Enoch Hwang

✶ ehwang@cs.ucr.edu ✶

OS Notes[1]
June 24, 1996

1. Processes

- A *process* is a program in execution. It is the unit of work. A program by itself is not a process, it is a passive entity. A process is more than just the program code. It also includes the current activity, as represented by the value of the program counter, registers, and stack. A process is an active entity.
- A process changes *states* as it executes. The state of a process is defined in part by the current activity of that process. Each process may be in one of the following states:
  - *New*. The process is being created.
  - *Running*. Instructions are being executed.
  - *Waiting*. The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
  - *Ready*. The process is waiting to be assigned to a processor.
  - *Terminated*. The process has finished execution.
  - Each process is represented in the OS by a *process control block* (PCB) which contains information such as: process state, program counter, registers, CPU scheduling, memory-management, accounting, and I/O status information.
- *Task* is the fundamental unit to which resources are allocated. Each task has its own address space and open file table.
  - A task with one thread is also known as *heavy-weight process*.
  - A task can have one or more threads.
- *Thread* (*light-weight* proces) is the fundamental unit of concurrent activity (CPU utilization). It is the entity to which CPUs are allocated. Each thread has its own stack, program counter, and registers. But they can share the same logical address space (i.e. both code and data).
- *Process state* can be *new, running, waiting, ready, or terminated*.
- *Process control block* contains the information regarding a process such as process state, program counter, & register contents.
- *Schedulers* - *long* & *short* term schedulers.

Context switch - switch CPU from one process to another.

2. CPU Scheduling

- *Multiprogramming* - when one process has to wait (for I/O), the OS takes the CPU away from that process and gives the CPU to another process.
- The selection process is carried out by the *short-term* scheduler under the following four circumstances:
  1. When process switches from *running* state to *waiting* state. (eg I/O request)
  2. When process switches from *running* state to *ready* state. (eg when an interrupt occurs)
  3. When process switches from *waiting* state to *ready* state. (eg completion of I/O)
  4. When a process terminates.

[1] Silberschatz & Galvin, Operating System Concepts, Addison-Wesley, 1994, 4th Ed.

- *Preemptive* (2 & 3) and *non-preemptive* (1 & 4) scheduling.
- *Dispatcher* - the module that gives control of the CPU to the process selected by the short-term scheduler. This function involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
  - *Dispatch latency* - the time it takes for the dispatcher to stop one process and start another running.
- **Scheduling criteria**
  - CPU utilization - how busy the CPU is.
  - Throughput - number of processes that are completed per time unit.
  - Turnaround time - how long it takes to execute that process - from start to finish.
  - Waiting time - sum of the periods spent waiting in the ready queue. The CPU scheduling algorithm does not affect the amount of time during which a process executes or does I/O; it affects only the amount of time that a process spends waiting in the ready queue.
  - Response time - the time from the submission of a request until the first response.
- **Scheduling algorithms**
  - ***First-come-first-served***. The CPU is allocated to the first process that requests for it.
    - Non-preemptive.
  - ***Shortest-job-first***. CPU is assigned to the process that has the smallest next CPU burst. FIFS scheduling is used to break the tie.
    - is provably optimal; gives the minimum average waiting time.
    - cannot be implemented at the level of short-term CPU scheduling. There is no way to know the length of the next CPU burst. Can predict its value by using previous ones.
    - can be either preemptive or non-preemptive. Preemptive SJF are also call *shortest-remaining-time-first* scheduling. A new process arrives while a previous process is executing. The new process may have a shorter next CPU burst than what is left of the currently executing process.

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

Preemptive:

P_1 P_2-----P_4-------- P_1---------- P_3----------

0 1 5 10 17 26

Avg waiting time = (10-1)+(1-1)+(17-2)+(5-3)=6.5

Non-preemptive:

P_1---------- P_2------P_4---------P_3----------

0 8 12 17 26

Avg waiting time = 0+(8-1)+(17-2)+(12-3)= 7.75

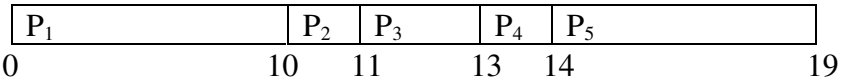
  - ***Priority scheduling***. A priority is associated with each process, and the CPU is allocated to the process with the highest priority.
    - An SJF algorithm is simply a priority algorithm where the priority is the inverse of the predicted next CPU burst.
    - problem is indefinite blocking or starvation; if there is a low priority job waiting and there is a steady stream of higher-priority jobs. Solution - *aging* - gradually increasing the priority of processes that wait in the system for a long time.
  - ***Round-robin scheduling***. Similar to FCFS, but preemption is added to switch between processes. Designed for time-sharing systems.

    - Each process is given one time *quantum*. If a process' CPU burst exceeds 1 time quantum, that process is preempted and is put back in the ready queue.
    - If the time quantum is infinite, then RR is the same as the FCFS policy. If the quantum is very small, then it is call processor sharing, and appears (in theory) to the users as though each of *n* processes has its own processor running at 1/*n* the speed of the real processor.
    - need also to consider the effect of context switching.
  - ***Multilevel queue scheduling*** partitions the ready queue into several separate queues. Processes are permanently assigned to one queue. Each queue has its own scheduling algorithm.
    - There must be scheduling between the queues, i.e. one queue must have a higher priority than another queue. Can also time slice between the queues, eg. the foreground queue gets 80% of the CPU time while the background queue gets 20%.
  - ***Multilevel feedback queue scheduling***. Like above but processes are allowed to move between queues.
- ***Multiple-Processor scheduling***. Scheduling for a system that can have more than one CPU. The CPUs can be identical (homogeneous) in terms of their functionality or different (heterogeneous).
- ***Real-time scheduling***. Processes are required to complete a critical task within a guaranteed amount of time.
  - The *dispatch latency* must be small. The problem is that many OSs are forced to wait for either a system call to complete or for an I/O block to take place before doing a context switch. The dispatch latency in such systems can be long, since some system calls are complex and some I/O devices are slow.

- Question 5.3: Given 5 processes all to have arrived at time 0 in the order P_1, P_2, P_3, P_4, P_5.

Process	Burst Time	Priority (hi=1, 4=low)
P_1	10	3
P_2	1	1
P_3	2	3
P_4	1	4
P_5	5	2

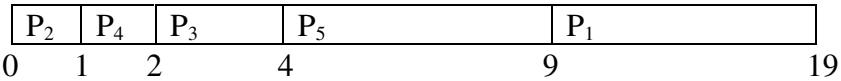
a) Gantt chart for FCFS



0 10 11 13 14 19

Average wait time: (0 + 10 + 11 + 13 + 14) / 5 = 9.6

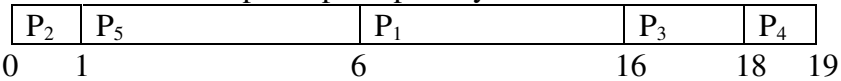
Gantt chart for SJF



0 1 2 4 9 19

Average wait time: (9 + 0 + 2 + 1 + 4) / 5 = 3.2

Gantt chart for nonpreemptive priority



0 1 6 16 18 19

Average wait time: (6 + 0 + 16 + 18 + 1) / 5 = 8.2

Gantt chart for RR (quantum = 4)

P_1	P_2	P_3	P_4	P_5	P_1	P_5	P_1

0 4 5 7 8 12 16 17 19

Average wait time: (9 + 4 + 5 + 7 + 12) / 5 = 7.4

3. Process Synchronization

A *cooperating* process is one that can affect or be affected by the other processes executing in the system. Cooperating processes may either directly share a logical address space (i.e., *threads* that share both code and data), or be allowed to share data only through files. Concurrent access to shared data may result in data inconsistency. Look at various mechanisms to ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained.

