ENCE360 Operating Systems

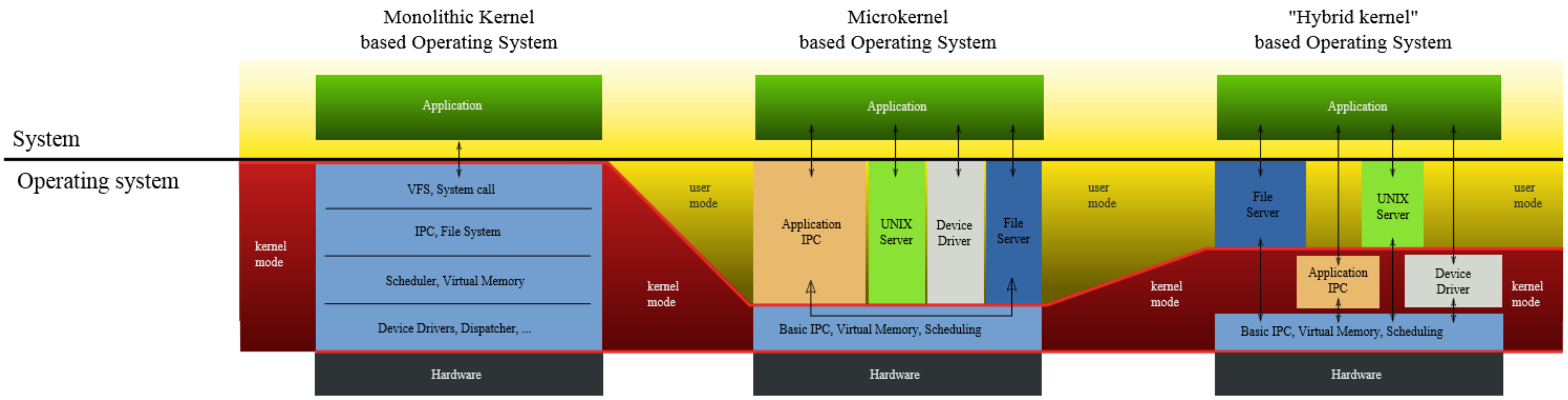
Distributed Operating Systems

Distributed Operating System

- A distributed operating system is a collection of independent, networked, communicating, and physically separate computational nodes.
- Each individual node holds a specific subset of the global aggregate OS with two distinct parts:
  - **microkernel**: ubiquitous minimal kernel that directly controls that node’s hardware
  - **management**: higher-level collection of system management components that coordinate the node's individual and collaborative activities

