TOPS-20’s Virtual Memory

Jim Goodwin, December 15, 1979

(Peter Fritzson, October 27, 1988)

(This paper was written by Jim Goodwin for the course on operating systems for the D-line during the first term in 1979. It is almost completely applicable to the new implementation of virtual memory used by Sun-Unix 4.0, i.e. Unix V.4. Just switch the size of the blocks from 512 words to 8192 bytes, the addressable memory area from 256 Kwords to 4 Gbytes, the address from 18 to 32 bits. File mapping and buffering is dealt with in the same way. On the other hand, the Unix kernel is permanently allocated, whereas parts of the TOPS-20 kernel was not and could be swapped out.)

This paper discusses a simplified version of the TOPS-20 virtual memory (VM). First the design of the VM itself is described from the bottom up. Then various applications of the VM, such as the TOPS-20 implementation of serial IO, are discussed.

1. Hardware

Everything starts with the hardware. The original TOPS-20 system, then called TENEX, was a PDP-10 model KA with a small store of about 12 fast associative registers (PAR’s, see Lister) about 128K words (36 bit) of primary core memory, a drum, and a disk. (An identical disk may be seen at Lysator: the giant Bryant disk, a real mastodon.) The memories in this list are given in order of speed and cost, from fast and expensive to slow and cheap.

On later systems, there was no drum: disks got better and faster, and made drums obsolete. The term "secondary memory" refers to disk as well as to drum if there is one.

The main problem we want to solve is: how shall we organize all this memory? The main answer we will give is: in uniform units called "pages" of 512 words each.

At some fairly low level, secondary memory may consist of tracks, cylinders, sectors, bands or whatever but only the most low level parts of their device handlers will know anything about this. This permits the rest of the system to believe that secondary memory consist of one long vector of pages. The vector index of a particular page will be called its "secondary address". The only operations necessary will be to read or write one page at a given address. (For efficiency reasons real systems also allow an operation that reads or writes a series of pages to a series of consecutive locations in secondary memory.)

Primary memory consists of a vector of words. A word’s address is its physical position in that vector. The instruction set of the computer assumes that there are no more than 256K words (later models of PDP-10 had tricks to allow more, which we ignore here). 256K is 2 to the 18th power a word address is therefore 18 bits. We agree to break up the memory in pages of 512 words for compatibility with our similar agreement about secondary memory. The CPU knows nothing about pages and treats all of primary memory as a uniform vector of words. But the 18 bit word address used by the CPU may be regarded as two nine-bit addresses laid side by side: a 9 bit page number and a 9-bit word address within the page.)
There are now two kinds of page, primary pages and secondary pages. Both together might be called "real" pages or "physical" pages, as against the "virtual" pages to be defined next. Lister uses the term "page frame" for primary physical pages.

2. VM.5 - Half a Virtual Memory

The complete VM is built in two steps. The first provides swapping the second provides address translation as well. When both are accomplished, we have a virtual memory consisting of virtual pages that are swapped in and out of primary store automatically and invisibly, and which can be mapped into the address space of any process or file at any desired location. But to simplify the presentation, swapping is handled first. The system with swapping but without address translation (mapping) might be called VM.5, because it is half the whole VM. VM.5 consists of virtual pages which are swapped but cannot be mapped or addressed yet. Let us call these pages "swapped" pages, reserving the term "virtual pages" for pages in the full VM.

VM.5 is the core of the system and consists of the disk device handler, which can take a disk address and read or write the correct 512 words on the correct sector and track and the FLIH, which is necessary to handle the interrupts from the disk channel which drive the device handler and the semaphore implementation, which is needed so that the rest of the system can communicate with the disk driver.

Above VM.5, nothing in the system can refer to disk or drum pages, or to primary or secondary pages, or to pages that are in or out. The rest of the system deals only with swapped pages. It is a little sloppy to speak of input or output of swapped pages to disk, for example. Swapped pages don’t exist on disk or primary, they exist in a virtual world of their own, implemented by a constantly changing combination of physical pages of different types in different places. Each swapped page has a distinct identity, and may be referred to by many processes, as well as possibly one file, at the same time it is still one swapped page, shared by all of them.

One of the main design goals for VM.5 is that when several processes do refer to the same swapped page at once, then the system implements the swapped page by exactly one physical primary page. It would ruin everything if, during swapping, one physical secondary page were input twice, to two different physical primary pages, and then different processes given the different copies. That would cause one swapped page to become two, to lose its identity.

