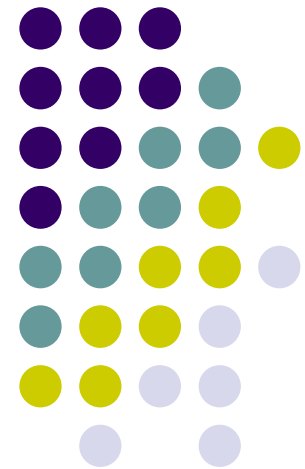
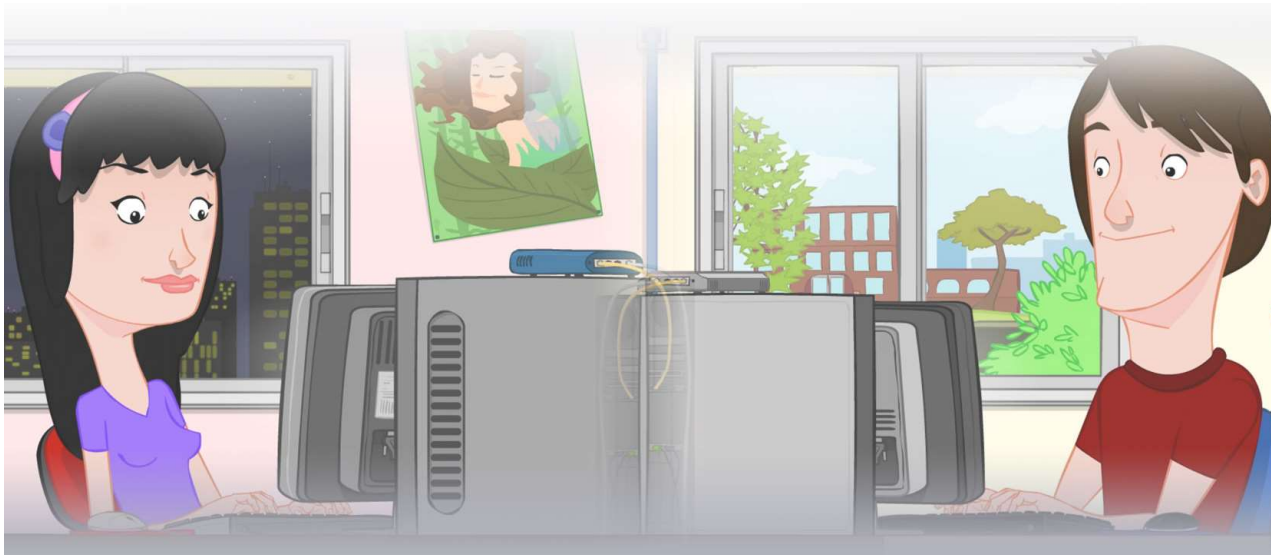


1. Blokea: Sistema Eragileak

Informazio-Sistemen Arkitektura

Telekomunikazio Teknologiaren Ingeniaritzako Gradua (3. Maila)

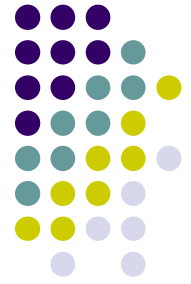


TELEK:O
UPV/EHU Bilbao



1. Blokea - Edukiak

1. Ordenagailuen Arkitekturako kontzeptuak
2. Sistema Eragileak. Sarrera.
3. Prozesuak
 - Prozesuak eta hariak
 - Prozesuen arteko komunikazioa
 - Komunikazio eta Sinkronizazio Mekanismoak
 - Planifikazioa
 - Itxaronaldi pasiboaren inplementazioa
4. Sarrera/Irteera
5. Memoriaren Kudeaketa
 - Memoria birtuala edo alegiazko memoria
 - Orrikapena eta Segmentazioa
 - Ordezkapeneko algoritmoak
6. Fitxategi Sistema
 - Fitxategiak eta direktorioak
 - Fitxategi Sistemaren antolaketa



3. Gaia – Edukiak PROZESUAK

1. Prozesuak
2. Prozesu arinak edo Hariak
3. Prozesuen arteko Komunikazioa
4. Komunikazio eta Sinkronizazio
Mekanismoak
5. Planifikazioa
6. Itxaronaldi pasiboaren inplementazioa

3. PROZESUAK

3.1 PROZESUAK

- **Prozesua** zer den:
 - Programa (errezeta) bat exekuzioan (ekintza)
 - SE-ak kudeatutako prozesamendu – unitatea
- SE – aren **prozesu – taula**, **PKB** egituretan oinarritutako datu – egitura da:
 - Memoriaren irudiaren (*core image*) segmentuak: C+D+S
 - Prozesadorearen egoera (programazio – ereduko erregistroak)
 - Irekitako fitxategien deskribatzaileak
 - Prozesuaren pid, erabiltzailearen uid...
 - Tenporizadoreak...

Iturria: <https://www.geeksforgeeks.org/process-table-and-process-control-block-pcb/>



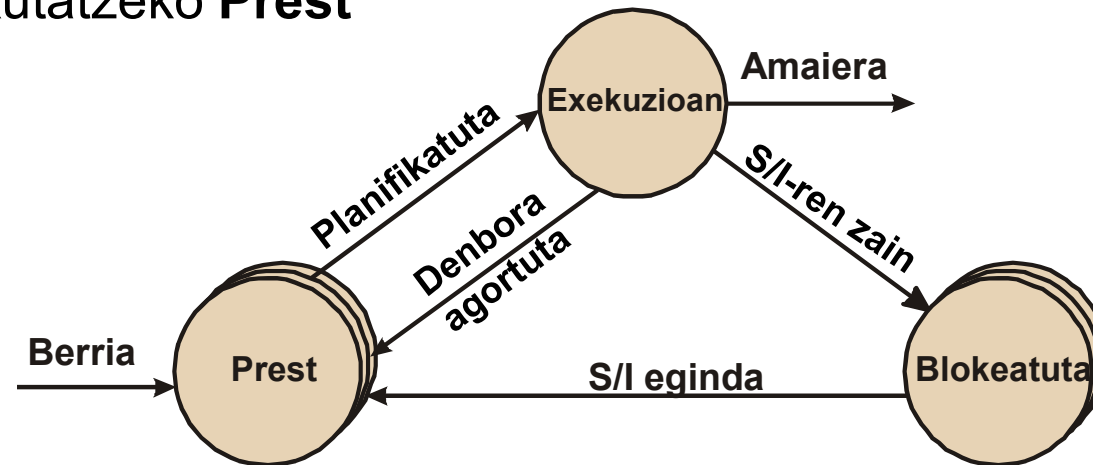
Pointer
Process State
Process Number
Program Counter
Registers
Memory Limits
Open File Lists
Misc. Accounting and Status Data

3. PROZESUAK

3.1 PROZESUAK



- Prozesu baten oinarritzko egoerak:
 - **Exekuzioan** (bana prozesadoreko)
 - **Blokeatuta** (S/I eragiketa baten edo gertaera baten zain)
 - Exekutatzeko **Prest**



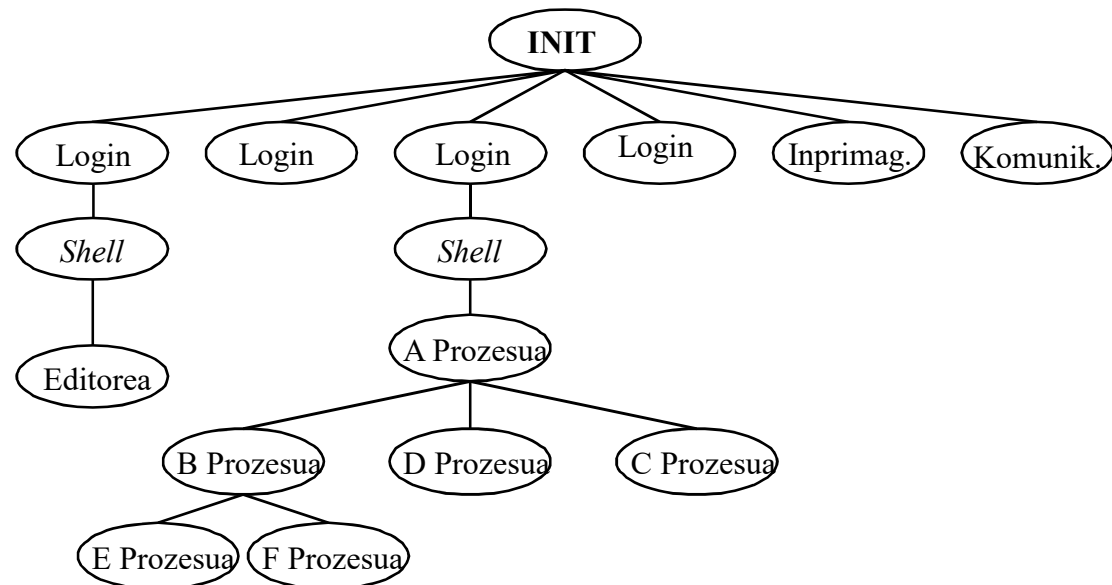
- **Planifikatzailea:**
 - Exekutatuko den hurrengo prozesua zein den erabakitzen duen SE-aren modulua
 - Prest ilara hutsik badago: Prozesu nagia (idle egoera)

3. PROZESUAK

3.1 PROZESUAK

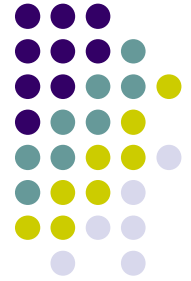


- Prosesuen **hierarkia**:
 - Aita, seme, anaiak... (generorik ez dutela pentsa)
 - Prosesuen bizi-zikloa
 - Sortu
 - Exekutatu
 - Hil edo amaitu
- Bi exekuzio modu:
 - **Batch** modua
 - **Interaktiboa**
- Prosesu-multzoak

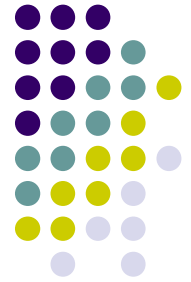


3. PROZESUAK

3.1 PROZESUAK



- Prosesuen **ingurunea**:
 - Prosesuari sorreran pasatzen zaizkion aldagaiak
 - SEk inguruneko aldagaien taula pasatzen dio programari bere exekuxioa hasi orduko.
 - Programak alda ditzake beraien balioak, baina aldaketa horiek bakarrik izango dute balioa prozesu horretan eta bere semeetan.
 - Prosesuaren hasierako pilan sartzen den IZEN-BALIO taula bat:
 - `int main(int argc, char **argv, char **envp);`
(edo `extern char **environ;`)



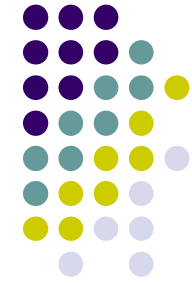
3. PROZESUAK

3.1 PROZESUAK

- Prosesuen **ingurunea**:
 - Ingurunea jartzeko modua:
 - Defektuz aitarengandik jasotzen dena
 - Shelleko aginduen bidez (export)
 - SE-aren API-aren bidez (putenv, getenv)
 - Inguruneko aldagaien adibideak:
 - PATH=/usr/bin:/home/pepe/bin
 - TERM=vt100
 - HOME=/home/pepe
 - PWD
 - USER
 - Balioak lortzeko:
 - echo \$aldagaia: Shell
 - getenv("aldagaia"): Shell eta C Kodean

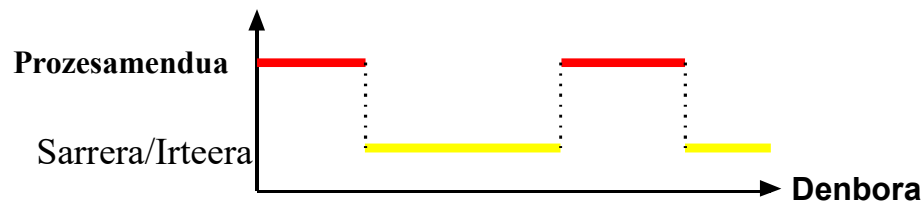
3. PROZESUAK

3.1 PROZESUAK



- **Multitaza edo Multiprogramazioa:**

- Ataza bakarreko sistema batean PUZak ez du ezer egiten S/I-ren zain dagoen bitartean:



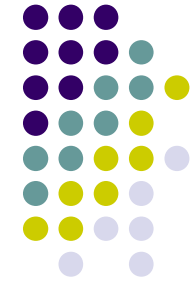
		Prozesu-kopurua	
		1	≥ 1
Erabiltzaileak	1	Prozesu bakarra Erabiltzaile bakar	Multitaza Erabiltzaile bakar
	≥ 1	—	Multitaza Multierabiltzaile

- **Multitazaren oinarria:**

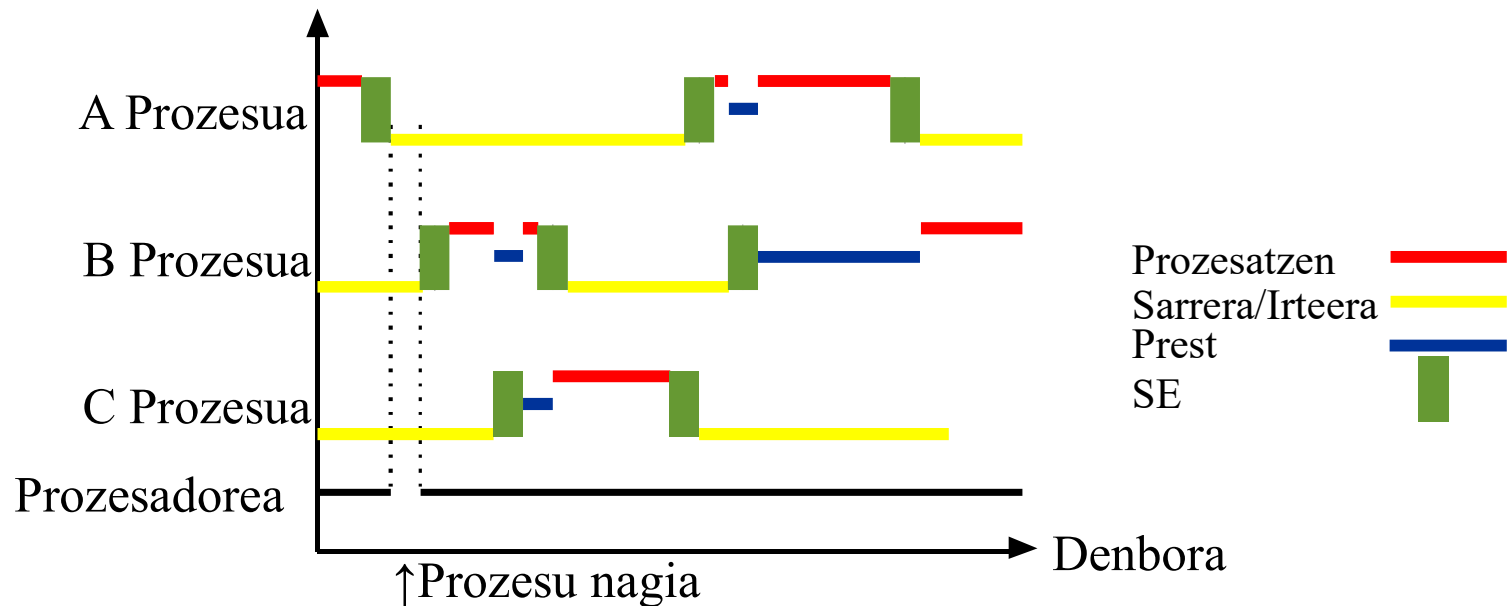
- PUZ eta S/I-ren arteko paralelismoa
- Prozesuen S/I SE-aren menpe utzi, PUZa, Prest egoeran dagoen beste prozesu bati emanaz
- Memoria “nagusian” prozesu bat baino gehiago gordetzeko gaitasuna

3. PROZESUAK

3.1 PROZESUAK



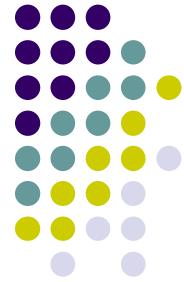
- **Multitaza** duen sistema batean:



