UNIT-I

MODES OF COMMUNICATION

SYSTEM PROCESS

- **Distributed operating system**. A **distributed operating system** is a software over a collection of independent, networked, communicating, and physically separate computational nodes. ... The first is a ubiquitous minimal kernel, or microkernel, that directly controls that node's hardware.

INTERRUPT HANDLING

Except for the last chapter, everything we did in the kernel so far we've done as a response to a process asking for it, either by dealing with a special file, sending an `ioctl()`, or issuing a system call. But the job of the kernel isn't just to respond to process requests. Another job, which is every bit as important, is to speak to the hardware connected to the machine.

There are two types of interaction between the CPU and the rest of the computer's hardware. The first type is when the CPU gives orders to the hardware, the other is when the hardware needs to tell the CPU something. The second, called interrupts, is much harder to implement because it has to be dealt with when convenient for the hardware, not the CPU. Hardware devices typically have a very small amount of RAM, and if you don't read their information when available, it is lost.

Under Linux, hardware interrupts are called IRQ's (*I*nterrupt*R*e *q*uests)[16]. There are two types of IRQ's, short and long. A short IRQ is one which is expected to take a *very* short period of time, during which the rest of the machine will be blocked and no other interrupts will be handled. A long IRQ is one which can take longer, and during which other interrupts may occur (but not interrupts from the same device). If at all possible, it's better to declare an interrupt handler to be long.

When the CPU receives an interrupt, it stops whatever it's doing (unless it's processing a more important interrupt, in which case it will deal with this one only when the more important one is done), saves certain parameters on the stack and calls the interrupt handler. This means that certain things are not allowed in the interrupt handler itself, because the system is in an unknown state. The solution to this problem is for the interrupt handler to do what needs to be done immediately, usually read something from the hardware or send something to the hardware, and then schedule the handling of the new information at a later time (this is called the "bottom half") and return. The kernel is then guaranteed to call the bottom half as soon as possible –– and when it does, everything allowed in kernel modules will be allowed.

INTERRUPT HANDLING

I had a problem with writing the sample code for this chapter. On one hand, for an example to be useful it has to run on everybody's computer with meaningful results. On the other hand, the kernel already includes device drivers for all of the common devices, and those device drivers won't coexist with what I'm going to write. The solution I've found was to write something for the keyboard interrupt, and disable the regular keyboard interrupt handler first. Since it is defined as a static symbol in the kernel source files (specifically, `drivers/char/keyboard.c`), there is no way to restore it. Before **`insmod`**'ing this code, do on another terminal **`sleep 120; reboot`** if you value your file system.

This code binds itself to IRQ 1, which is the IRQ of the keyboard controlled under Intel architectures. Then, when it receives a keyboard interrupt, it reads the keyboard's status (that's the purpose of the **`inb(0x64)`**) and the scan code, which is the value returned by the keyboard. Then, as soon as the kernel thinks it's feasible, it runs `got_char` which gives the code of the key used (the first seven bits of the scan code) and whether it has been pressed (if the 8th bit is zero) or released (if it's one).

INTERRUPT HANDLING

```
/*
 *  intrpt.c - An interrupt handler.
 *
 *  Copyright (C) 2001 by Peter Jay Salzman
 */

/*
 * The necessary header files
 */

/*
 * Standard in kernel modules
 */
#include <linux/kernel.h>       /* We're doing kernel work */
#include <linux/module.h>       /* Specifically, a module */
#include <linux/sched.h>
#include <linux/workqueue.h>
#include <linux/interrupt.h>    /* We want an interrupt */
#include <asm/io.h>

#define MY_WORK_QUEUE_NAME "WQsched.c"

static struct workqueue_struct *my_workqueue;

/*
 * This will get called by the kernel as soon as it's safe
 * to do everything normally allowed by kernel modules.
 */
static void got_char(void *scancode)
{
        printk(KERN_INFO "Scan Code %x %s.\n",
               (int)*((char *)scancode) & 0x7F,
               *((char *)scancode) & 0x80 ? "Released" : "Pressed");
}

/*
 * This function services keyboard interrupts. It reads the relevant
 * information from the keyboard and then puts the non time critical
 * part into the work queue. This will be run when the kernel considers it safe.
 */
irqreturn_t irq_handler(int irq, void *dev_id, struct pt_regs *regs)
```

INTERRUPT HANDLING

```
{
        /*
         * This variables are static because they need to be
         * accessible (through pointers) to the bottom half routine.
         */
        static int initialised = 0;
        static unsigned char scancode;
        static struct work_struct task;
        unsigned char status;

        /*
         * Read keyboard status
         */
        status = inb(0x64);
        scancode = inb(0x60);

        if (initialised == 0) {
                INIT_WORK(&task, got_char, &scancode);
                initialised = 1;
        } else {
                PREPARE_WORK(&task, got_char, &scancode);
        }

        queue_work(my_workqueue, &task);

        return IRQ_HANDLED;
}

/*
 * Initialize the module - register the IRQ handler
 */
int init_module()
{
        my_workqueue = create_workqueue(MY_WORK_QUEUE_NAME);

        /*
         * Since the keyboard handler won't co-exist with another handler,
         * such as us, we have to disable it (free its IRQ) before we do
         * anything.  Since we don't know where it is, there's no way to
         * reinstate it later - so the computer will have to be rebooted
         * when we're done.
         */
        free_irq(1, NULL);
```

INTERRUPT HANDLING

```
         * Request IRQ 1, the keyboard IRQ, to go to our irq_handler.
         * SA_SHIRQ means we're willing to have othe handlers on this IRQ.
         * SA_INTERRUPT can be used to make the handler into a fast interrupt.
         */
        return request_irq(1,   /* The number of the keyboard IRQ on PCs */
                           irq_handler, /* our handler */
                           SA_SHIRQ, "test_keyboard_irq_handler",
                           (void *)(irq_handler));
}

/*
 * Cleanup
 */
void cleanup_module()
{
        /*
         * This is only here for completeness. It's totally irrelevant, since
```

Chapter 12. Interrupt Handlers

The Linux Kernel Module Programming Guide

```
         * we don't have a way to restore the normal keyboard interrupt so the
         * computer is completely useless and has to be rebooted.
         */
        free_irq(1, NULL);
}

/*
 * some work_queue related functions are just available to GPL licensed Modules
 */
MODULE_LICENSE("GPL");
```

HANDLING SYSTEM CALLS

So far, the only thing we've done was to use well defined kernel mechanisms to register `/proc` files and device handlers. This is fine if you want to do something the kernel programmers thought you'd want, such as write a device driver. But what if you want to do something unusual, to change the behavior of the system in some way? Then, you're mostly on your own.

This is where kernel programming gets dangerous. While writing the example below, I killed the `open()` system call. This meant I couldn't open any files, I couldn't run any programs, and I couldn't **shutdown** the computer. I had to pull the power switch. Luckily, no files died. To ensure you won't lose any files either, please run **sync** right before you do the **insmod** and the **rmmod**.

Forget about `/proc` files, forget about device files. They're just minor details. The *real* process to kernel communication mechanism, the one used by all processes, is system calls. When a process requests a service from the kernel (such as opening a file, forking to a new process, or requesting more memory), this is the mechanism used. If you want to change the behaviour of the kernel in interesting ways, this is the place to do it. By the way, if you want to see which system calls a program uses, run **strace <arguments>**.

In general, a process is not supposed to be able to access the kernel. It can't access kernel memory and it can't call kernel functions. The hardware of the CPU enforces this (that's the reason why it's called `protected mode').

System calls are an exception to this general rule. What happens is that the process fills the registers with the appropriate values and then calls a special instruction which jumps to a previously defined location in the kernel (of course, that location is readable by user processes, it is not writable by them). Under Intel CPUs, this is done by means of interrupt 0x80. The hardware knows that once you jump to this location, you are no longer running in restricted user mode, but as the operating system kernel --- and therefore you're allowed to do whatever you want.

HANDLING SYSTEM CALLS

System calls are an exception to this general rule. What happens is that the process fills the registers with the appropriate values and then calls a special instruction which jumps to a previously defined location in the kernel (of course, that location is readable by user processes, it is not writable by them). Under Intel CPUs, this is done by means of interrupt 0x80. The hardware knows that once you jump to this location, you are no longer running in restricted user mode, but as the operating system kernel --- and therefore you're allowed to do whatever you want.

The location in the kernel a process can jump to is called *system_call*. The procedure at that location checks the system call number, which tells the kernel what service the process requested. Then, it looks at the table of system calls (`sys_call_table`) to see the address of the kernel function to call. Then it calls the function, and after it returns, does a few system checks and then return back to the process (or to a different process, if the process time ran out). If you want to read this code, it's at the source file `arch/$<$architecture$>$/kernel/entry.S`, after the line `ENTRY(system_call)`.

So, if we want to change the way a certain system call works, what we need to do is to write our own function to implement it (usually by adding a bit of our own code, and then calling the original function) and then change the pointer at `sys_call_table` to point to our function. Because we might be removed later and we don't want to leave the system in an unstable state, it's important for `cleanup_module` to restore the table to its original state.

The source code here is an example of such a kernel module. We want to `spy' on a certain user, and to `printk()` a message whenever that user opens a file. Towards this end, we replace the system call to open a file with our own function, called `our_sys_open`. This function checks the uid (user's id) of the current process, and if it's equal to the uid we spy on, it calls `printk()` to display the name of the file to be opened. Then, either way, it calls the original `open()` function with the same parameters, to actually open the file.

HANDLING SYSTEM CALLS

The `init_module` function replaces the appropriate location in `sys_call_table` and keeps the original pointer in a variable. The `cleanup_module` function uses that variable to restore everything back to normal. This approach is dangerous, because of the possibility of two kernel modules changing the same system call. Imagine we have two kernel modules, A and B. A's open system call will be A_open and B's will be B_open. Now, when A is inserted into the kernel, the system call is replaced with A_open, which will call the original sys_open when it's done. Next, B is inserted into the kernel, which replaces the system call with B_open, which will call what it thinks is the original system call, A_open, when it's done.

Now, if B is removed first, everything will be well---it will simply restore the system call to A_open, which calls the original. However, if A is removed and then B is removed, the system will crash. A's removal will restore the system call to the original, sys_open, cutting B out of the loop. Then, when B is removed, it will restore the system call to what *it* thinks is the original, A_open, which is no longer in memory. At first glance, it appears we could solve this particular problem by checking if the system call is equal to our open function and if so not changing it at all (so that B won't change the system call when it's removed), but that will cause an even worse problem. When A is removed, it sees that the system call was changed to B_open so that it is no longer pointing to A_open, so it won't restore it to sys_open before it is removed from memory. Unfortunately, B_open will still try to call A_open which is no longer there, so that even without removing B the system would crash.

Note that all the related problems make syscall stealing unfeasiable for production use. In order to keep people from doing potential harmful things sys_call_table is no longer exported. This means, if you want to do something more than a mere dry run of this example, you will have to patch your current kernel in order to have sys_call_table exported. In the example directory you will find a README and the patch. As you can imagine, such modifications are not to be taken lightly. Do not try this on valueable systems (ie systems that you do not own – or cannot restore easily). You'll need to get the complete sourcecode of this guide as a tarball in order to get the patch and the README. Depending on your kernel version, you might even need to hand apply the patch. Still here? Well, so is this chapter. If Wyle E. Coyote was a kernel hacker, this would be the first thing he'd try. ;)

HANDLING SYSTEM CALLS

```
/*
 *  syscall.c
 *
 *  System call "stealing" sample.
 */

/*
 * Copyright (C) 2001 by Peter Jay Salzman
 */

/*
 * The necessary header files
 */

/*
 * Standard in kernel modules
 */
#include <linux/kernel.h>       /* We're doing kernel work */
#include <linux/module.h>       /* Specifically, a module, */
#include <linux/moduleparam.h>  /* which will have params */
#include <linux/unistd.h>       /* The list of system calls */
```

HANDLING SYSTEM CALLS

```
 * For the current (process) structure, we need
 * this to know who the current user is.
 */
#include <linux/sched.h>
#include <asm/uaccess.h>

/*
 * The system call table (a table of functions). We
 * just define this as external, and the kernel will
 * fill it up for us when we are insmod'ed
 *
 * sys_call_table is no longer exported in 2.6.x kernels.
 * If you really want to try this DANGEROUS module you will
 * have to apply the supplied patch against your current kernel
 * and recompile it.
 */
extern void *sys_call_table[];

/*
 * UID we want to spy on - will be filled from the
 * command line
 */
static int uid;
module_param(uid, int, 0644);

/*
 * A pointer to the original system call. The reason
 * we keep this, rather than call the original function
 * (sys_open), is because somebody else might have
 * replaced the system call before us. Note that this
 * is not 100% safe, because if another module
 * replaced sys_open before us, then when we're inserted
 * we'll call the function in that module - and it
 * might be removed before we are.
 *
 * Another reason for this is that we can't get sys_open.
 * It's a static variable, so it is not exported.
 */
asmlinkage int (*original_call) (const char *, int, int);
```

HANDLING SYSTEM CALLS

```
/*
 * The function we'll replace sys_open (the function
 * called when you call the open system call) with. To
 * find the exact prototype, with the number and type
 * of arguments, we find the original function first
 * (it's at fs/open.c).
 *
 * In theory, this means that we're tied to the
 * current version of the kernel. In practice, the
 * system calls almost never change (it would wreck havoc
 * and require programs to be recompiled, since the system
 * calls are the interface between the kernel and the
 * processes).
 */
asmlinkage int our_sys_open(const char *filename, int flags, int mode)
{
        int i = 0;
        char ch;

        /*
         * Check if this is the user we're spying on
         */
```

HANDLING SYSTEM CALLS

```
        if (uid == current->uid) {
                /*
                 * Report the file, if relevant
                 */
                printk("Opened file by %d: ", uid);
                do {
                        get_user(ch, filename + i);
                        i++;
                        printk("%c", ch);
                } while (ch != 0);
                printk("\n");
        }

        /*
         * Call the original sys_open - otherwise, we lose
         * the ability to open files
         */
        return original_call(filename, flags, mode);
}

/*
 * Initialize the module - replace the system call
 */
int init_module()
{
        /*
         * Warning - too late for it now, but maybe for
         * next time...
         */
        printk(KERN_ALERT "I'm dangerous. I hope you did a ");
        printk(KERN_ALERT "sync before you insmod'ed me.\n");
        printk(KERN_ALERT "My counterpart, cleanup_module(), is even");
        printk(KERN_ALERT "more dangerous. If\n");
        printk(KERN_ALERT "you value your file system, it will ");
        printk(KERN_ALERT "be \"sync; rmmod\" \n");
        printk(KERN_ALERT "when you remove this module.\n");
```

HANDLING SYSTEM CALLS

```
        /*
         * Keep a pointer to the original function in
         * original_call, and then replace the system call
         * in the system call table with our_sys_open
         */
        original_call = sys_call_table[__NR_open];
        sys_call_table[__NR_open] = our_sys_open;

        /*
         * To get the address of the function for system
         * call foo, go to sys_call_table[__NR_foo].
         */

        printk(KERN_INFO "Spying on UID:%d\n", uid);

        return 0;
}

/*
 * Cleanup - unregister the appropriate file from /proc
 */
void cleanup_module()
{
        /*
         * Return the system call back to normal
```

HANDLING SYSTEM CALLS


```
        */
        if (sys_call_table[__NR_open] != our_sys_open) {
                printk(KERN_ALERT "Somebody else also played with the ");
                printk(KERN_ALERT "open system call\n");
                printk(KERN_ALERT "The system may be left in ");
                printk(KERN_ALERT "an unstable state.\n");
        }

        sys_call_table[__NR_open] = original_call;
}
```

PROTECTION OF RESOURCES

What do you do when somebody asks you for something you can't do right away? If you're a human being and you're bothered by a human being, the only thing you can say is: "Not right now, I'm busy. *Go away!*". But if you're a kernel module and you're bothered by a process, you have another possibility. You can put the process to sleep until you can service it. After all, processes are being put to sleep by the kernel and woken up all the time (that's the way multiple processes appear to run on the same time on a single CPU).

This kernel module is an example of this. The file (called `/proc/sleep`) can only be opened by a single process at a time. If the file is already open, the kernel module calls `wait_event_interruptible`[12]. This function changes the status of the task (a task is the kernel data structure which holds information about a process and the system call it's in, if any) to *`TASK_INTERRUPTIBLE`*, which means that the task will not run until it is woken up somehow, and adds it to WaitQ, the queue of tasks waiting to access the file. Then, the function calls the scheduler to context switch to a different process, one which has some use for the CPU.

When a process is done with the file, it closes it, and `module_close` is called. That function wakes up all the processes in the queue (there's no mechanism to only wake up one of them). It then returns and the process which just closed the file can continue to run. In time, the scheduler decides that that process has had enough and gives control of the CPU to another process. Eventually, one of the processes which was in the queue will be given control of the CPU by the scheduler. It starts at the point right after the call to `module_interruptible_sleep_on`[13]. It can then proceed to set a global variable to tell all the other processes that the file is still open and go on with its life. When the other processes get a piece of the CPU, they'll see that global variable and go back to sleep.

So we'll use **tail –f** to keep the file open in the background, while trying to access it with another process (again in the background, so that we need not switch to a different vt). As soon as the first background process is killed with **kill %1** , the second is woken up, is able to access the file and finally terminates.

To make our life more interesting, `module_close` doesn't have a monopoly on waking up the processes which wait to access the file. A signal, such as **Ctrl+c** (*`SIGINT`*) can also wake up a process. [14] In that case, we want to return with *`-EINTR`* immediately. This is important so users can, for example, kill the process before it receives the file.

There is one more point to remember. Some times processes don't want to sleep, they want either to get what they want immediately, or to be told it cannot be done. Such processes use the *O_NONBLOCK* flag when opening the file. The kernel is supposed to respond by returning with the error code *-EAGAIN* from operations which would otherwise block, such as opening the file in this example. The program **cat_noblock** available in the source directory for this chapter, can be used to open a file with *O_NONBLOCK*.

```
hostname:~/lkmpg-examples/09-BlockingProcesses# insmod sleep.ko
hostname:~/lkmpg-examples/09-BlockingProcesses# cat_noblock /proc/sleep
Last input:
hostname:~/lkmpg-examples/09-BlockingProcesses# tail -f /proc/sleep &
Last input:
Last input:
Last input:
Last input:
Last input:
Last input:
```



```
Last input:
tail: /proc/sleep: file truncated
[1] 6540
hostname:~/lkmpg-examples/09-BlockingProcesses# cat_noblock /proc/sleep
Open would block
hostname:~/lkmpg-examples/09-BlockingProcesses# kill %1
[1]+  Terminated              tail -f /proc/sleep
hostname:~/lkmpg-examples/09-BlockingProcesses# cat_noblock /proc/sleep
Last input:
hostname:~/lkmpg-examples/09-BlockingProcesses#
```

```
/*
 *  sleep.c - create a /proc file, and if several processes try to open it at
 *  the same time, put all but one to sleep
 */

#include <linux/kernel.h>       /* We're doing kernel work */
#include <linux/module.h>       /* Specifically, a module */
#include <linux/proc_fs.h>      /* Necessary because we use proc fs */
#include <linux/sched.h>        /* For putting processes to sleep and
                                   waking them up */
#include <asm/uaccess.h>        /* for get_user and put_user */

/*
 * The module's file functions
 */

/*
 * Here we keep the last message received, to prove that we can process our
 * input
 */
#define MESSAGE_LENGTH 80
static char Message[MESSAGE_LENGTH];

static struct proc_dir_entry *Our_Proc_File;
#define PROC_ENTRY_FILENAME "sleep"

/*
 * Since we use the file operations struct, we can't use the special proc
 * output provisions - we have to use a standard read function, which is this
 * function
 */
static ssize_t module_output(struct file *file, /* see include/linux/fs.h   */
                             char *buf, /* The buffer to put data to
                                           (in the user segment)    */
                             size_t len,        /* The length of the buffer */
                             loff_t * offset)
```

```
                                loff_t * offset)
{
        static int finished = 0;
        int i;
        char message[MESSAGE_LENGTH + 30];

        /*
         * Return 0 to signify end of file - that we have nothing
         * more to say at this point.
         */
        if (finished) {
                finished = 0;
                return 0;
```



```
        }

        /*
         * If you don't understand this by now, you're hopeless as a kernel
         * programmer.
         */
        sprintf(message, "Last input:%s\n", Message);
        for (i = 0; i < len && message[i]; i++)
                put_user(message[i], buf + i);

        finished = 1;
        return i;                /* Return the number of bytes "read" */
}

/*
 * This function receives input from the user when the user writes to the /proc
 * file.
 */
static ssize_t module_input(struct file *file,   /* The file itself */
```

PROTECTION OF RESOURCES

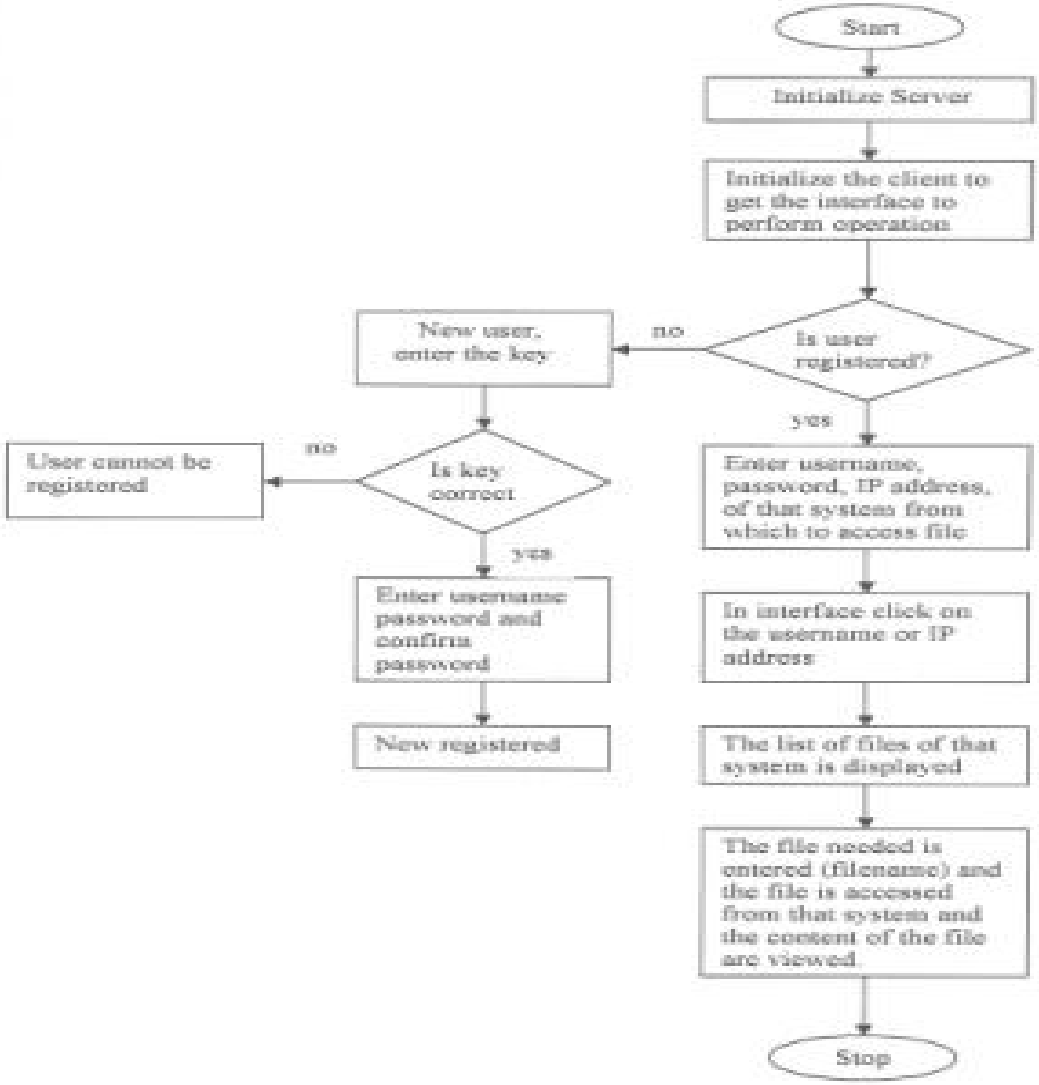
Example 9–2. cat_noblock.c

```
/* cat_noblock.c - open a file and display its contents, but exit rather than
 * wait for input */


/* Copyright (C) 1998 by Ori Pomerantz */



#include <stdio.h>     /* standard I/O */
#include <fcntl.h>     /* for open */
#include <unistd.h>    /* for read */
#include <stdlib.h>    /* for exit */
#include <errno.h>     /* for errno */

#define MAX_BYTES 1024*4


main(int argc, char *argv[])
{
  int     fd;  /* The file descriptor for the file to read */
  size_t bytes; /* The number of bytes read */
  char   buffer[MAX_BYTES]; /* The buffer for the bytes */


  /* Usage */
  if (argc != 2) {
    printf("Usage: %s <filename>\n", argv[0]);
    puts("Reads the content of a file, but doesn't wait for input");
    exit(-1);
  }
```

PROTECTION OF RESOURCES

```
  /* Read the file and output its contents */
  do {
    int i;

    /* Read characters from the file */
    bytes = read(fd, buffer, MAX_BYTES);

    /* If there's an error, report it and die */
    if (bytes == -1) {
      if (errno = EAGAIN)
        puts("Normally I'd block, but you told me not to");
      else
        puts("Another read error");
      exit(-1);
    }

    /* Print the characters */
    if (bytes > 0) {
      for(i=0; i<bytes; i++)
        putchar(buffer[i]);
    }

    /* While there are no errors and the file isn't over */
  } while (bytes > 0);
}
```

RESOURCES MANAGEMENT



MICRO- KERNEL

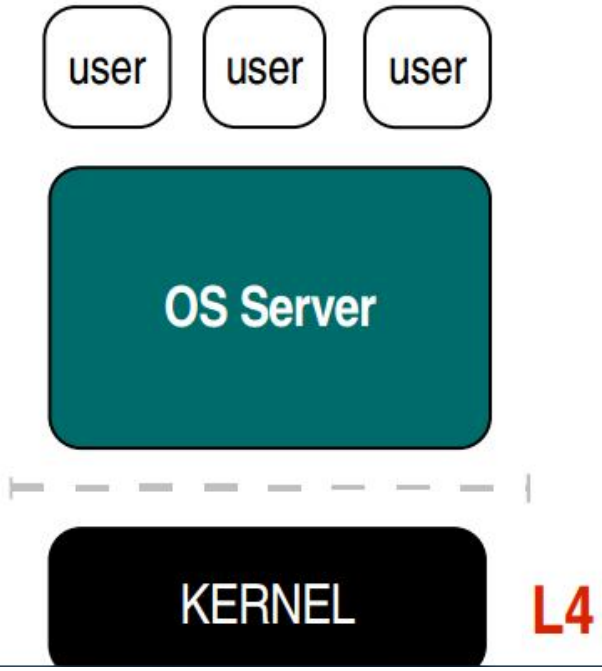
WHAT IS A MICROKERNEL-BASED DESIGN?

