ME Solutions: Today

- Memory-sharing
  - All previous primitives/mechanisms are memory-sharing
  - Semaphores as *mutex* and as *synchronization primitive*
  - Monitors
- Message-passing solutions
- Barriers
- Classical ME / concurrency problems
  - Access problems (Readers/Writers Problem)
  - Synchronization problems (Dining Philosophers Problem)
  - Scheduling (Sleeping Barber Problem)

Semaphores?

- SW mechanism for synchronization on higher abstraction than TSL assembly
- Semaphores (S): Integer Variables [System Calls]
  - Two standard *atomic* operations:
    - wait()
    - signal()

```
wait(S) {
    while S ≤ 0
          ;  // no-op
    S--;
  }
signal(S) {
    S++;
  }
```

S can be an integer resource counter;
If S is binary, it is called a "**mutex**"

wait(1) → progress & decrement
wait(0) → block
signal(0) → increment and unblock

ME using Mutexs

```
do {
    wait (mutex);            mutex initialized to 1
            // critical section
    signal (mutex);
            // non critical section
  } while (TRUE);
```

Binary Semaphores with TSL: Mutexs

```
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 was unlocked, so return
      CALL thread_yield         | mutex is busy; schedule another thread
      JMP mutex_lock            | try again later
ok:   RET | return to caller; critical region entered

mutex_unlock:
      MOVE MUTEX,#0             | store a 0 in mutex
      RET | return to caller
```

Ordering (Sync.) with Semaphores

- Consider 2 concurrent processes P1/P2 with statements S1/S2
- Would like S2 to be executed <u>only</u> after S1 completed
- Let P1 and P2 share a common semaphore "sync" set to 0

  - [in P1] S1; *Statement S1 executes in Process P1*

    signal(sync);

  - [in P2] wait(sync);

    S2; *Statement S2 executes in Process P2*

[sync = 0 ➔ P2 executes S2 only after P1 has invoked signal(sync); which is only after S1 has been executed]

Semaphores

```
                        #define N 100                      /* number of slots in the buffer */
                        typedef int semaphore;             /* semaphores are a special kind of int */
mutual exclusion →      semaphore mutex = 1;               /* controls access to critical region */
 synchronization →      semaphore empty = N;               /* counts empty buffer slots */
                        semaphore full = 0;                /* counts full buffer slots */

                        void producer(void)
                        {
                             int item;

                             while (TRUE) {                /* TRUE is the constant 1 */
                                  item = produce_item( );  /* generate something to put in buffer */
                                  down(&empty);            /* decrement empty count */
                                  down(&mutex);            /* enter critical region */
                        CS →      insert_item(item);       /* put new item in buffer */
                                  up(&mutex);              /* leave critical region */
                                  up(&full);               /* increment count of full slots */
                             }
                        }

                        void consumer(void)
                        {
                             int item;

                             while (TRUE) {                /* infinite loop */
                                  down(&full);             /* decrement full count */
                                  down(&mutex);            /* enter critical region */
                        CS →      item = remove_item( );   /* take item from buffer */
                                  up(&mutex);              /* leave critical region */
                                  up(&empty);              /* increment count of empty slots */
                                  consume_item(item);      /* do something with the item */
                             }
                        }
```

The producer-consumer problem using semaphores

Semaphore Constructs in Java

- Implemented by **java.util.concurrent.Semaphore** class
- The class uses the wait() and signal() system calls

- public Semaphore available = new Semaphore(100);

- available.**acquire()**; //decreases semaphore value, uses wait() syscall;
- available.**release()**; //increases semaphore value, uses signal() syscall;

…and other available methods, as acquire(int n); release (int n); acquireUninterruptibly() etc.

Semaphore Problems?

