VRIJE UNIVERSITEIT AMSTERDAM

VU Research Portal

A Survey of Operating Systems

Tanenbaum, A.S.

published in
Informatie
1976

document version
Publisher's PDF, also known as Version of record

Link to publication in VU Research Portal

citation for published version (APA)
Tanenbaum, A. S. (1976). A Survey of Operating Systems. *Informatie*, *18*(Dec.), 689-698.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

E-mail address:
vuresearchportal.ub@vu.nl

Download date: 14. Sep. 2021

A SURVEY OF OPERATING SYSTEMS

by Andrew S. Tanenbaum

Keywords: operating system, multiprogramming
Kodes: 000.015.580
H

Like the proverbial 3 blind men exploring the elephant, different people viewing the same operating system can come to radically different conclusions as to what the 'true nature' of the beast is. Some people define 'operating system' as the totality of software delivered by their computer manufacturer, including the COBOL compiler, payroll program and tape sort routine. Other, more discriminating, people view an operating system as a collection of programs whose function is to manage the computer's resources (e.g. CPU, primary memory, peripherals, disks, etc.) in order to maximize performance. Still other people regard the operating system as a partial interpreter whose task is to hide the unpleasant characteristics of the hardware from the programmer. In this view, the operating system's function is to provide the user with an 'extended' or 'virtual' machine, one which is more convenient to use than the original.
Prepared with the knowledge that no two people seem to agree on what an operating system (henceforth OS) is, it should come as no shock to the reader that there is no generally accepted breakdown of the subject into subdisciplines. In this article we will examine 6 topics that this author considers the essence of the subject: functional capabilities, processes, structure, resource management, protection, and evaluation. The subjects treated here are all areas of current research.

1 OPERATING SYSTEM CLASSIFICATION AND CAPABILITIES

1.1 Taxonomy of Operating Systems

The simplest OS type is the monoprogramming system. In this system the operator manually enters a job. When the job is finished, the operator manually enters the next one. Many stand alone minicomputers use a monoprogrammed OS.
To eliminate wasted time between jobs, the batch system was invented. In this system a collection of jobs are strung together on some input medium (magnetic tape, card decks, or a disk file). The batch system simply runs them in sequence without operator intervention. Each job waits until its predecessor has terminated before starting.
Most programs must occasionally pause to wait for i/o to complete. In a batch system the CPU is idle during i/o wait time. In a commercial data processing environment, i/o wait time is often 90% or more. To increase CPU utilization, multiprogramming systems were invented. In a multiprogramming system, several programs are kept in primary memory simultaneously. When one program is blocked, waiting for i/o to complete, the CPU can be given to another program. If the primary memory is sufficiently large, there will nearly always be an unblocked program around, and CPU utilization can be raised to close to 100%.
Multiprogramming, although a valuable technique, brings with it a host of complications. For example if there is insufficient primary memory, sometimes all programs will be blocked, and CPU time will be wasted. If primary memory is added to a system already running at 100% CPU utilizations, the added memory is unnecessary, hence wasted. Achieving a balance of CPU, memory and other resources in the face of a dynamically varying load is not easy. Nevertheless, most OS's nowadays, except for some mini's, are multiprogrammed, and we will concentrate our study on these.
If some of the programs running under a multiprogramming system can interact with a human being at a remote terminal, the system is called a timesharing

Met veel genoegen plaatsen wij het artikel van dr. A. S. Tanenbaum.
Het artikel is geschreven in het Engels, de moedertaal van de auteur. Wij hebben overwogen het te laten vertalen. Hiervan is afgezièn, niet alleen vanwege de hoge kosten, maar ook omdat het erg moeilijk is voor de onderhavige materie goede Nederlandse terminologie te vinden.
Zowel door de vreemde taal als door het onderwerp zal het voor vele lezers geen eenvoudige kost zijn. Als redaktie willen wij daarom een extra aanbeveling doen om het artikel te bestuderen.
Het geeft een zeer goed overzicht over de huidige stand van zaken op het gebied van de systeem programmatuur. Maar het is niet alleen voor systeemprogrammeurs van belang. Veel algemene aspekten van programmering komen aan de orde. Voorts zullen veel gebruikers hun voordeel kunnen doen met lezing. Met name de paragrafen 5. 'Protection' en 6 'Performance monitoring and evaluation' verdienen een brede belangstelling.
Namens de Redactie
J. A. van der Pool

system. Timesharing introduces the additional complication of the need to swap programs waiting for human input out of primary memory.
Some multiprogramming systems have remote terminals consisting of laboratory equipment or industrial process control devices instead of teletype or CRT terminals. These devices send data to the connected program for storage or processing. If there is no buffering at the equipment end, the computer must respond to a request for service before the data is lost. A multiprogramming system with terminals operating under strict time constraints is called a real time system. Often real time systems have hundreds of terminals, each of which requires service within microseconds of the time a request is posted.

Yet another species is the network or distributed system. These handle substantial numbers of CPU's (> 10). The distinction between a network and a distributed computer is one of computational intimacy. If the CPU's are physically far apart, and working on different problems, it is regarded as a network (e.g. FARBER's ring network [1]); if they are physically close together, working on the same problem it is regarded as a distributed system (e.g. the CMU Multimini, WULF [2]). These types of systems are new, but rapidly increasing in importance.
Lastly there are dedicated systems, which do not attempt to support general programming, but are specific to one application. Data base management, banking, and computer aided instruction are typical application areas.

