

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

ICS 143

1

Motivation

- semaphores and events are:
 - powerful but low-level operations
 - programming is highly error prone
 - programs are difficult to design, debug, and maintain
 - not usable in distributed memory systems
- need higher-level primitives
 - based on semaphores or messages

ICS 143

2

Shared Memory Methods

- Monitors
 - follow principles of abstract data type (object-oriented programming):
 - any data type is manipulated only by a set of predefined operations
 - monitor is
 - collection data (resource), together with
 - functions that manipulate the data

ICS 143

3

Monitors

- implementation must guarantee:
 - resource accessible *only* by monitor functions
 - monitor functions 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 c queue

ICS 143

4

Monitors

- note that c is not a *conventional* variable
 - c has no value
 - c is an arbitrary name chosen by programmer to designate an event, state, or condition
 - each c has a waiting queue associated
 - process may block itself on c -- it waits until another process issues a signal on c

ICS 143

5

Hoare Monitors

- after c.signal there are 2 ready processes:
 - the calling process
 - the process woken up c.signal
- which should continue?
(only one can be executing inside the monitor!)
- Hoare monitor:
 - woken-up process continues
 - calling process is placed on high-priority queue

Figure 3-2

ICS 143

6

Bounded buffer problem

```
monitor Bounded_Buffer {
  char buffer[n];
  int nextin=0, nextout=0, full_cnt=0;
  condition notempty, notfull;

  deposit(char data) {
    ...
  }

  remove(char data) {
    ...
  }
}
```

ICS 143

7

Bounded buffer problem

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

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

ICS 143

8

Priority waits

c.wait(p)

- p is an integer (priority)
- blocked processes are kept sorted by p

c.signal

- wakes up process with lowest p

ICS 143

9

Example: alarm clock

```
monitor Alarm_Clock {
  int now=0;
  condition wakeup;
wakeme(int n) {
  int alarm;
  alarm = now + n;
  while (now<alarm) wakeup.wait(alarm);
  wakeup.signal;
}
tick() {
  now = now + 1;
  wakeup.signal;
}
```

ICS 143

10

Example: alarm clock

- only one process wakes up at each tick
- without priority waits, all processes would need to wake up to check their alarm settings

ICS 143

11

Mesa and Java monitors

- after c.signal:
 - calling process continues
 - woken-up process continues when caller exits
- problem
 - caller may wake up multiple processes, pi, pj ...
 - pi could change condition on which pj was blocked

ICS 143

12

Mesa and Java monitors

- solution
 - instead of: `if (!condition) c.wait`
 - use: `while (!condition) c.wait`
- signal is sometimes called “notify”

ICS 143

13

Protected types

- special case of monitor where:
 - `c.wait` is the *first* operation of a function
 - `c.signal` is the *last* operation
- typical in producer/consumer situations
- `wait/signal` are combined into a *when* clause
 - *when* `c` forms a “barrier”:
 - function may proceed only when `c` is true

ICS 143

14

Example

```
entry deposit(char m)
  when full_cnt < n {
    buffer[nextin] = m;
    nextin = (nextin + 1) % n;
    full_cnt = full_cnt + 1;
  }

entry remove(char m)
  when full_cnt > 0 {
    m = buffer[nextout];
    nextout = (nextout + 1) % n;
    full_cnt = full_cnt - 1;
  }
```

ICS 143

15

Distributed Synchronization

- semaphore-based primitive require 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

ICS 143

16

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?

ICS 143

17

Types of send/receive

send	blocking	nonblocking
explicit naming	send m to r wait until accepted	send m to r
implicit naming	broadcast m wait until accepted	broadcast m
receive	blocking	nonblocking
explicit naming	wait for message from 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

ICS 143

18

Channels, ports, and mailboxes

- allow indirect communication:
 - senders/receivers name channel 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

ICS 143

19

Named message channels

- CSP (Communicating Sequential Processes)
 - named channel, $ch1$, connects processes $p1, p2$
 - $p1$ sends to $p2$ using: `send($ch1$, "a")`
 - $p2$ receives from $p1$ using: `receive($ch1$, x)`
 - guarded commands:
 - *when* (c) s
 - set of statements s executed only when c is true
 - allows to receive messages selectively based on arbitrary conditions

ICS 143

20

Bounded buffer with CSP

- producer P , consumer C , and buffer B are communicating processes
- problem:
 - 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 from P and C are guarded with *when*

ICS 143

21

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)`

ICS 143

22

Bounded buffer with CSP

```
process Bounded_Buffer {  
  ...  
  while (1) {  
    when ((full_cnt < n) &&  
          receive(deposit, buf[nextin])) {  
      nextin = (nextin + 1) % n;  
      full_cnt = full_cnt + 1;  
    }  
    or  
    when ((full_cnt > 0) && receive(req, )) {  
      send(remove, buf[nextout]);  
      nextout = (nextout + 1) % n;  
      full_cnt = full_cnt - 1;  
    }  
  }  
}
```

ICS 143

23

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

Figure 3-1

ICS 143

24

Procedure-based communication

- send/receive are too low level (like P/V)
- typical interactions:
 - send request -- receive result
 - make this into a single higher-level primitive
- use RPC or rendezvous
 - caller invokes procedure p on remote machine
 - remote machine performs operation and returns result
 - similar to regular procedure call but parameters cannot contain pointers -- caller and server do not share any memory

ICS 143

25

RPC

- caller issues: $res = f(params)$
- this is translated into:

```
Calling Process:      Server Process:
...
send(RP, f, params);  process RP_server {
rec(RP, res);         while (1) {
...                   rec(C, f, params);
                      res=f(params);
                      send(C, res);
                      }
                      }
```

