Concurrency: Mutual Exclusion and Synchronization

Chapter 5

Concurrency

Concurrency arises in 3 different contexts:

- **Multiple applications**
  - Multiprogramming: time slicing

- **Structured applications**
  - Develop a single application as set of concurrent processes

- **Operating system structure**
  - Often implemented as set of processes or threads
Concurrency: Related Terms

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>critical section</td>
<td>A section of code within a process that requires access to shared resources and which may not be executed while another process is in a corresponding section of code.</td>
</tr>
<tr>
<td>deadlock</td>
<td>A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.</td>
</tr>
<tr>
<td>livelock</td>
<td>A situation in which two or more processes continuously change their state in response to changes in the other process(es) without doing any useful work.</td>
</tr>
<tr>
<td>mutual exclusion</td>
<td>The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.</td>
</tr>
<tr>
<td>race condition</td>
<td>A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution.</td>
</tr>
<tr>
<td>starvation</td>
<td>A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.</td>
</tr>
</tbody>
</table>

Difficulties with Concurrency

- Sharing of global resources
  - Two processes reading from and writing to the same global variable… sequence of R/W is crucial

- Operating system managing the allocation of resources optimally
  - Process A acquires resource R and blocks, Process B wants resource R

- Difficult to locate programming errors
  - Non-deterministic behavior
Currency: Design Issues

- Communication among processes
- Sharing resources
- Synchronization of multiple processes
- Allocation of processor time

A Simple Example

Global Char: chin, chout

<table>
<thead>
<tr>
<th>Process P1</th>
<th>Process P2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>chin = getchar();</td>
<td>chin = getchar();</td>
</tr>
<tr>
<td></td>
<td>chout = chin;</td>
</tr>
<tr>
<td>chout = chin;</td>
<td>chout = chout;</td>
</tr>
<tr>
<td>putchar(chout);</td>
<td>putchar(chout);</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>“chin” in P1 is lost</td>
<td></td>
</tr>
</tbody>
</table>
Another Simple Example

Global Char: \( b = 1, c = 2; \)

Process P1

\( b = b + c \)

Process P2

\( c = b + c \)

\( b = 3, c = 5 \)

\( b = 4, c = 3 \)

Race Condition

Operating System Concerns

- Keeping track of multiple and distinct processes
- Allocate and deallocate resources
  - Processor time
  - Memory
  - Files
  - I/O devices
- Protect data and resources
- Output of process must be independent of the speed of execution of other concurrent processes
  - Deterministic
Process Interaction

Given concurrency, how can processes interact with each other?

• Processes *unaware* of each other
  – Independent processes not intended to work together
  – Compete for resources

• Processes *indirectly aware* of each other
  – Share access to resources
  – Sharing is cooperative

• Process *directly aware* of each other
  – Designed to work jointly on some activity
  – Sharing is cooperative

Resource Sharing Among Concurrent Processes

• Mutual Exclusion
  – Critical sections: used when accessing shared resource
    • Only one program at a time is allowed in its critical section
    • Example: one process at a time allowed to send command to printer

• Deadlock
  – No computational progress can be made because a set of processes are blocked waiting on processes that will never be available

• Starvation
  – A process’ resource request is never accommodated
Critical Section Problem (Revisited)

shared float balance;

/* Code schema for p1 */
.. balance = balance + amount;
.. /* Code schema for p2 */
.. balance = balance - amount;

/* Schema for p1 */
/* X == balance */
load R1, X
load R2, Y
add R1, R2
store R1, X

/* Schema for p2 */
/* X == balance */
load R1, X
load R2, Y
sub R1, R2
store R1, X

Critical Section Problem...

