SESO

2021.eko urtarrilaren 19

Euskal Herriko Unibertsitatea (UPV/EHU)

Sistema Eragileak

Unai Fernandez

Gaien Aurkibidea

1 Sarrera

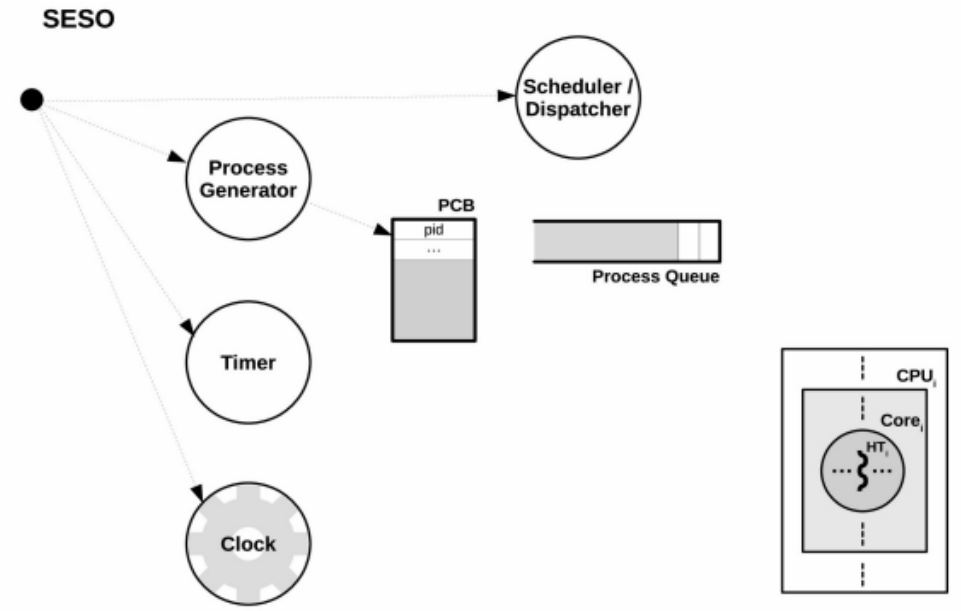
Proiektu honen helburua, sistema eragile baten kernela simulatzen duen hari anitzeko programa bat garatzea da. Proiektua hainbat fasetan banatuko da eta hauetako bakoitzean sistema baten oinarrizko osagarriak inplementatuko dira. Sistemaren programazioa, C lengoaian egina dago eta simulatzaileak hainbat parametro jasotzen ditu konfigurazio aukera bezala.

Proiektu honetan zehar sistema eragileekin eta hauen funtzionamenduarekin zerikusia duten kontzeptuak landuko dira, hauen ezagutza ezinbestekoa izango baita proiektuaren inplementaziorako. Gai horien artean honako hauek topa ditzakegu, scheduler motak eta hauen funtzionamendua, memoriaren kudeaketa, edota sistemaren oinarrizko ariktektura.

Proiektua hiru fase nagusitan banatu da, bakoitzean sistemako zati baten garapena egin da. Ondorengo ataletan fase hauetako bakoitza sakontasunean aztertuko dira.

2 Lehenengo Fasea: Sistemaren oinarriak

Leheneno fase honetan, gure sistemak proiektuan zehar erabiliko duen oinarrizko egitura inplementatuko da. Egitura hau konposatzen duten osagarri bakoitzak hari desberdin batean exekutatuko dira. Hurrengo irudian [1] ikus daitekeen bezala honako hau izango da kernel simulatzaileak izango duen egitura.



Irudia 1: Lehen faseko planteamendua

Aurreko eskeman antzeman daitekeen bezala egin beharreko programak hainbat osagaik osatzen dute. Zirkulu baten barruan daudenak kernelaren oinarrizko funtzioak dira; erlojua, timerra, prozesu sortzailea eta antolatzailea. Funtzio hauetako bakoitza hari independiente batean exekutatuko da, baina denen artean sinkronizatuta egotea ezinbestekoa da sistemak ondo funtziona dezan. Horretarako hainbat sinkronizazio baliabide erabili dira egoeraren arabera. Hauetaz aparte sistemak erabiliko dituen egitura batzuk definitu behar ditugu, esaterako PCB (Process Control Block) edota CPU-a.