- **Critical section problem** must satisfy the following 3 requirements:
  1. Mutual Exclusion - only one process can be executing its critical section.
  2. Progress - if no process is executing in the CS and there is a process that wish to enter the CS, then this process should be allowed to enter the CS. i.e. no deadlock.
  3. Bounded waiting - there must be a bound as to how long a process must wait after making a request to enter its CS before that request is granted. i.e. no starvation.

- Algorithm for 2 processes that solves the critical-section problem where all of the 3 requirements above are met for process P_i:

```
var flag: array[0..1] of boolean;  // init to false;
    turn: 0..1;                    // init to 0 or 1;
repeat
    flag[i] := true;
    turn := j;
    while (flag[j] and turn = j) do noop;
    critical section
    flag[i] := false;
until false;
```

- ***Bakery* algorithm** - Multiple-Process solution for process P_i

```
var choosing: array [0..n-1] of boolean;  initialize to false
    number: array[0..n-1] of integer;     initialize to 0

repeat
    choosing[i] := true;
    number[i] := max(number[0], number[1], ..., number[n-1]) + 1;
    choosing[i] := false;
    for j := 0 to n-1 do
        begin
        while choosing[j] do noop;
        while number[j] ≠ 0 and (number[j], j) < (number[i], i) do noop;
        end;

    critical section

    number[i] := 0;
until false;
```

- Synchronization hardware

Test-and-Set

```
function Test-and-Set (var lock: boolean): boolean;
    Test-and-Set := lock;
    lock := true;

usage:
    initially lock := false;
    repeat
        while Test-and-Set(lock) do noop;
        critical section
        lock := false;
    until false;
```

Swap

```
procedure Swap (var a, b: boolean);
    var temp: boolean;
    begin
        temp := a;
        a := b;
        b := temp;
    end;
```

```
usage: (do not satisfy the bounded-waiting)
    initialize lock := false;
    repeat
        key := true;
        repeat
            Swap(lock, key);
        until key = false;
            critical section
        lock := false;
    until false;
```

Bounded-waiting mutual exclusion with Test-and-Set

```
var waiting: array[0..n-1] of boolean = false;
    lock: boolean = false;
repeat
    waiting[i] := true;
    key := true;
    while waiting[i] and key do key := Test-and-Set(lock);
    waiting[i] := false;
        critical section
    j := i+1 mod n;
    while (j ≠ i) and (not waiting[j] do j := j + 1 mod n;
    if j = i then lock := false;
    else waiting[j] := false;
        remainder section
until false;
```

- ***Semaphore*** is an integer variable that, apart from initialization, is accessed only through two *atomic* operations:

```
P(S)    (proberen in Dutch, to wait):
        if S ≥ 1 then S := S - 1
        else block the process on the semaphore S queue;
V(S)    (verhogen in Dutch, to increment):
        if there is a process blocked on the semaphore S then unblock the process
        else S := S + 1;
second version:
wait(S) S --;
        if S < 0 then block process on the semaphore S queue
signal(S)
        S ++;
        if S ≤ 0 then wakeup process on the semaphore S queue
usage:
    mutex := 1;
```

```
repeat
    P(mutex);
        critical section
    V(mutex);
until false;
```

- S is initialize to 1 for a *binary semaphore*. *Binary semaphore* is one where the semaphore value can only be either 0 or 1.
- *Counting semaphores* can have any integer values.

Using binary semaphores to implement a counting semaphore *S*.

```
var S1: binary-semaphore;  // init to 1
    S2: binary-semaphore;  // init to 0; the P and V for S.
    S3: binary-semaphore;  // init to 1
    C: integer             // init to value of S; the count for S.
P(binaryS)  (proberen in Dutch, to wait):
    wait(S3);              // just to serialize the entire procedure
    wait(S1);              // serialize the critical section on the variable C
    C := C - 1;
    if C < 0 then
        signal(S1);  wait(S2);
    else signal(S1);
    signal(S3);
V(binaryS)  (verhogen in Dutch, to increment):
    wait(S1);
    C := C + 1;
    if C ≤ 0 then signal(S2);
    signal(S1);
```

- **Deadlock & Starvation**
  - a set of processes is in a *deadlock* state when every process in the set is waiting for an event that can be caused only by another process in the set.
  - *Starvation* is a situation where processes wait indefinitely within the semaphore.

- *Bounded Producer-Consumer problem*

```
empty := n;  full := 0;  mutex := 1;
producer:                                   consumer:
repeat                                      repeat
    produce an item in nextp;
    P(empty);                                   P(full);
    P(mutex);                                   P(mutex);
    add nextp to buffer;                        remove an item from buffer to nextc;
    V(mutex);                                   V(mutex);
    V(full);                                    V(empty);
                                                consume the item in nextc;
until false;                                until false;
```

- *Readers and Writers problem.* no reader will be kept waiting unless a writer has already obtained permission to use the shared object.

```
var mutex, wrt: semaphore = 1;              reader:
    readcount : integer = 0;                    wait(mutex);
                                                    readcount++;
writer:                                             if readcount = 1 then wait(wrt);
    wait(wrt);                                  signal(mutex);
        ...                                         ...
        writing is performed                        reading is performed
```

```
    ...                                     ...
signal(wrt);                            wait(mutex);
                                            readcount--;
                                            if readcount = 0 then signal(wrt);
                                        signal(mutex);
```

- *Dining philosophers problem*. Possible solutions:
  - ☑ Allow at most *n*-1 philosophers to be sitting simultaneously at the table.
  - ☑ Allow a philosopher to pick up her forks only if both forks are available.
  - ☑ Use an asymmetric solution; i.e. an odd philosopher picks up first her left fork and then her right fork, whereas an even philosopher picks up her right fork and then her left fork.
  - ☒ Number the forks. A philosopher must get the lower number fork before getting the higher number fork (in modulo).

- Implementation of the monitor mechanism (given by Hoare and Hansen) using semaphores.

```
var mutex : semaphore = 1;              procedure F:
    next : semaphore = 0;                   wait(mutex);
    next-count : integer = 0;                   ...
    x-sem : semaphore = 0;                      body of F;
    x-count : integer = 0;                      ...
                                            if next-count > 0
                                                then signal(next);
                                                else signal(mutex);

x.wait:                                 x.signal:
    x-count ++;                             if x-count > 0 then
    if next-count > 0                           next-count ++;
        then signal(next);                      signal(x-sem);
        else signal(mutex;                      wait(next);
    wait(x-sem);                                next-count --;
    x-count --;
```

- ***Critical Regions (conditional critical regions)*** A variable *v* of type T is to be shared among many processes. *v* can be accessed only inside a *region* statement when the condition is true:

  var *v*: shared T;
  region *v* when *condition* do *statement*;

  - This means that while *statement* is being executed, no other process can access the variable *v*. The *condition* governs the access to the critical region. When a process tries to enter the critical-section region, the *condition* is evaluated.
  - Example for the bounded producer-consumer problem:

```
var buffer: shared record
                pool: array[0..n-1] of item;
                count, in, out: integer;
            end;

producer process
region buffer when count < n do
    begin
    pool[in] := nextp;
    in := in + 1 mod n;
    count := count + 1;
    end;

consumer process
```

```
region buffer when count > 0 do
    begin
    nextc := pool[out];
    out := out + 1 mod n;
    count := count - 1;
    end;
```

