

3 Operating Systems

3.1 ORIGINS OF "OPERATING SYSTEMS"

Back in the early 1950s there were very few computers. Large universities might have one, as might government research laboratories, and a few big companies. A scientist or engineer wanting to run a program would book time on the machine. When their turn came, they would be totally in control of "THE COMPUTER".

They had to load the assembler (or high level language translator), use it to convert their program to binary instruction format, load the binary version of their program, start their program using the switches on the front of the computer, feed data cards to their program as it ran, etc. It all involved a lot of running around fiddling with card readers (and, later on, magnetic tapes), pressing buttons, flicking switches. Most people didn't do the job very well and wasted a lot of the time that they had booked on the machine.

Organizations that had computers soon started to employ professional "computer operators". Computer operators were responsible for getting maximum use out of the computing equipment. They would run the language translators and assemblers for users, they would load binary card decks and get the programs run. Since they worked full time with the computers, they were familiar with all the operating procedures involved and so made many fewer mistakes and wasted much less time.

Computer operators

Since users no longer directly controlled the running of their programs, they had to write "job control instructions" for the operators. These job control instructions would tell the operators of any special requirements such as the need to mount a particular magnetic tape on which a program was to store some generated data.

Computers were changing rapidly in the early '50s. Peripheral devices were becoming more complex and versatile. Many could operate semi-autonomously. They could accept a direction from the CPU telling them to transfer some data and could organize the data transfers so that these occurred while the CPU continued executing a program. Such changes made for more efficient program execution because the CPU spent less time waiting for data transfers. But such changes also meant that the code to control peripheral devices became a lot more complex. Any erroneous I/O code could seriously disrupt the workings of the computer system and might even result in damage to peripheral devices or to media like magnetic tapes or punched cards.

"System's code" for
i/o

Originally, a programmer would have been able to use all of the computer's memory (if possible, the area with the loader program was left alone so that the loader code wouldn't have to be toggled into memory again). The requirements of the new peripheral devices made it advantageous to reserve a part of memory for "systems code" – the specialized code that controlled the new devices. The new organization for main memory is shown in Figure 3.1; part of memory is reserved for the "system's code" (and, also, for system's data – the system needed memory space to record information such as identifiers associated with magnetic tapes on different tape decks).

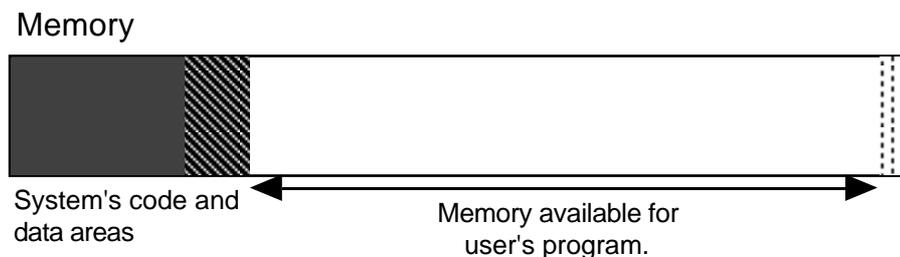


Figure 3.1 Part of main memory is reserved for "system's" code and data.

The "systems" area of memory had subroutines for whole variety of input output tasks. There would have been a subroutines to handle tasks such as reading one character from the control keyboard or one column of a punched card. Another routine would exist to organize the transfer of the contents of a number of words of memory (containing binary data) to a tape block. Programmers could have calls to these "systems subroutines" in the code of their programs. (Later, if you study computer architecture and operating systems you will learn why a specialized variant on the JSR subroutine call instruction was added to the CPU's instruction set. This specialized instruction would have had a distinct name, e.g. "trap" or "supervisor call (svc)". Calls to the systems subroutines were made using this "system call" instruction rather than the normal JSR jump to subroutine.)

Usually, rather than making direct calls to the "system's code" programmers continued to make use of library routines. For example, a program might use a number input routine. This routine would read characters punched on a card (making a call to one of the systems' input routines to get each character in turn) and would convert the character sequence to a numeric value represented as a bit pattern. There would be two such input routines (one for integers, the other for real numbers). There would be corresponding output routines that could generate character sequences that could be printed on a line printer.

These early systems established a kind of three component structure for a running program. The structure, shown in Figure 3.2, is the same as on modern computer systems. The three kinds of code are: the code written for the specific program, library code (provided by colleagues, commercial companies, or possibly the computer manufacturer), and "system's code" (usually provided by the computer manufacturer, though sometimes purchased from other commercial companies).

Memory

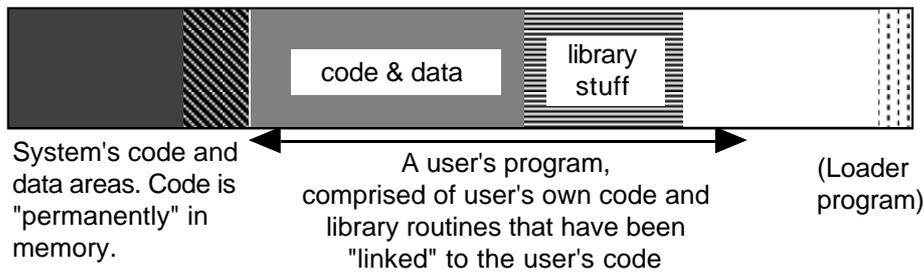


Figure 3.2 System's code, library code, and user's code.

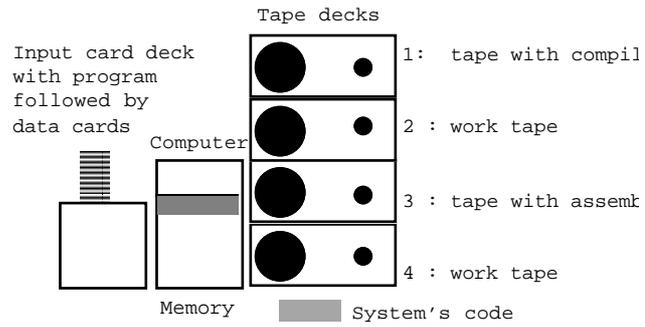
By the late 1950s, memories were getting larger. Many machines had as many as 4000 words of memory (equivalent to about sixteen thousand to twenty thousand bytes in modern terms). Some machines had as much as 32000 words of memory. These larger memories allowed the systems code to be extended in other ways that could improve the utilization and performance of the computer system.

