FIT3143

Parallel Computing

Week 5: Synchronization, Mutex, Deadlocks

Vishnu Monn and ABM Russel

Unit Topics

Week	Lecture Topic	Tutorial	Laboratory	Remarks
1	Introduction to Parallel Computing and Distributed Systems	None	Lab Week 01 - Introduction to Linux & Setting up VM (Not assessed)	Group formation for lab activities and assignments (Two students per group)
2	Parallel computing on shared memory (POSIX and OpenMP)	Tutorial Week 02 - Introduction to Parallel Computing	Lab Week 02 - C Primer (Not assessed)	
3	Inter Process Communications; Remote Procedure Calls	Tutorial Week 03 - Shared memory parallelism	Lab Week 03 - Threads (POSIX)	Assignment 1 specifications released
4	Parallel computing on distributed memory (MPI)	Tutorial Week 04 - Inter process communication	Lab Week 04 - Threads (OpenMP)	
5	Synchronization, MUTEX, Deadlocks	Tutorial Week 05 - Distributed memory parallelism	Lab Week 05 - Message passing interface	Assignment 2 specifications released
6	Election Algorithms, Distributed Transactions, Concurrency Control	Tutorial Week 06 - Synchronization	Lab Week 06 - Communication patterns	
7	Faults, Distributed Consensus, Security, Parallel Computing	Tutorial Week 07 - Transactions and concurrency	Lab Week 07 - Parallel data structure (I)	Assignment 1 due
8	Instruction Level Parallelism	Tutorial Week 08 - Consensus	1st assignment interview	
9	Vector Architecture	Tutorial Week 09 - Instruction level parallelism	Lab Week 09 - Parallel data structure (II)	
10	Data Parallel Architectures, SIMD Architectures	Tutorial Week 10 - Vector architecture	Lab Week 10 - Master slave	
11	Introduction to MIMD, Distributed Memory MIMD Architectures	Tutorial Week 11 - Data parallelism	Lab Week 11 - Performance tuning	Assignment 2 due
12	Recent development in Parallel Computing - Software, Hardware, Applications etc. (GPGPU etc.), Exam Revision	Tutorial Week 12 - Revision	2nd assignment code demo and interview	

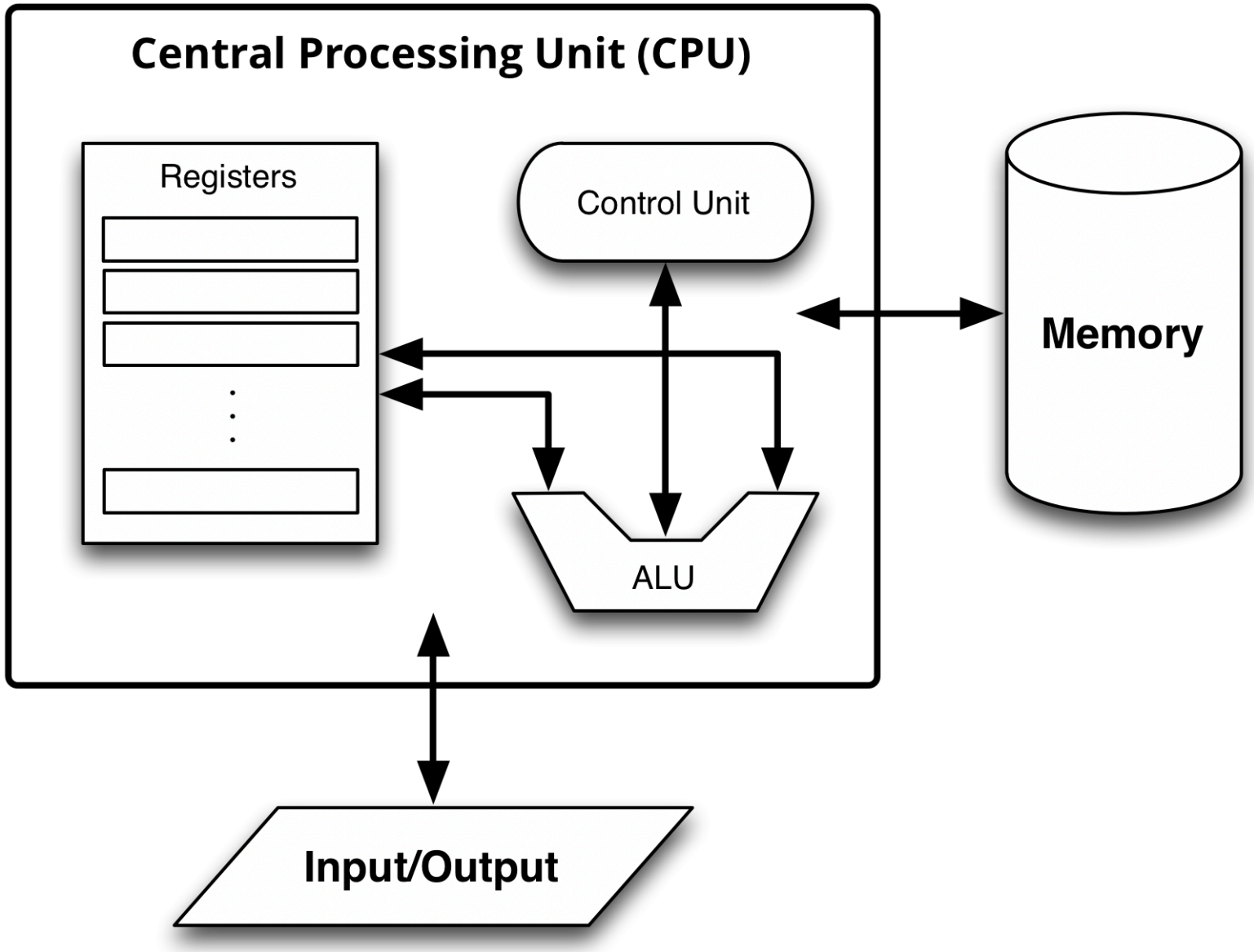
Assessment	Weight
Assignment 1	15%
Assignment 2	25%
Lab-work and Quizzes	10%
Final exam	50%

Feedback

- Assignments
- Lecture
- Tutorial
- Lab
- Consultations

Please share your experience on online learning.

Background


Central Processing Unit (CPU)
Registers
Control Unit
Memory
ALU
Input/Output

Today

- Recap: MPI
- Synchonisation
- Mutex
- Deadlock

Recap

Environment Management Routines

- MPI_Init (&argc, &argv)
- MPI_Comm_size (comm, &size)
- MPI_Comm_rank (comm, &rank)
- MPI_Abort (comm, errorcode)
- MPI_Get_processor_name (&name, &resultlength)
- MPI_Initialized (&flag)
- MPI_Wtime ()
- MPI_Wtick ()
- MPI_Finalize ()