- ***Monitors*** are abstract data types for defining shared objects and for scheduling their accesses. A monitor consists of procedures that operate on the shared object. Constraints:
  - Only one process can be active (i.e. executing a procedure) within the monitor at a time.
  - Procedures of a monitor can only access data local to the monitor.
  - The data local to the monitor can only be accessed by the monitor procedures.
  - Three special operations:
    1. condition_variable.**wait** suspends the caller process.
    2. condition_variable.**signal** cause a waiting process to regain control of monitor
    3. condition_variable.**queue** returns true if that particular queue is not empty.
  - Error if a process does not obey the following rules:
    - Access the resource without first gaining access permission to that resource.
    - Never release the resource once it has been granted access to that resource.
    - Attempt to release a resource that it never requested.
    - Request the same resource twice (without first releasing that resource).
  - Examples:

```
type readers-writers = monitor;
var readercount : integer;  busy : boolean;  Oktoread, Oktowrite : condition;

procedure startread;
    if busy then Oktoread.wait;
    readercount := readercount + 1;
    Oktoread.signal;

procedure endread;
    readercount := readercount - 1;
    if readercount = 0 then OKtowrite.signal;

procedure startwrite;
    if busy or readercount ≠ 0 then
        Oktowrite.wait;
    busy := true;

procedure endwrite;
    busy := false;
    if Oktoread.queue then Oktoread.signal;
    else Oktowrite.signal;

begin    // initialization for the monitor
    readercount := 0;
    busy := false;
end;
```

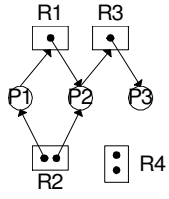
  - Monoitor solution to the dining philosophers problem. Starvation can occur in the example:

```
type dining-philosophers = monitor
var state: array[0..4] of (thinking, hungry, eating);
    self: array[0..4] of condition;

procedure pickup(i: 0..4);
    state[i] := hungry;
    test(i);
    if state[i] ≠ eating then self[i].wait;

procedure putdown(i: 0..4);
    state[i] := thinking;
    test(i+4 mod 5);
    test(i+1 mod 5);

procedure test(k: 0..4);
    if state[k+4 mod 5] ≠ eating
```

```
    and state[k] = hungry
    and state[k+1 mod 5] ≠ eating then
        begin
        state[k] := eating;
        self[k].signal;
        end;

begin   // initialization for the monitor
    for i := 0 to 4 do
        state[i] := thinking;
end;
```

4. Deadlocks

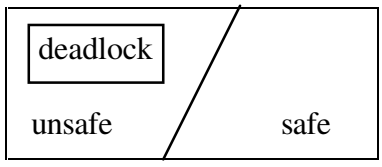
- A set of processes is in a *deadlock* state when every process in the set is waiting for an event that can be caused by only another process in the set.
- Four necessary conditions for deadlock; all four conditions must hold for a deadlock to occur:
  1. ***Mutual exclusion***. At least one resource must be held in a nonsharable mode.
  2. ***Hold and wait***. There must exist a process that is holding at least one resource and is waiting to acquire additional resources that are currently being held by other processes.
  3. ***No preemption***. Resources cannot be preempted by another process; i.e. a resource can be released only voluntarily by the process holding it.
  4. ***Circular wait***. There must exist a set $\{P_0, P_1, ..., P_n\}$ of waiting processes such that P_0 is waiting for P_1, ..., P_n is waiting for P_0.
- A *system resource-allocation graph*.



  No deadlock if the graph has no cycle. Deadlock may exist if the graph contains a cycle.
  - If each resource type in the cycle has exactly one instance then a cycle in the graph is both a necessary and a sufficient condition for the existence of a deadlock.
  - If each resource type in the cycle has several instances, then a cycle does not necessarily imply that a deadlock occurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of a deadlock.
- Methods for handling deadlocks
  1. Use a protocol to ensure that the system will never enter a deadlock state.
  - ***Deadlock prevention***. Ensuring that at least one of the above four necessary conditions cannot hold. Disadvantages: low device utilization and reduced system throughput.
    - No mutual exclusion. In general, it is not possible to prevent deadlocks by denying the mutual-exclusion condition; some resources are intrinsically nonsharable.
    - No hold and wait. Two possible protocols:
      - A process must acquire all the needed resources before starting.
      - A process can request resources only when it has none.
    - Preemption protocol.
      - 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 preempted. The process will restart only when it can regain its old resources, as well as the new one that it is requesting.
    - No circular-wait protocol.

    - Assign to each resource a unique integer number. A process can request resources only in an increasing order. If the process requests for a lower number resource, it must first release all the higher number resources.
- ***Deadlock avoidance***. A deadlock avoidance algorithm dynamically examines the resource allocation state to ensure that there can never be a circular wait condition. The resource allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes.
    - Requires that the OS be given in advance additional information concerning which resources a process will request and use during its lifetime. For each request, the OS must consider:
        - the resources currently available,
        - the resources currently allocated to each process, and
        - the future requests and releases of each process.

    The OS can then decide whether the request can be satisfied or must be delayed.
    - The system is in a s*afe state* if it can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. A sequence of processes $<P_1, P_2, ..., P_n>$ is a *safe sequence* for the current allocation state if, for each P_i, the resources that P_i can still request can be satisfied by the currently available resources plus the resources held by all the P_j, with $j < I$.
    - A safe state is not a deadlock state. Conversely, a deadlock state is an unsafe state. But not all unsafe states are deadlocks. An unsafe state may lead to a deadlock.



    - Note that, if a process requests a resource that is currently available, it may still have to wait. Thus, resource utilization may be lower than it would be without a deadlock avoidance algorithm.
    - Resource-allocation graph algorithm. with request-edge, assignment-edge, and claim-edge. A cycle indicates an unsafe state. Not applicable to multiple instances of a resource type.
    - Banker's algorithm.

Available[j]:	Number of available resources of each type.
Max[i,j]:	Maximum demand of each process.
Allocation[i,j]:	Number of resources of each type currently allocated to each process.
Need[i,j]:	Remaining resource need of each process. *Need = Max - Allocation*.

Safety:
    Work := *Available*;
    Finish := false;
2 find an *i* such that both
        Finish[*i*] = false and
        Need[*i*] ≤ *Work*.
    If no such *I* exists, goto step 4.
3 *Work* := *Work* + *Allocation*[*i*];
    Finish[*i*] := true;
    goto step 2
4 If *Finish*[*i*] = true for all *I*, then the system is in a safe state.

Resource-Request:
1 If *Request*[*i*] ≤ *Need*[*i*] then goto step 2 else error
2 If *Request*[*i*] ≤ *Available*[*i*] then goto step 3 else P_i must wait.

3 Pretend to have allocated the requested resources to process P_i by modifying the following
Available := *Available* - *Request*[*i*]
Allocation[*i*] := *Allocation*[*i*] + *Request*[*i*]
Need[*i*] := *Need*[*i*] - *Request*[*i*]
If the resulting state is safe, the transaction is completed and process P_i is allocated its resources. However, if the new state is unsafe, then P_i must wait.

- Example of Banker's algorithm:

		Allocation			Max			Need			Available		
T_0	P_0	0	1	0	7	5	3	7	4	3	3	3	2
	P_1	2	0	0	3	2	2	1	2	2			
	P_2	3	0	2	9	0	2	6	0	0			
	P_3	2	1	1	2	2	2	0	1	1			
	P_4	0	0	2	4	3	3	4	3	1			

Both Need(P_1) and Need(P_3) < Available, therefore, both P_1 and P_3 can go. After P_1 and P_3, Available=(4,6,5). The sequence <P_1,P_3,P_4,P_2,P_0> satisfies the safety criteria.

