CSCI 4730/6730 OS
(Chap #7 Sync. Examples – Part 3)

In Kee Kim

Department of Computer Science

University of Georgia

Midterm Exam

- Date: Oct 12th – Tuesday
- Time: 11:10 to 12:25 – 75 mins
- Location: Classroom (GEO 200B)
- Close-book, close-note, close-laptop

Midterm Exam

Undergrad

Grads

Mandatory
100/100

7-10 True/False or Multiple Choice Questions

5 Short Essay Questions

Mandatory

Optional

N extra Questions
- (May be) Short Essays

UGs can pick/solve 2 grad questions (5 points for each) and can get up to 10 points of extra credit
110/100

Midterm Exam

- Paper-based
  - If I or TA cannot understand your writing…
- Could be on the exam:
  - Everything that we’ve discussed in the classroom
  - Everything on the PDF (“slides”) in eLC
  - Two Quizzes
  - Programming Assignment #1
- To study
  - Focus on the PDFs (“slides”) in eLC
  - Review your notes from class discussions

Midterm

- Chapter 1 – Overview
- Chapter 2 – Operating Systems Structures
- Chapter 3 – Processes
- Chapter 4 – Thread and Concurrency
- Chapter 5 – CPU Scheduling
- Chapter 6 – Sync. Tools
- Chapter 7 – Sync. Examples

Thursday and Friday

- Thursday Class (10/7)
  - Q&A Session for Midterm
  - Ask questions

- Friday (10/8)
  - Office Hour for Midterm
  - From 3 p.m. to 5 p.m. @ 802 in BOYD

Where are we?

- Classical Problems in Synchronization
  - Bounded-Buffer Problem
  - **Readers and Writers Problem**
  - Dining-Philosophers Problem

Readers-Writers Problem - Requirements

1. Two or more readers access the shared data at the same time, no adverse effects will result
2. No simultaneous access from a writer and some other readers/writers – no reading while data is being updated
   - Writers should have exclusive access to the shared data.
   - In other words, if a writer is updating (writing) data, no other readers or writers are allowed to access the data.

Readers-Writers Problem

- You have a global variable **`count`**
- Reader
  - Do “**`printf count`**”
- Writer
  - Do “**`count = count*10;`**”
- Let’s use binary semaphore (or mutex)
  - We need two semaphores
  - **`rw_mutex`**
  - **`mutex`**

How about this?

```
writer() {
  lock(&rw_mutex);
  /* writing is performed */
  /* count = count * 10 */
  unlock(&rw_mutex);
  }

reader() {
  lock(&mutex);
  if(++read_count == 1)
       lock(&rw_mutex);
  unlock(&mutex);
  /* reading is performed */
  /* printf ("count = %d\n", count);
  lock(&mutex)
  if(--read_count == 0) unlock(&rw_mutex);
  unlock(&mutex);
}
```

This is one approach introduced in Textbook

- Shared Data
  - Data set
  - Semaphore **`rw_lock`** initialized to 1
  - Semaphore **`mutex`** initialized to 1
  - (Global) Integer **`read_count`** initialized to 0

Readers-Writers Problem

- The structure of a **`writer`** process

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

Readers-Writers Problem

- The structure of a **reader** process

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

Let's Implement "Readers-Writers Problem"

- APIs
  - **`sem_init`**
  - **`sem_destroy`**
  - **`sem_wait`** → **`wait()`**
  - **`sem_post`** → **`signal ()`**

Base Code

nike.cs.uga.edu - PuTTY

```c
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>

int max_writer = 1;
int max_reader = 13;
long count = 1; // shared data

void *writer(void *s)
{
    int self = (int *)s;

    count = count*10;
    printf("Writer %d modified count to %ld\n",self,count);

}

void *reader(void *s)
{
  int self = (int *)s;

  // Reading Section
  printf("Reader %d: read count as %ld\n",self,count);

}

int main()
{

  long i = 0;
  pthread_t read[max_reader],write[max_writer];


  for(i = 0; i < max_writer; i++)
    pthread_create(&write[i], NULL, (void *)writer, (void *)i);
  for (i = 0 ; i < max_reader; i++)
    pthread_create(&read[i], NULL, (void *)reader, (void *)i);

  for(i = 0; i < max_reader; i++)
    pthread_join(read[i], NULL);
  for(i = 0; i < max_writer; i++)
    pthread_join(write[i], NULL);

  return 0;
}
```

