

Memory Management

S.Rajarajan APII/CSE/SASTRA





Memory management basics

- Systems don't have infinite RAM
- Do have a memory hierarchy-
 - Cache (fast)
 - Main(medium)
 - Disk(slow)
- Applications require an infinite amount of memory.

Abstraction



- Over the years, people discovered the concept of a **memory hierarchy**.
- Computers have a few megabytes of ***very fast, expensive, volatile cache memory***, a few gigabytes of ***medium-speed, medium-priced, volatile main memory***, and a few terabytes of slow, ***cheap, nonvolatile magnetic or solid-state disk storage***, not to mention removable storage, such as DVDs and USB sticks.
- It is the job of the operating system to **abstract** this hierarchy into a useful model and then manage the abstraction.



- The part of the operating system that manages (part of) the memory hierarchy is called the **memory manager**.
- Its job is to efficiently manage memory:
 - keep track of which parts of memory are in use, **allocate memory to processes** when they need it, and
 - **de-allocate** it when they are done.

Basic Memory Management



- Memory management systems can be divided into two basic classes:
 - those that move processes back and forth between main memory and disk during execution (swapping and paging),
 - and those that do not

No memory abstraction



- The simplest memory abstraction is to have no abstraction at all.
- Early mainframe computers, early minicomputers, and early personal computers (before 1980) had no memory abstraction.
- Every program simply saw the **physical memory**.
- When a program executed an instruction like
 - `MOV REGISTER1,1000`
- The computer just moved the contents of physical memory location 1000 to *REGISTER1*.



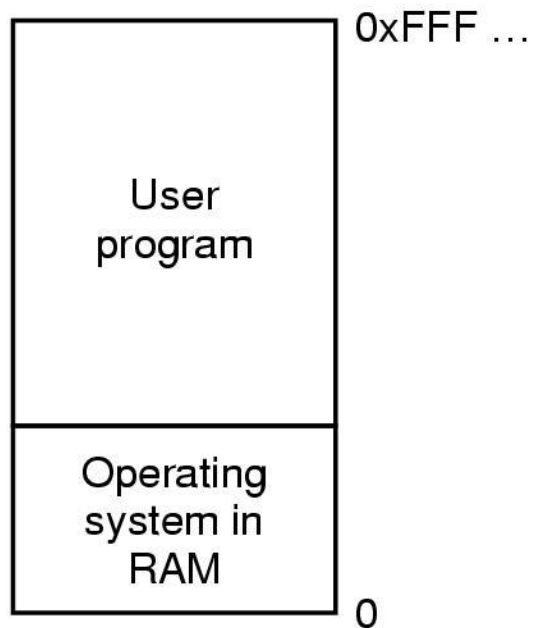
- Under these conditions, it was **not possible to have two running programs** in memory at the same time.
- If the first program wrote a new value to, say, location 2000, this would erase whatever value the second program was storing there.
- Nothing would work and both programs would crash almost immediately.
- Even with the model of memory being just physical memory, several options are possible



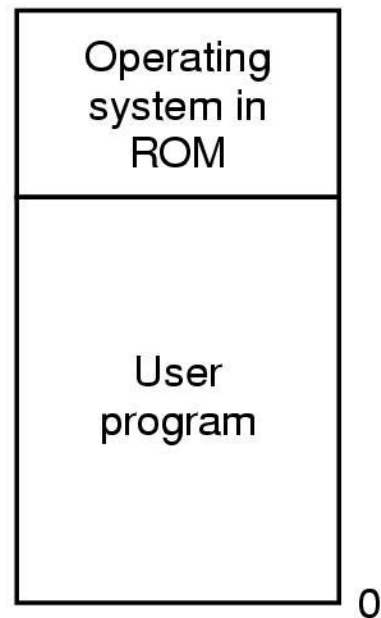
- Three variations are shown in Fig. 3-1.
 - Operating system may be at the bottom of memory in RAM (Random Access Memory)
 - It may be in ROM (Read-Only Memory) at the top of memory
 - The device drivers may be at the top of memory in a ROM and the rest of the system in RAM down below
- The first model was formerly used on ***mainframes*** and ***minicomputers*** but is rarely used any more.
- The second model is used on some ***handheld computers*** and ***embedded systems***.



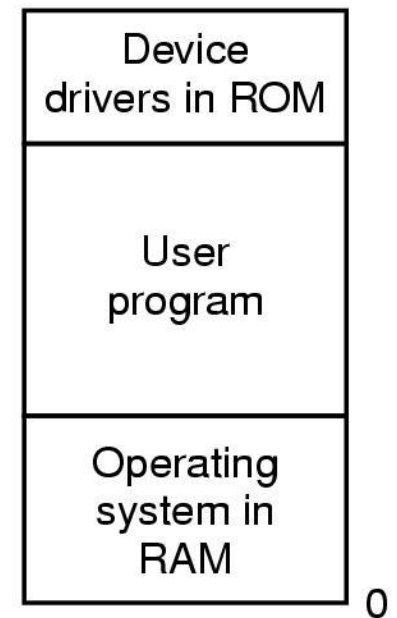
- The third model was used by early personal computers (e.g., running **MSDOS**), where the portion of the system in the ROM is called the **BIOS** (Basic Input Output System).
- Models (a) and (c) have the disadvantage that a bug in the user program can wipe out the operating system, possibly with disastrous results.



(a)



(b)



(c)



- When the system is organized in this way, generally only one process at a time can be running.
- As soon as the user types a command, the operating system copies the requested program from disk to memory and executes it.
- When the process finishes, the operating system displays a prompt character and waits for a user new command.
- When the operating system receives the command, it ***loads a new program into memory, overwriting the first*** one.



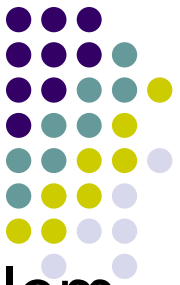
- One way to get some parallelism in a system with no memory abstraction is to program with ***multiple threads***.
- Since all threads in a process are supposed to see the same memory image, the fact that they are forced to is not a problem.
- While this idea works, it is of limited use since what people often want is **unrelated programs** to be running at the same time, something the threads abstraction does not provide.

Running Multiple Programs Without a Memory Abstraction



- However, even with no memory abstraction, it is possible to run multiple programs at the same time.
- What the operating system has to do is save the entire contents of memory to a **disk file**, then bring in and run the next program.
- As long as there is only one program at a time in memory, there are no conflicts.
- With the ***addition of some special hardware***, it is possible to run multiple programs concurrently, even without swapping.

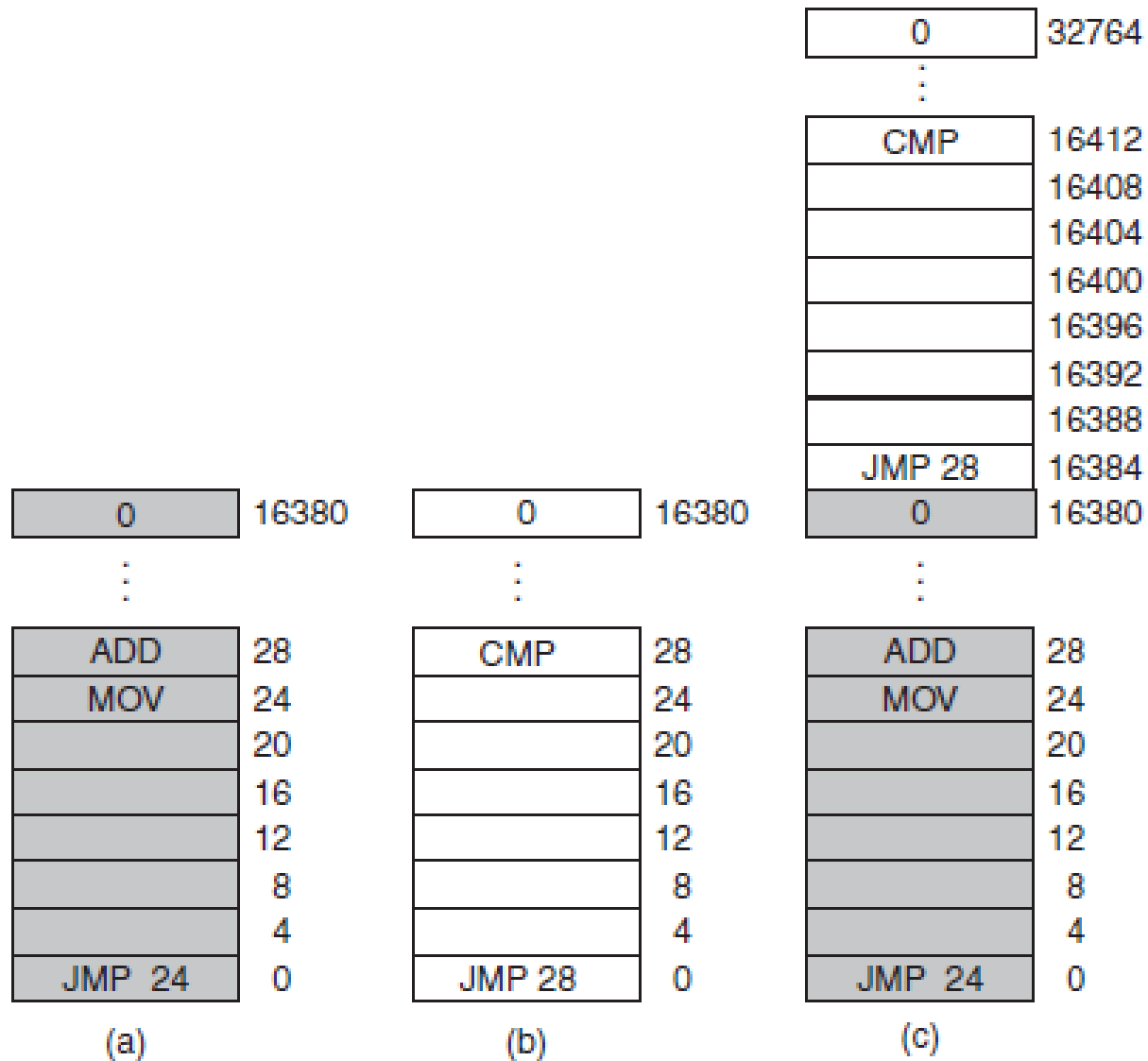
IBM 360 Solution



- The early models of the IBM 360 solved the problem as follows.
- Memory was divided into **2-KB blocks** and each was assigned a **4-bit protection key** held in special registers inside the CPU.
- A machine with a **1-MB (1024 KB / 2 KB)** memory needed only **512** of these **4-bit** registers for a total of **256 bytes** of key storage.
- The PSW (Program Status Word) also contained a **4-bit key**.
- The 360 hardware trapped any attempt by a running process to access memory with a protection code different from the PSW key.

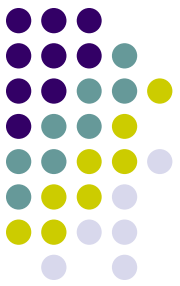


- Since only the operating system could change the protection keys, user processes were prevented from interfering with one another and with the operating system itself.
- Nevertheless, this solution had a major drawback, depicted in Fig. 3-2.





- The core problem here is that the two programs both reference absolute physical memory.
- What we want is that each program can reference a private set of addresses local to it.
- What the IBM 360 did as a stop-gap solution was modify the second program on the fly as it loaded it into memory using a technique known as **static relocation**.