T_1 Now P_1 request (1,0,2). First check (1,0,2) ≤ (3,3,2)? Yes. Therefore, Allocation(1)=(3,0,2), Need(1)=(0,2,0), Available=Available-Request = (2,3,0). Still safe? Yes, sequence <P_1,P_3,P_4,P_0,P_2>. Therefore, can grant P_1's request.

T_2 P_4 request (3,3,0). Cannot be granted because (3,3,0) > Available=(2,3,0).

T_3 P_0 request (0,2,0). Cannot be granted because unsafe. Allocation(0)=(0,3,0), Need(0)=(7,2,3), Available=Available-Request = (2,1,0).

2. Allow the system to enter a deadlock state and then recover from it.

- ***Deadlock detection*** two cases:
  1. Single instance of each resource type.
     - Use a *wait-for* graph. Nodes are the processes. An edge from P_i to P_j implies that process P_i is waiting for process P_j to release a resource that P_i needs.
     - A deadlock exists if and only if the wait-for graph contains a cycle.
  2. Several instances of a resource type
     - Similar to Banker's algorithm.

Available[j]: Number of available resources of each type.
Allocation[i,j]: Number of resources of each type currently allocated to each process.
Request[i,j]: Current request of each process.

1 *Work* := Available;
if *Allocation*[*i*] ≠ 0 then *Finish*[*i*] := false else *Finish*[*i*] := true;
2 find an *i* such that both
Finish[*i*] = false and
Request[i] ≤ *Work*.
If no such *I* exists, goto step 4.
3 Work := Work + Allocation[i];
Finish[*i*] := true;
goto step 2
4 If *Finish*[*i*] = false for some *I*, then the system is in a deadlock state.

- ***Deadlock recovery***
  - process termination
    - abort all deadlocked processes.
    - abort one process at a time until the deadlock cycle is eliminated.

- resource preemption
  - selecting which process and which resource.
  - rollback
  - starvation

3. Ignore the problem all together and pretend that deadlocks never occur in the system. e.g. Unix.

- Models of deadlock
  - Single-unit request model. A process is restricted to requesting only one unit of resource at a time.
  - The AND request model. A process is blocked until it is granted all of the requested resources. Deadlock if there is a **cycle** in the wait-for-graph.
  - The OR request model. A process is blocked until it is granted any one of the requested resources. Deadlock if there is a **knot** in the wait-for-graph.
  - The AND-OR request model. ex. R_1 and (R_2 or R_3). Deadlock if there is a knot in the wait-for-graph.
  - The P-out-of-Q request model. Process requests for Q resources and is blocked until it is granted any P of those resources. Deadlock if there is a knot in the wait-for-graph.
- General resource graph

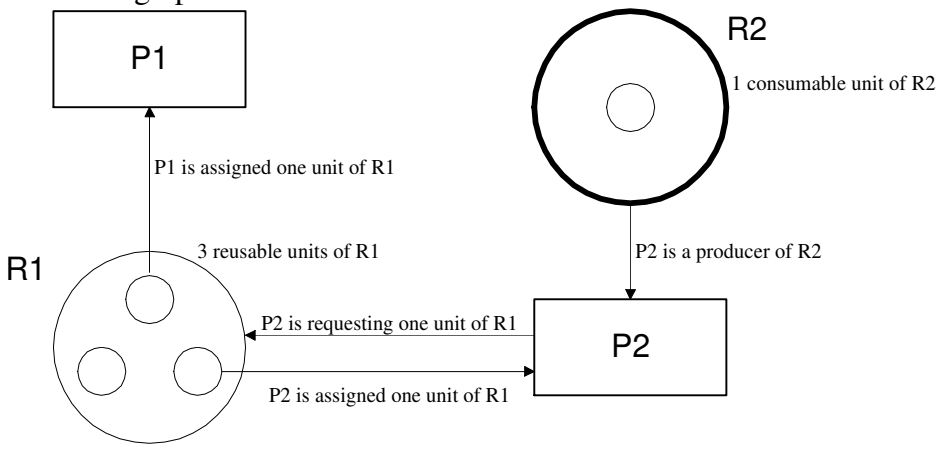


Figure 1. General resource graph.

- Graph reduction method.
  1. Delete all the reusable resource edges. For each assignment edge (R_j , P_i) deleted, increase r_j by one.
  2. For each consumable resource R_j ,
     - Decrement x_i by the number of assignment edges.
     - Set r_j to ∞ if there is a producer for it.

5. Memory Management

5.1 Address binding

- symbolic address in source program.
- the compiler binds the symbolic addresses to relocatable addresses.
- the linker or loader binds the relocatable addresses to absolute addresses.
- binding can occur during: compile time, load time, or execution time.

- Dynamic loading - a routine is not loaded until it is called. All routines are kept on disk in a relocatable load format. Does not require help from the OS.
- Dynamic linking - linking is postponed until execution time. With dynamic linking, all processes that use a library routine execute only one copy of the library code. Without dynamic linking, all these processes will need to have their own copy of the library code. Requires help from the OS.
- Overlays - keep in memory only those instructions and data that are needed at any given time. Does not require help from the OS.

5.2 Logical (Virtual) Address versus Physical Address Space

- an address generated by the CPU is referred to as a *logical* or *virtual address*.
- an address seen by the memory unit is referred to as a *physical address*.
- the logical & physical addresses are the same in the compile-time and load-time address-binding schemes. However, they are different in the execution-time address-binding scheme.
- *logical address space* - the set of all logical addresses generated by a program.
- *physical address space* - the set of all physical addresses generated by a program.
- the run-time mapping from virtual to physical addresses is done by the *memory-management unit*.
  - the relocation (base) register + logical address = physical address

5.3 Swapping

- a process is swapped temporarily out of memory to a disk.
- if address binding is done at compile or load time, then the process must be swapped back to the same memory space. If execution-time address binding is used, then it is possible to swap a process into a different memory space because the physical addresses are computed during execution time.
- a process must be completely idle before swapping, eg. cannot swap a process with pending I/O, otherwise, the I/O operation might attempt to use memory that now belongs to another process.

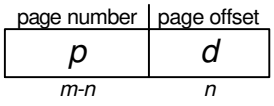
5.4 Contiguous Allocation

- partitioning of memory
- single-partition allocation
  - OS in one partition and user processess in another partition.
  - use a relocation register and a limit register. The relocation register contains the value of the smallest physical address; the limit register contains the range of logical addresses. Each logical address must be less than the limit register.
- multiple-partition allocation
  - either fixed or variable size partitions. Have a list of holes.
  - the dynamic storage allocation problem: first-fit, best-fit, and worst-fit.
  - *external fragmentation* - enough total memory space exists to satisfy a request, but it is not contiguous; i.e. large number of small holes.
    - solution - compaction. Possible only if relocation is dynamic and is done at execution time.

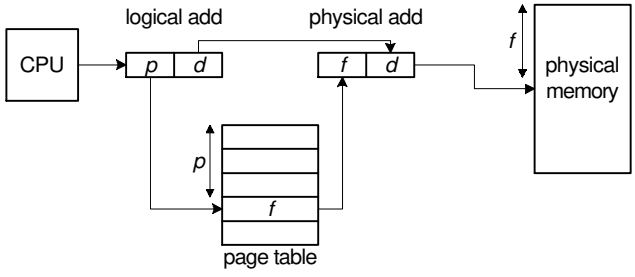
- another solution - use paging, i.e. permit the logical address space of a process to be noncontiguous.
- *internal fragmentation* - wasted memory that is internal to a partition.

5.5 Paging

- permit the logical address space of a process to be noncontiguous.
- physical memory is broken into fixed-sized blocks called *frames*. Logical memory is also broken into blocks of the same size called *pages*.
- The address generated by the CPU is divided into two parts: a *page number* (*p*) and a *page offset* (*d*). The page number is an index into a *page table* which contains the *base address* (*b*) of each frame in physical memory. The base address is combined with the page offset to define the physical memory address.

