Process Synchronization: Objectives

- Concept of process synchronization.
- The critical-section problem, whose solutions can be used to ensure the consistency of shared data.
- Software and hardware solutions of the critical-section problem.
- Classical process-synchronization problems.
- Tools that are used to solve process synchronization problems.

Slides based on:
- Text by Silberschatz, Galvin, Gagne
- Berkeley Operating Systems group
- S. Pallikara
- Other sources

Yashwant K Malaiya
CS370 Operating Systems
Fall 2015
Process Synchronization
Questions from Last Time

• Thread scheduling vs process scheduling: same or different?
  – PCB vs TCB, process-contention scope vs. system-contention scope (SCS)
• Round Robin, time quantum?
• Round robin for multi-level queue - why?
• Multi-level: what after low priority processes are done?
• Do Multicore share common cache?
• Starvation and aging
• Midterm (Oct 12 M): what will it contain?
Processes can execute concurrently
- May be interrupted at any time, partially completing execution

Concurrent access to shared data may result in data inconsistency

Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

**Illustration:**
Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having an integer counter that keeps track of the number of full buffers. Initially, counter is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.
Consumer-producer problem

**Producer**

```java
while (true) {
    /* produce an item*/
    while (counter == BUFFER_SIZE) ;
    /* do nothing */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;  
}
```

**Consumer**

```java
while (true) {
    /* do nothing */
    while (counter == 0);
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZ
    counter--;  
    /* consume the item in next consumed */
}
```

They run “concurrently” (or in parallel), and are subject to context switches at unpredictable times.
Race Condition

counter++ could be compiled as
counter-- could be compiled as

register1 = counter
register1 = register1 + 1
counter = register1

register2 = counter
register2 = register2 - 1
counter = register2

They run concurrently, and are subject to context switches at unpredictable times.

Consider this execution interleaving with “count = 5” initially:
S0: producer execute register1 = counter {register1 = 5}
S1: producer execute register1 = register1 + 1 {register1 = 6}
S2: consumer execute register2 = counter {register2 = 5}
S3: consumer execute register2 = register2 - 1 {register2 = 4}
S4: producer execute counter = register1 {counter = 6}
S5: consumer execute counter = register2 {counter = 4}

Overwrites!
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
Critical Section

- General structure of process $P_i$

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

do {
    while (turn == j);
    critical section
    turn = j;
    remainder section
} while (true);
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 $P_i$

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

- 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!
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.
Many systems provide hardware support for implementing the critical section code.

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

Uniprocessors – could disable interrupts
- Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable

Modern machines provide special atomic hardware instructions
- Atomic = non-interruptible
  - Either test memory word and set value
  - Or swap contents of two memory words
Solution to Critical-section Problem Using Locks

do {
    acquire lock
    critical section
    release lock
    remainder section
} while (TRUE);
test_and_set Instruction

Definition:

```c
boolean test_and_set (boolean *target) {
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
```

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to “TRUE”.

Colorado State University
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);
```
Definition:

```c
int compare_and_swap(int *value, int expected, int new_value)
{
    int temp = *value;

    if (*value == expected)
        *value = new_value;
    return temp;
}
```

1. Executed atomically
2. Returns the original value of passed parameter “value”
3. Set the variable “value” the value of the passed parameter “new_value” but only if “value” == “expected”. That is, the swap takes place only under this condition.
Solution using compare_and_swap

- Shared integer “lock” initialized to 0;
- Solution:

```c
    do {
        while (compare_and_swap(&lock, 0, 1) != 0)  
            ; /* do nothing */
        /* critical section */
        lock = 0;
        /* remainder section */
    } while (true);
```

- Does not guarantee bounded waiting.
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);
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 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**

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

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

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);
    }
}
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

  $P_0$

  \[
  \begin{align*}
  &\text{wait}(S); \\
  &\text{wait}(Q); \\
  &\ldots \\
  &\text{signal}(S); \\
  &\text{signal}(Q);
  \end{align*}
  \]

  $P_1$

  \[
  \begin{align*}
  &\text{wait}(Q); \\
  &\text{wait}(S); \\
  &\ldots \\
  &\text{signal}(Q); \\
  &\text{signal}(S);
  \end{align*}
  \]

- **Starvation** – indefinite blocking
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via priority-inheritance protocol
• 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
- Semaphore \texttt{mutex} initialized to the value 1
- Semaphore \texttt{full} initialized to the value 0
- Semaphore \texttt{empty} initialized to the value \( n \)
Bounded Buffer Problem (Cont.)

- The structure of the producer process

