Higher-Level Synchronization and Communication

Shared memory
distributed S&C
Classic synch problems

Monitor Implementation

1. Execution of procedures must be mutually exclusive.
2. A wait must block the current process on the corresponding condition.
3. When a process exits or is blocked on a condition and there are processes waiting to enter or reenter the monitor, one must be selected. If there is a process suspended as the result of executing a signal operation, then it is selected; otherwise, one of the processes from the initial queue of entering processes is selected (processes blocked on conditions remain suspended).
4. A signal must determine if any process is waiting on the corresponding condition. If this is the case, the current process is suspended and one of these waiting processes is reactivated (say, using a first-in/first-out discipline); otherwise, the current process continues.
Monitor solution for bounded buffer

```pascal
buffer: monitor;

type buf_storage = array[0 .. n - 1] of char;
var Buf: buf_storage; nextin, nextout, fullcnt: integer;
    notempty, notfull: condition;

procedure deposit(data: char);
begin
    if full_cnt = n then notfull.wait
    Buf[nextin] := data;
    nextin := nextin + n 1;
    full_cnt := full_cnt + 1;
    notempty.signal
end;

procedure remove(data: char);
begin
    if full_cnt = 0 then notempty.wait
    data := Buf[nextout];
    nextout := nextout + n 1;
    full_cnt := full_cnt - 1;
    notfull.signal
end;

begin
    full_cnt := nextin := nextout := 0 {initialization }
end (buffer)
```

Condition Variables

A special type of variable: c

- **c.wait**: Calling process is blocked and put on a waiting queue;
- **c.signal**: Calling process is suspended and put on a waiting queue; a waiting process is allowed to execute.
- **c.notify**: Calling process is not suspended and a waiting process is allowed to execute when this one blocks.
**Condition Variables**

The condition \( c \) is just a name for an expression that is dynamically evaluated.

\[ \text{If (x<0)} \]
\[ xIsPositive.wait \]

\[ X := 5; \]
\[ xIsPositive.signal \]

---

**Producer-Consumer - Bounded Buffer**

```cpp
monitor BoundedBuffer {  //>
    char buffer[n];
    int nextin = nextout = fullCnt = 0;
    condition notEmpty, notFull;

    deposit (char c) {
        if (fullCnt==n)
            notFull.wait;
        buffer[nextin] = c;
        inc(nextin);
        fullCnt++;
        notEmpty.signal;
    }

    remove (char c) {
        if (fullCnt==n)
            notEmpty.wait;
        c = buffer[nextout];
        inc(nextout);
        fullCnt--;
        notFull.signal;
    }
}  //< BoundedBuffer
```
Specializations - Extensions

\texttt{c.wait}(p)  \quad \text{The priority } p \text{ determines which waiting process will continue.}  
\text{(highest priority = lowest value ?)}

\texttt{c.notify}  \quad \text{A waiting process is released and put on the ReadyList; by the time it starts execution the condition might be FALSE.}

\texttt{if(!B1)c1.wait; if(!B2)c2.wait;}
\texttt{c1.notify; c2.notify;}
\texttt{B2 = FALSE;}

Specializations - Extensions

\texttt{if(!B) c.wait; \quad \text{---=} \quad \text{while(!B) c.wait;}}

\texttt{c.notify}  \quad \text{unicast / multicast / broadcast versions}

\text{Example languages: Mesa, Java}

\texttt{timeout}  \quad \text{Unblock the task by the specified time if it has not been "notified".}
Monitor Implementation

The body of each procedure inside the monitor is surrounded by entry and exit code as follows:

\[
P(\text{mutex});
\]

procedure body

\[
\text{if } \text{urgentcnt} > 0 \text{ then } V(\text{urgent}) \text{ else } V(\text{mutex});
\]

Each c. wait within the procedure body is coded as

\[
\text{condcnt} := \text{condcnt} + 1;
\]

\[
\text{if } \text{urgentcnt} > 0 \text{ then } V(\text{urgent}) \text{ else } V(\text{mutex});
\]

\[
P(\text{condsem}); \{ \text{process waits here} \}
\]

\[
\text{condcnt} := \text{condcnt} – 1;
\]

