Ingeniaritza Informatikoko Gradua

3.Zatia : Memoriaren Kudetzailea

Mikel Laorden Gonzalo

Sistema Eragileak

Irakaslea: Ander Soraluze Irureta

Inplementazioaren fitxategiak
https://github.com/mikelvin/SE_Proiektua

Aurkibidea:

1. Sarrera

Zati honen bitartez diseinatzen ari garen kernel-aren azken funtzionalitateak gehituko dira. Funtzionalitateak, memoriaren simulazioa, memoria birtualaren kudeaketa, prozesuen karga memorian eta prozesu hauen exekuzioan datza. Aurretik aipatutako funtzionalitateak gehitzeko "loader", "memory", "executor" eta moduluak gehituko dira, prozesuak kargatu, memoria kudeatu eta prozesuak exekutatuko dituztenak hurrenez hurren.

Fase hau 2 zatitan banatu da. Alde batetik, memoriaren simulazioa, honen birtualizazioa eta kudeaketa, prozesuak kargatzea ahalbidetzen duen datu egitura berrien sormena eta jada existititzen diren datu egituren moldaketa burutu da. Beste aldetik, prozesuen exekuzioa ahalbidetuko duen modulua eta beharrezkoak diren datu egituren moldaketak. Dokumentu honetan bi zati hauen ebazpena batera aurkeztuko da.

2. Soluzioaren diseinua:

Atal honetan, proiektuaren 3. zati honen inplementazioa aurrera eramateko hartu diren erabakiak eta burututako diseinua azalduko da.

2.1 Memoria simulazioa eta honen kudeaketa

Memoriaren kudeaketa inplementatzeko hurrengo erabakiak hartu dira. Erabaki hauek memoriaren kudeaketa aurrera eramaten duen modulu batean inplementatu dira.

Memoria fisikoaren diseinua:

Sistemak, definizioz, 24 bit-eko helbide bus bat eta 4 byteko (hitz bateko) datu bus bat izango du. Helbide bakoitzean byte bat gordeko da. Hortaz, helbide bat atzitzerako unean, helbide horretan dagoen bytea eta honen alboan dauden beste 3 byte gehiago lortuko dira datu busean. Hau da, memoria 4 byteko blokeetan atzituko da beti. Lortutako byteak, jasotako helbidearen azkenengo 2 bit-ak aldatuta lortu daitezkeen helbide-konbinazio guztieen byte-ak izango dira.

Aurreko datuak kontuan izanda sistemak totalean 2^{24} helbide, hau da, 16777216 helbide fisiko izango ditu. Ondorioz, memoriak 16777216 byteko ala 16MB-eko tamaina fisiko erabilgarri izango du.

Memoriaren kudeaketaren inguruan, lehenengo 4MB-ak erreserbatuta egongo dira hau kernel-aren espazioa izango baita, hau da, 0x000000 helbidetik 0x3FFFFF helbidera. Kernel espazio honetan, inplementatutako orrikatze sistemaren orri taulak gordeko dira. Bestetik, geratzen diren beste 12MB-ak, hau da , 0x400000 helbidetik 0x4FFFFF helbidera, erabiltzailearen espaziokoak izango dira. Erabiltzailearen espazioa orrikatuko da eta orri bakoitzean exekutatuko diren prozesuen informazioa gordeko da.

Orrikapen hau, memoria birtuala inplementatu ahal izateko definitu da. Orriaren tamainaren erabakia ez da arrazoi sendo baten atzetik artu. Era sinple batean, tamaina hau 256 bytekoa izatea egokiena izango zela hauteman eta ondorioz, erabaki hau hartu da.

Memoria birtualaren diseinua:

Aurretik aipatu bezala, orrikapena inplementatuko da. Hau sistemak memoria birtuala onartzeko sortu da.

Orrikapenaren inguruan, orri bakoitzaren tamaina definitzeke zegoen eta hortaz, proiektu honetan 256 bytetan finkatu da. Hau da, helbide birtualaren lehenengo LSB 8 bit-ak orriaren barneko helbidea adieraziko dute. Beste MSB 16 bit-ak aldiz orriaren zenbakia adieraziko dute.

Totalean, erabiltzailearen espazioa "memoria erabilgarri" bezala tratatzen badugu, orduan, 256 byteko 49152 orri fisiko definitu ahal izango dira. Orri birtualen inguruan aldiz, 2^{16} orri birtual ezberdin atzitu ahal izango dira.

Orri taularen inguruan, aurretik aipatu bezala, kernel espazioan kokatuko da. Proiektu honetan, memoria erreserbatu eta orri taula bat definitzerako unean, honek 2 zati izango ditu: Orriaren lehenengo 4 byteak, orri taula horren tamaina adieraziko du. Hau da, lehenengo lau byte horiek, orri taula horretan zehazten diren orrien kopurua adieraziko dute. Era honetan, sistemak taula horren tamainatik kanpo dagoen orriren bat atzitzen saiatzen baldin bada, sistemak errore bat kanporatuko du memoria atzipen ilegalak saihestuz. Lau byte hauen ondoren, orri bakoitzeko, beste 4 byte zehaztuko dira. Byte hauek, orri bakoitzaren hasiera zehazten duen helbide fisikoa adieraziko dute.

Memoria libre zein okupatuaren kudeaketa:

Erabiltzen ari den memoria kudeatzeko eta libre dagoen memoria erreserbatu ahal izateko, memoria-bloke libreen lista bat erabili da. Lista honetan, libre dauden memoria sekzioak agertuko dira. Sistemak prozesu bat memorian kargatu nahi izatekotan, lista honetara jo egingo du libre dagoen memoria zati baten bila. Behin memoria librea topatuta, hau erreserbatuko du memoria-bloke libreen listatik memoria zati hori erauziz eta lista eguneratuz.

Memoria erreserbatzen den moduan, hau ere askatu daiteke. Hau, adibidez, prozesu baten exekuzioa amaitzen denean gerta daiteke. Hau lortzeko, memoria-bloke libreen listan askatutako memoria barneratzen duen bloke bat gehitzen burutuko da.

