

Sistema-Deiak: Linux Kernel API



Kepa Bengoetxea
kepa.bengoetxea@ehu.es

Erreferentziak

- API: Application Programming Interface
 - en.wikipedia.org/wiki/Application_programming_interface
- POSIX : Portable Operating System Interface
 - en.wikipedia.org/wiki/POSIX
- C POSIX library:
 - en.wikipedia.org/wiki/C_POSIX_library
- Syscalls:
 - en.wikipedia.org/wiki/System_call

Sistema eragilea

- Ez dago Sistema Eragilearen(SE) definizio borobilik
- Objektiboki, SEa *hardware*tik(HW) gertuen dagoen *software*a da. Helburuak:

1- HWaren konplexutasunaz abstrakzioa egin⇒ Interfaze edo alegiazko makina, HW hutsa baino erabilgarriagoa

2- Sistema informatikoaren funtzionamendu zilegia ziurtatu
⇒ Baliabide guztien kudeaketa orekatua (PUZa, Memoria, S/I disp.)

Sistema eragilea

- Zer da SEa? Programadorearen ikuspuntu funtzionala: makinaren zehaztasunak alde batera utziz haren baliabideak erabiltzeko aukera eskaintzen duen errutina multzoa \Rightarrow alegiazko makina
- SEaren interfazearen osagaiak:
 - Sistema-deiak
 - Komando-interpretatzailea: testuzkoak, grafikoak...

Motibazioa

- SEek eskaintzen dituzten funtzioak aztertuko ditugu.
- **Sistema-deien interfazea** sistema eragilearen funtsezkotzat joko dugu hemendik aurrera, eta bera izango da **alegiazko makinaren funtzioak** definitzen dituenena. Beraz, **sistema-deien multzoak** definitzen du sistema eragilearen oinarritzko interfazea, eta berau zehazten du **iturburu-lengoiaren mailan makinaren arteko bateragarritasuna**.

Motibazioa

- Erabiltzailearentzat, sistema informatikoa aplikazioak egikaritzeko plataforma bat da.
- Programak garatzeko orduan HWaren kontrol zehatza eraman beharko balu programatzaileak programatzea ikaratzeko moduko lana izango litzateke.

Motibazioa

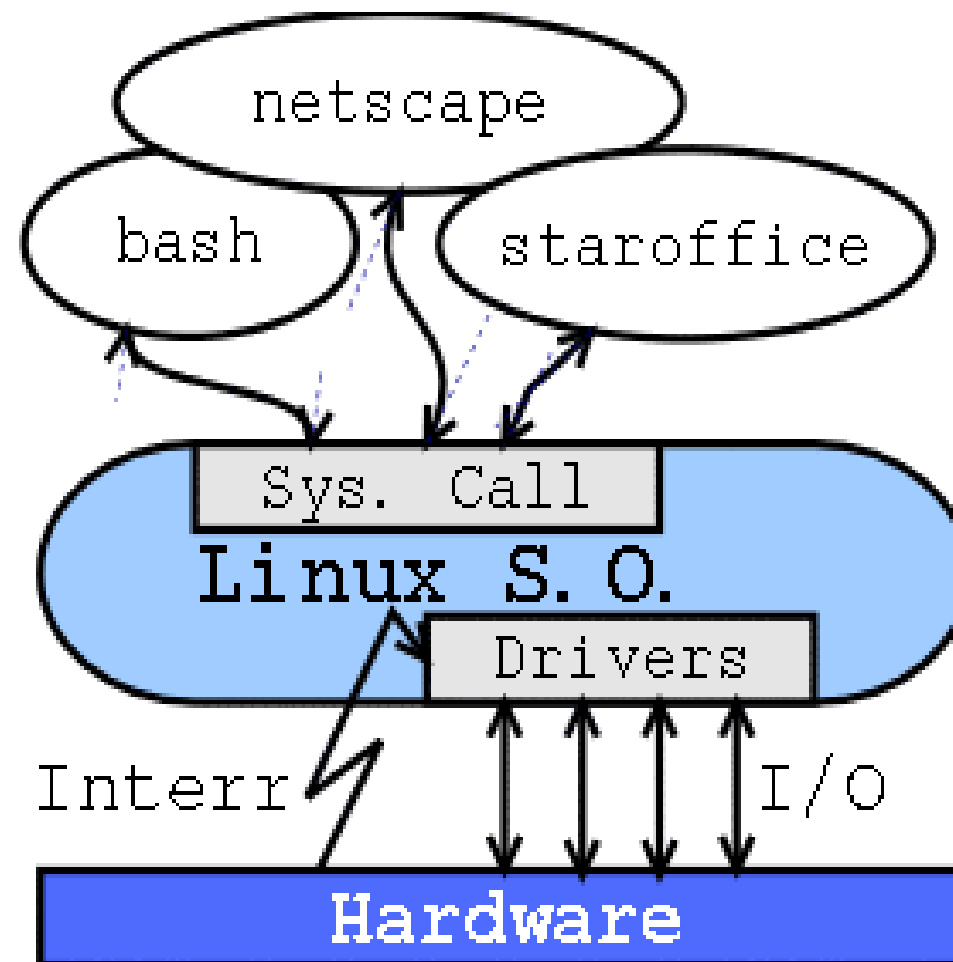
- Programazioa errazteko SEak sistema-deiak eta berarekin batera utilitate-programa multzo zabal bat eskaintzen du.
- Utilitate-programek programatzaileari laguntzeko zerbitzu anitz eskaintzen dituzte: testu-editoreak, konpiladoreak, araztaileak (debuggers), liburutegiak, etab. Orokorrean ez dira sistema eragilearen parteak, beraren gainean eraikita baitaude berarekin banatu arren.

Sistema Deiak: Linux Kernel API

- Sistema Deia (ingeleraz system call) aplikazio batek Sistema Eragileari zerbitzu bat eskatzeko erabiltzen da.
- Sistema Eragileak sistema-deien multzo bat eskaintzen du (linux-eko atentzio edo egoiliar errutina multzoa, linux Kernel API edo POSIX* API liburutegi bezala ere ezagutzen dena). Programazioa eta ordenagailuaren erabilera errazten dutenak. Hardware baliabideak: PUZ, Memoria, Diskoa eta abarrak kudeatuz, eta horri esker, ordenagailuari etekin handiagoa ateraz.

*POSIX:Portable Operating System Interface; X UNIX-etik dator.

Sistema Deiak: Linux Kernel API



Sistema Deiak: Linux Kernel API

- GNU C Library, **glibc** bezala ezagutzen dena, GNU-ko C-ko liburutegi estandarra da. Linux banaketan **libc6** bezala ezagutzen da.
- Instalatzeko: `sudo apt-get install libc6-dev`
- Libc6 liburutegi honetan:
 - Linux-eko Sistema Deiak
 - C-ko liburutegi estandarra dago
- Linux 4.x ia 300 sistema dei baino gehiago daude. Erabiliak: `open`, `read`, `write`, `close`, `wait`, `exec`, `fork`, `exit` eta `kill`. “`unistd.h`” aurkitu dezakegu euren erazagupena.

Sistema Deiak: man

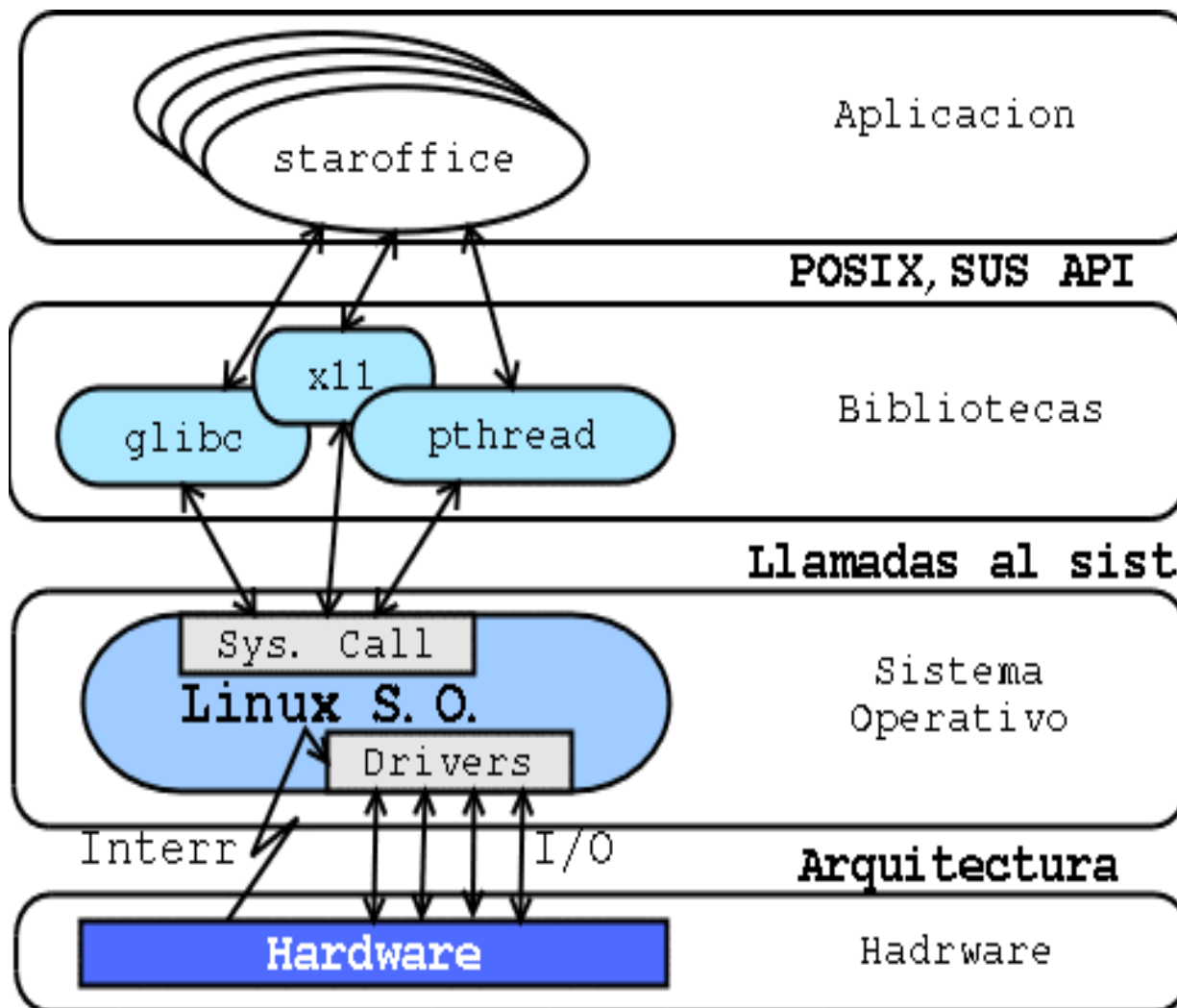
Orain arte 1 eta 3 atalak landu ditugu, orain 2 atala landuko dugu.

Man atalak

Atala Deskribapena

- 1 Exekutatzeko programak eta shell-en aginduak
- 2 Sistema-deiak(kernelak zerbitzatutakoak)
- 3 C Liburutegiko oinarrizko funtzio deiak
- 4 Artxibo bereziak
- 5 Artxibo formatuak eta konbentzioak
- 6 Jokoak
- 7 Makro paketeak eta konbentzioak
- 8 Rootengandik exekutatzeko aginduak

Sistema Deiak: Linux Kernel API



Sistema Deiak: Linux Kernel API

- Ze zerbitzu eskaintzen ditu kernelak?
 - Prozesuen kudeaketarako: fork, exec, setuid, getuid...
 - Fitxategien kudeaketarako: read, write, open, close, mkdir ...
 - Memoriaren kudeaketarako: mmap, sbrk, munmap, mlock ...
 - Sare zerbitzuen kudeaketarako: sethostname, setdomainname ...
 - Seinaleen kudeaketarako: sigaction, sigsuspend ...

Sistema Deiak: Linux Kernel API

Kodea emanda: C liburutegiak erabiliz

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

```
int main(){int i;
```

```
for (i=0;i<=5;i++)
```

```
{printf("El valor de i es %d",i);}
```

```
return 0;}
```

Nola izango litzateke sistema-deiak edo linuxen APIak(Application Programming Interface) zuzenean erabiliz?

Sistema Deiak: Linux Kernel API

“write” bezalako sistema-deiak nola funtzionatzen duten ikusteko man erabili daiteke:

man 2 write (non 2 sistema-deien atala da)

SINOPSIA: `#include <unistd.h>`

`ssize_t write(int fd, const void *buf, size_t num);`

DESKRIBAPENA: `write-k`, `buf-en` hasitako buferretik, `fd` fitxategiko deskribatzaileak adierazitako fitxategian `num` bytes-etaraino idazten du.

ITZULITAKO BALOREA: (>0 bada, idatzitako bytes kopurua; 0 bada, ez dela ezer idatzi; -1 bada errorea eta `errno` aldagaian erroreaki dagokion balioa jarriko da).

Sistema Deiak: Linux Kernel API

man 3 sprintf

SINOPSIA: `#include <stdio.h>`

`int sprintf(char *str, const char *format, ...);`

DESKRIBAPENA: str karaktere kate batean, parametrotzat pasatzen zaiona formatuarekin idazten du.

ITZULITAKO BALOREA: Funtzioak bihurtutako karaktere zenbakia itzultzen du

Sistema Deiak: Linux Kernel API

Kodea emanda: Kernelen sistema-deiak erabiliz

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

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

```
#include <string.h>
```

```
int main() {int i;char cad[100];
```

```
for (i=0;i<=5;i++)
```

```
    {sprintf(cad,"El valor de i es %d",i);
```

```
    write(1,cad,strlen(cad));
```

```
}
```

```
return 0;}
```

Sistema Deiaik: Linux Kernel API

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

```
int main(void)
```

```
{ printf("hello"); return 0; }
```

```
$gcc -o hello hello.c -static
```

```
$strace ./hello
```

```
stra
```

#execve() ejecuta el programa indicado por filename

execve("./hello", ["/.hello"], [/* 29 vars */]) = 0

write(1, "hello", 5hello) = 5

exit_group(0) = ?

Sistema Deiak: Erroreak kudeatu

Sistema deiek huts egiten dutenean, -1 balioa itzultzen dute, eta akatsaren zenbakia errno aldagaian gordetzen dute. perror() funtzioaren bitartez errno zenbakiari dagokion errore mezua pantailaratu dezakegu. man 3 perror

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

```
void perror(const char *s);
```

DESCRIBAPENA: perror() errutinak, errore estandar irteerara doan mezu bat sortzen du, sistema-dei batean zehar edo zenbait liburutegi funtziotan aurkitutako azken errorea deskribatuz.

Sistema Deiak: Erroreak kudeatu

/* erroreak.c * 53 lehen erroreak zerrendatzen ditu:

```
#include <stdio.h> //stderr, fprintf, perror
```

```
#include <errno.h> //errno
```

```
int main(){
```

```
int i;
```

```
for (i=0;i<=53;i++)
```

```
{fprintf(stderr,"%3d",i);
```

```
errno=i;
```

```
perror("errorea");
```

```
}
```

```
return 0;
```

Sistema Deiak: Erroreak kudeatu

0errorrea: Success

1errorrea: Operation not permitted

2errorrea: No such file or directory

3errorrea: No such process

4errorrea: Interrupted system call

5errorrea: Input/output error

....

Sistema Deiak: Erroreak kudeatu

```
#include <sys/types.h> //open
#include <sys/stat.h> //open
#include <fcntl.h> //open
#include <stdio.h> //printf, perror
#include <stdlib.h> //exit
#include <unistd.h> //close

int main(int argc, char *argv[])
{
    int fd; fd=open(argv[1], O_RDWR);
```

```
    if (fd==-1)
    {perror("open ");
    exit(-1);
    }

    printf( "Irekitako fitxategiak,
honako deskribatzailea du %d.\n",
fd);

    close( fd );

    return 0;}
```

Sistema Deiak: Fitxategiak kudeatzeko

fd = **open** (file, modo apertura r|w...[, permisos]) – idazteko/irakurtzeko fitxategia zabaldu

s = **close** (fd) – fitxategi bat itxi

n = **read** (fd, buffer, nbytes) – fitxategitik irakurritako nbyte buffer-ean gorde.

n = **write** (fd, buffer, nbytes) – bufferreko nbyte idatzi fitxategi batean.

pos = **lseek** (fd, offset, whence) – fitxategiaren erakuslea mugitu.

s = **stat** (name, &buf) – fitxategi baten izenarekin, i-nodo informazioa lortzeko

n = **chmod**(file, permisos)- fitxategi baten baimenak aldatzeko

Sistema Deiak: Fitxategiak kudeatzeko

- Norbaitek C liburutegiko oinarrizko funtzioak gustuko ez baditu, linuxen sistema-deiak erabil ditzake. Adb:
 - fwrite erabili beharrean
 - Zure fwrite sortu dezakezu ,write sistema-deia erabiliz

Sistema deiak: Memoria babestua

- Sistema deia SEari zerbitzu bat eskatzeko erabiltzen da. Sistema deiaren kodea exekutatzeko software eten bat gertatzen da.
- x86 konputagailuetan eten sistema mota bi daude:
 - Hardware etenak
 - Software etenak edo ezezpizioak

Sistema deiak: Memoria babestua

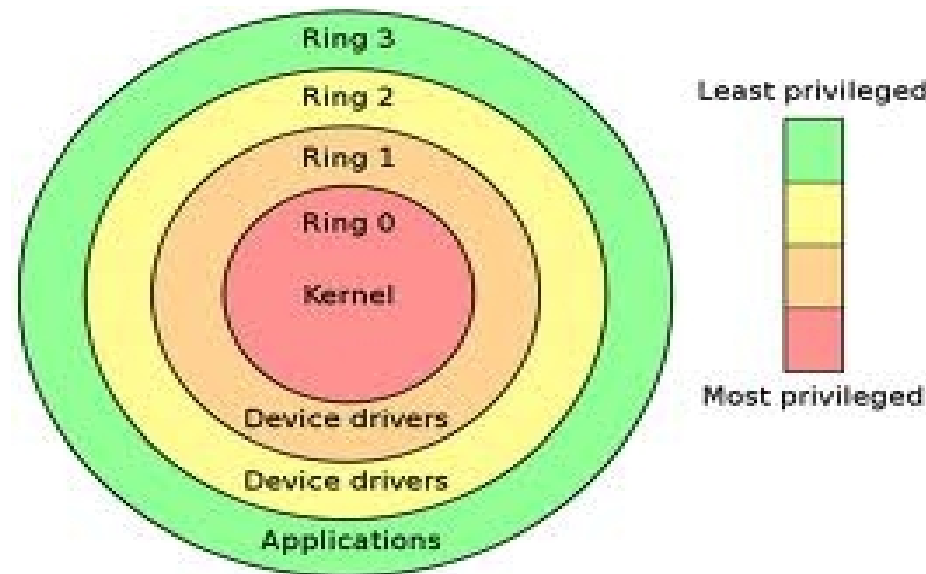
- **Hardware etenak:** Gailuak eta PUZa edo PUZak euren artean komunikatzeko erabiltzen dira. Ez daukate zer ikusirik momentuan exekutatzen dagoan prozesuarekin. Hau da, gailu batek zeozero behar duela adierazteko PUZari IRQ (“Interrupt Request”) bat bidaltzen dio, eta horrela PUZak egiten ari dena usten du, gailuaren eskaera betearazteko. Hardware etenen mekanismoak PUZa S/Iko gailuen egoerataz etengabe galdezka ibiltea ekiditzen du (Gailuak zeozero nahi izanez gero eten bat bidaltzen baitiote PUZari)

Sistema deiak: Memoria babestua

- Software etenak
 - sistemaren baliabideak erabiltzeko kernelak eskeintzen dituen zerbitzuei edo sistema deiei deitzeko erabiltzen dira.
 - Baita gure prozesuak sortutako ezezpizioei erantzuteko, hau da, divide-by-zero, Arithmetic Logic Unit (ALU) sortua edo illegal address, Memory Management Unit (MMU) sortua.

Sistema deiak: Memoria babestua

- x86 ordenagailuak kodea exekutatzeke 4 modu ezberdin daukate: zero, bat, bi eta hiru eraztunetan. Linuxen “gure kodea” hirugarren eraztunean exekutatzen da eta Hardware edo Software eten bat sortzen denean “kernel kodea” zero eraztunean exekutatzen da. Sistema deiari deitzaerakoan hirutik zerora salto egiten dugu. Zero eraztunean kodea edozein helbide atzitu dezake. Hiru eraztunean kodeak soilik gure aplikazioaren memoria gunera soilik atzitzeko baimena du.



Sistema deiak: Memoria babestua

Linuxeko zerbitzu bati edo atentzio edo egoiliar errutinei deitzerakoan, linuxek **indirekzio teknika** erabiltzen du. Teknika honek, **Etenaldi Deskriptoreen Taula (EDT)** bat erabiltzen du. **EDT** honetan ze etenaldiari ze atentzio errutina dagokion adierazten da. EDTaren sarrerak gutxitzeko, sarrera/irteerako errutinak, soft erroreko errutinak, eta abar... ,sakabangatze errutinatan taldekatzen dira. Horregatik, eten bat sortzen denean sakabangatze errutina bati deitzen zaio, parametrotzat ze atentzio errutinari deitu behar dion pasatuz.

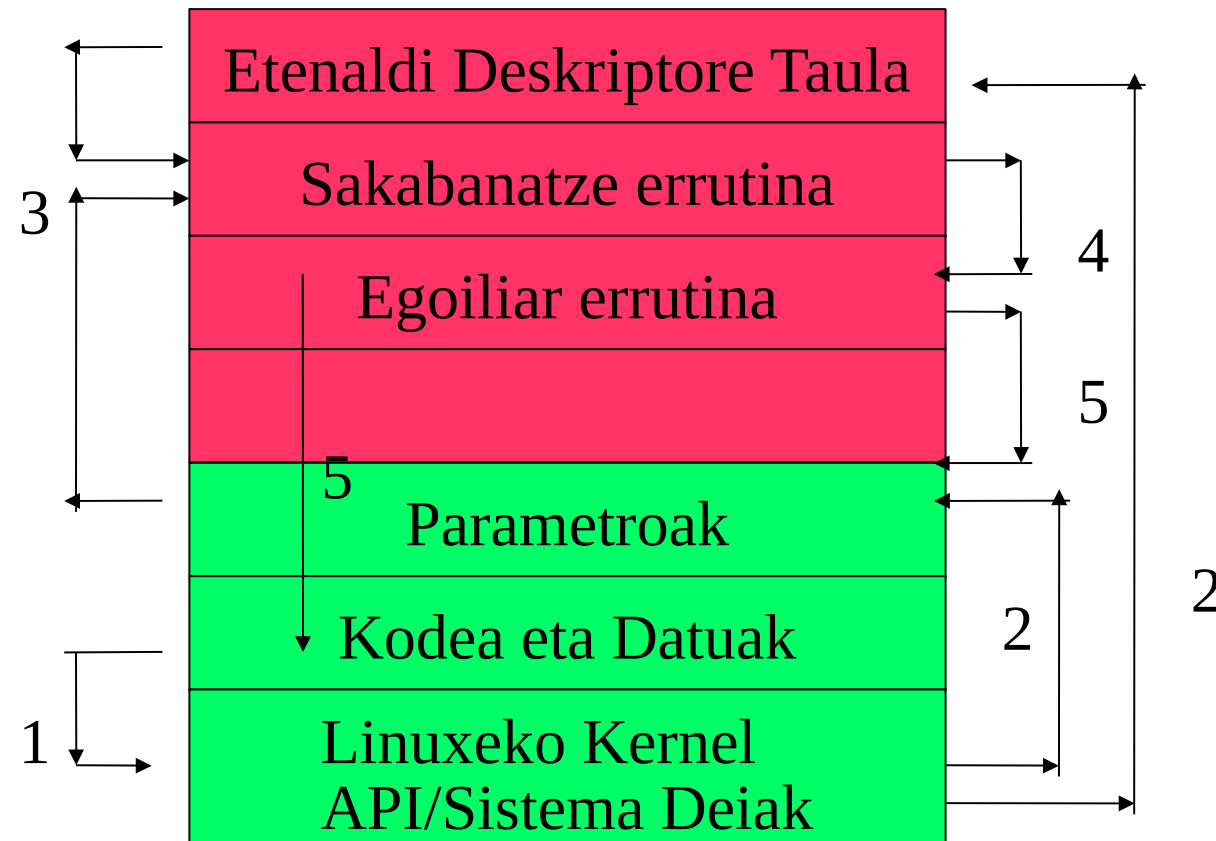
Sistema deiak: Memoria babestua

Sakabanatze errutinatik dagokion **atentzio errutinari** deitzen zaio.

Atentzio egoiliar errutinaren exekuzioa **bukatu ondoren, kontrola erabiltzailearen programari ematen zaio, emaitza parametroetan utziz.**



Sistema-Deiak: Memoria babestua



Sistema deiak: Memoria babestua

- EDT, beheko memorian kokatzen da eta 1024 byte ditu, hori dela eta, 4 byteko 256 sarrera soilik eduki dezake, sarrera bakoitza errutina egoiliar edo sakabاناتze errutina baten helbidea izango du.
- EDT honen lehenengo betebeharra BIOS zerbitzuen errutinen helbideak izatea da. Baina hauek ez dute sarrera asko betetzen. Beste gainerako sarrerak sistema zerbitzuekin betetzen dira.

Sistema deiak: Memoria babestua

- EDTn dauden BIOSeko sarrerak BIOSaren memoriara apuntatzen dute (helbide hauek berdinak dira edozein SEarentzako (Linux, Windows, Macintosh, ...) eta gainerako sarrerak memoria RAMean dauden SEaren errutinei apuntatzen diete.
- Sistema deiek 80. etena erabiltzen dute, sakabanatze errutinari deituz parametro baten bidez pasatuz, zein errutina egoiliar exekutatu nahi duen. Errutina hauek “modu kernelean” edo “zero eraztunean” exekutatzen dira.

Sistema deiak: Memoria babestua

- Sistema deien bitartez bakarrik, sistema eragileen errutinak erabili ditzakegu. Sistema honekin memoria, erabiltzaile programetatik babesten dugu.
- Erabiltzaile programa batek, sistema eragilearen errutina baten helbidean sartuz gero eta bertan kodea moldatuz gero, beste programa guztiak ez ziran ondo ibiliko = **birusa**. Hau dela eta, erabiltzaile programa bat ezin da sistema eragilearen memorian zuzenean sartu. Zuzenean sar daitekeen leku bakarra bere datu eta instrukzio gunera da.

Sistema deiak: Memoria babestua

- Sistema deia bat burutzerakoan: erabiltzaile modutik(3. eraztunetik), gainbegirale modura(1. eraztunera), aldatzen da. SEaren errutina exekutatu ondoren, erabiltzaile modura(3. eraztunera) bueltatuko da.
- Erabiltzaile moduan(kodea 3. eraztunean egikaritzen) dagoenean, hardware mekanismoaren bidez kontrolatzen da memoriaren sarbidea. Gainbegirale moduan, berriz, hardware mekanismoa desaktibatzen da.

Sistema deiak: Memoria babestua

- Prozedura hau, jatetxe baten jantoki batena gogorarazten dit. Jantokian, bezero guztiek zerbitzariari nahi dutena eskatzen diote, baina ez dira inoiz sukaldean sartzen. Zerbitzariak, sukaldetik pasa ondoren, bezeroak eskatutako platerra ekarriko du. Jankideak ezingo dute sukaldea hondatu, ezin direlako honetan sartu.

Sistema deiak: Memoria babestua

Laburbilduz

- Erabiltzaile kodea:
 - Ezin da ordenagailuaren baliabideetara zuzenean sartu
- Kernel kodea :
 - Sistema osorako sarbidea, ordenagailuaren baliabide guztiak kudeatuz
 - Edukia: gailu kontrolatzaileak, memoria gestioa, fitxategi sistema,...

Sistema deien mekanismoa ikusgai

Ze zerbitzu eskaintzen ditu kernelak?

\$uname -r ->4.4.0-116-generic bertsioan, /usr/include/asm-generic/unistd.h artxiboan zenbakitutako sistema deiak existitzen dira.

Fitxategi honetan, funtzioen kodea zein fitxategian dagoen ere esaten da:

```
/* fs/read_write.c */
```

```
#define __NR3264_lseek 62
```

```
__SC_3264(__NR3264_lseek, sys_llseek, sys_lseek)
```

```
#define __NR_read 63
```

```
__SYSCALL(__NR_read, sys_read)
```

```
#define __NR_write 64
```

```
SYSCALL( NR write, sys write)
```

Sistema deien mekanismoa ikusgai

- Linux-eko kodea ikusi ahal dugu? Bai. Nola?

`sudo apt-get install linux-source` -> linux-source ya está en su versión más reciente (4.4.0.116.122)

- aurreko komandoarekin /usr/src katalogora linux-en iturriak jaisten ditugu, hau da, `linux-source-4.4.0.tar.bz2` -> `linux-source-4.4.0/linux-source-4.4.0.tar.bz2` fitxategia.
- Mugitu hurrengo katalogora: `$ cd /usr/src`
- Hurrengo komandoarekin, urrats batean, fitxategia deskonprimitu eta paketea zabalduko dugu:
- `$ sudo tar jxvf linux-source-4.4.0.tar.bz2`

Sistema deien mekanismoa ikusgai

aplikazioa.c void main(){...fclose..}

Erabiltzaile
Modua

fclose

C-ko APIak edo Liburutegiak(/lib/x86_64-linux-gnu/libc.so.6):{ ...close..}

close

Linux-eko APIak edo liburutegiak

Syscall edo 80. etena

Eten Bektorea>sakabangatze errutina sys_call (/usr/src/linux-source-4.4.0/arch/x86/entry/entry_64.S)

Kernel
Modua

call *sys_call_table(,%rax,8)
* Register setup:
* rax system call number

Sistema deien mekanismoa ikusgai

```
entry_SYSCALL_64_fastpath:  
    call *sys_call_table(,%rax,8)
```

(rax=57->sys_close) beren kodea

/usr/src/linux-source-4.4.0/fs/open.c dago hemen

/usr/include/asm-generic/unistd.h esaten den moduan.

