The University of Melbourne
School of Computing and Information Systems

COMP30023
Computer Systems

Semester 1, 2017

Memory Management

Memory management

The main functions of the memory manager are

- to keep track of which parts of memory are free and which parts are allocated (and to which process),
- to allocate memory to processes when they require it,
- to protect memory against unauthorized accesses, and
- to simulate the appearance of a bigger main memory by moving data automatically between main memory and disk.

The sizes of main memories have been increasing dramatically, roughly quadrupling every three years.

However, the demands for main memory have been increasing almost as fast.

Swapping

In a multiprogramming system, the total size of all processes may exceed the size of main memory. The memory images of some processes may have to be kept on disk. The parts of the disk(s) used for this are collectively called swap space.

The kernel could *swap out* a process, i.e. copy *all* its memory to swap space, to free up some memory. It would do this to allow another process to be *swapped in*, i.e. have its image brought into memory from swap space.

Several small processes may have to be swapped out to allow one large process to be swapped in.

Swapping was the main method of memory management in the 1960s, and was one of several methods commonly used until the 1990s.

. . . we will examine variations of "swapping"

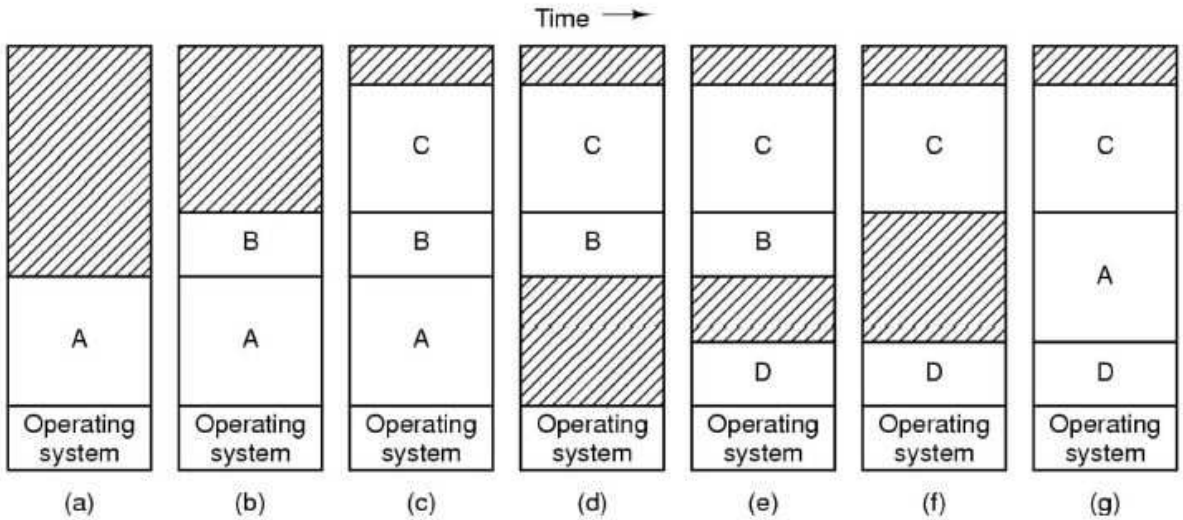
External fragmentation

When a process exits or is swapped out, it leaves its region of memory free. Unless a neighboring region of memory is already free, this region can hold a process only of equal or smaller size.

If it is used to hold a smaller process, a small unused region of memory is created. This "hole" region is often too small to hold any process.

In time, memory will have lots of such unusable regions. Eventually, the OS will not be able to find contiguous space for any process even when the total amount of free memory is sufficient.

External fragmentation



As processes are loaded and removed from memory, the free space is broken into "little pieces."

Memory compaction

This problem is called **external fragmentation** because the too-small holes (the fragments) are outside any allocated region.

One way to handle external fragmentation would be memory compaction: moving all processes into one contiguous area at one end of memory, leaving a single large unused area at the other end. (Alternatively, one could create an unused area in the middle.)

Unfortunately, the speeds of main memories have lagged way behind the speeds of CPUs; since 1980, they have improved only by a factor of about 10-20, compared to a factor of well over 2000 for CPUs.

This means that relative to other things a CPU can do, the cost of memory copying has actually *increased*. Combined with the explosion in the *size* of main memories, this makes memory compaction totally impractical.

Memory management using linked lists

A way of keeping track of memory is to maintain a linked-list of allocated and free memory *segments*, where a *segment* either contains a process or is an empty hole between two processes.

Several algorithms can be used to allocate memory for a created process (or an existing process swapped in from disk):

- **first fit** – scan list until a hole that is big enough is found
- **next fit** – similar to first fit, but continues scan from previous position
- **best fit** – search entire list, taking the smallest hole that is adequate
- **worst fit** – search entire list, taking the largest available hole

Absolute addresses . . .

If a program wants to refer to a variable by its absolute address, then it must know at what address that variable is located.

If a program can be loaded into memory at different locations on different invocations, then absolute addresses will not work without software or hardware tricks.

Base and limit registers

When loading the memory image of a process, the OS sets the base register to point to the start of the image.

The hardware adds the value in the base register to every address generated by the program before accessing memory.

Thus the address as it appears to the program and the address of the physically accessed memory cell are now two different things. However, *the program is not aware of this fact*.

Base and limit registers

When the program issues an instruction to load the contents of location 100 into register 1, the hardware will load the contents of location 500 into register 1 instead if the base register contains 400.

A similar displacement will happen to all other addresses. A program can thus think that it is using locations 0-999 when it is in fact using 400-1399.

Base and limit registers guarantee security if the hardware

- compares every program-generated address with the limit register, and generates an exception if the limit is exceeded; and
- allows the base and limit registers to be modified only in kernel mode.

If the limit register is set to 999, then the current process can only use addresses 0-999. This is its **address space**.

Virtual memory

The causes of the external fragmentation problem are based on two assumptions:

- all the code and data of a program has to be in main memory when the program is running, and
- this code and data has to be stored in contiguous locations.

In the interface they present to the programs running on top of them, virtual memory systems allow programs to continue making both assumptions.

The reason for the word "virtual" in their name is that internally, in order to cure external fragmentation, VM systems break both assumptions, without this fact being visible to their clients.

Virtual memory

- With virtual memory, only some of the code and data of a program have to be in main memory: the parts needed by the program now. The other parts are loaded into memory when the program needs them *without the program having to be aware of this*.
- Similarly, different parts of the program can be loaded into different parts of memory, again without the program having to be aware of this.

The first point means that the size of a program (including its data) can exceed the amount of available main memory.

This was very important in the 1960s/70s when main memories were very small (e.g. 64 kilobytes), and is still useful now.

The concept of a virtual address

There are two main approaches to virtual memory: **paging** and **segmentation**.

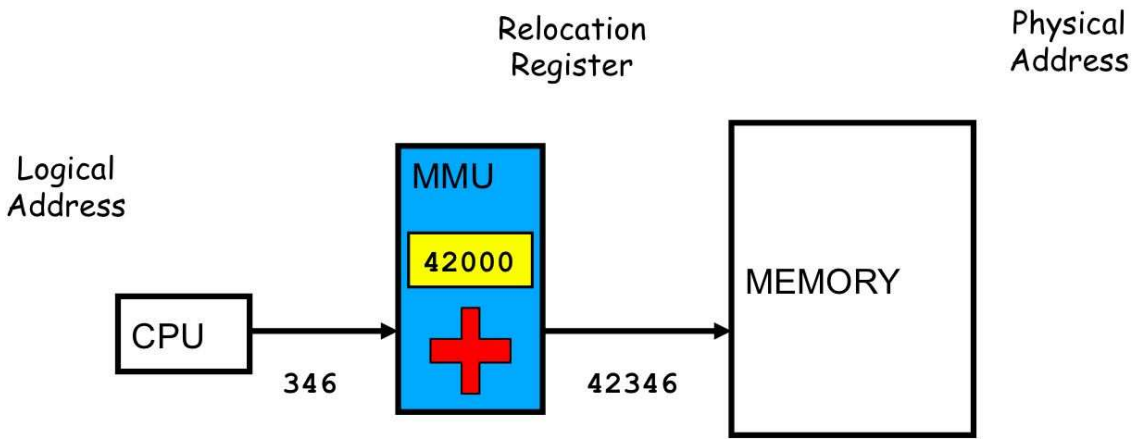
Paging relies on the separation of the concepts virtual address and physical address.

