Introduction (Chapter 1)

An **operating system (OS)** is a program that acts as intermediary between the user of a computer and the computer hardware. The purpose of an operating system is to provide an environment in which a user can execute programs in a *convenient* and *efficient* manner.

Mainframe OSs are designed primarily to optimize utilization of hardware,

PCs OSs support games and business applications, and everything in between,

Handheld computer OSs provide an easy (convenient) interface with the computer to execute programs.

So some OSs are designed to be convenient, other to be efficient, and others to be a combination of the two.

A computer has four components: hardware, OS, application programs, and the users.

The **hardware** (consisting of the CPU, memory, and I/O devices) provides the basic computing resources,

the **application programs** (e.g., word processors, spreadsheets, web browsers, compilers, etc.) define the ways in which these resources are employed by the users.

The OS controls and coordinates the use of the hardware among the various application programs for the various users.

The components of a computer system are its hardware, software, and data. The OS provides the means for the proper use of these resources in the operation of the computer system.

OS can be explored from two viewpoints: the **user** and the **system**.

User Viewpoint

Most users sit in front of a PC (consisting of monitor, keyboard, mouse, and system unit). Such a system is designed for one user to monopolize its resources, to maximize the work or play that the single user is performing. In this case, the OS paradigm is **ease of use**, with some attention paid to performance, and none to resource utilization.

Some users sit at a **mainframe or minicomputer**, with other users accessing the same computer through other terminals, sharing resources and information. The OS is designed to maximize **resource utilization**, to assure that CPU time, memory, and I/O devices are used efficiently and fairly.

Other users sit at **workstations**, connected to the network and other workstations and servers. These users have dedicated resources at their disposal, but they also share printers, compute time of servers, network, etc. Therefore, the OS is designed to compromise between individual usability and resource utilization.

System Viewpoint

The OS is intimate with the hardware. We can view the OS as a **resource allocator**; the OS acts as a manager of the CPU, memory, I/O devices, etc.

The OS must satisfy (sometimes conflicting) requests for resources, made by users and programs.

The OS can also be viewed as a **control program** that manages the execution of user programs to prevent errors and improper use of the computer. It is especially concerned with the operation and control of I/O devices.

We have no completely adequate definition of OSs; they exist because they are needed to create a usable computing system. However, a common definition is that the OS is the one program running at all times in the computer (usually called the **kernel**), with all else being application programs.

The matter of *what is an OS* is important: in 1998 the US Dept of Justice, filed suit against Microsoft, in essence claiming that Microsoft included too much functionality in its OS and prevented competition from other vendors.

Convenience and **efficiency** are (sometimes contradictory) goals for an OS.

OSs have evolved over time. For example, UNIX started with a keyboard and a printer as its interface, and over time **graphic user interfaces (GUI)** were added, allowing UNIX to be more convenient while still concentrating on efficiency.

OSs and computer architecture have influenced each other a great deal: to facilitate the use of hardware, researchers developed OSs. Users of OSs have then proposed changes to hardware to simplify them.

Mainframe systems

Mainframe computer systems were the first computers used to tackle many commercial and scientific applications.

These early computers were physically enormous machines run from a **console***; the common input devices were card readers and tape drives.

The common output devices were line printers, tape drives, and card punchers. The user did not interact directly with the computer system.

*The Oxford Dictionary defines a console as: 1 panel for switches, controls, etc. 2 cabinet for a television etc. 3 cabinet with a keyboard and stops of an organ.

The user did not interact directly with the computer; the user prepared a job (consisting of a program, the data and some control information about the nature of the job), and submitted it to the computer operator. The job was usually in the form of punch cards. Sometime later (minutes, hours, or days), the output appeared. The output consisted of the results of the program, as well as a dump of the final memory and register contents for debugging.

The OS in these early computers was simple: its major task was to transfer control automatically from one job to the next. The OS was always resident in memory.

To speed operation, operators **batched** together jobs with similar needs, and run them through the computer as a group.

In the batch execution environment, the CPU is often idle, because the speeds of the mechanical I/O devices are intrinsically slower than are those of electronic devices.

Even a slow CPU works in the microsecond range, while a fast card reader might read 1200 cards per minute. Thus, the difference in speed between the CPU and its I/O devices may be three orders of magnitude or more.

Over time fast disks were introduces, but CPUs have also become faster, so the problem, instead of being solved, has exacerbated.

The introduction of disk technology allowed the OS to keep all jobs on a disk, rather than on a serial card reader. With direct access to jobs, the OS can perform **job scheduling**, to use resources and perform tasks efficiently.

The most important aspect of job scheduling is **multiprogramming**. A single user cannot keep the CPU and I/O devices busy at all times; multiprogramming increases CPU utilization by organizing jobs so that the CPU has always one to execute.

IDEA: the OS keeps a set of jobs in the memory–a subset of the jobs in the **job pool** (the entire job pool may not fit in the memory). The OS executes a job until it has to wait for some I/O operation to complete. Instead of keeping the CPU idle, the OS switches to another job. Eventually, the first job finishes waiting and gets the CPU back.

Multiprogramming OSs are fairly sophisticated; making the decisions of which job gets to be executed when is called job scheduling.

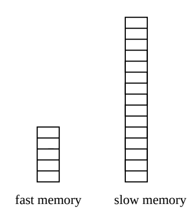
Time sharing (or **multitasking**) is a logical extension of multiprogramming. The idea is that the CPU executes multiple jobs by switching among them, but the switches occur so frequently that the user(s) can interact with each program while it is running.

An **interactive** computer system provides direct communication between the user and the system. The user gives instructions to the OS (with keyboard or mouse) and waits for immediate results. The **response time** should be short (1 second or so).

A time sharing OS allows many user to use the same computer; since the switching between jobs is fast, each user has the impression that the computer is completely dedicated to them.

Paging

Consider the following two-level **virtual memory system**:



Each level (slow and fast) can store a number of fixed-size memory units called **pages**. The slow memory stores N pages, and the fast memory stores $k < N$ pages.

Given a request for page p_i, the system must make the page p_i available in the fast memory.

If p_i is already in the fast memory (called a **hit**), the system need not do anything. Otherwise, on a **miss**, the system incurs one **page fault**, and must copy the page p_i from the slow memory to the fast memory.

In doing so, the system is faced with the problem of which page to evict from the fast memory to make space for p_i.

In order to *minimize* the number of page faults, the choice of which page to evict must be made wisely.

Examples of pairs of fast and slow memory are:

RAM and Disk

Cache and RAM

Because of its important role in the performance of almost every computer system, the paging problem has been extensively studied since the 1960s.

Often we will refer to the fast memory as the cache.

A large variety of page replacement algorithms can be found in existing computer systems:

LRU	(Least-recently-used)
CLOCK	(Clock-replacement)
FIFO	(First-in/First-out)
LIFO	(Last-in/First-out)
LFU	(Least-frequently-used)
LFD	(Longest-forward-distance)

All these algorithms, except for the last one, are **on-line algorithms**; that is, they are algorithms that make decisions based on past events, rather than the future.

The last algorithm, LFD, replaces the page whose next request is the latest. Thus, LFD requires full knowledge of future requests, and hence it is called an **off-line algorithm**.

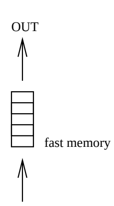
The traditional approach to studying online algorithms falls within the framework of **distributional** (or **average-case**) complexity, whereby one hypothesizes a distribution on event sequences and studies the expected payoff per event.

We are interested in the (more recent) approach of **competitive analysis**, whereby the payoff of an on-line algorithm is measured by comparing its performance to that of an *optimal offline algorithm*. Competitive analysis thus falls within the framework of **worst-case** complexity.

FIFO Page Replacement
(Section 10.4.2)

When a page must be replaced, the *oldest* page is chosen.

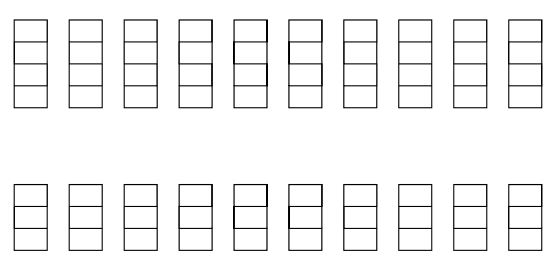
It is not necessary to record the time when a page was brought in; all we need to do is create a FIFO queue to hold all pages in memory:



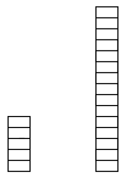
The FIFO algorithm is easy to understand and program, but its performance is not always very good. It suffers from the so called **Belady's Anomaly**.

Suppose that we have the following sequence of page requests: $1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5$. Then, we have more page faults when $k = 4$ than when $k = 3$. That is, we have more page faults with a bigger fast-memory when using FIFO!

$1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5$



Demand Paging



We consider the **page fault model**, for measuring the performance of paging algorithms. In this model, we charge 1 for bringing a page into the fast memory, and we charge 0 for accessing a page which is already in the fast memory.

This is a very simplistic model, which does not address a number of factors, notably it does not address the fact that access to a fast memory page is in practice not free of charge.

Also, our algorithms are **demand paging**, meaning that unless there is a page fault, they never evict a page from the fast memory.

Claim: Any page replacement algorithm (online or offline) can be modified to be demand paging without increasing the overall cost on any request sequence.

We are going to prove this claim. Keep in mind that in a demand paging algorithm:

1. Each page fault causes exactly one eviction,

2. there are no evictions between misses.

So let A be any paging algorithm. We show how to modify it to make it a demand paging algorithm A', in such a way that on any input sequence A' incurs at most the cost (makes at most as many page moves from slow to fast memory) as A.

Suppose that A has a cache of size k. Define A' as follows: A' also has a cache of size k, and also k registers. A' runs a simulation of A, keeping in its k registers the page numbers of the pages that A would have in its cache. Based on the behavior of A, A' makes decisions to evict pages.

Suppose page p is requested. If p is in the cache of A', do nothing. Otherwise, if a page fault occurs, A' behaves as follows:

- If A also has a "page fault" (that is, the number of p is *not* in the registers), and A "evicts" a page from register i to make room for p, then A' evicts a page from slot i in its cache, to make room for p.

- If A does not have a "page fault", then the number of p must be in, say, register i. In that case, A' evicts the contents of slot i in its cache, and moves p in there.

Thus A' is a demand paging algorithm.

We now show that A' incurs at most the cost of A on any input sequence; that is, A' has at most as many page faults as A.

To do this, we pair each page move of A' with a page move of A in a unique manner as follows: If A' and A both incur a page fault, then match the corresponding page moves.

Otherwise, if A already had the page in its cache, it must have moved it there before, so match that move with the current move of A'.

It is never the case that two different moves of A' are matched with a single move of A.

To see this, suppose that on some input sequence, we encounter for the first time the situation where two moves of A' are matched with the same move of A.

This can only happen in the following situation: page p is requested, A' incurs a page fault, it moves p into its cache, and we match this move with a past move of A, which has been matched already!

But this means that page p was already requested, and after it has been requested, it has been evicted from the cache of A' (otherwise, A' wouldn't have had a page fault).

Thus, A' evicted page p while A did not, so they were not in the same slot. But A' put (the first time) p in the same slot as A! Contradiction.

$\therefore$ We couldn't have matched a move twice.

Thus, we can match each move of A' with a move of A, in a one-to-one manner, and hence A' makes at most as many moves as A.

The idea is that A does not gain anything by moving a page into its cache pre-emptively (before the page is actually needed). A' waits for the request before taking the same action.

In the meantime (between the time that A moves in the page and the time that it is requested and A' brings it in), A' can only gain, because there are no requests for that page during that time, but there might be a request for the page that A evicted pre-emptively.

Note that in the simulation, A' only needs k extra registers, to keep track of the page numbers of the pages in the cache of A, so it is an *efficient* simulation.

Offline and Online Optimization Problems

An **optimization problem** can be a problem of cost minimization of profit maximization.

An optimization problem $\mathcal{P}$ of cost minimization consists of a set $\mathcal{I}$ of inputs and a cost function C. Associated with every input I is a set of feasible outputs (or solutions) $F(I)$, and associated with each feasible solution O in $F(I)$ is a positive real $C(I, O)$, representing the cost of the output O with respect to the input I.

There are analogous concepts for profit maximization.

For example, consider the bin packing problem. This problem presents an unbounded number of uniform bins, each having some fixed size and capacity, say 1. An input is a sequence of items:

$$x_1, x_2, \ldots, x_n$$

where x_i represents the size of the i-th item. All item sizes satisfy $0 < x_i \leq 1$. The goal is to "pack" all items into bins in the most compact way.

Given an input, a feasible solution is any assignment of all the input items x_i such that the sum of the item sizes assigned to any bin does not exceed the bin capacity (which is 1).

The cost of a (feasible) solution is the number of bins used to pack all the items. Originally, this bin packing problem was studied as an "offline problem." That is, an algorithm is allowed to consider the entire list of items in order to compute the best solution.

In order to view bin packing as an "online problem,":

- each item x_i must be assigned without knowledge of items $x_{i+1}, \ldots, x_n$,
- the assignment of items $x_1, \ldots, x_i$ must be a feasible solution for each i,
- when we pack x_i, we are not allowed to alter the bin assignment of previous packed items.

Thus, an online feasible solution is a sequence of assignments. Exactly as in the offline problem, the total cost of an online solution is the total number of bins used.

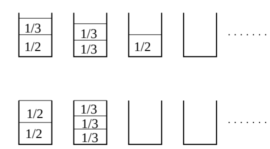
Equivalently, we can consider the cost of each online assignment of one item to be either 0 or 1, depending on whether or not the item assignment opened a new bin (during the online assignment) and the total cost becomes the sum of the individual item assignment costs.

