3. Higher-Level Synchronization

3.1 Shared Memory Methods
   – Monitors
   – Protected Types

3.2 Distributed Synchronization/Comm.
   – Message-Based Communication
   – Procedure-Based Communication
   – Distributed Mutual Exclusion

3.3 Other Classical Problems
   – The Readers/Writers Problem
   – The Dining Philosophers Problem
   – The Elevator Algorithm
   – Event Ordering with Logical Clocks
3.1 Shared Memory Methods

- Monitors
- Protected Types
Motivation

• Semaphores and Events are:
  – Powerful but low-level abstractions
    • Programming with them is highly error prone
    • Such programs are difficult to design, debug, and maintain
  – Not usable in distributed memory systems

• Need higher-level primitives
  – Based on semaphores or messages
Monitors

– Follow principles of abstract data types (object-oriented programming):
  • A data type is manipulated only by a set of predefined operations

– A monitor is
  1. A collection of data representing the state of the resource controlled by the monitor, and
  2. Procedures to manipulate the resource data
Monitors

• Implementation must guarantee:
  1. Resource is **only accessible by monitor procedures**
  2. Monitor procedures are **mutually exclusive**

• For coordination, monitors provide:
  c.wait
    • Calling process is blocked and placed on waiting queue associated with condition variable \( c \)
  c.signal
    • Calling process wakes up first process on queue associated with \( c \)
Monitors

• “condition variable” \( c \) is \textit{not a conventional variable}
  – \( c \) has no value
  – \( c \) is an arbitrary name chosen by programmer
    • By convention, the name is chosen to reflect the an event, state, or condition that the condition variable represents
  – Each \( c \) has a waiting queue associated
  – A process may “block” itself on \( c \) -- it \textit{waits} until another process issues a \textit{signal} on \( c \)
Monitors

• Design Issue:
  – After `c.signal`, there are 2 ready processes:
    • The calling process which did the `c.signal`
    • The blocked process which the `c.signal` “woke up”
  – Which should continue?
    (Only one can be executing inside the monitor!)

Two different approaches
– Hoare monitors
– Mesa-style monitors
Hoare Monitors

• Introduced by Hoare in a 1974 CACM paper
• First implemented by Per Brinch Hansen in Concurrent Pascal
• Approach taken by Hoare monitor:
  – After `c.signal`,
    • Awakened process continues
    • Calling process is suspended, and placed on high-priority queue
Hoare Monitors

Effect of \textbf{wait}

Effect of \textbf{signal}

Figure 3-2
Bounded buffer problem

monitor BoundedBuffer
{
    char buffer[n];
    int nextin=0, nextout=0, fullCount=0;
    condition notempty, notfull;

    deposit(char data)
    {
        ...
    }

    remove(char data)
    {
        ...
    }
}
Bounded buffer problem

deposit(char data)
{
    if (fullCount==n) notfull.wait;
    buffer[nextin] = data;
    nextin = (nextin+1) % n;
    fullCount = fullCount+1;
    notempty.signal;
}

remove(char data)
{
    if (fullCount==0) notempty.wait;
    data = buffer[nextout];
    nextout = (nextout+1) % n;
    fullCount = fullCount - 1;
    notfull.signal;
}
Priority waits

• Hoare monitor `signal` resumes longest waiting process (i.e., queue is a FIFO queue)

• Hoare also introduced “Priority Waits” (aka “conditional” or “scheduled”):
  – `c.wait(p)`
    – `p` is an integer (priority)
    – Blocked processes are kept sorted by `p`
  – `c.signal`
    – Wakes up process with *lowest* (!) `p`
Example: alarm clock

- Processes can call `wakeMe(n)` to sleep for `n` clock ticks
- After the time has expired, call to `wakeMe` returns
- Implemented using Hoare monitor with priorities
Example: alarm clock

monitor AlarmClock {
  int now=0;
  condition wakeup;

  wakeMe(int n) {
    int alarm;
    alarm = now + n;
    while (now<alarm)wakeup.wait(alarm);
    wakeup.signal;
  }
  tick() {
    /*invoked by hardware*/
    now = now + 1;
    wakeup.signal;
  }
}