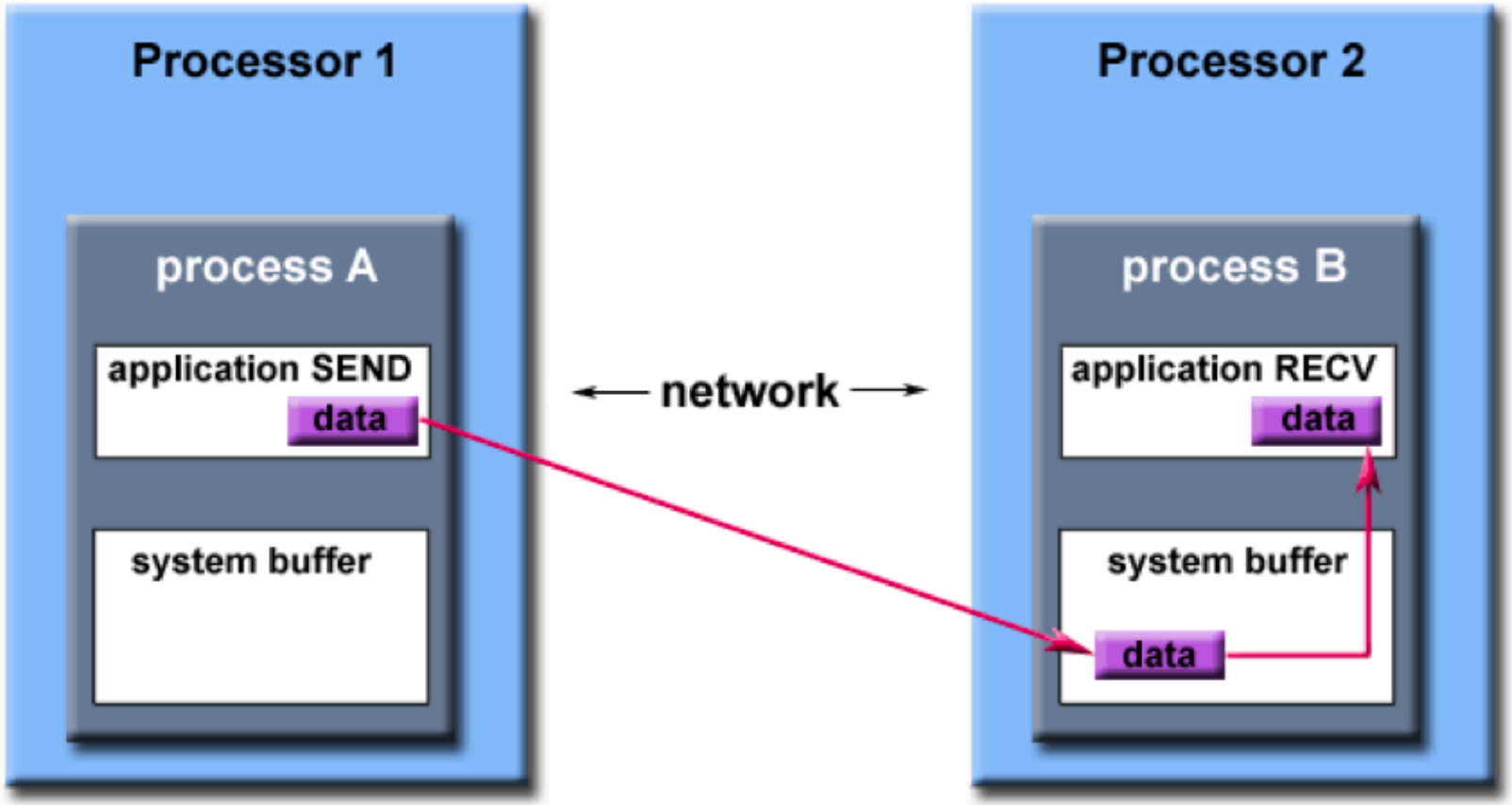
Environment Management Template …

```
#include "mpi.h"
#include <stdio.h>

int main(int argc, char *argv[])
{
    int numtasks, rank, rc;
    rc = MPI_Init(&argc,&argv);
    if (rc != MPI_SUCCESS)
   {  printf ("Error starting MPI program. Terminating.\n");
    MPI_Abort(MPI_COMM_WORLD, rc);
   }
    MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    printf ("Number of tasks= %d My rank= %d\n", numtasks,rank);

    /******* do some work *******/

    MPI_Finalize();
}
```

Point to Point Communication

- MPI point-to-point operations typically involve message passing between two, and only two, different MPI tasks. One task is performing a send operation and the other task is performing a matching receive operation.
- Different types of send and receive routines used for different purposes. For example:
    - Synchronous send
    - Blocking send / blocking receive
    - Non-blocking send / non-blocking receive
    - Buffered send
    - Combined send/receive
    - "Ready" send
- Any type of send routine can be paired with any type of receive routine.
- MPI also provides several routines associated with send - receive operations, such as those used to wait for a message's arrival or probe to find out if a message has arrived.

System Buffer



Path of a message buffered at the receiving process

Blocking Message Passing Routines

- MPI_Send (&buf,count,datatype,dest,tag,comm)
- MPI_Recv (&buf,count,datatype,source,tag,comm,&status)

Blocking …

- MPI_Ssend (&buf,count,datatype,dest,tag,comm)
- MPI_Bsend (&buf,count,datatype,dest,tag,comm)
  - MPI_Buffer_attach (&buffer,size)
  - MPI_Buffer_detach (&buffer,size)
- MPI_Rsend (&buf,count,datatype,dest,tag,comm)
- MPI_Sendrecv (&sendbuf,sendcount,sendtype,dest,sendtag, &recvbuf,recvcount,recvtype,source,recvtag, comm,&status)

Blocking …

- MPI_Wait (&request,&status)
- MPI_Waitany (count,&array_of_requests,&index,&status)
- MPI_Waitall (count,&array_of_requests,&array_of_statuses)
- MPI_Waitsome (incount,&array_of_requests,&outcount, &array_of_offsets, &array_of_statuses)
- MPI_Probe (source,tag,comm,&status)

Non-Blocking Message Passing Routines

- MPI_Isend (&buf,count,datatype,dest,tag,comm,&request)
- MPI_Irecv (&buf,count,datatype,source,tag,comm,&request)
- MPI_Issend (&buf,count,datatype,dest,tag,comm,&request)
- MPI_Ibsend (&buf,count,datatype,dest,tag,comm,&request)
- MPI_Irsend (&buf,count,datatype,dest,tag,comm,&request)

Non-Blocking Message Passing Routines

- MPI_Test (&request,&flag,&status)
- MPI_Testany (count,&array_of_requests,&index,&flag,&status)
- MPI_Testall (count,&array_of_requests,&flag,&array_of_statuses)
- MPI_Testsome (incount,&array_of_requests,&outcount, &array_of_indices, &array_of_statuses)
- MPI_Iprobe (source,tag,comm,&flag,&status)

FLUX: Blocking/Non-Blocking

Blocking/Non-Blocking

JYGAFY

Collective Communication

Collective Communication

- All or None
- Types of Collective Operations:
  - **Synchronization** - processes wait until all members of the group have reached the synchronization point.
  - **Data Movement** - broadcast, scatter/gather, all to all.
  - **Collective Computation** (reductions) - one member of the group collects data from the other members and performs an operation (min, max, add, multiply, etc.) on that data.