2.1 Osagaiak

Atal honetan sistemaren funtzionamendurako naitaezkoak diren osagaiak azalduko dira.

2.1.1 Clock

Sistema batean denborak aurrera egiten duela irudikatzeko, sistemako erlojua deritzona erabiltzen da. Sistemako erlojua edo System clock, seinale bat baino ez da eta aurretik zehaztutako maiztasun batean oszilatu egiten du. Uhin honetan pasatzen den periodo bakoitzeko *tick* bat bidaliko da CPU-a etenduz momentu horretan. Seinale hau edozein sisteman oso garrantzitsua da. Honi esker beste funtzioak, *clock*-aren pultso bakoitzeko, "mugitzeko"ahalmena ematen baitu.

2.1.2 Timer

Timerra, clock-aren antzeko seinalea da, baina kasu honetan denbora tarteak edo interbaloak neurtzeko erabiltzen da. *Clock* seinalearen antzean zehaztutako maiztasun batekin oszilatuko du seinaleak eta periodo bakoitzeko seinale bat bidaliko du.

2.1.3 Scheduler/Dispatcher

Schedulerra prozesuak planifikatzeaz arduratuko da. Behar den momentuan exekuzioan dagoen prozesua CPU-tik kenduko du eta beste bat aukeratuko du bertan exekuzioan sartzeko. Hau posible izateko hainbat planifikatzaile mota erabili daitezke, esaterako O(1) edota CFS. Diseinatutako sistema honetan CPU-ko core bakoitzeko scheduler bat egongo da, eta bakoitzak bere prozesu ilara izango du.

2.1.4 Process Generator

Izenak esaten duen bezala funtzio hau prozesuak sortzeaz arduratuko da. [1] irudian ageri den PCB egiturako osagai bakoitzari balio bat emango zaio prozesuak simulatu ahal izateko.

2.1.5 Process Control Block

PCB-a datu egitura bat da eta bertan prozesu bati buruzko informazioa gordeko da. Esaterako prozesuaren identifikadorea, erregistroak edota prozesuaren egoera.

```
struct process_control_block{
    int pid;
    int nice;
    int weight;
    int vruntime;
    float decay_factor;
    int egoera;
    int rtime;
    int ptbr;
    struct mm mm;
    int err[16];
};
```

2.1.6 Process Queue

Sistema eragile batean prozesuen egoeraren araberako bezainbeste ilara egongo dira. Gure sisteman prozesuek bi egoera izango dituzte, exekuzioan eta prest. Prozesu bat exekuzioan ez dagoen bitartean prozesu ilaran egongo da aukeratua izan arte.

2.1.7 CPU

CPU egiturak, gure sistemako prozesuak exekutatuko ditu, hainbat corez osatuta egongo da. Erabiltzaileak zehaztuko ditu programa exekutatzerakoan. Lehen esan bezala core bakoitzak scheduler bat eta prozesu ilara bat izango dituzte. Hasiera batean core bakoitzak hari bat izango du.

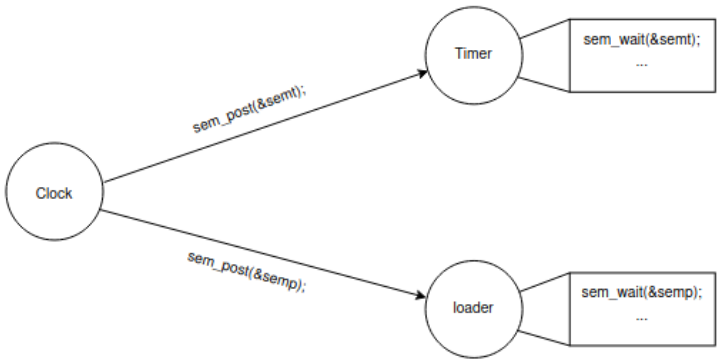
2.2 Osagaien sorketa eta sinkronizazioa

Lehen aipatu bezala sistemako osagai bakoitza bere hari independientean exekutatuko da. Hari hauek era egokian sortzeko *main* izaneko programa orokorra sortu da. Hariak sortzeaz arduratzeaz gain erabiltzaileak parametro moduan sartzen dituen konfigurazio parametroak irakurri egingo ditu getopt erabilita. Modu honetan eginez, programa komando lerroan exekutatzeko orduan parametro bakoitzaren aurrean identifikadore bat erabiliko da, balio bakoitzaren esanahia determinatzeko.

```
./binaries/seso -p4 -m1 -t40 -c1
```