Azkenik, bloke hauek fusionatzen dituen sistema bat inplementatu da. Sistema honen bitartez, bata besteari lotuta dauden bloke libreak, bloke libre handiago batean fusionatzen ditu. Aldiz, ez da memoria beraren desfragmentazioa burutzen duen algoritmorik inplementatu honek dakarren inplementazio kostuaren ondorioz.

Memory Management Unit-aren (MMU) diseinua:

CPU hari bakoitzean MMU bat gehitu da. MMU bakoitzak TLB (Translation Lookaside Block) bat izango du. Egitura hau memoria birtual eta fisikoaren arteko mapaketak kontsumitzen duten denbora murriztea du helburu. Mapaketa hau, beti memoria fisikoan dagoen orri taula atzitu beharrean, helbide birtualari dagokion mapaketa cache antzeko datu egitura batean (TLB-an) gordeko du. Era honetan, bakarrik behin memoria fisikoa atzitu behar izango da, ondoren bihurketa guztiak MMU-a bitartez burutuko dira.

TLB-a diseinatzeko hash-map baten antzeko datu egitura bat inplementatu da. Honen tamaina 64 sarreretara mugatu da, hau da, aldi berean bakarrik 64 helbide birtual-fisiko mapaketa gorde ahal izango dira.

2.2 Loaderra

Fase honetan prozesuak kargatzen dituen sistema bat sortu behar izan da. Hau, karpeta zehatz batean ("`LOAD_DIR`" konstantean definitutako helbideko karpetan zehazki) eta ".elf" formatuan dauden prozesuen deskribapen fitxategiak kargatzen burutu da.

Loader-ak kargatu behar duen prozesuarentzako, memoria moduluaren bitartez, beharrezkoa duen memoria erreserbatuko du. Ondoren, erreserbatutako memoria horretan kargatu beharreko prozesuaren edukia kargatuko du.

Gainera, kargatutako prezesua kontrolatzeko PCB bat sortu eta hau scheduler-aren egituran gehituko du.

2.3 Executor

Loader bitartez kargatzen diren prozesuak exekutatzeko motorra sortu da. Hau, hurrengo atal batean azalduko diren erregistro ugari CPU hariei gehitzen lortu da. Gainera, erregistro hauetan dauden balioak interpretatzen ala exekutatzen dituen sistema ere garatu da.

Exekuzio sistema, memoria moduluan definitu diren funtzioak zein datu egiturak (MMU) erabiliko ditu memoriaren gaineko irakurketak eta idazketak burutzeko.

3. Soluzioaren inplementazioa:

Aurretik aipatutako erabakiak hurrengo puntuetan azalduko den inplementazioaren bitartez aurrera eraman dira.

3.1 Egindako aldaketak

Soluzio berria inplementatzeko jada existitzen zen sisteman aldaketa batzuk gehitu dira. Aldaketa hauek, datu egitura ezberdinetan elementuak eta funtzionalitate berriak gehitzean oinarritu dira.

3.1.1 CPU egituran egindako aldaketak:

Programa baten exekuzioa burutu ahal izateko CPU hari bakoitza kontrolatzen duen egituran adaketak burutu behar izan dira. Aldaketa hauek hurrengo puntuetan azalduko dira. Hasteko, aldaketak hurrengo fitxategietan aurkituko dira:

- **"/include/mykernel/cpu_data_s.h"**: Core hariaren datu egitura aldatu egin da exekuxiorako beharrezkoak diren erregistroak gordetzeko
- **"/src/datu_egiturak/cpu_fun.c"**: Hasieraketa funtzioei aldatutako datu egitura era egokian hasieratzeko beharrezkoak diren aldaketak egin dira

Jada aipatu den bezala core haria kontrolatzen duen datu egituran (`core_hari_s` struct-ean) hurrengo parametroak gehitu dira:

- `uint32_t` **`PC`:** Program Counter. Une horretan exekutatzen ari den aginduaren helbidea gordeko den erregistroa edo aldagaia.
- `uint32_t` **`IR`:** Instruction Register. Une horretan exekutatzen ari den agindua gordeko den erregistroa.
- `uint32_t` **`PTBR`:** Page Table Base Register. Une horretan exekutatzen ari den prozesuaren orri taularen helbidea. Era honetan, core hariak exekutatu behar ddituen helbide birtualak MMU-ari eskaini eta honek helbide birtualak helbide fisikoetara itzuli ahal izango ditu.
- `int32_t` **`R[`**`CPU_HARI_REG_KOP`**`]`:** Core hariaren erregistro laguntzaileak. 16 erregistro ezberdinez osatuak egongo dira. Balio hau, `CPU_HARI_REG_KOP` konstantearen bitartez definituko da.
- `int32_t` **`cc`:** Konparaketa eragiketek aktibatuko duten flag-a.
- `struct mmu` **`mem_manag_unit`:** Core hariaren MMU-a. Honen bitartez, hariak memoria atzitu ahal izango du. Gainera, helbide birtualak era erraz batean mapatzea ahalbidetuko du. Datu egitura hau, hurrengo "3.2 Memoria modulua" atalean tratatuko da.

Bestetik, datu egituran egindako aldaketekin bat, "`hari_t_init`", "`core_t_init`" eta "`cpu_t_init`" funtzioetan, "`struct physycal_memory * ps`" parametro berria gehitu da. Era

honetan, core eta CPU bakoitzak duen CPU hari bakoitzean MMU egitura hasieratu ahal egingo da.

Azkenik, lenengo atalean azaldutako CPU, core eta core hariak osatzen duten egituratze ala kokatze habiatua mantendu egin da. Hau da, CPU batek, core-en array bat izango du bere barnean. Era berean, core bakoitzak core hari-en beste array bat izango du.

3.1.2 PCB egituran egindako aldaketak:

Process Control Block egituran ere aldaketak buru behar izan dira. Aldaketa hauen bitartez, PCB-ak exekuzio motorrerako eta prozesu bakoitzaren memoriaren erreserba eta kudeaketa sistemarako prestatu dira.

