CPS 310
Synchronization 101

Jeff Chase

Duke University

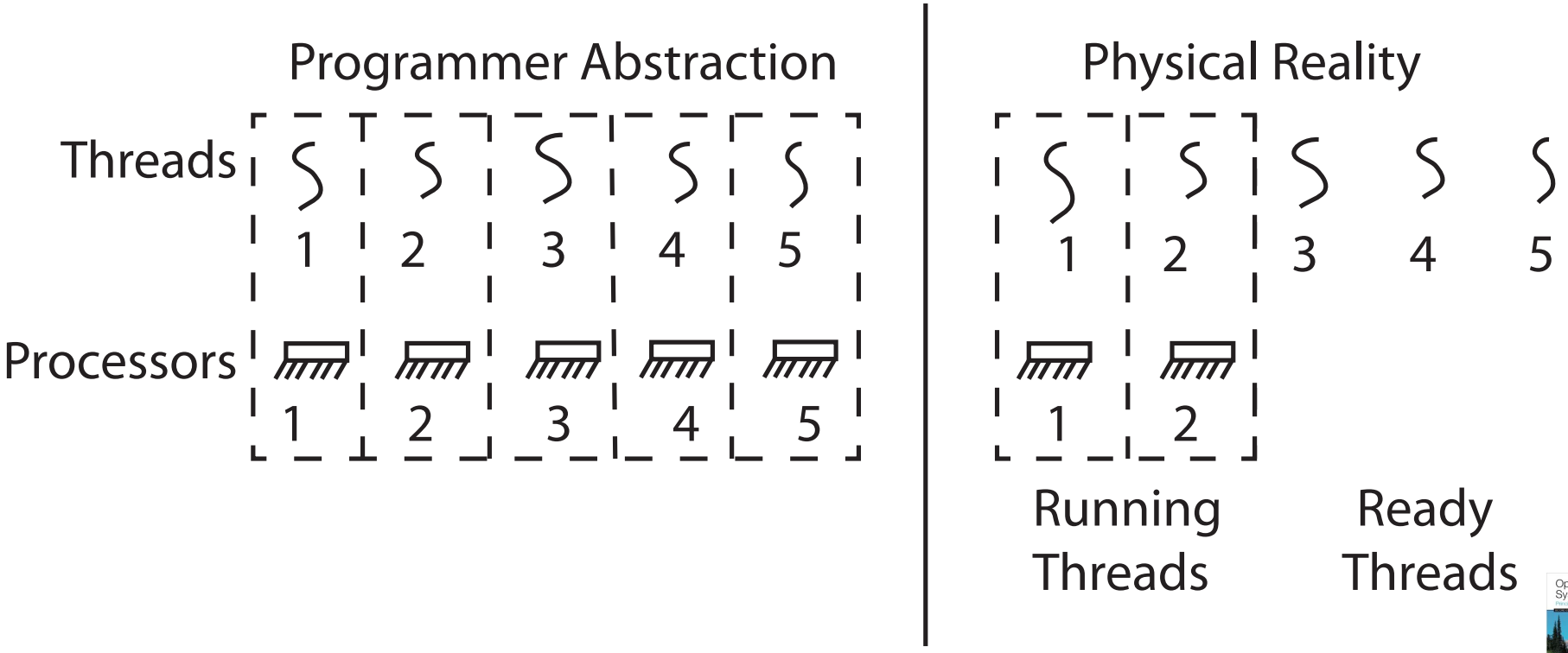
Concurrency
Control it!

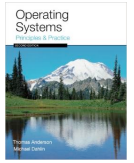
Sorcerer's Apprentice

- **The scheduler (and the machine) select the execution order of threads**
  - Each thread executes a sequence of instructions, but their sequences may be arbitrarily interleaved.
    - E.g., from the point of view of **loads/stores** on memory.
  - Each possible execution order is a **schedule**.
  - A **thread-safe** program must **exclude** schedules that lead to incorrect behavior.
  - It is called **synchronization** or **concurrency control**.

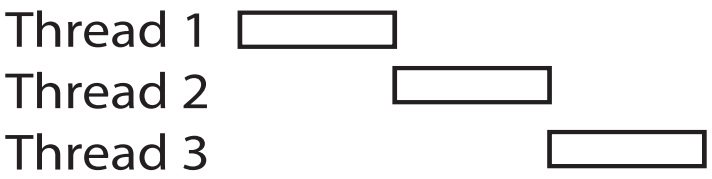
Thread Abstraction

- Infinite number of processors
- Threads execute with variable speed
  - Programs must be designed to work with any schedule

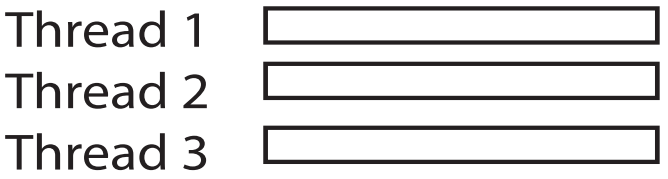




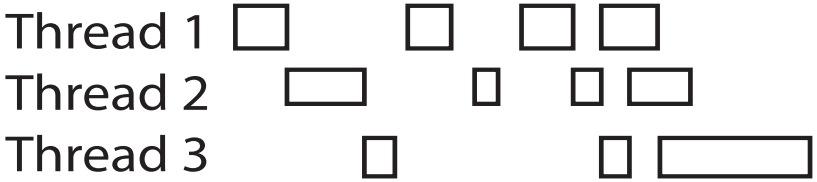
Possible Executions



a) One execution

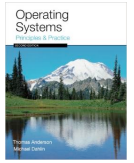


b) Another execution



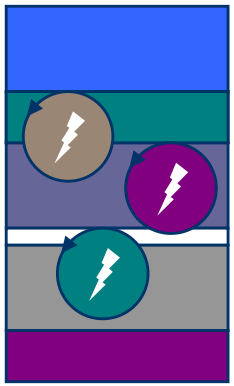
c) Another execution

These executions are "**schedules**" chosen by the system. Threads may constrain the schedule by blocking: a blocked thread cannot run.



