3. Partzialeko Apunteak

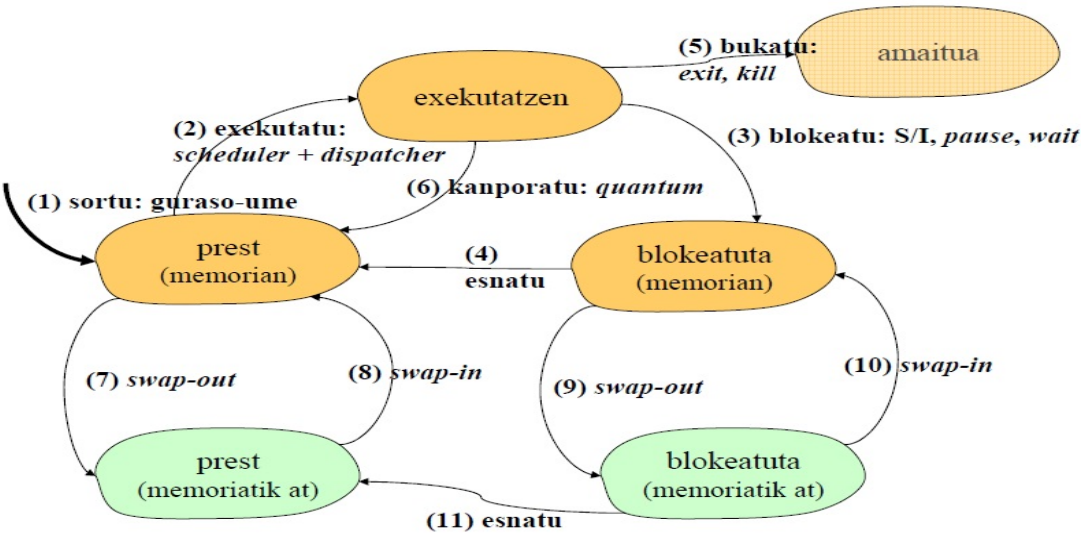
Prozesuak eta Egoerak

Programa bat exekutatzen ari garen bitartean hainbat egoera desberdin igaro dezake:

- ***Ranking:*** martxan egotea.
- ***Sleeping:*** lo edo zain egon daiteke.
- ***Stuck:*** Trabatuta.
- ***Zombie:*** prozesua zombi egotea.
- ***Stopped:*** gelditua egotea prozesua.

Exekutatzen ari diren programa guztiak ezin dira beti memorian egon, ondorioz, hainbat eragiketa daude arazo hau konpontzeko, prozesuen gaineko eragiketak:

1. ***Prozesua sortu:*** fork() erabiltzen da honetarako: funtzioak bueltatutako zenbakia identifikadorea da, ondoren, programa kargatzen da, eta testuingurua ezarri.
2. ***Exekutatu:*** Exekutatzen hari den bitartean *quantum*-aren bidez prozesua kanporatzen da. Honek prest egoeran utziko du.
   - ***SCHEDULER***-ak hurrengo exekutatuko den prozesua zein den erabakitzen du, hainbat politika daude erabaki hori hartzeko.
   - ***DISPATCHER***-ak exekutatu nahi den programa exekutatuko du.
3. ***Blokeatu:*** Sistema dei baten bidez, prozesua blokeatzen da. Blokeatutako prozesuaren testuingurua gordetzen da eta beste programa bat exekutatzen hasten da.
4. ***Esnatu:*** SE-ren errutina asinkrono bat erabiliz lortzen da eragiketa hau egitea. Aurretik blokeatutako prozesua prest-egoeran jarri daiteke.
5. ***Bukatu:*** exit() sistema-deia erabiltzen da honetarako, eta honen bidez, prozesua erabiltzen ari zen baliabideak askatzen dira, prozesua amaituta egoerara igaroz eta bukaera-kodea gordez gurasoarentzat.



OHARRA: Grafoan azaldu ez arren, posiblea da guraso-prozesua bukatzea ume-prozesua bukatu aurretik. Kasu horretan, umea umezurtz geratuko da eta init prozesuak adoptatuko du.

