FIT3143

Parallel Computing

Week 4: Parallel Computing In Distributed Memory - MPI

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	

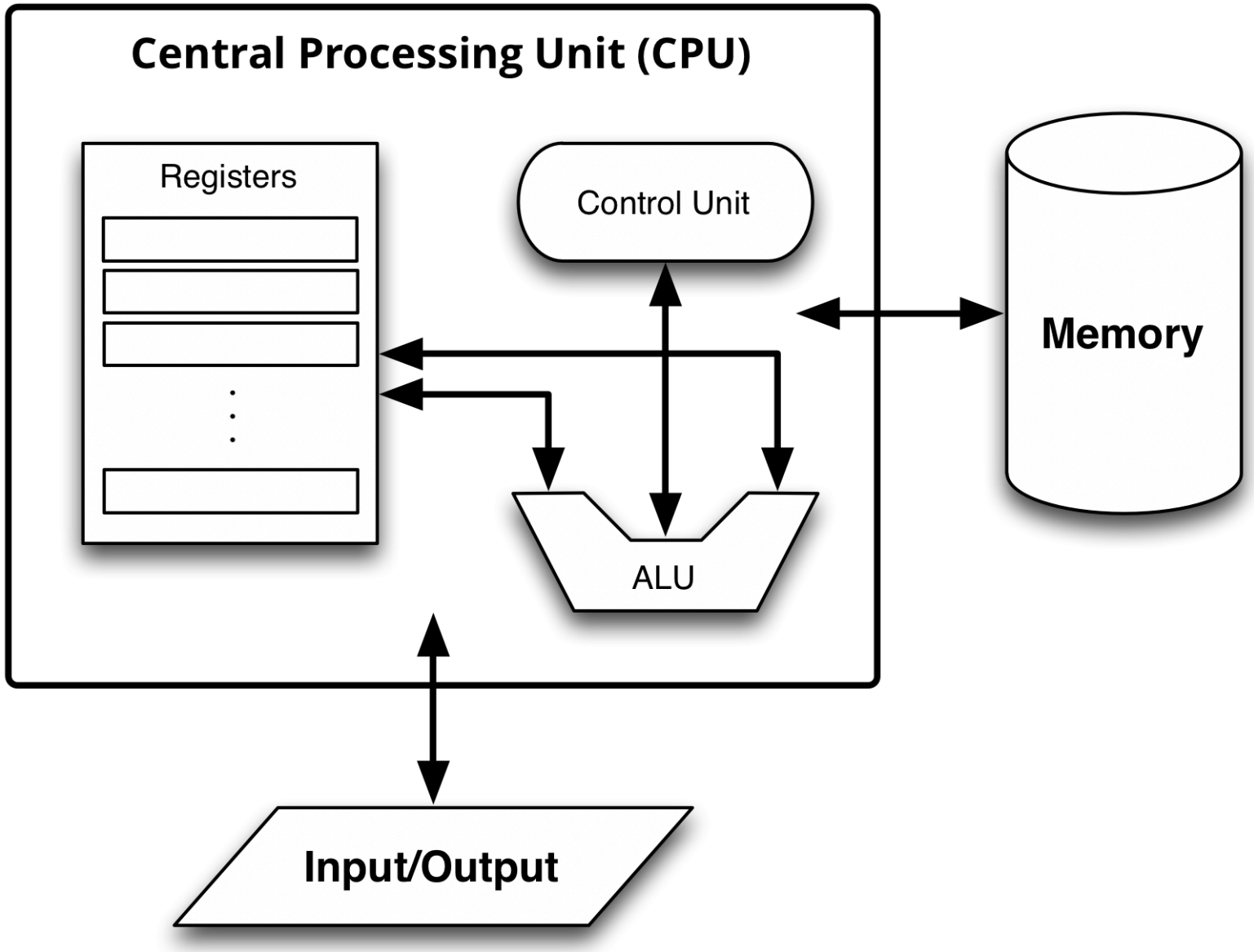
Assessment	Weight
Assignment 1	15%
Assignment 2	25%
Lab-work and Quizzes	10%
Final exam	50%

Feedback

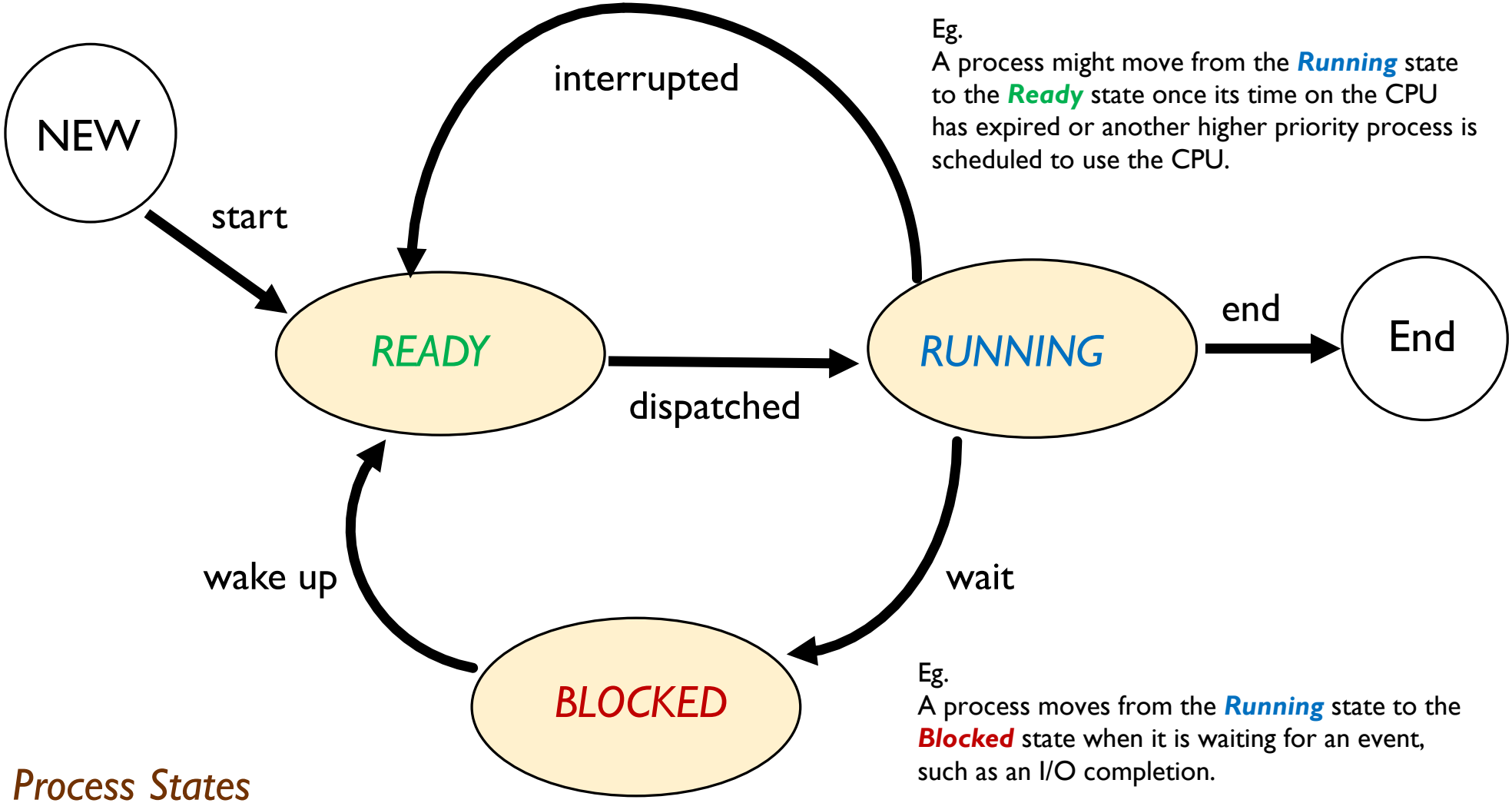
- Assignments
- Lecture
- Tutorial
- Lab
- Consultations

Please share your experience on online learning.

