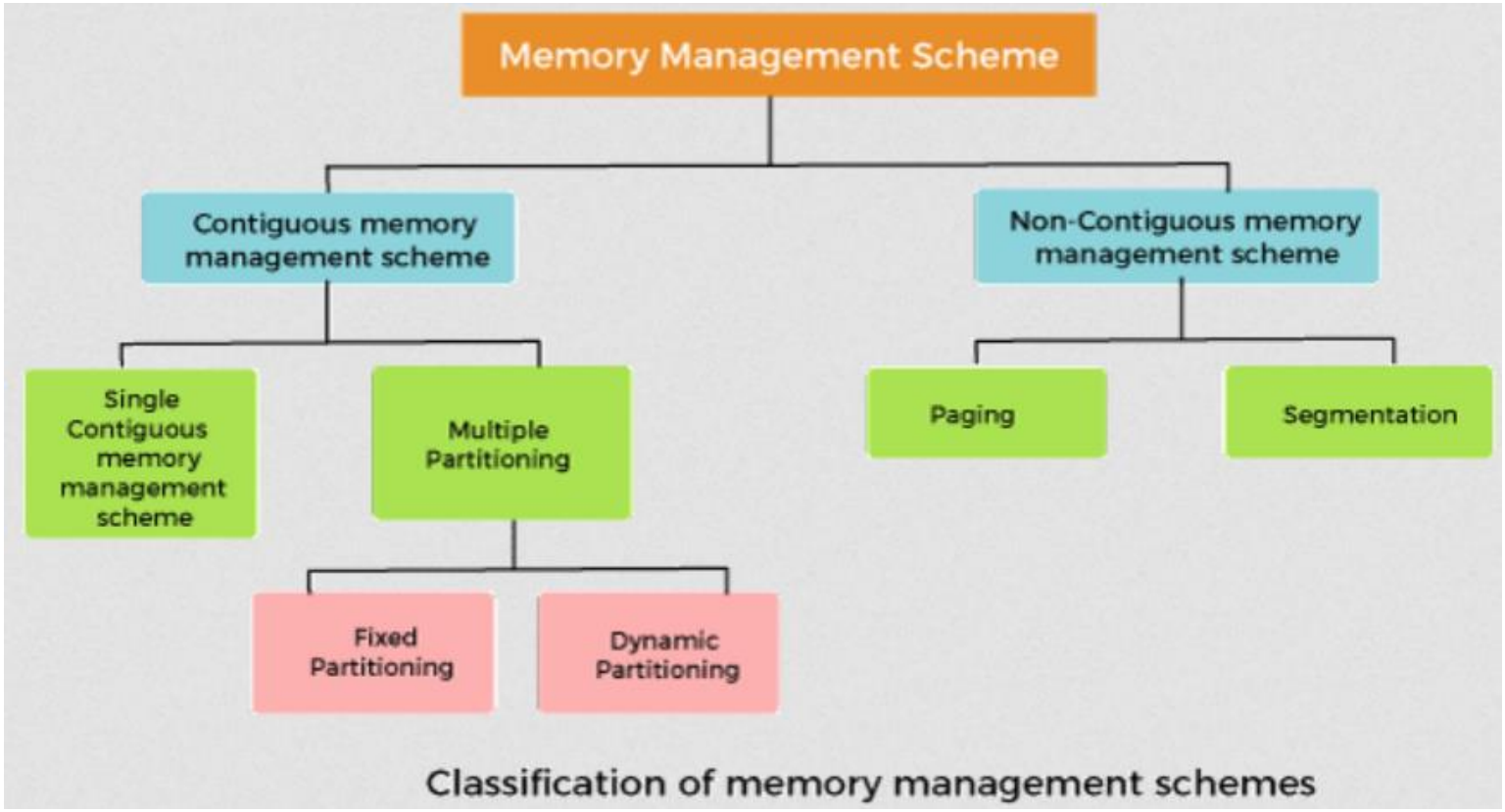
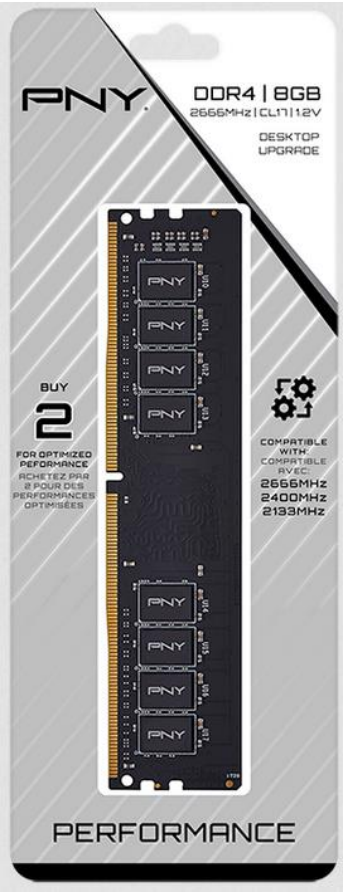
7. Operating system: memory control and I/O

