Lecture 7

Introduction to Distributed Programming

System V IPC:

Message Queues, Shared Memory, Semaphores

Introduction to Distributed Programming

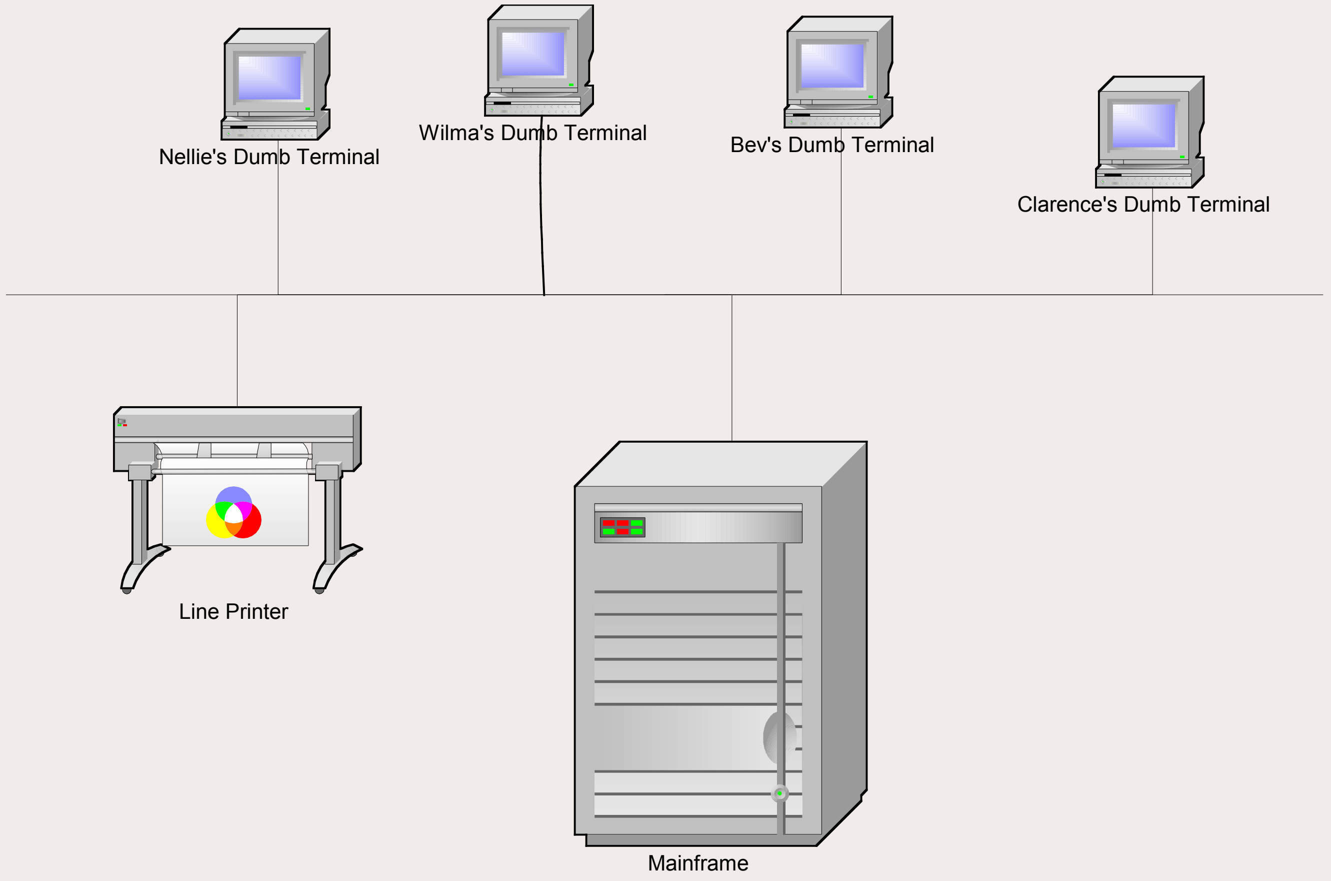
Definitions

- “Distributed programming is the spreading of a computational task across several programs, processes or processors.” – Chris Brown, *Unix Distributed Programming*
- “A distributed system is one in which the failure of a computer you didn’t even know existed can render your own computer unusable.” – Leslie Lamport
- “A parallel computer is a set of processors that are able to work cooperatively to solve a computational problem.” – Ian Foster, *Designing and Building Parallel Programs*
- “A distributed system is a system in which multiple processes coordinate in solving a problem and, in the process of solving that problem, create other problems.” – Mark Shacklette

