Semaphores

- An abstract data type that can be used for condition synchronization and mutual exclusion

- **Condition synchronization**
  - *wait* until invariant holds before proceeding
  - *signal* when invariant holds so others may proceed

- **Mutual exclusion**
  - only one at a time in a critical section
Semaphores

- An abstract data type
  - containing an integer variable \( S \)
  - Two operations: Wait \( S \) and Signal \( S \)

- Alternative names for the two operations
  - \( \text{Wait}(S) = \text{Down}(S) = P(S) \)
  - \( \text{Signal}(S) = \text{Up}(S) = V(S) \)
Classical Definition of Wait and Signal

Wait(S)
{
    while S <= 0 do noop; /* busy wait! */
    S = S - 1; /* S >= 0 */
}

Signal (S)
{
    S = S + 1;
}
Problems with classical definition

- **Waiting threads hold the CPU**
  - Waste of time in single CPU systems
  - Required preemption to avoid deadlock
Blocking implementation of semaphores

Semaphore S has a value, S.val, and a thread list, S.list.

**Wait (S)**

S.val = S.val - 1

If S.val < 0 /* negative value of S.val */

{   add calling thread to S.list; /* is # waiting threads */
    block; /* sleep */
}

**Signal (S)**

S.val = S.val + 1

If S.val <= 0

{   remove a thread T from S.list;
    wakeup (T);
}
Using semaphores

- Semaphores can be used for mutual exclusion
  - Semaphore value initialized to 1
  - Wait on entry to critical section
  - Signal on exit from critical section
Using Semaphores for Mutex

semaphore mutex = 1  -- unlocked

Thread A

1 repeat
2   wait(mutex);
3   critical section
4   signal(mutex);
5   remainder section
6   until FALSE

Thread B

1 repeat
2   wait(mutex);
3   critical section
4   signal(mutex);
5   remainder section
6   until FALSE
Using Semaphores for Mutex

semaphore mutex = 0 -- locked

Thread A
1 repeat
2 \text{wait}(\text{mutex}); \downarrow
3 \text{critical section}
4 \text{signal}(\text{mutex});
5 \text{remainder section}
6 \text{until FALSE}

Thread B
1 repeat
2 \text{wait}(\text{mutex});
3 \text{critical section}
4 \text{signal}(\text{mutex});
5 \text{remainder section}
6 \text{until FALSE}
Using Semaphores for Mutex

semaphore mutex = 0 --locked

1 repeat
2 wait(mutex);
3 critical section
4 signal(mutex);
5 remainder section
6 until FALSE

Thread A

Thread B

1 repeat
2 wait(mutex);
3 critical section
4 signal(mutex);
5 remainder section
6 until FALSE
Using Semaphores for Mutex

\[ \text{semaphore mutex} = 0 \quad \text{--- locked} \]

**Thread A**

1. repeat
2. \text{wait}(\text{mutex});
3. \text{critical section}
4. \text{signal}(\text{mutex});
5. \text{remainder section}
6. \text{until FALSE}

**Thread B**

1. repeat
2. \text{wait}(\text{mutex});
3. \text{critical section}
4. \text{signal}(\text{mutex});
5. \text{remainder section}
6. \text{until FALSE}
Using Semaphores for Mutex

$\text{semaphore mutex} = 0 \quad \text{-- locked}$

**Thread A**

1. repeat
2. $\text{wait(mutex);}$
3. $\text{critical section}$
4. $\text{signal(mutex);}$
5. $\text{remainder section}$
6. until FALSE

**Thread B**

1. repeat
2. $\text{wait(mutex);}$
3. $\text{critical section}$
4. $\text{signal(mutex);}$
5. $\text{remainder section}$
6. until FALSE
Using Semaphores for Mutex

```
semaphore mutex = 1  -- unlocked

1 repeat
2     wait(mutex);
3     critical section
4     signal(mutex);
5     remainder section
6     until FALSE

This thread can now be released!

Thread A

1 repeat
2     wait(mutex);
3     critical section
4     signal(mutex);
5     remainder section
6     until FALSE

Thread B
```
Using Semaphores for Mutex

semaphore mutex = 0  -- locked

Thread A
1 repeat
2  wait(mutex);
3  critical section
4  signal(mutex);
5  remainder section
6  until FALSE

Thread B
1 repeat
2  wait(mutex);
3  critical section
4  signal(mutex);
5  remainder section
6  until FALSE
Using semaphores

- Semaphores can also be used to count accesses to a resource
  - Semaphore value is initialized to the number of successive waits that should succeed without blocking
Exercise: Implement producer/consumer

Global variables

```c
semaphore full_buffs = ?;
semaphore empty_buffs = ?;
char buff[n];
int InP, OutP;
```

0 thread producer {
1   while(1){
2     // Produce char c...
3     buf[InP] = c
4     InP = InP + 1 mod n
5   }
6 }

