SESO

2020.eko azaroaren 15

Euskal Herriko Unibertsitatea (UPV/EHU)

Sistema Eragileak

Unai Fernandez

Gaien Aurkibidea

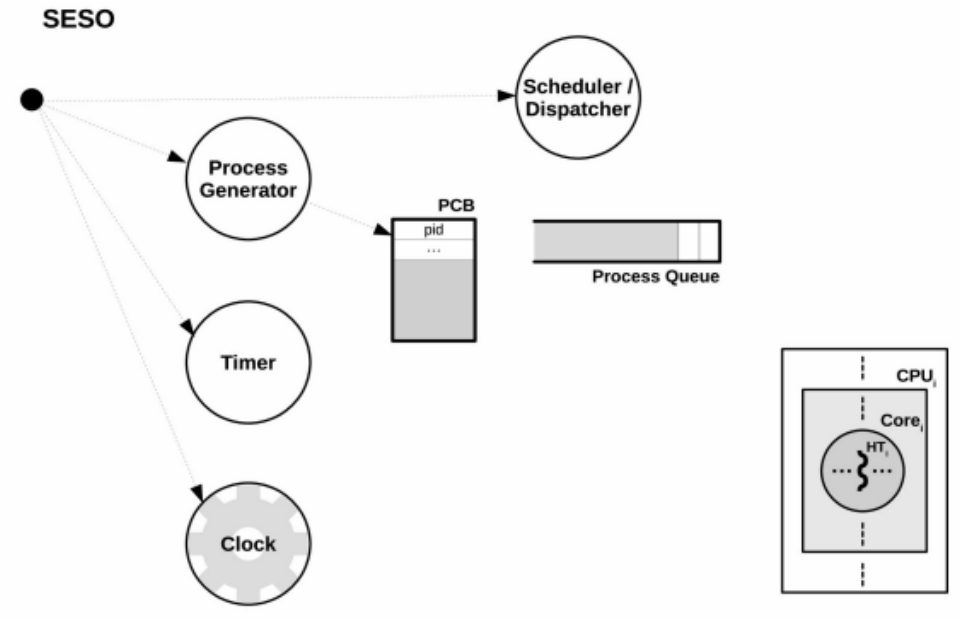
1 Sarrera

Proiektu[1] honen helburua, sistema eragile baten kernela simulatzen duen hari anitzeko programa bat garatzea da. Proiektua hainbat fasetan banatuko da eta hauetko 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.

2 Lehenengo Fasea

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 ikus daitekeen bezala honako hau izango da kernel simulatzaileak izango duen egitura.



Irudia 1

2.1 Osagaiak

Lehen aipatu bezala osagai bakoitza hari desberdin batean exekutatuko dira. Harien arteko sinkronizazioa egiteko, semaforoak eta mutex baldintzatuak erabili dira. Hauek dira osagai nagusiak:

2.1.1 Clock

Sistema batean, clock seinalea oso garrantzitsua da. Seinale honek segunduro hainbat aldiz pultso bat bidaltzen du, CPU etenduz bidaltzen den momentu horretan.Clock seinalearen maiztasuna parametro baten bidez zehaztuko da exekutatzerako orduan. Main programan zenbait unitate aldaketa egingo dira, periodoa lortzeko. Clock hariaren funtzioa nahiko simplea da, while begizta batean sartuta dagoen bitartean t balioa inkrementatuko da, eta sartutako maiztasunera iristerakoan *tick* seinaleari bat gehituko zaio. Timerra eta prozesu sortzailea sinkronizatuda egongo dira semaforoen bidez, *tick* handitzen den bakoitzeko Timerrak eta prozesu sortzaileak *tick*-a irakurri dutela jakinarazi beharko dute. Honako hau izango litzateke kodea, gutxi gorabehera.

```
void *clockfunc(void *hari_par){
    while(1){
        if(t < maiz){
            t++;
        }else{
            t = 0;
            tick++;
            printf("[CLOCK] tick: %d\n", tick);
            sem_post(&semt);
            sem_post(&semp);
        }
    }
    pthread_exit(NULL);
}
```

Ikus daitekeen bezala bi semaforo erabiltzen dira sinkronizaziorako, lehena timerrerako eta bigarrena prozesuentzako. Bi semaforoak main.c fitxategian daude hasieratuta, 0 balioarekin. Modu horrertan lehengo clock funtzioak semaforoaren balioa inkrementatzera itxaron behar dute beste bi hariek.