Collective Communication Routines

- MPI_Barrier (comm)
- MPI_Bcast (&buffer,count,datatype,root,comm)
- MPI_Scatter (&sendbuf,sendcnt,sendtype,&recvbuf, recvcnt,recvtype,root,comm)
- MPI_Gather (&sendbuf,sendcnt,sendtype,&recvbuf, recvcount,recvtype,root,comm)

Collective Communication …

- MPI_Allgather (&sendbuf,sendcount,sendtype,&recvbuf, recvcount,recvtype,comm)
- MPI_Reduce (&sendbuf,&recvbuf,count,datatype,op,root,comm)
- MPI_Allreduce (&sendbuf,&recvbuf,count,datatype,op,comm)
- MPI_Reduce_scatter (&sendbuf,&recvbuf,recvcount,datatype, op,comm)
- MPI_Alltoall (&sendbuf,sendcount,sendtype,&recvbuf, recvcnt,recvtype,comm)
- MPI_Scan (&sendbuf,&recvbuf,count,datatype,op,comm)

Example

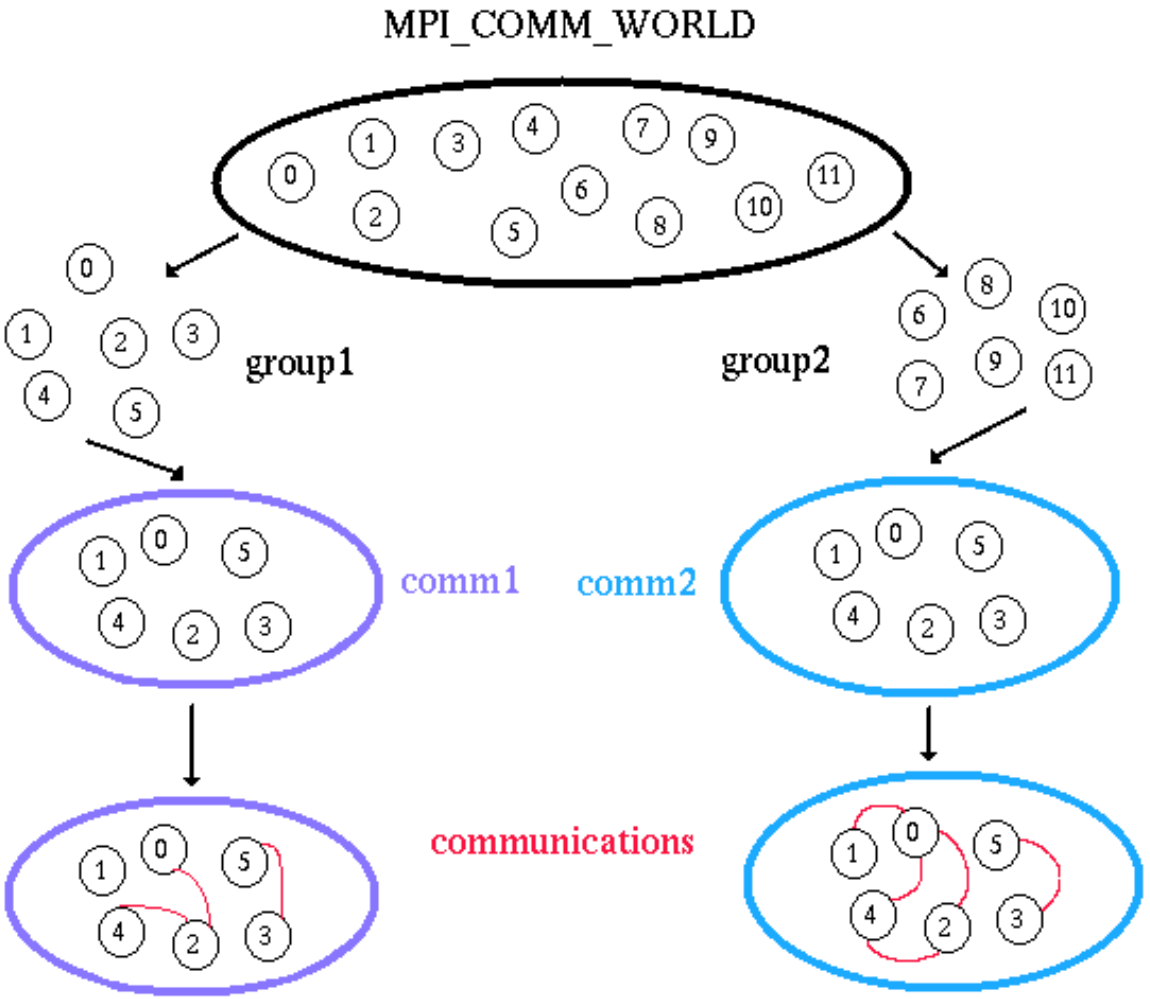
```
#include "mpi.h"
#include <stdio.h>
#define SIZE 4

int main(int argc, char *argv[])
{
int numtasks, rank, sendcount, recvcount, source;
float sendbuf[SIZE][SIZE] = {
  {1.0, 2.0, 3.0, 4.0},
  {5.0, 6.0, 7.0, 8.0},
  {9.0, 10.0, 11.0, 12.0},
  {13.0, 14.0, 15.0, 16.0}  };
float recvbuf[SIZE];

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

if (numtasks == SIZE) {
  source = 1;
  sendcount = SIZE;
  recvcount = SIZE;
  MPI_Scatter(sendbuf,sendcount,MPI_FLOAT,recvbuf,recvcount,
              MPI_FLOAT,source,MPI_COMM_WORLD);

  printf("rank= %d  Results: %f %f %f %f\n",rank,recvbuf[0],
         recvbuf[1],recvbuf[2],recvbuf[3]);
  }
else
  printf("Must specify %d processors. Terminating.\n",SIZE);

MPI_Finalize();
}
```

FLUX: Collective Operations

Collective operations

JYGAFY

Group and Communicator Management Routines



Cartesian virtual topology

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)
12 (3,0)	13 (3,1)	14 (3,2)	15 (3,3)

Synchronization in Distributed Systems

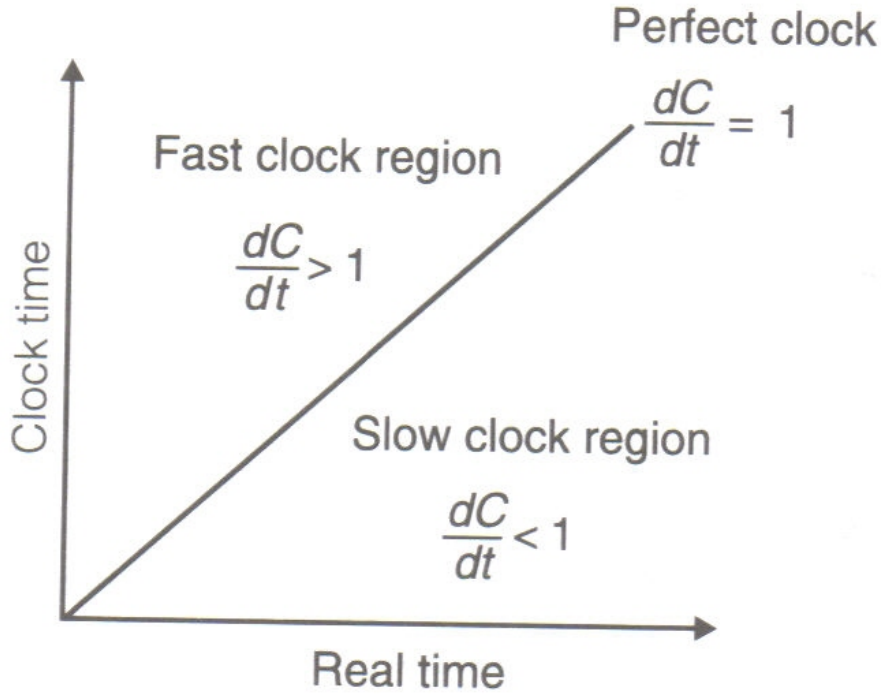
Semaphores and Mutex

We cannot use semaphores and monitors in distributed systems since two processes running on different machines cannot expect to have shared memory.

