IOWA STATE UNIVERSITY

Department of Electrical and Computer Engineering

Lecture 14:
Classic IPC Problems

Agenda

- Recap
- Classic IPC Problems
  - Dining Philosophers Problem
  - Readers Writers Problem

Recap

- Semaphore
  - Synchronization method that provides more sophisticated ways (than mutex locks) for process to synchronize their activities
  - Semaphore **S** – integer variable
    - Can only be accessed via two indivisible (atomic) operations
      - **down()** and **up()**
        - originally called **P()** and **V()**
        - also called **wait()** and **signal()**

Recap

- Semaphore **S**
  - Definition of the **down()** operation (atomic!)

```
down(S) {
    while (S <= 0)
      ; // busy waiting
    S--;
}
```

  - Definition of the **up()** operation (atomic!)

```
up(S) {
    S++;
}
```

Recap

- Producer-Consumer Problem
  - Shared bounded buffer
    - A “Producer” process inserts item
    - A “Consumer” process removes item

Recap

- Producer-Consumer Problem
  - Solution with semaphore

```
#define N 100                           /* number of slots in the buffer */
typedef int semaphore;                  /* semaphores are a special kind of int */
semaphore mutex = 1;                    /* controls access to critical region */
semaphore empty = N;                    /* counts empty buffer slots */
semaphore full = 0;                     /* counts full buffer slots */

void producer(void)
{
        int item;

        while (TRUE) {                  /* TRUE is the constant 1 */
                item = produce_item( ); /* generate something to put in buffer */
                down(&empty);           /* decrement empty count */
                down(&mutex);           /* enter critical region */
                insert_item(item);      /* put new item in buffer */
                up(&mutex);             /* leave critical region */
                up(&full);              /* increment count of full slots */
        }
}
```

Recap

- Producer-Consumer Problem
  - Solution with semaphore

```
void consumer(void)
{
        int item;

        while (TRUE) {                          /* infinite loop */
                down(&full);                    /* decrement full count */
                down(&mutex);                   /* enter critical region */
                item = remove_item( );          /* take item from buffer */
                up(&mutex);                     /* leave critical region */
                up(&empty);                     /* increment count of empty slots */
                consume_item(item);             /* do something with the item */
        }
}
```

Recap

- Conditional Variables
  - Allows a thread to wait till a condition is satisfied
  - Testing the condition must be done within a mutex
  - A mutex is associated with every condition variable

- Example
  - Write a program using two threads
    - Thread 1 prints “hello”
    - Thread 2 prints “world”
    - Thread 2 should wait till thread 1 finishes before printing
  - Use condition variables

Recap

- Example

```
int    thread1_done = 0;

pthread_cond_t    cv;
pthread_mutex_t  mutex;
```

Thread 1:

```
printf("hello ");

pthread_mutex_lock(&mutex);
thread1_done = 1;
pthread_cond_signal(&cv);
pthread_mutex_unlock(&mutex);
```

Thread 2:

```
pthread_mutex_lock(&mutex);

pthread_cond_wait(&cv, &mutex);

printf(" world\n");
pthread_mutex_unlock(&mutex);
```

Recap

- Example

```
int    thread1_done = 0;

pthread_cond_t    cv;
pthread_mutex_t  mutex;
```

Thread 1:

```
printf("hello ");

pthread_mutex_lock(&mutex);
thread1_done = 1;
pthread_cond_signal(&cv);
pthread_mutex_unlock(&mutex);
```

Thread 2:

```
pthread_mutex_lock(&mutex);

while (thread1_done == 0) {
    pthread_cond_wait(&cv,
    &mutex);
}

printf(" world\n");
pthread_mutex_unlock(&mutex);
```

Agenda

- ~~Recap~~
- Classic IPC Problems
  - Dining Philosophers Problem
  - Readers Writers Problem