Kernel
Modua

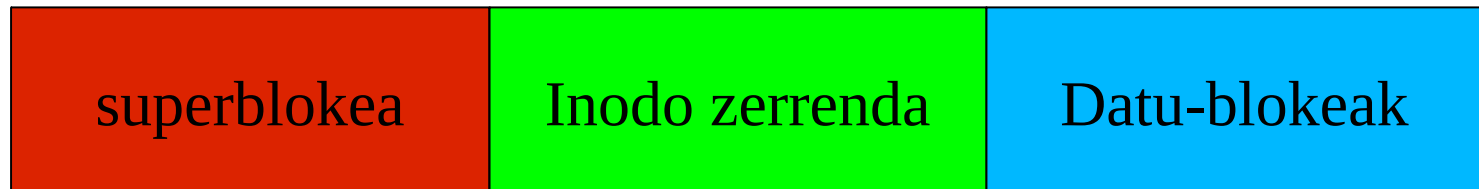
Fitxategi Sistema -> Hardware

Fitxategi Sistemaren Egitura

Kepa Bengoetxea
(LSI Saila)

UNIX SysV fitxategi-sistemaren egitura

partizioa



UNIX SysV fitxategi-sistemaren egitura

Superblokea: fitxategi sistemaren tamaina, fitxategi kopuru maximoa, erabilgarri dagoen lekua, inode zerrendaren tamaina, libre dagoen hurrengo nodoaren indizea, libre dagoen hurrengo blokearen indizea, *dirty flag*....

Superblokearen informazioa bistaratzeko:

```
sudo /sbin/dumpe2fs -h /dev/sda5
```

UNIX SysV fitxategi-sistemaren egitura

Inodo zerrenda: fitxategi edo karpeta bakoitzeko sarrera bat izango duen datu egitura . Inodo bakoitzean, fitxategiaren jabea, baimenak, erabilitako blokeen helbideak, esteka kopurua, etab. agertuko dira

Datu blokeak: fitxategien edukia, zatika, hemen gordeko da.

UNIX SysV fitxategi-sistemaren egitura

Funtzionamendua:

- prozesu batek fitxategi bat idatzi nahi duenean, **superblokea** irakurri eta eguneratu egin beharko du. Baita **inodo zerrenda** eguneratu ere.
- kernelak **buffer-cache** bat erabiliko du S/I eragiketak azkartzeko. Gainera, superbloke eta inodo zerrendaren kopia bana **memorian** gordeko du baita ere.
- *syncer* izeneko *daemon*-a, **aldi**ro, memoria dagoen egituren gorde diren aldaketak diska gogorrera kopiatuko ditu.

UNIX SysV fitxategi-sistemaren egitura

inodoak

* Linuxen, **fitxategi bakoitzak inodo bat** esleituta dauka

* Prozesu batek fitxategi bat irakurri nahi duenean, bere inodoa bilatu behar du.

```
Ls -i
```

Inodo batek gordetzen duen informazioa (besteak beste):

- + Jabearen UID eta GID zenbakiak
- + Fitxategi mota (arruntak, karpetak, dispositiboak...)
- + Baimenak
- + Atzipen eta azken aldaketaren datak
- + Fitxategiak duen esteka kopurua
- + Tamaina...

```
stat
```

UNIX SysV fitxategi-sistemaren egitura

inodoak

- * iNodo Zerrenda **superblokearen ondoan** dauden bloketan kokatzen da
- * Sistema abiatzerakoan, kernelak disko gogorreko inodo zerrenda irakurri eta memorian kopia bat kargatuko du. Kopia honi **inodo-taula** deritzo.
- * Fitxategi sistemak fitxategietan aldaketak egiten dituenean, inodo taulan egingo ditu (eta ez zuzenean inodo zerrendan). Horrela, fitxategien gainean egindako eragiketak eraginkorragoak izango dira.

UNIX SysV fitxategi-sistemaren egitura

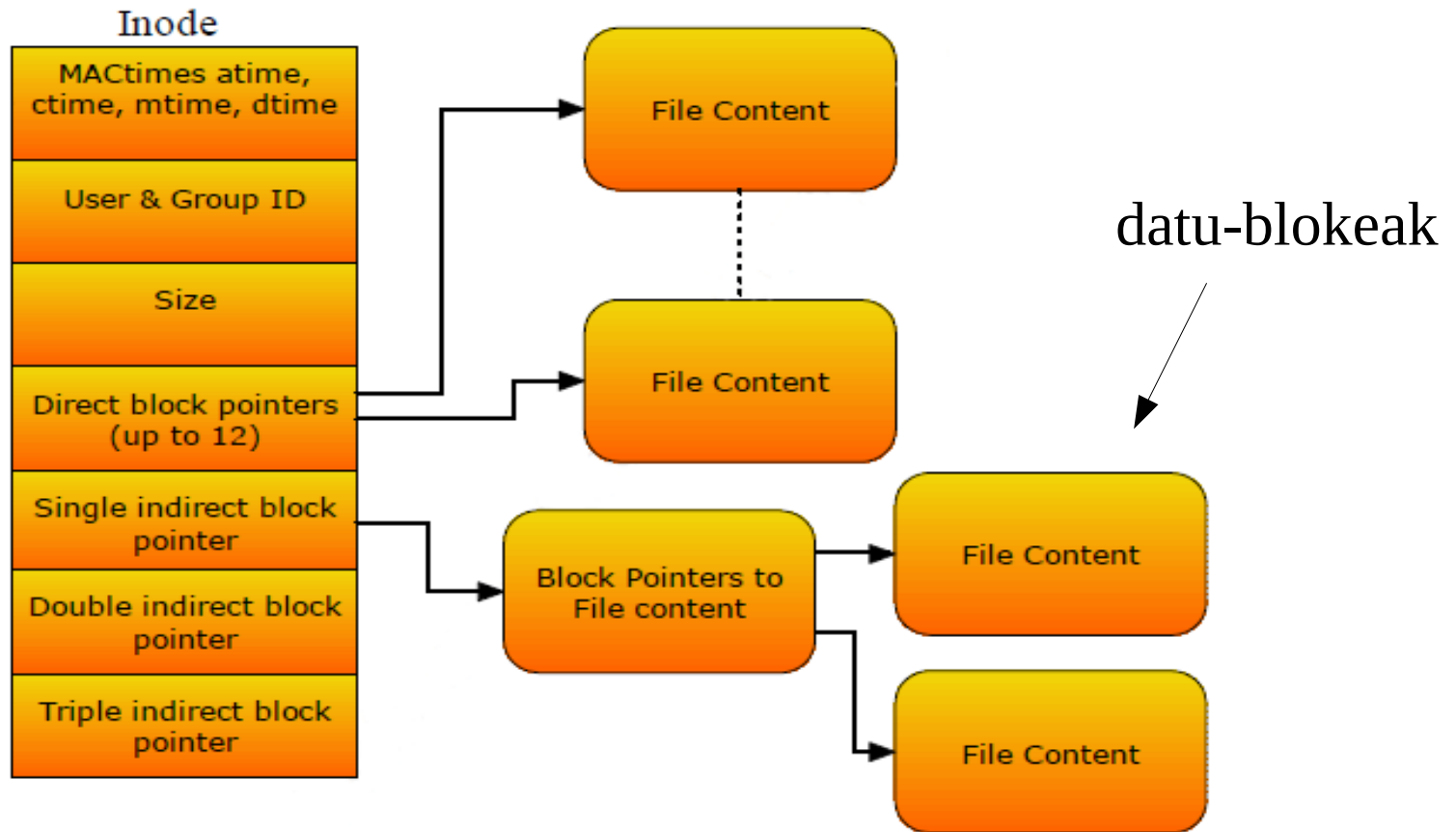
datu-blokeak

Funtzionamendua:

- * Fitxategi batek disko gogorrean okupatuko duen datu bloke kopurua bere tamainaren araberrakoa izango da. Alegia, zenbat eta handiagoa izan orduan eta datu-bloke gehiago izango ditu.
- * Inodoak fitxategi baten datu-bloke guztiak kudeatu behar baditu, zein izango da inodo baten tamaina?

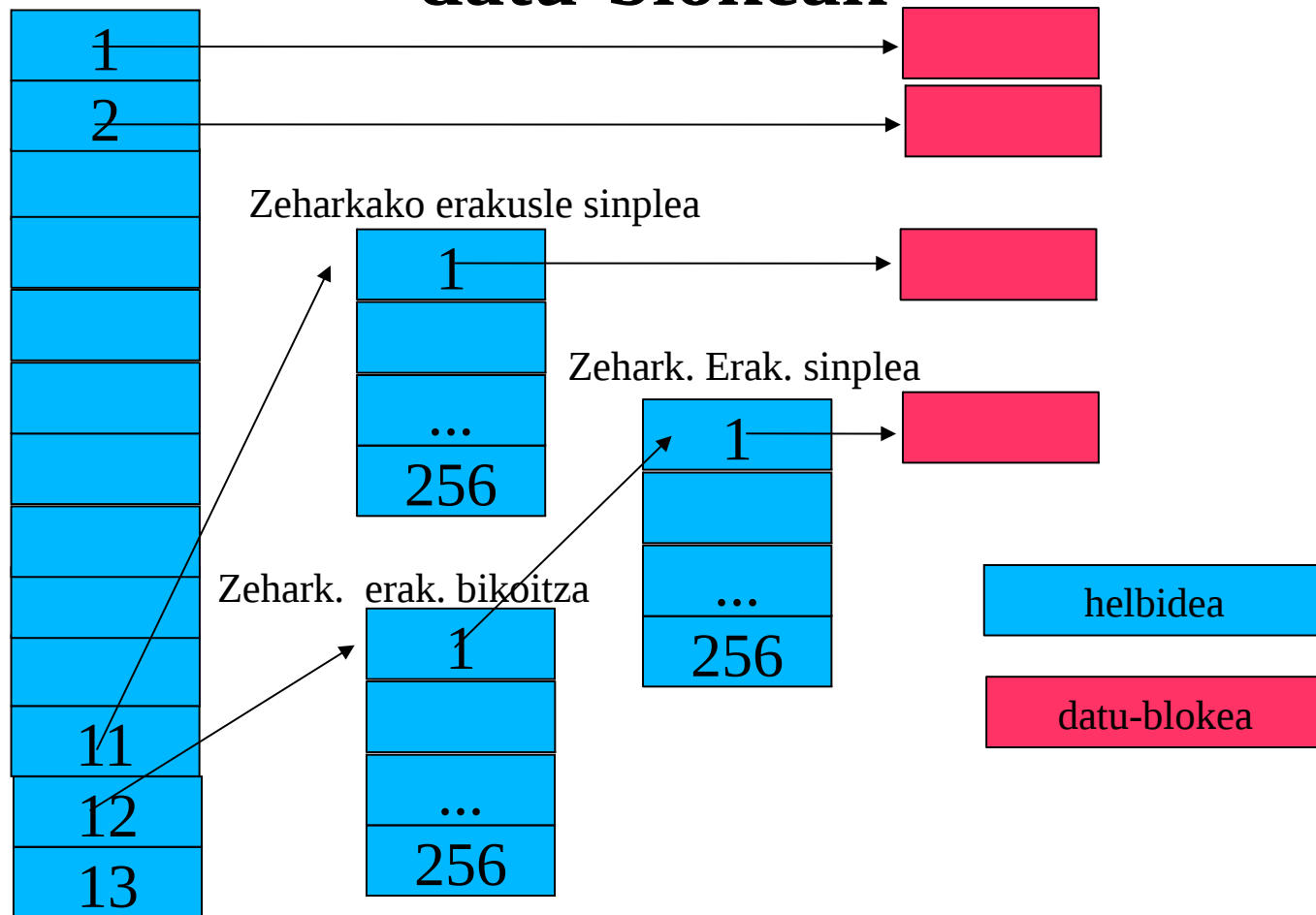
UNIX SysV fitxategi-sistemaren egitura

inodoak



UNIX SysV fitxategi-sistemaren egitura

datu-blokeak



Erantzuna: inodo batean datu-blokeen helbideak kudeatzeko, zuzeneko eta zeharkako erakusleak erabiltzen dira.

UNIX SysV fitxategi-sistemaren egitura

datu-blokeak

Sarrera mota	Atzigarriak diren bloke kopurua	Guztira, datu zatien tamaina
10 atzipen zuzeneko erak.	10 datu-bloke	10 KB
1 zeharkako erak. Sinplea	256 datu-bloke	256 KB
1 zeharkako erak. Bikoitza	$256 \times 256 = 65536$ datu-bloke	64 MB
1 zeharkako erak. Hirukoitza	$256 \times 256 \times 256 = 16.777.216$ datu-bloke	16 GB

- Horrelako egitura bat erabiliz eta KB bateko blokea erabiliz gero, fitxategi baten tamaina maximoa 16GBekoa izango da.
- Blokearen tamaina handituz gero, 16GB baino gehiago lortuko genuke baina **barne-fragmentazioaren** ordainean.
- Bloke baten tamaina fitxategi sistema sortzerakoan ezartzen da (1, 2 edo 4 KBekoa izanik)

Ext4 Fitxategi-Sistema . ACL

/etc/fstab fitxategian ere ACL aukera ezarri behar da

/dev/hda5	/home	ext3	rw, acl	1 2
-----------	-------	------	----------------	-----

#umount /home

#mount /home

#mount -l

/dev/hda2 on / type ext3 (rw) [/]

/dev/hda1 on /boot type ext3 (rw) [/boot]

/dev/hda5 on /home type ext3 (rw,acl) [/home]

Ext4 Fitxategi-Sistema . ACL

sudo apt-get install acl

`dpkg -l acl`

acl 2.2.42-1ubuntu1 Access control list utilities

`dpkg -L acl`

`/usr/bin/getfacl -> get file access control lists`

`/usr/bin/setfacl -> set file access control lists`

`/usr/bin/chacl -> change the access control list of a file or director`

...

Ext4 Fitxategi-Sistema . ACL

```
[tristan]$ ls -l pizza
```

```
-rw-r--r-- 1 tristan tristan 19936 May 28 16:59 pizza
```

```
[tristan]$ getfacl pizza
```

```
# file: pizza
```

```
# owner: tristan
```

```
# group: tristan
```

```
user::rw-
```

```
group::r--
```

```
other::r--
```

Prozesuen kontrola eta kudeaketa

Multiprogramazioa

Kepa Bengoetxea

Multiprogramazioa

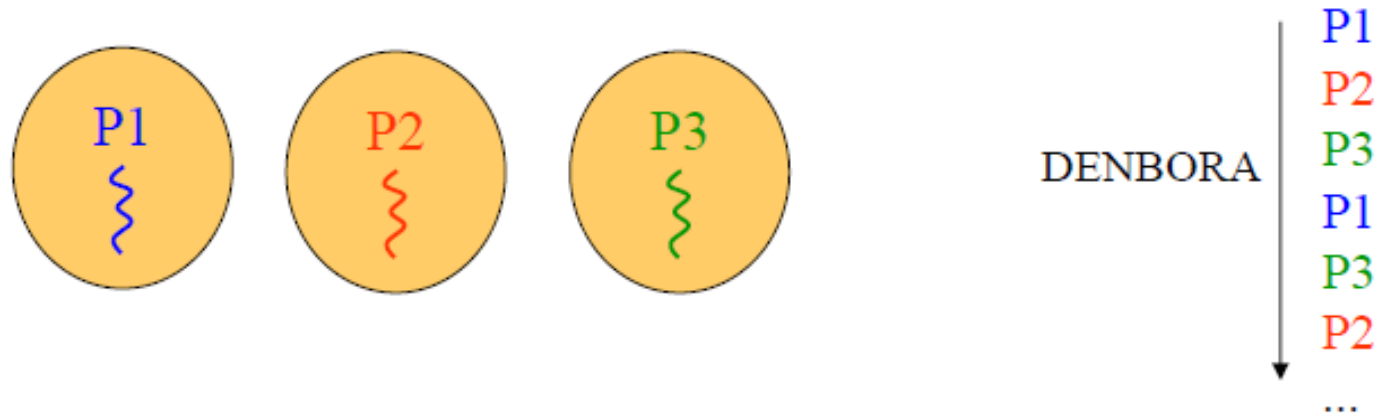
Motibazioa

- Prozesadorearen eta S/Iko dispositiboen arteko abiadura aldea.
- Prozesadorearen geldituak aprobetxatzeko mekanismoak.

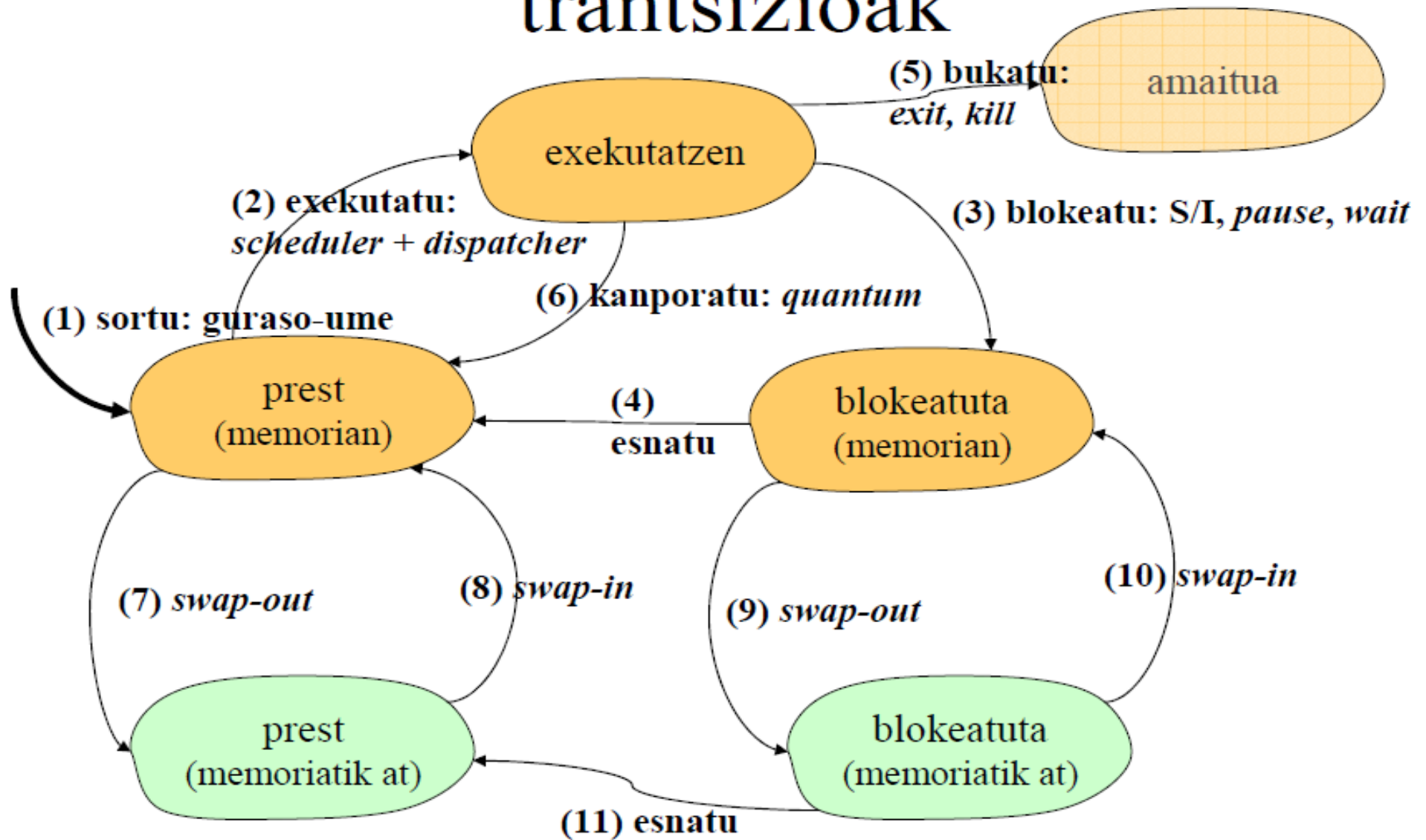
Soluzioa: konkurrentzia programen artean

- > Memoria nahikoa badago programa bat baino gehiago kargatu.
- > Exekututzen ari den programa S/I baten zain gelditzen denean, beste programaren bat prest badago, exekuta dadila.

MULTIPROGRAMAZIOA: Programa bat baino
gehiago memoria bere kodearen exekuzioa denboran
txandakatuz, denak SEaren kontrolpean



Prozesuen egoerak eta trantsizioak



Prozesuen egoerak eta trantsizioak

(1) Sortu

NOIZ: Sistema-dei berezi baten exekuzioa (fork)

EGINBEHARRA:

- * Identifikadore bat esleitu.
- * Programa memorian kargatu
- * Testuinguru informazioa hasieratu
- * Guraso prozesua – Ume prozesua
- * Huts egin dezake, adibidez programa ez bada aurkitzen, behar beste memoriarik ez badago, ...

Prozesuen egoerak eta trantsizioak

(2) Exekutatu

NOIZ: Exekutatzen ari den prozesua blokeatzen denean, bukatzen duenean edota **quantum**-a bukatzen zaionean, beste batek hartuko du bere lekua prozesadorean.

EGINBEHARRA:

* **SCHEDULER(planifikatzailea)**: SEaren funtzioa, exekutatuko den prozesua aukeratzeko duena.

Lehenetasuna kontuan hartuta.

* **DISPATCHER(jaurtitzaile)**: aukeratutako prozesua martxan jartzen arduratzen den SEaren funtzioa (testuinguru informazioa ezartzen du).

* Ez badago egikaritzeko prozesurik, prozesu NULUA exekutatu egiten da.

Prozesuen egoerak eta trantsizioak

(3) Blokeatu

NOIZ:

Sistema-deia blokeagarri bat egitea:

- * S/I
- * Denbora itxoite bat
- * Beste prozesu batekin sinkronizatzea

...

EGINBEHARRA:

Prozesua blokeatuz gero, SEak prozesuaren testuinguru informazioa gorde behar du eta beste prozesu bat exekutatzeko jarri behar du (2).

Prozesuen egoerak eta trantsizioak

(4) Esnatu

NOIZ:

- * Itxaroten zegoan gertaeraren bukaera:
 - * S/I eragiketa baten bukaera.
 - * timer edo alarma baten bukaera
 - * seinale baten etorrera, beste prozesu batekin duen, sinkronizazio baten bukaera.

EGINBEHARRA:

Esnatzen den prozesua prest egoeran ipini.

Esnatzen den prozesua exekutatzera pasatzeko aukera eman daiteke.

Prozesuen egoerak eta trantsizioak

(5) Bukatu

NOIZ:

Bukatzeko sistema-deia (exit).
Beste prozesu bat bukarazteko sistema-deia jaso (kill).

EGINBEHARRA:

SEak prozesua erabiltzen ari zen baliabideak askatu behar ditu.

Semeak aitarentzako prozesuen taulan beren irteera egoera utziko du. Semeak prozesuen taulan duen sarrera, bere aitak jaso arte egongo da. Horregatik prozesu semea zombie egoeran geratuko da, aita beren egoera jaso arte.

Prozesuen egoerak eta trantsizioak

(5) Bukatu

Gurasoa ume baten zain gelditzen denean (**wait**), Gurasoak ume baten bukaera kodea jasoko du, eta SEak ume horren informazioa ezabatuko du. Umeak baditu eta inork ez badu bukatu, orduan gurasoa blokeatu egingo da ume bat bukatu arte. Aita semeak baino arinago amaituz gero, init prozesuak semeen aitatasuna hartuko du.

Testuinguru-aldaketa

Prozesu bat exekutatzen ari denean, sistema **prozesuaren-testuinguruan** exekutatzen ari dela esaten dugu.

Prozesua = testuinguruaren informazioa + programa (aginduak eta datuak)

Testuinguru-informazioa: Prozesuen identifikadorea, Kanal-taula, PUZaren erregistroen balioak ...

Sistema Eragileak testuinguruaren informazioa PCB(Process Control Block) edo Prozesu Kontrol Bloke-an gordetzen du.

Bukatu gabeko prozesu batek PUZ-a utzi behar duenenean, SE-ak prozesuaren PCB-a prozesuen taulan gordeko du, eta PUZ-en sartuko den PCB-aren informazioarekin PUZ-aren ingurunea kargatuko du. Honi **Testuinguru-aldaketa** deitzen zaio.

Testuinguru-aldaketa

Noiz gertatzen da testuinguru-aldaketa?

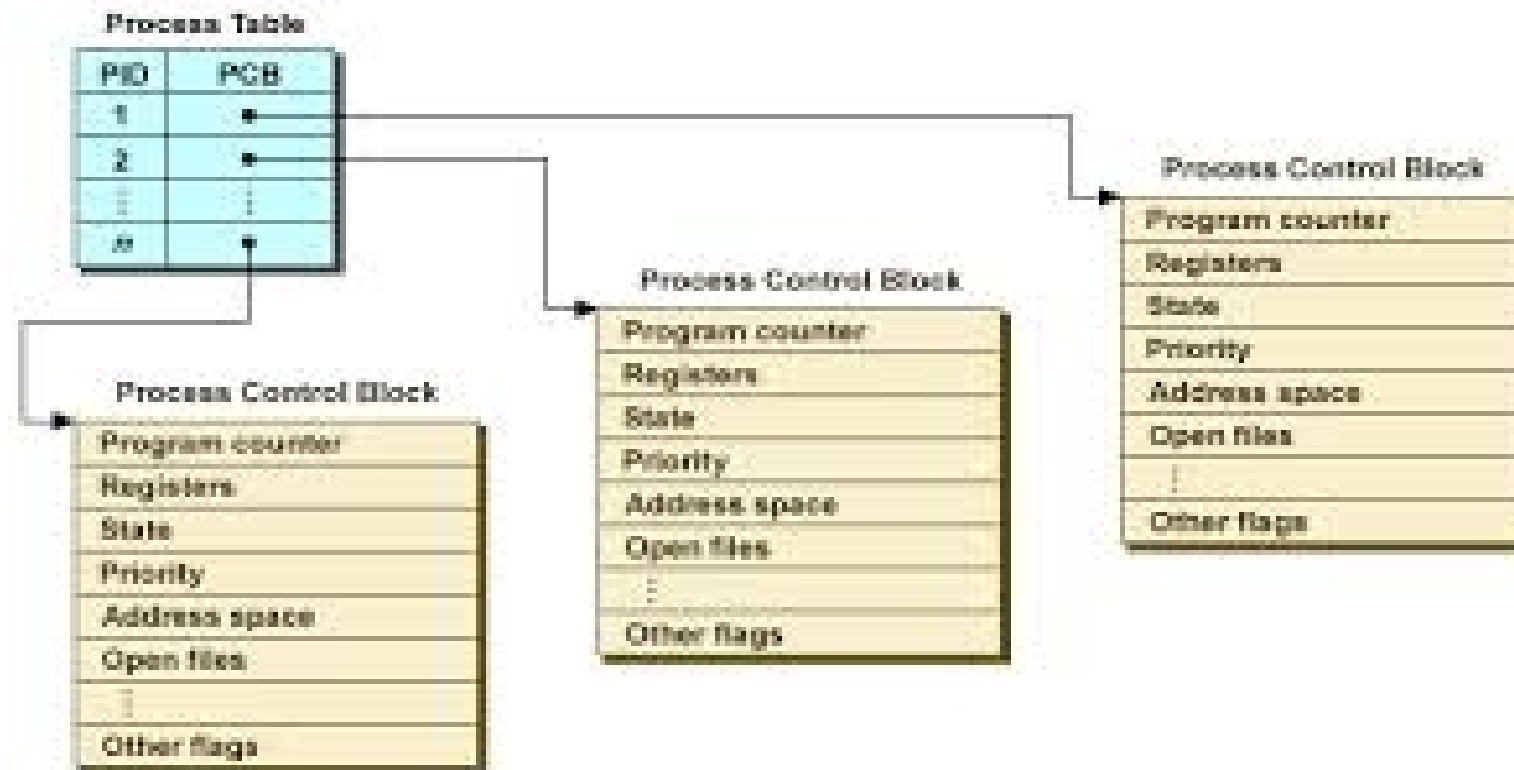
- Quantum-a bukatzerakoan
- Prozesua bukatzerakoan
- Sinkronizazioan, prozesu bat beste bati itxaron behar dionean.
- Software eten bat gertatzen denean
- Hardware eten bat gertatzen denean
- Lehentasun handiagoko prozesu bat helduz gero

PCB

Edukina:

- Prozesu-zenbakia
- Prozesu-egoera
- PUZ-eko erregistroen edukina(programaren kontagailua ...)
- Zabalik dauden fitxategien deskriptoreak
- Memoriaren informazioa
- Lehentasuna
- Kontabilitatearen informazioa (erabilitako PUZ denbora, ...)

PCB



Prozesuak: kontrola eta kudeaketa

Juanan Pereira

juanan.pereira@ehu.es

Kepa Bengoetxea

kepa.bengoetxea@ehu.es

Mikel Larrea

<http://tinyurl.com/yz5wk33>

Prozesu kudeaketa

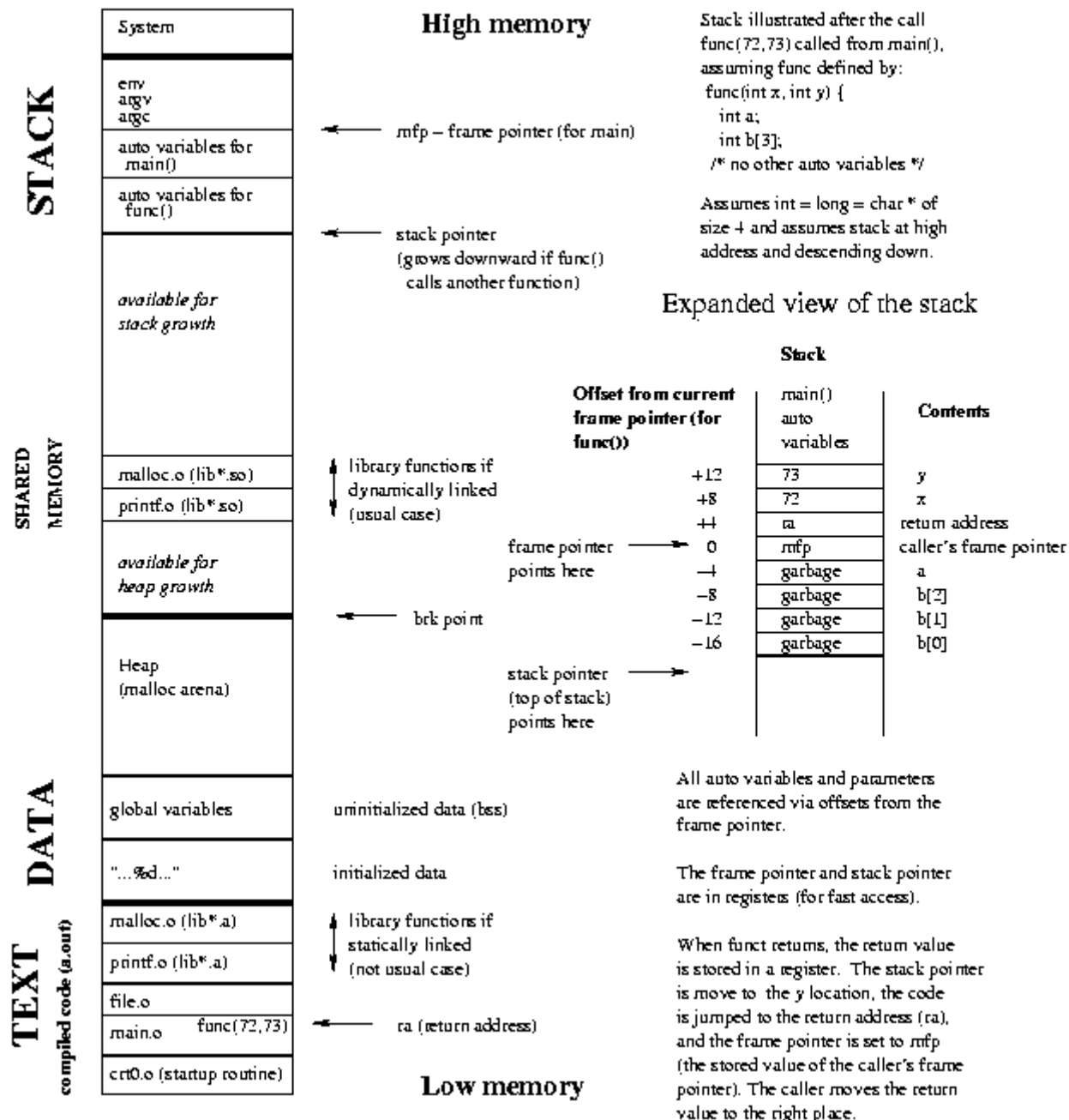
Prozesua: erabiltzaile baten eskaeraren ondorioz martxan dagoen programa. Oinarrizko elementua da sistemarentzat, eta **identifikadore** bat du esleiturik.

