Chapter 5

Memory Management

Primary memory is a prime resource in computer systems. Its management is an important function of operating systems. However, in this area the operating system depends heavily on hardware support, and the policies and data structures that are used are dictated by the support that is available.

Actually, three players are involved in handling memory. The first is the compiler, which structures the address space of an application. The second is the operating system, which maps the compiler's structures onto the hardware. The third is the hardware itself, which performs the actual accesses to memory locations.

5.1 Mapping Memory Addresses

Let us first review what the memory is used for: it contains the context of processes. This is typically divided into several segments or regions.

Different parts of the address space have different uses

The address space is typically composed of regions with the following functionalities:

Text segment — this is the code, or machine instructions, of the application being executed, as created by the compiler. It is commonly flagged as read-only, so that programs cannot modify their code during execution. This allows such memory to be shared among a number of processes that all execute the same program (e.g. multiple instances of a text editor).

Data segment — This usually contains predefined data structures, possibly initialized before the program starts execution. Again, this is created by the compiler.

Heap — this is an area in memory used as a pool for dynamic memory allocation. This is useful in applications that create data structures dynamically, as is common in object-oriented programming. In C, such memory is acquired using the

`malloc` library routine. The implementation of `malloc`, in turn, makes requests to the operating system in order to enlarge the heap as needed. In Unix, this is done with the `brk` system call.

Stack — this is the area in memory used to store the execution frames of functions called by the program. Such frames contain stored register values and space for local variables. Storing and loading registers is done by the hardware as part of the instructions to call a function and return from it (see Appendix A). Again, this region is enlarged at runtime as needed.

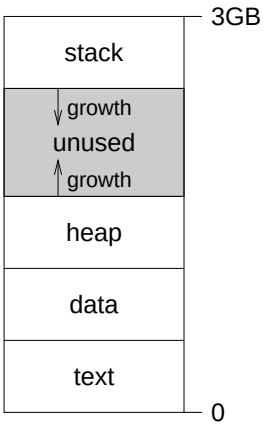
In some systems, there may be more than one instance of each of these regions. For example, when a process includes multiple threads, each will have a separate stack. Another common examle is the use of dynamically linked libraries; the code for each such library resides in a separate segment, that — like the text segment — can be shared with other processes. In addition, the data and heap may each be composed of several independent segments that are acquired at different times along the execution of the application. For example, in Unix it is possible to create a new segment using the `shmget` system call, and then multiple processes may attach this segment to their address spaces using the `shmat` system call.

These different parts have to be mapped to addresses

The compiler creates the text and data segments as relocatable segments, that is with addresses from 0 to their respective sizes. Text segments of libraries that are used by the program also need to be included. The heap and stack are only created as part of creating a process to execute the program. All these need to be mapped to the process address space.

The sum of the sizes of the memory regions used by a process is typically much smaller than the total number of addresses it can generate. For example, in a 32-bit architecture, a process can generate addresses ranging from 0 to $2^{32}-1 = 4,294,967,295$, which is 4 GB, of which say 3GB are available to the user program (the remainder is left for the system, as described in Section 9.7.1). The used regions must be mapped into this large virtual address space. This mapping assigns a virtual address to every instruction and data structure. Instruction addresses are used as targets for branch instructions, and data addresses are used in pointers and indexed access (e.g. the fifth element of the array is at the address obtained by adding four element sizes to the the array base address).

Mapping static regions, such as the text, imposes no problems. The problem is with regions that may grow at runtime, such as the heap and the stack. These regions should be mapped so as to leave them ample room to grow. One possible solution is to map the heap and stack so that they grow towards each other. This allows either of them to grow much more than the other, without having to know which in advance. No such solution exists if there are multiple stacks.



Exercise 105 *Is it possible to change the mapping at runtime to increase a certain segment's allocation? What conditions must be met? Think of what happens if the code includes statements such as "x = &y;".*

Exercise 106 *Write a program which calls a recursive function that declares a local variable, allocates some memory using malloc, and prints their addresses each time. Are the results what you expected?*

And the virtual addresses need to be mapped to physical ones

The addresses generated by the compiler are *relative* and *virtual*. They are relative because they assume the address space starts from address 0. The are virtual because they are based on the assumption that the application has all the address space from 0 to 3GB at its disposal, with a total disregard for what is physically available. In practice, it is necessary to map these compiler-generated addresses to physical addresses in primary memory, subject to how much memory you bought and contention among different processes.

In bygone days, dynamic memory allocation was not supported. The size of the used address space was then fixed. The only support that was given was to map a process's address space into a contiguous range of physical addresses. This was done by the *loader*, which simply set the *base address register* for the process. Relative addresses were then interpreted by adding the base address to each of them.

Today, paging is used, often combined with segmentation. The virtual address space is broken into small, fixed-size pages. These pages are mapped independently to frames of physical memory. The mapping from virtual to physical addresses is done by special hardware at runtime, as described below in Section 5.3.

5.2 Segmentation and Contiguous Allocation

In general, processes tend to view their address space (or at least each segment of it) as contiguous. For example, in C it is explicitly assumed that array elements will

reside one after the other, and can therefore be accessed using pointer arithmetic. The simplest way to support this is to indeed map contiguous segments of the address space to sequences of physical memory locations.

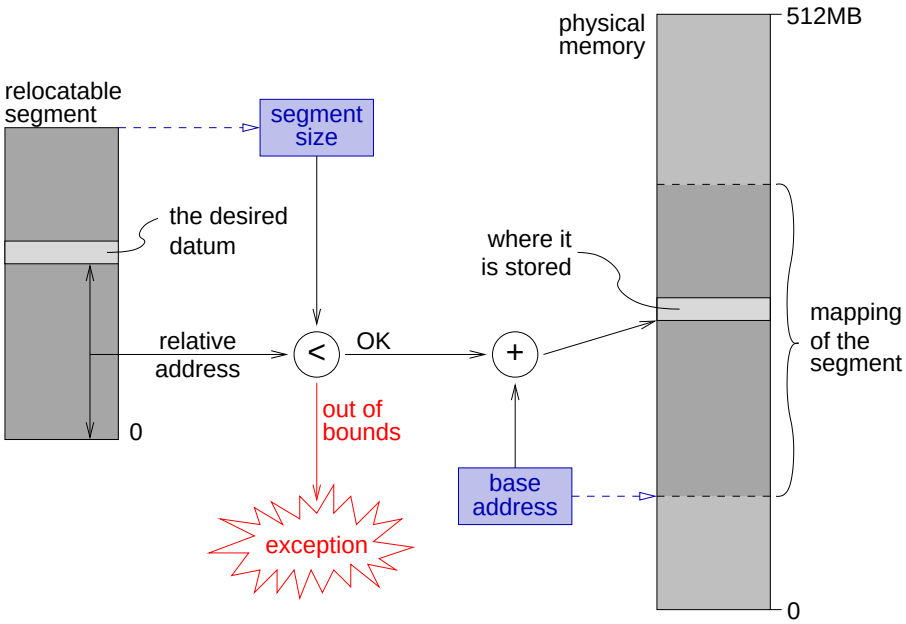
5.2.1 Support for Segmentation

Segments are created by the programmer or compiler

Segments are arbitrarily-sized contiguous ranges of the address space. They are a tool for structuring the application. As such, the programmer or compiler may decide to arrange the address space using multiple segments. For example, this may be used to create some data segments that are shared with other processes, while other segments are not.

Segments can be mapped at any address

As explained above, compilers typically produce *relocatable code*, with addresses relative to the segment's base address. In many systems, this is also supported by hardware: the memory access hardware includes a special architectural register[1] that is loaded with the segment's base address, and this is automatically added to the relative address for each access. In addition, another register is loaded with the size of the segment, and this value is compared with the relative address. If the relative address is out of bounds, a memory exception is generated.



