CSC 4320/6320: Operating Systems

Chapter 06: Synchronization Tools – Contd.

Spring 2025

Instructor: Dr. Md Mahfuzur Rahman
Department of Computer Science, GSU

Disclaimer

The slides to be used in this course have been created, modified, and adapted from multiple sources:

- *The slides are copyright Silberschatz, Galvin and Gagne, 2018. The slides are authorized for personal use, and for use in conjunction with a course for which Operating System Concepts is the prescribed text. Instructors are free to modify the slides to their taste, as long as the modified slides acknowledge the source and the fact that they have been modified.*

Review: Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
  - Boolean variable indicating if lock is available or not
- Protect a critical section by
  - First **`acquire()`** a lock
  - Then **`release()`** the lock
- Calls to **`acquire()`** and **`release()`** must be **atomic**
  - Usually implemented via hardware atomic instructions such as compare-and-swap.
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

Review: Solution to CS Problem Using Mutex Locks

```
while (true) {
    acquire lock

        critical section

    release lock

remainder section
}
```

```
acquire() {
   while (!available)
     ; /* busy wait */
   available = false;
}
release() {
  available = true;
}
```

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for processes to synchronize their activities.
- Semaphore ***S*** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **`wait()`** and **`signal()`**
    - Originally called **`P()`** and **`V()`**
- Definition of the **`wait()` operation**

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

- Definition of the **`signal()` operation**

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

Semaphore (Cont.)

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can implement a counting semaphore ***S*** as a binary semaphore
- With semaphores we can solve various synchronization problems

Semaphore Usage Example

- Solution to the CS Problem
  - Create a semaphore "**`mutex`**" initialized to 1 (no. of resources)

```
wait(mutex);
   CS
signal(mutex);
```

- Consider P_1 and P_2 that with two statements S_1 and S_2 and the requirement that S_1 to happen before S_2
  - Create a semaphore "**`synch`**" initialized to 0

```
P1:
   S1;
   signal(synch);
P2:
   wait(synch);
   S2;
```

Semaphore Implementation

- Must guarantee that no two processes can execute the `wait()` and `signal()` on the same semaphore at the same time (only one process will do either **`wait()`** or **`signal()`**)
- Thus, the implementation becomes the critical section problem where the `wait` and `signal` code are placed in the critical section
- Could now have **busy waiting** in critical section implementation
  - But implementation code is short
  - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - Value (of type integer)
  - Pointer to next record in the list
- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue

Implementation with no Busy waiting (Cont.)

- Waiting queue

```
typedef struct {
int value;
struct process *list;
} semaphore;
```

Implementation with no Busy waiting (Cont.)

```
wait(semaphore *S) {
   S->value--; /* Negative values help track waiting processes*/
   if (S->value < 0) {
      add this process to S->list;
      block(); /* blocking is to suspend the process*/
   }
}

signal(semaphore *S) {
   S->value++;
   if (S->value <= 0) { // at least one process was/is waiting
      remove a process P from S->list;
      wakeup(P);
   }
}
```

Problems with Semaphores

- Incorrect use of semaphore operations:
  - `signal(mutex)` …. `wait(mutex)`
    - several processes may be executing in their critical sections simultaneously, violating the mutual-exclusion requirement.
  - `wait(mutex)` … `wait(mutex)`
    - the process will permanently block on the second call to wait(), as the semaphore is now unavailable.
  - Omitting of **`wait (mutex)`** and/or **`signal (mutex)`**
    - either mutual exclusion is violated, or the process will permanently block.

- These – and others – are examples of what can occur when semaphores and other synchronization tools are used incorrectly.

Questions

1. A counting semaphore ____.
A) is essentially an integer variable ✓
B) is accessed through only one standard operation
C) can be modified simultaneously by multiple threads
D) cannot be used to control access to a thread's critical sections

2. _____ can be used to prevent busy waiting when implementing a semaphore.
A) Spinlocks
B) Waiting queues ✓
C) Mutex lock
D) Allowing the wait() operation to succeed

3. Mutex locks and binary semaphores are essentially the same thing.
True ✓
False

CSC 4320/6320: Operating Systems

Chapter 07: Synchronization Examples

Spring 2025

Instructor: Dr. Md Mahfuzur Rahman
Department of Computer Science, GSU

Outline

- Explain the bounded-buffer synchronization problem
- Explain the readers-writers synchronization problem
- Explain the dining-philosophers synchronization problems
- Illustrate how POSIX can be used to solve process synchronization problems

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

the producer and consumer processes share the following data structures:

- ***n*** buffers, each can hold one item
- Semaphore **`mutex`** initialized to the value 1
- Semaphore **`full`** initialized to the value 0
- Semaphore **`empty`** initialized to the value n

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);
}
```

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; they do ***not*** perform any updates
  - **Writers** – can both read and write
- Problem:
  - allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are considered – all involve some form of priorities
  - first readers–writers problem, requires that no reader be kept waiting unless a writer has already obtained permission to use the shared object.
  - The second readers–writers problem requires that, once a writer is ready, that writer perform its write as soon as possible.

First Readers-Writers Problem

- Shared Data
  - Data set
  - Semaphore **`rw_mutex`** initialized to 1
  - Semaphore **`mutex`** initialized to 1
  - Integer **`read_count`** initialized to 0

rw_mutex: to control access to the readers or to the writer
mutex: to control access to the shared variable **read_count** updates
read_count: will count how many readers are reading

Readers-Writers Problem (Cont.)

- The structure of a writer process

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

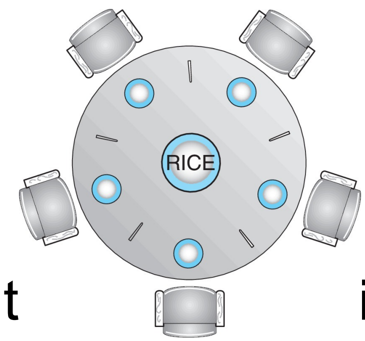
The first Readers-Writers Problem (Cont.)

- The structure of a reader process