DS Synchronization is difficult

- The relevant information is scattered among multiple machines.
- Processes make decisions based only on local information.
- A single point of failure in the system should be avoided.
- No common clock or other precise global time source exists.

Clock Synchronization



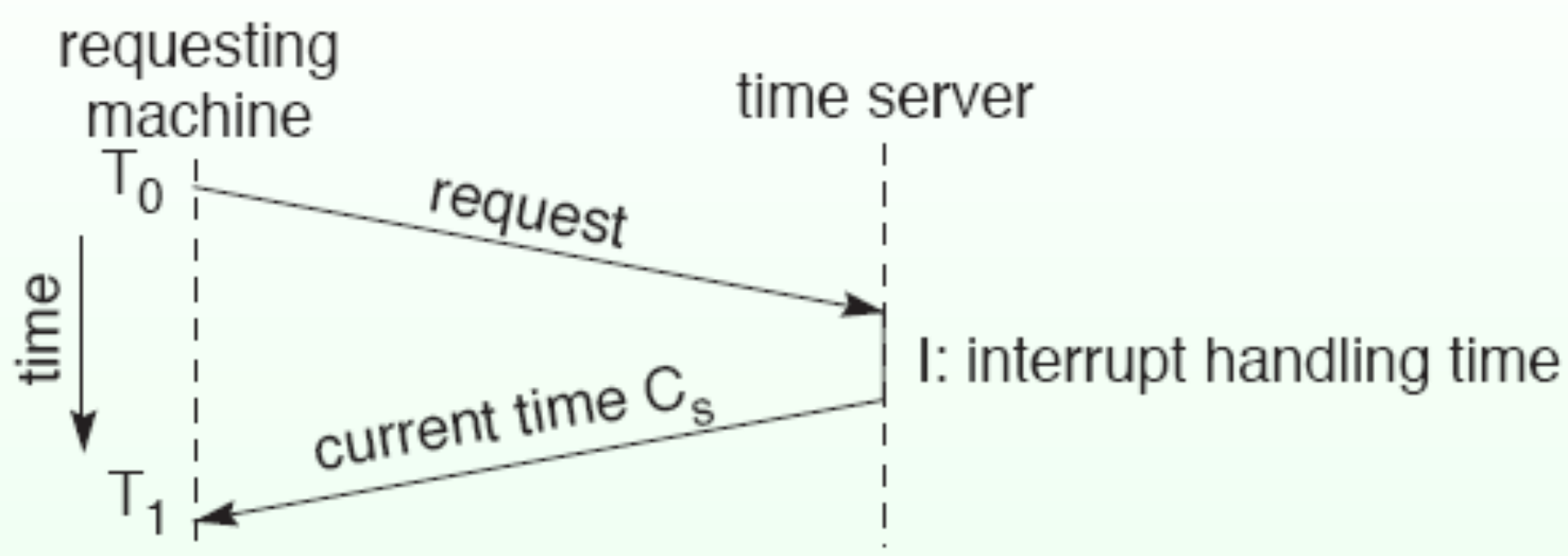
Drift

Suppose, when the real time is t the time value of a clock p is $C_p(t)$. If the maximum drift rate allowable is ρ, a clock is said to be non-faulty if the following condition holds

$$1 - \rho \leq \frac{dC}{dt} \leq 1 + \rho$$

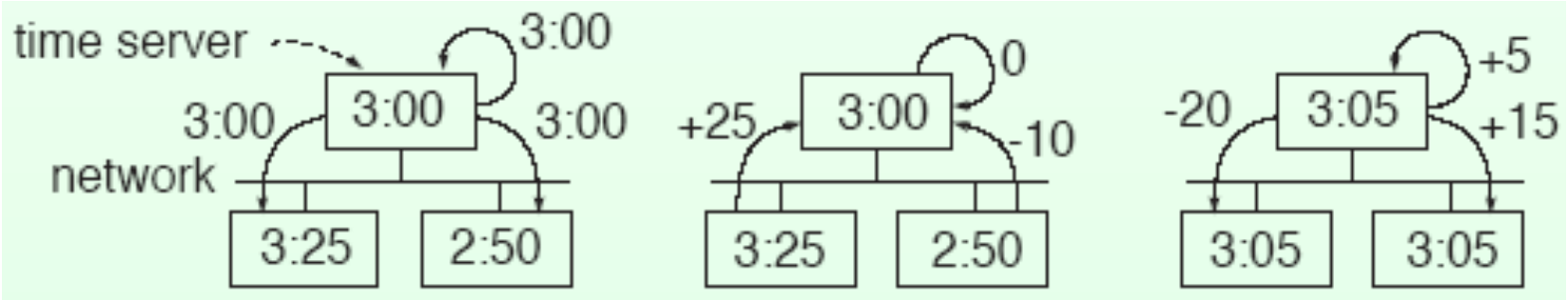
Physical Clock

Cristian's Algorithm



Requesting machine sets its clock to C_s +(T1 −T0 −I)/2

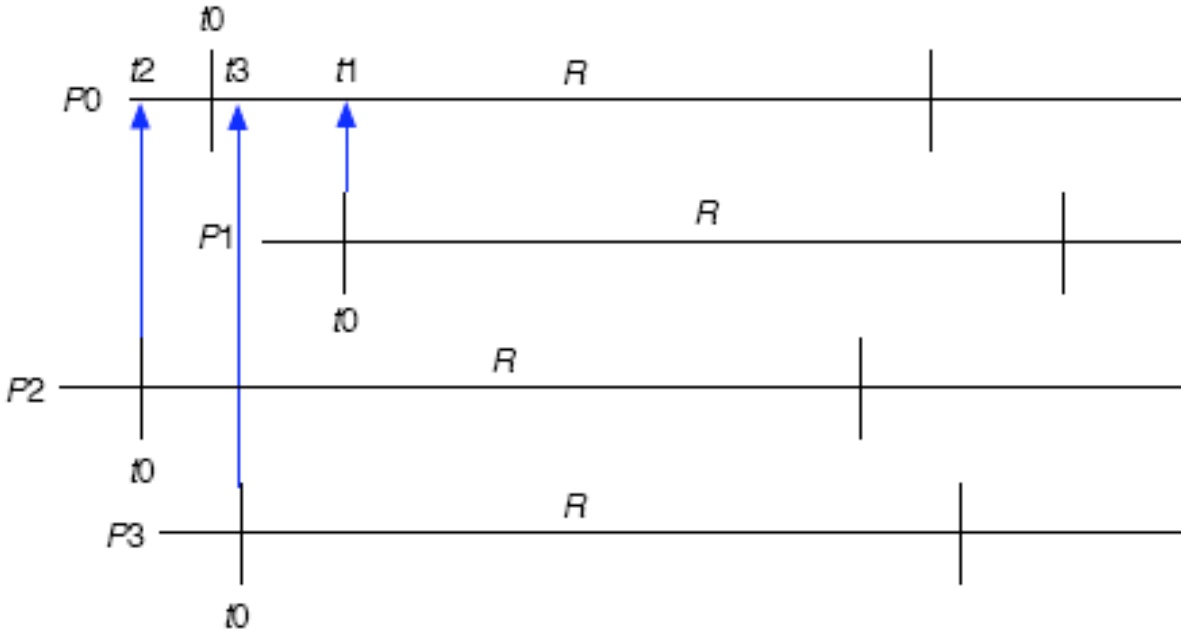
The Berkeley Algorithm



Computer systems normally avoid rewinding their clock when they receive a negative clock alteration from the master