- Semaphore → effective SW level synchronization
- System calls (simple!)
- But, synchronization errors are still possible through misuse of wait/signal ☹
  - programmer interchanges order of wait() and signal() ops on the semaphore
    - wait(mutex) CS signal(mutex) → signal(mutex) … CS … wait(mutex)
    - several processes may end up executing their CS concurrently
  - Suppose a user replaces signal(mutex) with wait(mutex)
    - wait(mutex) … CS … wait(mutex)
    - deadlock!!!
  - Suppose the process omits wait(mutex) or signal(mutex) or both
    - ME violated or deadlock

Monitors: Language Constructs not System Calls

```
monitor example
        integer i;                  // shared variable declarations
        condition c;

        procedure producer( );
        .
        .
        .
        end;

        procedure consumer( );
        .
        .
        .
        end;
end monitor;
```

Monitors

```
monitor ProducerConsumer
    condition full, empty;
    integer count;
    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count + 1;
        if count = 1 then signal(empty)
    end;
    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count := count − 1;
        if count = N − 1 then signal(full)
    end;
    count := 0;
end monitor;
```

```
procedure producer;
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item)
    end
end;
procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item)
    end
end;
```

Outline of producer-consumer problem with monitors

- only one monitor procedure active at one time
- buffer has *N* slots

Monitors in Java

```
static class our_monitor {                    // this is a monitor
    private int buffer[ ] = new int[N];
    private int count = 0, lo = 0, hi = 0;   // counters and indices
    public synchronized void insert(int val) {
        if (count == N) go_to_sleep( );     // if the buffer is full, go to sleep
        buffer [hi] = val;                  // insert an item into the buffer
        hi = (hi + 1) % N;                  // slot to place next item in
        count = count + 1;                  // one more item in the buffer now
        if (count == 1) notify( );          // if consumer was sleeping, wake it up
    }
    public synchronized int remove( ) {
        int val;
        if (count == 0) go_to_sleep( );     // if the buffer is empty, go to sleep
        val = buffer [lo];                  // fetch an item from the buffer
        lo = (lo + 1) % N;                  // slot to fetch next item from
        count = count – 1;                  // one few items in the buffer
        if (count == N – 1) notify( );      // if producer was sleeping, wake it up
        return val;
    }
    private void go_to_sleep( ) { try{wait( );} catch(InterruptedException exc) {};}
  }
}
```

Solution to producer-consumer problem in Java

Monitors in Java

```
public class ProducerConsumer {
      static final int N = 100;                      // constant giving the buffer size
      static producer p = new producer( );   // instantiate a new producer thread
      static consumer c = new consumer( );// instantiate a new consumer thread
      static our_monitor mon = new our_monitor( );  // instantiate a new monitor
      public static void main(String args[ ]) {
         p.start( );                                 // start the producer thread
         c.start( );                                 // start the consumer thread
      }
      static class producer extends Thread {
         public void run( )  {                       // run method contains the thread code
            int item;
            while (true) {                           // producer loop
               item = produce_item( );
               mon.insert(item);
            }
         }
         private int produce_item( ) { ... }     // actually produce
      }
      static class consumer extends Thread {
         public void run( )  {                       run method contains the thread code
            int item;
            while (true) {                           // consumer loop
               item = mon.remove( );
               consume_item (item);
            }
         }
         private void consume_item(int item) { ... }     // actually consume
      }
```

Solution to producer-consumer problem in Java

Problems?

