*.

Potential *missed wakeup*: holder could call to wake up before we are "fully asleep".

```
void Lock::Acquire() {
        while (held) {
                sleepers.Append((void*)currentThread);
                currentThread->Sleep();
        }
        held = 1;
}

void Lock::Release() {
        held = 0;
        if (!sleepers->IsEmpty())        /* somebody's waiting: wake up */
                scheduler->ReadyToRun((Thread*)sleepers->Remove());
}
```

Race to acquire: two threads could observe *held == 0* concurrently, and think they both can acquire the lock.

Using Sleep/Wakeup Safely

```
Thread* waiter = 0;

void await() {
        disable interrupts
        waiter = currentThread;                 /* "I'm sleeping" */
        currentThread->Sleep();                 /* sleep */
        enable interrupts
}

void awake() {
        disable interrupts
        if (waiter)                             /* wakeup */
                scheduler->ReadyToRun(waiter);
        waiter = (Thread*)0;                    /* "you're awake" */
        enable interrupts
}
```

Disabling interrupts prevents a context switch between “I’m sleeping” and “sleep”.

Nachos *Thread::Sleep* *requires* disabling interrupts.

Disabling interrupts prevents a context switch between “wakeup” and “you’re awake”.
Will this work on a multiprocessor?

What to Know about Sleep/Wakeup

1. *Sleep/wakeup* primitives are the fundamental basis for *all* blocking synchronization.
2. All use of *sleep/wakeup* requires some additional low-level mechanism to avoid missed and double wakeups.
   - disabling interrupts, and/or
   - constraints on preemption, and/or *(Unix kernels use this instead of disabling interrupts)*
   - spin-waiting *(on a multiprocessor)*
3. These low-level mechanisms are tricky and error-prone.
4. High-level synchronization primitives take care of the details of using *sleep/wakeup*, hiding them from the caller.
   - semaphores, mutexes, condition variables

Semaphores

Semaphores handle all of your synchronization needs with one elegant but confusing abstraction.

- controls allocation of a resource with multiple instances
- a non-negative integer with special operations and properties
  - initialize to arbitrary value with *Init* operation
  - "souped up" increment (*Up* or *V*) and decrement (*Down* or *P*)
- atomic sleep/wakeup behavior implicit in ***P*** and ***V***
  - ***P*** does an atomic ***sleep***, **if** the semaphore value is zero.
    - ***P*** means "probe"; it cannot decrement until the semaphore is positive.
  - ***V*** does an atomic ***wakeup*.**
  - num(P) <= num(V) + init

Semaphores vs. Condition Variables

1. *Up* 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!
   - *Up* has the same effect whether or not a thread is waiting.
     Semaphores retain a "memory" of calls to *Up*.
2. *Down* differs from *Wait* in that:
   - *Down* checks the condition and blocks only if necessary.
     no need to recheck the condition after returning from *Down*
     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

Semaphores using Condition Variables

```
void Down() {
        mutex->Acquire();
        ASSERT(count >= 0);
        while(count == 0)
                condition->Wait(mutex);
        count = count - 1;
        mutex->Release();
}

void Up() {
        mutex->Acquire();
        count = count + 1;
        condition->Signal(mutex);
        mutex->Release();
}
```

(Loop before you leap!)

This constitutes a proof that mutexes and condition variables are at least as powerful as semaphores.

Semaphores as Mutexes

```
semapohore->Init(1);

void Lock::Acquire()
{
        semaphore->Down();
}

void Lock::Release()
{
        semaphore->Up();
}
```

Semaphores must be initialized with a value representing the number of free resources: mutexes are a single-use resource.

Down() to acquire a resource; blocks if no resource is available.

Up() to release a resource; wakes up one waiter, if any.

Up and *Down* are *atomic*.

Mutexes are often called *binary semaphores*.
However, "real" mutexes have additional constraints on their use.

