Sistema Eragileak

Kernel baten simulatzailea

Josu Loidi

2021eko urtarrilaren 19a

Aurkibidea

Sarrera

Kernela sistema eragilearentzat behar-beharrezkoa den softwarea da. Programei ordenagailuaren edo bestelako gailu informatikoen *hardware*ra sarrera segurua ematea da bere arduretako bat, hau da, baliabideen eta prozesuen kudeaketaren arduraduna da.

Beraz, praktika honetan kernel bat simulatuko dugu. C programazio lengoaia erabiliko dugu horretarako, behe mailako programazio lengoaia baita, eta gure ezagutza handitzeko aukera ona izan daiteke.

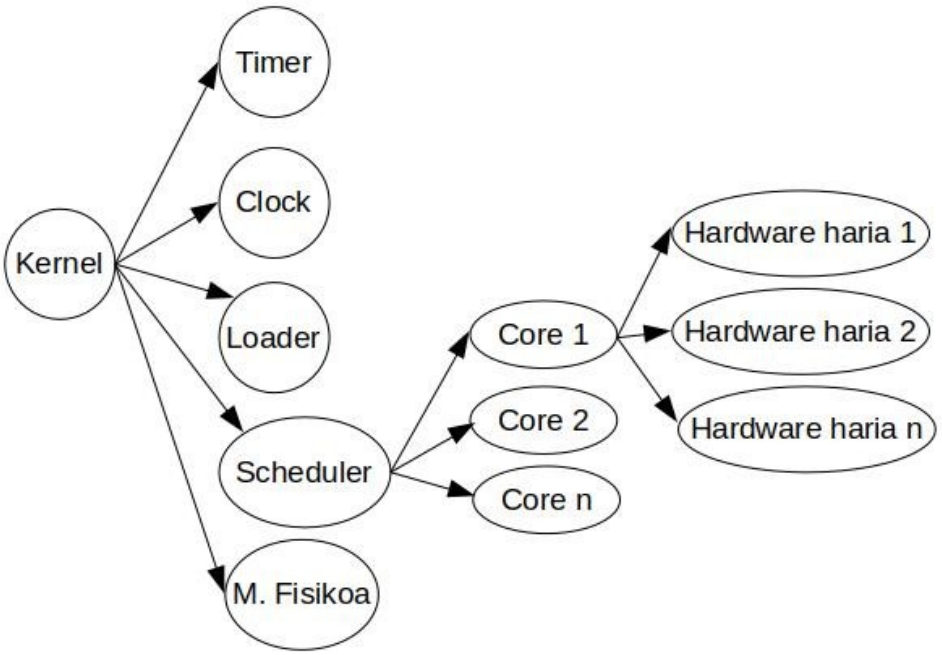
Praktika hainbat ataletan dago banatuta. Lehenik, sistemaren egitura definitu behar da, beharrezkoak izango zaizkigun datu egitura eta *thread*ak eraiki eta inplementatuz. Ondoren, oinarrizko funtzionamendua eginda dagoenean, prozesuen planifikazioa erabaki eta inplementatuko da, sortutako prozesuak dagozkien hardware harietan noiz eta nola exekutatu daitezken erabakiz. Behin prozesuak hardware hariei esleitu daitezkeenean, prozesu horiei forma eman behar zaie, hau da, exekutagarria den prozesu bat izango dugu, bere erregistroak aldatzeko aukera emango diguna.

Praktika 3 zatitan dago banatuta, baina txosten honetan zati guztiak batera azalduko dira, hala inplementatuta dagoelako.

Sistemaren egitura

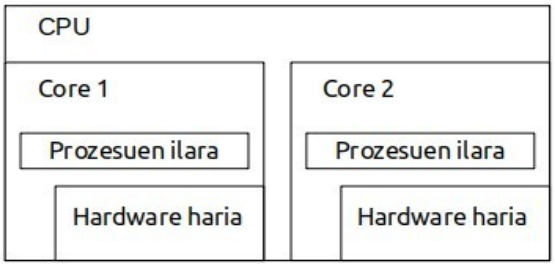
Simulazio honen sistemak funtzionatu dezan 6 atal nagusi sortu dira: *clock, timer, loader, scheduler, kernel* eta *memoriaFisikoa*. Atal horietako bakoitzak honako funtzioa betetzen du sisteman:

- *Kernel*: simulazioaren *main* programa. Bertan hasten da exekuzioa eta honek prestatzen du sistema guztia martxan jarri dadin (aldagaiak esleitu, *thread*ak sortu...).
- *Clock*: sistemaren erlojua. Sistemak funtzionatu dezan beharrezkoa da denbora kontrolatzea, eta horixe egiten du prozesu honek.
- *Timer*: sistemako seinaleak sortzeaz arduratzen da. Gure kasuan, erlojuak denbora tarte definitu bat pasatzen duenean seinalea egingo dio dagokion prozesuari.
- *Loader*: esan moduan, kernelak prozesuak kudeatzen ditu, eta hortaz, prozesu horiek simulatzeko sortu egin behar dira. Loaderrak hori egiten du, maiztasun baten arabera prozesuak sortzen ditu eta fitxategi batetik irakurriz kodea kargatzen zaio.
- *Scheduler*: aipatutako prozesuen kudeaketarako (noiz exekutatu, non, nola...) beharrezkoa izango da *schedulerr*a, honek egingo baititu beharrezkoa diren esleipen eta testuinguru aldaketak.
- *Memoria fisikoa:* memoria kontrolatzeko beharrezkoa izango da, izan ere, prozesuak eta orri taulak gordetzeko memoria bat sortuko dugu, eta hori kontrolatzeko beharrezkoa izango da modulu hau.

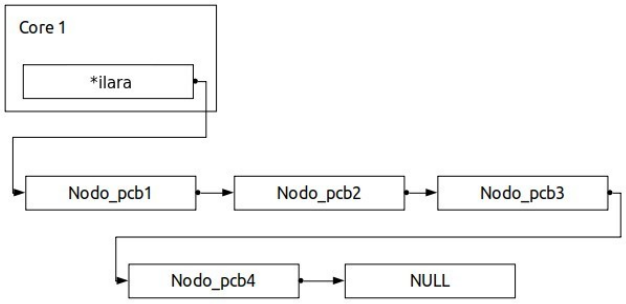


Bestalde, hainbat datu egitura definitu dira, oso beharrezkoak izango baitira informazioa guk nahi dugun moduan gorde eta kudeatzeko. Hemen aipatuko diren datu egitura guztiak C lengoaiak eskaintzen duen *struct* bidez sortuak izan dira.

- *CPU*: Sistemak izango duen CPU edo prozesagailua adierazten du. Bertan, honen IDa eta izango dituen *core*ak gordetzen dira.



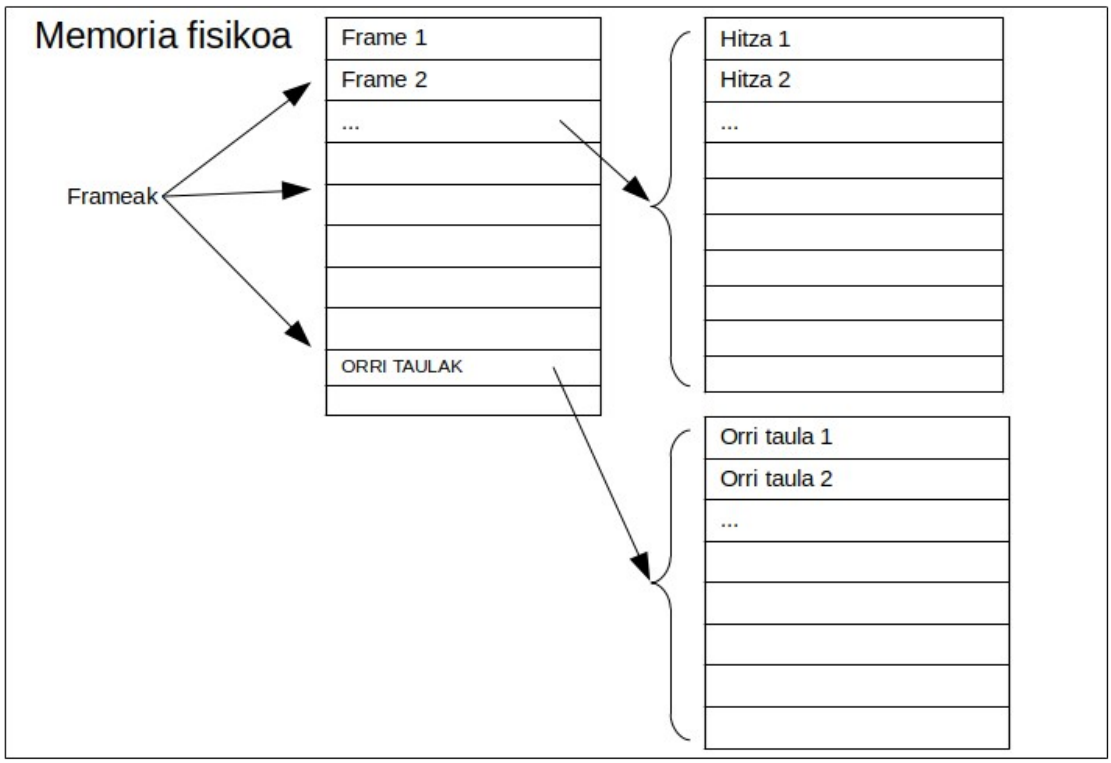
- *Core*: CPU bakoitzak dituen *core*ak adierazteko datu egitura. Kasu honetan, IDa izateaz gain, coreak izango duen prozesuen ilara ere gordetzen da. Horretaz gain, coreak izango dituen hardware hariak adierazteko ere erabiliko da.