/* Schema for p1 */
\begin{align*}
1 & \{ \text{load } R1, X \\
2 & \{ \text{load } R2, Y \\
3 & \{ \text{add } R1, R2 \\
4 & \{ \text{store } R1, X \\
\end{align*}

/* Schema for p2 */
\begin{align*}
5 & \{ \text{load } R1, X \\
6 & \{ \text{load } R2, Y \\
7 & \{ \text{sub } R1, R2 \\
8 & \{ \text{store } R1, X \\
\end{align*}

- Suppose:
  - Execution sequence : 1, 2, 3
    - Lost update : 2
  - Execution sequence : 1, 4, 3 ,6
    - Lost update : 3
- Together => non-determinacy
- Race condition exists
Requirements for Mutual Exclusion

• Only one process at a time is allowed in the critical section for a resource
• A process that halts in its noncritical section must do so without interfering with other processes
• No deadlock or starvation

Requirements for Mutual Exclusion

• A process must not be delayed when accessing a critical section if there is no other process using it
• No assumptions are made about relative process speeds or number of processes
• A process remains inside its critical section for a finite time only
Mutual Exclusion & Synchronization

Hardware Support

**Interrupt**

**Test & Set**

**Exchange**

---

**Mutual Exclusion: Hardware Support**

**Interrupt Disabling**

```java
While (true) {
    disable-interrupts
    critical section
    enable-interrupts
}
```

- Processor is limited in its ability to interleave programs
- Disabling interrupts guarantees mutual exclusion
- Multiprocessor Environment
  - disabling interrupts on one processor will not guarantee mutual exclusion
Critical Section Problem

shared float balance;

/* Code schema for p1 */
.. disable-interrupts;
  balance = balance + amount;
enable-interrupts;
.. /* Schema for p1 */

Interrupts turned off
  load R1, X
  load R2, Y
  add R1, R2
  store R1, X

//interruptible

Interrupts turned on

/* Code schema for p2 */
.. disable-interrupts;
  balance = balance - amount;
enable-interrupts;
.. /* Schema for p2 */

Interrupts turned off
  load R1, X
  load R2, Y
  sub R1, R2
  store R1, X

Interrupts turned on

---

Mutual Exclusion: Hardware Support

• Special Machine Instructions
  – Performed in a single instruction cycle
  – Performs memory access / manipulation
  – No concurrent access to that memory location

• Instructions
  – Test & Set
  – Exchange
The “Test & Set” Instruction

```java
boolean testset (int i) {
    if (i == 0) {
        i = 1;
        return true;
    }
    else {
        return false;
    }
}
EXECUTED ATOMICALLY
```

The “Test & Set” Instruction

```java
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
    while (true)
    {
        while (!testset (bolt))
            /* do nothing */;
        /* critical section */;
        bolt = 0;
        /* remainder */
    }
}
void main()
{
    bolt = 0;
    parbegin (P(1), P(2), . . . ,P(n));
}
```
The "Exchange" Instruction

```c
void exchange(int register, int memory)
{
    int temp;
    temp = memory;
    memory = register;
    register = temp;
}
```

EXECUTED ATOMICALLY

The "Exchange" Instruction

```c
/* program mutual exclusion */
int const n = /* number of processes*/;
int bolt;
void P(int i)
{
    int keyi;
    while (true)
    {
        keyi = 1;
        while (keyi != 0)
        {
            exchange (keyi, bolt);
            /* critical section */
            exchange (keyi, bolt);
            /* remainder */
        }
    }
}
void main()
{
    bolt = 0;
    parbegin (P(1), P(2), . . ., P(n));
}
```
Mutual Exclusion Machine Instructions

• Advantages
  – Applicable to any number of processes on either a single processor or multiple processors sharing main memory
  – It is simple and therefore easy to verify
  – It can be used to support multiple critical sections
    • Different variable set for each CR

• Disadvantages
  – Busy-waiting consumes processor time
  – Starvation is possible when a process leaves a critical section and more than one process is waiting.
  – Deadlock
    • If a low priority process has the critical region and a higher priority process needs it, the higher priority process will obtain the processor to wait for the critical region
Mutual Exclusion & Synchronization

Language / OS Defined

The Semaphore

Semaphore

- Dijkstra, 1965
- Synchronization primitive with no busy waiting
- It is an integer variable changed or tested by one of the two indivisible operations
- Actually implemented as a protected variable type
  
  ```
  var x : semaphore
  ```
Semaphore operations

- **semWait(S)** operation ("wait")
  - Requests permission to use a critical resource
    
    \[
    S := S - 1; \\
    \text{if } (S < 0) \text{ then} \\
    \quad \text{put calling process on queue}
    \]