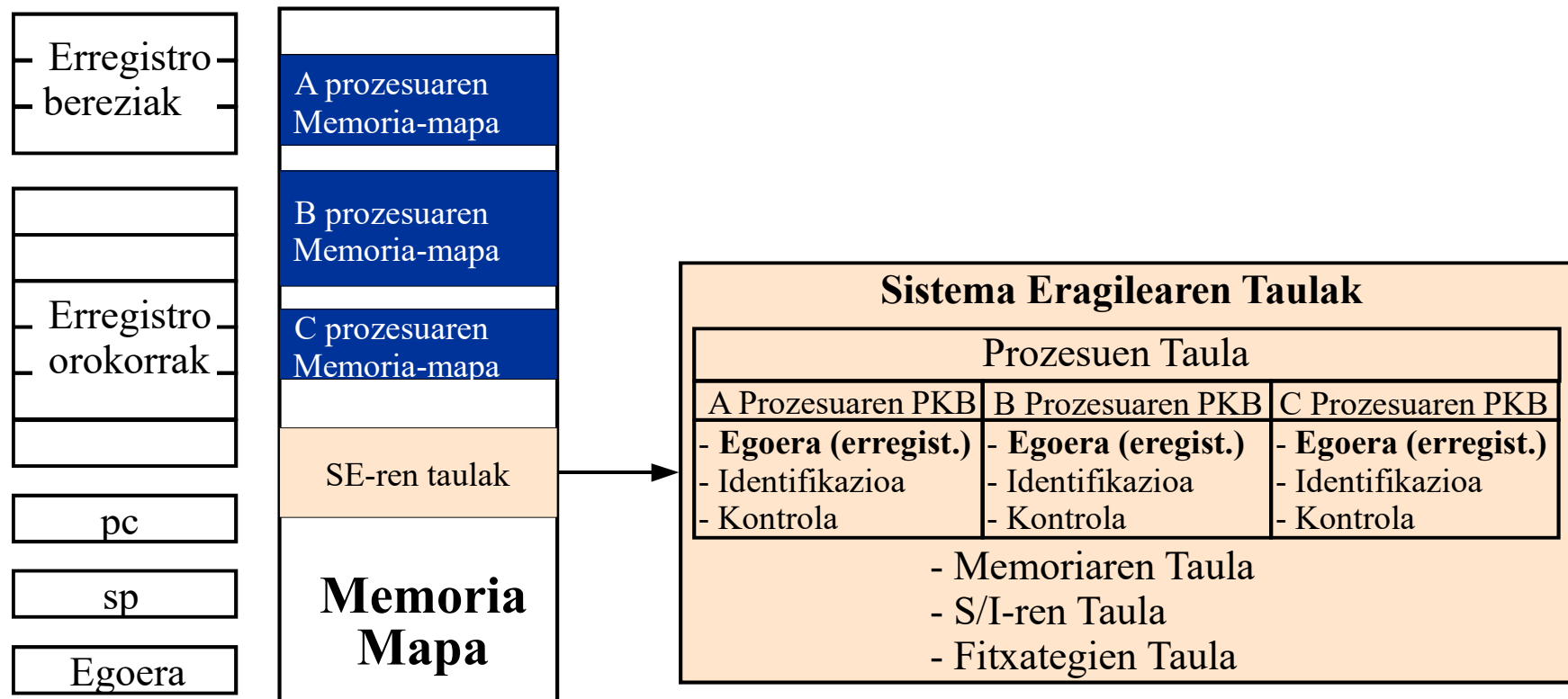
- Abantailak:
 - S/I hutsarteak aprobetxatu, PUZ-ari etekin hobea ateratzeko
 - Erabiltzaile interaktibo bat baino gehiago sisteman
 - Programazioa erraztu, programak prozesu desberdinetan exekutatu diren moduluetan banatuz

3. PROZESUAK

3.1 PROZESUAK



- **Prozesuaren egoera** eta prozesadorearen egoera:
 - Exekuzioan dagoen prozesu baten egoera prozesadorean dago.
 - Exekuzioan ez badago PKB-an dago, **Prozesuen Taulan**.



3. PROZESUAK

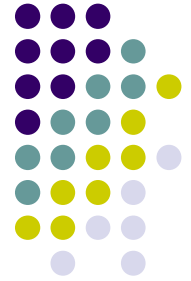
3.1 PROZESUAK



- **PKB-ko informazioa Prozesuen Taulan:**
 - Identifikazioa: Prozesuarena (PID), aitarena (PPID), erabiltzailearena (uid, euid), taldearena (gid, egid)
 - Prozesadorearen egoera: (erreg, pc, sp, psw)
 - Kontroleko egoera
 - Planifikaziorako egoera (prest, blokeatuta...)
 - Prozesuen arteko komunikazioa (mezuak nora edo nondik).
 - Seinaleak, alarma edo tenporizadoreak.
 - Prozesuak eta semeek erabilitako PUZ denbora (sistema, erabiltzaile)
- **Memoriaren Taula:**
 - Prozesuari esleitutako segmentuen (C+D+S) informazioa.
- **S/I-aren Taula:**
 - Esleitutako periferikoak eta S/I eragiketak.
- **Fitxategien taula:**
 - Irekitako fitxategien deskribatzaileak, (kokapenetarako FILP sarrerak).

3. PROZESUAK

3.1 PROZESUAK



- Nola lortu PUZ bakar batekin eta S/I gailu desberdinekin **paralelismo** ilusioa?
 - S/I gailuei etendura bektore bana dagokie.
 - Bektore bakoitzean zerbitzu baten helbidea dago (atentzio errutina).
 - S/I gailuen kontrolatzaileak blokeatuta egongo dira eskaeren zain.
 - IRQa gertatzen denean HW-ak pc, programaren egoera eta erregistroak gordetzen ditu. Ondoren, atentzio errutinaren helbidea kargatzen da.
 - Zerbitzu errutinetan datza ilusio horren mamia: gailuari dagokion kontrolatzailea desblokeatuko du, prest egoerara pasa eraziz.

3. PROZESUAK

3.1 PROZESUAK



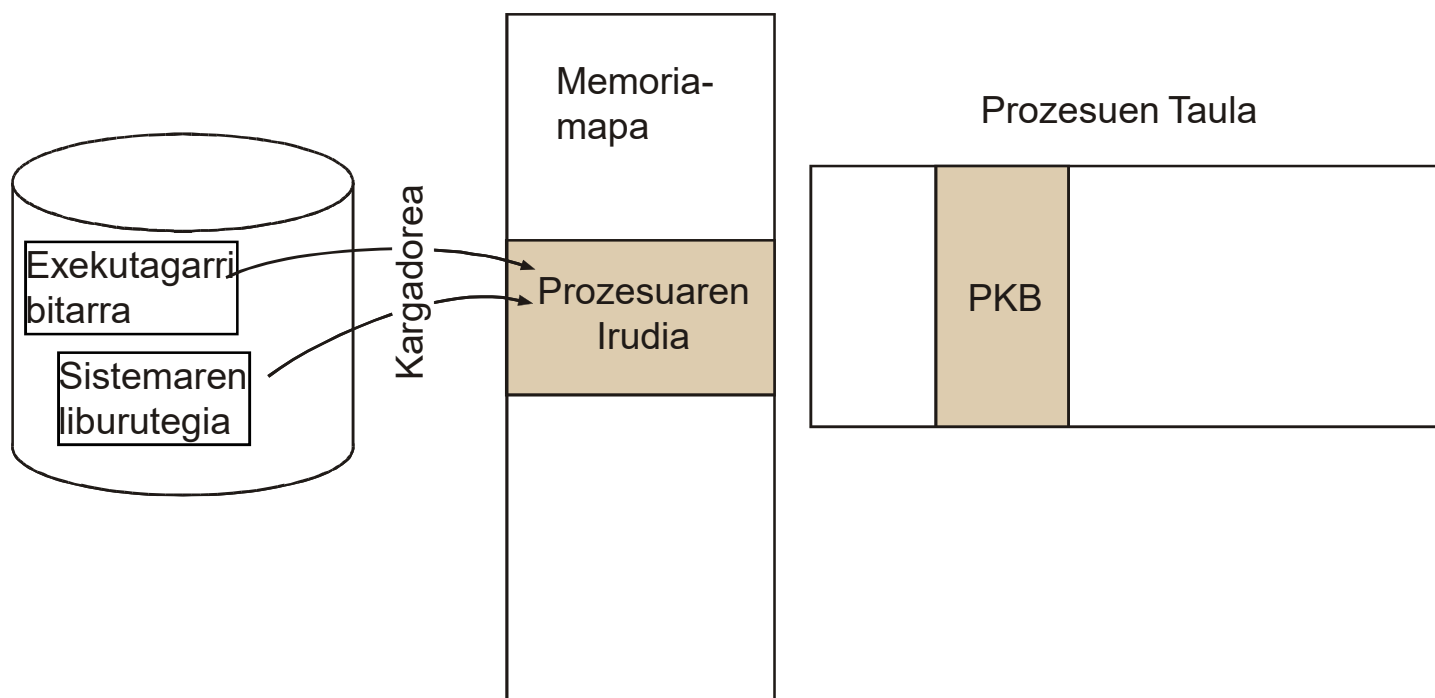
- **Testuinguru-aldaketa:**
 - Etendura bat gertatzen denean, exekuzioan dagoen prozesua (edo nagia) prest egoera pasatuko da.
 - Testuinguru-aldaketa bi eragiketaren multzoa da:
 - 1) Prozesadorearen egoera dagokion PKB-an gordetzen da
 - 2) SE etendura horri dagokion tratamendu-errutina exekutatzeraz pasatuko da. Horretarako:
- **Planifikatzailea (scheduler):** Exekutatuko den hurrengo prozesua zein den erabakitzen duen SE-aren modulua.
- **Aktibatzailea (dispatcher):** Hautatutako prozesua exekuzioan jartzen duen SE-aren modulua.
 - PKB-ko egoera PUZ-an kopiatzen du
 - Etenduraren tratamendu errutina bukatzen du RETI eginez (return from interrupt)
 - Egoerako erregistroa aldatzen du (exekuzio-mailaren bita)
 - pc hautatutako prozesuaren kokapenera begira jartzen du

3. PROZESUAK

3.1 PROZESUAK

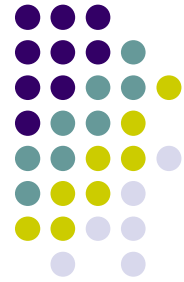


- **Prozesu** baten karga:

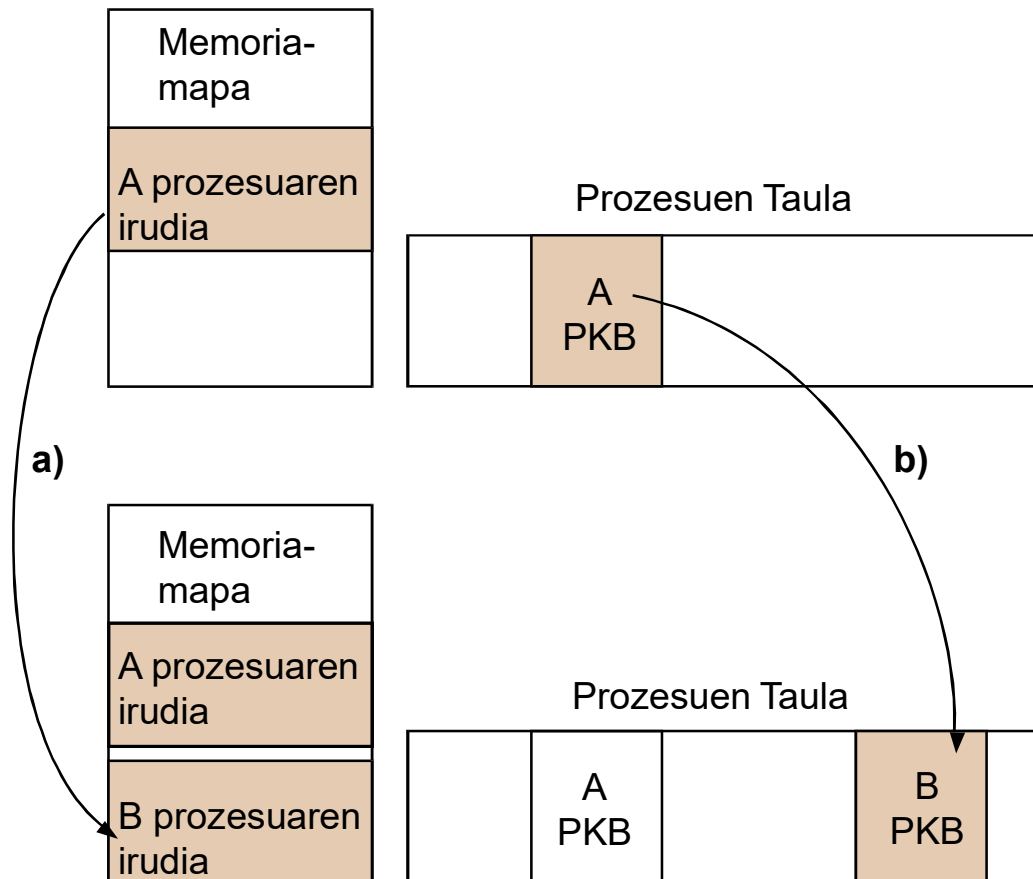


3. PROZESUAK

3.1 PROZESUAK



- Prozesu berriak sortzen, **fork()**: Aitaren “klonazioa”.



A prozesuak **fork()** eginez B semea sortzen du
a) Irudia hitzez-hitz kopiatzen da

b) B PKB-an A-ren kopia egiten da B egoera beretik has dadin: erregistroak, pc, sp, psw
Aldaketak:

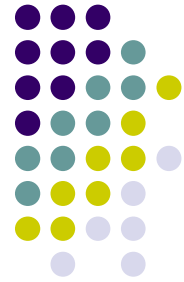
- B-ren itzarote egoeran dituen seinale eta alarmak, denborak (exekuzio-denbora), eta erakusleak zerora jartzen dira
- PID eta PPID desberdinak
- Memoria-espazio desberdinak-> atal berberak memoriako posizio desberdinetan
- Fitxategi berberak, kokapenak partekatzen

Itzulera-balio desberdinak:

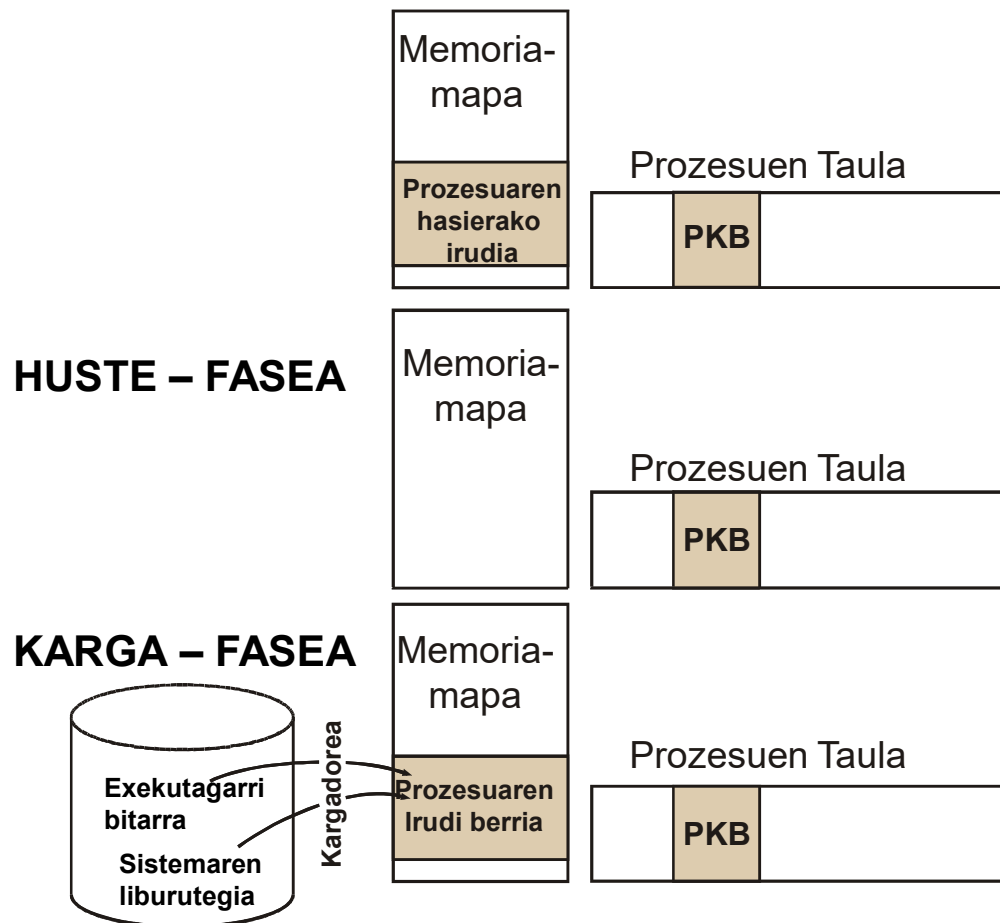
Aitari semearen **PID**
Semeari 0

3. PROZESUAK

3.1 PROZESUAK



- Prozesuaren programa aldatzen, **exec()**: “mutazioa”.



Prozesuak exec() deia egiten du.