The swapper controls all this via two tables. The first has an entry for each primary physical page and may be called the PPM, Primary (physical) Page map. The second one has one entry for each swapped page, and is called the SPM, or Swapped Page Map. The limit on the number of swapped pages that the system can keep track of is in practice the maximum number of pages it can fit onto secondary, so the SPM has that number of entries. An index into this table is the unique identity of one swapped page.

The SPM contains the following information, among other things. First, the secondary address of the secondary page which represents this swapped page. Second, an indication whether the page is currently in primary, and if so, in what primary page.

The PPM has one entry for each primary physical page, containing among other information an indication whether that page is in use, and if so, which swapped page is in it, i.e. a pointer to the SPM entry for that secondary page.
We can now do the following things. When a swapped page is requested, we can tell from the SPM if it is already in or not and if so where, so that we will never create a second copy in primary. Given a primary page, we can tell what swapped page is in it, and from the SPM entry for that swapped page we can tell where to put it if we want to remove it.

The scheme with SPM and PPM is illustrated in my figure 1. In this figure, memories (vectors of pages of 512 words) are drawn wide, maps of memories (vectors of entries describing pages, usually 1 word in the actual system) are drawn narrow. Pointers that are explicitly stored in the maps are drawn as arrows. Note that the arrows between SPM and PPM are double-ended there is a pointer in each table entry to the other. There is no arrow from PPM to primary memory, since no pointers are stored in this case the Nth word of PPM always describes the Nth page of primary. A page of SPM which is not in primary is marked "out" instead of having an arrow to PPM.

This scheme is now made more efficient by some extensions.

We add a "dirty bit" to the PPM. This bit is one if the swapped page in that primary page frame has been modified since it came in. The dirty bit is for the removal algorithm. When a page is removed, if the dirty bit is not set, then the page does not need to be written back to secondary. The primary page frame can be used again immediately.

How do we guarantee that the dirty bit is always correct? When a page is first brought in, it is protected against write (whether that is the correct protection or not). Then when the first write occurs, a trap happens. The trap handler notices when a page is not really write protected, so it sets the dirty bit and clears the spurious write protection. Most of the time the page will never get written, so this technique saves lots of disk output.

We add more information to PPM for the replacement (removal) algorithm. If we are using LRU, then the time the page was most recently used is stored here if we are using LFU, then the count of uses during the current sampling interval is stored here if we are using plain FIFO noth-
ing is needed but if we are using FIFO second chance then the "half dead" bit is kept here. Replacement algorithms are discussed in Lister 5.4(3).

We add a lock bit to the PPM, which tells the removal algorithm never to remove this page at all. This is used for the OS code which should never be swapped out because it contains the swapper. It is also used for pages used by real-time processes, which must update values every few milliseconds and cannot wait for page faults. The OS has many such processes, primarily interrupt driven processes such as device drivers and the dispatcher. There may also be real time user processes, for example a video RAM which continuously updates a TV screen from primary memory. The number of locked pages (i.e. the amount of resident memory required) is a critical figure for the cost and performance of any VM. Therefore the OS runs as much as possible in swapped rather than resident memory.

Notice that the SPM allows a given swapped page to be kept anywhere on secondary memory. It could even move around when it is written back, it is only necessary to update the SPM to show where it was put this time. However, this means we need a new table, a bittable with one bit per secondary page frame (per disk address), to show which addresses are free.

An alternative way of doing it is to omit both this bittable and the information in SPM saying where the page is kept on secondary. Instead, there is a one-to-one correspondence between secondary pages and swapped pages. Then the correct disk address can be calculated from the index into the SPM. This is obviously more thrifty with critical resident OS table space, because the SPM is big. The only problem is that now pages must always be kept in the same place in secondary, but that may be worth it.

3. The Full VM - Address Translation Too

This completes VM.5. From here on up we deal only with swapped pages. But so far we have said nothing at all about process or file address spaces, address translation, and mapping. When these are added, VM.5 becomes the full VM.

The ideas of address translation and virtual address space go together of course: address translation is the conversion of virtual addresses to physical primary ones. Every process has its own virtual address space so does every file. A virtual address space is implemented by a "virtual page map" or VPM. (Lister uses the term "page table"). A process’s virtual address space is limited to 512 pages, so its VPM is 512 entries long. A file is the same but without the limit of 512. For each page P in the virtual address space, the Pth entry of the VPM specifies what swapped page shall represent it, by giving the SPM index of that swapped page. Figure 2 shows the relation between VPM’s and the SPM. Note that process page 2 and file page 0 are shared, i.e. they specify the same SPM page.

This VPM is used to drive the hardware address translation. To translate a virtual to a physical address, the page bits of the virtual address are replaced by the page bits of the corresponding physical primary page.

