Plan

- Project: Due
  - » Demos following week (Wednesday during class time)
- Next Week –
  - » New phase of presentations
  - » Deadlock, finish synchronization
- Course Progress:
  - » History, Structure, Processes, Threads, IPC, Synchronization
  - » Remainder: Deadlock, Memory and File

Demos

- Next next Wednesday Demos
- Preparation (show & tell)
  - » 3 precompiled kernels (original, lottery, stride, dynamic)
  - » 1 prepared document to tell me what is working what is not – overview what you did (5 minutes), script, and hand-in Tuesday
- How will it work (details)
  - » Show Data Structures in code
  - » Show Functionality added in code/kernel
  - » Show that it runs
  - » Demonstrates your testing & evaluation strategy
  - » Compile (not) & run (this will be done last)

Next Project

- Add System Call (again, yes)
- Add a service (see how this fit in shortly)
  - » It is a synchronization service (semaphore or monitor)
  - » Waking up and putting processes to sleep
- Write a simple application program that use this new service.


CSCI [4|6]730
Operating Systems

Synchronization Part 2

Process Synchronization Part II

- How does hardware facilitate synchronization?
- What are problems of the hardware primitives?
- What is a spin lock and when is it appropriate?
- What is a semaphore and why are they needed?
- What is the Dining Philosophers Problem and what is ‘a good’ solution?

Hardware Primitives

*Many modern operating systems provide special synchronization hardware to provide more powerful **atomic** operations*

- `testAndSet( lock )`
  - » atomically reads the original value of lock and then sets it to true.
- `Swap( a, b )`
  - » atomically swaps the values
- `compareAndSwap( a, b )`
  - » atomically swaps the original value of lock and sets it to true when they values are different
- `fetchAndAdd( x, n )`
  - » atomically reads the original value of x and adds n to it.

Hardware: testAndSet();

```
boolean testAndSet ( boolean *lock )
  {
  boolean old_lock = lock ;
  lock = true;
  return old_lock;
  }
```

```
// initialization
lock = false ; // shared -- lock is available
void deposit( int amount )
   {
   // entry to critical section - get the lock
   while( testAndSet( &lock ) == true ) {} ;
   balance += amount // critical section
   // exit critical section - release the lock
   lock = false;
   }
```

- If someone has the lock (it returns TRUE) and wait until it is available (until some-one gives it up, sets it to false).
- Atomicity guaranteed - even on multiprocessors

Hardware: Swap();

```
void Swap( boolean *a, boolean *b )
  {
  boolean temp = *a ;
  *a = *b;
  *b = temp;
  }
```

```
// initialization
lock = false ; // global shared -- lock is available
void deposit( int amount )
   {
   // entry critical section - get local variable key
   key = true; // key is a local variable
   while( key == true ) Swap( &lock, &key );
   balance += amount // critical section
   // exit critical section - release the lock
   lock = false;
   }
```

- Two Parameters: a global and local (when lock is available (false) get local key (false)).
- Atomicity guaranteed - even on multiprocessors
- Bounded waiting?
  - » No! How to provide?

Hardware with Bounded Waiting

- Need to **create** a waiting line.
- **Idea**: "Dressing Room" is the critical section, only one person can be in the room at one time, and **one waiting line** outside dressing room that serves customer first come first serve.
  - » `waiting[n]` : Global shared variable
  - » `lock`: Global shared variable
- Entry get a local variable 'key' and check via `testAndSet()` if someone is 'in' the dressing room

Hardware with Bounded Waiting

```
// initialization
lock = false ;   // shared -- lock is available
waiting[0.. n-1] = {false} ; // shared -- no one is waiting
void deposit( int amount )
   {
   // entry to critical section
   waiting[tid] = true; // signal tid is waiting
   key = true; // local variable
   while( ( waiting[tid] == true ) and ( key == true ) )
         key = testAndSet( &lock );
   waiting[tid] = false; // got lock done waiting
   balance += amount      // critical section

   // exit critical section - release the lock
   j = (tid + 1) mod n; // j is possibly waiting next in line
   while( ( j != tid ) and ( waiting[j] == false ) )
         j = (j + 1) mod n;   // check next if waiting
   if( j == tid )                // no one is waiting unlock room
         lock = false;
   else
         waiting[j] = false  // hand over the key to j
   }
```


