OPERATING SYSTEMS

1) INTRODUCTION

There are two chief design objectives in the design of an O/S:
- to transform the hardware into a 'virtual machine' which is amenable to the user
- to make best possible use of the available hardware.

There are two areas of application of O/S. These are real-time systems, and job-shop systems. A real-time system is one in which a response must be given to an external stimulus within a limited amount of time. Examples are control systems (eg nuclear reactor), file management and data base systems and transaction processing systems (eg airline reservations). Job-shop systems processes a sequence of tasks with no particular time constraints except those of overall throughput. Because of the variety of jobs which may be submitted, a job-shop O/S must support a large number of general utilities such as compilers and editors.

There are two types of job-shop systems: batch and multi-access. In a batch system, once the job enters the computer the user has no further contact with it until it has been completed. In a multi-access system the user may use a terminal to monitor and control the job as it runs. Multi-access systems are half way between real-time and job-shop.

Both real-time and job-shop systems may be implemented on either a single computer or a number of connected computers. In the latter case, the O/S must also ensure that the machines work together effectively.

2) FUNCTIONS & CHARACTERISTICS OF A JOB-SHOP O/S

2.1) FUNCTIONS

An operating system should be able to perform at least the following function:

- job sequencing
- job control language interpretation (user language)
- error handling
- I/O handling
- interrupt handling
- scheduling
- resource control
- memory protection
- user multi access

Furthermore, the operator should be able to run the O/S easily, and the ~~com~~ O/S should also perform computing of accounting.

2.2) CHARACTERISTICS

2.2.1) CONCURRENCY

This is the existence of several simultaneous or parallel activities. It raises problems of switching between activities, protecting activities from each other's effects, and synchronising activities which are mutually dependent.

2.2.2) SHARING

Concurrent activities may be required to share resources or information (eg library routines, databases, compilers). This raises problems of resource allocation, mutual exclusion, and memory protection.

2.2.3) LONG TERM STORAGE

The problems this raises are those of providing easy access, protection against interference and protection against system failure.

2.2.4) NONDETERMINACY

The O/S must be determinate in that the same program executed at two different times with the same data should provide the same result. It must be indeterminate in that it must respond to events which will occur in an unpredictable order. The system should thus be able to handle any sequence of events.

2.3) DESIRABLE FEATURES

2.3.1) EFFICIENCY

2.3.2) RELIABILITY

2.3.3) MAINTAINABILITY

2.3.4) SMALL SIZE

3) CONCURRENT PROCESSES.

The processes within a computing system must co-operate to achieve the desired object of running user jobs, although they are in competition for the use of limited resources. The the element of co-operation and competition imply the need for some form of communication between processes.

3.1) COMMUNICATION BETWEEN PROCESSES.

The areas in which communication is essential may be categorised as follows.

3.1.1) MUTUAL EXCLUSION

System resources may be classified as shareable or non-shareable (restricted to one process at a time). A resource may be non-shareable because of its physical nature, or because processes would interfere with one another if the resource was shared. For example, a ROM is shareable, whereas a card reader is not. The mutual exclusion problem is that of ensuring that non-shareable resources are accessed by only one process at a time.

3.1.2) SYNCHRONISATION

Generally, the speed of one process relative to another is unpredictable. We say that processes run

asynchronously with respect to each other. However, to achieve successful cooperation there are certain points at which processes must synchronise their activities. It is the O/S's task to provide synchronisation mechanisms.

3.1.3) DEADLOCK

When several processes compete for resources it is possible for a situation to arise in which no process can continue because the resources each one requires are held by another. This is called deadlock. The avoidance of deadlock, or the limitation of its effects, must be carried out by the O/S.

3.2) SEMAPHORES

Inter-process communication is generally effected by semaphores and the primitive operations WAIT and SIGNAL which operate on them. A semaphore is a non-negative integer which, apart from initialisation of its value, can only be operated on by WAIT and SIGNAL. SIGNAL(s) increments the value of s by 1 (signals are mutually exclusive). WAIT(s) decrements the value of the semaphore s by 1 as soon as the result would be non-negative. Thus if s = 0, WAIT(s) cannot be carried out until some other process increases the value of s.

Semaphores can be used to solve mutual exclusion, synchronisation and deadlock problems, as follows:

3.2.1) MUTUAL EXCLUSION

Non-shareable resources can be protected from simultaneous access by several processes by preventing the processes from concurrently executing the pieces of program through which access is made. These sections of program are called critical sections, and mutex in the use of resources can be regarded as mutex in the execution of critical sections.

Mutex can be achieved by enclosing each critical section by wait and signal operations on a single semaphore whose initial value is 1. Thus each critical section is coded as:

```
wait (mutex);
    critical section;
signal (mutex);
```

3.2.2) SYNCHRONISATION

The simplest form of synchronisation is that a process A should not proceed beyond a point L1 until some other process B has reached L2. The synchronisation can be programmed as follows:

```
Process A                    Process B
    :                            :
L1: wait (proceed);          L2: signal (proceed);
    :                            :
```

where proceed is a semaphore with initial value 0.

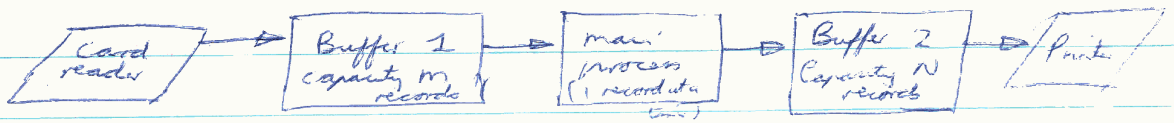
3.2.3) DEADLOCK

Deadlock can occur when processes are all waiting for signals, and are thus unable to proceed beyond the WAIT's to the point where they will SIGNAL. This can be likened to 'sharing' semaphores - if the value of a semaphore is 1, it is unshareable, and attempting to share it may lead to deadlock.

3.3 SOLVING PROBLEMS WITH SEMAPHORES.

In this section, an example of a typical problem requiring solution will be given.

Say we have a system where we wish to read, process and print records, and our system consists of a card reader, printer, and main processor. In between these we have buffers. Our system is thus:



We wish to prevent reading from an empty buffer, prevent writing to a full buffer, and avoid reading and writing to a buffer at the same time. We then require three semaphores for each buffer, the spaces available ($S_1 = m$, $S_2 = N$), the items in the buffers ($I_1 = 0$, $I_2 = 0$) and mutual exclusion semaphores ($m_1 = m_2 = 1$). Each process (reading, processing and printing) can then be done as follows:

Reading a record and depositing in buffer 1

read a record
WAIT (S1) (we do not want to put buffer
WAIT (M1) (mutual exclusion
deposit record in buffer 1
SIGNAL (M1) (end of mutex
SIGNAL (I1) (increment no of items in buffer

Read record from buffer 1, process and deposit - buffer 2

WAIT (I1)
WAIT (M1)
extract record from buffer 1
SIGNAL (M1)
SIGNAL (S1)
process the record
WAIT (S2)
WAIT (M2)
deposit record in buffer 2
SIGNAL (M2)
SIGNAL (I2)

Extract record from buffer 2 and print it

WAIT (I2)
WAIT (M2)
extract record from buffer 2
SIGNAL (M2)
SIGNAL (S2)
print the record

4) THE SYSTEM NUCLEUS

The provides the major interface between the hardware and O/S. Its purpose is to provide an environment where processes can exist - this means handling interrupts, switching processors between processes, and implementing methods for inter-process communication.

4.1) ESSENTIAL HARDWARE FACILITIES

4.1.1) INTERRUPT MECHANISM

The machine should provide an interrupt mechanism which at least saves the value of the program counter and transfers control to a fixed location in memory. This location is the start of the interrupt handler whose purpose is to determine the source of the interrupt and respond to it in an appropriate manner.

4.1.2) MEMORY PROTECTION

Concurrent processes must be protected from corrupting each other memory space. This protection mechanism must be built into the memory addressing hardware.

4.1.3) PRIVILEGED INSTRUCTION SET

In order that concurrent processes cannot interfere with one another, part of the instruction set of the computer

must be carried for use by the O/S only. These privileged instruction perform such functions as:

- enabling and disabling interrupts
- switching a processor between processes
- accessing registers used by the memory protection hardware
- performing I/O
- halting a processor.

