Operating Systems

11. Memory: Main Memory

Hyunchan, Park

http://oslab.chonbuk.ac.kr

Division of Computer Science and Engineering

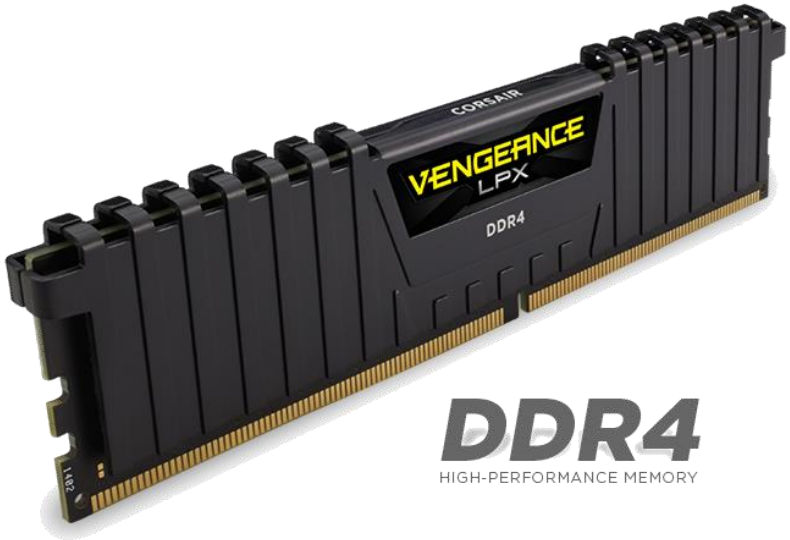
Chonbuk National University

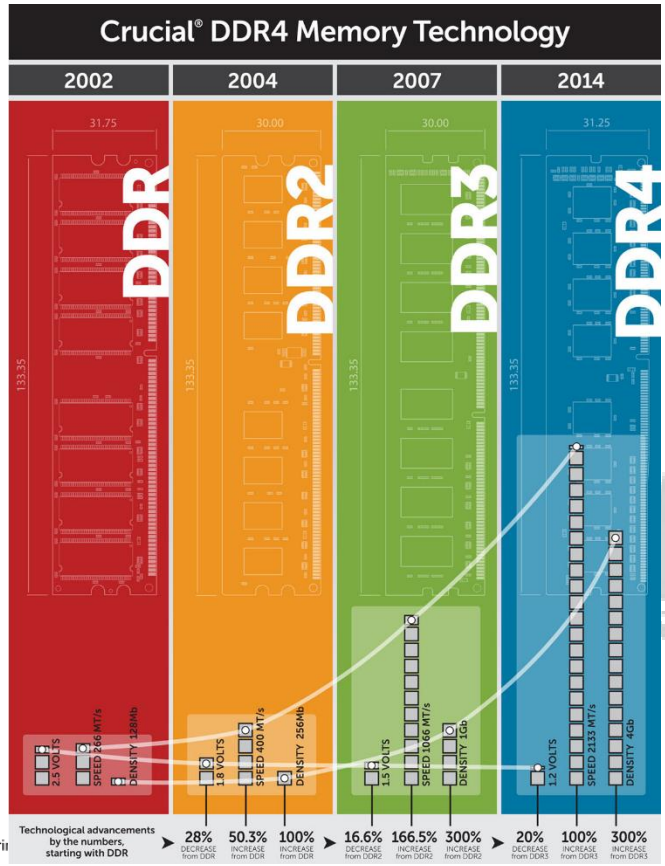
Contents

- Background
  - Address binding Schemes: Compile, Load and Execution time
  - Logical and physical memory
  - Address Translation w/ MMU: Dynamic relocation
- Memory Allocation: Contiguous and Multiple partition
  - Issues: Swapping and Fragmentation