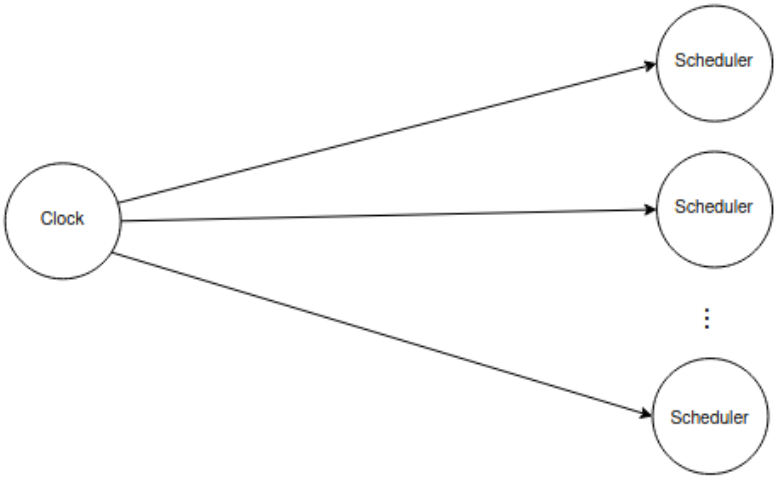
Parametro bakoitza prozesatzeko *switch* bat erabiliko da, zenbakiaren aurreko letra edo identifikadorearen arabera datuan interpretatzeko. Hau eginda core kopurua, prozesu kopurua edota erloju maiztasuna zehaztu dezakegu.

Hari guztiak sortzeko funtzio bat erabiliko da *sortu_hariak*. Lehen prozesatutako parametroak kontuan hartuko dira, funtzio bakoitza sortzerako orduan. Adibidez, *clock* funtzioa sortzerakoan erloju maiztasuna beharko dugu. Hari guztiek elkarren artean lan egin beharra dutenez sortu eta gero sinkronizatu egingo dira. Horretarako bi modu desberdin erabili dira, semaforoekin eta mutex baldintzatu bat erabiliz. Semaforoen kasuan, timerra eta prozesu sortzailea clock-arekin sinkronizatzeko primerakoak dira. Bakoitzak semaforo desberdin bat erabiliko du. Clock funtzioak *tick* bat sortzen duenean semaforo bakoitzean post egingo du. Timerrak eta loaderrak beraien semaforoak wait egoeran daudenez leku bat egon arte itxaroten egongo dira, beraz clock-ak post egiten duen bezain pronto exekutetuko dute bere kodea. Ikus [2] irudia.



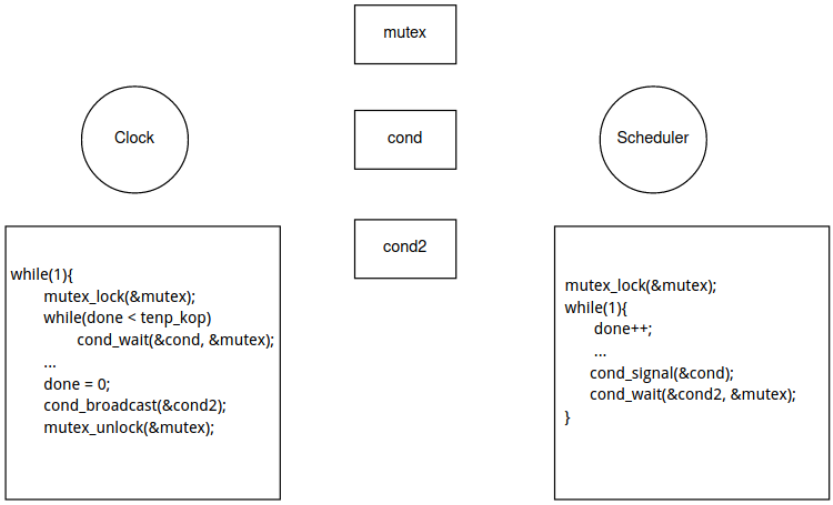
Irudia 2: Clock-a timerra eta loaderraren arteko sinkronizazioa

