OPERATING SYSTEMS
CSC3002F

Textbook: Operating System Concepts
Chapter exercises: https://codex.cs.yale.edu/avi/os-book/OS10/index.html

Introduction

- OS = program
  - Run when PC booted
  - Runs all other programs
- Examples: Windows written mostly in C and assembly; MAC also C
- Written in low lvl languages which can directly interface with computer's hw
  - Done so that OS can start and run other programs – these are systems level apps and user programs.

OS Specifics

- OS: program that lets you run other programs
- A program providing controlled access to resources
  - CPU
  - Memory
  - Devices – display, keyboard, mouse
  - Persistent storage
  - Network
- Uses this access to give high lvl apps indirect access to these resources

System Software

- Broadly OS is example of systems level sw
- Other examples of sys lvl sw:
  - C compiler and library functions
  - Dtb management system – provides high lvl scripting and querying languages that interact with underlying dtb
- OS is sw that operates directly with system hw
  - Controls and coordinates use of computer hardware among various apps and user programs

OS as Illusionist Government

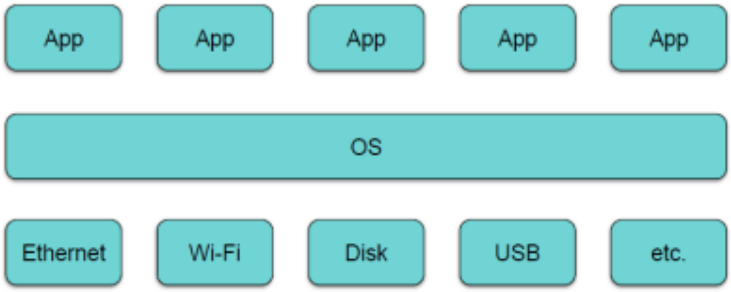
- OS as an illusionist – clever resource management to overcome hw limitations
  - Divvies up resources to handle user requests without user noticing
  - For example, at any one time, the OS will be running many programs, all requiring memory to run.
    - With limited physical memory available the OS has to carefully prioritize and manage the running of programs
- OS as a government
  - Protects users from each other – so that they don't overwrite/interfere with each other's programs
  - Allocate resources efficiently and fairly
  - Provides an env within which other progs can do useful things

- Overall key function of an OS is to provide a computational environment within which other programs can do useful work

OS Systems Level View

- From a computer-systems (Computer system = hardware + software + data) pov, the OS is the program most closely involved with the computer's hardware
  - There are two different OS views from a systems perspective
- OS is a **resource allocator**:
  - Manages all resources e.g. hw
  - Handles conflicting requests for efficient and fair resource use
- OS is a **control program**:
  - Controls execution of programs to prevent errors, ensures programs run

OS is an Abstraction



- OS can also be viewed as an abstraction of the workings of a computer system
  - OS is an interface between users and hardware and as such is a computer system environment architecture
- Allows convenient usage of its app and user programs - hides all the tedious stuff such as memory access and allocation for running programs
- Efficiency and Control:
  - Allocates resources, allows parallel applications, protects information
- Diagrams shows system lvl and user programs running on top of OS
  - OS takes request from user programs and allocates resources to them

Reality VS Abstraction

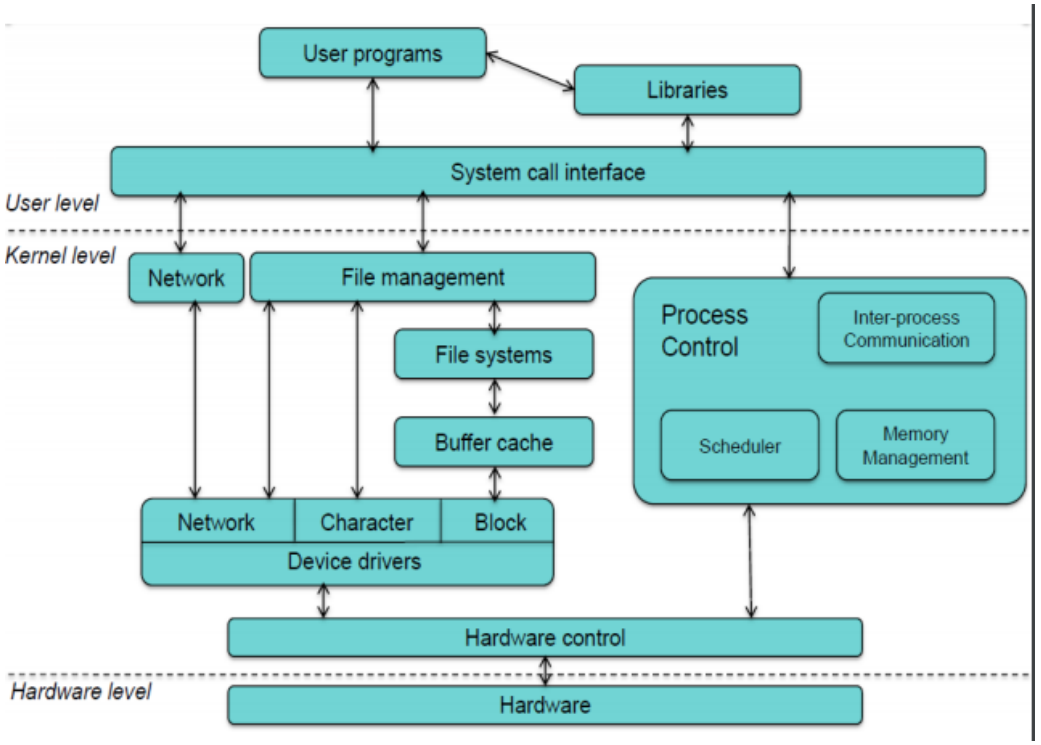
Reality	***Abstraction***
A single CPU	Multiple CPUs
Limited RAM capacity	Infinite capacity
Mechanical disk	Memory speed access
Insecure and unreliable networks	Reliable and secure
Many physical machines	A single machine

- Reality column = what is in a given computer system

- Abstraction column = what the OS should ideally be doing – gives view on running of the OS such that it has much more than reality
    - Infinite capacity – users should be able to run as many programs as they want
    - Single machine – don't have to worry about resources e.g. files being transferred from other parts of the nw

OS Structure

- High Lvl view of OS Structure:



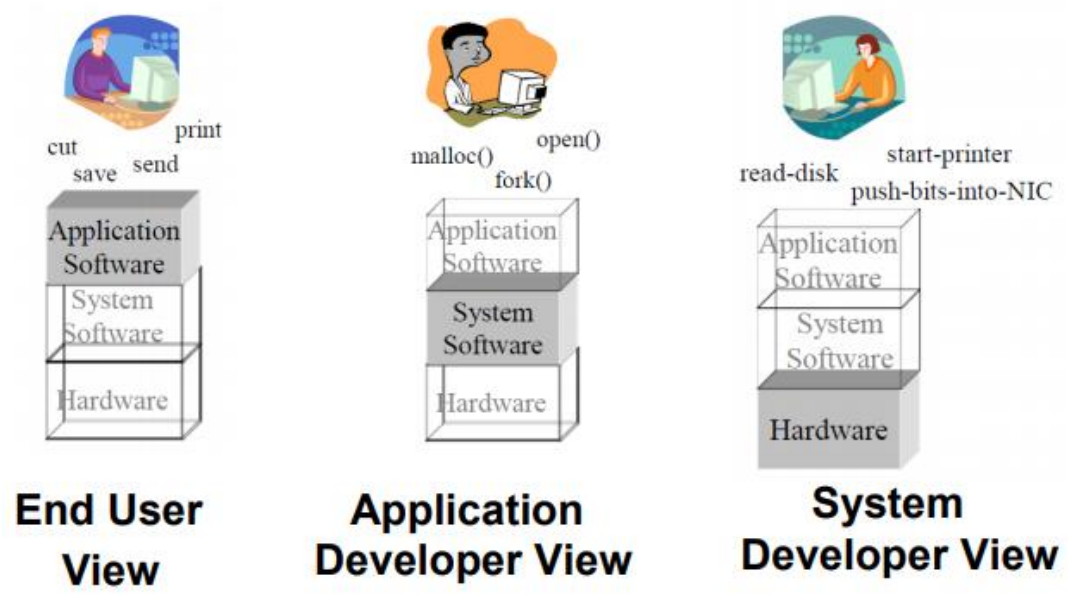
- Main point of this figure is to show that an OS is complicated, so we need abstractions in order for it to be readily usable by any user
- The structure of the OS is divided into three parts: user level, kernel level and hardware level.
- *User level*: part of the OS that the user directly interacts with
    - Consists of programs that the user creates or imports from other external media and devices (user programs), and libraries (created by the user or others) that these user programs need to run, as well as system application programs (system call interface) that are part of the OS and needed for its basic operations (e.g. disk management applications)
    - When compile and run programs, link libraries to them and run using calls to system level apps in the OS
- *Kernel level*: core of the OS that is kept resident in main (physical) memory at all times
    - Kernel ensures that essential operations of the OS are kept running such as file management and storage in secondary memory (hard-disk) storage and process control (scheduling and executing programs) to run in main memory and to allocate each process appropriate computational resources (CPU time).
    - Also runs device drivers that enable the OS to interact with the hardware of the computer system

    - e.g. writing to and reading from a physical hard-disk drive
- *Hardware level*: all the physical devices that the OS uses such as external hard-drives, CPUs, monitors, keyboards and other peripherals (i.e. all hardware that typically comprises any computer system)

Why have abstractions?

- Reduce functional complexity – user doesn't have to worry about low lvl hw access or divvying up resources
- Provide single abstraction over multiple devices – can have multiple machines in a nw involved but user only has to worry about the single machine
- Resource sharing: i.e. special functions can be created that abstract over multiple resources and devices in the OS, in order to make the resource use **fair, efficient and secure**.
    - Efficiency
    - Fairness – all user programs given roughly equal share of CPU and memory time
    - Security – user programs can't interfere with each other

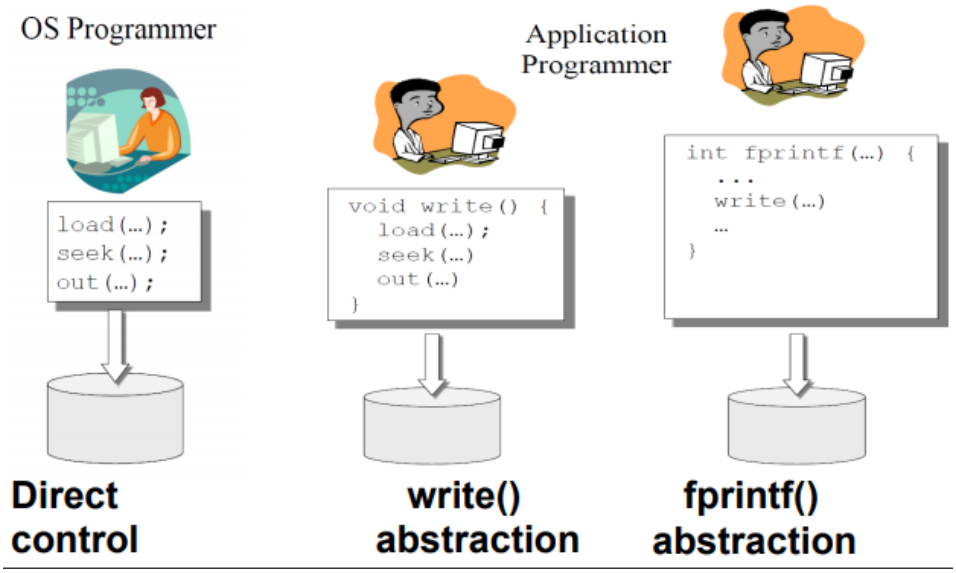
User Abstractions



- In terms of users working with an OS, there are multiple levels of abstraction for how one may interact with the OS
- End user of an OS interacts only with application software (user programs)
    - Uses high level functionality – uses pre-defined functions and methods in the context of other programming languages
    - User doesn't need to worry about how functions are operating at a lower level
    - App level sw running on top of sys level sw and hw
- App level dev working with sys lvl sw
    - Interacts with the OS specific application programs that directly affect the running of the OS
    - Working with sw that directly interacts with hw beneath – lower lvl functions
    - Working with program functionality such that they can directly interface with hw – must know more about how hw works, as well as compile and write sys lvl sw

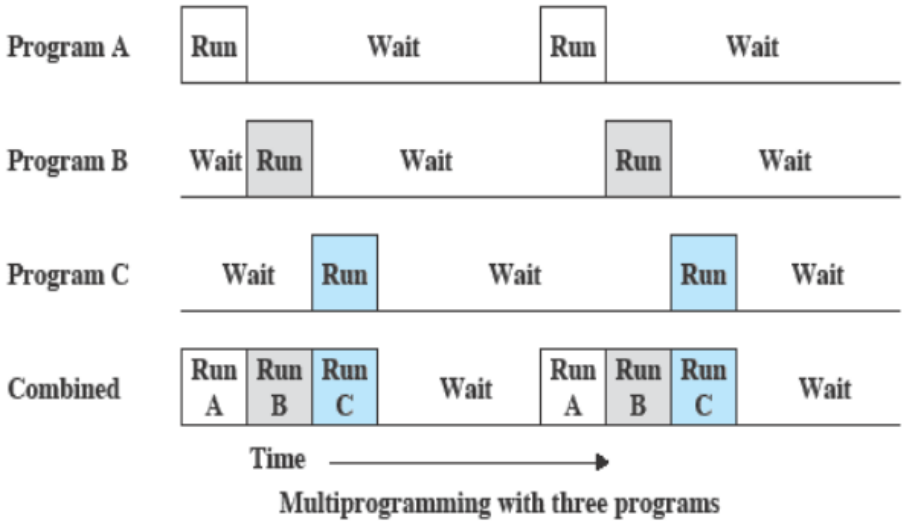
    - Dev in this case must know something about the underlyng hardware and how to interface application software directly with this computer system hardware
- System Dev view
    - Someone who works directly with hw – control and design for e.g. hard disks and CPUS
    - In diagram – dev is working with machine code, needs to know how hw works and make adjustments at the fundamental lvl (e.g. write from/to physical disk)
        - User is working directly with the hardware and is writing hardware controllers that can be invoked by higher-level OS applications, in order to perform basic hardware functions (e.g. write to physical memory or read from physical memory)
    - Lowest level and most basic of OS operations but are essential in order for the OS to work at all

Disk Abstractions



- Different levels of abstraction for OS developers working with disk operations
- Lowest lvl of abstraction – OS programmer
    - Can work directly with the disk controller that controls basic data retrieval and manipulation operations on a physical disk
    - Programmer must know how disk works fundamentally, and what all functions do ito how they manipulate data directly on the physical hard disk
- Middle app programmer
    - Can write their own disk-operation functions and programs, that encapsulate all the lowest level data retrieval and manipulation operations of the disk controller
    - Such functions and programs typically operate as part of OS application level software
- Rightmost app programmer
    - Uses already written disk manipulation functions to create even higher-level functions and programs that are used by programming languages
    - E.g. fprintf() abstraction for printing strings is given, which encapsulates the lower level write function, which prints individual characters

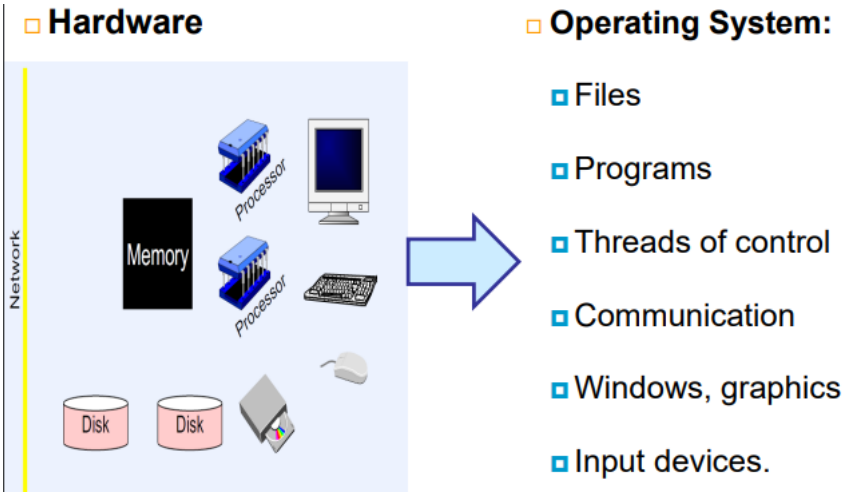
Process Abstractions



Multiprogramming with three programs

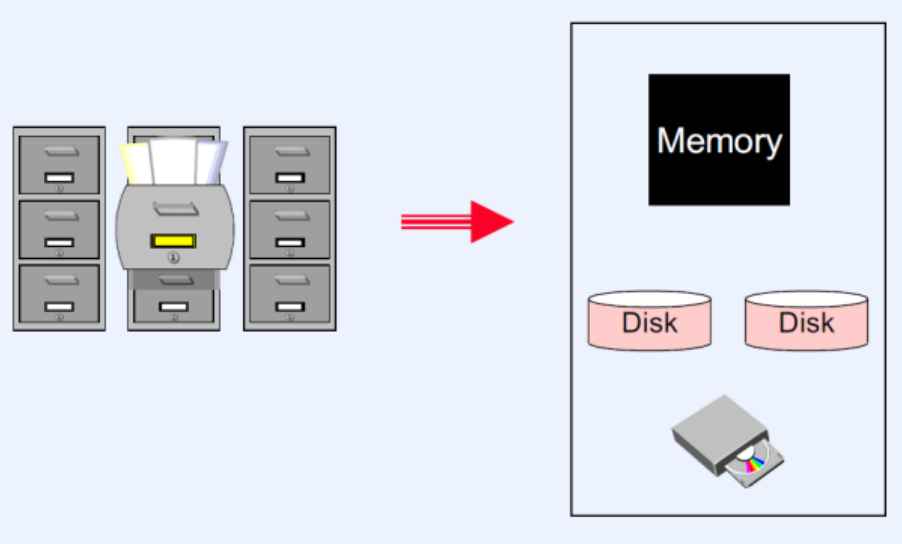
- Process control ito abstractions
- In the context of an OS, a process is the executing (running) of a program loaded into physical memory and run for some number of CPU processing cycles
- Diagram shows programs A, B, C running
    - Program has run-wait cycle – since there's limited physical memory and CPU processing time, so some programs must wait while another executes
    - Shows resource allocation done by OS – infeasible to have one program just run and finish
        - Must have switching between running programs – multiprogramming with 3 programs
        - Abstraction is multi-programming (process switching) over all concurrently running processes
        - Rapid switching between the running of programs and keeping all programs partially loaded into memory, allows the CPU to execute a small slice of program A, and then programs B and C, and then back to program A
        - Gives the user the impression that the OS is smoothly running all programs in parallel, when in reality this is not possible (i.e. there is just one CPU and limited memory)

Operating Systems Abstractions



- Abstraction is an important concept in OS design and execution, and there are various forms of abstraction.
    - Changing the level of abstraction, changes the level of detail and knowledge required of a user (or application developer) when using or designing an OS
- E.g. common OS abstractions are those over files and programs
    - File abstraction allows users of a OS easy and convenient use of file manipulation and storage, such as read and write operations of files between different directories, without having to know or worry about how basic read and write operations required to store and manipulate data on a physical disk
- There are many other abstractions used by the OS to hide unnecessary detail from the user
    - E.g. Threads and communication (e.g. communication between processes that are executing programs to ensure that one user program does not over-write another in memory)

Files

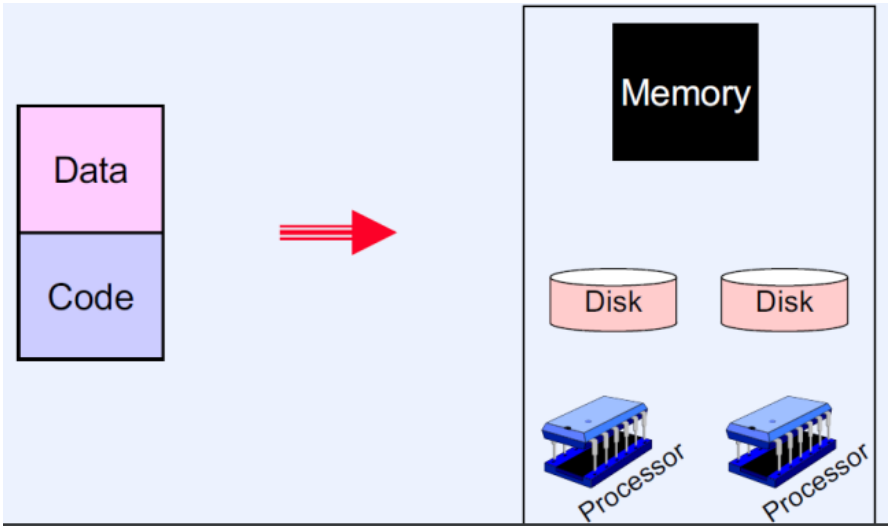


- Files as an abstraction of memory
- Specifically, the files that any OS user is familiar with, and manipulates with operations such as copy and paste, are simply an abstraction over lower-level data-manipulation that occurs using some basic (hardware-level) operations on physical memory
- Diagram shows files
    - These files get manipulated
    - Have high lvl cmds in OS – these are just an abstraction of underlying physical hw
        - Minute, low lvl manipulation of physical memory on the hard disk and other storage media

Issues with files

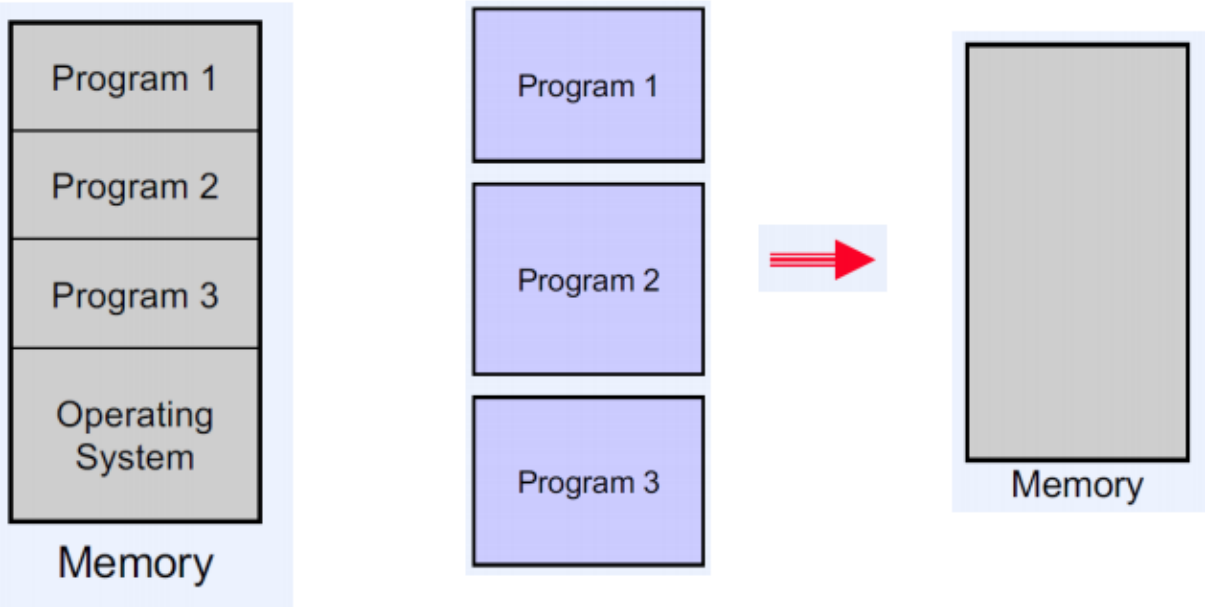
- While the abstractions of files used by an OS makes file use convenient and easy for the user, it comes with many problems at lower levels of abstraction that must be solved by OS developers
- Naming
    - OS must allocate unique names and addresses in physical memory space to files, so as two different files are not written to the same memory space and create a conflict
    - Allows files to be recalled quickly and efficiently
- Allocating space on disk (permanent storage)
    - How to organise for fast access and minimise waste
    - As the number and size of files increases over time with OS use, there will invariably be many more files than is physical memory space to store all the files
    - At the lowest levels of abstraction (physical memory and the OS app system software that operates directly on the physical memory), the OS must have some way of being able to effectively and efficiently store all files
- How to shuffle data between disk and memory (high-speed temporary storage)?
- How to handle crashes and (disk / memory) errors?
- Virtual memory is used to deal with most of these issues

Programs



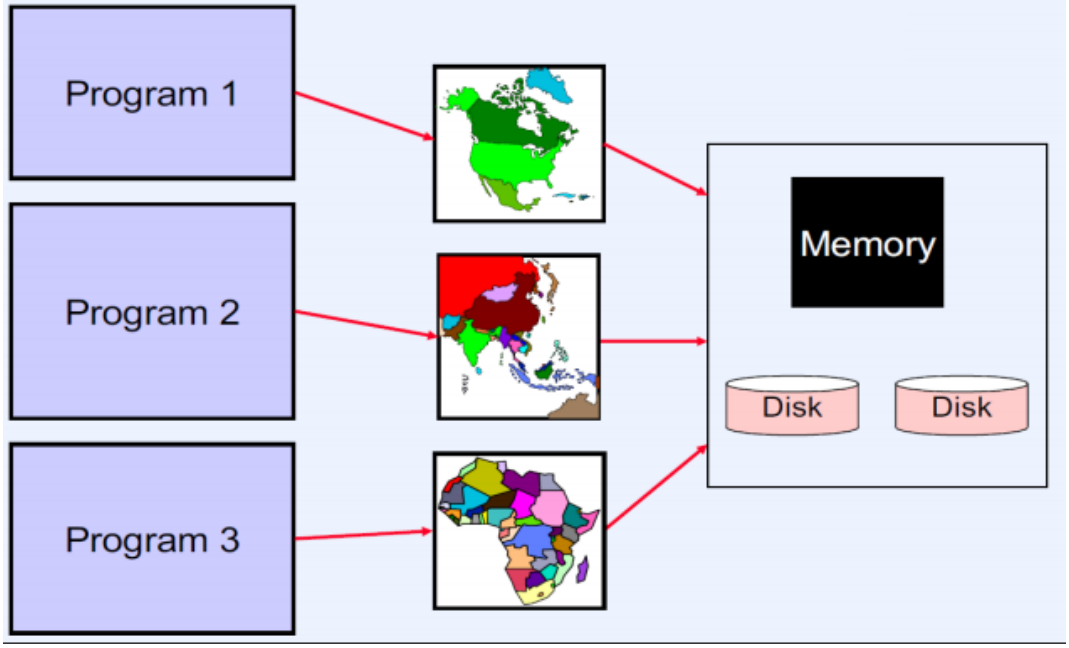
- There is an abstraction between a program code (e.g. algorithmic instructions) and data that the code uses (e.g. files of text)
- Such programs must be compiled and executed, transferred from permanent memory (e.g. an internal HD) to volatile temporary memory (RAM)
    - Typical prg has source code that gets compiled and data that it uses – but underlying mechanisms is that it's stored in some memory, using some processor, CPU time
        - But all this is hidden from the user
- The function of a program is an abstraction over the underlying hardware layers
    - Workings underneath are hidden
    - E.g. below the disc operations (for storage) is how the CPU architecture is directing its hardware components (e.g. disks) to operate when a file is named
        - This file could be a program which when compiled, in turn creates a bunch of other files with names which calls for further data-manipulation (e.g. read, write) operations on the hard-disk and RAM

Memory sharing



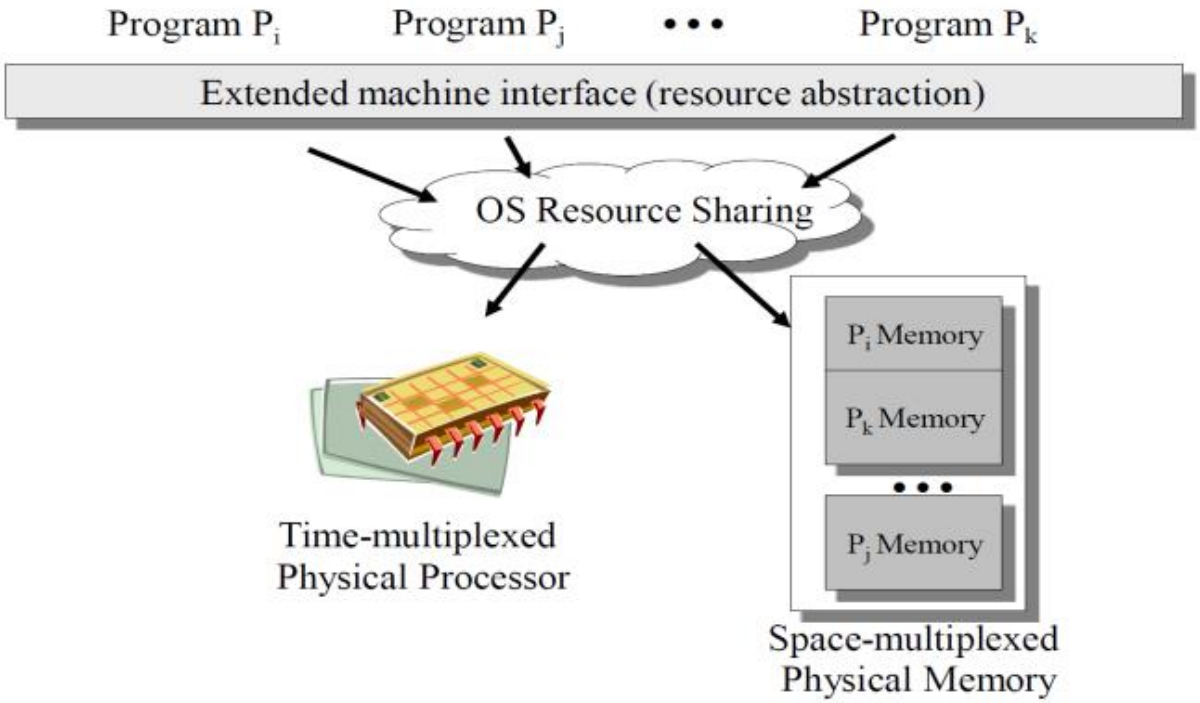
- Have many programs concurrently – with physical mem that they all need to run on
  - But there is only so much RAM and secondary storage
  - This is where OS as resource allocator comes into play
- Diagram:
  - LHS shows the ideal scenario for memory sharing in an OS – all programs as well as OS are concurrently stored in memory
  - RHS shows the reality of practical OS operation - there is limited physical memory and this memory must be shared between all programs currently running on the OS
- CPU can load instructions only from memory, so any programs to run must be stored there
  - Personal computers run most of their programs from rewritable memory, called main memory (RAM)
- Ideally, we want the programs and data to reside in main memory permanently, but this arrangement usually is not possible for two reasons:
  - (1) RAM is usually too small to store all needed programs and data permanently
  - (2) RAM is a volatile storage device that loses its contents when power is turned of
- Thus, most computer systems provide secondary storage (e.g. magnetic hard-disk drives) as an extension of main memory
  - The main requirement for secondary storage is that it be able to hold *large quantities of data permanently*
  - Secondary-storage devices provides storage for both programs and data, where most programs (system and application) are stored on a disk until they are loaded into memory
    - This allows programs to be shared in main memory (via being swapped in and out of secondary storage) and hence for the OS to concurrently run multiple programs with limited main memory

Virtual Memory



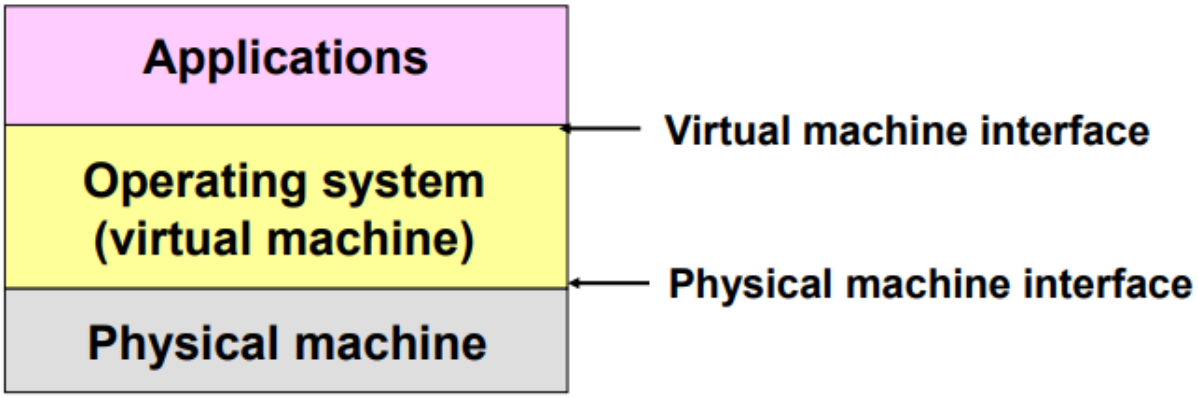
- Virtual memory allows the execution of programs (processes) that are not completely in memory
- Diagram:
    - LHS has programs 1, 2, and 3 point to continents (indicating large memory hungry tasks)
    - RHS has main memory (RAM) and secondary storage (hard-drives) are shown as the means to enable the concurrent running of multiple memory-hungry programs
- Virtual memory allows programs to use **more memory than is available in main (physical) memory**, via abstracting main memory into an extremely large, uniform memory array.
    - Array uses secondary storage to temporarily hold programs that must eventually be transferred to main memory for execution
    - VM emulates an infinite amount of main memory by using secondary storage – idea is that prgs 1 to n are assigned temp storage in secondary storage, then quickly swopped into main memory to do some processing, then swopped out again
        - OS is responsible for this – gives illusion that programs are running concurrently and smoothly in main memory
- Virtual memory also allows programs (processes) to share files (data) easily and to implement shared memory
- Using virtual memory, a program is no longer constrained by the amount of physical memory that is available and users are able to write programs for an extremely large virtual address space, simplifying their programming tasks
    - With the use of virtual memory, each user program takes less physical memory, more programs can be run at the same time
    - Corresponding increase in CPU use and program throughput (successful execution), but with no increase in program run-time

