CS370 Operating Systems

Colorado State University

Yashwant K Malaiya

Fall 2016 Lecture 18

Slides based on

- Text by Silberschatz, Galvin, Gagne
- Various sources

FAQ

- Why are critical sections important?
  - Correctness, data corruption
- Can't critical sections cause starvation?
  - Not if they satisfy ..
- Two processes do not share any resources, do they need critical sections?
- Why we didn't study critical sections before?
- Are critical sections for two interacting processes the same length?

Critical Section

```
do {

      entry section

            critical section

      exit section

            remainder section

 } while (true);
```

Request permission to enter

Housekeeping to let processes to enter other

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
- Protect a critical section by first **`acquire()`** a lock then **`release()`** the lock
  - Boolean variable indicating if lock is available or not
- Calls to **`acquire()`** and **`release()`** must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

acquire() and release()

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

- **Usage**

```
do {
   acquire lock
      critical section
   release lock
      remainder section
} while (true);
```

acquire() and release()

- Use board

Colorado State University

Semaphores by Dijkstra

- Synchronization tool 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
  - **`wait()`** and **`signal()`**
    - Originally called **`P()`** and **`V()`** **`based on Dutch words`**
- Definition of the **`wait() operation`**

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

Waits until another process makes S=1

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

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

Binary semaphore: When s is 0 or 1, it is a mutex lock

wail() and signal()

- Use board

Semaphore Usage

- **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 solve various synchronization problems
- Consider P_1 and P_2 that require S_1 to happen before S_2

  Create a semaphore "`synch`" initialized to 0

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

- Can implement a counting semaphore ***S*** as a binary semaphore

Colorado State University

The counting semaphore

- **Controls access to a finite set of resources**
- Initialized to the number of resources
- Usage:
  - Wait (S): to use a resource
  - Signal (S): to release a resource
- When all resources are being used: S == 0
  - Block until S > 0 to use the resource

Semaphore Implementation

- Must guarantee that no two processes can execute the `wait()` and `signal()` on the same semaphore at the same time
- 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
- Alternative: block and wakeup (next slide)

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

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

Colorado State University
  ```

Implementation with no Busy waiting (Cont.)

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}

signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```

If value < 0
abs(value) is the number
of waiting processes

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

Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let *S* and *Q* be two semaphores initialized to 1

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
`...`	`...`
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

- **P0 executes wait(s), P1 executes wait(Q)**
  - P0 must wait till P1 executes signal(Q)
  - P1 must wait till P0 executes signal(S)   Deadlock!

Colorado State University

Priority Inversion

- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
- Solved via **priority-inheritance protocol**
  - Process accessing resource needed by higher priority process Inherits higher priority till it finishes resource use
  - Once done, process reverts to lower priority

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

- *n* buffers, each can hold one item
- Binary semaphore (mutex)
  - Provides mutual exclusion for accesses to buffer pool
  - Initialized to 1
- Counting semaphores
  - empty: Number of empty slots available
    - Initialized to n
  - full: Number of filled slots available n
    - Initialized to 0

Bounded-Buffer : Note

- Producer and consumer must be ready before they attempt to enter critical section
- Producer readiness?
  - When a slot is available to add produced item
    - wait(empty): empty is initialized to n
- Consumer readiness?
  - When a producer has added new item to the buffer
    - wait(full) : full initialized to 0

Bounded Buffer Problem (Cont.)

The structure of the producer process

```
do {
        ...
         /* produce an item in next_produced */
        ...
     wait(empty);            wait till slot available
     wait(mutex);        Allow producer OR consumer to (re)enter critical section
         ...
         /* add next produced to the buffer */
         ...
     signal(mutex); Allow producer OR consumer to (re)enter critical section
     signal(full);          signal consumer that a slot is available
 } while (true);
```

Colorado State University

Bounded Buffer Problem (Cont.)

The structure of the consumer process