In more detail, suppose the process refers to word N of its virtual page M. Then by looking up the Mth entry of the VPM, we can find that this process’s Mth page is swapped page S. Then, by looking up the Sth entry of the SPM, we can find out if the Sth page is in primary memory now, and if it is, where. Suppose it is in page P of physical primary memory. Then the Nth word of page P is the word desired. Compare Lister’s figures 5.2 [or Silberschatz & Galvin 8.16].
If, however, S is not currently in primary, then a "page fault" occurs. The hardware address translator aborts the current instruction and traps. All the necessary volatile state of the CPU is saved, either by the hardware or in the FLIH. The process is suspended its process descriptor is marked "unrunnable - blocked for page fault". The removal algorithm is run and an unused primary page P is located. An IORB is created, requesting the disk driver to read page S into P. The request is passed to the disk driver the FLIH clears the interrupt and runs the dispatcher the dispatcher finds some other process for the system to run while this process waits for its page.

When the disk driver has gotten swapped page S into primary page P, it marks the process descriptor of the waiting process runnable again. Notice that this means that IORB’s must point out the process descriptor of the process that is waiting on that IO request to complete. This is the waiting process queue associated with the semaphore "request serviced" in the IORB, see Lister’s figure 6.3.

The dispatcher will pick the process again now, and restart it at the instruction which failed before because the page was not there. This time address translation succeeds, and the process is able to proceed. The page fault has been handled.

Two small details may test your understanding. I omitted to mention that at some point the SPM entry for page S must be updated to show that page S is now in primary page P and similarly the PPM entry for P updated to show that it contains S. I also have not considered what happens when a second process tries to reference S while the IO is in progress. These two problems interact with each other. We cannot link the PPM and SPM when the IO is requested, because then the second process would think the page was already in primary when it wasn’t. If we link up after IO is complete, then the second process may bring the page in again to a second place, causing the page to lose its identity.
The simplest solution is for the disk driver (1) after IO is complete, to link S and P, and (2) before processing any request, double-check to see if it is still necessary. Consider what happens when a second process faults for a page for which IO is already in progress. It will not notice that IO is in progress it will allocate a new hole it will place a new IORB in the driver’s queue. But when this IORB is removed from the queue by the driver for service, the driver will double-check. It will see that the page is in fact now already in. It can deallocate the extra hole and allow the second process to proceed right away. Obviously there is some wasted work here, in that the extra hole is allocated and deallocated needlessly, but the solution is simple and the situation probably does not arise often enough to make much difference for the system’s efficiency.

At this point we have reached a full VM. First, VM.5 provided convenient subroutines for swapping. Then, address translation was used to make the swapping fully invisible and automatic as far as the using process can see. Actually only the translation is automatic in hardware when it fails, the resulting trap and call to the swapper is of course OS software.

4. The Meaning of "Page Mapping"

No process is ever allowed to know whether a given page is in primary memory at any given moment. They are not even allowed to use or know about secondary addresses, the page’s “identity” for the OS. They are entirely restricted to referring to pages by their position in an address map, a VPM. However, they can and do often refer to the VPM’s of other processes or of files. Thus they can say, "make page N of my space be page M of that process’s space” or "of that file’s space”. This is called "mapping" pages, and is accomplished by the TOPS-20 monitor call PMAP. PMAP is the only way to change a map. There is also a corresponding monitor call RMAP for reading one entry of a page map. Of course RMAP only tells what PMAP was done last it would never tell anything about secondary addresses or SPM indexes. Notice well that mapping references only maps, not the page itself.

A page can go into and out of as many maps as you please without ever being written, read, or swapped in from secondary. Mapping is best defined as a declaration to the system that two virtual pages are to be made equivalent, or unified. This declaration involves no IO, nor reference to the pages in question, but only a change to the VPM table, the map.

Certainly it all seems natural enough that no IO is involved when two pages are shared between processes. No one expects process memory sharing to involve IO it is merely a matter of process memories overlapping. The thing that surprises people with TOPS-20 is that all information transfer to the disk occurs the same way: by uniting the address spaces of processes and files, and then writing on memory in the usual way. At the VM level, there just is no such thing as IO (to disk). There are only shared memories and memory references.

