Memory Systems - Optimization and Performance

Lecture on

Memory Systems

Organization, Performance and Design Issues

Walter Kriha

Goals

- Understand the differences between kernel and user space with respect to performance and isolation.
- Learn how memory is administered by the OS kernel. Notice differences and similarities to file systems.
- Memory hierarchies and performance. Multiprocessing problems
- Understand the consequences of separate memory spaces per process: Interprocess communication.
- Understand the price of frequent memory allocations.

Memory management is another case of general resource management by the operating system.

Procedure

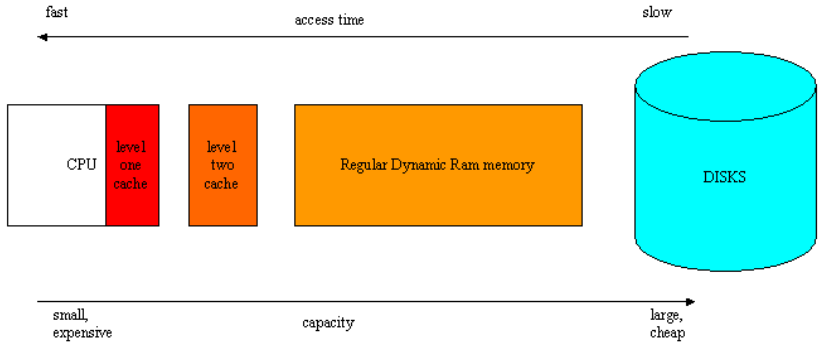
- We will learn how memory is separated and organized
- How the kernel maps virtual address spaces to physical memory
- Take a look at shared memory

Memory Resources vs. File Resources

- volatile vs persistent
- fast access vs slow access
- different owners
- large resource size
- concurrent access
- from bits to blocks

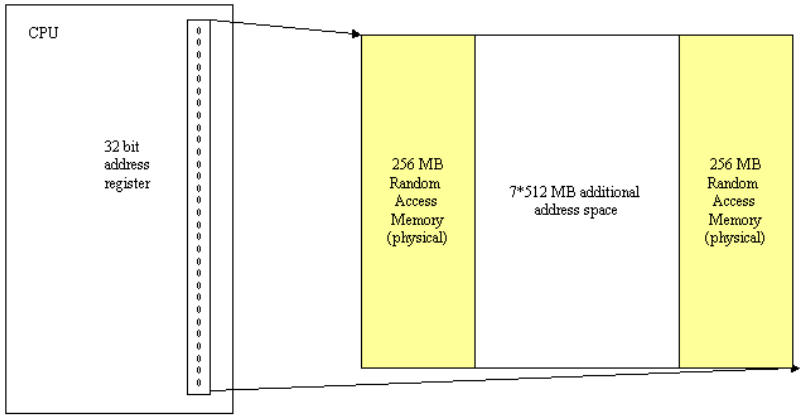
Looks like files and memory are not really so much different. In fact, the memory API can be used to handle files as well. The other way round does not make much sense.

Memory Hierarchy



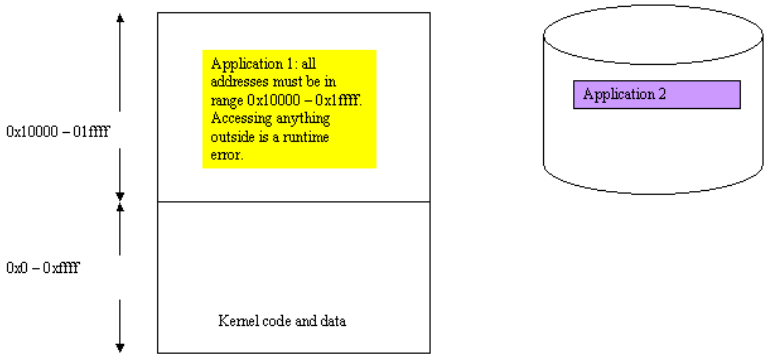
Access time and capacity are exactly reversed. To keep prices reasonable a compromise between size and speed must be met. The trick is to use caching as much as possible to keep freqeuently used data in the small but fast areas of the memory hierarchy. Virtual memory management allows us to treat non-Ram space like discs as memory extensions invisible to programs. This is called paging or swapping and will be explained shortly.

Address Size vs. Memory Size



A 32-bit CPU can address 4GB of RAM. Most machines do not have so much physical memory. Notice that physical memory can be at several location in the overall addressable space. It is even possible to have systems with more memory than addressing capabilities. In those cases „bank switching“ is used to allow the CPU to address different memory regions over time

Physical Memory Systems



This type of system was used in early timesharing machines. The application is completely within the physical memory and limited to its size. During code generation the linker needs to know the starting address and size of the application memory area. Realtime and embedded control systems frequently use the same design. Its major advantage is its simplicity and predictability with respect to response times. Disadvantages are the restriction of one application with a fixed size only.