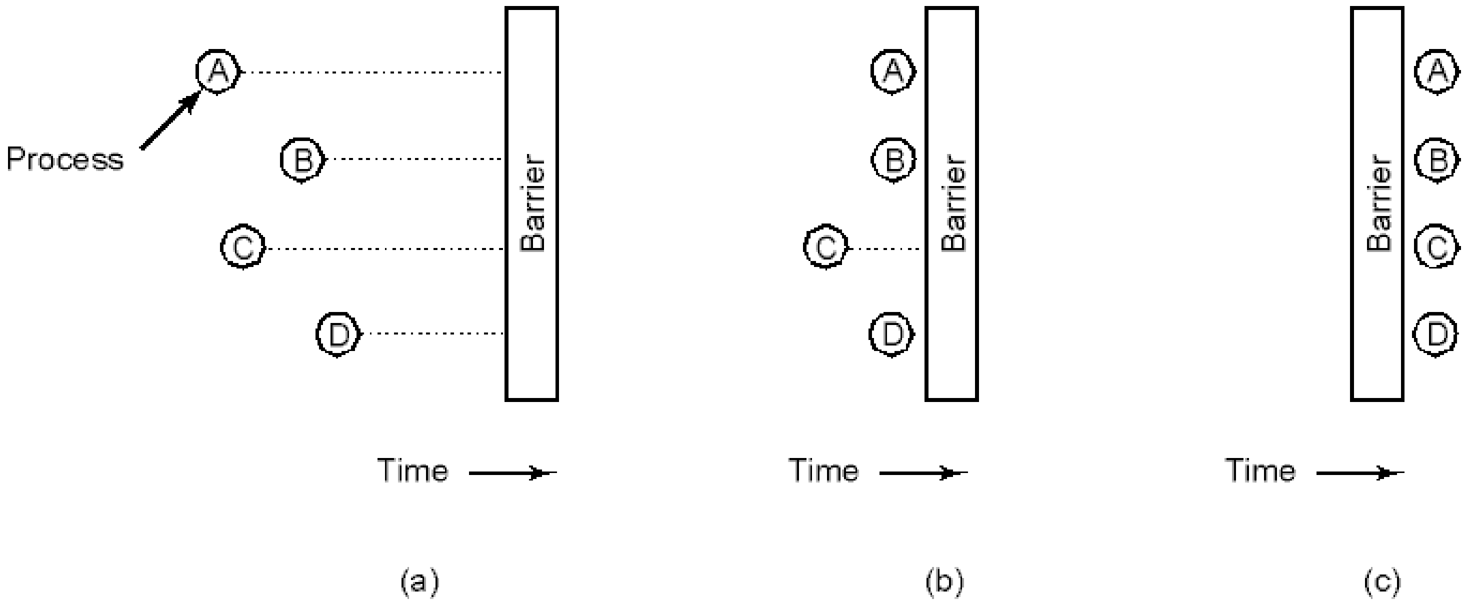
- TSL: lowest level (HW),
- Semaphores: low-level (kernel), depend too much on programmer's skills
- Monitors: need language support (C/Pascal?)
- All: **memory sharing** solutions, work only on the same machine but not if the processes sit in different machines (LAN etc.)
- Let's look at **message passing** solutions (send/receive)

Producer-Consumer with Message Passing

```
#define N 100                              /* number of slots in the buffer */

void producer(void)
{
      int item;
      message m;                           /* message buffer */

      while (TRUE) {
            item = produce_item( );        /* generate something to put in buffer */
            receive(consumer, &m);         /* wait for an empty to arrive */
            build_message(&m, item);       /* construct a message to send */
            send(consumer, &m);            /* send item to consumer */
      }
}

void consumer(void)
{
      int item, i;
      message m;

      for (i = 0; i < N; i++) send(producer, &m); /* send N empties */
      while (TRUE) {
            receive(producer, &m);         /* get message containing item */
            item = extract_item(&m);       /* extract item from message */
            send(producer, &m);            /* send back empty reply */
            consume_item(item);            /* do something with the item */
      }
}
```

Ques: what happens if the producer (or the consumer) is much faster at processing messages than the consumer (or producer)?

Barriers (primitives) for Synchronization



Use of a barrier (~ AND operation)

a) processes approaching a barrier
b) all processes blocked at barrier, waiting for C
c) last process (C) arrives, all are let through

Synchronization Implementations

- Solaris
  – adaptive mutex, semaphores, RW locks, threads blocked by locks etc

- Windows XP
  – interrupt masking, busy-waiting spin locks (for short code segments), mutex, semaphores, monitors, msg. passing

- Linux
  – pre v2.6 (non-preemptible); post v2.6 preemptible: interrupts
  – semaphores, spin-locks (for short CS's in kernel only)

