Administrivia

- Nachos guide and Lab #1 are on the web.
  http://www.cs.duke.edu/~chase/cps210
- Form project teams of 2-3.
- Lab #1 due February 5.
- Synchronization problem set is up: due January 29.
- Synchronization hour exam on January 29.
- Readings page is up.
- Read Tanenbaum ch 2-3 and Birrell for Thursday's class.

Threads and Concurrency

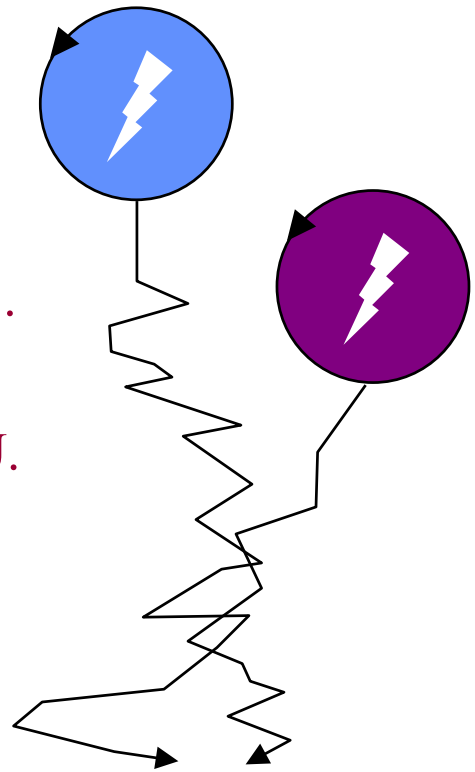
Threads

A *thread* is a schedulable stream of control.

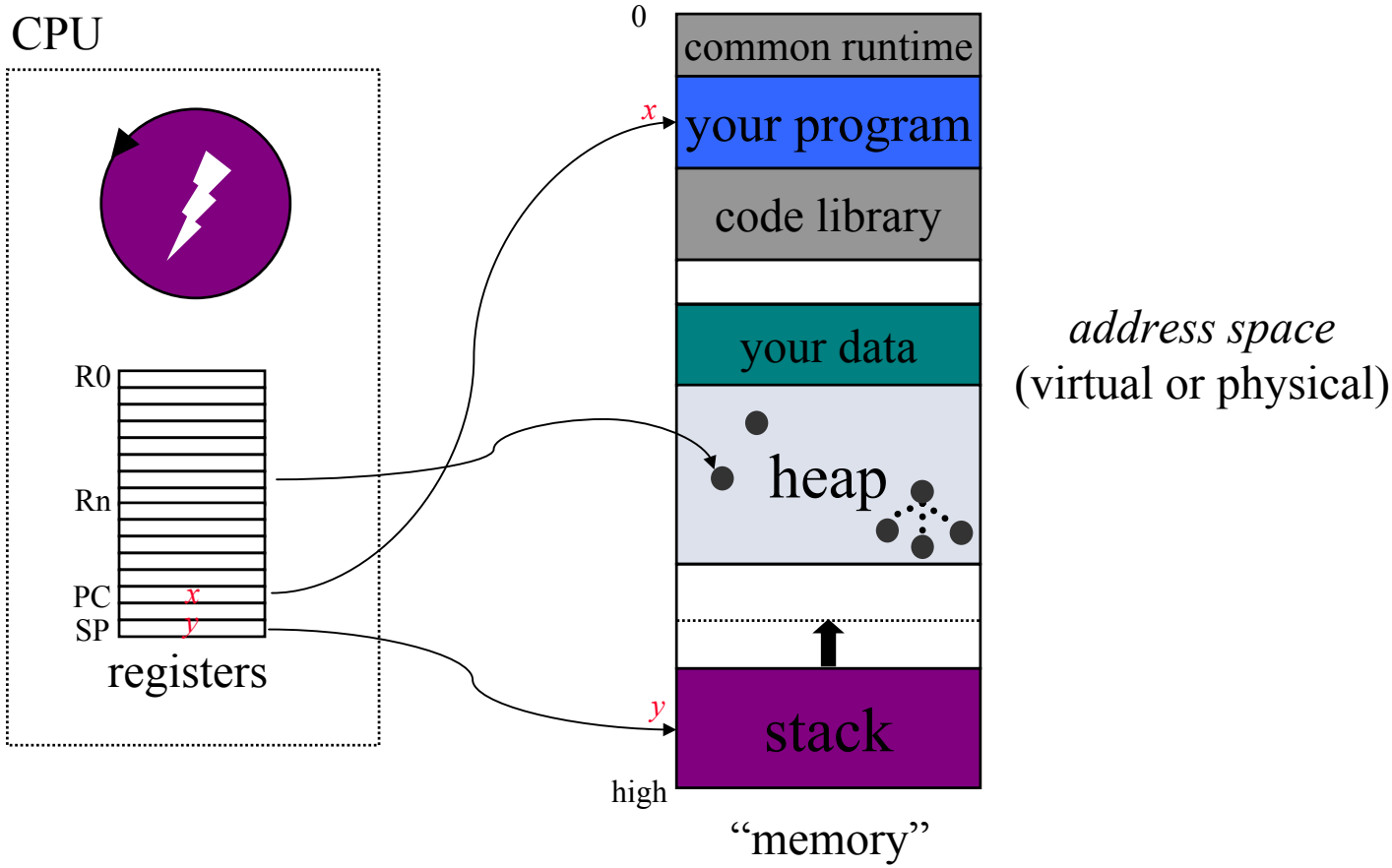
- defined by CPU register values (PC, SP)
- *suspend*: save register values in memory
- *resume*: restore registers from memory