0 thread consumer {
1   while(1){
2     // Consume char...
3     OutP = OutP + 1 mod n
4   }
5 }
}
Exercise: Implement producer/consumer

Global variables
    semaphore full_buffs = 0;
    semaphore empty_buffs = n;
    char buff[n];
    int InP, OutP;

0 thread producer {
    while(1){
        // Produce char c...
        buf[InP] = c
        InP = InP + 1 mod n
    }
}

0 thread consumer {
    while(1){
        c = buf[OutP]
        OutP = OutP + 1 mod n
        // Consume char...
    }
}
Counting semaphores in producer/consumer

Global variables
semaphore full_buffs = 0;
semaphore empty_buffs = n;
char buff[n];
int InP, OutP;

0 thread producer {
    while(1){
        // Produce char c...
        wait(empty_buffs)
        buf[InP] = c
        InP = InP + 1 mod n
        signal(full_buffs)
    }
}

0 thread consumer {
    while(1){
        wait(full_buffs)
        c = buf[OutP]
        OutP = OutP + 1 mod n
        signal(empty_buffs)
        // Consume char...
    }
}
Implementing semaphores

- Wait () and Signal () are assumed to be atomic

*How can we ensure that they are atomic?*
Implementing semaphores

- Wait() and Signal() are assumed to be **atomic**

*How can we ensure that they are atomic?*

- **Implement Wait() and Signal() as system calls?**
  - how can the kernel ensure Wait() and Signal() are completed atomically?
  - avoid scheduling another thread when they are in progress?
  - ... but how exactly would you do that?
  - ... and what about semaphores for use in the kernel?
Semaphores with interrupt disabling

```c
struct semaphore {
    int val;
    list L;
};

Wait(semaphore sem) {
    DISABLE_INTS
    sem.val--
    if (sem.val < 0) {
        add thread to sem.L
        block(thread)
    }
    ENABLE_INTS
}

Signal(semaphore sem) {
    DISABLE_INTS
    sem.val++
    if (sem.val <= 0) {
        th = remove next
        thread from sem.L
        wakeup(th)
    }
    ENABLE_INTS
}
```
Semaphores with interrupt disabling

```c
struct semaphore {
    int val;
    list L;
}

Wait(semaphore sem) {
    DISABLE_INTS
    sem.val--
    if (sem.val < 0) {
        add thread to sem.L
        block(thread)
    }
    ENABLE_INTS
}

Signal(semaphore sem) {
    DISABLE_INTS
    sem.val++
    if (sem.val <= 0) {
        th = remove next thread from sem.L
        wakeup(th)
    }
    ENABLE_INTS
}
```
But what are block() and wakeup()?

- If block stops a thread from executing, how, where, and when does it return?
  - which thread enables interrupts following Wait()?
  - the thread that called block() shouldn’t return until another thread has called wakeup()!
  - … but how does that other thread get to run?
  - … where exactly does the thread switch occur?

- Scheduler routines such as block() contain calls to switch() which is called in one thread but returns in a different one!!
Thread switch

- If thread switch is called with interrupts disabled
  - where are they enabled?
  - ... and in which thread?
Semaphores using atomic instructions

- Implementing semaphores with interrupt disabling only works on uni processors
  - What should we do on a multiprocessor?

- As we saw earlier, hardware provides special atomic instructions for synchronization
  - test and set lock (TSL)
  - compare and swap (CAS)
  - etc

- Semaphore can be built using atomic instructions
  1. build mutex locks from atomic instructions
  2. build semaphores from mutex locks
Building *spinning* mutex locks using TSL

**Mutex_lock:**

```
TSL REGISTER,MUTEX | copy mutex to register and set mutex to 1
CMP REGISTER,#0    | was mutex zero?
JZE ok             | if it was zero, mutex is unlocked, so return
JMP mutex_lock     | try again
Ok: RET            | return to caller: enter critical section
```

**Mutex_unlock:**

```
MOVE MUTEX,#0      | store a 0 in mutex
RET                | return to caller
```
To block or not to block?

- **Spin-locks do busy waiting**
  - wastes CPU cycles on uni-processors
  - Why?