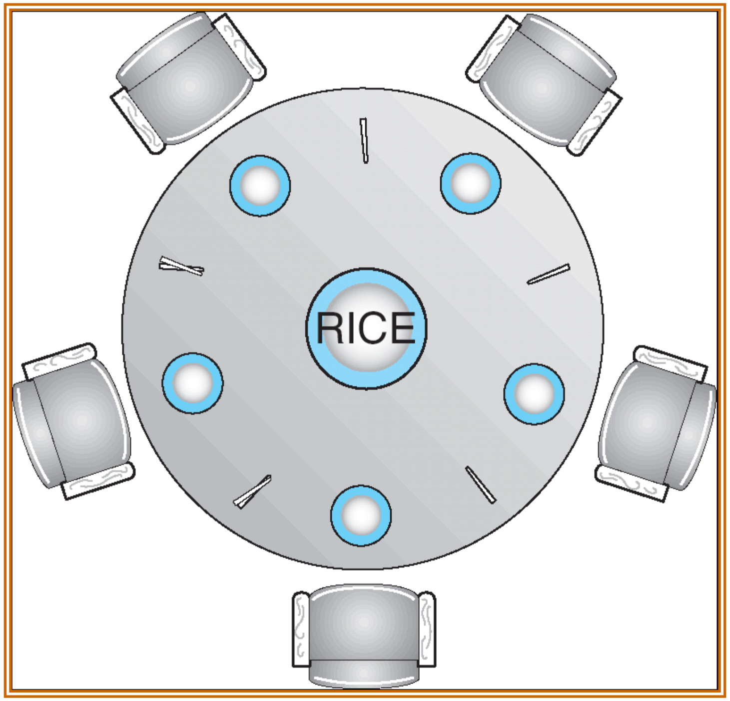
Classical ME/Concurrency Problems

- Synch. problems (Dining Philosophers Problem)
- Access problems (Readers/Writers Problem)
- Scheduling (Sleeping Barber Problem)

Dining Philosophers

- Philosophers eat/think
- Eating needs 2 chopsticks (forks)
- Pick one instrument at a time

- Solutions (with semaphores?)…



Dining Philosophers – Obvious Solution

```
#define N 5                              /* number of philosophers */

void philosopher(int i)                  /* i: philosopher number, from 0 to 4 */
{
     while (TRUE) {
            think( );                    /* philosopher is thinking */
wait        take_fork(i);                /* take left fork */
wait        take_fork((i+1) % N);        /* take right fork; % is modulo operator */
            eat( );                      /* yum-yum, spaghetti */
signal      put_fork(i);                 /* put left fork back on the table */
signal      put_fork((i+1) % N);         /* put right fork back on the table */
     }
}
```

Deadlocks? – all pick the left fork at the same time
→ add a check if the fork is available → Livelock!

Dining-Philosophers Problem

- Philosopher *i*:

```
while (true)  {

  …think…

 wait ( chopstick[i] );
 wait ( chopstick[ (i + 1) % N] )
        signal (…) ?
  …eat…

  signal (chopstick[i] );
  signal (chopstick[ (i + 1) % N] );
 }
```

Needs:
- No deadlock
- No starvation for anyone
- Maximum parallelism

Dining Philosophers – No deadlocks - Max Parallelism Solution

```
#define N           5             /* number of philosophers */
#define LEFT        (i+N−1)%N     /* number of i's left neighbor */
#define RIGHT       (i+1)%N       /* number of i's right neighbor */
#define THINKING    0             /* philosopher is thinking */
#define HUNGRY      1             /* philosopher is trying to get forks */
#define EATING      2             /* philosopher is eating */
typedef int semaphore;            /* semaphores are a special kind of int */
int state[N];                     /* array to keep track of everyone's state */
semaphore mutex = 1;              /* mutual exclusion for critical regions */
semaphore s[N];                   /* one semaphore per philosopher */

void philosopher(int i)           /* i: philosopher number, from 0 to N−1 */
{
    while (TRUE) {                /* repeat forever */
        think( );                 /* philosopher is thinking */
        take_forks(i);            /* acquire two forks or block */
        eat( );                   /* yum-yum, spaghetti */
        put_forks(i);             /* put both forks back on table */
    }
}
```