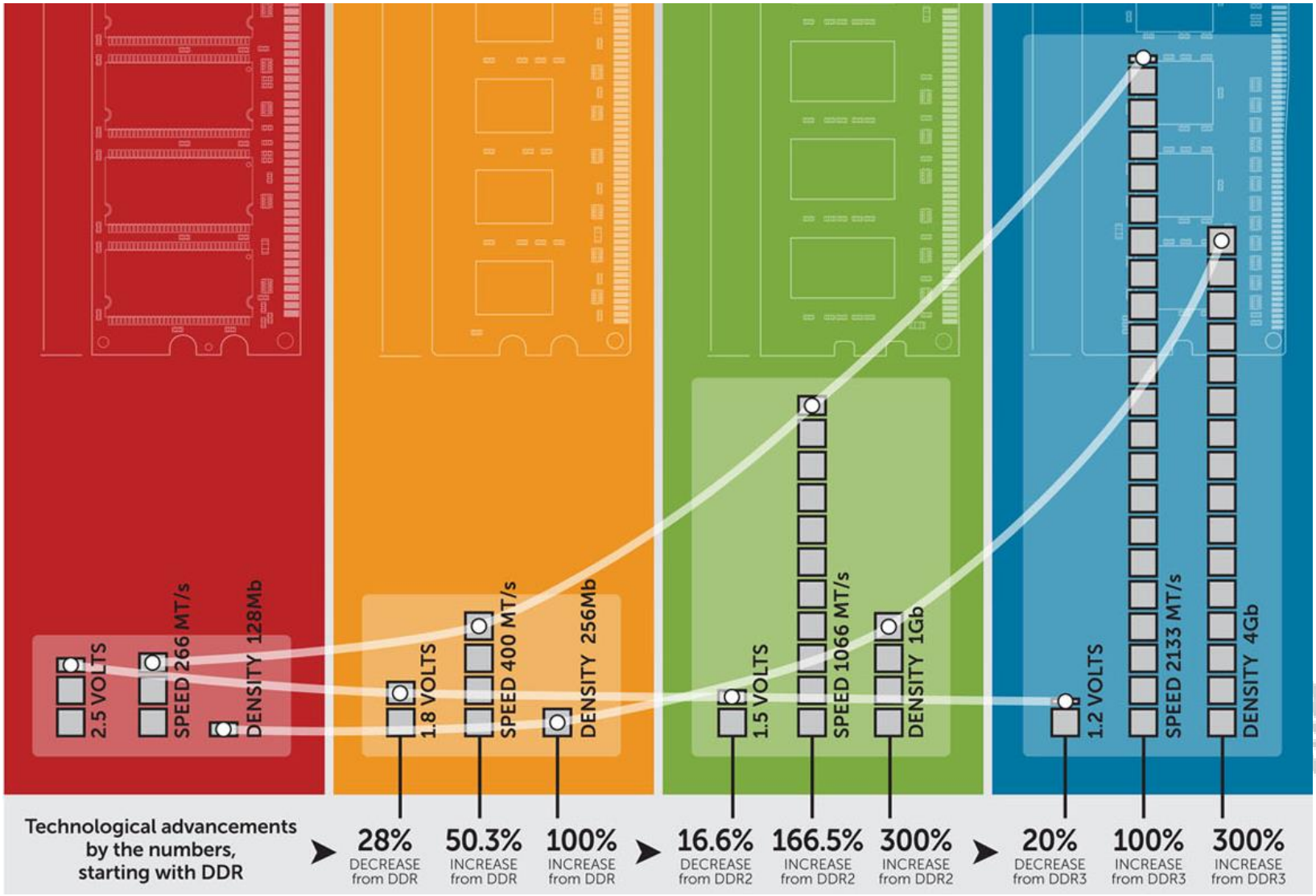
Main memory


DDR3
DDR4


CORSAIR
VENGEANCE
LPX
DDR4
DDR4
HIGH-PERFORMANCE MEMORY


Crucial® DDR4 Memory Technology
2002
2004
2007
2014
DDR
DDR2
DDR3
DDR4
31.75
30.00
30.00
31.25
133.35
2.5 VOLTS
SPEED 266 MT/s
DENSITY 128Mb
1.8 VOLTS
SPEED 400 MT/s
DENSITY 256Mb
1.5 VOLTS
SPEED 1066 MT/s
DENSITY 1Gb
1.2 VOLTS
SPEED 2133 MT/s
DENSITY 4Gb
Technological advancements by the numbers, starting with DDR
28% DECREASE from DDR
50.3% INCREASE from DDR
100% INCREASE from DDR
16.6% DECREASE from DDR2
166.5% INCREASE from DDR2
300% INCREASE from DDR2
20% DECREASE from DDR3
100% INCREASE from DDR3
300% INCREASE from DDR3

Main memory


2.5 VOLTS
SPEED 266 MT/s
DENSITY 128Mb
1.8 VOLTS
SPEED 400 MT/s
DENSITY 256Mb
1.5 VOLTS
SPEED 1066 MT/s
DENSITY 1Gb
1.2 VOLTS
SPEED 2133 MT/s
DENSITY 4Gb
Technological advancements by the numbers, starting with DDR
28% DECREASE from DDR
50.3% INCREASE from DDR
100% INCREASE from DDR
16.6% DECREASE from DDR2
166.5% INCREASE from DDR2
300% INCREASE from DDR2
20% DECREASE from DDR3
100% INCREASE from DDR3
300% INCREASE from DDR3