Core baten ilararen errepresentazioa. Prozesuak gordetzeko ilara orokorra ere modu berdinean dago osatuta.

- *Hardware haria*: core bakoitzak dituen hardware hariak gordetzeko datu egitura. Egitura honetan, IDa gordetzeaz gain, hariak exekuzioan izango duen prozesuaren helbidea gordeko du. Gainera, exekuziorako beharrezkoak izango diren parametroak ere gordeko dira (PC, IR eta R erregistroak adibidez).
- *PCB*: prozesuak adierazteko datu egitura. Prozesu bat sortzen denean informazio asko gorde behar da berez, duen kodeaz gain. Hala ere, gure kasuan, prozesu zenbakia, *quantum*a, prozesuaren egoera eta memoria kudeatzeko aldagaiak izango ditugu.
- *Memory management (mm)*: prozesu bakoitzak memoriarekin izango dituen hartu-emanak kudeatzeko beharrezkoak izango diren aldagaiak gordetzen dira, hala nola, orri taularen helbide fisikoa, kodea eta datuen helbide birtualak eta exekuziorako erregistroak.

- *Memoria fisikoa*: memoria fisikoa simulatzen duen egitura. Array batez osatuta dago, posizio bakoitza frame bat izango da eta frame bakoitzak beste array bat izango du frame horretako desplazamendurako.



- *Process queue* (*Node_pcb*): *processGenerator*-ek sortzen dituen prozesuak gordetzeko ilara bat behar da, eta kasu honetan, *linked list* deitzen diren ilarak erabili dira. Ilarako elementu bakoitzak *PCB*a eta hurrengo ilarako elementua gordetzen ditu.
- Parametroak: simulazioa exekutatzerako garaian parametro batzuk sartu behar dira, eta hauek gordetzeko erabiltzen da datu egitura hau. Tartean daude, timer-aren maiztasuna, prozesuak sortzeko maiztasuna, CPU kopurua eta core kopurua.

Hala ere, aipatu behar da sistema fitxategi desberdinez osatuta egon arren, beharrezkoa dela datu egitura hauek dituzten aldagaiak elkarbanatzea. Horregatik, datu egitura hauek eta aldagai asko *globals.h* fitxategian daude definituta, eta beste fitxategi guztiek hau *inportatzen* dutenez, erabilgai daude toki guztietan. Hala ere, aldagai bera erabili nahi bada, *extern* aurrizkia jarri behar zaio aldagaiari definizioan, jakin dezaten beste fitxategiek kanpoko aldagai bat izango dela.

Sistemaren diseinua

Sistema honek zein egitura duen gutxi gorabehera azaldu ondoren, praktika burutzeko erabili diren politika eta inplementazioak azalduko ditugu. Esan bezala, praktikaren helburua, prozesuak sortzea eta exekutatzea da, horregatik, beharrezkoa da sistema guztiak nola funtzionatzen duen eta zein politika erabili diren ondo azaltzea.