- **Microkernel implements only privileged OS functionality**
  - E.g., interrupt handling, scheduling, programming the hardware
- **Remove everything that can be done outside the kernel**
  - E.g., memory management, file servers, network stack, device drivers
- **Note: microkernel by itself is not a complete OS!**
  - User-space parts implement OS functionality and provide application API
    - Low-level microkernel API is typically not exposed to applications

MICRO- KERNEL

HOW TO ORGANIZE THE USER-SPACE PARTS?

- **Single-server OS design is not good enough**
  - Still runs in single protection domain
- **Single failure is still fatal**
  - Microkernel may survive
  - Requires reboot of the OS
    - Applications and user data will be lost

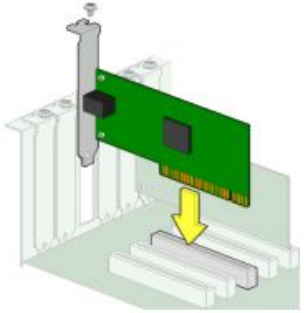


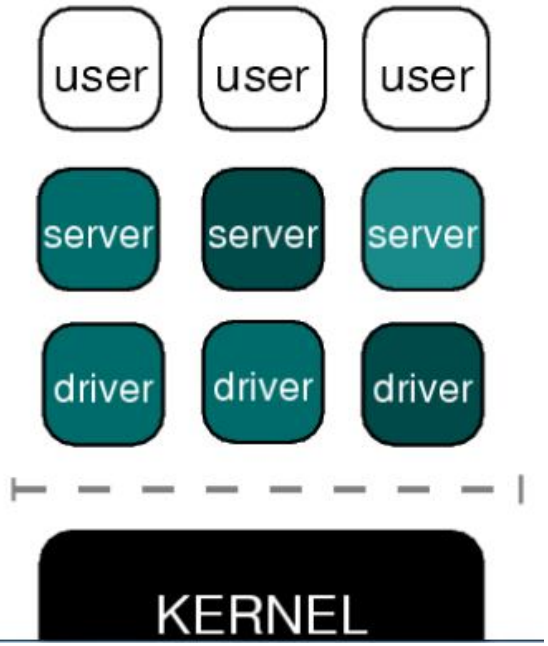
L4

MICRO- KERNEL

A COMPLETELY COMPARTMENTALIZED OS

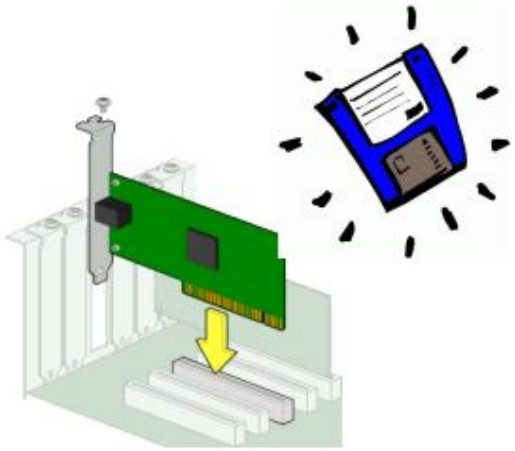
- Both servers and drivers are isolated in user space
- Separate protection domains

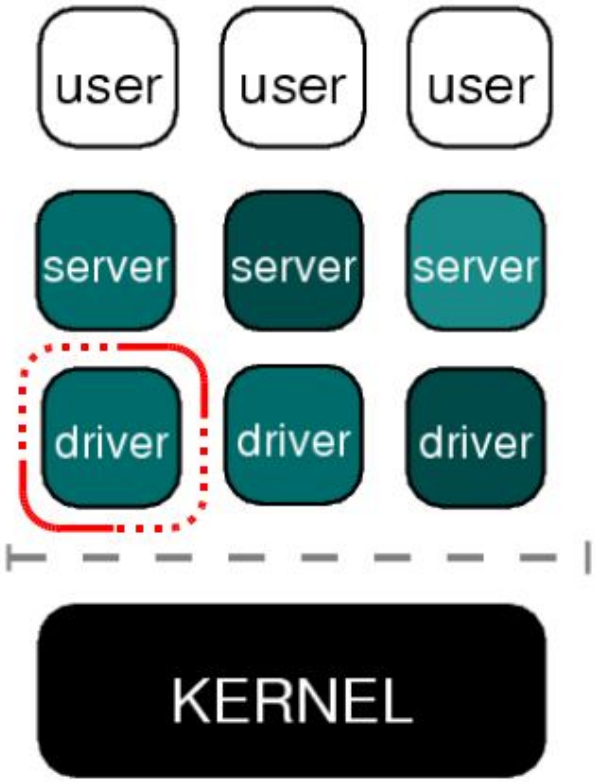




MICRO- KERNEL

- **Both servers and drivers are isolated in user space**
- **Separate protection domains**
  - Local failures cannot spread





MICRO- KERNEL

HOW DOES A USER-MODE DRIVER WORK?

- **Driver is unprivileged process with private address space**
  - Just like ordinary application program such as Firefox
- **Drivers require special privileges, however**
  - For example, exchange data with file server and other drivers
  - Perform device I/O and hardware interrupt handling
- **Only kernel has these privileges, so request for help**
  - Drivers can also request services from other OS servers

MICRO- KERNEL

- **Interprocess communication (IPC) facilities**
  - Usually works by passing messages between processes
  - Trap to kernel in order to have message copied from A to B
    - Process A does IPC_SEND(B, &message)
    - Process B does IPC_RECEIVE(ANY, &message)
- **Calls kernel's system task to perform privileged operations**
  - SAFECOPY: capability-protected data copy between processes
  - DEVIO: kernel performs device I/O on behalf of driver
  - IRQCTL: control IRQ line and acknowledge hardware interrupts
    - Kernel forwards interrupts to driver in an IPC message

MICRO- KERNEL

- Communication overhead
- Copying of data

- **Times have changed ...**
  - New insights reduced performance penalty (only 5-10%)
    - Results from four independent studies
  - Absolute performance penalty is minimal these days
  - Users gladly sacrifice some performance for reliability

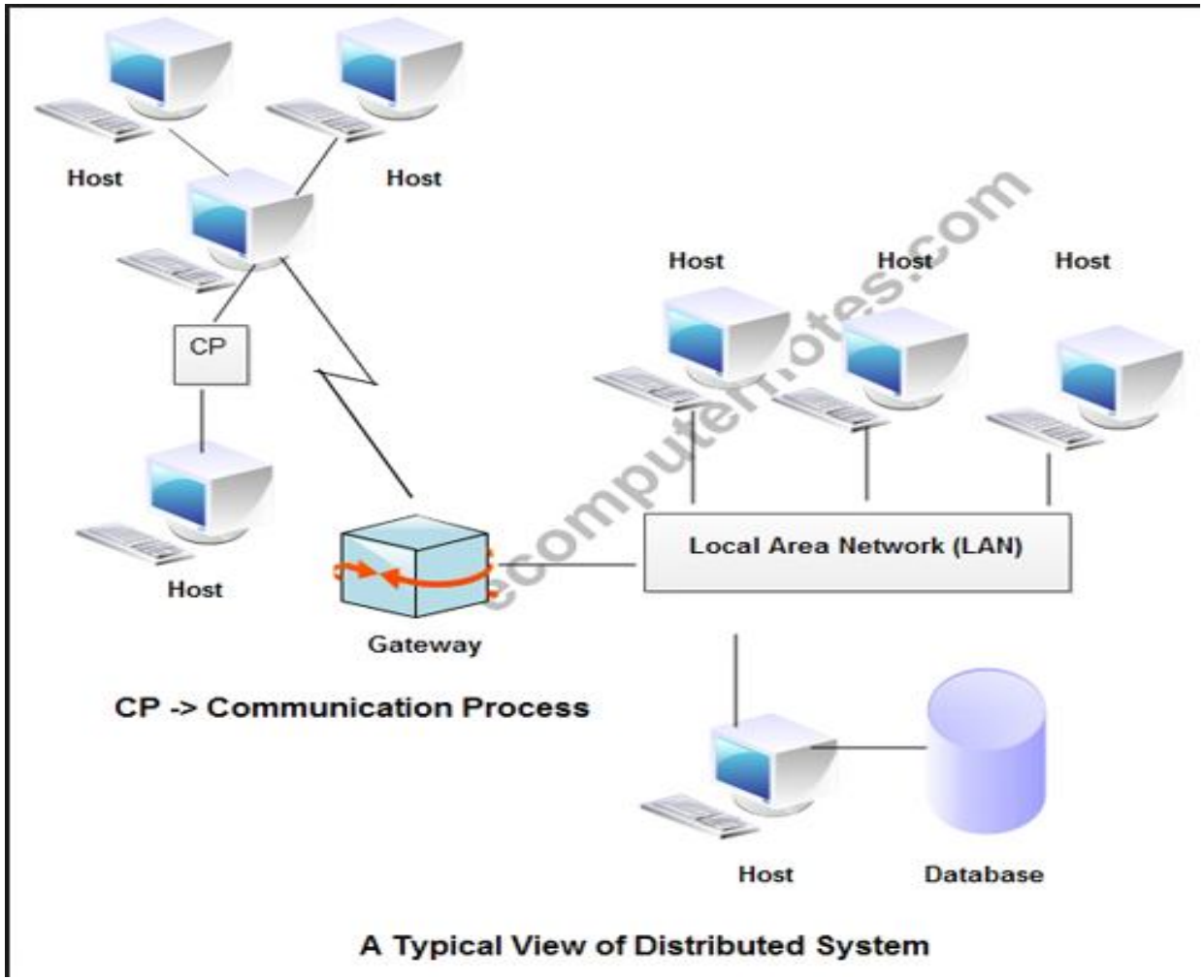
UNIT -II

REVIEW OF NETWORK OPERATING SYSTEM

DISTRIBUTED OS

- A **distributed** operating system is a software over a collection of independent, networked, communicating, and physically separate computational nodes. Each individual node holds a specific software subset of the global aggregate operating system. Each subset is a composite of two distinct service provisioners.

DISTRIBUTED OS



DISTRIBUTED OS

A distributed operating system must be designed to provide all the advantages of a distributed system to its users. That is, the users should be able to view a distributed system as a virtual centralized system that is flexible, efficient, reliable, secure, and easy to use. To meet this challenge, the designers of a distributed operating system must deal with several design issues. Some of the key design issues are described below.

- Access transparency.
- Location transparency.
- Replication transparency.
- Failure transparency.
- Migration transparency.
- Concurrency transparency.
- Performance transparency.
- Scaling transparency.

DISTRIBUTED OS

4.2.Reliability

Distributed system, which manages multiple resources, must be designed properly to increase the system's reliability by taking full advantage of this characteristic feature of a distributed system. For higher reliability, the fault-handling mechanisms of a distributed operating system must be designed properly to avoid faults, to tolerate faults, and to detect and recover from faults. Commonly used methods for dealing with these issues are fault avoidance and fault tolerance.

4.3.Flexibility

Another important issue in the design of distributed operating systems is flexibility. Flexibility is the most important feature for open distributed systems. The design of distributed operating system should be flexible due to the following reasons:

- Ease of modification
- Ease of enhancement

DISTRIBUTED OS

4.4.Performance

The overall performance should be better than or at least equal to that of running the same application on a single-processor system. Some design principles considered useful for better performance are as follows:

- Batch if possible.
- Cache whenever possible.
- Minimize copying of data.
- Minimize network traffic.
- Take advantage of fine-grain parallelism for multiprocessing.

4.5.Scalability

A distributed operating system should be designed to easily cope with the growth of nodes and users in the system. That is, such growth should not cause serious disruption of service or significant loss of performance to users. Some guiding principles for designing scalable distributed systems are as follows:

- Avoid centralized entities.
- Avoid centralized algorithms.

DISTRIBUTED OS

4.6.Heterogeneity

A heterogeneous distributed system consists of interconnected sets of dissimilar hardware or software systems. Because of the diversity, designing heterogeneous distributed systems is far more difficult than designing homogeneous distributed systems, in which each system is based on the same, or closely related, hardware and software. However, as a consequence of large scale, heterogeneity is often inevitable in distributed systems [2]. Furthermore, often heterogeneity is preferred by many users because heterogeneous distributed systems provide the flexibility to their users of different computer platforms for different applications.

DISTRIBUTED OS

4.7.Security

In order that the users can trust the system and rely on it, the various resources of a computer system must be protected against destruction and unauthorized access. Enforcing security in a distributed system is more difficult than in a centralized system because of the lack of a single point of control and the use of insecure networks for data communication. Therefore, as compared to a centralized system, enforcement of security in a distributed system has the following additional requirements [12]:

- It should be possible for the sender of a message to know that the message was received by the intended receiver.
- It should be possible for the receiver of a message to know that the message was sent by the genuine sender.
- It should be possible for both the sender and receiver of a message to be guaranteed that the contents of the message were not changed while it was in transfer.

COMPUTER N/W

Today the world scenario is changing. Data Communication and network have changed the way business and other daily affair works. Now, they rely on computer networks and internetwork. A set of devices often mentioned as nodes connected by media link is called a Network. A node can be a device which is capable of sending or receiving data generated by other nodes on the network like a computer, printer etc. These links connecting the devices are called Communication channels.

Computer network is a telecommunication channel through which we can share our data. It is also called data network. The best example of computer network is Internet. Computer network does not mean a system with control unit and other systems as its slave. It is called a distributed system

A network must be able to meet certain criteria, these are mentioned below:

1. Performance
2. Reliability
3. Scalability

COMPUTER N/W

Performance

It can be measured in following ways :

- **Transit time :** It is the time taken to travel a message from one device to another.
- **Response time :** It is defined as the time elapsed between enquiry and response.

Other ways to measure performance are :

1. Efficiency of software
2. Number of users
3. Capability of connected hardware

Reliability

It decides the frequency at which network failure take place. More the failures are, less is the network's reliability.

Security

It refers to the protection of data from the unauthorised user or access. While travelling through network, data passes many layers of network, and data can be traced if attempted. Hence security is also a very important characteristic for Networks.

COMPUTER N/W

Properties of Good Network

1. **Interpersonal Communication :** We can communicate with each other efficiently and easily example emails, chat rooms, video conferencing etc.
2. **Resources can be shared :** We can use the resources provided by network such as printers etc.
3. **Sharing files, data :** Authorised users are allowed to share the files on the network.

Basic Communication Model

Communication model is used to exchange data between two parties. For example communication between a computer, server and telephone (through modem).

COMPUTER N/W

Source

Data to be transmitted is generated by this device, example: telephones, personal computers etc.

Transmitter

The data generated by the source system are not directly transmitted in the form they are generated. The transmitter transforms and encodes the information in such a form to produce electromagnetic waves or signals.

Transmission System

A transmission system can be a single transmission line or a complex network connecting source and destination.

Receiver

Receiver accepts the signal from the transmission system and converts it to a form which is easily managed by the destination device.

Destination

Destination receives the incoming data from the receiver.

COMPUTER N/W

Data Communication

The exchange of data between two devices through a transmission medium is Data Communication. The data is exchanged in the form of 0's and 1's. The transmission medium used is wire cable. For data communication to occur, the communication device must be part of a communication system. Data Communication has two types Local and Remote which are discussed below :

Local :

Local communication takes place when the communicating devices are in the same geographical area, same building, face-to-face between individuals etc.

Remote :

Remote communication takes place over a distance i.e. the devices are farther. Effectiveness of a Data Communication can be measured through the following features :

1. Delivery : Delivery should be done to the correct destination.
2. Timeliness : Delivery should be on time.
3. Accuracy : Data delivered should be accurate.

COMPUTER N/W

Line Configuration in Computer Networks

Network is a connection made through connection links between two or more devices. Devices can be a computer, printer or any other device that is capable to send and receive data. There are two ways to connect the devices :

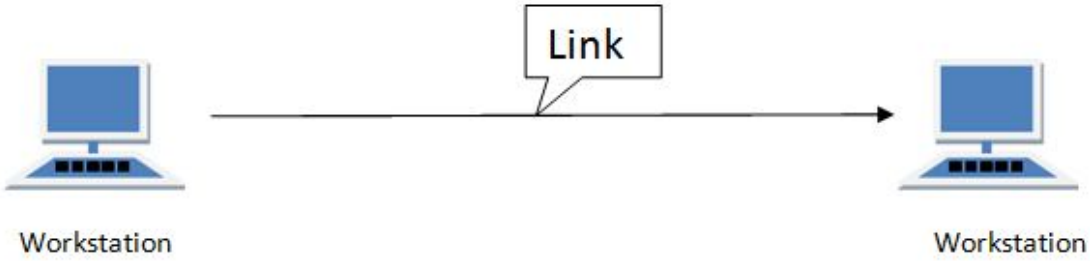
1. Point-to-Point connection
2. Multipoint connection

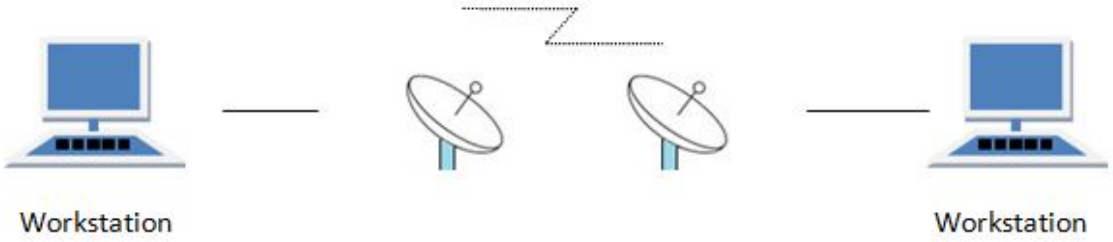
COMPUTER N/W

Point-To-Point Connection

It is a protocol which is used as a communication link between two devices. It is simple to establish. The most common example for Point-to-Point connection (PPP) is a computer connected by telephone line. We can connect the two devices by means of a pair of wires or using a microwave or satellite link.

Example: Point-to-Point connection between remote control and Television for changing the channels.





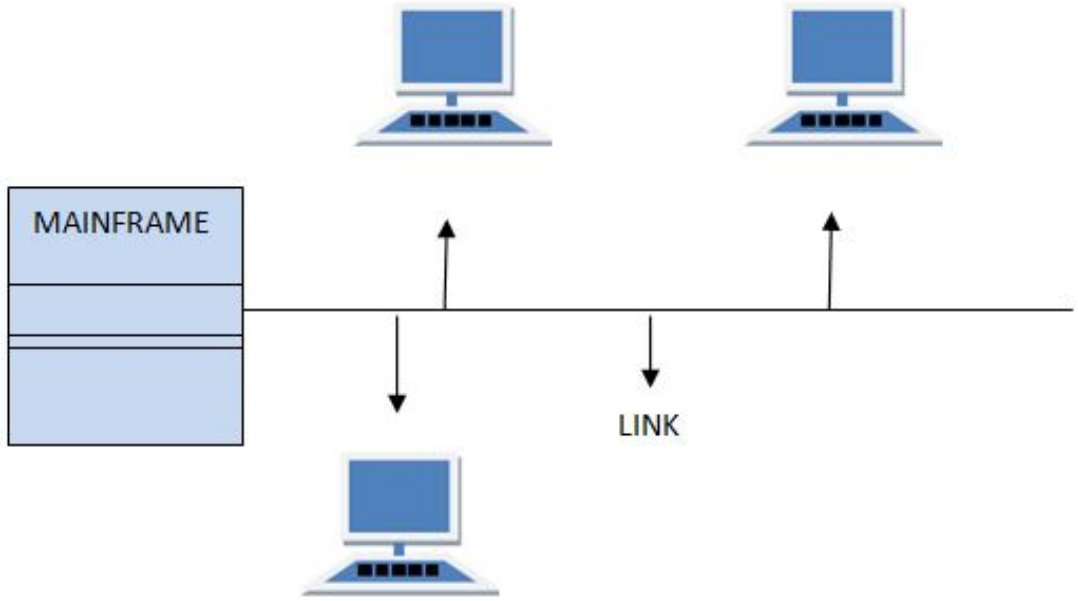
COMPUTER N/W

MultiPoint Connection

It is also called Multidrop configuration. In this connection two or more devices share a single link.

There are two kinds of Multipoint Connections :

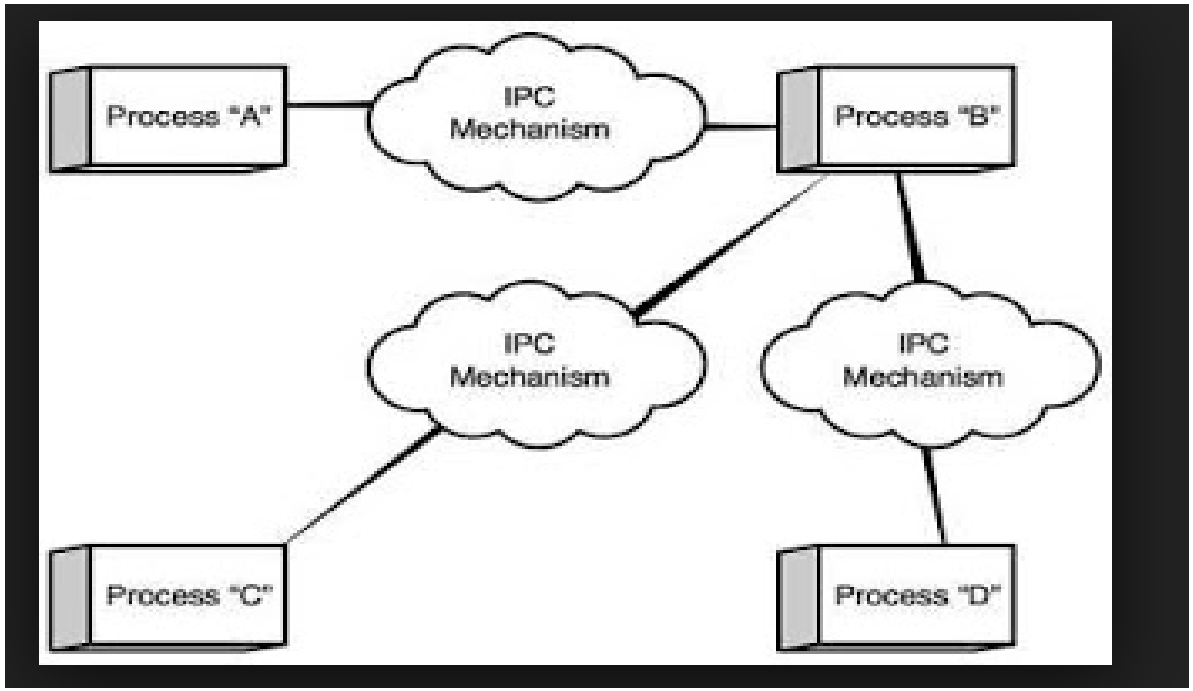
- If the links are used simultaneously between many devices, then it is spatially shared line configuration.
- If user takes turns while using the link, then it is time shared (temporal) line configuration.



