Taszkok közötti kommunikáció a gyakorlatban

Unix jelzések keltése és kezelése

- Jelzések keltése

```
#include <signal.h>        /* kill() */
kill(pid, SIGUSR1);        /* jelzés küldése */
```

- Jelzések kezelése

```
signal(SIGALRM, alarm);    /* kezelőfüggvény beállítása */
```

  - beépített jelzéskezelők
    - `Core`: core dump és leállítás (`exit()`)
    - `Term`: leállítás (`exit()`)
    - `Ign`: figyelmen kívül hagyás
    - `Stop`: felfüggesztés
    - `Cont`: visszatérés a felfüggesztett állapotból (vagy ignore)

  - saját kezelőfüggvény

```
signal(SIGALRM, alarm);    /* kezelőfüggvény beállítása */
void alarm(int signum) { ... }  /* a kezelőfüggvény */
```

Nem mindig írható felül az alapértelmezett jelzéskezelő (pl. a SIGKILL nem)

`man -s 7 signal` (részlet)

Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from abort(3)
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers
SIGALRM	14	Term	Timer signal from alarm(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25	Cont	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at tty
SIGTTIN	21,21,26	Stop	tty input for background process
SIGTTOU	22,22,27	Stop	tty output for background process

Példák jelzések használatára (demó)

```
kill(pid, SIGSTOP);          /* STOP jelzés küldése */
kill -STOP <PID>

signal(SIGCLD, SIG_IGN);  /* nem foglalkozunk a gyerekekkel */

signal(SIGINT, SIG_IGN);  /* nem foglalkozunk a ctrl+c jelzéssel */

signal(SIGINT, SIG_DFL);  /* alapértelmezett kezelő beállítása */

signal(SIGALRM, myalarm); /* jelzéskezelő függvény beállítása */
void myalarm(int signum) { ... }  /* az eljárás */
alarm(30);                /* alkalmazás: ALARM jelzés 30mp múlva */
```

Unix csővezetékek: `pipe()`

- Cél: folyamatok közötti adatátvitel (`ls -la | more`)

Hogyan működik?
Megoldás

- Jellemzők
  - **csak szülő-gyerek (leszármazott, testvér) viszonylatban**
  - adatfolyam (nincs üzenethatár)
  - nincs adatszerkezet, adattípus
  - egyirányú adatfolyam (író → olvasó) (több író és olvasó is lehet!)
  - limitált kapacitás: pl. 4k (Linux < 2.6.11), 65k (Linux >= 2.6.11)

Miért?

- Alkalmazás (demó)

```
int pipefd[2];
pipe(pipefd);
```

Miért kell lezárni? Ötlet

  - író

```
close(pipefd[0]);
write(pipefd[1], string,
      strlen(string)+1);
close(pipefd[1]);
```

  - olvasó

```
close(pipefd[1]);
read(pipefd[0], buffer,
     sizeof(buffer));
close(pipefd[0]);
```

Elnevezett csővezetékek (named pipe, FIFO)

- Cél: folyamatok közötti adatátvitel
- Jellemzők
  - független folyamatok között is működő csővezeték
  - fájlrendszeri bejegyzéssel azonosítható (`mkfifo`, `mknod`)
  - kétirányú kommunikáció (megnyitás olvasásra és írásra)
- Alkalmazás + demó

```
int pipefd[2];
char *fifo_fname="/path/to/fifo_filename";
mkfifo(fifo_fname, 0600);
```

  - író

```
pipefd[1] =
  open(fifo_fname, O_WRONLY);
write(pipefd[1], string,
      strlen(string)+1);
close(pipefd[1]);

unlink(fifo_fname);
```

  - olvasó

```
pipefd[0] =
  open(fifo_fname, O_RDONLY);
read(pipefd[0], buffer,
     sizeof(buffer))
close(pipefd[0]);
```

Ki és mikor szünteti meg?