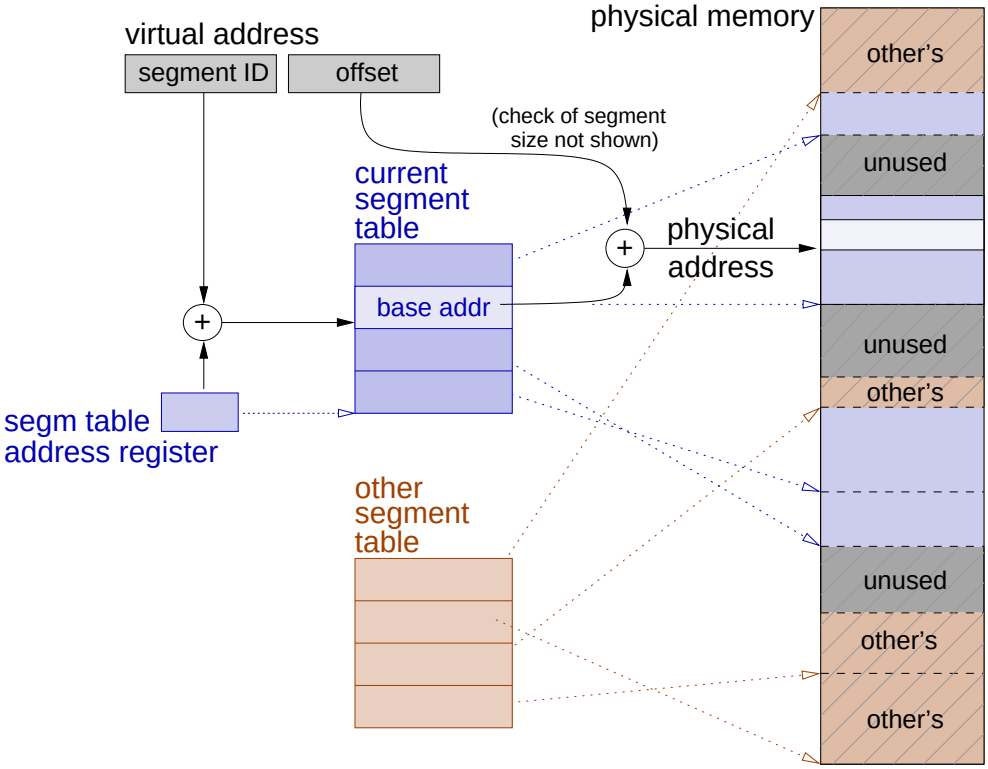
[1]This means that this register is part of the definition of the architecture of the machine, i.e. how it works and what services it provides to software running on it. It is not a general purpose register used for holding values that are part of the computation.

Exercise 107 *What are the consequences of not doing the bounds check?*

Multiple segments can be supported by using a table

Using a special base registers and a special length register is good for supporting a single contiguous segment. But as noted above, we typically want to use multiple segments for structuring the address space. To enable this, the segment base and size values are extracted from a segment table rather than from special registers. The problem is then which table entry to use, or in other words, to identify the segment being accessed. To do so an address has to have two parts: a segment identifier, and an offset into the segment. The segment identifier is used to index the segment table and retrieve the segment base address and size, and these can then be used to find the physical address as described above.

Exercise 108 *Can the operating system decide that it wants to use multiple segments, or does it need hardware support?*



Once we have multiple segments accessed using a table, it is an easy step to having a distinct table for each process. Whenever a process is scheduled to run, its table is identified and used for memory accesses. This is done by keeping a pointer to the table in the process's PCB, and loading this pointer into a special register when the process is scheduled to run. This register is part of the architecture, and serves as the basis of

the hardware address translation mechanism: all translations start with this register, which points to the segment table to use, which contains the data regarding the actual segments. As part of each context switch the operating system loads this register with the address of the new process's segment table, thus effectively switching the address space that will be used.

Note that using such a register to point to the segment table also leads to the very important property of memory protection: when a certain process runs, only its segment table is available, so only its memory can be accessed. The mappings of segments belonging to other processes are stored in other segment tables, so they are not accessible and thus protected from any interference by the running process.

The contents of the tables, i.e. the mapping of the segments into physical memory, is done by the operating system using algorithms described next.

5.2.2 Algorithms for Contiguous Allocation

Assume that segments are mapped to contiguous ranges of memory. As jobs are loaded for execution and then terminate, such ranges are allocated and de-allocated. After some time, the memory will be divided into many allocated ranges, separated by unallocated ranges. The problem of allocation is just one of finding a contiguous range of free memory that is large enough for the new segment being mapped.

First-fit, best-fit, and next-fit algorithms just search

The simplest data structure for maintaining information about available physical memory is a linked list. Each item in the list contains the start address of a free range of memory, and its length.

Exercise 109 *The two operations required for managing contiguous segments are to allocate and de-allocate them. Write pseudo-code for these operations using a linked list of free ranges.*

When searching for a free area of memory that is large enough for a new segment, several algorithms can be used:

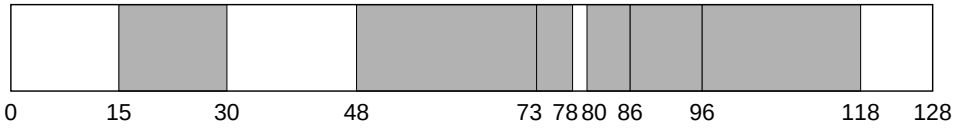
First-fit scans the list from its beginning and chooses the first free area that is big enough.

Best-fit scans the *whole* list, and chooses the smallest free area that is big enough. The intuition is that this will be more efficient and only waste a little bit each time. However, this is wrong: first-fit is usually better, because best-fit tends to create multiple small and unusable fragments [10, p. 437].

Worst-fit is the opposite of best-fit: it selects the biggest free area, with the hope that the part that is left over will still be useful.

Next-fit is a variant of first-fit. The problem is that first-fit tends to create more fragmentation near the beginning of the list, which causes it to take more time to reach reasonably large free areas. Next-fit solves this by starting the next search from the point where it made the previous allocation. Only when the end of the list is reached does it start again from the beginning.

Exercise 110 *Given the following memory map (where gray areas are allocated), what was the last segment that was allocated assuming the first-fit algorithm was used? and what if best-fit was used?*

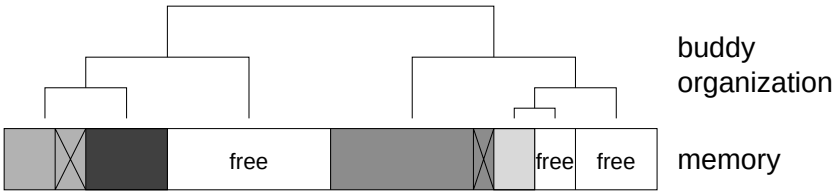


Exercise 111 *Given a certain situation (that is, memory areas that are free and allocated), and a sequence of requests that can be satisfied by first-fit, is it always true that these requests can also be satisfied by best-fit? How about the other way around? Prove these claims or show counter examples.*

To read more: There is extensive literature regarding the detailed analysis of these and other packing algorithms in an off-line setting. See Coffman et al. for a good survey [2].

Buddy systems use predefined partitions

The complexity of first-fit and best-fit depends on the length of the list of free areas, which becomes longer with time. An alternative with constant complexity is to use a buddy system.



With a buddy system, memory is partitioned according to powers of two. Each request is rounded up to the nearest power of two (that is, the size of the address space that is allocated is increased — represented by an area marked with an X in the figure). If a block of this size is available, it is allocated. If not, a larger block is split repeatedly into a pair of buddies until a block of the desired size is created. When a block is released, it is re-united with its buddy if the buddy is free.