Schedulerraren kasuan aldiz ezin izango dira semaforoak erabili. Kasua da ezin izango dela kontrolatu scheduler bakoitzak behin exekutatu dela. Semaforo bat erabili beharko litzateke scheduler bakoitzeko hori kontrolatu nahi badugu, ondoko irudian [3] agertzen den bezala. Scheduller kopurua txikia bada arazoa ez litzateke handia izango baina kopurua handitzen doan heinean zailagoa egiten da semaforoak sortzea.



Irudia 3: Schedulerra eta clock-aren arteko sinkronizazioa semaforoekin

Arazoa konpontzeko mutex baldintzatu bat erabili dezakegu, teknika honen bidez lortuko baitugu scheduler bakoitza behin exekutatzea. Ondoko irudian [4] ikusi daieten honen eskema orokorra.



Irudia 4: Schedulerra eta clock-aren arteko sinkronizazioa mutex baldintzatuarekin

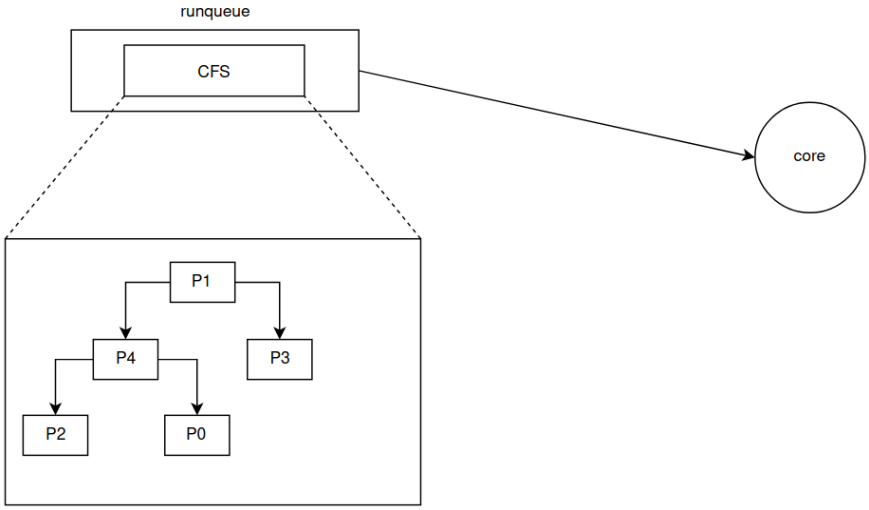
Honen funtzionamendua semaforoena baino konplexuagoa da. Irudian [4] ikus daitekeen bezala *done* aldagaia erabiltzen da scheduler kopurua kontrolatzeko. Scheduler bakoitza exekuzioan hasten denean bakioa inkrementatu egiten du. *done*-ren balioa core kopuruarena baino txikiagoa den bitartean exekutatuko da scheduller bakoitzaren kodea. Nola dakigu planifikatzaile bakoitza behi exekutatu dela?
cond erabilita jakingo dugu. Aurreko irudian ikusi daiteke cond_signal exekutatu eta gero cond_wait exekutatzen dela. Horrek schedulerra itxaroten geratuko dela cond_broadcast exekutatu arte esan nahi du, eta hori lau schedulerrak exekutatu eta gero da.

3 Bigarren Fasea: Prozesuen planifikazioa

Proiektuko bigarren fase honetan, lehenengo zatian garatutakoa, oinarri bezala hartuta, scheduler/dispatcher bat inplementatu behar da. Scheduler baten helburua, prozesuak planifikatzea da, prozesadorearen errendimendua miximizatzeko asmoarekin.

3.1 Prozesuen planifikazioa

Gure kasuan prozesu bakoitzak bi egoera izango ditu itxaroten eta exekuzioan. Honako hau izango litzateke lortu nahi dugun egituraren itxura.



