Today

- Synchronization
  - Implementing Locks

Review

- What are these in the context of synchronization?
  - Liveness/Progress
  - Safety/Mutual Exclusion
- What is a Lock?
  - Why use locks?
  - What is its API? What do those method calls do?
  - What are the rules of Locks?
- Why is debugging concurrency/non-determinism difficult?

Review: Terminology

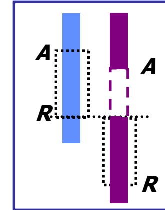
- ***Safety/Mutual Exclusion***: only one thread in the critical section
- ***Liveness/Progress***: if no threads are executing a critical section and a thread wishes to enter a critical section, that thread must be guaranteed to *eventually* enter the critical section
- ***Lock***: synchronization mechanism to prevent concurrent access (mutual exclusion)
  - Also called ***mutex*** or ***mutex lock***

Review: Locks

- `Acquire`
  - wait until lock is free, then take it
- `Release`
  - release lock, waking up anyone waiting for it

1. At most one lock holder at a time (safety)
2. If no one holding, `acquire` gets lock (progress)
3. If all lock holders finish and no higher priority waiters, waiter eventually gets lock (progress)

Review: A Lock or Mutex



- Locks enforce **mutual exclusion** in conflicting critical sections
- API methods: `Acquire` and `Release`
  - Also called Lock and Unlock
  - Call `Acquire` upon entering a critical section
  - Call `Release` upon leaving a critical section
- Between `Acquire`/`Release`, the thread holds the lock
- `Acquire` does not return until any previous holder releases

Review: Rules for Using Locks

- Lock is initially free
- **Always** `acquire` lock before accessing shared data structure
  - Likely: Beginning of procedure
- **Always** `release` after finished with shared data
  - Likely: End of procedure
  - Only the lock holder can release
- **Never** access shared data without lock

Review: Debugging non-determinism

- Requires worst-case reasoning
  - Eliminate all ways for program to break
- Debugging is hard
  - Can't test all possible interleavings
  - Bugs may only happen sometimes
- Heisenbug
  - Re-running program may make the bug disappear
  - Doesn't mean it isn't still there!

IMPLEMENTING LOCKS

Lock Goals

- What are our goals for locks?
  - That will help us to figure out how to implement them
- Consider a lock
  - For a highly contended resource
  - On a resource-strapped system

Lock Goals

- Must enforce mutual exclusion
- Reasonable fairness (liveness)
  - Does each thread contending for the lock get a fair shot at acquiring it once it is free?
  - Does any thread contending for the lock starve while doing so, thus never obtaining it?
- Reasonable performance
  - Overhead in using the lock
  - Scenarios: one thread acquiring/releasing lock, multiple threads/single CPU, multiple threads/multiple CPUs

Key Observations

- Why do we need mutual exclusion?
  - The scheduler!
- On a uniprocessor, a operation is *atomic* if no context switch can occur in the middle of the operation
- So, how about **mutual exclusion by preventing the context switch?**

What causes context switches?

Key Observations

- Why do we need mutual exclusion?
  - The scheduler!
- On a uniprocessor, a operation is *atomic* if no context switch can occur in the middle of the operation
  - **Mutual exclusion by preventing the context switch**
- Context switches occur because of
  - Internal events: systems calls and exceptions
  - External events: interrupts

Disabling Interrupts

Assume: single processor system

- Tells the hardware to delay handling any external events until after the thread is finished modifying the critical section
- In some implementations, done by setting and unsetting the interrupt status bit

Disabling Interrupts for Locks

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

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

Analyze the solution:
- Does it work?
- What are its strengths and weaknesses?

Disabling Interrupts for Locks

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

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

Works in that it enforces mutual exclusion but …

- Once interrupts are disabled, thread can't be stopped
  - Critical section can be very long
  - Can't wait too long to respond to interrupts → may be lost/missed
- Any program can call lock methods. So…
- Only works for single processor

Disabling Interrupts: Simple Solution

```
Lock::Acquire(){
  disable interrupts;
  while(value == BUSY){
    enable interrupts;
    disable interrupts;
  }
  value = BUSY;
  enable interrupts;
}
```

```
Lock::Release(){
  disable interrupts;
  value = FREE;
  enable interrupts;
}
```

Idea: Shorten the length of the critical section.
But then …?

Larger Question: Is this a good idea?

- Should *user* processes be able to disable interrupts?
  - No.
- What happens on multiprocessors?
  - Disabling interrupts affects only the CPU on which the thread is executing
    - Threads on other CPUs can enter the critical section!
  - Or, need to disable interrupts on all CPUs – expensive!
- On a uniprocessor, the OS may use this technique when it is updating kernel data structures

What are we trying to do?