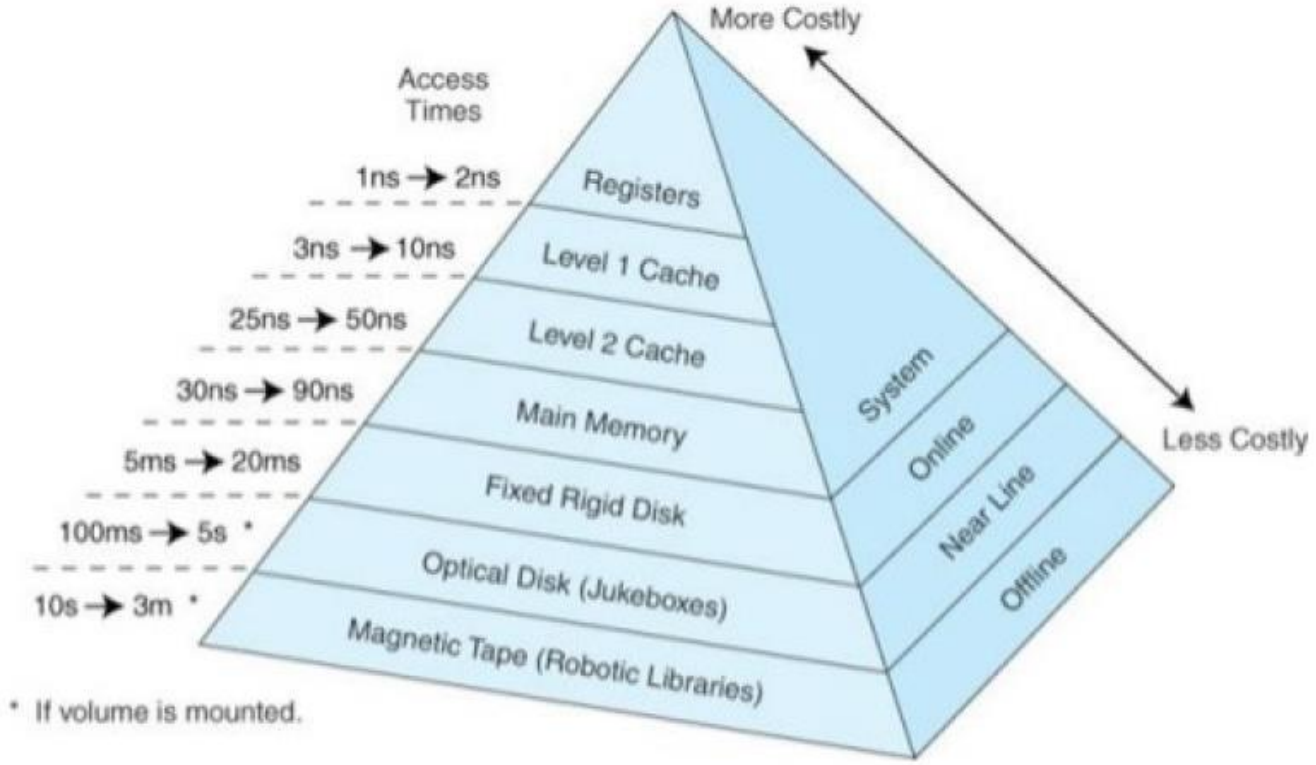


Classification of memory management schemes



Memory hierarchy

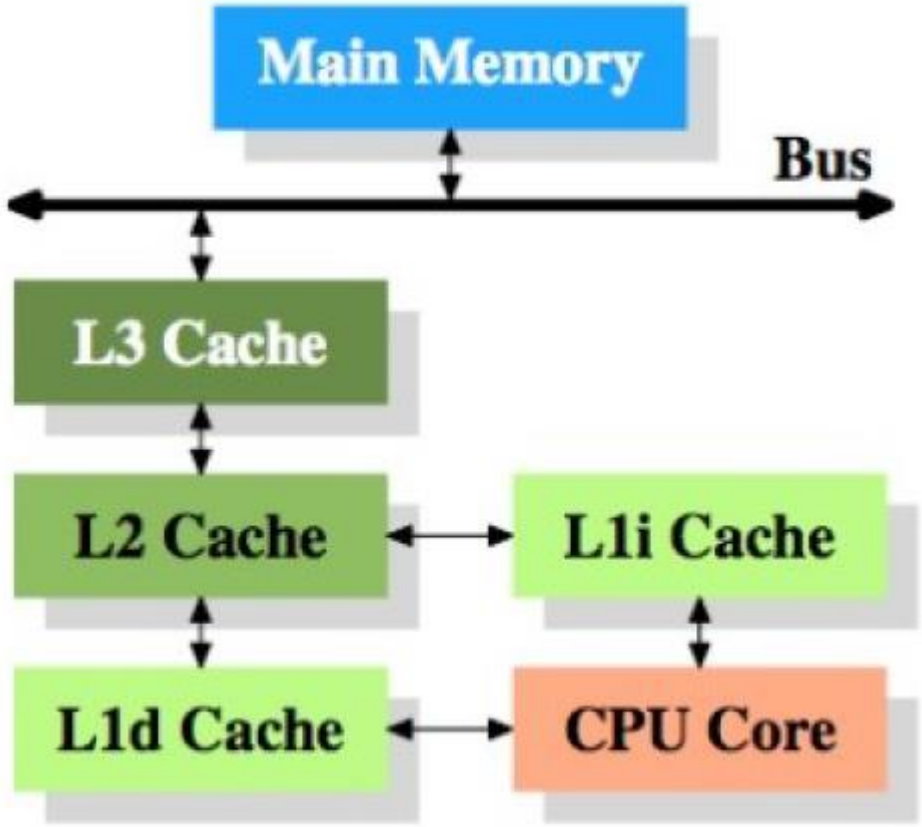
- There are 3 different types of memory in a computer:
  - CPU memory (registers & cache)
  - Main memory (RAM)
  - Auxiliary memory (disk/tape drives)
- The faster the memory, the more expensive it is
- Auxiliary memory can be located also offline outside the computer
  - Naturally, accessing this memory requires it to be mounted



Magnetic tape drives are not ancient history – they're still used today! IBM is a big manufacturer of tape libraries. Good read here: https://spectrum.ieee.org/why-the-future-of-data-storage-is-still-magnetic-tape

Caching

- Size of memory has positive correlation with access time
- For this reason, CPU cache has been divided to several levels (nowadays usually 3)
  - If the desired information is not found from the nearest cache, search continues to next one (and onwards to main memory, if needed)
  - Information that is needed most often is therefore stored nearest the CPU core
- For example, Intel core i7-12700:
  - L1 cache: 960 KB (80KB per core)
  - L2 cache: 15 MB (1.25 MB per core)
  - L3 cache: 25 MB (shared between cores)



Level 1 cache is, in some designs, divided to instruction (L1i) and data (L1d) parts.