Unix System V IPC / POSIX IPC

- Cél: folyamatok közötti egységes kommunikáció

- Erőforrások
  - adatátvitel: üzenetsor, osztott memória
  - szinkronizáció: szemafor

- Közös alapok
  - kulcs: azonosító az erőforrás eléréséhez (egy 32 bites szám)
  - közös kezelőfüggvények: `*ctl(), *get( … kulcs …)`
  - jogosultsági rendszer (szereplők és hozzáférési szabályok)
  - bővebb infó: `man svipc ipc ipcs`

- POSIX IPC
  - kulcs helyett szöveges azonosítók
  - más (egyszerűbb) kezelőfüggvények
  - példákkal illusztrált különbségek

SysV szemaforok

- Cél: folyamatok közötti szinkronizáció
  - P() és V() operátorok
  - szemaforcsoportok kezelése
- Alkalmazás
  - szemaforok létrehozása (vagy elérése)

```
sem_id = semget(kulcs, szám, opciók);
```

  - az `ops` struktúrában leírt műveletek végrehajtása (részletek: `man semop`):

```
status = semop(sem_id, ops, ops_méret);
```

    - egyszerre több művelet, több szemaforon is végrehajtható
    - blokkoló és nem blokkoló P() operáció is lehetséges
    - egyszerű tranzakciókezelésre is van lehetőség

Unix System V IPC: üzenetsorok

- Cél: folyamatok közötti adatátvitel
  - diszkrét, tipizált üzenetek
  - nincs címzés, üzenetszórás
- Alkalmazás
  - üzenetsor létrehozása (vagy elérése)

```
msgq_id = msgget(kulcs, opciók);
```

  - üzenetküldés és -fogadás

POSIX MQ demó

```
msgsnd(msgq_id, msg, méret, opciók);
```

az msg tartalmaz egy típust is

```
msgrcv(msgq_id, msg, méret, típus, opciók);
```

a `típus` (egész szám) beállításával szűrést valósíthatunk meg

`= 0`	a következő üzenet (tetszőleges típusú)
`> 0`	a következő adott típusú üzenet
`< 0`	a következő üzenet, amelynek a típusa kisebb vagy egyenlő

Unix System V IPC: osztott memória

- Cél: folyamatok közötti egyszerű és gyors adatátvitel
  - PRAM modell szerint működik
- Alkalmazás + demó
  - osztott memória létrehozása (vagy elérése)

```
shm_id = shmget(kulcs, méret, opciók);
```

  - hozzárendelés saját virtuális címtartományhoz

```
változó = (típus) shmat(...);
```

az adott változót hozzákötjük a visszakapott címhez

  - lecsatolás

```
shmdt(cím);
```

  - a kölcsönös kizárást meg kell valósítani (pl. szemaforokkal)

Érdekesség: Make Dragonfly BSD great again!

2017-03-23

```
$ uname -s
DragonFly
$ id
uid=1001(shm) gid=1001(shm) groups=1001(shm)
$ ./shellcode3
$ uname -s
FreeBSD
$ ipcs
Message Queues:
T     ID     KEY        MODE       OWNER    GROUP

Shared Memory:
T     ID     KEY        MODE       OWNER    GROUP

Semaphores:
T     ID     KEY        MODE       OWNER    GROUP
s  65536    4196472 --rw-------      shm      shm

$ ipcrm -S 4196472
$ id
uid=1001(shm) gid=1001(shm) groups=1001(shm)
$ ./final
# id
uid=0(root) gid=0(wheel) egid=1001(shm) groups=1001(shm)
#
```

Hálózati (socket) kommunikáció

- Cél: címzéssel és protokollokkal támogatott adatátvitel
  - kliens – szerver architektúra
  - sokféle célra (egy gépen belül / gépek között)
  - megjelenés: BSD UNIX (Berkeley sockets)
  - később Windows Winsock, POSIX socket

- Fogalmak
  - hálózati csatoló avagy azonosító (socket): a kommunikáció végpontja
  - IP cím és portszám (I. hálózatok)

- Alkalmazás

```
sfd = socket(domén, típus, protokoll);
szerver: bind(sfd, cím, ...);
kliens: connect(sfd, cím, ...);
szerver: listen(sfd, sor_ajánlott_mérete);
szerver: accept(sfd, cím, ...);
send(sfd, üzenet, ...);
recv(sfd, üzenet, ...);
shutdown(sfd);
```

