Java, Python, and C are all imperative languages with data types, variables, and control statements, for example. The primitive data types in C are integers (including short and long ones), characters, and floating-point numbers. Composite data types can be constructed using arrays, structures, and unions. The control statements in C are similar to those in Java, including if, switch, for, and while statements. Functions and parameters are roughly the same in both languages. One feature C has that Java and Python do not is explicit pointers. A pointer is a variable that points to (i.e., contains the address of) a variable or data structure. Consider the statements:

char c1, c2, \*p;

c1 = ’c’;

p = &c1;

c2 = \*p;

which declare c1 and c2 to be character variables and p to be a variable that points to (i.e., contains the address of) a character. The first assignment stores the ASCII code for the character ‘‘c’’ in the variable c1. The second one assigns the address of c1 to the pointer variable p. The third one assigns the contents of the variable pointed to by p to the variable c2, so after these statements are executed, c2 also contains the ASCII code for ‘‘c’’. In theory, pointers are typed, so you are not supposed to assign the address of a floating-point number to a character pointer, but in practice compilers accept such assignments, albeit sometimes with a warning. Pointers are a very powerful construct, but also a great source of errors when used carelessly. Some things that C does not have include built-in strings, threads, packages, classes, objects, type safety, and garbage collection. The last one is a show stopper for operating systems. All storage in C is either static or explicitly allocated and released by the programmer, usually with the library functions malloc and free. It is the latter property—total programmer control over memory—along with explicit pointers that makes C attractive for writing operating systems.

Good C programming practice is to name every constant except 0, 1, and −1, and sometimes even them. Macros can have parameters, such as

#define max(a, b) (a > b ? a : b)

which allows the programmer to write

i = max(j, k+1)

and get

i = (j > k+1 ? j : k+1)

to store the larger of j and k+1 in i. Headers can also contain conditional compilation, for example

#ifdef X86

intel int ack();

#endif

which compiles into a call to the function intel int ack if the macro X86 is defined and nothing otherwise. Conditional compilation is heavily used to isolate architecture-dependent code so that certain code is inserted only when the system is compiled on the X86, other code is inserted only when the system is compiled on a SPARC, and so on. A .c file can bodily include zero or more header files using the #include directive. There are also many header files that are common to nearly every .c and are stored in a central directory.

Object files, which have the suffix .o, contain binary instructions for the target machine. They will later be directly executed by the CPU. There is nothing like Java byte code or Python byte code in the C world. The first pass of the C compiler is called the C preprocessor. As it reads each .c file, every time it hits a #include directive, it goes and gets the header file named in it and processes it, expanding macros, handling conditional compilation (and certain other things) and passing the results to the next pass of the compiler as if they were physically included. Once all the .o files are ready, they are passed to a program called the linker to combine all of them into a single executable binary file. Any library functions called are also included at this point, interfunction references are resolved, and machine addresses are relocated as need be. When the linker is finished, the result is an executable program, traditionally called a.out on UNIX systems. Once the operating system binary has been linked, the computer can be rebooted and the new operating system started. At run time the operating system may consist of multiple segments, for the text (the program code), the data, and the stack. The text segment is normally immutable, not changing during execution. The data segment starts out at a certain size and initialized with certain values, but it can change and grow as need be. The stack is initially empty but grows and shrinks as functions are called and returned from. Often the text segment is placed near the bottom of memory, the data segment just above it, with the ability to grow upward, and the stack segment at a high virtual address, with the ability to grow downward, but different systems work differently. In all cases, the operating system code is directly executed by the hardware, with no interpreter and no just-in-time compilation, as is normal with Java.

Hvorfor lære: c brukes til å programmere alt -hele operativsystemer, regneark og tekstbehandlere, tungregneprogrammer, ekspertsystemer for beslutningstøtte, c-kompilatorer finnes for alle maskiner og OS. Standard C-programmer er 100% portable, godt utvalg av programvarebibliotek, presis syntaks, lett tilgang til maskinvaren når det er nødvendig, c-kompilatorer lager effektiv kode raskere enn Java bytecode. Mellomnivå programmeringsspråk: inneholder mindre ferdig kode, men har byggestein for å lage alt og gir mulighet for lavnivå tilgang til maskin(instruksjonssett). Ulemper: mindre intuitiv syntaks, krever litt kjennskap til hvordan OS og maskinvare fungere, ikke objektorientert, letter å lage bugs og vanskeligere å finne.

1972: C lages bygget dirkete på B, 1973: UNIX skrives i C første portable OS

C-kode kan skrives bare med disse tegnsett:

A-Z, a-z, 0-9 space . , : ‘ $ ” # % & ! \_ [] <> | + - / \* =

{brukes til å markere start og slutt på blokker med setninger som hører sammen, f.eks. koden som hører til en funksjon}

(brukes bl.a. til parameterlisten for funksjon)

Settinger avsluttes med ;

/\*kommentarer\*/

Alle variabler må deklareres før de brukes, de må ligge først i funksjon, variabelnavn kan ikke være lengre enn 31 tegn, kan bare bruke bokstaver, tall og underscore, må starte med bokstav/\_, kompilator gir ikke feilmelding hvis variabel brukes før initalisering, en ny variabel får bare verdien som ligger på den plassen i minnet som reservers for den(tilfeldelig), derfor er det god pratice å initialsere alle variabler når de deklarers(gi verdier)

variabel Navn = verdi;

== er sum som operator

Aritmetiske operatorer: + - \* / % ++ -- ()

ANSI C har 32 reserverte nøkkelord/variabel: auto break case char(enkelt tegn 1 byte) const continue default do double(persisjon flyttall) else enum extern float(kommatall) for goto if int(heltall) long(Dobbel lengde heltall, store tall) register return short signed sizeof(datatype størrelse) static struct switch typedef union unsigned void(uten verdi) volatile while

Størrelsen(bytes) avhenger av maskin og kompilator

C har ingen boolean datatype, int variabel brukes for å representer 0 = false

void navn1(void){printf)();}

Mange operasjoner ligger i egne bibliotek som inkluderes i koden og linkes sammen med kompilert kode til kjørbart program

preprosessor direktiver: cpp- The C Preprocessor, gjør automatisk en preprosessering av programkoden før kompileringen utføres.

alle direktiver starter med # først på en egen linje og avsluttes IKKE med ;   
vanligste er: #include headfiler <filnavn.h> (system) ”filnavn.h” (katalog) og #define

#ifndef \_NAVN\_H #define \_NAVN\_H #endif for å unngå at samme headerfil inkluderes flere ganger i sammensatt kode  
void navn(void); int main(){navn1();} gcc -o navn1 navn.c navn1.c

headerfil inneholder beskrivelser av funksjoner, typisk fra standardbibliotek i form av prototyper: hvilken parametre funksjonen har, hvilken type funksjonene returnerer   
ferdig kompilert kode for funksjoner fra bibliotekene linkes inn i programmet etter kompilering

math.h i C-programmet (se man sqrt)

-lm for å linke ferdig kompilerte matematikkfunksjoner

For at programmet skal kompilere korrekt, må det angis eksplisitt at biblioteket for matematikkfunksjoner skal lenkes sammen med programmet. Dette gjøres ved å bruke opsjonen -l til C-kompilatoren.

#include <stdio.h> standard I/O

Main(){;} – metode, globale deklarasjoner

printf(”\n”)

int printf(char \* streng, …); kan ta en eller flere parametere, første parameter er alltid strengen som skrives ut, strengen kan også inneholde formatkoder, strengen etterfølges av null eller flere parametere med verdier som skrives ut som del av strengen

printf(”%d”, variabelnavn); variabelen blir lagt inn hvor formatkoden er

<stdlib.h> rand()

Formatkode som begynner med % refererer til listen med verdier etter strengen, første % brukes for første verdi i listen osv … en bokstav etter % forteller printf hvordan verdien skal konverteres til tegn som skal skrives

Vanlige sepsifier: %c single char, %d single int, %e float or double exponential format (tier potens), %f float or double signed decimal, %s string

Formatering: modifier fw.p kan brukes samtidig eller: w-total bredde, p -antall siffer etter komma/antall tegn, f- flag et tegn som justerer plassering + skriv alltid ut fortegn på tall, - venstrejuster feltet, space skriv space, 0 legg inn nuller foran tall for å fylle feltet

Kontrollkoder: legges inn i første streng-parameter i printf, vanlige: \t horisontal tabulator \’ single quote \” double quote \\ backslash

Innlesing av data: funksjonen scanf leser fra standard innenhet, parametere: int scanf(char \* streng, …);  
første parameter beskriver en streng med data som skal leses inn, med formateringskoder, strengen etterfølges av null eller flere parametere som er adressen i primærhukommelsen til variablene som skal lagre de innleste dataene