Interprocess communication

- **Interprocess communication** (IPC) is a set of programming interfaces that allow a programmer to coordinate activities among different program**process**es that can run concurrently in an operating system. This allows a program to handle many user requests at the same time.

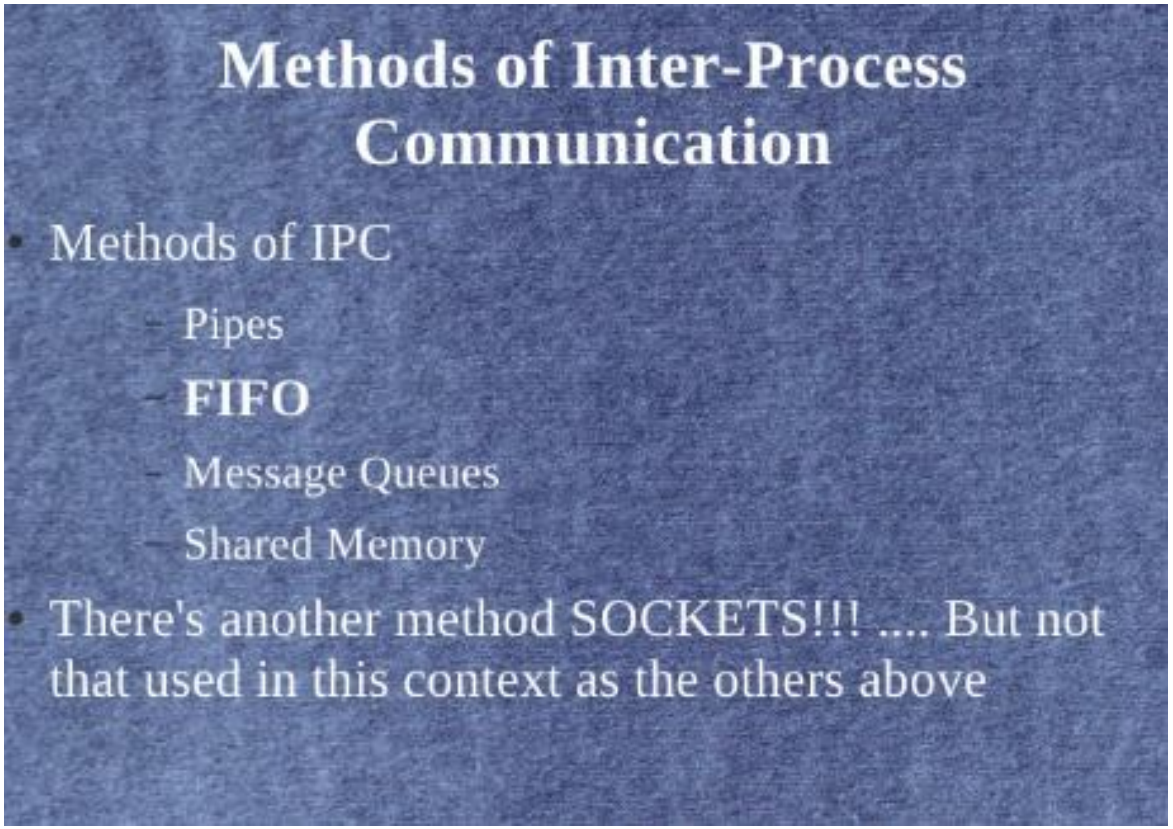
Interprocess communication



LINUX IPC

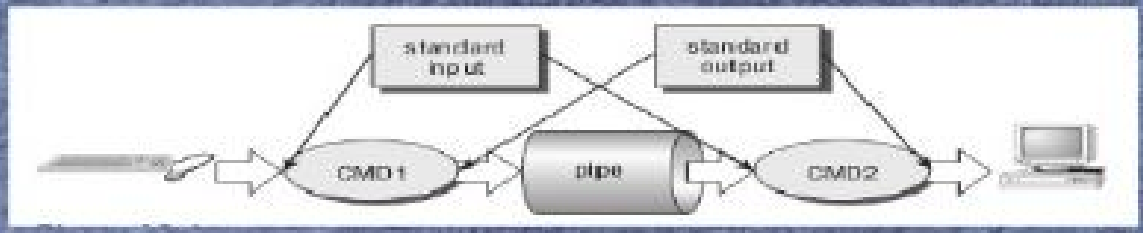
- **Interprocess Communication**Mechanisms. Processes **communicate** with each other and with the kernel to coordinate their activities. ... Signals and pipes are two of them but **Linux** also supports the System V IPC mechanisms named after the Unix ™ release in which they first appeared.

LINUX IPC


Methods of Inter-Process Communication
Methods of IPC
Pipes
FIFO
Message Queues
Shared Memory
There's another method SOCKETS!!! .... But not that used in this context as the others above

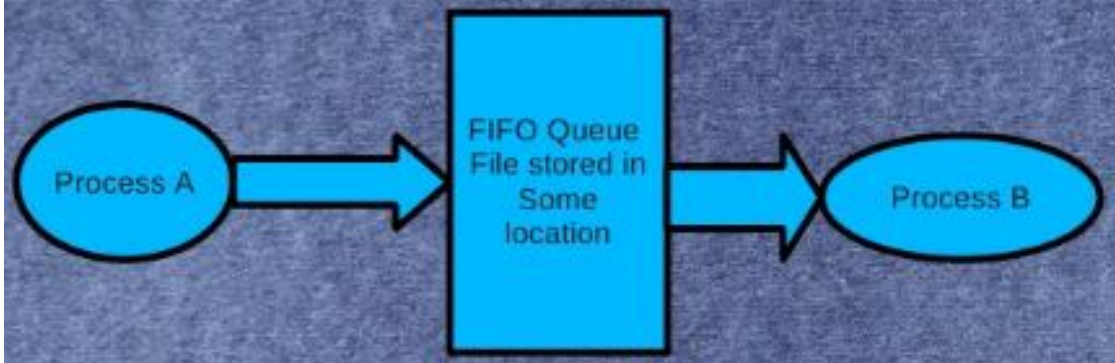
LINUX IPC

- A *pseudofile* which redirects data from one process to other.
- Below the "*cmd1*" and "*cmd2*" takes input from ***stdin*** and outputs to ***stdout*** , but a ***pipe*** in-between passes information from "***stdout*** of *cmd1*" to "***stdin*** of *cmd2*" which outputs to *stdin* i.e monitor
- e.g. *ls -aRl home | less ; cat file1 file2 file3 | grep sample*
- *Simplest of all the IPCs*



LINUX IPC

- Stands for First In First Out (obvious in CS)
- A named pipe implementation on Linux
- Unidirectional in nature (Simplex Communication)
- Two system calls available in sys/types.h
  - *int mkfifo(const char *pathname, mode_t mode)*
  - *int mknod(const char *path, mode_t mode, dev_t dev)*



LINUX IPC

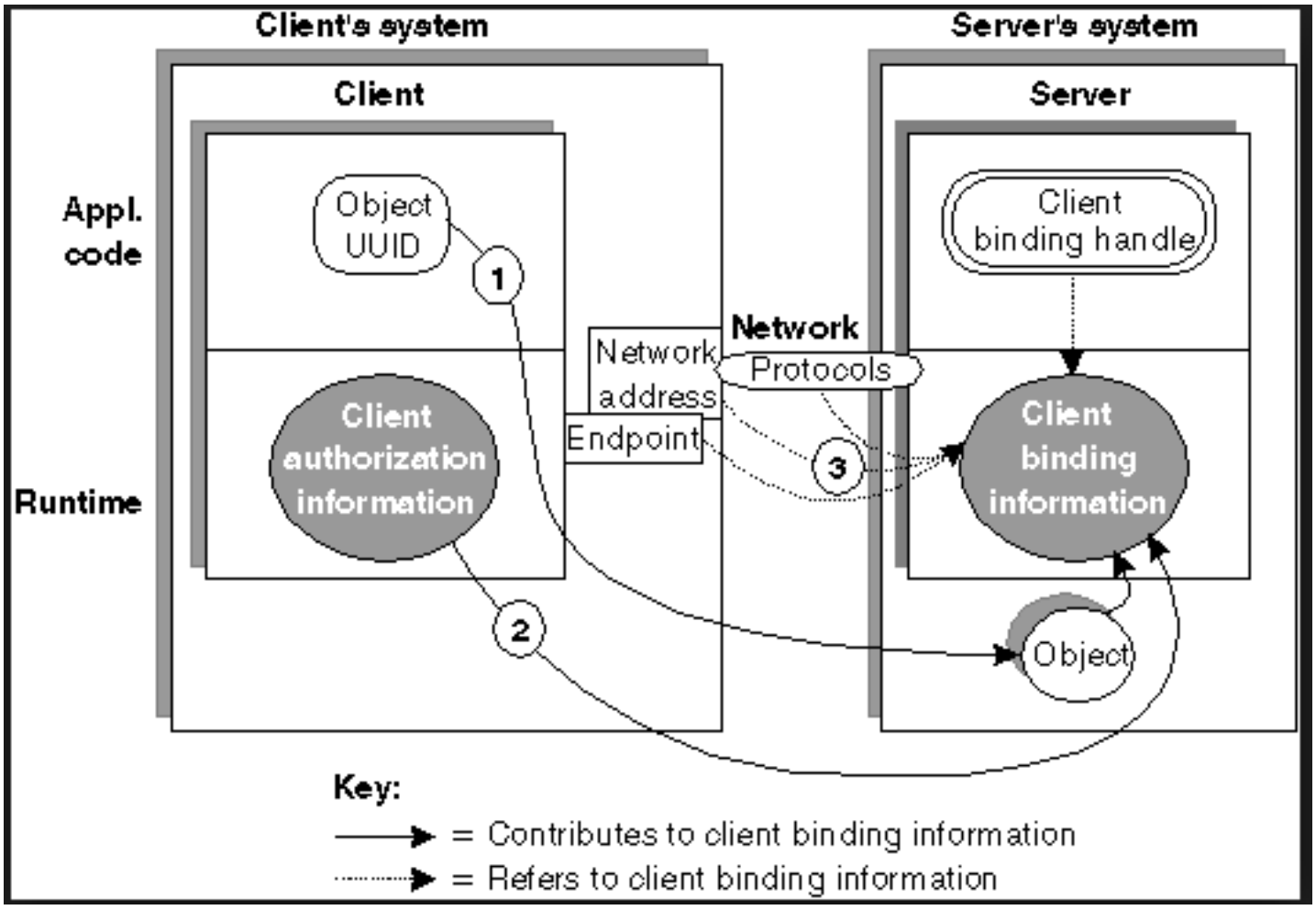
Message Queues

- Released in AT & T System V.2 (a release of Unix)
- Part of System V IPC, the other being Shared Memory and Semaphores
- Implementation of Message Passing IPC
- Message queue will remain (*until deleted*) , even if the process have existed
- Can be made unidirectional or bidirectional

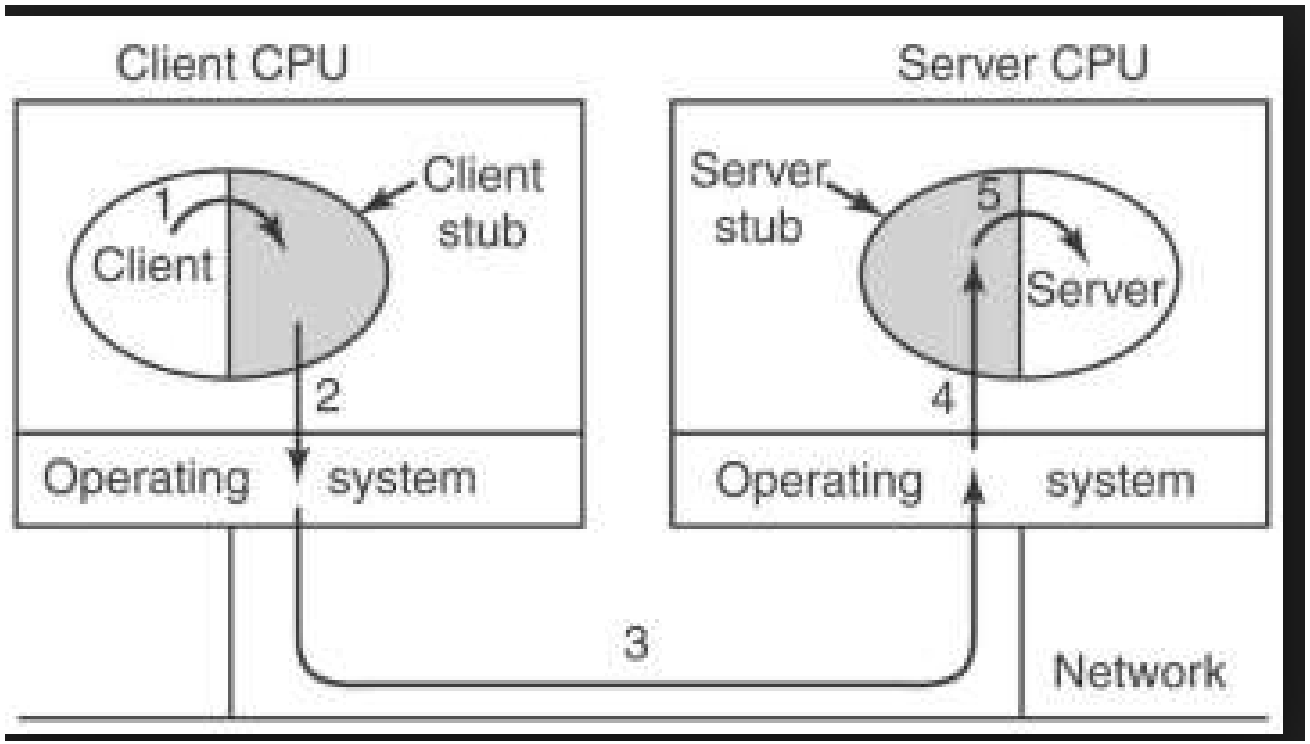
REMOTE PROCEDURE CALLS

- n distributed computing, a **remote procedure call** (**RPC**) is when a computer program causes a **procedure**(subroutine) to execute in another address space (commonly on another computer on a shared network), which is coded as if it were a normal (local) **procedure call**, without the programmer explicitly coding the details ...

REMOTE PROCEDURE CALLS



REMOTE PROCEDURE CALLS



REMOTE PROCEDURE CALLS

Message Passing Model

- Message passing is a general technique for exchanging information between two or more processes
- Basically an extension to the `send/recv` I/O API
  - *e.g.*, UDP, VMTP
- Supports a number of different communication styles
  - *e.g.*, request/response, asynchronous oneway, multicast, broadcast, etc.
- May serve as the basis for higher-level communication mechanisms such as RPC

Message Passing Model (cont'd)

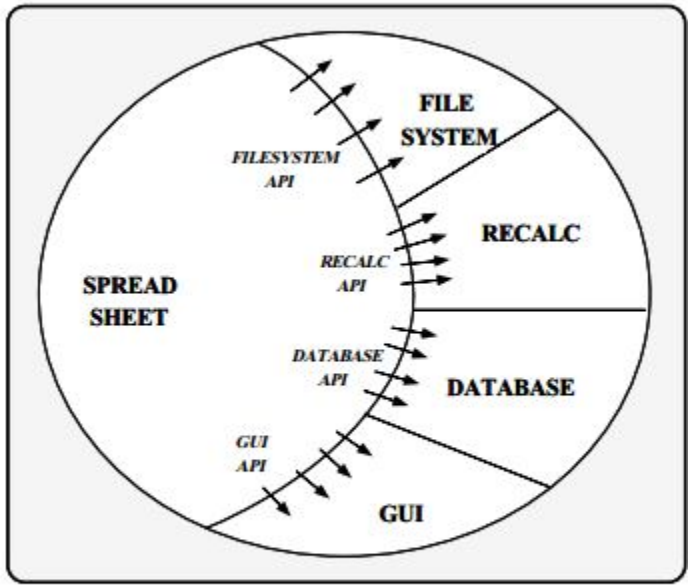
- In general, message passing does not make an effort to hide distribution
  - *e.g.*, network byte order, pointer linearization, addressing, and security must be dealt with explicitly
- This makes the model efficient and flexible, but also complicate and time consuming

REMOTE PROCEDURE CALLS

Message Passing Design Considerations

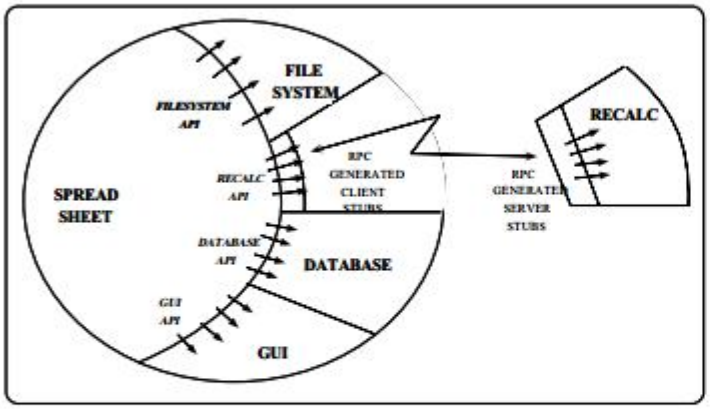
- *Blocking vs. nonblocking*
  - Affects reliablility, responsiveness, and program structure
- *Buffered vs. unbuffered*
  - Affects performance and reliability
- *Reliable vs. unreliable*
  - Affects performance and correctness

Monolithic Application Structure



REMOTE PROCEDURE CALLS

RPC Application Structure



- Note, RPC generators automate most of the work involved in separating client and server functionality

Basic Principles of RPC

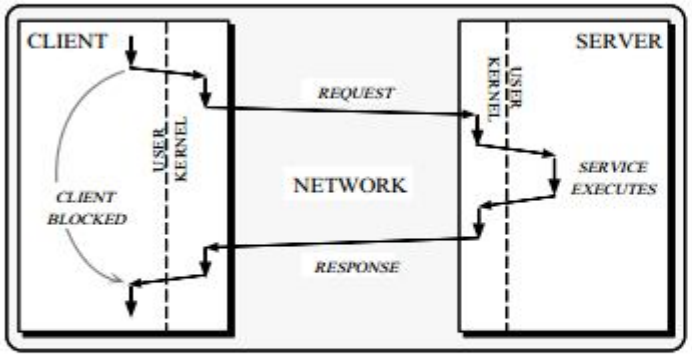
1. Use traditional programming style for distributed application development

2. Enable selective replacement of local procedure calls with remote procecure calls

   - *Local Procedure Call* (LPC)
     - A well-known method for transferring control from one part of a process to another
       - Implies a subsequent return of control to the caller
   - *Remote Procedure Call* (RPC)
     - Similar LPC, except a local process invokes a procedure on a remote system
       - *i.e.*, control is transferred *across* processes/hosts

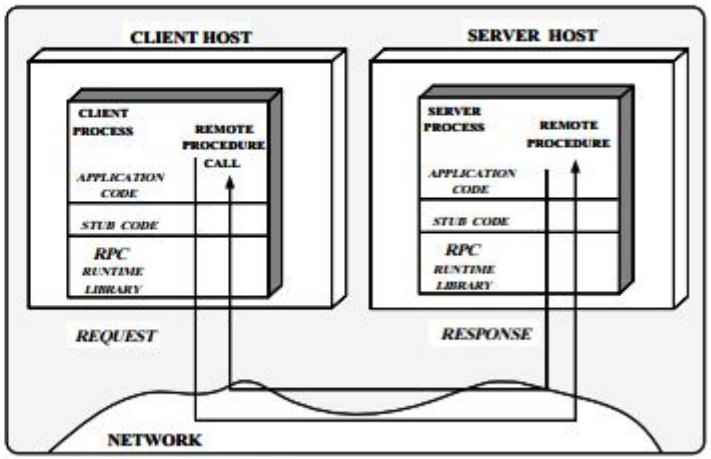
REMOTE PROCEDURE CALLS

A Temporal View of RPC



- An RPC protocol contains two sides, the *sender* and the *receiver* (*i.e.*, *client* and *server*)
  - However, a server might also be a client of another server and so on...

A Layered View of RPC

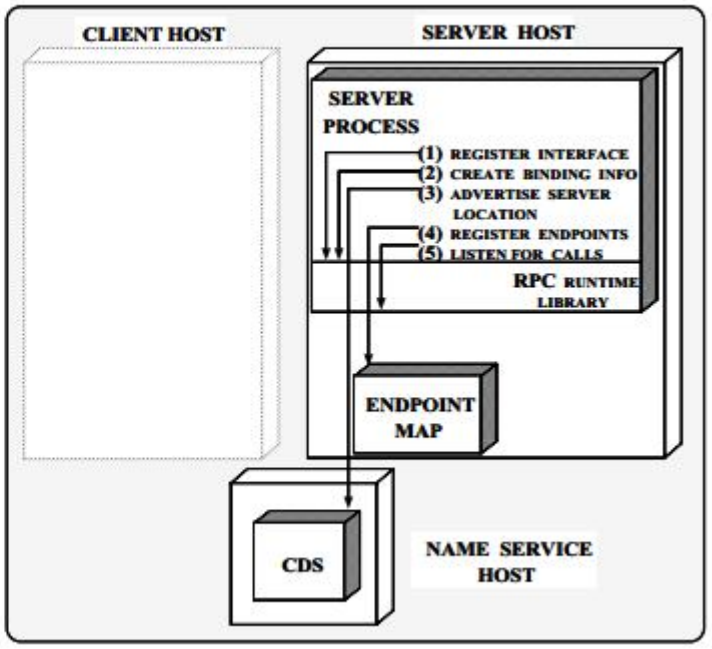


REMOTE PROCEDURE CALLS

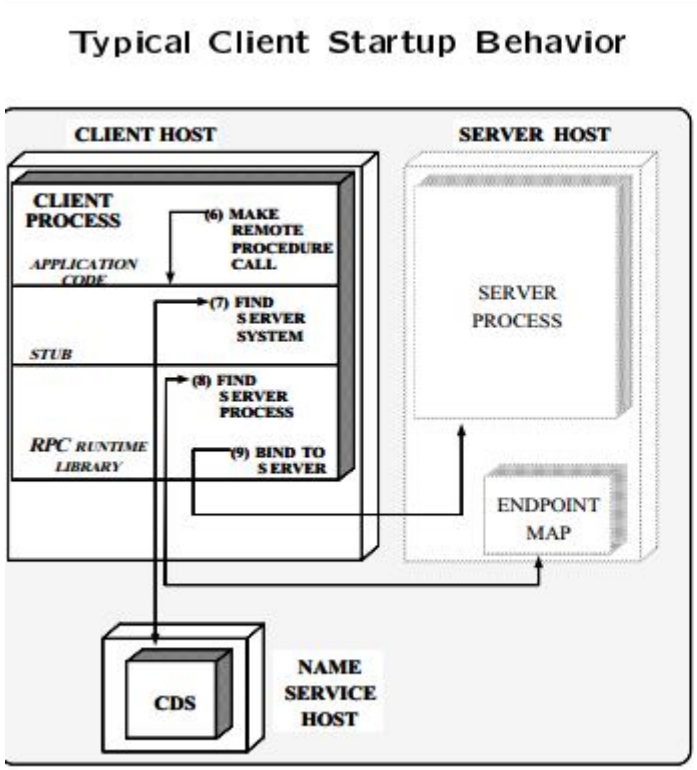
RPC Automation

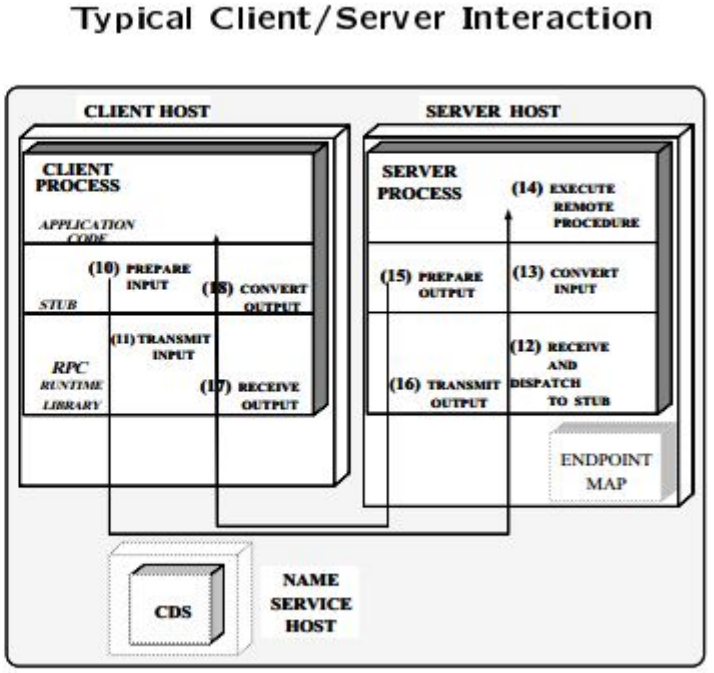
- To help make distribution transparent, RPC hides all the network code in the client *stubs* and server *skeletons*
  - These are usually generated automatically...
- This shields application programs from networking details
  - *e.g.*, sockets, parameter marshalling, network byte order, timeouts, flow control, acknowledgements, retransmissions, etc.
- It also takes advantage of recurring communcation patterns in network servers to generate most of the stub/skeleton code automatically