- **"/include/data_structures/pcb_s.h"**: Fitxategi honetan, lehenik "`pcb_t`" datu egiturari aldaketak egin zaizkio. Bestetik, beste 2 datu egitura berri gehitu dira: "`cpu_snapshot`" eta "`mm`". Hauek, prozesu baten CPU hariaren azken egoera gorde eta honek erreserbatutako memoria atzitzea ahalbidetuko dute hurrenez hurren.
- **"/src/datu_egiturak/pcb_s.c"**: Aurretik aipatutako datu egiturak hasieratzeko funtzioak sortu eta eguneratu dira. Funtzio hauek "`mm_init`", "`cpu_snapshot_init`" eta "`pcb_init`" dira, "`mm`", "`cpu_snapshot`" eta "`pcb_t`" datu egiturak hasieratuko dituzte hurrenez hurren. Bestetik, "`pcb_destroy`" funtzioa ere sortu da, PCB batek okupatzen duen memoria askatzen duena.

Aurretik aipatu bezala, "`cpu_snapshot`" datu egitura gehitu da. Egitura honek, izenak dioen bezala, CPU hari baten argazki bat bezala jokatzen du. Honen bitartez, exekuzioan dagoen prozesu batek kontextu aldaketa egiterakoan, une horretan hau exekutatzen ari duen hariaren egoera gorde ahal izango du. Era honetan, ondoren, prozesua berriz exekuziorako kargatzen denean, CPU hariaren aurreko egoera berreskuratu ahal izango du aurreko eragiketak mantenduz. Ondorioz, datu egitura horrek, CPU hariaren erregistroak gordetzeko sarrerak izango ditu, hala nola: "`PC`", "`IR`", "`R[CPU_HARI_REG_KOP]`" erregistroen array-a eta "`cc`" erregistroak.

Bestetik "`mm`" datu egitura, prozesu baten "Memory Management" ("mm" izenaren arrazoia) ala memoria kudeaketa burutuko duen struct-a izango da. Datu egitura honetan, prozesuaren PTBR-a ("Page Table Base Register" ala orri taularen helbidea) eta kode segmentuaren zein datu segmentuaren hasierako helbide birtualak gordeko ditu. Prozesuak, datu egitura honekin, honek erreserbatutako memoria zatia atzitu ahal izango du.

Hortaz, prozesu baten egoera osotasunean gordetzeko "`pcb_t`" datu egituran "`cpu_snapshot`" eta "`mm`" bana gehitu dira.

3.1.3 Linked List funtzionaltasunari egindako aldaketak:

Memoriaren kudeaketa errazagoa izateko, hurrengo funtzio berriak gehitu dira. Alde batetik funtzio propio batzuk sortu dira, hauek hurrengo fitxategietan aurkituak izan ahalko dira:

- **"/include/data_structures/lnklist_s.h"**: "`__search_procedure`", "`lnklst_LFRL_popFirstMatchFromRear`" eta "`lnklst_LFRL_peekFirstMatchFromRear`" funtzioen deskribapena egin da. Gainera, kodea sinplifikatzeko, "`__unlink_procedure_of_current`" funtzioa ere definitu da.
- **"/src/datu_egiturak/lnklist_LFRL.c"**: Aurretik aipatutako funtzioen inplementazioa burutu da.

Hurengo puntuetan funtzio bakoitzaren gaineko funtzionalitatea azalduko da sistemaren ikuspegi orokor bat izateko:

- "`int __unlink_procedure_of_current(lnklist_LFRL * p_list, struct lnk_node * prev, struct lnk_node * current)`": Funtzio honi, lista baten nodo bat listatik erauzten du. Hau lortzeko erauzi nahi den nodoa eta honen aurreko nodoa eskaini behar da. Erauzketa listaren integritatea mantendu egingo du.
- "`int __search_procedure(lnklist_LFRL * p_list,struct lnk_node ** p_prev, struct lnk_node ** p_current, int (* predicate) (void * data, void * args), void * s_args)`": Parametro bitartez pasatutako "`predicate`" funtzioa exekutatuko du listan dauden nodo guztien datuak eta "`s_args`" aldagaieko datuak parametroz eskainiz. Hau, "`predicate`" funtzioak, nodo bateko datu zehatz batekin "1" balioa itzuli arte, ala beste era batean esanda, "match" bat lortu arte egingo da. Behin "match" bat lortuta, "1" balioa bueltatzeaz gain, datu hori duren nodoaren helbidea "`p_current`" helbideko aldagaian eta nodo honen aurreko nodoa "`p_prev`" helbideko aldagaian zehaztuko ditu. "Match"-k topatu ezean, "0" balioa bueltatuko du.
- "`void * lnklst_LFRL_peekFirstMatchFromRear(lnklist_LFRL * p_list, int (* predicate) (void * data, void * args), void * s_args)`": "`predicate`" funtzioaren emaitza "1" izatea lortzen duen lehenengo nodoaren datua bueltatzen du hau listatik kanporatu gabe.
- "`void * lnklst_LFRL_popFirstMatchFromRear(lnklist_LFRL * p_list, int (* predicate) (void * data, void * args), void * s_args)`": "`lnklst_LFRL_peekFirstMatchFromRear`" funtzioa bezala baina topatutako elementua listatik kanporatzen du .

Azkenik, lista estekatuaren funtzionalitate aurreratuak inplementatzeko "**src/datu_egiturak/linklist_LFRL_utils.c**" eta "**include/data_structures/lnklist_functionalities_s.h**" fitxategietan definitzen diren funtzioak sortu dira. Funtzio hauek lista estekatu bat ordenatzea ahalbidetzen duten metodoak dira. Hainbat ordenatze algoritmo existitzen direnez proiektu honetan integratutakoa https://www.geeksforgeeks.org/merge-sort-for-linked-list/ webguneetik lortu da.