```
Do {
    wait(full);     wait till slot available for consumption
    wait(mutex);    Only producer OR consumer can be in critical section
        ...
    /* remove an item from buffer to next_consumed */
        ...
    signal(mutex);  Allow producer OR consumer to (re)enter critical section
    signal(empty);  signal producer that a slot is available to add
        ...
    /* consume the item in next consumed */
        ...
 } while (true);
```

Colorado State University

Notes

- Midterm Friday
- Background, Processes/Threads, Scheduling, Process Synchronization (exclude Classical Problems)
- Homework due Tuesday
  - No late period, solution shared Wednesday
- PA 3 available
  - Extension of PA2: interaction using pipes and shared memory
- Help session Thursday: Midterm, PA3

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
- Shared Data
  - Data set
  - Semaphore **`rw_mutex`** initialized to 1 (mutual exclusion for writer)
  - Semaphore **`mutex`** initialized to 1 (mutual exclusion for read_count)
  - Integer **`read_count`** initialized to 0 (how many readers?)

Readers-Writers Problem (Cont.)

- The structure of a writer process

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

Readers-Writers Problem (Cont.)

- The structure of a reader process

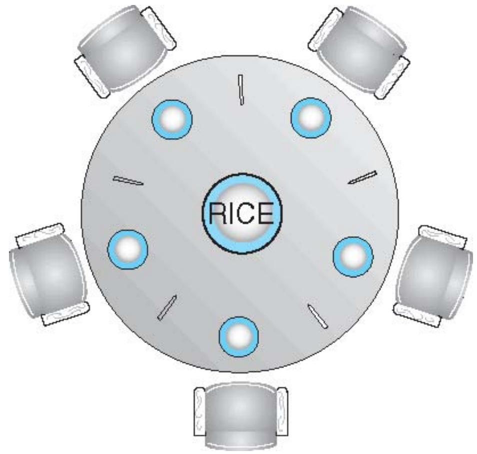
```
do {
     wait(mutex);
       read_count++;
       if (read_count == 1)
            wait(rw_mutex);
    signal(mutex);
         ...
       /* reading is performed */
         ...
    wait(mutex);
       read count--;
       if (read_count == 0)
           signal(rw_mutex);
    signal(mutex);
} while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

Colorado State University

Readers-Writers Problem Variations

- ***First*** variation – no reader kept waiting unless writer has permission to use shared object
- ***Second*** variation – once writer is ready, it performs the write ASAP
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks

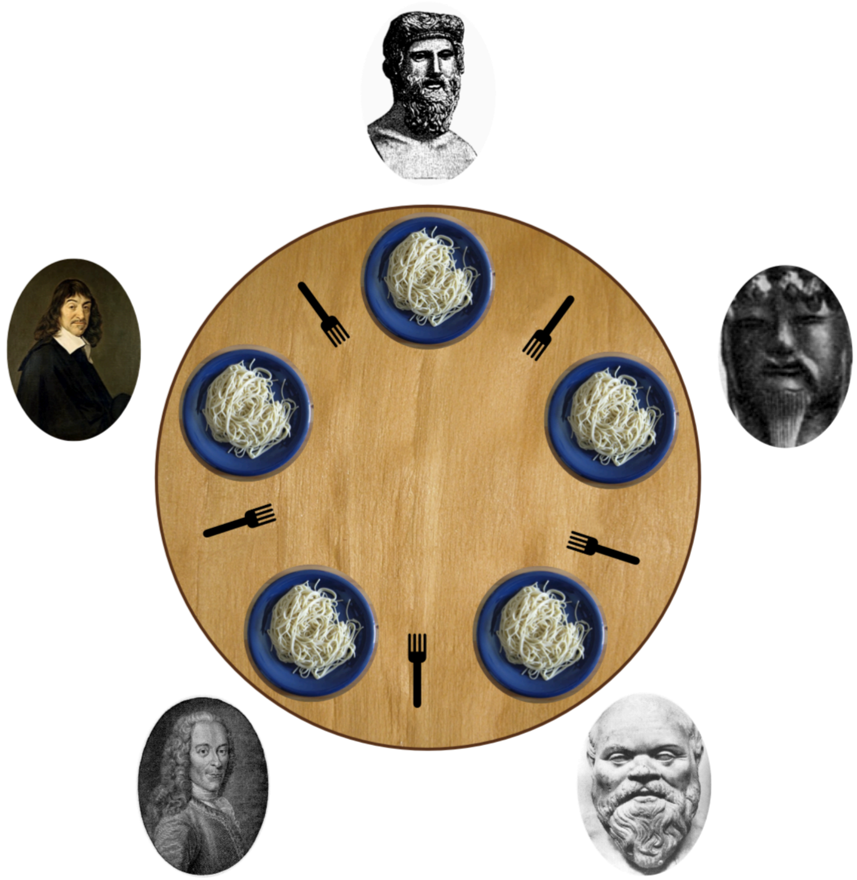
Dining-Philosophers Problem



- 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
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1

Colorado State University

Dining-Philosophers Problem



Dining-Philosophers Problem Algorithm: Simple solution?

- The structure of Philosopher *i*:

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

                  //  eat

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

                   //  think

} while (TRUE);
```

- What is the problem with this algorithm?
  - If all of them pick up the the left chopstick first - Deadlock

Colorado State University

Dining-Philosophers Problem Algorithm (Cont.)

- Deadlock handling
  - Allow at most 4 philosophers to be sitting simultaneously at the table (with the same 5 forks).
  - Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section.
  - Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.

Problems with Semaphores

- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex): what happens?
  - wait (mutex) … wait (mutex)): what happens?
  - Omitting of wait (mutex) or signal (mutex) (or both): what happens?
- Deadlock and starvation are possible.