Ping-Pong with Semaphores

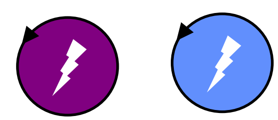
```
blue->Init(0);
purple->Init(1);
```

```
void
PingPong() {
      while(not done) {
            blue->P();
            Compute();
            purple->V();
      }
}
```

```
void
PingPong() {
      while(not done) {
            purple->P();
            Compute();
            blue->V();
      }
}
```

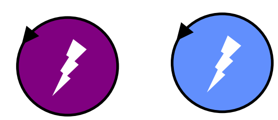
Ping-Pong with One Semaphore?

```
sem->Init(0);
blue:   { sem->P(); PingPong(); }
purple: { PingPong(); }

void
PingPong() {
        while(not done) {
            Compute();
            sem->V();
            sem->P();
        }
}
```



Ping-Pong with One Semaphore?

sem->Init(0);
blue: { sem->P(); PingPong(); }
purple: { PingPong(); }



void
PingPong() {
    while(*not done*) {
        Compute();
        sem->V();
        sem->P();
    }
}

Nachos semaphores have Mesa-like semantics:
They do not guarantee that a waiting thread wakes up "in time" to consume the count added by a *V().*
- semaphores are not "fair"
- no count is "reserved" for a waking thread
- uses "passive" vs. "active" implementation

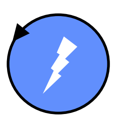
Another Example With Dual Semaphores

```
blue->Init(0);
purple->Init(0);

void Blue() {
    while(not done) {
        Compute();
        purple->V();
        blue->P();
    }
}

void Purple() {
    while(not done) {
        Compute();
        blue->V();
        purple->P();
    }
}
```

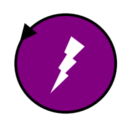
Basic Barrier

```
blue->Init(0);
purple->Init(0);
```

```
void
IterativeCompute() {
    while(not done) {
        Compute();
        purple->V();
        blue->P();
    }
}
```



```
void
IterativeCompute() {
    while(not done) {
        Compute();
        blue->V();
        purple->P();
    }
}
```



How About This? (#1)

```
blue->Init(1);
purple->Init(1);
```

```
void
IterativeCompute?() {
        while(not done) {
                blue->P();
                Compute();
                purple->V();
        }
}
```

```
void
IterativeCompute?() {
        while(not done) {
                purple->P();
                Compute();
                blue->V();
        }
}
```

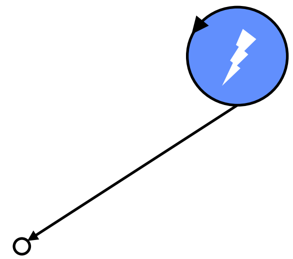
How About This? (#2)

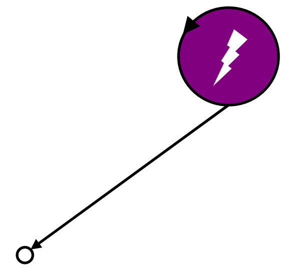
```
blue->Init(1);
purple->Init(0);
```

```
void
IterativeCompute?() {
    while(not done) {
        blue->P();
        Compute();
        purple->V();
    }
}
```

```
void
IterativeCompute?() {
    while(not done) {
        purple->P();
        Compute();
        blue->V();
    }
}
```

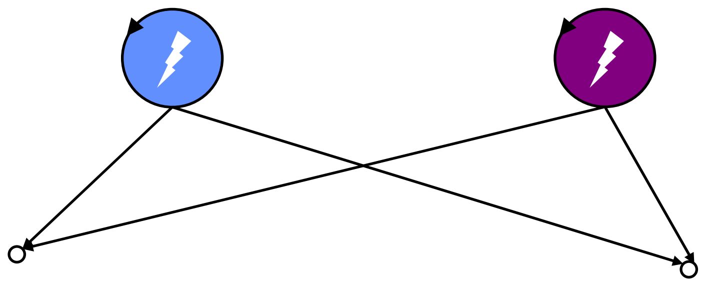
How About This? (#3)





```
blue->Init(1);
purple->Init(0);

void CallThis() {
        blue->P();
        Compute();
        purple->V();
    }
}

void CallThat() {
        purple->P();
        Compute();
        blue->V();
}
```

How About This? (#4)



```
blue->Init(1);
purple->Init(0);

void CallThis() {
        blue->P();
        Compute();
        purple->V();
    }
}

void CallThat() {
        purple->P();
        Compute();
        blue->V();
}
```

Basic Producer/Consumer

```
empty->Init(1);
full->Init(0);
int buf;

void Produce(int m) {
        empty->P();
        buf = m;
        full->V();
}
```

```
int Consume() {
        int m;
        full->P();
        m = buf;
        empty->V();
        return(m);
}
```

This use of a semaphore pair is called a *split binary semaphore*: the sum of the values is always one.

