CS370 Operating Systems
Colorado State University
Yashwant K Malaiya
Fall 2019 L11
Synchronization

Slides based on
• Text by Silberschatz, Galvin, Gagne
• Various sources
• **Round Robin scheduling: role of Ready Queue**
  – Scheduled in the order in which they are in RQ, but preempted after time q.
  – See appendix slide.

• **Why are critical sections needed?**
  – Mutual exclusion: Correctness, avoiding data inconsistency.

• **Two processes do not share any resources, do they need critical sections?**

• **How do we know what data is shared?**

• **Can’t critical sections cause starvation?**
  – Not if they satisfy ..

• **What if a process gets stuck in a critical section?**
FAQ

• How does an atomic instruction (or sequence) work?
• What does a process do during busy waiting?
Peterson’s Solution

• Good algorithmic description of solving the problem
• Two process solution only
• Assume that the load and store machine-language instructions are atomic; that is, cannot be interrupted
• The two processes share two variables:
  – int turn;
  – Boolean flag[2]
  – The variable turn indicates whose turn it is to enter the critical section
  – The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process \( P_1 \) is ready to enter!
Algorithm for Process $P_i$

```plaintext
Algorithm for Process $P_i$

```do``` 

```plaintext
{ 
flag[i] = true; 
turn = j; 
while (flag[j] && turn = = j); /*Wait*/
critical section
flag[i] = false; 
remainder section

} while (true);
```

• The variable $\text{turn}$ indicates whose turn it is to enter the critical section
• The $\text{flag}$ array is used to indicate if a process is ready to enter the critical section. $\text{flag}[i] = true$ implies that process $P_i$ is ready!
Provable that the three CS requirement are met:

1. Mutual exclusion is preserved
   \( P_i \) enters CS only if:
   - either \( \text{flag}[j] = \text{false} \) or \( \text{turn} = i \)

2. Progress requirement is satisfied

3. Bounded-waiting requirement is met. A process waits only one turn.

Detailed proof in the text.

Note: there exists a generalization of Peterson’s solution for more than 2 processes, but bounded waiting is not assured.
Many systems provide hardware support for implementing the critical section code.

All solutions below based on idea of locking
  - Protecting critical regions via locks

Modern machines provide special atomic hardware instructions
  - Atomic = non-interruptible
    - test memory word and set value
    - swap contents of two memory words
Solution 1: using test_and_set()

- Shared Boolean variable lock, initialized to FALSE
- Solution:
  ```
  do {
    while (test_and_set(&lock)) ; /* do nothing */
    /* critical section */
    ....
    lock = false;
    /* remainder section */
    ...
  } while (true);
  ```

To break out:
Return value of TestAndSet should be FALSE

Lock TRUE: locked,    Lock FALSE: not locked.
If two TestAndSet() are attempted simultaneously, they will be executed sequentially in some arbitrary order
Solution 2: Swap: Hardware implementation

```c
void Swap(boolean *a, boolean *b) {
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```
Using Swap (concurrently executed by both)

```c
do {
    key = TRUE;
    while (key == TRUE) {
        Swap(&lock, &key)
    }

    critical section

    lock = FALSE;

    remainder section
} while (TRUE);
```

Lock == false when no process is in critical section.

Cannot enter critical section UNLESS lock == FALSE by other process

Lock is a SHARED variable.
Key is a variable local to the process.

If two Swap() are executed simultaneously, they will be executed sequentially in some arbitrary order.
Swap()

Key = TRUE, swap
Key == FALSE, enter

Critical section

Locked by Process 0

Lock = FALSE

Locked by Process 1

Lock = TRUE

Key = TRUE, wait

Busy waiting

Swap ( ), Key == False

Critical section

Lock = FALSE
For process i:
do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = test_and_set(&lock);
    waiting[i] = false;
    /* critical section */
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = false;
    else
        waiting[j] = false;
    /* remainder section */
} while (true);

Shared Data structures initialized to FALSE
• boolean waiting[n]; Pr n wants to enter
• boolean lock;

The entry section for process i:
• First process to execute TestAndSet will find key == false; ENTER critical section,
• EVERYONE else must wait

The exit section for process i:
Attempts to finding a suitable waiting process j (while loop) and enable it to exit its while loop.
or if there is no suitable process, make lock FALSE.
1. **Mutual Exclusion**

2. **Progress** - If no process is executing in its critical section and there are some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.

3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.
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()

<table>
<thead>
<tr>
<th>acquire() {</th>
<th>release() {</th>
</tr>
</thead>
<tbody>
<tr>
<td>while (!available)</td>
<td>available = true;</td>
</tr>
<tr>
<td>; /* busy wait */</td>
<td>}</td>
</tr>
</tbody>
</table>

**Usage**

