Contents

6 Introduction to Memory Management and Filesystems

There are two important resources which we have not yet treated in detail, and both are concerned with data storage—one temporarily and one permanently: Memory management is all about how processes will be given access to the computer's RAM, and filesystems deal with the question of how to store files (and possibly directory structures) on a harddisk.

We will treat these two subjects in one chapter because there are many concepts that appear in both topics, and there is a certain relatedness of both fields.

Note that from a process' point of view, memory is a direct resource and is needed for running the process: If the process' program code is not available in memory (at least partially), it cannot be executed, because the CPU can only execute instructions that are located in RAM. On the other hand, disk space is not something that a process will (directly) need, instead access to certain files may be a necessity for running a program. Now if we look at disk space from a file's point of view, we could say that a file (in order to be read, modified or grown) needs disk space in a similar way as a process needs memory to be run.

Let us look at some of the concepts that appear in both memory management and filesystems:

partitioning of resources: On a system that will handle several processes in parallel, memory must somehow be partitioned so that each process can use a fraction of the RAM. Individual cells of memory are exclusive: They can hold precisely one byte of information, and it has to belong to a specific process (or the operating system itself) at any given time. It may not be necessary that a process has memory throughout his whole lifetime (for we will see that concepts such as swapping and paging allow data to be stored on the disk for a while), but at least in those moments when a process is actually excecuted by the CPU, it will have to be given at least some RAM. It may be useful to limit the maximum RAM that a process can access at any given time.

Similarly, if a system allows several files (and possibly directories) to be created, accessed and modified on the disk, disk space has to be

partitioned in a way that the disk can hold all these files and present simple means to lookup files on the disk and access them. The smallest unit of storage would in theory also be a byte, and such a byte (now meaning the fixed location on disk) can only belong to one file (or to the filesystem metadata) at any given time. As in the memory situation, it may be useful to define a maximum filesize so as not use too much of this resource, though most filesystems that implement such limits do this on a per-user basis and not on per-file basis—limiting disk usage per user (or per user group) is called a quota system.

access control: Access to memory locations should always be exclusive to one process (or the operating system), otherwise a process could read or even modify the process memory of a different process which is not advisable, because it would be a source of instability or security problems.

Access to files is also often handled in a way that makes it exclusive to a file owner, typically the file creator (or perhaps some other users, depending of the access concepts a specific filesystem may have). And from the view of processes it may be necessary to restrict file access to only one process (even in a situation when several processes belong to the same user who is also the owner of the file), so that no errors can result from parallel access to a file.

free space management: Memory and disk usage must be handled dynamically, because processes newly appear and are removed from the system all the time, their needs for memory may change during the process runtime, and also files can be created and deleted as well as grown while the system is active.

In many memory management schemes there will be a list of free memory locations. We will see that is not useful to grant memory access byte-wise, memory will often be partitioned into equal-sized smallest chunks of memory that can be assigned to a process or removed from it. If we call these smalles chunks (say, of size 1 KByte) memory frames, then there will be need of a "free frame list" that knows which frames are currently unused.

In the same way disks aren't typically accessed byte-, but block-wise, a block being a fixed size segment of the disk space. Note also that read and write operations on the raw disk device always transfer a whole block of data, and not a single byte. We will need a "free block list" in order to know which blocks are still available for file storage and which are not.

Methods for administering such free frame lists and free block lists will be similar.

... ...

6.1 Contiguous allocation

In this section we present the most simple methods to distribute memory among processes and disk space among files.

6.1.1 Fixed-size partitioning

Consider a computer that has 1 GByte of RAM. If we divide this memory into 1-MByte-sized partitions, then we get 1024 such partitions, some of which will have to be reserved to the operating system itself. Assuming that 1000 "unused" partitions will remain, such a system would allow for up to 1000 processes to be started and hold in memory in parallel. Each of the processes will then have its own 1 MByte memory partition, meaning it can use up to this 1 MByte for storing its own program code, stack, and data.

Obviously this method is not very flexible and it limits the possible usage of the system in two ways:

- No more than about 1000 processes can be run in parallel. If there was need for, say, running 2000 or more processes at the same time, then the whole system would have to be reconfigured (with smaller, but more memory partitions) and completely rebooted.
- No more than 1 MByte of RAM can be given to a single process. If a program required more than that, say 2 or more MByte RAM, again the system would have to be reconfigured.
- It would be completely impossible to change the system parameters in both directions, i. e. allow for more than 1000 processes *and* some of them using more than 1 MByte RAM.