ALDATU:

- Memoriako irudia ezabatzen da
- Memoriaren Taulako datuak hustu
- Seinaleak eta alarmak hustu

MANTENDU:

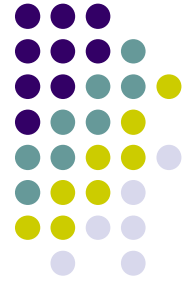
- Prozesuaren ingurunea mantendu
- PKB-ko PID, PPID, uid mantentzen dira
- Fitxategien fd-ak mantentzen dira

Prozesuari memoria tarte berri bat esleitu
Irudi berria kargatzen da.

pc hasierako helbidera
sp hasierako pilara

3. PROZESUAK

3.1 PROZESUAK

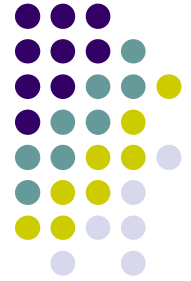


- FUNTZIO GOIBURUAK

```
int execl(const char *path, const char *arg0, ... /* (char *) NULL */);  
int execv(const char *path, char *const argv[]);  
int execl(const char *path, const char *arg0, ... /* (char *) NULL, char *const envp[] */);  
int execve(const char *path, char *const argv[], char *const envp[]);  
int execlp(const char *file, const char *arg0, ... /* (char *) NULL */);  
int execvp(const char *file, char *const argv[]);
```

3. PROZESUAK

3.1 PROZESUAK



- ADIBIDEA

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

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

```
main ()
```

```
{
```

```
    pid_t pid;
```

```
    int status;
```

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

```
    switch(pid){
```

```
        case -1: /* fork()-ek akatsa itzultzen du*/
```

```
            exit(-1);
```

```
        case 0: /*semea*/
```

```
            execlp("ls", "ls", "-l", NULL);
```

```
            perror("exec");
```

```
            break;
```

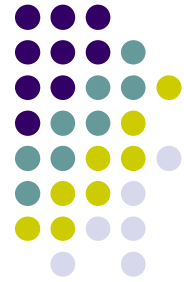
```
        default: /*aita*/
```

```
            printf("Aita prozesua\n");
```

```
    }
```

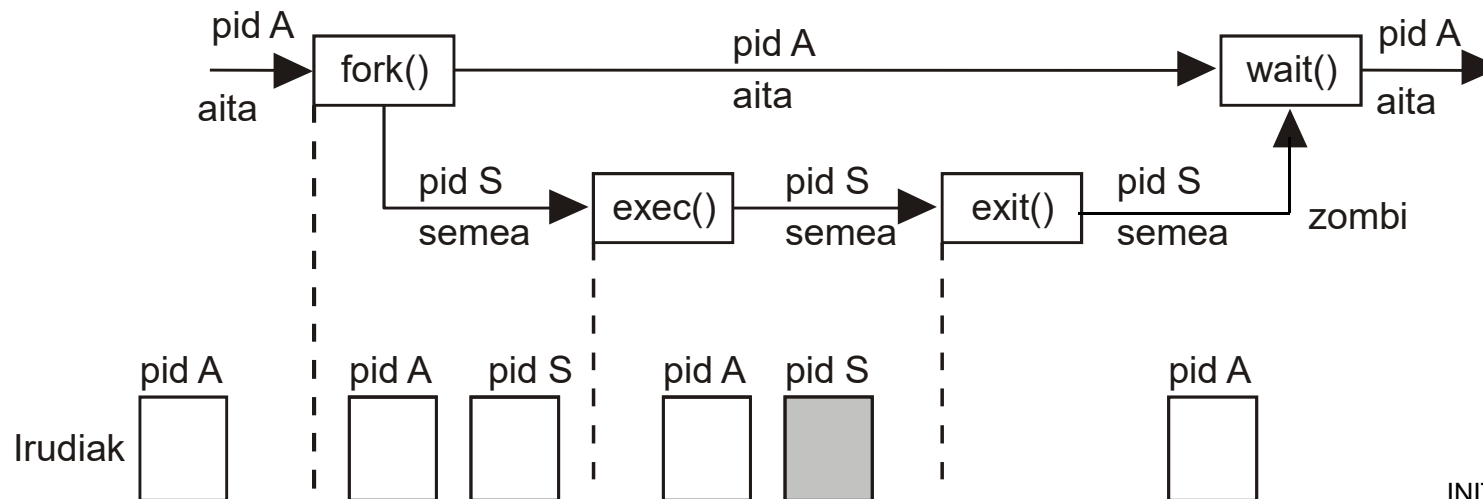
3. PROZESUAK

3.1 PROZESUAK



● Aita-semeen **bizi-zikloa**:

- Prosesuek semeak sortzen dituzte morroiak izateko
- Semeek kode berria exekutatu ondoren hil egiten dira
- Aita semearen zain ez dagoen bitartean, zombie egoera
- Aitak semearen heriotzaren berri jasotzean, aurrera egiten du



INIT-ek adoptatzen duenez:

- Aita semeari begira egon balitz, ez zen egongo zombirik
- Aita semeak baino lehenago hilez gero, INIT bihurtzen da umezurtzen aitaorde.

Prozesu zombi bat izateko egoera bakarra gurasoa bizirik egotea (wait egin gabe) eta umea hiltzea da.

3. PROZESUAK

3.1 PROZESUAK



- Prozesuei **Seinaleak**:

- Prozesuen artean bidalitako etendurak dira

- Sorburua

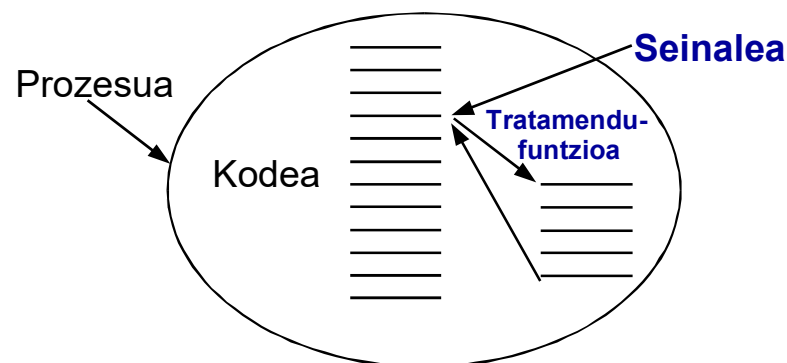
- Prozesutik – Prozesura: **kill** (pid, sigzbk)

- Erabiltzaile beraren prozesuen artean bakarrik (supererabiltzaileak edozeini)

- SE-tik – Prozesura

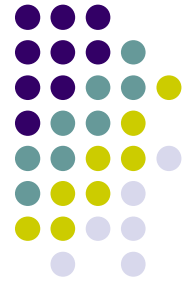
- HW salbuespenak, komunikazioa, teklatura, erlojua

Seinaleak ERABILTZAILE BEREAN bakarrik bidali daitezke!! (Admin izan ezik B)



3. PROZESUAK

3.1 PROZESUAK



- **Seinaleen tratamendu-funtzioak:**

- Jatorri bakoitzak bere sigzbk dauka. Adibideak:

- SIGILL HW agindu ilegalak
- SIGALRM tenporizadore baten amaiera
- SIGKILL prozesua akabatzeko
- SIGUSR1 eta SIGUSR2 erabiltzailearenak

- **Prozesua seinalea jasotzeko prest egon behar da**

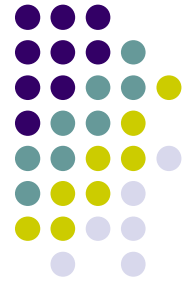
- Seinale bakoitzari tratamendu-funtzio bat esleitu behar zaio:
 - **signal(sigzbk, funtzioaren_izena);**
- ez badago prest → defektuz dagoena edo hil (normalena)

- **Funtzio/sistema-deiak**

- **pause()** zerbitzuak prozesua geldiarazten du seinale bat (edozein) jaso arte
- **alarm(segundoak)** prozesuari SIGALRM bidaltzen dio
 - alarm(0) aurreko eskaera guztiak ezgaitzen ditu

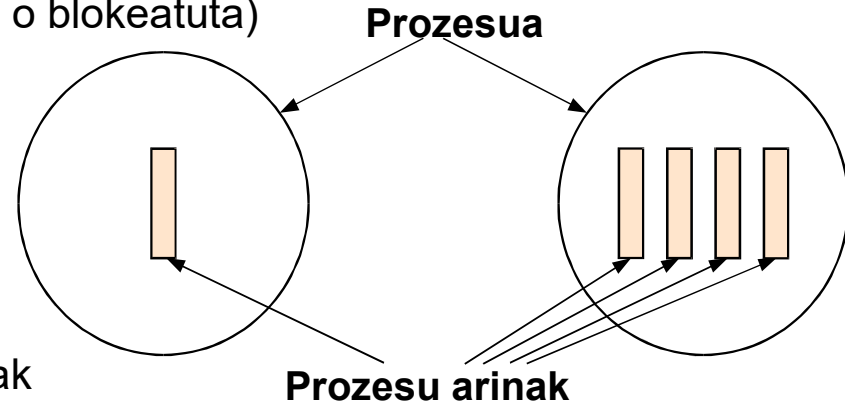
3. PROZESUAK

3.2 PROZESU ARINAK EDO HARIAK



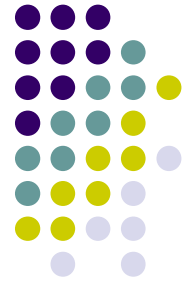
● Thread, Hariak edo Prozesu Arinak:

- Exekuzioan dagoen prozesu batek memoriako irudia eta beste zenbait informazio partekatzen du exekuzio hari desberdinekin
- Hari bakoitzak
 - Bere pc, erregistroak
 - Bere pila eta sp propioak
 - Egoera propioa (exekutatzen, prest o blokeatuta)
- Prozesuak
 - Memoria-espazioa
 - Aldagai orokorrak
 - Irekitako fitxategiak
 - Prozesu seme arruntak
 - Tenporizadoreak, seinaleak, alarmak
 - Kontabilitatea



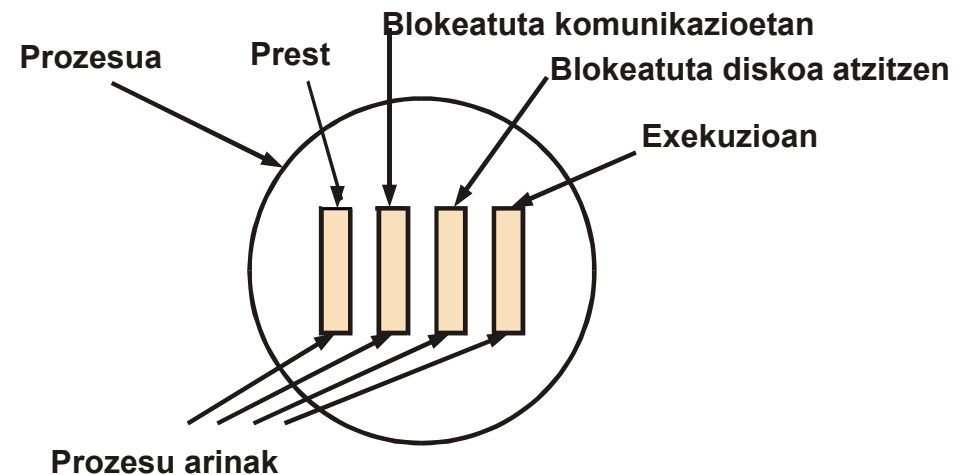
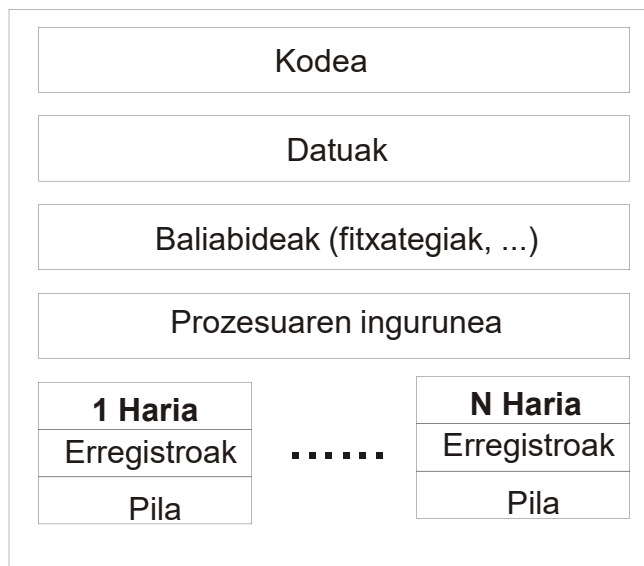
3. PROZESUAK

3.2 PROZESU ARINAK EDO HARIAK



- Prozesu Arinaren egoerak:

Prozesua



3. PROZESUAK

3.2 PROZESU ARINAK EDO HARIAK



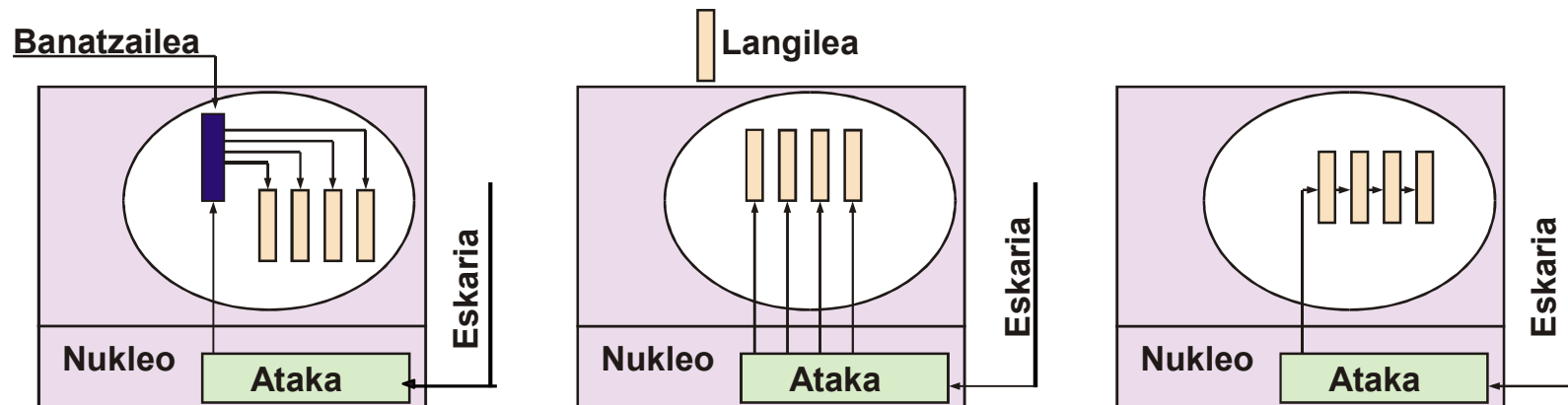
- **Prozesu arinen abantailak:**
 - **Hariek memoria partekatzen dute zuzenean** (abantailak eta arriskuak)
 - Hariak sortzeak eta askatzeak **zama gutxiago sistemarako**
 - Paralelismoa eta partekatutako aldagaiak
 - Sistemara eginiko deiek prozesu arina bakarrik blokeatzen dute
 - **Atazak bereizten eta banatzen ahalbidetzen dute**
 - Lanaren abiadura handitu egiten da
 - **Programazio konkurrentea (aldagaiak partekatuz):**
 - Partekatutako **aldagaiak atzitzen akatsak egin daitezke.**
 - **Sinkronizazio-mekanismoen premia (mutex).**
 - Kontuan izan hariekin diseinuak egiterakoan.

3. PROZESUAK

3.2 PROZESU ARINAK EDO HARIAK

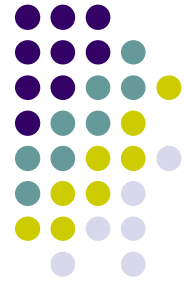


- Zerbitzari multiharien (*multithread*) diseinuak:
 - Banatzaile batek eskariak entzun eta berauei erantzuteko hariak sortu (edo birziklatu) egiten ditu
 - Maila bereko hari multzo bat eskariak jasotzen eta erantzuten
 - Lanak zati edo fase berezietan banatzen dira, eta katean edo *pipe-line* moduan exekutatzen dira



3. PROZESUAK

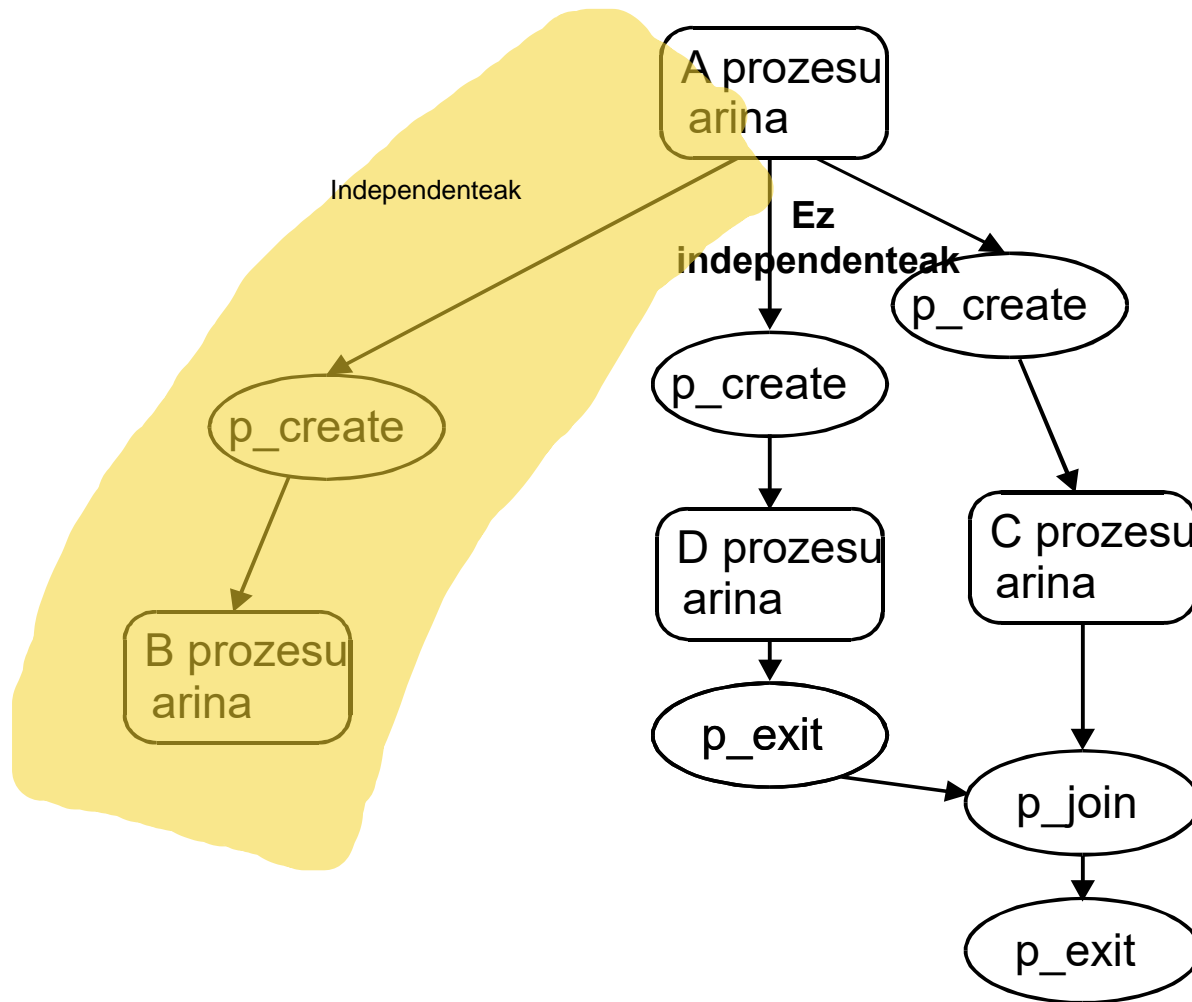
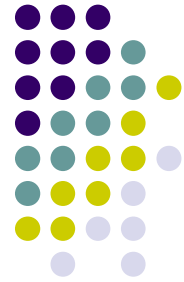
3.2 PROZESU ARINAK EDO HARIAK