Hardware Solution: Proof "Intuition"

- **Mutual Exclusion:**
  - » A thread **enters** only if it is waiting or if the dressing room is unlocked
    - – First thread to execute `testAndSet( &lock )` gets the lock all others will wait
    - – Waiting becomes false only if the thread with the lock leaves its CS and only one waiting is set to false.
- **Progress:**
  - » Since an exiting thread either unlocks the dressing room or hands the 'lock' to another thread progress is guaranteed because both allow a waiting thread access to the dressing room
- **Bounded Waiting:**
  - » Leaving threads scans the waiting array in cyclic order thus any waiting thread enters the critical section within n-1 turns.

Synchronization Layering

- Build higher-level synchronization primitives in OS
  - » Operations that ensure correct ordering of instructions across threads
- **Motivation**: Build them once and get them right
  - » Don't make users write entry and exit code

Monitors ☺ *Locks
Condition Variables *Semaphores

Loads Stores Test&Set
Disable Interrupts

Locks

- **Goal**: Provide mutual exclusion (mutex)
  - » The other criteria for solving the critical section problem may be violated
- Three common operations:

Allocate and Initialize

```
pthread_mutex_t mylock;
mylock = PTHREAD_MUTEX_INITIALIZER;
```

Acquire

Acquire exclusion access to lock; Wait if lock is not available

```
pthread_mutex_lock( &mylock );
```

Release

Release exclusive access to lock

```
pthread_mutex_unlock( &mylock );
```


Lock Examples

- After lock has been allocated and initialized

```
void deposit( int amount )
  {
  pthread_mutex_lock( &my_lock );
  balance += amount; // critical section
  pthread_mutex_unlock( &my_lock );
  }
```

- One lock for each bank account (maximize concurrency)

```
void deposit( int account_tid, int amount )
  {
  pthread_mutex_lock( &locks[account_tid] );
  balance[account_tid] += amount; // critical section
  pthread_mutex_unlock( &locks[account_tid] );  }
```


Implementing Locks: Atomic loads and stores

```
typedef struct lock_s
  bool lock[2] = {false, false};
  int turn = 0;

void acquire( lock_s *lock )
  lock->lock[tid] = true;
  turn  = 1-tid;
  while( lock->lock[1-tid] && lock->turn ==1-tid )

void release( lock_s lock )
  lock->lock[tid] = false;
```

- Disadvantage: Two threads only

Implementing Locks: Hardware Instructions (now)

```
typedef boolean lock_s;

void acquire( lock_s *lock )
  while( true == testAndSet( theLock ) ) {} ; // wait

void release( lock_s lock )
  lock = false;
```

- **Advantage**: Supported on multiple processors
- **Disadvantages**:
  - » Spinning on a lock may waste CPU cycles
  - » The longer the CS the longer the spin
    - – Greater chance for lock holder to be interrupted too!

Implementing Locks: Disable/Enable Interrupts

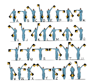
```
void acquire( lock_s *lock )
  disableInterrupts();

void release( lock_s lock )
  enableInterrupts();
```

- **Advantage**: Supports mutual exclusion for many threads (prevents context switches)
- **Disadvantages**:
  - » Not supported on multiple processors,
  - » Too much power given to a thread (may not release lock_
  - » May miss or delay important events

Spin Locks and Disabling Interrupts

- Spin locks and disabling interrupts are useful only for short and simple critical sections (not computational or I/O intensive):
  - » Wasteful otherwise
  - » These primitives are *primitive* -- don't do anything besides mutual exclusion (doesn't 'solve' the critical section problem).
- Need a higher-level synchronization primitives that:
  - – Block waiters
  - – Leave interrupts enabled within the critical section
  - » All synchronization requires atomicity
    - – So we' ll use our "atomic" locks as primitives to implement them

Semaphores

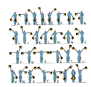


- Semaphores are another data structure that provides mutual exclusion to critical sections
  - » Described by Dijkstra in the THE system in 1968
  - » Key Idea: A data structure that counts number of "wake-ups" that are saved for future use.
    - – Block waiters, interrupts enabled within CS
- Semaphores have two purposes:
  - » Mutual Exclusion: Ensure threads don't access critical section at same time
  - » Scheduling constraints (ordering) Ensure thhat threads execute in specific order (implemented by a waiting queue).