Prozesu kudeaketa

A process can broadly be defined into following segments :

- Stack*: Stack contains all the data that is local to a function like variables, pointers etc. Each function has its own stack. Stack memory is dynamic in the sense that it grows with each function being called.
- Heap*: Heap segment contains memory that is dynamically requested by the programs for their variables.
- Data*: All the global and static members become part of this segment.
- Text*: All the program instructions, hard-coded strings, constant values are a part of this memory area.

Memory Layout (Virtual address space of a C process)



Prozesu kudeaketa

Prozesuak (egikaritzen ari diren programak) zuhaitz bat osatzen dute. Zuhaitz horren erroan **systemd** prozesua (PID 1 duena) kokatzen da.

“pstree” komandoak prozesuen zuhaitza pantailaratuko du.

Aukerak:

- u prozesua hasi zuenaren izena pantailaratzeko
- p prozesuaren identifikatzailea bistaratzeko

```
$pstree -p
systemd(1)─┬─ModemManager(871)─┬─{gdbus}(973)
            │                   └─{gmain}(966)
            └─NetworkManager(909)─┬─dhclient(1390)
                                    ├──dhclient(2029)
                                    ├──dnsmasq(1399)
                                    ├──{gdbus}(1069)
                                    └─{gmain}(1067)
```

Prozesu kudeaketa

ps komandoa: prozesuen informazioa lortzeko

- u erab01 (erab01 erabiltzailearen prozesuak bistaratu)
- t pts/0 (lehenengo terminaleko prozesuak bistaratu, ze terminalean gauden jakiteko tty komandoa erabili)
- p pid (pid identifikatzailea duen prozesuaren informazioa bistaratu)
- a beste erabiltzaileen prozesuak bistaratu (eta ez soilik uneko erabiltzailearenak)
- x sistemaren prozesuak ere bistaratu
- f full (prozesuen zuhaitza pantailaratu)

Prozesu kudeaketa

ps -aux komandoak hurrengo informazioa pantailaratu dezake:

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
kepa	3496	0.0	0.0	210380	7820	?	Sl	08:53	0:00	/usr/lib/libreoffice/program/oosplash --impress file:///media/datos/Dropbox/docencia/isobilbo/ISO17_18/Enlace%20hacia %20Gaiak/5.ProzesuenKudeaketa/teoria/5_2_prozesuak_laburtua.odp
kepa	3515	0.5	0.9	1427988	152876	?	Sl	08:53	0:05	/usr/lib/libreoffice/program/soffice.bin --impress file:///media/datos/Dropbox/docencia/isobilbo/ISO17_18/Enlace%20hacia %20Gaiak/5.ProzesuenKudeaketa/teoria/5_2_prozesuak_laburtua.odp --splash-pipe=5
kepa	3570	0.3	0.2	666136	38504	?	Sl	09:02	0:01	/usr/lib/gnome-terminal/gnome-terminal-server
kepa	3577	0.0	0.0	25296	5372	pts/2	Ss	09:02	0:00	bash
kepa	3690	0.0	0.0	39936	3340	pts/2	R+	09:11	0:00	ps -aux
kepa	3691	0.0	0.0	16764	940	pts/2	S+	09:11	0:00	grep --color=auto kepa

*RSS the Resident Set Size and is used to show how much memory is allocated to that process and is in RAM. It does not include memory that is swapped out. It does include memory from shared libraries as long as the pages from those libraries are actually in memory. It does include all stack and heap memory.

*VSZ is the Virtual Memory Size. It includes all memory that the process can access, including memory that is swapped out and memory that is from shared libraries.

For example: if process A has a 500K binary and is linked to 2500K of shared libraries, has 200K of stack/heap allocations of which 100K is actually in memory (rest is swapped), and has only actually loaded 1000K of the shared libraries and 400K of its own binary then:

RSS: 400K + 1000K + 100K = 1500K

VSZ: 500K + 2500K + 200K = 3200K

Prozesu kudeaketa

Seinaleak: kernelak prozesuei bidaltzen dizkien mezuak. Seinaleek osoko identifikatzaile bat dute. **kill**: prozesu bati seinale bat bidaltzeko komandoa. Adibidez: **kill -9 PID**

kill -l

1) SIGHUP 2) SIGINT 3) SIGQUIT 4) SIGILL 5) SIGTRAP

6) SIGABRT 7) SIGBUS 8) SIGFPE 9) SIGKILL 10) SIGUSR1

11) SIGSEGV 12) SIGUSR2 13) SIGPIPE 14) SIGALRM 15) SIGTERM

16) SIGSTKFLT 17) SIGCHLD 18) SIGCONT 19) SIGSTOP 20) SIGTSTP

21) SIGTTIN 22) SIGTTOU

Prozesu kudeaketa

- Erabiltzaile arrunt batentzat benetan interesgarriak diren seinaleak honakoak dira:
- KILL**: PID prozesua amaitu (baldintzarik gabe eta berehala)
- HUP**: PID prozesuari bere konfigurazio fitxategia irakurtarazi
- TERM**: PID prozesua amaitu (-TERM-ek prozesuari ondo bukatzeko aukera ematen dio. -KILL-ek, berriz, ez)
- STOP**: =Ctrl+Z (prozesua eten, lotan utzi)
- CONT**: PID prozesuari jarraitzeko seinalea bidali (orokorrean, -STOP seinalea jaso eta gero lantzen ohi den seinalea)

Prozesu kudeaketa

- Adibideak

Kill komandoa balio lehenetsi bezala TERM seinalea bidaliko du.

- \$ kill 4541
- \$ kill -15 4541
- \$ kill -TERM 4541
- \$ kill -s SIGTERM 4541

- \$ kill -9 3454
- \$ kill -KILL 3454
- \$ kill -s SIGKILL 3454

Prozesu kudeaketa

top: prozesuak baliabideen kontsumoaren arabera ikusteko (segundo batzuen maiztasun finkoarekin eguneratzen da informazioa)

Laguntza: "h" edo "?"

Ordenatzeko irizpidea hautatu : ">" edo "<" sakatu

Txikietatik handienera edo alderantziz ordenatu: R

Nahi ditugun zutabeak hautatu: "f"

Koloreak aldatu (oso gomendagarria): z

Azpimarratu ordenatzeko irizpidea: x

Prozesu kudeaketa

nohup komandoa:

Seme prozesu bat hil baino lehen aita hiltzen bada, semea **systemd** prozesuak adoptatuko du. Sabuespen bakarra “bash” aitarekin bidalitako prozesu semeak. “Bash” aita itxiz gero, bere “bash”etik bidalitako prozesu seme guztiak hilko ditu. Hori ekiditzeko, nohup komandoa erabil daiteke.

nohup komandoa

Noiz erabili? Zerbitzari batean komando edo ataza bat bidaltzerakoan

\$ ssh **erabiltzailea@helbidea**

nohup komandoa

Prozesu kudeaketa

nice komandoa: prozesu baten lehentasuna aldatu.

Sintaxia: (egikaritu behar den komando baten lehentasuna aldatu)

nice -n <x> komandoa

<x> -20 eta +19 bitartean egongo da, -20 lehentasun handiena izanik (cpu denbora gehien hartuko duena)

root erabiltzaileak esleitu ditzake 0 baino txikiagoak diren lehentasunak.

Jada egikaritu den komando baten lehentasuna aldatu nahi bada:

renice <x> PID

Prozesu kudeaketa

nice komandoa. Adibidea:

```
$yes > /dev/null &
```

```
[1] 9862
```

```
$ps -l
```

F	S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD
0	S	500	9841	5770	0	75	0	- 1409	wait	pts/1		00:00:00	bash
0	R	500	9862	9841	65	85	0	- 1203	-	pts/1		00:00:03	yes
0	R	500	9863	9841	0	76	0	- 597	-	pts/1		00:00:00	ps

```
$ renice +19 9862
```

9862: prioridad antigua 0, nueva prioridad 19

```
$ sudo renice -20 9862 (como root, para menores de 0)
```

9862: prioridad antigua 19, nueva prioridad -20

```
kill -9 9862
```

Prozesu kudeaketa

```
bcplemza@B900112:~$ ps -l
```

F S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD
0 S	1000	2466	2461	7	80	0	-	6938	wait	pts/1	00:00:00	bash
0 R	1000	2521	2466	0	80	0	-	3379	-	pts/1	00:00:00	ps

```
bcplemza@B900112:~$ nice -n 5 xclock &
```

```
[1] 2522
```

```
bcplemza@B900112:~$ ps -l
```

F S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD
0 S	1000	2466	2461	1	80	0	-	6938	wait	pts/1	00:00:00	bash
0 S	1000	2522	2466	0	85	5	-	16049	poll_s	pts/1	00:00:00	xclock
0 R	1000	2523	2466	0	80	0	-	3379	-	pts/1	00:00:00	ps

```
bcplemza@B900112:~$ renice 12 2522
```

```
2522: prioridad antigua 5, nueva prioridad 12
```

```
bcplemza@B900112:~$ ps -l
```

F S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD
0 S	1000	2466	2461	0	80	0	-	6938	wait	pts/1	00:00:00	bash
0 S	1000	2522	2466	0	92	12	-	16049	poll_s	pts/1	00:00:00	xclock
0 R	1000	2525	2466	0	80	0	-	3379	-	pts/1	00:00:00	ps

```
bcplemza@B900112:~$
```

Prozesu kudeaketa

tee komandoa: sarrera estandarretik irakurritakoa irteera estandarrean eta argumentu bezala emandako fitxategian kopiatzen du

Adibidea:

```
kepa@cox:/tmp$ who | tee konektatuta | sort | tee ordenatua  
kepa pts/0      2009-11-03 21:30 (:0.0)  
kepa pts/1      2009-11-03 22:01 (:0.0)  
kepa tty7       2009-11-03 20:31 (:0)
```

Prozesu kontrola

sleep komandoa: eten bat sortzen du, guk parametroan ezarritako denbora tarte bitartean

Adibidea:

```
sleep 15m 10s ; mplayer iratzargailua.mp3
```

15 minutu eta 10 segundu barru, iratzargailua.mp3 soinu fitxategia jo.

Prozesu kontrola

at komandoa: prozesuen denbora jakin batean **behin** exekutatzeko erabiltzen da (atd deabrua haietaz arduratzen da).

Instalatzeko: sudo aptitude install at

Sintaxia:

at <ordua> <agindua>

Aukerak:

at -l (programatuta dauden atazen zerrenda eskatu)

at -d <n> (n zenbakia duen ataza zerrendatik kendu)

man at (-f aukera ikusteko)

Prozesu kontrola

Adibidea:

```
$ at now +2 minutes -f eskripta.sh
```

```
$ at now +2 minutes
```

```
at> echo kaixo > /home/kepa/Desktop/kaixo.txt
```

```
at> <EOT>  <--- Ctrl+D
```

```
job 1 at Sun Nov 15 19:50:00 2009
```


Prozesu kontrola

Atd zerbitzua kudeatu:

```
sudo service atd stop      #gelditu  
sudo service atd start     #hasieratu  
sudo service atd status    #egoera
```

Prozesu kontrola

crontab komandoa: prozesuak **aldizka** exekutatzeke erabiltzen da (crond deabrua haietaz arduratzen da).

• Zertarako erabiltzen da?

- Katalogo batzuen segurtasun kopiak egiteko.
- Minutu gutxi batzuetara erabiltzaile konektatuen informazioa bildu egiteko.
- Sistemaren eguneraketa egiteko
- ...

Prozesu kontrola

- Sintaxia: **crontab -e** , “/var/spool/cron/crontabs/kepa” katalogoan gordeko du, fitxategi honen egitura, hilara bakoitzean 6 zutabe hutsune bategaz banatuta:
 - 1.-Minutuak 0-59
 - 2.-Orduak 0-23
 - 3.-Hileko zein eguna 1-31
 - 4.-Hila 1-12
 - 5.-Asteko eguna 0 (igandea) eta 6 (larunbata)
 - 6.-komandoa edo ataza

Prozesu kontrola

•Adibidez:crontab -e

```
# m h dom mon dow  command
```

```
30 9 * * * touch /home/kepa/Escritorio/proba.txt
```

Prozesu kontrola

• "crontab"-aren aukerak:

- l ikusi nire crontab lerroak
- e editatu edo borratu nire crontab lerroak
- r nire crontab fitxategia ezabatu
- u <erabiltzailea> (root edo sudo bezala soilik)

Prozesu kontrola

• Hartu dezakeen baloreak:

* edozein

2-6(2tik 6ra)

2,4,6(soilik 2,4 eta 6)

*/5 (5 minuturo, 5 orduro...)

Prozesu kontrola

•Gure lana errazteko definitutako katalogo batzuk daude /etc barruan. Katalogo horretan jartzen dituzun exekutagarriak (scriptak, komandoak eta abar ...) orduro, egunero, eta abar exekutatuko dira. Katalogoak, cron.hourly, cron.daily.... dira.

```
$less /etc/crontab
```

```
SHELL=/bin/bash
```

```
PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin
```

```
# m h dom mon dow user  command
```

```
17 * * * * root    cd / && run-parts --report /etc/cron.hourly
```

```
25 6 * * * root    test -x /usr/sbin/anacron || ( cd / && run-parts --report /etc/cron.daily )
```

Oharra: anacron martxan egonez gero, /etc/cron.daily katalogoan dauden scriptak edo exekutagarriak ez dira exekutatu.

Prozesu kontrola

- Adibidez: “proba.sh” scripta egunero exekutatzeko
cd /etc/cron.daily/
gedit proba.sh
echo kaixo > /home/kepa/Escritorio/kaixo.txt
- Loga edo bitakora: exekutatu den jakiteko. less /var/log/syslog | grep cron.hourly

Nov 30 10:17:01 kepa-laptop CRON[9303]: (root) CMD (cd / && run-parts
--report /etc/cron.hourly)

Nov 30 11:17:01 kepa-laptop CRON[9824]: (root) CMD (cd / && run-parts
--report /etc/cron.hourly)

...

Crontab & vi

```
sudo apt-get remove vim.tiny
```

```
sudo apt-get install vim
```

Command mode: The ESC key can end a command

Insert mode: Text is inserted. The ESC key ends insert mode and returns you to command mode. One can enter insert mode with the "i" (insert), "a" (insert after), "A" (insert at end of line), "o" (open new line after current line) or "O" (Open line above current line) commands.

Command line mode: One enters this mode by typing ":" which puts the command line. **q!** (Ignore changes and quit. No changes from last write will be saved.) **wq** (Save (write) changes to current file and quit.

Crontab & gedit

Añadir la siguiente línea en el archivo .bashrc

```
export EDITOR=gedit;
```

```
source .bashrc
```

```
env
```

```
crontab -e
```

```
# m h dom mon dow  command
```

```
38 9 * * * touch /home/euiti/Escritorio/prueba.txt
```

Prozesu kontrola

- "**anacron**" komandoa: crontab bezalakoa da, baina ordenagailua momentu oro martxan ez dauden kasuetarako erabiltzen da.

Konfigurazio fitxategia: **/etc/anacrontab**:

SHELL=/bin/sh

PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin

MAILTO=root

#aldia atzerapena ataza-identificador-tarea komandoa

1 5 cron.daily nice run-parts --report /etc/cron.daily

7 10 cron.weekly nice run-parts --report /etc/cron.weekly

@monthly 15 cron.monthly nice run-parts --report /etc/cron.monthly

Prozesu kontrola

- Anacron komandoa exekutatzeko, **aldi** zutabearen dagoan azkenengo **n** egunetan komandoa exekutatu ez bada, **atzerapen minutuak** itxaron ostean ataza hori exekutatu du.

Logak: `less /var/log/syslog | grep anacron`

Nov 30 10:07:13 kepa-laptop anacron[1061]: Job `cron.daily' terminated

Nov 30 10:07:13 kepa-laptop anacron[1061]: Normal exit (1 job run)

Anacron ordenagailua abiatzerakoan exekutatzen da eta gero amaitzen da.

Jakiteko noiz exekutatu den azkenengo aldiz: `ls /var/spool/anacron/`

`cron.daily cron.monthly cron.weekly`

`sudo less /var/spool/anacron/cron.daily --> 20170322`

Prozesu kontrola

Gehiago jakiteko:

<https://help.ubuntu.com/community/CronHowto>

Prozesuen kontrolerako sistema-deiak Unix-en

kepa.bengoetxea@ehu.es

Prozesuen kontrolerako sistema-deiak Unix-en

Prozesuen identifikazioa

getpid, getppid, getuid

Prozesuen sorrera

fork, exec

Prozesuen bukaera/sinkronizazioa

exit, wait, kill

Seinaleen kontrola

kill, alarm, pause, signal

Denboraren kontrola

sleep, time, ctime

Prozesuen identifikazioa

int `getpid()`;

- Prozesuaren identifikadorea (pid) bueltatzen du

int `getppid()`;

- gurasoaren identifikadorea (pid) bueltatzen du

int `getuid()`;

- Prozesuaren “jabea” den erabiltzailearen identifikadorea (uid) bueltatzen du

Prozesuen identifikazioa

```
//prozesuak.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
```

```
int main () {
    int id_prozesua, id_gurasoa, id_erabiltzailea;

    id_prozesua = getpid();
    id_gurasoa = getppid();
    id_erabiltzailea = getuid();

    printf("Prozesua: %d\n", id_prozesua);
    printf("gurasoa: %d\n", id_gurasoa);
    printf("Erabiltzailea: %d\n", id_erabiltzailea);
    exit(0);
}
```

```
$ gcc -o prozesuak prozesuak.c
$ ./prozesuak
Prozesua: 4542
gurasoa: 4535
Erabiltzailea: 1000
```

Prozesuen sorrera

int `fork()`;

- Sistema-dei honek prozesu berri bat sortzen du, prozesu deitzailearen “klona” dena. Prozesu deitzaileari gurasoa deituko diogu, eta sortutako prozesuari umea
- umeak bere exekuzioa `fork`-aren hurrengo agindutik hasten du (gurasoa ere bai)
- umeak gurasotik dena heredatzen du, baina umeak pid desberdina du!
- `fork` deiak honakoa bueltatzen du:
 - umeari: 0 (zero)
 - gurasoari: umearen pid-a
 - -1 errore bat gertatuz gero

Prozesuen sorrera

```
int main ()
{
    int pid;
    pid = fork();
    if (pid == -1)
        {perror(fork);
         exit(-1);
        }

    if (pid == 0) /* umea */
        {printf("%d umea naiz, %d gurasoarena\n",
                getpid(), getppid());}
    else /* gurasoa */
        {printf("%d gurasoa naiz, %d umearena\n",
                getpid(), pid);}

    printf("Bukatzera noa %d\n", getpid()); /*biak*/
    exit(0);
}
```

Prozesuen sorrera

```
2121 gurasoa naiz, 3456 umearena  
3456 umea naiz, 2121 gurasoarena  
Bukatzera noa 2121  
Bukatzera noa 3456
```

```
3456 umea naiz, 2121 gurasoarena  
2121 gurasoa naiz, 3456 umearena  
Bukatzera noa 2121  
Bukatzera noa 3456
```

```
2121 gurasoa naiz, 3456 umearena  
Bukatzera noa 2121  
3456 umea naiz, 1 gurasoarena  
Bukatzera noa 3456
```

... "1" da bere gurasoa
bukatu duelako, eta
Unix-en umezurtz
prozesuak "init"-ek
adoptatzen dituelako

fork sistema-deiaren proba

```
main()
```

```
{
```

```
    int pid;
```

```
    printf("fork deiaren proba\n");
```

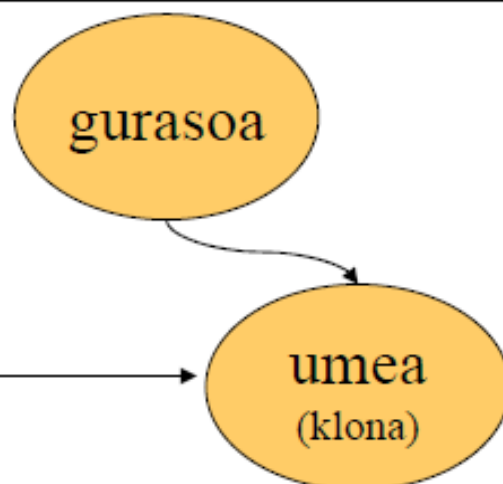
```
    pid = fork();
```

```
    printf("kontrola itzuli zaio ");
```

```
    if (pid == 0) printf("umeari\n");
```

```
    else printf("gurasoari, %d izanik ume berriaren pid-a\n", pid);
```

```
}
```



gurasoaren irteera

fork deiaren proba

kontrola itzuli zaio gurasoari, 999 izanik ume berriaren pid-a

umearen irteera

kontrola itzuli zaio **umeari**

•gurasaia

•**umea**

•gurasaia + umea

fork sistema-deiaren proba

Irteeraren kasu erreal posible batzuk

gurasoa

umea

gurasoa + umea

fork deiaren proba

kontrola itzuli zaio gurasoari, 999 izanik umearen pid-a

kontrola itzuli zaio umeari

fork deiaren proba

kontrola itzuli zaio umeari

kontrola itzuli zaio gurasoari, 999 izanik umearen pid-a

fork deiaren proba


kontrola itzuli zaio kontrola itzuli zaio umeari

gurasoari, 999 izanik umearen pid-a

...

571

```
Int i = 3, c_pid = -1;  
c_pid = fork();  
if(c_pid == 0)  
    printf("child process");  
else if(c_pid > 0)  
    printf("parent process");
```




Data

i = 3
c_pid = 574

574

```
Int i = 3, c_pid = -1;  
c_pid = fork();  
if(c_pid == 0)  
    printf("child process");  
else if(c_pid > 0)  
    printf("parent process");
```



Data

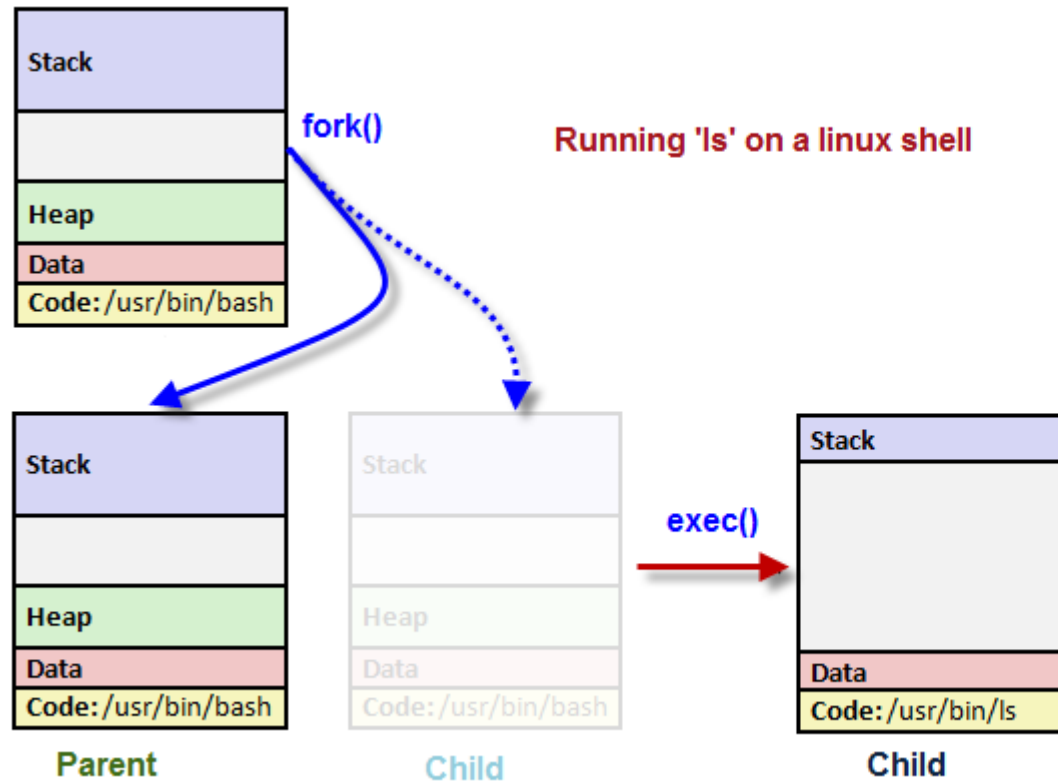
i = 3
c_pid = 0

Prozesuen sorrera

int `exec??`(...);

- Prozesua exekutatzen ari den programa aldatzen du
 1. Prozesuaren edukia husten du, testuingurua mantenduz
 2. Programa berria kargatzen du
- `exec??` deia ongi burutzen bada, ez du ezer bueltatzen... `programa aldatu egin delako!`
- Erroreren bat gertatzen bada, `-1` bueltatzen du

Prozesuen sorrera



Prozesuen sorrera

2121

```
main() /* adibide3 */
{
    int pid;

    pid = fork();
    switch (pid){
        case -1: /* errorea */
            exit(-1);
        case 0: /* umea */
            execlp("ls", "ls", "-al", NULL);
            errore("execlp");
            break;
        default: /* gurasoa */
            printf("%d gurasoa, %d umearena\n",
                getpid(), pid);
    }
}
```



3456

```
main() /* ls */
{
    /* programa berria */
}
```

Prozesuen sorrera

exec funtzio familia:

parametro zerrenda:

int **execl** (char *path, char *arg0, ..., NULL);

parametro bektore:

int **execv** (char *path, char *arg[]);

parametro zerrenda + environment:

int **execle** (char *path, char *arg0, ..., char *envp[]);

parametro bektore + environment:

int **execve** (char *path, char *arg[], char *envp[]);

parametro zerrenda (izena path barik PATH-aldagaian bilatu):

int **execlp** (char *file, char *arg0, ..., NULL);

parametro bektore (izena path barik PATH-aldagaian bilatu):

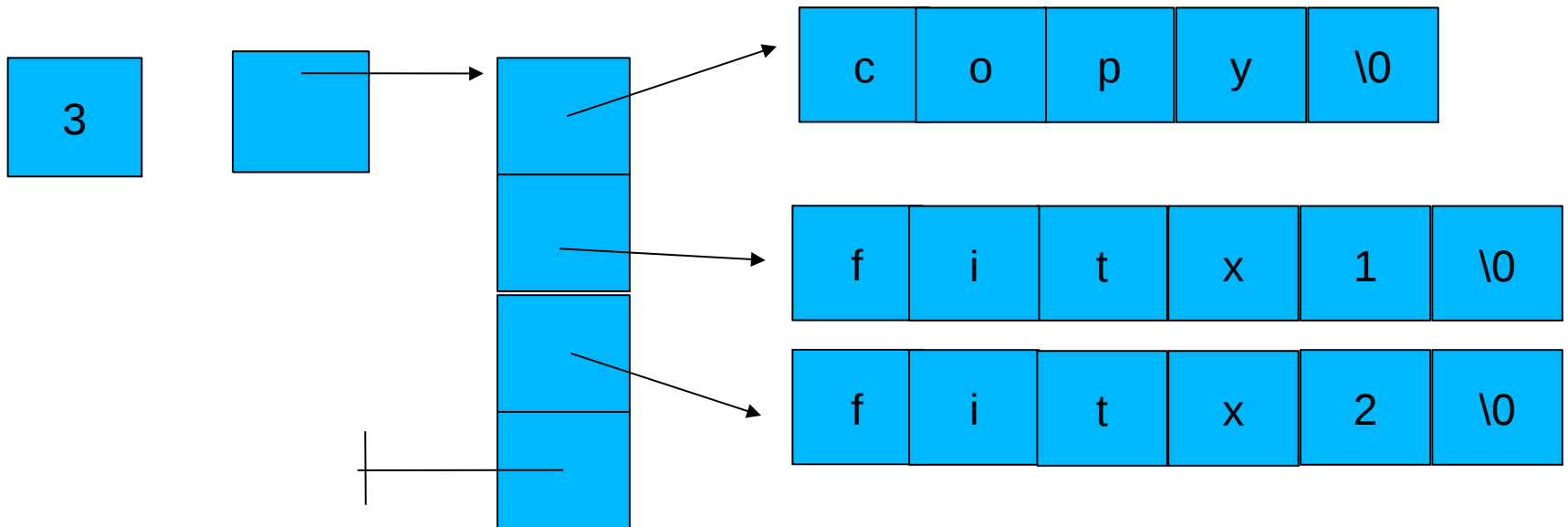
int **execvp** (char *file, char *arg[]);

- Demagun C lengoaian idatzita dagoen copy.c programa dugula. Bere main() metodoan:

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

```
argv[0]="copy"  
argv[1]="fitx1"  
argv[2]="fitx2"
```

Deia: copy fitx1 fitx2



Prozesuen Sorrera

Exec deien adibideak:

```
execl("/bin/ls", "ls", "-l", NULL);  
execle("/bin/ls", "ls", "-l", NULL, NULL);  
execlp ("ls", "ls", "-l", NULL);
```

```
char *v[] = {"ls", "-l", NULL};  
execv ("/bin/ls", v);  
execve("/bin/ls", v, NULL);  
execvp("ls", v);
```

Prozesuen sorrera:exec

```
int main() /* execlp aginduaren adibidea */
{
    int pid;
    pid = fork();
    switch (pid){
        case -1: /* errorea */
            perror("execlp:");
            exit(-1);
        case 0: /* umea */
            execlp("ls", "ls", "-al", NULL);
            printf("Ezin izan da execlp egikaritu\n");
            exit(-1);
        default: /* gurasoa */
            printf("%d gurasoa, %d umearena\n", getpid(), pid);
    }
    exit(0);
}
```

Prozesuen sorrera

```
kepa@cox:/tmp/c$ ./execlp
```

```
15316 gurasoa, 15317 umearena
```

```
drwxr-xr-x 2 kepa kepa 4096 2009-11-14 16:42 .
```

```
drwxrwxrwt 19 root root 524288 2009-11-14 16:42 ..
```

```
-rwxr-xr-x 1 kepa kepa 9232 2009-11-14 16:42 execlp
```

```
-rw-r--r-- 1 kepa kepa 512 2009-11-14 16:42 execlp.c
```

```
kepa@cox:/tmp/c$ ./execlp
```

```
drwxr-xr-x 2 kepa kepa 4096 2009-11-14 16:42 .
```

```
drwxrwxrwt 19 root root 524288 2009-11-14 16:42 ..
```

```
-rwxr-xr-x 1 kepa kepa 9232 2009-11-14 16:42 execlp
```

```
15320 gurasoa, 15321 umearena
```

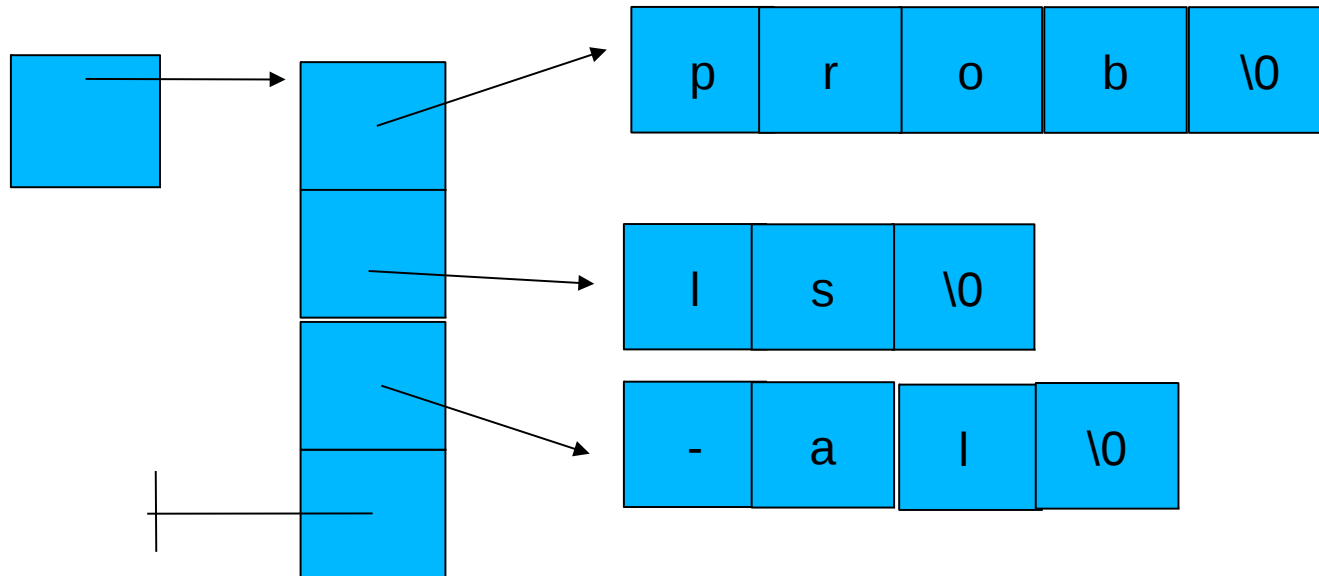
```
-rw-r--r-- 1 kepa kepa 512 2009-11-14 16:42 execlp.c
```

Prozesuen sorrera

```
Int main(int argc, char *argv[]) /* prob */
{
    int pid;
    pid = fork();
    switch (pid){
        case -1: /* errorea */
            perror("fork:");
            exit(-1);
        case 0: /* umea */
            execvp(argv[1], &(argv[1]));
            perror("execvp:");
            exit(-1);
        default: /* gurasoa */
            printf("%d gurasoa, %d umearena\n", getpid(),
                                                           pid);
    }
    exit(0);
}
```


Prozesuen sorrera

- Zergatik `execvp(argv[1], &argv[1])`?