Hálózati (socket) kommunikáció alkalmazása

Kliens program

```
socket()


connect()


send()
recv()
close()
```

Szerver program

```
sfd1 = socket()
bind(sfd1)
listen(sfd1)
while
    sfd2 = accept(sfd1)
    fork()
```

szülő: vissza a ciklusba

gyerek:

```
recv(sfd2)
send(sfd2)
close(sfd2)
exit()
```

Készítsünk egy egyszerű webszervert!

```
sfd1 = socket()
bind(sfd1, ...)
listen(sfd1, 10)
while
    sfd2 = accept(sfd1)
    fork()
    szülő:      close(sfd2)

    gyerek:     recv(sfd2)
                ...
                send(sfd2)
                close(sfd2)
                exit()
```

- Létrehozza a csatolót
- A 8080-as porthoz köti
- Beállítja a várakozási sort

- Beérkező kérésre vár
- Elindít hozzá egy kiszolgálót

  `fork()`

  `pthread_create()`
- Fogadja a kérést

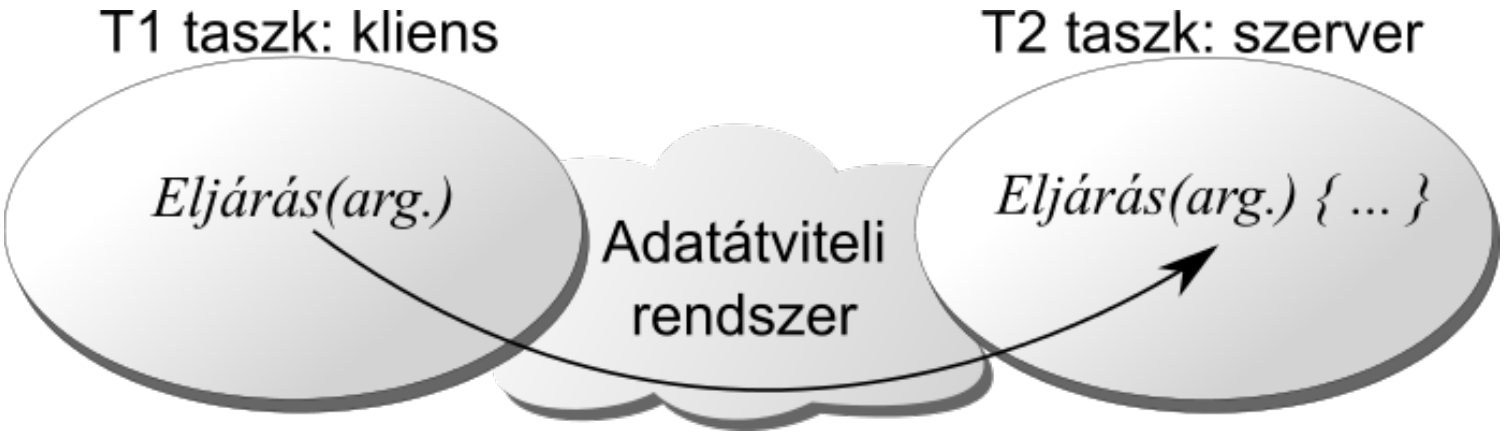
  (A szülő már új kérésre vár.)
- Elküldi a választ
- Lezárja a kliens kapcsolatot
- A kiszolgáló kilép

  (A szülő még fut.)

A szerver (túl)terhelésének kezelése

- A TCP/IP háromfázisú kézfogást használ a kapcsolat felépítésekor
  - a kialakuló kapcsolat először egy SYN állapotba kerül a szerveren (1.)
  - majd szerver (2.) és kliens (3.) nyugtázás után lép ESTABLISHED állapotba
- A párhuzamos kapcsolatok nyilvántartására két lehetőség van
  - egy LISTEN sor minden kapcsolat tárolására (az eredeti BSD megvalósítás)
  - külön-külön SYN és ACCEPT sorok a fenti két állapot számára (pl. Linux)
    - ebben az esetben a `listen()` rendszerhívás az ACCEPT sorra vonatkozik
    - ha betelik az ACCEPT sor (azaz a SYN állapotból nem lehet oda átlépni)
      - a szerver nem nyugtázza a kliens kapcsolatát
      - ezért a kliens később (exponenciálisan növekvő várakozással) újra próbálkozni fog
      - ha közben sikerült helyet felszabadítani az ACCEPT sorban, akkor rendben
      - ha nem, akkor a szerver előbb-utóbb zárni fogja a kapcsolatot
- Az időegység alatt kiszolgálható kérések száma a fő kérdés
  - a `listen()` által beállított várakozási sor kezelheti a **rövid**, átmeneti túlterhelést
  - ha tartósan gyorsabban érkezhetnek be, mint ahogy kiszolgáljuk őket baj van
    - ha a baj a mi folyamatunkban van, akkor növelhetjük a kiszolgálási kapacitást
      - pl. új folyamatokat, vagy szálakat indítunk
    - előfordulhat, hogy kernel szinten telnek be a sorok (rendszerszintű gond)
      - ekkor a kernel kiszolgálás nélkül zárni fog a beérkező kapcsolatokat (nem jó)
      - a szerverkapacitás növelésével és terheléselosztással kezelhető a helyzet