Typical Server Startup Behavior



REMOTE PROCEDURE CALLS



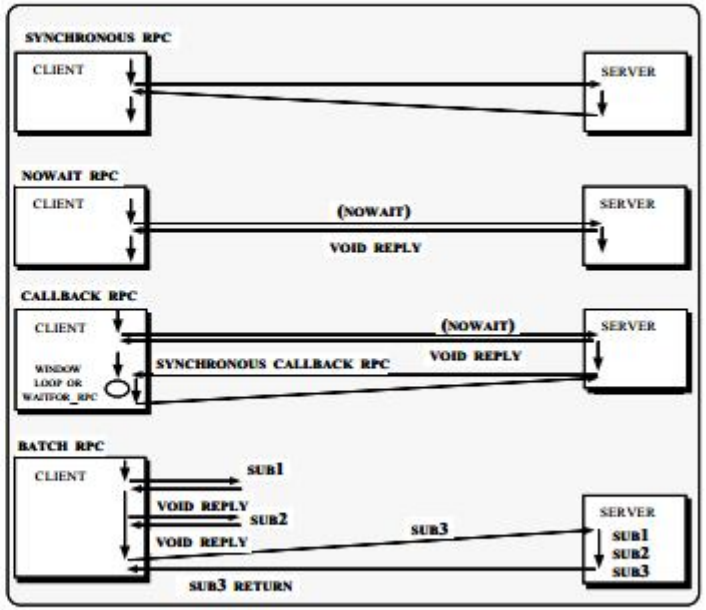


REMOTE PROCEDURE CALLS

RPC Models

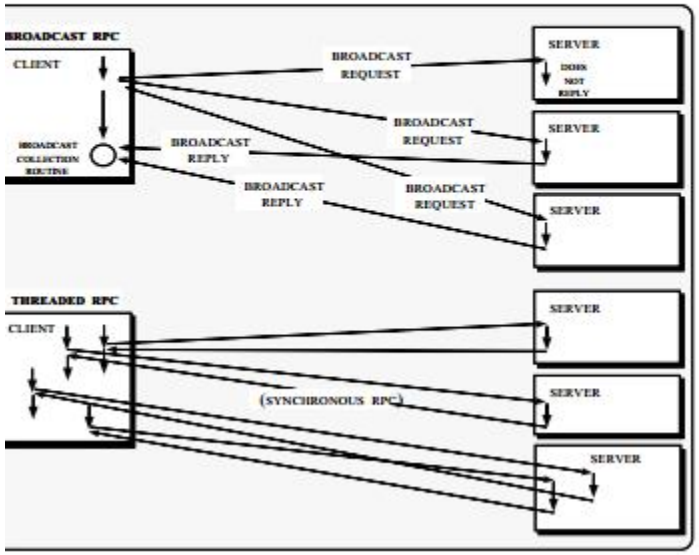
- There are several variations on the standard RPC "synchronous request/response" model
- Each model provides greater flexibility, at the cost of less transparency
- Certain RPC toolkits support all the different models
  - *e.g.*, ONC RPC
- Other DOC frameworks do not (due to portability concerns)
  - *e.g.*, OMG CORBA and OSF DCE

RPC Models



REMOTE PROCEDURE CALLS

RPC Models (cont'd)



Transparency Issues

- RPC has a number of limitations that must be understood to use the model effectively
  - Most of the limitations center around *transparency*
- Transforming a simple local procedure call into *system calls*, *data conversions*, and *network communications* increases the chance of something going wrong
  - *i.e.*, it reduces the *transparency* of distribution

REMOTE PROCEDURE CALLS

Tranparency Issues (cont'd)

- Key Aspects of RPC Transparency
  1. *Parameter passing*
  2. *Data representation*
  3. *Binding*
  4. *Transport protocol*
  5. *Exception handling*
  6. *Call semantics*
  7. *Security*
  8. *Performance*

Parameter Passing

- Functions in an application that runs in a single process may collaborate via parameters and/or global variables
- Functions in an application that runs in multiple processes on the same host may collaborate via message passing and/or non-distributed shared memory
- However, passing parameters is typically the only way that RPC-based clients and servers share information
  - Hence, we have already given up one type of transparency...

REMOTE PROCEDURE CALLS

Parameter Passing (cont'd)

- Passing parameters across process/host boundaries is surprisingly tricky...
- Parameters that are passed by value are fairly simple to handle
  - The client stub copies the value from the client and packages into a network message
  - Presentation issues are still important, however
- Parameters passed by reference are *much* harder
  - *e.g.*, in C when the address of a variable is passed
    - *e.g.*, passing arrays
  - Or more generally, handling pointer-based data structures
    - *e.g.*, pointers, lists, trees, stacks, graphs, etc.

Parameter Passing (cont'd)

- Typical solutions include:
  - Have the RPC protocol only allow the client to pass arguments by value
    - However, this reduces transparency even further!
  - Use a presentation data format where the user specifically defines what the input arguments are and what the return values are
    - *e.g.*, Sun's XDR routines
  - RPC facilities typically provide an "interface definition language" to handle this
    - *e.g.*, CORBA or DCE IDL

REMOTE PROCEDURE CALLS

Data Representation

- RPC systems intended for heterogeneous environments must be sensitive to byte-ordering differences
  - They typically provide tools for automatically performing data conversion (*e.g.*, `rpcgen` or `idl`)
- Examples:
  - *Sun RPC (XDR)*
    - Imposes "canonical" big-endian byte-ordering
    - Minimum size of any field is 32 bits
  - *Xerox Courier*
    - Uses big-endian
    - Minimum size of any field is 16 bits

Data Representation (cont'd)

- Examples (cont'd)
  - *DCE RPC (NDR)*
    - Supports multiple presentation layer formats
    - Supports "receiver makes it right" semantics...
      - Allows the sender to use its own internal format, if it is supported
    - The receiver then converts this to the appropriate format, if different from the sender's format
      - This is more efficient than "canonical" big-endian format for little-endian machines

REMOTE PROCEDURE CALLS

Binding

- Binding is the process of mapping a request for a service onto a physical server somewhere in the network
  - Typically, the client contacts an appropriate name server or "location broker" that informs it which remote server contains the service
    - Similar to calling 411...
- If service migration is supported, it may be necessary to perform this operation multiple times
  - Also may be necessary to leave a "forwarding" address

Binding (cont'd)

- There are two components to binding:
  1. Finding a remote host for a desired service
  2. Finding the correct service on the host
     - *i.e.*, locating the "process" on a given host that is listening to a well-known port
- There are several techniques that clients use to locate a host that provides a given type of service
  - These techniques differ in terms of their performance, transparency, accuracy, and robustness

REMOTE PROCEDURE CALLS

Call Semantics (cont'd)

- Note that if a connectionless transport protocol is used then achieving "at most once" semantics becomes more complicated
  - The RPC framework must use sequence numbers and cache responses to ensure that duplicate requests aren't executed multiple times
- Note that accurate distributed timestamps are useful for reducing the amount of state that a server must cache in order to detect duplicates

Security

- Typically, applications making local procedure calls do not have to worry about maintaining the integrity or security of the caller/callee
  - *i.e.*, calls are typically made in the same address space
    - ▹ Note that shared libraries may complicate this...
- Local security is usually handled via access control or special process privileges
- Remote security is handled via distributed authentication protocols
  - *e.g.*, Kerberos...

REMOTE PROCEDURE CALLS

- Usually the performance loss from using RPC is an order of magnitude or more, compared with making a local procedure call due to
  1. *Protocol processing*
  2. *Context switching*
  3. *Data copying*
  4. *Network latency*
  5. *Congestion*

- Note, these sources of overhead are ubiquitous to networking...

- RPC also tends to be much slower than using lower-level remote IPC facilities such as sockets directly due to overhead from
  1. *Presentation conversion*
  2. *Data copying*
  3. *Flow control*
     - *e.g.*, stop-and-wait, synchronous client call behavior
  4. *Timer management*
     - Non-adaptive (consequence of LAN upbringing)

- Note, these sources of overhead are typical of RPC...

REMOTE PROCEDURE CALLS

Summary

- RPC is one of several models for implementing distributed communication
  - It is particular useful for transparently supporting request/response-style applications
  - However, it is not appropriate for all applications due to its performance overhead and lack of flexibility
- Before deciding on a particular communication model it is crucial to carefully analyze the distributed requirements of the applications involved
  - Particularly the tradeoff of security for performance...

3. Synchronization in Distributed Systems

- In a centralized system: all processes reside on the same system utilize the same clock.

- In a distributed system: like synchronize everyone’s watch in the classroom.

Global Time

- **Global Time** is utilized to provide timestamps for processes and data.
- ✓ **Physical clock**: concerned with “People” time
- ✓ **Logical clock**: concerned with relative time and maintain logical consistency

Physical Clocks

- There are two aspects:

✓ Obtaining an accurate value for physical time

✓ Synchronizing the concept of physical time throughout the distributed system

■ These can be implemented using **centralized algorithms** or **distributed algorithms**

Obtaining an Accurate Physical Time

- **A physical time server** is needed to access the current time from a universal time coordinator (**UTC**).
- Two sources for UTC:

✓ WWV shortwave radio station in Ft. Collins, Colorado

✓ Geostationary Operational Environmental Satellites (GEOS)

Synchronizing Physical Time

- The difference in time between two clocks due to drifting is defined as **clock skew**. As long as any and every two clocks differ by a value less than the maximum skew value, the time service is considered to be maintaining synchronization.

How to synchronize two clocks in **A** and **B**?

- The information necessary to read the value must be communicated across the network to location B.
- B's clock value must be read.
- B's clock value is communicated back to location A.
- B's clock value is adjusted to reflect the time necessary to travel across the network.
- B's clock value is compared to A's clock value.

Centralized Physical Time Services

- Broadcast Based
- Request Driven

Broadcast Based – first approach

- The centralized time server's action:

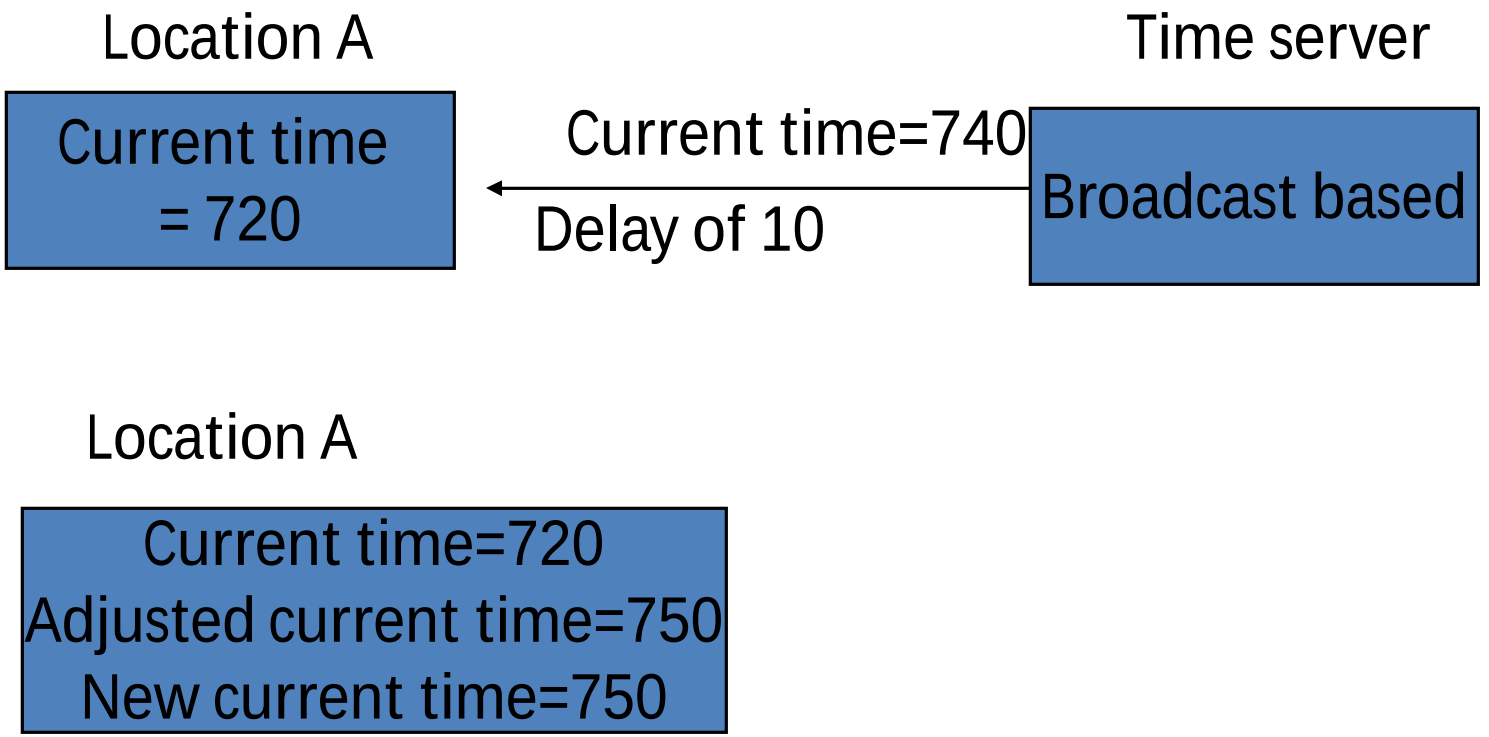
  The physical time service broadcasts periodically the current time to members of the distributed systems.

- The participants' action:

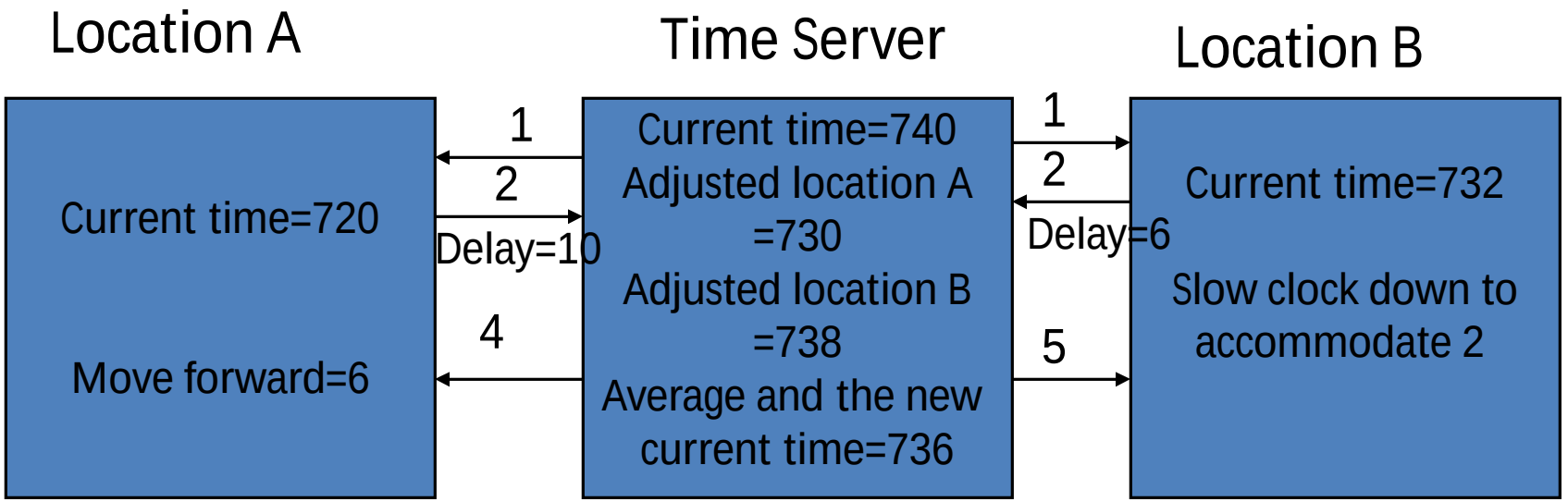
✓ If a given participant's clock is ahead of the time server's clock, the participant slows down its clock so that it will continually move closer to the accurate time.

✓ If a participant's clock is behind the time server's clock, the participant moves its clock forward. Alternatives do include gradually speeding up the clock.

For example

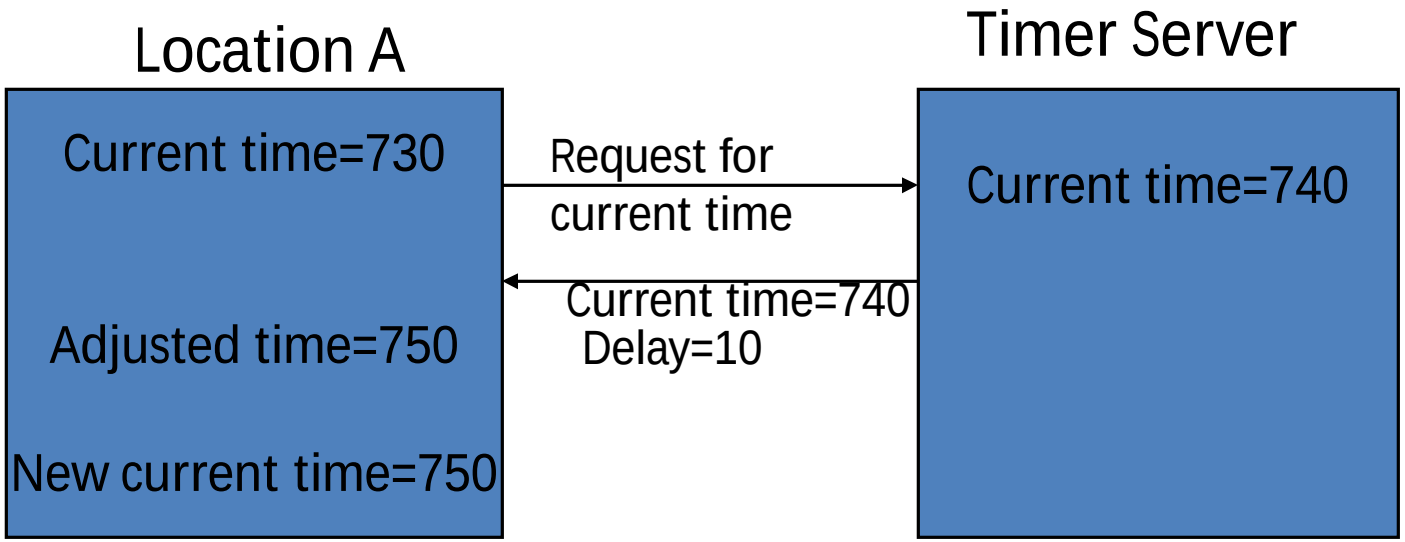

Location A
Time server
Current time
= 720
Current time=740
Delay of 10
Broadcast based
Location A
Current time=720
Adjusted current time=750
New current time=750

Broadcast Based – second approach (Berkeley algorithm)



1. Current time = 740
2. My current time = 720
3. My current time = 732
4. Adjust forward = 6
5. Adjust slowdown to accommodate 2

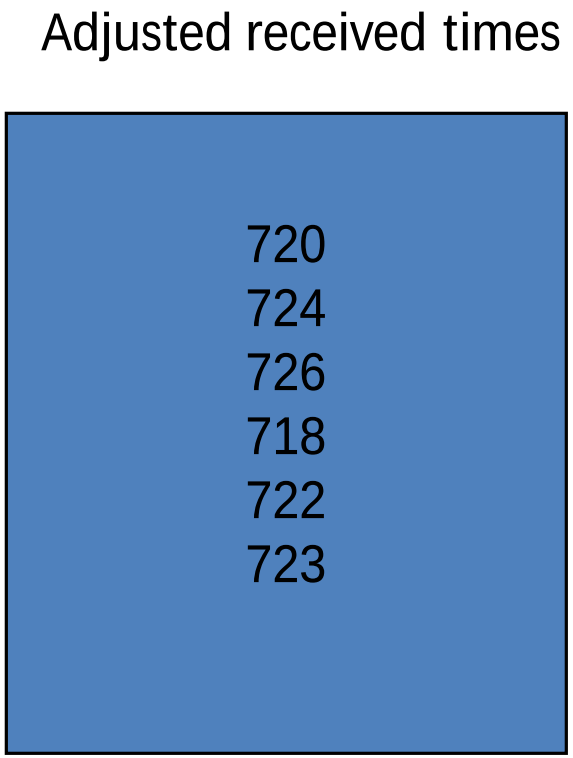
Request Driven



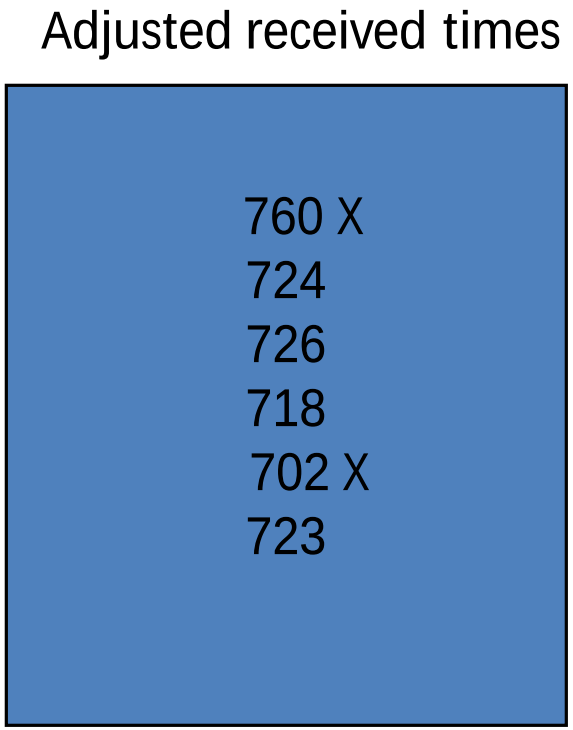
Distributed Physical Time Service

- Each location broadcasts its current time at predefined set intervals. Once a location has broadcast its time, it starts a timer. It then collects time messages that it receives. Each time message that arrives is stamped with the local current time. This process continues until the timer expires. Upon the expiration of the timer, each message is adjusted to reflect the network delay time estimated for the message source. At this stage, the participant calculates the average time according to one of the following approaches:

- Calculate the average of all messages

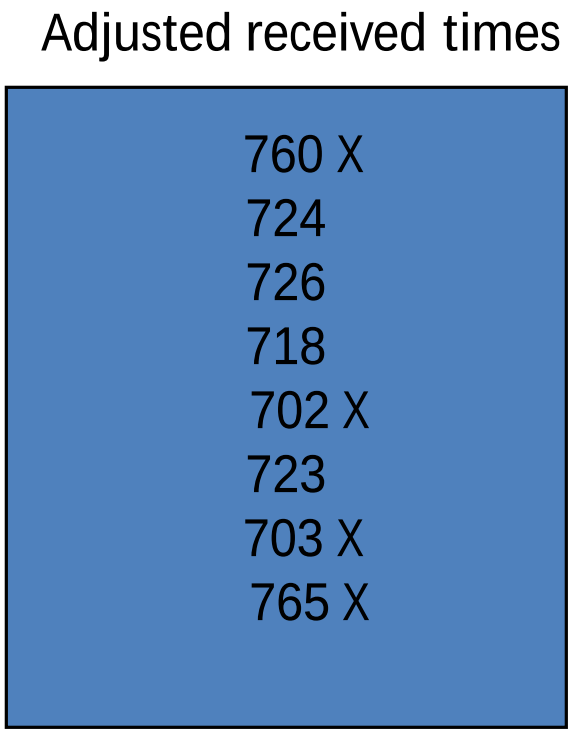


- Delete the times that are above the threshold and then average the rest.



The numbers besides X are deleted.
The rest are averaged.

- Discard the highest *x* and the lowest *x* values and then average



Logical Clocks

- **Why Logical Clocks?**

  It is difficult to utilize physical clocks to order events uniquely in distributed systems.
- The essence of logical clocks is based on the happened-before relationship presented by **Lamport.**

Happen-Before Relationship

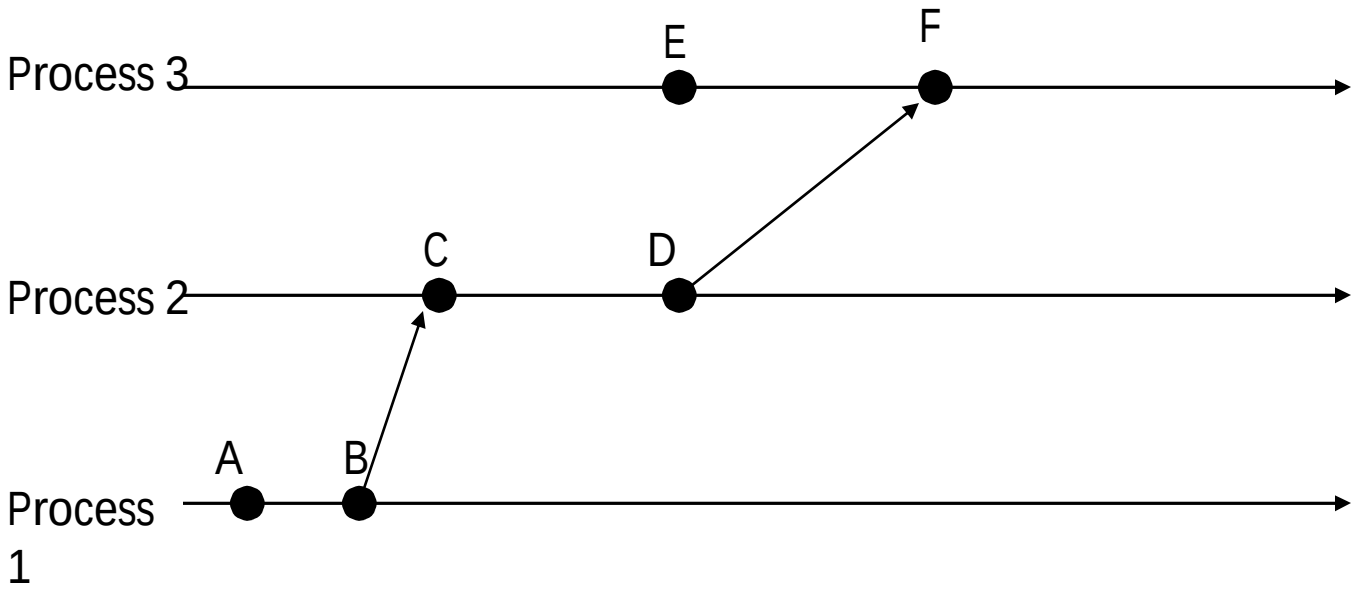
- If two events, *a* and *b*, occurred at the same process, they occurred in the order of which they were observed. That is, $a > b$.
- If a sends a message to *b*, then $a > b$. That is, you cannot receive something before it is sent. This relationship holds regardless of where events *a* and *b* occur.

- The happen-before relationship is transitive. If *a* happens before *b* and *b* happens before *c*, then *a* happens before *c*. That is, if $a > b$ and $b > c$, then $a > c$.

Logical Ordering

- If *T(a)* is the timestamp for event *a*, the following relationships must hold in a distributed system utilizing logical ordering.
- If two events, *a* and *b*, occurred at the same process, they occurred in the order of which they were observed. That is *T(a)* > *T(b).*
- If a sends a message to *b*, then *T(a)* > *T(b).*
- If *a* happens before *b* and *b* happens before *c*, *T(a)* > *T(b), T(b)* > *T(c),* and *T(a)* > *T(c).*

For example

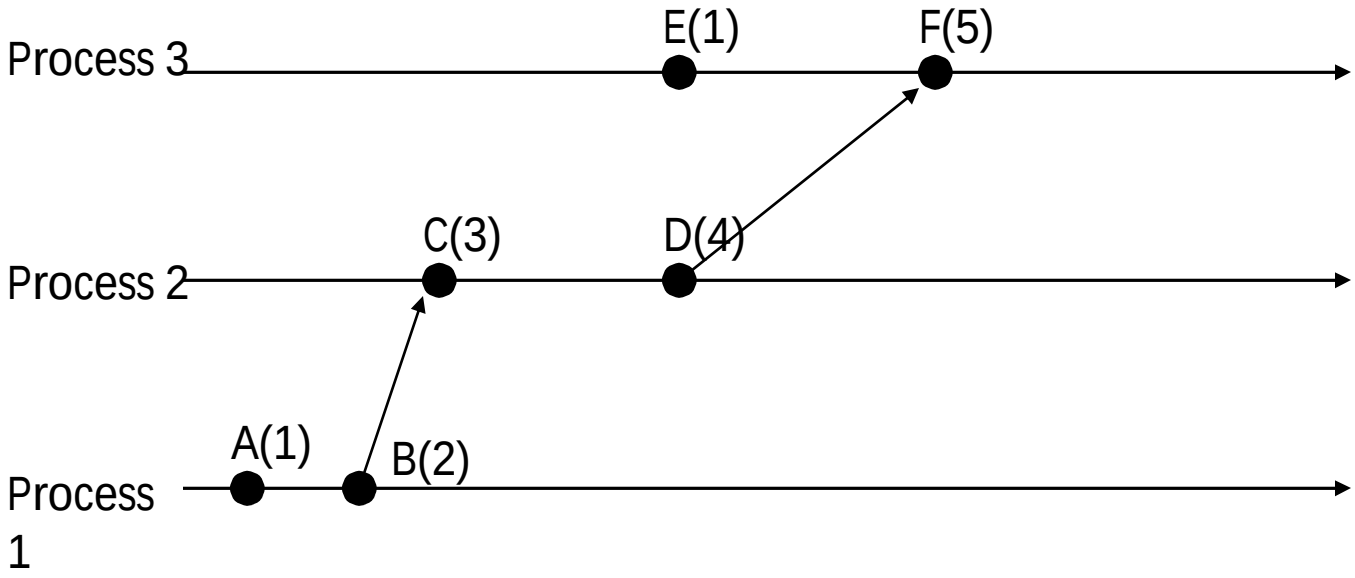


A>B>C>D>F E

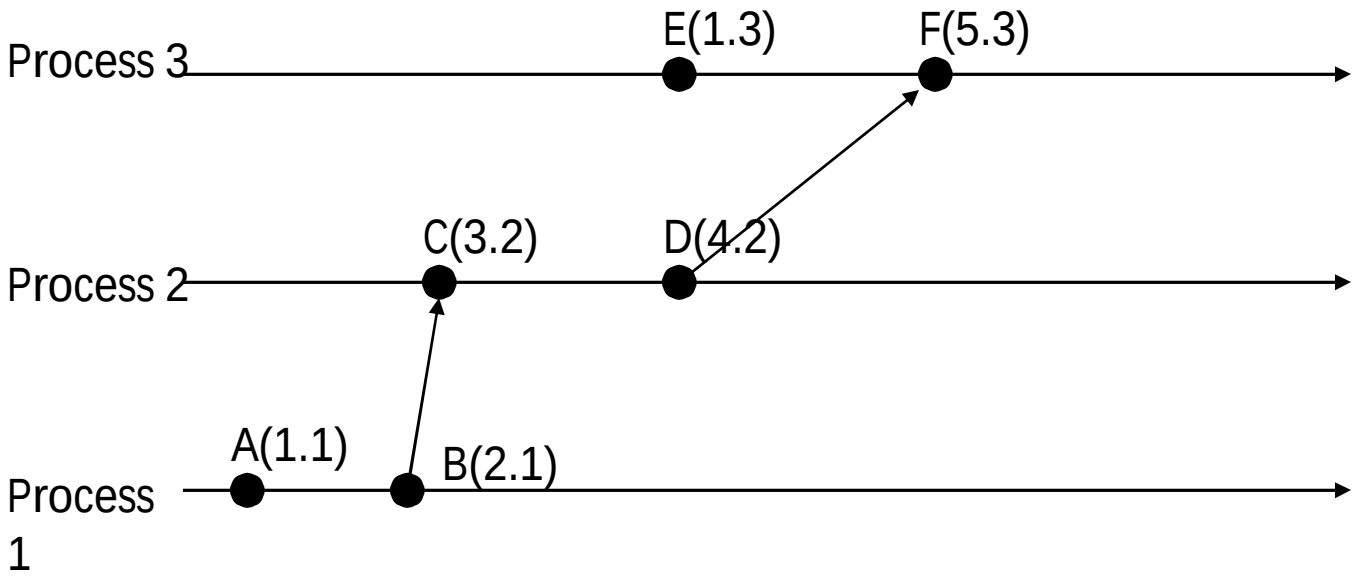
Lamport's Algorithm

- Each process increments its clock counter between every two consecutive events.
- If *a* sends a message to *b*, then the message must include *T(a).* Upon receiving *a* and *T(a),* the receiving process must set its clock to the greater of [*T(a)+d, Current Clock*]. That is, if the recipient's clock is behind, it must be advanced to preserve the happen-before relationship. Usually *d=1*.

For example



Total Ordering with Logical Clocks

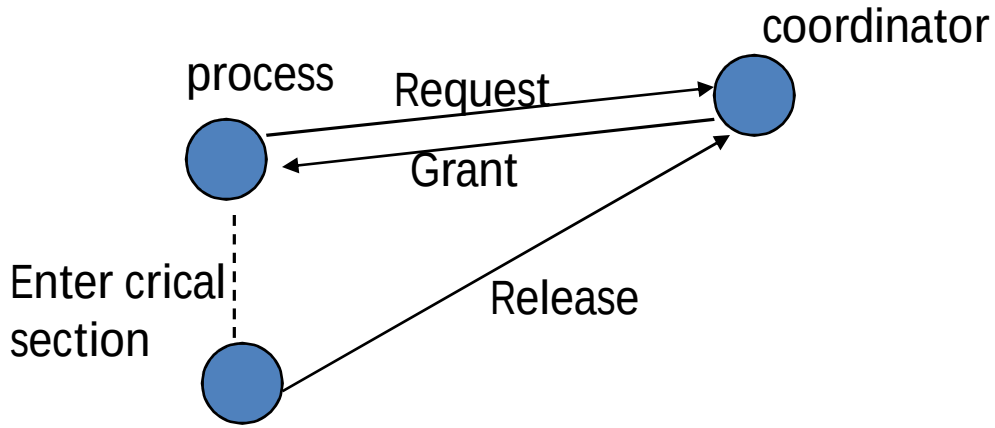


A>E>B>C>D>F

Mutual Exclusion

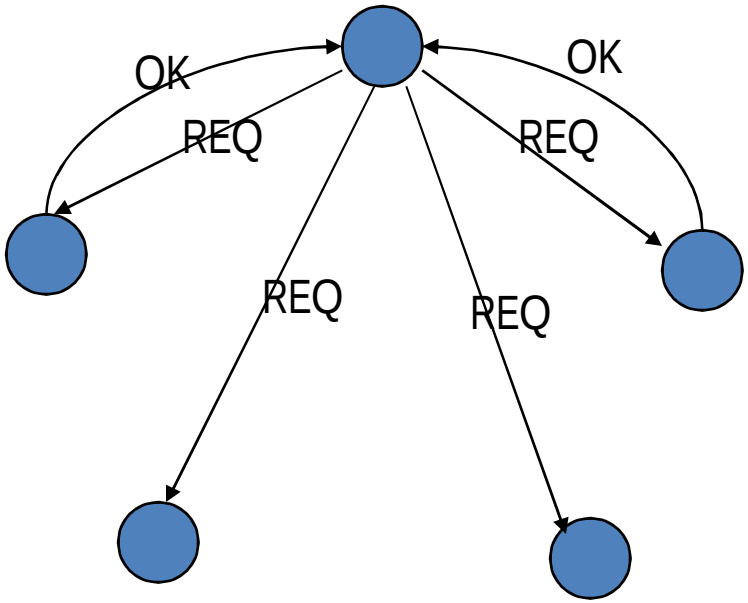
- In single-processor systems, critical regions are protected using semaphores, monitors, and similar constructs.
- In distributed systems, since there is no shared memory, these methods cannot be used.

A Centralized Algorithm

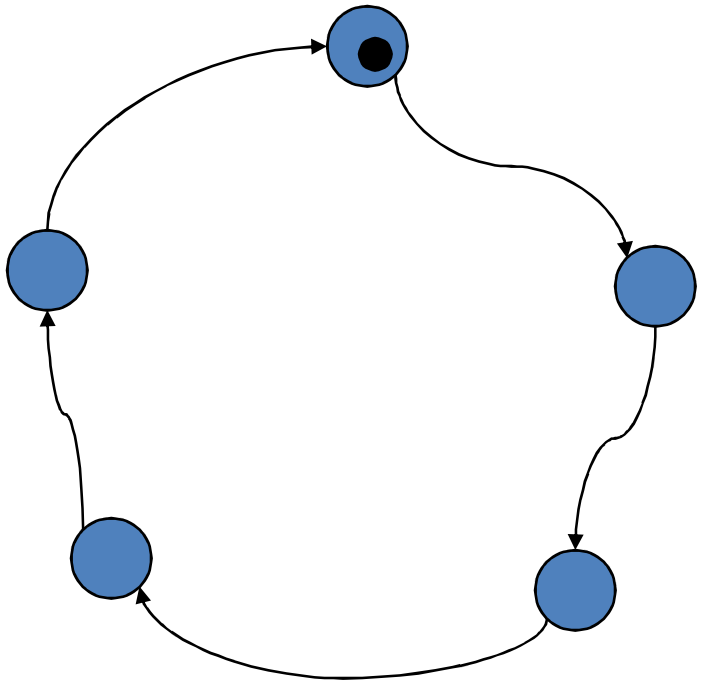


- Advantages: It is fair, easy to implement, and requires only three messages per use of a critical region (request, grant, release).
- Disadvantages: single point of failure.

Distributed Algorithm



Token Ring Algorithm



A Comparison of the Three Algorithms

Algorithm	Messages per entry/exit	Delay before entry	Problems
Centralized	3	2	Coordinator crash
Distributed	2(n-1)	2(n-1)	Crash of any process
Token ring	1 to ∞	0 to n-1	Lost token, process crash

Election Algorithm

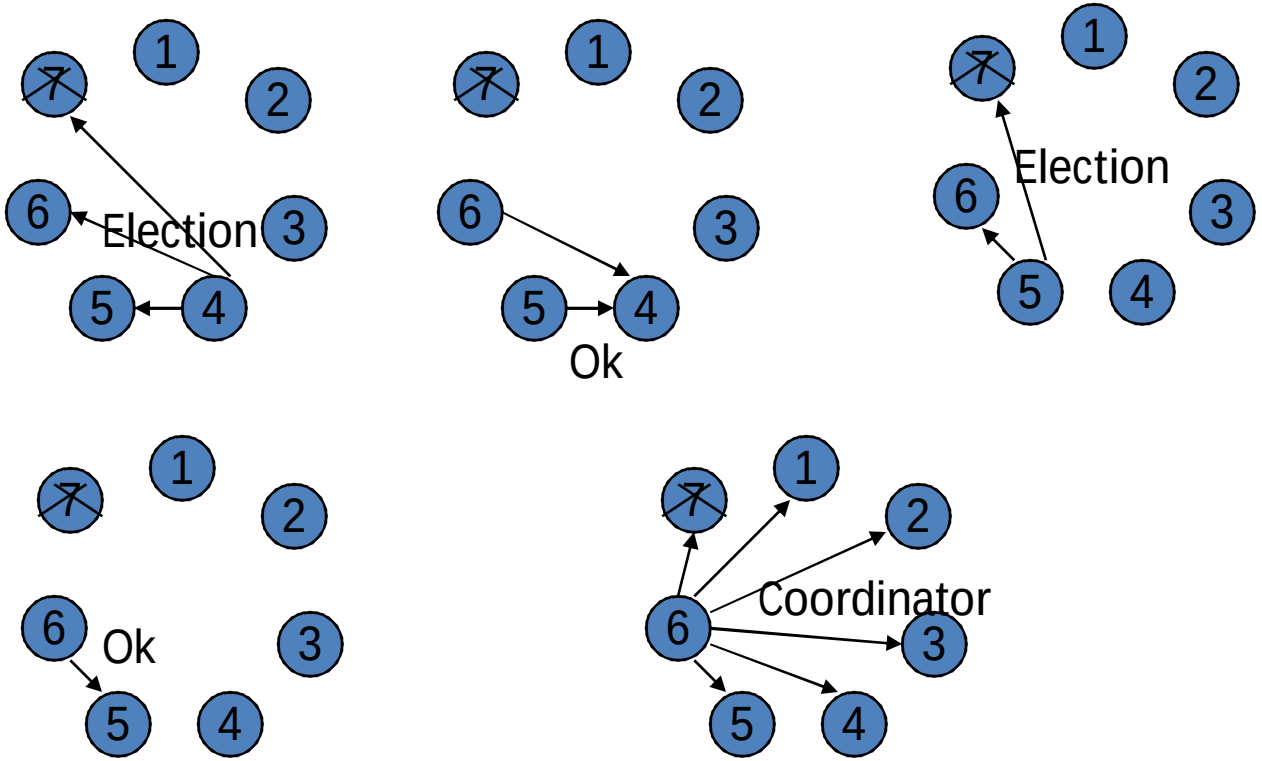
- **The bully algorithm**
- When a process notices that the coordinator is no longer responding to requests, it initiates an election. A process, P, holds an election as follows:

✓ P sends an ELECTION message to all processes with higher numbers.

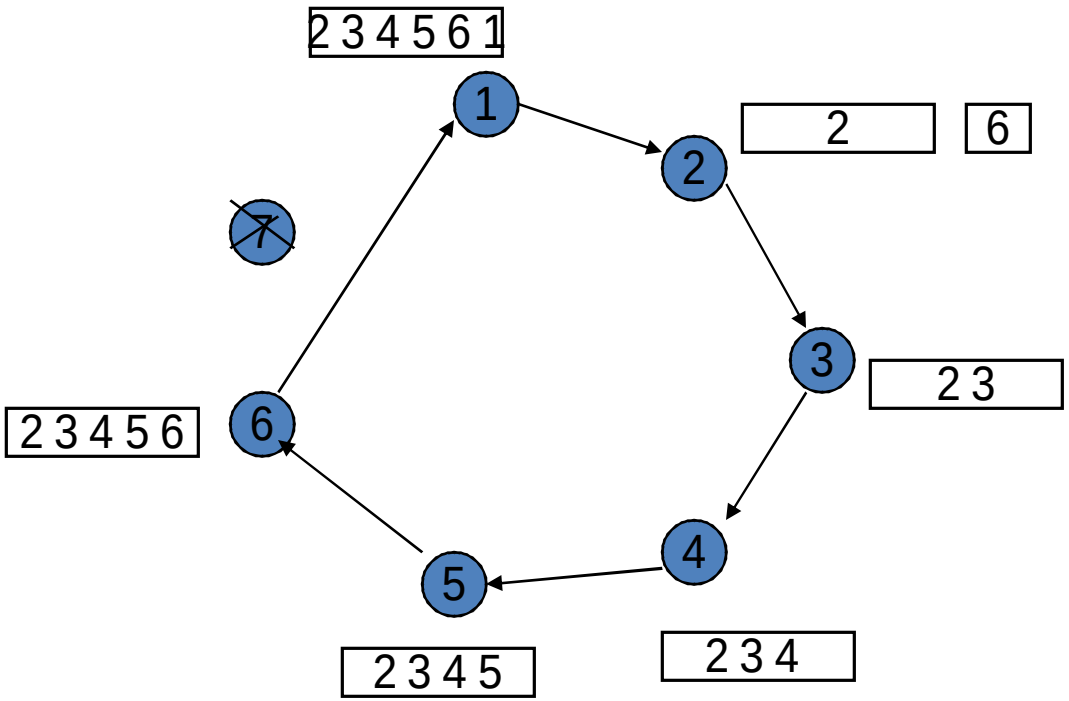
✓ If no one responds, P wins the election and becomes coordinator.

✓ If one of the higher-ups answers, it takes over. P's job is done.

For example


1
7
2
6
Election
3
5
4
1
7
2
6
3
5
4
Ok
1
7
2
Election
6
3
5
4
1
7
2
6
Ok
3
5
4
1
7
2
Coordinator
6
3
5
4

- **A Ring Algorithm**



Atomic Transactions

- All the synchronization techniques we have studied so far are essentially low level, like semaphores.
- What we would really like is a much higher-level abstraction such as **atomic transaction**.

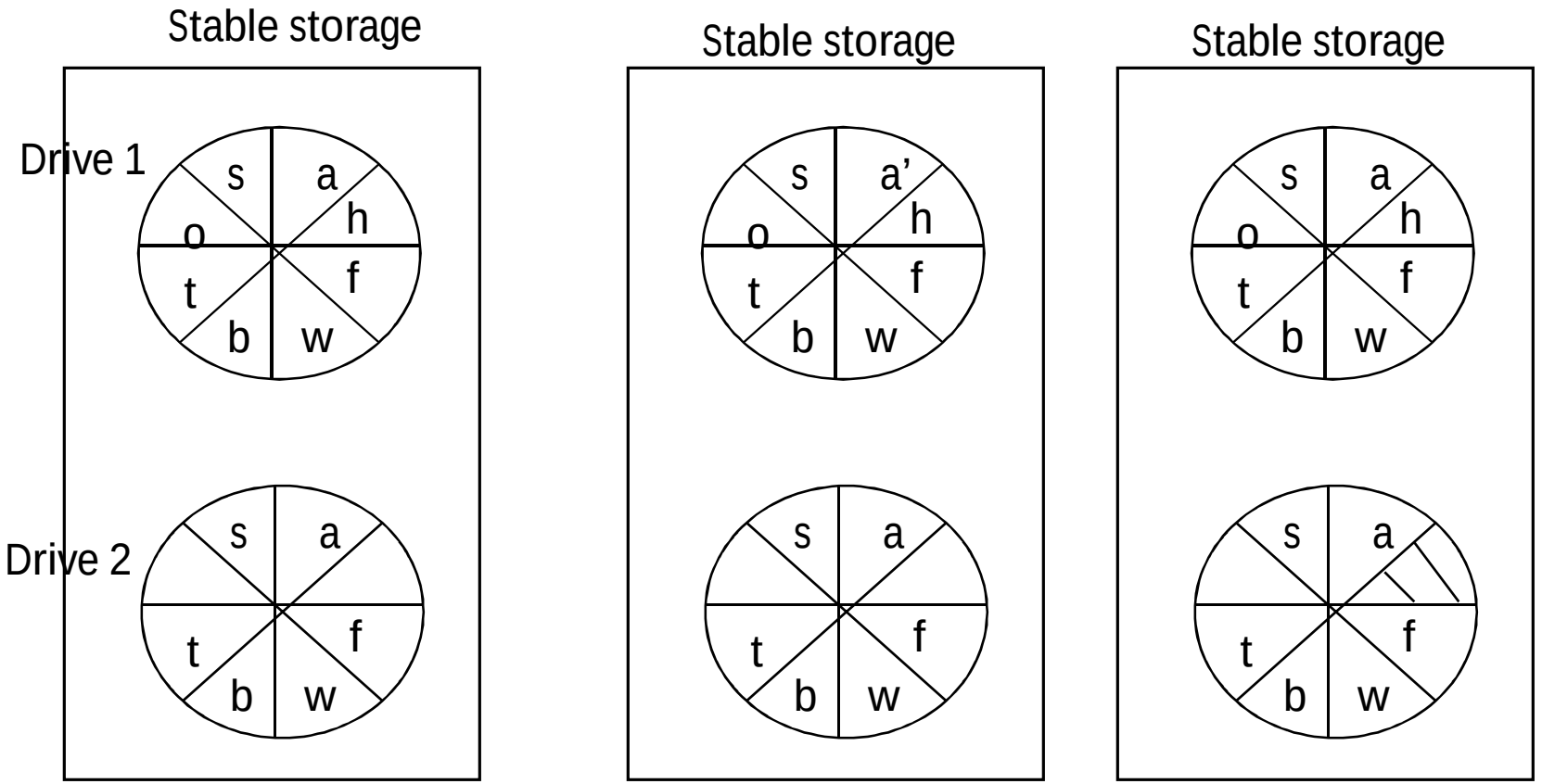
For example

- Atomic bank transactions:

1. Withdraw(amount, account1)
2. Deposit(amount, account2)

Stable Storage

- Stable storage is designed to survive anything except major calamities such as floods and earthquakes.
- Stable storage can be implemented with a pair of ordinary disks.
- Stable storage is well suited to applications that require a high degree of fault tolerance, such as atomic transactions.



(a) Stable storage

(b) Crash after drive 1 is updated

© Bad spot

Transaction Primitives

- 1 BEGIN_TRANSACTION: Mark the start of a transaction.
- 2 END_TRANSACTION: Terminate the transaction and try to commit.
- 3 ABORT_TRANSACTION: Kill the transaction; restore the old values.
- 4 READ: Read data from a file (or other object).
- 5 WRITE: Write data to a file (or other object).

- For example,
- BEGIN_TRANSACTION
- reserve Austin-Houston;
- reserve Houston-Los Angeles;
- reserve Los Angeles-Seatle;
- END_TRANSCATION

Properties of Transactions

- 1 Atomic: To the outside world, the transaction happens indivisibly.
- 2 Consistent: The transaction does not violate system invariants.
- 3 Isolated: Concurrent transactions do not interfere with each other.

  4 Durable: Once a transaction commits, the changes are permanent.

Isolated or serializable

- Isolated or serializable means that if two or more transactions are running at the same time, to each of them and to other processes, the final result looks as though all transactions ran sequentially in some (system dependent) order.

An example

- BEGIN_TRANACATION
- X = 0;
- X=X+1;
- END_TRANSACTION
- (a)
- BEGIN_TRANSACTION
- X=0;
- X= X+2;
- END_TRANSACTION
- (b)
- BEGIN_TRANSACTION
- X=0;
- X=X+3;
- END_TRANSACTION
- (c)

Schedule 1	x=0; x=x+1; x=0; x=x+2; x=0; x=x+3;	legal
Schedule 2	x=0; x=0; x=x+1; x=x+2; x=0; x=x+3;	legal
Schedule 3	x=0; x=0; x=x+1; x=0; x=x+2; x=x+3;	illegal

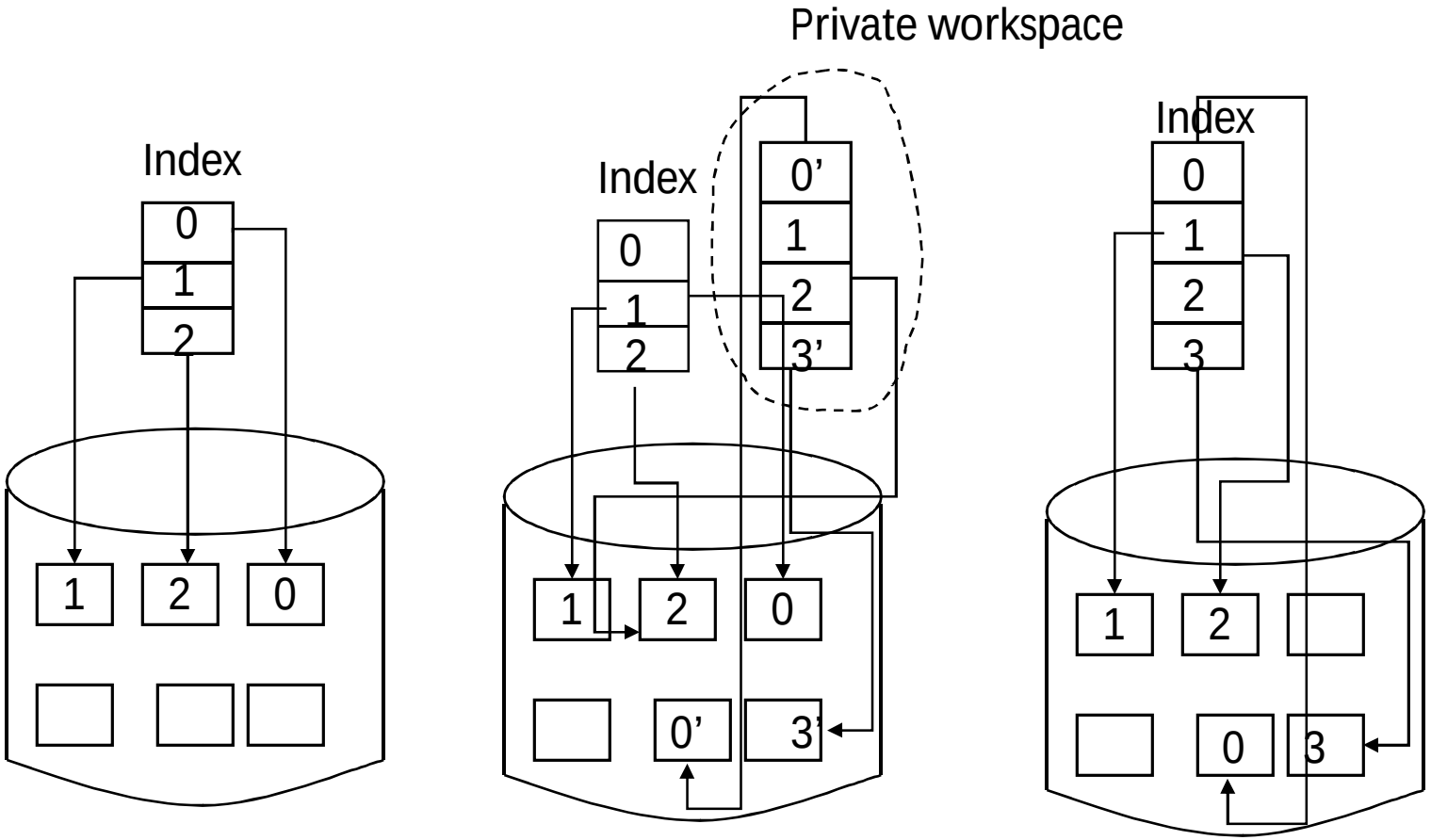
Nest Transactions

- Transactions may contain subtransactions, often called **nested transactions**.

- If the subtransaction commits and the parent transaction aborts, the permanence applies only to top-level transactions.

Implementation

- **Private Workspace**



- **Writeahead log**

x=0;

y=0;

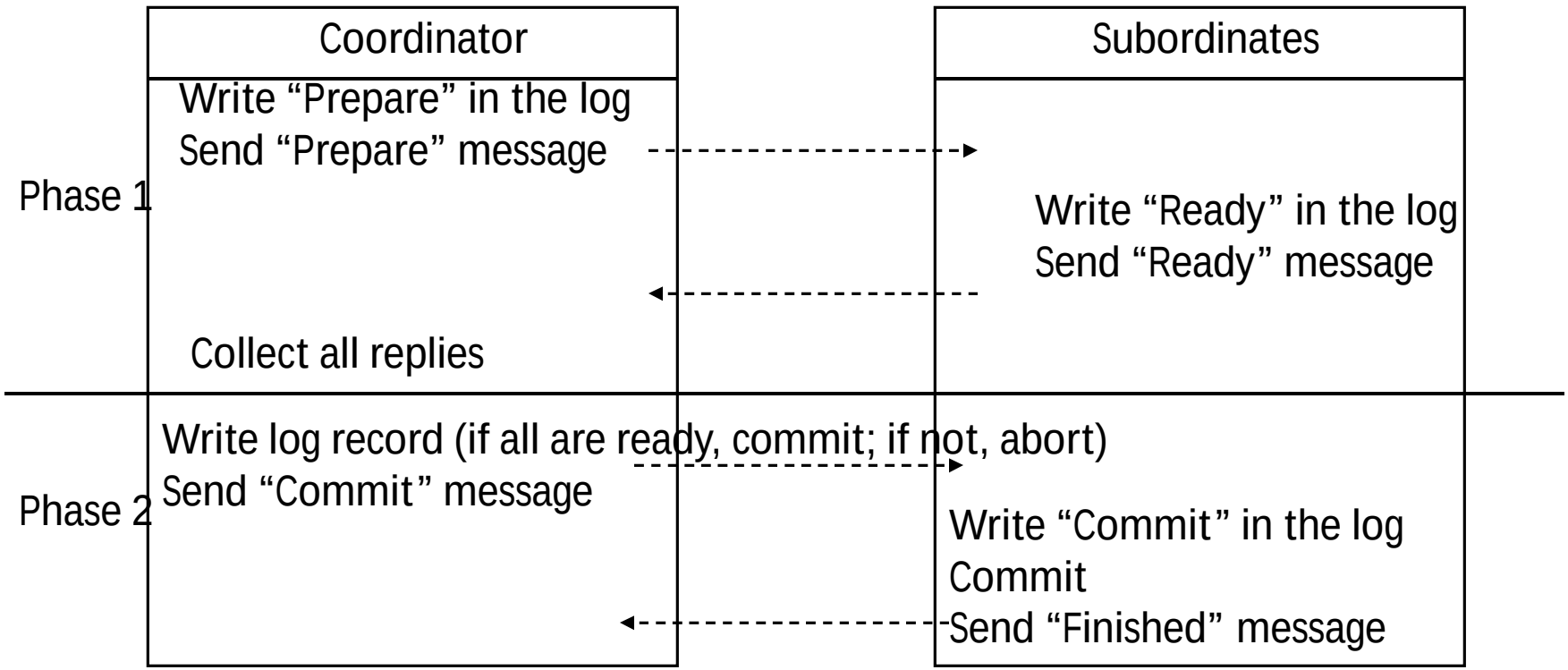
BEGIN_TRANSACTION

x=x+1;	**log: x=0/;**
y=y+2;	**log: x=0/1; y=0/2;**
x=y * y;	**log: x=0/1; y=0/2; x=1/4;**

END_TRANSACTION

Achieving atomic commit in a distributed system

- **Two-Phase Commit Protocol**



Concurrency Control

- When multiple transactions are executing simultaneously in different processes, some mechanism is needed to keep them out of each other's way. That mechanism is called a **concurrency control algorithm**.

Concurrency control algorithms

- **Locking**

✓ In the simplest form, when a process needs to read or write a file (or other object) as part of a transaction, it first locks the file.

✓ Distinguishing read locks from write locks.

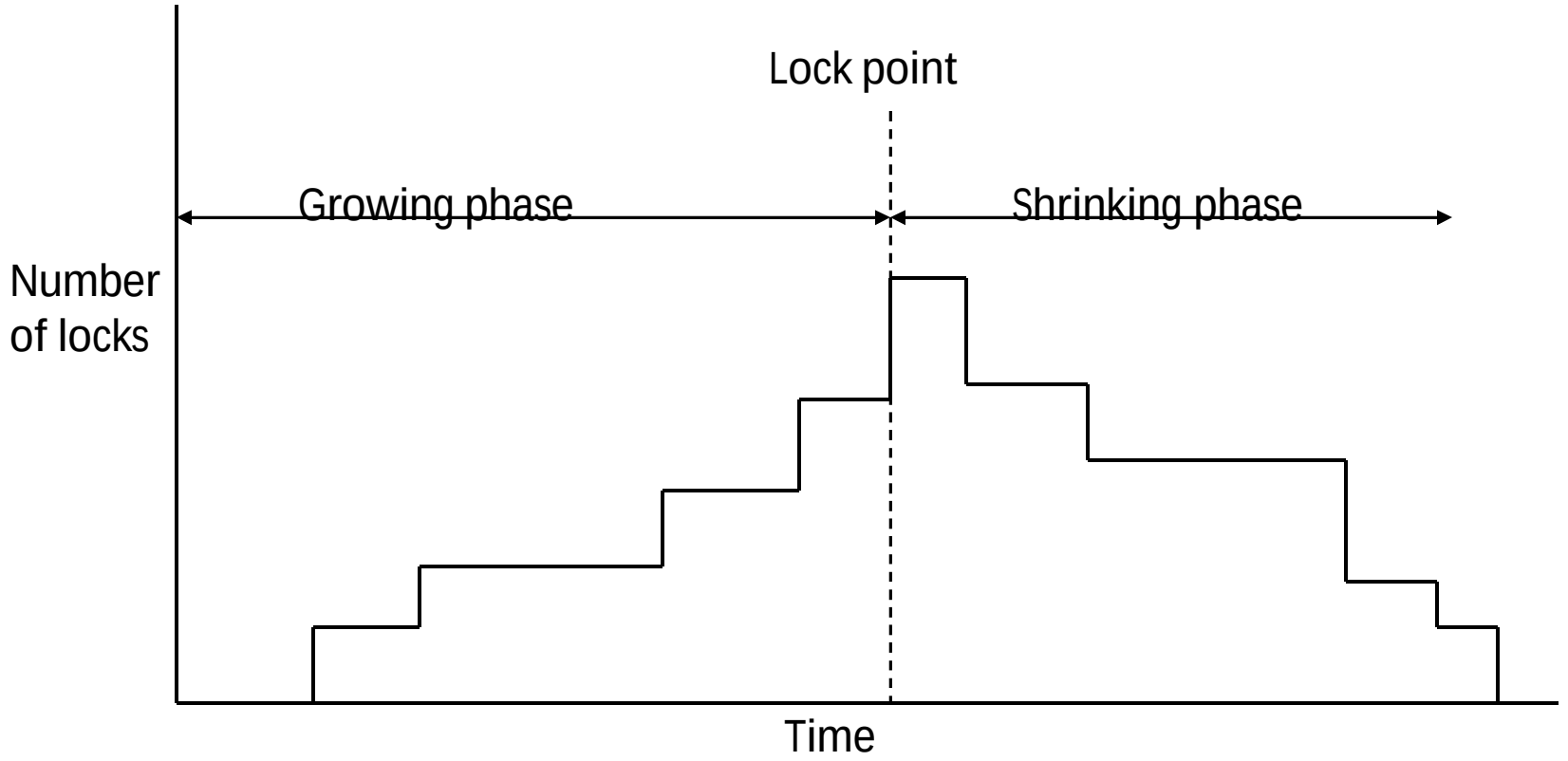
✓ The unit of locking can be an individual record or page, a file, or a larger item.

- **Two-phase locking**

✓ The process first acquires all the locks it needs during the **growing phase**, then releases them during the **shrinking phase**.

✓ In many systems, the shrinking phase does not take place until the transaction has finished running and has either committed or aborted. This policy is called **strict two-phase locking.**

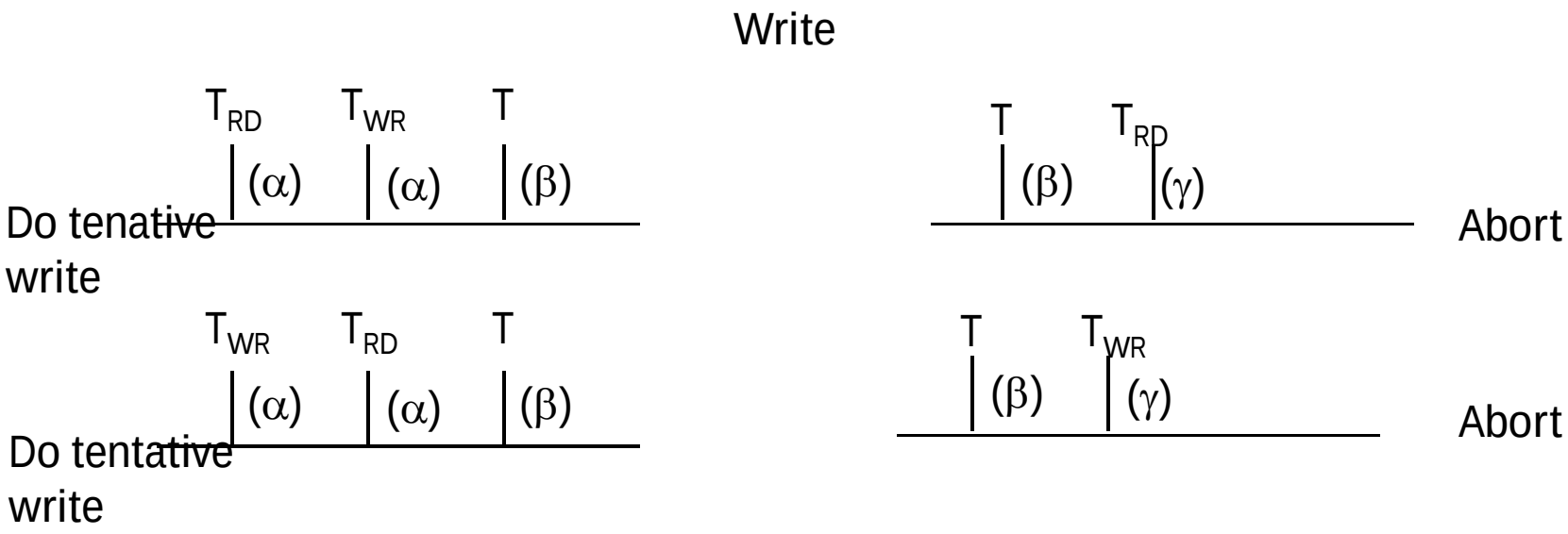
Two-phase locking

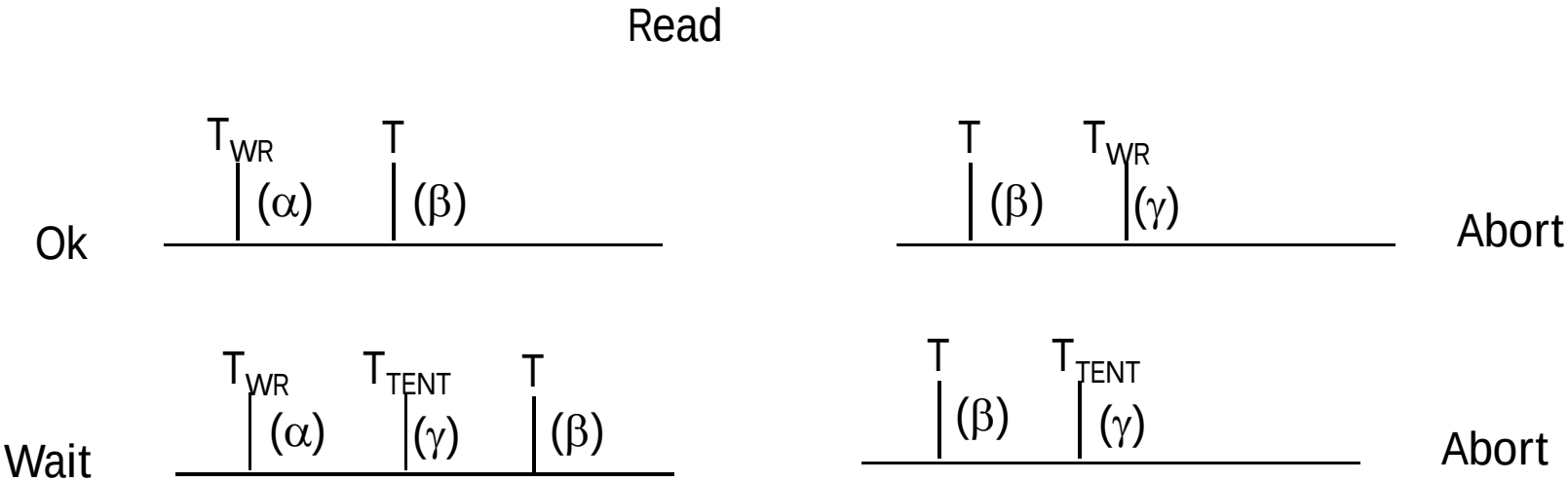


- **Optimistic Concurrency Control**

  A second approach to handling multiple transactions at the same time is **optimistic concurrency control.** The idea is simple: just go ahead and do whatever you want to, without paying attention to what anybody else is doing. If there is a problem, worry about it later.

- **Timestamps**

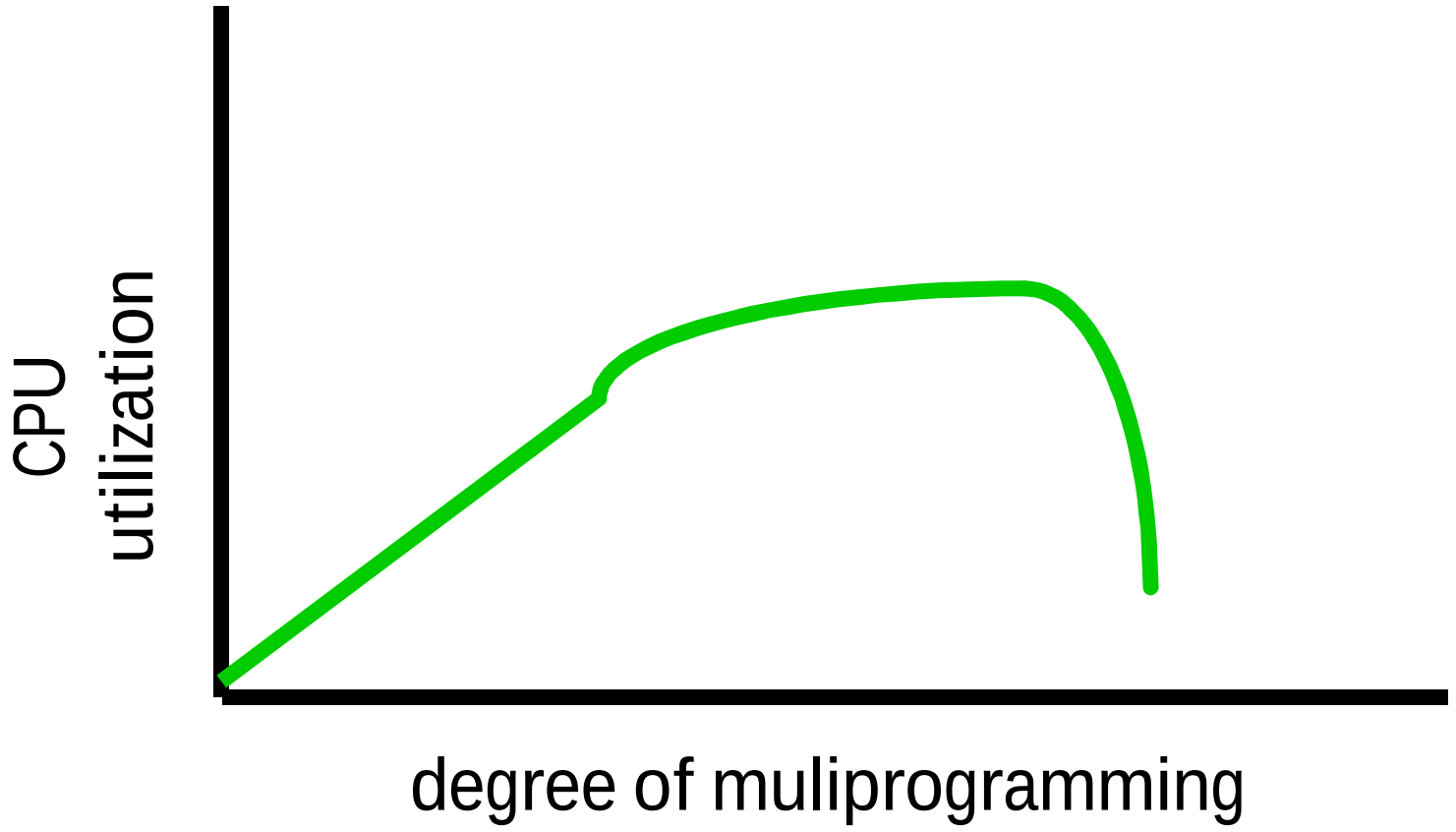

Write
T_{RD}
T_{WR}
T
(α)
(α)
(β)
Do tentative write
T
T_{RD}
(β)
(γ)
Abort
T_{WR}
T_{RD}
T
(α)
(α)
(β)
Do tentative write
T
T_{WR}
(β)
(γ)
Abort


Read
Ok
T_{WR}
(α)
T
(β)
T
(β)
T_{WR}
(γ)
Abort
Wait
T_{WR}
(α)
T_{TENT}
(γ)
T
(β)
T
(β)
T_{TENT}
(γ)
Abort

THRASHING

- If a process does not have “enough” pages, the page-fault rate is very high
  - low CPU utilization
  - OS thinks it needs increased multiprogramming
  - adds another process to system
- *Thrashing* is when a process is busy swapping pages in and out

Thrashing



Cause of Thrashing

- Why does paging work?
  - Locality model
    - process migrates from one locality to another
    - localities may overlap
- Why does thrashing occur?
  - sum of localities > total memory size
- How do we fix thrashing?
  - *Working Set Model*
  - *Page Fault Frequency*

HETEROGENOUS DSM

- The design, implementation, and performance of heterogeneous distributed shared memory (HDSM) are studied. A prototype HDSM system that integrates very different types of hosts has been developed, and a number of applications of this system are reported. Experience shows that despite a number of difficulties in data conversion, HDSM is implementable with minimal loss in functional and performance transparency when compared to homogeneous DSM systems

HETEROGENOUS DSM

- Data conversion - Data converted when a block is migrated between two nodes
- Structuring the DSM system as a collection of source language objects
  - Unit of data migration is object
  - Usually objects are scalar data types, making system very inefficient
- Allowing only one type of data in a block
  - Page size is block size
  - DSM page table keeps additional information identifying type of data maintained in that page.
  - Wastage of memory due to fragmentation
  - As size of application level data structures differs for two machines, mapping between pages on two machines would not be one to one.
  - Entire pages are converted even if small portion required
  - Requires users to provide conversion routines.

RESOURCE MANAGEMENT

- In organizational studies, **resource management** is the efficient and effective development of an organization's **resources** when they are needed. Such **resources** may include financial **resources**, inventory, human skills, production **resources**, or information technology (IT).

