Outline of Lecture-09

- Problems with concurrent access to shared data
  - Race condition and critical section
  - General structure for enforce critical section
- Software based solutions:
  - Simple solution
  - Peterson's solution
- Hardware based solution
  - Disable interrupts
  - TestAndSet
  - Swap
- OS solution -- Semaphore
  - Using semaphores to solve real issues

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 introduce the concept of an atomic transaction and describe mechanisms to ensure atomicity

Shared Data

- at the same logical address space
- at different address space through messages (later in DS)

Threads Memory Model

- Conceptual model:
  - Multiple threads run in the same context of a process
  - Each thread has its own separate thread context
    - Thread ID, stack, stack pointer, PC, and GP registers
    - All threads share the remaining process context
      - Code, data, heap, and shared library segments
      - Open files and installed handlers
- Operationally, this model is not strictly enforced:
  - Register values are truly separate and protected, but...
  - Any thread can read and write the stack of any other thread
Mapping Variable Instances to Memory

<table>
<thead>
<tr>
<th>Name</th>
<th>Referenced by instance</th>
<th>Main thread?</th>
<th>Peer thread 0?</th>
<th>Peer thread 1?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td></td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>msgs</td>
<td></td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>i.m</td>
<td></td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>msgs.m</td>
<td></td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>myid.t0</td>
<td></td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>myid.t1</td>
<td></td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>cnt</td>
<td></td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Local state var: 1 instance (cnt) [data]

**Concurrent Access to Shared Data**

- Two threads A and B access a shared variable “Balance”

**Thread A:**
Balance = Balance + 100

A1. LOAD R1, BALANCE
A2. ADD R1, 100
A3. STORE BALANCE, R1

**Thread B:**
Balance = Balance - 200

B1. LOAD R1, BALANCE
B2. SUB R1, 200
B3. STORE BALANCE, R1

**Different outputs**

[0]: Hello from foo (svar-1)
[1]: Hello from bar (svar-2)
[1]: Hello from bar (svar-1)
[0]: Hello from foo (svar-2)
[0]: Hello from foo (svar-1)
[1]: Hello from bar (svar-1)
[1]: Hello from bar (svar-1)

Conclusion: different threads may execute in a random order

**What is the problem?**

- Observe: In a time-shared system, the exact instruction execution order cannot be predicted

- **Scenario 1:**
  - A1. LOAD R1, BALANCE
  - A2. ADD R1, 100
  - A3. STORE BALANCE, R1
  - Context Switch
  - B1. LOAD R1, BALANCE
  - B2. SUB R1, 200
  - B3. STORE BALANCE, R1
  - Sequential correct execution
  - Balance is effectively decreased by 100

- **Scenario 2:**
  - B1. LOAD R1, BALANCE
  - B2. SUB R1, 200
  - Context Switch
  - A1. LOAD R1, BALANCE
  - A2. ADD R1, 100
  - A3. STORE BALANCE, R1
  - Context Switch
  - B3. STORE BALANCE, R1
  - Mixed wrong execution
  - Balance is effectively decreased by 200

**What are possible results for this program?**
\[ a = 0; \quad b = 0; \quad \text{// Initial state} \]

**Thread 1**
- T1-1: if \((b == 0)\)
- T1-2: \(a = 1\)

**Thread 2**
- T2-1: if \((a == 0)\)
- T2-2: \(b = 1\)

\[
\begin{array}{c|c|c|c|c}
\text{Thread} & \text{state} & \text{99.43%} & \text{0.56%} & \text{0.01%} \\
\hline
\text{1} & a = 1 & \text{b = 0} & \\
\text{2} & a = 0 & \text{b = 1} & \\
\text{3} & a = 1 & \text{b = 1} & \\
\text{4} & a = 1 & \text{b = 1} & \\
\end{array}
\]

**Common Properties of Two Examples**
- There are some shared variables
  - Account
  - \(a, b\)
- At least one operation is a write operation
- The results of the execution are non-deterministic, depending on the execution order

**Race Conditions**
- Multiple processes/threads write/read shared data and the outcome depends on the particular order to access shared data are called race conditions.
  - A serious problem for concurrent system using shared variables!