Background


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

Process States

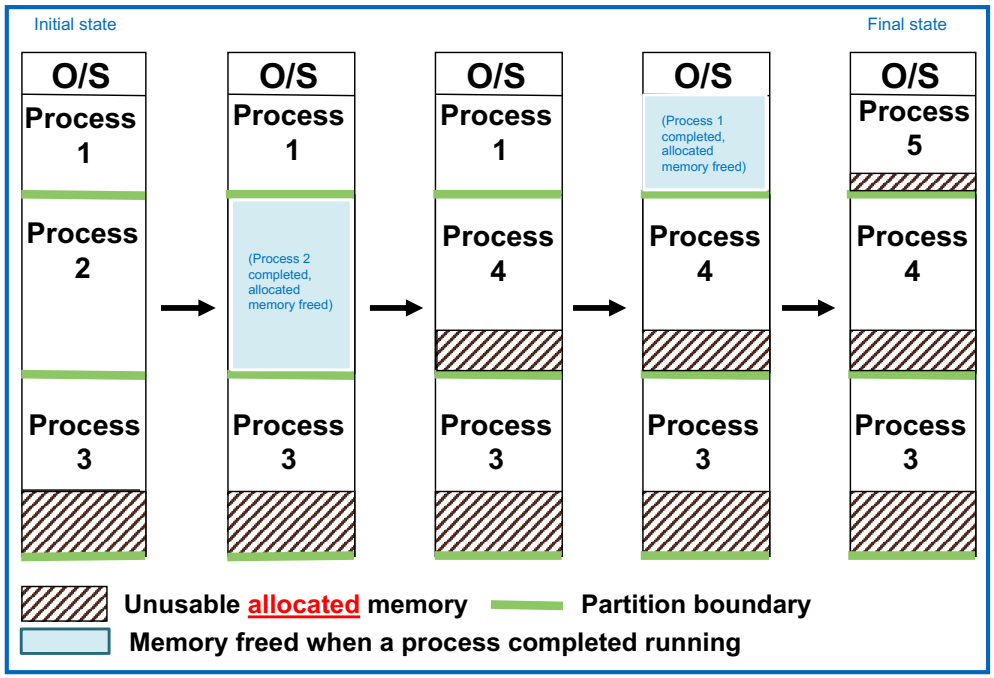


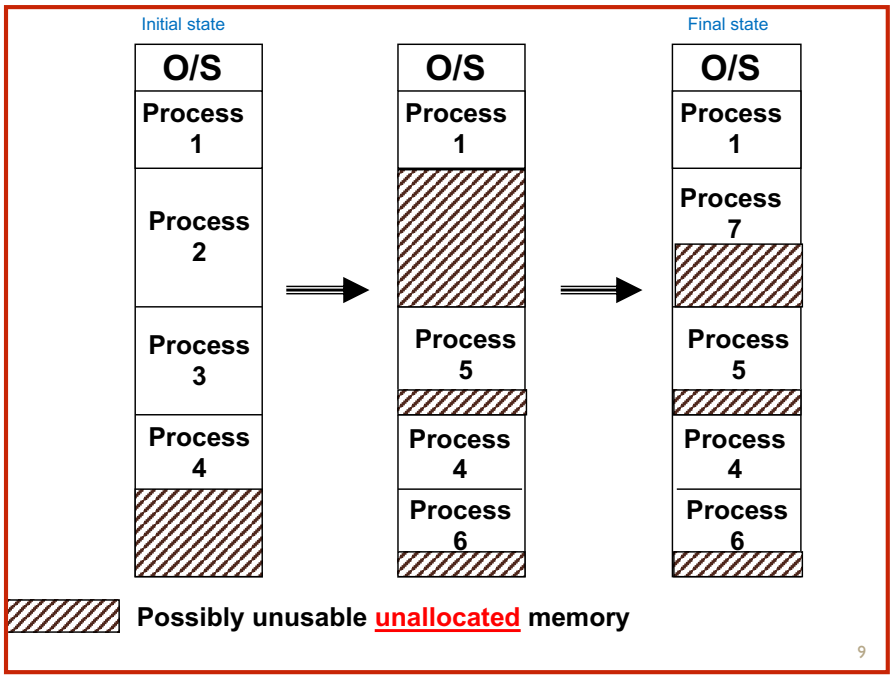
Eg.
A process might move from the ***Running*** state to the ***Ready*** state once its time on the CPU has expired or another higher priority process is scheduled to use the CPU.

Eg.
A process moves from the ***Running*** state to the ***Blocked*** state when it is waiting for an event, such as an I/O completion.