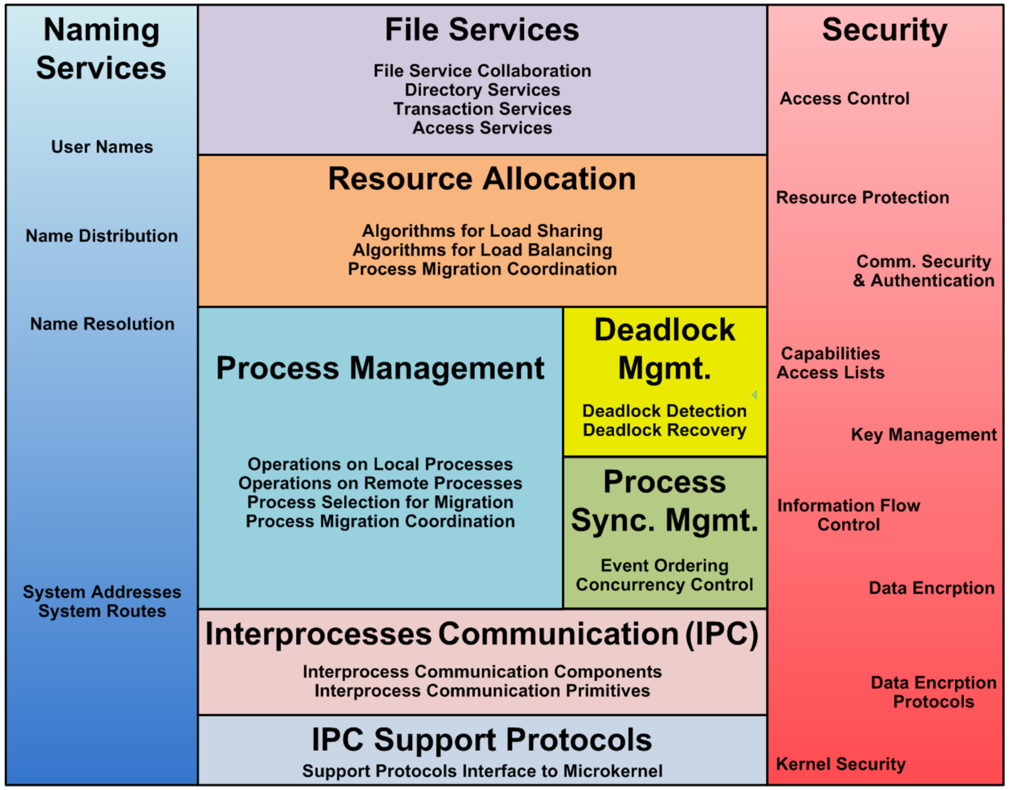
Distributed Operating System

- In a distributed OS, the kernel often supports a minimal set of functions, including low-level address space management, thread management, and inter-process communication.
- A kernel of this design is referred to as a **microkernel**. Its modular nature enhances reliability and security, essential features for a distributed OS.


Monolithic Kernel based Operating System
Microkernel based Operating System
"Hybrid kernel" based Operating System
System
Operating system
Application
VFS, System call
IPC, File System
Scheduler, Virtual Memory
Device Drivers, Dispatcher, ...
Hardware
kernel mode
user mode
Application IPC
UNIX Server
Device Driver
File Server
Basic IPC, Virtual Memory, Scheduling

Distributed Operating System

- System **management** components are software processes that define the node's policies. These components are the part of the OS outside the kernel.
- These components provide higher-level communication, process and resource management, reliability, performance and security.



ENCE360 Operating Systems

Distributed Systems

Distributed Systems

- Advantages of Distributed Systems
- Types of Network-Based Operating Systems
- Network Structure
- Communication Structure
- Communication Protocols
- An Example: TCP/IP
- Robustness
- Design Issues
- Distributed File System

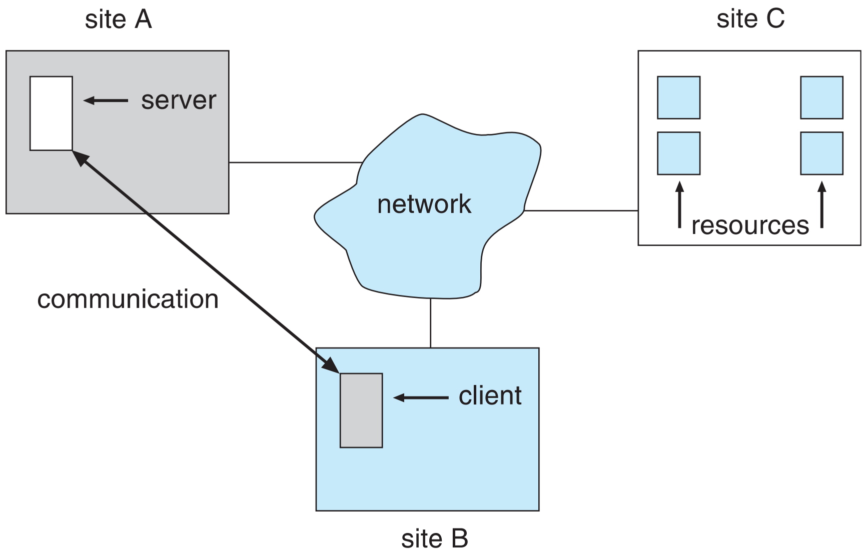
Objectives

- To provide a high-level overview of distributed systems and the networks that interconnect them
- To discuss the general structure of distributed operating systems
- To explain general communication structure and communication protocols
- To describe issues concerning the design of distributed systems

Overview

Distributed system is collection of loosely coupled processors that do not share memory or a clock, interconnected by a communications network or high speed buses

- Processors variously called ***nodes, computers, machines, hosts***
  - ***Site*** is location of the processor
  - Generally a ***server*** has a resource a ***client*** node at a different site wants to use