Pthread (posix thread) example

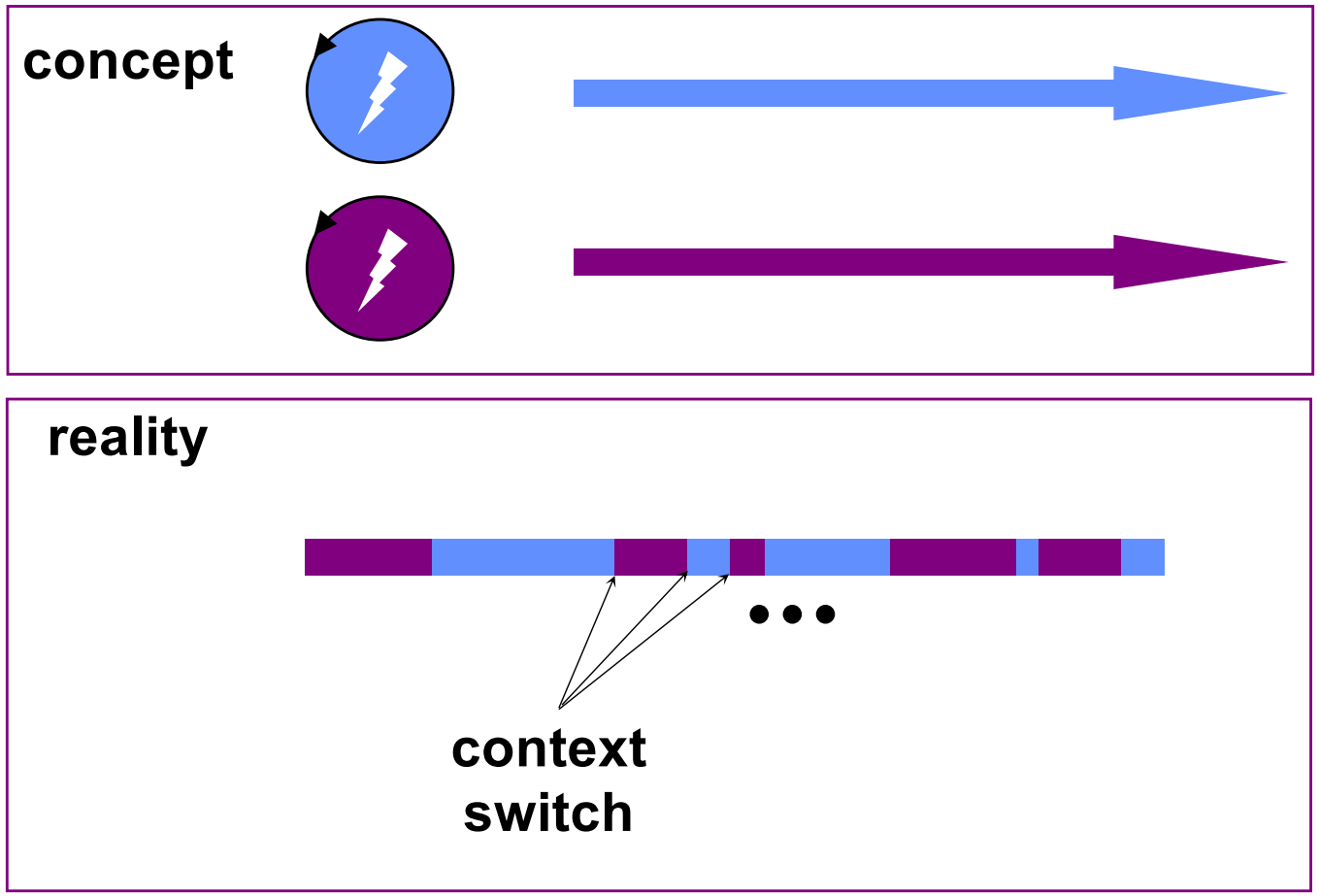
```
volatile int counter = 0;
int loops;

void *worker(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
      counter++;
    }
    pthread_exit(NULL);
}
```



```
int main(int argc, char *argv[]) {
    if (argc != 2) {
      fprintf(stderr, "usage: threads <loops>\n");
      exit(1);
    }
    loops = atoi(argv[1]);
    pthread_t p1, p2;
    printf("Initial value : %d\n", counter);
    pthread_create(&p1, NULL, worker, NULL);
    pthread_create(&p2, NULL, worker, NULL);
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("Final value   : %d\n", counter);
    return 0;
}
```

