CSc 360
Operating Systems
Semaphores and Monitors

Wenjun Yang

Fall 2025

UVSS 2025

AGM

Annual General Meeting

OCT. 28TH AT 4PM | ONLINE VIA ZOOM

Each Student that attends is entered to win:

A BAGGU MEDIUM CRESCENT BAG
$100 TO THRIFTY FOODS
AIRPODS!

Introducing
DECOLONIZATION POLICY

Board Changes:
SHOULD THERE BE A UVSS PRESIDENT?

AND MORE!

UVSS.CA/AGM

Hardware-based: “test-and-set”

- Test and set value **atomically**

```
boolean TestAndSet(boolean &target) {
            boolean rv = target;
            target = true;
            return rv;
      }
boolean lock = false; /* shared variable */
do {
      while (TestAndSet(lock)) ;  // wait
            /* critical section */
      lock = false;
            /* remainder section */
}
```

- Any problem?

Hardware-based: “swap”

- Exchange value **atomically**

```
void Swap (boolean *a, boolean *b)
 {
        boolean temp = *a;
        *a = *b;
        *b = temp:
 }
 while (true)  {
         key = TRUE;
         while (key == TRUE) Swap (&lock, &key );
                 //      critical section
          lock = FALSE;
                 //      remainder section
  }
```

Software-based: mutex

- Mutual exclusion (mutex)
  - only two states
    - unlocked: there is no thread in critical section
    - locked: there is one thread in critical section
  - state change is atomic (by lower/hardware)
    - if it is unlocked, it can be locked by at most one thread when entering the critical section
    - if it is locked, it can “only” be unlocked by the locking thread when leaving the critical section

Mutex: more

- Mutex procedures
  - create a mutex variable (initially unlocked)
  - (some threads) attempt to lock the mutex
    - only one can lock the mutex
      - others may be blocked and waiting
    - the one with the mutex
      - execute the critical section
      - unlock the mutex variable eventually
  - destroy the mutex variable

see pthread_mutex_*()

Software-based: semaphores

- Semaphore API
  - Semaphore *S* – ***integer*** variable
    - binary semaphore (~ mutex)
    - counting semaphore
  - two indivisible (atomic) operations
    - also known as P() and V() due to Dijkstra

wait (*S*):

  while *S*<= 0 do *no-op*;
  ***S*--;**

signal (*S*):

  S++;

Dutch: **Probe**er vs Verhoog (**inc**) Q: busy-wait problem?

Using semaphores

- Mutual exclusion
  - binary semaphore
  - shared data

    semaphore mutex;   // initially *mutex* = 1
  - process *Pi*

```
do {
   wait(mutex);
               /* critical section */
   signal(mutex);
               /* remainder section */
} while (1);
```

- Resource access
  - counting semaphore
  - initially, the number of resource instances

see sem_*() in pthread

Semaphore implementation

- Semaphores ***without*** busy waiting
  - block(): block the caller process
  - wakeup(): wake up another process

wait(*S*):

```
S.value--;
if (S.value < 0) {
        add this process to S.L;
        block();
}
```

signal(*S*):

```
S.value++;
if (S.value <= 0) {
        remove a process P from S.L;
        wakeup(P);
}
```

More on using semaphores

- Ordered execution
  - initially, flag = 0;
  - P1: …; *do_me_first*; signal (flag);
  - P2: …; wait (flag); *then_follow_on*;
- Caution
  - deadlock
    - wait (A); wait (B); …; signal (A); signal (B);
    - wait (B); wait (A); …; signal (B); signal (A);
  - starvation

The producer-consumer problem

- With semaphore

```
while (true)  {
  //  produce an item
    wait (empty);
    wait (mutex);
    //  add the item to the  buffer
   signal (mutex);
   signal (full);
}
```

```
while (true) {
    wait (full);
    wait (mutex);
    //  remove an item
    signal (mutex);
    signal (empty);
    //  consume the item
}
```

The readers-writers problem

- First reader**s**-writer**s** problem
  - no readers kept waiting unless writer is writing

```
while (true) {
    wait (wrt) ;
    //    writing is performed
    signal (wrt) ;
}
```

```
while (true) {
    wait (mutex) ;
    readcount ++ ;
    if (readcount == 1)  wait (wrt) ;
    signal (mutex);
    // reading is performed
    wait (mutex) ;
    readcount  -- ;
    if (readcount  == 0)  signal (wrt) ;
    signal (mutex) ;
}
```

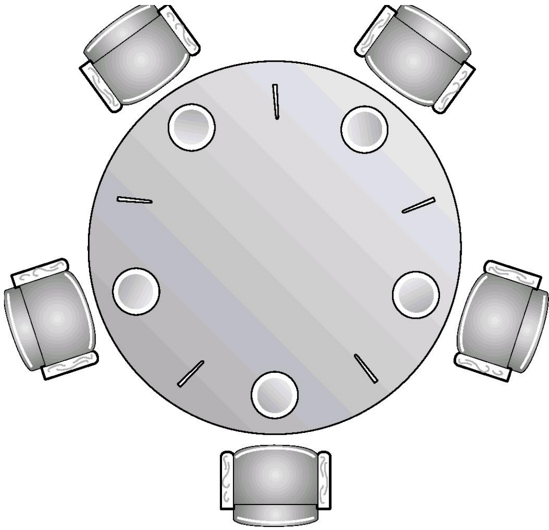

* what does *mutex* protect? what does *wrt* protect?