> **Overhead Vs. Utilization**
> The complexities of the various algorithms differ: best fit is linear in the number of free ranges, first fit is also linear but with a constant smaller than one, and buddy systems

are logarithmic. On the other hand, the different algorithms achieve different levels of memory utilization. The question is then which effect is the dominant one: is it worth while to use a more sophisticated algorithm to push up the utilization at the price of more overhead, or is it better to forgo some utilization and also reduce overhead?

In order to answer such questions it is necessary to translate these two metrics into the same currency. A promising candidate is the effect on the performance of user applications. With this currency, it is immediately evident that algorithms with a higher complexity are detrimental, because time spent running the algorithm is not spent running user programs. Likewise, an inefficient algorithm that suffers from fragmentation and reduced memory utilization may cause user processes to have to wait for memory to become available. We can use detailed simulations to assess whether the more sophisticated algorithm manages to reduce the waiting time enough to justify its cost, or whether it is the other way round. The results will depend on the details of the workload, such as the distribution of job memory requirements.

Fragmentation is a problem

The main problem with contiguous allocation is fragmentation: memory becomes split into many small pieces that cannot be used effectively. Two types of fragmentation are distinguished:

- *Internal fragmentation* occurs when the system allocates more than the process requested and can use. For example, this happens when the size of the address space is increased to the next power of two in a buddy system.
- *External fragmentation* occurs when unallocated pieces of memory are left between allocated areas, and these unallocated pieces are all too small to satisfy any additional requests (even if their sum may be large enough).

Exercise 112 *Does next-fit suffer from internal fragmentation, external fragmentation, or both? And how about a buddy system?*

One solution to the problem of fragmentation is *compaction*: relocate the various segments so as to accumulate all the free space in one place. This will hopefully be large enough for additional allocations. However this suffers from considerable overhead. A better solution is to use paging rather than contiguous allocation.

Exercise 113 *Does compaction solve internal fragmentation, external fragmentation, or both?*

Allocations from the heap may use caching

While operating systems typically use paging rather than contiguous allocation, the above algorithms are by no means obsolete. For example, allocation of memory from

the heap is done by similar algorithms. It is just that the usage has shifted from the operating system to the runtime library.

Modern algorithms for memory allocation from the heap take usage into account. In particular, they assume that if a memory block of a certain size was requested and subsequently released, there is a good chance that the same size will be requested again in the future (e.g. if the memory is used for a new instance of a certain object). Thus freed blocks are kept in lists in anticipation of future requests, rather than being merged together. Such reuse reduces the creation of additional fragmentation, and also reduces overhead.

5.3 Paging and Virtual Memory

The instructions and data of a program have to be in main memory for the CPU to use them. However, most applications typically use only a fraction of their instructions and data at any given time. Therefore it is possible to keep only the needed parts in memory, and put the rest on disk. This decouples the memory as seen and used by the application from its physical implementation, leading to the concept of "virtual memory".

Technically, the allocation and mapping of virtual memory is done in fixed-size units called *pages*. The activity of shuttling pages to the disk and back into main memory is called *paging*. It is used on practically all contemporary general purpose systems.

To read more: Jacob and Mudge provide a detailed survey of modern paging schemes and their implementation [8]. This doesn't exist in most textbooks on Computer Architecture.

5.3.1 The Concept of Paging

Paging provides the ultimate support for virtual memory. It not only decouples the addresses used by the application from the physical memory addresses of the hardware, but also decouples the amount of memory used by the application from the amount of physical memory that is available.

Paging works by shuttling pages of memory between the disk and physical memory, as needed

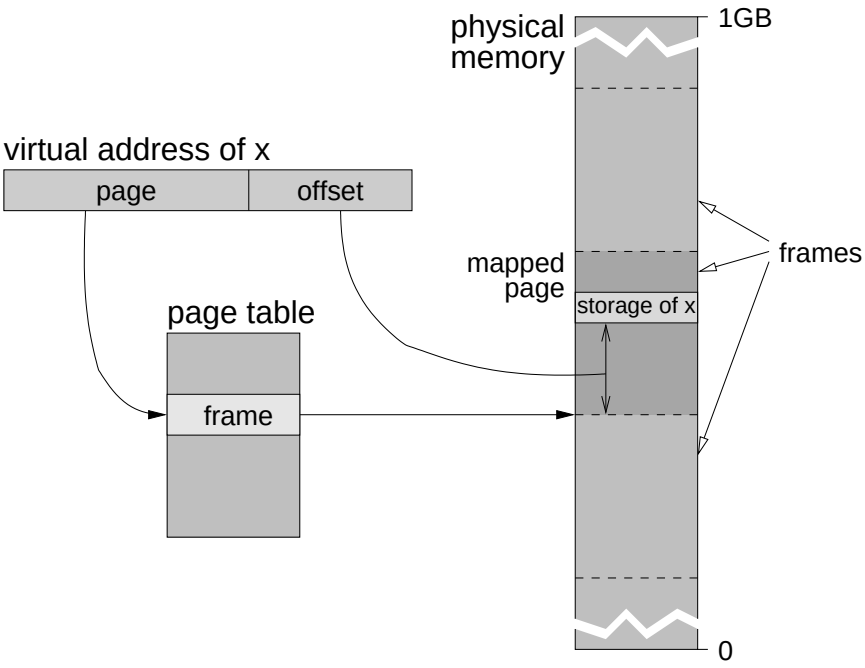
The basic idea in paging is that memory is mapped and allocated in small, fixed size units (the pages). A typical page size is 4 KB. Addressing a byte of memory can then be interpreted as being composed of two parts: selecting a page, and specifying an offset into the page. For example, with 32-bit addresses and 4KB pages, 20 bits indicate the page, and the remaining 12 identify the desired byte within the page. In mathematical notation, we have

$$\text{page} = \left\lfloor \frac{\text{address}}{4096} \right\rfloor$$

$$\text{offset} = \text{address} \bmod 4096$$

(but note that the hardware does not need to perform mathematical opeerations — it just selects a subset of the bits!) Using bit positions like this leads to page sizes that are powers of two, and allows for simple address translation in hardware.

The operating system maps pages of virtual addresses to *frames* of physical memory (of the same size). The mapping is kept in the *page table*. When the CPU wants to access a certain virtual address (be it the next instruction or a data item), the hardware uses the page table to translate the virtual page number to a physical memory frame. It then accesses the specified byte in that frame.



If a page is not currently mapped to a frame, the access fails and generates a *page fault*. This is a special type of interrupt. The operating system handler for this interrupt initiates a disk operation to read the desired page, and maps it to a free frame. While this is being done, the process is blocked. When the page becomes available, the process is unblocked, and placed on the ready queue. When it is scheduled, it will resume its execution with the instruction that caused the page fault — but this time it will work. This manner of getting pages as they are needed is called *demand paging*.

Exercise 114 *Does the page size have to match the disk block size? What are the considerations?*

A possible cloud on this picture is that there may be no free frames to accommodate the page. Indeed, the main issue in paging systems is choosing pages to page out in order to make room for other pages.

> **Virtual and Physical Addresses**
> Understanding the meaning of virtual and physical addresses is crucial to understand paging, so we'll give an analogy to drive the definition home. In a nutshell, a virtual or logical address is the name you use to refer to something. The physical address is the location where this something can be found.
>
> This distinction doesn't always make sense in real life. For example, when referring to the building at 123 Main Street, the string "123 Main Street" serves both as its name and as an indication of its location.
>
> A better example for our purpose is provided by articles in the scientific literature. For example, we may have the citation "*Computer* Magazine, volume 23, issue 5, page 65". This is a logical address, or name, of where the article was published. The first part, "*Computer* Magazine", is like the page number. Your librarian (in the capacity of an operating system) will be able to translate it for you into a physical address: this is the journal in the third rack on the left, second shelf from the top. The rest, "volume 23, issue 5, page 65", is an offset into the journal. Using the physical address you got, you go to the shelf and find the desired article in the journal.