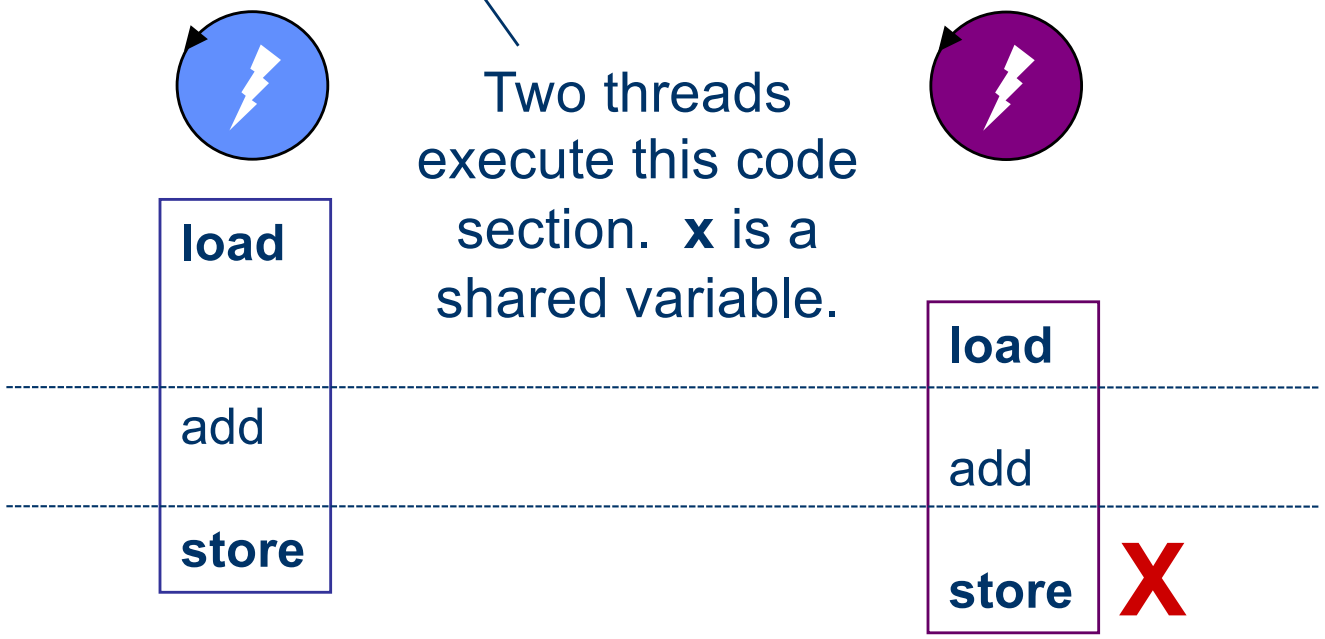
[pthread code from OSTEP]

Two threads sharing a CPU/core



Interleaving matters

```
load    x, R2          ; load global variable x
add     R2, 1, R2      ; increment: x = x + 1
store   R2, x          ; store global variable x
```



In this schedule, **x** is incremented only once: last writer wins.
The program breaks under this schedule. This bug is a **race**.

OSTEP pthread example: before and after

```
chase$ cd c-samples
chase$ cc -o before threads.v0.c
chase$ cc -o after threads.v1.c
```

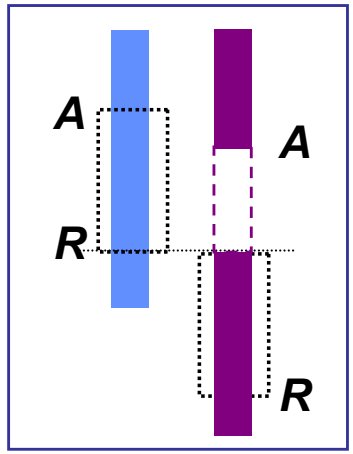
```
chase$ ./before 10000
Initial value : 0
Final value   : 15949
chase$ ./before 10000
Initial value : 0
Final value   : 17581
chase$ ./before 10000
Initial value : 0
Final value   : 15363
chase$ ./before 10000
Initial value : 0
Final value   : 15482
chase$ ./before 10000
Initial value : 0
Final value   : 13956
chase$ ./before 10000
Initial value : 0
Final value   : 15115
```

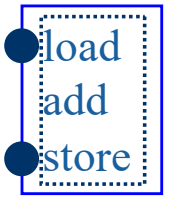
```
chase$ ./after 10000
Initial value : 0
Final value   : 20000
chase$ ./after 10000
Initial value : 0
Final value   : 20000
chase$ ./after 10000
Initial value : 0
Final value   : 20000
chase$
```

OSTEP pthread example (2)

```
pthread_mutex_t m;
volatile int counter = 0;
int loops;

void *worker(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&m);
        counter++;
        Pthread_mutex_unlock(&m);
    }
    pthread_exit(NULL);
}
```

“Lock it down.”