Speed, latency and bandwidth

- When we're talking about memory performance (and especially main memory), three important concepts that often emerge are *speed*, *latency* and *bandwidth*
- Each data transfer (read or write) is performed on a clock cycle
- RAM "speed" = RAM frequency [MHz]; how many million data transfers the memory module is able to complete per second
  - In SDR (single data rate) memory, data is only transferred on one edge of the clock, so 1 transfer per clock cycle
  - In DDR (double data rate) memory, data is transferred on both edges of the clock, so 2 transfers per clock cycle
- Latency = response waiting time while moving the data [ms or ns]
- Bandwidth = how much data can be transferred per time unit [Mbits/s or MB/s]

Calculation of latency and bandwidth

- RAM modules are usually given latency values as CAS latency (CL)
  - CL = number of clock cycles it takes for the memory module to respond
- Actual latency (L) in [ns] can then be calculated using the latency equation:
  - Example: 2400 MHz DDR4, n = 2, CL15 → L = 12.5 ns

$$L = \frac{1000 \cdot n \cdot CL}{S}$$

CL = CAS latency
S = RAM speed [MHz]
n = Transfer factor (for SDR = 1, for DDR = 2)

- Theoretical maximum bandwidth B in [MB/s] similarly using the bandwidth equation:
  - Example: 2400 MHz DDR4, b = 64 bits/s, c = 4 → B = 76 800 MB/s

$$B = \frac{S \cdot b \cdot c}{8}$$

S = RAM speed [MHz]
b = Memory bus width [bits/s]
c = Number of channels supported by CPU

Memory control & fragmentation

- Alongside process scheduling, memory control is one of the most important tasks of an operating system
- Things to consider during this process include
  - Keeping track of the state of each memory location (is it free or not?)
  - Allocation of memory according to the memory requirements of the process
  - Protection (process can only access memory that has been allocated to it)
  - Efficient use of memory, so that processes don’t have to wait for memory in vain
- Inefficient use of memory is usually caused by fragmentation issues
  - Internal fragmentation = amount of memory allocated to a process is greater than required
  - External fragmentation = free memory consists of small slots which are unusable due to their size being too small to fit a process
  - External fragmentation can be ”cured” by rearranging the memory at time intervals (compaction); internal fragmentation, on the other hand, is harder to tackle

Memory addresses

- When a program is translated, compiler (or interpreter) generates the memory addresses for all data and instructions
- These addresses are called program addresses or logical (virtual) addresses
- When a process is prepared for execution in a computer, the data and instructions must be loaded to the main memory (in some addresses)
- These actual addresses are called memory addresses or physical addresses
- So, in order to execute the program, the OS has to:
  - Allocate a sufficient amount of memory to the process(es)
  - Convert the logical addresses to physical addresses
- This conversion can be done either in static or dynamic fashion

Static vs. dynamic conversion

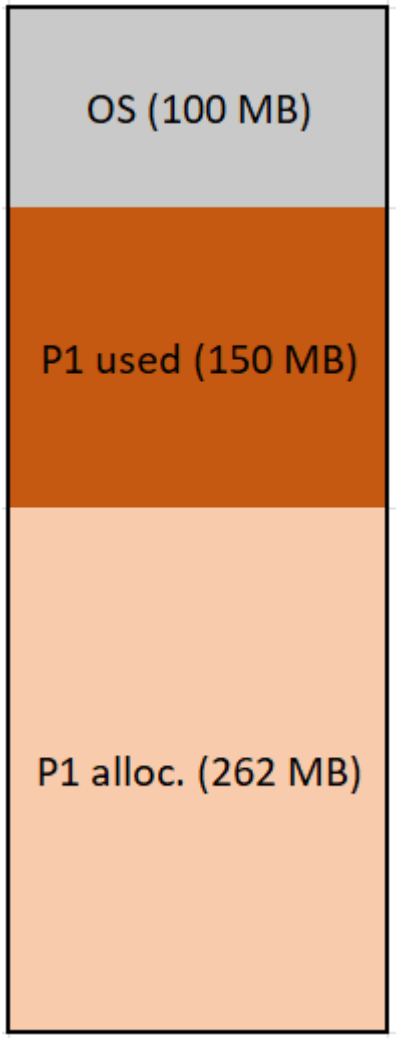
- In static address conversion, all data and instructions needed by the process must be loaded to physical memory at once
  - So, the amount of (free) memory in our system limits the number & size of processes
  - Address conversion is done by the loader before execution starts
- In dynamic address conversion the CPU (or MMU, to be exact) converts the program addresses to physical addresses during the execution
- Therefore, it enables us much more freedom with our memory usage:
  - Instructions can be loaded as the execution proceeds
  - Old instructions can be discarded and replaced with new ones
  - Amount of memory will not be a “hard” limit (although constant loading and discarding is slow)
- Dynamic conversion methods that enable this partial loading are *paging* and *segmentation*

Memory allocation techniques

- There are several techniques to implement memory allocation
- In the following slides, these techniques are demonstrated by using an example computer, that has 512 MB of main memory – of which 100 MB is reserved for the operating system – and a couple of processes running
- Note: do not confuse static/dynamic address conversion with static/dynamic memory allocation!
  - These share some similarities, but they don’t mean the same thing
  - Dynamic memory allocation means that we can change the size of memory areas that we allocate to processes
  - Dynamic memory allocation therefore enables us to eliminate (or at least minimize) internal fragmentation