clock slew

Averaging Algorithm



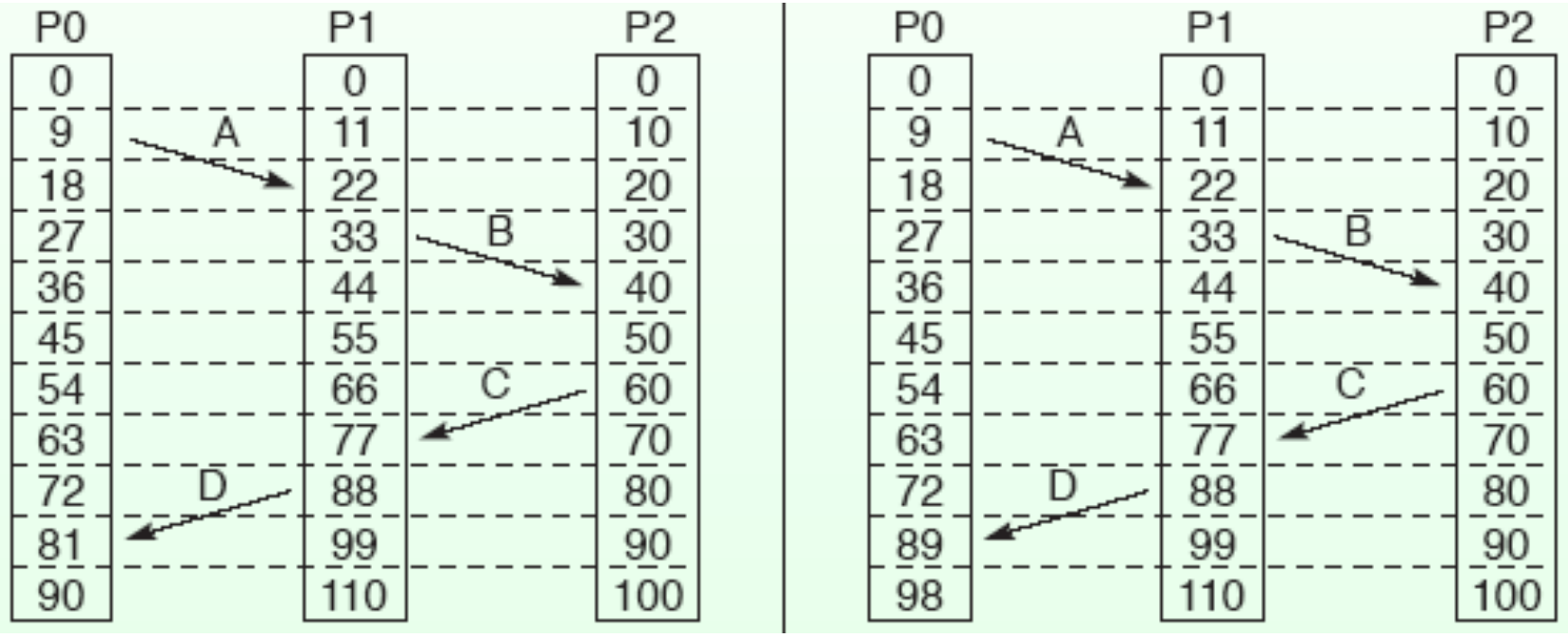
- Clock on processor *P*0 should be advanced by Δ*t*0 as below

$$\Delta t_0 = \frac{t_0 + t_1 + t_2 + t_3}{4} - t_0$$

Logical Clock

Lamport's Synchronization Algorithm

"Happens-before" relation: "A →B" is read "A happens before B": This means that all processes agree that event A occurs before event B.

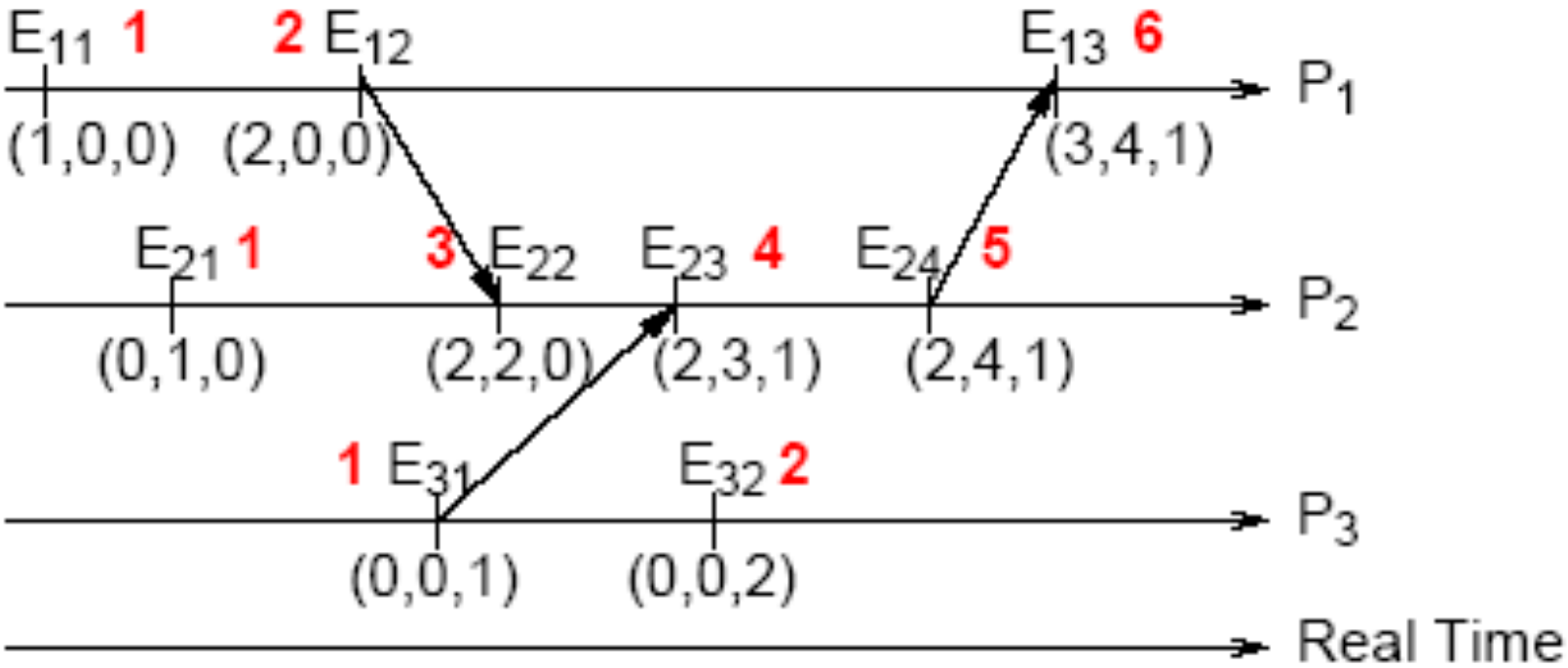


To assign a time value $C(A)$ on which all processes agree for every event A. The time value must have the following properties:

- If $A \rightarrow B$, then $C(A) < C(B)$.
- The clock time must always go forward, never backward.