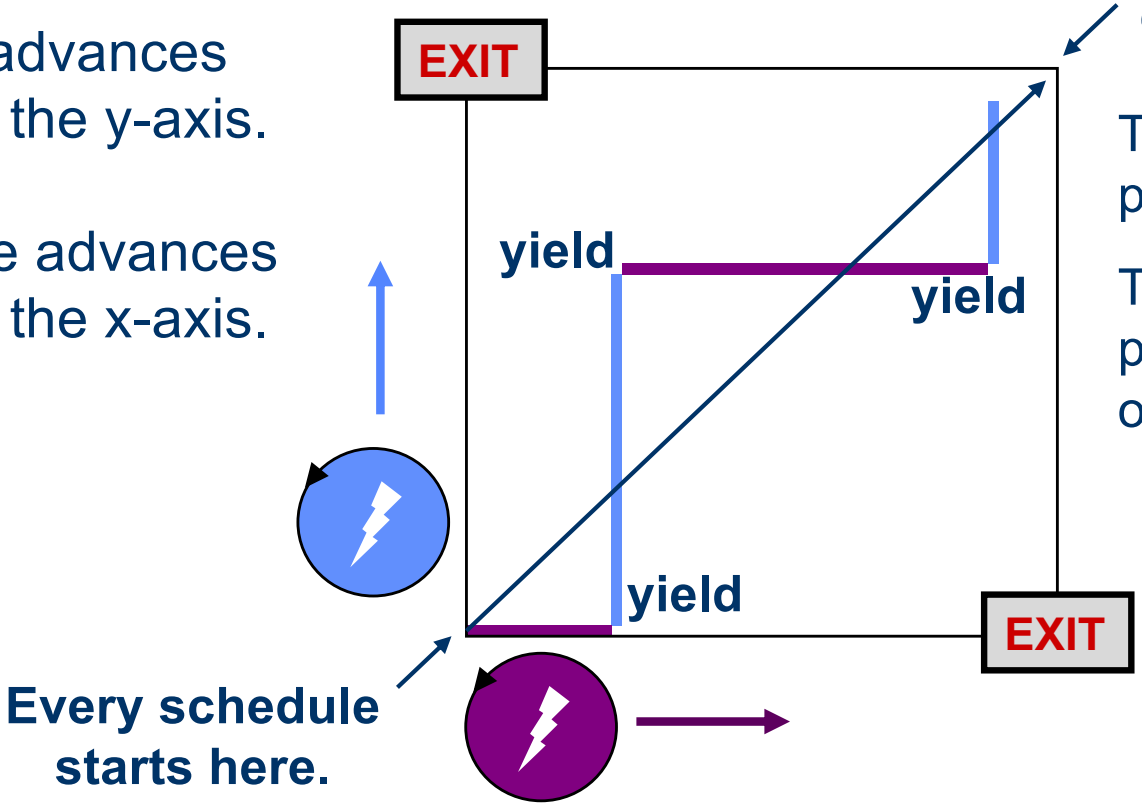


Resource Trajectory Graphs

This RTG depicts a schedule within the space of possible schedules for a simple program of two threads sharing one core.

Blue advances along the y-axis.

Purple advances along the x-axis.



Every schedule ends here.

The diagonal is an idealized parallel execution (two cores).

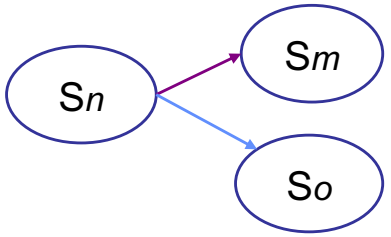
The scheduler chooses the path (schedule, event order, or interleaving).

Every schedule starts here.

From the point of view of the program, the chosen path is **nondeterministic**.

Resource Trajectory Graphs

Resource trajectory graphs (RTG) depict the "random walk" through the space of possible program states.



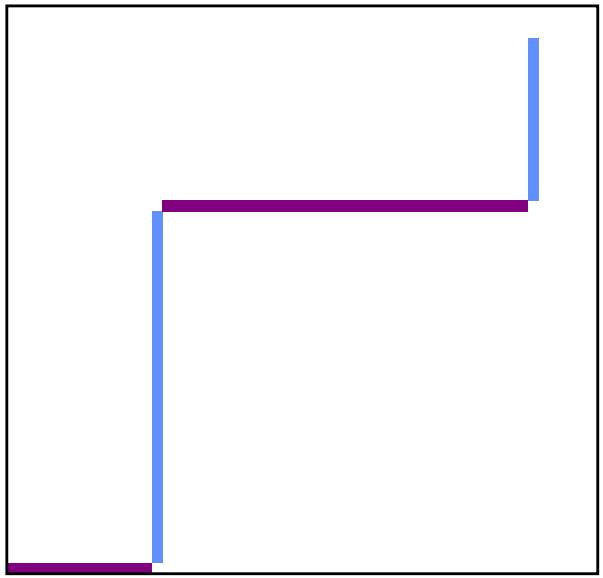
RTG is useful to depict all possible executions of multiple threads. I draw them for only two threads because slides are two-dimensional.

RTG for N threads is N-dimensional.

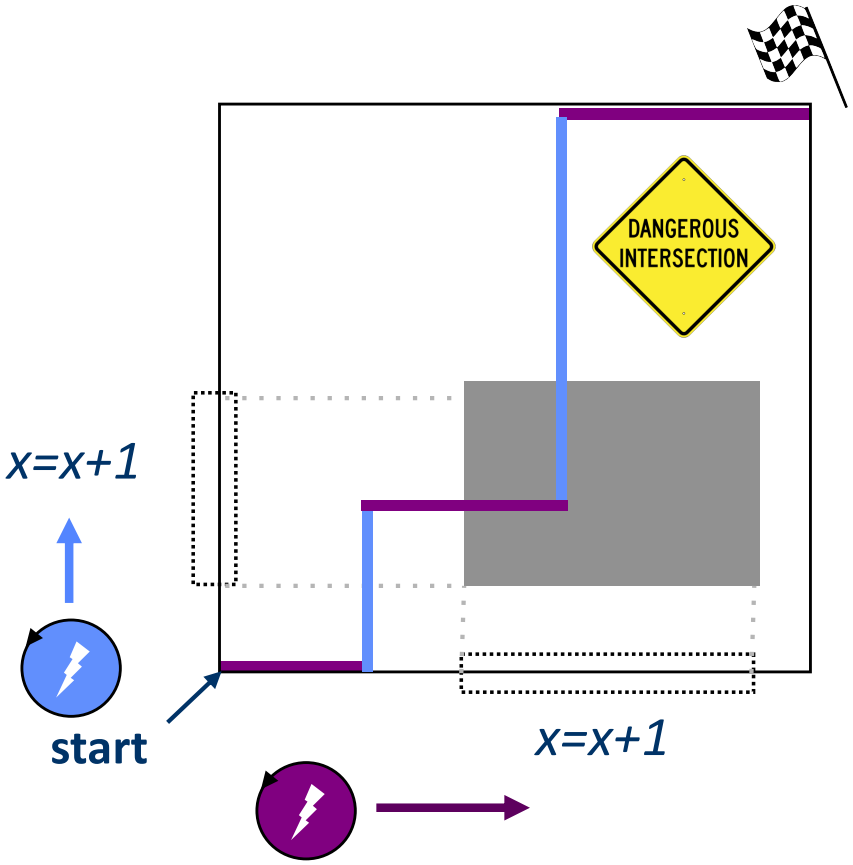
Thread *i* advances along axis *i*.