Prozesuen sorrera:exec

Gedit guraso.c

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

```
#include <unistd.h> /*man fork,exec */
```

```
#include <sys/types.h> /*man 2 wait */
```

```
#include <sys/wait.h> /*man 2 wait */
```

```
int main() {
```

```
int pid,status=1;
```

```
pid=fork();
```

Prozesuen sorrera:exec

```
if (pid == 0) { /* Umea */  
    if (execl("umea","umea", "izena", "-a", NULL) == -1) {  
        perror("execl");  
        exit(1);} }  
else { /* Guraso */  
    wait(&status);  
    printf("\nSemearen bukaerako egoera: %d\n", status);  
    exit(0);}  
}
```

Prozesuen sorrera:exec

Gedit umea.c

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

```
int main(int argc, char *argv[]) {
```

```
    int i = 0;
```

```
    for (i = 0; i < argc; i++) printf("\nArgumentua [%d]: %s", i, argv[i]);
```

```
    exit(0);}
```

```
gcc -o guraso guraso.c
```

```
gcc -o umea umea.c
```

```
./guraso
```

Prozesuen sorrera:exec

./guraso

Argumentua [0]: umea

Argumentua [1]: izena

Argumentua [2]: -a

Semearen bukaerako egoera: 0

Prozesuen bukaera/sinkronizazioa

void **exit**(int egoera);

- Prozesuaren bukaera “kontrolatua”
- Unix-ek bukaera-kodea (egoera) gorde egiten du gurasoak wait() bukatu arte

int **wait**(int *egoera);

- Deitzailea bere umearen bukaera arte geldiarazten du
- Umerik ez badu, -1 bueltatzen du, deitzailea blokeatu gabe
- Bukatzen duen ume-prozesuaren identifikadorea bueltatzen du
- **egoera** umea bueltatutako bukaera-kodea da

int **kill**(int pid, int SIGKILL);

- SIGKILL seinalea pid prozesuari bidaltzen dio, bukaraziz

Prozesuen bukaera/sinkronizazioa

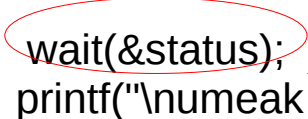
```
#include <stdio.h>
#include <unistd.h> /*man 2 fork */
#include <sys/types.h> /*man 2 wait */
#include <sys/wait.h>
#include <stdlib.h>

int main() {
    int pid = 0, status = -1;

    if ((pid = fork()) == -1)
    { printf('Errorea umea sortzerakoan \n');exit(1); }

    if (pid == 0)
    { printf('Nire gurasoaren PID zenbakia: %d\n',getppid()); exit(0); }

    else
    { printf("Nire PID zenbakia: %d eta nire
                                     umearena: %d\n", getpid(), pid);
      wait(&status);
      printf("\n umeak itzuli duen egoera zenbakia: %d \n", status);
      exit(0);
    }
}
```



umea bukatu arte itxaron

Prozesuen bukaera/sinkronizazioa

```
kepa@cox:/tmp$ ./wait  
Nire PID zenbakia: 22079 eta nire umearena: 22080  
Nire gurasoaren PID zenbakia: 22079
```

```
umeak itzuli duen egoera zenbakia: 0
```

```
kepa@cox:/tmp$ ./wait  
Nire PID zenbakia: 22081 eta nire umearena: 22082  
Nire gurasoaren PID zenbakia: 22081
```

```
umeak itzuli duen egoera zenbakia: 0
```

```
kepa@cox:/tmp$ ./wait  
Nire gurasoaren PID zenbakia: 22083  
Nire PID zenbakia: 22083 eta nire umearena: 22084
```

```
umeak itzuli duen egoera zenbakia: 0
```


Prozesuen bukaera/sinkronizazioa

- **kill:**

- `#include <sys/types.h> eta <signal.h>: int kill(pid_t pid, int sig);`
- Prozesu bati SIGXXXX seinale bat bidaltzeko.
- 0 itzuliko du ondo joanez gero eta -1 errore bat gertatuz gero.
- `Adb:kill(pid,SIGTERM)`

Prozesu bati seinaleak bidali. Adibidea:

```
#include <sys/types.h> /*kill y wait*/

#include <sys/wait.h> /*wait*/

#include <signal.h> /*kill*/

#include <stdlib.h> /*puts y exit*/

#include <stdio.h> /*printf*/

#include <unistd.h> /*fork*/

int main(void) { pid_t hijo; int condicion, valor_retornado;

hijo=fork();

if (hijo==-1) {perror("fork");exit(1);}

if (hijo==0) {sleep(1000);exit(0);}

else {valor_retornado=kill(hijo,SIGKILL);

        if (valor_retornado==-1) {perror("kill");wait(&condicion);}

        else {printf("%d ezabatua \n",hijo);}

exit(0); }}
```

Prozesuen bukaera/sinkronizazioa

```
$gcc -o killer killer.c
```

```
$ ./killer
```

6680 ezabatua

Prozesuen bukaera/sinkronizazioa

- Prozesu baten exekuzioan, edozein momentuan eman ahal den gertaerari seinalea deritzo (Seinalea = Software etena)
- Seinale bakoitza izen bat du (kill -l edo man 7 signal) eta seinaleekin lan egiten dituzten funtzioak `</usr/include/signal.h>` daude.
- Prozesu batek seinale bat jasoz gero :
 - Seinalea kontuan ez hartu
 - Seinale horri dagokion funtzio lehenetsia (gehienetan exit bat core edo core barik)
 - Seinale horri lotu ahal diogu beste funtzio bat (handler), eta gero exit edo gertaera gertatu aurreko instrukziora itzuli.

Prozesuen bukaera/sinkronizazioa

\$ man 7 signal

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

Prozesuen bukaera/sinkronizazioa

man 7 signal

signal - lista de las señales disponibles

Señal	Valor	Acción	Comentario
-------	-------	--------	------------

SIGINT	2	A	Interrupción procedente del teclado(Ctrl-C)(salida)
SIGQUIT	3	C	Terminación procedente del teclado(Ctrl-4)(salida con core)
SIGABRT	6	C	Señal de aborto procedente de abort(3)
SIGKILL	9	AEF	Señal de matar
SIGALRM	14	A	Señal de alarma de alarm(2)
SIGTERM	15	A	Señal de terminación
SIGUSR1	30, <u>10</u> ,16	A	Señal definida por usuario 1
SIGCONT	19, <u>18</u> ,25		Continuar si estaba parado
SIGSTOP	17, <u>19</u> ,23	DEF	Parar proceso

Prozesuen bukaera/sinkronizazioa

A-La acción por defecto es terminar el proceso

B-La acción por defecto es ignorar la señal

C-La acción por defecto es terminar el proceso con core dump

D-La acción por defecto es parar la ejecución del proceso

E-La señal no puede ser capturada por el programa(manipulada)

F-La señal no puede ser ignorada

***core dump: volcado de memoria del contexto del proceso a la carpeta del proceso, para poder ver con un programa de depuración como gdb, sdb o adb**

Seinaleen kontrola

`int kill(int pid, int SIGKILL);`

SIGKILL seinalea pid prozesuari bidaltzen dio, bukaraziz

`void pause();`

Funtzio hau seinale bat jaso arte lo geratzen da. -1 itzultzen du, seinaleari dagokion funtzioa exekutatu ostean.

`int signal(int seinalea, void funtzioa());`

Seinalea jasoz gero exekutatuko den funtzioa

Seinaleen kontrola

Adibidez: gedit pausa.c

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

```
#include <stdlib.h>
```

```
int main(void){pause();
```

```
exit(0);}
```

```
$gcc -o pausa pausa.c
```

```
$/pausa
```

```
Ctrl-4 edo Ctrl-C edo kill -USR1 <PID>
```

Irten

Seinaleen kontrola

```
int signal(int seinala, void funtz());
```

- Seinaleari funtz funtzioa lotzen dio . Signal funtzioak 2 parametro jasotzen ditu:
 - Jaso nahi dugun seinala zenbakia. Zeintzuk? “kill -l”
 - Seinala hori jaso ostean exekutatuko den funtzioaren erakuslea. Hurrengo makroak ere erabili ahal dira: SIG_IGN (seinala kontuan ez hartu) edo SIG_DFL(dagokion portaera lehenetsia uztea)
- Errore bat gertatuz gero SIG_ERR itzultzen du

Seinaleen kontrola

- Funtzio baten erakuslea:
 - C-n funtzio bat konpilatzerakoan bere kodean sarrera puntu bat sortzen da.
 - Funtzio baten deia egiterakoan, kode sarrera puntuari dei bat egiten ari gara.
 - Kode sarrera puntua ere erabili ahal da funtzioari dei egiteko.
 - Kode sarrera puntuaren helbidea funtzioaren izena da. (arraian bezala)

Adibidez, “printf” funtzioaren erakusle bat erabiltzeko. Jakinda: printf-ren erazagupena “int printf(const char *format, ...);” dela eta funtzio erakuslea horrela erazagutzen dela “funtzioak_itzultako_mota (*p) ();” ,non “p” erakuslearen izena den:

Erazagupena: int (*p)();

Esleipena: p=printf;

Deia: (*p)(“kaixo”);

Seinaleen kontrola

- **Adibidez:**

```
#include <stdio.h>
#include <signal.h>
int sig;
void jaso (sig)
{printf("jasotako seinalea:
    %d\n",sig);}
int main(){
/*USR1 seinaleari jaso funtzioa
    esleitu */
signal(SIGUSR1,jaso);
```

```
kill(getpid(),SIGUSR1);

/*seinalea kontuan ez hartu*/

signal(SIGUSR1,SIG_IGN);

kill(getpid(),SIGUSR1);

/*dagokion portaera lehenetsia uztea*/

signal(SIGUSR1,SIG_DFL);

kill(getpid(),SIGUSR1);

return 0;

}
```

Denboraren kontrola

alarm: unsigned int alarm(unsigned int seg); SEak deitzaileari SIGALARM seinalea bidaltzen dio seg segundo igaro ondoren. Funtzio honek >0 balore bat itzuliz gero, jadanik alarma bat programatua dagoela adieraziko digu.

sleep: int sleep(unsigned int seconds); SEak prozesua seconds segundu lotan utziko du. Adb:sleep(30)

time: time_t time(time_t *t); devuelve el tiempo transcurrido, medido en segundos, desde 1 de enero de 1970. Si t no es el puntero nulo, el valor devuelto también se guarda en la zona de memoria a la que apunte. Adb:segundoak=time(0);

Denboraren kontrola

Adb: gedit alarma.c

```
#include ...
```

```
int main(void) {
```

```
if ((alarm(5))>0)
```

```
    {printf("jadanik alarma bat programatua dago");exit(0);}
```

```
sleep(30);
```

```
printf("Nola liteke, hemendik igarotzea?");
```

```
exit(1); }
```

```
$gcc -o alarma alarma.c
```

```
$ ./alarma
```

Alarm clock

Denboraren kontrola

Adibide-programa

Guraso-prozesu batek programa baten exekuzioa abiarazten du, honen izena eta argumentuak bigarren parametrotik pasatzen zaiolarik. Lehen argumentuak programaren exekuzio-denbora maximoa adieraziko du; denbora hori pasa eta programak bukatu ez badu, guraso prozesuak umearen exekuzioa amaiaraziko du. Guraso-prozesuak bere identifikadorea eta abiarazten duen programarena ere idazten ditu. Programa berriak amaitzean, gurasoak bere iraupena idatzi eta itzulera-kodea itzultzen du, edo 1 balioa programa amaiarazi badu.

Exekuzio-adibidea:

> ./guraso 60 umea

```
umea.c
#include <unistd.h> /*sleep*/
int main(){
sleep(6);
return 0;
}
```



```
erlojua.c
#include <stdlib.h> /*atoi and exit*/
#include <stdio.h> /*printf*/
#include <unistd.h> /*alarm*/
#include <signal.h> /*signal and SIGALRM*/

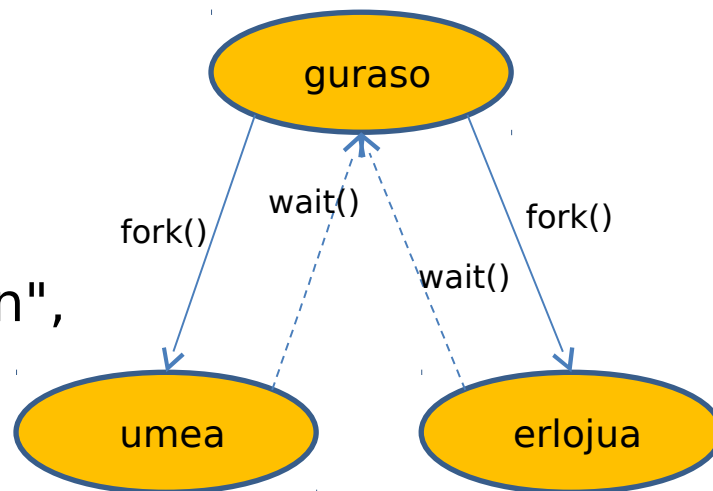
void xtimer(){
printf("time expired.\n");
}

int main(int argc, char *argv[]){
unsigned int sec;
sec=atoi(argv[1]);
printf("time is %u\n",sec);
signal(SIGALRM,xtimer);
alarm(sec);
pause();
return 0;
}
```

gurasoa.c

```
int main(int argc, char *argv[]){
int idHijo, idReloj, id, t1, status1,
status2;
id = getpid();
printf("Proceso padre: %d\n", id);
idReloj = fork();
if(idReloj == 0) { /* hijo Reloj */
execl("/home/kepa/erlojua",
"erlojua", argv[1], NULL);
}
if((idHijo = fork()) == 0) {
execv(argv[2], &argv[2]);
}
printf("Proceso hijo Perezoso:
%d\n", idHijo);
printf("Proceso hijo Reloj: %d\n",
idReloj);
t1 = time(0);
```

```
id = wait(&status1)
if((id == idHijo) {
kill(idReloj, SIGKILL);
//Erlojuaren egoera jaso
wait(&status2);
} else {
kill(idHijo, SIGKILL);
//Semearen egoera jaso
wait(&status2);
status1 = 1;}
t1=time(0)-t1;
printf("Tiempo del proceso hijo : %d\n", t1);
exit(status1);
}
```



```
gcc guraso.c -o guraso
gcc erloju.c -o erloju
gcc umea.c -o umea
./guraso 20 umea
Proceso padre: 3327
Proceso hijo Perezoso: 3329
Proceso hijo Reloj: 3328
time is 20
Tiempo del proceso hijo : 6
$ echo $?
0
```

```
./guraso 3 umea
Proceso padre: 3315
Proceso hijo Perezoso: 3317
Proceso hijo Reloj: 3316
time is 3
time expired.
Tiempo del proceso hijo : 3
echo $?
1
```

Prozesu konkurrenteen arteko komunikazioa eta sinkronizazioa

Egilea

Kepa Bengoetxea Kortazar
kepa.bengoetxea@ehu.es

Bibliografia

- <http://www.chuidiang.com/clinux/procesos/procesoshilos.php>
- Descripción Funcional de los Sistemas Operativos.-Iñaki Alegria
- UNIX.Programación Avanzada.-Manuel Márquez
- Programación en Linux con ejemplos. Kurt Wall
- <http://wwdi.ujaen.es/~lina/TemasSO/CONCURRENCIA/1ComunicacionySincronizacion.htm>
- <http://tiny.uasnet.mx/prof/cln/ccu/mario/sisop/sisop03.htm>(exclusión mutua)
- <http://tiny.uasnet.mx/prof/cln/ccu/mario/sisop/sisop05.htm>(semáforos)

Motibazioa

Sistema eragile multiprogramatua: bi prozesu edo bi prozesu baino gehiago era konkurrentean egikaritzen ari dira.

Baliabideren bat erabiltzerakoan prozesuek kooperatu edota lehiatu egiten dute.

Prozesuek kooperatu egiten dute helburu berdina lortzeko eta lehiatu baliabideak mugatuak direlako: prozesadorea, memoria, fitxategiak eta abar.

Motibazioa

- Prozesu ezberdinen artean komunikazioa eta sinkronizazioa erabiliko da, baliabideak partekatzeko.
- Baliabide posibleak: PUZ, S/I-rako gailuak, aldagaiak, errutinak etab.
- Baliabide mota bi daude: partekagarriak / ez partekagarriak.

Motibazioa

Momentu berean **partekagarriak** diren baliabideak:

- Aldi berean konpartitu daitezke.
- k graduko baliabide partekagarriak ere badaude, k prozesuek batera atzitu dezakete baliabidea. Adibidez: buffer bat $p1$ idatzi eta $p2$ irakurri.
- Adibidez: irakurtzeko fitxategiak, memoria gune edo datu ez aldagarriak

Motibazioa

Baliabide **ez partekagarriak**:

- Aldi berean prozesu batek bakarrik atzitu ditzake
- Idazteko soilik irekitako fitxategiak, teklatua, aldagai globalak, inpresora ...

Komunikaziorako eta sinkronizaziorako mekanismoak

Komunikatzeko mekanismo ezberdinak daude:

- Seinaleen bidez (ikusita)
- Fitxategien bidez (ikusita)
- Fitxategi bereziak: Tutuak (prozesuak lotzeko adib. ls | less)
- Aldagai partekagarriak erabiliz =Memoria partekagarria (ikusteko)
- ...

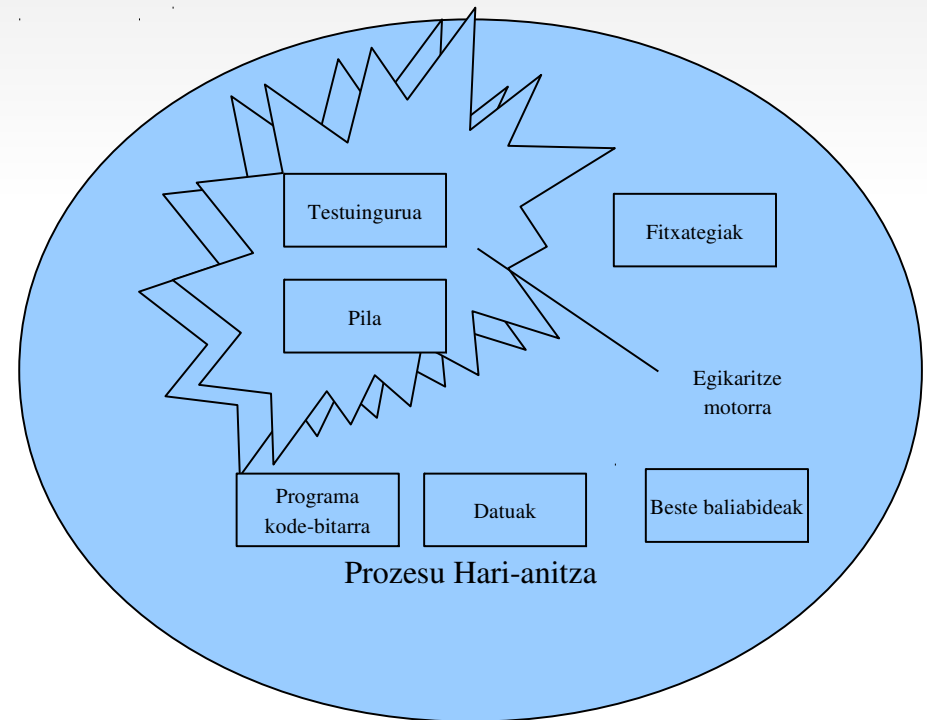
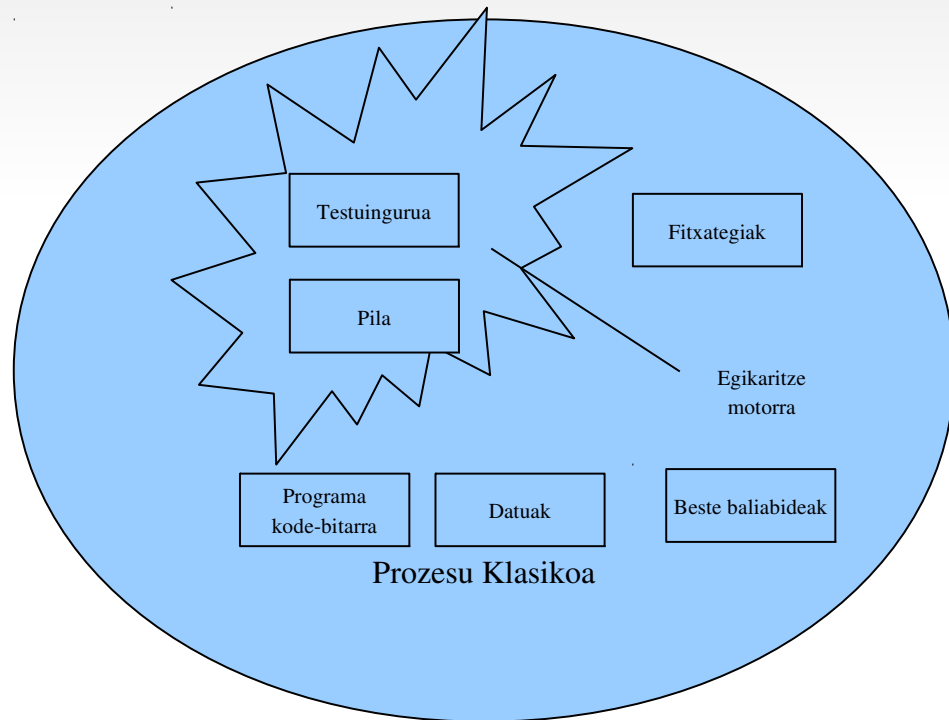
Aldagai partekagarriak.

Hariak vs Prozesuak

- Prozesua hurrengo ingurune konputazionallean egikaritzen da:
 - Testu segmentua: programaren kode bitarra.
 - Datuen segmentua: aldagai globalak eta estatikoak
 - Zabaldutako fitxategiak eta beste baliabide batzuk: inpresora, teklatua eta abar.
 - Egikaritze-motorra:
 - Pila segmentua: hariko/prozesuko funtzio bati deitu ostean, pila-pilegitura bat sartzen da funtzioak erabiltzen dituen aldagai lokal eta parametroen informazioarekin.
 - SEko testuinguru informazioa: SE-ak hariak/prozesuak kudeatzeko: PUZ erregistroen egoera eta programa kontagailua (egikaritutako azken agindua zein den) gordeko du.

Aldagai partekagarriak.

Hariak vs Prozesuak



Aldagai partekagarriak.

Hariak vs Prozesuak

- Gaur egungo SE-etan prozesuek hari-anitz izan ditzake.
- Hari-anitzeko prozesuetan, hari ezberdinak kode-bitarra, aldagai globalak eta estatikoak, fitxategiak, eta hainbat baliabide konpartitzen dituzte, baina hari bakoitzak bere pila (aldagai lokalak, funtzio parametroak ...) eta bere testuingurunea erabiltzen ditu (PUZ erregistroak, kontadore programa etab.)
- Prozesu klasikoa: Prozesu bakoitza hari bakarra da, hau da, ez du ezer partekatzen.

Aldagai partekagarriak.

Hariak vs Prozesuak

- Hari-anitzeko prozesu bat izanez gero sistema multiprogramatu batean bi hari edo gehiago une berean egikaritzen egon ahal dira, nahiz eta datu eta kode bera egikaritu. PUZ bat erabiltzen bada denboran txandakatuko dira, baina bi PUZ edo gehiago izanez gero paraleloki egikarituko dira, bakoitza PUZ batean.

Aldagai partekagarriak.

Hariak vs Prozesuak

- Hariak, prozesuak baino arinago sortzen eta bukatzen dira. Ez da beharrezkoa, baliabideak esleitzea, guztiak konpartitzen baitituzte.
- PUZ bat egonez gero harien txanda aldaketa azkarragoa da, soilik pila eta testu-ingurunea gorde behar baitira.
- Hariak memoria gune bera konpartitzen dute. Eta euren arteko komunikazioa errazten da. Adibidez: aldagai globalak erabil ditzakegu.
- Hariak ikusteko: `ps -eLf` (-e Select all processes eta -f When used with -L, the NLWP (number of light-weight processes) and LWP (light-weight process ID) columns will be added.

Aldagai partekagarriak.

Hariak vs Prozesuak

- `pthread_create`: hari bat sortzeko. Haria func funtzioarekin hasten da eta parametroak args-en bidez:
`int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*rutina)(void *), void *arg)`
 - `pthread_t * thread`: **hariaren helbidea**
 - `const pthread_attr_t *attr`: hari-atributoen egituraren helbidea (NULL defektuz)
 - `void *(*start_routine)(void *)`: funtzioaren helbide
 - `void *arg`: argumentuaren helbidea(asko egituraren bidez)
- `pthread_join (thread_id)`: Lo geratzen da `thread_id` haria bukatu arte.
- `pthread_exit (status)`: haria bukatzeko.

Thread

- Suppose we have to declare integer pointer, character pointer and float pointer then we need to declare 3 pointer variables.
- Instead of declaring different types of pointer variable it is feasible to declare single pointer variable which can act as integer pointer, character pointer.

```
void *ptr; // ptr is declared as Void pointer
```

```
char cnum;
```

```
int inum;
```

```
float fnum;
```

```
ptr = &cnum; // ptr has address of character data
```

```
ptr = &inum; // ptr has address of integer data
```

```
ptr = &fnum; // ptr has address of float data
```

Aldagai partekagarriak.

Hariak vs Prozesuak

```
#include <stddef.h>

#include <stdio.h>

#include <pthread.h>

#include <stdlib.h>

int sum=0;

void * process(void * arg){

int * num;

num=(int *)arg;

sum=sum+*num;

pthread_exit(0);

}
```

```
int main(){

pthread_t th_a, th_b;

int arg1=100,arg2=200;

pthread_create(&th_a, NULL, process, (void*)(&arg1));

pthread_create(&th_b, NULL, process, (void*)(&arg2));

pthread_join(th_a, NULL);

pthread_join(th_b, NULL);

printf("%d ", sum);

exit(0);

}
```

```
gcc -pthread -o hariak2 hariak2.c
```

Aldagai partekagarriak.