Example: alarm clock

- **tick** only wakes up one process
- Multiple processes with same alarm time awaken in a chain:
  - **tick** wakes up the first process
  - the first process wakes up the second process via the `wakeup.signal` in `wakeme`
  - etc.
- Without priority waits, all processes would need to wake up to check their alarm settings
Mesa-style monitors

• Variant defined for the programming language Mesa

• `notify` is a variant of `signal`

• After `c.notify`:
  – Calling process continues
  – Awakened process continues when caller exits

• Problem
  – Caller may wake up multiple processes $P_1, P_2, P_3, \ldots$
  – $P_1$ could change condition on which $P_2$ was blocked.
Mesa monitors

• Solution
  instead of: if (!condition) c.wait
  use: while (!condition) c.wait

• signal vs notify
  – *(Beware: There is no universal terminology)*
  – signal may involve caller “stepping aside”
  – notify usually has caller continuing
  – signal “simpler to use” but notify may be more efficiently implemented
Monitors in Java

• Java supports **synchronized** methods, which permit Java objects to be used somewhat similarly to Mesa monitors
  – Every object has an implicit lock, with a single associated condition
  – If a method is declare synchronized, the object’s lock protects the entire method
  – \texttt{wait()} causes a thread to wait until it is notified
  – \texttt{notifyAll()} awakens all threads waiting on the object’s lock
  – \texttt{notify()} awakens a single randomly chosen thread waiting on the object’s lock

• But there are differences…
Differences between Java objects and monitors

- **Monitors**
  1. Resource is only accessible by monitor procedures
  2. Monitor procedures are mutually exclusive

- **Java objects**
  1. Fields are not required to be private
  2. Methods are not required to be synchronized

Per Brinch Hansen: “It is astounding to me that Java’s insecure parallelism is taken seriously by the programming community, a quarter of a century after the invention of monitors and Concurrent Pascal. It has no merit.” [Java’s Insecure Parallelism, ACM SIGPLAN Notices 34: 38-45, April 1999].
Protected types (Ada 95)

• Encapsulated objects with public access procedures called *entries*.

• Equivalent to special case of monitor where
  – *c.wait* is the *first* operation of a procedure
  – *c.signal* is the *last* operation

• *wait/signal* combined into a *when* clause
  – The *when c* construct forms a *barrier*
  – Procedure continues only when the condition *c* is true
Example

entry deposit(char c)
    when (fullCount < n)
    {
        buffer[nextin] = c;
        nextin = (nextin + 1) % n;
        fullCount = fullCount + 1;
    }

entry remove(char c)
    when (fullCount > 0)
    {
        c = buffer[nextout];
        nextout = (nextout + 1) % n;
        fullCount = fullCount - 1;
    }
3.2 Distributed Synchronization and Communication

• Message-based Communication
  – Direct message passing
  – Indirect message passing: channels, ports, mailboxes

• Procedure-based Communication
  – Remote Procedure Calls (RPC)
  – Rendezvous

• Distributed Mutual Exclusion
Distributed Synchronization

• Semaphore-based primitive requires shared memory
• For distributed memory:
  – send(p,m)
    • Send message m to process p
  – receive(q,m)
    • Receive message from process q in variable m
• Semantics of send and receive vary significantly in different systems.
Distributed Synchronization

• Types of send/receive:
  – Does sender wait for message to be accepted?
  – Does receiver wait if there is no message?
  – Does sender name exactly one receiver?
  – Does receiver name exactly one sender?
## Types of send/receive