Quite substantial amounts of time tended to be wasted between jobs and between each phase of a job. For example a user might submit a job written in the high level language Fortran along with a deck of data cards to be processed. The operator would have to load the Fortran compiler (translator) and feed in the cards with the program; the translator would output assembly language code that would be punched on cards or written to tape. When the compiler stopped, the operator had to load the assembler program and feed in the generated assembly language source, along with source card decks for any assembly language library routines the user had requested. The assembler would produce binary code (on cards or tape). The operator then had to load the binary and start the program running. Once the program was running, the operator had to feed in the data cards. When the program stopped, the operator had to collect up cards and printed output and then reset the machine ready for the next job. Each of these steps would involve the operator attending card readers, punches, tape decks, and control consoles.

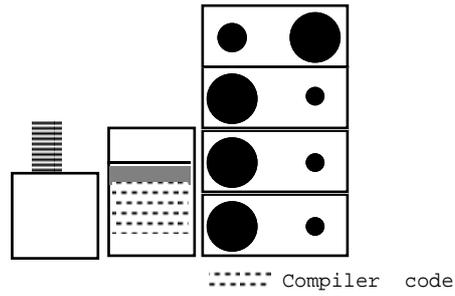
System's code to help the operators

If a computer system only had card input and output, the entire process was very slow. It could take several minutes to load a Fortran translator, run a user program through the translator and punch a deck of cards with the assembly language translation. Most large systems would have had several tape decks which made things a little easier. But it was still an involved process; the operator would have to work at the control switches on the front of the computer, entering instructions that would cause the computer to load the translator or assembler program from tape, then setting the switches again before starting the loaded program.

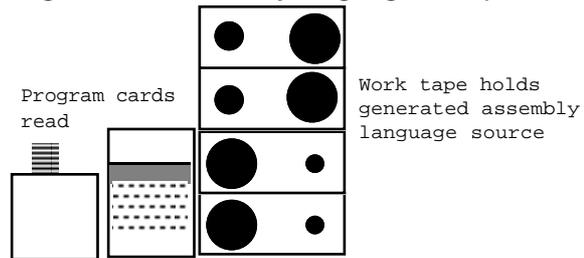
The process is illustrated in the various panes in Figure 3.3. The computer system would have had four or more tape decks; one would have the tape with the code for the compiler, the other would have the assembler. The other two tape decks would hold work tapes where intermediate results could be written. The operator would start by loading the compiler into memory and putting the card deck with program and data on the card reader.



Operator gets computer to load compiler



Operator runs compiler, it reads program and generates assembly language on tape



Operator loads assembler and rewinds tape with generated assembly language

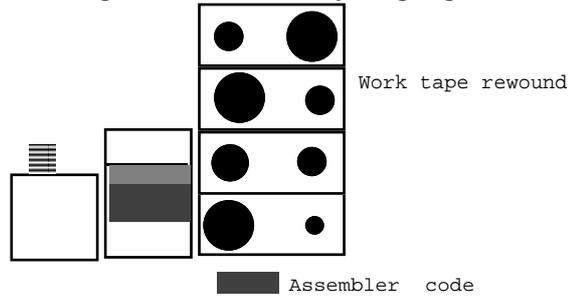


Figure 3.3 Operating the computer.

When the compiler had been loaded, the operator would use the control switches to start it running. The compiler would read the cards with the source code and generate assembly language that would be written to tape. The operator then had to rewind this tape, and load the assembler. The assembler then had to be run; it would read its input from the first work tape and produce the binary code (with the bit patterns for instructions) on a second work tape. When the assembler finished, the operator again had to rewind a tape and then again get a program (now the user's program) loaded into memory. Finally, the user's program could be started. It would read its data cards and produce output on a line printer.

Every time the operator had to do something like enter data on switches or rewind a tape, the CPU would be left idle. Since operators wouldn't know how long it would take to translate a program, or assemble it, or run it, they couldn't be instantly ready to start the next step in the process. So time was inevitably wasted. The CPUs were desperately expensive (CPUs with a fraction of the power of a modern personal computer would have costed millions of dollars) so it was very important to try to keep them working as much as possible.

Most of the operators' tasks involved simple, routine things that could be automated and speeded up. Control routines were written to automate the routine tasks like loading the compiler and rewinding a tape. This control code was added to the systems' i/o code in the reserved area of memory.

These control programs, written to automate routine tasks for the computer operators, were the first "Operating Systems". They worked by reading "control cards" and performing standard actions.

Programs were submitted along with "job control cards" that could be interpreted by the computer's control program. An input card deck (circa ≈1960) would have been something like the following:

*Job control
"languages"*

```
JOB          USERNAME
FORTRAN      ...
C  MY PROGRAM TO COMPUTE HEAT FLOW IN REACTOR
   DIMENSION A(10,10)
   ...
   END
ASSEMBLE     ...
LINK         ...
RUN          1.2    4.5
ENDJOB
```

The control program used the control cards like the "JOB" card. These would have a keyword (and usually additional information, the JOB card might have identified the user who was to be charged). A job would be terminated by an ENDJOB card; if anything went wrong (like a coding error detected by the compiler or an arithmetic error during execution) the operating system could tidy up by reading all remaining cards in the card reader until it found the ENDJOB.

The operating system code handled each control card in a standard way. When it read a JOB card, it would print an accounting message on the operator's terminal. A FORTRAN card was processed by loading the FORTRAN compiler into memory and then "calling" it to read all the FORTRAN source cards. The operating system

would also arrange for an output tape to be ready for the compiler to use. If there were no compile errors, the operating system organized the rewinding of the output tape and the loading of the assembler. The other steps, like the linking process where "library" code was read from tapes were handled in similar fashion. If the process resulted in successful construction of an executable program, this was then "RUN" so that it could process its data cards. The FORTRAN compiler, the ASSEMBLER, the LINKER, and the user program were all executed rather as if they were subroutines called by the controlling operating system.

3.2 DEVELOPMENT OF OPERATING SYSTEMS

By the early 1960s, the larger computer systems had the form shown in Figure 3.4. Disks were not common, most systems made do with tape decks. An early disk unit would have been about five feet high, would have had a multi-platter arrangement with disks about 18 inches across. A complete disk unit would have held about as much data as a modern floppy and would have costed – well certainly more than a new car, probably something close to the price of a small suburban house.

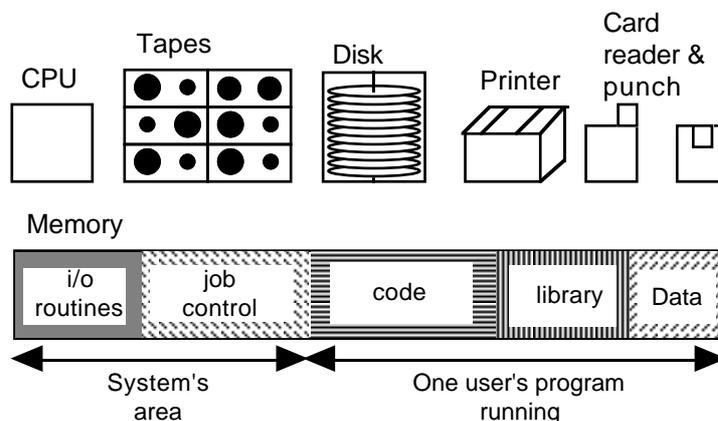


Figure 3.4 "Mainframe" computer system of early 1960s.