Irudia 5: Schedulerraren egitura

Irudiaren [5] ezkerraldean ikusi daitekeen bezala, itxarote ilara bat dago bertan prozesuek, core-an exekutatuak izatera itxarongo dute. Scheduler baterako hainbat planifikazio politika existitzen dira, hala nola, O1 edo CFS politikak. Proiekturako CFS politikaren inplementazioaren antzekoa egiten ahalegindu naiz. CFS (Completely Fair Scheduller) politikan, itxaroten dauden prozesuak zuhaitz gorri-beltz batean gordetzen dira. Zuhaitz mota honen konplexutasunagatik, BST bat erabili dut prozesuentzako, antzeko funtzionamendua lor daitekeelako. Bigarren fase honetan ere, prozesuek bere buruaren informazio gehiago gorde behar dute, *vruntime*, *weight* eta $hondatze faktorea$.

- *vruntime*: exekuzio denboraren balioespena.
- *weight*: *nice* balio bakoitzari esleitzen zaion pisua.

- $decay - factor$: prozesu baten hondatze faktorea.

```
struct process_control_block{
    int pid;
    int weight;
    int vruntime;
    float decay_factor;
};
```

Prozesu bakoitza zuhaitzean sartzerakoan, bere *vruntime*-ren arabera ordenatuko dira, txikienak ezkerraldean daudelarik. Beraz, Zuhaitzean *vruntime* txikiena daukana, hau da ezkerraldean dagoena, izango da exekuziora pasako dena tokatzen denean. Prozesu berri bakoitza exekutatzera pasako da, exekuzioan dagoen prozesuak exekuzioa bukatzen badu edo quantum-a agortzen bazaio. Bigarren kasuan, exekuzioan zegoen prozesuaren *vruntime* txikiagoa izaten jarraitzen badu exekuzioan jarraituko du, bestela prozesu berria sartuko da core-an eta bestea zuhaitzera joango da. Prozesu bat core-an edo exekuzioan dagoen bitartean *vruntime*-ren balioa aldatzen joango da. Aipatutako balio hori aldatzeko, prozesu baten beste bi parametroak eta timer-aren maiztasuna beharko ditugu. *weight* balioa, prozesu baten hondatze faktorea kalkulatzeko balio du. Parametro honek -20-tik 15-era balioak har ditzakeen *nice*-ren araberakoa da.

```
static const int weight[40] = {
/* -20 */ 88761, 71755, 56483, 46273, 36291,
/* -15 */ 29154, 23254, 18705, 14949, 11916,
/* -10 */ 9548, 7620, 6100, 4904, 3906,
/* -5 */ 3121, 2501, 1991, 1586, 1277,
/* 0 */ 1024, 820, 655, 526, 423,
/* 5 */ 335, 272, 215, 172, 137,
/* 10 */ 110, 87, 70, 56, 45,
/* 15 */ 36, 29, 23, 18, 15};
```

Ikus daitekeenez esan bezala balio guztiak hasieratik daude definituta, eta prozesuak hartzen duen *nice* balioaren arabera $(-20, ..., 15)$ *weight*-en array-ko balio bat hartuko du. Hartzen den balio berri hori izango da $decay factor$ kalkulatzeko erabiliko dena.

$$decay\_factor = \frac{weight_0}{weight_i} \tag{1}$$

Adibide moduan, gure prozesuak -10 balioa badauka *weight* parametroan gordeta,

array-ko 9548 balioa hartuko luke, eta hau izango litzateke bera hondatze faktorearen kalkulua:

$$decay\_factor = \frac{weight_0}{weight_i} = \frac{1024}{9548} = 0,11 \tag{2}$$

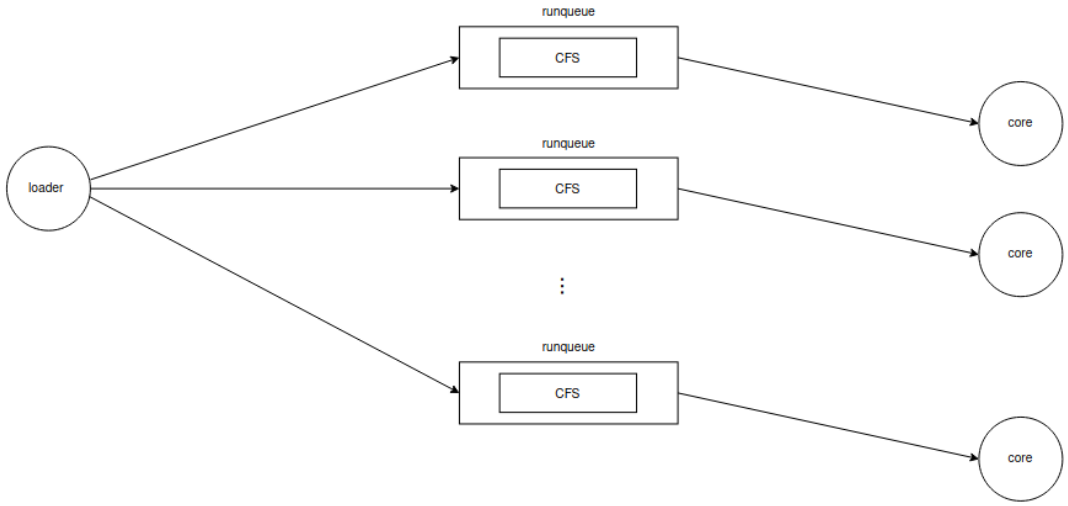
Lortu berri dugun hondatze faktorearen balioa garrantzitsua izango da, *vruntime* eguneratzeko. Honako formula hau erabili beharko da horretarako:

$$vruntime = vruntime + \sum t_{cpu} * decay\_factor \tag{3}$$

Egindako sisteman kalkulu hauek guztiak timerrak *tick* bat bidaltzen duenean exekutatuko dira core bakoitzean. Azkenik lortutako balioaren arabera erabakiko da prozesua zuhaitzeko zer posiziotan sartuko den.

3.2 Prozesuen sorketa

Aurretik esan bezala, prozesuek informazio berria gorde behar dute, beraz prozesuak sortzeaz arduratzen den harian aldaketak egin beharko ditugu. Lehenik eta behin, prozesu baten parametro berriei balio bat eman behar diegu, horretarako ausaz zenbaki bat esleituko zaio bai *vruntime* eta bai *weight* parametroei. Prozesuetako parametro guztiei balioak eman eta gero, sortutako prozesu berria itxarote ilara batean sartu beharko da. Prozesua zein core-ko zuhaitzean sartu behar den jakiteko funtzio bat erabiliko da, *getmin*. Funtzio honek core gustien artean prozesu gutxien dituena identifikatuko du, prozesu berria bertan sartzeko.



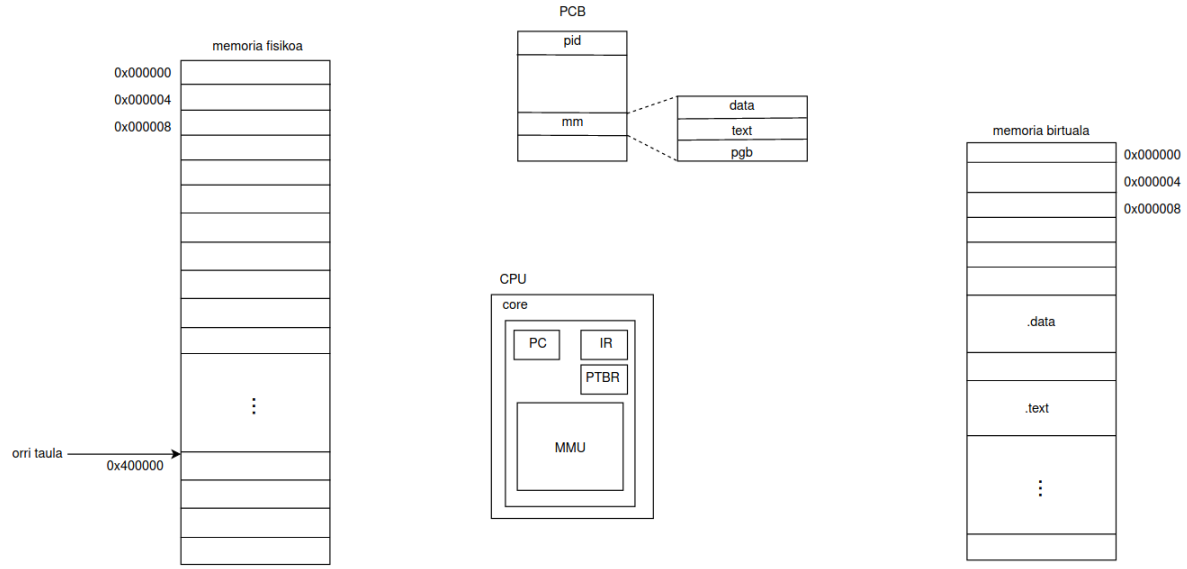
Irudia 6: loader eta schedulerrak bateratuta