Blocking in Semaphores

- Idea: Associated with each semaphore is a queue of waiting processes (typically the ones that want to get into the critical section)
- `wait()` tests (probes) the semaphore (DOWN) (wait to get in).
  - » If semaphore is open, thread continues
  - » If semaphore is closed, thread blocks on queue
- `signal()` opens (verhogen) the semaphore (UP): (lets others in)
  - » If a thread is waiting on the queue, the thread is unblocked
  - » If no threads are waiting on the queue, *the signal is remembered for the next thread (i.e., it stores the "wake-up").*
    - – signal() has history
    - – This 'history' is a counter

Semaphore Operations



- Allocate and Initialize
  - » Semaphore contains a non-negative integer value
  - » User cannot read or write value directly after initialization
    - – `sem_t sem;`
    - – `int sem_init( &sem, is_shared, init_value );`
- `wait()` … or *test* or *sleep* or *probe* or *down (block)* or *decrement*.
  - » `P()` for "test" in Dutch (proberen) also `down()`
  - » Waits until semaphore is open (`sem>0`) then decrement `sem` value
    - – `int sem_wait( &sem );`
- `signal()` … or *wakeup* or *up* or *increment* or *post*. (done)
  - » V() for "increment" in Dutch (verhogen) also `up()`, `signal()`
  - » Increments value of semaphore, allow another thread to enter
    - – `int sem_post(&sem);`

A Classic Semaphore

```
typedef struct
        {
        int value;             // Initialized to #resources available
        }semaphore;

sem_wait( semaphore *S )       // Must be executed atomically
        while S->value <= 0;
        S->value--;

sem_signal( semaphore *S )     // Must be executed atomically
        S->value++;
```

- S->value = 0 indicates all resources are exhausted/used.
  - » Note that S->value is never negative here (it spins), this is the classic definition of a semaphore
- Assumption: That there is atomicity between all instructions within the semaphore functions and across (incrementing and the waking up – i.e., you can't perform wait() and signal () concurrently.

Semaphore Implementation (that avoids *busy waiting*)
System V & Linux Semaphores

```
typedef struct
        {
        int value;
        queue tlist;           // blocking list of 'waiters'
        } semaphore;

sem_wait( semaphore *S )       // Must be executed atomically
        S->value--;
        if( S->value < 0 )
                add this process to S->tlist;
                block();

sem_signal( semaphore *S )     // Must be executed atomically
        S->value++;
        if( S->value <= 0 )     // Threads are waiting
                remove thread t from S->tlist;
                wakeup(t);
```


Semaphore Example

What happens when ***sem.value*** *is initialized to 2?*

Assume ***three*** *threads call sem_wait(&sem)*

```
typedef struct {
        int value;      /* initialize to 2 */
        queue tlist;
} semaphore;
sem_wait( semaphore *S )
        S->value--;
        if (S->value < 0)
           add calling thread to S->tlist;
           block();

sem_signal( semaphore *S )
        S->value++;
        if (S->value <= 0)
          remove a thread t from S->tlist;
          wakeup(t);
```

- Observations?
  - `sem` value is negative (what does the magnitude mean)?
    - – Number of waiters on queue
  - » `sem` value is positive? What does this number mean, e.g., What is the largest possible value of the semaphore?
    - – Number of threads that can be in critical section at the same time

Mutual Exclusion with Semaphores

- **Previous example with locks:**

```
void deposit( int amount )
  {
  pthread_mutex_lock( &my_lock );
  balance += amount; // critical section
  pthread_mutex_unlock( &my_lock );
  }
```

- **Example with Semaphore:**

```
void deposit( int amount )
  {
  sem_wait( &sem );
  balance += amount; // critical section
  sem_post( &sem );
  }
```

*What value should **`sem`** be initialized to provide ME?*

Beware: OS Provided Semaphores

- **Strong Semaphores: Order in semaphore is specified (what we saw, and what most OSs use). FCFS.**
- **Weak Semaphore: Order in semaphore definition is left unspecified**

- *Something to think about:*
  - » *Do these types of semaphores **solve** the Critical Section Problem? Why or Why not?*