- **Blocking locks put the thread to sleep**
  - may waste CPU cycles on multi-processors
  - Why?
  - ... and we need a spin lock to implement blocking on a multiprocessor anyway!
Building semaphores using mutex locks

**Problem:** Implement a counting semaphore
  
  Up ()
  Down ()
  ...using just Mutex locks
How about two “blocking” mutex locks?

var cnt: int = 0 -- Signal count
var m1: Mutex = unlocked -- Protects access to “cnt”
m2: Mutex = locked -- Locked when waiting

Down():
Lock(m1)
cnt = cnt - 1
if cnt<0
  Unlock(m1)
  Lock(m2)
else
  Unlock(m1)
endIf

Up():
Lock(m1)
cnt = cnt + 1
if cnt<=0
  Unlock(m2)
endIf
Unlock(m1)
How about two "blocking" mutex locks?

```plaintext
var cnt: int = 0         -- Signal count
var m1: Mutex = unlocked -- Protects access to "cnt"
    m2: Mutex = locked   -- Locked when waiting

Down ():

    Lock(m1)
cnt = cnt - 1
if cnt<0
    Unlock(m1)
    Lock(m2)
else:
    Unlock(m1)
endIf

Up ():

    Lock(m1)
cnt = cnt + 1
if cnt<=0
    Unlock(m2)
    Unlock(m1)
else:
    Unlock(m1)
endIf
```

Contains a Race Condition!
Oops! How about this then?

```
var cnt: int = 0         -- Signal count
var m1: Mutex = unlocked -- Protects access to “cnt”
    m2: Mutex = locked   -- Locked when waiting

Down ():
   Lock(m1)
   cnt = cnt - 1
   if cnt<0
       Lock(m2)
       Unlock(m1)
   else
       Unlock(m1)
   endIf

Up ():
   Lock(m1)
   cnt = cnt + 1
   if cnt<=0
       Unlock(m2)
   endIf
   Unlock(m1)
```
Oops! How about this then?

var cnt: int = 0         -- Signal count
var m1: Mutex = unlocked -- Protects access to “cnt”
    m2: Mutex = locked   -- Locked when waiting

Down ():
    Lock(m1)
cnt = cnt - 1
    if cnt<0
        Lock(m2)
        Unlock(m1)
    else
        Unlock(m1)
    endIf
Unlock(m1)

Up ():
    Lock(m1)
cnt = cnt + 1
    if cnt<=0
        Unlock(m2)
    endIf
Unlock(m1)
Ok! Let's have another try!

var cnt: int = 0 -- Signal count
var m1: Mutex = unlocked -- Protects access to "cnt"
    m2: Mutex = locked -- Locked when waiting

Down ():

Lock(m2)
Lock(m1)
cnt = cnt - 1
if cnt>0
    Unlock(m2)
endIf
Unlock(m1)

Up ():

Lock(m1)
cnt = cnt + 1
if cnt=1
    Unlock(m2)
endIf
Unlock(m1)

... is this solution valid?
What about this solution?

Mutex m1, m2; // binary semaphores
int C = N;     // N is # locks
int W = 0;     // W is # wakeups

Down():
    Lock(m1);
    C = C – 1;
    if (C<0)
        Unlock(m1);
    Lock(m2);
    Lock(m1);
    W = W – 1;
    if (W>0)
        Unlock(m2);
    endif;
    else
        Unlock(m1);
        Unlock(m1);
    endif;

Up():
    Lock(m1);
    C = C + 1;
    if (C<=0)
        W = W + 1;
        Unlock(m2);
        else
            Unlock(m1);
            endif;
Classical Synchronization problems

- Producer Consumer (bounded buffer)
- Dining philosophers
- Sleeping barber
- Readers and writers
Producer consumer problem

- Also known as the bounded buffer problem
Is this a valid solution?

thread producer {
    while(1){
        // Produce char c
        while (count==n) {
            no_op
        }
        buf[InP] = c
        InP = InP + 1 mod n
        count++
    }
}

thread consumer {
    while(1){
        while (count==0) {
            no_op
        }
        c = buf[OutP]
        OutP = OutP + 1 mod n
        count--
        // Consume char
    }
}

Global variables:
char buf[n]
int InP = 0   // place to add
int OutP = 0  // place to get
int count
Does this solution work?

Global variables

semaphore full_buffs = 0;
semaphore empty_buffs = n;
char buff[n];
int InP, OutP;

0 thread producer {
  while(1) {
    // Produce char c...
    down(empty_buffs)
    buf[InP] = c
    InP = InP + 1 mod n
    up(full_buffs)
  }
}

0 thread consumer {
  while(1) {
    down(full_buffs)
    c = buf[OutP]
    OutP = OutP + 1 mod n
    up(empty_buffs)
    // Consume char...
  }
}
Producer consumer problem

- What is the shared state in the last solution?
- Does it apply mutual exclusion? If so, how?

Producer and consumer are separate threads
Dining philosophers problem

- Five philosophers sit at a table
- One fork between each philosopher

```c
while(TRUE) {
    Think();
    Grab first fork;
    Grab second fork;
    Eat();
    Put down first fork;
    Put down second fork;
}
```

- Why do they need to synchronize?
- How should they do it?

