# SYSTEMNÄRA PROGRAMMERING

## STARTAR 13:15

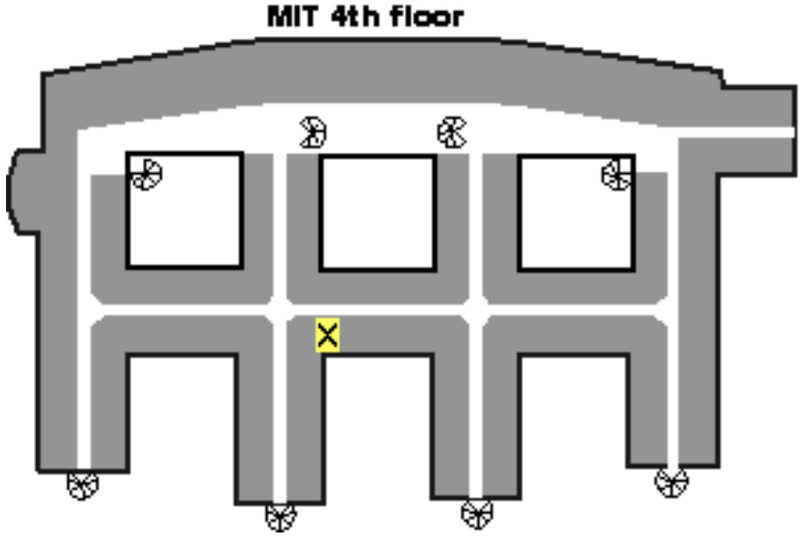
5DV088 – HT21

## LÄRARE OCH HANDLEDARE

- Föreläsningar
  - Mikael Rännar `mr@cs.umu.se`
- Gruppövningar & Handledning

  `5dv088ht21-handl@cs.umu.se`
  - Elias Häreskog
  - Oscar Kamf
  - Jonas Ernerstedt

## MIKAEL (B437)




## MIKAEL

- 40 % Institutionen för datavetenskap
  - Systemnära Programmering
  - Parallel programming for multicore-based systems
  - *Design och analys av algoritmer för parallelldatorsystem*
  - *Operativsystem*
  - *Översättarteknik*
- Ordförande i BS Nordboulen
- 60% HPC2N
  - Kurser för användare
  - *SeSE-kurser*
  - Användarsupport
  - *PRACE (EU-projekt)*
  - Koordinator

# I DAG

- Information om kursen
- Labbar
- Introduktion till C och Unix
- Lite filer och IO

# INFORMATION

- Generell Covid-information
  - https://support.cs.umu.se/covid-information/info
- Datorlab
  - https://support.cs.umu.se/covid-information/labs
- Kontaktinformation
  - https://support.cs.umu.se/covid-information/contact

# OM KURSEN

- Det är viktigt att ni kommer igång med labbarna så fort som möjligt annars kommer ni att få stora problem rent tidsmässigt.
- Tänk på att ni har andra kurser parallellt

# OM KURSEN

- Föreläsningar – Zoom
- Handledning – Tutorqueue/zoom
- Canvas, mail
  - Nästan all information ges i elektronisk form – antingen via kursens Canvas-sida eller via mail till gruppen `5DV088ht21@cs.umu.se`
  - Viktigt att läsa mail
- Kommandon – om du inte är van
  - Två länkar i Canvas till självstudier

# KURSENS CANVAS-SIDA

- Schema
- Examination
- Preliminär planering
- Filsamling
- Länkar

# OBLIGATORISKA INLÄMNINGS-UPPGIFTER ('LABBAR')

- 3 stycken labbar (länk till kursens git-repo under modulen Moment 2)
  - mexec
  - mmake
  - mdu
- Inlämningstider finns i planeringen/schemat
- Rapport – olika delar av en rapport på labbarna (se spec)
- Tänk på att labbarna ska lösas **enskilt!**
- Börja i tid !
- Inget labb är bokat specifikt för er, men jag rekommenderar unix/linux labben på i MIT-huset (labbarna ska vara gjorda för linux och det kan löna sig att sitta och jobba vid datorer med detta operativsystem)

# GRUPPÖVNINGAR / LEKTIONER

- Gruppövningar: övningar kring labbarna och kursmaterial (via Zoom)
- Preliminärt innehåll: (se planeringen på Canvas)
  - Tillfälle 1: kring lab 1
  - Tillfälle 2: kring lab 2
  - Tillfälle 3: kring lab 3
  - Tillfälle 4: …
  - Tillfälle 5: övningar inför tentamen