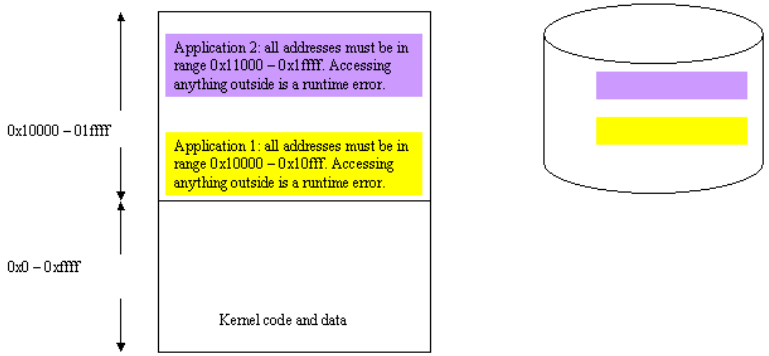
Why Multiprogramming?

- Optimize CPU usage
- Optimize User experience

Most programs perform some type of Input/Output (I/O) operations which perform factors slowers than raw CPU speed. If a machine can run only one program CPU usage will be far below optimum (100%) which is a waste of resources. By having several programs concurrently in memory the hope is that there will always be at least one that is runnable. The formula to calculate CPU utilization is: CPU utilization = $1-p^n$ where p is the wait time fraction and n the number of processes. CPU usage is only a measure for overall throughput and not how responsive a system will feel for a user. (Tanenbaum pg. 193ff.)

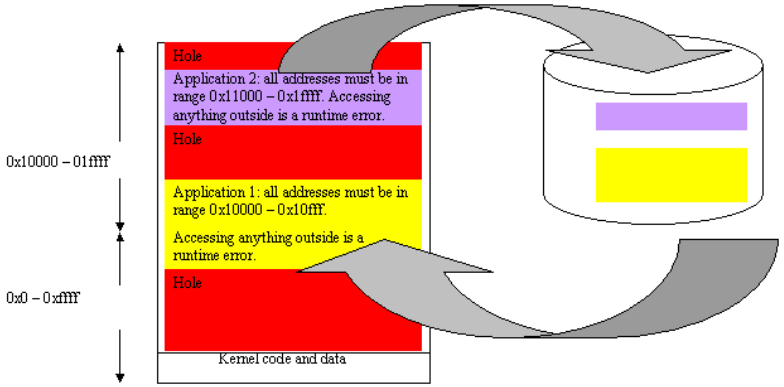
Multiprogramming allows interactive tasks which service users directly to run almost concurrently (if not really given multiprocessors) to batch jobs. Interactive programs usually have long wait-times (user think time) and therefore mix well with background processing.

Multiprogramming Physical Memory Systems



To the size limit due to physical addresses come two other problems: relocation and protection. The purple colored application needs to be created with different addresses than the yellow one. This can happen dynamically during program load or statically at creation time through the linker. And every additional program increases the risk that due to a programming error memory locations from other programs are touched. This design is nowadays only used in small embedded control systems. The advantage lies in cheaper hardware due to the missing memory management unit.

Swapping in Memory Systems

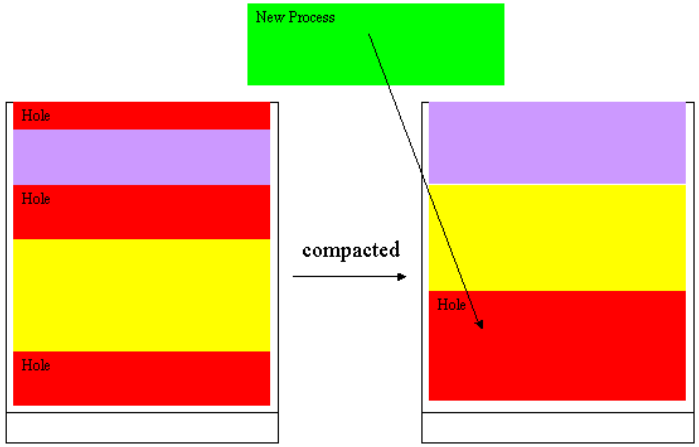


Several processes can run concurrently as long as they fit into memory. The system needs to find proper places for processes and deal with memory fragmentation due to different process sizes. If a process needs to grow and there is no free memory, some process will be „swapped“ to disk to make room. Different strategies for process selection exist. Swapping can be implemented with or without separate process spaces.

Swapping and realtime applications

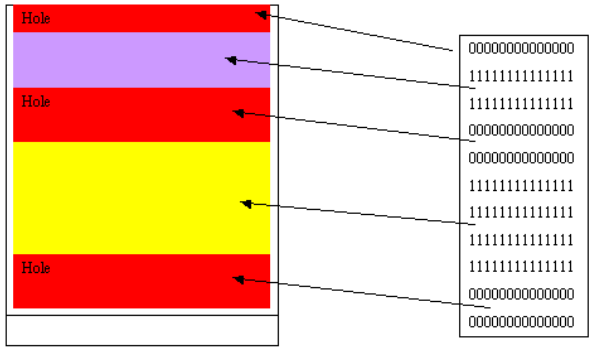
A realtime application needs to be able to respond within a certain time period. Systems that use swapping cannot guarantee this because the application might be swapped at that time. Most systems allow to „pin“ an application into memory – basically making an exemption from the overall memory management policy. Realtime systems always try to keep all applications complete in memory.