Background

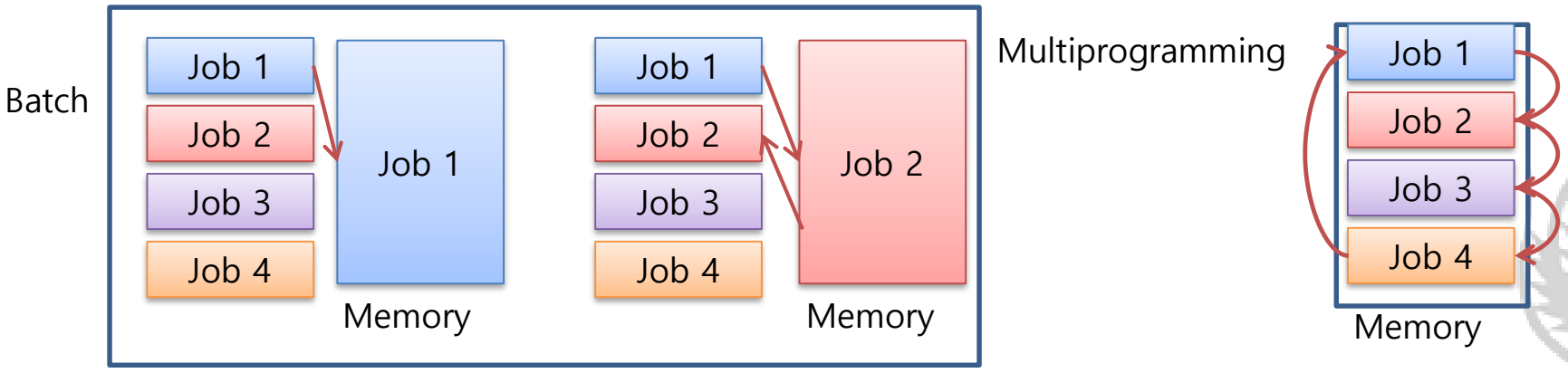
- Program must be brought (from) into and placed within a process for it to be run
- M and are only storage CPU can access directly
- Memory unit only sees a stream of
  - addresses + read requests, or
  - address + data and write requests
- Register access in CPU clock (or less)
- Access for can take many cycles, causing a stall
- C sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

Background

- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Memory unit only sees a stream of
  - addresses + read requests, or
  - address + data and write requests
- Register access in one CPU clock (or less)
- Access for main memory can take many cycles, causing a stall
- Cache sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

Remind: Batch and Multiprogramming

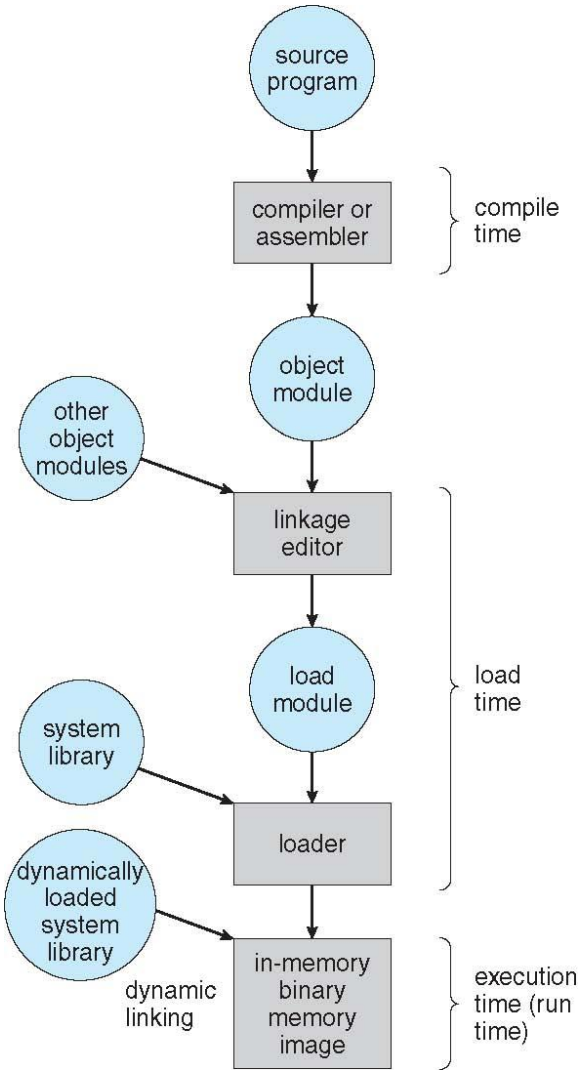
- 2개 이상의 작업을 동시에 실행
  - 운영체제는 여러 개의 작업을 메모리에 동시에 유지
  - 현재 실행중인 작업이 I/O를 할 경우 다음 작업을 순차적 실행
  - 스케줄링 고려사항 – first-come, first-served
- 장점: CPU 활용률 극대화
  - CPU가 쉬는 시간이 없음



