Operating System Concepts

Lecture 20: Synchronization Examples

Omid Ardakanian
oardakan@ualberta.ca
University of Alberta

MWF 12:00-12:50 VVC 2 215

Today’s class

- Classic synchronization problems
- POSIX synchronization

CLASSIC PROBLEMS OF SYNCHRONIZATION

The Bounded-Buffer Problem

- shared data: a finite buffer
- requirements
  - mutual exclusion: no lost items & no corrupt writes
  - producers must wait if buffer is full
  - consumers must wait if buffer is empty
- real world example: a web server

The Bounded-Buffer Problem

- shared data: a finite buffer
- requirements
  - mutual exclusion: no lost items & no corrupt writes
  - producers must wait if buffer is full
  - consumers must wait if buffer is empty
- real world example: a web server

```
class BoundedBufferMonitor{
  public:
  void P(Item item);
  void C(Item &item);
  private:
  Item buffer[N];
  int last, count;
  Condition full, empty;
  Lock lock;
}

BoundedBufferMonitor::BoundedBufferMonitor(){
  count = 0; // we need to keep track of the used spaces
  last = 0;
}
```

The Bounded-Buffer Problem

- shared data: a finite buffer
- requirements
  - mutual exclusion: no lost items & no corrupt writes
  - producers must wait if buffer is full
  - consumers must wait if buffer is empty
- real world example: a web server

```
class BoundedBufferMonitor{
  public:
  void P(Item item);
  void C(Item &item);
  private:
  Item buffer[N];
  int last, count;
  Condition full, empty;
  Lock lock;
}

BoundedBufferMonitor::BoundedBufferMonitor(){
  count = 0; // we need to keep track of the used spaces
  last = 0;
}
```

You can also write it with binary and counting semaphores

The Bounded-Buffer Problem

```
void BoundedBufferMonitor::P(Item item){
  lock.Acquire();
  while(count == N)
    empty.Wait(lock);
  buffer[last] = item;
  last = (last + 1)%N;
  count++;
  full.Signal();
  lock.Release();
}

void BoundedBufferMonitor::C(Item &item){
  lock.Acquire();
  while(count == 0)
    full.Wait(lock);
  item = buffer[(last-count)%N]; // first=(last-count) mod N
  count--;
  empty.Signal();
  lock.Release();
}
```

The Readers-Writers Problem

- shared data: a buffer or database
- two types of threads: readers and writers
  - readers only read the shared data without modifying it
  - writers read the data and modify it

The Readers-Writers Problem

- shared data: a buffer or database
- two types of threads: readers and writers
  - readers only read the shared data without modifying it
  - writers read the data and modify it
- requirements
  - writers must have exclusive access to the shared data
  - multiple readers should be allowed to access shared data simultaneously as long as there is no writer

The Readers-Writers Problem

- shared data: a buffer or database
- two types of threads: readers and writers
  - readers only read the shared data without modifying it
  - writers read the data and modify it
- requirements
  - writers must have exclusive access to the shared data
  - multiple readers should be allowed to access shared data simultaneously as long as there is no writer
- real world example: account accesses and updates in a banking application

The Readers-Writers Problem

- shared data: a buffer or database
- two types of threads: readers and writers
  - readers only read the shared data without modifying it
  - writers read the data and modify it
- requirements
  - writers must have exclusive access to the shared data
  - multiple readers should be allowed to access shared data simultaneously as long as there is no writer
- real world example: account accesses and updates in a banking application
- why do we use different logics for readers and writers?
  - using a single lock is overly restrictive
  - helps a lot if
    - there are many readers but only a few writers
    - we can easily differentiate readers from writers (i.e., a thread does not alternate between reading and writing)

The Readers-Writers Problem

```
class ReadWrite {
  public:
  void Read();
  void Write();
  private:
  int readers;      // number of readers (shared between readers)
  Semaphore mutex; // controls access to readers
  Semaphore wrt;   // controls entry to first writer or reader
}

ReadWrite::ReadWrite {
  readers = 0;
  mutex.value = 1;
  wrt.value = 1;
}
```

