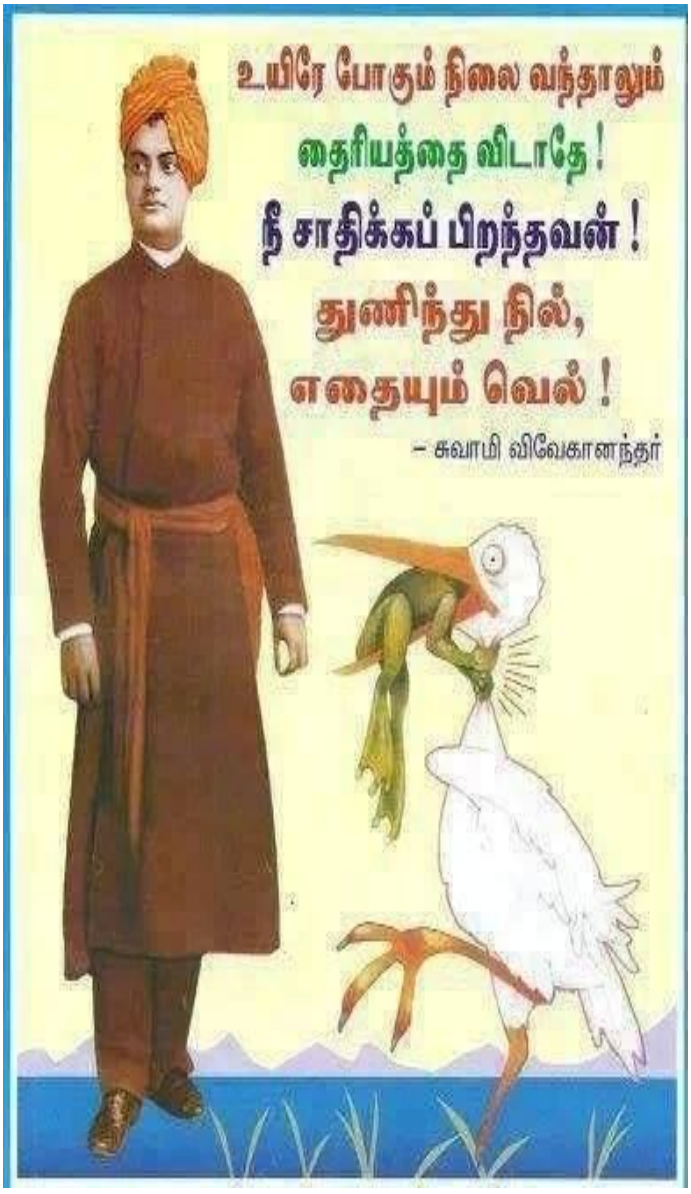
ADVANCED OPERATING SYSTEMS

(CP7204)



Name :

Reg. No :

Depart :

UNIT 1

Overview and History

What is an operating system? Hard to define precisely, because operating systems arose historically as people needed to solve problems associated with using computers.

Much of operating system history driven by relative cost factors of hardware and people. Hardware started out fantastically expensive relative to people and the relative cost has been decreasing ever since. Relative costs drive the goals of the operating system.

- In the beginning: **Expensive Hardware, Cheap People** Goal: maximize hardware utilization.
- Now: **Cheap Hardware, Expensive People** Goal: make it easy for people to use computer.

What does a modern operating system do?

- **Provides Abstractions** Hardware has low-level physical resources with complicated, idiosyncratic interfaces. OS provides abstractions that present clean interfaces. Goal: make computer easier to use. Examples: Processes, Unbounded Memory, Files, Synchronization and Communication Mechanisms.
- **Provides Standard Interface** Goal: portability. Unix runs on many very different computer systems. To a first approximation can port programs across systems with little effort.
- **Mediates Resource Usage** Goal: allow multiple users to share resources fairly, efficiently, safely and securely. Examples:
    - Multiple processes share one processor. (preemptable resource)
    - Multiple programs share one physical memory (preemptable resource).
    - Multiple users and files share one disk. (non-preemptable resource)
    - Multiple programs share a given amount of disk and network bandwidth (preemptable resource).
- **Consumes Resources** Solaris takes up about 8Mbytes physical memory (or about $400).

Processes and Threads

A process is an execution stream in the context of a particular process state.

- An execution stream is a sequence of instructions.

- Process state determines the effect of the instructions. It usually includes (but is not restricted to):
    - Registers
    - Stack
    - Memory (global variables and dynamically allocated memory)
    - Open file tables
    - Signal management information

  Key concept: processes are separated: no process can directly affect the state of another process.

Process is a key OS abstraction that users see - the environment you interact with when you use a computer is built up out of processes.

- The shell you type stuff into is a process.
- When you execute a program you have just compiled, the OS generates a process to run the program.
- Your WWW browser is a process.

Two concepts: uniprogramming and multiprogramming.

- Uniprogramming: only one process at a time. Typical example: DOS. Problem: users often wish to perform more than one activity at a time (load a remote file while editing a program, for example), and uniprogramming does not allow this. So DOS and other uniprogrammed systems put in things like memory-resident programs that invoked asynchronously, but still have separation problems. One key problem with DOS is that there is no memory protection - one program may write the memory of another program, causing weird bugs.
- Multiprogramming: multiple processes at a time. Typical of Unix plus all currently envisioned new operating systems. Allows system to separate out activities cleanly.

Thread Creation, Manipulation and Synchronization

We first must postulate a thread creation and manipulation interface. Will use the one in Nachos:

```
class Thread {
 public:
```

```
    Thread(char* debugName);
    ~Thread();
    void Fork(void (*func)(int), int arg);
    void Yield();
    void Finish();
}
```

The Thread constructor creates a new thread. It allocates a data structure with space for the TCB.

To actually start the thread running, must tell it what function to start running when it runs. The Fork method gives it the function and a parameter to the function.

Let's do a few thread examples. First example: two threads that increment a variable.

```
int a = 0;
void sum(int p) {
  a++;
  printf("%d : a = %d\n", p, a);
}
void main() {
  Thread *t = new Thread("child");
  t->Fork(sum, 1);
  sum(0);
}
```

Semaphore concept

- Integer variable, with initial non-negative value
- Two atomic operations
  - wait
  - signal (not the UNIX signal() call…)
  - p & v (*proberen* & *verhogen*), up and down, etc
- wait
  - wait for semaphore to become positive, then decrement by 1
- signal
  - increment semaphore by 1

Semaphore without busy-wait

```
struct sem {
        value: int;
        L: list of processes
} S;
Init_Sem(S, Count)
{
        S.value = Count;
}
```

- OS support
  - block
  - wakeup
  - P and V atomic

```
P(S)
{S.value = S.value -1;
 if (S.value < 0) {
        add T to S.L;
        block;
 }
V(S)
{S.value = S.value + 1;
 if (S.value <=0) {
        select a T from S.L;
        wakeup(T);
 }
```

Implementing Synchronization Operations

 How do we implement synchronization operations like locks? Can build synchronization operations out of atomic reads and writes. There is a lot of literature on how to do this, one algorithm is called the bakery algorithm. But, this is slow and cumbersome to use. So, most machines have hardware support for synchronization - they provide synchronization instructions.

 On a uniprocessor, the only thing that will make multiple instruction sequences not atomic is interrupts. So, if want to do a critical section, turn off interrupts before the critical section and turn on interrupts after the critical section. Guaranteed atomicity. It is also fairly efficient. Early versions of Unix did this.

 Why not just use turning off interrupts? Two main disadvantages: can't use in a multiprocessor, and can't use directly from user program for synchronization.

 Test-And-Set. The test and set instruction atomically checks if a memory location is zero, and if so, sets the memory location to 1. If the memory location is 1, it does nothing. It returns the old value of the memory location. You can use test and set to implement locks as follows:

- The lock state is implemented by a memory location. The location is 0 if the lock is unlocked and 1 if the lock is locked.
- The lock operation is implemented as:
- while (test-and-set(l) == 1);
- The unlock operation is implemented as: *l = 0;

The problem with this implementation is busy-waiting. What if one thread already has the lock, and another thread wants to acquire the lock? The acquiring thread will spin until the thread that already has the lock unlocks it.

 What if the threads are running on a uniprocessor? How long will the acquiring thread spin? Until it expires its quantum and thread that will unlock the lock runs. So on a uniprocessor, if can't get the thread the first time, should just suspend. So, lock acquisition looks like this:

```
while (test-and-set(l) == 1) {
 currentThread->Yield();
}
```

Can make it even better by having a queue lock that queues up the waiting threads and gives the lock to the first thread in the queue. So, threads never try to acquire lock more than once.

There are three components of the cost: spinning, suspending and resuming.

What is the cost of spinning? Waste the CPU for the spin time.

What is cost of suspending and resuming? Amount of CPU time it takes to suspend the thread and restart it when the thread acquires the lock.

Requirements of Synch Mech

- modularity
  - separation of resource and its access operations, and synchronizer and its mechanisms
- expressive power
  - specifying the needed constraints (exclusion and priority constrains in terms of relevant information…)

- ease of use
  - composing a set of constraints (for complex synch. schemes)
- modifiability
  - changing the constraints if needed
- correctness
  - safety net against inadvertent user errors

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem

Bounded-Buffer Problem

- *N* buffers, each can hold one item
- Semaphore mutex initialized to the value 1
- Semaphore full initialized to the value 0
- Semaphore empty initialized to the value N

The structure of the producer process

```
do {
// produce an item in nextp
    wait (empty);
    wait (mutex);
// add the item to the buffer
    signal (mutex);
    signal (full);
} while (TRUE);
```

- The structure of the consumer process

```
do {
    wait (full);
    wait (mutex);
// remove an item from buffer to nextc
    signal (mutex);
    signal (empty);
// consume the item in nextc
} while (TRUE);
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do not perform any updates
  - Writers – can both read and write

- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
  - Several variations of how readers and writers are treated – all involve priorities
- Shared Data
  - Data set
  - Semaphore mutex initialized to 1
  - Semaphore wrt initialized to 1
  - Integer readcount initialized to 0

- **The structure of a writer process**

```
do {
        wait (wrt) ;
   //   writing is performed
        signal (wrt) ;
} while (TRUE);
```

- **The structure of a reader process**

```
do {
        wait (mutex) ;
        readcount ++ ;
        if (readcount == 1)
                wait (wrt) ;
        signal (mutex)
    // reading is performed
        wait (mutex) ;
        readcount  - - ;
        if (readcount  == 0)
                signal (wrt) ;
        signal (mutex) ;
} while (TRUE);
```

Readers-Writers Problem Variations

- *First* variation – no reader kept waiting unless writer has permission to use shared object
- *Second* variation – once writer is ready, it performs write asap
- Both may have starvation leading to even more variations

  Problem is solved on some systems by kernel providing reader-writer locks

Process and threads

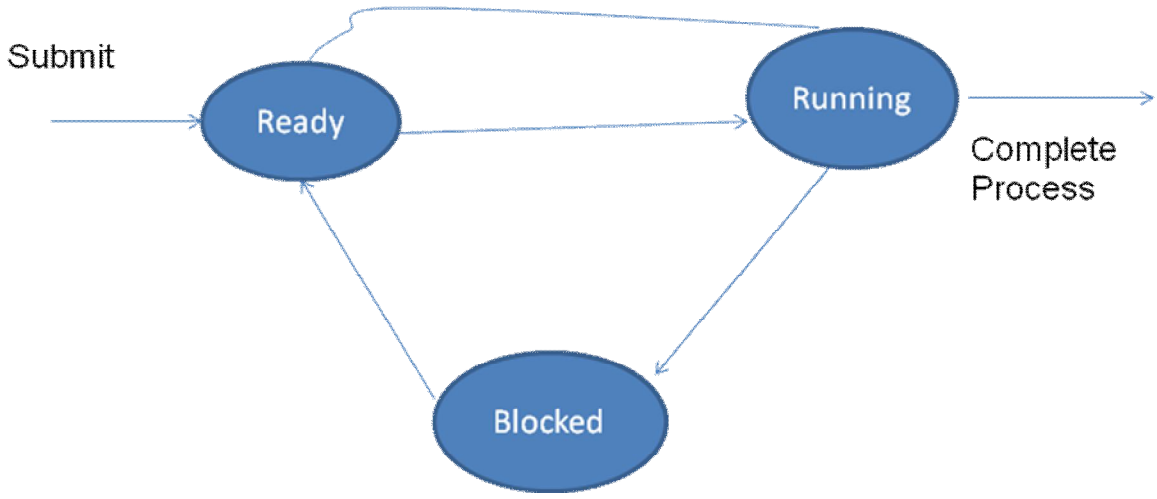
Process :

- Its an executing instance of a Program is called a process
- Its always stored in the main memory
- The several process may be associated with a same program.
- It has 3 basic states

✓ running

✓ Ready

✓ Blocked



Thread:

- ⊡ A thread is a sub set of the process.
- ⊡ Its Termed as a Lightweight Process.
- ⊡ Usually a process has only one thread of control.

A flow of control through an address space, each address space can have multiple concurrent control flows.

Why use threads ?

- ⊡ Large multi processors need many computing entities(one per CPU)
- ⊡ Threads have full access to address space(easy sharing)
- ⊡ Threads can execute in parllel on multi processor

Different Between threads and Process

Thread:	Process:
✓ It share the address space of the process that created it	Processes have their own address space.
✓ It have direct access to the segment of its process.	it have their own copy of data The data segment of the parent process.
✓ Threads can directly communicate with other threads of its process.	Process must use inter Process communication to communicate with sibling process.
✓ Threads have almost no overhead	Process have considerable overhead.
✓ New threads are created easily.	New process require duplication of the parent process

Process Scheduling Algorithm

- ⊡ FIFO (FCFS)
- ⊡ Shortest – Job – First (SJF)
- ⊡ Priority Based
- ⊡ Round Robin
- ⊡ Multilevel Queue.
- ⊡ Multilevel Feedback Queue.

FIFO

- Easily implemented with a FIFO queue
- Process's PCB initially placed on FIFO Ready queue
- When CPU is free the next Process is selected
- *Problem: Average waiting time may be long with this policy*
  - Consider the case where a long running process precedes a short running process

Example:

Process	Burst Time
P_1	24

P_2 3

P_3 3

Assume processes arrive as: P_1 , P_2 , P_3

The **Gantt Chart** for the schedule is:

- *Waiting time* for P_1 = 0; P_2 = 24; P_3 = 27
- *Average waiting time*: (0 + 24 + 27)/3 = 17

Suppose The Process will be arrives : P2, P3 ,P1

Waiting Time for P1= 6, P2= 0 ,P3= 3

Average Waiting Time: $\frac{(6+0+0)}{3} = \frac{9}{3} = 3$

Shortest-Job-First (SJR) Scheduling

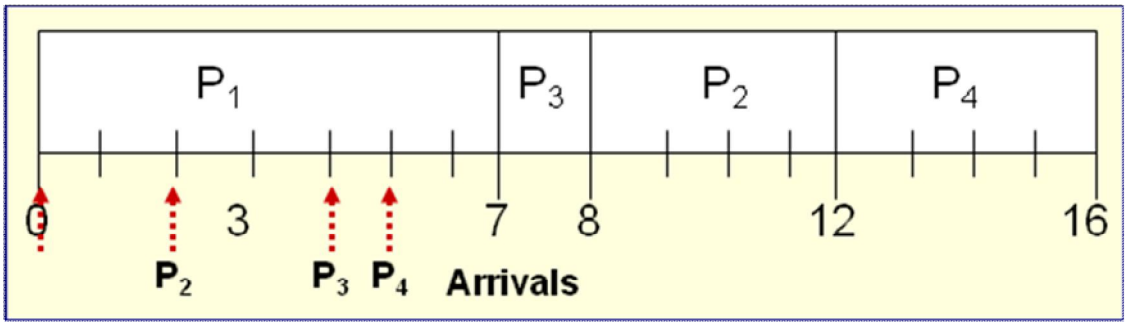
- Process with shortest burst goes next
  - if tie then use FCFS to break tie
- Scheduler must “know” the next CPU burst length of each process in Ready queue
  - either process declares burst length or system “predicts” next length based on previous usage
- SJF is optimal in that it provides the minimum average waiting time for a given set of processes.

- Two schemes:

– *non-preemptive* – once CPU assigned, process not preempted until its CPU burst completes.
– *Preemptive* – if a new process with CPU burst less than remaining time of current, preempt.
Shortest-Remaining-Time-First (SRTF).

Example of Non-Preemptive SJF

- T = 0: RQ = {P_1}
  Select P_1
- T = 2: RQ = {P_2}
  No-Preemption
- T = 4: RQ = {P_3, P_2}
  No-Preemption
- T = 5: RQ = {P_3, P_2 , P_4}
  No-Preemption
- T = 7: RQ = {P_3, P_2 , P_4}
  P_1 completes, Select P_3
- T = 8: RQ = {P_2 , P_4}
  P_3 completes, Select P_2
- Average Waiting Time:
  [0 + (8 – 2) + (7 – 4) + (12 – 5)]/4 =
  [6 + 3 + 7]/4 = 4



- T = 12: RQ = {P_4}
  P_2 completes, Select P_4
- T = 16: RQ = {}
  P_4 completes

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- T = 0: RQ = {P_1}
  Select P_1

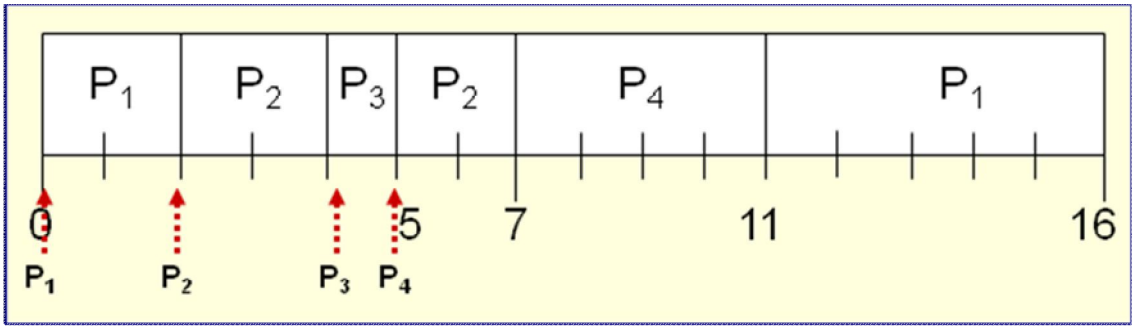
Example of Preemptive SJF

- T = 2: RQ = {P_2} *preempt P_1 , Select P_2*
- T = 4: RQ = {P_3, P_1} *preempt P_2 , Select P_3*
- T = 5: RQ = {P_2, P_4 , P_1} *P_3 completes, Select P_2*
- T = 7: RQ = {P_4, P_1} *P_2 completes, Select P_4*
- T = 11: RQ = {P_1} *P_4 completes, Select P_1*

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- Average Waiting Time:
  [(11-2) + (5-4) + (0) + (7-5)]/4 =
  [9 + 1 + 0 + 2]/4 = 3

- T = 16: RQ = {} *$P_2$1completes*



Priority Scheduling

- SJF is an example of priority-based scheduling
- Associate priority with each process and select highest priority when making scheduling decision
  - Preemptive or non-preemptive
- Priority may be internally defined by OS or externally by the user
  - statically assigned priority or dynamically set based on execution properties

- User may declare relative importance

- Problem: Starvation
  - low priority processes may never execute.
- Solution: Aging
  - as time progresses increase the priority of the process.

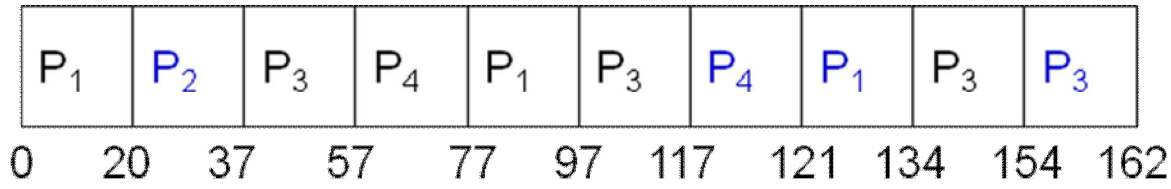
Round Robin (RR)

- Policy is targeted to time-sharing systems
- Similar to FCFS but permit preemption
  - preempt after fixed time interval and place on tail of ready-queue.
  - always select next process from head of ready-queue
- Each process assigned a *time quantum*, typically 10-100ms (40ms).
  - At quantum expiration process moved to tail of Ready Q
- if *n* processes in ready queue, time quantum = *q*, then each process receives *1/n* of CPU time in chunks of at most *q* units.
  - No process waits more than *(n-1)q* time units for CPU.
- Performance depends on size of time quantum
  - *q* large then behaves like FCFS
  - *q* small then appears as dedicated processor with speed 1/n actual – called *processor sharing*
  - *Want q large compared to context switch time*
- Turnaround time also depends on quantum
  - better performance if most processes complete within one time quantum

Example: RR, Quantum = 20

Process	**Burst Time**
P_1	**53**
P_2	**17**
P_3	**68**
P_4	**24**

- **The Gantt chart is:**



Typically, higher average turnaround than SJF, but better *response*.

CPU Scheduling

 What is CPU scheduling? Determining which processes run when there are multiple runnable processes. Why is it important? Because it can can have a big effect on resource utilization and the overall performance of the system.

 By the way, the world went through a long period (late 80's, early 90's) in which the most popular operating systems (DOS, Mac) had NO sophisticated CPU scheduling algorithms. They were single threaded and ran one process at a time until the user directs them to run another process. Why was this true? More recent systems (Windows NT) are back to having sophisticated CPU scheduling algorithms. What drove the change, and what will happen in the future?

 Basic assumptions behind most scheduling algorithms:

- There is a pool of runnable processes contending for the CPU.
- The processes are independent and compete for resources.
- The job of the scheduler is to distribute the scarce resource of the CPU to the different processes ``fairly'' (according to some definition of fairness) and in a way that optimizes some performance criteria.

What are possible process states?

- Running - process is running on CPU.
- Ready - ready to run, but not actually running on the CPU.
- Waiting - waiting for some event like IO to happen.

How to evaluate scheduling algorithm? There are many possible criteria:

- CPU Utilization: Keep CPU utilization as high as possible. (What is utilization, by the way?).
- Throughput: number of processes completed per unit time.
- Turnaround Time: mean time from submission to completion of process.
- Waiting Time: Amount of time spent ready to run but not running.
- Response Time: Time between submission of requests and first response to the request.
- Scheduler Efficiency: The scheduler doesn't perform any useful work, so any time it takes is pure overhead. So, need to make the scheduler very efficient.

The Deadlock Problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
- Example
  - System has 2 tape drives.
  - P_1 and P_2 each hold one tape drive and each needs another one.
- Example
  - semaphores *A* and *B*, initialized to 1

P_0	P_1
wait (A);	*wait(B)*
wait (B);	*wait(A)*

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously**.**

- Mutual exclusion: only one process at a time can use a resource.
- Hold and wait: a process holding at least one resource is waiting to acquire additional resources held by other processes.
- No preemption: a resource can be released only voluntarily by the process holding it, after that process has completed its task.
- Circular wait: there exists a set $\{P_0, P_1, \ldots, P_0\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by

P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_0 is waiting for a resource that is held by P_0.

Methods for Handling Deadlocks

- Ensure that the system will *never* enter a deadlock state.
- Allow the system to enter a deadlock state and then recover.
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.

Deadlock Prevention

Restrain the ways request can be made.

- **Mutual Exclusion** – not required for sharable resources; must hold for nonsharable resources.
- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources.
  - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none.

  Low resource utilization; starvation possible.
- **No Preemption** –
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released.
  - Preempted resources are added to the list of resources for which the process is waiting.
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.

Deadlock Avoidance

Requires that the system has some additional *a priori* information available.

- Simplest and most useful (??) model requires that each process declare the *maximum number* of resources of each type that it may need.
- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes.
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.

Deadlock

 You may need to write code that acquires more than one lock. This opens up the possibility of deadlock. Consider the following piece of code:

```
Lock *l1, *l2;
void p() {
  l1->Acquire();
  l2->Acquire();
  code that manipulates data that l1 and l2 protect
  l2->Release();
  l1->Release();
}
void q() {
  l2->Acquire();
  l1->Acquire();
  code that manipulates data that l1 and l2 protect
  l1->Release();
  l2->Release();
}
```

If p and q execute concurrently, consider what may happen. First, p acquires l1 and q acquires l2. Then, p waits to acquire l2 and q waits to acquire l1. How long will they wait? Forever.

 This case is called deadlock. What are conditions for deadlock?

- Mutual Exclusion: Only one thread can hold lock at a time.

- Hold and Wait: At least one thread holds a lock and is waiting for another process to release a lock.
- No preemption: Only the process holding the lock can release it.
- Circular Wait: There is a set t_1, ..., t_n such that t_1 is waiting for a lock held by t_2, ..., t_n is waiting for a lock held by t_1.

How can p and q avoid deadlock? Order the locks, and always acquire the locks in that order. Eliminates the circular wait condition.

Memory Management

- Operating systems must employ techniques to
  - Track where and how a program resides in memory
  - Convert **logical addresses** into actual **addresses**
- **Logical address** (sometimes called a virtual or relative address) A value that specifies a generic location, relative to the program but not to the reality of main memory
- **Physical address** An actual address in the main memory device

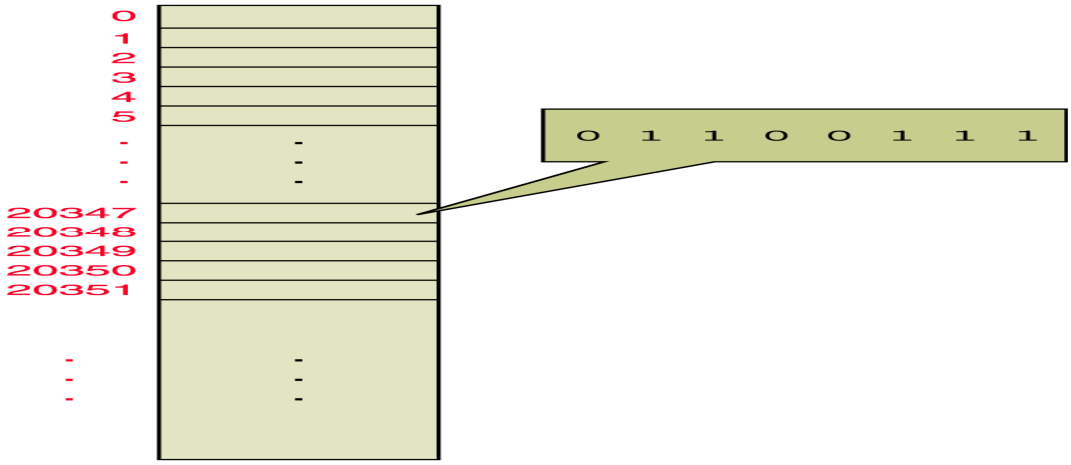


Figure 10.3
Memory is a continuous set of bits referenced by specific addresses

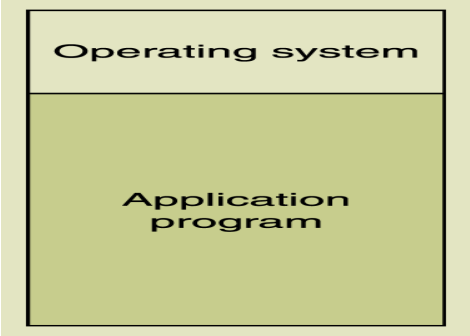


Figure 10.4 Main memory divided into two sections

- **There are only two programs in memory**
  - **The operating system**
  - **The application program**
- **This approach is called single contiguous memory management**
- A logical address is simply an integer value relative to the starting point of the program
- To produce a physical address, we add a logical address to the starting address of the program in physical main memory

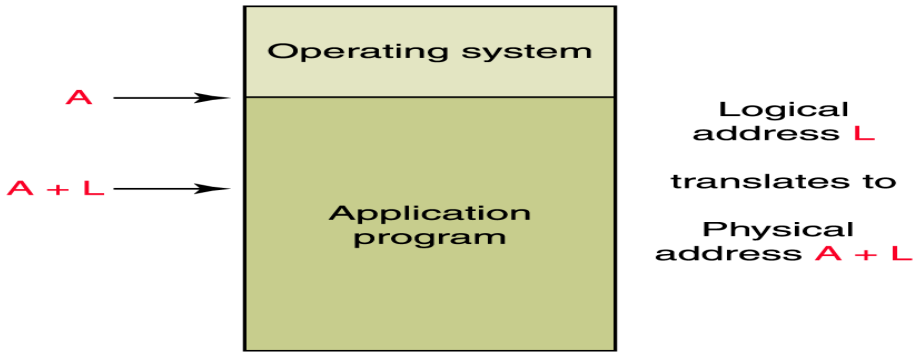
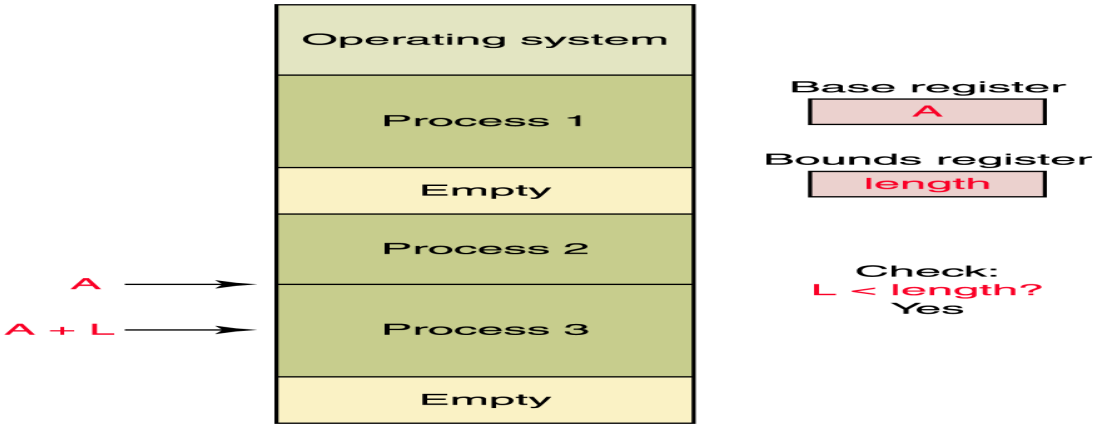


Figure 10.5 binding a logical address to a physical one

Partition Memory Management

- Fixed partitions Main memory is divided into a particular number of partitions
- Dynamic partitions Partitions are created to fit the needs of the programs



- At any point in time memory is divided into a set of partitions, some empty and some allocated to programs
- **Base register** A register that holds the beginning address of the current partition
- **Bounds register** A register that holds the length of the current partition

Which partition should we allocate to a new program?

- **First fit** Allocate program to the first partition big enough to hold it
- **Best fit** Allocated program to the smallest partition big enough to hold it
- **Worst fit** Allocate program to the largest partition big enough to hold it

Paged memory technique

A memory management technique in which processes are divided into fixed-size **pages** and stored in memory **frames** when loaded into memory

- **Frame** A fixed-size portion of *main memory* that holds a process page
- **Page** A fixed-size portion of a *process* that is stored into a memory frame
- **Page-map table** (PMT) A table used by the operating system to keep track of page/frame relationships

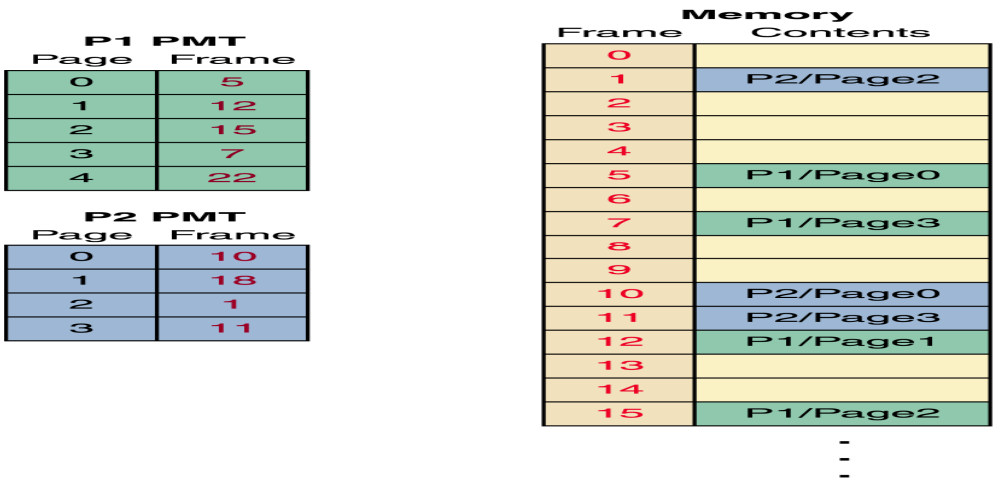


Figure 10.7 A paged memory management approach

- To produce a physical address, you first look up the page in the PMT to find the frame number in which it is stored
- Then multiply the frame number by the frame size and add the offset to get the physical address
- **Demand paging** An important extension of paged memory management
  - Not all parts of a program actually have to be in memory at the same time
  - In demand paging, the pages are brought into memory on demand
- Page swap The act of bringing in a page from secondary memory, which often causes another page to be written back to secondary memory
- The demand paging approach gives rise to the idea of **virtual memory,** the illusion that there are no restrictions on the size of a program
- Too much page swapping, however, is called **thrashing** and can seriously degrade system performance.

Introduction to Memory Management

 Point of memory management algorithms - support sharing of main memory. We will focus on having multiple processes sharing the same physical memory. Key issues:

- Protection. Must allow one process to protect its memory from access by other processes.
- Naming. How do processes identify shared pieces of memory.

- Transparency. How transparent is sharing. Does user program have to manage anything explicitly?
- Efficiency. Any memory management strategy should not impose too much of a performance burden.

 Why share memory between processes? Because want to multiprogram the processor. To time share system, to overlap computation and I/O. So, must provide for multiple processes to be resident in physical memory at the same time. Processes must share the physical memory.

Introduction to Paging

 Basic idea: allocate physical memory to processes in fixed size chunks called page frames. Present abstraction to application of a single linear address space. Inside machine, break address space of application up into fixed size chunks called pages. Pages and page frames are same size. Store pages in page frames. When process generates an address, dynamically translate to the physical page frame which holds data for that page.

 So, a virtual address now consists of two pieces: a page number and an offset within that page. Page sizes are typically powers of 2; this simplifies extraction of page numbers and offsets. To access a piece of data at a given address, system automatically does the following:

- Extracts page number.
- Extracts offset.
- Translate page number to physical page frame id.
- Accesses data at offset in physical page frame.

 How does system perform translation? Simplest solution: use a page table. Page table is a linear array indexed by virtual page number that gives the physical page frame that contains that page. What is lookup process?

- Extract page number.
- Extract offset.
- Check that page number is within address space of process.
- Look up page number in page table.
- Add offset to resulting physical page number

- Access memory location.

Problem: for each memory access that processor generates, must now generate two physical memory accesses.

 Speed up the lookup problem with a cache. Store most recent page lookup values in TLB. TLB design options: fully associative, direct mapped, set associative, etc. Can make direct mapped larger for a given amount of circuit space.

 How does lookup work now?

- Extract page number.
- Extract offset.
- Look up page number in TLB.
- If there, add offset to physical page number and access memory location.
- Otherwise, trap to OS. OS performs check, looks up physical page number, and loads translation into TLB. Restarts the instruction.

What about protection? There are a variety of protections:

- Preventing one process from reading or writing another process' memory.
- Preventing one process from reading another process' memory.
- Preventing a process from reading or writing some of its own memory.
- Preventing a process from reading some of its own memory.

  How is this protection integrated into the above scheme?

  Preventing a process from reading or writing memory: OS refuses to establish a mapping from virtual address space to physical page frame containing the protected memory. When program attempts to access this memory, OS will typically generate a fault. If user process catches the fault, can take action to fix things up.

- Preventing a process from writing memory, but allowing a process to read memory. OS sets a write protect bit in the TLB entry. If process attempts to write the memory, OS generates a fault. But, reads go through just fine.

Virtual Memory Introduction.

 When a segmented system needed more memory, it swapped segments out to disk and then swapped them back in again when necessary. Page based systems can do something similar on a page basis.

 What is advantage of virtual memory/paging?

- Can run programs whose virtual address space is larger than physical memory. In effect, one process shares physical memory with itself.
- Can also flexibly share machine between processes whose total address space sizes exceed the physical memory size.
- Supports a wide range of user-level stuff - See Li and Appel paper.

 Disadvantages of VM/paging: extra resource consumption.

- Memory overhead for storing page tables. In extreme cases, page table may take up a significant portion of virtual memory. One Solution: page the page table. Others: go to a more complicated data structure for storing virtual to physical translations.
- Translation overhead.

Issues in Paging and Virtual Memory

 Page Table Structure. Where to store page tables, issues in page table design.

 In a real machine, page tables stored in physical memory. Several issues arise:

- How much memory does the page table take up?
- How to manage the page table memory. Contiguous allocation? Blocked allocation? What about paging the page table?

 On TLB misses, OS must access page tables. Issue: how the page table design affects the TLB miss penalty.

 Real operating systems provide the abstraction of sparse address spaces. Issue: how well does a particular page table design support sparse address spaces?

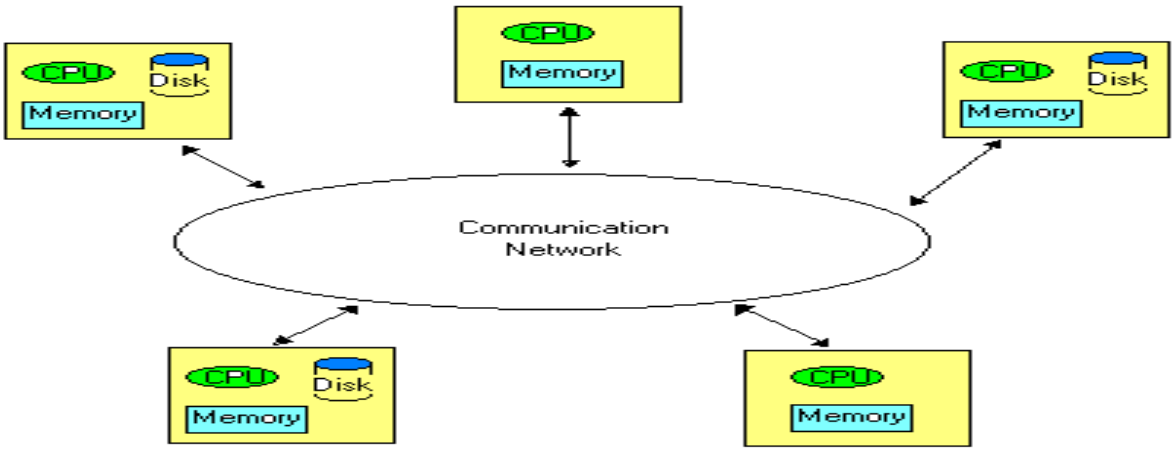
UNIT 2

UNIT II DISTRIBUTED OPERATING SYSTEMS 9

Issues in Distributed Operating System – Architecture – Communication Primitives – Lamport's Logical clocks – Causal Ordering of Messages – Distributed Mutual Exclusion Algorithms – Centralized and Distributed Deadlock Detection Algorithms – Agreement Protocols.

Distributed Operating Systems

- Consists of several computers that do not share a memory or a clock;
- The computers communicate with each other by exchanging messages over a communication network; and
- Each computer has its own memory and runs its own operating system.



Definition of Distributed OS

It extends the concepts of resource management and user friendly interface for shared memory computers a step further, encompassing a distributed computing system consisting of several autonomous computers connected by a communicating network.

Issues in Distributed OS

- Global Knowledge
- Naming
- Scalability
- Compatibility
- Process Synchronization
- Resource Management
- Security
- Structuring

The Main Advantage of Distributed Systems is:

- They have a decisive price/ performance advantage over more traditional time-sharing systems
- Resource Sharing
- Enhanced Performance
- Improved Reliability and availability
- Modular expandability

Categories of Distributed Systems:

- Minicomputer Model
- Workstation Model
- Processor Pool Model

Logical clocks

Assign sequence numbers to messages

- All cooperating processes can agree on order of events
- vs. physical clocks: time of day

Assume no central time source

- Each system maintains its own local clock

– No total ordering of events

• No concept of *happened-when*

Happened-before

Lamport's "happened-before" notation

$a \rightarrow b$ event *a* happened before event *b*

e.g.: *a:* message being sent, *b*: message receipt

Transitive:

if $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$

Logical clocks & concurrency

Assign "clock" value to each event

– if $a \rightarrow b$ then clock(a) < clock(b)

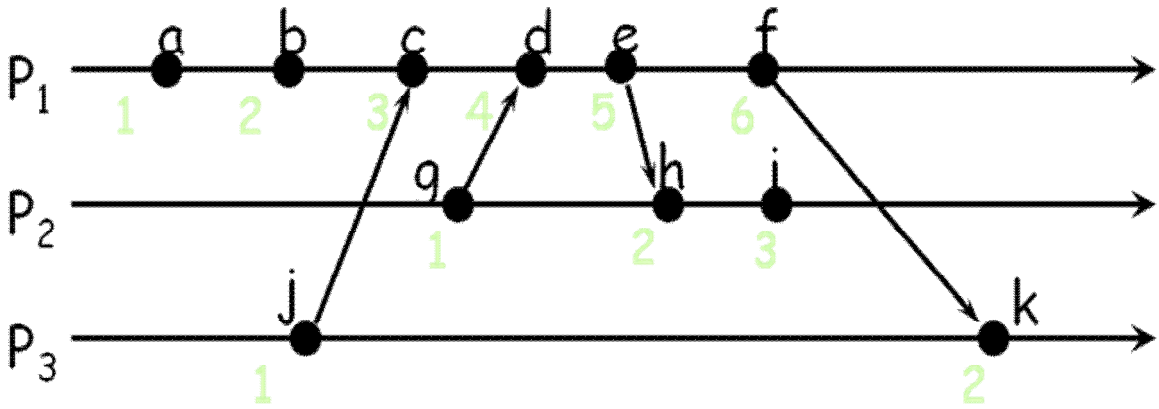
– since time cannot run backwards

If *a* and *b* occur on different processes that do not exchange messages, then neither $a \rightarrow b$ nor $b \rightarrow a$ are true

– These events are concurrent

Event counting example

- Three systems: P_0, P_1, P_2
- Events *a*, *b*, *c*, …
- Local event counter on each system
- Systems occasionally communicate

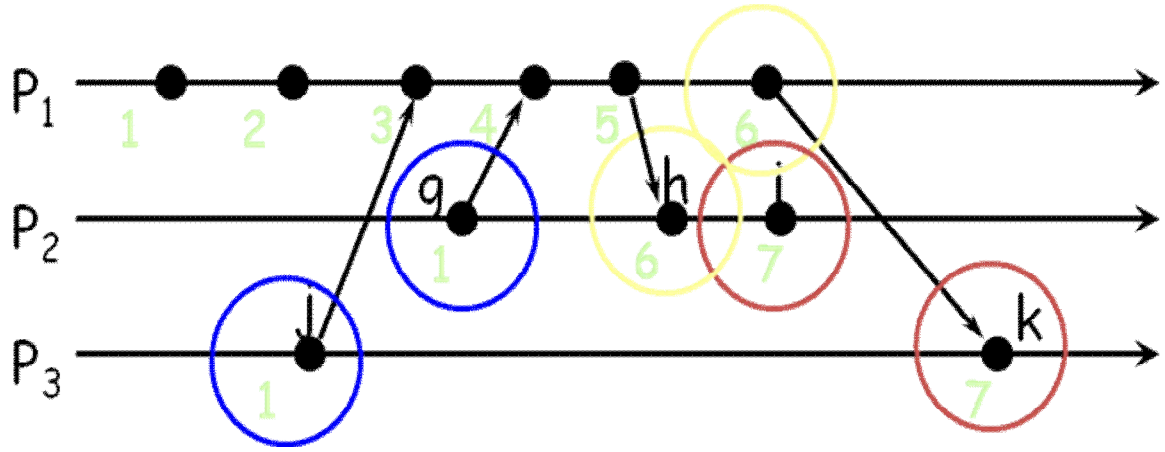


Summary

- Algorithm needs monotonically increasing software counter
- Incremented at least when events that need to be timestamped occur
- Each event has a **Lamport timestamp** attached to it
- For any two events, where a → b:

  L(a) < L(b)

Problem: Identical timestamps



a→b, b→c, …: local events sequenced

i→c, f→d , d→g, … : Lamport imposes a *send→receive* relationship

Concurrent events (e.g., a & i) *may* have the same timestamp … or not

Unique timestamps (total ordering)

We can force each timestamp to be unique

- Define global logical timestamp (T_i, i)
  - T_i represents local Lamport timestamp
  - i represents process number (globally unique)
    - E.g. (host address, process ID)
- Compare timestamps:

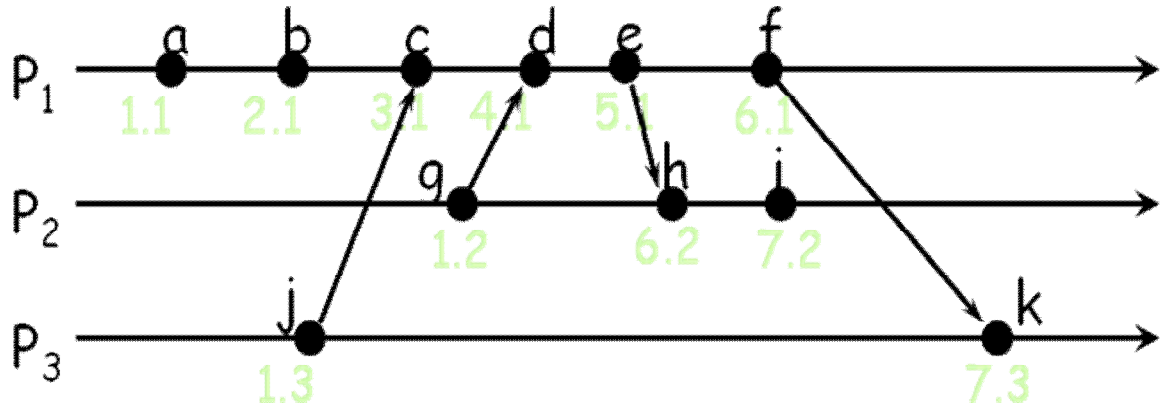
  (T_i, i) < (T_j, j)

  if and only if

  $T_i < T_j$ or

  $T_i = T_j$ and i < j

Does not relate to event ordering



Problem: Detecting causal relations

If L(e) < L(e')

- Cannot conclude that e→e'

Looking at Lamport timestamps

- Cannot conclude which events are causally related

Solution: use a vector clock

Vector clocks:

Rules:

1. Vector initialized to 0 at each process
   $V_i[j] = 0$ for $i, j = 1, \ldots, N$
2. Process increments its element of the vector in local vector before timestamping event:
   $V_i[i] = V_i[i] + 1$
3. Message is sent from process P_i with V_i attached to it
4. When P_j receives message, compares vectors element by element and sets local vector to higher of two values
   $V_j[i] = \max(V_i[i], V_j[i])$ for i=1,

Comparing vector timestamps:

Define

$V = V'$ iff $V[i] = V'[i]$ for $i = 1 \ldots N$

$V \leq V'$ iff $V[i] \leq V'[i]$ for $i = 1 \ldots N$

For any two events e, e'

if e → e' then V(e) < V(e')

- Just like Lamport's algorithm

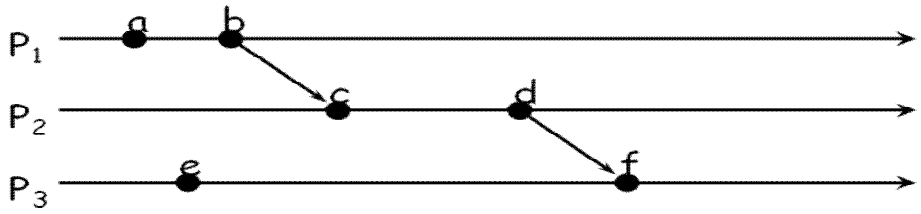
if **V(e) < V(e') then e → e'**

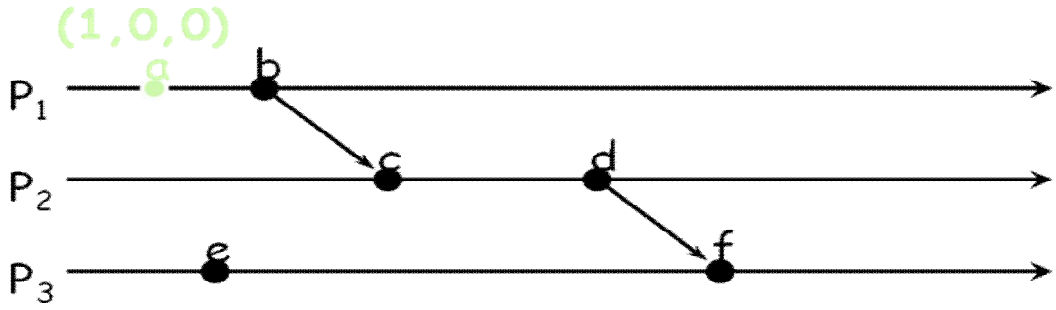
Two events are **concurrent** if **neither**

V(e) ≤ V(e') nor V(e') ≤ V(e)

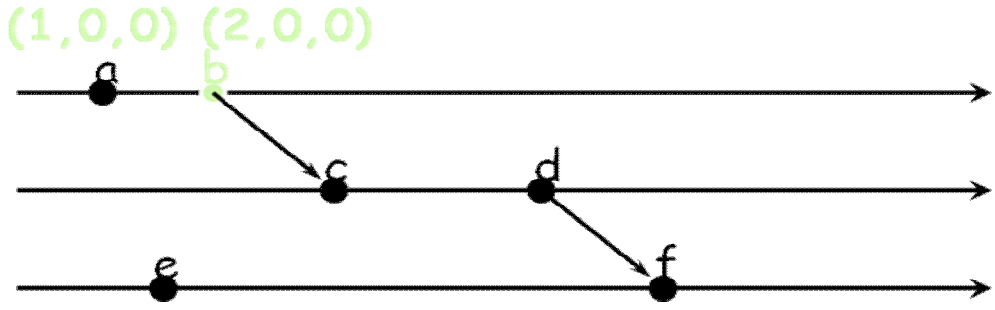
Vector timestamps

(0,0,0)
(0,0,0)
(0,0,0)

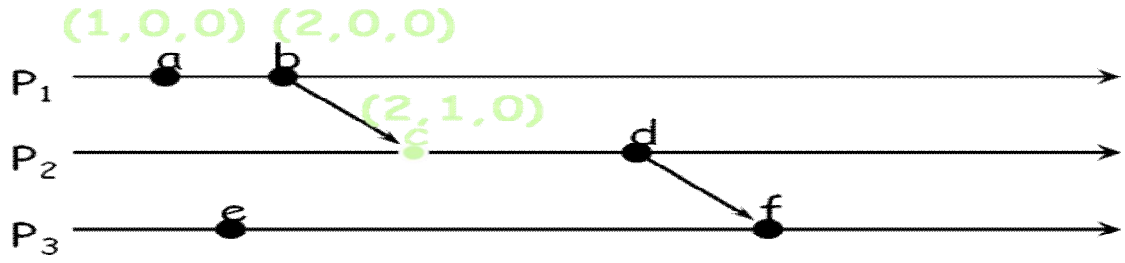




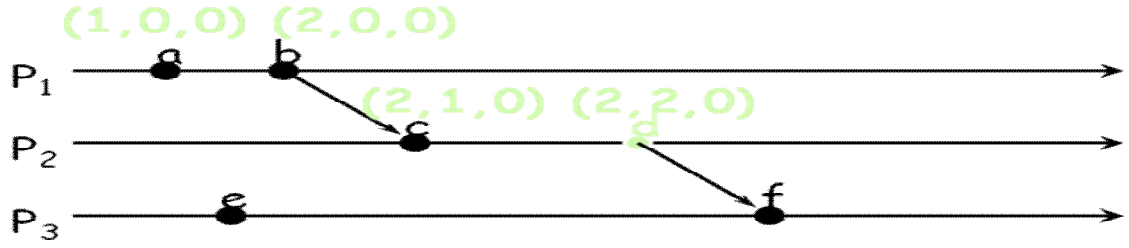
Event	timestamp
a	(1,0,0)



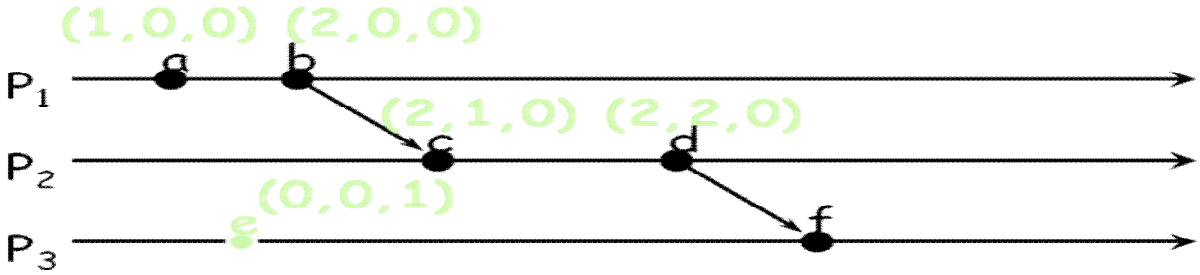
Event	timestamp
a	(1,0,0)
b	(2,0,0)



Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)

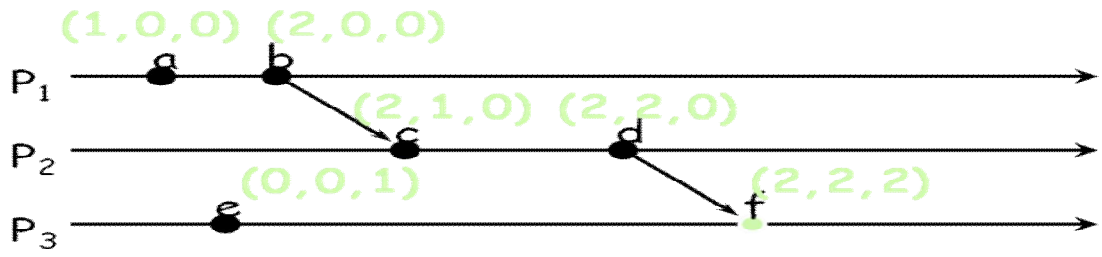


Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)

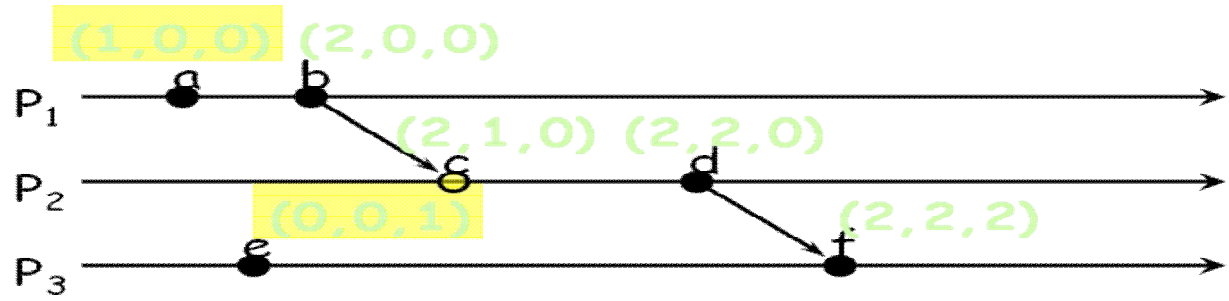


Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)

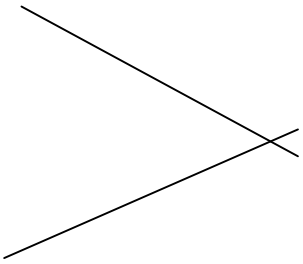
d	(2,2,0)
e	(0,0,1)



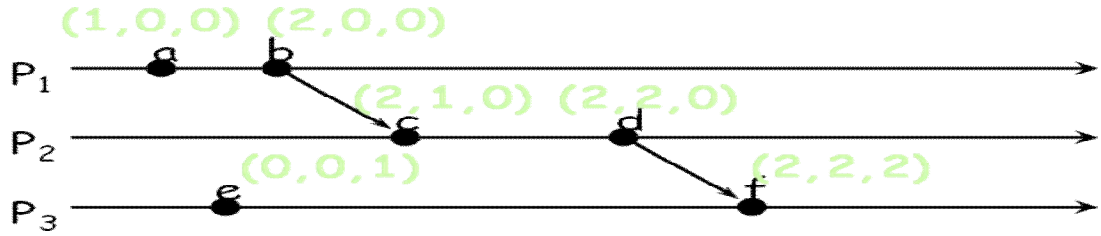
Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)
e	(0,0,1)
f	(2,2,2)



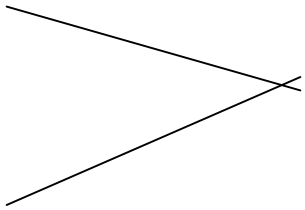
Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)
e	(0,0,1)
f	(2,2,2)



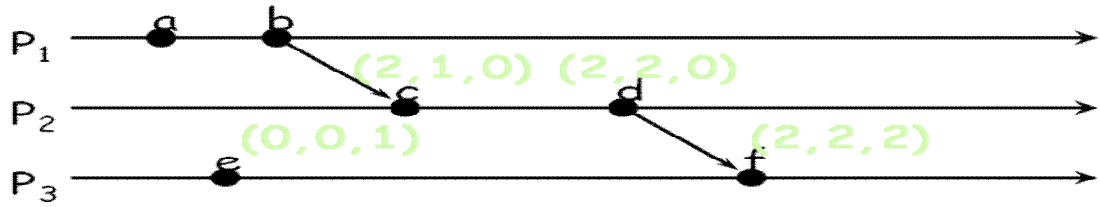
concurrent events



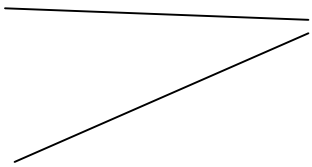
Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)
e	(0,0,1)
f	(2,2,2)



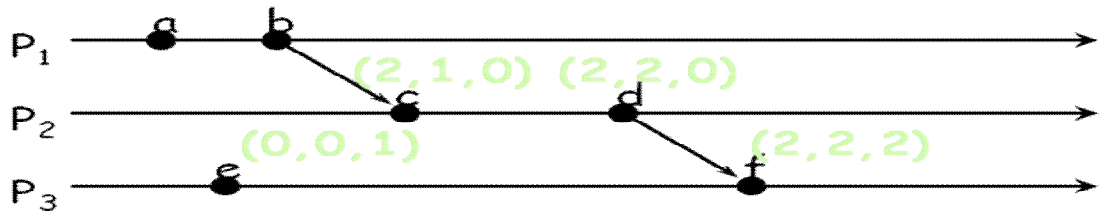
concurrent events



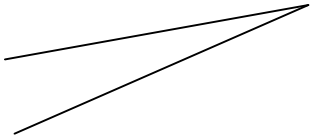
Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)
e	(0,0,1)
f	(2,2,2)



concurrent events



Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)
e	(0,0,1)
f	(2,2,2)



concurrent events

Summary: Logical Clocks & Partial Ordering

- Causality
  - If *a*->*b* then event *a* can affect event *b*
- Concurrency
  - If neither *a*->*b* nor *b*->*a* then one event cannot affect the other
- Partial Ordering
  - Causal events are sequenced
- Total Ordering
  - All events are sequenced

Causal Ordering of Messages:

An Application of Vector Clocks

- **Premise**: Deliver a message only if messages that causally precede it have already been received
  - *i.e.*, if $send(m_1) \rightarrow send(m_2)$, then it should be true that $receive(m_1) \rightarrow receive(m_2)$ at each site.
  - If messages are not related ($send(m_1) \parallel send(m_2)$), delivery order is not of interest.

Compare to Total Order

- Totally ordered multicast (TOM) is stronger (more inclusive) than causal ordering (COM).
  - TOM orders *all* messages, not just those that are causally related.
  - "Weaker" COM is often what is needed.

Enforcing Causal Communication

- Clocks are adjusted only when sending or receiving messages; i.e, these are the only events of interest.
- Send *m*: P_i increments $VC_i[i]$ by 1 and applies timestamp, $ts(m)$.
- Receive *m*: Pi compares VC_i to $ts(m)$; set $VC_i[k]$ to $max\{VC_i[k], ts(m)[k]\}$ for each k, $k \neq i$.

Message Delivery Conditions

- Suppose: P_J receives message *m* from P_i
- Middleware delivers *m* to the application iff
  - $ts(m)[i] = VC_j[i] + 1$
    - all previous messages from P_i have been delivered
  - $ts(m)[k] \leq VC_i[k]$ for all $k \neq i$
    - P_J has received all messages that P_i had seen before it sent message *m*.
- In other words, if a message *m* is received from P_i, you should also have received every message that P_i received before it sent m; e.g.,

- if m is sent by P_1 and *ts(m)* is (3, 4, 0) and you are P_3, you should already have received exactly 2 messages from P_1 and at least 4 from P_2
- if m is sent by P_2 and ts(m) is (4, 5, 1, 3) and if you are P_3 and VC_3 is (3, 3, 4, 3) then you need to wait for a fourth message from P_2 and at least one more message from P_1.

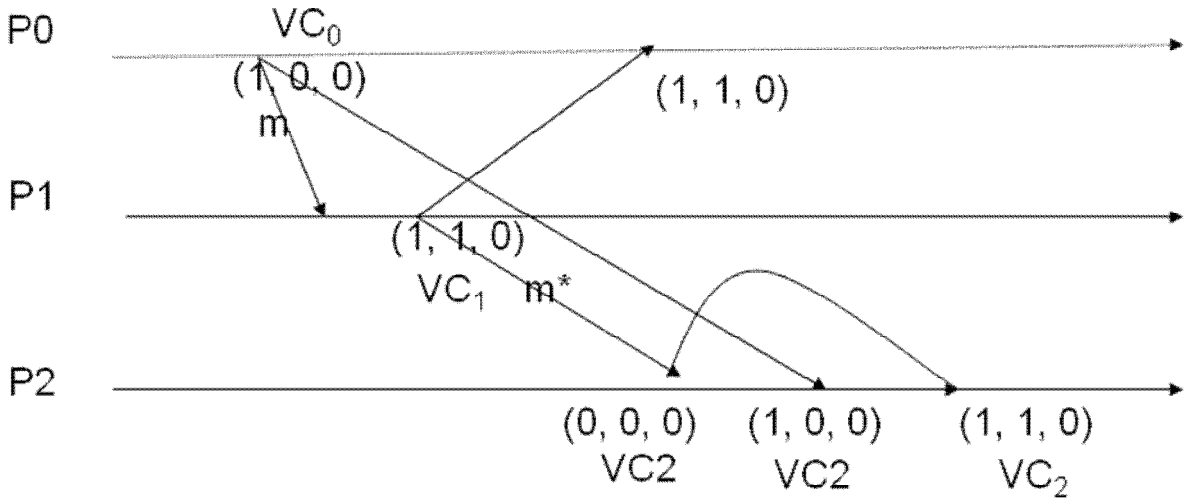


Figure . Enforcing Causal Communication

P1 received message *m* from P0 before sending message *m** to P2; P2 must wait for delivery of *m* before receiving *m**

(Increment own clock only on message send)

Before sending or receiving any messages, one's own clock is (0, 0, …0)

Location of Message Delivery

- Problems if located in middleware:
  - Message ordering captures only potential causality; no way to know if two messages from the same source are actually dependent.
  - Causality from other sources is not captured.
- End-to-end argument: the application is better equipped to know which messages are causally related.
- But … developers are now forced to do more work; re-inventing the wheel.

Distributed Mutual Exclusion Algorithms

Mutual exclusion is a collection of techniques for sharing resources so that different uses do not conflict and cause unwanted interactions.

primary objective of mutual exclusion?

To maintain mutual exclusion; that is, to guarantee that only one request accesses the critical section at a time.

Characteristics are considered important in a mutual exclusion algorithm:

- Freedom from Deadlocks
- Freedom from Starvation
- Fairness
- Fault Tolerance

How to measure the performance

- The number of messages necessary per CS invocation.
- The synchronization delay
- The response time

There are three major communication scenarios:

1. One-way Communication usually do not need synchronization.

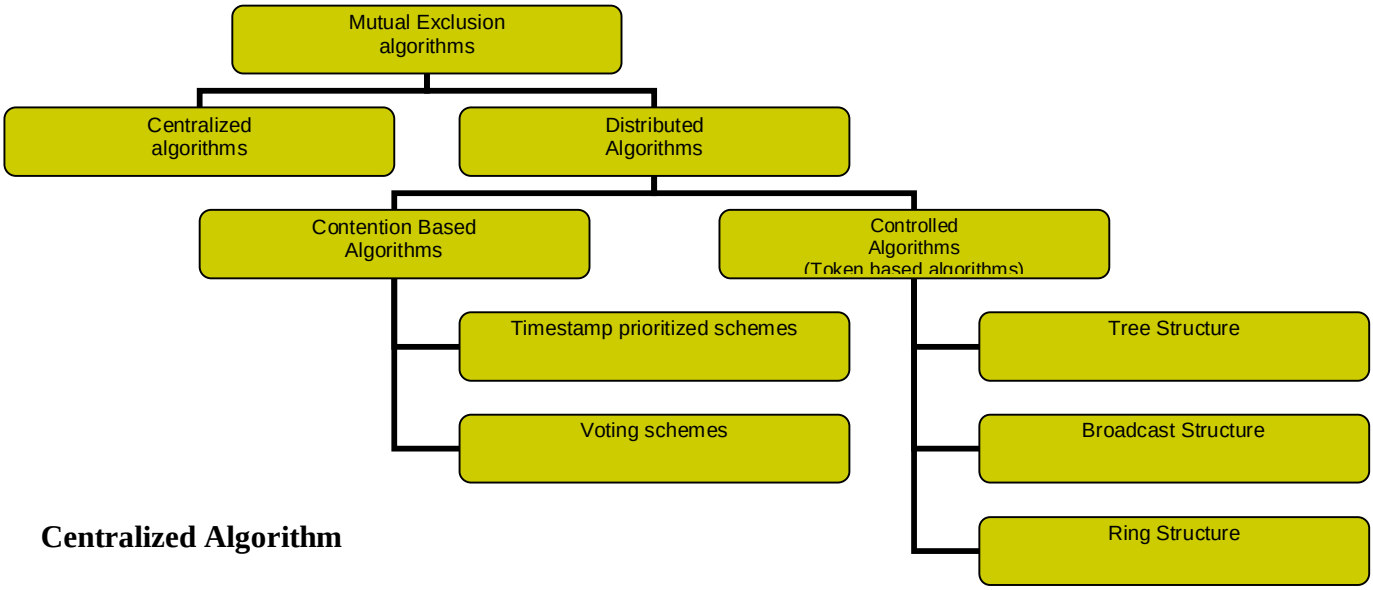
2. Client/server communication is for multiple clients making service request to a shared server. If co-ordination is required among the clients, it is handled by the server and there is no explicit interaction among client process.
3. Interprocess communication:

   -> not limited to making service requests.

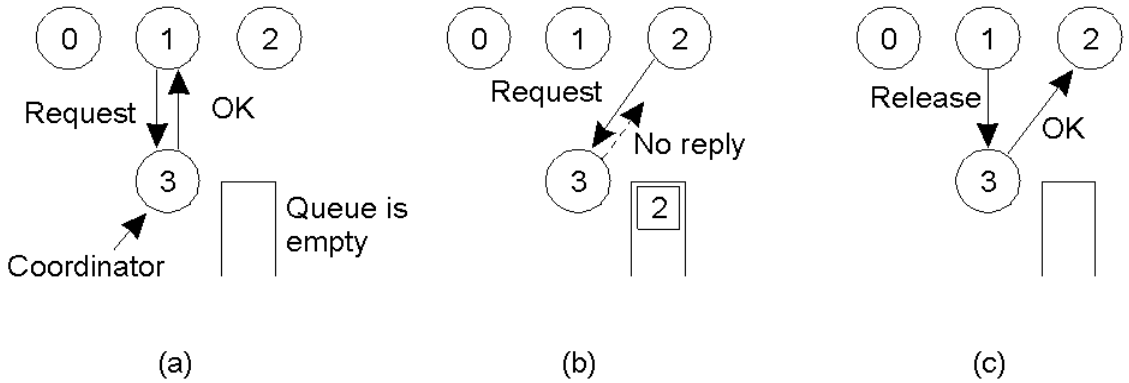
   -> processes need to exchange information to reach some conclusion about the system or some agreement among the cooperating processes.

These activities require peer communication; there is no shared object or centralized controller.

Mutual Exclusion Algorithms



Centralized Algorithm



(a) (b) (c)

a) . Process 1 asks the coordinator for permission to enter a critical region. Permission is granted

b) . Process 2 then asks permission to enter the same critical region. The coordinator does not reply.

c) . When process 1 exits the critical region, it tells the coordinator, which then replies to 2

Advantages

- Fair algorithm, grants in the order of requests
- The scheme is easy to implement
- Scheme can be used for general resource allocation

Critical Question: When there is no reply, does this mean that the coordinator is "dead" or just busy?

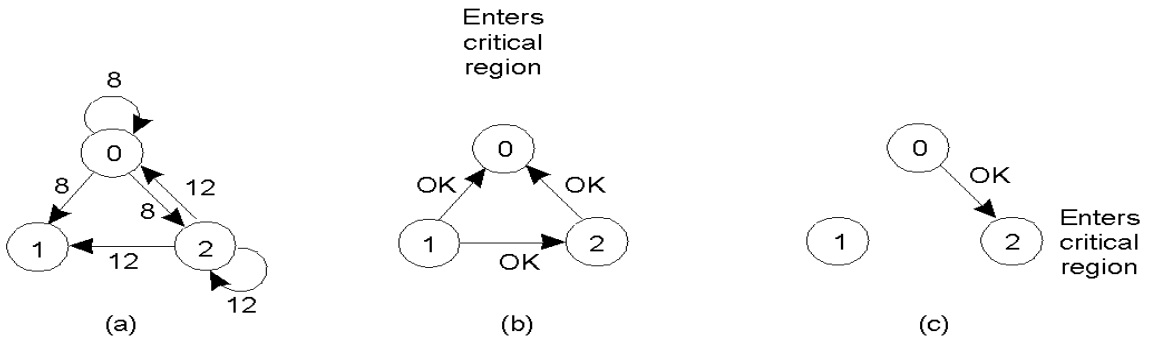
Shortcomings

- Single point of failure. No fault tolerance
- Confusion between No-reply and permission denied
- Performance bottleneck of single coordinator in a large system

Distributed Mutual Exclusion

- Contention-based Mutual Exclusion
  - Timestamp Prioritized Schemes
  - Voting Schemes
- Token-based Mutual Exclusion
  - Ring Structure
  - Tree Structure
  - Broadcast Structure

Timestamp Prioritized Schemes



A> Two processes want to enter the same critical region .

B> Process 0 has the lowest timestamp, so it wins.

C> When process 0 is done, it sends an OK also, so 2 can now enter the critical region

Lamport's "logical clock" is used to generate timestamps. These are the general properties for the **method**:

- The general mechanism is that a process **P[i]** has to send a **REQUEST** (with ID and time stamp) to **all** other processes.
- When a process **P[j]** receives such a **request**, it sends a **REPLY** back.

  When responses are received from all processes, then **P[i]** can enter its Critical Section.
- When **P[i]** exits its critical section, the process sends **RELEASE** messages to all its deferred requests.

  The total message count is 3*(N-1), where N is the number of cooperating processes.

Ricart and Agrawala algorithm

Requesting Site:

- A requesting site *Pi* sends a message *request*(*ts*,*i*) to all sites.

Receiving Site:

- Upon reception of a *request*(*ts*,*i*) message, the receiving site *Pj* will immediately send a timestamped *reply*(*ts*,*j*) message if and only if:
  - *Pj* is not requesting or executing the critical section OR
  - *Pj* is requesting the critical section but sent a request with a higher timestamp than the timestamp of *Pi*

  Otherwise, *Pj* will defer the *reply* message.

Performance

- Number of network messages; 2*(N-1)
- Synchronization Delays: One message propagation delay
- Mutual exclusion: Site *Pi* enters its critical section only after receiving all *reply* messages.
- Progress: Upon exiting the critical section, *Pi* sends all deferred *reply* messages.

Disadvantage of Ricart and Agarwala Algorithm:

- Failure of a node – May result in starvation.
- Solution: This problem can be solved by detecting failure of nodes after some timeout.

Voting schemes

Requestor:

- Send a request to all other processes.
- Enter critical section once REPLY from a **majority** is received
- Broadcast RELEASE upon exit from the critical section.

Other processes:

- REPLY to a request if no REPLY has been sent. Otherwise, hold the request in a queue.
- If a REPLY has been sent, do not send another REPLY till the RELEASE is received.

Possibility of a Deadlock

Consider a situation when each candidate wins one-third of votes…..

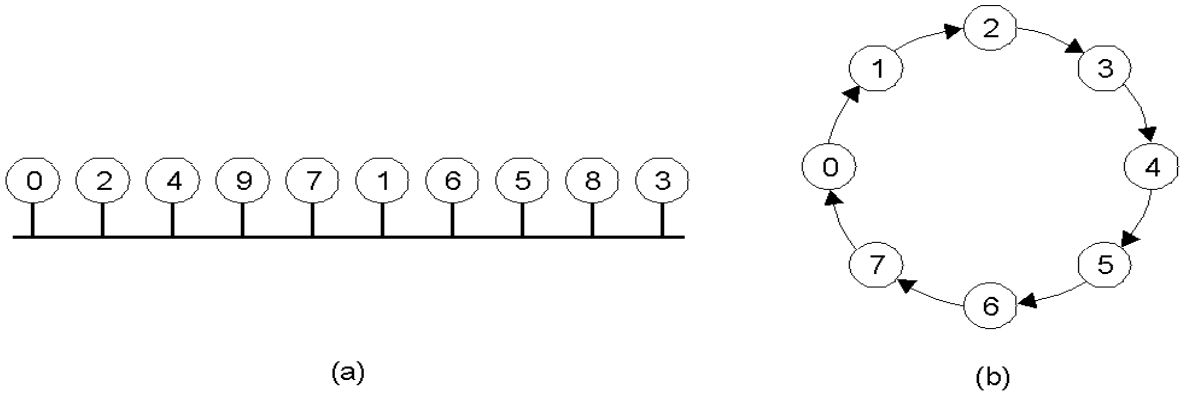
- One of the possible solutions would be :

Any process retrieves its REPLY message by sending an INQUIRY if the requestor is not currently executing in the critical section. The Requestor has to return the vote through a RELINQUISH message.

Token-based Mutual Exclusion

Although contention-based distributed mutual exclusion algorithms can have attractive properties, their messaging overhead is high. An alternative to contention-based algorithms is to use an explicit control token, possession of which grants access to the critical section.

Ring Structure



In software, a logical ring is constructed in which each process is assigned a position in the ring, as shown in the previous Fig.

The ring positions may be allocated in numerical order of network addresses or some other means.

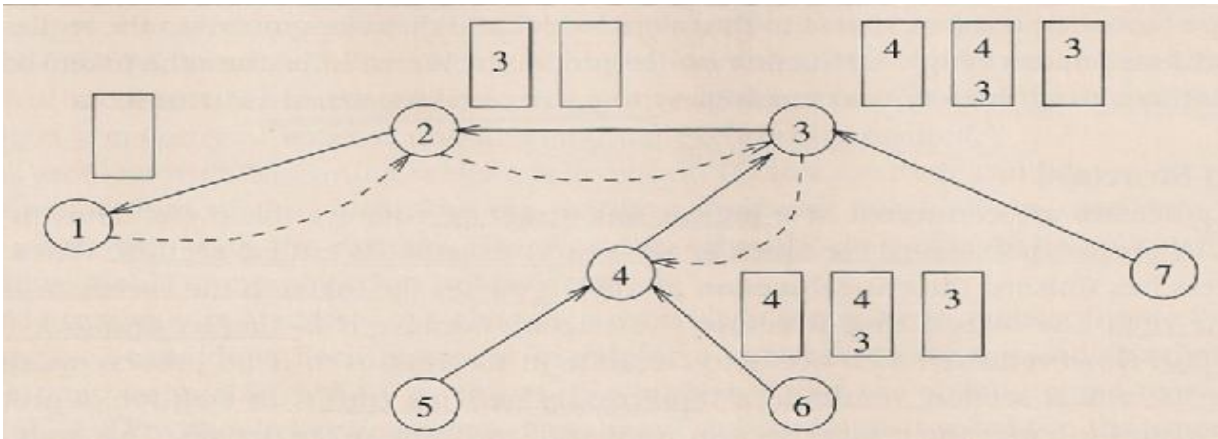
It does not matter what the ordering is. All that matters is that each process knows who is next in line after itself.

- simple, deadlock-free, fair.
- The token circulates even in the absence of any request (unnecessary traffic).
- Long path (O(N)) – the wait for token may be high.
- Works out when the traffic load is high.
- Token can also carry state information.

Tree structure(Raymond's Algorithm)

- Each process explicitly requests for a token and the token is moved only when no process if the process knows of a pending request.
- The root of the tree holds the token to start off.
- The processes are organized in a logical tree structure, each node pointing to its parent.

- Further, each node maintains a FIFO list of token requesting neighbors.
- Each node has a variable *Tokenholder* initialized to *false* for everybody except for the first token holder (token generator).



The processes are organized in a logical tree structure, each node pointing to its parent. Further, each node maintains a FIFO list of token requesting neighbors. Each node has a variable *Tokenholder* initialized to *false* for everybody except for the first token holder (token generator).

- Entry Condition

If not Tokenholder

If the request queue empty

request token from parent;

put itself in request queue;

block self until Tokenholder is true;

Exit condition:

If the request queue is not empty

parent = dequeue(request queue);

send token to parent;

set Tokenholder to false;

if the request queue is still not empty, request token from parent;

Upon receipt of token

Parent = Dequeue(request queue);

if self is the parent

Tokenholder = true

else

send token to the parent;

if the queue is not empty

request token from parent;

Broadcast structure(Suzuki/ Kasami's algorithm).

- Imposing a logical topology like a ring or tree is efficient but also complex because the topology has to be implemented and maintained.
- Group communication:
  - Unaware of the topology
  - More transparent and desirable.

Data Structure:

The token contains

- Token vector *T(.)* – number of completions of the critical section for every process.
- Request queue *Q(.)* – queue of requesting processes.
- Every process (*i*) maintains the following
- *seq_no* – how many times *i* requested critical section.
- *Si(.)* – the highest sequence number from every process *i* heard of.

Entry condition:

- Broadcast a REQUEST message stamped with *seq_no*.
- Enter critical section after receiving token

Exit Condition:

- Update the token vector *T* by setting *T(i)* to $S_i(i)$.
- If process *k* is not in request queue *Q* and there are pending requests from *k* ($S_i(k)>T(k)$), append process *k* to *Q*.
- If *Q* is non-empty, remove the first entry from *Q* and send the token to the process indicated by the top entry.

Comparison of the Mutual exclusion algorithms

Algorithm	Messages per entry/exit	Delay before entry (in message times)	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

None are perfect – they all have their problems!

Distributed Deadlock Detection

- Assumptions:
    a. System has only reusable resources
    b. Only exclusive access to resources
    c. Only one copy of each resource
    d. States of a process: running or blocked
    e. Running state: process has all the resources
    f. Blocked state: waiting on one or more resource

Resource vs Communication Deadlocks

- **Resource Deadlocks**
    - A process needs multiple resources for an activity.
    - Deadlock occurs if each process in a set request resources

held by another process in the same set, and it must receive

all the requested resources to move further.

- **Communication Deadlocks**
    - Processes wait to communicate with other processes in a set.

- Each process in the set is waiting on another process's message, and no process in the set initiates a message until it receives a message for which it is waiting.

Deadlock Handling Strategies

- Deadlock Prevention: difficult
- Deadlock Avoidance: before allocation, check for possible deadlocks.
  - Difficult as it needs global state info in each site (that handles resources).
- Deadlock Detection: Find cycles. Focus of discussion.
- Deadlock detection algorithms must satisfy 2 conditions:
  - No undetected deadlocks.
  - No false deadlocks.

Distributed Deadlocks

- **Centralized Control**
  - A *control site* constructs wait-for graphs (WFGs) and checks for directed cycles.
  - WFG can be maintained continuously (or) built on-demand by requesting WFGs from individual sites.
- **Distributed Control**
  - WFG is spread over different sites.Any site can initiate the deadlock detection process.
- **Hierarchical Control**
  - Sites are arranged in a hierarchy.
  - A site checks for cycles only in descendents.

Centralized Algorithms

- **Ho-Ramamurthy 2-phase Algorithm**
  - Each site maintains a status table of all processes initiated at that site: includes all resources locked & all resources being waited on.
  - Controller requests (periodically) the status table from each site.
  - Controller then constructs WFG from these tables, searches for cycle(s).
  - If no cycles, no deadlocks.

- Otherwise, (cycle exists): Request for state tables again.
- Construct WFG based *only* on common transactions in the 2 tables.
- If the same cycle is detected again, system is in deadlock.

- **Ho-Ramamoorthy 1-phase Algorithm**
  - Each site maintains 2 status tables: *resource status* table and *process status* table.
  - Resource table: transactions that have locked or are waiting for resources.
  - Process table: resources locked by or waited on by transactions.
  - Controller periodically collects these tables from each site.
  - Constructs a WFG from transactions common to both the tables.
  - No cycle, no deadlocks.

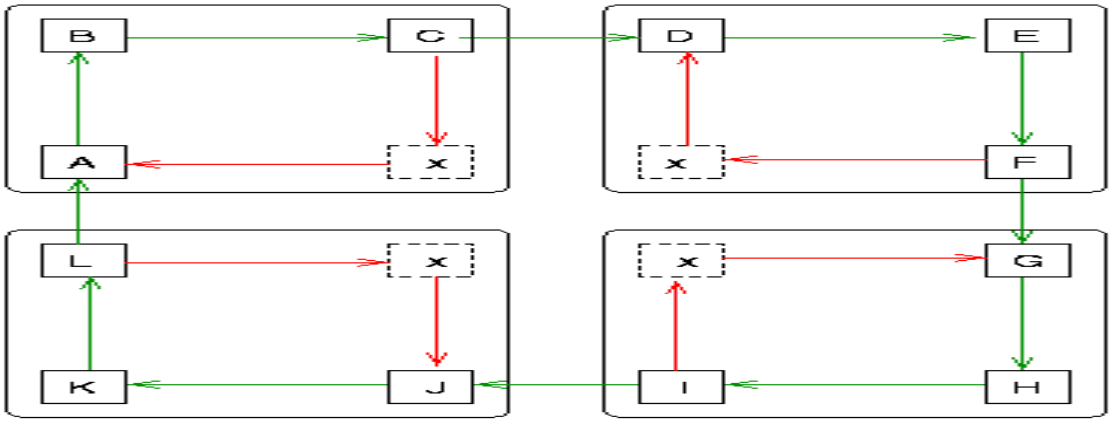
    A cycle means a deadlock.

Distributed Algorithms

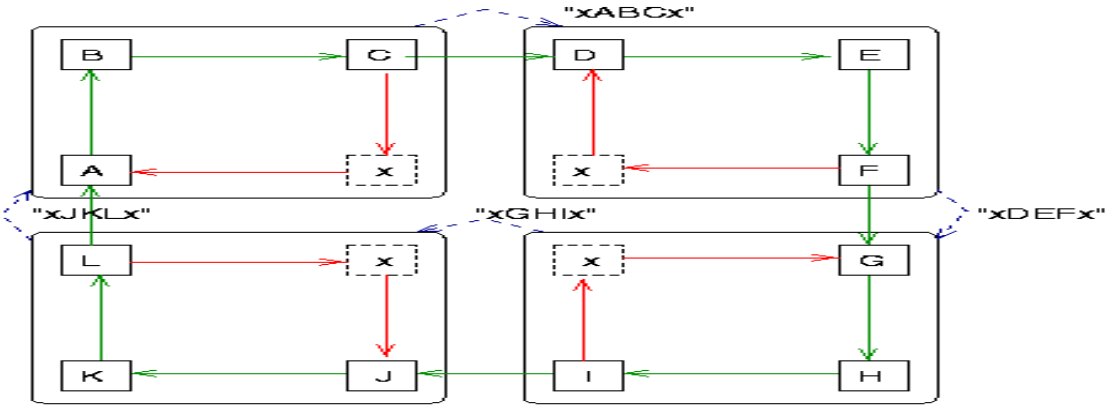
- Path-pushing: resource dependency information disseminated through designated paths (in the graph) [Examples : Menasce-Muntz & Obermarck]
- Edge-chasing: special messages or probes circulated along edges of WFG. Deadlock exists if the probe is received back by the initiator. [Examples :CMH for AND Model , Sinha-Natarajan]
- Diffusion computation: queries on status sent to process in WFG. [Examples :CMH for OR Model, Chandy-Herman]

Path-pushing

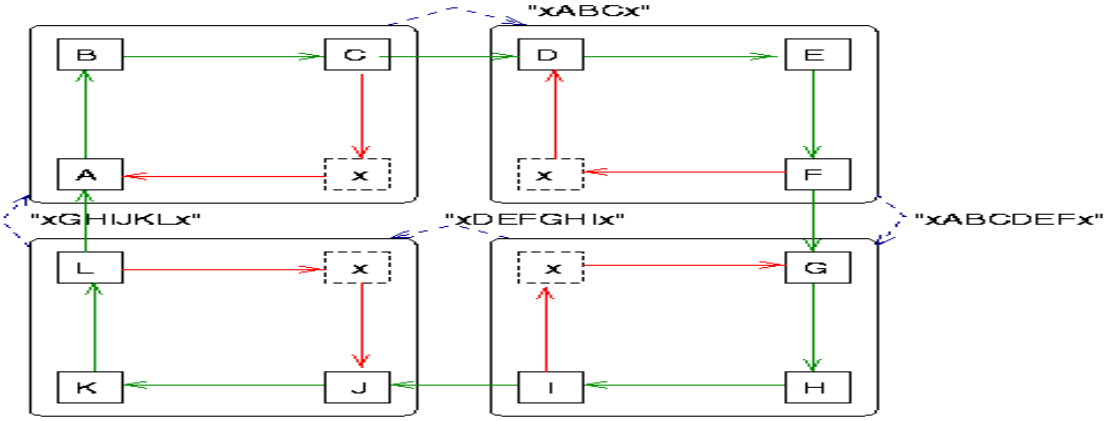
- **Obermarck's Algorithm (AND model)**
  - Path Propagation Based Algorithm
  - Based on a database model using transaction processing
  - Sites which detect a cycle in their partial WFG views convey the paths discovered to members of the (totally ordered) transaction
  - Algorithm can detect ***phantoms*** due to its asynchronous snapshot method
- **Obermark's Algorithm Example Intial State**


B
C
D
E
A
x
x
F
L
x
x
G
K
J
I
H

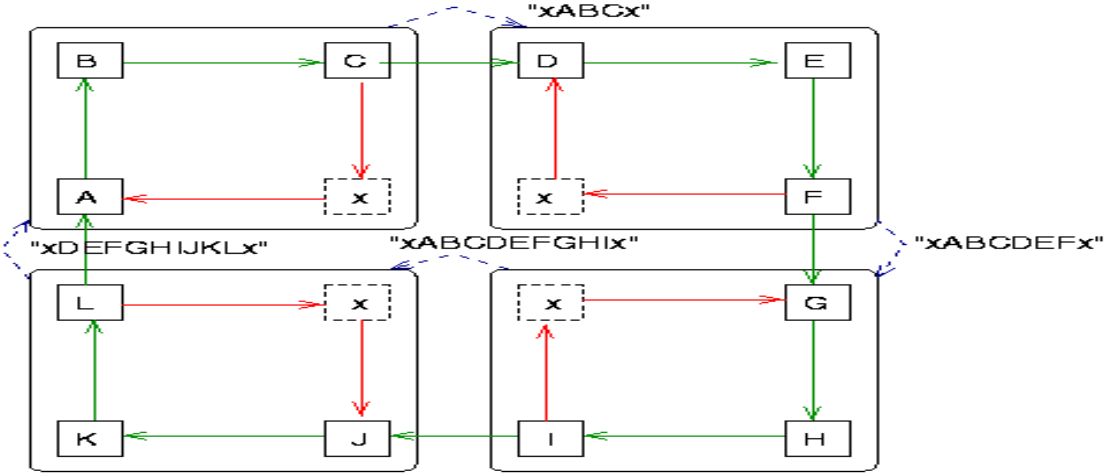
Iteration 1



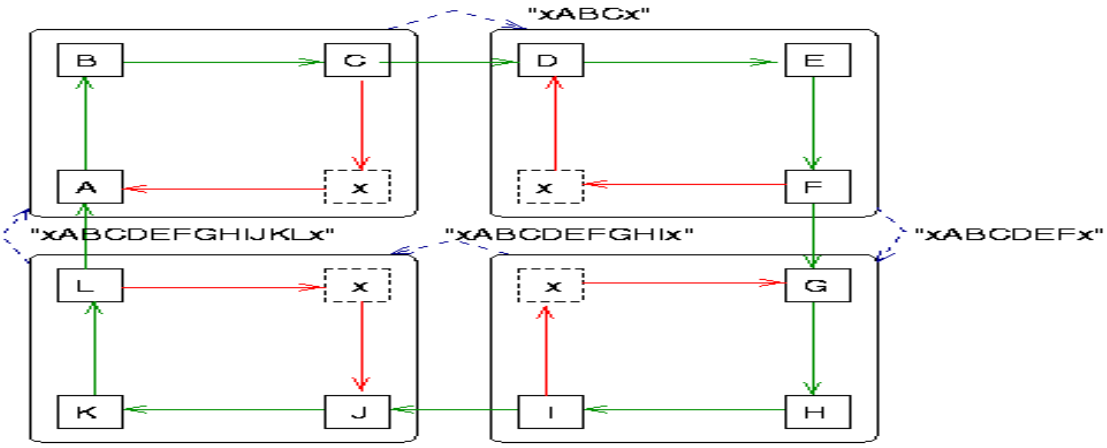
Iteration 2



Iteration 3



Iteration 4



Edge-Chasing Algorithm

- **Chandy-Misra-Haas's Algorithm (AND MODEL)**:
  - A probe(i, j, k) is used by a deadlock detection process Pi. This probe is sent by the home site of Pj to Pk.
  - This probe message is circulated via the edges of the graph. Probe returning to Pi implies deadlock detection.
  - Terms used:
    - Pj is *dependent* on Pk, if a sequence of Pj, Pi1,.., Pim, Pk exists.
    - Pj is *locally dependent* on Pk, if above condition + Pj,Pk on same site.
    - Each process maintains an array *dependenti: dependenti(j)* is true if Pi knows that Pj is dependent on it. (initially set to false for all i & j).
- **Sending the probe:**

  if Pi is locally dependent on itself then deadlock.

  else for all Pj and Pk such that

  (a) Pi is locally dependent upon Pj, and

  (b) Pj is waiting on Pk, and

  (c) Pj and Pk are on different sites, send probe(i,j,k) to the home site of Pk.

- **Receiving the probe:**

  if (d) Pk is blocked, and

(e) *dependentk(i)* is false, and

(f) Pk has not replied to all requests of Pj,

then begin

dependentk(i) := true;

if k = i then Pi is deadlocked

else ...

Receiving the probe:

…….

else for all Pm and Pn such that

(a') Pk is locally dependent upon Pm, and

(b') Pm is waiting on Pn, and

(c') Pm and Pn are on different sites, send probe(i,m,n)

to the home site of Pn.

end.

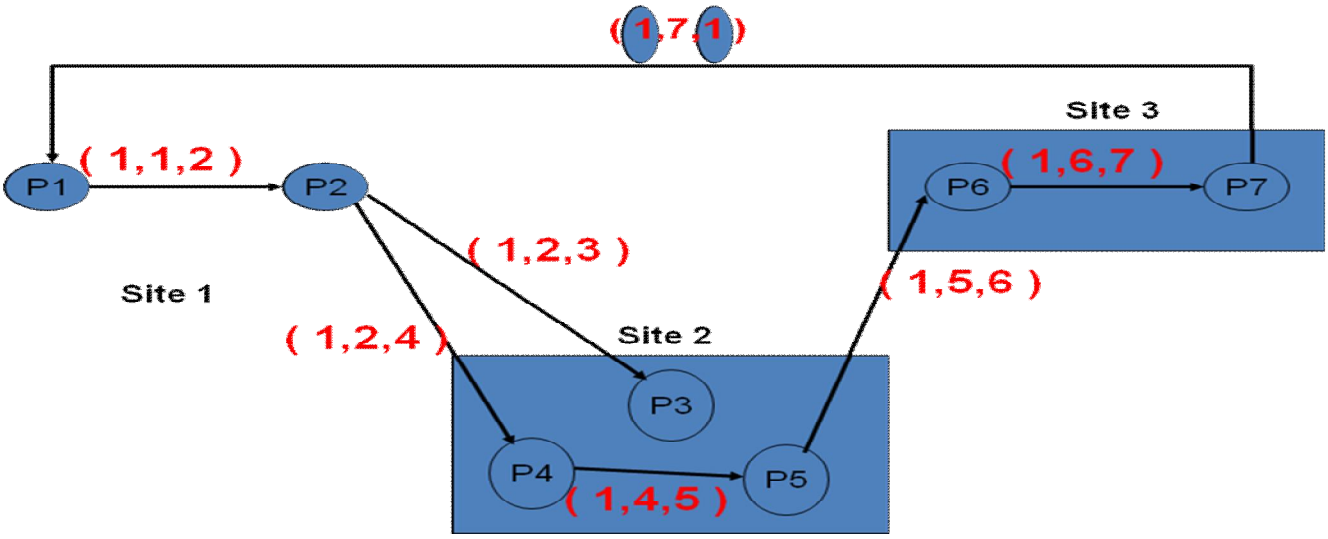
Performance:

For a deadlock that spans m processes over n sites, m(n-1)/2 messages are needed.

Size of the message 3 words.

Delay in deadlock detection O(n).

C-M-H Algorithm: Example



P1 initiates Deadlock Detection by sending Probe Message (1,1,2) to P2

Diffusion-based Algorithm (CMH Algorithm for OR Model)

Initiation by a blocked process Pi:

send query(i,i,j) to all processes Pj in the dependent set DSi of Pi;

num(i) := |DSi|; waiti(i) := true;

Blocked process Pk receiving query(i,j,k):

if this is *engaging* query for process Pk /* first query from Pi */

then send query(i,k,m) to all Pm in DSk;

numk(i) := |DSk|; waitk(i) := true;

else if waitk(i) then send a reply(i,k,j) to Pj.

Process Pk receiving reply(i,j,k)

if waitk(i) then

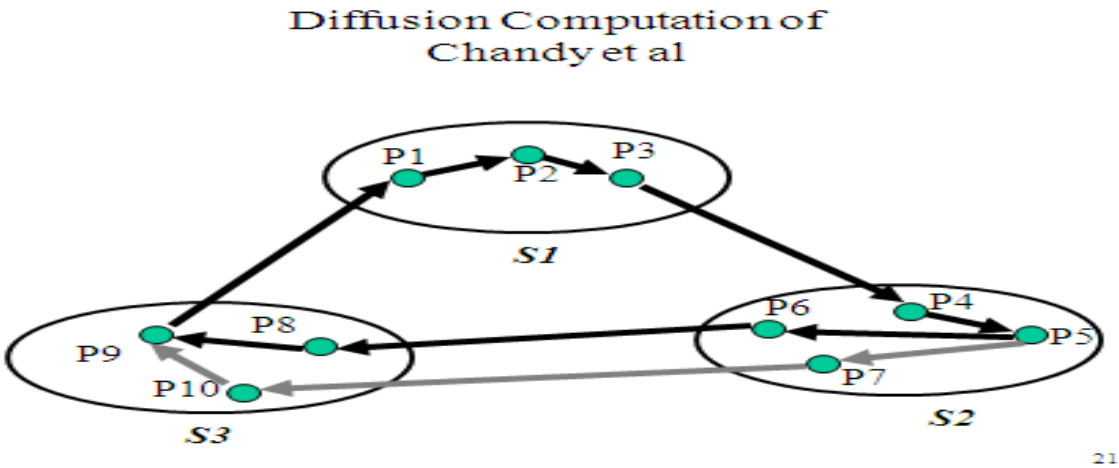
numk(i) := numk(i) - 1;

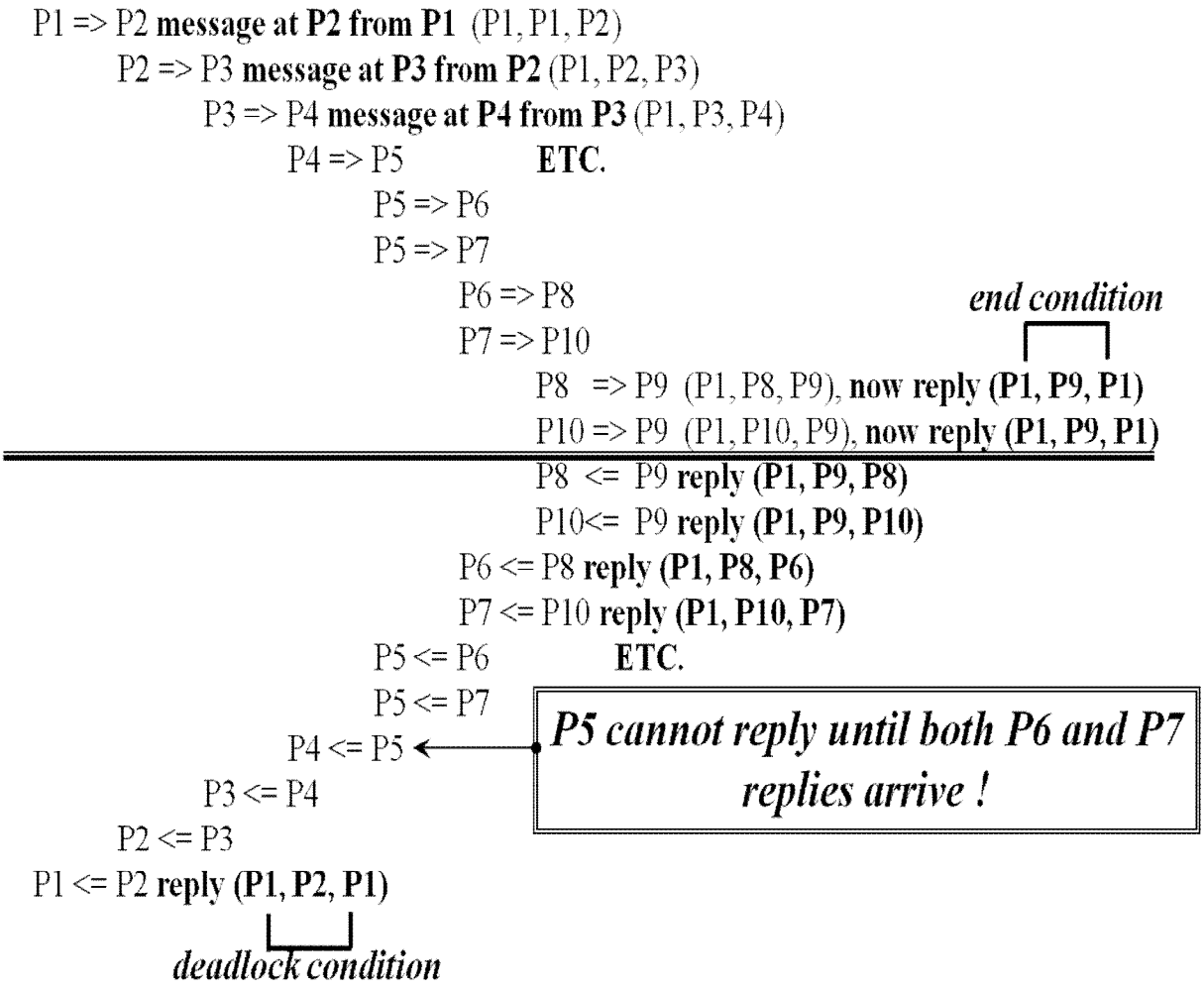
if numk(i) = 0 then

if i = k then declare a deadlock.

else send reply(i, k, m) to Pm, which sent the engaging query.

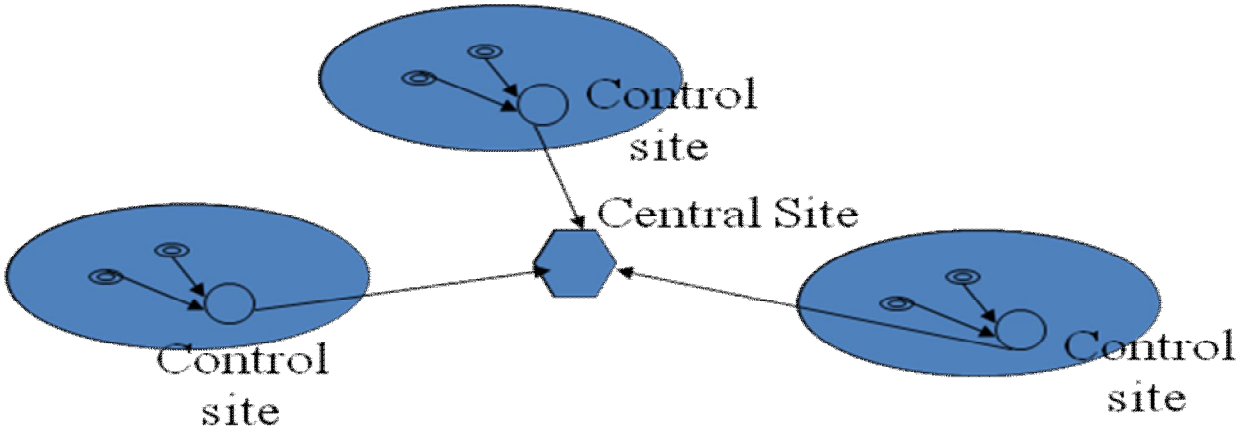
Diffusion Algorithm: Example





Hierarchical Deadlock Detection

- Follows Ho-Ramamoorthy's 1-phase algorithm. More than 1 control site organized in hierarchical manner.
- Each control site applies 1-phase algorithm to detect (intracluster) deadlocks.
- Central site collects info from control sites, applies 1-phase algorithm to detect intracluster deadlocks.



Persistence & Resolution

- Deadlock persistence:
  - Average time a deadlock exists before it is resolved.
- Implication of persistence:
  - Resources unavailable for this period: affects utilization
  - Processes wait for this period unproductively: affects response time.
- Deadlock resolution:
  - Aborting at least one process/request involved in the deadlock.
  - Efficient resolution of deadlock requires knowledge of all processes and resources.
  - If every process detects a deadlock and tries to resolve it independently -> highly inefficient ! Several processes might be aborted.

Deadlock Resolution

- Priorities for processes/transactions can be useful for resolution.
  - Consider priorities introduced in Obermarck's algorithm.
  - Highest priority process initiates and detects deadlock (initiations by lower priority ones are suppressed).
  - When deadlock is detected, lowest priority process(es) can be aborted to resolve the deadlock.
- After identifying the processes/requests to be aborted,

- ❑ All resources held by the victims must be released. State of released resources restored to previous states. Released resources granted to deadlocked processes.
- ❑ All deadlock detection information concerning the victims must be removed at all the sites.

Agreement Protocols

- When distributed systems engage in cooperative efforts like enforcing distributed mutual exclusion algorithms, processor failure can become a critical factor
- Processors may fail in various ways, and their failure modes and communication interfaces are central to the ability of healthy processors to detect and respond to such failures

The System Model

- The are *n* processors in the system and at most *m* of them can be faulty
- The processors can directly communicate with others processors via messages (fully connected system)
- A receiver computation always knows the identity of a sending computation
- The communication system is pipelined and reliable

Faulty Processors

- May fail in various ways
  - Drop out of sight completely
  - Start sending spurious messages
  - Start to lie in its messages (behave maliciously)
  - Send only occasional messages (fail to reply when expected to)
- May believe themselves to be healthy
- Are not know to be faulty initially by non-faulty processors

Communication Requirements

- Synchronous model communication is assumed in this section:
  - Healthy processors receive, process and reply to messages in a lockstep manner
  - The receive, process, reply sequence is called a ***round***
  - In the synch-comm model, processes know what messages they expect to receive in a round

- The synch model is critical to agreement protocols, and the agreement problem is not solvable in an asynchronous system

Processor Failures

- Crash fault
  - Abrupt halt, never resumes operation
- Omission fault
  - Processor "omits" to send required messages to some other processors
- Malicious fault
  - Processor behaves randomly and arbitrarily
  - Known as *Byzantine faults*

Authenticated vs. Non-Authenticated Messages

- Authenticated messages (also called *signed* messages)
  - assure the receiver of correct identification of the sender
  - assure the receiver the the message content was not modified in transit
- Non-authenticated messages (also called ***oral*** messages)
  - are subject to intermediate manipulation
  - may lie about their origin
- To be generally useful, agreement protocols must be able to handle non-authenticated messages
- The classification of agreement problems include:
  - The Byzantine agreement problem
  - The consensus problem
  - the interactive consistency problem

Agreement Problems

Problem	**Who initiates value**	**Final Agreement**
Byzantine Agreement	One Processor	Single Value

Consensus	All Processors	Single Value
Interactive Consistency	All Processors	A Vector of Values

- **Byzantine Agreement**
  - One processor broadcasts a value to all other processors
  - All non-faulty processors agree on this value, faulty processors may agree on any (or no) value

- **Consensus**
  - Each processor broadcasts a value to all other processors
  - All non-faulty processors agree on one common value from among those sent out. Faulty processors may agree on any (or no) value
- **Interactive Consistency**
  - Each processor broadcasts a value to all other processors
  - All non-faulty processors agree on the same vector of values such that $\boldsymbol{v_i}$ is the initial broadcast value of non-faulty $\boldsymbol{processor_i}$. Faulty processors may agree on any (or no) value.

UNIT – 3

UNIT III DISTRIBUTED RESOURCE MANAGEMENT 9

Distributed File Systems – Design Issues - Distributed Shared Memory – Algorithms for Implementing Distributed Shared memory–Issues in Load Distributing – Scheduling Algorithms – Synchronous and Asynchronous Check Pointing and Recovery – Fault Tolerance – Two-Phase Commit Protocol – Nonblocking Commit Protocol – Security and Protection.

Distributed Resource Management

- Good resource allocation schemes are needed to fully utilize the computing capacity of the DS
- Distributed scheduler is a resource management component of a DOS
- It focuses on judiciously and transparently redistributing the load of the system among the computers
- Target is to maximize the overall performance of the system
- More suitable for DS based on LANs

Motivation

- A locally distributed system consists of a collection of autonomous computers connected by a local area communication network
- Users submit tasks at their host computers for processing
- Load distributed is required in such environment because of random arrival of tasks and their random CPU service time
- There is a possibility that several computers are heavily loaded and others are idle of lightly loaded
- If the load is heavier on some systems or if some processors execute tasks at a slower rate than others, this situation will occur often

DISTRIBUTED FILE SYSTEMS

- A **Distributed File System** (DFS) is simply a classical model of a file system (as discussed before) distributed across multiple machines. The purpose is to promote sharing of dispersed files.
- This is an area of active research interest today.
- The resources on a particular machine are **local** to itself. Resources on other machines are **remote**.
- A file system provides a service for clients. The server interface is the normal set of file operations: create, read, etc. on files.
-

Clients, servers, and storage are dispersed across machines. Configuration and implementation may vary -

a) Servers may run on dedicated machines, OR
b) Servers and clients can be on the same machines.
c) The OS itself can be distributed (with the file system a part of that distribution.
d) A distribution layer can be interposed between a conventional OS and the file system.

Clients should view a DFS the same way they would a centralized FS; the distribution is hidden at a lower level.

Performance is concerned with throughput and response time.

ISSUES:

Naming and Transparency

Naming is the mapping between logical and physical objects.

- Example: A user filename maps to <cylinder, sector>.
- In a conventional file system, it's understood where the file actually resides; the system and disk are known.

- In a **transparent** DFS, the location of a file, somewhere in the network, is hidden.
- **File replication** means multiple copies of a file; mapping returns a SET of locations for the replicas.

Location transparency -

a) The name of a file does not reveal any hint of the file's physical storage location.
b) File name still denotes a specific, although hidden, set of physical disk blocks.
c) This is a convenient way to share data.
d) Can expose correspondence between component units and machines.

Location independence -

- The name of a file doesn't need to be changed when the file's physical storage location changes. Dynamic, one-to-many mapping.
- Better file abstraction.
- Promotes sharing the storage space itself.
- Separates the naming hierarchy from the storage devices hierarchy.

Most DFSs today:

- Support location transparent systems.
- Do NOT support **migration**; (automatic movement of a file from machine to machine.)
- Files are permanently associated with specific disk blocks.

The ANDREW DFS AS AN EXAMPLE:

- Is location independent.
- Supports file mobility.
- Separation of FS and OS allows for disk-less systems. These have lower cost and convenient system upgrades. The performance is not as good.

NAMING SCHEMES:

- There are three main approaches to naming files:

1. Files are named with a **combination** of host and local name.

    - This guarantees a unique name. NOT location transparent NOR location independent.
    - Same naming works on local and remote files. The DFS is a loose collection of independent file systems.

2. Remote directories are **mounted** to local directories.

    - So a local system seems to have a coherent directory structure.
    - The remote directories must be explicitly mounted. The files are location independent.
    - SUN NFS is a good example of this technique.

3. A **single global name structure** spans all the files in the system.

    - The DFS is built the same way as a local filesystem. Location independent.

IMPLEMENTATION TECHNIQUES:

Can Map directories or larger aggregates rather than individual files.

- A **non-transparent** mapping technique:

name ----> < system, disk, cylinder, sector >

- A **transparent** mapping technique:

  name ----> file_identifier ----> < system, disk, cylinder, sector >

- So when changing the physical location of a file, only the file identifier need be modified. This identifier must be "unique" in the universe.

Remote File Access

CACHING

Reduce network traffic by retaining recently accessed disk blocks in a cache, so that repeated accesses to the same information can be handled locally.

If required data is not already cached, a copy of data is brought from the server to the user.

Perform accesses on the cached copy.

Files are identified with one master copy residing at the server machine,Copies of (parts of) the file are scattered in different caches.

Cache Consistency Problem -- Keeping the cached copies consistent with the master file.

A remote service ((RPC) has these characteristic steps:

a) The client makes a request for file access.
b) The request is passed to the server in message format.
c) The server makes the file access.
d) Return messages bring the result back to the client.

This is equivalent to performing a disk access for each request.

CACHE LOCATION:

Caching is a mechanism for maintaining disk data on the local machine. This data can be kept in the local memory or in the local disk. Caching can be advantageous both for read ahead and read again.

The cost of getting data from a cache is a few HUNDRED instructions; disk accesses cost THOUSANDS of instructions.

The master copy of a file doesn't move, but caches contain replicas of portions of the file. Caching behaves just like "networked virtual memory".

What should be cached? << blocks <---> files >>.

Bigger sizes give a better hit rate;

Smaller give better transfer times.

Caching on disk gives:

— Better reliability.

Caching in memory gives:

— The possibility of diskless work stations,

— Greater speed,

Since the server cache is in memory, it allows the use of only one mechanism.

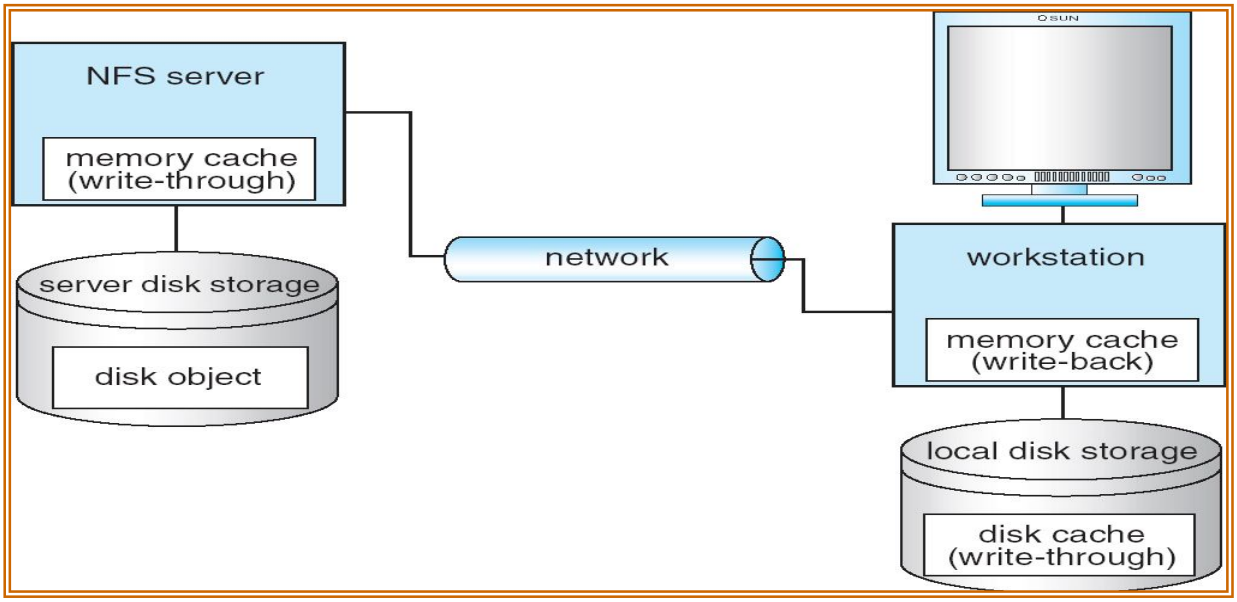
CACHE UPDATE POLICY:

A **write through** cache has good reliability. But the user must wait for writes to get to the server. Used by NFS.

Delayed write - write requests complete more rapidly. Data may be written over the previous cache write, saving a remote write. Poor reliability on a crash.

- Flush sometime later tries to regulate the frequency of writes.

- Write on close delays the write even longer.
- Which would you use for a database file? For file editing?



CACHE CONSISTENCY:

The basic issue is, how to determine that the client-cached data is consistent with what's on the server.

- **Client - initiated approach -**

The client asks the server if the cached data is OK. What should be the frequency of "asking"? On file open, at fixed time interval, ...?

- **Server - initiated approach -**

Possibilities: A and B both have the same file open. When A closes the file, B "discards" its copy. Then B must start over.

The server is notified on every open. If a file is opened for writing, then disable caching by other clients for that file.

Get read/write permission for each block; then disable caching only for particular blocks.

COMPARISON OF CACHING AND REMOTE SERVICE:

- Many remote accesses can be handled by a local cache. There's a great deal of locality of reference in file accesses. Servers can be accessed only occasionally rather than for each access.
- Caching causes data to be moved in a few big chunks rather than in many smaller pieces; this leads to considerable efficiency for the network.
- Cache consistency is the major problem with caching. When there are infrequent writes, caching is a win. In environments with many writes, the work required to maintain consistency overwhelms caching advantages.

- Caching requires a whole separate mechanism to support acquiring and storage of large amounts of data. Remote service merely does what's required for each call. As such, caching introduces an extra layer and mechanism and is more complicated than remote service.

FILE REPLICATION:

- Duplicating files on multiple machines improves availability and performance.
- Placed on failure-independent machines (they won't fail together).

  Replication management should be "location-opaque".

- The main problem is consistency - when one copy changes, how do other copies reflect that change? Often there is a tradeoff: consistency versus availability and performance.

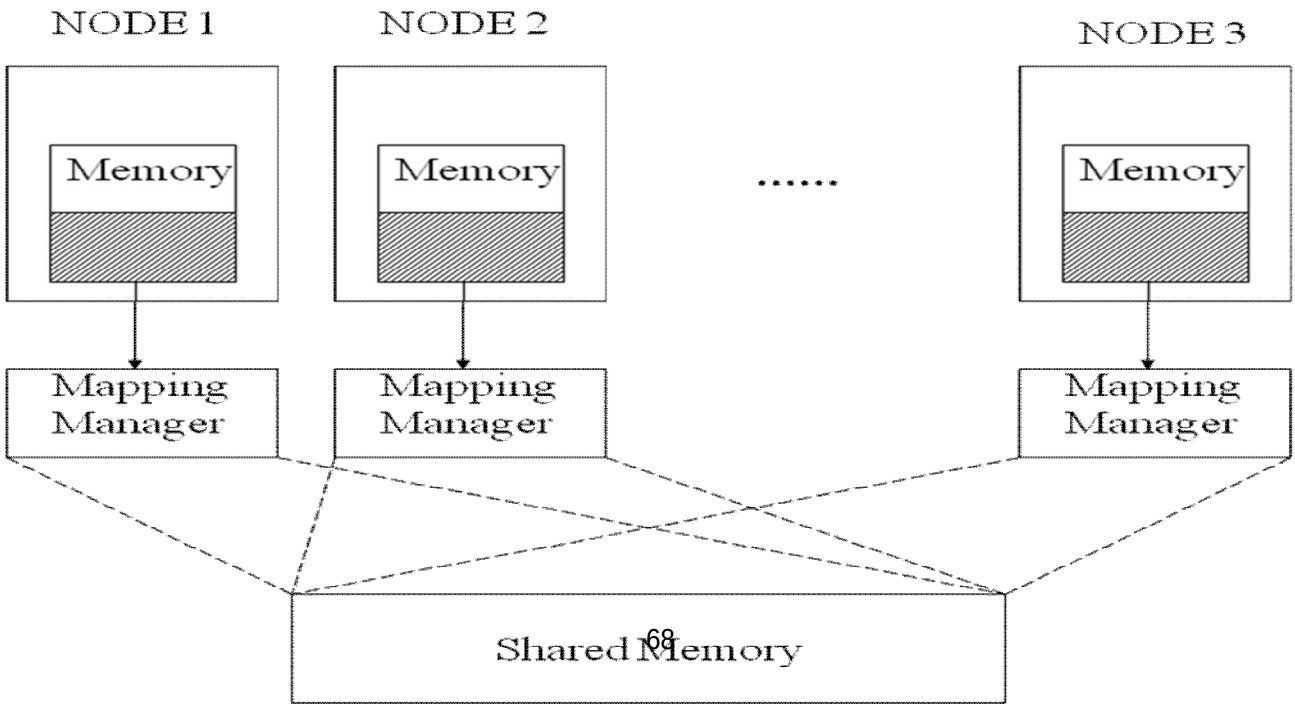
Example:

"Demand replication" is like whole-file caching; reading a file causes it to be cached locally. Updates are done only on the primary file at which time all other copies are invalidated.

- Atomic and serialized invalidation isn't guaranteed (message could get lost / machine could crash.)

Distributed Shared Memory

- What
  - The distributed shared memory (DSM) implements the shared memory model in distributed systems, which have no physical shared memory
  - The shared memory model provides a virtual address space shared between all nodes
  - The overcome the high cost of communication in distributed systems, DSM systems move data to the location of access
- How:
  - Data moves between main memory and secondary memory (within a node) and between main memories of different nodes
  - Each data object is owned by a node
    - Initial owner is the node that created object
    - Ownership can change as object moves from node to node
  - When a process accesses data in the shared address space, the mapping manager maps shared memory address to physical memory (local or remote)



Advantages of distributed shared memory (DSM)

- Data sharing is implicit, hiding data movement (as opposed to 'Send'/'Receive' in message passing model)
- Passing data structures containing pointers is easier (in message passing model data moves between different address spaces)
- Moving entire object to user takes advantage of locality difference
- Less expensive to build than tightly coupled multiprocessor system: off-the-shelf hardware, no expensive interface to shared physical memory
- Very large total physical memory for all nodes: Large programs can run more efficiently
- No serial access to common bus for shared physical memory like in multiprocessor systems
- Programs written for shared memory multiprocessors can be run on DSM systems with minimum changes

Algorithms for implementing DSM

- Issues
    - How to keep track of the location of remote data
    - How to minimize communication overhead when accessing remote data
    - How to access concurrently remote data at several nodes

1. The Central Server Algorithm

    - Central server maintains all shared data
        - Read request: returns data item
        - Write request: updates data and returns acknowledgement message
    - Implementation
        - A timeout is used to resend a request if acknowledgment fails

    - Associated sequence numbers can be used to detect duplicate write requests
    - If an application's request to access shared data fails repeatedly, a failure condition is sent to the application
- Issues: performance and reliability
- Possible solutions
    - Partition shared data between several servers
    - Use a mapping function to distribute/locate data

2. The Migration Algorithm

- Operation
    - Ship (migrate) entire data object (page, block) containing data item to requesting location
    - Allow only one node to access a shared data at a time
- Advantages
    - Takes advantage of the locality of reference
    - DSM can be integrated with VM at each node
        - Make DSM page multiple of VM page size
        - A locally held shared memory can be mapped into the VM page address space
        - If page not local, fault-handler migrates page and removes it from address space at remote node
- To locate a remote data object:
    - Use a location server
    - Maintain hints at each node
    - Broadcast query
- Issues
    - Only one node can access a data object at a time

- Thrashing can occur: to minimize it, set minimum time data object resides at a node
-

3. The Read-Replication Algorithm
   - Replicates data objects to multiple nodes
   - DSM keeps track of location of data objects
   - Multiple nodes can have read access or one node write access (multiple readers-one writer protocol)
   - After a write, all copies are invalidated or updated
   - DSM has to keep track of locations of all copies of data objects. Examples of implementations:
     - IVY: owner node of data object knows all nodes that have copies
     - PLUS: distributed linked-list tracks all nodes that have copies
   - Advantage
     - The read-replication can lead to substantial performance improvements if the ratio of reads to writes is large
     -

4. The Full–Replication Algorithm

   - Extension of read-replication algorithm: multiple nodes can read and multiple nodes can write (multiple-readers, multiple-writers protocol)
   - Issue: consistency of data for multiple writers
   - Solution: use of gap-free sequencer
     - All writes sent to sequencer
     - Sequencer assigns sequence number and sends write request to all sites that have copies
     - Each node performs writes according to sequence numbers
     - A gap in sequence numbers indicates a missing write request: node asks for retransmission of missing write requests

Issues in Load Distributing

- Load
  - Resource queue lengths and particularly the CPU queue length are good indicators of load
  - Measuring the CPU queue length is fairly simple and carries little overhead
  - CPU queue length does not always tell the correct situation as the jobs may differ in types
  - Another load measuring criterion is the processor utilization
  - Requires a background process that monitors CPU utilization continuously and imposes more overhead
  - Used in most of the load balancing algorithms

Classification of LDA

- Basic function is to transfer load from heavily loaded systems to idle or lightly loaded systems

- These algorithms can be classified as :

  - Static
    - decisions are hard-wired in the algorithm using a prior knowledge of the system
  - Dynamic
    - use system state information to make load distributing decisions
  - Adaptive
    - special case of dynamic algorithms in that they adapt their activities by dynamically changing the parameters of the algorithm to suit the changing system state

- Load Balancing vs. Load sharing

  - Load sharing algorithms strive to reduce the possibility for a system to go to a state in which it lies idle while at the same time tasks contend service at another, by transferring tasks to lightly loaded nodes
  - Load balancing algorithms try to equalize loads at al computers
  - Because a load balancing algorithm transfers tasks at higher rate than a load sharing algorithm, the higher overhead incurred by the load balancing algorithm may outweigh this potential performance improvement

Load Distributing Algorithms

- Sender-Initiated Algorithms
- Receiver-Initiated Algorithms
- Symmetrically Initiated Algorithms
- Adaptive Algorithms

Sender-Initiated Algorithms

- Activity is initiated by an overloaded node (sender)
- A task is sent to an underloaded node (receiver)

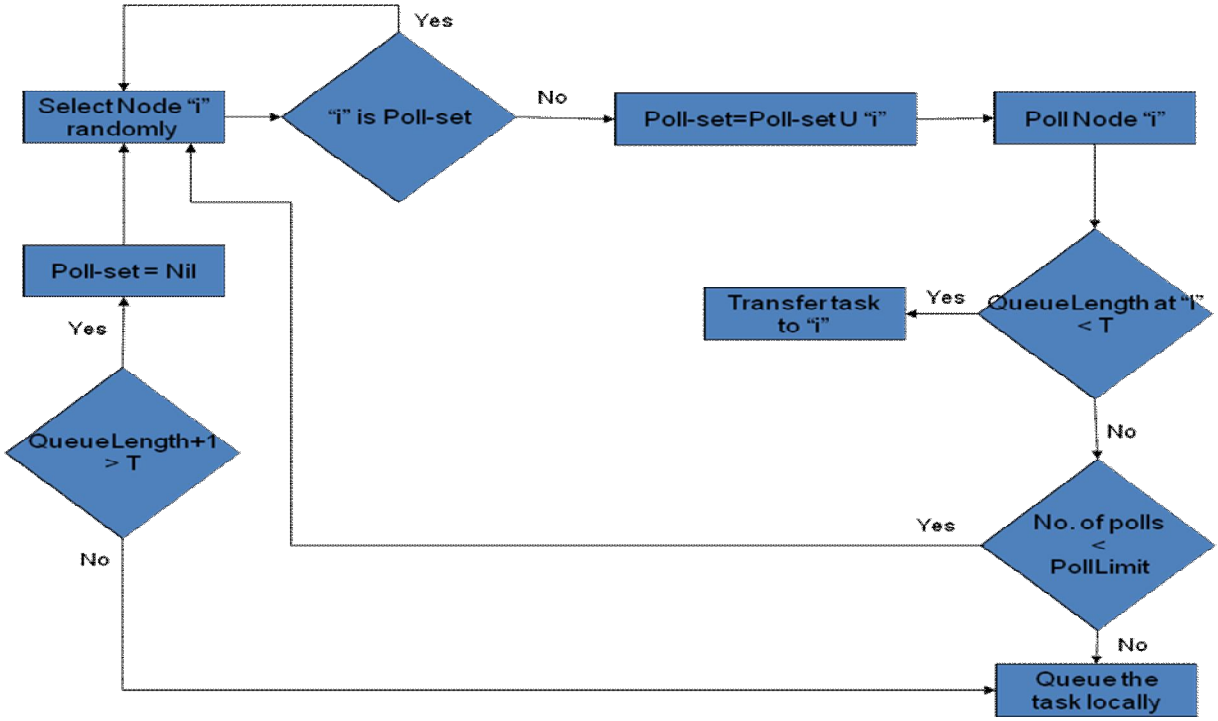
  - Transfer Policy
    - A node is identified as a sender if a new task originating at the node makes the queue length exceed a threshold T.

  - Selection Policy
    - Only new arrived tasks are considered for transfer

– Location Policy
  - Random: dynamic location policy, no prior information exchange
  - Threshold: polling a node (selected at random) to find a receiver
  - Shortest: a group of nodes are polled to determine their queue

– Information Policy
  - A demand-driven type

– Stability
  - Location policies adopted cause system instability at high loads



Receiver-Initiated Algorithms

- Initiated from an underloaded node (receiver) to obtain a task from an overloaded node (sender)
-

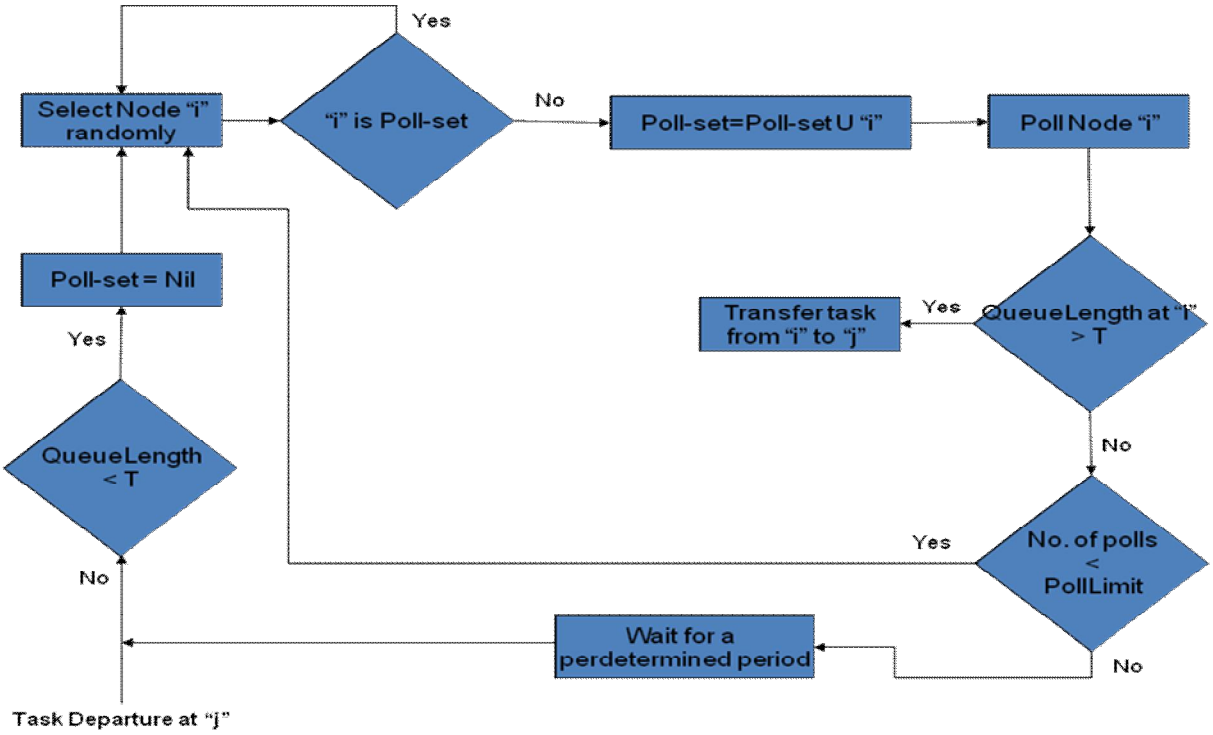
- Transfer Policy
  - Triggered when a task departs

- Selection Policy
  - Same as the previous

- Location Policy
  - A node selected at random is polled to determine if transferring a task from it would place its queue length below the threshold level, if not, the polled node transfers a task.

- Information Policy
  - A demand-driven type

- Stability
  - Do not cause system instability in high system load, however, in low load it spare CPU cycles
  - Most transfers are preemptive and therefore expensive



Symmetrically Initiated Algorithms

- Both senders and receivers search for receiver and senders, respectively, for task transfer.
- The Above-Average Algorithm
  - Transfer Policy
    - Thresholds are equidistant from the node's estimate of the average load across all node.
  - Location Policy
    - Sender-initiated component: Timeout messages TooHigh, TooLow, Accept, AwaitingTask, ChangeAverage
    - Receiver-initiated component: Timeout messages TooLow, LooHigh, Accept, AwaitingTask, ChangeAverage
  - Selection Policy
    - Similar to both the earlier algorithms

- Information Policy
    - A demand-driven type but the acceptable range can be increased/decreased by each node individually.

Adaptive Algorithms

- A Stable Symmetrically Initiated Algorithm
    - Utilizes the information gathered during polling to classify the nodes in the system as either Sender, Receiver or OK.
    - The knowledge concerning the state of nodes is maintained by a data structure at each node, comprised of a senders list, a receivers list, and an OK list.
    - Initially, each node assumes that every other node is a receiver.
    - Transfer Policy
        - Triggers when a new task originates or when a task departs.
        - Makes use of two threshold values, i.e. Lower (LT) and Upper (UT)
    - Location Policy
        - Sender-initiated component: Polls the node at the head of receiver's list
        - Receiver-initiated component: Polling in three order
            - Head-Tail (senders list), Tail-Head (OK list), Tail-Head (receivers list)
    - Selection Policy: Newly arrived task (SI), other approached (RI)
    - Information Policy: A demand-driven type

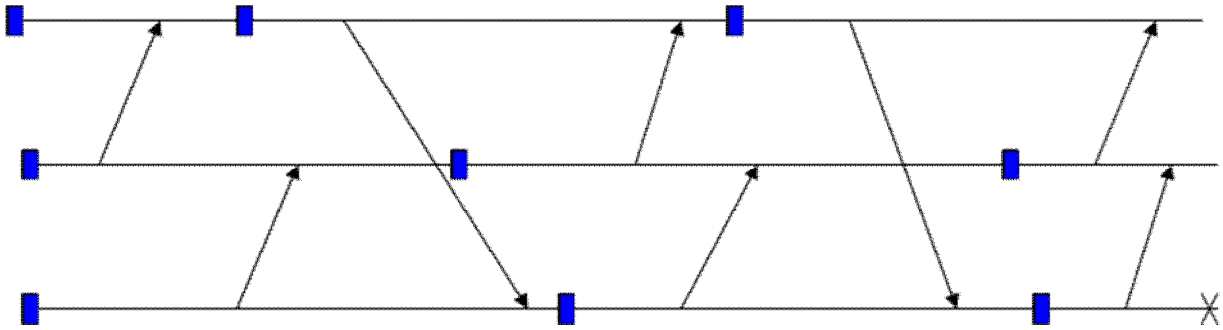
Synchronous and Asynchronous Check Pointing and Recovery

Approaches

- Asynchronous
    - The local checkpoints at different processes are taken independently
- Synchronous
    - The local checkpoints at different processes are *coordinated*
    - They may not be at the same time

Asynchronous Checkpointing:

- Problem
  - Domino effect



Failed process

Other Issues with Asynchronous Checkpointing:

- Useless checkpoints
- Need for garbage collection
- Recovery requires significant coordination
- Identify dependency between different *checkpoint intervals*
- This information is stored along with checkpoints in a stable storage
- When a process repairs, it requests this information from others to determine the need for rollback

Two Examples of Asynchronous Checkpointing:

- Bhargava and Lian
- Wang et al

Algorithm by Bhargava et al:

- Draw an edge from $c_{i,x}$ to $c_{j,y}$ if either
  - $i = j$ and $y = x+1$

- $i \neq j$ and a message m is sent from $I_{i, x}$ and received in $I_{j, y}$
    - Where $I_{i, x}$ is the interval between $c_{i, x-1}$ and $c_{i, x}$
    - Rollback recovery line used for recovery as well as garbage collection

Algorithm by Wang et al :

- Difference
    - If a message sent from $I_{i, x}$ is received in $I_{j, y}$ then draw an edge between $c_{j, x-1}$ to $c_{j, y}$
- Recovery line obtained is similar to that by by Bhargava and Lian
- Advantage
    - Number of useful checkpoints is at most N(N+1)/2
        - This can be shown that the number of checkpoints that are *ahead of recovery line*

Coordinated Checkpointing :

- Using diffusing computation
    - How can we use diffusing computation to obtain a consistent snapshot?

Algorithm by Tamir and Sequin:

- Blocking checkpoint
    - A coordinator decides when a checkpoint is taken
    - Coordinator sends a request message to all
    - Each process
        - Stops executing
        - Flushes the channels
        - Takes a tentative checkpoint
        - Replies to coordinator

- When all processes send replies, the coordinator asks them to change it to a permanent checkpoint

Minimal Checkpoint Coordination:

- Approach by Koo and Toueg
  - Require processes to take a checkpoint only if they have to

Logging Protocols

- Pessimistic
- Optimistic
- Causal

Fault Tolerance

Concepts of Fault Tolerance

- Hardware, software and networks cannot be totally free from failures
- Fault tolerance is a non-functional (QoS) requirement that requires a system to continue to operate, even in the presence of faults
- Fault tolerance should be achieved with minimal involvement of users or system administrators
- Distributed systems can be more fault tolerant than centralized systems, but with more processor hosts generally the occurrence of individual faults is likely to be more frequent
- Notion of a partial failure in a distributed system

Types of Fault (*wr*t time)

Hard or Permanent – repeatable error, e.g. failed component, power fail, fire, flood, design error (usually software), sabotage

Soft Fault

Transient – occurs once or seldom, often due to unstable environment (e.g. bird flies past microwave transmitter)

Intermittent – occurs randomly, but where factors influencing fault are not clearly identified, e.g. unstable component

Operator error – human error

Type of failure	Description
Crash failure	A server halts, but is working correctly until it halts
Amnesia crash	Lost all history, must be reboot
Pause crash	Still remember state before crash, can be recovered
Halting crash	Hardware failure, must be replaced or re-installed
Omission failure	A server fails to respond to incoming requests
Receive omission	A server fails to receive incoming messages
Send omission	A server fails to send messages
Timing failure	A server's response lies outside the specified time interval
Response failure	The server's response is incorrect
Value failure	The value of the response is wrong
State transition failure	The server deviates from the correct flow of control
Arbitrary failure	A server may produce arbitrary responses at arbitrary times

- Fault tolerance is related to **dependability**

Dependability Includes

- Availability
- Reliability
- Safety
- Maintainability

- **Availability**: A measurement of whether a system is *ready to be used immediately*
  - System is up and running at any given moment
- **Reliability**: A measurement of whether a system can *run continuously without failure*
  - System continues to function for a long period of time

  A system goes down 1ms/hr has an availability of more than 99.99%, but is unreliable

  A system that never crashes but is shut down for a week once every year is 100% reliable but only 98% available

- **Safety**: A measurement of *how safe failures are*
  - System fails, nothing serious happens
  - For instance, high degree of safety is required for systems controlling nuclear power plants
- **Maintainability**: A measurement of *how easy it is to repair a system*
  - A highly maintainable system may also show a high degree of availability
  - Failures can be detected and repaired automatically? Self-healing systems?

Distributed Commit

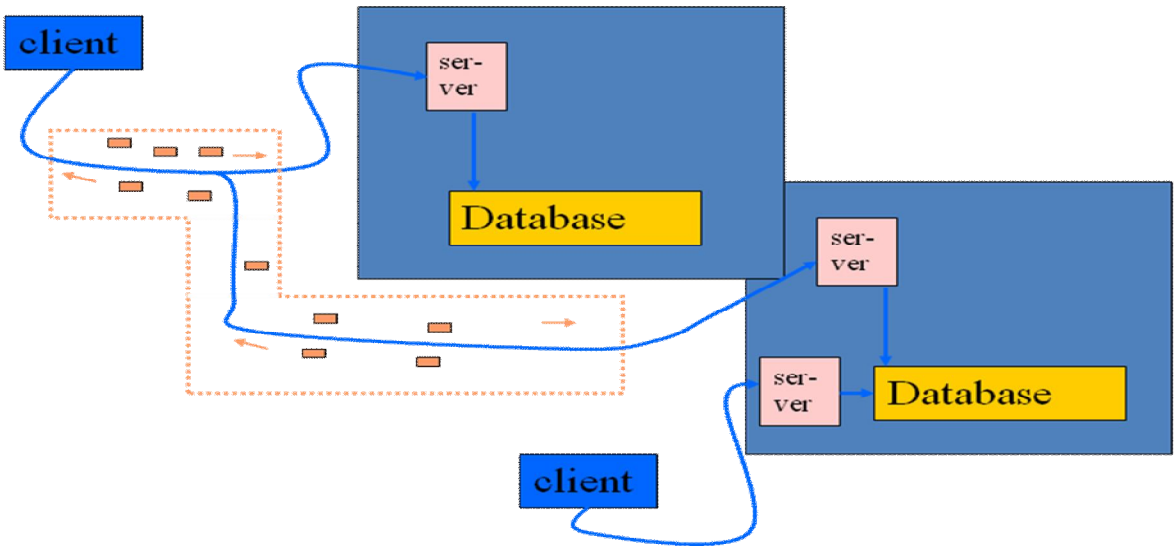
- **Goal**: Either **all** members of a group decide to perform an operation, or **none** of them perform the operation
- **Atomic transaction**: a transaction that happens completely or not at all
- Failures:
  - Crash failures that can be recovered

- Communication failures detectable by timeouts

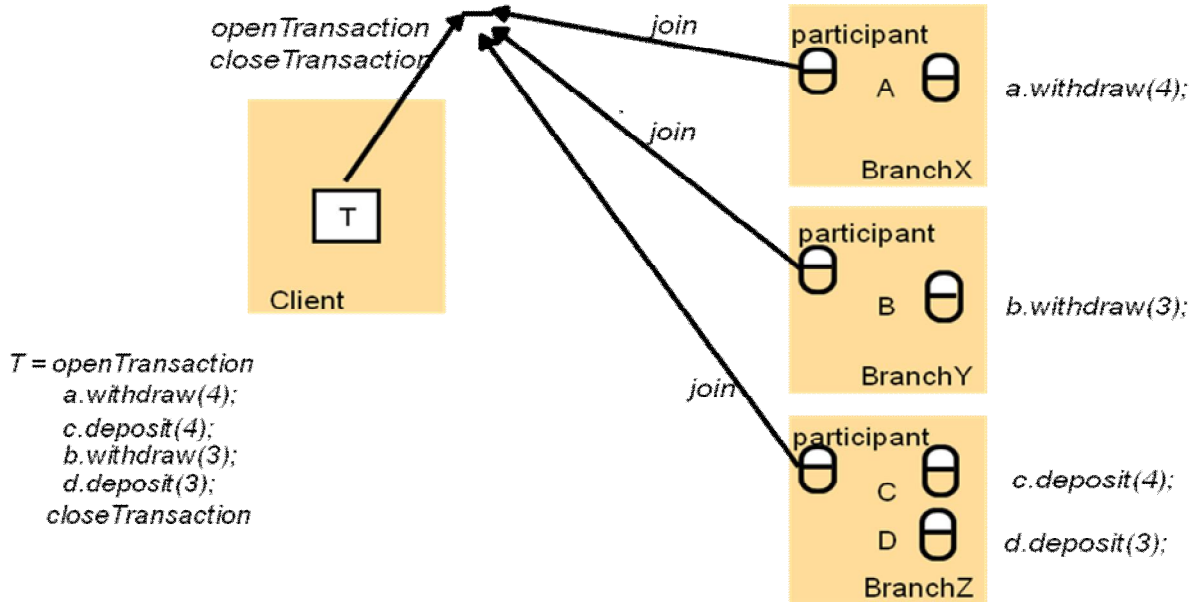
Notes:

- Commit requires a set of processes to agree…
- …similar to the Byzantine generals problem…
- … but the solution much simpler because stronger assumptions

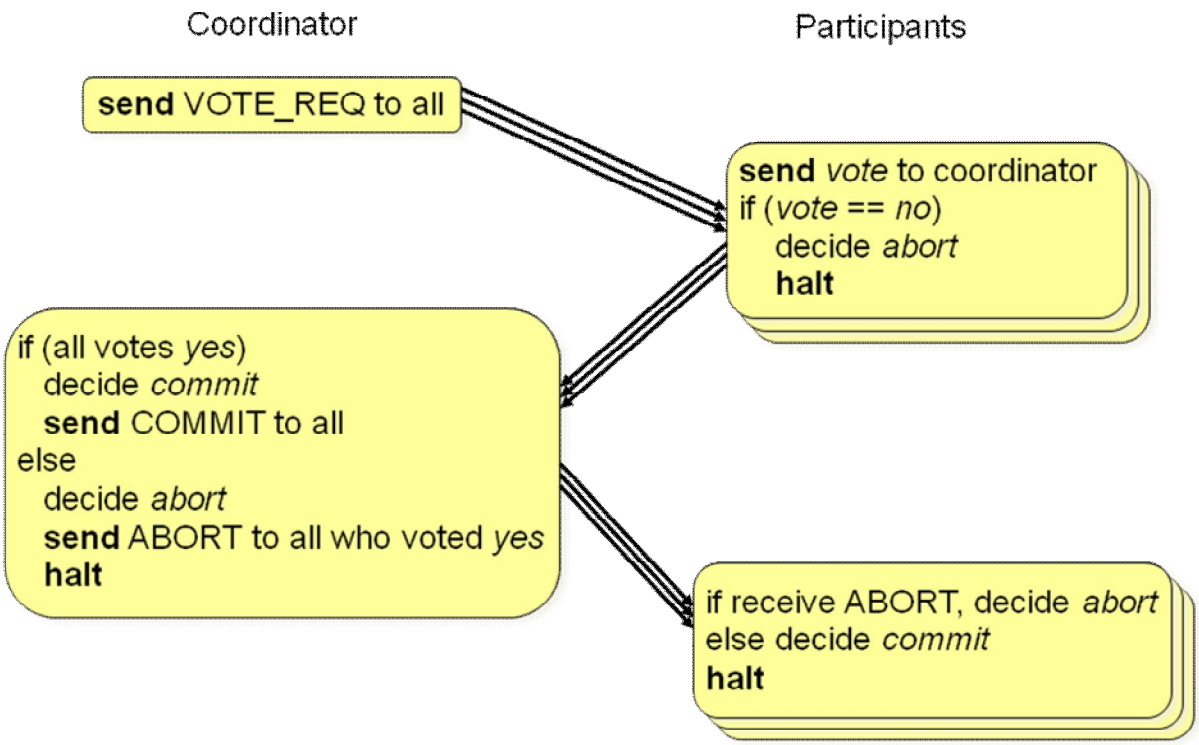
Distributed Transactions

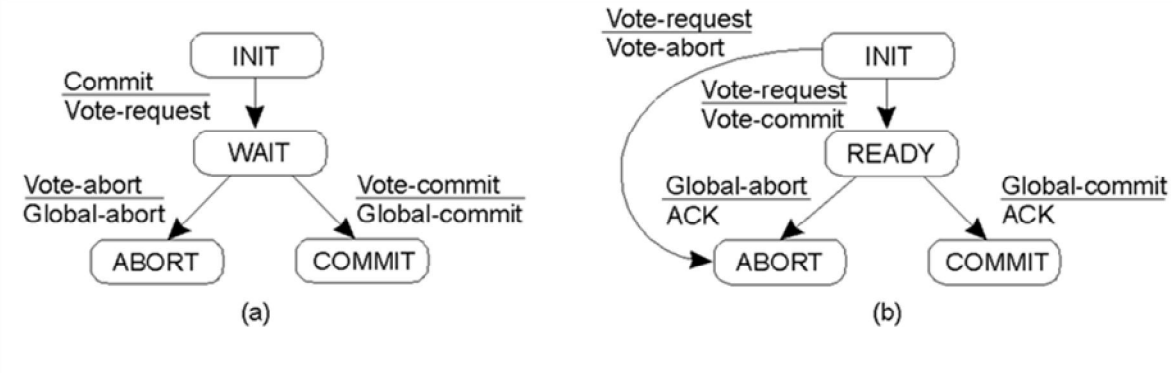


A Distributed Banking Transaction



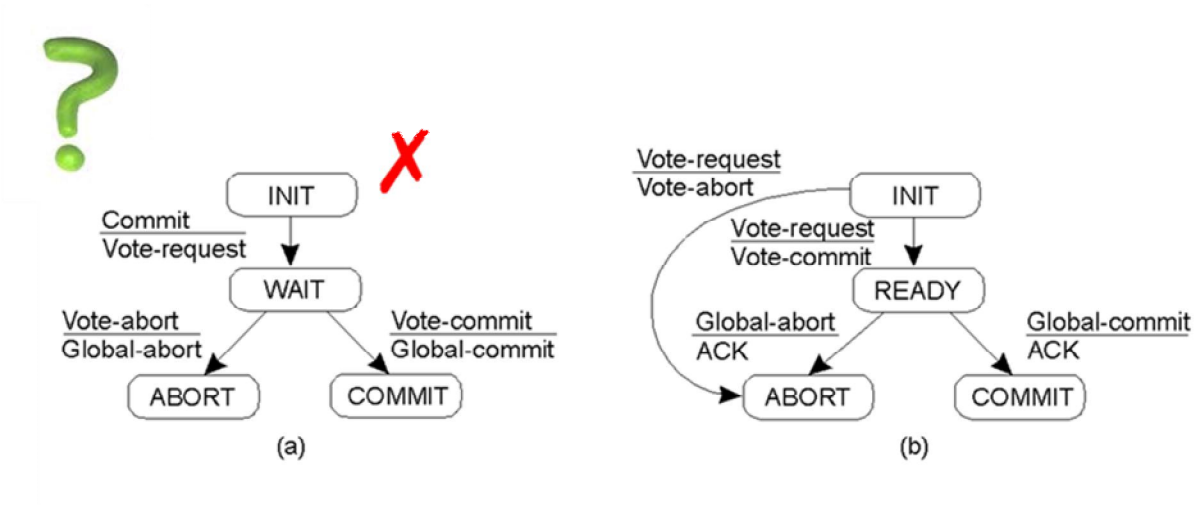
Two Phase Commit (2PC)





a) The finite state machine for the coordinator in 2PC.

b) The finite state machine for a participant.



a) The finite state machine for the coordinator in 2PC.

b) The finite state machine for a participant.

State of Q	Action by P

COMMIT	Make transition to COMMIT
ABORT	Make transition to ABORT
INIT	Make transition to ABORT
READY	Contact another participant

Actions taken by a participant *P* when residing in state *READY* and having contacted another participant *Q*.

UNIT IV

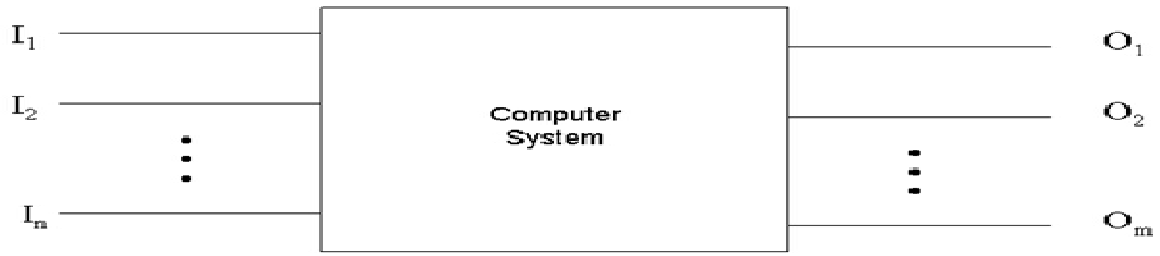
REAL TIME AND MOBILE OPERATING SYSTEMS 9

Basic Model of Real Time Systems - Characteristics- Applications of Real Time Systems – Real Time Task Scheduling - Handling Resource Sharing - Mobile Operating Systems – Micro Kernel Design - Client Server Resource Access – Processes and Threads - Memory Management - File system.

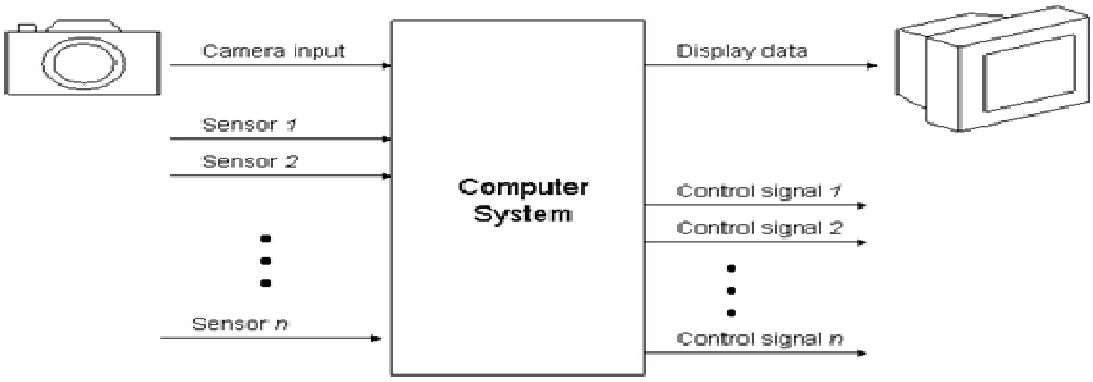
Basic Model of Real Time Systems

Systems concepts :

Definition: A system is a mapping of a set of inputs into a set of outputs.



A system with *n* inputs and *m* outputs.



Typical real-time control system including inputs from sensors and imaging devices and producing control signals and display information.

Definition: The time between the presentation of a set of inputs to a system (stimulus) and the realization of the required behavior, including the availability of all associated outputs, (response) is called the response time of the system

Real-time definitions:

Definition: A real-time system is a system that must satisfy explicit (bounded) response-time constraints or risk severe consequences, including failure.

Definition: A failed system is a system that cannot satisfy one or more of the requirements stipulated in the formal system specification.

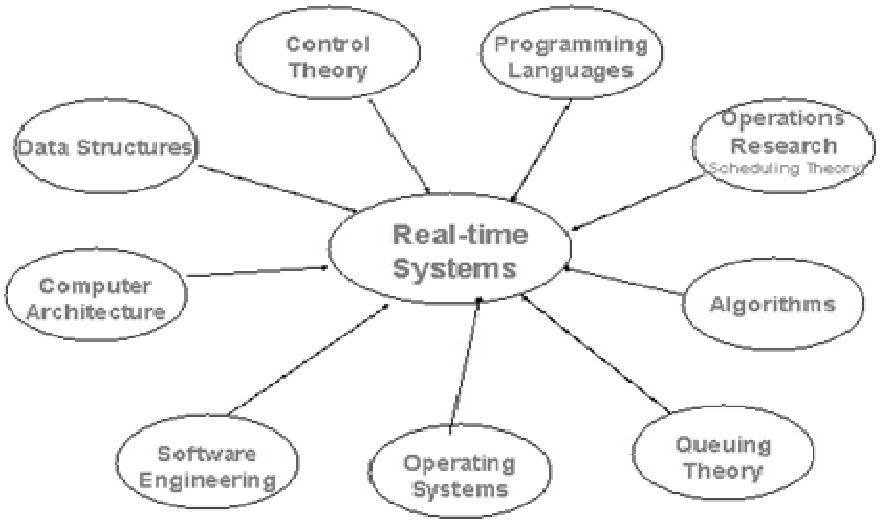
Definition: A real-time system is one whose logical correctness is based on both the correctness of the outputs and their timeliness.

Definition: A soft real-time system is one in which performance is degraded but not destroyed by failure to meet response-time constraints.

Definition: A hard real-time system is one in which failure to meet a single deadline may lead to complete and catastrophic system failure.

Definition: A firm real-time system is one in which a few missed deadlines will not lead to total failure, but missing more than a few may lead to complete and catastrophic system failure.

Real-time design issues



Disciplines that impact real-time systems.

- The selection of hardware and software, and evaluation of the tradeoff needed for a cost-effective solution, including dealing with distributed computing systems and the issues of parallelism and synchronization.
- Specification and design of real-time systems and correct representation of temporal behavior.
- Understanding the nuances of the programming language(s) and the real-time implications resulting from their translation into machine code.
- Maximizing of system fault tolerance and reliability through careful design.
- The design and administration of tests, and the selection of test and development equipment.
- Taking advantage of open systems technology and interoperability.
- Measuring and predicting response time and reducing it.
- Performing a schedulability analysis, that is, determining and guaranteeing deadline satisfaction, *a priori*, is largely "just" scheduling theory.

Example real-time systems

Domain	Applications
Avionics	• Navigation • Displays
Multimedia	• Games • Simulators
Medicine	• Robot surgery • Remote surgery • Medical imaging
Industrial Systems	• Robotic assembly lines • Automated inspection
Civilian	• Elevator control • Automotive systems

Real-time application domains

- Aircraft inertial measurement system
- Nuclear plant control
- Airline reservation system
- Pasta sauce bottling plant
- Traffic light control system for 4-way intersection

Characteristics of real-time software

- no virtual memory
- priority-based scheduling
- static resource allocation
- no file system
- usually one application program
- fast interprocess data transmission
- fast context switching and interrupt processing
- small size

Real-Time Scheduling

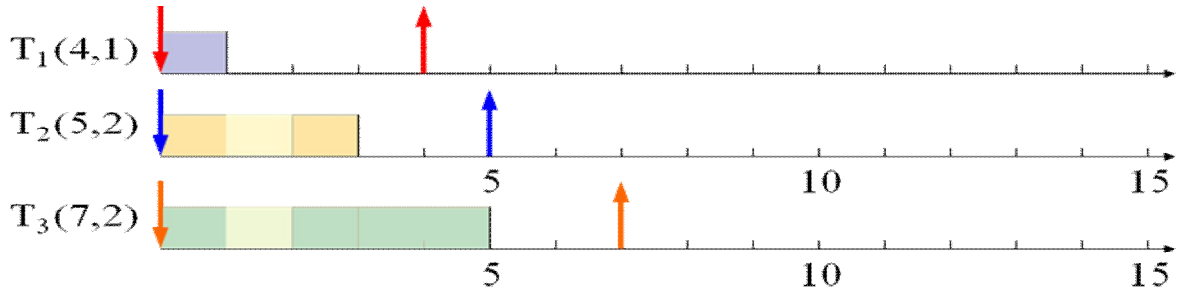
- **Fixed-priority algorithm (RM)**
- **Dynamic-priority algorithm (EDF)**

Determines the order of real-time task executions

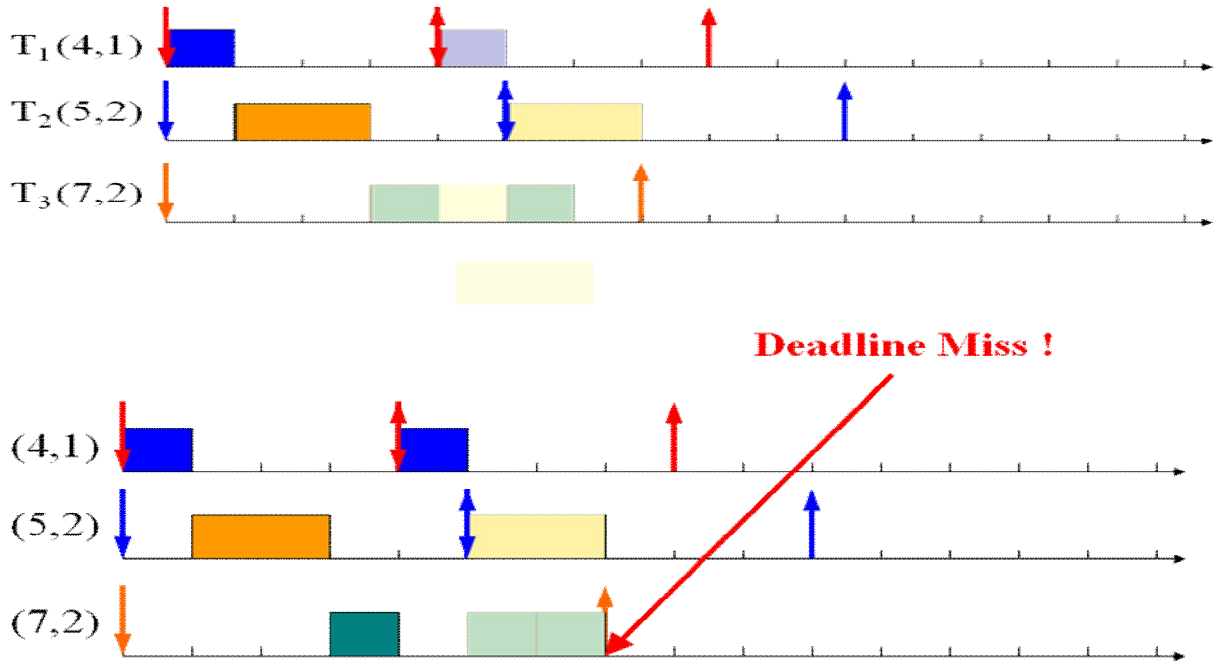
- Static-priority scheduling
- Dynamic-priority scheduling

RM (Rate Monotonic) :

- Optimal static-priority scheduling
- It assigns priority according to period
- A task with a shorter period has a higher priority
- Executes a job with the shortest period

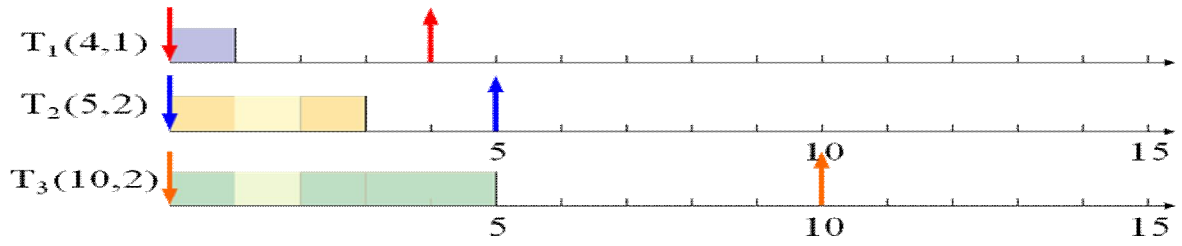


- **Executes a job with the shortest period**

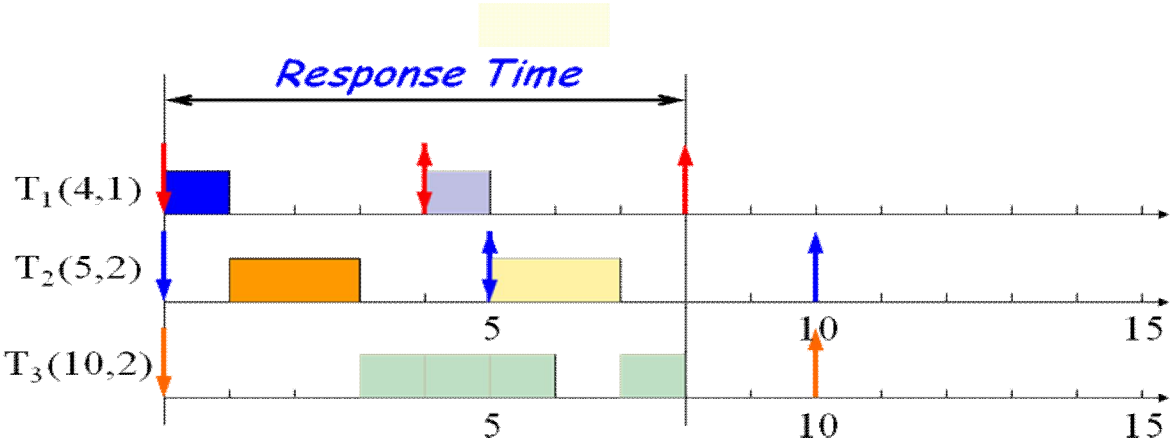


Response Time

- Response time
  - Duration from released time to finish time



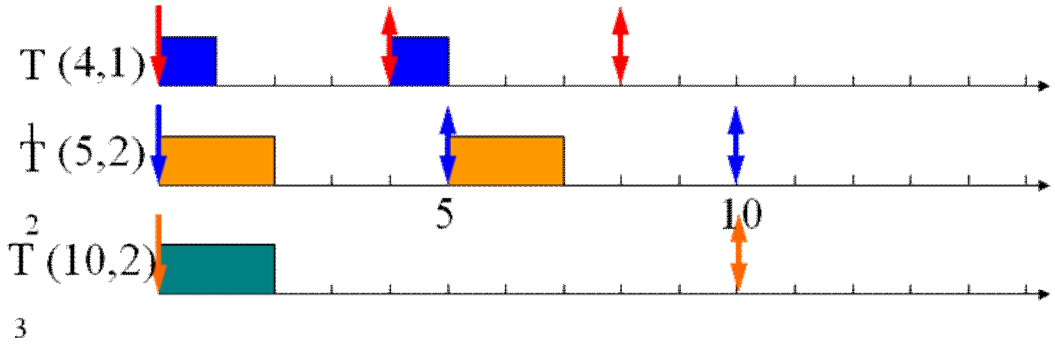
$$r_i = e_i + \sum_{T_k \in HP(T_i)} \left\lceil \frac{r_i}{p_k} \right\rceil \cdot e_k$$



- Response Time (r_i) [Audsley et al., 1993]

$$r_i = e_i + \sum_{T_k \in HP(T_i)} \left\lceil \frac{r_i}{p_k} \right\rceil \cdot e_k$$

HP(T_i) : a set of higher-priority tasks than T_i



RM - Schedulability Analysis

- Real-time system is schedulable under RM

  if and only if $r_i \leq p_i$ for all task $T_i(p_i,e_i)$

  Joseph & Pandya,

  "Finding response times in a real-time system",The Computer Journal, 1986.

RM – Utilization Bound

- Real-time system is schedulable under RM if

$$\sum U_i \leq n\,(2^{1/n}-1)$$

Liu & Layland,

"Scheduling algorithms for multi-programming in a hard-real-time environment", Journal of ACM, 1973.

- Real-time system is schedulable under RM if

$$\sum U_i \leq n\,(2^{1/n}-1)$$

- Example: $T_1(4,1)$, $T_2(5,1)$, $T_3(10,1)$,

$$\sum U_i = 1/4 + 1/5 + 1/10$$

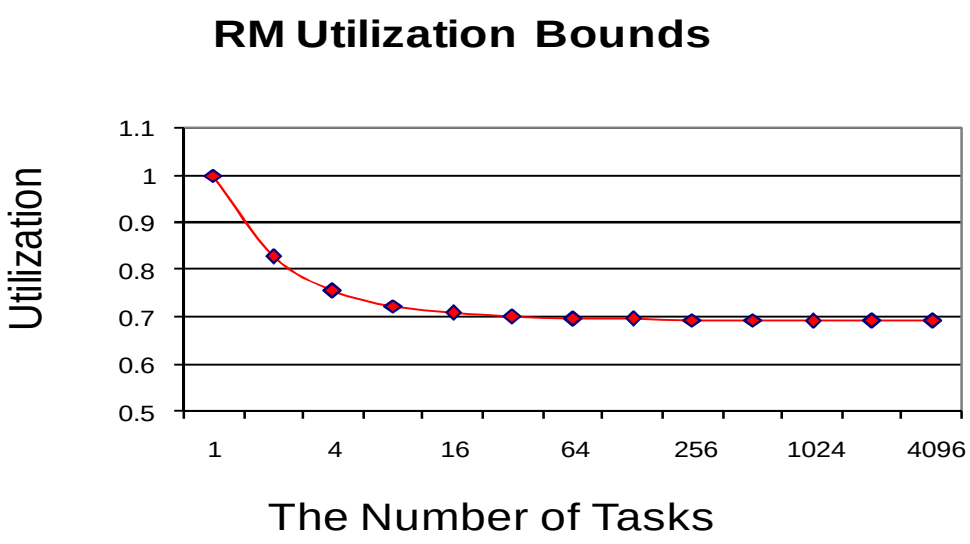
$$= 0.55$$

$$3\,(2^{1/3}-1) \approx 0.78$$

Thus, $\{T_1, T_2, T_3\}$ is schedulable under RM.

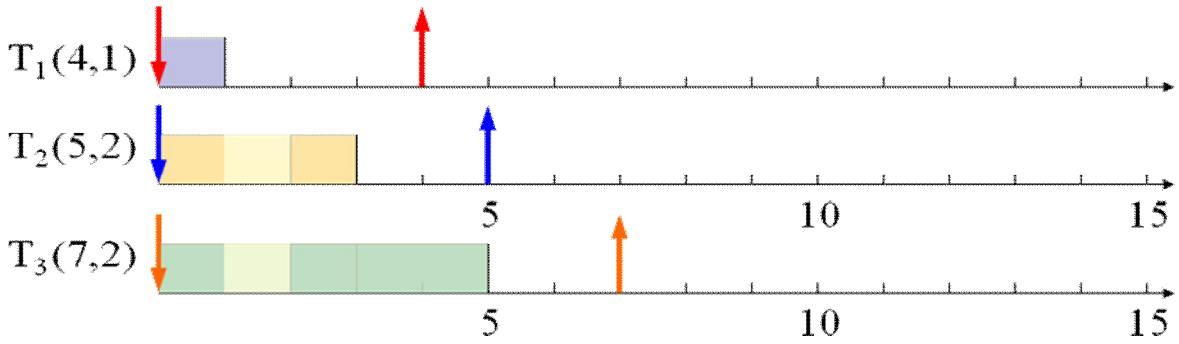
- Real-time system is schedulable under RM if

$$\sum U_i \leq n\,(2^{1/n}-1)$$

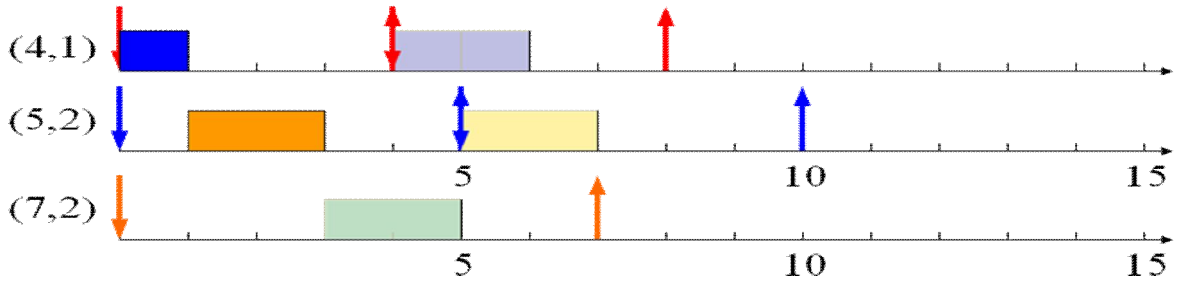


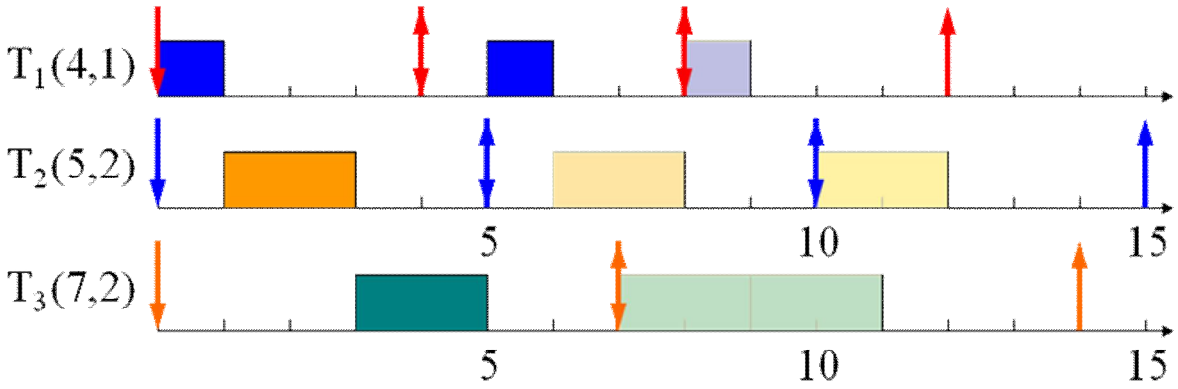
EDF (Earliest Deadline First):

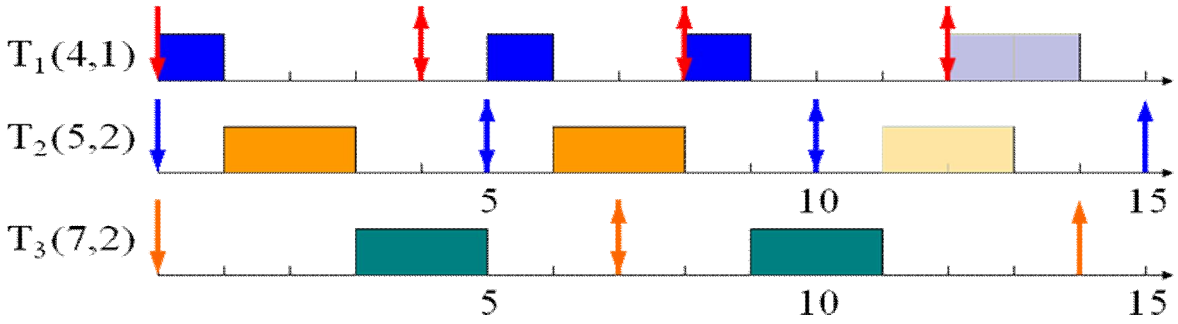
- Optimal dynamic priority scheduling
- A task with a shorter deadline has a higher priority
- Executes a job with the earliest deadline



- **Executes a job with the earliest deadline**

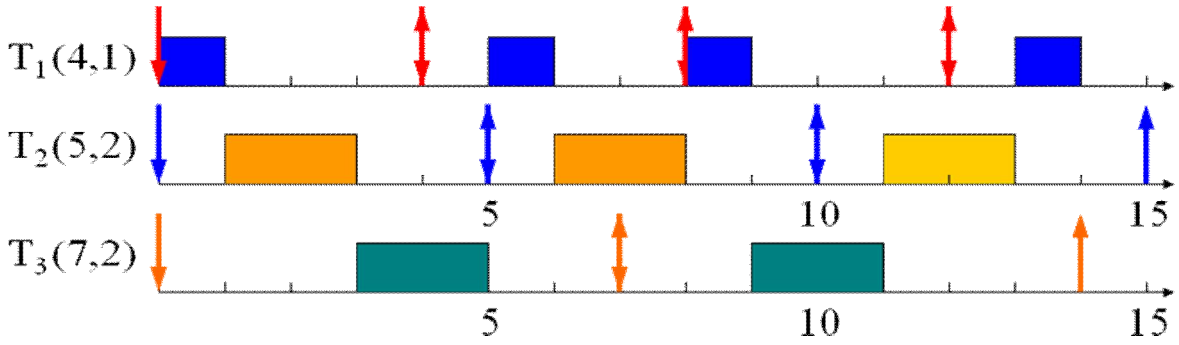






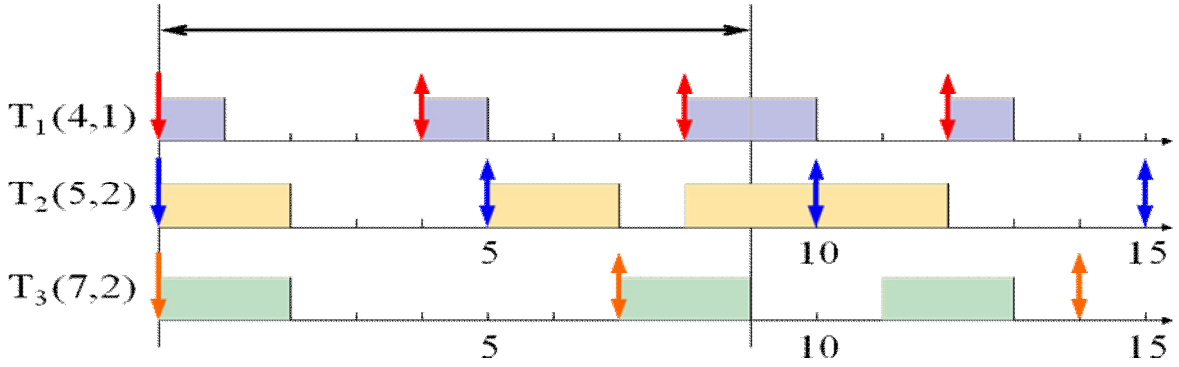
- Optimal scheduling algorithm
  - if there is a schedule for a set of real-time tasks,

  EDF can schedule it.



Processor Demand Bound:

- Demand Bound Function : *dbf(t)*
  - the maximum processor demand by workload over any interval of length *t*



- Real-time system is schedulable under EDF

  if and only if **dbf(t) ≤ t** for all interval t

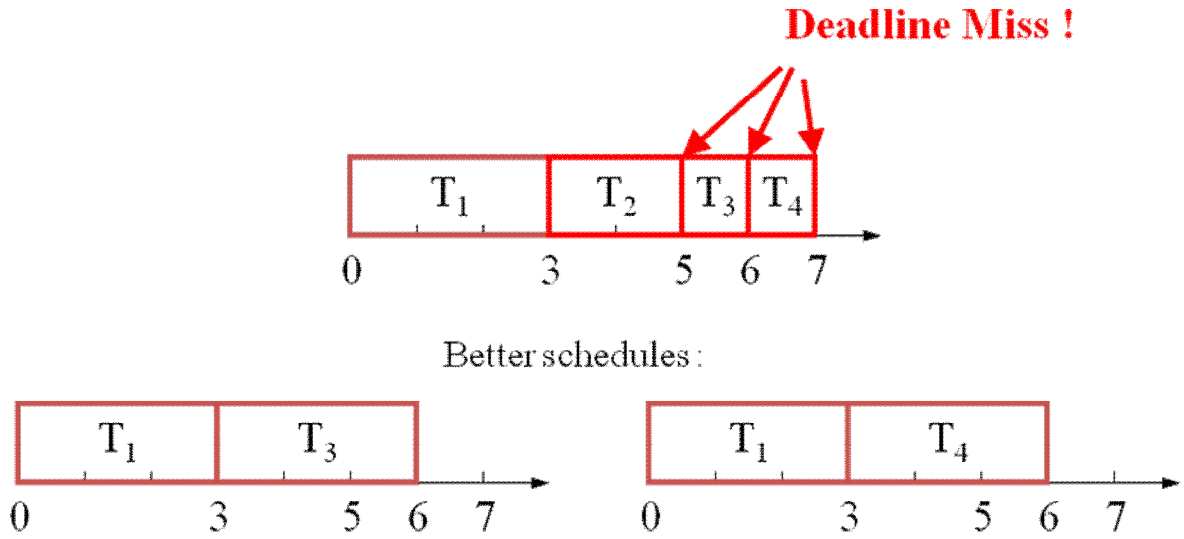
Baruah et al.

"Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor", Journal of Real-Time Systems, 1990.

- Demand Bound Function : dbf(t)
  - the maximum processor demand by workload over any interval of length t

EDF – Overload Conditions

- Domino effect during overload conditions
  - Example: $T_1(4,3)$, $T_2(5,3)$, $T_3(6,3)$, $T_4(7,3)$



RM vs. EDF

Rate Monotonic

- Simpler implementation, even in systems without explicit support for timing constraints (periods, deadlines)
- Predictability for the highest priority tasks

EDF

- Full processor utilization
- Misbehavior during overload conditions

Microkernels:

- Different Types of Kernel Designs
    - Monolithic kernel
    - Microkernel
    - Hybrid Kernel
    - Exokernel

Monolithic Kernels

- All OS services operate in kernel space
- Good performance
- Disadvantages
    - Dependencies between system component
    - Complex & huge (millions(!) of lines of code)
    - Larger size makes it hard to maintain

E.g. Multics, Unix, BSD, Linux

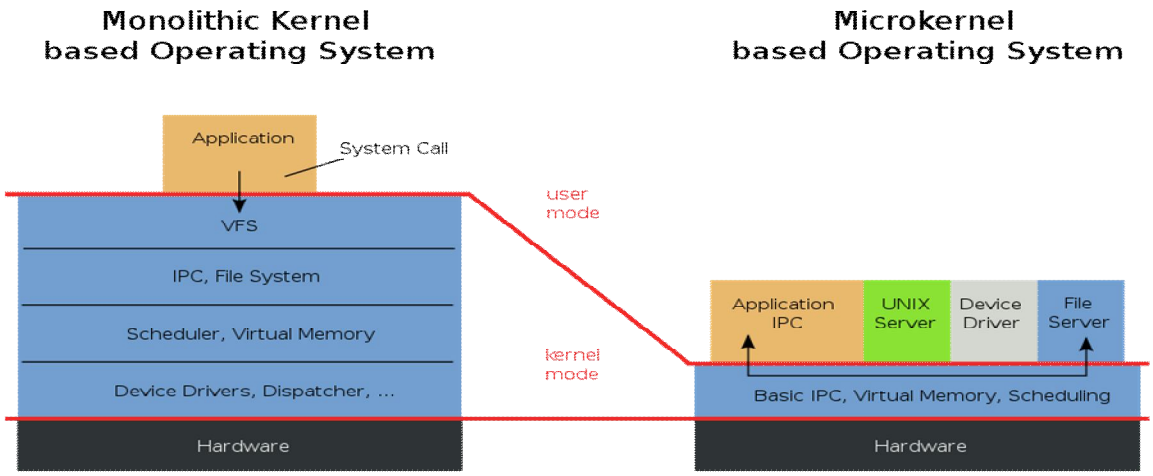
- Advantages: performance
- Disadvantages: difficult to debug and maintain

Microkernels

- Minimalist approach
    - IPC, virtual memory, thread scheduling
- Put the rest into user space
    - Device drivers, networking, file system, user interface
- More stable with less services in kernel space
- Disadvantages
    - Lots of system calls and context switches
- E.g. Mach, L4, AmigaOS, Minix, K42

- Microkernels
  - Advantages: more reliable and secure
  - Disadvantages: more overhead

Monolithic Kernels VS Microkernels



Hybrid Kernels

- Combine the best of both worlds
  - Speed and simple design of a monolithic kernel
  - Modularity and stability of a microkernel
- Still similar to a monolithic kernel
  - Disadvantages still apply here
- E.g. Windows NT, NetWare, BeOS

  - Advantages: benefits of monolithic and microkernels
  - Disadvantages: same as monolithic kernels

Exokernels

- Follows end-to-end principle
  - Extremely minimal
  - Fewest hardware abstractions as possible
  - Just allocates physical resources to apps
- Disadvantages
  - More work for application developers
- E.g. Nemesis, ExOS

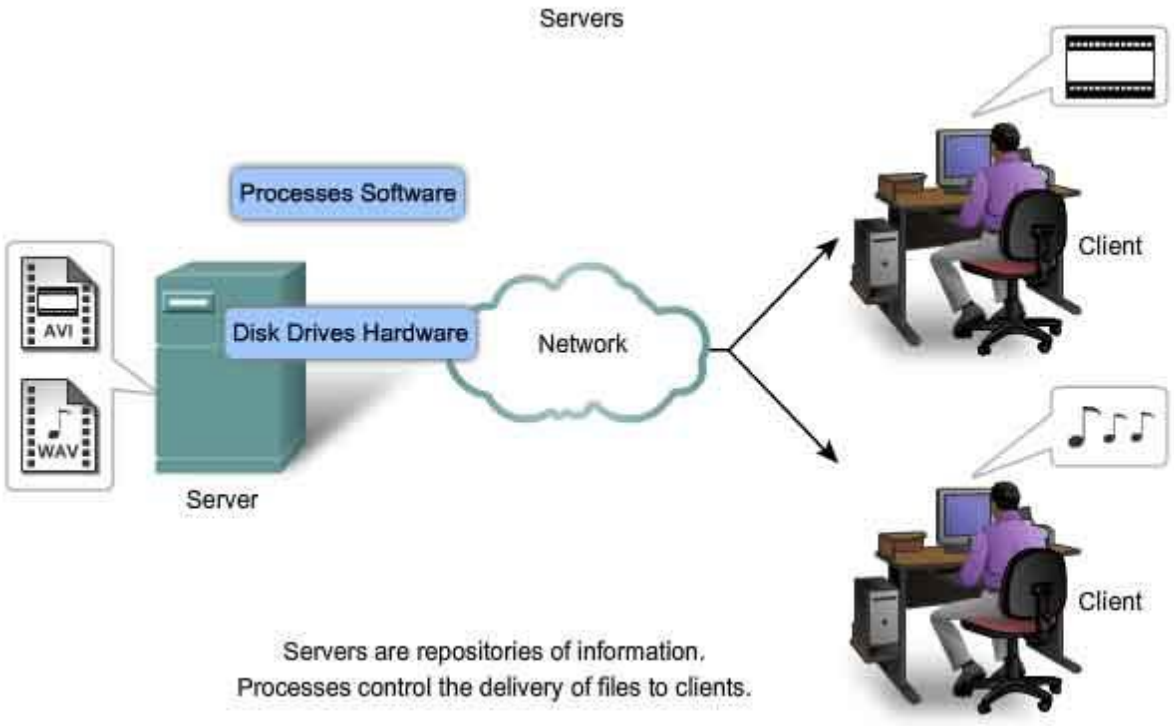
  - Advantages: minimal and simple
  - Disadvantages: more work for application developers

Client Server Resource Access

Client/Server Model:

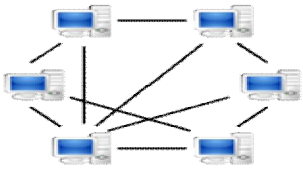
- The client/server model database example
  - SQL
- Client/server advantages
- Migration to client/server architecture
  - Workstations

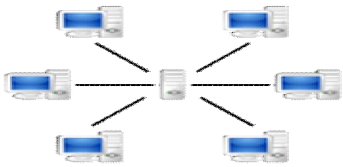
Client/Server Networking Model



Networking Categories

- Peer-to-peer
- Server-based





UNIT V CASE STUDIES

Linux System: Design Principles - Kernel Modules - Process Management Scheduling -Memory Management - Input-Output Management - File System – Interprocess Communication. iOS and Android: Architecture and SDK Framework - Media Layer -Services Layer - Core OS Layer - File System.

Linux System: Design Principles

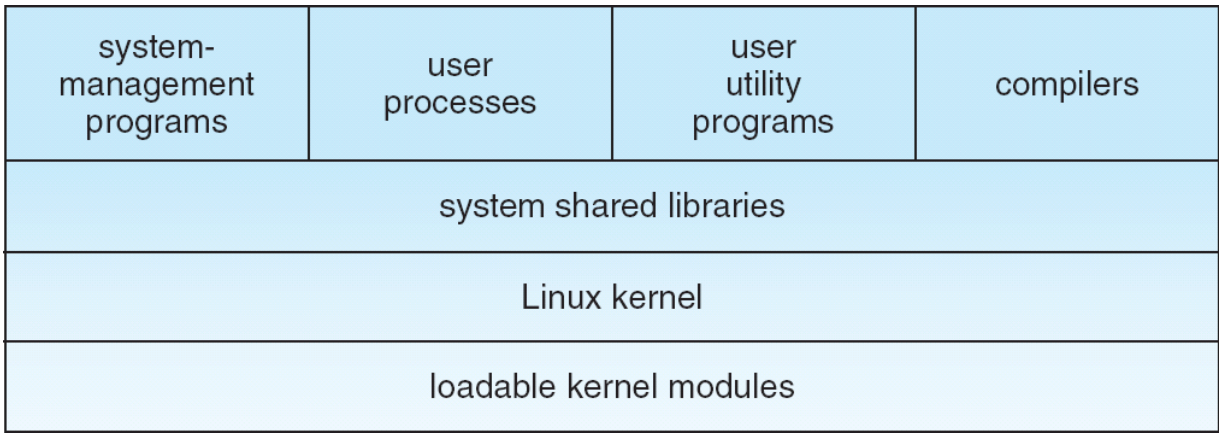
Objectives

1. To explore the history of the UNIX operating system from which Linux is derived and the principles which Linux is designed upon
2. To examine the Linux process model and illustrate how Linux schedules processes and provides interprocess communication
3. To look at memory management in Linux
4. To explore how Linux implements file systems and manages I/O devices

Design Principles

- Linux is a multiuser, multitasking system with a full set of UNIX-compatible tools
- Its file system adheres to traditional UNIX semantics, and it fully implements the standard UNIX networking model
- Main design goals are speed, efficiency, and standardization
- Linux is designed to be compliant with the relevant POSIX documents; at least two Linux distributions have achieved official POSIX certification
- The Linux programming interface adheres to the SVR4 UNIX semantics, rather than to BSD behavior

Components of a Linux System



- Like most UNIX implementations, Linux is composed of three main bodies of code; the most important distinction between the kernel and all other components
- The kernel is responsible for maintaining the important abstractions of the operating system
- Kernel code executes in *kernel mode* with full access to all the physical resources of the computer All kernel code and data structures are kept in the same single address space
- The system libraries define a standard set of functions through which applications interact with the kernel, and which implement much of the operating-system functionality that does not need the full privileges of kernel code
- The system utilities perform individual specialized management tasks

Kernel Modules

- Sections of kernel code that can be compiled, loaded, and unloaded independent of the rest of the kernel
- A kernel module may typically implement a device driver, a file system, or a networking protocol
- The module interface allows third parties to write and distribute, on their own terms, device drivers or file systems that could not be distributed under the GPL

- Kernel modules allow a Linux system to be set up with a standard, minimal kernel, without any extra device drivers built in

Three components to Linux module support:

- ✓ module management
- ✓ driver registration
- ✓ conflict resolution

Module Management

- Supports loading modules into memory and letting them talk to the rest of the kernel
- Module loading is split into two separate sections:
  - ✓ Managing sections of module code in kernel memory
  - ✓ Handling symbols that modules are allowed to reference
- The module requestor manages loading requested, but currently unloaded, modules; it also regularly queries the kernel to see whether a dynamically loaded module is still in use, and will unload it when it is no longer actively needed

Driver Registration

- Allows modules to tell the rest of the kernel that a new driver has become available
- The kernel maintains dynamic tables of all known drivers, and provides a set of routines to allow drivers to be added to or removed from these tables at any time
- Registration tables include the following items:
  - Device drivers
  - File systems
  - Network protocols
  - Binary format

Conflict Resolution

- A mechanism that allows different device drivers to reserve hardware resources and to protect those resources from accidental use by another driver
- The conflict resolution module aims to:
  - ✓ Prevent modules from clashing over access to hardware resources

- ✓ Prevent *autoprobes* from interfering with existing device drivers
- ✓ Resolve conflicts with multiple drivers trying to access the same hardware

Process Management

- UNIX process management separates the creation of processes and the running of a new program into two distinct operations.
  - ✓ The fork system call creates a new process
  - ✓ A new program is run after a call to execve
- Under UNIX, a process encompasses all the information that the operating system must maintain to track the context of a single execution of a single program
- Under Linux, process properties fall into three groups: the process's identity, environment, and context

Process Identity

- Process ID (PID). The unique identifier for the process; used to specify processes to the operating system when an application makes a system call to signal, modify, or wait for another process
- Credentials. Each process must have an associated user ID and one or more group IDs that determine the process's rights to access system resources and files
- Personality. Not traditionally found on UNIX systems, but under Linux each process has an associated personality identifier that can slightly modify the semantics of certain system calls
  - ✓ Used primarily by emulation libraries to request that system calls be compatible with certain specific flavors of UNIX

Process Environment

- The process's environment is inherited from its parent, and is composed of two null-terminated vectors:
  - ✓ The argument vector lists the command-line arguments used to invoke the running program; conventionally starts with the name of the program itself
  - ✓ The environment vector is a list of "NAME=VALUE" pairs that associates named environment variables with arbitrary textual values

- Passing environment variables among processes and inheriting variables by a process's children are flexible means of passing information to components of the user-mode system software
- The environment-variable mechanism provides a customization of the operating system that can be set on a per-process basis, rather than being configured for the system as a whole

Process Context

- The (constantly changing) state of a running program at any point in time
- The scheduling context is the most important part of the process context; it is the information that the scheduler needs to suspend and restart the process
- The kernel maintains accounting information about the resources currently being consumed by each process, and the total resources consumed by the process in its lifetime so far
- The file table is an array of pointers to kernel file structures
    - ✓ When making file I/O system calls, processes refer to files by their index into this table
- Whereas the file table lists the existing open files, the file-system context applies to requests to open new files
- The current root and default directories to be used for new file searches are stored here
- The **signal-handler table** defines the routine in the process's address space to be called when specific signals arrive
- The **virtual-memory context** of a process describes the full contents of the its private address space

Processes and Threads

- Linux uses the same internal representation for processes and threads; a thread is simply a new process that happens to share the same address space as its parent
- A distinction is only made when a new thread is created by the clone system call
    - o fork creates a new process with its own entirely new process context

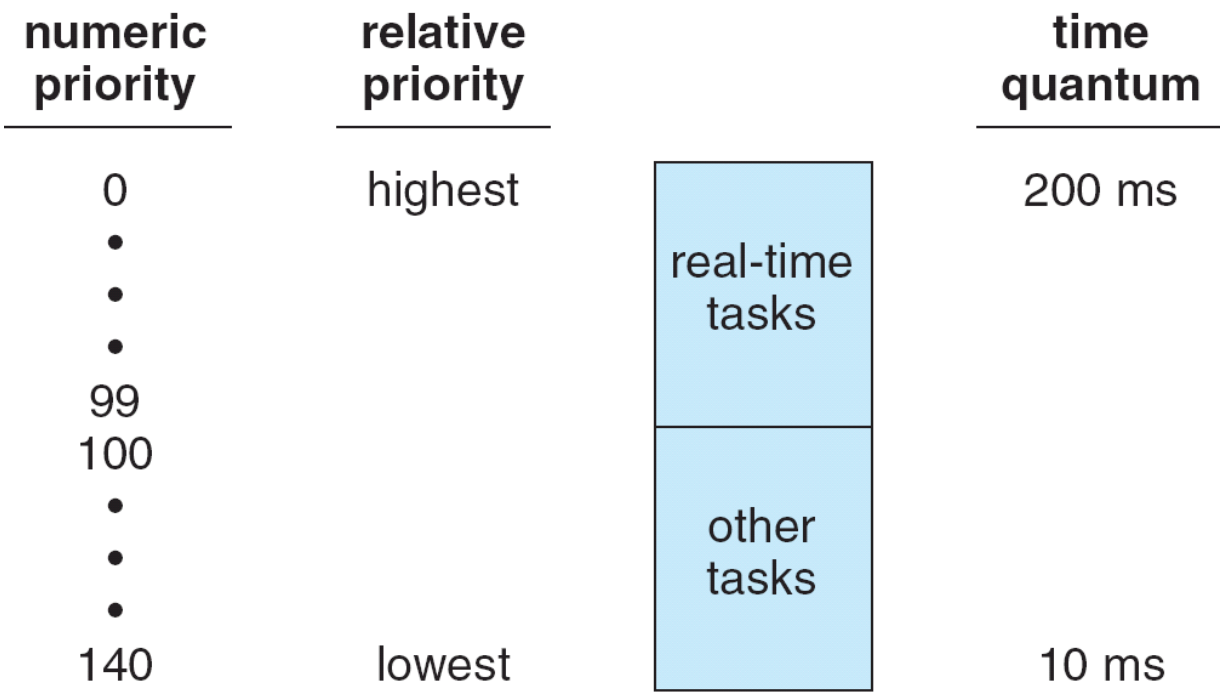
- clone creates a new process with its own identity, but that is allowed to share the data structures of its parent

- Using clone gives an application fine-grained control over exactly what is shared between two threads

Scheduling

- The job of allocating CPU time to different tasks within an operating system
- While scheduling is normally thought of as the running and interrupting of processes, in Linux, scheduling also includes the running of the various kernel tasks
- Running kernel tasks encompasses both tasks that are requested by a running process and tasks that execute internally on behalf of a device driver
- As of 2.5, new scheduling algorithm – preemptive, priority-based
  - Real-time range
  - nice value

Relationship Between Priorities and Time-slice Length



List of Tasks Indexed by Priority

Kernel Synchronization

- A request for kernel-mode execution can occur in two ways:
  - A running program may request an operating system service, either explicitly via a system call, or implicitly, for example, when a page fault occurs
  - A device driver may deliver a hardware interrupt that causes the CPU to start executing a kernel-defined handler for that interrupt
- Kernel synchronization requires a framework that will allow the kernel's critical sections to run without interruption by another critical section

Process Scheduling

- Linux uses two process-scheduling algorithms:
  - A time-sharing algorithm for fair preemptive scheduling between multiple processes
  - A real-time algorithm for tasks where absolute priorities are more important than fairness
- A process's scheduling class defines which algorithm to apply
- For time-sharing processes, Linux uses a prioritized, credit based algorithm
  - The crediting rule factors in both the process's history and its priority

$$\text{credits} := \frac{\text{credits}}{2} + \text{priority}$$

  - This crediting system automatically prioritizes interactive or I/O-bound processes

- Linux implements the FIFO and round-robin real-time scheduling classes; in both cases, each process has a priority in addition to its scheduling class
  - The scheduler runs the process with the highest priority; for equal-priority processes, it runs the process waiting the longest
  - FIFO processes continue to run until they either exit or block
  - A round-robin process will be preempted after a while and moved to the end of the scheduling queue, so that round-robing processes of equal priority automatically time-share between themselves

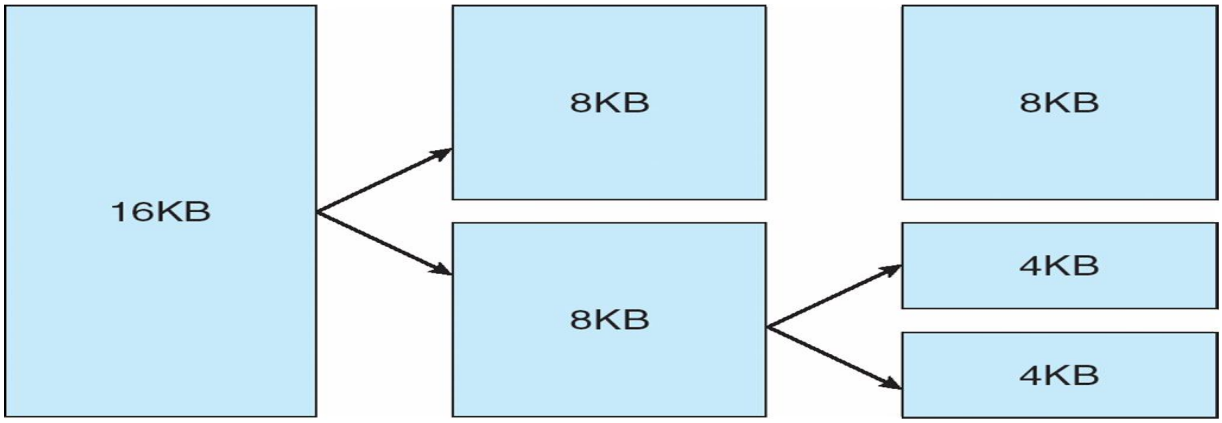
Memory Management

- Linux's physical memory-management system deals with allocating and freeing pages, groups of pages, and small blocks of memory
- It has additional mechanisms for handling virtual memory, memory mapped into the address space of running processes
- Splits memory into 3 different zones due to hardware characteristics

Relationship of Zones and Physical Addresses on 80x86

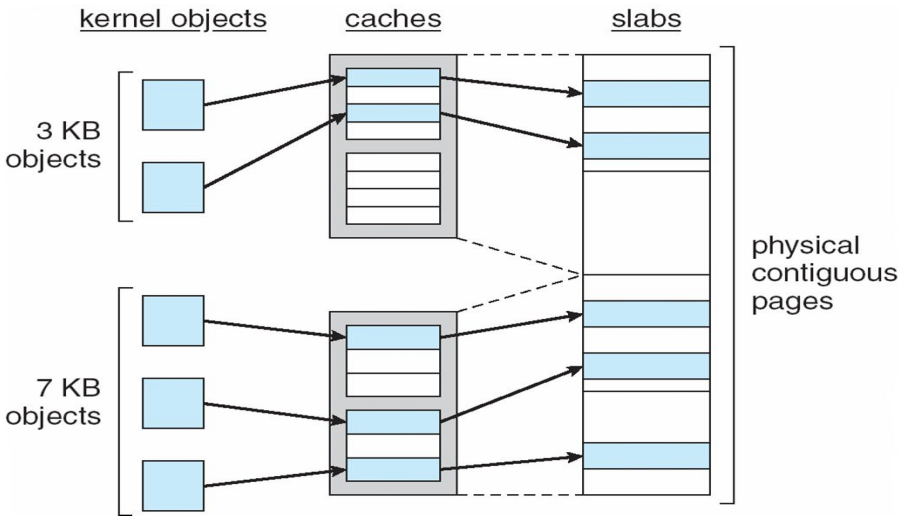
zone	physical memory
ZONE_DMA	< 16 MB
ZONE_NORMAL	16 .. 896 MB
ZONE_HIGHMEM	> 896 MB

Splitting of Memory in a Buddy Heap



Managing Physical Memory

- The page allocator allocates and frees all physical pages; it can allocate ranges of physically-contiguous pages on request
- The allocator uses a buddy-heap algorithm to keep track of available physical pages
  - Each allocatable memory region is paired with an adjacent partner
  - Whenever two allocated partner regions are both freed up they are combined to form a larger region
  - If a small memory request cannot be satisfied by allocating an existing small free region, then a larger free region will be subdivided into two partners to satisfy the request
- Memory allocations in the Linux kernel occur either statically (drivers reserve a contiguous area of memory during system boot time) or dynamically (via the page allocator)
- Also uses slab allocator for kernel memory



Virtual Memory

- The VM system maintains the address space visible to each process: It creates pages of virtual memory on demand, and manages the loading of those pages from disk or their swapping back out to disk as required
- The VM manager maintains two separate views of a process's address space:
  - A logical view describing instructions concerning the layout of the address space
    - The address space consists of a set of nonoverlapping regions, each representing a continuous, page-aligned subset of the address space
  - A physical view of each address space which is stored in the hardware page tables for the process
- Virtual memory regions are characterized by:
  - The backing store, which describes from where the pages for a region come; regions are usually backed by a file or by nothing (*demand-zero* memory)
  - The region's reaction to writes (page sharing or copy-on-write)
- The kernel creates a new virtual address space
    - 1.When a process runs a new program with the exec system call
    - 2. Upon creation of a new process by the fork system call
- On executing a new program, the process is given a new, completely empty virtual-address space; the program-loading routines populate the address space with virtual-memory regions
- Creating a new process with fork involves creating a complete copy of the existing process's virtual address space
  - ✓ The kernel copies the parent process's VMA descriptors, then creates a new set of page tables for the child
  - ✓ The parent's page tables are copied directly into the child's, with the reference count of each page covered being incremented
  - ✓ After the fork, the parent and child share the same physical pages of memory in their address spaces

- ❖ The VM paging system relocates pages of memory from physical memory out to disk when the memory is needed for something else

- ❖ The VM paging system can be divided into two sections:
- ❖ The pageout-policy algorithm decides which pages to write out to disk, and when
- ❖ The paging mechanism actually carries out the transfer, and pages data back into physical memory as needed

File Systems

- To the user, Linux's file system appears as a hierarchical directory tree obeying UNIX semantics
- Internally, the kernel hides implementation details and manages the multiple different file systems via an abstraction layer, that is, the *virtual file system (VFS)*
- The Linux VFS is designed around object-oriented principles and is composed of two components:
    - o A set of definitions that define what a file object is allowed to look like
        - The *inode-object* and the *file-object* structures represent individual files
        - the *file system object* represents an entire file system
    - o A layer of software to manipulate those objects

The Linux Proc File System

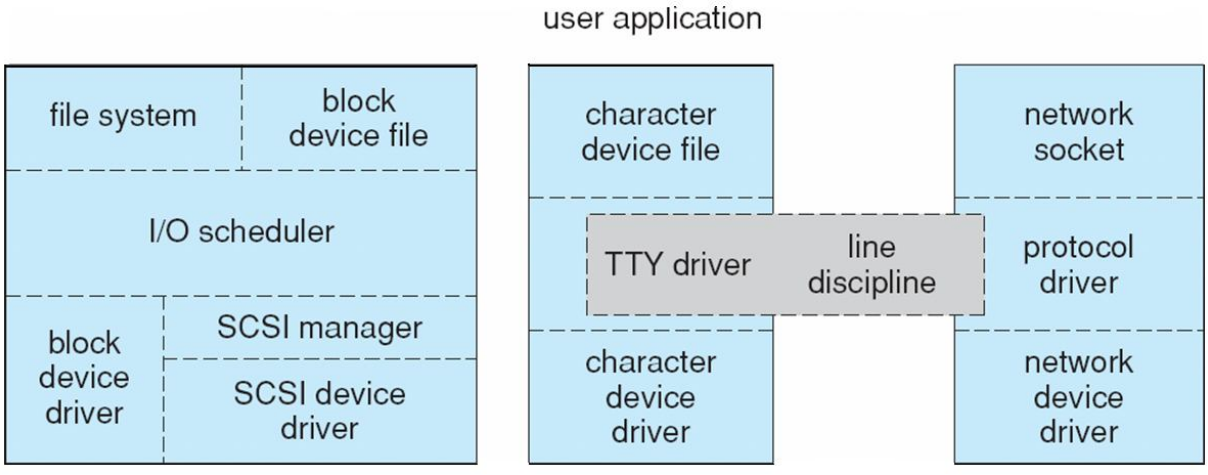
- The proc file system does not store data, rather, its contents are computed on demand according to user file I/O requests
- **proc** must implement a directory structure, and the file contents within; it must then define a unique and persistent inode number for each directory and files it contains
    - o It uses this inode number to identify just what operation is required when a user tries to read from a particular file inode or perform a lookup in a particular directory inode

- When data is read from one of these files, **proc** collects the appropriate information, formats it into text form and places it into the requesting process's read buffer

Input and Output

- The Linux device-oriented file system accesses disk storage through two caches:
  - ✓ Data is cached in the page cache, which is unified with the virtual memory system
  - ✓ Metadata is cached in the buffer cache, a separate cache indexed by the physical disk block
- Linux splits all devices into three classes:
  - ✓ *block devices* allow random access to completely independent, fixed size blocks of data
  - ✓ *character devices* include most other devices; they don't need to support the functionality of regular files
  - ✓ *network devices* are interfaced via the kernel's networking subsystem

Device-Driver Block Structure



Block Devices

- Provide the main interface to all disk devices in a system
- The *block buffer* cache serves two main purposes:
    - it acts as a pool of buffers for active I/O
    - it serves as a cache for completed I/O
- The *request manager* manages the reading and writing of buffer contents to and from a block device driver

Character Devices

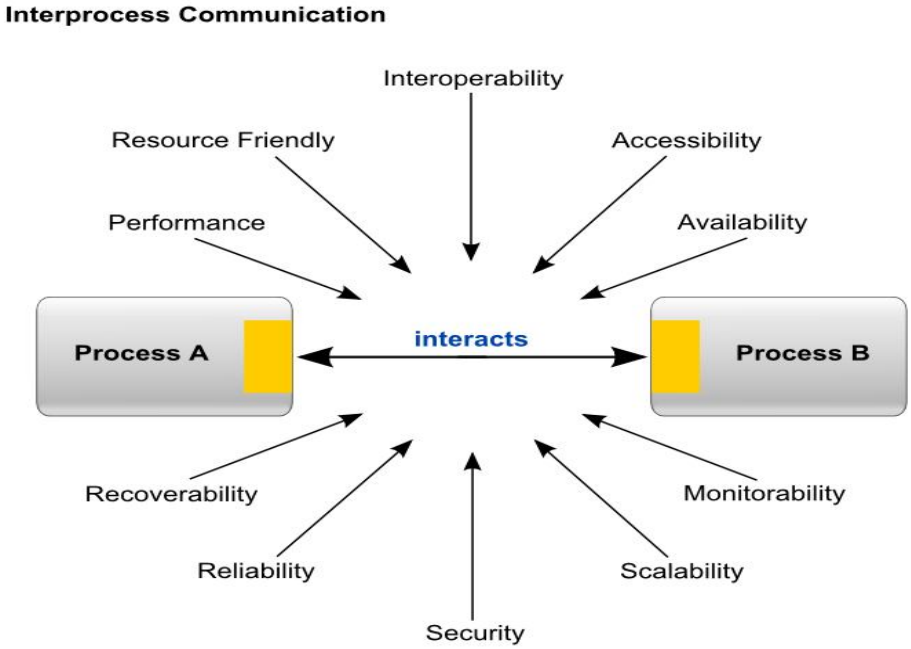
- ❖ A device driver which does not offer random access to fixed blocks of data
- ❖ A character device driver must register a set of functions which implement the driver's various file I/O operations