Resource Sharing



- OS important task – to manage and allocate resources so as they are *effectively and efficiently shared* across all programs (processes) and users of the OS
- Resource sharing is made possible by virtual memory and is a memory management function of the OS
    - Running N programs is an abstraction over physical memory and processor
        - i.e. program code and data stored in main memory and compiled machine-code that must be run by the CPU
- Diagram shows programs Pi to Pk – these are programs OS is concurrently running
    - Below is resource sharing between some memory and CPU that programs have to run on
    - Thus, programs need two things:
        - CPU time
        - Their compiled code needs to be placed in physical memory
- Diagram shows ***extended machine interface*** - a resource abstraction over N programs that the OS must concurrently run
    - This resource abstraction means that these N programs use virtual memory to share the space of limited main memory
    - Multiplexing: sharing of main memory by multiple programs
        - Where programs are partially loaded into memory (where the rest of a given program is kept in secondary storage), and each of the programs is executed bit by bit
- Multiplexing occurs on both the processor (CPU) and main memory (RAM)
    - CPU can only execute one program at a time, but N programs are being run concurrently on the OS, hence these N programs must be executed in brief slices of CPU time with rapid switching between them (to give the illusion of seamless concurrent program execution)

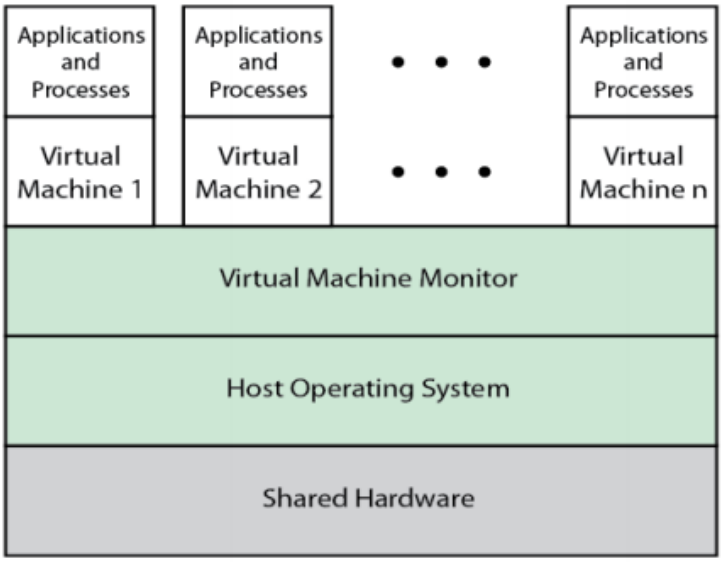
OS as a Kind of Virtual Machine



- OS can be viewed as its own type of virtual machine
  - Key idea of a virtual machine is that each process seems to execute on its own processor with its own memory, and devices
    - It's just an abstraction of the physical hw config that lies underneath – emulates all the components of the physical machine
  - Necessary, since the running of any OS mandates that the (limited) resources of the physical machine must be shared
    - That is, virtual devices are sliced out of the physical ones
    - e.g. virtual memory is an abstraction of physical memory
- Hides complexity and limitations of hardware from application developers

App Virtual Machines

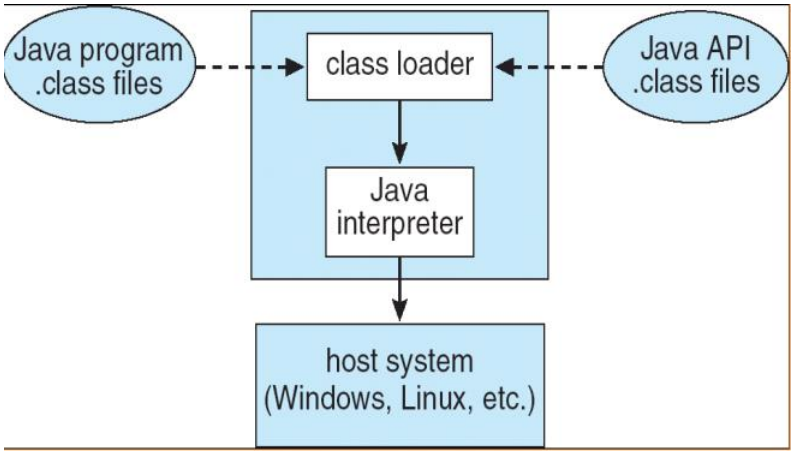
- **Vmware:** Hypervisors
- **Java Virtual Machine (JVM)**



- Diagram illustrates the running of multiple virtual machines (VM) which in turn run their own applications
- Diagram:

- VM is emulating the physical machine and provides an interface to the OS application layer
  - VMs provide functionality needed to execute entire OSs and as such VMs are useful for running different operating systems simultaneously on the same physical machine
- Layers show lvls of abstraction
- VM is emulating OS – vm monitor manages that
  - VM monitor, which in the case of running multiple VMs, ensures that there is protection between VMs (i.e. no sharing or interference between the running of multiple VMs on the same host OS)
- Java vm – emulates underlying OS, so as long as programs compiled into byte code => interoperability and can be run anywhere because Java vm emulates its own OS

Example: Java VM



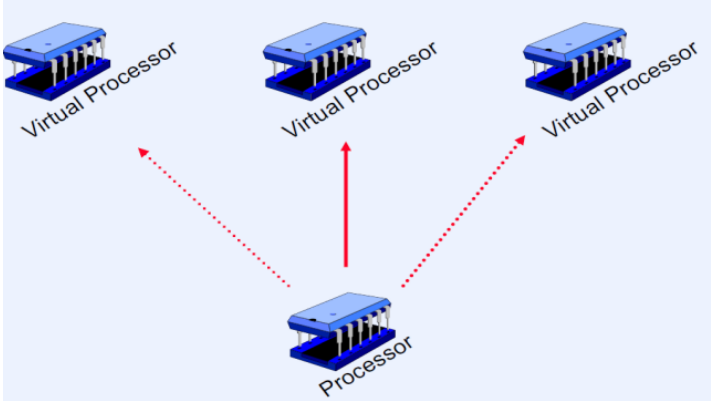
- Java Virtual Machine (JVM) allows Java code to be portable between various hardware and OS platforms
- JVM enables a computer to run Java programs that are compiled into Java byte-code
- JVM allows inter-operability of Java programs across different implementations so that Java developers need not worry about the underlying hardware platform and OS

Concurrency vs Parallelism

- Concurrency – can exist on single CPU
  - Multi-tasking OS
- Parallelism: Speed-up via N computations in parallel
  - Computer system has N CPUs or a single CPU has N cores, meaning that N programs are able to truly run in parallel (as opposed to concurrency where N programs must run on one CPU)
- Both are important topics in OS design, especially concurrency, since many operating systems run on single processor computer systems, where multiple programs must be run concurrently
  - Concurrency is essential if any multitasking (concurrent execution of programs) is to be done by an OS
- Running multiple programs on shared hw platform introduces many problems – must have **shared state** as programs are accessing shared memory

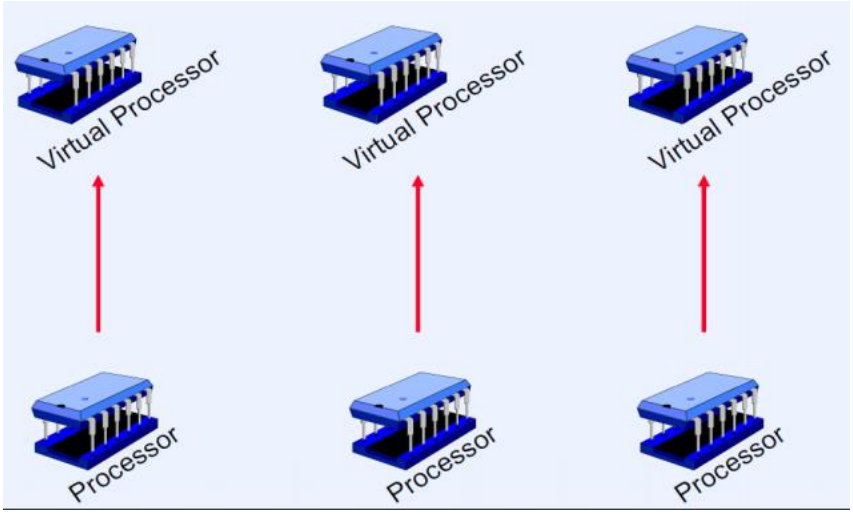
- Key problem with concurrency is that multiple programs (processes) must be simultaneously executed on a single CPU, and that these multiple processes must safely interact (e.g. not over-write each other)
    - Common approach to achieving this is multi-threading, where a program is split into multiple processes (threads) that run concurrently on one CPU
- Problems get tricky when you introduce parallelism:
    - Multi-core
    - Multiple CPUs
    - e.g.: C/C++ with MPI parallel processing libraries
- Current popular idea of common shared state between multiple threads is risky

Concurrency



- Concurrency: **multiple programs** (processes) are run on a **single processor** (CPU)
- Figure illustrates the use of multiple *virtual processors* as a means of running multiple programs on a single physical processor
    - A virtual processor is its own virtual machine representing all of the functionality of the underlying physical processor, meaning that a program can run on a virtual processor just as it would on the physical processor
    - Advantage is that we can have multiple virtual processors running simultaneously in order to emulate a multi-processor system
        - These virtual processors are protected from and cannot interfere with each other during the concurrent running of N programs

Parallelism



- Diagram shows a multi-processor set-up that an OS could use to achieve true computation processing speed-up when running N programs concurrently
    - Computer system has N CPUs, meaning that N programs are able to truly run in parallel (as opposed to concurrency where N programs must run on one CPU)
- As in the case of concurrency, such an OS would make use of virtual processors, but where one virtual processor would be run for each physical processor
    - This N virtual processor for N physical processor setup is used so as the processes running on each processor are protected from and cannot interfere with each other during the parallel running of the N programs

Memory Management

Virtual Resource

- Have limited main mem – but have many processes running and must accommodate this processing
- Physical Reality – different processes (threads) share the same hardware
- Two main approaches to multiplexing in an OS
    - (1) Multiplex **CPU operations** - scheduling processes to run concurrently on a single processor
        - Scheduling / Concurrency
    - (2) Multiplexing **memory** - using virtual memory to enable many processes to share physical memory
        - Virtual Memory
- Why do we care about memory sharing?
    - Complete working state of a process (and kernel) is defined by its data in memory
        - At any given time, many processes will be running on the OS, but these different processes must not use the same physical memory addresses
    - Hence, some solution is needed that allows multiple processes to concurrently run but to effectively and efficiently share limited physical memory
- Different processes must not access each other's memory – except when…?
    - In some cases we do want a process to access the memory of another process, such as inter-process communication (e.g. process A needs the computational result of process B)
        - Shared memory access is even more complicated

Memory Multiplexing

- Memory multiplexing (memory sharing)
- Protection: prevent memory access by other processes
    - Processes are protected from each other - memory access of one process by another is prevented
        - Thus, they cannot accidently overwrite each other's data
    - Hence necessary to have different memory chunks (pages) be given special access
        - e.g. read only, write only, invisible to process x
    - Example
        - Kernel data - used exclusively by system-level applications in the OS
            - Protected from that of user programs - there is a specific range of physical memory addresses that can only be used for storing kernel data, and can never be accessed by user programs
            - Special part of main memory is allocated to OS sys kernel

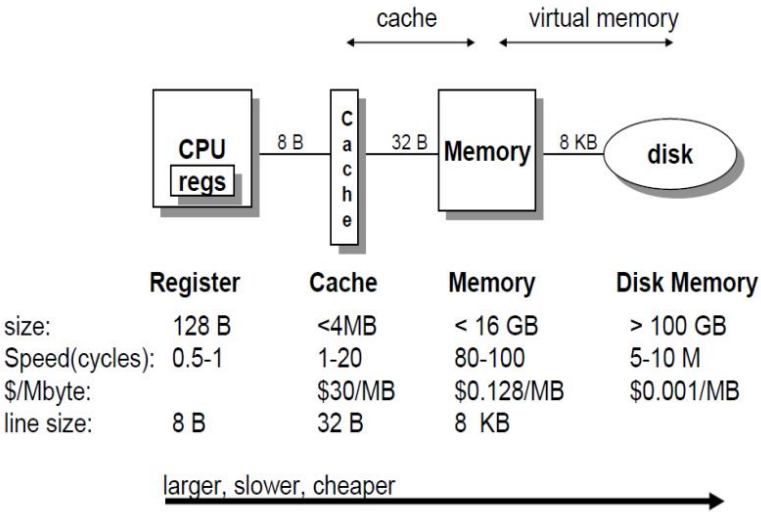
Controlled Overlap

- Any implementation of process protection requires controlled overlap between concurrently executing processes, given that some applications require inter-process communication
    - Usually – process (thread) states should not access same physical memory address – unexpected overlap causes chaos!
    - But would like to process states to overlap in some cases - inter-process communication
- E.g. files shared by multiple concurrently running processes, and TCP/IP socket communication by multiple concurrently running processes
- In such cases, there must be computational mechanisms that ensure controlled overlap between multiple processes
    - Processes effectively communicate as to when each will access the shared resource so that there is no usage overlap

Translation

- Memory address translation: mapping of memory access from **virtual addresses** (a specific memory address in a very large range of possible memory addresses) to **physical addresses** (a specific memory address in a very limited range of possible memory addresses)
    - Virtual is some *temp storage* for processes that're scheduled to run – either have not yet run, or have run and been suspended
    - Physical address is main memory i.e. RAM , where processes are currently allocated memory and running
- Essential for memory management by the OS and for virtual memory to function properly
- For example, a computer's main memory (RAM) uses physical addresses to specify specific memory locations for programs currently loaded into main memory and executing
    - Whereas the CPU uses virtual addresses, thus giving it access to all programs that need to be scheduled to run (loaded into main memory)

Memory Hierarchy



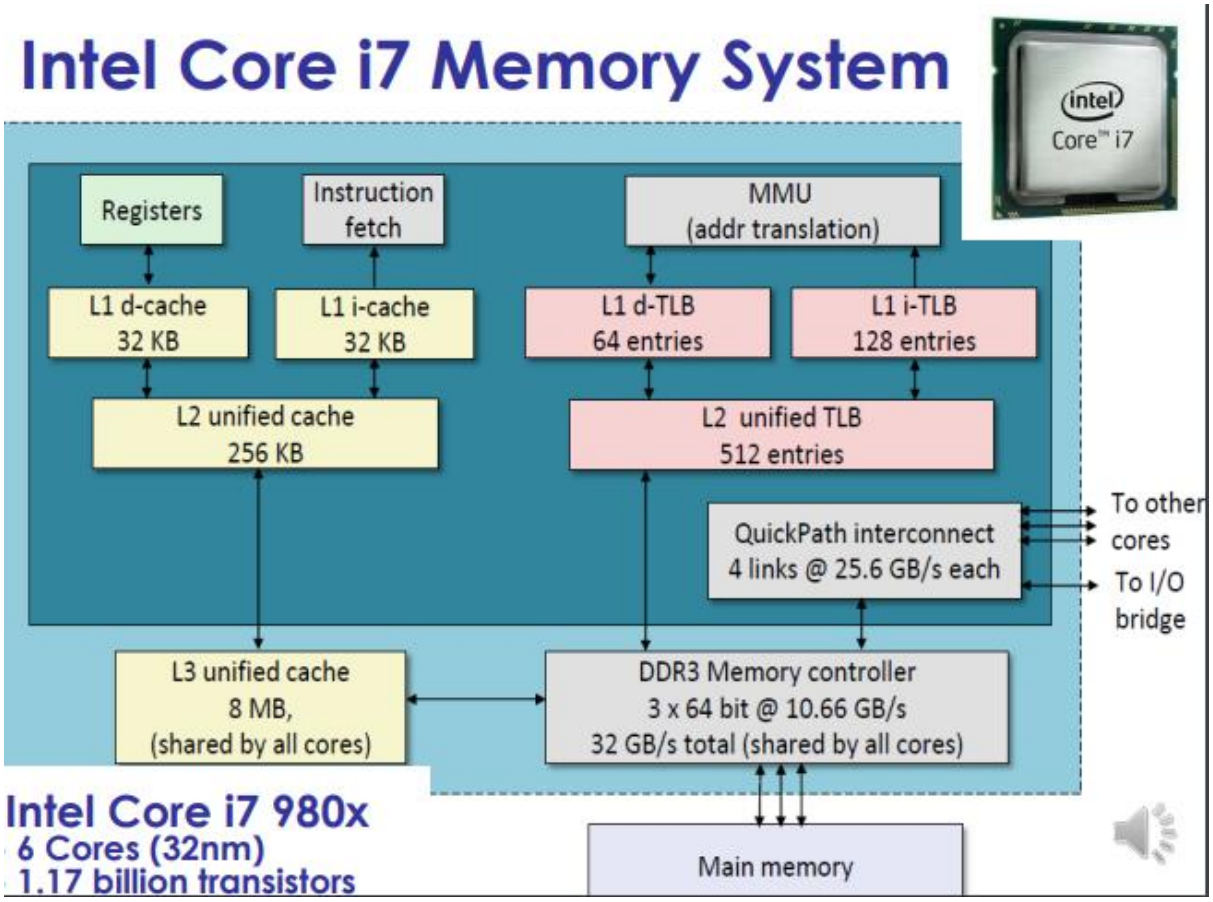
- Diagram shows relative access speeds (for retrieval and use of stored data) for cache versus virtual memory
    - Arrow – larger, slower cheaper => larger memory is cheaper but slower
- Virtual memory uses both main memory (RAM) as well as secondary storage (e.g. external hard-disks) to manage the concurrent running of multiple processes
    - Cache memory uses a type of volatile random-access semiconductor memory (only retains its stored data as long as the semiconductor is powered)
- LHS shows CPU and registers – regs are small memory caches associated with CPU
    - Regs are fast (in terms of the number of CPU computation clock cycles, e.g. 0.5 to 1 clock cycle) and quick to access – typically hold machine code instructions used by CPU to run
    - CPU must be able to access and retrieve this data very quickly
- RHS – disk is hd and memory is RAM
    - RAM has limited storage capacity compared to secondary storage (hard-disk memory), but the access speed of data stored in RAM is significantly faster than the access speed of hard-disk memory
- Other cache memory is implemented as blocks of memory used for **temporary storage** of data that is **frequently used**
    - E.g. cache memory is typically attached to CPUs or hard-disk drives
    - Such cache memory has limited capacity (compared to main memory), but is much faster ito access time, and is also more expensive (per MB), compared to main memory
    - Used to perform kernel lvl instructions that need to run all the time in order for OS to work at all
    - Each cache typically has hard coded machine code instructions in it – can't be changed by user
- Cache Memory: computer memory with very short access time – used for storage of frequently used instructions or data
    - Typically used by CPUs, disk-drives and other hardware devices that must quickly access and retrieve frequently used data

OS Storage versus Performance

Level	1	2	3	4	5
Name	registers	cache	main memory	solid state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25 - 0.5	0.5 - 25	80 - 250	25,000 - 50,000	5,000,000
Bandwidth (MB/sec)	20,000 - 100,000	5,000 - 10,000	1,000 - 5,000	500	20 - 150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

- Memory size versus bandwidth (data transfer speed) versus memory access time (in nanoseconds)
- Trend that as we move from specialised hardware cache memory to less specialised secondary storage (from left to right in the table), then storage capacity increases orders of magnitude
    - But at the cost of significantly slower data access times and lower data-transfer bandwidths.

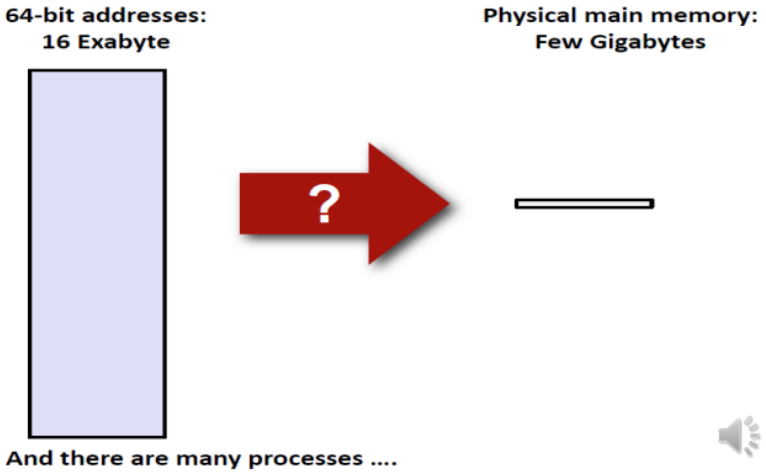
Example: Intel Core i7 Memory System



- Note the arrangement of cache memory attached to the CPU and the relative storage capacities of these memory caches and the connection of this cache memory (via a memory controller) to main memory (RAM).
- Each cache memory on the CPU is used to store instructions for specific low-level hardware functionality of the CPU - meaning that low storage capacity is sufficient (for storing series of instructions in machine-code), but very fast data access times are needed (otherwise the CPU would be too slow to be practically useful)
- MMU responsible for translation between virtual and physical memory
    - Entries (64,512 etc) – lookup tables which allow CPU to do the translation

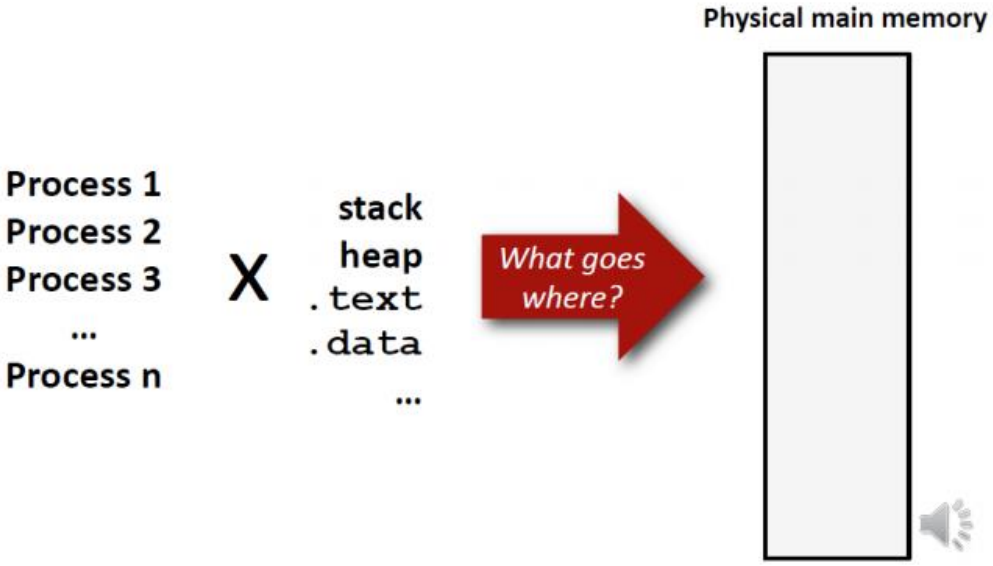
Memory Management Problems

Memory Translation



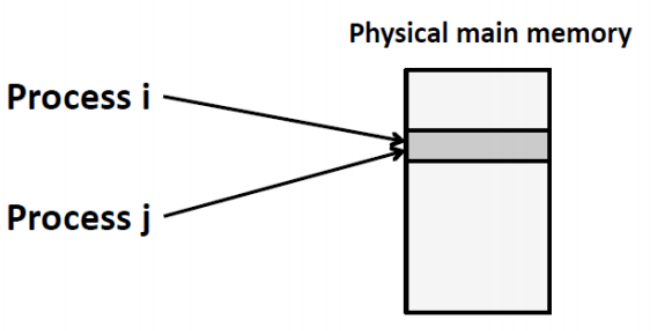
- The basic problem of memory translation, i.e. that there are many processes that can be assigned logical memory addresses (e.g. using 64 bit memory addresses), but there are relatively few physical memory addresses (e.g. typical RAM capacity for a personal computer is approximately 16 GB)
- While the OS can have a great many processes assigned virtual memory addresses, the number of actual memory addresses available in physical main memory are very small by comparison
    - The memory translation problem is then how does the OS map (translate) a very large number of process memory addresses to a very small number of memory process addresses, such that all processes are able to be periodically swapped in and out of main (physical) memory and run effectively and efficiently
    - A memory translation (mapping) function is needed to ensure that a process with a given virtual address is translated to a valid physical address that is free (currently unused by another process) when the given process is swapped into main memory for execution
- Diagram – LHS shows a virtual mem address space
    - Need to translate all these addresses to a relatively few physical main mem address (RHS)

Memory Assignment



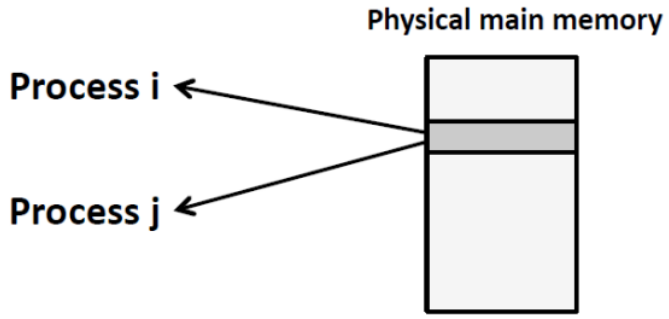
- Each process (after being scheduled to be swapped into main memory for execution), will have its virtual memory address translated to a physical memory address
    - But the assigned physical memory must be a *currently free* chunk of main memory
- Assigned chunk of main memory must be sufficiently large to temporarily store the assigned process, but not so large that there is additional left-over memory (wasted while the process is executing)
- Diagram:
    - LHS shows n processes running concurrently (each assigned a virtual memory address)
    - LHS shows n processes scheduled to be loaded into main memory and run
    - Each process is composed ito memory, the stack, heap, text, data – all of this needs to be assigned to some physical mem address
        - Each process will have different size mems (some complex, some simple) – how to assign what process to what physical mem address, and how much time to allow process to stay in main mem in order to run
            - Large processes require lots of memory and time to run – should they have more/less time in main mem compared to a smaller process that will run quicker?
    - Middle of the figure, shows that size of data (e.g. files and other shared memory) used by a process has to be accounted for during the assignment of a process to physical memory

Memory Protection



- Given that n processes have been assigned virtual memory addresses, translated to physical memory addresses and assigned to be run at these main memory addresses, the problem remains that **main memory capacity is significantly limited** so the n processes must often **share main memory addresses**
- Thus, some form of memory protection is needed to ensure that processes that are assigned to the same physical memory address do not access this common address at the same time
- Diagram:
    - Process i and j being assigned the same main memory address
    - Need to ensure that process i and j, given that they're assigned to the same memory slot, are not assigned at the same time (so as not to overwrite each other)
    - Must have mechanism that allows process i to run for some time (for a given number of CPU computation cycles) before it is suspended, swopped out and process j swopped in
- Goal of memory protection is to ensure that one process does not over-write another in main memory, meaning that memory protection (between processes) is ensured

Memory Sharing



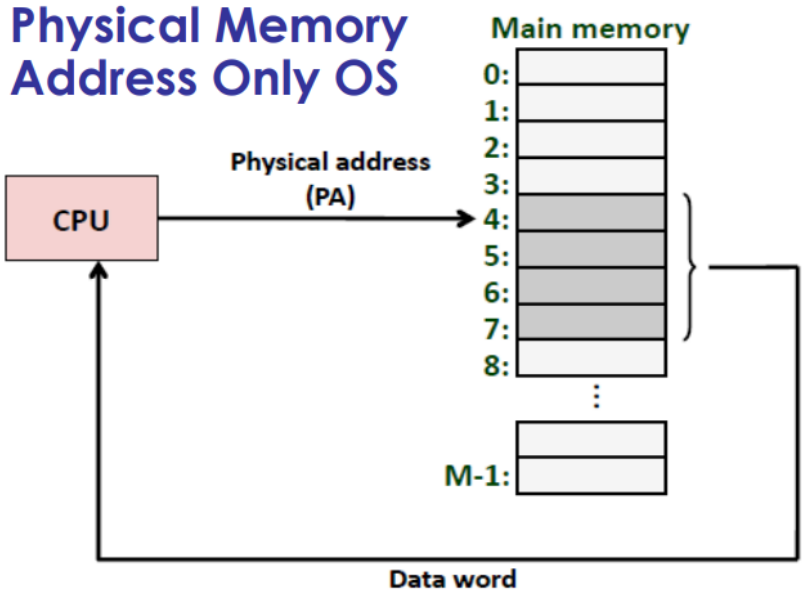
- N processes (scheduled to run and thus be swapped into main memory) can share the same address space in physical main memory
- Problem is that the n processes must be suitably divided up, so as all processes are assigned a physical memory address and that **an approximately equal number of processes share a physical address in main memory**
    - What # of processes should share a given mem address?
    - Should one group all small processes together in a large chunk of memory?

    - Allocation of processes to a given shared memory address – need to ensure fair # of processes given memory address and at the same time ensure memory protection between processes
- Thus, the OS memory management system must ensure effective sharing of each specific physical main memory address
    - Specifically, that multiple processes are assigned to a given physical memory address
        - i and j shown in the example
    - That the processes are protected from each other
    - That only one process is resident at the physical memory address
        - For example, that process j is not swapped in while process i is resident in main memory

Virtual Memory

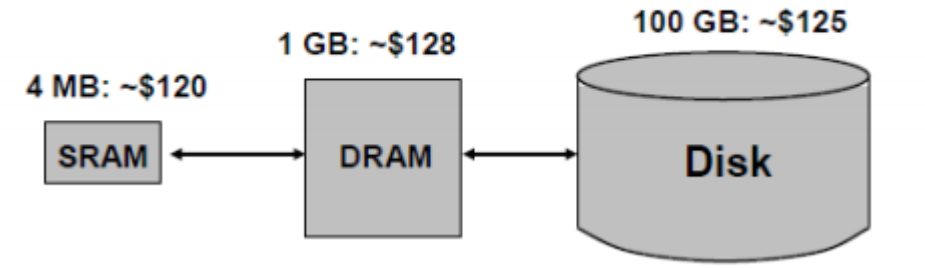
- Virtual memory consists of a large set of memory address references (temporarily kept in hard-disk secondary storage) for processes that are scheduled to be swapped into main (physical) memory and executed (for an allocated number of CPU computation cycles)
    - References to virtual memory are translated to physical addresses before execution
- Process: Made up of pieces (**pages** or segments)
    - Processes do not need to be in continuous regions of main memory
    - Processes can be moved around different regions of main memory
    - Parts of a process need not be in main memory during its execution
- Sum of logical memory (for all processes) can thus exceed available physical memory
    - Main concept and benefit of virtual memory

Physical Memory Addresses



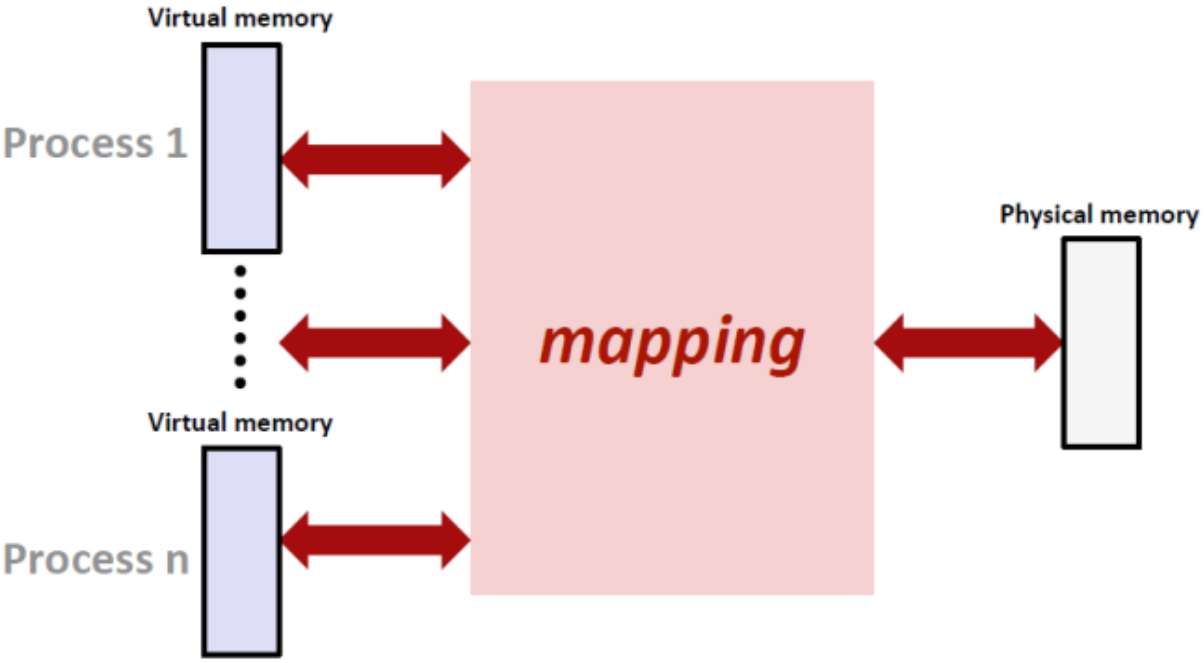
- Diagram shows role of CPU in generating physical addresses in main memory
    - Addresses generated by the CPU point directly to bytes in physical memory
- Once physical address is generated, physical chunk of memory assigned – means process can be loaded into main memory and executed by CPU
    - Each address is one data word – 32/64 bits
    - In the context of an OS, words are **fixed-sized data-chunks** used by the CPU for physical memory assignment
        - The number of bits in a word (word size) depends on the specific CPU design or computer architecture
- When the CPU assigns an executing process to physical memory, this process is assigned a chunk of memory comprising n data words

Virtual Memory: Motivations



- Key motivation - the amount of memory required to run all processes on any modern OS invariably exceeds the amount of physical (main) memory available
    - Fitting everything in – Process address space can exceed physical memory size
        - Vm has many virtual addresses for all processes that are scheduled to run by OS
            - Virtual address space depends on OS – usually 32/64 bit addressing
- Thus, all possible virtual addresses map to relatively few physical addresses
    - E.g. Address space -> corresponding number of bytes available for storing the addresses of processes (scheduled to be executed)
        - 32-bit address ~ 4,000,000,000 (4 billion) bytes
            - Available as address space for virtual memory to assign process's addresses
        - 64-bit ~ 16,000,000,000,000,000,000 (16 quintillion) bytes
- Diagram shows that the virtual memory address space of all processes can easily exceed physical (main) memory size