5.3.2 Benefits and Costs

Paging provides improved flexibility

When memory is partitioned into pages, these pages can be mapped into unrelated physical memory frames. Pages that contain contiguous virtual addresses *need not* be contiguous in physical memory. Moreover, all pages and all frames are of the same size, so any page can be mapped to any frame. As a result the allocation of memory is much more flexible.

Paging relieves the constraints of physical availability

An important observation is that not all the used parts of the address space need to be mapped to physical memory simultaneously. Only pages that are accessed in the current phase of the computation are mapped to physical memory frames, as needed. Thus programs that use a very large address space may still be executed on a machine with a much smaller physical memory.

Exercise 115 *What is the maximal number of pages that may be required to be memory resident (that is, mapped to physical frames) in order to execute a single instruction?*

Paging provides reduced fragmentation

Because page sizes match frame sizes, external fragmentation is eliminated. Internal fragmentation is also small, and occurs only because memory is allocated in page units. Thus when memory is allocated for a segment of B bytes, this will require a total of $\lceil B/P \rceil$ pages (where P is the page size). In the last page, only $B \bmod P$ bytes will be used, and the rest wasted. On average, this will lead to a waste of half a page per segment.

An obvious way to reduce the internal fragmentation is to reduce the page size. However, it is best not to make pages too small, because this will require larger page tables, and will also suffer from more page faults.

Exercise 116 *Can the operating system set the page size at will?*

It depends on locality

The success of paging systems depends on the fact that applications display *locality of reference*. This means that they tend to stay in the same part of the address space for some time, before moving to other remote addresses. With locality, each page is used many times, which amortizes the cost of reading it off the disk. Without locality, the system will *thrash*, and suffer from multiple page faults.

Luckily, this is a good assumption. The vast majority of applications do indeed display a high degree of locality. In fact, they display locality at several levels. For example, locality is also required for the efficient use of caches. Indeed, paging may be viewed as a coarse-grain version of caching, where physical memory frames are used to cache pages of the virtual address space. This analogy is shown by the following table:

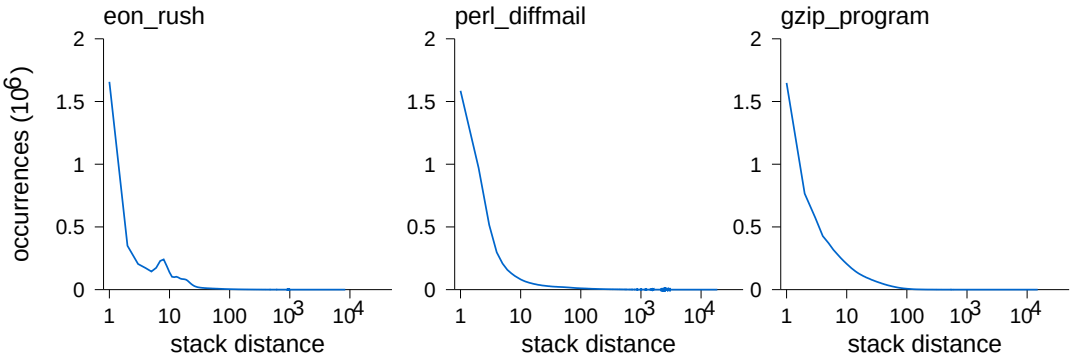
	fast storage	*slow storage*	*transfer unit*
processor cache	cache	primary memory	cache line
paging	primary memory	disk	page

However, one difference is that in the case of memory the location of pages is fully associative: any page can be in any frame, as explained below.

Exercise 117 *The cache is maintained by hardware. Why is paging delegated to the operating system?*

> **Detail: measuring locality**
> Locality can be measure using the "average stack depth" where data would be found, if all data items were to be kept in a stack. The idea is as follows. Assume everything is kept in one large stack. Whenever you access a new datum, one that you have not accessed before, you also deposit it on the top of the stack. But when you re-access a datum, you need to find it in the stack. Once it is found, you note its depth into the stack, and then you move it from there to the top of the stack. Thus the order of items in the stack reflects the order in which they were last accessed.

But how does this measure locality? If memory accesses are random, you would expect to find items "in the middle" of the stack. Therefore the average stack depth will be high, roughly of the same order of magnitude as the size of all used memory. But if there is significant locality, you expect to find items near the top of the stack. For example, if there is significant temporal locality, it means that we tend to repeatedly access the same items. If such repetitions come one after the other, the item will be found at the top of the stack. Likewise, if we repeatedly access a set of items in a loop, this set will remain at the top of the stack and only the order of its members will change as they are accessed.



experimental results from several SPEC 2000 benchmarks, shown in these graphs, confirm that they have very strong locality. Low stack depths, typically not much more than 10, appear hundreds of thousands of times, and account for the vast majority of references. Only in rare cases an item is found deep into the stack.

Another measure of locality is the size fo the *working set*, loosely defined as the set of addresses that are needed at a particular time [6]. The smaller the set (relative to the full set of all possible addresses), the stronger the locality. This may change for each phase of the computation.

It also depends on the memory/disk cost ratio

Another underlying assumption in paging systems is that disk storage costs less than primary memory. That is why it is worth while to store most of the data on (slower) disks, and only keep the working set in (fast CPU-accessible) memory. For the time being, this is a safe assumption. While the price of memory has been dropping continuously, so has that of disks. But it is possible that in the future memory will become cheap enough to call the use of paging into question [7, sect. 2.4].

Exercise 118 *So when memory becomes cheaper than disk, this is the end of paging? Or are there reasons to continue to use paging anyway?*

The cost is usually very acceptable

So far we have only listed the benefits of paging. But paging may cause applications to *run slower*, due to the overheads involved in interrupt processing. This includes

both the direct overhead of actually running the interrupt handler, and the indirect overhead of reduced cache performance due to more context switching, as processes are forced to yield the CPU to wait for a page to be loaded into memory. The total effect may be a degradation in performance of 10–20% [9].

To improve performance, it is crucial to reduce the page fault rate. This may be possible with good prefetching: for example, given a page fault we may bring additional pages into memory, instead of only bringing the one that caused the fault. If the program subsequently references these additional pages, it will find them in memory and avoid the page fault. Of course there is also the danger that the prefetching was wrong, and the program does not need these pages. In that case it may be detrimental to bring them into memory, because they may have replaced other pages that are actually more useful to the program.

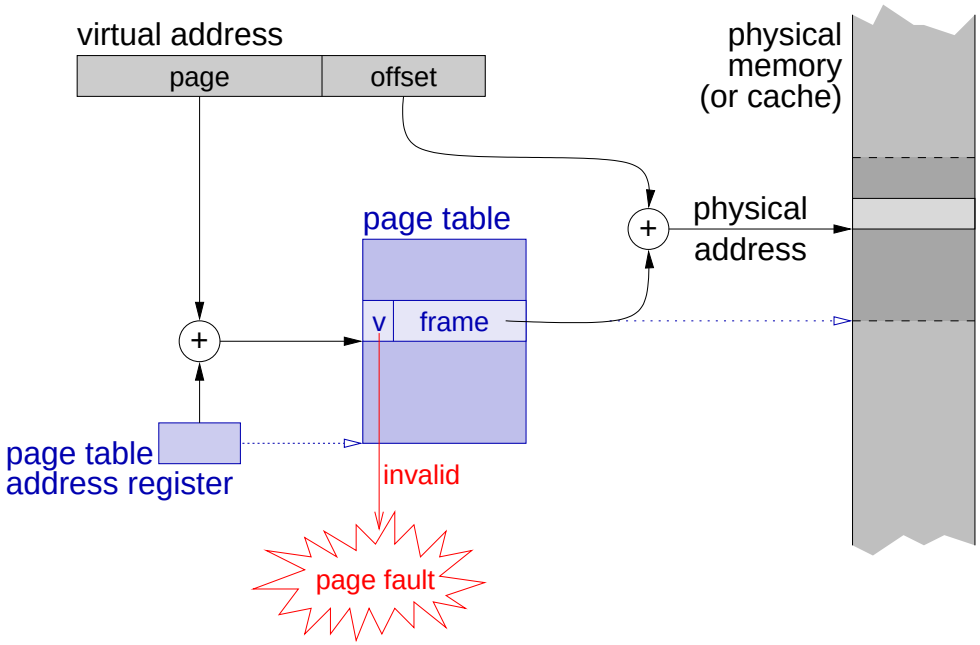
5.3.3 Address Translation

As mentioned above, paging hinges on hardware support for memory mapping that allows pages to be mapped to any frame of memory.