The Readers-Writers Problem

```
void ReadWrite::Write(){
  wrt.Wait();    // any writers or readers in the critical section
  <perform write>
  wrt.Signal();
}

void ReadWrite::Read(){
  mutex.Wait(); // mutual exclusion for data shared between readers
  readers += 1;      // increment the number of readers
  if(readers == 1)  // it's the first reader
    wrt.Wait();      // block writers or this reader
  mutex.Signal();

  <perform read>

  mutex.Wait(); // mutual exclusion for data shared between readers
  readers -= 1;       // reader is done
  if (readers == 0)  // it's the last reader
    wrt.Signal();     // unblock writers
  mutex.Signal();
}
```

The Readers-Writers Problem

```
void ReadWrite::Write(){
  wrt.Wait();    // any writers or readers in the critical section
  <perform write>
  wrt.Signal();
}

void ReadWrite::Read(){
  mutex.Wait(); // mutual exclusion for data shared between readers
  readers += 1;      // increment the number of readers
  if(readers == 1)   // it's the first reader
    wrt.Wait();      // block writers or this reader
  mutex.Signal();

  <perform read>

  mutex.Wait(); // mutual exclusion for data shared between readers
  readers -= 1;       // reader is done
  if (readers == 0)   // it's the last reader
    wrt.Signal();     // unblock writers
  mutex.Signal();
}
```

first critical section

second critical section

Scenario 1

```
Reader Thread 1          Reader Thread 2          Writer Thread 1

ReadWrite::Read()

                         ReadWrite::Read()


                         calls wrt.Signal();

                                                  ReadWrite::Write()
```

Scenario 2

```
Writer Thread 1          Reader Thread 1             Reader Thread 2

ReadWrite::Write()
calls wrt.Wait();

                         ReadWrite::Read()
                         blocks at wrt.Wait();

                                                     ReadWrite::Read()
                                                     blocks at mutes.Wait();

                         unblocks to perform read
                                                     unlocks to perform read
```

Scenario 3

Reader Thread 1	Writer Thread 1	Reader Thread 2
`ReadWrite::Read()`		
	`ReadWrite::Write()` blocks at `wrt.Wait();`	
		`ReadWrite::Read()`
	unblocks to perform write	calls `wrt.Signal();`

Scenario 3

Reader Thread 1	Writer Thread 1	Reader Thread 2
`ReadWrite::Read()`		
	`ReadWrite::Write()` `blocks at wrt.Wait();`	
		`ReadWrite::Read()`
	`unblocks to perform write`	`calls wrt.Signal();`

Writer has to wait until all readers are finished!

Discussions

- implementation notes
  - the first reader blocks (inside the first critical section) if there is a writer; any other readers who try to enter block on mutex
  - the last reader to exit signals a waiting writer
  - when a writer exits, if there is both a reader and a writer waiting, the scheduler determines which one goes next
    - if a reader goes next, then all readers that are waiting will fall through (at least one is waiting on wrt and zero or more can be waiting on mutex)
    - it a writer goes next, then readers will continue waiting

Discussions

- implementation notes
  - the first reader blocks (inside the first critical section) if there is a writer; any other readers who try to enter block on mutex
  - the last reader to exit signals a waiting writer
  - when a writer exits, if there is both a reader and a writer waiting, the scheduler determines which one goes next
    - if a reader goes next, then all readers that are waiting will fall through (at least one is waiting on wrt and zero or more can be waiting on mutex)
    - it a writer goes next, then readers will continue waiting
- alternative desirable semantics
  - let a writer enter its critical section as soon as possible

Favouring writers

to block readers as soon as a writer enters, we need to keep track of the number of writers and use a writer mutex lock to update this number