Danger Zone Ahead

Dangers with Semaphores

- **Deadlock:**
  - » **Two or more threads are waiting indefinitely for an event that can be caused by only one of the waiting processes**
- **Example:**
  - » **Two threads: Maria and Tucker**
  - » **Two semaphores: `semA`, and `semB` both initialized to 1**

Thread Maria	Thread Tucker

`sem_wait( semA )`
`sem_wait( semB )`
 	
`sem_post( semA );`
`sem_post( semB );`
	`sem_wait( semB )`
	`sem_wait( semA )`
	
	`sem_post( semB );`
	`sem_post( semA );`

Semaphore Jargon

- **Binary semaphore is sufficient to provide mutual exclusion (restriction)**
  - » **Binary semaphore has boolean value (not integer)**
  - » **bsem_wait(): Waits until value is 1, then sets to 0**
  - » **bsem_signal(): Sets value to 1, waking one waiting process**
- **General semaphore is also called counting semaphore.**

Semaphore Verdict

- **Advantage:**
  - » **Versatile, can be used to solve any synchronization problems!**
- **Disadvantages:**
  - » **Prone to bugs (programmers' bugs)**
  - » **Difficult to program: no connection between semaphore and the data being controlled by the semaphore**
- **Consider alternatives: Monitors, for example, provides a better connection (data, method, synchronization)**

- Next will look at:
  - » synchronization problems &
  - » start on deadlock

Classes of Synchronization Problems (Thu)

- Uniform resource usage with simple scheduling constraints
  - » No other variables needed to express relationships
  - » Use one semaphore for every constraint
  - » Examples: producer/consumer
- Complex patterns of resource usage
  - » Cannot capture relationships with only semaphores
  - » Need extra state variables to record information
  - » Use semaphores such that
    - – One is for mutual exclusion around state variables
    - – One for each class of waiting
- Always try to cast problems into first, easier type

Classical Problems: *Readers Writers*

Set of problems where data structures, databases or file systems are read and modified by concurrent threads

- Idea:
  - » While data structure is updated (write) often necessary to bar other threads from reading
- *Basic* Constraints (Bernstein's Condition):
  - » Any number of readers can be in CS simultaneously
  - » Writers must have exclusive access to CS
- Some Variations:
  - » First Readers: No reader kept waiting unless a writer already in CS - so no reader should wait for other readers if a writer is waiting already (reader priority)
  - » Second Readers: Once a writer is ready the writer performs write as soon as possible (writer priority)

First Readers: Initialization

- Reader priority
- First readers: simplest reader/writer problem
  - » requires no reader should wait for other readers to finish even if there is a writer waiting.
  - » Writer is easy – it gets in if the room is available
- Two semaphores both initialized to 1
  - » Protect a counter
  - » Keep track whether a "room" is empty or not

```
int reader = 0              // # readers in room
sem_t mutex;                // 1 available - mutex to protect counter
sem_t roomEmpty;            // 1 (true) if no threads and 0 otherwise
int sem_is_shared = 0;      // both threads accesses semaphore

sem_init( &mutex, sem_is_shared, 1 );
sem_init( &roomEmpty, sem_is_shared, 1 );
```


First Reader: Entrance/Exit *Writer*

```
void enterWriter()
  sem_wait
(&roomEmpty)
```

```
void exitWriter()
    sem_post( &roomEmpty );
```

- Writer can go if the room is empty (unlocked)

First Reader: Entrance/Exit *Reader*

```
void enterReader()
  sem_wait(&mutex);
    reader++;
    if( reader == 1 )
       sem_wait( &roomEmpty );  // first in locks
  sem_post( &mutex );
```

```
void exitReader()
  sem_wait(&mutex) ;
    reader--;
    if( reader == 0 )
      sem_post( &roomEmpty );  // last out unlocks
  sem_post( &mutex );
```

- Only ONE reader is queued on roomEmpty, but several writers may be queued
- When a reader signals roomEmpty no other readers are in the room (the room is empty, key unlocked)

Evaluation: First Reader

```
void enterReader()
  sem_wait(&mutex)
    reader++;
    if( reader == 1 )
       sem_wait( &roomEmpty );  // first on in locks
  sem_post( &mutex );
```

```
void enterWriter()
  sem_wait(&roomEmpty)
```

```
void exitWriter()
  sem_post(&roomEmpty);
```

```
void exitReader()
  sem_wait(&mutex)
    reader--;
    if( reader == 0 )
       sem_post( &roomEmpty );  // last unlocks
  sem_post( &mutex );
```

- Only one reader is queued on roomEmpty
- When a reader signals roomEmpty no other readers are in the room
- Writers Starve? Readers Starve? Both?