& returnerer adressen til variabel i minnet – alle funksjoner som skal endre innholdet i en variabel må ha variabeladressen som parameter

Første parameter angir at det skal leses inn et heltall og et flyttall, skilt med space formatkodene er det samme som for printf

Kontrollstrukturer: seleksjon: if if-else switch iterasjon: while do-while for  
kan ikke deklarere løkkevariabel lokalt i toppen a for-løkker i C

Logiske tester: sammenligning av verdier: == != < > <= >=, logiske operatorer && ||/eller !/ikke ()

Eks: If ((a == b) && !(c <d)){betingelser;}

Kan teste dirkete false på verdien av variabler eller regneuttrykk i if-tester

Legge konstant først if (10.0 == a)

While (condition) {statemnet(s);}

Do{statement(s);} while(condition);

For (init; condition; increment) {statement(s);}

Break- avslutter den innerste løkken der break-setningen forekommer, eksekvering forsetter med første setning etter løkken

Continue- avslutter nåværende gjennomløp av løkken, eksekvering foresetter med løkketesten før et evt. nytt gjennomløp

Switch-setningen: switch(expression){case constant1: statement(s);break; case constant2: statement(s);break; case constant3: statement(s);break; default:statement(s);break;}

Mer kompakt skrivemåte for flergrenet if-tester, tester oftest innholdet av en variabel mot en heltalls- eller tegnkonstant, passer for tilfeller der det skal velges en verdi fra en liste med mange verdier.

kompilering i linux: cc filnavn.c, dette produserer en eksekverbar fil a.out som kjøres med: ./a.out   
for å gi navn til programet: gcc -o **navn** filnavn.c  
./**navn**

./ - stående katalog i linux

printf(”int %ld\n”, sizeof(int));

#include <stdio.h>

#include <stdlib.h>

main(){ system("date"); system("who");}

Følgende kode beregner og skriver ut summen av de første 100 positive heltallene ved å bruke en *for*-løkke:

#include <stdio.h>

int main() { int i, sum = 0; for (i = 0; i < 100; i++) sum += i; printf("Summen er %d\n", sum);}

funksjon: en samling kode som gjenbrukes, kan kalles fra andre funksjoner, kan returnere en verdi av en bestemt datatype, kan ha parameter av ulike typer  
før en funksjon kan brukes må enten: -hele funksjonene være beskrevet eller -prototype: funksjonenes navn, datatypen for verdien som returneres, parametre: antall, rekkefølge, datatyper

Følgende program bruker en funksjon for å summere to tall:

#include <stdio.h>

int beregn\_sum(int a, int b){ return (a + b);} int main(){int a, b; printf("Skriv inn et heltall: "); scanf("%d", &a); printf("Skriv inn enda et heltall: "); scanf("%d", &b); printf("Summen er %d\n", beregn\_sum(a, b));}

I eksemplet ovenfor vil funksjonen *beregn\_sum()* returnere summen av to heltall *a* og *b* som sendes som input til funksjonens argumenter *(int a, int b)*. Funksjonen kalles fra hovedprogrammet *main()* som alltid starter eksekveringen av et C-program

En tabell eller array i C er en samling av samme type dataelementer som lagres i en **sammenhengende** del av minnet. Hvert element i tabellen kan lagre én forekomst av en og samme primitive/atomære datatype (*int, long, char, double, float, ...*)

En 1-dimensjonal tabell i C med navn *A* som har plass til 100 heltall kan opprettes slik:

int A[100];

deklarasjon med initialisering:  
int A[] = {1,2,3};

Siden elementene i en tabell ligger lagret rett etter hverandre i minnet, kan du enkelt få tilgang til dem ved å bruke indekser:

* Den første indeksen i en tabell er 0 i C, ikke 1:

int first = A[0];

* Hvis lengden på en tabell er *n*, får du tilgang til det siste elementet ved å bruke indeks *(n − 1)*:

int last = A[n-1];

arraygrenser kontrolleres ikke i C, indeksering utenfor array i C er tillatt, men kan gi feil, en arrayvariabel er bare en peker til tabellens første element, og kjenner ikke sin egen lengde. For å overføre array som parameter i et funksjonskall brukes navnet til arrayen. Kan utføre regneoperasjoner på pekere, kan bruke operatorene ++ og --:  
int \*p; p++; /\*flytter peker i minnet med antall bytes som trengs for å lagre en int, til neste heltall i minnet. Typer array: statisk- lengde er konstant, endres ikke under kjøring. Dynamisk- opprettes under eksekvering av programmet, lengde kan endres. C-funksjoner for dynamiske minnehåndtering er definert i stdlib.h (man malloc)  
allokering: systemfunksjon calloc – setter plass i minnet til array, returnerer peker til første element, har to parametre: -antall elementer i arrayen og størrelsen i bytes til hvert element, det frigis med funksjonen **free** etter bruk for å unngå minnelekkase

tekststreng: det er en array der hvert element er en verdi av type char, siste tegn skal være ‘\0’(legges automatisk ved ulike tilfeller, må settes ved enkle tegn), funksjonene for håndtering av strenger vil ignorere alle tegn i arrayen son kommer etter det terminerende nulltegnet. Innlesing: bruker array med tegn til å lagre innlest streng, biblioteksfunksjoner sjekker ikke om strengen som leses er lengre enn array(overskriving), derfor vanlig å skrive egne innlesingsfunksjoner som leser ett og ett tegn  
fra standard input: funksjonen scanf med formatspesifikasjon %s – leser forbi whitespace, deretter frem til første whitespace, eller kan vi gi maks antall tegn som skal leses.  
(funksjonen gets: leser frem t.o.m. første linjeskift eller EOF, regnes som usikker å bruke)  
C-biblioteket for strenghåndtering: innebygde strengoperasjoner -ligger i **string.h**(man string), noen: strlen – lengden, strcpy strdup -kopiere, strcmp -sammenligne strcat -slå sammen

legge funksjoner som hører sammen i en egen fil-en modul som kan gjenbrukes, en modul består av en headerfil med prototypene og en kodefil med fullstendig kode for alle funksjonene, moduler kan separatkompileres og senere lenkes sammen med annen kode til fullstendige programmer, kun en av filene som utgjør et helt program kan inneholde funksjonen main

en funksjon kan returnere bare en verdi, for å returnere flere verdier/endre på variable må funksjonene kjenne minneadresse til parameter, i C brukes parametre som pekere

peker er en variabel som kan lagre en adresse i minnet, alle pekere må ha en datatype for å vite hvor mange bytes skal lagres, de lagrer minneadressen til en variabel av samme type, pekere deklareres med en \*(dereferanseoperator) foran variabelnavnet, de henter verdien til variabelen den peker på  
datatype \*peker\_navn; /\*peker til heltall\*/  
datatype \*peker\_navn2, variabelnavn /\*peker og vanlig heltall\*/  
peker kan få verdi ved bruk av adresseoperatoren &(returnerer minneadresse til variabelen): int \*p, i=0; p= &i; /\*p tilordnes adressen til i\*/  
peker kan også få verdien til pekere av samme type: int i=0; int \*p1, \*p2; p1=&i; p2=p1; /\*p1 tilordnes adressen til i, p2 tilordnes verdien til p1, begge peker til i\*/

int \*p,i=0, j; p=&i; j=\*p; det er det samme som j = i;

\* er kontekstavhengig som betyr at den har flere bruk: multiplikasjon (**må ha space mellom hva skal ganges**), opprette peker: int \*p; med en operand dereferes peker: i = \*p;

Indirekte tilordning, brukes i funksjoner som skal forandre på verdien av parametere (returnere flere enn en verdi) int i =0; int \*p = &i; \*p = 15; /\* p peker på i, indirekte tilordning i får verdien 15\*/

C tillater deklarasjon av pekere til funksjoner void skriv(int) main() {void(\*function)(int); function = skriv;(\*function)(10);} /\*det samme som skriv(10);\*/

