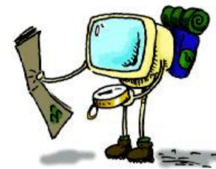
Programming Languages

Tevfik Koşar

Lecture - XXVII
May 2nd, 2006

Roadmap

- Shared Memory
  - Cigarette Smokers Problem
  - Monitors
- Message Passing
  - Cooperative Operations
  - Synchronous and Asynchronous Sends
  - Blocking and Non-blocking Operations
  - Collective Communications



2

Cigarette Smokers Problem

- Assume a cigarette requires three ingredients to smoke:
  - Tobacco
  - Paper
  - Match
- Assume there are also three smokers around a table, each of whom has an infinite supply of *one* of the three ingredients
- Assume there is also a non-smoking arbiter. The arbiter enables the smokers to make their cigarettes by
  - selecting two of the smokers at random,
  - taking one item out of each of their supplies,
  - and placing the items on the table.
  - He then notifies the third smoker that he has done this.
- The third smoker removes the two items from the table and uses them (along with his own supply) to make a cigarette, which he smokes for a while.
- Meanwhile, the arbiter, seeing the table empty, again chooses two smokers at random and places their items on the table. This process continues forever.

3

Solution

- Let us define an array of binary semaphores *A*, one for each smoker; and a binary semaphore for the table, *T*.
- Initialize the smokers' semaphores to zero and the table's semaphore to 1.
- Then the arbiter's code is :

```
while true {
    wait(T);
    choose smokers i and j randomly, making the third smoker k
    signal(A[k]);
}
```

4

Solution *(cont.)*

- And the code for smoker *i* is:

```
while true {
   wait(A[i]);
   make a cigarette
   signal(T);
   smoke the cigarette
}
```

5

Semaphores

- Although widely used, they are considered as too "low level", and generally used low level code, eg. operating systems
- Since their operations are simply subroutine calls, it is very easy to leave out one (eg. forgetting to release a semaphore that has been taken)
- Generally scattered throughout a program, and hard to debug and maintain

6

Monitors

- Suggested by Dijkstra (‘72)
- Formulized by Hoare (’74)

- Monitors encapsulate all synchronization code into a single structure (like objects in OOP)
- Only one operation of a given monitor is allowed to be active at a given time (ensured by the compiler)
- A thread which calls a busy monitor is automatically delayed until the monitor is free

- No need for P and V, and correctly ordering them

Monitors

A monitor consists of:

- a set of procedures that allow interaction with the shared resource
- a mutual exclusion lock
- the variables associated with the resource
- a monitor invariant that defines the assumptions needed to avoid race conditions

A monitor procedure takes the lock before doing anything else, and holds it until it either finishes or waits for a condition

If every procedure guarantees that the invariant is true before it releases the lock, then no task can ever find the resource in a state that might lead to a race condition.

Monitors

As a simple example, consider a monitor for performing transactions on a bank account.

```
monitor account {
    int balance := 0

  function withdraw(int amount) {
    if amount < 0 then error "Amount may not be negative"
    else if balance < amount then error "Insufficient funds"
    else balance := balance - amount
  }

  function deposit(int amount) {
    if amount < 0 then error "Amount may not be negative"
    else balance := balance + amount
  }
}
```

9

Dining Philosophers using Monitors

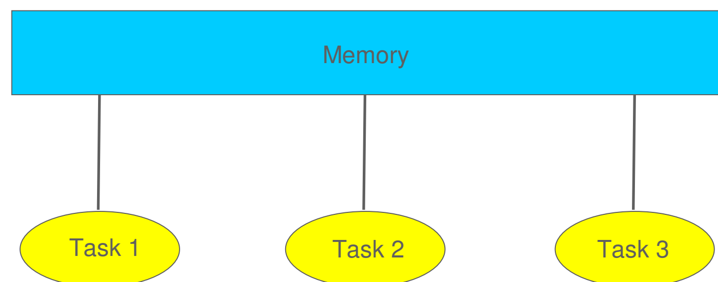
```
class Monitor{
    boolean forks[];

    public Monitor(int size) {
        forks=new boolean[size];
        for(int i=0;i<forks.length;i++)
            forks[i]=true;
    }

    public synchronized void getForks(int id) throws Exception{
        int index1;
        int index2;
        if(id==0) // Philosopher 0 {
            index1=0;
            index2=forks.length-1;
        } else {
            index1=id;
            index2=id-1;
        }
        while(!forks[index1] || !forks[index2])
            wait();
            forks[index1]=false;
            forks[index2]=false;
            System.out.println("Philospher "+id+" got the forks: "+index1+" "+index2);
        }
```

