Chapter 5: Process Synchronization
Chapter 5: Process Synchronization

- Background
- The Critical-Section Problem
- Peterson’s Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples
- Alternative Approaches
Objectives

- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data

- To present both software and hardware solutions of the critical-section problem

- To examine several classical process-synchronization problems

- To explore several tools that are used to solve process synchronization problems
Background

- 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 of the problem:
  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.
while (true) {
    /* produce an item in next produced */

    while (counter == BUFFER SIZE) ;
    /* do nothing */
    buffer[in] = next produced;
    in = (in + 1) % BUFFER SIZE;
    counter++;
}

Consumer

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

Race Condition

- `counter++` could be implemented as
  
  ```
  register1 = counter
  register1 = register1 + 1
  counter = register1
  ```

- `counter--` could be implemented as
  
  ```
  register2 = counter
  register2 = register2 - 1
  counter = register2
  ```

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

- Consider system of \( n \) processes \( \{ P_0, P_1, ..., P_{n-1} \} \)

- Each process has a critical section segment of code
  - Process may be changing common variables, updating table, writing file, etc.
  - When one process is in its critical section, no other may be executing in its critical section

- Critical-section problem is to design a protocol to solve this

- Each process must ask permission to enter its critical section in entry section, may follow critical section with exit section, the remaining code is in its remainder section
Critical Section

- General structure of process $P_i$ is

\[
\text{do } \{
\begin{array}{c}
\text{entry section} \\
\text{critical section} \\
\text{exit section} \\
\text{remainder section}
\end{array}
\} \text{ while (true);} 
\]
### 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 only the processes not in their remainder section can participate in the selection of the process that will enter its critical section next; the selection 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.
Solution to Critical-Section Problem

Two approaches for handling critical sections in OSes, depending on if kernel is preemptive or non-preemptive

- **Preemptive** – allows preemption of process when running in kernel mode
  - Specially difficult in multiprocessor architectures, but it makes the system more responsive
- **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
  - Essentially free of race conditions on kernel data structures in kernel mode, since only one active process in the kernel at a time
Peterson’s Solution

- Good algorithmic description of solving the problem (but no guarantees for modern architectures)
- Solution restricted to two processes in alternate execution (critical section and remainder section)
- Assume that the `load` and `store` instructions are atomic; i.e., cannot be interrupted

- The two processes share two variables:
  - `int turn`
  - `boolean flag[2]`

- The variable `turn` indicates whose turn it is to enter its critical section

- The `flag` array is used to indicate if a process is ready to enter its critical section
  - `flag[i] == true` implies that process $P_i$ is ready to enter its critical section
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);
        critical section
    flag[i] = false;
        remainder section
} while (true);

■ Provable that
1. Mutual exclusion is preserved
2. Progress requirement is satisfied
3. Bounded-waiting requirement is met
Synchronization Hardware

- Many systems provide hardware support for 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
do {
    acquire lock
    critical section
    release lock
    remainder section
} while (TRUE);
Solution using test_and_set()

- Two `test_and_set()` cannot be executed simultaneously
- Shared boolean variable `lock`, initialized to FALSE
- Solution:

```c
boolean test_and_set(boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
do {
    while (test_and_set(&lock))
        ; /* do nothing */
    /* critical section */
    lock = false;
    /* remainder section */
} while (true);
```
compare_and_swap Instruction

- Definition:

```c
int compare_and_swap(int *value, int expected, int new_value) {
    int temp = *value;
    if (*value == expected)
        *value = new_value;
    return temp;
}
```
Solution using compare_and_swap

- Shared boolean variable `lock` initialized to FALSE
- Each process has a local boolean variable `key`
- Solution:

```c
do {
    while (compare_and_swap(&lock, 0, 1) != 0)  //value,expected,new
        ; /* do nothing */
    /* critical section */
    lock = 0;
    /* remainder section */
} while (true);
```
Bounded-waiting Mutual Exclusion with test_and_set

do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = test_and_set(&lock);  //only first lock==false will set key=false
    waiting[i] = false;
    /* critical section */
    j = (i + 1) % n;    //look for the next P[j] waiting: bound-waiting req.
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = false;
    else
        waiting[j] = false; //wakeup only one process P[j] without releasing lock
    /* remainder section */
} while (true);
Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest synchronization is done with mutex lock
- Protect critical regions by first **acquire**() a lock (then all other processes attempting to get the lock are blocked), and then **release**() it
  - Boolean variable indicating if lock is available or not

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

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

- Calls to `acquire()` and `release()` must be atomic
  - Usually implemented via hardware atomic instructions