  **How do we solve the problem??**
- Need to make sure that some high-level code sections are executed atomically
  - Atomic operation means that it completes in its entirety without worrying about interruption by any other potentially conflict-causing process

**Definition of Synchronization**
- Cooperating processes/threads share data & have effects on each other \(\rightarrow\) executions are NOT reproducible with non-deterministic exec. speed
- Concurrent executions
  - Single processor \(\rightarrow\) achieved by time slicing
  - Parallel/distributed systems \(\rightarrow\) truly simultaneous
- **Synchronization** \(\rightarrow\) getting processes/threads to work together in a coordinated manner
Critical-Section (CS) Problem

- Multiple processes/threads compete to access the shared data

- **Critical section/region**: a piece of code that accesses a shared resource (data structure or device) must not be concurrently accessed by more than one thread of execution.

- Problem – ensure that only one process/thread is allowed to execute in its critical section (for the same shared data) at any time. The execution of critical sections must be **mutually exclusive in time**.

General Structure for Critical Sections

```plaintext
do { 
    entry section
    critical section
    exit section
    remainder statements
} while (1);
```

In the *entry section*, the process requests "permission".

Requirements for CS Solutions

- **Mutual Exclusion**
  - At most one task can be in its CS at any time

- **Progress**
  - If all other tasks are in their remainder sections, a task is allowed to enter its CS
  - Only those tasks that are not in their remainder section can decide which task can enter its CS next, and this decision cannot be postponed infinitely.

- **Bounded Waiting**
  - Once a task has made a request to enter its CS, other tasks can only enter their CSs with a bounded number of times

Mutual Exclusion
Solutions for CS Problem

- Software based solution
  - Peterson’s solution

- Hardware based solutions
  - Disable interrupts
  - Atomic instructions: TestAndSet and Swap

- OS solutions:
  - Semaphore

Outline of Lecture-09

- Problems with concurrent access to shared data
  - Race condition and critical section
  - General structure for enforce critical section

- Software based solutions:
  - Simple solution
  - Peterson’s solution

- Hardware based solution
  - Disable interrupts
  - TestAndSet
  - Swap

- OS solution -- Semaphore
  - Using semaphores to solve real issues

A Simple Solution

- Bool turn; indicate whose turn to enter CS
- T0 and T1: alternate between CS and remainder

**Process 0:**

```c
while(TRUE) {
  while (turn = 0) {
    critical section
    turn = 1;
  }
  remainder section
}
```

**Process 1:**

```c
while(TRUE) {
  while (turn = 1) {
    critical section
    turn = 0;
  }
  remainder section
}
```

**Pros:**
1. P_i’s critical section is executed iff turn = i
2. Pi is busy waiting if P_j is in CS (mutual exclusion)

**Cons:**
1. Progress is not satisfied since it requires strict alternation
2. A process cannot enter the CS more often than the other

Peterson’s Solution

- Three shared variables: turn and flag[2]

**Process 0 loop:**

```c
flag[0] = i;
while (flag[0] = 1 && turn = 0)
  critical section
flag[0] = 0;
remainder section
```

**Process i loop:**

```c
flag[i] = 1;
while (flag[0] = 1 && turn = 0)
  critical section
flag[i] = 0;
remainder section
```

How does this solution satisfy the CS requirements (mutual exclusion, progress, bounded waiting)?
Peterson's Solution

**Guarantee mutual exclusion?**

Q: when process 0 is in its critical section, can process 1 get into its critical section?
→ flag[0] = 1 and either flag[1] = 0 or turn = 0
→ Thus, for process 1: under flag[0] = 1, if flag[1] = 0, P1 is in reminder section; if turn = 0, it waits before its CS

---

**Progress Guarantee?**

Q: when P0 is in its reminder section, can P1 get into its critical section?
→ flag[0] = 0
→ Process 1: if flag[0] = 0, then P1 can enter without its CS without waiting

---

**Mutual exclusion**

**Progress**

**Bounded waiting:** alternation

Any issues of Peterson's solution?

- Busy waiting
- Each thread should have different code
- Can it support more than two threads?
- Error-prone

---

Peterson's Solution (cont.)

**Toggle first two lines for one process**

**How does this solution break the CS requirements (mutual exclusion, progress, bounded waiting)?**
Outline of Lecture-09

- Problems with concurrent access to shared data
  - Race condition and critical section
  - General structure for enforce critical section
- Software based solutions:
  - Simple solution
  - Peterson’s solution
- Hardware based solution
  - Disable interrupts
  - TestAndSet
  - Swap
- OS solution -- Semaphore

Using semaphores to solve real issues

Hardware Solution 1: Disable Interrupt

- Unprocessors – could disable interrupts
  - Currently running code would execute without preemption

```
    do {
        ...  
        DISABLE INTERRUPT
        critical section
        ENABLE INTERRUPT
        RemainderSection
    } while (1);
```

What are the problems with this solution?