Multiple threads can execute independently:

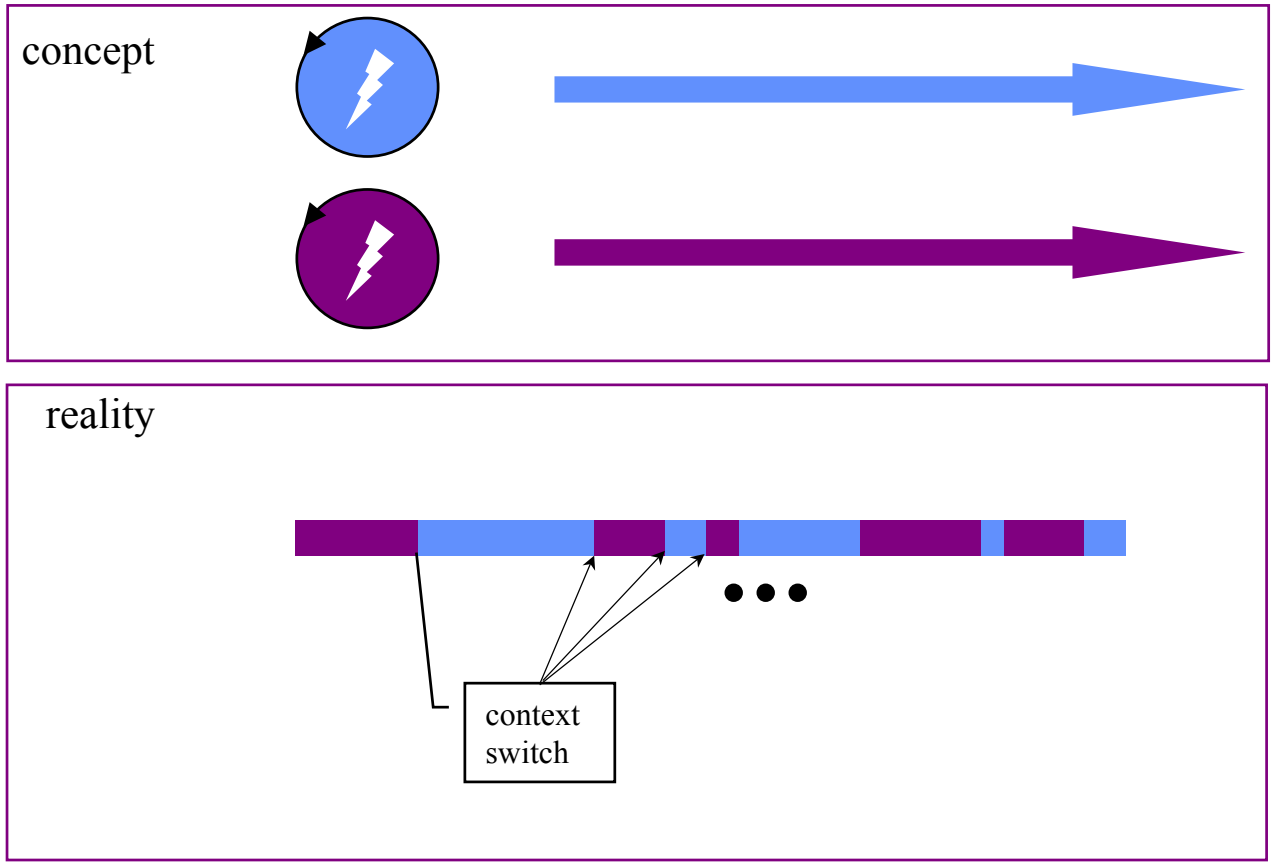
- They can run in parallel on multiple CPUs...
  - ***physical concurrency***
- …or arbitrarily interleaved on a single CPU.
  - ***logical concurrency***
- Each thread must have its own stack.