- **int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*func)(void *), void *arg)**
 - “attr” ezaugarriak hari bat sortu, “func” funtzioa exekutzeko “arg” pasatutako argumentuekin.
 - Atributuetan honakoak: pilaren luzera, lehentasuna, planifikazio-politika, etab.
 - Atributuak aldatzeko dei-multzo bat dago.
- **int pthread_join(pthread_t thid, void **value)**
 - Exekuzioan dagoen hari “thid” zenbakiko hariaren amaieraren zain gelditzen da blokeatuta.
 - waitpid() funtzioaren baliokidea
 - Prozesu arinaren amaierako balio itzultzen du.
- **int pthread_exit(void *value)**
 - Hari baten amaieran, bere amaieraren emaitza itzultzeko.
- **pthread_t pthread_self(void)**
 - Hariaren identifikatzailea. Hariak nor den jakin dezan.
- **int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate)**
 - Hariaren amaiera-mota finkatzeko funtzioa.
 - Baldin “detachstate” = PTHREAD_CREATE_DETACHED baliabideak askatzen dira exekuzioaren amaieran.
 - Baldin “detachstate” = PTHREAD_CREATE_JOINABLE ez dira askatzen baliabideak pthread_join() egin arte.

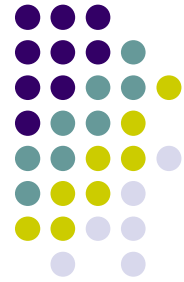
3. PROZESUAK

3.2 PROZESU ARINAK EDO HARIAK



3. PROZESUAK

3.3 PROZESUEN ARTEKO KOMUNIKAZIOA



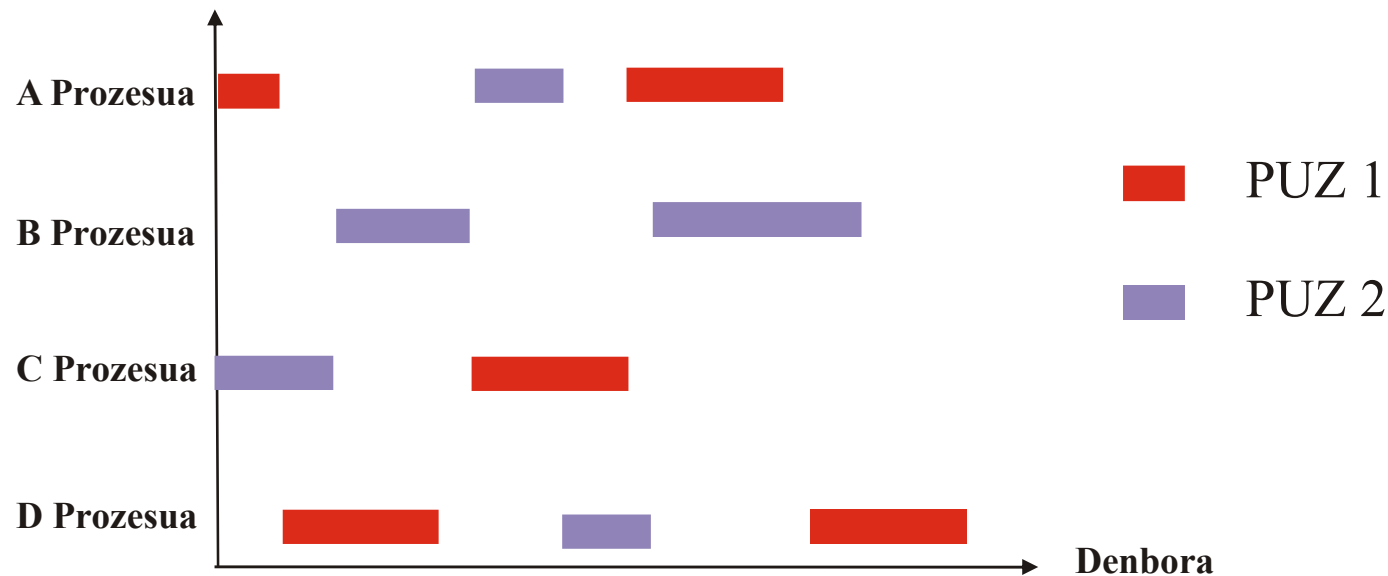
- Zertarako Konkurrentzia:
 - Baliabide fisiko/logikoen ustiapena hobetzeko
 - Modularitatea, prozesu lankideak
 - Kalkuluak azkartu, paralelismoa
 - Interakzioa hobetzeko
- Prozesu konkurrenteen motak
 - Lankideak
 - Independentek
- Interakzio motak
 - Baliabideak partekatu/baliabideengatik lehiatu
 - Komunikatu eta sinkronizatu: lankideak
- SE-ak → Prozesuak komunikatzeko/sinkronizatzekeo zerbitzuak eskaini behar ditu

3. PROZESUAK

3.3 PROZESUEN ARTEKO KOMUNIKAZIOA

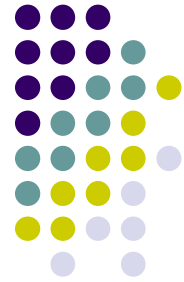


- Konputagailu motak:
 - Multiprogramazioa prozesadore bakarraz (itxurazko konkurrentzia)
 - **Sistema multiprozesadorea** (konkurrentzia osoa)
 - Multikonputagailua (prozesu banatuak)



3. PROZESUAK

3.3 PROZESUEN ARTEKO KOMUNIKAZIOA



● IPC-PAK Arazo klasikoak:

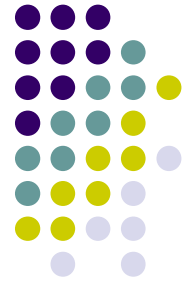
Inter-Process Communication

1. Atal Kritikoa
2. Ekoizle-kontsumitzailea
3. Irakurle-idazleen arazoa
4. Bezzero-zerbitzaria

- SE-ak arazo hauei aurre egiteko mekanismoak eskaini behar ditu

3. PROZESUAK

3.3 PROZESUEN ARTEKO KOMUNIKAZIOA



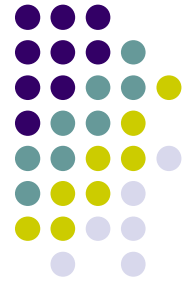
- Atal Kritikoa:
 - Demagun n prozesu dituen sistema bat. Denek partekatzen dituzte elementuak beren kodearen zatiren batean: [Atal kritikoak \(AK\)](#)
 - Bi prozesu edo gehiagok batera AK exekutatzen badute lehia baldintzak gerta daitezke (batzuetan bai, bestetan ez)
 - Exekuzio konkurrentearen arazorik ohikoena: bai lankideekin, bai independenteekin
 - Adibide bat: Balizko kontu korrante batean eragiketak egiten

```
void dirua_sartu(char *kontua, int kantitatea)
{
    int saldoa;
    int fd;

    fd = open(kontua, O_RDWR);
    read(fd, &saldoa, sizeof(int));
    saldoa = saldoa + kantitatea;
    lseek(fd, 0, SEEK_SET);
    write(fd, &saldoa, sizeof(int));
    close(fd);
}
```


3. PROZESUAK

3.3 PROZESUEN ARTEKO KOMUNIKAZIOA



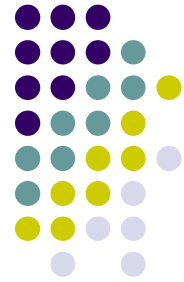
- Atal Kritikoaren ebazpenak bete behar dituen baldintzak:
 - **Elkarren arteko baztertzea:** Ezingo dira bi prozesu batera egon atal kritikoan.
 - **Aurrerapena:** Gune kritikotik kanpo dauden prozesuek ezingo dute beste prozesu bat blokeatu
 - **Itxaronaldi mugatua:** Prozesuek ez dute betiko itxaron beharko gune kritikoan sartzeko

```
void dirua_sartu(char *kontua, int kantitatea)
{
    int saldoa;
    int fd;

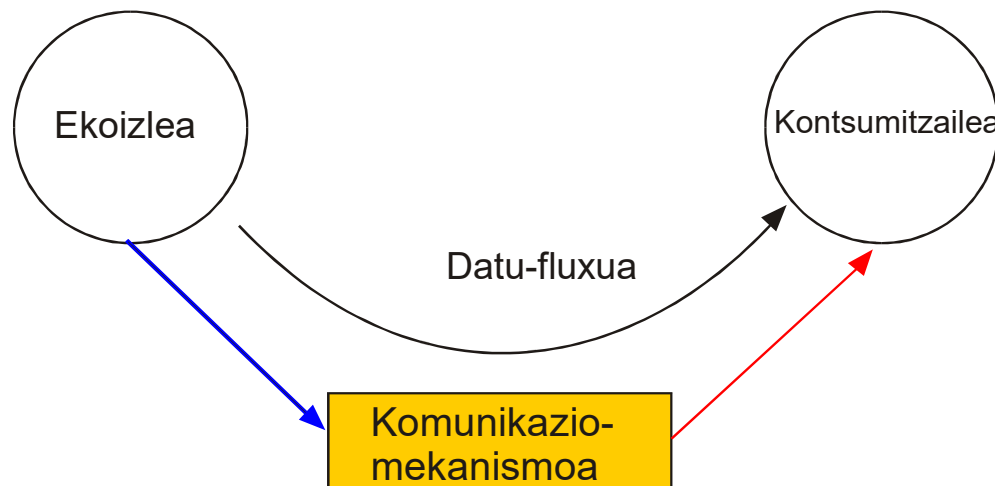
    fd = open(kontua, O_RDWR);
    <Atal kritikoan sartu>
    read(fd, &saldoa, sizeof(int));
    saldoa = saldoa + kantitatea;
    lseek(fd, 0, SEEK_SET);
    write(fd, &saldoa, sizeof(int));
    <Atal kritikotik irten>
    close(fd);
}
```

3. PROZESUAK

3.3 PROZESUEN ARTEKO KOMUNIKAZIOA

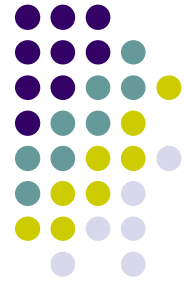


- Ekoizle-kontsumitzailearen arazoa:
 - Ekoizleak eta kontsumitzaileak komunikatu eta sinkronizatu, mugatuta dagoen komunikazio-mekanismo baten bidez.
 - Komunikazio-mekanismoa betetik dagonean, ekoizlea blokeatu.
 - Hutsik dagoenean, kontsumitzailea.

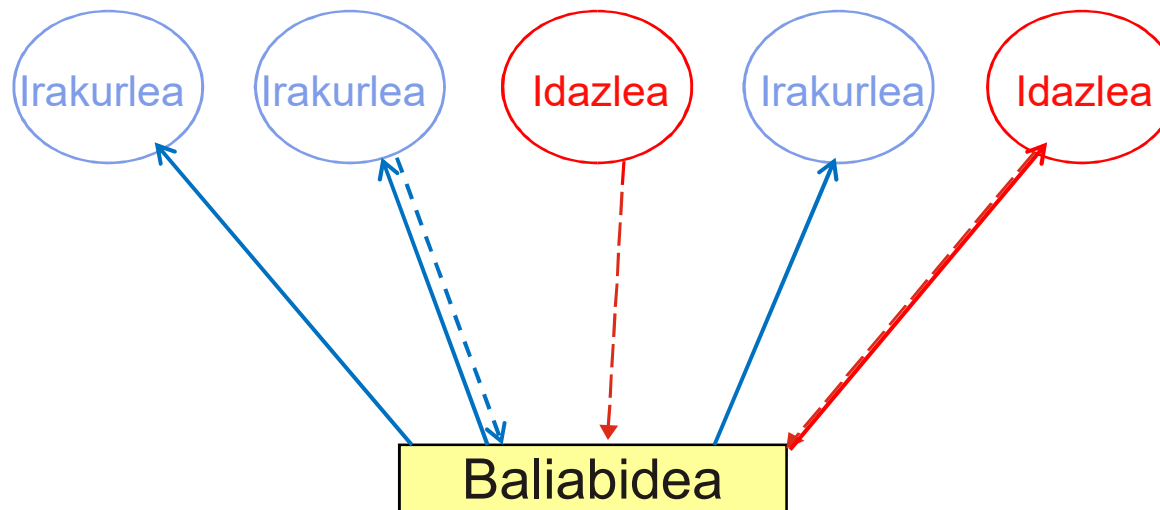


3. PROZESUAK

3.3 PROZESUEN ARTEKO KOMUNIKAZIOA

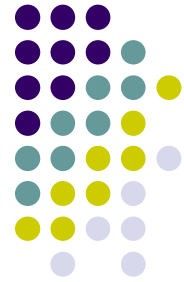


- Irakurle-idazleen arazoa:
 - Atzipen modua: Modu partekatua/Ardura bakarrekoa modelizatzen du.
 - Fitxategi edo DB base baten atzipena

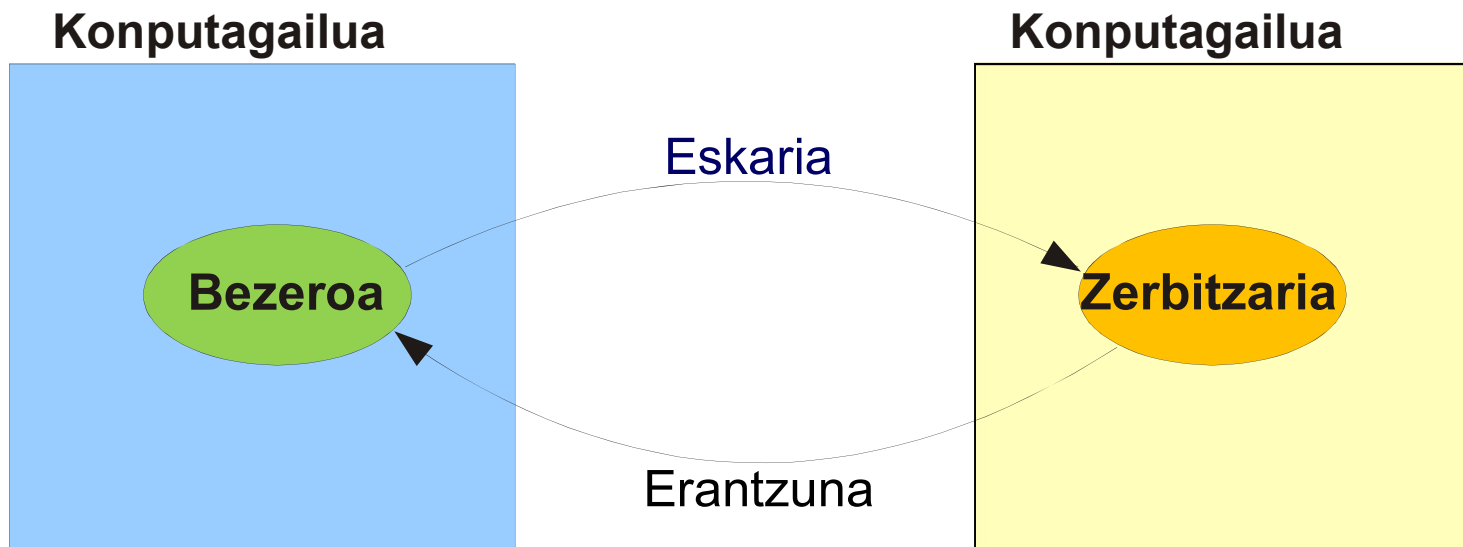


3. PROZESUAK

3.3 PROZESUEN ARTEKO KOMUNIKAZIOA



- Bezero-zerbitzariaren arazoa:
 - SE-aren mekanismoak bezeroak eta zerbitzariak komunikatzeko eta sinkronizatzeko:
 - Sasi-fitxategietan eta partekatutako memorian oinarritutakoak
 - Makina berean
 - Mezuetan oinarritutakoak
 - Makina berean
 - Makina desberdinen artean → Sareko zerbitzuak (socketak)