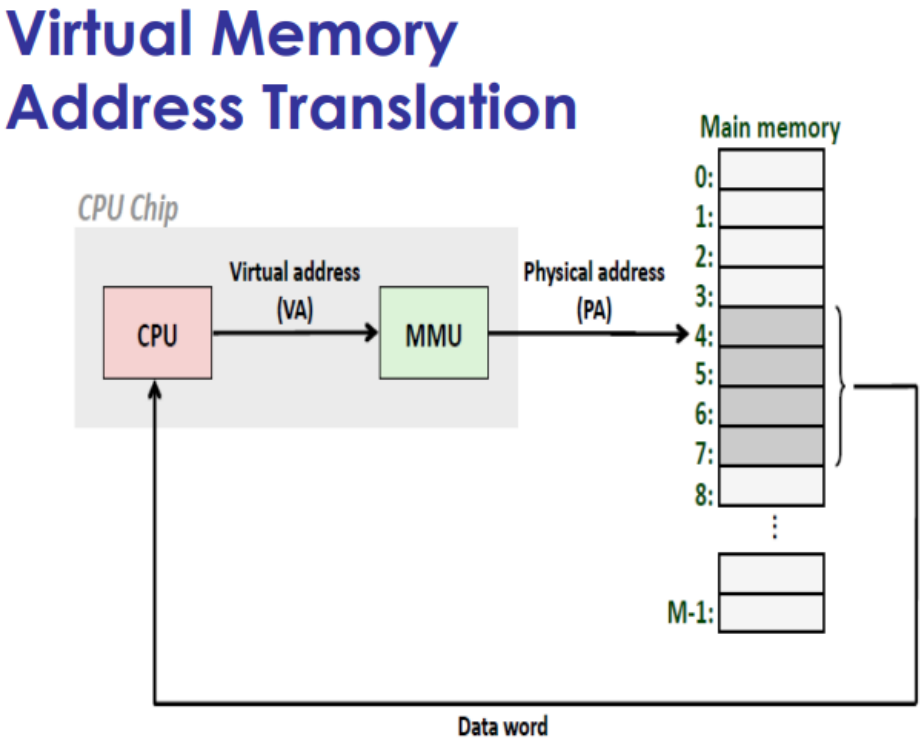
Overview



- Abstraction view of virtual memory in an OS

- Diagram - shows n processes that are currently scheduled to be loaded into physical memory and executed
    - Each process is assigned a virtual memory address
    - The memory needed to run each process is given a virtual memory allocation
        - These processes are temporarily kept in secondary storage while waiting to be swapped into main memory
    - Mapping (translation) happens between all virtual memory address spaces and physical memory
        - OS-system level mapping (translation) function ensures that every process (currently assigned a virtual memory address) can be loaded into a valid region of physical memory
            - i.e. a main memory address space that is currently free and addresses a chunk of main memory sufficiently large to hold the assigned process
        - Mapping done by MMU – employs translation for virtual to physical mem addresses, protection and assignment of each process schedule to run in vm to an actual physical mem address

Virtual Memory Address Translation



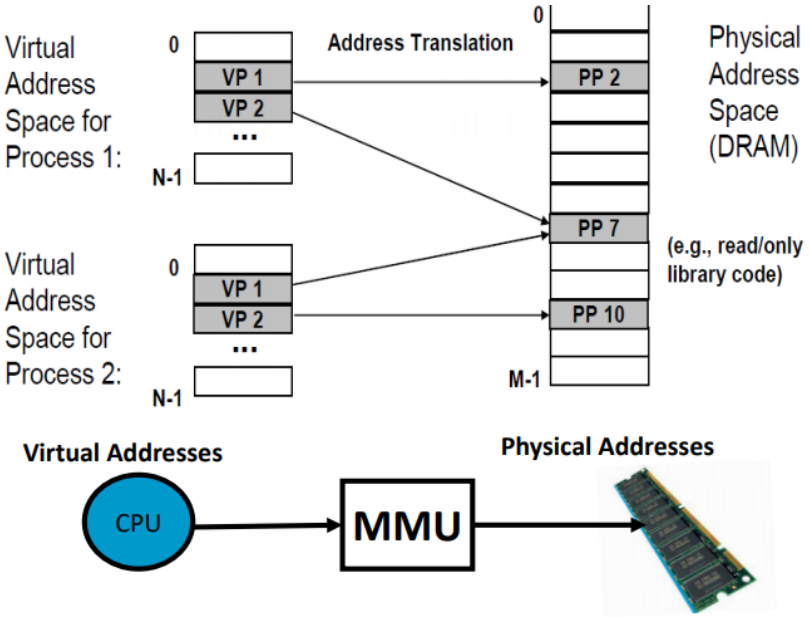
- Virtual memory addresses (VA) are translated to physical memory addresses (PA) (in order to be loaded into main memory and executed)
- Memory Management Unit (MMU) of an OS performs the VA to PA address translation (mapping)
- MMU used for this address translation is an OS-managed look-up table (page table) stored in cache memory, attached to the CPU
- Process stored in virtual memory is scheduled to be run and must be loaded into main memory
    - VA to PA memory address translation occurs

- MMU checks the cache, maps the VA to PA address, so as the process addressed by the VA can be loaded into the PA (and subsequently run)

Simplifying Memory Management

- Multiple processes can reside in main memory
- Main motivation for virtual memory is to simplify memory management
  - i.e. virtual memory allows **multiple processes to be in memory at same time**, sharing physical memory addresses, but protected from each other
  - Only "active" code and data actually in memory
- The translation (mapping) between virtual memory and physical memory raises the problem of resolving physical memory address conflicts
  - How do we resolve address conflicts?
    - i.e. when two virtual memory addresses translate to the same physical address
- What if two processes access something at the same physical address?

Virtual to Physical Memory Address Translation

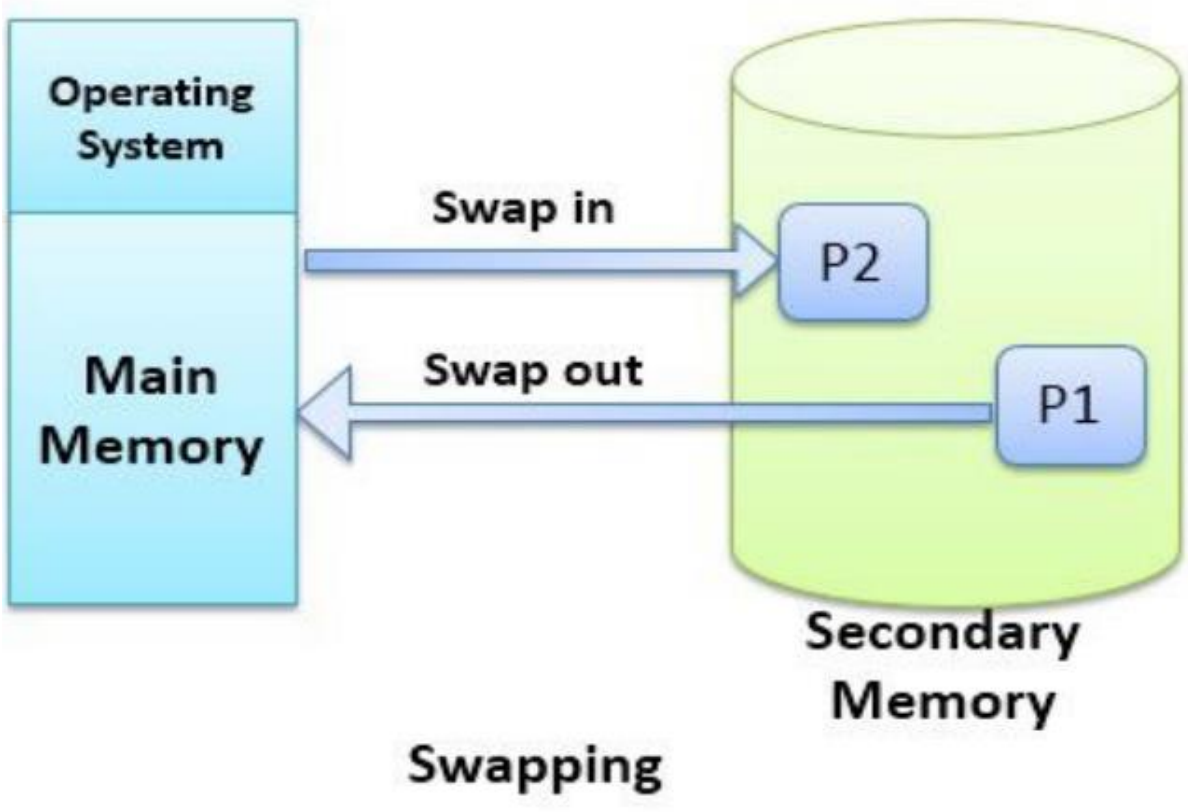


- Main point – each *virtual memory space* for each process can be broken down into *pages* (memory chunks), which can then be *assigned to non-continuous* parts in physical memory space
- Solution to address conflict
  - OS ensures that a separate virtual address space is used to address the memory needed to run every process that has been scheduled to run
- During the general running of an OS, some processes are active (loaded into main memory and using CPU computation time), while others are suspended (stored in virtual memory)
  - Hence, when a currently suspended process is scheduled to start running, then the MMU (Memory Management Unit) must translate the suspended processes virtual address to a physical address
- Issue - the physical address will most likely be occupied by another (currently running) process, and thus an address conflict resolution check is needed before a new process can be swapped in from virtual memory

- Diagram:
    - Shows an example of a virtual address space for two processes (stored in virtual memory
    - Each process split up into VP1 (virtual process 1) and VP2 (virtual process 2)
        - i.e. Each process is split up into two chunks of memory (e.g. process machine-code instructions) that run in main memory
    - Process 1 – shown as PP1 and PP7
    - Process 2 – shown as PP7 and PP10
        - These PPxs are specific physical memory addresses that point to specifically sized chunks of memories
    - Any process can be split-up into discontinuous (disjoint) chunks of memory at various main memory addresses
- Task of MMU to ensure that one process has been swapped out of this physical memory before the next process is swapped in
    - E.g. VP2 swopped out of physical memory PP7, before VP1 swopped in

Virtual to Physical Memory Swapping

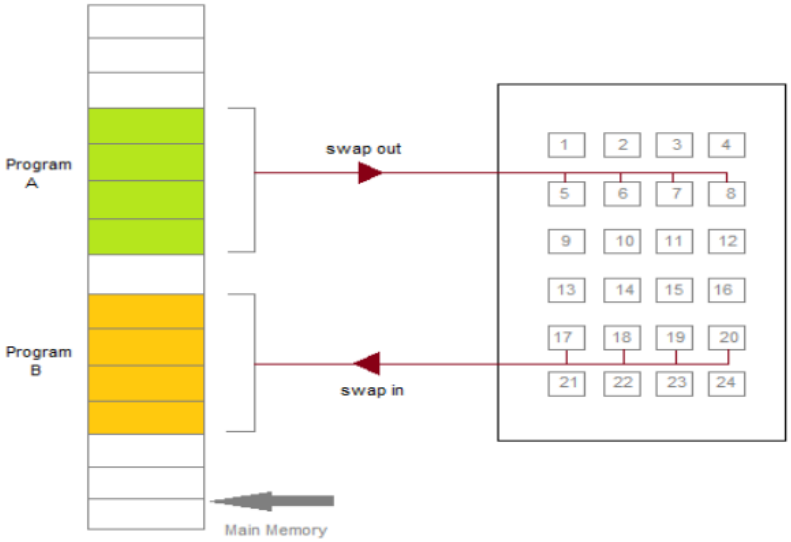
(1)



- Memory swapping is necessary in order for virtual memory to function effectively
- Diagram 1 – shows two processes (P1, P2) being stored in virtual memory (secondary memory)
    - LHS has RAM (limited), RHS has secondary memory – shows bits of processes that need to be swapped in and out in order for them to be run
    - Both process 1 and then process 2 has been scheduled to run by the OS

- Example assumes that main memory is only large enough to accommodate one process at a time
  - So, process 1 must be swapped back into secondary memory before process 2 can be swapped out of secondary memory and into main memory

(2)



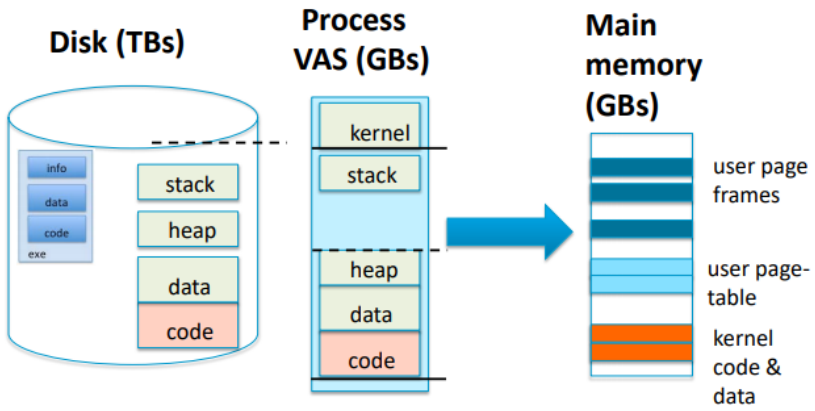
- Diagram illustrates the notion of each process being comprised of multiple memory chunks (pages)
  - Memory chunks correspond to pages which have their own vm addresses
  - LHS shows two programs (processes) A and B resident in main (physical memory)
    - Each process is comprised of four (4) pages (right-hand side) in virtual memory (residing in secondary storage)
  - Four pages (for each process) are shown as residing in consecutive main memory addresses - these pages in main memory could be at discontinuous memory spaces
- Can swop individual pages / a few at a time in order to partially run each process, given that there might not be enough main physical mem to swop in all pages necessary (in order to run the entire process)

Loading Into Memory

- A program is the source code, libraries and data necessary to run the program
- .exe is the program executable actually running
  - Executable (exe) is thus a process that has been scheduled to run by the OS
    - Is currently suspended (stored in virtual memory) or is currently running (stored in main memory)
  - Lives on disk in the file system
  - Contains contents of code and data segments, relocation entries and symbols
  - OS loads it into memory, initialises registers (and initial stack pointer)
  - Program sets up stack and heap upon initialisation

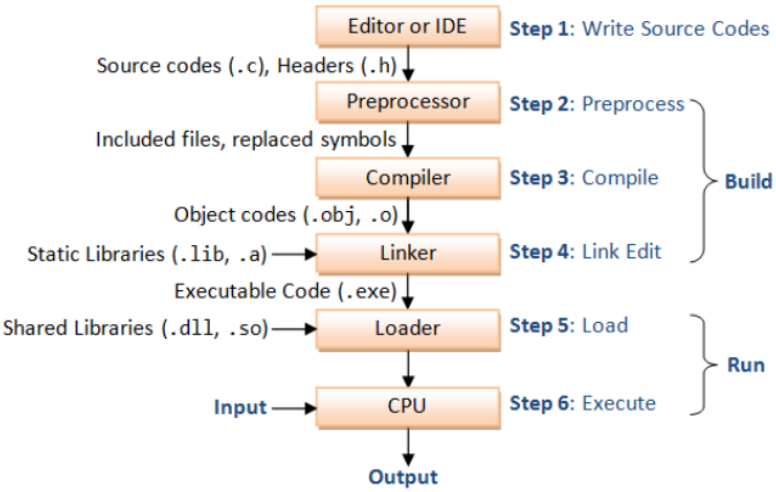
- Executable corresponds to a set of machine-code instructions - produced after compiling the program
    - Incorporates everything needed to run the process

Process Virtual Address Space



- exe = Process Virtual Address Space (VAS)
    - VAS provides spec for each process, of what process needs to execute
        - Have compiled code, data, associated files, heap and stack (dynamic and stack mem allocation for that process)
        - Each of these parts of the process, can be split up into memory chunks
            - Then assigned, chunk by chunk, into main memory – so process can execute bit by bit
- All processes (executables) scheduled to run by the OS actually correspond to a *process stack* stored in secondary storage (virtual memory)
    - Loaded into main memory bit by bit
- Process stack is stored in a Virtual Address Space (VAS) which corresponds to many chunks of memory (pages) kept in secondary storage
- Diagram shows VAS as being gbs in size and storing the process stack for all executables (processes) scheduled to run
    - VAS is stored in the secondary storage memory which is typically on the order of tbs in size
- When a process is swapped into main memory for execution, the pages that comprise the process are typically split-up and assigned to currently free (and most likely disjoint) physical memory addresses
    - Shown dark-blue chunks on rhs figure

Program Compilation Into An Executable



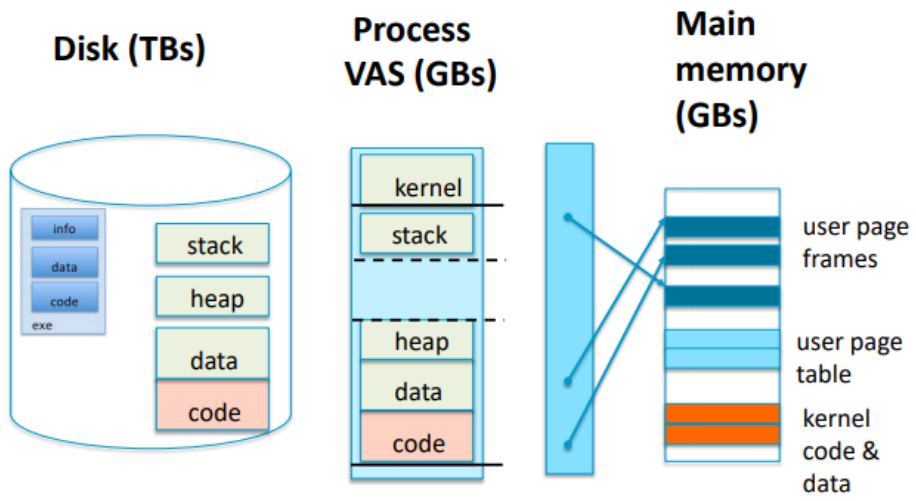
- Diagram – shows steps necessary to compile a C program into an executable (exe), before it can be scheduled to run in main memory by the OS
    - After source code is written - there are two key steps:
        - (1) build (done by the compiler)
        - (2) run (done by the OS)
- Build step:
    - Pre-processor does three tasks (as part of program compilation)
        - Removes comments from the code
        - Includes the header file (standard in C files) into the generated file itself
        - If any macros were used, will replace the macro name with (executable) code
    - Compiler will take the file (created by the pre-processor) code and create the assembly code
    - Assembler (still as a part of the compilation process) converts the assembly code into the object code (i.e. executable machine-code)
    - Linker links the machine-code generated from the assembler output with library function code and also merges multiple C files by compiling them into one executable file
        - Packs all the code into a single file, a .exe file
- Many OS environments allow dynamic linking, deferring the resolution of some undefined symbols until a program is run
    - Executable code still contains undefined symbols, plus a list of libraries that will provide definitions for these
    - Loading the program will load these libraries and perform a final linking
        - Load step in the diagram
- Final step is the running of the executable (exe), loaded into main memory, by the CPU

Program Execution

- Linking of libraries to be used by a given program may be done at compile-time (static linking)

    - E.g. in UNIX, ld combines a number of object and archive files, relocates their data and ties up symbol references
        - Usually the last step in compiling a program is to run ld
        - Otherwise, the linking of libraries is done at run-time (dynamic linking)
- Time:
    - Compile time
    - Link & Load time
    - Execution time
- Dynamic Libraries (dynamic linking done at run-time)
    - Linking postponed until execution – defers the resolution of undefined symbols until a program is run
        - Executable code will still contain undefined symbols, plus a list of objects or libraries that will provide definitions for these
        - Loading the program will load these libraries and perform a final linking
    - Stub code gets placed in executable at compile-time
        - Stub code – locates a memory-resident library routine at run time
        - Stub (in the executable code) then replaces itself with library address and executes it (the library at the address)
- Linker/link editor: a programming language utility program that takes one or more object files generated by a compiler and combines them into a single executable file, library file, or another object file

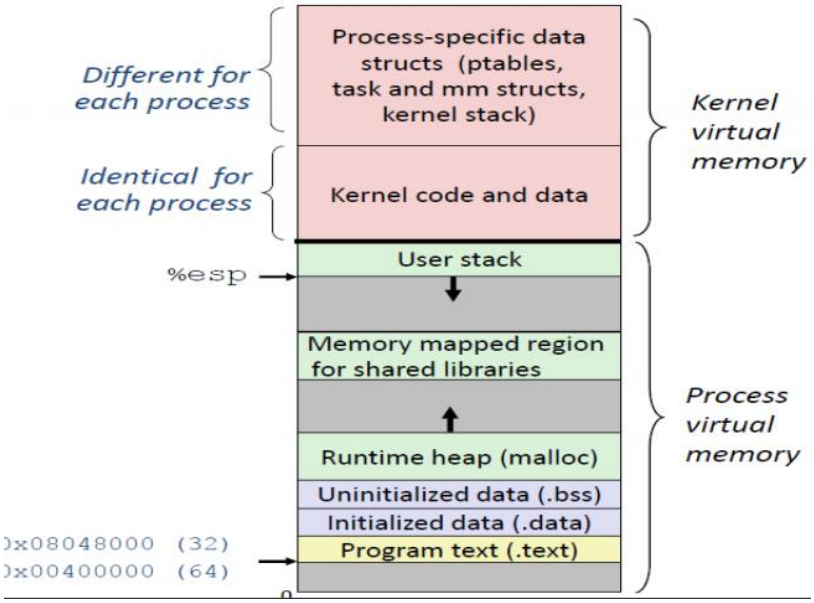
Mapping Virtual to Physical



- Diagram shows the mapping done from the Virtual Address Space (VAS) to main memory, necessary to run processes (executables)
- Reference to page frames and a page table (light blue, middle rhs)
    - **Page table holds the mapping (translation) function** which decides what pages (memory chunks of processes) are to be in main memory (and executed) and what is not (suspended in virtual memory)
        - Mapping function aka **page-replacement algorithm**

- Page-replacement algorithm (attached to pg table) decides what memory chunks (pages) to swap-into main memory (and out of virtual memory), and thus what processes (i.e. split into pages) should be running in main memory at any given time
    - Algorithmic process is known as *paging*

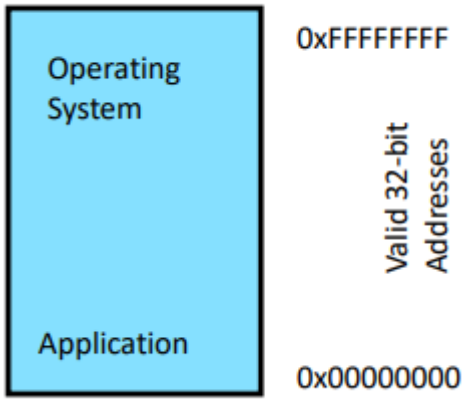
Process Memory Stack



- Process composed of a series of memory chunks stacked on top of each other
- Figure shows what is loaded into physical memory for execution
    - i.e. each process has its own memory allocation which includes kernel and process virtual memory parts
    - Linux process
        - .text – files, code used by program
        - .data – files
        - Runtime heap – dynamic mem allocation
        - Shared mem libraries
        - Kernel vm code – virtual addresses of kernel node used by OS only
- Kernel part of any process contains addresses of OS-specific libraries and other data that must be executed in order for the process to run at all
        - Note that the specific libraries and data could be different for different processes but the addresses where these are resident is the same for each process
- In both virtual memory parts, note that the process is broken down into memory chunks
    - Each chunk storing a specific part of the process
    - Memory chunks are essentially stacked on-top of each other starting at some base address in virtual memory
        - Diagram shows this as 0x08048000
- Each process memory chunk corresponds to some specific aspect of the original program
    - E.g. data segment, .data, is a portion of the virtual address space of a program that contains initialised static variables
        - i.e. global variables and static local variables

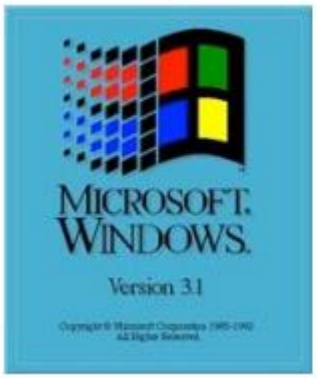
- The size of this segment is determined by the size of the values in the program source code

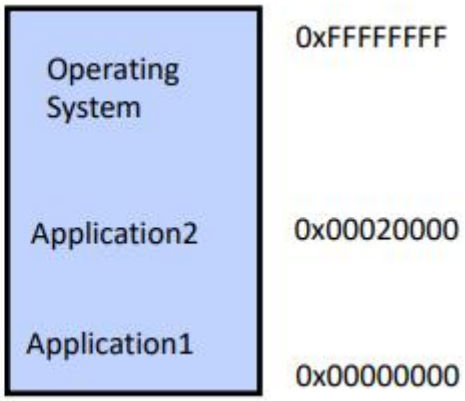
Uni-Programming



- Know how to get a process (application) executable – how to get it into main memory and run it?
  - Simplest way to do this process (executable) assignment to main memory is with uni-programming
- No translation (mapping from virtual to physical memory) or protection (between concurrently running processes)
  - **Only one process runs at a time** and a process always runs at the same place in physical memory
    - Application in diagram always runs at same place in physical memory since only one application runs at a time
  - But a process can access any physical memory address
    - Application in diagram can access any physical address
- By-passes the problems of translation and protection in virtual memory by only using physical (main) memory
- This approach is infeasible on any modern OS

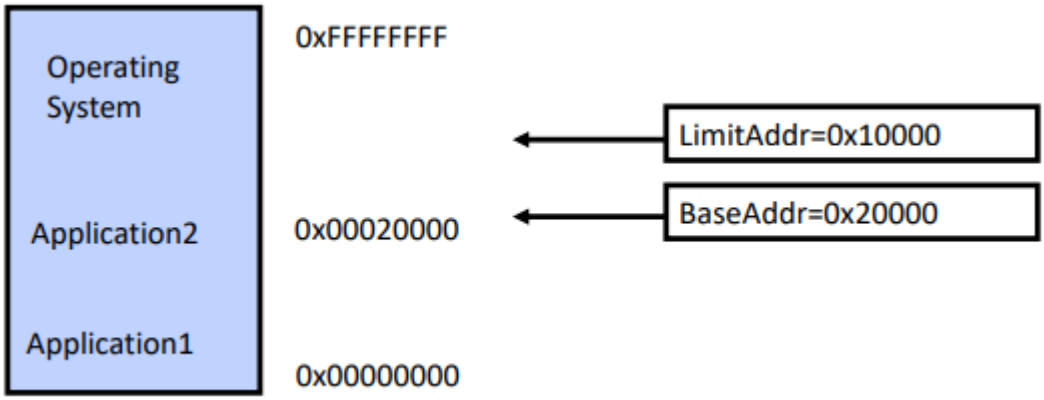
Multi-Programming





- Original concept of multi-programming - each process (application) was allocated a specific address in main (physical) memory, but there was still no memory address translation or protection
- Loader & Linker: Adjusts addresses while program loaded into memory (loads, stores, jumps)
    - Dynamically adjusted program addresses when they were loaded to try and prevent overlap of processes concurrently loaded into main memory
- No protection => bugs in any program can cause other programs to crash
    - e.g. via one program trying to access physical memory running another process
- Common approach for early OSes

With Protection

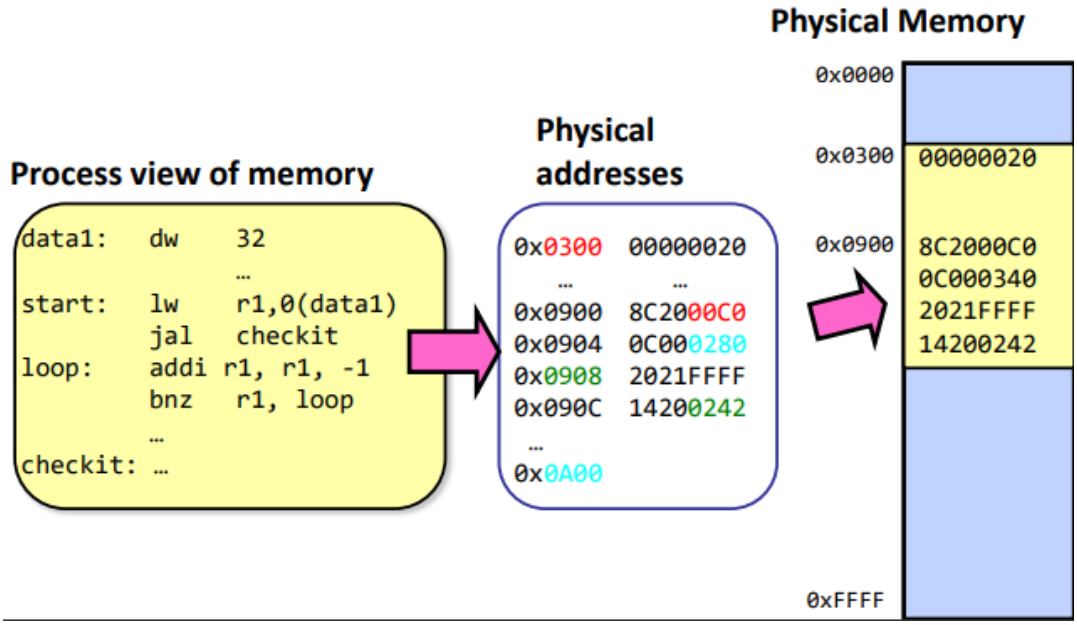


- Later version multi-programming OSs used *two special processor (CPU) registers* to enable process protection without memory translation
    - A processor register is a quickly accessible **cache memory location**, consisting of a small amount of fast storage, available to a CPU
- Two special registers: BaseAddr and LimitAddr prevented users from straying outside their designated memory areas when running processes
    - If a user tried to access an illegal address, this caused an error and the process would not try to execute
    - Users could not change the base/limit registers

- Diagram:
    - BaseAddr indicates the starting point of a physical memory address that can be allocated for a given user process
    - LimitAddr indicates the end-point of the physical memory address for the given process
    - in this case, each user process is assigned a fixed sized chunk of memory and user processes are essentially stacked on-top of each other in main memory
        - Hence preventing unintentional over-write of main memory by different processes
        - Where one process would end, the next starts ito physical memory address

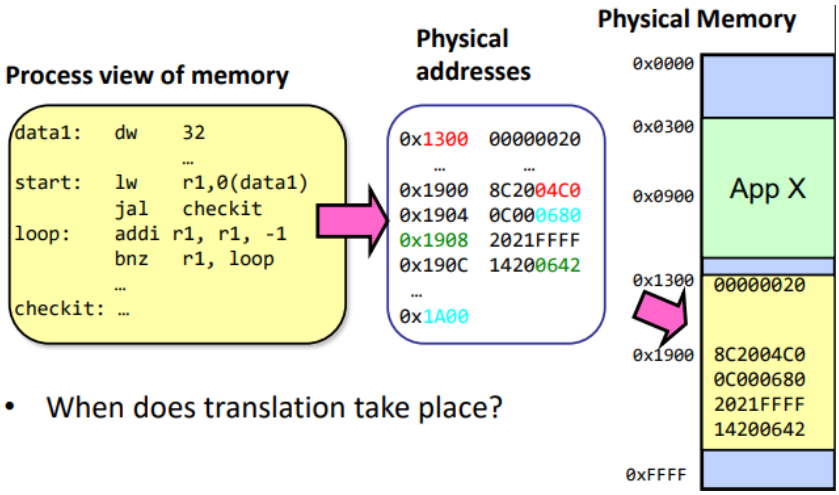
Programs

Program Execution: Data to Memory



- When process scheduled to run – it's a compilation of the program i.e. set of assembly lvl machine code instructions that are allocated memory
    - Must be allocated into main memory and executed instruction by instruction
- Diagram
    - LHS shows process view of memory, where the program has been linked with necessary data (libraries) and compiled into *executable machine-code*
    - Executable (process) is then assigned a **specific address in main (physical) memory**, where this physical memory address is a translation (mapping) from a virtual memory address
    - This translation happens at execution time when the Memory Management Unit (MMU) assigns each process chunks of physical (main) memory

Program Translation and Execution



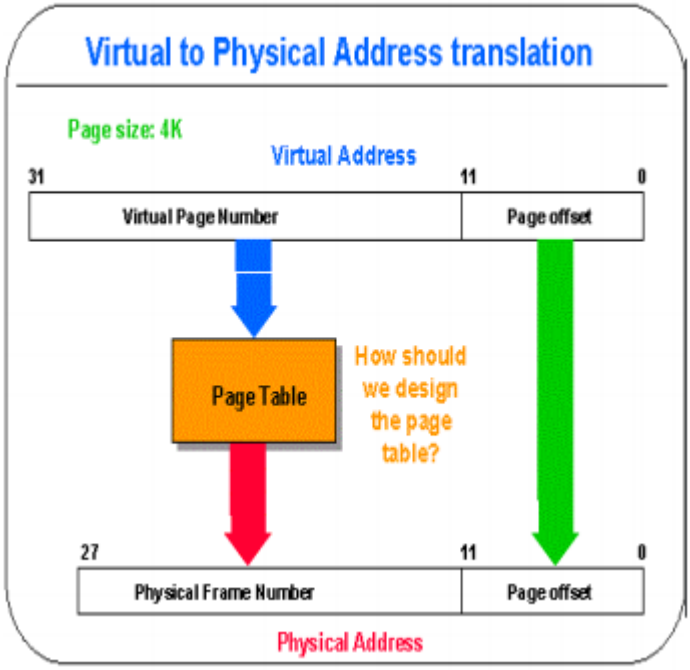
- When does translation take place?