Each c. signal within the procedure body is implemented as

\[
\text{urgentcnt} := \text{urgentcnt} + 1;
\]

\[
\text{if } \text{condcnt} > 0 \text{ then } \text{begin} \text{ } V(\text{condsem}); P(\text{urgent}) \text{ end};
\]

\[
\text{urgentcnt} := \text{urgentcnt} - 1
\]

Monitors - Protected Types

Typical usage pattern

[guard]wait

\[
\text{if } (B) \text{ c.wait}
\]

\[
c.signal/ c.notify
\]

Combine entry/wait with exit/signal.
Protected Types

- Encapsulated object with public entry points.

```java
when (B) // (wait)

Execute code when the Boolean guard evaluates to TRUE.

// (signal)
}
```

Protected Types - Bounded Buffer

```java
protected boundedBuffer{
    entry deposit(char data);
    entry remove(char data);
    private
    char buffer[n];
    int nextin=nextout=0;
    int fullCnt=0;
}
```

Declaration of the interface (specification)
**Protected Types - Bounded Buffer**

protected boundedBuffer {

    entry deposit(char c)
    when (fullCnt < n) {
        buffer[nextin] = c;
        inc (nextin);
        fullCnt++;
    }

    entry remove(char c)
    when (fullCnt > 0) {
        c = buffer[nextout];
        inc (nextout);
        fullCnt--;
    }

} // boundedBuffer  of the body.

**Message-based IPC**

All previously presented solutions are designed for shared-memory systems. Distributed systems and multiprocessor systems with local memory require a different approach.

Direct communication:

\[ \text{send}( \ p, \ msg \ ); \quad \text{receive}( \ q, \ msg \ ); \]

\( p, \ q \) - process identifiers

Indirect communication:

\[ \text{send}( \ mbx, \ msg \ ); \quad \text{receive}( \ mbx, \ msg \ ); \]

\( mbx \) - channel (mailbox) identifier
Distributed Synchronization and Communication Mechanisms

shared memory

common channel

Semantics of send/receive Primitives

<table>
<thead>
<tr>
<th>send</th>
<th>synchronous /blocking</th>
<th>asynchronous /non-blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>explicit</strong></td>
<td><strong>send</strong></td>
<td><strong>send</strong></td>
</tr>
<tr>
<td><strong>naming</strong></td>
<td><strong>message</strong> m to</td>
<td><strong>message</strong> m to</td>
</tr>
<tr>
<td></td>
<td><strong>receiver</strong> r - wait</td>
<td><strong>receiver</strong> r - and</td>
</tr>
<tr>
<td></td>
<td><strong>until message</strong> is</td>
<td><strong>and continue</strong></td>
</tr>
<tr>
<td></td>
<td><strong>accepted</strong></td>
<td></td>
</tr>
</tbody>
</table>

| **implicit**  | **send**               | **broadcast**               |
| **naming**    | **message** m -        | **message** m -             |
|               | **wait until message** | **and continue**            |
|               | **is accepted**        |                             |
### Semantics of send/receive Primitives

<table>
<thead>
<tr>
<th>receive</th>
<th>synchronous /blocking</th>
<th>asynchronous /non-blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>explicit naming</strong></td>
<td>wait for message from sender ( s )</td>
<td>if there is a message from sender ( s ) accept it – otherwise just go on.</td>
</tr>
<tr>
<td><strong>implicit naming</strong></td>
<td>wait for message from any sender</td>
<td>if there is a message from any sender then accept it – otherwise proceed.</td>
</tr>
</tbody>
</table>

### CSP – the "Rendezvous"

CSP - communicating sequential processes (Hoare)

```
send(ch, m);  receive(ch, m);
```
Mailboxes and Ports

Indirect communication:

\[ \text{send}( \text{mbx}, \text{msg} ); \quad \text{receive}( \text{mbx}, \text{msg} ); \]

\( \text{mbx} \) - channel (mailbox) identifier

A queue which accepts messages from many and delivers to many processes.
A queue which accepts messages from many and delivers to only one processes.