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



A race



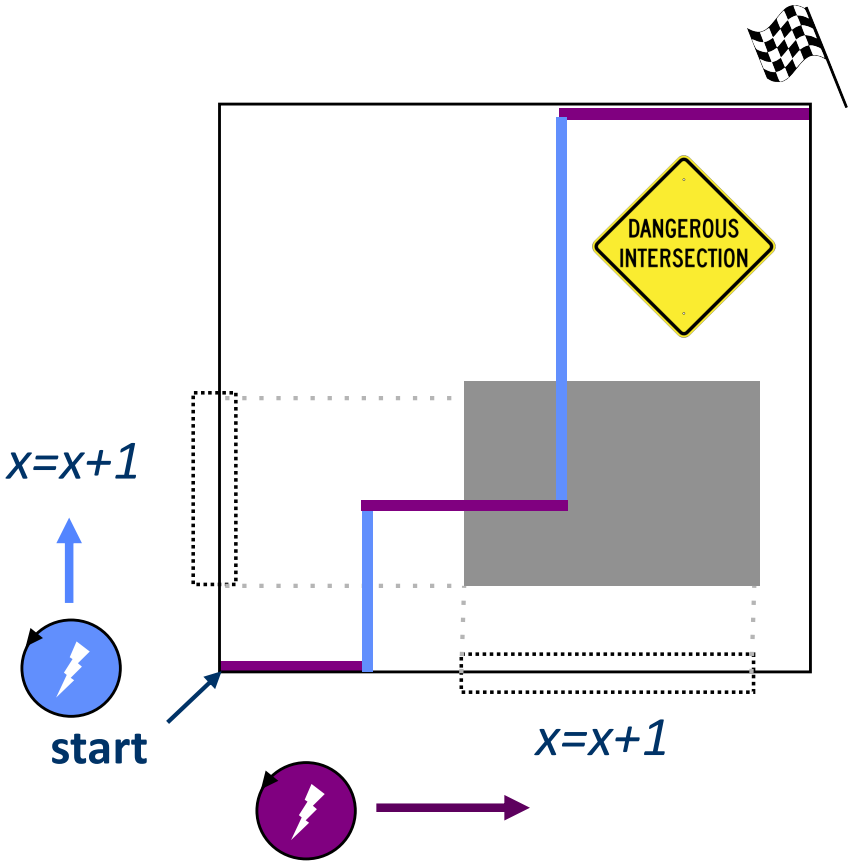
This valid schedule **interleaves executions** of the code "x = x+ 1" in the two threads.

The variable **x** is **shared**, like the counter in the pthreads example.

This schedule can corrupt the value of the shared variable x, causing the program to execute incorrectly.

This is an example of a **race**: the behavior of the program depends on the schedule, and some schedules yield incorrect results. A race results from **concurrent conflicting accesses**: two or more threads access the same shared location concurrently, and at least one of the accesses is a write.

Critical sections



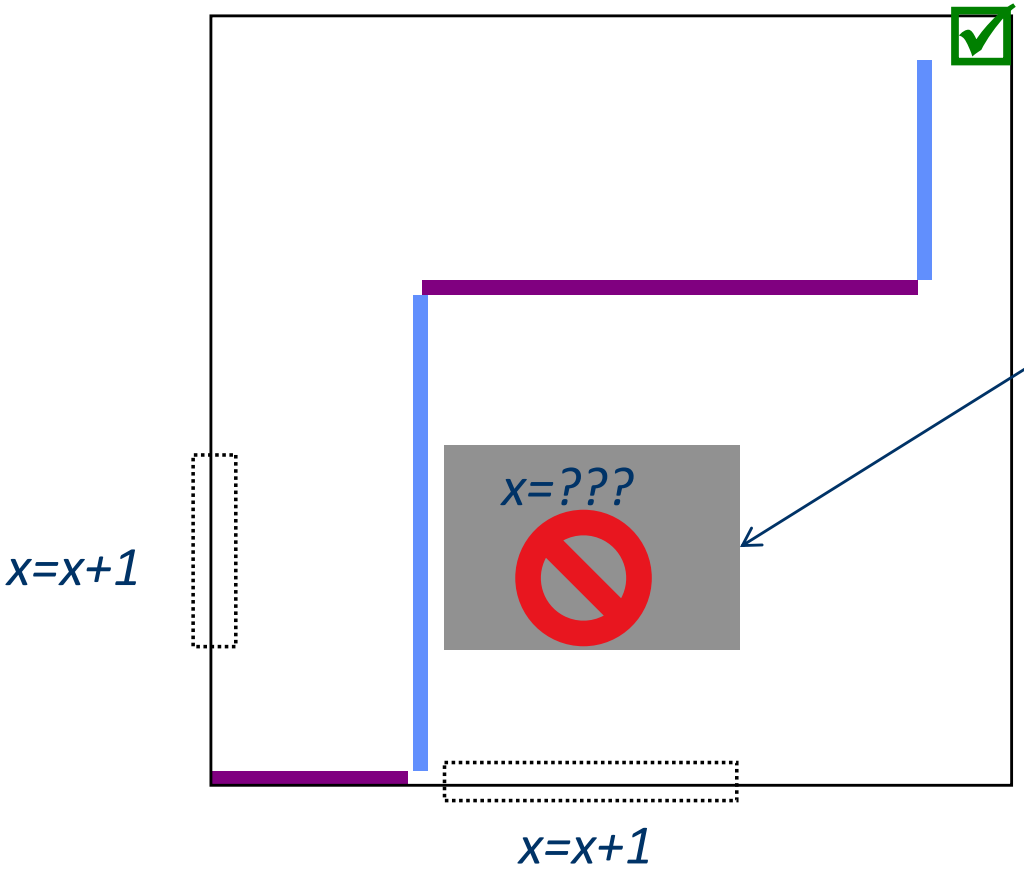
This schedule **interleaves executions** of the shaded code section ("x = x+ 1") in the two threads, creating an unsafe race.