Most of the computers of the time got by with just tapes, but some had disks. Disks were a lot faster than tapes (faster transferring data, no need to read all the data sequentially because the disk control could move the disk heads to the blocks containing the specific data required, no need to "rewind", ...). A disk unit could substitute for several tapes. Space on the disk would be partitioned: e.g. blocks 0-159 holding the FORTRAN compiler, 160-235 holding the COBOL compiler, 236-263 with the RPG compiler, 264-274 for the ASSEMBLER, 275-304 containing the FORTRAN subroutine library, 305-310 for the LINKER, 311-410 for "workspace 1", 411-511 for "workspace 2". But even if a disk was being used, the operations remained much the same as described for the tape-based machines.

Part of the machine's main memory would be reserved for the primitive Operating System. This would have consisted of a set of standard i/o subroutines that controlled the various peripheral devices and the subroutines that interpreted the job control cards. In addition to this Operating System (OS), the computer would have a number of pieces of software supplied by its manufacturer:

- FORTRAN compiler (translator for a programming language intended for use by scientists and engineers)
- FORTRAN subroutine libraries (by now these would contain subroutines for doing things like sorting, complex mathematical operations like 'matrix inversion', as well as more standard things like converting digits to numeric values or calculating SINE(), LOG() etc)
- COBOL compiler (translator for a business programming language)

and others.

The job control component of the OS minimized the time needed by the operators to set up the machine between successive jobs, and the time spent organising successive phases of an individual job. The system's i/o routines could arrange for i/o to proceed simultaneously with computations. Together these improved the efficiency of machine usage. But there were problems. For example:

- CPU usage was still relatively low. Even with smart peripherals running simultaneously with computations, most programs spent a large amount of time (often 50% or more) in "wait loops" waiting for input data to be read from devices, or waiting for output data to be printed.
- Programmers were inconvenienced by slow turn around and lack of opportunity to test and debug their programs. (It was common to have to wait 24 hours for results to be returned, and the 'results' might be simply a statement to the effect that the compiler had found the second card of the program to be wrongly punched!)
- Some programs needed more memory than could be obtained.

These problems, and other related problems, were dealt with in a series of developments undertaken throughout the 1960s and early 1970s that lead to more sophisticated operating systems.

3.2.1 Multiprogramming

The most pressing problem was the "waste" of CPU cycles when the single job in the computer's memory had to wait for an i/o transfer to complete. The CPU was by far the most costly part of the computer system of this period. The prices of other components like the peripherals and memory were falling. Larger memories were practical. The availability of larger main memories provided the solution to the problem of keeping the CPU busy. If a single job in main memory couldn't

keep the CPU occupied all the time, then have two or more jobs in memory (Figure 3.5).

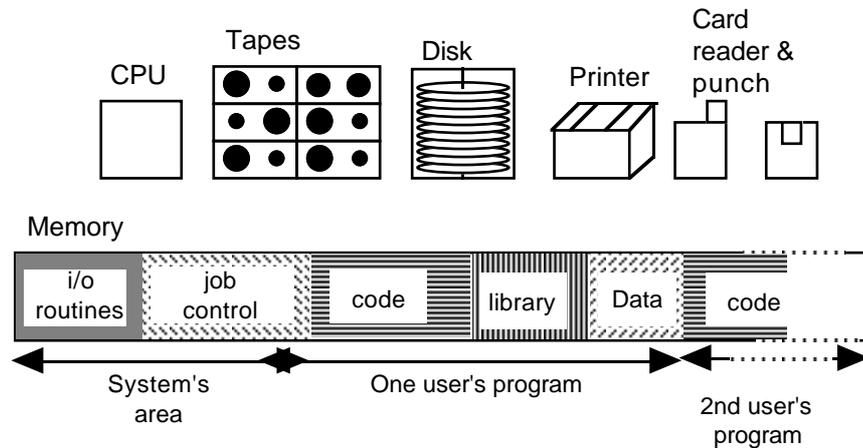


Figure 3.5 Early multiprogrammed system.

The computer's memory was be split up into "partitions".

- One held the OS.
- Another (tiny) part would hold the 'boot' loader (the name 'boot' loader apparently relates to some argument that starting the computer using this loader to read the OS from tape/disk was like 'picking oneself up by one's bootstraps'. ? !)
- The rest of memory was split up among two, three or more program partitions.

A user's program would be run in a partition in much the same way as it had previously been run using all memory not allocated to the OS.

In the early versions of this scheme, only the CPU and the OS were shared. Each memory "partition" had to have its own assigned peripheral devices – a line printer, a card reader, two or more tape decks, and a separate allocation of space on disk (if there was a disk). The OS had to be more elaborate because it had to keep track of each of the different jobs and of the way peripheral devices had been allocated to jobs. If a job asked the OS to perform some i/o operation that involved delays, the OS had to switch the CPU to execute instructions for a job in a different memory partition. Really, the more elaborate OS was using a computer with a large memory to simulate the working of two (or more) smaller, slower computers!

The "partitioning" of memory was adjustable. The operators could enter commands at the control terminal that would get the OS to rearrange the way memory was used. For example, in the morning the operators might arrange to have two medium sized partitions to run average jobs. In the afternoon, when lots of programmers had submitted programs that needed small test runs, the operators could reconfigure the machine to have three small partitions. Overnight, the

machine would be used with all the non-OS memory in a single partition so as to allow bigger programs to run.

The human operators were involved in the relatively slow, and infrequent, process of reconfiguring the machine for a different degree of "multiprogramming". The process of switching the CPU among the different programs was all done automatically, in mere millisecond times, under the control of the code in the OS. Switching would normally happen only when a running job made a request for i/o to the OS; a request like "card reader, read a card and store all 80 characters in this portion of memory ...", or "line printer, this portion of memory holds the 132 characters to be printed on a line, please print them", or "tape drive 8, transfer your next data block into memory starting here ...". The OS would get the device to start the data transfer, and arrange to be notified when the transfer was complete. But the requesting program usually wouldn't be able to continue until the transfer was done – so the OS switched the CPU to work for another program.

Essentially all operating systems now use some form of multiprogramming system. Support for multiprogramming may be limited in an OS for a personal computer (after all a personal computer is to be used by just one person and that person can probably only concentrate on one thing at a time); a personal computer may have many different programs concurrently in memory but the CPU may only get switched when the user selects the window associated with a different program. Larger machines use elaborate multiprogramming schemes. Human intervention is no longer needed to change the degree of multiprogramming. The job control portions of the OS now include code for elaborate algorithms that work out how many jobs should be allowed to run to get best performance. Rather than fixed memory partitions, these modern multiprogramming systems can reorganize memory whenever this is found to be necessary.