3.1.4 Scheduler-ari egindako aldaketak:

Scheduler-ak, "`sched_execute`" funtzio nagusia deitzeko momentuan, prozesu bat amaitu den ala ez zehazteko, prozesuaren egoera begiratuko du. Prezesu baten egoera "`__sched_updateRunning_pcb`" funtzioa bitartez eguneratuko da. Funtzio honek exekutatzen hari den CPU hari-aren IR ala "Instruction Register" agindua begiratuko du. Bertan dagoen dagoen agindua "`EXIT_CODE`" agindua izatekotan, orduan prozesua maitutzat hartuko du eta

prezesuari "`PR_FINI`" egoera esleituko dio. Bestela, exekutatzen ari den prozesuari "`PR_EXEC`" egoera esleituko dio, prozesua exekuzioan dagoela adierazteko.

CPU-etan prozesuen kontextu aldaketa aurrera eramateko hurrengo funtzioak sortu dira:

- "`__disp_load`": Parametro bitartez jasotako PCB baten "`struct cpu_snapshot s`" CPU argazkia beste parametro baten bidez jasotako CPU hari-an kargatzen du.
- "`__disp_unload`": "`__disp_load`"-en kontrakoa. Cpu hari baten egoera, parametro bitartez jasotako PCB-aren "`struct cpu_snapshot s`" CPU hari argazkian gordetzen du. Gainera, prozesuaren egoera "`PR_EXEC`"-era aldatuko du.
- "`disp_Terminate`": PCB bat eskainiz, PCB hau kontrolatzen duen prozesuak erreserbatutako memoria askatu egiten du hurrengo Oinarrizko Memoria Modulua atalean azaldutako "`mmu_free`" funtzioaren bitartez.
- "`disp_Dispatch`": Cpu hari batean bi prozesuen arteko kontextu aldaketa burutzen duen funzioa. Honi, 2 PCB eskainiz, lehenegoan "`__disp_unload`" funtzioa erabiliz CPU hariaren egoera egorde eta "`__disp_load`" erabiliz bigarren prozesua kargatzen du. Gainera MMU-aren TLB-a berhasieratzea erabingo du "`tlb_flush`" erabiliz.

3.2 Memoria modulua

Memoria modulua aurrera eramateko hainbat azpisistema kontuan behar izan dira. Azpisistema hauek, era orokorran, "memory_s.h" eta "memory_s.c" fitxategietan definitu dira:

- **"/include/data_manager/memory_s.h"**: Memoria fisikoa kudeatzen duen funtzioen deskribapena edo definizioa. Memoria era egokian atzitzeaz arduratzen da. Memoria era egokian erreserbatzeko eta askatzeko funtzioak aurkitzen dira. Gainera MMU gailuaren funtzionalitateak simulatzen dituen funtzioak ere definituta daude.
- **"/src/data_manager/memory_s.c"**: Aurretik aipatutako funtzionalitateak inplementatzen dituen kodea aurkitu daiteke.

Memoria modulua azaltzeko hau 3 zatitan banatu da. Alde batetik oinarrizko memoria modulua, memoria fisiko libre zein okupatuaren kudeaketa egiten duena eta memoriaren oinarrizko atzipena ahalbidetzen duena. Ondoren orrikapen sistemaren integritatea bermatzen duen sistema. Azkenik MMU-aren eta honek duen TLB-aren funtzionalitatea.

Funtzio hauek, "loader" eta "executor" moduluek erabiliko dituzte. Era honetan, memoriaren kudeaketa mudulu zehatz honetan bakarrik tratauko da, erantzunkizun hau beste moduluetatik kenduz.

Oinarrizko Memoria Modulua:

Zati honetan memoria librearen eta okupatuaren kudeaketa burutzen duten eta memoria atzitzea ahalbidetzen duten oinarrizko funtzio eta datu egiturak azalduko dira.

Datu egiturak:

Oinarrizko memoria modulua ala sistema aurrera eramateko hurrengo datu egiturak sortu dira:

- **`struct physycal_memory :`** Oinarrizko memoria egitura. Egitura honen bitartez definitu beharreko memoria espazio osoa definituko eta errepresentatuko da. Bertan gure prozesuen edukiak eta orri taulak gordeko dira. Struct hau, 2 sarrera nagusi izango ditu: "`char * memory`" array bat eta "`struct free_block`"-ez osatuta egongo den "`lnklist_LFRL free_blocks`" lista estekatu bat. Sarrera hauetan, memoria bera eta memorian dauden bloke edo zati libreak gordeko dira hurrenez hurren.
- **`struct free_block :`** Memoria bloke libre bat erregistratzen duen egitura. Hau 2 sarreraz osatuta egongo da, hau da, memoria bloke librearen hasierako ("`uint32_t start_adress`") eta amaierako helbideak ("`uint32_t end_adress`") gordeko ditu. Muga helbide hauek memoria bloke librearen parte izango dira.
- **`struct free_block_search_args :`** Struct hau egitura lagungarri bat da definitutako "**`free_block`**" bilaketa funtzio batzuei beharrezkoak diren parametroak eskeintzeko. Funtzio hauek, "**`struct physycal_memory`**"-ko "**`lnklist_LFRL free_blocks`**" listan aurkitzen diren blokeen artetik, "**`free_block_search_args`**" egitura horretan aurkitzen diren parametroak kontuan izanda, egokia den "**`struct free_block`**" bat bilatzeko erabiliko da. Struct-ak 3 parametro ditu: "**`start_limit_adress`**", "**`end_limit_adress`**" eta "**`min_adress_kont`**". Hauek bilatu behar den blokeak izan behar duen hasierako helbidea, amaierako helbidea eta tamaina minimoa adieraziko dute hurrenez hurren.