3. PROZESUAK

3.4 KOMUNIKAZIO eta SINKRONIZAZIO MEKANISMOAK



- SE-aren **Komunikazio mekanismoak**:
 - Prosesuen artean datuak trukatzeko ahalbidetzen dute
 - Garrantzitsuenak:
 - Fitxategiak
 - Hodiak (pipe, FIFO)
 - Partekatutako memoria
 - Mezuak

3. PROZESUAK

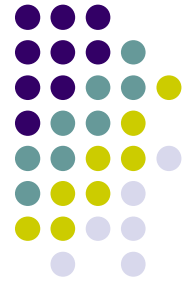
3.4 KOMUNIKAZIO eta SINKRONIZAZIO MEKANISMOAK



- SE-aren **Sinkronizazio mekanismoak**:
 - Prosesuak blokeatu/desblokeatzen ahalbidetzen dute zenbait gertaeren aurrean
 - Garrantzitsuenak:
 - Seinaleak (asinkronismoa)
 - Hodiak (pipe, FIFO)
 - Semaforoak
 - Mutex eta baldintza-aldagaiak
 - Mezuak
 - Sinkronizazio eragiketek **atomikoak** izan behar dute
 - Badira ere zenbait programazio lengoai konkurrenteren erremintak (monitoreak, hariak), baina ez daude SE-ren menpe

3. PROZESUAK

3.4 KOMUNIKAZIO eta SINKRONIZAZIO MEKANISMOAK



- Fitxategiak:
 - Komunikaziorako soilik
 - Abantailak:
 - Prozesu kopuru mugagabea, sarbide baimendua badute
 - Fitxategi zerbitzariak erabilerrazak dira
 - Desabantailak
 - Errendimendu txarra (R/W astuna)
 - Sinkronizazio mekanismo baten beharra dute
 - Normalean, ez dira erabiltzen

3. PROZESUAK

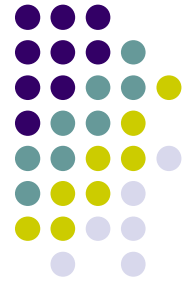
3.4 KOMUNIKAZIO eta SINKRONIZAZIO MEKANISMOAK



- Seinaleak:
 - Sinkronizaziorako soilik
 - Blokeatze/desblokeatzearen simulazioa:
 - **pause()** seinale baten zain blokeatuta
 - **kill()** pausen blokeatuta egondako prozesua desblokeatu
 - Ez da egokia blokeatu/desblokeatzeko:
 - Izaera asinkronoa
 - Ez dira pilatzen: Mota bereko bi seinale jasotzen badira, azkena bakarrik entregatuko da

3. PROZESUAK

3.4 KOMUNIKAZIO eta SINKRONIZAZIO MEKANISMOAK

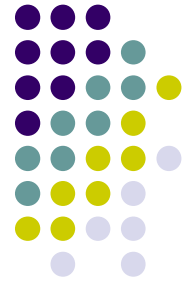


- Hodiak edo *pipe*
 - Komunikazio eta sinkronizazio mekanismoa
 - Informazio gordetzeko ahalmena
 - S.E.-ak mantentzen duen fitxategi antzeko bat
 - Irakurketarako eta idazketarako fitxategientzat erabiltzen diren funtzioak erabiltzen dira
 - Bi fitxategi-deskribatzaile:
 - `fildes[0]` irakurtzeko
 - `fildes[1]` idazteko

"fildes" izen arbitrario bat da!

3. PROZESUAK

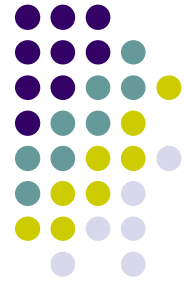
3.4 KOMUNIKAZIO eta SINKRONIZAZIO MEKANISMOAK



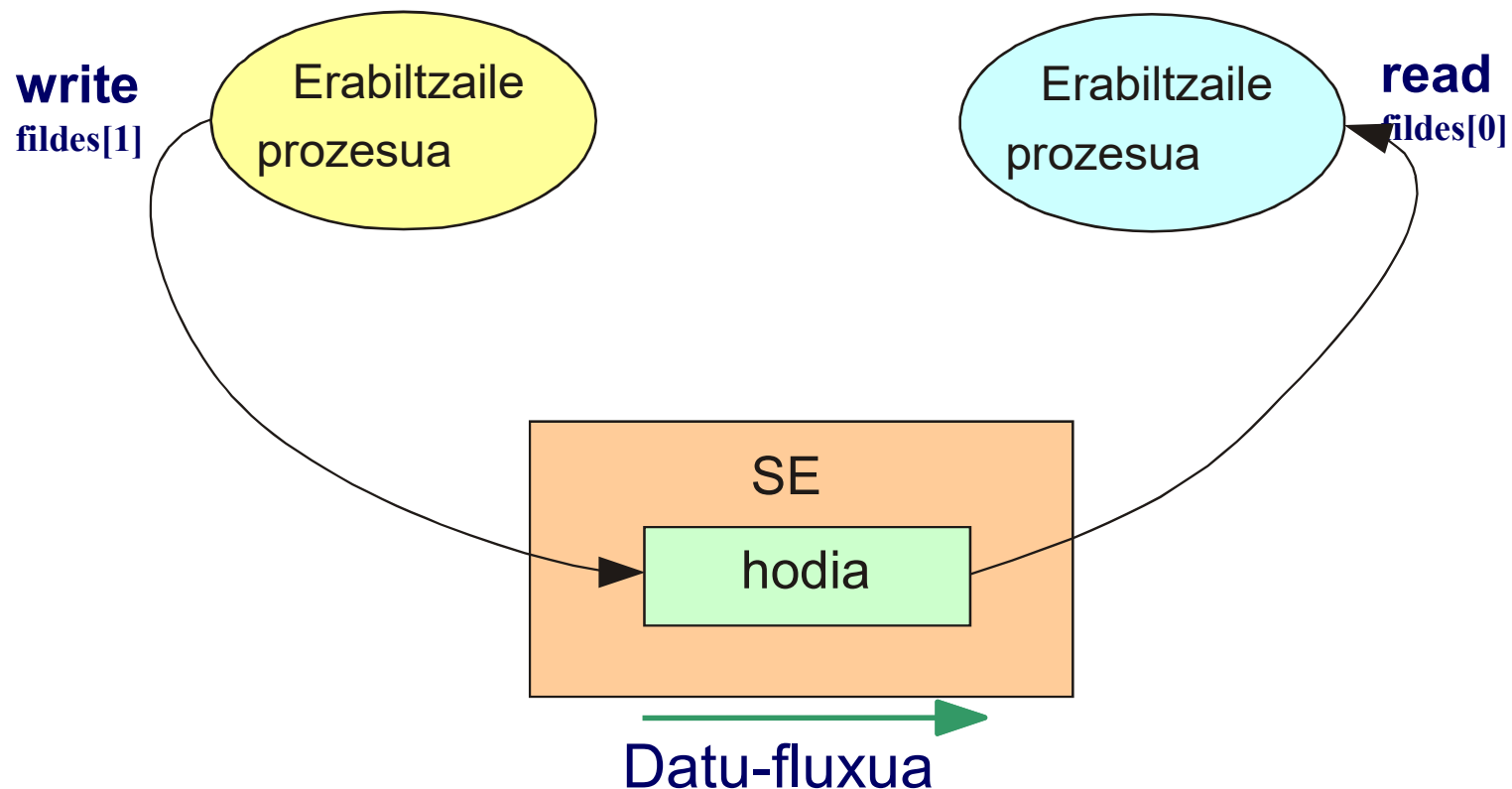
- Hodiak edo *pipe*
 - Motak
 - Izen gabeak edo **pipe**
 - Aita- seme prozesuak: prozesu sortzailearen hierarkiaren barruan (interpretazioa aplikazioaren ardura pean)
 - `int pipe(int fildes[2]);`
 - Izendunak *named* edo **FIFO**
 - Prozesu independenteak
 - Lokala edo sarekoa `#include <sys/types.h>`
 - `int mkfifo(const char *pathname, mode_t mode);`
 - Datu-fluxua: noranzko bakarrean
 - Bi norantzetan izan dadin, bi hodi sortu behar dira

3. PROZESUAK

3.4 KOMUNIKAZIO eta SINKRONIZAZIO MEKANISMOAK

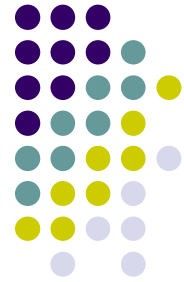


- Hodiak: Datu-fluxua noranzko bakarrekoa

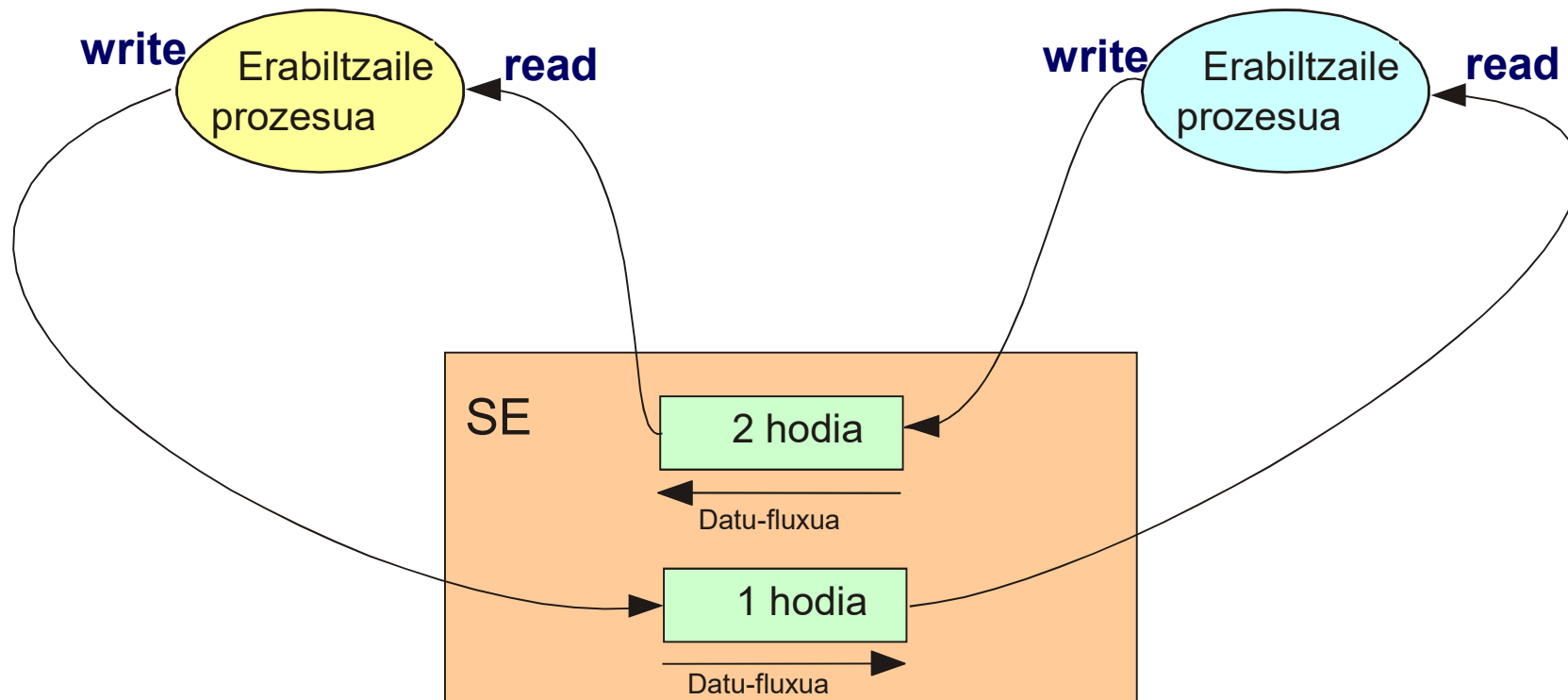


3. PROZESUAK

3.4 KOMUNIKAZIO eta SINKRONIZAZIO MEKANISMOAK

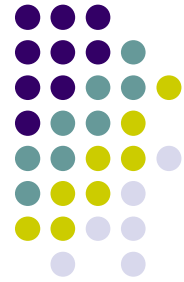


- Hodiak: Bi noranzkoetarako, hodi bana



3. PROZESUAK

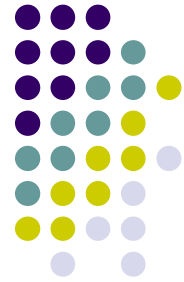
3.4 KOMUNIKAZIO eta SINKRONIZAZIO MEKANISMOAK



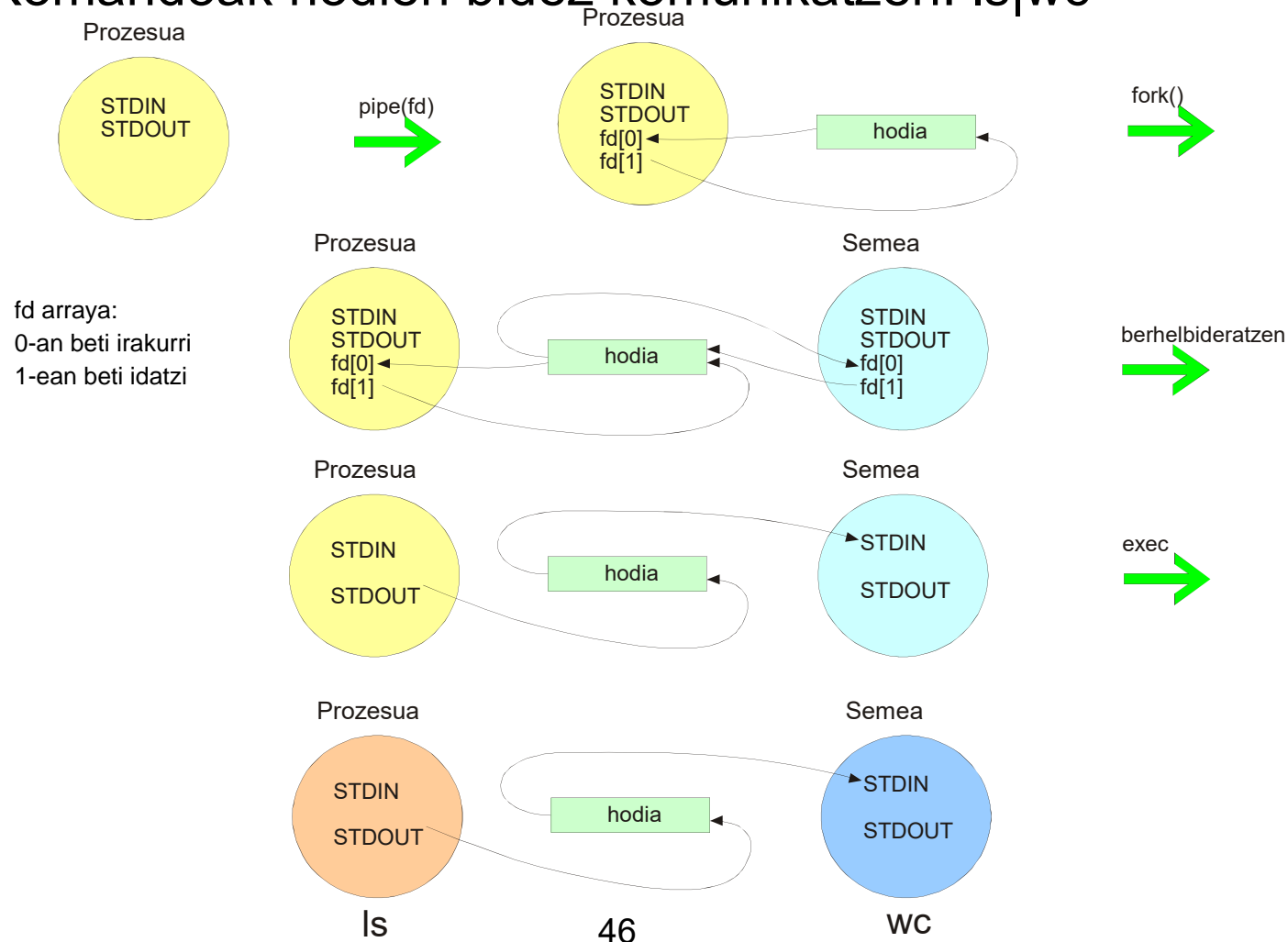
- Hodietan irakurri eta idatzi:
- **read**(`fildes[0]`, `buffer`, `n`)
 - Hodia hutsik badago, blokeatu egiten da
 - Hodian `p` byte (interpretazioa aplikazioen ardurapean)
 - Baldin $p \geq n$, `n` itzuli
 - Baldin $p < n$, `p` itzuli
 - Hodia hutsik eta beste muturrean inor ere ez, 0 itzuli (`eof`)
- **write**(`fildes[1]`, `buffer`, `n`)
 - Hodia beterik badago, blokeatu
 - Beste muturrean inor ere ez, -1 itzuli (`errno=EPIPE`, `SIGPIPE`)
- Idazketak eta irakurketak, biak atomikoak:
 - Uneoro gehienez prozesu bat, gainerakoak blokeatuta

