INF1060 - Introduksjon til operativsystemer og datakommunikasjon - Kompendium

VEGARD BERGSVIK ØVSTEGÅRD

25-11-2017

Contents

1 Programmeringsspråket C

1.1 Kort om C

- Født i Palo Alto fra ca 1960
- Far: Dennis Ritchie
- Standard i system-programmering siden fødsel.
- C og Unix er uløselige siamesiske tvillinger.
- C er super rask kode.
- Gir innsikt i hvordan datamaskin og OS fungerer.
- Formålet er å gi tilgang til maskinens ressurser
- Simplifisere maskinkode
- Kompakte programmer
- Raske programmer

1.2 Fordeler:

- Kjapp kompilering
- God standardisering

1.3 Ulemper:

- Ingen standard feilhåndtering
- Ikke portabelt, må rekompileres for ulike arkitekturer.

1.4 Kompilering

C kompileres med `gcc hello.c -o hello` og kjøres med `./hello`.

1.5 C i forhold til Java

- Variabler deklareres likt.
- If-else skrives likt.
- Løkker er så og si likt
  - I for-løkker må `int i` deklareres først.
- Funksjoner er like
- Arrays er nesten like
  - C:
    ```
    int arr1[] = {1,2,3};
    ```
  - Java:

```
int[] arr1 = {1,2,3};
```

- C har ikke Objekter og klasser men strukturer:
- Lenkede lister er noe annerledes.
- Komplekse datastrukturer er ikke innebygd i C, algoritmene etc må skrives selv.

1.6 Viktige forskjeller

- Ingen boolean type i C
  - 0 er `false`
  - Alt annet er `true`
- String er ikke innebygget i C.
  - Må bruke en array av char.
- Objekter:
  - Ikke egne metoder/funksjoner
  - Ingen private verdier

1.7 String

- Er egentlig arrayer med bokstaver.
- Deklareres slik

```
char name[] = "Trunald Dump";
```

1.8 `printf`

- Deklarering:

```
int printf(const char *format, ...)
```

- %s er string og %d er integer
- Eksempel:

```
char * name = "John Smith";
int age = 27;
/* prints out 'John Smith is 27 years old.' */
printf("%s is %d years old.\n", nae, age);
```

1.9 Pekere - * og & - Dette tar tid..

- Alle variabler er en plass i minnet, og alle plasser i minnet har en adresse.
- Disse kan aksesseres ved å bruke pekeren &.

- Eksempelvis:

```c
#include <stdio.h>
int main () {
   int  var1;
   char var2[10];
   printf("Address of var1 variable: %x\n", &var1  );
   printf("Address of var2 variable: %x\n", &var2  );
   return 0;
}
```

Vil returnere:

```
Address of var1 variable: bff5a400
Address of var2 variable: bff5a3f6
```

- En peker er en variabel som har adressen til en annen variabel i minnet som verdi.
- Deklarering:

```
type *var-name;
```

- Asterisken, eller gange-tegnet, brukes for å sette en variabel som en peker.
- Kokebok:

1. Definer peker variabel.
2. Sett adressen til en variabel til pekeren
3. Til slutt, hent ut verdien tilgjengelig ved adressen i peker-variabelen.

- Eksempel:

```c
#include <stdio.h>
int main () {
   int  var = 20;   /* actual variable declaration */
   int  *ip;        /* pointer variable declaration */
   ip = &var;  /* store address of var in pointer variable*/
   printf("Address of var variable: %x\n", &var  );
   /* address stored in pointer variable */
   printf("Address stored in ip variable: %x\n", ip );
   /* access the value using the pointer */
   printf("Value of *ip variable: %d\n", *ip );
   return 0;
}
```

Gir:

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
```

1.10 Minneallokering

- malloc(*antall bytes*) lar oss allokere vilkårlig store minneområder.
  - `char* s = malloc(5000)` - Char array med lengde 5000 bytes.
- Spør først øm størrelse av input, så lengde.
- Generelt:
  - `type* s = (type*) malloc(size * sizeof(type));`
  - Eks: `int* arr = (int*) malloc(5000 * sizeof(int));`

1.11 Innlesing fra bruker

- atoi() = asci to integer
- Alloker plass til "0" byten

1.11.1 Filer

- fgets(xxx,xxx,stdin) leser fra brukeren
  - stdin er input

1.12 Innlesing fra fil

```c
#include <stdio.h>
#include <stdlib.h>
char* read_file(char *file_name) {
int read_size;
char buf[64];
FILE *file = fopen(file_name, "r");
fgets(buf, sizeof(buf), file);
read_size = atoi(buf);
char *s = (char*) malloc(read_size * sizeof(char) + 1);
fgets(s, read_size + 1, file);
return s;
}
int main(void) {
char *s = read_file("test.txt");
printf("%s\n", s);
}
```

1.13 Man-sider

- Dokumentasjon som følger med software

- C:
  - Dokumentasjon for funksjonene i C-bibliotekene
  - "standard" funksjoner
  - Prøv: `man 3 fgets`

1.14 Minnet i datamaskinen

- En rekke 0 og 1
- Adresserbar
- Som en veldig lang array eller streng
- Vi kan:
  - Skrive
  - Lese
  - Flytte(Sletter ikke den gamle verdien)

1.15 Bitoperasjoner

- Vi kan utføre bit-operasjoner på hvert enkelt byte
- Operatører:
  - NOT 0101 gir 1010 (~)
  - 1111 AND 1010 gir 1010 (&)
  - OR (|)
  - XOR (^)
  - SHIFT R (>>)
  - SHIFT L (<<)

1.16 Dobbelpekere

- Repetisjon: En peker er en variabel som peker et sted i minnet.
- Dobbelpeker er da en peker hvor det stedet det pekes på også er en peker.