10

Dining Philosophers using Monitors *(cont.)*

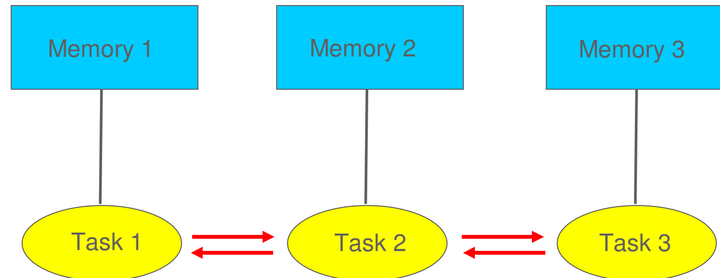
```
    public synchronized void release(int id) throws Exception{
        int index1;
        int index2;
        if(id==0) // Philosopher 0 {
                index1=0;
                index2=forks.length-1;
        } else {
                index1=id;
                index2=id-1;
        }
        forks[index1]=true;
        forks[index2]=true;
        System.out.println(" Philosopher "+id+" has finished a meal");
        notifyAll();
    }
}
```


Shared Memory



- Each process/task has access to the same memory

Distributed Memory



- Each process/task has access to its own memory
- Processes/Tasks can communicate via messages

13

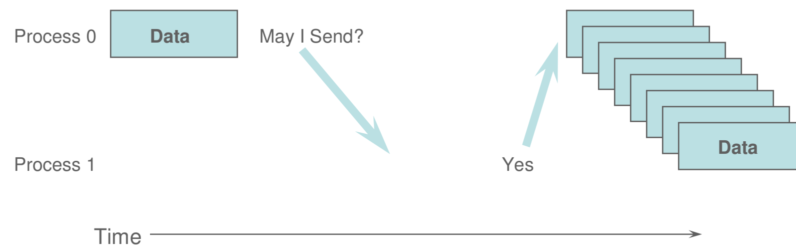
Message Passing

- What is message passing?
  - A mechanism for communicating data between processes
  - In point-to-point communication, one process sends data and another process receives data
  - In collective communication, there may be more than one sending and/or receiving process
- Message passing is important in many areas:
  - parallel computing
  - distributed computing
  - operating systems

14

What is message passing?

- Data transfer plus synchronization

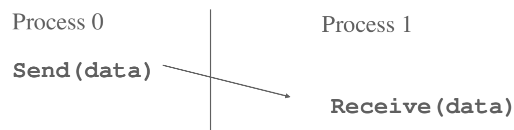


- Requires cooperation of sender and receiver

15

Cooperative Operations for Communication

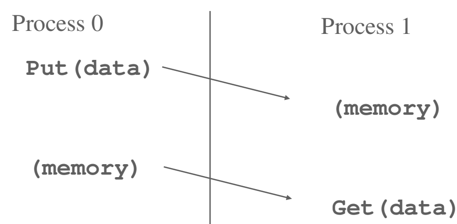
- The message-passing approach makes the exchange of data *cooperative*.
- Data is explicitly *sent* by one process and *received* by another.
- An advantage is that any change in the receiving process's memory is made with the receiver's explicit participation.
- Communication and synchronization are combined.



16

One-Sided Operations for Communication

- One-sided operations between processes include remote memory reads and writes
- Only one process needs to explicitly participate.
- An advantage is that communication and synchronization are decoupled
- One-sided operations are part of MPI-2
- Not part of the basic message passing approach!