Funtzionalitateak:

- "**`int pmemo_init(struct physycal_memory * pm)`**": Memoria fisikoa kudeatzen duen egitura hasieratzen du. Era berean, memoria bloke libreen lista ere hasieratzen du, memoria guztia barneratzen duen memoria bloke bat listan gehituz.
- "**`uint32_t pm_read_word(struct physycal_memory * ps, uint32_t phys_adress)`**": Memoria 4 byteko blokeetan irakurtzen du. Zehaztutako helbidearen lehenengo 2 LSB bit-ak zerora jarri ostean lortutako helbidetik hasi egingo da byte-ak lortzen.
- "**`int pm_write_word(struct physycal_memory * ps, uint32_t phys_adress, uint32_t word)`**". Zehaztutako helbidea habiapuntutzat hartuta, 4 byte idazten ditu.
- "**`int __pm_memory_allocation_procedure(struct physycal_memory * ps, struct free_block_search_args * request, uint32_t * ret_adress)`**": Erreserba eskaera bat emanda, erreserba hori egin ahal den bueltatzen du. Erreserba egin ahal izatekotan, erreserba burutzen du eta honen hasierako helbidea "`ret_adress`" helbidean dagoen aldagaian gordetzen du.

- "`int pm_memory_allocation_request(..., uint32_t * ret_adress)`": Funtzio hau, "`__pm_memory_allocation_procedure`" funtzioa deitzen du. Honek, memoriaren erreserban arazoak izatekotan, beste memoria erreserba eskaera bat egin baino lehen, "`__optimize_free_block_list`" funtzioa deituko du hurrengo deian memoria bloke egoki bat lortzeko helburuarekin. Erreserbatutako memoriaren lehenengo helbidea "`ret_adress`" aldagaian gordeko du. Funtzio honen sinadura laburtu da, hau, "`__pm_memory_allocation_procedure`" funtzioaren sinaduraren berdina da.
- "`int __search_space4free(void * free_block, void * search_args)`": Funtzio laguntzailea, "`lnklst_LFRL_peekFirstMatchFromRear`" eta "`lnklst_LFRL_popFirstMatchFromRear`" funtzioak deitzeko erabili izango da. "`free_block`" eta "`free_block_search_args`" bana emanda, bloke libre hori eskaerarekin bat badatorren adieraziko du. Honen bitartez, eskaera zehatz baterako, eskaera batetzen duen lehenengo bloke librea topatu ahal izango da.
- "`int __optimize_free_block_list(struct physycal_memory * ps)`": Sistemaren memoria bloke libreak ordenatzeko funtzioa. Honen bitartez, bloke libreak ordenatu eta fusionatzen dira. Era honetan, bloke libre fragmentatuak bloke handiagoetan transformatuko dira, memoria espazio handiago bat segituan erreserbatzea ahalbidetuz. Funtzio hau, sistemak memoria erreserba bat egiten zailtasunak dituenean exekutatuko da bloke libreen fragmentazioaren arazoa saihesteko.

Orrikapen modulua:

Orri taulen eta frame-en kudeaketa burutzen duen funtzioanalitate multzoa. Orri taulak eraiki eta suntzitzen ditu. Gainera frame-ak ere sortu eta borratzen ditu beti memoriaren integritatea mantenduz.

Datu egiturak:

"Oinarrizko Memoria Modulua" ataleko "Datu egiturak" azpi-atalean aurkezten diren datu egitura berdinak erabiliko dira.

Funtzionalitateak:

"Oinarrizko Memoria Modulua" ataleko "Funtzionalitateak" azpi-atalean aurkezten diren oinarrizko memoria kudeaketa funtzioak erabiliz, hurrengo funtzio espezifikoagoak sortu dira:

- "`int pm_m_frame_free(struct physycal_memory * ps, uint32_t phys_adress)`": Zehaztutako frame-aren hasierako helbidea emanda, honek okupatzen duen memoria askatzen du memoria bloke libre bat bloke libreen listara gehituz.
- "`int pm_m_frame_malloc(struct physycal_memory * ps, uint32_t * adress)`": Erabiltzaile espazioaren memorian 256 byteko memoria erreserba bat eskatzen da, hau da, frame baten erreserba egiten da. Espazio nahikorik egotekotan, sistemak

espazio hori erreserbatu, erreserbatutako frame-aren lehen memoria helbidea "`adress`" aldagaian txertatu eta "1" bueltatuko du. Espaziorik ez egotekotan "0" bueltatuko du.

- "`int pm_m_frame_malloc_bulk(struct physycal_memory * ps, uint32_t frame_adress_array[], int frame_kop)`": "`frame_kop`" aldagaian eskatzen diren frame kopurua erreserbatzen du trantsakzio baten antzera. Hau da, eskatzen diren frame guztien erreserba egiten du ala ez du frame bat ere ez erreserbatzen. Erreserba era egokian egitekotan, funtzioak "1" bueltatuko du. Bestela "0" bueltatuko da.
- "`int pm_pt_page_table_malloc(struct physycal_memory * ps, uint32_t * adress, uint32_t entries)`": Funtzio honen bitartez, kernel espazioan "`entries`" tamaina duen orri bat erreserbatzen da. Prozesu honek beharrezkoak diren orrien erreserba ere egiten du. Erreserba hau "`pm_m_frame_malloc_bulk`" funtzioa laguntzailearen bitartez egiten da. Kernel zein erabiltzaile espazioan lekurik ez egotekotan, sistemak errore bat jaurtiko du eta programa amaituko da.
- "`int pm_pt_page_table_free(struct physycal_memory * ps, uint32_t adress)`": Orri taula baten oinarri helbidea emanda, orri taulan erregistratutako frame-en eta orri taula bera erreserbatutako espazioa askatzen du.
- "`int mmu_malloc(struct mmu * target_mmu, uint32_t * ptbr, int32_t word_kop)`": Nahi diren 4 bytetako hitz kopurua adieraziz, sistemak orri taula bat sortu eta hitz horiek gordetzeko behar den espazioa erreserbatzen du. Honekin, orri taula zein frame-ak erreserbatzen eta sortzen dira. Funtzio hau "`pm_pt_page_table_malloc`" funtzioa erabiliko du.
- "`int mmu_free(struct mmu * target_mmu, uint32_t ptbr)`": "`pm_pt_page_table_free`" funtzioaren alias bat. Kasu honetan, "`mmu`" datu egituraren bitartez deitzen da.

Memory Managent Unit (MMU) modulua

Helbide birtual eta fisikoen arteko mapaketa burutzen duen sistema.

