COMPUTER ARCHITECTURE

1946 – ENIAC [pg. 3]

First electronic general-purpose computer.
Designers: Mauchly and Eckert

Von Neumann Architecture [pg. 9]

1. A *Central Processing Unit* performs operations and controls the sequence of operations.
2. A *Memory Unit* contains code and data.
3. Some kind of *Input and Output* mechanisms (I/O).

Memory Unit or RAM (Random Access Memory) [pg. 13]

- The basic unit of memory is the byte (or octet, or octad, or octade) 1 Byte = 8 bits
- The memory is basically an indexed array of bytes.
- The memory contents have various useful meaning: integers, addresses (pointers), instructions understood by a CPU, etc.
- Once a program is loaded in memory its address space contains both code and data.
    Indirection: The content at a memory location is the address of another memory location: we call this a pointer/reference.

Central Processing Unit (CPU) [pg. 36]

Registers: a few bytes of memory in which things can be written/read extremely rapidly.
Arithmetic Logic Unit (ALU): the hardware that knows how to compute stuff.
Control Unit: is the component in charge of controlling the program execution

Registers
Special-purpose registers: have well-defined purposed.
General-purpose registers: used to store user program data.
ALU
Many of its operations take 1 cycle, but several are more expensive (e.g., a division is expensive)
Control Unit
It uses to special-purpose registers:
Program Counter: Contains the address of the next instruction that must be executed next.
Current Instruction: The binary code of the instruction that is currently being executed.

Fetch-Decode-Execute Cycle [pg. 44]

1. The Control Unit fetches the next program instruction from memory using the program counter, and increments the program counter.
2. The instruction is decoded and signals are sent to hardware components (memory controller, ALU, I/O controller)
3. The instruction is executed: i) Values are read from memory into registers, if needed ii) Computation is performed by the ALU and results are stored in registers iii) Register values are written back to memory if needed.

Speed Up Things [pg. 77]

The RAM is Slow: Accessing a register is very fast e.g., a 4GHz CPU can update a register in 0.25 nanosecond (1 cycle). Accessing the memory takes about 10 ns. The memory is **~40 times** slower than the CPU. What does the CPU do while it's waiting for the memory to give it data? NOTHING!! (yes, this is a problem). This is the famous "*Von-Neumann Bottleneck".*

The Memory Hierarchy [pg. 86]

A trick to provide the illusion of a fast memory.

When a program accesses a byte in memory:

1. It checks whether the byte is in *cache*, and if so, it just gets it
2. Otherwise, the byte value is brought from the (slow) memory into the (fast) cache
3. The values around the byte are also brought into the cache

The Caches are managed solely by hardware.

The Memory (RAM) and the Disk are managed by software (i.e., the OS)

Locality [pg. 92]

Temporal Locality: A program tends to reference addresses it has already referenced. The first access is expensive, but each subsequent access is cheap.

Spatial Locality: A program tends to reference addresses next to addresses it has already referenced. The access to element i is expensive, access to elements i + 1, ... are cheap.

Memory Caches

Cache Hit: When a data item is found in cache.

Cache Miss: When a data item is not found in cache.

Direct Memory Access (DMA) [pg. 99]

Copies occur independently so that the CPU can do something useful while the memory copy is taking place. However, DMA controller and code executed by the CPU likely use the memory bus, thus, they can interfere with each other. But there are several solutions to that.

Current Architectures [pg. 106]

Because constructors cannot increase clock rate further (power/heat issues), our current CPUs are *multi-core*. Multiple "low" clock rate CPUs on a single chip.

OS OVERVIEW AND INTERFACES

- **Overview (*The three easy pieces, the Kernel, Kernel as an Event Handler*)**

1. Virtualization: Abstraction and Allocation [pg, 7]

The OS is a *Resource Abstractor*: it defines a set of logical resources that correspond to hardware resources, and well-defined operations on these logical resources.

The OS IS a *Resource Allocator*: : it decides who gets how much and when.

Why do we want virtualization? [pg. 10]

1. To make the computer easier to program
2. To provide each program the illusion that it is alone on the computer, going through its fetch-decode-execute cycle.

Virtualization allows for Multi-Programming [pg. 12]

Multi-programming: The OS-provided capability to execute multiple programs concurrently on a computer. Since multi-programming came about, a big issue has been...

2. **Concurrency [pg. 16]**: Juggling many things at the same time. It leads to deep/difficult/interesting issues within the OS.
3. **Persistence [pg. 19]**: the ability to store data that survives a program termination / a computer shutdown.

OS Kernel [pg. 21]

The OS is software, and the "core" part of this software is called the *kernel.* The kernel is in charge of implementing resource abstraction and allocation.
The kernel is code and data that always resides in RAM after boot:
It is NOT A RUNNING PROGRAM BUT AN EVENT HANDLER.

Programming Language for Kernel Development [pg. 26]

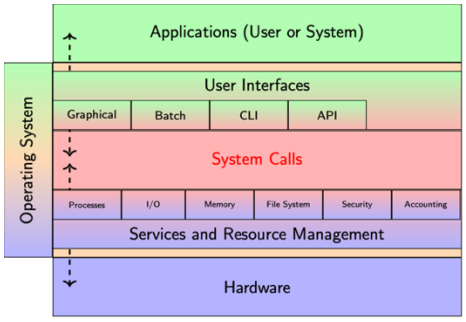
Initially, kernels were written in assembly only. Since 1960s: written in high-level languages (MS-DOS and alike being the exception).

What happens when you turn on your Computer? (pg. 44)

Two kinds of OS Events: Interrupts and Traps

Interrupts: Asynchronous because generated in real time from the "outside world". (e.g., "incoming data on keyboard")
Traps: Synchronous because generated as part of the fetch-decode-execute cycle from the "inside world" (also called exceptions or faults). An important kind of trap are *system calls.*

- **Interfaces**



Interactive User Interfaces

1. *Batch: Historical Human Interface [pg. 11]*
2. *GUI: Graphical User Interface*
3. *CLI: Command-Line Interface*

CLI [pg. 14]

Provides built-in commands: cd, bg, exit
Can start: 1) *System Programs*: "Simple" wrapper around a system call: sleep. "Complex" programs involving more than one system calls: rm, ls. 2) *User Programs*: anything else.

GUI [pg. 16]

Is the GUI part of the OS or not? Windows: YES! MacOS: YES! Linux: NO!

The System Call API (Application Programming Interface) [pg. 19]

"Lowest-level" interface to OS. Services GUI and CLI are built on top of this API as well as any user program.
Every (useful) program uses this API: either directly, or indirectly.

System Call [pg. 23]
Each system call is identified by a unique number. System calls in Linux are implemented in the Kernel as mixtures of assembly and C/C++. These numbers are stored in an internal table named the *syscall table.*

- There are system calls for all types of OS services (*Process Management, Memory Management, File Management, Device Management, Memory, Communications, Protection, etc*.)
- System calls make it possible to access virtualized hardware resources.

Measuring System Time with time
Real time = User time + System time + I/O time
System call interface: a set of useful functions that are "easier-to-use wrappers" around the raw system calls and that are provided in standard libraries. e.g., the fork()

- **History (Operating Systems)**

Early OSes [pg. 2]
Early OSes were just libraries: 1. Just some code as wrapper around tedious low-level stuff 2. No real abstractions, and in particular no virtualization 3. One program ran at a time, controller by a human operator. This was known as "*batch mode*"

System Calls [pg. 3]
Beyond Libraries: System calls. Realization that *user code* should be differentiated from *kernel code*, and that kernel code should be "special". In previous OSes, any program could do anything to any hardware resource!!

Multiprogramming [pg.4]
Multiprogramming led to the first "real OSes". Came about to improve CPU utilization (while program #1 is idling, program #2 should be able to utilize the CPU). Development of context-switching, of memory protection. Beginning of concurrency Development of UNIX.

OS Design Goals [pg. 10]
Abstractions: to using the computer convenient.
Performance: Minimize OS overhead (time, space). Often conflicts with the previous goal!
Protection: Programs must execute in isolation (comes from virtualization).
Reliability: The OS must not fail Thus OS software complexity is a concern (e.g., is it worth adding 2,000 lines of complex code to improve something by some epsilon?)
Resource efficiency: The OS must make it possible to use hardware resources as best as possible.

Design Principle [pg. 12]
One ubiquitous principle: separating mechanisms and policies

Policy: what should be done (e.g., an algorithm)
Mechanism: how it should be done (e.g., methods at your disposal)

Monolithic [pg. 14]
MS-DOS was written to run in the smallest amount of space possible, leading to poor modularity, separation of functionality, and security (e.g., user programs can directly access some devices and no difference in execution of user code and kernel code.)
Layered [pg. 21]
How many layers? Too many: high overhead/bad performance. Too few: poor security/modularity.
Microkernels [pg.23]
Major issue: increased overhead because of IPC
Modules [pg. 30]
Advantages of microkernels without the poor performance (communication does not use IPC).
Hybrid [pg. 33]
Don't stray too far away from monolithic, so as to have good performance.