- It worked like this. When a program was loaded at address 16,384, the constant 16,384 was added to every program address during the load process (so “JMP 28” became “JMP 16,412”, etc.).
- While this mechanism works if done right, it is **not a very general solution** and **slows down loading**.
- Furthermore, it **requires extra information** in all executable programs to indicate which words contain **(relocatable) addresses** and which do not.
- After all, the “28” in example has to be relocated but an instruction like
 - MOV REGISTER 1,28
- which moves the number 28 to *REGISTER1* must not be relocated



- While direct addressing of physical memory is but a distant memory on mainframes, minicomputers, desktop computers, notebooks, and smart phones, the lack of a memory abstraction is still common in embedded and smart card systems.
- Devices such as radios, washing machines, and microwave ovens are all full of software (in ROM) these days, and in most cases the software addresses absolute memory.
- This works because all the programs are known in advance and users are not free to run their own software



- While high-end embedded systems (such as smart phones) have elaborate operating systems, simpler ones do not.
- In some cases, there is an operating system, but it is just a library that is linked with the application program and provides system calls for performing I/O and other common tasks.
- The **e-Cos** operating system is a common example of an operating system as library.

MEMORY ABSTRACTION: ADDRESS SPACES



- All in all, exposing physical memory to processes has several major drawbacks.
 - First, if user programs can address every byte of memory, they can easily trash the operating system, intentionally or by accident.
 - Second, with this model, it is difficult to have multiple programs running at once (taking turns, if there is only one CPU).

The Notion of an Address Space



- Two problems have to be solved to allow multiple applications to be in memory at the same time without interfering with each other: **protection** and **relocation**.
- A better solution is to invent a new abstraction for memory: the **address space**.
- Just as the process concept creates a kind of abstract CPU to run programs, the address space creates a kind of abstract memory for programs to live in.
- An **address space** is the set of addresses that a process can use to address memory.
- Each process has its own address space, independent of those belonging to other processes (except in some special circumstances where processes want to share their address spaces).

Base and Limit Registers



- This simple solution uses a particularly simple version of **dynamic relocation**.
- What it does is map each process' address space onto a different part of physical memory in a simple way.
- Classical solution which was used is to equip each CPU with two special hardware registers, usually called the **base** and **limit** registers
- When these registers are used, programs are loaded into consecutive memory locations wherever there is room and without relocation during loading,



- When a process is run, the **base register** is **loaded** with the physical address where its **program begins** in memory and the **limit register** is loaded with the **length of the program**.
- Every time a process references memory, either to fetch an instruction or read or write a data word, the CPU hardware automatically adds the base value to the address generated by the process before sending the address out on the memory bus.
- Simultaneously, it checks whether the address offered is equal to or greater than the value in the limit register, in which case a fault is generated and the access is aborted.



- Using base and limit registers is an easy way to give **each process its own private address space** because every memory address generated automatically has the base-register contents added to it before being sent to memory.
- In many implementations, the base and limit registers are protected in such a way that **only the operating system can modify them**.
- A disadvantage of relocation using base and limit registers is the **need to perform an addition and a comparison** on every memory reference.
- Comparisons can be done fast, but **additions are slow due to carry-propagation time** unless special addition circuits are used.



Swapping

- If the physical memory of the computer is large enough to hold all the processes, the schemes described so far will more or less do.
- But in practice, the total amount of RAM needed by all the processes is often much more than can fit in memory.
- Two general approaches to dealing with memory overload have been developed over the years.
- The simplest strategy, called **swapping**, consists of bringing in each process in its entirety, running it for a while, then putting it back on the disk.



- Idle processes are mostly stored on disk, so they do not take up any memory when they are not running.
- The other strategy, called **virtual memory**, allows programs to run even when they are only partially in main memory.
- The operation of a swapping system is illustrated in Fig.
- Initially, only process *A* is in memory. Then processes *B* and *C* are created or swapped in from disk. In Fig. 3-4(d) *A* is swapped out to disk. Then *D* comes in and *B* goes out. Finally *A* comes in again. Since *A* is now at a different location, addresses contained in it must be relocated, either by software when it is swapped in or (more likely) by hardware during program execution e.g base & limit

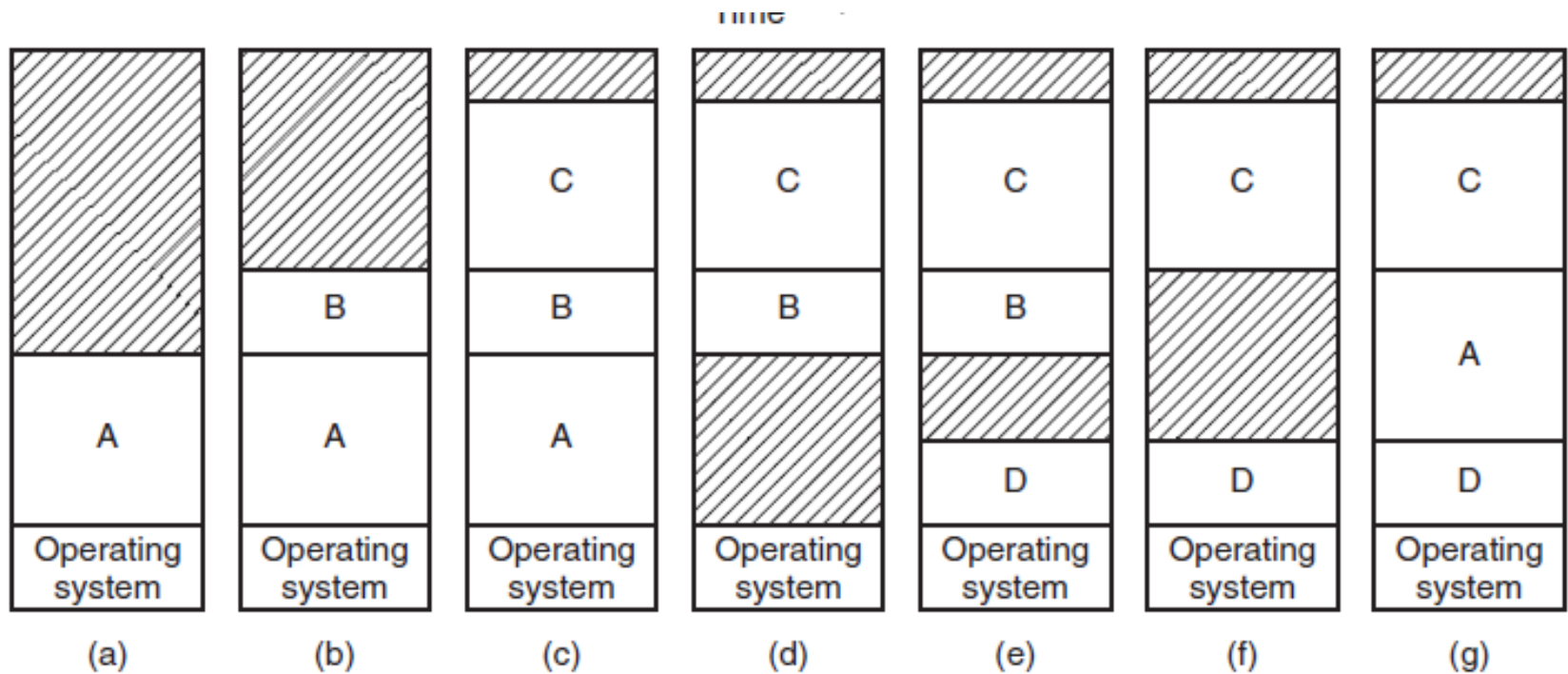
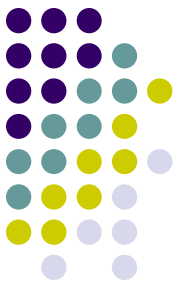


Figure 3-4. Memory allocation changes as processes come into memory and leave it. The shaded regions are unused memory.



- When swapping creates multiple holes in memory, it is possible to combine them all into one big one by moving all the processes downward as far as possible.
- This technique is known as **memory compaction**
- It is usually not done because it requires a lot of CPU time. For example, on a 16-GB machine that can copy 8 bytes in 8 nsec, it would take about 16 sec to compact all of memory.
- A point that is worth making concerns how much memory should be allocated for a process when it is created or swapped in.
- If processes are created with a fixed size that never changes, then the allocation is simple: the operating system allocates exactly what is needed, no more and no less



- If, however, **processes' data segments can grow**, for example, by dynamically allocating memory from a heap, as in many programming languages, a problem occurs whenever a process tries to grow.
- If a **hole is adjacent to the process**, it can be allocated and the process allowed to grow into the hole.
- On the other hand, if the **process is adjacent to another process**, the growing **process will either have to be moved to a hole in memory large enough** for it, or one or more processes will have to be swapped out to create a large enough hole.
- If a process cannot grow in memory and the swap area on the disk is full, the **process will have to be suspended** until some space is freed up



- If it is **expected that most processes will grow** as they run, it is probably a good idea to allocate a **little extra memory** whenever a process is swapped in or moved, to reduce the overhead associated with moving or swapping processes that no longer fit in their allocated memory.
- However, **when swapping processes** to disk, only the memory actually in use should be swapped; it is wasteful to swap the extra memory as well.



- If processes can have **two growing segments**—for example, the **data segment** being used as a **heap** for **variables that are dynamically allocated** and released and a **stack segment** for the normal **local variables** and return addresses—an alternative arrangement suggests itself.
- Each process illustrated has a stack at the top of its allocated memory that is growing downward, and a data segment just beyond the program text that is growing upward.
- The memory between them can be used for either segment. If it runs out, the process will either have to be moved to a hole with sufficient space, swapped out of memory until a large enough hole can be created, or killed.