Datu egiturak:

- "`struct pte`": Page Table Entry izenarekin ezagutua. TLB-aren edukiak izango dira. Honen bitartez, helbide birtual bati dagokion helbide fisikoaren arteko mapaketa bat gorde ahal izango da. Egiturak 3 balio nagusi ditu: "**`virtual_adress`**", "**`physycal_adress`**" eta "**`active`**", helbide virtual bat, helbide birtualari dagokion helbide fisikoa eta PTE sarrera hori baliozkoa den ala ez adieraziko duten balioak izango dira hurrenez hurren.
- "`struct mmu`": Memory Management Unit-a (MMU-a) errepresentatzen duen datu egitura. Struct honen bitartez, memoria fisikoa atzitu eta TLB bat erabili ahal izango da. Ondorioz, egiturak hurrengo balioez osatuta egongo da: "**`tlb_hashArr`**",

"`tlb_max_space`" eta "`ps`". Hauek "`struct pte`"-ez osatutako array bat (hau da, TLB-a), TLB-aren sarrera kopuru maximoa ala "`tlb_hashArr`"-aren tamaina eta memoria fisikoa kontrolatzen duen datu egiturari erreferentzia izango dira hurrenez hurren.

Funtzionalitateak:

- "`int mmu_init(struct mmu * target_mmu, struct physycal_memory * pm, int max_tlb_space)`": "`struct mmu`" egitura hasieratzen duen funtzioa.
- "`int tlb_get_match(struct mmu * p_mmu, struct pte ** match_pte, uint32_t ptbr, uint32_t virt_adress)`": MMU-an dagoen TLB-a bitartez helbide birtual baten helbide fisikoa lortzen saiatzen da. Hau topatzekotan, helbide birtualari dagokion helbide fisikoa duen "`pte`"-a "`match_pte`" aldagaiaren bitartez eskaini eta "1" balioa bueltatzen du. Helbide birtual honekin bat egiten ez duen PTE sarrera aktiborik aurkitzen ez badu, orduan "0" bueltatuko du.
- "`int tlb_add_entry(struct mmu * p_mmu, uint32_t ptbr, uint32_t virt_adr, uint32_t phys_adr)`": TLB-an PTE sarrera berri eta erabilgarri bat gehitzen du, parametro bitartez pasatutako helbide fisikoa eta birtuala mapatuko duena. TLB-ak hash map mugatu baten antzera jokatzen duenez, mapaketa berri bat gehitzeak beste mapaketa erabilgarri bat TLB-tik kanporatzea eragin dezake.
- "`int  mmu_resolve_frame_rootadr_from_memo(struct mmu * p_mmu, uint32_t * frame_adress, uint32_t PTBR, uint32_t virt_adress)`": Helbide birtual baten helbide fisikoa lortzen du memoria fisikoan gordeta dagoen orri taula atzituz. Hau, helbide birtuala bera eta helbideari dagokion "PTBR"-a eskainiz burutzen da. Helbide fisikoa "`frame_adress`" aldagaiak duen helbidean gordeko da. Prozesu hau errore bat eman dezake jasotako helbide birtuala orri taularen mugatik kanpo baldin badago.
- "`uint32_t mmu_resolve(struct mmu * p_mmu, uint32_t PTBR, uint32_t virt_adress)`": TLB-a erabiliz helbide birtual baten helbide fisikoa lortzen saiatzen da. TLB-an "Match" bat aurkitzekotan orduan TLB-arekin helbide fisikoa lortzen da. Adiz, "match"-ik ez aurkitzekotan, "`mmu_resolve_frame_rootadr_from_memo`" funtzioaren bitartez helbide fisikoa lortu eta hau TLB-an "`tlb_add_entry`" funtzioaren bitartez gehitzen du. Azkenean, helbide birtual egoki bat eskainiz gero, helbide birtual honen helbide fisikoa bueltatuko du parametro bitartez. Helbide birtuala ez egokia baldin bada, sistemak salbuespen bat jaurti eta programa geldiaraziko du legez kontrako memoria atzipenak saihesteko.
- "`int tlb_flush(struct mmu * p_mmu)`": Zehaztutako MMU-aren barnean dauden TLB-ko sarrera guztiak (PTE guztiak) ezgaitzen ditu. Era honetan, MMU-ak ezin izango du TLB-a bitartez helbiderik ebatzi eta memoria fisikoa atzitu behar izango du, PTE guztien berriketa prozesua bortxatuz.

3.3 Loader Modulua:

Loder modulua aurrera eramateko hurrengo fitxategiak kontuan hartu dira:

- **"/include/data_manager/loader.h"**: Prozesuen PCB-ak sortu eta hauen edukiak memorian kargatzen dituen sistemaren funtzioak definitzen dira.
- **"/src/data_manager/loader_s.c"**: Aurretik aipatutako funtzionalitateak inplementatzen dituen kodea aurkitu daiteke.

Datu egiturak:

Prozesuen karga modulu hau hau aurrera eramateko datu egitura bakar bat sortu da.

- "`typedef struct loader_s`": Lehenengo fasean aurkeztutako "program generator"-ak zuen datu egituraren antzekoa. Kargatuko dituen prozesuen lehentasun maximoa, esleitu den azken PID-aren balioa, memoria atzizea ahalbidetuko duen "`struct mmu`" laguntzaile bat, "`sched_basic_t * sched`" scheduler-aren helbidea eta "**`unloaded_elf_list`**" izena duen kargatu gabeko prozesuen "elf" fitxategien helbideen lista.

Bestetik "`LOAD_DIR`" konstantea ere definitzen da non kargatu beharreko prozesuen karpetaren helbidea definitzen den.

Funtzionalitateak:

Fitxategi hauetan jada "process generator" moduluan agertzen ziren "`__loader_getValidPrio`" eta "`__loaden_getValidPid`" funtzioak definitzen dira, prezesu batentzako lehentasun zein PID egoki bat lortzea ahalbidetzen dutenak.

Bestetik "`__hexString2int32`" eta "`__hexChar2int4`" funtzio laguntzaileak definitzen dira, hauek irakurritako "elf" fitxategian dauden karaktere hamaseitarrak era egokian interpretatzeko sortu dira. Azkenean, ASCII-n zehaztutako kode hamaseitarra C lengoaiaren "`uint8_t`" formatura transformatzen du.