3.2.2 Timesharing

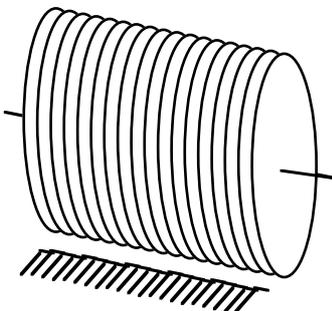
Multiprogrammed systems improved the usage of the CPU – but didn't do that much for the programmers.

Programmers were lucky to get one test run of their programs each day. While this may have encouraged a meticulously careful style of coding it really didn't do much for the productivity of the average programmer. Programmers needed a system where they could edit their code making additions and corrections, give the new code to a language translator (compiler) and either rapidly get any coding errors identified or get a program that could be run. If a program was successfully compiled, programmers needed to make small test runs that allowed their programs to be tested with small data sets.

"Timesharing" systems, pioneered at MIT in Boston USA starting around 1960, provided programmers with more appropriate development environments. It was the development of disk (and drum) storage that made these new systems possible. (Drums have ceased to be used. They are a variation on disks in which data is recorded in tracks on a cylinder rather than on a flat platter. Each track had its own read-write head, see Figure 3.6. Drums were a lot faster than disks, but the

multiple read-write heads made them expensive. They have gone out of fashion because their maximum storage capacity is a lot less than a disk.)

Metal cylinder, surface covered by magnetic oxide, tracks on surface of cylinder



A read/write head for each track

Figure 3.6 Drum storage.

The CPU of MIT's original time-sharing computer was a slightly modified version of the largest computer then commercially available – an IBM 7094. The machine had two blocks of memory, both with 32K (i.e. 32768) words. One block of memory held the operating system. The other held the program and data for a single user. The computer had several very large disks with a total capacity of several million words of storage (≈ 50 Megabyte in modern terms?).

The computer was intended to support up to 32 people working simultaneously. Users worked at 'teletypes' (a kind of electric typewriter) that were connected to the computer via a special controller (see Figure 3.7).

Only one user's program was actually running at any time. It was the program that was currently in memory. The other 'users' would be waiting. Their programs would be saved on the drum.

A user's program was allowed to run until either it asked the OS to arrange some input or output (to the user's teletype or to a disk file); or until a 'time quantum' expired. (A time quantum might be ≈ 0.1 seconds.) When a program asked for i/o or used up its time quantum it would be '*swapped out to the drum*'. 'Swapping out' meant that the content of user memory was written to the drum. A different user's program would then be 'swapped in'. This program would then be allowed to run until it exceed its time quantum or asked for i/o. The second program would then get swapped out and a third program would be swapped in. Eventually, it would be program one's turn to run again.

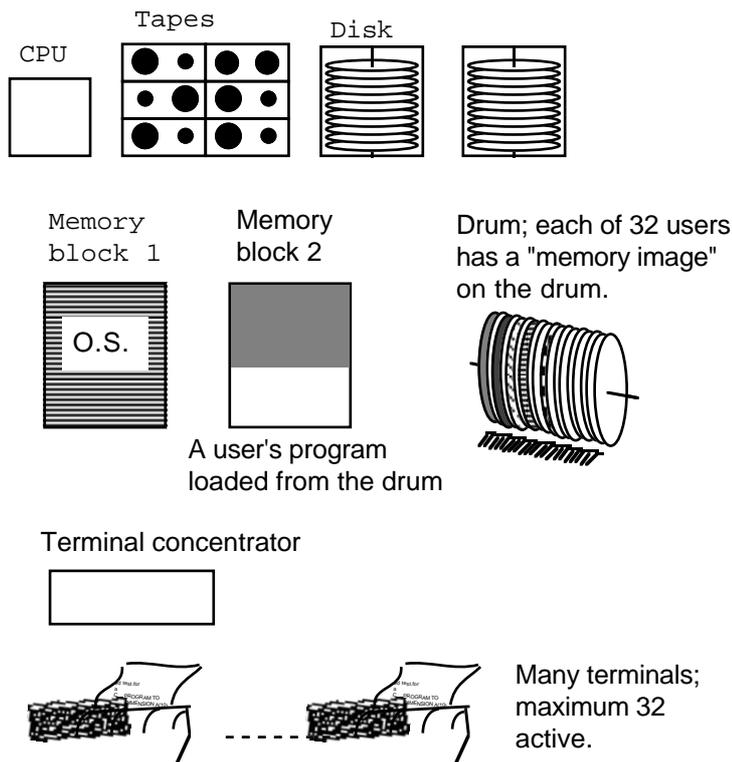


Figure 3.7 MIT's pioneering "time-sharing" computer system.

Even if the system was very busy with lots of users, each individual user would get ≈ 0.1 seconds worth of CPU time every three seconds. So it would seem to an individual user that their program was running continuously, albeit on a rather slow computer.

Operating systems for personal computers don't use timesharing. (They are personal computers because you don't want to share!) But most other operating systems (e.g. the Unix system you may use in more advanced CS subjects) incorporate sophisticated versions of the timesharing scheme pioneered by MIT.

Modern computer systems have much more memory and so don't have to do quite as much "swapping"; but swapping is still necessary. (Swaps are between memory and disk as the machines won't have drums; but quite often there will be a special small high speed disk that is used only for swapping and not to hold files.) If you get to use Unix, you will notice when you've been "swapped out"! (The machine seems to ignore you.)

There are obvious similarities in "multiprogramming" and "timesharing"; both involve the OS controlling the use of a computer system so that it appears as several computers (lots of little ones, or lots of slow ones). Multiprogramming and timesharing evolved to meet different needs; multiprogramming was to optimize CPU usage, timesharing was an attempt to optimize programmer usage! The

switching of the CPU and swapping of programs after a time quantum is quite wasteful; during the switch the CPU isn't doing any useful work. In the '60s when CPU power was at a premium, timesharing systems were not common.

Modern OSs combine timesharing with "background batch streams" (successor to multiprogramming) to get the best of both approaches. If no interactive user needs the CPU, the OS allocates it to a long running, multi-programmed "background job".

3.2.3 File Management

Users of the MIT system were able to keep their own files on disk. Operating systems had to be extended quite a lot before they could allow users to have their own files like this.