Dining philosophers

- Shared data
  - Initially all values are 1

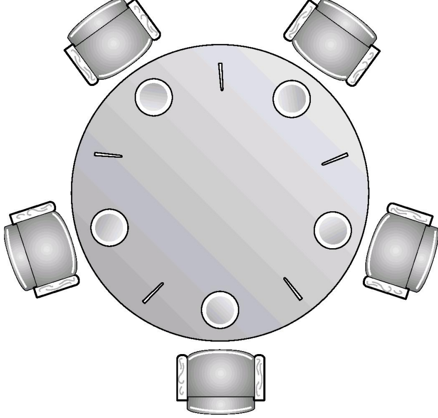
    semaphore chopstick[5];
- Philosopher *i*:

```
do {
        wait(chopstick[i])
        wait(chopstick[(i+1) % 5])
                …
                eat
                …
        signal(chopstick[i]);
        signal(chopstick[(i+1) % 5]);
                …
                think
                …
        } while (1);
```



Q: any possible problems?

This lecture so far

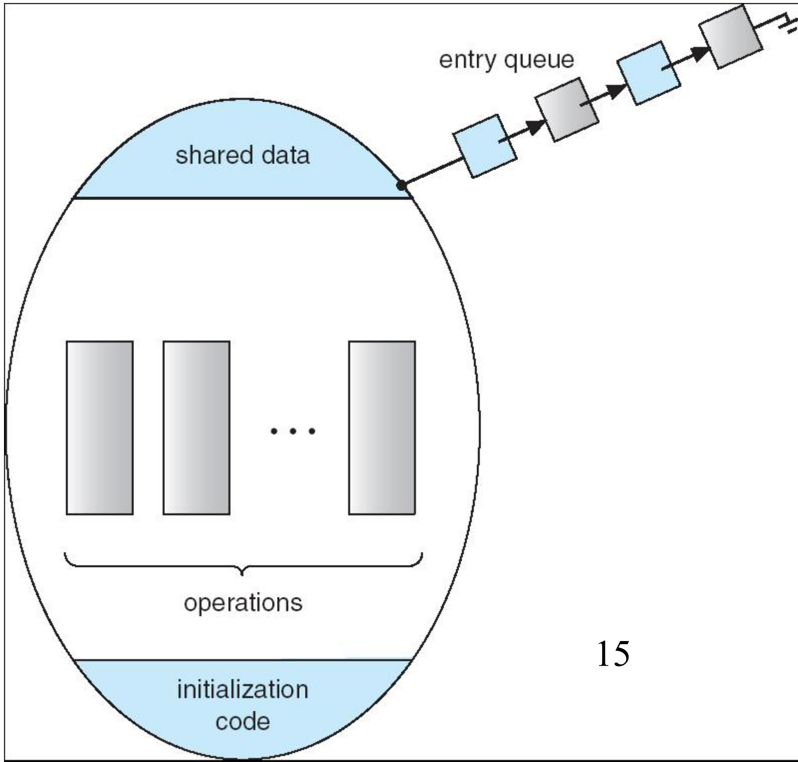


- Hardware-assisted synchronization
  – test-and-set and *swap*
- Mutex
- Semaphores
  – with(out) busy waiting
- Properties
  – mutual exclusion, making process, bounded waiting

Monitor

- A high-level abstraction (**OO** design) that provides a convenient and effective mechanism for process synchronization
- Only ***one*** process may be active within the monitor at a time

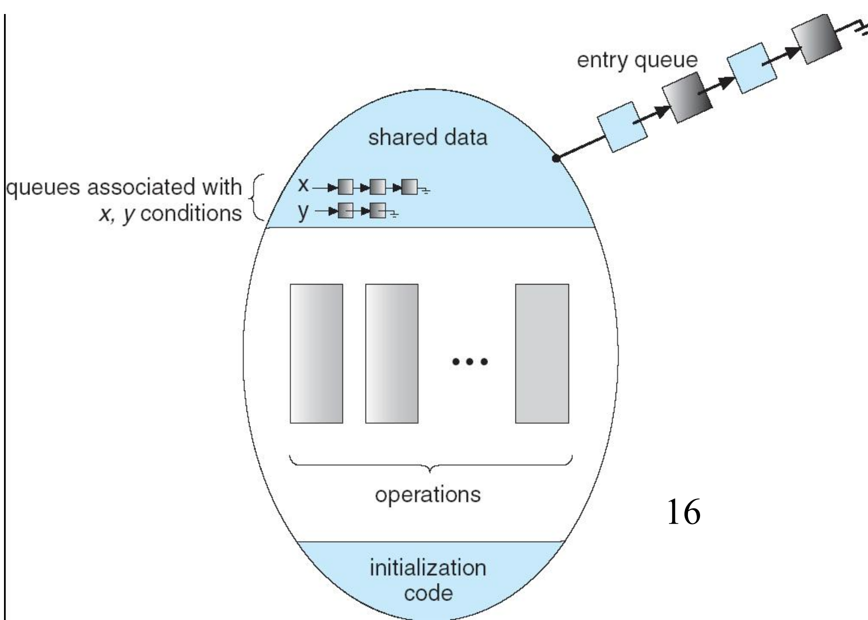
```
monitor monitor-name
{
   // shared variable declarations
   procedure P1 (…) { …. }
           …
   procedure Pn (…) {……}
    Initialization code ( ….) { … }
           …
}
```



