CS370 Operating Systems
Colorado State University
Yashwant K Malaiya
Spring 1018 L11
Synchronization

Slides based on
• Text by Silberschatz, Galvin, Gagne
• Various sources
• Multilevel “feedback” queue: processes moved to higher or lower queue based on CPU burst time/aging.
  – Implementations vary
  – ARPACI-DUSSEAU

• Little’s formula: example of formal queuing theory. In steady state: \( n = \lambda \times W \)
  – Steady state queue length: \( n = 14 \)
  – Arrival rate \( \lambda = 7 \) per second
  – Then average waiting time \( w = 14/7 = 2 \) sec.

• Critical section in synchronization
Critical Section Problem

We saw race condition between counter ++ and counter –

Solution to the “race condition” problem: critical section

- Consider system of \( n \) processes \( \{p_0, p_1, \ldots, p_{n-1}\} \)
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- **Critical section problem** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow **critical section** with **exit section**, then **remainder section**

Race condition: when outcome depends on timing/order that is not predictable
Process Synchronization: Outline

- critical-section problem to ensure the consistency of shared data
- Software and hardware solutions of the critical-section problem
  - Peterson’s solution
  - Atomic instructions
  - Mutex locks and semaphores
- Classical process-synchronization problems
  - Bounded buffer, Readers Writers, Dining Philosophers
- Another approach: Monitors
General structure: Critical section

do {
    entry section
    critical section
    exit section
    remainder section
} while (true);
Algorithm for Process $P_i$

```c
do {
    while (turn == j);
    critical section
    turn = j;
    remainder section
} while (true);
```

Process $P_j$ is also trying to enter the critical section, and is also running similar code.
Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process \( P_i \) is executing in its critical section, then no other processes can be executing in their critical sections.

2. **Progress** - If no process is executing in its critical section and there exist 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.
   - Assume that each process executes at a nonzero speed.
   - No assumption concerning relative speed of the \( n \) processes.
Two approaches depending on if kernel is preemptive or non-preemptive

- **Preemptive** – allows preemption of process when running in kernel mode
- **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
  - Essentially free of race conditions in kernel mode
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_i is ready!
Algorithm for Process $P_i$

For process $P_j$

\[
\text{do } \\
\{ \\
\text{flag}[i] = true; \\
\text{turn} = j; \\
\text{while } (\text{flag}[j] \land \text{turn} = j); \quad /*\text{Wait}*/ \\
\text{critical section} \\
\text{flag}[i] = false; \\
\text{remainder section} \\
\} \text{ while (true);} \\
\]

- 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. $\text{flag}[i] = \text{true}$ 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

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.
Synchronization: Hardware Support

• 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 using test_and_set()

- Shared Boolean variable lock, initialized to FALSE
- Solution:
  
  ```c
  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 FALSE: not locked.
If two TestAndSet() are attempted simultaneously, they will be executed sequentially in some arbitrary order

test_and_set(&lock) returns the lock value and sets it to True.
void Swap(boolean *a, boolean *b) {
    boolean temp = *a;
    *a = *b;
    *b = temp;
}

Using Swap

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

**critical section**

**remainder section**

---

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.
For process \( i \):

\[
\begin{align*}
\text{do } & \{ \\
& \quad \text{waiting}[i] = \text{true}; \\
& \quad \text{key} = \text{true}; \\
& \quad \text{while } (\text{waiting}[i] \land \text{key}) \\
& \qquad \quad \text{key} = \text{test}\_\text{and}\_\text{set}(\&\text{lock}); \\
& \qquad \quad \text{waiting}[i] = \text{false}; \\
& \quad \quad /* \text{critical section} */ \\
& \quad \quad j = (i + 1) \mod n; \\
& \quad \quad \text{while } ((j \neq i) \land \!\!\!\text{waiting}[j]) \\
& \qquad \quad \quad j = (j + 1) \mod n; \\
& \quad \quad \text{if } (j == i) \\
& \quad \quad \quad \text{lock} = \text{false}; \\
& \quad \quad \text{else} \\
& \quad \quad \quad \text{waiting}[j] = \text{false}; \\
& \quad \quad /* \text{remainder section} */ \\
& \} \text{ while (true);} \\
\end{align*}
\]

Shared Data structures initialized to FALSE
- boolean waiting[n];
- 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 \( i \):

Part I: Finding a suitable waiting process \( j \) and
enable it to get through the while loop,
or if there is no suitable process, make lock FALSE.
The previous algorithm satisfies the three requirements

- **Mutual Exclusion**: The first process to execute TestAndSet(lock) when lock is false, will set lock to true so no other process can enter the CS.

- **Progress**: When a process exits the CS, it either sets lock to false, or waiting[j] to false (allowing j to get in), allowing the next process to proceed.

- **Bounded Waiting**: When a process exits the CS, it examines all the other processes in the waiting array in a circular order. Any process waiting for CS will have to wait at most n-1 turns.
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

do {  
   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><strong>Critical section</strong></td>
<td>closed</td>
<td>..</td>
</tr>
</tbody>
</table>
How are locks supported by hardware?

• Atomic read-modify-write
  • Atomic instruction in x86
  
    – LOCK instruction prefix, which applies to an instruction does a read-modify-write on memory (INC, XCHG, CMPXCHG etc)
  
• In RISK processors? Instruction-pairs

  – 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
  ```c
  wait(S) {
    while (S <= 0) // busy wait
      S--;
  }
  ```
• Definition of the `signal()` operation
  ```c
  signal(S) {
    S++;
  }
  ```

Waits until another process makes $S=1$

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

Process 1
- Wait (S)
- Busy waiting
- Gets lock, S -
- Critical section
- Signal (S)
### 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>
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:**
  
  ```
  S_1;
  signal(synch);
  ```

  **P2:**
  
  ```
  wait(synch);
  S_2;
  ```
- 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) {
        // If value < 0
        // abs(value) is the number of waiting processes
        add this process to S->list;
        block();
    }
}
```

```c
signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        // If value <= 0
        // abs(value) is the number of waiting processes
        remove a process P from S->list;
        wakeup(P);
    }
}
```

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

  $P_0$
  
  ```
  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 \( P_M \), 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
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
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
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

• 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

• 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 (Cont.)

• 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 */
    ...
    wait(mutex);
    read_count--;
    if (read_count == 0)
      signal(rw_mutex);
    signal(mutex);
  } while (true);
  ```

  mutex for mutual exclusion to readcount
  
  Cannot read if writer is writing
  
  When: writer in critical section and if n readers waiting 1 is queued on rw_mutex (n-1) queued on mutex
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