CS 4410
Operating Systems

Synchronization
Locks - Semaphores

Summer 2016

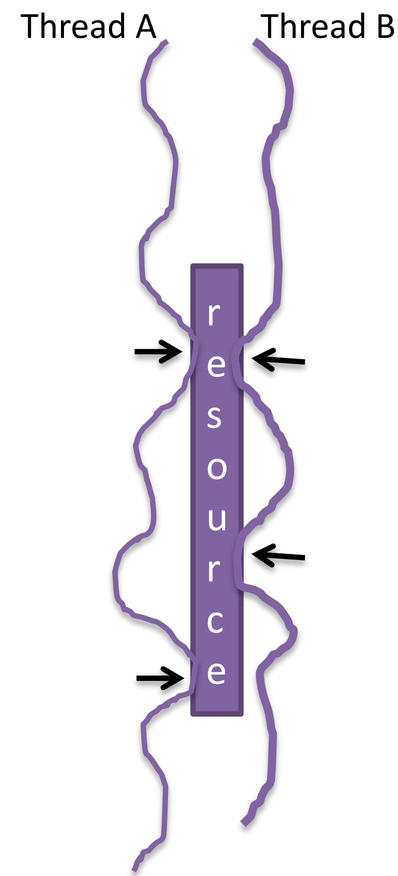
Cornell University

Today

- Need for synchronizing threads when they access shared data.
- Locks
- Semaphores

Racing for shared data

- Threads of the same process are not completely independent.
- Sometimes, they access shared data.
  - Shared data reside in the memory space shared by the threads.
- For a program to be correct, there might be some restrictions imposed on when threads are supposed to access shared data.
- It is hard to reason about when threads access shared data, due to:
  - preemptive scheduling,
  - multiprocessors.
- So, it is hard to reason about the satisfaction of these restrictions and the correctness of the program.



Example: Share Counting

- Mr Skroutz wants to count his $1-bills.
- Initially, he uses one thread that increases a variable *bills_counter* for every $1-bill.
- Then he thought to accelerate the counting by using two threads and keeping the variable *bills_counter* shared.

Share Counting

bills_counter = 0

- Thread A

  while (machine_A_has_bills)

  bills_counter++

- Thread B

  while (machine_B_has_bills)

  bills_counter++

print *bills_counter*

- Restriction: *bills_counter* should be updated by only one thread each time.
- Is this restriction satisfied?

Share Counting: A closer look

- Thread A

  r1 = *bills_counter*

  r1 = r1 +1

  bills_counter = r1

- Thread B

  r2 = *bills_counter*

  r2 = r2 +1

  bills_counter = r2

Possible executions

- Thread A

r1 = bills_counter

r1 = r1 +1

bills_counter = r1

- Thread B

r2 = bills_counter

r2 = r2 +1

bills_counter = r2

- If *bills_counter = 42, what are its possible values after the execution of one A/B loop ?*

Possible executions

- Thread A

r1 = bills_counter

r1 = r1 +1

bills_counter = r1

- Thread B

r2 = bills_counter

r2 = r2 +1

bills_counter = r2

- If *bills_counter = 42, what are its possible values after the execution of one A/B loop ?*

Share Counting: A closer look

- Thread A

  r1 = *bills_counter*

  r1 = r1 +1

  bills_counter = r1

- Thread B

  r2 = *bills_counter*

  r2 = r2 +1

  bills_counter = r2

- The restriction is not satisfied.
- The behavior of the program is unexpected. The program is not correct.

Need for synchronization

- For a multithreaded program to be correct,
  - some restrictions on accessing shared data by threads should be satisfied.
- Threads' access to shared resources should be coordinated.
- Assume resources themselves are not clever enough to know the restrictions (VS network card).
- Assume there is no entity that has global view of threads' execution and knows the restrictions (VS operating system).
- So, threads should coordinate on their own their access to shared data.
- All threads should still be able to make **progress**!

Critical Section

- Thread A

while (machine_A_has_bills)

> *r1 = bills_counter*
>
> *r1 = r1 +1*
>
> *bills_counter = r1*

- Thread B

while (machine_B_has_bills)

> *r2 = bills_counter*
>
> *r2 = r2 +1*
>
> *bills_counter = r2*

Critical Section

- Restriction rephrased: commands in critical section should be executed one after the other without interruption.

Lock: A synchronization primitive

- A thread must **acquire** a lock to enter a critical section.
  - Only one thread can acquire the lock at a time.
  - The thread **releases** the lock once it exits the critical section.
- Locks model restrictions on accessing shared data.
- Locks are themselves shared resources among threads.
  - But is it just the problem we want to solve?
- Access to locks through acquire and release actions is **atomic**.
- Atomic access to locks gives atomic access to critical sections!

Share Counting with lock

bills_counter = 0
lock = released

- Thread A

while (machine_A_has_bills)

acquire (lock)

r1 = bills_counter

r1 = r1 +1

bills_counter = r1

release (lock)

- Thread B

while (machine_B_has_bills)

acquire (lock)

r2 = bills_counter

r2 = r2 +1

bills_counter = r2

release (lock)

Critical Section

- Restriction rephrased: commands in critical section should be executed one after the other without interruption.

Achieving atomic access

- **TestAndSet** harware instruction.
  - **Test** and **modify** the content of **one word atomically**.

```
boolean TestAndSet(boolean *target){
        boolean rv = *target;
        *target = TRUE;
        return rv; }
```

- **Disable interrupts** before accessing a target.
  - Modify target (the modification procedure should be short and simple).
  - Enable interrupts after access.

Implementing a lock: an example

atomic

- Lock is a boolean variable.
- acquire(Lock): **while** (TestAndSet(&Lock)) skip;
- release(Lock) : *lock = FALSE;*

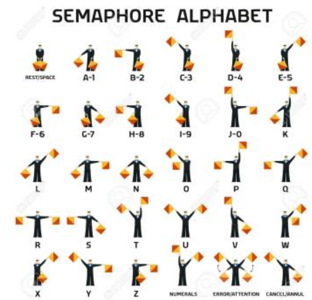
atomic

- Any implementation of acquire and release should be atomic!

Spinlock VS queuing lock

- This implementation of lock uses spinlock.
  - It requires **busy waiting**.
- Threads waiting to acquire the lock should **loop continuously** before the critical section.
- Valuable **CPU** cycles are **wasted**.
- Solution: queuing lock!
  - **Block** the waiting thread and add it in a waiting queue.
  - **Unblock** the first thread in the waiting queue and add it in the ready queue, when the lock is “available”.

Semaphore: synchronization primitive



- Semaphores: integer values
- A lock is abstracted by a semaphore S.
- Init(S,N): S=N
- P(S): while S <= 0 skip; S--;
- V(S): S++

Semantics.
Not real
implementation!

- Can be used for:
  - Mutual exclusion (mutex)
  - Condition synchronization (counter semaphor)

Synchronization: abstraction layers

Locks (acquire, release), semaphores (Init,P, V)
Spinlocks, queuing locks
TestAndSet, disable interrupts

Today

- Need for synchronizing threads when they access shared data.
- Locks
- Semaphores

Coming up…