INF1060:
Introduction to Operating Systems and Data Communication

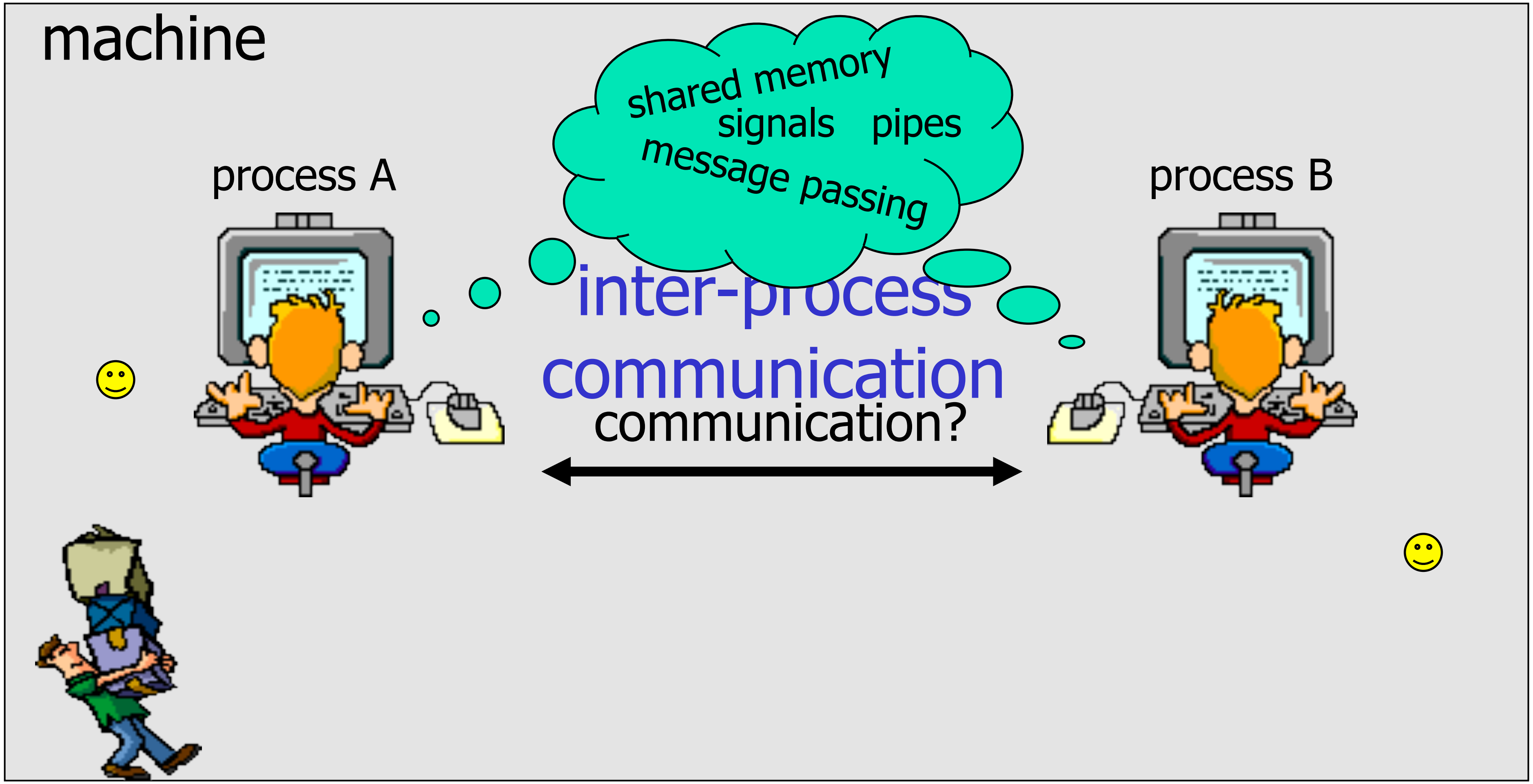
## Operating Systems:

# Inter-Process Communication

Pål Halvorsen

Monday, November 19, 2012

# Big Picture


machine
shared memory
signals pipes
message passing
process A
process B
inter-process
communication
communication?