1,1 Top

Global variable

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>

sem_t rw_mutex;
sem_t mutex;

int max_writer = 1;
int max_reader = 13;

long count = 1; // shared data
int read_count = 0;
```

Main function

```
int main()
{
  long i = 0;
  pthread_t read[max_reader],write[max_writer];

  //initialize semaphore
  sem_init (&rw_mutex, 0, 1);
  sem_init (&mutex, 0, 1);

  for(i = 0; i < max_writer; i++)
    pthread_create(&write[i], NULL, (void *)writer, (void *)i);
  for (i = 0 ; i < max_reader; i++)
    pthread_create(&read[i], NULL, (void *)reader, (void *)i);

  for(i = 0; i < max_reader; i++)
    pthread_join(read[i], NULL);
  for(i = 0; i < max_writer; i++)
    pthread_join(write[i], NULL);

  sem_destroy(&rw_ mutex);
  sem_destroy(&mutex);

  return 0;
}
```

Writer code

```
void *writer(void *s)
{
    int self = (int *)s;

    sem_wait (&rw_mutex);
    count = count*10;
    printf("Writer %d modified count to %ld\n",self,count);
    sem_post (&rw_mutex);

}
```

Reader code

```
void *reader(void *s)
{
  int self = (int *)s;

  sem_wait(&mutex);
  read_count++;
  if (read_count == 1)
     sem_wait(&rw_mutex);
  sem_post(&mutex);

  // Reading Section
  printf("Reader %d: read count as %ld\n",self,count);

  sem_wait(&mutex);
  read_count--;
  if(read_count == 0) {
    sem_post(&rw_mutex); // If this is the last reader, it will wake up the writer.
  }
  sem_post(&mutex);

}
```

Limitation?

- **Starvation with multiple readers**
- Who will starve?
- Suppose there are multiple readers, **`R1`**, **`R2`**, …
  - It is possible that a reader **`R1`** might have "**`rw_mutex`**", a writer **`Writer`** be waiting for "**`rw_mutex`**", and then a reader **`R2`** requests access.
  - It would be unfair for **`R2`** to jump in immediately, ahead of **`Writer`**; if that happened often enough, **`Writer`** would starve.

Real-World Examples of Readers-Writers Problem

- Any Data Management/Sharing Applications
  - DBMS (Data Management Systems)
- Systems with concurrent readers and writers
- Banking Systems
  - Read Account Balances vs. Updates

Extra: Bounded-Buffer Problem Again

- This is important for your next programming assignment
- Three requirements:
  - The Producer (**P**) must not insert item when buffer is full
  - The Consumer (**C**) must not remove item when buffer is empty
  - **P** and **C** should not insert and remove must at the same time

Producer:
```
while (true) {
    /* produce an item in
    next produced */

    while (counter == BUFFER_SIZE);
    /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

Consumer:
```
while (true) {
    while (counter == 0);
    /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    counter--;
    /* consume the item in next
    consumed */
}
```

Extra: Bounded-Buffer Problem Again

- Semaphore **`mutex`** initialized to the value ***1***
  - To protect **`count`** operation
- Semaphore **`full`** initialized to the value ***0***
  - Number of full buffers
- Semaphore **`empty`** initialized to the value ***n***
  - Number of empty buffers

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```

Extra: Bounded-Buffer Problem Again

Producer (w/ Semaphore)

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

Consumer (w/ Semaphore)

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

Extra: Bounded-Buffer Problem: Implementation

- Let's use two semaphores and one mutex lock

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```

```
int n_items_in_buffer = 0;
pthread_mutex_t lock;
sem_t sem_empty;
sem_t sem_full;
```

- APIs
  - **`pthread_mutex_init`**
  - **`pthread_mutex_lock & pthread_mutex_unlock`**
  - **`sem_init & sem_destroy`**
  - **`sem_wait & sem_post`**

Extra: Bounded-Buffer Problem: Base Code

nike.cs.uga.edu - PuTTY

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <semaphore.h>

int n_items_in_buffer = 0;
pthread_mutex_t lock;
sem_t sem_empty;
sem_t sem_full;

int MAX_SLOT = 5;
int NUM_CONSUMER = 3;

void *producer() {
  while(1) {
    printf("[Producer] produce an item\n");

    n_items_in_buffer++;
    printf("[Producer] added the item to the buffer. # items in buffer = %d\n"
          , n_items_in_buffer);
    usleep(rand()%500000); // processing overhead