Address Binding

- Programs on disk, ready to be brought into memory to execute form an input queue
  - Without support, must be loaded into address 0x0000
- Inconvenient to have first user process physical address always at 0x0000
  - How can it not be?
- Further, addresses represented in different ways at different stages of a program's life
  - Source code addresses usually symbolic
  - Compiled code addresses bind to relocatable addresses
    - i.e. "14 bytes from beginning of this module"
  - Linker or loader will bind relocatable addresses to absolute addresses
    - i.e. 74014
  - Each binding maps one address space to another

Multistep Processing of a User Program



FYI. Dynamic Linking

- Static linking – system libraries and program code combined by the loader into the binary program image
- Dynamic linking –linking postponed until execution time
- Small piece of code, stub, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system checks if routine is in processes' memory address
  - If not in address space, add to address space
- Dynamic linking is particularly useful for libraries
  - System also known as shared libraries

Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at three different stages
  - Compile time: If memory location known a priori, absolute code can be generated; must recompile code if starting location changes
  - Load time: Must generate relocatable code if memory location is not known at compile time
  - Execution time: Binding delayed until run time if the process can be moved during its execution from one memory segment to another
    - Need hardware support for address maps (e.g., base and limit registers)

Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a separate physical address space is central to proper memory management
  - Logical address – generated by the CPU; also referred to as virtual address
    - The set of all logical addresses generated by a program
  - Physical address – address seen by the memory unit
- Logical and physical addresses are the same
  - in compile-time and load-time address-binding schemes;
- Logical (virtual) and physical addresses differ
  - in execution-time address-binding scheme