Food for though

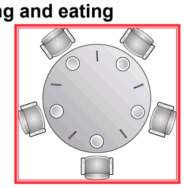
- How would you implement Second Reader?

Classical Problems: *Dining Philosophers*

Classic Multiprocess synchronization that stemmed from five computers competing for access to five shared tape drive peripherals.

- **Problem Definition Statement:**
  - » N Philosophers sit at a round table
  - » Each philosopher shares a chopstick (a shared resource) with neighbor
  - » Each philosopher must have both chopsticks to eat
  - » Immediate Neighbors can't eat simultaneously
  - » Philosophers alternate between thinking and eating

```
void philosopher( int i )
  while(1)
     think()
     take_chopstick(i);
     eat();
     put_chopstick(i);
```



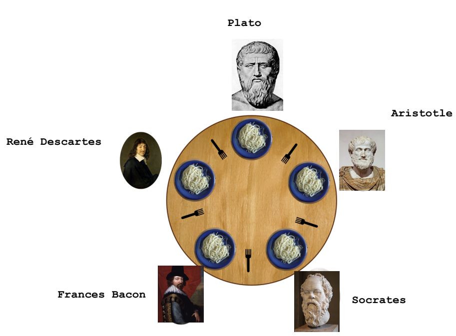
Beware of the Imposters!

Who is who?

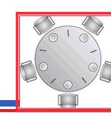
- René Descartes
- Aristotle
- Plato
- Frances Bacon
- Socrates

Answers next slide:

Beware of the Imposters



Dining Philosophers



```
void philosopher( int i )
  while(1)
     think()
     take_chopstick(i);
     eat();
     put_chopstick(i);
```

- Two neighbors can't use chopstick at same time
- Must test if chopstick is there and grab it atomically
  - » Represent EACH chopstick with a semaphore
  - » Grab right chopstick then left chopstick
  - » `sem_t chopstick[5]; // Initialize each to 1`

```
take_chopstick( int i )
   sem_wait( &chopstick[i] );
   sem_wait( &chopstick[(i+1) % 5] );
```

```
put_chopstick( int i )
   sem_post( &chopstick[i] );
   sem_post( &chopstick[(i+1) % 5] );
```

- Guarantees no two neighbors eats simultaneously
- Does this work? Why or Why Not?
- What happens if *all* philosophers wants to eat and grabs the left chopstick (at the same time)?
- Is it efficient? – (assuming we are lucky and it doesn't deadlock)?

Dining Philosophers: *Attempt 2*
Serialize

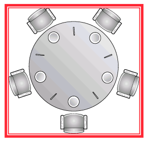
- Add a mutex to ensure that a philosopher gets both chopsticks.

```
take_chopstick( int i )
   sem_wait( &chopstick[i] );
   sem_wait( &chopstick[(i+1) % 5] );
```

```
put_chopstick( int i )
  sem_post( &chopstick[i] );
  sem_post( &chopstick[(i+1) % 5] );
```

```
void philosopher( int i )
  while(1)
      think()
      sem_wait( &mutex );
      take_chopstick(i);
      eat();
      put_chopstick(i);
      sem_post( &mutex )
```

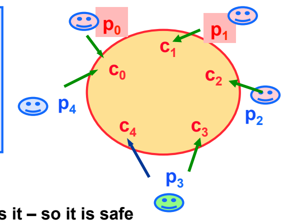
- Problems?
  - » How many philosophers can dine at one time?
  - » How many should be able to eat?