Later on, we will build another layer of software on top of VM, which reintroduces the notion of serial IO to disk files at a higher level, because it is convenient to think in those terms for many problems. But when we examine how the OS implements this, it will turn out that it is entirely accomplished by page mapping. The file’s space is unified for a time with some process’s the data is transferred via ordinary memory writes into shared memory the pages are then unmapped. At the end the file is there and has the correct contents, but nobody ever did any IO to the file.
5. Extending the VM

The VPM is now extended to provide memory protection. Protection bits are kept in each entry of the VPM telling whether the process in question is allowed to read and/or write the page mapped there. The address translation hardware uses these bits and traps any attempt to read or write illegally.

It is possible for several processes to map the same swapped page into their respective VPM’s, specifying different protections. This would allow, for example, a page in a database file to be mapped in by some processes with write access, and by others only for reading.

Of course, when a process maps a page, the OS must check that the access requested is really allowed to that process. Because mapping requests must always take the form, "give me page N from that process’s map" or "from that file’s map", it is sufficient if the system checks whether the requesting process has the desired access (read or write) to the pages in the other map. If it is a file page, for example, then "give me page N of the file with write access" is legal if the process has been able to open the file for write access, and otherwise not.

A given virtual page may specify no swapped page this is a "hole" in the address space. The process can specify the protection anyway, however! This can be very useful, so I give some examples.

First, some programs know that there are large parts of the virtual address space which they will never use. Any reference to these areas of memory is an error. Therefore, they set both read and write protect on all virtual pages in this area. But they never specify what page to put there. The VPM contains zero for the SPM entry, but has the read and write protection bits on. It is a read and write protected virtual hole (om nu det kan vara något). If the program refers to it, the program is halted by the OS with a memory violation error.

A second example: some programs allocate space dynamically as they run. Such programs typically lower the protection on the unused part of their address space, so that it permits both read and write, but still they specify no page. Eventually their allocator allows the virtual page to be used for some purpose in the program. When this page is first referenced, it still has zero in the VPM. The address translator traps, and the trap handler notices that the problem is just a non-existent page. So it allocates a new swapped page, initializes it to zeros, and maps it there. The effect, seen from the program’s point of view, is as if the hole was simply a page full of zeros that had been there all the time.

A third example is the INTERLISP system. This program can at times have need to write on any page in its address space. From the point of view of the program logic, there is no page that is unwritable. However, the system does need to know which pages are altered during the run, so that it can save the state of the run in a smaller file, by just saving changed pages. Therefore, all pages are initially write protected. As the system runs, pages are written on, and traps occur. The trap is intercepted by the INTERLISP system, which resets protection on the page to permit write, and marks the page in a table as having been changed.

6. Optimizing the VM

The VM seems to require that every memory reference requires at least two more to perform translation. The solution to this problem is to use a bank of associative registers in the address
translation hardware, which remember the 16 or 24 most recent translations. This is well described in Lister section 5.3(2). A distinction to bear in mind is between "page fault" and "page trap". In Lister’s diagram 5.4, a page trap is what occurs when the box "Match Found" answers "No", that is, the PAR’s do not contain the answer. The trap works just like any other interrupt, going via the FLIH to the service routine for this interrupt, called the page trap handler. This routine is the one that actually checks the VPM and SPM. If the page is really not in ("Page Present" is "No") then a page fault occurs. Every page fault involves a page trap, but not vice versa.

7. The Removal Algorithm

Lister calls this "the Page Turning Algorithm" and "Replacement Algorithm". Its job is to find a hole when we need to bring in a new page. The standard LRU, LFU and FIFO strategies are covered in Lister 5.4(3). The lecture added only the refinement to FIFO called FIFO second chance. While FIFO is marginally inferior to LRU and LFU, FIFO second chance is not (as far as I know), and it is easier to implement, for two reasons. First, it does not require repeated searches through all of PPM to find the oldest or least used page it normally should pick a page with a smaller search. Second, it requires less special help from the hardware. Whereas LRU and LFU must update a date or count on every reference to a page, FIFO second chance needs only some way to protect a page against all references for a while, which can usually be accomplished using the system’s protection mechanism.

There are many applications in systems programming for removal algorithms, because caching is such a general principle. Whereas the designers of the OS can perhaps influence the hardware design, to include what they need for LRU or LFU in the address translator, when the rest of us write caching schemes for various purposes, we have to either perform several extra instructions in software to update the date or count, or else use FIFO second chance and make use of the protection mechanism.

The FIFO scheme keeps a pointer into the PPM. In pure FIFO, when a page is to be removed, this pointer is advanced one step. (If it reaches the end of PPM, it is reset to the first entry the PPM is treated as a ring.) The page now pointed to is removed. Obviously this scheme offers only a very weak guarantee that the page removed is not being used. It is the page which has been in longest, but that tells us nothing about how much it is being used.