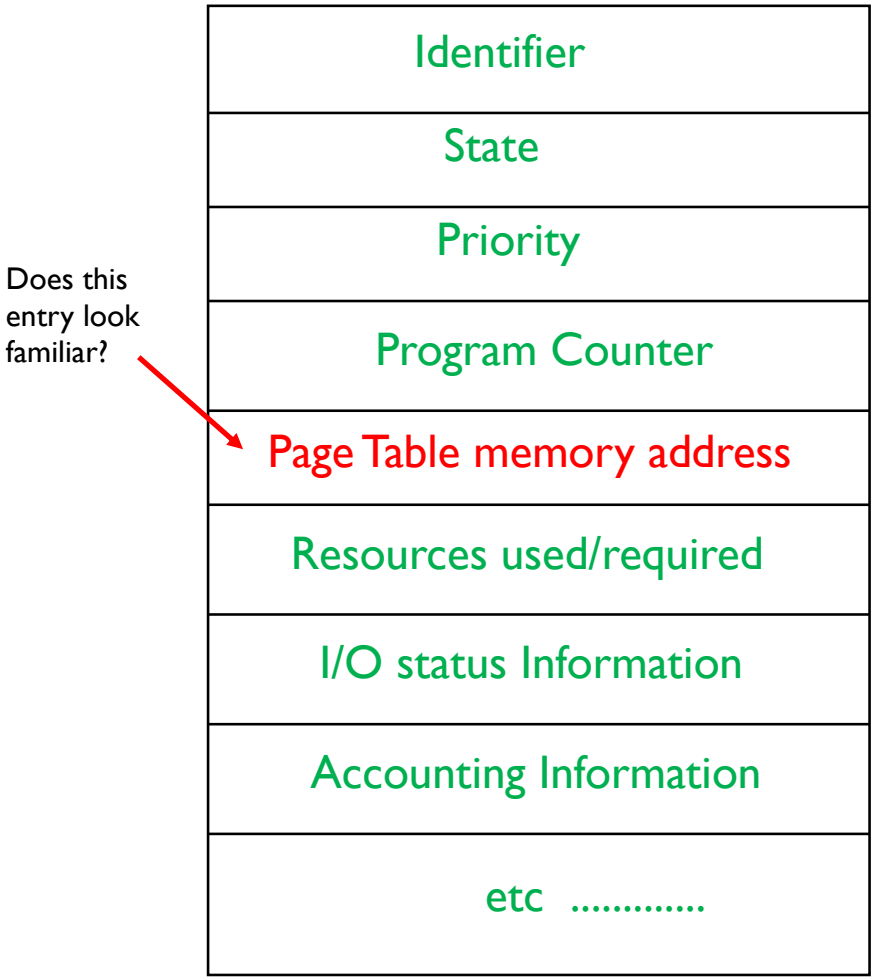
Process States

Memory Fragmentation

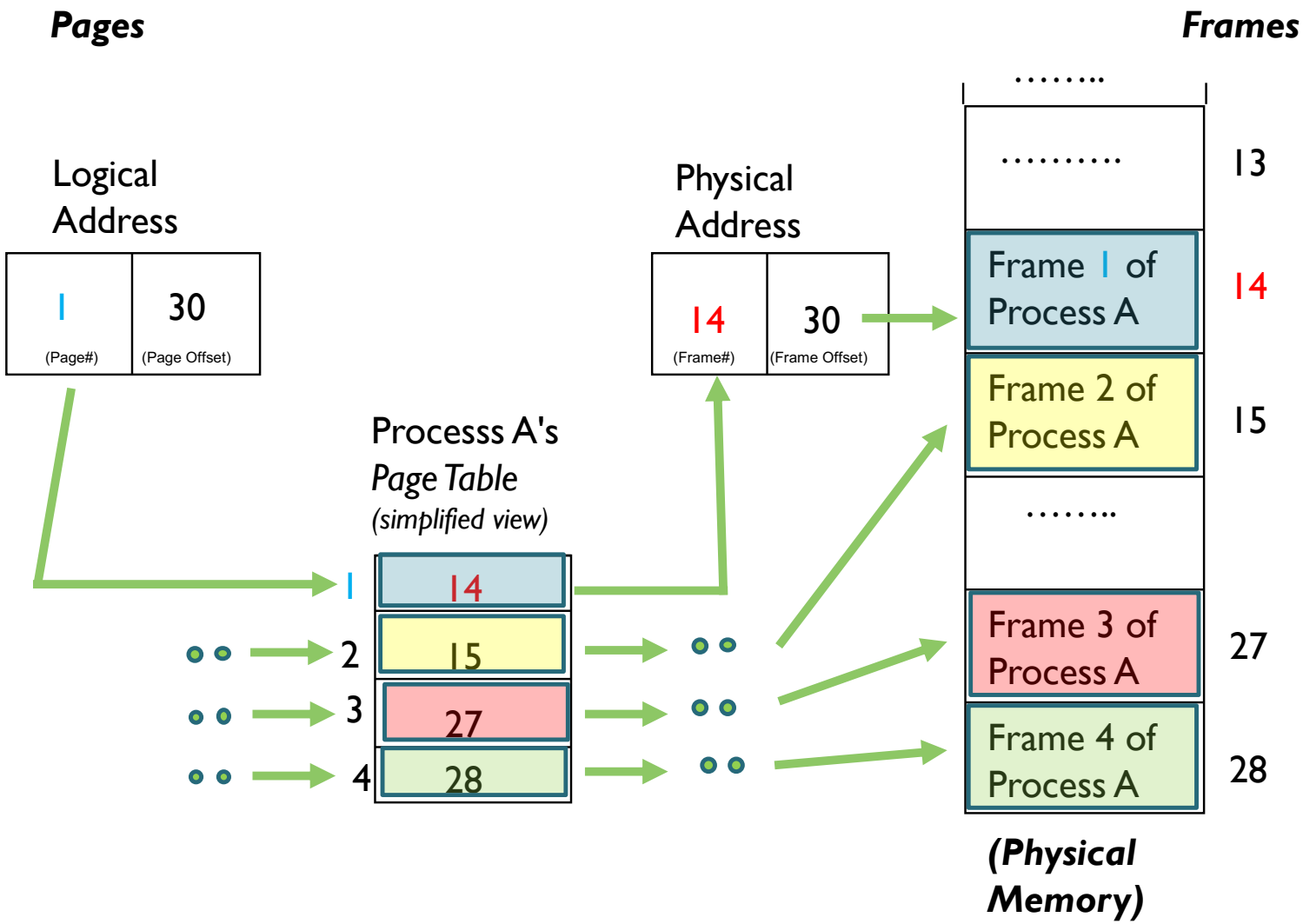




Process Control Block

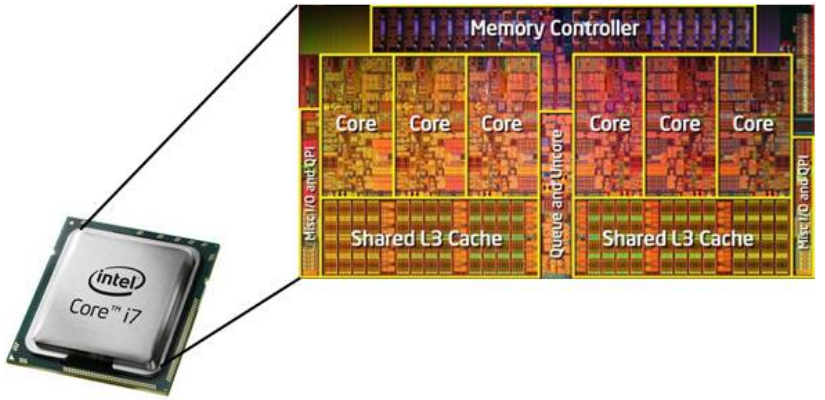


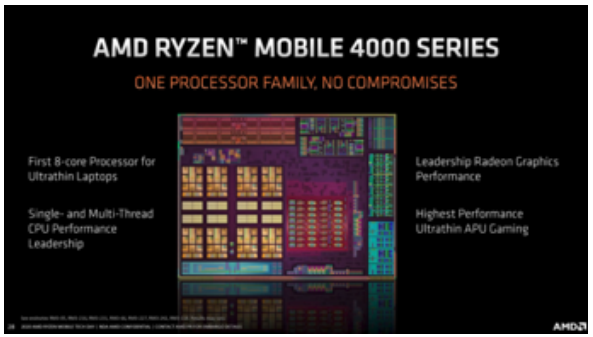
Virtual Memory

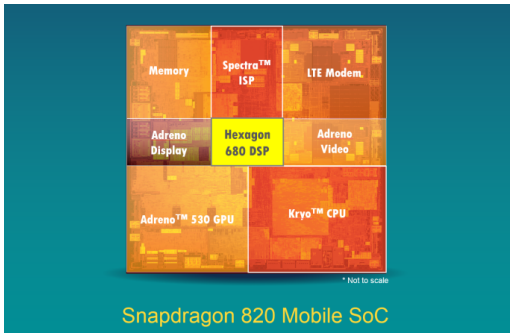


Parallel Computing (Week 1)

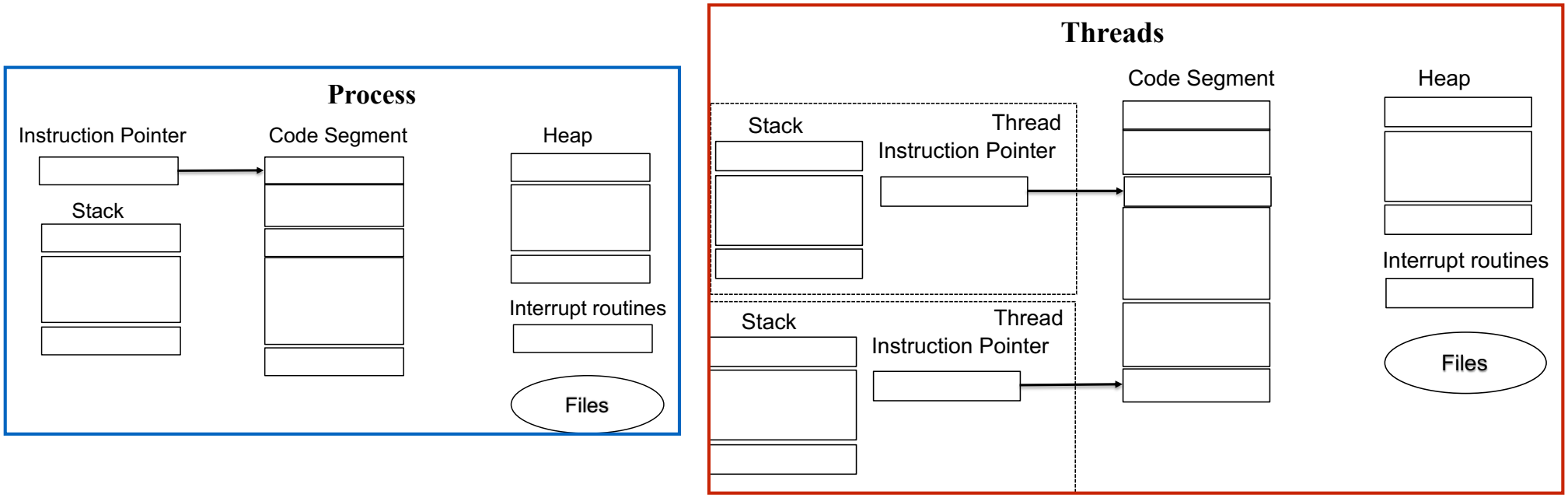
- More than one computer
- More than one CPUs
- More than one Cores
- Speedup s(n)?







Process vs Threads (Week 2)



- Shared memory (PThreads, OpenMP)

OpenMP

Serial Code

```
#include <stdio.h>

int main() {
        int i;
        printf("Hello World\n");

        for(i=0;i<6;i++)
                printf("Iter:%d\n",i);

        printf("GoodBye World\n");
}
```

Parallel Code

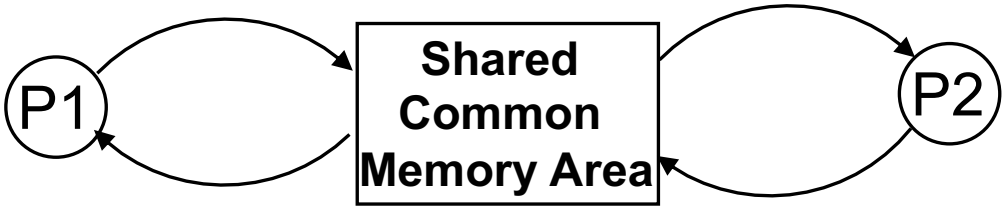
```
#include <stdio.h>
#include <omp.h>

int main() {
#pragma omp parallel
{
        int i;
        printf("Hello World\n");

#pragma omp for
        for(i=0;i<6;i++)
                printf("Iter:%d\n",i);

}
        printf("GoodBye World\n");
}
```

Today

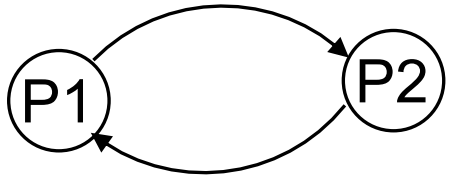
- Recap: Inter-Process Communications & RPC
- Distributed Memory Message Passing

Recap

IPC: Information sharing



Shared data approach



Message passing approach

Message Passing System

- A *message-passing system* is a sub-system of a distributed system that provides a set of message-based IPC protocols and does so by shielding the details of complex network protocols and multiple heterogeneous platforms from programmers.