Funksjonspeker kan gis som parameter til en annen funksjon(!): int func\_1(int(\*func\_2)(int), int x){return (\*func\_2)(x);}  
gir mulighet for å skrive polymorf kode

strømmer og filler: I/O til et program er en sekvens med data som kommer fra eller går til en ytre enhet, en strøm (stream) er kommunikasjonskanalen mellom C-program og I/O-enhetene, grensesnittet mot strømmen er uavhengig av hva slags ytre enhet det er «alt er filer»  
forhåndsdefinerte strømmer åpnes automatisk i et C-program -stdin(tastatur), stdout(skjerm) og stderr, de er definert i stdio.h brukes av printf og scanf, andre strømmer kan opprettes i programmet og kobles til ytre enheter som fil på disk  
brukerdefinerte strømmer: strømmer deklareres som pekere av typen FILE \*filepointer;  
en fil åpnes med fopen som har prototypen: FILE \*fopen(char \*filename, char \*mode); /\*mode forteller hvor filen skal brukes\*/ vanlige bruket: r- åpne tekstfil for lesning, w-lag tekstfil for skriving, a-append to a text file, rb-open binary file, wb-binary file for writing, ab-append binary file, r+/w+ åpne tekstfil for lesning/skriving  
filen lukkes med fclose etter bruk: int fclose(FILE \* filepointer);  
I/O-funksjoner fra stdio.h: int fgetc(FILE \*filepointer);-leser ett tegn fra fil og returnerer neste tegn fra en åpen tekstfil -kan leses inn i en char -returnerer EOF hvis fil slutt/feil, int fputc(int ch, FILE \* filepointer); -skriver tegnet ch til åpen fil og returnerer tegnet som er skrevet ut, char \*fgets(char \*s, int n, FILE \*fp);-leser høyst n-1 tegn stopp etter linjeskift -lagrer innleste tegn inkludert linjeskift i strengen s -returnerer peker til lest streng/NULL hvis feil, int fputs(char \*s, FILE \*fp); -skriver strengen s til fil/returnerer NULL hvis feil -tar ikke med ‘\0’ legger ikke på linjeskift,   
int fscanf(FILE \*fp, char \*ctrl\_str, …); -leser formattert, int fprintf(FILE \*fp, char \*ctrl\_str, …); -skriver formattert (formatert lesing/skriving virker på samme måte som vanlig men leser heller fra fil i stedet for fra standard input),  
fread-leser blokk med data fra binær fil, fwrite- skriver datablokk til binær fil, feof

Vi illustrerer bruk av filer med et enkelt eksempel:

En fil med navn *array.txt* inneholder 10 heltall. Filen kan f.eks. se slik ut:

8 14 22 11 3 0 53 19 103 17

Følgende kode leser innholdet i *array.txt* og setter de 10 heltallene inn i en tabell *A*:

#include <stdio.h>

#include <stdlib.h> // Bruker "exit" som er definert i stdlib

int main(){// Deklarer en fil-peker som kan referere til en fil

FILE\* fil;

int n=10, i, A[10];

int sum = 0; // Variabel for å beregne summen av tallene

// Åpner filen array.txt for lesing (parameteren "r")

fil = fopen("array.txt", "r");

// Sjekker om filåpning gikk OK

if (fil == NULL) {printf("Feil ved lesing av fil\n");// Terminerer ved feil, se "man exit" i shellet

exit(1);}

// Leser de 10 tallene

for (i = 0; i < n; i++) {fscanf(fil, "%d", &A[i]);}

// Skriver ut hele tabellen A og beregner summen av tallene i A

for (i = 0; i < n; i++) {printf("A[%d] = %d\n", i, A[i]);sum += A[i];}

// Skriver ut summen

printf("Summen av tallene i A er: %d\n", sum);

// Lukker filen og avslutter

fclose(fil); exit(0);}

Kommandolinjeargumenter: C-program kan ta argumenter fra kommandolinjen, tilgjengelige i parametrene til hovedprogrammet  
int main(int argc, char \*argv[])   
/\*argc: antall kommandolinjeargumenter, argv: array med argumentene som strenger argv[0] er programmets navn, argv[1] inneholder første argument osv…\*/

more file.txt

Strukturer i C brukes til å definere egne sammensatte datatyper (sammensatt av flere variabler som kalles strukturens medlemmer, de kan være av ulik type)

typedef struct  
{  
 <Deklarasjon av medlemmer>

} navn\_t;  
eksempel brøk: {int teller; int nevner;}brok\_t; main(){brok\_t b; b.teller=1; b.nevner=2;}

Arv finnes ikke i C.  
kopiering av innholdet i en strukturvariabel: la b1 og b2 begge være strukturvariable av samme type(brok\_t)   
b2 = b1; /\* fører til at innholdet i hvert av feltene i strukturvariablene b1 kopieres over til tilsvarende felt i strukturvariabelen b2\*/  
struktur med medlemmer av ulik type: medlemmer kan være både enkle variable og arrayer:  
#define MAX 80  
typedef struct{char etternavn[MAX]; char fornavn[MAX]; int alder; char snittkarakter;}student\_t;

Array med strukturvariable: I C kan du også lage arrayer med egendefinerte datatyper  
student\_t studenter[10]; /\*det settes av plass til 10 student-variable som legges rett etter hverandre i minnet\*/

Når et medlem i en strukturvariabel i en array skal aksesseres, plasseres operatoren **.** mellom indeks og medlemmets variabelnavn:  
int a = studenter[4]**.alder**;

Sammenligning av strukturvariable: operatorene for sammenligning av enkle variable kan ikke anvendes direkte på sammensatte datatyper, hvert enkelt medlem må sammenlignes separat for de to strukturvariablene som skal sammenlignes, derfor skriver vi egne funksjoner

Strukturer som funksjonsparametre: når vi overfører en strukturvariabel som parameter til en funksjon, kopieres hele innholdet av strukturen over i en lokal strukturvariabel i funksjonen, innholdet i hele arrayer i strukturvariabelen kopieres også, funksjonene kan ikke endre på innholdet i en vanlig strukturvariabel som brukes som parameter, for å endre verdier på medlemmene i en struktur må vi i stedet overføre en peker til strukturvariabelen.

Pekere til strukturer: for å aksessere et medlem til en strukturvariabel gjennom en peker brukes pil-operatoren ‘->’. For at en funksjon skal kunne endre verdiene på medlemmene i en struktur, overfører vi en peker til strukturvariabelen som parameter, da bruker funksjonen strukturvariabelens adresse som utgangspunkt og jobber direkte mot innholdet i strukturen, det er raskere å overføre en peker(minneadresse) enn en strukturvariabel.

main(){brokt\_t b, \*b\_p; b\_p = &b; b\_p->teller = 1; b\_p->nevner = 2;}  
endreEtternavn(&student, ‘’); int endreEtternavn(student\_t \*student, char \*nyEtternavn){ strcpy(student->etternavn, nyttEtternavn);}

Struktur som returverdi fra funksjoner: -en funksjon kan returnerer en verdi som er en variabel av selvdefinert struktur-datatype -funksjonene returnere ikke en peker til strukturvariable, men verdiene til alle medlemmene -funksjoner kan også returnere pekere til strukturvariable raskere enn å returnerer hele innholdet av strukturen

Spawning gjøres av shellet eller av en annen applikasjon, (vanligvis) ved å utføre C-systemkallet fork() i Linux-kjernen

Linux-systemkallene fork og vfork  
● Oppretter en barneprosess som er en kopi av forelder-prosessen, inkludert programkoden,  
innholdet av RAM, registre, filer, og prosessdata  
● fork: – Barneprosessen får et eget minneområde i RAM  
● vfork: – Barne- og forelderprosess deler minneområdet. Forelder er blokkert inntil barneprosess avsluttes

execvp – for å starte nye programmer  
● Systemkallet execvp erstatter nåværende prosess med en ny prosess\*  
● Hele innholdet av memory for prossesen som kaller execvp slettes, og byttes ut med et nytt program som leses inn fra disk  
● Vanlig å først gjøre en fork, og deretter la den nye barneprosessen gjøre execvp for å bytte ut seg selv med et nytt program som skal startes

\*: execvp er et av flere beslektede systemkall i Linux som kan starte nye prosesser

*execvp()* har to parametre:

int execvp(const char \*file, char \*const argv[]);

* Første parameter er en tekststreng som må inneholde navnet på **kommandoen/programmet** som skal kjøres. Hvis den eksekverbare filen som angis her ligger i en av katalogene på shellets søkesti, holder det å bare angi filnavnet. Hvis ikke må det også angis full søkesti før filnavnet.
* Andre parameter er en **array** med tekststrenger:
  + Første element i denne arrayen skal være lik strengen som er den første parameteren til *execvp()*, dvs. filnavnet for kommandoen/programmet som skal kjøres (dette er egentlig unødvendig, men er beholdt av hensyn til kompatibilitet).
  + Hver av de neste strengene i denne arrayen skal være en opsjon eller et argument til kommandoen/programmet som skal kjøres.
  + Siste streng i arrayen skal alltid være *NULL*.

*unistd*, som bl.a. inneholder funksjoner for å håndtere prosesser og hente informasjonen fra systemets prosesstabell.

Prosessen som kjører dette programmet vil først skrive ut sin egen PID som hentes med funksjonen *getpid()*, og deretter forelderprosessens PID som hentes med *getppid()*. Deretter kalles funksjonen *sleep()*, som her vil få prosessen til å "sove" (være inaktiv) i 120 sekunder før den "våkner igjen" og terminerer.

*fork()* er definert i biblioteket *unistd.h*. Datatypen *pid\_t* er definert i en headerfil *types.h* som også inkluderes i C-koden når du skal bruke *fork()*, som vist ovenfor.