Sistemaren oinarrizko funtzionamendua

Sistema martxan jartzeaz *kernel.c* fitxategiko prozesua arduratzen da, hau izango da prozesu nagusia. Hasteko, sartutako parametroak gordetzen ditu eta CPU datu egitura osatzen du. Ondoren, aipatutako modulu bakoitzarentzako (*clock, timer, loader, memoria* eta *scheduler*) hari bat sortzen da *pthread* liburutegia erabiliz, eta hari horiek jarraituko dute exekuzioa etengabe, ez baita aurreikusten programa inoiz amaitzea.

Clock.c fitxategian erlojuaren kontrola eramaten da. *Mutex* motako blokeatzailea erabilita, sekzio kritiko batean dagoen aldagaia babesten da. Aldagai horrek programa osoaren denbora kontatzen du, erlojuaren ziklo bakoitzeko balioa handituz. Gainera, exekuzioan dagoen prozesu (PCB) bakoitzaren denbora ere kudeatzen du, horretarako erabili den aldagai baten balioa aldatuz.

Bestetik, *timer.c* fitxategia dago. Fitxategi honetan *clock*ak eguneratzen duen aldagaia irakurtzen da etengabe, eta maiztasun finko bat igaro ondoren, seinale bat bidaltzen dio *scheduler*ari, honek bere lanarekin jarraitu dezan. Maiztasuna pasa denean, denbora kontrolatzen duen aldagaia babesteko, *mutex* motako blokeatzailea erabiltzen da hemen ere, *clock*ean erabiltzen den aldagai bera; horrela, ez da errorerik gertatuko aldagaiaren idazketan. Honetaz gain, *timer* eta *scheduler* prozesuak sinkronizatuta daude, izan ere, semaforo bat erabiltzen da bi hauen arteko komunikaziorako. Maiztasuna igaro denean, *sem_post* eginez, *scheduler*-a geldi egotetik martxan jartzera pasatzen da, eta bere lana amaitzen duenean berriz gelditzen da *timer*aren hurrengo seinalea jaso arte.

Prozesuak sortzeko *loader.c* fitxategiko prozesua erabiltzen da. Kasu honetan, parametro moduan pasatako zenbaki bat hartzen da maximo moduan, eta maximo horren azpitik dagoen ausazko segundo kopurua hartuz, *pcb* egiturako prozesu bat sortzen da. Gainera, *pcb* bakoitzari kodea esleitu behar zaio fitxategi batetik, eta horretarako aurrez *prometheus* programak sortutako 50 fitxategietatik bat aukeratzen da ausaz, eta bertako kodea esleitzen zaio. Ondoren, lehen aipatu bezala, prozesu hori ilara batean sartzen da aurretik sortu diren beste prozesu guztiekin batera.

Oraingo honetan memoria ere erabiltzen eta kontrolatzen da, eta horretarako erabiliko dugu *memoriaFisikoa.c* fitxategiko modulua. Modulu honekin, memoria fisikoa “sortuko” dugu, aurrez esan bezala, bi array erabiliz. Lehen arrayak frameak adierazten ditu, eta beharrezkoa izango da “libre” aldagai bat, frame hori libre dagoen ala zerbait daukan adierazteko. Gainera, bertan definitzen dira *MMU* eta programen fitxategiak irakurtzeko eta esleitzeko funtzioak ere.

Amaitzeko, *scheduler.c* fitxategian aurkitu ditzakegu prozesurik konplexuenak. Bertan, hasteko, egongo den *core* bakoitzarentzako *thread* edo hari bat sortzen da, eta honek izango duen prozesuen ilara ere definitzen da. Ondoren, *timer*raren seinalea jasotzen den bakoitzean prozesuen ilara orokorretik hari bakoitzari dagokion prozesua gehitzen zaio ilaran, orain aipatuko den politikaren arabera. Amaitzeko, *core* bakoitzak duen hardware hari bakoitzarentzako ere *thread* bat sortuko da eta etengabe egongo da begizta infinitu batean berari dagokion ilaratik prozesuak hartu eta exekutatzen.

Planifikazio politika

Planifikazioa gauzatzeko modu desberdin asko daude, oraindik eta gehiago sistema osoaren inplementazioa librea denean. Hala ere, honakoa da planifikazioa egiteko egin diren erabakiak eta baldintzak.