# Message Passing

- Threads may communicate using shared variables in the same address space

- What is message-passing for?
  - communication across address spaces and protection domains
  - synchronization

- Generic API
  - `send( dest, &msg )`
  - `recv( src, &msg )`

- 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

- What should "`msg`" be?
  - need both buffer and size for a variable sized message

# 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

# Indirect Communication

- "`dest`" and "`src`" are a shared (unique) queue
- Use a shared queue to allow many-to-many communication
- Where should the buffer be?
  - a buffer (and its mutex and conditions) should be at the mailbox

# Mailboxes

- Mailboxes are implemented as message queues sorting messages according to FIFO
  - messages are stored as a sequence of bytes
  - system V IPC messages also have a type:

```
struct mymsg {
         long mtype;
         char mtext[..];
}
```

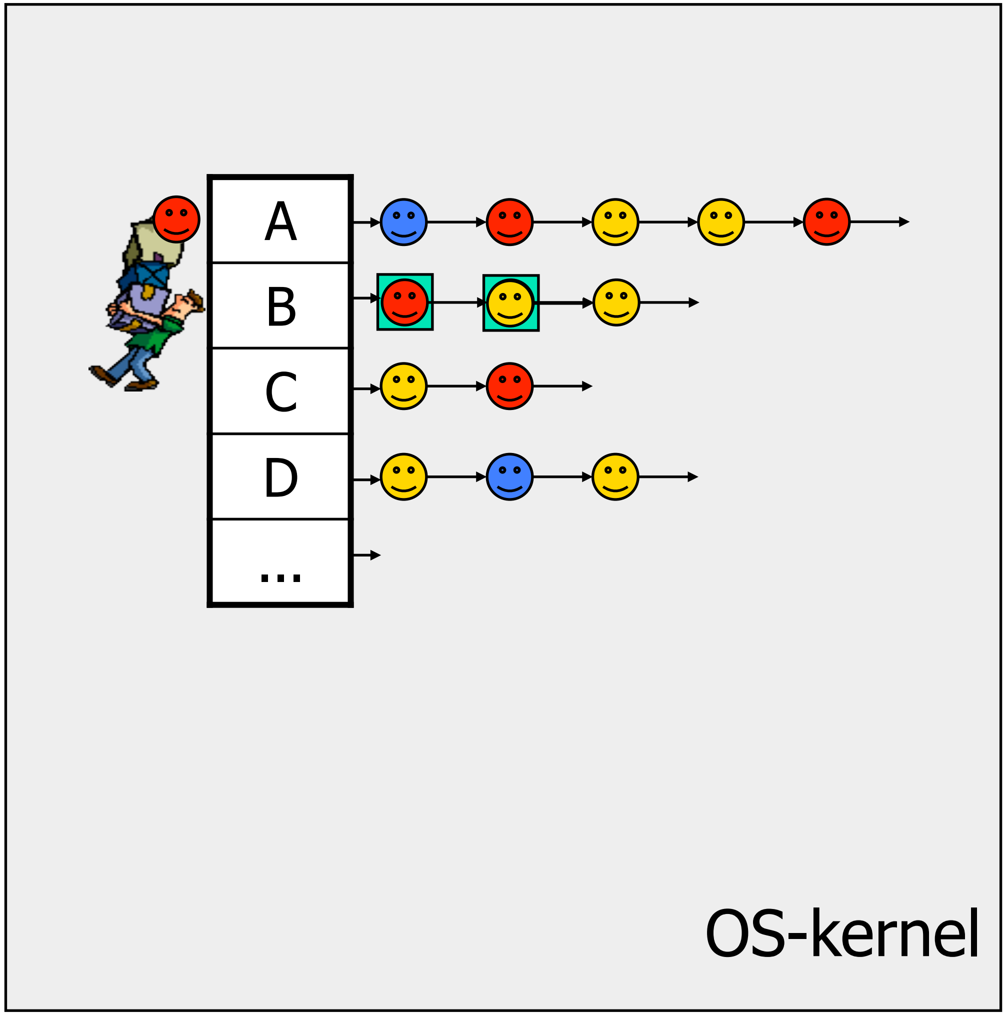
  - get/create a message queue identifier: `Qid = msgget(key, flags)`
  - sending messages: `msgsnd(Qid, *mymsg, size, flags)`
  - receiving messages: `msgrcv(Qid, *mymsg, size, type, flags)`
  - control a shared segment: `msgctl( … )`