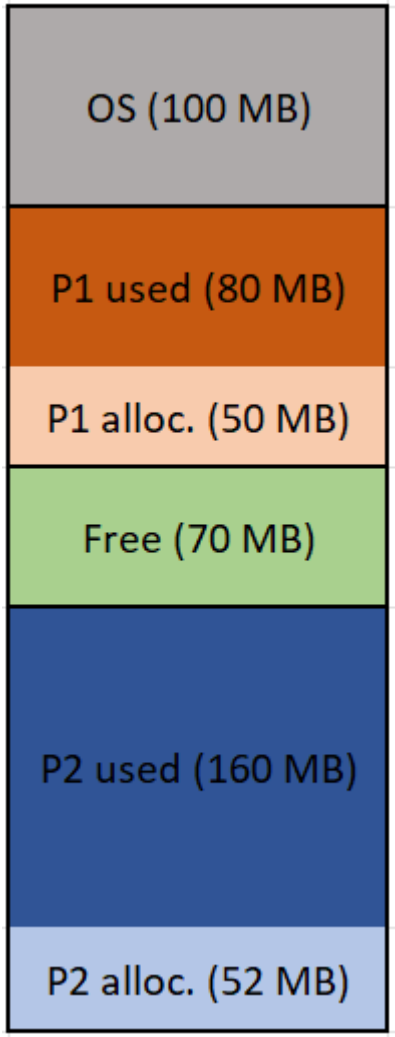
Single contiguous allocation

- All memory (that is not used by OS) is allocated to a single process, no matter how much it actually requires
- Very inefficient way of memory management
- No (external) fragmentation, though
- Also, very secure method
- Bad idea for computers that have to run several processes at the same time
- Plausible method for simple computers that do just one thing at a time (for example, embedded systems)



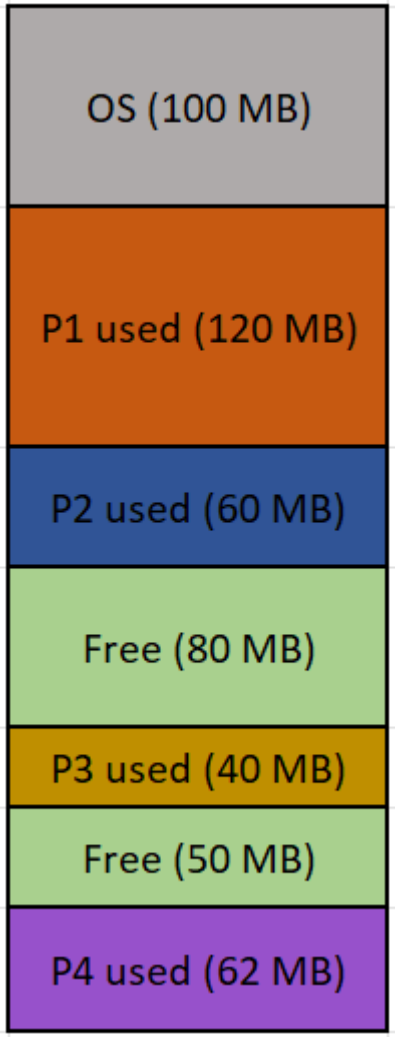
Partitioned contiguous allocation (static)

- Main memory is divided into fixed-size partitions
  - Example on the right: 4 partitions – 100, 130, 70 and 212 MB
- All memory in a single partition is allocated to a single process
- Number of partitions limits the number of processes that can be run simultaneously
- Better alternative, but not very flexible
- Both external and internal fragmentation
  - Example on the right: system could initiate a 3rd process (one free partition), but if the 3rd process requires more than 70 MB of memory, it can't fit in memory and it has to wait



Partitioned contiguous allocation (dynamic)

- Memory is divided in partitions, but the division is done dynamically – i.e. partitions are created "on the fly"
  - Size of partitions not fixed; process can be allocated a partition that matches exactly the amount that the process requires
  - Number of partitions is not limited in any way
- Free parts are called *holes*
- New processes are placed in holes, if the hole is large enough to fit the process
  - Leftover part of the hole forms a new (smaller) hole
  - Example on the right: if P5 which requires 65 MB of memory is initiated, it will be placed in the 80-MB hole → results in a 65-MB partition for P5 and a 15-MB hole
- External but no internal fragmentation

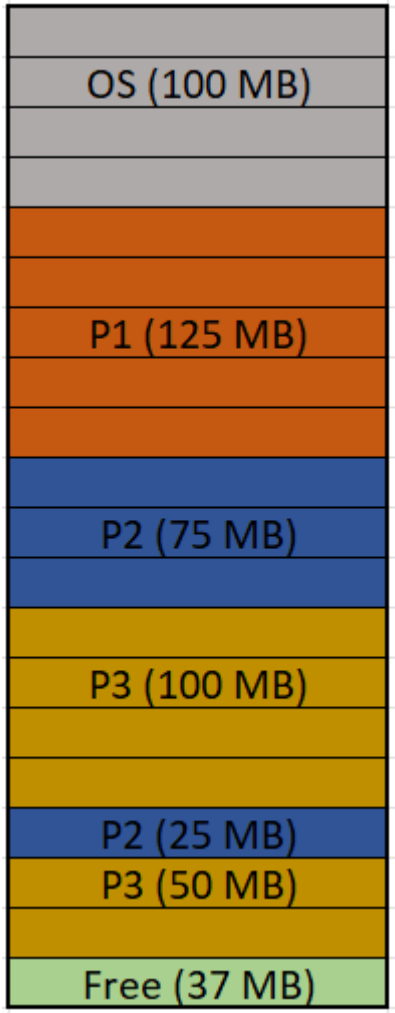


Hole fitting techniques

- If there are multiple holes, where should we place the next process?
  - Several techniques for hole selection
- First fit = start search from beginning, place the process in the first hole that is large enough
- Next fit = as previous, but when the next process arrives, we continue hole search from the point of last allocation (instead of starting from beginning again)
- Best fit = search the whole memory for holes, place the process in the smallest hole that is large enough
  - Produces the smallest leftover hole
- Worst fit = search the whole memory for holes, place the process in the largest hole
  - Produces the largest leftover hole