Prozesuei dagokienez, erabaki da prozesu bakoitzak *quantum* denbora propioa izatea, hau da, *quantum* denbora hori pasatzen denean prozesua exekuziotik aterako da eta testuinguru aldaketa bat gauzatuko da. Gainera, beharrezkoa izango da prozesu bakoitza zein egoeratan dagoen adieraztea, horregatik, hiru une posible definitzeko aldagai bat definitu da *EG_ZAIN*, *EG_EXEK* eta *EG_AMAI* egoerekin (zain, exekuzioan eta amaituta).

Planifikatzaileari dagokionez, kontuan hartzekoa da aurreko bertsioak ez zuela *hyper-threading*-a jasaten, hau da, core bakoitzeko hardware hari bakar bat egon zitekeen. Hala ere, oraingo honetan posible da *hyper-threading* egitea, aurrerago azalduko den algoritmoa inplementatu delako.

Bestalde, erabaki da *core* bakoitzak prozesuak gordetzeko ilara bat izatea, *core* horretan exekutatuko diren prozesu guztiak gordetzen dituena. Gainera, *core* bateko ilaran sartu den prozesua ez da inoiz beste *core* baten ilarara pasako, beraz, bertan exekutatu beharko da amaitzen den arte.

Prozesuen esleipenari dagokionez, honako politika hau erabiltzea erabaki da. *Timerr*ak seinalea bidaltzen duenean, ilara orokorreko prozesuak banan banan esleituko dira *core*etako ilaretan, lehenengotik azken *core*raino ordenan; horrela, prozesu guztiak esleitu arte. Ilarako prozesu guztiak amaitzean, ordea, ilara ezabatu egiten da, orain prozesuen objektuak (pcb-ak) *core*etako ilaretan baitaude. Esan dezakegu *FIFO* politika erabiltzen dela esleipen honetan.

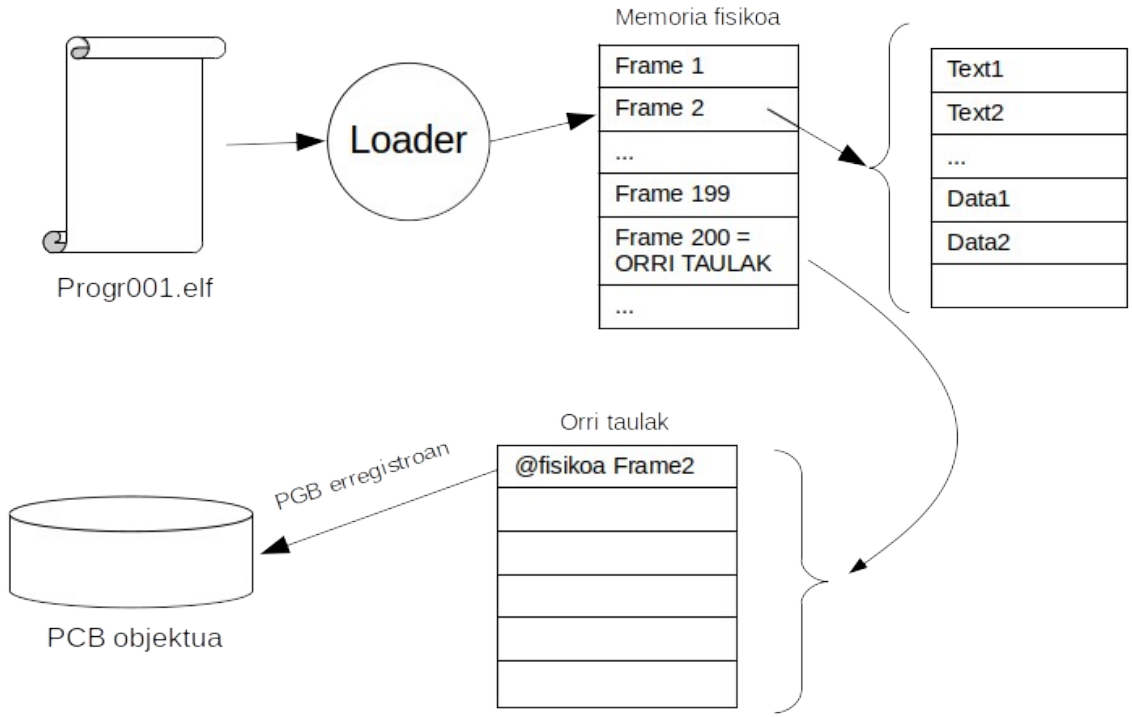
Prozesu bakoitzaren exekuzioa ilarako ordenan egingo da, hau da, ilarako lehen elementua (prozesua) hartuko da eta exekuziora “bidaliko” da. Ondoren, prozesuaren exekuzioa amaitzeko bi modu izango ditugu, *quamtuma* edo *exit* agindua. *Quantum*a igaro baldin bada, esan nahi du prozesua ez dela amaitu, eta berriz exekutatu beharra dagoela, beraz, *core* bereko ilaran sartuko da berriz, baina, azken posizioan. *Exit* agindua aurkitu baldin bada, berriz, prozesuaren exekuzioa amaitutzat ematen da eta testuinguru aldaketa hasten da, eta ilaratik ezabatzen da. Kontuan hartu beharra dago proposatu diren programak berez sortzen dituen programak oso motzak direla, eta *quantumari* ez diola denborarik ematen pasatzeko, lehenago exekutatzen baitira; eta *quantuma* oso txikia jartzen baldin badugu posible da agindu bat bera ere ez exekutatzea eta inoiz ez bukatzea. Horregatik, *Programak* karpetako programak luzeagoak dira, *prometheus* programako aldagaien balioak aldatuta.

