Memory Management

Week 07 – Lecture

Address Space

Team

- Instructor
  - Giancarlo Succi
- Teaching Assistants
  - Nikita Lozhnikov (also Tutorial Instructor)
  - Manuel Rodriguez
  - Shokhista Ergasheva

Sources

- These slides have been adapted from the original slides of the adopted book:
  - Tanenbaum & Bos, Modern Operating Systems: 4th edition, 2013
    Prentice-Hall, Inc.

  and customised for the needs of this course.
- Additional input for the slides are detailed later

Memory (1)

- Paraphrase of Parkinson's Law, "*Programs expand to fill the memory available to hold them.*"
- Average home computer nowadays has 10,000 times more memory than the IBM 7094, the largest computer in the world in the early 1960s

Memory (2)

- We all want private, infinitely large, infinitely fast and inexpensive memory that does not lose its contents when the electric power is switched off
- The second choice is the concept of a **memory hierarchy**:
  - a few megabytes of very fast, expensive, volatile cache memory
  - a few gigabytes of medium-speed, medium-priced, volatile main memory
  - a few terabytes of slow, cheap, nonvolatile magnetic or solid-state disk storage
- It is the job of the operating system to abstract this hierarchy into a useful model and then manage the abstraction

Memory (3)

- The part of the operating system that manages 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
  - deallocate it when they are done

No Memory Abstraction (1)

- Early mainframe computers, early minicomputers and early personal computers had no memory abstraction
- All the physical memory was available to every program and it was not possible to have two running programs in memory at the same time
- The first program could easily erase data belonging to the second one

No Memory Abstraction (2)

- Even without memory abstraction several variations are possible:
  - The OS may be at the bottom of memory in RAM (Fig. 3-1a). It was used on mainframes and minicomputers
  - The OS may be at the top of memory in ROM (Fig. 3-1b). Is used on some handheld devices and in embedded systems
  - The device drivers may be at the top of memory in ROM and the rest of the system in RAM down below (Fig. 3-1c). It was used by early personal computers with BIOS in ROM and MS-DOS running in RAM, for example

No Memory Abstraction (3)

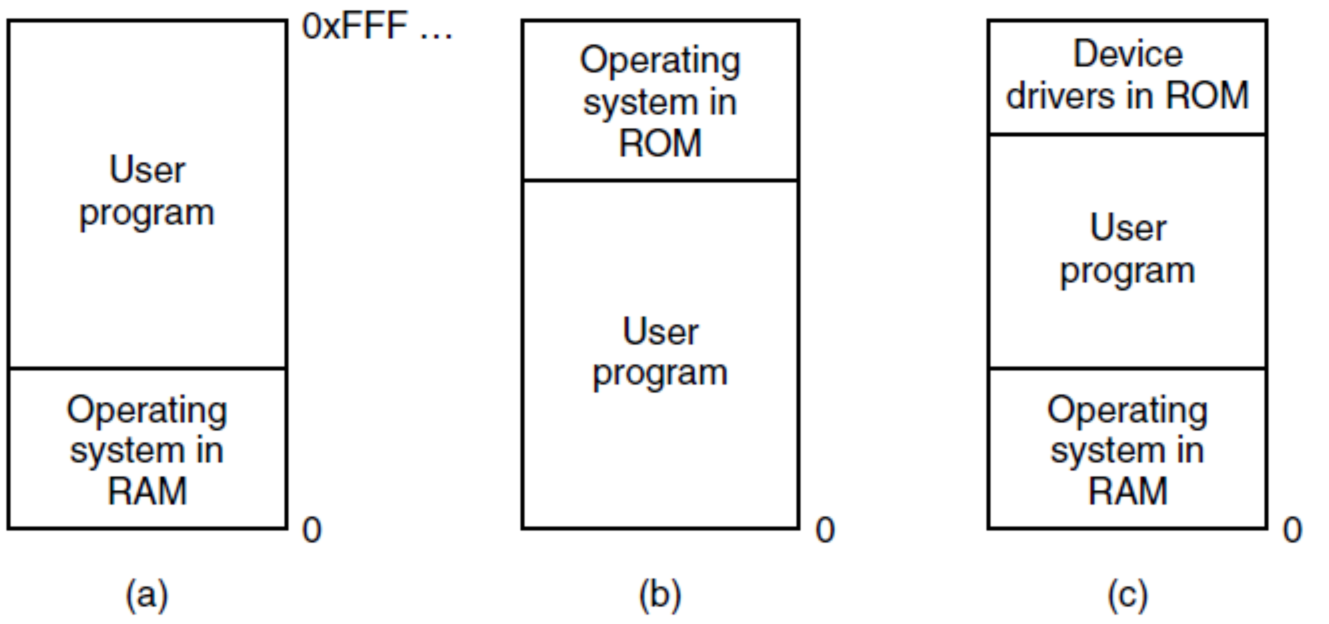


Figure 3-1. Three simple ways of organizing memory with an operating system and one user process. Other possibilities also exist

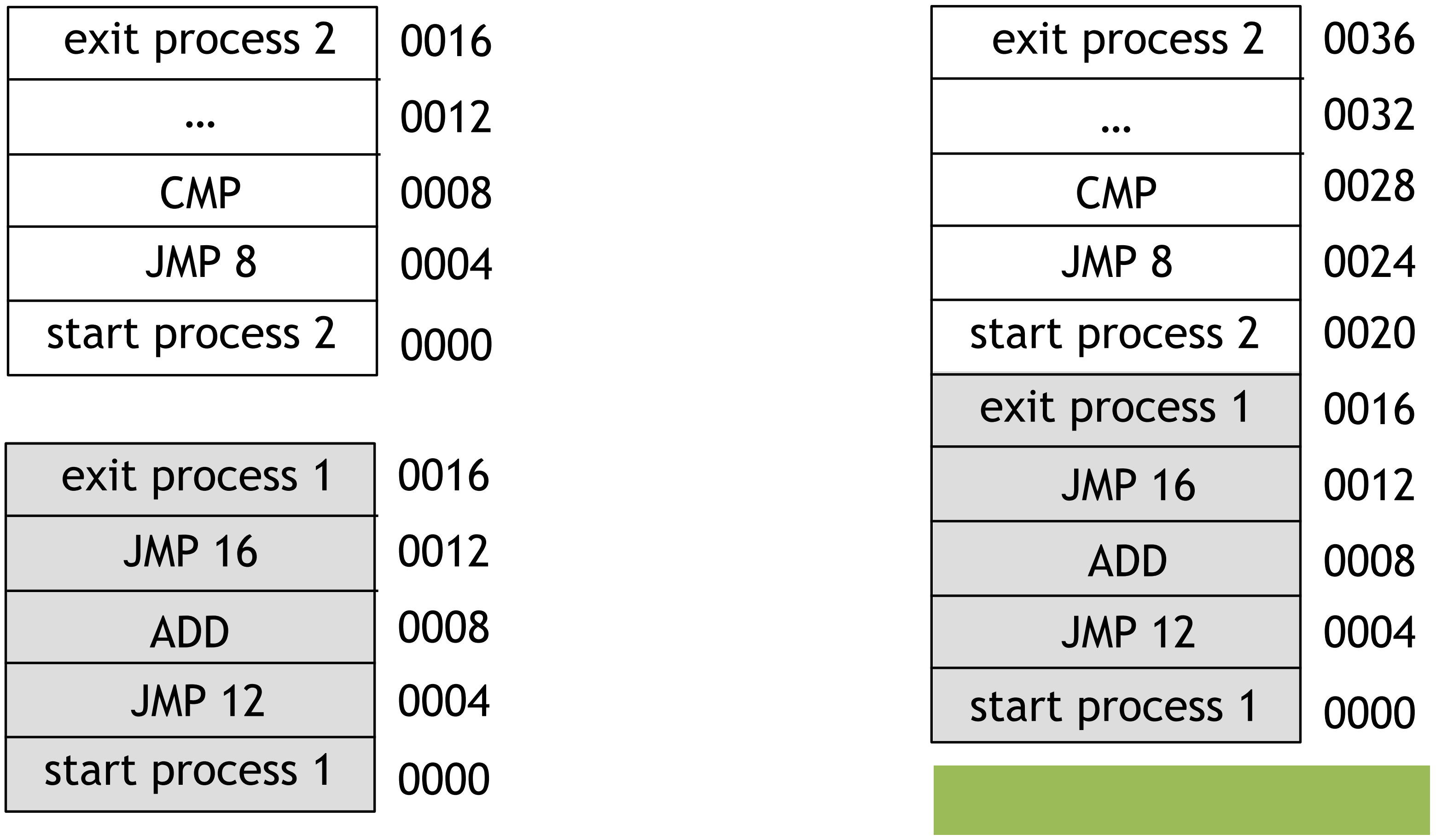
Running Multiple Programs Without a Memory Abstraction (1)

- It is still possible to run multiple programs even with no memory abstraction
- The OS has to save the entire contents of memory to a disk file, then bring in and run the next program (swapping)
- As long as there is only one program at a time in memory, there are no conflicts

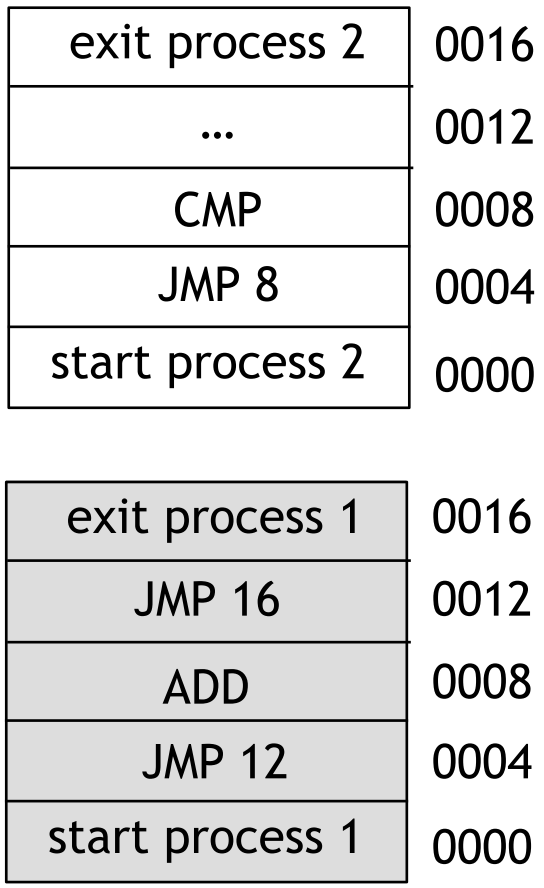
Running Multiple Programs Without a Memory Abstraction (2)

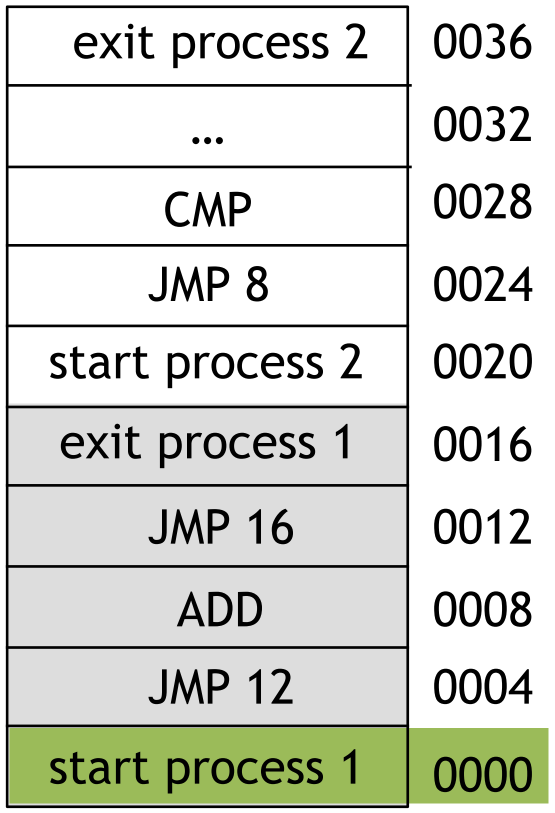
- With the addition of some special hardware, it is possible to run multiple programs concurrently, even without swapping like it was done in early models of the IBM 360:
  - memory was divided into 2-KB blocks and each is assigned a 4-bit protection key held in special registers inside the CPU
  - the PSW (Program Status Word) also contained a 4-bit key
  - the hardware trapped any attempt by a running process to access memory with a protection code different from the PSW key
  - Since only the OS could change the protection keys, user processes were prevented from interfering with one another and with the OS itself

Running Multiple Programs Without a Memory Abstraction (3)