Dining Philosophers Problem

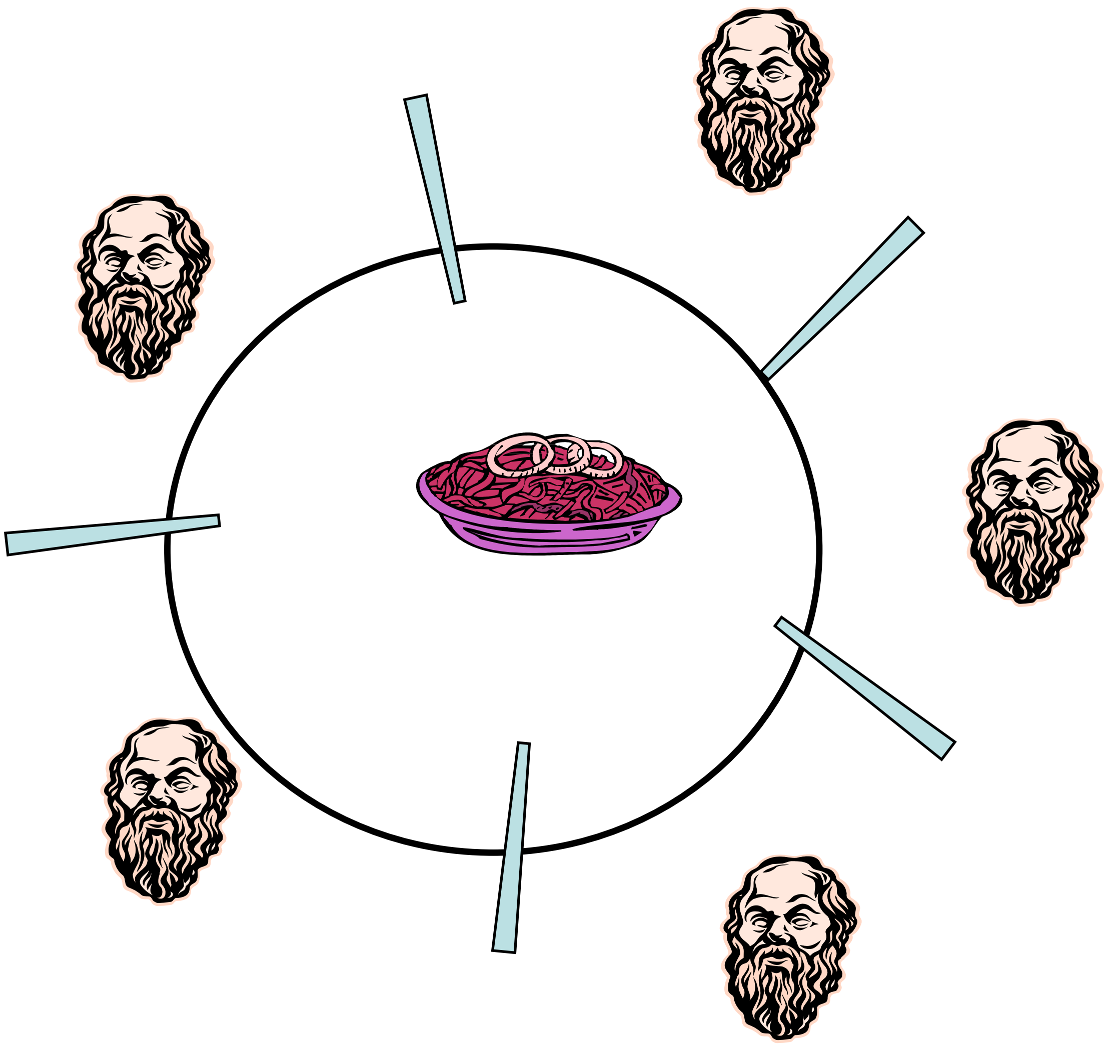
- Classic Synchronization Problem

- Philosopher
  - eat, think
  - eat, think
  - ……..

- Philosopher = Process

- Eating needs two resources (chopsticks)

Dining Philosophers Problem



Problem:
need two
chopsticks
to eat

Dining Philosophers Problem

- First Attempt

One Mutex for each chopstick

Philosopher i:

```
while (1) {
    Think();

    lock(Left_Chopstick);
    lock(Right_Chopstick);

    Eat();

    unlock(Left_Chopstick);
    unlock(Right_Chopstick);
}
```

Dining Philosophers Problem

- First Attempt

One Mutex for each chopstick

Philosopher i:

```
while (1) {
    Think();

    lock(Left_Chopstick);
    lock(Right_Chopstick);

    Eat();

    unlock(Left_Chopstick);
    unlock(Right_Chopstick);
}
```

Deadlock!