1.2 Characteristics of Multiprogrammed Operating Systems

First, they are huge. OS/360, for the IBM 360, consists of 30,000 pages of assembly code. It took 5000 man-years to construct it. The MULTICS system, for the Honeywell 6180 consists of 4000 pages of PL/I.
Second, they involve parallelism, both in hardware and software. Many simultaneous activities must be coordinated.
Third, they are nondeterminate. If two travel agents simultaneously try to reserve the last seat on an airplane, one will win and one will lose (hopefully). Operating system behaviour is not reproducible, unless timing considerations down to the nanosecond level are taken into account, and even then, the problem of arbitration of exactly simultaneous events must be considered.
Fourth, multiprogramming systems must provide facilities for permanent long term storage of information (file system). If the owner of a 100,000 volume medical library available from remote terminals had to retype the entire library every day, he would probably not be wildly enthusiastic about his operating system.
Fifth, multiprogramming systems must provide for the controlled sharing of information and recources. There is a crucial distinction between technological sharing and intrinsic sharing. Technological sharing is done for economic reasons only. When CPU's become cheap enough to give every program its own (1980-1985), there will be no need to share CPU's among programs. On the other hand, the sharing of the passenger lists in an airline reservation system among the various programs (one per travel agent) is essential. No improvement in technology will eliminate this sharing (except maybe giving each passenger his own airplane).
In a large system, with many independent user groups, adequate facilities for, and controls on, sharing is of great importance. The MULTICS project began as a prototype of a national or regional computer utility, which would provide 24-hour a day continuous service on a metered basis, much like telephone service or electricity distribution. This would eliminate the need for potential computer users to start out by buying or renting an entire computer, with its attendant space, staff and maintenance headaches. In a computer utilily, providing adequate facilities for, and controls on, sharing is probably the dominant design consideration.
Sixth, multiprogramming systems must automatically manage the allocation and use of the various hardware facilities, without expecting much help on the part of the users.

1.3 Services Provided by Typical Operating System

One view of an operating system is that it provides its user with a virtual machine, one that is less awkward than the bare hardware. This is accomplished by implementing a class of instructions and features not present in most third generation hardware. Special instructions are usually invoked via a 'supervisor call' or 'emulator trap' instruction.
One major feature provided by most large scale OS's is virtual memory. Programs can address a large address space – often larger than the machine's physical memory – with the mapping of virtual to real addresses handled by hardware, except when the addressed piece of program is not in primary memory. In this case the OS intervenes to fetch it.

Virtual instructions to read and write files are common. Files may be read or written sequentially or randomly. OS/360 provides a multitude of access methods and options, including automatic buffering, deblocking, queueing, and searching. An OS must also provide for creating, destroying, protecting and cataloging files, as well as many other functions relating to file directory management. Users have come to expect all file operations to be device independent, so that a program written to accept card input will also run using tape input.
All operating systems provide a Job Control Language (JCL) which allows the user to communicate with the OS. Typical JCL's look like assembly language, with one instruction and its parameters per line. Instructions are things like call a compiler, execute a program, print a file, request a resource (primary or secondary memory space, tape drive, plotter, etc.), produce a memory printout, catalog a file etc. It is an unfortunate property of most OS's that the JCL interpreter is buried deep inside the OS. A better strategy would be to provide 'virtual instructions' for all the available facilities, thus allowing anyone who wanted to, to provide his own JCL compiler or interpreter.

2 PROCESSES

2.1 Characterization of a Process

The concept of a process is central to understanding how an OS works. A *process* is a program in execution. At every point in time a process has a certain *state,* consisting of the program, the values of all its variables, registers, program counter etc. As the process runs, its state changes; the variables take on new values, the program counter advances, and so forth. It is useful to distinguish that part of the process that does not change in time (e.g. the program itself, assuming it is reentrant, and perhaps certain tables) from the part that does change in time. The changeable part is called the *state vector.*
One way of looking at a process is to say that a process consists of 2 parts: The program (including fixed tables) and the state vector. The program is a set of rules describing how the state vector is to be changed in time. The CPU is regarded as an engine that forces the state vector through the sequence described by the program. Another viewpoint regards a process as a (past and future, history of a program's state vector. Either way, a process is an active entity. It can cause events in the outside world to occur, for example, writing Chinese on the plotter, or playing chess with another process or being. In contrast, a program is a passive entity. It just sits there and does nothing.
Note carefully that a process is a strictly sequential entity. There is no parellelism within a process.

In order to keep track of the status of each process under its control, an OS must maintain a small table for each process. This table, sometimes called a Process Control Block (PCB), contains information such as the process' status (running, ready to run, blocked), primary memory allocation (the memory contains the program and variables), files in use, i/o device status, accounting information (i.e. process history), program status word, quotas (e.g. maximum disk space allowed) etc. Whenever a process is suspended for any reason, its PCB must be updated, in order that it can be restarted in the same state that it stopped in. In fact, the contents of the PCB's in a particular system is an operational definition of what processes consists of. It should be clear now that a program is but one of many components of a process. In fact, from the OS's point of view, the function carried out by the program is irrelevant, only its resource usage and demand for services is visible.
Although sharing of reentrant code is sometimes possible, each process normally runs in its own private address space. No process can just reach into another process' address space and examine or change data, except by prior consent. Protection mechanisms use this feature heavily.

2.2 Interprocess Communication

Processes need to communicate with other processes for a variety of reasons. One of the most important is to communicate information. For example, in an OS consisting of several processes, there is likely to be a process that manages the line printer. All printing is handled by the line printer process. To have a file printed, a user process must somehow convey to the printer process the fact that it wants a file printed, and which one. Situations like this are very common. Another reason interprocess communication (IPC) is needed is to allow processes to synchronize their activities. As an example consider a producer-consumer problem with a shared buffer capable of holding only a single item. The sequence of execution must be: producer fills buffer; consumer empties buffer; producer fills buffer; consumer empties buffer, etc. In other words, the two processes must communicate with each other in order to strictly alternate their buffer accesses.

