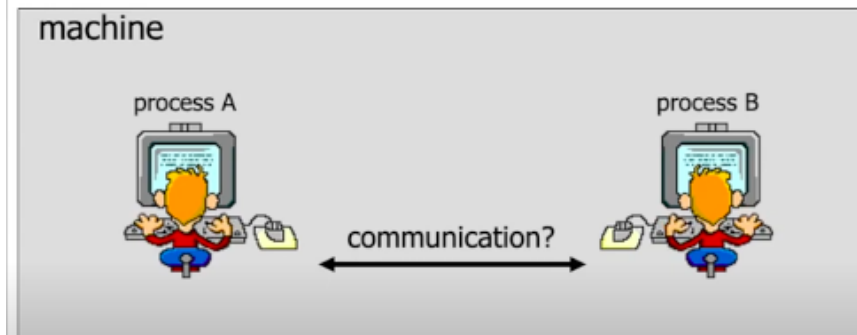


Big Picture



I denne delen så introduserer Pål til kommunikasjon mellom klienter som vi ser i bildet over hvor han da viser et scenario. Scenarioet går ut på at, hvordan skal en datamaskin med prosess "A" utveksle meldinger til en annen datamaskin med prosess "B"? Det skal gås mer i dybden av Carsten, men i denne forelesningen så går vi igjennom ulike måter å utveksle meldinger på.

Meldingsutveksling

Message-passing går ut på å kommunisere igjennom adresseområder og beskyttelsesdomener. For å kunne gjøre dette så trenger man to enkle primitiver som er da følgende: Noe å sende; `send(dest(destinasjon), &msg(melding))` og motta; `recv(src, &msg(melding))`. Nå til å beskrive hva "dest", "src" og "msg" kan være.

What should the "dest" and "src" be?

- pid
- file, e.g., a pipe
- port: network address, etc
- no dest: send to all
- no src: receive any message

"Dest" kan være en prosessorid hvor da prosessor med id "517" skal sende til destinasjonasjon til en prosessor med id "529".

What should "msg" be? We need ...

- buffer to store the message
- size ? ... for variable sized messages
- type ? ... to distinguish between messages

(Hvordan ta høyde for variabelstørrelse og typen variabler?)

Direct Communication



- Must explicitly name the sender/receiver ("`dest`" and "`src`") processes
- Requires buffers...
 - ... at the receiver
 - more than one process may send messages to the receiver
 - to receive from a specific sender, it requires searching through the whole buffer
 - ... at each sender
 - a sender may send messages to multiple receivers

Man trenger å vite adressering av to prosesser som utveksle meldinger til hverandre. Dette kan da være en utfordring hvor å finne den riktige adressen som meldingen skal sendes til. Trenger i tillegg bufferet som begge prosesser trenger for å kunne oppbevare meldingene som sendes og mottas mellom hverandre.

Indirect Communication



Man har en form for indirekte kommunikasjon hvor man har denne postboksen som fungerer som et slags mellomledd mellom to prosesser. Når prosessen til venstre legger til meldinger i denne postboksen, så kan den andre prosessen hente meldingen som er lagt til i postboksen.

- "`dest`" and "`src`" are shared (unique) queues
 - a shared queue also allows many-to-many communication
- Where should the buffer be?
 - a buffer (and its mutex, conditions, ...) should be at the mailbox

Sender og mottaker-adressene er en felles unik-kø med tanke på postboksen over. Det fører til mange-til-mange kommunikasjon hvor flere kan da dele mail med hverandre.

Postbokser

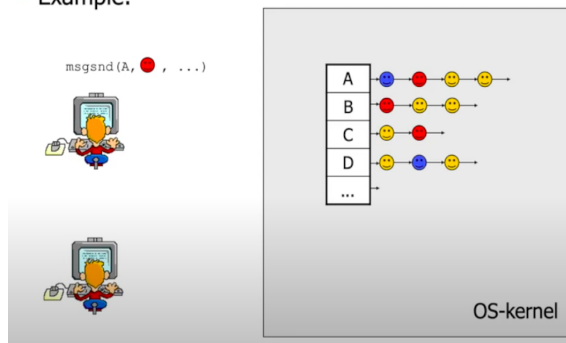
Hvordan skal man implementere slike postbokser eller meldingskøer som vi så over? En løsning er å implementere dem som en "FIFO"-kø hvor meldingene blir sortert i en FIFO-rekkefølge.