Now, in the same way consider a harddisk of size 1 GByte and a similar partitioning scheme that would allow 1024 (minus a few) files of up to 1 MByte size to be written to this disk. The same problems as in the memory example would occur: There would be a filesize limit as well as a limit on the number of files, and changing the filesystem structure in order to either allow more or larger files would require the disk to be newly formatted, and a change would only increase one of these numbers while reducing the other.

This simple approach is called "fixed equal size partitioning" in both the memory and the harddisk case, and besides the limitations already discussed it leads to a problem called internal fragmentation: While the RAM is fully split into partitions, i. e. there remains no unpartitioned and possibly unusable

memory (that would be external fragmentation), a lot of memory will go unused, e. g. when a process runs that needs only a few kilobytes of RAM but still gets the whole 1 MByte. There is no way for other processes to claim some of this unused memory because the fixed partitioning forbids this.

It is an example of contiguous[1] allocation methods: Contiguity means that all the parts of a process' memory (or of a file on disk) are stored in consecutive[2] frames/blocks, and also in order. So no jumps to other memory locations or disk blocks are necessary when reading the whole file (or the process' whole memory) from the first to the last byte. The opposite of this in non-contiguous allocation, and we will get to that approach in section 6.2.

Note that we use the term "disk partition" in a non-standard way; we do not mean the logical partitions into which a disk is separated on today's standard computers in order to create several logical volumes (in Windows language: drives) each of which is formatted with its own filesystem. For simplicity we assume that a harddisk contains exactly one filesystem and that this filesystem uses all of the disk, as it is the case on floppy disks and (some) USB sticks. See section 6.1.4 for a few words about the classical understanding of "disk partition".

6.1.2 Variable-size partitioning

So far we have seen partitioning schemes where all partitions are of equal size, which caused the two limitations in file size and file number. A little more flexibility is introduced when we raise the equality condition: That leads us to a new method of creating fixed partitions, but of varying sizes.

It is just a small alteration, but it already improves the situation a lot: In the memory case, if a process is associated with a memory partition and it wants to extend its memory usage beyond the current partition's limits, it can be relocated to a different (larger) partition, and on the other hand many more processes can use the system if there is a good mix of processes with small and large memory demands. Note that this partitioning scheme is still fixed: At system boot-time, the memory partitions are created and cannot be modified until the next booting (and a modification is likely to require a recompilation of the operating system kernel).

In the same way a filesystem with fixed partitions will profit from this modification by allowing both more and (some) larger files. The strictness of the partitioning applies here as well: Once the disk has been formatted, the partition (i. e.: maximum file) sizes can only be changed by reformatting the whole disk.

[1] DICTIONARY: *engl.* contiguous = *dt.* zusammenhängend
[2] DICTIONARY: *engl.* consecutive = *dt.* fortlaufend

6.1.3 Dynamic partitioning

A lot more freedom in memory or disk space allocation is possible if the partitioning becomes fully dynamic: This means that no partitioning occurs at the system start or during the formatting of the disk, but instead partitions are created as need for them occurs.

This has the effect that a lot more administrative work must be carried out by the operating system, for example keeping an overview of free areas of memory becomes more complicated, because whatever data structures are used for the memory or disk allocation, they are now dynamic.

6.1.3.1 Free frame lists

The most simple approach is to keep a list of free partitions. This list will be called a free frame list in memory management or a free block list in filesystems. Typically there is a smallest possible fragment that can be allocated, called a frame or block, and free space managements only deals with these frames/blocks. The smaller the frame or block is, the more of them exist and the more of them have to be handled by the free frame list.

One approach is to have a linked list that contains descriptions of free areas, e. g. a start address and a length for each one. In the list each entry points to the next entry when working with pointers. In order to find a free area of a given size an algorithm will walk through this list and stop when it finds an area of sufficient size. For this purpose it may be necessary to scan the whole list if (in the worst case) the only fitting area is at the end of this list. When a number of previously free frames is allocated to a process (or blocks to a file), the list has to be modified,

- either by removing the entry if the whole lot of contiguous blocks are allocated,
- or by modifying the entry if only a few of the blocks are allocated, and they are located at the beginning or end of the area described by this entry,
- or by splitting the entry in two parts, if (for whatever reason) a section taken from the middle is allocated, leaving free areas in front of and behind them.