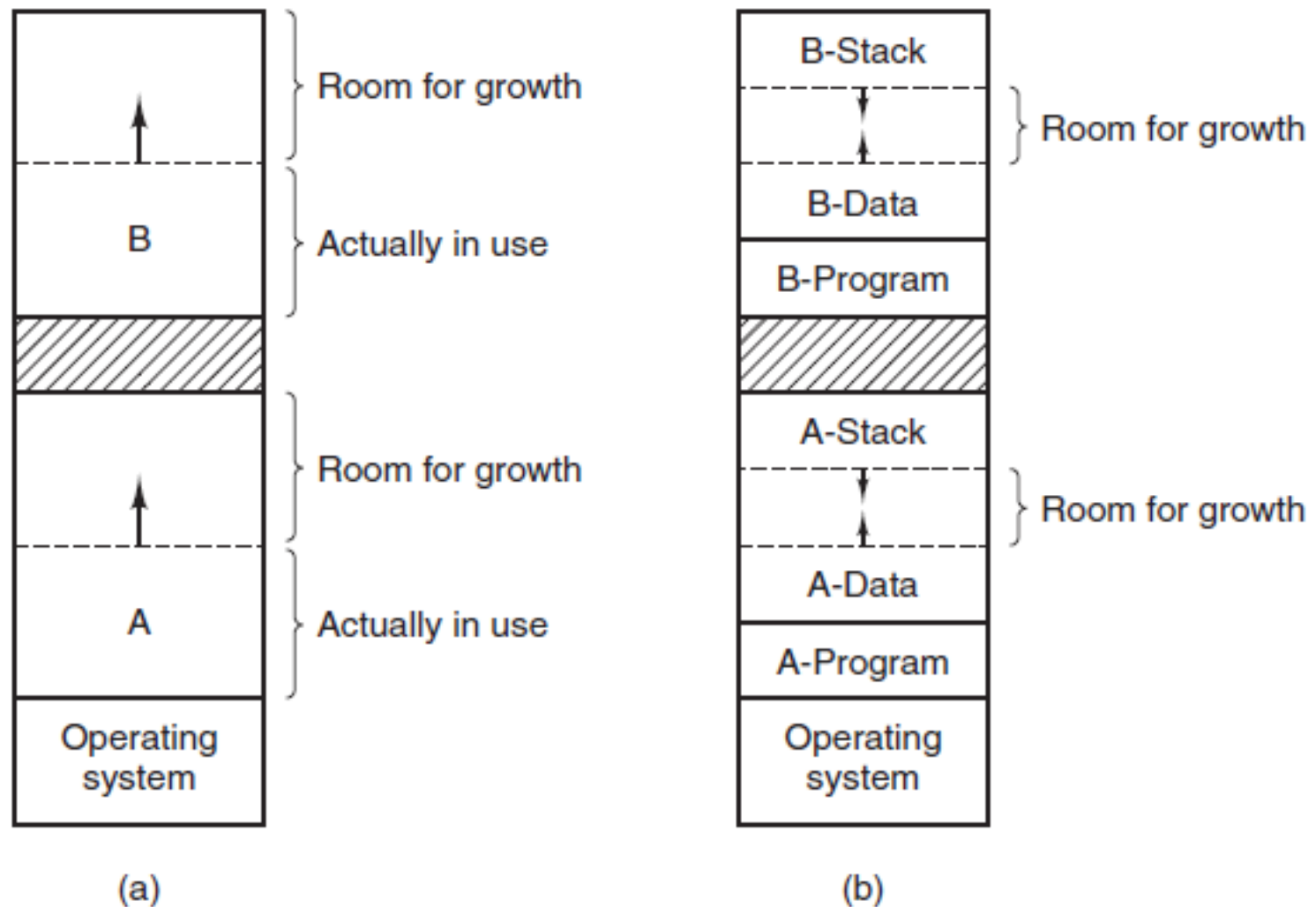


Figure 3-5. (a) Allocating space for a growing data segment. (b) Allocating space for a growing stack and a growing data segment.

Managing Free Memory

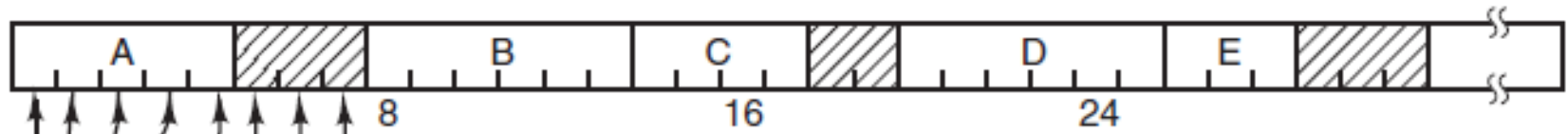
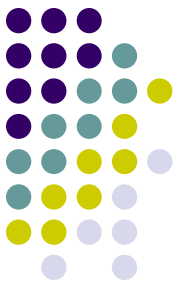


- When memory is assigned dynamically, the operating system must manage it.
- In general terms, there are two ways to keep track of memory usage: ***bitmaps*** and ***free lists***.

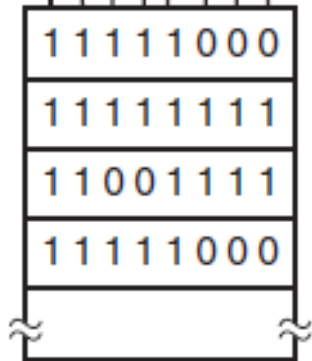
Memory Management with Bitmaps



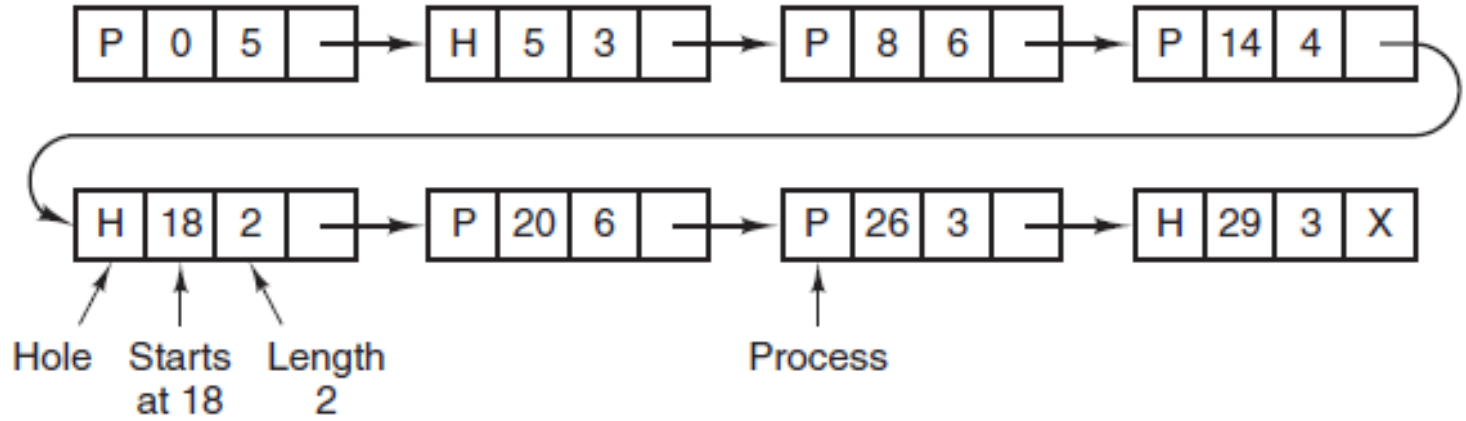
- With a bitmap, memory is divided into **allocation units** as small as a few words and as large as several kilobytes
- Corresponding to each allocation unit is a bit in the bitmap, which is **0 if the unit is free** and **1 if it is occupied**
- The **size of the allocation unit** is an important design issue.
- The smaller the allocation unit, the larger the bitmap.
- However, even with an allocation unit as small as 4 bytes, 32 bits of memory will require only 1 bit of the map.



(a)



(b)



(c)

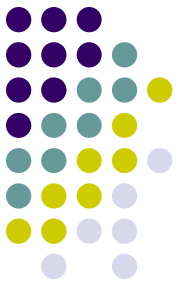


- A memory of $32n$ bits will use n map bits, so the bitmap will take up only $1/32$ of memory.
- If the **allocation unit is chosen large**, the **bitmap will be smaller**, but appreciable **memory may be wasted in the last unit** of the process if the process size is not an exact multiple of the allocation unit
- A bitmap provides a simple way **to keep track of memory words** in a fixed amount of memory because the size of the bitmap depends only on the size of memory and the size of the allocation unit



- The main problem is that when it has been decided to bring a ***k*-unit process** into memory, the memory manager must search the bitmap to find a run of ***k* consecutive 0 bits in the map.**
- Searching a bitmap for a run of a given length is a slow operation

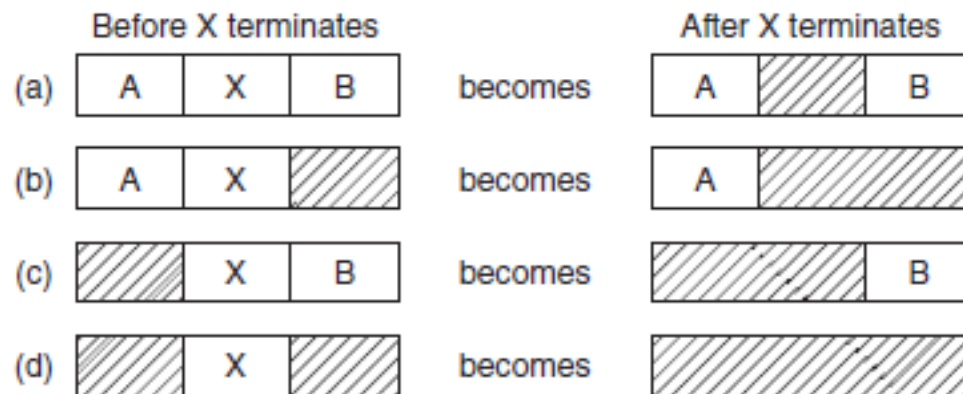
Memory Management with Linked Lists



- Another 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.
- Each entry in the list specifies a **hole (H)** or **process (P)**, the **address at which it starts**, the **length**, and a **pointer to the next** item.
- In this example, the segment list is **kept sorted by address**.
- Sorting this way has the advantage that when a process terminates or is swapped out, updating the list is straightforward.



- A terminating process normally has two **neighbors** (except when it is at the very top or bottom of memory).
- These may be **either processes or holes**, leading to the four combinations.





- When the processes and holes are kept on a list sorted by address, several algorithms can be used to allocate memory for a created process (or an existing process being swapped in from disk).
- We assume that the memory manager knows how much memory to allocate.


Memory allocation algorithms



- The simplest algorithm is **first fit**.
- The memory manager scans along the list of segments until it finds a hole that is big enough.
- The hole is then broken up into two pieces, one for the process and one for the unused memory, except in the statistically unlikely case of an exact fit.
- First fit is a fast algorithm because it searches as little as possible.
- A minor variation of first fit is **next fit**. It works the same way as first fit, except that it keeps track of where it is whenever it finds a suitable hole.
- The next time it is called to find a hole, it starts searching the list from the place where it left off last time, instead of always at the beginning, as first fit does.



- Another well-known and widely used algorithm is **best fit**.
- Best fit searches the entire list, from beginning to end, and takes the smallest hole that is adequate.
- Rather than breaking up a big hole that might be needed later, best fit tries to find a **hole that is close to the actual size** needed, to best match the request and the available holes.

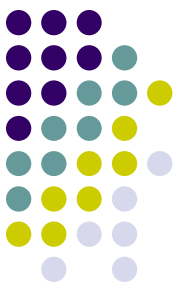
- 
- Best fit is **slower than first fit** because it must search the entire list every time it is called.
 - Somewhat surprisingly, it also **results in more wasted memory** than first fit or next fit because it tends to fill up memory with tiny, useless holes.
 - **First fit generates larger holes on the average.**
 - To get around the problem of breaking up nearly exact matches into a process and a tiny hole, one could think about **worst fit**, that is, always take the largest available hole, so that the new hole will be big enough to be useful.



- All four algorithms can be speeded up by **maintaining separate lists for processes and holes.**
- In this way, all of them devote their full energy to inspecting holes, not processes.
- If distinct lists are maintained for processes and holes, the **hole list may be kept sorted on size**, to make best fit faster.
- When **best fit searches a list of holes from smallest to largest**, as soon as it finds a hole that fits, it knows that the hole is the smallest one that will do the job.
- With a hole list sorted by size, **first fit and best fit are equally fast**, and next fit is pointless.



- The price that is paid for this speedup on allocation is the additional complexity and slowdown when deallocating memory, since a **freed segment has to be removed** from the process list and **inserted into the hole list**.
- When the holes are kept on separate lists from the processes, a small optimization is possible.
- Instead of having a separate set of data structures for maintaining the hole list, as is done in Fig. 3-6(c), the information can be stored in the holes.
- The **first word of each hole** could be the **hole size**, and the **second word a pointer** to the following entry



- Yet another allocation algorithm is **quick fit**, which **maintains separate lists for some of the more common sizes** requested.
- With quick fit, finding a hole of the required size is extremely fast, but it has the same disadvantage as all schemes that sort by hole size, namely, when a process terminates or is swapped out, finding its neighbors to see if a merge with them is possible is quite expensive.
- If merging is not done, memory will quickly fragment into a large number of small holes into which no processes fit.