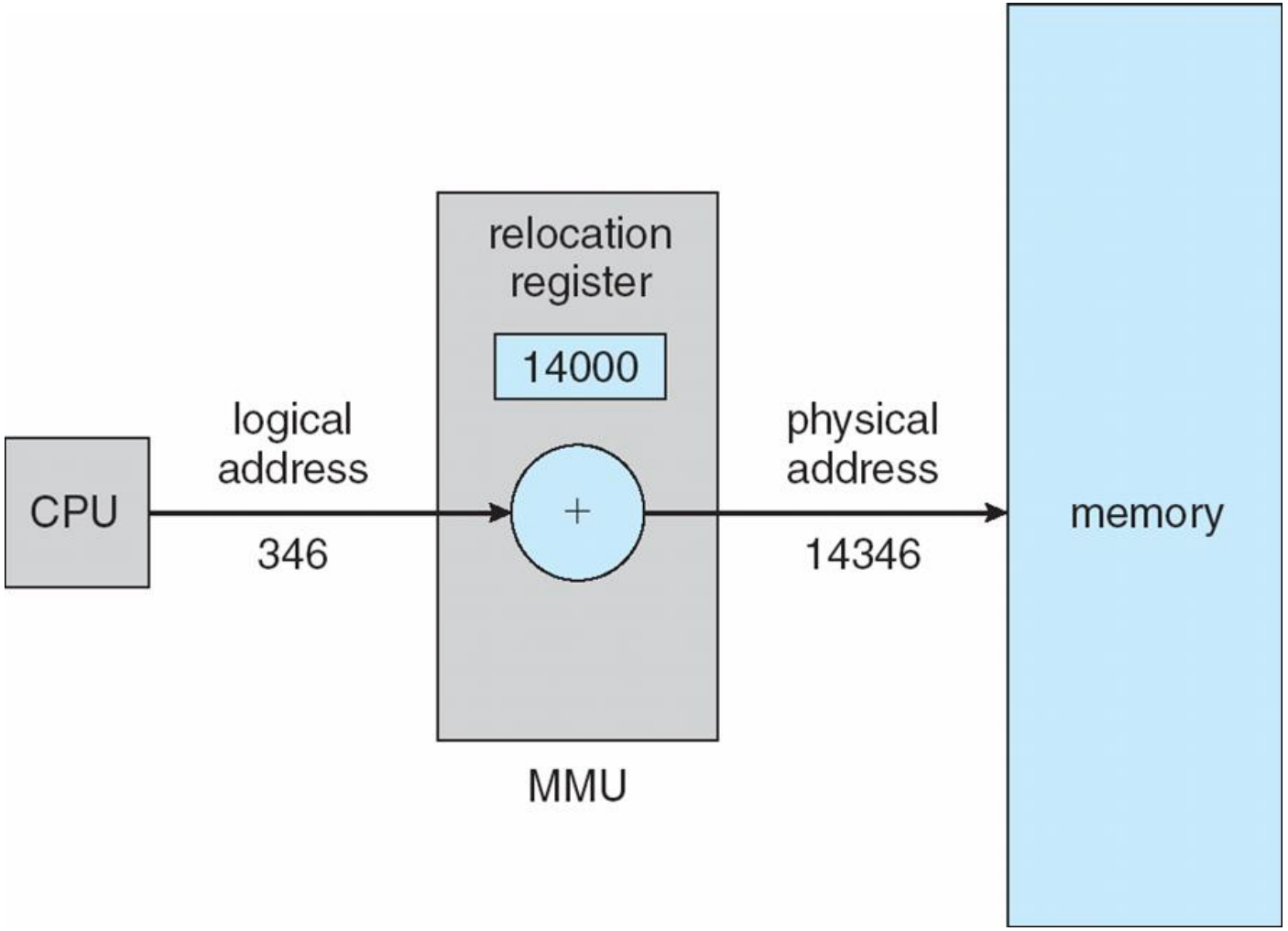
Virtual address space

- Virtual address space – logical view of how process is stored in memory
  - Usually start at address 0, contiguous addresses until end of space
  - MMU must map logical to physical
- The user program deals with logical addresses; it never sees the real physical addresses
  - Execution-time binding occurs when reference is made to location in memory
  - Logical address bound to physical addresses
- Virtual memory can be implemented via:
  - Demand paging and Demand segmentation (will be covered next)

Address translation with Memory-Management Unit (MMU)

- MMU: Hardware device that at run time maps virtual to physical address
  - Many methods possible, covered in the rest of this chapter
- To start, consider simple scheme where the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
  - Base register now called relocation register
  - MS-DOS on Intel 80x86 used 4 relocation registers

Dynamic relocation using a relocation register

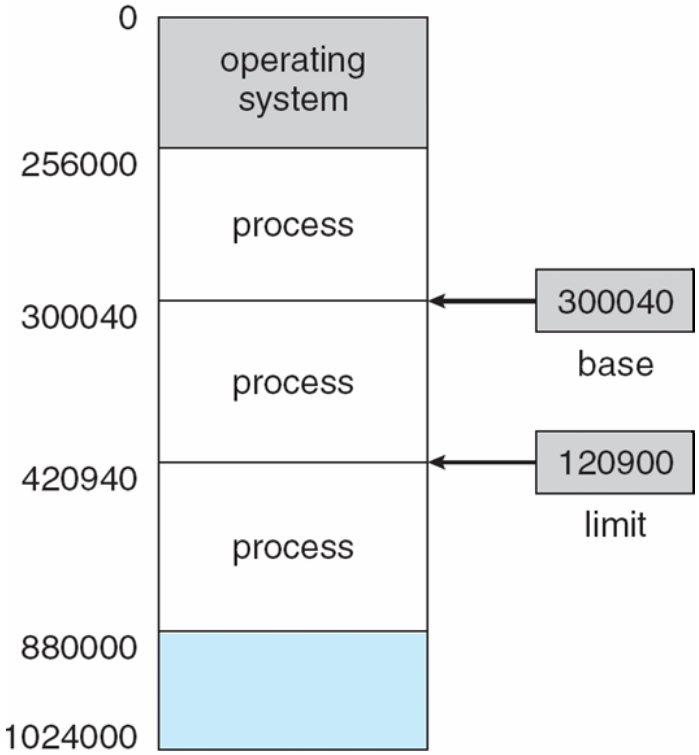


Dynamic allocation using a relocation register

- Routine is not loaded until it is called
  - All routines kept on disk in relocatable load format
  - Better memory-space utilization; unused routine is never loaded
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required
  - Implemented through program design
  - OS can help by providing libraries to implement dynamic loading

HW Protection with Base and Limit Registers

- A pair of base and limit registers define the logical address space
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user