Távoli eljáráshívás (RPC) a gyakorlatban



RPC a gyakorlatban

- Kommunikációs infrastruktúra
  - elosztott rendszer
  - hálózati kommunikáció
    - címzés: gép, taszk, eljárás
    - adatátvitel: paraméterek, eredmények
  - implementációfüggetlen adatátviteli formátum
    - képes az implementációs különbségek áthidalására
  - portmapper: a programazonosítók és a hálózati portok összerendelése

- RPC nyelv
  - a hívható eljárások és típusaik (interfész) leírása
    - megnevezés, azonosító, paraméterek és visszatérési értékek
  - adattípusok és strukturált adatszerkezetek deklarációja
  - implementációfüggetlen

- Programozói támogatás
  - rpcgen: az interfészleírásból programkódot generál
    - kliens csonk
    - szerver csontváz

RPC interfészleírás és programgenerátor

- RPC nyelv (példa: date.x)

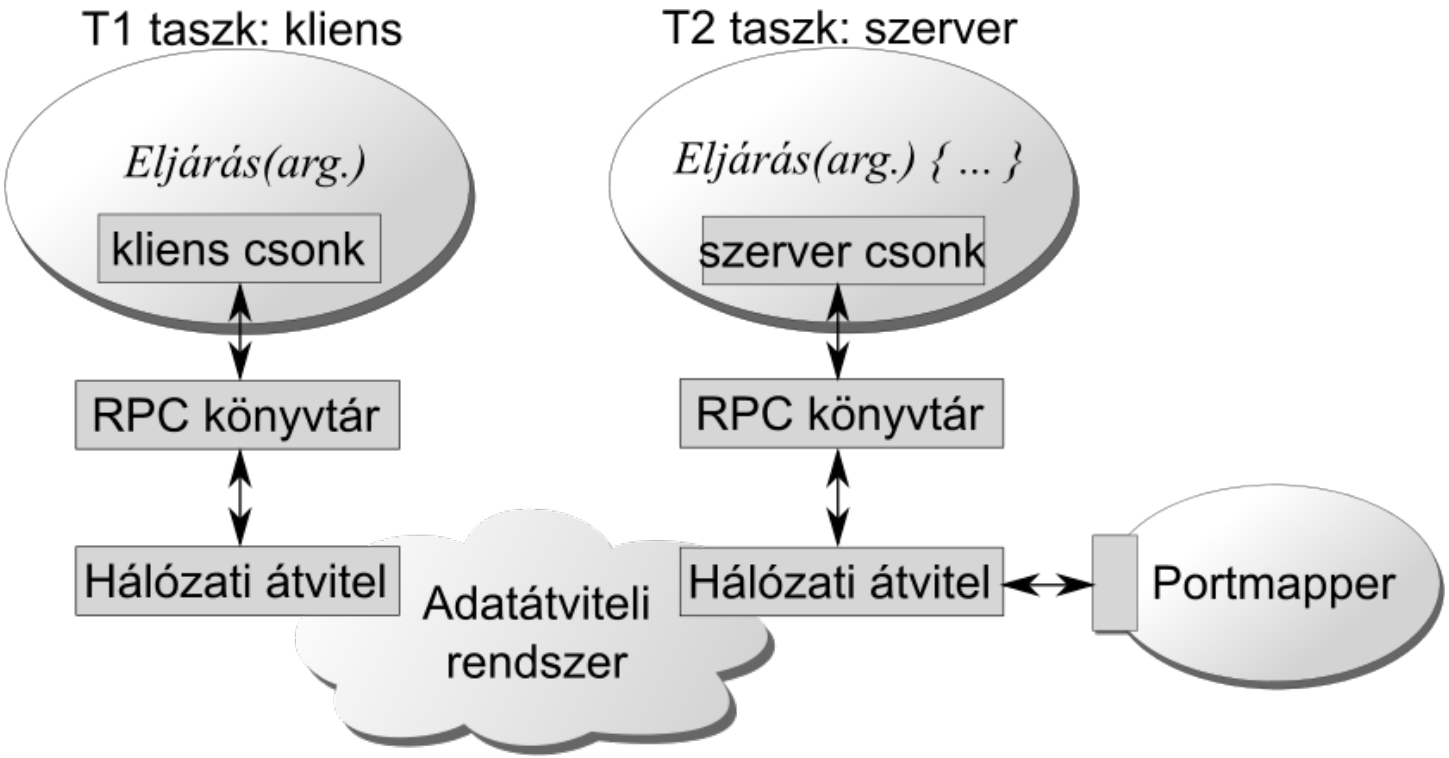
```
program DATE_PROG {
    version DATE_VERS {
        long BIN_DATE(void) = 1;     /* eljárás azon. = 1 */
        string STR_DATE(long) = 2;   /* eljárás azon. = 2 */
    } = 1;                           /* verziószám = 1 */
} = 0x31234567;                      /* program azon. = 0x31234567 */
```