These privileged instructions are allowed during system mode but not user mode. The control passing to system mode is made automatically when one of the following occurs:

- a system call is made
- an interrupt occurs
- an error occurs
- an attempt is made to execute a privileged instruction while in user mode

The control from system mode back to user mode is effected by a privileged instruction.

4.1.4) REAL TIME CLOCK

A hardware clock that interrupts at fixed intervals of real time is essential for some scheduling policies and for accounting purposes.

4.2) OUTLINE OF THE NUCLEUS

There are three main pieces of program in the system nucleus. These are the first-level interrupt handler, the dispatcher, and the inter-process communication primitives WAIT and SIGNAL.

4.3) REPRESENTATION OF PROCESSES

Each process can be represented by a process descriptor or state vector which is an area of memory containing all relevant information about the process. This includes the process name and status. The principal status is either running (it is executing on a processor), runnable (it could run if a processor were allocated to it) or unrunnable (it could not run even if a processor were allocated to it). A process may be unrunnable because it is waiting for I/O, etc. The status of a process is essential information for the dispatcher when it comes to allocating processors. A process which is running on a processor is called the current process for that processor.

A further part of the process descriptor can be used to store the processes volatile environment, that is, the subset of modifiable shared facilities of the system which are accessible to the process.

The process descriptor of each process is linked into a process structure, which acts as a description of all processes within the system. There is also a central table

which serves as a means of access to all system structures. This contains pointers to each data structure, and may also hold global information such as the time and date.

4.4) THE FIRST-LEVEL INTERRUPT HANDLER

This is responsible for responding to external signals (interrupts) and internal signals (error traps and extracodes). We shall refer to both of these as interrupts, the former external and the latter internal. The function of the FLIH is to determine the source of and service interrupts.

The determination of the source of interrupts is hardware dependent, but is usually done by flag examination. It may sometimes be necessary to interrupt the FLIH itself. Depending on the system, this will either not be allowed (and all such interrupts will be held pending) or a form of prioritised interrupts may be implemented.

4.5) THE DISPATCHER (LOW-LEVEL SCHEDULER)

This is responsible for allocating the processors amongst the processes. It is entered whenever a current process cannot continue, or the processor may be better employed elsewhere (i.e., after an interrupt which alters the status of some process).

The operation is as follows:

- is the current process on this processor still the most suitable to run ? If so, resume execution, else:
- store the process' volatile environment in it process descriptor
- retrieve the v.e. of the most suitable process
- transfer control to this new process

The determination of suitability is done by priorities. These priorities are assigned by the high level scheduler and are given a priori to the dispatcher.

It is sometimes possible that there are fewer processes than processors. It is then useful to introduce an extra process, the null process, which has the lowest priority and is always runnable. The null process may be an idle loop, but can be used for more useful functions.

4.6) THE IMPLEMENTATION OF SEMAPHORES

The semaphore operations are included in the nucleus because:

- they must be available to all processes, and hence must be implemented at a low level.
- the wait operation may result in a process being blocked, causing an entry to the dispatcher to reallocate the processor, hence it must have access to the dispatcher.
- a convenient way for an interrupt routine to make runnable a process is to execute signal on a semaphore on which the process has executed wait, hence signal must have access to the interrupt routines.

The operations we wish to implement are wait and signal

4.6.1 SEMAPHORE STRUCTURE

The semaphore needs three components:
- an integer component (s)
- a pointer to its queue
- a queue organisation element.

The queue is required for blocking and unblocking processes. Every unsuccessful wait causes the process to be added to the queue and made unrunnable. Similarly each signal allows a process to be removed from the queue and made runnable.

Different semaphores may require different queue structures (although usually FIFO is sufficient). The queue organisation element is either a coded description of the queue org. or else it may be a pointer to a routine which performs queuing and dequeuing operations. Thus our semaphores appear as:

INTEGER (s)
Q-POINTER
Q-ORGANISATION

4.6.2 WAIT AND SIGNAL IMPLEMENTATION

Both wait and signal may alter the status of a process, so an exit must be made to the dispatcher for a

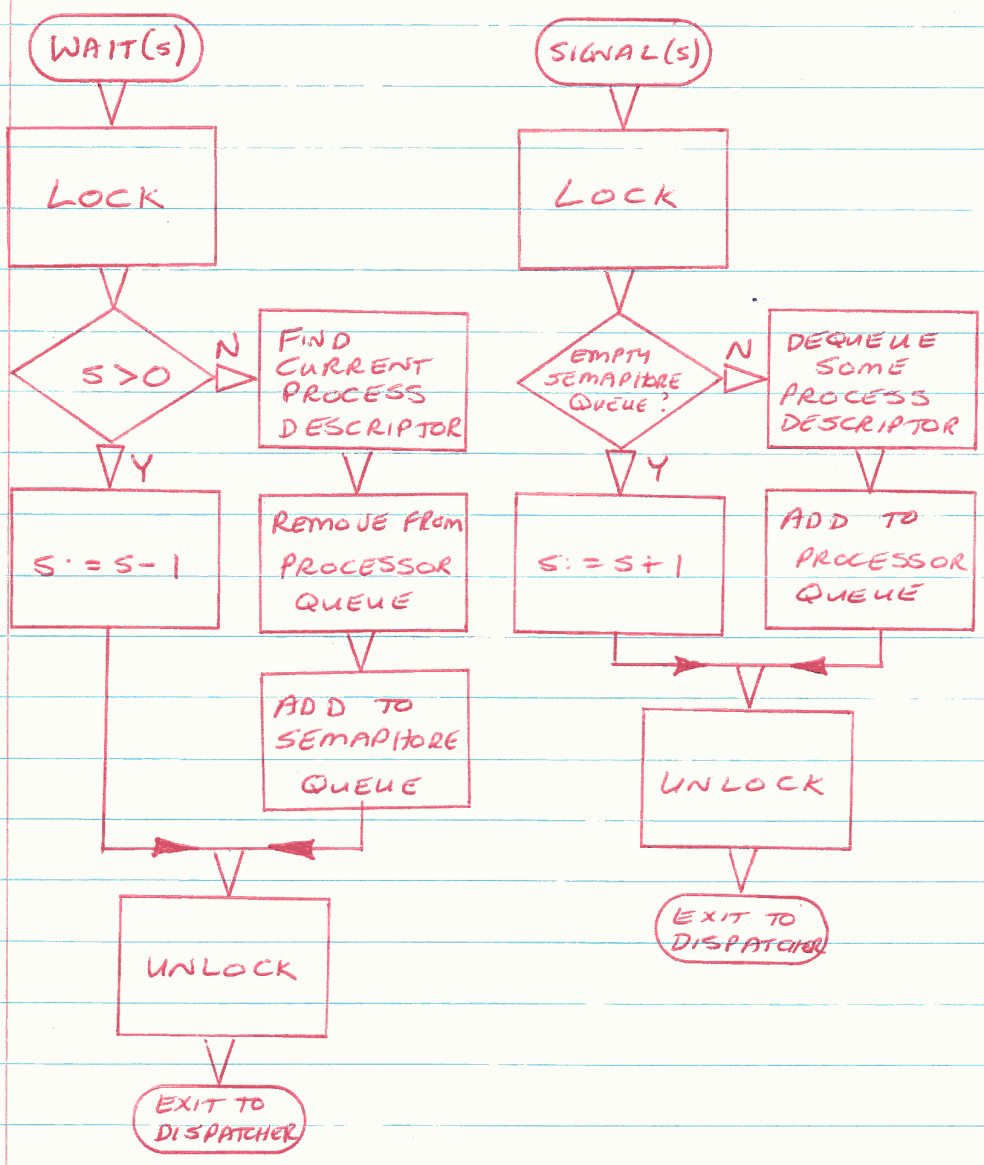
decision on which process to run next. If no change has occurred, the current process will be resumed. Some implementations only exit to the dispatcher when a status change has occurred, an increase in both efficiency and complexity.