A Bounded Resource with a Counting Semaphore

```
semaphore->Init(N);

int AllocateEntry() {
        int i;
        semaphore->Down();
        ASSERT(FindFreeItem(&i));
        slot[i] = 1;
        return(i);
}

void ReleaseEntry(int i) {
        slot[i] = 0;
        semaphore->Up();
}
```

A semaphore for an N-way resource is called a *counting semaphore*.

A caller that gets past a *Down* is guaranteed that a resource instance is reserved for it.

Problems?

Note: the current value of the semaphore is the number of resource instances free to allocate.

But semaphores do not allow a thread to read this value directly. Why not?

Bounded Resource with a Condition Variable

```
Mutex* mx;
Condition *cv;

int AllocateEntry() {
        int i;
        mx->Acquire();
        while(!FindFreeItem(&i))
                cv.Wait(mx);
        slot[i] = 1;
        mx->Release();
        return(i);
}

void ReleaseEntry(int i) {
        mx->Acquire();
        slot[i] = 0;
        cv->Signal();
        mx->Release();
}
```

"Loop before you leap."

Why is this *Acquire* needed?

Reader/Writer with Semaphores

```
SharedLock::AcquireRead() {
    rmx.P();
    if (first reader)
        wsem.P();
    rmx.V();
}

SharedLock::ReleaseRead() {
    rmx.P();
    if (last reader)
        wsem.V();
    rmx.V();
}
```

```
SharedLock::AcquireWrite() {
    wsem.P();
}

SharedLock::ReleaseWrite() {
    wsem.V();
}
```

Reader/Writer with Semaphores: Take 2

```
SharedLock::AcquireRead() {
    rblock.P();
    rmx.P();
    if (first reader)
        wsem.P();
    rmx.V();
    rblock.V();
}

SharedLock::ReleaseRead() {
    rmx.P();
    if (last reader)
        wsem.V();
    rmx.V();
}

SharedLock::AcquireWrite() {
    wmx.P();
    if (first writer)
        rblock.P();
    wmx.V();
    wsem.P();
}

SharedLock::ReleaseWrite() {
    wsem.V();
    wmx.P();
    if (last writer)
        rblock.V();
    wmx.V();
}
```

Reader/Writer with Semaphores: Take 2+

```
SharedLock::AcquireRead() {
    rblock.P();
    if (first reader)
        wsem.P();
    rblock.V();
}

SharedLock::ReleaseRead() {
    if (last reader)
        wsem.V();
}
```

```
SharedLock::AcquireWrite() {
    if (first writer)
        rblock.P();
    wsem.P();
}

SharedLock::ReleaseWrite() {
    wsem.V();
    if (last writer)
        rblock.V();
}
```

The rblock prevents readers from entering while writers are waiting.

Spin-Yield: Just Say No

```
void
Thread::Await() {
        awaiting = TRUE;
        while(awaiting)
                Yield();
}


void
Thread::Awake() {
        if (awaiting)
                awaiting = FALSE;
}
```

Tricks of the Trade #1

```
int initialized = 0;
Lock initMx;

void Init() {
        InitThis(); InitThat();
        initialized = 1;
}

void DoSomething() {
        if (!initialized) {            /* fast unsynchronized read of a WORM datum */
               initMx.Lock();          /* gives us a “hint” that we’re in a race to write */
               if (!initialized)       /* have to check again while holding the lock */
                  Init();
               initMx.Unlock();        /* slow, safe path */
        }
        DoThis();  DoThat();
}
```

The "Magic" of Semaphores and CVs

Any use of *sleep/wakeup* synchronization can be replaced with semaphores or condition variables.

- Most uses of blocking synchronization have some associated state to record the blocking condition.
  - e.g., list or count of waiting threads, or a table or count of free resources, or the completion status of some operation, or....
  - The trouble with *sleep/wakeup* is that the program must update the state atomically with the *sleep/wakeup*.
- Semaphores integrate the state into atomic *P/V* primitives.
  - ....but the only state that is supported is a simple counter.
- Condition variables (CVs) allow the program to define the condition/state, and protect it with an integrated mutex.