**Process State Vector**

**Process Management Info**
- Registers, Program Counter
- PSW, stack pointer
- Process state, priority
- Scheduling parameters
- Proc_id, parent Proc_id
- Process group, proc.start time
- CPU time used, next alarm time
- children’s CPU time used

**Memory Management Info**
- Pointer to next segment
- Pointer to data segment
- Pointer to stack segment

**File Management Info**
- Root directory
- Working directory
- File descriptors
- User_id, Group_id


**Process and Thread State**

<table>
<thead>
<tr>
<th>Per Process Info</th>
<th>Per Thread Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>address space</td>
<td>program counter – PC</td>
</tr>
<tr>
<td>global variables</td>
<td>registers</td>
</tr>
<tr>
<td>open files</td>
<td>stack</td>
</tr>
<tr>
<td>child processes</td>
<td>state</td>
</tr>
<tr>
<td>pending alarm timeouts</td>
<td></td>
</tr>
<tr>
<td>signals and signal handlers</td>
<td></td>
</tr>
<tr>
<td>accounting information</td>
<td></td>
</tr>
</tbody>
</table>

**Context Switch**

- Hardware (interrupt, SVC instruction) stacks PC and additional state info
- Hardware loads PC with new address from interrupt vector
- Handler routine (assembly code) saves registers and sets up new stack
- Interrupt service routine takes over (e.g. fills and moves buffers etc.)
- Control is passed (1) back directly (dispatcher -> RTI) or (2) on to the scheduler
- [2] Scheduler decides which process to run next
- Control is passed to the Dispatcher
- Dispatcher starts up the [new] process after restoring state vector
Process Termination

Conditions which terminate processes
1. Normal exit (voluntary)
2. Error exit (voluntary)
3. Fatal error (involuntary)
4. Killed by another process (involuntary)

Process Hierarchies

- Parent creates a child process, child processes can create its own process
- Forms a hierarchy
  - UNIX calls this a "process group"
- Windows has no concept of process hierarchy
  - all processes are created equal
Process States (1)

- Possible process states
  - running
  - blocked
  - ready
- Transitions between states shown

1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Process States (2)

- Lowest layer of process-structured OS
  - handles interrupts, scheduling
- Above that layer are sequential processes
Message-based IPC

Direct communication:

\begin{align*}
\text{send}( p, \text{msg} ); & \quad \text{receive} \ ( q, \text{msg} ); \\
p, q & - \text{process identifiers} \\
\end{align*}

Frequently the transmission of a message involves kernel functions (context switching) and copying of the buffer.

Some Fundamental Questions

1. When a message is emitted, does the sending process have to wait until the message has been accepted by the receiver, or can it continue processing?

2. What should happen when a receive is issued and there is no message waiting?

3. Does the sender have to specify exactly one receiver to which it wishes to transmit a message, or can messages be accepted by any of a group of receivers?

4. Does the receiver have to specify exactly one sender from which it wishes to accept a message, or can it accept messages arriving from different senders?
### Types of send/receive Primitives

<table>
<thead>
<tr>
<th>send</th>
<th>blocking</th>
<th>non-blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicit naming</td>
<td>Send message m to receiver r; wait until message is accepted.</td>
<td>Send message m to receiver r.</td>
</tr>
<tr>
<td>Implicit naming</td>
<td>Broadcast message; wait until message is accepted.</td>
<td>Broadcast message.</td>
</tr>
</tbody>
</table>

### Receive

| Explicit naming | Wait for message from sender s. | If there is a message from s receive it; else proceed. |
| Implicit naming  | Wait for message from any sender. | If there is a message from any sender receive it; else proceed. |

### Producer/Consumer with Messages

```plaintext
process producer;
  loop
    produce_data;
    send( buffer_manager, data);
  end {loop};
process consumer;
  loop
    send( buffer_manager, request);
    receive(buffer_manager, data);
    consumedata;
  end {loop};
process buffetimanager;
```


**Producer/Consumer with Messages**

```plaintext
process buffer_manager;
loop
  if full_cnt == 0 then
    receive(producer, data);
    Buf[nextin] := data;
    nextin := nextin + 1; fullcnt := fullcnt + 1;
  elseif full_cnt == n then
    data := Buf[nextout];
    receive(consumer, request);
    send(consumer, data);
    nextout := nextout + 1; fullcnt := fullcnt - 1;
  else
    receive(producer, data);
    Buf[nextin] := data;
    nextin := nextin + 1; fullcnt := fullcnt + 1;
  end
end (loop)
```