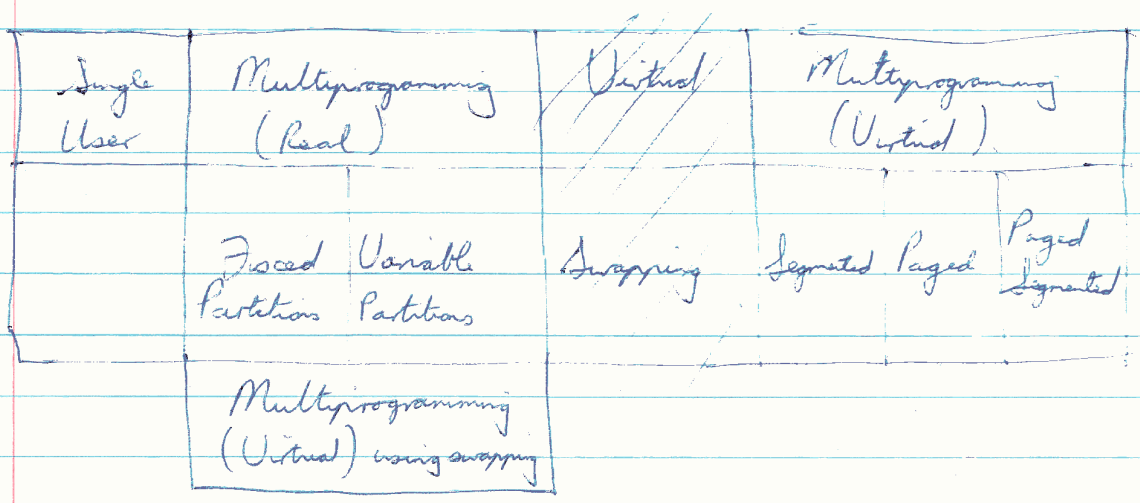
Also, wait and signal must be indivisible operations. On a single-processor machine, this may be done by disabling the interrupts (lock) and re-enabling them (unlock). On multi-processor machines, a flag method can be used.

We can summarise the lock and unlock operations as follows:

	Wait & Signal	Lock & Unlock
Purpose	General process synchronisation.	Mutual exclusion of processes from wait and signal procedures
Level	Software	Hardware
Delay	Queue	Busy/waiting flag interrupt inhibition
Signal delay time	Several seconds	Several microseconds

The actual structure of the implementations is given overleaf.


WAIT(s)
LOCK
S>0
N
Y
FIND CURRENT PROCESS DESCRIPTOR
S := S-1
REMOVE FROM PROCESSOR QUEUE
ADD TO SEMAPHORE QUEUE
UNLOCK
EXIT TO DISPATCHER
SIGNAL(s)
LOCK
EMPTY SEMAPHORE QUEUE?
N
Y
DEQUEUE SOME PROCESS DESCRIPTOR
S := S+1
ADD TO PROCESSOR QUEUE
UNLOCK
EXIT TO DISPATCHER


Single User
Multiprogramming (Real)
Fixed Partitions
Variable Partitions
Multiprogramming (Virtual) using swapping
Virtual
Swapping
Multiprogramming (Virtual)
Segmented
Paged
Paged Segmented

5 MEMORY MANAGEMENT

Ref: Memory management is the policies and mechanisms governing the organisation and allocation of main memory.

In single user systems, this entails protecting the O/S from being accessed and preventing program overflow. The former can be done by comparing each instructions addresses with a boundary register in the control unit. Program overflows can be done by overlays. Using this, the program is split into sections, called phases, which are compiled separately. A root phase driver calls the various sections via the linker/loader and allocates memory for global data, etc. Unfortunately this system is complex and requires care when modifying programs.

In multi programming systems, mem manage design considerations are

- organisation (paged, fixed partitions, variable partitions)
- program division (none, logical segmentation, fixed, pages)
- storage hierarchy (real, real & virtual memory)
- strategies:
  - fetch (demand, anticipatory)
  - place (where to put the page)
  - replace (which page to overwrite)
- relocation
- protection

5.1) PARTITIONED SYSTEMS

MM Organisation:	Fixed partition (eg IBM DOS)	Variable partition (eg IBM OS/MFT)
Program division:	none (whole contiguous prog)	none (whole contiguous prog)
Hierarchy:	real memory only	real memory only
Strategies:		
Fetch:	Whole prog loaded according to job scheduler	Whole prog loaded when scheduled allocated sufficient memory.
Place:	Entered into appropriate slot according to job class	Best fit, first fit, etc.
Replace:	Only on job completion	Only on job completion.
Pros:	Simple multiprogramming	More flexible scheduling No size restriction
Cons:	Wastage by partitions Inflexible scheduling by class Restriction on program sizes	Fragmentation.

To overcome fragmentation, the holes should be coalesced where possible; a better strategy is to use dynamic relocation to compact (pack) memory.

5.2) PAGED SYSTEMS.

The philosophy of paged systems is to allow the user a virtual memory which is larger than the available physical memory, thus eliminating the need for overlays. Secondary memory is made to appear as an extension of main memory (one-level store).

The virtual address space is divided into pages of equal size (usually about 512 words), and main memory is divided into page frames of the same size. Any process usually has a few active pages in m.m and the remaining inactive pages rest in secondary mem. Thus the paging mechanism has two functions:

- to perform the address mapping operation (mapping program addresses onto page frames)
- to swap pages in and out of memory.

5.2.1) PAGE ADDRESS MAPPING

The high order bits are interpreted as the page number and the low order bits the word within the page. For example, with a page size of 512 words (2^9) the lower 9 bits represent the word, and the rest the page.

The map from page & word numbers to physical addresses is done by a page table, the p^{th} entry of which contains the location p' of the page frame containing page number p, ie:

$$f(p, w) = p' + w \quad (w = \text{word number})$$

We also have the relations :

$p := INT(a/z)$
$w := MOD(a, z)$

{ a = prog. address
{ z = page size.

5.2.2) PAGE SWAPPING AND THE PAGE TABLE

It is quite possible for a program address to refer to a page not held in m.m. In this case the corresponding entry in the page table will be empty, and a 'page fault' interrupt is given if an attempt is made to access it. The interrupt causes the paging mechanism to initiate the transfer of the page into m.m. and update the page table, with the process made unrunnable until the transfer is complete. Alternatively, the address of the page in secondary memory can be held in the page table, and a 'presence' bit can be added to each table entry.

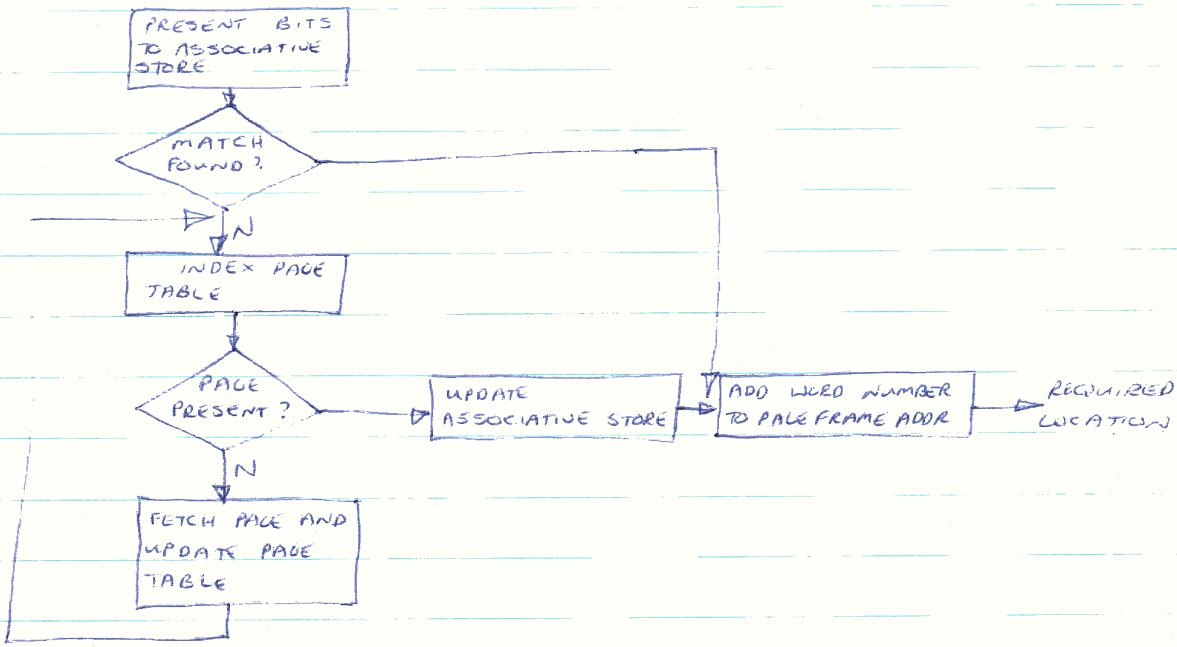
The pages to be swapped out when m.m. is full are determined by a page turning algorithm. The information required for this can be held in the page table. This might be:

- how many times page has been referenced
- time of last reference
- whether the page has been written to or not.

