Interprocess Synchronization and Communication

Monitors and Priority Waits

- Monitor signals restart processes in FIFO ordering
- Sometimes a priority ordering is better
  - For example, waiting for a specific time of day
- Syntax: `c.wait(p)`
  - *p*: integer priority
  - *c*: condition variable
- If more than 1 process waiting on *c*, then if another process sends *c*.signal, the one with the lowest *p* would be unblocked

Example: Alarm Clock

- Process calls *alarmclock*.*wakeme*(*n*) and suspends for *n* seconds
- We're assuming the hardware invokes procedure *tick* to update the clock every second

```
alarmclock: monitor
   now: integer;
   wakeup: condition;
```

Example: Alarm Clock

```
    procedure wakeme(n: integer)
    begin
        alarmsetting := now + n;
        while now < alarmsetting do
                wakeup.wait(alarmsetting);
        wakeup.signal;
    end;
    procedure tick
    begin
        now := now + 1;
        wakeup.signal;
    end
end.
```

Eventcounters and Sequencers

- Provide synchronization without mutual exclusion
  - Can provide mutual exclusion, but need not

Eventcounters

- Eventcounter *e* is a non-decreasing integer initially 0

- `advance(e): e := e + 1`
  - This is atomic, indicates an event of interest occurred
- `read(e): return(e)`
  - So if *e* is *n*, at lease *n* advance(*e*)s have occurred
- `await(e, v):` block until *e* has value *v*
  - So continues only after at least *v* advance(*e*)s have occurred

Sequencers

- Sequencer *e* is an increasing integer initially 0

- `ticket(s):  olds := s; s := s + 1; return(olds);`
  - This is atomic, and requires mutual exclusion so no 2 calls return same value

- Enables mutual exclusion:

```
await(e, ticket(s));
. . .
advance(e);
```

Producer-Consumer Solution

```
var nextp, nextc: item;
    IN, OUT: eventcounter;
    T: sequencer;
```

- IN, OUT synchronize the producers and consumers so at most 1 accesses the buffer

Producer-Consumer Solution

```
procedure producer;
begin
      var t: integer;
      while true do begin
            (* produce item in nextp *)
            t := ticket(T);
            await(IN, t);
            await(OUT, t - N + 1);
            buffer[(t + 1) mod N] := nextp;
            advance(IN);
      end;
end;
```

Producer-Consumer Solution

```
procedure consumer;
begin
      var i: integer;
      i := 1;
      while true do begin
            await(IN, i);
            nextc := buffer[i mod N];
            (* consume item in nextc *)
            advance(OUT);
            i := i + 1;
      end;
end;
```

Producer-Consumer Solution

```
parbegin
     producer;
     consumer;
parend
```

Analysis

- For producer: first producer grabs a ticket; it will be 0
  - IN is also 0, so you pass first await() — this orders the producers trying to get in so at most 1 will proceed
  - OUT is 0, so you pass second await() — this checks that there are open slots in the buffer
  - Add item to buffer
  - Increment eventcounter IN — this releases any producers waiting to proceed past the first await()

Analysis

- For producer: first producer grabs a ticket; it will be 0
  - IN is also 0, so you pass first await() — this orders the producers trying to get in so at most 1 will proceed
  - OUT is 0, so you pass second await() — this checks that there are open slots in the buffer
  - Add item to buffer
  - Increment eventcounter IN — this releases any producers waiting to proceed past the first await()

Analysis

- For consumer: note *i* is initially set to 1
  - IN is more than 1 only if 1 producer has advanced IN, which happens only when a producer inserts an item into the buffer
  - The consumer extracts the item from the buffer
  - It then advances OUT to indicate there is an empty slot in the buffer
  - Increment *i* by 1 to advance to the next slot in the buffer

Analysis

- Mutual exclusion: the key is that ticket(T) is atomic and always issues the next number, so no two ts will have the same value

  Given that, as IN is incremented only when a producer exits the critical section, at most 1 producer can be in the critical section

  Note consumer also waits on IN, until its value is no less than the temporary variable in the for loop; as IN can only have 1 value, if consumer is in, producer blocks at await(IN, t) and vice versa

Analysis

- Progress: only processes in the entry or exit sections control advancing IN, OUT and issuing sequence numbers
- Bounded wait: for producers, as tickets are increasing in value, they enter in the order they received their ticket, and at most have to wait until the number of processes that is the value of the ticket enter (and, usually, considerably fewer)

  For consumer, they enter when IN is equal to the number *i*; there is at most 1 instance of this, and so at most IN consumers and producers could have entered.

Shared Memory Synchronization

- Sometimes none of the above mechanisms are satisfactory:
  - Security considerations may prevent sharing memory

    Example: each process must run in strict isolation, in its own logical space with all interactions under its own control; this is not possible with monitor, as any process with access to the monitor can get global data stored within the monitor.
  - It may not be possible to share memory

    Example: in a distributed system, each processor may have its own local memory and so processes on different processors cannot share data.
- So need a mechanism other than those based on shared memory
  - New schemes are called *message-based synchronization schemes*.