- messages are stored as a sequence of bytes
- system V IPC messages also have a type:

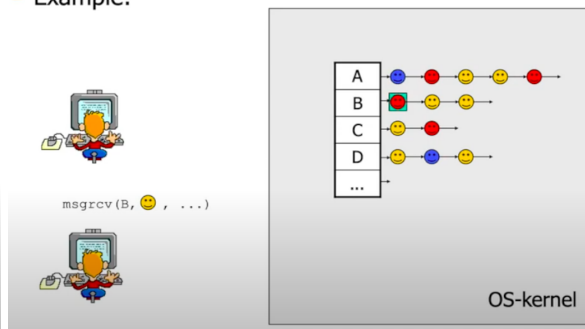

```
struct mymsg {
    long mtype;
    char mtext[..];
}
```
- get/create a message queue identifier: `Qid = msgget(key, flags)`
- sending messages: `msgsnd(Qid, *mymsg, size, flags)`
- receiving messages: `msgrcv(Qid, *mymsg, size, type, flags)`
- control a shared segment: `msgctl(...)`

Vi ser da ulike kommandoer som vi ser over som er selvforklarende. F.eks `msgget(key,flags)` som oppretter en identifikator for meldingskøen for å finne den. Videre har vi send og receive som gjør som de beskriver. I tillegg til en kontroll som kan sette rettigheter på meldingen hvor en rettighet kan være at meldingen kan “leses” eller “skrives” på.

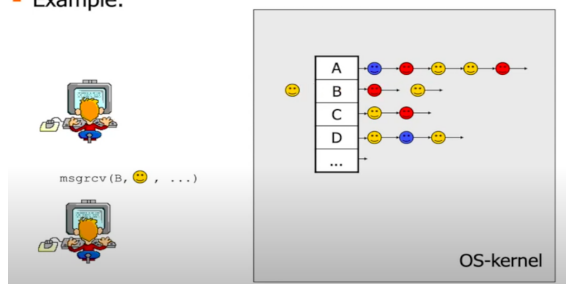
■ Example:



■ Example:



■ Example:



Nå til å vise hvordan dette gjøres i praksis, vi har da en melding som skal bli lagt til inn i postboksen av med id “A” og av typen “rød”. Denne skal settes helt mot slutten siden det er “FIFO”-prinsipp. Videre har vi bildet til høyre hvor vi skal hente ut meldingen av type “gul”. Da vil vi hente siste forekomst av denne i bildet nederst til venstre.

Mailboxes Example – command line **send**

```
#include <stdio.h> ... /* More includes in the real example files */
#define MSGLEN 100

struct text_message { long mtype; char mtext[MSGLEN]; };

int main(int argc, char *argv[])
{ int msqID, len;
  struct text_message msg;

  if (argc != 4) { printf("Usage: msgsnd <key> <type> <text>\n"); exit(1); }

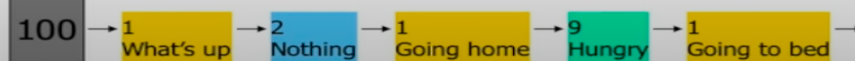
  len = strlen(argv[3]);
  if (len > MSGLEN-1) { printf("String too long\n"); exit(1); }

  /* get the message queue, which may need to be created */
  msqID = msgget((key_t) atoi(argv[1]), IPC_CREAT | 0666);
  if (msqID == -1) { perror("msgget"); exit(1); }

  /* build message */
  msg.mtype = atoi(argv[2]);
  strcpy(msg.mtext, argv[3]);

  /* place message on the queue */
  if (msgsnd(msqID, (struct msgbuf *) &msg, len+1, 0) == -1) {
    perror("msgsnd");
    exit(1);
  }
}
```

```
> ./msgsnd 100 1 "What's up"
> ./msgsnd 100 2 "Nothing"
> ./msgsnd 100 1 "Going home"
> ./msgsnd 100 9 "Hungry?"
> ./msgsnd 100 1 "Going to bed"
```



Over så beskriver vi hvordan “send” gjøres. Setter da ulike rettigheter på hva som skal gjøres med meldingen.

Mailboxes Example – command line **rcv**

```
#include <stdio.h> ... /* More includes in the real example files */
#define MSGLEN 100

struct text_message { long mtype; char mtext[MSGLEN]; };

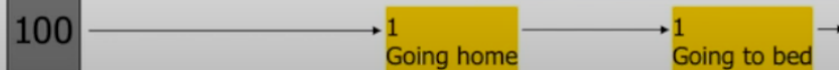
int main(int argc, char *argv[])
{
  int msqID;
  struct text_message msg;

  if (argc != 3) { printf("Usage: msgrcv <key> <type>\n"); exit(1); }

  /* get the existing message queue */
  msqID = msgget((key_t)atoi(argv[1]), 0);
  if (msqID == -1) { perror("msgget"); exit(1); }

  /* read message of the specified type; do not block */
  if (msgrcv(msqID, (void *) &msg, MSGLEN, atoi(argv[2]), IPC_NOWAIT) == -1)
  {
    if (errno == ENOMSG) printf("No suitable message\n");
    else printf("msgrcv() error\n");
  }
  else
    printf("[%ld] %s\n", msg.mtype, msg.mtext);
}
```

```
> ./msgrcv 100 1
[1] What's up
> ./msgrcv 100 9
[9] Hungry
> ./msgrcv 100 0
[2] Nothing
```



Over beskriver hvordan man henter melding, altså prosessen for dette.

Mailboxes Example – command line **ctl**

```
#include <stdio.h> ... /* More includes in the real example files */

int main(int argc, char *argv[])
{
    key_t mkey;
    int msqid;
    struct msqid_ds mstatus;

    if (argc != 2) { printf("Usage: show_Q_stat <key>\n"); exit(1); }

    /* access existing queue */
    mkey = (key_t) atoi(argv[1]);
    if ((msqid = msgget(mkey, 0)) == -1) { perror("msgget"); exit(2); }

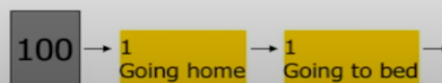
    /* get status information */
    if (msgctl(msqid, IPC_STAT, &mstatus) == -1) { perror("msgctl"); exit(3); }

    /* print status info */
    printf("\nKey %d, queue ID %d, ", (long int) mkey, msqid);
    printf("%d msgs on queue\n", mstatus.msg_qnum);
    printf("Last send by pid %d at %s\n", mstatus.msg_lspid, ctime(&(mstatus.msg_stime)));
    printf("Last rcv by pid %d at %s\n", mstatus.msg_lrpid, ctime(&(mstatus.msg_rtime)));
}
```

```
> ./show_Q_stat 100
```

```
Key 100, queue ID 0, 2 msgs on queue
```

```
Last send by pid 17345 at Wed Oct 12 8:37:56 2016
Last rcv by pid 17402 at Wed Oct 12 8:39:45 2016
```

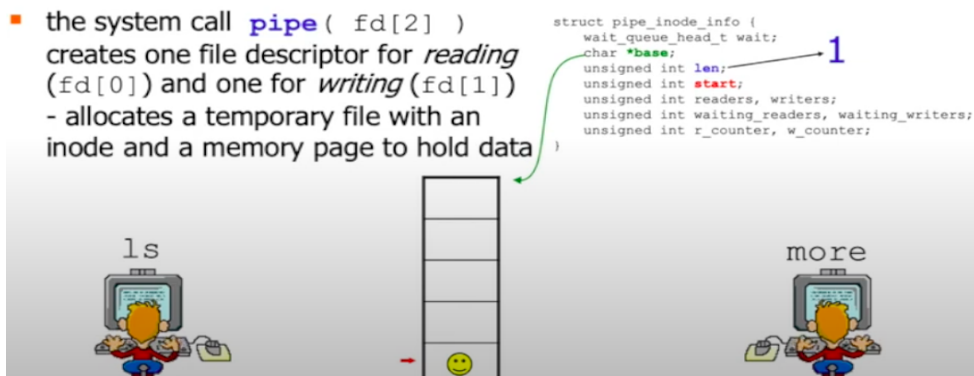


Tilslutt har vi en kontroll som beskriver meldingsutvekslingen mellom ulike prosesser.

Pipes

- Classic IPC method under UNIX:
 - > `ls -l | more`
 - shell runs two processes `ls` and `more` which are linked via a pipe
 - the first process (`ls`) writes data (e.g., using `write`) to the pipe and the second (`more`) reads data (e.g., using `read`) from the pipe

Pipes er en metode for å hente data fra en prosess og overfører dets output til en annen.