4 Hirugarren fasea: Memoriaren kudeaketa

Hirugarren zati honen helburua, aurreko bi ataletan garatutako sistemari memoria birtuala kudeatzeko gaitasuna ematea da. Zati honen garapenarekin hasi aurretik, erabili beharreko baliabide guztiak argi eduki behar dira, horregatik hasi aurretik egitura berria definituko da.

4.1 Sistemaren osagaiak

Fase honetan hainbat aldaketa egongo dira aurretik geneukan sisteman, osagai berriak gehitzeaz gain. Ondorengo irudian [7] ikusi daitezke atal honetako osagai garrantzitsuenak.



Irudia 7: Memorien eskema orokorra

Hurrengo lerroetan aurreko irudian ikus daitezkeen osagaiak banan-banan aztertuko dira.

4.1.1 memoria fisikoa

memoria fisikoan gordeko dira programa guztiak, bai bakoitzaren kodea zein datuak. Memorian bi zati nagusi egongo dira, orri taula eta frameak. Memoriako azken helbidea $0xFFFFFF$ izango da, horrek esan nahi du memoriaren tamaina $2MB$-koa izango dela. Bertatik hasita gure memoria konfiguratzeko informazio asko atera daiteke.

- helbidea fisikoen tamaina: 24 bit

- frame (marko) kopurua: 2^{20}
- desplazamendua: 4 bitekoa izango da, [7] irudian ikusten den bezala

Irudia 8: Helbideak

[8] irudian ikusi daitekeen bezala helbide logikoak ere balio berdinak hartuko dituzte.

4.1.2 MMU

Lehen esan dugu, memoria bi zati nagusitan dagoela banatuta, frameak dauden zatia eta orritaulak dauden zatia. Hasteko orri taulen helbideratze tartea definitu behar da. Kasu honetan $0x400000$ helbidetik bukaeraraino izango da. Beraz hasieratik orri taulak hasten diren arte frameak biltegiratuko dira. Orri taula baten barruan orriak egongo dira eta orri bakoitzean frame baten helbidea. Prozesu batek bere helbideratze tarte logiko izango du, memoria fisikoaren desberdina. Memoriatik datu bat eskuratzeko helbide logikotik orri zenbakia eta desplazamendua aterako dira. Orri zenbakia orri taulan bilatuta behar dugun datuaren balioa eskuratuko dugu. Bihurketa hori egiteaz MMU-a arduratuko da. Frameak eta orriek ahalbidetzen dute, prozesu baten datu guztiak memorian ez dutela zertan bata bestearen atzean egon, baizik eta frametan sakabanatuta egon daitezke. Programa bakoitzeko datuak gordeta dauden frameak bilatzeko orriak erabiltzen dira beraz.

4.1.3 PCB

PCB-a egitura ezaguna daukagu dagoeneko, aurreko bi ataletan erabili delako. Atal honetan ere osagai honek aldaketak jasango ditu. Hasteko *mm* (memory managament) egitura gehituko zaio, egitura honen helburua *.text*, *.data* eta *pgb* balioak gordetzea da. *.text* prozesuan aginduak hasten diren helbidea gordetzen du, *.data*-k aldiz datuak noiz hasten diren eta *pgb*-k ordea orri taularen helbide fisikoa. Horretaz gain lehen ausaz

jartzen ziren *vruntime* eta *rtime* balioak orain programaren zikloen araberakoak izango dira. Programaren ziklo kopurua kontatzeko komando bakoitzari ziklo kopuru bat esleitu zaio, eta komandoak serialki exekutatzen direla kontuan hartu da.