External Fragmentation



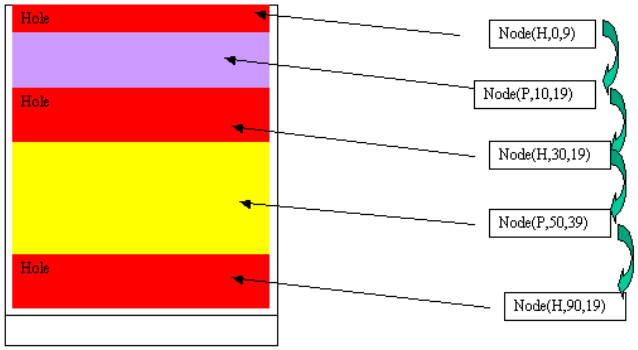
Allocation strategies with different allocation sizes suffer from external fragmentation. One way to cope with it is to perform compaction. This is quite expensive as it requires a lot of memory copying. We will see the same strategy used by some garbage collectors. It is important to always re-combine adjacent free memory areas into a single free area.

Memory Management with bitmaps



A bitmap is a fast and compact way of storing metadata about a large number of resources. If a new process needs to be created the memory manager must walk through the bitmap to find a piece of 0's large enough to fit the process in. This can take a while.

Memory Management with linked lists



The linked list contains both program (P) and holes (H). Different lists could hold different sizes of holes so that allocation is faster.

Memory Management Strategies

- First fit: take the first hole that is big enough
- Next fit: same as first fit but starts from last location
- worst fit: always split a big hole into smaller chunks
- quick fit: separate lists for different hole sizes.

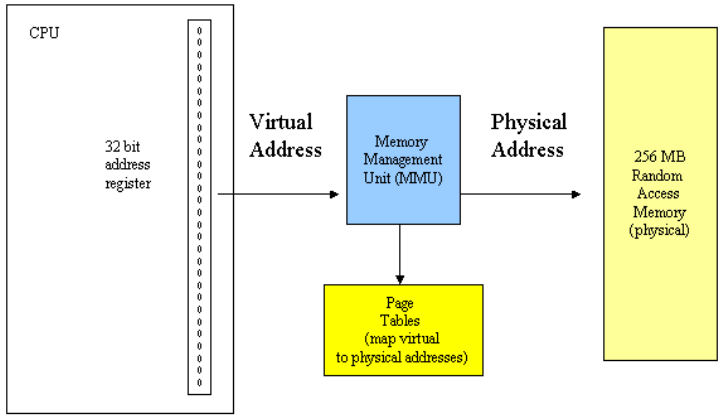
Find more about these strategies at Tanenbaum pg. 200ff. Notice that not only allocation time is important here. When memory becomes free because a process is removed the now free pages need to be re-combined with neighbouring free pages which can be very hard e.g. with quick fits different lists.

Reasons for physical/virtual organization

- avoid binding program locations to fixed physical addresses (programs could not be moved in memory)
- increase program size beyond physical memory size
- create separate address areas for programs. Programs cannot directly access the memory area of other programs.

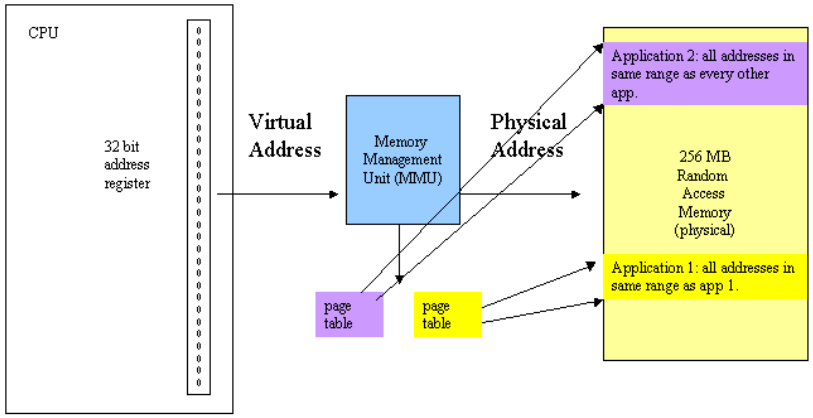
In times of huge physical memory sizes the last point is probably the most important one: Avoid unintended side-effects. Program errors cannot affect other programs. This creates a new problem: how then can programs interact and share data?

Virtualizing Memory (1)



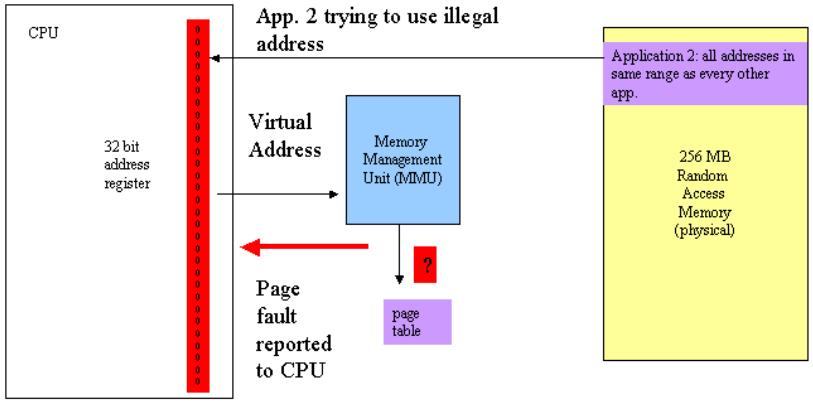
All addresses in programs are now considered „virtual“. When the CPU accesses memory the virtual memory address is on the fly translated into a physical address. The MMU is a piece of hardware, nowadays mostly within the CPU but page tables are maintained by the operating system. The relocation problem is solved because all programs contain the same virtual addresses which are mapped to different physical addresses.

