Review:

Interprocess communication

- **Possible problems**
  - **Print spooler**
  - **Bookkeeping**
- **Some important concepts**
  - **Race condition**
  - **Critical section**
  - **Mutual exclusion mechanism**
- **Four conditions for a good solution**
- **Some tentative solutions**
  - **Disable interrupts**
  - **Lock variables**
  - **Strict alteration**

- **Semaphores**

  - **A synchronization integer variable**
  - **Two atomic operations: down and up**
  - **A queue for blocking**
  - **Implementation of semaphores**

```
type  semaphore = record
        value : integer
        l: queue of processes
end;
down(s):  If s.value ≥ 1 then
             s.value = s.value - 1
             else block the process on the
             semaphore queue s.l
             (i.e. add the process to queue s.l)
up(s):  If some processes are blocked on s
          then unblock a process
          (remove a process from queue s.l)
          else s.value = s.value + 1
```

- **Two types of semaphores**

  * **Binary semaphore:**
    two values 0 and 1, used for mutual exclusion (i.e. to ensure that only one process is accessing shared information at a time)

    ```
    semaphore mutex = 1
    down(mutex);
    critical-section();
    up(mutex);
    ```

  * **Counting semaphore:**
    used for synchronizing access to a shared resource by several concurrent processes (i.e. to control how many processes can concurrently perform operations on the shared resource).

- **Semaphores are not supported by hardware, but can be easily implemented using TEST and SET LOCK instruction and enable/disable interrupts.**

- **Solving the producer consumer problem by semaphores:**

- **The sequence of down and up operations matters.**
  - **Reverse the sequence of downs in producer:**
    1. **Buffer empty.**
    2. **Run producer through while loop n times.**
    3. **Buffer full.**
    4. **Run producer again.**
    5. **Producer sleeps on semaphore** *empty*.
    6. **Run consumer.**
    7. **Consumer sleeps on semaphore** *mutex*.
    8. **Deadlock.**

* **Reverse the sequence of downs in consumer:**

  1. **Buffer empty.**
  2. **Run consumer.**
  3. **Consumer sleeps on semaphore *full*.**
  4. **Run producer.**
  5. **Down(empty), ok.**
  6. **Producer sleeps on semaphore *mutex*.**
  7. **Deadlock again.**

- **Although semaphores provide a simple and sufficiently general scheme for IPC, they suffer from the following drawbacks:**

  1. **A process that uses a semaphore has to know WHICH other processes use these semaphore. May also have to know HOW these processes are using the semaphore.**

  2. **Semaphore operations must be carefully installed in a process. The OMISSION of a up or down may result in inconsistencies or deadlocks.**

  3. **Programs using semaphores can be extremely hard to verify for correctness.**

- **Event counter.**
  (A solution without requiring mutual exclusion.

  - **A special kind of variable.**

  - **Three operations on an event counter E:**

    1. **Read(E): return the value of E.**

    2. **Advance(E): increment E by 1.**

    3. **Await(E, v): wait (block) until E has a value of v or more.**

  - **Solving the producer consumer problem by event counter:**

- **Monitors:**
  - **A high-level synchronization primitive.**
  - **Combine three features:**
    1. **Shared data**
    2. **Operations on the data**
    3. **Synchronization**
  - **Programming constructs, implemented by compiler**
  - **Only one process active in a monitor at a time (implicitly controlled by monitor lock)**
  - **Easier and safer to use.**

– **Structure of a monitor:**

<Monitor name>: monitor
begin

Declaration of data local to the monitor

⋮

procedure <name> (<formal parameters>);
begin
procedure body
end;

Declaration of other procedures.

⋮

begin
Initialization of local data of the moni-tor
end;
end;

- **Need some way to wait, two choices:**
  - **Busy-wait inside monitor**
  - **Put process to sleep insider monitor**
- **Condition variables (things to wait on)**
  - **wait(condition): release monitor lock, and put process to sleep. When process wakes up again, re-acquire monitor lock immediately.**
  - **signal(condition): wake up one process waiting on condition variable (FIFO). If no body waiting, do nothing (no history).**
  - **broadcast(condition): wake up all processes waiting on condition variables.**

- **Need to decide who gets the monitor lock after a signal:**
  - **On signal, signaler keeps monitor lock. Awakened process waits for monitor lock with no special priority.**
  - **On signal, awakened process gets the monitor lock. Signaler exits from monitor immediately.**

Review:

Solutions for IPC problems

- **Peterson's solution:**
  Combine lock and take turn. Busy waiting.

- **Test and Set Lock (tsl) instruction:**
  Simple hardware solution. Busy waiting.

- **Sleep and wakeup:**
  Avoid busy waiting.
  Wakeup lost causes problems.