**Remote Procedure Call**

Implementation of RPC

```plaintext
process RP_guard;
loop
  receive(caller, parameters);
  body of RP;
  send(caller, results);
end (loop);
```

Process containing RP
Rendezvous

Execution of Rendezvous

calling process is delayed

called process is delayed

Select Statement

select
  [when B_1 :
    accept E_1(\ldots) do S_1 end
  or
  [when B_2 :
    accept E_2(\ldots) do S_2 end
  or
  . . . . . . .
  [when B_n :
    accept E_n(\ldots) do S_n end
  [else] R
end {select}
Select Statement - Bounded Buffer

process buffer;
select
  when full_cnt < n:
    accept deposit(data: char) do
      Buf[nextin] := data;
      nextin := nextin + n 1;
      full_cnt := full_cnt + 1;
    end
  or
  when full_cnt > 0:
    accept remove(data: char) do
      data := Buf[nextout];
      nextout := nextout + n 1;
      full_cnt := full_cnt - 1;
    end
end {select}

Classic Problems

• Readers/ Writers
• Dining Philosophers (buffer allocation)
• Disk head scheduling

• read the chapters in the book/script!!
Semaphores

```c
#define N 100
typedef int semaphore;
semaphore max = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void) {
    int item;
    while (TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}

void consumer(void) {
    int item;
    while (TRUE) {
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

The producer-consumer problem using semaphores

---

Atomic Instruction

![Diagram](fetch decode opnd-fetch execute opnd-store)

uninterruptible sequence of mikro instructions

- `Test_and_Set, x`
- `Swap(x, y)`
- `R_Ad_One(s)`
### The Thread Model

<table>
<thead>
<tr>
<th>Per process items</th>
<th>Per thread items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address space</td>
<td>Program counter</td>
</tr>
<tr>
<td>Global variables</td>
<td>Registers</td>
</tr>
<tr>
<td>Open files</td>
<td>Stack</td>
</tr>
<tr>
<td>Child processes</td>
<td>State</td>
</tr>
<tr>
<td>Pending alarms</td>
<td></td>
</tr>
<tr>
<td>Signals and signal handlers</td>
<td></td>
</tr>
<tr>
<td>Accounting information</td>
<td></td>
</tr>
</tbody>
</table>

- Items shared by all threads in a process
- Items private to each thread

---

### Dining Philosophers

- Philosophers eat/think
- Eating needs 2 forks
- Pick one fork at a time
- How to prevent deadlock
**Dining Philosophers**

```c
#define N 5          /* number of philosophers */
void philosopher(int i)          /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think();  /* philosopher is thinking */
        take_fork(i);    /* take left fork */
        take_fork((i+1) % N); /* take right fork; % is modulo operator */
        eat();       /* yum-yum, spaghetti */
        put_fork(i);  /* put left fork back on the table */
        put_fork((i+1) % N); /* put right fork back on the table */
    }
}
```

**A nonexistent solution to the dining philosophers problem**

---

**Dining Philosophers**

```c
#define N 5          /* number of philosophers */
#define LEFT (i+N-1)%N /* number of i's left neighbor */
#define RIGHT (i+1)%N /* number of i's right neighbor */
#define THINKING 0    /* philosopher is thinking */
#define HUNGRY 1      /* philosopher is trying to get forks */
#define EATING 2      /* philosopher is eating */
typedef int semaphore;
int state[N];    /* array to keep track of everyone's state */
semaphore mutex = 1; /* mutual exclusion for critical regions */
semaphore s[N];  /* one semaphore per philosopher */
void philosopher(int i)          /* i: philosopher number, from 0 to N-1 */
{
    while (TRUE) {
        think();  /* repeat forever */
        take_forks(i);    /* philosopher is thinking */
        eat();       /* yum-yum, spaghetti */
        put_forks();  /* put both forks back on table */
    }
}
```

**Solution to dining philosophers problem (part 1)**
Dining Philosophers

void take_forks(int i) {
    down(&mutex); // enter critical region
    state[i] = HUNGRY;
    test(i);
    up(&mutex); // exit critical region
    down(&mutex); // block if forks are not acquired
}

void put_forks(i) {
    down(&mutex); // enter critical region
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    up(&mutex);
}

void test(i) {
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&mutex);
    }
}

Solution to dining philosophers problem (part 2)