RESOURCE MANAGEMENT

- DIVIDED INTO TWO TECHNIQUES
  1. LOAD BALANCING APPROACH
  2. LOAD SHARING APPROACH

Load-balancing approach

Type of dynamic load-balancing algorithms

- Centralized versus Distributed
  - Centralized approach collects information to server node and makes assignment decision
  - Distributed approach contains entities to make decisions on a predefined set of nodes
  - Centralized algorithms can make efficient decisions, have lower fault-tolerance
  - Distributed algorithms avoid the bottleneck of collecting state information and react faster

Load-balancing approach

Type of distributed load-balancing algorithms

- Cooperative versus Noncooperative
  - In Noncooperative algorithms entities act as autonomous ones and make scheduling decisions independently from other entities
  - In Cooperative algorithms distributed entities cooperatewith each other
  - Cooperative algorithms are more complex and involve larger overhead
  - Stability of Cooperative algorithms are better

Load-sharing approach

- Drawbacks of Load-balancing approach
  - Load balancing technique with attempting equalizing the workload on all the nodes is not an appropriate object since big overhead is generated by gathering exact state information
  - Load balancing is not achievable since number of processes in a node is always fluctuating and temporal unbalance among the nodes exists every moment
- Basic ideas for Load-sharing approach
  - It is necessary and sufficient to prevent nodes from being idle while some other nodes have more than two processes
  - Load-sharing is much simpler than load-balancing since it only attempts to ensure that no node is idle when heavily node exists
  - Priority assignment policy and migration limiting policy are the same as that for the load-balancing algorithms

Load estimation policies

for Load-sharing algorithms

- Since load-sharing algorithms simply attempt to avoid idle nodes, it is sufficient to know whether a node is busy or idle
- Thus these algorithms normally employ the simplest load estimation policy of counting the total number of processes
- In modern systems where permanent existence of several processes on an idle node is possible, algorithms measure CPU utilization to estimate the load of a node

Location policies I.

for Load-sharing algorithms

- Location policy decides whether the sender node or the receiver node of the process takes the initiative to search for suitable node in the system, and this policy can be the following:
  - Sender-initiated location policy
    - Sender node decides where to send the process
    - Heavily loaded nodes search for lightly loaded nodes
  - Receiver-initiated location policy
    - Receiver node decides from where to get the process
    - Lightly loaded nodes search for heavily loaded nodes

PROCESS MANAGEMENT

Three important concepts are used to achieve this goal:

-Processor allocation

-Process migration

-Threads

PROCESS MANAGEMENT

Process allocation deals with the process of deciding which process should be assigned to which processor.

Process migration deals with the movement of a process from its current location to the processor to which it has been assigned.

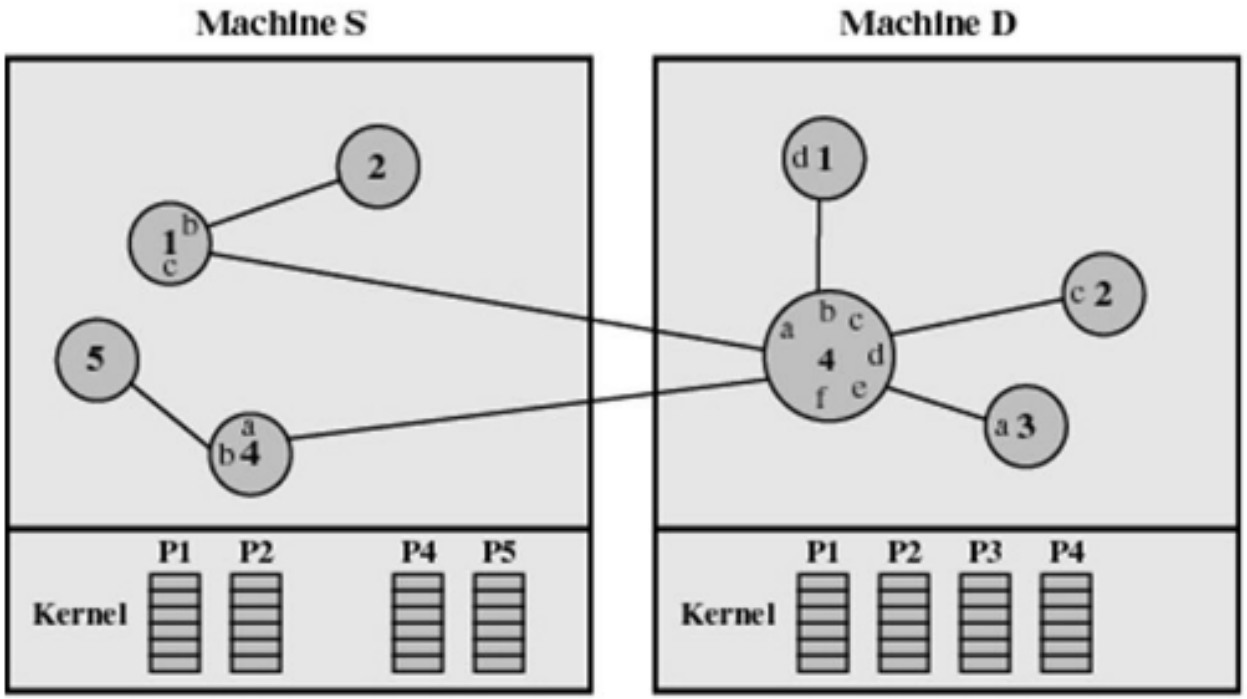
Threads deals with fine-grained parallelism for better utilization of the processing capability of the system.

3

PROCESS MANAGEMENT

- **Process Migration** refers to the mobility of executing (or suspended) **processes** in a **distributed** computing environment. Usually, this term indicates that a **process**uses a network to **migrate** to another machine to continue its execution there.

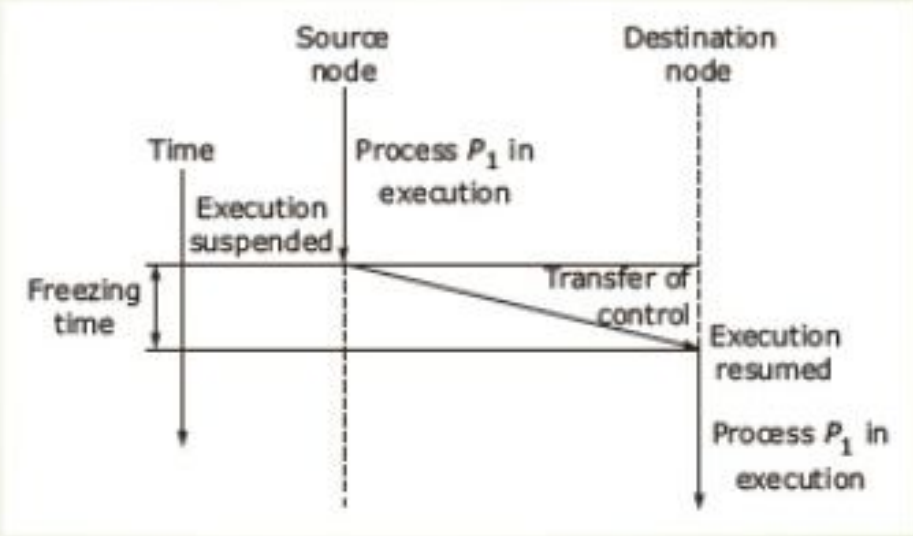
PROCESS MANAGEMENT



(b) After migration

PROCESS MANAGEMENT

Mechanism

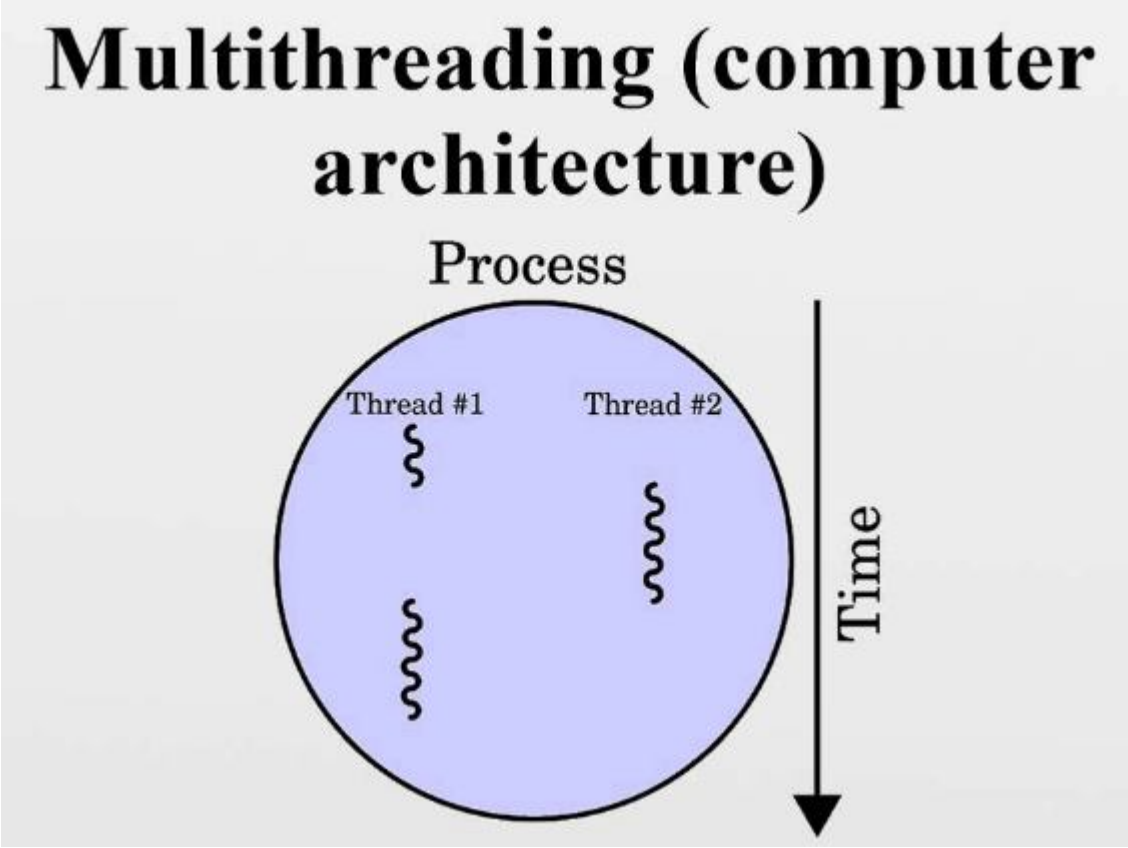


Process migration mechanism

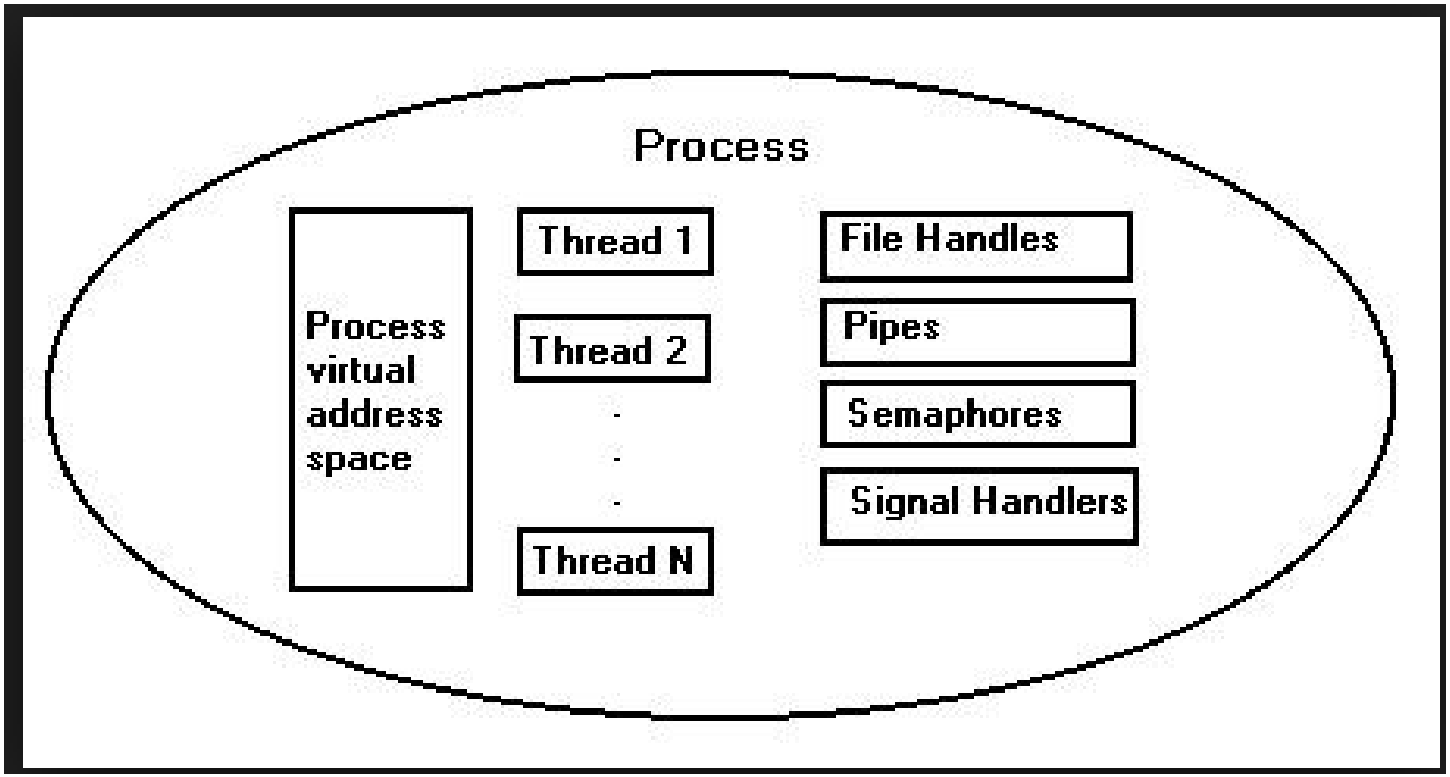
THREAD

- Multithreading. Multithreading is mainly found in multitasking operating **systems**. Multithreading is a widespread programming and execution model that allows multiple **threads** to exist within the context of one process. These **threads** share the process's resources, but are able to execute independently.

THREAD


Multithreading (computer architecture)
Process
Thread #1
Thread #2
Time

THREAD


Process
Process virtual address space
Thread 1
Thread 2
Thread N
File Handles
Pipes
Semaphores
Signal Handlers

THREAD

- *Responsiveness*: multithreading can allow an application to remain responsive to input. In a one-thread program, if the main execution thread blocks on a long-running task, the entire application can appear to freeze. By moving such long-running tasks to a *worker thread* that runs concurrently with the main execution thread, it is possible for the application to remain responsive to user input while executing tasks in the background. On the other hand, in most cases multithreading is not the only way to keep a program responsive, with non-blocking I/O and/or UniX signals being available for gaining similar results

THREAD

- *Faster execution*: this advantage of a multithreaded program allows it to operate faster on computer systems that have multiple central processing units (CPUs) or one or more multi-core processors, or across a clusterof machines, because the threads of the program naturally lend themselves to parallel execution, assuming sufficient independence (that they do not need to wait for each other).

THREAD

- *Parallelization*: applications looking to use multicore or multi-CPU systems can use multithreading to split data and tasks into parallel subtasks and let the underlying architecture manage how the threads run, either concurrently on one core or in parallel on multiple cores. GPU computing environments
like CUDA and OpenCLuse the multithreading model where dozens to hundreds of threads run in parallel across data on a large number of cores.

UNIT -IV

OVERVIEW OF SHARED MEMORY

CONSISTENCY MODEL

- A **memory consistency model** only applies to systems that allow multiple copies of**shared** data; e.g., through caching....

  Other aspects include the order in which a processor issues **memory** operations to the **memory** system, and whether a write executes atomically.

CONSISTENCY MODEL

- In describing the behavior of these memory models, we are only interested in the shared memory behavior - not anything else related to the programs. We aren't interested in control flow within the programs, data manipulations within the programs, or behavior related to local (in the sense of non-shared) variables. There is a stnadard notation for this, which we'll be using in what follows.
- In the notation, there will be a line for each processor in the system, and time proceeds from left to right. Each shared-memory operation performed will appear on the processor's line. The two main operations are Read and Write, which are expressed as
- W(*var*)*value* which means "write *value* to shared variable *var*", and
- R(*var*)*value* which means "read shared variable *var*, obtaining *value*."
- So, for instance, W(x)1 means "write a 1 to x" and R(y)3 means "read *y*, and get the value 3."
- More operations (especially synchronization operations) will be defined as we go on. For simplicity, variables are assumed to be initialized to 0.
- An important thing to notice about this is that a single high-level language statement (like x = x + 1;) will typically appear as several memory operations. If x previously had a value of 0, then that statement becomes (in the absence of any other processors)
- P1: R(x)0 W(x)1 ----------
- On a RISC-style processor, it's likely that C statement would have turned into three instructions: a load, an add, and a store. Of those three instructions, two affect memory and are shown in the diagram.
- On a CISC-style processor, the statement would probably have turned into a single, in-memory add instruction. Even so, the processor would have executed the instruction by reading memory, doing the addition, and then writing memory, so it would still appear as two memory operations.
- Notice that the actual memory operations performed could equally well have been performed by some completely different high level language code; maybe an if-then-else statement that checked and then set a flag. If I ask for memory operations and there is anything in your answer that looks like a transformation or something of the data, then something is wrong!

CONSISTENCY MODEL

- **Strict Consistency**
- The intuitive notion of memory consistency is the *strict consistency* model. In the strict model, **any read to a memory location X returns the value stored by the most recent write operation to X**. If we have a bunch of processors, with no caches, talking to memory through a bus then we will have strict consistency. The point here is the precise serialization of all memory accesses.
- We can give an example of what is, and what is not, strict consistency and also show an example of the notation for operations in the memory system. As we said before, we assume that all variables have a value of 0 before we begin. An example of a scenario that would be valid under the strict consistency model is the following:
- P1: W(x)1 ---------- P2: R(x)1 R(x)1
- This says, ``processor P1 writes a value of 1 to variable x; at some later time processor P2 reads x and obtains a value of 1. Then it reads it again and gets the same value"
- Here's another scenario which would be valid under strict consistency:
- P1: W(x)1 ---------- P2: R(x)0 R(x)1
- This time, P2 got a little ahead of P1; its first read of x got a value of 0, while its second read got the 1 that was written by P1. Notice that these two scenarios could be obtained in two runs of the same program on the same processors.
- Here's a scenario which would not be valid under strict consistency:
- P1: W(x)1 ---------- P2: R(x)0 R(x)1
- In this scenario, the new value of x had not been propagated to P2 yet when it did its first read, but it did reach it eventually.
- I've also seen this model called *atomic consistency*.

CONSISTENCY MODEL

- **Sequential Consistency**
- Sequential consistency is a slightly weaker model than strict consistency. It was defined by Lamport as **the result of any execution is the same as if the reads and writes occurred in some order, and the operations of each individual processor appear in this sequence in the order specified by its program.**
- In essence, any ordering that could have been produced by a strict ordering regardless of processor speeds is valid under sequential consistency. The idea is that by expanding from the sets of reads and writes that *actually* happened to the sets that *could* have happened, we can reason more effectively about the program (since we can ask the far more useful question, "could the program have broken?"). We can reason about the program itself, with less interference from the details of the hardware on which it is running. It's probably fair to say that if we have a computer system that really uses strict consistency, we'll want to reason about it using sequential consistency

CONSISTENCY MODEL

The third scenario above would be valid under sequential consistency. Here's another scenario that would be valid under sequential consistency:

```
P1:  W(x)1
----------------------
P2:       R(x)1 R(x)2
----------------------
P3:       R(x)1 R(x)2
----------------------
P4:  W(x)2
```

This one is valid under sequential consistency because the following alternate interleaving would have been valid under strict consistency:

```
P1:  W(x)1
---------------------------
P2:       R(x)1      R(x)2
---------------------------
P3:       R(x)1      R(x)2
---------------------------
P4:            W(x)2
```

CONSISTENCY MODEL

Here's a scenario that would not be valid under sequential consistency:

```
P1:  W(x)1
----------------------
P2:       R(x)1 R(x)2
----------------------
P3:       R(x)2 R(x)1
----------------------
P4:  W(x)2
```

Oddly enough, the precise definition, as given by Lamport, doesn't even require that ordinary notions of causality be maintained; it's possible to see the result of a write before the write itself takes place, as in:

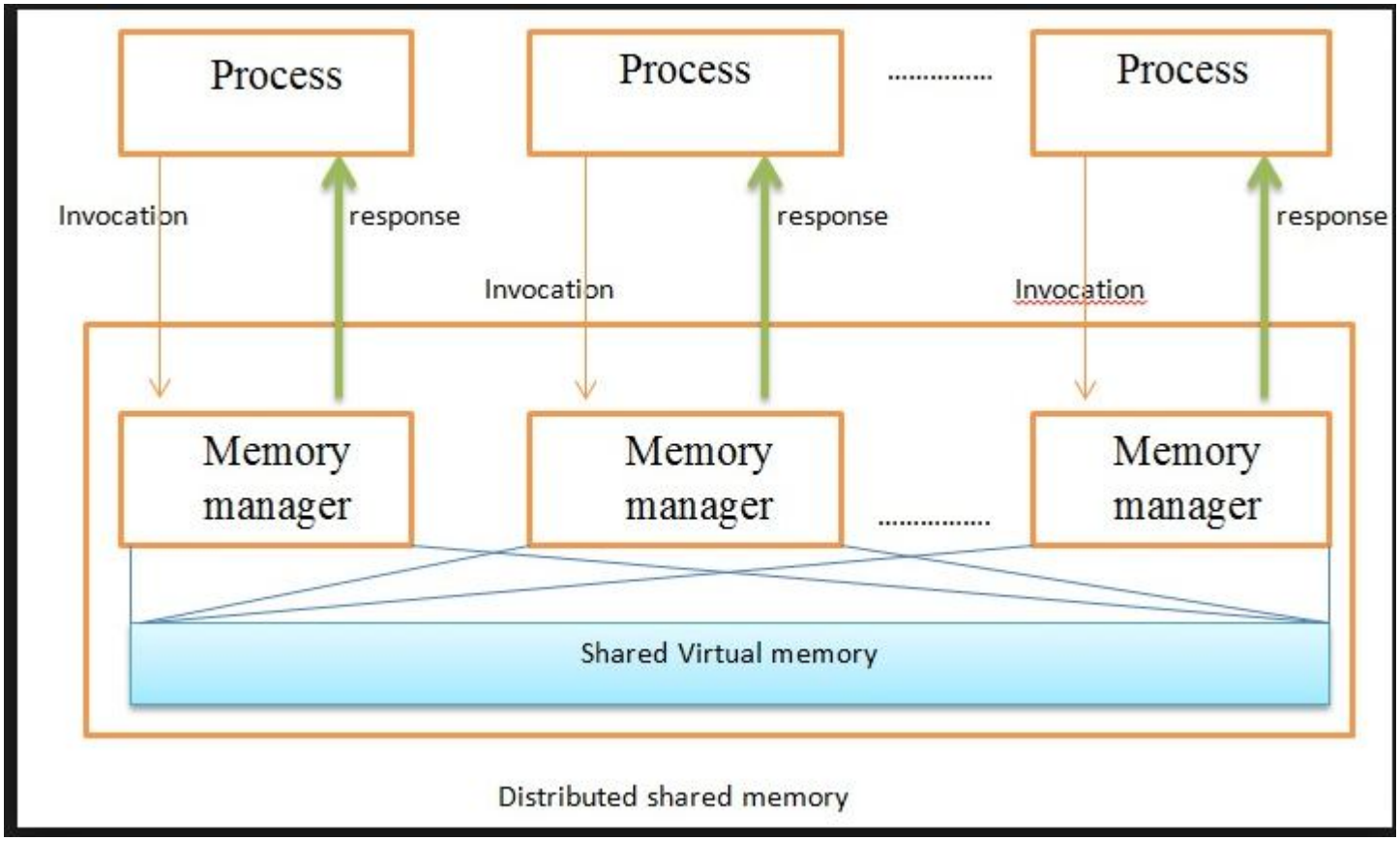
```
P1:       W(x)1
----------------------
P2:  R(x)1
```

This is valid because there is a different ordering which, in strict consistency, would yield P2 reading x as having a value of 1. This isn't a flaw in the model; if your program can indeed violate causality like this, you're missing some synchronization operations in your program. Note that we haven't talked about synchronization operations yet; we will soon.

SHARED MEMORY

- In computer science, **distributed shared memory**(DSM) is a form of **memory** architecture where physically separated memories can be addressed as one logically **shared** address space.