Such a sensitive code piece is called a **critical section**: it accesses a shared location in a way that may result in a conflict with another thread executing its own critical section.

E.g., two threads reference a shared variable, and at least one of them writes to it.

We need a way to protect critical sections, so that threads do not execute **mutually critical** code sections **concurrently**.

The need for mutual exclusion



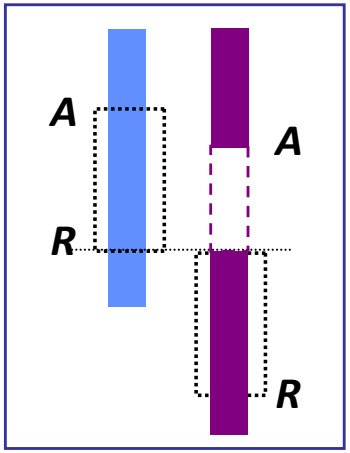
The program may fail if the schedule enters the grey box (i.e., if two threads execute the critical section concurrently).

The two threads must not both operate on the shared global x “at the same time”.

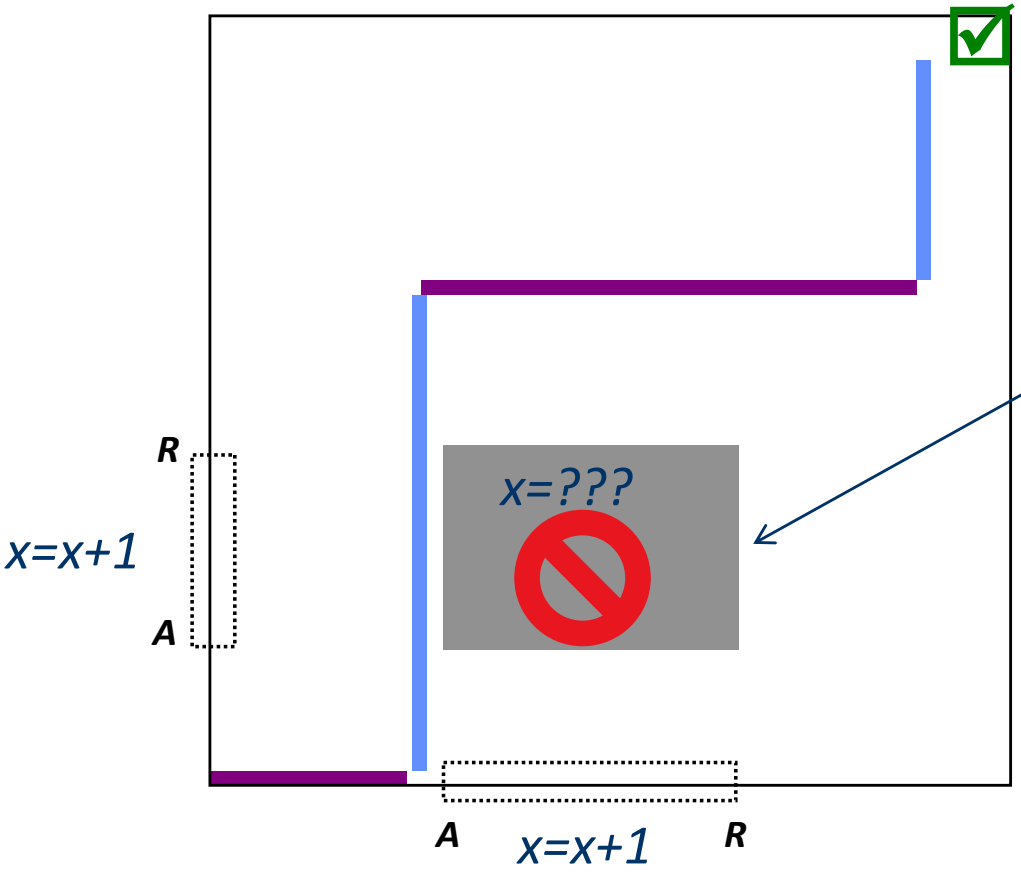
A Lock/Mutex/Monitor

Lock is a construct to enforce mutual exclusion in conflicting code sections (critical sections).

- A lock is a special data item/object in memory.
- API methods: Acquire() and Release().
- Also called Lock() and Unlock().
- *Acquire* upon entering a critical section.
- *Release* upon leaving a critical section.
- Between *Acquire*/*Release*, the thread **holds** the lock.
- *Acquire* **waits if another thread holds the lock**.
- Waiting locks can spin (a *spinlock*) or block (a *mutex*).
- Also called a **monitor**: threads enter (acquire) and exit (release).