The page turning algorithm, and page table updating are performed by software, but the address mapping operation is implemented in hardware.

The problem with this method is that every ~~address~~ instruction involves both its own memory access, plus accessing the page table. This can be overcome by using an associative store, with a set of page address registers (PARs) each of which contains the page number of an active page. These can be searched simultaneously, thus reducing the time overhead considerably. As each active page must have a PAR, there should be as many PARs as page frames. On large machines this is not always possible.

On large machines, a small associative store can be used holding PARs of the most recently active processes. Its operation would be:



The PARs can be extended to include process identification as well as the page bits, like a single bit flag

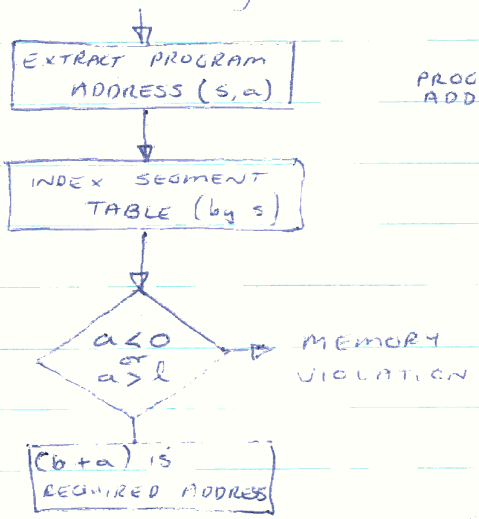
can be used to indicate whether the page belongs to the current process or not (resulting in a separate associative store for each processor).

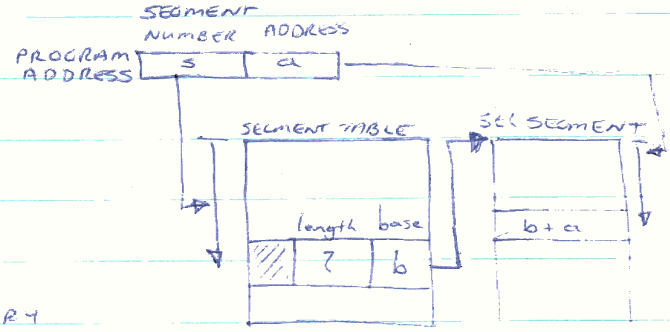
The size of the page tables can be ~~used by~~ reduced to the number of pages actually used by using hashed ~~addressing~~ access, although this increases the access time. One other item which can be included is a bit to indicate whether a page has been written to or not. If it has and it is replaced, it must be written back into secondary memory, else it may simply be erased.

5.3) SEGMENTED SYSTEMS

The organisation of the address space to reflect logical divisions in program and data can be achieved by segmentation. This can be simply done by adding several pairs of base and limit registers to each processor to demarcate the address space areas.

Alternatively, a more flexible arrangement is to allow the programmer to assign names to segments ('files'), and have a segment table accessed by segment number or name. The mapping operation is:





Among the information held in the segment table are access modes, checksums etc.

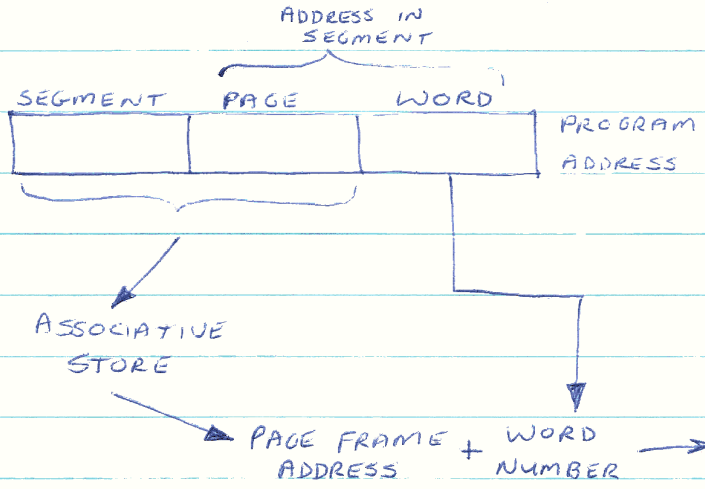
5.4) PAGED - SEGMENTED SYSTEMS

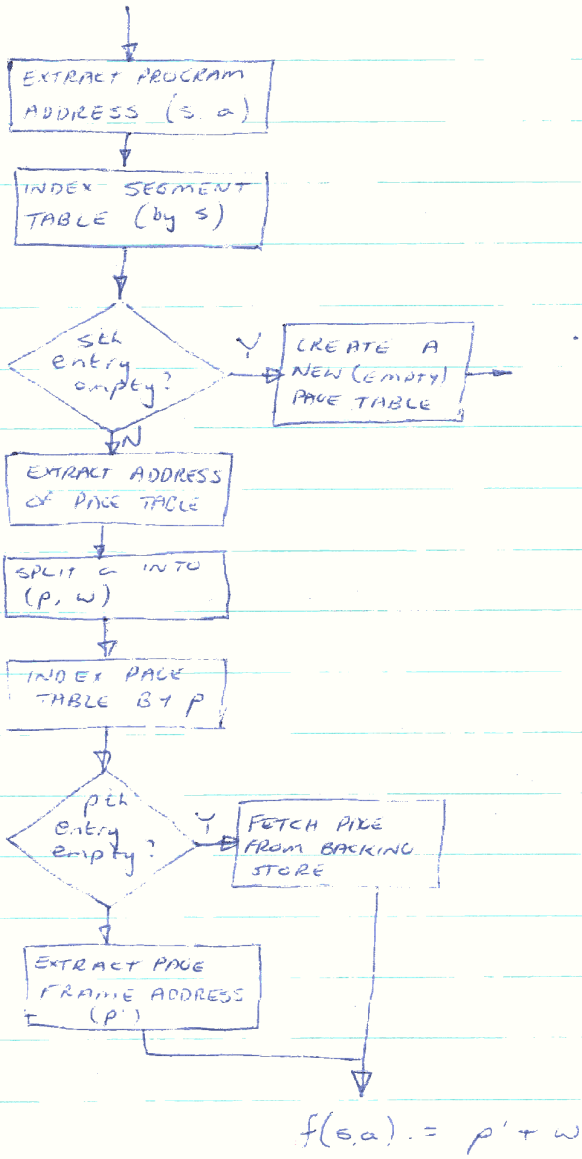
The address maps for paging and for segmentation are different in the following respects:

- the purpose of segmentation is logical division; of paging the physical division of memory to implement a one-level store.
- pages are fixed size, segments variable
- the addressing in paged system allows page overflow at hardware level, whereas segmentation must be under software control.

For large applications, it may be impossible to hold all segments in memory, but this can be solved by using either paging or swapping (cf § 5.5)

In paged-segmentation, each segment has its own page table. The address mapping operation is schematicised opposite. Associative stores can again be used, containing both the segment and the page numbers, as below.





This has two advantages: the entire segment need not be present in main memory, and the pages of a segment need not be contiguous. This eases placement and improves efficiency by removing rarely used code (initialisation procedures, etc)

~~5.5 ALLOCATION POLICIES~~

5.5 ALLOCATION POLICIES

5.5.1. PARTITIONED SYSTEMS.

§ 5.1 discusses policies for non-virtual partitioned systems. For virtual machines, swapping is used. The policies are:

Placement: Wherever suitable space exists, preferably with I (instruction) and D (data) banks in separate partitions to allow concurrent access.

Replacement: Programs may be swapped out in favour of others with higher core priority when
- they go into long wait states
- they exceed their time-slice (although a minimum time is usually also used to prevent too much swapping).

Fetching - According to core priority, possibly replacing a lower priority swappable run.