- **semSignal(S)** operation ("signal")
  - Releases the critical resource
    
    \[
    S := S + 1; \\
    \text{if } (S \leq 0) \text{ then} \\
    \quad \text{remove one process from queue}
    \]

- Queues are associated with each semaphore variable

Semaphore : Example

Critical resource T
Semaphore S \leftarrow \text{initial_value}
Processes A,B

<table>
<thead>
<tr>
<th>Process A</th>
<th>Process B</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>\text{semWait}(S);</td>
<td>\text{semWait}(S);</td>
</tr>
<tr>
<td>&lt;CS&gt; /* access T */</td>
<td>&lt;CS&gt; /* access T */</td>
</tr>
<tr>
<td>\text{semSignal}(S);</td>
<td>\text{semSignal}(S);</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
</tbody>
</table>
Semaphore: Example...

```plaintext
var S : semaphore ← 1

Queue associated with S

Value of S: 1

<table>
<thead>
<tr>
<th>Process A</th>
<th>Process B</th>
<th>Process C</th>
</tr>
</thead>
<tbody>
<tr>
<td>semWait(S);</td>
<td>semWait(S);</td>
<td>semWait(S);</td>
</tr>
<tr>
<td>&lt;CS&gt;</td>
<td>&lt;CS&gt;</td>
<td>&lt;CS&gt;</td>
</tr>
<tr>
<td>semSignal(S);</td>
<td>semSignal(S);</td>
<td>semSignal(S);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Types of Semaphores

- **Binary Semaphores**
  - Maximum value is 1

- **Counting Semaphores**
  - Maximum value is greater than 1

- Both use similar `semWait` and `semSignal` definitions
- Synchronizing code and initialization determines what values are needed, and therefore, what kind of semaphore will be used

The remaining discussion will focus primarily on **counting semaphores**
Using Semaphores

Shared semaphore \texttt{mutex} <= 1;

\begin{verbatim}
proc_1() {
  while(true) {
    <compute section>;
    semWait(mutex);
    <critical section>;
    semSignal(mutex);
  }
}
proc_2() {
  while(true) {
    <compute section>;
    semWait(mutex);
    <critical section>;
    semSignal(mutex);
  }
}
\end{verbatim}

1. P1 \implies \texttt{semWait(mutex)}
   - Decrements; \textless 0 ?; NO (0);
   - P1 Enters CS;
   - P1 interrupted

2. P2 \implies \texttt{semWait(mutex)}
   - Decrements; \textless 0 ?; YES (-1)
   - P2 \texttt{blocks on mutex}

3. P1 finishes CS work
   - P1 \implies \texttt{semSignal(mutex)};
   - Increments; \textless= 0 ?; YES (0)
   - P2 woken & proceeds

Non-Interruptible “Test & Sets”

\begin{verbatim}
proc_0() {
  ...
  semWait(mutex);
  balance = balance + amount;
  semSignal(mutex);
  ...
}
proc_1() {
  ...
  semWait(mutex);
  balance = balance - amount;
  semSignal(mutex);
  ...
}
\end{verbatim}

Suppose P1 issues \texttt{semWait(mutex)} first

\begin{verbatim}
Suppose P2 issues \texttt{semWait(mutex)} first
\end{verbatim}

\textbf{No Problem}

Note: Could use Interrupts to implement solution,
But (1) with interrupts masked off, what happens if a prior I/O request is satisfied
(2) Interrupt approach would not work on Multiprocessor
Using Semaphores – Example 2

Shared semaphore: s1 <= 0, s2 <= 0;  

Note: values started at 0... ok?

proc_A() {
    while(true) {
        <compute A1>
        write(x);
        semSignal(s1);
        <compute A2>
        semWait(s2);
        read(y);
    }
}

proc_B() {
    while(true) {
        A signals B that "write to x" has completed
        semWait(s1);
        read(x);
        <compute B1>
        write(y);
        semSignal(s2);
        <compute B2>
    }
}

• Cannot use Interrupt disable/enable here because we have multiple distinct synchronization points
• Interrupt disable/enable can only distinguish 1 synchronization event
• Therefore, 2 Semaphores

B blocks till A signals

A blocks until B signals

Producer / Consumer Problem
(Classic)

• Critical resource
  – Set of message buffers