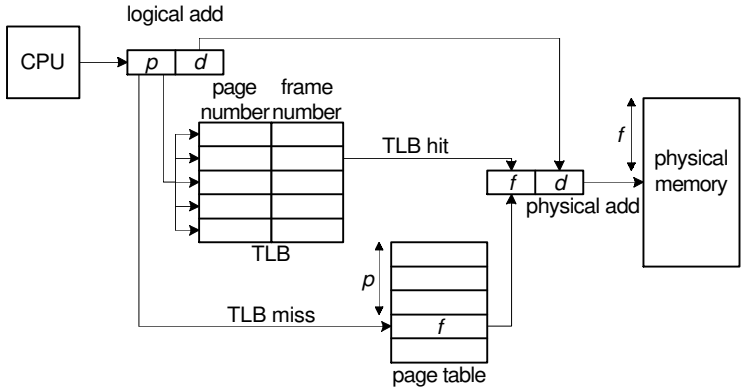


- logical address space is 2^m
- page size is 2^n

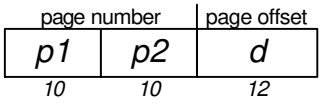


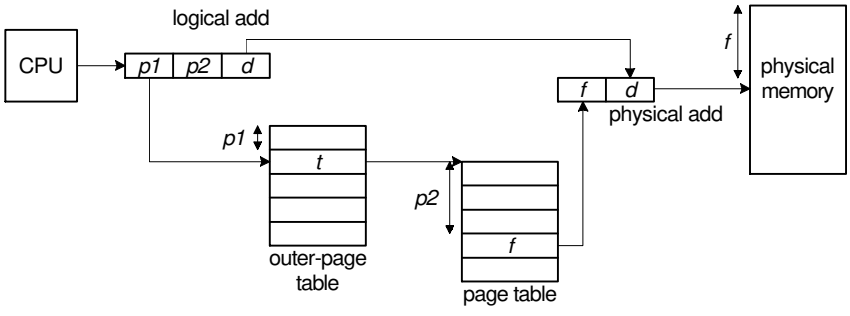
- no external fragmentation but may have internal fragmentation.
- The OS is managing physical memory, it must be aware of the allocation details of physical memory: which frames are allocated, which frames are available, how many total frames there are, etc. This information is generally kept in a data structure called a *frame table*. The frame table has one entry for each physical page frame, indicating whether the frame is free or allocated and, if it is allocated, to which page of which process.
- each process has its own page table. A pointer to the page table is stored with the other register values in the process control block.
- Hardware support of the page table:
  - in the simplest case, the page table is implemented as a set of dedicated registers. Changing the page table requires reloading these registers.
  - if the page table is large, it is stored in memory instead of registers, and a *page-table base register* points to the page table in memory. With this method, changing page tables requires changing only this one register, thus reducing context-switch time.
  - The standard solution is to use *associative registers* or *translation look-aside buffers* (TLB). Each register consists of two parts: a key and a value. When the associative registers are presented with an item, it is compared with all keys simultaneously. If the item is found, the corresponding value field is output.
  - When a logical address is generated by the CPU, its page number is presented to a set of associative registers that contain page numbers and their corresponding frame numbers.
    - If the page number is found in the associative registers, its frame number is immediately available and is used to access memory. (1 memory read).
    - If the page number is not in the associative registers, a memory reference to the page table must be made (1) to obtain the frame number. When the frame number is obtained, we can access memory to get the data (2). (2 memory reads).

- Every time a new page table is selected (for instance, each context switch), the TLB must be flushed (erased) to ensure that the next executing process does not use the wrong translation information.



- Memory protection in a paged environment is accomplished by protection bits that are associated with each frame. One bit can define a page to be read and write or read-only.
- A *valid - invalid* bit is also generally attached to each entry in the page table. This bit, when set to *valid*, indicates that the associated page is in the process's logical address space.
- A *page-table length register* (*PTLR*) is used to indicate the size of the page table.
- **Multilevel paging**. When the logical address space is very large, the page table itself becomes large. The solution is to use a two-level paging scheme in which the page table itself is also paged.





- **Inverted Page Table**. Usually, each process has a page table associated with it. The page table has one entry for each page that the process is using. Problem is the page tables may consume large amounts of memory. Solution is to use an *inverted page table*. An inverted page table has one entry for each real page (frame) of memory. Each entry consists of the logical address of the page stored in that frame (real memory location).
  - the whole table might need to be searched for a logical address match.
- Another advantage of paging is the possibility of sharing common code. The code needs to be reentrant. e.g. one copy of the editor code can be shared between several processes.

5.6 Segmentation

- A logical address space is a collection of segments. Addresses specify both the segment *number* and the *offset* within the segment.
- Use a *segment table*. Each entry has a segment *base* and a segment *limit*. The segment base is the starting physical address where the segment resides in memory. The segment limit specifies the length of the segment. The segment number is the index into the

segment table. A *segment-table base register (STBR)* points to the segment table and a *segment-table length register (STLR)* specifies the length of the segment table.

- Given a logical address <seg number, seg offset>.
  - *STLR* must be ≥ seg number.
  - STBR + seg number gives the segment base and limit.
  - The segment offset must be ≤ the segment limit.
  - The segment base + segment offset = physical memory address.

5.7 Segmentation with Paging

- The segment table entry consists of the segment limit and the page-table base (PTBR).
- a logical addres is a two tuple <segment number, segment offset>. The segment offset is further broken into <page number, offset>
  - segment length must be ≥ segment offset.
  - STBR + segment number gives segment length and PTBR.
  - PTBR + page number gives frame number.
  - frame number + offset gives physical memory address.

6. Virtual Memory

- Virtual memory is the separation of user logical memory from physical memory. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available.
- advantages:
  - programs can be larger than the size of physical memory.
  - programs can take less physical memory, therefore, more programs can be run at the same time, thus increasing CPU utilization and throughput.
  - less I/O would be needed to swap programs.
- implemented by *demand paging* or *demand segmentation*.

6.1 Demand Paging

- Similar to a paging system with swapping. When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory, however, we use a lazy swapper, i.e. swap a page into memory only when it is needed. Use the term *pager* (moves only one page) instead of *swapper* (moves entire process).
- When trying to access a page that is not in memory, a *page-fault* occurs.
- *Pure demand paging*: never bring a page into memory until it is required.
- Swapping to the swap space is generally faster than that to the file system.

6.2 Page Replacement

- If no frame is free, we find one that is not currently being used and free it.
- We can free a frame by writing its contents to swap space. Can use a *dirty* bit to see if rolling out is necessary.
- Algorithms:
  - FIFO.
    - Usually as the number of frames increase, the page-fault rate will decrease.

    - ***Belady's anomaly*** reflects the fact that, for some page-replacement algorithms, the page-fault rate may increase as the number of allocated frames increases.
  - Optimal. Replace the page that will not be used for the longest period of time.
  - LRU (least recently used).
  - MFU (most frequently used). Replace the page with the largest count because the page with the smallest count was probably just brought in and has yet to be used.
- Implementations:
  - Counters. Use a counter to count every memory reference. Whenever a reference to a page is made, the counter is copied to the time-of-use field in the page table for that page. Replace the page with the smallest counter value.
  - Stack of page numbers. Whenever a page is referenced, it is removed from the stack and put on the top. The bottom is the LRU page.
- LRU Approximation Algorithm.
  - Use a *reference bit* for each page. Initially set at 0. When the page is referenced, it is set to 1.
  - Use a history byte (can be any number of bits). At regular intervals, right shift the reference bit into the high-order bit of this history byte. The LRU page is the one with the lowest history byte number.
  - Second-chance algorithm - where the number of bits for the history is zero, i.e. use only the reference bit. When a page is selected for replacement, check the reference bit. If the value is 0, then replace the page. If the value is 1, then change it to 0 and don't replace the page. Use a circular queue to implement.
  - Enhanced second-chance algorithm. Use both the reference bit and the dirty bit.
- Page Buffering Algorithm.
  - Keep a pool of free frames.
  - Write the page out when the pager is idle.