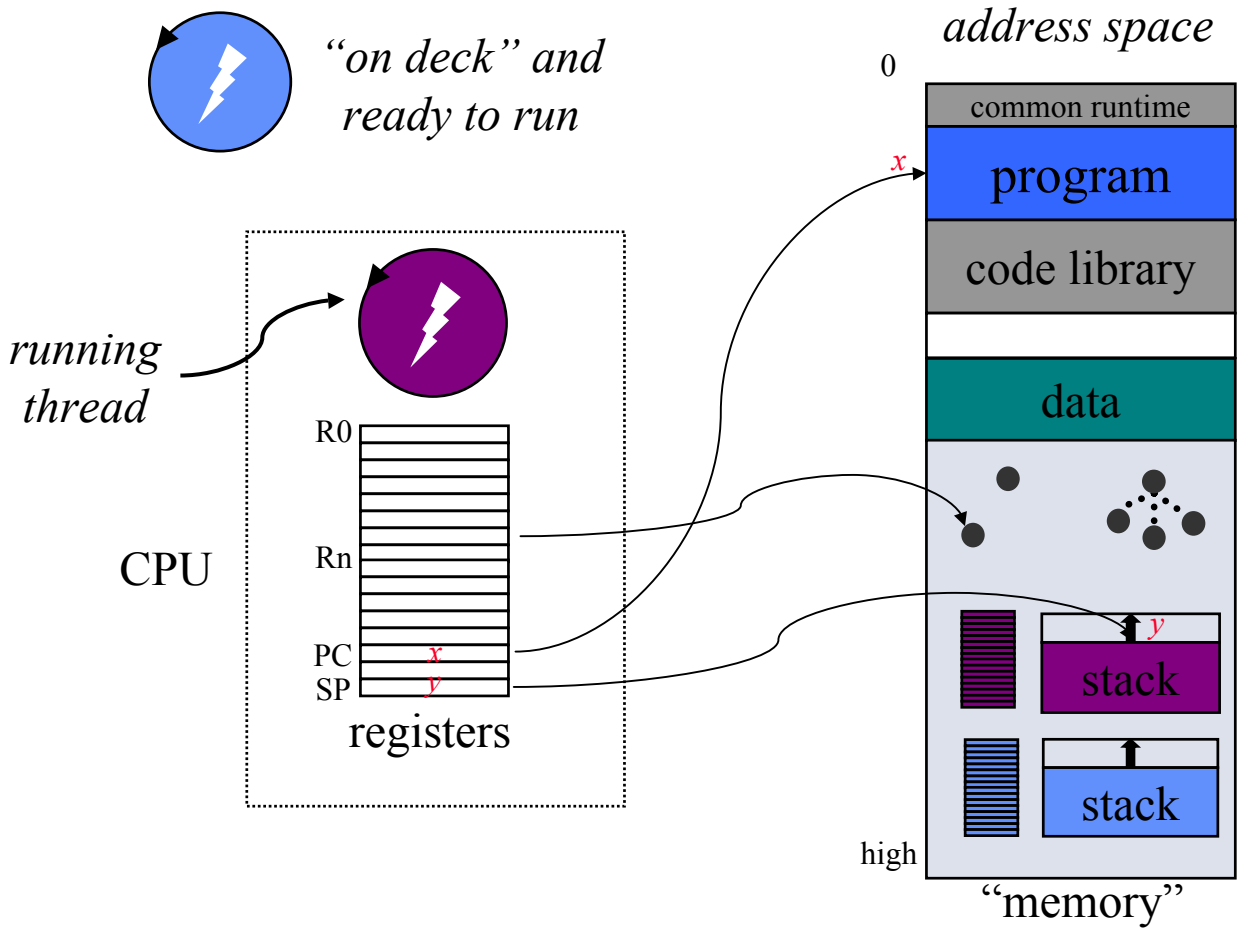
A Peek Inside a Running Program



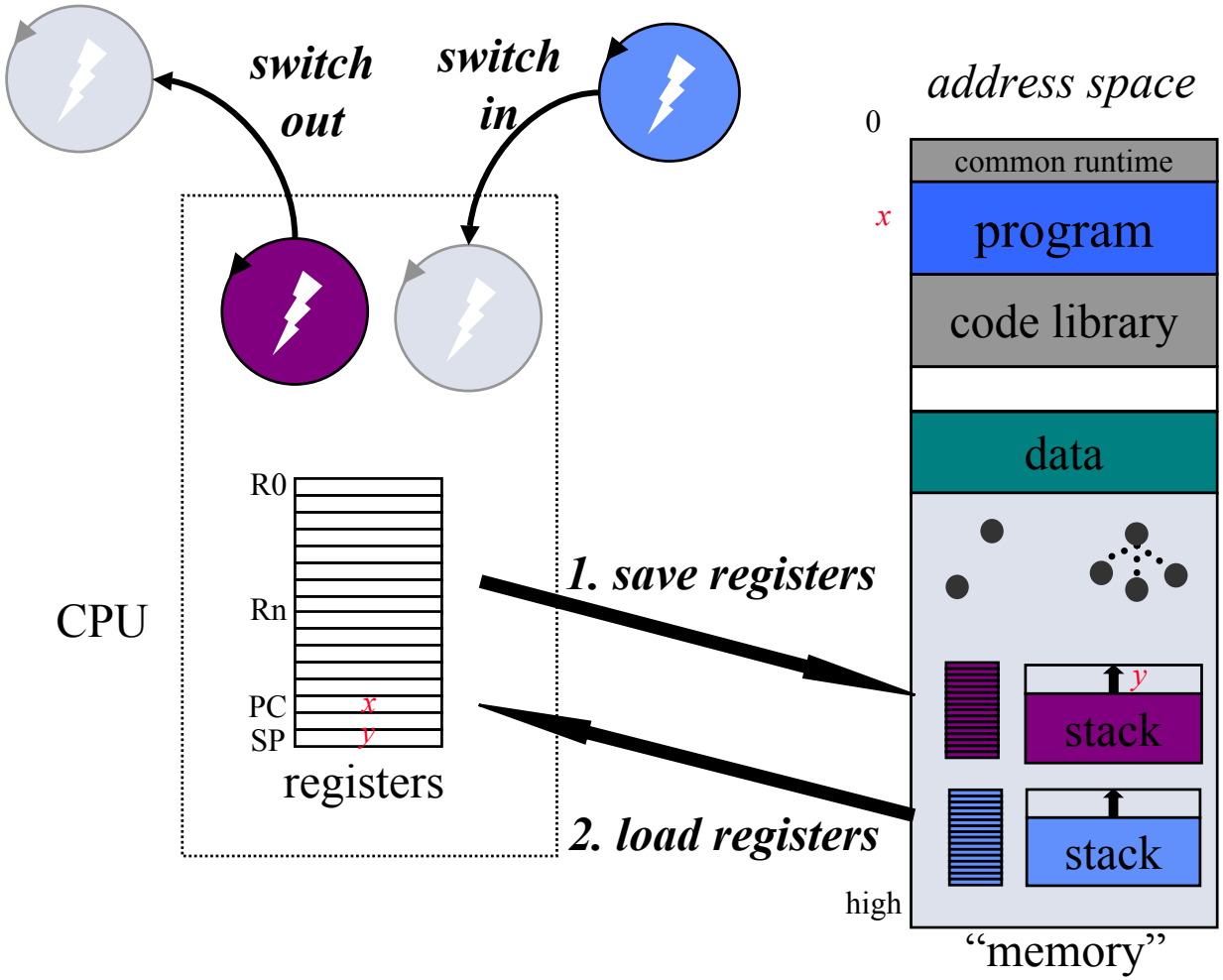
Two Threads Sharing a CPU