```c
    do {
        ...
        /* produce an item in next_produced */
        ...
        wait(empty);
        wait(mutex);
        ...
        /* add next produced to the buffer */
        ...
        signal(mutex);
        signal(full);
    } while (true);
```
The structure of the consumer process

Do {
    wait(full);
    wait(mutex);
    ...
    /* remove an item from buffer to next_consumed */
    ...
    signal(mutex);
    signal(empty);
    ...
    /* 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

• 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?)
• The structure of a writer process

```c
    do {
        wait(rw_mutex);
        ...
        /* writing is performed */
        ...
        signal(rw_mutex);
    } while (true);
```
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);
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
Dining-Philosophers Problem
• 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.
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.
Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- But not powerful enough to model some synchronization schemes

```plaintext
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (...) { .... }

    procedure Pn (...) {......}

    Initialization code (...) { ... }
}
```
Schematic view of a Monitor

- Provides an easy way to achieve mutual exclusion
- But ... we also need a way for processes to block when they cannot proceed
The **condition** construct

- `condition x, y;`

- Two operations are allowed on a condition variable:
  - `x.wait()` — a process that invokes the operation is suspended until `x.signal()`
  - `x.signal()` — resumes one of processes (if any) that invoked `x.wait()`
    - If no `x.wait()` on the variable, then it has no effect on the variable

Compare with semaphore
Monitor with Condition Variables

queues associated with $x, y$ conditions

shared data

operations

initialization code

entry queue
Condition Variables Choices

• If process P invokes `x.signal()`, and process Q is suspended in `x.wait()`, what should happen next?
  – Both Q and P cannot execute in parallel. If Q is resumed, then P must wait

• Options include
  – **Signal and wait** – P waits until Q either leaves the monitor or it waits for another condition
  – **Signal and continue** – Q waits until P either leaves the monitor or it waits for another condition
  – Both have pros and cons – language implementer can decide
  – Monitors implemented in *Concurrent Pascal (‘75)*
    • P executing signal immediately leaves the monitor, Q is resumed
  – Implemented in other languages including C#, Java
Difference between the signal() in semaphores and monitors

• Monitors {condition variables}: Not persistent
  – If a signal is performed and no waiting threads?
    • Signal is simply ignored
  – During subsequent wait operations
    • Thread blocks

• Semaphores
  – Signal increments semaphore value even if there are no waiting threads
  – Future wait operations would immediately succeed!
Monitor Solution to Dining Philosophers: Deadlock-free

```java
enum {THINKING,HUNGRY,EATING} state[5];

• state[i] = EATING only if
  – state[(i+4)%5] != EATING && state[(i+1)%5] != EATING !

• condition self[5]
  – Delay self when HUNGRY but unable to get chopsticks

Sequence of actions

• Before eating, must invoke pickup()
  – May result in suspension of philosopher process
  – After completion of operation, philosopher may eat

DiningPhilosophers.pickup(i);
...
eat
...
DiningPhilosophers.putdown(i);
```
monitor DiningPhilosophers
{
    enum { THINKING, HUNGRY, EATING) state [5];
    condition self [5];

    void pickup (int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING) self[i].wait;
    }

    void putdown (int i) {
        state[i] = THINKING;
        // test left and right neighbors
        test(((i + 4) % 5);
        test(((i + 1) % 5);
    }
}

Suspend self if unable to acquire chopstick

Check to see if person on left or right can use the chopstick
void test (int i) {
    if (((state[(i + 4) % 5] != EATING) &&
    (state[i] == HUNGRY) &&
    (state[(i + 1) % 5] != EATING) ) {
        state[i] = EATING ;
        self[i].signal () ;
    }
}

initialization_code() {
    for (int i = 0; i < 5; i++)
        state[i] = THINKING;
}
Possibility of starvation

• Philosopher i can starve if eating periods of philosophers on left and right overlap
• Possible solution
  – Introduce new state: STARVING
  – Chopsticks can be picked up if no neighbor is starving
    • Effectively wait for neighbor’s neighbor to stop eating
    • REDUCES concurrency!
Monitor Implementation Using Semaphores

- Variables