The first disks were limited in capacity and all their storage capacity was needed for systems software – compilers, assemblers, subroutine libraries, work areas for the compilers etc. As suggested in an earlier example, the system's administrator could allocate the space; e.g. "FORTRAN compiler in blocks 0..159" etc. This was adequate when there wasn't much disk space and whatever there was had all to be used for systems software. As disks got larger, it became possible to allow users to have some space. But this space couldn't be allocated to users in the same way as the system's administrator had allocated it for systems software. It was no good telling a user "You can have blocks 751 to 760" because they'd make mistakes and use blocks 151 to 160 (and so overwrite the compiler!).

Every attempt by a user program to read from or to write to the disk has to be checked another task for the OS. The OS has to keep records of the disk space used by each user. The simple form of "file directory" (illustrated in Figure 1.14) has to be made more elaborate to suit a shared environment. In shared environments, the disk directories contain details of file ownership and access permissions as well as the standard information like file name and block allocation. The time-shared computer systems started to have files owned by the operating system with lists of the names of users; the main directory on the disk would have an individual directory for each user. These individual directories contained details of the files belonging to that user. The OS could impose restrictions on the number of files, or total space allocated to an individual user. Users could specify that they wanted files to be private, or to be readable by other users in a cooperative work group.

Once the OS had acquired records of users, it could keep records of jobs run, CPU time consumed and so forth. Previously, such accounting information had had to be manually logged by the computer operators. By the mid to late 1960s, these accounting tasks had become another responsibility of the OS; the operators merely had to run a program that printed out the bills that had been calculated using the information that the OS had stored.

3.2.4 Virtual Memory

In the late '50s and early '60s, computer memories were still quite small; it was rare for a computer to have more than 32K (32768) words of main memory (in modern terms, that is about 120,000 to 160,000 bytes). Some of this memory had to be reserved for the OS. This meant that programmers were limited in the amount of code and data that they could get in memory.

Once disks became available, schemes were devised that allowed programmers to have programs and data that exceeded the machine's main memory capacity. When the programming was running, only parts of its code and data would be in main memory, the rest would be on disk. Initially, individual programmers were responsible for organizing their program code and data so that both could be stored partly in memory and partly on disk. As shown in Figure 3.8, the programmer had to break up a program into groups of subroutines that got used in the same "phases" of a program.

"Overlays"

A program would typically start by using one group of routines (some taken from a library) to read in data, then another group of routines would do calculations needed to set up for the main processing phase, a different group of routines would do the main calculations, another calculation phase would follow to get results that were to be printed, and then some routines would be used in an output phase. At any one time, only a few of the routines needed to be in main memory.

The program would have some fixed amount of space for code in main memory. The complete program was stored on disk (spread over many disk blocks). When the program was run, a small part of this code would have to be in main memory. This chunk of code would have included the main program, and any frequently used subroutines. The main program had to have extra code to load the different 'overlays' as each was needed. It would start by loading the 'overlay' with the input routines (Figure 3.8A). Once the main program had loaded the input 'overlay' it would call the input routines; these would read the program's data. When the input phase was complete, the main program would resume and arrange the loading of code for phase 1 of the calculations. The phase 1 routines would then be called (Figure 3.8B). The main program would continue by loading the overlay with the routines for the next phase. The phase 2 routines would then be executed (Figure 3.8C). The program would continue in this way, loading each bit of code as it was needed.

As well as bothering about shuffling the code between disk and main memory, the programmers had to devise equally complex schemes to shuffle their data. It was very common to have far more data than would fit in main memory. For example, an engineer who wanted to model heat flow in a metal beam might need to work out the temperature at successive time intervals at a number of points on the beam. To get meaningful results, the engineer would have needed a fairly fine grid of points, e.g. modelling 1000 points along the beam and 100 points across the beam. But you can't fit 100,000 real numbers into main memory if you've only got 32K words for the OS, your program and your data.

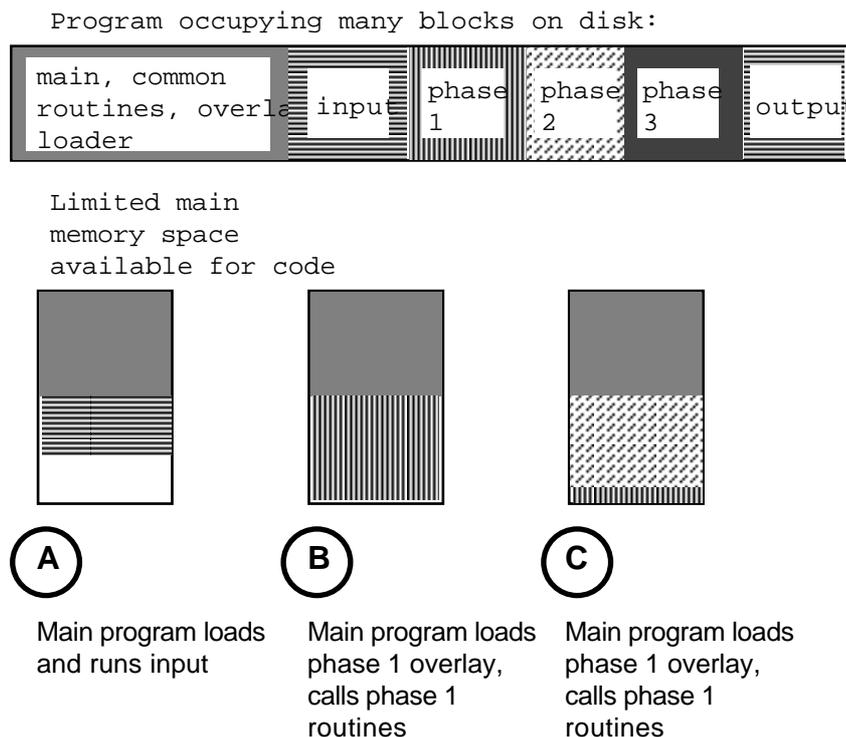


Figure 3.8 Program "overlay" scheme.

When shuffling code, programmers had standard 'overlay' models to follow. When shuffling data, they were on their own and had to invent their own schemes (because the best way of shuffling data between memory and disk depends on the frequency of access and the access patterns used in a particular program).

This need to bother about shuffling code and data around was a distraction. It didn't have anything to do with solving a problem (like working out something about heat flow in beams); it was simply a matter of fighting with limitations of the computing systems available. The programmers just wanted big programs with big "address spaces", but they were being forced to think in terms of a two level storage system – the main memory, and a secondary disk memory.

Another extension to the computer's operating system hid the sordid details of the "two level store" from the programmers. With a little help from some special extra hardware in the CPU, an OS can be made to fake things, so that it seems as if a computer has a very much larger memory than it really does possess.

The OS can arrange to know which parts of a program's code and data are in memory and which parts are on disk. When a chunk of code or data not in memory is needed, the OS can shuffle things around to get that code or data into memory. The programmer need never know that this is happening.

Paging

The Atlas computer built by the University of Manchester, England, around about 1960 was the first with what was then called a "one level store". It combined

16K words of real memory and storage on a drum. The OS, and special "paging" hardware in the CPU, faked things so that the Atlas machine seemed to be a computer with one million words of memory.

It took more than ten years before these memory management schemes became widely used on large computers. The name for the scheme was changed to *virtual storage* or *virtual memory*. (The use of the word virtual is slightly odd. It is really the same usage as in optics where one talks of "virtual images" being produced by magnifying glasses and by mirrors. The virtual store isn't real store – it just looks that way. The term has acquired wider usage, so multiprogramming is now sometimes discussed in terms of *virtual processors* – the OS makes it look as if you have many processors. "Virtual reality" is the latest extension.)

In addition to special "paging" hardware that is needed in the CPU, support for virtual memory requires a lot more code in the OS. But now it is all commonplace; the CPU chips and OSs even for personal computers started to support virtual memory in the late 1980s.

Virtual memory

3.2.5 "Spooling"

Developers of Operating Systems found yet another way in which they could make the OS fake things so that the computer seemed different from what it really was.

A program that wants to print data can ask the OS – "please send this information to the line printer". But what happens if you have a multiprogrammed computer, where the execution of several programs is being interleaved, and more than one of the programs wants to print data?

It obviously won't do to have a program print a line or two then stop, have another program start and print a line, then the first program resume. Interleaved printouts aren't much use.

The first solution was a little expensive; you had to have many peripheral devices. If you intended to have 4 memory partitions, for 4-way multiprogramming, then you had had to buy four separate printers. Although printer manufacturers liked this scheme, it was not widely popular.

Eventually, once most systems were equipped with disks, the OS developers found a work around. Programmers never send output directly to devices like printers – they merely think they do. The OS fakes it so that each program that wants to print thinks that it has its own exclusive printer ("virtual printers").

Rather than send output directly to a device like a printer, the OS would collect ("spool up") all the output from the program and save it in a temporarily allocated file on disk. Each running program could be allocated a separate area on disk to store its output temporarily. When a program finished, the file containing the output that it had written was tidied up by the OS and then transferred to a queue. A little printer control program would form another part of the OS. It would run the printer(s) attached to the computer. For each printer that the system had, the print control program would take a file from the output queue and arrange to get it printed in one piece. As each print file finished, the print control program would arrange to take the next file from the head of the output queue.

3.2.6 Late 1960s early 1970s system

By the late 1960s early 1970s, Operating Systems were becoming large and complex, see Figure 3.9. The figure shows some of the software components that would have made up the OS in memory (the relative sizes of components are not significant).

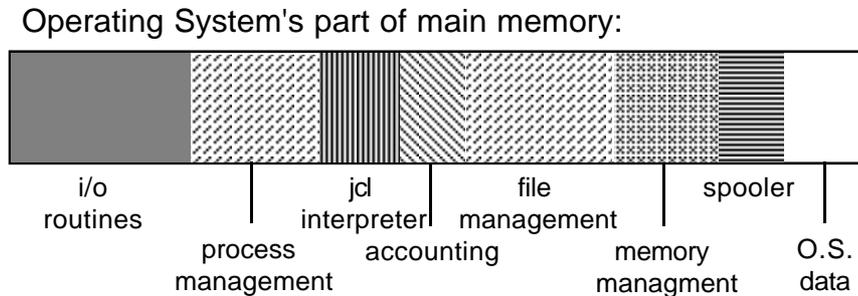


Figure 3.9 OS. of late 1960s early 1970s.

"Device drivers" and "interrupt handlers"

There would be a number of i/o handling routines. These would include "device driver" routines that organize data transfers to/from peripheral devices. Other routines would deal with 'interrupts' that devices use to indicate when they've completed a data transfer. The OS would respond to interrupts by updating its records of which programs were waiting for i/o.

"Process management"

Another large chunk of code would be devoted to "process management", i.e. the scheduling of the allocation of the CPU, the processor, to programs. This code kept track of which programs were ready to use the CPU and which were waiting for i/o from terminals, or disks, or other devices. If the system supported interactive users with 'time sharing', this scheduling code had to make sure that no program ran for too long without stopping to let other programs have some CPU cycles. There would be complex rules to apply to determine which to pick if there were several multiprogrammed jobs ready to run. The system would typically have a large number of jobs queued up on disk, these queued jobs would have been read earlier from cards. Another part of "process management" code would be run regularly to check whether any queued job should be started.

Job control interpreter

The system would have some routines for interpreting the cards with "Job Control Language" (JCL) statements that specify the various processing steps a job required. "JOB" cards would have information that the JCL interpreter had to check to identify the job's owner (so that accounts could be kept and file use could be checked). Other JCL cards would specify things like the compiler (translator) to be used, or files needed to hold data.

File management

File management code would take up another large chunk of memory. This code would allow users to create files to store programs or data on disk; checks would have to be made to restrict users to allowed disk quotas, other checks would be made to see that only authorized users could access a file, and the code would have to make certain that disk blocks were allocated properly so that files didn't get

mixed up. File management code also had to look after things like providing temporary files for 'spooling' program output, special files for "swapping" time-shared programs, and other special files that modelled "virtual memory" for big programs.

Memory management code would organize the allocation of main memory to user's programs. This might be something relatively easy, like maintaining a fixed number of 'partitions' in which programs could run. Where used, things like "virtual memory" made memory management a lot more complex.

Memory manager

There would be some code used to keep accounts so that people could be charged for their computer usage. The OS code would also include minor specialized components like the "spooler".

Spooler and other minor components

All of the code components in the OS would have needed their own tables of data, these data tables would have taken up another area of memory.

OS data areas

3.3 NETWORKING

Originally, computers were located in isolated, air-conditioned computer rooms and they communicated only through their card readers, line printers, and the operators control terminal. Gradually, they began to be linked up. It was limited at first:

- machines like the MIT time-share machine would be connected to many terminals in surrounding buildings,
- a university with a central computer might have extra line printers and card readers located off campus at research centres; these would be linked to the computer by specially installed non-standard telephone lines.

The first real networked systems would have been the US's SAGE air defence system and the air-line seat reservation systems. These were fairly specialized. The main computer only ran a single program. SAGE had a program that kept track of aircraft movements; the seat reservation systems looked after files that contained details of bookings for each scheduled flight in the next couple of weeks.

The main computers were linked by special high-speed "leased line" telephone connections to smaller computers in other towns. These smaller computers acted as concentrators for messages coming from many terminals, either directly connected terminals or terminals linked via slow-speed telephone connections, see Figure 3.10.

During the 1960s, a number of organizations like large banks and airlines started to "network" their computer systems. But the scope was always fairly limited. The organizations had a main computer that ran one program – a seat reservation program, or a bank accounting record keeping program ("transaction programs" they keep track of individual transaction like the cashing of a check). The remote computers joined via special telephone lines just provided input data for transactions. But, a much more innovative scheme for networking appeared around 1968.

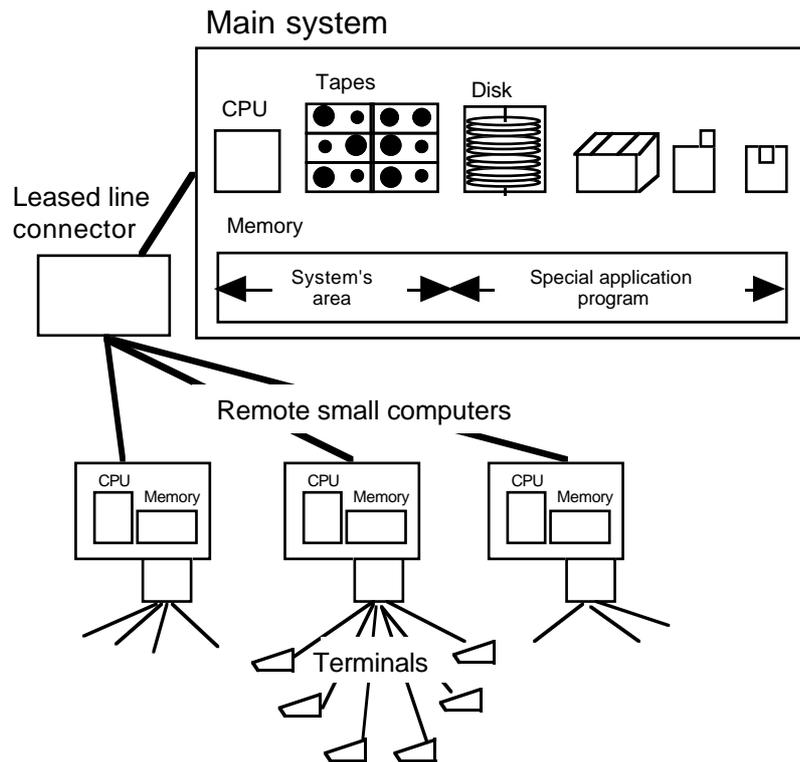


Figure 3.10 Early networked system, such as the airline seat reservation systems.

**The "ARPA" net
(Internet)**

The US Department of Defence (DOD) sponsored a lot of research at universities and research institutes through its "Advanced Research Projects Agency" (ARPA). DOD-ARPA paid for large CDC computers for scientific computing at labs. like Oak Ridge, Los Alamos, and Livermore. It paid for DEC-6 and DEC-10 systems for research into Artificial Intelligence at places like MIT. There were ARPA funded scientists and engineers at many places remote from the centres with the ARPA computers. ARPA wanted to improve access to the shared computers by this wider community of users. This was achieved through the creation of the "ARPA network" (ARPAnet), which is the ancestor of the modern Internet.

The ARPAnet used small computers, IMPs (Interface Message Processors) connected by special telephone lines that allowed high speed data transfers. A particular IMP could be connected directly to one of the main CDC or DEC PDP-10 computers that were ARPA's shared computer resources. Other IMPs simply acted as terminal concentrators, connecting several local terminals to the network.

Scientist and engineers, who were allowed to use an ARPA computer, would have access to a terminal that was connected to an IMP located in their university or research laboratory. At this terminal, they could "login" to any computer on which they had an account --- e.g. RLOGIN HOSTNAME=MITDEC10

USER=WILENSKY. If login was permitted, the chosen host computer would send a response message back to the IMP with details of the job that had been started for the new user. The IMP would record the details – "user on terminal 3 @ me connected to job 20 on host MITDEC10." Subsequently, each line typed by the user was packaged by the IMP.

A packaged message had address details as well as the user's data e.g. if the user entered a "dir" command to list a file directory, the IMP would create a message like "3,IMP16,20,MITDEC10,dir" (terminal 3 @ me to job 20 @ MITDEC10, content 'dir').

An IMP didn't keep open a permanent connection to a remote host. Each message between user and host computer would be routed individually. Each IMP had a map of the network and the IMPs exchanged messages that described current traffic on links and identified areas with congestion. Using this information, an IMP could determine which link to use to forward a message. An IMP that received a message would check whether it should deliver it to a local host machine or forward it. Messages were stored in the IMPs until there was an opportunity to forward them across the next link. A message would make hops from IMP to IMP until it arrived at its destination.

The message would be taken by the host, and the contents delivered to the user's job. Output from the job would be packaged by the local IMP. The response message would work its way back through the network, the return route might differ from the original route.

The original intent of the ARPAnet was simply to provide this fairly flexible "remote login service" so that expensive ARPA computers could be shared by users all over the US. Very soon additional uses were found.

As well as having accounts and file allocations for individual users, most computer OSs have some form of "group accounts"; a systems administrator can add an individual user to any number of these groups. Once the ARPA net was running, researchers located at different places in the US could form a "group" on a particular research computer and work on shared files. Many found that the easiest way to contact their coworkers was to leave messages in shared files (avoiding problems with nobody answering phones, difficulties of different time zones etc). This kind of messaging led to primitive "electronic mail" (email) systems.

These mailing systems were regularized. Each user of a given host machine was allocated a "mail box" file; other users could append messages to this file. News services started about 10-15 years later.

Within a few months of its establishment, the ARPA net went international making use of a telecommunications link via one of the first geostationary satellites to link up with a European subnetwork.

Starting with just a few machines in the late '60s, the ARPAnet grew to several hundred machines by the late 1970s, several thousand machines by the late 1980s (with a name change to Internet – it went between other more localized networks) and now the network has thousands of machines. Nowadays, most host machines manage without IMPs, doing message packaging and routing for themselves.

In the mid-1970s, different types of computer network started to appear. By this time, computers were much less costly; and while it was common for an organization to have a large "mainframe" machine, this would be supplemented by

Message packets

Store and forward message delivery

Remote login services

Collaborative group working and email

Local networks

a large number of "minicomputers" of various forms. Each minicomputer would have some disks, a display screen and keyboard, but they typically wouldn't all have printers, tape units etc. Organizations set up "local networks" that simply joined up the different computers in any one office or department so as to allow them to share things like printers, and have some shared files. These shared file systems allow individuals working on different minicomputers to work together on a large project.

These networks use messages to pass data in way somewhat similar to the wide-area networks. For example if a user at one minicomputer typed in a command that specified that a file was to be copied from a local disk to a shared disk on another computer, the operating system on the mini would exchange messages with the computer that controlled the shared files; the first few messages exchanged would create the new file, subsequent messages would copy the existing file chunk by chunk.

***Broadcasting
message packets***

The message passing mechanism was rather different from that used on the wide area networks. There are a variety of local network schemes ("token ring", "ethernet") with ethernet being currently the most popular. All the computers on an ethernet are connected to something like a co-axial cable. Messages are "broadcast" on this cable. All the computers can read messages at the same time, but only the computer to which a message is addressed will actually deal with the data.

Ethernet networks

Modern personal computers typically have hardware interfaces and software in their operating systems that allow them to be connected to "ethernet" style networks. Each individual ethernet network has to be limited in size, but different local networks can be joined together by "bridges" and "routers" that can forward messages from one network to another. (A bridge or router serves much the same purpose as an ARPAnet IMP; it is a simple computer that runs a program that organizes the forwarding of messages.) Often local networks will have "bridges" that connect them to the worldwide Internet.

3.4 MODERN SYSTEMS

The operating systems written for the mainframe computers of the late 1960s and early 1970s did succeed in delivering efficient CPU usage through "multi-programmed batch computing", as well as supporting interactive timesharing. But these systems were often difficult to use (the "job control language", JCL, statements that had to be written to run a job were hard for the average user to understand, and the job control mechanisms were quite limited). The systems were proprietary; each computer manufacturer supplied an operating system for their computers, but these different operating systems were quite different in their styles of JCL and the facilities that they provided. Such differences hindered those users who had to work with more than one kind of computer.

Further the code for these OSs was often poor. Most of the systems had not been designed, they had just grown. More and more features would have been added to an original simple multiprogrammed OS. Consequently, the systems were difficult to maintain.

Modern operating systems started to appear in the 1970s. Their designers had different objectives. With hardware costs declining, there was no longer a need to try to maximize machine usage. Some computer power could be "wasted" if this made the system easier to use and so made the users more productive. Unix was one of the first of the modern systems; its designers sought to create a system that would make programmers more productive. Later systems, like the Macintosh OS, have focussed on the needs of other less sophisticated users.

3.4.1 UNIX

Unix started to be developed around 1969, with the first published description appearing in 1973. It was developed at an ATT research lab. and was made freely available to universities, encouraging its widespread adoption.

Unlike earlier operating systems which had been written in assembly language, the code for Unix was largely in C. The use of a high level language made the code much easier for programmers to understand and maintain. Further, the Unix OS was designed! The programmers who developed Unix started with a clear idea of how their OS was to work and what services it was to provide.

Unix was more limited in its aims than many other OSs of the time. It was intended solely to provide a good environment for timeshare style program development. Other OSs were attempting to do timesharing, and database transactions, and run large jobs, and ...; but such different uses of a computer tend to conflict resulting in poor performance in all areas.

The design for Unix modelled the OS in terms of several layers (when describing the design, someone made an analogy with a nut or an onion and introduced terms like "kernel", "shell" etc --- these names have stuck):

- the innermost layer (the "kernel") has the code for the i/o handling routines ("device drivers") etc;
- another layer contains the code for process management, file management, and memory management;
- further layers contain code for looking after wide area and local networks and so forth;
- the next layer out comprised large numbers of useful utility programs – programs for copying files, comparing files to find differences etc
- the outermost layer (the "shell") was the job control language interpreter, but this JCL interpreter was much more flexible than any that had been proposed previously.

Unix was originally written for a particular kind of computer (the "PDP11/20") manufactured by Digital Equipment Corporation (DEC). But, the relatively clean design of the system, and the use of a high level language, made it possible for the system to be adapted to other computers (only the "device drivers" and other really low level code had to be redone). Unix was moved to related but more powerful computer architectures (DEC's VAX series of computers) and to totally different computer architectures. During the 1980s, Unix was adapted to run on computers

as diverse as the modern Cray supercomputers down to personal computers with Intel-80386 CPU chips.

The US Department of Defence's Advanced Research Projects Agency (ARPA) sponsored development of Unix at the University of Berkeley. The Berkeley developers added features to support virtual memory and networking (both wide area and local networking). Late in the 1980s, many computer companies, ATT, Berkeley University, IEEE, etc got together and established standards for all Unix systems.

Unix thus has the advantage of being a system that is non-proprietary, is widely available, and is effective in its original role of supporting program development. Most students continuing with computing studies will eventually get to work with some Unix systems. Modern Unix systems have been expanded so as to handle tasks other than the "programmers' workbench" of the original design. These extensions (to handle large databases, some transaction processing and so forth) were demanded by customers. In some respects, these extensions detract from Unix which no longer has a quite the simplicity and elegance of its early forms.

3.4.2 Macintosh OS

The Macintosh OS (1984), and things like Windows 3 (late 1980s), represent more modern operating systems, having evolved in the ten to fifteen years after the start of Unix.

The important ideas in the Mac OS (and later systems meant to work in similar style) were developed at Xerox's Palo Alto Research Centre during the 1970s and early 1980s (Apple started the Mac OS by getting a licence to use Xerox's ideas).

Starting around 1972, Xerox PARC had had a project that aimed to explore what the "office of the future" would be like. Obviously, the office workers were going to make heavy use of computers. The Xerox researchers realized that the old systems were inappropriate.

The old systems had the computer as sort of oracle, surrounded by priests (the system's programmers and system's administrators) and neophytes (the computer operators); even the newer Unix systems had to have "system's gurus" to attend them and keep users at bay.

In an "office of the future", individual's would have their own computers, and these therefore would have to have operating systems that did not need priestly ministrations from gurus or others.

Unlike other developers of that period, the Xerox group realized that the cost of CPU power was going to drop dramatically. Consequently, it wasn't going to be important to keep the CPU efficiently employed, what was going to be important was the efficient use of time of the office workers. So it was going to be worthwhile "wasting" CPU cycles with the computer doing extra work if this would simplify the tasks of the user.

Given these premises, the Xerox group focussed on what they thought would be the needs of users; they identified factors such as:

- visual displays for "high-bandwidth" communication (show the user what programs and files are available for use etc);
- direct manipulation (use of mouse pointer, selection of object represented visually [as an "icon"], picking a command from a menu – the "point-and-click" interface rather than the "remember-and-type-command" interface of Unix and older systems);
- consistency (every program working in a similar manner);
- intercommunication (e.g. easy transfer of pictures, text and other data between programs).

Xerox developed a variety of experimental systems embodying the features that they felt would empower users and make computers more useful. However, Xerox never really brought these experimental systems to the level of practical, affordable products.

Steve Jobs and others at Apple in the early 1980s recognized the importance of the Xerox ideas and worked to make them practical. The Mac OS of 1984 was the first system that could really deliver computer power to all users.