Det finnes to systemkall i Linux som kan brukes for å **garantere** at en forelderprosess **venter** på at en eller flere barneprosesser er ferdige, før den selv fortsetter å eksekvere egen kode. Disse systemkallene er *wait()* og *waitpid()*.

***wait()***:

Denne funksjonen **blokkerer** forelderprosessen *P* som kaller den, helt til **en** av *P* sine barneprosesser terminerer. Hvis *P* ikke har noen barneprosesser, har kallet ingen effekt. Etter at barneprosessen avsluttes, fortsetter forelderprosessen programutførelsen rett etter kallet på *wait()*.

Funksjonen *wait()* returnerer en verdi av typen *pid\_t*, som er PID til barneprosessen som har terminert. Den har også en parameter *status* som vil inneholde returverdien (satt med et *exit()* eller *return* statement) fra barneprosessen. For å bruke *wait()* må vi inkludere en headerfil *wait.h* i koden

***waitpid():***

Hvis forelderprosessen har **flere** barneprosesser, kan vi bruke systemkallet *waitpid()* til å angi hvilken av barneprosessene det skal ventes på. *waitpid()* returnerer også PID til barneprosessen som har terminert, og har følgende tre parametere:

1. PID til barneprosessen som det skal ventes på.
2. En retur-status som vil inneholde returverdien fra barneprosessen.
3. En opsjon som kan endre virkemåten for *waitpid()*. Hvis denne settes lik 0, vil funksjonen vente bare på barneprosessen angitt i første parameter.

Du kan også sende et signal til en annen prosess fra et kjørende C-program, ved å bruke systemkallet *kill()*. *kill()* har to parametre: PID til prosessen som skal motta signalet, og signalet som skal sendes (angitt med signalnummer/navn).

Here the main program loops NUMBER OF THREADS times, creating a new thread on each iteration, after announcing its intention. If the thread creation fails, it prints an error message and then exits. After creating all the threads, the main program exits.

#include <pthread.h> #include <stdio.h> #include <stdlib.h>

#define NUMBER OF THREADS 10

void \* print hello world(void \* tid) { / \* This function prints the thread’s identifier and then exits. \*/ printf("Hello World. Greetings from thread %d\n", tid); pthread exit(NULL);}

int main(int argc, char \*argv[]) { /\* The main program creates 10 threads and then exits. \*/ pthread t threads[NUMBER OF THREADS]; int status, i;

for(i=0; i < NUMBER OF THREADS; i++) { printf("Main here. Creating thread %d\n", i); status = pthread create(&threads[i], NULL, print hello world, (void \*)i);

if (status != 0) { printf("Oops. pthread create returned error code %d\n", status); exit(-1);} } exit(NULL); }

When a thread is created, it prints a one-line message announcing itself, then it exits. The order in which the various messages are interleaved is nondeterminate and may vary on consecutive runs of the program.

pthreads – C-bibliotek for å skrive portabel kode med tråder: – Implementerer POSIX-standarden for trådprogrammering – Tilbyr mer enn 60 ulike funksjoner for trådhåndtering – C-programmer med POSIX-tråder må inkludere pthread.h og  
kompileres med opsjonen -pthread  
pthreads tilbyr funksjoner for bl.a. å:– Lage og starte nye tråder – Terminere en kjørende tråd – La en tråd vente på at en annen tråd termineres – La kjørende tråd tilby andre tråder å ta over CPU

pthreads: Opprettelse av nye tråder: En ny tråd startes i en prosess ved å kalle funksjonen pthread\_create, den nye tråden vil eksekveres parallelt med de andre trådene i prosessen. En av parametrene til pthread\_create er en peker til en funksjon som tråden vil starte sin eksevering i. Se man 3 pthread\_create for full dokumentasjon på parametre og virkemåte

pthread\_exit Terminate the calling thread  
pthread\_join Wait for a specific thread to exit  
pthread\_yield Release the CPU to let another thread run  
pthread\_attr\_init Create/initialize a thread's attribute structure  
pthread\_attr\_destroy Remove a thread's attribute structure

POSIX pthreads er fullstendig portable brukertråder som OS-kjernen ikke trenger å vite noe om, men...  
POSIX-standarden definerer egentlig bare et grensesnitt mot programmereren, selve implementasjonen av pthreads kan være svært forskjellig for ulike OS

Peterson’s algorithm is shown in Fig. 2-24. This algorithm consists of two procedures written in ANSI C, which means that function prototypes should be supplied for all the functions defined and used. However, to save space, we will not show prototypes here or later.

#define FALSE 0

#define TRUE 1

#define N 2 /\* number of processes \*/

int turn; /\* whose turn is it? \* /

int interested[N]; /\* all values initially 0 (FALSE) \*/

void enter region(int process); /\* process is 0 or 1 \* /

{

int other; /\* number of the other process \* /

other = 1 − process; /\* the opposite of process \* /

interested[process] = TRUE; /\* show that you are interested \*/

turn = process; /\* set flag \*/

while (turn == process && interested[other] == TRUE) /\* null statement \* / ;

}

void leave region(int process) /\* process: who is leaving \* /

{

interested[process] = FALSE; /\* indicate departure from critical region \* /

}

Figure 2-24. Peterson’s solution for achieving mutual exclusion.

How can this instruction be used to prevent two processes from simultaneously entering their critical regions? The solution is given in Fig. 2-25.

enter region:

TSL REGISTER,LOCK | copy lock to register and set lock to 1  
CMP REGISTER,#0 | was lock zero?  
JNE enter region | if it was not zero, lock was set, so loop  
RET | return to caller; critical region entered

leave region:

MOVE LOCK,#0 | store a 0 in lock  
RET | return to caller

Figure 2-25. Entering and leaving a critical region using the TSL instruction. An alternative instruction to TSL is XCHG, which exchanges the contents of two locations atomically, for example, a register and a memory word. The code is shown in Fig. 2-26, and, as can be seen, is essentially the same as the solution with TSL. All Intel x86 CPUs use XCHG instruction for low-level synchronization.

enter region:

MOVE REGISTER,#1 | put a 1 in the register  
XCHG REGISTER,LOCK | swap the contents of the register and lock variable  
CMP REGISTER,#0 | was lock zero?  
JNE enter region | if it was non zero, lock was set, so loop  
RET | return to caller; critical region entered

leave region:

MOVE LOCK,#0 | store a 0 in lock  
RET | return to caller

Figure 2-26. Entering and leaving a critical region using the XCHG instruction.

Semaphores solve the lost-wakeup problem, as shown in Fig. 2-28:

#define N 100 /\* number of slots in the buffer \* /

typedef int semaphore; /\* semaphores are a special kind of int \*/

semaphore mutex = 1; /\* controls access to critical region \*/

semaphore empty = N; /\* counts empty buffer slots \* /

semaphore full = 0; /\* counts full buffer slots \* /

void producer(void)

{

int item;

while (TRUE) { /\* TRUE is the constant 1 \*/

item = produce item( ); /\* generate something to put in buffer \* /

down(&empty); /\* decrement empty count \*/

down(&mutex); /\* enter critical region \*/

inser t item(item); /\* put new item in buffer \*/

up(&mutex); /\* leave critical region \*/

up(&full); /\* increment count of full slots \*/

}

}

void consumer(void)

{

int item;

while (TRUE) { /\* infinite loop \*/

down(&full); /\* decrement full count \*/

down(&mutex); /\* enter critical region \*/

item = remove item( ); /\* take item from buffer \*/

up(&mutex); /\* leave critical region \*/

up(&empty); /\* increment count of empty slots \*/

consume item(item); /\* do something with the item \*/

}

}

Figure 2-28. The producer-consumer problem using semaphores.

The semaphore operation will take only a few microseconds, whereas the producer or consumer might take arbitrarily long.

The code for mutex lock and mutex unlock for use with a user-level threads package are shown in Fig. 2-29. The solution with XCHG is essentially the same.

mutex lock:

TSL REGISTER,MUTEX | copy mutex to register and set mutex to 1

CMP REGISTER,#0 | was mutex zero?

JZE ok | if it was zero, mutex was unlocked, so return

CALL thread yield | mutex is busy; schedule another thread

JMP mutex lock | try again

ok: RET | return to caller; critical region entered

mutex unlock:

MOVE MUTEX,#0 | store a 0 in mutex

RET | return to caller

Figure 2-29. Implementation of mutex lock and mutex unlock.

Thread call Description

Pthread mutex init Create a mutex

Pthread mutex destroy Destroy an existing mutex

Pthread mutex lock Acquire a lock or block

Pthread mutex trylock Acquire a lock or fail

Pthread mutex unlock Release a lock

Figure 2-30. Some of the Pthreads calls relating to mutexes.