Prozesuak

Prozesuak independenteak edo haien artean dependenteak izan daitezke, hainbat desberdintasun daude, bi prozesu hauen artean:

- ***Prozesu independenteak:*** Kasu honetan, prozesu baten exekuzio desberdinetan, datu berdinak sartu ezkero, emaitza berdina izango da. Gainera, prozesu baten exekuzioak ez du besteengan eraginik.
- ***Prozesu ez-independenteak:*** Aurreko kasuan ez bezala, prozesu desberdinetan, nahiz eta datu berdinak sartu, emaitza desberdinak lortu genezake. Emaitza prozesuen exekuzioaren ordenaren menpe baitago. Bi eredu mota desberdin bereizten dira:
  - ***Ekoizle-kontsumitzaile eredua:*** ekoizleak tamaina finkoko buffer batean elementuak uzten ditu eta kontsumitzaileak bertatik hartzen ditu.
  - ***Bezero-Zerbitzari ereduan:*** elementuak uzten dira, eta periodikoki, beste prozesu batek ea elementurik ba al dagoen konprobatzen du.

Komunikazioa eta sinkronizazioa

Komunikazioa prozesuen arteko informazio trukaketa da. Sinkronizazioa, ordea, bi modutara egin daiteke: Esplizituki, hau da, funtzioen bidez, edo Inplizituki, komunikazio-mekanismoen bidez.

Sekzio-kritikoa: Hainbat prozesuek atzitzen duten kode-zatiari deritzo. Adibidez, hainbat prozesadoreek batera loop bat aztertzen badute, bakoitzak loop-aren zati bat hartuz. Garrantzitsua da i++, egiterakoan, i horren balioa, guztientzat desberdina da, ondorioz, kontu handiarekin ibili beharra dago. Sekzio mota honetan, ziurtatu behar da uneoro prozesadore bakarra egon daitekeela exekutatzen.

FIFO fitxategi bat Pipe bat bezalakoa da, baina modu desberdinean sortzen dira. FIFO-a ez da komunikaziorako erabiltzen den kanal anonimo bat, izenduna da, eta funtzio baten bidez sortutakoa . Edozein fitxategi bezala, irakurri eta idatzi daiteke, baina hori egiteko bi aldeetatik ireki behar da lehenik eta operazioa bukatu ondoren. Gainera, beste prozesuren bat idazten edo irakurtzen ari bada, itxaron egin beharko du, honek bukatu arte. Metodo hauei ***Mezu-Trukearen bidezko komunikazioa*** esaten zaie.

- ***Buzoiak, FIFO (First In First Out):*** Hainbat funtzioren bidez sortu daiteke buzoi mota hau, gehien erabiltzen direnak, hauek dira:
  - ***int mknod():*** Funtzio honek path-ak emandako izenarekin fitxategi bat sortuko du, berari bideratuta, pointer baten bidez.

Int **mknod(** char ***path**, int **modua**, int **disp)**

FIFO bat sortu nahi bada erabilera ondorengoa da:

Fd = mfnod(str, S_IFIFO|0666, 0);

- ***int mkfifo():*** FIFO izeneko fitxategi berezia sortzen du path-ean jarritako izenarekin. “modua” zatian fitxategi honen baimenak alda daitezke chmod() erabiltzen genuen bezala.

Int **mkfifo** (char ***path**, int **modua**)