- ❖ The kernel performs almost no preprocessing of a file read or write request to a character device, but simply passes on the request to the device
- ❖ The main exception to this rule is the special subset of character device drivers which implement terminal devices, for which the kernel maintains a standard interface

Interprocess Communication

inter-process communication (**IPC**) is a set of methods for the exchange of data among multiple threads in one or more processes. Processes may be running on one or more computers connected by a network. IPC methods are divided into methods for message passing, synchronization, shared memory, and remote procedure calls (RPC). The method of IPC used may vary based on the bandwidth and latency of communication between the threads, and the type of data being communicated.

There are several reasons for providing an environment that allows process cooperation:

- Information sharing
- Computational speedup
- Modularity
- Convenience
- Privilege separation



Main IPC methods

Method	Short Description	Provided by (operating systems or other environments)
File	A record stored on disk that can be accessed by name by any process	Most operating systems
Signal	A system message sent from one process to another, not usually used to store information but instead give commands.	Most operating systems; some systems, such as Win NT subsystem, implement signals in only the C run-time library and provide no support for their use as an IPC method[*citation needed*]. But other subsystems like the POSIX subsystem provided by default until windows 2000. Then available with interix in XP/2003 then with « windows services for UNIX » (SFU).

Socket	A data stream sent over a network interface, either to a different process on the same computer or to another computer	Most operating systems
Message queue	An anonymous data stream similar to the pipe, but stores and retrieves information in packets.	Most operating systems
Pipe	A two-way data stream interfaced through standard input and output and is read character by character.	All POSIX systems, Windows
Named pipe	A pipe implemented through a file on the file system instead of standard input and output.	All POSIX systems, Windows
Semaphore	A simple structure that synchronizes threads or processes acting on shared resources.	All POSIX systems, Windows
Shared memory	Multiple processes given access to the same memory, allowing all to change it and	All POSIX systems, Windows

	read changes made by other processes.	
Message passing (shared nothing)	Similar to the message queue.	Used in MPI paradigm, Java RMI, CORBA, DDS, MSMQ, MailSlots, QNX, others
Memory-mapped file	A file mapped to RAM and can be modified by changing memory addresses directly instead of outputting to a stream, shares same benefits as a standard file.	All POSIX systems, Windows

Overview of the iOS 6 Architecture

iOS consists of a number of different software layers, each of which provides programming frameworks for the development of applications that run on top of the underlying hardware. These operating system layers can be presented diagrammatically as illustrated in Figure 5-1:

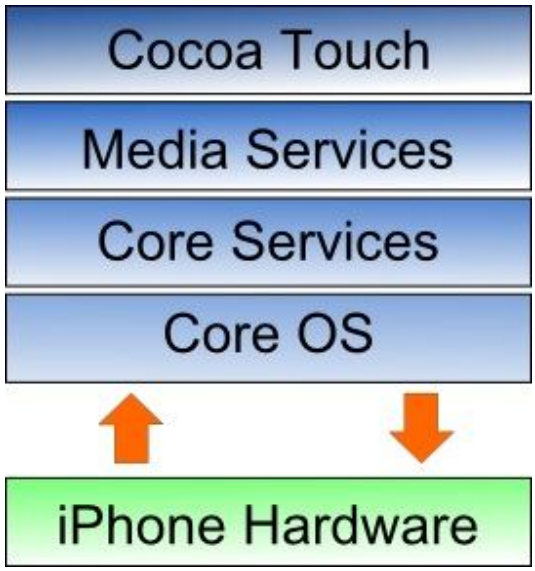


Figure 5-1

Some diagrams designed to graphically depict the iOS software stack show an additional box positioned above the Cocoa Touch layer to indicate the applications running on the device. In the above diagram we have not done so since this would suggest that the only interface available to the app is Cocoa Touch. In practice, an app can directly call down any of the layers of the stack to perform tasks on the physical device.

That said, however, each operating system layer provides an increasing level of abstraction away from the complexity of working with the hardware. As an iOS developer you should, therefore, always look for solutions to your programming goals in the frameworks located in the higher level iOS layers before resorting to writing code that reaches down to the lower level layers.

In general, the higher level of layer you program to, the less effort and fewer lines of code you will have to write to achieve your objective. And as any veteran programmer will tell you, the less code you have to write the less opportunity you have to introduce bugs.

Now that we have identified the various layers that comprise iOS 6 we can now look in more detail at the services provided by each layer and the corresponding frameworks that make those services available to us as application developers.

The Cocoa Touch Layer

The Cocoa Touch layer sits at the top of the iOS stack and contains the frameworks that are most commonly used by iPhone application developers. Cocoa Touch is primarily written in Objective-C, is based on the standard Mac OS X Cocoa API (as found on Apple desktop and laptop computers) and has been extended and modified to meet the needs of the iPhone hardware.

The Cocoa Touch layer provides the following frameworks for iPhone app development:

- UIKit Framework - The UIKit framework is a vast and feature rich Objective-C based programming interface.

- Map Kit Framework - The Map Kit framework provides a programming interface which enables you to build map based capabilities into your own applications. This allows you to, amongst other things, display scrollable maps for any location, display the map corresponding to the current geographical location of the device and annotate the map in a variety of ways.

- Push Notification Service -The Push Notification Service allows applications to notify users of an event even when the application is not currently running on the device. Since the introduction of this service it has most commonly been used by news based applications. Typically when there is breaking news the service will generate a message on the device with the news headline and provide the user the option to load the corresponding news app to read more details. This alert is typically accompanied by an audio alert and vibration of the device. This feature should be used sparingly to avoid annoying the user with frequent interruptions.

- Message UI Framework - The Message UI framework provides everything you need to allow users to compose and send email messages from within your application. In fact, the framework even provides the user interface elements through which the user enters the email addressing information and message content. Alternatively, this information may be pre-defined within your application and then displayed for the user to edit and approve prior to sending.

The iOS Media Layer

The role of the Media layer is to provide iOS with audio, video, animation and graphics capabilities. As with the other layers comprising the iOS stack, the Media layer comprises a number of frameworks which may be utilized when developing iPhone apps. In this section we will look at each one in turn.

- Core Video Framework -The Core Video Framework provides buffering support for the Core Media framework. Whilst this may be utilized by application developers it is typically not necessary to use this framework.

- Core Text Framework - The iOS Core Text framework is a C-based API designed to ease the handling of advanced text layout and font rendering requirements.

- Core Graphics Framework - The iOS Core Graphics Framework (otherwise known as the Quartz 2D API) provides a lightweight two dimensional rendering engine. Features of this framework include PDF document creation and presentation, vector based drawing, transparent layers, path based drawing, anti-aliased rendering, color manipulation and management, image rendering and gradients. Those familiar with the Quartz 2D API running on MacOS X will be pleased to learn that the implementation of this API is the same on iOS.

- Core Image Framework - A new framework introduced with iOS 5 providing a set of video and image filtering and manipulation capabilities for application developers.

The iOS Core Services Layer

The iOS Core Services layer provides much of the foundation on which the previously referenced layers are built and consists of the following frameworks.

- Address Book Framework - The Address Book framework provides programmatic access to the iPhone Address Book contact database allowing applications to retrieve and modify contact entries.

- CFNetwork Framework - he CFNetwork framework provides a C-based interface to the TCP/IP networking protocol stack and low level access to BSD sockets. This enables application code to be written that works with HTTP, FTP and Domain Name servers and to establish secure and encrypted connections using Secure Sockets Layer (SSL) or Transport Layer Security (TLS).

- Core Data Framework - This framework is provided to ease the creation of data modeling and storage in Model-View-Controller (MVC) based applications. Use of the Core Data framework significantly reduces the amount of code that needs to be written to perform common tasks when working with structured data within an application.

The iOS Core OS Layer

The Core OS Layer occupies the bottom position of the iOS stack and, as such, sits directly on top of the device hardware. The layer provides a variety of services including low level networking, access to external accessories and the usual fundamental operating system services such as memory management, file system handling and threads.

- Accelerate Framework - The Accelerate Framework provides a hardware optimized C-based API for performing complex and large number math, vector, digital signal processing (DSP) and image processing tasks and calculations.

- External Accessory Framework - Provides the ability to interrogate and communicate with external accessories connected physically to the iPhone via the 30-pin dock connector or wirelessly via Bluetooth.

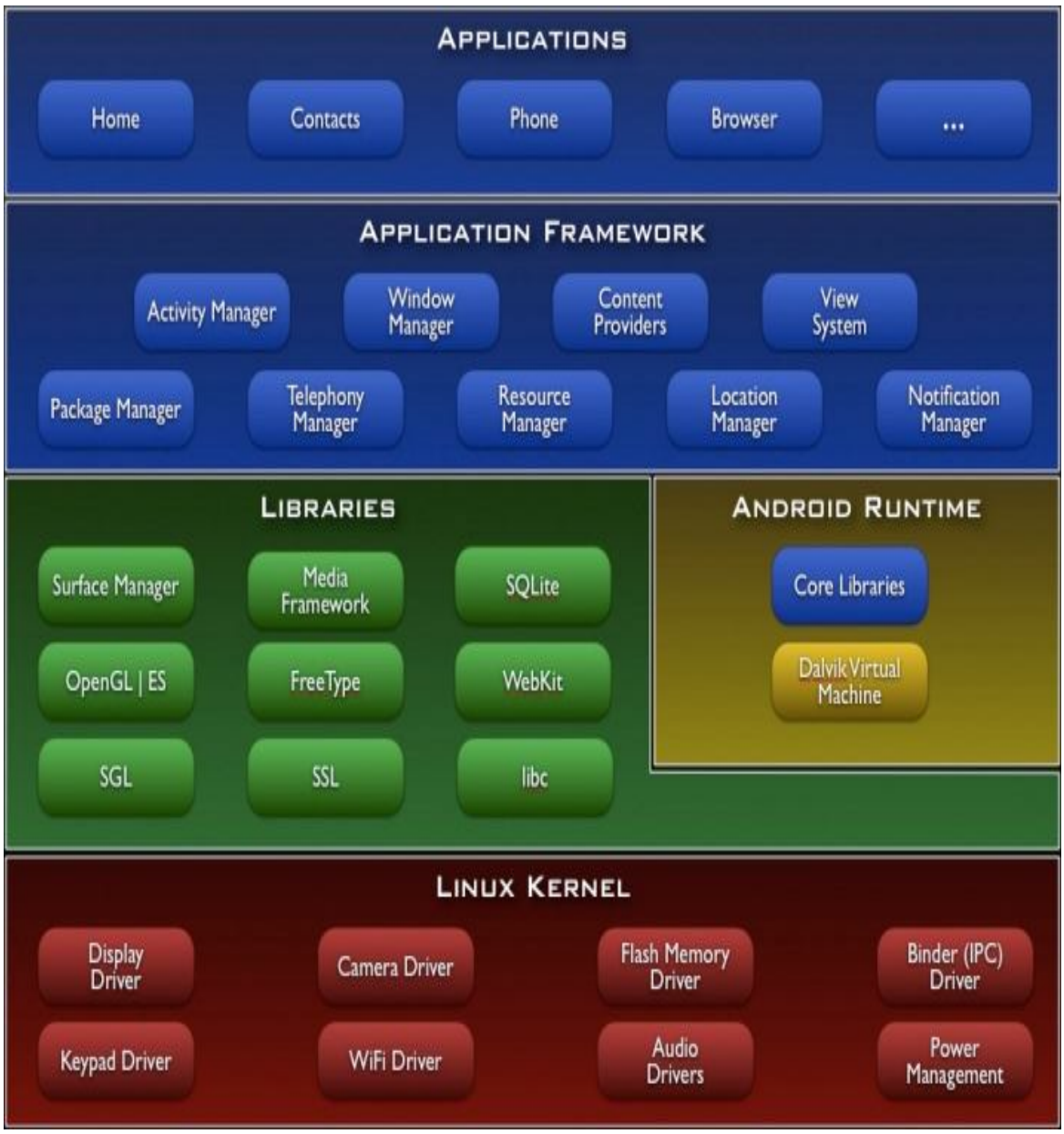
- Security Framework - The iOS Security framework provides all the security interfaces you would expect to find on a device that can connect to external networks including certificates, public and private keys, trust policies, keychains, encryption, digests and Hash-based Message Authentication Code (HMAC).

Android (SDK)

- ❖ Internally, Android uses Linux for its memory management, process management, networking, and other operating system services. The Android phone user will never see Linux or in other words our application will not make Linux calls directly.
- ❖ Android is built on top of a Linux kernel which provides a hardware abstraction layer for it for e.g. Bluetooth stack. This distribution makes Android easily portable to a variety of platforms.

- A layer consisting of C/C++ libraries are embedded on top of Linux kernel. Some of these libraries handle tasks related to graphics, multimedia codecs, database and browsers (WebKit).
- Android runtime has its own custom optimized java virtual machine called Dalvik VN. This is because of the reason that native Android Java libraries are different from standard Java or Java Mobile Edition (JME) libraries. JME is an adaptation of Java Standard Edition to allow Java to run on embedded devices such as mobile phones.
- Application framework provides various managers to access all hardware related services.
- Most of these managers come as bundle of Java classes. A detail overview about application framework is given below.
- Before we dive in to Application Framework layer, note that the Application layer is the tip of Android iceberg. This is where a user interacts with the phone while unaware of all the action taking place beneath this layer. Some applications are pre-installed on a phone while most are developed and placed on the Android market for free download or for a small price. Our objective is to develop a application and put it on the Android market for free download.

Fig. 1 Android Architecture Layer



Application Framework

Application Framework sits on top of native libraries, android runtime and Linux kernel. This framework come pre-installed with high-level building blocks that developers can use to program applications. Following are the most important application framework components for our application and Android development in general.

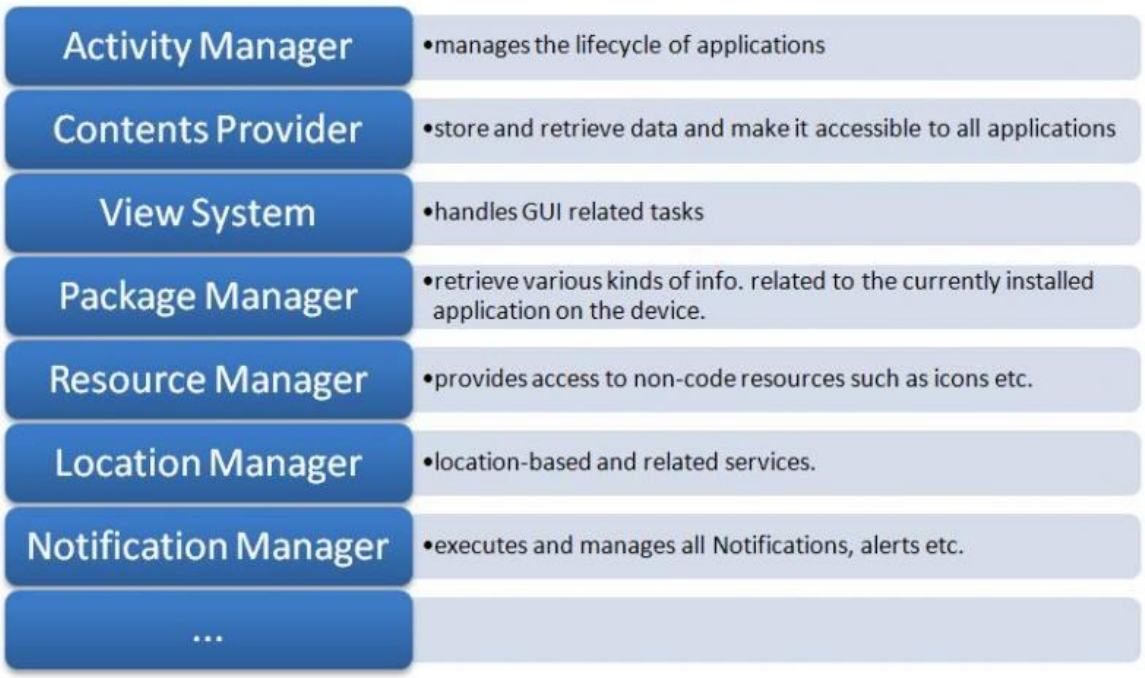


Fig. 2 Android Application Framewrok Bundles

Activity Manager

Activity is a single focused thing. Activities can run in the foreground giving direct interaction to the user e.g. current window/tab, they can run as background services or they can be embedded in other activities.

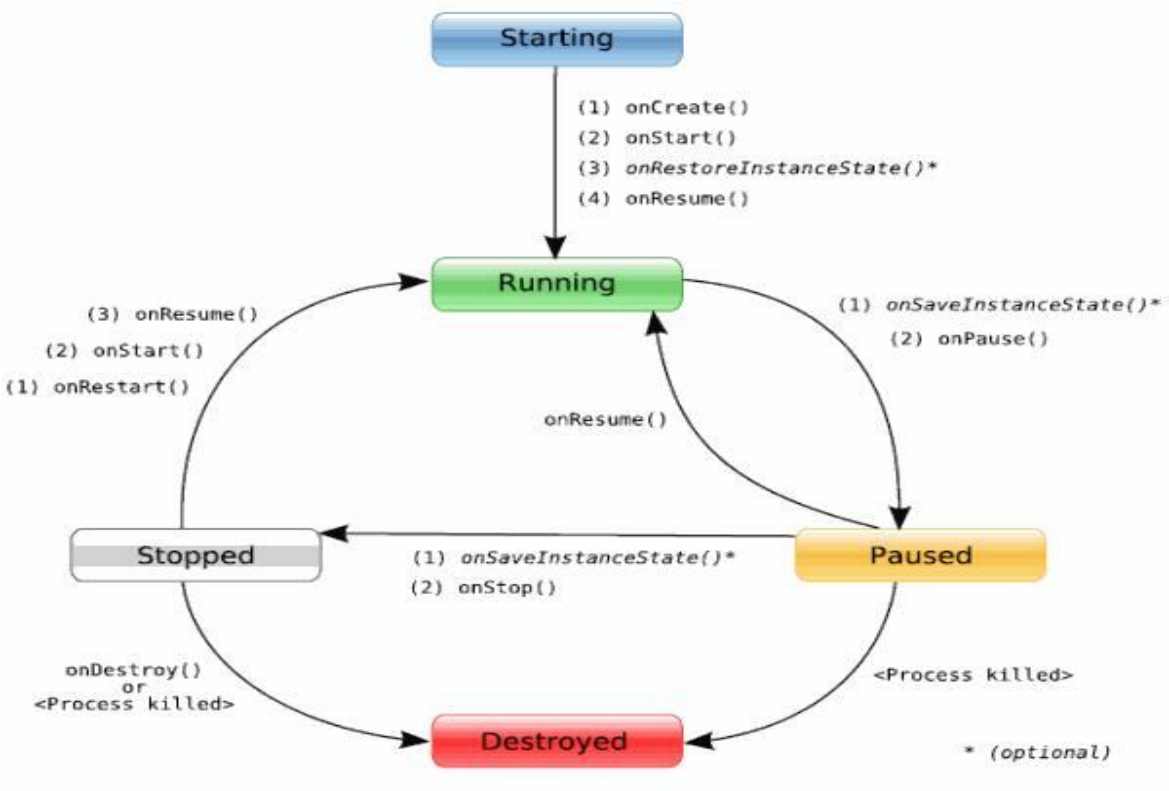


Fig 3. Android Activity Lifecycle

Contents Provider

Content provides handles data across applications globally. Android comes with a set of built in content providers to handle multi-media data or contacts etc. Developers can make up their own providers for flexibility or they can incorporate their data in one of the existing providers. For our specific application, we are interested in content://browser to access online data through browser interface.

View System

View system binds all the classes together that handle graphical user interface (GUI) related elements. All views elements are arranged in a hierarchical single tree manner. They can be called from a java code or included in XML layout files. One good thing that we noticed about Android development is extensive use of XML files. These files provide a nice abstraction

between backend java code and layout elements. Designing UI related elements is done in the same fashion as HTML based web designing.

Resource Manager

Resource Manager handles all non-code things. These can be anything ranged from icons, graphics or text. Such resources reside under res directory as can be seen under Eclipse Project Explore in the following figure4.

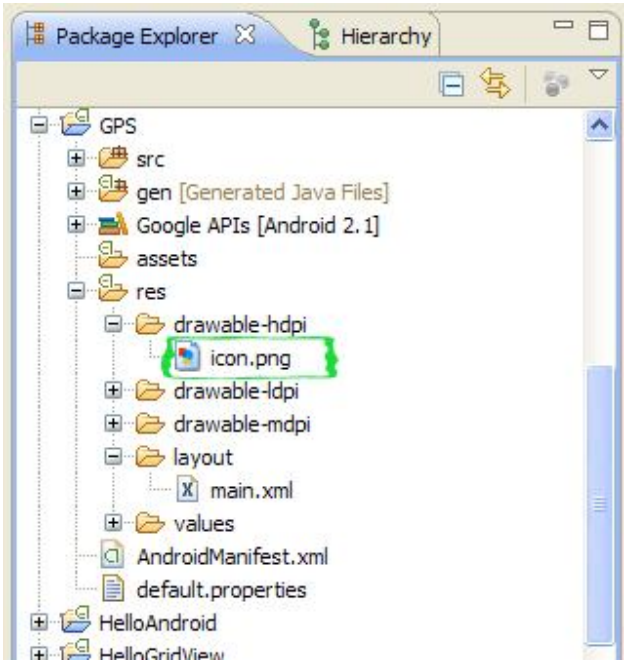


Fig. 4. Eclipse Project Explorer

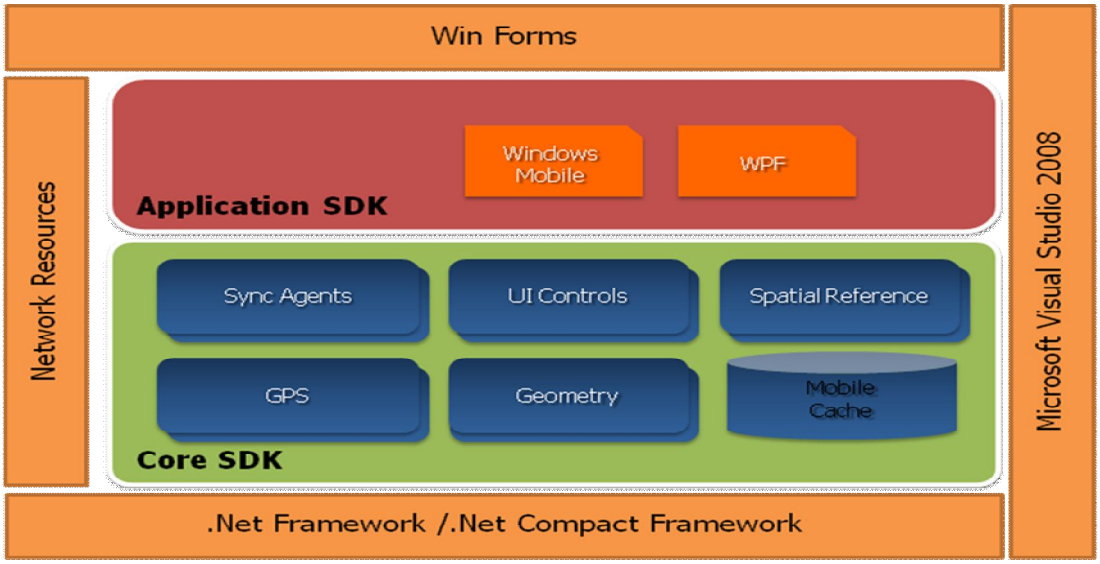
All the icons and design work that we have done so far using Adobe Illustrator will reside under these layout directories.

Location Services

This bundle supports fine-grained location providers such as GPS and coarse-grained location providers such as cell phone triangulation. LocationManager system service is the central component of the location framework. This system service provides an underlying API to access device location information. Besides LocationManager class, are several other important

classes from android.location that are important to location aware applications such as Geocoder, GpsSatellite and LocationProvider.

Core SDK and Application SDK



Application SDK

- Designed for ready-to-deploy tablet and Windows Mobile applications
- Allows developers to customize the applications
    - Changes existing tasks/functions
    - Integrates new business logic and Implementations