Suppose that we have the following sequence of weights:

$$1/2, 1/3, 1/3, 1/3, 1/2$$

The figure below shows the result of packing these weights into bins according to two strategies: the first (naive), packs the bins with the weights as they come, and uses three bins.

The second strategy, waits until it sees all the weights, and then packs them into the bins, in a clever way, to minimize space, and it uses only two bins.



The point of this example is that an online bin-packing algorithm is at a disadvantage, since it cannot wait and see all the weights, and then pack them, and naive bin packing does not yield an optimal solution (in terms of bins used).

Given any legal input I, an algorithm ALG for an optimization problem $\mathcal{P}$ computes a feasible (solution) $\text{ALG}[I] \in F(I)$. The cost associated with this feasible output is denoted by $\text{ALG}(I) = C(I, \text{ALG}[I])$. An **optimal algorithm** OPT is such that for all legal inputs,

$$\text{OPT}(I) = \min_{O \in F(I)} C(I, O)$$

An algorithm ALG is a c**-approximation algorithm** for a minimization problem $\mathcal{P}$ if there is a constant $\alpha \geq 0$ such that for all legal inputs:

$$\text{ALG}(I) - c \cdot \text{OPT}(I) \leq \alpha$$

Optimization problems in which the input is received in an online manner and in which the output must be produced online are called **online problems**.

In some problems, such as jobs scheduling and bin packing, both the online and offline versions of the problem are naturally meaningful.

However, many problems such as:

- paging
- telephone circuit switching
- investment planning
- CPU scheduling

are online problems. In particular, the two big problems in Operating Systems: paging and scheduling, are intrinsically online, and offline algorithms for them are not acceptable.

Still, even for intrinsically online problems, we could hypothesize an offline algorithm (that must have clairvoyant abilities) for the sake of comparative analysis. For example, we use the OPT paging algorithm as a paradigm for page-replacement algorithms.

The Competitive Ratio and Competitiveness

An online algorithm ALG is c**-competitive** if there is a constant α such that for all finite input sequences I,

$$\text{ALG}(I) - c \cdot \text{OPT}(I) \leq \alpha$$

When the **additive constant** α is less than or equal to zero (i.e., $\text{ALG}(I) \leq c \cdot \text{OPT}(I)$) we may say for emphasis that ALG is **strictly** c**-competitive**.

Allowing a positive constant α reflects the view that for intrinsic online problems such as paging we have an arbitrary long input sequence with unbounded cost. The constant α becomes insignificant as we consider longer and longer (and more costly) initial subsequences.

A (strictly) c-competitive online algorithm ALG is a c-approximation algorithm with the restriction that ALG must compute online. Thus, for each input I, a c-competitive algorithm is guaranteed to incur a cost within a factor c of the optimal offline cost.

Note that the competitive ratio is always at least 1, and the smaller it is, the better ALG performs with respect to OPT. If ALG is c-competitive, we say that ALG attains a **competitive ratio** c. An algorithm is **competitive** if a "c" can be found for it.

Although c may be a function of the problem parameters, it must be independent of the input I. For example, in a scheduling problem with N machines, we might have a competitive ratio c that depends on N, but c must be independent of the number and type of jobs being scheduled.

The infimum over the set of all values c such that ALG is c-competitive is called the **competitive ratio** of ALG, and is denoted by $\mathcal{R}(\text{ALG})$.

ASIDE: One way to analyze online algorithms is to view the problem as a game between an **online player** and a malicious **adversary**.

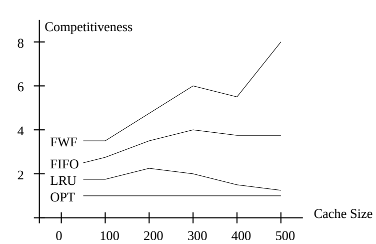
The online player runs an online algorithm on an input that is created by the adversary.

The adversary, based on the knowledge of the algorithm used by the online player, constructs the worst possible input so as to maximize the competitive ratio. That is, the adversary tries to make the task costly to the online player but, at the same time, inexpensive for the optimal offline algorithm.

We sometimes identify the adversary and the optimal offline algorithm as one entity: the **offline player**.

Experimental Evidence

The following figure summarizes the results of experimental evidence. It graphs the "competitiveness" of various paging algorithms as a function of the cache size with respect to a particular input sequence consisting of 692,057 requests to 642 distinct pages of 1024 bytes each. The sequence was generated by two "X windows" network processes, a "makefile" (program compilation), and a "disk copy" running concurrently. The requests include data reads and writes and instruction fetches.



The question now is: will our competitive analysis reflect these empirical findings?

Given the importance of memory hierarchies, paging (caching) continues to be a very active research area, and we need methods for measuring performance of page-replacement algorithms.

LRU (Section 10.4.4)

The optimal (infeasible) algorithm for page replacement, OPT, evicts the page whose next request is the latest (and if some pages are never requested again, then one of them must be evicted, and the choice of which one is unimportant).

If we use the recent past as an approximation of the near future, then we will replace the page that:

has not been used for the longest period of time

This approach is the **Least Recently Used (LRU)** algorithm.

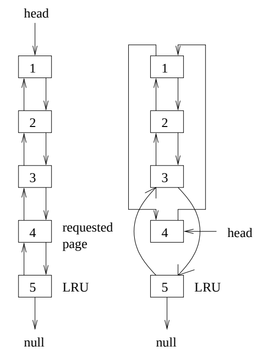
LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses that page that has not been used for the longest period of time.

The LRU algorithm is considered to be good, and is often implemented. The major problem is *how* to implement it.

Two implementations are feasible:

- **Counters:** Keep track of the time when a given page was last referenced, updating the counter every time we request it. This scheme requires a search of the page table to find the LRU page, and a write to memory for each request. (Problems: clock overflow, and others).
- **Stack:** Keep a stack of page numbers. Whenever a page is referenced, it is removed from the stack and put on the top. In this way, the top of the stack is always the most recently used page, and the bottom is the LRU page. Because entries are removed from the middle of the stack, it is best implemented by a:

  <u>Doubly-linked List</u>



How many pointer operations have to be performed in the above example? In class we said that 6, where we counted them as follows:
-remove old head and add new head: 2 operations
-connect 4 with 1: 2 operations
-connect 3 with 5: 2 operations

However, we could have also counted disconnecting 3 with 4 and 4 with 5, giving 4 more pointer operations, giving us a total of 10. A third strategy would be not to count disconnecting pointers, in which case we would get half of these operations, 5. It does not really matter how we count, because the point is that in order to move a requested page (after a hit) to the top, we require a small constant number of pointer operations, regardless of how we count them.

Exercises

(Not to hand in, but a good preparation for the test.)

1. List the pointer operations that have to be performed if the requested page is *not* in the cache. Note that you should list the pointer operations, not count them, since we just showed that there are three different (all reasonable) ways to count them. Again, the point is, that if a page has to be brought in from the slow memory to the cache, a small constant number of pointer operations have to be performed.

2. We are using a doubly-linked list to keep track of the LRU page. Explain what would go wrong if we used a normal linked list, that is, every page has a pointer to the next page ($i \longrightarrow j$, meaning that i was requested more recently than j, but no page was requested later than i and sooner than j). Is there is simple fix to the linked list that would make it work?

Assignment 1, Question 2

Recall the LRU page-replacement algorithm, presented in Section 10.4.4 in the textbook. Show that LRU does *not* incur Belady's Anomaly (on any cache size and any request sequence).

Let σ be any request sequence, $\sigma = p_1, p_2, \ldots, p_n$, and let

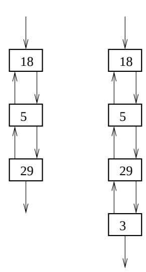
$$\mathsf{LRU}_i(\sigma)$$

be the number of faults that LRU incurs on σ with a cache of size i. We are going to show that for all i and σ, the following property holds:

$$\mathsf{LRU}_i(\sigma) \geq \mathsf{LRU}_{i+1}(\sigma) \qquad (1)$$

Once you have (1), it is easy to show that for any $i < j$ and σ, $\mathsf{LRU}_i(\sigma) \geq \mathsf{LRU}_j(\sigma)$ (you do this!). Conclude that LRU does not incur Belady's Anomaly.

The work is showing (1). To show it, we define a property of doubly-linked lists which we call embedding. We say that a doubly-linked list of size i can be **embedded** in another doubly-linked list of size $i+1$, if the two doubly-linked lists are identical, except that the longer one may have one more item at the "end." For example:



Here, the doubly-linked list of size 3 on the left can be embedded in the doubly-linked list of size 4 on the right.

At the beginning of processing the request sequence, when the caches are getting filled up, the two lists are identical, but once the caches are full, the LRU_{i+1} cache will have one more item in general.

Now prove the following Claim:

Claim: After each step of processing a sequence of requests, the doubly-linked list of LRU_i can be embedded in the doubly-linked list of LRU_{i+1}.

We prove this claim by induction on the number of steps. Basis Case: if $n = 1$, then both LRU_i and LRU_{i+1} incur a fault and bring in p_1.

Induction Step: suppose that the claim holds after step n; we show that it also holds after step $n+1$. Consider the following cases:

1. LRU_i has a hit on p_{n+1}.
2. LRU_i has a fault on p_{n+1}.
   (a) LRU_{i+1} also has a fault.
   (b) LRU_{i+1} does not have a fault.

Show that in each case the embedding property is being preserved (using the Induction Hypothesis, and some basic reasoning about the LRU algorithm).

Finally, use the claim to prove (1).

Marking Algorithms

Consider a cache of size k. Fix any request sequence σ. We divide the request sequence into **phases** as follows: phase 0 is the empty sequence.

For every $i \geq 1$, phase i is the maximal sequence following phase $i-1$ that contains at most k distinct page requests; that is, if it exists, phase $i+1$ begins on the request that constitutes the $(k+1)$-st distinct page request since the start of the i-th phase.

Such a partition is called a **k-phase partition**. This partition is well defined and is independent of how any particular algorithm processes σ.

Example of a 3-phase partition:

$$\underbrace{1,2,1,2,1,2,3}_{1},\underbrace{4,5,6,6,6,6,6,6,6,4,5,4}_{2},\underbrace{7,7,7,7,1,2}_{3}$$

Let σ be any request sequence and consider its k-phase partition. Associate with each page of the slow memory a bit called the **mark**.

The marking is done for the sake of analysis (this is not implemented by the OS, but by "us" to keep track of the doings of an algorithm).

Each page, when its mark bit is set we say that the page is **marked**, and otherwise, **unmarked**.

Suppose that at the beginning of each k-phase we unmark all the pages, and we mark a page when it is first requested during the k-phase. A **marking algorithm** never evicts a marked page from its fast memory. We can consider FWF as the most elementary marking algorithm.

We are interested in the following two results:

- LRU is a marking algorithm
- Let ALG be any marking algorithm with a cache of size k, and let OPT be the optimal offline algorithm with a cache of size $h \leq k$. Then:

$$\text{ALG is } \left(\frac{k}{k-h+1}\right)\text{-competitive}$$

Example

Suppose that $k = 2$, and σ is a request sequence. We show the 2-phases of σ:

$$\sigma = \underbrace{1,1,3,1}_{\text{2-phase \#1}}, \underbrace{5,1,5,1,5,1}_{\text{2-phase \#2}}, \underbrace{3,4,4,4}_{\text{2-phase \#3}}, \underbrace{2,2,2,2}_{\text{2-phase \#4}}$$

In this table, we show who is marked after step i. Note that after each phase, everyone is unmarked, and we begin marking fresh:

step	1	2	3	4	5
1	x				
2	x				
3	x		x		
4	x		x		
5					x
6	x				x
7	x				x
8	x				x
9	x				x
10	x				x
11			x		
12			x	x	
13			x	x	
14			x	x	
15		x			
16		x			
17		x			
18		x			

Observation 1: Except for the last phase, all phases are always complete (they have exactly k distinct requests, 2 in this case).

Observation 2: With a marking algorithm, once a request for page p in phase i is made, p stays in the cache until the end of phase i (the first time p is requested, it is marked, and it stays marked for the entire phase, and a marking algorithm never evicts a marked page).

Observation 3: Intuitively, marking algorithms seem like good page-replacing algorithms because, in a any given phase, there are at most k distinct pages, so they all fit in a cache of size k, so it does not make sense to evict them, in that phase (we can only loose by evicting, as the evicted page might be requested again).

LRU is a marking algorithm

PROOF: We argue by contradiction: suppose that LRU is not a marking algorithm.

Let σ be a request sequence where there exists a k-phase partition, during which some marked page p_i is evicted. Consider the first request for p_i during this k-phase:

$$\sigma = p_1, \ldots, \ldots, \underbrace{\ldots, p_i, \ldots, \ldots, \ldots,}_{k\text{-phase}} \ldots, \ldots$$

Immediately after p_i is serviced, it is marked as the most recently used page in the cache (i.e., it is at the top of the doubly-linked list).

In order for p_i to leave the cache, LRU must incur a page fault while p_i is the least recently used page. It follows that during the k-phase in question, $k+1$ distinct pages were requested: there are the $k-1$ pages that pushed p_i to the end of the list, there is p_i, and the page that got p_i evicted.

Contradiction; a k-phase has at most k distinct pages.

Exercise: Show that FWF is a marking algorithm (the demand-paging version that marks all the pages when the cache is full, and evicts, when necessary, an arbitrary marked page). Show that FIFO is not a marking algorithm.

Note that FWF is an **explicit** marking algorithm (pages are actually marked by the OS). LRU is an **implicit** marking algorithm (the pages are not given marks, but LRU *is* a marking algorithm).