- Diagram – shows detailed memory assignment step
    - The assignment of a process (executable) to main (physical) memory also entails assigning enough memory for the process to partially/completely execute (if the process is assigned enough memory)
    - RHS - AppX is a memory chunk large enough to run the given process
        - All memory chunks for given process that denotes AppX have been assigned memory and are ready to run in physical memory (run by CPU)
    - Every other address (in the yellow block) are physical memory addresses that are currently free
        - It is the task of the Memory Management Unit (MMU) to update currently available physical memory addresses whenever a process is assigned a memory chunk in physical memory
            - Ensure another process is not assigned to a physical memory address that is somewhere in memory already allocated to an executing process

Partial Loading Advantages

- In most cases a process is nearly always partially loaded (unless the process executable requires very little memory) due to **memory constraints of main (physical) memory**
- Many processes can concurrently be maintained in main memory for execution, but only a few pieces (memory pages) are loaded for each process
- Many processes can be concurrently executed without filling up main memory
- Such concurrent bit-by-bit loading and execution of many processes also means that with many processes in memory, on average many more processes will be executing
    - Many processes in memory increases CPU utilisation – avoids waste of computational resources
- Process memory pages (segments) loaded as needed
- So – can execute processes that use more memory than is available
    - That is – logical address space is much larger than physical address space

Memory Address Translation



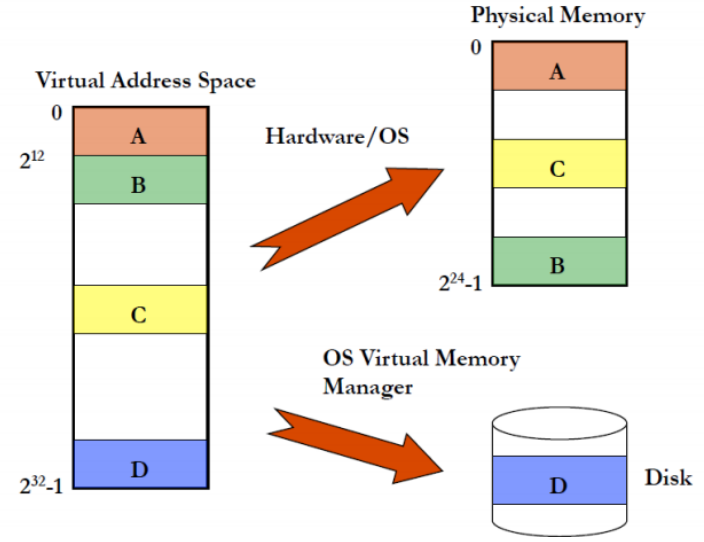
- Page: a small chunk of memory that constitutes a small part (e.g. 4KB) of the total memory needed to execute a process
    - Assigned a vm address – looked up in page table and has corresponding physical memory address
        - When it's time for that pg to be run, page table does the mapping and translation and the pg is assigned into physical memory and executed
    - For each process that the OS is concurrently running, many pages are assigned (from their temporary storage in virtual memory) to currently available physical memory addresses
- Page table: look-up table used by the MMU to decide (for a given process) how many pages should be swapped in to physical memory (translated and assigned to main memory addresses), and how many pages should be swapped out to virtual memory
    - This virtual to physical translation and assignment of memory pages is done by the Memory Management Unit (MMU)
- Diagram:
    - Virtual address space is 32 bits (0,. . .,31), which corresponds to that used by a 32 – bit OS
    - Physical address space (where processes can be assigned memory to execute) only extends from: 0,. . .,27
        - This is because the address space: 28,. . .,31 is reserved for OS-kernel level applications

Virtual Memory: Motivation

- **Remove burden of dealing with limited main memory** (when have many programs that need to be run at the same time)
    - Total memory for all programs should be larger than the physical memory

        - Virtual memory uses secondary storage to create an illusion of an even larger memory space without incurring any cost
    - Addresses for programs to make them all fit simultaneously in memory
- **Efficient and safe sharing of limited main (physical) memory for many programs** (multiple concurrently running processes)
    - User programs should not have access to kernel program memory
    - Control access by each user program to memory of other user programs
    - Each process gets a *private, isolated* address space
    - One process cannot interfere with another processes address space
    - User processes cannot access privileged information (e.g. that used by the OS-kernel)

Example - Virtual to Physical Memory Address



- Diagram – shows how virtual to physical memory address translation occurs for a process in a 32-bit OS
    - Have to run processes A, B and C (scheduled) – want as many as possible to fit into main memory
        - D is suspended in secondary storage
    - LHS – how virtual to physical memory address translation occurs for a process in a 32-bit OS
    - RHS - process D kept in secondary storage, thus this process has still not been scheduled to execute and hence still suspended
    - Processes A, B, and C have been scheduled to execute and thus their virtual memory addresses have been translated to physical memory addresses
        - These processes have been moved from secondary storage (virtual memory) and assigned to physical memory addresses
- Two different parts of the MMU are used in this memory translation and assignment
    - (1) A **virtual memory manager** retrieves processes currently in secondary storage (virtual memory) for transfer to physical memory
    - (2) A **page table** (look-up table in cache memory) translates virtual to physical addresses
        - Labelled Hardware/OS in the figure since this page table is hard-coded into cache memory, only accessible by the OS-kernel

Virtual Addresses

- Each virtual address can be mapped into one (1) of three (3) categories:
  - (1) Physical memory address, by way of Translation Lookaside Buffer (TLB)/Page Table (PT)
    - Translation Lookaside Buffer is a special type of page table
  - (2) Page miss (fault)
    - Memory page that is to be translated (from its virtual to physical address) is not yet accessible to the MMU – as it is still resident in secondary storage,
    - For memory address translation, pages (to be translated) should already be loaded into an intermediatory virtual memory cache
  - (3) Nothing – an invalid memory address

Virtual Memory

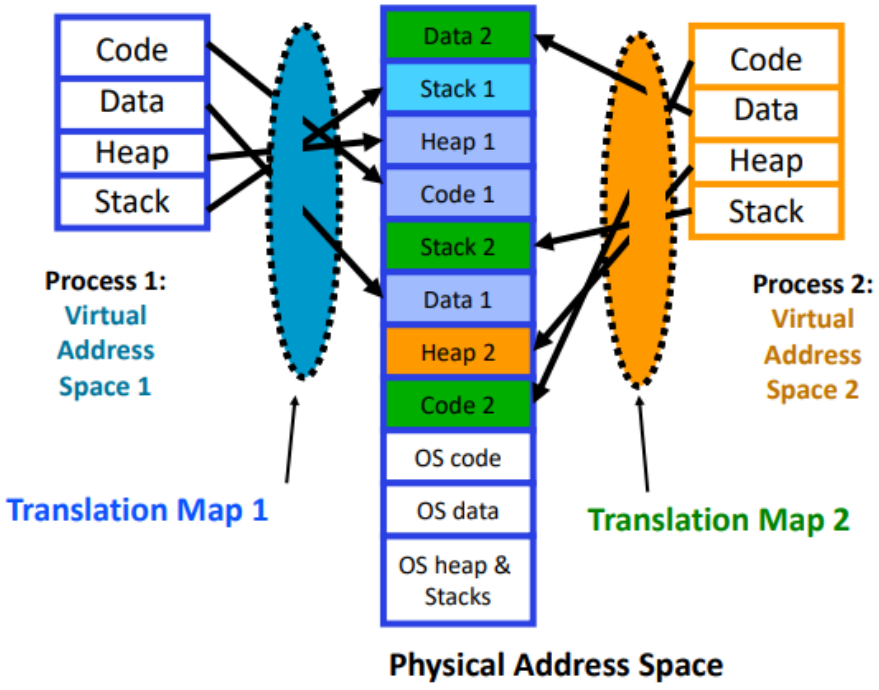
- **Larger logical address space than physical memory**
  - All memory pages addressed in the virtual address space cannot be in main memory
- All addressed pages do not need to be in memory:
  - Full (used) address space is on disk
    - All pages referenced by the virtual address space are actually kept in secondary storage
  - **Main memory used as a (page) cache for running processes concurrently**
    - Bit-by-bit, unless the process uses little memory and can be run very quickly
- Demand Paging: pgs moved to free memory "frames" as needed
  - On demand means when a process is scheduled to run then memory pages (needed for the process to execute) will be allocated to free frames in main memory
  - If there are currently no free frames in main memory, then pages of other (currently running) processes will be *evicted to create free frames*

Paging Memory Approach

- Main point of virtual memory is to make it look like each process has access to all the physical memory that it needs, which thus allows the OS to run many processes concurrently
  - Looks like each process (program) has its *own complete copy of physical memory* (its own copy of address space)
  - This is accomplished by the OS Memory Management Unit (MMU) using demand paging
- Demand paging
  - Memory pages most in demand (for processes that are due to execute) are swapped in and memory pages least in demand (for processes that have been executed for their allotted time) are swapped out
  - Most used pages overall (for high priority processes) are kept in a temporary memory cache for fast access and retrieval
  - Pages that are little used (for low priority processes) are written back to secondary storage
  - Allows N (user and kernel) processes to run concurrently

Swapping Segments into Memory

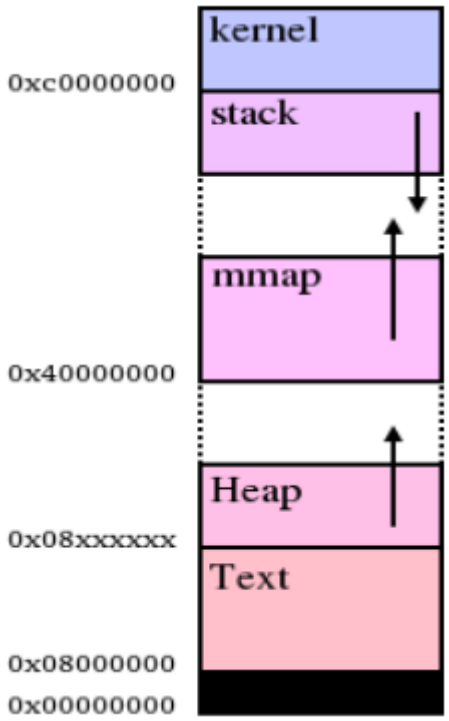
- Context switching: the process of swapping in and out parts of process between virtual and main memory
    - To make room for next process, some or all of a process is moved to disk
    - Allows the OS to keep active, portions of multiple processes in main memory, while the rest of each process is stored in virtual memory (secondary storage)
    - This fine-grained control of what parts of what processes are currently in physical versus virtual memory necessitates the use of paging
- Context switching happens anytime when we **switch between user and kernel operations**
    - i.e. when we need to hand things over to the OS temporarily in order to execute a process (swap it into main memory) or to temporarily suspend a process (swap it into virtual memory)
    - Thus, context switching is an **OS level software interrupt**



- Figure - shows an example of memory swapping for virtual to physical address translation for two processes: 1 and 2
    - Each process is allocated memory that is comprised of several parts: code, data, heap and stack
        - For each process, code is the compiled source code, data is any data (e.g. external files) used by the program (process), the stack is used for the process' static memory allocation and the heap for dynamic memory allocation
    - Translation map positioned between each process and the physical address space corresponds to the mapping obtained via the MMU translating each virtual memory address to a physical memory address (for each process)
    - Physical address space is comprised of memory pages (memory chunks) that correspond to specific parts of process 1 and process 2
        - Data 2 and stack 2 from process 2, and data 1 and code 1 from process 1

- Middle part represents the notion that appropriate memory pages from different processes are swapped into physical memory on demand
  - i.e. as they are needed for the process to execute
- Lower part of the physical address space shows the portion of physical memory reserved for OS kernel-level applications

Process Address Space



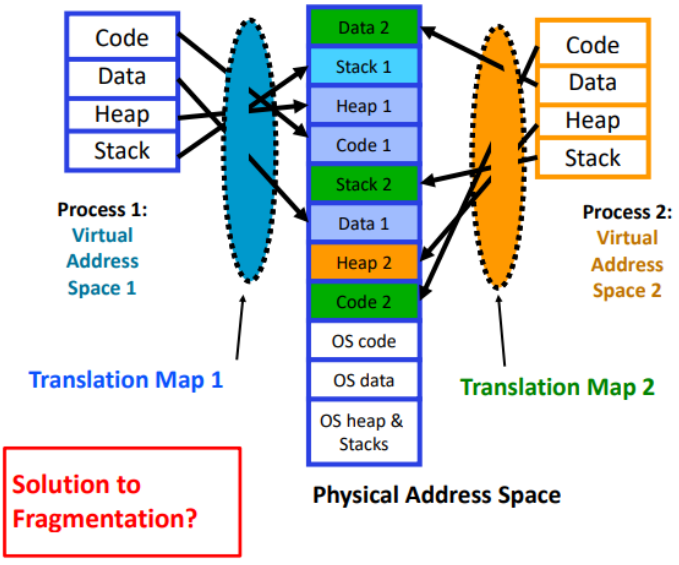
- The design of process address space is such that the
  - Stack: Starts at Max logical address and *grows "down"*
  - Heap: Starts above data and *grows "up"*
- Sparse (null) memory in between
  - Other memory chunks necessary for the process to run (e.g. data and compiled code) are assigned to other parts of the process address space
- Stack: a special region of memory that stores temporary variables created by each function in a program
  - LIFO data structure – managed and optimised by the CPU
  - Every time a program function declares a new variable, it is pushed onto the stack
  - Every time a function exits, all of the variables pushed onto the stack by that function, are freed (deleted)
  - Once a stack variable is freed, that region of memory becomes available for other stack variables
  - Advantage – **memory is managed for the user**, who doesn't have to allocate memory by hand, or free it once it isn't needed any more
- Heap: A special region of memory that is *not managed automatically* for the user and is not as tightly controlled (as the stack) by the CPU
  - A larger, free-floating region of memory

- To allocate memory to the heap a user must use something equivalent to malloc() or calloc() (memory allocation C functions)
- Once memory is allocated to the heap, the user is responsible for using free() (or an equivalent function) to deallocate memory that is no longer needed
- If the user fails to do this, the program will have what is known as a memory leak

Swapping Memory Chunks / Segments

- There are problems associated with swapping chunks of memory between virtual and physical memory
- Must fit *variable-sized chunks* into physical memory
- Frequent swapping between virtual and physical memory is inefficient as it requires an OS-context switch
  - During a context-switch the CPU is temporarily (albeit very briefly) unused
  - Given that we do not want empty (null) memory gaps between the memory chunks swapped into main memory and we do not want any waste of main memory, it is often necessary for the MMU unit to context switch several times so as a suitable number of memory chunks (pages) have been swapped in and **main memory is fully utilised**
- Constant swapping of memory chunks (pages) between virtual and physical memory introduces the problem of memory fragmentation – wasted space
  - External – free gaps between allocated chunks
  - Internal – don't need all memory within allocated chunks.
    - i.e. too much main memory was allocated for the corresponding virtual memory chunk

Fragmentation Solution?



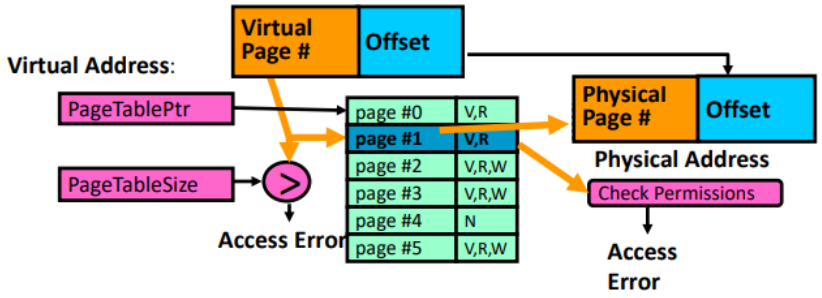
- What is not shown on this figure is that there is invariably wasted space between the process memory chunks assigned to main memory (external fragmentation) and within the memory chunks themselves (internal fragmentation)
  - E.g. internal fragmentation – stack 1 (from process 1) is half of the size that was allocated to it

    - E.g. external fragmentation – gap between data 2 and stack 1 (region of physical memory space)
- One of the **benefits of paging** (the swapping of small chunks of process memory into physical memory as needed), is that **the MMU can have fine grained control** over what parts of process is currently running and have many processes loaded into main memory at the same time (thus running concurrently)
    - So, given that we do not want to give up this benefit of paging, how do we solve the problem of memory fragmentation?

Paging

- Fragmentation is a problem – most common solution to memory fragmentation is paging
- Paging: a memory management scheme by which a computer stores and retrieves data (memory blocks called pages) from secondary storage for use in main memory
    - Important part of virtual memory implementations as it uses secondary storage to let programs exceed the size of available physical memory
- Suitable page-size is an ongoing debate in paging algorithm design
    - Large page sizes can lead to lots of fragmentation and that smaller page sizes (e.g. 1KB – 16KB) lead to less fragmentation overall
- Carefully designing paging algorithms, and choosing suitable page sizes – can avoid internal and external fragmentation

Implementation



- Have mapping between virtual and physical mem addresses with pg table in between
    - For each pg scheduled to run, has a pg # and corresponding vm addresses
    - Pg # looked up in pg table – that pg table entry gives mapping to corresponding physical address
    - Allows pg to be loaded to that pa and executed by CPU
- For each process (that the OS has scheduled to run), a page table resides in physical memory
    - Page table is a data structure that (for each process) maps page numbers (references to memory chunks in virtual memory) to frame numbers (memory slots in physical memory)
    - Column in table – v (valid), r (read access), w (write access)
    - Pg table is as long as there are pgs comprising given process
- Millions of possible virtual addresses but page table only has so many entries (since it resides in physical memory)
    - Given that pages are swapped in from secondary storage for a given process that is scheduled to execute, we have to check whether the page can fit in the page table/not
        - If not, then the MMU executes a page that is already in the page table (translating it to a physical address)
        - Subsequently, the *larger page can then be entered into the page table*

        - Generally, page tables must be large enough to hold as many pages as we would expect to have running (concurrently in physical memory)
- Figure shows each entry of the page table as containing a (physical memory) page and permissions (e.g. read, write, valid) associated with this page
    - The ith entry in the table array gives the frame number in which the ith page (swapped in from virtual memory) is stored in physical memory
    - The offset from the virtual address is copied to the physical address to indicate how large the page is and thus how much memory of memory chunk must be allocated at each physical memory address (frame)

Page Table Entry (PTE)

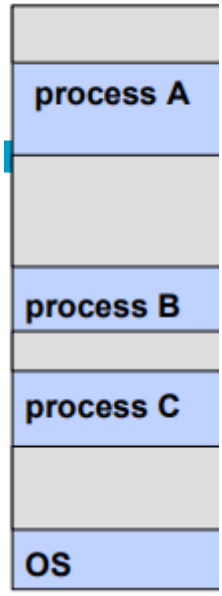
- Page Table Entry (PTE): the ith entry (row) of a page table and is implemented as a pointer to an actual memory page (and for more complicated multi-dimensional page tables: a pointer to a next-level page table)
    - i.e. Each row is a pointer to given pg to a corresponding physical mem address to which mem for that pg will be transferred in order to run
    - Each pte includes a set of permission bits (about the referenced page)
        - These permission bits tell the MMU something about the page
        - E.g. whether the page is read-only, read-write, write-only and valid
- Valid Bit = 1 => the *page is in memory* (i.e. PTE points to a page in physical memory)
- Valid Bit = 0 => page is **not in memory**, and must be swapped in from virtual memory

Page Table Size

- Depends on how much main mem available and how big it is assumed processes can be
    - i.e. how many pgs they can possibly comprise
- Page table needs *one entry per page referenced*
- 1 page table must be kept for each process - main function of the page table is to map virtual page numbers (pages swapped in from virtual memory) to physical frame numbers (specific physical memory addresses for these pages)
- Thus, the page table must be large enough to accommodate all memory pages for all processes concurrently scheduled to run at any given time
    - But page table size is usually calculated using the size of memory
- Assume: 4GB (2^{32} byte) virtual & physical address space
    - Page size: 4KB (2^{12} bytes)
    - So - 2^{32} byte address space must be split into 2^{20} (1,048,576) pages
    - Each PTE contains a physical page (frame) number
    - 2^{32} physical addresses divided into frame sizes of 2^{12} , there are 2^{20} frames (page table rows)
    - Page table must have 2^{20} entries
- Each page table needs one entry per page referenced but the question remains as to how large each page table entry (PTE) must be?
    - Size of each PTE depends upon the size of the page table as determined by the total amount of physical memory available
    - How much info in each row?
        - i.e. how many bits needed to address physical mem and how many is needed to be designated for permissions etc
- E.g. 4GB of main memory

    - 20 bits to store each frame number in the page table
    - In addition to these 20 bits (just to store a page frame number in main memory) each PTE must include 5 additional bits to accommodate the permission bits, valid bit and dirty bit
    - Also 7 unused bits in each PTE to bring the required memory per PTE to 4 Bytes
- The valid bit in each PTE to indicates whether the page is valid (i.e. currently present in memory) or not
    - The dirty bit tells the memory management unit whether the page has been modified by processor or not (since being swapped in)
    - Permission bits: read, write, execute
    - E.g. 4GB main memory scenario
        - 25 bits (+7 unused) = 4 Bytes per PTE
        - Page table size = 2^{20} * 2^{2} (per PTE) Bytes = 4MB
- As we increase the size of the address space then the size of the page table (for each process) increases
    - i.e. Pg table sizes increase with memory size, larger address space allows for larger page tables
- Even Larger Page Tables – assume 64-bit OS (64-bit virtual address space), 4KB page sizes, 4 GB of main memory
    - NB still need one page table for each process currently running
        - Tells MMU which pgs currently swopped out of mem for each process
    - 2^{64} addresses divided into frame sizes of 2^{12} , thus = 2^{52} PTEs
    - Page table size = 2^{52} * 2^{2} (per PTE)
    - Bytes = 2^{54} bytes = 16000000000000000 bytes (16 petabytes per process)
- What can we do about such massive memory requirements for page tables?
    - Solution - multi-level page table

Additional Problems



- Consider many processes running in main memory - memory allocation is sparse
    - Process memory allocation is non-contiguous and memory is wasted

- if there are many processes running concurrently then heap (allocated memory) size increases and page tables need to be maintained (also stored in main memory) for each of these processes
    - The memory wastage problem is exacerbated if memory allocation is sparse
- Problem can be solved by multi-level page tables

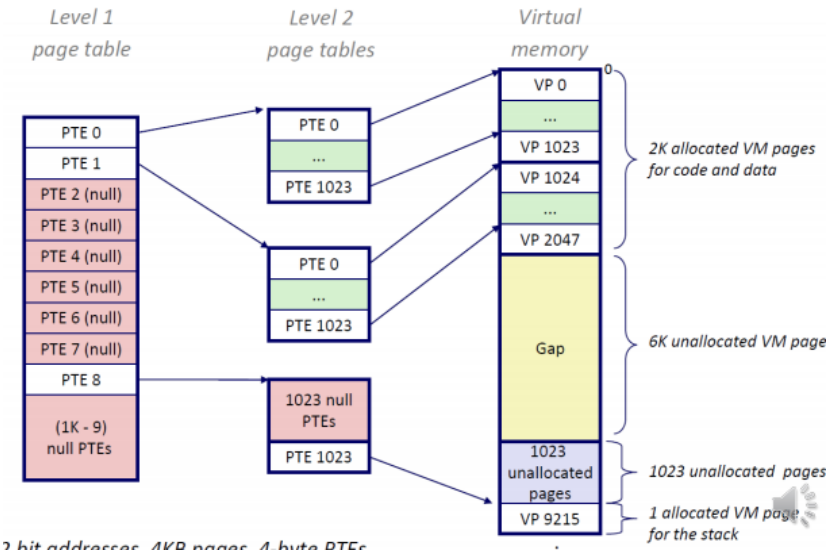
N-Level Page Tables

https://www.youtube.com/watch?v=Z4kSOv49GNc

- Assume: 64-bit OS (64 bit virtual address space), 4KB page sizes, 4 GB of main memory.
- 2^{64} addresses divided into frame sizes of: $2^{12} = 2^{52}$ PTEs.
- Page table size = $2^{52} * 2^{2}$ (per PTE) Bytes = 2^{54} bytes = 16000000000000000 bytes (16 petabytes per process).
- **Paging the page table can lower memory requirements to ~ 4MB per page table (N=4).**

- Each pg table entry points to another pg table – thus, only need to look at as many entries as there are pgs currently in memory
    - Don't need to have each pg table entry accounted for – only need to worry about pgs in main memory
- Consider that for N = 2, only level 1 of the table must be in memory in any time.
- So, for example, a page table entry in level 1 of the table = null, then the corresponding level 2 page table is not needed at all.
    - So, level 2 page tables can be swapped in and out by the MMU as they are needed

Example: 2-Level Page Table

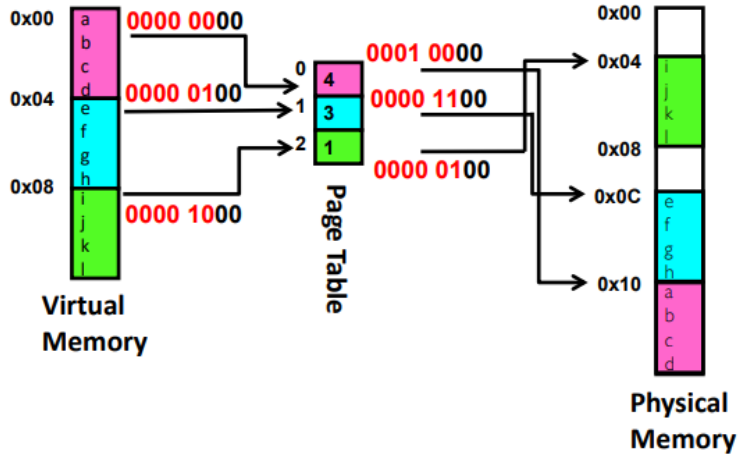


- A 2-level page table reduces memory requirements

- Lvl 1 of the page table is comprised of Page Table Entries (PTE), where each level 1 PTE is a pointer to a level 2 page table
    - Each level 2 PTE is then a pointer to a virtual memory address
- Diagram
    - Only the level 1 table needs to be in main memory at all times
    - Diagram shows many ptes in the level 1 table as being null, and only three PTEs as used (pte0, pte1, pte8)
    - Hence, only three (level 2) page tables need to be stored in memory in order for the Memory Management Unit to map virtual to physical memory addresses in this case
    - If a pte in the level 1 table is null, then the corresponding level 2 page table does not even have to exist
    - Level 2 page tables can be created and paged in and out by the MMU as they are needed
- If a single page table was used then all ptes would need to be allocated memory and overall main memory requirements would be much higher

Pages

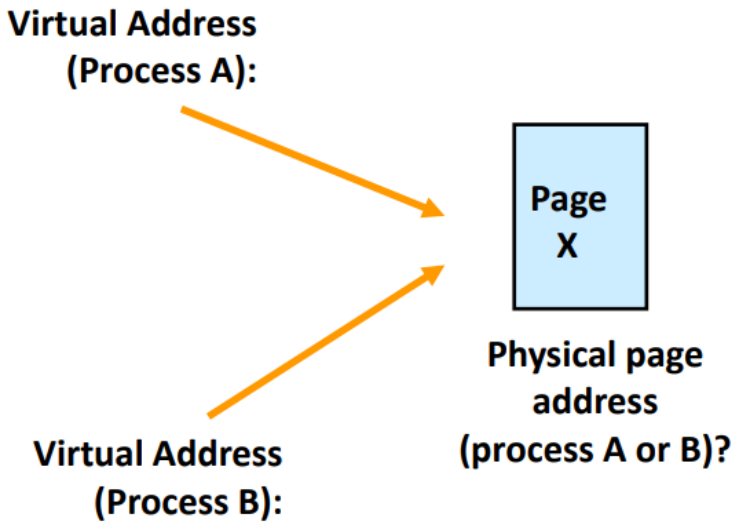
Example: 4 Byte Pages



- Diagram shows a detailed process of mapping of virtual memory addresses to physical memory addresses via a page table
    - Translation from virtual (left-hand side) to physical (right-hand side) memory addresses occurs via adding a 3 − bit off-set to every virtual memory address
    - These offset values (4, 3, 1) are in each entry of the page table and are added just before the (black) permission bits in each address
        - Added 3 (011) and offset 1 (001) to the virtual memory addresses referencing these pages
- First page table entry (purple) includes the offset 4 (= 100 in binary)
    - This offset is then added to the virtual address and this then becomes the physical address that this (purple) page table entry points to (i.e. 00010000)
    - Hence - purple virtual memory page is referenced with binary vector 00000000 (page 0), and the off-set (for mapping to physical memory = 4), (i.e. 0100 in binary

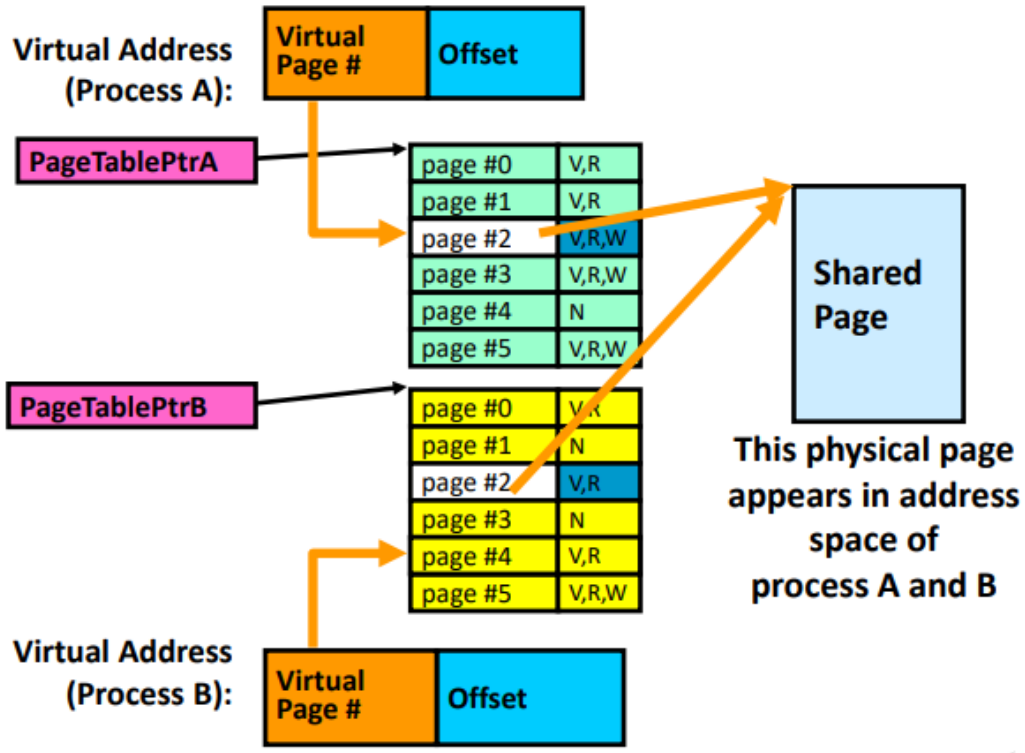
        - Adding this off-set to the virtual page reference we get the physical memory page reference: 000100(00), where the last 2 bits (black) are permission bits
- Virtual memory pages 1 (blue) and 2 (green) to be translated to their corresponding physical memory pages, we must add offset

Memory Page Sharing

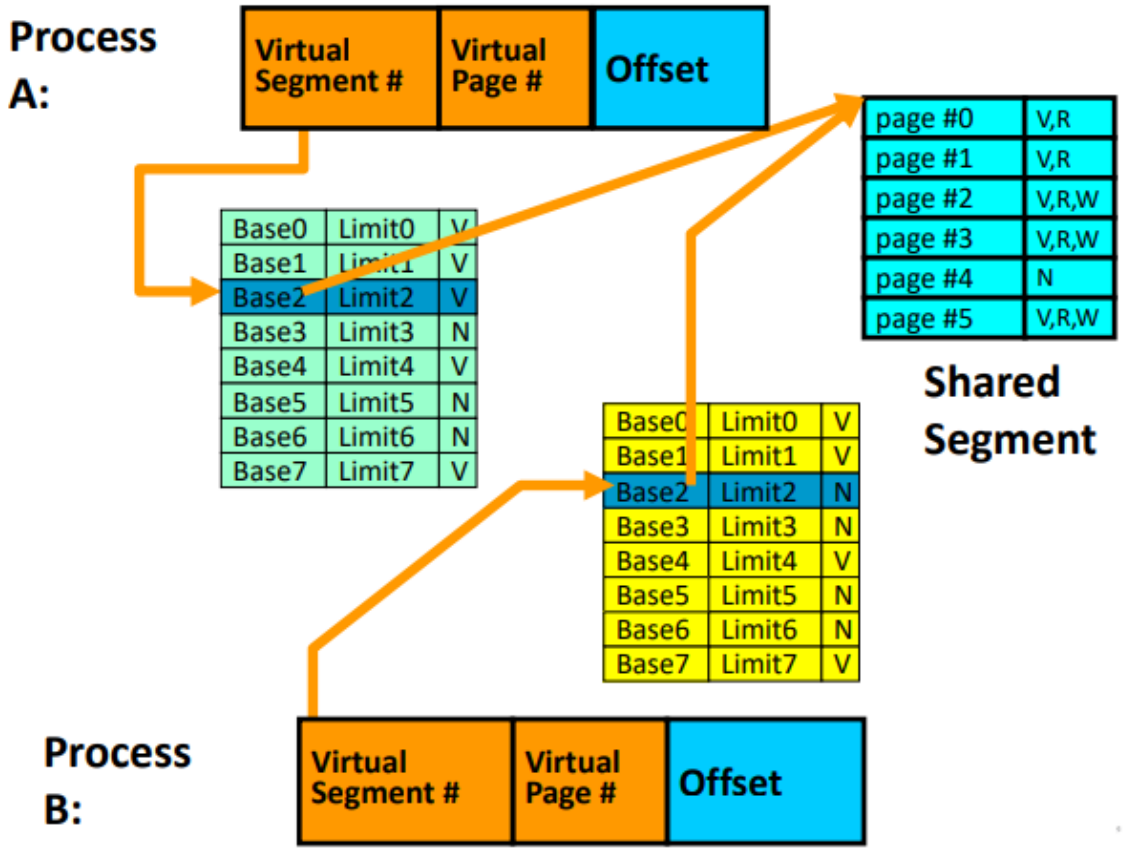


- Figure shows an example of two processes (A, B) assigned virtual memory addresses, where the page table has translated a virtual page address for each process to the same physical page address
- Given that we have limited physical memory available and many processes that are scheduled to run concurrently, we often get the situation when two or more processes try to concurrently access (i.e. are translated to, via the page table) the same page in physical memory
    - In such a case, access to such shared pages is simply controlled by toggling a permission (access) bit in the page table

