Sistema eragileak
Kernel baten simulatzea

1. Zatia: Sistemaren arkitektura

2. Zatia: Planifikazioa

Josu Loidi Astiazaran

2020/11/14

Aurkibidea

Sarrera

Kernela sistema eragilearentzat behar-beharrezkoa den software bat 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.

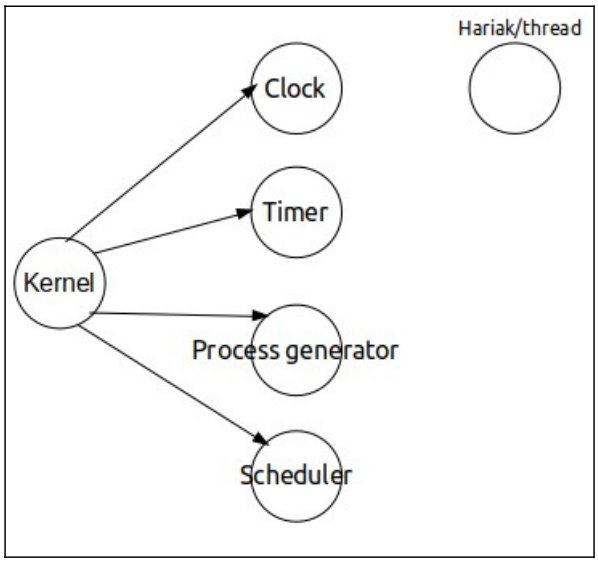
Praktika hainbat ataletan dago banatuta, baina txosten honetan lehen bi zatiak soilik azaltzen dira. Lehenik, sistemaren egitura definitu behar da, beharrezkoak izango zaizkigun datu egitura eta *thread*ak eraiki eta inplementatuz. Ondoren, oinarrizko funtzionamendua eginda dagoenean, prozesuen planifikazioa erabaki eta inplementatuko da, sortutako prozesuak dagozkien hardware harietan noiz eta nola exekutatu behar diren erabakiz.

Hala ere, txosten honetan ez dira bi zatiak asko desberdinduko, eta sistema guztiaren diseinua eta inplementazioa batera azalduko dira.

Praktika honetako sistemaren iturburu kodea *GitHub*eko honako biltegian aurkitu daiteke: https://github.com/josuloo99/SE_Praktika_JosuLoidi

Sistemaren egitura

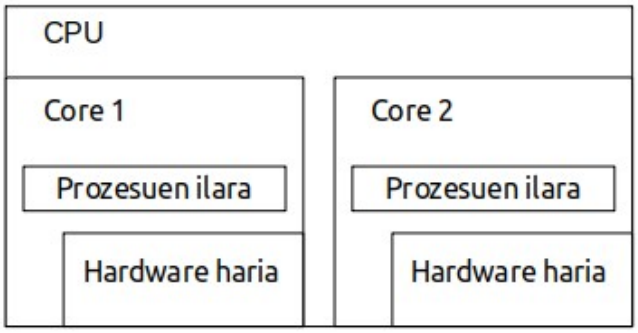
Simulazio honen sistemak funtzionatu dezan 5 atal nagusi sortu dira: *clock, timer, process generator, scheduler* eta *kernel*. Atal horietako bakoitzak honako funtzioa betetzen du sisteman:



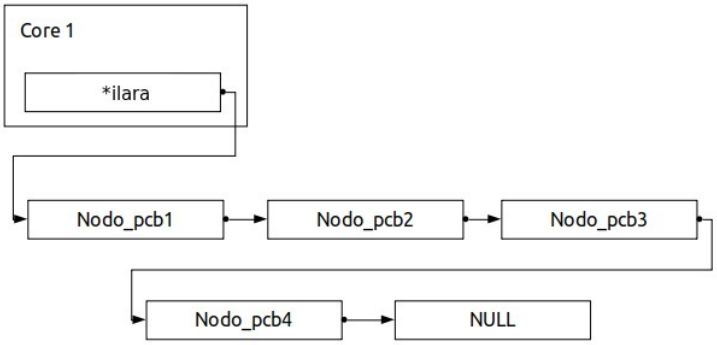
- *Kernel*: simulazioaren *main* programa. Bertan hasten da exekuzioa eta honek prestatzen du sistema guztia martxan jarri dadin (aldagaiak esleitu, *thread*ak sortu...).
- *Clock*: sistemaren erlojua. Sistemak funtzionatu dezan beharrezkoa da denbora kontrolatzea, eta horixe egiten du prozesu honek.
- *Timer*: sistemako seinaleak sortzeaz arduratzen da. Gure kasuan, erlojuak denbora tarte definitu bat pasatzen duenean seinalea egingo dio dagokion prozesuari.
- *Process generator*: esan moduan, *kernelak* prozesuak kudeatzen ditu, eta hortaz, prozesu horiek simulatzeko sortu egin behar dira. Horixe egiten du honek, prozesuak sortu maiztasun baten arabera.
- *Scheduler*: aipatutako prozesuen kudeaketarako (noiz exekutatu, non, nola...) beharrezkoa izango da *schedulerr*a, honek egingo baititu beharrezkoa diren esleipen eta testuinguru aldaketak.