If the used space is later released, it must be added to the list again, possibly creating a list entry that has to be merged with entries describing directly neighboring areas.

Another possibility is to work with bitmaps: For each frame/block a bit in this bitmap defines whether it is free (0) or in use (1). Here no complex

list administration (with the mentioned splitting and mergers of list entries) is required, however allocation and release of blocks lead to modification of several bits in the bitmap, and looking up free space of a given size means finding a number of consecutive 0-bits in the bitmap.

Note that it does not matter at all whether we think of memory frames or disk blocks, the concepts are identical. Differences will however appear when thinking of storage of these lists or bitmaps: In the memory case it is obvious that the list must also lie in memory for quick access. In the filesystem case it might make sense to store the list in memory (and not on disk as well) in order to speed up the lookup of free areas—but depending on the size of the free block list, it may be too large to keep all of it in memory.

6.1.3.2 Allocation

When working with dynamic allocation of free areas, there will typically be a choice among several free areas which are of sufficient size, and the procedure for choosing one of them will have consequences both on performance and on (external) fragmentation: If the decision algorithm is very complex, allocation will always take a lot of time; if it is simple, there will be many small unpartitioned (not allocated) areas which are too small to be useful anymore, so this external fragmentation will lead to memory or the disk filling up more quickly than necessary.

On the following pages we will present five simple approaches to allocation called first-fit, next-fit, best-fit, worst-fit, and quick-fit; and after that a more advanced concept called the buddy system will be introduced.

First-fit Bla

Next-fit Bla

Best-fit Bla

Worst-fit Bla

Quick-fit Bla

6.1.3.3 Buddy system

6.1.4 A few words on Disk Partitions

As mentioned above, we have not been talking about hard disk partitions in the sense of creating several logical volumes on a disk for use by various operating systems (e. g. a Windows and a Linux partition) or for structuring the disk so that different data can be stored on different partitions (e. g.

"drives" C: and D: for Windows or partitions /, /home, and /usr for Linux)—now we do, because this kind of partitioning is another example for contiguous allocation with flexible size. Most disks have a partition table as created by Windows, Linux, DOS, and other operating systems when initializing a hard disk. (The BSD operating systems use a different method to partition disks, calling the partitions "slices" and the partition table "disklabel".)

A classical partition table puts no limits on the sizes of individual partitions, but allows only up to four ("primary") partitions for whose administrative data it reserves space in the first blocks of the disk. There, you basically find the start address and the length of each partition. If more than four partitions are needed, one of the four must be set up as an "extended partition" that holds an additional partition table and the "logical partitions" that reside inside the extended partition.

If we ignore logical partitions, we see that this is a simple implementation of dynamical contiguous allocation wth flexible size; partitions can be created and deleted, each partition has to be contiguous, and in principle the partitioning also suffers from external fragmentation: If you start with a 40 GByte disk that is partitioned into four 10 GByte partitions and you resize each of them to 9 GByte, you end up with four unused 1 GByte areas that cannot be used. If there was no "four partitions" limitation, the four free areas could be made into four separate 1 GByte partitions, but never into one 4 GByte one, since these four areas are not contiguous.

In order to change the size of a formatted partition (i. e. one with a valid filesystem on it), always two steps are necessary, with their order depending on whether the partition is to be extended or shrunk: The logical filesystem must be resized and the partition itself must be resized.

- When extending a partition, the operation on the partition (and partition table) comes first, only when this is completed, can the filesystem size be increased as well so that into grows into the newly available space.
- When shrinking a partition, the filesystem has to be modified first, because for example files residing in the parts of the partition that is to be removed must be relocated to a different area on the partition first. Only then can the partition itself be resized (making the removed parts unaccessible to the filesystem).

Note that modifying a filesystem size requires more than (possibly moving files from an area that is to be removed, and) changing the information about the partition size in the partition's metadata, for example on a Unix filesystem the free block bitmap has to be grown or shrunk as well in order to correspond to the changed number of blocks.