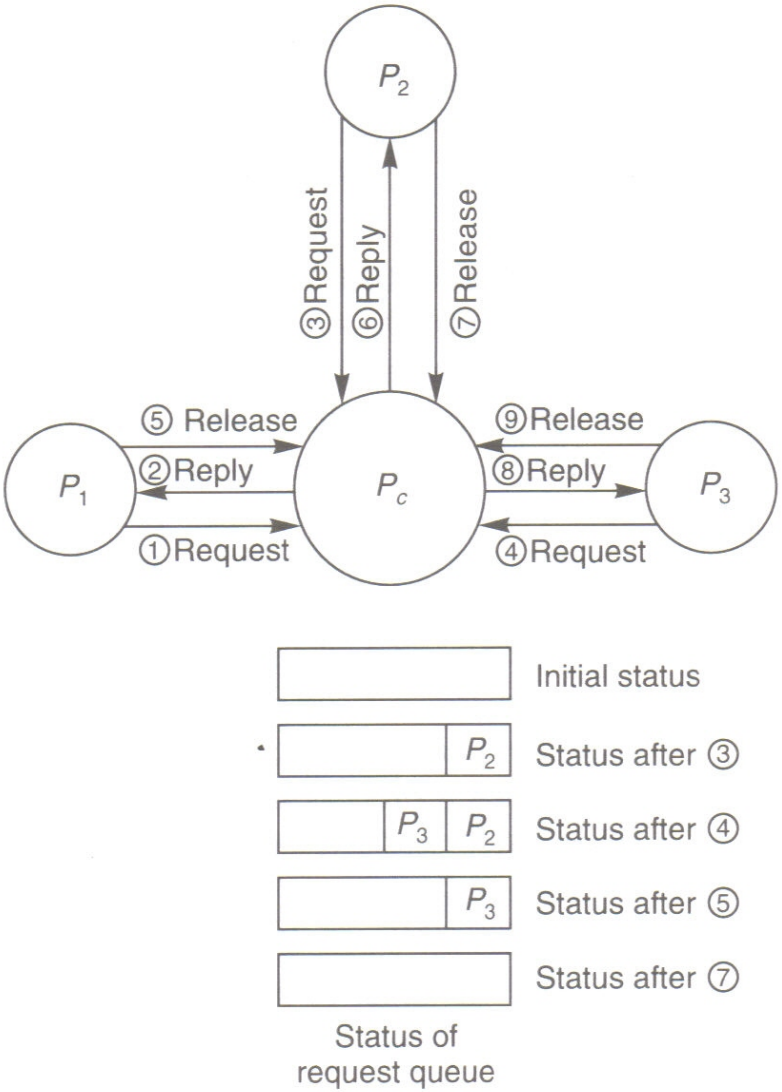
Vector Clock

- Each event is annotated with both its vector clock value (the triple) and the corresponding value of a scalar Lamport clock (Red color).
- For $C_1(E_{12})$ and $C_3(E_{32})$, we have 2 = 2 versus (2, 0, 0) ≠ (0, 0, 2). Likewise we have $C_2(E_{24}) > C_3(E_{32})$ but (2, 4, 1) NOT> (0, 0, 2) and thus E_{32} NOT$\rightarrow E_{24}$