Dining Philosophers Problem

- Second Attempt

Philosopher i:

```
Think();

unsuccessful = 1;
while (unsuccessful) {
        lock(left_chopstick);
        if (0==try_lock(right_chopstick))   /* try_lock returns non-0 immediately if
                                               unable to grab the lock */
                unsuccessful = 0;
        else
                unlock(left_chopstick);
}

Eat();

unlock(left_chopstick);
unlock(right_chopstick);
```

Dining Philosophers Problem

- Second Attempt

Philosopher i:

```
Think();

unsuccessful = 1;
while (unsuccessful) {
        lock(left_chopstick);
        if (0==try_lock(right_chopstick))   /* try_lock returns non-0 immediately if
                                               unable to grab the lock */
                unsuccessful = 0;
        else
                unlock(left_chopstick);
}

Eat();

unlock(left_chopstick);
unlock(right_chopstick);
```

Starvation if unfavorable scheduling!

Dining Philosophers Problem

- In practice ....
  - Starvation will probably not occur
  - How to ensure?

Dining Philosophers Problem

- In practice ....
  - Starvation will probably not occur
  - We can ensure this by adding randomization to the system:
    - Add a random delay before retrying
    - Unlikely that our random delays will be in sync too many times

Dining Philosophers Problem

- Solution with Random Delays

```
Philosopher i:
    Think();

    while (unsuccessful) {
            wait(random());

            lock(left_chopstick);
            if (trylock(right_chopstick))
                        unsuccessful = 0;
            else
                        unlock(left_chopstick);
    }
    Eat();

    unlock(left_chopstick);
    unlock(right_chopstick);
```

Dining Philosophers Problem

- Solution without Random Delays?

Dining Philosophers Problem

- Solution without Random Delays
  - Do not try to take chopsticks one after another
    - Don’t have each chopstick protected by a different mutex
  - Try to grab both chopsticks at the same time

Dining Philosophers Problem

- Another possible solution

  - Use a mutex for the whole dinner-table (all chopsticks)

  - Philosopher i:

    lock(table);

    Eat();

    Unlock(table);

Dining Philosophers Problem

- Another possible solution
  - Use a mutex for the whole dinner-table (all chopsticks)
  - Philosopher i:

```
lock(table);
Eat();
Unlock(table);
```

Performance problem!

Agenda

- ~~Recap~~
- Classic IPC Problems
  - ~~Dining Philosophers Problem~~
  - Readers Writers Problem

Readers-Writers Problem

- Multiple threads reading/writing
  - e.g., databases
- Many threads can read simultaneously
- Only one can be writing at any time
  - When a writer is executing, nobody else can read or write

Readers-Writers Problem

- Solution Idea
  - Readers:
    - First reader locks the database
    - If a reader inside, other readers enter without locking again
    - Checking for readers occurs within a mutex
  - Writer:
    - Always lock database before entering

Readers-Writers Problem

- One solution

READER:

```
While (1) {
    down(protector);
    rc++;
    if (rc == 1)  //first reader
        down(database);
    up(protector);

    read();

    down(protector);
    rc--;
    If (rc == 0) then // last one
        up(database);
    up(protector);
    ….
}
```

WRITER:

```
While (1) {
    generate_data();
    down(database);
    write();
    up(database);
}
```

Two semaphores:

database

protector

Initial: protector=1, database =1 rc =0

Readers-Writers Problem

- Potential problem
  - Writer Starvation
    - Readers might continuously enter while a writer waits

- Other variants possible
  - Give writer priority

Agenda

- ~~Recap~~
- ~~Classic IPC Problems~~
    - ~~Dining Philosophers Problem~~
    - ~~Readers Writers Problem~~

Questions?

*acknowledgement: slides include content from “Modern Operating Systems” by A. Tanenbaum, “Operating Systems Concepts” by A. Silberschatz etc., “Operating Systems: Three Easy Pieces” by R. Arpaci-Dusseau etc., and anonymous pictures from internet.