CS370 Operating Systems
Colorado State University
Yashwant K Malaiya
Fall 2021 L11
Synchronization

Slides based on
• Text by Silberschatz, Galvin, Gagne
• Various sources
FAQ

• What are the shared “resources”? Memory, shared variables, ..

• Two processes do not share any resources, do they need critical sections? No

• What does a process do in a critical section? Access a shared resource.

• It is unlikely that two processes will try to access a resources at the same time. Do they need a critical section? Probably not.

• I want to know more about queuing theory. Videos and on-line books.
• Peterson’s solution
  – Two processes, i and j, may want to enter their critical sections around the same time.
  – Why does Pi do this:
    \[\text{turn} = \text{j};\]
  – You can go ahead if you want to (if not, I will go ahead)
    \[\text{while (flag[j] && turn == j);} /*\text{Wait}*/\]

• Synchronization examples:
  – remember multiple processes are \textit{interacting}, even though code for just one is usually given.
Synchronization: Hardware Support

• Most modern processors provide hardware support (ISA) for implementing the critical section code. FAQ

• 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
    – others
Solution 1: 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 TRUE: locked, 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 then sets it to True.
Using Swap (concurrently executed by both)

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

    critical section

    lock = FALSE;

    remainder section
} while (TRUE);

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

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

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

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

Key = TRUE
Swap ( )
Key ==FALSE, enter

Critical section
Lock = FALSE

Locked by Process 0

Lock = TRUE
Locked by Process 1

Note: I created this to visualize the mechanism. It is not in the book. - Yashwant
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,
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 \(i\) exits the CS, it either sets lock to false, or \(\text{waiting}[i]\) 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()

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

• Usage
  do {                         
    acquire lock               
    critical section           
    release lock               
    remainder section          
  } while (true);
acquire() and release()
How are locks supported by hardware?

• Atomic read-modify-write

• Atomic instructions 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? Instruction-pairs
  – LL (Load Linked Word), SC (Store Conditional Word) instructions in MIPS
  – LDREX, STREX in ARM
  – Creates an atomic sequence
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--;
  }
  ``
- 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
    - Locked by Process 1
  - S = 0
  - Gets lock, S -

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

- S = 1
  - Locked by Process 1
  - Gets lock, S -
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>
</table>
| $S_1$;  
signal(synch); | $\text{wait(synch)}$;  
$S_2$; |

- 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) {
        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** – 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

\[
P_0 \\
\text{wait}(S); \\
\text{wait}(Q); \\
\ldots \\
\text{signal}(S); \\
\text{signal}(Q);
\]

\[
P_1 \\
\text{wait}(Q); \\
\text{wait}(S); \\
\ldots \\
\text{signal}(Q); \\
\text{signal}(S);
\]

– 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!
• **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
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 buffer
    • wait(full)
      – full initialized to 0

empty: Number of empty slots available
wait(empty) wait until at least 1 empty

full: Number of filled slots available
wait(full) wait until at least 1 full
The structure of the producer process

```c
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);
```
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);
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 \texttt{rw\_mutex} initialized to 1 (mutual exclusion for writer)
  – Semaphore \texttt{mutex} initialized to 1 (mutual exclusion for \texttt{read\_count})
  – Integer \texttt{read\_count} initialized to 0 (how many readers?)
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
Readers-Writers Problem (Cont.)

- 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 read_count
- 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
- First reader needs to wait for the writer to finish. If other readers are already reading, a new reader Process just goes in.
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
• 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
• 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