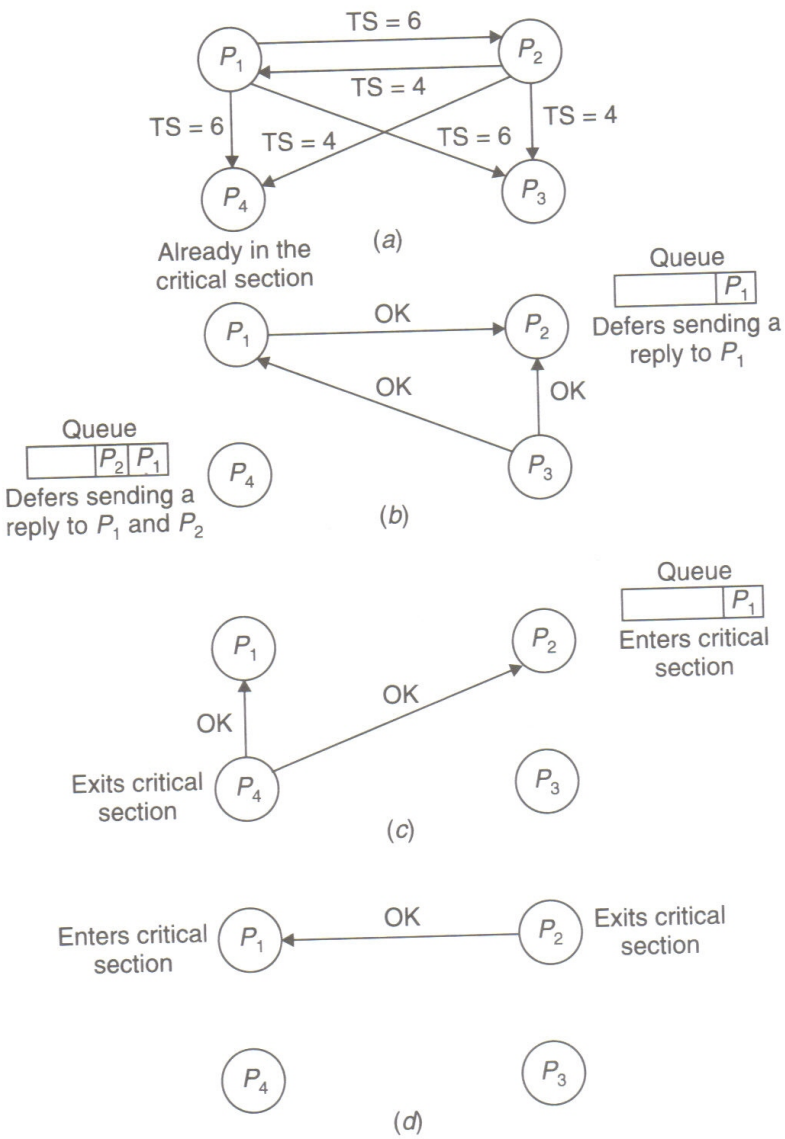


Mutual Exclusion in DS

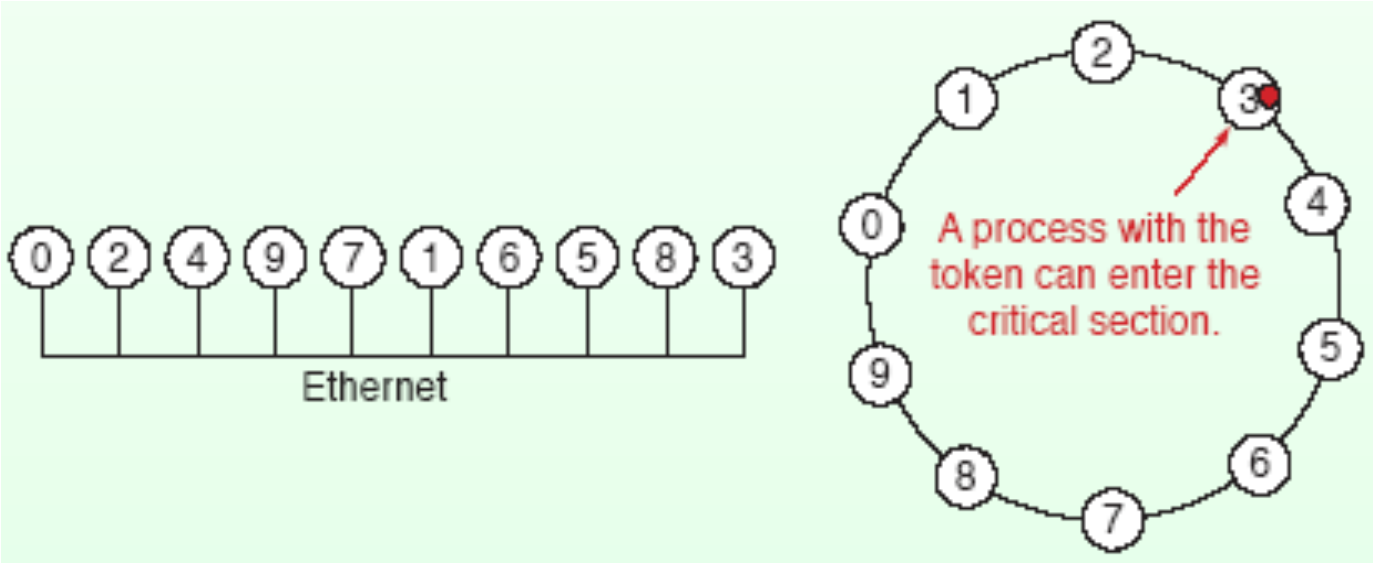
Centralised Algorithm



Distributed Algorithm

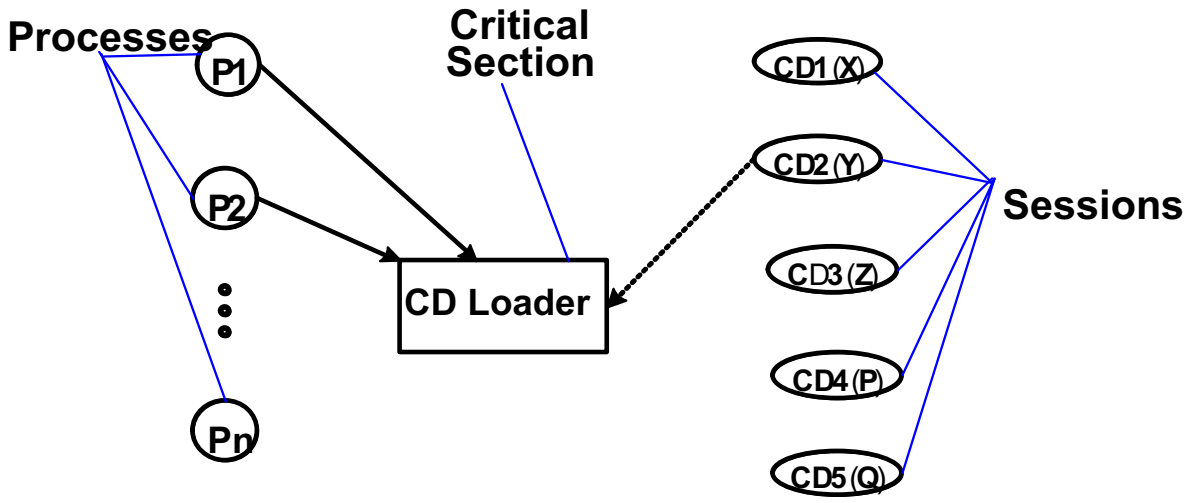


Token Ring Algorithm



Group Mutual Exclusion

In the group mutual exclusion problem, which generalizes mutual exclusion, processes choose ‘session’ when they want entry to the Critical Section (CS); processes are allowed to be in the CS simultaneously provided they request the same session.



Deadlock

DS Synchronization is difficult

- Four conditions have to be met for deadlock to be present:
  - **Mutual exclusion.** A resource can be held by at most one process
  - **Hold and wait.** Processes that already hold resources can wait for another resource.
  - **Non-preemption.** A resource, once granted, cannot be taken away from a process.
  - **Circular wait.** Two or more processes are waiting for resources held by one of the other processes.

Check Deadlock

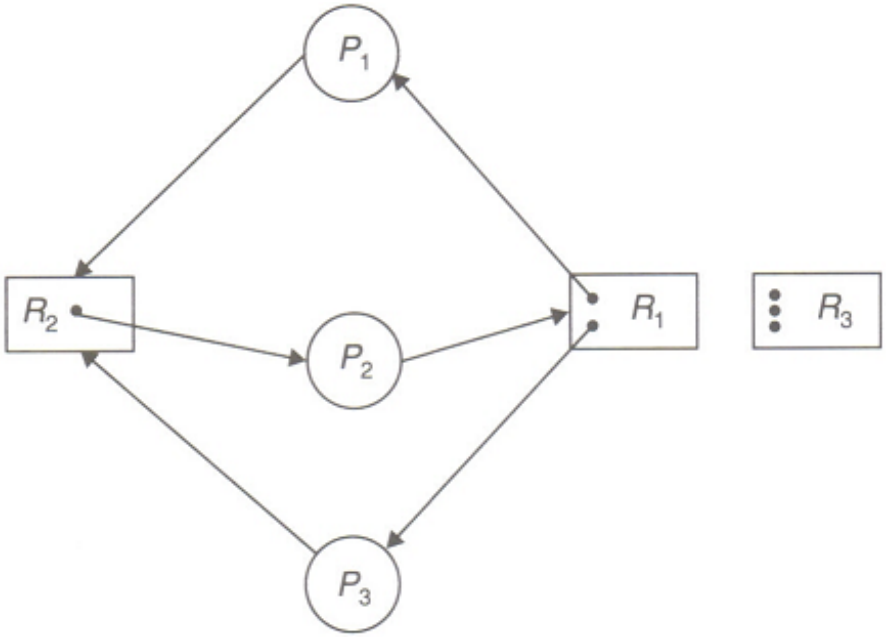
Cycles:

1. (P_1, R_2, P_2, R_1, P_1)
2. (P_3, R_2, P_2, R_1, P_3)

Knot:

1. {P_1, P_2, P_3, R_1, R_2}

Thus here is a deadlock !



