CSc 360
Operating Systems
Semaphores

Jianping Pan

Summer 2015

Dekker's solution

- requirements: mutex, no deadlock, no livelock
- Process Pi

```
while (true) {
    flag[i] = true;
    while (flag[j]) {
       if (turn == j) {
           flag[i] = false;
           while (turn == j);   // wait
           flag[i] = true;
       }
    }
    /* critical section */
    turn = j;
    flag[i] = false;
    /* remainder section */
}
```

- Be polite: meet all three requirements; solve the critical-section problem for ***two*** processes

Review: synchronization

- Peterson's solution
  - software-based solution

```
do {
    flag [i]:= true;
    turn = j;
    while (flag [j] and turn == j) ;
    /* critical section */
    flag [i] = false;
    /* remainder section */
} while (1);
```

  - assumption? limitation?

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)) ;
        /* 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
}
```


Q: swap without temp?

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
    - 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
    - counting semaphore
  - two indivisible (atomic) operations
    - also known as P() and V()

wait (*S*):

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

signal (*S*):

```
S++;
```


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

Semaphore implementation

- Semaphores without busy waiting
  - block(): block the caller process
  - wakeup(): wakeup 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 readers-writers problem
  - no readers kept waiting unless writer is writing

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

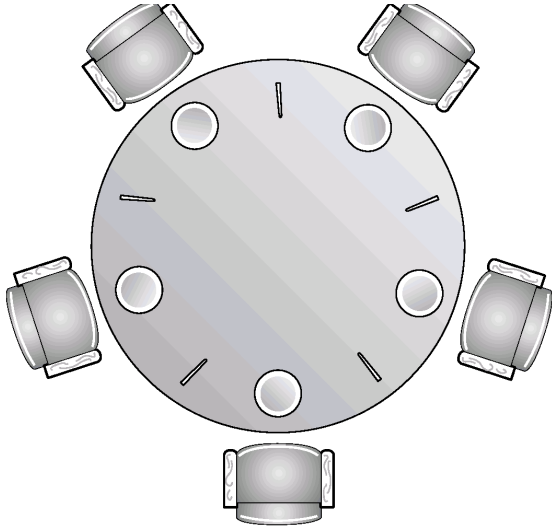
```
while (true) {
    wait (mutex) ;
    readcount ++ ;
    if (readcount == 1)  wait (wrt) ;
    signal (mutex);
    // reading is performed
    wait (mutex) ;
    readcount  - - ;
    if (readcount  == 0)  signal (wrt) ;
    signal (mutex) ;
}
```

Example: 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

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

Next lecture

- More on synchronization
  - read OSC7Ch6