A somewhat related problem is that of mutual exclusion. In many systems certain data bases must only be accessed by one process at a time, although the order of access is not important. A typical example is a flight list in an airline reservation system; bad things could happen if two processes tried to reserve one seat simultaneously. From the viewpoint of the processes and the OS, it does not matter which of several interested processes gets the data base first, so long as no more than one has it at any given time.
All interprocess communication involves sharing of some address space (or object upon which an instruction operates) between the communicating processes. Sometimes this is disguised, as in the case where all processes can (indirectly) read and write from a common table within the OS. Three of the many interprocess communication mechanisms in use are described below.

2.2.1 *Semaphores*

Excluding very low level communication primitives, like test-and-set and lock-or-skip type instructions, DIJKSTRA's semaphore system [3] is about as simple as IPC can be. To communicate, processes arrange for the existence of a protected, non-negative integer variable (semaphore) shared between (among) themselves. Two instructions on the semaphore are provided by the OS: *down* and *up*. If a process does a *down* on a positive semaphore, its value is simply decremented by 1. If a process does a *down* on a zero valued semaphore, it is blocked and the *down* is not completed. If a process does an *up* on a semaphore, its value is increased by 1. However, if another process was blocked on the semaphore, it can now complete its *down* instruction and proceed. If several processes were blocked on the same semaphore, one of them is chosen by magic to continue. The others remain blocked.

Semaphores are simple to implement. The OS need only maintain for each semaphore a list of processes blocked on it. If a field is provided in the PCB, a semaphore can be represented by two items of information: a value, and a pointer to the first PCB linked on the blocked list for that semaphore. Subsequent processes blocking on the same semaphore are linked together via the PCB field.

Semaphores also have a severe disadvantage: processes can use them to synchronize control, but not to pass information. The example of a process want-

ing to print a file given above would be impossible to implement using semaphores alone; a second IPC mechanism (e.g. address space sharing would be needed in addition).

2.2.2 *Messages*
With this IPC mechanism, every process has a mailbox associated with it. Two primitive instructions are provided: *get* and *send*. When a process executes a *send* instruction it specifies a destination process and message buffer within its own address space. The OS then copies the message to the receiver's mailbox, and the sending process is then free to continue. The mailboxes are maintained within the OS itself, so no process can examine its own or any other process' mailbox. To get a message, a process executes a *get* instruction. If there is a message waiting, the OS copies into the receiver's address space, along with the sender's identity, provided by the OS, so as to be unforgeable. If no message is waiting, the receiver blocks until such time as a message arrives.

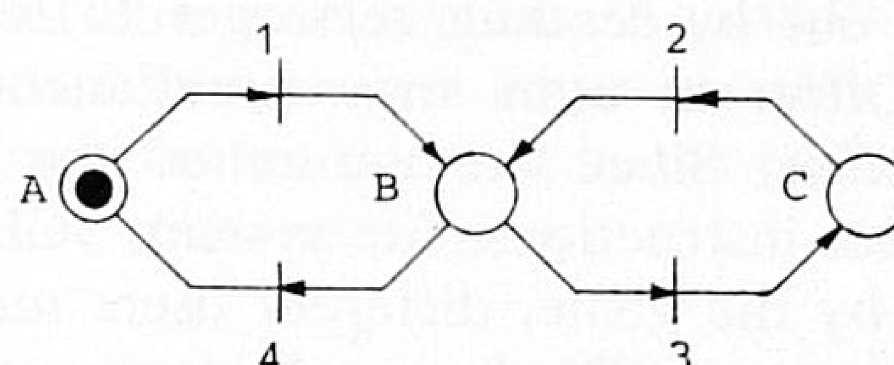
One can regard each mailbox as a private semaphore associated with each process. Doing a *get* is analogous to doing a *down;* sending a process a message to a process is analogous to doing an *up* on its semaphore. The number of messages in the mailbox is analogous to the semaphore value. Many variations of this popular scheme exist, such as the ability for two processes to create a private tube (event channel) between them for sending and getting messages.
Messages have the obvious advantage of combining communication of information with synchronization. They have the disadvantage of requiring a space consuming message table within the OS. If the table threatens to fill up, processes attempting to send messages can be retroactively suspended (at the *send* instruction) until there is more room.

2.2.3 *Monitors*
A *monitor* (BRINCH HANSEN [4]) is a data structure consisting of local data, and 1 or more procedures that processes can execute. Monitors have the property that only 1 process at a time may be invoking a given procedure. There is a special data type called a *condition*. A process may execute a *wait* instruction on a condition variable, which causes it to block until another process executes a *signal* instruction on the same variable. For example, here is a monitor to allocate and release a single dedicated resource.

```
monitor resource =
begin bool busy: = false; condition flag;
        proc acquire = void:
        (if busy then wait flag fi; busy; = true);
        proc release = void:
        (busy: = false; signal flag)
end
```

Monitors may contain procedures and initialized local data. To acquire the resource, a process executes acquire. If the resource is in use, the flag busy will be true and the process will execute the *wait* flag instruction, blocking itself. When some other process calls release, it will *signal* flag, freeing the blocked process to continue.

2.3 **Petri Nets**
Multiprocess systems can be modeled by Petri Nets. A *Petri Net* (PETRI [5]) is a directed graph containing two kinds of nodes, places (circles) and transitions (bars). There are a finite number of tokens distributed among the places. The Petri Net moves from state to state in time. A transition is said to be enabled if every input place has at least one token. At every step in time, exactly 1 transition fires, removing exactly 1 token from each input place, and depositing exactly 1 token in each output place. Tokens are not conserved. The following Petri Net models mutual exclusion.