Reasons for Distributed Systems

- Reasons for distributed systems
  - **Resource sharing**
    - Sharing and printing files at remote sites
    - Processing information in a distributed database
    - Using remote specialised hardware devices
  - **Computation speedup** – **load sharing** or **job migration**
  - Reliability – detect and recover from site failure, function transfer, reintegrate failed site
  - Communication – **message** passing
    - All higher-level functions of a standalone system can be expanded to encompass a distributed system
  - Computers can be downsized, more flexibility, better user interfaces and easier maintenance by moving from large system to multiple smaller systems performing distributed computing

Types of Distributed Operating Systems

- Network Operating Systems
- Distributed Operating Systems

Network-Operating Systems

- Users are aware of multiplicity of machines
- Access to resources of various machines is done explicitly by:
  - Remote logging into the appropriate remote machine (telnet, ssh)
  - Remote Desktop (Microsoft Windows)
  - Transferring data from remote machines to local machines, via the File Transfer Protocol (FTP) mechanism
- Users must change paradigms – establish a **session**, give network-based commands
  - More difficult for users

Distributed-Operating Systems

- Users not aware of multiplicity of machines
  - Access to remote resources similar to access to local resources
- **Data Migration** – transfer data by transferring entire file, or transferring only those portions of the file necessary for the immediate task
- **Computation Migration** – transfer the computation, rather than the data, across the system
  - Via remote procedure calls (RPCs)
  - or via messaging system

Distributed-Operating Systems (Cont.)

- Process Migration – execute an entire process, or parts of it, at different sites
  - **Load balancing** – distribute processes across network to even the workload
  - **Computation speedup** – subprocesses can run concurrently on different sites
  - **Hardware preference** – process execution may require specialised processor
  - **Software preference** – required software may be available at only a particular site
  - **Data access** – run process remotely, rather than transfer all data locally
- Consider the World Wide Web

Robustness

- Failure detection
- Reconfiguration

Failure Detection

- Detecting hardware failure is difficult
- To detect a link failure, a **heartbeat** protocol can be used
- Assume Site A and Site B have established a link
  - At fixed intervals, each site will exchange an *I-am-up* message indicating that they are up and running
- If Site A does not receive a message within the fixed interval, it assumes either (a) the other site is not up or (b) the message was lost
- Site A can now send an *Are-you-up?* message to Site B
- If Site A does not receive a reply, it can repeat the message or try an alternate route to Site B

Failure Detection (Cont.)

- If Site A does not ultimately receive a reply from Site B, it concludes some type of failure has occurred
- Types of failures:
  - Site B is down
  - The direct link between A and B is down
  - The alternate link from A to B is down
  - The message has been lost
- However, Site A cannot determine exactly **why** the failure has occurred

Reconfiguration

- When Site A determines a failure has occurred, it must reconfigure the system:
  1. If the link from A to B has failed, this must be broadcast to every site in the system
  2. If a site has failed, every other site must also be notified indicating that the services offered by the failed site are no longer available
- When the link or the site becomes available again, this information must again be broadcast to all other sites

Design Issues

- **Transparency** – the distributed system should appear as a conventional, centralised system to the user
- **Fault tolerance** – the distributed system should continue to function in the face of failure
- **Scalability** – as demands increase, the system should easily accept the addition of new resources to accommodate the increased demand
  - Consider **Hadoop** open source programming framework for processing large datasets in distributed environments (based on Google search indexing)
- **Clusters** – a collection of semi-autonomous machines that acts as a single system

Distributed File System

- **Distributed file system** (**DFS**) – a distributed implementation of the classical time-sharing model of a file system, where multiple users share files and storage resources
- A DFS manages set of dispersed storage devices
- Overall storage space managed by a DFS is composed of different, remotely located, smaller storage spaces
- There is usually a correspondence between constituent storage spaces and sets of files
- Challenges include:
  - Naming and Transparency
  - Remote File Access

DFS Structure

- **Service** – software entity running on one or more machines and providing a particular type of function to a priori unknown clients
- **Server** – service software running on a single machine
- **Client** – process that can invoke a service using a set of operations that forms its client interface
- A client interface for a file service is formed by a set of primitive file operations (create, delete, read, write)
- Client interface of a DFS should be transparent, i.e., not distinguish between local and remote files
- Sometimes lower level **intermachine** interface need for cross-machine interaction