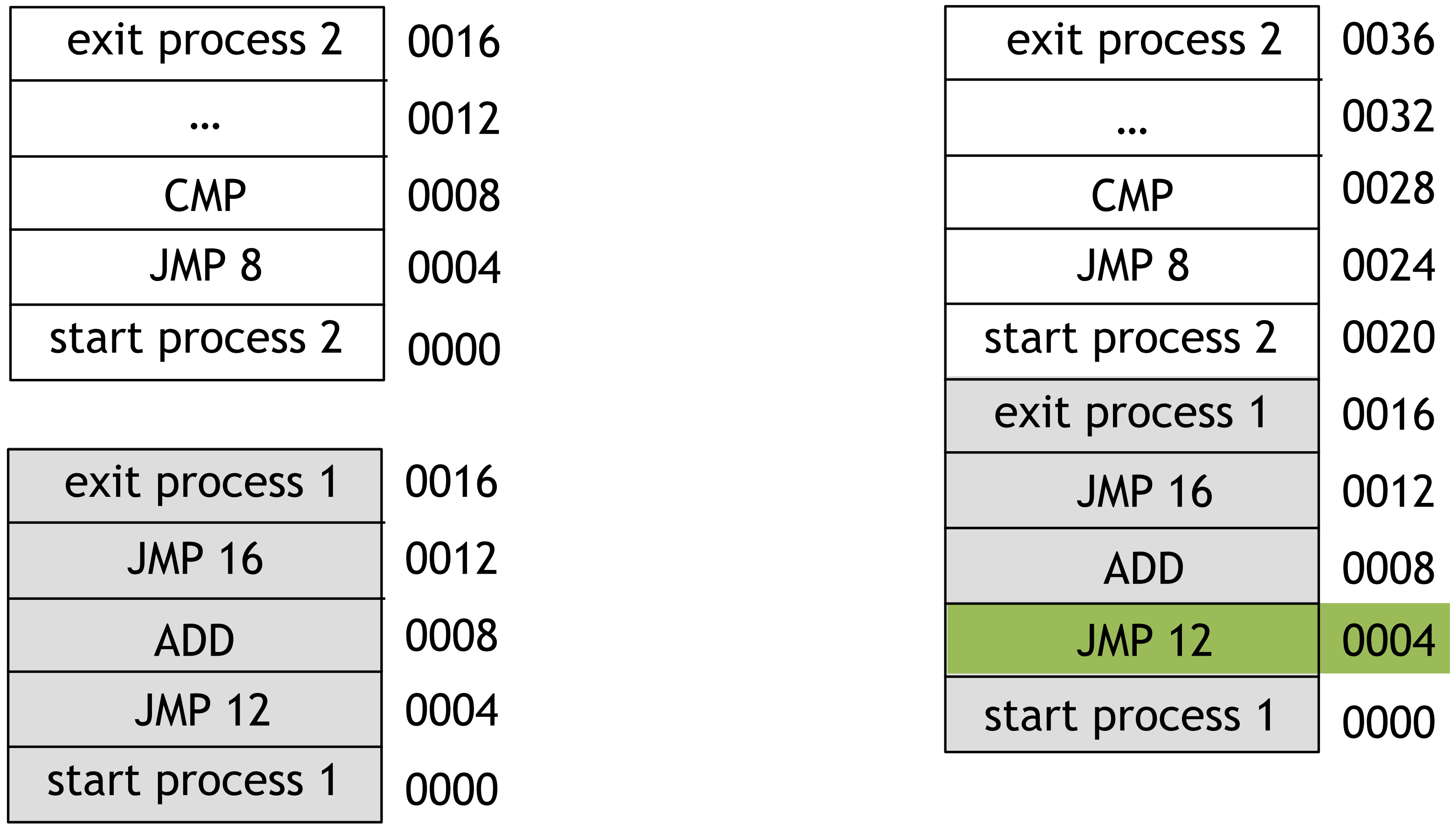


Running Multiple Programs Without a Memory Abstraction (3)





Running Multiple Programs Without a Memory Abstraction (3)



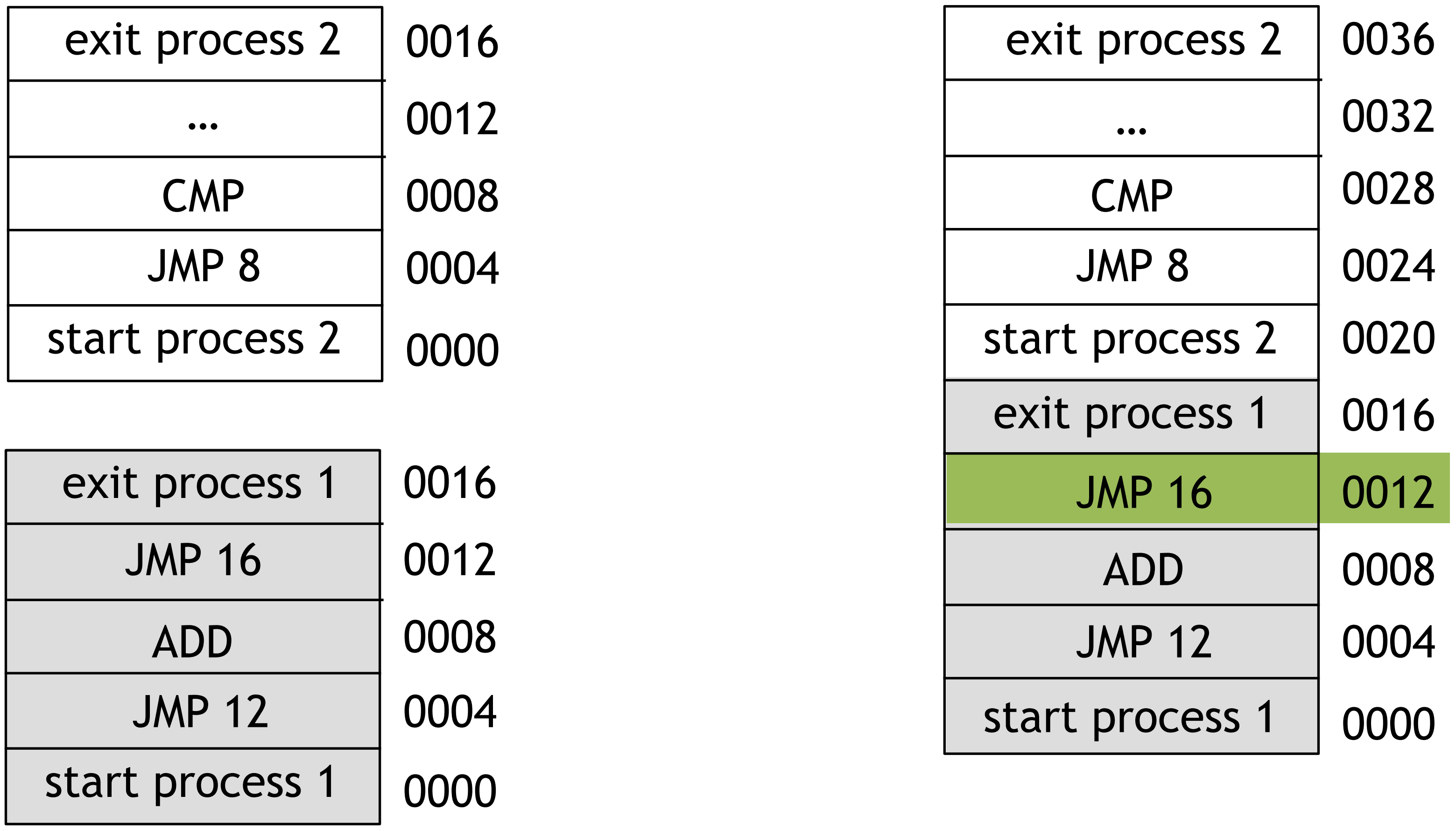
Running Multiple Programs Without a Memory Abstraction (3)

exit process 2	0016
…	0012
CMP	0008
JMP 8	0004
start process 2	0000

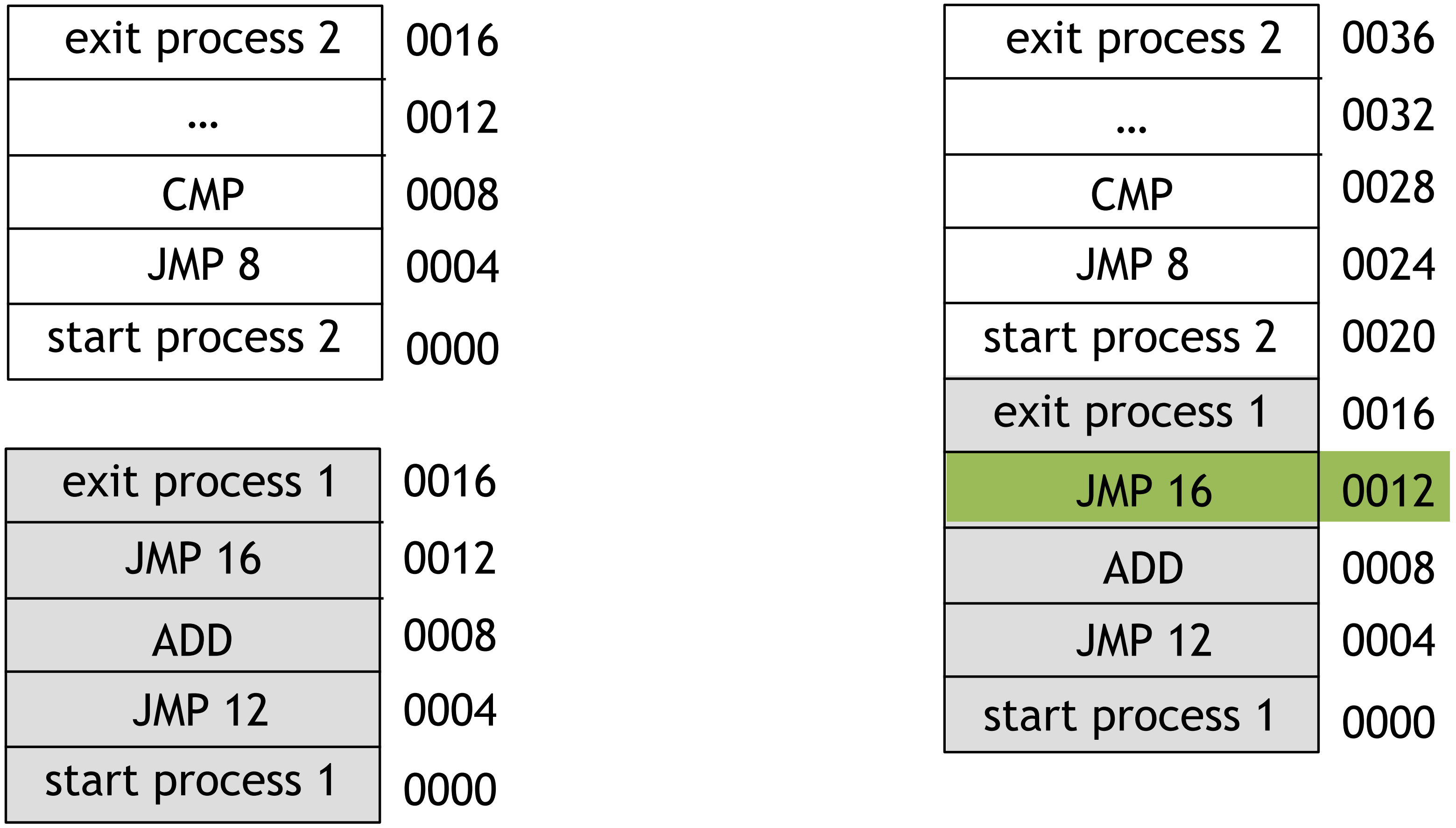
exit process 1	0016
JMP 16	0012
ADD	0008
JMP 12	0004
start process 1	0000

exit process 2	0036
…	0032
CMP	0028
JMP 8	0024
start process 2	0020
exit process 1	0016
JMP 16	0012
ADD	0008
JMP 12	0004
start process 1	0000

Running Multiple Programs Without a Memory Abstraction (3)



Running Multiple Programs Without a Memory Abstraction (3)



Running Multiple Programs Without a Memory Abstraction (3)

exit process 2	0016
…	0012
CMP	0008
JMP 8	0004
start process 2	0000

exit process 1	0016
JMP 16	0012
ADD	0008
JMP 12	0004
start process 1	0000

exit process 2	0036
…	0032
CMP	0028
JMP 8	0024
start process 2	0020
exit process 1	0016
JMP 16	0012
ADD	0008
JMP 12	0004
start process 1	0000

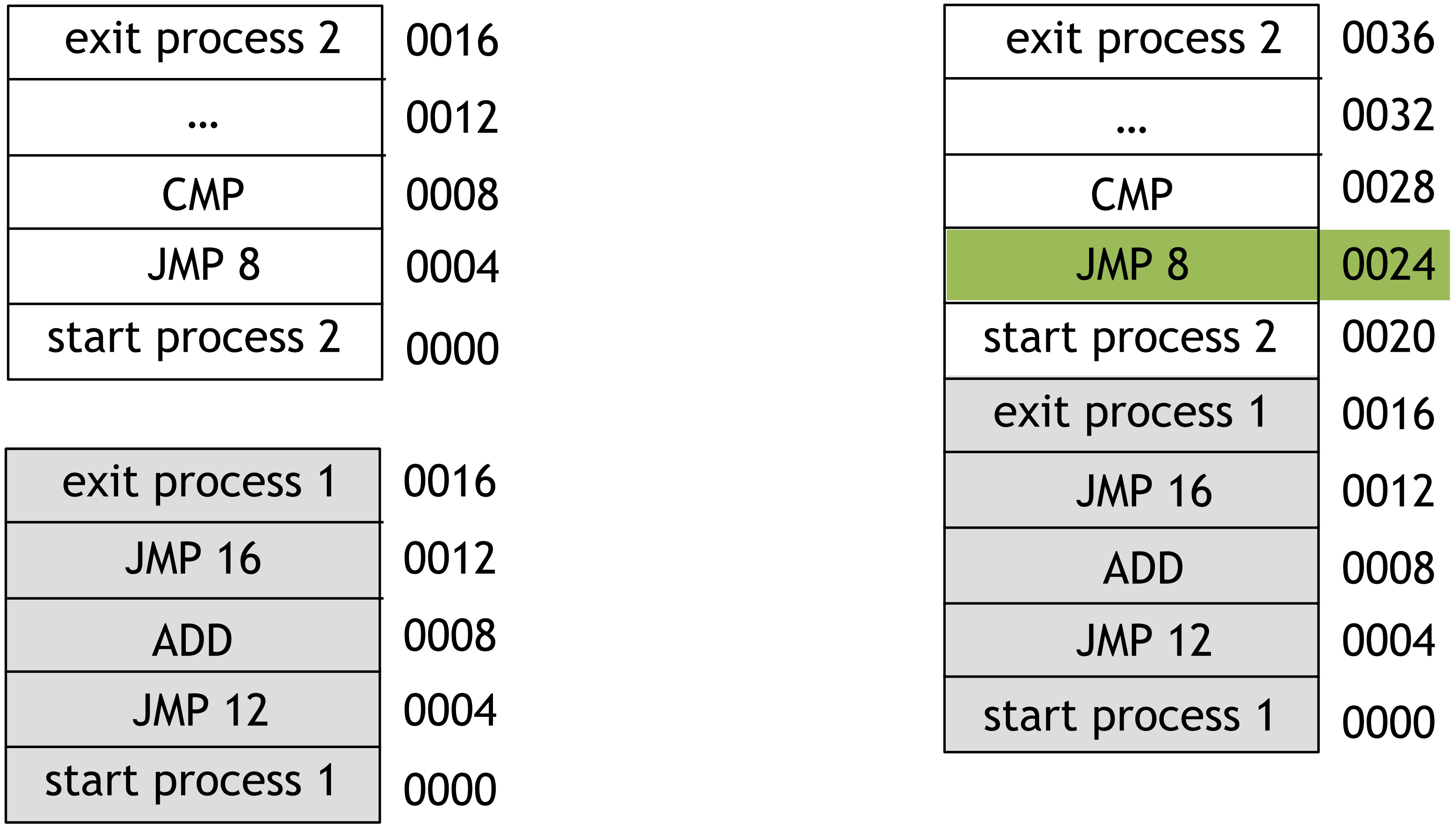
Running Multiple Programs Without a Memory Abstraction (3)

exit process 2	0016
…	0012
CMP	0008
JMP 8	0004
start process 2	0000

exit process 1	0016
JMP 16	0012
ADD	0008
JMP 12	0004
start process 1	0000

exit process 2	0036
…	0032
CMP	0028
JMP 8	0024
start process 2	0020
exit process 1	0016
JMP 16	0012
ADD	0008
JMP 12	0004
start process 1	0000