Contiguous Allocation

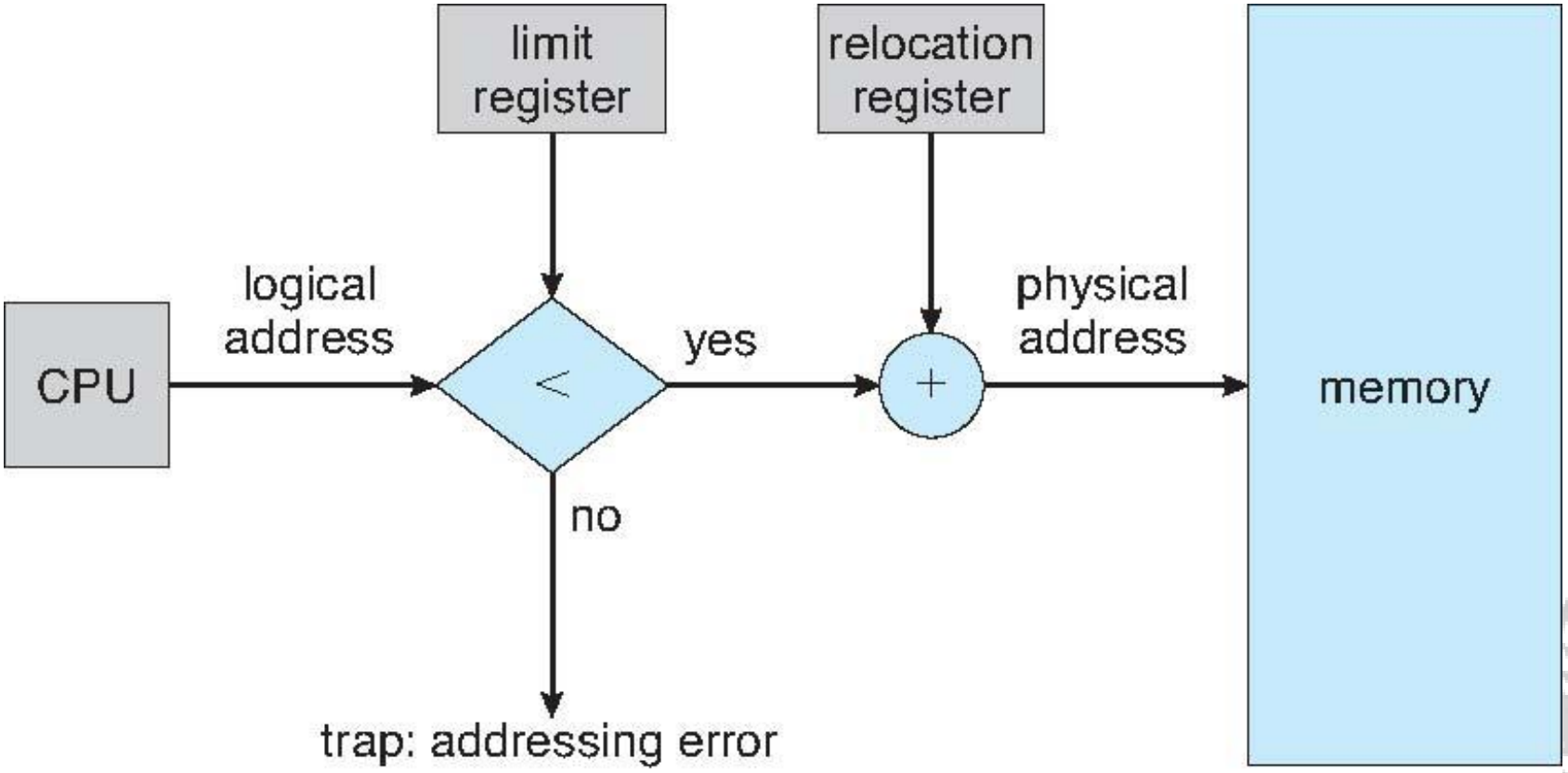
- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently

- Contiguous allocation is one early method
- Main memory usually into two partitions:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
  - Each process contained in single contiguous section of memory

Contiguous Allocation (Cont.)

- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Base register contains value of smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address dynamically
  - Can then allow actions such as kernel code being transient and kernel changing size

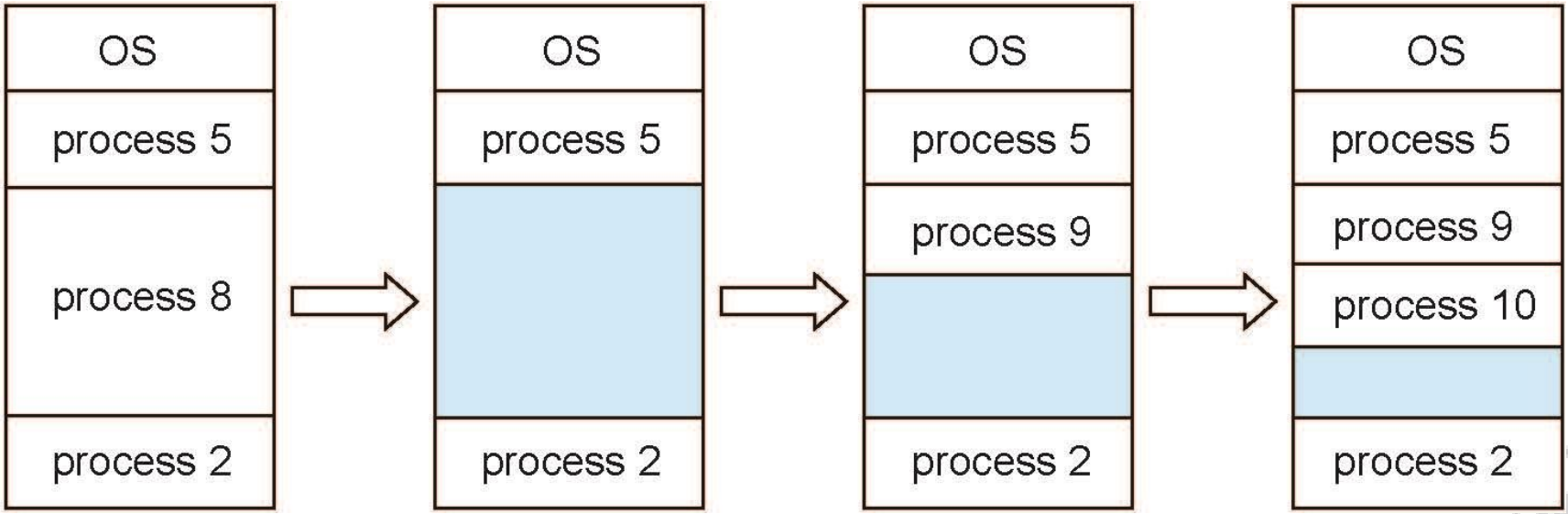
HW Support for Relocation and Limit Registers



Multiple-partition allocation

- Multiple-partition allocation
  - Variable-partition sizes for efficiency (sized to a given process' needs)
  - Hole – block of available memory; holes of various size are scattered throughout memory
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it
  - Process exiting frees its partition, adjacent free partitions combined
  - Operating system maintains information about:
    a) allocated partitions b) free partitions (hole)

Multiple-partition allocation



Problem of Dynamic Allocation

- How to satisfy a request of size n from a list of free holes?
  - First-fit: Allocate the first hole that is big enough
  - Best-fit: Allocate the smallest hole that is big enough; must search entire list, unless ordered by size
    - Produces the smallest leftover hole
  - Worst-fit: Allocate the largest hole; must also search entire list
    - Produces the largest leftover hole
- First-fit and best-fit better than worst-fit in terms of speed and storage utilization

Issue: Fragmentation

- External Fragmentation – total memory space exists to satisfy a request, but it is not contiguous
- Internal Fragmentation – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- First fit analysis reveals that given N blocks allocated, 0.5 N blocks lost to fragmentation
  - 1/3 may be unusable -> 50-percent rule