Naming and Transparency

- **Naming** – mapping between logical and physical objects
- **Multilevel mapping** – abstraction of a file that hides the details of how and where on the disk the file is actually stored
- A **transparent** DFS hides the location where in the network the file is stored
- For a file being **replicated** in several sites, the mapping returns a set of the locations of this file’s replicas; both the existence of multiple copies and their location are hidden

Naming Structures

- **Location transparency** – file name does not reveal the file's physical storage location
- **Location independence** – file name does not need to be changed when the file's physical storage location changes

Naming Schemes

- Files named by combination of their host name and local name; guarantees a unique system-wide name
- Attach remote directories to local directories, giving the appearance of a coherent directory tree; only previously mounted remote directories can be accessed transparently
- Total integration of the component file systems
  - A single global name structure spans all the files in the system
  - If a server is unavailable, some arbitrary set of directories on different machines also becomes unavailable
- In practice most DFSs use static, location-transparent mapping for user-level names
  - Some support file migration
  - Hadoop supports file migration but without following POSIX standards

Remote File Access

- **Remote-service mechanism** is one transfer approach
- Reduce network traffic by retaining recently accessed disk blocks in a cache, so that repeated accesses to the same information can be handled locally
  - If needed data not already cached, a copy of data is brought from the server to the user
  - Accesses are performed on the cached copy
  - Files identified with one master copy residing at the server machine, but copies of (parts of) the file are scattered in different caches
  - **Cache-consistency problem** – keeping the cached copies consistent with the master file
    - Could be called **network virtual memory**

Cache Location – Disk vs. Main Memory

- Advantages of disk caches
  - More reliable
  - Cached data kept on disk are still there during recovery and don't need to be fetched again
- Advantages of main-memory caches:
  - Permit workstations to be diskless
  - Data can be accessed more quickly
  - Performance speedup in bigger memories
  - Server caches (used to speed up disk I/O) are in main memory regardless of where user caches are located; using main-memory caches on the user machine permits a single caching mechanism for servers and users

Cache Update Policy

- **Write-through** – write data through to disk as soon as they are placed on any cache
  - Reliable, but poor performance
- **Delayed-write** (**write-back**) – modifications written to the cache and then written through to the server later
  - Write accesses complete quickly; some data may be overwritten before they are written back, and so need never be written at all
  - Poor reliability; unwritten data will be lost whenever a user machine crashes
  - Variation – scan cache at regular intervals and flush blocks that have been modified since the last scan
  - Variation – **write-on-close**, writes data back to the server when the file is closed - best for files that are open for long periods and frequently modified

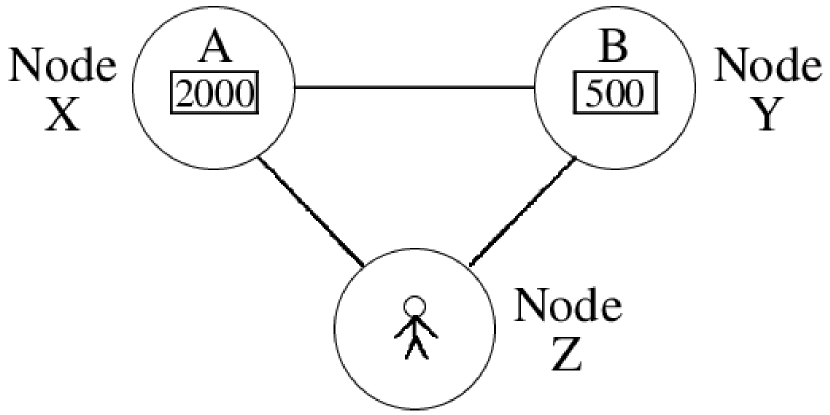
Consistency

- Is locally cached copy of the data consistent with the master copy?
- **Client-initiated approach**
  - Client initiates a validity check
  - Server checks whether the local data are consistent with the master copy
- **Server-initiated approach**
  - Server records, for each client, the (parts of) files it caches
  - When server detects a potential inconsistency, it must react

Summary
Issues in Distributed OS

- Individual computers in a distributed system have their own clocks and memories
  - Clocks in different computers are not synchronized, hence they may show different times
  - It is difficult to obtain a consistent view of data and activities
    - E.g., balances in two accounts, recorded while transferring funds between them, may not be mutually consistent (see next slide)
- A distributed OS has to use special techniques to ensure consistency and reliability
  - Important during normal operation and recovery from a failure

Consistency Issues in Distributed OS



- An observer in node Z records balances in accounts A and B while funds are being transferred
- Z may record balance of X before $100 is transferred from it to Y, and balance of Y after the transfer—$100 is generated!
- Z may record balance of X after $100 is transferred from it to Y, and balance of Y before the transfer—$100 is lost!