- Kódgenerátor: `rpcgen`

  `rpcgen date.x` eredményei

  - `date.h`: adattípusok deklarációja
  - `date_clnt.c`: a kliens kódjában felhasználható date_... függvények
  - `date_srv.c`: a szerver date implementációját meghívó függvények
  - `date_xdr.c`: adatkezelő eljárások architektúrafüggetlen XDR formátumhoz

A ONC (Sun) RPC részletes felépítése és működése



Az RPC-alapú és ihletésű megoldások

- OS szolgáltatások
  - pl. hálózati fájlrendszerek
    NFS (Network File System)

- Elosztott alkalmazásfejlesztés (kitekintés)
  pl. **CORBA**, DCOM, Java RMI, .NET Remoting, D-Bus, XML-RPC, SOAP stb.

  - interfészleíró nyelv (interface description/definition language, IDL)
    - implementációs nyelvtől független
      de a nyelvi kötés definiált
    - széles körben használt
      OMG IDL, WSDL, Microsoft MIDL, Facebook/Apache Thrift, LibreOffice UNO
    - szabványokban is
      lásd W3C szabványok formális leírása, pl. DOM IDL

  - programkód generálása interfész leírásból
    - IDL leírás → forráskód
    - pl. CORBA IDL fordítók, IDL2Java, MS IDL fordítók stb.

Választás a kommunikációs lehetőségek között

- (Implementációs kötöttségek: programozási nyelv, környezet, stb.)
- A kommunikáció végpontjai szerint
  - számítógépen belül: mindegyik, RPC esetleg nem (nincs lokális portmapper)
  - számítógépek között: socket, RPC, hálózati diszkeken fájlon keresztül is
- A kommunikáció jellege
  - értesítés eseményekről: jelzések (SIGUSR1)
  - szinkronizálás: szemaforok
  - adatfolyam (pl.: csővezeték, socket) vs. diszkrét üzenetek (üzenetsorok)
  - üzenettípusok, szűrés? (üzenetsorok)
  - adatmennyiség (osztott memória: kicsi, csővezeték: közepes, socket: nagy)
- Teljesítmény
  - gyorsak: osztott memória, csővezeték, socket (PF_UNIX)
  - korlátos méret: osztott memória
- Kényelem
  - RPC, osztott memória (szemaforok?)

Programozási példák: http://beej.us/guide/bgipc/

A kommunikáció alapvető formái (összefoglalás)

- közös memórián keresztül
  - PRAM modell: sorrendezi (nem keveri) a kéréseket
  - hatékony
  - szinkronizáció szükséges
  - megvalósítás: szálak, SHMEM, mmap()

- üzenetváltás segítségével
  - Küld() és Fogad() műveletek
    - többféle címzés
    - szinkron és aszinkron működés
    - többféle adatátviteli szemantika
  - közvetítő komponens
    - késleltet, lassít
  - sokféle megvalósítás
    - jelzések: értesítés eseményekről
    - csővezetékek, üzenetsorok, hálózati kommunikáció: adatátvitel
    - távoli eljáráshívás: egy távoli taszk eljárásainak egyszerű meghívása