<table>
<thead>
<tr>
<th></th>
<th>blocking</th>
<th>nonblocking</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>send</strong></td>
<td><strong>blocking</strong></td>
<td><strong>nonblocking</strong></td>
</tr>
<tr>
<td>explicit naming</td>
<td>send m to r</td>
<td>send m to r</td>
</tr>
<tr>
<td></td>
<td>wait until accepted</td>
<td></td>
</tr>
<tr>
<td>implicit naming</td>
<td>broadcast m</td>
<td>broadcast m</td>
</tr>
<tr>
<td></td>
<td>wait until accepted</td>
<td></td>
</tr>
<tr>
<td><strong>receive</strong></td>
<td><strong>blocking</strong></td>
<td><strong>nonblocking</strong></td>
</tr>
<tr>
<td>explicit naming</td>
<td>wait for message from s</td>
<td>if there is a message from s, receive it; else proceed</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>implicit naming</td>
<td>wait for message from any sender</td>
<td>if there is a message from any sender, receive it; else proceed</td>
</tr>
</tbody>
</table>
Channels, Ports, and Mailboxes

• Allow indirect communication
• Senders/ Receivers name channel/port/mailbox instead of processes
• Senders/ Receivers determined at runtime
  – Sender does not need to know who receives the message
  – Receiver does not need to know who sent the message
Named Message Channels

- Named channel, \( 	ext{ch1} \), connects processes \( p1 \) and \( p2 \)
- \( p1 \) sends to \( p2 \) using \( \text{send(ch1, }"a"\text{)} \)
- \( p2 \) receives from \( p1 \) using: \( \text{receive(ch1,x)} \)
- Used in CSP/Occam: Communicating Sequential Processes in the Occam Programming Language (Hoare, 1978)
Named Message Channels in CSP/Occam

– Receive statements may be implemented as *guarded commands*

  • Syntax: \texttt{when (c1) s1}
  
  • \texttt{s} is *enabled* (able to be executed) only when \texttt{c} is true
  
  • If more than one guarded command is enabled, one of them is selected for execution
  
  • The condition \texttt{c} may contain receive statements, which evaluate to true if and only if the sending process is ready to send on the specified channel.
  
  • Allow processes to receive messages selectively based on arbitrary conditions
Example: Bounded buffer with CSP

• Producer $P$, Consumer $C$, and Buffer $B$ are Communicating Sequential Processes

• Problem statement:
  – When Buffer full: $B$ can only send to $C$
  – When Buffer empty: $B$ can only receive from $P$
  – When Buffer partially filled: $B$ must know whether $C$ or $P$ is ready to act

• Solution:
  – $C$ sends request to $B$ first; $B$ then sends data
  – Inputs to $B$ from $P$ and $C$ are guarded with `when` clause
Bounded Buffer with CSP

• Define 3 named channels
  – deposit: \( P \rightarrow B \)
  – request: \( B \leftarrow C \)
  – remove: \( B \rightarrow C \)

• P does:
  – send(deposit, data);

• C does:
  – send(request)
  – receive(remove, data)

• Code for B on next slide
Bounded buffer with CSP

process BoundedBuffer
{
   ...
   while (1) {
      when (((fullCount<n) && receive(deposit, buf[nextin])))
      {
         nextin = (nextin + 1) % n;
         fullCount = fullCount + 1;
      }  or
      when (((fullCount>0) && receive(request)))
      {
         send(remove, buf[nextout]);
         nextout = (nextout + 1) % n;
         fullCount = fullCount - 1;
      }
   }
}
Ports and Mailboxes

• Indirect communication (named message channels) allows a receiver to receive from multiple senders (nondeterministically)

• When channel is a queue, send can be nonblocking

• Such a queue is called *mailbox* or *port*, depending on number of receivers:
  – A **mailbox** can have multiple receivers
    • This can be expensive because receivers referring to the same mailbox may reside on different computers
  – A **port** can have only one receiver
    • So all messages addressed to the same port can be sent to one central place.
Ports and Mailboxes

Figure 3-2
UNIX implements of interprocess communication

2 mechanisms: pipes and sockets

• **Pipes**: Sender’s standard output is receiver’s standard input
  \[ p_1 \mid p_2 \mid \ldots \mid p_n \]