Message Passing

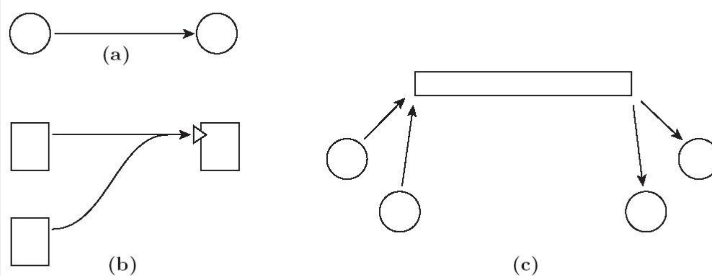
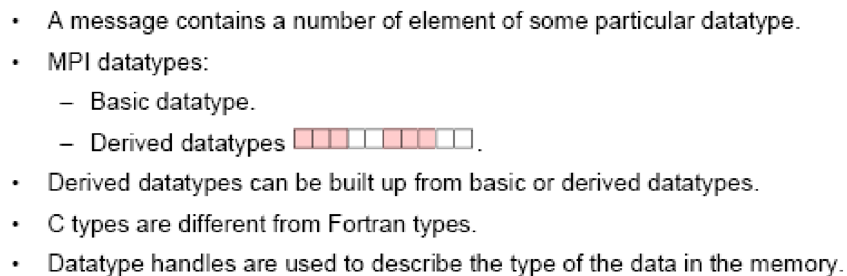


Figure 12.18: **Three common schemes to name communication partners.** In (a), processes name each other explicitly. In (b), senders name an *input port* of a receiver. The port may be called an *entry* or an *operation*. The receiver is typically a module with one or more threads inside. In (c), senders and receivers both name an independent *channel* abstraction, which may be called a *connection* or a *mailbox*.

Messages



Example: message with 5 integers

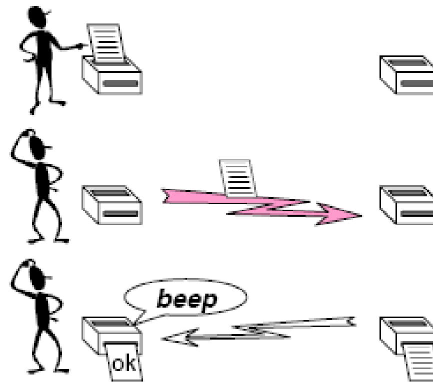
2345	654	96574	-12	7676

Message Passing Paradigms

- MPI (Message Passing Interface)
  - aims to develop a standard for writing message passing programs
  - designed for homogeneous architectures
- PVM (Parallel Virtual Machine)
  - allows a heterogeneous network of computers (parallel, vector, or serial) to appear as a single concurrent computational resource - a virtual machine.
- Both provides libraries for C and Fortran programs

Synchronous Sends

- The sender gets an information that the message is received.
- Analogue to the *beep* or *okay-sheet* of a fax.



21

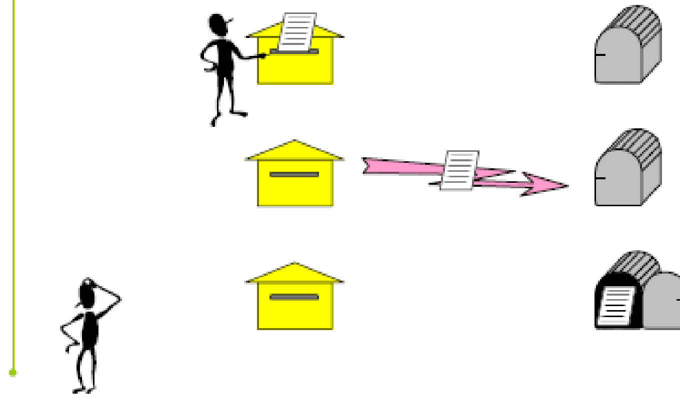
Blocking Operations

- Operations are local activities, e.g.,
  - sending (a message)
  - receiving (a message)
- Some operations may **block** until another process acts:
  - synchronous send operation **blocks until** receive is posted;
  - receive operation **blocks until** message is sent.
- Relates to the completion of an operation.
- Blocking subroutine returns only when the operation has completed.