Virtualizing Memory (2)



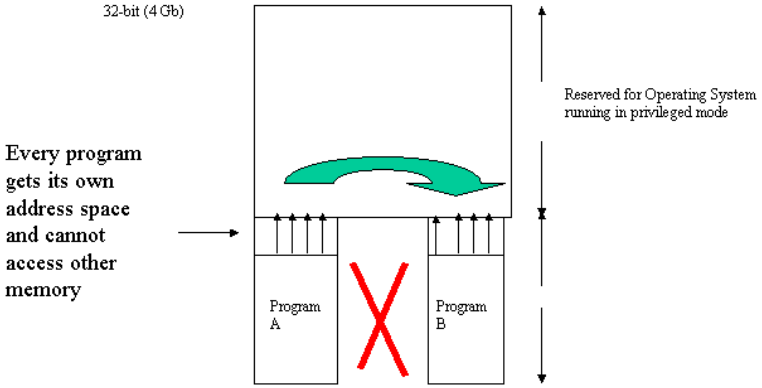
All addresses in programs are now considered „virtual“. When the CPU accesses memory the virtual memory address is on the fly translated into a physical address. The MMU is a piece of hardware, nowadays mostly within the CPU but page tables are maintained by the operating system. The relocation problem is solved because all programs contain the same virtual addresses which are mapped to different physical addresses.

Virtualizing Memory (3)



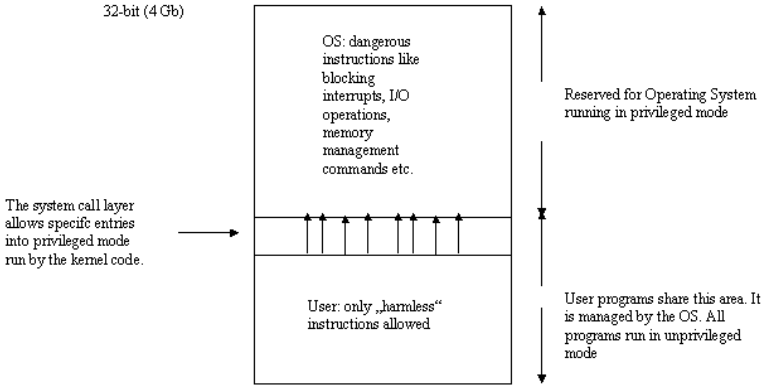
If a program wants to step out of its address range with some calculated address, the MMU tries to find a mapping for this address in the page tables for this process. A so called „page fault“ is generated by the MMU and control goes back to the CPU and then to the operating system. If the address was potentially correct (e.g. a piece not yet loaded from the program) a new page mapping is allocated, together with a new physical frame and the instruction is repeated. Otherwise the process is killed or get an error signal.

The program-program barrier



The price of separate program memory areas is that user programs which want send or receive data from other programs must do so via kernel interfaces (system calls). This is not cheap as data need to be copied back and forth from user space to kernel and from kernel to user space. (We will consider context switching times in our lecture on processes and concurrency)

The kernel-user barrier



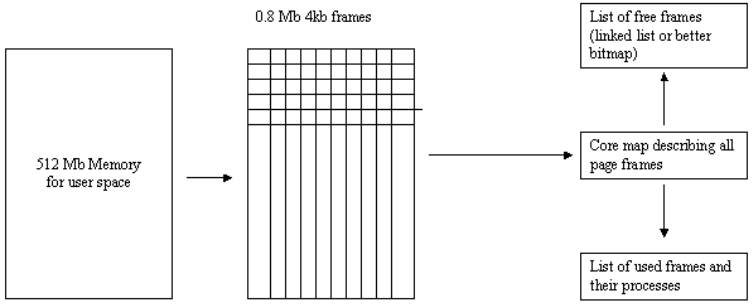
The price of separate kernel/user memory areas is that user programs which want to use kernel functions need to use a system call interface to ask the kernel to process the request. The kernel can do this either in the user program context (Unix) or as a separate process. Another price to pay is that data from I/O devices typically arrive in kernel memory space and need to be COPIED over into user memory. This is costly, especially for large multi-media data. Clever memory mapping interfaces can avoid those costs.