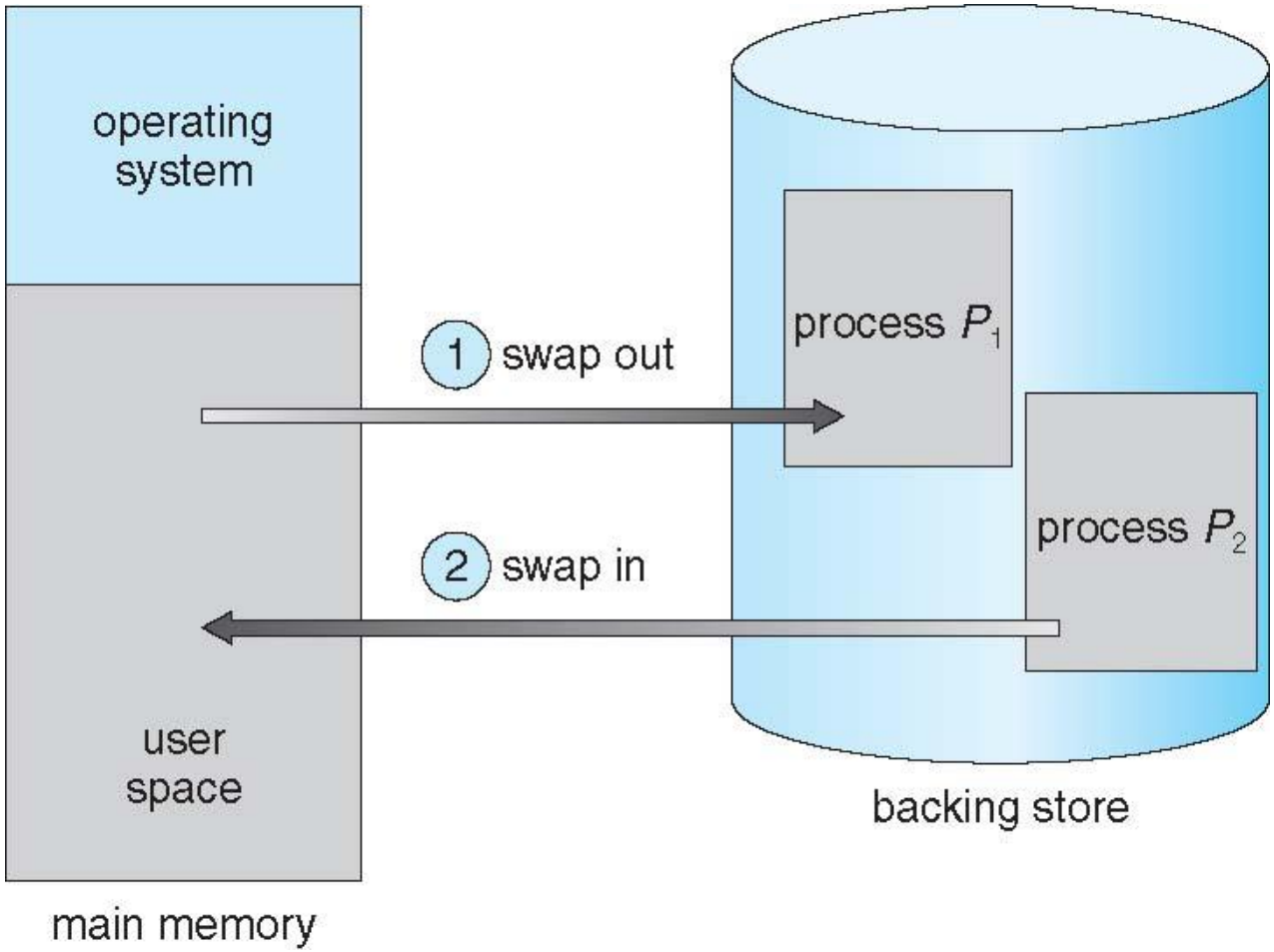
Issue: Fragmentation (Cont.)

- Reduce external fragmentation by compaction
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible only if relocation is dynamic, and is done at execution time
  - I/O problem
    - Latch job in memory while it is involved in I/O
    - Do I/O only into OS buffers
- Now consider that backing store has same fragmentation problems

Issue: Swapping

- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution
  - Total physical memory space of processes can exceed physical memory
- Backing store: fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- Roll out, roll in: swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- System maintains a ready queue of ready-to-run processes which have memory images on disk

Issue: Schematic View of Swapping



Issue: Swapping (Cont.)

- Does the swapped out process need to swap back in to same physical addresses?
  - Depends on address binding method
  - Plus consider pending I/O to / from process memory space
- Standard swapping not used in modern operating systems
  - But modified version common
    - Swap only when free memory extremely low
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
  - Swapping normally disabled
  - Started if more than threshold amount of memory allocated
  - Disabled again once memory demand reduced below threshold

Issue: Context Switch Time including Swapping

- If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
- Context switch time can then be very high
- 100MB process swapping to hard disk with transfer rate of 50MB/sec
  - Swap out time of 2000 ms
  - Plus swap in of same sized process
  - Total context switch swapping component time of 4000ms (4 seconds)
- Can reduce if reduce size of memory swapped – by knowing how much memory really being used
  - System calls to inform OS of memory use via request_memory() and release_memory()

Issue: Context Switch Time and Swapping (Cont.)

- Other constraints as well on swapping
  - Pending I/O – can't swap out as I/O would occur to wrong process
  - Or always transfer I/O to kernel space, then to I/O device
    - Known as double buffering, adds overhead