Virtual Memory



- While base and limit registers can be used to create the abstraction of address spaces, there is another problem that has to be solved: **managing bloatware**.
- While memory sizes are increasing rapidly, software sizes are increasing much faster.
- The trend toward multimedia puts even more demands on memory.
- As a consequence of these developments,
 - **there is a need to run programs that are too large to fit in memory,**
 - **and there is certainly a need to have systems that can support multiple programs running simultaneously,** each of which fits in memory but all of which collectively exceed memory



- Swapping is not an attractive option, since a typical SATA disk has a peak transfer rate of several hundreds of MB/sec, which means it takes seconds to swap out a 1-GB program and the same to swap in a 1-GB program.
- A solution adopted in the 1960s was to split programs into little pieces, called **overlays**.
- The main problem in Fixed partitioning is the size of a process has to be limited by the maximum size of the partition, which means a process can never be span over another.
- In order to solve this problem, earlier people have used some solution which is called as Overlays.



Drawbacks

- Overlap map must be specified by programmer
- Programmer must know memory requirement
- Overlapped module must be completely disjoint
- Programming design of overlays structure is complex and not possible in all cases



- The best example of overlays is assembler.
- Consider the assembler has 2 passes, 2 pass means at any time it will be doing only one thing, either the 1st pass or the 2nd pass.
- Which means it will finish 1st pass first and then 2nd pass.
- Let assume that available main memory size is 150KB and total code size is 200KB

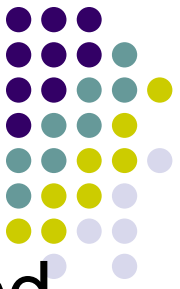


- Pass 1..... 70KB
- Pass 2..... 80KB
- Symbol table..... 30KB
- Common routine..... 20KB
- As the total code size is 200KB and main memory size is 150KB, it is not possible to use 2 passes together.
- So, in this case, we should go with the overlays technique.
- According to the overlays concept at any time only one pass will be used and both the passes always need **symbol table and common routine**

Overlay Driver



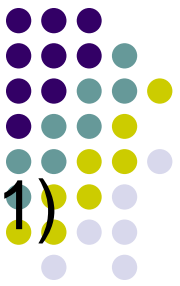
- It is the user responsibility to take care of overlaying, the operating system will not provide anything.
- Which means the user should write even what part is required in the 1st pass and once the 1st pass is over, the user should write the code to pull out the pass 1 and load the pass 2.
- That is what is the responsibility of the user, that is known as the Overlays driver.
- Overlays driver will just help us to move out and move in the various part of the code.



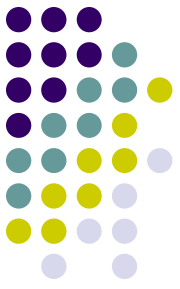
- When a program started, all that was loaded into memory was the overlay manager, which immediately loaded and ran overlay 0.
- When it was done, it would tell the overlay manager to load overlay 1, either above overlay 0 in memory (if there was space for it) or on top of overlay 0 (if there was no space).
- Some overlay systems were highly complex, allowing many overlays in memory at once.



- The overlays were kept on the disk and swapped in and out of memory by the overlay manager.
- Although the actual work of swapping overlays in and out was done by the operating system, the work of splitting the program into pieces had to be **done manually by the programmer**.
- Splitting large programs up into small, modular pieces was time consuming, boring, and error prone.



- The method that was devised (Fotheringham, 1961) has come to be known as **virtual memory**.
- The basic idea behind virtual memory is that each program has its own address space, which is broken up into chunks called **pages**.
- Each **page is a contiguous range of addresses**. These pages are mapped onto physical memory, but not all pages have to be in physical memory at the same time to run the program.
- When the program references a part of its address space that is in physical memory, the hardware performs the necessary mapping on the fly.

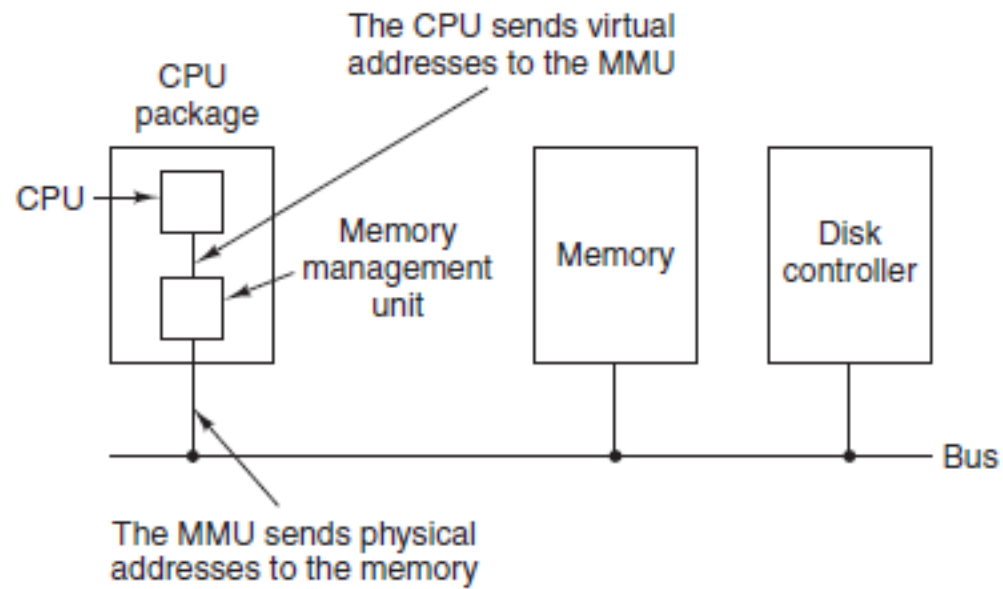
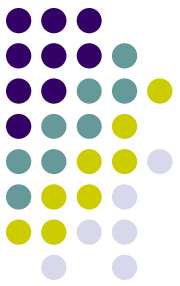


- When the program references a part of its address space that is *not* in physical memory, the operating system is alerted to go get the missing piece and re-execute the instruction that failed.
- Virtual memory works just fine in a multiprogramming system, with bits and pieces of many programs in memory at once.
- While a program is waiting for pieces of itself to be read in, the CPU can be given to another process.

Paging



- Most virtual memory systems use a technique called **paging**
- On any computer, programs reference a set of memory addresses.
- When a program executes an instruction like
 - `MOV REG,1000`
- It does so to copy the contents of memory address 1000 to REG (or vice versa, depending on the computer)
- Addresses can be generated using indexing, base registers, segment registers, and other ways.



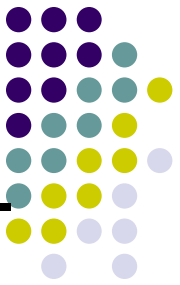


- These program-generated addresses are called **virtual addresses** and form the **virtual address space**.
- On computers without virtual memory, the address is put directly onto the memory bus and causes the physical memory word with the same address to be read or written.
- When virtual memory is used, the virtual addresses do not go directly to the memory bus. Instead, they go to an **MMU (Memory Management Unit)** that maps the virtual addresses onto the physical memory addresses



Example

- In this example, we have a computer that generates 16-bit addresses, from 0 up to $64K - 1$.
- These are the virtual addresses.
- This computer, however, has only 32 KB of physical memory.
- So although 64-KB programs can be written, they cannot be loaded into memory in their entirety and run.
- A complete copy of a program's core image, up to 64 KB, must be present on the disk, however, so that pieces can be brought in as needed.



- The virtual address space consists of fixed-size units called pages.
- The corresponding units in the physical memory are called **page frames**.
- The pages and page frames are generally the same size. In this example they are 4 KB, but page sizes from 512 bytes to a gigabyte have been used in real systems.
- With 64 KB of virtual address space, page size of 4 KB and 32 KB of physical memory, we get 16 virtual pages and 8 page frames.



- Transfers between RAM and disk are always in whole pages.
- Many processors support multiple page sizes that can be mixed and matched as the operating system sees fit
- For instance, the x86-64 architecture supports 4-KB, 2-MB, and 1-GB pages, so we could use 4-KB pages for user applications and a single 1-GB page for the kernel
- It is sometimes better to use a single large page, rather than a large number of small ones.



- The range marked 0K–4K means that the virtual or physical addresses in that page are 0 to 4095. The range 4K–8K refers to addresses 4096 to 8191, and so on.
- Each page contains exactly 4096 addresses
- When the program tries to access address 0, for example, using the instruction
 - `MOV REG,0`
- virtual address 0 is sent to the MMU. The MMU sees that this virtual address falls in page 0 (0 to 4095), which according to its mapping is page frame 2 (8192 to 12287).
- It thus transforms the address 0 to 8192 and outputs address 8192 onto the bus.



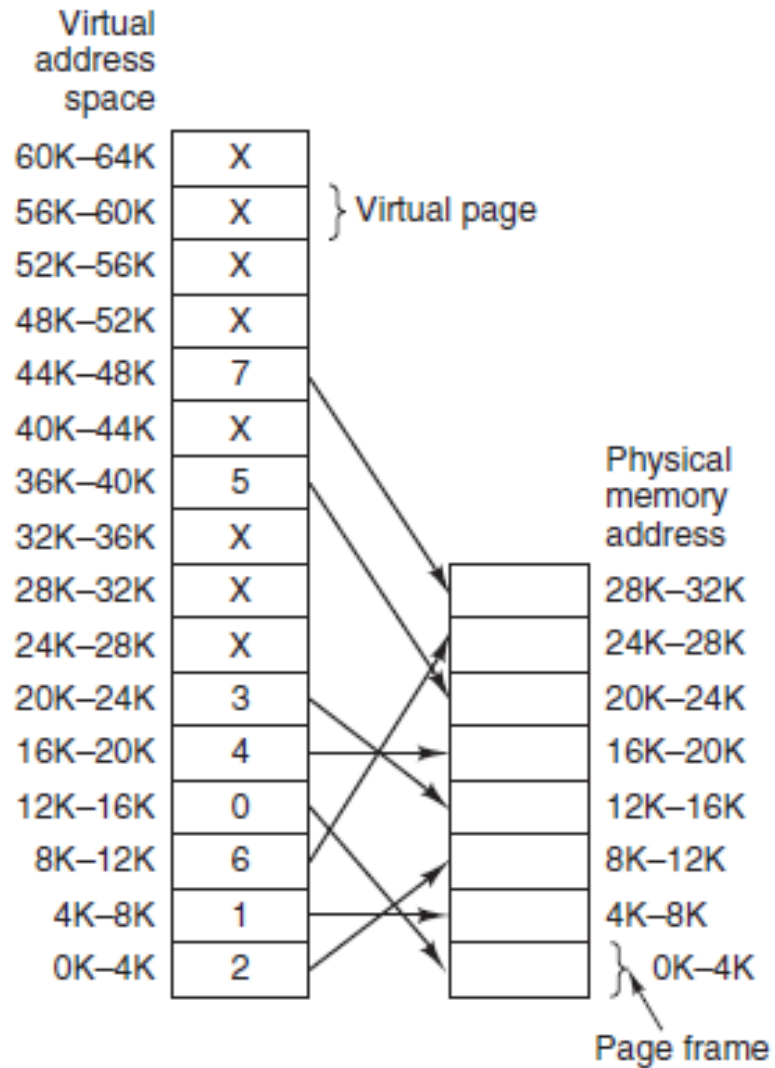
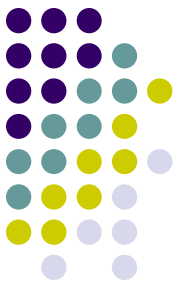
- The memory knows nothing at all about the MMU and just sees a request for reading or writing address 8192, which it honors.
- Thus, the MMU has effectively mapped all virtual addresses between 0 and 4095 onto physical addresses 8192 to 12287.
- Similarly, the instruction
 - `MOV REG,8192`
- is effectively transformed into
 - `MOV REG,24576`



- because virtual address 8192 (in virtual page 2) is mapped onto 24576 (in physical page frame 6).
- As a third example, virtual address 20500 is 20 bytes from the start of virtual page 5 (virtual addresses 20480 to 24575) and maps onto physical address $12288 + 20 = 12308$.
- By itself, this ability to map the 16 virtual pages onto any of the eight page frames by setting the MMU's map appropriately does not solve the problem that the virtual address space is larger than the physical memory.