6.2 Non-contiguous Allocation

So far we have seen several examples for contiguously assigning memory or disk space to processes or files. That allows for a very simple handling of accesses, because only an absolute start address and the length of a (memory or disk) partition must be known.

However, since this leads to strong limitations in usability, all modern operating systems use a more flexible approach both for process memory and files, assigning memory frames and harddisk blocks non-contiguously.

Non-contiguous allocation makes things more complicated, and for process memory it is worse than for files:

- If a file consists of several, non-contiguous blocks which are spread all over the disk, there has to be a list of blocks that tells the operating system where to find the data. When trying to read a specific byte from the file, the address within the file has to be translated into a disk block and a relative address inside that block. It also means that reading a file from beginning to end can no longer be achieved by reading several disk blocks in their natural order, but the operating system has to jump from one location to another all the time, so several disk head movements are involved which speeds down the access.

- With memory things become even more complicated: Here also some kind of table is needed that will be used for address translation, but memory access is different from file access. A program contains a lot of memory accesses: every (absolute) jump to another instruction in the program code and every direct data access (where the content of some memory address is loaded into a CPU register) and similarly each preparation for indirect access (e. g. `ld hl,0xa000; ld a,(hl)`; what's that in i386 code?) contains an address.

A process could be told the absolute address ranges of the memory locations it was given access to, imagine it has a list like this:

1. partition 1: 0x10000–0x10FFF (4 KByte)
2. partition 2: 0x14000–0x15FFF (8 KByte)
3. partition 3: 0x20000–0x2FFFF (64 KByte)

Assume further that the program code is 6 KByte long and the rest of the memory will be used for data (no stack in this simple example). Then the program code will have to be split between the first and second partition, and the data between the second and third one. (For this example we ignore that

it might be a better approach to store all of the program code in the second partition and use the first and third one for data, especially since the program might not know the precise number and sizes of partitions before it actually gets them.)

If there is a jump instruction in the program code that leads from the front part of the code to the rear part, it will cross partition borders. Also when the programs needs to access its data it has to be aware whether the currently needed data reside in the second or in the third partition.

All these problems can be solved with a method called *address relocation*. When using this system, at compile time a list of address references will be generated. It lists all references to data or instruction addresses that are used within the program code. At load time the system must know the maximum memory demand of the program, assign memory partitions and then use the relocation list to adjust the memory references to the concrete partition locations.

While this means some overhead during compile and load time, it works quite well, but only for static addresses. If the program dynamically "acquires" memory using some function such as `malloc()`, then this function will also have to be informed about the memory partitions and return proper addresses.

Another problem with this approach occurs when the operating system allows a process to be swapped out to disk and swapped in again later: At swapping it back in, the process may be given different partitions as before, and then all address references have to be relocated again, this time not only considering the addresses that were stored in the relocation table, but also the dynamically assigned addresses.

The relocation approach makes it hard to protect one process' memory against accesses by another process, because the address calculation in the relocation step only guarantees that static address references are fixed at program start; the program would however be free to access any part of the memory unless the operating system somehow checked each memory access against the partition list for this process. The simple start and length registers from the contiguous case would no longer be sufficient, because there are possibly a lot of partitions if a process uses much memory.

Note that for files a similar scheme can be adopted, and it causes much fewer problems: typical operations on a file are seek, read and write operations, and they will require translation of linear file positions (thinking of a file's bytes as numbered from 0 to $n-1$ for a file size of n) to absolute disk addresses inside the disk partitions. This translation occurs with every single access to the file. It could be avoided by also using some kind of address relocation as in the memory case, but the gained performance would not be worth the extra effort, because address translation is fast in comparison to disk access.

6.2.1 Virtual memory: Paging

All modern operating systems use a *virtual* memory management mechanism called *paging*. The idea behind paging is to give each process a virtual memory space that is addressed contiguously and linearly (starting with an address 0 and ending with an adress $size - 1$) that is partitioned into a set of so called memory pages. All pages have the same size, say, 1 KByte, and they are mapped to so called page frames which are equally sized chunks of the real memory. With the help of a page table each access to a virtual address is translated to a real address by first calculating to which page the address belongs, then looking up the corresponding page frame via the page table and finally locating the relative position within that page frame.