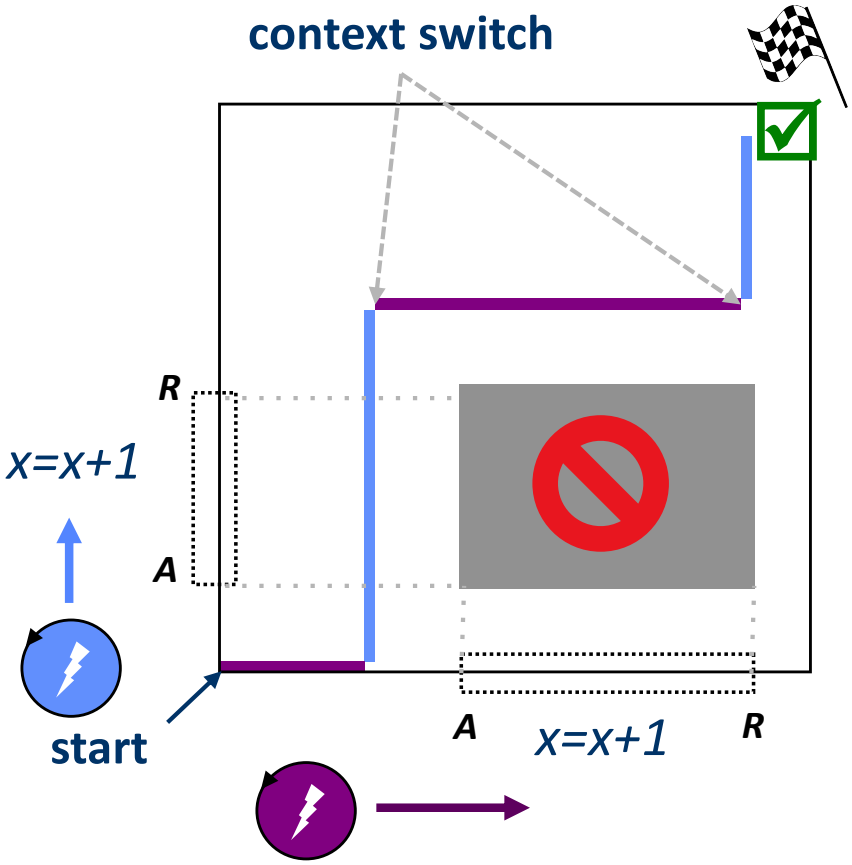
Using a lock/mutex



The program may fail if it enters the grey box.

A lock (mutex) prevents the schedule from ever entering the grey box, ever: both threads would have to hold the same lock at the same time, and locks don't allow that.

“Lock it down”



Use a **lock** (**mutex**) to synchronize access to a data structure that is shared by multiple threads.

A thread **acquires** (locks) the designated mutex before entering the critical section.

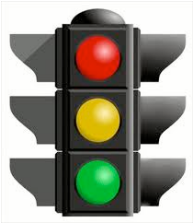
The thread **holds** the mutex. At most one thread can hold a given mutex at a time (**mutual exclusion**).

Thread **releases** (unlocks) the mutex when done. If other threads are waiting to acquire, then at least one of them wakes.

The mutex bars entry to the grey box: the threads cannot both hold the mutex.

Mutex: the contract

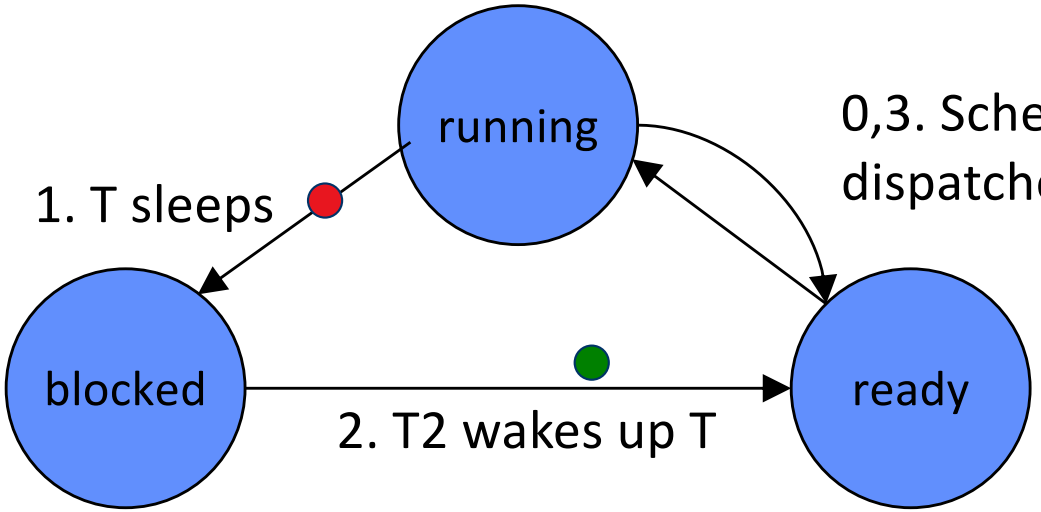
- Program promises that acquire+release ops on each lock *L* are strictly paired within each thread.
  - After acquire completes, the caller holds (owns) the lock *L* until the matching release.
- The lock promises that the acquire+release pairs on each *L* are ordered across threads.
  - Total order: each lock *L* has at most one holder at a time. Thus mutually critical sections run one at a time.
  - That property is mutual exclusion; *L* is a mutex.



Waiting for conditions

- We need more general synchronization primitives.
- In particular, we need some way for a thread to **sleep until some other thread or handler wakes it up**.
- This enables explicit signaling over any kind of condition, e.g., changes in the program state or state of a shared resource.
- Ideally, the threads don't have to know about each other explicitly. They should be able to coordinate around shared objects.