ICS 143

26

Rendezvous

- with RPC: p is part of a dedicated server
- rendezvous:
 - p is part of an arbitrary process
 - maintains state between calls
 - may accept/delay/reject call
 - is symmetrical: any process may be a client or a server

ICS 143

27

Rendezvous

- caller: similar syntax/semantics to RPC
`q.f(param)`
where `q` is the called process (server)
- server: must indicate willingness to accept:
`accept f(param) S`
- rendezvous:

Figure 3-3

ICS 143

28

Rendezvous

- to permit selective receive:

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

ICS 143

29

Example: bounded buffer

```
process Bounded_Buffer {
  while(1) {
    select {
      when (full_cnt < n):
        accept deposit(char c) {
          buffer[nextin] = c;
          nextin = (nextin + 1) % n;
          full_cnt = full_cnt + 1;
        }
      or
      when (full_cnt > 0):
        accept remove(char c) {
          c = buffer[nextout];
          nextout = (nextout + 1) % n;
          full_cnt = full_cnt - 1;
        }
    }
  }
}
```

ICS 143

30

Distributed Mutual Exclusion

- CS problem in a distributed environment
- central controller solution
 - requesting process sends request to controller
 - controller grant it to one processes at a time
 - problems:
 - single point of failure
 - performance bottleneck
- fully distributed solution
 - processes negotiate access among themselves

ICS 143

31

Distributed Mutual Exclusion

- token ring solution
 - each process has a controller
 - controllers are arranged in a ring
 - controllers pass “token” around ring
 - process whose controller holds token may enter CS

Figure 3-4

ICS 143

32

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;
  }
}
```

ICS 143

33

Readers/Writers Problem

- extension of basic CS problem
- two types of processes entering a CS:
 - only one W may be inside CS, or
 - any number of R may be inside CS
- prevent starvation of either process type
 - if Rs are in CS, a new R must not enter if W is waiting
 - if W is in CS, all Rs waiting should enter (even if they arrived after new Ws)

ICS 143

34

Solution using monitor

```
monitor readers/writers {  
  ...  
  start_read() {  
    if (writing || !empty(OK_W)) OK_R.wait;  
    read_cnt = read_cnt + 1;  
    OK_R.signal; }  
  end_read() {  
    read_cnt = read_cnt - 1;  
    if (read_cnt == 0) OK_W.signal; }  
  start_write() {  
    if ((writing || read_cnt != 0) OK_W.wait;  
    writing = 1; }  
  end_write() {  
    writing = 0;  
    if (!empty(OK_R)) OK_R.signal;  
    else OK_W.signal; } }  
}
```

ICS 143

35

Dining philosophers

figure 3-5

- 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

ICS 143

36

Dining philosophers

```
p(i) {
  while (1) {
    think(i);
    grab_forks(i);
    eat(i);
    return_forks(i);
  }

  grab_forks(i): P(f[i]); P(f[(i+1)%5])
  return_forks(i): V(f[i]); V(f[(i+1)%5])
}
```

- only one process can grab any fork
- may lead to deadlock

ICS 143

37

Dining philosophers

- solutions to deadlock
 - use a counter: at most $n-1$ philosophers may attempt to grab forks
 - one philosopher requests forks in reverse order, e.g.:
`grab_forks(1): P(f[2]); P(f[1])`
 - violates concurrency requirement: while P(1) is eating, all other could be blocked in a chain
 - divide philosophers into two groups; odd grab left fork first, even grab right fork first

ICS 143

38

Elevator algorithm

Figure 3-6

- pressing button at floor i or button i inside elevator invokes: `request(i)`
- when door closes, invoke: `release()`
- scheduler 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

ICS 143

39

Elevator algorithm

```
monitor elevator {
  ...
  request(int dest) {
    if (busy) {
      if ((position < dest) ||
          ((position == dest) && (dir == up)))
        upsweep.wait(dest);
      else downsweep.wait(-dest);
    }
    busy = 1;
    position = dest;
  }
  • if position < dest: wait in upsweep queue
  • if position > dest: wait in downsweep queue
  • if position == dest: service immediately--wait in upsweep or
    downsweep, depending on current direction
}
```

ICS 143

40

Elevator algorithm

```
release() {
  busy = 0;
  if (direction == up) {
    if (!empty(upsweep)) upsweep.signal;
    else {
      direction = down;
      downsweep.signal;
    }
  } else if /*direction == down*/
    (!empty(downsweep)) downsweep.signal;
  else {
    direction = up;
    upsweep.signal;
  }
}
```

ICS 143

41

Logical clocks

- many applications need to **time-stamp** events
(e.g., for debugging, recovery, distributed mutual exclusion, ordering of broadcast messages, transactions)
- **centralized** system: attach clock value:
 $C(e1) < C(e2)$ means $e1$ happened before $e2$
- clocks in **distributed** systems are skewed
- example:

Figure 3-7

- log shows: $e3, e1, e2, e4$ (impossible sequence)
- possible: $e1, e3, e2, e4$, or $e1, e2, e3, e4$

ICS 143

42

Logical clocks

- use counters as **logical clocks** instead
 - **within** a process p , increment counter for each new event:

$$L_p(e_{i+1}) = L_p(e_i) + 1$$

- label each **send** event with new clock value:

$$L_p(e_s) = L_p(e_i) + 1$$

- label each **receive** event with *maximum* of local clock value and value of message:

$$L_q(e_r) = \max(L_p(e_s), L_q(e_i)) + 1$$

ICS 143

43

Logical clocks

- the scheme follows **happened-before** relation: $e_i \rightarrow e_j$ holds if:
 - e_i and e_j belong to the same process and e_i happened before e_j
 - e_i is a send and e_j is the corresponding receive
- example:

Figure 3-8

ICS 143

44