This approach makes compiling an application very easy: All references to addresses, be they jump instructions or data accesses, can be stored with absolute addresses inside the program, and no relocation takes place when loading the program. Instead each memory address will be translated using the page table. In order to make this fast, the CPU has to help: paging only works properly on computers that have a memory management unit (MMU) that takes care of the address translations: One of the CPU registers must point to the page table, and then the rest is all being done in hardware. So no computing capacity is wasted on the address translations.

When, for whatever reasons, locations of page frames have to be changed, it only takes a correction of the page table to make sure that the program continues to be runnable. This scheme also allows for individual pages to be removed from memory altogether and stored on the hard disk for later retrieval—this is called paging as well, and it is not to be confused with swapping a process' memory (meaning: writing all of it to the disk). A process that has some of its page frames paged to disk can still be run, while a process that was swapped to disk, must first be swapped in before it can resume action. However the disk space reserved for paging is often called swap space for historical reasons. E. g. the Linux operating system calls paging partitions or files swap partitions and swap files, but it does not implement swapping; it pages.

6.2.2 File Allocation Tables and Indirection

Somewhat similar to the way in which a page table holds information about the page frames currently used by a process, a file allocation table keeps record of disk areas used by a file. A thing that is shared by both methods is the use of equal-sized partitions of the medium—in the case of hard disks they are called blocks and typically have the size of a few kilobytes, 1, 2, 4, or 8 KByte, e. g.

For each file the operating system has to keep a list of blocks that the file's

data occupy. With very large files this list also becomes very large, because a file of size 1 MByte uses 1024 blocks, if the block size is 1 KByte.

Storing the block list in the overall data structure that the operating systems keeps for administering the filesystem, is not very efficient, because in order to allow for huge files, each such entry would have to reserve space for a possibly very long list—even for those files that only use a few blocks. Thus many filesystems store the block list in special data blocks. This approach is called *single indirection*: From wherever the information about a certain file is stored, entries do not point directly to a data block, but to an indirection block that contains further pointers to several other data blocks. These entries can be block numbers, since by multiplying the block number with the block size the absolute disk address can be calculated. If it takes 2 bytes to store a block number and a data block has size 4 KByte, then 2048 block addresses can be stored in one indirection block. When the first indirection block is fully used, a second one can be introduced in order to allow for even bigger files.

Typically the administration data will not only contain pointers to indirection blocks but also a few direct pointers (to data blocks) so that in the case of small files it is possible to find all data blocks without going through indirection blocks. Only when the number of data blocks exceeds the number of directly stored block addresses, a first indirection block will be used.

With single indirection the maximum size of files grows a lot; however it is still limited: If there are 20 pointers to indirection blocks and such a block stores 2048 block numbers (as above), then this allows for 40 K data blocks or file sizes of up to 40 K $\times$ 4 KByte = 160 MByte. By adding more and more indirect pointers in order to allow for yet bigger files, the administrative data for a single file grows equally; so a second level of indirection is introduced to keep the file entries small. With *double indirection* there are pointers that point to indirection blocks which link to further indirection blocks. Those then finally point to address blocks. What we said about the number of block addresses, remains valid in the case of double indirection, but now one double indirection pointer allows to address $512 \times 512 = 512^2$ (or roughly a quarter million) data blocks.

If this is still not good enough, *triple indirection* or even higher levels of indirection can be introduced: With each additional indirection step the maximum file size grows by the same factor (512 in the example). But notice that it makes no point to use, say, ten or eleven layers of indirection just to be prepared for any possible future demands on file sizes: Indirection leads to extra accesses; in order to read a specific block from the disk whose block number is only available through a long indirection, several blocks have to be read from the disk. If the block is on triple indirection path, it actually takes at least five read operations to retrieve the data: The first one is for looking