Addresses generated by programs are virtual addresses. The actual memory cells have physical addresses.

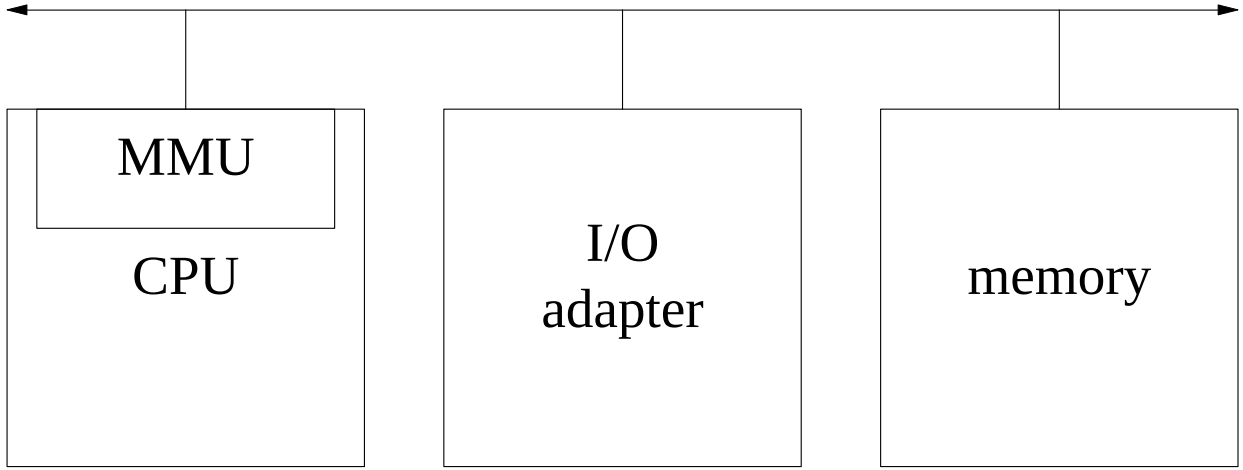
A piece of hardware called a *memory management unit* **(MMU)**, translates virtual addresses to physical addresses at run-time.

Simple example: mapping to a physical address using MMU

The mapping is generated by MMU. All actual memory access done with physical address value.



Location of the MMU



When the CPU accesses memory, it gives a virtual address to the MMU.

The MMU then sends the corresponding physical address to the memory as the target of the read or write operation.

CPUs typically provide a mechanism for code executing in kernel mode to access main memory using physical addresses directly, bypassing the MMU.

Address spaces

The physical address space of a machine contains one address for each memory cell. In almost all modern machines, one cell contains one byte.

The virtual address space of a machine is the set of addresses that programs on that machine may generate. Each process has its own virtual address space.

The virtual address space may be bigger than its physical address space. It may also be the same size, or even smaller, although such systems are rare.

The number of bits in a virtual address is virtually always a power of two. PCs are just finishing a transition from 32 bit to 64 bit virtual addresses; mainframes and servers finished that transition a decade ago.

Address spaces – historical perspective

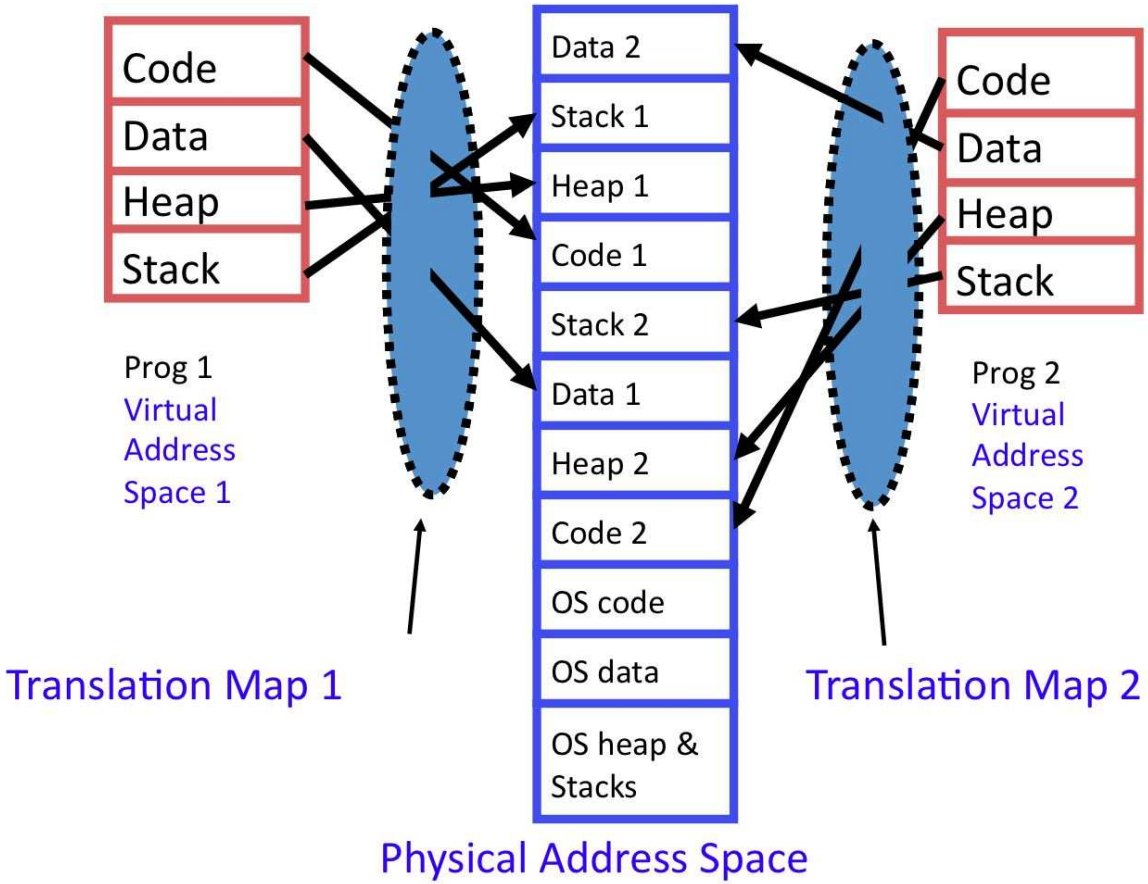
In the 1960s/70s, word-addressed machines used to be common. In these machines, one memory cell contained one word, which was variously 12, 16, 20, 24, 32 or 36 bits in size. They are now extinct.

Some machines in the 1960s and 1970s had e.g. 24 and 36 bits in their virtual addresses, but these are now also extinct.

Minicomputers in the 1970s and the early PCs in the 1980s had 16-bit virtual addresses. The descendants of the ones that aren't extinct now have 32 or 64 bit virtual addresses.

Some CPUs with 16 bit virtual and/or physical addresses are still used, embedded in consumer products such as microwave ovens and cars.