3. PROZESUAK

3.4 KOMUNIKAZIO eta SINKRONIZAZIO MEKANISMOAK

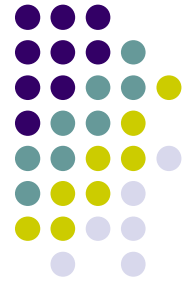


- Komandoak hodian bidez komunikatzen: ls|wc



3. PROZESUAK

3.4 KOMUNIKAZIO eta SINKRONIZAZIO MEKANISMOAK



- Komandoak hodian bidez komunikatzen: ls|wc

```
/* "ls | wc" komandoak hodi baten bidez lotzeko programa*/

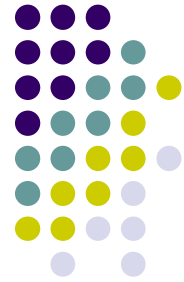
int main(void)
{
    int fd[2];
    pid_t pid;

    if (pipe(fd) < 0) {                               /* hodia sortu */
        perror("Errorea pipe sortzerakoan");
        return(0);
    }

    pid = fork();
    switch (pid) {
        case -1:                                       /* errorea */
            perror("Fork-en errorea");
            return(0);
    }
}
```

3. PROZESUAK

3.4 KOMUNIKAZIO eta SINKRONIZAZIO MEKANISMOAK



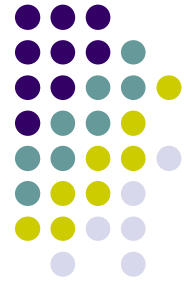
- Komandoak hodian bidez komunikatzen: ls|wc

```
case 0:                                /* semeak ls exekutatu */
    close(fd[0]);
    close(STDOUT_FILENO);
    dup(fd[1]); //Duplikatu -> standard-en zerbaiteira konektatu
    close(fd[1]);
    execlp("ls", "ls", NULL);
    perror("Exec-en errorea");
    break;

default:                               /* aitak wc exekutatu */
    close(fd[1]);
    close(STDIN_FILENO);
    dup(fd[0]);
    close(fd[0]);
    execlp("wc", "wc", NULL);
    perror(" Exec-en errorea ");
} /*switch*/
return(0);
} /*main*/
```


3. PROZESUAK

3.4 KOMUNIKAZIO eta SINKRONIZAZIO MEKANISMOAK



- Ekoizle-kontsumitzailearen arazoa hodian bidez:

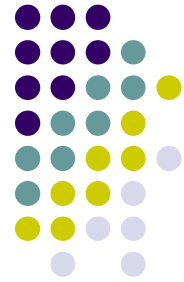
```
int main(void)
{
    int datua=0;                /* ekoizpena */
    int fildes[2];              /* hodia */
    pid_t pid;

    datua = 0;
    if (pipe(fildes) < 0){
        perror("Errorea pipe sortzerakoan");
        return (0);
    }

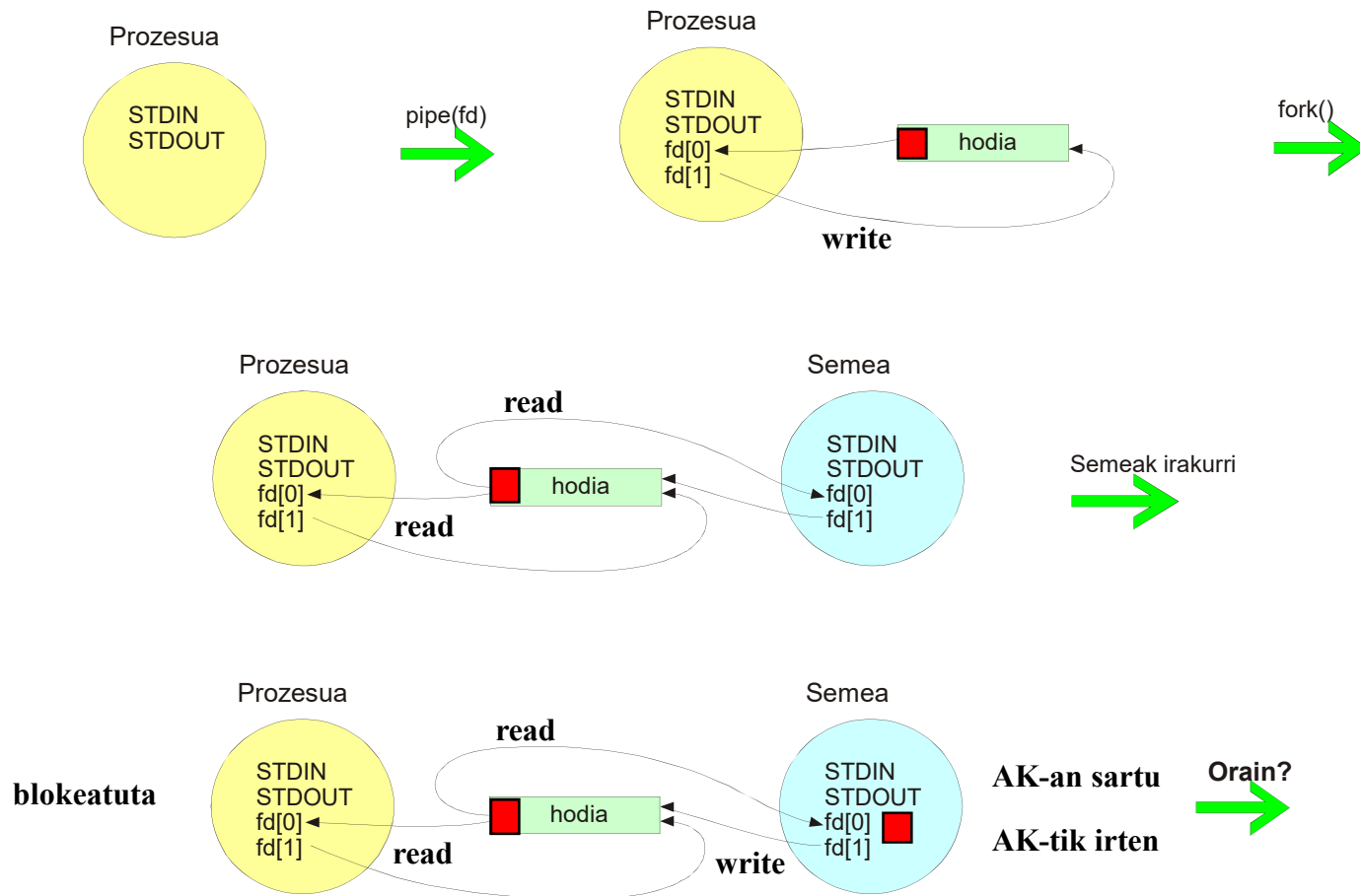
    if ( (pid=fork()) == 0){     /* semeak: 100 elementu ekoitzi */
        while(datua < 100) {
            printf ("Semeak %d datua sortu du\n", ++datua);
            write(fildes[1], (char *) &datua, sizeof(int));
        }
    }
    else if ( pid>0 ) {         /* aitak: 100 elementu kontsumitu */
        while(datua < 100) {
            read(fildes[0], (char *) &datua, sizeof(int));
            printf ("Aitak %d datua kontsumitu du\n", datua);
        }
    }
    return (0);
}
```

3. PROZESUAK

3.4 KOMUNIKAZIO eta SINKRONIZAZIO MEKANISMOAK

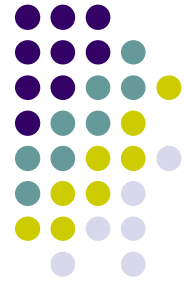


- Atal kritikoaren arazoa hodian bidez:



3. PROZESUAK

3.4 KOMUNIKAZIO eta SINKRONIZAZIO MEKANISMOAK



- Atal kritikoaren arazoa hodian bidez:

```
int main(void)
{
    int fildes[2];                /* Sinkronizaziorako hodia */
    char c = 'L';                /* Sinkronizatzeko erabiliko den lekukoa */
    pid_t pid;

    if( pipe(fildes) > 0) {
        write(fildes[1], &c, 1);  /* Lekukoa hodian sartu */
    } else {
        return(-1);
    }
    if ((pid=fork()) == 0){       /* semea */
        while(1) {
            read(fildes[0], &c, 1); /* <atal kritikoan sartu> */
            < Atal Kritikoa >
            write(fildes[1], &c, 1); /* <atal kritikotik irten> */
        }
    }
    else if ( pid > 0 ) {        /* aita */
        while(1) {
            read(fildes[0], &c, 1); /* <atal kritikoan sartu> */
            < Atal Kritikoa >
            write(fildes[1], &c, 1); /* <atal kritikotik irten> */
        }
    }
}
```

3. PROZESUAK

3.4 KOMUNIKAZIO eta SINKRONIZAZIO MEKANISMOAK



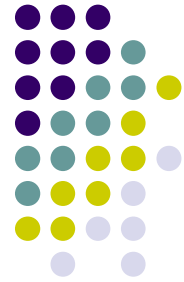
- **FIFO**, hodi izendunak:
 - Pipe bezalakoak, baina makina bereko prozesu independenteak komunikatzeko.

baimenak

- FIFO bat sortu: `int mkfifo(char *name, mode_t mode);`
- FIFO name izenduna (irakurtzeko, idazteko edo gauza bietarako):
 - `fd = open(char *name, int flag);`
 - Blokeatuta mutur bietatik ireki arte
- Irakurtzeko `read()` eta idazteko `write()`
 - Pipe-en semantika bera
- FIFOa ixteko `close()`
- FIFOa ezabatzeko `unlink()`
 - Horrela FIFOan gelditu den zaborra desagertu egiten da

3. PROZESUAK

3.4 KOMUNIKAZIO eta SINKRONIZAZIO MEKANISMOAK



- **Semaforoak:**

- Sinkronizazio Mekanismoa
- Makina bereko prozesuak sinkronizatzeko
- Objektu bat aldagai oso batekin: s
- Eta bi eragiketa **atomikorekin**

GOTO: 60

```
wait(s)    /* down */
{
    s = s - 1;
    if (s <= 0) {
        <Prozesua blokeatu>
    }
}
```

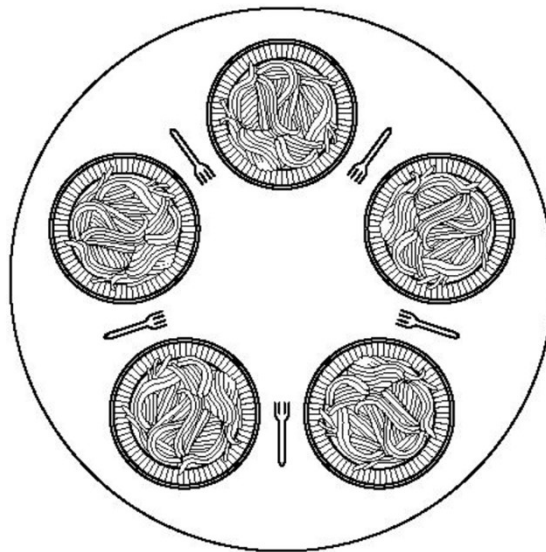
```
signal(s)    /* up */
{
    s = s + 1;
    if (s > 0)
        <wait egiten blokeatutako
        prozesu bat desblokeatu>
    }
}
```

3. PROZESUAK

3.4 KOMUNIKAZIO eta SINKRONIZAZIO MEKANISMOAK

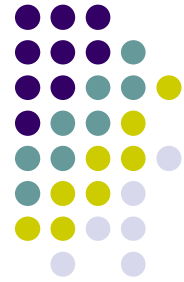


- Semaforoak eta Filosofoen afaria:
 - Dijkstra-ren ariketa bat semaforo-en gaitasuna erakusteko, 1965.
 - Mugatutako baliabideen erabilera partekatua erakusten du.
 - Filosofoek pentsatu egiten dute.
 - Gosetzen direnean, sardexkak binaka hartu eta bete arte jan.
 - Horrela, behin eta berriro
 - 5 filosofoek berdin jokatzeko dute



3. PROZESUAK

3.4 KOMUNIKAZIO eta SINKRONIZAZIO MEKANISMOAK



- Semaforoak eta Filosofoen afaria: 1. Ebazpena

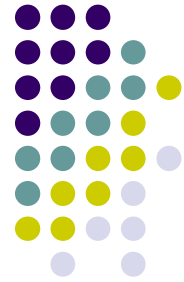
```
#define N 5                                /* number of philosophers */

void philosopher(int i)                    /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think();                          /* philosopher is thinking */
        → take_fork(i);                   /* take left fork */
        take_fork((i+1) % N);              /* take right fork; % is modulo operator */
        eat();                             /* yum-yum, spaghetti */
        put_fork(i);                       /* put left fork back on the table */
        put_fork((i+1) % N);               /* put right fork back on the table */
    }
}
```

- Bost filosofoek batera hartu dute ezkerreko sardexka → “**deadlock**” edo (elkar)blokeaketa.

3. PROZESUAK

3.4 KOMUNIKAZIO eta SINKRONIZAZIO MEKANISMOAK



- Semaforoak eta Filosofoen afaria: 2. Ebazpena

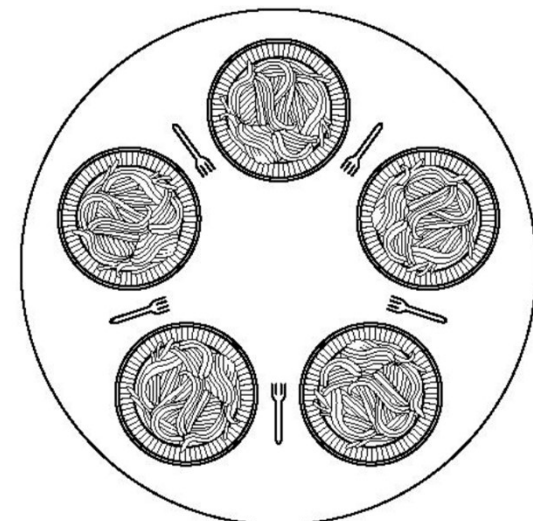
take_fork() funtzioa aldatu:

- Ezkerreko sardexka libre egon arte itxaron
- Ezkerreko sardexka hartu
- Eskuinekoa libre badago, hartu
- Libre ez badago:
 - Ezkerreko sardexka askatu
 - Denbora batean itxaron (1* aldaezina, 2* zorizkoa)
 - Errepikatu sardexka biak lortu arte.

1* **Starvation** edo **gosetea**

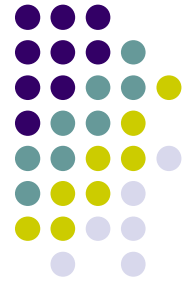
2* Ez dira gosez hiltzen, baina ez da oso eraginkorra

KONPONTZEKO: Mutex eta baldintza-aldagaiak



3. PROZESUAK

3.4 KOMUNIKAZIO eta SINKRONIZAZIO MEKANISMOAK



- Semaforoak eta Filosofoen afaria: 3. Ebazpena

```
#define N 5                                /* number of philosophers */
typedef int semaphore;                     /* semaphores are a special kind of int */
semaphore mutex = 1;                       /* mutual exclusion for critical regions */
sem_t mutex = 1;

void philosopher(int i)                    /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think();                           /* philosopher is thinking */
        down(mutex);                        /* enter critical region */
        take_fork(i);                       /* take left fork */
        take_fork((i+1) % N);               /* take right fork; % is modulo operator */
        eat();                             /* yum-yum, spaghetti */
        put_fork(i);                        /* put left fork back on the table */
        put_fork((i+1) % N);               /* put right fork back on the table */
        up(mutex);                          /* exit critical region */
    }
}
```

BELTZEZ: PSEUDOKODEA

HORIZ: BENETAKO KODEA

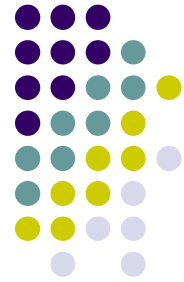
- Ez dago ez “*deadlock*”ik ez “*starvation*”ik
- Uneoro gehienez filosofo bat egon daiteke jaten

Kontuz! pthread-en mutex ere existitzen da, baina hori ez da guk erabiliko duguna.

“Mutual exclusion” harietan era askotan egin nahi da.