This is typical of multi-access systems of the 60's. A possible implementation could be:

~~Types~~
1. Real time
2. Non-resident O/S routines
3. Demand
4. Batch

The demand runs are initially at level 3. On the expiry of its time slice, they are marked as swappable, dropped in priority by one level, they have their time slice doubled. On entering long wait states, they are marked as swappable, but retain their priority.

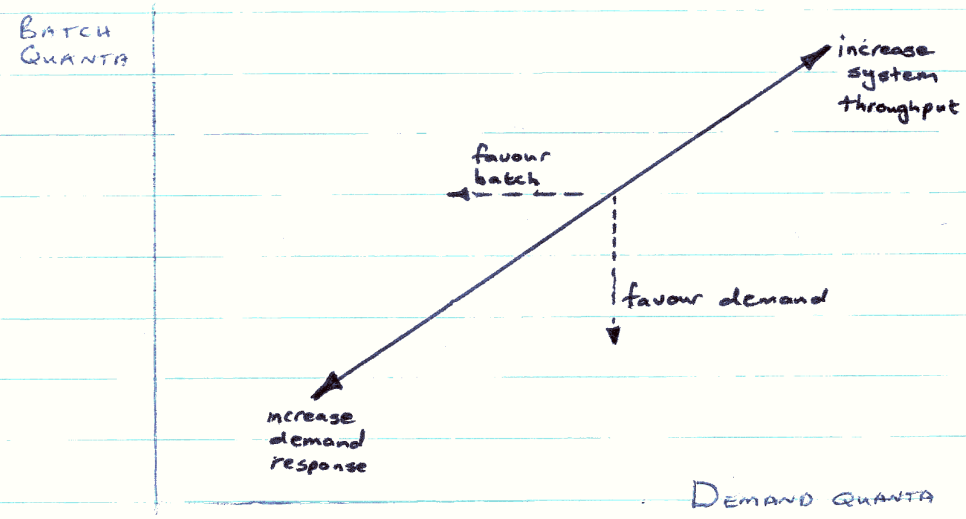
In batch, the level is determined by run card, and never changes. Batch runs are swappable only in favour of other types, to improve their throughput.

The system can be tuned for various possibilities as follows:

TO IMPROVE DEMAND RESPONSE TIME: reduce batch job time slices, or reduce demand job quanta to favour interactive runs even more.

TO INCREASE BATCH THROUGHPUT: increase the batch quantum, perhaps even to ∞

TO INCREASE DEMAND RUN THROUGHPUT: increase demand quanta.



5.5.2) PAGED SYSTEMS

PLACEMENT - This takes place in any free frame, thus is very simple. Fragmentation is not a problem as pages are of fixed size, although internal fragmentation (the wastage in each processes last page, ~ ½ page per process) does occur.

REPLACEMENT - Various policies are available, and will be covered in sub-sections.

5.5.2.1) OPTIMAL REPLACEMENT

This was proposed by Belady et al in 1969. It is not realisable in practice but provides bench mark of optimal performance.

5.5.2.2) RANDOM REPLACEMENT

This results in no overhead, although it is not very effective.

5.5.2.3) LEAST RECENTLY USED (L.R.U) REPLACEMENT.

At each reference the time must be saved, and, based on the assumption that future behaviour will closely follow past behaviour, the page which has the oldest time is replaced. The overhead is the time-stamping.

5.5.2.4) LEAST FREQUENTLY USED (LFU) REPLACEMENT

At each reference, a counter is incremented (the overhead). This has two drawbacks - it favours replacing pages which have just been loaded and avoids those which have been accessed many times, even though these may be initialisation routines and may not be called again. These can be solved by resetting the counters to zero periodically, and inhibiting the replacement of pages loaded recently.

5.5.2.5) FIFO REPLACEMENT

This entails replacing the page which has been resident longest. It is not as effective as LRU and LFU but has lower overhead (recording the sequence of page loads).

5.5.2.6) NOT USED RECENTLY (N.U.R.) REPLACEMENT

Each page has one bit set to 1 when used and 0 initially. Preference is then given to the page(s) which has been used but not recently. A limit check must also be made on unmodified list of pages to prevent them being kept too long.

5.5.3) SEGMENTED SYSTEMS

The policies included in this section can also be used for variable partitioned systems (base - limit pairs) As a result of the memory structure and fragmentation, it is necessary to keep a hole list, or list of all unallocated areas. If the segment to be placed is smaller than the hole used it is placed at the 'bottom end' of the hole to reduce fragmentation. We shall denote the size of the holes by $x_1, x_2, \ldots x_n$.

5.5.3.1) PLACEMENT

Best fit: The holes are listed in order of increasing size, and the smallest hole larger than the segment is used (i.e. x_i is used where $x_i \geq S > x_{i-1}$)

Worst fit: The holes are listed in descending order, the segment placed in the first (biggest) hole, and the remainder of the hole inserted back into the list at an appropriate point (assuming that the remainder of a big hole is more useful than the remainder of a small one)

First fit: The holes are listed in order of increasing base address, and the first one large enough for segment used. This leads, after a time, to an accumulation of small holes near the head of the list. (The list is useful for compacting (packing) memory)

Buddy: For this algorithm segment sizes must be powers of 2, with seperate lists maintained for each power. The list for the power just greater than the segment size

is then searched. If it is empty, the next biggest hole can be split in two, creating two 'buddies' to add to the list. Similarly, buddies can be coalesced into larger holes.

The problem of fragmentation can be overcome by compaction. This can be done only when necessary by referencing the hole table, or it can be done every time a segment is removed, avoiding fragmentation altogether and eliminating the need for a hole table.

5.3.3.2) REPLACEMENT.

Replacement in non-paged systems is similar to that for paged systems with the qualification that size must be taken into account. Possibly the simplest algorithm is to replace the single segment (or smallest set of contiguous segments) which, together with any holes adjacent will free enough space for the incoming segment. If there are several, a policy such as LRU can be used to choose between them, although this algorithm has many drawbacks, however, and replacement policies for non-paged systems tend to be very complex.

5.5.4) FETCH POLICIES (GENERAL)

There are two broad types: demand and anticipatory. Demand fetch blocks when they are needed; anticipatory policies fetch them in advance. Demand are the easiest to implement - a missing block fault generates a fetch request, and the placement/replacement policies allocate memory for the new block.

Anticipatory policies rely on predictions of future program behaviour in order to be fully effective. Prediction can be based on two things:

- the nature of construction of programs
- inference from a processes' past behaviour.

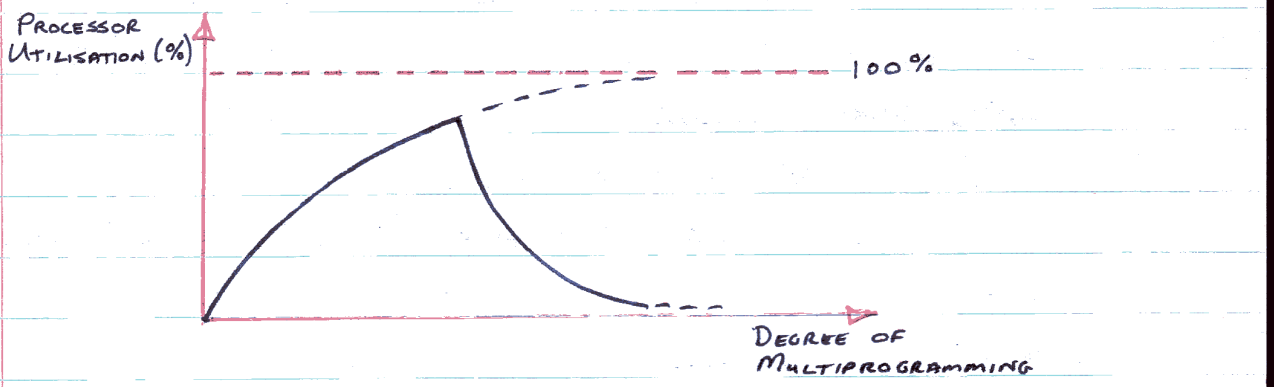
Many programs exhibit behaviour known as operating in context, ie, in any small time interval a program tends to operate within a particular logical module, drawing its instructions from a single procedure and its data from a single data space. This is ~~elongated~~ strengthened by the frequent occurrence of looping. This leads to the principle of locality: Program references tend to be grouped into small localities of address space, and these localities tend to change only intermittently.

The validity of this principle varies from program to program. It is used to formulate the concept of the working set model of program behaviour.