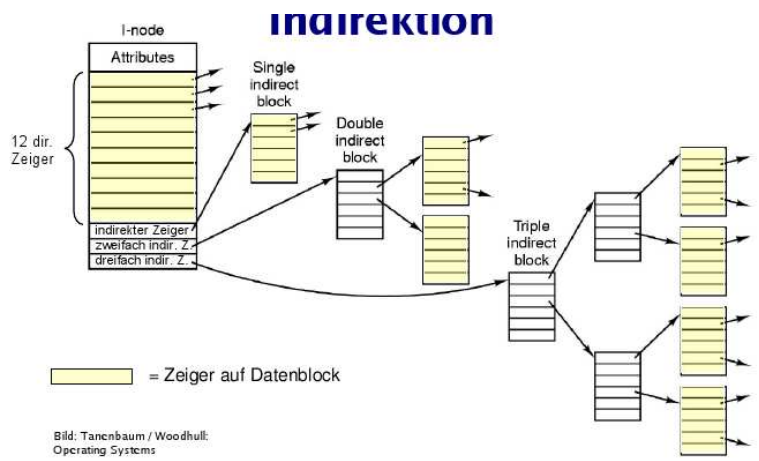


Figure 6.1: Multiple indirection in Unix filesystems

up the address of the first level indirection block in the file's administrative data. The second to fourth are for reading the indirection blocks, and the fifth one is the data block itself.

Whatever level of indirection is used, there are typically also indirection entries of all lower levels: In the same way that it makes sense to keep a few direct block number entries to speed up access to very small files, it is useful to have one (or a few) single indirections for those medium size files that do not require double indirection, and so on.

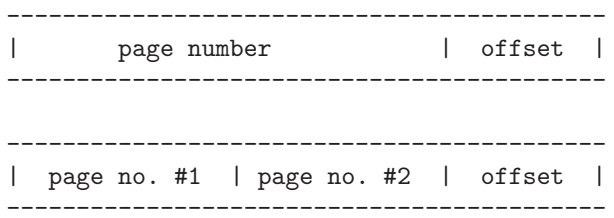
Figure 6.1 shows an example for multiple indirection in a Unix type filesystem. What is called "inode" in the image is a special administrative entry for a file that holds most of this file's attributes including direct and indirect block numbers as well as things such as owner, owning group, access rights, but not a filename. We will come to this later when we discuss examples of real filesystems.

6.2.3 Paging with split tables

In the same way that we found indirection to be helpful in order to decrease the size of a file's administrative data, classical page tables can become very large—so large that it wastes too much memory to store them, especially since in most cases a page table will only be used very sparsely: Most entries will be null pointers.

So in a way that is very similar to the filesystems' indirection concept, it is possible to split page tables so that they will become smaller, but there will

be more of them.



For example when having a 32 bit (virtual) address that consists of a 20 bit page number and a 12 bit offset, it is possible to split the 20 page number bits in two halves with the first halve pointing to a secondary page table in which the second halve is used to look up the page frame.

The first ten bits in this example can be called upper page number, the last ten bits lower page number. The effect on the page table size is reducing it to roughly the square root, e. g. from 2^{20} to 2^{10} entries. If the size of one entry were one byte, the reduced table would have exactly square root size.

Notice however that while going from a million entries to 1000, this introduces 1000 secondary page tables. If a process actually used this much RAM, all the secondary page tables would be filled, and no space would be saved. (Actually, in that case you get an increased amount of space used for tables, because the primary table counts extra.) But normal processes will not have such enormous memory demands, and this saves a lot of space because secondary page tables can be created on demand—as long as a process uses only a few kilo- or megabytes of RAM, only the first few secondary page tables need exist.

Using such split page tables is similar to the indirection in filesystems' data block lists. What makes it indirect is that in order to find the corresponding page frame for a page means no longer just looking into one page table; instead it requires looking in the primary page table first in order to find the right secondary page table and then looking there. So in the same way that using indirection in filesystems increases the number of disk accesses necessary to find a data block on the disk, introducing split page tables increases the number of memory accesses needed for finding a page frame.

6.2.4 Translation Look-ahead Buffers and the Locality Principle

Each memory access takes a little time, so it makes sense to use some kind of caching mechanism because most programs will not randomly access memory but instead access addresses which are close to one another. Think of loops reading all the elements of an array: they will be stored consecutively. So after

one access to a memory frame it is likely that further accesses to the same frame will occur soon after the first one. This is called the *locality principle*. Lookups of the same frame would mean translating the page number to a page frame number again and again—in order to speed up this process many memory management units contain a *translation look-ahead buffer* (TLA). That is a special type of memory called *associative memory* which can store page/page frame pairs and allows lookup in constant time: In order to find the page frame for a given frame (assuming it is stored in the buffer) there is no need to loop over the entries in the buffer, but the buffer will return the frame number immediately if it contains the page number. If it does not, the result is an error, and the normal lookup process will start. But finding a frame via the TLA is orders of magnitude faster than going through the regular tables, and this holds even more if split page tables are used.