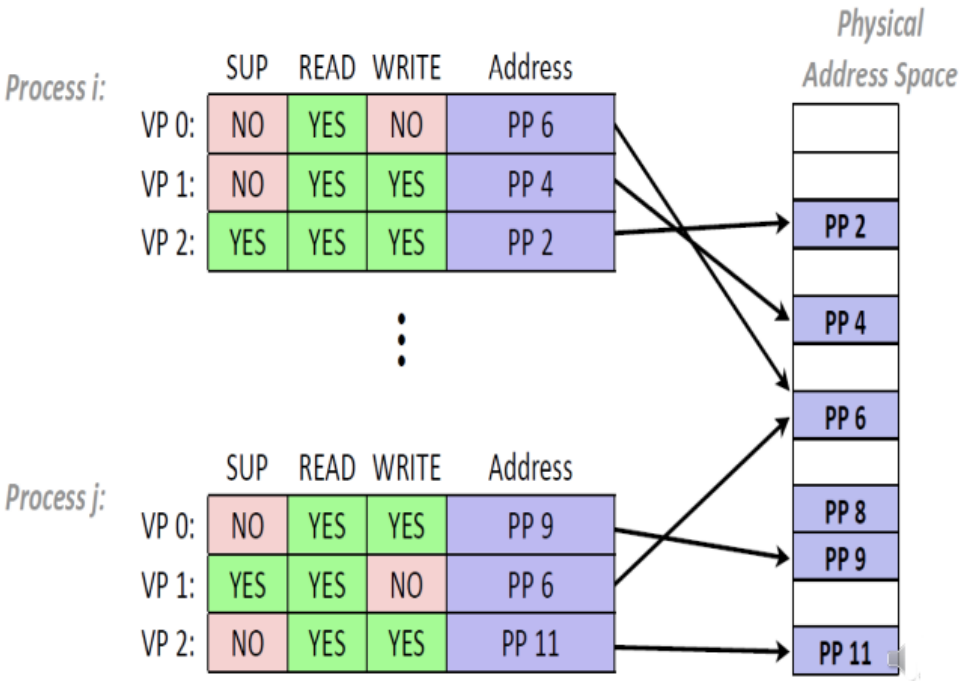
Page Tables and Shared Pages



- Have two tables, PageTablePtrA and PageTablePtrB – corresponding processes A and B
    - In this example, the virtual page 2 in each page table points to the same (shared) page in physical memory
- To allow these different virtual page addresses (in processes A and B) to point to (share) the same page of physical memory, each page table entry has permission bits controlling read-write, read-only or execute-only access
- Process A, page 2 has the permission bits set to: V, R, W (valid, read, write)
    - Process B, page 2 has the permission bits set to: V, R (valid, read)
    - Thus, only page 2 of process A can be written to the physical memory page

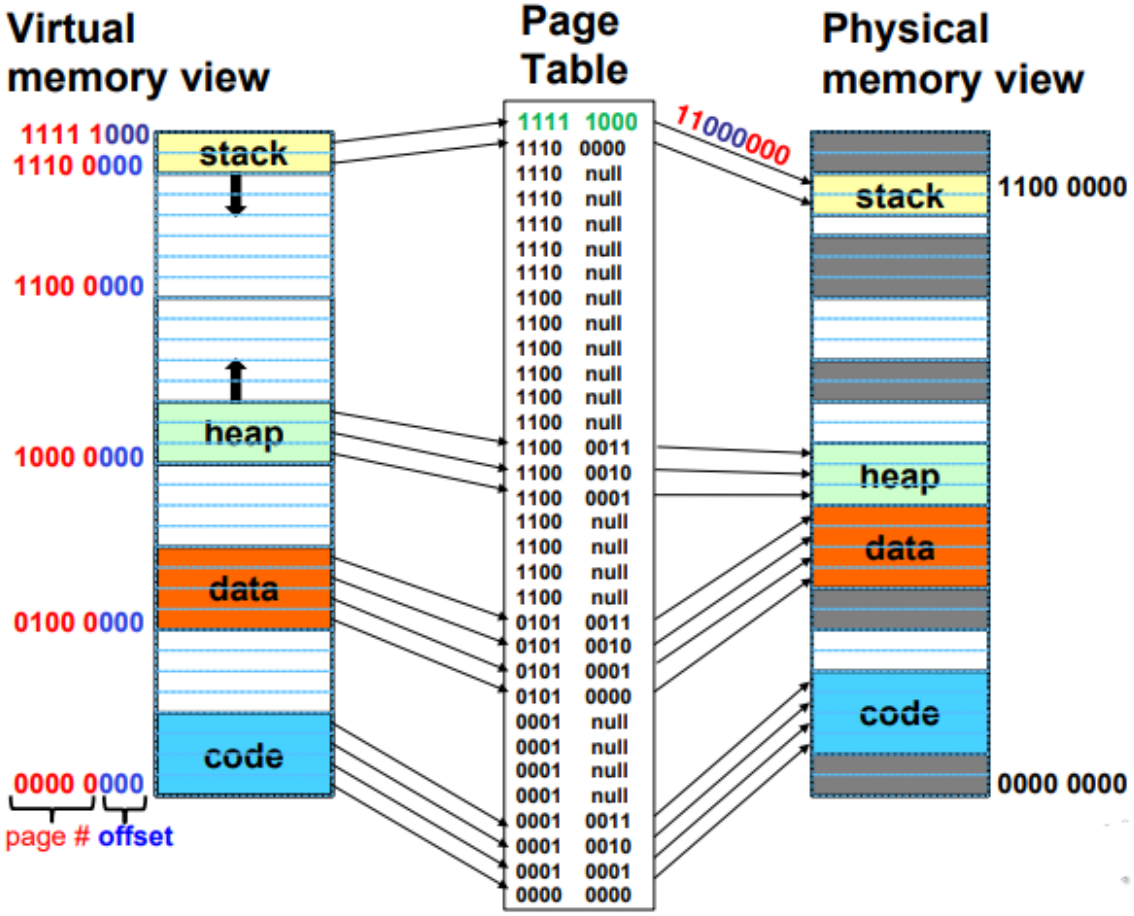


- Page tables associated with different processes can also share sets of pages (segments)
- Figure - shows a page table for each process (A and B)
    - Third page table entry for process A (Base2) points to page 0 in physical memory, as does the third page table entry for process B – both pointing to same set of pages
    - Each page table entry now has a Limit2 entry as the second column
    - Value in this second column specifies the size of this memory segment that the given page table entry points to (in this case, six pages: 0 . . . 5)
    - Notice that for process A – valid bit set (3rd column)
        - Process B – set to N
        - Indicates that there is another process sharing this segment – so process B can't access it

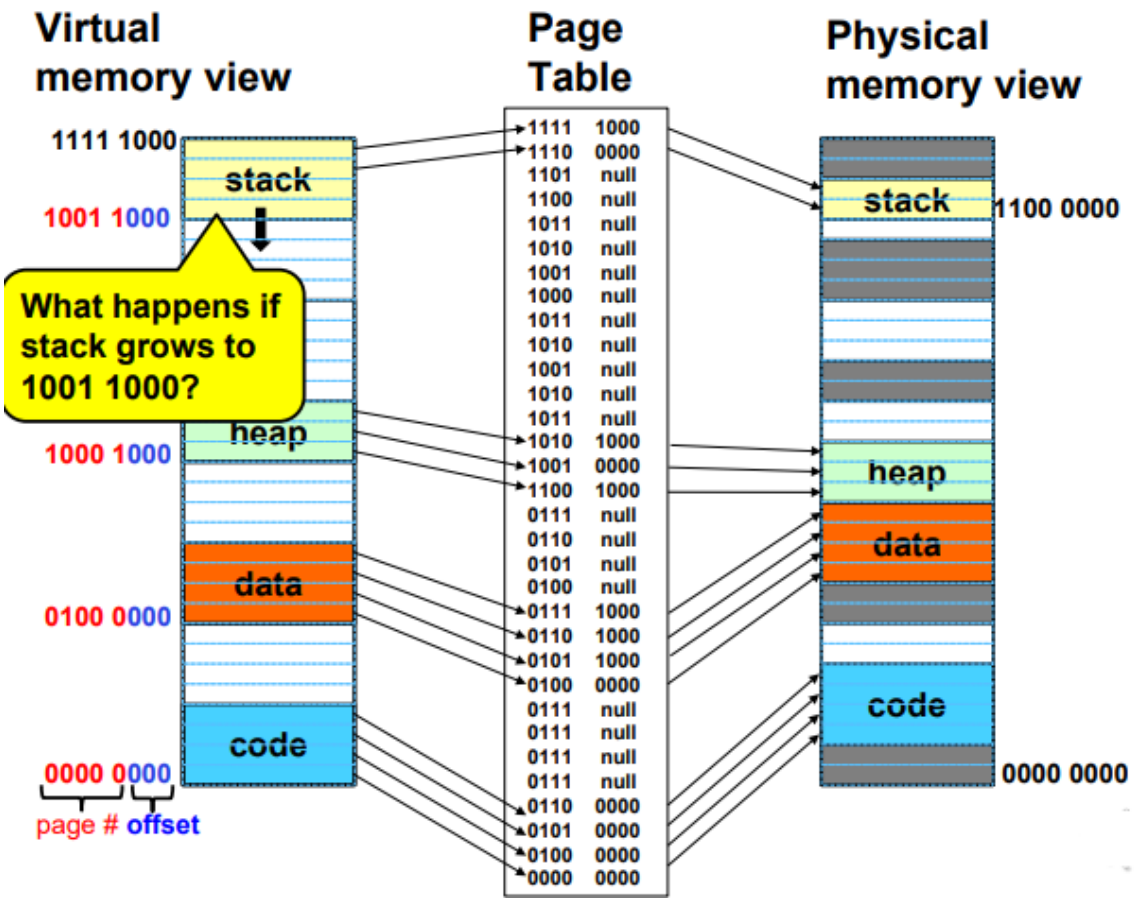


- One of the key requirements for virtual memory to work is protection, i.e. effectively and efficiently allowing multiple processes to safely access a limited physical memory space
    - i.e. Multiple processes needing to access same physical mem addresses
    - Key idea is that some parts of the processes (pgs) can be suspended when they are swopped out of physical memory
- The page table for each process may also contain information about whether the memory pages (comprising the given process) are suspended or not (SUP)
    - SUP flag (bit) can be used in addition to the permission access bits for a given page, occupies the first column of each page table entry, and acts as quick page accessibility check
- Diagram
    - Virtual pages comprising process i and j, both share six pages in physical memory (P P2, P P4, P P6, P P8, P P9 and P P11
    - MMU has effectively allowed shared access (by processes i and j) to these memory pages
        - Done **not** via suspending the virtual pages VP0 and VP1 of process i and the virtual pages VP0 and VP2 of process j, but suspending virtual page V P2 (process i) and virtual page VP1 (process j)
    - For example, suspending virtual page VP1 (process j) allows physical page memory (PP6) to be accessed by process i

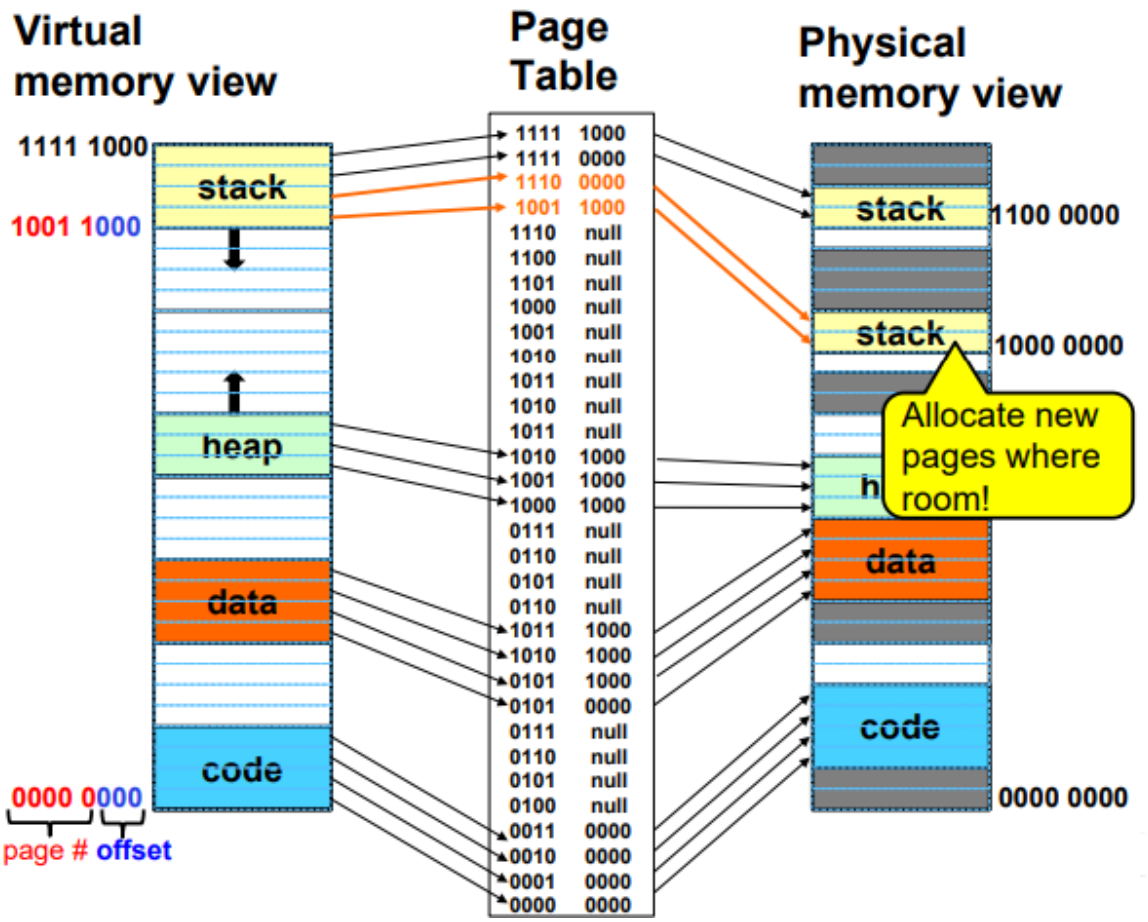
Example: Memory Mapping with a Page Table



- Individual page-table entries (centre of figure) that translate a virtual memory addresses (left-hand side) to physical memory addresses (right-hand side), using the 3 – bit offset (000)
- The process stack is composed of memory chunks corresponding to different parts of the compiled (executable) program: stack, heap, data, code
    - LHS shows process which is comprised of above parts
        - Size of pages depends on complexity of running process
        - Each page has virtual address i.e. bit string address space
- Page table does mapping/translation to corresponding physical bit address space
    - Page can be loaded into main memory



- What happens (i.e. how is virtual to physical memory mapping done?) when the process stack in virtual memory grows (e.g. when a new process is scheduled to run)?
    - Need to allocate more pages in order to run process – so need more virtual mem addresses (must be translated to physical addresses), so just need more entries in pg table to be filled (to account for it)



- Process stack in virtual memory has grown (downwards) to use a total of four memory pages
  - This increase in number of virtual memory pages necessitates that two more pages must be allocated in physical memory if the process is to continue running
  - The exact addresses of these newly assigned pages in physical memory depends upon the offset used to translate (map) the virtual to physical addresses
- An OS design challenge is to make page table size (allocate enough memory initially to store page tables) such that the number of page table entries will equal the number of pages in virtual memory at any given time

Page Tables: Summary

- Memory divided into fixed-sized chunks of memory
- Virtual page number from virtual address mapped through page table to physical page number
- Offset of virtual address same as physical address
- Each process has its own page table (stored in kernel address space, main memory since MMU uses it to lookup mapping of pg scheduled to run)
  - Translates (maps) virtual pages (making up all memory of a given process) from virtual page addresses to physical page addresses
  - Mapping is done for each page via looking up the translated address in the page table entry corresponding to a given virtual page
  - Translation from virtual to physical page addresses is usually done by adding an offset value (given in the page table entry) to the virtual page address

Virtual Memory: Fundamentals

Terminology

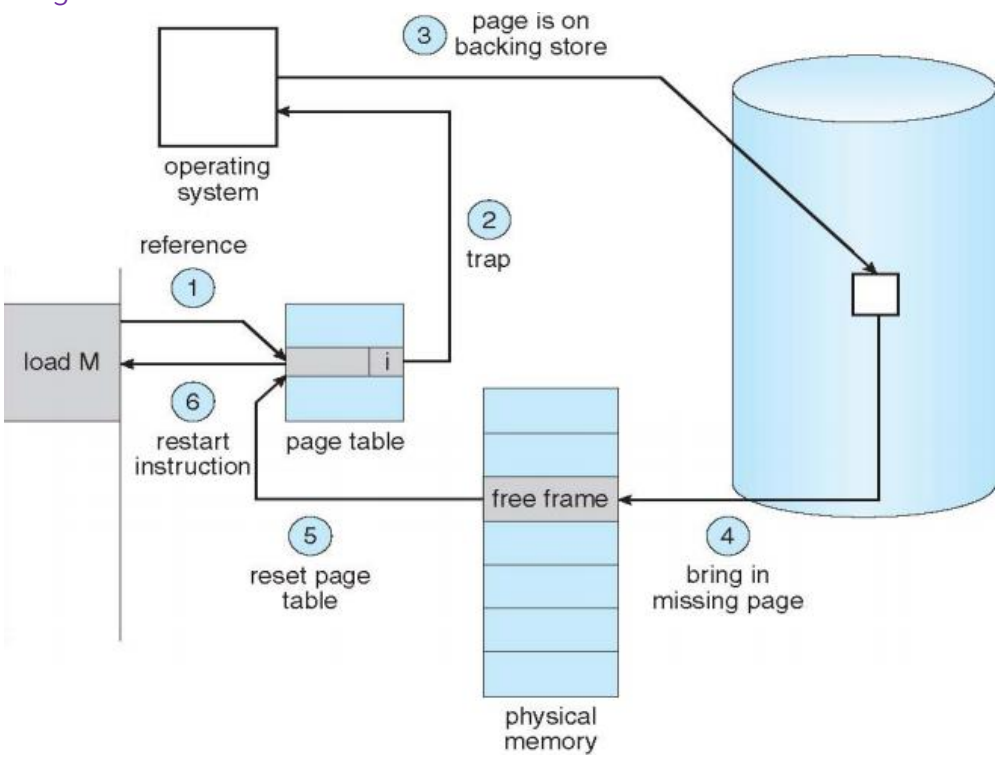
- Demand paging: Load pages into memory only when page faults occur
- Replacement strategies: What to do when there are more processes than can fit in memory
- Load control strategies: How many processes can fit in memory at one time

Page Fault

https://www.youtube.com/watch?v=bShqyf-hDfg

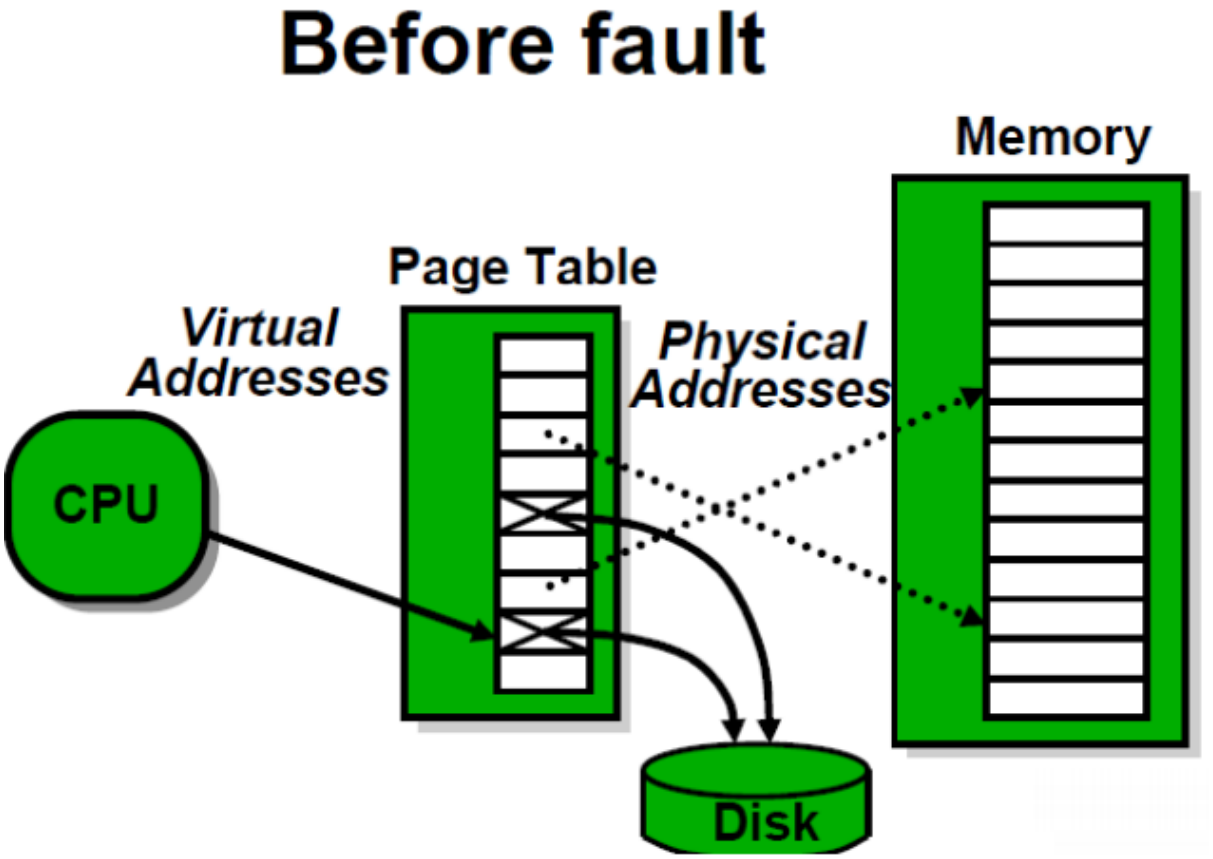
- Page fault/miss: occurs when a page for a given process is to be mapped by the MMU from a virtual to physical memory address, but the given page is not currently loaded into a physical address space
    - The process page virtual address may be accessible to the process in the virtual address space, but the memory page or the mapping for this given page (from virtual to physical addresses) has not yet been added to the process page tables
    - Thus, it's usually necessary for the actual page (memory contents) to be loaded from secondary storage
- **Page faults are detected by the MMU**
    - Exception handling for page faults is done by OS-kernel level apps
    - When handling a page fault, the OS tries to make the required page accessible and map it to physical memory
    - Otherwise if the required memory page cannot be accessed the process is terminated and an **illegal memory access error** thrown
- Trap: an exception or a fault, where the OS temporarily switches to kernel–mode, performs some fault-handling action before returning control to the originating process (being run when the exception was thrown)
    - Examples – breakpoints, division by zero, and invalid memory access
- Process references page with invalid PTE?
    - MMU traps to OS
    - Resulting trap is a page fault
    - While pulling pages off disk for one process, OS runs another process from ready queue
    - Suspended process sits on wait queue

Diagram



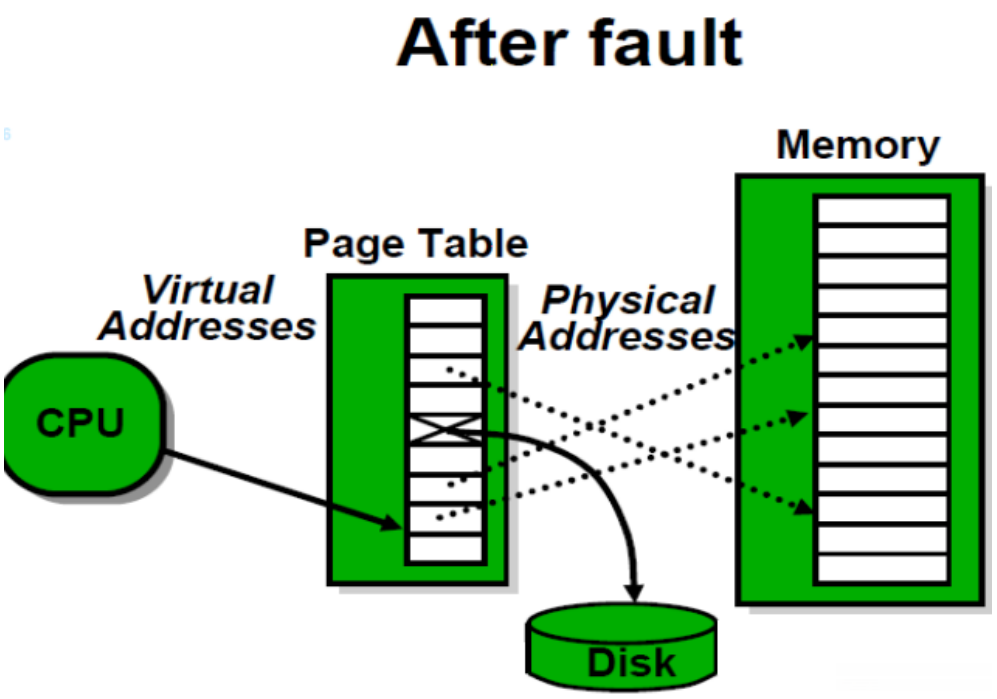
- Steps taken by the OS to handle a page fault:
    - (1) when MMU references a page table, then the referenced virtual memory address is either **already loaded into memory** (i.e. accessed via a temporary memory cache used by the page table to store frequently used pages) or is otherwise **not in memory**
        - e.g. the MMU tries to do the mapping from a virtual to physical address but there is no accessible memory page
        - (2) In the latter case the OS elicits a trap
- In the case of a trap, the missing page is pulled from secondary storage and then mapped by the MMU into a physical memory address
    - e.g. the cache memory attached to the page table
    - (4 – 5) Then *loaded into a main memory address* (M) as defined by the mapping in the given page table entry
- (6) Once loaded into physical memory the OS exits from kernel control (released from the trap), and the process continues executing from the point where the page fault occurred
    - i.e. the missing page just loaded into main memory is executed by the CPU

Example: Before Page Fault



- Figure shows a virtual to physical memory mapping scenario before a page fault
- The pages in page table marked with X are next to be loaded into main memory and executed by the CPU
    - In the case of both of these X pages, there is currently no memory page already loaded for that entry of the page table
    - Meaning that the virtual to physical mapping given in the page table entry will fail (since there is no actual memory page to load into main memory)
- Thus, when a specific (missing) page is scheduled to be executed by the CPU (e.g. the second from bottom page), then a page fault is thrown and the missing page must first be loaded in from secondary storage (i.e. Disk)

Example: After Page Fault



- Figure shows the case after the page fault has been thrown, and subsequently handled by the OS
- The missing page (second from the bottom) has now been loaded in from secondary storage and subsequently mapped (dotted line between page table and physical memory now shown) into physical memory for execution by the CPU
- Another page fault will occur when it is turn of the other X page (referenced in the page table) to be executed by the CPU
    - Because this page also is still stored on secondary storage and thus the virtual to physical address mapping specified in the given page table entry will fail

Page Fault Causes

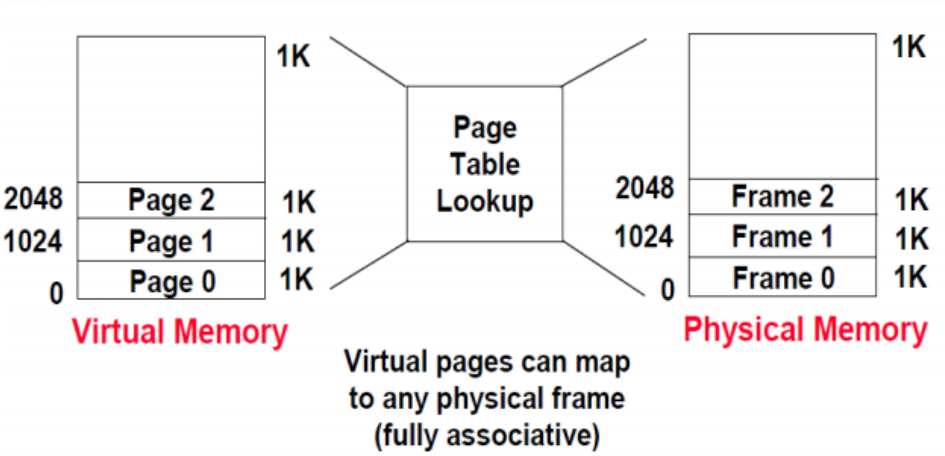
- 3 Types of page faults:
    - Compulsory faults, capacity faults and policy faults
- Compulsory Faults – this is the most common form of page fault
    - The **required pages** (referenced in the page table) **have not been paged into memory**
        - i.e. they are currently not in main memory already or they are not in any physical memory (e.g. cache memory attached to the page table)
    - A typical way to avoid such page faults is pre-fetching – frequently used pages are loaded into cache memory, accessible by the page table
    - Speeds up access to frequently used pages that need to be temporarily swapped out of main memory
- Capacity Faults – these faults occur when a page is scheduled to be loaded into main memory (for execution by the CPU), but **main memory is full of other pages** (currently being executed by the CPU)

    - In this case, the page to be executed must be swapped in, but another page must first be swapped out
    - The decision of which page to swap-out is made by the page replacement algorithm that is run by the MMU
- Policy Faults – in this case, the page fault is caused by the page (previously loaded into main memory) **being prematurely swapped out of main memory** because of the replacement policy (algorithm)
    - A given page was scheduled to run by the CPU and since being loaded into main memory, the page was swapped out by the MMU page replacement algorithm to make room for another page (scheduled to run just prior to this given page)

Page Fault Handling

- Just like thread scheduling – OS must "schedule" memory resources
- 3 Important Questions:
    - How to map virtual to physical addresses?
    - How do we find a page table and page?
    - What is the page replacement policy?
- Effective and efficient scheduling of pages means that the **CPU should be fully utilised** (maximum number of pages loaded) - all pages for all processes should receive roughly equal time in memory (fair access to CPU runtime) and some processes will have their pages run before others (those with higher priority)
    - Page replacement policy – **utilisation, fairness, priority**
    - All pages for all processes should receive roughly equal time in memory (fair access to CPU runtime)
- This scheduling of page resources is done by the *page replacement policy* (an algorithm run by the MMU)
- Page fault handling makes use of three core implementations in the MMU:
    - (1) virtual to physical memory **address mapping**
    - (2) **Page tables** - where page table entries use these address mappings for specific pages
    - (3) A **page replacement policy** - algorithm that decides what pages get to be loaded into main memory and what pages must be swapped out
- For now, page replacement algorithms assume that the MMU is using a fully associative memory mapping
    - i.e. Any page in virtual memory (addressed in virtual memory space) can be translated (mapped) to any frame in physical (main memory) – excluding those frames exclusively dedicated to running kernel-level applications

Memory Mapping: Fully Associative Mapping

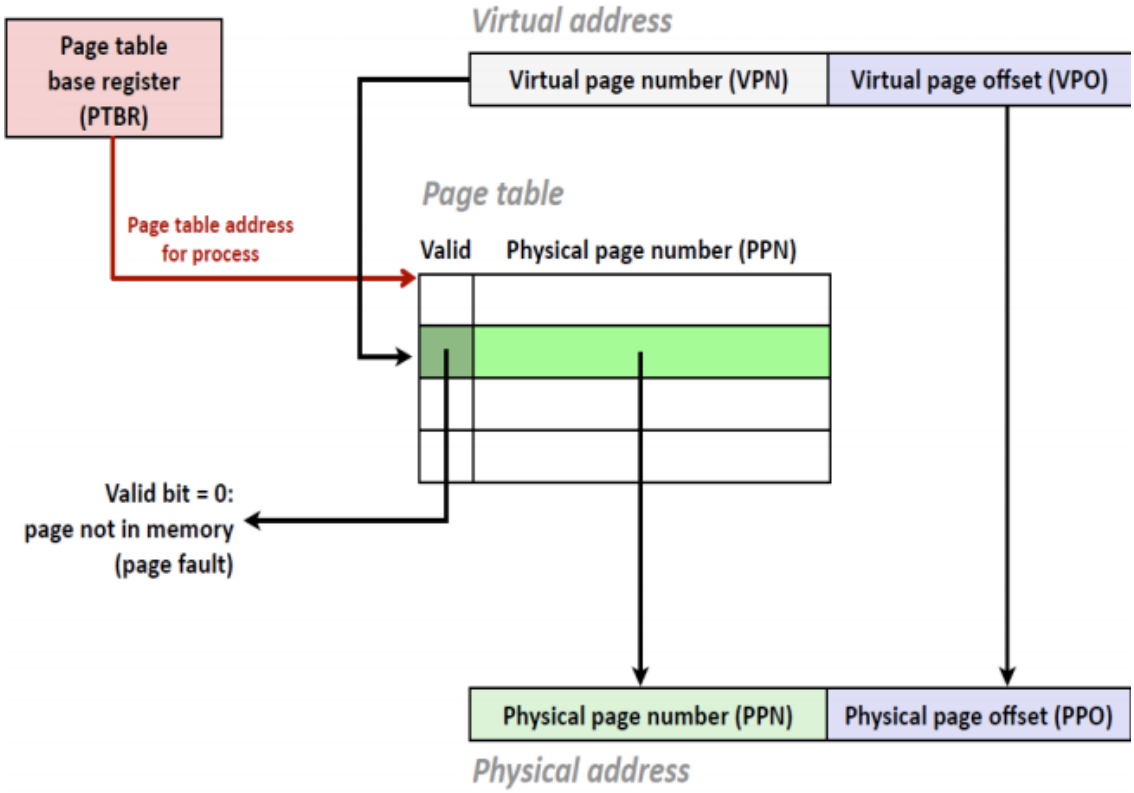


- Any page in virtual memory can be translated to any frame in main memory
    - Addressable space in main memory is broken down into frames – *pages are assigned into those frames*
    - Virtual address space has pages 0 to N – each page is a certain size and is assigned a row in the page lookup table
    - Every page in vm can be put into a page lookup able – that's where mapping to physical mem occurs
        - Page table shows whether that page is in a particular frame in physical memory or that it's not in physical memory
            - For the latter case, it must be retrieved from secondary storage
- Diagram:
    - LHS: pages addressed in virtual memory space
    - Centre: mapped via looking up the page number in the page table
    - RHS: physical memory address frames that pages can be slotted into

Page Fault Handling II

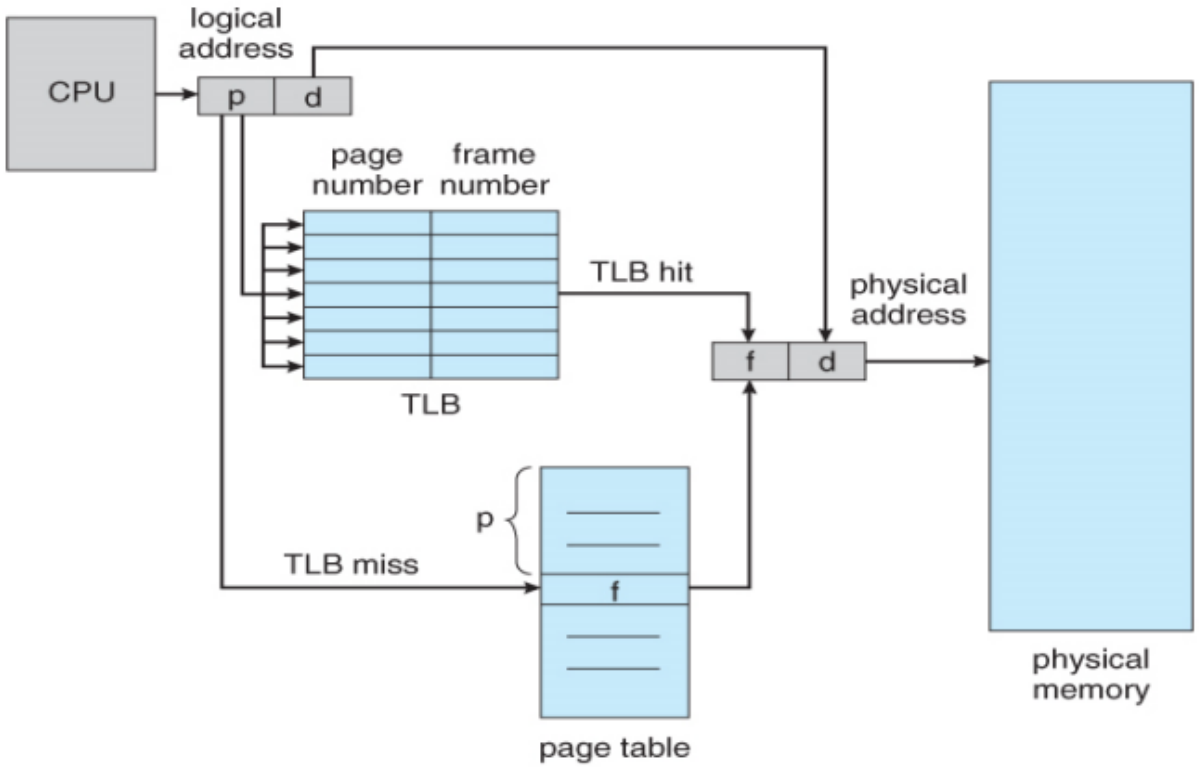
- How do we find and load pages?
- Have a page table that translates memory pages from their virtual to physical addresses – next issue is for the MMU to locate the memory page and load it into physical (main) memory
    - This location is done with the **Page Table Base Register** (PTBR) – tells the MMU whether the page is already in memory or not
    - If the page is not in memory, then the MMU proceeds to look in the Translation Lookaside Buffer (TLB) - a special memory cache for storing frequently used pages
        - Or otherwise the page will be loaded from secondary storage
        - TLB is cache memory associated with page table

Diagram 1



- Diagram shows how a memory page is located by the MMU and loaded into main (physical) memory
    - First column of page table is PTBR – tells MMU whether page is valid/not
    - The PTBR is how the MMU knows where to find the page table
    - That is, the valid bit in the first column of the page table tells the MMU whether the page is already loaded into main memory or whether it must look elsewhere to load the memory page
    - If the page is already in main memory (valid = 1), then the page is accessed at its physical memory address – given as a physical page number (memory frame) in main memory
        - Page is then run
- If the page is not already in main memory (valid = 0), then this means there is a *page-fault*
    - When trying to load a page, the *MMU first looks in the Translation Lookaside Buffer* (TLB) where frequently used pages are stored
    - If the page is not in the TLB, then the MMU looks for the page in *secondary storage* (this is where most pages addressed in virtual memory are stored)
    - Once retrieved, it will be swopped in, switched to valid – can then be run
- Virtual page offset (physical page offset) is simply the value added to the physical page number in order to get the physical address at which the page will be loaded

Diagram 2



- Figure shows detailed memory page look-up using the Translation Lookaside Buffer (TLB)
    - CPU tries to run a page reference – **MMU looks in TLB**
    - Shows MMU first checking the TLB rather the page table, since pages stored in the TLB are those most frequently used, and the TLB is very quick to access
        - Thought the MMU implementation could just as easily have the page table accessed first and then TLB, and then finally secondary storage, if the page is not already loaded into main memory or in the TLB
- MMU first checks the TLB – that's when the logical (virtual) address of a page is accessed, then the MMU immediately checks the TLB for the corresponding page (since the TLB is cache memory the lookup and access time for the TLB is extremely fast)
    - If page is valid – retrieves frame number for that page number
        - i.e. a TLB hit
    - If the required page is not in the TLB then this is flagged as a TLB miss – meaning the page is not in the TLB
        - **Have to look up in page table** – page table then performs address mapping between virtual and physical
        - Page is loaded into physical memory – and is assigned a frame number and run
        - TLB bit for that page set to 1 – indicates that it's valid (loaded into main memory)
    - MMU then proceeds to check the page table to see if the page is already loaded into main memory – if so, then the corresponding physical address of the page is accessed, otherwise MMU must load the page from secondary storage
- What is the page replacement policy?