Running Multiple Programs Without a Memory Abstraction (3)



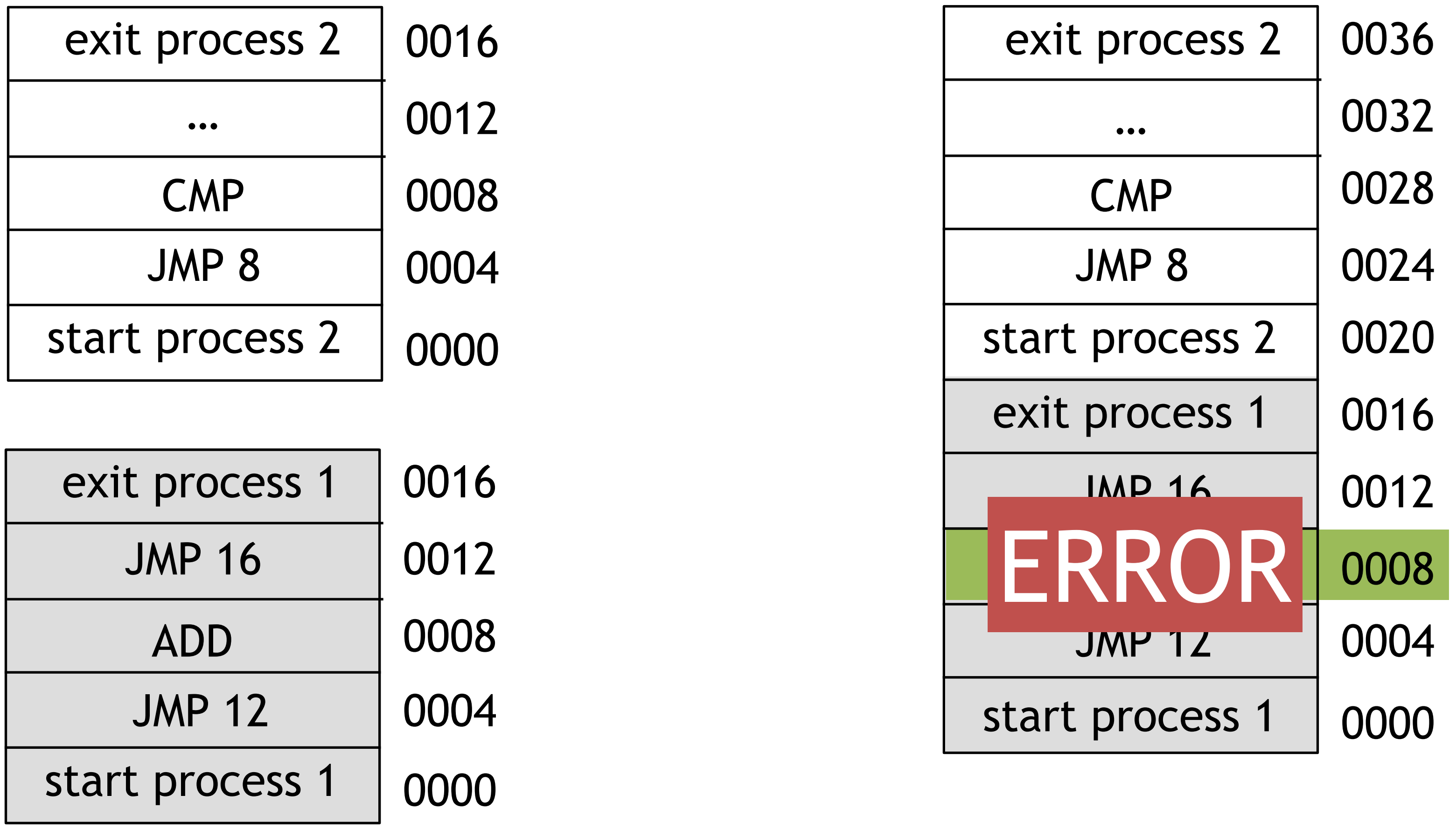
Running Multiple Programs Without a Memory Abstraction (3)

exit process 2	0016
…	0012
CMP	0008
JMP 8	0004
start process 2	0000

exit process 1	0016
JMP 16	0012
ADD	0008
JMP 12	0004
start process 1	0000

exit process 2	0036
…	0032
CMP	0028
JMP 8	0024
start process 2	0020
exit process 1	0016
JMP 16	0012
ADD	0008
JMP 12	0004
start process 1	0000

Running Multiple Programs Without a Memory Abstraction (3)



Running Multiple Programs Without a Memory Abstraction (4)

- The problem of the approach described above:
  - The first program will function normally
  - However, after the OS decides to run the second program, it will execute the first instruction, which is *JMP 8*
  - Instead of jumping to the CMP instruction as expected, it will jump to the ADD instruction of the first program and most likely crash

Running Multiple Programs Without a Memory Abstraction (5)

- The solution is to use **static relocation**:
  - when a program is loaded at address 0020, the constant 0020 is added to every program address during the load process. For example, *JMP 8* becomes *JMP 28*
- Drawbacks of such an approach are:
  - It is not a very general solution and slows down loading
  - It requires extra information to indicate which words contain (relocatable) addresses and which do not. For example, 8 in *JMP 8 (address)* vs. *MOV REGISTER1, 8 (number)*

A Memory Abstraction: Address Spaces (1)

- Exposing physical memory to processes has several major drawbacks:
  - if user programs can address every byte of memory, they can intentionally or by accident crash the OS even if only one application is running
  - it is difficult to have multiple programs running at once if there is only one CPU

A Memory Abstraction: Address Spaces (2)

- Two problems have to be solved: **protection** and **relocation**
- An **address space** is a concept that helps solving both problems. It is the set of addresses that a process can use to address memory
- The address space creates a kind of abstract memory for programs to live in
- Each process has its own address space, independent of those belonging to other processes

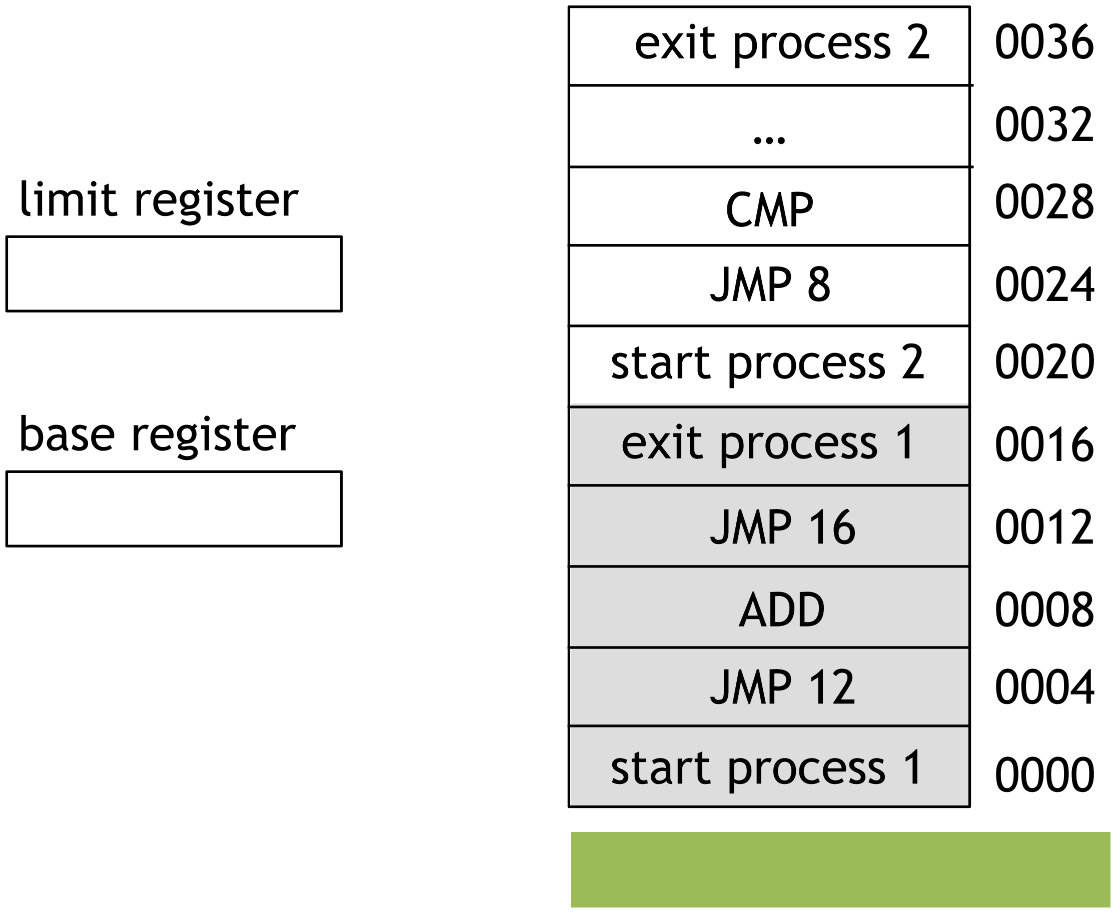
Base and Limit Registers (1)

- **Dynamic relocation** – mapping each process' address space onto a different part of physical memory
- **Base** and **limit** registers – two special hardware registers. Base register contains the physical address of the beginning of the program. Limit register contains its length
- When these registers are used, programs are loaded into **consecutive memory locations** wherever there is room and without relocation during loading

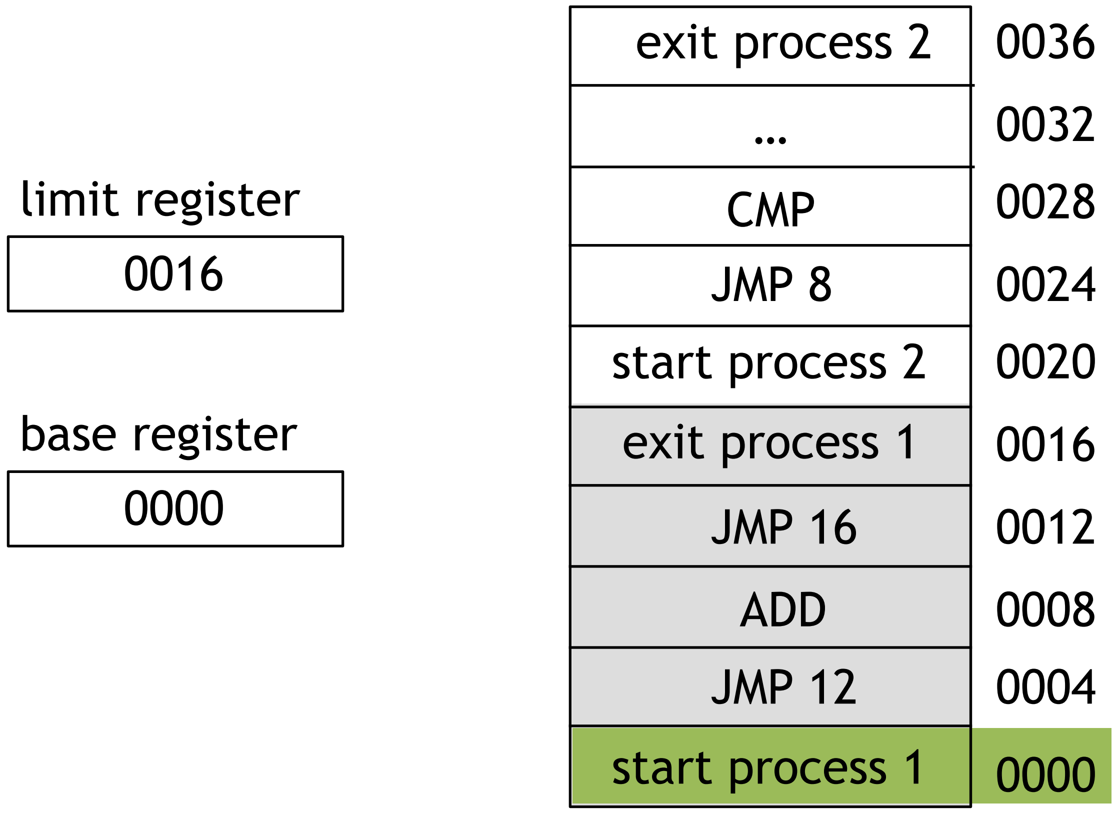
Base and Limit Registers (2)