- **Classical IPC problem:**
  Producer consumer problem.

- **Use sleep and wakeup to solve producer consumer problem.**

Review:

Solutions for IPC problems (Cont.)

- **Semaphore**
  - **Up and down operations**
  - **Implementation**
  - **Binary semaphore and counting semaphore**
  - **Drawbacks**
- **Event counter**
  - **No mutual exclusion required, more efficient.**
  - **Drawbacks**
- **Monitor: high-level IPC primitive**

- **Message passing**
  - **Why use message passing**
    - **Two parts of communication can be totally separated (no shared data)**
    - **No invisible side effects**
    - **No need to know the other part**
  - **Message:**
    a piece of information that is passed from one process to another.
  - **Mailbox: a shared data structure where messages are stored between the time they are sent and the time they are received.**

– **Operations:**

  * **send: copy a message into mailbox. If the mailbox is full wait until there is enough space in the mailbox.**

    Format: send(destination, message)

  * **receive: copy a message out of mailbox, and delete from mailbox. If the mailbox is empty, then wait until a message arrives.**

    Format: receive(source, message)

- **Design issues of message system**
  - **Addressing: how to specify the sending and receiving processes.**
    - **Direct addressing: sender and receiver communicate directly.**
      send: a specific identification of the destination process, such as process@machine.domain
      receive:
      (a) explicit addressing
      (b) implicit addressing

    - **Indirect addressing: messages are sent to a shared data structure called mailboxes (queues that can temporarily hold messages)**
      Relationship between mailboxes and processes
      (a) One mailbox per process. Use process name in send, no name in receive.
      (b) No strict mailbox-process association, use mailbox name.

∗ **Extent of buffering**

- **Buffering**
- **None – rendezvous protocol**

∗ **Blocking vs. non-blocking operations**

- **Blocking receive:**
  receive message if mailbox is not empty, otherwise wait until message arrives

- **Non-blocking receive:**
  receive message if mailbox is not empty, otherwise return.

- **Blocking send:**
  wait until mailbox has space.

- **Nonblocking send:**
  return "full" if no space in mailbox.
- **Four possible send and receive combinations.**

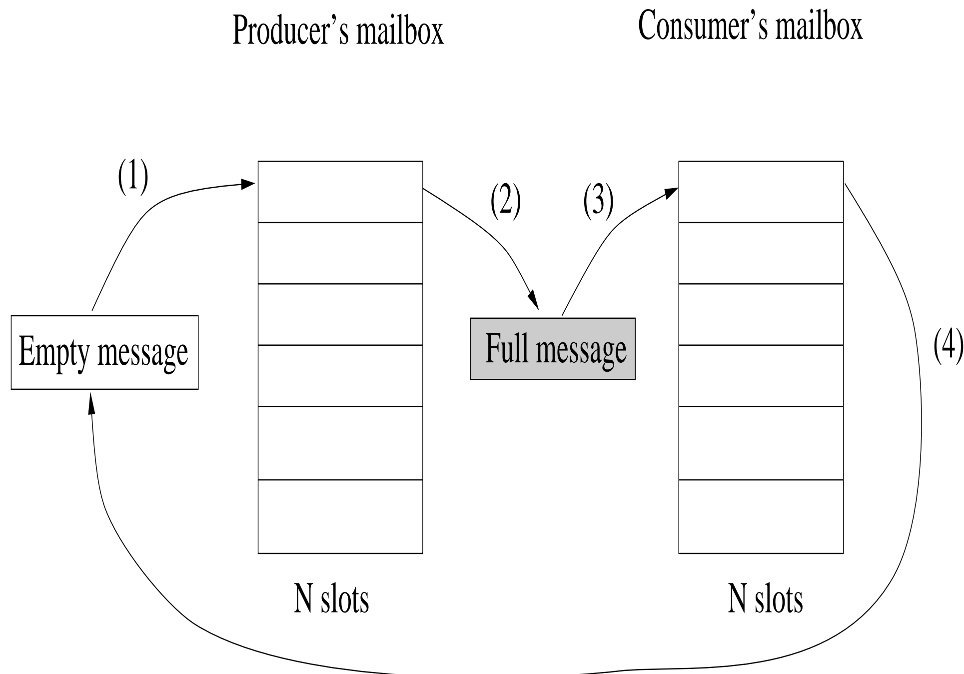
∗ **Message format**

Source
Destination
Message length
Control information
Message type
Message contents

∗ **Queueing discipline**

- **First in first out (FIFO)**
- **Priority**

– **Solving producer consumer problem with message passing (use mailboxes).**



(1) Consumer sends empty message to producer's mailbox.

(2) Producer takes empty message and builds full message.

(3) Producer sends full message to consumer's mailbox.

(4) Consumer takes full message out and consumes it.