  }
}

void *consumer(void *arg) {
  printf("[Consumer %d] joined.\n", syscall(186)-getpid());
  while(1) {

    n_items_in_buffer--;
    printf("[Consumer %d] removed an item from the buffer. # items in buffer = %d\n"
          , syscall(186)-getpid(), n_items_in_buffer);
    usleep(rand()%800000); // processing overhead
  }
}

                                                        14,1          Top
```

nike.cs.uga.edu - PuTTY

```c
int main(int argc, char *argv[])
{
  int i;

  pthread_t p;
  pthread_t c[10];

  for(i = 0; i < NUM_CONSUMER; i++)
  {
    pthread_create(&(c[i]), NULL, consumer, NULL);
  }
  pthread_create(&p, NULL, producer, NULL);

  for(i = 0; i < NUM_CONSUMER; i++)
    pthread_join(c[i], NULL);
  pthread_join(p, NULL);

  return 0;
}
                                                        57,1          Bot
```

Bounded-Buffer: Main function

```
int main(int argc, char *argv[])
{
  int i;
  pthread_t p, c[10];

  pthread_mutex_init(&lock, NULL);
  sem_init(&sem_empty, 0, MAX_SLOT);
  sem_init(&sem_full, 0, 0);

  for(i = 0; i < NUM_CONSUMER; i++)
     pthread_create(&(c[i]), NULL, consumer, NULL);
     pthread_create(&p, NULL, producer, NULL);

  // join waits for the threads to finish
  for(i = 0; i < NUM_CONSUMER; i++)
    pthread_join(c[i], NULL);
  pthread_join(p, NULL);
  sem_destroy (
  return 0;
}
```

Bounded-Buffer: Producer Code

```
void *producer() {
  while(1) {
    printf("[Producer] produce an item\n");

    sem_wait(&sem_empty);
    pthread_mutex_lock(&lock);

    n_items_in_buffer++;
    printf("[Producer] added the item to the buffer. # items in buffer = %d\n", n_items_in_buffer);
    usleep(rand()%500000);

    pthread_mutex_unlock(&lock);
    sem_post(&sem_full);
  }
}
```

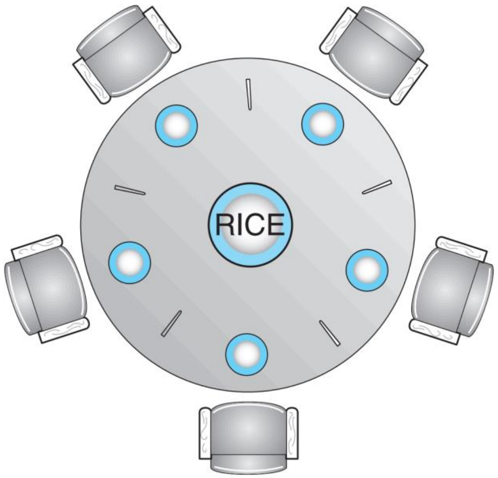
Bounded-Buffer: Consumer Code

```
void *consumer(void *arg) {
  printf("[Consumer %d] joined.\n", syscall(186)-getpid());
  while(1) {
    sem_wait(&sem_full);
    pthread_mutex_lock(&lock);

    n_items_in_buffer--;
    printf("[Consumer %d] removed an item from the buffer. # 
items in buffer = %d\n", syscall(186)-getpid(), 
n_items_in_buffer);
    usleep(rand()%800000);