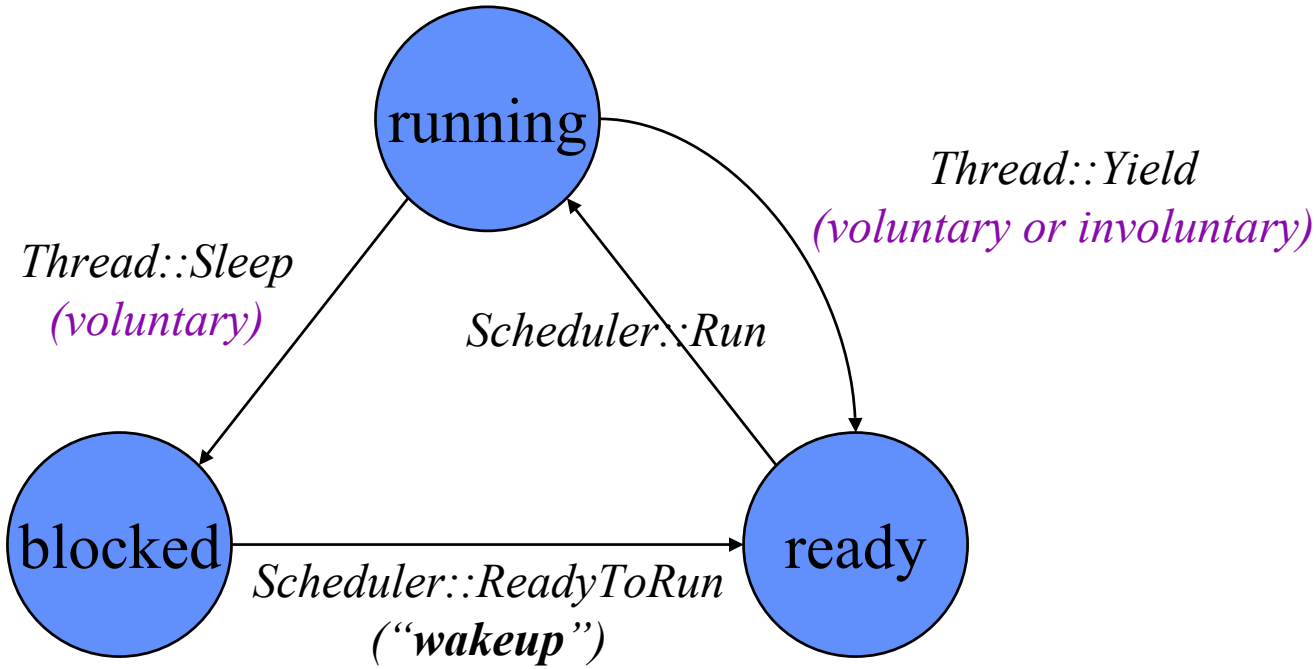
A Program With Two Threads



Thread Context Switch



Thread States and Transitions



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.