Memoriari dagokionez, hasieran aipatu da matrize antzeko egitura bat jarraituko duela, bi *array* erabiltzen baitira bata bestearen barnean. Egitura hori honela dago osatuta: 24 biteko helbideak dauzkagu eta erabaki da 4KBeko (2^12B) orriak izango ditugula, hau da, 12 biteko desplazamendua izango dugu. Gainera, helbideratze tarte fisikoa eta birtuala berdina izango dela erabaki dugu, beraz, bi kasuetarako 24 biteko helbideak izango ditugu, horietako 12 bit desplazamenduarentzat gordeta.

Sortuko den prozesu bakoitzak, gutxienez memoriako frame oso bat izango du berarentzako, eta memoriara esleitzen joan ahala, frame bateko hitz guztiak betetzen baditu, hurrengo framea izango du eskuragarri. Frameen esleipena ordenan egingo da, hau da, aurkitzen den lehen frame librea emango zaio prozesuari, eta frame bakoitzean sartuko diren “hitzak” ere ordenan sartuko dira.

Bestetik, garrantzitsua da orri taulak erabiltzea. Orri taulekin, prozesu baten helbide birtuala jakinda bere helbide fisikoa lortu genezake, hau da, prozesuaren kodea eta datuak benetan memoriako zein helbidetan dauden jakin dezakegu. Orri taula hori, erabaki da frame bakar batean gordetzea, eta posizio edo desplazamendu bakoitza helbide fisiko bat gordetzeko erabiliko da.

Orri taula hori betetzeko, datuak eta kodea gordetzeko erabili den frame bakoitzeko orri taulako hitz bat erreserbatzen da, libre dagoen lehenengoa. Ondoren, *pcb* bakoitzari esleitutako orri taulako lehenengo orriaren helbide fisikoa gehitzen zaio *pgb* erregistroan.



Planifikazioaren inplementazioa

Erabakitako politika hau inplementatu egin dugu, azken finean hau baita praktikaren helburua, eta horretarako erabili diren algoritmo eta moduak azalduko ditugu orokorrean.

Hasieran aipatu da baina esan beharra dago prozesuen ilara guztiak *linked list* erabiliz inplementatu direla. Inplementazio honekin indizeez ahaztu gaitezke eta elementuen posizio aldaketak askoz ere azkarrago eta eraginkorrago egin ditzakegu. Beraz, bai prozesuak gordetzeko ilara orokorrak eta baita *core*etan dauzkagun ilarak *linked list*ak dira.