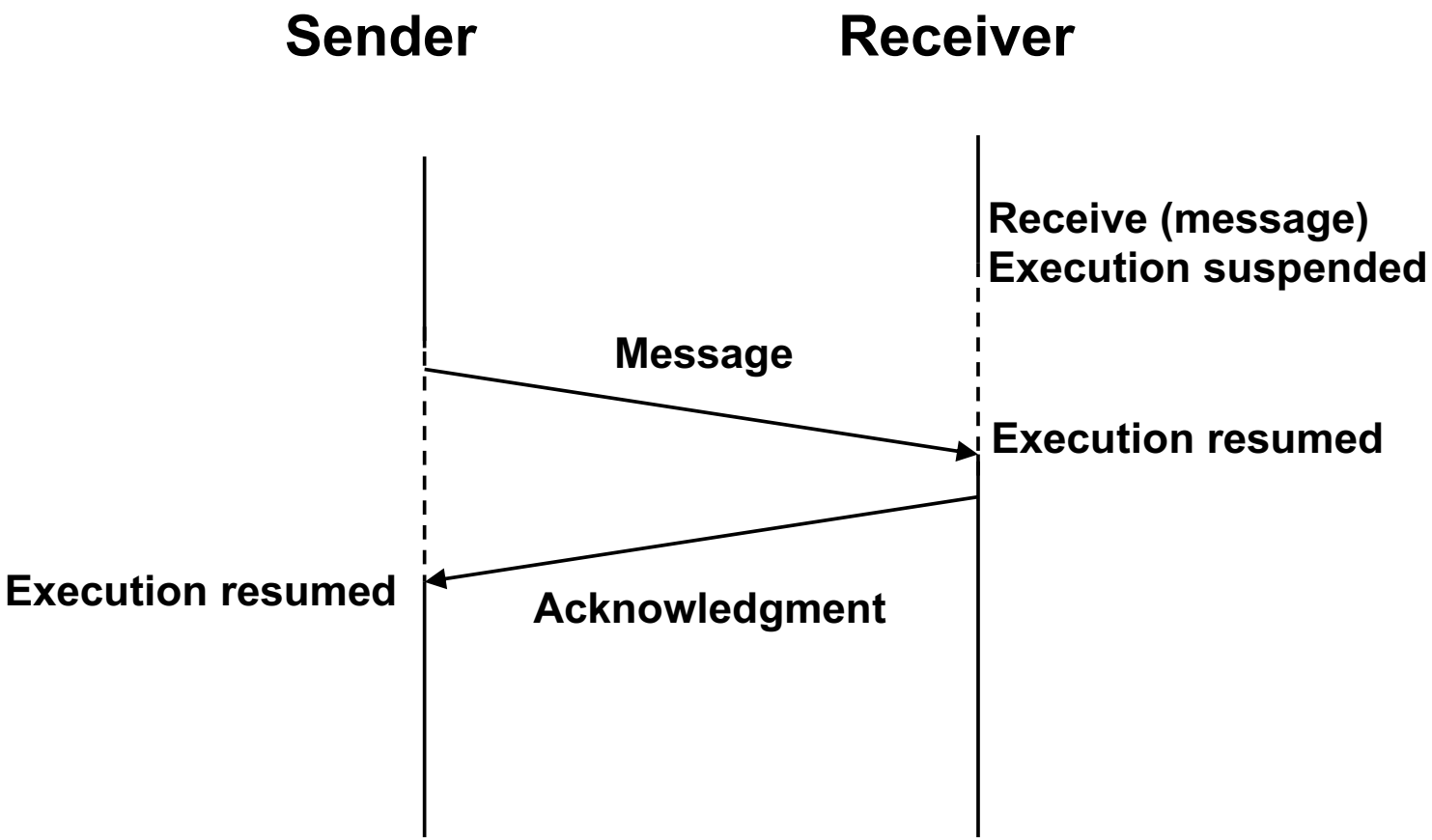
A typical message structure
–Header
  - Addresses
    - ✓Sender address
    - ✓Receiver address
  - Sequence number
  - Structural information
    - ✓Type
    - ✓Number of bytes

–Message

Issues

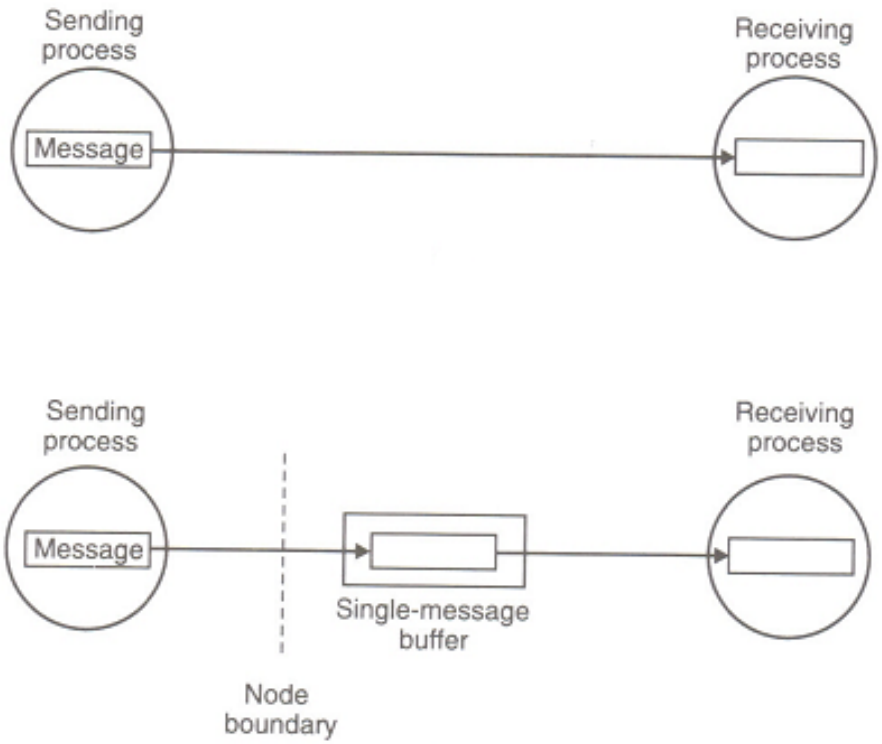
- Identity related
- Network Topology related
- Flow control related
- Error control and channel management

Synchronous & Asynchronous Communication

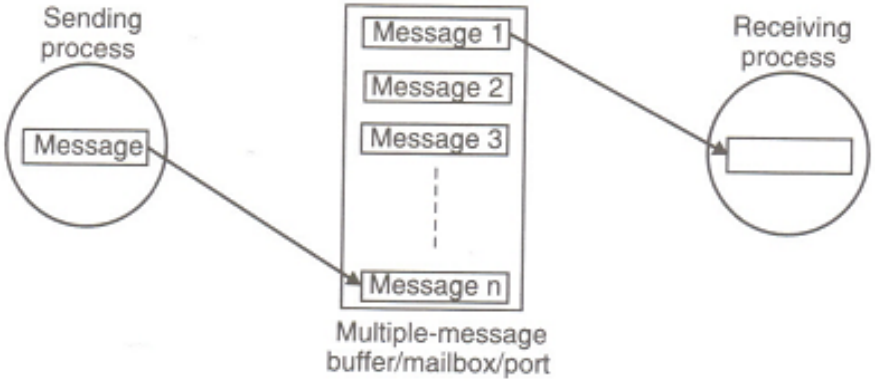


Synchronous mode with blocking send and receive primitives

Buffering



Synchronous System



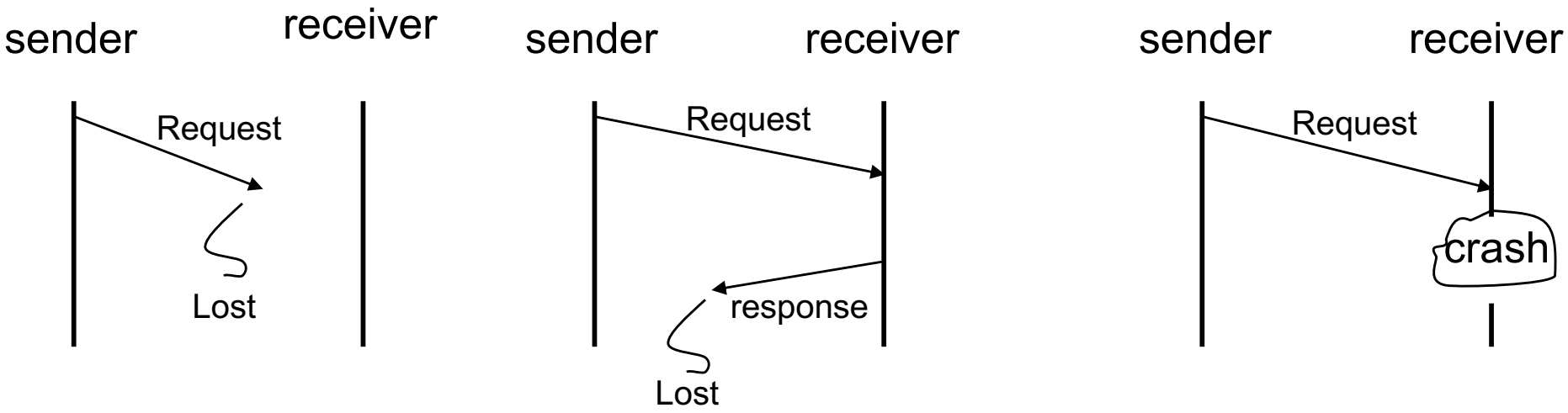
Asynchronous System

Encoding Decoding

- Encoding/Decoding is needed if sender and receiver have different architecture
- Even for Homogeneous Encoding/Decoding is needed for using an absolute pointer and to know which object is stored in where and how much storage it requires