Fungerer slik at pipen allokerer et minneområde med to fildeskriptorer som fungerer henholdsvis for skrivning og lesing. Når “ls” skal skrive data til pipen, så vil denne outputen settes inn i pipen. “more” kan da lese dataen som finnes i pipen og når dette gjøres, så vil “inoden” oppdateres slik at vi kan hente neste data som er lagret i pipen.

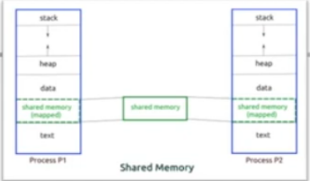
Postbokser vs Pipes

- Are there any differences between a mailbox and a pipe?
 - Message types
 - mailboxes may have messages of different 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

Over ser vi forskjellene mellom begge to og er ganske selvforklarende ved å se på forskjellene mellom dem.

Delt minne

Shared Memory

- Shared memory is an efficient and fast way for processes to communicate
 - multiple processes can attach a segment of physical memory to their virtual address space
- 
- create a shared segment: `shmid = shmget(key, size, flags)`
 - attach a shared segment: `shmat(shmid, *shmaddr, flags)`
 - detach a shared segment: `shmdt(*shmaddr)`
 - control a shared segment: `shmctl(shmid, cmd, *buf)`
 - if more than one process can access a segment, an outside protocol or mechanism (like semaphores) should enforce consistency/avoid collisions

Delt minne fungerer slik som selve begrepet er beskrevet, man har et minneområdet som er delt mellom to prosesser. Flere prosesser kan knytte seg til et segment av det fysiske minne til deres virtuelle adresseområdet slik at en prosess kan da kommunisere med en annen.

Det kan oppstå en del utfordringer som beskriver hvordan man ikke skal overskrive data som ikke burde blitt overskrevet, osv, men dette går ikke i detalj i dette kurset. Men det fungerer slik at hvis en prosess skriver data til dette delte minneområdet, så kan den andre prosessen direkte lese inn disse dataene.

Signaler

- Signals are software generated "interrupts" sent to a process
 - hardware conditions
 - software conditions
 - input/output notification
 - process control
 - resource control
- Sending signals
 - `kill(pid, signal)` – system call to send any *signal* to *pid*
 - `raise(signal)` – call to send *signal* to current process
 - `kill(getpid(), signal)`
 - `pthread_kill(pthread_self(), signal)`

Signaler er en annen måte for meldingsutvekslinger som signaliserer “interrupts” som blir sendt til en prosess. Dette kan være om en prosess gjør noe uforventet og dermed blir den tilsendt en signal og da blir prosessen avbrutt for meldingsutveksling inntil videre. Har da “kill” og “raise” som gjør som blir beskrevet over.

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 (some signals cannot be ignored)
 - `signal(sig_nr, SIG_IGN)`
 - `SIG_KILL` and `SIG_STOP` cannot be blocked
 - catch the signal with a handler
 - `signal(sig_nr, void (*func)())`
 - write a function yourself - `void func() {}`

“Signal-handler” er en type funksjon som opptrer når en gitt signal(“raise” eller “kill”) blir sendt. En slik signal kan være “CTRL+C” i terminalen når du vil at programmet skal stoppe som da “signal-handler” håndterer og vil da avslutte programmet. Men det går ann å kontrollerer hvordan signaler skal håndteres hvor man kan “disable CTRL-C”.

```
#include <stdio.h>
#include <signal.h>

void sigproc()
{
    signal(SIGINT, sigproc); /* NOTE some versions of UNIX will reset
                             * signal to default after each call. So for
                             * portability reset signal each time */

    printf("you have pressed ctrl-c - disabled \n");
}

void quitproc()
{
    printf("ctrl-\\ pressed to quit\n"); /* this is "ctrl" & "\\ " */
    exit(0); /* normal exit status */
}

main()
{
    signal(SIGINT, sigproc); /* ctrl-c : DEFAULT ACTION: term */
    signal(SIGQUIT, quitproc); /* ctrl-\\ : DEFAULT ACTION: term */
    printf("ctrl-c disabled use ctrl-\\ to quit\n");

    for(;;);
}
```

I bildet over så kan man da kode inn slik at man kan “disable CTRL-C” ved å endre oppførselen dets fra å ikke avslutte programmet, men gjøre noe annet.