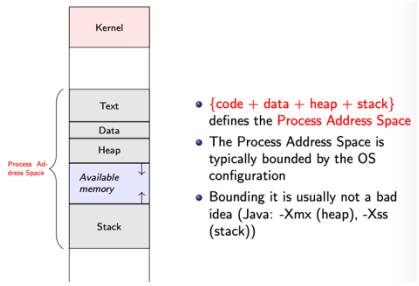
PROCESSES

- **The Process Abstraction**

Definition [pg. 2]
The *process:* the OS abstraction to virtualize the CPU.
- A program becomes a process when it is loaded into memory, at which point the fetch-decode-execute cycle can begin.
- Multiple processes can be associated to the same program
- A user can start multiple instances of the same program, and different users can start multiple instances of the same program

A Process is defined by… [pg. 8]
- The program, or code (also called text)
  A list of binary instructions, initially stored in an executable file and then loaded into RAM by the OS.
- The program (static) data
  The global variables and static local variables, which can be initialized or not.
- The content of all registers
  They represent the current state of the CPU in the current fetch-decode-execute cycle
  This includes the program counter (PC)
- The heap
  The zone of RAM in which new "objects" have been dynamically allocated (using malloc, new, ...)
- The runtime stack
  The zone of RAM for all bookkeeping related to method/procedure/function calls (more in the next slides)
- The page table

Process Address Space [pg. 10]



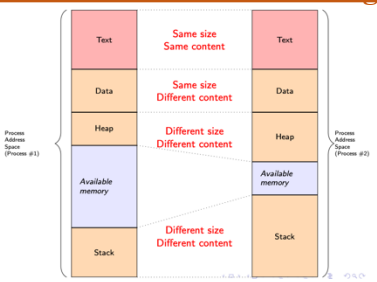
The Heap [pg. 11]

Can be handled by a memory manager (e.g., the JVM, a library) but ultimately it is the OS that provides dynamic memory allocations.

The Runtime Stack [pg. 12]

- It helps manage method/procedure/function calls and how to return from them.
- The code to manage the stack is generated entirely by the compiler/JVM.
- Items on the stack are pushed/popped in groups, or activation records, or frame
- An activation record contains all the bookkeeping necessary for placing and returning from one method/procedure/function call

Two Processes for One Program [pg. 17]



Process Life Cycle [pg. 18]

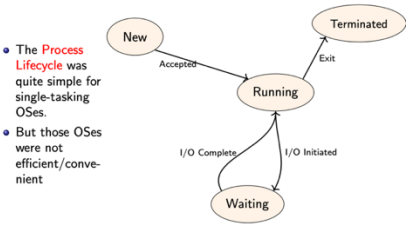
Each process goes through a lifecycle

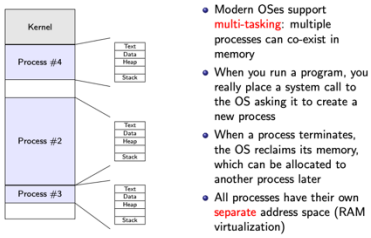
- A process can be in a finite number of different states
- There are allowed transitions between some pairs of states

Single-Tasking OSes [pg. 19]

OSes used to be single-tasking: only one process can be in memory at a time

Single-Tasking OS Process Lifecycle [pg. 32]





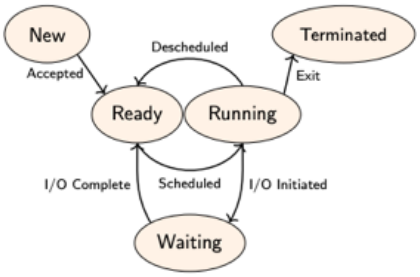
Multi-tasking (multi-Programming) [pg. 41]

Context-switching [pg. 46]

- Thanks to CPU virtualization, each program instance has the illusion that it's alone on the CPU
- A "running program" doesn't always actually "run".

Process Lifecycle [pg. 62]

- There are variations in state names across OSes
- Note that you go from Waiting to Ready, **not** to Running (When you do I/O, you "lose your spot")
- You must have this diagram in mind (just practice drawing this diagram by telling yourself the story, not by blindly memorizing the states and transitions!)



Process Control Block [pg. 70]

The OS uses a data structure to keep track of a process. This structure is called the Process Control Block (PCB) and contains:

1. Process state
2. Process ID (aka PID)
3. User ID
4. Saved Register Values (include PC)
5. CPU-scheduling information (see the "Scheduling" Module)
6. Some Memory-management information (see the "Main Memory" and "Virtual Memory" modules)
7. Accounting information (amount of hardware resources used so far)
8. I/O Status Info (e.g., for open files) ... and a lot of other useful things

The Process table [pg. 72]

The OS has in memory (in the Kernel space) one PCB per process

- A new PCB is created each time a new process is created and is destroyed each time a process terminates. The OS keeps a "list" of PCBs: the *Process Table*
- Because Kernel size is bounded, so is the Process Table. Therefore, it can fill up!

- **The Process API**

Process Creation [pg. 10]

Each process has a PID (a Process ID), which is an integer

- The PID is picked by the OS, and is increasing
- The C POSIX function *getpid()* returns the PID
- The PID of the parent of a process is called the *PPID* (Parent Process ID)

The fork() System Call [pg. 13]

The child is an almost exact copy of the parent except for:

1. Its PID (two processes can't have the same ID)
2. Its PPID (its parent cannot also be its grandparent)
3. Its resource utilization (set to 0 since it's just started)

The confusing part: fork() returns an integer value. fork() returns 0 to the child. fork() returns the child's PID to the parent.

Fork Bombs [pg. 42]
Best way to prevent forkbombs from killing your system: limit the number of processes that a user can create: **ulimit -u**.
The exec() Family [pg. 46]
This system call replaces the process image (i.e., the process address space) by that of a specific program.
They are dead…but alive! [pg. 60]
When a child process terminates, it remains as a zombie in the Terminated state.
A zombie lingers on until: 1. Its parent has acknowledged its termination; 2. or Its parent dies.
Process Terminations [pg. 63]
A process terminates itself with the exit() system call. All resources of the process are then deallocated by the OS. A process can also cause the termination of another process. This is done using signals and the kill() system call...
Signals and Signal Handlers [pg. 66]
Signals are software interrupts, i.e., a signal is an asynchronous event that the program must act upon in some way. (Ex: ^C on the command-line, access a invalid address, etc).
Some signals cannot be reprogrammed by the user: SIGKILL, SIGSTOP, SIGCONT, ...
wait() and waitpid() [pg. 72]
A parent can wait for a child's completion
The *wait()* system call blocks until any child completes and returns the pid of the completed child and the child's exit code.
The *waitpid()* system call blocks until a specific child completes. Can be made non-blocking
The SIGCHLD Signal [pg. 75]
The typical convenient way to avoid zombies altogether: 1. The parent associates a handler to SIGCHLD 2. The handler calls wait() 3. This way all children terminations are acknowledged.
Orphans [pg. 80]
A PID 1 doesn't leave zombies behind because it acknowledges all its (adopted) children's deaths!

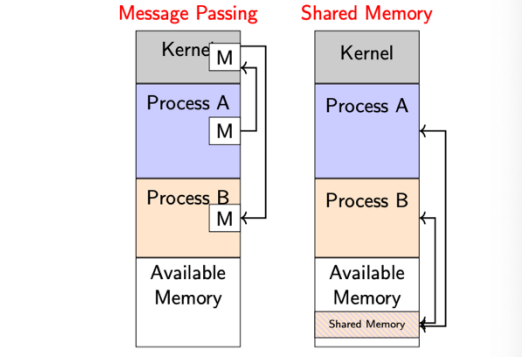
- **Inter-Process Communication (IPC)**

Communicating Processes [pg. 4]
So far, we have seen independent processes, but often we need processes to cooperate.
1. To share information (e.g., access to common data)
2. To speed up computation (e.g., to use multiple cores)

In general, the means of communication between cooperating processes is called *Inter-Process Communication (IPC)*

Two Broad Communication Models [pg. 10]

- Example: Process A needs to communicate with Process B



Message Passing	Shared Memory
☹ Requires data copies (user space → kernel space → user space)	☺ Fast because no memory c is needed
☹ High overhead because one system call per operation (`send` and `receive`)	☺ Low overhead because syst calls only to set up the shared memory region (after that it's memory reads and writes)
☹ Cumbersome for developers: code will be sprinkled with send and receive everywhere	☺ Easy for developers: Just read/write to shared memory region
☺ Preserves memory protection / isolation across processes	☹ Violates the principle of memory protection between processes, which is dangerous bug-prone (more on this in Synchronization module)

Pros and Cons [pg. 14]

A Word on Abstraction and API Design

There are many design options for IPC system calls, especially for Message Passing.