- Every time a process references memory, the CPU hardware automatically adds the base value to the address generated by the process
- 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
- Disadvantage: the need to perform an addition and a comparison on every memory reference (additions are slow unless special addition circuits are used)

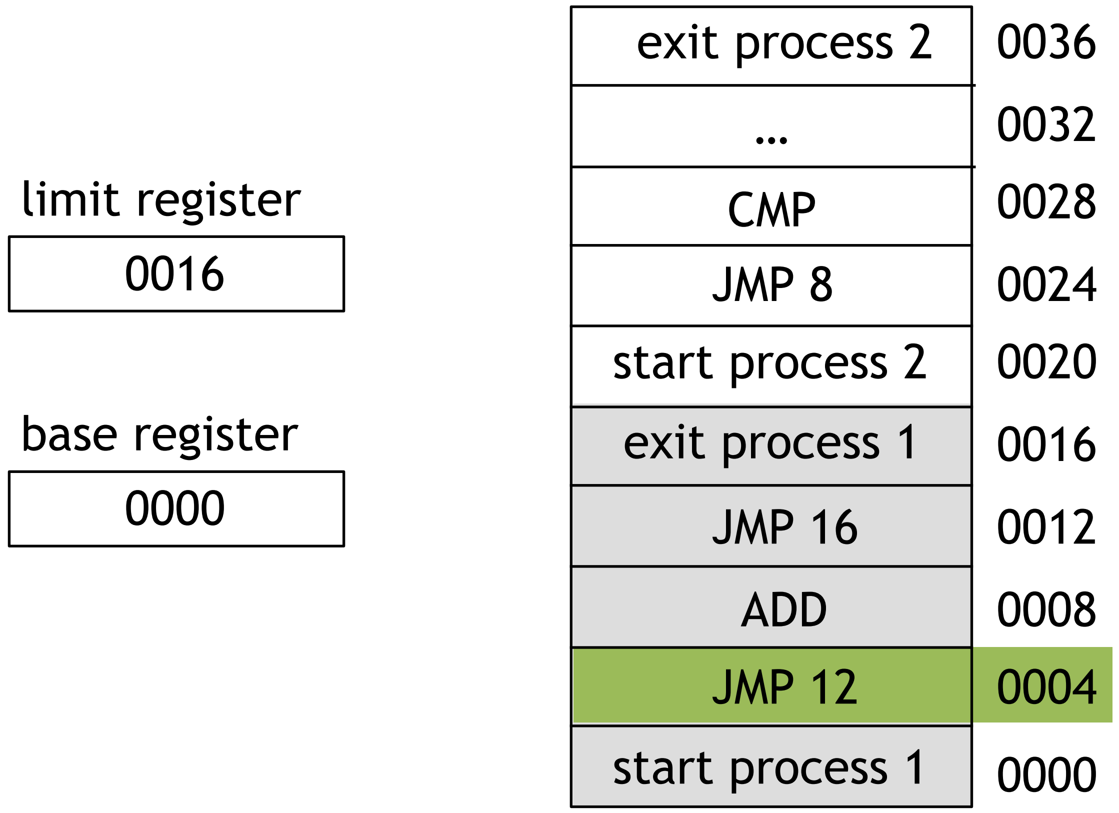
Running Multiple Programs Without a Memory Abstraction (3)



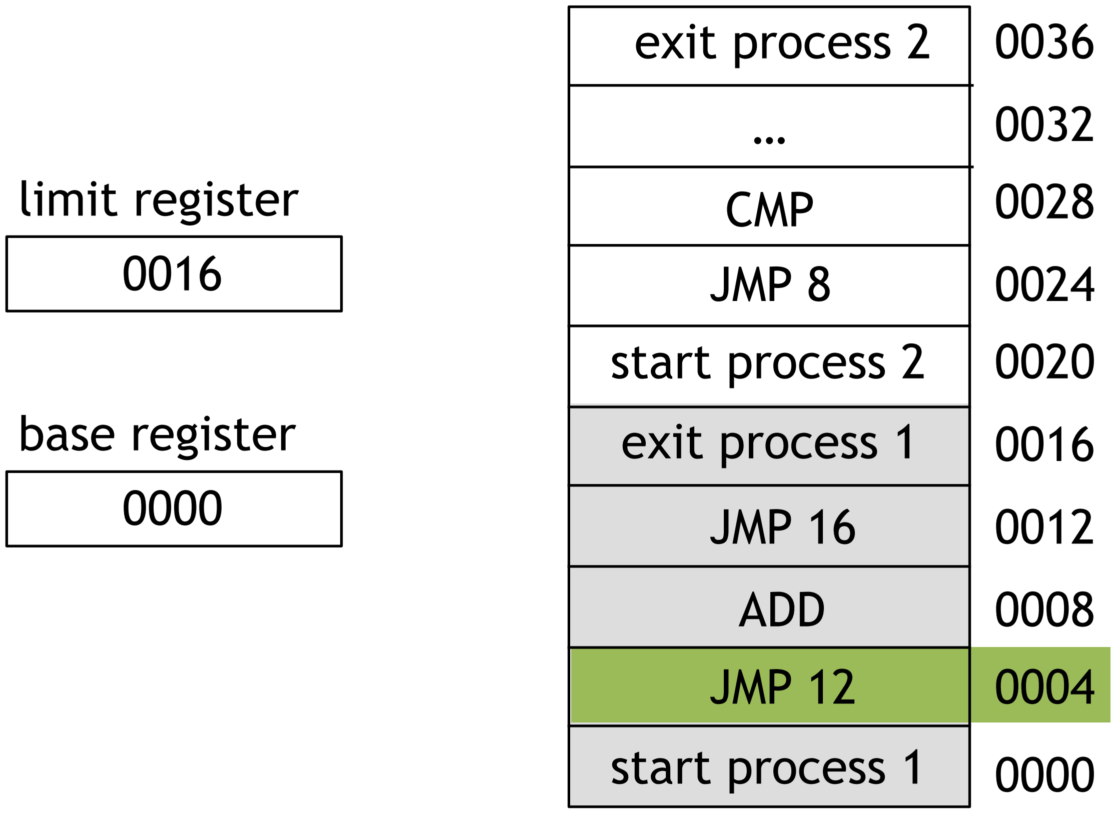
Running Multiple Programs Without a Memory Abstraction (3)



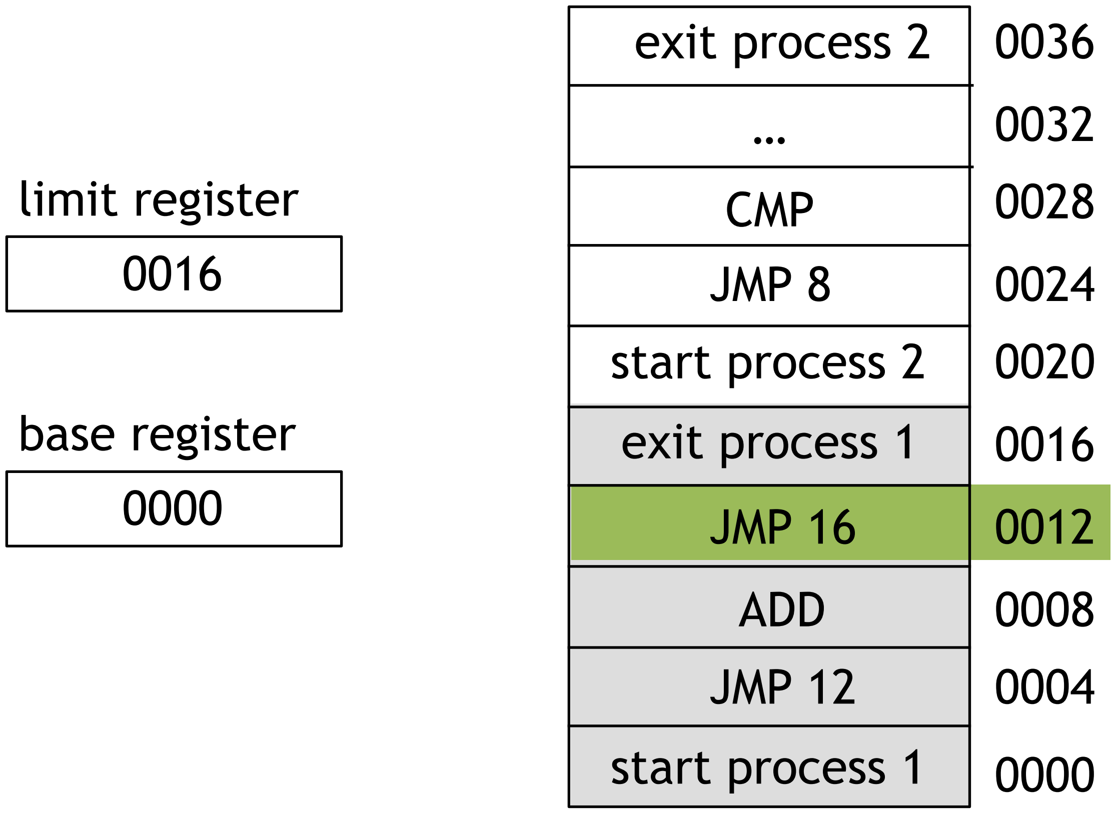
Running Multiple Programs Without a Memory Abstraction (3)



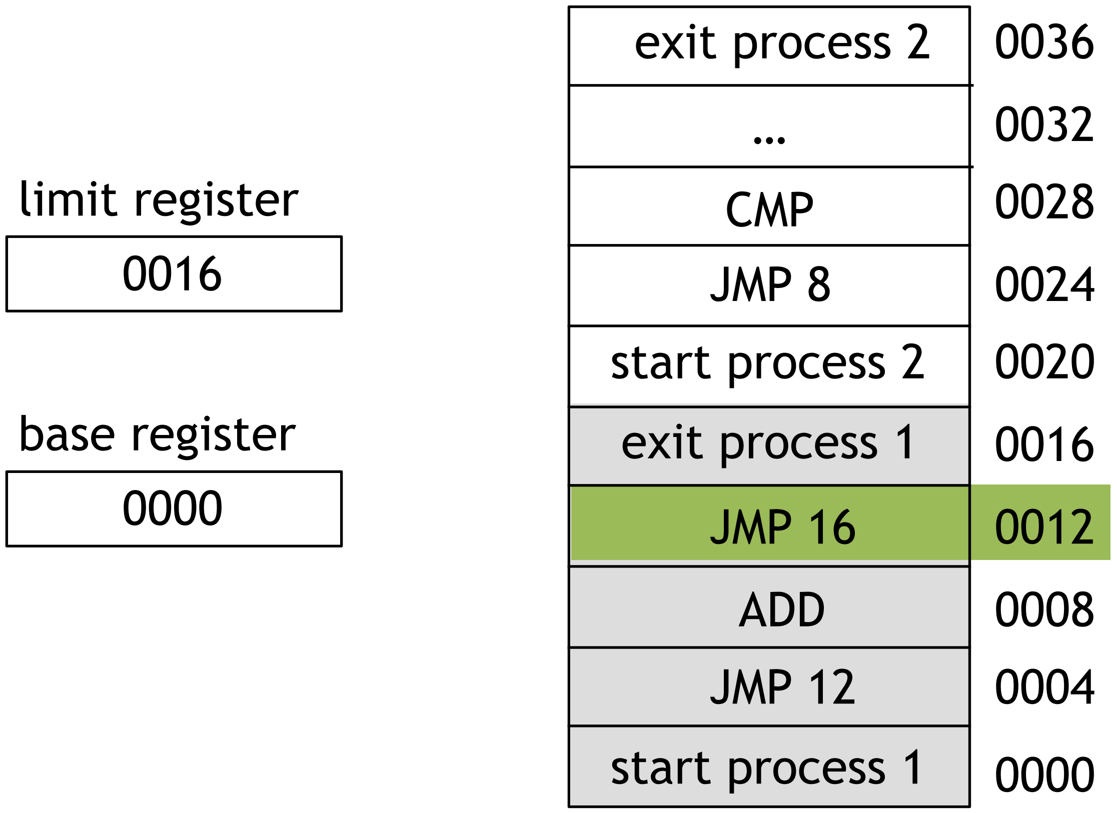
Running Multiple Programs Without a Memory Abstraction (3)



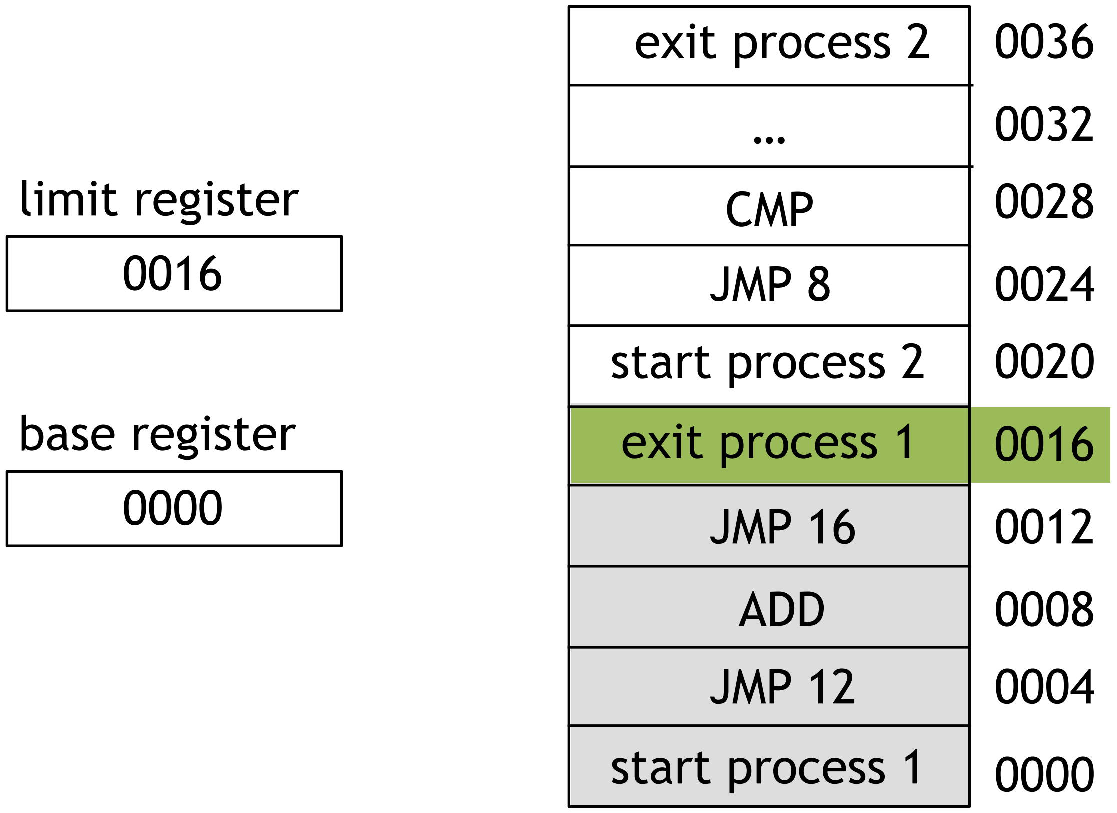
Running Multiple Programs Without a Memory Abstraction (3)



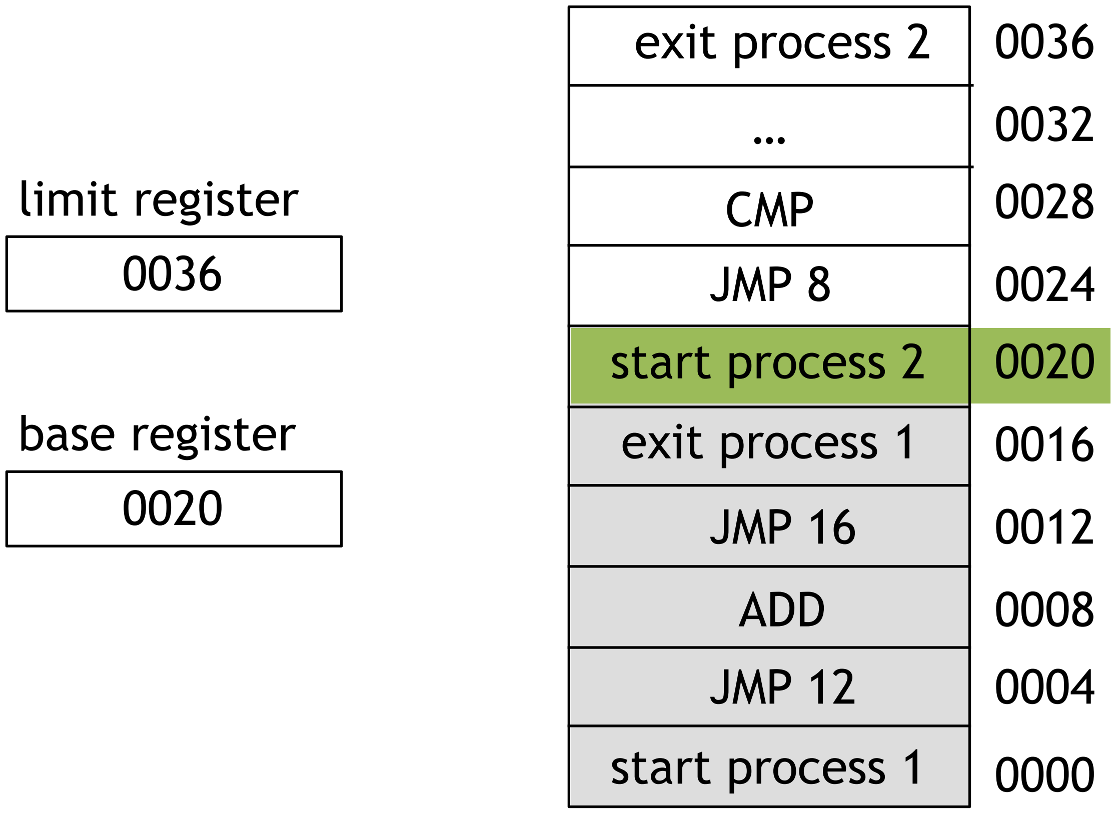
Running Multiple Programs Without a Memory Abstraction (3)



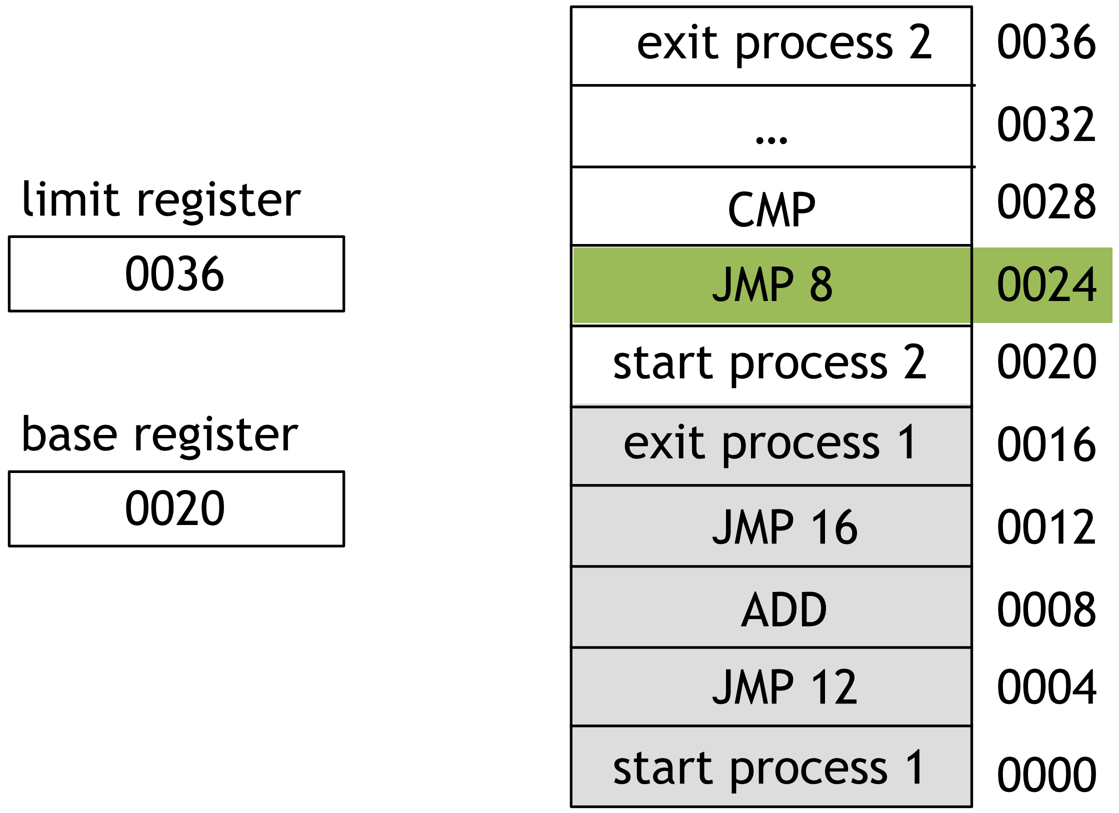
Running Multiple Programs Without a Memory Abstraction (3)



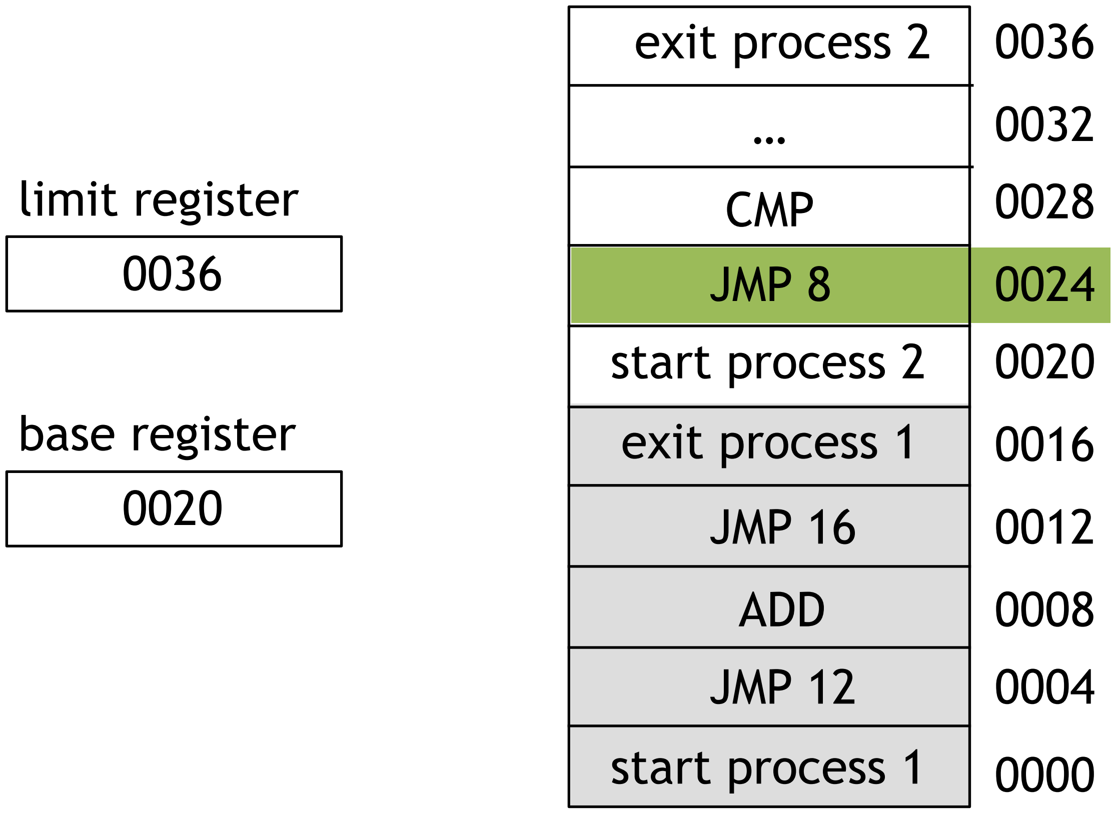
Running Multiple Programs Without a Memory Abstraction (3)



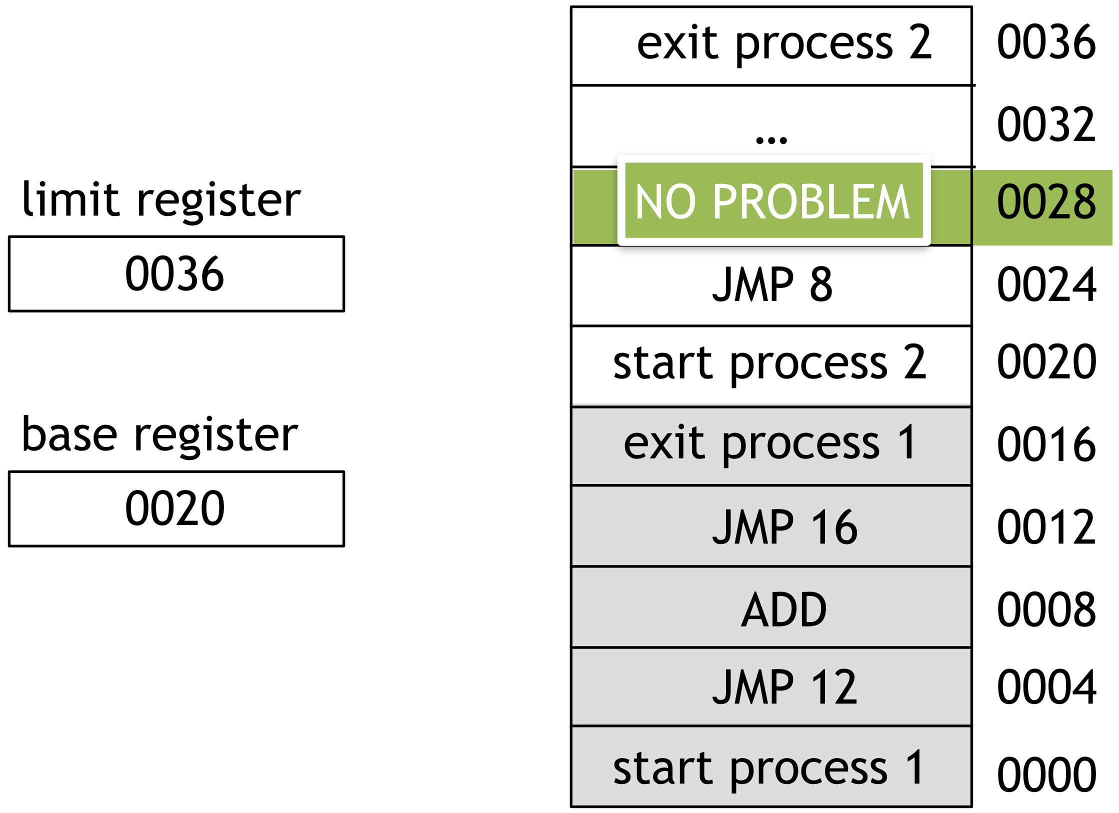
Running Multiple Programs Without a Memory Abstraction (3)