22

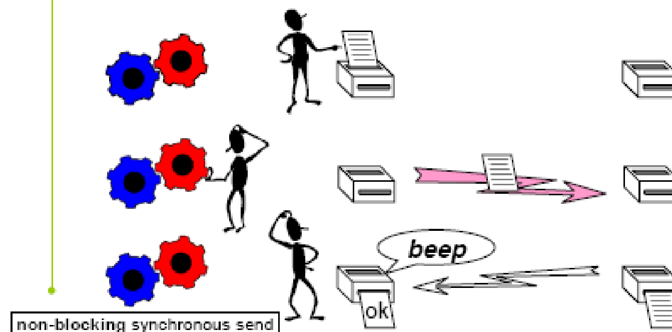
Buffered = Asynchronous Sends

- Only know when the message has left.



Non-blocking Operations

- Non-blocking operation: returns immediately and allow the sub-program to perform other work.
- At some later time the sub-program must ***test*** or ***wait*** for the completion of the non-blocking operation.



Non-blocking Operations *(cont.)*

- All non-blocking operations must have matching wait (or test) operations. (Some system or application resources can be freed only when the non-blocking operation is completed.)
- A non-blocking operation immediately followed by a matching wait is equivalent to a blocking operation.
- Non-blocking operations are not the same as sequential subroutine calls:
  - the operation may continue while the application executes the next statements!

25

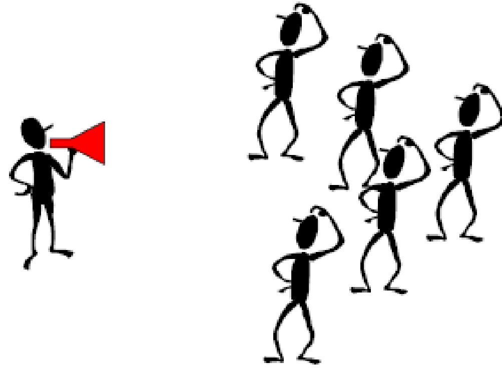
Collective Communications

- Collective communication routines are higher level routines.
- Several processes are involved at a time.
- May allow optimized internal implementations, e.g., tree based algorithms
- Can be built out of point-to-point communications.

26

Broadcast

- A one-to-many communication.

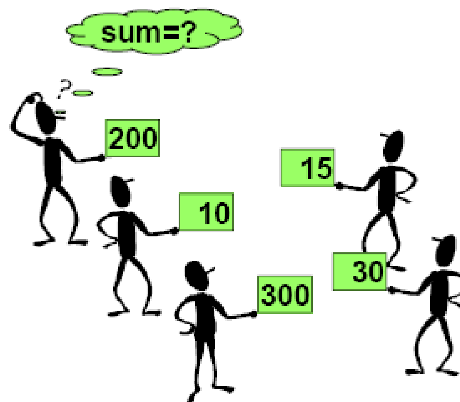


MPI_BCast(…);

27

Reduction Operations

- Combine data from several processes to produce a single result.

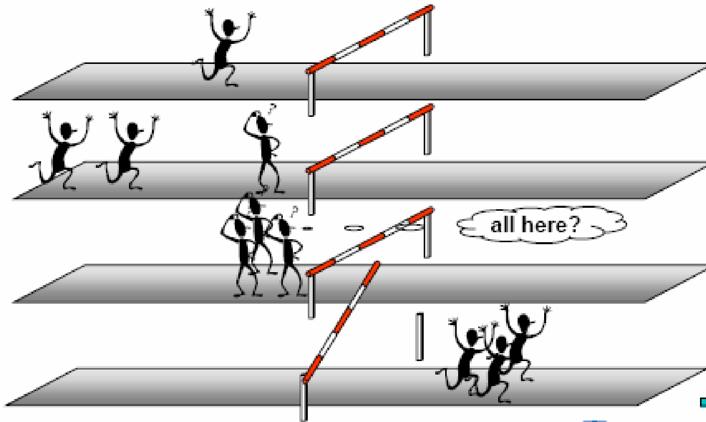


MPI_Reduce(…);

