Synchronization for Concurrent Tasks

Minsoo Ryu

Department of Computer Science and Engineering
Hanyang University

Outline

Race Condition

- The situation where several processes access – and manipulate shared data concurrently
  - The final value of the shared data depends upon which process finishes last

Thread A

```
item nextProduced;

if (user_wants_to_write == 1) {
        while (counter == BUFFER_SIZE)
                ; /* do nothing */
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

Thread B

```
item nextConsumed;

If (user_wants_to_read == 1) {
        while (counter == 0)
                ; /* do nothing */
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
}
```

General Solution Structure

- General structure of process P_i (other process P_j)

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (1);
```

- Processes may share some common variables to synchronize their actions

Example

Thread A

```
item nextProduced;

if (user_wants_to_write == 1) {
        while (counter == BUFFER_SIZE)
                ; /* do nothing */
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        pthread_mutex_lock(&mutex);
        counter++;
        pthread_mutex_lock(&mutex);
}
```

Thread B

```
item nextConsumed;

If (user_wants_to_read == 1) {
        while (counter == 0)
                ; /* do nothing */
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        pthread_mutex_lock(&mutex);
        counter--;
        pthread_mutex_lock(&mutex);
}
```

Three Approaches to Critical Sections

- **Algorithmic approaches**
  - Algorithmically solves the critical section problem without using any special HW and OS support
- **Hardware support**
  - Use special hardware support to achieve atomicity
  - Interrupt disabling, Test and Set instruction, Swap instruction
- **OS primitives**
  - Use OS primitives
  - Semaphore, mutex, spin lock, reader-writer lock, …

Algorithmic Approach: Bakery Algorithm by Lamport

- Before entering its critical section, process receives a number
  - Holder of the smallest number enters the critical section

- If processes P_i and P_j receive the same number, if $i < j$, then P_i is served first; else P_j is served first

- The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1,2,3,3,3,3,4,5...

Bakery Algorithm by Lamport

- **Process P_i**
  - **Initially, number[i] = 0**

```
do {
    choosing[i] = true;
    number[i] = max (number[0], number[1] , …, number [n – 1]) + 1;
    choosing[i] = false;
    for (j = 0; j < n; j++) {
        while (choosing[ j ]) ;
        while ((number[ j ] != 0) && ((number[ j ] < (number[i]) || (j < i)) ;
    }
        critical section
    number[i] = 0;
        remainder section
} while (1);
```

ties are broken by comparing j and i

Hardware Approach: Synchronization by Interrupt Disabling

- Guarantee atomicity for accessing a critical section
  - Forbid interrupts to occur in a critical section
  - The current sequence of instructions would be allowed to execute in order without preemption (atomicity)
  - No unexpected modifications could be made

- Unfortunately, this solution is not feasible in a multiprocessor environment
  - Disabling interrupts on a multiprocessor requires significant time to propagate the message to all the processors

Synchronization by Test-and-Set

- **Many machines provide special HW instructions**
  - **To allow us to test and modify the content of a word, or**
  - **To swap the contents of two words, atomically**

- **Test and set instruction**

```
boolean TestAndSet(boolean &target) {
    boolean rv = target;
    target = true;

    return rv;
}
```

Mutual Exclusion with Test-and-Set

- **Shared data:**

```
boolean lock = false;
```

- **Process P_i**

```
do {
    while (TestAndSet(lock)) ;
        critical section
    lock = false;
        remainder section
} while(1)
```

OS Primitives for Critical Sections

- OS provide several primitives for critical sections
  - Mutual exclusion, progress, and bounded waiting are guaranteed by OS
  - Most popular primitives are semaphores and mutexes

```
int main()
{
  sem_t semaphore;
  int count = 0;

  sem_init( &semaphore, 0, 1 );
  sem_wait( &semaphore );
  count++;
  sem_post( &semaphore );
  sem_destroy( &semaphore );
}
```

```
void * count( void * param )
{
  for ( int i=0; i<100; i++ )
  {
    pthread_mutex_lock( &mutex );
    counter++;
    printf( "Count = %i\n", counter );
    pthread_mutex_unlock( &mutex );
  }
}
```