2.1.2 Timer

Timerra, clock-aren antzeko seinalea da, baina kasu honetan denbora tarteak edo interbaloak neurtzeko erabiltzen da. Praktika honetan timerrraren funtzioa oso erraza da, denbora tartea, parametro bezala sartzen da eta main funtzioa arduratzen da datua tratatzeaz. Clock-etik *tick* bat jasotzen duenero, timerraren t balioa, zerotik banan-banan incrementatzen joango da, sartutako denbora tartearen baliora iritsi arte. Momentu horretan timerrak denbora tartea bukatu dela adieraziko du. Ondorengoan izango litzateke timerraren inplementazioa.

```
void *timer(void *hari_par){
    t = tick;

    while(1){
        sem_wait(&semt);
        if(tick == t+param->timer){
            pthread_mutex_lock(&mutex);
            printf("[TIMER] seinalea bidalita\n");
            fflush(stdout);
            while(done < param->core_kop)
                pthread_cond_wait(&cond, &mutex);
            t = tick;
            printf("\n");
            done = 0;
            pthread_cond_broadcast(&cond2);
            pthread_mutex_unlock(&mutex);
        }
    }
    t = tick;
    pthread_exit(NULL);
}
```

Semaforoaren wait funtzioaz aparte mutex balditzatu batenak ere ageri dira. Mutex baldintzatuaren erabilerak, bere zergatia dauka. Timerrarekin scheduler bat baino gehiago sinkronizatzeko beharrezkoak dira. Semaforoak erabiltzerakoan ezin da kontrolatu scheduler guztiak mezua jaso dutela, x schedulerrek mezua jaso dutela baizik. Horregatik batzuetak probak egiterakoan echeduler berdinaren erantzuna behin baino gehiagotan agertzen zen. Hobekuntza honekin, arazoa konpontzea, eta semaforoak aurreztea lortu

dugu.

2.1.3 Scheduler/Dispatcher

Schedulerra prozasuak planifikatzeaz arduratzen da berez, baina lehengo fase honetan ez du ezer berezirik egiten. Timerrarekin mutex baldintzatuekin sinkronizatzen da eta seinalea jaso duela pantailaratzen du. Hau da bere kodea.

```
void *scheduler_dispatcher(void *hari_par){
    pthread_mutex_lock(&mutex);
    while(1){
        done++;
        printf("[SCHEDULER/DISPATCHER %d] tick read!\n", id);
        pthread_cond_signal(&cond);
        pthread_cond_wait(&cond2, &mutex);
        pthread_mutex_unlock(&mutex);
    }
    pthread_exit(NULL);
}
```

Kode zati honetan baldintzapeko mutexaren beste zatia nola funtzionatzen duen ikus daiteke.

2.1.4 Process Generator

Izenak esaten duen bezala hari honetan prozasuak sortuko dira. Parametro baten bitartez zehaztuko da prozesu kopurua, eta sortzerako orduan ausazko identifikadore bana emango zaie. Hurrengo faseari begira, weight atributuan -20 tik 15-era tartean berriro ere ausazko zenbaki bat emango zaio.

```
void *process_generator(void *hari_par){
    i = 0;
    p = 0;
    tam = TAM;
    while(i <= tam){
        srand(tick*time(NULL));

        sem_wait(&semp);

        printf("[PROCESS GENERATOR] tick read! %d\n", tick);
        pcb.pid = rand() % 100;
        pcb.weight = (rand() % (upper - lower + 1)) + lower;
        printf("[PROCESS GENERATOR] Ni ume bat naiz, nire identifikatzailea %d
            da, eta lehentasuna %d da\n", pcb.pid, pcb.lehen);

        i++;
    }
    pthread_exit(NULL);
}
```