The size of the TLA is typically very small, because those kinds of chips are limited in their size but the locality principle will guarantee that for "well-behaving programs" (i. e. those that respect this principle) it will be sufficient to dramatically speed up the address translation.

Since the TLA is part of the memory management unit, it will be used automatically by the CPU; no specific programming is necessary to activate or use it.

Note that the page $\mapsto$ page frame mapping exists for every single process in the system: Since each process has its own virtual memory space, it makes no sense to combine their page tables in some kind of system-wide table. This has consequences for the TLA as well: If it, as described so far, only stores page and frame numbers, then every context switch to another process will invalidate all its entries. So if the scheduler switches processes very often, this will limit the use of the TLA. Alternatively the TLA could be constructed in a way that maps (process id, page number) pairs to frames: That would keep all entries valid across context switches, but with different processes always accessing different page frames, it would only work well in a setup with either very few processes or with a sufficiently increased TLA size.

6.2.5 Caching for Filesystems

As a TLA increases virtual memory address translation, so does caching file access. However when implementing caching, filesystems typically do not differentiate between disk blocks that contain administrative data (i. e. the general file info blocks and indirection blocks containing addresses) and regular data blocks; instead all typed of blocks are stored in the cache. We will not go into the details of caching (and possible caching strategies), but just point out the similarity between caching disk blocks and using a TLA.

6.2.6 Page Replacement Strategies

When memory gets full, eventually the system will have to move pages to the disk in order to make room for other processes' memory demands. Paging out a page (i. e. writing it to disk and releasing the page frame that held the page) and assigning a different process' page to this page frame is called page replacement. The algorithm that decides which page to page out is called a page replacement algorithm, and it implements a page replacement strategy. The chosen strategy is a part of the memory management system's design, and there are several choices.

One possible choice would be a random selection: Whenever there is need for a free page frame (and none available) just pick any odd page frame and page out its contents. This strategy would not be much good, but we can think of even worse ones, e. g. always pick the very first page frame in the RAM.

The selection process has no consequences on the overall functioning of memory management: Even the worst strategy (and "pick the first page frame" is a good candidate for that) will lead to a working memory management system. However, the selection process decides how efficiently the resulting system behaves.

Before going into details, let us note that there is no direct equivalent to page replacement in filesystems—unless you had another layer of the memory hierarchy that is above disk access, e. g. an automatic tape backup system with a tape robot that can write files to a tape and delete them on disk when disk space gets low. If you had such a setup, you would move from a CPU–cache–RAM–disk memory hierarchy to a CPU–cache–RAM–disk–tape one, and accessing a file currently on tape would cause something that could be called a file access fault, resulting in the system automatically fetching the file back from tape (and keeping the requesting process blocked during all the time until the file becomes available again). Strategies for deciding which files to temporarily transfer from the disk to a tape would be called a file replacement strategy and be somewhat similar to the page replacement strategies. Distributed filesystems (or "distributed network filesystems") that allow files to either exist on a local machine or on a remote host's disk do something similar if they make file access transparent, no matter whether things are stored locally or remotely. We will not look any further into this. A true analog of page replacement would operate on disk block level, i. e. remove individual blocks from the disk in order to store them elsewhere, and that is something that does not make much sense since files are typically accessed fully when they are accessed at all. It might however make sense to keep the first block of a file on disk when removing the file, because often only the first block of a file is read in order to find out its filetype (think of

"magic numbers").

A way of measuring a page replacement strategy is the average number of page faults that it causes. It is not possible to truly calculate this number, because it depends on so many things, e. g.:

- The absolute memory demands depend on all the processes currently running on a system.
- Even if sample situations (test cases) are created that consist of predefined processes with fixed start times and memory requirements (such as: process will access its page number n at instruction i) it is not possible to predict when precisely this process will execute this instruction—scheduling the processes will always result in slightly different orders of execution each time the test case is run.

So all we can do is think of theoretical properties of replacements strategies and, when implementing a strategy, observe its effetcs on a number of test cases which are tested several times in order to calculate an average number of page faults for each test case. Looking at the design of a strategy will however allow us to make some principle predictions.