1. *Asynchronous vs. Synchronous communication*: is the sender "stuck" until the receiver has received or not?
2. *Direct vs. Indirect*: as a sender, do I send to another process (i.e., a PID) or to some abstract "mailbox" from which more than one process could read?
3. *Buffered vs. not Buffered*: can the sender overwrite the data it has just sent safely? (at the cost of an extra memory copy)

Higher-Level IPC Abstractions [pg. 17]

All OSes provide system calls for *Shared Memory*, and *Message Passing.*
There are also higher-level, convenient abstractions. Let's talk about two of them: *Remote*, and *Procedure Calls (RPC) Pipes.*

Remote Procedure Calls (RPC)[pg. 19]

RPC provides a procedure invocation abstraction across processes (and even across machines)

- i.e., a process invokes a procedure in another process

It is used of course for client-server applications. Performed through a client stub with automatically generated code to: **Marshal** the parameters (*structured data → bytes stream*). **Send** the data over to the server. **Wait** for the server's answer. **Unmarshal** the returned values (*bytes stream → structured data*).

Pipes [pg. 22]

A pipe is an abstraction of an actual physical pipe with data that flows between two processes. It has two ends: A *write end* and a *read end*. A process can write a stream of bytes to the write end (not structured as separate messages). And a process can read from the read end. When a process closes one of the ends, the other will get some EOF (End Of File) notification.

File Descriptors [pg. 25]

Each process has a set of opened files. Each opened file is associated with an integer called the *file descriptor*. The file descriptor is the index in the array of the process' opened files. This array is stored in the process' PCB!

Every process starts with three already opened files:

1. File descriptor 0: standard input (stdin)
2. File descriptor 1: standard output (stdout)
3. File descriptor 2: standard error (stderr)

Every programming language provides access to these (e.g., Java's System.err)

The Shell uses pipe a lot! [pg. 41]

dup() allocates a new file descriptor that's an alias for an existing file descriptor. Typical example sequence: create a file descriptor, say x, for something, close file descriptor 1 (stdout), call dup so that file descriptor 1 is now the same as x.

- **Process Mechanisms**

Direct Execution is not a Good Idea [pg. 16]

- Figure 6.1 in OSTEP shows a simple timeline for an OS to run a program (slightly modified below):

OS	Program
Create PCB and add it to Process Table	
Allocate memory for process	
Load program into memory	
Set up stack with argc/argv	
Clear registers	
Start fetch-decode-execute (call main)	
	Run main()
	Return from main
Free memory of process	

Problem #1: If the process needs to access hardware resources, then the only option is to give the process full access to the hardware. A bug in a user program could corrupt hardware status, bring the machine down, overwrite data...

Problem #2: How do we kick a process out of the CPU and give the CPU to another process? We need to limit the way in which a process runs on the hardware. In other words, we need mechanisms for virtualizing the CPU to solve both problems above.

Limited Execution: Restricted Operations [pg. 19]

This is done by building CPUs that have **two kinds of instructions**!

Unprotected instruction that a program can execute at any time.

Protected (or Privileged) instructions that do "special" things and a program can't just execute.

User Mode vs Kernel Mode [pg. 23]

1. The *User Mode* where protected instructions cannot be executed. *User code* executes in user mode.
2. The *Kernel Mode* where all instructions can be executed. *Kernel code* executes in kernel mode.

The mode is indicated by a status bit (the mode bit) in a protected control register in the CPU. The CPU checks the mode bit before executing a protected instruction.

In the Fetch-Decode-Execute cycle, steps are added to the Decode stage:

- Decode instruction.
- If the instruction is protected and the mode bit is not set to "Kernel mode", abort and raise a trap, otherwise, execute the instruction.

Which instructions are protected? [pg. 25]

The following are protected instructions on modern OSes:

1. Updating the mode bit
2. Halt the CPU
3. Update CPU control registers
4. Change the system clock
5. Read/Write I/O device control/status registers
6. In general, interact with hardware components

Therefore, all these operations can only happen in Kernel mode and only kernel code can use them.

But what about system calls? [pg. 27]

Question: How do we do system calls? This is exactly why the CPU has a "system call instruction". This instruction is a trap, to which the OS must react.

The Trap Table [pg. 29]

At boot time, the OS initializes a Trap Table

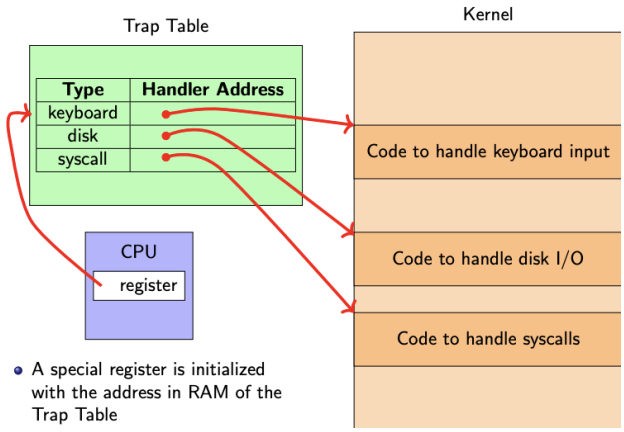
1. The Trap Table is stored in RAM (as an array of consecutive bytes), and the CPU has a register that points to it.
2. For each event type that the CPU could receive, this table indicates the address in the kernel of the code that should be run to react to the event.
3. Whenever an event occurs the CPU can just do:
   a. Look at the Trap Table in RAM
   b. Lookup the entry in the Trap Table for the event and find the kernel handler's address
   c. Set the mode bit to "Kernel"
   d. Jump to the kernel handler and fetch-decode-execute it.

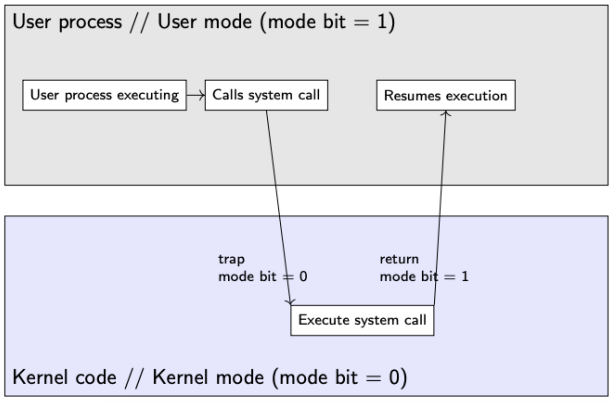
The trap instruction [pg. 34]

A CPU has an instruction often called the "*trap instruction*". The trap instruction does:

1. Set the mode bit to "kernel"
2. Jump to the "handle system call" kernel code
3. Set the mode bit to "user"
4. Jump back to user code

There are many system calls, but a single system call handler. Therefore, the user must specify which system call to run as a system call number. The handler checks that the system call number if valid, and then jumps to the corresponding kernel code.





Limited Execution: Restricted Time [pg. 44]

The approach is to enforce some switching between processes

It's about Regaining Control [pg. 47]

It's not so easy: if a process is running on the CPU, the OS is not running! Meaning Kernel code is not running.

The Cooperative Approach [pg. 51]

For instance, each time a process places a system call, then by definition Kernel code is running, and then the OS can take whatever action.

-There could be a yield() system call to just give up the CPU.

The old Macintosh OS is a famous example that used this approach. On an old Mac, a while(1){} program will lock up the machine and we'll need to reboot! How can we avoid this? Answer: with a timer.

The Timer Interrupt [pg. 54]

To deal with non-cooperative processes, whenever the OS starts the fetch-decode-execute cycle of a process it sets a timer.

When the timer goes off, a trap is generated, so that the CPU will stop what it's doing and notify the OS. The kernel has a handler for this trap. This handler is the way in which the OS regains control. Setting and enabling/disabling the timer are privileged instructions.

Next up: how to switch between processes

Context Switching [pg. 57]

The mechanism to kick a process off the CPU and give the CPU to another process is called a *context switch*:

a. Save the context of the running process to the PCB in RAM (i.e., all register values).
b. Make its state Ready
c. Restore from the PCB in RAM the context of a Ready process (i.e., register values)
d. Make its state Running
e. Restart its fetch-decode-execute cycle

The context switch code is in assembly. It should be as fast as possible because it is pure overhead. Nowadays it's under 1µs. Context switch is a mechanism, and deciding when to context switch is a policy, which is called *scheduling*.