6.3 Allocation of Frames

- The minimum number of frames required per process is defined by the computer architecture (the instruction). i.e. an instruction can have at most how many memory references.
- The maximum number of frames is defined by the amount of available physical memory.
- Allocation algorithms.
  - Equal allocation.
  - Proportional allocation; either proportional to the size or priority of the process.
- Global versus local allocation. Global replacement allows a process to select a replacement frame from the set of all frames. Local replacement selects from only its own set of allocated frames.

6.4 Thrashing

- A process is thrashing if it is spending more time paging than executing.
- Ex. Increase page fault will decrease CPU utilization. The CPU scheduler sees the decreasing CPU utilization, and increases the degree of multiprogramming as a result. More processes means less frames per process, therefore, more page fault.
- To prevent thrashing, we must provide a process as many frames as it needs.

- Use the *working set* model which is based on the assumption of locality. The locality model states that as a process executes, it moves from locality to locality. A locality is a set of pages that are actively used together.
  - Use a parameter, Δ, to define the *working set window*.
  - The set of pages in the most recent Δ page references is the working set. If a page is in active use, it will be in the working set. If it is no longer being used, it will drop from the working set Δ time units after its last reference.
  - The accuracy of the working set depends on the selection of Δ.
  - The total demand D for frames = sum of all the working set size. If $D >$ the total number of available frames then thrashing will occur.
- *Page-Fault frequency* - another method to solve the thrashing problem. Thrashing has a high page-fault rate. Thus, we want to control the page-fault rate. We establish an upper and lower bounds on the desired page-fault rate.

6.5 Other considerations

- Prepaging is an attempt to prevent the high level of initial paging. The strategy is to bring into memory at one time all the pages that will be needed.
- Page size
- Program structure. e.g. to initialize a 2-D array. Do the rows first or the columns first.
- I/O interlock. Need to lock a page to prevent the page being paged out for I/O.
- Real-time processing.

6.6 Demand Segmentation

- When a process addresses a segment containing either code or data, the hardware checks a valid bit to see whether the segment is in memory. If the segment is in memory, the access continues on. Otherwise, there is a trap to the OS to bring in the entire segment, after which the interrupted instruction (the one causing the segment fault) then continues.

7. File System Interface

7.1 File Concepts

- File attributes: name, type, location, size, protection, time, data and user ID.
- File operations: create, write, read, repositioning within a file, delete, truncate, append, rename.
- The OS keeps a small table containing information about all open files (the *open-file table*). When a file is first opened, a new entry is inserted into the open-file table. When a file operation is requested, an index into this table is used. When the file is no longer actively used, it is closed by the process and the OS removes its entry in the open-file table.

7.2 Access Methods

- Sequential access.
- Direct (relative) access.
  - A file is made up of fixed-length logical records that allow programs to read and write records rapidly in no particular order.

- Indexed (e.g. indexed sequential access method ISAM).
  - Search an index, which gives a pointer to the data block.

7.3 Directory Structure

- The file system is broken into *partitions*.
- Each partition contains information about files within it. This information is kept in entries in a device directory or volume table or directory.
- Single-level directory. All files are contained in the same directory. All files must have unique names.
- Two-level directory. path, search path, partition name.
- Tree-structured directory. current directory. absolute and relative paths.
- Acyclic-graph directory. To share a common subdirectory. With a shared file, there is only one actural file. This is different from having two copies of the same file. Link.

7.4 Protection

- Two extremes: no access by other users, free access by all users.
- Need controlled access: read, write, execute, append, delete, list.
- Access lists for each file, specifying the user name and the types of access allowed by that user. Problem, the list gets too long.
- Groups. owner, group, universe.
- Password associated with each file.

7.5 Consistency Semantics

- Specify when modifications of data by one user are observable by other users.
- Unix semantics. Writes to an open file by a user are visible immediately to other users that have this file open at the same time. There is a mode of sharing where users share the pointer of current location into the file. Here, a file has a single physical image. Contention for this single image results in user processes being delayed.
- Session semantics. Writes to an open file by a user are not visible immediately to other users that have the same file open simultaneously. Onces a file is closed, the changes made to it are visible only in sessions starting later. Already-open instances of the file do not reflect these changes. Here, a file may be associated temporarily with several (possibly different) images at the same time. Thus, user processes are not delayed.

8. File System Implementation

- File directory (Unix ***inode***) to store information regarding a file. Name, owner, size, date, block index pointer. (see page 395, 401).

8.1 Allocation Methods

- How to allocate space to these files so that disk space is utilized effectively and files can be accessed quickly.
- Contiguous allocation.
  - the general dynamic storage-allocation problem: first-fit, best-fit
  - suffer from external fragmentation.
  - can be for either sequentail or direct access.

- Modified contiguous allocation.
  - a contiguous chunk of space is allocated initially, and then, when that amount is not large enough, another chunk of contiguous space, and *extent*, is added.
- Linked allocation.
  - each file is a linked list of disk blocks.
  - no external fragmentation.
  - can only be used by sequential-access files.
  - the directory contains a link to the first block of the file. Each block contains a link to the next block.
- Linked allocation with the use of a file-allocation table (FAT).
  - each partition has a FAT which is a table with one entry for each disk block, and is indexed by block number.
  - the file directory entry contains a link to the first block of the file. The FAT entry indexed by that block number then contains the block number of the next block in the file. Thus, can find block *i* of a file by only searching in the FAT.
- Indexed allocation.
  - like linked allocation but with an added index to support efficient direct access.
  - brings all the pointers together into one location, the *index block*.
  - each file has its own index block, which is an array of disk-block addresses.
  - the directory contains the address of the index block.
  - the *i*th entry in the index block points to the *i*th block of the file.
  - similar to the paging scheme.
  - the pointer overhead is generally greater than for the linked allocation.
  - how large the index block should be? Every file has an index block, so want to be small. However, if it is too small, it will not be able to hold enough pointers for a large file.
    - linked scheme. can link together several index blocks.
    - multilevel index. use a separate index block to point to the index blocks.
    - combined scheme - used in Unix.
      - the directory entry points to the *inode* (index block).
      - the first 12 of the 15 pointers point to the file's data blocks.
      - the next 3 pointers point to a *single, double, and triple indirect block*s respectively. The single indirect block is an index block that contains pointers to the data blocks.

8.2 Free-Space Management

- to keep track of free disk space, the system maintains a *free-space list* which records all disk blocks that are free.
- Bit vector. A 1 bit represents a free block and a 0 for an allocated block.
- Linked list. Link together all the free disk blocks, keeping a pointer to the first free block. The FAT method incorporates free-block accounting into the allocation data structure.
- Grouping. A modification of the free-list approach is to store the addresses of *n* free blocks in the first free block. The last block contains the addresses of another *n* free blocks.
- Counting. In the free-space link list, keep the address of the first free block and the number *n* of free contiguous blocks that follow the first block.

8.3 Directory Implementation

- Linear list
- Hast table

9. Secondary Storage Structure

9.1 Disk Structure

- sector, cylinder, track.
- $b = k + s \times (j + i \times t)$. where b = block number, k = sector, j = surface, i = cylinder, s = number of sectors per track, t = number of tracks per cylinder.

