Using Synchronization

Dr. Daniel Andresen

CIS520

Avoiding Races #1

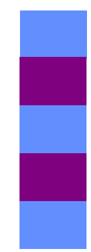
1. Identify *critical sections*, code sequences that:
    - rely on an invariant condition being true;
    - temporarily violate the invariant;
    - transform the data structure from one legal state to another;
    - or make a sequence of actions that assume the data structure will not "change underneath 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

Mutual Exclusion

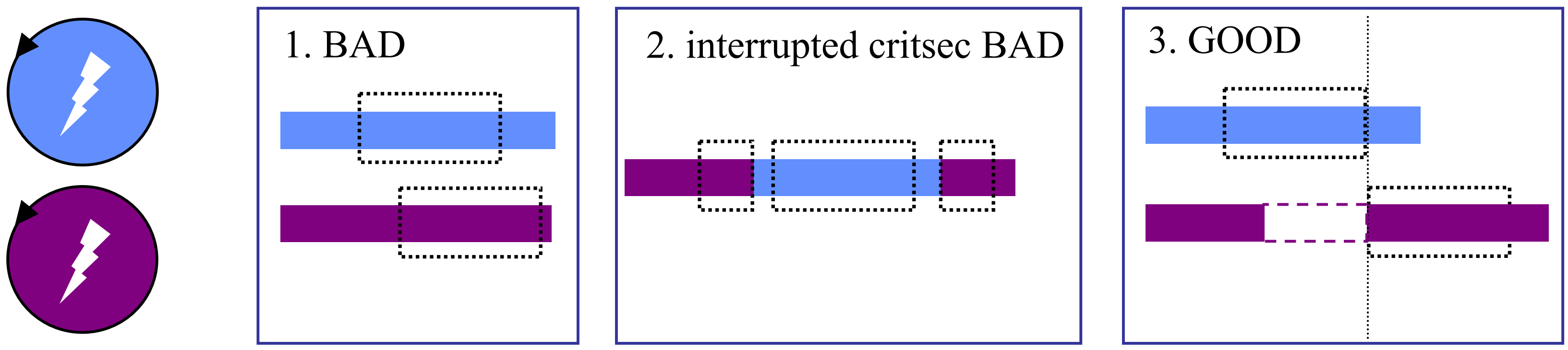
Race conditions can be avoiding by ensuring *mutual exclusion* in critical sections.

- Critical sections are code sequences that contribute 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.



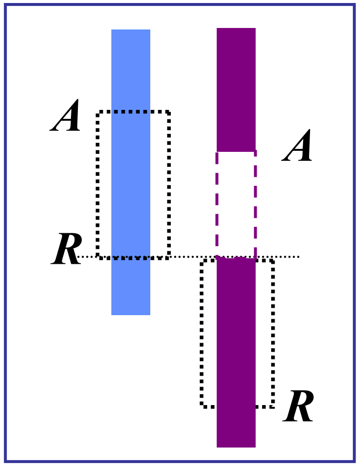
Locks

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*.

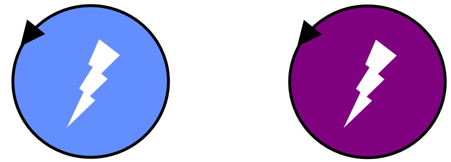
- Threads pair calls to *Acquire* and *Release*.
- *Acquire* before entering a critical section.
- *Release* after leaving a critical section.
- Between *Acquire*/*Release*, the lock is *held*.
- *Acquire* does not return until any previous holder releases.
- Waiting locks can spin (a *spinlock*) or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



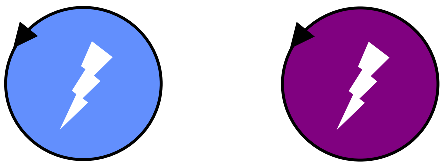
Example: Per-Thread Counts and Total

```
/* 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();
}
```

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.

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 as Mutexes

```
semapohore->Init(1);

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

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

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

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

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

P and *V* 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);
```

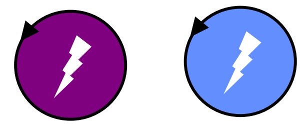
```
void
PingPong() {
    while(not done) {
        blue->P();
        Compute();
        purple->V();
    }
}
```

```
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();
        }
}
```



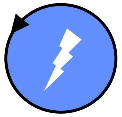
Nachos semaphores have Mesa-like semantics:
They do not guarantee that a waiting thread wakes up "in time" to consume the count added by a *V()*.

- semaphores are not "fair"
- no count is "reserved" for a waking thread
- uses "passive" vs. "active" implementation

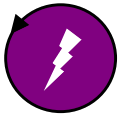
Another Example With Dual Semaphores

```
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 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);
}
```

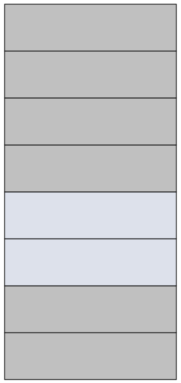
This use of a semaphore pair is called a *split binary semaphore*: the sum of the values is always one.

A Bounded Resource

```
int AllocateEntry() {
        int i;
        while (!FindFreeItem(&i))
                block and wait for a free slot
        slot[i] = 1;            /* grab free slot */
        return(i);
}

void ReleaseEntry(int i) {
        slot[i] = 0;
        wakeup waiter, if any
}

boolean FindFreeItem(int* index) {
        for (i = 0; i < TableSize; i++)
                if (slot[i] == 0)  return it;
        return (FALSE);
}
```



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?

Spin-Yield: Just Say No

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


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