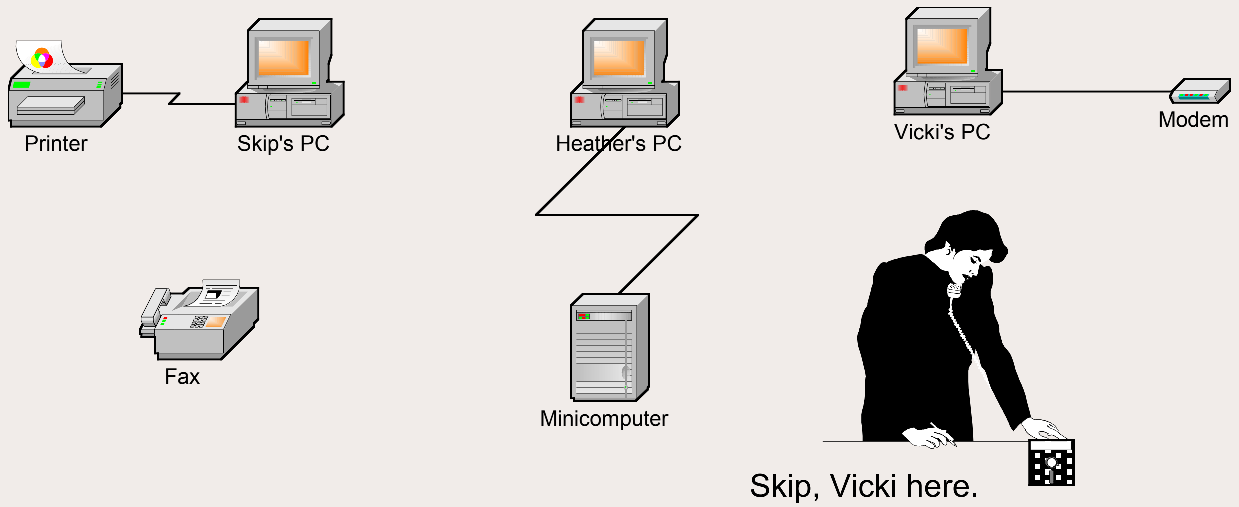
Benefits of Distributed Programming

- Divide and Conquer
  - Concurrency
  - Parallelism
- Component Reuse via pipelines (Modularity)
- Location Independence
- Scalability
- Resource Sharing