- ld eta st komandoak: 7 ziklo [B D Ir AM M M Id]
- add komandoa 5 ziklo [B D Ir A Id]
- exit komandoa: 3 ziklo

4.1.4 CPU

CPU-ra era hainbat osagai berri gehotu behar dira, hala nola, PC, IR eta $PTBR$.

- PC (program counter): Uneko aginduaren helbide logikoa gordeko du.
- IR (Instruction Register): Erregistro honetan uneko agindua gordeko da.
- PTBR (Page Table Base Register): Orri taularen helbidea gordeko du.

4.2 Funtzionamendua

Sistemaren memoriaren oinarria aztertu eta gero bere funtzionamendua azalduko da hemen sistema guztiarekin bateratuta. Lehenik eta behin exekutatuko diren prozesuen kodean sortuko da, horretarako emandako programa bat erabiliko da, *prometheus*. Programak ld, add, st eta exit komandoetaz osatutako programak sortuko ditu eta *.elf* fitxategietan gordeko ditu. Honako itxura izango dute fitxategiek.

```
.text 000000
.data 000014
09000028
0A00002C
2B9A0000
1B000030
F0000000
FFFFFFF9
00000087
FFFFFFF2
FFFFFF94
00000031
00000015
000000A2
FFFFFF53
0000005A
FFFFFF73
```

Beharrezko fitxategiak sortu eta gero, sistemako loaderrak, fitxategi kopuru bat irakurriko ditu prozesu kopuruaren arabera, eta datu bakoitza memorian gordetzen joango da. Programa bateko datuak sartzeko lehenengo datu kopurua kalkulatuko du eta horren arabera, memoria guztian zehar aurkitzen duen, programa sartzen den, lehenengo lekuan gordeko ditu datu guztiak, first match. Hau egiteko linked list batean libre dauden tokien helbidea eta leku kopurua gordeko dira. Loaderrean, prozesubakoitzaren datuak memorian gordetzeaz aparte, prozesua, daukan *vruntime*-aren arabera gordeko du coren baten itxarote ilaran.

Prozesuak gorde eta gero, schedulerrak datuak prozesatzen hasiko da. Lehenengo prozesuko bakoitzeko PC-ak daukan helbidea MMU-an sartuko da memoria nagusitik

behar den agindua lortzeko. Ondoren aginduaren arabera datua memorian gorde edo memoriatik eskuratu behar da. kasu hauetan azken 6 digituak erabiliko dira helbide bezala MMU-an sartzeko eta datua eskuratzeko edo gordetzeko. Memoriatik kargatutako datuak erregistro batean gordeko dira, prozesu bakoitzak 16 erregistro izango ditu datuak gordetzeko. Erregistro hauek PC-aren balioarekin batera ere *PCB*-an gorde behar dira, prozesua exekuziotik atera behar badu bukatu gabe, gero toki berdinean jarrai dezan, datu berdinekin. Prozesu bat exekuziotik aterako da baldin eta *quantum*-a bukatuta edo *exit* agindua exekutatzen bada.

5 Ondorioak

Proiektuaren bidez sistema eragile batek dituen oinarrizko ezaugarriak eta hauen funtzionamendua praktikoki aztertu ahal izan dut. Askotan nahiz eta fase bakoitzean egin beharrekoa ulertu, C lengoaiak hainbat traba jarri dizkit, nire maila baxua zela eta. Izan ere proiektu honi esker, neukan C lengoaian idazteko maila nabarmen igo dela uste dut. Proiektuan hainbat funtzio sartzeko denbora gabe geratu naiz esaterako TLB-a, MMU-arekin erabiltzeko, batez ere proiektuko beste zati batzuetan denbora gehiegi eman dudalako. Bigarren fasean core kopurua bat baino handiagoa zenean prozesuak hauetako bakoitzeko zuhaitzetan sartzea asko kostatu zitzaidan. Azkenik loaderrean komando bakoitza interpretatzeko hainbat funtzio egin ditut eta fitxategiak irakurtzeko orduan arazoak sortu zaizkit. Arazo hauek izan arren pozik nago lortutako emaitzarekin.

Bibliografia

[1] Unai Fernandez. *SESO*. URL: https://github.com/UnaiFernandez/SE_PRAKTIKA.