- But this solution requires **busy waiting**
  - All other processes trying to get the lock must continuously loop
  - This lock is therefore called a **spinlock**
  - Very wasteful of CPU cycles
  - Might still be more efficient than (costly) context switches for shorter wait times
Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore $S$ – integer variable
- Two standard operations modify $S$: `wait()` and `signal()`
- Less complicated
- Can only be accessed via two indivisible (atomic) operations

```c
wait (S) {
    while (S <= 0)
        ; // busy wait
    S--;
}
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
  - Then equivalent to a mutex lock
- Can solve various synchronization problems
- Consider \( P_1 \) and \( P_2 \) that require \( S_1 \) to happen before \( S_2 \)

\[\begin{align*}
P_1: & \\
& S_1; \\
& \text{signal(synch); } // \text{sync++ has added one resource} \\

P_2: & \\
& \text{wait(synch); } // \text{executed only when synch is } > 0 \\
& S_2;
\end{align*}\]
Semaphore Implementation

- Must guarantee that no two processes can execute `wait()` and `signal()` on the same semaphore at the same time.

- Thus, implementation becomes the critical section problem, where the `wait` and `signal` codes 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 in the appropriate waiting queue
  - wakeup – remove one of the processes in the waiting queue and place it in the ready queue

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

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

void 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

  \[
  \begin{align*}
  P_0 & \quad P_1 \\
  & \text{exec 1st} & \text{exec 2nd} \\
  \text{wait}(S) & \text{wait}(Q) \\
  \text{wait}(Q) & \text{wait}(S) \\
  \ldots & \ldots \\
  \text{signal}(S) & \text{signal}(Q) \\
  \text{signal}(Q) & \text{signal}(S)
  \end{align*}
  \]

- **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

- Scenario:
  - Processes with priorities $L < M < H$ request resource $r$
  - $L$ first locks on $r$; then $H$ requests $r$, but must wait until $L$ ends
  - In the meantime, $M$ requests $r$ and preempts $L$
  - $H$ is waiting longer for lower-priority processes

- Not a problem with only two levels of priorities, but two levels are insufficient for most OSes

- Solved via **priority-inheritance protocol**
  - All lower-priority processes with a resource requested by a higher-priority process, inherits the higher-priority level until it releases the resource
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

- Shared data structures:
  - $n$ buffers, each can hold one item
  - Semaphore `mutex` initialized to the value 1
  - Semaphore `full` initialized to the value 0
  - Semaphore `empty` initialized to the value $n$

- 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
  
  ```c
  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 a single writer can access the shared data at the same time

- Several variations of how readers and writers are treated – all involve priorities

- Variations to the problem:
  - First variation – no reader kept waiting, unless writer has permission to use shared object
  - Second variation – once writer is ready, it performs a write as soon as possible
  - Both may have starvation, leading to even more variations
    - first: readers keep coming in while writers are never treated
    - second: writers keep coming in while readers are never treated
  - Problem is solved on some systems by kernel providing reader-writer locks
Readers-Writers Problem

- Shared data
  - Data set
  - Semaphore `rw_mutex` initialized to 1
  - Semaphore `mutex` initialized to 1
  - Integer `read_count` initialized to 0

- The structure of a writer process
  ```
  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);
  ```
Dining-Philosophers Problem

- Philosophers spend their lives thinking and eating
- They do not interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time, on either side) to eat from bowl
  - Need both chopsticks to eat
  - Put back in their place both chopsticks when done eating
- Shared data for 5 philosophers
  - Bowl of rice (data set)
  - Semaphore `chopstick[5]` initialized to 1

The structure of Philosopher $i$:
```c
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?
Problems with Semaphores

- Incorrect use of semaphore operations can be difficult to detect, e.g.:
  - Inversed order: \texttt{signal(mutex)} ... \texttt{wait(mutex)}
  - Repeated calls: \texttt{wait(mutex)} ... \texttt{wait(mutex)}
  - Omitted calls: \texttt{wait(mutex)} or \texttt{signal(mutex)} (or both)

- Results in deadlock and/or starvation
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