3. PROZESUAK

3.4 KOMUNIKAZIO eta SINKRONIZAZIO MEKANISMOAK



- Semaforoak eta Filosofoen afaria: 4. Ebazpena **paralelismoa**

```
#define N 5                                /* number of philosophers */
#define LEFT      (i-1)%N                  /* number of i's left neighbor */
#define RIGHT     (i+1)%N                  /* number of i's right neighbor */
#define THINKING  0                        /* philosopher is thinking */
#define HUNGRY    1                        /* philosopher is trying to get forks */
#define EATING    2                        /* philosopher is eating */
typedef int semaphore;                     /* semaphores are a special kind of int */
int state[N];                             /* array to keep track of everyone's state */
semaphore mutex = 1;                       /* mutual exclusion for critical regions */
semaphore s[N];                           /* one semaphore per philosopher */

void philosopher(int i)                    /* i: philosopher number, from 0 to 4 */
{
    state[i] = THINKING;
    s[i] = 0;
    while (TRUE) {
        think();                          /* philosopher is thinking */
        take_forks(i);                     /* acquire two forks or block */
        eat();                             /* yum-yum, spaghetti */
        put_forks(i);                      /* put both forks back on table */
    }
}
```

3. PROZESUAK

3.4 KOMUNIKAZIO eta SINKRONIZAZIO MEKANISMOAK



- Semaforoak eta Filosofoen afaria: 4. Ebazpena **paralelismoa**

```
void take_forks(int i)           /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);                /* enter critical region */
    state[i] = HUNGRY;           /* record fact that philosopher i is hungry */
    test(i);                     /* try to acquire 2 forks */
    up(&mutex);                  /* exit critical region */
    down(&s[i]);                  /* block if forks were not acquired */
}

void put_forks(int i)           /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);                /* enter critical region */
    state[i] = THINKING;         /* philosopher has finished eating */
    test(LEFT);                  /* see if left neighbor can now eat */
    test(RIGHT);                 /* see if right neighbor can now eat */
    up(&mutex);                  /* exit critical region */
}

void test(int i)                /* i: philosopher number, from 0 to N-1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

3. PROZESUAK

3.4 KOMUNIKAZIO eta SINKRONIZAZIO MEKANISMOAK



- POSIX Semaforoak:

- Izenik gabeko semaforo bat sortu eta desegin:

- `int sem_init(sem_t *sem, int shared, int val);`
 - `int sem_destroy(sem_t *sem);`

0: Hari guztiekin partekatzen da semaforoa!0: Gura

- Semaforo izendunak ireki, itxi eta ezabatu:

- `sem_t *sem_open(char *name, int flag, mode_t mode, int val);` SORTZEAN
 - `int sem_close(sem_t *sem);`
 - `int sem_unlink(char *name);`

- Wait (down) eta Signal (up) eragiketak:

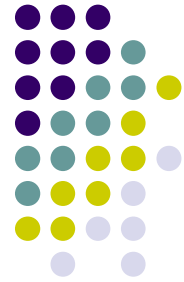
- `int sem_wait(sem_t *sem);`
 - `int sem_post(sem_t *sem);`

- Laborategian System V Semaforoak erabiliko ditugu.

Komunikatzeko ahalmenik ez duten semaforoak dira.

3. PROZESUAK

3.4 KOMUNIKAZIO eta SINKRONIZAZIO MEKANISMOAK



- Irakurle/Idazleen Arazoa POSIX Semaforoak erabiliz:

```
int datua = 100;
int n_irakurle = 0;
sem_t sem_irak;
sem_t mutex;

void main(void)
{
    pthread_t th1, th2, th3, th4;

    sem_init(&mutex, 0, 1);
    sem_init(&sem_irak, 0, 1);
    pthread_create(&th1, NULL, Irakurle, NULL);
    pthread_create(&th2, NULL, Idazle, NULL);
    pthread_create(&th3, NULL, Irakurle, NULL);
    pthread_create(&th4, NULL, Idazle, NULL);
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
    pthread_join(th3, NULL);
    pthread_join(th4, NULL);
    sem_destroy(&mutex);
    sem_destroy(&sem_irak);
}
```

/* Irakurri/Idatzi beharreko baliabidea */
/* Irakurle kopurua */
/* “n_irakurle” atzitzeko kontrola */
/* “datua” atzitzeko kontrola MUTual EXclusion*/

/* 2 Irakurle eta 2 idazle */

/* Semaforoak = 1 */

/* Hariak sortu */

Irakurle eta idazle hauek oso idealak dira, eta oso sekuentzialak.
Berez, edozein momentutan konektatukodira.

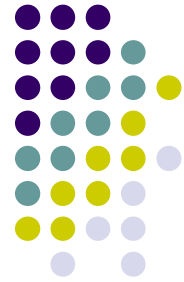
/* Harien zain gelditu */

Join-ak: Hari guztiak hil direnean egiten du exit-a
denak ariketa berdinarean parte direlako.

/* Semaforoak desegin */

3. PROZESUAK

3.4 KOMUNIKAZIO eta SINKRONIZAZIO MEKANISMOAK



- Irakurle/Idazleen Arazoa POSIX Semaforoak erabiliz:

```
/* Irakurlearen kodea */
```

```
void Irakurle(void) {
```

```
    sem_wait(&sem_irak);
```

```
    n_irakurle = n_irakurle + 1;
```

```
    if (n_irakurle == 1)
```

```
        sem_wait(&mutex);
```

```
    sem_post(&sem_irak);
```

```
    printf("%d\n", datua); /* Datua irakurri */
```

```
    sem_wait(&sem_irak);
```

```
    n_irakurle = n_irakurle - 1;
```

```
    if (n_irakurle == 0)
```

```
        sem_post(&mutex);
```

```
    sem_post(&sem_irak);
```

```
    pthread_exit(0);
```

```
}
```

```
/* Idazlearen kodea */
```

```
void Idazle(void) {
```

```
    sem_wait(&mutex);
```

```
    datua = datua + 2; /* Edukia aldatu */
```

```
    sem_post(&mutex);
```

```
    pthread_exit(0);
```

```
}
```

Zenbat itxarongo du idazleak idazteko baimena izan arte?

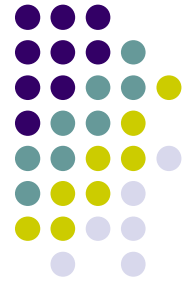
EZ DAKIGU - Irakurle kopuruaren araberakoa!

Nola aldatu beharko litzateke kodea idazleak lehentasuna izan dezan?

Ettxerako-lan-edo moduan eman da. Ez dut ideiarik ez xD
Zeozter egiteko kapaz izan banaiz repo bat egongo da.

3. PROZESUAK

3.4 KOMUNIKAZIO eta SINKRONIZAZIO MEKANISMOAK

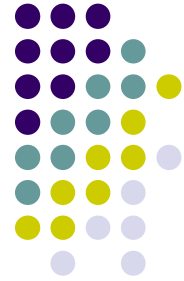


- Mezuen transferentzia:
 - Komunikazio eta sinkronizazio mekanismoa. Abantailak:
 - Elkarren arteko baztertzea
 - Mezua bidali eta mezua jaso egiten duten prozesuak sinkronizatu
 - Memoria-espazio desberdinen artean komunikatu (konputagailu berean edo desberdinen artean)
 - Oinarrizko primitiboak:
 - **send(helburura, mezua)** mezu bat igorri helburu prozesuari
 - **receive(jatorritik, mezua)** jaso mezu bat jatorri prozesutik

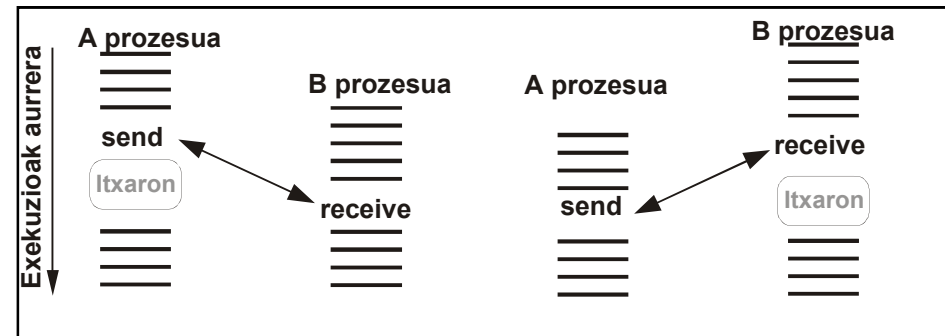
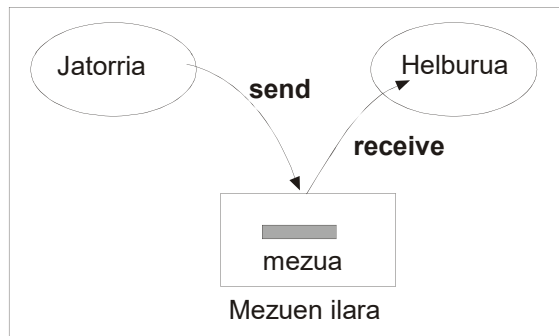
JATORRIA ESPEZIFIKATU BEHAR DA! KONTUZI!

3. PROZESUAK

3.4 KOMUNIKAZIO eta SINKRONIZAZIO MEKANISMOAK

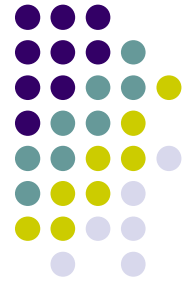


- Mezen transferentzia:
 - Diseinuan kontuan hartu beharrekoak:
 - Mezuaren tamaina, formatuak...
 - Datu-fluxuaren noranzkoa (noranzko bakarrekoa, noranzko bikoa)
 - Izendapena
 - Zuzena (pid)
 - Ez-zuzena, mekanismo baten bidez (ataka, ilara)
 - Sinkronizazioa (sinkronoa, asinkronoa)
 - Bidaltze eta jasotzea blokeatzailea (sinkronoa). **Hitzordua** edo **Rendez vous**.
 - Bidaltzea ez blokeatzailea, jasotzea blokeatzailea (asinkronoa).
 - Bidaltze eta jasotzea ez-blokeatzailea (asinkronoa). Beharrezkoa jakitea mezua noiz heldu den.
 - Tarteko biltegitratzea ala ez



3. PROZESUAK

3.4 KOMUNIKAZIO eta SINKRONIZAZIO MEKANISMOAK



- POXISeko Mezu-ilarak:

- “name” mezu-ilara sortu “attr” atributuekin (mezu-kopuru maximoa, mezuen tamaina, blokeatzailea/ez blokeatzailea...)

Fitxategi baten formatua izango du.

O_CREAT | O_EXCL | O_RDWR, O_WRONLY, O_RDONLY

- `mqd_t mq_open(char *name, int flag, mode_t mode, mq_attr *attr);`

SORTZEAN SOILIK

- Ilara itxi eta ezabatu

- `int mq_close (mqd_t mqdes);`
- `int mq_unlink (char *name);`

- “mqdes” ilara mezuak bidali eta ilaratik “len” luzerako “msg” mezuak jaso:

- `int mq_send(mqd_t mqdes, char *msg, size_t len, int prio);`
- `int mq_receive(mqd_t mqdes, char *msg, size_t len, int prio);`

Defektuz NULL izango dugu -> Prioritateak aldatzeko INT-ak eman

- Laborategian System V Mezu-ilarak erabiliko ditugu.

3. PROZESUAK

3.5 PLANIFIKAZIOA



- **Planifikatzailea, Antolatzailea** edo **Scheduler** : Lanerako prest dauden prozesuak zein ordenetan exekutatu erabakitzen duen SE-aren modulua da. Helburua PUZ-aren kudeaketa.
- Betebeharrak:
 - Inpartzialtasuna: prozesu bakoitzak PUZ denbora zuzena izan dezala ziurtatu
 - Errendimendua: PUZa denboraren % 100 lanean edukitzea
 - Erantzun-denbora: erabiltzaile **interaktibo**ei erantzun-denbora **minimizatu**
 - Itxarote-denbora: **batch** erabiltzaileek emaitza lortzeko denbora **minimizatu**
 - Throughput: Orduko prozesatutako lanen kopurua maximizatu

KONTRAESANA



3. PROZESUAK

3.5 PLANIFIKAZIOA

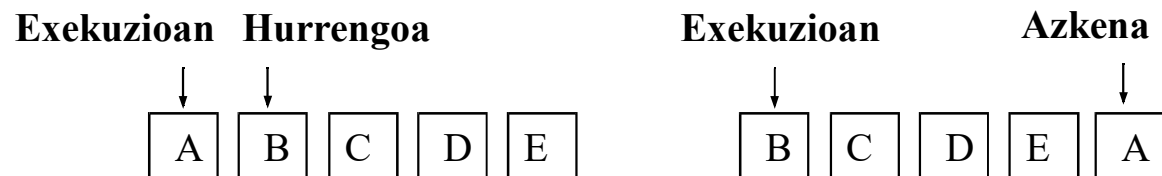
- Planifikazio-motak:
 - Kanporatze gabeak: prozesua PUZan behar duen artean.
 - Kanporatzeaz: SE-ak PUZa kentzen die prozesuei
 - Erlojuak etendura periodikoak sortu behar ditu horretarako
 - **Prozesu-motak:**
 - S/l asko eta prozesamendu gutxi → Azkar konmutatu
 - S/l gutxi eta prozesamendu asko → Astiro konmutatu
 - Planifikazio-polika desberdinak daude:
 - Planifikatzaileak politika inplementatzen du: prozesua hautatu
 - Aktibatzaileak (*dispatcher*) testuinguru aldaketaz arduratu.

3. PROZESUAK

3.5 PLANIFIKAZIOA

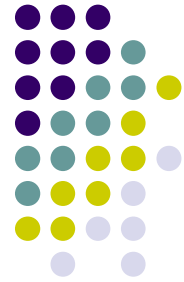


- **Round Robin** edo **Txandaketa**:
 - Prest dauden prozesu guztiek lehentasun berdina daukate. Ilara zirkularrean antolatzen dira txandak.
 - Prozesu guztiek exekuzio denbora finko bera dute → **Kuantu bana**
 - Prozesuek ez dute kuantu osoa zertan agortu
 - **Kuantuaren luzera**:
 - laburra → testuinguru aldaketa asko, PUZaren errendimendua ↓
 - luzea → ingurune interaktiboetan erantzun pobrea
 - Oreka



3. PROZESUAK

3.5 PLANIFIKAZIOA



● Lehenetasunak:

- Estatikoak:
 - Tarifa edo boterearen arabera.
 - Inplementazio erreza, gainzama gutxi
 - Maila baxua duten prozesuetan gosete arriskua
- Dinamikoak:
 - $1/f$ algoritmoa, S/I asko duten prozesuen alde egiteko
 - 100msko kuantu batetik 2ms Lehenetasuna 50.
 - 100msko kuantu batetik 25ms Lehenetasuna 4.
 - Sentikorra ingurune aldaketekiko, baina gainzama handiagoa
- Lehenetasun-klaseak: Klase bakoitzaren artean txandak

Zahartze sistema bat egin!
Lehenetasuna txikiagotu denbora pasa ahala

3. PROZESUAK

3.5 PLANIFIKAZIOA



- **Ilara desberdinak.** Adibide bat:
 - CTSS sisteman (denbora partitua) prozesu bat baino ezin zen egon memorian → swapping memoria eta disko gogorraren artean
 - Helburua: Testuinguru aldaketa kopurua gutxitu → kuantu handiak
 - Klase goreneko prozesuek → 1 kuantu
 - Hurrengo klase handiko prozesuek → 2 kuantu
 - Hurrengo klaseko prozesuek → 4 kuantu
 - n. klaseko prozesuek → 2^n kuantu
 - Baldintza: Prozesu batek kuantu kopurua agortzen duenean beheko klasera pasatuko da

• Ilaren arteko planifikazio algoritmoa definitu beharra: - Lehentasun handieneko ilarako prozesuak bukatu arte ez da beheko klaseko ilara

3. PROZESUAK

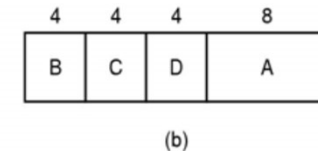
3.5 PLANIFIKAZIOA



- **Lehenbizi lanik laburrena SJF:**

- Batch sistemetan erantzun-denbora laburtzeko, prozesuen exekuzio denbora ezaguna delako.
- Lau lanen batezbesteko turnaround: $(4a + 3b + 2c + d) / 4$

- (a) $(32+12+8+4)/4=14$
- (b) $(16+12+8+8)/4=11$



- Sistema interaktiboetarako iraupenaren estimazioa egin daiteke, zahartzapenean oinarrituz:
 - $aT(0) + (1-a)T(1)$
 - $T(0)$ aurreko estimazioa
 - $T(1)$ azken exekuzioaren iraupena
 - $a=1/2$ izanik
 - $T_0, T_0/2+T_1/2, T_0/4+T_1/4+T_2/2, T_0/8+T_1/8+T_2/4+T_3/2$

3. PROZESUAK

3.5 PLANIFIKAZIOA



- **Loteria** planifikazioa edo **Zorizkoa** :
 - Prosesuen artean txantelak banatu eta PUZ txandak zotzetara egin
 - Txantel gehiago → PUZ aukera gehiago
 - Prosesuen lankidetza eskaintzen du: blokeatuta gelditu den prozesuak iratzartu eraziko duenari ematen dizkio txantelak.
- Adibidea: Bideostream zerbitzari batean:
 - 25 frame/seg. → %25 txantel
 - 20 frame/seg. → %20 txantel

3. PROZESUAK

3.5 PLANIFIKAZIOA



- **Denbora errealeko** planifikazioa:
 - Deadline edo epemugaren arabera:
 - Soft R.T. zenbait estimulu galtzeak ez dauka ondorio larririk
 - Hard R.T. estimulu guztiei erantzun behar zaie
 - Denbora errealeko sistema planifikagarria izateko:
 - Sistemak m gertaera periodikori erantzun behar badie, erantzunaren periodoa eta PUZ denbora P_i eta C_i izanik:

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq 1$$

- Adibidea: hiru gertaera P_i 100, 200, 500ms C_i 50, 30, 100ms
- $0,5 + 0,15 + 0,2 \rightarrow$ Planifikagarria
- Gertaera berri bat segundoro $\rightarrow C_i < 150\text{ms}$



3. PROZESUAK

3.5 PLANIFIKAZIOA

- **Denbora errealeko hiru algoritmo:**

- **Rate monotonic edo tasa monotonoa:**

- Lehenasuna periodoaren alderantziz proportzionala:

- P_i 100 → Lehenasuna 10
- P_i 20 → Lehenasuna 50

- **Earliest deadline first edo lehenago larriena:**

Lehenasunaren arabera

- Gertaera bat ematen denean → Prozesu bat prest egoerara
- Prest dauden prozesuen artean “larriena” edo muga hurren duena exekutatu.

- **Least laxity edo nasaitasun txikienekoa:**

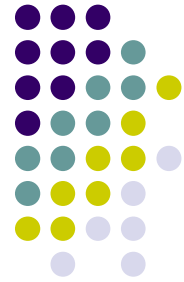
- P_i 250ms eta C_i 200 ms → Nasaitasuna 50ms

- Denbora errealeko S.E.ak bete beharreko baldintzak:

- Atal txikitan banatuta egotea, exekuzio azkarrekoak
- Testuinguru aldaketa azkarrak (Hobe memoria birtualik gabe!)
- Etendurak oso denbora laburrez desgaitu
- Timerrak oso txikiak: ms edo μ s.

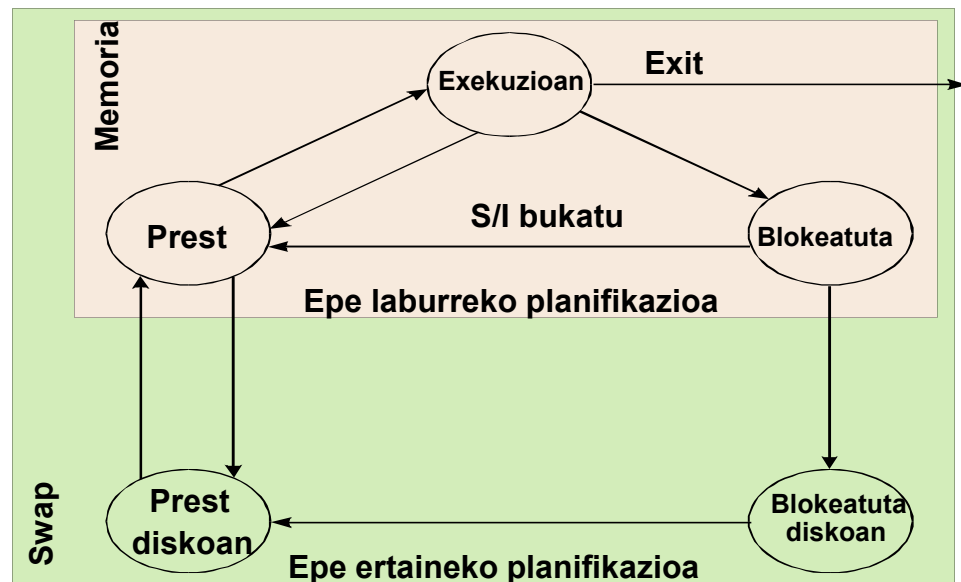
3. PROZESUAK

3.5 PLANIFIKAZIOA



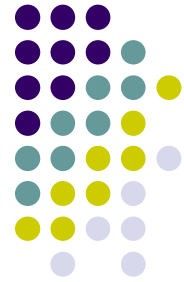
- **Epe ertain eta epe laburreko** planifikazioa, **maila biko** planifikazioa:
 - Memorian prozesu bakan batzuk bakarrik gordetzeko lekua dagoenean, swapping edo truke egin behar izaten da memoria eta diskoaren artean → gainzama handia
 - Epe laburrekoak → RAMEko prozesuen artean
 - Ertainekoak → RAM eta swap arteko txanda

- Zenbat denbora RAMEan
- Zenbat PUZ denbora azken txandan
- Prozesua norainoko haundia den (txikiak ez hartu kontutan *erabakitze*)
- Zein den prozesuaren lehentasuna



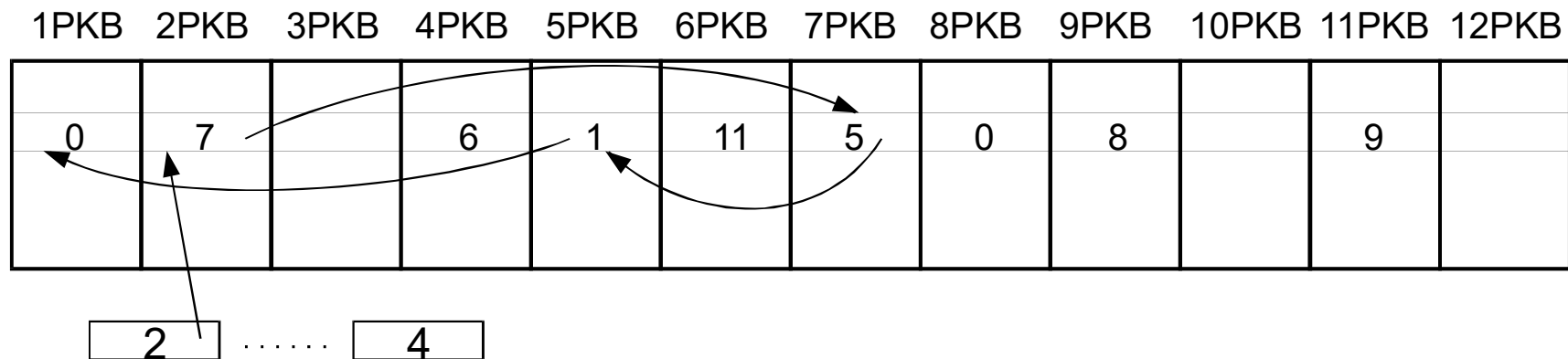
3. PROZESUAK

3.5 PLANIFIKAZIOA



- Lehen tasun mailen inplementazioa:
 - SE-ak ilara desberdinak antolatzen ditu.
 - Zerrendaburu desberdinak
 - PKB-etako erakusleak erabiltzen dira zerrendak antolatzeko
 - Atzipena eraginkorra da, nukleoan

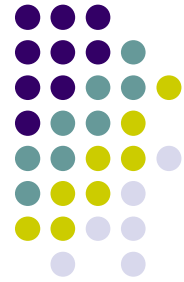
Prozesuen Taula



Ilara desberdinen erakusleak (Kernelean)

3. PROZESUAK

3.6 ITXARONALDI PASIBOAREN IMPLEMENTAZIOA



- Itxaronaldi aktiboa:

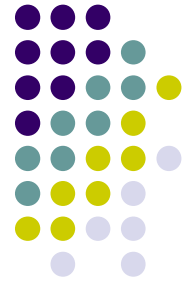
```
wait(s) {  
    s = s - 1;  
    while (s < 0);           /*PUZa erabiliz*/  
}  
signal(s) {  
    s = s + 1;  
}
```

- Itxaronaldi pasiboa

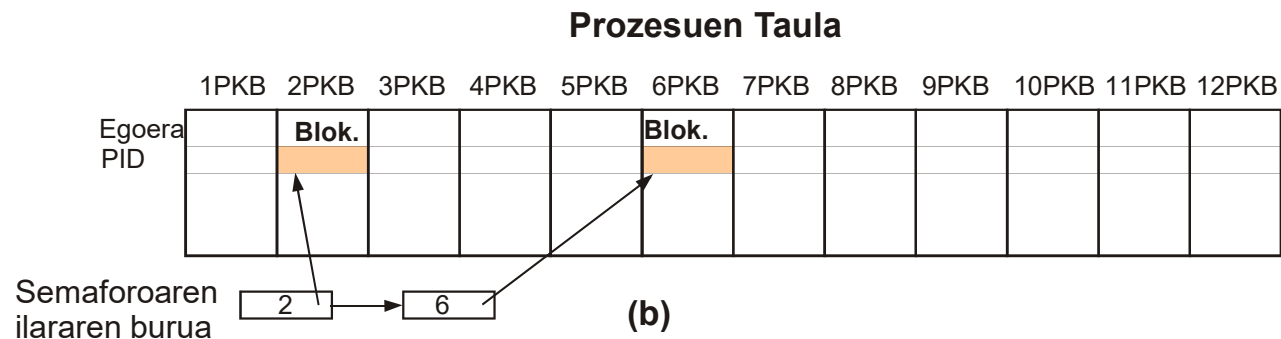
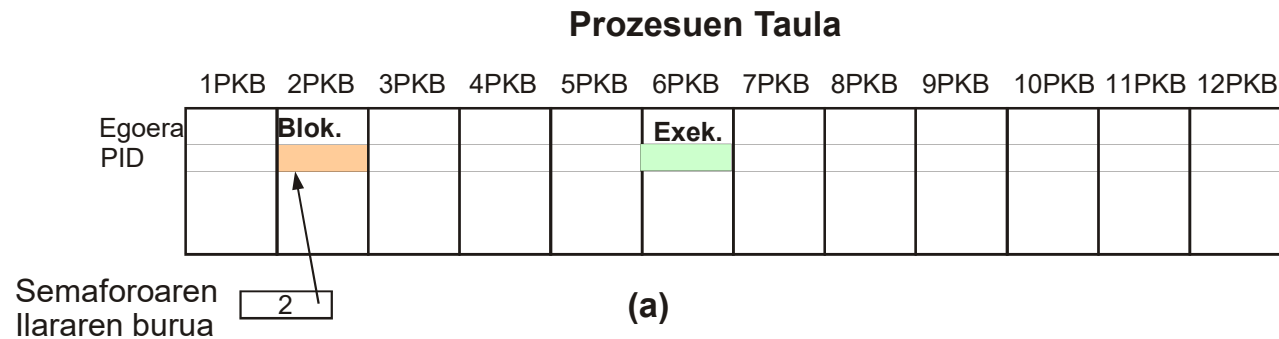
```
wait(s) {  
    s = s - 1;  
    if (s < 0)  
        Prozesua blokeatu;           /* PUZa libre */  
}  
signal(s) {  
    s = s + 1;  
    if (s <= 0)  
        wait batean blokeatutako prozesu bat desblokeatu;  
}
```

3. PROZESUAK

3.6 ITXARONALDI PASIBOAREN INPLEMENTAZIOA

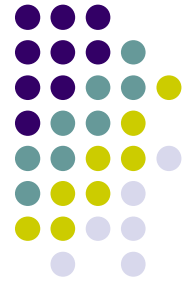


- Prozesu bat semaforo batean blokeatu
 - Adibidea: 6PKB-ko prozesuak wait(s) egin du

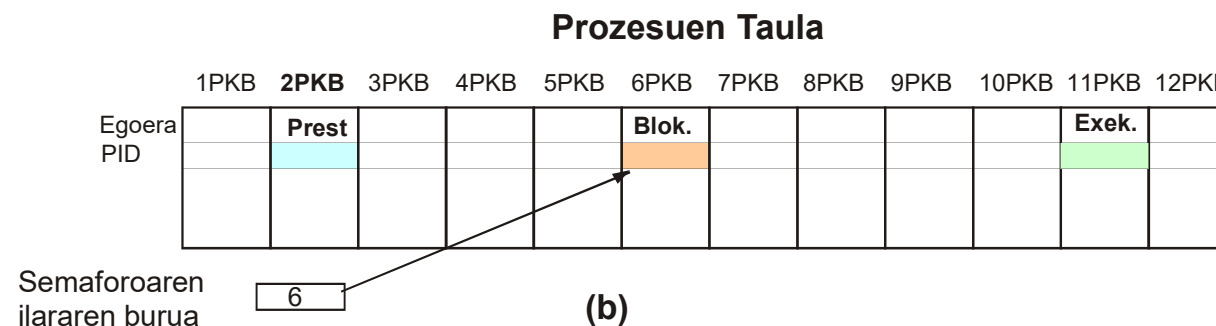
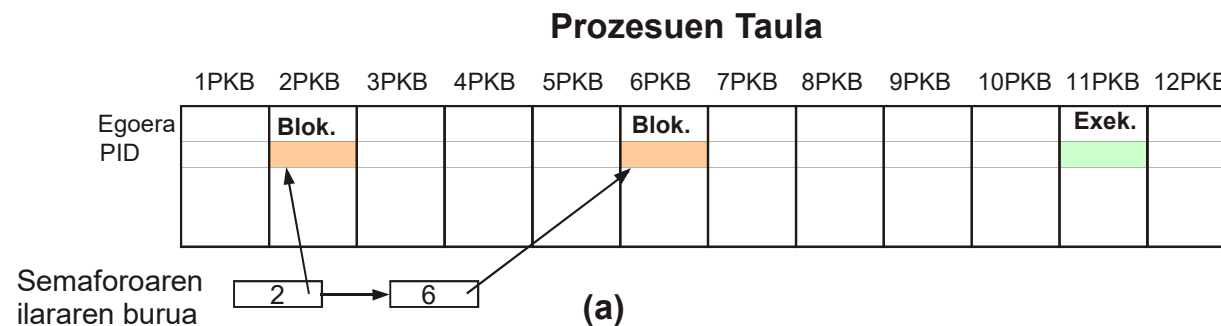


3. PROZESUAK

3.6 ITXARONALDI PASIBOAREN INPLEMENTAZIOA

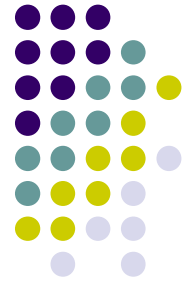


- Semaforoan blokeatutako prozesu bat desblokeatu:
 - Adibidea: 11PKB-ko prozesuak signal(s) egin du



3. PROZESUAK

3.6 ITXARONALDI PASIBOAREN INPLEMENTAZIOA



- *TSL (Test and Set Lock)* makina-agindu atomikoa.
 - Multzoa atomikoki exekutatu bitartean memoria-busa blokeatuta dago

```
int test-and-set(int *balioa) { /* TSL aginduaren simulazioa*/
    int temp;

    temp = *balioa;
    *balioa = 1;                /* true */
    return temp;
}
```

- Atal kritikoa TSL erabiliz:
 - Prozesuek `lock` aldagaia partekatzen dute (hasierako balioa false)

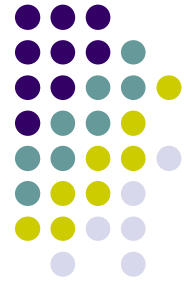
```
lock = false
```

```
while (test-and-set(&lock));
<Atal kritikoaren kodea>
lock = false;
```

```
while (test-and-set(&lock));
<Atal kritikoaren kodea>
lock = false;
```


3. PROZESUAK

3.6 ITXARONALDI PASIBOAREN IMPLEMENTAZIOA



```
wait(s) {
```

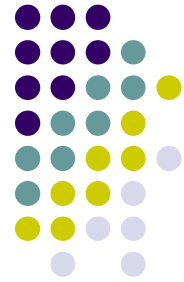
```
    while (test-and-set(&s_lock));
    s = s - 1;
    if (s < 0){
        s_lock = false;
        Prozesua blokeatu;
    }
    else
        s_lock = false;
}
```

```
signal(s) {
```

```
    while (test-and-set(&s_lock));
    s = s + 1;
    if ( s <= 0){
        wait egiten blokeatutako prozesu
        bat desblokeatu;
    }
    s_lock = false;
}
```

3. PROZESUAK

3.6 ITXARONALDI PASIBOAREN INPLEMENTAZIOA



- PKB-en egituraren adibidea (MINIX 1.0 Prozesuen Taula):

0757	EXTERN struct proc {		
0758	int p_reg[NR_REGS];	/* process' registers */	
0759	int *p_sp;	/* stack pointer */	
0760	struct pc_psw p_pcpsw;	/* pc and psw as pushed by interrupt */	
0761	int p_flags;	/* P_SLOT_FREE, SENDING, RECEIVING, etc.*/	
0762	struct mem_map p_map[NR_SEGS];	/* memory map */	
0763	int *p_splimit;	/* lowest legal stack value */	Prozesadorearen
0764	int p_pid;	/* process id passed in from MM */	egoera
0765			
0766	real_time user_time;	/* user time in ticks */	Denboraren
0767	real_time sys_time;	/* sys time in ticks */	kudeaketa
0768	real_time child_utime;	/* cumulative user time of children */	
0769	real_time child_stime;	/* cumulative sys time of children */	
0770	real_time p_alarm;	/* time of next alarm in ticks, or 0 */	
0771			
0772	struct proc *p_callerq;	/* head of list of procs wishing to send */	Mezuen
0773	struct proc *p_sendlink;	/* link to next proc wishing to send */	ilarak
0774	message *p_messbuf;	/* pointer to message buffer */	
0775	int p_getfrom;	/* from whom does process want to receive? */	
0776			
0777	struct proc *p_nextready;	/* pointer to next ready process */	Plangintzako
0778	int p_pending;	/* bit map for pending signals 1-16 */	ilarak
0779	} proc[NR_TASKS+NR_PROCS];		