A token at A indicates process 1 is running; a token at C indicates that process 2 is running. A token at B indicates that the OS is about to choose a process to run, by the nondeterministic firing of either transition 3 or 4 but not both. The firing sequence 141414141414 shows process 1 hogging the CPU, whereas 132413241324 is a round robin scheduler.

3 OPERATING SYSTEM STRUCTURE

3.1 The Big Mess
This is the traditional form used by nearly all commercial systems. It consists of having hundreds of procedures randomly calling one another. The result is systems that crash 3 times a day, leaving the systems programmers to wonder how they work at all.

3.2 Hierarchial Systems
This system design was made popular by DIJKSTRA's THE system [6]. It is essentially a bottom-up design in which a virtual machine is built up in layers, each layer adding new features to the virtual machine used to construct the next higher layer.
At the bottom level is the bare machine, which is nondeterministic due to interrupts. The flow of control is unpredictable because any instruction may be followed by the first instruction of some interrupt routine. The next level has processes that communicate via semaphores. Each process is strictly sequential and deterministic. A hardware interrupt merely causes the CPU to switch from whichever process it was running to the interrupt process. Subsequent layers add features such as virtual memory, virtual terminals, virtual i/o instructions etc.

3.3 Extendable Systems
Another design technique, made popular by the RC4000 system [7] of BRINCH HANSEN is that of providing a common nucleus for different OS's. In the case of the RC4000 OS, the common nucleus is based on a variation of the message handling IPC mechanism. The nucleus provides certain basic facilities to higher level processes, which can be used to build OS's.

As an example of an extendable system that is also hierarchically structured, we note a time sharing system for the PDP-11/45, TSS-11 (TANENBAUM [8]). In this system, the facilities available are gradually increased over a number of levels until the 'genie' level is reached. A process running at the genie level has virtual memory, IPC via messages, virtual i/o (by sending messages to the terminal handling processes, by sending file descriptors to printer, punch, etc. processes) etc.
Associated with each user process is a genie process, the latter of which runs at the genie level. The only operation a user process is capable of executing is sending a message to its genie. The genie then carries the request out by sending messages to lower level processes. Different users may simultaneously have different genies. Since the nature of the user machine's virtual instructions, file system, JCL, etc. are determined by the genie, different users may appear to be running on different and incompatible OS's simultaneously.

3.4 **Self Virtualizing Machines**

A novel and extremely powerful technique for structuring an OS follows from two observations: 1) In a multiprogramming system, each user has a subset of the 'real hardware' instructions, like register add, and a set of 'virtual' instructions, like read file. The former are carried out by the hardware or microprogram, and the latter are carried out by the OS; 2) If the OS is working properly nothing any user process does can damage any other process.

In a self virtualizing machine system (GOLDBERG [9]), the virtual machine presented to the user is an exact duplicate of the complete real machine, including all the i/o instructions, ability to change or circumvent the protection mechanisms etc. Each virtual machine runs its own complete OS. This is accomplished as follows. To use this technique one needs a machine with two modes – user mode and system mode, in which all sensitive instructions (i/o, fiddling with the protection, changing the memory, mapping etc.) are trapped in user mode. In a standard OS, when a user process executes a sensitive instruction, a hardware trap to the OS occurs, and the process is abruptly aborted with a suitably rude message such as INV INSTR AT 43B7A.
In a self virtualizing machine system the OS running on the real hardware forces all virtual machines to always run in user mode, but it keeps careful track of whether each virtual machine is in 'virtual user mode' or 'virtual system mode'. When a virtual machine executes a sensitive instruction in virtual user mode, the true OS running on the real machine (called a virtual machine monitor, or VMM) gets control. Instead of aborting the offending program, it passes control to the OS in the virtual machine causing the trap. This OS then does whatever it normally does when a user process does something wrong, such as abruptly aborting the process.
However when a virtual machine in virtual system mode executes a sensitive instruction, the VMM handles the trap completely differently. It simulates the instruction. If it was an i/o instruction, the VMM carries out the i/o (using virtual card readers, virtual disks, etc.) bit for bit the way the real hardware does. When the sensitive instruction has been completed, the virtual machine can continue.
Note that there is no way for a virtual machine to damage its neighbors. It can erase the non-resident portion of its own operating system and destroy all the files on its virtual disk, but it cannot affect any other virtual machine's virtual disk.
This design has effected a complete partitioning between the multiprogramming and user command interpretation functions of an OS. The VMM handles all the multiprogramming, and the simulation of the sensitive instructions (of which there are usually fewer than 10). The OS's in the virtual machines can be simple monoprogrammed systems that only need worry about interpreting and executing user service requests. By splitting an OS into two very distinct parts, it becomes much more manageable. This type of system is available on the IBM 370's under the name VM/370. Certain defects in the 370's design made writing the VMM more difficult than it would otherwise have been (the 360's designers back in 1962 never envisioned such a system), but it is easy to design a machine to make VMM's easy to write. Future computers will probably use this technique heavily.

3.5 **No Operating System**

In MULTICS, there is no distinction made between the system processes and the user processes. Each user process has as part of its address space (albeit protected) routines to schedule the CPU, handle page faults, allocate disk space, etc. Thus each user process is self supporting, never needing to invoke 'the system' for anything. Of course this does not address the issue of how the protected routines within each process are structured (hierarchically, as it turns out), but the idea is intriguing.