- Any problem?

Condition variables

- No busy-waiting!
  - condition variables
- Two operations on a condition variable
  - x.wait () – a process that invokes the operation is suspended.
  - x.signal () – resumes one of processes (if any) that invoked x.wait ()



Dining philosophers: monitors

```
monitor DP {
    enum { THINKING, HUNGRY, EATING} state [5] ;
    condition self [5];

    void pickup (int i) {
          state[i] = HUNGRY;
          test(i);
          if (state[i] != EATING) self [i].wait;
    }

      void putdown (int i) {
          state[i] = THINKING;
                // test left and right neighbors
          test((i + 4) % 5);
          test((i + 1) % 5);
      }
```

DP monitor: more

```
void test (int i) {
        if ( (state[(i + 4) % 5] != EATING) &&
        (state[i] == HUNGRY) &&
        (state[(i + 1) % 5] != EATING) ) {
             state[i] = EATING ;
                   self[i].signal () ; // no effect if not blocked
         }
 }

  initialization_code() {
       for (int i = 0; i < 5; i++)
        state[i] = THINKING;
  }
}
```

- Any problem?

- Using monitors

dp.pickup (i)

...

EAT

...

dp.putdown (i)

Implementing monitors

- Variables
  semaphore mutex; // for the monitor, initially = 1
  semaphore next; // for suspended processes, initially = 0
  int next-count = 0; // # of suspended processes
- Each procedure ***F*** will be replaced by

```
wait(mutex);                // wait for the access to the monitor
        …
        body of F;
        …
if (next-count > 0)         // whether there are suspended processes
        signal(next);
else                        // free the monitor to other processes
        signal(mutex);
```

- Mutual exclusion within a monitor is ensured

Implementing Condition Variables

- For each condition variable ***x***, we have:

```
semaphore x-sem; // initially  = 0
int x-count = 0;
```

- The operation x.wait can be implemented as:

```
x-count++;
if (next-count > 0)
        signal(next);    // wake up the first one of the suspended q
else
        signal(mutex); // free the monitor
wait(x-sem);          // join the x queue
x-count--;
```

- The operation x.signal can be implemented as:

```
if (x-count > 0) {     // no effect if x is not blocked
        next-count++;
        signal(x-sem); // wake up the first one of the x queue
        wait(next);             // join the suspended queue
        next-count--;
}
```


* waiting → ready (next) → running

Mutex with pthread

- Create mutex
  - int pthread_mutex_init (mutex, attributes);
- Attempt to lock
  - int pthread_mutex_lock (mutex);
    - if unlocked, lock and return immediately
    - if locked
      - "fast" lock: blocked until the mutex is unlocked
      - "test" lock: return immediately with error
      - "recursive" lock: "over"-lock
        - » multiple pthread_mutex_unlock() to unlock

Mutex with pthread: more

- Try to lock
  - int pthread_mutex_trylock (mutex);
    - if locked, return immediately with error code
- Unlock
  - int pthread_mutex_unlock (mutex);
    - if “recursive” lock, multiple pthread_mutex_unlock necessary to fully unlock the mutex
- Destroy mutex
  - int pthread_mutex_destroy (mutex);

Condition variable

- Used together with mutex
  - mutex: control access to shared data
  - condition: synchronize by condition “predict”
- Wait for condition
  - pthread_cond_wait (condition, mutex);
    - automatically unlock and wait for signal
    - on signal, wake up and automatically lock
- Signal or broadcast
  - pthread_cond_signal (condition);

Example: mutex and condition

- Main thread
  - global variable
  - create mutex and condition variable

- Wait to be signaled
  - pthread_mutex_lock();
  - pthread_cond_wait();
  - pthread_mutex_unlock();

- Send the signal
  - pthread_mutex_lock();
  - pthread_cond_signal();
  - pthread_mutex_unlock();

The 2nd half of this lecture

- Synchronization with monitors
  - monitor: a high-level ADT
  - using monitors
    - with condition variables
  - implementing monitors
    - with semaphores
- Practice semaphores/monitors with
  - classical synchronization problems
  - pthreads mutex and convar

Next lecture

- Deadlocks
  - read OSC7 Chapter 7 (or OSC6 Chapter 8)