- Ensure mutual exclusion, liveness, fairness, etc.
- But, practically?
  - See if another thread is executing the section (*read* a variable)
  - If it isn't, grab the lock (*modify* and *write* a variable)
  - If it is, wait
  - Atomically

Proposed Lock Implementation

ASSERT: if expression evaluates to 0,
Display error message and abort program

```
avail = 0;

acquire() {
      while (avail == 1)
           {;}
      ASSERT (avail == 0);
      avail = 1;
}

release() {
      ASSERT(avail == 1);
      avail = 0;
}
```

Global lock variable

Busy-wait until lock is free.

Spinlock: a First Try

ASSERT: if expression evaluates to 0,
Display error message and abort program

```
avail = 0;

acquire() {
      while (avail == 1)
           {;}
      ASSERT (avail == 0);
      avail = 1;
}

release() {
      ASSERT(avail == 1)
      avail = 0;
}
```

Global spinlock variable

Busy-wait until lock is free.

Spinlocks provide mutual exclusion among cores without blocking
→ don't need to context switch

Spinlocks are useful for lightly contended critical sections
where there is no risk that a thread is preempted while it is holding the lock

Spinlock: What Went Wrong

```
avail = 0;

acquire() {
    while (avail == 1)
        {;}
    ASSERT (avail == 0);
    avail = 1;
}

release() {
    ASSERT(avail == 1);
    avail = 0;
}
```

Race to acquire
Two (or more) cores may see avail == 0.

How do we fix this problem?

Hardware Support

- To implement mutual exclusion, we need support with a "magic toehold"
  - Lock primitives themselves have critical sections to test and/or set the lock flags.
- Safe mutual exclusion on multicore systems requires some hardware support: ***atomic instructions***
  - Examples: test-and-set, compare-and-swap, fetch-and-add.
    - Perform an ***atomic*** *read-modify-write* of a memory location
    - Expensive but necessary
  - If we have any of those atomic instructions, we can build higher-level synchronization objects.

Takeaway: Mutexes are often implemented using hardware

“Test and Set” Instruction

- Retrieve a value from memory and set the value at that location to 1; return the original value
- Atomic exchange
  - Simultaneously test old value (returned) and set a new value
- C Pseudocode:

```
int test_and_set(int *addr) {
    int result = *addr;
    *addr = 1;
    return result;
}
```


Implementing Locks with test_and_set

```
mutex = 0; // 0→ free, 1→ locked

Lock::Acquire(){
    while
   (test_and_set(mutex)==1)
       ;
}

Lock::Release(){
    mutex = 0;
}
```

Evaluate this implementation

- If lock is free (value==0), test_and_set reads 0, sets value to 1, and returns 0.
  - Lock is now locked
  - while condition is false, Acquire is complete
- If lock is busy (value==1), test_and_set reads 1, sets value to 1, and returns 1.
  - while continues to loop until a Release executes

Evaluating Spin Lock

- Provides mutual exclusion
- There is low latency to acquire the lock
  - If it becomes free, waiting thread gets it as soon as it is scheduled again
- No fairness guarantees
- Occupies CPU by performing busy waiting or *spinning*
  - Okay if critical section is much shorter than the scheduling quantum
- What happens if threads have different priorities?
  - If the thread waiting for the lock has higher priority than the thread using the lock?
  - Called the ***priority inversion*** problem
    - Possible whenever there is a busy wait

Less Spinning Solution

```
mutex = 0; // 0→ free, 1→ locked

Lock::Acquire(){
    while (test_and_set(mutex)==1)
        yield();
}

Lock::Release(){
    mutex = 0;
}
```

Voluntarily give up CPU

Evaluate this implementation

Less Spinning Solution

```
mutex = 0; // 0→ free, 1→ locked

Lock::Acquire(){
   while (test_and_set(mutex)==1)
      yield();
}

Lock::Release(){
   mutex = 0;
}
```

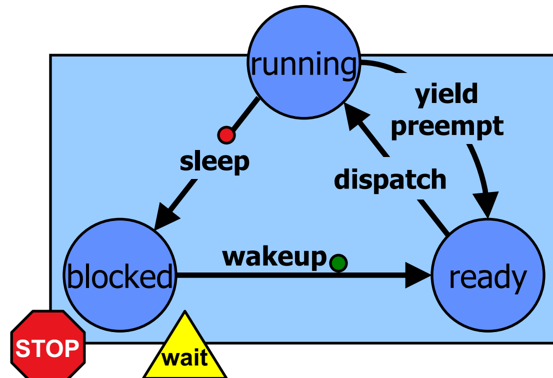
Voluntarily give up CPU

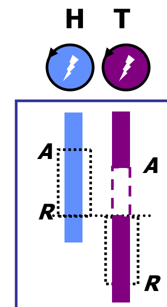
Evaluate this implementation

- Less busy waiting
- But, still no guarantees about fairness, starvation

Locking with blocking

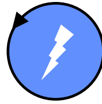
If thread **T** attempts to acquire a lock that is busy (held), **T** must **spin and/or block** (*sleep*) until the lock is free. By sleeping, **T** frees up the core for some other use. Just sitting and spinning is wasteful.