Issues in Distributed OS

- Theoretical issues
  - How to know the order in which events have occurred?
    - Important for FCFS allocation, etc.
  - How to know the state of a system
    - Important for avoiding inconsistencies and performing ‘load balancing’ across nodes to obtain good performance
- Distributed control algorithms
  - Parts of the algorithm run in different nodes
    - Obviates the need to collect ‘global’ state in one node
    - Special techniques are used to ensure consistency of actions
  - Used for resource allocation, scheduling, deadlock handling, etc.

Issues in Distributed OS

- Recovery
  - User computations should not suffer due to node or link failures
    - OS restores computations in a failed node to a previous state
    - Special techniques are used to ensure consistency across nodes
- Distributed file systems
  - Files can be accessed from any node
    - Special techniques are needed to ensure good performance when a non-local file is accessed
    - Special techniques are also needed to ensure reliability when nodes or network links fail

Issues in Distributed OS

- Distributed system security
  - Security threats arise from intruders, and viruses and worms
  - OS must provide measures to
    - Prevent intruders from corrupting or fabricating network messages
    - Enable processes to verify identities of other processes
    - Ensure data security
      - Enable users to know who created a document or data
      - Enable users to know whether documents or data have been tampered with

Summary
of Theoretical Issues in Distributed Systems

- Operating systems use notions of time and state
  - Local state: State of an entity
  - Global state: States of all entities at the same time instant
- *Precedence* of events may be deduced using the *causal relationship,* i.e., cause-and-effect relationship, and *transitivity* property of precedence
- Some events may be *concurrent*
- It is laborious to deduce the precedence of events by using transitivity, hence *timestamps* are used instead
  - Use local clocks in processes

Summary
of Theoretical Issues in Distributed Systems

- Using local clocks in processes
  - Logical clocks
  - Vector clocks
- Include process ids in timestamps for total ordering
- It is not possible to record the global state of a system
- Chandy-Lamport algorithm obtains *consistent recording of process states* using special messages called *markers*

Summary of Distributed Control Algorithms

- Actions of a *distributed control algorithms* are performed in many nodes of the system
  - Two aspects of correctness are *liveness* and *safety*
- Distributed system models: *physical* and *logical*
- Examples of distributed control algorithms:
  - Distributed mutual exclusion: e.g., *token* based
  - Distributed deadlock detection: *diffusion computation*
  - Distributed scheduling (to balance load)
  - Distributed termination: e.g., credit-based
  - Election (highest priority process wins)

Summary
of Recovery and Fault Tolerance

- *Recovery* and *fault tolerance* are two approaches to reliability of a computer system
  - Generically called *recovery*
- A third recovery approach is *resiliency*
- A *fault* causes an *error* in the state of the system, which leads to a *failure*
  - A *fail-stop* fault brings the system to a halt
  - An *amnesia* fault makes it lose a part of its state
  - A *Byzantine* fault makes it behave in an unpredictable manner

Summary
of Recovery and Fault Tolerance

- Recovery from non-Byzantine faults can be performed by using two approaches:
  - *Backward recovery* and *forward recovery*
- Fault tolerance implemented by maintaining *logs*
  - E.g., *undo* or *do logs*
  - Logs used to implement *atomic transactions*
    - *Two-phase commit protocol* (2PC protocol) is used
- *Nested transactions* are a *resiliency* technique
  - Used when transaction involves data in many nodes

Summary of Distributed File Systems

- *Transparency* concerns association between path name of a file and location of the file
- *File sharing semantics* may differ between DFSs*:*
  - *Linux semantics*
  - *Session semantics*
  - *Transaction semantics* (*atomic transactions*)
- *Stateless server* design provides high availability
  - Notion of a *hint* used to improve performance
- DFS uses *file caching* to improve performance
  - *Cache coherence* techniques are needed

Message Passing Interface (MPI)

- MPI is a language-independent communications protocol used to program distributed computers.
- MPI is a message-passing application programmer interface, together with protocol and semantic specifications for how its features must behave in any implementation.
- MPI's goals are high performance, scalability, and portability.
- Both point-to-point and collective communication are supported.
- MPI remains the dominant model used in high-performance and distributed computing today.