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*.)

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)

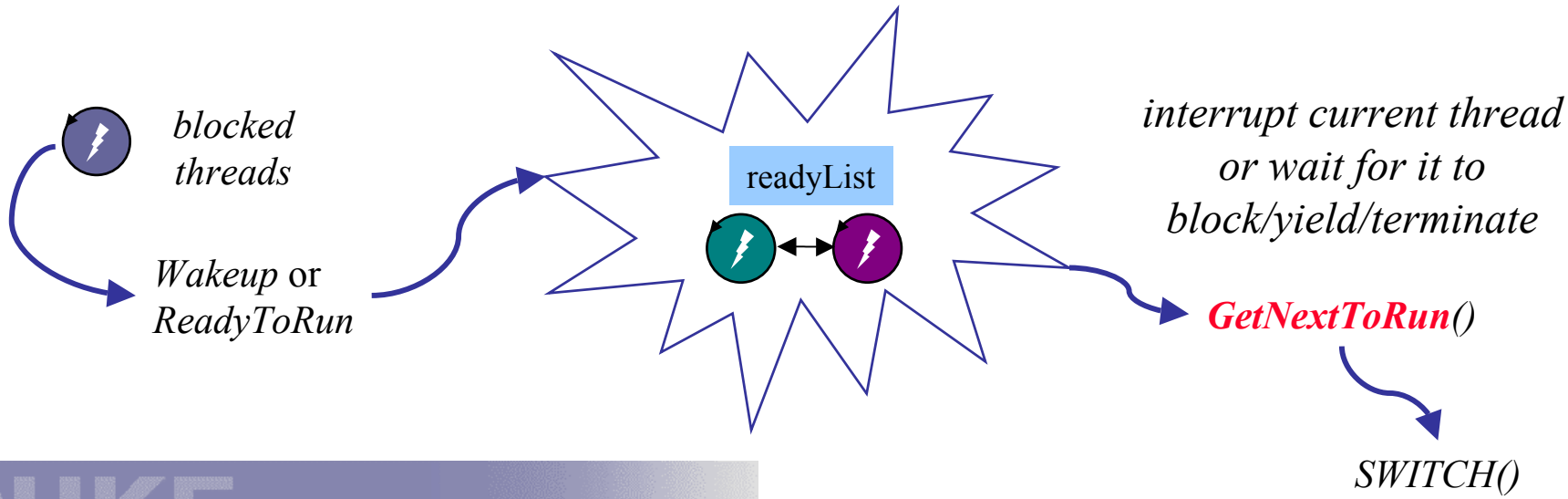
Warning: concurrency can cause your programs to behave unpredictably, e.g., crash and burn.

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*.



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 (e.g., **nachos -rs**)

  Involuntary context switches can happen "any time".

The Dark Side of Concurrency

With interleaved executions, the order in which 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

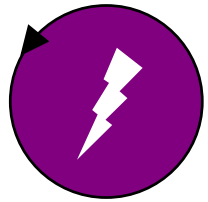
Some schedule interleavings may lead to incorrect behavior.

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

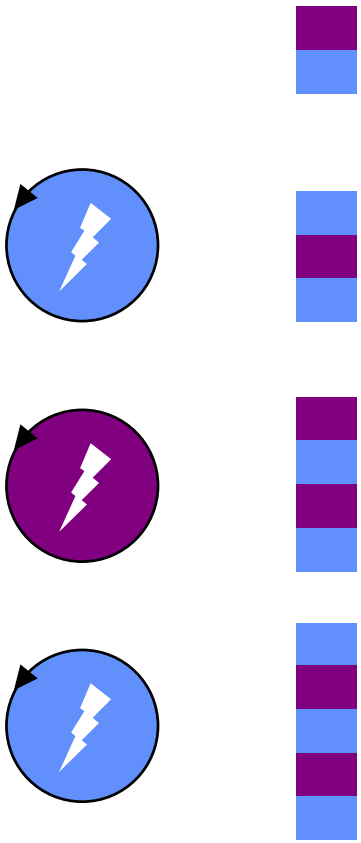
The system must provide a way to coordinate concurrent activities to avoid incorrect interleavings.