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 duen hardware haria adierazteko ere erabiliko da.



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

- *Hardware haria*: aurrerago aipatuko da, baina gure sisteman *core* bakoitzak bere harware hari bakarra izango du, eta hori adierazteko sortu da honako datu egitura. Egitura honetan, *IDa* gordetzeaz gain, hariak exekuzioan izango duen prozesuaren ilarako helbidea gordeko du.
- *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, exekuzioan pasatako denbora eta amaitzeko ziklo kopurua soilik gordeko dira, simulazio bat baita.
- *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 ilarako hurrengo elementuaren helbidea 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 fitxategi guztiek hau *inportatzen* dutenez, ikusgai 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, bi zati hauen helburua prozesuak kudeatzen dituen kernel baten simulazioa egitea da, eta hori burutzeko hainbat modu desberdin

daude. Horregatik, beharrezkoa da sistema guztiak nola funtzionatzen duen eta zein politika erabili diren ondo azaltzea.

Sistemaren oinarrizko funtzionamendua

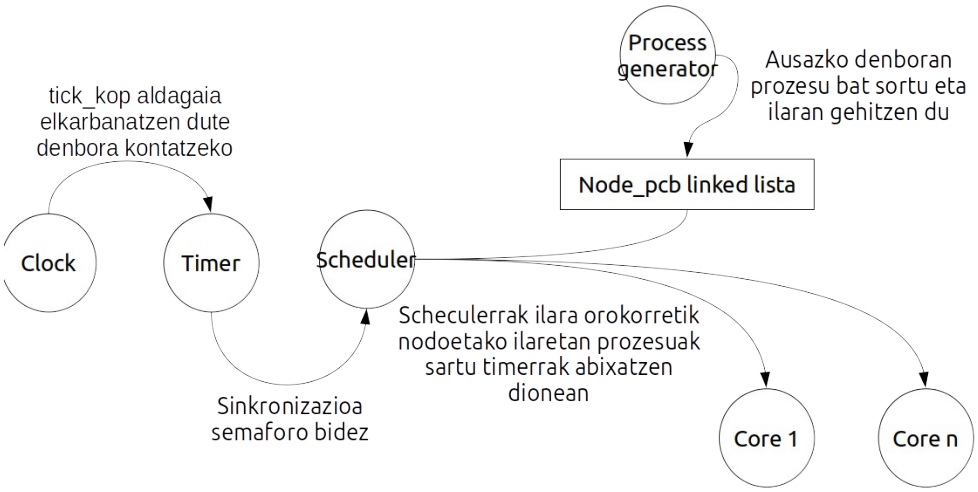
Sistema martxan jartzeaz *kernel.c* fitxategiko prozesua arduratzen da, hau izango da prozesu nagusia. Hasteko, sartutako parametroak gordetzen ditu eta CPU datu egitura osatzen du. Ondoren, aipatutako modulu bakoitzarentzako (*clock, timer, process generator* eta *scheduler*) hari bat sortzen da *pthread* 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 lana egin behar duela jakin 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 zain.

Prozesuak sortzeko *processGenerator.c* fitxategiko prozesua erabiltzen da. Kasu honetan, parametro moduan pasatako zenbaki bat hartzen da denbora maximo moduan, eta maximo horren azpitik dagoen ausazko segundo kopurua hartuz, *pcb* egiturako prozesu bat sortzen da. Ondoren, lehen aipatu bezala, prozesu hori ilara batean sartzen da aurretik sortu diren beste prozesu guztiekin batera, azken posizioan.

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*aren seinalea jasotzen den bakoitzean prozesuen ilara orokorretik hari bakoitzari dagokion prozesua gehitzen zaio ilaran, eta hariak exekutatu egiten ditu etengabe, orain aipatuko den politikaren arabera.