Address translation is done by the hardware, not the operating system

Essentially every instruction executed by the CPU requires at least one memory access — to get the instruction. Oftentimes it also requires 1 to 3 additional accesses to fetch operands and store the result. This means that memory access must be fast, and obviously cannot involve the operating system. Therefore translating virtual addresses to physical addresses must be done by the hardware.

Schematically, the translation is done as follows.



The page part of the virtual address is used as an index into the page table, which contains the mapping of pages to frames (this is implemented by adding the page number to a register containing the base address of the page table). The frame number is extracted and used as part of the physical address. The offset is simply appended. Given the physical address, an access to memory is performed. This goes through the normal mechanism of checking whether the requested address is already in the cache. If it is not, the relevant cache line is loaded from memory.

To read more: Caching is covered in all textbooks on computer architecture. It is largely orthogonal to the discussion here.

The page table also contains some additional bits of information about each page, including

- A *valid bit* (also called *present bit*), indicating whether the page is assigned to a frame or not. If it is not, and some word in the page is accessed, the hardware will generate a page fault. This bit is set by the operating system when the page is mapped.
- A *modified* or *dirty bit*, indicating whether the page has been modified. If so, it has to be written to disk when it is evicted. This bit is cleared by the operating system when the page is mapped. It is set automatically by the hardware each time any word in the page is written.
- A *used bit*, indicating whether the page has been accessed recently. It is set automatically by the hardware each time any word in the page is accessed. "Recently" means since the last time that the bit was cleared by the operating system.
- *Access permissions*, indicating whether the page is read-only or read-write. These are set by the operating system when the page is mapped, and may cause the hardware to trigger an exception if an access does not match the permissions.

We discuss the use of these bits below.

Useful mappings are cached in the TLB

One problem with this scheme is that the page table can be quite big. In our running example, there are 20 bits that specify the page, for a total of $2^{20} = 1048576$ pages in the address space. This is also the number of entries in the page table. Assuming each one requires 4 bytes, the page table itself uses 1024 pages of memory, or 4 MB. Obviously this cannot be stored in hardware registers. The solution is to have a special cache that keeps information about recently used mappings. This is again based on the assumption of locality: if we have used the mapping of page X recently, there is a high probability that we will use this mapping many more times (e.g. for other addresses that fall in the same page). This cache is called the *translation lookaside buffer* (TLB).

With the TLB, access to pages with a cached mapping can be done immediately. Access to pages that do not have a cached mapping requires two memory accesses: one to get the entry from the page table, and another to the desired address.

Note that the TLB is separate from the general data cache and instruction cache. Thus the translation mechanism only needs to search this special cache, and there are no conflicts between it and the other caches. One reason for this separation is that the TLB includes some special hardware features, notably the used and modified bits described above. These bits are only relevant for pages that are currently being used, and must be updated by the hardware upon each access.

Inverted page tables are smaller

The TLB may make access faster, but it can't reduce the size of the page table. As the number of processes increases, the percentage of memory devoted to page tables also increases. This problem can be solved by using an inverted page table. Such a table only has entries for pages that are allocated to frames. Inserting pages and searching for them is done using a hash function. If a page is not found, it is inferred that it is not mapped to any frame, and a page fault is generated.

The drawback of inverted page tables is that another structure is needed to maintain information about where pages are stored on disk, if they are not mapped to a frame. With conventional page tables, the same space can be used to hold the mapping to a frame *or* the location on disk.

Exercise 119 *Another way to reduce the size of the page table is to enlarge the size of each page: for example, we can decide that 10 bits identify the page and 22 the offset, thus reducing the page table to 1024 entries. Is this a good idea?*

The operating system only handles problems

So far we have only discussed hardware support. Where does the operating system come in? Only when things don't go smoothly.

Pages representing parts of a process's virtual address space can be in one of 3 states:

- Mapped to a physical memory frame. Such pages can be accessed directly by the CPU, and do not require operating system intervention. This is the desired state.
- Backed on disk. When such a page is accessed, a page fault occurs. This is a type of interrupt. The operating system handler function then initiates a disk operation to read the page and map it to a frame of physical memory. When this is completed, the process continues from the instruction that caused the page fault.

- Not used. Such pages are not part of any used memory segment. Trying to access them causes a memory error, which causes an interrupt. In this case, the operating system handler kills the job.

This involves maintaining the mappings

When a page fault occurs, the required page must be mapped to a free frame. However, free frames are typically in short supply. Therefore a painful decision has to be made, about what page to evict to make space for the new page. The main part of the operating system's memory management component is involved with making such decisions.

In a nutshell, the division of labor is as follows: the operating system maps pages of virtual memory to frames of physical memory. The hardware uses the mapping, as represented in the page table, to perform actual accesses to memory locations.

In fact, the operating system maintains multiple mappings. If there are multiple processes, each has its own page table, that maps its address space to the frames that have been allocated to it. If a process has multiple segments, they may each have an independent page table (as shown below). Therefore the operating system has to notify the hardware which mapping is in effect at any given moment. This is done by loading a special architectural register with the address of the table that should be used. To switch to another mapping (e.g. as part of a context switch), only this register has to be reloaded.

It also allows various tricks

Controlling the mapping of pages allows the operating system to play various tricks. A few famous examples are

- Leaving unmapped regions to catch stray memory accesses.

  As noted above, access to unmapped pages causes a memory exception, which is forwarded to the operating system for handling. This can be used to advantage by leaving unmapped regions in strategic places. For example, access to local variables in functions is done by using a calculated offset from the stack pointer (SP). By leaving a few unmapped pages beyond the end of the stack, it is possible to catch erroneous accesses that extend farther than the pre-allocated space.

  Exercise 120 *Will all erroneous accesses be caught? What happens with those that are not caught?*

- Implementing the copy-on-write optimization.

  In Unix, forking a new process involves copying all its address space. However, in many cases this is a wasted effort, because the new process performs an exec

system call and runs another application instead. A possible optimization is then to use copy-on-write. Initially, the new process just uses its parent's address space, and all copying is avoided. A copy is made only when either process tries to modify some data, and even then, only the affected page is copied.

This idea is implemented by copying the parent's page table to the child, and marking the whole address space as read-only. As long as both processes just read data, everything is OK. When either process tries to modify data, this will cause a memory error exception (because the memory has been tagged read-only). The operating system handler for this exception makes separate copies of the offending page, and changes their mapping to read-write.

- Swapping the Unix u-area.

  In Unix, the u-area is a kernel data structure that contains important information about the currently running process. Naturally, the information has to be maintained somewhere also when the process is not running, but it doesn't have to be *accessible*. Thus, in order to reduce the complexity of the kernel code, it is convenient if only one u-area is visible at any given time.

  This idea is implemented by compiling the kernel so that the u-area is page aligned (that is, the data structure starts on an address at the beginning of a page). For each process, a frame of physical memory is allocated to hold its u-area. However, these frames are not mapped into the kernel's address space. Only the frame belonging to the current process is mapped (to the page where the "global" u-area is located). As part of a context switch from one process to another, the mapping of this page is changed from the frame with the u-area of the previous process to the frame with the u-area of the new process.