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);
        }
}
```

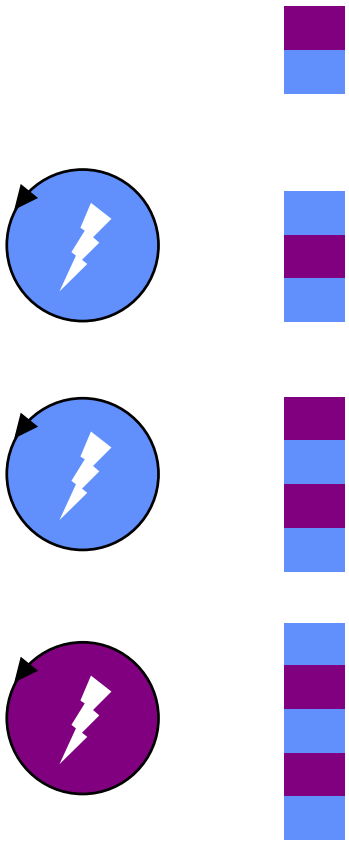


Interleaving the Color Stack #1



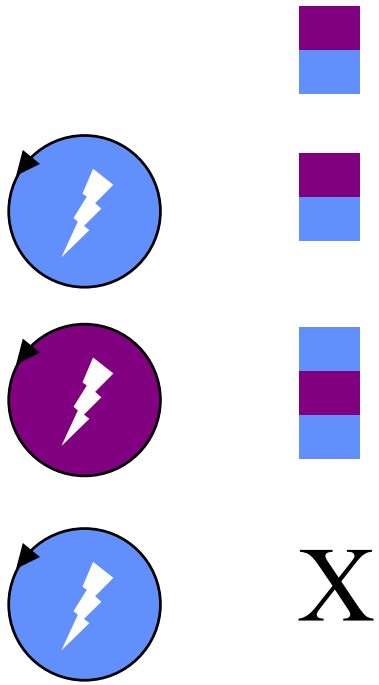
```
PushColor() {
        if (s[top] ==  purple) {
                ASSERT(s[top-1] ==  blue);
                push(blue);
        } else {
                ASSERT(s[top] == blue);
                ASSERT(s[top-1] == purple);
                push(purple);
        }
}

ThreadBody() {
        while(1)
                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);
}
```

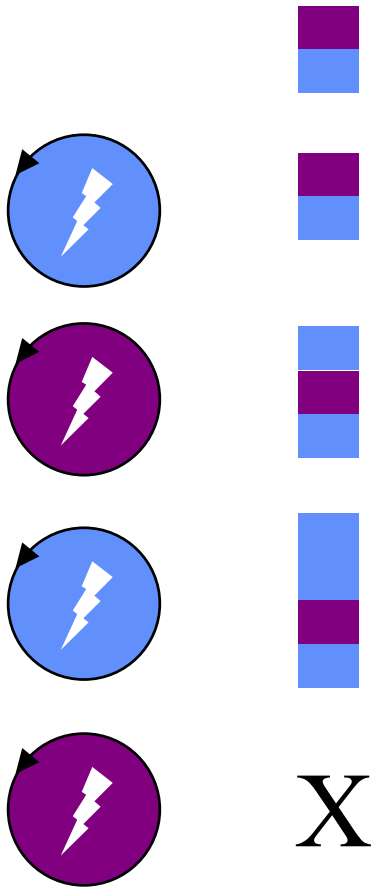
Interleaving the Color Stack #3



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);
}
```

Threads vs. Processes

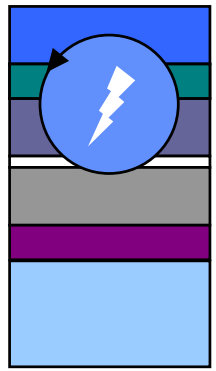
1. The *process* is a *kernel abstraction* for an independent executing program.