    - i.e. what pages (for a given process that has been scheduled to execute), should be swapped (loaded) into main memory next, and what pages should be removed (swapped-out) from main memory in order to make way for new pages that must be swapped-in
- This decision is made by the page replacement policy (algo)
    - FIFO (First In, First Out)
    - LRU (Least Recently Used)
    - OPT (Optimal)

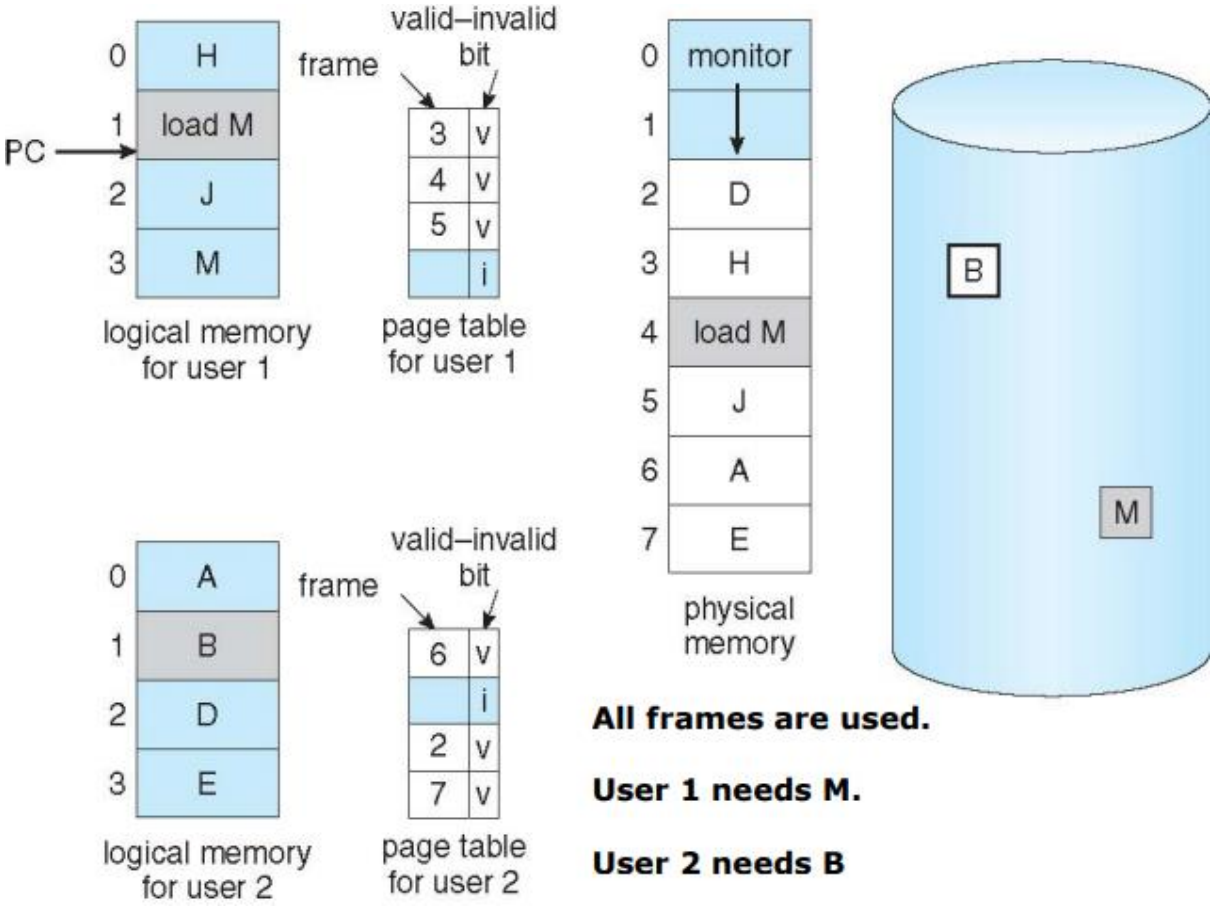
Page Replacement Algorithms

- Goal: **Want lowest page-fault rate** on both first access and re-access
    - Keep core set of frequently used pages in memory
- Page replacement algorithms (policies) run on strings of memory page references (sets of virtual memory addresses for pages)
    - These page references are just pointers (page numbers) to actual memory pages that are stored somewhere in memory
- If some page X is already loaded into main (physical) memory and repeatedly accessed (run by the CPU), then there is no page fault
    - Page fault only occurs if the given page (X) is not already loaded into physical memory and ready for immediate execution

Basic Page Replacement

- Page replacement algorithm is run only when a page fault occurs
    - i.e. when a page referenced in the virtual address space is accessed but this page is not already loaded into physical (main) memory
    - It is then the task of the page replacement algorithm to find an already loaded page to replace
- (1) **Select and find (memory) location of the desired page**
    - Select a page from a string of referenced pages (pages scheduled to be run), then retrieve the page from its memory storage, and find a free frame in physical (main) memory
- (2) **Find a free frame:**
    - If there is a free frame – use it
        - Slot memory page into it and run it
    - If no free frame: use page replacement algorithm
    - Select a victim frame
        - All memory frames in main memory have been allocated to pages, then an already allocated page must be evicted from its frame (victim frame) so as the selected page can be loaded
        - The selected page is then loaded into this victim frame and the evicted page is swapped-out to a temporary memory cache (translation look-aside buffer) or to secondary storage
- (3) **Bring the desired page into the (newly) free frame; update the page table**
    - After the selected page has been loaded into main memory, the page table is updated so as to reflect the mapping (translation) between the virtual address of this newly loaded page and its physical memory address
- (4) **Restart instruction that caused trap**
    - OS exits from kernel mode and continues from where it was interrupted (by the page fault trap)

Page Replacement Policy Diagram



- Figure illustrates the page replacement process
- Two processes (for two users) currently scheduled to run by the OS
    - Each process is composed of four memory pages, where a page table (for each process) maps virtual page addresses to physical (main) memory addresses
- Page table for each process shows which page frames in main memory are currently occupied (valid), indicated by the valid–invalid bit in the page table, where v indicates the given page frame is valid, i indicates the given page frame is not occupied (invalid)
    - Valid => indicates page is currently loaded into a frame in physical memory
    - Invalid => no page currently loaded into main memory for that row of the page table
- Page replacement algo must decide which pages are in physical memory at any time, and which pages to remove so as to allow either process to run all the pgs it needs to run
- RHS figure shows the memory pages currently loaded into each physical memory frame
- Far RHS figure shows two pages currently in secondary storage:
    - Page B, to be used by process 2
    - Page M to be used by process 1
- Memory page M for process 1 (user 1) is selected as the next to be loaded into main memory
    - Pg table has assigned this page to memory frame 4 in main memory, though in order for this to happen the current page allocated to memory frame 4 (page B used in process 2) had to be swapped-out first

- Hence, memory page B was swapped back to secondary storage and the reference to memory frame 4 in page table 2 (for process 2) was updated via making the valid – invalid bit for row 2 of the page table = i
  - Meaning there's no longer a reference to physical memory frame 4 and thus no longer any virtual to physical address mapping for page B

Page Eviction

- Which page to evict?
  - Given that a page has been selected to be loaded into physical memory, but all frames in physical memory are full (already have pages loaded into them), it is the task of the page replacement algorithm to select a victim frame and thus remove (evict) the page from that frame
- Goal of any page replacement algorithm:
  - **Reduce fault rate by removing "best" victim page**
  - **Best page to evict is one never to be referenced again**
    - Ideally the page replacement algorithm selects pages to evict that are unlikely to be used again in the near future (and thus not immediately swapped back in again)
    - Minimise page fault rate – CPU will keep referencing pages already in main memory
- Analysis of page replacement algorithms:
  - Assumes the use of a fixed size string of memory page references (virtual addresses) which are necessary to run (load into main memory) in order to complete the execution of a user process (where all the pages referenced make up the total memory for this process)
  - Assume a process pages against itself (local page replacement)
  - Using a fixed number of page frames
    - Page replacement algorithms assume that main (physical) memory contains a fixed number of frames
- Goal of the design and analysis of any page replacement algorithm is to design it such that for any given process (number of pages), and number of physical memory frames, the algorithm will on average replace pages such that the number of page faults is minimised

Random Page Replacement

- A **random frame in physical memory is selected** and the page in this frame removed (evicted), and the evicted page replaced with the page next scheduled to run on the CPU
- Simple, easy to implement
- Unpredictable and does not account for the basic goal of all page replacement algorithms (to minimise the number of overall page faults), and the random behaviour of this page replacement does not fit with the principle of locality (important for the design of effective and efficient page replacement algorithms)
- Principle of locality: the tendency of the CPU to access the **same set of memory locations repetitively over a short time period**, since these memory locations typically correspond to a few processes that the CPU is scheduled to execute during the given (short) time period

Page Replacement: FIFO

- **Allows every page to spend the same amount of time in a physical memory frame**
  - Enabled by the first−in, first−out implementation, which ensures that the page that has been loaded into a memory frame for the longest will be the page swapped-out when a new page is to be swapped-in
- Main advantage:
  - Easy to implement with just a single pointer to the list of frames in physical memory (pointing to the page that has been in main memory the longest)
- Main disadvantage:
  - Doesn't account for one of the goals of all page replacement algorithms, which is to keep frequently used pages in main memory and only replace infrequently used pages in order to minimise the number of page faults
    - That is, the page that has been in main memory the longest could be a frequently used page
  - *Doesn't speak to principle of locality* – core set of heavily used pages will still be swopped out by this algo
- *FIFO assumes that if a page has been occupying memory for a long time then it is the safest to replace*, where in reality that simply isn't the case
  - Where FIFO fails is that statistically, if a page has been called frequently, it's more likely to be called again than another page which has been called recently
  - In other words, frequency is a far better determiner of page loading than age

FIFO Algorithm Example

- Reference string (N = 20):

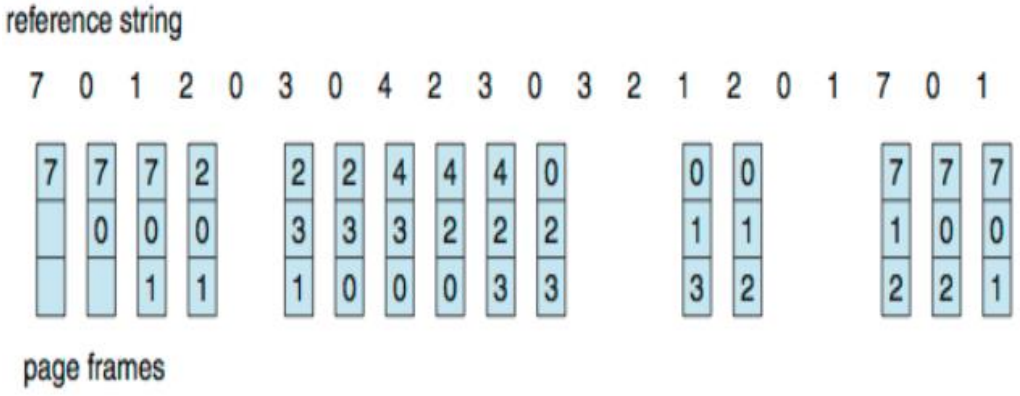
7, 0 , 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

- **Frames = 3**

- **i.e.**: 3 pages can be in memory at a time (per process).

- *How many page faults?*

- FIFO algorithm applied to a reference string for 20 pages
  - Each value in the string indicates a page number
- Page reference string accounts for all pages used by one given process, where there are only 3 frames available in physical memory
  - i.e. only 3 pages can be loaded in main memory concurrently
- FIFO algorithm maintains a linked list of all pages which keeps the order in which the pages entered memory
  - Hence, when a page is to be replaced, the page at front of list is replaced
- Effectiveness of FIFO algorithm can be determined by calculating the number of page faults for this given reference string of 20 pages

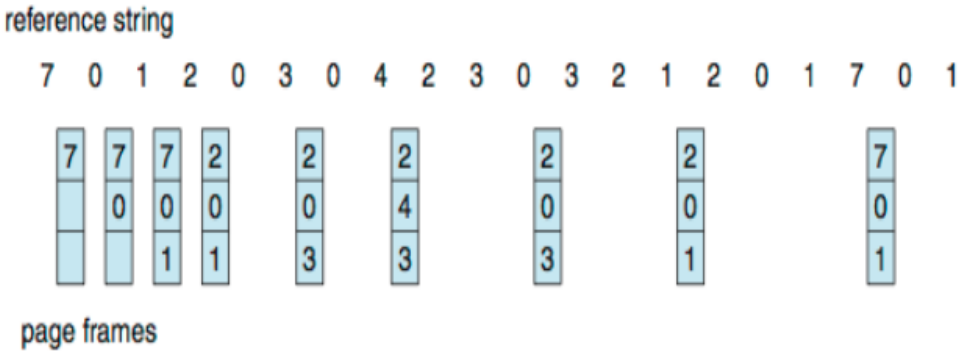


- FIFO algorithm run on this page reference string results in 15 page faults
- Number of page faults can be counted (from left to right) by simply looking at the page number in the reference string and checking if that page number is already loaded into a memory page frame below
- If the page is not in a frame, this causes a page fault, and the page is placed into an empty frame
    - If there is no empty frame then the page that has been loaded into a frame for the longest is removed (evicted)
- In this example diagram – note that when there is no page fault there is no need to update the pages already loaded into the physical memory frames (hence there is blank space in these cases)

Page Replacement: OPT

- Optimal Page Replacement
    - Not something that can be implemented for page replacement in any real-world OS, but is rather used as a bench-mark algorithm for the design and testing of other page replacement algorithms
- OPT algo assumes that the **page that will not be used for longest period of time will be the page selected for replacement**
    - However, in the process scheduling and running of any real-world OS, we cannot know what pages will be used next or in the near future (since we do not know what processes the user will run at any point in time)
- For page replacement algorithm design and testing, if we assume perfect knowledge of the future (i.e. we know what the next N pages to be scheduled to run by the OS will be), and we *derive a page replacement algorithm that yields a task performance (minimising page faults) within 5% of the OPT algorithm* (for any given reference string size and number of frames), *then this is deemed to be an effective page replacement algorithm*

OPT Algorithm Example



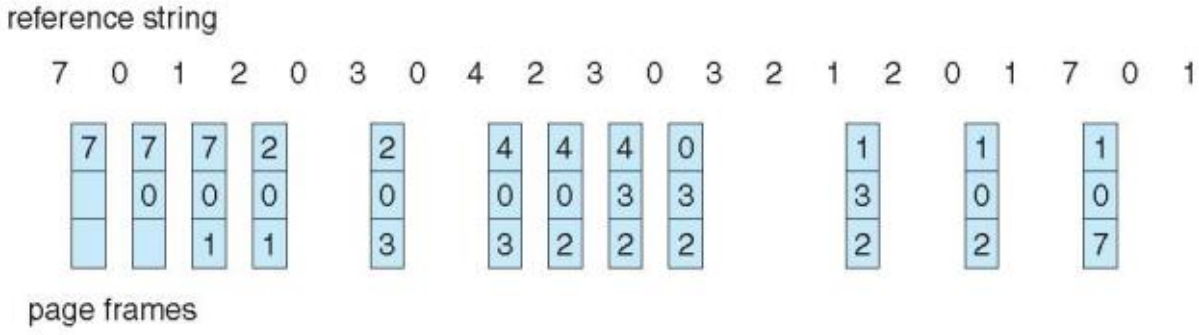
- **e.g.:** First replace "**7**", then "**1**", then "**0**", …
- **Optimal case:** 9 page faults (N = 20).

- Same ref string as FIFO example
- As soon as all 3 frames are assigned pages and a new page is scheduled (page 2), we look-ahead and check which page will not be used for the longest and replace this page (in this case page 7)
- Same procedure is followed for every page fault (i.e. whenever a scheduled page is not already in a main memory frame)
    - This results in 9 page faults (compared to 15 page faults for FIFO) for a reference string size of 20

Page Replacement: LRU

- Least Recently Used – **algorithm uses a record of past page replacement to replace pages that have not been used recently**
- LRU algo implemented as a linked list but on each use (page replacement), it places the new page at the head of the list and the LRU page (at the list tail) is removed
    - Could also implement as stack
- On average LRU yields a better task performance (lower number of page faults) than FIFO, but a worse task performance than OPT
- Main advantage:
    - It accounts for one of the key disadvantages of FIFO (replacing pages regardless of usage)
    - Also **accounts for the principle of locality** in the running of programs (processes)
        - That is, that the same set of pages will tend to be frequently used while the process is executing, so these pages should not be swapped out of main memory

LRU Algorithm Example



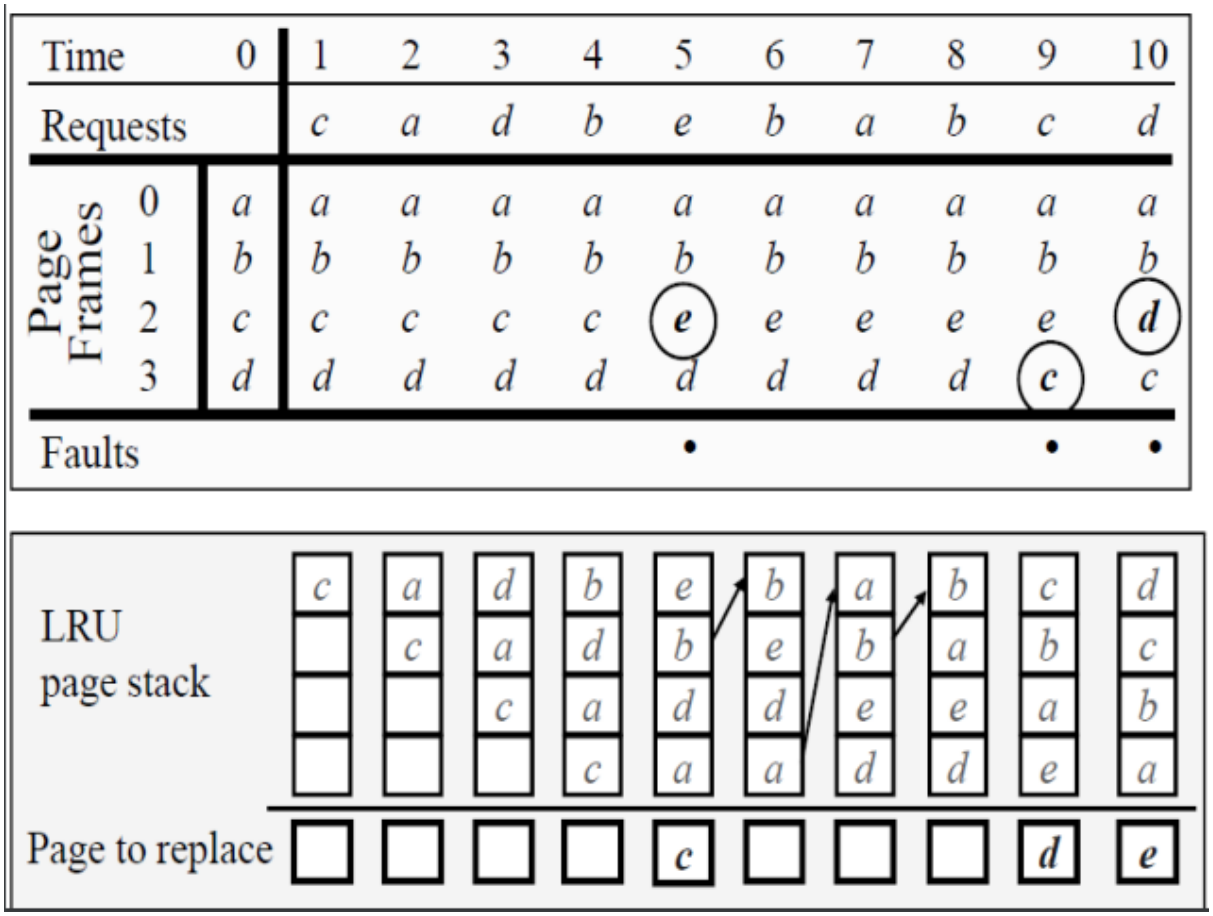
- **12 faults** – Better than FIFO – worse than OPT.

- Same ref string as FIFO example
- With each page fault that occurs after all frames are full (e.g. with the scheduling of 4th page in the reference string, page 2), the algorithm looks at all pages used thus far and selects the least used page to remove (in this example, page 7 is the first evicted as it has not been used, i.e. scheduled to run for the longest period)
- Thus, via associating the time of last usage with each page, the LRU algorithm is able to replace the page that has not been used for the longest period of time
    - On the same page reference string, the LRU algorithm yields better task performance (fewer page faults: 12) than FIFO, but worse performance than OPT

LRU Counter Implementation

- Every page has a counter
- Every time a page is referenced, counter is incremented (current time copied in)
    - Every time the page is used the page counter is incremented (represents the time at which the page was used in main memory)
- Replace the page with the smallest time value
- Data structure – linked list or stack
    - Replace the page with the smallest time value (oldest page) whenever there is a page fault (i.e. a new page is swapped in and all memory frames are full)

LRU Example



- Diagram shows an example of the LRU algorithm running for a string of 4 page references: a, b, c, d using 4 memory frames
    - Note that the time-steps (1, . . . , 10) are shown on the top row, and the page requests on the next row, where the pages swapped-in after a page fault are circled
- In this example there was a page fault at time-step 5, where page (e) was swapped-in and the oldest page replaced (c)
    - Looking at the stack data-structure below, one can see that the oldest page is stored at the bottom of the stack
    - When a new page is swapped into a memory frame then this page is pushed into the top of the stack and the second oldest page now becomes the oldest (at the bottom of the stack)
- The same procedure is followed when pages (c) and (d) are swapped-in at time-steps 9 and 10, respectively

Page Replacement Algorithms Analysis

- Decision of what pages to swap-in next and what pages to swap-out, is made by the page replacement policy (algorithm)
    - Most common page replacement algorithms are FIFO, LRU, OPT

FIFO: Frames versus Page Faults

- Does allocating more frames (for pages in main memory), lead to fewer page faults (for any given string of page references)?
- Example:
    - Consider the reference string – 3, 2, 1, 0, 3, 2, 4, 3, 2, 1, 0, 4
    - How many page faults if have frames = 3?

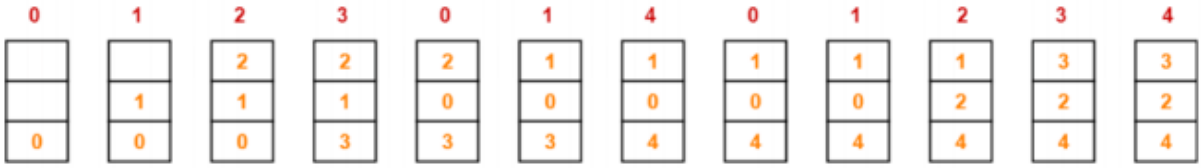
Page requests	3	2	1	0	3	2	4	3	2	1	0	4
Newest page	3	2	1	0	3	2	4	4	4	1	0	0
		3	2	1	0	3	2	2	2	4	1	1
Oldest page			3	2	1	0	3	3	3	2	4	4

    - FIFO algo run on the given string of 12 page references – where page faults are coloured in red and an indication of the oldest versus newest pages are given next to each frame number (where the oldest is replaced at each page fault)
    - Running FIFO on this page reference string results in a total of 9 page faults
- Example 2:
    - Consider the reference string – 3, 2, 1, 0, 3, 2, 4, 3, 2, 1, 0, 4
    - How many page faults if have frames = 4?

Page requests	3	2	1	0	3	2	4	3	2	1	0	4
Newest page	3	2	1	0	0	0	4	3	2	1	0	4
		3	2	1	1	1	0	4	3	2	1	0
			3	2	2	2	1	0	4	3	2	1
Oldest page				3	3	3	2	1	0	4	3	2

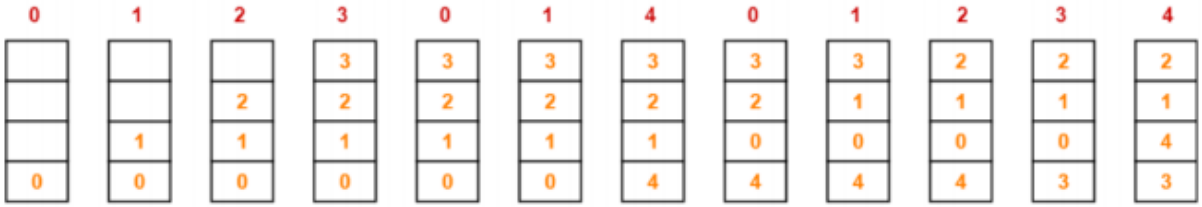
    - Number of page faults increased to 10 when running the FIFO algorithm – illustrates Belady's Anomaly
- Comparison of FIFO algo run on the same given string of 12 page references, for 3 page frames (top) versus 4 page frames (bottom)
    - More frames = increased page faults

Case-01: When frame size = 3 **FIFO**



Number of page faults = 9

Case-02: When frame size = 4

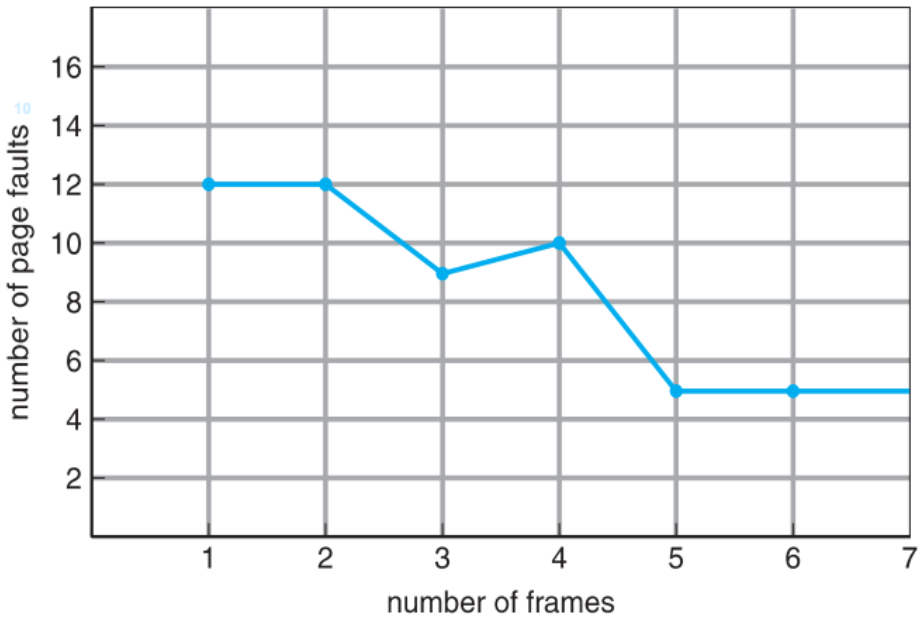


Number of page faults = 10

Belady's Anomaly

- **Phenomenon where increasing the number of page frames results in an increase in the number of page faults for a given memory access pattern**
  - This phenomenon is commonly experienced in the FIFO, second chance and random page replacement algorithms, but not in the OPT and LRU page replacement algos
- Belady's anomaly will not occur, for any page reference string, in the LRU and OPT algorithms as they *belong to a class of stack-based page replacement algorithms*
  - A stack-based algorithm is one for which it can be shown that the set of pages in memory for N frames is always a subset of the set of pages that would be in memory with N + 1 frames
  - So, if the number of frames increases then these N pages will still be the most recently referenced and so will still be in memory
- For LRU page replacement algorithm, the set of pages in memory would be the N most recently referenced pages, while in FIFO, for example, if page b came into physical memory before page a, then priority of replacement of page b is greater than that of page a, but this is not independent of the number of page frames
  - So, using FIFO, increasing the number of page frames *can increase the fault rate for some page access patterns (this depends on the process, so Belady's anomaly is not always evident)*
  - If there are more page frames, recently requested pages can remain at the bottom of the FIFO queue longer
  - Hence, FIFO does not follow a stack page replacement policy and therefore suffers Belady's anomaly

- Graph showing the number of page faults versus the number of frames given the execution of the FIFO algo applied to any given page reference string
    - Belady's anomaly is illustrated for an increase from 3 to 4 memory frames, but otherwise increasing the number of frames reduces the number of page faults as one would intuitively expect
    - **Note that graph is for particular set of page reference strings – it won't hold for all sets of page reference strings**
        - Depends on what sequence of pages are referenced by the FIFO algo as to whether Belady's anomaly occurs or not

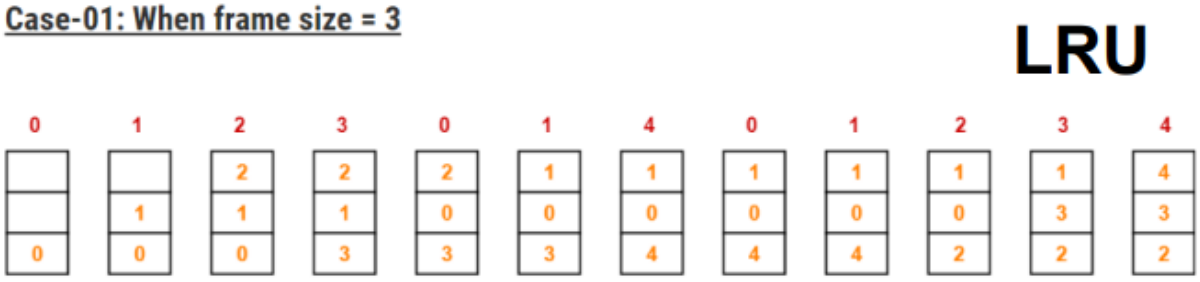


LRU

- LRU page replacement is a stack algorithm and hence does not suffer from Belady's anomaly
- Stack algorithms: an algorithm for which it can be shown that the set of pages in memory for N frames is always a subset of the set of pages that would be in memory with N + 1 frames
    - **Increasing memory size is guaranteed to reduce (or at least not increase) the number of page faults**
    - Therefore, an additional frame will never cause an additional page fault
    - Example – LRU is a stacking algorithm, and using k frames will always be a subset of k + N frames for LRU, any page-faults that may occur for k + N frames will also occur for k frames (these k pages will still be the most recently referenced and so will still be in memory), which in turn means that LRU does not suffer from Belady's anomaly