Favouring writers

to block readers as soon as a writer enters, we need to keep track of the number of writers and use a writer mutex lock to update this number

```
void ReadWrite::Write(){
  write_mutex.Wait(); // ensure mutual exclusion
  writers += 1;       // another pending writer
  if(writers == 1)    // first writer blocks readers
    read_block.Wait();
  write_mutex.Signal();

  write_block.Wait(); // ensure mutual exclusion
  <perform write>
  write_block.Signal();

  write_mutex.Wait(); // ensure mutual exclusion
  writers -= 1;       // writer done
  if(writers == 0)    // enable readers
    read_block.Signal();
  write_mutex.Signal();
}
```

Favouring writers

to block readers as soon as a writer enters, we need to keep track of the number of writers and use a writer mutex lock to update this number

first critical section

```
void ReadWrite::Write(){
  write_mutex.Wait(); // ensure mutual exclusion
  writers += 1;       // another pending writer
  if(writers == 1)    // first writer blocks readers
    read_block.Wait();
  write_mutex.Signal();

  write_block.Wait(); // ensure mutual exclusion
  <perform write>
  write_block.Signal();
```

second critical section

```
  write_mutex.Wait(); // ensure mutual exclusion
  writers -= 1;       // writer done
  if(writers == 0)    // enable readers
    read_block.Signal();
  write_mutex.Signal();
}
```

Favouring writers

how to modify the `Read()` method?

- it’s your homework

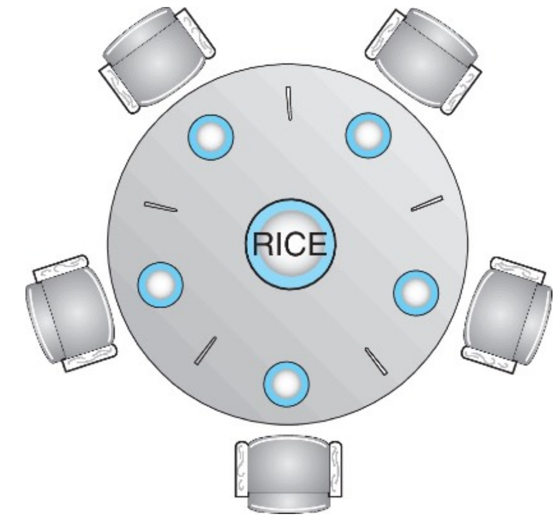
Read/Write locks

- pthreads support read/write locks
  - a thread can acquire a read lock or a write lock
  - many threads can hold the same read lock concurrently
  - only one thread can hold a write lock
- use pthread functions:

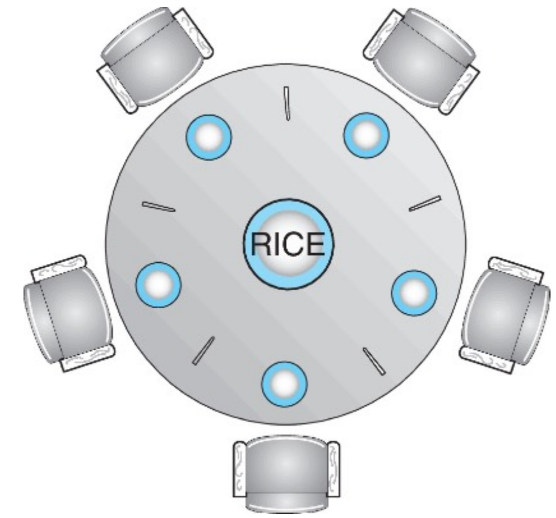
```
pthread_rwlock_init()
pthread_rwlock_rdlock()
pthread_rwlock_wrlock()
pthread_rwlock_unlock()
```

The Dinning-Philosophers Problem

- shared data: a fixed number of chopsticks
- setting:
  - philosophers sit around a circular table
  - chopsticks are placed between philosophers
  - hence, #philosophers = #chopsticks