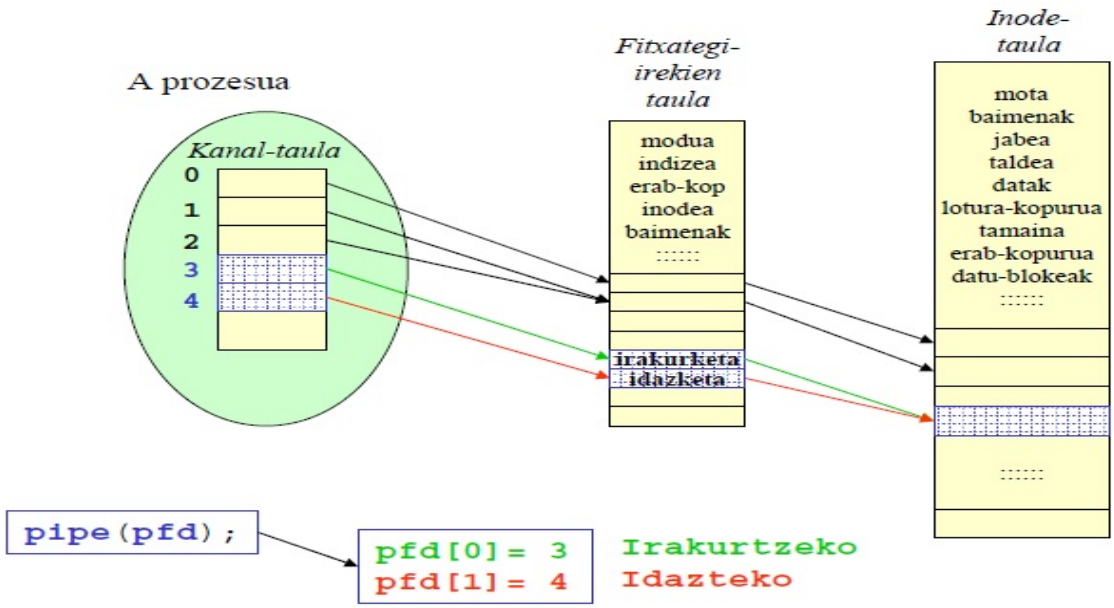
FIFO bat sortu nahi bada, erabilera ondorengoa da:

Fd = mkfifo(str, 0666)

- ***Izenik gabeko buzoiak, Pipeak:***
  - **int pipe():** Funtzio honek pipe bat sortzen du, fitxategi bat non bertan adierazitako kanalak birbideratuko dira fitxategi horretara (edo horietara). Normalean, behean adierazitako pfd hori 1 luzerako zenbaki-katea da, non pfd[0] irakurketak egiteko eta pfd[1] idazketak egiteko erabiltzen diren. Behin eragiketa bat egindakoan, pipe-aren informazioa gaurkotzen da.

int **pipe**(int ***pfd**)

OHARRA: pipe bat noranzko batekoa da bakarrik, ondorioz, bi prozesuen arteko pipe batean, pipe-a sortu duen prozesua bestearekin sinkroniza daiteke, baina ez alderantziz, betiere berak ez badu beste pipe bat sortzen fitxategi berari begiratzen.



Kasu honetan, ikusi daitekeen bezala, 3 eta 4 kanalek pipe bateri apuntatzen diote.

Prozesuen kontrolerako sistema-deiak

Prozesuen identifikazioa

- ***getpid():*** Prozesuaren identifikadorea bueltatzen du.

  int ***getpid()***
- ***getppid():*** Prozesuaren, prozesu gurasoaren identifikadorea bueltatzen du.

  int ***getppid()***
- ***getuid():*** Prozesuaren jabea den erabiltzailearen identifikadorea (uid) bueltatzen du.

  int ***getuid()***

Prozesuen sorrerarako

- ***fork():*** Sistema-dei honek prozesu berri bat sortzen du, non prozesu-umea deritzon. Honek bere exekuzioa fork-aren hurrengo agindutik hasiko du. KONTUZ! Umeak dena heredatzen du gurasotik, baina bere pid-a desberdina da. Fork-ek umeari 0 pid-a ematen dio, eta gurasoari umearen pid-a.

  int ***fork()***
- ***execlp():*** Argumentu bat edo gehiago eta fitxategi bat sarturik, argumentu horiek exekutatuko dit. Bukaeran beti NULL balioa jarri behar zaio. Oso baliagarria gertatzen da komandoak exekutatzeko.

  execlp (char *file, char *arg0, … , NULL***)***
- ***execvp():*** Oso baliagarria gertatzen da beste programa bati deitzeko eta honi argumentuak sartzeko.

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

A eta B prozesuak izanda zein hartuko luke lehenengo

Demagun adibidez bi prozesu daudela sistema eragilearen *Prest* ilaran CPU-a libre dagoen une batean. Demagun A dela ilarako lehenengoa eta amaitzeko CPU-aren segundo bat beharko duela. B prozesuak bestalde CPU-aren 0,01 segundo bakarrik behar ditu. Sistema eragileak bi prozesu hauen exekuzio-ordena antolatu beharko du. Bi prozesu hauetatik zein aukeratu beharko luke *Exekutatzen* egoerara pasa eta CPU-a betetzeko? Sistema eragileak irizpideren bat aplikatu beharko du erabakia hartu ahal izateko.