# LITTERATUR

- Stevens, Rago: *Advanced programming in the UNIX environment, 3 ed*
  - En slags 'illustrerad manual'
  - Inte så mycket teori om uppbyggnaden av operativsystem, men det är inte kursens huvudmål
  - Många exempel; de finns på webben också: `http://www.apuebook.com/`
- Dessutom kommer ni säkert behöva någon C-bok. Har ni ingen finns tex följande:
  - Kelley, Pohl: *A Book on C,* 4th ed
  - Hanley Koffman: *Problem Solving and Program Design in C*

# KURSPLAN

- **Innehåll:** genomgång av ett operativsystems (Unix) gränsytor och viktiga systemprogram-varor, filsystem och processhantering i Unix, introduktion till parallella processer och trådar, principer för synkronisering och kommunikation mellan processer/trådar samt programutveckling, verktyg och felsökningsmetodik i Unix-miljö.
- **Förväntade studieresultat**
  - skriva strukturerade program i programspråket C
  - använda gränsytan till ett operativsystem (Unix) för att implementera operativsystemsberoende program
  - beskriva vad en process/tråd är, hur den skapas/hanteras/avslutas i Unix-miljö
  - beskriva interna strukturer som används av operativsystemet, exempelvis ett filsystems uppbyggnad
  - redogöra för och implementera olika principer för synkronisering och kommunikation mellan processer/trådar
  - använda befintliga verktyg för programvaruutveckling i Unix-miljö

# KURSUTVÄRDERING

# EXAMINATION

- Tentamen (U/3/4/5)
  - Får ha med sig en C-bok
  - Tentamensregler
    - Se länk på Canvas
- Obligatoriska uppgifter (U/G)
- Fusk

# SYSTEMPROGRAMMERING

- Syftet med kursen är att ni skall lära er att programmera i en Unix-omgivning på en nivå som kanske är lägre än vad ni tidigare är vana vid.
- Under kursen kommer ni att utnyttja finesser i operativsystemet på ett sätt som ni inte har gjort i tidigare kurser.
- Unix är det operativsystem som vi använder, det finns hundratals andra, från gigantiska system till små realtidskärnor.

# C-PROGRAMMERING

- Förhoppningsvis kan ni redan programmera i C
- Exemplen liknar Unix:s källkod, men kanske inte är de mest läsbara
- Tidiga system anpassade till låga prestanda hos maskinvaran -> mycket 'tricks' för att spara tid, minne
- Kompilatorer numera är ganska intelligenta
- 'Okynnesoptimering' kan t.o.m. leda till att program går långsammare ...
- Välj bra algoritmer men låt kompilatorn sköta resten !
- Programmerartid är dyrare än cpu-tid !