FIFO second chance adds a "half-dead" bit to the PPM. The pointer is advanced, and if the half-dead bit is on for the page now pointed to, it is removed. Otherwise, the half-dead bit is set for this page, and we try again (advance the pointer, test the bit, etc). The address translation hardware treats the half-dead bit as a trap condition, so that any reference to a page whose half-dead bit is one will cause a trap. The trap handler must notice that the cause is the half-dead bit the bit is cleared and the process causing the trap is then continued.

In this way, no page can be removed unless (1) the pointer has first come to it and set the half-dead bit then (2) the pointer advanced all the way around primary memory and came back to it, and during that time (3) no reference to the page occurred (since any such would clear the half-dead bit). Normally this second trip around is expected to take some time, and involve the removal of many other pages along the way. This means that no page can be removed unless it goes unreferenced for a reasonable period of time every page which ceases to be referenced will be removed after at most two such trips around.
8. Every Page has a Home

The removal algorithm is the only way that anything ever gets sent back to the disk. What happens when a program unmaps a page? After a while the removal algorithm will find it and send it back to disk. If the page is a file page, then it has a permanent home on the disk. If the page is a process page, however, things are different. A special disk area, not part of any file, is reserved for pages which are mapped into processes but not into any file. A home is provided for all such pages here.

Now there is only one problem left: when a process unmaps a page which is not mapped by any other process nor a file, then this page is garbage. There is no way anybody can ever refer to it again. The only way any process can refer to a page is via PMAP, which refers to pages only as "page N of such and such a map". If the page is not in any map at all, it is garbage, and the space for it should be reclaimed. I don’t actually know how TOPS-20 does this probably a reference count is kept in SPM of the users of a page.

9. Running Absolute vs. Running Resident

It was noted above that some pages must be resident for real time processes and for the swapper itself to function. There are in addition a few addresses that are subject to an even stricter requirement: they must be able to run absolute, which means with address translation turned off. Addresses are interpreted directly as addresses in physical primary memory. The code for the page trap handler must be such. It is invoked when the PAR’s do not contain the information to translate a memory reference. It cannot rely on them for running that would be circular.

When a page trap occurs, the address translation hardware (which is causing the trap) turns itself off. The FLIH and the page trap handler thus run absolute. The pages holding this code must not only be resident at all times, they must always be in the same position in physical primary memory, such as page zero. When the FLIH has finished with the interrupt, and transfers control to the dispatcher, it can turn the address translation back on, using the OS VPM.

10. Sequential Disk File IO

This completes the VM level. The next major subject is to build sequential IO to disk files on top of the VM. Information about files can be divided into three categories. Some is needed all the time, whether the file is in use or not. Such information is kept in the file directory structure the entry there for one particular file is called that file’s FDB (File Directory Block). Second, some information is needed only when the file is in use. The system has a table for such information, and every file that is in use has an entry there. The index of the entry is called the OFN of the file (Open File Number). There is one OFN for the file, even if several users are using it at once. Finally, there is some information which must be different for each of these several users therefore each has a JFN (Job File Number). There is a JFN for each distinct use of a file.

Typical information in the FDB is protection and access, dates of use, pointer to the file’s page table (VPM), and so on. Typical OFN information is the number of current readers and writers an indication whether multiple writers are allowed simultaneously now or not a pointer back to the FDB. Typical JFN information is the current position in the file if serial IO is in progress whether the file is open and if so for what, and so on.
The user must always start by requesting a JFN. There is a special monitor call for this, GTJFN. GTJFN is the only, repeat the ONLY, thing in the system that knows about file names. It converts file names to JFN’s all other operations on files require JFN’s to specify the file.

For example, to open a file for input, first get a JFN for it. To open it, use the monitor call for opening files, OPENF. OPENF requires a JFN (for an un-open file) and some information about modes: for read or write, what byte size, etc. Note that before the OPENF is done, the JFN exists, and the file is in use, but not open. Similarly, to open a new file for output, first get a JFN for the new name, using GTJFN, and then call OPENF to open it for output. Until it is opened, the JFN is for a file that is not open and in fact does not exist (yet). This file is just a half-finished entry in the directory structure so far.

We now consider how sequential output of bytes to a new file is done.

The user has done GTJFN on the desired name, and opened it for output. The FDB, OFN and JFN have been created by the system. A page table (VPM) for the file has been created. Instead of "allocating a buffer" to hold the text which the user will be outputting, the OS maps the first page of the file into its own space at some convenient place. This page is then pointed to from the JFN. It is called the "current page" of the JFN.