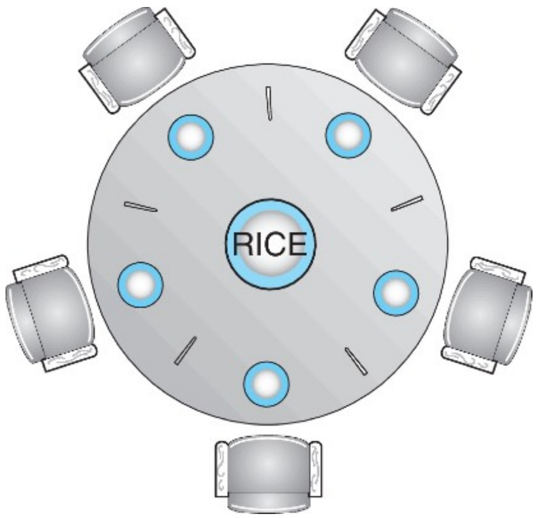


The Dinning-Philosophers Problem



- shared data: a fixed number of chopsticks
- setting:
  - philosophers sit around a circular table
  - chopsticks are placed between philosophers
  - hence, #philosophers = #chopsticks
- requirements
  - philosophers spend their lives alternating thinking and eating (**what a boring life!**)
    - they occasionally get hungry and decide to eat!
    - they don't like to talk to each other and coordinate
  - philosophers need two chopsticks to eat from bowl
    - but can pick up one at a time
    - they can only pick up the two chopsticks that are closest to them
  - philosophers put down chopsticks when finished
  - they seek a deadlock-free and starvation-free solution

The Dinning-Philosophers Problem



- shared data: a fixed number of chopsticks
- setting:
  - philosophers sit around a circular table
  - chopsticks are placed between philosophers
  - hence, #philosophers = #chopsticks
- requirements
  - philosophers spend their lives alternating thinking and eating (**what a boring life!**)
    - they occasionally get hungry and decide to eat!
    - they don't like to talk to each other and coordinate
  - philosophers need two chopsticks to eat from bowl
    - but can pick up one at a time
    - they can only pick up the two chopsticks that are closest to them
  - philosophers put down chopsticks when finished
  - they seek a deadlock-free and starvation-free solution
- real world example:
  - cooperating processes shared constrained resources
  - travel reservation: hotel, airline, car rental

First attempt

```
#define N  5              // number of philosophers
#define L  i              // index of i's left neighbour
#define R  (i+1)%N        // index of i's right neighbour

semaphore chopstick[N]; // one semaphore per chopstick

void philosopher(int i) { // i is the index of the philosopher
  while(true) {
    chopstick[L].wait();
    chopstick[R].wait();
    /* eat for awhile */
    chopstick[L].signal();
    chopstick[R].signal();
    /* think for awhile */
  }
}
```

First attempt

```
#define N  5             // number of philosophers
#define L  i             // index of i's left neighbour
#define R  (i+1)%N       // index of i's right neighbour

semaphore chopstick[N]; // one semaphore per chopstick

void philosopher(int i) { // i is the index of the philosopher
  while(true) {
    chopstick[L].wait();
    chopstick[R].wait();
    /* eat for awhile */
    chopstick[L].signal();
    chopstick[R].signal();
    /* think for awhile */
  }
}
```

What's the problem with this solution? **could create a deadlock**

Second attempt

```
#define N  5              // number of philosophers
#define L  (i+N-1)%N      // index of i's left neighbour
#define R  (i+1)%N        // index of i's right neighbour
#define THINKING  0
#define HUNGRY    1
#define EATING    2

int state[N];             // array that keeps everyone's state
semaphore mutex = 1;      // binary semaphore (mutual exclusion)
semaphore s[N];           // one semaphore per philosopher

void philosopher(int i) { // i is the index of the philosopher
  while(true) {
    think();              // return when philosopher gets hungry
    takeForks(i);
    eat();                // eating can take a while...
    releaseForks(i);
  }
}
```