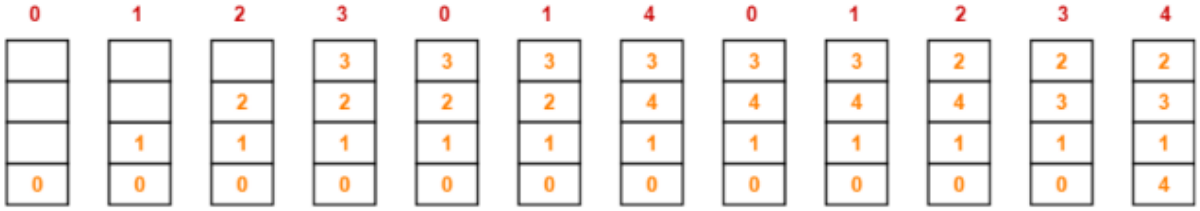
LRU Algorithm Example

Case-01: When frame size = 3



Number of page faults = 10

Case-02: When frame size = 4



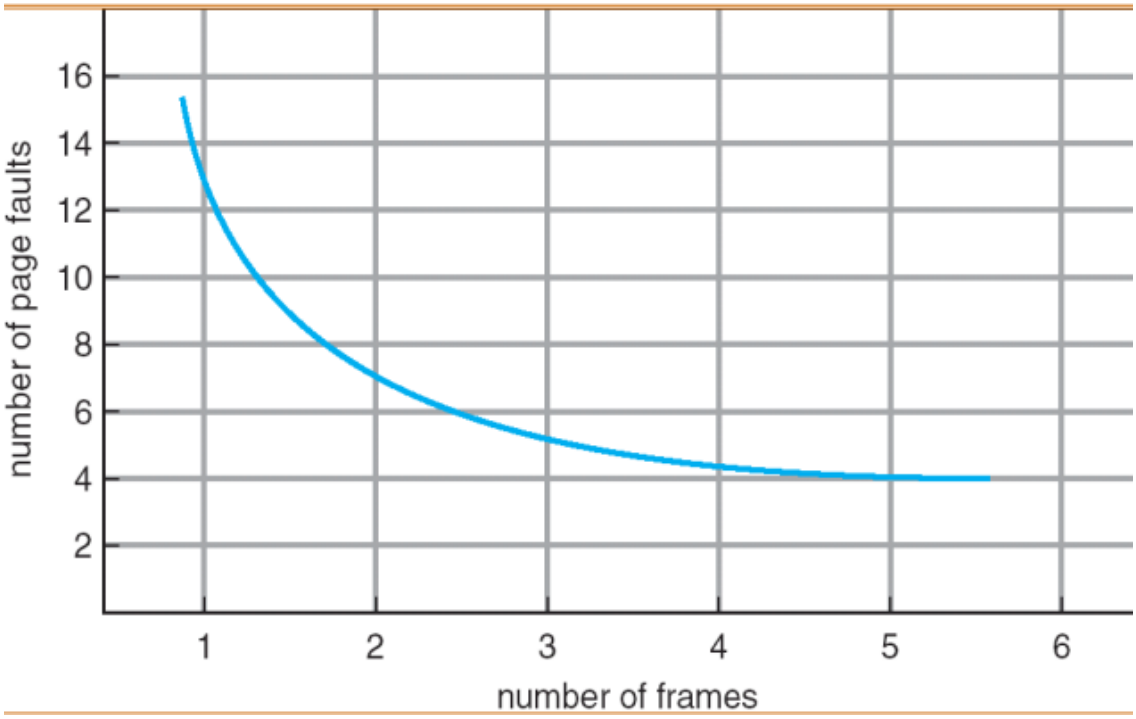
Number of page faults = 8

- Diagram shows that increasing the number of memory frames in this case, reduced the number of page faults (from 10 to 8)

LRU Example

- LRU algorithm working on the given page reference string, with 4 memory frames
  - Recall that LRU looks at page reference history and replaces the page not used in longest time

    - Can view this is as a stack, where the most recent page is on-top and other pages are pushed down, as opposed to the FIFO queue
- Look at 3 frames at any point, this set of page references is always in there for 4 frames, since we always replacing the bottom of the stack



- **For LRU (and related algorithms using age-based page replacement), increasing the number of frames available reduces the page fault rate**

Most Frequently Used (MFU)

- Variation of LRU
- Both the LRU and MFU algorithms keep a counter of the number of references that have been made to each page
    - Whereas, LRU replaces the page with the smallest count, MFU does the opposite by replacing the most frequently used page
- **MFU algorithm is based on the argument that the page with the largest count has been used enough and that pages with smaller counts were probably just brought into main memory and have yet to be used**
- Note that both the LRU and MFU make assumptions about future referencing (use) of pages by the CPU based on the history of page references, though depending on the exact nature of the process scheduled to run these assumptions may or may not hold
    - Such age-based page replacement algorithms need to make use of a counter

Counting Algorithms

- **All implementations and variations of the LRU algorithm require some hardware assistance**
    - In such cases, the OS keeps a time stamp for each page with the time of the last access in order to swap-out the least recently used page

- LRU requires the OS to keep a doubly linked-list of pages, where the front of the list is the most recently used page, and the end is the least recently used page
- However, *updating a counter with each page reference is computationally expensive*, since the counter must be frequently updated and stored in cache memory directly accessible by the memory management unit
    - Also, the linked-list must also be maintained in quickly accessible cache memory and frequently modifying pointers to a link-list can also become computationally expensive if the cache memory is not sufficiently large or fast
- Hence a perfect LRU is not usually practically feasible and most modern operating systems use some approximation of LRU for page replacement

Approximating LRU

- A perfect version of the Least Recently Used (LRU) algorithm assumes that the OS keeps a timestamp for each page reference and keeps a list of pages ordered by time of reference
    - However, this is computationally too expensive to implement given the very large memory and processing demands of any modern OS
- The **clock replacement algorithm uses a reference bit to approximate LRU**, i.e. to try to achieve some of the benefits of LRU replacement, but without the computational overhead of manipulating the LRU bits on every page hit
    - This reference bit corresponds to a used-bit that is set for each page reference
- If this used-bit is not set, it means the corresponding page has not been referenced in a long time
    - This used-bit is usually stored, for each page, in the page table or Translation Look-aside Buffer (TLB), and as it is just one bit, stored in cache memory, it can be quickly accessed and modified
- When a page reference occurs, the clock replacement algorithm assumes a page can be in one of three states, which is determined by this reference bit
    - **The clock algorithm replaces an old page but not necessarily the oldest page**

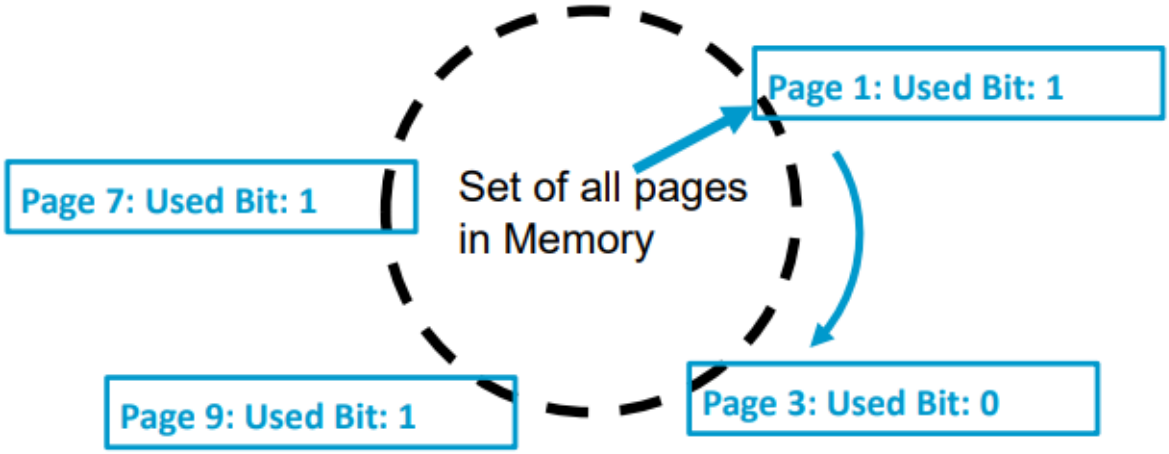
Clock Approximation of LRU

https://www.youtube.com/watch?v=b-dRK8B8dQk

- When a page reference occurs, the clock replacement algorithm assumes a page can be in one of three states, which is determined by this reference bit
- (1) **Used-bit = 1** – used bit is true, so the **page is present in memory and recently used**.
    - In this case there will be no page fault when this page is accessed, so no bits will change
- (2) **Used – bit = 0** – used bit is false, so the **page is present in memory but not recently used**
    - If a page fault occurs in this case, e.g. if the page is being temporarily stored in the Translation Look-aside Buffer (TLB) rather than being loaded into main memory, the only thing the page replacement algorithm does is change the state to recently used (used-bit = 1)
- (3) The page is not present in memory
    - Look at clock-hand
    - Clock algo arranges physical pages in a conceptual circle with single clock hand (pointer) pointing to one page at any given time

    - If the clock-hand is pointing to a page with the used-bit set to true, then we flip the used-bit to false and increment the clock-hand to point to the next page (in the circle of pages)
    - *When we find a page with the used-bit already cleared (used-bit = 0), that is the page we replace*
    - Then we mark the new page as recently used (used-bit = 1) and increment the clock-hand to the next page
- When a page reference occurs, the reference bit (used-bit) for the given page is set
    - The clock algorithm then advances the page pointer until a page with a cleared used-bit (= 0) is found
- The **clock algorithm thus replaces pages that have not been referenced for one complete revolution of the page pointer**

Example



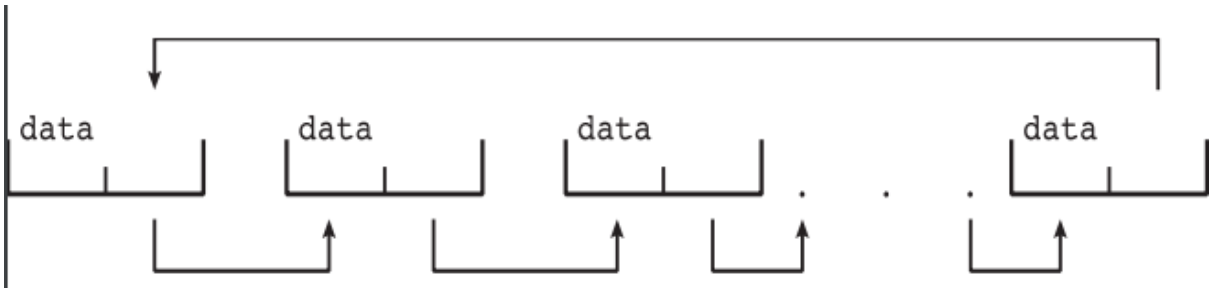
- Logically, all physical page frames are arranged in a circle using a circular linked list
- **The goal of the clock algorithm is replace pages that are old enough and not necessarily the oldest**
    - Hence the clock-hand is used to select a good least recently used candidate page via sweeping through the pages in circular order like a clock, until a page with its used – bit = 0, is found
- That is, if the used-bit is off (= 0), this is an indication that the page has not been used recently, and is thus selected for replacement
    - Whereas, if the used-bit is on (= 1), it is switched off (= 0) and go to next page
- In the example figure shown, the clock-hand advances to page 1 (with used-bit set).
    - So, the clock algorithm will set the used-bit (= 1) for page 1 and then move onto to the next page in the circular-list: page 3
    - This page has its used-bit cleared (= 0), so it is selected for replacement (i.e. victim page)

Clock Replacement Algorithm

- Pseudocode

```
func Clock_Replacement
begin
   while (victim page not found) do
      if (used bit for current page = 0) then
         replace current page
      else
         reset used bit
      end if
      advance clock pointer
   end while
end Clock_Replacement
```

Clock Algorithm Data Structure



- **Clock algorithm stores pages in circular linked list, with one reference bit per page**
- Recall that when a page reference occurs, the algorithm checks if the page is already loaded into main memory
    - If not, a page fault occurs, and the clock page replacement algorithm accesses this circular linked list (clock of pages currently in main memory frames) to determine which page to replace (with the just referenced page)
- The clock checks the reference bit of the next page, and if reference bit is 0, then replace this page and set its bit to 1
    - If reference bit is 1, it set its reference bit to 0, and advance the clock pointer to the next page
- In the best case, this circular linked-list is very efficient since the next page will have its reference bit cleared (= 0), and will be replaced
    - i.e. 1 element before page can be replaced
    - Worst case – all reference bits are set (= 1), the pointer will need to check every page in the list before eventually looping around and replacing the page where the pointer (clock-hand) started

        - i.e. N elements before page can be replaced
- **Hw (cache memory) requirements for the clock algorithm is that it maintain just one reference bit with each page (currently in main memory)**
    - Makes page replacement by the clock algorithm generally fast and efficient
    - However, a reference bit only indicates if page was used at all since last checked and not how long it has been since last used
        - Not as accurate as timestamp saying when that bit was referenced – just an indication if it has been recently referenced
- To address this disadvantage, a more accurate way to implement the clock algorithm is to have extra reference bits associated with each page
- Example:
    - 8 reference bits could be associate with each page rather than just one bit
    - At each page reference, we could then increment this 8-bit reference for each page to reflect how old a page is, where there's a page fault, the oldest page is replaced
- This approach is still approximate, since it does not give a perfect time-stamp (representing real-time access) for the pages, and is fast, since it is still just setting a single bit on each memory access
- However, a page fault will require a search through all the pages in the linked list, in order to be sure that the oldest page is replaced
    - Page fault is faster, since we only search the pages until we find one with a '0' reference bit

Clock Algorithm: Issues

- In the best case the circular linked list implemented by the clock algorithm is very efficient since the next page will have its reference bit cleared (= 0), and will be replaced
    - Whereas, in the worst case, if all reference bits are set (= 1), the pointer will need to check every page in the list before eventually looping around and replacing the page where the pointer (clock-hand) started
        - But **a page to replace will always be found**
- Note that if the clock-hand is advancing slowly (pointer passes few pages before a page is replaced), then this means that there have been many page faults and pages replaced and thus few reference bits set
    - However, if the clock-hand is moving fast (pointer passes many pages before a page is replaced), then this means there have been relatively few page faults, pages replaced and thus reference bits set
- Also, there is *added computational cost for replacing pages with the dirty bit set*

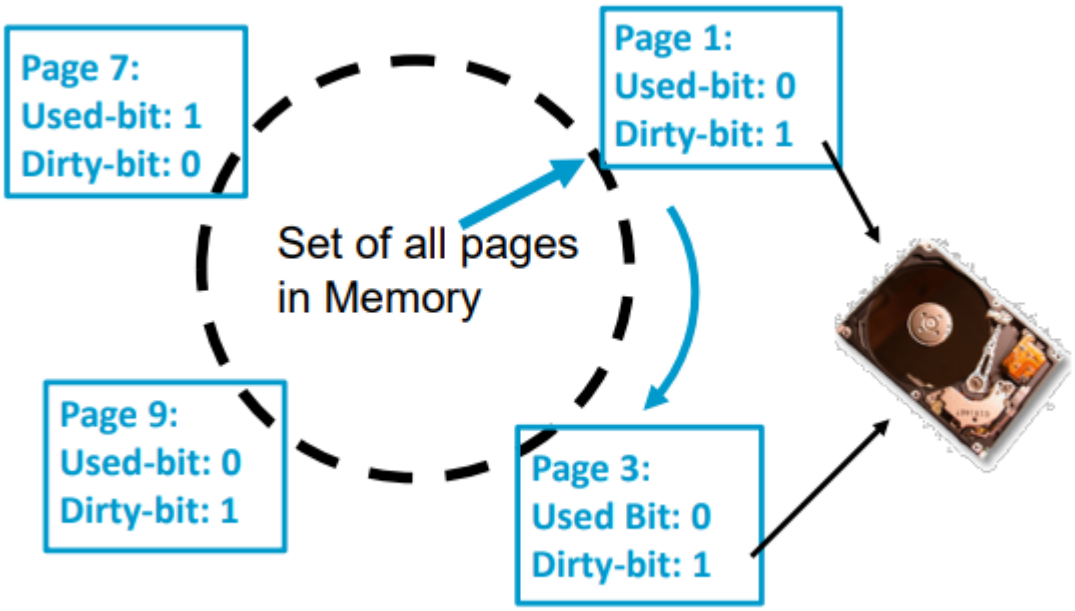
Dirty Pages

- **A page which is modified in a buffer cache between swap-out from secondary storage and swap-in to a main memory frame**
    - i.e. a page with its corresponding dirty-bit being set in the page table
    - Significant cost to replacing dirty pages – added computational complexity
- There is one dirty bit assigned for each physical page frame, where this dirty bit is set whenever the page is modified
    - *If a dirty page is replaced, it must be written to disk before its page frame is reused*
- To account for this, the clock algorithm typically gives additional preference to dirty pages

- Example – if reference bit for a page is clear, but the dirty bit is set, then the algorithm will not replace this page now, but rather clear the dirty bit and start writing the page to disk
- That is, the clock algorithm must be modified to allow dirty pages to always survive one sweep of the clock-hand (to enable them to be written back to disk)

- This modified clock algorithm is known as the *second chance clock algorithm*

Second Chance Clock



- Diagram – shows second chance clock algorithm running on a set of pages (currently stored in physical memory frames)
  - Note that each page now has a dirty-bit and used- bit associated with it
- In this case, **when there is a page fault, the algorithm checks if both dirty bit is cleared** (= 0), i.e, the page is changed but already written back to disk, **and the used-bit is cleared** (= 0), i.e, this page is the victim page selected for replacement

Second Chance Algorithm

Before clock sweep

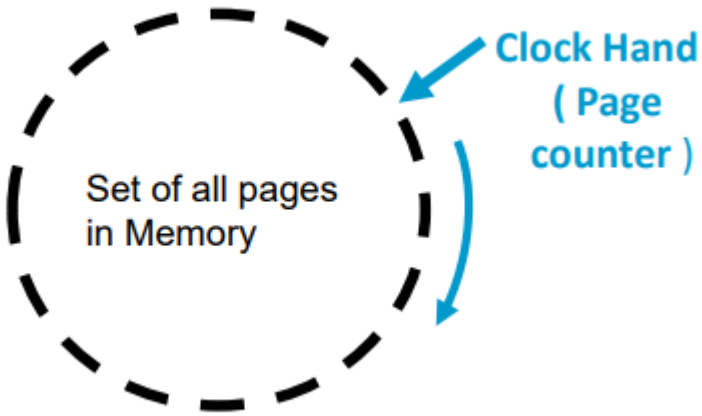
used	*dirty*
0	0
0	1
1	0
1	1

After clock sweep

used	*dirty*
replace	*page*
0	0
0	0
0	1

- Tables shows how the used-bit and dirty-bit should be set before and after the clock-hand (page pointer) passes a given page (with specific bit settings)
- Both the used-bit and dirty-bit are used to drive page replacement given a page fault (recall that if a page reference is invoked for a page already in memory, then the replacement algorithm does not get invoked at all)
    - i.e. need 0 (used and 0 (dirty) to replace a page

Extending the Chance Clock Algorithm – Nth Chance Clock



- What if we don't want to replace heavily used pages?
    - Can we give these pages more of a chance to stay in main memory longer?
- **In terms with the principle of locality, many processes will heavily use and reuse a specific subset of pages in order to execute the core part of the process**
    - In such cases, we do not want to replace frequently used pages, but rather (ideally), leave them swapped into main memory for as long as possible

- Hence, the idea of the N chance clock algorithm is to give such frequently used pages as much of a chance as possible to remain in main memory before being replaced
    - This algorithm keeps a counter for each page, meaning that the clock-hand (page pointer) can pass each page N times before the page will be selected for replacement
    - This is Nth chance clock algo
- Keep counter per page: number of sweeps
    - Nth chance algorithm gives each page N chances, where a counter is stored (per page in cache memory), to indicate the number of sweeps
- On page fault – checks use bit (of the page the clock-hand is currently pointing at):
    - 1 -> clear use bit and also clear counter (i.e. page used in last sweep)
    - 0 -> increment counter; if count=N, then replace page
        - Means that **clock hand has to sweep by N times without page being used before page is replaced**

Choosing N

- Main disadvantage of the Nth Chance clock algorithm is that we now have an extra parameter: N, that we must somehow decide a value for
- **Value of N we select depends on how well we want to approximate the LRU page replacement algorithm**
    - Larger values of N better approximate LRU since least recently used pages get to spend longer in main memory (before replacement)
    - The algorithm using a lower value of N may select a more recently used page for replacement, but such an approach tends to be more efficient (especially compared to very large values of N)
    - That is, for very large values of N the algorithm will tend to cycle around looking for a page with its counter set to N
- Also, the Nth chance clock algorithm uses additional cache memory to store this N counter for each page, and even more computational overhead is needed to account for dirty pages (in addition to the page use counter)

Summary

- Main practical page replacement algorithms – FIFO, LRU and Clock
    - These page replacement algorithms are only invoked when a page fault occurs (there is a page reference to a page not currently stored in one of the physical memory frames)
- FIFO: Place pages in queue, replace end page
- LRU: Replace least recently used page
- Clock: Approximation to LRU
    - Arrange all pages in circular linked list
    - Sweep through them (with a page pointer), marking as not “in use”
    - If page not “in use” for N passes, then replace
- **Experimental Analysis – all algorithms perform badly if processes have insufficient physical memory**
- All algorithms approach optimal – as physical memory allocated to processes approaches virtual memory size

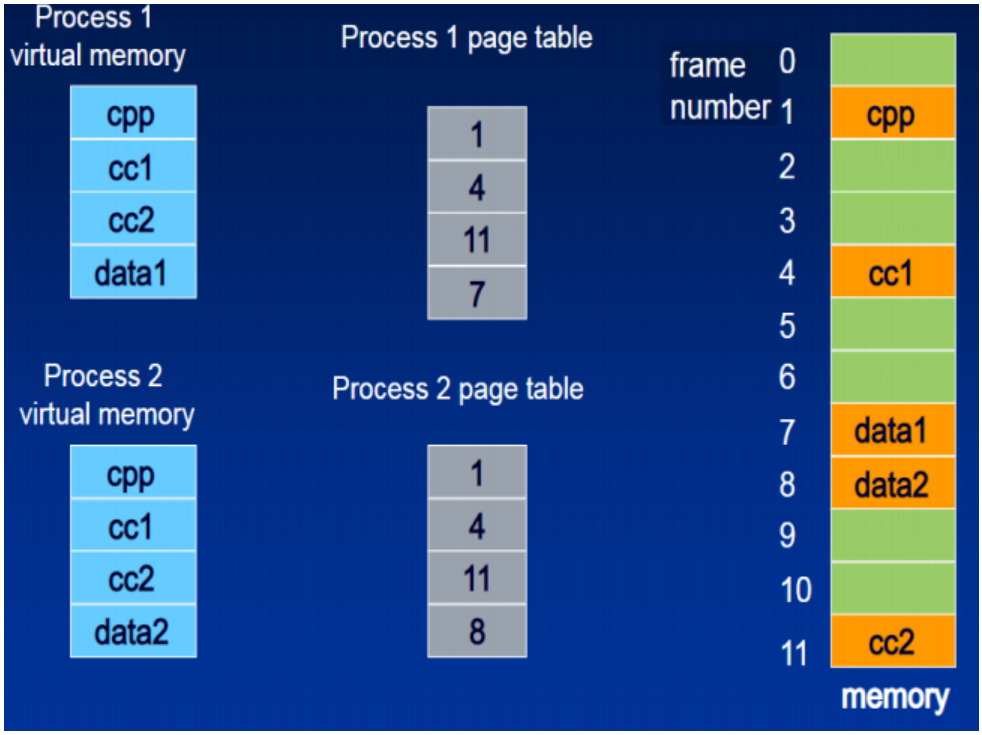
- *More processes running concurrently – less physical memory each process can have*
- Critical OS issue – must decide how many processes (and frames per process) concurrently share memory

Frames

- Load control strategies: How many processes can fit in memory at one time

Overview: Memory Management

- All algorithms perform badly if processes have insufficient physical memory
- All algorithms approach optimal as physical memory allocated to processes approaches virtual memory size
- More processes running concurrently means the less physical memory each process can have
- **Critical OS issue – must decide how many processes (and frames per process) concurrently share memory**
    - Such algorithms are known as **load-control strategies**
- Memory Frame Allocation
    - How do we allocate memory among different processes?
    - Does every process get the same fraction of memory? Different fractions?
    - Should we completely swap some processes out of memory?
- *How should the OS best allocate memory (pages) between different processes (scheduled to run), before allocating these pages to memory frames in main memory where the pages (partial processes) can be executed by the CPU?*
    - **Each process needs some minimum number of pages in order to fully execute and we want to make sure that all processes that are loaded into memory can make forward progress**
    - Generally, we will have much more process memory demand than we have available free frames in main memory
- Example:



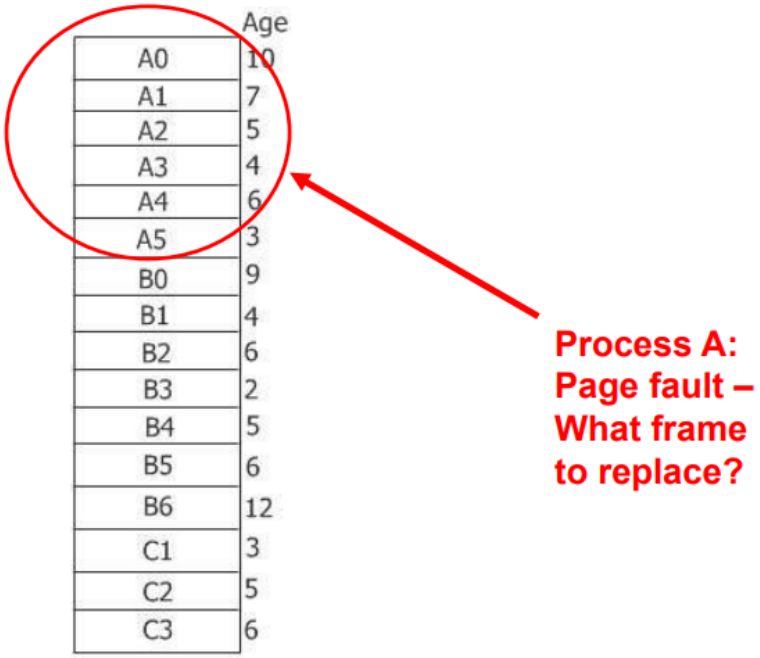
- Two processes running in virtual memory, each with its own page table
  - Each process has 4 executable parts
  - Each page table indicates the 4 frames in physical memory that have been allocated to each process
  - For each process, pages are then mapped from virtual memory, via the page table, into the physical memory frames
- Processes 1 and 2 share frames in main memory to allow both processes to run, where memory protection is used to ensure that the page for one process is temporarily swapped-out and suspended, while the page for the other process is swapped-in and running in a given physical memory frame
- However, memory protection aside, there must be some-kind of load balancing algorithm to determine how many page frames each process is assigned and can maximally use

Page Replacement

- **All page replacement algorithms can be implemented using local or global replacement**
  - *The goal of any page frame allocation (i.e. local or global replacement) is to ensure that each process scheduled to execute, has a minimum number of page frames allocated, in order that enough of its pages can be loaded into memory and that the process can run*
    - Though with minimal page frames this may require pages being loaded incrementally and the process being executed bit by bit
  - The general scheme of page frame allocation can be decomposed into local and global replacement approaches
- **Global replacement: process selects replacement frame from the set of all frames**
  - i.e. one process can take a frame from another process
  - Victim page chosen from all page frames (regardless of process)

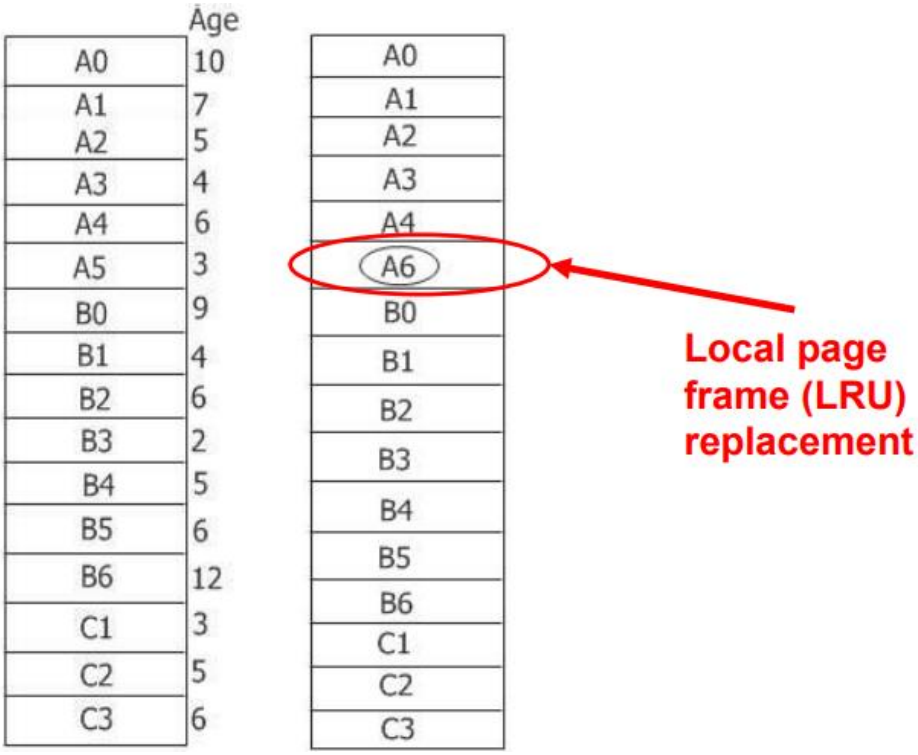
- **Local replacement: each process selects replacement frames from only its own set of allocated frames**
    - Process has a limit of pages it can use and pages against itself (evicts its own pages)

Local versus Global Frame Allocation



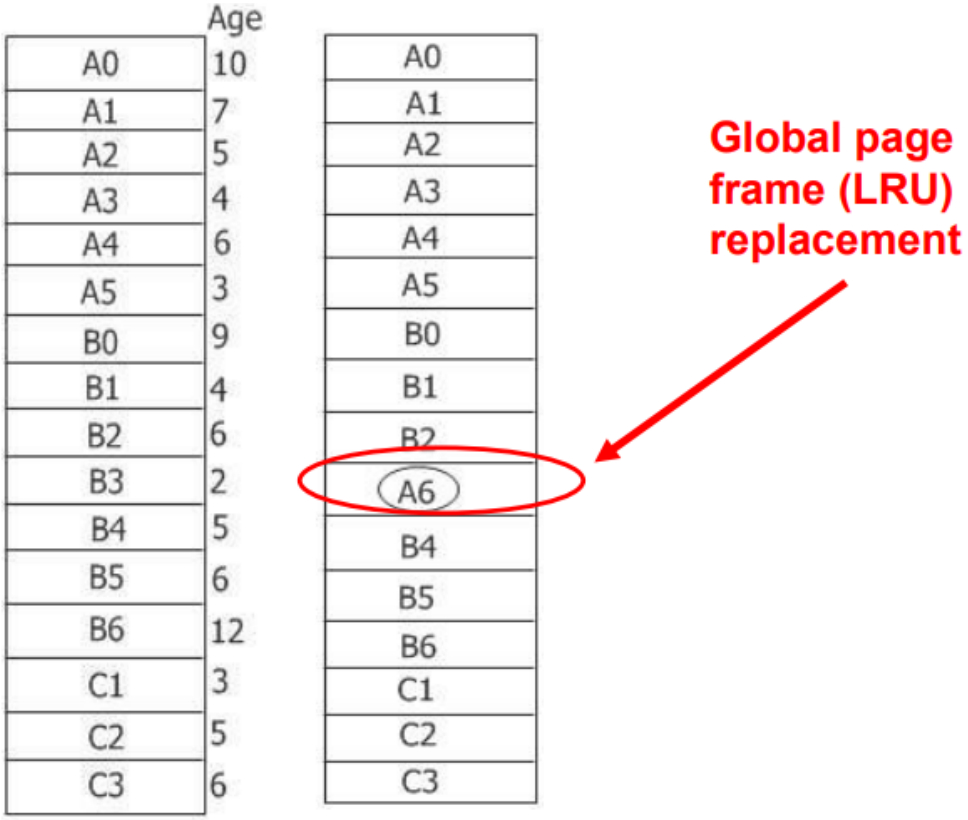
- Diagram shows an example of a set of memory page frames that comprise main memory and the set of frames allocated to process A, B and C
    - Specifically, page frames: A0, . . ., A5, are allocated to process A, and page frames: B0, . . ., B6, are allocated to process B, and page frames: C1, . . ., C3, are allocated to process C
- Age value next to each frame is how long a page has occupied the given frame, where the time counter started at 1 and is currently at 12
    - i.e. B6 is the most recently used page

Example: Local Frame Allocation with LRU



- Figure shows an example of local page replacement using the LRU page replacement algorithm with the illustrated page frame allocations per process
- A new page A6 has been referenced and must be loaded into one of the page frames allocated to process A (as mandated by local replacement)
    - Since, the LRU algorithm is used, the oldest page is swapped out (page A5 was swapped in at time = 3), and the new page A6 is swapped-into this memory frame