Equivalence of primitives:

Semaphores, monitors and messages are equivalence. Each of these methods can be used to implement the other methods.

1. **Implement monitors with semaphores**

   - **Associate with each monitor a binary semaphore mutex (monitor lock), initially 1.**

   - **Associate with each condition variable a semaphore, initially 0.**

     Translate:

     $$\textbf{wait(c)} \Rightarrow \begin{cases} \textbf{up(mutex)} \\ \textbf{down(c)} \\ \textbf{down(mutex)} \end{cases}$$

     $$\textbf{signal(c)} \Rightarrow \textbf{up(c)}$$

2. Implement message passing with semaphores.

- **Associate with each process a semaphore, initially 0, on which it will block.**
- **A shared buffer area holds mailboxes. Each mailbox contains:**
  - **# full slots**
  - **# empty slots**
  - **send queue (those processes which cannot send their messages to the mailbox**
  - **receive queue (those processes which cannot receive their message from the mailbox**
  - **messages linked together**
- **A semaphore, mutex, to protect the shared buffer area.**

- **send/receive operations:**

  Case 1. Mailbox has at least one empty or full slot:
  down(mutex)
  insert/remove message
  update counters and links
  up(mutex)

  Case 2. Process i does receive on an empty mailbox:
  down(mutex)
  enter receive queue
  up(mutex)
  down(P_i)
  down(mutex)

  Case 3. Process i does send on a full mailbox:
  down(mutex)
  enter send queue
  up(mutex)
  down(P_i)
  down(mutex)

- **How to wake up sleeping processes?**
  - **If a receiver receives a message from the full mailbox, wakes up (does up) the first process in the send queue.**
  - **If a sender sends a message to the empty mailbox, wakes up the first process in the receive queue.**

3. Implement semaphore with monitors

- **Associate with each semaphore a counter and a linked list.**
  - **Counter stores the value of the semaphore**
  - **Linked list stores the processes sleeping on the semaphore**
- **Associate with each process a condition variable**
- **Operations:**

  down(s) : If $counter_s > 0$**, then** $counter_s^{--}$
  else {enter linked list of s**; wait (**P_i**)}**

  up(s) : If linked list not empty,
  then {remove one process from the list, say P_i**; signal(**P_i**)}**
  else $counter_s^{++}$**;**

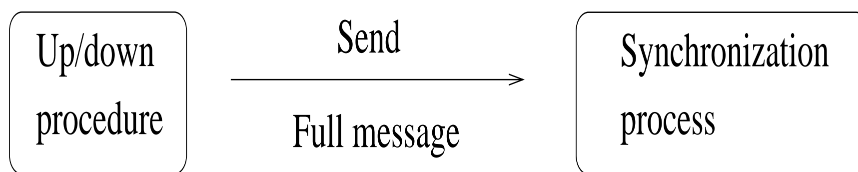
4. Implement messages with monitors.

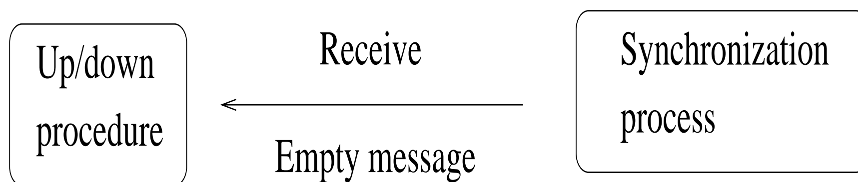
- **Associate with each process a condition variable**

- **A shared buffer**
- **Similar to semaphores except no mutex necessary.**

5. Implement semaphores with messages.

- **For mutual exclusion, introduce a new process, synchronization process.**
- **Associate each semaphore with a counter and a linked list of waiting processes.**
- **Operations:**
  To do up or down on a semaphore, call the corresponding library procedure up or down.





Synchronization process does:

down: **If count $>$ 0 {counter$--$; send back empty message}**
else { enter caller into queue and does not send reply;}

up: **If counter $=$ 0 {move one process out of queue; send this process a reply }**
else counter$++$;

6. Implement monitor with message passing

Combine (5) and (1). That is, using messages to implement semaphores first, then using semaphores to implement monitors.

Review:

Solutions for IPC problems (Cont.)

- **Monitor: high-level IPC primitive**
  - **Implemented in programming languages**
  - **Mandatory mutual exclusion**
  - **Easier and safer to use**
  - **Wait and signal**
  - **Drawbacks**
- **Message passing**
  - **No shared memory necessary**
  - **Communicating processes can be totally separated**
  - **Concepts**
    - **Message**
    - **Mailbox**
    - **Send/receive operations**
  - **Design issues**
    - **Addressing: indirect and direct**
    - **Buffering**
    - **Blocking and non-blocking**