```
int a = 123;
int *p = &a;
int **double_p = &p;
```

1.16.1 Hensikten

- La en funksjon endre adressen til en peker.
- Dobbel-array eksempler:
  - En liste med bokstaver: `char *ord`
  - En liste med ord: `char **setning`

1.16.2 typedef

- Eksempel:

```
typedef struct person{
int age;
} person _s;
person_s **person = malloc(..);
```

1.16.3 Heap og Stack

Dynamisk og statisk allokering

- Statisk minneallokering kan ikke endres og fungere ikke utenfor en funksjon. `char buf[100]` (STACK)
- Dynamisk minneallokering kan endres og fungerer utenfor en funksjon. `char *s = malloc(sizeof(char) * 100);` (HEAP)

Stack

- Last in first out.
- Brukes til:
  - Lokale variable
  - Funksjonskall
- Har begrenset plass.
  - "Stack overflow" kommer når man "bruker opp" stacken.

Heap

- "Uendelig" stort minneområde
- Styres av brukeren
  - Brukes til:
  - Mellomlagre variabler
  - Global tilgang
- malloc()
  - Allokerer minnet vi trenger til senere i heapen
- free()
  - Frigjør minnet allokert av brukeren
  - Minnet er begrenset
  - Hvorfor holde på en tung stein du ikke skal bruke?

1.17 Flere filer

- Husk pre-deklarering

1.18 Header-filer

- Kan inneholde felles definisjoner
  - `"header.h"` -> Ligger i samme mappe
  - `<header.h>` -> Ligger i C biblioteket
- `#include`limer inn koden hvor vi jobber
- `#ifndef __PERSON_H` -> Hvis IKKE definert
  - `#define __PERSON_H` -> Så definer

2 OS: Introduksjon

2.1 Hardware

- CPU
- Memory
- I/O devices
- Links

2.2 What is an operating system (OS)?

- Briefly told a set of codes that manage hardware to perform a set of very complicated operations, involving registers, stacks, calls and so on, to make our life simpler.
- A middle man that tells hardware what to do when commanded by different applications.
- Hardware -> OS -> Application -> User
- Hides messy details and presents a virtual machine. (top-down view)
- Resource manager. (bottom up-view)

2.3 OS Categories

- Single user, single-task
- Singe user, multi-tasking
- Multi user, multi-tasking
- Distributed OSes
- Real-time OS
- Embedded OSes

2.4 Why study OSes?

- WRITE FASTER AND MORE EFFICIENT CODE!
  - Think of a car where the OS prioritizes the wrong sensors.

- Understand trade-offs between performance and functionality, division of labor between HW and SW.

2.5 OS components and services

2.5.1 Primary components

- User interface
- **File management**
- Device management
- **Processor (prosess) management**
- **Memory management**
- Communication services

2.5.2 Device management

- Manages keyboards, mouse, disc, camera etc.
- Large diversity
- HUGE amount of code (ca 95% of the Linux code)
- Device drivers talk to to the controller and gives commands
- Each device may need different device-specific software

2.5.3 Interfaces

- System calls:
  - Applications call the system call interface witch calls the OS components.
    * Example: read(…)

2.6 Interrupts

- Electronic signals that result in a forced transfer of control to an interrupt handling routine.
  - alternative to “polling”
  - caused asynchronous events
  - IDT associates each interrupt with a code descriptor
  - can be disabled

2.7 Exceptions

- Another way for the processor to interrupt program execution.
  - **Traps**
  - **Faults**
  - **Aborts**

2.8 Booting, protection, kernel organization

- Booting:
  - read root file system and locate file with OS kernel
  - load kernel into memory
  - run kernel
- Protection
  - OS distinguish user-level and kernel-level protection
  - Applications and many sub-systems run in user mode (level 3)
  - OSes run in kernel mode (level 0)
    * Real mode
    * access to the entire memory
    * all instructions can be executed
    * bypass security
- OS organization
  - Monolithic kernels (“The big mess”):
    * a collection of functions linked into a single object
    * usually efficient
    * easy to crash
    * UNIX, Linux, Windows 7+..
  - Micro kernels
    * minimal functionality
    * other services are implemented in server processes running in user space used in a client-server model
    * lot of message passing
    * small, modular, portable
    * MACH, L4, Chrous
  - Virtualisation
    * Save energy
    * “Fail-safe”
    * Type 1 Hypervisor
    * Type 2 Hypervisor
    * Sandboxing

2.9 Summary

- Work as resource managers
- Provide different services
- Users access services using a interface like system calls

2.9.1 Why we have it and why you should care (OS)

Instead of telling the CPU, memory, GPU and all the other stuff packed in you computer, the OS makes life a lot simpler for e.g Google Chrome to browse to Facebook during a lecture. It is a middleman between hardware and software that works pretty much the same no matter what HW your machine is running. This in turn makes programs portable over different hardware and it also makes the HWs resources abstract and shareable. As a programmer one can access these resources and manipulate them to do awesome stuff. One last thing to note is that it provides a security framework in the digital world.

Having some fundamental knowledge about this colossus digital labyrint, can help save lives or lessen user aggression when constructing an application. One also learns to debug and understand software when programming in low-level languages such as C, and the trade-offs between performance and functionality and the division of labour between HW and SW.

{% capture images %} /images/os.jpg {% endcapture %} {% include gallery images=images caption="" cols=1 %}

2.9.2 What does BIOS and bootstrap do?

The **Basic Input-Output System** is a mini-program designed to prepare and start the HW so that the OS can do its magic on it. It is run of a non-volatile flash memory built in on the mother-board. It does a self-test and checks the HW when the system is powers up. Then it loads boot data from the boot sector, drags it to the system memory, executes the program on the CPU witch in turn loads the OS and runs it in a similar fashion.

2.9.3 Levels and their importance.

The OS distinguishes user-level and kernel-level protection to do just that, protect the system. The OS runs in kernel mode(level 0) which is the top, or bottom, level. This level runs in real mode(Google it), has access to the entire memory, has no security restraints and can run what ever instruction it needs. Applications and many sub-systems run in user mode(level 3) which has a much higher security restriction and a bigger safe-guard for memory f***ups.

2.9.4 Calling a system function (eg read)