• 2 Processes
  – Producer: Creates a message and places it in the buffer
  – Consumer: Reads a message and deletes it from the buffer
• Objective
  – Allow the producer and consumer to run concurrently
P/C...

• Constraints
  – Producer must have a non-full buffer to put its message into
  – Consumer must have a non-empty buffer to read
  – Mutually exclusive access to Buffer pool

• Unbounded Buffer problem
  – Infinite buffers
  – Producer never has to wait
  – Not interesting nor practical

• Bounded Buffer Problem
  – Limited set of buffers

---

P/C - Solution

Shared Full: semaphore \(\sigma\) \(\leftarrow 0\);
Empty semaphore \(\sigma\) \(\leftarrow \text{MaxBuffers}\);
MEPC: semaphore \(\sigma\) \(\leftarrow 1\);

Producer
Begin
...
semWait(Empty);
semWait(MEPC);
<add item to buffer>
semSignal(MEPC);
semSignal(Full);
...
End;

Consumer
Begin
...
semWait(Full);
semWait(MEPC);
<remove item from buffer>
semSignal(MEPC);
semSignal(Empty);
...
End;
P/C – Another Look

- 9 Baskets – Bounded
- Consumer – Empties basket
  - Can only remove basket from Full Pool, if one is there
    => Need “full” count
  - Empties basket and places it in Empty pool
- Producer – Fills basket
  - Can only remove basket from Empty pool, if one is there
    => Need “empty” count
  - Fills basket and places it in Full pool
P/C - Another Look

Shared semaphore: Emutex = 1, Fmutex = 1; full = 0, empty = 9;
Shared buf_type: buffer[9];

producer() {
    buf_type *next, *here;
    while(True) {
        produce_item(next);
        semWait(empty); /*Claim empty buff*/
        semWait(Emutex); /*Manipulate pool*/
        here = obtain(empty);
        semSignal(Emutex);
        copy_buffer(next, here);
        semWait(Fmutex); /*Manipulate pool*/
        release(here, fullpool);
        semSignal(Fmutex); /*Sgnl full buff*/
        semSignal(full);
    }
}

c consumer() {
    buf_type *next, *here;
    while(True) {
        semWait(full); /*Claim full buff*/
        semWait(Fmutex); /*Manipulate pool*/
        here = obtain(full);
        semSignal(Fmutex);
        copy_buffer(here, next);
        semWait(Emutex); /*Manipulate pool*/
        release(here, emptypool);
        semSignal(Emutex); /*Sgnl empt buff*/
        semSignal(empty);
        consume_item(next);
    }
}

P/C - Example

- How realistic is P/C scenario?
- Consider a circular buffer
  - 12 slots
  - Producer points at next one it will fill
  - Consumer points at next one it will empty
- Don’t want:

  Producer = Consumer

  => (1) Consumer “consumed” faster than producer “produced”, or
  (2) Producer “produced” faster than consumer “consumed”.

![Diagram showing synchronization access to buffer](image-url)
P/C – Real World Scenario

• CPU can produce data faster than terminal can accept or viewer can read

![Diagram showing CPU and Terminal with communication buffers and Xon/Xoff Flow Control]

Communication buffers in both
Xon/Xoff Flow Control

Semaphores: Other Primitives

Semaphore: S = 1;

• S.queue: interrogate whether the queue is empty or non-empty

• S.count: current semaphore value
Mutual Exclusion and Synchronization

Language Defined

The Monitor

Monitors

• Monitor is a software module

• Chief characteristics
  – Local data variables are accessible only by the monitor
  – Process enters monitor by invoking one of its procedures
  – Only one process may be executing in the monitor at a time
Monitor Structure

- Entrance only through monitor procedure
- Condition variables allows process suspension and “removal” from monitor
  Queue associated with each condition variable
- Local data can only be accesses through monitor procedures

Producer / Consumer: Monitor Solution