    pthread_mutex_unlock(&lock);
    sem_post(&sem_empty);
  }
}
```

Classical Problems of Synchronization

- Classical Problems in Synchronization
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - **Dining-Philosophers Problem**

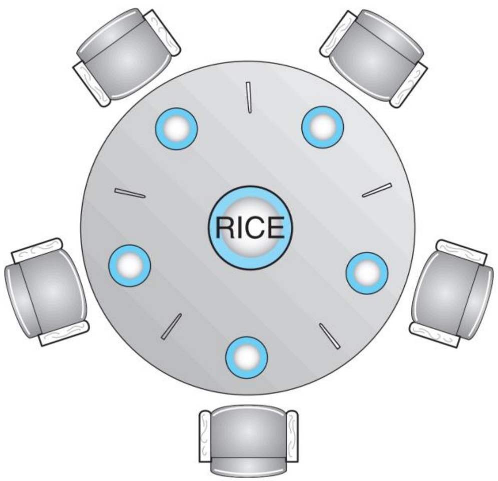
Dining-Philosophers Problem



N philosophers sit at a round table with a bowel of rice in the middle.

- Philosophers spend their lives alternating **thinking** and **eating**
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done

Dining-Philosophers Problem



Philosophers

Thinking

No interaction w/ others

Eating

When gets hungry, tries to pick up two chopsticks.

Can pick up only one chopstick at a time.

Cannot pick up chopstick that is already picked up by neighbor

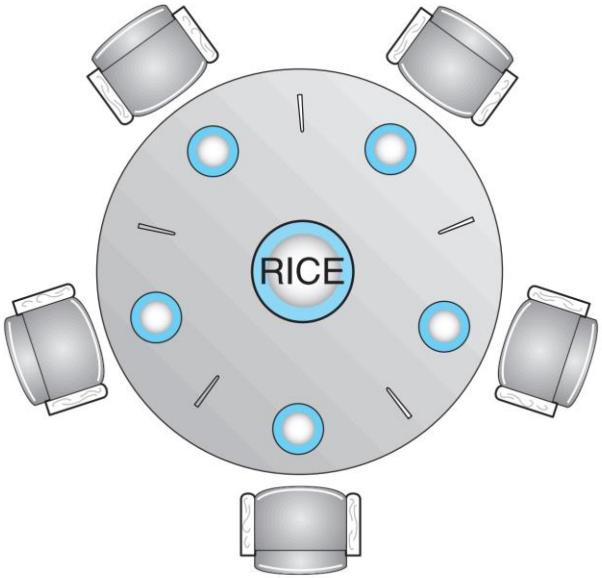
When having two chopsticks, can eat w/o releasing chopsticks

When finished, releasing both chopsticks, and starts thinking again…

- Challenge??
  - Two philosophers cannot access a chopstick at the same time!

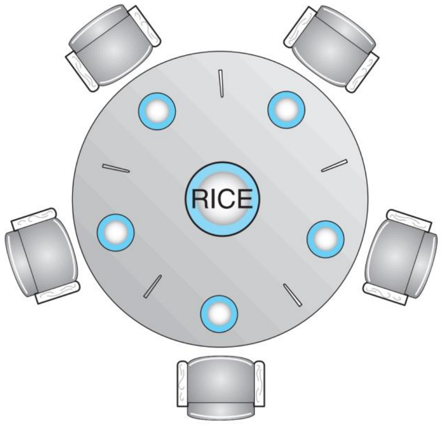
Dining-Philosophers Problem

- What would be shared data?
  - Bowl of rice (data set)
  - Semaphore **chopstick[5]**
    - Binary or Counting Semaphore?
    - Initialized value?
    - Initialized to 1



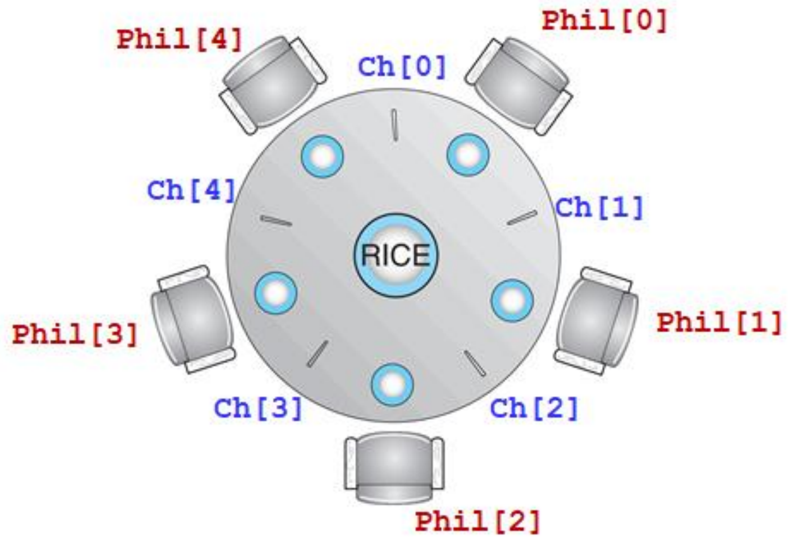
Dining-Philosophers Problem

- Approach
  1. Represent each **`chopstick`** as a semaphore
  2. A philosopher tries to grab a chopstick by executing a **`wait()`** operation
  3. Rhilosopher releases her chopsticks by executing **`signal()`** operation
  4. The shared data are semaphores (binary semaphore) **`chopstick[5]`**, where all chopsticks are initialized to 1.
     - **`chopstick`** is available (free, value == 1)
     - **`chopstick`** is not available (used by someone else, value == 0)



The structure of Philosopher *i*