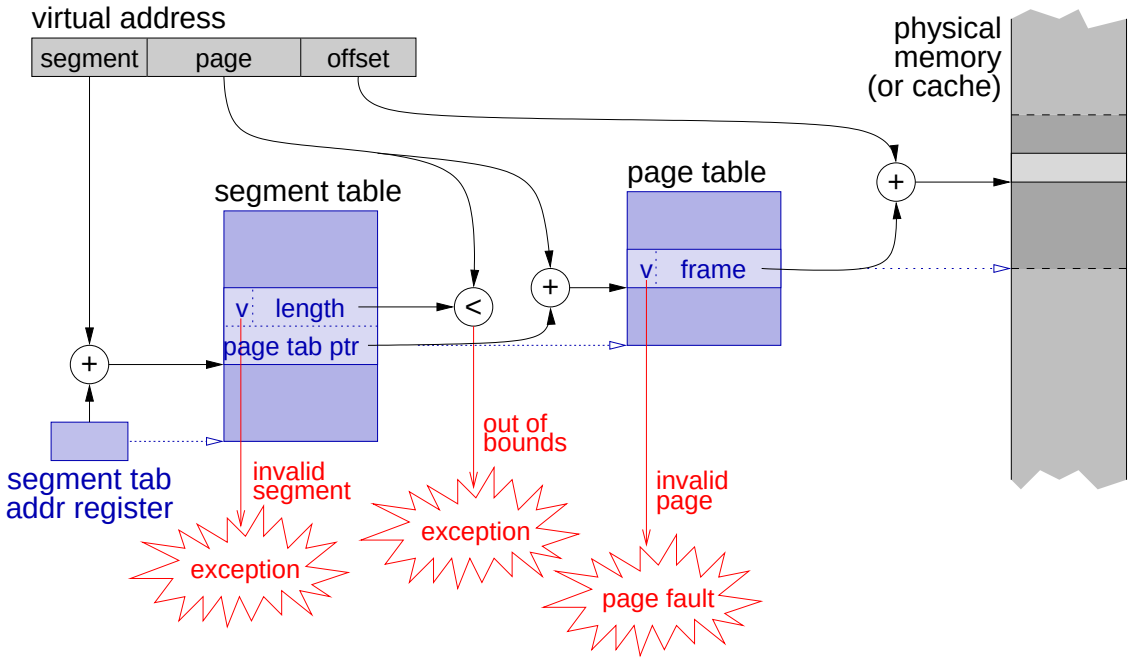
  If you understand this, you understand virtual memory with paging.

Paging can be combined with segmentation

It is worth noting that paging is often combined with segmentation. All the above can be done on a per-segment basis, rather than for the whole address space as a single unit. The benefit is that now segments do not need to be mapped to contiguous ranges of physical memory, and in fact segments may be larger than the available memory.

Exercise 121 *Does this mean that there will be no fragmentation?*

Address translation for paged segments is slightly more complicated, though, because each segment has its own page table. The virtual address is then parsed into three parts:



1. The segment number, which is used as an index into the segment table and finds the correct page table. Again, this is implemented by adding it to the contents of a special register that points to the base address of the segment table.
2. The page number, which is used as an index into the page table to find the frame to which this page is mapped. This is mapped by adding it to the base address of the segment's page table, which is extracted from the entry in the segment table.
3. An offset into the page.

As far as the compiler or programmer is concerned, addresses are expressed as a segment and offset into the segment. The system interprets this offset as a page and offset into the page. As before, the segment offset is compared with the segment length to check for illegal accesses (the drawing assumes this is done in page units). In addition, a page may be absent from memory, thereby generating a page fault.

Exercise 122 *Why is the case of an invalid segment shown as an exception, while an invalid page is a page fault?*

Exercise 123 *When segments are identified by a set of bits in the virtual address, does this impose any restrictions?*

To read more: Stallings [12, Sect. 7.3] gives detailed examples of the actual structures used in a few real systems (this was deleted in the newer edition). More detailed examples are given by Jacob and Mudge [8].

Protection is built into the address translation mechanism

An important feature of address translation for paging is that each process can only access frames that appear in its page table. There is no way that it can generate a physical address that lies in a frame that is allocated to another process. Thus there is no danger that one process will inadvertently access the memory of another.

Moreover, this protection is cheap to maintain. All the address translation starts from a single register, which holds the base address of the page table (or the segment table, in case paged segmentation is used). When the operating system decides to do a context switch, it simply loads this register with the address of the page table of the new process. Once this is done, the CPU no longer "sees" any of the frames belonging to the old process, and can only access pages belonging to the new one.

A slightly different mechanism is used by architectures that use a single page table, rather than a separate one for each process. In such architectures the operating system loads a unique address space identifier (ASID) into a special register, and this value is appended to each address before the mapping is done. Again, a process is prevented from generating addresses in the address spaces of other processes, and switching is done by changing the value of a single register.

5.3.4 Algorithms for Page Replacement

As noted above, the main operating system activity with regard to paging is deciding what pages to evict in order to make space for new pages that are read from disk.

FIFO is bad and leads to anomalous behavior

The simplest algorithm is FIFO: throw out the pages in the order in which they were brought in. You can guess that this is a bad idea because it is oblivious of the program's behavior, and doesn't care which pages are accessed a lot and which are not. Nevertheless, it is used in Windows 2000.

In fact, it is even worse. It turns out that with FIFO replacement there are cases when adding frames to a process causes *more* page faults rather than less page faults — an effect known as Belady's anomaly. While this only happens in pathological cases, it is unsettling. It is the result of the fact that the set of pages maintained by the algorithm when equipped with n frames is not necessarily a superset of the set of pages maintained when only $n-1$ frames are available.

In algorithms that rely on usage, such as those described below, the set of pages kept in n frames *is* a superset of the set that would be kept with $n-1$ frames, and therefore Belady's anomaly does not happen.

The ideal is to know the future

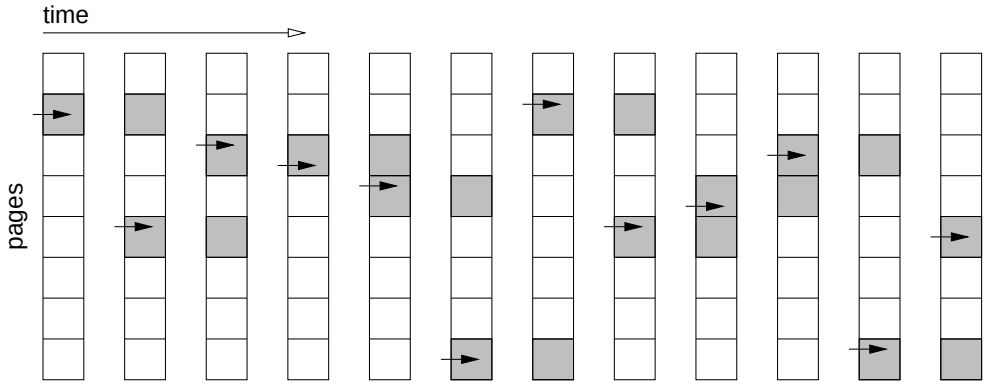
At the other extreme, the best possible algorithm is one that knows the future. This enables it to know which pages will not be used any more, and select them for replacement. Alternatively, if no such pages exist, the know-all algorithm will be able to select the page that will not be needed for the longest time. This delays the unavoidable page fault as much as possible, and thus reduces the total number of page faults.

Regrettably, such an optimal algorithm cannot be implemented. But if we do not know the future, we can at least leverage our knowledge of the past, using the principle of locality.