6.2.6.1 FIFO Page Replacement

A simple approach to page replacement is using a FIFO (first in, first out) list that keeps record of pages as they come into memory (either by being newly created, e. g. because a new process was started, or by being brought back in from disk after they had been swapped out earlier). The list can grow up to a size that is determined by the number of available page frames in the system's memory. When this limit is reached, the list will be chopped from the top: The page that is first in the list is removed and paged out. If the owning process tries to access this (paged-out) page again, a page fault occurs, and the memory manager has to page it back in, adding it at the end of the FIFO list.

This approach is simple because administering a FIFO list is simple, and selecting the next page to be paged out only requires reading the list head and removing it. However it has the problem of totally ignoring that some pages are accessed much more frequently than others. All pages travel from the list end to the list head at equal speed as pages are continuously paged out and back in, and for constantly and frequently used pages this means they will be paged out and in very often. It would make sense to be informed about the access frequency and keep the more frequently used pages in memory all the time, resulting in a much increased overall performance (with less page faults).

6.2.6.2 Second Chance Algorithm

An attempt to bring the frequency of page access into the FIFO strategy is the introduction of a "second chance": The idea to set an access bit for a page each time it is accessed by its owning process. This is something that the MMUs of most processors can do automatically—which is important because it reduces the necessary efforts of the memory management system.

The modification of the FIFO strategy is the following:

- A simple FIFO list of all pages works in principle as in the FIFO case.
- The MMU sets bits for each page access, as described above.
- When a page frame has to be freed, the system looks at the list entry (as before). If the page at the list end has its access bit set, it is *not* paged out, but instead moved to the list head, and its access bit is cleared: it gets a second chance.

So the second chance algorithm selects the page from the subset of pages with unset access bits that is the last in the FIFO list. Not using the chance then means that after being spared when first found at the list end, it will travel all the way from list head to list end without being accessed another time. Then the memory manager will page it out.

6.2.6.3 Linux' Page Replacement Strategy

Linux uses a modified LRU strategy:

> The key question faced by the swapping subsystem is always the same. Which pages can be swapped out to ensure maximum benefits at minimum cost to the system? The kernel uses a mixture of the ideas discussed earlier and implements a rough-grained LRU method that makes use of only one hardware feature – the setting of an accessed bit following a page access – because this function is available on all supported architectures and can be emulated with little effort.
>
> In contrast to the general algorithms, the LRU implementation of the kernel is based on two linked lists that are referred to as the active and the inactive list (separate lists exist for each memory zone in the system). As the two names imply, all the pages in active use are on the one list, while all inactive pages that may be mapped into one or more processes but are not very frequently used are held on the other. To distribute the pages between the lists, the kernel performs a regular balancing operation that determines – by means of the above accessed bit – whether a page is

> regarded as active or inac- tive, in other words, whether or not it is frequently accessed by the applications in the system. Transfers between the two lists are possible in both directions. Pages can be transferred from active to inactive and vice versa. However, transfer does not take place after every single page access but at longer intervals.
>
> In the course of time, the least frequently used pages collect at the end of the inactive list. When there is a memory shortage, the kernel selects these pages for swap-out. Since these pages have been little used so far, the LRU principle dictates that this will prove least disruptive to system operation.

(quoted from Wolfgang Mauerer, Linux Kernel Architecture, p. 1029)

6.2.7 Organization of Swap Space

Some operating systems to not page out individual pages, but several contiguous pages, for example Linux calls these *clusters*, typically consisting of 256 pages that are contiguous in the sense that they represent a contiguous set of virtual memory addresses from the process' linear view on memory.

Ideally these clusters are stored contiguously in the swap space[3] When re-paging in a page this contiguous storage is useful because it allows the system to quickly read more than just the one page requested (assuming that the following pages are going to be requested in the near future as well). If the (logically) contiguous pages were stored at random positions in the swap space, paging in such a set of pages would require several seek operations on the disk (and they slow down this process).

[3] Note that we call the disk areas used for storing paged-out memory pages "swap space" despite our earlier explanation that it is in fact not swap space because it is not used by a swapping mechanism but by a paging mechanism. We just follow the general use of calling this swap space.