When we call a system function from a user-level, the function parameters are pushed to the stack. The library code is then called and a system call number is added to the register. As the library procedure says, the kernel is called and examines the system call number. The correct system call handler is found and executes the requested operation. After the function-result is pushed from the stack to the buffer, the stack is cleaned and the process continues.

2.9.5 Monolithic vs micro kernels

The big mess of monolithic kernels are a collection of functions linked into a single object. They are usually efficient but very easy to crash hence due to their *big mess* of a complexity. Micro-kernels have very little functionality

compared, but they are very modular and portable. The goal is to have as little code as possible to run a OS. The device-drivers, file-systems and other services are run on servers.It is less likely to crash, but has a lot of message passing and this is where the bottlenecks appear.

2.9.6 What is an *interrupt*?

They are a Sub-type of exceptions and an hardware driven electronic signal that result in a forced transfer of control to an interrupt handling routine. **asynchronous** events cause them like finished disk operation, expired timers etc. Each interrupt is associated with a pointer to a code segment by the *Interrupt descriptor table (IDT)*. When an interrupt occurs, the current state is captured, the controlled is transferred and the correct interrupt handler is ordered to perform its routines. When the job is done, the previously captured stats is resumed and all is (usually) good.

3 OS: Processes & CPU Scheduling

3.1 Processes

- The "execution" of a program
  - Program is the "cook-book"
  - Process is the "cooking"
- Process table entry (process control block, PCB):
  - Data structure to store the information needed to manage a process.
  - "The manifestation of a process in an operation system"

3.1.1 Process Creation

- A process can create another process using the `pid_t fork(void)` system call
  - See `man 2 fork`
  - Process ##1 and #2 can be identical right after `for()` but can give different results after termination

3.1.2 Program Execution

- To make a process execute a program one might use `int execve(char *filename, char *params[], char envp[])` system call
  - See `man 2 execve`

3.1.3 Process Waiting

- To make a process wait for another process one can use `pid_t wait(int *status)` system call
  - See `man 2 wait`

3.1.4 Process Termination

- A process can terminate in several ways:
  - no more instructions to execute
  - a function finishes with a `return`
  - the system call `void exit(int status)`
    * Terminates a process and return status value
    * See `man 3 exit`
  - the system call `int kill(pid__t pid, int sig)`
    * Sends a signal to a process to terminate it
    * See `man 2 kill`, `man 7 signal`
- Usually, status value of `0` indicates **success** other indicates **errors**

3.1.5 Process states

- running
  - running instructions
- ready
  - ready for instructions
- blocked
  - waiting for “something/external event”

3.1.6 Context Switches

- The process of switching one running process to another
  - essential feature of multi-tasking systems
  - computationally intensive
- Possible causes:
  - interrupts
  - kernel-user mode transition
  - scheduled switch due to algorithm and time slices

3.1.7 Processes vs Threads

- Processes: resource grouping and execution {% capture images %} /images/process.JPG {% endcapture %} {% include gallery images=images caption=“” cols=1 %}
- Threads (light-weight processes)
  - efficient cooperation between execution units
  - share process resources (address space)
  - have their own state, stack, processor registers and program counter

  - no memory address switch
  - thread switching is much cheaper
  - parallell execution of concurrent tasks within a process {% capture images %} /images/thread.JPG {% endcapture %} {% include gallery images=images caption="" cols=1 %}

3.1.8 Scheduling

- A **task** is a scheduleable entity/something that can run
- Several tasks may with to use a resource simultaneously
- A scheduler decides which task can use the resource

Why do you care?

- Learning priority support makes a "yuuuge" difference
- Optimize the system to the given goals:
  - CPU utilization
  - Throughput
  - Response time
  - Fairness etc
- Prevent unnecessary waiting for e.g CPU, disk etc when they could have finished the job quicker

3.2 CPU Scheduling

3.3 FIFO and Round Robin

- FIFO:
  - Very simple but may wait forever and short jobs get behind long jobs
  - Unfair, but some are lucky
- Round Robin
  - FIFO Que
  - Each process runs a given time
    * 1/n of the CPU in max t time units per round
  - Fair, but no one is lucky
- Comparisons
  - FIFO better for ling CPU-intensive jobs
  - RR much better for interactivity #### Time Slice Size in RR
- Right time slice can improve overall utilization
- CPU bound: benefits from having longer time slices(

$$> 100ms$$

)

- I/O bound: benefits from having shorter time slices(

$$\leq 10ms$$

)

3.4 Scheduling: Goals

- Factors to consider:
  - treat similar tasks in a similar way
  - no process should wait forever
  - short response times
  - maximize throughput
  - maximum resource utilization
  - minimize overhead
  - predictable access
  - …
- Not possible to achieve all goals
- "Most reasonable" criteria depend on:
  - Kernel:
    - Resource management
      - prosessor utilization, throughput, fairness
    - User
      - Interactivity
  - Environment
    - Server vs end-system
    - Stationary vs mobile
  - Target systems
    - Most type of systems
    - Batch systems
    - Interactive systems
    - Real-time systems

3.5 Scheduling classification

- Scheduling algorithm classification:
  - dynamic
  - static
  - preemptive
  - non-preemptive

3.5.1 Preemption

- Tasks waits for processing
- Scheduler assigns priorities
- Task with highest priority will be scheduled first

3.6 Summary

3.6.1 Preemptive vs non-preemptive

Preemption is the act of temporarily interrupting a task. The running tasks are often interrupted for some time and resumed later when a more prioritized task has finished its execution. *Round Robin* algorithm is often used here. In **non-preemptive scheduling** a running task is executed till completion, they do not get interrupted. Typical algorithm for this is *FIFO*. If a user is running a program or pushing a button on a system with a non-preemptive-based scheduler, it works almost like talking to your girlfriend when she is on here phone. It might take some time, or you might not get a response at all.

3.6.2 Cooperative Scheduling/Multitasking

This is very similar to Preemptive scheduling, however task do not get suspended or interrupted. They relinquish control of the CPU at it synchronization point, or when it needs another process to do something for them. This causes less overhead than preemptive scheduling but also causes less reaction time and can some time crash a system due to an infinite loop.

3.6.3 Virtual memory > Program relocation

Virtual memory gives the illusion of having infinite memory by utilizing the HDD as a temporary storage. This allows multiple programs which demand more memory than what is available to run at the same time. Virtual addresses are pointer to data going “back and fourth” from the HDD and RAM. Programs use virtual addresses to store instructions and data; when a program is executed, the virtual addresses are converted into actual memory addresses. Instead of relocating a program due to lack of memory, the assignment to a physical address space is deferred until the program executes and the virtual memory address handler transfers data from the HDD to the RAM.

3.6.4 Simple UNIX

The scheduling algorithm in in UNIX is fairly simple. A process priority is calculated as the ratio between used CPU time(average ###ticks) and real time. Lower numbers gets higher priority. So kernel processes are prioritized hence they also get a negative priority. Without any I/O which gets high priority, the algorithm is reduced to a round-robin algorithm because no process with higher priority will interrupt the “current” cue.

3.6.5 `fork()` and `execve()`

System call fork() is used to create processes. It takes no arguments and returns a process ID. The purpose of fork() is to create a new process, which becomes the child process of the caller. After a new child process is created, both processes will execute the next instruction following the fork() system call. Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of fork(): * If fork() returns a negative value, the creation of a child process was unsuccessful. * fork() returns a zero to the newly created child process. * fork() returns a positive value, the process ID of the child process, to the parent. The returned process ID is of type pid_t defined in sys/types.h. Normally, the process ID is an integer. Moreover, a process can use function getpid() to retrieve the process ID assigned to this process.fork() returns a positive value, the process ID of the child process, to the parent. The returned process ID is of type pid_t defined in sys/types.h. Normally, the process ID is an integer. Moreover, a process can use function getpid() to retrieve the process ID assigned to this process.

`execve()` executes the program pointed to by filename. See `man 2 execve` or this video for more info. It sort of replaces one process with another. Porcess with pid "123" will run some other task afte `execve()` but the process still has pid "123".

3.6.6 `fork()` bomb with `while(1) fork()` and `if(fork() != 0) exit(0)`

Fork Bomb, a form of denial-of-service attack against a computer system, implements the fork operation (or equivalent functionality) whereby a running process can create another running process. Fork bombs count as wabbits: they typically do not spread as worms or viruses. To incapacitate a system they rely on the (generally valid) assumption that the number of programs and processes which may execute simultaneously on a computer, has a limit. They are very difficult to stop when started, because the parents exit successfully(`exit(0)`) which makes the child more difficult to fine hence it gets a new PID, and it allows the children to carry on without being auto-killed.

3.6.7 Context switching

When the preemptive processor scheduling interrupts a process, its state is stored and saved so that it can be resumed on some point. This operation is called context switching. So when the current process is interrupted or a system call is run, the state is saved into a datastructure called **process control block(PCB)**. This includes all the registers that the process is using, especially the **instruction pointer(IP)** and any other necessary instructions. When the process is successfully suspended, it **switches context** and chooses the next process from the ready-queue and restores its **PCB**. While the state of the interrupted process is saved into the PCB and the state from the next process is loaded, the CPU does not do any other work and one might call it wasted time. This is where most of the overhead is created in process scheduling.

Managing memory is: * Allocating space to precesses * Protecting memory regions * Providing "unlimited" memory via virtual memory * Control the levels of memory

4 OS: Managing memory

4.1 Hierarchies

- Memory closer to the CPU is faster but have less capacity.
- Lower levels have a copy of data in higher levels.
- Speeds:
  - cache(s)
  - main memory(ram)
  - secondary storage(disks)
  - tertiary storage(tapes)
- Hierarchies:
  - tertiary storage (tapes)
  - secondary storage (disks)
  - main memory (ram etc)
  - caches If the processor where to use 1 second on a task, the cache would use 2 seconds, main memory 1.5 minutes, secondary storage 3.5 months and tertiary storage a couple of hundreds of years.

4.2 Absolute and Relative addressing

Hardware often use **absolute addressing**. This is an absolute address and not a relative one which is the offset from another address. * The have reserved memory regions * This is FAST!! * Reads data by referencing the byte numbers in memory e.g : read absolute byte 0x000000ff

Software uses relative memory. This is independent of the process position in the memory, and the addresses are expressed *relative* to some base location. * **Dynamic address translation** is finding the absolute address during run-time and adding the *relative* and base addresses.

4.3 Processes Memory layouts

Most architectures processes partitions its available memory * A code segment * Usually read only * Can be shared * A data segment -> Grows to higher addresses and towards stack * global variables * static variables * heap etc * A stack segment -> Grows to lower addresses and towards heap * stores parameters/variables * System data segment(Process control bloc) * segment pointers * pid etc * Command line arguments, environment variables, threads etc at highest addresses

4.4 Global Memory layout

Memory is usually divided into regions * OS occupies low memory * sys-Control * resident routines * Remaining area used for transient OS routines

4.5 Multiprogrammed and memory management

- Use of secondary storage
- Swapping: remove a process from memory
  - Store state in secondary storage
- Overlays: manually replace parts of code and data
- Segmentation/paging: remove parts of a process from memory
  - usually fixed sizes stored on secondary medium

4.5.1 Fixed Partitioning

- Divide memory into static partitions
- Advantages:
  - easy implementation
  - can support swapping
- **Equal-size partitions**
  - Large programs cannot be executed
  - Small programs waste memory
- **Unequal-size partitions**
  - Large programs can me run
  - Less fragmentation
  - Waste memory if only small processes or few processes if they are large
  - can have one queue per partition

4.5.2 Dynamic Partitioning

Divides memory dynamically at run-time and they are removed after jobs finish. Fragmentation increases with system running time because memory is not re-organized. {% capture images %} /images/dynpart1.JPG {% endcapture %} {% include gallery images=images caption="" cols=1 %} **Compaction** moves the data in memory and removes the fragmentation. This takes time and consumes processing resources. **Placement algorithms** can help reduce the need compaction. * First fit - > Best * Next fit - > Not good for large segments * Best fit - > Slowest, small fragments, worst {% capture images %} /images/dynpart2.JPG {% endcapture %} {% include gallery images=images caption="" cols=1 %}

4.5.3 Buddy System

- Mix of fixed and dynamic partitioning
  - sizes of

$$2^k$$

- Maintain list of holes with sizes {% capture images %} /images/buddsys.JPG {% endcapture %} {% include gallery images=images caption="" cols=1 %}

4.5.4 Segmentation

Segments memory and addresses so that the process can fit more efficiently. Same principles as dynamic partitioning, gives no internal fragmentation and much less external fragmentation but adds a step to address translation.

4.5.5 Paging

When using paging, the OS reads data from the secondary storage in blocks called pages, all of which have identical sizes. The physical region of memory containing a single page is called a frame. This strategy does not require processes to be lined up consecutively into the memory. Like segmentation, addresses are dynamically translated during run-time. This is where the overhead and latency is created.

4.5.6 Virtual Memory

Originally made for machines with little main memory, but also used today. * Break program into smaller independent parts(Pages) * Some pages are in main memory and some on the HDD. * The MMU translates virtual addresses to physical addresses using a page table

Basics:

The process access memory via the page table. The addresses are translated to either the main or secondary storage. When the address is located at the RAM, the process handles it and goes off for the next address. If that address is translated to being on the secondary storage ei the HDD, some form of algorithm determine that som part of the main memory will be replaced with the memory waiting on the HDD. This operation is called **Memory Lookup**

4.5.7 Page fault

Page fault is an interrupt where the virtual memory address points to memory on the secondary storage and not the primary. Page fault handling is the process where the current process state is saved, the correct page is handled and replaced with another page in the main memory and at the end the control is returned to user. The magic of selecting the page to replace is done with some very smart algorithms. * Random * FIFO, NRU, LRU Second chance etc.

Most existing systems use a LRU-variant.

4.5.8 Speeding up paging

- Translation look-aside buffers is a hardware chache for the page table. It is a fixed number of slots containing the last page table entries.
- Lager page sizes reduce number of pages.
- Multi level page tables

4.6 Summary

4.6.1 Multiprocessing

Even with limited processor cores, modern computers run several thousand processes simultaneously. The handling and many strategies of multiprocessing is a very sentral aspect of OS.

4.6.2 Partitioning

Handling memory on a computer is very dependent on which partitioning-strategies one chooses to use.

4.6.3 Segmentation

To better utilize the memory and lessen fragmentation one often segment coherent blocks of data so that more memory is utilized. This also works as a security measure hence each segment has different read, write and execution permissions. It does however create some overhead by translating incoming addresses to the correct byte in the segments.

4.6.4 Fragmentation

A memory consisting of small “holes” due to lack of or poor memory management strategies, is called a fragmented memory. This can make allot of the memory useless due to some processes might need larger consecutive memory slots. Fragmentation is almost completely removed when implementing virtual memory, except for some small internal fragmentation. This is dependent of the framing of the paging method and size of the OS.

4.6.5 Paging

One of the most important concepts in memory handling and crucial for virtual memory. It allows us to put some parts of the memory out of the main storage to the secondary e.g HDD. This gives the illusion of having far more memory than we physically do and allows several large processes to run simultaneously. The big strategy discussions with paging regards the algorithms which decides which pages to replace in the main memory when a new page from the secondary storage needs to be allocated. The processor translates virtual addresses to physical ones, almost like in Segmentation, by looking up paging tables that the MMU handles. A separate hardware component called *Translation Lookaside Buffer*(TLB) is used to speed up the address translation and lookup.

4.6.6 Absolute and relative addresses

When utilising Virtual memory the computer handles relative addresses which is a pointer with an offset to the base address(The address we want).

4.6.7 LRU

Last recently used is a very good algorithm in many cases when swapping pages, however it is somewhat difficult to implement it. A linked list of all pages in memory needs to be maintained, and updated on every memory reference. Finding a page in the list, deleting it, and the moving it to the front is a very time consuming operation, even in HW.

5 OS: Storage: Disks & File Systems

5.1 (Mechanical) Disks

- Used to have a persistent Systems
- Cheaper and have more capacity than main memory
- Vastly slower

5.1.1 But we have SSDs!

- Sort of like memory
- Much faster than disks (ms vs us)
- More expensive

Large data centers and storage locations still use disks due to the cost of using SSDs. And they will continue to be used for a long while.

5.1.2 Mechanics of Disks - Video

A disk consists of: * Platters * Spindle * Tracks * Disk Heads * Sectors * Cylinders

5.1.3 Disk capacity

6 Storage space is dependent on: * # Platters * One or both sides *

7 Tracks per surface * # sectors per track * bytes per sector

7.0.1 Disk Access Time

The time between the moment issuing a disk request and the time the block is resident in memory. Consists of: * Seek time * Rotational delay * Transfer time * Other delays

Seek Time

Seek time is the time to position the head Time to move the head:

$$\alpha + \beta\sqrt{n}$$

*

$$\alpha$$

= fixed overhead *

$$\beta$$

= seek time constant *

$$n$$

= number of tracks

Rotational delay

Time for the disk platters to rotate so the first of the required sectors are under the disk head * Average delay i *1/2 revolution*

Transfer Time

Time for data to be read by the disk head, i.e., time it takes the sectors of the requested block to rotate under the head * Dependent on *data density* and *rotation speed*

Other delays

- CPU time to issue and process I/O
- contention for controller, bus, memory
- verifying block correctness with checksums (retransmissions)
- waiting in scheduling queue

7.0.2 Writing and modifying blocks

- A write operation is analogous to read operations
  - must potentially add time for block allocation
  - a complication occurs if the write operation has to be verified - must usually wait another rotation and then read the block again
  - Total write time = read time (+ time for one rotation)
- A modification operation is similar to read and write operations
  - cannot modify a block directly:
    * read block into main memory

    * modify the block
    * write new content back to disk
  - Total modify time = read time (+ time to modify) + write time

7.0.3 Disk Controller

A small processor which: * Controls the actuator moving the head * Selects head, platter and surface. * Knows when the right sector is under the head * Transfers data between main memory and disk

7.1 Data Placement

- *Interleaved* placement is fine for predictable workloads reading multiple files. But we get no gain if we have unpredictable disk accesses.
- *Non-interleaved* (or even random) placement can be used for highly unpredictable workloads
- *Contiguous* placement stores disk blocks contiguously on disk.
  - Minimal disk arm movement reading the whole file
  - no inter-operation gain if we have unpredictable disk accesses (but still not worse than random placemen)

7.2 Disk Scheduling

Seek time is the dominant factor of the total disk I/O time Several traditional algorithms: * First-Come-First-Serve (FCFS) * Shortest Seek Time First (SSTF) * SCAN (and variations) * Look (and variations)

7.2.1 Modern Disk Scheduling

- Hide their true layout
- Transparently move blocks ta spare Cylinders
- Have different zones(More data crammed on the edges)
- Head accelerates
- Prefetching with buffer caches
- low-level scheduler
- Considered “black boxes”

7.3 Summary

7.3.1 Disk

A disk is a nonvolatile storage device consisting of rotating disks with a magnetic layer. * Disks: The rotating plates with the magnetic layer. HDDs often consist of 12 disks with data stored on both sides. * Tracks are the “lines” on

the disks similar to vinyls, except they run in circles and not spirals, which contains the bits. * Head: The sensor which reads the data from the tracks. * Sector: Areas of which the tracks are divided into. Newer disks take into account the variable data-storage capability of the tracks at the ends of the platters. Larger circles gives longer tracks etc. * Cylinder: Tracks on top of tracks on the platters make cylinders.

7.3.2 Access time

The time is takes from requesting data until it is received from the disk. This time consists roughly of seek time, rotational delay, transfer time and some other delays.

7.3.3 Disk scheduling

The strategic protocols on how to administrate the mechanical parts in a disk to get the smallest possible Accesstime. There are many algorithms to use and the choice is never easy.

7.3.4 Caching

Storing copies of data temporarily to make to prohibit to many disk accesses. It is a very important tool in increasing the efficacy of a system by caching often used data closer to the process e.g main memory or the CPU cache.

8 OS: Inter-Process

8.1 Managing Mailboxes

- Mailboxes are implemented as message queues sorting messages according to FIFO
- See man:
  - msgget
  - msgsnd
  - msgrcv
  - msgctl

8.2 Pipes

- Classic IPC method under UNIX
- A method of connecting the standard output of one process to the standard input of another.
- The system call pipe(fd[2]) creates one file descriptor for reading (fd[0]) and one for writing (fd[1])

8.3 Mailboxes vs Pipes

- Msg types
  - Mailboxes may have different msg types
  - pipes do not have different types
- Buffer
  - pipes - one or more pages storing messages contiguously
  - mailboxes - linked list of messages of different types
- More than two processes
  - a pipe often (not in Linux) implies one sender and one receiver
  - many can use a mailbox

8.4 Share memory

- Shared memory is an efficient and fast way for processes to communicate
- Shared memory can best be described as the mapping of an area (segment) of memory that will be mapped and shared by more than one process.
- This is by far the fastest form of IPC.
- A segment can be created by one process, and subsequently written to and read from by any number of processes.
- System calls:
  - shmget()
  - shmat()
  - shmctl()
  - schmdt()

8.5 Signals

- Signals are software generated “interrupts” sent to a process
- Sending signals:
  - kill(pid, signal)signal system call to send any signal to pid
  - raise(signal)signal call to send signal to current process

8.5.1 Signal handling

- A signal handler can be invoked when a specific signal is received A process can deal with a signal in one of the following ways:
  - default action
  - block the signal (SIG_KILL & SIG_STOP cannot be ignored)
  - catch the signal with a handler

8.6 Summary

8.6.1 IPCs