Blocking in *Sleep*

- An executing thread may request some resource or action that causes it to *block* or *sleep* awaiting some event.
  - passage of a specific amount of time (a ***pause*** request)
  - completion of I/O to a slow device (e.g., keyboard or disk)
  - release of some needed resource (e.g., memory)
  - In Nachos, threads block by calling ***Thread::Sleep.***
- A sleeping thread cannot run until the event occurs.
- The blocked thread is awakened when the event occurs.
  - E.g., ***Wakeup*** or Nachos ***Scheduler::ReadyToRun(Thread* t)***
- In an OS, threads or processes may sleep while executing in the kernel to handle a system call or fault.

Avoiding Races #2

Is caution with *yield* and *sleep* sufficient to prevent races?

No!

Concurrency races may also result from:

- involuntary context switches (timeslicing)

  driven by timer interrupts, which may occur at any time

- external events that asynchronously change the flow of control

  interrupts (inside the kernel) or signals/APCs (outside the kernel)

- physical concurrency (on a multiprocessor)

How to ensure atomicity of critical sections in these cases?

Synchronization primitives!

Synchronization 101

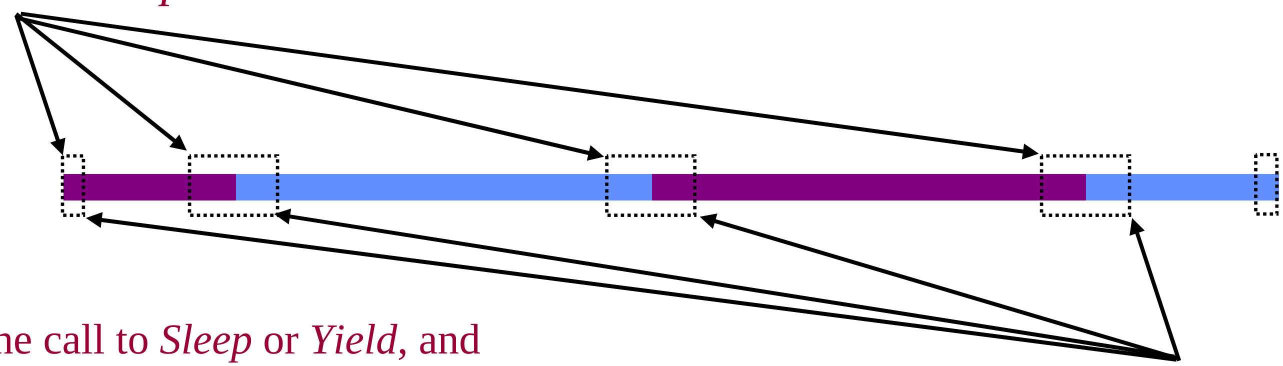
Synchronization constrains the set of possible interleavings:

- Threads can't prevent the scheduler from switching them out, but they can "agree" to stay out of each other's way.
  - voluntary blocking or spin-waiting on entrance to critical sections
  - notify blocked or spinning peers on exit from the critical section
- In the kernel we can *temporarily* disable interrupts.
  - no races from interrupt handlers or involuntary context switches
  - a blunt instrument to use as a last resort
    - *Disabling interrupts is not an accepted synchronization mechanism!*
  - insufficient on a multiprocessor