Hariak vs Prosesuak

```
#include <stddef.h>
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
int sum=0;
void * process(void * arg)
{
    int i; int * num;
    num=(int *)arg;
    for (i= 1;i<=*num;i++)
        {sum=sum+1;}
    pthread_exit(0);
}
```

```
int main(){
    pthread_t th_a, th_b;
    int arg1=10000,arg2=20000;
    pthread_create(&th_a, NULL, process, (void*)
        (&arg1));
    pthread_create(&th_b, NULL, process, (void*)
        (&arg2));
    pthread_join(th_a, NULL);
    pthread_join(th_b, NULL);
    printf("%d ", sum);
    exit(0);
}
```

Aldagai partekagarriak. Hariak vs Prozesuak

```
gcc -pthread -o hariakzenbatzen2 hariakzenbatzen2.c
```

```
./hariakzenbatzen2
```

22905

Zergatik?

bi hari erabiliko ditugu lerro-kopurua zenbatzeko, P1 hariak 1etik 10000ra eta P2 1etik 20000ra:

SK konpartitu ezin den kodea hurrengo hau izango da: `sum=sum+1`

Bi prozesuak momentu berean SK-an sartuz gero, batuketaren bat galdu ahal da. Nahiz eta pentsatu agindu hau atomikoa dela, ez da horrela:

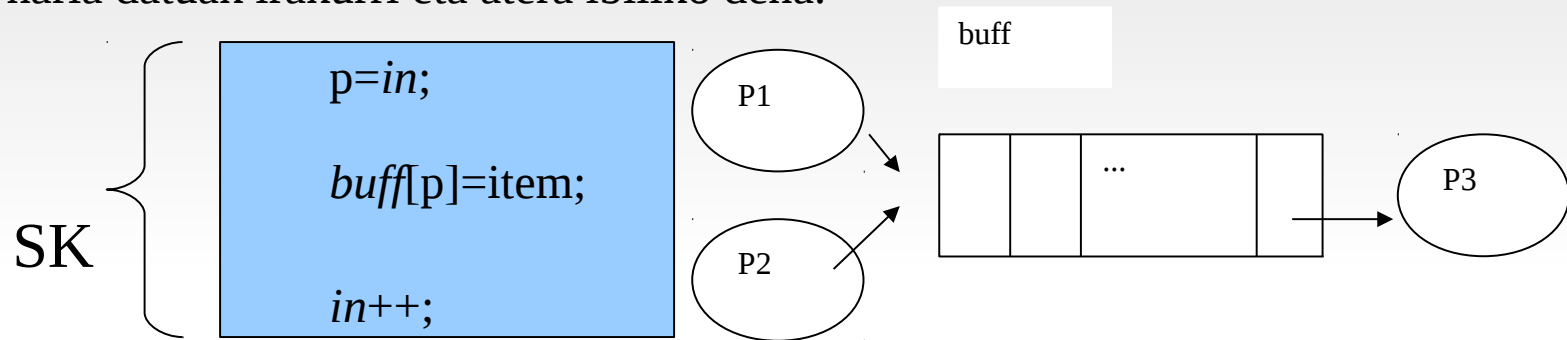
```
mov sum regX;
```

```
inc regX;
```

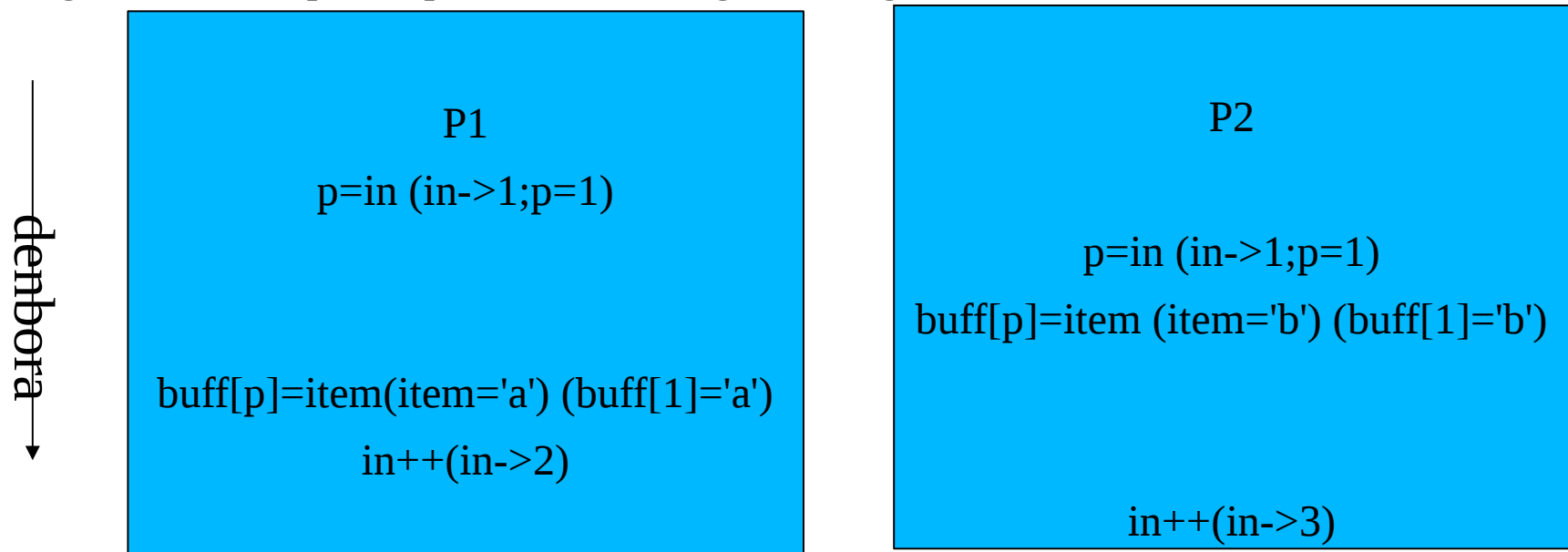
```
mov regX sum;
```

Aldagai partekagarriak. Sekzio kritikoa

- Sekzio kritikoa: Zenbait prozesu konpartitutako baliabideak atzitzen duen kodea. Adb: Hurrengo prozesu honek bi aldagai global eta partekagarriak ditu, "buff" eta "in". Programa honek bi hari ditu p1 eta p2 buff bufferrean datuak sartzen ibiliko direnak. Eta p3 haria datuak irakurri eta atera ibiliko dena.



- Zer gertatuko zen p1 eta p2 era honetan egikaritzuz gero?



Aldagai partekagarriak.

Sekzio kritikoa

Adb: Telemaratoia, teleoperadore bakoitzak exekutagarri bat du, eta datu-basea konpartitzen dute.

BEGIN WORK*

Select dinero_acum,num_llamada from SOLIDARIDAD;

dinero=dinero_acum+dfdinero

numllamada=num_llamada+1

update SOLIDARIDAD set dinero_acum
=dinero,num_llamada=numllamada

COMMIT*

*A COMMIT statement in SQL ends a transaction within a relational database management system (RDBMS) and makes all changes visible to other users. The general format is to issue a BEGIN WORK statement, one or more SQL statements, and then the COMMIT statement. Alternatively, a ROLLBACK statement can be issued, which undoes all the work performed since BEGIN WORK was issued.

Aldagai partekagarriak.

Sekzio kritikoa

- Exekuzioaren arabera emaitzak ezberdinak ez izateko, SK zein den, detektatu behar da. Eta sarrera eta irteera protokolo bat ezarriko dugu. Hurrengo propietateak aztertuz:
 - Elkarrekiko esklusioa
 - Progrezio finitua
 - Itxarote mugatua
 - Elkar-blokeaketarik ez

Aldagai partekagarriak.

Sekzio kritikoa

Elkarrekiko esklusioa:

- Prozesu bat baino gehiago, ezin da SKan aldi berean egon. k graduko baliabidea partekagarria baldin bada, k prozesu baino gehiago ezin izango dira aldi berean bere SKan egon. Adibidez, `p=in; buff[p]=item; in++;` SK bat da. Hori esan nahi du prozesu batek instrukzio hauek egikaritzen duen bitartean, ezin dela besteren bat sartu hauek egikaritzera.

Progrezio finitua:

- SKan ez badago prozesurik, SKan sartu nahi direnen artean erabaki behar da, ze prozesu sartu behar den. Sartu nahi ez dutenak ezin dute erabakian parte hartu. Gainera erabaki hori denbora finitu batean gertatu behar da.

Aldagai partekagarriak.

Sekzio kritikoa

Itxarote mugatua: Prozesu bat ezin da denbora luzez itxaroten egon SKan sartzeko.

Elkar-blokeaketarik ez: bi prozesu $p1$ eta $p2$ -en artean elkar-blokeaketa ematen da, baliabide batzuk partekatzen eta hauek lortzeko leiatzerakoan, bata eta bestea ezin jarraituta geratzen direnean, $p1$ -ek $p2$ behar duen baliabidea duelako eta $p2$ -k $p1$ -ek behar duena duelako, hori dela eta biak ezin aurrera jarraituta geratuko dira.

Komunikatzeko metodoak

- Sekzio kritikoa= kode zati bat da, atzipen eksklusiboa duen baliabide partekatu kudeatzeko, hau da, kode zati hori momentu berean ezin dena elkarbanatu k prozesu baino gehiagoekin. Gehienetan $k=1$ izango da.
- SKaren kodea egikaritzeko orduan, momentu oro prozesu bakarra dagoela kontrolatzeko, SK sartu baino lehen, sarrera-protokolo bat eta SKtik irten ostean irteera-protokolo bat ezarriko dugu.

sarrera-protokoloa

SK

irteera-protokoloa

- Sarrera eta irteera protokoloak definitzerako orduan metodo ezberdinak erabil ditzakegu:
 - Itxarote aktiboa (inkesta)
 - Blokeatzearen bidezko itxarotea: Semaforoak

Komunikatzeko metodoak

- Protokoloak hurrengo usteetarako soilik erabiliko ditugu:
 - PUZ bat (taskset -c 1 ./exekutagarria)
 - Bi hari aldi berean egikaritzen.
 - Bi hariak SK partitu egiten dute.
 - Testuingurune aldaketa edozein momentutan gerta daiteke.
 - Hariak ondo bukatuko dira (errorerik ez dela gertatzen).
 - Nahiz eta hari bat bukatu, bestea jarraitu ahal du hainbat aldiz SKan sartzen.

Komunikatzeko metodoak

itxarote aktiboa: Lehenengo proposamena:

Itxarote aktiboa: Hariak, bere txanda den edo ez jakiteko, behin eta berriro, etengabe, aldagai baten balioaz galdetuko du. VAB aldagai globala erabiliko da. VAB=i, izanez gero Pi hariak aurrera egin dezake. Adibidez: int VAB=1 bezala hasieratuz gero

```
i=1;//P1  
  
while (i<5){  
  
    //sarrera-protokoloa_1:  
    while (VAB==2) {NOP;}  
  
    SK  
  
    //irteera-protokola_1:  
    VAB=2;  
  
    i++;}
```

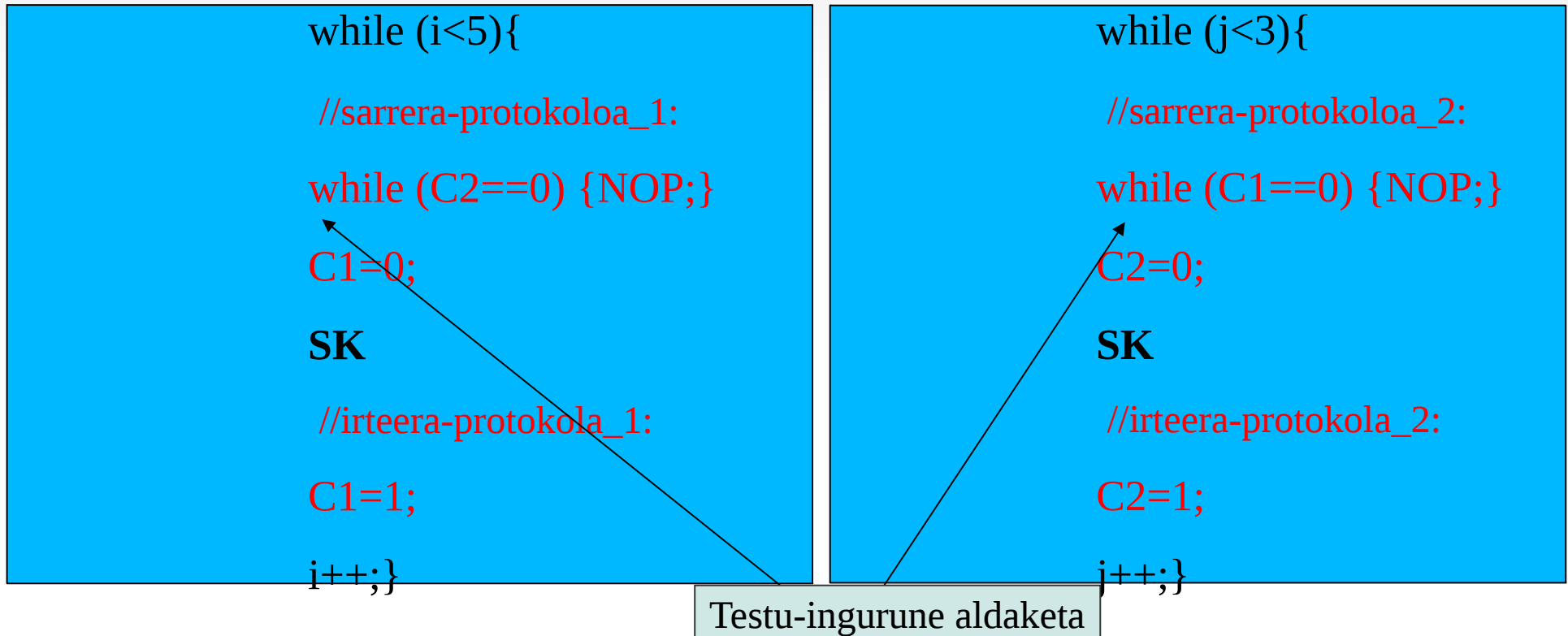
```
j=1;//P2  
  
while (j<3){  
  
    //sarrera-protokoloa_2:  
    while (VAB==1) {NOP;}  
  
    SK  
  
    //irteera-protokola_2:  
    VAB=1;  
  
    j++;}
```

- Arazoa: Progrezio finitua ez da betetzen, hau da, bigarren prozesuak $j < 3$ daukana seguraski arinago bukatuko du, eta prozesu hau bukatu ostean ezin izango du besteak SK-an sartu.

Komunikatzeko metodoak

Itxarote aktiboa: Bigarren prosamena:

Hari bakoitzak (P1, P2) aldagai propio bat erabiltzen du. C1=0 baldin bada SK sartzeko txanda P1ena izango da, eta C2=0 izanez gero P2rena dela esan nahi du. Hasieran C1 eta C2 aldagai globalak 1era hasieratzen dira.



Arazoa:Elkarrekiko esklusioa ez da betetzen. Gertatu ahal da C1=0 edo C2=0 markatu baino lehen testuingune aldaketa gertatzea, eta biak SKan sartzea.

Komunikatzeko metodoak

itxarote aktiboa: Hirugarren proposamena

Aurreko proposamenaren aldagai berak erabiliko ditugu, baino oraingoan sarrerako_protokoloan lehenengo eta behin Ora jarriko dugu aldagaia eta gero galdetuko dugu. Hau da, P1 SKan sartu nahi izanez gero $C1=0$ jarriko du. Eta gero galdetuko du, bestea sartu nahi den “while”aren bitartez. $C1$ eta $C2$ 1era hasieratzen dira.

```
while (i<5){  
    //sarrera-protokoloa_1:  
    C1=0;  
    while (C2==0) {NOP;}  
    SK  
    //irteera-protokola_1:  
    C1=1;  
    i++;}
```

```
while (j<3){  
    //sarrera-protokoloa_2:  
    C2=0;  
    while (C1==0) {NOP;}  
    SK  
    //irteera-protokola_2:  
    C2=1;  
    j++;}
```

Testu-ingurune aldaketa

Arazoa:bi prozesuen arteko elkar-blokeaketa gertatu ahal da, biak $C1=0$ eta $C2=0$ jartzea, eta ezin denez euren balorea 1 jarri “while”aren barruan blokeatuta geratuko dira biak.

Komunikatzeko metodoak

itxarote aktiboa: Laugarren proposamena

Aurreko proposamenaren aldagai berak erabiliko ditugu, eta lehenengo eta behin 0ra jarriko dugu aldagaia eta gero galdetuko dugu. Baina proposamen honetan NOP agindua jarri beharrean, elkar-blokeoa ekiditzeko $C1=1;C1=0$ aginduak jarriko ditugu. Horrela $C1=1$ aginduen ostean testuingurune aldaketa gertatuz gero elkar-blokeoa ekiditzeko. $C1$ eta $C2$ 1 hasieratuko ditugu.

```
while (i<5){  
    //sarrera-protokoloa_1:  
    C1=0;  
    while (C2==0) {C1=1;C1=0;}  
    SK  
    //irteera-protokola_1:  
    C1=1;  
    i++;}
```

```
while (j<3){  
    //sarrera-protokoloa_2:  
    C2=0;  
    while (C1==0) {C2=1;C2=0;}  
    SK  
    //irteera-protokola_2:  
    C2=1;  
    j++;}
```

Arazoa: Itxarote mugatua ez betetzea gerta daiteke. Adibidez $P1$ eta $P2$ “tandem”ean (agindu bat eta testuingurune aldaketa gertatzea) egikarituz gero mugagabeko atzerapena gerta daiteke. Proposamen hau ez du balio hegaldi edo taupada-markagailu baterako.

Komunikatzeko metodoak

itxarote aktiboa: soluzioa erakusteko hurrengo adibidea aurkeztuko dugu: bi hari erabiliko ditugu sum aldagai globalari bi balio banan-banan gehitzeko, p1 hariak 1tik 10000ra eta p2 1tik 20000ra:

SK konpartitu ezin den kodea hurrengo hau izango da: $sum = sum + 1$;

Bi prozesuak momentu berean SK-an sartuz gero, lerro batuketaren bat galdu ahal da. Nahiz eta pentsatu agindu hau atomikoa dela, ez da horrela:

```
mov sum regX;
```

```
inc regX;
```

```
mov regX sum;
```


Komunikatzeko metodoak

Denb	Haria	Aginduak	erreg1	erreg2	sum
1	P1	mv sum erreg1	0	0	0
2	P1	inc erreg1	1	0	0
3	P2	mv sum erreg2	1	0	0
4	P2	inc erreg2	1	1	0
5	P1	mv erreg1 sum	1	1	1
6	P2	mv erreg2 sum	1	1	1

Komunikatzeko metodoak

itxarote aktiboa: Dekker soluzioa:

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

void * process1(void * arg);
void * process2(void * arg);
int sum=0;

int turno=1;
int c1=1;
int c2=1;

int main(){
    pthread_t th_a, th_b;
    int arg1=10000,arg2=20000;
    pthread_create(&th_a, NULL, process1,(void *) (&arg1));
    pthread_create(&th_b, NULL, process2,(void *) (&arg2));

    /// wait all threads by joining them

    pthread_join(th_a, NULL);

    pthread_join(th_b, NULL);

    printf("batuketa: %d\n ", sum);
    exit(0);
}
gcc -pthread -o dekker dekker.c
```

Itxarote aktiboa: Dekker soluzioa

Haria 1:

```
void * process1(void * arg){
int *num;
int i;
num=(int *)arg;
for (i=1;i<=*num;i++)
{
//PE:
c1=0;
while (c2==0)
    {if (turno==2)
        {c1=1;
        while (turno==2);
        c1=0;
        }
    }
//SK:
sum=sum+1;
//PS:
c1=1;
turno=2;
}
pthread_exit(0);
}
```

Haria 2:

```
void * process2(void * arg){
int *num;
int i;
num=(int *)arg;
for (i=1;i<=*num;i++)
{
//PE:
c2=0;
while (c1==0)
    {if (turno==1)
        {c2=1;
        while (turno==1);
        c2=0;
        }
    }
//SK:
sum=sum+1;
//PS:
c2=1;
turno=1;
}
pthread_exit(0);
}
```

Komunikatzeko metodoak

itxarote aktiboa: Dekker soluzioa

Soluzio honetan “c1” eta “c2” 0 izanez gero, “turno” hirugarren aldagaia erabiliko da prozesu bietatik zein jarraituko duen esateko.

- **P1** SK-an sartu nahi denean “c1” 0ra jarri, eta “c2”ren baloreagatik galdetuko du, “c2” 0 izanez gero P2 ere sartzeko asmoa duela esan nahi du eta bietatik SKan zein sartuko den jakiteko “turno” aldagaian begiratzen du.
 - “turno” berdin 1 izanez gero P1 SKan sartuko zen.
 - “turno” berdin 2, izanez gero “c1=1” jarriko du, P2ri SK-an sartzeko aukera emateko eta elkar-blokeoa ekiditzeko; “turno” berdin 2 den bitartean P1 ez du ezer egingo. “turno” berdin 1 jarritz gero, “c1=0” jarriko da, eta P1 zuzenean SKan sartuko da.
- P1ek SKtik irteterakoan “turno=2” eta “c1=1” jarriko ditu.

Komunikatzeko metodoak

itxarote aktiboa: Dekker soluzioa

Elkarrekiko esklusioa: Probatu biak sartu nahi izanez gero, bakarra sartu ahal dela. BAI, biak sartu nahi izanez gero “turno”ren balorearen arabera, bat sartuko da eta bestea blokeatuta geratuko da, “turno” ezin delako momentu berean 1 edo 2 izan. Adibidez: P1 eta P2, “c1” eta “c2” Ora jarritz gero, eta “turno” berdin 2 baldin bada. Zer gertatuko zen? P1 “if”ko “then”aren barruan sartuko zen, “c1=1” jarritz, eta “turno=2” den bitartean NOP geratuko da. Orduan P2, kanpoko while egonez gero “c1=1” denez zuzenean SKan sartuko zen.

Komunikatzeko metodoak

itxarote aktiboa: Dekker soluzioa

Progrezio finitua: Hau probatzeko, hari batek bukatuz gero, besteak jarraitu ahal duen ikustea da.

Abibidez: P2k bukatzerakoan “c2=1” eta “turno=1” jartzen ditu, hori dela eta, P1 kanpoko “while”tik SKan sartu ahal da edo “turno=1”ekin barruko “while”tik atara, kanpoko “while”ra joan eta bertatik zuzenean SKan sartuko zan.

Komunikatzeko metodoak

itxarote aktiboa: Dekker soluzioa

Elkar-blokeoa: bi hari p_1 eta p_2 ren artean elkar-blokeoa ematen dela esaten dugu. Bi hariak baliabide batzuk partekatzen eta hauek lortzeko leiatzen direnean, hurrengo hau gerta denean, bata eta bestea ezin jarraituta geratzea, p_1 ek p_2 behar duen baliabidea duelako eta p_2 k p_1 ek behar duena, hori dela eta biak ezin aurrera jarraituta geratuko dira. Adibide honetan ezin dira elkar-blokeatuta geratu, “turno” aldagaiak ezin duelako bi balore ezberdin izan momentu berean. Beren balorearen arabera bata edo bestea SKan sartuko da.

Komunikatzeko metodoak

itxarote aktiboa: Dekker soluzioa

Itxarote mugatua: Prozesu bat ezin da denbora luzez itxaroten egon SKan sartzeko. Hau probatzeko ikusi behar da, ia prozesu batek denbora luzez egon ahal den SKan sartzen besteari utzi gabe. Soluzio honetan hau ez da gertatzen. Adibidez, P2k sartu nahi izanez gero eta P1 SKtik irten eta berriro sartu nahi izanez gero, zer gertatuko den aztertuko dugu. P1ek SKtik irtetzerakoan “c1=1” eta “turno=2” jartzen ditu. Orduan P2 kanpoko “while”an egonez gero zuzenean SKan sartuko da eta barruko “while”an egonez gero, bertatik irten “c2=0” jarri eta kanpoko “while”ra joango gara. P1 eta P2 kanpoko “while”an egonda; “c1” eta “c2” 0ra daude baina “turno=2” da, hori dela eta P1 barruko “while”an geratuko da eta P2 SKan sartuko da P1ek “c1=1” jartzen duen momentutik.

Komunikatzeko metodoak

itxarote aktiboa: Peterson soluzioa:

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

void * process1(void * arg);
void * process2(void * arg);
int sum=0;

int turno=1;
int c1=1;
int c2=1;

int main(){
    pthread_t th_a, th_b;
    int arg1=10000,arg2=20000;
    pthread_create(&th_a, NULL, process1,(void *) (&arg1));
    pthread_create(&th_b, NULL, process2,(void *) (&arg2));

    /// wait all threads by joining them

    pthread_join(th_a, NULL);

    pthread_join(th_b, NULL);

    printf("batuketa: %d\n ", sum);
    exit(0);
}
```

Komunikatzeko metodoak

itxarote aktiboa: Peterson soluzioa:

Haria 1:

```
void * process1(void * arg){
int *num;
int i;
num=(int *)arg;
for (i=1;i<=*num;i++)
{
//PE:
c1=0;
turno=2;
while ((c2==0)&&(turno==2));
//SK:
sum=sum+1;
//PS:
c1=1;
}
pthread_exit(0);
}
```

Haria 2:

```
void * process2(void * arg){
int *num;
int i;
num=(int *)arg;
for (i=1;i<=*num;i++)
{
//PE:
c2=0;
turno=1;
while ((c1==0)&&(turno==1));
//SK:
sum=sum+1;
//PS
c2=1;
}
pthread_exit(0);
}
```

Komunikatzeko metodoak

itxarote aktiboa: Peterson soluzioa:

```
$gcc -pthread -o peterson peterson.c
```

```
**$taskset -c 1 ./peterson
```

**En algoritmos con un fuerte dependencia al orden de ejecución de las instrucciones, ya que, cuando se trabaja con procesadores multicore por optimización, los procesadores utilizan un orden de ejecución débil (adelantando y atrasando ciertas instrucciones para optimizar en tiempo de procesador).

Komunikatzeko metodoak

itxarote aktiboa: Peterson soluzioa:

- P1-ek SK-an sartzeko $C1=0$ eta $turno=2$ jartzen ditu. Honela P2 sartu nahi izanez SK-an sartuko da.
- Momentu berean sartzeko ahaleginak eginez gero, “turno”ren balioak 1 eta 2 balioak ia momentu berean izango ditu. Baina soilik bat iraunduko du, besteak berehala galduko du bere balorea. Eta irauten duen “turno” baloreak erabakiko du zein sartuko den SK-an.

Komunikatzeko metodoak

- **Blokeo bidezko itxarotea: Semaforoak**
 - Itxarote aktiboan Prozesadore denbora xahutzen edo alferrigaltzen du. SK-rekiko itxarote-aldia laburra denean erabiltzen da. Adibidez: produzitzaile-kontsumitzaile buffer batekin. **Blokeo bidezko itxarotea:** SK-rekiko itxarote-aldia luzea denean erabiltzen da. Adibidez: disko S/I. Sistema honek prozesua lotan jartzen du, eta ez da esnatuko itxaroten(lotan) jarritako gertaera bukatu arte.

Komunikatzeko metodoak

-Blokeo bidezko itxarotea: Semaforoak

- **Semaforoa** bat, aldagai babestu bat da, non bere balioa solik **wait, signal eta initial** metodoen bitartez atzitu eta aldatu daitekeen.
- **Semaforo bitarrek** (mutex deritzonak) **soilik 0 eta 1 balioak** har ditzakete.
- **Semaforo-kontagailuek** berriz, **bat baino handiagoko zenbaki osoak** har ditzakete.

Komunikatzeko metodoak

Adibidez:

initial(s,1)

wait (s)

S.K

signal(s)

* wait,signal,initial agindu atomikoak dira.

Komunikatzeko metodoak

Wait(S) eragiketak, horrela egiten du lan:

$S = S - 1;$
 $\text{if } S < 0 \text{ then itxaron}(S);$ } Atomikoa da

* wait-ek prozesua blokeatuta jarriko du, eta S baloreari dagokion itxarote ilaran.

Signal(S) eragiketak, horrela funtzionatzen du:

$S = S + 1;$
 $\text{if } S \leq 0 \text{ then esnatu}(S);$ } Atomikoa da

*signal-ek, S baloreari dagokion itxarote ilaratik, prozesu bat atera eta esnatuko du.

initial(S,balorea):

Semaforoa balorea-ren edukinarekin jarri. Balorea momentu berean SK-an zenbat prozesu sartu ahal diren adierazten du. Adibidez: $\text{initial}(S,5)$ $S=5$ jartea bezala da.