The most used formes of IPCs on a Unix machine are mailboxes, pipes, shared memory and signals. Both mailboxes and pipes are unidirectional and based upon the message queue. How ever mailboxes can hava different message types, but pipes can not. Pipes have one or more pages which stores messages contiguously and mailboxes have linked list of messages of different types. Many processes can use a mailbox, but pipes often implies just one sender and receiver.

8.6.2 Shared memory segments

When using shared memory as an IPC channel, we have to allocate a memory segment which can be shared by all the necessary processes. This can not be done with malloc hence it allocates a block in a specific processes heap, which again is private. We have make sure that all the data we want exposed to other processes is in a shared memory segment.

8.6.3 `mmap` and `smget`

The `smget` allocates a shared memory segment and returns the identifier to that segment. What `mmap` does is it maps files or devices into the memory. This can be done private or as a shared mapping so that it is visible to other processes mapping the same region. A “good” use of `mmap` is when one uses `fork` to create child processes hence it is only one call.

9 DC: Intro to data communication

9.0.1 Internet

- Billions of interconnected devices
- Communication links such as fiber, copper etc.
- Protocols: TCP, IP, HTTP, FTP, PPP..
- Internet standards:
  - RFC: Request for comments
  - IETF: Internet Engineering Task Force

9.0.2 End systems

- Run application programs
  - Web browser, web server, email etc.
  - Edges

- Client/server model
  - Clients ask for, and get a service from the servers
- Peer-to-peer model
  - Interactions are symmetrical
  - E.g. telephone conferences

9.0.3 Protocols

Protocols define formats, order of sending and receiving of messages, and the actions that the reception initiates.

Protocol Layers

- Modularisation simplifies
  - Design
  - Maintenance
  - Updating of a system
- Explicit structure allows
  - Identification of the individual parts
  - Relations among them
- Clear structure: layering

9.0.4 TCP/IP - protocol stack

- **application**: supports network applications
  - ftp, smtp, http
  - Your applications
- **transport**: data transfer from end system to end system
  - TCP, UDP
- **network**: finding the way through the world wide network from machine to machine
  - IP
- (data) **link**: data transfer between two neighbors in the network
  - PPP (point-to-point protocol), Ethernet
- **physical**: bits on the wire

{% capture images %} /images/dc2.JPG {% endcapture %} {% include gallery images=images caption="" cols=1 %}

9.0.5 OSI - model

- Open Systems Interconnection
- Two additional layers to those of the Internet stack
- **presentation**: translates between different formats

- **session**: manages connection, control and disconnection of communication sessions

{% capture images %} /images/dc1.JPG {% endcapture %} {% include gallery images=images caption="" cols=1 %}

9.0.6 Layering: logical communication

- units implement functionality of each layer in each node
- Units execute operations, and exchange messages with other units of the same layer

9.0.7 Protocol layer and data

- Each layer takes data from next higher layer
- Adds header information to create a new data unit (message, segment, frame, packet)
- Send the new data unit to next lower layer

{% capture images %} /images/dc3.JPG {% endcapture %} {% include gallery images=images caption="" cols=1 %}

9.0.8 Core networks

10 - Graph of interconnected routers ### Circuit Switching

- Setup phase is required
- Dedicated resources
- Link bandwidth, router capacity
- Guaranteed throughput

Packet Switching

- Data streams share network resources
- Each packet uses the entire bandwidth of a link
- Resources are used as needed
- Packet switching allows more users in the net!
- Good for data with bursty behavior
- Resource sharing
- No setup phase required

Delay in packet switching networks

- Four sources of delay in each hop
- Node processing:
  - Determining the output link
    * address lookup

    - Queuing
        * Waiting for access to the output link
        * Depends on the congestion level of the router

{% capture images %} /images/dc4.JPG {% endcapture %} {% include gallery images=images caption="" cols=1 %}

- Transmission delay:
    - Time required to send a packet onto the link = L(packet size)/R(link bandwith)
- Propagation delay:
    - Propagation delay = d(physical link length)/s(propagation speed in the medium)
- traffic intensity = La(average packet arrival rate)/R

Packet switched network: Routing

- Goal: move packets from router to router between source and destination
- Datagram network:
    - Destination address determines the next hop.
    - Path can change during the sessions.
    - Routers need no information about sessions.
    - Analogy: ask for the way while you drive.
- Virtual circuit network:
    - Each packet has a tag (virtual circuit ID), which determines the next hop.
    - Path is determined when connection is set up, and remains the same for the entire session.
    - Routers need state information for each virtual circuit.

10.0.1 Network layer: IP

- Datagram switching
- Offers:
    - Addressing
    - Routing
    - Datagram service
        * Unreliable
        * Unordered
- Can use virtual circuits

10.0.2 Transport layer: TCP

- Connection-oriented service of the Internet
- Flow control
- Congestion control
- Reliable, ordered, streamoriented data transfer

10.0.3 Transport layer: UDP

- Unreliable, unordered, packetoriented data transfer
- No flow control
- No congestion control

10.1 Summary

10.1.1 Internet

Internet(capitol I) represents a specific network. It is the backbone for the world wide web and connects a lot of computer together. It helps us surf the web, send email and a whole lot of other protocols.

10.1.2 End system

The definition of an end system is a client which does not perform any tasks on behalf of the network, such as routing etc.

10.1.3 Protocols

Protocols define how we communicate between computers. We need a lot of protocols because there are several ways to communicate and use the network.

10.1.4 Protocol stack

A protocol stack is a collection of protocols in different layer, whom each on its own controls and defines certain parts of a transfer.

10.1.5 OSI-model

The OSI-model is a theoretical model used for describing how a network is built up by several layers with its own protocols. It is a useful basis when we design networks in real life.

Layers explained:

Each layer is a logical division of tasks, and each layer uses services offered by the underlying layer. The are set up so that we do not have to think about how they work. They allow a network to use several link-technologies, handle multiple protocols and used by several applications.