CPU-aren ikuspuntutik, prest dauden prozesua guztien artetik batek lehentasuna izango du. Lehentasuna izango duen prozesua, erabiltzen den politikaren araberakoa izango da. Hainbat politika daude eta guztiak dute arazoren bat, ondorioz, etengabe ari dira politikak berriak sortzen.

- ***FIFO politika:*** Firts in First out politika da, hau da, lehen sartzen dena irtengo da lehen, ondorioz, zain denbora gehien dagoen prozesua aukeratuko da.
- ***Starling politka:*** Denbora gutxien behar duena izango da lehenengoa exekutatzen. Arazoa: beti egon daiteke denbora gutxiago behar izango duen prozesu bat, eta horrela, handiak ezingo dira inoiz exekutatu.

Nola onartzen duen Linuxek exekuzioa konkurrentea

Linuxen gai gara prozesu berriak sortzeko, komandoak edo gure programak exekutatzeko, era sekuentzialean eta baita konkurrentean.

1. Programak prozesuei buruzko zer sistema-dei ditu? Zer egiten du bakoitzak?
   Fork() eta getpid() sistema-deiak erabiltzen dira.

   fork(): prozesu-ume berri bat sortzen du, non honen ondorengo agindutik hasiko den exekutatzen. Prozesu-ume honek bere prozesu-gurasoaren informazio eta ezaugarri guztia heredatuko du, baina kasu honetan pid berri batekin.
   getpid(): prozesuaren identifikadorea itzuliko du.

2. Zenbat prozesu egongo dira martxan programa honetan?
   Bi prozesuk parte hartuko dute. Kasu honetan, 3227 eta 3228.

3. Prozesu bakoitzak ze kode exekutatuko du?
   Gurasoak fork-aren aurreko kode guztia exekutatuko du.
   Case 0: prozesu umeak soilik exekutatuko du.
   Default: gurasoak exekutatuko du.
   Azken zatia biek exekutatuko dute.

4. Zein prozesuri dagokio marka aldagaia?
   Prozesu bakoitzak bere marka aldagaia dauka, H marka 3228 prozesuari dagokio eta P 3227 prozesuari.

5. Marka aldagaiari esleitutako balioak, exekuzioaren araberakoak dira(hots, sartu dituzuen atzerapenen araberakoak) edo beti berdinak?
   Bai, prozesu bakoitzak bere marka aldagai propioa dauka, ondorioz, prozesu bakoitzak egiten duena, bere marka aldagaiari eragingo dio.

6. Zer informazio du prozesu bakoitzaren kanal-taulak A, B eta C-n?
   A: 2 prozesu dira, ondorioz 4 kanal prozesu bakoitzean.
   B: Gurasoa: 4 kanal.
   Semea: 4 kanal edo 3, gurasoa edo semea heldu lehenago Close-ra.
   C: Gurasoa: 2 kanal
   Semeak ez du idazten.

7. Zer aldatuko da irteeran close(fd); sententzia C puntuaren justu aurretik jartzen bada?

8. Irekitako fitxategia adierazten duen idazketa-erakuslea, pribatua da prozesu bakoitzean edo bien arten partekatzen da?

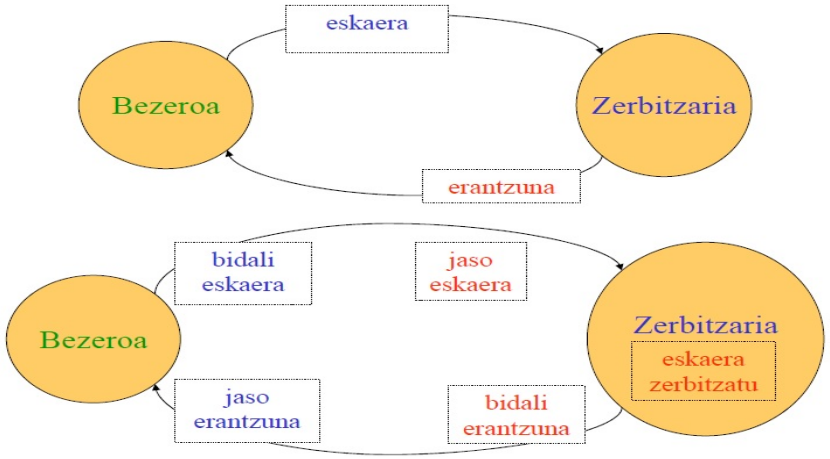
Irekita dauden fitxategien taulari dagokio.

a Baldin eta Open prozesu-umea sortu aurretik egiten bada eta irekita dauden fitxategien taula heredatzen badu, sarrera berdinera apuntatuko dute biek.

b Open ondoren egin bada 2 puntero izango dira, non bakoitzak bere sarrerara eramango gaituzte.