- Since we have only eight physical page frames, only eight of the virtual pages are mapped onto physical memory. The others, shown as a cross in the figure, are not mapped.
- In the actual hardware, a **Present/absent bit** keeps track of which pages are physically present in memory
- The MMU notices that the page is unmapped (indicated by a cross in the figure) and causes the CPU to trap to the operating system. This trap is called a **page fault**

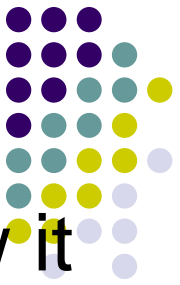




- What happens if the program references an unmapped address, for example, by using the instruction
 - MOV REG,32780
- which is byte 12 within virtual page 8 (starting at 32768)? The MMU notices that the page is unmapped and causes the CPU to trap to the operating system. This trap is called a **page fault**



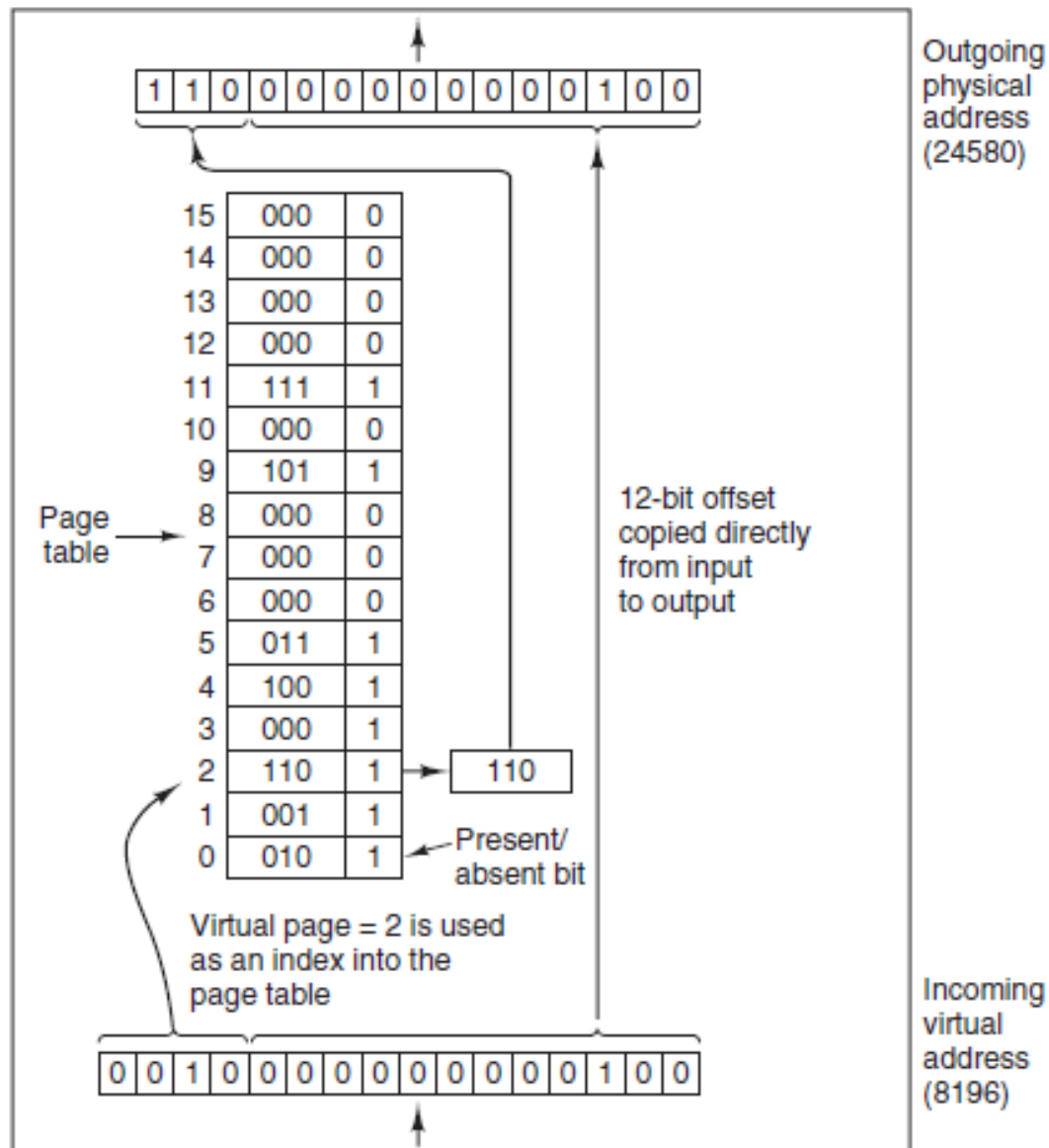
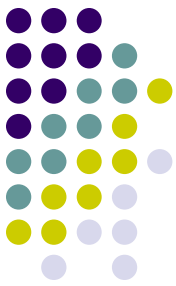
- The operating system picks a little-used page frame and writes its contents back to the disk (if it is not already there).
- It then fetches (also from the disk) the page that was just referenced into the page frame just freed, changes the map, and restarts the trapped instruction.



- Now let us look inside the MMU to see how it works and why we have chosen to use a page size that is a power of 2.
- In Fig. 3-10 we see an example of a virtual address, 8196 (00100000000000100 in binary), being mapped using the MMU map of Fig. 3-9.
- The incoming 16-bit virtual address is split into a 4-bit page number and a 12-bit offset.



- With **4 bits** for the page number, we can have **16 pages**, and with **12 bits** for the offset, we can address all 4096 bytes within a page
- The page number is used as an index into the **page table**, yielding the number of the page frame corresponding to that virtual page.
- If the *Present/absent* bit is 0, a trap to the operating system is caused. If the bit is 1, the page frame number found in the page table is copied to the high-order **3 bits** of the output register, along with the **12-bit offset**, which is copied unmodified from the incoming virtual address.
- Together they form a **15-bit** physical address
- (2 bits for 8 frames, 12 bits for 4096 bytes)





Page table

- In a simple implementation, the mapping of virtual addresses onto physical addresses can be summarized as follows:
 - the virtual address is split into a virtual page number (high-order bits) and an offset (low-order bits).
 - For example, with a 16-bit address and a 4-KB page size, the upper 4 bits could specify one of the 16 virtual pages and the lower 12 bits would then specify the byte offset (0 to 4095)



- The virtual page number is used as an index into the page table to find the entry for that virtual page.
- From the page table entry, the page frame number (if any) is found.
- The page frame number is attached to the high-order end of the offset, replacing the virtual page number, to form a physical address that can be sent to the memory.
- Thus, the purpose of the page table is to map virtual pages onto page frames.
- Mathematically speaking, the page table is a function, with the virtual page number as argument and the physical frame number as result.

Structure of a Page Table Entry



- The exact layout of an entry in the page table is highly machine dependent, but the kind of information present is roughly the same from machine to machine.

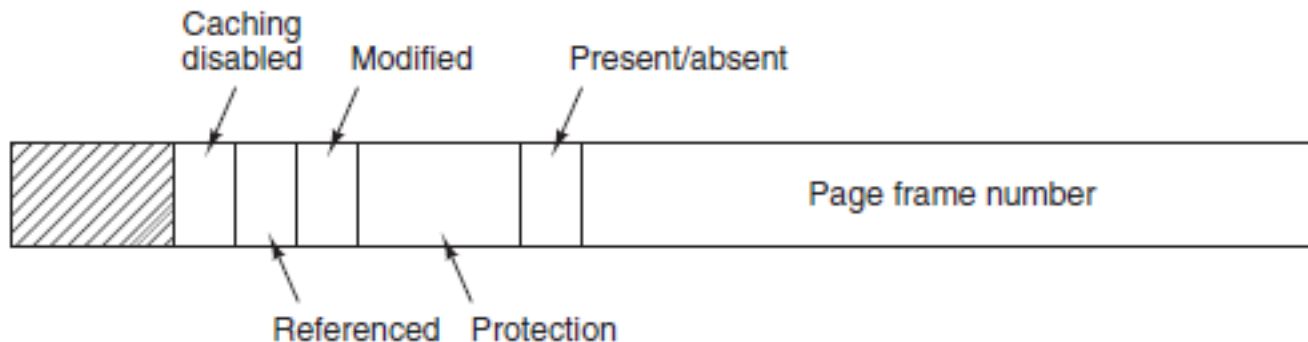
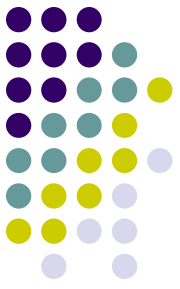


Figure 3-11. A typical page table entry.



- The ***Protection*** bits tell what kinds of access are permitted. In the simplest form, this field contains 1 bit, with 0 for read/write and 1 for read only.
- A more sophisticated arrangement is having 3 bits, one bit each for enabling reading, writing, and executing the page



- The ***Modified*** bit keep track of page usage.
- When a page is written to, the hardware automatically sets the *Modified* bit.
- This bit is of value when the operating system decides to reclaim a page frame.
- If the page in it has been modified (i.e., is “dirty”), it must be written back to the disk. If it has not been modified (i.e., is “clean”), it can just be abandoned, since the disk copy is still valid.
- The bit is sometimes called the **dirty bit**, since it reflects the page’s state.



- The ***Referenced*** bit is set whenever a page is referenced, either for reading or for writing. Its value is used to help the operating system choose a page to evict when a page fault occurs
- Finally, the last bit allows caching to be disabled for the page. This feature is important for pages that map onto device registers rather than memory.
- If the operating system is sitting in a tight loop waiting for some I/O device to respond to a command it was just given, it is essential that the hardware keep fetching the word from the device, and not use an old cached copy.
- With this bit, caching can be turned off.



Speeding Up Paging

- In any paging system, any paging system, two major issues must be faced:
- 1. The mapping from virtual address to physical address must be fast.
- 2. If the virtual address space is large, the page table will be large.
- The first point is a consequence of the fact that the virtual-to-physical mapping must be done on every memory reference.
- All instructions must ultimately come from memory and many of them reference operands in memory as well.



- Consequently, it is necessary to make one, two, or sometimes more page table references per instruction.
- If an instruction execution takes, say, 1 nsec, the page table lookup must be done in under 0.2 nsec to avoid having the mapping become a major bottleneck.
- The second point follows from the fact that all modern computers use virtual addresses of at least 32 bits, with 64 bits becoming the norm for desktops and laptops.
- With, say, a 4-KB page size, a 32-bit address space has 1 million pages, and a 64-bit address space has more than you want to contemplate



- With 1 million pages in the virtual address space, the page table must have 1 million entries.
- And remember that each process needs its own page table (because it has its own virtual address space).
- The need for large, fast page mapping is a very significant constraint on the way computers are built.



- The simplest design (at least conceptually) is to have a single page table consisting of an array of ***fast hardware registers***, with one entry for each virtual page, indexed by virtual page number.
- When a process is started up, the operating system loads the registers with the process' page table, taken from a copy kept in main memory.
- During process execution, no more memory references are needed for the page table.
- The advantages of this method are that it is straightforward and requires no memory references during mapping.
- A disadvantage is that it is unbearably expensive if the page table is large; it is just not practical most of the time



- Another one is that having to load the full page table at every context switch would completely kill performance.
- At the other extreme, the page table can be entirely in main memory.
- All the hardware needs then is a single register that points to the start of the page table.
- This design allows the virtual-to-physical map to be changed at a context switch by reloading one register.
- Of course, it has the disadvantage of requiring one or more memory references to read page table entries during the execution of each instruction, making it very slow.

Translation Lookaside Buffers



- The starting point of most optimization techniques is that the page table is in memory.
- Potentially, this design has an enormous impact on performance.
- Consider, for example, a 1-byte instruction that copies one register to another. In the absence of paging, this instruction makes only one memory reference, to fetch the instruction.
- With paging, at least one additional memory reference will be needed, to access the page table.
- Since execution speed is generally limited by the rate at which the CPU can get instructions and data out of the memory, having to make two memory references per memory reference reduces performance by half.
- Under these conditions, no one would use paging.



- Their solution is based on the observation that most programs tend to make a large number of references to a small number of pages, and not the other way around.
- Thus only a small fraction of the page table entries are heavily read; the rest are barely used at all.
- The solution that has been devised is to equip computers with a small hardware device for mapping virtual addresses to physical addresses without going through the page table.
- The device, called a **TLB (Translation Lookaside Buffer)** or sometimes an **associative memory**



- It is usually inside the MMU and consists of a small number of entries, eight in this example, but rarely more than 256.
- Each entry contains information about one page, including the virtual page number, a bit that is set when the page is modified, the protection code (read/write/execute permissions), and the physical page frame in which the page is located.
- These fields have a one-to-one correspondence with the fields in the page table, except for the virtual page number, which is not needed in the page table.
- Another bit indicates whether the entry is valid (i.e., in use) or not.



- An example that might generate the TLB of Fig. 3-12 is a process in a loop that spans virtual pages 19, 20, and 21, so that these TLB entries have protection codes for reading and executing.
- The main data currently being used (say, an array being processed) are on pages 129 and 130.
- Page 140 contains the indices used in the array calculations.
- Finally, the stack is on pages 860 and 861.



Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Figure 3-12. A TLB to speed up paging.



- Let us now see how the TLB functions. When a virtual address is presented to the MMU for translation, the hardware first checks to see if its virtual page number is present in the TLB by comparing it to all the entries simultaneously (i.e., in parallel).
- Doing so requires special hardware, which all MMUs with TLBs have.
- If a valid match is found and the access does not violate the protection bits, the page frame is taken directly from the TLB, without going to the page table.
- If the virtual page number is present in the TLB but the instruction is trying to write on a read-only page, a protection fault is generated.



- The interesting case is what happens when the virtual page number is not in the TLB.
- The MMU detects the miss and does an ordinary page table lookup.
- It then evicts one of the entries from the TLB and replaces it with the page table entry just looked up.
- Thus if that page is used again soon, the second time it will result in a TLB hit rather than a miss



Software TLB Management

- Up until now, we have assumed that every machine with paged virtual memory has page tables recognized by the hardware, plus a TLB.
- In this design, TLB management and handling TLB faults are done entirely by the MMU hardware.
- Traps to the operating system occur only when a page is not in memory



- In the past, this assumption was true. However, many RISC machines, including the SPARC, MIPS, and (the now dead) HP PA, do nearly all of this page management in software.
- On these machines, the TLB entries are explicitly loaded by the operating system.
- When a TLB miss occurs, instead of the MMU going to the page tables to find and fetch the needed page reference, it just generates a TLB fault and tosses the problem into the lap of the operating system.
- The system must find the page, remove an entry from the TLB, enter the new one, and restart the instruction that faulted.



- And, of course, all of this must be done in a handful of instructions because TLB misses occur much more frequently than page faults.
- Various strategies were developed long ago to improve performance on machines that do TLB management in software.
- One approach attacks both reducing TLB misses and reducing the cost of a TLB miss when it does occur
- To reduce TLB misses, sometimes the operating system can use its intuition to figure out which pages are likely to be used next and to preload entries for them in the TLB.



- For example, when a client process sends a message to a server process on the same machine, it is very likely that the server will have to run soon.
- Knowing this, while processing the trap to do the send, the system can also check to see where the server's code, data, and stack pages are and map them in before they get a chance to cause TLB faults.



- A **soft miss** occurs when the page referenced is not in the TLB, but is in memory.
- All that is needed here is for the TLB to be updated. No disk I/O is needed.
- Typically a soft miss takes 10–20 machine instructions to handle and can be completed in a couple of nanoseconds.
- In contrast, a **hard miss** occurs when the page itself is not in memory (and of course, also not in the TLB).



- disk access is required to bring in the page, which can take several milliseconds, depending on the disk being used.
- A hard miss is easily a million times slower than a soft miss.
- Looking up the mapping in the page table hierarchy is known as a **page table walk**.



- A miss is not just soft or hard. Some misses are slightly softer (or slightly harder) than other misses.
- For instance, suppose the page walk does not find the page in the process' page table and the program thus incurs a page fault.
- There are three possibilities.
- First, the page may actually be in memory, but not in this process' page table.
- For instance, the page may have been brought in from disk by another process. In that case, we do not need to access the disk again, but merely map the page appropriately in the page tables
- This is a pretty soft miss that is known as a **minor page fault**



- Second, a **major page fault** occurs if the page needs to be brought in from disk.
- Third, it is possible that the program simply accessed an invalid address and no mapping needs to be added in the TLB at all.
- In that case, the operating system typically kills the program with a **segmentation fault**.

Page Tables for Large Memories



- TLBs can be used to speed up virtual-to-physical address translation over the original page-table-in-memory scheme.
- But that is not the only problem we have to tackle.
- Another problem is how to deal with very large virtual address spaces.
- Below we will discuss two ways of dealing with them.



Multilevel Page Tables

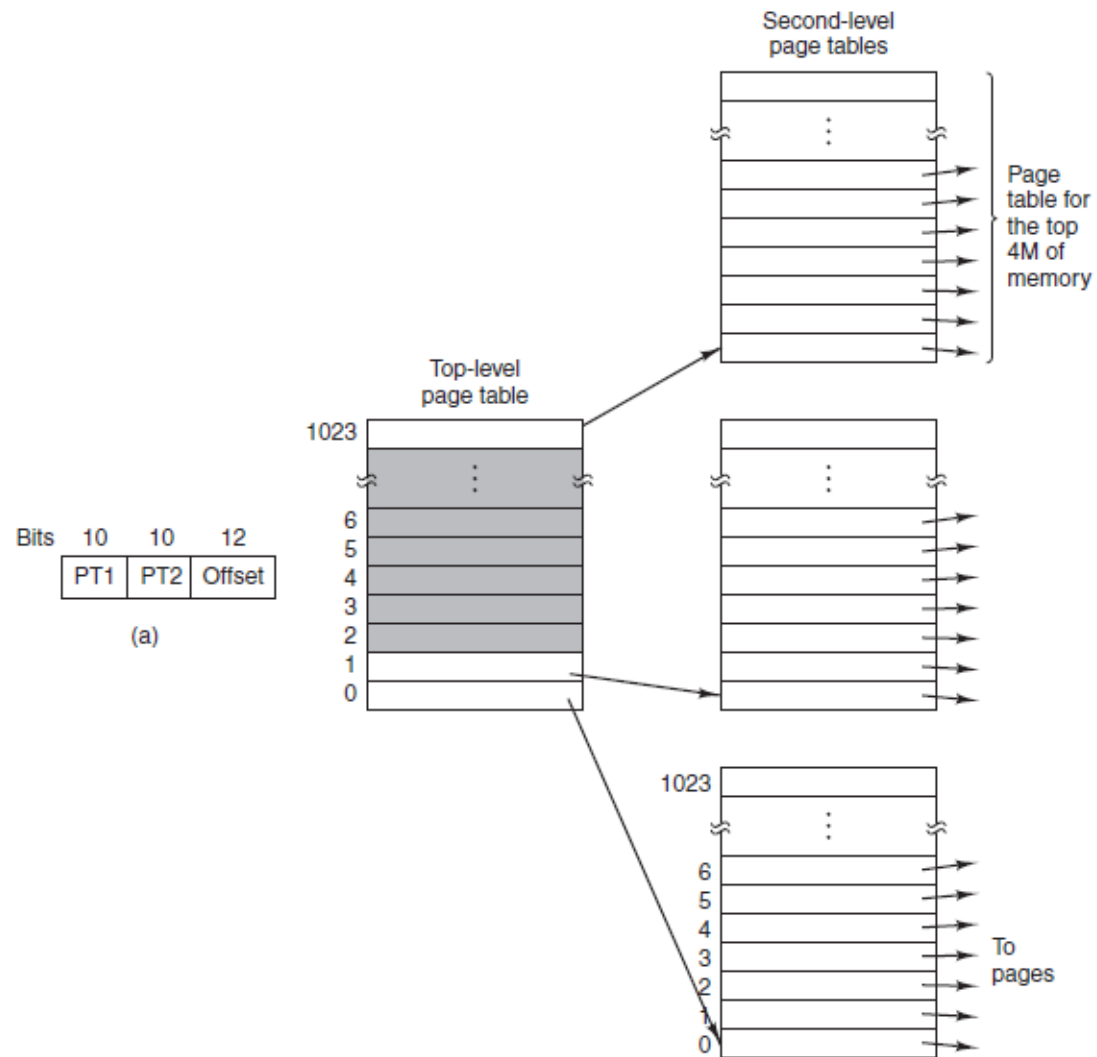
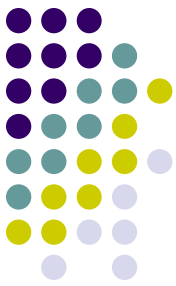
- In Fig. 3-13(a) we have a 32-bit virtual address (4 GB) that is partitioned into a 10-bit *PT1* field, a 10-bit *PT2* field, and a 12-bit *Offset* field.
- Since offsets are 12 bits, pages are 4 KB size, and there are a total of 2^{20} of them (1048576 pages).
- The secret to the multilevel page table method is to avoid keeping all the page tables in memory all the time.



- In particular, those that are not needed should not be kept around.
- In Fig. 3-13(b) we see how the two-level page table works.
- On the left we see the top-level page table, with 1024 entries, corresponding to the 10-bit *PT1* field.
- When a virtual address is presented to the MMU, it first extracts the *PT1* field and uses this value as an index into the top-level page table.
- Each of these 1024 entries in the top-level page table represents 4M because the entire 4-gigabyte (i.e., 32-bit) virtual address space has been chopped into chunks of 4096 bytes.



- The entry located by indexing into the top-level page table yields the address or the page frame number of a second-level page table.
- The *PT2* field is now used as an index into the selected second-level page table to find the page frame number for the page itself.





- The interesting thing to note about Fig. 3-13 is that although the address space contains over a million pages, only four page tables are needed: the top-level table, and the second-level tables for 0 to 4M (for the program text), 4M to 8M (for the data), and the top 4M (for the stack).
- The *Present/absent* bits in the remaining 1021 entries of the top-level page table are set to 0, forcing a page fault if they are ever accessed.
- The two-level page table system of Fig. 3-13 can be expanded to three, four, or more levels.
- Additional levels give more flexibility



Inverted Page Tables

- An alternative to ever-increasing levels in a paging hierarchy is known as **inverted page tables**.
- In this design, there is one entry per page frame in real memory, rather than one entry per page of virtual address space.
- For example, with 64-bit virtual addresses, a 4-KB page size, and 4 GB of RAM, an inverted page table requires only 1,048,576 entries.
- The entry keeps track of which (process, virtual page) is located in the page frame.



- Although inverted page tables save lots of space, at least when the virtual address space is much larger than the physical memory, they have a serious downside:
- virtual-to-physical translation becomes much harder.
- When process n references virtual page p , the hardware can no longer find the physical page by using p as an index into the page table.
- Instead, it must search the entire inverted page table for an entry (n, p) .
- Furthermore, this search must be done on every memory reference, not just on page faults. Searching a 256K table on every memory reference is not the way to make your machine blindingly fast.



- The way out of this dilemma is to make use of the TLB.
- If the TLB can hold all of the heavily used pages, translation can happen just as fast as with regular page tables.
- On a TLB miss, however, the inverted page table has to be searched in software.
- One feasible way to accomplish this search is to have a hash table hashed
- on the virtual address. All the virtual pages currently in memory that have the same hash value are chained together.
- Once the page frame number has been found, the new (virtual, physical) pair is entered into the TLB.

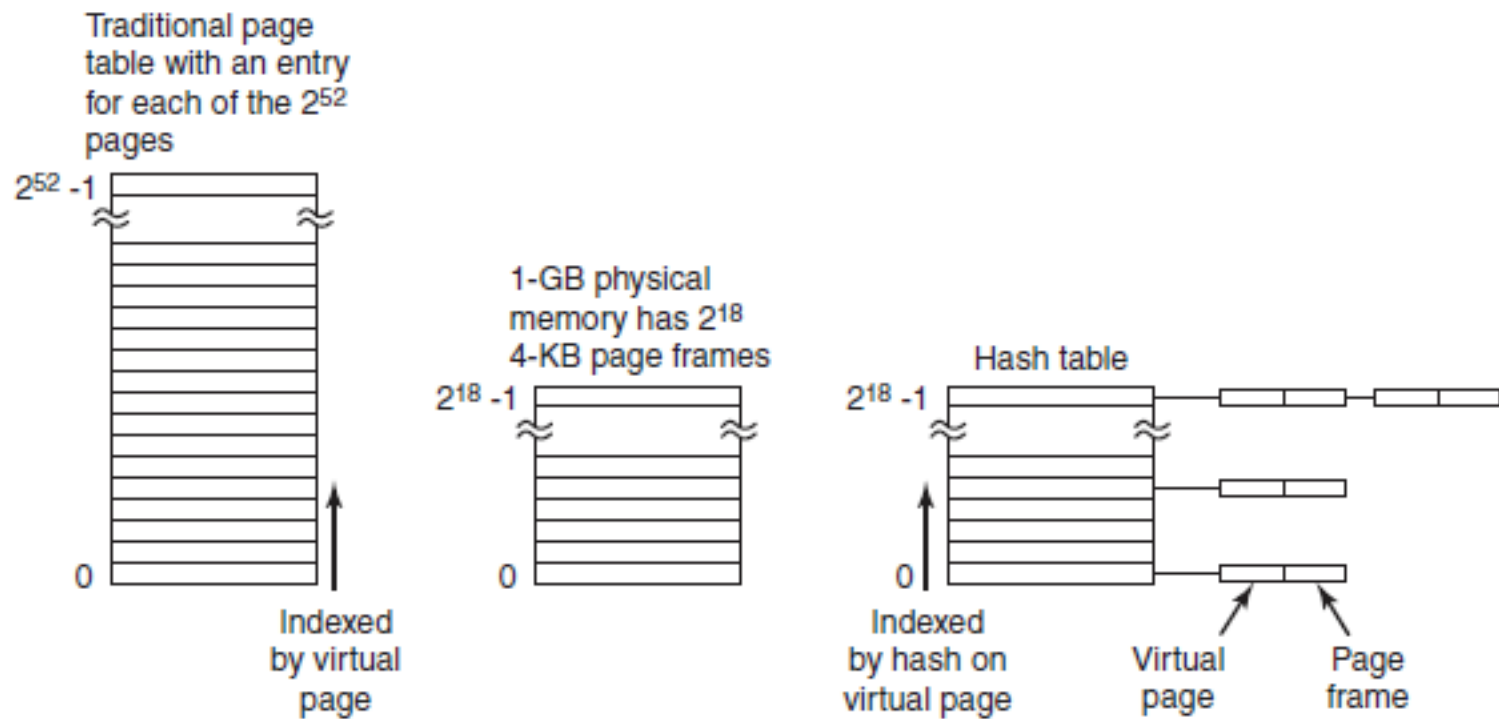
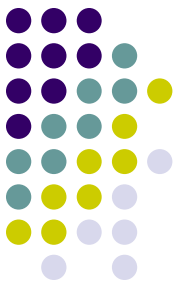
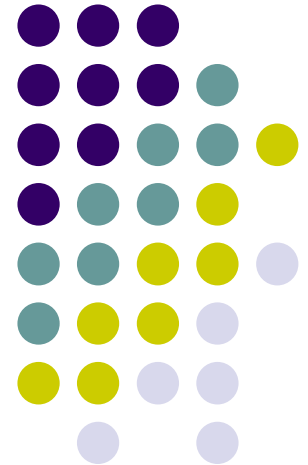


Figure 3-14. Comparison of a traditional page table with an inverted page table.

DESIGN ISSUES FOR PAGING SYSTEMS



Local versus Global Allocation Policies



- A major issue is how memory should be
- allocated among the competing runnable processes.
- In this figure, three processes, A , B , and C , make up the set of runnable processes.
- Suppose A gets a page fault. Should the page replacement algorithm try to find the least recently used page considering only the six pages currently allocated to A , or should it consider all the pages in memory?



- If it looks only at A 's pages, the page with the lowest age value is $A5$
- On the other hand, if the page with the lowest age value is removed without regard to whose page it is, page $B3$ will be chosen.
- First algorithm is said to be a **local** page replacement.
- Second is said to be **Global** page replacement algorithm.



Age	
A0	10
A1	7
A2	5
A3	4
A4	6
A5	3
B0	9
B1	4
B2	6
B3	2
B4	5
B5	6
B6	12
C1	3
C2	5
C3	6

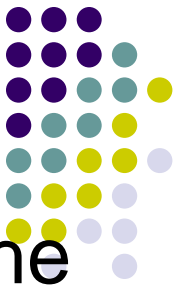
(a)

A0
A1
A2
A3
A4
A6
B0
B1
B2
B3
B4
B5
B6
C1
C2
C3

(b)

A0
A1
A2
A3
A4
A5
B0
B1
B2
A6
B4
B5
B6
C1
C2
C3

(c)



- Local algorithms effectively correspond to allocating every process a fixed fraction of the memory.
- Global algorithms dynamically allocate page frames among the runnable processes.
- Thus the number of page frames assigned to each process varies in time.
- In general, global algorithms work better, especially when the working set size can vary a lot over the lifetime of a process.
- If a local algorithm is used and the working set grows, thrashing will result, even if there are a sufficient number of free page frames.



- If the working set shrinks, local algorithms waste memory.
- If a global algorithm is used, the system must continually decide how many page frames to assign to each process.
- One way is to monitor the working set size as indicated by the aging bits, but this approach does not necessarily prevent thrashing.
- The working set may change size in milliseconds, whereas the aging bits are a very crude measure spread over a number of clock ticks.



- Another approach is to have an algorithm for allocating page frames to processes.
- One way is to periodically determine the number of running processes and allocate each process an equal share.
- Thus with 12,416 available (i.e., non-operating system) page frames and 10 processes, each process gets 1241 frames.
- The remaining six go into a pool to be used when page faults occur.



- Although this method may seem fair, it makes little sense to give equal shares of the memory to a 10-KB process and a 300-KB process.
- Instead, pages can be allocated in proportion to each process' total size, with a 300-KB process getting 30 times the allotment of a 10-KB process.
- It is probably wise to give each process some minimum number, so that it can run no matter how small it is.



- If a global algorithm is used, it may be possible to start each process up with some number of pages proportional to the process' size, but the allocation has to be updated dynamically as the processes run.
- One way to manage the allocation is to use the **PFF (Page Fault Frequency)** algorithm.
- It tells when to increase or decrease a process' page allocation but says nothing about which page to replace on a fault. It just controls the size of the allocation set.



- For a large class of page replacement algorithms, including LRU, it is known that the fault rate decreases as more frames are assigned.
- But for some algorithms like FIFO increase in page frames may results in increase in page faults.
- This situation is called **Belady's anamoly**.



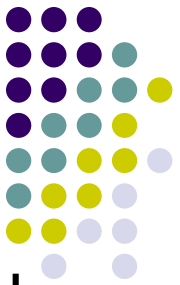
Load Control

- Even with the best page replacement algorithm and optimal global allocation of page frames to processes, it can happen that the system thrashes.
- In fact, whenever the combined working sets of all processes exceed the capacity of memory, thrashing can be expected.
- In this case, there is no way to give more memory to those processes needing it without hurting some other processes.



- The only real solution is to temporarily get rid of some processes.
- A good way to reduce the number of processes competing for memory is to swap some of them to the disk and free up all the pages they are holding
- However, another factor to consider is the degree of multiprogramming.
- When the number of processes in main memory is too low, the CPU may be idle for substantial periods of time.

Page Size



- The page size is a parameter that can be chosen by the operating system.
- Determining the best page size requires balancing several competing factors.
- As a result, there is no overall optimum.
- To start with, two factors argue for a small page size.
- On the average, half of the final page will be empty.
- The extra space in that page is wasted.
- This wastage is called **internal fragmentation**
- With n segments in memory and a page size of p bytes, $np/2$ bytes will be wasted on internal fragmentation.
- This reasoning argues for a small page size.



- Another argument for a small page size becomes apparent if we think about a program consisting of eight sequential phases of 4 KB each.
- With a 32-KB page size, the program must be allocated 32 KB all the time. With a 16-KB page size, it needs only 16 KB. With a page size of 4 KB or smaller, it requires only 4 KB at any instant.
- In general, a large page size will cause more wasted space to be in memory than a small page size.

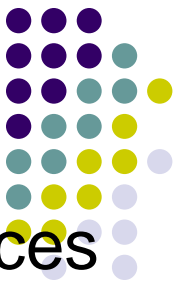


- On the other hand, small pages mean that programs will need many pages, and thus a large page table.
- Also, small pages use up much valuable space in the **TLB**.
- As TLB entries are scarce, and critical for performance, it pays to use large pages wherever possible.
- To balance all these trade-offs, operating systems sometimes use different page sizes for different parts of the system.
- For instance, large pages for the kernel and smaller ones for user processes.

Separate Instruction and Data Spaces



- Most computers have a single address space that holds both programs and data.
- If this address space is large enough, everything works fine.
- However, if it's too small, it forces programmers to stand on their heads to fit everything into the address space.
- One solution, pioneered on the (16-bit) PDP-11, is to have separate address spaces for instructions (program text) and data, called **I-space** and **D-space**, respectively.
- Each address space runs from 0 to some maximum, typically $2^{16} - 1$ or $2^{32} - 1$.



- The linker must know when separate I and D-spaces are being used, because when they are, the data are relocated to virtual address 0 instead of starting after the program.
- In a computer with this kind of design, both address spaces can be paged, independently from one another.
- Each one has its own page table, with its own mapping of virtual pages to physical page frames. When the hardware wants to fetch an instruction, it knows that it must use I-space and the I-space page table.
- Similarly, data must go through the D-space page table.



Shared Pages

- In a large multiprogramming system, it is common for several users to be running the same program at the same time.
- Even a single user may be running several programs that use the same library.
- It is clearly more efficient to share the pages, to avoid having two copies of the same page in memory at the same time.
- One problem is that not all pages are sharable. In particular, pages that are read-only, such as program text, can be shared, but for data pages sharing is more complicated.



- When two or more processes share some code, a problem occurs with the shared pages.
- Suppose that processes *A* and *B* are both running the editor and sharing its pages.
- If the scheduler decides to remove *A* from memory, evicting all its pages and filling the empty page frames with some other program will cause *B* to generate a large number of page faults to bring them back in again



- Searching all the page tables to see if a page is shared is usually too expensive, so special data structures are needed to keep track of shared pages.
- Sharing data is trickier than sharing code, but it is not impossible.
- In particular, in UNIX, after a fork system call, the parent and child are required to share both program text and data



- In a paged system, what is often done is to give each of these processes its own page table and have both of them point to the same set of pages.
- Thus no copying of pages is done at fork time.
- However, all the data pages are mapped into both processes as READ ONLY.
- As long as both processes just read their data, without modifying it, this situation can continue.
- As soon as either process updates a memory word, the violation of the read-only protection causes a trap to the operating system.



- A copy is then made of the offending page so that each process now has its own private copy.
- This approach, called **copy on write**, improves performance by reducing copying.



Shared Libraries

- Sharing can be done at other granularities than individual pages.
- In modern systems, there are many large libraries used by many processes, for example, multiple I/O and graphics libraries.
- Statically binding all these libraries to every executable program on the disk would make them even more bloated than they already are.
- Instead, a common technique is to use **shared libraries** (which are called **DLLs** or **Dynamic Link Libraries** on Windows).

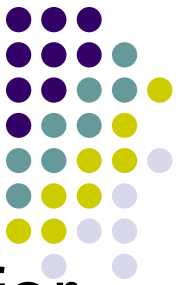


- When a program is linked with shared libraries (which are slightly different than static ones), instead of including the actual function called, the linker includes a small stub routine that binds to the called function at run time.
- Depending on the system and the configuration details, shared libraries are loaded either when the program is loaded or when functions in them are called for the first time.
- Of course, if another program has already loaded the shared library, there is no need to load it again—that is the whole point of it.



Mapped Files

- Shared libraries are really a special case of a more general facility called **memory-mapped files**.
- The idea here is that a process can issue a system call to map a file onto a portion of its virtual address space.
- In most implementations, no pages are brought in at the time of the mapping, but as pages are touched, they are demand paged in one page at a time, using the disk file as the backing store.
- When the process exits, or explicitly unmaps the file, all the modified pages are written back to the file on disk



- Mapped files provide an alternative model for I/O. Instead, of doing reads and writes, the file can be accessed as a big character array in memory.
- If two or more processes map onto the same file at the same time, they can communicate over shared memory.
- Writes done by one process to the shared memory are immediately visible when the other one reads from the part of its virtual address space mapped onto the file.

Cleaning Policy



- Paging works best when there is an abundant supply of free page frames that can be claimed as page faults occur. If every page frame is full, and furthermore modified, before a new page can be brought in, an old page must first be written to disk.
- To ensure a plentiful supply of free page frames, paging systems generally have a background process, called the **paging daemon**, that sleeps most of the time but is awakened periodically to inspect the state of memory.
- If too few page frames are free, it begins selecting pages to evict using some page replacement algorithm.
- If these pages have been modified since being loaded, they are written to disk.



- Keeping a supply of page frames around yields better performance than using all of memory and then trying to find a frame at the moment it is needed.
- At the very least, the paging daemon ensures that all the free frames are clean, so they need not be written to disk in a big hurry when they are required



SEGMENTATION

- The virtual memory discussed so far is one-dimensional because the virtual addresses go from 0 to some maximum address, one address after another.
- For many problems, having two or more separate virtual address spaces may be much better than having only one.
- For example, a compiler has many tables that are built up as compilation proceeds, possibly including



- 1. The source text being saved for the printed listing (on batch systems).
- 2. The symbol table, containing the names and attributes of variables.
- 3. The table containing all the integer and floating-point constants used.
- 4. The parse tree, containing the syntactic analysis of the program.
- 5. The stack used for procedure calls within the compiler.



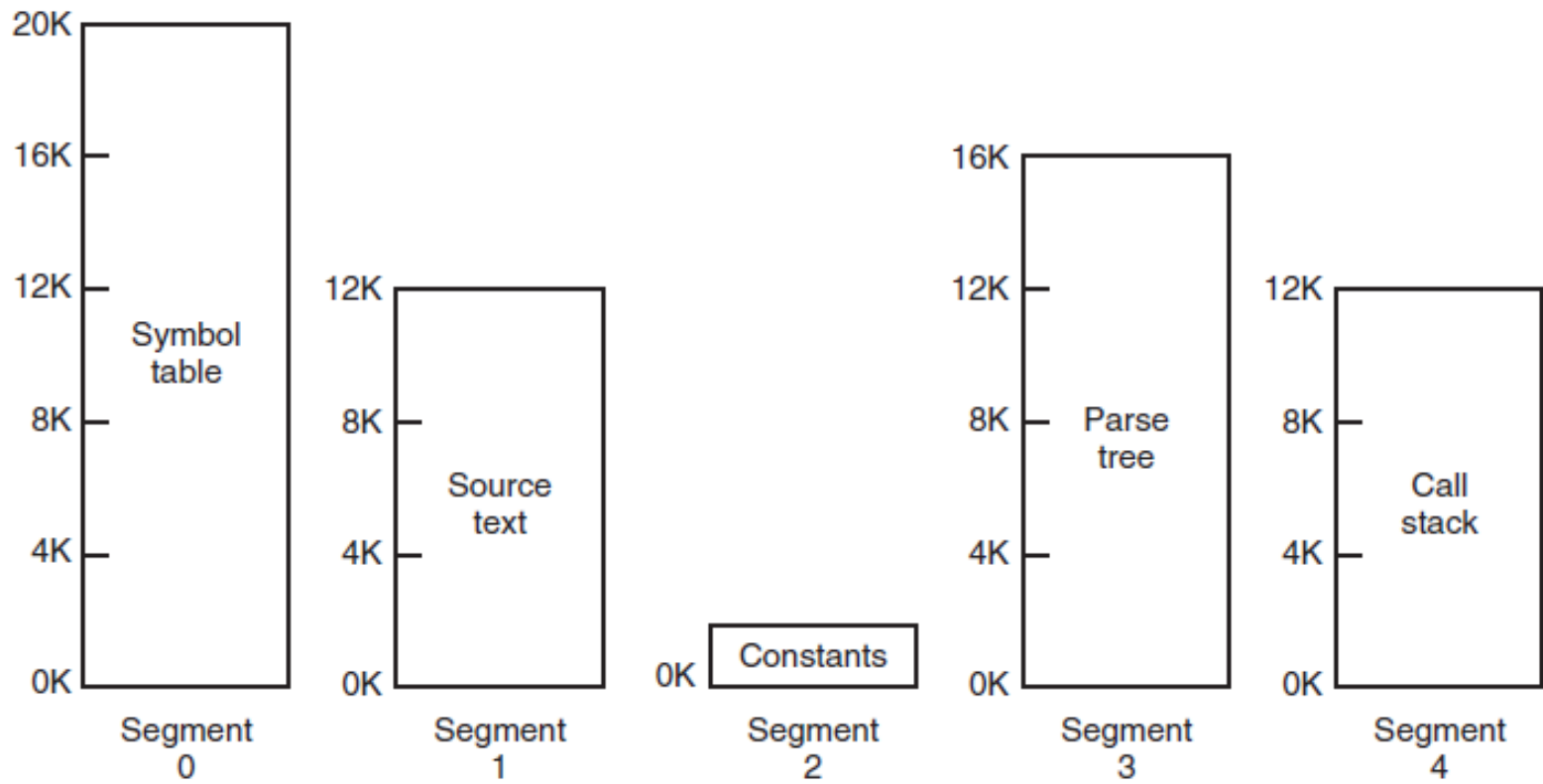
- Each of the first four tables grows continuously as compilation proceeds.
- The last one grows and shrinks in unpredictable ways during compilation.
- In a one-dimensional memory, these five tables would have to be allocated contiguous chunks of virtual address space.
- Consider what happens if a program has a much larger than usual number of variables but a normal amount of everything else.
- The chunk of address space allocated for the symbol table may fill up, but there may be lots of room in the other tables.



- What is needed is a way of freeing the programmer from having to manage the expanding and contracting tables.
- A straightforward and quite general solution is to provide the machine with many completely independent address spaces, which are called **segments**.
- Each segment consists of a linear sequence of addresses, starting at 0 and going up to some maximum value.
- The length of each segment may be anything from 0 to the maximum address allowed.
- Different segments may, and usually do, have different lengths.

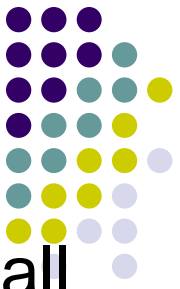


- Moreover, segment lengths may change during execution.
- The length of a stack segment may be increased whenever something is pushed onto the stack and decreased whenever something is popped off the stack.
- Because each segment constitutes a separate address space, different segments can grow or shrink independently without affecting each other.





- We emphasize here that a segment is a logical entity, which the programmer is aware of and uses as a logical entity.
- A segment might contain a procedure, or an array, or a stack, or a collection of scalar variables, but usually it does not contain a mixture of different types.
- A segmented memory has other advantages besides simplifying the handling of data structures that are growing or shrinking.



- After all the procedures that constitute a program have been compiled and linked up, a procedure call to the procedure in segment n will use the two-part address $(n, 0)$ to address word 0 (the entry point).
- If the procedure in segment n is subsequently modified and recompiled, no other procedures need be changed (because no starting addresses have been modified), even if the new version is larger than the old one. With a one-dimensional memory, the procedures are packed tightly right up next to each other, with no address space between them.
- Consequently, changing one procedure's size can affect the starting address of all the other (unrelated) procedures in the segment. .



- This, in turn, requires modifying all procedures that call any of the moved procedures, in order to incorporate their new starting addresses.
- If a program contains hundreds of procedures, this process can be costly