Azkenik prozesuak kargatzen dituzten funtzionalitateak ere inplementatuak izan dira. Hauek hurrengo puntuetan azaltzen dira:

- "`int loader_startup(loader_s * loader, struct physycal_memory *ps, sched_basic_t * sched, int max_prio)`": Loader-aren egitura hasieratzen du. Hau jasotako "`ps`" memoria fisikoaren erreferentzia erabilita prozesuak kargatuko dituen MMU lagungarri bat sortuko du. Gainera "`LOAD_DIR`" aldagaian dauden fitxategi guztien helbideak, "**`unloaded_elf_list`**" helbideen listan gehituko ditu.
- "`int __load_file(struct mmu * p_mmu, char * d_name, struct pcb_t * loading_pcb)`": "`d_name`" helbidean dagoen "elf" motako fitxategia kargatzen du. Bertako edukiak interpretatzen ditu eta beharrezko memoria erreserba egiten du prozesuaren edukiak memorian kargatu daitezen. Kargatutako prozesuaren datuak,

parametro bitartez jasotako "`loading_pcb`" PCB-an txertatuko ditu. Memoria erreserba parametro bitartez jasotako "`p_mmu`" MMU egituraren bitartez egingo du.

- "`int loader_loadNextProcess(loader_s * loader)`": Funtzioa deitzen den momentuan, "**`unloaded_elf_list`**" listatik helbide bat erauzi eta "`__load_file`" bitartez helbide honi dagokion "elf" fitxategian dagoen prozesua kargatzen da. Behin prozesua memorian kargatuta eta PCB-a sortuta, loader-ak PCB-a "`loader`" egiturak duen "`sched_basic_t * sched`" scheduler-ean gehitzen du. Hau da timer batek deituko duen funtzio nagusia.
- "`int loadNullPCB(struct mmu * p_mmu, struct pcb_t * loading_pcb)`": Funtzio honen bitartez EXIT agindua bakarrik duen prozesu baten PCB-a sortu eta honentzako memoria minimoa (sarrera bateko orri taula eta frame bat) erreserbatzen da. Prozesu nulu hau scheduler-ak erabiliko du PCB-en lista hutsik dagoenean.

3.4 Executor Modulua:

Exekuzio motorra eta hau inplementatzen duen modulua aurrera eramateko hurrengo fitxategiak kontuan hartu dira:

- **"/include/data_manager/executor.h"**: core hari bakoitza kargatutako instrukzioak interpretatuko dituzten funtzioak definitzen dira.
- **"/src/data_manager/executor_s.c"**: Aurretik aipatutako funtzionalitatea inplementatzen duen kodea aurkitu daiteke.

Datu egiturak:

Modulu honetan, sistemak dituen CPU egitura guztiak biltzen dituen datu egitura sortu da. Honen bitartez, exekuzio motorrak CPU-ak atzitu ahal izango ditu. Datu egitura hurrengoa da:

- "`struct executor_t`": Egitura honek, "`cpu_s`" motako array bat izango du. Gainera, beste aldagai baten bitartez, array honen tamaina gordeko da.

Datu egitura hau hasieratzeko "`int executor_init(executor_t * exec, cpu_s cpu_arr[], int cpu_arr_len)`" funtzioa sortu da. Honek, aurretik aipatutako array-a eta array honen tamaina gordetzen duen aldagaia hasieratuko du parametro bitartez eskeinitako "`cpu_arr`"-a eta "`cpu_arr_len`" erabiliz hurrenez hurren.

Funtzionalitatea:

Prozesu bat exekutatu ahal izateko, prozesuaren agindu bakoitza identifikatzen duten funtzioak sortu dira. Funtzio hauek, behin exekutatu nahi den agindua identifikatu dutela, agindua exekutatzen duen funtzio espezifikoa deituko dute. Totalean 16 exekutatu daitezke. Aginduak 3 taldetan banandu dira: Memoria atzipenaren, eragiketa logiko/aritmetiko eta programaern fluxua kontrolatzen duten aginduen arabera banatuta. Agindu hauek kudeatzeko edo identifikatzeko 3 funtzio nagusi sortu dira.

- "`void __ex_MEM(struct core_hari_s *cpu_context, uint32_t plain_command)`": Memoria atzitzen duten aginduak kontrolatzen dituen funtzioa. Agindu hauek

hurrengoak dira: LD ("`__ex_MEM_ld`") eta ST ("`__ex_MEM_st`"). Gainera, program counter-a inkrementatzen du.

- "`void __ex_ALU(struct core_hari_s *cpu_context, uint32_t plain_command)`": Funtzio honek eragiketa aritmetiko / logikoak taldekatzen ditu. Funtzioak kudetuko dituen aginduak SUB ("`__ex_arithm_add`"), ADD ("`__ex_arithm_sub`"), MUL ("`__ex_arithm_mul`"), DIV ("`__ex_arithm_div`"), AND ("`__ex_logic_and`"), OR ("`__ex_logic_or`"), XOR ("`__ex_logic_xor`"), MOV ("`__ex_arithm_mov`") eta CMP ("`__ex_arithm_cmp`") dira. Azken batean "`cpu_context`" parametroan zehaztutako core harian dauden erregistroen arteko operazioak eta operazio hauekin lortutako emaitzak eguneratzen ditu. Gainera, program counter-a inkrementatzen du.
- "`void __ex_CU(struct core_hari_s *cpu_context, uint32_t plain_command)`": Funtzio honek, programaren fluxua aldatzen duten agintuduak kudeatzen ditu. Agindu hauek hurrengoak dira: B ("`__ex_b`"), BEQ ("`__ex_beq`"), BGT ("`__ex_bgt`", BLT ("`__ex_blt`") eta EXIT ("`__ex_exit`"). Ez du program counterra (PC-a) inkrementatzen. Aginduaren arabera PC-a aldatzea eragingo du. Aipatu beharra dago, EXIT aginduaren inplementazioak ez duela koderik exekutatzen, hau da, sistema askotan existitzen den NOP agindu baten antzera jokatzen du.