MMU Design

- virtual to physical mapping performed for every address
- needs hardware support (Translation lookaside buffer)
- mostly integrated with CPU
- needs integration with operating system (page tables)

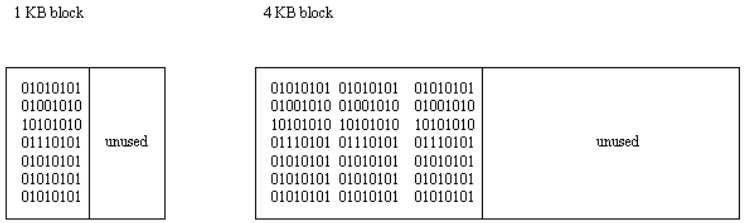
We will take a look at TLB design and performance in the session on computer organization

Basics of Resource Management: blocks



How do you administrate a large unstructured resource like computer memory? Just like a magnetic disc the first step is to create chunks or blocks. Physical blocks are called frames and have all the same size, e.g. 4kb. Now you need a free-list where you hold free frames and another meta-data structure to hold busy frames and who owns them. These lists can become quite large and in those cases one uses cascaded tables with frames containing frame addresses or sparse frame sets etc. All these data structures are maintained by the operating system. Otherwise processes could manipulate other processes memory or get undue numbers of frames.

Internal Fragmentation



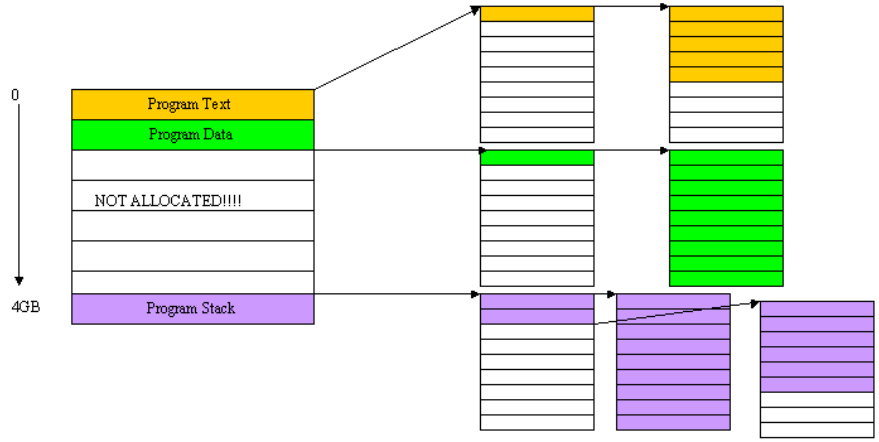
Larger block sizes are more efficient to read/write because of constant setup times for those operations. But memory usage gets less optimal the larger the block sizes are: On average 50% in all end blocks are wasted. This is called internal fragmentation.

Page Table Problems: size and speed

A 32-bit address space creates 1 million pages at 4KB page size. This means 1 million page table entries! (How many for a 64 bit address space? Just cut off the lowest 12 bits from a 64 bit address. The rest is your number of pages!

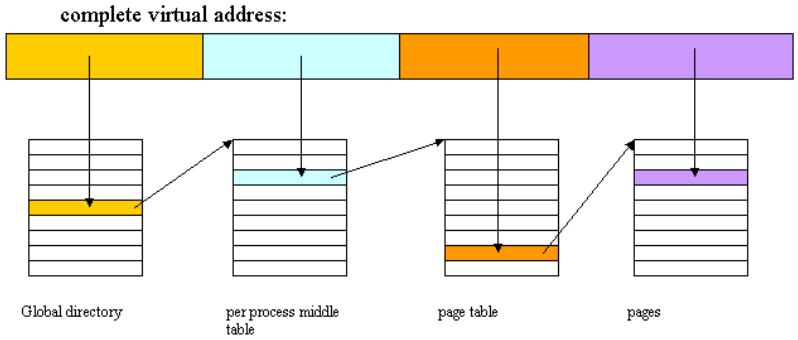
On EVERY memory access those tables have to be consulted! This can be a huge performance hit.

Multi-level page tables for sparse maps



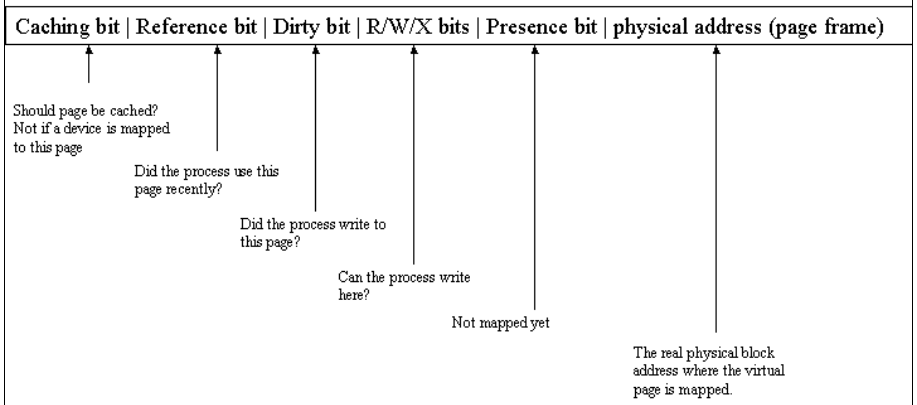
Multi-level page tables leverage the fact that most processes stay far beyond the theoretical virtual address space limit (e.g. 4GB). In fact, they need some memory for text, data and stack segment at addresses far apart. Everything between is just empty and need not be allocated at first.