Bezero-Zerbitzari eredua

Eredu honetan, bezeroak hasten du komunikazioa. Horretarako, zerbitzariari eskari bat egiten dio (informazioren bat, datuak...). Honek, eskaera horri erantzunez, betiere ahal izan ezkero, bezero-prozesuari eskatutakoa emango dio. Ondorengo eskeman ikusi daiteke bezero-zerbitzari eredua.

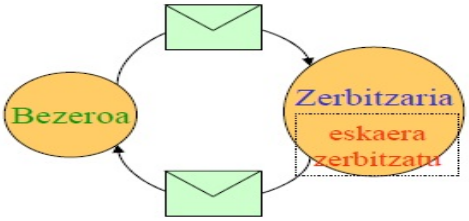


Bi motakoak izan daitezke bezeroaren eskaera, itxaroteko denboraren araberakoak biak ere:

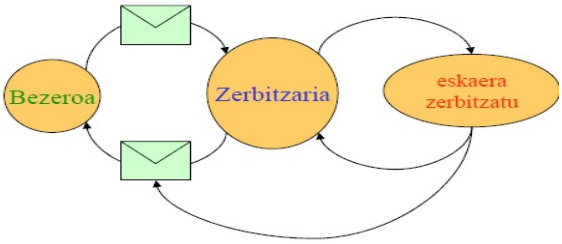
- ***Sinkronoak:*** Zerbitzariaren erantzunaren zain egoten da. Ondorioz, erantzuna heldu bitartean ez da beste ezer egingo.
- ***Asinkronoak:*** Eskaera bidali ondoren, beste prozesu batzuk exekutatzen jarraitu dezake, ez baitago zerbitzariaren erantzunaren zain.

Zerbitzariaren erantzunen kudeaketa ere bi motakoa izan daiteke.

- ***Elkarreragilea:*** Aldiko eskaera bat zerbitzatzen da, ondorioz, beste eskaera bat heldu ezkero zain egon beharko da eta ez da zerbitzatuko aurreko eskaerarekin bukatu arte.



- ***Konkurrentea:*** Zerbitzariak eskaera bat baino gehiago zerbitzatu dezake aldiro, eskaera bakoitza beste prozesu bati pasatzen baitio.



Nola kontrolatzen ditu prozesuak Linuxek, denak pozik egon daitezen?

Sistema batean aldi berean exekutatzen diren prozesuak ikusi ahal izateko ***top(1)*** komandoa erabiltzen da. Komando honek erabiltzen ari garen sistemaren prozesuak erakusten ditu denbora errealean, segunduro eguneratzen delarik. Hainbat zutabe agertzen dira, honako informazioa ematen dutelarik.

PID: Prozesuaren identifikadorea.
USER: Erabiltzailearen izena.
PR: Prozesuaren lehentasuna.
NI: NICE balorea, zenbat eta txikiago edo negatiboagoa, lehentasun handiagoa
VIRT: Prozesu bakoitzak erabilitako memoria birtual kopurua, Kbetan.
RES: Alteratu gabeko memoria fisikoa: RES = CODE + DATA.
SHR: Prozesuak erabiltzen duen partekatutako memoria kopurua.
%CPU: Prozesuak erabili duten CPU-denboraren ehunekoa.
%MEM: Prozesuek erabili duten memoria diskoaren ehunekoa.
TIME+: Prozesua hasi denetik erabilitako CPPU-denbora totala.
COMMAND: Prozesua exekutatzen ari den komando edo programaren izena.

SCHEDULER-aren lehentasun-algoritmoa:

Scheduler-a, prozesuak exekutatuko diren ordena adierazten duen, sistemak hainbat irizpideen arabera ordena ditzake exekutatzeko zain dauden prozesu horiek, orain artean honako irizpide hauek ditugu.