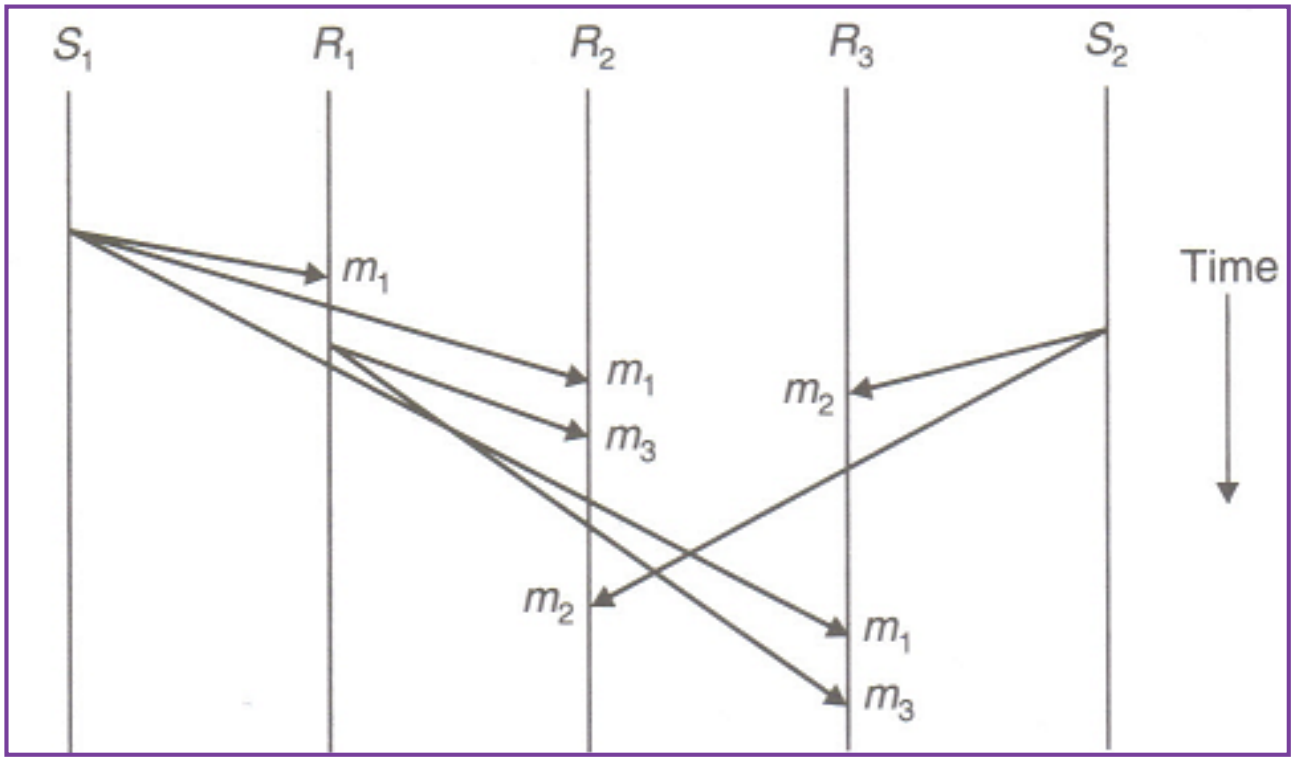
Failure handling

- Failure classification
  - Loss of request message
  - Loss of response message
  - Unsuccessful execution of the request

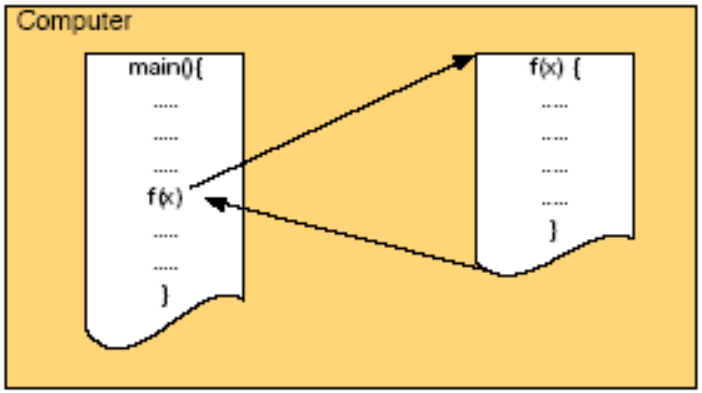


Group communication

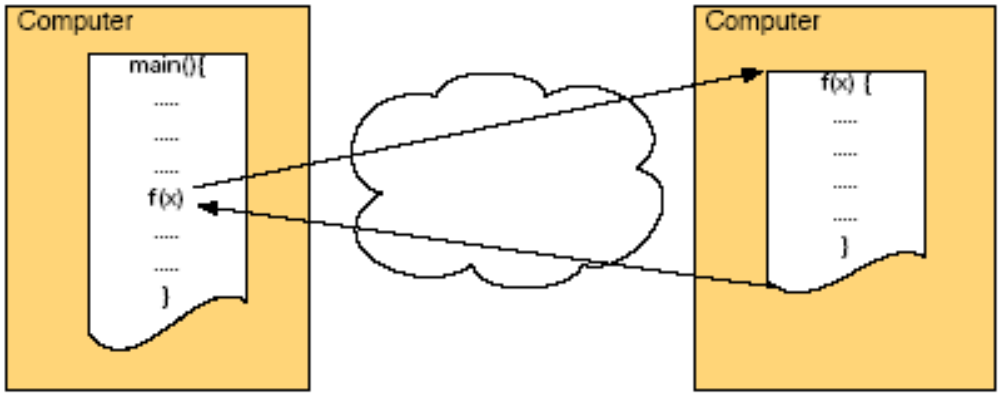
- *Ordered message delivery* is an important issue.
- No ordering
- Absolute
- Consistent
- Causal



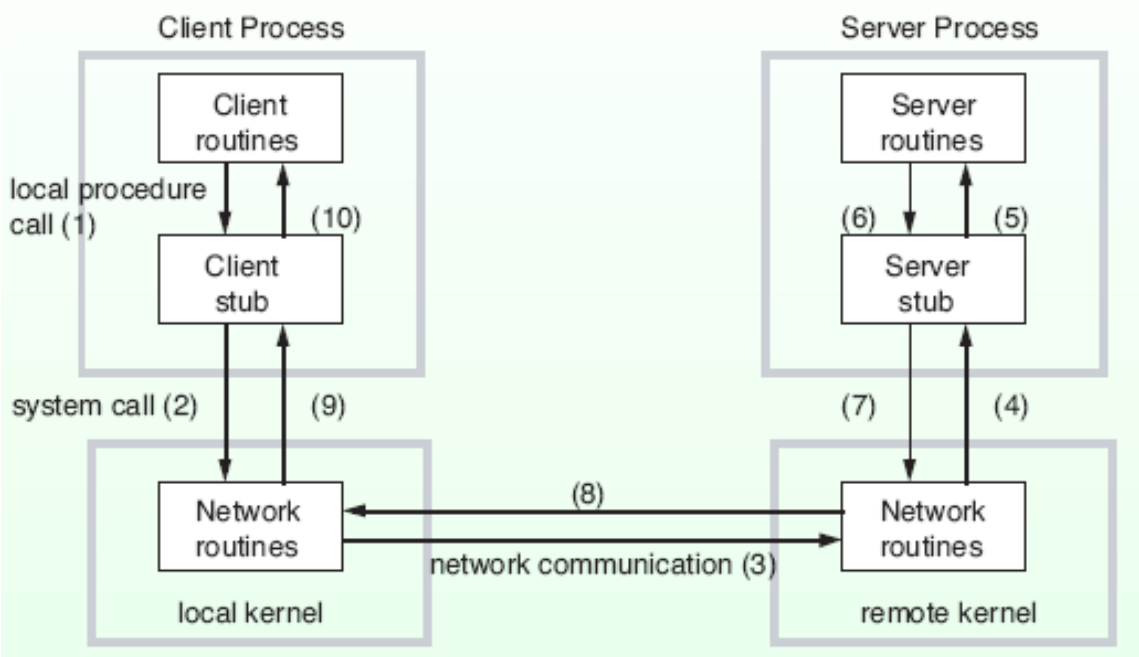
Local and Remote Procedure Call



Local Procedure Call



Remote Procedure Call



Concurrent Access to Multiple Servers

- One of the following three may be adopted:
  - Threads
  - Early reply approach [Wilbur and Bacarisse]
  - Call buffering approach [Gimson]

Serving Multiple Requests Simultaneously

- Following types of delays are common:
  - A server, during the course of a call execution, may wait for a shared resource
  - A server calls a remote function that involves computation or transmission delays
- So the server may accept and process other requests while waiting to complete a request.
- Multiple-threaded server may be a solution.