Organizing blocks into page table hierarchies



This 3-level page table organization is used by Linux (see Tanenbaum pg. 721 ff.) The whole virtual address is split into several different indices which finally select a specific page

A page table entry

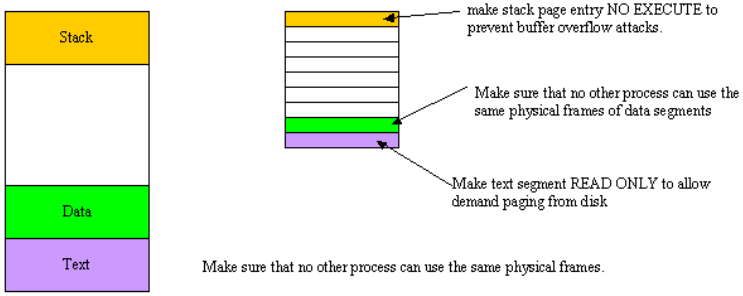


Page Table Games or the power of indirection

1. Protecting memory segments
2. Implement demand paging to grow segments automatically
3. Sharing memory between processes
4. Saving useless copies through copy-on-write
5. Finding the best pages to remove if low on memory.

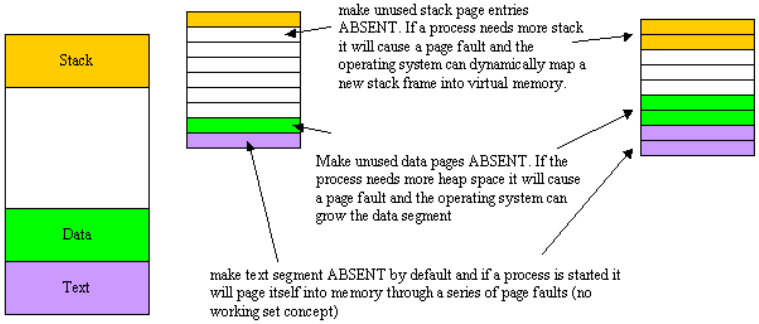
Those tricks use the fact that some bits in a page table entry are updated by hardware (e.g. reference bit) and that instead of copying lots of memory blocks around one can simply change block addresses in page table entries.

Protecting Memory Segments



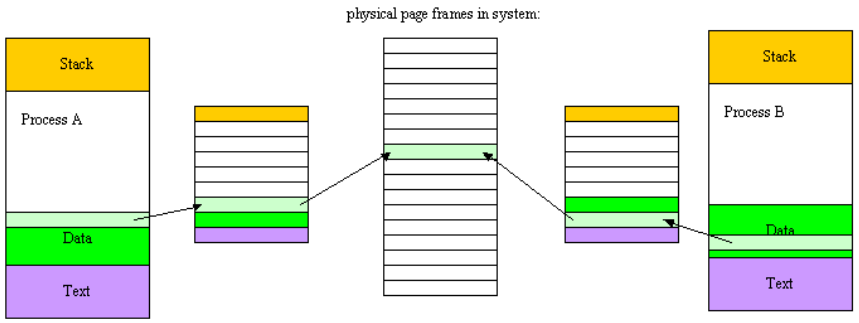
It is through the bits in the page table entries that the operating system protects a processes pages. Buffer overflows are security attacks where attack code is planted on the stack and executed there. Preventing execution in the stack segment is one way to prevent those attacks (at least partially). Most paging systems lock code (text) during execution. Self-modifying code proved to be extremely hard to program.

Grow segments automatically: demand paging



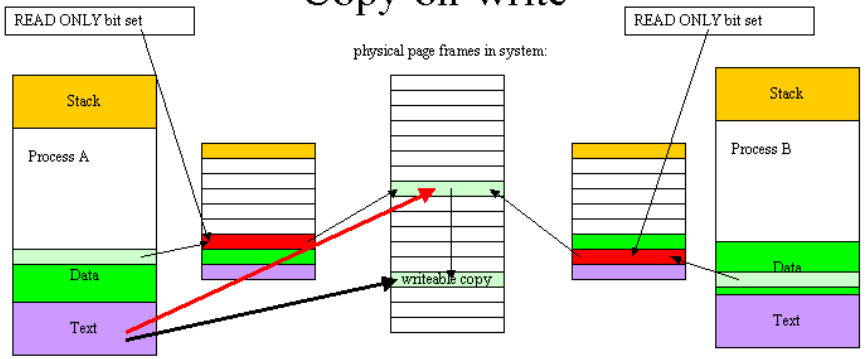
Growing segments automatically after page faults is a very elegant way to increase process resources. It has some limitations though: when does the OS know that a page fault should result in a new page allocated or the program killed because of a program execution bug? And starting a program through a series of expensive page faults is not really cheap. Better solutions use the concept of a „working set“ of pages that must be present for a process to run.

Sharing Memory Between Processes



Simply by putting the same physical address into two different page table entries from different processes they can share this piece of memory. This is usually caused by both processes executing a special memory map system call. Notice that the shared memory is in different virtual memory locations in both processes.