In fact, the JFN constructs a byte pointer to the first character position on the page, and initializes a counter of the number of free characters left on the page. Now the user can call BOUT (the OS routine for sequential Byte Output) giving it a character to output and the JFN. The BOUT routine will decrement the count, and if the result is not zero (the page is not full) it will store the character at the position indicated by the byte pointer and increment that pointer.

This will proceed until the first file page is full. At this point, the next attempt to output a character will notice from the count that there is no more room. The current page is unmapped. The next file page is mapped in instead. The counter is reset to indicate 512*5 characters free on the current page the byte pointer is reset to the beginning of the current page, and everything goes on as before.

The JFN also counts the total number of characters output to the file. When the file is closed, this value is copied into the FDB as its end of file (EOF). There is also a "byte size" figure in the FDB, which in our case will be set to 7 when the file is closed, because we were outputting 7 bit bytes. Note that EOF and byte size are merely the current values when the close is performed.

In this way, every JFN has a "current pointer" to some position in the file, which is represented by (1) having the page containing that position mapped in somewhere in the OS address space, (2) having a byte pointer to the byte (or word) position within that page, and (3) having a count of the bytes (or words) remaining on this page.

Figure 3 illustrates this. The JFN and current page of the file are mapped into the OS address space. Nothing is in the user space except the user program. The current page is shared between the OS process and the file just as a page was shared between a process and file in Figure 2.

It is very useful to compare the TOPS-20 organization to conventional methods with buffers, in the case where several processes are doing IO simultaneously, including possibly some writing. Suppose further that they are all doing IO at places in the file that are very near one another.
In TOPS-20, this will mean that the various JFN’s will have the same current page. There will really be only one copy of that page in primary. No matter how the reads and writes are sequenced, each process will see at every instant the correct state of the file. If process A writes the 900th character of the file, and process B reads the 900th character just after the write, process B will see the change.

In a buffered system, this can fail. A reading process can hold data in its input buffer arbitrarily long before actually reading it if in the meantime some writing process has changed the file, the reading process will still get the old data. Similarly, an output process can hold data in its output buffer arbitrarily long before actually writing into the file in the meantime no other process can see the change. "Arbitrarily long" can even mean "forever" if the job is killed or crashed before some condition occurs which forces the IO to actually occur.

The whole problem here is that the buffers are essentially copies of the file contents, raising questions as to which copy is the authoritative one. TOPS-20 avoids this because the VM guarantees that there is at all times exactly one copy of any page. Multiple processes doing IO to a file are just sharing the file’s address space, and they all read and write directly to it.

### 11. Random IO

Random IO to the disk, also called Direct Disk IO, is very convenient in this scheme, because the whole scheme treats files as address spaces (i.e. random access memories) at bottom, rather than serial devices. To write at a particular byte address, one calls SFPTR, giving it the JFN and the desired address. Thereafter one can write sequentially with BOUT as much as desired, before adjusting the address again with SFPTR. How does SFPTR work?

SFPTR calculates which virtual file page the specified address lies on, say page P. It then maps the Pth page of the file into the OS as the current page of this JFN. The JFN’s byte pointer and count of free characters left on the page are calculated and set. And that is all.

### 12. EOF and Byte Size

Whenever a file is closed after doing sequential output, the current position and byte size are copied from the JFN to the FDB. In other words, the EOF and byte size of the file are set. Whene-
ever a file is opened for input, the EOF and byte size in the FDB is copied to the JFN. When the last page of the file becomes the JFN’s current page, the count of characters remaining on the page will include only the characters that are actually in use. When an attempt to do sequential input causes this count to run out, the EOF signal for that JFN is generated.

Notice that the input process need not open the file in the byte size indicated in the FDB. The EOF and byte size are converted to the size requested by the opening process when they are copied to the JFN. Think of the byte size as the unit of length in which the EOF is expressed.

A file with a certain true internal structure -- say a text file of 7 bit ASCII bytes -- can therefore be processed in other byte sizes for many purposes. Copying to and from disk, or over network connections to other PDP-10’s, is normally done in full words (byte size 36 bits) for all files. Such a copy will correctly reproduce any file in all cases, since at bottom all files are made up of pages of words. The file thus written will show a byte size of 36 bits, and a length in words. But if its true internal structure is text, it will be used with programs which expect and require text, such as compilers and text editors and print programs. These programs open the file for 7-bit bytes, and they will see the same result, despite the apparent byte size of the file, because the underlying pages contain the same bits. So all programs normally ignore the byte size altogether and open files in the byte size they require. The OS takes care of converting the EOF in the size stored in the FDB to the size in which the JFN is opened.