Example: Global Frame Allocation with LRU



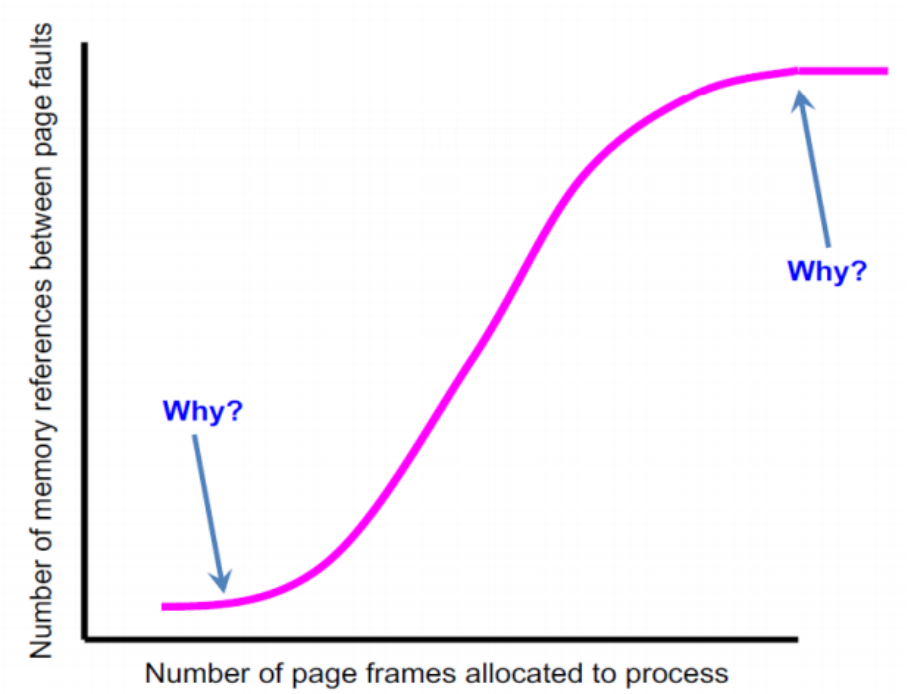
- Diagram shows an example of global page replacement using the LRU page replacement algorithm with the illustrated page frame allocations per process
    - A new page A6 has been referenced and must be loaded into one of the page frames allocated to all processes A, B and C (as allowed by global replacement)
    - Since, the LRU algorithm is used, the oldest page (for all processes in this case) is swapped out (page B3 was swapped in at time = 2), and the new page A6 is swapped-into this memory frame

Types of Page Replacement

- Advantages and disadvantages of using global versus local page replacement approaches
- Global:
    - Pro – **page frames are dynamically allocated among processes**, i.e., the number page frames assigned to a process can vary
        - Makes global page replacement algorithms more efficient (in memory allocation) overall
    - BUT also means that *large computationally expensive processes can be assigned too many memory frames and monopolize memory usage*
- Local:
    - Tend to be **more consistent in the per process memory allocation**
    - However, since processes are allocated a fixed fraction of the page frames, there is potentially *under-utilisation of memory since some processes may have slightly lower memory demands and other processes slightly higher memory demands*

- Thus, **global page replacement algorithms generally work better than local page replacement algorithms, especially when the size of working set can vary over the process lifetime**

Page Faults versus Frames Allocated

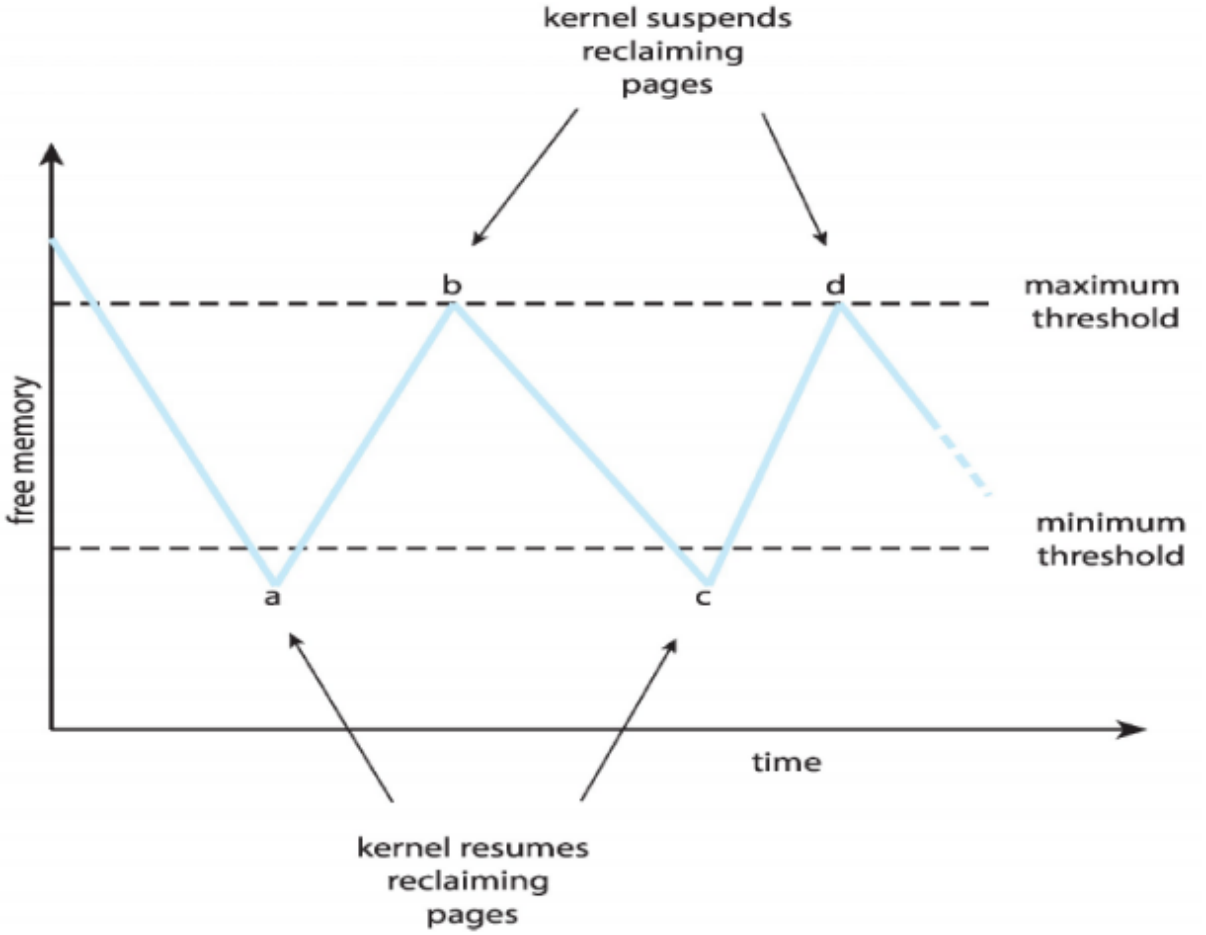


- Graph of the relationship between the number of memory references between page faults and the number of page frames allocated to each process
- **With more memory frames allocated per process (far right), we have more memory references between page faults meaning that we have fewer page faults**
    - i.e. more time on average for memory references (non-faulting page access)
- **With few memory frames allocated per process (far left), then we have more page faults and less time for memory references (accesses)**
    - *Since we spend most of the time handling faults and not doing memory accesses* (non-faulting page accesses)

Reclaiming Pages

- **When a page fault occurs, the OS must bring the desired page from secondary storage into main memory**
    - **Most OSes maintain a free frame list – a pool of free frames for satisfying such requests**
    - This free frame list is just a list of free memory frames in main memory
- Page replacement (allocation of pages to frames for various processes), occurs when the number of free frames in the list falls below a given threshold
    - Ensures that the *OS always has some free memory frames to assign to newly scheduled processes or processes with increasing memory demands*
- When an OS starts up, all available main memory is usually placed on the free frame list

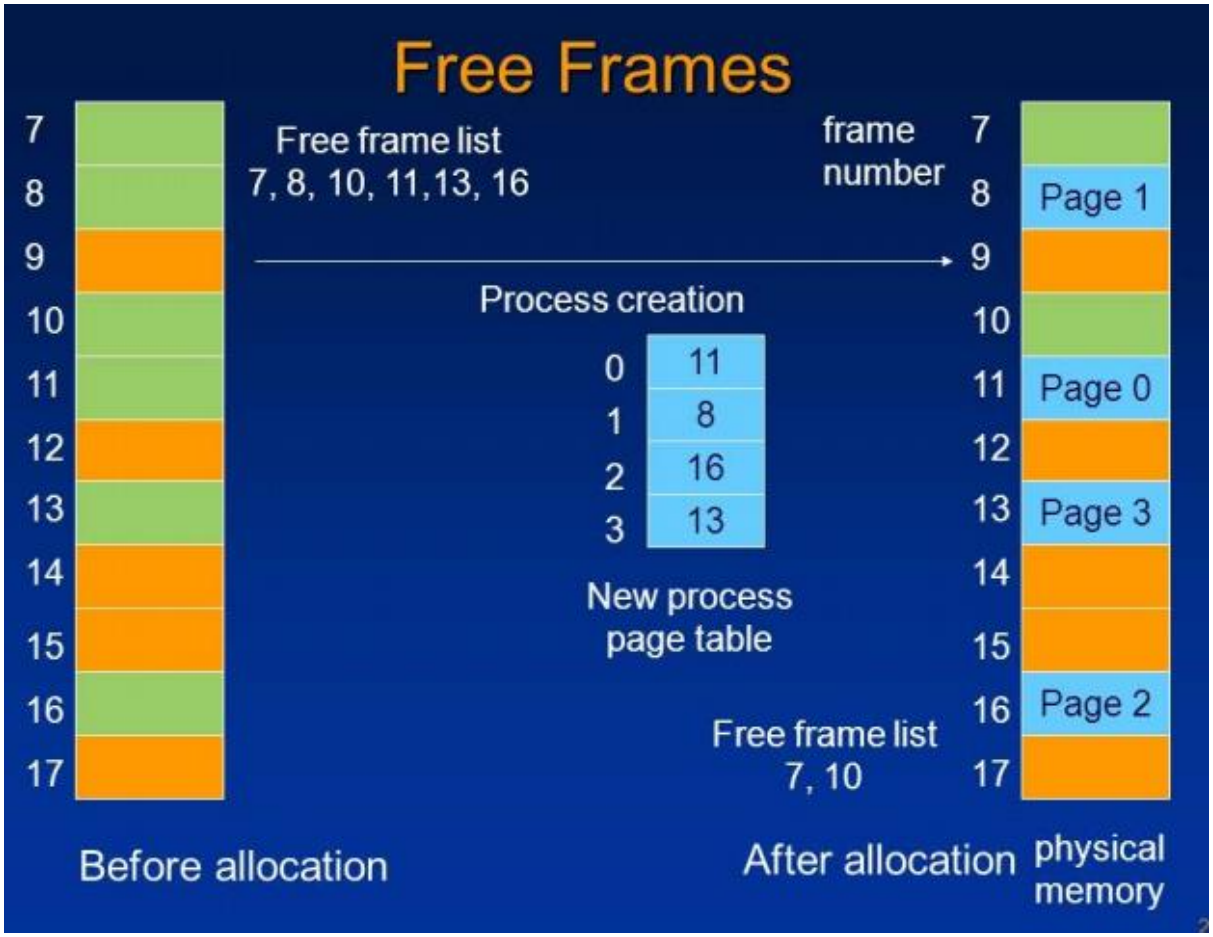
Reclaiming Pages Threshold



- Diagram shows an example of maximum and minimum thresholds for the reclaiming of pages (memory frames) from the free frame list
    - The approach illustrated for reclaiming pages is used together with a global page replacement algorithm, since the reclaiming of pages accounts for all memory frames (in main memory) allocated to all process (currently scheduled to run by the OS)
- In the figure, the vertical axis shows the amount of main (physical) memory that is currently free (not allocated to page frames for specific processes), and the horizontal axis shows the time for which the OS has been running
- The free frame list is just a list of free memory frames in main memory (typically joined together with a linked list)
    - Assuming global page replacement (i.e., where the free list encompasses all running processes), ideally want a page replacement algorithm to activate when the number of free frames (in the list) falls below a given minimum threshold
- **If min threshold is exceeded – then have too little free memory**
    - *OS resumes reclaiming pages* – h*ave to kick out memory pages from some number of frames in order to free up the min number of frames* (in order to keep free frame list)
- However, don't want too many free frames to be assigned, as this will cause an increase in page faults if new processes are scheduled to run (thus needing their own memory frame allocations)

- Thus, **if the number of free frames assigned to currently scheduled processes would exceed a maximum threshold**, **then the OS (kernel) would suspend the reclaiming of pages until a currently scheduled process finishes running**, and the memory frames assigned to that process can be returned to the free frame list

Free Frame List



- Free frame list: a (linked list) data structure used in a scheme for dynamic memory allocation
- Figure shows the free frame list (green blocks, left-hand side), before pages for a specific scheduled process (centre) was allocated to frames (blue blocks, right-hand side)
- The newly scheduled process is accordingly assigned a page table, and each row of the page table is a specific frame number in physical memory (rhs)
- The free frame list was (7, 8, 10, 11, 13, 16), prior to the scheduling of this process to run
    - OS kernel then assigned the process four (4) frames to run, reducing the free frame list to (7,10)
    - The orange blocks in main memory denote memory frames already assigned to currently executing processes
- **A free frame list operates by connecting unallocated regions of memory together in a linked list**
    - i.e. green memory frames on left and right-hand side of example figure
    - Using first word of each unallocated frame as a pointer to the next frame
    - Free frame lists make memory allocation and deallocation operations very simple
        - That is, to free a memory frame, after swapping out the process memory page in the frame, one would just link the given frame to the free frame list

Page Frame Allocation

- **Two approaches to page frame allocation for N processes that are currently scheduled to run** – equal and proportional allocation
- **Equal**: each process receives an equal share of frames from the free frame list
    - Advantage – *no one process (with very large memory demands) can monopolize usage of a majority of memory frames*
    - e.g. 100 frames, 5 processes
        - Each gets 20 frames
- **Proportional**: *assigns a number of frames proportional to the size (number of required pages) for each process*
    - Advantage – ensures that large processes do not run too slowly and thus have their pages residing in main memory for too long

According to process size:

s_i = size of process p_i and $S = \sum s_i$

m = total number of frames

a_i = (allocation for p_i) $= \frac{s_i}{S} \times m$

Priority Allocation

- Priority memory frame allocation: the **use of process priorities instead of process size**
- Same type of computation as fixed allocation:
    - If a page fault is generated, requiring a page to be swapped in for a given process p, then a page currently residing in a memory frame must be evicted so as the page for process p can be swapped in
- In such a case, **all frames are assigned a priority number, indicating the priority (importance) of the page currently residing in the given frame**
    - The frame with the lowest priority is then the one that is selected to have its residing page swapped out, so as a page for process p can be swapped in
    - *High priority frames are less likely to be selected for replacement* compared to low priority frames
        - High priority processes will consist of high priority memory pages – less likely to be swopped out
- Main problem – a priority number has to be assigned to each page (or each page of each process)
    - And the best assignment of priorities between processes (especially between user processes) is often a contentious issue

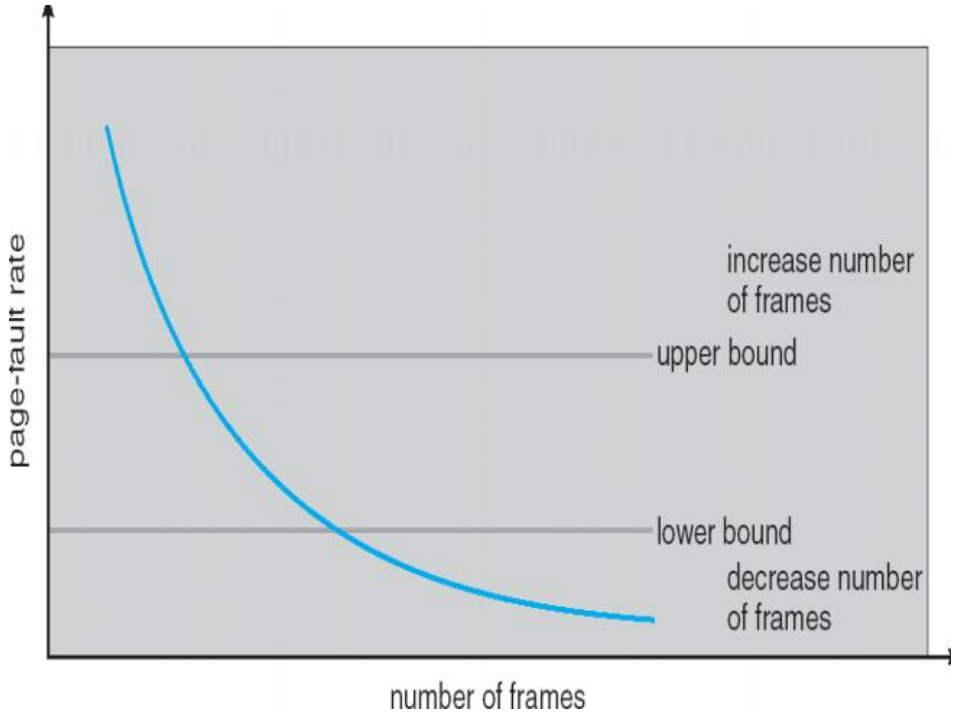
Page Fault Frequency Allocation

- **Another way to handle the allocation of main memory frames between processes is the use of a Page Fault Frequency (PFF)**

- PFF metric is used with **local page replacement** and works via monitoring the pff of each process (currently scheduled to run)

- **If the pff, measured over some run-time (T) of a given process is too low, then this process loses a frame to the free frame list**
- Whereas **if the pff is measured over the same process run-time (T) for the same given process is too high, then this process gains a frame from the free frame list**
- The pff policy is thus a direct method for reducing page faults via dynamically changing the number of page frames assigned per process
    - However, this policy *introduces additional parameters* (T and the too low and too high pff thresholds) that must be decided ahead of time and *it is difficult to determine values that generally work well*

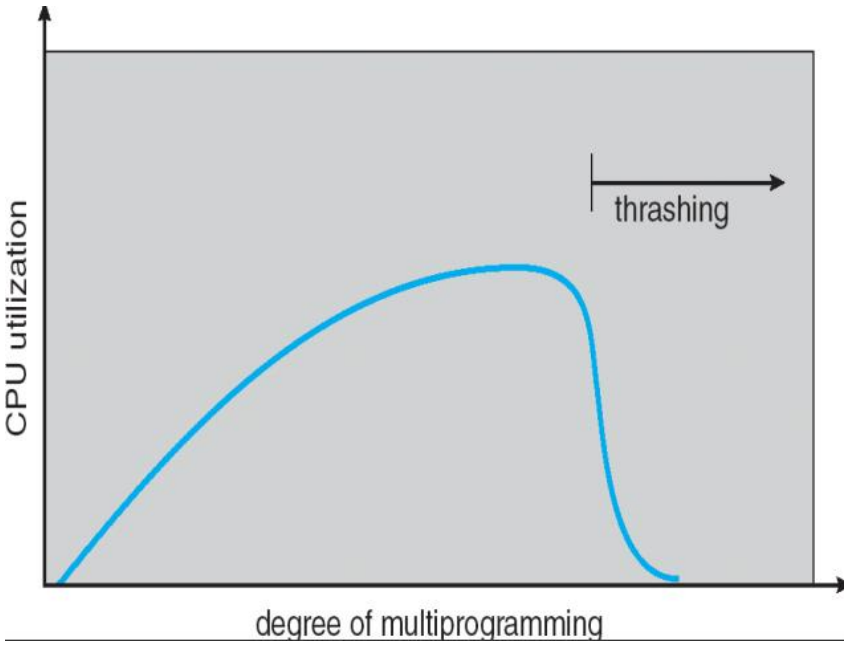
Page Fault Rate versus Number of Frames



- Graph of the ideal (i.e. given a suitable page replacement algorithm) decrease in page fault rate as the number of frames increases (for all processes currently scheduled to run)
- **It's difficult to establish pff parameters that work generally well (for all OS running conditions and numbers of scheduled processes)**
- However, to help determine these parameters, we can measure the pff over some time period, where we do test runs across a range of different process load conditions for the OS, in order to come up with some average upper and lower bound of PFF
- The upper bound and lower bound correspond to the too high and too low thresholds
    - Where, **a given process that has a pff that exceeds the upper bound, is assigned more page frames from the free frame list, and a process that has pff which is lower than the lower bound, has its number of allocated page frames reduced (returned to the free frame list)**

Localities, Working Set & Thrashing

Thrashing



- **Thrashing: occurs when processes does not have enough assigned memory page frames and the page fault rate is very high**
    - Leads to *low CPU utilisation since the OS spends most of its time swapping pages in and out between main and secondary storage*
- Graph
    - Degree of multiprogramming is just process switching
    - **The higher the amount of multi-programming, the more processes the OS has scheduled to run, and thus the more time spent switching between the execution of each process and thus swapping pages out of memory frames for one process to accommodate the running of another process**
- WC scenario (rhs) – when the degree of multi-programming is increased by too much, causing uncontrollable thrashing, and eventual crashing of all process execution (i.e. zero CPU utilisation)
- **Thrashing occurs when a very high page fault rate causes the OS to spend most of its time servicing page fault requests rather than actually executing the pages of processes in main (physical) memory**
    - System spends most of its time servicing page faults, and little time doing useful work
    - Main memory overcommitted

- Enough memory – but some bad page replacement algorithm, incompatible with program behaviour, is used
    - Swop out the wrong pages at the wrong time
    - OS spends more time swopping in correct pages than executing instructions within pages
- Thrashing is caused by a lack of main memory or when main memory is overcommitted
    - That is, if there is too little main memory, then as more processes are scheduled to run, then page frames are assigned to these processes
    - Though the page fault rate increases as a result of pages constantly being swapped in and out between process page frames (thus over-committing memory), to satisfy the execution of each process (i.e. high degree of multi-programming)
- Scheduling of new processes to run usually occurs in the first place as a result of the OS monitoring the degree of multiprogramming, observing that the CPU is under-utilised and thus scheduling the execution of more processes
    - OS sees CPU is under-utilised so adds more processes
    - But too many active processes only makes problem worse

Prevention

- Option 1:
    - **Use local or priority page replacement**, so as page frame allocation only occurs for a specific process
    - Thus, **if one process starts thrashing, then it is limited to that process** and will not affect other processes currently scheduled to run
    - However, this **does not stop of the problem of N processes independently thrashing** at the same time
- Option 2:
    - **Dynamically allocate every process as many page frames as it needs**, meaning thrashing will not occur
    - How w can this be made possible with limited main (physical) memory?
    - Use dynamic memory allocation and load-balancing methods known as working sets

Locality

https://www.youtube.com/watch?v=3XmALGPW0zA

Why does Virtual Memory work?

- Locality – **all program (process) memory access patterns have temporal and spatial locality**
    - That is, **for any given process, only a few memory chunks (pages) of the process are used during a small time period (temporal locality) and these pages tend to be located close to each other in terms of memory address locations (spatial locality)**
- Thus, the locality of references implies that memory page references for a process tends to be grouped, where group of pages accessed along a given time slice are called the working set
    - **A working set defines some minimum number of pages needed for a process to behave well (i.e. minimising page faults and thus avoid thrashing)**
- Hence, if the working set size for each process is sufficiently less than main memory size then this generally leads to good process performance, otherwise the process can

potentially thrash, which in the worst case causes process execution to halt as all process memory pages are swapped (copied) in and out continuously

Principle of Locality

- Locality occurs because of the way in which computer programs are created for algorithmically solving problems
    - Generally, related program data is stored in nearby memory locations in storage
- **Most program execution is sequential**
    - A common algorithmic approach involves the processing of several items, one at a time
    - *This means that if a lot of computational processing is done by the program (process), a single data-structure (memory reference) will likely be accessed many times, which implies temporal locality of reference*
- **Programs tend to reuse data and instructions near those they have used recently**
    - *CPU sequentially moves between data-structures during execution of processes, which implies spatial locality of reference, since memory locations are typically read in batches*
- Most iterative constructs use relatively few instructions
- **Locality often occurs because program code often contains loops that tend to reference arrays or other data structures by indices**
    - Sequential locality is a special case of spatial locality that occurs when relevant data elements are arranged and accessed linearly
    - Example – the simple traversal of elements in a one-dimensional array, from the base address to the highest element would exploit the sequential locality of the array in memory
- When processing large data structures – main cost is sequential processing on data elements
- Equidistant locality is a more general case that occurs when the linear traversal is over a longer area of adjacent data structures with identical structure and size, accessing mutually corresponding elements of each structure rather than each entire structure
    - Example – when a matrix is represented as a sequential matrix of rows and the requirement is to access a single column of the matrix

Principle of Locality and Memory

- **Temporal locality: a given memory location just referenced will likely be referenced again in the near future**
    - e.g., statements in loops, stacks, linked-lists and similar data-structures
- **Spatial locality: if a memory location was just referenced, nearby memory locations will likely be referenced in the near future**
    - e.g., sequentially executed statements, array and linked list (and similar data-structure) elements in program code

Locality Example

- **Data:** Reference array elements in succession **(spatial)**.
- **Instructions:**
  - Reference instructions in sequence **(spatial)**.
  - Cycle through loop repeatedly **(temporal)**.

```
sum = 0;
for (i = 0; i < n; i++)
   sum += a[i];
*v = sum;
```

- The code segment shows a for loop that calculates a sum value via consecutively adding values from an array elements
- In this case the array elements (memory locations) are referenced in succession, meaning that there is spatial locality in the program data
  - In terms of the program instructions (statements), these are executed in sequence and the statement in the loop is accessed repeatedly
  - This implies spatial locality in that the same memory locations are accessed multiple times or otherwise references are made to memory addresses close to each other (as a result of sequential statement execution in a given block of code)
- Code segment also presents an example of temporal locality, in that the for loop executes the same statement repeatedly within a short time period

Working Set

Using Locality: Working Set Model

- **Locality in program design and execution assumes that recently referenced pages are likely to be referenced again soon, thus the OS only needs to keep such recently referenced pages in memory**
  - This is the working set
- **Using the working set model for page replacement, a process is allowed to execute only if its working set fits into (currently free) main memory**
- The working set model performs **implicit load control** since in many operating systems the **size of the working set is dynamically changed as the memory demands of processes change**

Program Memory: Working Set

- **Program (process) memory access is defined by temporal and spatial patterns**
  - i.e. by spatial and temporal locality

- Given this, the working set is defined as sets of pages accessed in a given time period (for a given process)
- That is, where such access is defined by a minimum number of pages needed for the process to behave well,
    - i.e. some minimum number of pages loaded into main memory, for the given process, that ensures minimal page faulting and thus no thrashing
- **Determining working set is a core part of having a dynamic, load-balancing algo**

Working Set Algorithm

- Have a method for estimating the working set for each process
    - Algo assumes that can estimate the working set size for each process
- Estimate: |WS(0, w)| for each process:
    - 0 is the start time of working set and w is the size of the working set, measured in number of page references
    - Process starts only if can allocate it enough frames
        - A process is allowed to start only if it can be allocated so many (w) page frames
    - Use a local page replacement algorithm
        - Ensures that the working set is only occupying the process's frames
        - Number of frames assigned to each process in the first place, depends on the working set size
    - Track each process's working set size
    - Re-allocate frames among processes dynamically
        - According to changing process memory demands
        - Larger working set size => more frames allocated
        - Smaller => less frames allocated

Working Set Definition

- Set of pages process currently *needs* :

WS(t, w) = { **P** pages, where **P** was referenced in time interval: **(t, t - w)** } ;

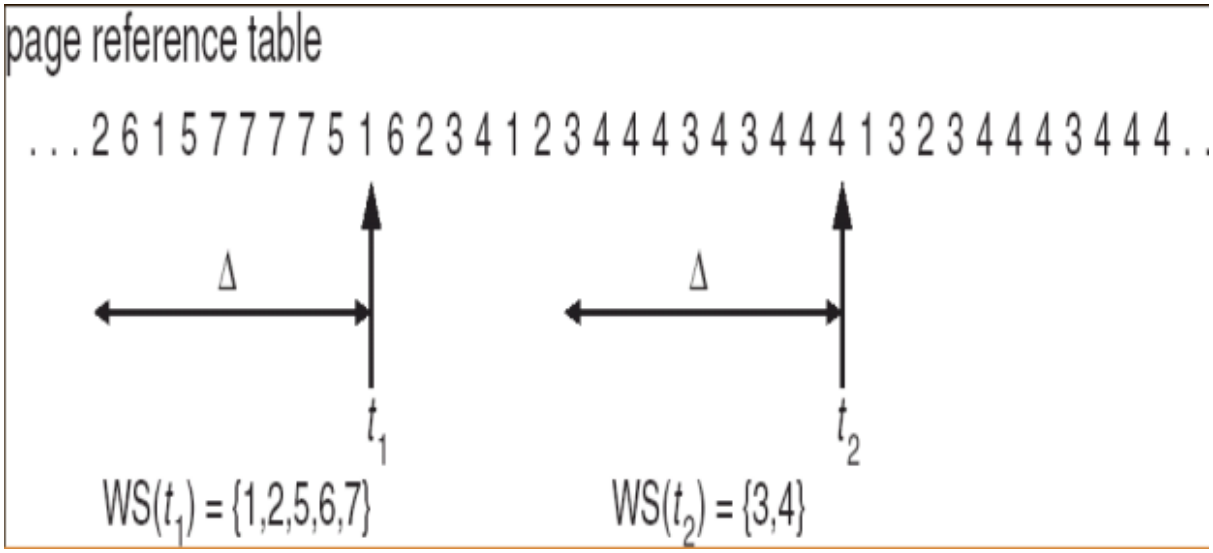
- **t:** time ;
- **w:** working set window (measured in page references) ;

- *Page is in **WS** only if referenced in last **w** references.*

- The Working Set (WS) for a given process is defined as the set of pages (P) the process currently needs (loaded into main memory)
    - To run and not thrash
- WS(t,w) is start time for process

- In the WS definition given, t is the current time for which the process has been executing, and w is the working set window (measured as a number of memory page references)
    - The size of w has to be estimated for each process, and any page (for the given process) is only in the working set (WS) if it was referenced in the last w page references
- For a time interval t, t-w (i.e. for past w references), measure the number of page references

Working Set Example

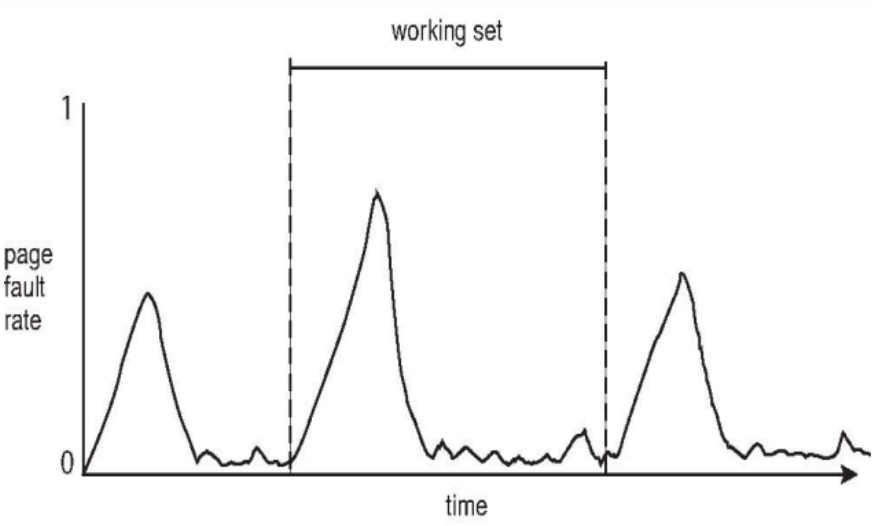


- Have two working set examples for a given set of page references (string of values at the top)
- In each case the size of the working set is indicated by the Δ value (working set window)
    - $WS(t_1)$ indicates the list of page references specific to the first working set that has been currently been running until time t_1
    - Similarly, $WS(t_2)$ indicates the list of page references specific to the second working set that has been currently been running until time t_2
- In both cases, note that the size of the working set is the same, but the number of unique page references is different (5 for the first working set and 2 for the second working set)
    - This is result of varying locality and computational complexity for these different processes

Working Set Size

- **The working set models program (process) behaviour in that it reflects the dynamic locality of process memory usage**
- Low temporal and spatial locality is indicative of more pages being referenced by the process
    - i.e. **if memory pages referenced are not close to each other in time of reference or space (memory addresses) then this usually indicates a computationally large and complex process that requires many pages to execute**
- If a process has a low temporal and spatial locality for long periods, this indicates that the process is computational large and complex and thus requires a large working set size
- If the working set (minimum number of pages for the process to run well) is not in memory, then there is a risk of the process thrashing

Estimate Working Set Size



- How many pages does a process need?
    - i.e. its working set size?
    - In order to run and not thrash
- **Finding the correct working set size (w) can be tricky and depends on the memory demands and computational complexity of the given process**
    - Informally, the working set size is just a minimum number of pages a process must use in order to effectively run (without thrashing)
    - However, **the working set size can also be stated as all pages referenced in the past w seconds of process execution**
- What should w be?
- Graph – shows that there is a direct relationship between the working set size of a process and its page fault rate
    - **Specifically, groups of page faults are typically clustered within program (process) spatial locality, meaning that page faults necessarily occur to bring all the required pages into memory so as the process can execute before the execution of another process is switched to**
- Given that we define working set size (w) as the number of pages referenced in the past w seconds of process execution, we must consider how many page faults are likely to occur per second
- To select w – consider:
    - One (1) page fault occurs (on average) per 10 milliseconds
        - E.g. approximately 2 million instructions executed by the CP U
    - Then w must be larger than 10 milliseconds in order that the given process can swap in the required pages necessary to run

Problems with Set Size

- **If w is too small then the working set will not cover enough of the process locality, meaning the process will not have the minimum required number of page frames to run effectively**
    - That is, *if working set (required minimum number of pages) is not currently in memory then there is a risk of the process thrashing*

- Whereas, **if w is too large then the working set will cover several process localities, meaning that the process will have more memory frames than it needs to run effectively, which leads to inefficient use of page frames**
- Note that, in general, if the memory demands of any given process exceeds the memory page frames allocated to the process then thrashing will likely occur
    - Thus, a popular anti-thrashing policy is to temporarily swap out and suspend any process that has increasing memory demands (exceeding allocated page frames)

Summary

- Program (process) memory access patterns typically high have temporal and spatial locality
    - A locality can be simply be defined as a set of pages (closely related in terms of address space or time of access) being accessed within a short time period
- Different processes have different temporal and spatial localities, defined by the process size and computational complexity
    - The complexity and memory requirements for one process may also change during its execution meaning that the process locality can change during its execution
- For a given process, the group of pages accessed during a given time slice is called the working set – which ideally corresponds to a process' locality (if the size of the working set is properly defined)
    - If the working set (locality) size for a process (or all processes) does not fit in the memory page frames allocated to the process (or processes) then thrashing will occur