5.6) THE WORKING SET MODEL (follow by good)

This is an attempt to understand the peformance of paging system in a multiprogramming environment

Consider a single processor system with paged memory in which the degree of multiprogramming is steadily increased. As it rises, so does the processor utilisation, until a certain point which is dependent on the amount of memory available. Beyond this point there is an increase in ~~paging~~ swapping which results in a decrease in processor utilisation. This situation is called THRASHING



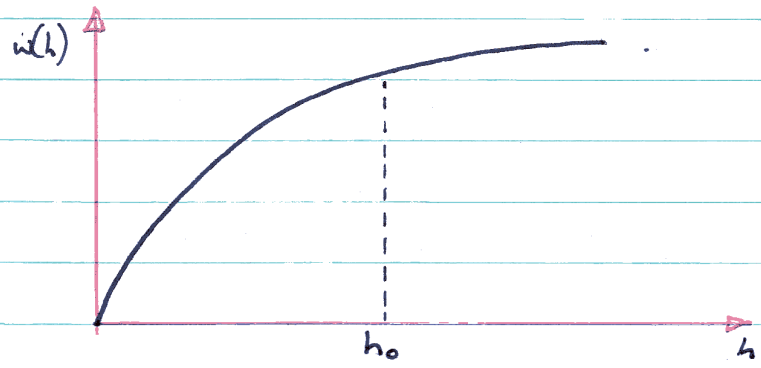
Thus each process requires a minimum number of pages, called its WORKING SET, to be held in MM in order to use the processor efficiently. For thrashing to be avoided the degree of multiprogramming must be no greater than the level at which the working sets of all processes can be held in main memory.

The working set of a process of a process at time t is defined to be:

$$w(t,h) = \{\text{page } i \mid \text{page } i \in N \text{ and page } i \text{ appears in the last } h \text{ references}\}$$

In other words, it is the set of pages which have recently been referred to, 'recentness' being represented by the parameter h.

Denning has shown that w(h) varies with h as shown:



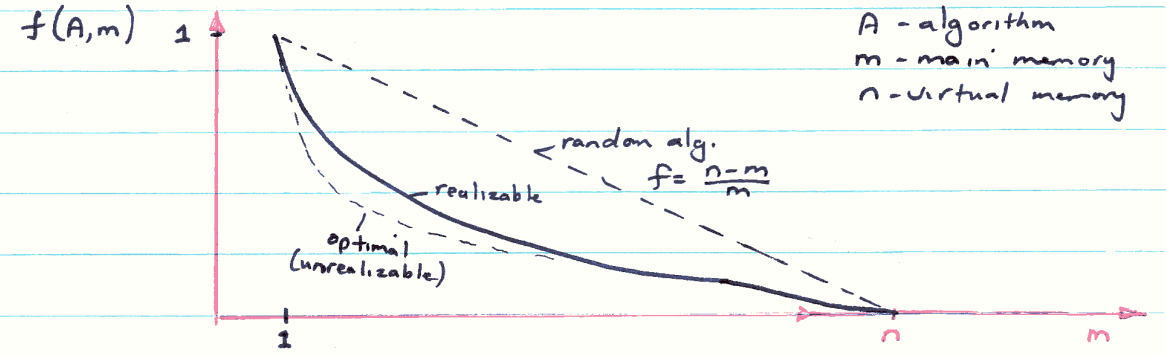
As can be seen, one can establish a reasonably small value for h (eg h_0) as a larger value of h will not significantly increase the working set size. The working set principle leads to the following rule:

'Run a process only if its entire working set is in MM, and never remove a page which is part of the working set of some process'.

Although the working set concept is rather arbitrarily defined, it does make a significant contribution to the prevention of thrashing.

5.7) GENERAL.

In order to evaluate the various algorithms, we may consider the page fault probability. It happens that all the algorithms give similar performance:



What does make a difference is the ratio of virtual to real memory. If this is greater than 2, performance decreases and f(n,m) increases. The choice of algorithm is fairly important but the ratio n:m is <u>critical</u>

There are also resident pages in memory which are marked as unswappable (eg the replacement algorithm itself) which must also be taken into account.

Whatever implementation we use, we must add to the volatile environment of each process either a copy of the base & limit registers (var. part) or a pointer to the segment table or a pointer to its page table. If we used a non-paged system we must also add the 'hole list' as a further data structure pointed to from the central table.

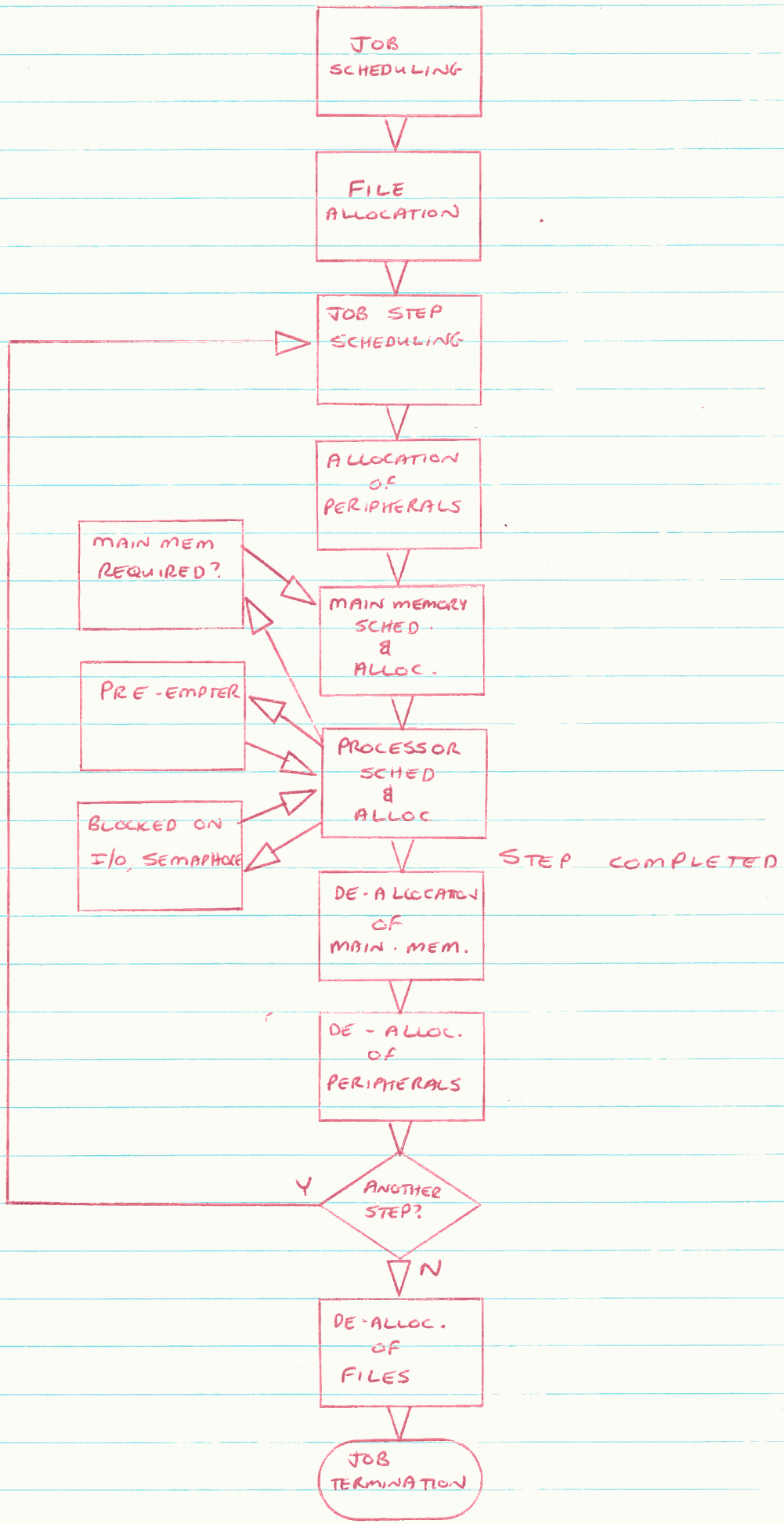
8) SCHEDULING & ALLOCATION

These are methods of sharing a limited set of resources among a set of claimants according to a set of externally determined rules and priorities. Scheduling is the process of deciding to which claimants resources should be allocated, and allocation is the granting of a resource to a claimant. Strictly speaking, scheduling is a part of allocation. The objectives of S & A are:

- throughput optimisation
- response time "
- deadlock prevention
- avoiding indefinite postponement of a process
- fairness between processes
- graceful degradation under heavy load
- mutual exclusion of unshareable resources.