4 **RESOURCE MANAGEMENT**

4.1 **CPU Management**

One resource that all OS's must manage is the CPU. Usually there are several processes competing for the CPU. The OS must have some policy about deciding in what order to run the processes. In a multiprogramming system with no time sharing users, a common scheduling algorithm is Run To Completion. Each job is given an external priority (e.g. professors get better service than students, rich people get better service than poor people, everybody gets so many minutes of 'big rush' time to use when he needs it, etc.). Whenever the CPU is idle, the highest priority program is given the processor. It continues running until it finishes or blocks for i/o. Then the next highest priority program is run. A variation of this scheduling algorithm has the priorities dynamically adjusted by the OS. For example, highly i/o bound programs could be given high CPU priority to make sure their i/o keeps going continuously.
Another common algorithm is Round Robin. All the programs bidding for service are given short quanta (e.g. 100ms) in succession. Four programs would run in the sequence 123412341234. In Run To Comple-

tion, a greedy process can get the CPU and keep it for 2 hours; Round Robin prevents this.
Another well known scheduling algorithm is the Feedback Queue system. The schedular maintains N queues, the highest getting the best service. When a process must be chosen, the first process on the highest populated queue is selected. If a process terminates by blocking (for input) it is removed from the queue system. When it later unblocks, it is reinjected on the highest queue. If a process terminates by using up its quantum, it moves down 1 queue. The result of this algorithm is to keep the highly interactive processes on the top queue, and the more compute bound processes on successively lower queues.
Yet another scheduling algorithm attempts to give process i N seconds of CPU time per minute, no more and no less. Such an algorithm tends to give the user uniform response, independent of system loading.
Butler LAMPSON has suggested that next to each terminal there be a large red button. If a user is unhappy with the service he is getting, he may opt for pushing aforesaid button, in which case one of 2 things happens: either his service gets better or he is automatically logged off. The CPU time acquired by logging people off in this fashion goes into a reserve pool to be used for providing better service to the winners of the big red button show. The idea is that most people are reluctant to leave their terminal, even when the service is bad and they have other work to do.

4.2 Memory Management

OS/360 and some other systems divide primary memory into a number of partitions. In some versions the number and size of the partitions is keyed in by the console operator in the morning and remains fixed all day. In this type of system each program is loaded into some partition where it remains until finished. In other versions the number and size of the partitions is dynamically variable. Sometimes the OS can compact memory by squeezing the partitions together, in order to gather all the interpartition space into a single hole, hopefully large enough for another program. Moving programs around can be tricky, however, if i/o is in progress. Compacting outward from the middle is more efficient than 'top down' of 'bottom up'.

For time sharing systems, programs must frequently be swapped out to secondary memory. Since it is usually difficult to insure that they will be put back later at the same address, some mechanism is needed to allow programs to be reloaded anywhere. The most common method is paging. The address space is broken up into units called *pages* (typically 256-1024 words). Special hardware defines the memory locations (page frame) corresponding to each page of address space. When a program references its address space, the hardware correctly maps the reference onto the proper physical memory address, or traps to the OS if the page is in secondary memory (page faults).
It is also possible to implement a 2-dimensional address space, by having each instruction specify an address space (segment) and a location within it. This is called *segmentation*. Paging and segmentation are convenient for the user, giving him the illusion of a large address space, and other advantages. For the OS, paging and segmentation mean more work, keeping track of which pages are where, and managing all the paging traffic between primary memory and (perhaps a hierarchy of) secondary memory.
After a process has been running for a while, the pages being heavily used will naturally gravitate towards primary memory, whereas the less heavily used pages will be kept in secondary memory. The heavily used pages are known as the *working set*. An OS can attempt to determine the working set of all its processes. When it comes time to run a process, the OS can preload the working set. Alternatively the OS can just start the process anyway, and let it cause repeated page faults, bringing in the pages as needed. This is called *demand paging*.

When a page fault occurs, the OS must choose a page to remove. This is called a page replacement algorithm. Some algorithms use a local strategy, removing only the faulting process' own pages, while others use a global strategy, removing any convenient page. Popular algorithms include removing the least recently used page, or removing the earliest loaded page, regardless of its usage. A variation removes all the pages of some unlucky process, the argument being that it can't run without its complete working set, so why keep any of it around.

4.3 Management of Rotating Memories

Just as there innumerable algorithms for managing primary memory, there are innumerable algorithms for page placement in secondary memory, and traffic control between the two. Some systems try to minimize page traffic to keep overhead low; others try to maximize it, to produce a short turnaround by keeping processes flowing through primary memory at a high rate.
The simplest scheduling stategy for rotating memories is first come first served. It is also the worst. If many requests are pending when a decision has to be made, one algorithm is to choose the one that can be completed fastest. On a moving head disk this may keep the arm shuttling back and forth near the middle of the disk, giving terrible service to pages at the extremes. The elevator algorithms sweeps from one cylinder to another, changing direction only at the ends. Algorithms taking rotational position into account as well exists for both disks and drums.
Another parameter to be considered is whether pages should always be rewritten in the same place. In some systems a page is rewritten in the first empty position that comes by. This drastically reduces waiting for completion and arm movement, and drastically increases the bookkeeping required to find the page again later.
Since pages that have not been modified since being brought into primary memory (clean pages) need not be rewritten (a good copy already exists out there), there is a high premium to maintaining a high ratio of clean pages/dirty pages. Some systems use idle disk or drum time to rewrite dirty pages, even if there is no specific reason, just to maintain this ratio high.

These are called sneaky writes, and whether the overhead required to perform them is worth the gain is an open question.
Note that the previous 3 sections all deal with technological sharing, which can be expected to decrease in importance in the future.

4.4 Deadlocks

The CPU and primary memory are pre-emptable resources. If somebody more important comes along, they can simply be snatched away from their current owner. Tape drives, plotters, and certain other resources cannot be effectively pre-empted without abandoning all work currently in progress. Non-preamptable resources can lead to trouble. Consider a system with 1 tape drive and 1 plotter. Process 1 asks for the tape drive and gets it. Process 2 asks for the plotter and gets it. Now process 1 asks for the plotter, and is blocked until it becomes available. Unfortunately, process 2 does not release the plotter, instead it asks for the tape drive. Both processes are stuck and will remain so forever. This is called a *deadlock*.