Event	Time	Process #1	OS	Process #2
-	1	Running	-	Ready
Timer!	-	Running	-	Ready
-	2	Ready	(Context switch begin)	Ready
-	3	Ready	Save state in PCB#1	Ready
-	4	Ready	-	Ready
-	5	Ready	Restore state from PCB#2	Ready
-	6	Ready	-	Ready
-	7	Ready	(Context switch end)	Running
-	8	Ready	-	Running
-	9	Ready	-	Running
-	...	...	...	...
-	30	Ready	-	Running
Timer!	31	Ready	-	Running
-	322	Ready	(Context switch begin)	Ready
-	...	...	...	...

THREADS

Concurrent Computing [pg. 4]

Several operations are performed during overlapping time periods. As opposed to sequential execution, which is: one operation runs to completion, followed by another computation to completion, etc.

Concurrency: A feature of a program that can do multiple things "at the same time".

A program is concurrent if it consists of units that can be executed independently (but may need to synchronize among each other to co-operate).

What is Concurrency Used for [pg. 12]

Make programs faster

- Running multiple activities at once can use the machine more effectively because there are multiple hardware components
- e.g., while one activity computes on the CPU, another activity sends data to the network card

To make programs more responsive

- Structuring a program as concurrent activities can make it more responsive because while one activity blocks waiting for some event, another can do something
- e.g., in some server, spawn a new activity to answer each client request

Concurrency with Processes [pg. 15]

Processes run concurrently on the computer. And therefore, they can be used for better responsiveness and speed. But because the OS virtualizes memory, processes don't share memory naturally. This can make it difficult to program processes that have complicated cooperative behaviors, and so come *threads*...

Threads [pg. 20]

A *thread* is a basic unit of CPU utilization within a process

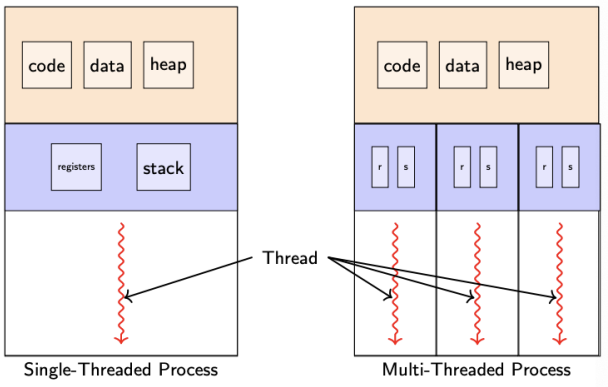
Multi-threaded process: Concurrent execution of different parts of the same running program.

Each thread has its own...

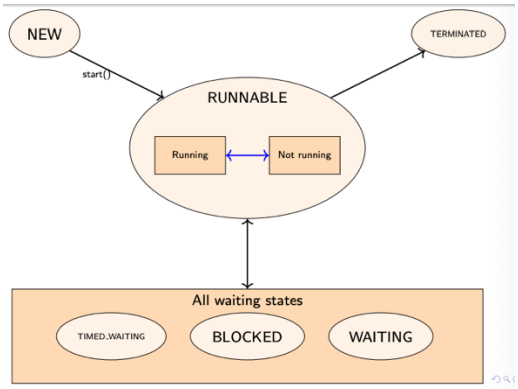
a. Thread ID (assigned by the OS)
b. Program Counter (which instruction it currently executes)
c. Registers Set (which values are stored in registers)
d. Stack (bookkeeping of its function/method invocations)

The above fully defines "what a thread is doing right now". But it shares with other threads in the same process:

a. the code/text section
b. the data segment (global variables)
c. the list of open file descriptors (at the moment of thread creation)
d. the heap
e. the signal behaviors (handlers)



Typical representation



Java Threads: Thread States - Thread::getState(

Advantages of Threads vs Processes [pg. 31]

Resource Sharing

1. Threads "naturally" share memory
   - Provides a direct Shared Memory IPC mechanism
   - No need for special system calls
2. Having concurrent activities in the same address space is very powerful
3. It makes it possible to implement all kinds of concurrent behaviors

Economy

1. Creating a thread is cheap
2. Context-switching between threads is cheaper than between processes
3. So if you can do with threads what you can do with processes, then you likely can do it a bit faster

Drawbacks of Threads vs Processes [pg. 34]

1. If one thread fails with an error/exception which is not managed, all threads (and therefore the whole process) fail
2. Threads may be more memory-constrained than processes

   Since they execute in the same process address space, and an OS can bound the size of a process' address space, but that's typically not a big deal
3. Threads do not benefit from memory protection

This is a feature, since you want them to see the same memory, but it can cause difficult bugs.

User Threads vs. Kernel Threads [pg. 44]

Threads can be supported solely in User Space (*User Threads*)
Advantage: low overhead (e.g., because no system calls)
Drawbacks:

1. If one thread blocks, all other threads block
2. All threads run on the same core (because the OS doesn't know that there are threads within a process)

All OSes today provide support for threads (*Kernel Threads*)
The kernel provides data structures and system calls to handle threads

Threads Libraries [pg. 46]

Thread libraries provide users with ways to create threads in their own programs (so that users don't have to place the low-level system calls necessary to create threads)
Ex: Java: *Java threads* (implemented by the JVM, which relies on Pthreads)

Java Threads [pg. 49]

Java does it by providing Thread-related classes and interfaces
There is a *Thread class*
There is a *Runnable interface*
There is a Callable interface
There is an *ExecutorService interface*

Java Threads Takeaways [pg. 59, 73]

-It's wise to use Runnable, rather than extend the Thread class because Java disallows multiple inheritance.
-To launch or spawn a Thread/Runnable it is necessary to call the start() method (instead of run()) If you call run(), it's just a regular method call that does not create a thread but will happily run the method's code!! The start() method hides all the "details" of thread creation
-ExecutorServices are powerful, and very convenient to run things in threads without having to do much work.

Multi-Threading Programming Challenge [pg. 75]

Major challenge: You cannot make any assumption about thread scheduling, since the OS is in charge. And what the OS does depends on the hardware and on other running processes .

Major difficulty: You may not be able to reproduce a bug because each execution is different! Therefore, you may think your code is working, but that's because you haven't been able to observe the bug yet.

Java Threads/ Kernel Threads [pg. 77]

The JVM is itself multi-threaded!
-The JVM has a thread scheduler for application threads, which are mapped to kernel threads. Several application threads could be mapped to the same kernel thread.
-That thread scheduler runs itself in a dedicated thread
-The OS is in charge of scheduling kernel threads

Linux/MacOS X Threads [pg. 98]

Linux does not distinguish between processes and threads: they are called tasks.
The *clone()* syscall is used to create a task
It can be invoked with several options, called the flag

It is not possible to perform an asynchronous cancellation of a thread.
The *Thread::stop()* method is deprecated, you cannot kill a Java thread. Instead, you can do deferred cancellation. This is because asynchronous cancellation is dangerous.

Java Thread Cancellation [pg. 103]

SCHEDULING

- **Scheduling Basics**

CPU Scheduling [pg. 4]

CPU Scheduling: The process by which the OS decides which processes/threads should run (and for how long)
-Necessary in a multi-programming environment
-Reminder: only READY processes/threads can be scheduled

The policy: the *scheduling strategy*
The usual broad goal is to improve system performance and productivity, including:
Maximize CPU utilization (whenever the CPU idle, pick a READY jobs to run)
The mechanism: the *dispatcher*
The OS component that knows how to switch between jobs on the CPU.

Long-term/ Short-term Scheduling [pg. 17]

Long-Term Scheduling: done by non-OS software (Job Schedulers)
Long-Term Scheduler
-Selects jobs from a submitted pool of jobs and loads them to memory
-Orchestrate jobs executions in the long term
-Executed every [10 minutes, or hour...]
-Can construct complex schedules
-Uses sophisticated decision algorithms

OSes: Short-Term Scheduler (or CPU Scheduler)
Short-Term Scheduler
Selects already-in-memory jobs to run
-Orchestrate jobs executions in the very short term
-Executed every [10's of milliseconds,]
-Cannot make complex decision
-Use simple decision algorithms

CPU or I/O Burst Cycles / CPU- or I/O-bound jobs [pg. 25]

I/O-bound job: a job that is mostly waiting for I/O with mostly short CPU bursts.

CPU-bound job: a job that is mostly using the CPU with mostly short I/O bursts.