Ikusi daitekeenez, funtzio hauek, CPU hari bat eta "elf" fitxategi bateko agindu bat eskainiz, dagokion taldeko aginduaren kodea exekutatuko du. Agindu bakoitzeko, c lengoaieko azpi funtzio bat sortu da egindu bakoitzaren funtzionamendua bata bestetik abstraitzeko.

Bestetik aurretik aipatutako funtzioak biltzen dituen eta agindu bakoitzeko egokia dena exekutatzen duen funtzioak definitu dira. Hauek maila altuagoko funtzioak dira:

- "`int __execute(struct core_hari_s * cpu_context, uint32_t plain_command)`": Sartutako agindua aztertu ondoren, aurretik aipatutako funtzio egokia deituko du agindu zehatz horretarako.
- "`int execute_current(struct core_hari_s * cpu_context)`": CPU hari baten PC erregistroan dagoen helbidearen agindua lortzen du MMU-a erabiliz. Ondoren, agindu hau IR erregistroan kargatu eta "`__execute`" futnzioa deitzen du.
- "`int execute_all(cpu_s cpu_arr[], int cpu_arr_len)`": Parametro bitartez jasotako CPU array-aren CPU bakoitzaren core hari bakoitzeko "`execute_current`" funtzioa exekutatuko du.
- "`void executor_exec(executor_t * exec)`": Timer-ak exekutatuko duen executor-aren funtzio nagusia. Executor egitura batean dagoen CPU array-a parametro bitartez pasatuz "`execute_all`" funtzioari deitzen zaio.

Azkenik, aginduaren byte konkretu batzuk erauztea ahalbidetzen duen "`uint32_t getNibbleRange(uint32_t item, int leftmost_nibble, int rightmost_nibble)`" funtzioa sortu da. Honen bitartez, agindua disekzionatu daiteke, aginduaren arabera, beharrezkoak diren datuak bakarrik lortuz.

4. Soluzioaren integrazioa:

Atal honetan era laburtu batean, aurretik aipatutako funtzioak sisteman nola integratu diren azalduko da.

Main programa

Hasteko beharrezkoak diren aldagaiak eta datu egiturak definituko dira:

```
loader_s my_loader;
sched_basic_t my_scheduler;
struct physycal_memory system_memory;
```

Jarraituz "loader"-a kontrolatuko duen datu egitura bat hasieratuko da:

```
loader_startup(&my_loader, &system_memory, &my_scheduler, MAX_PRIO);
```

Ondoren "memoria fisikoa" kontrolatuko duen datu egitura bat hasieratuko da:

```
pmemo_init(&system_memory);
```

Bestetik, CPU-en array-a hasieratuko da. Array hau "executor" datu egitura hasieratzeko erabiliko da:

```
struct cpu_s my_cpu_arr [CPU_KOP];
for(int i = 0;i < CPU_KOP; i++)
    cpu_t_init(&my_cpu_arr[i], CORExCPU_KOP, HARIxCORE_KOP,&system_memory);
```

```
executor_t my_exec;
executor_init(&my_exec, my_cpu_arr, CPU_KOP);
```

Azkenik, behin CPU-ak, "executor"-a eta "loader"-ak hasieratu ondoren, scheduler datu egitura hasieratuko da. Orden hau ezinbestekoa da, sheduler-ak loader-a eabiliz null prozesu bat kargatu eta sortutako CPU-etan hasieratuko duelako.

```
sched_init(&my_scheduler, &tmr1_scheduler,
    my_cpu_arr,CPU_KOP, politika);
```

5. Erabilpena:

Programa hoen funtzionalitateak frogatzeko hurrengoa kontuan izan beharko da:

Scheduler-aren egoera pantaiaratzen duen sistema:

Terminalean erlojuaren ziklo bakoitzeko scheduler-aren eta CPU hari bakoitzaren informazioa pantaiaratuko da. Hau ezagaitzeko, kodean timer bat ezgaitu behar izango da. Timer hau hurrengoa da:

```
clk_timer_init(&debug_tmr3_print_pqueue, &clkm_1,
    EGOERA_PRINT_FREQUENCY, &sched_print_sched_State, &my_scheduler, "Sched Status Print");
```

Lerro hau komentatuz gero, sistemak ez du scheduler-aren hainbeste informazio pantaiaratuko.

CPU, CORE eta HARI-en konfigurazioa:

Sistemak CORE eta HARI bana duten 2 CPU-rekin exekutatzeko konfiguratu da. Hau, kodeko "CPU_KOP", "COREXCPU_KOP" eta "HARIXCORE_KOP" preprozesagailuaren aldagaien balioak handituz aldatu daiteke. Prozesu hau programa konpilatu aurretik egin behar izango da.

Kargatu nahi diren prozesuen direktorioa aldatu:

Bi era daude:

1. **"./include/data_manager/loader.h"** fitxategian aurkitzen den "LOAD_DIR" aldagaiaren balioa aldatuz eta programa birkonpilatuz.
2. "LOAD_DIR" helbidean dauden prozesuak nahi kargatu nahi diren prozesuekin ordezkatu.

Bigarren aukerarako, entregan, "LOAD_DIR"-en balioa **"./data/programlist/"** izango da. Hau da, programa exekutatzen den direktoriotik, "**data**" eta jarratuan "**programlist**" direktorioak existitu beharko dira. Direktorio honetan kargatu nahi diren "elf" fixategiak aurkituko dira.

Konpilazio sistema:

Programa konpilatzeko linux terminal batean, proiektuaren erroan, hurengo komandoa exekutatu behar izango da:

$ make

Konpilatutako programa hurrengo fitxategian aurkituko da:

./build/my_sched.out

Exekuzio sistema:

Aurreko zatian azaldu den bezala, programak parametro bakar bat jasoko du. Balio hau zenbaki bat izango da. Zenbaki honek, inplementatutako scheduler politiketatik exekutatuko den politika zehaztuko du.