9.2 Disk Scheduling

- *seek time* - head movement time.
- *latency time* - rotational delay.
- *transfer time* - actual transfer of data between the disk and the main memory.

1. FCFS scheduling.
2. SSTF (shortest seek time first) scheduling. Seek time is proportional to the track difference. Moving the head to the closest track in the request queue.
3. SCAN (elevator) scheduling.
4. C-SCAN.
5. LOOK and C-LOOK. Like (C-)SCAN but the head is only moved as far as the last request in each direction.

9.3 Disk Management

- disk formatting.
- boot block
- bad blocks

9.4 Swap-Space Management

- swap space can be carved out of the normal file system or can be a separate disk partition.

10. Distributed OS

10.1 Motivations.

- resource sharing
- enhanced performance
- improved reliability and availability
- modular expandability (scalability)

10.2 Issues

- No global knowledge, i.e. no global memory, and no global clock.
- Naming
- Scalability

- Compatibility - refers to the notion of interoperability among the resources in a system.
- Process synchronization.
- Resource management.
  - Data migration
  - Computation migration

10.3 Topology.

- Issues: 1) basic cost in connecting the network. 2) communication cost - time to send a message. 3) reliability - what happens if a site fails.
  - fully connected network
  - partially connected network
  - hierarchical network
  - star network
  - ring network
  - multiaccess bus network
  - hybrid network

10.4 Network types

- local-area networks
- wide-area networks

10.5 Communication

- Naming and name resolution. How do two processes locate each other to communicate?
  - <host name, process-id>
  - *domain name service* (*DNS*).
- Routing strategies. How are messages sent through the network?
  - each site has a *routing table*.
  - fixed routing. path is fixed.
  - virtual circuit. path is fixed for the duration of one session. (like telephone).
  - dynamic routing. path is chosen only when a message is sent. messages may arrive out of order.
  - *gateway* a host that connects a LAN to other networks.
  - *router* is responsible for routing messages.
- Connection strategies. How do two processes send a sequence of messages?
  - circuit switching. if two processes want to communicate, a permanent physical link is established between them for the duration of the entire communcation. (like telephone).
  - message switching. if two processes want to communicate, a temporary link is established for the duration of one message transfer. (like the post office).
  - packet switching. similar to message switching but the message is broken up into smaller packets, frames, or datagrams. Thus, the packets must be reassembled into messages as they arrive.
- Contention for shared resources. How to resolve conflicting demands?
  - carrier sense with multiple access / collision detection. listen to determine whether another message is currently being transmitted over the link. If the link is free, the site

can start transmitting. If two or more sites begin transmitting at exactly the same time, then they will detect a collision and will stop transmitting.
- token passing. a token is passed around and whoever has the token can send a message.
- message slots. a number of fixed-length message slots continuously circulate in the system. wait for an empty slot and insert message into the slot.

11. Distributed-System Structures

11.1 Network-Operating System

- provides an environment where users, who are aware of the multiplicity of machines, can access remote resources.
- remote login
- remote file transfer

11.2 Distributed-Operating System

- the users access remote resources in the same manner as they do local resources. Data and process migration from one site to another is transparent to the user, i.e. under the control of the distributed OS.
- data migration
  - when a file is needed, automatically transfers the whole file or only portions of the file.
- computation migration
  - transfer the computation, rather than the data, across the system.
- process migration. reasons for process migration:
  - load balancing
  - computation speedup
  - hardware/software preference
  - data access

11.3 Remote Services

- remote procedure calls
- threads. threads may be used to send and receive messages while other operations within the task continue asynchronously.

11.4 Robustness

- To ensure that the system is robust, we must:
  - detect any failures.
  - reconfigure the system so that computation may continue.
  - recover when a site or a link is repaired.

12. Distributed File Systems

- A distributed system is a collection of loosely coupled machines interconnected by a communication network.

- A DFS is a file system whose clients, servers, and storage devices are dispersed among the machines of a distributed system. The distinctive features are the multiplicity and autonomy of clients and servers in the system.
- The multiplicity and dispersion of its servers and storage devices should be transparent to the users.

12.1 Naming and Transparency

- *Naming* is a mapping between logical and physical objects.
- Two related notions that need to be differentiated:
  1. *location transparency*. the name of a file does not reveal any hint of the file's physical storage location.
  2. *location independence*. The name of a file does not need to be changed when the file's physical storage location changes.
- Naming schemes:
  1. A file is identified uniquely by the name *host:local-name*.
  2. *NFS* (Network File System) provides means to attach remote directories to local directories, thus giving the appearance of a coherent directory tree.
  3. A single global name structure spans all the files in the system. Total integration of the component file systems.

12.2 Remote File Access

- using *remote procedure calls* and/or *caching*.
- caching:
  - something like network virtual memory
  - cache location. in memory or on disk?
  - cache update policy. *write-through*, *delayed-write*, *write-on-close*.
  - cache-consistency problem. client-initiated approach, server-initiated approach.

12.3 Stateful versus Stateless Service

- There are two approaches to server-side information. Either the server tracks each file being accessed by each client (stateful), or it simply provides blocks as they are requested by the client without knowledge of the blocks' usage (stateless).
- *Stateful file server*
  - characterized as a connection between the client and the server during a session.
  - a client performs an open on a file before accessing that file.
  - the server fetches some information about the file (the inode) and gives the client a file descriptor (which serves as an index to a table of inodes).
  - this identifier (file descriptor) is used for subsequent accesses until the session ends.
  - increased performance. Loses its state in a crash.
- *Stateless file server*
  - avoids this state information by making each request self-contained.
  - the server does not need to keep a table of open files in memory.
  - there is no need to establish and terminate a connection by open and close operations.
  - the effects of server failures and recovery is like a slow server from a client's point of view.

- the penalty for using the robust stateless service is longer request messages, and slower processing of requests. also file requests must be *idempotent*.

12.4 File Replication

- Replication of files on different machines is a useful redundancy for improving availability.
- Problem with updates.

13. Distributed Coordination

13.1 Event Ordering

- the *happened-before* relation is only a partial ordering of the events in distributed systems because there is no common memory or common clock.
- define the ***happened-before*** relation (denoted by $\rightarrow$) on a set of events as follows:
  1. if A and B are events in the same process, and A was executed before B, then $A \rightarrow B$.
  2. if A is the event of sending a message by one process and B is the event of receiving that message by another process, then $A \rightarrow B$.
  3. if $A \rightarrow B$ and $B \rightarrow C$ then $A \rightarrow C$.
- if two events, A and B, are not related by the $\rightarrow$ relation then we say that these two events were executed ***concurrently***. In this case, neither event can causally affect the other. If, however, $A \rightarrow B$, then it is possible for event A to affect event B causally.
- for ***global ordering***, use a ***timestamp***. For every pair of events, if $A \rightarrow B$, then the timestamp of A is less than the timestamp of B. However, the converse does not need to be true.
- use a *logical clock* LC_i within each process P_i. If $A \rightarrow B$ both in P_i then $LC_i(A) < LC_i(B)$.
- if A in P_1 sends a message to B in P_2 then it is possible that $LC_1(A) = 90$ and $LC_2(B) = 70$ because the logical clock of P_2 is slower.
  - to solve this problem, we require a process to advance its logical clock when it receives a message whose timestamp is greater than the current value of its logical clock. eg $LC_2(B)$ should be advanced to 91.
- finally, to realize a total ordering, if the timestamps of two events are the same, we can use their IDs to break ties and to create a total ordering.