Message Passing Interface

MPI

- MPI is a ***specification*** for the developers and users of message passing libraries.
- The goal of the Message Passing Interface is to provide a widely used standard for writing message passing programs.

Why MPI

- **Standardization** - MPI is the only message passing library which can be considered a standard. It is supported on virtually all major platforms and many specialised HPC systems. Practically, it has replaced all previous message passing libraries.

- **Portability** - There is no need to modify your source code when you port your application to a different platform that supports (and is compliant with) the MPI standard.

- **Performance Opportunities** - Vendor implementations should be able to exploit native hardware features to optimize performance.

- **Functionality** - Over 115 routines are defined in MPI-1 alone.

- **Availability** - A variety of implementations are available, both vendor and public domain.

Programming Model

- MPI lends itself to virtually any distributed memory parallel programming model. In addition, MPI is commonly used to implement (behind the scenes) some shared memory models, such as Data Parallel, on distributed memory architectures.

- Hardware platforms:

  - Distributed Memory: Originally, MPI was targeted for distributed memory systems.

  - Shared Memory: As shared memory systems became more popular, particularly SMP / NUMA architectures, MPI implementations for these platforms appeared.

  - Hybrid: MPI is now used on just about any common parallel architecture including massively parallel machines, SMP clusters, workstation clusters and heterogeneous networks.

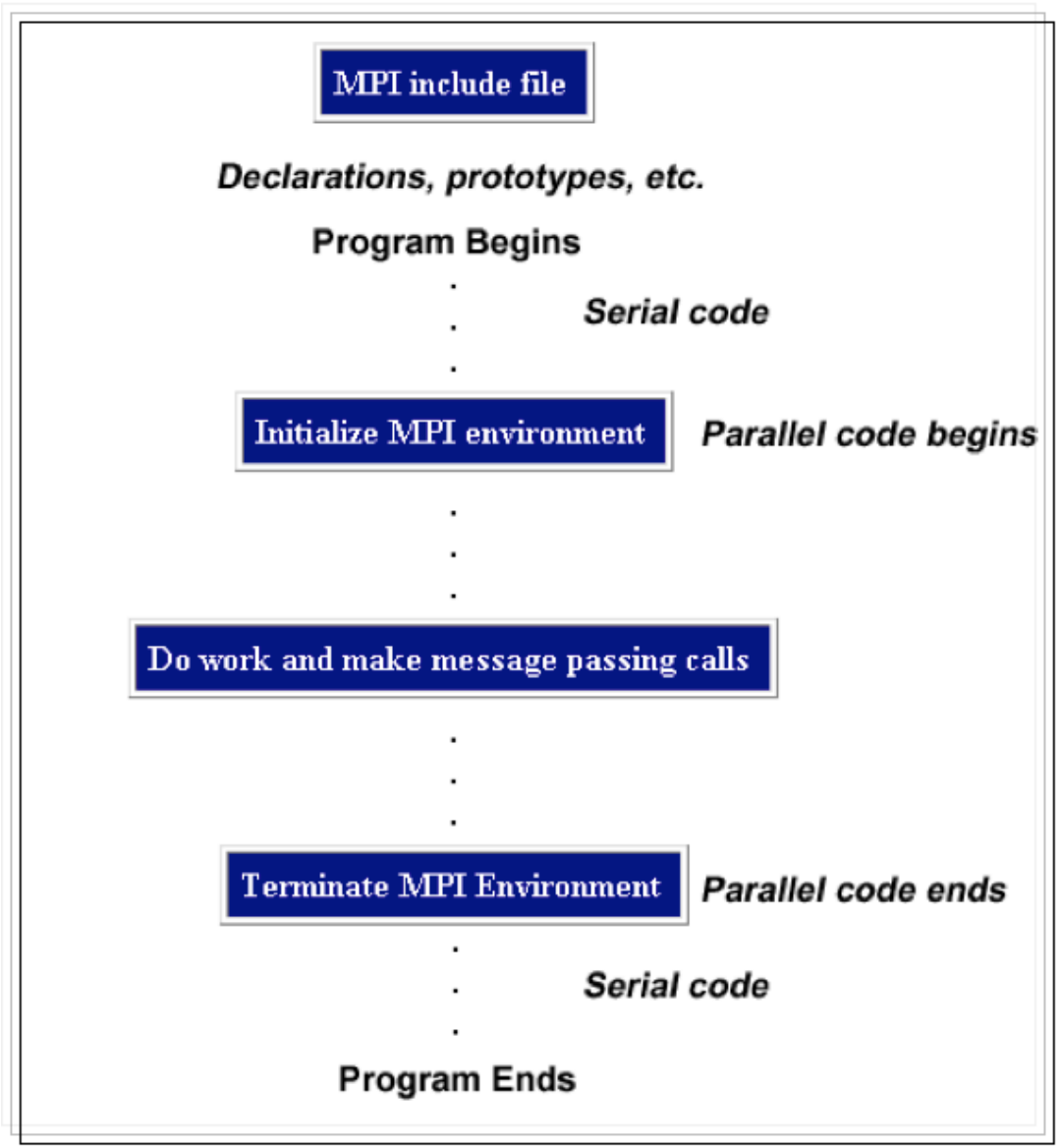
Programming Model

- All parallelism is explicit: the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs.
- The number of tasks dedicated to run a parallel program was static. New tasks couldn't not be dynamically spawned during run time. (MPI-2 addressed this issue).
- OpenMPI is an open source implementation of MPI (message-passing interface).

Getting Started

- MPI is native to ANSI C
  - C++ and Java bindings are available
    - MPI C++ classes – www.mcs.anl.gov
  - mpiJava API – www.hpjava.org
  - Python - https://mpi4py.readthedocs.io/en/stable/
- MPI versions
- MPI - C
  - Header File: Required for all programs/routines which make MPI library calls.
    - #include "mpi.h" Or #include <mpi.h>
  - Format of MPI Calls:
  - Format: rc = **MPI_X**xxxx(parameter, ...)
    - Example: rc = MPI_Bsend(&buf,count,type,dest,tag,comm)
    - **Error code: rc value is** set to MPI_SUCCESS if successful

General MPI Program Structure



MPI's "Hello World"

1. **Create Source Code File: hello.c**

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

int main(int argc, char *argv) {
    // Initialize the MPI environment
    MPI_Init(NULL, NULL);

    // Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Get the name of the processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    // Print off a hello world message
    printf("Hello world from processor %s, rank %d out of %d processors\n",
           processor_name, world_rank, world_size);