1. Mutual exclusion is preserved but efficiency of execution is degraded
   - while in CS, we cannot interleave execution with other processes that are in RS
2. On a multiprocessor, mutual exclusion is not preserved
3. Tracking time is impossible in CS

Hardware Support for Synchronization

- Synchronization
  - Need to check and set a value atomically
- IA32 hardware provides a special instruction: xchg
  - When the instruction accesses memory, the bus is locked during its execution and no other process/thread can access memory until it completes!!!
  - Other variations: xchgb, xchgw, xchg1
- Other hardware instructions
  - TestAndSet (a)
  - Swap (a,b)

Hardware Instruction: TestAndSet

- The TestAndSet instruction tests and modifies the content of a word atomically (non-interruptible)
- Keep setting the lock to 1 and return old value.

```c
    bool TestAndSet(bool *target){
        boolean m = *target;
        *target = true;
        return m;
    }
```

What’s the problem?

1. Busy-waiting, waste cpu
2. Hardware dependent, not bounded-waiting
Another Hardware Instruction: Swap

Swap contents of two memory words

```c
void Swap (bool *a, bool *b)
{
    bool temp = *a;
    *a = *b;
    *b = temp;
}
```

```c
bool lock = FALSE;
while(key == TRUE) {
    Swap(&key, &lock);
}
critical section;
lock = FALSE; //release permission
```

What’s the problem?

1. Busy-waiting, waste cpu
2. Hardware dependent, not bounded-waiting

LOCK == FALSE

Outline of Lecture-09

- Problems with concurrent access to shared data
  - Race condition and critical section
  - General structure for enforce critical section
- Software based solutions:
  - Simple solution
  - Peterson’s solution
- Hardware based solution
  - Disable interrupts
  - TestAndSet
  - Swap
- OS solution -- Semaphore
  - Using semaphores to solve real issues

Semaphores

- Synchronization without busy waiting
  - Motivation: Avoid busy waiting by blocking a process execution until some condition is satisfied
- Semaphore S – integer variable
- Two indivisible (atomic) operations:
  - wait(s) (also called P(s) or down(s) or acquire());
  - signal(s) (also called V(s) or up(s) or release())
  - User-visible operations on a semaphore
  - Easy to generalize, and less complicated for application programmers

Semaphores without Busy Waiting

- The idea:
  - Once need to wait, remove the process/thread from CPU
  - The process/thread goes into a special queue waiting for a semaphore (like an I/O waiting queue)
  - OS/runtime manages this queue (e.g., FIFO manner) and remove a process/thread when a signal occurs
- A semaphore consists of an integer (S.value) and a linked list of waiting processes/threads (S.list)
  - If the integer is 0 or negative, its magnitude is the number of processes/threads waiting for this semaphore.
- Start with an empty list, and normally initialized to 0
Implement Semaphores

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

wait(semaphore * s){
    s->value--;
    while(s->value < 0){
        enlist(s->list);
        block();
    }
}

signal(semaphore * s){
    s->value++;
    if (s->value <= 0){
        delist(P, s->list);
        wakeup(p);
    }
}

Is this one without busy waiting?

Semaphore Usage

- **Counting semaphore** — Can be used to control access to a given resource with finite number of instances.

- **Binary semaphore** — integer value can range only between 0 and 1; Also known as mutex locks.

Attacking CS Problem with Semaphores

- **Shared data**
  - Semaphore mutex = 1; /* initially mutex = 1 */

- For any process/thread
  do {
      wait(semaphore * s){
      ... 
      wait(mutex);    if (s->value < 0){
          critical section      enlist(s->list);
          signal(mutex);      block();
      }  
      remainder section  }
  while(1);
Revisit “Balance Update Problem”

- **Shared data:**
  - int Balance;
  - semaphore mutex = 1; // initially mutex = 1

- **Process A:**
  - ....
  - wait (mutex);
  - Balance = Balance – 100;
  - signal (mutex);
  - ....