Mainframe Topology

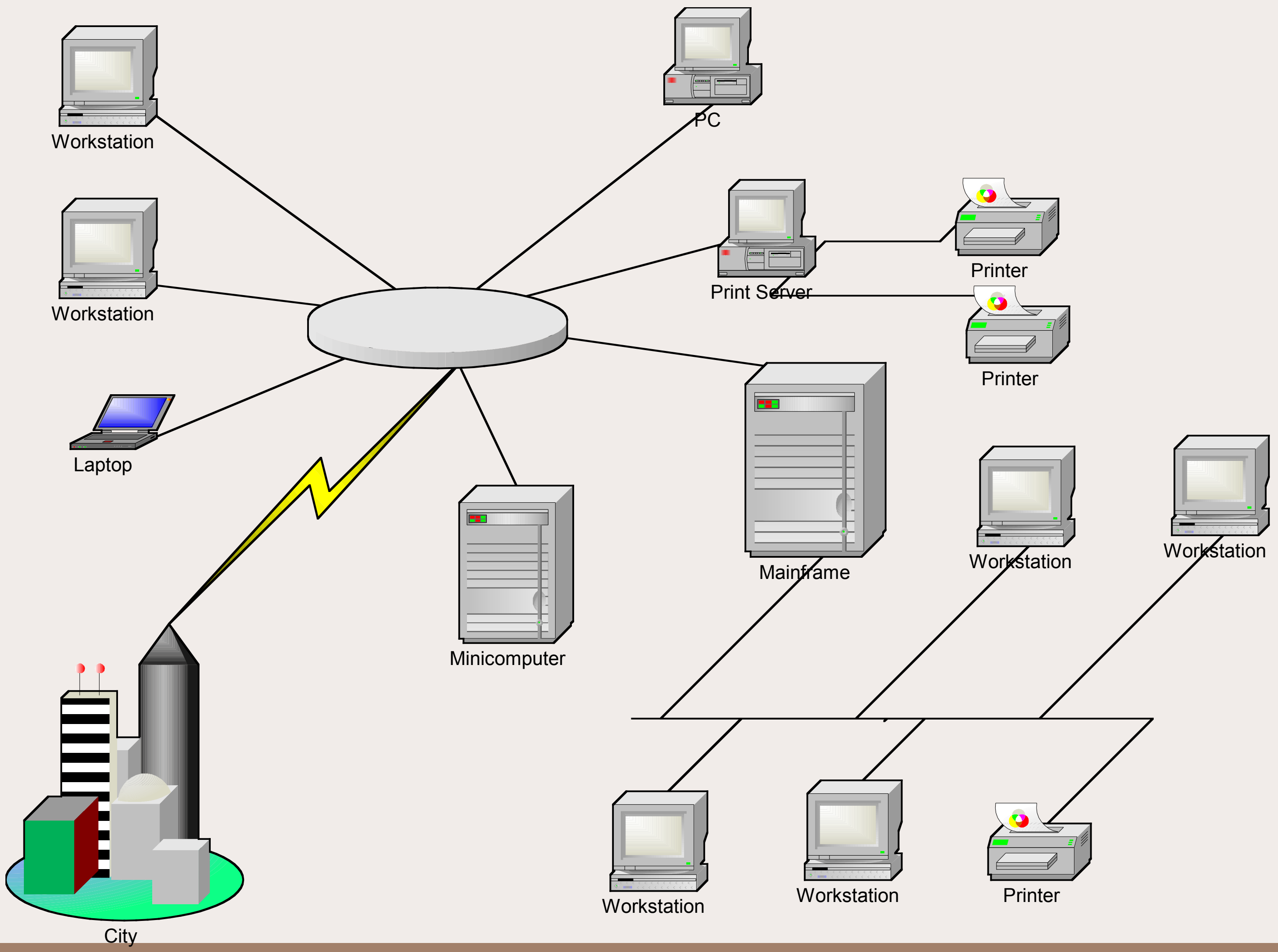


Sneaker Net



Skip, Vicki here.
Can I come down
and use your
printer?

Modern Network



Problem Space

- Problem 1
  - You have 1 hour to peel 1000 potatoes
  - You have 10 people available
- Problem 2
  - You have 1 hour to do the dishes after a dinner for 1000 guests
  - You have 10 people available
- Problem 3
  - You have 1 hour to lay the brick around a 5’ square dog house
  - You have 10 people available

Facilitating Division of Labor: Work and Communication

- Single Machine Inter-process Communication
  - (Signals)
  - Pipes (named and unnamed)
  - System V and POSIX IPC
- Multiple Machine Inter-process Communication
  - Sockets
  - Remote Procedure Calls (Sun ONC, OSF DCE, Xerox Courier (4.3BSD))
  - Distributed Shared Memory (Berkeley mmap)
- Single Machine Division of Labor:
  - Processes
  - Threads

Methods of Solution Distribution: Input Distribution (Division of Labor)

- Workload Decomposition
  - Potato Peelers aboard the USS Enterprise
    - loosely coupled (little coordination)
  - Roofers or Bricklayers
    - tightly coupled (high coordination)
- Software: large database query of all records with a given characteristic
  - Strategy: Divide and Conquer
  - Key: *Exact same code* is operating on different *sets* of input data
- Software: large matrix multiplication
  - Strategy: Divide and Conquer
  - Key: *Exact same code* is operating on different *parts* of the matrices

Methods of Solution Distribution: Process Decomposition (Inter-process Communication)

- Divide not the *work*, but the *process* of conducting the work
  - Factory Production Line:
    - Identical widgets are coming along the converyor belt, but several things have to be done to each widget
  - Dish Washing Example
    - collector, washer, dryer, cabinet deployer
    - multiple washers and dryers can be employed (using Input Distribution)
- Software: A Trade Clearing System
  - Each trade must be entered, validated, reported, notified
  - Each task can run within a different process on a different processor
  - Strategy: divide the work to be done for each trade into separate processes, thus increasing overall system *throughput*

Problems in Distributed Solutions

- Data access must be synchronized among multiple processes
- Multiple processes must be able to communicate among themselves in order to coordinate activities
- Multiple coordinating processes must be able to *locate* one another

Interprocess Communication and Synchronization using System V IPC

Message Queues

Shared Memory

Semaphores

System V IPC

- System V IPC was first introduced in SVR2, but is available now in most versions of unix
- Message Queues represent linked lists of messages, which can be written to and read from
- Shared memory allows two or more processes to share a region of memory, so that they may each read from and write to that memory region
- Semaphores synchronize access to shared resources by providing synchronized access among multiple processes trying to access those critical resources.