13.2 Mutual Exclusion

- **Centralized Approach**. One of the processes in the system is chosen to coordinate the entry to the critical section. Requires 3 messags per CS entry:
  - Each process that wants to invoke mutual exclusion sends a *request* message to the coordinator.
  - When the process receives a *reply* message from the coordinator, it can proceed to enter its CS.
  - After exiting its CS, the process sends a *release* message to the coordinator and proceeds with its execution.
  - The coordinator is the bottle neck. If the coordinator process fails, then a new process must take its place.

- **Fully Distributed Approach**. Distribute the decision making across the entire system. More complicated. Based on the event-ordering scheme described above.
  1. When process P_i wants to enter its CS, it generates a new timestamp, TS, and sends the message *request(P_i, TS)* to *all* other processes in the system (including itself).
  2. On receiving a *request* message, a process P_j may reply immediately or it may defer sending a reply back based on three factors:
     - if P_j does *not* want to enter its CS, then it sends a reply immediately to P_i.
     - if P_j is in its CS, then it defers and queues its reply to P_i.
     - if P_j wants to enter its CS but has not yet entered it, then it compares its own *request* timestamp with the timestamp TS of the incoming request made by P_i. If its own *request* timestamp is greater than TS, then it sends a reply immediately to P_i (P_i asked first). Otherwise, the reply is deferred.
  3. When P_i has received a *reply* message from all other processes in the system, then it can enter its CS.
  4. After exiting its CS, the process sends *reply* messages to all its deferred requests.
  - this algorithm exhibits: mutual exclusion, no deadlock, no starvation, $2 \times (n\text{-}1)$ messags per CS entry.
  - every processes in the system must know the identity of all the other processes. When a new process joins the group, it must receive the names of all the other processes and its own name must be distributed to all the other processes.
  - if one of the processes fails, then the entire scheme collapses. We can continuously monitor the state of all the processes. When a process fails, then all other processes are notified.
- **Token-Passing Approach**
  - Circulate a token among the processes in the system. The token is passed around in one direction in a logical ring. Possession of the token entitles the holder to enter the CS. When a process exits its CS, it sends the token on to the next process.
  - what if the token is lost? An election must be called to generate a new token.
  - what if a process fails, breaking the logical ring? A new ring must be established.

13.3 Atomicity

- An atomic transaction is a program unit that must be executed *atomically*. i.e. either all the operations are executed to completion, or none are performed.
- In a distributed system, several sites may be participating in the execution of a single transaction. Thus, the failure of one of these sites, or the failure of a communication link connecting these sites, may result in erroneous computations.
- Each site has its own local *transaction coordinator* that is responsible for coordinating the execution of all the transactions initiated at that site:
  - starting the execution of the transaction.
  - breaking the transaction into a number of subtransactions, and distributing these subtransactions to the appropriate sites for execution.
  - coordinating the termination of the transaction, which may result in the transaction being committed at all sites or aborted at all sites.
- **Two-Phase Commit Protocol**
  - a *commit protocol* executed by the transaction coordinator. Let T be a transaction initiated at site S_i, whose transaction coordinator is C_i. When T completes its

execution - that is, when all the sites at which T has executed inform C_i that T has completed - then C_i starts the 2PC protocol:

Phase 1: C_i adds the record <prepare T> to the log and forces the record onto stable storage. Sends a <prepare T> message to all sites at which T executed. On receiving the message, the transaction coordinator at that site determines whether it is willing to commit its portion of T. If the answer is no, it adds a record <no T> to the log and sends the message <no T> to C_i. If the answer is yes, it adds a record <ready T> to the log, and forces all the log records corresponding to T onto stable storage. It then sends the <ready T> message to C_i.

Phase 2: When C_i receives responses to the prepare T message from all the sites (or time out), it can determine whether T can be committed or aborted. T can be committed if C_i received a <ready T> message from all the participating sites. Otherwise, T must be aborted. Depending on the verdict, either a record <commit T> or <abort T> is added to the log and is forced onto stable storage. C_i then sends the verdict message to all participating sites.

- Failure of a participating site S_k. When S_k recovers from a failure, it must examine its log to determine the fate of those transactions T that were in the midst of execution when the failure occurred. Possible cases:
  - the log contains a <commit T> record. In this case, S_k executes **redo**(T).
  - the log contains a <abort T> record. In this case, S_k executes **undo**(T).
  - the log contains a <ready T> record. In this case, S_k must consult C_i to determine the fate of T. If C_i is up, it notifies S_k regarding whether T committed or aborted. On receiving the message, S_k then executes **redo**(T) or **undo**(T) respectively. If C_i is down, S_k must find the fate of T from the other sites. If no site has the answer then S_k must wait with the pending decision of whether to commit or abort T.
  - the log contains no control records (abort, commit, ready) concerning T. The absence of control records implies that S_k failed before responding to the <prepare T> message from C_i. Since the failure of S_k precludes the sending of such a response, by our algorithm C_i must abort T. Hence, S_k must execute **undo**(T).
- Failure of the coordinator site C_i.
  - if S_k contains a <commit T> record in its log, then T must be committed.
  - if S_k contains an <abort T> record in its log, then T must be aborted.
  - if S_k does not contain a <ready T> record in its log, then the failed C_i cannot have commited T because S_k has not sent a <ready T> message to C_i. Abort T.
  - if none of the preceding cases holds, then all participating sites must have a <ready T> record in their logs, but no additional control records. Since C_i has failed, it is impossible to determine whether a decision has been made, or what that decision is, until C_i recovers. While waiting, other resources may be blocked by T.

13.4 Concurrency Control

- Locking protocols:

1. **Nonreplicated scheme**. Each site maintains a local lock manager. When a transaction wishes to lock data item Q at site S_i, it simply sends a message to the lock manager at site S_i requesting a lock. If Q is currently locked in an incompatible mode, then the request is delayed until that request can be granted. If the lock can be granted, the lock manager sends a message back to the initiator indicating that the lock request has been granted.

2. **Single-coordinator approach**. The system maintains a single lock manager that resides in a single chosen site, say S_i. All lock and unlock requests are made at site S_i. When the request is granted, the transaction can read the data item from any one of the sites where a replica of the data item resides. In the case of a write, all the sites where a replica of the data item resides must be involved in the writing.
3. **Multiple-coordinator approach**. The lock-manager function is distributed over several sites. Each lock manager administers the requests for a subset of the data items.
4. **Majority protocol**. A modification of the nonreplicated scheme. A lock manager at each site that manages the locks for all the data or replicas of data stored at that site. When a transaction wishes to lock a data item Q, which is replicated in *n* different sites, it must send a lock request to more than one-half of the *n* sites in which Q is stored. The transaction operates on Q after successfully obtaining a lock on a majority of the replicas of Q.
5. **Biased protocol**. Like majority protocol except that requests for shared locks are given more favorable treatment than are requests for exclusive locks.
6. **Primary copy**.

13.5 Deadlock Handling

- Deadlock prevention
- Deadlock detection
  - centralized approach. a global wait-for-graph is constructed as the union of all the local wail-for graphs, and is maintained in a single deadlock-detection coordinator.
    - may have false cycles.
    - unnecessary rollbacks.
  - fully distributed approach. every site constructs its own local wait-for graph with an additional node $P_{external}$ to the graph. An arc $P_i \rightarrow P_{external}$ exists in the graph if P_i is waiting for a data item in another site being held by any process.

13.6 Election Algorithms

- Algorithm to elect which site will have the coordinator process.
- Bully algorithm. Each process has a unique priority number. The process with the highest priority number is the coordinator process.
- Ring algorithm.

13.7 Reaching Agreement

- A mechanism that allows a set of processes to agree on a common "value."
- *Byzantine generals problem.*
- either unreliable links or faulty processes.