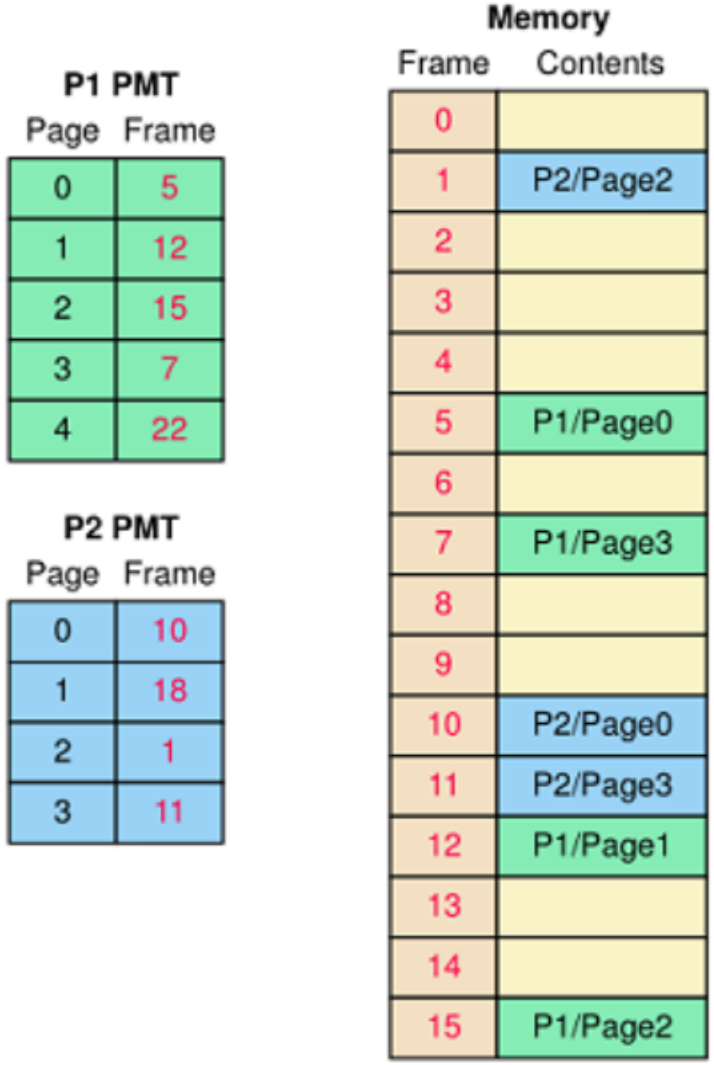
Paging

- In order to eliminate external fragmentation, we have to get rid of the contiguousness constraint
- In paging, the program memory is divided into fixed-size parts called pages, and the physical memory is divided into same-size parts called frames
- This allows the program memory to be contiguous, but the information can be stored anywhere in physical memory – in as many parts as needed
- A common page size is 4 KB = 4096 B
- Internal fragmentation remains a (limited) problem
  - Last page of the program is left partially empty (example: if the program size is 9000 B, it uses 3 pages – last page is only 808 B)



Page table

- Computer naturally needs to be able to link pages and frames together
- This is done via a *page table*, that is maintained by OS
- Page table contains information about all pages:
  - If the page is in main memory, what frame is it linked to
  - If the page is not yet in main memory, where is it located on hard disk (so that it can be fetched to main memory)
  - Possible additional Y/N fields (read-only, modified, etc.)
- Decreasing page size reduces internal fragmentation, but increases the number of pages → increases the size of the page table
  - Therefore, page size choice is always a compromise

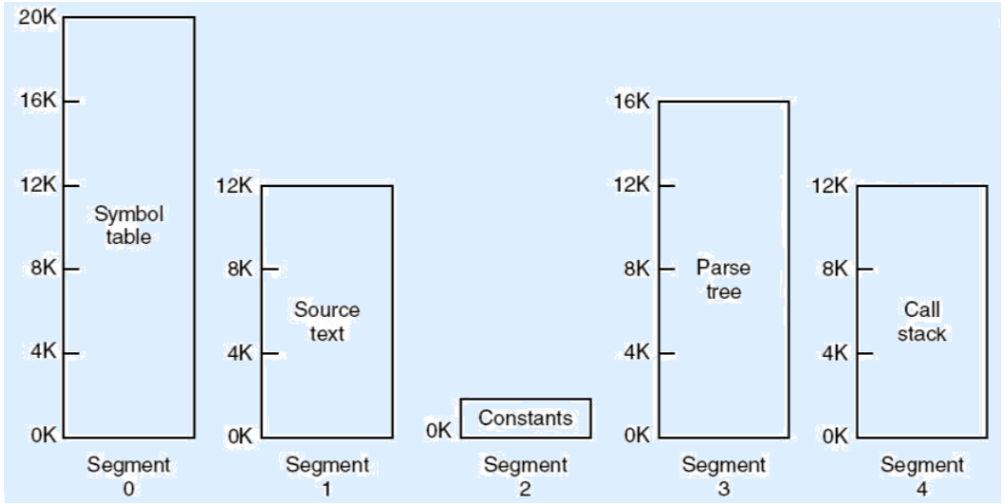


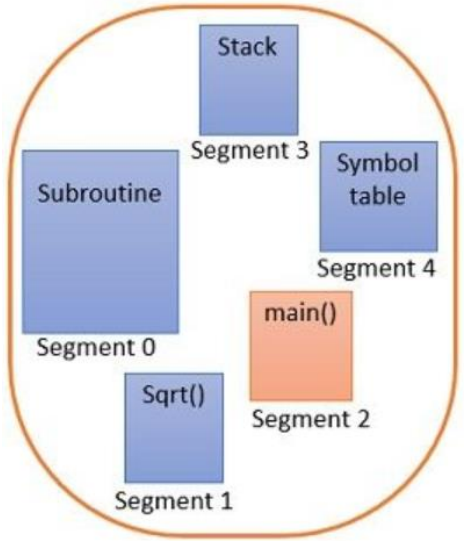
Segmentation

- Programs are seldom just one entity, but they comprise of several elements:
  - Main program, shared libraries, data stack etc.
  - Size of these elements may change during the execution
- Especially if the memory address conversion is done in dynamic fashion, this causes problems: the main program can easily be loaded to main memory “little by little”, but the shared libraries and data stack are used constantly
  - Paging results in all these elements getting mixed up in different pages, so there may be a need to load and then re-load the same pages several times
- Segmentation aims to solve this issue by creating segments (of variable size) where all these elements are stored (one element per segment)
  - Segments that have shared libraries can be kept in memory all the time
- Eliminates internal fragmentation, but external remains a bit of a problem
- Solution: combined segmentation and paging