Note: Lets assume that we always consider the following version of the FWF algorithm: each slot in the cache has a single bit associated with it. At the beginning, all these bits are set to zero. When a page p is requested, FWF checks only the slots with a marked bit. If p is found, it is serviced. If p is not found, then it has to be brought in from the slow memory (even if it actually is in the cache, in an unmarked slot). FWF looks for a slot with a zero bit, and one of the following happens:

- A slot with a zero bit (an unmarked page) is found. Then, FWF replaces that page with p.

- A slot with a zero bit is not found (all pages are marked). Then, FWF unmarks all the slots, and replaces any page with p, and it marks p's bit.

Question 4, Assignment 1

For this exercise assume that the cache is of size k. Otherwise the claim is not true: for example, suppose that we have a cache of size 1, and the following sequences:

$$1, 2, 1, 2$$

Then, in that sequence of 4 requests there are only 2 distinct requests, yet with a cache of size 1, there would be 4 faults, for *any* page-replacement algorithm.

Question 3, Assignment 1

We will show that if ALG is any paging algorithm (on-line or off-line), then on any sequence of requests σ, ALG$(\sigma) \geq$ OPT(σ). As usual, ALG(σ) denotes the number of page faults of ALG on the sequence of requests σ. We assume throughout that all algorithms are working with a cache of a fixed size k.

We need to prove the following claim:

Claim: Let ALG be any paging algorithm. Let $\sigma = p_1, p_2, \ldots, p_n$ be any request sequence. Then, it is possible to construct an off-line algorithm ALG$_i$ that satisfies the following three properties:

1. ALG$_i$ processes the first $i-1$ requests of σ exactly as ALG does,

2. if the i-th request results in a page fault, ALG$_i$ evicts from the cache the page with the "longest forward distance",

3. ALG$_i(\sigma) \leq$ ALG(σ)

First show how to use the **Claim** to prove that OPT is in fact the optimal algorithm. Let $\sigma = p_1, p_2, \ldots, p_n$ be once again any sequence of requests. By the claim, we know that:

$$\text{ALG}_1(\sigma) \leq \text{ALG}(\sigma)$$

Applying the claim again, we get:

$$\text{ALG}_2(\sigma) \leq \text{ALG}_1(\sigma)$$

YOU FINISH THIS!

Proof of Claim: Divide σ into three segments as follows:

$$\sigma = \sigma_1, p_i, \sigma_2$$

Since on σ_1, ALG$_i$ acts just like ALG, it follows that ALG$_i(\sigma_1)$ = ALG(σ_1), and also, they both do or don't incur a page fault on p_i.

1. They don't incur a fault.
2. They do incur a page fault, ALG$_i$ evicts the LFD page p from its cache, and replaces it with p_i.
   (a) ALG also evicts the same page (the LFD page). What should ALG$_i$ do on σ_2?
   (b) ALG evicts a different page, q.
      i. $p, q \notin \sigma_2$
      ii. $q \in \sigma_2$ but $p \notin \sigma_2$
      iii. $p, q \in \sigma_2$
      What should ALG$_i$ do on σ_2?

Any marking algorithm ALG is $\frac{k}{k-h+1}$-competitive
(where k is the size of its cache, and h is the size of the cache of OPT)

PROOF: Fix any request sequence σ and consider its k-phase partition. Assume, for now, that the last phase of σ is complete (in general, the last phase may be incomplete).

Claim 1: For any phase $i \geq 1$, a marking algorithm ALG incurs at most k page faults.

Proof of Claim 1: This follows because there are k distinct page references in each phase. Once a page is requested, it is marked and therefore cannot be evicted until the phase has been completed. Consequently, ALG cannot fault twice on the same page.

If we denote the i-th k-phase of σ by σ_i, we can express the above claim as ALG$(\sigma_i) \leq k$.

Thus, if there are s phases, ALG$(\sigma) \leq s \cdot k$.

Claim 2: OPT$(\sigma) \geq s \cdot (k - h + 1)$, where again we assume that $\sigma = \sigma_1, \sigma_2, \ldots, \sigma_s$, where σ_s is complete.

Proof of Claim 2: Let p_a be the first request of phase i, and p_b the last request of phase i. Suppose first that phase $i+1$ exists (that is, i is *not* the last phase). Then:

$$\sigma = \ldots, p_{a-1}, \underbrace{p_a, p_{a+1}, \ldots, p_b}_{i}, \underbrace{p_{b+1}, \ldots, \ldots}_{i+1}$$

After processing request p_a, OPT has at most $h-1$ pages in its cache, not including p_a. From (and including) p_{a+1} until (and including) p_{b+1}, there are at least k distinct requests. So OPT must incur at least $k - (h-1) = k - h + 1$ faults on this segment.

If i is the last phase (so $i = s$), we don't have p_{b+1}, so we can only say that we have at least $k-1$ faults, but we make it up with p_1 which has not been counted.

It follows from the above two claims, that:

$$\text{ALG}(\sigma) \leq s \cdot k \text{ and } \text{OPT}(\sigma) \geq s \cdot (k - h + 1)$$

so that:

$$\frac{\text{ALG}(\sigma)}{s \cdot k} \leq 1 \leq \frac{\text{OPT}(\sigma)}{s \cdot (k-h+1)}$$

so finally:

$$\text{ALG}(\sigma) \leq \left(\frac{k}{k-h+1}\right) \cdot \text{OPT}(\sigma)$$

In the case that σ can be divided into s complete phases.

As was mentioned above, in general, the last phase may not be complete. Then, we repeat this analysis with $\sigma = \sigma_1, \sigma_2, \ldots, \sigma_{s-1}$, and for σ_s we use α at the end, so we get:

$$\text{ALG}(\sigma) \leq \left(\frac{k}{k-h+1}\right) \cdot \text{OPT}(\sigma) + \alpha$$

Exercise: work this out.

Therefore, in either case we obtain that any marking algorithm ALG is $\left(\frac{k}{k-h+1}\right)$-competitive.

Nachos

Over the years, numerous projects have been developed for teaching OSs. Many of these projects were motivated by the development of UNIX [Ritchie and Thompson 1974]. At Berkley, they implemented a new teaching OS and simulation environment, called

Nachos

(available from: `http://www.cs.washington.edu/homes/tom/nachos`)

Nachos makes it possible to write portions of each of the major pieces of a modern OS:

–thread management
–files systems
–multiprogramming
–virtual memory
–networking

Any one can download and install Nachos from the above address. They have ports to:

–DOS/Borland
–Linux
–Solaris (Sun's UNIX based environment)
–RS/6000 (IBM large scale, high performance computer server)
–HP-UX (The Hewlett-Packard UNIX)

Nachos is implemented in a subset of C++. Object-oriented programming is becoming more popular, and C++ is a natural idiom for stressing the importance of modularity and clean interfaces in building OSs.

As a results of the designers emphasis on simplicity, the Nachos OS is about 2500 lines of code, about half of which are devoted to interface description and comments. (The hardware simulator takes up another 2500 lines, but you do not need to understand the workings of the simulator, to design parts of the OS.)

Conclusion:

If you download Nachos, you can learn about the principal OS ideas by adding functionality to the bare-bones Nachos system and improving the performance on benchmarks that are provided. This provides you with a working example of a UNIX-like OS, albeit overly simplistic.

Linux Kernel

Why not take advantage of a great OS whose code is available? Linux is a member of the large family of UNIX-like OSs. A relative newcomer experienced sudden spectacular popularity starting in the late 1990s, Linux joins such well-known commercial Unix OSs as:

–System V Release 4 (SVR4) developed at AT&T (now owned by Novell)
–4.4 BSD release from the University of California at Berkeley
–Digital Unix form Digital Equipment Corporation (now Compaq)
–AIX from IBM
–HP-UX from Hewlett-Packard
–Solaris from Sun Microsystems

Linux was initially developed by Linus Torvalds in 1991 as an operating system for IBM-compatible personal computers based on the Intel 80386 microprocessors. Linus remains deeply involved with improving Linux, keeping it up-to-date with various hardware developments and coordinating the activity of hundreds of Linux developers around the world. Over the years, developers have worked to make Linux available on other architectures, including:

–Alpha
–SPARC
–Motorola MC680x0
–Power PC
–IBM System/390

One of the more appealing benefits to Linux is that it isn't a commercial OS: its source code, under the **GNU Public License**, is open and available to anyone to study. The official site of the Linux kernel is:

`http://www.kernel.org/`

If you download it, you will be able to explore from top to bottom one of the most successful, modern OSs.

There are many **distributions** of Linux. Perhaps the most popular one is RedHat (current version 7.2). If you have a RedHat distribution, you can download the latest version of the kernel from:

`www.redhat.com`

as RPM files:

```
kernel-2.4.7-10.src.rpm
kernel-source-2.4.7-10.i386.rpm
```

Linux source code for all supported architectures is contained in about 4500 C and Assembly files stored in about 270 subdirectories; it consists of about 2 million lines of code, which occupy more than 58Mb of disk space.

What do the kernel numbers mean?

$$\underbrace{2}_{\text{version nr}} \;.\; \underbrace{4}_{\text{stable kernel}} \;.\; \underbrace{7}_{\text{patchlevel}} - \underbrace{10}_{\text{release}}$$

Thus, the naming convention is:

major.minor.patchlevel

Linus Torvalds maintains the “official” kernel release. By convention, the even-numbered kernel versions (2.0, 2.2, and so on) are “stable” releases, patches that contain only bug fixes and no new features. Odd-numbered kernel versions (2.1, 2.3, and so on) are “development” releases.

New releases are made very rapidly, sometimes daily!

The main reason is that nearly all device drivers are contained within the kernel code, and every time someone updates a driver, a new release is necessary. As the Linux community moves toward loadable device drivers, the maintainers of those drivers can release them independently of the main kernel alleviating the necessity of such rapid updates.

If you want to see the history of kernel releases, go to:

`http://www.memalpha.cx/Linux/Kernel/`

Linux Distributions

Some of the most popular distributions of Linux are:

–Caldera OpenLinux
–Debian GNU/Linux
–Mandrake Linux
–Slackware Linux
–S.u.S.E. Linux

These are only the best known distributions; there is about 160 distributions at the last count!

Read the Distributions-HOWTO, for all the information about the different Linux Distributions.

The different HOWTOs are available from the **Linux Documentation Project** web site:

`http://www.linuxdoc.org/`

We mentioned GNU on the last slide. The GNU project is coordinated by the Free Software Foundation, Inc.:

`http://www.gnu.org/`

its aim is to implement a whole OS freely usable by everyone. The availability of a GNU C compiler has been essential for the success of the Linux project.

Technically speaking, Linux is a true UNIX kernel, although it is not a full UNIX OS, because it does not include all the applications such as filesystem utilities, windowing systems and graphical desktops, system administrator commands, text editors, compilers, and so on. However, since most of these programs are freely available under the GNU General Public License, they can be installed into one of the filesystems supported by Linux.

In order to define a common user interface, UNIX-like kernels often share fundamental design ideas and features. Thus, understanding the Linux Kernel will help to understand other UNIX variants.

All commercial variants of UNIX were derived from either SVR4 or 4.4BSD; all of them tend to agree on some common standards like:

–IEEE’s POSIX (Portable Operating Systems based on UNIX)
–X/Open’s CAE (Common Application Environment)

The 2.2 and up versions of the Linux kernel aim to be compliant with the IEEE POSIX standard. This means that most existing UNIX programs can be compiled and executed on a Linux system with very little effort or even without the need of patches to the source code. Moreover, Linux includes all the features of a moder UNIX OS, like:

–virtual memory
–virtual filesystem
–lightweight processes
–reliable signals
–SVR4 interprocess communication
–support for Symmetric Multiprocessor (SMP)
–etc.

By itself, the Linux Kernel is not very innovative. When Linus Torvalds wrote the first kernel, he referred to some classical books on UNIX internals, like Maurice Bach’s:

The Design of the Unix Operating System

(Prentice Hall, 1986).

Process

A **process** is a unit of work; it is a job. The "definition" of a process is: a process is a program in execution (an active entity).

As a process executes, it changes state. Each process may be in one of the following states:

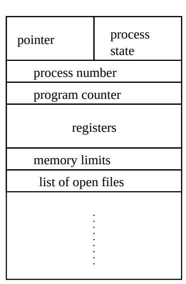
New: the process is being created.

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.



Process Control Block

Process State: the state may be new, ready, running, waiting, halted, etc.

Program counter: the address of the next instruction to be executed by this process.

CPU registers: depend on the computer architecture, but they include accumulator, index registers, stack pointers. Along with the program counter, this info must be saved when an interrupt occurs.

CPU-scheduling information: includes a process priority, pointers to scheduling queues, and other scheduling parameters.

Memory-management information.

Accounting Info: amount of CPU and real-time used, time limits, account numbers, job or process numbers, etc.

I/O status information: list of I/O devices allocated to this process, a list of open files, and so on.

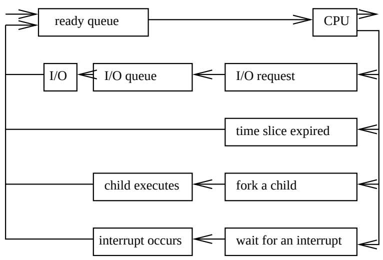
Process Scheduling

- The objective of multiprogramming is to have some process running at all times, so as to maximize CPU utilization.
- The objective of time-sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running.

As processes enter the system they are put into a **job queue**. This queue consists of all processes in the system. The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue**.

This queue is generally stored as a linked list. A ready-queue header contains pointers to the first (and last) PCB in the list. We extend each PCB to include a pointer field that points to the next PCB in the ready queue.

The list of processes waiting for a particular I/O device is called a **device queue**. Each device has its own device queue.