• **Sockets** are named endpoints of a 2-way channel between 2 processes. Processes may be on different machines. To establish the channel:
  – One process acts as a server, the other a client
  – Server binds it socket to IP address of its machine and a port number
  – Server issues an `accept` statement and blocks until client issues a corresponding `connect` statement
  – The `connect` statement supplies the client’s IP address and port number to complete the connection.
Procedure-Based Communication

• Send/Receive are low level (like P/V)
• Typical interaction: Send Request and then Receive Result
  Make this into a single higher-level primitive
• Use RPC (Remote Procedure Call) or Rendezvous
  – Caller invokes procedure on remote machine
  – Remote machine performs operation and returns result
  – Similar to regular procedure call, but parameters cannot contain pointers or shared references, because caller and server do not share any memory
**RPC**

- Caller issues:
  \[ \text{result} = f(\text{params}) \]
- This is translated into:

```
Calling Process
...
send(server,f,params);
receive(server,result);
...
```

```
Server Process
process RP_server
{
  while (1)
  {
    receive(caller,f,params);
    result=f(params);
    send(caller,result);
  }
}
```
Rendezvous

– With RPC: Called process $p$ is part of a dedicated server

– With Rendezvous:
  • $p$ is part of an arbitrary process
  • $p$ maintains state between calls
  • $p$ may accept/delay/reject call
  • Setup is symmetrical:
    Any process may be a client or a server
Rendezvous (Ada 95)

• Caller: Similar syntax/semantics to RPC
  \[ q.f(param) \]
  where \( q \) is the called process (server)

• Server: Must indicate willingness to accept:
  \[ \text{accept } f(param) S \]

• Rendezvous:
  Caller (calling process) or Server (called process)
  waits for the other,
  Then they execute in parallel.

• (“Rendezvous” is French for “meeting.”)
Rendezvous

Figure 3-3
Rendezvous

• To permit selective receive, Ada provides guarded when clauses (like in CSP/Occam) through the select statement.

• For an accept statement to be selected:
  – the when clause guarding it must be true; and
  – there must be at least one pending procedure call to the accept statement.

```
select {
    [when B1:] accept E1(…) S1;
    or
    [when B2:] accept E2(…) S2;
    or
    ...
    [when Bn:] accept En(…) Sn;
    [else R]
}
```
Example: Bounded Buffer

process BoundedBuffer {
  while(1) {
    select {
      when (fullCount < n):
        accept deposit(char c) {
          buffer[nextin] = c;
          nextin = (nextin + 1) % n;
          fullCount = fullCount + 1;
        }
    }
    or
    when (fullCount > 0):
      accept remove(char c) {
        c = buffer[nextout];
        nextout = (nextout + 1) % n;
        fullCount = fullCount - 1;
      }
  }
}
Distributed Mutual Exclusion

• Critical Section problem in a Distributed Environment
  – Several processes share a resource (a printer, a satellite link, a file…)
  – Only one process can use the resource at a time

• Additional Challenges:
  – No shared memory
  – No shared clock
  – Delays in message transmission.
Distributed Mutual Exclusion

• Central Controller Solution
  – Requesting process sends request to controller
  – Controller grants it to one processes at a time
  – Problems with this approach:
    • Single point of failure,
    • Performance bottleneck

• Fully Distributed Solution:
  – Processes negotiate access among themselves
Distributed Mutual Exclusion

• Token Ring solution
  – Each process has a controller
  – Controllers are arranged in a ring
  – Controllers pass a token around the ring
  – Process whose controller holds token may enter its CS
Distributed Mutual Exclusion with Token Ring

Figure 3-4
Distributed Mutual Exclusion

process controller[i] {
    while(1) {
        accept Token;
        select {
            accept Request_CS() {busy=1;}
            else null;
        }
        if (busy) accept Release_CS() {busy=0;}
        controller[(i+1) % n].Token;
    }
}

process p[i] {
    while(1) {
        controller[i].Request_CS();
        CSi;
        controller[i].Release_CS();
        programi;
    }
}
3.3 Other Classical Synchronization Problems

• The Readers/Writers Problem
• The Dining Philosophers Problem
• The Elevator Algorithm
• Event Ordering with Logical Clocks
Readers/Writers Problem