```
Semaphore chopstick[5];
while (true) {
   wait(chopstick[i]);  // right chopstick
   wait(chopstick[(i+1) % 5]); // left chopstick
   ...
   /* eat for a while */
   ...
   signal(chopstick[i]); // right chopstick
   signal(chopstick[(i+1) % 5]); // left chopstick
   ...
   /* think for awhile */
   ...
}
```



The structure of Philosopher *i*

1. Think until the right chopstick is available; when it is *available*, pick it up;
2. Think until the left chopstick is available; when it is *available*, pick it up;
3. Then both chopsticks are held, eat for a fixed amount of time;
4. Then, put the right chopstick down;
5. Then, put the left chopstick down;
6. Repeat from the beginning.

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

    ...
    /* eat for a while */
    ...

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

    ...
    /* think for awhile */
    ...
}
```

Pros and Cons

- Pros
  - No two neighbors are eating simultaneously
- Cons
  - **Deadlock**

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

   ...
   /* eat for a while */
   ...

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

   ...
   /* think for awhile */
   ...
}
```

Recap: What is Deadlock?

- **Deadlock** – A condition where two or more processes (threads) are waiting indefinitely for an event that can be caused by only one of the waiting processes

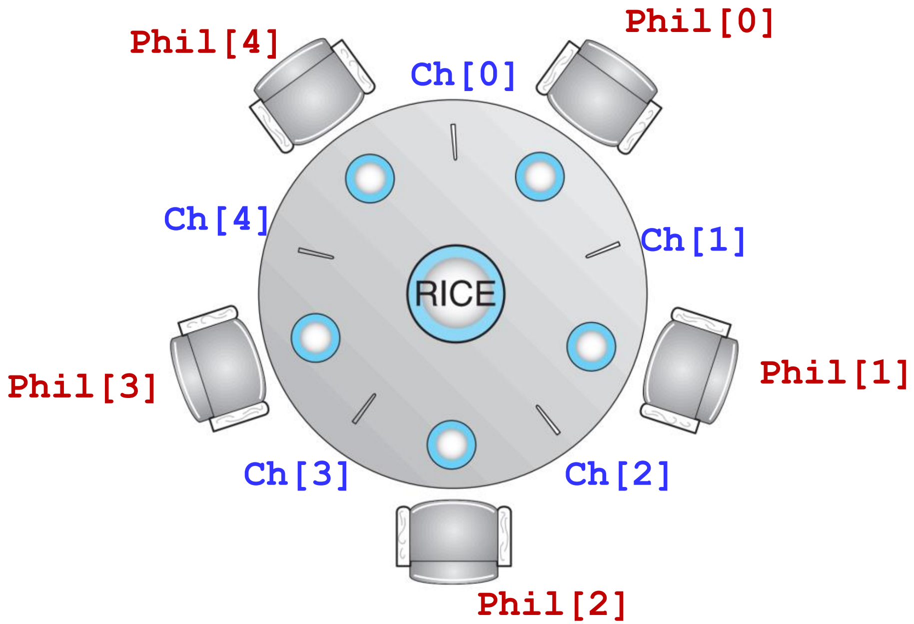
```
P0                  P1
wait (S);           wait (Q);
wait (Q);           wait (P);

...                 ...

signal(S);          signal(Q)
signal(Q);          signal(P)
```

Recap: What is Deadlock?

- What is the result of deadlock?

Deadlock in Dining-Philosophers Problem

- When does Deadlock happen?



Deadlock Handling

- How to avoid Deadlock?

1. Allow at most four philosophers to be sitting simultaneously at the table.
2. Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).
3. Use an asymmetric solution— that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even-numbered philosopher picks up her right chopstick and then her left chopstick.

Real-World Examples of Dining-Philosophers Problem

- Pretty common resource allocation problem
- **Cooperating processes** requiring shared limited resources
- Set of processes that need to lock multiple resources
  - Disk and backup tapes
  - Travel reservation: hotel, airlines, car rental databases
    - Successful reservation needs reservations of all of three.

End of Chapter 7

- Last Topic for Midterm