28

Barriers

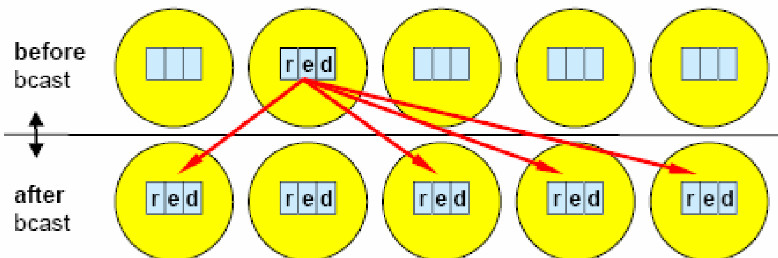
- Synchronize processes.



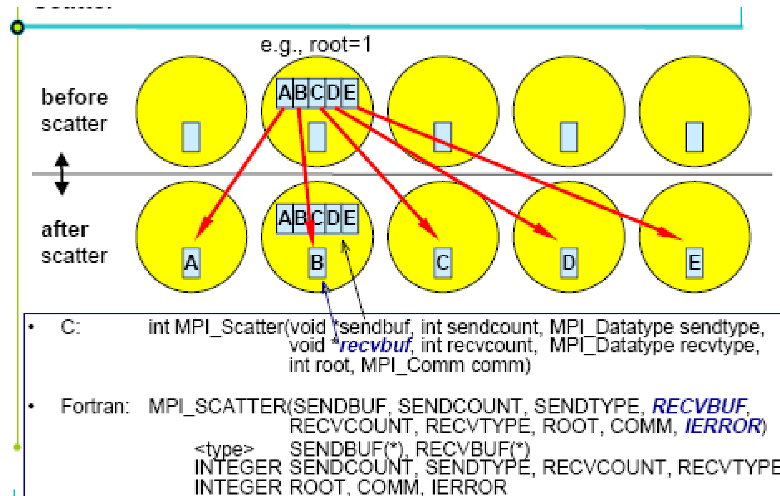
MPI_Barrier(…);

Broadcast

- C: int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
- Fortran: MPI_Bcast(BUF, COUNT, DATATYPE, ROOT, COMM, *IERROR*)
  <type> BUF(*)
  INTEGER COUNT, DATATYPE, ROOT
  INTEGER COMM, IERROR

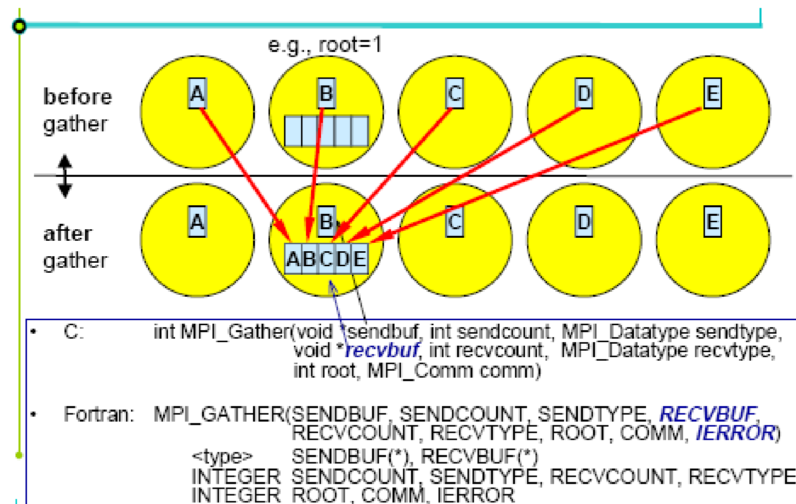


Scatter



31

Gather



32