Continuation…

```
void take_forks(int i)                /* i: philosopher number, from 0 to N–1 */
{
      down(&mutex);                   /* enter critical region */
      state[i] = HUNGRY;              /* record fact that philosopher i is hungry */
      test(i);                        /* try to acquire 2 forks */
      up(&mutex);                     /* exit critical region */
      down(&s[i]);                    /* block if forks were not acquired */
}

void put_forks(i)                     /* i: philosopher number, from 0 to N–1 */
{
      down(&mutex);                   /* enter critical region */
      state[i] = THINKING;            /* philosopher has finished eating */
      test(LEFT);                     /* see if left neighbor can now eat */
      test(RIGHT);                    /* see if right neighbor can now eat */
      up(&mutex);                     /* exit critical region */
}

void test(i)                          /* i: philosopher number, from 0 to N–1 */
{
      if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
            state[i] = EATING;
            up(&s[i]);      // acquire forks
      }
}
```

Deadlock free + max. parallelism

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the database; they do not perform any updates
  - Writers – can both read and write.
- Problem – allow multiple readers to queue to read at the same time. Only one single writer can access the shared data at a time.
- Shared Data
  - Database
  - Integer readcount initialized to 0 (# of processes currently reading object)
  - Semaphore mutex initialized to 1; controls the access to readcount
  - Semaphore db initialized to 1; controls access to database;

The Readers and Writers Problem

```
typedef int semaphore;              /* use your imagination */
semaphore mutex = 1;                /* controls access to 'rc' */
semaphore db = 1;                   /* controls access to the database */
int rc = 0;                         /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {                  /* repeat forever */
        down(&mutex);               /* get exclusive access to 'rc' */
        rc = rc + 1;                /* one reader more now */
        if (rc == 1) down(&db);     /* if this is the first reader ... */
        up(&mutex);                 /* release exclusive access to 'rc' */
        read_data_base( );          /* access the data */
        down(&mutex);               /* get exclusive access to 'rc' */
        rc = rc − 1;                /* one reader fewer now */
        if (rc == 0) up(&db);       /* if this is the last reader ... */
        up(&mutex);                 /* release exclusive access to 'rc' */
        use_data_read( );           /* noncritical region */
    }
}


void writer(void)
{
    while (TRUE) {                  /* repeat forever */
        think_up_data( );           /* noncritical region */
        down(&db);                  /* get exclusive access */
        write_data_base( );         /* update the data */
        up(&db);                    /* release exclusive access */
    }
}
```

The Sleeping Barber Problem

- 1 Barber
- 1 Barber Chair
- N Customer Chairs

The Sleeping Barber Problem

```
#define CHAIRS 5                    /* # chairs for waiting customers */

typedef int semaphore;              /* use your imagination */

semaphore customers = 0;            /* # of customers waiting for service */
semaphore barbers = 0;              /* # of barbers waiting for customers */
semaphore mutex = 1;                /* for mutual exclusion */
int waiting = 0;                    /* customers are waiting (not being cut) */

void barber(void)
{
     while (TRUE) {
          down(&customers);         /* go to sleep if # of customers is 0 */
          down(&mutex);             /* acquire access to 'waiting' */
          waiting = waiting – 1;    /* decrement count of waiting customers */
          up(&barbers);             /* one barber is now ready to cut hair */
          up(&mutex);               /* release 'waiting' */
          cut_hair( );              /* cut hair (outside critical region) */
     }
}


void customer(void)
{
     down(&mutex);                  /* enter critical region */
     if (waiting < CHAIRS) {        /* if there are no free chairs, leave */
          waiting = waiting + 1;    /* increment count of waiting customers */
          up(&customers);           /* wake up barber if necessary */
          up(&mutex);               /* release access to 'waiting' */
          down(&barbers);           /* go to sleep if # of free barbers is 0 */
          get_haircut( );           /* be seated and be serviced */
     } else {
          up(&mutex);               /* shop is full; do not wait */
     }
}
```