13. Sequential Output to Non-Disk Devices

There are several different choices one must make in designing the IO for an operating system. Most are well covered in Lister here I only want to note which choice TOPS-20 made in each case.

The first choice is to have synchronous vs. asynchronous IO. With synchronous IO, a process requesting IO is blocked until the IO requested is complete. With asynchronous IO, the process continues running after placing the request, although the request will not be fully processed for some undetermined time yet. The process must use some additional monitor call to test if the IO is finished or else it must be interrupted when the IO is finished or else it may be provided with no way to test, but a monitor call to wait until it is finished.

The second choice is whether and how much to buffer. Buffering is the only reasonable choice really. When people talk about unbuffered IO, what they really mean is that they want to keep the buffers in their own space. When the buffer is full, the process tells the OS where it is, and the OS sends it to the device, without copying it unnecessarily into another buffer in the OS. So the real choice with buffering is whether to have the buffers in the process’s space or in the OS.

Having the buffers hidden in the OS is good, because it makes the IO operations higher level and relieves the programmer of lots of unnecessary detailed work administering buffers. It also makes the OS more reliable, since the calls to the OS are simpler and are more easily checked if the OS has to use buffers built by the user, then the OS must be made secure against all sorts of rubbish which users could put in their buffers, such as incorrect pointers and counts.

Having buffers hidden in the OS is bad, because it is less efficient in common cases. If the program is generating IO one character at a time, for example, then all the overhead of switching from user mode to monitor mode, checking the legality of the user’s JFN, and so on, must be paid for each character. In such cases the programmer soon starts building lines or larger units
in his own space and delivering them to the OS all at once, using the monitor call to output whole strings. In other words he goes back to buffering in his own space, but now with the added disadvantage that the OS will then buffer it again.

TOPS-20 as usual tries first for convenience and then provides also the inconvenient but faster options. Convenience in this case means the buffers are in general hidden in the OS. Then certain special cases are met by added extra monitor calls that provide faster solutions.

For magtape, in addition to the normal IO which is synchronous and buffered (in monitor), there is a special "dump mode" which is asynchronous and unbuffered (i.e. buffered in user space).

For disk, speed may be obtained by mapping your own pages around. This is what I called "Do it yourself IO" in the lecture. In the discussion of serial disk IO above, concerning the JFN, the current page, the count and the current byte pointer, there was nothing the user cannot do for himself. I do not mean that the user may reference an OS JFN, that is of course forbidden. But he can map in the next file page and keep a table like a JFN, and perform all the same operations himself. On input, this is about 10 times faster than reading characters one at a time from the OS with BIN one avoids the overhead of monitor call entry and exit, and verification of the legality of the request. The only reason for not doing so all the time is that it is inconvenient.

Speed is obtainable for some other devices, such as line printer and card devices, because they are in fact spooled to disk. For terminals the only optimization the user can perform is to choose a restrictive wakeup set (see below) and read whole lines into his program.

There remains the question of how much to buffer. One of the advantages of buffering is that the process is blocked less often, since it does not have to block on every character or word of IO, but only when the buffer becomes empty or full. Obviously bigger buffers are better in this regard. They are also much better for keyboards. In the case of keyboards, the input data is not lying passively on some storage medium, with nothing better to do than wait for the process to read it. There is a person there, and it is quite important for interactive systems that the person be able to type ahead as far as possible. Sometimes this is simply because the program is going to take a long time, the user knows it, and he wants to type ahead 20 minutes’ worth of work in three or four lines, and then take a coffee break. At other times it is just because the user types faster than the system can read.

So in general bigger buffers are better. This is approximately the place in courses on operating systems to discuss double buffering. But double buffering has very little use in TOPS-20. Why is this?

Discussions of double buffering usually fail to make explicit one of the assumptions involved, which is that the user program cannot be entering new output in the output buffer at the same time the device handler is removing data from it. In fact, this is not impossible, just rather inconvenient for devices like magtape. Therefore, for such devices, most systems provide two (or more) buffers, and the process can fill one while the device handler empties the other.

However, for keyboards the solution is in fact to deny that hidden assumption. Using mutual exclusion to synchronize, the user process can read characters from the buffer while the user is typing more, and analogously on output. The solution is more or less forced. Double buffering for terminals (both in and out) makes little sense, because the speeds are so low, there can be indefinite waits before data comes in, the data does not arrive in any special block format, and so on.
For magtape, I do not know if TOPS-20 double buffers, but since the programmer can use asynchronous unbuffered IO, he can program double or multiple buffering as he sees fit.