Three deadlock management strategies are in use: 1) Prevent deadlocks by system design, 2) avoid deadlocks at resource allocation time, 3) do not prevent deadlocks; let them occur and have the operator rerun the jobs. If studies show that a deadlock can be expected once every 79 years in a particular system, clearly alternative 3 is preferable to permanently tying up 10k of precious primary memory with somebody's deadlock algorithm, no matter how brilliant.
The most common ways of preventing deadlocks by system design are to reserve all necessary resources when a process starts, or to require a process to release all previously acquired resources before making additional requests. Strategy 2 above is excellent for generating Ph. D. Theses. All methods require the OS to know the maximum requirements of each process in advance. One algorithm grants or denies resource requests according to this criterion: after each resource grant, there must be some execution sequence that allows each process to complete.

5 PROTECTION

5.1 Why Bother?

If a system had no protection one user could invade the privacy of another. The public would not stand for a system in which salary information, medical histories, police records, etc. were essentially open to the public.
Another problem is industrial espionage. Companies have secret information, both technical and financial, which they do not want their competitors to know.
If a system had no protection, one user could destroy the irreplaceable data of another, or in malicious cases, secretly change it.
Several years ago a California university student found a way to penetrate the telephone company's OS. He formed his own corporation and rented a small computer. Using this computer he called up the telephone company's computer and ordered expensive electronic equipment to be delivered to warehouses around the state at 3 a.m. Shortly after each delivery, he would show up with a truck and steal the equipment. He managed to steal several million dollars worth of equipment before he was accidentally caught. After getting out of jail he became a consultant on computer security.
Not too long ago another computer thief managed to steal 200 railroad cars full of merchandise by convincing the railroad's computer that the cars had been scrapped. It is estimated that computer theft in the U.S. alone amounts to 2-3 x 10^9 dollars/year (ALLEN [12]).
Still another reason why protection is important in OS's (as if the above were insufficient) is that without it one user could degrade the service given to other users, for example by hoarding all the disk space.
The problem of protection is greatly compounded by the need for sharing, both technological and intrinsic. When one conceives of a computer utility with thousands of users, some of whom are offering services (programs, data, etc.) and some of whom are using these services, the problem of allowing only permitted accesses becomes very difficult.

5.2 Defects in Present Operating Systems

The protection mechanisms used in most commercial systems are very inhomogeneous. One mechanism is used to protect this, and another to protect that. For example, memory is protected by address mapping, relocation and bounds registers, or lock and key schemes. Files and logging in are protected by passwords. Access to peripherals and terminals is protected by internal OS tables. Privileged instructions are protected by a user mode/system mode bit in the PSW, which is in turn protected by yet another mechanism.
It would be much more secure if a single mechanism could be used for all protection. The PDP-11 for example, has no i/o instructions. Instead i/o is performed by setting bits in special memory locations. Thus the memory protection mechanism also protects i/o instructions. A generalization of this idea to the point where memory protection protected everything would be desirable.

Here are some general techniques for compromising security in present OS's. Systems are always in a state of flux. New patches, changes, versions, and releases occur often. As a result, a systems programmer could be bribed into putting a 'trapdoor' in the system, so under certain very, very rare circumstances, security could be breached.
Many OS's allow processes to request memory, but do not erase it before giving it to the process. By sitting in a loop requesting, examining and releasing memory, a process might discover all kinds of interesting things in the residue, such as the system password table.
Some multiprogramming systems have stringent controls on interactive users, but none on batch jobs. This leads to the Trojan Horse attack. Many systems search the user's file directory before searching the system directory. A potential spy could submit a batch job to catalog a special version of the editor or loader in the victim's directory. Whenever the victim called the editor or loader, he would get the spy's

version, which as a side effect would spy on him and record the information somewhere.
Hitting the user abort key on a terminal often leaves the system in a peculiar state. This is a good target for potential attack.
Files are usually protected by passwords. For users with a few files, this is adequate, but for a large project with many employees and many files it is not secure. With 15% annual personnel turnover, a 50 person project will have employees coming and going monthly. A disgruntled employee with knowledge of all the file passwords could wreak havoc. Changing hundreds of passwords every time an employee departed would require a constant stream of memos to all remaining programmers to tell everyone the new passwords. In addition to being an enormous nuisance to the programmers and an administrative headache, having hunderds of paper copies of the password lists floating around is obviously a serious security problem itself.
Similar problems exist for a software house that makes a program available, and charges by usage (e.g. 75 cents per compilation). Preventing customers or competitors from copying the program is nearly impossible with present OS's. Furthermore, taking back permission from customers who terminate the service but continue using the computer for other reasons is well nigh impossible with password protection.

5.3 **The Confinement Problem**

As a test to see how much protection an OS offers, consider the following problem (LAMPSON [11]). There are 2 processes involved, the service and the customer. The customer makes contact with the service and provides it with information. In return the service performs some computation, and gives the customer the result. The cost of the service rendered is given to the service's owner, who then mails a bill to the customer's owner. (Owners are human beings, not processes). One such service is a program to help people with their income tax. Neither process trusts the other. The customer might steal the tax program, and the tax program might steal the customer's financial information.
Some of the ways the service might leak information to a third party are quite subtle. If the service has memory, it can store the information for later collection. If the service can create permanent files, it can store the information there. If the service can create a temporary file and grant access permission to a third party, the third party can read the file quickly while the service is still active. The service might be able to leak via the interprocess communication facility. The service could encode information in the bill sent to its owner. The service could lock and unlock a file in a clocked manner, so a third party process could acquire information by continually checking to see if the file were locked, because the time function locked(t) is a binary bit stream. Worst of all, the service could degrade system performance (by heavy paging, CPU usage etc.) in a clocked manner. A third party could notice the degradation and decode it as a binary bit stream. Of course this channel is very noisy and has a low bandwidth, but information theory techniques could be used to insure reliable transmission.

