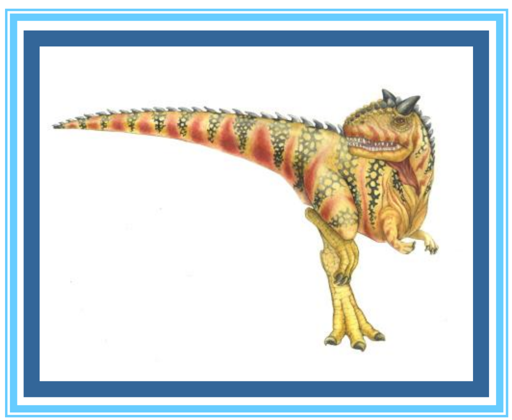
Chapter 7: Synchronization Examples

Chapter 7: 7.1, 7.3



Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Bounded-Buffer Problem

- A producer process and a consumer process share a pool of n buffers, each can hold one item
  - A buffer becomes **full** when the producer process writes a new item into it
  - A buffer becomes **empty** when the consumer process consumes an item contained in it
- A solution must satisfy the following conditions:
  1. Producer cannot produce an item if all buffers are full
  2. Consumer cannot consume an item if all buffers are empty
  3. Producer and consumer must access the buffer pool in a mutually exclusive manner

A Solution to Bounded-Buffer Problem

- ***n*** buffers, each can hold one item
- Producer and consumer share the following semaphores:
  - Semaphore **`mutex`** initialized to 1
    - Provides mutual exclusion for accesses to the buffer pool
  - Semaphore **`full`** counts the number of full buffers, initialized to 0
    - Consumer can consumer an item if **`full`** > 0
  - Semaphore **`empty`** counts the number of empty buffers, initialized to n
    - Producer can produce an item if **`empty`** > 0

A Solution to Bounded-Buffer Problem (Cont.)

- The structure of the producer process

```
while (true) {
        ...
        /* produce an item in next_produced */
        ...
     wait(empty);
     wait(mutex);
        ...
        /* add next produced to the buffer */
        ...
     signal(mutex);
     signal(full);
 }
```

A Solution to Bounded-Buffer Problem (Cont.)

- The structure of the consumer process

```
while (true) {
    wait(full);
    wait(mutex);
        ...
        /* remove an item from buffer to next_consumed */
        ...
    signal(mutex);
    signal(empty);
        ...
        /* consume the item in next consumed */
        ...
    }
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - **Readers** – only read the data set
  - **Writers** – can both read and write
- A solution must satisfy the following:
  1. Only one writer can perform writing at any time
  2. Reading is not allowed while a writer is writing
  3. Many readers can perform reading concurrently

Readers-Writers Problem

Shared variables:

- Semaphore **rw_mutex** initialized to 1
  - Used by both readers and writers to ensure that the writers have exclusive access to the shared data set
- Integer **read_count** counts the number of readers that are currently reading, initialized to 0
- Semaphore **mutex** initialized to 1
  - Used by readers to ensure mutual exclusion when **read_count** is updated

Readers-Writers Problem (Cont.)

- The structure of a writer process

```
while (true) {
        wait(rw_mutex);
            ...
        /* writing is performed */
            ...
        signal(rw_mutex);
}
```

Readers-Writers Problem (Cont.)

- The structure of a reader process

```
while (true){
        wait(mutex);
        read_count++;
        if (read_count == 1)
                wait(rw_mutex);
        signal(mutex);
        ...
        /* reading is performed */
        ...
        wait(mutex);
        read count--;
        if (read_count == 0)
                signal(rw_mutex);
        signal(mutex);
}
```

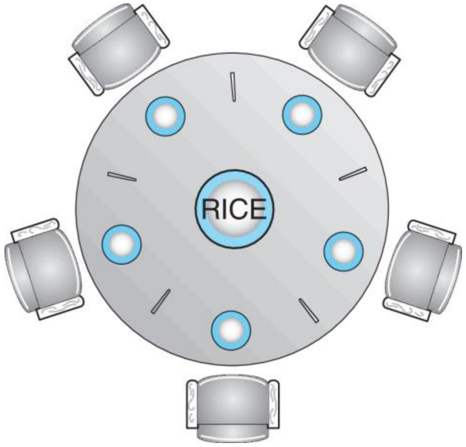
Note:

- No reader is kept waiting unless a writer is writing
- If a writer is writing and n readers arrive, first reader waits on rw_mutex and n-1 readers wait on mutex

Readers-Writers Problem Variations

- ***First*** variation – no reader is kept waiting unless a writer is writing
- ***Second*** variation – once writer is ready, it performs the write ASAP. That is, if a writer is waiting, no new readers may start reading
- Both may have starvation

Dining-Philosophers Problem



- Five philosophers spend their lives alternating between thinking and eating
- To eat, a philosopher must pick up 2 chopsticks on each side, one at a time
  - Release both chopsticks when done

Semaphore Solution to Dining-Philosophers

- Shared data
  - Semaphore chopstick [5], all elements initialized to 1
- Structure of Philosopher i :

```
while (true){
     wait (chopstick[i] );
     wait (chopStick[ (i + 1) % 5] );

      /* eat for awhile */

     signal (chopstick[i] );
     signal (chopstick[ (i + 1) % 5] );

      /* think for awhile */

}
```

- Any problem with this algorithm?
  - Deadlock!

Dining-Philosophers Problem: Deadlock Handling

- Some remedies to the deadlock problem:
  1. Allow at most 4 philosophers to be sitting simultaneously at the table.
  2. Allow a philosopher to pick up the chopsticks only if both are available (picking up must be done in a critical section).
  3. An odd-numbered philosopher first picks up the left chopstick and then the right chopstick; an even-numbered philosopher first picks up the right chopstick and then the left chopstick.

Monitor Solution to Dining Philosophers

```
monitor DiningPhilosophers
{
   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);
   }
```

Monitor Solution to Dining Philosophers (Cont.)