Thread call Description

Pthread cond init Create a condition variable

Pthread cond destroy Destroy a condition variable

Pthread cond wait Block waiting for a signal

Pthread cond signal Signal another thread and wake it up

Pthread cond broadcast Signal multiple threads and wake all of them

Figure 2-31. Some of the Pthreads calls relating to condition variables.

En såkalt "race condition" oppstår når:

* To eller flere tråder skal lese eller skrive til/fra samme delte sted i minnet eller samme fil, og:
* Det endelige resultatet, når trådene er ferdige, avhenger av i hvilken rekkefølge de kjøres.

Race conditions bør unngås fordi de kan gjøre programvaren uforutsigbar og gi feil resultater. Debugging kan også være svært vanskelig hvis feilene skyldes at trådene deler minne. Mye RAM med mange prosesser/tråder, flere CPUer/ kjerner og rask timesharing øker faren for race conditions

Et såkalt kritisk område i koden for et trådprogram, er den delen av koden hvor en race condition oppstår. Som oftest er dette kode som kan gjøre at flere tråder prøver å lese eller skrive samme del av minnet samtidig. Noe av "kunsten" med å programmere med tråder er å kunne identifisere de delene av koden som er kritiske områder, og prøve å gjøre disse små. Deretter må vi håndtere dem riktig for å unngå at det oppstår race conditions

For å unngå race conditions i trådprogrammering kan man bruke såkalt "gjensidig utelukkelse" ("mutual exclusion"):

* Kun én tråd om gangen får lov til å kjøre koden som ligger i et kritisk område.
* Hvis en tråd *A* kjører i kritisk område, må alle andre tråder som ønsker å kjøre samme kode vente til *A* er ferdig med å eksekvere den kritiske koden — trådene utelukker hverandre fra kritisk område.

Hvis de kritiske områdene i et program er store og krever mye kjøretid, vil effektiviseringen vi får av å bruke flere tråder være dårlig. Dette fordi den kritiske koden kjøres sekvensielt (en tråd om gangen) når vi bruker gjensidig utelukkelse. Trådene vil gå mye på "tomgang" (være "idle") mens de venter på å få gå inn i det kritiske området.

Det finnes ulike verktøy i forskjellige OS og programmeringsspråk for å unngå race conditions.

Trådbiblioteket *pthreads* tilbyr en egen type "låsvariabel", en såkalt mutex, for å få til "**mut**ual **ex**clusion". En mutex kan brukes til å blokkere/stoppe tråder som er på vei inn i et kritisk område, og vekke/starte dem igjen når kritisk område ikke lenger er opptatt av andre tråder. Mutexer er implementert i OS på en slik måte at det ikke kan bli en race condition på selve mutex-variabelen.

En mutex er en C-variabel av datatypen *pthread\_mutex\_t*.

Vi bør alltid initialisere mutexer før bruk. Det er en egen metode for dette i *pthreads* som heter *pthread\_mutex\_init()*. Vi kommer til å bruke funksjonen på denne måten for f.eks. å initialisere mutexen *lock*:

pthread\_mutex\_init(&lock, NULL);

Når en tråd er ferdig med å eksekvere koden i sitt kritiske område, kan den kalle funksjonen *pthread\_mutex\_unlock()* med samme mutex-variabel som parameter:

pthread\_mutex\_unlock(&lock);

En av trådene som er blokkert på denne mutexen, fordi den tidligere har kalt *pthread\_mutex\_lock()*, vil da starte opp igjen. Systemet velger selv hvilken tråd som starter hvis det er flere som er blokkert på samme mutex.

Når vi er ferdig med å bruke en mutex, kan vi fjerne den igjen ved å bruke:

pthread\_mutex\_destroy(&lock);

For å få tråder til å samarbeide om å løse problemer trenger vi mer generelle mekanismer for synkronisering. En slik mekanisme i *pthreads* er betingede variabler, som kan brukes for å sikre at en tråd venter med å kjøre videre inntil en bestemt betingelse er oppfylt.

En betinget variabel ("conditional variable") tilhører datatypen *pthread\_cond\_t* som er definert i *pthreads.h*. Opprettelse av en betinget variabel med navn *cond\_var* gjøres slik i C:

pthread\_cond\_t cond\_var;

Betingede variabler skal initialiseres før de brukes, med et kall på *pthread\_cond\_init()*:

pthread\_cond\_init(&cond\_var, NULL);

Når vi er ferdig med å bruke en betinget variabel, kan vi fjerne den igjen ved å bruke:

pthread\_cond\_destroy(&cond\_var);

En tråd kan "blokkere på en betinget variabel" (stoppe eksekvering) ved selv å kalle funksjonen *pthread\_cond\_wait()*. Tråden kan vekkes opp/startes igjen ved at en annen tråd kaller funksjonen *pthread\_cond\_signal()*.

For at en betinget variabel skal fungere sikkert og pålitelig i *pthreads*, er det viktig at variabelen alltid brukes sammen med en mutex. Mutexen brukes til å beskytte håndtering av den betingede variabelen, slik at vi ikke får en race condition.

Anta at *cond\_var* er en betinget variabel og at *lock* er en mutex. For å få en tråd til å blokkere (vente på den betingede variabelen), brukes følgende kall:

pthread\_cond\_wait(&cond\_var, &lock);

Før dette kallet skal mutexen *lock* låses med et kall på *pthread\_mutex\_lock()*.

Et kall på *pthread\_cond\_wait()* fungerer slik:

* Tråden som kaller *pthread\_cond\_wait()* blokkeres. Den vil da være stoppet inntil den "vekkes" av en annen tråd som sender et signal til den betingede variabelen *cond\_var*, med et kall på *pthread\_cond\_signal()*.
* Kallet på *pthread\_cond\_wait()* låser opp mutexen *lock* før tråden blokkeres. Mutexen låses igjen før tråden vekkes opp.

Flere tråder kan vente på samme betingede variabel, med ulike betingelser for å gå videre. Det kan hende at en tråd frigis fra en blokkering på en betinget variabel uten at betingelsen den venter på faktisk er oppfylt (se virkemåten for *pthread\_cond\_signal()* beskrevet nedenfor). Det er derfor vanlig (men ikke nødvendig) å legge kallet på *pthread\_cond\_wait()* i en løkke som sjekker om tråden faktisk kan gå videre fordi betingelsen den venter på er oppfylt.

Standard kode for å blokkere på en betinget variabel ser omtrent slik ut:

pthread\_mutex\_lock(&lock); // Lås mutex

while (!betingelse) // Så lenge tråden ikke kan gå videre:

pthread\_cond\_wait(&cond\_var, &lock); // Blokker tråden på betinget variabel

pthread\_mutex\_unlock(&lock); // Lås opp mutex igjen

Merk at denne måten å bruke en betinget variabel på henger nøye sammen med hvordan tråden vekkes opp igjen med *pthread\_cond\_signal()*.

For å vekke opp (starte igjen) en tråd som er blokkert på en betinget variabel *cond\_var*, kan en annen tråd kalle funksjonen *pthread\_cond\_signal()*:

pthread\_cond\_signal(&cond\_var);

Kallet på *pthread\_cond\_signal()* sender et "oppvåknings-signal" til den betingede variabelen. Dette fungerer slik:

* En tråd som venter på variabelen *cond\_var*, fordi den har kalt *pthread\_cond\_wait()*, vekkes opp igjen.
* Hvis ingen tråder venter på variabelen, ignoreres signalet.
* Hvis det er flere tråder som venter på samme betingede variabel, velger systemet selv hvilken av dem som vekkes opp. Vi har her ingen kontroll på hvilken tråd som faktisk vekkes opp. Dette er forklaringen på hvorfor det er vanlig å legge kallet på *pthread\_cond\_wait()* i en løkke, som vist ovenfor — det er ikke sikkert at tråden kan gå videre selv om den vekkes.

Grunnen til at betingede variable virker på denne måten, er at de vanligvis er implementert som en kø av tråder som er stoppet. Tråder som stoppes med *pthread\_cond\_wait()* legges bakerst i køen. Et kall på *pthread\_cond\_signal()* vil ta ut første tråd fra køen igjen og starte denne

unksjonen *pthread\_cond\_signal()* brukes vanligvis slik:

pthread\_mutex\_lock(&lock); // Lås mutex

pthread\_cond\_signal(&cond\_var); // Send signal til variabel

pthread\_mutex\_unlock(&lock); // Lås opp mutex igjen

Det er igjen viktig å låse koden med en mutex før kallet på *pthread\_cond\_signal()*. Hvis dette ikke gjøres, kan vi få en race condition og risikerer å miste signalet.

Race conditions og kritiske områder i koden: Det er oftest kun i noen få, begrensede deler av programkoden – kritiske områder ("critical regions") – at en prosess håndterer delt minne eller delte filer. I resten av koden gjør prosessen bare "ufarlige"  
operasjoner som ikke kan lede til race conditions. Hvis vi alltid kan unngå at to prosesser som deler ressurser er i et kritisk område av sin egen kode samtidig, vil race conditions ikke forekomme. En prosess som er på vei inn i et kritisk område må kunne stoppes midlertidig, hvis en annen prosess som har tilgang til samme delte ressurser allerede er inne i sitt kritiske område

Krav til implementasjon av mutual exclusion: 1. To prosesser kan aldri samtidig være inne i hver sin kritiske region 2. Implementasjonen må ikke bygge på antagelser om  
CPU-hastighet eller antall CPUer 3. Ingen prosesser som kjører utenfor sin egen kritiske region kan blokkere andre prosesser 4. Ingen prosesser må kunne "vente evig" på å gå inn i sin egen kritiske region

"Interrupt disable" brukes ikke i prosesshåndtering (Hvis vi har én enkelt CPU med én kjerne), men er nyttig internt i OS-kjernen ved behov for f.eks. å sikre intern konsistens ved feilsituasjoner

En del andre løsninger for å få til mutual exclusion krever at prosessene som venter på å starte i kritisk region, går i en løkke og sjekker f.eks. status på en delt låsvariabel: Dette kalles "busy waiting", låsvariabelen kalles en "spin lock". "Busy waiting" er "sløseri" med CPU-sykler, og bør bare brukes hvis forventede ventetider på en spin lock er korte. En løsning for mutual exclusion som bruker busy waiting er  
"annenhver gang"-strategien ("strict alternation").

Prosess 0 og prosess 1 deler en "spin lock" turn, **turn** angir hvilke prosess som "står for tur" til å gå inn i sin kritiske region, Prosessene får kjøre i egen kritisk region annenhver gang, Begge prosesser har hver sin funksjon som utfører hhv. (neste del av) kritisk region og ikke-kritisk region:  
void critical\_region();  
void noncritical\_region();

"Strict Alternation" er en dårlig løsning: Garanterer at vi ikke får race conditions, men…  
Problematisk hvis arbeidsmengdene i CR\* varierer mye: – P0 forlater CR, turn = 1, P1 starter CR – P1 blir raskt ferdig med CR, begge er i NCR, turn = 0 – P0 går tilbake til CR, kjører CR ferdig, turn = 1 – P0 og P1 er nå begge i NCR, turn = 1 – P0 blir igjen ferdig med NCR, men må vente fordi turn = 1 – P0 blir blokkert selv om ingen annen prosess er i CR!  
Strict alternation bryter mot krav 3 til implementasjon av mutual exclusion: Prosess 0 blokkeres av prosess 1 som ikke er i CR  
\*: CR . Criticial Region, NCR: Non-Critical Region

Petersons algoritme: Får til mutual exclusion uten å tvinge prosessene til å kjøre i kritisk region annenhver gang, Bruker samme låsvariabel turn som Strict alternation, Bruker i tillegg et boolsk flagg interested for hver prosess, som settes til true når en prosess er klar for å gå inn i kritisk region, Flagget settes til false når prosessen er ferdig i kritisk region, Virker fordi både turn og interested brukes som spin locks, Alle prosesser må kalle en funksjon enter\_region før de går inn i kritisk region, og en funksjon leave\_region etter at de er ferdige i kritisk region, for å sette låsvariablene riktig   
Egenskaper ved Petersons algoritme. Kan bevises at: – De to prosessene kan aldri være i CR samtidig – Kun prosess som er i CR kan påvirke hvilken prosess som får  
gå inn i CR neste gang – En prosess som har meldt "interesse for" å gå inn i CR vil aldri måtte vente hvis ingen andre prosesser er i CR. Kan enkelt utvides til å håndtere flere enn to prosesser

Mutual Exclusion with Busy Waiting i hardware: Kan lage enklere(?) løsning enn Peterson ved å implementere "busy while waiting" med spin lock i hardware og lavnivåkode. Slike løsninger har de samme gode egenskaper som Peterson, men trenger ikke å bruke "interested"-variablene. Utnytter i stedet at mange CPUer tilbyr såkalte "atomære" eller "udelelige" maskininstruksjoner – TSL og/eller XCHG - som kan lese/skrive til minnet og garantere at ingen andre kjerner/CPUer kan aksessere minnet samtidig\*  
Låsvariabelen som er spin lock kan derfor leses og oppdateres sikkert, uten at avbrudd kan ødelegge logikken  
\*: Gjøres ved at adressebussen låses i hardware  
Fysisk låse adressebussen

Problemer med "Busy Waiting"-løsninger: Prosessene som venter "kaster bort" CPU-tid på å sjekke endring i en variabel ved å gå i løkker. Kan gi feilsituasjoner/stopp hvis scheduleren bruker prioritering av prosessene: – Anta to prosesser: L, med lav prioritet og H, med høy prioritet – Scheduleren velger alltid å kjøre H når den er "ready"  
Systemet vil da stoppe hvis: – H blokkerer for I/O – L starter å kjøre og befinner seg nå i kritisk region – H blir ferdig med I/O, er "ready" og får ta over CPU igjen   
– Systemet er låst: H venter i løkke på L som aldri får kjøre

Mulig løsning: sleep og wakeup: I stedet for å gå i "venteløkke", blokkeres/suspenderes en prosess hvis den ikke kan gå inn i kritisk område – den  
"sover" og gjør ingenting mens den venter. Prosessen "vekkes opp" igjen når den kan fortsette å kjøre. Vi kan tenke oss to systemkall som en programmerer kan  
bruke for å få til dette:  
sleep() Får kallende prosess til å "sovne"  
wakeup(procID) "Vekker opp" en sovende prosess  
Et klassisk problem løst med sleep og wakeup: "Producer-Consumer" (aka. "Bounded-Buffer") problemet. Race condition i Producer-Consumer: Dette er egentlig det samme problemet som for spooling-eksemplet.

Semaforer\* kan løse problemet. En spesiell variabel som bl.a. kan brukes for å få til mutual exclusion, ved å telle/lagre antallet wakeup som er gjort. Verdi større enn 0 betyr at det er en eller flere wakeup som er "pending" – venter på å bli utført  
To operasjoner kan gjøres på en semafor, up og down\*\*: – down: Tilsvarende en sleep, men mer generell – up: Tilsvarende en wakeup, men mer generell  
\*: Nederlandsk(!) oppfinnelse fra 60-tallet, av en av de største navnene i informatikken, Edsger W. Dijkstra  
\*\*: Andre betegnelser som brukes er: "signal/wait", "release/acquire", "post/pend" og "vacate/procure"

Operasjonen down på en semafor: Hvis semaforen er lik 0: – Betyr at det er "ingen flere wakeups igjen" – Kallende prosess "sovner" (blokkeres) uten at down- operasjonen fullføres – Vi sier at prosessen "sover på semaforen"  
Når down fullføres, fordi semaforen var større enn 0 eller den sovende prosessen vekkes opp igjen: – Verdien av semaforen reduseres med 1, dvs. at systemet  
"bruker opp" en allerede lagret wakeup – Kallende prosess fortsetter

Operasjonen up på en semafor: Øker først verdien av semaforen med 1 – "lagrer en wakeup"  
Hvis en eller flere prosesser sover på denne semaforen: – Systemet velger én av disse prosessene, som vekkes – Prosessen kan velges tilfeldig eller på annen måte – Prosessen som vekkes gjør ferdig sin down, som vil redusere semaforen med 1, og fortsetter deretter å eksekvere – Etter en up på en semafor som prosesser sover på vil verdien fortsatt være 0, men antallet prosesser som sover på den vil være én mindre  
Et kall på up på en semafor vil aldri kunne blokkere den kallende prosessen

Implementasjon av semaforer: Både up og down på semaforer må implementeres som atomære/ udelelige operasjoner, slik at det garantert ikke er noen andre prosesser som kan ha tilgang til semaforen mens de ekseverer  
Atomære operasjoner er helt nødvendig for å unngå race conditions  
Implementeres oftest med å bruke maskininstruksjoner som TSL eller XCHG som blokkerer adressebussen når de leser/skriver RAM  
Operasjoner på en semafor er svært mye raskere enn å la en prosess henge i løkke på en spin lock

Bruk av semaforer: Kan brukes til synkronisering: – Sikre at hendelser skjer (eller ikke skjer) i en bestemt rekkefølge – "Produsenten må produsere før konsumenten kan konsumere"  
Kan også brukes for mutual exclusion i kritiske områder: – En mutex er en binær semafor som bare kan være lik 0 eller 1 – Mutex settes til 1 initielt – Hvis alle prosesser gjør en down på samme mutex før de entrer kritisk område, og en up når de forlater kritisk område, er vi garantert mutual exclusion

The Dining Philosophers Problem: flere prosesser som konkurrerer om et begrenset antall delte ressurser. Hvis ventetidene er av tilfeldig lengde vil dette oftest gå bra – men deadlock kan allikevel forekomme: Kan løses med én mutex. Bruker en mutex og en array med én semafor.

Mutexer og semaforer i pthreads: Brukertrådpakken pthreads for C-programmer tilbyr  
mekanismer både for mutual exclusion og synkronisering av de kjørende trådene - Vi kan deklarere spesielle mutex-variable, som er binære semaforer for å blokkere og vekke opp tråder som er på vei inn i et kritisk område - pthreads tilbyr også såkalte betingende ("conditional") variable som kan brukes til å synkronisere tråder, slik at de  
kan vente på hverandre inntil f.eks. en betingelse er oppfylt

● En mutex er en C-variabel av datatypen pthread\_mutex\_t:  
pthread\_mutex\_t lock;  
● Initialisering av mutexer før bruk:  
pthread\_mutex\_init(&lock, NULL);  
● Låsing av mutex, blokkerer kallende tråd hvis allerede låst:  
pthread\_mutex\_lock(&lock);  
● Låse opp mutex, én av trådene som er blokkert på denne  
mutexen frigis og får kjøre igjen:  
pthread\_mutex\_unlock(&lock);

Free mutex: destroy

Betingede variable i pthreads:  
● En betinget/condition variabel er av datatypen pthread\_cond\_t:  
pthread\_cond\_t cond\_var;  
● Initialisering av betinget variabel før bruk:  
pthread\_cond\_init(&cond\_var, NULL);  
● For å blokkere en tråd med en betinget variabel brukes:  
pthread\_cond\_wait →  
● For å vekke opp en tråd som venter på en betinget variabel:  
pthread\_cond\_signal →  
● Alle wait- og signal-operasjoner på betingede variable er atomære

Blokkere tråd med betinget variabel: En betinget variabel brukes alltid sammen med en mutex: – Tråden låser på en mutex, oftest før start på en kritisk region – Venter/blokkerer deretter på en betinget variabel inntil tråden "får det den venter på" fra en annen tråd  
● Kall for å blokkere/vente på en betinget variabel:  
pthread\_cond\_wait(&cond\_var, &lock);  
● Kallende tråd blokkeres inntil en annen tråd sender et signal til  
cond\_var (med et kall på pthread\_cond\_signal)  
pthread\_cond\_wait vil låse opp mutexen lock før blokkering, og låser mutexen igjen før tråden vekkes opp

Vekke opp blokkert tråd med betinget variabel  
● Kall for å sende et "oppvåknings-signal" til en betinget variabel  
pthread\_cond\_signal(&cond\_var);  
● En av trådene som venter på cond\_var (fordi den har kalt pthread\_cond\_wait) vekkes opp igjen  
Hvis ingen tråder venter på variabelen, ignoreres signalet  
Betingende variable teller ikke opp antallet wake-ups slik som semaforer gjør

hread Call Description  
pthread\_cond\_init Create a condition variable  
pthread\_cond\_destroy Destroy a condition variable  
pthread \_cond\_wait Block waiting for a signal  
pthread\_cond\_signal Signal another thread and wake it up  
pthread\_cond\_broadcast Signal multiple threads and wake all of them

Programmering med betingede variable: Er ofte "fiklete" å få til å fungere, vanskelig å forstå og forutse hva som vil/kan skje i kode  
●Deadlocks, livelocks og starvation kan forekomme  
● Mekanismene garanterer ikke mutual exclusion eller riktige rekkefølger, programmeren må bruke dem riktig  
● Spesielt bør (må?) det sjekkes at en betingelse faktisk er oppfylt når en blokkering på en betinget variabel frigis\*:  
pthread\_mutex\_lock(&lock);  
while (!betingelse)  
pthread\_cond\_wait(&cond\_var, &lock);  
pthread\_mutex\_unlock(&lock);  
\*. Bl.a. fordi runtime-systemet velger hvilken av flere tråder som frigis når det sendes signal til variabelen

Vi kan også bruke named pipes i C-programmene våre for å få prosesser til å snakke sammen. For å opprette en fil som er en named pipe, brukes systemkallet *mkfifo()*:

#include <sys/stat.h>

#include <sys/types.h>

int mkfifo(const char \*pathname, mode\_t mode);

For å bruke *mkfifo()* må vi inkludere headerfilene *sys/types.h* og *sys/stat.h*, som vist ovenfor. Første parameter til *mkfifo()* er filnavnet (evt. inkludert full søkesti) for pipe-filen. Hvis filen ikke finnes fra før, opprettes den. Hvis den finnes fra før, vil kallet ikke ha noen effekt. Parameteren *mode* er en såkalt "file mode" som angir tilgangsrettigheter til filen. Vi kommer tilbake til file modes litt senere i kurset, i eksemplene nedenfor bruker vi bare en standardverdi for *mode*.

Signaler kan brukes i interprosesskommunikasjon, til å synkronisere prosesser og få dem til å samarbeide. Vi kan f.eks. la en barneprosess sende et signal til forelderprosessen for å si fra at den er ferdig med å utføre en bestemt oppgave.

Når vi bruker signaler på denne måten i C/Linux, må vi programmere egne funksjoner som håndterer signalene. Slike funksjoner ("signal handlers") aktiveres/kalles automatisk av systemet når en prosess mottar et bestemt signal.

Linux tilbyr bl.a. systemkallet *signal()* for signalhåndtering:

#include <signal.h>

sighandler\_t signal(int signum, sighandler\_t handler);

Her er *sighandler\_t* en datatype som er definert i headerfilen *signal.h*. Systemkallet *signal()* har to parametre:

* Første parameter, *signum*, angir hvilket signal som skal få en signalhåndterer til å starte. De to signalene *SIGKILL* og *SIGSTOP* kan ikke stoppes og håndteres med *signal()*, de vil alltid sendes direkte til operativsystemet for håndtering der.
* Andre parameter, *sighandler*, kan være en peker til en funksjon som vi oftest vil ha skrevet selv. Det er denne funksjonen som startes av *signal* og gjør selve signalhåndteringen. Funksjonen kan være av typen *void*. Den skal ha én parameter, en *int* som vil være signalnummeret som funksjonen håndterer.
* Andre parameter kan også ha en av to spesielle verdier som er definert i *signal.h*):
  + *SIG\_DFL*: Signalet skal håndteres på standard/default måte (se [GNU Standard Signals](https://www.gnu.org/software/libc/manual/html_node/Standard-Signals.html)

 [Links to an external site.](https://www.gnu.org/software/libc/manual/html_node/Standard-Signals.html)).

 *SIG\_IGN*: Signalet ignoreres, prosessen bare fortsetter uten noe videre avbrudd.

En kompilator, som oversetter programkode skrevet i f.eks. C til eksekverbar maskinkode, er et eksempel der det er nyttig å kunne ha flere uavhengige minneområder

Segmentation: Compilers  
Examples of dynamic/growing tables generated by compiler:  
1. The source text being saved for the printed listing  
2. The symbol table, names and attributes of variables.  
3. The table containing integer and floating-point constants  
used.  
4. The parse tree, syntactic analysis of the program.  
5. The stack used for procedure calls within compiler.

Bruk av segmentering (ikke i Linux): Et segment inneholder ofte en logisk samling av data  
eller programkode, f.eks.: – Koden som svarer til en metode/funksjon i et program   
– Dataene som ligger lagret i en array  
Gjør det enklere å lage programmoduler som kan deles og gjenbrukes, fordi de har eget adresserom. Segmenter kan håndteres individuelt mht. både beskyttelse og deling med andre prosesser: – Et arraysegment kan f.eks. ha lese- og skrivetilgang, men ikke eksekveringstilgang – Et segment med programkode kan f.eks. bare ha  
eksekveringstilgang, slik at koden ikke kan forandres

Implementasjon av segmentert minne: Memory Manager bruker en segmenttabell for hver prosess  
For hvert minnesegment inneholder tabellen bl.a.: – Starten på segmentet i minnet   
– Lengden på segmentet – Tilgangsinformasjon – lesetilgang, skrivetilgang, eksekverbart – Statusinformasjon – er segmentet i RAM eller swappet ut til disk  
Segmentert minne kan implementeres enten uten paging ("pure segmentation") eller med bruk av paging. Mange mikroprosessorer støtter implementasjon av  
segmentert minne i hardware

Segmentert minne med paging: Hvert segment betraktes som virtuelt minne oppdelt i pages. Segmenttabellen peker ikke til minnet, men til page tabeller. Paging og page faults kan håndteres på samme måte som for vanlig virtuelt minne. Segmenteringen blir bare et ekstra lag med abstraksjon i minnehåndteringen.