   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)*



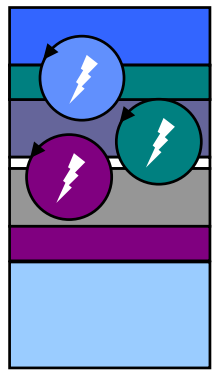
2. Threads may share an address space

   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

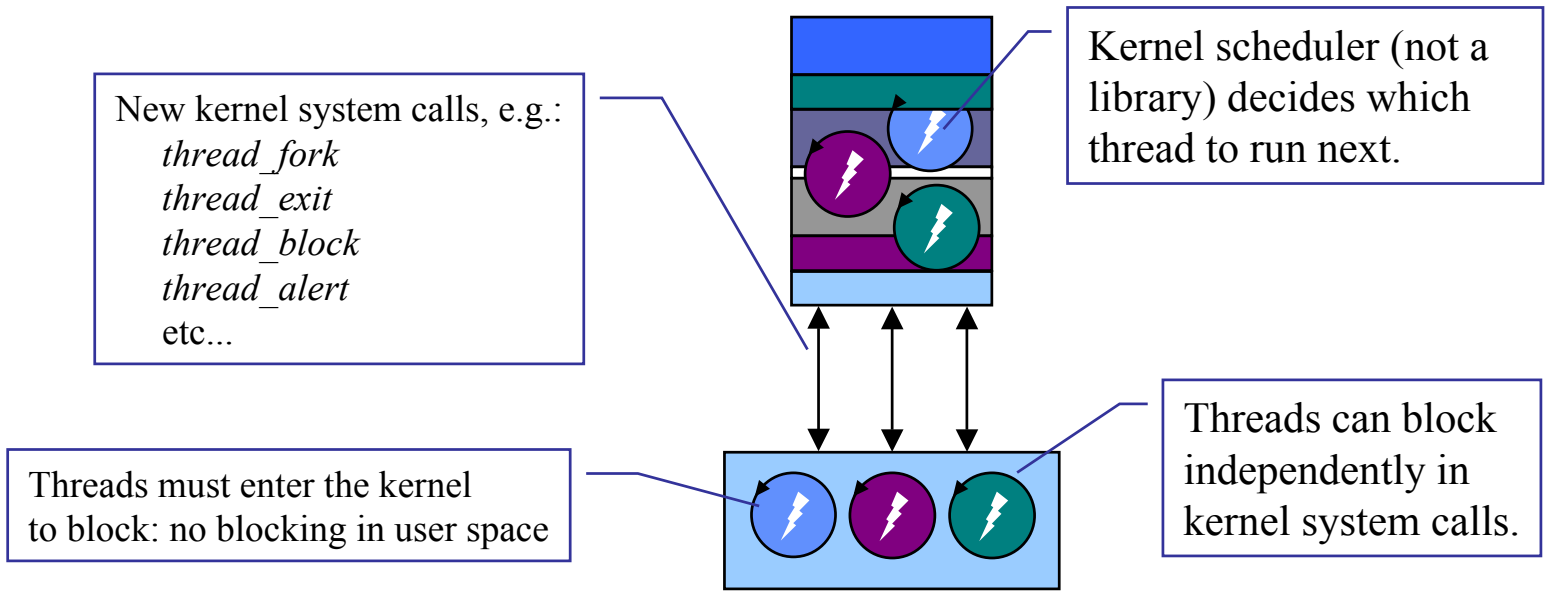
Kernel-Supported Threads

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

- thread model and scheduling defined by OS

  Nachos kernel can support them: extra credit in Labs 4 and 5

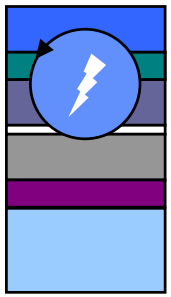
  NT, advanced Unix, etc.



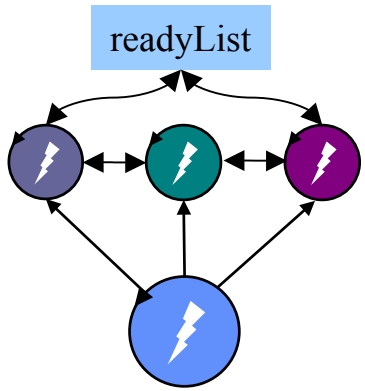
User-level Threads

The Nachos library implements *user-level threads*.

- 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

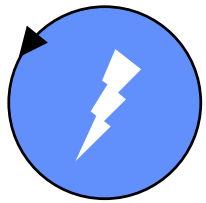


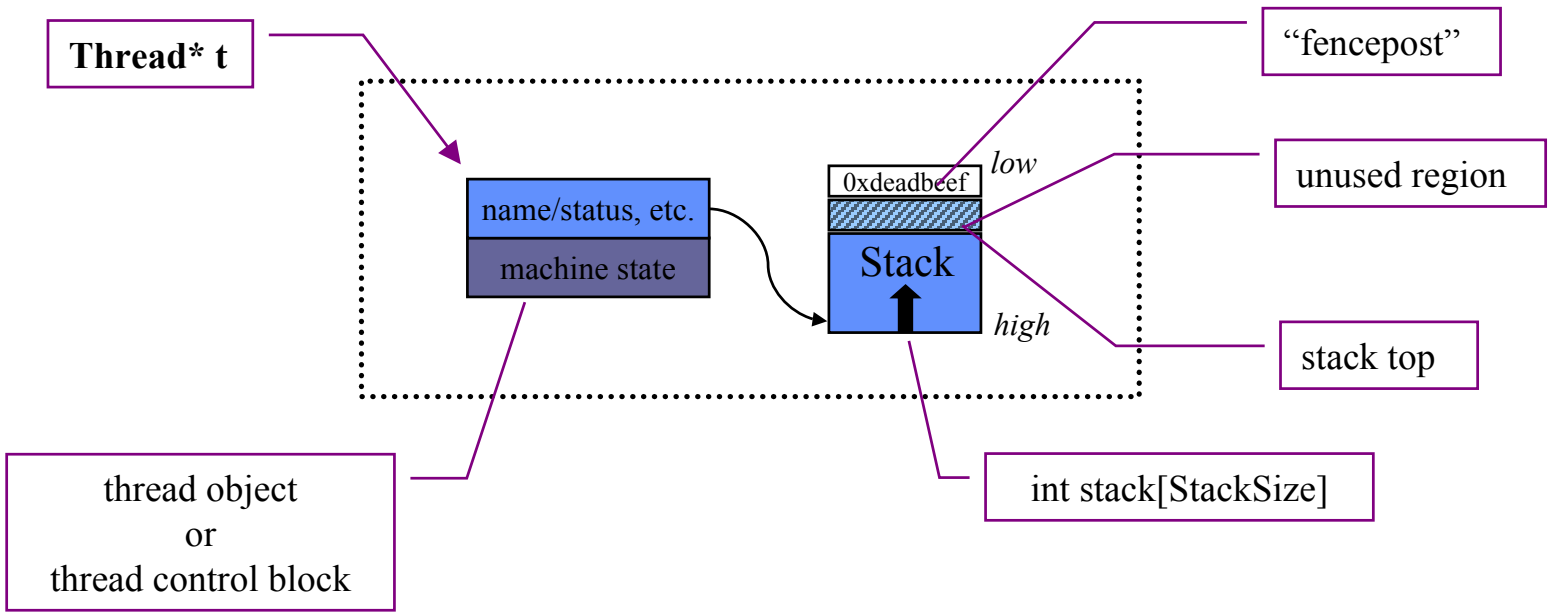
```
while(1) {
    t = get next ready thread;
    scheduler->Run(t);
}
```



A Nachos Thread

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





A Nachos Context Switch

```
/*
 * 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.*

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.

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);
        }
}
```

Avoiding Races #2

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

No!