A typical job progression is shown on the facing page. The job allocation is usually for the entire job to prevent intervening alteration by other processes. Peripheral allocation is done by step to prevent unnecessarily long holding periods. This can sometimes lead to delays between steps. Main memory requirements may also vary widely from step to step (except demand paging with global competition).

The two types of scheduling, job and processor, will be dealt with in separate sections.


JOB SCHEDULING
FILE ALLOCATION
JOB STEP SCHEDULING
ALLOCATION OF PERIPHERALS
MAIN MEM REQUIRED?
MAIN MEMORY SCHED. & ALLOC.
PRE-EMPTER
PROCESSOR SCHED & ALLOC
BLOCKED ON I/O, SEMAPHORE
STEP COMPLETED
DE-ALLOCATION OF MAIN. MEM.
DE-ALLOC. OF PERIPHERALS
Y
ANOTHER STEP?
N
DE-ALLOC. OF FILES
JOB TERMINATION

8.1) JOB SCHEDULING

This is the admission of new processes. It occurs according to predetermined limits, possibly for each class, or according to resource utilisation level (eg page fault rate) or when a demand user logs in. The choice of process is governed by combinations of:

- new step in existing job if possible
- next in queue (or class queue) ie FIFO
- shortest job first (batch)
- according to priority (priorities can be increased with time)
- one that fits current pattern of resource usage
- one approaching latest start time
- whether submitted by demand or batch (applies only to non-interactive work, ie, long executions)
- round robin (time sharing)

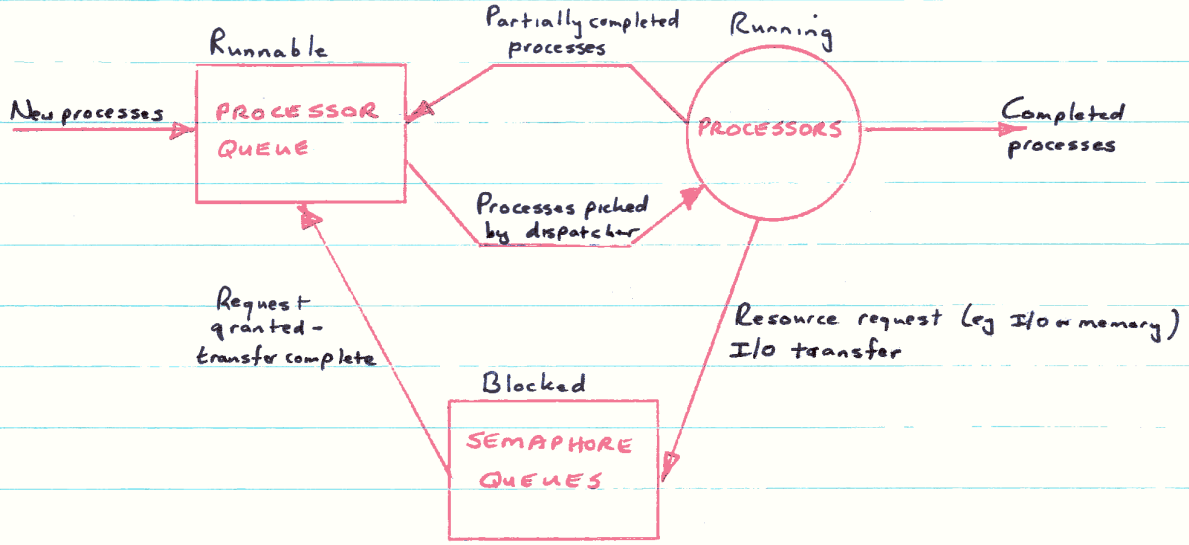
8.2) PROCESSOR SCHEDULING

The scheduler is called after every interrupt (as interrupts usually mean current running process has become unrunnable for some reason). It is usually done on a priority basis, with the scheduler itself having the highest priority. The priority of the other processes can be determined by various factors, such as:

- type (real time, demand, batch)
- external priority
- access to resources held
- interactive, I/O bound or CPU bound

- round robin within priority level.

In the last instance, the choice of time slice depends on which is more important - high throughput (large t-s) or fast response (small t-s). Various queues with different t-s's can be used, and long processes siphoned off to queues with higher t-s's. A general scheduling model is:



One of the aims of the scheduling algorithm could be to minimise the number of transitions from the running to the blocked state. As I/O transfers are outside its control, it should instead minimise the number of resource requests which are refused. Thus processes should only be introduced into the system if their likely resource demands can be satisfied by the resources currently available.

8.3) DEADLOCK

The necessary and sufficient conditions for deadlock are:

- the resources are unshareable
- the processes hold resources while waiting for new ones
- the resources cannot be pre-empted from the holding processes
- a circular chain of resource holdings and requests occurs.

There are three main strategies which can be used:

- prevention (by ensuring at least 1 condition never holds)
- detection and recovery
- avoidance (by anticipatory action).

As far as prevention is concerned, the first condition is unchangeable, and the third is often impossible or undesirable to change. The second condition may be avoided by an 'all or nothing' allocation, although this can be wasteful. The fourth condition can be avoided by imposing an order on the allocation (to force linear instead of circular form) although this can also be wasteful.

Detection is possible by maintaining a resource state graph. If deadlock occurs, recovery is possible by either aborting all the deadlocked processes (and returning to a checkpoint if possible) or by successive abortion or pre-emption.

For avoidance, the avoidance algorithm must be invoked prior to granting each request. The algorithm must thus

have prior knowledge of the pattern of requests, as deadlock may be several steps ahead but must be detected before it becomes inevitable.

One possible avoidance algorithm is Dijkstra's Banker's Algorithm. The prior knowledge required is the claim on each resource by each process. Requests are only granted if they are within the claim and a sequence exists in which all processes can run to completion. In most systems main memory and processors are adequate and deadlock only really occurs in I/O.

8.4) GENERAL ALLOCATION

Processor allocation is done by the dispatcher. The data describing the processor can be held in a processor descriptor which typically contains the processor id, current status (user / supervisor), a pointer to the current process descriptor, and if necessary, an indication of the processor characteristics (in systems with different types of processors).

Memory is allocated by the memory management section of the O/S. Page and segment requests are queued as I/O request blocks and serviced by the device handler for the appropriate secondary memory device.

Processes awaiting the allocation of a peripheral can be queued on a semaphore which is included in the peripheral

device descriptor. Mutual exclusion can be ensured by initialising the semaphore value to 1.

Backing store used as virtual memory is allocated by the memory management system, and that used as file space is allocated by the filing system.

8.5) GENERAL SCHEDULING

Although the scheduler is entered upon interrupts, it is usually only activated when a resource is requested or released, a process terminates, or a new job arrives in the job pool. When it has finished its work it suspends itself by executing a wait on a semaphore which is signalled whenever a scheduling event occurs. In an O/S which neither prevents nor avoids deadlock, the scheduler may never be called and thus the deadlock will not be detected. Thus it is usual that the scheduler is also called by a clock interrupt every few seconds.

The request and release of resources can be implemented by providing two O/S system procedures:

request (resource, result) release (resource)

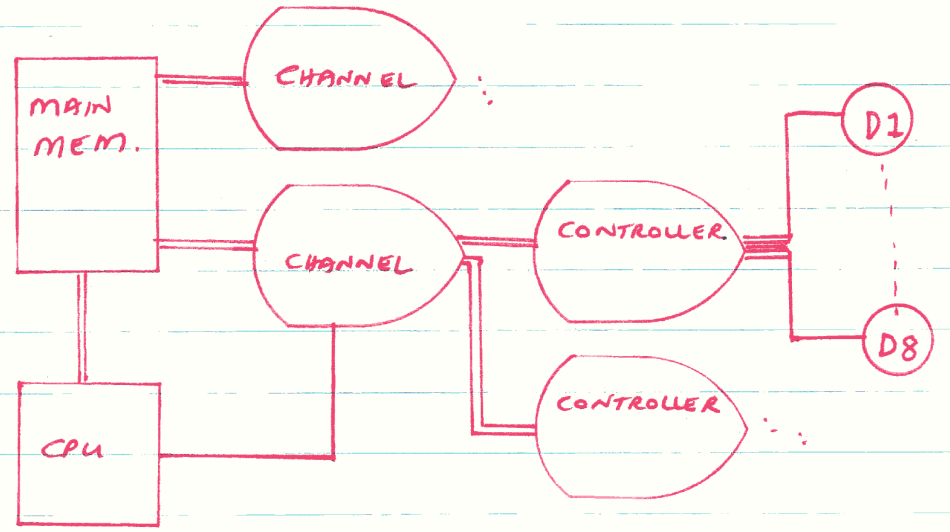
These place the necessary information about the resource and the identity of the calling process in a data area accessible to the scheduler, and then signal the scheduler semaphore. The second parameter of request is used to convey the result of the request.