Reservering av en eksklusiv ressurs i software: Eksempel med pthreads i C  
#include <pthread.h>  
pthread\_mutex\_t resource\_1;  
void thread\_A(void \*arg)  
{  
pthread\_mutex\_lock(&resource\_1);  
use\_resource\_1();  
pthread\_mutex\_unlock(&resource\_1);  
}

Reservering av flere ressurser med pthreads:  
#include <pthread.h>  
pthread\_mutex\_t resource\_1;  
pthread\_mutex\_t resource\_2;  
void thread\_A(void \*arg)  
{  
pthread\_mutex\_lock(&resource\_1);  
pthread\_mutex\_lock(&resource\_2);  
use\_both\_resources();  
pthread\_mutex\_unlock(&resource\_2);  
pthread\_mutex\_unlock(&resource\_1);  
}

To prosesser/trader og to eksklusive ressurser: Vil virke så lenge alle tråder/prosesser "oppfører seg pent" og låser/frigir mutex før/etter bruk av ressurser. Koden vist her kan ikke gi deadlock hvis operasjonene på mutex er atomære (udelelige). En tråd som må vente på ressurs 1 vil ikke kunne kjøre videre før den andre tråden har frigitt begge ressursene  
  
#include <pthread.h>  
pthread\_mutex\_t resource\_1;  
pthread\_mutex\_t resource\_2;  
void thread\_A(void \*arg)  
{  
pthread\_mutex\_lock(&resource\_1);  
pthread\_mutex\_lock(&resource\_2);  
use\_both\_resources();  
pthread\_mutex\_unlock(&resource\_2);  
pthread\_mutex\_unlock(&resource\_1);  
}  
void thread\_B(void \*arg)  
{  
pthread\_mutex\_lock(&resource\_1);  
pthread\_mutex\_lock(&resource\_2);  
use\_both\_resources();  
pthread\_mutex\_unlock(&resource\_2);  
pthread\_mutex\_unlock(&resource\_1);  
}

The program has minimal functionality and even worse error reporting, but it gives a reasonable idea of how some of the system calls related to files work.

/\* File copy program. Error checking and reporting is minimal. \*/

#include <sys/types.h> /\* include necessary header files \*/

#include <fcntl.h>

#include <stdlib.h>

#include <unistd.h>

int main(int argc, char \*argv[]); /\* ANSI prototype \* /

#define BUF SIZE 4096 /\* use a buffer size of 4096 bytes \*/

#define OUTPUT MODE 0700 /\* protection bits for output file \*/

int main(int argc, char \*argv[])

{

int in fd, out fd, rd count, wt count;

char buffer[BUF SIZE];

if (argc != 3) exit(1); /\* syntax error if argc is not 3 \*/

/ \* Open the input file and create the output file \*/

in fd = open(argv[1], O RDONLY); /\* open the source file \*/

if (in fd < 0) exit(2); /\* if it cannot be opened, exit \* /

out fd = creat(argv[2], OUTPUT MODE); /\* create the destination file \*/

if (out fd < 0) exit(3); /\* if it cannot be created, exit \*/

/ \* Copy loop \*/

while (TRUE) {

rd count = read(in fd, buffer, BUF SIZE); /\* read a block of data \*/

if (rd count <= 0) break; /\* if end of file or error, exit loop \*/

wt count = write(out fd, buffer, rd count); /\* write data \* /

if (wt count <= 0) exit(4); /\* wt count <= 0 is an error \*/

}

/ \* Close the files \*/

close(in fd);

close(out fd);

if (rd count == 0) /\* no error on last read \*/

exit(0);

else

exit(5); /\* error on last read \*/

}

The allowed system calls for managing directories exhibit more variation from system to system than system calls for files. To give an impression of what they are and how they work, we will give a sample (taken from UNIX).   
1. Create. A directory is created. It is empty except for dot and dotdot, which are put there automatically by the system (or in a few cases, by the mkdir program).   
2. Delete. A directory is deleted. Only an empty directory can be deleted. A directory containing only dot and dotdot is considered empty as these cannot be deleted.   
3. Opendir. Directories can be read. For example, to list all the files in a directory, a listing program opens the directory to read out the names of all the files it contains. Before a directory can be read, it must be opened, analogous to opening and reading a file.  
4. Closedir. When a directory has been read, it should be closed to free up internal table space.  
5. Readdir. This call returns the next entry in an open directory. Formerly, it was possible to read directories using the usual read system call, but that approach has the disadvantage of forcing the programmer to know and deal with the internal structure of directories. In contrast, readdir always returns one entry in a standard format, no matter which of the possible directory structures is being used.  
6. Rename. In many respects, directories are just like files and can be renamed the same way files can be.  
7. Link. Linking is a technique that allows a file to appear in more than one directory. This system call specifies an existing file and a path name, and creates a link from the existing file to the name specified by the path. In this way, the same file may appear in multiple directories. A link of this kind, which increments the counter in the file’s i-node (to keep track of the number of directory entries containing the file), is sometimes called a hard link.  
8. Unlink. A directory entry is removed. If the file being unlinked is only present in one directory (the normal case), it is removed from the file system. If it is present in multiple directories, only the path name specified is removed. The others remain. In UNIX, the system call for deleting files (discussed earlier) is, in fact, unlink. The above list gives the most important calls, but there are a few others as well, for example, for managing the protection information associated with a directory. A variant on the idea of linking files is the symbolic link. Instead, of having two names point to the same internal data structure representing a file, a name can be created that points to a tiny file naming another file. When the first file is used, for example, opened, the file system follows the path and finds the name at the end. Then it starts the lookup process all over using the new name. Symbolic links have the advantage that they can cross disk boundaries and even name files on remote computers. Their implementation is somewhat less efficient than hard links though.

Systemkall i C for filoperasjoner i Linux: Prosesser i Linux har en array der hvert element inneholder informasjon om én åpen fil som prosessen har tilgang til. Systemkallene for filer bruker en indeks i denne tabellen – en fildeskriptor – for å referere til filen som skal brukes  
● De tre første fildeskriptorene – 0, 1 og 2 – er reservert for: standard input: 0, standard output: 1, standard error: 2  
● Andre filer som åpnes av prosessen får fildeskriptor 3, 4, 5, ...  
● Noen systemkall som bruker slike fildeskriptorer \*:  
open, close, creat, read, write, lseek

\*: Det finnes mange flere. Full dokumentasjon finnes i manualsystemet til Linux, se f.eks.: man 2 lseek

Systemkall\*: Åpne og lukke en fil  
int open(char\* filename, int flags [, int mode ]);  
int close(int file\_descriptor);  
● open returnerer fildeskriptor til filen som er åpnet, -1 hvis feil  
● flags (evt. med bit-for-bit OR i mellom hvis flere skal brukes):  
O\_RDONLY Opens the file for reading  
O\_WRONLY Opens the file for writing.  
O\_RDWR Opens for reading and writing  
O\_APPEND Start writing at end of file  
O\_CREAT File is created if not already existing  
O\_TRUNC If file exists all of its content will be deleted  
● mode: Tilgangrettigheter for filen, kan sløyfes  
\*: Bruk av systemkallene for fil-I/O krever at vi inkluderer headerfilene <unistd.h> og <fcntl.h>

Systemkall: Lage en ny tom fil  
int creat(char\* filename, int mode);  
● creat returnerer fildeskriptor til filen som er laget, -1 hvis feil  
● Filen åpnes for skriving  
● Hvis filen finnes fra før, slettes alt innholdet i "gammel fil"(!):  
● Et kall på creat virker på samme måte som:  
open(path, O\_WRONLY | O\_CREAT | O\_TRUNC, mode);  
● mode: Tilgangrettigheter for filen

Systemkall: Lese/skrive fra/til fil  
int read(int fd, void\* buf, int cnt);  
int write(int fd, void\* buf, int cnt);  
● Leser (skriver) cnt bytes fra (til) filen med fildeskriptor fd  
● Data som leses (skrives) lagres i (hentes fra) arrayen buf  
● Lesing/skriving starter i nåværende posisjon på filen, flytter  
nåværende posisjon cnt bytes fremover  
● Returnerer:  
○ Antall bytes lest/skrevet hvis operasjonen gikk bra  
○ 0 (null) hvis end-of-file lest/overskrevet  
○ -1 ved feil

Systemkall: Sette nåværende posisjon i filen  
int lseek(int fd, int offset, int ref);  
● Flytter nåværende posisjon på filen med fildeskriptor fd  
● offset – Antall bytes (pluss eller minus) posisjonen skal flyttes  
● ref – Startpunkt for hvor ny posisjon skal beregnes fra:  
○ SEEK\_SET Fra starten på filen  
○ SEEK\_CUR Fra nåværende posisjon  
○ SEEK\_END Fra slutten på filen  
● Returnerer:  
○ Antall bytes fra start på fil til ny posisjon  
○ -1 ved feil