Segmentation

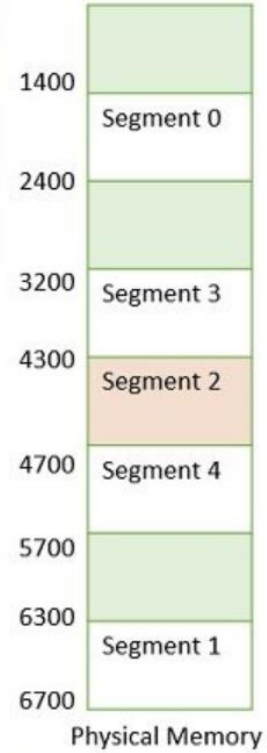
- Segmentation allows each table (segment) to grow or shrink independently
- Segmentation is the only memory management method where even the program memory is not contiguous
- Information of physical address starting point (*Base*) as well as size of segment (*length* or *limit*) are stored in *segment table*
  - Memory address = Base + offset





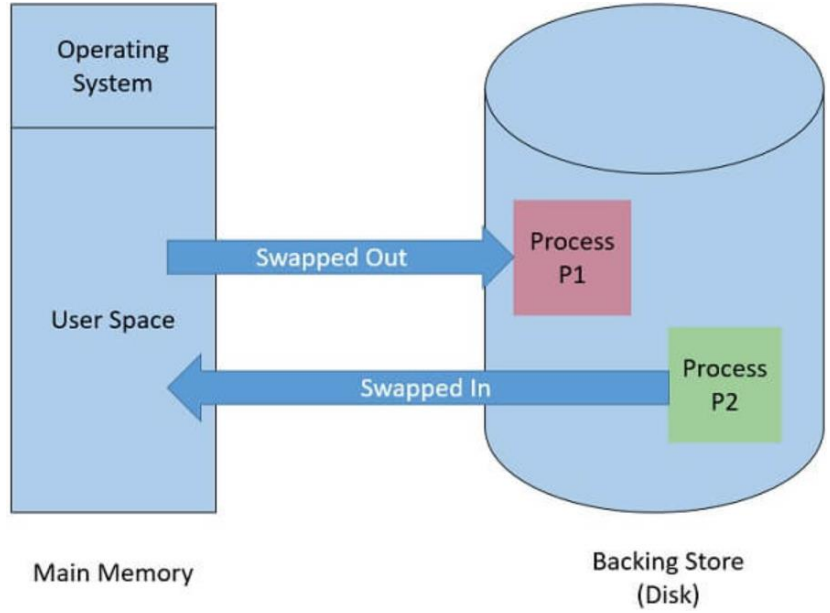
	limit	Base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

Segment Table



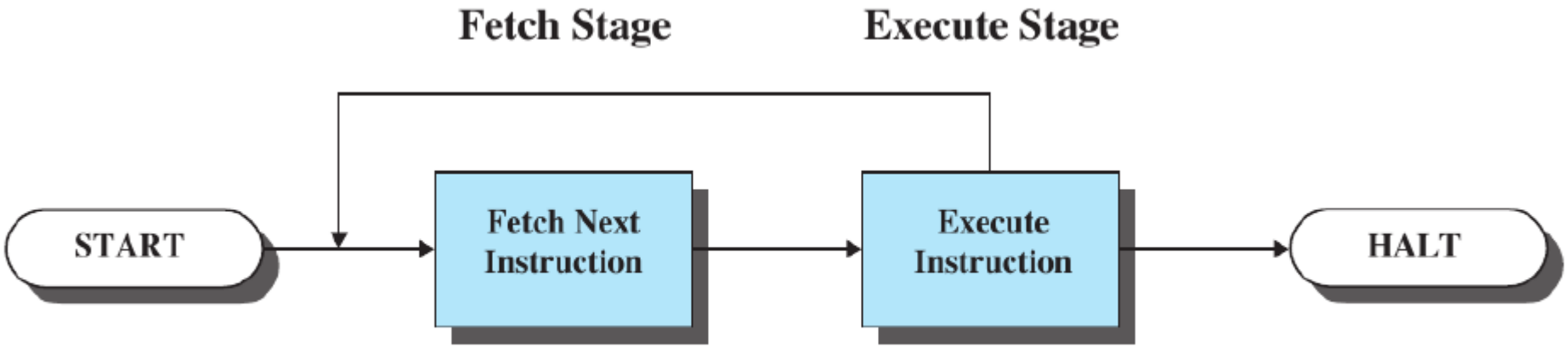
Swapping

- If we don't have enough space in our (physical) memory for the process we'd need to execute, we can swap out information that is not needed right now to hard disk
- This frees us main memory space so that the information needed by the executing process can be swapped in
- Swapped-out processes are usually ones that have performed a (probably lengthy) I/O request and hence are in waiting state
- Multiple possible criteria for swap-out process selection



Simple process execution cycle

- CPU fetches an instruction from a memory address given by the program counter
- Program counter value is increased and needed operands fetched
- Instruction is executed and the result is placed in memory or register
- Continue to the next instruction