Running Multiple Programs Without a Memory Abstraction (3)



Running Multiple Programs Without a Memory Abstraction (3)



Swapping (1)

- All the running processes may require more memory than it is physically available
- There are two approaches to deal with this problem:
  - **Swapping** – idle processes are mostly stored on disk and are brought into main memory when needed, then are put back on disk
  - **Virtual memory** – programs run even when they are only partially in main memory (will be discussed later)

Swapping (2)

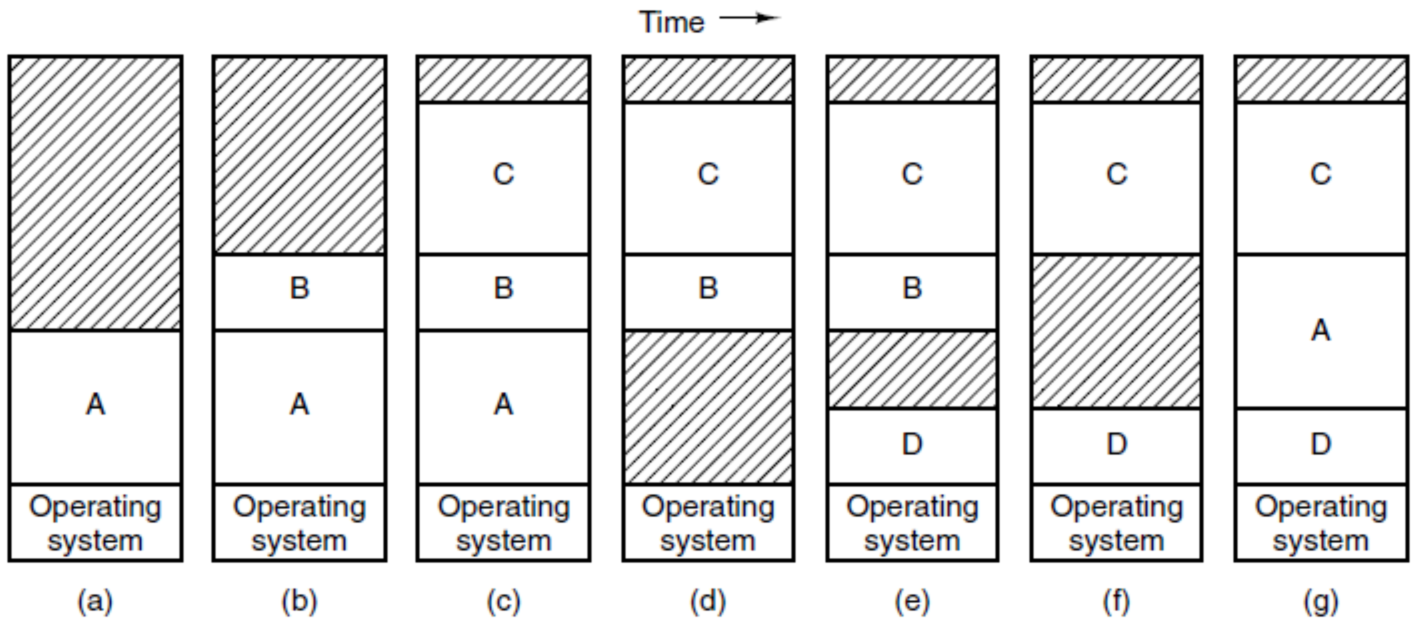


Figure 3-4. Memory allocation changes as processes come into memory and leave it. The shaded regions are unused memory

Swapping (3)

- After swapping in a process might be at a different location and addresses contained in it must be relocated, either by software or, more likely, by hardware during program execution. Base and limit registers would work fine here
- Swapping creates multiple holes in memory which are possible to combine into a big one by moving all the processes downward as far as possible
- Such a technique is known as **memory compaction**. However, it it requires a lot of CPU time, so it is usually not done

Swapping (4)

- 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, the OS allocates exactly what is needed
  - If processes' data segments can grow, for example, by dynamically allocating memory from a heap, a problem occurs whenever a process tries to grow

Swapping (5)

- If a hole is adjacent to the process, it can be allocated and the process allowed to grow into the hole
- 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 suspended until some space is freed up (or it can be killed)

Swapping (6)

- 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 (Fig. 3-5a)
- If processes can have two growing segments (the data segment being used as a heap and a stack segment), the memory between them can be used for either segment (Fig. 3-5b)

Swapping (7)

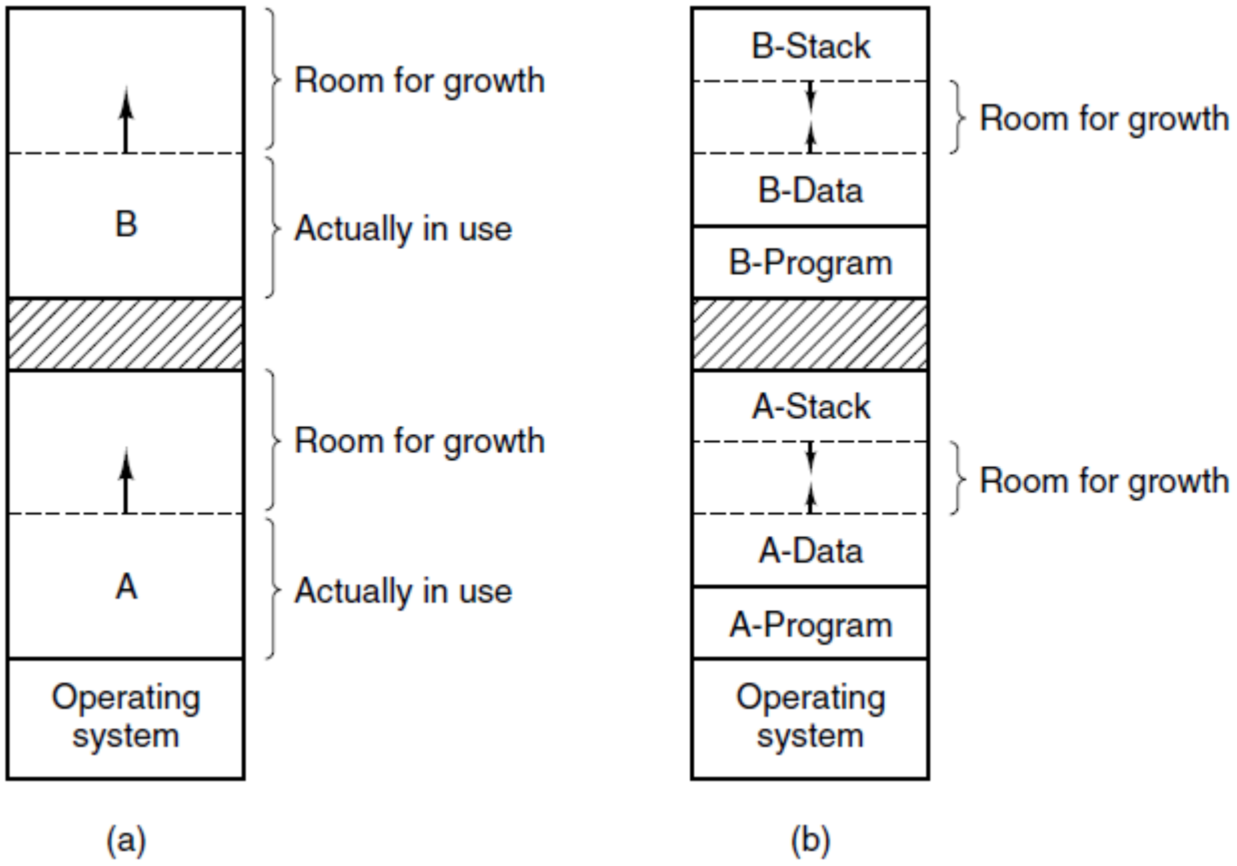


Figure 3-5. (a) Allocating space for a growing data segment. (b) Allocating space for a growing stack and a growing data segment.

Memory Management with Bitmaps (1)

- In general terms, there are two ways to keep track of memory usage: **bitmaps** and **free lists**
- 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 or vice versa (Fig. 3-6)

Memory Management with Bitmaps (2)

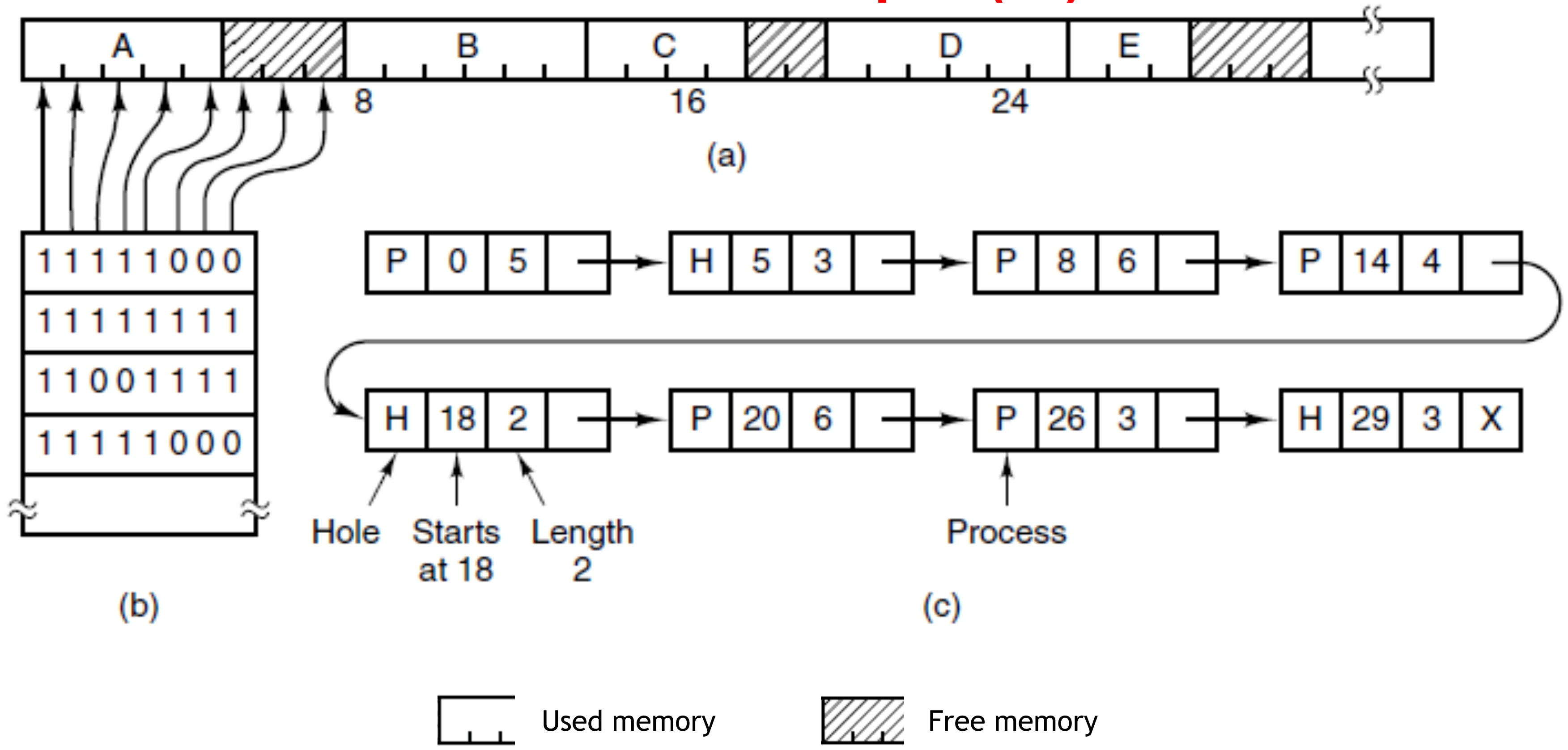


Figure 3-6. (a) A part of memory with five processes and three holes. (b) The corresponding bitmap. (c) The same information as a list.