Digression: Sleep and Yield in Nachos

disable interrupts

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



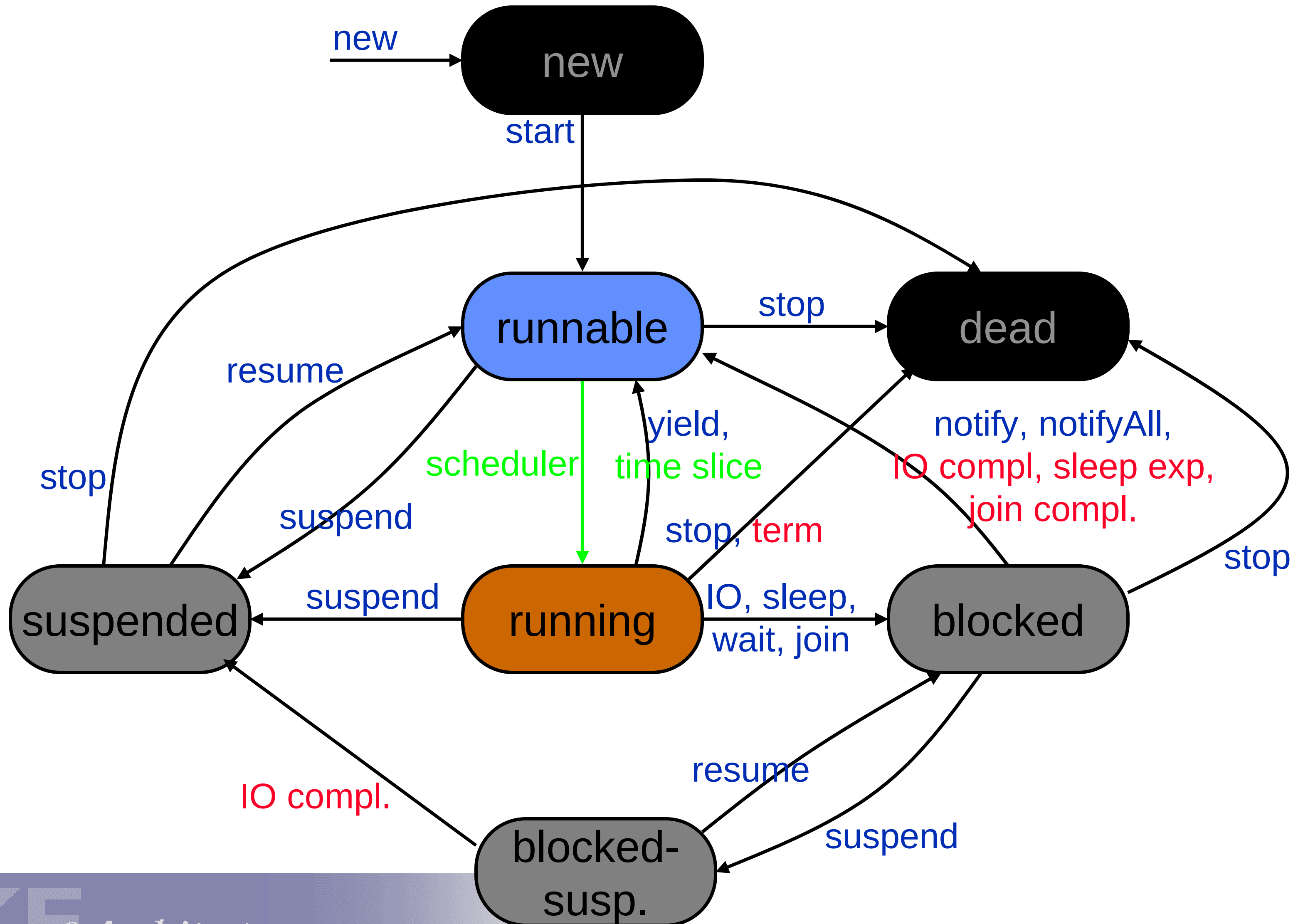
Disable interrupts on the call to *Sleep* or *Yield*, and rely on the "other side" to re-enable on return from its own *Sleep* or *Yield*.

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);
}
```

Thread state transitions in Java 1.1 and earlier



Context Switches: Voluntary and Involuntary

On a **uniprocessor**, the set of possible execution schedules depends on *when context switches can occur*.

- *Voluntary*: one thread explicitly yields the CPU to another.

  E.g., a Nachos thread can suspend itself with ***Thread::Yield***.

  It may also *block* to wait for some event with ***Thread::Sleep***.

- *Involuntary:* the system *scheduler* suspends an active thread, and switches control to a different thread.

  Thread scheduler tries to share CPU fairly by *timeslicing*.

  Suspend/resume at periodic intervals

  Involuntary context switches can happen "any time".

Why Threads Are Important

1. There are lots of good reasons to use threads.

    “easy” coding of multiple activities in an application

        e.g., servers with multiple independent clients

    parallel programming to reduce execution time

2. Threads are great for experimenting with concurrency.

    context switches and interleaved executions

    race conditions and synchronization

    can be supported in a library (Nachos) without help from OS

3. We will use threads to implement processes in Nachos.

    (Think of a thread as a process running *within the kernel*.)