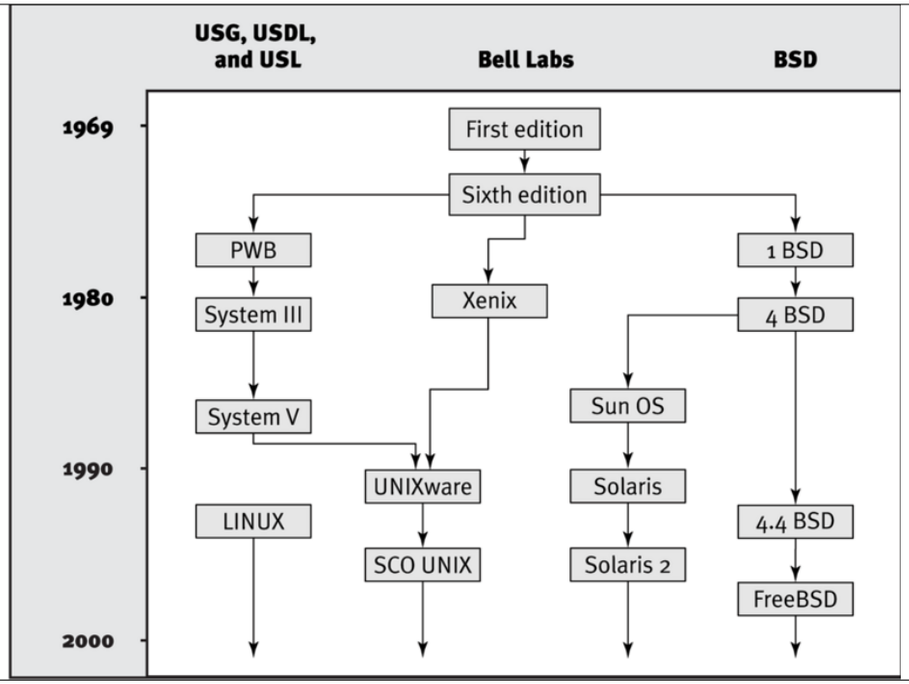
```
#define    DIT        (
#define    DAH        )
#define    __DAH      ++
#define    DITDAH     *
#define    DAHDIT     for
#define    DIT_DAH    malloc
#define    DAH_DIT    gets
#define    _DAHDIT    char
_DAHDIT _DAH_[]="ETIANMSURWDKGOHVFaLaPJBXCYZQb54a3b!2f16a/e7c8a901?e'b.k-g;i,d:"
;main                  DIT                    DAH{_DAHDIT
DITDAH                 _DIT,DITDAH            DAH_,DITDAH DIT_,
DITDAH                 _DIT_,DITDAH           DIT_DAH DIT
DAH,DITDAH             DAH_DIT DIT            DAH;DAHDIT
DIT _DIT=DIT_DAH       DIT 81                 DAH,DIT_=_DIT
__DAH;_DIT==DAH_DIT    DIT _DIT               DAH;__DIT
DIT'\n'DAH DAH         DAHDIT DIT             DAH_=_DIT;DITDAH
DAH_;__DIT             DIT                    DITDAH
_DIT_?_DAH DIT         DITDAH                 DIT_ DAH:'?'DAH,__DIT
DIT' 'DAH,DAH_ __DAH   DAH DAHDIT             DIT
DITDAH                 DIT_=2,_DIT_=_DAH_;    DITDAH _DIT_&&DIT
DITDAH _DIT_!=DIT      DITDAH DAH_>='a'?      DITDAH
DAH_&223:DITDAH        DAH_ DAH DAH;          DIT
DITDAH                 DIT_ DAH __DAH,_DIT_   __DAH DAH
DITDAH DIT_=           DIT DITDAH _DIT_>='a'? DITDAH _DIT_-'a':?
DAH;}_DAH DIT DIT_     DAH{                   __DIT DIT
DIT_>>?_DAH            DIT                    DIT_>>1 DAH:'\0'DAH;return
DIT_&1?'-':'.';}__DIT  DIT                    DIT_ DAH _DAHDIT
_DIT_;{DIT void DAH write DIT                 1,&DIT_,1 DAH;}
```

# UNIX GRUNDER, IO OCH FILER

# UNIX

- Unix är skrivet i C, ursprungligen av Dennis Ritchie, Ken Thompson och andra, på Bell Labs i början av 70-talet.
- Ursprung i Multics (Multiplexed Information and Computing Service).
- Många versioner
  - BSD - Berkeley
  - System V – AT&T
  - POSIX – IEEE/ISO standard
  - Linux – Linus Torvalds
  - ...

# HISTORIA

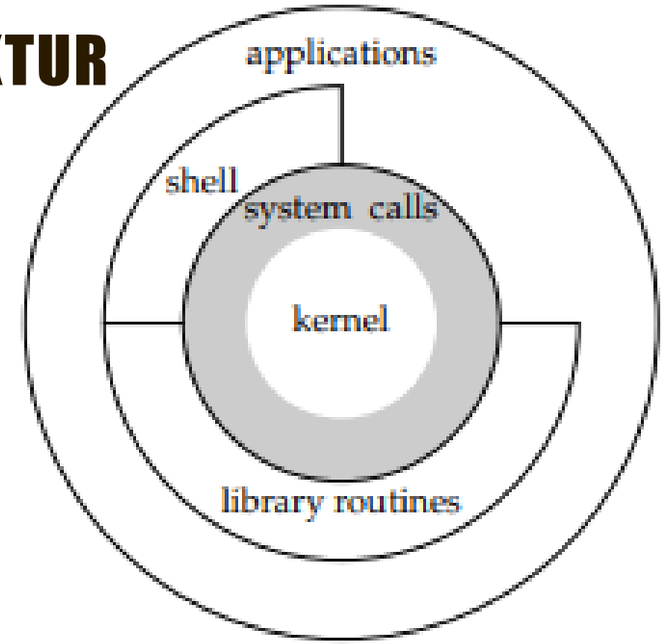



# OPERATIVSYSTEM

Många funktioner:

- Resurshantering
  - Cpu-tid
  - Primärminne
  - Lagringsutrymme
- Användargränssnitt
- Filsystem
- Säkerhet
- Nätverk
- API

# UNIX ARKITEKTUR




# VANLIGA BEGREPP

- Skalprogram – shell
- Kommandorad
- Filsystem - är som de allra flesta andra filsystem hierarkiskt i sin uppbyggnad.
- Sökväg
- Working directory
- Hemkatalog

# VANLIGA BEGREPP

- Filer
  - File pointers
    - `stdin, stdout, stderr`
  - File descriptors
    - `STDIN_FILENO, STDOUT_FILENO, STDERR_FILENO`
- Buffrad och obuffrad I/O
- Signaler

# VANLIGA BEGREPP

- Program
- Processer
- Process control
  - Från skalprogram
  - Från ett C program
- User ID
- Group ID

# VANLIGA BEGREPP

- Systemanrop (API)
- Biblioteksanrop
  - ofta med tillhörande header-fil (med definitioner och prototyper)
- Både C och Unix deklarerar ett antal olika begränsningar, tex på hur stora tal som kan representeras, hur långt ett filnamn kan vara eller hur många filer som kan vara öppna samtidigt.
  - limits.h
- Systemdatatyper

# SYSTEMANROP I C

- Ett litet C-program (hello.c):

```
#include <stdio.h>
int main(void) {
  printf("hello, world!\n");
  return 0;
}
```

- `printf` är en bibliotektsfunktion som gör ett systemanrop, kan vi se genom att använda kommandot strace

```
$ cc hello.c
$ strace ./a.out
…
write(1, ”hello, world!\n”, 14)    =       14
…
```

# SYSTEMANROP I C

- Då ser vi ju hur vi kan använda write istället

```
#include <unistd.h>
int main(void) {
  write(1, "hello, world!\n", 14);
  return 0;
}
```

- MEN detta är fortfarande en C-funktion
- write är en ”wrapper” för systemanropet och dess implementation varierar från OS till OS. (Därför fungerar det på tex både Linux och MacOS.) Undvik därför att ha funktioner/variabler med detta namn.

# SYSTEMANROP I C

- Vi kan gå lite djupare

```
#include <unistd.h>
#include <sys/syscall.h>
int main(void) {
  syscall(SYS_write, 1,"hello, world!\n", 14);
  return 0;
}
```

- `SYS_write` är en konstant som specificerar vilket systemanrop det rör sig om. Kan vara olika på olika OS, dvs nu är vi nere på OS- och arkitektur-beroende saker!

# SYSTEMANROP I C

- Vad gör då `syscall`? Skriven i assembler. Bl a läggs argumenten i rätt register. (OBS utanför kursen!) Tex:

```
.text
ENTRY (syscall)
movq %rdi, %rax          /* Syscall number -> rax.  */
movq %rsi, %rdi          /* shift arg1 - arg5.  */
movq %rdx, %rsi
movq %rcx, %rdx
movq %r8, %r10
movq %r9, %r8
movq 8(%rsp),%r9         /* arg6 is on the stack.  */
syscall                  /* Do the system call.  */
cmpq $-4095, %rax        /* Check %rax for error.  */
jae SYSCALL_ERROR_LABEL  /* Jump to error handler if error.  */
ret                      /* Return to caller.  */
PSEUDO_END (syscall)
```

# SYSTEMANROP I C

- Vi kan göra det själv med lite magisk gcc-inline assembler. Tex:

```
int main(void) {
  register int   syscall_no  asm("rax") = 1;
  register int   arg1        asm("rdi") = 1;
  register char* arg2
                 asm("rsi") = "hello, world!\n";
  register int    arg3       asm("rdx") = 14;
  asm("syscall");
  return 0;
}
```

# ERRNO OCH FELMEDDELANDEN

- Då ett systemanrop (eller systemnära funktioner) misslyckas sätts ett värde på variabeln `errno`. Detta värde kan användas för att undersöka vad som gick fel.
- `errno` är definierad i `errno.h`. Där finns även konstanter definierade för de värden `errno` kan anta.
- Funktionen
  `void perror(const char *string);`
  med flera kan användas till att generera vettiga felmeddelanden utifrån `errno`. Kolla man-sidan för denna funktion för mer info.