Semaforoak

- `pthread_mutex_t mutex;` semaforo baten erazagupena
- `pthread_mutex_init(&mutex, NULL);` //init bezala, semaforaren erakuslea argumentu bezala eta semaforoaren ezaugarriak defektuz NULL(semaforo mutex arrunta)
- `pthread_mutex_lock(&mutex);` //wait bezala, semaforaren erakuslea argumentu bezala.
- `pthread_mutex_unlock(&mutex);` // signal bezala, semaforaren erakuslea argumentu bezala.

semaforoak

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

int sum=0;
pthread_mutex_t mutex;
void * process(void * arg){
    int * num;
    num=(int *)arg;
    int i;
    for (i= 1;i<=*num;i++)
    {
        pthread_mutex_lock(&mutex);
        sum=sum+1;
        pthread_mutex_unlock(&mutex);
    }
    printf("%d haria bukatuta\n", *num);
    pthread_exit(0);}
```

```
int main(){
    pthread_t th_a, th_b;
    pthread_mutex_init(&mutex,NULL);
    int arg1=10000,arg2=20000;
    pthread_create(&th_a, NULL, process,(void *)
        (&arg1));
    pthread_create(&th_b, NULL, process,(void *)
        (&arg2));
    /// wait all threads by joining them
    pthread_join(th_a, NULL);
    pthread_join(th_b, NULL);
    printf("batuketa: %d\n ", sum);
    exit(0);
}
```

```
gcc -o hariakzenbatzensemaforoekin
    hariakzenbatzensemaforoekin.c -lpthread
```

Programación en POSIX Threads (Pthreads)

Victor J. Sosa Sosa

1. Programación en POSIX Threads (Pthreads)

1.1. Introducción

La biblioteca de pthreads es una biblioteca que cumple los estándares POSIX y que nos permite trabajar con distintos hilos de ejecución (threads) al mismo tiempo.

La diferencia entre un thread y un proceso es que los procesos no comparten memoria entre sí, a no ser que se haya declarado explícitamente usando alguno de los mecanismos de IPC (InterProcess Communication) de Unix, mientras que los threads sí que comparten totalmente la memoria entre ellos. Además, para crear threads se usan las funciones de la biblioteca pthread o de cualquier otra que soporte threads mientras que para crear procesos usaremos la llamada al sistema fork(), que se encuentra en todos los sistemas unix.

Ya que pthreads es una biblioteca POSIX, se podrán portar los programas hechos con ella a cualquier sistema operativo POSIX que soporte threads. Ejemplos de ello son IRIX, los unix'es de BSD, Digital Unix OSF/1, etc.

1.2. 1.2. Como compilar un programa con pthreads

Para crear programas que hagan uso de la biblioteca pthreads necesitamos, en primer lugar, la biblioteca en sí. Esta viene en la mayoría de distribuciones linux, y seguramente se instale al mismo tiempo que los paquetes incluidos para el desarrollo de aplicaciones (es decir, cuando instalamos la libc o algún paquete tipo libc-devel)

Si no es así, o usas un sistema que no sea linux, la biblioteca no debería ser muy difícil de encontrar en la red, porque es bastante conocida y se suele usar bastante.

Una vez tenemos la biblioteca instalada, deberemos compilar el programa y "linkarlo" con la biblioteca dependiendo del compilador que estemos usando.

La forma más usual de hacer esto es, si estamos usando como compilador GNU gcc con el comando:

```
gcc programa_con_pthreads.c -o programa_con_pthreads -lpthread
```

Si por el contrario no estamos usando el compilador de GNU, lo mejor será que miremos la página man del compilador de C instalado en el sistema. Por ejemplo, en el caso del compilador de Digital para OSF/1, éste tiene un parámetro especial para compilar con pthreads:

```
cc programa_con_pthreads.c -o programa_con_pthreads -pthread
```

Como verás la sintaxis no es muy diferente pero el compilador de Digital no aceptará la de gcc y viceversa.

1.3. 1.3. Creación y manipulación de threads

Para crear un thread nos valdremos de la función `pthread_create` de la biblioteca y de la estructura de datos `pthread_t` que identifica cada thread diferenciándolo de los demás y que contiene todos sus datos.

El prototipo de la función es el siguiente:

```
int pthread_create(pthread_t * thread, pthread_attr_t *attr,  
void * (*start_routine)(void *), void *arg)
```

- **thread:** Es una variable del tipo `pthread_t` que contendrá los datos del thread y que nos servirá para identificar el thread en concreto cuando nos interese hacer llamadas a la biblioteca para llevar a cabo alguna acción sobre él.

- attr: Es un parámetro del tipo `pthread_attr_t` y que se debe inicializar previamente con los atributos que queramos que tenga el thread. Entre los atributos hay la prioridad, el quantum, el algoritmo de planificación que queramos usar, etc. Si pasamos como parámetro aquí NULL, la biblioteca le asignará al thread unos atributos por defecto (RECOMENDADO).
- start_routine: Aquí pondremos la dirección de la función que queremos que ejecute el thread. La función debe devolver un puntero genérico (`void *`) como resultado, y debe tener como único parámetro otro puntero genérico.
- La ventaja de que estos dos punteros sean genéricos es que podremos devolver cualquier cosa que se nos ocurra mediante los castings de tipos necesarios.
- Si necesitamos pasar o devolver más de un parámetro a la vez, se puede crear una estructura y meter allí dentro todo lo que necesitemos. Luego pasaremos o devolveremos la dirección de esta estructura como único parámetro. (ver código de ejemplo)
- arg: Es un puntero al parámetro que se le pasará a la función. Puede ser NULL si no queremos pasarle nada a la función.
- En caso de que todo haya ido bien, la función devuelve un 0 o un valor distinto de 0 en caso de que hubo algún error.

Una vez hemos llamado a esta función, ya tenemos a nuestro(s) thread(s) funcionando, pero ahora tenemos dos opciones: esperar a que terminen los threads, en el caso de que nos interese recoger algún resultado, o simplemente decirle a la biblioteca de pthreads que cuando termine la ejecución de la función del thread elimine todos sus datos de sus tablas internas.

Para ello, disponemos de dos funciones más de la biblioteca: `pthread_join` y `pthread_detach`.

```
int pthread_join(pthread_t th, void **thread_return)
```

- Esta función suspende el thread llamante hasta que no termine su ejecución el thread indicado por th. Además, una vez éste último termina, pone en thread_return el resultado devuelto por el thread que se estaba ejecutando.
- th: Es el identificador del thread que queremos esperar, y es el mismo que obtuvimos al crearlo con `pthread_create`.
- thread_return: Es un puntero a puntero que apunta (valga la redundancia) al resultado devuelto por el thread que estamos esperando cuando terminó su ejecución. Si este parámetro es NULL, le estamos indicando a la biblioteca que no nos importa el resultado.
- Devuelve 0 en caso de todo correcto, o valor diferente de 0 si hubo algún error.

```
int pthread_detach(pthread_t th)
```

- Esta función le indica a la biblioteca que no queremos que nos guarde el resultado de la ejecución del thread indicado por th. Por defecto la biblioteca guarda el resultado de ejecución de todos los threads hasta que nosotros hacemos un `pthread_join` para recoger el resultado.
- Es por eso que si no nos interesa el resultado de los threads tenemos que indicarlo con esta función. Así una vez que el thread haya terminado la biblioteca eliminará los datos del thread de sus tablas internas y tendremos más espacio disponible para crear otros threads (IMPORTANTE)
- th: Es el identificador del thread
- Devuelve 0 en caso de que todo haya ido bien o diferente de 0 si hubo error.

Hasta ahora hemos estado hablando de devolver valores cuando un thread finaliza, pero aún no hemos dicho como se hace. Pues bien, para ello tenemos la función `pthread_exit`

```
void pthread_exit(void *retval)
```

- Esta función termina la ejecución del thread que la llama.
- `retval`: Es un puntero genérico a los datos que queremos devolver como resultado. Estos datos serán recogidos más tarde cuando alguien haga un `pthread_join` con nuestro identificador de thread.
- No devuelve ningún valor.

Con todo lo que hemos visto hasta ahora ya estamos preparados para hacer nuestro primer programa con pthreads. El programa de ejemplo creará `MAX_THREADS` threads que ejecutarán la función `funcion_thr`.

Esta función sacará un mensaje por pantalla del tipo "hola, soy el thread número x", donde x será un número diferente para cada thread.

Para pasar esos parámetros a la función usaremos un struct del C, donde meteremos la cadena que debe imprimir cada thread más su identificador. (La cadena la podríamos haber metido directamente dentro de la función, pero así veremos como se pasa más de un parámetro al thread)

Una vez termina su ejecución, el thread devolverá como resultado su identificador (codificado en un entero), que será imprimido por pantalla por el thread padre que esperará que todos los threads terminen.

```
/** <CORTAR AQUI > */
/** Archivo ej1.c *****/
/* Creamos MAX_THREAD threads que sacan por pantalla una cadena y su
   identificador
   Una vez terminan su ejecución devuelven como resultado su
   identificador */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_THREADS 10
// tabla con los identificadores de los threads
pthread_t tabla_thr[MAX_THREADS];
// tipo de datos y tabla con los parametros
typedef struct {
    int id;
    char *cadena;
} thr_param_t;
thr_param_t param[MAX_THREADS];
// tenemos que crear una tabla para los parámetros porque los pasamos
por
// referencia. Así, si solo tuviéramos una variable para los
parámetros
// al modificar esta modificaríamos todas las que habíamos pasado
anteriormente
// porque los threads no se quedan con el valor sino con la dirección
void *funcion_thr(thr_param_t *p)
{
    // Esta es la funcion que ejecutan los threads
    // como veras, no tiene mucho secreto...
    printf("%s %d\n", p->cadena, p->id);
    // una vez terminamos, devolvemos el valor
    pthread_exit(&(p->id));
}
```

```

int main(void)
{
    int i, *res;
    // creamos los threads
    printf("Creando threads...\n");
    for (i=0; i<MAX_THREADS; i++) {
        param[i].cadena = strdup("Hola, soy el thread");
        param[i].id      = i;
        pthread_create(&tabla_thr[i], NULL, (void *)&funcion_thr,
            (void *)&param[i]);
    }
    // esperamos que terminen los threads
    printf("Threads creados. Esperando que terminen...\n");
    for (i=0; i<MAX_THREADS; i++) {
        pthread_join(tabla_thr[i], (void *)&res);
        printf("El thread %d devolvio el valor %d\n", i, *res);
    }
    // sacamos el mensajito y salimos del programa
    printf("Todos los threads finalizados. Adios!\n");
    return 0;
}
/*** <CORTAR AQUI> ***/

```

Para compilar (bajo Linux y gcc): gcc ej1.c -o ej1 -lpthread

El ejemplo en sí es MUY tonto, pero es el esquema básico que siguen todas las aplicaciones que lanzan threads para realizar un cálculo y esperan su resultado:

1. Crear el/los thread(s)
2. Esperar que terminen
3. Recoger y procesar el/los resultado(s)

Esto es un ejemplo de lo que se llama paralelismo estructurado.

Un ejemplo de un programa que use la función `pthread_detach` podría ser el de un servidor (de cualquier cosa: de correo, de http, de ftp, etc) que cree un hilo para cada petición que reciba. Como que no nos interesa el resultado de la ejecución, una vez hayamos creado el thread llamaremos la función `pthread_detach`.

Esto es lo que se conoce por paralelismo no estructurado. Es decir, nuestros programas no siguen una estructura concreta sino que se van ramificando según nuestras necesidades.

1.4. Otras funciones útiles de la biblioteca pthreads

Hasta ahora hemos visto las funciones más básicas para tratar con pthreads, pero aún queda alguna otra función útil:

`pthread_t pthread_self(void)`

- Esta función devuelve al thread que la llama su información, en forma de variable del tipo `pthread_t`.

Es útil si el propio thread que se está ejecutando quiere cambiar sus atributos, hacerse él mismo un `pthread_detach`, etc.

- Devuelve el identificador del thread. Ejemplo:

```

#include <pthread.h>
...
void *funcion_threads(void *param)
{
    pthread_t yo_mismo;
    ...
    /* nosotros mismos nos hacemos el detach */
    yo_mismo = pthread_self();
    pthread_detach(yo_mismo);
}

```

```

    ...
}
int main(void)
{
    ...
}
int pthread_kill(pthread_t thread, int signo)

```

- Envía un signal especificada al thread especificada. Un signal útil de enviar puede ser el SIGKILL, o alguno de los definidos por el usuario, SIGUSR1 y SIGUSR2.
- Aunque pueda parecer útil a primera vista, la única utilidad que tiene es matar un thread desde el proceso padre. Si se quiere usar con fines de sincronización hay formas mejores de hacerlo tratándose de threads: mediante semáforos y variables de condición (enseguida lo veremos)
- thread: identifica el thread al cual le queremos enviar el signal.
- signo: número de la señal que queremos enviar al thread. Podemos usar las constantes definidas en /usr/include/signal.h
- Devuelve 0 si no hubo error, o diferente de 0 si lo hubo.

Hasta aquí la primera parte del tutorial de programación en Pthreads. Aquí hemos visto las funciones básicas que nos ofrece la biblioteca para la creación, manipulación y eliminación de threads, pero aún nos quedan algunas cosas por ver.

2. 2. Problemas de concurrencia con Pthreads

2.1. 2.1. Introducción

Cuando decidimos trabajar con programas concurrentes uno de los mayores problemas con los que nos podremos encontrar, y que es inherente a la concurrencia, es el acceso a variables y/o estructuras compartidas o globales. Esto se entenderá mejor con un ejemplo:

<pre> Hilo 1 void *funcion_hilo_1(void *arg) { int resultado; ... if (i == valor_cualquiera) { ... resultado = i * (int)*arg; ... } pthread_exit(&resultado); } </pre>	<pre> Hilo 2 void *funcion_hilo_2(void *arg) { int otro_resultado; ... if (funcion_sobre_arg(*arg) == 0) { ... i = *arg; ... } pthread_exit(&otro_resultado); } </pre>
--	--

Este código, que tiene la variable 'i' como global, aparentemente es inofensivo, pero nos puede traer muchos problemas si se ejecuta en paralelo y se dan ciertas condiciones.

Supongamos que el hilo 1 se empieza a ejecutar antes que el hilo 2, y que casualmente se produce un cambio de contexto (el sistema operativo suspende la tarea actual y pasa a ejecutar la siguiente) justo después de la línea que dice if

(i==valor_cualquiera). La entrada en ese if se producirá si se cumple la condición, que suponemos que sí.

Pero justo en ese momento el sistema hace un cambio de contexto y pone a ejecutar al hilo2, que se ejecuta el tiempo suficiente como para ejecutar la línea i = *arg. Al poco rato hilo 2 deja de ejecutarse y vuelve a ejecutarse el hilo 1, pero, qué valor tiene ahora i? El que el hilo 1 está "suponiendo" que tiene (o sea, el mismo que comprobó al entrar en el if) o el que le ha asignado el hilo 2? La respuesta es fácil... ;-) i ha tomado el valor que le asignó hilo 2, con lo que el resultado que devolverá el hilo 1 después de sus cálculos será totalmente inválido e inesperado.

Claro que todo esto puede que no pasará si el sistema tuviera muy pocos procesos en ese momento (con lo cual cada proceso se ejecutaría por más rato) y si el código del hilo 1 fuera lo suficientemente corto como para no sufrir ningún cambio de contexto en medio de su ejecución... Pero NUNCA deberemos hacer suposiciones de éstas, porque no sabremos dónde se van a ejecutar nuestros programas y siempre más vale prevenir.

El problema que tienen estos bugs es que son los más difíciles de detectar en el caso que no nos fijáramos en que podría pasar una cosa de estas el día que escribimos el código. Puede que a veces vaya todo a la perfección y que otras salga todo mal... A esto se le conoce por Race Conditions (o en cristiano, Condiciones de Carrera) porque según como vaya la cosa puede funcionar o no.

2.2. 2.2. Mecanismos de Pthreads para prevenir esto

La biblioteca de Pthreads nos ofrece unos mecanismos básicos pero muy útiles para definir esto. Estos mecanismos son los llamados semáforos binarios, y se usan para implementar las llamadas regiones críticas (RC) o zonas de exclusión mutua (ZE).

Y qué es una RC? Pues una parte de nuestro código que es susceptible de verse afectada por cosas como la del ejemplo. Como regla general, SIEMPRE que haya variables o estructuras globales que vayan a ser accedidas por más de un thread a la vez, el acceso a éstas deberá ser considerado una región crítica, y protegido con los medios que vamos a explicar a continuación. Incluso si estamos seguros que solo un hilo va a acceder a una determinada estructura, no sería mala idea meter ese código en una RC porque tal vez en un futuro ampliemos nuestro código y no recordemos que teníamos esos accesos por ahí escondidos, con el consiguiente riesgo de bugs que ello conlleva.

Lo que Pthreads nos ofrece son los semáforos binarios, semáforos mutex o simplemente mutexs, como cada uno quiera llamarlo ;-) Un semáforo binario es una estructura de datos que actúa como un semáforo porque puede tener dos estados: o abierto o cerrado. Cuando el semáforo está abierto, al primer thread que pide un bloqueo se le asigna ese bloqueo y no se deja pasar a nadie más por el semáforo. Mientras que si el semáforo está cerrado, porque algún thread ya tiene el bloqueo, el thread que lo pidió parará su ejecución hasta que no sea liberado el susodicho bloqueo.

Solo puede haber un solo thread poseyendo el bloqueo del semáforo, mientras que puede haber más de un thread esperando para entrar en la RC, encolados en la cola de espera del semáforo. Es decir, los threads se excluyen mutuamente (de ahí lo de mutex para el nombre) el uno al otro para entrar.

Pues con una cosa tan sencilla en concepto se implementan las RC: se pide el bloqueo del semáforo antes de entrar, éste es otorgado al primero que llega, mientras que los demás se quedan bloqueados esperando a que el que entró primero libere el bloqueo o exclusión. Una vez el que entró sale de la RC, éste debe notificarlo a la biblioteca de pthreads para que mire si había algún otro thread esperando para entrar en la cola. Si lo había, le da el bloqueo al primero y deja que siga ejecutándose.

Las funciones que ofrece Pthreads para llevar esto a cabo son:

```
int pthread_mutex_init(pthread_mutex_t *mutex,
```



```
const pthread_mutexattr_t *attr)
```

- Esta función inicializa un mutex. Hay que llamarla antes de usar cualquiera de las funciones que trabajan con mutex.
- mutex: Es un puntero a un parámetro del tipo `pthread_mutex_t`, que es el tipo de datos que usa la biblioteca Pthreads para controlar los mutex.
- attr: Es un puntero a una estructura del tipo `pthread_mutexattr_t`, y sirve para definir qué tipo de mutex queremos: normal, recursivo o errorcheck (esto se verá más adelante)
- Si este valor es NULL (recomendado), la biblioteca le asignará un valor por defecto.
- La función devuelve 0 si se pudo crear el mutex o -1 si hubo algún error.

```
int pthread_mutex_lock(pthread_mutex_t *mutex)
```

- Esta función pide el bloqueo para entrar en una RC. Si queremos implementar una RC, todos los thread tendrán que pedir el bloqueo sobre el mismo semáforo.
- mutex: Es un puntero al mutex sobre el cual queremos pedir el bloqueo o sobre el que nos bloquearemos en caso de que ya haya alguien dentro de la RC.
- Como resultado, devuelve 0 si no hubo error, o diferente de 0 si lo hubo.

```
int pthread_mutex_unlock(pthread_mutex_t *mutex)
```

- Esta es la función contraria a la anterior. Libera el bloqueo que tuviéramos sobre un semáforo.
- mutex: Es el semáforo donde tenemos el bloqueo y queremos liberarlo.
- Retorna 0 como resultado si no hubo error o diferente de 0 si lo hubo.

```
int pthread_mutex_destroy(pthread_mutex_t *mutex)
```

- Le dice a la biblioteca que el mutex que le estamos indicando no lo vamos a usar más, y que puede liberar toda la memoria ocupada en sus estructuras internas por ese mutex.
- mutex: El mutex que queremos destruir.
- La función, como siempre, devuelve 0 si no hubo error, o distinto de 0 si lo hubo.

Estas son las funciones más básicas. Ahora, reescribiremos el pseudocódigo del ejemplo anterior con lo que hemos visto hasta ahora.

Variables globales: <pre>pthread_mutex_t mutex_acceso; int i;</pre>	
Hilo 1 (Versión correcta) <pre>void *funcion_hilo_1(void *arg) { int resultado; ... pthread_mutex_lock(&mutex_acceso); if (i == valor_cualquiera) { ... < resultado = i * (int)*arg; ... < } }</pre>	Hilo 2 (Versión correcta) <pre>void *funcion_hilo_2(void *arg) { int otro_resultado; ... if (funcion_sobre_arg(*arg) == 0) { ... pthread_mutex_lock(&mutex_acceso); i = *arg; pthread_mutex_unlock(&mutex_acceso); ... } }</pre>

<pre>pthread_mutex_unlock(&mutex_acceso); pthread_exit(&resultado); }</pre>	<pre>} pthread_exit(&otro_resultado); }</pre>
<pre>int main(void) { ... pthread_mutex_init(&mutex_acceso, NULL); ... }</pre>	

En color azul han sido añadidas las líneas que antes no estaban.

Como se puede ver lo único que hay que hacer es inicializar el semáforo, pedir el bloqueo antes de las RC y liberarlo después de salir de la RC, aunque a veces es cuestión también de tener un poco de vista.

Contra más pequeñas hagamos las RC, más concurrentes serán nuestros programas, porque tendrán que esperar menos tiempo en el caso de que haya bloqueos.

2.3. 2.3. Problemas... ¿Más problemas?

Aunque esto realmente soluciona el problema de los accesos concurrentes, también nos puede traer más problemas.

Y los problemas aquí también tienen nombre: los Deadlocks (o Abrazos Mortales) Los Deadlocks se producen cuando un hilo se bloquea esperando un recurso que tiene bloqueado otro hilo que está esperando un recurso. Si el recurso para el segundo thread no llega nunca, no se desbloqueará nunca, con lo cual tampoco se desbloqueará nunca el primer thread.

Resultado: nuestro fantástico programa bloqueado.

Solución? Pues aunque la biblioteca de Pthreads nos de algún mecanismo para intentar prevenir que esto se produzca, no hay ningún mecanismo fiable al 100% para prevenirlos.

El modelo más sencillo de Deadlock es el circular:

<pre>Hilo 1 void *funcion_hilo_1(void *arg) { ... pthread_mutex_lock(&mutex_1); ... pthread_mutex_unlock(&mutex_2); ... }</pre>	<pre>Hilo 2 void *funcion_hilo_2(void *arg) { ... pthread_mutex_lock(&mutex_2); ... pthread_mutex_unlock(&mutex_1); ... }</pre>
--	--

Parece un poco raro, pero puede llegarse a producir.

Mecanismos que ofrece la biblioteca Pthreads:

1. Semáforos recursivos

Estos semáforos solo aceptarán una sola petición de bloqueo por el mismo thread. Con los semáforos normales, si el mismo thread hace 10 llamadas a `pthread_mutex_lock` sobre el mismo semáforo, luego tendrá que hacer 10 llamadas a `pthread_mutex_unlock`, es decir, tantas como haya hecho a `pthread_mutex_lock`.

En cambio, los del tipo recursivo solo aceptarán una sola llamada a `pthread_mutex_lock`. Las siguientes llamadas serán ignoradas, con lo que ya eliminamos un tipo de deadlock.

Para poder crear un semáforo recursivo, tendremos que decírselo a `pthread_mutex_init`, indicándole como atributo el resultado de una llamada a `pthread_mutexattr_settype`. El procedimiento es:

- Definir una variable del tipo `pthread_mutexattr_t`:
`pthread_mutexattr_t mutex_attr;`
- Inicializarla con la llamada a `pthread_mutexattr_init`:
`pthread_mutexattr_init(&mutex_attr);`
- Indicarle el tipo explícitamente mediante `pthread_mutexattr_settype`:
`pthread_mutexattr_settype(&mutex_attr, tipo);`

Donde tipo puede ser `PTHREAD_MUTEX_NORMAL`, `PTHREAD_MUTEX_DEFAULT` (el que se usa por defecto), `PTHREAD_MUTEX_RECURSIVE` o `PTHREAD_MUTEX_ERRORCHECK`.

2. Probar antes de entrar

Si creemos que la siguiente llamada a `pthread_mutex_lock` va a ser bloqueante y que puede provocar un deadlock, la biblioteca de Pthreads nos ofrece una función más para comprobar si eso es cierto:

`pthread_mutex_trylock`.

`int pthread_mutex_trylock(pthread_mutex_t *mutex);`

- `mutex`: Es el mutex sobre el cual queremos realizar la prueba de bloqueo.
- La función devuelve `EBUSY` si el thread llamante se bloqueará o 0 en caso contrario. Si no se produce el bloqueo, la función actúa igual que `pthread_mutex_lock`, adquiriendo el bloqueo sobre el semáforo.

3. Funciones avanzadas

Otras funciones de uso avanzado con pthreads son las siguientes:

`int pthread_attr_init(pthread_attr_t *attr);`

`int pthread_attr_destroy(pthread_attr_t *attr);`

`int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);`

`int pthread_attr_getdetachstate(const pthread_attr_t *attr, int *detachstate);`

`int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);`

`int pthread_attr_getschedpolicy(const pthread_attr_t *attr, int *policy);`

`int pthread_attr_setschedparam(pthread_attr_t *attr, const struct sched_param *param);`

`int pthread_attr_getschedparam(const pthread_attr_t *attr, struct sched_param *param);`

Su utilización va más allá de los objetivos del curso y su estudio se deja al alumno

Aurkibidea

PCB, blokeatuta-prest, memoria babestua	2
A3 azterketa ereduko lehen galdera: exekutagarria moduluekin lerro batean ala askotan sortu?	2
Liburutegi dinamikoak sortzeko -fPIC flag-a erabili.....	3
A3 azterketa ereduko 2. eta 3. galderetarak hobeto ulertzeko hainbat azalpen	3
gcc-en erabili -lorokor eta ez liborokor.so edo liborokor.a.....	4
SIGKILL seinalearen funtzionalitatea ezinda berdefinitu	5
C-ko azterketa ereduko lehen ariketa: funtzioa eta modulua	5
Pakete/zerbitzuen... konprobaketak eta bash -x	6
Seinale tratatu gabeekin zer? alarn eta pause beti batera?	7
Prozesuetarako Linux API-a atalerako laguntzatxoa	9
Prozesuetarako Linux API-a atalerako laguntzatxoa	10
cron eta anacron.....	11

PCB, blokeatuta-prest, memoria babestua

Kaixo,

PCB

Process Control Block-a (PCB), datu egitura bat da, non sistema eragileak prozesu bati dagokion informazio guztia gordeko duen. Beraz, prozesu berri bat sortzen denean berarekin batera sistema eragileak PCB berri bat sortuko du. Beraz, prozesu bakoitza bere PCB-an gordetzen dela esan daiteke.

Non gordetzen da prozesu bat blokeatuta dagoenean? eta prest dagoenean?

Prozesu bat memorian ala prest egon, printzipioz, memorian kargatuta, gordeta, egongo da (ikus 5_1_multiprogramazioa.pdf-ko 4. gardenkiko diagrama). Gauza da, prest bezala markatuta badago, schedulerrak exekutatzera pasatzeko hautagai bezala hartuko duela.

Memoria babestua

Erabiltzaileak sistema eragilearen APIa erabiltzen du, alegia, erabiltzaileak sistema dei hau erabili nahi dut parametro hauekin adieraziko dio. EDT-a atzitzea jada kernel-modua da, kernelak kontrolatuko du. Horregatik, erabiltzaileak sistemaren baliabideak sistema deien bitartez atzitzeko ditu, ez nahieran, eta honela lortzen da sistema erabiltzailearen "baldarkerietatik" babestea.

Kodean zuzenean edo zeharka sistema dei bati dei egitean, EDT taulara joango da, sistema dei hori zehazki memoriako zein helbidetan dagoen kargatuta jakiteko. Makina bakoitzean, memorian kargatutako sistema deiaren zatiak, egoiliar errutina izena hartuko du. Gauza da, EDT taulan sarrerak gutxitzeko sistema deiak multzokatu egiten direla, sakabanatze errutinatan. Beraz, EDT taulan sakabanatze errutinari dagokion helbidea lortuko da eta honen parametroen bidez jakingo da ondoren zehazki zein egoiliar errutinari hots egin. Ikus 4-SistemaDeiak.pdf-ko 31. gardenkiko irudia.

Ondo izan,

Iñigo Perona Balda

A3 azterketa ereduko lehen galdera: exekutagarria moduluekin lerro batean ala askotan sortu?

Kaixo,

A3 azterketa ereduko lehen galdera: exekutagarria moduluekin lerro batean ala askotan sortu?

Bi moduak ontzat hartuko dira. Modulu bakoitzaren .o konpilazio-emaitzak banan bana sortzeak duen abantaila, gero, hurrengo azpi-atalean makefile-a sortzeko behar diren komandoak jada sortuak eta probatuak dituzuela da. Nik gomendatu, lerro askotan egitea gomendatuko nuke, make atalerako lan erdia aurreratzen delako.

Ondo izan,

Liburutegi dinamikoak sortzeko -fPIC flag-a erabili

Kaixo,

Liburutegi dinamikoa sortzeko .so-n enpaketatuko diren .o konpilazio-emaitzak -fPIC flag bidez sortuak izan behar dute, estatikoan ez bezala. Honela, .o Position Independent Code (PIC) bezala sortuko da, liburutegia ez baita exekutagarrian txertatuko, estatikoan ez bezala eta beraz, liburutegiko elementuek posizio independentzia behar dute exekutagarriko posizioekiko.

Kontuan izan, despiste hori behin baino gehiagotan ikusi dizuet eta.

Ondo izan,

A3 azterketa ereduko 2. eta 3. galderetarak hobeto ulertzeko hainbat azalpen

Kaixo,

[A3] Azterketa ereduko 2. galderako kodea ulertzeko hainbat azalpen:

"man 2 lstat" eginez ikus daiteke, lstat-ek lehen argumentu bezala fitxategi izena hartzen duela eta bigarren argumentu bezala stat motako erregistrora pointera.

"man 2 lstat" eginez RETURN VALUE aztertuz, 0 ondo badoa -1 arazoak egon badira. Beraz, -1 lortzen denean, fitxategia atzitzean arazoren bat egin den seinale.

"man 2 lstat"-etik baita ere, st_ino /* Inode number */ dela ikus daiteke. Inode zenbakia.

Beraz, programak, bi fitxategiek inode berdina duten ala ez esango du, beharrezko errore tratamenduak eginda.

Programa batek zenbat deskriptore zabalduko dituen galdetzen denean, kontuan izan defektuz beti prozesu orok 3 deskriptore irekita dituela: sarrera estandarra (stdin, 0), irteera estandarra (stdout, 1) eta erreentzako irteera (stderr, 2). Gogoratu berbideraketak egitean nola erabiltzen genituen 1>, 2> eta &>. Ondoren erabiltzen den fitxategi bakoitzeko honi deskriptore bat sortzen zaio, normalean honela: FILE *fd = open("fitx.txt", "rw"); Kasu honetan lstat-ek fitxategi izena jasotzen du parametro bezala ez deskriptorea. Bere bixkia den fstat-ek, aldiz, deskriptorea jasotzen du. Ikus "man 2 lstat". Hala ere, lstat-ek fitxategiak atzitu ahal izateko ezinbestean bere

barnean deskriptorea sortzen du. Beraz, programa honek 5 deskriptore irekitzen ditu. Hiru defektuzko eta beste bi argv[1] eta argv[2] fitxategiak atzitzeko. Erantzuna beraz bost da.

perror-i dagokionez, sistema deiek huts egiten dutenean, -1 balioa itzultzen dute, eta akatsaren zenbakia erro aldagaian gordetzen dute. perror() funtzioaren bitartez erro zenbakiari dagokion errore mezua pantailaratu dezakegu. Alegia, erro aldagaiak duen kodea, errore mezu irakurgarri batean pantailaratzen du perror-ek. Ikus man 3 perror.

[A3] Azterketa ereduko 3. galderako kodea ulertzeko hainbat azalpen:

for(;;) -> infinitorarte iteratzen geratzeko erabiltzen da. Gure kasuan iterazio infinitu hauek seinaleen bidez eteten saiatzen gara.

"kill -l" eginda ikus daiteke zein seinalerentzako berridazten den funtzionalitatea.

"man 7 signal" eginda, ikus daiteke seinale bakoitzaren besterik ezeko funtzioa:

SIGALRM 14 Term Timer signal from alarm(2)

SIGUSR1 30,10,16 Term User-defined signal 1

SIGUSR2 31,12,17 Term User-defined signal 2

Programan erabiltzen diren seinale guztiek besterik ezeko funtzionalitatea, prozesua akabatzea, terminatzea dute: Term. Baina 14a eta 10a birdefinitzen ditugu, beraz, ez dute besterik ezeko bere funtzioa exekutatuko, baizik eta guk birdefinitua. Alegia, ez dute Term exekutatuko.

Programaren jarraipena segundoz segundo ea hobeto ulertzen den, hala ere kodea kopiatu, konpilatu eta exekutatzea lagungarri izango zaizue:

- Exekuzioa hasi eta 3 segundora: umeak gurasoari 10 seinalea bidali
- Exekuzioa hasi eta 3 segundora: gurasoak 10 seinalea jaso trapper() exekutatu eta pause()-tik atera. Segidan umeak 12 seinalea bidali (Term).
- Exekuzioa hasi eta 3 segundora: umeak 12 jaso eta hilko da.
- Exekuzioa hasi eta 3+5 segundora: gurasoak 14 seinalea (SIGALRM) jasoko du, trapper exekutatu etta pause()-tik aterako da. Jada gurasoaren kill-ek ez du eraginik.
- Exekuzioa hasi eta 3+5+5 segundora: berdina, eta honela betirarte.

Ondo izan,

[gcc-en erabili -lorokor eta ez liborokor.so edo liborokor.a](#)

Kaixo,

gcc-ri liburutegiak adieraztean ez erabili liborokor.so edo liborokor.a formak, baizik eta erabili beti -lorokor forma. Liburutegiak adieraztean ez da liburutegiaren izen osoa erabiliko. Adibidez, liborokor.so beharrean -lorokor idatziko dugu. liborokor.so izenari hasierako lib eta bukaerako .so kenduko zaio. Ondoren liburutegia adierazteko -l (hau -l library-ri dagokio ez -l Include-ri) flag-ari itsatsita liburutegi izen murriztua ipiniko da.

Beti erabili modu hau, bai liburutegi dinamiko, bai estatikoetan, bestela liburutegiak uneko karpetan dagoenean bakarrik funtzionatzen du. **Laburbilduz, gcc-n liburutegiak adierazteko beti erabili behar den modua: -lorokor da.**

Ondo izan.

SIGKILL seinalearen funtzionalitatea ezinda berdefinitu

Kaixo,

Seinaleen atalean ikusi dugu, C kodea idaztean, prozesuak jasoko dituen seinaleen funtzionalitateak berridatzi ditzakegula. Seinaleari gure funtzio bat esleitzeko honela egiten da:

```
signal(SIGINT, trapper);
```

Seinaleen posibleen zerrenda: kill -l

Baina badira bi seinale, SIGKILL eta SIGSTOP, zeintzuen funtzionalitateak ezin diren berridatzi. Logika ere badu, bestela programa bat egin genezake inongo seinalek ere gelditu ezingo lukeena. Kode gaizto batek esplotatu lezakeen ahulgunnea. SIGKILL aukera ukiezina izanda, erabiltzaileari programa bat geratzeko ahalmena bermatzen zaio.

Izan ere, batzuk SIGKILL-en funtzionalitatea berdefinitzen saiatu zarete eta ez zizuen ezer egiten. Arrazoa jakin dezazuen.

Ondo izan,

C-ko azterketa ereduko lehen ariketa: funtzioa eta modulua

Kaixo,

C-ko atalaren azterketa adibideko lehen ariketarako laguntzatxo bat:

- Enuntziatutik kodea kopiatzean kakotxekin kontuz, batzuetan konpilatzaileak karaktere arraro bezala detektatzen baititu.

- Lehenik probatu KateLuzapena.c, behar diren include-ak ipiniz (osotu gabe dagoela horregatik dio). Alegia, string.h-ko strlen funtzioa erabili eta probatu kodea:

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

```
#include <string.h>
```

- Warning-a kentzeko, gets-en ordez fgets erabiltzeko iradokitzen du, gets deprekatuta baitago. Beno, hau ez da beharrezkoa baina. "man 3 fgets" egin, nola funtzionatzen duen ikusi eta, honela ordezkatzeko genezake:

```
fgets(kat, N, stdin);
```

- Norberak garatutako funtzioak ongi funtzionatzen duen ikusteko lehenik programa nagusia dagoen fitxategian probatu sortu duzuen kodea. Kasu honetan kendu `#include <string.h>`, ordezkatu `srtlen(kat)` `mystrlen(kat)`-gatik eta inplementatu `mystrlen` funtzioa bertan. Eta probatu kodea. Behin funtzionatzen duela, ekin modulua sortzeari.

- Eta funtzioa inplementatzen ez baduzue asmatzen, ez ataskatu puntu honetan, ipini funtzioaren gorputzean `printf("String luzera kalkulatu\n")` eta hurrengo pausuak egiten jarraitu.

- Behin funtzioak ondo egiten duela ziurtatuta sortu modulua: `fkateak.h` eta `fkateak.c` fitxategiak. Kontuan izan `#ifndef` definitzean fitxategiaren izena letra larriz jarri ohi dela:

```
vim fkateak.h
#ifndef _FKATEAK_H
#define _FKATEAK_H
int mystrlen(char kat[]);
#endif
```

- Gehitu moduluari dagokion goiburukoa programa nagusian, guk sortua denez kakotx bikoitzen artean:

```
#include "fkateak.h"
```

Lagungarri izango zaizuelakoan.

[Pakete/zerbitzuen... konprobaketak eta bash -x](#)

Zer konprobatu behar den eta zer ez

Irizpide orokor bezala eduki buruan, esandakoa automatizatu behar dela eta erabilterraza izango dela, erabiltzaile ez aditu batek exekutatzeko modukoa.

`menu.sh`-aren aukerak ordenean exekutatuko dira eta behin aukera bat exekutatu denean bera eta bere aurrekoak berriz exekutatzean arazorik ez luke eman behar.

Zerbait instalatzean edo sortzean, ea lehendik instalatua zegoen ala ez ikustea komeni da. Funtzioren batek pakete askoren instalazioa egiten badu, hauetatik nagusiena begiratzearekin nahiko, izan ere, gure script-ak, guztiak batera instalatu eta desinstalatuko ditu, blokean.

Zerbait testatzean, berau testatzeko behar den zerbitzua/zerbitzuak martxan dagoen/dauden begiratu da.

`netstat` komandoa begiratu egingo da instalatuta dagoenentz, erabiltzaileari

automatikoki instalatzea erabilerrazago egiten baitu.

ssh komandoa ez da automatikoki instalatuko. Hau zerbitzu ezaguna da eta instalatuta dagoela suposatuko da, baita bertan autentifikazio oker eta zuzenak eginak dituela ere.

Erroreak topatzeko laguntzatxoa

Script-a egiten ari dena jarraitzeko eta erroreak topatzeko ondo etor dakizueke. Exekutatu scripta modu honetan:

```
bash -x menu.sh
```

man bash-etik:

-x: Print commands and their arguments as they are executed.

Seinale tratatu gabeekin zer? alarm eta pause beti batera?

Zer gertatuko litzateke prozesu batek tratatu gabeko seinaleren bat jasoko balu?

Linux-eko prozesu batek, jaso ditzakeen 64 seinaleek (kill -l) dute defektuzko funtzioen bat lotuta. Beraz, prozesu batek dena delako seinalea jasotzen badu, defektuzko, besterik ezeko funtzioa exekutatu du. Defektuzko funtzio hauek "man 7 signal" egin eta bertako Signal-Value-Action-Comment tauletan ikus daitezke Action zutabearen. Akzio hauek Term, Ign, Core, Ign, Stop eta Cont. "man 7 signal"-ek honela definitzen ditu:

The entries in the "Action" column of the tables below specify the default disposition for each signal, as follows:

- Term: Default action is to terminate the process.
- Ign: Default action is to ignore the signal.
- Core: Default action is to terminate the process and dump core (see core(5)).
- Stop: Default action is to stop the process.
- Cont: Default action is to continue the process if it is currently stopped.

Beraz guk kodean seinale bat tratatzean, defektuzko akzioa gainidatziko dugu eta seinale hau jasotzean gure funtzioa exekutatu du da defektuzkoa beharrez.

alarm deiaren ondoren beti dator pause?

alarm ez doa beti pause batekin. Pause-k seinaleren bat jaso artean (SIGALRM izan daiteke edo beste edozein) prozesua pausan uzten du. Probak egiteko alarm-pause bikotea egoki datorkigu baina ez dute zertan batera joan. Alegia, alarm bat ezarri dezakegu eta gero begizta luze batean sartu adibidez. Kasu honetan, SIGALRM seinalearen bidez begizta horretan zein puntutan aurkitzen garen informatzeko programatu dezakegu. Adibidez, ondoko programak zenbaki lehenak aurkitzen ditu, bat aurkitutakoan segundu bateko sleep-a egin eta aurrera jarraitzen du, horrela lehen MAX_TAM lehenak aurkitu arte. SIGALRM seinalea begiztan zein puntutan gauden

inprimatzeko erabiliko da, alegia, aldagai globalen informazioa inprimatuko du.

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

#define MAX_TAM 10000

int g_i = 0;
int g_lehena = 2;

void nirealarm(int sig){
    printf("ALARM: %d zenbaki lehenean, %d-garreanean doa.\n", g_lehena, g_i);
    alarm(3);
}

int main(){
    int j, lehena, kont;
    signal(SIGALRM, nirealarm);
    alarm(3);
    while(g_i<MAX_TAM){
        kont = 0;
        for(j=2; j<g_lehena; j++){
            if(g_lehena%j==0){
                kont++;
                break;
            }
        }
        if(kont==0){
            g_i++;
            sleep(1);
        }
        g_lehena++;
    }
}
```

Beraz, alarm baten ondoren pause bat ez da derrigorra, adibide hau kasu.

Ea hobeto ulertzen laguntzen dizuen.

Zaindu,

Prozesuetarako Linux API-a atalerako laguntzatxoa

Kaixo,

8) Beheko programak emanda, azaldu `"/.padre 60 hijo"` deiak egiten duena:

padre-k reloj eta erabiltzaileak pasatzen dion programaren (kasu honetan hijo) arteko lasterketa moduko batean epaile lanak egiten ditu, non lehenaren, reloj-en exekuzio denbora erabiltzaileak finkatzen baitu (kasu honetan 60 segundo). Beraz, joku moduko honetan, erabiltzaileak lehen parametroan emaniko segundo kopurua bigarrenean emaniko programarena baino luzeagoa izatea lortu beharko da. Kasu horretan 0 itzuliko du, dena ondo joan dela adieraziz. Kontuan izan UNIX-eko komandoek dena ondo joan denean 0 itzultzen dutela, beraz konbenioa hau denez horrela suposatuko dugu. `"/.padre 60 hijo"` exekutatu ondoren `"echo $?"` egin dezakegu aurreko aginduak itzuli duen kodea ikusteko. Gero exekutatu `"/.padre 5 hijo"` eta jarraian `"echo $?"` eta ikusi zer pasatzen den.

Azalpen luzeagoa jarraian, argigarri izan daitekeelakoan:

reloj exekutagarriak parametrotzat hartzen duen segundo kopurura alarma ipintzen du eta seinaleren baten zain pause-n geratzen da. Beste seinalerik ezean SIGALRM helduko zaio eta berau tratatua duenez xtimer funtzioa exekutatu da.

hijo-k, beste seinalerik ezean 6 segundo itxaron eta amaitzen du.

padre-k bi fork egiten ditu. Lehenengo fork-agatik sortutako prozesu klonak (umeak) exec egitean reloj exekutagarriaz ordezkatu da. reloj-ek hasiera batean 60 segundo pasako ditu amaitzen, padre-ri deia ondokoa delako: `"/.padre 60 hijo"`. Bigarren fork-agatik sortutako klonak (umeak) exec egitean hijo exekutagarriaz ordezkatu da. hijo-k hasiera batean 6 segundo beharko ditu amaitzen.

padre-k bi ume sortu dituenaz, bi wait beharko ditu biei itxaron nahi badie. Eta nola daki zein prozesuk amaitu duen? bi wait-ek amaitu duen prozesu umearen PID-a itzultzen duelako. Beraz, lehenik hijo-k bukatzen badu (probableena, reloj-i 60 segundo pasatzen bazaio) reloj-i SIGKILL bidaltzen zaio eta akabatzen da. Lehenik reloj-ek amaitzen badu, guk beste terminal batetik kill komando bidez akabatzeko seinaleren bat bidali diogulako adibidez, hijo-ri SIGKILL bidaltzen zaio eta akabatzen du.

Azkenik prozesu umearen bizi denbora (hurbilpena behetik) kalkulatu eta inprimatzen da. exit kode bezala 1 itzultzen du lehenik reloj-ek amaitu badu eta 0 lehenik hijo-k amaitu badu. Izan ere, reloj eta hijo-k euren exekuzioa ondo joan bada 0 itzultzen dute eta padre-n status1 1-era lehenik reloj-ek amaitzen duen kasuan ipintzen da.

Ea argigarri zaizuen,

Prozesuetarako Linux API-a atalerako laguntzatxoa

Kaixo,

Erakutsi dizkidazuen zalantzetatik gauza batzuk argitzea komeni dela iruditzen zait eta honetarako, ariketa batzuen erantzunak baliatuko ditut. Erantzun asko azalpen zabalagoak emateko aprobeztatzen ditut, berez hain erantzun luzeak ez lukete behar.

1b) Zer egikaritzen da lehenago, gurasoa edo semea? Zergatik?

Umea. Gurasoak `wait(&status)` duelako, zeinaren bidez bere umeak bukatu arte zain geratuko baita.

Komentatu, `fork()`-ak bere programa klon bat sortuko duela exekuzio puntu berdinean, baina umeari zero itzuliko dio eta gurasoari `>0`, hau da, prozesu umearen PID-a. `if`-ren baldintzan ez bada konparaziorik egiten, bertako balioa `>0` bada `TRUE` eta `0` bada `FALSE`. Beraz konturatu, besterik ezean, `if`-aren bi adarrak exekutako direla paraleloki, adar bat gurasoak eta bestea umeak.

1c) Hurrengo aginduan, `wait(&status)`, zer da status aldagaian jasotzen dena?

`status`-ean gurasoak umearen egoera jasoko du eta egoera honen balioa umeak `exit` egitean itzultitako balioa izango da.

2.4) Aldatu aurreko `trapper.c` 1tik 9arteko seinaleak tratatzeko. Egikaritu ostean `SIGUSR1` seinalea bidatzeko. Zein da erabili duzun komandoa?

Kodean 64 zenbakia 9-gatik ordezkatzeari da ideia: `for(i=1;i<=9;i++)` egitea. Modu honetan lehen 9 seinaleak tratatzen dira (ikus `kill -l`) eta beste seinaleren bat jasoko balitz defektuzko funtzioa exekutatu luke. Beraz, prozesu batek, besterik ez bada adierazten, defektuz, `kill -l` -eko seinale guztientzat defektuzko funtzio bat du, askotan programa etetea izango dena.

Seinale bat tratatzea programaren hasieran seiale hau gure funtzio bati lotzea da: `signal(1, trapper)`. Gure funtzioaren barruan, `trapper`-en barruan, berriz `signal` azaltzen da baina hau sistema ezberdinen bateragarritasun kontuengatik da, gure sistemetan hori gabe ere funtzionatzen du, beraz ez eman garrantzirik, kodea irakurtzean ez balego bezala jokatu. Iruditu zait programaren ulermena nahastu egiten dizuela eta.

Kontuan izan baita ere, 64 direla erabili daitezkeen seinaleak, `kill -l` ageri direnak. Beraz, adibidez, 70. seinalea ez da existitzen eta beraz ezin da tratatu.

Beraz, `SIGUSR1` edo 10. seinalea bidaltzen dugunean aldaketaren ostean (1etik 9ra) tratatu ganeko seinalea izango da eta defektuzko funtzioa exekutatu da, kasu honetan programa etetea. Hau `kill` komandoaz egingo da, aukera ezberdinak ditugu 10. seinalea bidaltzeko:
`kill -s SIGUSR1 1234`

kill -10 1234
kill -USR1 1234

6. eta 7. ariketak) alarm-pause eta sleep

6. ariketan alarm(5) egitean hemendik 5 segundurako alarma ezartzen da, eta berehala pause()-ra pasako dela. Pausen seinale bat jaso zai geratuko da programa, 5 segundura jasoko duen seinalea SIGALRM izango da eta aurrera jarraituko du.

7. ariketan sleep(1) egiten denean, segundu batez geratzen da programa eta gero aurrera jarraituko du.

Konturatu alarm-pause bikoteaz eta sleep-ez kasu honetan efektu bera lortzen dugula, beti ere kodetik aparteko seinalerik ez dagoela suposatuz.

8. ariketa) execve

Konturatu execve sistema-deia dela, egin man 2 execve eta besteak bere front-end-ak, adibidez, man 3 execlp.

Konturatu execve agindua ongi burutzen bada bere ondorengo koderik ez dela exekutatuko. Kasu honetan, uneko programa mkdir programaz ordeztua baita, beraz, aurreko prozesuaren exekuzio-lerroa exec aginduarekin amaituko da eta beste programa batena kargatuko eta hasiko da. exec-en ondorengo lerroak programa ordezkapen hau egitean zerbait gaizki joan bada bakarrik exekutatuko dira.

Fijatu apunteetan, lehenik fork() egin eta gero umeari egiten zaiola exec. 12. gardenkiko irudiak ondo ilustratzen du egoera hau. 8. ariketako kasuan zuzenean programa nagusiari egiten zaio exec.

Ea laguntzen dizuen,

[cron eta anacron](#)

Kaixo,

Prozesuen kudeaketarako bash komandoen atalean, seigarren ariketan, cron eta anacron-ekin zailtasunak daudela dirudi eta hemen laguntzatxo bat:

c) hileko lehenengo bost egunetan, goizeko 9:00etan.

[crontab -e](#)

00 minututan

9 ordutan

1-5: hilabeteko lehen bost egunetan

*: edozein hilabetetan

```
# *: edozein asteko egunetan  
00 09 1-5 * * /home/lsi/backup.sh
```

d) orduro, abuztuko ostiraletan.

```
crontab -e  
# 00 minututan  
# *: edozein ordutan  
# *: edozein hilabeteko egunetan  
# 8: 8garren hilabetea, abuztuan  
# 5: ostiraletan  
00 * * 8 5 /home/lsi/backup.sh
```

Gogoratu cron-en berezitasuna, makina beti piztuta egongo dela suposatzen duela dela, zerbitzarietarako pentsatua, beraz adierazi unean egikarituko du adierazi komandoa. Makina itzalirik badago une horretan, piztean ez dira komando horiek berreskuratuko eta exekutatu gabe pasako dira.

e) Hilabetea behin exekuta dadin (man anacron).

```
# MODUA OROKORRA.  
vim /etc/anacrontab  
# aldia: @monthly  
# atzerapena (minututan): 15  
# lanaren identifikatzailea, log fitxategietan identifikatzeko, guk nahi duguna:  
anacron.hilabetero  
@monthly 15 anacron.hilabetero /home/lsi/backup.sh
```

e) Hilabetea behin exekuta dadin (man anacron).

```
# DEFETUZKO KARPETAK ERABILIZ: @hourly, @daily, @weekly eta @monthly denean  
defektuzko karpetak erabili daitezke (ls /etc/cron.*)  
vim /etc/anacrontab  
@monthly 15 anacron.hilabetero nice run-parts --report /etc/cron.monthly  
# nice: adierazi script/programari lehentasuna emateko, besterik ezean, ez bada ezer  
adierazten 10 lehentasuna emango dio.  
# run-parts: karpeta bateko programa eta script guztiak exekutatzen ditu  
# --report parametroaz komandoen irteerak adierazi fitxategira bolkatuko ditu  
  
# Behin /etc/anacrontab-en lerroa gehituta,defektuzko karpetara gehitu zuen scripta:  
sudo cp backup.sh /etc/cron.monthly
```

Gogoratu anacron-ek, makina itzali daitekeela suposatzen duela eta beraz, makina piztean exekutatu gabe pasa diren komandoak berreskuratuko ditu eta exekutatu. Fijatu baita ere, anacron-en atzerapena ere adierazi behar dela (kasu honetan 15 minutu), pendiente dauden programak piztu berritan exekutatu ezker makina gehiegi kargatzeko arriskua ekiditeko da.

Ondo izan,

Helburuak:

man komandoa ezagutu (eta erabili)

Bash komando interpretea erabiliz prozesuak kudeatu

Bash komando interpretea erabiliz prozesuei seinaleak bidali

Prosezuen sistema-deiak erabiliz prozesuak kudeatu

Prosezuen sistema-deiak erabiliz prozesuei seinaleak bidali

Bash komandoak prozesuak kudeatzeko:

1.- ps komandoak hainbat aukera ditu. Aztertu itzazu -a -u -x aukerak (zer egiten dute, zertarako erabili daitezke?)

-a: Prozesu guztiak aukeratu terminalarekin ez erlazionatutakoak eta sesio liderrak izan ezik.

-u: Aukeratu EUID-aren arabera

-x: Zure prozesu guztiak aukeratu (ps komandoa egin duen erabiltzaileak).

2.-Egin c lengoian programa bat begizta infinito batekin. Egikaritu duzunez, komando interpretea blokeatuta geratu da. Ireki ezazu beste terminal bat eta prozesua amaitu (kill komandoa erabili)

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

```
int main(){
    int bueltaKop = 0;

    while (bueltaKop>=0) {

        bueltaKop++;
        printf("%d",bueltaKop);

    }
}
```

Terminalean:

```
ps -u
```

```
kill 4436
```

3.-Lortu ezazu euiti erabiltzaileak egikaritzen ari den prozesuen zerrenda.

```
ps -x
```

4.-Gauza bera egin baina lortu duzun zerrenda bidali ezazu, aldi berean, irteera.txt fitxategira eta sarrera estandarrera (tee komandoaz baliatu zaitezke)

```
ps -x | tee irteera.txt
```

5.-Aurreko programa infinitoa egikaritu. Aurrekoa baino lehentasun txikiagoa esleitu iezaiozu prozesuari. Eta gero, SIGKILL seinalea bidaliozu.

```
Renice +19 4436
```

```
kill -SIGKILL 4436
```

6.-"backup.sh" skripta egin /home/euiti/proba katalogoaren backup-a egiten duena /tmp/homeeuiti.tar.gz izenarekin.

```
Rsync -arv ~/proba ~/homeeuiti
```

```
tar -cvf homeeuiti
```

```
gzip homeeuiti homeeuiti.tar
```

a)bost minutu barru egikaritzeko, at komandoa erabiliz

```
at 10:50am March 25
```

```
at> ./backup.sh
```

b)egunero, arratsaldeko 16:00etan.

```
0 16 * * * ~/backup.sh
```

c)hileko lehenengo bost egunetan, goizeko 9:00etan.

```
0 9 * 1,2,3,4,5 * ~/backup.sh
```

d)orduro, abuztuko ostiraletan.

```
0 * 5 * 8 ~/backup.sh
```

e)hilean behin exekuta dadin (man anacron).

Linux-eko APIak prozesuak kudeatzeko:

1. Egikaritu ezazu hurrengo programa:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main(int argc,char **argv){
    int rc=0,status;
    if (argc!=1) {
        printf("uso: %s\n",argv[0]);
        rc=1;
    }
    else if (fork()) {
        wait(&status);
        printf("GURASOA: pid=%8d ppid=%8d user-id=%8d \n",getpid(), getppid(), getuid());
        printf("Semeak itzuli duen egoera-kodea: %d\n", status);
    }
    else {
        printf("SEMEA: pid=%8d \n", getpid());
    }
    exit(rc);
}
```

eta erantzun hurrengo galdereei:

a) Zer bistaratzen da pantailan?

```
SEMEA: pid= 4557
```

```
GURASOA: pid= 4556 ppid= 4436 user-id= 1000
```

```
Semeak itzuli duen egoera-kodea: 0
```

b) Zer egikaritzen da lehenago, gurasoa edo semea? Zergatik? Umea, gurasoak wait duelako. Semea bukatu arte itxaron egingo du.

c) Hurrengo aginduan, wait(&status), zer da status aldagaian jasotzen dena?

status-ean gurasoak umearen egoera jasoko du eta egoera honen balioa umeak exit egitean itzulitako balioa izango da.

d) Zer itzuliko du exit aginduak errore bat jaso z gero? eta errorerik gabe amaitzen bada programa? Errorerik badago, 1 itzuliko du exit-ek, bestela 0.

2. Azaldu zer egiten duen hurrengo programak:

```
// trapper.c
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
#include <stdio.h>
void trapper(int);
int main(int argc, char *argv[])
{
    int i;
    for(i=1;i<=64;i++){
        signal(i, trapper);}
    printf("Identificador del proceso: %d\n", getpid() );
    pause();
    printf("Continuando...\n");
    return 0;
}
void trapper(int sig)
{
    signal(sig, trapper);
    printf("Señal que he recogido: %d\n", sig);
}
```

2.1 Konpilatu eta egikaritu. Beren PID-a jaso.

2.2 Beste terminal batetik SIGUSR1 seinalea bidali. Zein da erabili duzun komandoa?

[kill -SIGUSR1 4747](#)

2.3 Zer gertatzen da prozesu batek tratatu gabeko seinale heltzen zaionean?

2.4 Aldatu aurreko trapper.c 1tik 9arteko seinaleak tratatzeko. Egikaritu ostean SIGUSR1 seinalea bidatzeko. Zein da erabili duzun komandoa?

Kodean 64 zenbakia 9-gatik ordezkatzeta da ideia: for(i=1;i<=9;i++) egitea. Modu honetan lehen 9 seinaleak tratatzen dira (ikus kill -l) eta beste seinaleren bat jasoko balitz defektuzko funtzioa exekutatu luke. Beraz, prozesu batek, besterik ez bada adierazten, defektuz, kill -l -eko seinale guztientzat defektuzko funtzio bat du, askotan programa etetea izango dena.

2.5 Zer gertatu da SIGUSR1 bidali ostean? Zergatik?

Señal definida por el usuario 1

3. Moldatu programa SIGUSR1 seinalea jasoz gero “Kaixo” mezua ateratzeko.

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
#include <stdio.h>
void trapper(int);
int main(int argc, char *argv[])
{
    int i;
    for(i=1;i<=64;i++){
        signal(i, trapper);}
    printf("Identificador del proceso: %d\n", getpid() );
    pause();
    printf("Continuando...\n");
    return 0;
}
void trapper(int sig)
{
    printf("Kaixo\n");
}
```

```

    if(sig==10){
        printf("Kaixo");
    }
    signal(sig, trapper);
    printf("Señal que he recogido: %d\n", sig);
}

```

3.1. Moldatu mezu bera ateratzeko hurrengo seinaleekin SIGUSR2, SIGCONT, SIGSTOP eta SIGKILL. Posible da hori egitea? Idatzi seinale bakoitzaren portaera.

SIGUSR2: Erabiltzaileak deitutako seinalea

SIGCONT: Jarraitu geldituta bazegoen

SIGSTOP: Prozesua gelditu

SIGKILL: Prozesua hil

Ezin da programa moldatu SIGSTOP eta SIGKILL-ekin lan egiteko, baina SIGUSR2 eta SIGCONT-ekin bai:

```

#include <signal.h>
#include <stdio.h>
#include <unistd.h>
#include <stdio.h>
void trapper(int);
int main(int argc, char *argv[])
{
    int i;
    for(i=1;i<=64;i++){
        signal(i, trapper);}
    printf("Identificador del proceso: %d\n", getpid() );
    pause();
    printf("Continuando...\n");
    return 0;
}
void trapper(int sig)
{
    if(sig==10||sig==12||sig==19||sig==18||sig==9){
        printf("Kaixo");
    }
    signal(sig, trapper);
    printf("Señal que he recogido: %d\n", sig);
}

```

4. Trapper() funtzioaren barruan beharrezkoa da signal funtzioari berriz deitzea? [Ez](#)

5. Hurrengo killer.c programa konpilatu

```

// killer.c
#include <sys/types.h>
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
    pid_t pid;
    int sig;
    if(argc==3)
    {
        pid=(pid_t)atoi(argv[1]);
        sig=atoi(argv[2]);
        kill(pid, sig);
    } else {
        printf("Uso correcto:\n %s pid signal\n", argv[0]);
    }
}

```

```

        return -1;
    }
    return 0;
}

```

5.1. Zer egiten du atoi funtzioak?

Kate bateko punteroa int batean bihurtzen du

5.2. Linuxen, pid_t mota, ze mota da?

Prozesu baten identifikatzailea da (data type)

5.3. Trapper egikaritu. Bere PID jaso. Eta “./killer PID 9” egikaritu ostean. Ze gertatu da eta zergatik?

6. Hurrengo programa alarm.c egikaritu.

```

#include <signal.h>
#include <unistd.h>
#include <stdio.h>
void trapper(int);
int main(int argc, char *argv[])
{
    int i;
    signal(14, trapper);
    printf("Identificador proceso: %d\n", getpid() );
    alarm(5);
    pause();
    alarm(3);
    pause();
    for(;;)
    {
        alarm(1);
        pause();
    }
    return 0;
}

void trapper(int sig)
{
    signal(sig, trapper);
    printf("RIIIIIIIING!\n");
}

```

```
gcc -o alarm alarm.c
```

```
./alarm
```

```
ander@anderSanju:~/Escritorio$ ./alarm
```

```
Identificador proceso: 5817
```

```
RIIIIIIIING!
```

```
RIIIIIIIING!
```

```
RIIIIIIIING!
```

```
...
```

6.1. Zer egiten dute hurrengo aginduak?

```
signal(14, trapper);
```

```
for(;;) {
```

```
    alarm(1);
```

```
    pause(); }
```

Segunduro SIGALARM seinalea bidali, bukle infinitu batean.

6.2. Zer egiten du programak?

7. Hurrengo signalfork.c programa egikaritu:

```

#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

```

```

void trapper(int sig)
{
    signal(sig, trapper(sig));
    printf("SIGUSR1\n");
}

int main(int argc, char *argv[])
{
    pid_t padre, hijo;
    padre = getpid();
    signal( SIGUSR1, trapper );
    if ( (hijo=fork()) == 0 )
    { /* hijo */
        sleep(1);
        kill(padre, SIGUSR1);
        sleep(1);
        kill(padre, SIGUSR1);
        sleep(1);
        kill(padre, SIGUSR1);
        sleep(1);
        kill(padre, SIGKILL);
        exit(0);
    }
    else
    { /* padre */
        for (;;)
        {
            return 0;
        }
    }
}

```

- 7.1. Programa honetan semeak aitari bidalitako seinaleak ditugu. Baina zer gertatuko zan aitak semeari **SIGUSR1** seinalea bidaliz gero? Signal funtzioaren portaera fork egiterakoan semearen eragin bera du (portaera heredatzen du?). Zer gertatuko da? Aldatu programa aitak semeari SIGUSR1 seinalea bidaltzeko, eta portaera heredatzen den jakiteko.

```

#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

void trapper(int sig)
{
    signal(sig, trapper(sig));
    printf("SIGUSR1\n");
}

int main(int argc, char *argv[])
{
    pid_t padre, hijo;
    padre = getpid();
    signal( SIGUSR1, trapper );
    if ( (hijo=fork()) == 0 )
    { /* hijo */
        sleep(10);
    }
    else
    { /* padre */
        sleep(2);
    }
}

```

```

        kill(hijo, SIGUSR1);
    }

    return 0;
}

```

7.2. Aldatu semearen hurrengo lerro biak:

```

/*hijo */
sleep(10);
Aldatu aitaren lehenengo lerroak:
/* padre */
sleep(2);
kill(hijo, SIGUSR2);
for (;;)

```

konpilatu eta egikaritu. Blokeatzen da? Zergatik?