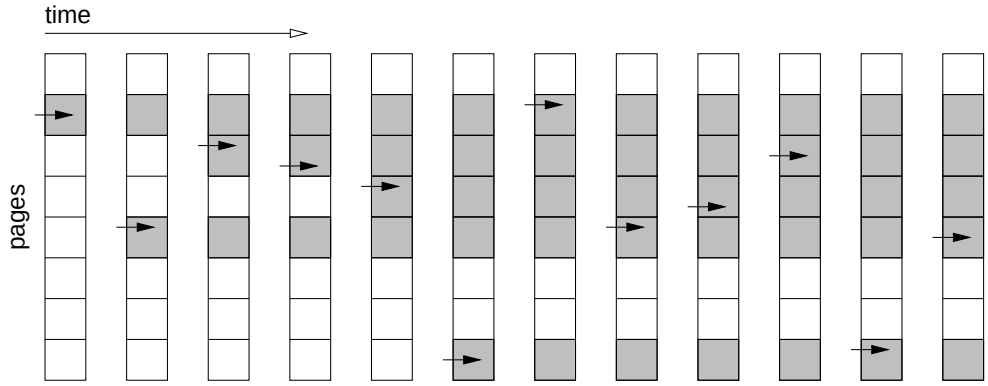
The alternative is to have each process's working set in memory

The most influential concept with regard to paging is the *working set*. The working set of a process is a dynamically changing set of pages. At any given moment, it includes the pages accessed in the last Δ instructions; Δ is called the *window size* and is a parameter of the definition.

The importance of the working set concept is that it captures the relationship between paging and the principle of locality. If the window size is too small, then the working set changes all the time. This is illustrated in this figure, where each access is denoted by an arrow, $\Delta = 2$, and the pages in the working set are shaded:



But with the right window size, the set becomes static: every additional instruction accesses a page that is already in the set. Thus the set comes to represent that part of memory in which the accesses are localized. For the sequence shown above, this requires $\Delta = 6$:



Knowing the working set for each process leads to two useful items of information:

- How many pages each process needs (the *resident set size*), and
- Which pages are not in the working set and can therefore be evicted.

However, keeping track of the working set is not realistically possible. And to be effective it depends on setting the window size correctly.

Evicting the least-recently used page approximates the working set

The reason for having the window parameter in the definition of the working set is that memory usage patterns change over time. We want the working set to reflect the current usage, not the pages that were used a long time ago. This insight leads to the LRU page replacement algorithm: when a page has to be evicted, pick the one that was least recently used (or in other words, was used farthest back in the past).

Note that LRU automatically also defines the resident set size for each process. This is so because *all* the pages in the system are considered as potential victims for eviction, not only pages belonging to the faulting process. Processes that use few pages will be left with only those pages, and lose pages that were used in the more distant past. Processes that use more pages will have larger resident sets, because their pages will never be the least recently used.

Regrettably, the LRU algorithm cannot be implemented in practice: it requires the system to know about the order in which all pages have been referenced. We need a level of simplification, and some hardware support.

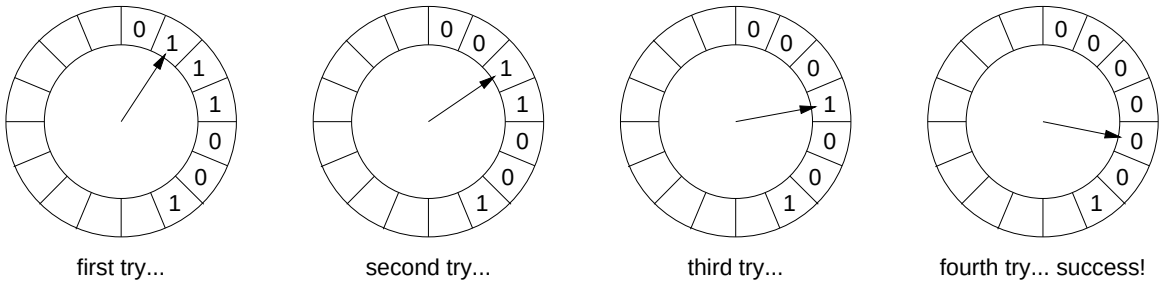
LRU can be approximated by the clock algorithm

Most computers only provide minimal hardware support for page replacement. This is done in the form of *used bits*. Every frame of physical memory is associated with a single bit. This bit is set automatically by the hardware whenever the page mapped to this frame is accessed. It can be reset by the operating system.

The way to use these bits is as follows. Initially, all bits are set to zero. As pages are accessed, their bits are set to 1 by the hardware. When the operating system

needs a frame, it scans all the pages in sequence. If it finds a page with its used bit still 0, it means that this page has not been accessed for some time. Therefore this page is a good candidate for eviction. If it finds a page with its used bit set to 1, it means that this page has been accessed recently. The operating system then resets the bit to 0, but does not evict the page. Instead, it gives the page a *second chance*. If the bit stays 0 until this page comes up again for inspection, the page will be evicted then. But if the page is accessed continuously, its bit will be set already when the operating system looks at it again. Consequently pages that are in active use will not be evicted.

It should be noted that the operating system scans the pages in a cyclic manner, always starting from where it left off last time. This is why the algorithm is called the "clock" algorithm: it can be envisioned as arranging the pages in a circle, with a hand pointing at the current one. When a frame is needed, the hand is moved one page at a time, setting the used bits to 0. When it finds a page whose bit is already 0, it stops and the page is evicted.



Exercise 124 *Is it possible to implement the clock algorithm without hardware support for used bits? Hint: remember that the operating system can turn on and off the present bit for each page.*

Using more than one bit can improve performance

The clock algorithm uses only one bit of information: whether the page was accessed since the last time the operating system looked at it. A possible improvement is for the operating system to keep a record of how the value of this bit changes over time. This allows it to differentiate pages that were not accessed a long time from those that were not accessed just now, and provides a better approximation of LRU.

Another improvement is to look at the *modify bit* too. Then unmodified ("clean") pages can be evicted in preference over modified ("dirty") pages. This saves the overhead of writing the modified pages to disk.

The algorithm can be applied locally or globally

So far, we have considered all the physical memory frames as candidates for page eviction. This approach is called *global paging*.

The alternative is *local paging*. When a certain process suffers a page fault, we only consider the frames allocated to that process, and choose one of the pages belonging to that process for eviction. We don't evict a page belonging to one process and give the freed frame to another. The advantage of this scheme is isolation: a process that needs a lot of memory is prevented from monopolizing multiple frames at the expense of other processes.

The main consequence of local paging is that it divorces the replacement algorithm from the issue of determining the resident set size. Each process has a static set of frames at its disposal, and its paging activity is limited to this set. The operating system then needs a separate mechanism to decide upon the appropriate resident set size for each process; for example, if a process does a lot of paging, its allocation of frames can be increased, but only subject to verification that this does not adversely affect other processes.

Regardless of algorithm, it is advisable to maintain a certain level of free frames

Another problem with paging is that the evicted page may be *dirty*, meaning that it was modified since being read off the disk. Dirty pages have to be written to disk when they are evicted, leading to a large delay in the service of page faults (first write one page, then read another). Service would be faster if the evicted page was clean, meaning that it was not modified and therefore can be evicted at once.

This problem can be solved by maintaining a certain level of free frames, or clean pages. Whenever the number of free frames falls below the predefined level, the operating system initiates the writing of pages to disk, in anticipation of future page faults. When the page faults indeed occur, free frames are available to serve them.

All algorithms may suffer from thrashing

All the page replacement algorithms cannot solve one basic problem. This problem is that if the sum of the sizes of the working sets of all the processes is larger than the size of the physical memory, all the pages in all the working sets cannot be in memory at the same time. Therefore every page fault will necessarily evict a page that is in some process's working set. By definition, this page will be accessed again soon, leading to another page fault. The system will therefore enter a state in which page faults are frequent, and no process manages to make any progress. This is called *thrashing*.

Thrashing is an important example of a threshold effect, where positive feedback causes system peformance to collapse: once it starts things deteriorate sharply. It is therefore imperative to employ mechanisms to prevent this and keep the system stable.