Second attempt

```
void takeForks(int i) { // i is the index of the philosopher
  mutex.wait();
  state[i] = HUNGRY;    // update the state
  test(i);              // pick up the two chopsticks and eat, if possible
  mutex.signal();
  s[i].wait();
}

void releaseForks(int i) { // i is the index of the philosopher
  mutex.wait();
  state[i] = THINKING;  // update the state
  test(L);              // let the left neighbour eat, if possible
  test(R);              // let the right neighbour eat, if possible
  mutex.signal();
}

void test(int i) { // i is the index of the philosopher
  if(state[i] == HUNGRY && state[L] != EATING && state[R] != EATING) {
    // can pick up two chopsticks and eat
    state[i] = EATING;
    s[i].signal();
  }
}
```

Second attempt

```
void takeForks(int i) { // i is the index of the philosopher
  mutex.wait();
  state[i] = HUNGRY;    // update the state
  test(i);              // pick up the two chopsticks and eat, if possible
  mutex.signal();
  s[i].wait();
}

void releaseForks(int i) { // i is the index of the philosopher
  mutex.wait();
  state[i] = THINKING;  // update the state
  test(L);              // let the left neighbour eat, if possible
  test(R);              // let the right neighbour eat, if possible
  mutex.signal();
}

void test(int i) { // i is the index of the philosopher
  if(state[i] == HUNGRY && state[L] != EATING && state[R] != EATING) {
    // can pick up two chopsticks and eat
    state[i] = EATING;
    s[i].signal();
  }
}
```

Is it a good solution?

POSIX SYNCHRONIZATION

POSIX synchronization

- POSIX API provides
  - mutex locks
  - semaphores
  - condition variables
- they are widely used on UNIX, Linux, and macOS

POSIX mutex locks

- creating and initializing the lock

```
#include <pthread.h>

pthread_mutex_t mutex;

/* create and initialize the mutex lock */
pthread_mutex_init(&mutex, NULL);
```

- acquiring and releasing the lock

```
/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/* critical section */

/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

POSIX semaphores

- POSIX provides two versions — named and unnamed
- named semaphores can be used by unrelated processes, unnamed semaphores cannot be used in this case
- see https://linux.die.net/man/7/sem_overview

POSIX named semaphores

- creating an initializing the semaphore:

```
#include <semaphore.h>
sem_t *sem;

/* create the semaphore and initialize it to 1 */
see = sem_open("SEM_NAME", O_CREAT, 0666, 1);
```

- another process can access the semaphore by referring to its name SEM

- acquiring and releasing the semaphore:

```
/* acquire the semaphore */
sem_wait(sem);

/* critical section */

/* release the semaphore */
sem_post(sem);
```

POSIX unnamed semaphores

- creating an initializing the semaphore:

```
#include <semaphore.h>
sem_t sem;

/* create the semaphore and initialize it to 1 */
sem_init(&sem, 0, 1); // second arg determines if sem is shared
                      // between threads of a process or between
                      // processes
```

- acquiring and releasing the semaphore:

```
/* acquire the semaphore */
sem_wait(&sem);

/* critical section */

/* release the semaphore */
sem_post(&sem);
```

POSIX condition variables

- POSIX condition variables are associated with a POSIX mutex lock to provide mutual exclusion
- creating and initializing the condition variable:

```
pthread_mutex_t mutex;
pthread_cond_t cond_var;

pthread_mutex_init(&mutex,NULL);
pthread_cond_init(&cond_var,NULL);
```

POSIX condition variables

- thread waiting for the condition `a == b` to become true:

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

- thread signalling another thread waiting on the condition variable:

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

Linux synchronization

- prior to kernel version 2.6
  - disables interrupts (kernel preemption) to implement short critical sections
- version 2.6 and later
  - fully preemptive
- Linux provides:
  - semaphores
  - atomic integers
  - spinlocks
  - reader-writer versions of both