Planifikazioaren lehen zatia *core*etako ilaretan prozesuak esleitzean datza, eta esan bezala, banan banan esleitzen dira prozesuak *core*ak dauden ordenean. Horretarako, prozesu bakoitzeko nodo berri bat sortu da (prozesu bat izango duena) eta nodo berri hori esleitu zaio *core*ko ilaran azken posizioan. *Core*aren ilararen atzipena hardware hariak ere egiten du, beraz, *mutex* bat jarri da *core* bakoitzeko komunikazioan arazorik egon ez dadin.

Bestetik, *scheduler* prozesuak *core* bakoitzeko *thread* bat sortuko du, eta honi dagokion zenbakia, ilara eta *mutex* aldagaia pasako dizkio. Gainera, *core* bakoitzak duen *hardware hari* bakoitzeko ere beste *thread* bat sortuko du, egingo diren eragiketak paraleloan exekutatu daitezen. Hortaz, behin *thread*ak sortuta daudela, honakoa da programen exekuzioaren inplementazioa:

Hasteko, hari bakoitzak ilarako prozesu bat izan behar du esleituta, hala ez baldin badauka, eta horretarako *core*aren ilarararen hasierara joaten da eta ilara osoa pasatzen du azken prozesura iritsi arte edo EG_*ZAIN* egoeran dagoen prozesu bat aurkitu arte (exekutatu beharko den prozesu bat).

Beraz, behin ilara korrituta, badaezpada zain egoten da ea hurrengo ilarako elementua ez den hutsa, izan ere, kontuan hartu behar da ilararekin egiten ditugun atzipen ia guztiak aurreko elementutik hurrengo elementura begira egiten ditugula, oraindik gehitu gabe dauden prozesuen helbideetara iritsi gaitezen. Hona hemen aipatutako inplementazioaren zatiaren kodea, hobeto ulertzeko:

```
       /* ilara: ilara hasiera
          ilaraN: hariak duen ilararen punteroa */
       ilaraN = ilara;

       // Joan ilaran aurrera NULL bat aurkitu arte edo EG_ZAIN egoeran dagoen
prozesu bat aurkitu arte
       while (ilaraN->next != NULL) {
        if (ilaraN->next->data.martxan != EG_ZAIN)
         ilaraN = ilaraN->next;
        else
         break;
       }

      // Zain egon hurrengo prozesua dagen arte

      while (ilaraN->next == NULL);
      ilaraN = ilaraN->next;
```

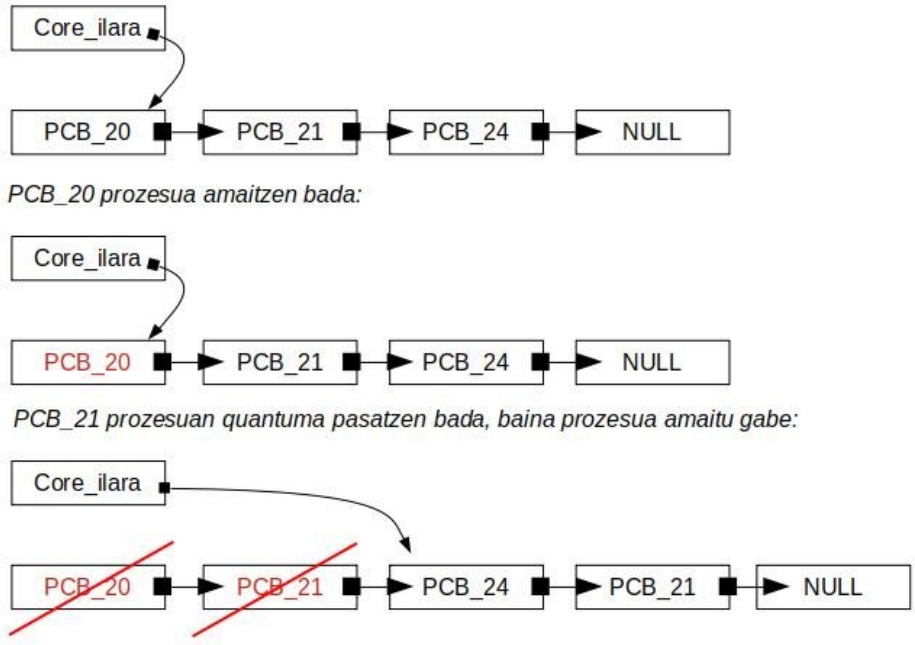
Ondoren, aukeratutako prozesua zain egoeran dagoela ziurtatzen du eta hala baldin bada, hariari prozesua esleitzen zaio (helbidea aldagai batean gehituz) eta mutex tartean prozesuaren egoera aldatzen da (exekuzioan dagoela adieraziz). Gainera, exekuziorako beharrezkoak izango diren erregistro guztiak ere kopiatzen dira, tartean, *PC*, *IR*, *PTDR* eta *R* erregistroak.