    // Finalize the MPI environment.
    MPI_Finalize();
}
```

2. **Compile: `mpicc hello.c -o hello-mp`**
3. **Execute: `mpirun -np 2 hello-mp`**

Communicators and Groups

- MPI uses objects called *communicators* and *groups* to define which collection of processes may communicate with each other. Most MPI routines require you to specify a communicator as an argument.
- Use MPI_COMM_WORLD whenever a communicator is required - it is the predefined communicator that includes all of your MPI processes.
- Rank:
  - Within a communicator, every process has its own unique, integer identifier assigned by the system when the process initializes. A rank is sometimes also called a "task ID". Ranks are contiguous and begin at zero.
  - Used by the programmer to specify the source and destination of messages. Often used conditionally by the application to control program execution (if rank=0 do this / if rank=1 do that etc).

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)

Environment Management Routines …

- MPI_Wtime ()
- MPI_Wtick ()
- MPI_Finalize ()

Environment Management Template …

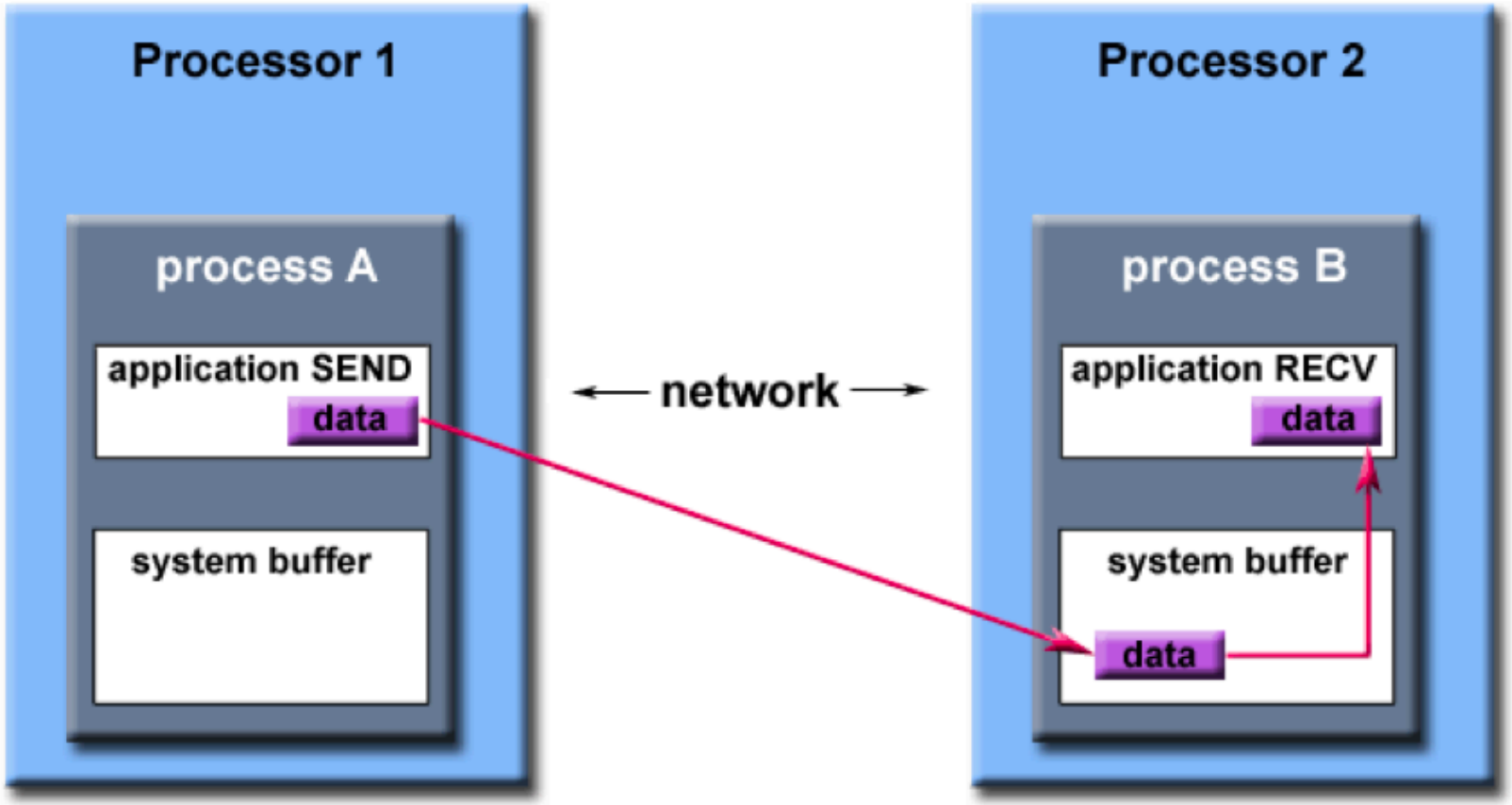
```
#include "mpi.h"
#include <stdio.h>

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

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.

P2P: System Buffer



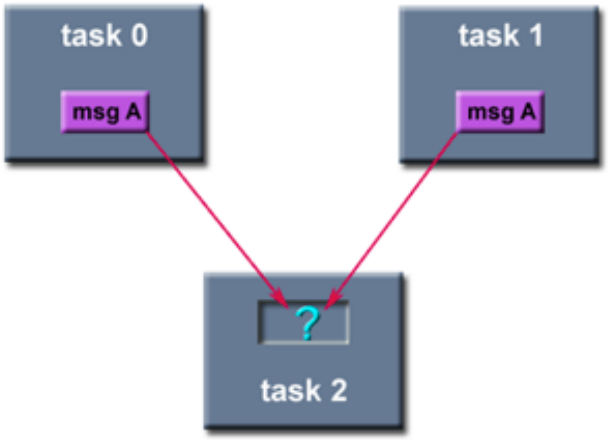
Path of a message buffered at the receiving process

P2P: Blocking & Non-blocking

- Blocking:
  - Send routine will only "return" after it is safe to modify the application buffer (your send data) for reuse.
  - Can be synchronous or asynchronous
  - Receive only "returns" after the data has arrived and is ready for use by the program.
- Non-blocking:
  - Send and receive do not wait for any communication events to complete, such as message copying from user memory to system buffer space or the actual arrival of message.
  - Perform the operation when it is able. The user can not predict when that will happen.
  - It is unsafe to modify the application buffer (your variable space) until you know for a fact the requested non-blocking operation was actually performed by the library. There are "wait" routines used to do this.

P2P: Order & Fairness

- Order:
  - MPI guarantees that messages will not overtake each other.
  - If a sender sends two messages (Message 1 and Message 2) in succession to the same destination, and both match the same receive, the receive operation will receive Message 1 before Message 2.
  - If a receiver posts two receives (Receive 1 and Receive 2), in succession, and both are looking for the same message, Receive 1 will receive the message before Receive 2.
  - Order rules do not apply if there are multiple threads participating in the communication operations.
- Fairness:
  - MPI does not guarantee fairness - it's up to the programmer to prevent "operation starvation".
  - Example: task 0 sends a message to task 2. However, task 1 sends a competing message that matches task 2's receive. Only one of the sends will complete.



P2P Routines and Arguments

- MPI Message Passing Routine Arguments

Blocking sends	**MPI_Send(buffer, count, type, dest, tag, comm)**
Non-blocking sends	**MPI_Isend(buffer, count, type, dest, tag, comm, request)**
Blocking receive	**MPI_Recv(buffer, count, type, source, tag, comm, status)**
Non-blocking receive	**MPI_Irecv(buffer, count, type, source, tag, comm, request)**

  - Type, Tag, Status, Request

Data Types

MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	8 binary digits
MPI_PACKED	data packed or unpacked with MPI_Pack()/ MPI_Unpack

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)

Example

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

int main(int argc, char *argv[])
{
int numtasks, rank, dest, source, rc, count, tag=1;
char inmsg, outmsg='x';
MPI_Status Stat;

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

if (rank == 0) {
  dest = 1;
  source = 1;
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
  }

else if (rank == 1) {
  dest = 0;
  source = 0;
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  }

rc = MPI_Get_count(&Stat, MPI_CHAR, &count);
printf("Task %d: Received %d char(s) from task %d with tag %d \n",
       rank, count, Stat.MPI_SOURCE, Stat.MPI_TAG);

MPI_Finalize();
}
```

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)

Example

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

int main(int argc, char *argv[])
{
int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
MPI_Request reqs[4];
MPI_Status stats[4];

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

prev = rank-1;
next = rank+1;
if (rank == 0)  prev = numtasks - 1;
if (rank == (numtasks - 1))  next = 0;

MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);

MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);

MPI_Waitall(4, reqs, stats);

printf("I am proc %d buf[0]=%d buf[1]=%d\n",rank,buf[0],buf[1]);

MPI_Finalize();
}
```

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

Derived Data Type Routines

Derived data type

- MPI predefines its primitive data types:
  - MPI also provides facilities for you to define your own data structures based upon sequences of the MPI primitive data types. Such user defined structures are called derived data types.
  - Primitive data types are contiguous. Derived data types allow you to specify non-contiguous data in a convenient manner and to treat it as though it was contiguous.
  - MPI provides several methods for constructing derived data types:
    - Contiguous
    - Vector
    - Indexed
    - Struct

MPI_CHAR
MPI_SHORT
MPI_INT
MPI_LONG
MPI_UNSIGNED_CHAR
MPI_UNSIGNED_SHORT
MPI_UNSIGNED_LONG
MPI_UNSIGNED
MPI_FLOAT
MPI_DOUBLE
MPI_LONG_DOUBLE
MPI_BYTE
MPI_PACKED

Derived Data Type Routines

- MPI_Type_contiguous (count,oldtype,&newtype)
- MPI_Type_vector (count,blocklength,stride,oldtype,&newtype)
- MPI_Type_indexed (count,blocklens[],offsets[],old_type,&newtype)
- MPI_Type_struct (count,blocklens[],offsets[],old_types,&newtype)
- MPI_Type_extent (datatype,&extent)
- MPI_Type_commit (&datatype)
- MPI_Type_free (&datatype)

Example: Contiguous

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

int main(int argc, char *argv[])

{
int numtasks, rank, source=0, dest, tag=1, i;
float a[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 b[SIZE];

MPI_Status stat;
MPI_Datatype rowtype;