• Extension of basic Critical Section (CS) problem (Courtois, Heymans, and Parnas, 1971)
• Two types of processes entering a CS: Readers (R) and Writers (W)
• CS may only contain
  – A single W process (and no R processes); or
  – Any number of R processes (and no W processes).
• This is a relaxation of the mutual exclusion condition, because multiple readers are allowed at one.
• A good solution should:
  – Satisfy this relaxed extended mutual exclusion condition
  – Take advantage of the fact that multiple R processes can be in the CS simultaneously
  – Prevent starvation of either process type
Readers/Writers Problem

- Two possible algorithms:
  1. *R* has priority over *W*: No *R* is kept waiting unless a *W* has already obtained permission to enter the CS.
  2. *W* has priority over *R*: When a *W* is waiting, only those *R* processes already granted permission to read are allowed to continue. All other *R* processes must wait until the *W* completes.
- Both of the above algorithms lead to starvation.
Readers/Writers Problem

- Solution that prevents starvation of either process type:
  1. If \textbf{R} processes are in CS, a new \textbf{R} cannot enter if a \textbf{W} is waiting
  2. If a \textbf{W} is in CS, once it leaves, all \textbf{R} processes waiting can enter, \textit{even if they arrived after new \textbf{W} processes that are also waiting.}
Solution using monitor

monitor Readers_Writers {
  int readCount=0, writing=0;
  condition OK_R, OK_W;

  start_read()
  {
    if (writing || !empty(OK_W))
      OK_R.wait;
    readCount = readCount + 1;
    OK_R.signal;
  }

  end_read()
  {
    readCount = readCount - 1;
    if (readCount == 0)
      OK_W.signal;
  }

  start_write()
  {
    if ((readCount !=0) || writing)
      OK_W.wait;
    writing = 1;
  }

  end_write()
  {
    writing = 0;
    if (!empty(OK_R))
      OK_R.signal;
    else OK_W.signal;
  }
}
Dining philosophers Problem

• Each philosopher needs both forks to eat
• Requirements
  – Prevent deadlock
  – Guarantee fairness: no philosopher must starve
  – Guarantee concurrency: non-neighbors may eat at the same time

Figure 3-5
Dining philosophers problem

- One obvious solution: each philosopher grabs left fork first

\[
p(i) : \\
\text{while } (1) \\
\text{think}(i); \\
\text{grab_forks}(i); \\
\text{eat}(i); \\
\text{return_forks}(i); \\
\}
\]

\[
\text{grab_forks}(i): \{ \text{P}(f[i]); \text{P}(f[i\%5 + 1]) \}
\]

\[
\text{return_forks}(i): \{ \text{V}(f[i]); \text{V}(f[i\%5 + 1]) \}
\]

- May lead to deadlock (each philosopher has left fork, is waiting for right fork)
Dining Philosophers

- Two possible solutions to deadlock
  1. Use a counter:
     At most n–1 philosophers may attempt to grab forks
  2. One philosopher requests forks in reverse order, e.g.,
     
     \[
     \text{grab\_forks}(1): \{ P(f[2]); P(f[1]) \}
     \]

- Both violate concurrency requirement:
  - While \( P(1) \) is eating the others could be blocked in a chain.

(Exercise: Construct a sequence of requests/releases where this happens.)
Dining Philosophers

Solution that avoids deadlock and provides concurrency:

- Divide philosophers into two groups
  - Odd-numbered philosophers (1,3,5) grab left fork first
  - Even-numbered philosophers (2,4) grab right fork first
Elevator Algorithm

• Loosely simulates an elevator
• Same algorithm can be used for disk scheduling
• Organization of elevator
  – $n$ floors
  – Inside elevator, one button for each floor
  – At each floor, outside the door, there is a single (!) call button
• Elevator scheduling policy
  – When elevator is moving up, it services all requests at or above current position; then it reverses direction
  – When elevator is moving down, it services all requests at or below current position; then it reverses direction
• We will present a monitor that governs the motion according to these scheduling rules
Elevator Algorithm