Memory Management with Bitmaps (3)

- The size of the allocation unit is an important design issue
- The smaller the allocation unit, the larger the bitmap. With 4-bytes allocation unit 32 bits of memory will require 1 bit of the map, so the bitmap will take up only 1/32 of memory
- If the allocation unit is chosen large, the bitmap will be smaller, but memory may be wasted in the last unit of the process if the process size is not an exact multiple of the allocation unit

Memory Management with Bitmaps (4)

- A bitmap provides a simple way to keep track of memory words in a fixed amount of memory
- 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
- There is an argument against bitmaps: searching a bitmap for a sequence of a given length is a slow operation since the sequence may cross word boundaries in the map

Memory Management with Linked Lists (1)

- Another way of keeping track of memory is to maintain a linked list of allocated and free memory segments each element of which has the next structure:

H/P	**18**	**2**	●→
Hole or Process	Start of the segment	Length of the segment	Pointer to the next element

Memory Management with Linked Lists (2)

- Sorting the list by address has the advantage that when a process terminates or is swapped out, updating the list is straightforward
- If we have three consequent processes (Fig. 3-7a), updating the list requires replacing P by an H
- In case when there are two consequent processes (Fig. 3-7bc) two entries are merged into one
- The last case (Fig. 3-7d) is when a process is surrounded by two holes. It may be more convenient to have the list as a double-linked list, rather than the single-linked list and it will make it easier to find the previous entry and to see if a merge is possible

Memory Management with Linked Lists (3)

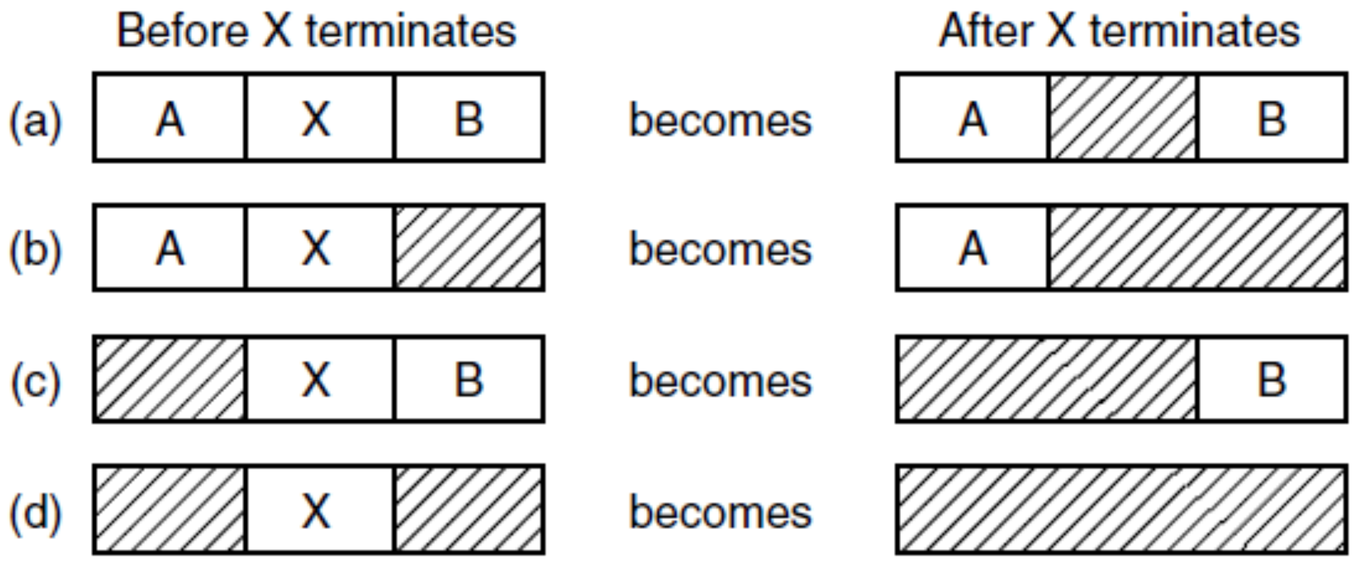


Figure 3-7. Four neighbor combinations for the terminating process, *X*.

Memory Management Algorithms (1)

- Several algorithms can be used to allocate memory for a created process:
  - **First fit** – the memory manager scans along the list of segments until it finds a hole that is big enough
  - **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

Memory Management Algorithms (2)

- **Best fit** – searches the entire list, from beginning to end, and takes the smallest hole that is adequate
- **Worst fit** – always takes the largest available hole, so that the new hole will be big enough to be useful
- **Quick fit** – maintains separate lists for some of the more common sizes requested

Virtual Memory (1)

- There is a need to run programs that are too large to fit in memory
- Swapping is not a solution since transfer rate of a hard disk is too slow and it will take seconds to swap in or out a large program
- Solution adopted in the 1960s, split programs into little pieces, called **overlays**
  - Kept on the disk, swapped in and out of memory

Virtual Memory (2)

- When a program started, only the overlay manager was loaded into memory
- It ran overlay 0 and, when it was done, the overlay manager loaded overlay 1 either above overlay 0 or on top of it (if there was no space)
- The work of splitting the program into pieces had to be done manually by the programmer

Virtual Memory (3)

- **Virtual memory**: each program has its own address space, broken up into chunks called **pages**
- Each page is a contiguous range of addresses and is mapped onto physical memory
- 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 not in physical memory, the OS gets the missing piece and re-executes the instruction that failed

Paging (1)

- On any computer, programs reference a set of memory addresses which 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**
- When virtual memory is used, the virtual addresses go to an MMU (Memory Management Unit) that maps the virtual addresses onto the physical memory addresses (Fig. 3-8)
- MMU is commonly a part of the CPU chip nowadays. However, logically it could be a separate chip and was years ago

Paging (2)

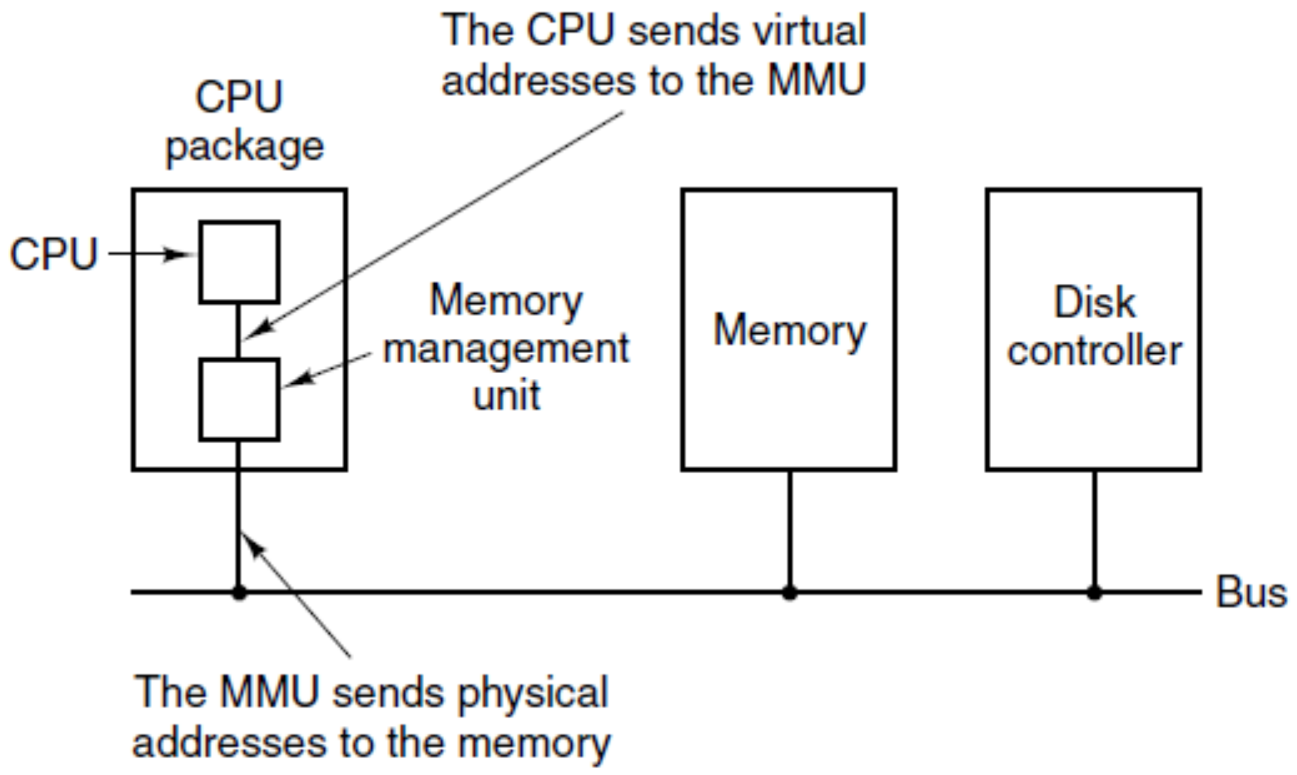

The CPU sends virtual addresses to the MMU
CPU package
CPU
Memory management unit
Memory
Disk controller
Bus
The MMU sends physical addresses to the memory

Figure 3-8. The position and function of the MMU

Paging (3)

- Paging example:
  - The computer has 32 KB of physical memory
  - It generates a set of 16-bit virtual addresses from 0 to 64K - 1
  - The virtual address space consists of fixed-size units called pages
  - The corresponding units in the physical memory are called **page frames**
  - Transfers between RAM and disk are always in whole pages

Paging (4)

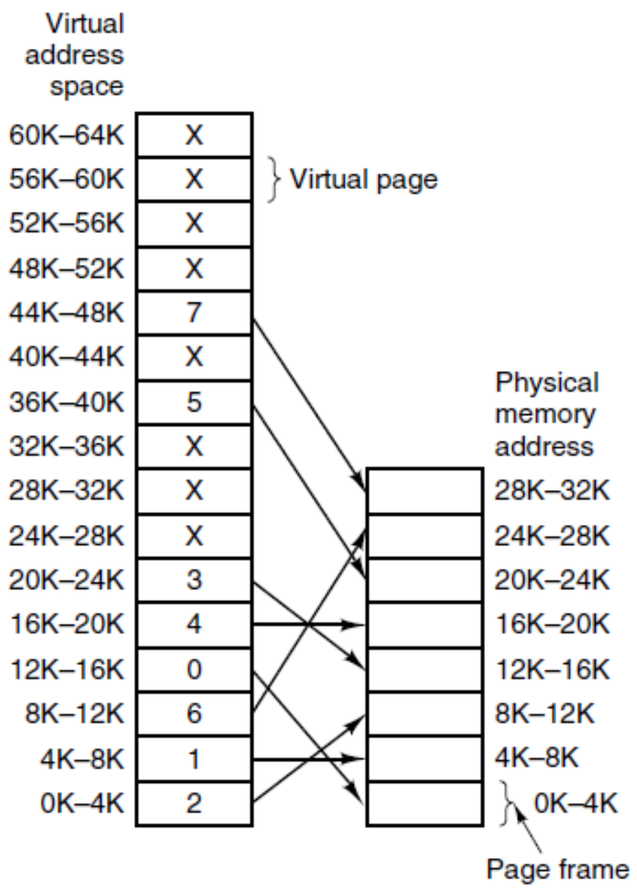


Figure 3-9. The relation between virtual addresses and physical memory addresses

Paging (5)

- 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
- When the program tries to access address 0, for example, using the instruction MOV REG,0 virtual address 0 is sent to the MMU which according to the mapping converts it to 8192 (first address of page frame 2)

Paging (6)

- Other examples:
  - the instruction MOV REG,8192 is converted into MOV REG,24576
  - 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

Paging (7)

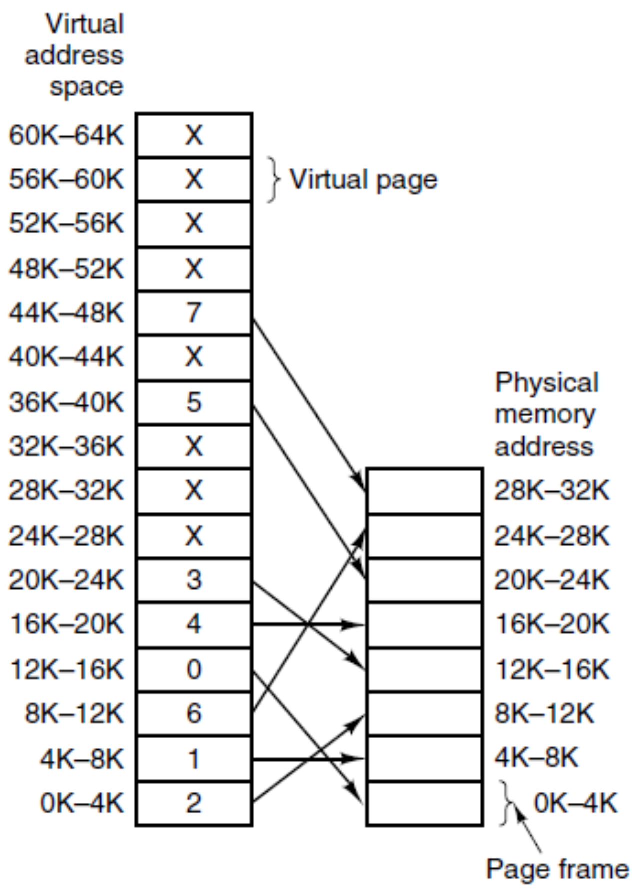


Figure 3-9. The relation between virtual addresses and physical memory addresses (notice X signs in virtual memory)

Paging (8)

- MMU's map does not solve the problem that the virtual address space is larger than the physical memory
- In the actual hardware, a **Present/absent bit** keeps track of which pages are physically present in memory
- If the program references an unmapped address, the MMU causes the CPU to trap to the OS
- Such a trap is called a **page fault**
- The OS swaps the page that is needed with a little-used frame, changes the map and restarts the instruction

Page Tables (1)

- The purpose of the page table is to map virtual pages onto page frames
- 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) within the selected page

Page Tables (2)

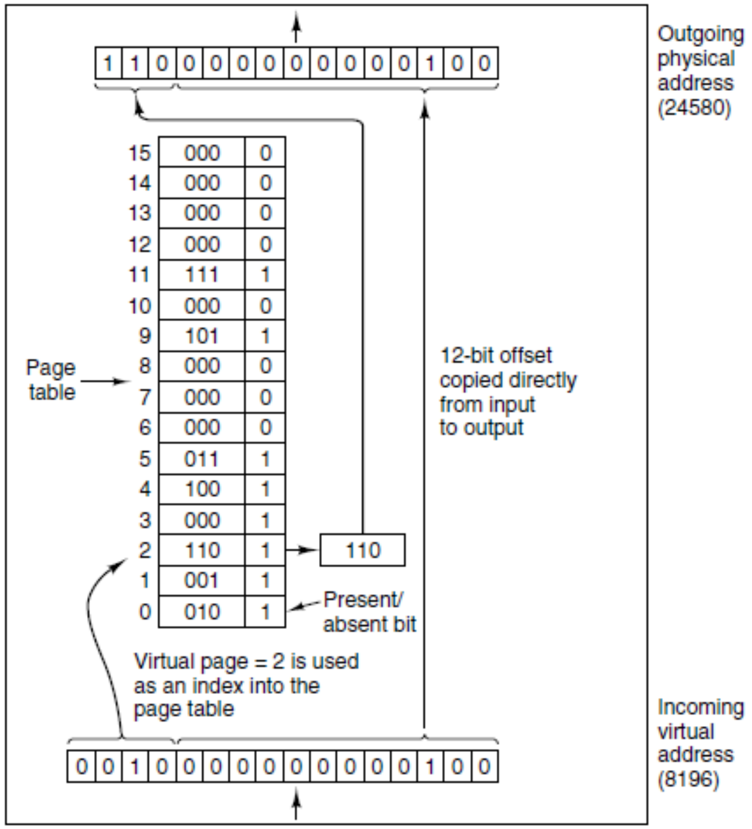


Figure 3-10. The internal operation of the MMU with 16 4-KB pages

Page Tables (3)