# FILER

- Vad är en fil?
  - Det enklaste sättet är att se på filer som en "ström" av tecken. Normalt läser man dessa ”strömmar” från ”en sida” tills den tar slut, eller också fyller man på den i slutet.
  - Unix gör ingen skillnad på om filerna innehåller text eller något annat utan det är upp till programmet att tolka informationen på önskat sätt.
  - Ofta finns ett ’magic number’ i början av filen, som en ledtråd till vilken typ av innehåll den har.
- Hur använder man en fil?
  - Principen är enkel: man öppnar den, läser det man vill veta och sedan stänger man den.

# STRÖMMAR

- Normalt ser man alltså filer som strömmar där man läser från en sida.
- Denna modell passar ju mycket bra om man tänker sig hur tangentbord och enkla terminaler fungerar. Dock har det sina begränsningar om hur man lagrar information i en databas, där är man intresserad av att hoppa till en viss plats i filen för att få fram informationen så fort som möjligt.
- Man kan betrakta tangentbordet, skärmen, andra program och till och med nätverksförbindelser som ’strömmar’.

# FILE

- I ANSI C har man definierat ett "standard IO bibliotek" som man kan använda för att slippa från en del bökiga detaljer.
  - Fördefinierad typ: `FILE`
  - En instans av denna typ innehåller all den information som vi behöver för att kunna hantera filer.
- Tre fördefinierade filer är:
  `stdin, stdout, stderr`

# BUFFRAD IO

- Man skiljer på tre olika typer av IO rutiner, helt buffrade, radbuffrade och obuffrade.
  - Buffrade rutiner innebär att man inte skriver ut allting på en gång utan man väntar tills en buffert har fyllts.
    - Anledningen till detta är att själva utmatningen blir effektivare om man har en buffert som gör att man kan skriva hela sjok på en gång och på så sätt göra det hela effektivare.
- På samma sätt gör man vid inläsning: genom att läsa in ett helt datasjok på en gång så blir man effektivare (själva hårdvaran som läser från en disk är långsammare än att läsa från internminnet)
- Vilka strömmar buffras och vilka buffras inte?
- Vad man bör tänka på?

# BIBLIOTEKSFUNKTIONER I C

- Öppna, stäng
  - `fopen`
  - `fclose`
- Ett tecken i taget
  - `getc (ett macro?)`
  - `fgetc`
  - `getchar`
  - `ferror`
  - `feof`
  - `putc (ett macro?)`
  - `fputc`
- En rad i taget
  - `fgets`
  - `(gets)`
  - `fputs`
  - `puts`
- Läsa strukturer
  - `fread`
  - `fwrite`

# BIBLIOTEKSFUNKTIONER I C

- Flytta runt i filer
  - `ftell`
  - `fseek`
  - `fgetpos`
  - `fsetpos`
- Temporära filer
  - `tmpnam`
  - `tmpfile`
- Formatterad skrivning/läsning
  - `printf`
  - `fprintf`
  - `sprintf`
  - `scanf`
  - `fscanf`
  - `sscanf`

# FILER I KERNEL API

- På operativsystemets nivå (dvs 'under' c-biblioteket) hanteras filer med hjälp av *File Decriptors*. Blanda inte filhantering direkt mot API:et med filer via biblioteket.
- File descriptors
  - Kärnan, dvs operativsystemets innersta delar, håller reda på filerna med hjälp av så kallade "file descriptors". En sådan "descriptor" får man genom att öppna eller skapa en fil. De är ett index i processens fil-tabell
  - `open`
    - Flaggor: Välj en av `O_RDONLY`, `O_WRONLY` eller `O_RDWR`
    - Sedan kan man "or'a" dem med 0 eller flera av följande konstanter
      - `O_APPEND` Append, lägg till i slutet
      - `O_CREAT` Skapa filen om det inte finns
      - `O_TRUNC` Om filen finns och öppnas så "kapas" den till 0
      - Det finns fler men dessa är de viktigaste.

# FLER SYSTEMRUTINER FÖR IO

- `creat`
  - Skapa en ny fil
- `close`
  - Stäng en öppen fil
- `lseek`
  - Hoppa till plats i fil. Motsvarighet till fseek i c-biblioteket.
- `read`
  - Läser in till en buffert
- `write`
  - Skriver en buffert till fil

# EXEMPEL

- Textfil med ca 20000 tecken:

- ```
  while ((c = fgetc(fpr)) !=EOF)
      fputc(c, fpw);
  ```
  - Tid: 0.3 ms

- ```
  while (read(fdr, &cc, sizeof(char))!=0)
      write(fdw, &cc, sizeof(char));
  ```
  - Tid: 12 ms