Application (Layer 7)

OSI Model, Layer 7, supports application and end-user processes. Communication partners are identified, quality of service is identified, user authentication and privacy are considered, and any constraints on data syntax are identified. Everything at this layer is application-specific. This layer provides application services for file transfers, e-mail, and other network software services. Telnet and FTP are applications that exist entirely in the application level. Tiered application architectures are part of this layer.

Presentation (Layer 6)

This layer provides independence from differences in data representation (e.g., encryption) by translating from application to network format, and vice versa. The presentation layer works to transform data into the form that the application layer can accept. This layer formats and encrypts data to be sent across a network, providing freedom from compatibility problems. It is sometimes called the syntax layer.

Session (Layer 5)

This layer establishes, manages and terminates connections between applications. The session layer sets up, coordinates, and terminates conversations, exchanges, and dialogues between the applications at each end. It deals with session and connection coordination.

Transport (Layer 4)

OSI Model, Layer 4, provides transparent transfer of data between end systems, or hosts, and is responsible for end-to-end error recovery and flow control. It ensures complete data transfer.

Network (Layer 3)

Layer 3 provides switching and routing technologies, creating logical paths, known as virtual circuits, for transmitting data from node to node. Routing and forwarding are functions of this layer, as well as addressing, internetworking, error handling, congestion control and packet sequencing.

Data Link (Layer 2)

At OSI Model, Layer 2, data packets are encoded and decoded into bits. It furnishes transmission protocol knowledge and management and handles errors in the physical layer, flow control and frame synchronization. The data link layer is divided into two sub layers: The Media Access Control (MAC) layer and the Logical Link Control (LLC) layer. The MAC sub layer controls how a computer on the network gains access to the data and permission to transmit it. The LLC layer controls frame synchronization, flow control and error checking.

Physical (Layer 1)

OSI Model, Layer 1 conveys the bit stream - electrical impulse, light or radio signal through the network at the electrical and mechanical level. It provides the hardware means of sending and receiving data on a carrier, including defining cables, cards and physical aspects. Fast Ethernet, RS232, and ATM are protocols with physical layer components.

10.1.6 Physical/Logical communication

Data moves physically between a lot of layer and communication links, however the application layers on each end system has a direct logical communications line. The experience a direct link and do not care about the physical communication in between.

10.1.7 Communication Media

Physical data can transfer between a lot of different communications mediums, such as satellite, radio, and cables:copper, fiber, coax, TP(CAT5 etc.)

10.1.8 Circuit switching

The communications method of Circuit switching in a network creates and connects a logical direct connection between two machines. It passes through a defined set of routers and locks down the needed resources until the connection os closed.

10.1.9 Packet switching

Packet switching attaches an address to the data being sent which allows the routers to pass it on using tables of neighbors and certain algorithms.

10.1.10 Headers

Headers refer to data placed at the beginning of a block of data by the different layers. Every time a data-package passes a layer, the corresponding header is “pealed” off or added depending og Rx vs Tx.

10.1.11 Trailers

Trailer are similar to headers, but they are added to the back of the data-packet i.e trailer. They often mark the end of the data-packet to increase performance by dropping the calculation of data length.

11 DC: Introduction to Berkeley sockets

11.1 Read & Write

- Same functions used for files etc.
- The call `read(sd, buffer, n);`
  - Reads up to n characters
  - From socket `sd`
  - Stores them in the character array `buffer`
- The call `write(sd, buffer, n);`
  - Writes up to n characters
  - From character array `buffer`
  - To the socket `s`

11.2 Alternatives to Read & Write

- The call `recv(sd, buffer, n, flags);`
  - Flags, normally just 0, but e.g., `MSG_DONTWAIT`,`MSG_MORE`,..
    - Used to control the behavior of the function
    - Several flags can be specified at once with bitwise or operaions
    - `MSG_DONTWAIT | MSG_MORE`
- The call `send(sd, buffer, n, flags);`
  - Flags, same as above

11.3 Creation of a connection

- One side must be the active one
  - Take the initiative in creating the connection
  - This side is called the **client**
- The other side must be passive
  - It is prepared for accepting connections
  - Waits for someone else to take initiative
  - This side is called the **server**

11.4 Special for the server side

- In case of TCP
  - One socket on the server side is dedicated to waiting for a connection
  - For each client that takes the initiative, a separate socket on the server side is created
  - This is useful for all servers that must be able to serve several clients concurrently (web servers, mail servers, ...)

11.4.1 Client:

```
<Necessary includes>
int main()
{
char buf[13];
    <Declare some more data structures>
    <Create a socket called sd>
    <Identify the server that you want to contact>
    <Connect to the server>

    /* Send data */
    write(sd, Hello world!, 12);

    /* Read data from the socket */
    read(sd, buf, 12);

    /* Add a string termination sign,
    and write to the screen. */
    buf[12] = \0;
    printf(%s\n, buf);

    <Closing code>
}
```

11.4.2 Server:

```
<Necessary includes>
int main()
{
    char buf[13];
    <Declare some more data structures>
    <Create a socket called request-sd>
    <Define how the client can connect>
    <Wait for a connection, and create a new socket sd
    for that connection>

    /* read data from the sd and
    write it to the screen */
    read(sd, buf, 12);
    buf[12] = \0;
```

```
    printf(%s\n, buf );

    /* send data back over the connection */
    write(sd, buf, 12);

    <Closing code>
}
```