```c
acquire lock

critical section

release lock

remainder section

} while (true);
```
acquire() and release()
### acquire() and release()

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Lock</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>..</td>
<td>open</td>
<td>Attempt to acquire lock</td>
</tr>
<tr>
<td>..</td>
<td>closed</td>
<td>Acquires lock</td>
</tr>
<tr>
<td>Attempt to acquire lock</td>
<td>closed</td>
<td>Critical section</td>
</tr>
<tr>
<td>Attempt to acquire lock</td>
<td>closed</td>
<td>Critical section</td>
</tr>
<tr>
<td>Attempt to acquire lock</td>
<td>open</td>
<td>Release lock</td>
</tr>
<tr>
<td>Acquires lock</td>
<td>closed</td>
<td>..</td>
</tr>
<tr>
<td>Critical section</td>
<td>closed</td>
<td>..</td>
</tr>
</tbody>
</table>

**Colorado State University**

17
How are locks supported by hardware?

Atomic read-modify-write: Examples

• Atomic instruction in x86
  – LOCK instruction prefix, which applies to an instruction does a read-modify-write on memory (INC, XCHG, CMPXCHG etc)
  – Ex: lock cmpxchg <dest>, <source>

• In RISK processors?
  – Test-and-set in early MIPS
  – Instruction-pairs: Creates an atomic sequence
    • LL (Load Linked Word), SC (Store Conditional Word) instructions in MIPS
    • LDREX, STREX in ARM
• 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--;
  }
  ```
• Definition of the **signal()** operation
  
  ```
  signal(S) {
    S++;
  }
  ```
**Wait(S) and Signal (S)**

- **Process 0**
  - Wait(S)
  - Critical section
  - Signal (S)

- **Semaphore S**
  - S = 1
  - S = 0
  - S = 1
  - Locked by Process 1
  - S = 0

- **Process 1**
  - Wait (S)
  - Busy waiting
  - Gets lock, S -
  - Critical section
  - Signal (S)

- Semaphore values:
  - S = 0
  - S = 1

- Process 1 uses Semaphore S to control access to the critical section.
### acquire() and release()

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Semaphore S</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Critical section</td>
<td>0</td>
<td>wait ( ), busy waiting</td>
</tr>
<tr>
<td>Signal ( ) S++</td>
<td>1</td>
<td>Waiting, finished</td>
</tr>
<tr>
<td>..</td>
<td>0</td>
<td>S--</td>
</tr>
<tr>
<td>Wait( )</td>
<td>0</td>
<td>Critical section</td>
</tr>
<tr>
<td>Wait( )</td>
<td>0</td>
<td>Critical section</td>
</tr>
<tr>
<td>Waiting finished</td>
<td>1</td>
<td>Signal ( ) S++</td>
</tr>
<tr>
<td>S--</td>
<td>0</td>
<td>..</td>
</tr>
<tr>
<td>Critical section</td>
<td>0</td>
<td>..</td>
</tr>
</tbody>
</table>
I was hoping the distance learning service might use more up-to-date technology
Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Practically same as a **mutex lock**
- Can solve various synchronization problems
- Ex: Consider $P_1$ and $P_2$ that requires event $S_1$ to happen before $S_2$
  Create a semaphore “synch” initialized to 0 i.e not available

<table>
<thead>
<tr>
<th>P1:</th>
<th>P2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1;$</td>
<td><strong>wait(synch);</strong></td>
</tr>
<tr>
<td>signal(synch);</td>
<td>$S_2;$</td>
</tr>
</tbody>
</table>