Blokeatutzen da, SIGUSR2-en defektuzko funtzio exekutatzen delako. Gurasoak semea seinalia bidaliko dio semea lotan dagoenean, horrela umea aktibatzen. Hori dela eta, umearen 'kill' funtzioak ez dira exekutatuko eta bukle infinitu bat sortuko da.

8. Hurrengo programak "lana" katalogoa sortzen du. Egin gauza bera gainerako exec funtzioak erabiliz.

```

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
int main(void)
{char *args[]={"/bin/mkdir","lana",NULL};
if (execve("/bin/mkdir",args,NULL)==-1)
{perror("execve");
exit(EXIT_FAILURE);}
puts("No debería llegar aqui");
exit(EXIT_SUCCESS);}

```

```

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
int main(void)
{char *args[]={"/bin/mkdir","lana",NULL};
if (execve("/bin/mkdir",args[0],args[1],args[2],NULL)==-1)
{perror("execve");
exit(EXIT_FAILURE);}
puts("No debería llegar aqui");
exit(EXIT_SUCCESS);}

```

9. Beheko programak emanda, azaldu "./padre 60 hijo" deiak egiten duena:

reloj.c

```

#include <stdlib.h> /*atoi and exit*/

#include <stdio.h> /*printf*/

#include <unistd.h> /*alarm*/

#include <signal.h> /*signal and SIGALRM*/

```

```

void xtimer(){
printf("time expired.\n");
}
int main(int argc, char *argv[]){
unsigned int sec;
sec=atoi(argv[1]);
printf("time is %u\n",sec);
signal(SIGALRM,xtimer);
alarm(sec);
pause();
return 0;
}

```

hijo.c

```

#include <unistd.h> /*sleep*/
int main(){
sleep(6);
return 0;
}

```

padre.c

```

#include <sys/types.h> /*kill y wait*/
#include <sys/wait.h> /*wait*/
#include <signal.h> /*kill*/
#include <stdlib.h> /*exit*/
#include <stdio.h> /*printf*/
#include <unistd.h> /*fork and exec...*/
#include <time.h> /*time*/
int main(int argc, char *argv[]){
int idHijo, idReloj, id, t1, status1, status2;
id = getpid();
printf("Proceso padre: %d\n", id);
if((idReloj = fork()) == 0) { /* hijo Reloj */
execl("reloj", "reloj", argv[1], NULL);
}
if((idHijo = fork()) == 0) {
execv(argv[2], &argv[2]);
}
printf("Proceso hijo Perezoso: %d\n", idHijo);
}

```



```

printf("Proceso hijo Reloj: %d\n", idReloj);
t1 = time(0);
if((id = wait(&status1)) == idHijo) {
kill(idReloj, SIGKILL);
//Si el hijo ha cambiado de estado, antes de la ejecución del wait
//entonces la llamada a wait retornará inmediatamente con su estado
wait(&status2);
} else {
kill(idHijo, SIGKILL);
wait(&status2);
status1 = 1;}
t1=time(0)-t1;
printf("Tiempo del proceso hijo : %d\n", t1);
exit(status1);
}

```

Prozesua abiaraziko da gurasoaren bidez. Programaren exekuzio denbora minimoa (lehen argumentua) pasatzen baldin bada programa bukatu baino lehen, gurasoa semea amaituko du eta ondoren bere id eta prorgamaren pantailan idatziko ditu. Amaitzerakoan gurasoa bere iraupena idatziko du eta itzulera kodea.

PRAKTIKA: Sistema deiak**Helburuak:**

- Fitxategiak kudeatzen dituzten sistema deiak ezagutzea.
- Liburutegien funtzio tipikoak erabili beharrean, linuxeko APIak erabiltzen dituzten programak sortu.

Erreferentziak:

<http://blog.txipinet.com/index.php/Informatica/>

<http://www.e-ghost.deusto.es/docs/2006/ProgramacionGNULinux.odt>

Fitxategien Sistema-Deiak

Sistema Deiak ulertzeko modurik errezena oinarritzko funtzioak erabiltzea da, hau da, fitxategiak maneiatzen dituztenak daude oinarritzkoen artean. Esanda dago, UNIX-en eta GNU/LINUX-en konkretuki dena fitxategi bat dela eta honez gero syscall-ak UNIX-en sistema programazioaren oinarria dira.

Has gaitezen fitxategi bat sortzen. Fitxategi bat irekitzeko bi aukera ezberdin ditugu, open() eta creat(). Aspaldi, open() erabiliz, bakarrik jadanik sortuta zeuden fitxategiak ireki ahal genituen. Beharrezkoa genuen creat() dei bat egitea open() deitu aurretik. Gaur egun, open() erabiliz fitxategi bat sortu dezakegu, bere erazagupenari, parametro berri bat erantsi zaiolako:

```
int creat( const char *pathname, mode_t mode )
int open( const char *pathname, int flags )
int open( const char *pathname, int flags, mode_t mode )
```

Ikusten denez, open() berria aspaldiko open()-aren eta creat()-aren funtzionaltasunen batura bat da. Beste berezitasunen artean azpimarratu dezakegu “mode” parametroa orain “mode_t” dela. Mota hau UNIX-en askotan erabiltzen da eta normalean “int” edota “unsigned int” bati dagokio. Hala ere, aurreko bertsioetako bateragarritasunagatik mota hori mantentzen da. Horregatik, syscall hauek erabiltzeko, hurrengo goiburu-fitxategiak gehitu behar ditugu:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

open() funtzioaren parametroak:

- pathname: path + fitxategiaren izena, kate baten helbide bezala edo kakotzen artean.
- “flags” fitxategia zertarako eta nola zabalduko dan adierazteko. Hurrengo taulan hartu dezaketen balioak daude:

Adierazlea	Edukia	Deskribapena
O_RDONLY	0000	Fitxategia soilik irakurtzeko irekitzen da.
O_WRONLY	0001	Fitxategia soilik idazteko irekitzen da.
O_RDWR	0002	Fitxategia idazteko eta irakurtzeko irekitzen da.
O_SEQUENTIAL	0020	Fitxategia sekuentzialki atzituta izateko (zintak kudeatzeko erabiliak).
O_TEMPORARY	0040	Fitxategiak denbora izaera du.
O_CREAT	0100	Fitxategia sortu aurkitu ezean.
O_EXCL	0200	O_CREAT aukerarekin batera zehazten bada, existitu ezker hutsitzen du.
O_NOCTTY	0400	Fitxategia terminal-gailu (TTY) bat baldin bada, ez da prozesu kontrolen terminalean bihurtuko (CTTY).
O_TRUNC	1000	Fitxategia hutsitzen du.
O_APPEND	2000	Idazkera erakuslea fitxategiaren bukaeran kokatzen du, eta datu berriak bukaeran idatziko dira.
O_NONBLOCK	4000	Fitxategiaren irekiera ez-blokeatzailea izango da. O_NDELAY -aren baliokidea da.
O_SYNC	10000	Fitxategian egiten diren idazketak sinkronoak dira.
O_ASYNC	20000	Fitxategian egiten diren idazketak asinkronoak dira.
O_DIRECT	40000	Disko-sarbidea zuzenki egingo da, buffer barik.
O_LARGEFILE	100000	Soilik oso handiak diren fitxategientzako.
O_DIRECTORY	200000	Katalogo bat denean.
O_NOFOLLOW	400000	Lotura sinbolikoak ez jarraitzeko aukera.

1.5.1 Taula “Flag”-en balioak

Lerroa nahiko luzea da eta baloreak bat baino gehiago kateatzeko aukera dago. Hau da, balore ezberdinen artean OR logiko bat egitea, nahi dugun efektua lortuz. Horrela, creat() -aren parekoa izango den hurrengo open()-a lor dezakegu:

```
open( pathname, O_CREAT | O_TRUNC | O_WRONLY, mode )
```

“mode” argumentua fitxategien sistemaren barnean baimenak zehazteaz arduratzen da (chmod komandoarekin egiten genuen bezala). Balore posibleen taula honako hau da:

Adierazlea	Edukia	Deskripzioa
S_IROTH	0000	Gainerako Erabiltzaileek irakurtzeko aukera
S_IWOTH	0001	Gainerako Erabiltzaileek idazteko aukera
S_IXOTH	0002	Gainerako Erabiltzaileek exekutatzeko aukera
S_IRGRP	0010	Taldeko Erabiltzaileek irakurtzeko aukera
S_IWGRP	0020	Taldeko Erabiltzaileek idazteko aukera
S_IXGRP	0040	Taldeko Erabiltzaileek exekutatzeko aukera
S_IRUSR	0100	Jabeak irakurtzeko aukera
S_IWUSR	0200	Jabeak idazteko aukera
S_IXUSR	0400	Jabeak exekutatzeko aukera
S_ISVTX	1000	Fitxategian “sticky bit”-a gehitu

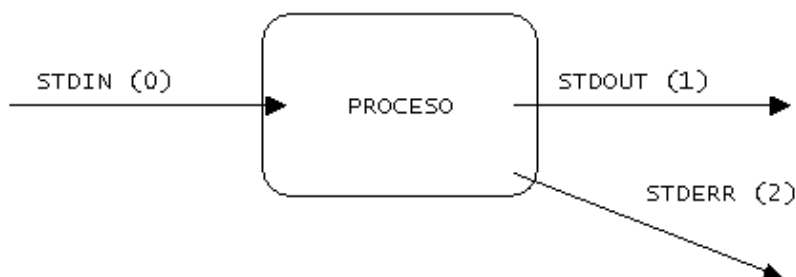
S_ISGID	2000	Fitxategian SUID bita gehitu
S_ISUID	4000	Fitxategian SGID bita gehitu
S_IRWXU	S_IRUSR + S_IWUSR + S_IXUSR	Jabeak irakurtzeko, idazteko eta exekutatzeko aukera
S_IRWXG	S_IRGRP + S_IWGRP + S_IXGRP	Taldeko Erabiltzaileek irakurtzeko, idazteko eta exekutatzeko aukera
S_IRWXO	S_IROTH + S_IWOTH + S_IXOTH	Gainerako Erabiltzaileek irakurtzeko, idazteko eta exekutatzeko aukera

Taula1 “Mode” argumentuaren balore posibleen lista

Balore guzti hauek goiburu-fitxategi batean zehazten dira eta horregatik barneratzea komeni da:

```
#include <sys/stat.h>
```

open() fitxategi bat ondo ireki ezkerro, irekitako fitxategia maneiatzeko fitxategiaren deskribatzailearen zenbakia itzultzen du. Prozesu bakoitzak, fitxategiak errez maneiatzeko, fitxategien deskribatzaileen taula bat erabiltzen du. Hasieran, taula horren 0,1 eta 2 sarrerak STDIN, STDOUT eta STDERR fitxategiek betetzen dituzte, hau da, sarrera estandarra, irteera estandarra eta errore estandarra.



1. Irudia Prozesu baten hasierako fitxategien deskribatzaileak.

Fitxategien deskribatzaileen taula hau, hotel baten moduan ulertu dezakegu, non hasieran lehenengo hiru logelak STDIN, STDOUT eta STDERR bezeroek okupatzen dituzten. Bezero gehiago etorri ezkerro (fitxategi gehiago ireki ezkerro), bezero hauek hurrengo logeletan sartuko dira. Horrela, prozesuaren hasieran irekitako fitxategi batek, normalean, 2-ra hurbiltzen den fitxategiaren deskribatzaile bat izango du. “Hotel” honetan, logela baxuena beti bezero berrienarentzat izango da. Hau guztia kontuan izan beharko dugu etorkizunean.

Ongi, orain existitzen ez direnean fitxategiak irekitzen eta sortzen badakigu, baina ezin ditugu fitxategiak irekita utzi, hau da, ondo itxi gabe. Badakizue C-k bere programatzaileak pertsona arduratsu moduan hartzen dituela. Fitxategi bat ixteko nahikoa da syscall close()-ari fitxategiaren deskribatzailea argumentu moduan pasatzea:

```
int close( int fd)
```

Ariketa 1:

- Hurrengo adibidearekin ariketa1.c osotu.
[subl ariketa1.c](#)
- Konpilatu warning guztiak ezabatu arte.

Nahiko erreza da. Ikus dezagun adibide batekin nola funtzionatzen duen hau guztia:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char *argv[] )
{
    int fd;
    if( (fd = open( argv[1], O_RDWR )) == -1 )
    {
        perror( "open" );
        exit( -1 );
    }
    printf( "Irekita dagoen fitxategia %d deskribatzailea du\n", fd );
    close( fd );
    return 0;
}
```

Hasieran beharrezko goiburu-fitxategiak ditugu, honaino azaldu dugun bezala. Segidan, fitxategiaren deskribatzailea barnean edukiko duen “fd” aldagaia deklaratu dugu, eta open()-i dei egiten diogu, dei honen emaitza “fd”-n gordetzen dugularik. “fd” -1 izan ezker, fitxategia irekitzean errore bat eman duela esango dugu eta programatik irtengo gara. Ordea, beste edozein kasutan, programaren gauzatzearekin jarraituko dugu, fitxategiaren deskribatzailea pantailatik erakutsiz eta ondoren fitxategia itxiz. Programa honen funtzionamendua hemen ikus dezakegu:

```
txipi@neon:~$ gcc ariketa1.c -o ariketa1
txipi@neon:~$ ./ariketa1 ariketa1.c
Irekita dagoen fitxategia 3 deskribatzailea du.
```

Hurrengo pauso logikoa, maneiatzen ditugun fitxategietan irakurri eta idatzi, ahal izatea da. Horretarako, antzeko bi “syscall” erabiliko ditugu: read() eta write(). Erazagupenak:

```
ssize_t read(int fd, void *buf, size_t count);
ssize_t write( int fd, void *buf, size_t count )
```

Lehenengoak “fd”-n zehaztutako fitxategiaren deskribatzailetik “count” bytak irakurtzen saiatzen da, “buf” buffer-an gordetzeko. Saiatzen dela esaten dugu, read()-ek irakurritako byte kopurua itzultzen duelako eta balore hau “count” aldagaiarekin konparatuz, eskatzen genuen byte kopurua irakurri duen edo ez jakin dezakegu. Irakurritako byte-ak zenbatzeko erabilitako datu motak pixka bat arraroak izan daitezke baina GNU/Linux bertsio honetan soilik zenbaki osoak dira, goiburu fitxategian ikus daiteken bezala.

2. Ariketa:

ssize_t eta size_t, ze motatakoa diren jakiteko, aurreprosezadorea erabili. Hau da, .c .i batean bihurtu eta aurkitu ssize_t eta size_t ze motatakoak diren.

```
Gcc -o ariketa1.i -E ariketa1.c
gedit ariketa1.i
ssize_t long int bat da
```

size_t long unsigned int bat da

write() funtzioaren erabilera oso antzekoa da. “buf” bufferra, idatzi nahi dugunarekin betetzea nahikoa izango da. “count”-en bere tamaina zehaztuko dugu eta “fd”-n bere fitxategiaren deskribatzailearekin idatziko dugun fitxategia zehaztu:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

```
#define STDOUT 1
#define SIZE 512
```

3. Ariketa

Hurrengo programak parametroak behar ditu. Parametro barik deituz gero, erabiltzeko zenbat eta zein parametroren beharra dagoan laguntzen duen mezua gehitu, ariketa3.c deitu programari.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int main( int argc, char *argv[] )
{
    int fd, readbytes;
    char buffer[SIZE];
    if (argc != 2)
        {printf (" erabili hurrengo deia: %s fitxategiaren izena", argv[0]);
        exit (-1);
        }
    if( (fd = open( argv[1], O_RDWR )) == -1 )
    {
        perror( "open" );
        exit( -1 );
    }
    while( (readbytes = read( fd, buffer, SIZE )) != 0 )
    {
        /*          write( STDOUT, buffer, SIZE ); */
        write( STDOUT, buffer, readbytes );
    }

    close( fd );

    return 0;
}
```

Ikus daitekenez, hasieran bi konstante zehaztuko ditugu, STDOUT, irteera estandarra zehazten duen fitxategiaren deskribatzailea, 1 dena eta SIZE, erabiliko dugun buffer-aren tamaina adierazten duena. Segidan, beharrezkoak diren aldagaiak erazagutzen ditugu eta parametrotik pasatutako fitxategia irekitzen saiatzen gara (argv[1]), irakurtzeko eta idazteko sarbidearekin. Errore bat gertatzen bada open() funtzioak -1 itzultzen du, eta errore mezua aterako du eta bestela aurrera egingo dugu. Honen ondoren, amaitu (read()-ek irakurritako 0 byte bueltatzen ditutenean) arte SIZE-naka bytak irekitako fitxategitik (“fd”) irakurriko dituen kiribil bat daukagu. Kiribilaren buelta bakoitzean iraXkurritakoa STDOUT irteera estandarretik idatziko da. Azkenean fitxategiaren deskribatzailea close() batekin itxiko dugu.

Azken finean, programa honek egiten duen gauza bakarra fitxategi baten edukia irteera estandarretik erakustea da, hau da, “cat” komandoak gehienetan egiten duena.

Aurreko programan komentatutako lerro bat dago:

```
/*          write( STDOUT, buffer, SIZE ); */
```

Write() dei honetan, ez daukagu read() deiak itzuli duena kontuan izaten, SIZE byteak idazten saiatzen baizik, hau da, 512 byte bakoitzeko. Zer gertatuko zan, lerro hau erabiliko bazan, bestearen truke? Parametro moduan pasatzen dugun fitxategia nahiko handia bada, while kiribilaren lehenengo zikloak ondo ibiliko dira, read()-ek 512 karaktere bueltatuko duelako irakurritako byte kopuru moduan, eta write()-k behar bezala idatziko dituelako. Baina kiribilaren azken iterazioan, read()-ek 512 baino byte gutxiago irakurriko dugu, parametrotik pasatutako fitxategiaren tamaina 512 byten multiploa izatea oso arraroa izango zelako. Orduan, read()-ek 512 byte baino gutxiago irakurriko ditu eta write()-k 512 idatziko ditu. Horren ondorioz, write()-k karaktere zaborrak idatziko ditu.

4. Ariketa: Kopiatu hurrengo ariketa “ariketa4.c” fitxategian eta konpilatu.

```
txipi@neon:~ $ gcc files.c -o files
txipi@neon:~ $ ./files files.c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#define STDOUT 1
#define SIZE 512

int main(int argc, char *argv[])
{
    int fd, readbytes;
    char buffer[SIZE];

    if( (fd=open(argv[1], O_RDWR)) == -1 )
    {
        perror("open");
        exit(-1);
    }

    while( (readbytes=read(fd, buffer, SIZE)) != 0 )
    {
/*          write(STDOUT, buffer, readbytes); */
        write(STDOUT, buffer, SIZE);
    }

    close(fd);

    return 0;
}
@p@N"@@Àýÿ¿4ýÿ¿ô¢%@"@ì8ýÿ¿D&@
"&@"@Xýÿ¿Ù'@Xýÿ¿@txipi@neon:~ $
```

Adibide honek erakusten duen moduan, hasieran programa ondo doa, baina read()-ak irakurritako bytak kontuan hartzen ez baditugu, azkenean karaktere zaborrak idazten amaituko dugu.

Lagundu ahal gaituen beste funtzio bat lseek()-a da. Askotan ez gara idazteko edota irakurtzeko fitxategi baten hasieran kokatu nahi, baizik eta fitxategiaren hasierari edota amaierari erlatiboa den

desplazamendu konkretu batean kokatzea da interesatzen zaiguna. “lseek()” funtzioa posibilitate hori ematen digu eta honako prototipoa dauka:

```
off_t lseek(int fildes, off_t offset, int whence);
```

5. Ariketa:

lseek funtzioak off_t itzultzen du, off_t ze motatakoa den jakiteko aurreprozesadorea erabili.

```
Subl files.c
gcc -o files.i -E files.c
gedit files.i
typedef long int __off_t;
```

Jasotzen dituen parametroak oso ezagunak dira. “fildes” fitxategiaren deskribatzailea da, “offset” kokatu nahi garen desplazamendua da, “whence”-k kokapena adierazten du eta hurrengo balioak hartu ditzake:

Erakuzlea	Edukina	Deskribapena
SEEK_SET	0	Erakuslea fitxategiaren hasieratik “offset” bytetara kokatzen du.
SEEK_CUR	1	Erakuslea momentuan erakusleak daukan posiziotik “offset” bytetara kokatzen du.
SEEK_END	2	Erakuslea fitxategiaren amaieratik “offset” bytetara kokatzen du.

Taula 2 “Whence” argumentuak jaso ditzazkeen baloreen zerrenda

Adibidez, fitxategi bat irakurri nahi badugu eta 200 karaktereko goiburua bat pasatu nahi badugu, horrela egin genezake:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#define STDOUT 1
#define SIZE 512

int main( int argc, char *argv[] )
{
    int fd, readbytes;
    char buffer[SIZE];

    if( (fd = open( argv[1], O_RDWR )) == -1 )
    {
        perror( "open" );
        exit( -1 );
    }

    lseek( fd,200, SEEK_SET );

    while( (readbytes = read( fd, buffer, SIZE )) != 0 )
    {
        write( STDOUT, buffer, SIZE );
    }
}
```



```

close(fd);

return 0;
}

```

6. Ariketa:

Aurreko kodea ariketa6.c bezala gorde eta write kodea aldatu zaborrik ez ateratzeko.

```

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#define STDOUT 1
#define SIZE 512

int main( int argc, char *argv[] )
{
    int fd, readbytes;
    char buffer[SIZE];

    if( (fd = open( argv[1], O_RDWR )) == -1 )
    {
        perror( "open" );
        exit( -1 );
    }

    lseek( fd, 200, SEEK_SET );

    while( (readbytes = read( fd, buffer, SIZE )) != 0 )
    {
        write( STDOUT, buffer, readbytes);
    }

    close(fd);

    return 0;
}

```

Sortze, irekitzen, ixten, irakurtzen eta idazten badakigu eta honekin, denetarik egin dezakegu! Fitxategiak maneiatzeko funtzioekin amaitzeko, chmod(), chown() eta stat() ikusiko ditugu, fitxategiaren modua eta jabea aldatzeko edota beren ezaugarriak ikusi ahal izateko.

chmod() funtzioa izen bereko komanduaren erabilera dauka: fitxategi konkretu bateko sarbide moduak aldatu. Nahiz eta C erabili, gure programa fitxategi-sistemaren murrizketei helduta dago eta soilik beren jabea edo root aldatu ditzakete fitxategi konkretu baten sarbide motak. Fitxategi bat sortzean, bai creat() erabiliz nahiz open() erabiliz, fitxategi honek, konfiguratuta dagoen modumaskararen arabera (ikus dezagun “man umask”) dagoen modu bat dauka. Hala ere, honako funtzioak erabiliz beren moduak zuzenean alda ditzakegu:

```
int chmod(const char *path, mode_t mode);
```

```
int fchmod(int fildes, mode_t mode);
```

Funtzio bakoitzaren prototipoa ikusiz, bere funtzionamendua aztertu dezakegu: lehendabizikoa, `chmod()`, “path” katean adierazitako fitxategiaren modua aldatzen du. Bigarrena, `fchmod()`, fitxategiaren ibilbidea daukan karaktere-katea jaso beharrean, fitxategiaren deskribatzaile bat jasotzen du, “fildes”. “mode” parametroa “mode_t” motatakoa da, baina GNU/Linux-en osoa den aldagai bat erabiltzearen baliokidea da. Bere balioa “chmod” komandoa deitzean erabiliko dugunaren bera da, adibidez:

```
chmod( "/home/txipi/prueba", 0666 );
```

Fitxategiaren jabea aldatzeko hurrengo funtzioak erabiliko ditugu:

```
int chown(const char *path, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
int lchown(const char *path, uid_t owner, gid_t group);
```

Haiekin, bere ibilbidearen arabera (`chown()` y `lchown()`) eta fitxategiaren deskribatzailearen arabera (`fchown()`), fitxategi baten jabea eta taldea alda ditzakegu. Jabea (“owner”) eta taldea (“group”) erabiltzaileak eta taldeak identifikatzen dituzten osoak dira, hala nola “/etc/passwd” eta “/etc/group” fitxategiak azaltzen duten. Parametro haietako baten bat -1 balioarekin finkatzen badugu (“owner” edo “group”), zegoen bezala utzi nahi dugula ulertuko du. `lchown()` funtzioa `chown()` bezalakoa da, fitxategiekiko lotura sinbolikoetan ezik. Linux 2.1.81 bertsioaren aurrekoetan (2.1.46 bertsioa izan ezik), `chown()` ez zituen lotura sinbolikoak jarraitzen. Linux 2.1.81-etik aurrera hasi zen `chown()` lotura sinbolikoak jarraitzen eta `syscall` berri bat sortu zen, `lchown()`, lotura sinbolikoak jarraitzen ez zituena. Honez gero, gure programen segurtasuna hobetu nahi badugu `chown()` erabiliko dugu, lotura sinboliko nahasgarriekin gaizki-ulertuak saihesteko.

7. Ariketa:

Azaldu adibide batekin zer den SUID eta SGID bita.

Laguntza: <http://es.wikipedia.org/wiki/Setuid>

SUID-ek erabiltzaileari baimenak emango dizkio eta SGID-ek berriz erabiltzaile horren taldeei.

Edozein erabiltzaile arrunt batek fitxategiaren jabea aldatu ezker, fitxategi horrek bit SUID edo SGID izan ezker desaktibatu egingo dira segurtasunagatik. POSIX estandarrak, root-ek akzio bera egin ezker gertatuko dena ez du garbi ezartzen, orduan linux-aren arabera bit SUID eta bit SGID aktibatuta edo desaktibatuta geratzea gerta daiteke. Erabilpenaren adibide bat honako hau izango litzateke:

```
gid_t grupo = 100; /* 100 es el GID del grupo users */
chown( "/home/txipi/prueba", -1, 4);
ikusteko taldeak:
```

```
/etc/group
```

Dei honekin “/home/txipi/prueba” fitxategiaren jabea eta taldea aldatu nahi ditugula adierazten dugu, jabea zegoen moduan (-1) utziz eta taldea 100 balorearekin aldatuz, “users” taldeari dagokiona:

```
txipi@neon:~$ grep 100 /etc/group
users:x:100:
```

Geratzen zaigun gauza bakarra, fitxategi baten ezaugarrietara sartzen jakitea da, `stat()` funtzioa

erabiliz. Funtzio honen portaera, orain arte ikusitako funtzioek daukatenaren ezberdina da. Fitxategi baten i-nodo informazio guztia (hau da, ezaugarri guztiak) estruktura batean sartzen du. Eta stat() funtzioari estruktura baten erreferentzia pasatuko diogu. Stat syscall funtzioari deitu ostean, estruktura horretan fitxategiaren ezaugarri guztiak izango ditugu, behar bezala beteak. Honekin erlazionatutako funtzioak honakoak dira:

```
int stat(const char *file_name, struct stat *buf);
int fstat(int fildes, struct stat *buf);
int lstat(const char *file_name, struct stat *buf);
```

Hau da, chown(), fchown() eta lchown()-en antzekoak, baina fitxategiaren jabeak zehaztu ordez, bigarren parametro bezala “stat” moduko estruktura batekiko erakusle bat behar dute:

```
struct stat {
    dev_t      st_dev;    /* dispositivo */
    ino_t      st_ino;    /* numero de inode */
    mode_t     st_mode;   /* modo del fichero */
    nlink_t    st_nlink;  /* numero de hard links */
    uid_t      st_uid;    /* UID del propietario */
    gid_t      st_gid;    /* GID del propietario */
    dev_t      st_rdev;   /* tipo del dispositivo */
    off_t      st_size;   /* tamaño total en bytes */
    blksize_t  st_blksize; /* tamaño de bloque preferido */
    blkcnt_t   st_blocks; /* numero de bloques asignados */
    time_t     st_atime;  /* ultima hora de acceso */
    time_t     st_mtime;  /* ultima hora de modificación */
    time_t     st_ctime;  /* ultima hora de cambio en inodo */
};
```

8. Ariketa:

Linux-en dena da fitxategi bat, eta edozein fitxategi i-nodo batekin identifikatzen da. “ls -l” komandoarekin jakin dezakegu fitxategiaren i-nodoa zein den. Sortu bi fitxategi “p1” eta “p2” “/tmp” katalogoan eta dagozkion inodoak pantailaratzen duen programa egin.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>

int main( int argc, char *argv[] )
{
    struct stat estructura;
    if( ( lstat( argv[1], &estructura ) ) < 0 )
    {
        perror( "lstat" );
        exit( -1 );
    }

    printf( "i-nodo: %lu\n", estructura.st_ino );
    return 0;
}
```

Ikusten dugunez, fitxategiaren oso informazio garrantzitsua ikus dezakegu. Hurrengo adibideak hau

guztia nola funtzionatzen duen erakusten du:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>

int main( int argc, char *argv[] )
{
    struct stat estructura;
    if( ( lstat( argv[1], &estructura ) ) < 0 )
    {
        perror( "lstat" );
        exit( -1 );
    }
    printf( "Fitxategiaren propietateak" <%s>\n", argv[1] );
    printf( "i-nodo: %lu\n", estructura.st_ino );
    printf( "modua: %#o\n", estructura.st_mode );
    printf( "loturak: %ld\n", estructura.st_nlink );
    printf( "jabea: %d\n", estructura.st_uid );
    printf( "taldea: %d\n", estructura.st_gid );
    printf( "dispositibo mota: %ld\n", estructura.st_rdev );
    printf( "tamaina totala, byte-tan: %ld\n", estructura.st_size );
    printf( "blokearen gustuko tamaina: %ld\n", estructura.st_blksize );
    printf( "esleitutako bloke kopurua: %ld\n", estructura.st_blocks );
    return 0;
}
```

Aurreko kodean esanguratsuak diren zenbait xehetasun daude: “%#o” erabiltzen dugu, fitxategiaren sarbide modua era zortzitar eran erakusteko. Hala ere, ezagutzen ditugun lau zenbaki baino gehiago agertzen dira. Hau, eremu horretan fitxategiaren motari buruz ere informazioa agertzen zaigulako da (fitxategi bat, gailu bat, sekuentziala, FIFO bat, etab. dena)

Aurreko kodearen egikaritzearen emaitza bat honakoa izan daiteke:

```
txipi@neon:~$ gcc stat.c -o stat
txipi@neon:~$ ./stat stat.c
Fitxategiaren propietateak" <./prueba.txt>
i-nodo: 3149211
modua: 0100600
loturak: 1
jabea: 1000
taldea: 1000
dispositibo mota: 0
tamaina totala, byte-tan: 9
blokearen gustuko tamaina: 4096
esleitutako bloke kopurua: 8
```