Job Turnaround Time [pg. 29]

CPU-bound jobs care about turnaround time, but for other jobs perhaps this metric makes little sense e.g., your Web browser! $T_{urnaround} = T_{completion} - T_{arrival}$

FCFS (or FIFO) [pg. 30]

Say we have 3 jobs, A, B, and C, each running for 10 seconds. These jobs are in some order in the Ready Queue, say, A, B, C. The average turnaround time is:
$T_{bturnaround} = [10+20+30]/3 = 20$

This is only 2x more than the average job duration

The Problem with FCFS: Convoy Effect

Let's now remove one of our assumptions:
A takes 100 seconds
The average turnaround time is:

$T_{bturnaround}$ = [100+110+120]/3 = 110
2.75x more than the average job duration.
This *Convoy Effect* is basically the "Stuck behind somebody with a full cart at the supermarket" problem.

Shortest Job First (SJF) [pg. 32]

We can fix our problem by sorting: **dispatch shortest jobs first**
The average turnaround time is $T_{bturnaround}$ = [10+20+120]/3 = 50

Arriving at Different Times

Say A arrives at time 0, and B and C arrive at time 10. At time 0 we have to dispatch job A. The average turnaround time is:
$T_{bturnaround}$ = [100+(110-10)+(120-10)]/3 = 103.3.

Preempting Jobs [pg. 34]

A simple algorithm: *Shortest Time-to-Completion First* (STCF)
Whenever a job arrives, if it has shorter time-to-completion than the running job, then preempt the running job and run the new job. Whenever a job finishes, run the job with the shortest time-to-completion.
Solves the "Stuck behind big cart at the supermark" problem. But now we have starvation problem: a job may never run.

Round-Robin (RR) [pg. 38]

Let a job run for at most a little bit, then preempt it, then run the next job, etc. So we don't need to know how long a job runs for.
The (maximum) amount of time a job runs for before preemption is called the time slice, time quantum, or scheduling quantum.

Time Quantum [pg. 39]

Too large: poor response time
Too small: too much context-switching overhead
New Metric of "goodness": *Response Time*
$T_{response} = T_{firstrun} - T_{arrival}$

Dealing with I/O [pg. 45]

When a job issues an I/O operation, it vacates the CPU. When the I/O operation completes, the job is back in the *Ready Queue* and will be given the CPU at some point in the future.
Essentially, **each CPU burst is treated as a job**.

- **Advanced Scheduling**

Multi-Level Feedback Queue (MLFQ) [pg. 4]

We want jobs to be demoted or promoted to lower/higher queues when they stop/start being less/more interactive **Solution**: Use priority levels, and use one Round-Robin Ready Queue per level.

1.If Priority(A) > Priority(B) then A runs and B doesn't
2.If Priority(A) = Priority(B) then A and B run in Round-Robin

-If a job uses its full time quantum, it is demoted
-If a job does not use it's full time quantum, then it stays at the same priority level.

A higher-priority job always preempts a lower-priority job. Problem: Starvation. A simple solution: every S seconds move all the jobs to the top-priority queue, and let them trickle back

down. This is called a *Priority Boost*. How many queues? Which time quantum for each queue? What to use for S? **Typical approach**: larger time quanta for lower-priority queues.

Multi-Processor Scheduling [pg. 22]

All our processors are multi-core. So, OSes need to do scheduling across multiple cores.
However, having jobs jump around cores makes cache use inefficient.
-A job runs on Core 1 and has its data in the cache: experiences cache hits during its execution.
-Then it does some I/O, goes back to the *Ready Queue*, and get scheduled on Core 2. It will get many cache misses: terrible performance.
Each job has some *affinity* to some core: the core at which it has some data in cache.

Linux Scheduling[pg.25]

O(1) *Scheduler*: multiple queues, tricks to make quick decisions, accounting of CPU usage by each job, akin to MLFQ .
CFS (Completely Fair Scheduler): stores jobs in a red-black tree instead of queues, implements proportional-share approach for fairness.
BFS (BF Scheduler): simple algorithm, single queue, also focused on fairness (never made it to the mainstream kernel releases.

SYNCHRONIZATION (Race Conditions)

False Concurrency [pg. 3]

Provides the illusion of concurrency to a human. Increases CPU utilization (while a process/thread is blocked on I/O, another can run)

True Concurrency [pg. 4]

Our computers are multi-core, so we can have True Concurrency. Still with False Concurrency within each core.

True/False Concurrency [pg. 8]

The programmer should not have to care/know whether concurrency will be true or false
A multi-threaded program should reach higher interactivity and performance with True and/or False concurrency
-Unless all threads are I/O-bound, in which case there could be no concurrency (e.g., there is only one disk)
Concurrency is not only about cores: there can be concurrency between any two hardware resources

A "let's just add threads and things will be better/faster" approach can work sometimes
-The OS makes it all transparent because it virtualizes the CPU

Two Threads *(Race Condition)* [pg. 53]

Instruction	Value of A	Value at (@)
	undefined	0
LOAD (@), A	0	0
ADD A, 1	1	0
STORE A, (@)	1	1
LOAD (@), A	1	1
Context Switch #1 Save blue registers: A = 1, PC, ... Restore red registers: A = undefined, PC, ...		
	undefined	1
LOAD (@), A	1	1
ADD A, 1	2	1
STORE A, (@)	2	2
LOAD (@), A	2	2
Context Switch #2 Save red registers: A = 2, PC, ... Restore blue registers: A = 1, PC, ...		
	1	2
ADD A, 1	2	2
STORE A, (@)	2	2

This is WRONG!!

We executed 3 ADD instructions
We executed 3 STORE instructions
(just like the 1-thread execution)
Yet our final value in RAM is 2 and not 3!!

Just because the OS did Context Switch #2 at the "wrong" time

Race Condition [pg. 55]

In this case the bug is called a *lost update*. The outcome depends on when context-switches occur. Such non-deterministic bugs make concurrent programming difficult.

Lost Update [pg. 57]

In general when a thread does x+=a and an another does x+=b three things can happen:
-Both updates go through and x is incremented by a+b
-The x+=a update is lost and x is incremented only by a
-The x+=b update is lost and x is incremented only by b

Example: Two variables: a and b, both initially set to 1
Thread #1: a+=1; b=a+2;
Thread #2: a-=1; Once both threads are finished, the values of a an b are printed.

Interleaving

a = 1	b = 3
a = 1	b = 4

Lost Updates

a = 0	b = 2
a = 2	b = 4
a = 0	b = 4

New Concept: Critical Section [pg. 65]

We want to make parts of the source code where a race condition can happen into critical sections. A *critical section* is a region of code in which only one thread can be at a time.
-If a thread is already executing code in the critical section, then all other threads are "blocked" before being allowed to enter the critical section
-Only one thread will be allowed to enter when a thread leaves the critical section
- A critical section does not have to be a contiguous section of code

Critical Section [pg. 67]

Common misconception: *A critical section corresponds to a variable*. This is incorrect: Critical section corresponds to section(s) of code.

There are three requirements to execute critical sections:

Mutual Exclusion: If a thread is executing in the critical section, no other thread can be executing in it

Progress: If a thread wants to enter a critical section, it will enter it at some point in the future

Bounded Waiting: Once a thread has declared intent to enter the critical section, there should be a bound on the number of threads that can enter the critical section before it

The Kernel is not Immune to Race Conditions [pg. 70]

-Consider a process that places a system call
-It begins running kernel code
-And then a context switch happens!

Modern kernels allow the above (*preemptive kernels*). They must use critical section.

Critical Section Mechanisms [pg. 74]

What we need to are enter critical section() and a leave critical section() mechanisms, to lock and unlock access to the critical section. The current solution: our CPUs provide *atomic instructions*.
-Instructions that can never be interrupted
-Once a thread begins executing the instruction, it is guaranteed to finish it right away without the CPU doing anything else

Locks — Main Concept [pg. 77]

Without going into details, with atomic instructions it is possible to implement a lock data type.
A lock can be in one of two states taken or not taken.
There are two fundamental operations:
acquireLock(): atomically acquires (i.e., puts it in the "taken state") the lock if it's not taken, otherwise fail
releaseLock(): releases the lock (i.e., puts it in the "not taken" state)

Spinlocks [pg. 79]

The good:
A thread will enter the critical section as soon as another has left it
Very little overhead (the OS is not involved)

The bad:
If the critical section is long and a thread is already in it, a thread wanting to get it will spin for a long time. This wastes CPU cycles, power, and generates heat .