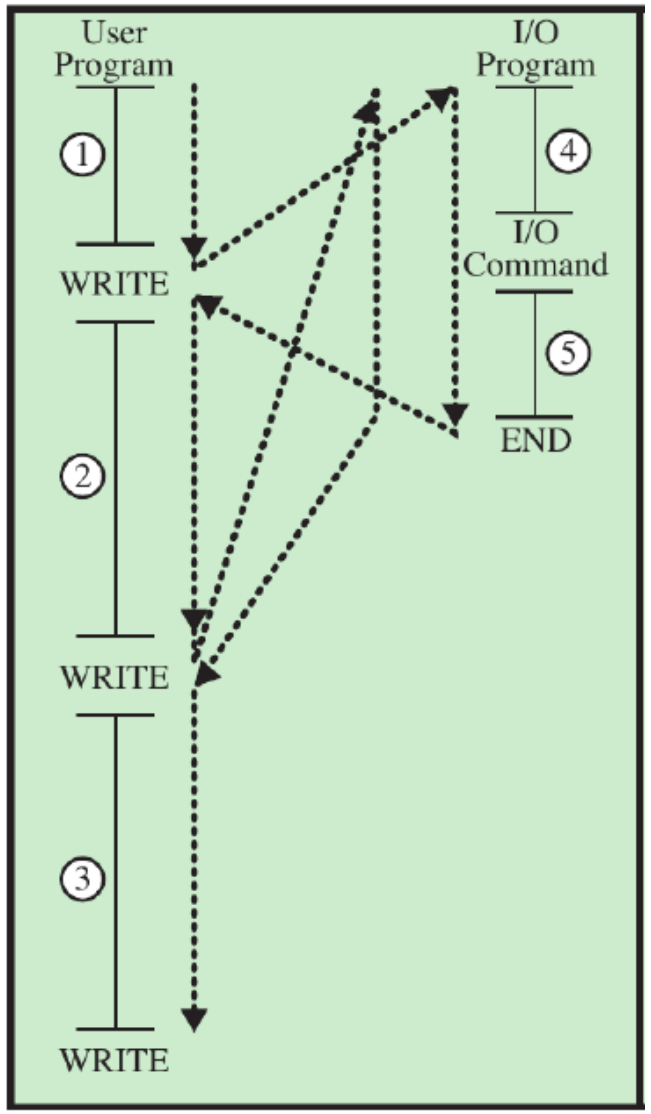


Stallings: Operating Systems, 9e, Prentice Hall

I/O in a simple program

- In a simple non-interrupting cycle, I/O works like this:
  - User program encounters “write” command and makes an IPC request that gives control to an I/O program (1)
  - I/O program initiates the I/O process and gives the I/O command (4)
  - I/O program waits (polls) that the I/O operation is complete (5)
  - I/O program performs the end tasks and returns the solution information about its state (2)
  - Process continues until another “write” command surfaces – then the previous steps (4) and (5) will be repeated

- The CPU waits a “long” time in phases (4) and (5)!!
  - Inefficient use of CPU resource



Interrupts

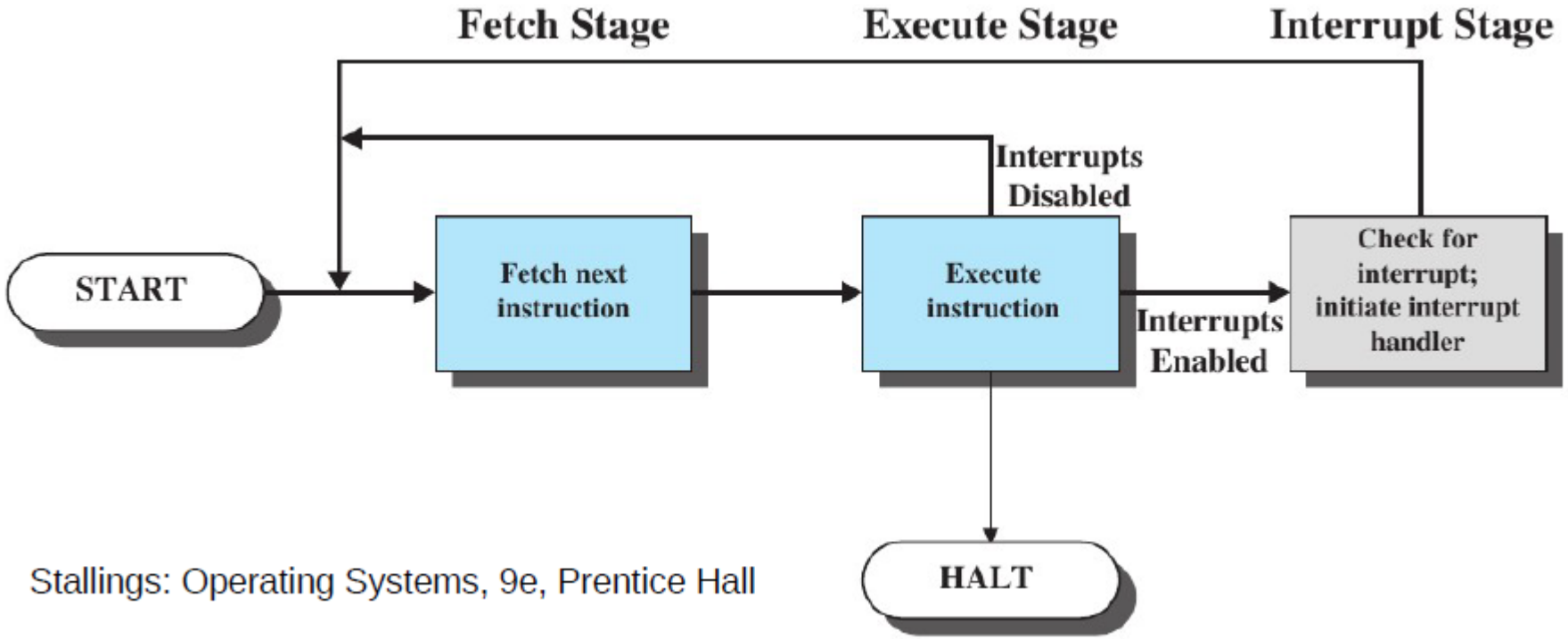
- Unlike in real life, in computers interrupts are a “good” thing: they enable the simultaneous use of CPU and I/O programs
- Data transfer is slow (especially if data is fetched from hard drive) and the original process often can’t continue until it’s done
  - The goal is to execute other process(es) meanwhile waiting for I/O to finish
- CPU only initiates data transfer – after that it’s free to continue execution of other processes (no CPU idle time)
- When the I/O is completed, the I/O program interrupts the CPU
- CPU then starts interrupt handling
- When the handling is done, the process that was waiting for I/O completion can be taken back to execution

I/O control methods

- Programmed I/O
  - No interrupts from the I/O program; CPU checks whether I/O is done
  - CPU must wait in idle for I/O completion
- Interrupt-driven I/O
  - When an I/O event is started by process 1, the CPU continues to execute other processes
  - When I/O has been completed, the I/O program interrupts the execution of other processes and saves the data of the process currently in execution
  - Data from I/O is loaded to the CPU
- Direct memory access (DMA)
  - Like interrupt-driven I/O, but when the I/O has been completed, the I/O program transfers the data directly to the main memory (without going through CPU)
  - Nowadays the most common method due to effectiveness