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

MPI_Type_contiguous(SIZE, MPI_FLOAT, &rowtype);
MPI_Type_commit(&rowtype);

if (numtasks == SIZE) {
  if (rank == 0) {
     for (i=0; i<numtasks; i++)
       MPI_Send(&a[i][0], 1, rowtype, i, tag, MPI_COMM_WORLD);
     }

  MPI_Recv(b, SIZE, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &stat);
  printf("rank= %d  b= %3.1f %3.1f %3.1f %3.1f\n",
         rank,b[0],b[1],b[2],b[3]);
  }
else
  printf("Must specify %d processors. Terminating.\n",SIZE);

MPI_Finalize();
}
```

Example: Vector

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

int main(int argc, char *argv[])
{
int numtasks, rank, source=0, dest, tag=1, i;
float a[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 b[SIZE];

MPI_Status stat;
MPI_Datatype columntype;

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

MPI_Type_vector(SIZE, 1, SIZE, MPI_FLOAT, &columntype);
MPI_Type_commit(&columntype);

if (numtasks == SIZE) {
  if (rank == 0) {
     for (i=0; i<numtasks; i++)
       MPI_Send(&a[0][i], 1, columntype, i, tag, MPI_COMM_WORLD);
        }

  MPI_Recv(b, SIZE, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &stat);
  printf("rank= %d  b= %3.1f %3.1f %3.1f %3.1f\n",
        rank,b[0],b[1],b[2],b[3]);
  }
else
  printf("Must specify %d processors. Terminating.\n",SIZE);

MPI_Finalize();
}
```

Example: Indexed

```
#include "mpi.h"
#include <stdio.h>
#define NELEMENTS 6

int main(int argc, char *argv[])
{
int numtasks, rank, source=0, dest, tag=1, i;
int blocklengths[2], displacements[2];
float a[16] =
  {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 b[NELEMENTS];

MPI_Status stat;
MPI_Datatype indextype;

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

blocklengths[0] = 4;
blocklengths[1] = 2;
displacements[0] = 5;
displacements[1] = 12;

MPI_Type_indexed(2, blocklengths, displacements, MPI_FLOAT, &indextype);
MPI_Type_commit(&indextype);

if (rank == 0) {
  for (i=0; i<numtasks; i++)
     MPI_Send(a, 1, indextype, i, tag, MPI_COMM_WORLD);
  }

MPI_Recv(b, NELEMENTS, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &stat);
printf("rank= %d  b= %3.1f %3.1f %3.1f %3.1f %3.1f %3.1f\n",
     rank,b[0],b[1],b[2],b[3],b[4],b[5]);

MPI_Finalize();
}
```

Group and Communicator Management

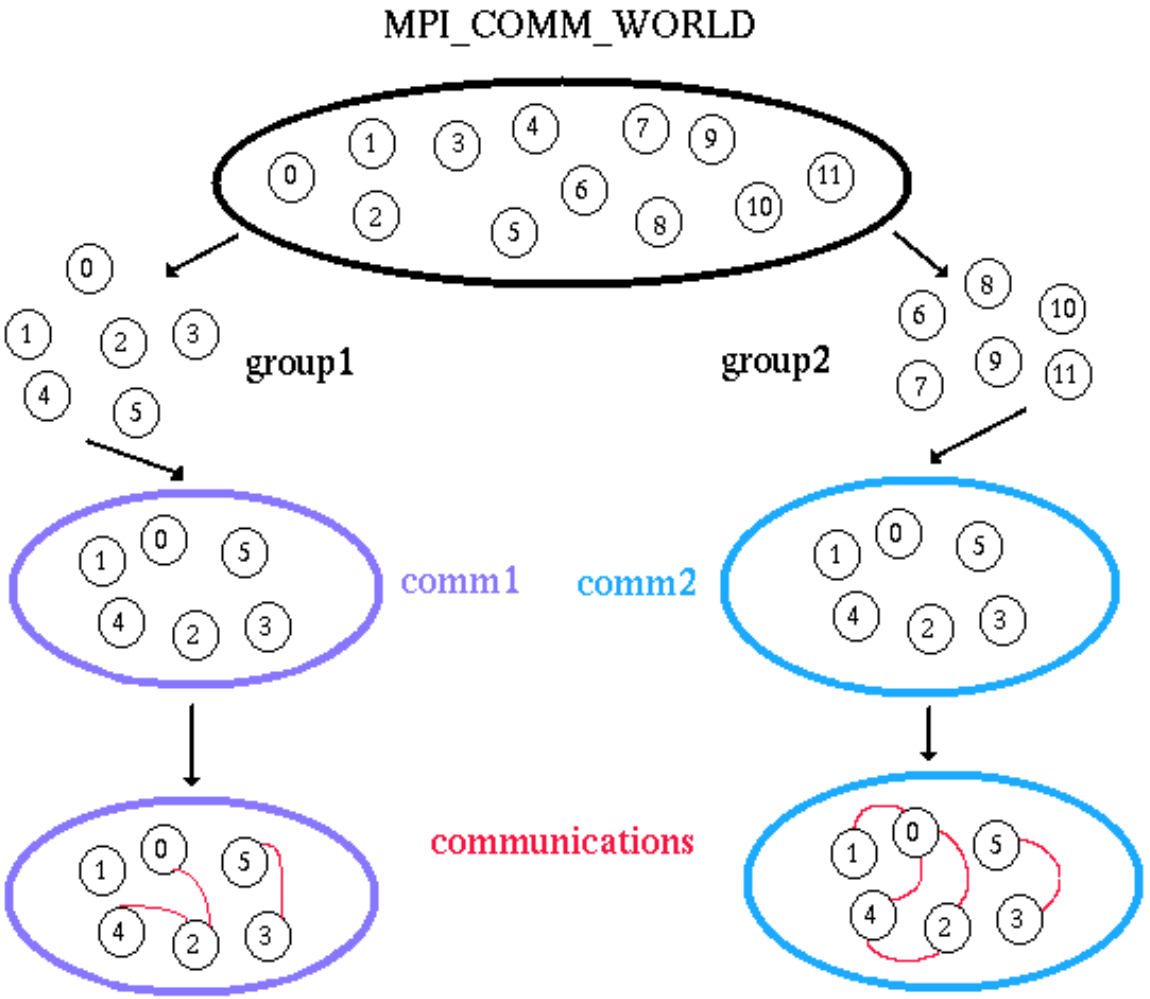
Groups vs Communicators

- A group is an ordered set of processes.
  - Each process in a group is associated with a unique integer rank. Rank values start at zero and go to N-1, where N is the number of processes in the group.
  - In MPI, a group is represented within system memory as an object. It is accessible to the programmer only by a "handle".
  - A group is always associated with a communicator object.
- A communicator encompasses a group of processes that may communicate with each other. All MPI messages must specify a communicator. In the simplest sense, the communicator is an extra "tag" that must be included with MPI calls. Like groups, communicators are represented within system memory as objects and are accessible to the programmer only by "handles". For example, the handle for the communicator that comprises all tasks is MPI_COMM_WORLD.
- From the programmer's perspective, a group and a communicator are one. The group routines are primarily used to specify which processes should be used to construct a communicator.

Restrictions

- Programming Considerations and Restrictions:
  - Groups/communicators are dynamic - they can be created and destroyed during program execution.
  - Processes may be in more than one group/communicator. They will have a unique rank within each group/communicator.
  - MPI provides over 40 routines related to groups, communicators, and virtual topologies.
- Typical usage:
  - Extract handle of global group from MPI_COMM_WORLD using MPI_Comm_group
  - Form new group as a subset of global group using MPI_Group_incl
  - Create new communicator for new group using MPI_Comm_create
  - Determine new rank in new communicator using MPI_Comm_rank
  - Conduct communications using any MPI message passing routine
  - When finished, free up new communicator and group (optional) using MPI_Comm_free and MPI_Group_free

Group and Communicator Management Routines



MPI Group Routines

- Group Accessors
- Group Constructors
- Group Destructors
- Communicator Accessors
- Communicator Destructors

Virtual Topologies

Virtual Topologies

- Convenience
  - Virtual topologies may be useful for applications with specific communication patterns - patterns that match an MPI topology structure.
  - For example, a Cartesian topology might prove convenient for an application that requires 4-way nearest neighbor communications for grid based data.
- Communication Efficiency
  - Some hardware architectures may impose penalties for communications between successively distant "nodes".
  - A particular implementation may optimize process mapping based upon the physical characteristics of a given parallel machine.
  - The mapping of processes into an MPI virtual topology is dependent upon the MPI implementation, and may be totally ignored.

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)

MPI Implementations

- IBM MPI library
- Quadrics MPI
- MPICH
- MVAPICH
- MPICH2
- Open MPI

Lab Week 4 Overview

- Problem solving using OpenMP

Tutorial Week 4 Overview

- MPI
- Virtual topologies
- Blocking and non-blocking message passing

Next week: Synchronization, MUTEX, Deadlocks

- Synchronization, MUTEX, Deadlocks