Address spaces: a simple illustration



Paging

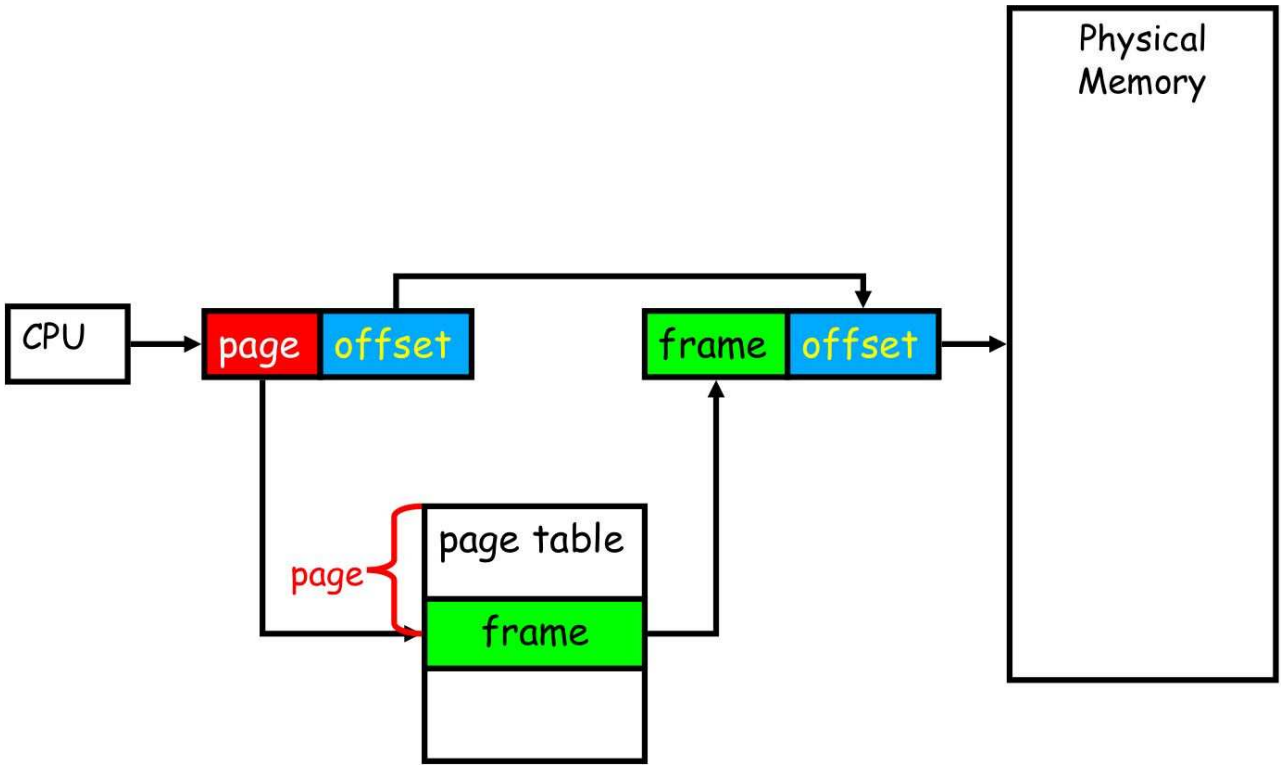
A paged system divides both virtual and physical address spaces into fixed size **pages**. The MMU can map any virtual page onto any physical page. As the job of a physical page is to hold a virtual page, physical pages are also called **page frames**.

During the lifetime of a process, pages in its virtual address space all start out on disk, and may move between disk and memory any number of times. When a virtual page is moved to disk and back again, it may be put in a different page frame than before.

The amount of memory a process needs is rarely an exact multiple of the page size, but each process is always allocated an integral number of pages. This means that paging wastes a fraction of a page for each process. This is called **internal fragmentation**, since the wasted space is inside a region allocated to a process.

Example: Address translation architecture

High level overview of paging. Details are discussed on the following slides.

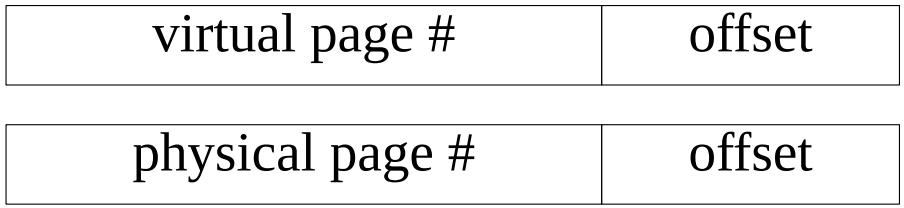


Virtual address components

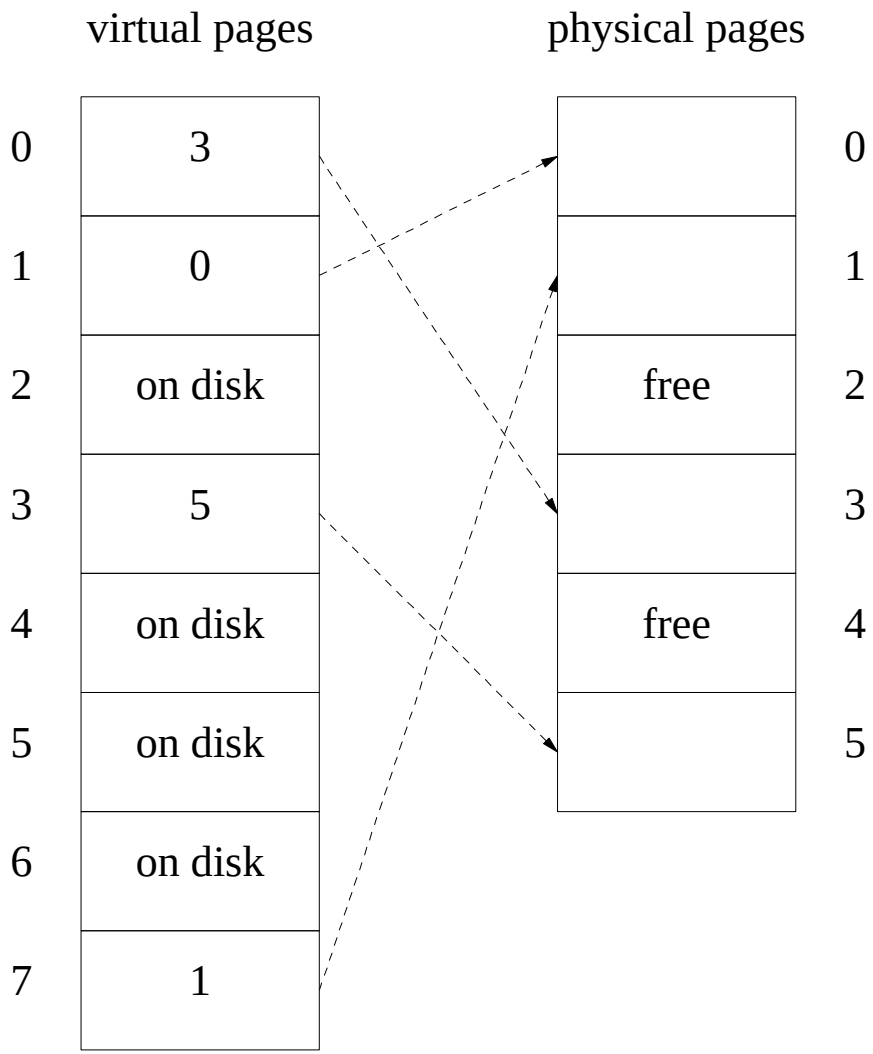
In a paged system, an address is always viewed as $page\_number * page\_size + offset$. The page size in bytes is 2 raised to the power of the number of bits in the offset:

offset bits:	9	10	11	12	13
page size:	512	1024	2048	4096	8192

This means that a virtual address has two parts: a virtual page number, and an offset within the page, and that a physical address has two parts: a physical page number, and an offset within the page.



A small page map



Operation of paging

Whenever the CPU accesses memory, the MMU transforms the addresses according to the mapping.

For example, assuming the previous page map and a four kilobyte page size, when the program reads the contents of location 12328 = 3 * 4096 + 40 (virtual), the MMU instructs the memory to deliver the contents of location 20520 = 5 * 4096 + 40 (physical), because virtual address 12328 is in virtual page 3, which the map maps to physical page 5. Of course it performs the same mapping for writes.

The information required to do the mapping is recorded in a *page table*. Each process has its own virtual address space and thus its own page table.

Page tables

A page table has one entry for each virtual page number used by the process. In a typical computer, a *page table entry (PTE)* contains:

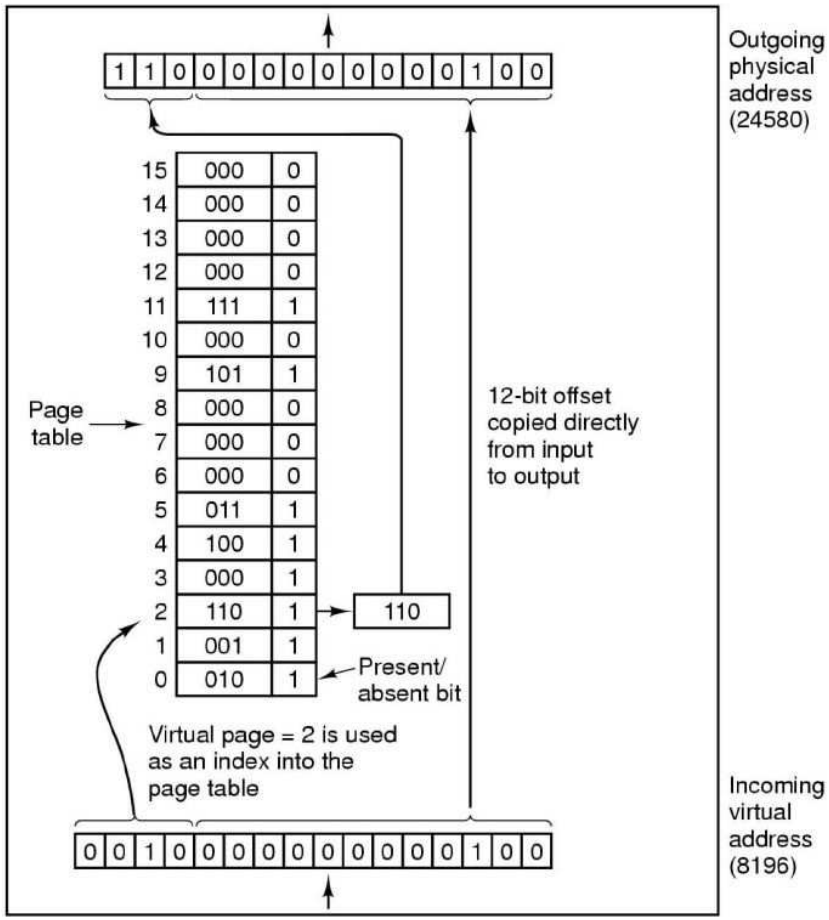
- a physical page number
- a valid bit
- a referenced bit
- a modified bit
- read, write, and execute permission bits

The valid bit says whether the virtual page is in main memory or not.

In many systems, the physical page number field is used to hold the location of the page on disk when the page is not in memory.

The arrows pointing off the left side represent the valid bit and the permission bits that need to be checked.

Page table operation: exercise



MMU checks

The MMU must check that the selected PTE has the valid bit set to one (true) and that the permissions permit the requested memory access.

If both conditions are met, it will construct the physical address using the physical page number field of the PTE.

It will then set the referenced bit, and if the access was a write, it will also set the modified bit.

If either condition isn't met, the MMU will cause an exception called a page fault. This must be handled by the kernel.

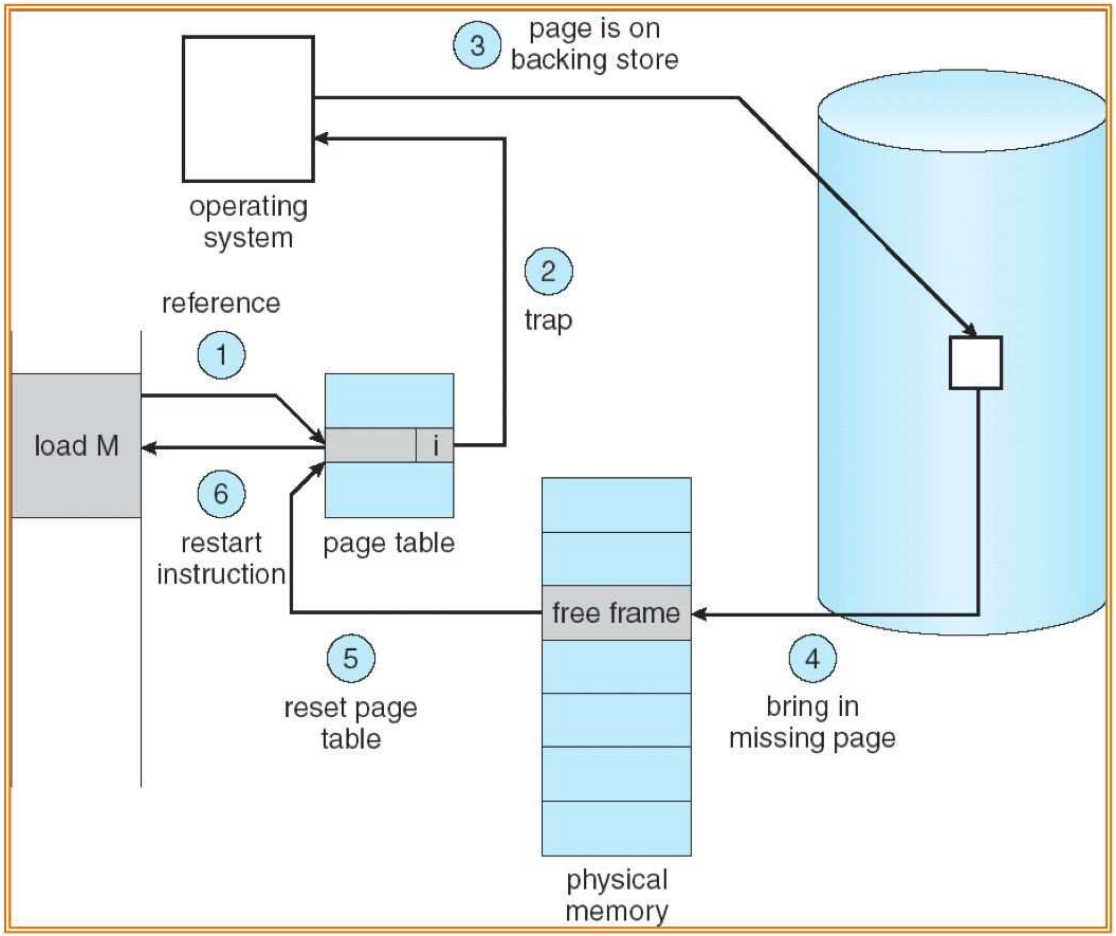
Page fault handling

If the page fault is caused by the permissions being violated, the page fault handler will usually terminate the process.

Otherwise, i.e. when the valid bit is zero (representing false), the OS must:

- suspend the process,
- free up a page frame (if there are none available now),
- load the required virtual page from swap space into a free page frame,
- cause the MMU to map the virtual page onto the physical page, and
- restart the process at the *same* instruction.

Example: Page fault handling



TLB

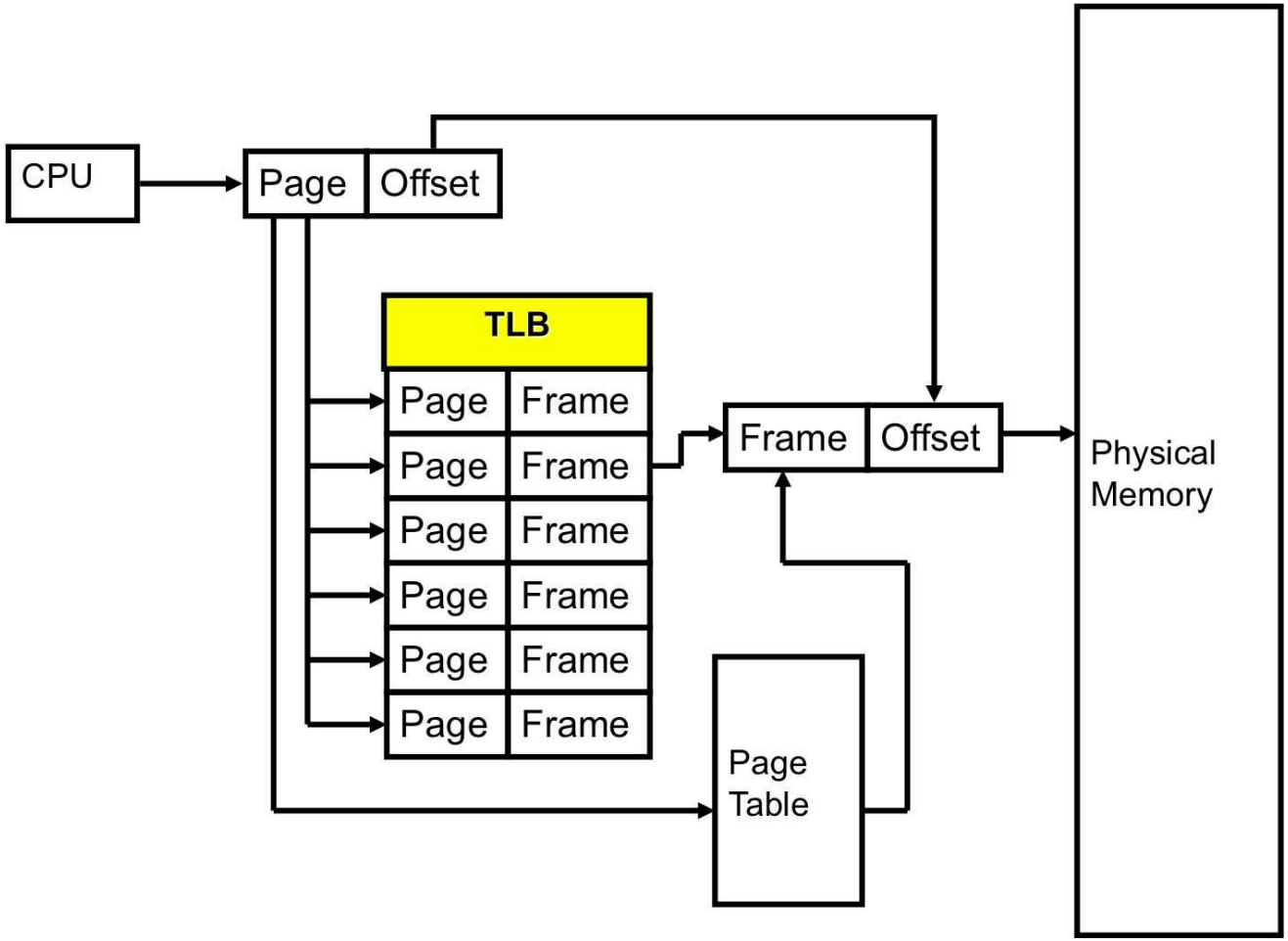
Without optimization, each memory reference by the program in a paging system would require two memory accesses: one to access the PTE and one to access the data.

To avoid this performance penalty, paged computers have a translation lookaside buffer **(TLB)** in the MMU.

The TLB serves as a *cache*, holding copies of recently accessed page table entries (PTEs). Every newly accessed PTE is loaded into the TLB, discarding an old TLB entry to make room if necessary.

The TLB is implemented using associative circuitry that is much faster than main memory, and its access can usually be overlapped with access to the computer's data and instruction caches.

TLB


CPU
Page
Offset
TLB
Page
Frame
Page
Frame
Page
Frame
Page
Frame
Page
Frame
Page
Frame
Frame
Offset
Physical Memory
Page Table

TLBs and context switches

Each process has its own page table. When the OS switches from process 1 to process 2, it must make sure that process 2 does not use TLB entries belonging to process 1. The two processes will map the same virtual page number to different physical page numbers.

The simplest way to assure this is to clear the TLB at the context switch.

Page table structure

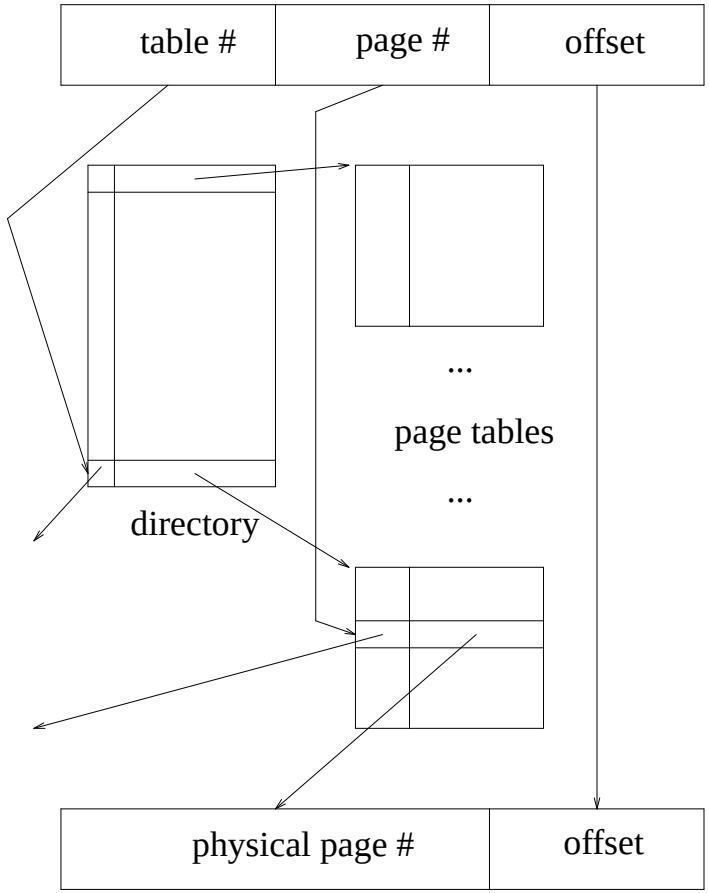
The OS should allocate PTEs only to the regions of virtual memory the process uses.

A scheme that used to be popular was splitting the page table in two: one for code and the heap at the start of the address space, and one for the stack at the end. However, this can't handle dynamic linking and shared memory operations, which both want to use pages somewhere in the middle of the address space.

A better scheme uses a page table *directory* that is always in memory when the process runs; its entries point to page table *fragments*.

The fragments may be in memory or on disk, or they may not exist at all if that region of the address space is unused.

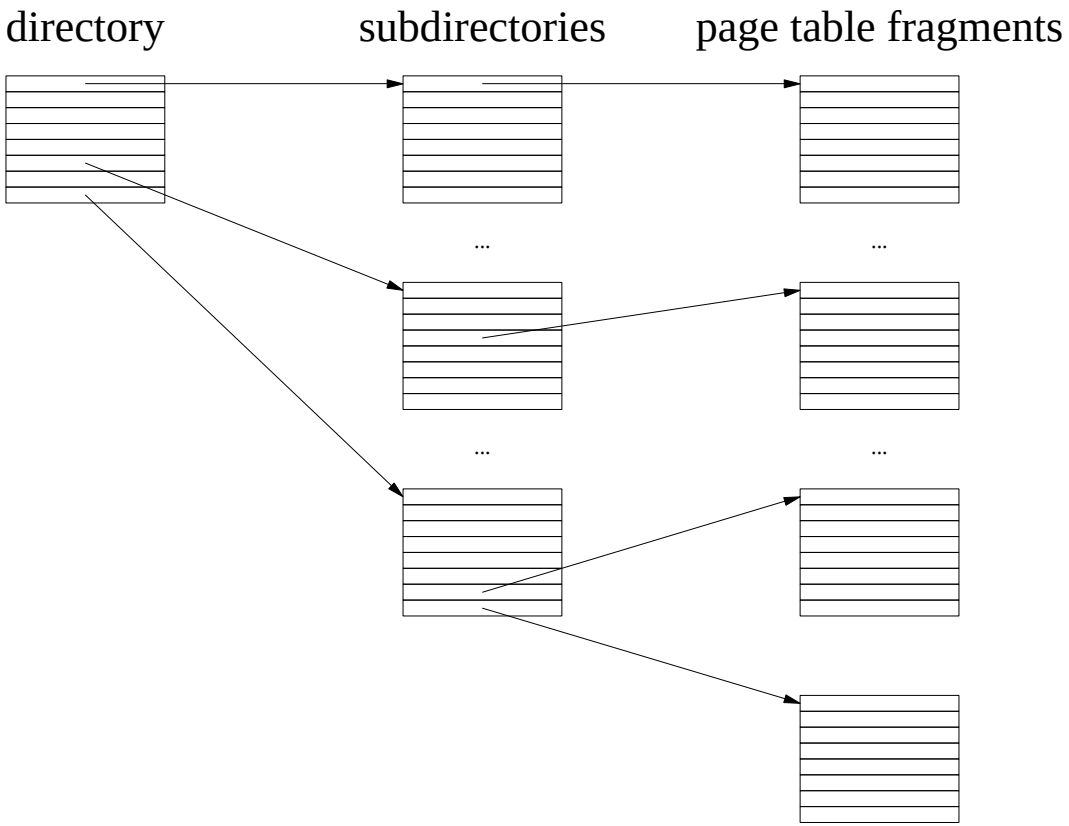
Two level page table



As before, the arrows pointing off the left side represent bits that need to be checked: a valid bit for the directory and valid bit and permission bits for the page tables.

Three level page table

Linux on 64 bit machines uses a three level version of this scheme. This diagram shows the general arrangement, without the details (e.g. PTE fields).



Three level page table

The previous diagram shows what the page table might look like for a process that has a stack in the top part of the address space, the code, readonly data, writeable data and bss space (for malloc) at the bottom of the address space, and another memory region (maybe holding a dynamically linked library) in the middle of the address space.

Current machines with 64 bit virtual addresses usually impose the restriction that e.g. the top 20 bits have to be either all zeros or all ones. Without this restriction, covering the entire address space would require a multi-level page table to have more levels (e.g. five, for reasonably sized table fragments).

Where is the table?

With the split table organization one must know the physical address of the beginning of each half of the page table, and the length (number of PTEs in) each half.

With a directory organization, one must know the physical address of the directory and the length of the directory and each fragment.

These values reside in MMU registers writeable only in kernel mode.

A CPU register, usually called something like “page table base register”, contains the address of the first word of the page table of the currently running process.

User vs System space

In most computers, the kernel is actually a part of the address space of every process. A virtual address with the high bit set refers to *system space*, one with the high bit clear refers to *user space*.

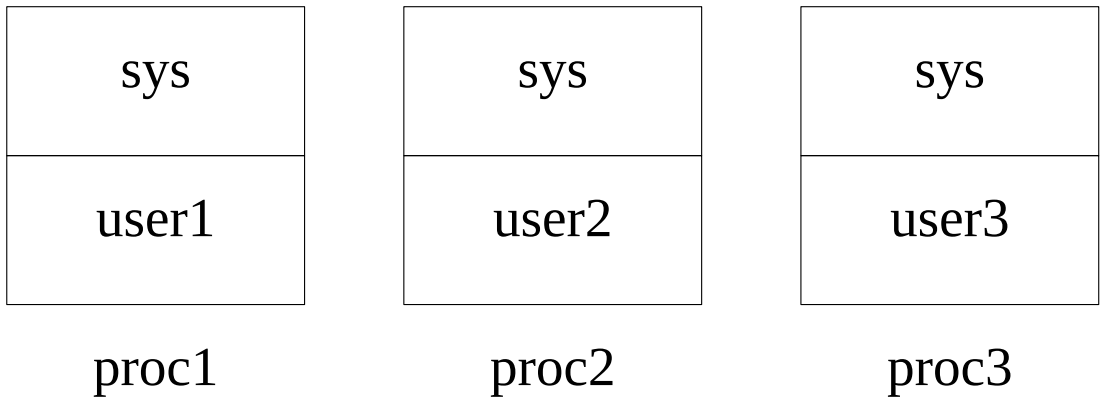
Processes should be allowed to access system space only when executing in kernel mode, i.e. during a system call.

The hardware can ensure this by inspecting the high bit, or by using two sets of permission bits in PTEs: one for user mode and one for kernel mode.

User space

Some system calls take pointers as arguments. The kernel code for the system call can simply access the memory of the current process after verifying that the pointer has the high bit clear.

There is a single page table for system space. There are several page tables for user space, one for each process.



Recursive pitfalls

The page fault handler *must* be kept in memory. If it isn't, the attempt to access it will itself result in a page fault.

When the TLB misses, the lookup of the PTE must *not* go through the MMU in the usual way. If it does, the PTE lookup may itself miss in the TLB.

In a single level scheme, after a TLB miss in a user program, the hardware knows the physical address of the relevant PTE. (In a two- or three-level scheme, the hardware must look it up in the page table directory structure.)

Page replacement

Unless the system has unused page frames, the OS must choose a page to remove from main memory when a page fault occurs.

If the chosen page has been modified since being brought in from disk (you can tell by looking at the modified bit in its PTE), the OS must first write the page out to the disk.

If it hasn't been changed, the copy on disk is already up-to-date and no disk write is needed.

A good replacement algorithm is critical for good system performance.

The principle of locality

All realistic page replacement algorithms rely on the principle of predicting that the near future will be about the same as the near past.

They have to, because programmers don't want to be burdened with providing this kind of information; most programmers don't even *know* this kind of information.

Spatial locality: if a location has been referenced lately, locations near it are likely to be referenced soon.

Temporal locality: if a location has been referenced lately, it is likely to be referenced again soon.

Though the descriptions sound similar, the two kinds of locality are in fact orthogonal.

Page replacement policies

Optimal policy: from all the pages in main memory, pick the page that whose next reference is as far in the future as possible. This requires foreknowledge of the actions of all programs, which is just about never available. Establishes the high bound on how good an algorithm can be.

Random policy: pick a page at random. Establishes the high bound on how bad a realistic algorithm can be. In some cases (e.g. during garbage collection) one cannot do better than this bound.

FIFO

The first-in first-out (FIFO) algorithm selects the page that has been in main memory longest.

Although it is simple to implement, it has poor performance because it disregards the principle of locality.

It pays no attention to page usage: it throws out a page that has been heavily used recently as easily as a page that has not been referred to in a long time.

Least recently used (LRU)

LRU picks the page that has been unused for the longest time, reckoning that it will *stay* unused for the longest time.

This algorithm requires hardware to record times of access to each page; this "time" may be reckoned in page faults or clock ticks. When a new page frame is needed, the OS scans all these timestamps and selects the page frame that was accessed least recently.

Recording timestamps effectively doubles the cost of memory accesses, so it is not done in practice.

Least frequently used (LFU)

This algorithm also associates a counter with each page frame, and also adjusts these counters on every clock interrupt. However, the LFU algorithm simply increments the counter of the pages that have been accessed since the last clock interrupt.

The page with the smallest counter is the one used least frequently.

If a page was once heavily referenced, LFU tends to leave it in memory even after it has outlived its usefulness.

The clock algorithm (not Tanenbaum's)

This variation of "not recently used" slowly sweeps through memory in a circular manner, clearing the **Referenced** bit of each page.

If this bit is still clear when the "clock hand" reaches it again, the page has not been used in the intervening interval; therefore it can be made a candidate for replacement.

With the typical main memory sizes of current systems, even junk pages have a good chance of being referenced between visits, so the clock should have two hands, the first resetting Referenced bits and the second testing them.

The second hand will be behind the first hand by a fixed number of pages or by a fixed number of milliseconds.

Free page pool

The OS can minimize the time it takes to handle a page fault by endeavoring to keep at least one page (but preferably many more) in a list of clean, free-to-be-replaced page frames at all times.

The page fault handler can then start paging in the wanted page immediately; *then* it can find another page to free.

A page in the free page pool can be given back to its process if the process accesses that page.

This significantly reduces the cost of any mistake made by the main page replacement algorithm. The free page pool can thus improve the performance of pretty much any such algorithm.

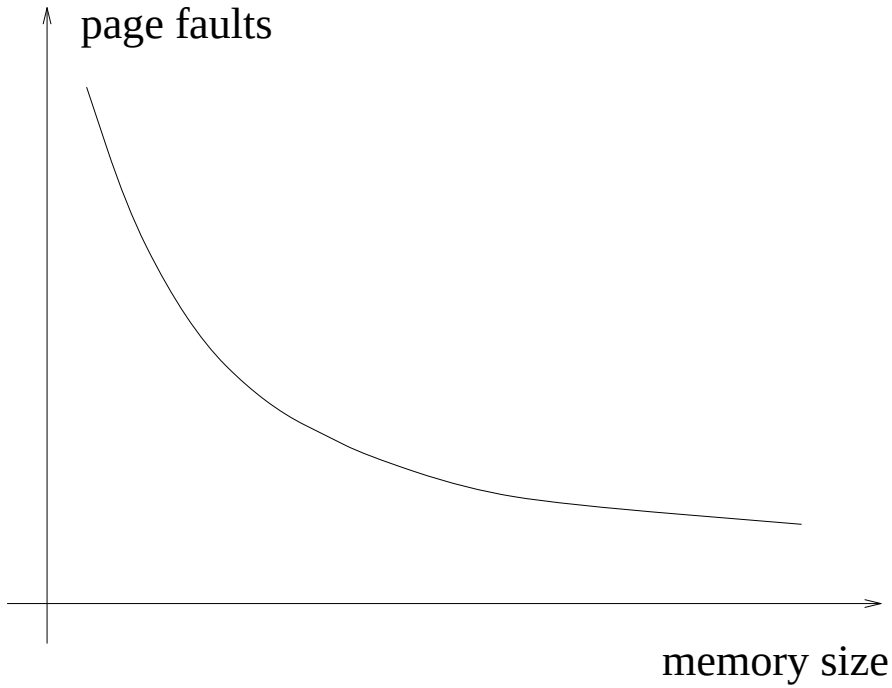
Page write daemon

Paging in a page to a dirty page frame costs two disk accesses: one to write the old page to disk and one to read in the new page.

Paging in a page to a clean page frame costs only one disk access. The free page pool makes this happen much more often.

Whenever a paging disk is unused, the daemon can find some dirty old pages, write them to disk, and put them in the pool.

Expected performance



With a small number of page frames, processes will continually fault for pages that have been just paged out.

Multiprogramming and memory

If the number of processes in memory is too high, each process will only have a few page frames, resulting in high page fault rates. The queue of processes waiting for disk will become longer, and the CPU will be underutilized. This is called *thrashing*.

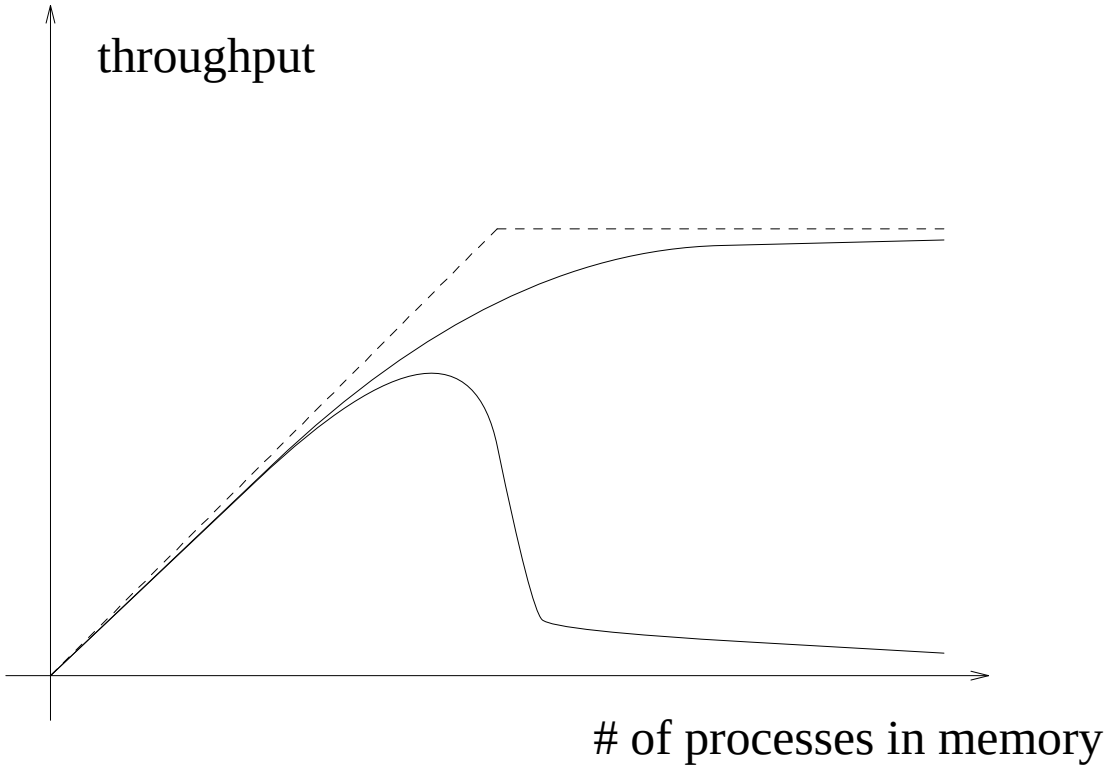
If the OS responds to the underutilization of the CPU by bringing more processes into main memory, the situation becomes worse; this is a vicious positive feedback loop.

If the machine will be under significant load, the OS should

- recognize imminent thrashing, and avoid (or delay) actions that would create it, and
- recognize that thrashing has started, and take steps to end it.

Mainframe OSs are *much* better at both than PC OSs.

Thrashing



To the left of the turning point, the system is CPU bound (normal operation); to the right, it is I/O bound (thrashing).

Working sets

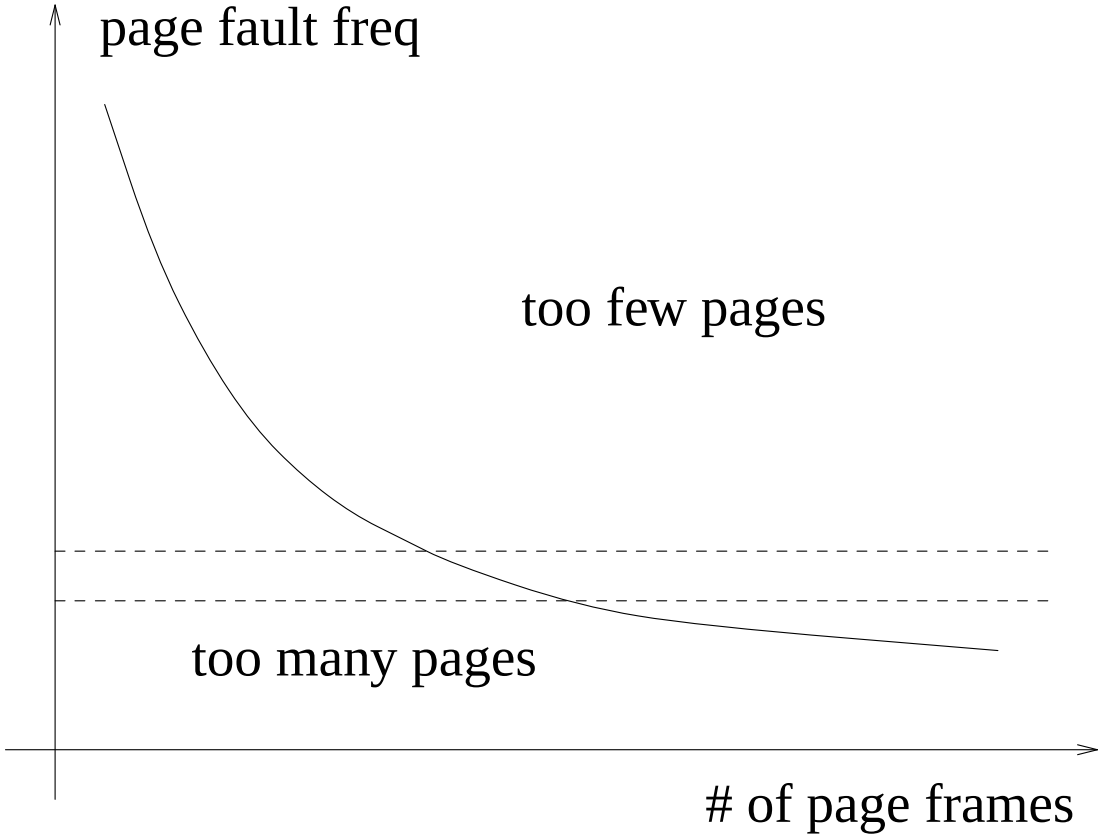
The working set principle is that a process should not be run unless the set of pages containing the memory regions it needs regularly (its "working set of pages") is in memory.

A process without its working set in memory should either have more pages allocated to it or should be moved to disk altogether.

One can model the working set of a process as the set of pages the process has referenced in its last N instructions. (N is a parameter of the model.)

This definition is useful mostly for theoretical and simulation purposes. In practice, the working set must be approximated. One way is by substituting "clock tick" for "instruction" in the definition.

Approximation via page fault frequency



The OS can adjust the "normal" band to aim for utilizing a given percentage of the bandwidth of the paging disk(s).

Locality shift

Some programs can be naturally divided into phases that access different code and (usually) different data structures. E.g. the passes of a compiler usually have little code in common, and they have their own data structures beside the shared data representing the program.

When such a program finishes one of its phases and starts another, it will momentarily experience a sharp increase in the size of the working set.

The "normal" page fault rate band should be wide enough to accommodate this behaviour.

Locality shift illustration

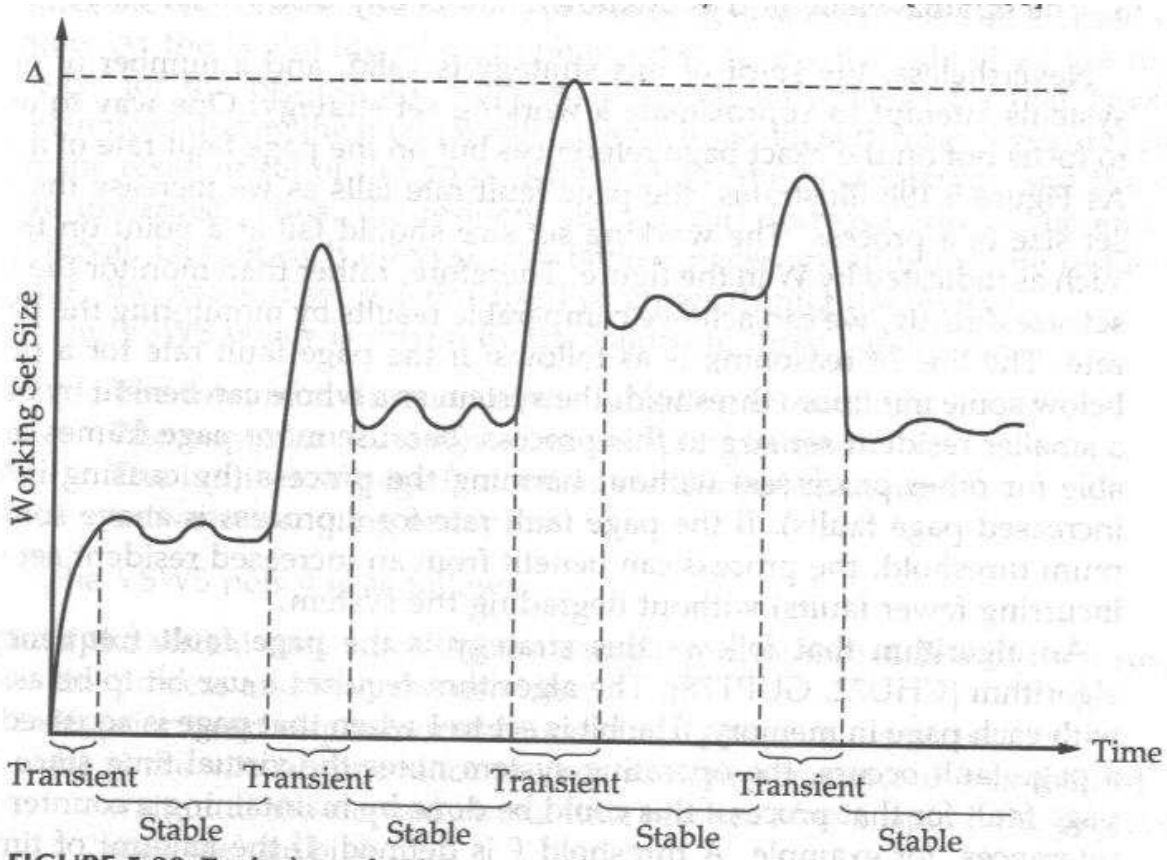


FIGURE 5.28 Typical graph of true working set size [MAEK87]

Working set policy

When the processes in memory all have their working sets, and there are page frames available, the OS may bring in a new process from disk.

When a process in memory does not have its working set and there are no page frames available, a process must be moved out to disk (i.e. swapped out).

The policy discriminates against some processes to allow other processes to make progress. Without it, multiprogrammed systems thrash under load.

Process creation ... revisited

There are two main ways of creating a process.

- Issue a system call specifying what program the new process is to run, and all the attributes of the new process.
- Issue a system call that creates a copy of the current process. The copy can then execute system calls to set up its attributes or run another program.

Most OSs use the first approach. One disadvantage is that as the number of attributes of a process grows (because of new OS features, e.g. the ability to support realtime), you need new system calls.

Unix uses the second approach: it is more flexible, but can be more expensive. However, the extra cost can be managed, and process creation is faster on Unix than on most other OSs.

Unix system calls

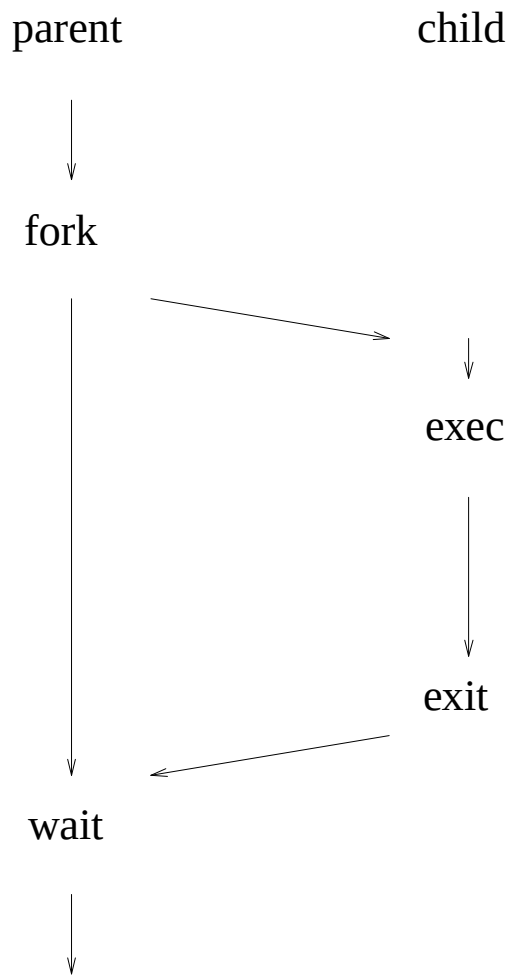
fork Makes a copy of the current process; returns the PID of the child process to the parent, zero to the child.

exec Throws away the current address space, replaces it with the specified program and branches to its start.

exit Destroys the current process, specifying an exit code.

wait Waits until a child process exits; returns the PID of that child and its exit code.

Process history



An example

```
if ((pid = fork()) == 0) {
    /* code for the child: set up the environment */
    exec("/bin/sh", "sh", "-c", cmd, NULL);
    perror("cannot execute command");
    _exit(1);
}
/* code for the parent */
exitpid = wait(&status);
...
if (status.w_termsig != 0) {
    printf("killed by signal\n");
} else if (status.w_retcode != 0) {
    printf("abnormal exit\n");
}
```

Advantages of fork

In some cases, it creates exactly what is needed: a copy of the current process. (Fork does *not* have to be followed by an exec.)

In the usual case, the advantage is that the child process can adjust its environment before the call to exec.

This adjustment can use *any* information available to the parent at the time of the fork, and does *not* require any changes in the code of the exec'd program.

Disadvantage of fork

Most of the time, fork is followed almost immediately by an exec. The work done to create a copy of the old process is therefore almost totally wasted.

One solution of this problem is vfork, a new system call. After vfork the child "borrows" the address space of the parent until it executes exec or exit.

This has the weird effect that the child process can affect the address space of its parent; it can modify its contents or even its size.

A better solution is to make the copying operation cheaper by only copying the parts that need to be copied.

Copy-on-write

Instead of copying all pages of the parent process at the time of the fork, the OS copies only the page tables, and marks both sets of pages read-only and copy-on-write (COW).

Whenever either process gets a protection violation when trying to write to a read-only page that they should be able to write to, the OS checks the COW bit, and if it finds it to be set,

- copies the page,
- sets the write permission bits and resets the COW bits in both PTEs, and
- restarts the faulting instruction.