- Two monitor calls
  - \texttt{request}(i): called when a stop at floor \textit{i} is requested, either by pushing call button at floor \textit{i} or by pushing button \textit{i} inside the elevator.
  - \texttt{release}(): called when elevator door closes

- Usage:
  - Process representing users call \texttt{request}(i)
  - Elevator process (or hardware) calls \texttt{release}()

- Two condition variables (\texttt{upsweep, downsweep})

- Boolean \textit{busy} indicates that either
  - the door is open or
  - the elevator is moving to a new floor.
Elevator algorithm

• When call arrives for floor \texttt{dest} and elevator is currently at floor \texttt{position}
  – If elevator is busy
    • If \texttt{position} < \texttt{dest} wait in upsweep queue
    • If \texttt{position} > \texttt{dest} wait in downsweep queue
    • If \texttt{position} == \texttt{dest} wait in upsweep or downsweep queue, depending on current direction
  – Otherwise, no wait is necessary
• On return from wait (i.e., when corresponding signal is received), or if no wait was necessary, service the request
  – set \texttt{busy} = 1
  – move to the requested floor (\texttt{dest})
Elevator algorithm

Monitor elevator {
    int direction = 1, up = 1, down = 0,
    position = 1, busy = 0;
    condition upsweep, downsweep;

    request(int dest) {
        if (busy) {
            if (position < dest) ||
                ( (position == dest) &&
                (direction == up) )
            upsweep.wait(dest);
        else
            downsweep.wait(-dest);
        } else
            downsweep.wait(-dest); // Called when door closes

        busy = 1;
        position = dest;
    }

    //Called when door closes
    release() {
        busy = 0;
        if (direction==up)
            if (!empty(upsweep))
                upsweep.signal;
        else {
            direction = down;
            downsweep.signal;
        } else /*direction==down*/
            if (!empty(downsweep))
                downsweep.signal;
        else {
            direction = up;
            upsweep.signal;
        }
    }
}
Logical Clocks

- Many applications need to *time-stamp* events for debugging, recovery, distributed mutual exclusion, ordering of broadcast messages, transactions, etc.
- In a *centralized* system, can attach a clock value:
  - \( C(e1) < C(e2) \) means \( e1 \) happened before \( e2 \)
- Physical clocks in *distributed* systems are skewed. This can cause anomalies…
Skewed Physical Clocks

Based on times, the log shows an impossible sequence:
\[ e3, e1, e2, e4 \]

Message arrived before it was sent!!

Possible sequences:
\[ e1, e3, e2, e4 \quad \text{or} \quad e1, e2, e3, e4 \]
Logical Clocks

• Solution: time-stamp events using *counters* as *logical clocks*:

1. Within a process $p$, increment counter for each new event:
   \[ L_p(e_{i+1}) = L_p(e_i) + 1 \]

2. Label each *send* event with new clock value:
   \[ L_p(e_s) = L_p(e_i) + 1 \]

3. Label each *receive* event with new clock value based on maximum of local clock value and label of corresponding *send* event:
   \[ L_q(e_r) = \max( L_p(e_s), L_q(e_i) ) + 1 \]
Logical Clocks

- Logical Clocks yield a distributed *happened-before* relation:
  - $e_i \rightarrow e_k$ holds if
    - $e_i$ and $e_k$ belong to the same process and $e_i$ happened before $e_k$, or
    - $e_i$ is a *send* and $e_k$ is the corresponding *receive*
Logical Clocks

$L_{p1}(u) = 4$

$L_{p2}(v) = \max(4, 1) + 1 = 5$

$L_{p3}(x) = \max(6, 12) + 1 = 13$

$L_{p2}(y) = \max(7, 14) + 1 = 15$

Figure 3-8
History

• Originally developed by Steve Franklin
• Modified by Michael Dillencourt, Summer, 2007
• Modified by Michael Dillencourt, Spring, 2009
• Modified by Michael Dillencourt, Winter, 2010
• Modified by Michael Dillencourt, Summer, 2012