Synchronization

Shared Memory Thread Synchronization

- Threads cooperate in multithreaded environments
  - Share resources and data structures
    - E.g. memory cache in a web server
  - Coordinate execution
    - Producer/consumer model
- For correctness, cooperation must be controlled
  - Must assume threads interleave execution arbitrarily
  - Control mechanism: synchronization
    - Allows restriction of interleaving
- Note: This is a global issue (kernel and user)
  - Also in distributed systems

Shared Resources

- Our focus: Coordinating access to shared resource
- Basic Problem:
  - Two concurrent threads access a shared variable
  - Access methods: Read-Modify-Write
- Two levels of approach:
  - Mechanisms for control
    - Low level locks
    - Higher level mutexes, semaphores, monitors, condition variables
  - Patterns for coordinating access
    - Bounded buffer, producer/consumer, …

The Classic Example

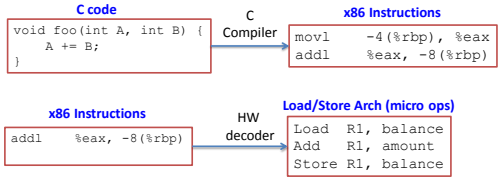
- Managing a bank account:

```
int balance;

void deposit(int amount) {
       balance += amount;
}

int withdraw(int amount){
       balance -= amount;
       return balance;
}
```

- Suppose account is shared between 2 people
  - What happens if both people go to ATMS and deposit $$?

But doesn't 'a += b' take care of it?

- A single C operation is not a single instruction
  - C operations are decomposed into multiple ASM instructions
  - Up to the compiler
- A single Instruction is not a single HW operation
  - X86 is load/store under the covers
  - So a "single" x86 instruction is actually a sequence of micro instructions



Cooperating Requires Synchronization

- Deposit implemented as sequence of ASM instructions

```
void deposit(int amount) {
        balance += amount;
}
```

→

```
Load  R1, balance
Add   R1, amount
Store R1, balance
```

- What happens if two threads call deposit at same time?

Process 1

```
Load  R1, balance
Add   R1, amount
Store R1, balance
```

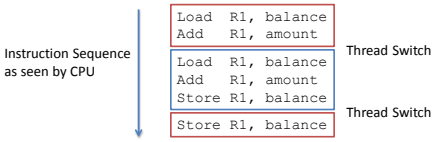
Process 2

```
Load  R1, balance
Add   R1, amount
Store R1, balance
```

- Race Condition: Result depends on operation interleaving

Interleaved Schedules

- Execution of the two threads can be interleaved



- What's the account balance after this sequence?

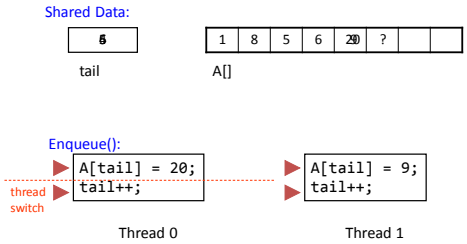
The crux of the matter

- Two concurrent threads access a shared resource without any synchronization
  - Creates a Race condition
    - Output is non-deterministic and depends on timing
- We need mechanisms for controlling access to shared resources
  - Allows us to reason about program operation
    - Re-introduces determinism
- Synchronization is necessary for any shared data structure

When are resources shared?