Blocking Locks [pg. 80]

The main idea:
If the lock cannot be acquired, then ask the OS to put me to sleep (to the WAITING / BLOCKED state)

Whenever the lock is released, then the OS will wake me up (to the READY state)
The good: no wasted CPU cycles if the wait it long
The bad: high-overhead (due to OS

About the duration of critical sections [pg. 83]

If a critical section is long, you should use a *blocking lock*. BUT you should make critical sections should be as short as possible
-Not in number of lines of code, but in time to run these lines
Long critical sections: only one thread can do work for a while, so we have reduced concurrent execution/parallelism, and thus reduced interactivity and/or performance
Extreme situation: put the whole code in a critical section
-Guaranteed to have no race condition, but can only use one core
Instead, one should use possibly many very short critical sections (each protected by a different lock), so that many threads can do useful work simultaneously

Locks in OSes [pg.86]

All OSes provide spinlocks and blocking locks, in one shape or another

Many provide *adaptive locks*: will spin for a short while, and then will block

Deadlocks

Defining a Dreadlock [pg.15]

We have a system with Resources and Processes

The *Resources*:

-There can be resources of types: R1, R2, . . ., Rm

-There are multiple resource of each type: e.g., 3 NICs, 4 disks

The Processes (or Threads):

P1, P2, . . ., Pn Each process can:

-Request a resource of a given type and block/wait until one resource instance of that type becomes available

-Use a resource

-Release a resource

Deadlock State [pg. 20]

A deadlock state happens if every process Pi is waiting for a resource instance that is being held by another process

Three necessary conditions for a deadlock to occur:

Mutual exclusion: At least one resource is non-shareable: at most one process at a time can use it

No preemption: Resources cannot be forcibly removed from processes that are holding them.

Circular wait: There exists a set {P0, P1, ..., Pp} of waiting processes such that ($\forall i \in \{0, 1, ...p - 1\}$) Pi is waiting for a resource held by Pi+1 and Pp is waiting for a resource held by P0.

Resource Allocation/Request Graph [pg. 29]

Describing the system can be done precisely and easily with a system resource-allocation-request graph, where . The set of vertices is made of:

The set of processes {P0, P1, ..., Pn}, and **The set of resource types {R0, R1, ..., Rm}**

Each resource instance is a black dot

The set of directed edges is made of:

Request edges where a request edge is built from a process Pi to a resource Rj if Pi has requested a resource of type Rj

Assignment edges where an assignment edge is built from an instance of a resource type Rj to a process Pi if Pi holds a resource instance of type Rj

Note: if a request can be fulfilled, the assignment edge replaces immediately the request edge

Cycles in the Graph [pg. 31]

Theorem: *If the resource-allocation-request graph contains no (directed) cycle, then there is no deadlock in the system*

-If the graph contains a cycle then there may be a deadlock

If there is only one resource instance (black dot) per resource type then we have a stronger theorem: *The existence of a cycle is a necessary and sufficient condition for the existence of a deadlock.*

Strategies and Deadlocks [pg. 56]

Prevention — Just build all programs so that at least one of the previous 3 necessary conditions can never be true, by design

Avoidance — If we are aware of the resources that the processes/threads will use, we could avoid deadlocks, more of a watchdog approach
Detection and recovery — Use algorithms to detect whether a deadlock has happened and try to recover: a let's fix it approach

Deadlock Prevention [pg. 60]

Removing necessary condition #1 (Mutual Exclusion: "At least one resource is non-shareable")
-Non-shareable resources are too useful to disallow them!
Removing necessary condition #2 (No Preemption: "Resources cannot be forcibly removed")
-But how do we even program in an environment in which an acquired resource can be taken away at any time?
Removing necessary condition #3 (Circular Wait)
-This can be done, e.g., by imposing an ordering on the resources and force processes to acquire them in that order
-FreeBSD provides an order-verifier for locks (called witness)
-Lock acquisition order is recorded, and locking locks out of order causes errors/warnings

Deadlock Avoidance [pg. 63]

The OS maintain the resource-allocation-request graph at all times
Whenever a process requests a resource, the OS determines whether giving that resource to the process would create a cycle in the graph
If it would, then reject the request, otherwise, add an edge
In a nutshell: never add an edge that would create a cycle
-Detecting a cycle in a graph with n vertices is usually O(n^2)
This approach is sometimes known as a "*Graph-based Avoidance Algorithm*"

Deadlock Detection/Recovery [pg. 67]

Detection:
Use an algorithm to determine whether we're in a deadlock state
If only one resource (black dot) per resource type, easy
-Build the resource-allocation-request graph, and if it has a cycle, we have a deadlock
If more than one resource per resource type, harder
-Use the Banker's Algorithm
This takes time, so we can only do this occasionally

Recovery:
Option #1: Process termination
-Option A: Kill all deadlocked processes
-Option B: Kill one deadlocked process at a time until no deadlock
-Dangerous program behaviors are then likely
Option #2: Resource preemption
-Select a resource to be preempted
-Rollback the process that has it

Priority Inversion [pg. 75]

Solution → *Priority inheritance*: If a process requesting a resource has higher priority than the process locking the resource, the process locking the resource is temporarily given the higher priority. This is one thing that some OSes (real-time OSes in particular) implement for you!

MAIN MEMORY
Address Virtualization

Main Memory: Basics [pg. 4]

Main Memory = Memory Unit (in Von Neumann model)
-(Large) contiguous array of bytes/words, each with its own address
-Stream of addresses coming in on the *memory bus*
-Each incoming address is stored in the memory-address register of the memory unit
-Called the "Main" memory by contrast with registers, caches, which are all managed 100% by -the hardware

Processes share the main memory, therefore the OS must manage the main memory
The CPU only works with registers, but it can issue addresses that correspond to words of the main memory

Contiguous Memory Allocation – Early Systems [pg. 9]
Contiguous Memory Allocation – Multiprogramming [pg. 10]

Address Binding - Absolute Addressing [pg. 12]
The assembler transforms the assembly code into a binary executable. One approach is to use *absolute addressing* so that the binary executable contains physical addresses.
Problems of Absolute Addressing [pg. 15]
With absolute addressing a program must be loaded exactly at the same place into memory each time we run it, otherwise the addresses will be wrong.
Corollary: We cannot run multiple instances of a single program.

One solution: recompile a program each time you need to run it, but this has a problem.
Address Binding – Relative Addressing [pg. 17]
We can solve the problem of absolute addressing with a very simple idea called *relative addressing*. Assume the address space starts at some *BASE address*, and computes all addresses as an *offset* from the BASE. The code is now completely relocatable: Only the BASE needs to be determined before running it.

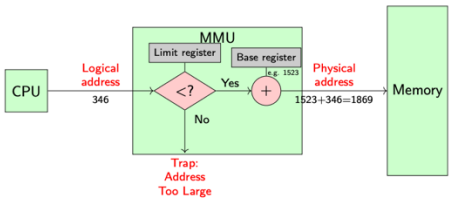
RAM Virtualization [pg. 18]
All addresses in the process address space are expressed as an offset relative to the base value.
Ex: the 4th byte in my address space is at address BASE + 4.

Memory Virtualization [pg. 24]
Thanks to virtualization: each program instance has the illusion that it's alone in RAM and that its address space starts at address 0

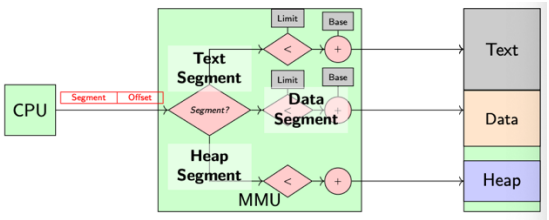
This gives us Memory Protection
Bottom Line: A program references a *logical address space*, which corresponds to a *physical address space* in the memory.

Virtualizing the process address space [pg. 29]
Some hardware component needs to translate virtual addresses into physical addresses: *add an offset to the BASE*
Address translation happens very frequently (each load, store, jump)
-Therefore: The BASE Address is accessed very frequently
-And: Offsets are added to the BASE address very frequently
Furthermore: It would be nice if only valid logical addresses were translated for memory protection. So we use a *limit register* that stores the largest possible logical address.

Memory Management Unit (MMU) [pg. 33]



Implementing Segmentation is easy. Reserve bits in the logical address to reference a segment (the *segment bits*).



Memory Unit and Segmentation [pg. 40]

Segmentation [pg. 35]

Problem: An address space is full of empty space in which the heap/stack will grow. Therefore having a single contiguous "segment" is wasteful.

Segmentation: Avoid waste by breaking up the address space into pieces, each piece has its own base/limit register.

Segment Table [pg. 38]

A *segment table* with one entry per segment number is used to keep track of segments, for each segment, its entry stores:

Base: Starting address of the segment

Limit: Length of the segment

-The segment table is stored in memory:

Segment-Table Base Register (STBR): Points to the segment table address

Segment-Table Length Register (STLR): Gives the length of the segment table

These registers are saved/restored at each context switch

Segment for Protection [pg. 39]

The segment table can include bits that answer: **readability, writability, executable.**

Any combination of 3 bits (allows the CPU to detect errors/bugs): **RWX**

RX: Read and execute (e.g. text)

RW: Read and write (e.g. stack)

Swapping

Moving processes back and forth between main memory and the disk is called *swapping*. When a process is swapped back in, it may be put into the same physical memory space or not.

Swapping and DMA [pg. 11]

With swapping, a process can be kicked out from RAM to disk by the OS at any time.

This raises a concern with *Direct Memory Access* (DMA). The DMA controller may have no idea and happily continue to write data (into some other process' address space, which has replaced that of the one that was swapped out)

One option could be: never swap a process engaged in DMA

The Bad News about Swapping [pg. 15]

The disk is slow. Several ways to cope with slow disks have been used:

-An OS could swap in/out only processes with small address space.

-One can dedicate a disk/partition to swapping (minimize disk seeks on a hard drive)

One approach is to just not swap

-Swapping should be an exceptional occurrence

-Swapping is now often

A key solution is to not swap whole address spaces ("paging", see next Module)

Where are we? [pg. 17]

We now have the mechanisms we need:

-Give each process a "slab" of memory that can fit anywhere in RAM (address virtualization)

-Swap processes in and out of memory

We now need a policy to decide how to place each slab in memory:

- Have as many processes address spaces in memory as possible
- Minimize swapping

Memory Allocation [pg. 21]

The kernel must keep a list of available memory regions or "holes". When a process arrives, before scheduling it, it is placed in an input queue.

The kernel must make decisions:

-Pick a process from the input queue

- Pick a hole in which the process will be placed, then, the process can be placed in the ready queue.

This problem is known as the *dynamic storage allocation problem*

Objective: Hold as many processes in RAM as possible

Memory Allocation Strategies [pg. 33, 36, 39]

Question 1/3: Which process should be picked?

-*First Come First Serve?* Easy, fast to compute, may delay small processes.

-*Allow smaller processes to jump ahead?* Slower to compute, favors small processes

-*Something cleverer?* Limit the "jumping ahead", Look ahead.

Question 2/3: Which hole should be picked?

First Fit? Pick the first hole that is big enough

Best Fit? Pick the smallest hole that is big enough

Worst Fit? Pick the biggest hole

Question 3/3: How should the picked process be placed in the picked hole?

Top? Bottom? Middle?

Memory Allocation [pg. 41]

FCFS + First Fit + Top?

Jump Ahead + Worst Fit + Bottom?

The above combinations are *heuristics* that hopefully produce decent solutions.

External Fragmentation [pg. 44]

Recall our objective: hold as many processes as possible in memory. What makes it difficult is *external fragmentation*. The external fragmentation is defined as the number of holes.

What about compaction? Just like defragging a hard drive, but moving processes around means a lot of slow memory copies and it creates complicated issues with I/O, DMA, etc.

Internal Fragmentation [pg. 46]

In practice, an OS would allocate slabs that are multiples of some "block size" (e.g., a number of KiB) **Downside**: a process may then not use the whole slab and some space is wasted. This is called *internal fragmentation.*

Conclusions [pg. 51]

Our **objective** was to allocate a contiguous slab of memory to each process (or to each process segment) so that their address spaces can be in RAM

Dynamic Linking and Loading

Smaller Address Space [pg. 3]

Having small address spaces is good for swapping: don't swap as often, not as slow to swap.

Three common-place techniques:

- *Dynamic Memory Allocation*
  - Ask programs to tell the OS exactly how much memory they need when they need it (malloc, new) so that we don't always allocate the maximum allowed RAM to each process.
- *Dynamic Loading*
- Dynamic Linking

Dynamic Loading [pg. 4]

Dynamic loading: only load code/text when it's needed

-Dynamic loading is enacted by code written by the programmer for this purpose. The OS is not involved, although it provides tools to make dynamic loading possible.

-*Dynamic unloading* is usually possible

Static/ Dynamic Linking [pg. 7]

-Static Linking is the historical way of reusing code

Add the assembly code of useful functions (printf...) collected in an archive or library to your own executable.

-**Issue 1**: Large text.

-**Issue 2**: Some code is (very likely) duplicated in memory.

Key idea: Why not share text (i.e., code) between processes in a similar way as data can be shared through shared memory

Dynamic Linking [pg. 10]

In spirit similar to dynamic loading, but the OS loads the code automatically and different running programs can share the code. The code is shared in shared libraries: *e.g., libc.so for Linux (so = shared object). On my Linux VM, the HelloWorld executable is 8KiB (compared to the 892KiB statically linked one)!*

Shared Libraries – How does it work [pg. 15]

When dynamic linking is enabled, the linker just puts a *stub* in the binary for each shared library routine reference.

-That *stub* is a piece of code that:

- checks whether the routine is loaded in memory
- if not, then loads it into memory "shared" (with all processes)
- then replaces itself with a simple call to the routine (it's self-modifying code!)
- future calls will be "for free"

Chances are that when you run HelloWorld, the printf code is already in memory: **⇒ Save space! ⇒ Save time!**

Shared Libraries – Easy Updates [pg. 18]

How come you can update your system (i.e., libraries) and not have to recompile all your executables?

- Provided the APIs have not changed you can just:
  - Replace a shared library (.so, .dll) by a new one
  - Ask the system to "reload" it
  - If the update was critical (i.e., security) then a reboot may be required

-Dynamic Linking requires help from the OS

- To break memory isolation and allow shared text segments among processes

Details of Shared Libraries [pg. 19]
On Linux system the **ldd** command will print the shared libraries required by a program. It turns out that, in Linux, you can override functions from loaded shared libraries by creating yourself a small shared library.

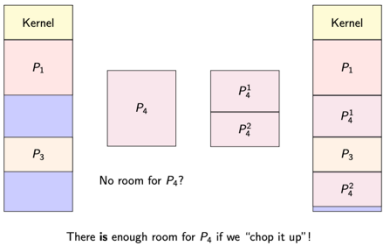
Example: C and Memory Leaks [pg. 22]
First Step: Overriding exit() [pg. 23]
Overriding malloc() and exit()! [pg. 24]

Conclusions
-Part of this is on the developer:
- Just use space-efficient data structures
- Use Dynamic Memory Allocation

-Part of this is provided by languages/compilers/OS and can be used by developers at will: *Dynamic loading and Dynamic linking*

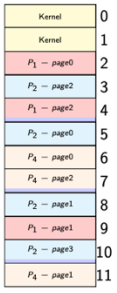
Virtual Memory – Paging I

Conclusion (of previous module)
Assumption so far: Each process is in a contiguous address space. I'll assume a single segment, for simplicity ("address space" = "segment" in these lecture notes)
Good: Address virtualization is simple (base register)
Bad: No "best" memory allocation strategies. First Fit Worst Fit, Best Fit, others??
Worse: Fragmentation can be very large RAM is wasted
Even Worse: There can be process starvation in spite of sufficient available RAM due to fragmentation 100 1MiB holes don't allow a 100MiB process to run!
Conclusion: Our base assumption is flawed!
So.... address spaces shouldn't be contiguous!?!



Paging [pg. 38]
Let's use same-size chunks, or *pages*
- The process address space is split into fixed-size pages, a policy called paging
- The physical memory is split in fixed-size frames, and **each frame can hold a page**.
- And just like that, we have non-contiguous memory allocation.
- A page is "virtual" (or "logical"): *Virtual Page Number (VPN)*
- A frame is physical: *Physical Frame Number (PFN)*

We still have internal fragmentation, but never external fragmentation!



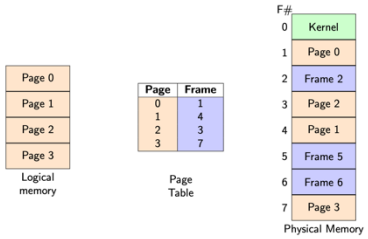
Virtual Page Number [pg. 43]
Virtual address issued by the CPU are split into two parts:
- The *virtual/logical page* number: **p**

- The *offset* within the page: **d**

The process still has the illusion of a contiguous address space starting at page 0, continuing at page 1, etc. But, each page is in a memory frame anywhere: We say *"page p is in frame f"*

Page-to-Frame Translation [pg. 45]

- The Virtual Page Number (VPN) has to be translated to the corresponding Physical Frame Number (PFN)
- This is more sophisticated address translation scheme than what we saw in the previous module for contiguous memory allocation
- We need to keep track for each process of the mapping between each of its pages and the physical frame that page is in.



-x86-64: 4 KiB, 2 MiB, and 1 GiB
-ARM: 4 KiB, 64 KiB, and 1 MiB
The page size in bytes is always a power of 2

The **pagesize** command gives you the page size on UNIX-like systems. You can easily reconfigure your OS to use a different page size, but that page size has to be supported by the hardware.

Page Size [pg. 53]

The page size is defined by the architecture

Page Size: Address Decomposition [pg. 56]

Say the size of the logical address space is $\mathbf{2^m}$ bytes. Say a page is 2^n bytes **(n < m),** then...

- The n low-order bits of a logical address are the *offset* into the page (offset ranges between 0 and $\mathbf{2^n - 1}$, each one corresponding to a byte in the page)
- The remaining **m – n** high-order bits are the *logical page number*

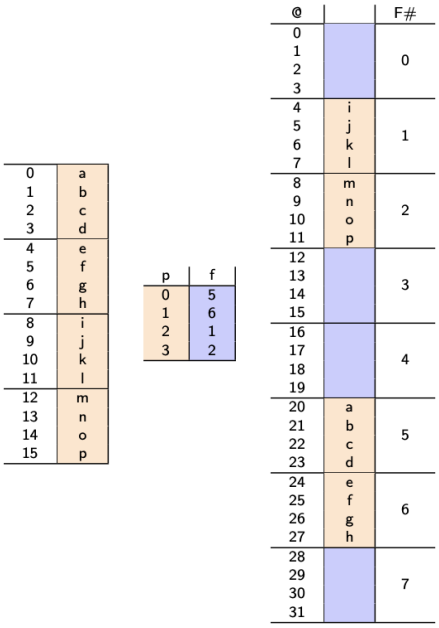
Example [pg. 63]:

Physical memory size = 2^5 = 32 bytes
How many bits in a physical address? 5 bits
Say we pick frame size = 4 bytes, so we also pick page size = 4 bytes.

- How many 4-byte frames are there? 2^5 (bytes)/ 2^2(bytes/frame) = 23 = 8 frames in RAM
- How many pages if we have a process with a 16-byte? 16(bytes)/ 4(bytes/page) = 4 pages

Say the OS has placed Page 0 into Frame 5, Page 1 into Frame 6, Page 2 into Frame 1, and Page 3 into Frame 2. Therefore, the OS will have created a page table with 4 entries for that process.

- How many bits in a virtual address for that process?
  4 bits (because we have 2^4 bytes)
  2-bit page index
  2-bit offset in the page



Example (end) [pg. 72]

What is the logical address of g?

logical @ = (page nbr)*(page size) + offset

Page = 1; Offset = 2 (often written 1:2) = 1×4+2 = 6

What is the physical address of g?

physical @ = (frame nbr)*(frame size) + offset

Page 1 is in Frame 6, same offset: 2, therefore: 6 ×4+2 = 26

In-class Exercise (1) [pg. 77]

A computer has 4 GiB of RAM with a page size of 8KiB. Processes have 1 GiB address spaces.

- **How many bits are used for physical addresses?**
  Physical RAM: 4GiB = 2^{32} bytes ⇒ 32-bit physical addresses
- **How many bits are used for logical addresses?**
  Logical Address space: 1GiB = 2^{30} bytes ⇒ 30-bit logical addresses
- **How many bits are used for logical page numbers?**
  Page size = 2^{13} bytes
  Number of pages in logical address space: $2^{30}/2^{13} = 2^{17}$. To address 2^{17} things, we need 17 bits ⇒ 17-bit logical page numbers (and 13-bit offsets)

In-class Exercise (2) [pg. 79]

Logical addresses are 44-bit, and a process can have up to 2^{27} pages.

- **What is the page size?**

The address space can have up to 2^{44} bytes. There are up to 2^{27} pages. Therefore, a page is $2^{44}/2^{27} = 2^{17}$ bytes

In-class Exercise (3) [pg. 80]

On my computer the page size is 16 KiB, and my process' address space is 4GiB.

- **How many bits are used for the page number in a logical address?**

The address space contains 2^{32} bytes. A page is 2^{14} bytes. My address space has $2^{32}/2^{14} = 2^{18}$ pages. Therefore, we need 18 bits for the page number if a logical address (and we have 14 bits in the offset)

In-class Exercise (4) [pg. 83]

A computer has 32-bit physical addresses. The logical page number of a logical address is 14-bit. A process can have up to a 2GiB address space. Let's consider a process with currently a 1GiB address space (i.e., it can get up to another 1GiB during execution).

- **What is the page size?**
  How many bytes in 2GiB (the max address space): 2^{31}. Therefore: 31-bit logical addresses. Then, 31 - 14 = 17-bit offsets, 2^{17} bytes in a page. Thus, 128KiB pages.
- **How many entries are there in the process' page table?**
  The process has a 1GiB = 2^{30}-byte address space. Number of pages in the address space: $2^{30}/2^{17} = 2^{13}$. Therefore: there are 2^{13} entries in the page table (one entry per page)

In-class Exercise (5) [pg. 85]

Logical addresses are 40-bit, and a process can use at most 1/4 of the physical RAM.

- **How bit is the RAM?**
  With 40-bit logical addresses, an address space is at most 2^{40} bytes. So, the RAM is 4 times as big: 2^{42} bytes which is 4TiB.
- **My process has 2^{22} pages, how many bits are used for the "offset" part of logical addresses?** 22 bits are used for the page number, 40 - 22 = 18 bits are used for the offset

In-class Exercise (6) [pg. 87]

Consider a system with 4-byte pages. A process has the following entries in its page table:

logical	physical
0	4
1	5
2	30

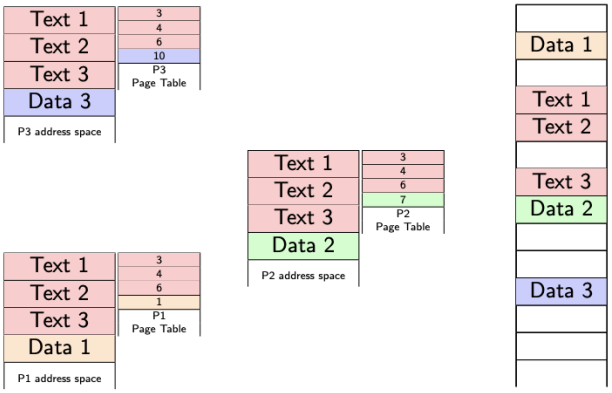
What is the physical address of the byte with logical address 9?

The byte with logical address 9 is in page 9 / 4 = 2 (integer division)
Its offset in that page is 9 % 4 = 1
Page 2 is in frame 30 Therefore, the byte at logical address 2 is at physical address 30 × 4 + 1 = 121

What is the physical address of the byte with logical address 2?

The byte with logical address 2 is the 3rd byte in page 0 (because that page contains the bytes at addresses 0, 1, 2, and 3)
Page 0, according to the page table is in physical frame 4
The first byte of physical frame 0 is at physical address 4 × 0 = 0 (the first byte in physical RAM)
The first byte of physical frame 1 is at physical address 4 × 1 = 4 (the fifth byte in physical RAM) ...
The first byte of physical frame 4 is at physical address 4 × 4 = 16
The 3rd byte of physical frame is thus at address 16 + 2. Therefore, the byte at logical address 2 is at physical address 18

Sharing Memory Pages Across Processes? EASY! [pg. 96]



Just insert page table entries that point to the same physical frames!

Valid bit [pg. 100]

- Each page entry is augmented by a valid bit.
- Set to valid if the process is allowed to access the page, set to invalid otherwise
- If the process references a page whose entry's valid bit is not set, then a trap is generated (and the process is killed)

What about Fragmentation? [pg. 106]

No external fragmentation!! This is of course the HUGE advantage of paging.
Only internal fragmentation

- Worst case: A process address space is n pages plus 1 byte In this case, we waste 1 page minus 1 byte
- Average case: Uniform distribution of address space sizes: 50% i.e., on average we waste 1/2 page per process

Using smaller pages reduces *internal fragmentation*, but large pages have advantages:

- Smaller page tables (and less frequent page table lookups)
- Loading one large page from disk takes less time than loading many small ones

Typical sizes: 4KiB, 8KiB (Linux: pagesize)

Frames Management [pg. 113]

The OS needs to keep track of the frames. The OS thus has a data structure: the *free frame list*
Much simpler than a list of holes with different sizes. When a process needs a frame, then the OS takes a frame from the free frame list and allocate them to a process.

Segmentation and Paging: e.g., IA 32/64

The Intel architecture provides both segmentation and paging
A logical/virtual address is transformed into a *linear address* via segmentation

- **logical address = (segment selector, segment offset)**

A *linear address* is transformed into a physical address via paging

- **linear address = (page number level-1, p-2, p-3, p-4, offset)**