- **Structure of a Page Table Entry:**
  - The exact layout of an entry in the page table is highly machine dependent, but the information is roughly the same (Fig. 3-11):
    - Page frame number
    - Present/absent bit
    - Protection bits (r/w – one bit or r/w/x – three bits)
    - Modified bit (dirty bit)
    - Referenced bit
    - Caching disabled bit

Page Tables (4)

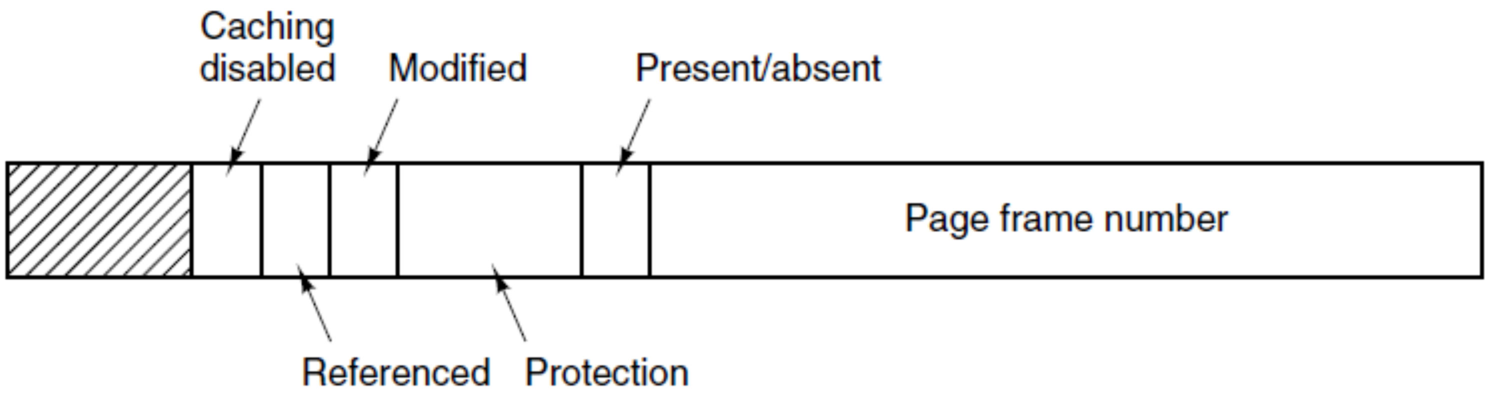


Figure 3-11. A typical page table entry.

Speeding Up Paging

- Major issues faced:
  - The mapping from virtual address to physical address must be fast: the virtual-to-physical mapping must be done on every memory reference
  - If the virtual address space is large, the page table will be large: a 32-bit address space has 1 million pages, the page table must have 1 million entries and each process requires its own page table

Translation Lookaside Buffers (1)

- In the absence of paging, a 1-byte 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 which would reduce performance by half

Translation Lookaside Buffers (2)

- The solution is to equip computers with a small hardware device for mapping virtual addresses to physical addresses without going through the page table
- It 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
- The device is called a **TLB** (**Translation Lookaside Buffer**) or an **associative memory** (Fig. 3-12)

Translation Lookaside Buffers (3)

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 structure

Translation Lookaside Buffers (4)

- 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 in parallel.
- If a valid match is found and the access does not violate the protection bits, the page frame is taken directly from the TLB (a protection fault is generated otherwise)

Translation Lookaside Buffers (5)

- When the virtual page number is not in the TLB, the MMU does an ordinary page table lookup
- It removes one of the entries from the TLB and replaces it with the page table entry just looked up
- When an entry is purged from the TLB, only the **modified bit** is copied back into the page table entry in memory since the other values are already there except the reference bit
- When the TLB is loaded from the page table, all the fields are taken from memory

Software TLB Management (1)

- If the TLB is moderately large (say, 64 entries) to reduce the miss rate, software management of the TLB turns out to be acceptably efficient
- The main gain is a much simpler MMU, which frees up a considerable amount of area on the CPU chip for caches and other features that can improve performance

Software TLB Management (2)

- When software TLB management is used, it is essential to understand the difference between different kinds of misses:
  - A **soft miss** occurs when the page referenced is not in the TLB, but is in memory
  - A hard miss occurs when the page itself is neither in TLB nor in memory. A **page table walk** (a process of looking up the mapping in the page table hierarchy) is needed

Software TLB Management (3)

- There are three types of situations that might occur when the page walk does not find the page in the process' page table:
  - A **minor page fault** occurs when the page is actually in memory, but not in this process' page table (it may have been brought in from disk by another process)
  - A **major page fault** occurs if the page needs to be brought in from disk
  - A **segmentation fault** occurs when the program simply accesses an invalid address and no mapping needs to be added in the TLB

Multilevel Page Tables (1)

- The purpose of the multilevel page table method is to avoid keeping all the page tables in memory all the time
- Each top-level page table entry contains PT1 field, PT2 field and offset (Fig. 3-13a)
- 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 (Fig. 3-13b)

Multilevel Page Tables (2)

- 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
- After the frame page is found accessing the necessary memory address is pretty straightforward by using the offset

Multilevel Page Tables (3)

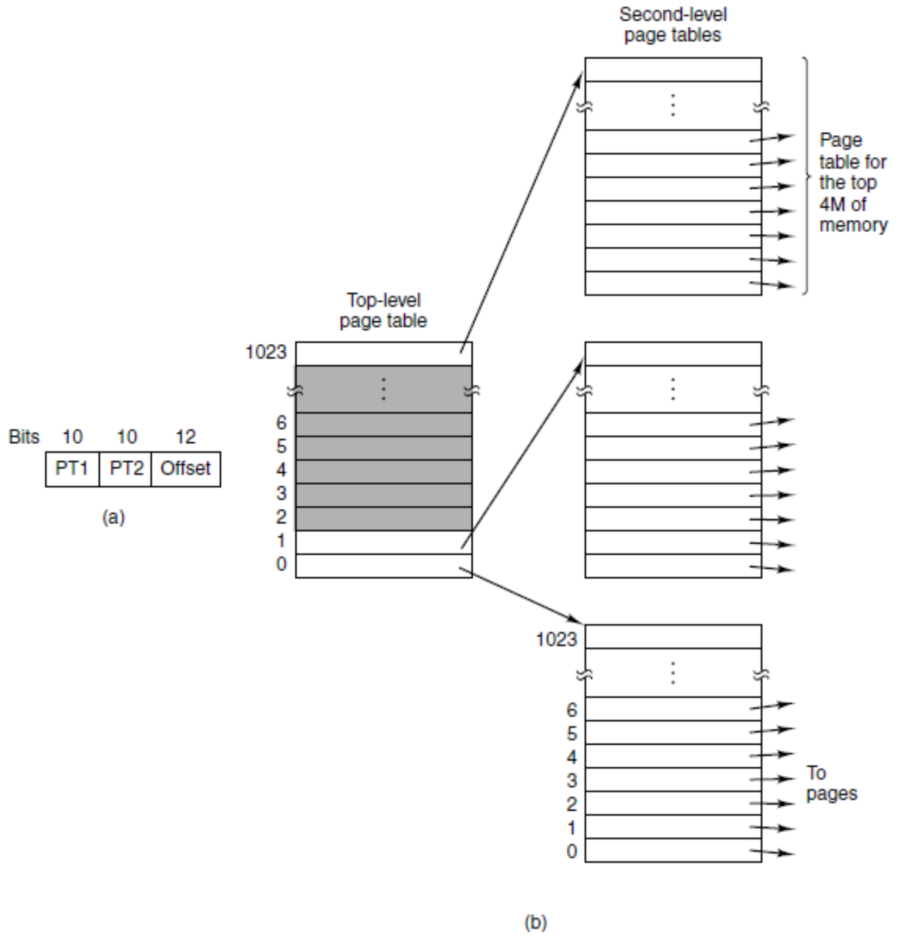


Figure 3-13. (a) A 32-bit address with two page table fields. (b) Two-level page tables.

Inverted Page Tables (1)

- An alternative to a paging hierarchy is known as **inverted page tables**
- There is one entry per page frame in real memory, rather than one entry per page of virtual address space, so the size is proportional to physical memory, not the virtual address space
- The entry contains a pair (process ID, virtual page address) and refers to a page frame

Inverted Page Tables (2)

- Such an approach helps saving space, but has a serious pitfall:
  - when process *n* references virtual page p, it is no longer possible to find the physical page by using *p* as an index into the page table. Instead, we must search the entire inverted page table for an entry (n, p). This search must be done on every memory reference, not just on page faults

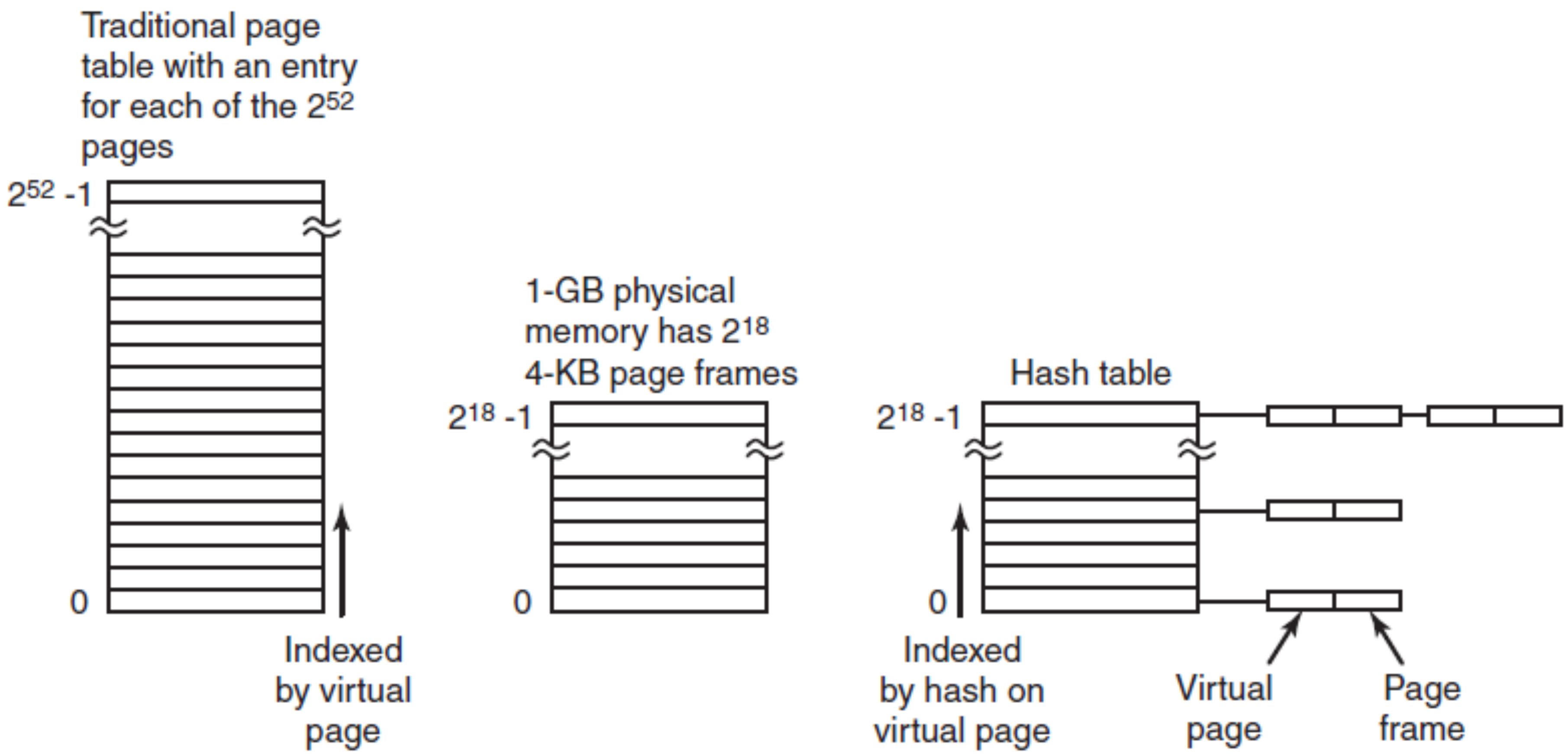
Inverted Page Tables (3)

- To solve this problem we can use the TLB. If the TLB can hold all of the heavily used pages, translation is as fast as with regular page tables
- On a TLB miss the inverted page table has to be searched in software
- We can use hash tables to make the search faster (Fig. 3-14)

Inverted Page Tables (4)

- Now the algorithm to process the misses looks like this:
  - We organize the inverted page table as a hash table
  - We use hash function for given (PID, page number)
  - If we have a match, translate the address. If not, use collision resolution technique (rehash, search, linear probing) and search again

Inverted Page Tables (5)



Traditional page table with an entry for each of the 2^{52} pages
2^{52} -1
0
Indexed by virtual page
1-GB physical memory has 2^{18} 4-KB page frames
2^{18} -1
0
Hash table
2^{18} -1
0
Indexed by hash on virtual page
Virtual page
Page frame

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

End

Week 7

References

- Tanenbaum & Bos, Modern Operating Systems: 4th edition, 2013 Prentice-Hall, Inc.