```c
void producer()
char x;
{
    while (true)
    {
        produce(x);
        append(x);
    }
}

void consumer()
char x;
{
    while (true)
    {
        take(x);
        consume(x)
    }
}

void main()
{
    parbegin {produced, consumer};
}
```

Monitor

```c
monitor boundedbuffer;
char Buffer (N)

void append (char x)
{
    :
    :
}

void take (char x)
{
    :
    :
}
```
Monitor Accolades

- Provides equivalent functionality to that of semaphore
- Monitor construct itself enforces mutual exclusion
- Abstract Data Type – data, procedures, encapsulation
  - Initialization procedures
  - Local data only accessible to monitor procedures
  - Procedures (methods)
- All access and data manipulation defined / controlled at one place
Mutual Exclusion & Synchronization through Message Passing

Message Passing

- Enforce mutual exclusion
- Exchange information

**Typical Forms**

send (destination, message)  
receive (source, message)
Send / Receive Scenarios

- Send primitive is executed
  - Sender is blocked until message is received, or
  - Sender continues
- Receive primitive is issued
  - Message previously sent, message received, execution continues, or
  - No message waiting and
    - Process blocks until message arrives, or
    - Process continues executing… abandons attempt to read a message

Send / Receive Synchronization

- Blocking send, blocking receive
  - Both sender and receiver are blocked until message is delivered
  - Called a *rendezvous*
- Nonblocking send, blocking receive
  - Sender continues on
  - Receiver is blocked until the requested message arrives
- Nonblocking send, nonblocking receive
  - Neither party is required to wait
Direct Addressing

- Send primitive includes a specific identifier of the destination process
  - Send (452, Msg)

- Receive primitive could know ahead of time from which process a message is expected
  - Receive (384, &Msg)

- Receive primitive could use source parameter to return a value when the receive operation has been performed
  - Receive (&PID, &Msg)

Indirect Addressing

Messages are sent to a shared data structure NOT to a specified process

- Queues are called mailboxes / tuple-space

- One process sends a message to the mailbox and the other process picks up the message from the mailbox
  - Mailboxes may / may not be tied to process instances
Indirect Addressing

One to One

- Private communication link
- Connections through *ports*
- Reduces potential interference from other processes
- Client / Server Applications
- Mail referred to as a *port*

Many to One

Indirect Addressing

One to Many

- One sender, multiple receivers
- Broadcast

Many to Many

- Multiple Servers providing *concurrent* services to multiple clients
General Message Format

<table>
<thead>
<tr>
<th>Header</th>
<th>Body</th>
<th>Allows for variable length messages (most common)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Type</td>
<td>Message Contents</td>
<td></td>
</tr>
<tr>
<td>Destination ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Source ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Message Length</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Control Information</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Achieving Mutual Exclusion via Messages

```c
/* program mutualexclusion */
const int n = /* number of processes */;
void P(int i)
{
    message msg;
    while (true)
    {
        receive (mutex, msg);
        /* critical section */;
        send (mutex, msg);
        /* remainder */;
    }
}
void main()
{
    create_mailbox (mutex);
    send (mutex, null);
    parbegin (P(1), P(2), . . . , P(n));
}
```

- Blocking
  Receive / Send
- One message
  - “token”

He who gets the token, enters the Critical Section
Solving the P/C Problem

void producer()
{
    message m;
    while (true)
    {
        receive (mayproduce, m);
        m = produce();
        send (mayconsume, m);
    }
}

void consumer()
{
    message m;
    while (true)
    {
        receive (mayconsume, m);
        consume (m);
        send (mayproduce, null);
    }
}

void main()
{
    create_mailbox (mayproduce);
    create_mailbox (mayconsume);
    for (int i = 1; i <= capacity; i++)
    {
        send (mayproduce, null);
        parbegin (producer, consumer);
    }
}

Readers/Writers Problem

- Any number of readers may simultaneously read the file
- Only one writer at a time may write to the file
- If a writer is writing to the file, no reader may read it
Readers have Priority

```c
void main()
{
    readcount = 0;
    parbegin (reader, writer);
}
```

- "x" guards updating of readcount
- "wsem" informs writer process if
  - one or more readers reading, or
  - another writer writing

Reader Priority: As long as any reader is "reading", another reader can enter to read.

Writers Have Priority

```c
void main()
{
    readcount = writecount = 0;
    parbegin (reader, writer);
}
```