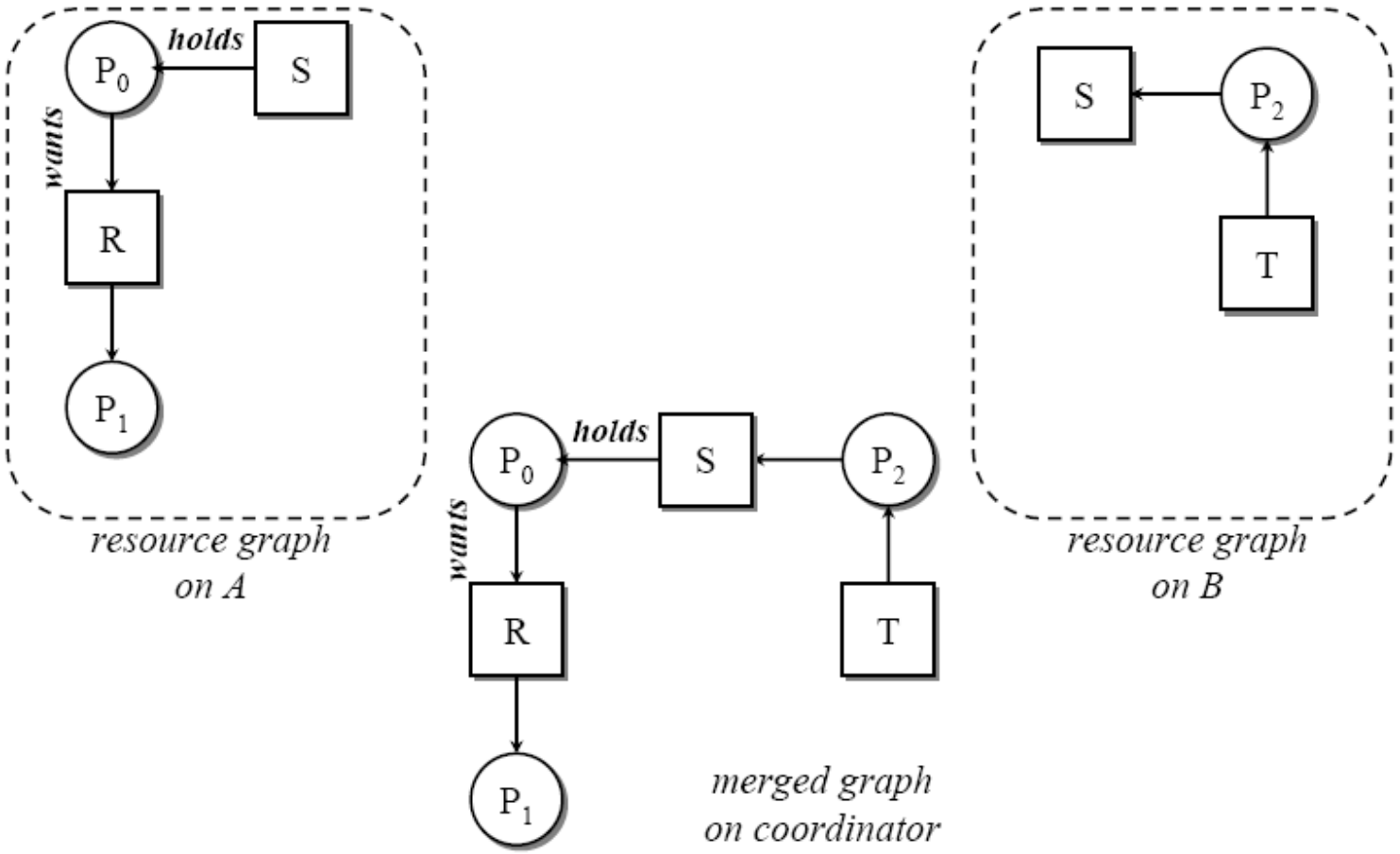
Ostrich Algorithm

Ignore the deadlock problem

Deadlock Detection in DS

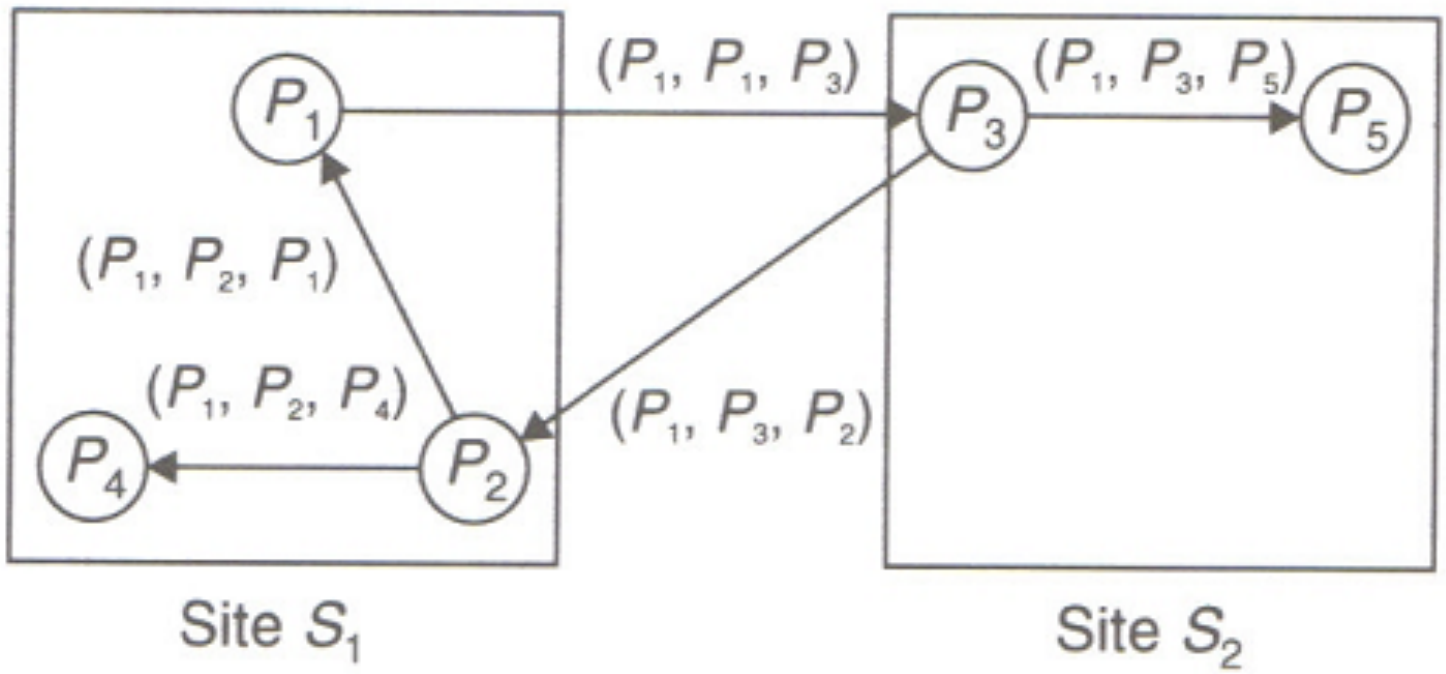
- Preventing or avoiding deadlocks can be difficult.
  - Detecting them is easier.
  - When deadlock is detected
    - kill off one or more processes
    - if system is based on atomic transactions, abort one or more transactions

Centralized deadlock detection Algorithm



A central coordinator maintains a graph for the entire system

Distributed: Chandy-Misra-Haas Algorithm



If a message goes all the way around and comes back to the original sender, a cycle exists

Recovery after Detection

- One of the following methods may be used
  - Asking for operator intervention
  - Terminating of process(es)
  - Rollback of process(es)
- Issues in recovery from deadlock
  - Minimization of recovery cost
  - Prevention of starvation

Lab Week 5 Overview

- Introduction to MPI

Tutorial Week 5 Overview

- Virtual topologies supported in MPI
- General coding pattern for an MPI application
- Difference between a blocking and non-blocking message passing

Next week: Election algorithms, Transactions, Concurrency

- Election algorithms, Transactions, Concurrency