- Can implement a counting semaphore $S$ as a binary semaphore
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;
Implementation with no Busy waiting (Cont.)

```c
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;

Colorado State University
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

  \[ P_0 \]
  
  ```
  wait(S);
  wait(Q);
  ... 
  signal(S);
  signal(Q);
  ```

  \[ P_1 \]
  
  ```
  wait(Q);
  wait(S);
  ... 
  signal(Q);
  signal(S);
  ```

  - \( P_0 \) executes `wait(s)`, \( P_1 \) executes `wait(Q)`
  - \( P_0 \) must wait till \( P_1 \) executes `signal(Q)`
  - \( P_1 \) must wait till \( P_0 \) executes `signal(S)`  Deadlock!
Priority Inversion

• **Priority Inversion** – Scheduling problem when lower-priority process $P_L$ holds a lock needed by higher-priority process $P_H$.
  
  – The low priority task may be preempted by a medium priority task $P_M$ which does not use the lock, causing $P_H$ to wait because of $P_M$.

• 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

Mars pathfinder Mission problem 1997
Classical Problems of Synchronization

• Classical problems used to test newly-proposed synchronization schemes
  – Bounded-Buffer Problem
  – Readers and Writers Problem
  – Dining-Philosophers Problem

• Monitors
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

3 semaphores needed, 1 binary, 2 counting
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
    - wait(full)
      - full initialized to 0
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);
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);
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. No readers permitted when writer is accessing the data.

• Several variations of how readers and writers are considered – all involve some form of priorities
Readers-Writers Problem

• 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?)
• The structure of a writer process

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

When: writer in critical section and if n readers waiting:
- 1 reader is queued on rw_mutex
- (n-1) readers queued on mutex
• The structure of a reader process

```c
    do {
        wait(mutex);
        read_count++;
        if (read_count == 1)
            wait(rw_mutex);
        signal(mutex);
        /* reading is performed */
        ...
    }
```

1 is queued on rw_mutex
(n-1) queued on mutex

mutex for mutual exclusion to readcount

When: writer in critical section
and if n readers waiting

Cannot read if writer is writing

signal(mutex);
wait(mutex);
read_count--;  
if (read_count == 0)
    signal(rw_mutex);
    signal(mutex);
}
while (true);

Do not hallucinate.
Readers-Writers Problem Variations

- **First** variation – no reader kept waiting unless writer has already obtained permission to use shared object
- **Second** variation – once writer is ready, it performs the write ASAP, i.e. if a writer is waiting, no new readers may start.
- 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
- Each chopstick is a semaphore
  - Grab by executing `wait()`
  - Release by executing `signal()`
- Shared data
  - Bowl of rice (data set)
  - Semaphore `chopstick [5]` initialized to 1
Dining-Philosophers Problem

Plato, Confucius, Socrates, Voltaire and Descartes
Dining-Philosophers Problem Algorithm: Simple solution?

- **The structure of Philosopher \( i \):**
  
  ```plaintext
  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
• 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.
Related classes

• Classes that follow CS370
  – CS455 Distributed Systems  Spring
  – CS457 Networks  Fall
  – CS470 Computer Architecture  Spring
  – CS475 Parallel Programming  Fall
  – CS435: Introduction to Big Data  Spring
Problems with Semaphores

- Incorrect use of semaphore operations:
  - Omitting of wait (mutex)
    - Violation of mutual exclusion
  - or signal (mutex)
    - Deadlock!
- Solution: Monitors
Shortest Job First Preemptive

Time 0: P1 arrives and starts executing.
2: P2 arrives and preempts P1. P1 in Ready Queue. RQ = {P1 rt=5}.
4: P3 arrives, preempts P2. RQ={P2 rt=2,P1 rt=5}
5: P3 done. P4 arrives rt=4. P2 rt=2 chosen & starts. RQ={P4 rt=4,P1 rt=5}
7: P2 done. P4 chosen & starts. RQ={P1 rt=5}
11: P4 done. P1 starts.
16: P1 done
Round Robin Scheduling

Quantum 2, arriving process has priority over one just preempted to get into RQ.

Time 0: P1 arrives, starts.
Time 1: P2 arrives, gets in RQ. RQ={P2}
Time 2: P2 starts. P3 arrives, gets in RQ, P1 gets in RQ.
    P1 gets in first. RQ={P1, P3}
Time 3: P2 executing. P4 arrives, gets in RQ, RQ={P4, P1, P3}
Time 4: P3 starts. P5 arrives, gets in RQ, P2 gets in RQ. RQ={P2, P5, P4, P1}
Time 5: no change
Time 6: P3 done, P1 starts. P6 arrives, gets in RQ, RQ={P6, P2, P5, P4}
Time 8: P1 done, P4 starts. RQ={P6, P2, P5}
Time 9: P4 done, P5 starts. RQ={P6, P2}
Time 11: P2 starts. RQ={P5, P6}
Time 13: P6 starts. RQ={P2, P5} ...
Round robin continues until all processes done.