Prozesuen arteko elazioak

Planifikazio politika

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

Prozesuei dagokienez, erabaki da prozesu bakoitzak *quantum* denbora propioa izatea, eta amaitu ahal izateko ziklo kopuru jakin bat beharko dituela. Gainera, beharrezkoa izango da prozesu bakoitza zein egoeratan dagoen adieraztea, horregatik, prozesu bakoitzak uneko egoera definituko duen aldagai bat izango du.

Planifikatzaileari dagokionez, esan daiteke sistemak ARM arkitektura duela, ez duelako *hyper-threading*a onartuko, hau da, ez duelako *core* bakoitzak hardware hari bat baino gehiago izatea onartuko. Erabaki hau batez ere inplementazio arazoengatik hartu da, ez baita lortu modu orekatu eta eraginkor batean hari bat baino gehiago erabiltzea. Hala ere, proiektuaren bertsio berriagoetan aurreikusten da arazo hau konpontzea eta *hyper-threading*a onartzea sistemak.

Bestalde, erabaki da *core* bakoitzak prozesuak gordetzeko ilara bat izatea, *core* horretan exekutatuko diren prozesu guztiak gordetzen dituena. Hardware hari bakarra egongo denez *core* bakoitzeko, ilarako elementu edo prozesu guztiak *core* horretako hariak exekutatu beharko ditu. 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, *quantum*a pasatzen denean, prozesua exekuziotik aterako da eta erabakiko da amaitua izan den ala ez. Prozesu hori amaitua izan ez bada, esan nahi du berriz exekutatu beharra dagoela, beraz, *core* bereko ilaran sartuko da berriz, baina, azken posizioan.

Planifikazioaren inplementazioa

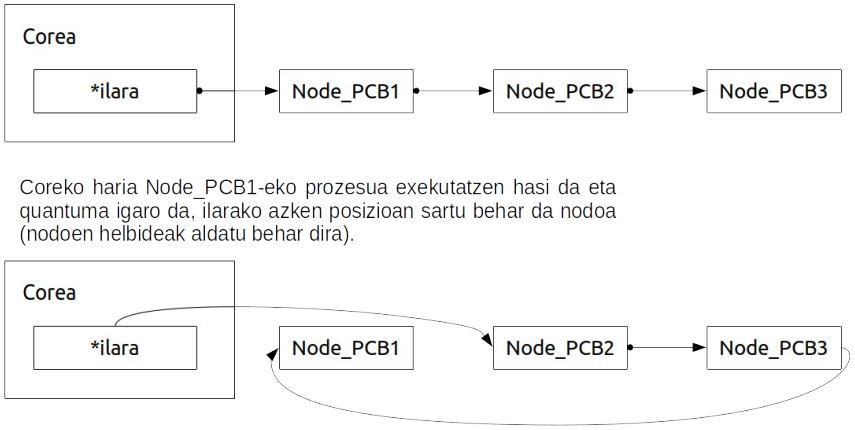
Erabakitako politika hau inplementatu beharra izan da, azken finean hau baita praktikaren helburua, eta horretarako erabili diren algoritmo eta moduak azalduko ditugu orokorrean.

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 ilarari azken posizioan. *Core*aren ilararen atzipena hardware hariak ere egiten du, beraz, *mutex* bat jarri da, komunikazio arazorik egon ez dadin.

Bestetik, *scheduler*rak hasi bezain laster *core* bakoitzeko *thread* bat sortuko du, eta honi dagokion zenbakia, ilara eta *mutex* aldagaia pasako dizkio.

Amaitzeko, honakoa da hardware hari bateko inplementazioa. Hariaren exekuzioa simulatu egin da, hau da, ez da benetan beste *thread* bat sortu exekuzioa egin dadin, baizik eta *core*aren *thread* horretan bertan begizta mugagabe bat egin da momentu oro hariaren prozesua begiratzen duena (libre edo okupatuta dagoen). Ondoren, horren arabera, prozesu berri bat esleituko zaio exekutatzeko (ilaran dagoen lehenengoa, baldin badago), edo bestela, exekuzioko prozesuak igaro duen denbora *quantum*arekin konparatuko da eta behar baldin bada exekuziotik aterako da haria libre utziz. Prozesuak *quantum*a igaro badu baina ez bada amaitu, ilarako azken posizioan kokatuko da eta ilararen lehenengo elementua hurrengo prozesua izango da.



Testuinguru aldaketa bat gertatzen denean ilaretan egiten den aldaketa