5.4 **Protection Mechanisms**

In MULTICS and some other advanced OS's, a user can map a file onto a segment, i.e. the file becomes part of the virtual memory. This allows the segment protection mechanism to be used for active files as well. One protection mechanism has a protection matrix encompassing all active segments. The entry M_{ij} specifies the access segment i has to segment j (read, write, call, etc.). Unfortunately, the matrix M is too large to be of much use in most systems.
In some systems protection on segments is provided within each process, with the possibility of shared segments. In MULTICS each segment has a ring number, the lower the ring number the more protected. If the curent procedure is in ring i, any attempt to access ring j where $j < i$ will be trapped by the hardware. The OS itself operates in each process' address space in ring 0. A user wishing to build a debugging system to debug his own programs can put the debugger in ring k, and the debuggee in ring $k + 1$, thus protecting the debugger.
A number of recent systems use the concept of a *domain* as a protection environment, a generalization of the MULTICS rings. In such systems, there exist various classes of objects-processes, files, segments, mailboxes, i/o channels, terminals, peripherals etc. Within each domain, a certain set of objects is accessible, and with certain strings attached e.g. a read only file. Each process is in some domain at each moment, and has access to all the objects of that domain. Note that the same object may appear in several domains with different permissions.
One can imagine a giant matrix whose rows are the domains and whose columns are the objects. Each entry tells what access the domain has to the object, if any. There are 2 practical ways to implement this scheme. First, associated with each domain is a list of all the objects it has access to. This list contains all the nonzero entries in its row of the matrix. Each item in this list is called a *capability* (DENNIS & VAN HORN [12]).

A capability has 4 parts: a unique number (e.g. date and time the object was created, expressed in microseconds elapsed since 0000 GMT, 1 Jan. 1901.), the object type, the access bits, and a pointer to the object (its PCB), disk location etc.). The object itself also contains the unique number. If an object is destroyed and the space it occupied reused, an attempt to access it via an old capability will be detected because the unique numbers will not match.
The access bits depend on the object type. For a file, *read, write, execute, copy, give away, destroy* and *extend* might be reasonable. For a process, *stop destroy,* and *communicate with* might be appropriate accesses. In some systems users or at least subsystem writers, can create new object types, and define what the access bits mean.
Needless to say, the protection system would be worthless if a process could manipulate its own capability list. The capability litsts must be maintained by the OS. Whenever the OS creates a new object

for a process (e.g. a file) it inserts the capability for the new object in the process' capability list, and returns the index, i, of the new capability (i.e. its position in the C-list). To refer to an object, a process uses the index of the capability. Using this scheme, all objects are protected in a simple, homogeneous way. Capabilities can also be used for accounting by using some of the access bits for integer values. For example, to use the CPU one would need a capability for it, one of whose fields was the CPU time allowed. The other way to slice the domain-object matrix is by columns. Associated with each object is a procedure which is invoked on every access. The procedure could use any method it wanted to in order to restrict access (such as asking permission from a logged in user).

5.5 Fundamental Principles of Protection

Protection should be based on explicit permission, not exclusion. The default is no access. Among other things, security failures then generally show up in the form of prohibiting an allowed access instead of allowing a forbidden one. Check every access for current authority; otherwise permission gets stored away in local memories. The design should be public. Protection should not be based on the assumption that a potential attacker is ignorant. Do not give anyone more access than he needs. The human interface must be easy to use, or people will not use it. Some users want strange things; provide him with facilities to create his own protection subsystems.

5.6 User Authentication

In MULTICS, each registered user of the system has his own private password. That is the only place passwords are used. The system keeps track of who may access what (e.g. associated with every file is an explicit list of users allowed to access it). Once you are logged in as JANSEN you can do everything JANSEN can do.
To reduce the chance of one person logging in as another the following measures are taken. Terminals do not echo when passwords are typed, so there is no written record of the password. A user may change his password at any time. Not even the system administrator or computer center director can discover a user's password. Passwords are stored internally in an encrypted form that cannot be inverted. This list could be posted on the terminal room wall and it would not compromise system security (see EVANS, KANTROWITZ & WEISS [13]). To discourage users from picking easy to guess passwords, the system supplies a random password generator that uses English digram frequencies (e.g. foat, zabel, norbid but not qfupz, hqiznp, jjaqrp). All batch jobs must be started by a logged in user. After a period of N minutes of inactivity and after all crashes, the user must log in again. The system forcibly breaks the telephone connection after 10 unsuccessful log in attempts (to make random searches harder). All login's are recorded, and the time and place of the previous one is printed after each login (if someone else logged in as you, you will at least be aware). Lastly, a user can be set up to run a specific program after login. This program could begin asking questions, etc. to give a user even more protection against an intruder.
Other authentication systems used elsewhere include the following. Passwords that are good for one use only, with the new one being typed at logout, or presupplied by user. (Stealing used passwords has no value). Special terminals that require insertion of a magnetically striped card during password type in. Systems that hang up after login, and automatically call the user back (this requires the intruder to get into the user's office).

6 PERFORMANCE MONITORING AND EVALUATION

Operating systems are so complicated that it is difficult to tell how well they are working. In order to tell if a new version is better than the old one, it is necessary to make extensive measurements on both. Such measurements include things like CPU idle time, time spent in various processes, including system activities (IPC, scheduling, paging etc.), virtual/real memory ratio, paging behavior etc.