Message Queues

- A Message Queue is a linked list of message structures stored inside the kernel’s memory space and accessible by multiple processes
- Synchronization is provided automatically by the kernel
- New messages are added at the end of the queue
- Each message structure has a long *message type*
- Messages may be obtained from the queue either in a FIFO manner (default) or by requesting a specific *type* of message (based on *message type*)

Message Structs

- Each message structure must start with a long message type:

```
struct mymsg {
    long msg_type;
    char mytext[512]; /* rest of message */
    int somethingelse;
    float dollarval;
};
```

Message Queue Limits

- Each message queue is limited in terms of both the maximum number of messages it can contain and the maximum number of bytes it may contain
- New messages cannot be added if *either* limit is hit (new writes will normally block)
- On linux, these limits are defined as (in /usr/include/linux/msg.h):
  - MSGMAX 8192 /*total number of messages */
  - MSBMNB 16384 /* max bytes in a queue */

Obtaining a Message Queue

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget(key_t key, int msgflg);
```

- key is either a number or the constant IPC_PRIVATE
- a msgid is returned
- key_t ftok(const char * path, int id) will return a key value for IPC usage
- The key parameter is either a non-zero identifier for the queue to be created or the value IPC_PRIVATE, which guarantees that a new queue is created.
- The msgflg parameter is the read-write permissions for the queue OR'd with one of two flags:
  - IPC_CREAT will create a new queue or return an existing one
  - IPC_EXCL added will force the creation of a new queue, or return an error

Writing to a Message Queue

int msgsnd(int msqid, const void * msg_ptr, size_t msg_size, int msgflags);

- msgqid is the id returned from the msgget call
- msg_ptr is a pointer to the message structure
- msg_size is the size of that structure
- msgflags defines what happens when the queue is full, and can be set to the following:
  - IPC_NOWAIT (non-blocking, return –1 immediately if queue is full)

Reading from a Message Queue

```
int msgrcv(int msqid, const void * msg_ptr, size_t msg_size, long msgtype, int msgflags);
```

- msgqid is the id returned from the msgget call
- msg_ptr is a pointer to the message structure
- msg_size is the size of that structure
- msgtype is set to:

= 0	first message available in FIFO stack
> 0	first message on queue whose type equals type
< 0	first message on queue whose type is the lowest value less than or equal to the absolute value of msgtype

- msgflags defines what happens when no message of the appropriate type is waiting, and can be set to the following:
  - IPC_NOWAIT (non-blocking, return –1 immediately if queue is empty)
- *example: ~mark/pub/51081/message.queues/potato.*.c*

Message Queue Control

```
struct msqid_ds {
  ...                          /* pointers to first and last messages on queue */
  __time_t msg_stime;                  /* time of last msgsnd command */
  __time_t msg_rtime;                  /* time of last msgrcv command */
  ...
  unsigned short int __msg_cbytes;     /* current number of bytes on queue */
  msgqnum_t msg_qnum;                  /* number of messages currently on queue */
  msglen_t msg_qbytes;                 /* max number of bytes allowed on queue */
  ...                                  /* pids of last msgsnd() and msgrcv() */
};
```

- int msgctl(int msqid, int cmd, struct msqid_ds * buf);
- cmd can be one of:
  - IPC_RMID — destroy the queue specified by msqid
  - IPC_SET — set the uid, gid, mode, and qbytes for the queue, if adequate permission is available
  - IPC_STAT — get the current msqid_ds struct for the queue
- *example: query.c*

Shared Memory

- Normally, the Unix kernel prohibits one process from accessing (reading, writing) memory belonging to another process
- Sometimes, however, this restriction is inconvenient
- At such times, System V IPC Shared Memory can be created to specifically allow one process to read and/or write to memory created by another process

Advantages of Shared Memory

- Random Access
  - you can update a small piece in the middle of a data structure, rather than the entire structure
- Efficiency
  - unlike message queues and pipes, which copy data from the process *into* memory within the kernel, shared memory is directly accessed
  - Shared memory resides in the user process memory, and is then shared among other processes

Disadvantages of Shared Memory

- No automatic synchronization as in pipes or message queues (you have to provide any synchronization). Synchronize with *semaphores* or signals.
- You must remember that pointers are only valid within a given process. Thus, pointer offsets cannot be assumed to be valid across inter-process boundaries. This complicates the sharing of linked lists or binary trees.

Creating Shared Memory

int shmget(key_t key, size_t size, int shmflg);

- key is either a number or the constant IPC_PRIVATE (man ftok)
- a shmid is returned
- key_t ftok(const char * path, int id) will return a key value for IPC usage
- size is the size of the shared memory data
- shmflg is a rights mask (0666) OR'd with one of the following:

– IPC_CREAT	will create or attach
– IPC_EXCL	creates new or it will error if it exists

Attaching to Shared Memory

- After obtaining a shmid from shmget(), you need to *attach* or map the shared memory segment to your data reference:

```
void * shmat(int shmid, void * shmaddr, int shmflg)
```

- shmid is the id returned from shmget()
- shmaddr is the shared memory segment address. Set this to NULL and let the system handle it.
- shmflg is one of the following (usually 0):

– SHM_RDONLY	sets the segment readonly
– SHM_RND	sets page boundary access
– SHM_SHARE_MMU	set first available aligned address

Shared Memory Control

```
struct shmid_ds {
 int shm_segsz;                          /* size of segment in bytes */
  __time_t shm_atime;                    /* time of last shmat command */
  __time_t shm_dtime;                    /* time of last shmdt command */
  ...
  unsigned short int __shm_npages;       /* size of segment in pages */
  msgqnum_t shm_nattach;                 /* number of current attaches */
    ...                                  /* pids of creator and last shmop */
};
```

- int shmctl(int shmid, int cmd, struct shmid_ds * buf);
- cmd can be one of:
  - IPC_RMID destroy the memory specified by shmid
  - IPC_SET set the uid, gid, and mode of the shared mem
  - IPC_STAT get the current shmid_ds struct for the queue
- example: ~mark/pub/51081/shared.memory/linux/*

Matrix Multiplication

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j}$$

- Multiply two n x n matrices, *a* and *b*
- One each iteration, a row of A multiplies a column of b, such that:

$$\mathrm{c}_{\mathrm{p,k}} = \mathrm{c}_{\mathrm{p,k}} + \mathrm{a}_{\mathrm{p,p\text{-}1}} \mathrm{b}_{\mathrm{p\text{-}1,k}}$$

Semaphores

- Shared memory is not access controlled by the kernel
- This means critical sections must be protected from potential conflicts with multiple writers
- A critical section is a section of code that would prove problematic if two or more separate processes wrote to it simultaneously
- Semaphores were invented to provide such locking protection on shared memory segments

System V Semaphores

- You can create an array of semaphores that can be controlled as a group
- Semaphores (Dijkstra, 1965) may be binary (0/1), or counting

  1 == unlocked (available resource)

  0 == locked
- Thus:
  - To unlock a semaphore, you +INCREMENT it
  - To lock a semaphore, you -DECREMENT it
- Spinlocks are busy waiting semaphores that constantly poll to see if they may proceed (Dekker's Algorithm)

How Semaphores Work

- A critical section is defined
- A semaphore is created to protect it
- The first process into the critical section locks the critical section
- All subsequent processes *wait* on the semaphore, and they are added to the semaphore's "waiting list"
- When the first process is out of the critical section, it *signals* the semaphore that it is done
- The semaphore then *wakes up* one of its waiting processes to proceed into the critical section
- All waiting and signaling are done *atomically*

How Semaphores “Don’t” Work: Deadlocks and Starvation

- When two processes (p,q) are both waiting on a semaphore, and p cannot proceed until q signals, and q cannot continue until p signals. They are both asleep, waiting. Neither can signal the other, wake the other up. This is called a *deadlock*.
  - P1 locks a which succeeds, then waits on b
  - P2 locks b which succeeds, then waits on a
- Indefinite blocking, or *starvation*, occurs when one process is constantly in a wait state, and is never signaled. This often occurs in LIFO situations.
- *example: ~mark/pub/51081/semaphores/linux/ shmem.matrix.multiplier2.c*