Queuing-diagram

A process migrates between the various scheduling queues throughout its lifetime. The OS must select processes from these queues; this is done by the appropriate **scheduler**.

Switching the CPU to another process requires saving the state of the old process and loading the saved state for the new process. This task is known as a **context switch**. The context of a process is represented in the PCB of that process.

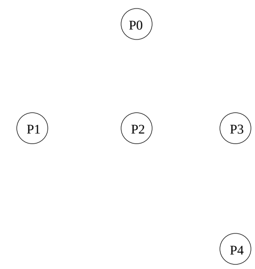
Context-switch times are dependent on hardware support. For instance, the Sun UltraSPARC provides multiple sets of registers, so a context-switch simply includes changing the pointer to the current register set. (When active processes exceed register sets, then the system must resort to copying register data to and from memory.)

Context switching has become such a performance bottleneck that programmers are using threads to avoid them whenever possible.

The processes in the system can execute concurrently, and they must be created and deleted dynamically. Thus the OS must provide a mechanism for process creation and termination.

A process (**parent**) may create several new processes (**children**).

For example, in Linux:



P0 creates P1,P2, and P3, and then P3 in turn creates P4.

The descriptor of each process P includes the following fields:

`p_opptr` points to the process that created P or to Process 1 (init) if the parent process no longer exists. For example, if a shell user starts a background process and exits the shell, the background process becomes child of init.

`p_cptr` points to the youngest child of P.

`p_ysptr` points to the youngest sibling, that is, to the process that has been created immediately after P by P's parent.

`p_osptr` points to the older sibling.

When a process creates a new process, two possibilities exist in terms of execution:

1. the parent continues to execute concurrently with its children

2. the parent waits until some or all of its children have terminated

There are two possibilities in terms of the address space of the new process:

1. the child process is a duplicate of the parent process

2. the child process has a program loaded into it

In Linux (UNIX) each process is identified by its **process identifier**, which is a unique integer. For example (`ps -A`):

PID	CMD
1	init
2	keventd
3	kapm-idled
4	ksoftirqd_CPU0
5	kswapd
6	kreclaimd
7	bdflush
8	kupdated
9	mdrecoveryd
13	kjournald
88	khubd
181	kjournald
712	adsl-connect
725	pppd
726	pppoe
850	syslogd
855	klogd
875	portmap
903	rpc.statd
1015	apmd

Init: is the parent of all processes. Its primary role is to create processes from a script stored in the file /etc/inittab. This file usually has entries which cause init to spawn gettys on each line that users can log in.

(`top`)

PID	PRI	SIZE	SHARE	STAT	CPU	MEM	COMMAND
1328	17	15916	2268	S	2.3	8.9	X
2294	11	1044	836	R	0.1	0.8	top
1	8	120	68	S	0.0	0.0	init
2	9	0	0	SW	0.0	0.0	keventd
3	9	0	0	SW	0.0	0.0	kapm-idled
5	9	0	0	SW	0.0	0.0	kswapd
6	9	0	0	SW	0.0	0.0	kreclaimd
7	9	0	0	SW	0.0	0.0	bdflush
8	9	0	0	SW	0.0	0.0	kupdated
9	-1	0	0	SW	0.0	0.0	mdrecoveryd
13	9	0	0	SW	0.0	0.0	kjournald
88	9	0	0	SW	0.0	0.0	khubd
181	9	0	0	SW	0.0	0.0	kjournald
712	9	616	456	S	0.0	0.5	adsl-connect
725	8	528	392	S	0.0	0.4	pppd
726	9	100	56	S	0.0	0.0	pppoe
850	9	212	156	S	0.0	0.1	syslogd
855	9	988	336	S	0.0	0.6	klogd
875	9	84	4	S	0.0	0.0	portmap
903	9	384	268	S	0.0	0.2	rpc.statd
1015	9	212	188	S	0.0	0.1	apmd
1080	9	172	4	S	0.0	0.0	cardmgr
1113	9	696	512	S	0.0	0.4	xinetd
1139	8	624	456	S	0.0	0.4	lpd
1167	9	1436	880	S	0.0	0.9	sendmail
1186	9	100	44	S	0.0	0.0	gpm
1204	9	224	156	S	0.0	0.1	crond
1254	9	3988	1012	S	0.0	2.7	xfs
1290	9	100	44	S	0.0	0.0	atd
1313	9	64	4	S	0.0	0.0	mingetty
1314	9	64	4	S	0.0	0.0	mingetty
1315	9	64	4	S	0.0	0.0	mingetty
1316	9	64	4	S	0.0	0.0	mingetty
1317	9	64	4	S	0.0	0.0	mingetty
1318	9	64	4	S	0.0	0.0	mingetty
1319	9	1380	1232	S	0.0	1.0	gdm
1327	9	1912	1564	S	0.0	1.4	gdm

PID The process ID of each task.

PRI The priority of the task.

SIZE The size of the task's code plus data plus stack space, in kilobytes.

SHARE The amount of shared memory used by the task.

STAT The state of the task is shown here. The state is either S for sleeping, D for uninterruptible sleep, R for running, Z for zombies, or T for stopped or traced.

CPU The task's share of the CPU time since the last screen update, expressed as a percentage of total CPU time per processor.

MEM The task's share of the physical memory.

COMMAND The task's command name.

Forking

Before you long in, your terminal is monitored by a

getty

process. After you log in, the getty process dies (a new one is started by the kernel when you log out) and your terminal is managed by your shell, which is a different process. The shell creates a new process each time you enter a command. The creation of a new process is called

forking

because one process splits into two.

If you are using the X Window System, each process starts up one or more windows. Thus, the window in which you are typing commands is owned by an xterm process. That process forks a shell to run within the window. And that shell forks yet more processes as you enter commands.

Thus, new processes are created by the `fork` system call. The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process.

Both processes (parent and child) continue execution at the instruction after the fork system call, with one difference: the return code for the fork system call is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.

For example:

```
#include <stdio.h>

void main()
{
        int pid;
        pid = fork();
        if (pid < 0){
                fprintf(stderr, "Fork Failed");
                exit(-1);
        }
        else if (pid == 0){
                execlp("/bin/ls","ls",NULL);
        }
        else{
                wait(NULL);
                printf("Child Complete\n");
                exit(0);
        }
}
```


The first argument of `execlp` is the filename of the command. The second argument is the command name. The third is the list of arguments for the command.

The `execlp` call overlays the existing program with the new one, runs that, then exits. The original program gets control back only when there is an error, for example if the file can't be found or is not executable.

The fork makes two copies of the program. In the child, the value returned by fork is zero, so it calls execlp, which does the ls and then it dies. In the parent, fork returns non-zero so it skips the execlp to the last condition. (If there is an error, fork returns -1.)

The parent waits for the child to terminate before continuing itself; this is done with the system call wait. The wait system call return the process identifier of a terminated child, so that the parent can tell which of its possibly many children has terminated. If the parent terminates, all its children have assigned as their new parent the init process.

A process terminates when it finishes executing its final statement and asks the OS to delete it by using the `exit` system call. At that point, the process may return data (output) to its parent process (via the `wait` system call). All the resources of the process (including physical and virtual memory, open files, and I/O buffers) are deallocated by the OS.

Termination can also occur if a parent process causes an `abort` sustem call. A parent may terminate the execution of one of its children for a veriety of reasons, such as:

- The child has exceeded its usage of some of the resources that it has been allocated.

- The task assigned to the child is no longer required.

- The parent is exiting, and the OS does not allow the child to continue without the parent.

Cooperating Processes

The concurrent processes executing in the OS may be independent or cooperating. A process is **independent** if it cannot affect or be affected by the other processes executing in the system (a process that does not share data with other processes is independent).

A process is **cooperating** if it can affect or be affected by the other processes executing in the system. Any process that shares data with other processes is cooperating.

The producer-consumer problem is a common paradigm for cooperating processes. A **producer** process produces information that is consumed by the **consumer** process. For example:

A print program produces characters which are consumed by the print driver.

When you invoke GNU CC, it normally does:
1. preprocessing (produces macro substiution, file inclusion, etc.)
2. compilation (produces assembly code)
3. assembly (produces object code)
4. linking

To allow produces and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer. A producer can produce one item, while the consumer is is consuming another item. The producer and consumer must be synchronized, so that the consumer does not try to consume an item which has not yet been produced.

The buffer may be either provided by the OS through the use of an **interprocess-communication (IPC)** facility, or be explicitly coded by the application programmer with the use of shared memory.

Shared-memory solution to the bounded-buffer problem

```
#define BUFFER_SIZE 10

typedef struct { ... } item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

The producer process has a local variable `nextProduced` in which the new item to be produced is stored:

```
while (1) {
        while (((in + 1) % BUFFER_SIZE) == out)
                ; /* buffer is full */
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
}
```

The consumer process has a local variable `nextConsumed` in which the item to be consumed is stored:

```
while (1) {
        while (in == out)
                ; /* buffer is empty */
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
```


PIPES
(an example of an IPC facility)

Pipes are an interprocess communication mechanism that is provided in all flavors of UNIX. A **pipe** is a one-way flow of data between processes: all data written by a process to the pipe is routed by the kernel to another process, which can then read it.

In UNIX command shells, pipes can be created by the means of the | operator. For instance, the following statement instructs the shell to create two processes connected by a pipe:

```
$ ls | more
```

The standard output of the first process, which executes the ls program, is redirected to the pipe; the second process, which executes the more program, reads its input from the pipe.

Note that the same result can also be obtained by issuing two commands such as the following:

```
$ ls > tmp
$ more < tmp
```

The first command redirects the output of ls into a regular file; then the second command forces more to read its input from the same file.

Using pipes instead of temporary files is usually more convenient since:

- The shell statement is much shorter and simpler.
- There is no need to create temporary files, which must be deleted later.

Pipes may be considered open files that have not corresponding image in the mounted filesystem. A new pipe can be created by means of the

`pipe()`

system call, which returns a pair of file descriptors. The process can read form the pipe by using the

`read()`

system call with the first file descriptor; likewise, it can write into the pipe by using the

`write()`

system call with the second descriptor.

POSIX defines only half-duplex pipes, so even though the `pipe()` system call returns two file descriptors, each process must close one before using the other. If a two-way flow of data is required, the processes must use two different pipes by invoking `pipe()` twice.

SVR4 allows full-duplex.

Resume the example: `ls | more`

When the command shell interprets the `ls | more` statement, it essentially performs the following actions:

1. Invokes the `pipe()` system call; assume that `pipe()` returns the file descriptor 3 (the pipe's *read channel*) and 4 (the pipe's *write channel*).
2. Invokes the `fork()` system call twice.
3. Invokes the `close()` system call twice to release file descriptors 3 and 4.

The first child process, which must execute the ls program, performs the following operations:

1. Invokes `dup2(4,1)` to copy file descriptor 4 to file descriptor 1. From now on, file descriptor 1 refers to the pipe's write channel.
2. Invokes the `close()` system call twice to release file descriptors 3 and 4.
3. Invokes the `execlp()` system call to execute the `/bin/ls` program. By default, such a program writes its output to the file having descriptor 1 (the standard output), that is, it writes into the pipe.

The second child process must execute the more program; therefore, it performs the following operations:

1. Invokes `dup2(3,0)` to copy file descriptor 3 to file descriptor 0. From now on, file descriptor 0 refers to the pipe's read channel.
2. Invokes the `close()` system call twice to release file descriptors 3 and 4.
3. Invokes the `execlp()` system call to execute `/bin/more`. By default, that program reads its input from the file having file descriptor 0 (the standard input); that is, it reads from the pipe.

Application programmers have a variety of needs that call for different communication mechanisms. Here are the basic IPCs for Linux:

- Pipes and FIFOs: Best suited to implement producer-consumer interactions among processes. Some processes fill the pipe with data while other extract data from the pipe.
- Semaphores: the User Mode version of the kernel semaphores.
- Messages: Allow processes to exchange messages (short blocks of data) in an asynchronous way. They can be thought of as signals carrying additional information.
- Shared memory regions: Best suited to implement interaction schemes in which processes must share large amounts of data in an efficient way.
- Sockets: They were introduced initially to allow data communication between application programs and the network interface. They can also be used as a communication tool for processes located on the same host computer; the X Window System graphic interface, for instance, uses a socket to allow client programs to exchange data with the X server.

Threads

A **thread**, sometimes called a **lightweight process (LWP)** is a basic unit of CPU utilization. It comprises:

–thread ID
–program counter
–register set
–stack

It shares with other threads belonging to the same process its code section, data section, and other OS resources, such as open files and signals.

Many software packages are **multi-threaded**. They are implemented as a single process with several threads of control.

Example: a web browser might have one thread displaying images or text, while another thread retrieves data from the network. A server may be running several threads to service different requests from clients.

The benefits of multithreaded programming can be broken into four major categories:

–Responsiveness: multi-threading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation.

–Resource sharing: threads share the memory and the resources of the process to which they belong. The benefit of code sharing is that it allows an application to have several different threads of activity all within the same address space.

–Economy: in general it is much more time consuming to create and manage processes than threads. It is not as costly to context switch threads.

–Utilization for multiprocessor architectures: each thread may be running in parallel on a different processor. A single threaded process can only run on one CPU, no matter how many are available.

User Threads: are supported above the kernel and are implemented by a thread library at the user level. The library provides support for thread creation, scheduling, and management with no support from the kernel.

Kernel Threads: are supported directly by the OS: the kernel performs thread creation, scheduling, and management in kernel space. Since the kernel is doing the scheduling, if a thread performs a blocking call, the kernel can schedule another thread in the application for execution.

Linux provides a

`fork`

system call with the traditional functionality of duplicating a process. Linux also provides the

`clone`

system call that is analogous to creating a thread. `clone` behaves much like `fork`, except that instead of creating a copy of the calling process, it creates a separate process that shares the address space of the calling process. It is through this sharing of the address space of the parent process that a cloned task behaves much like a separate thread.

Linux does not distinguish between processes and threads. In fact, Linux often uses the term **task** when referring to a flow of control within a program.

Aside from the cloned process, the Linux kernel does not support multi-threading. However, various Pthreads implementations are available for user-level multi-threading.

Pthreads

Pthreads refers to the POSIX standard definign an API for thread creation and synchronization. OS designers may implement the specification in any way they wish. Generally, libraries implementing the Pthread specifications are restricted to UNIX OSs.

In Linux, the libaray specifications for Pthreads are located in `/usr/include/pthread.h`; Linuxthreads is a simple `clone()`-based implementation of Posix threads.

We give as example a C program that creates a separate thread that determines the summation of a non-negative integer. In a Pthread program, separate threads begin execution in a specified function.