```c
semaphore mutex; // (initially = 1) allows only one process to be active
semaphore next;  // (initially = 0) causes signaler to sleep
int next_count = 0;  num of sleepers since they signalled
```

- Each procedure $F$ will be replaced by the compiler by

```c
wait(mutex);
...
body of F;
...
if (next_count > 0)
    signal(next)
else
    signal(mutex);
```

- Mutual exclusion within a monitor is ensured
Monitor Implementation – Condition Variables

• For each condition variable \( x \), we have:

\[
\text{semaphore } x\textunderscore \text{sem}; \quad // \quad (\text{initially } = 0) \text{ causes caller of wait to sleep}
\]
\[
\text{int } x\textunderscore\text{count} = 0; \quad // \quad \text{number of sleepers on condition}
\]

• The operations \textbf{\textit{x.wait}} and \textbf{\textit{x.signal}} can be implemented as:

<table>
<thead>
<tr>
<th>The operation x.wait can be implemented as:</th>
<th>The operation x.signal can be implemented as:</th>
</tr>
</thead>
</table>
| \begin{align*}
\text{x\_count} & \text{++} \\
\text{if (next\_count} & \text{> 0)} \\
& \quad \text{signal(next)} \\
\text{else} \\
& \quad \text{signal(mutex)} \\
\text{wait(x\_sem)} \\
\text{x\_count} & \text{--} \\
\end{align*} | \begin{align*}
\text{if (x\_count} & \text{> 0)} \\
& \quad \{ \\
& \quad \quad \text{next\_count} & \text{++} \\
& \quad \quad \text{signal(x\_sem)} \\
& \quad \quad \text{wait(next)} \\
& \quad \quad \text{next\_count} & \text{--} \\
& \quad \} \\
\end{align*} |
Resuming Processes within a Monitor

• If several processes queued on condition x, and x.signal() is executed, which should be resumed?
• FCFS frequently not adequate
• **conditional-wait** construct of the form x.wait(c)
  – Where c is **priority number**
  – Process with lowest number (highest priority) is scheduled next
Single Resource allocation

- Allocate a single resource among competing processes using priority numbers that specify the maximum time a process plans to use the resource

  ```
  R.acquire(t);
  ...
  access the resource;
  ...
  R.release;
  ```

- Where R is an instance of type `ResourceAllocator`
monitor ResourceAllocator
{
    boolean busy;
    condition x;

    void acquire(int time) {
        if (busy)
            x.wait(time);
        busy = TRUE;
    }

    void release() {
        busy = FALSE;
        x.signal();
    }

    initialization code() {
        busy = FALSE;
    }
}

Synchronization Examples

• Solaris
• Windows
• Linux
• Pthreads
Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses **adaptive mutexes** for efficiency when protecting data from short code segments
  - Starts as a standard semaphore spin-lock
  - If lock held, and by a thread running on another CPU, spins
  - If lock held by non-run-state thread, block and sleep waiting for signal of lock being released
- Uses **condition variables**
- Uses **readers-writers** locks when longer sections of code need access to data
- Uses **turnstile**s to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock
  - Turnstiles are per-lock-holding-thread, not per-object
- Priority-inheritance per-turnstile gives the running thread the highest of the priorities of the threads in its turnstile
Windows Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses **spinlocks** on multiprocessor systems
  - Spinlocking-thread will never be preempted
- Also provides **dispatcher objects** user-land which may act mutexes, semaphores, events, and timers
  - **Events**
    - An event acts much like a condition variable
  - Timers notify one or more thread when time expired
  - Dispatcher objects either **signaled-state** (object available) or **non-signaled state** (thread will block)
Linux Synchronization

• Linux:
  – Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  – Version 2.6 and later, fully preemptive

• Linux provides:
  – Semaphores
  – atomic integers
  – spinlocks
  – reader-writer versions of both

• On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption
• Pthreads API is OS-independent

• It provides:
  – mutex locks
  – condition variable

• Non-portable extensions include:
  – read-write locks
  – spinlocks
Alternative Approaches

• Transactional Memory

• OpenMP

• Functional Programming Languages
• A **memory transaction** is a sequence of read-write operations to memory that are performed atomically.

```c
void update()
{
    /* read/write memory */
}
```
OpenMP

- OpenMP is a set of compiler directives and API that support parallel programming.

```c
void update(int value)
{
    #pragma omp critical
    {
        count += value
    }
}
```

The code contained within the `#pragma omp critical` directive is treated as a critical section and performed atomically.