```
while (true){
    wait(mutex); /* wait for the permission to update */
    read_count++;
    if (read_count == 1) /* first reader: make sure lock granted */
        wait(rw_mutex); /* wait for the reader's turn */
    signal(mutex);  /*release to let readers update */
        ...
     /* reading is performed */
        ...
    wait(mutex);
    read_count--;
    if (read_count == 0) /*last reader:responsible to release lock */
         signal(rw_mutex);
    signal(mutex);
}
```

Dining-Philosophers Problem

- N philosophers’ sit at a round table with a bowl of rice in the middle.



- They spend their lives alt ing and eating.
- They do not 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
- In the case of 5 philosophers, the shared data
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1 (Each chopstick is treated as a **binary semaphore**)

Dining-Philosophers Problem Algorithm

- Semaphore Solution
- The structure of Philosopher i :

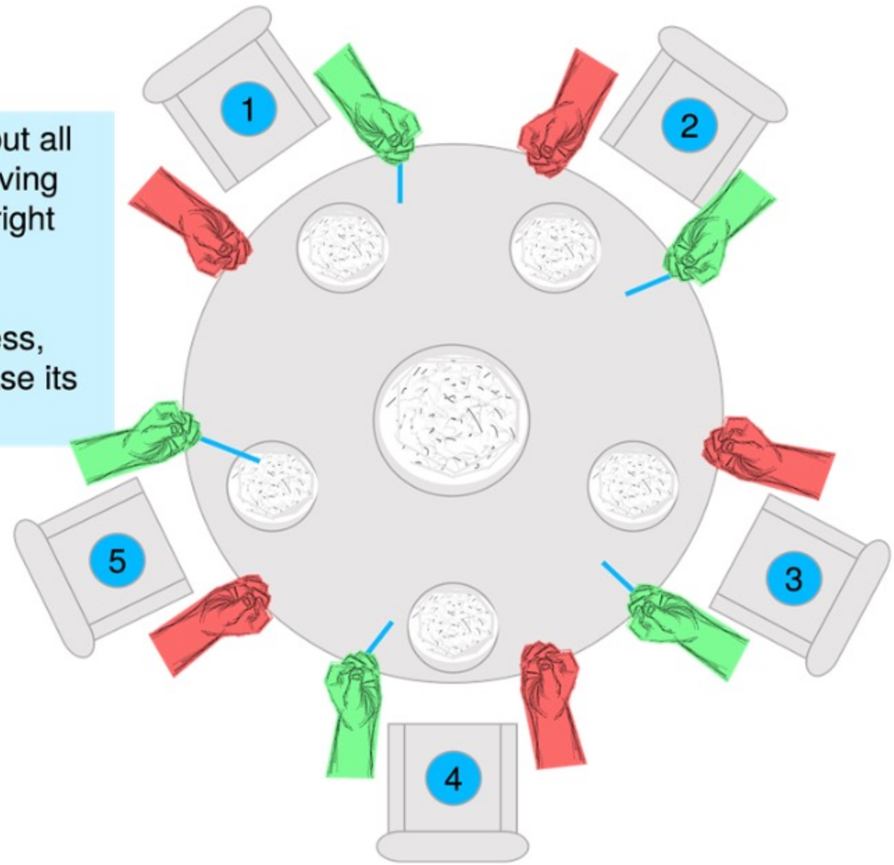
```
while (true){
    wait (chopstick[i] );
    wait (chopStick[ (i + 1) % 5] );

     /* eat for awhile */

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

     /* think for awhile */

}
```

- What is the problem with this algorithm?

The problem is deadlock

each grasps the left chopstick, but all the chopsticks are allocated, leaving none free for any philosopher's right hand

no philosopher can make progress, therefore none will eat and release its resources



Questions

1. What is the purpose of the mutex semaphore in the implementation of the bounded-buffer problem using semaphores?

A) It indicates the number of empty slots in the buffer.
B) It indicates the number of occupied slots in the buffer.
C) It controls access to the shared buffer.
D) It ensures mutual exclusion. ✓

2. The first readers-writers problem ____.

A) requires that, once a writer is ready, that writer performs its write as soon as possible.
B) is not used to test synchronization primitives.
C) requires that no reader will be kept waiting unless a writer has already obtained permission to use the shared database. ✓
D) requires that no reader will be kept waiting unless a reader has already obtained permission to use the shared database.

3. How many philosophers may eat simultaneously in the Dining Philosophers problem with 5 philosophers?

A) 1
B) 2 ✓
C) 3
D) 5

POSIX Synchronization

- POSIX API provides
  - mutex locks
  - semaphores
  - condition variable
- Widely used for thread creation and synchronization by developers 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 cannot.

POSIX Named Semaphores

- Creating an initializing the semaphore:

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

/* Create the semaphore and initialize it to 1 */
sem = sem_open("SEM", 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);
```