```
1  #include <pthread.h>
2  #include <stdio.h>

3  int sum;
4  void *runner(void *parm);

5  main(int argc, char *argv[])
6  {
7          pthread_t tid;
8          pthread_attr_t attr;
9
10         pthread_attr_init(&attr);
11         pthread_create(&tid,&attr,runner,argv[1]);
12         pthread_join(tid,NULL);
13         printf("sum = %d\n",sum);
14 }

15 void *runner(void *parm)
16 {
17         int i;
18         sum = 0;
19         for (i = 1; i <= atoi(parm); i++)
20                 sum += i;
21         pthread_exit(0);
22 }

$ gcc example.c -o example -lpthread
$ ./example 20
sum = 210
```


1 Include the Pthreads library; all Pthread programs must include `pthread.h`

5 When the program begins, a single thread of control begins in `main`. After some initialization, `main` creates a second thread that begins control in the `runner` function.

7 Declare the identifier for the thread we will create.

8 Each thread has a set of attributes including stack size and scheduling information.

10 Set the attributes; because we did not explicitly set any attributes, we use the default attributes provided.

11 A separate thread is created: we pass the thread identifier, the attributes for the thread, the function name where the thread will start execution (`runner` function), and the integer parameter provided on the command line (`argv[1]`).

12 At this point, the program has two threads: the initial thread in `main` and the thread performing the summation in the runner function. After creating the second thread, the main thread will wait for the runner thread to complete by calling the pthread_join function. The runner thread will complete when it calls the function pthread_exit.

The pthreads library is a set of C calls that provides a mechanism for writing multithreaded code. There are three major classes of calls in this library:

1. calls to create and destroy threads
2. calls to synchronize threads and lock program resources
3. calls to manage thread scheduling

Thread scheduling can considerably complicate your algorithm, and it can also get you in trouble when migrating the code from uniprocessor to multiprocessor machines.

Each thread sees the same global data and files; however, each thread has its own stack and registers.

```
+-----------------------------------------------------+
|                                                     |
|                       Process                       |
|                                                     |
|+-------+  +-----------+ +-----------+ +-----------+|
|| Files |  | Thread    | | Thread    | | Thread    ||
|+-------+  |+---------+| |+---------+| |+---------+||
|           || Registers|| || Registers|| || Registers|||
|           |+---------+| |+---------+| |+---------+||
|           |           | |           | |           ||
|.....................................................|
|           |           | |           | |           ||
|  Memory   |           | |           | |           ||
|           | +-------+ | | +-------+ | | +-------+ ||
|+--------+ | | Stack | | | | Stack | | | | Stack | ||
||  Heap  | | |       | | | |       | | | |       | ||
|+--------+ | |       | | | |       | | | |       | ||
|           | |       | | | |       | | | |       | ||
|+--------+ | |       | | | |       | | | |       | ||
|| Static | | |       | | | |       | | | |       | ||
|+--------+ | |       | | | |       | | | |       | ||
|           | |       | | | |       | | | |       | ||
|+--------+ | |       | | | |       | | | |       | ||
||  Code  | | +-------+ | | +-------+ | | +-------+ ||
|+--------+ +-----------+ +-----------+ +-----------+|
|                                                     |
|                                                     |
+-----------------------------------------------------+
```

From the DECthreads manual, from the POSIX Threads tutorial by Mark Hays.

Another example of thread creation and destruction: the "Hello, world" program. The flow of the program is as follows:

```
Time     Master Thread
  |            |
  |            |        create workers with
  |            |        pthread_create()
  |            |
  |          // \\      workers start up
 \|/        / | | \
            | | | |
            | | | |     workers do their jobs
            | | | |
            \ \ / /
             \\ //      workers terminate
               |
               |        join workers with
               |        pthread_join()
               |
         Master Thread
```

Starting from an initial thread in `main()`, we create several worker threads. Each of these will print its message while the original thread waits for them to complete. As each worker thread exits, the master thread will join it and destroy the system resources it uses. Finally, the original thread will call `exit()` and the whole process will terminate.


```
5  #define NTHREADS 4
6  void *hola(void * arg) /* this is the thread code */
7  {
8    int myid=*(int *) arg;
9
10   printf("Hello, world, I'm %d\n",myid);
11   return arg;
12 }
13 /* this is the main thread's code */
14 int main(int argc,char *argv[])
15 {
16   int worker;
17   pthread_t threads[NTHREADS];/* holds thread info  */
18   int ids[NTHREADS];          /* holds thread args  */
19   int *status;                /* holds return code  */
20   /* create the threads */
21   for (worker=0; worker<NTHREADS; worker++) {
22     ids[worker]=worker;
23       pthread_create(&threads[worker],/*thread struct */
24                    NULL, /* default thread attributes */
25                    hola,         /* start routine */
26                    &ids[worker]))/* arg to routine */
27   }
28   /* reap the threads as they exit */
29   for (worker=0; worker<NTHREADS; worker++) {
30     /* wait for thread to terminate */
31     pthread_join(threads[worker],(void *) &status))
32   }
33   return(0);
34 }
```


The program starts in `main()` on line 14.

It will create NTHREADS threads (NTHREADS is defined on line 5) on line 23, each of which starts executing the function `hola()` on line 6.

The main thread waits for each of the worker threads to terminate by calling `pthread_join()` for each thread on line 31.

Meanwhile, each of the worker threads prints a message and its thread ID and then exits. Once the main thread has reaped all of the workers, the main program exits.

Threads are created with the `pthread_create()` call. The first argument is a pointer to a `pthread_t`, which is an opaque data structure (i.e., it is a black box) that describes the thread.

Each thread must have a unique `pthread_t` to describe it. In this case, we store the descriptor for each thread in the array `threads[]` defined on line 17.

The second argument to `pthread_create()` is of type `pthread_attr_t` and specifies the attributes of the thread being created (scheduling priority, etc). Normally you can use the value NULL for this argument.

The third argument is the name of the function in which the thread should start — this function is expected to return a `void *`.

The last argument is passed untouched to the start routine. In this case, we just pass the thread's number (loop count) to the start routine. In more complicated situations, you can pass a pointer to a block of thread-specific data.

`pthread_create()` returns 0 on success and an error number on failure.

The main thread next waits for each of the workers to finish in lines 29–32. It calls `pthread_join()` to wait for the thread to terminate. The thread we're waiting on is specified in the first argument and its exit code is placed into the second argument (declared on line 19 as type int *).

The code that each worker thread executes occupies lines 6–12. First, we cast the argument to the start routine to an int (since we passed a pointer to an int to `pthread_create()`). The thread then prints its message and returns its thread ID. At this point, the thread terminates and is reaped when the master thread calls `pthread_join()`. You can also have a thread terminate itself with the `pthread_exit()` call.

The call `pthread_join()` causes the calling thread to go to sleep until the specified thread terminates. In the examples given in this document, the master thread cannot proceed until all the workers complete their tasks. Once a thread has been joined, its system resources are reclaimed. If you need to terminate a running thread, use `pthread_cancel()`.

Once you join a thread, you cannot reassign it a new task. The reason is that once a thread has been joined, it has (by definition) terminated. If you need more threads to perform another task, you have to call `pthread_create()` again.

Aside from returning a thread's exit status, the major function of `pthread_join()` is to reclaim the system resources used by the terminated thread. If you forget to call `pthread_join()` in a program that creates many threads, you will run into problems sooner or later (like what happens when you call `malloc()` many times and never call `free()`).

From last week:

In a threaded environment, instead of there being several singly threaded processes, we have one process consisting of multiple threads. In other words, each user process can have several threads of execution — different parts of your code can be executing simultaneously. The collection of threads form a single process: each thread in the process sees the same virtual address space, files, etc.

This last fact has interesting implications. For example, when one thread opens a file, it is immediately possible for all other threads to access the file. Similarly, if one thread alters the value of a global variable, all other threads instantly see the new value.

When writing multithreaded applications, you must take the initiative to protect shared resources. For example, consider the code:

```
i = i + 1
```

This statement is well defined for a single thread of execution: get `i`, add 1, and put the result back. For multiple threads of execution, the statement is **not** well defined!

If 2 or more threads attempt to perform the operation:

```
i = i + 1
```

simultaneously, the result is **indeterminate**.

The variable `i` will get incremented anywhere from one to N times, where N is the number of threads that execute this statement.

In a multithreaded environment, it is necessary for the programmer to ensure that access to shared resources is managed in a well defined fashion.

In order to deal with the "`i=i+1` problem", it is necessary to invent some kind of locking mechanism that serializes access to the variable `i`.

Pthreads provides an object called a **mutex** (short for "mutual exclusion") to facilitate resource locking. Once you create a mutex, there are two things you can do with it:

lock it
unlock it

Mutexes have the property that they can only be locked by one thread at a time. If two threads attempt to lock the same mutex, only one of them succeeds. The other one blocks until the first thread releases the mutex.

In general, you use mutexes to protect some piece of shared data. In order to be useful, the mutex itself must be global so that all threads can see it (this is true of all pthreads synchronization objects).


```
int i;                          /* i is shared */
pthread_mutex_t ilock;          /* controls access to i */

void func()
{


pthread_mutex_lock(&ilock);  /* lock i */
something = i++;             /* access i */
pthread_mutex_unlock(&ilock);/* unlock i */