Depending upon the system, these operations can be included as part of the scheduler or separately. Note that a request which is at first refused may be queued by the scheduler until the resource becomes available. The queuing and associated delay are transparent to the requesting process.

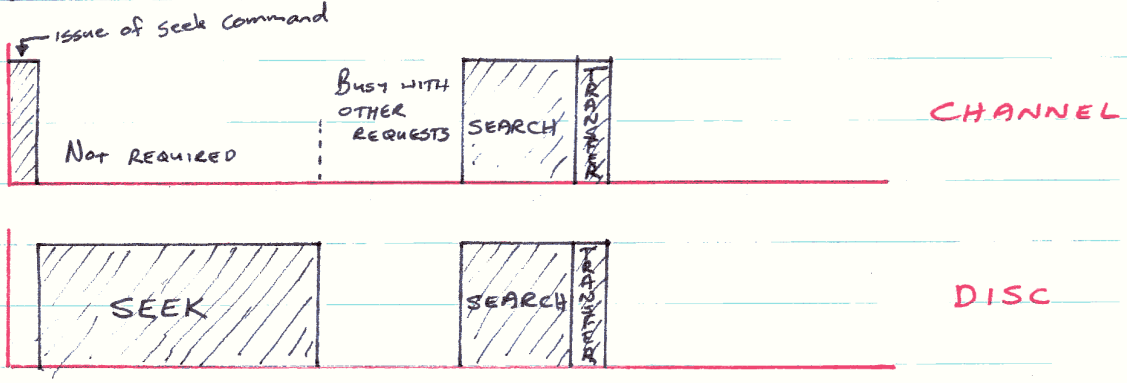
A few general criteria for assigning process priorities are:

- processes which possess a large number of resources may be given high priority in an attempt to speed their completion. This has the dangers that large jobs may monopolise the machine and that users may generate dummy resource requests, although this latter can be avoided by ensuring that jobs with small resource requirements (total) are given preference.
- processes which possess a large number of resources could have further requests granted whenever possible.
- memory allocation in paged systems can be performed according to the working set principle.
- O/S processes should have priorities which reflect their importance and urgency. Peripheral device handlers should have high priorities proportional to the device speeds. Similarly, spoolers should also have high priority.
- unless deadlock is being prevented, all resource requests should be subjected to an avoidance algorithm.
- the overhead in making scheduling decisions should not be out of proportion to the ensuing gains.

8.6) DISC SCHEDULING

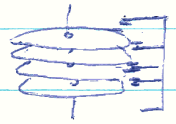


The channel is an I/O processor having 3 basic instructions - seek, search, and transfer (read/write). With each instruction there must be a MM address, a device address (disc, cylinder, head, block), a size, and a semaphore address (to signal upon completion). The use and channel utilization during a typical SST sequence is indicated schematically below:



The seek locates the specified cylinder, the search the specified block.

Disc storage devices usually have 10 to 20 platters with recording surfaces above and below each platter, except the extreme ones. The heads are close to the surfaces, brought into contact with spinning platters by air pressure. Each recording surface has 200 to 400 concentric tracks, all with the same capacity (inner tracks have denser storage). The transfer rate is constant (∝ rotation speed). The read/write arms 'comb' the surface. Addresses are specified by cylinder, head and block. (A cylinder is the set of all the tracks on all surfaces). A typical rotation speed is 3000 RPM, or 20 msec per revolution (causing a search latency time average of 10 msec). Transfer ~~rate~~ times depend on the quantity of data, typically ± 1 MB/sec.



Drums have 1 R/W head per track - their capacity is much smaller, but they have large rotation speed, and are useful for rapid transfers (e.g. paging & swapping)

Seeks require a fixed stabilisation time (for the heads to accelerate/decelerate) as well as a travel time ∝ the number of cylinders traversed. For a random seek pattern, the average number of cylinders traversed is about 1/3 of the total. Typical seeks are 15 msec + 0,25 msec/cyl.

In general, we attempt to minimise the mean seek time while keeping variance in seek time within tolerable limits. Various scheduling algorithms can be used:

FIFO (possibly with priority) - no optimisation, lowest throughput but small variance

SHORTEST SEEK FIRST - highest throughput but high variance as extreme cylinders get poor service

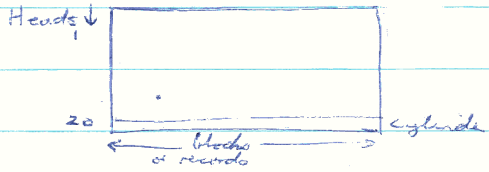
SCAN - this is an SSF but in a fixed direction or alternate directions. Unidirectional scans are also called C-SCAN. The throughput is lower than SSF, but variance is less, and this is even more so in C-SCAN

N-STEP - this is a C-SCAN but will only the existing requests at the beginning of each sweep being scheduled. Once again, throughput is lowered with a corresponding improvement (reduction) in variance.

Rotational optimisation (shortest latency time first) can be used to minimise search times for multiple requests on the same cylinder. A further improvement can be achieved by using rotational position sensing, which frees the channel during searches.

9) FILE STRUCTURES

9.1) SEQUENTIAL FILES



Access by <cyl, block>

No random access, but excellent sequential processing and economical in space. However, maintenance is complex, as we cannot insert without rewriting.

9.2) DIRECT ACCESS FILES

Access by some key.

Excellent random access, good sequential (but reads blank records)

Space is wasted, but maintenance is easy. Must choose key carefully.

Eg 15 000 employees, nos from 1 – 25 000 (allows for growth)

Assume 20 Records / track & 10 tracks / cylinder

with the file starting on cylinder 31 head 1

Cylinders required = $\frac{25\,000}{20 \times 10}$ = 125, & 31 – 156 inclusive

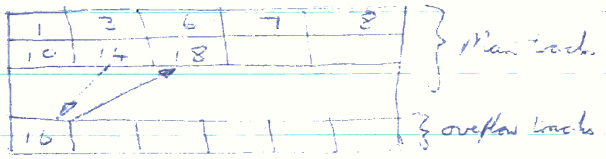
Calculate address for key 14237

Cyl = 31 + (14237 – 1) DIV 200 = 102

Head = 1 + (37 – 1) DIV 20 = 2

Rec = 17

9.3) INDEXED SEQUENTIAL FILES



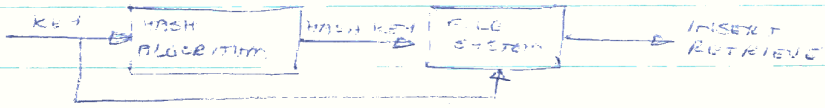
Each record has a pointer to cylinder & track address

Several random accesses give variable performance, depending on when last the file was reorganised.

Sequential access is good but slow for high overflow %.

About 20% of space must be allocated to overflow which can be wasted, and maintenance is complex, with periodic reorganisation required.

9.4) HASHED RANDOM FILES



Random access average 1.5 accesses/record, depending on space allocation and hashing method. Using hash keys in C-K-D (count-key-data) format performance is similar to direct.

Sequential access requires a prior sort.

About 50% access space must be allowed (although C-K-D allows no blocking so it has low space utilisation)

Periodic reordering to remove records flagged as deleted (and possibly sorting) is required.

9.5) COMBINATION STRUCTURES

Using a combination of methods, eg indexed unordered which is useful when space is at a premium. Many records are held unordered in minimal space, and the index file may be direct or random, on disc or in mm.