Interprocess Communication (IPC)

- Two primitive operations in all such systems:
- `send(`*`p, msg`*`)`
  - Transfers message *msg* to process *p*
  - Special (implementation-dependent) values of *p* can be used to indicate that the message goes to all processes
  - Called *broadcast*.
- `receive(`*`q, msg`*`)`
  - Obtains message *msg* from process *q*
  - Special (implementation-dependent) values of *q* can be used to indicate that the message can come from any process

Characterizations

- Four basic properties
  - Does the sender wait until its message is accepted by the recipient, or does it continue processing?
  - What happens when a receive call is issued, but there is no message waiting?
  - Must the sender specify *exactly* 1 recipient, or can messages be sent to any (or all) of a number of recipients?
  - Must the recipient specify *exactly* 1 sender, or can messages be accepted from any (or all) of a number of senders?

Sending Message

- Does the sender wait until its message is accepted by the recipient, or does it continue processing?
- If the sender blocks, the send is called *blocking* or *synchronous*
- If the sender may proceed while the message is being delivered, send is *non-blocking* or *asynchronous*

Receive But No Message

- What happens when a receive call is issued, but there is no message waiting?
  - If the process waits for a message to arrive, the receive is called *blocking* or *synchronous*
  - If the process continues, the receive is called *nonblocking* or *asynchronous*

Related: Size of Queue

Queue associated with connection or link between the two processes, has a *capacity* for a certain number of messages

- *Zero capacity*: link cannot have any messages waiting; sender must wait until recipient gets message or message will be lost; most useful when sending from buffer in the process
  - Sometimes called *rendezvous*
- *Bounded capacity*: limits the number of messages that can be on the queue; if full, sender must wait or message will be lost
- *Unbounded capacity*: all messages can be stored in the queue

Explicit Naming

- Also called *direct communication*
- Link is established automatically; processes need each others' identity
- Link associated with exactly 2 processes (sender, receiver)
- At most 1 link between the 2 processes
- Link is bidirectional
- Variant: sender specifies recipient, but recipient will accept messages from any sender; when receive function returns, it also returns the name of the sending process
- Problem: if a process changes its name, all references to it have to be changed

Example: Producer Consumer Problem

```
procedure producer;
begin
	while true do begin
		// produce a nextp
		send(“Consumerprocess”, nextp);
	end;
end;
procedure consumer;
begin
	while true do begin
		receive(“Consumerprocess”, nextc);
		// consume nextc
	end;
end;
```

Implicit Naming

- Also called in*direct communication*
- Messages go to a mailbox or drop box
- Link exists only if they share something like a mailbox
- Link associated with any number of processes
- Any number of links can exit between processes
- Links can be unidirectional or bidirectional

Summary Chart

send	blocking	non-blocking
explicit naming	send message to receiver; wait until message accepted	send message to receiver
implicit naming	broadcast message; wait until all processes accept messages	broadcast message

receive	blocking	non-blocking
explicit naming	wait for message from named sender	if there is a message from the named sender, get it; otherwise, proceed
implicit naming	wait for message from any sender	if there is a message from any snder, get it; otherwise, proceed

Example: Producer Consumer Problem

```
procedure producer;
begin
        while true do begin
                // produce a nextp
                send(“mailbox-pc”, nextp);
        end;
end;
procedure consumer;
begin
        while true do begin
                receive(“mailbox-pc”, nextc);
                // consume nextc
        end;
end;
```

Problems

- 2 processes do a receive on mailbox; who gets message?
  - Each link associated with exactly 2 processes (no problem here)
  - Only 1 processes art a time may do a receive on a particular mailbox (in this case, it's called a *port*)
  - System chooses which process gets message
- Creating a mailbox
  - Process declares mailbox like it declares a variable; process gets all messages put into mailbox, which goes away when process exits
  - Operating system defines mailbox, system calls to send, receive messages on it

Other Issues

- Communications delay; wait until message acknowledged before sending another
  - TCP works this way
- Process terminates before message processed
  - Recipient p_1 waiting for message from terminated process p_2 using a blocking receive: p_1 stays blocked
  - Sender p_1 sends message to terminated process p_2 using a blocking send: p_1 blocks
  - Solution: notify p_1 that p_2 has terminated, or terminate p_1

Other Issues

- Message lost in transfer
  - Operating system responsible for detecting this; notify sender message was not received, or sender retransmits after some time when acknowledgement received
    - TCP works this way
  - Sender responsible for detecting this and (if needed) resend message
    - UDP works this way
- Messages altered in transit
  - Use Message Integrity Codes (MICs) to detect this
  - Example: CRCs, great for accidental changes
  - Use cryptographic hashes for detecting deliberate changes (attacks)