- Local variables are not shared
  - Refer to data on stack, each thread has its own stack
  - But… only if you don't give another thread a reference to a local variable
- Global variables are shared
  - Stored in static data segment
  - Globally accessible by any thread
- Dynamic objects are shared
  - Stored in the heap, shared if you have a pointer

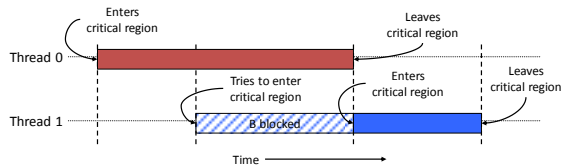
Race Condition



Mutual Exclusion

- Mutual exclusion synchronizes access to shared resources
- Code requiring mutual exclusion for correctness is a "critical section"
  - Only one thread at a time can execute in critical section
  - All other threads must wait to enter
  - Only when a thread leaves can another can enter

Critical Regions



Synchronization

- Scheduling can be random and preemption can happen at any time
- Need some way to make critical regions **"atomic"**
- Need help from lower layers
  - Operating System – system calls
  - Hardware – atomic instructions

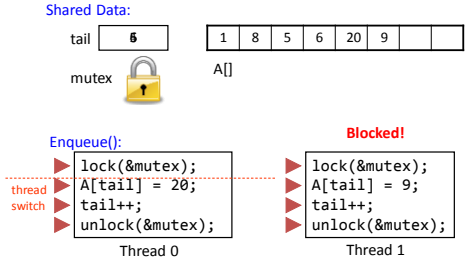
Mutex

- MUTual EXclusion
- A mutex is a lock that only one thread can acquire
- All other threads attempting to enter the critical region will be blocked

Blocking vs Busy Waiting

- What should happen when you try to acquire a locked mutex?
- Two approaches
  - Blocking
    - Put thread to sleep until mutex is released
    - Requires notification for when mutex is released
    - Good if lock hold times are long
    - Bad for latency
  - Busy waiting
    - Continue to run, polling the mutex
    - Good for quickly acquiring lock after its released
    - Wastes CPU time spinning on mutex (no forward progress)

Critical Sections



pthread_mutex_t

```
#include <stdio.h>
#include <pthread.h>

int tail = 0;
int A[20];

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void enqueue(int value)
{
        pthread_mutex_lock(&mutex);
        A[tail] = value;
        tail++;
        pthread_mutex_unlock(&mutex);
}
```

Producer/Consumer Problem

Shared variables

```
#define N 10;

int buffer[N];
int in = 0, out = 0, counter = 0;
```

Producer

```
while (1) {
  if (counter == N)
    sleep();

  buffer[in] = ... ;
  in = (in + 1) % N;

  counter++;

  if (counter == 1)
    wakeup(consumer);
}
```

Consumer

```
while (1) {
  if (counter == 0)
    sleep();

  ... = buffer[out];
  out = (out + 1) % N;

  counter--;

  if (count == N - 1)
    wakeup(producer);
}
```

Enforcing Mutual Exclusion

- How do you make sure every thread honors mutual exclusion?
  - **You Can't!!!**
- Mutual Exclusion is only possible if every thread honors a mutex
  - Programmers must acquire and release mutexes themselves
  - If a single thread does not, then mutual exclusion is broken

Deadlocks

- **When Locking Goes Wrong**
- "A set of processes is **deadlocked** if each process in the set is waiting for an event that only another process in the set can cause."
- Caused when:
  1. Mutual exclusion
  2. Hold and wait
  3. No preemption of resource
  4. *Circular wait*

Condition Variables

- A condition under which a thread executes or is blocked
- pthread_cond_t
- pthread_cond_wait (condition, mutex)
- pthread_cond_signal (condition)

Producer/Consumer

```
#define N 10
int buffer[N];
int counter = 0, in = 0, out = 0, total = 0;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t prod_cond = PTHREAD_COND_INITIALIZER;
pthread_cond_t cons_cond = PTHREAD_COND_INITIALIZER;
```

```
void *producer(void *junk) {
    while(1) {
        pthread_mutex_lock(&mutex);
        if( counter == N )
            pthread_cond_wait(&prod_cond,
                              &mutex);

        buffer[in] = total++;
        printf("Produced: %d\n",
                              buffer[in]);
        in = (in + 1) % N;
        counter++;

        if( counter == 1 )
            pthread_cond_signal(&cons_cond);

        pthread_mutex_unlock(&mutex);
    }
}
```

```
void *consumer(void *junk) {
    while(1) {
        pthread_mutex_lock(&mutex);

        if( counter == 0 )
            pthread_cond_wait(&cons_cond,
                              &mutex);

        printf("Consumed: %d\n",
                              buffer[out]);
        out = (out + 1) % N;
        counter--;

        if( counter == (N-1) )
            pthread_cond_signal(&prod_cond);

        pthread_mutex_unlock(&mutex);
    }
}
```

Semaphores

- Allow multiple threads inside a critical section
  - A semaphore is a lock with an associated count
  - Up to **count** threads can acquire semaphore simultaneously
- Mutexes are a special case of Semaphores that only count to 1
- Used in special cases
  - In other words rarely
- **#1 Recommendation for mutual exclusion**
  - **Keep it as simple as possible!!!**

Producer/Consumer

```
#include <semaphore.h>

#define N 10
int buffer[N];
int counter = 0, in = 0, out = 0, total = 0;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
sem_t semfull;  // sem_init(&semfull, 0, 0); in main()
sem_t semempty; // sem_init(&semempty, 0, N); in main()
```

```
void *producer(void *junk) {
    while(1) {
        sem_wait(&semempty);
        pthread_mutex_lock(&mutex);

        buffer[in] = total++;
        printf("Produced: %d\n",
                              buffer[in]);
        in = (in + 1) % N;
        counter++;

        pthread_mutex_unlock(&mutex);
        sem_post(&semfull);
    }
}
```

```
void *consumer(void *junk) {
    while(1) {
        sem_wait(&semfull);
        pthread_mutex_lock(&mutex);

        printf("Consumed: %d\n",
                              buffer[out]);
        out = (out + 1) % N;
        counter--;

        pthread_mutex_unlock(&mutex);
        sem_post(&semempty);
    }
}
```

Producer/Consumer

```
#include <semaphore.h>

#define N 10
int buffer[N];
int counter = 0, in = 0, out = 0, total = 0;

sem_t semmutex; // sem_init(&semmutex, 0, 1); in main()
sem_t semfull;  // sem_init(&semfull, 0, 0); in main()
sem_t semempty; // sem_init(&semempty, 0, N); in main()
```

```
void *producer(void *junk) {
    while(1) {
        sem_wait(&semempty);
        sem_wait(&semmutex);

        buffer[in] = total++;
        printf("Produced: %d\n",
                            buffer[in]);
        in = (in + 1) % N;
        counter++;

        sem_post(&semmutex);
        sem_post(&semfull);
    }
}
```

```
void *consumer(void *junk) {
    while(1) {
        sem_wait(&semfull);
        sem_wait(&semmutex);

        printf("Consumed: %d\n",
                                buffer[out]);
        out = (out + 1) % N;
        counter--;

        sem_post(&semmutex);
        sem_post(&semempty);
    }
}
```

Deadlock!

```
#include <semaphore.h>

#define N 10
int buffer[N];
int counter = 0, in = 0, out = 0, total = 0;

sem_t semmutex; // sem_init(&semmutex, 0, 1); in main()
sem_t semfull;  // sem_init(&semfull, 0, 0); in main()
sem_t semempty; // sem_init(&semempty, 0, N); in main()
```

```
void *producer(void *junk) {
    while(1) {
        sem_wait(&semmutex);
        sem_wait(&semempty);

        buffer[in] = total++;
        printf("Produced: %d\n",
                            buffer[in]);
        in = (in + 1) % N;
        counter++;

        sem_post(&semfull);
        sem_post(&semmutex);
    }
}
```

```
void *consumer(void *junk) {
    while(1) {
        sem_wait(&semmutex);
        sem_wait(&semfull);

        printf("Consumed: %d\n",
                                buffer[out]);
        out = (out + 1) % N;
        counter--;

        sem_post(&semempty);
        sem_post(&semmutex);
    }
}
```