Each philosopher is modeled with a thread
Is this a valid solution?

```c
#define N 5

Philosopher() {
    while(TRUE) {
        Think();
        take_fork(i);
        take_fork((i+1) % N);
        Eat();
        put_fork(i);
        put_fork((i+1) % N);
    }
}
```
Working towards a solution ...

```c
#define N 5

Philosopher() { 
    while(TRUE) {
        Think();
        take_fork(i);
        take_fork((i+1)% N);
        Eat();
        put_fork(i);
        put_fork((i+1)% N);
    }
}
```

take_forks(i)

put_forks(i)
Working towards a solution ...

```c
#define N 5

Philosopher() {
    while(TRUE) {
        Think();
        take_forks(i);
        Eat();
        put_forks(i);
    }
}
```
Picking up forks

```c
int state[N]
semaphore mutex = 1
semaphore sem[i]

take_forks(int i) {
    wait(mutex);
    state[i] = HUNGRY;
    test(i);
    signal(mutex);
    wait(sem[i]);
}

// only called with mutex set!
test(int i) {
    if (state[i] == HUNGRY &&
        state[LEFT] != EATING &&
        state[RIGHT] != EATING){
        state[i] = EATING;
        signal(sem[i]);
    }
}
```
Putting down forks

```c
int state[N]
semaphore mutex = 1
semaphore sem[i]

put_forks(int i) {
    wait(mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    signal(mutex);
}

// only called with mutex set!
test(int i) {
    if (state[i] == HUNGRY &&
        state[LEFT] != EATING &&
        state[RIGHT] != EATING)
    {
        state[i] = EATING;
        signal(sem[i]);
    }
}
```
Dining philosophers

- Is the previous solution correct?
- What does it mean for it to be correct?
- Is there an easier way?
The sleeping barber problem
The sleeping barber problem

- **Barber:**
  - While there are people waiting for a hair cut, put one in the barber chair, and cut their hair
  - When done, move to the next customer
  - Else go to sleep, until someone comes in

- **Customer:**
  - If barber is asleep wake him up for a haircut
  - If someone is getting a haircut wait for the barber to become free by sitting in a chair
  - If all chairs are all full, leave the barbershop
Designing a solution

- How will we model the barber and customers?
- What state variables do we need?
  - .. and which ones are shared?
  - .... and how will we protect them?
- How will the barber sleep?
- How will the barber wake up?
- How will customers wait?
- What problems do we need to look out for?
Is this a good solution?

const CHAIRS = 5
var customers: Semaphore
barbers: Semaphore
lock: Mutex
numWaiting: int = 0

**Barber Thread:**
while true
    Wait(customers)
    Lock(lock)
    numWaiting = numWaiting-1
    Signal(barbers)
    Unlock(lock)
    CutHair()
endWhile

**Customer Thread:**
Lock(lock)
if numWaiting < CHAIRS
    numWaiting = numWaiting+1
    Signal(customers)
    Unlock(lock)
    Wait(barbers)
    GetHaircut()
else  -- give up & go home
    Unlock(lock)
endIf
The readers and writers problem

- Multiple readers and writers want to access a database (each one is a thread)
- Multiple readers can proceed concurrently
- Writers must synchronize with readers and other writers
  - *only one writer at a time!*
  - *when someone is writing, there must be no readers!*

Goals:
- Maximize concurrency.
- Prevent starvation.
Designing a solution

- How will we model the readers and writers?
- What state variables do we need?
  - .. and which ones are shared?
  - .... and how will we protect them?
- How will the writers wait?
- How will the writers wake up?
- How will readers wait?
- How will the readers wake up?
- What problems do we need to look out for?
Is this a valid solution to readers & writers?

```plaintext
var mut: Mutex = unlocked
db: Semaphore = 1
rc: int = 0

Reader Thread:
while true
    Lock(mut)
    rc = rc + 1
    if rc == 1
        Wait(db)
    endIf
    Unlock(mut)
    ... Read shared data...
    Lock(mut)
    rc = rc - 1
    if rc == 0
        Signal(db)
    endIf
    Unlock(mut)
    ... Remainder Section...
endWhile

Writer Thread:
while true
    ... Remainder Section...
    Wait(db)
    ... Write shared data...
    Signal(db)
endWhile
```
Readers and writers solution

- Does the previous solution have any problems?
  - is it “fair”?
  - can any threads be starved? If so, how could this be fixed?
  - ... and how much confidence would you have in your solution?
Quiz

- What is a race condition?
- How can we protect against race conditions?
- Can locks be implemented simply by reading and writing to a binary variable in memory?
- How can a kernel make synchronization-related system calls atomic on a uniprocessor?
  - Why wouldn’t this work on a multiprocessor?
- Why is it better to block rather than spin on a uniprocessor?
- Why is it sometimes better to spin rather than block on a multiprocessor?
Quiz

- When faced with a concurrent programming problem, what strategy would you follow in designing a solution?
- What does all of this have to do with Operating Systems?