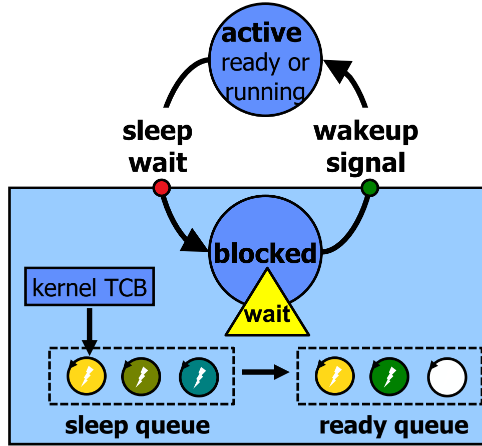


H is the lock holder when **T** attempts to acquire the lock.

OS Support for Blocking

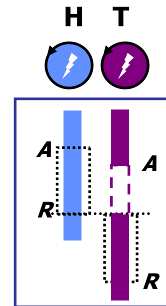


When a thread is **blocked** on a **synchronization object,** its TCB is placed on a ***sleep queue*** of threads waiting for an **event** on **that object**.

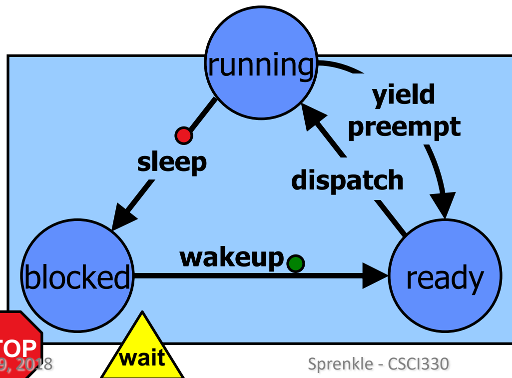


Applies to the process abstraction too,
or, more precisely, to the main thread of a process.

Locking with blocking

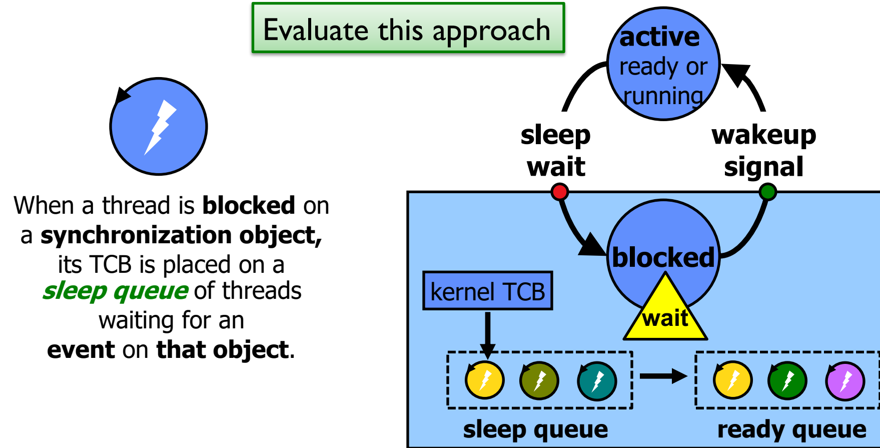


T calls acquire and enters the kernel (via syscall) to block because H has the lock.
T sleeps in the kernel to wait for the contended lock.
When the lock holder **H** *releases*, **H** enters the kernel (via syscall) to wakeup a waiting thread (e.g., **T**).



H can block too, perhaps for some other resource.
H doesn't implicitly release the lock just because it blocks.

OS Support for Blocking



Applies to the process abstraction too,
or, more precisely, to the main thread of a process.

Spinlocks vs Blocking/Queuing Locks

- Spinlocks
  - Useful in kernel for shared data structures (e.g., ready queue)
- Block/Queueing Locks
  - Can be more efficient for those waiting for acquire
    - Added on a queue in kernel mode
    - But, not busy waiting and consuming resources
  - Can prevent starvation

Looking Ahead

- Project 3 due Friday