- **Process B:**
  - ....
  - wait (mutex);
  - Balance = Balance – 200;
  - signal (mutex);
  - ....

Semaphore for General Synchronization

- Execute code B in P2 after code A was executed in P1.
- Use semaphore `flag` : what is the initial value?
- **Code**
  ```
  Semaphore flag = 0;
  P1 : 
  - A
  - signal(flag)
  B
  ```

- What about 2 threads wait for 1 thread?
  - Or 1 thread waits for 2 threads?

Classical Synchronization Problems

- **Producer-Consumer Problem**
  - Shared bounded-buffer
  - Producer puts items to the buffer area, wait if buffer is full
  - Consumer consumes items from the buffer, wait if is empty

- **Readers-Writers Problem**
  - Multiple readers can access concurrently
  - Writers mutual exclusive with writes/readers

- **Dining-Philosophers Problem**
  - Multiple resources, get one at each time
Producer-Consumer Problem

With Bounded-Buffer

- Need to make sure that
  - The producer and the consumer do not access the buffer area and related variables at the same time
  - No item is available to the consumer if all the buffer slots are empty.
  - No slot in the buffer is available to the producer if all the buffer slots are full.

Producer/Consumer Loops

- Producer Loop
- Consumer Loop

Potential Solution I

Problem: if counter = n now, then the producer is holding the semaphore

Potential Solution II

Problem: if there are multiple threads, and a producer produces one item, then it is possible that they will compete for the item
What Semaphores are needed?

- semaphore mutex, full, empty;

What are the initial values?

Initially:

- prodItems = 0 /* The number of produced items */
- availSlots = n /* The number of empty slots */
- mutex = 1 /* controlling mutual access to the buffer pool */

Producer/Consumer Loops

- Producer Loop

  ```
  produce an item in nextp
  while (counter == n)
  buffer[in] = nextp;
  in = (in + 1) % n;
  counter++;
  ```

- Consumer Loop

  ```
  consume the item in nextc
  while (counter == 0)
  nextc = buffer[out];
  out = (out + 1) % n;
  counter--; 
  ```

Producer-Consumer Codes

- Producer

  ```
  do { ...
  produce an item in nextp
  ... }
  wait(availSlots);
  wait(mutex);
  ... 
  add nextp to buffer
  ... 
  signal(mutex);
  signal(prodItems);
  } while (1)
  ```

- Consumer

  ```
  do { ...
  wait(prodItems)
  wait(mutex);
  ... 
  remove an item from buffer to nextc
  ... 
  signal(mutex);
  signal(availables);
  ... 
  consume the item in nextc
  ... 
  } while (1)
  ```

What will happen if we change the order?

Dining-Philosophers Problem

Five philosophers share a common circular table. There are five chopsticks and a bowl of rice (in the middle). When a philosopher gets hungry, he tries to pick up the closest chopsticks. A philosopher may pick up only one chopstick at a time, and cannot pick up a chopstick already in use. When done, he puts down both of his chopsticks, one after the other.

- Shared data

  ```
  semaphore chopstick[5];
  ```

Initially all semaphore values are 1

A classic example of a synchronization problem: allocate several resources among several processes in a deadlock-free and starvation-free manner.
Problems with Using Semaphores

Let $S$ and $Q$ be two semaphores initialized to 1.

\[
\begin{align*}
T_0 & \quad T_1 \\
\text{wait (S)}; & \quad \text{wait (Q)}; \\
\text{wait (Q)}; & \quad \text{wait (S)}; \\
… & \quad … \\
\text{signal (S)}; & \quad \text{signal (Q)}; \\
\text{signal (Q)}; & \quad \text{signal (S)};
\end{align*}
\]

What is the problem with the above code?

Problems with Using Semaphores (cont.)

- Strict requirements on the sequence of operations
  - Correct: wait (mutex) … signal (mutex)
  - Incorrect: signal (mutex) … wait (mutex);
  - wait (mutex) … wait(mutex);

- Complicated usages
- Incorrect usage could lead to deadlock

Summary

- Synchronization for coordinated processes/threads
- The produce/consumer and bounded buffer problem
- Critical Sections and solutions’ requirements
- Peterson’s solution for two processes/threads
- Synchronization Hardware: atomic instructions
- Semaphores
  - Its two operations wait and signal and standard usages
- Bounded buffer problem with semaphores
  - Initial values and order of operations are crucial