Beraz, behin hardware hariak prozesu bat esleituta duela, *quantuma* pasatzen ez den bitartean begizta batean sartzen da, eta begiztako iterazio bakoitzeko agindu bat exekutatzen da. Agindua exekutatzeko, PC erregistroa irakurtzen da jakin dezan zein den exekutatu behar den agindua, eta *MMU* funtzioa erabiliz, zein helbide fisikotan dagoen lortzen dugu. Funtzio honen eginbeharra sinplea da, prozesu baten orri taulako helbidea eta jakin nahi den helbide fisikoa edukita, memoria fisikoko orri taulako dagokion hitzean begiratuko du behar duen helbide fisikoa. Hori jakiteko, beharrezkoa da helbide birtualak zenbatgarren framea irudikatzen duen kalkulatzea, hau da, zein den helbide birtualaren lehen 12bit-en balioa.

Behin helbide fisikoa daukagula, desplazamendua kalkulatzen dugu maskara bat erabiliz, izan ere, desplazamendua azken 12bit-ek irudikatzen dute, eta hori baliagarria izango zaigu dagokigun frameko desplazamendurako. Beraz, helbide fisikoa eta desplazamendua edukita, agindua lortu dezakegu memoria fisikotik eta sortu dugun funtzioari deitu agindu hori exekutatu dezan.

Agindua exekutatzeko ere beharrezkoak dira maskarak, izan ere, agindua zenbaki hamaseitar bat da, eta aginduaren barnean sartuta daude zein operazio den, eta zein erregistro eta balio behar diren. Beraz, balio horiek guztiak lortu behar dira lehendabizi, eta ondoren, *switch* klausula bat erabiliz agindu bat edo bestea gauzatuko da, zuzenean *R*ko erregistroak aldatuz. Programaren hasieran eta amaieran memorian dauden balioak behar dira (irakurri eta idazteko) eta kasu horietan zuzenean memoria fisikora deitzen da eta helbideak badakizkigunez erraza da atzipena egitea. Programa amaitzeko, bi emaitza eman ditzake; batetik, agindua ondo bete bada 0 itzultzen du, baina *exit* agindu bat irakurri baldin bada 1 itzultzen du zuzenean funtzioak.

Hardware hariekin jarraituz, agindua exekutatu ondoren begiratzen da ea programa amaitu den ala ez, eta hala ez bada, *PC* eta *IR* erregistroak eguneratzen dira eta hurrengo iteraziora egiten da salto. Aitzitik, programa amaitu baldin bada, edo *quantum*a pasa baldin bada testuinguru aldaketa hasiko da. Testuinguru aldaketa gauzatzeko, lehenik erregistro guztien balioak *pcb*ari esleitzen zaizkio hurrengo exekuzio baterako, eta egoera aldatzen zaio *EG_AMAI* egoerara. Hala ere, prozesua ez bada amaitu eta hurrengo exekuzio batean exekutatu behar bada, ilararen atzera eramaten da, hau da, prozesuaren *pcb* kopia bat egiten da (egoera berriz ere *EG_ZAIN* jarriz) eta ilara korritzen da. Tartean zenbaki berdina duen prozesu bat aurkitzen badu zerrendatik “ezabatu” egiten du punteroak aldatuz, eta amaieran prozesuaren kopia hori gehitzen zaio. Horretaz gain, funtzio honi deitzen zaion guztietan, zenbaki berdina duen prozesua ezabatzeaz gain, *EG_AMAI* egoeran dauden prozesu guztiak ere ezabatzen dira (garbiketa bat egiten da noizean behin, hau da, amaitu diren prozesuak ere ezabatzen dira).



Amaitzeko, hardware hariari esleituta zeukan prozesua kentzen zaio *NULL* ipiniz, eta hurrengo iteraziora pasako da, berriz ere exekutatuz.