Copy-on-write



During process creation parent and child process share addess spaces. To prevent data corruption the OS sets the READ ONLY bits in the page table entries. If a process tries a write a page fault happens which makes the OS allocate a new physical frame as a writeable copy of the shared frame. From then on both processes have writeable frames. Instead of copying page frames for the new process only the page table entries are copied and protected.

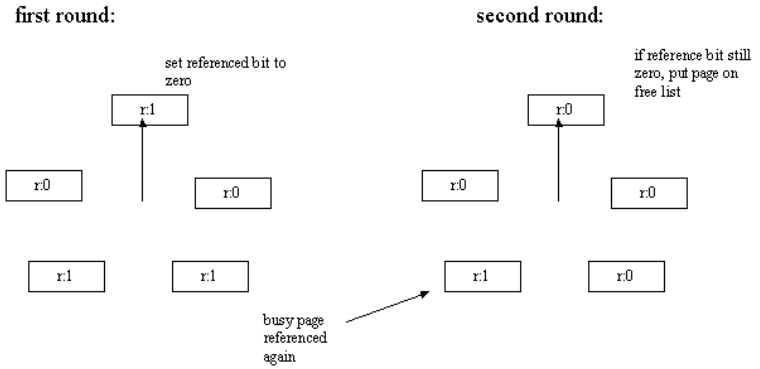
Page Replacement Algorithms

xxxx| Reference bit | Dirty bit | xxxx

1. Not referenced, not modified (good to kick out)
2. not referenced, modified (after reset of reference bit)
3. referenced, not modified (busy but clean)
4. referenced, modified

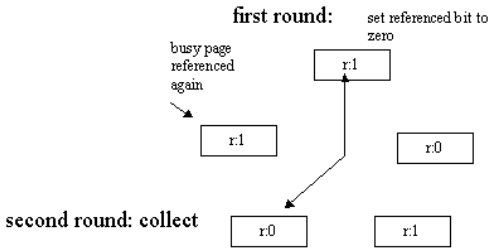
These bit combinations allow the system to implement a page freeing algorithm based on page usage. The key to success is to always have free pages available and not to make some free when you need them. Most systems use a „page daemon“ background process which scans the page usage bits. Dirty pages are scheduled to be written to swap space and clean/unmodified pages just get on the free list. Text pages are never saved (they are unchanged) because they will be paged in from disk if needed again.

The clock collection algorithm



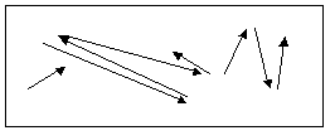
In a first go all referenced bits are set to zero. Between turns a busy page might be referenced again and stays in memory. Otherwise if on the second turn a page reference bit is still zero the page gets kicked out.

The two-handed clock algorithm



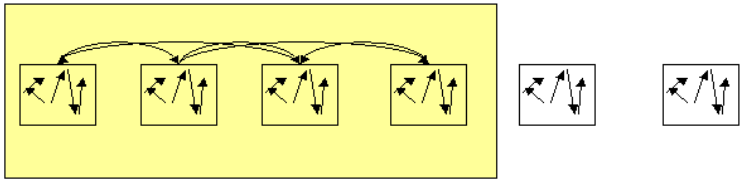
With large amounts of memory it takes one hand too long to search through all pages. A second „hand“ is used as another pointer into memory which will perform the second round task of freeing all unreferenced pages. Busy pages now need to be referenced in the time between first and second hand. The whole algorithm is usually performed by the page daemon, a background process. Best performance is achieved by making sure that there is always a supply of free pages (i.e. the page daemon runs more often if more memory is needed)

Intra-Page Locality of Reference



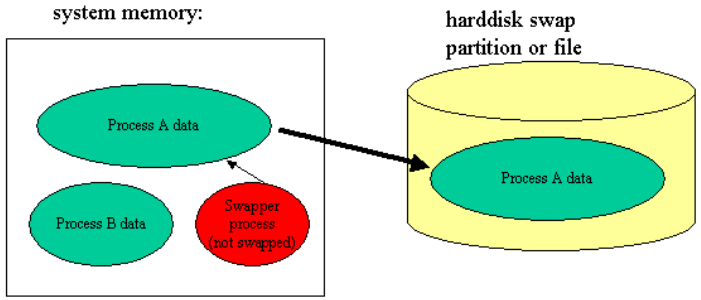
A good compiler tries to maximize intra-page locality by grouping functions which use each other into one page. Such references will never cause a page fault and are especially well suited for caching. Unfortunately this is not always possible e.g. because code from different modules must run.

Inter-Page Locality: Working Set



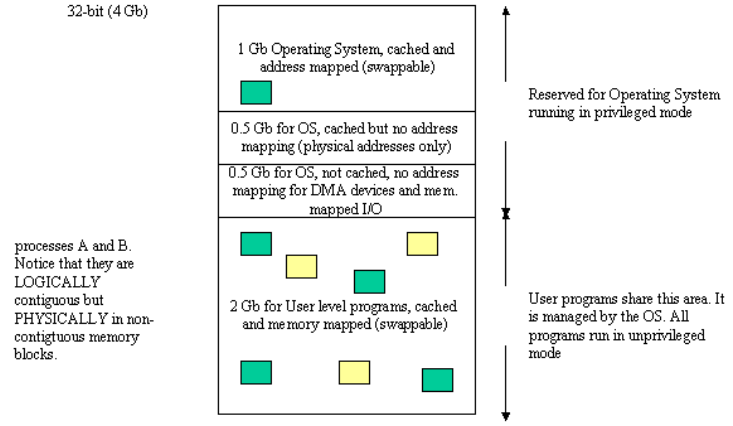
The yellow pages are called the current working set of a process: those pages which are needed in memory to avoid unnecessary page faults (thrashing). The working set changes over time. To calculate whether a page belongs to the working set a virtual time value is set for every page during access. If later on the current time – time of last access is over a certain threshold, the page is considered to be no longer in the current working set.

If all memory is in use: swapping



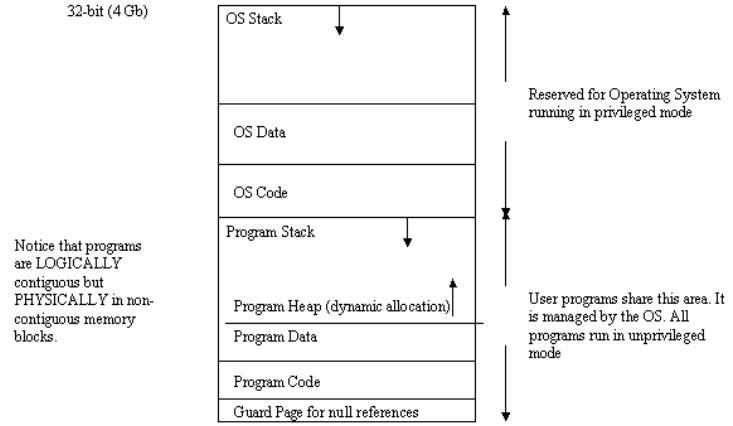
An operating system can swap out complete processes if the system memory is all used up. It will usually pick large processes for this and wait to swap them back in for some time to avoid thrashing. Before paging most OS used this mechanism which is more of a last resort nowadays. Text need not be stored because it can be reloaded from disk easily. A system process called „swapper“ selects processes for swapping.

System Memory Organization



(extended) Example of MIPS Risc architecture taken from Bacon/Harris. These separations are created by programming the memory management unit (MMU).

Per Process Memory Organization



Notice that stack and heap areas of programs can grow during runtime. Remember that every function call places values on the stack. A series of calls makes the stack grow depending on how many local variables are allocated. When heap and stack meet the program has reached ist maximum size. Also notice that the kernel code is part of the program address space but runs in protected mode.

Memory Related System Calls in Unix

- s = brk(addr) // change data segment size to addr – 1
- address = mmap(addr, len, prot, flags, fd, offset) // map open file into data segment of process
- status = unmap(address, len) // remove mapped file from process memory

There are not many such calls. The reason for this is simple: many memory related tasks are performed automatically by the kernel, e.g. if the process needs additional stack space or a new piece of code needs to be loaded into memory. If the running program tries to access its code a so called page fault results and the kernel takes over control. It loads the code into memory and hands control back to the process.

Allocating different memory sizes

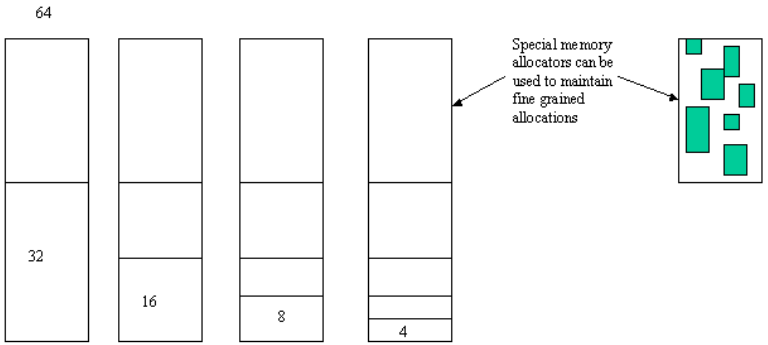


Diagram from Tanenbaum pg. 721. The buddy algorithm avoids total fragmentation of memory by merging neighbouring chunks if they become free. Allocation is by power of two only which can waste quite a bit of memory. Internal fragmentation is high.

Tracing Memory Access

1. Simulation Guided Memory Analyzer SiGMA is a toolkit designed to help programmers understand the precise memory references in scientific programs that are causing poor use of the memory subsystem. Detailed information such as this is useful for tuning loop kernels, understanding the cache behavior of new algorithms, and investigating how different parts of a program compete for and interact within the memory subsystem. http://www.alphaworks.ibm.com/tech/sigma?Open&ca=daw-flts-032703
2. Common os tools: ps, top, sar tell about memory usage

Resources (1)

- Jean Bacon, Tim Harris, Operating Systems. Concurrent and distributed software design,
- Andrew Tanenbaum, Modern Operating Systems, 2nd edition.
- Homework: http://research.microsoft.com/~lampson/33-Hints/WebPage.html Butler Lampson, Hints for computer system design. Read it – understand it. We will discuss it in the next session.