Concurrency races may also result from:

- involuntary context switches (timeslicing)

  e.g., caused by the Nachos thread scheduler with **-rs** flag

- 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

- If we're "inside the kernel" (e.g., the Nachos kernel), we can *temporarily* disable interrupts.

  no races from interrupt handlers or involuntary context switches

  a blunt instrument to use as a last resort

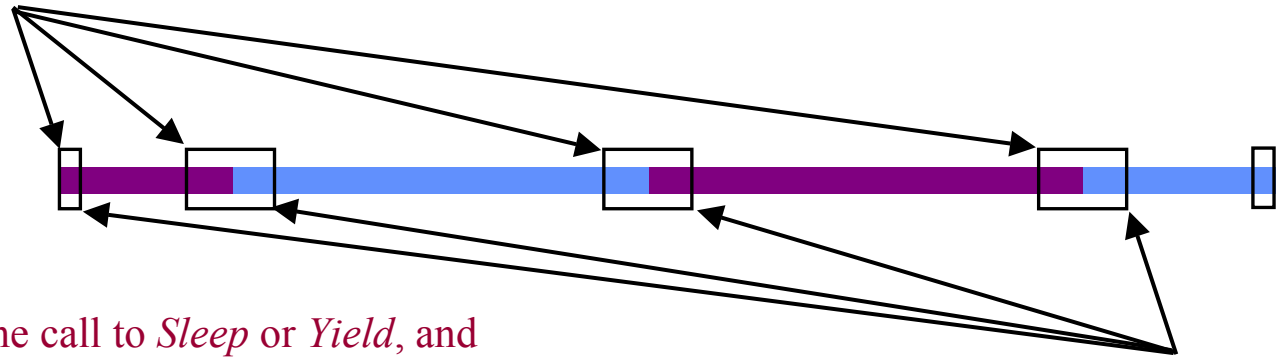
  Disabling interrupts is not an accepted synchronization mechanism!

  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*.

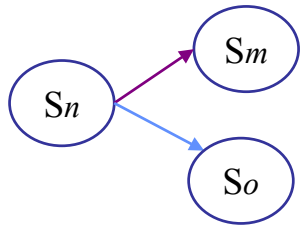
enable interrupts

```
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);
}
```

Resource Trajectory Graphs

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



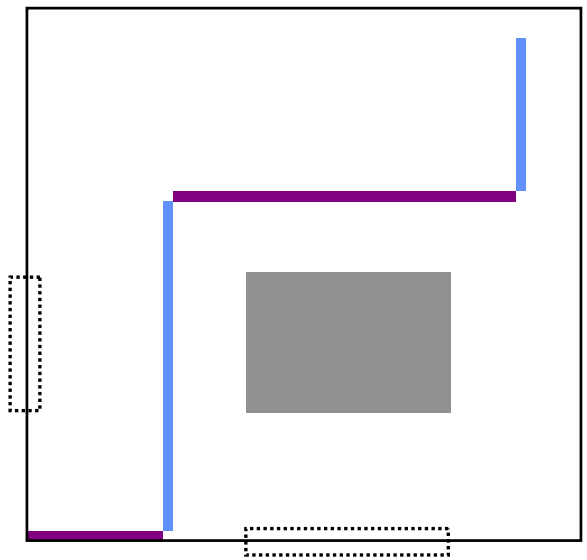
RTG for N threads is N-dimensional.

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.

Questions about Nachos Context Switches

- Can I trace the stack of a thread that has switched out and is not active? What will I see?

  (a thread in the READY or BLOCKED state)

- What happens on the first switch into a newly forked thread?

  Thread::Fork (actually **StackAllocate**) sets up for first switch.

- What happens when the thread's main procedure returns?
- When do we delete a dying thread's stack?
- How does Nachos know what the current thread is?

Debugging with Threads

Lab 1: demonstrate races with the Nachos List class.

- Some will result in a crash; know how to analyze them.

  > **gdb nachos** [or **ddd**]

  (gdb) **run *program_arguments***

  Program received signal SIGSEGV, Segmentation Fault.

  0x10954 in *function_name*(*function_args*) at *file.c*:*line_number*

  (gdb) **where**

- Caution: gdb [ddd] is not Nachos-thread aware!

  A context switch will change the stack pointer. Before stepping:

  (gdb) **break SWITCH**