6.1 Measurement Tools

Some of the basic tools used for performance monitoring are as follows. Counting invocations and time spent in each OS procedure. Histograms of the program counter sampled every T milliseconds (clock interrupt). Records of page and segment faults. Event tracing (messages sent, interrupts, i/o, process switching etc.). Use of an external computer to display statistics in real time. A program sitting in a tight loop reading the clock, and logging all gaps (this can measure interrupt handling time, and find scheduler errors). The ability to have charged CPU time, number of page faults, etc. printed on each terminal after each command. To reproduce input behavior for making tests, an external minicomputer attached to several telephone lines is best.
When constructing a synthetic workload, one method is to use a known mix of assemblies, compilations, sorts, matrix inversions, etc. Another method is to characterize each job by its demand of system resources, e.g. CPU, memory, page faults, file usage, etc. Thus each job can be characterized by an n-tuple. Run the OS for a while, and collect statistics on these n-tuples. Use these to construct the probability density function in n-space. Write a test program that demands services randomly according to this p.d.f.

6.2 Modeling

Analytic models of OS behavior are useful for understanding complex systems. For example, should a computer center buy a 500 nsec CPU with 500k primary memory or a 200 nsec CPU with 300k primary memory for the same price?
To give an idea of how modeling can be used, consider a system in which the mean time between page faults is linearly proportional to the primary memory, M, allocated to a process' pages, i.e. 1 page fault every aM sec. [14] Assume a normal instruction takes T_1 and a page fault causing instruction takes T_2 sec. The average instruction time, $T = T_1 + T_2/aM$ sec/instr. Suppose the machine rental cost is $C_1 + C_2M$

guilders/sec, where C_2 accounts for memory charge, and C_1 everything else. The cost per instruction is then $(C_1 + C_2M)(T_1 + T_2/aM)$. Minimizing this with respect to M we find that optimal memory size is $(C_1T_2/aC_2T_1)^{\frac{1}{2}}$.

References

1 FARBER, D. J., *A Distributed Computer System,* Report TR-4, Dept. of ICS, Univ. of Calif., Irvine, 1970.

2 WULF, W., E. COHEN, W. CORWIN, A. JONES, R. LEVIN, C. PIERSON, & F. POLLACK, *HYDRA: The Kernel of a Multiprocessor Operating System,* CACM 17 (1974) 337-345.

3 DIJKSTRA, E. W., *Cooperating Sequential Processes,* in: Programming Languages, F. Genuys (ed.), Academic Press, New York, 1968.

4 BRINCH HANSEN, P., *Operating System Principles.* Prentice-Hall, Englewood Cliffs, N.J., 1973.

5 PETRI, C. A., *Communication With Automata,* Rome Air Develop. Cent., Suppl. I to Tech. Rep. No. RADC-TR-65-377, Reconnaissance-Intelligence Data Handling Branch, Rome Air Develop. Center, Griffin AFB, New York, 1966.

6 DIJKSTRA, E. W., *The Structure of the 'THE' Multiprogramming System,* CACM 11 (1968) 341-346.

7 BRINCH HANSEN, P., *The Nucleus of a Multiprogramming System,* CACM 13 (1970) 238-241.

8 TANENBAUM, A. S., *A General Purpose Timesharing System for the PDP-11/45.* Wiskundig Seminarum Report IR-2, Vrije Universiteit, Amsterdam, 1973.

9 GOLDBERG, R. P., *Survey of Virtual Machine Research,* Computer 7 (1974) June, 35-45.

10 ALLEN, B., *Embezzler's Guide to the Computer,* Harvard Business Review, July-August, 1975, 79-89.

11 LAMPSON, B. W., *A Note on the Confinement Problem,* CACM 16 (1973) 613-615.

12 DENNIS, J. & E. VAN HORN, *Programming Semantics for Multiprogrammed Computations,* CACM 9 (1966) 143-155.

13 EVANS, A., JR., W. KANTROWITZ, & E. WEISS, *A User Authentication Scheme Not Requiring Secrecy in the Computer,* CACM 17 (1974) 442-445.

14 SALTZER, J. H., *A Simple Linear Model of Demand Paging Performance,* CACM 17 (1974) 181-186.

Reading list (books on operating systems)

BROOKS, F., *The Mythical Man Month.* Addison-Wesley, Reading, Mass., 1975.

BRINCH HANSEN, P., *Operating Systems Principles.* Prentice-Hall, Englewood Cliffs, N.J., 1973.

COFFMAN, E. G., JR. & P. J. DENNING, *Operating Systems Theory.* Prentice-Hall, Englewood Cliffs, N.J., 1973.

FREEMAN, P., *Software Systems Principles.* Science Research Associates, Chicago, 1975.

KATZAN, H., JR., *Operating Systems: A Pragmatic Approach.* Van Nostrand Reinhold, N.Y., 1973.

MADNICK, S. E. & J. J. DONOVAN, *Operating Systems.* McGraw-Hill, N.Y., 1974.

ORGANICK, E., *The Multics System.* MIT Press, Cambridge, Mass., 1972.

SAYERS, A. P., S. KURZBAN, & T. S. HEINES, *Operating Systems Principles.* Petrocelli/charter, N.Y., 1975.

SHAW, A., *The Logical Design of Operating Systems.* Prentice-Hall, Englewood Cliffs, N.J., 1974.

TSICHRITZIS, D. C. & P. A. BERNSTEIN, *Operating Systems.* Academic Press, N.Y., 1974.

WATSON, R. W., *Timesharing System Design Concepts.* McGraw-Hill, N.Y., 1970.