For disk, processes can map as many file pages as they want, fill them in any order, unmap and remap them as they want, and rely on the removal algorithm to do the output at some reasonable time. This is like having infinite buffering.

So in TOPS-20, one does not talk much about double or multiple buffering, first because all buffering is normally hidden from the user, and second because few devices are double buffered: they are infinitely buffered, or unbuffered, or the single buffer is asynchronously accessible. This does not make multiple buffering a bad idea. Not at all it’s just that in TOPS-20 it typically is called something else, like paging.

14. The Terminal

The terminal (and its cousin, the network connection) is the hardest device to handle, without a doubt. If you do not believe it, make up a list of 10 Good Things To Have on your terminal and then go see how many operating systems exist that have five of them. The reason is that terminals and network connections are much more truly asynchronous than other devices, mainly with respect to input. With other devices, control and initiative remains with the CPU. It is asynchronous because the CPU does not want to wait for slow mechanical things to happen, and therefore prefers to be notified of their completion by interrupt. But things happen only when the CPU requests it.

With a keyboard, initiative and control lies (or should lie) with the user. He can initiate IO at any time what is much more difficult for the OS, he can (or should be able to) interrupt his program. By interrupt is meant of course NOT JUST KILL, but truly interrupt, that is, asynchronously cause a transfer of control in the program. Whereas ordinary characters should be held in the input buffer till the program needs them, interrupt characters should pass straight through, causing the interrupt when they are typed. Thus the terminal not only has an asynchronous relation to the CPU as do all devices it is also asynchronous "within itself" in that an interrupt character can affect the processing of ordinary characters which were typed before it was. The simplest example of this is the very useful keyboard interrupt which causes any typeahead to be thrown away. Suppose I have given a compile command, for a large, long compilation. I then type ahead the load command which logically follows but I misspell it. Now what I want is a keyboard interrupt to say "forget it", i.e. to throw away my typeahead. After that I will try again to type ahead the load command correctly.

Because the terminal is so asynchronous, there are many operations on it which depend on real time in some way, despite the general organization of terminal IO in TOPS-20 as synchronous IO buffered in the OS. Among them are:

Flush output buffer. When a program outputs text to the terminal, it may sit in the OS output buffer for some time before the printer actually catches up. This buffer can be fairly large, perhaps several hundred characters. Making it bigger helps the OS scheduler reduce process switching overheads, because the user program can be given more service less often. But if the user then aborts the current operation with an interrupt, he may be in for a long, unwanted printout the computer seems to him to keep going for a long time. The code that handles aborts of this kind wants to clear the output buffer.
Flush input buffer. An example was already given above.

Test input buffer. Many programs want to test whether the user has typed anything or not yet, for example to suppress prompting for the next input if the user has already begun typing it.

Test output buffer full. This allows a program to keep a printer going while doing something else too, by polling the printer now and then. If its output buffer has room, put some data in it.

Wait for output buffer empty. This is used a lot for error messages. When an error occurs, flush output, print the error message, and wait until the message has definitely been output, i.e. seen by the user. Then flush his typeahead.

Interrupt on a given control character.

Interrupt on next character. This allows a program to process background jobs, and break off to respond to the user whenever he types anything.

Interrupt on output buffer empty. This allows interrupt driven programming of printers.

In addition there are of course lots of other operations to read and set the terminal characteristics: does it have tabs, formfeed, backspace and so on.

Finally, "wakeup sets" should be mentioned. The wrong way to do terminal input, which unfortunately is nearly universal apart from DEC computers, is to accumulate lines in the terminal device handler. The user program does not get run until a carriage return is typed. This has the unfortunate implication that line editing and many other useful interactions within the line cannot occur in real time. Your terminal can erase one character if it receives a backspace, and the computer can send it one, but when you type rubout, the rubout just sits in the input buffer until you finish the line.

If efficiency did not matter, the terminal handler should pass along each character to the program as soon as typed. But efficiency does matter, and if every character meant the program had to be scheduled by the dispatcher, its pages faulted into primary, and a few instructions executed to receive the character, then 50 users banging away on their terminals could keep the biggest mainframe busy just collecting their input. So we need a choice of wakeup sets. TOPS-20 has four, and a program can tell the OS which one to use. The strictest one wakes up the program only on control characters. This includes carriage return of course. The most generous one wakes the program up on every character, and there are two in between that distinguish between alphanumerics and punctuation, for programs that want to be wakened on every "symbol" (where the program’s scanner breaks symbols on spaces and other punctuation marks). Interrupt characters are of course always a wakeup condition.