# Mailboxes

- Example:

```
msgsnd(A, ☺ , ...)
```

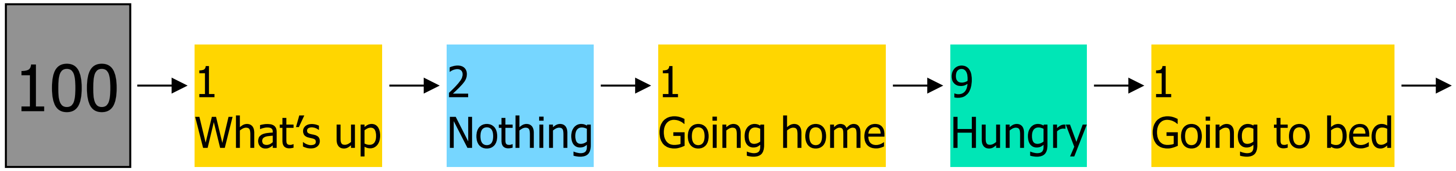
```
msgrcv(B, ☺ , ...)
```



# 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 mesg;

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

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

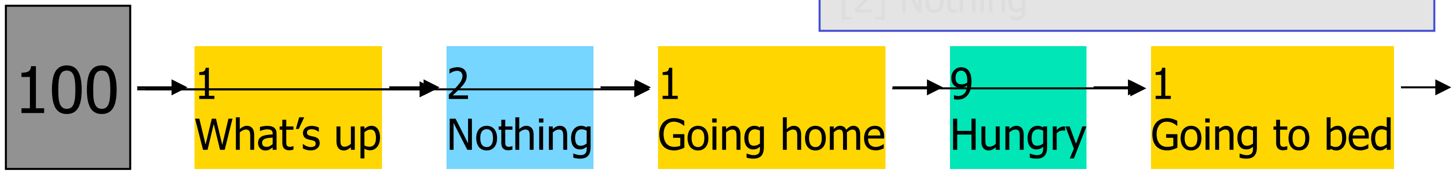
```
> ./msgsnd 100 1 "What's up"
> ./msgsnd 100 2 "Nothing"
> ./msgsnd 100 1 "Going home"
> ./msgsnd 100 9 "Hungry?"
> ./msgsnd 100 1 "Going to sleep"
```



# 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 mesg;

  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, (struct msgbuf *) &mesg, 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", mesg.mtype, mesg.mtext);
}
```

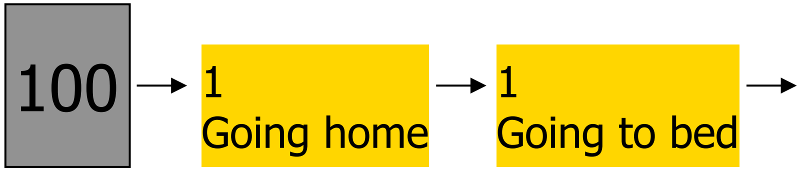
```
> ./msgrcv 100 1
[1] What's up
> ./msgrcv 100 9
[9] Hungry
> ./msgrcv 100 0
[2] Nothing
```



# 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 %ld, queue ID %d, ", (long int) mkey, msqID);
  printf("%d msgs on queue\n\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 Tue Oct 9 10:37:56 2012
Last rcv by pid 17402 at Tue Oct 9 10:39:45 2012
```



# Pipes

- Classic IPC method under UNIX:

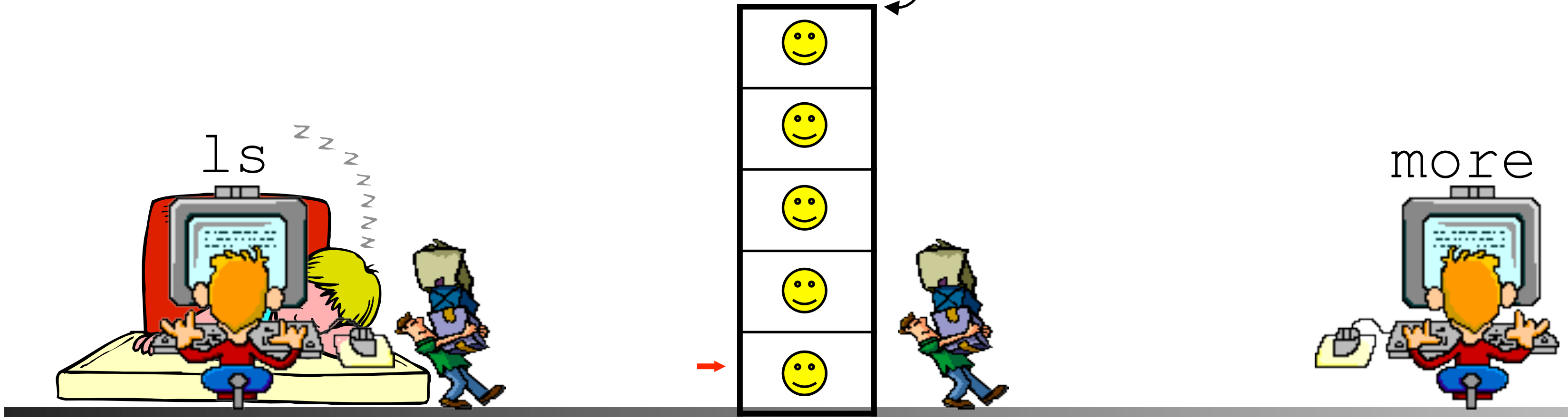
  `> ls -l | more`

  – 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

- the system call **`pipe`**`( fd[2] )` creates one file descriptor for *reading* (`fd[0]`) and one for *writing* (`fd[1]`) - allocates a temporary file with an inode and a memory page to hold data

```
struct pipe_inode_info {
    wait_queue_head_t wait;
    char *base;
    unsigned int len;
    unsigned int start;
    unsigned int readers, writers;
    unsigned int waiting_readers, waiting_writers;
    unsigned int r_counter, w_counter;
}
```



# Pipe Example – fork,child writing to parent

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

char *msg = "hello";

main()
{
   char inbuf[MSGSIZE];
   int p[2];
   pid_t pid;

   /* open pipe */
   if (pipe(p) == -1) { perror("pipe call error"); exit(1); }

   switch( pid = fork() ) {

   case -1: perror("error: fork call");
            exit(2);

   case 0:  close(p[0]);  /* CHILD: close the read end of the pipe */
            write(p[1], msg, MSGSIZE);
            printf(“Child: %s\n", msg);
            break;

   default: close(p[1]);  /* PARENT: close the write end of the pipe */
            read(p[0], inbuf, MSGSIZE);
            printf("Parent: %s\n", inbuf);
            wait(0);
   }
   exit(0);
}
```

# Mailboxes 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

# 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 segment, an outside protocol or mechanism (like semaphores) should enforce consistency/avoid collisions

# Shared Memory Example – read/write alphabet

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

#define SHMSZ     27

main()
{
    int shmid;
    key_t key;
    char c, *shm, *s;

    key = 5678; /* selected key */

    /* Create the segment.*/
    if ((shmid = shmget(key,SHMSZ,IPC_CREAT | 0666)) < 0)
     {
        perror("shmget"); exit(1);
    }

    /* Now we attach the segment to our data space.*/
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat"); exit(1);
    }

    /* put some things into the memory */
    for (s = shm, c = 'a'; c <= 'z'; c++) *s++ = c;
    *s = NULL;

    /* wait until first character is changed to '*' */
    while (*shm != '*') sleep(1);

    exit(0);
}
```

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

#define SHMSZ     27

main()
{
    int shmid;
    key_t key;
    char *shm, *s;

    key = 5678; /* selected key by server */

    /* Locate the segment. */
    if ((shmid = shmget(key,SHMSZ,0666)) < 0)
     {
        perror("shmget"); exit(1);
    }

    /* Now we attach the segment to our data space. */
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat"); exit(1);
    }

    /* read what the server put in the memory. */
    for (s = shm; *s != NULL; s++) putchar(*s);
    putchar('\n');

    /* change the first character in segment to '*' */
    *shm = '*';

    exit(0);
}
```

# Signals

- 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)

# 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 Example – 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(;;);
}
```

# Signal Example – parent terminating child

```c
void sighup()
{
    signal(SIGHUP,sighup); /* reset signal */
    printf("CHILD: I received a SIGHUP\n");
}
```

```c
void sigint()
{
    signal(SIGINT,sigint); /* reset signal */
    printf("CHILD: I received a SIGINT\n");
}
```

```c
void sigquit()
{
    printf("My DADDY has Killed me!!!\n");
    exit(0);
}
```

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

void sighup();
void sigint();
void sigquit();

main()
{
    int pid;

    /* get child process */
    if ((pid=fork()) < 0)
    { perror("fork"); exit(1); }

    if (pid == 0) {  /* child */
          signal(SIGHUP, sighup);
          signal(SIGINT, sigint);
          signal(SIGQUIT, sigquit);
          for(;;);
     } else {        /* parent */
          printf("\nPARENT: sending SIGHUP\n\n");
          kill(pid,SIGHUP);
          sleep(3);
          printf("\nPARENT: sending SIGINT\n\n");
          kill(pid,SIGINT);
          sleep(3);
          printf("\nPARENT: sending SIGQUIT\n\n");
          kill(pid,SIGQUIT);
          sleep(3);
     }
}
```

# Summary

- Many ways to send messages or perform IPC within a machine

  - mailboxes – FIFO, messages have types
  - pipes – FIFO, no type
  - shared memory – shared memory mapped into virtual space
  - signals – send a signal which can invoke a special handler

- Next, communication between processes on different machines using **networks** (with Tor Skeie)