SHARED MEMORY



SHARED MEMORY

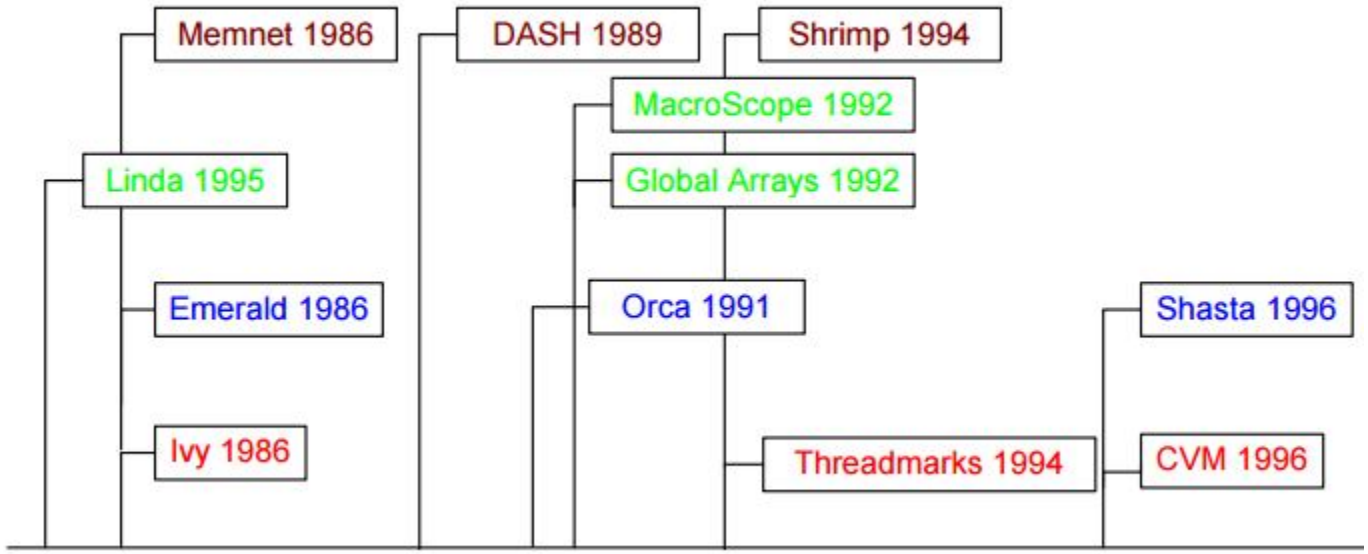
Page based DSM

- **Emulates a standard symmetrical shared memory multi processor**
- **Always hardware supported to some extend**
  - **May use customized hardware**
  - **May rely only on the MMU**
- **Usually independent of compiler, but may require a special compiler for optimal performance**

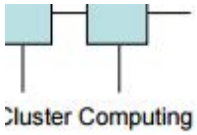
SHARED MEMORY

- **Distribution methods are:**
  - **Migration**
  - **Replication**
- **Examples of Page based DSM systems are:**
  - **Ivy**
  - **Threadmarks**
  - **CVM**
  - **Shrimp-2 SVM**

SHARED MEMORY



SHARED MEMORY



Three example DSM systems

- **Orca**
  Object based language and compiler sensitive system
- **Linda**
  Language independent structured memory DSM system
- **IVY**
  Page based system

SHARED MEMORY

Ivy

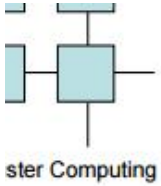
- **The first page based DSM system**
- **No custom hardware used - only depends on MMU support**
- **Placed in the operating system**
- **Supports read replication**
- **Three distribution models supported**
  - **Central server**
  - **Distributed servers**
  - **Dynamic distributed servers**
- **Delivered rather poor performance**

SHARED MEMORY

Ivy

- **Advantages**
  - **No new language introduced**
  - **Fully transparent**
  - **Virtual machine is a perfect emulation of an SMP architecture**
  - **Existing parallel applications runs without porting**
- **Disadvantages**
  - **Exhibits trashing**
  - **Poor performance**

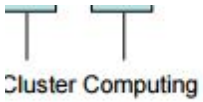
SHARED MEMORY



IVY Status

- **Dead!**
- **New SOA is Shrimp-2 SVM and CVM**
  - **Moved from kernel to user space**
  - **Introduced new relaxed consistency models**
  - **Greatly improved performance**
  - **Utilizing custom hardware at firmware level**

VARIABLE BASED SHARED MEMORY



Variable based DSM

- **Delivers the lowest distribution granularity**
- **Closely integrated in the compiler**
- **May be hardware supported**
- **Possible distribution models are:**
  - **No migration**
  - **Demand migration**
  - **Replication**
- **Variable based DSM systems have never really matured into systems**

VARIABLE BASED SHARED MEMORY

er Computing

Linda

- **Tuple based**
- **Language independent**
- **Targeted at MPP systems but often used in NOW**
- **Structures memory in a tuple space**

```
("Person", "Doe", "John", 23, 82, BLUE)
("pi", 3.141592)
("grades", 96, [Bm, A, Ap, Cp, D, Bp])
```

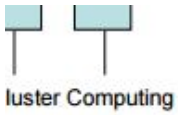
VARIABLE BASED SHARED MEMORY

er Computing

Linda

- **Linda consists of a mere 3 primitives**
  - **out - places a tuple in the tuple space**
  - **in - takes a tuple from the tuple space**
  - **read - reads the value of a tuple but leaves it in the tuple space**
- **No kind of ordering is guarantied, thus no consistency problems occur**

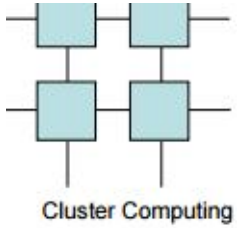
VARIABLE BASED SHARED MEMORY



Linda

- **Advantages**
  - **No new language introduced**
  - **Easy to port trivial producer-consumer applications**
  - **Esthetic design**
  - **No consistency problems**
- **Disadvantages**
  - **Many applications are hard to port**
  - **Fine grained parallelism is not efficient**

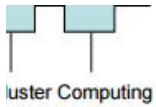
VARIABLE BASED SHARED MEMORY



Linda Status

- **Alive but low activity**
- **Problems with performance**
- **Tuple based DSM improved by PastSet:**
  - **Introduced at kernel level**
  - **Added causal ordering**
  - **Added read replication**
  - **Drastically improved performance**

OBJECT BASED SHARED MEMORY



Object based DSM

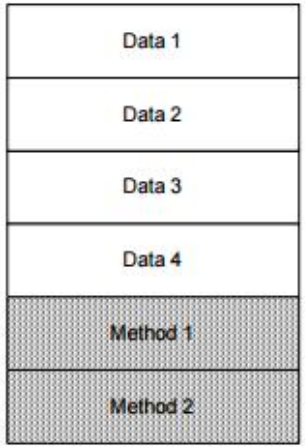
- **Probably the simplest way to implement DSM**
- **Shared data must be encapsulated in an object**
- **Shared data may only be accessed via the methods in the object**
- **Possible distribution models are:**
  - **No migration**
  - **Demand migration**
  - **Replication**
- **Examples of Object based DSM systems are:**
  - **Shasta**
  - **Orca**
  - **Emerald**

OBJECT BASED SHARED MEMORY

luster Computing

Orca

- **Three tier system**
  - **Language**
  - **Compiler**
  - **Runtime system**



- **Closely associated with Amoeba**
- **Not fully object orientated but rather object based**

OBJECT BASED SHARED MEMORY

Orca

- **Claims to be be Modula-2 based but behaves more like Ada**
- **No pointers available**
- **Includes both remote objects and object replication and pseudo migration**
- **Efficiency is highly dependent of a physical broadcast medium - or well implemented multicast.**

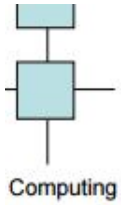
OBJECT BASED SHARED MEMORY

er Computing

Orca

- **Advantages**
  - **Integrated operating system, compiler and runtime environment ensures stability**
  - **Extra semantics can be extracted to achieve speed**
- **Disadvantages**
  - **Integrated operating system, compiler and runtime environment makes the system less accessible**
  - **Existing application may prove difficult to port**

OBJECT BASED SHARED MEMORY



Orca Status

- **Alive and well**
- **Moved from Amoeba to BSD**
- **Moved from pure software to utilize custom firmware**
- **Many applications ported**

UNIT -V

FILE MODELS

FILE ACCESS

Accessing remote files

- One of the following model is used when request to access remote file
  - Remote service model
  - Data catching model

FILE ACCESS

Remote service model

- Processing of client request is performed at server's node
- Client request is delivered to server and server machine performs on it and returns replies to client
- Request and replies transferred across network as message
- File server interface and communication protocol must be designed carefully so as to minimize the overhead of generating the messages
- Every remote file access results in traffic

FILE ACCESS

Data catching model

- Reduced the amount of network traffic by taking advantage of locality feature
- If requested data is not present locally then copied it from server's node to client node and catching there
- LRU is used to keep the cache size bounded
- Cache Consistency problem

FILE ACCESS

Unit of data transfer

- Refers to fraction of file data that is transferred to and from client as a result of single read write operation
- Four data transfer models
  - File level transfer model
  - Block level transfer model
  - Byte level transfer model
  - Record level transfer model

FILE SHARING

File level transfer model

- When the operation required file data, the whole file is moved
- Advantages are
  - Efficient because network protocol overhead is required only once
  - Better scalability because it requires fewer access to file server and reduce server load and network traffic
  - Disk access routines on server can be better optimized
  - Offers degree of resiliency to server and network failure
- Drawbacks is it requires sufficient storage space
- Ex are amoeba, CFS, Andrew file system

FILE SHARING

Block level transfer model

- file data transfer take place in units of data blocks
- A file block is contiguous portion of file and fixed in length
- Advantage is does not required large storage space
- Drawback is more network traffic and more network protocol overhead
- Poor performance
- Ex are Sun microsystem's NFS, Apollo domain file system

FILE SHARING

Byte level transfer model

- File data transfer take place in units of bytes
- Provides maximum flexibility
- Difficulty in cache management

FILE SHARING

Record level transfer model

- Suitable with structured files
- File data transfer take place in unit of records
- Ex. RSS(research storage system)

FILE CACHING

- Caching is based on the idea of replicating frequently accessed data items in lower latency storage units:
  - performance is the main goal here…
  - …but caches can also provide better availabilty, resource utilization…

- Caches are widely employed at a wide variety of levels:

Hardware:

–(Multi) Processor caches:

- SRAM vs DRAM
- Local SRAM vs Remote SRAM/DRAM

–Disk Caches:

- RAM vs Disk

Software:

–(Distributed) File Systems:

- Main Memory vs Disk
- Local FS vs Remote FS

–World Wide Web:

- browser vs forward/reverse proxy vs Web Server RAM vs Web Server Disk

–(Distributed) Database Systems:

- RAM vs Disk
- Local DBMS vs Remote DBMS

FILE CACHING

Distributed Cache Systems

- Locality principles still drive the design process, just like in the non distributed case…

- …but now distribution raises a number of additional issues…
  - high/unpredictable communication latency:
    - we don't want highly/unpredictable inconsistent caches!
  - possibility for cooperation among caches, but…
  - …need for system autonomy despite single caches failures
  - mutual effect of mutliple caching tiers on actual miss rates:
    - trickle-down effect
  - trackability:
    - how many hits on my web page?
  - security, just like in any distributed data replication scheme

FILE CACHING

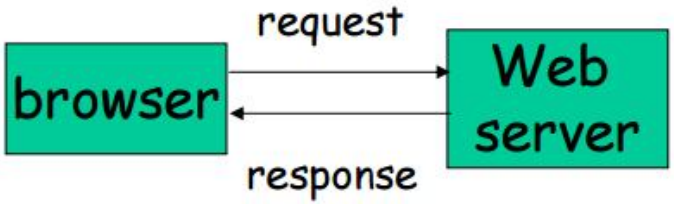
- The consistency requirements play a fundamental role in the design of a distributed cache management scheme…
- Discussing the manifold formal consistency models presented in literature is out of the scope of this course - you're already seeing them in the Distributed Systems course.
- We'll rather take a pragmatical approach and analyze two case studies representative of weak and strong consistency constraints:
  - WWW Caching
  - Transactional Caching

FILE CACHING

Web Caching

Simplest model:

- clients are read-only, only server updates data
- content staleness is tolerated



FILE CACHING

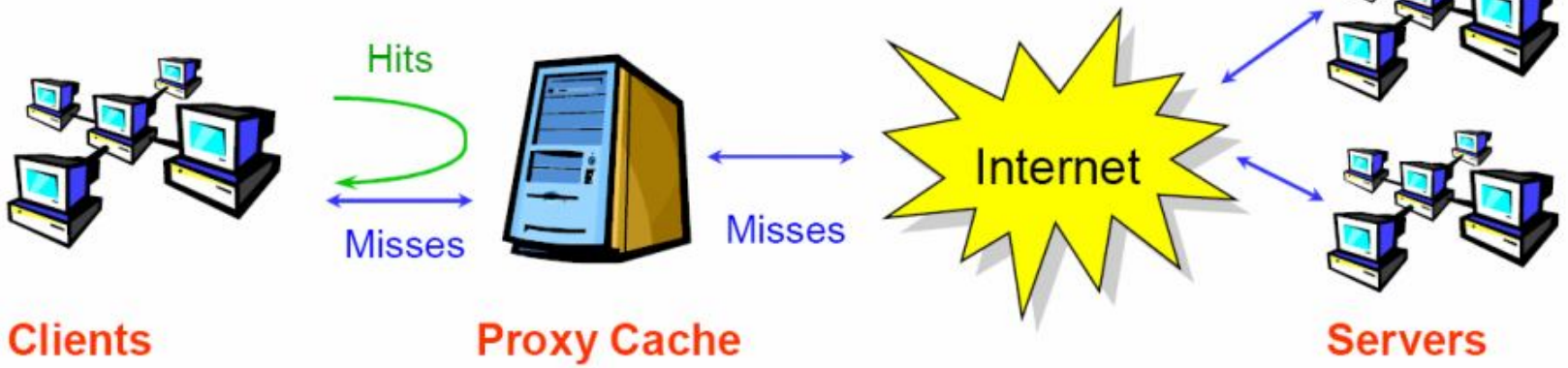
Why Web Caching?

- Cost
  - Original motivation for adopting caches (esp. internationally)
  - Caching saves bandwidth (bandwidth is expensive)
  - 50% byte hit rate cuts bandwidth costs in half
- Performance
  - User: Reduces latency
    - » RTT to cache lower than to server
  - Server: Reduces load
    - » Caches filter requests to server
  - Network: Reduces load
    - » Requests that hit in the cache do not travel all the way to server

FILE CACHING

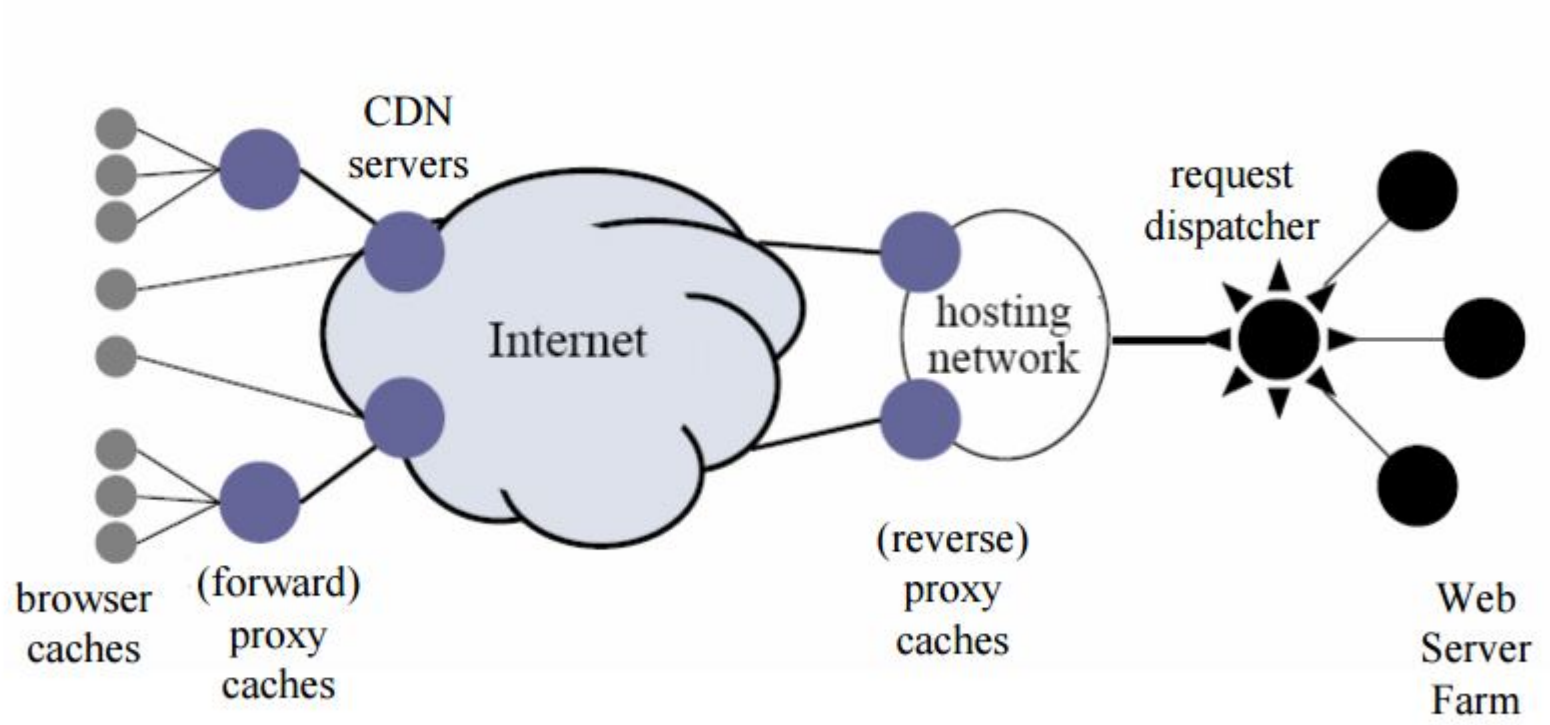
Proxy Caching

- Proxy caching is one of the most common methods used to improve Web performance
  - Duplicate requests to the same document served from cache
  - Hits reduce latency, b/w, network utilization, server load
  - Misses increase latency (extra hops)



FILE CACHING

Where to cache?



FILE CACHING

Cache Misses

- There are a number of reasons why requests miss:
  - Compulsory (50%)
    - Object uncacheable (20%)
    - First access to an object (30%)
  - Capacity (<5%)
    - Finite resources (objects evicted, then referenced again)
  - Consistency (10%)
    - Objects change ("../today") or die (deleted)

FILE REPLICATION

- There are three basic replication models the
- master-slave
- client-server
- peer-to-peer models.

FILE REPLICATION

- **Master-slave model**

  In this model one of the copy is the master replica and all the other copies are slaves. The slaves should always be identical to the master. In this model the functionality of the slaves are very limited, thus the configuration is very simple. The slaves essentially are read-only. Most of the master-slaves services ignore all the updates or modifications performed at the slave, and “undo” the update during synchronization, making the slave identical to the master [3]. The modifications or the updates can be reliably performed at the master and the slaves must synchronize directly with the master.

FILE REPLICATION

2.3.2 Client-server model

The client-server model like the master-slave designates one server, which serves multiple clients. The functionality of the clients in this model is more complex than that of the slave in the master-slave model. It allows multiple inter-communicating servers, all types of data modifications and updates can be generated at the client. One of the replication systems in which this model is successfully implemented is Coda. Coda is a distributed file system with its origin in AFS2. It has many features that are very desirable for network file systems [9]. Optimistic replication can use a client-server model. In Client- server replication all the updates must be propagated first to the server, which then updates all the other clients. In the client-server model, one replica of the data is designated as the special server replica. All updates created at other replicas must be registered with the server before they can be propagated further. This approach simplifies replication system and limits cost, but partially imposes a bottleneck at the server [11]. Since all updates must go through the server, the server acts as a physical synchronization point [13]. In this model the conflicts which occur are always be detected only at the server and only the server needs to handle them. However, if the single server machine fails or is unavailable, no updates can be propagated to other replicas. This leads to un-consistency as individual machines can accept their local updates, but they cannot learn of the updates applied at other machines.

In a mobile environment where connectivity is limited and changing, the server may be difficult or impossible to contact, while other client replicas are simple and cheap to contact. The peer model of optimistic replication can work better in these conditions [13].

FILE REPLICATION

- Peer-to-peer model The Peer-to-peer model is very different from both the master-slave and the client-server models. Here all the replicas or the copies are of equal importance or they are all peers. In this model any replica can synchronize with any other replica, and any file system modification or update can be applied at any replica. Optimistic replication can use a peer-to-peer model. Peer-to-peer systems allow any replica to propagate updates to any other replicas [11]. The peer-to-peer model has been implemented in Locus, Rumor and in other distributed environments such as xFS in the NOW project. Peer-to-peer systems can propagate updates faster by making use of any available connectivity. They provide a very rich and robust communication framework. But they are more complex in implementation and in the states they can achieve [11]. One more problem with this model is scalability. Peer models are implemented by storing all necessary replication knowledge at every site thus each replica has full knowledge about everyone else. As synchronization and communication is allowed between any replicas, this results in exceedingly large replicated data structures and clearly does not scale well. Additionally, distributed algorithms that determine global state must, by definition, communicate with or hear about (via gossiping) each replica at least once and often twice. Since all replicas are peers, any single machine could potentially affect the outcome of such distributed algorithms; therefore each must participate before the algorithm can complete, again leading to potential scaling problems [3]. Simulation studies in the file system arena have demonstrated that the peer model increases the speed of update propagation among a set of replicas, decreasing the frequency of using an outdated version of the data

FAULT TOLERANCE

- **Fault tolerance** is the property that enables a **system** to continue operating properly in the event of the failure of (or one or more **faults** within) some of its components.

FAULT TOLERANCE

- Ability of a system to continue functioning in the event of a partial failure.
- Though the system continues to function but overall performance may get affected.
- Distributed systems are made up of a large number of components, developing a system which is hundred percent fault tolerant is practically very challenging.
- Two main reasons for the occurrence of a fault

  1)Node failure -Hardware or software failure.

  2)Malicious Error-Caused by unauthorized Access.

FAULT TOLERANCE

4.Fault Tolerance Techniques

Replication

- Creating multiple copies or replica of data items and storing them at different sites
- Main idea is to increase the availability so that if a node fails at one site, so data can be accessed from a different site.
- Has its limitation too such as data consistency and degree of replica.

Check Pointing

- Saving the state of a system when they are in a consistent state and storing it on a stable storage.
- Each such instance when a system is in the stable state is called a check point.
- In case of a failure, system is restored to its previous consistent state.
- Saves useful computation.

FAULT TOLERANCE

- Fault Tolerance is needed in order to provide 3 main feature to distributed systems.

  1)Reliability-Focuses on a continuous service with out any interruptions.

  2)Availability - Concerned with read readiness of the system.

  3)Security-Prevents any unauthorized access.

- examples-Patient Monitoring systems, flight control systems, Banking Services etc.

NETWORK FILE SHARING

- A *Network File System* (*NFS*) allows remote hosts to mount file systems over a network and interact with those file systems as though they are mounted locally. This enables system administrators to consolidate resources onto centralized servers on the network.
- This chapter focuses on fundamental NFS concepts and supplemental information. For specific instructions regarding the configuration and operation of NFS server and client software, refer to the chapter titled ***Network File System (NFS)*** in the ***Red Hat Enterprise Linux System Administration Guide***.

NETWORK FILE SYSTEM

- **Required Services**
- Red Hat Enterprise Linux uses a combination of kernel-level support and daemon processes to provide NFS file sharing. NFS relies on *Remote Procedure Calls* (*RPC*) to route requests between clients and servers . RPC services under Linux are controlled by the portmap service. To share or mount NFS file systems, the following services work together:
- nfs — Starts the appropriate RPC processes to service requests for shared NFS file systems.
- nfslock — An optional service that starts the appropriate RPC processes to allow NFS clients to lock files on the server.
- portmap — The RPC service for Linux; it responds to requests for RPC services and sets up connections to the requested RPC service.
- The following RPC processes work together behind the scenes to facilitate NFS services:
- rpc.mountd — This process receives mount requests from NFS clients and verifies the requested file system is currently exported. This process is started automatically by the nfs service and does not require user configuration.
- rpc.nfsd — This process is the NFS server. It works with the Linux kernel to meet the dynamic demands of NFS clients, such as providing server threads each time an NFS client connects. This process corresponds to the nfs service.
- rpc.lockd — An optional process that allows NFS clients to lock files on the server. This process corresponds to the nfslock service.
- rpc.statd — This process implements the *Network Status Monitor (NSM)* RPC protocol which notifies NFS clients when an NFS server is restarted without being gracefully brought down. This process is started automatically by the nfslock service and does not require user configuration.
- rpc.rquotad — This process provides user quota information for remote users. This process is started automatically by the nfs service and does not require user configuration.

NETWORK FILE SYSTEM

- **Troubleshooting NFS and portmap**
- Because portmap provides coordination between RPC services and the port numbers used to communicate with them, it is useful to view the status of current RPC services usingportmap when troubleshooting.
  The rpcinfo command shows each RPC-based service with port numbers, an RPC program number, a version and an IP protocol type (TCP or UDP).