}
```

It is up to the programmer to determine exactly what the mutex protects. Here the mutex `ilock` is used to protect `i` (it serializes the access to `i`).

All global data that can be read and written by simultaneously executing threads must be protected by a mutex.

In order to minimize ill effects, it is important to "get in and out" as quickly as possible. For example, if a thread needs to read some global data, consider copying the data into the thread's private storage instead of keeping a mutex locked for a long period of time.

Or, if you need to update a piece of global data, calculate (if possible) the new value with the mutex unlocked – lock it only when you dump the new value in.

Why Use Threads?

Why should you use threads? Well, maybe you shouldn't. Developing a threaded program can take longer than a nonthreaded program. If you don't need the benefits of threads or don't intend to run your code many times, it's probably not worth the trouble. On the other hand, threading your code has several payoffs:

Speed: Communication between threads is very fast. If your algorithm is bound by communication speed, threads will put a smile back on your face.

I/O throughput: In a traditional UNIX process, a blocked I/O request stops the whole process dead in its tracks. Things are much nicer in POSIX threads: a blocked I/O request only puts the calling thread to sleep – the other threads go about their business.

Portability: POSIX threads are fairly portable. They can be emulated on uniprocessor systems at a slight (or sometimes no) penalty in speed. On shared memory multiprocessor machines, there is often kernel support for threads; this means that your application can take full advantage of all available CPUs and achieve (near) maximum performance.

Practice Exercises:

From the textbook: 5.1–6

5.1 Provide two programming examples of multithreading that improve performance over a single-threaded solution:

–A Web server that services each request in a separate thread

–A parallelized application such as matrix multiplication where different parts of the matrix may be worked in parallel.

–An interactive GUI program such as a debugger where a thread is used to monitor user input, another thread represents the running application, and a third thread monitors performance.

5.3 What are two differences between user-level threads and kernel-level threads?

–User-lever threads are unknown by the kernel, whereas the kernel is aware of kernel threads.

–User threads are scheduled by the thread library (such as Pthreads for C) and the kernel schedules kernel threads.

–Kernel threads need not be associated with a process whereas every user thread belongs to a process.

5.4 Describe the actions taken by a kernel to context switch between kernel-level threads.

Context switching between kernel threads typically requires saving the value of the CPU registers from the thread being switched out and restoring the CPU registers of the new thread being scheduled.

5.6 What resources are used when a thread is created? How do they differ from those used when a process is created?

Because a thread is smaller than a process, thread creation typically uses fewer resources than process creation. Creating a process requires allocating a process control block (PCB), a rather large data structure. The PCB includes a memory map, list of open files, and environmental variables. Allocating and managing the memory map is typically the most time-consuming activity. Creating either a user or kernel thread invloves allocating a small data structure to hold a register set, stack, and priority.

Developing New Operating Systems

(from a COMPUTERWORLD article by Gary H. Anthes)

Research and development work on operating systems, broadly defined, is going on in many quarters. Here are some projects:

The Legion Project:

Researchers at the University of Virginia are developing Legion, a highly flexible, wide-area operating system designed to build a virtual computer from millions of distributed hosts and trillions of objects while presenting to the user the image of a single computer. Legion applies many of the object-based interoperability principles in the Common Object Request Broker Architecture. It finds and schedules resources and handles security issues among disparate operating systems and objects written in different languages, freeing users from the need to negotiate with outside systems and administrators.

The Globus Project:

This is a university/government research and development project focused on applying grid concepts to scientific and engineering computing. The grid refers to an infrastructure that enables the integrated, collaborative use of high-end computers, networks, databases and scientific instruments owned and managed by multiple organizations. Grid applications often involve large amounts of data and/or computing and often require secure resource sharing across organizational boundaries.

Globus research areas include resource management, data management and access, application development environments, information services and security. The project has developed the Globus Toolkit, a set of services and software libraries to support Grids and Grid applications. The Globus Toolkit includes software for security, information infrastructure, resource management, data management, communication, fault detection and portability.

The Endeavour project:

Researchers involved in the Endeavour project at the University of California, Berkeley, envision an "information utility" that adapts on the fly to the supply and demand for computing resources anywhere in the world. "Fluid software" will choose where to execute and where to access storage. It will requisition resources as needed and negotiate interoperation agreements, ensuring always-available processing. Endeavour is a collection of projects. One, called OceanStore, will provide "nomadic data access – serverless, homeless and freely flowing through the network" – that will make the storage available to a user essentially infinite, according to researchers.

The WebOS project:

The goal of this research effort at Duke University in Durham, N.C., is to provide a common set of operating system services to wide-area applications, including mechanisms for resource discovery, a global name space, remote process execution, resource management, authentication and security. WebOS provides basic operating system services needed to build applications that are geographically distributed, highly available, incrementally scalable and dynamically reconfiguring.

The Globe Project:

Researchers involved in The Globe Project at Vrije University in Amsterdam are working on construction of large-scale wide-area distributed systems enabled by distributed shared objects. In the Globe model, the universe consists of a vast number of shared objects, each with some associated methods. Users of an object may invoke an object's methods. A method can return the contents of a Web page, but it can also accept an e-mail or news message, look up a name in a worldwide distributed database, access a file and perform other actions. Objects are distributed, with active copies on multiple machines at the same time, and they use peer-to-peer communication.

MIT Project Oxygen:

Sponsored by the U.S. Department of Defense, this project at MIT is intended to bring computation and communication to users through natural spoken and visual interfaces, in order to make it easy for users to collaborate, access knowledge and automate repetitive tasks. The Oxygen system rests on an infrastructure of mobile and stationary devices connected by a self-configuring network. Researchers say the Oxygen system must be:

- Pervasive: It must be everywhere, with every portal reaching into the same information base.
- Embedded: It must live in its users' world, sensing and affecting it.
- Nomadic: Its users and computations must be free to move around according to their needs.
- Eternal: It must never shut down or reboot; components may come and go in response to demand, errors and upgrades, but Oxygen as a whole must be nonstop.

Correction to Question 2b on Test 1

Count the number of faults that OPT incurs on $\sigma_{k,i}$. Suppose, for example, that $k = 2, i = 4$. Then:

$$\sigma_{2,4} = 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3$$

My solution said that the number of faults is OPT($\sigma_{k,i}$) $= k + i = 6$. However:

$$\left|\begin{array}{c|c|c} 1 & 1 & 1 \\ & 2 & 3 \\ \text{x} & \text{x} & \text{x} \end{array}\right|, \left|\begin{array}{c|c|c} 1 & 2 & 2 \\ 3 & 3 & 3 \\ & \text{x} & \end{array}\right|, \left|\begin{array}{c|c|c} 2 & 2 & 3 \\ 1 & 1 & 1 \\ \text{x} & & \text{x} \end{array}\right|, \left|\begin{array}{c|c|c} 3 & 3 & 3 \\ 1 & 2 & 2 \\ & \text{x} & \end{array}\right|$$

So, the number of faults is 7, not 6. It turns out that every k segments (AFTER the first segment) there is one extra fault. Thus, the correct formula for OPT($\sigma_{k,i}$) is the following:

$$k + i + \left\lfloor \frac{i-1}{k} \right\rfloor$$

(the $i - 1$ comes from counting the segments without the first segment, and then, to add up the extra faults that occur every k-th segment, we divide by k and take the floor). Try this with $\sigma_{4,9}$ to see this extra fault on another example.

Note that this is a "small" addition to the number of faults incurred by OPT. Thus, 2c is still correct (note that OPT($\sigma_{k,i}$) $= k + i + \lfloor \frac{i-1}{k} \rfloor \leq 2k + i$, so now:

$$\text{FIFO}(\sigma_{k,i}) \leq \left(\frac{i(k+1)}{2k+i} \right) \text{OPT}(\sigma_{k,i})$$

so the same analysis still holds.

CPU Scheduling

CPU Scheduling is the basis of multiprogrammed OSs. By switching the CPU among processes, the OS can make the computer more productive.

The object of multiprogramming is to have some process running at all times, in order to maximize CPU utilization. A process is executed until it must wait, typically for the completion of some I/O request. When a process has to wait, the OS gives the CPU to another process.

Scheduling is a fundamental OS function. Almost all computer resources are scheduled before use. The CPU is, of course, one of the primary computer resources. Thus, CPU scheduling is central to OS design.

Process execution consists of a **cycle** of CPU execution and I/O wait. Processes alternate between these two states. Process execution begins with a **CPU burst**, which is then followed by an **I/O burst**, and so on, they alternate.

The durations of these CPU bursts have been extensively measured and researched, but it is difficult to make general assertions about them. However, they can be roughly characterized as follows:

–few long CPU bursts
–many short CPU bursts
–many very short I/O bursts

Whenever the CPU becomes idle, the OS must select one of the processes in the ready queue to be executed. The selection process is carried out by the **CPU scheduler**.

CPU scheduling decisions may take place under the following four circumstances: a process switches

1. running state → waiting state (e.g., I/O request)
2. running state → ready state (e.g., interrupt)
3. waiting state → ready state (e.g., I/O completion)
4. or a process terminates

In 1 and 4 a new process **must** be selected; when scheduling takes place only under circumstances 1 and 4, it is **nonpreemptive**. Otherwise, it is **preemptive**.

Under nonpreemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting queue.

Windows 3.1 and Apple Macintosh are nonpreemptive. Linux allows only user processes to be preempted.

The **dispatcher** is the module that gives control of the CPU to the process selected by the CPU scheduler. His work involves:

- Switching control
- Switching to user mode
- Jumping to the proper location in the user program to restart that program.

The dispatcher should be as fast as possible, given that it is involved during every process switch. The time taken by the dispatcher to stop one process and start another is known as the **dispatch latency**.

Scheduling Criteria

- **CPU utilization:** Want to keep the CPU as busy as possible.
- **Throughput:** The number of processes completed per time unit.
- **Turnaround time:** The time interval from the submission of a process to its completion.
- **Waiting time:** It is the sum of the periods spent waiting in the ready queue.
- **Response time:** The time from the submission of a request until the first response is produced.

In most cases, we want to optimize the average measure. In some cases, we want to optimize max and min values, such as response time, so that users get good service.

Scheduling Algorithms

First-Come, First-Served (FCFS)

With this scheme, the process that requests the CPU first is allocated the CPU first.

Implementation: FIFO queue.

When a process enters the ready queue, its PCB is linked to the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue, and the running process is removed from the queue.

Problem: Average waiting time quite long. For example:

Processes (in order of arrival)	Burst Time
P_1	24
P_2	3
P_3	3

(Gantt chart)

Convoy effect: Many short processes wait for one long process.

The FCFS algorithm is nonpreemptive. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU. It is not very good for time-sharing systems, where each user needs to get a share of the CPU at regular intervals.

Shortest-Job-First Scheduling (SJF)

This algorithms associates with each process the length of its next CPU burst.

When the CPU is available, it is assigned the process that has the smallest next CPU burst. If two process have the same length next CPU burst, FCFS scheduling is used to break the tie.

For example:

Processes (in order of arrival)	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

SJF is *provably optimal*, in that it gives the minimum average waiting time for a given set of processes.

The Proof: essentially the same idea as the proof of the fact that OPT page-replacing algorithm is Optimal! Also, the same problem: in an online setting, when we don't know what processes are coming up, how do we compute future Burst Times?

Although SJF is optimal, it cannot be implemented at the level of CPU scheduling; there is no way to know the length of the next CPU burst.

What did we do in the context of page-replacing? We approximated the future, by looking at the past: LRU.

We predict the value of the next Burst Time. We make the assumption that the next CPU burst will be similar in length to the previous ones.

The next CPU burst is generally predicted as an exponential average of the measured lengths of previous CPU bursts.

Let t_n be the length of the n-th CPU burst, and let τ_{n+1} be our predicted value for the next CPU burst. Then, for $0 \leq \alpha \leq 1$, define:

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$$

This formula defines the **exponential average**. The value t_n contains the most recent information, and τ_n stores the past history (let $\tau_1 = t_1$).

The parameter α controls the relative weight of recent and past history in our prediction. If $\alpha = 0$, then $\tau_{n+1} = \tau_n$, and the recent history has no effect (current conditions are assumed to be transient).

If $\alpha = 1$, then $\tau_{n+1} = t_n$, and only the most recent CPU burst matters (history is assumed to be old and irrelevant).

The SJF algorithm may be either preemptive or nonpreemptive. The choice arises when a new process arrives at the ready queue 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.

Preemptive SJF scheduling is called **shortest-remaining-time-first** scheduling.

The SJF algorithm is a special case of the general **priority-scheduling algorithm**. A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order.

The SJF algorithm is simply a priority algorithm where the assigned priority is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority.

Priorities

In general, process priorities can be defined either internally or externally.

Internal: time limits, memory requirements, nr of open files, etc.

External: importance of a process, often political in nature.

Priority scheduling can be preemptive or nonpreemptive. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process.

A preemptive algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.

A nonpreemptive algorithm will put the new process at the head of the ready queue.

Problem with priority scheduling: **Starvation**. A process has a very low priority, and is never scheduled.

Solution: **Aging**. Gradually increase the priority of the processes that wait in the system for a long time.

Round-Robin Scheduling (RR)

- Designed especially for time-sharing systems.
- Similar to FCFS scheduling, but preemption is added to switch between processes.
- A **time slice** is defined, usually between 10 and 100 milliseconds. The ready queue is treated as a circular queue, the scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time slice
- To implement the RR scheduling, we keep the ready queue as a FIFO queue. New processes are added at the tail. The scheduler picks the first process from the queue, sets a timer to interrupt after one time slice, and dispatches the process.
- One of two things will happen:
  1. The process has a CPU burst of less than 1 time slice. Then, the process itself will release the CPU voluntarily.
  2. The timer will go off, and cause an interrupt. A context switch will be executed, and the process will be put at the tail of the ready queue.

- If the time slice $= \infty$, then RR=FCFS.
- If the time slice is very small (1 microsecond), then the RR approach is called **Processor Sharing**, and "appears" as if each of n processes had its own processor, running at the speed of $1/n$ of the real processor (in theory, because we have to take into account context switches—we want time slices to be bigger than contex-switch time).

Linux

Linux provides two separate process-scheduling algorithms:

1. A time-sharing algorithm for fair preemptive scheduling among multiple processes.
2. An algorithm for real-time tasks where absolute priorities are more important than fairness.

Linux allows only processes running in user mode to be preempted. A process may not be preempted while it is running in kernel mode, even if a real-time process with a higher priority is available to run.

In Linux, part of every PCB is a scheduling class, that defines which of the two algorithms to apply to the process.

For time-shared processes, Linux uses a prioritized, credit-based algorithm. Each process possesses a certain number of scheduling credits.

When a new task must be chosen to run, the process with the most credits is selected. Every time that a time interrupt occurs, the currently running process looses one credit. When the number of credits reaches zero, it is suspended and another process is chosen.

If no runnable processes have any credits, then Linux performs a recrediting operation, adding credits to every process in the system (rather than just to the runable ones), according to the following rule:

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

This algorithm mixes two factors:

1. history
2. priority

One-half of the credits that a process still holds since the previous recrediting operation will be retained after the algorithm has been applied. This way, it retains some history of recent behaviour.

Background batch jobs can be given a low priority. Interactive, or I/O bound processes, for which a rapid response time is important, can be given a higher priority.

Exercises: 6.1, 6.2, 6.3, 6.4, 6.5, 6.7.

6.1 Given n processes to be scheduled on one processor, how many different schedules are possible?

$n!$

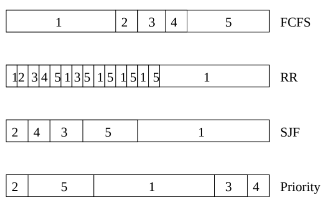
6.2 Define the difference between preemptive and nonpreemptive scheduling.

Preemptive scheduling allows a process to be interrupted in the midst of its execution, taking the CPU away and allocating it to another process. Nonpreemptive scheduling ensures that a process relinquishes control of the CPU only when it finishes with its current CPU burst.

6.3(a) Consider the following set of processes:

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	3
P_4	1	4
P_5	5	2

Draw four Gantt charts illustrating the execution of these processes using FCFS, SJF, a nonpreemptive priority (a smaller priority number implies a higher priority), and RR (time slice = 1) scheduling.



The UNIX File System

- Everything in the UNIX system is a file.
- When the first version of the system was being designed, before it even had a name, the discussion focused on the structure of a file system that would be clean and easy to use.
- The file system is central to the success and convenience of UNIX. It is a good example of the “keep it simple” philosophy, and the power achieved by careful implementation of a few well chosen ideas.
- Details of using the file system:
  –what files are
  –how they are represented
  –directories and the file system hierarchy
  –permissions
  –inodes (system’s internal record of files)
  –device files

- A file is a sequence of bytes. No structure is imposed on the file by the system, and no meaning is attached to its contents—the meaning of the bytes depends solely on the program that interprets the file.
- This is true not just of disc files, but of peripheral devices as well. Magnetic tapes, mail messages, characters typed on the keyboard, line printer output, data flowing in pipes—each of these files is just a sequence of bytes as far as the system and the programs in it are concerned.

Create a file:

```
~$ ed
a
now is the time
for all good people
.
w junk
36
q
~$ ls -l junk
-rw-rw-r--    1 you   you    36 Mar  9 07:56 junk
~$
```

junk is a file with 36 bytes—the 36 characters typed while appending. To see the file:

```
~$ cat junk
now is the time
for all good people
~$
```

The command od (octal dump) prints a visible representation of all the bytes of a file:

```
~$ od -c junk
0000000   n   o   w       i   s       t   h   e
0000020   f   o   r       a   l   l       g   o
0000040   p   l   e  \n
0000044
~$
```

- The 7-digit numbers down the left side are positions in the file, that is, the ordinal numbers of the next character shown, in octal. (The emphasis on octal numbers is a holdover from the PDP-11, for which octal was a preferred notation.)
- Each line ends with a character with octal value 012. This is the ASCII newline character; it is what the system places in the input when you press the RETURN key. By a convention borrowed from C, the character representation of a newline is `\n`. But this is just a convention to make it easy to read—the value stored in the file is the single byte 012.
- Other special characters include backspace (octal value 010, printed as `\b`), tab (011, `\t`), etc.
- It is important to distinguish between how a character is stored in a file and how it is interpreted in various situations. For example, when you type a backspace on your keyboard, the kernel intrprets it to mean that you want to discard whatever character you typed previously. Both that character and the backspace disappear, but the backspace is echoed to your terminal, where it makes the cursor move one position backward. (You can insert a backspace character in a file by typing \← .)

- The UNIX system is unusual in its approach to representing control information, particularly its use of newlines to terminate lines.
- Many systems provide instead "records," one per line, each of which contains not only your data but also a count of the number of characters in the line (and no newline).
- Other systems terminate each line with a carriage return and a newline, because that sequence is necessary for output on most terminals. (The word "linefeed" is a synonym for newline, so this sequence is often called "CRLF.")
- The UNIX system does neither—there are no records, no record counts, and no bytes in any file that you or your programs did not put there. A newline is expanded into a carriage return and a newline when sent to a terminal, but programs need only deal witht he single newline character, because that is all they see.
- The kernel keeps track of file lengths, so there is no need to add a special end-of-file character at the end; a program encounters end-of-file when it has processed all the bytes in the file.

- Programs retrieve the data in a file by a system call (a subroutine in the kernel) called

  `read`

  Each time `read` is called, it returns the next part of a file. It also says how many bytes of the file were returned, so end of file is assumed when 0 bytes are returned.
- This scheme makes a lot of sense: all files must end. Since all files must be accessed through `read`, returning zero is an interpretation independent way to represent the end of file without introducing a new special character.
- **What's in a file?** The format of a file is determined by the program that uses it; it is not determined by the file system. Thus, the kernel cannot tell you the type of a file, because it does not know it.
- The command `file` makes an educated guess:

```
~$ file /bin /bin/ed junk
/bin:    directory
/bin/ed: ELF 32-bit LSB executable, Intel 80386, version 1,
dynamically linked (uses shared libs), stripped
junk:    ASCII English text
~$
```


- To determine the file types, `file` didn't pay any attention to the file names (although it could have), because naming conventions are just conventions.
- Instead, `file` reads the first few hundred bytes of a file and looks for clues to the file type. Sometimes the clues are obvious: a runnable program is marked by a binary "magic number" at its beginning. `od` with no options dumps the file in 16-bit (2-byte) words and makes the magic number visible:

```
~$ od /bin/ed
0000000 042577 043114 000401 000001 000000 000000 000000 000000
0000020 000002 000003 000001 000000 110600 004004 000064 000000
0000040 014464 000001 000000 000000 000064 000040 000006 000050
...
```

  The octal value 42577 marks an executable. Specific magic numbers are system dependent.
- For text files, the clues may be deeper in the file, so `file` looks for words line `#include` to identify C source, or line beginning with a period to identify `nroff` or `troff` input.

Directories and Filenames

- All files have unambigous names. If you have a file `/home/you/junk` and you type `ls junk`, then you obtain: `junk`, not `/home/you/junk`, why?

- Because each process has a **current directory**, and all file names are implicitly assumed to start with the name of that directory, unless they begin directly with a slash. Your login shell and `ls`, therefore have a curent directory. The command `pwd` (print working directory) identifies the current directory:

```
$ pwd
/home/you
$
```

- The current directory is an attribute of a process, not a person or a program—people have login directories, processes have current directories. If a process creates a child process, the child inherits the current directory of its parent. If the child changes to a new directory, the parent is unaffected.

- The notion of current directory is a notational convenience, but its real purpose is organizational.

Permissions

- Every file has a set of **permissions** associated with it, which determine who can do what with the file. However, there is a special user, the **super-user**, who can read and modify any file in the system, (login name `root`.)

- You can, however, encrypt files in your directory (using PGP, for example). It is believed that in real life, most security breaches are due to passwords that are given away or easily guessed.

- When you log in, you type a name and a password, The system actually recognizes you by your user-id, **uid**. Besides a uid you are assigned a **group-id**, which places you in a class of users. The file system, and the UNIX system in general, determines what you can do by the permissions granted to your uid and group-id.

- The file `/etc/passwd` is the password file; it contains all the login information about each user. You can discover your uid and group-id, as does the system, by looking up your name in `/etc/passwd`:

```
$ grep you /etc/passwd
you:gkmbCTrJ04COM:500:500:You:/home/you:/bin/bash
$
```

  The fields in the password file are separated by colons and are laid out like this:

  login-id:encrypted-password:uid:group-id:miscellany:login-directory:shell

- The file is ordinary text, but the field definitions and separator are a convention agreed upon by the programs that use the information in this file.

- Note that your password appears here in an encrypted form, since anyone can read this file. When you give your password at login, it is encrypted and compared with the results in `/etc/passwd`.

-

```
~$ ls -l /etc/passwd
-rw-r--r-- 1 root root 1271 Feb 12 12:03 /etc/passwd
~$
```


- Anybody can open `/etc/passwd` and make changes, but only root can write the changes back. The program to change passwords is called `passwd` and:

```
~$ ls -l /usr/bin/passwd
-r-s--x--x 1 root root 13476 Aug 7 2001 /usr/bin/passwd
~$
```

- The `s` instead of `x` in the execute field for the file owner states that, when the command is run, it is to be given the permissions corresponding to the file owner, in this case root. Because `passwd` is "set-uid" to root, any user can run the `passwd` command to edit the password file.

- The set-uid bit is a simple and elegant idea (patented by Dennis Ritchie) that solves a number of security problems. For example, the author of a game program can make the program set-uid to the owner, so that it can update a score file that is otherwise protected form other users' access.

- But the set-uid concept is potentially dangerous. `passwd` has to be correct; if it were not, it could destroy system information under root's auspices. If, for example, it had the permissions:

  ```
  -rwsrwxrwx
  ```

  it could be overwritten by any user, who could therefore replace the file with a program that does anything.

- The set-uid bit is powerful, but used primarily for a few system programs such as `passwd`. A more ordinary file:

  ```
  ~$ ls -l /usr/bin/who
  -rwxr-xr-x 1 root root 14364 Jul 23 2001 /usr/bin/who
  ~$
  ```

  is executable by everybody, and writable by root and the group root. When you type `who` to the shell, the shell "finds" the file `who`, and checks if it has execute permission. If it does, the shell calls the kernel to run it.

- Note that a program is just a file with execute permission.

- Directory permissions operate a little differently, but the basic idea is the same:

  ```
  ~/somedir$ ls -ld .
  drwxrwx--x 46 you you 4096 Jan 31 11:48 .
  ~/somedir$
  ```

  - The leading `d` signifies that `.` is indeed a directory.
  - An `r` means that you can read the directory, with the `ls` command, for example.
  - A `w` means that you can create and delete files in this directory, because that requires modifying and therefore writing the directory file. (There are system calls that create and remove files, and only through them is it possible to change the contents of a directory—the `w` tells you who can use these system calls.)
  - Permission to remove a file is independent of the file itself. If you have write permission in a directory, you may remove files there, even files that are protected against writing.
  - The `x` field in the permissions on a directory does not mean execution; it means "search." The mode `--x` for other users means that users may access any file that they know about in that directory, but may not run `ls` on it or read it (with `od`) to see what files are there.

- The `chmod` command changes permissions on files (the permission descriptions can be provided by symbolic descriptions, or by octal numbers). Only the owner of the file (and the su) may change permissions on a file.

- If a directory is writable, people can remove files in it regardless of the permissions on the files themselves. Remove write permission from directory to make sure no one can delete files from it.

Inodes

- A file has several components: a name, contents, adm info, etc. The adm info is store in the inode (originally, "i-node"), along with essential system data such as how long it is, where on the disc the contents of the file are stored, and so on.

- There are three times in the inode:
  - time when the contents of the file were last modified
  - time when the file was last used
  - time the inode itself was last changes (e.g., to reset permissions)

- The system's internal name for a file is its i-number: the number of the inode holding the file's information. `ls -i` reports the i-numbers in decimal.

- The `ln` command makes a link to an existing file, with the syntax: `ln old-file new-file`. The purpose of a link is to give two names to the same file, often so that it can appear in two different directories. Two links to a file point to the same inode, and hence have the same i-number:

  ```
  $ ls -li junk
  1531917 -rw-rw-r-- 1 you you 8 Mar 12 08:30 junk
  $ ln junk trash
  $ ls -li
  1531917 -rw-rw-r-- 2 you you 8 Mar 12 08:30 junk
  1531917 -rw-rw-r-- 2 you you 8 Mar 12 08:30 trash
  ```

  Each link is equally important, and there is no difference between the first link, and subsequent ones (the "first link" is junk!).

- On the other hand:

  ```
  $ ls -li
  1531917 -rw-rw-r-- 1 you you 8 Mar 12 08:30 junk
  $ cp junk trash
  $ ls -li
  1531917 -rw-rw-r-- 1 you you 8 Mar 12 08:30 junk
  1531916 -rw-rw-r-- 1 you you 8 Mar 12 08:36 trash
  ```

Devices

- One of the nicest ideas in the UNIX system is the way it deals with peripherals—discs, tape drives, printers, terminals, etc. Rather than having special routines to, for example, read magnetic tape, there is a file called `/dev/mt0`. Inside the kernel, references to that file are converted into hardware commands to access the tape, so if a program reads `/dev/mt0`, the contents of a tape mounted on the drive are returned.
- For example,

  ```
  $ cp /dev/mt0 junk
  ```

  copies the contents of the tape to a file called `junk`. `cp` has no idea there is anything special about `/dev/mt0`; it is just a file—a sequence of bytes.


```
69445 crw-rw---- 1 root uucp  4, 66 Aug 30  2001 ttyS2
70258 lrwxrwxrwx 1 root root      8 Nov 15 06:36 cdrom -> /dev/hdc
65683 brw------- 1 root disk 22,  0 Aug 30  2001 hdc
65649 brw-rw---- 1 root disk  3,  0 Aug 30  2001 hda
66109 crw-r----- 1 root kmem  1,  2 Aug 30  2001 kmem
65242 crw-r----- 1 root kmem  1,  1 Aug 30  2001 mem
69064 crw--w---- 1 root root  4,  0 Mar 12 07:17 tty0
69065 crw------- 1 root root  4,  1 Mar 12 07:18 tty1
```

For a device file, the inode contains the internal name for the device, which consists of:

- Its type: character (c) or block (b). For example, disks and tapes are block devices, and terminals and modems are character devices.
- A pair of numbers, called the major and minor device numbers. The major number encodes the type of device, while the minor number distinguishes different instances of the device.
- All files accessible in a UNIX system are arranged in one big tree, the file hierarchy, rooted at /. These files can be spread out over several devices. The `mount` command serves to attach the file system found on some device to the big file tree. Conversely, the `umount` command will detach it again.
- For example:

  ```
  mount -t iso9660 /dev/cdrom /mnt/cdrom
  ```


Public-Key Cryptography

- The problems of key distribution are solved by public key cryptography, the concept of which was introduced by Whitfield Diffie and Martin Hellman in 1975. (There is now evidence that the British Secret Service invented it a few years before Diffie and Hellman, but kept it a military secret—and did nothing with it. [J H Ellis: The Possibility of Secure Non-Secret Digital Encryption, CESG Report, January 1970])
- Public key cryptography is an asymmetric scheme that uses a pair of keys for encryption: a public key, which encrypts data, and a corresponding private, or secret key for decryption. You publish your public key to the world while keeping your private key secret. Anyone with a copy of your public key can then encrypt information that only you can read. Even people you have never met.
- It is computationally infeasible to deduce the private key from the public key. Anyone who has a public key can encrypt information but cannot decrypt it. Only the person who has the corresponding private key can decrypt the information.

- The primary benefit of public key cryptography is that it allows people who have no preexisting security arrangement to exchange messages securely. The need for sender and receiver to share secret keys via some secure channel is eliminated; all communications involve only public keys, and no private key is ever transmitted or shared.
- Some examples of public-key cryptosystems are:

  –Elgamal (Taher Elgamal)
  –RSA (Ron Rivest, Adi Shamir, Leonard Adleman)
  –Diffie-Hellman
  –DSA, the Digital Signature Algorithm (David Kravitz).
- Because conventional cryptography was once the only available means for relaying secret information, the expense of secure channels and key distribution relegated its use only to those who could afford it, such as governments and large banks. Public key encryption is the technological revolution that provides strong cryptography to the adult masses.

RSA Cryptosystem

The RSA cryptosystem is named after Rivest, Shamir, and Adleman.

Once Again: Public-Key Encryption Scheme

Alice can set up a mechanism whereby she can receive and decode encoded messages from an arbitrary person.

This message can be transmitted over a public channel because the system ensures that nobody else can decode the message.

Alice advertises an encoding function E, which has the property that anyone can efficiently compute $E(M)$ for a message M, but no one but Alice may efficiently compute M form $E(M)$.

In fact, Alice has a decoding function D such that, for all M, $D(E(M)) = M$.

In the RSA scheme, Alice constructs functions E and D as follows:

- She chooses two distinct (odd) primes p, q, and computes $n = p \cdot q$.
- Alice keeps p, q secret, while n is given to the public.
- Alice also chooses an element $k \in \mathbb{Z}^*_{\phi(n)}$ with $k > 1$, and advertises k along with n. (Note that $\phi(n) = \phi(pq) = (p-1)(q-1)$, so it is easy to compute.)
- The encoding function is $E(M) = M^k \pmod{n}$, assuming that messages correspond to the elements of $\mathbb{Z}_n$ (if messages are bigger, then divide them into portions, each portion in $\mathbb{Z}_n$, and encode each portion separately).
- Alice can easily compute $l = k^{-1}$, that is, the multiplicative inverse of k in the group $\mathbb{Z}^*_{\phi(n)}$, so that her decoding function is $D(C) = C^l \pmod{n}$.
- Note that if $C = E(M)$, then $D(C) = M^{kl} = M \pmod{n}$ since $kl = 1 \pmod{\phi(n)}$.

PGP

http://www.pgpI.org/

PGPi is the international variant of PGP (Pretty Good Privacy), a public key encryption program originally written by Phil Zimmermann in 1991. Later PGP versions have been developed and distributed by MIT, ViaCrypt, PGP Inc., and now Network Associates Inc. (NAI). PGP is the de-facto standard for email encryption today, with millions of users worldwide.

Can PGP be cracked? Yes. Any PGP version can be cracked, provided that the attacker has enough time and resources (= money) for the job. However, a typical 1024-bit PGP message would take about 300,000,000,000 MIPS year to crack, so the ordinary citizen is relatively safe off, at least for the next few decades.

(MIPS Technologies, Inc. designs high-performance, low-power, 32- and 64-bit RISC (reduced instruction-set computing) processor architectures and cores for embedded systems.)

How PGP works

PGP combines some of the best features of both conventional and public key cryptography. PGP is a hybrid cryptosystem. When a user encrypts plaintext with PGP, PGP first compresses the plaintext. Data compression saves modem transmission time and disk space and, more importantly, strengthens cryptographic security. Most cryptanalysis techniques exploit patterns found in the plaintext to crack the cipher. Compression reduces these patterns in the plaintext, thereby greatly enhancing resistance to cryptanalysis. (Files that are too short to compress or which don't compress well aren't compressed.)

PGP then creates a session key, which is a one-time-only secret key. This key is a random number generated from the random movements of your mouse and the keystrokes you type. This session key works with a very secure, fast conventional encryption algorithm to encrypt the plaintext; the result is ciphertext. Once the data is encrypted, the session key is then encrypted to the recipient's public key. This public key-encrypted session key is transmitted along with the ciphertext to the recipient.

Decryption works in the reverse. The recipient's copy of PGP uses his or her private key to recover the temporary session key, which PGP then uses to decrypt the conventionally-encrypted ciphertext.

The combination of the two encryption methods combines the convenience of public key encryption with the speed of conventional encryption. Conventional encryption is about 1, 000 times faster than public key encryption. Public key encryption in turn provides a solution to key distribution and data transmission issues. Used together, performance and key distribution are improved without any sacrifice in security.


```
~$ pgp -kg
Pretty Good Privacy(tm) Version 6.5.8
(c) 1999 Network Associates Inc.
Uses the RSAREF(tm) Toolkit, which is copyright RSA
Data Security, Inc.
Export of this software may be restricted by the
U.S. government.

Choose the public-key algorithm to use with your new key
1) DSS/DH (a.k.a. DSA/ElGamal) (default)
2) RSA
Choose 1 or 2: 1
Choose the type of key you want to generate
1) Generate a new signing key (default)
2) Generate an encryption key for an existing signing key
Choose 1 or 2: 1
Pick your DSS ‘‘master key’’ size:
1)  1024 bits- Maximum size (Recommended)
Choose 1 or enter desired number of bits: 1
Generating a 1024-bit DSS key.

You need a user ID for your public key.  The desired form
for this user ID is your name, followed by your E-mail
address enclosed in <angle brackets>, if you have an E-mail
address. For example:  John Q. Smith <jqsmith@nai.com>
Enter a user ID for your public key:
```



```
Enter the validity period of your signing key in days
from 0 - 10950 0 is forever (the default is 0):

You need a pass phrase to protect your DSS secret key.
Your pass phrase can be any sentence or phrase and may have
many words, spaces, punctuation, or any other printable
characters.

Enter pass phrase:
Enter same pass phrase again:

PGP will generate a signing key. Do you also require an
encryption key? (Y/n) Y
Pick your DH key size:
1)  1024 bits- High commercial grade, secure for many years
2)  2048 bits- "Military" grade, secure for future
3)  3072 bits- Archival grade, slow, highest security
Choose 1, 2, 3, or enter desired number of bits: 1

Enter the validity period of your encryption key in days
from 0 - 10950 0 is forever (the default is 0):
```



```
Note that key generation is a lengthy process.

PGP needs to generate some random data. This is done by
measuring the time intervals between your keystrokes.

Please enter some random text on your keyboard until the
indicator reaches 100%.
Press ^D to cancel
100% of required data
Enough, thank you.
.******* ......................******* .
Make this the default signing key? (Y/n) Y
.........................................
...............................................********
*******
Key generation completed.
```

Chp 15: Network Structures

Background:

* Topology
* Network Types
* Communication
* Communication Protocol
* Robustness
* Design Strategies

Motivation:

* Resource sharing:

  -- sharing and printing files at remote sites
  -- processing information in a distributed database
  -- using remote specialized hardware devices

* Computation speedup: load sharing

* Reliability: detect and recover from site failure, function transfer, reintegrate failed site

* Communication: message passing

Network-Operating Systems:

* Users are aware of multiplicity of machines. Access to resources of various machines is done explicitly by:

  -- Remote logging into the appropriate remote machine.

  -- Transferring data from remote machines to local machines, via the File Transfer Protocol (FTP) mechanism.

Distributed-Operating Systems:

* Users not aware of multiplicity of machines. Access to remote resources similar to access to local resources.

* Data Migration: transfer data by transferring entire file, or transferring only those portions of the file necessary for the immediate task.

* Computation Migration: transfer the computation, rather than the data, across the system.

Distributed-Operating Systems (Cont.):

* Process Migration: execute an entire process, or parts of it, at different sites.

  -- Load balancing: distribute processes across network to even the workload.

  -- Computation speedup: subprocesses can run concurrently on different sites.

  -- Hardware preference: process execution may require specialized processor.

  -- Software preference: required software may be available at only a particular site.

  -- Data access: run process remotely, rather than transfer all data locally.

Topology:

* Sites in the system can be physically connected in a variety of ways; they are compared with respect to the following criteria:

  -- Basic cost: How expensive is it to link the various sites in the system?

  -- Communication cost: How long does it take to send a message from site A to site B?

  -- Reliability: If a link or a site in the system fails, can the remaining sites still communicate with each other?

* The various topologies are depicted as graphs whose nodes correspond to sites. An edge from node A to node B corresponds to a direct connection between the two sites.

* The following six items depict various network topologies.

Network Types:

* Local-Area Network (LAN): designed to cover small geographical area.

  -- Multiaccess bus, ring, or star network.

  -- Speed H 10 megabits/ second, or higher.

  -- Broadcast is fast and cheap.

  -- Nodes:

    o usually workstations and/or personal computers

    o a few (usually one or two) mainframes.

Network Types (Cont.):

* Wide-Area Network (WAN) links geographically separated sites.

  -- Point-to-point connections over long- haul lines (often leased from a phone company).

  -- Speed about 100 kilobits/second.

  -- Broadcast usually requires multiple messages.

  -- Nodes:

    o usually a high percentage of mainframes

Communication:

The design of a communication network must address four basi issues:

* Naming and name resolution : How do two processes locate each other to communicate?

* Routing strategies . How are messages sent through the network?

* Connection strategies . How do two processes send a sequence of messages?

* Contention: The network is a shared resource, so how do we resolve conflicting demands for its use?

Naming and Name Resolution:

* Name systems in the network

* Address messages with the process- id.

* Identify processes on remote systems by

  <host-name, identifier> pair.

* Domain name service (DNS) specifies the naming structure of the hosts, as well as name to address resolution (Internet).

Routing Strategies:

* Fixed routing: A path from A to B is specified in advance; path changes only if a hardware failure disables it.

  -- Since the shortest path is usually chosen, communication costs are minimized.

  -- Fixed routing cannot adapt to load changes.

  -- Ensures that messages will be delivered in the order in which they were sent.

* Virtual circuit: A path from A to B is fixed for the duration of one session. Different sessions involving messages from A to B may have different paths.

  -- Partial remedy to adapting to load changes.

  -- Ensures that messages will be delivered in the order in which they were sent.

Routing Strategies (Cont.):

* Dynamic routing: The path used to send a message form site A to site B is chosen only when a message is sent.

  -- Usually a site sends a message to another site on the link least used at that particular time.

  -- Adapts to load changes by avoiding routing messages on heavily used path.

  -- Messages may arrive out of order. This problem can be remedied by appending a sequence number to each message.

Connection Strategies:

* Circuit switching: A permanent physical link is established for the duration of the communication (i. e., telephone system).

* Message switching: A temporary link is established for the duration of one message transfer (i. e., post- office mailing system).

* Packet switching: Messages of variable length are divided into fixed-length packets which are sent to the destination. Each packet may take a different path through the network. The packets must be reassembled into messages as they arrive.

* Circuit switching requires setup time, but incurs less overhead for shipping each message, and may waste network bandwidth. Message and packet switching require less setup time, but incur more overhead per message.

Contention:

Several sites may want to transmit information over a link simultaneously. Techniques to avoid repeated collisions include:

* CSMA/CD: Carrier sense with multiple access (CSMA); collision detection (CD)

  -- A site determines whether another message is currently being transmitted over that link. If two or more sites begin transmitting at exactly the same time, then they will register a CD and will stop transmitting.

  -- When the system is very busy, many collisions may occur, and thus performance may be degraded.

* SCMA/CD: is used successfully in the Ethernet system, the most common network system.

Contention (Cont.):

* Token passing: A unique message type, known as a token, continuously circulates in the system (usually a ring structure). A site that wants to transmit information must wait until the token arrives. When the site completes its round of message passing, it retransmits the token. A token- passing scheme is used by the IBM and Apollo systems.

* Message slots: A number of fixed-length message slots continuously circulate in the system (usually a ring structure). Since a slot can contain only fixed-sized messages, a single logical message may have to be broken down into a number of smaller packets, each of which is sent in a separate slot. This scheme has been adopted in the experimental Cambridge Digital Communication Ring

Communication Protocol:

The communication network is partitioned into the following multiple layers:

* Physical layer: handles the mechanical and electrical details of the physical transmission of a bit stream.

* Data-link layer: handles the frames, or fixed-length parts of packets, including any error detection and recovery that occurred in the physical layer.

* Network layer: provides connections and routes packets in the communication network, including handling the address of outgoing packets, decoding the address of incoming packets, and maintaining routing information for proper response to changing load levels.

Communication Protocol (Cont.):

* Transport layer: responsible for low-level network access and for message transfer between clients, including partitioning messages into packets, maintaining packet order, controlling flow, and generating physical addresses.

* Session layer: implements sessions, or process-to-process communications protocols.

* Presentation layer: resolves the differences in formats among the various sites in the network, including character conversions, and half duplex/full duplex (echoing).

* Application layer: interacts directly with the users deals with file transfer, remote-login protocols and electronic mail, as well as schemas for distributed databases.

Robustness:

* Failure detection

* Reconfiguration

Failure Detection:

* Detecting hardware failure is difficult.

* To detect a link failure, a handshaking protocol can be used.

* Assume Site A and Site B have established a link. At fixed intervals, each site will exchange an I-am-up message indicating that they are up and running.

* If Site A does not receive a message within the fixed interval, it assumes either (a) the other site is not up or (b) the message was lost.

* Site A can now send an Are-you-up? message to Site B.

* If Site A does not receive a reply, it can repeat the message or try an alternate route to Site B.

Failure Detection (cont):

* If Site A does not ultimately receive a reply from Site B, it concludes some type of failure has occurred.

* Types of failures:

  -- Site B is down

  -- The direct link between A and B is down

  -- The alternate link from A to B is down

  -- The message has been lost

* However, Site A cannot determine exactly why the failure has occurred.

Reconfiguration:

* When Site A determines a failure has occurred, it must reconfigure the system:

  1. If the link from A to B has failed, this must be broadcast to every site in the system.

  2. If a site has failed, every other site must also be notified indicating that the services offered by the failed site are no longer available.

* When the link or the site becomes available again, this information must again be broadcast to all other sites.

Design Issues:

* Transparency: the distributed system should appear as a conventional, centralized system to the user.

* Fault tolerance: the distributed system should continue to function in the face of failure.

* Scalability: as demands increase, the system should easily accept the addition of new resources to accommodate the increased demand.

* Clusters: a collection of semi- autonomous machines that acts as a single system.

Networking Example:

* The transmission of a network packet between hosts on an Ethernet network.

* Every host has a unique IP address and a corresponding Ethernet (MAC) address.

* Communication requires both addresses. Domain

* Name Service (DNS) can be used to acquire IP addresses.

* Address Resolution Protocol (ARP) is used to map MAC addresses to IP addresses.

* If the hosts are on the same network, ARP can be used. If the hosts are on different networks, the sending host will send the packet to a router which routes the packet to the destination network.