Message Passing Interface (MPI)

- Most MPI implementations consist of a specific set of routines (i.e., an API) directly callable from C, C++, Fortran (and any language able to interface with such libraries, including C#, Java or Python)

- The advantages of MPI is
  - portability - because MPI has been implemented for almost every distributed memory architecture)
  - speed - because each implementation is in principle optimised for the hardware on which it runs

Message Passing Interface (MPI)

- Threaded shared memory programming models, such as Pthreads and OpenMP, can be considered as complementary programming approaches, and can occasionally be seen together in applications, e.g. in servers with multiple large shared-memory nodes.

MPI Python

- **MPI Python** implementations include: pyMPI, mpi4py,pypar, MYMPI, and the MPI submodule in ScientificPython.
- **pyMPI** is notable because it is a variant python interpreter, while **pypar**, **MYMPI**, and **ScientificPython's** module are import modules. They make it the coder's job to decide where the call to MPI_Init belongs.
- Recently the well known Boost C++ Libraries acquired **Boost:MPI** which included the MPI Python Bindings. This is of particular help for mixing C++ and Python.

MPI – Matlab and R

- **Matlab**: There are a few academic implementations of MPI using Matlab. Matlab has their own parallel extension library implemented using MPI and PVM.

- **R** implementations of MPI include Rmpi and pbdMPI, where Rmpi focuses on manager-workers parallelism while pbdMPI focuses on SPMD parallelism. Both implementations fully support Open MPI.

Message Passing Interface (MPI)

- MPI function parameters: if the data type is a standard one (int, char, double, etc), you can use predefined MPI datatypes (MPI_INT, MPI_CHAR, MPI_DOUBLE, etc)

Here is an example in C that passes an array of int’s and all the processors want to send their arrays to the root with MPI_Gather:

```
int array[100];
int root, total_p, *receive_array;

MPI_Comm_size(comm, &total_p);
receive_array=malloc(total_p*100*sizeof(*receive_array));
MPI_Gather(array, 100, MPI_INT, receive_array, 100, MPI_INT, root, comm);
```

MPI Example Program

- Here is a "Hello World" program in MPI written in C.
- In this example, we send a "hello" message to each processor, manipulate it trivially, return the results to the main process, and print the messages.

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

#define BUFSIZE 128
#define TAG 0

int main(int argc, char *argv[])
{
  char idstr[32];
  char buff[BUFSIZE];
  int numprocs;
  int myid;
  int i;
  MPI_Status stat;
  /* MPI programs start with MPI_Init; all 'N' processes exist thereafter */
  MPI_Init(&argc,&argv);
  /* find out how big the SPMD world is */
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  /* and this processes' rank is */
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);

  /* At this point, all programs are running equivalently, the rank
     distinguishes the roles of the programs in the SPMD model, with
     rank 0 often used specially... */
  if(myid == 0)
  {
    printf("%d: We have %d processors\n", myid, numprocs);
    for(i=1;i<numprocs;i++)
    {
      sprintf(buff, "Hello %d! ", i);
      MPI_Send(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD);
    }
    for(i=1;i<numprocs;i++)
    {
      MPI_Recv(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD, &stat);
      printf("%d: %s\n", myid, buff);
    }
  }
  else
  {
    /* receive from rank 0: */
    MPI_Recv(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD, &stat);
    sprintf(idstr, "Processor %d ", myid);
    strncat(buff, idstr, BUFSIZE-1);
    strncat(buff, "reporting for duty", BUFSIZE-1);
    /* send to rank 0: */
    MPI_Send(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD);
  }

  /* MPI programs end with MPI Finalize; this is a weak synchronization point */
  MPI_Finalize();
  return 0;
}
```