Dining Philosophers: *Common Approach*

- Grab lower-numbered chopstick first, then higher-numbered

```
take_chopstick( int i )
  if( i < 4 )
   sem_wait( &chopstick[i] );      //* Right
   sem_wait( &chopstick[(i+1)] ); //* Left
  else
   sem_wait( &chopstick[0] );      //* Left
   sem_wait( &chopstick[4] );      //* Right
```



- Problems?
  - » Safe: Deadlock? Asymmetry avoids it – so it is safe
- Performance (concurrency?)
  - » P_0 and P_4 grabs chopstick simultaneously - assume P_0 wins
  - » P_3 can now eat but P_0 and P_1 are not eating even if they don't share a chopstick with P_3 (so it is not as concurrent as it could be)

What Todo: Ask *Dijkstra?*

- Want to eat the cake too: Guarantee two goals:
  - » Safety (mutual exclusion): Ensure nothing bad happens (don't violate constraints of problem)
  - » Liveness (progress) : Ensure something good happens when it can (make as much progress as possible)
- Introduce state variable for each philosopher `i`
  - » `state[i] = THINKING, HUNGRY, or EATING`
- Safety: No two adjacent philosophers eat simultaneously (ME)
  - » `for all i: !(state[i]==EATING && state[i+1%5] == EATING)`
- Liveness: No philosopher is HUNGRY *unless* one of his neighbors is eating (actually eating)
  - » ! - Not the case that :
    - – a philosopher is hungry and his neighbors are not eating --
  - » `for all i: !(state[i]==HUNGRY && (state[i+4%5]!=EATING && state[i+1%5]!=EATING))`

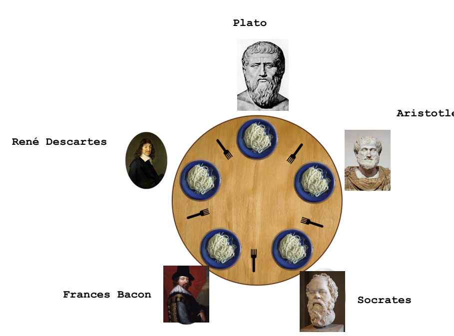
What Todo: Ask *Dijkstra?*

- Guarantees the two goals (helps to solve the problem):
  - » Safety (mutual exclusion): Ensure nothing bad happens (don't violate constraints of problem)
  - » Liveness (progress) : Ensure something good happens when it can (make as much progress as possible)
- Introduce a state variable for each philosopher `i`
  - » `state[i] = THINKING, HUNGRY, or EATING`
- Safety: No two adjacent philosophers eat simultaneously (ME)
  - » `for all i: !(state[i]==EATING && state[i+1%5] == EATING)`
- Liveness: No philosopher is HUNGRY *unless* one of his neighbors is eating
  - » Not the case that a philosopher is hungry and his neighbors are not eating --
  - » `for all i: !(state[i]==HUNGRY && (state[i+4%5]!=EATING && state[i+1%5]!=EATING))`

Dining Philosophers: *Dijkstra*

```
sem_t mayEat[5] = {0};              // permission to eat (testSafety will grant)
sem_t mutex = {1} ;                 // how to init
int state[5] = {THINKING};

take_chopsticks(int i)
  sem_wait( &mutex );               // enter critical section
  state[i] = HUNGRY;
  testSafetyAndLiveness(i);         // check for permission
  sem_post( &mutex );     // exit critical section
  sem_wait(&mayEat[i]);

put_chopsticks(int i)
  sem_wait(&mutex);                 // enter critical section
  state[i] = THINKING;
  testSafetyAndLiveness(i+1 %5);   // check if left neighbor can run now
  testSafetyAndLiveness(i+4 %5);  // check if right neighbor can run now
  sem_post(&mutex);                // exit critical section

testSafetyAndLiveness(int i)
  if( state[i]==HUNGRY && state[i+4%5]!= EATING&&state[i+1%5]!= EATING )
        state[i] = EATING;
        sem_post( &mayEat[i] );
```


Yum!



http://users.erols.com/ziring/diningAppletDemo.html
http://www.doc.ic.ac.uk/~jnm/book/book_applets/Diners.html

- http://www.doc.ic.ac.uk/~jnm/concurrency/classes/Diners/Diners.html

Monitors make things easier!

- Motivation:
  - » Users can inadvertently misuse locks and semaphores (e.g., never unlock a mutex)
- Idea:
  - » Languages construct that control access to shared data
  - » Synchronization added by compiler, enforced at runtime
- Monitor encapsulates
  - » Shared data structures
  - » Methods
    - – that operates on shared data structures
  - » Synchronization between concurrent method invocations
- Protects data from unstructured data access
- Guarantees that threads accessing its data through its procedures interact only in legitimate ways