Using local paging reduces the effect of thrashing, because processes don't automatically steal pages from each other. But the only real solution to thrashing is to

reduce the memory pressure by reducing the *multiprogramming level*. This means that we will have less processes in the system, so each will be able to get more frames for its pages. It is accomplished by swapping some processes out to disk, as described in Section 5.4 below.

Exercise 125 *Can the operating system identify the fact that thrashing is occurring? how?*

To read more: There is extensive literature on page replacement algorithms. A recent concise survey by Denning tells the story of locality [5]. Working sets were introduced by Denning in the late 1960's and developed till the 1980's [3, 6, 4]. A somewhat hard-to-read early work, noted for introducing the notion of an optimal algorithm and identifying the importance of use bits, is Belady's report of research at IBM [1]. The notion of stack algorithms was introduced by Mattson et al. [11].

5.3.5 Disk Space Allocation

One final detail we did not address is how the system allocates disk space to store pages that are paged out.

A simple solution, used in early Unix systems, is to set aside a partition of the disk for this purpose. Thus disk space was divided into two main parts: one used for paging, and the other for file systems. The problem with this approach is that it is inflexible. For example, if the system runs out of space for pages, but still has a lot of space for files, it cannot use this space.

An alternative, used e.g. in Windows 2000, is to use a paging file. This is a large pre-allocated file, that is used by the memory manager to store paged out pages. Pre-allocating a certain size ensures that the memory allocator will have at least that much space for paging. But if needed, and disk space is available, this file can grow beyond its initial allocation. A maximal size can be set to prevent all disk space from being used for paging, leaving none for the file system.

5.4 Swapping

In order to guarantee the responsiveness of a computer system, processes must be preemptable, so as to allow new processes to run. But what if there is not enough memory to go around? This can also happen with paging, as identified by excessive thrashing.

Swapping is an extreme form of preemption

The solution is that when a process is preempted, its memory is also preempted. The contents of the memory are saved on secondary storage, typically a disk, and the primary memory is re-allocated to another process. It is then said that the process

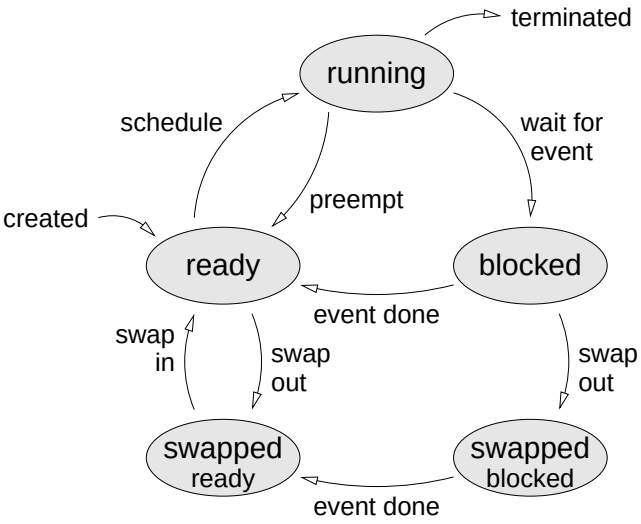
has been *swapped out*. When the system decides that the process should run again, its address space is *swapped in* and loaded into primary memory.

The decisions regarding swapping are called long-term or medium-term scheduling, to distinguish them from the decisions about scheduling memory-resident jobs. The criteria for these decisions are

- Fairness: processes that have accumulated a lot of CPU usage may be swapped out for a while to give other processes a chance.
- Creating a good job mix: jobs that compete with each other will be swapped out more than jobs that complement each other. For example, if there are two compute-bound jobs and one I/O-bound job, it makes sense to swap out the compute-bound jobs alternately, and leave the I/O-bound job memory resident.

"Swapped out" is a new process state

Swapped out processes cannot run, even if they are "ready". Thus adding swapping to a system enriches the process state graph:



Exercise 126 *Are all the transitions in the graph equivalent, or are some of them "heavier" (in the sense that they take considerably more time) ?*

And there are the usual sordid details

Paging and swapping introduce another resource that has to be managed: the disk space used to back up memory pages. This has to be allocated to processes, and a mapping of the pages into the allocated space has to be maintained. In principle, the methods described above can be used: either use a contiguous allocation with direct mapping, or divide the space into blocks and maintain an explicit map.

One difference is the desire to use large contiguous blocks in order to achieve higher disk efficiency (that is, less seek operations). An approach used in some Unix systems is to allocate swap space in blocks that are some power-of-two pages. Thus small processes get a block of say 16 pages, larger processes get a block of 16 pages followed by a block of 32 pages, and so on up to some maximal size.

5.5 Summary

Memory management provides one of the best examples in each of the following.

Abstractions

Virtual memory is one of the main abstractions supported by the operating system. As far as applications are concerned, they have large contiguous stretches of address space at their disposal. It is up to the operating system to implement this abstraction using a combination of limited, fragmented primary memory, and swap space on a slow disk.

Resource management

Memory management includes many themes related to resource management.

One is the classical algorithms for the allocation of contiguous resources — in our case, contiguous addresses in memory. These include First Fit, Best Fit, Next Fit, and the Buddy allocation scheme.

Another is the realization that it is better to break the resources into small fixed-size pieces — in this case, pages — rather than trying to allocate contiguous stretches. This is based on a level of indirection providing an associative mapping to the various pieces. With the use of such pieces comes the issue of how to free them. The most popular algorithm is LRU.

Finally, there is the distinction between implicit allocation (as in demand paging) and explicit allocation (as is needed when paging is done locally within the allocation of each process).

Needless to say, these themes are more general than just memory management.

Workload issues

Workloads typically display a high degree of locality. Without it, the use of paging to disk in order to implement virtual memory would simply not work. Locality allows the cost of expensive operations such as access to disk to be amortized across multiple memory accesses, and it ensures that these costly operations will be rare.

Hardware support

Memory management is probably where hardware support to the operating system is most prominent. Specifically, hardware address translation as part of each access is a prerequisite for paging. There are also various additions such as support for used and modify bits for each page.

Bibliography

[1] L. A. Belady, "*A study of replacement algorithms for a virtual-storage computer*". *IBM Syst. J.* **5(2)**, pp. 78–101, 1966.

[2] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson, "*Approximation algorithms for bin-packing — an updated survey*". In *Algorithm Design for Computer Systems Design*, G. Ausiello, M. Lucertini, and P. Serafini (eds.), pp. 49–106, Springer-Verlag, 1984.

[3] P. J. Denning, "*The working set model for program behavior*". *Comm. ACM* **11(5)**, pp. 323–333, May 1968.

[4] P. J. Denning, "*Working sets past and present*". *IEEE Trans. Softw. Eng.* **SE-6(1)**, pp. 64–84, Jan 1980.

[5] P. J. Denning, "*The locality principle*". *Comm. ACM* **48(7)**, pp. 19–24, Jul 2005.

[6] P. J. Denning and S. C. Schwartz, "*Properties of the working-set model*". *Comm. ACM* **15(3)**, pp. 191–198, Mar 1972.

[7] G. A. Gibson, *Redundant Disk Arrays: Reliable, Parallel Secondary Storage*. MIT Press, 1992.

[8] B. Jacob and T. Mudge, "*Virtual memory: Issues of implementation*". *Computer* **31(6)**, pp. 33–43, Jun 1998.

[9] B. L. Jacob and T. N. Mudge, "*A look at several memory management units, TLB-refill mechanisms, and page table organizations*". In 8th *Intl. Conf. Architect. Support for Prog. Lang. & Operating Syst.*, pp. 295–306, Oct 1998.

[10] D. E. Knuth, *The Art of Computer Programming. Vol 1: Fundamental Algorithms*. Addison Wesley, 2nd ed., 1973.

[11] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "*Evaluation techniques for storage hierarchies*". *IBM Syst. J.* **9(2)**, pp. 78–117, 1970.

[12] W. Stallings, *Operating Systems*. Prentice-Hall, 2nd ed., 1995.