```
        void test (int i) {
                if ((state[(i + 4) % 5] != EATING) &&
                (state[i] == HUNGRY) &&
                (state[(i + 1) % 5] != EATING) ) {
                     state[i] = EATING ;
                     self[i].signal () ;
                }
        }

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

Monitor Solution to Dining Philosophers (Cont.)

- Each philosopher ***i*** invokes the operations **`pickup()`** and **`putdown()`** in the following sequence:

```
DiningPhilosophers.pickup(i);

    /** EAT **/

DiningPhilosophers.putdown(i);
```

- No deadlock, but starvation is possible

POSIX Synchronization

- POSIX API provides
  - Mutex locks
  - Semaphores
  - Condition variables
- Widely used by developers on UNIX, Linux, and macOS systems

POSIX Mutex Locks

- A mutex lock is used to protect critical sections of code
- A mutex lock has two possible states
  - **unlocked** - not owned by any thread
  - **locked** - owned by a thread
  - Initial state is **unlocked**
- Only one thread can own a mutex lock at any given time
- Include <pthread.h> to use mutex locks

Creating/Destroying a Mutex

int pthread_mutex_init(pthread_mutex_t *mutex,

const pthread_mutexattr_t *mutexattr)

- This function initializes a mutex lock
- First parameter is a pointer to the mutex
- Second parameter specifies the attributes of the mutex
  - If mutexattr is NULL, default attributes are used
- Return 0 on success; otherwise, an error number is returned

int pthread_mutex_destroy(pthread_mutex_t *mutex)

- This function destroys a mutex
  - The mutex must be unlocked when called
  - Attempting to destroy a locked mutex results in undefined behavior
- Return 0 on success; otherwise, an error number is returned

Locking/Unlocking a Mutex

int pthread_mutex_lock(pthread_mutex_t *mutex) – acquire the mutex lock

- If the mutex is unlocked, it becomes locked and owned by the calling thread
- If the mutex is already locked, the calling thread blocks until the mutex is unlocked
- Return 0 on success; otherwise, an error number is returned

int pthread_mutex_unlock(pthread_mutex_t *mutex) – release the mutex lock

- This function unlocks a mutex if called by the owning thread
  - An error will be returned if the mutex is owned by another thread
- Return 0 on success; otherwise, an error number is returned

POSIX Semaphores (1)

Include <semaphore.h> to use semaphores

int sem_init(sem_t *sem, int pshared, unsigned int value);

• The function initializes an unnamed semaphore pointed to by **sem**

• pshared indicates whether the semaphore is to be shared between the threads of a process or between processes

  – **pshared** = 0 – the semaphore is shared between the threads of a process

  – **pshared** ≠ 0 – the semaphore is shared between processes, and should be located in a region of shared memory

• The initial value of the semaphore is set to **value**

• Return 0 on success, -1 on error

POSIX Semaphores (2)

int sem_wait(sem_t * sem) – decrement a semaphore

• If semaphore value > 0, then the value is atomically decremented

• If semaphore value = 0, then the call blocks until the value is positive and then the value is atomically decremented

• Return 0 on success, -1 on error

int sem_post(sem_t * sem) – increment a semaphore

• This function atomically increments the value of the semaphore; if the value becomes positive, then another thread blocked in a **sem_wait()** call will be woken up

• Return 0 on success, -1 on error

int sem_destroy(sem_t * sem) – destroy a semaphore

• No threads should be blocked on the semaphore when called

– Destroying a semaphore on which other threads are currently blocked produces undefined behavior

• Return 0 on success, -1 on error

POSIX Condition Variables

- A condition variable allows a thread to suspend execution until a condition on shared data holds
- A condition variable is always used in conjunction with a mutex lock
  - The mutex is used to protect the data in the conditional clause from a possible race condition
  - A thread must acquire the mutex before checking the condition
- Include <pthread.h> to use condition variables

Creating/Destroying Condition Variables

int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr);

- This function initializes the condition variable specified by cond with attributes specified by cond_attr
  - If cond_attr is NULL, default attributes are used
- Return 0 on success; otherwise, an error number is returned.

int pthread_cond_destroy(pthread_cond_t *cond)

- This function destroys the condition variable specified by cond
- No threads should be blocked on the condition variable when the function is called
  - Attempting to destroy a condition variable upon which other threads are currently blocked results in undefined behavior
- Return 0 on success; otherwise, an error number is returned.

Wait/Signal on Condition Variables

int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex) – wait on a condition

- mutex must be locked by the calling thread when this function is called
- The function atomically unlocks the mutex and blocks the calling thread on cond (so that another thread could update variables)
- Before returning to the calling thread, the function re-locks mutex
- Return 0 on success; otherwise, an error number is returned

int pthread_cond_signal(pthread_cond_t *cond) – signal a condition

- The function unblocks one of the threads that are blocked on cond; nothing happens if no threads are blocked on cond
- The function should be called after the mutex associated with cond is locked
- The mutex associated with cond should be unlocked after calling this function
- Return 0 on success; otherwise, an error number is returned

Example

Thread A

```
pthread_mutex_lock(&mutex);
while (a != b)
     pthread_cond_wait(&cond_var, &mutex);
pthread_mutex_unlock(&mutex);
```

Thread B

```
pthread_mutex_lock(&mutex);
a = b;
pthread_cond_signal(&cond_var);
pthread_mutex_unlock(&mutex);
```

Project 2 Released

- You should be ready to do Project 2
- Start early!!!