Semaphores vs. Mutexes

Semaphore

- **A semaphore is a counter that can be either incremented or decremented**
- **Two types of semaphores**
  - Counting semaphores (integer numbers)
  - Binary semaphores (0 or 1)
- **Semaphores can be shared by different processes**

Mutex

- **A binary semaphore**
- **Mutex has a sense of ownership**
  - Only the task that has lock the mutex can unlock it
- **By default, a mutex can be shared between threads**

Deadlock

- Semaphores or mutexes must be used carefully
  - Otherwise, a deadlock or a livelock can happen

Thread 1

```
void update1()
{
   acquire(A);
   acquire(B); <<< Thread 1
                   waits here
   variable1++;
   release(B);
   release(A);
}
```

Thread 2

```
void update2()
{
   acquire(B);
   acquire(A); <<< Thread 2
                   waits here
   variable1++;
   release(B);
   release(A);
}
```

Livelock

- Threads can be trapped in a livelock of constantly acquiring and releasing mutexes

```
  int done=0;
  while (!done)
  {
    acquire(A);
    if ( canAcquire(B) )
    {
      variable1++;
      release(B);
      release(A);
      done=1;
    }
    else
    {
      release(A);
    }
  }
}
```

```
  int done=0;
  while (!done)
  {
    acquire(B);
    if ( canAcquire(A) )
    {
      variable2++;
      release(A);
      release(B);
      done=1;
    }
    else
    {
      release(B);
    }
  }
}
```

Spin Locks

- **Spin locks are essentially mutex locks**
  - Tasks waiting for mutex locks can sleep or spin
  - Tasks using a spin lock keeps trying to acquire the lock without sleeping

- **Advantage of spin locks**
  - Tasks will acquire the lock as soon as it is released
  - For a mutex lock that sleeps, the task needs to be woken by the operating system before it can get the lock

- **Disadvantage of spin locks**
  - A spin lock will be monopolizing the CPU

POSIX Spinlocks

```
pthread_spinlock_t lock;

void lockandunlock()
{
  int i = 10000;
  while ( i>0 )
  {
    pthread_spin_lock( &lock );
    i--;
    pthread_spin_unlock( &lock );
  }
}

int main()
{
  pthread_spin_init( &lock, PTHREAD_PROCESS_PRIVATE );
  lockandunlock();
  pthread_spin_destroy( &lock );
}
```

Synchronization in Real World

- OS primitives often serve as a powerful tool for many applications

- However, there are many real world problems that are hard to solve only with OS primitives
  - We often need a more complex approach
  - One difficult problem is the "Readers-Writers Synchronization"

The First Readers-Writers Problem

- **The first readers-writers problem**
  - **Give preferential treatment to readers**
  - **Writers may suffer unbounded waiting**

- **Shared data**
  - semaphore S, wrt;
  - **Initially,** S = 1, wrt = 1, readcount = 0