Interrupting program execution cycle

- If our control method enables interrupts, CPU checks for interrupts before fetching the next instruction
- If there's an interrupt in the interrupt register, interrupt handler is initiated



Stallings: Operating Systems, 9e, Prentice Hall

Classification of interrupts

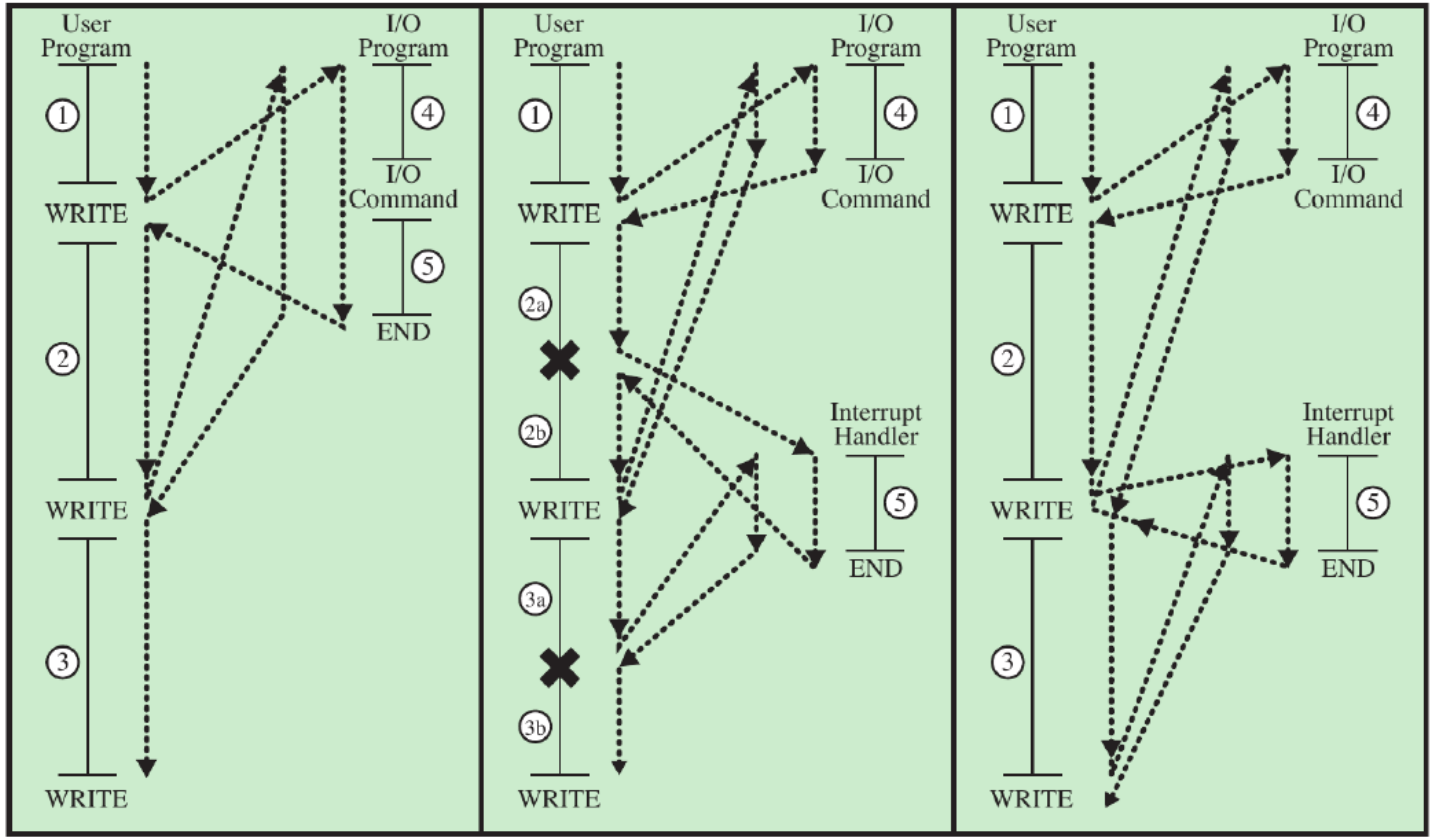
- Program interrupt (normal or exceptional)
  - Normal: generated by a program sending an IPC or by user request
  - Exceptional: generated by some condition that occurs as a result of an instruction execution, such as arithmetic overflow/division by zero/illegal machine instruction/etc.
- Timer interrupt
  - Generated by a timer within the CPU; allows the OS to perform functions on a regular basis
  - Time-slice based scheduling, timed processes (f. ex. Check for updates or viruses)
- I/O interrupt
  - Generated by an I/O controller, to signal normal completion of an operation OR to signal a variety of error conditions
- Hardware interrupt
  - Generated by hardware input or a failure (power failure, memory parity error, etc.)
  - Maskable (can wait if higher priority interrupt occurs) or non-maskable (must be handled ASAP)

Interrupt handler

- Interrupt handler is a process ran in kernel mode in OS
- Initiated every time an interrupt occurs
- Finds out the reason for interrupt (classification)
- Initiates actions for handling the situation
  - Desired interrupt (normal program/timer/I/O) → switch the process
  - Undesired interrupt → what went wrong, how to recover?
- In order to enable continuation of the interrupted process later, its registry values must be stored:
  - Program counter, program status word & other registers used by the process
- Note! Interrupt handling can be interrupted, too – this risk must be addressed
  - For example, a hardware failure during file write leads to a file corruption hazard

Short I/O wait vs. long I/O wait

- Short I/O wait = the interrupt is handled right after it occurs
- Long I/O wait = the interrupt is handled when the next I/O request arrives
- When long I/O wait is used, there can only be 1 I/O request in line at a time
- "Microwave example"



Stallings: Operating Systems, 9e, Prentice Hall

Thank you for listening!