CPU-denbora beste prozesu bati ezin diote kendu:

- ***First-Come First-Serve (FCFS):*** Programatzeko oso sinplea da FIFO baten erabiliz inplementatzen baita, ailegatu bezala exekutatzen dira prozesuak. Horrek, bere abantailak dauzka, sinplea da, eta programatzeko oso erraza. Baina ez da eraginkorra ez baita batere egokia prozesuak exekutatzerakoan.
- ***Shortsest Job First (SJF):*** CPU-denbora gutxien behar duten prozesuak exekutatzen dira, eta ondoren, denbora gehien behar dutenak. 2 prozesuk denbora berdina behar badute exekutatzen, DCDS aplikatzen da prozesu horiekin. Aurrekoa baino optimoagoa da, baina arazo handi bat izan dezake. Beti egon daitezke denbora gutxiago behar dituzten prozesuak, eta horrela, denbora gehiago behar dituztenak ez lirateke inoiz sartuko CPU-ra.

CPU-denbora kendu diezaieke beste prozesuei:

Mota honetako algoritmoak denbora-errealeko prozesu batean aplika daitezke.

- ***Round Robin:*** Algoritmo sinple eta justua da, non prozesu bati exekuzio-denbora maximo bat aplikatzen zaio, quantum-a. Prozesuari bere quantum-a bukatzen bazaio eta ez badu oraindik bukatu, prozesu-kolaren bukaeran jarriko da bere exekuzioa bukatzeko eta beste prozesu bat sartuko da exekutatzeko.

Prozesuak asignatzen lehentasunaren arabera:

Prozesu bati, CPU-an exekutatzeko jarri aurretik, scheduler-ak zenbaki bat ezartzen dio lehen aipatutako algoritmoen arabera, eta normalean, zenbaki txikienak izaten du lehentasuna.
Zenbaki hauek ezartzeko orduan bi modutara egin daitezke lehentasunaren arabera:

- ***Lehentasun estatikoa:*** Lehentasuna ez da aldatzen prozesua existitzen den bitartean. Honen bidez ez da gainkargarik sortzen eta programatzeko erraza da.
- ***Lehentasun dinamikoa:*** Prozesua bukatu gabe dagoen bitartean, bere lehentasuna alda daiteke denboran zehar, denbora asko zain dauden prozesuetan lehentasuna handiagoa izango da. Programatzea zailagoa da eta gainkargaren bat sor daiteke.

Jarduera partekatzearen arazoak

Fitxategi bat partekatzen dugun bitartean, arazoak sortzen dira, baldin eta, pertsona batek baino gehiagok aldi berean editatu nahi badute. Irakurtzerako orduan berriz ez. Moduetako bat fitxategi batetan idaztea da editatu nahi den fitxategia libre dagoenean.
Programa zatia ondorengoa izan daiteke:

If [-f /tmp/lekukoa] //fitxategia existitzen bada /tmp/lekukoa...
Echo Lanpeturik
Else
Echo X > /tmp/lekukoa
Echo Libre
Amaitu

Programa hau exekutatu eta gero eginbeharreko guztia egina dagoela adierazi behar dugu, beraz dokumentua askatu beharko da. Honen bidez:

Rm /tmp/lekukoa
Echo aske

Jakiteko libre dagoen edo ez lekukoa fitxategia behin eta berriz irakurri beharko da.

Partekatutako atzipena kudeatzeko FIFO ilarak:

FIFO bat erabiliz, helburua da sortzen den FIFO fitxategia irakurri eta idatzi ahal izatea aldi berean. Horrela eskatzen duen lehenengoak egingo du prozesua lehenago. Horrela lortzen da, idatzi behar duten prozesuak ez egotea behin eta berriz fitxategia libre dagoen galdetzen baizik eta era ordenatu batean, ilara bat eginez, banaka idazten joango direla fitxategia libratzen den heinean.

Sistema Eragilea Prozesu-kudeatzaile multzo gisa

Sistema-eragile batek sistemako prozesuek egiten duten baliabideen erabilpena kudeatzen du. Sistema-eragile batzuk bezero-zerbitzari eredua jarraitzen dute. Hasieratzerako orduan sortatako prozesuen zuhaitza egiaztatzeko ***pstree*** komandoa erabiltzen da. Honen eskema ondorengoa da: