Week 2 Notes:

- Simplified view of FDE:
    - The control and data are implemented by hardware comps
    - Multiple ALUs
        - Power consumption, limitations on cost and physics due to heat
    - Caches between CPU and Memory
    - 1+ CPUs
    - Pipelined cycle (instruction i + 1 is fetched while i is executed)
        - Makes more sense to have an instruction to be executed while the other one is being fetched
    - Improvement of processing/computer speeds have been ongoing for decades, typically leading to high hardware complexity
- Still FDE (Fetch-Decode-Execute)
- CPU ←→I/O, CPU←→ Memory (I/O will be saved for the end, however)
- Issue with RAM: it is slow, though accessing a register is fast
    - 4ghz CPU is equivalent to 1 cycle as it can update in a fourth of a nanosecond
    - Access to memory is roughly 10 ns
    - Memory is 40 times slower than CPU
    - The CPU does NOTHING while waiting for memory to give data, known as the Von Neumann bottleneck. There are many methods to mitigate this, however
        - Memory Hierarchy
    - **CPU Registers (for faster processes)←Memory Bus→ Memory (for more data)**
        - **Fast ← Slow**
        - **Small ← Large**
    - **CPU Registers←Cache→*memory buss*→ Memory←*I/O bus*→I/O Devices**
        - Numbers tend to change as time goes on.
        - Few 100 bytes, < 1 ns ,Compiler; kb to mb, 1ns, Hardware; gb, 10ns, OS; tb 1+ms OS
    - When a program access a byte in memory
        - Check whether said byte is in the cache, getting it if it is. Otherwise, it is brought from memory into cache. Values around the byte are also brought into the cache
        - Cache is useful when repeating instructions.
    - **Temporal locality**
        - A program tends to reference addresses it has already referenced such as counters
        - Fetching value takes many cycles, making the first access expensive. Subsequent accesses, however, are cheap as the value is in the cache
        - “I need that same book again”
    - **Spatial locality**
        - A program tends to references addresses next to addresses it has already referenced, such as manipulating arrays
        - The access to element i is expensive, as in temporal. As such, subsequent elements are also cheap.

        - "I need a book on that same shelf"
- Cache is managed by hardware and hardware only, while RAM and the Disk are managed by software such as the OS.
    - CPU knows nothing about what is happening in RAM (addresses are given to work with), and the OS knows nothing about whats going on in the caches (memory access is faster due to cache hits)
- May have multiple levels of caches, where L1 is the closest/fastest CPU and is split into two different caches: data and instructions
    - Tradeoffs on size, speed, cost
- Chunks tend to be brought from memory and copied/kept in caches nearby. Said data exists in multiple memories at once which can lead to issues/problems
    - Copy large chunks of data to/from RAM from/to a peripheral i.e., GPU, network card, sound card, etc.
    - CPU has to be involved in every copy operation in the Von-Neumann model
    - Problem: memory copies take a while and the CPU tends to wait as the copying operations proceeds.
- **Cache Hit (when data is found in cache) and Cache miss (not found in cache)**
    - Cache hit is a good thing EXCEPT for when it is failed data.
    - Cache hit tends to be better because it is faster
- **DMA:** Used on all modern computers, i.e. Intel i7 chipsets
    - Uses a memory bus
    - Code executed by the CPU, also using a memory bus, thus leading to interference
    - Can be managed such as providing priority to DMA or CPU, usage, etc)
    - Leads to better performance, however, and software should use it as often as possible
- Current architectures tend to use multi-core architectures, though many would prefer 100GHz single core versus 50GHz dual core
- **OPERATING SYSTEMS**
- CPU is used to run programs, like games, applications, web browsers, etc.
    - Typically limited to 32 separate processes
- OS's are resource abstractors
    - they define a set of logical resources that correspond to hardware resources
    - well defined operations run on logical resources
        - CPU ←→ Running Programs
        - Memory (DRAM, SDRAM, Future RAM) ←→ Data
        - Storage Medium (HDD/SSD/Punchcard) ←→Files
- It is also a Resource Allocator
    - It decides who and gets how much and when (after all, you run more than one program on a computer)
        - CPU ←→ Should the currently running program keep going? What runs next?
        - RAM ←→Where in RAM should a running program's data be?
        - Storage←→Where on the disk should pieces of the file be?
- **Why do we want virtualization?**
    - 1) Make the computer easier to program

        - Back then, various components needed to be learned about computers. Requires less work (without learning the heavy specifics) but with equal amount of understanding
    - 2) Provide the illusion that the program is alone on the computer
        - You may play one game and it may seem like one program is being run, but multiple programs are being run in the background.
    - Multi-programming: Execute multiple programs on a single system
        - Computers used to be single user, where a computer is alone until completion, and then the next, and the next…
        - Multiple drawbacks:
            - Computer can only do one task at a time
            - CPU is wasted if the program is idle for a while
            - If an app crashes, so does the computer most of the time. The computer would need to reboot
    - **Concurrency**: Running multiple programs at the same time
        - OS tends to run multiple things at once
        - Programs can also be concurrent (run on multiple cores)
    - **Persistence**: data stored from termination/shutdown.
- OS is software, and the core is called the Kernel (real part of the OS)
    - In charge of implementing resource abstraction and allocation, has special tasks that deal with hardware and can also be dangerous
    - Always resides in RAM and is NOT A RUNNING PROGRAM
- Things like the Unix Shell is not part of the kernel, technically
    - Originally written in assembly ONLY
    - Written in high level languages since the 60s, save for MS-DOS and similar)
    - Typically a C-related language
        - Closer to hardware and easier for developers to play tricks to make code space and time efficient
        - Compilers are good at making fast executables
        - Phased out assembly, though some sections can be written in assembly when needed
    - Concerns for kernel development: reducing memory footprint
        - Struggle on making lean and mean code/data structures and struggle on adding new features
        - Speed
        - Cannot use standard libraries

Kernel code uses non-portable compiler directives

Faster conditional with a gcc directive

```
if (__builtin_expect(n == 0, 0)) {
            return NULL;
}
```

- In kernel code you often see the above
- The __building_expect keyword is a gcc directive where you get to indicate whether the condition is typically true or false
- In the example above, the 0 second argument means "typically false"
- This is useful because then the compiler can generate faster code (by 1 or 2 cycles)
- This has to do with pipelining and branch prediction (see a Computer Architecture course/text)

- Bitwise operators tend to be used

Kernel code: bit-wise operations and macros

Bitwise operations galore, often macro-ed

```
#define MODIFY_BITS(port, mask, dir)            \
        if (mask) {                             \
                val = sa1111_readl(port);       \
                val &= ~(mask);                 \
                val |= (dir) & (mask);          \
                sa1111_writel(val, port);       \
        }

MODIFY_BITS(gpio + SA1111_GPIO_PADDR, bits & 15, dir);
MODIFY_BITS(gpio + SA1111_GPIO_PBDDR, (bits >> 8) & 255, dir >> 8);
MODIFY_BITS(gpio + SA1111_GPIO_PCDDR, (bits >> 16) & 255, dir >> 16);
```

- Many students feel that bitwise operations will never be useful
- They are much more useful/frequent than you think!
- And if you do kernel code, then they are all over the place

-
- Inline code/assembly

Kernel code: in-line assembly

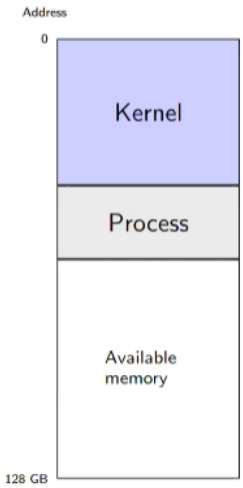
Code fragment with in-line assembly

```
while (size >= 32) {
                asm("movq    (%0), %%r8\n" "movq   8(%0), %%r9\n"
                    "movq  16(%0), %%r10\n" "movq  24(%0), %%r11\n"
                    "movnti  %%r8,   (%1)\n" "movnti  %%r9,  8(%1)\n"
                    "movnti %%r10, 16(%1)\n" "movnti %%r11, 24(%1)\n"
                    :: "r" (source), "r" (dest)
                    : "memory", "r8", "r9", "r10", "r11");
                dest += 32;
                source += 32;
                size -= 32;
        }
```

- A many points in the kernel code there is inline assembly
- These are lines of assembly code that are spliced into the C code
- These are done for speed or for doing things that would be difficult or impossible in C
- (the syntax above is x86 ATT syntax)

- Kernel debugging can be hard
    - Complex code that interacts with hardware
    - No overwatch on the kernel
    - Kernel related bugs often means a reboot may occur
- What occurs when powering on the computer?

- POST: Power On Self Tests are performed by the BIOS in firmware/ROM (Read Only Memory)
    - Booting: BIOS runs a first program, the bootstrap, running a series of instructions
    - Bootstrap then initializes computer (register contents, device controller contents, etc)
    - Loads another program in RAM and runs it, the bootstrap loader
        - Part of the kernel
    - Loaded into RAM at a fixed address. Can then run another bootstrap loader and so on, referred to as chain loading
    - Once done, nothing happens. Waits on an instruction. Take it like running on a while loop



- Each program is loaded in the RAM, often referred to as a process
    - User code: Code written by you and the library devs
    - Kernel code: Code written by the kernel devs
    - Can run user code, but also run kernel code via a system call, making a call to a function
    - Remember: Processes never step on each other in the RAM. This is known as memory protection

- Kernel is an event handler; all entries in the kernel occur as a result of an event
    - Defines handler for each type of event
    - Upon occurrence, Kernel code is run in place of the CPU FDE
- Interrupts: Asynchronous
    - Hardware
    - “incoming data on keyboard”
    - Generated in real time from outside world
- Traps: Synchronous
    - Running program
    - “Tried to divide by 0”
    - Generated as part of FDE cycle from inside

- **OS Design Goals**
    - Abstractions – making the computer more convenient
    - Performance – minimize overhead such as time and space, conflicts with abstractions
    - Protection – programs must execute in isolation
    - Reliability – OS must not fail, OS software complexity is a concern
    - Resource Efficiency – OS must make it possible to use hardware resources as best as possible
    - Comes down to a ubiquitous principle:

        - **Policy: What should be done**
        - **Mechanism: How it should be done**
        - Separation is important so that a policy can be changed without changing code
            - Mechanisms should be low level enough that many useful policies can be built on top of them. Should also be high level enough that implementing useful policies is not too labor intensive
    - Early OSes: Monolithic
        - No precisely defined structure. New features piled upon old ones (snowball effect)
        - MS-DOS was written to run in the smallest amount of space possible. This lead to poor modularity, separation of functionality, and security
    - Layered
        - First Unix had some
        - Monolithic kernel oversaw everything and difficult to maintain/evolve
            - Idea (concept 1967, Practice 1980s): Remove as much as possible from the kernel and put it all in system programs
            - Kernel only does essential management and basic IPC
            - Implemented in client-server fashion
                - Client = user
                - Server = runs system program in user space that provides some service
                - Communication through microkernel communication functionality
            - 1980s: First LANs
    - Modules
        - Take good things from all kernel design
        - Modern OSes use modules
            - Object-oriented
            - Core components are separate
            - API
        - Load at boot time/runtime when needed (loadable modules)
        - Layered-like: module has its own interface
        - Microkernel-like: module can talk to any other module, but not through IPC
            - Advantages of microkernels but with better performance
    - Aside from educational purposes, no one OS adheres to these designs. Though the best approach recommends that we do not stray too far from monolithic for the sake of performance and modularize everything to have manageable code base
- Interactive User Interfaces:
    - Batch: Historical Human Interface
    - GUI: Graphical User Interface
    - CLI: Command-Line Interface