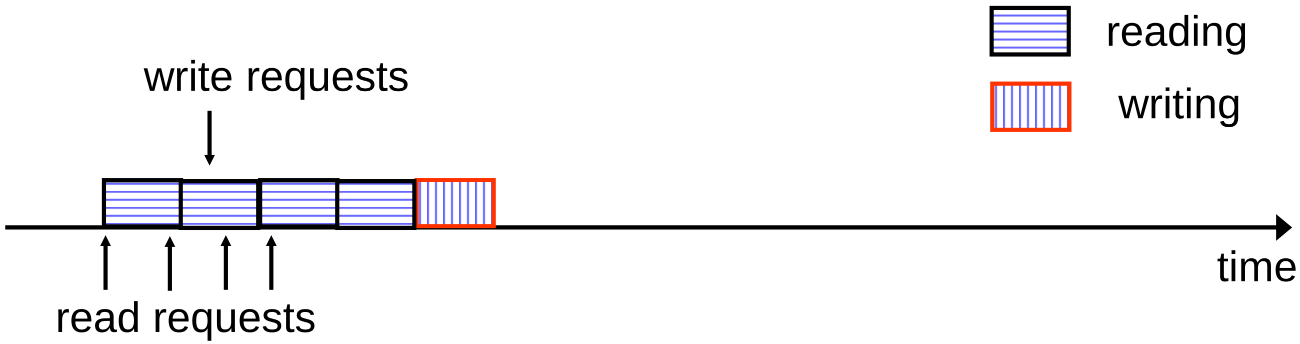
The First Readers-Writers Problem

```
/* Writer */
wait(wrt);
    …
    writing is performed
    …
signal(wrt);
```

```
/* Reader */
wait(S);
readcount++;
if (readcount == 1)
    wait(wrt);
signal(S);
    …
    reading is performed
    …
wait(S);
readcount--;
if (readcount == 0)
signal(wrt);
signal(S):
```

The First Readers-Writers Problem

- The first readers-writers problem



The Second Readers-Writers Problem

- The second readers-writers problem
  - Give preferential treatment to writers
  - Readers may suffer unbounded waiting

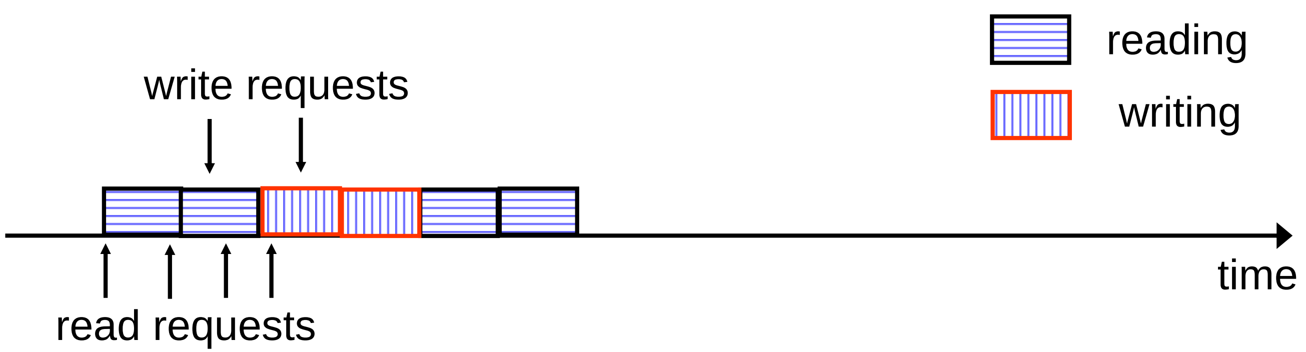
- Shared data
  - semaphore S1, S2, wrt, rd, wrtpending;
  - Initially,
    - S1 = 1, S2 = 1, wrt = 1, rd = 1, wrtpending = 1,
    - readcount = 0, writecount = 0

The Second Readers-Writers Problem

```
/* Writer */
wait(S2);
writecount++;
if (writecount == 1)
    wait(rd);
signal(S2);
wait(wrt)
    …
    writing is performed
    …
signal(wrt);
wait(S2);
writecount--;
if (writecount == 0)
    signal(rd);
signal(S2);
```

```
/* Reader */
wait(wrtpending);
wait(rd);
wait(S1);
readcount++;
if (readcount == 1)
    wait(wrt);
signal(S1);
signal(rd);
signal(wrtpending);
    …
    reading is performed
    …
wait(S1);
readcount--;
if (readcount == 0)
signal(wrt);
signal(S1):
```

The Second Readers-Writers Problem




Q&A