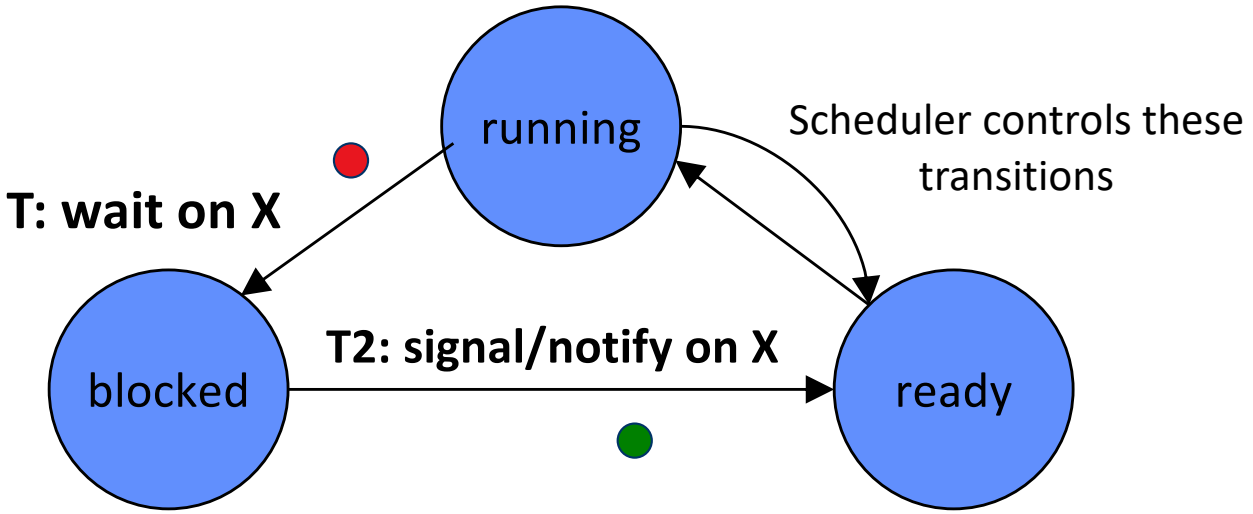
States and transitions for thread T



Waiting for conditions

- In particular, a thread might wait for some logical condition to become true. A **condition** is a predicate over state: it is any statement about the "world" that is either true or false.
  - Wait until a new event arrives; wait until event queue is not empty.
  - Wait for certain amount of time to elapse, then wake up at time t.
  - Wait for a network packet to arrive or an I/O operation to complete.
  - Wait for a shared resource (e.g., buffer space) to free up.
  - Wait for some other thread to finish some operation (e.g., initializing).

States and transitions for thread T



Condition variables (conditions)

- A **condition variable (CV)** is an object with an API.
  - **wait**: block until some condition becomes true
    - Not to be confused with Unix **wait*** system call
  - **signal** (also called **notify**): signal that the condition is true
    - Wake up one waiter.
- Every CV (or **condition**) is bound to exactly one mutex, which is necessary for safe use of the CV.
  - The mutex protects shared state associated with the condition
- A mutex may have any number of CVs bound to it.
- CVs also define a **broadcast** (**notifyAll**) primitive.
  - Signal all waiters.

Using conditions

Get from list

```
lock(mx);
while(list.empty())
      wait(mx, cv);
element = list.pop();
unlock(mx);
```

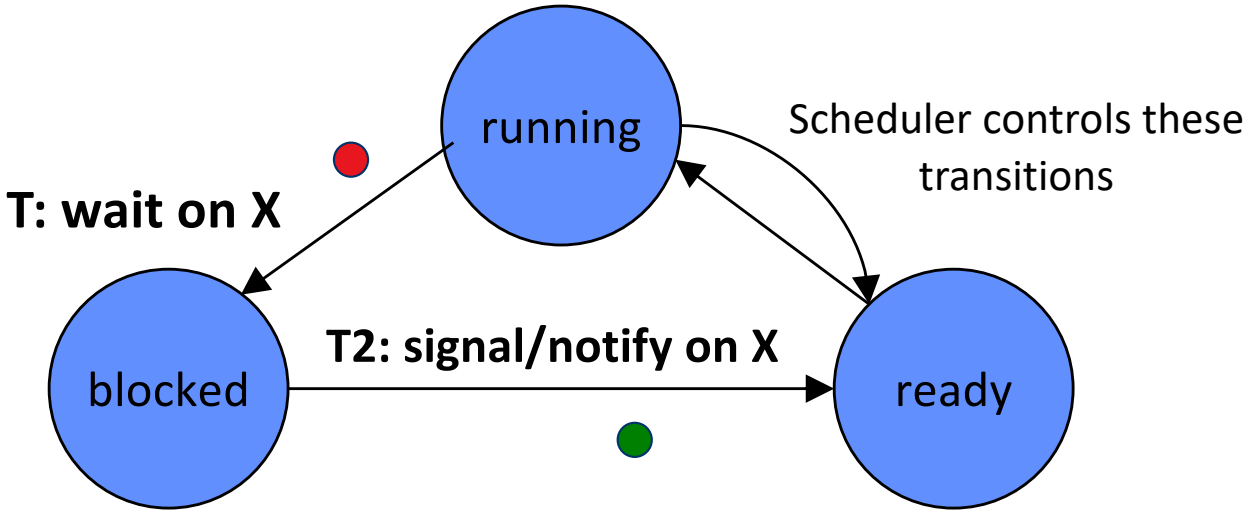
1. Must hold mutex to wait().
2. Wait() unlocks so another thread can work.
3. Wait() re-locks before return.

- It is automatic and atomic.
- Other threads run between.
- Loop before you leap! (**Mesa**)

Put to list

```
lock(mx);
list.push(element);
signal(mx, cv);
unlock(mx);
```

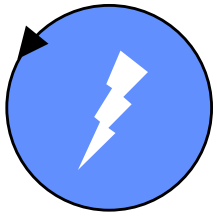
States and transitions for thread T



Threads project APIs

Threads

```
thread_create(func, arg);
thread_yield();
```



Locks/Mutexes

```
thread_lock(lockID);
thread_unlock(lockID);
```

Condition Variables

```
thread_wait(lockID, cvID);
thread_signal(lockID, cvID);
thread_broadcast(lockID, cvID);
```

Mesa monitors

All functions return an error code: 0 is success, else -1.

Java uses mutexes and CVs

Every Java object has a mutex ("monitor") and condition variable ("CV") built in. You don't have to use it, but it's there.

```
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 ("**synchronized**") to call wait/notify, else the method raises an *IllegalMonitorStateException*.

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

Mutual exclusion in Java

- Mutexes are built in to every Java object.
  - no separate classes
- Every Java object is/has a **monitor**.
  - At most one thread may "own" a monitor at any given time.
- A thread becomes **owner** of an object's monitor by
  - executing an object method declared as **synchronized**
  - executing a block that is **synchronized** on the object

```
public synchronized void increment()
{
        x = x + 1;
}
```

```
public void increment()  {
        synchronized(this) {
            x = x + 1;
        }
}
```