```c
monitor monitor-name
{
    // shared variable declarations
    function P1(...) { ... }
    ... 
    function Pn(...) { ... }

    initialization_code (...) { ... }
}
```
Condition Variables

- Synchronization mechanism for monitors declared as variables:
  - condition x, y;

- Only two operations can be invoked 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 condition variable, then it has no effect on the variable
Condition Variables Choices

- If process P invokes \( x\.signal() \), with Q in \( x\.wait() \) state, what should happen next?
  - If Q is resumed, then P must wait, otherwise two processes would be active in the monitor

- Options include
  - **Signal and wait** – P waits until Q leaves monitor or waits for another condition
  - **Signal and continue** – Q waits until P leaves the monitor or waits for another condition

- Both have pros and cons – language implementer can decide
- Monitors implemented in Concurrent Pascal compromise
  - P executing signal immediately leaves the monitor, Q is resumed
- Implemented in other languages including Mesa, C#, Java
Solution to Dining Philosophers

```c
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);
    }

    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;
    }
}
```
Each philosopher \( i \) invokes the operations `pickup()` and `putdown()` in the following sequence:

```c
DiningPhilosophers.pickup(i);
...
EAT
...
DiningPhilosophers.putdown(i);
```

- No deadlocks, but starvation is possible
Monitor Implementation Using Semaphores

- Variables

```c
semaphore mutex;  // (initially  = 1)
semaphore next;   // (initially  = 0)
int next_count = 0;
```

- Each external function $F$ will be replaced 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:
  
  \[
  \begin{align*}
  \text{semaphore } x\text{\_sem;} & \quad // (\text{initially }= 0) \\
  \text{int } x\text{\_count }= 0;
  \end{align*}
  \]

- The operation \( x\text{.wait()} \) can be implemented as:

  \[
  \begin{align*}
  \text{x\_count++;} \\
  \text{if (next\_count } > 0) \\
  \quad \text{signal(next);} \\
  \text{else} \\
  \quad \text{signal(mutex);} \\
  \text{wait(x\_sem);} \\
  \text{x\_count--;}
  \end{align*}
  \]

- The operation \( x\text{.signal()} \) can be implemented as:

  \[
  \begin{align*}
  \text{if (x\_count } > 0) \{ \\
  \quad \text{next\_count++;} \\
  \quad \text{signal(x\_sem);} \\
  \quad \text{wait(next);} \\
  \quad \text{next\_count--;}
  \}
  \end{align*}
  \]
Resuming Processes within a Monitor

If several processes queued on condition $x$, and $x$.signal() executed, which one should be resumed?

First-come, first-served (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
- Example of ResourceAllocator, next
A Monitor to Allocate Single Resource

```java
monitor ResourceAllocator
{
    boolean busy;
    condition x;

    void acquire(int time) {
        if (busy)
            x.wait(time); //time: maximum time it will keep resource
        busy = TRUE;
    }

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

    initialization_code() {
        busy = FALSE;
    }
}
```
Synchronization Examples

- Windows XP
- Solaris
- Linux
- Pthreads
Windows XP 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 (outside the kernel) which may act as mutex locks, semaphores, events, and timers
  
  - Event acts much like a condition variable (notify thread(s) when condition occurs)
  - Timer notifies one or more threads when specified amount of time has expired
  - Dispatcher objects either signaled-state (object available) or non-signaled state (thread will block)
Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing

- Uses adaptive mutex locks for efficiency when protecting data from short code segments
  - Starts as a standard semaphore spin-lock
  - If lock is held, and by a thread running on another CPU, spins
  - If lock is 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 turnstiles 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
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
  - spinlocks
  - reader-writer versions of both

- On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption
Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variables
- Non-portable extensions include:
  - read-write locks
  - spinlocks
Alternatives to Thread-safe Concurrent Applications

- Transactional memory
- OpenMP
- Functional programming languages
Transactional Memory

- Assures that operations happen as a single logical unit of work, in its entirety, or not at all
- Related to the field of database systems
- Challenge is assuring atomicity, despite computer system failures
- **Transaction** - collection of instructions or operations that performs single logical function
  - Here we are concerned with changes to stable storage – disk
  - Transaction is series of *read* and *write* operations
  - Terminated by *commit* (transaction successful) or *abort* (transaction failed) operation
  - Aborted transaction must be rolled back to undo any changes it performed
- In software, compiler is responsible to identify the concurrent sections of the code and to augment them with proper lock mechanisms
- In hardware, implemented as cache hierarchies, and cache coherency mechanisms
- Transactional memory is not widespread yet, but…
OpenMP

- Indicates where in the code only one thread may be active at a time
  - similar to a mutex lock
- Special compiler instruction
  - `#pragma omp critical`
- Programmer must still handle race conditions and deadlocks
Functional Languages

- E.g.: Erlang, Scala
- Functional languages do not maintain state
  - Once a variable is assigned a value, it cannot change anymore
- No such problems as race conditions and deadlocks
- Therefore suitable for concurrent/parallel programming on multicores
End of Chapter 5