Timerraren antzera hemen ere ikus ddezakegu, semaforoaren wait funtzioa, kode zatiaren hasieran kokatua. Ausazko zenbakiak sortzea C programazio lengoaian erronka bat da, srand() funtzioa zenbakiak sortzeko hazia zehateko da. Parentesi artean time(NULL) bakarrik jartzean identifikadore eta weight guztiak berdinak sortzen zituen, hori ekiditeko time funtzioak bueltatzen duen balioa * *tick* eginez, *tick* bakoitzeko hazi desberdina izango dugu, horrela zebaki desberdinak sortuz.

2.1.5 Main

Aurreko hari guztiak sortzeko eta beharrezko datu guztiak hartzeko main funtzioa erabiliko da. Bertan, hasteko parametro bezalako datuak jasoko dira, getopt erabilita. Getopt erabiltzeak, parametroen aurretik identifikadore bat jartzea ahalbidetzen digu, hurrengo adibidean ikus daitekeen bezala.

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

Identifikadore horren arabera begiratuko da zein datu den bakoitza switch bat erabilita. Datuen tratamendua bukatzerakoan, erabili behar ditugun semaforoak semaforoak hasieratuko dira, eta gero suntsitu hariak sortzeko funtzioa exekutatu eta gero.

```
int main(int argc, char *argv[]){

    int proz_kop, c, i, maiz, tim, core_kop, maizMHz;
    float periodoa;
    char *p;

    //getopt erabilita parametroak eskuratu
    while ((c = getopt (argc, argv, "p:m:t:c:")) != -1){
        switch (c){
            case 'p':
                proz_kop = atoi(optarg);
                break;
            case 'm':
                maiz = strtol(optarg, &p, 10);
                break;
            case 't':
                tim = strtol(optarg, &p, 10);
                break;
            case 'c':
                core_kop = strtol(optarg, &p, 10);
                break;
            case '?':
                if (optopt == 'p')
                    fprintf (stderr, "Option -%c requires an argument.\n",
                        optopt);
                else if (isprint (optopt))
```

```
                fprintf (stderr, "Unknown option `-%c'.\n", optopt);
            else
                fprintf (stderr,"Unknown option character `\\x%x'.\n",
                    optopt);
            return 1;
        default:
            abort ();
      }
    }

    maizMHz = maiz * 1000000;
    periodoa = (1.0/maizMHz)*1000000000.0;

    sem_init(&semt, 0, 0);
    sem_init(&semp, 0, 0);
    sem_init(&semc, 0, 0);
    sortu_hariak(HARIKOP, proz_kop, maizMHz, tim, core_kop);

    sem_destroy(&semp);
    sem_destroy(&semc);
    sem_destroy(&semt);

    free((void *)sch_arr);
    return(0);
}
```

sortu_hariak funtzioan, sartutako parametroak erabilita hariak sortzen ditu. Horretarako for begizta bat erabili dut, iterazioaren zenbakiaren arabera hari bat sortuko da, eta iterazioa zenbakia emango zaio identifikadore moduan hariari. Hari bakoitzak hainbat parametro ditu eta hauek gortdetzeko, *struct* bat erabiliko da.

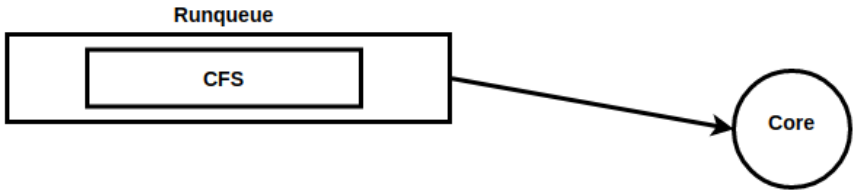
```
struct hari_param{
    int id;
    char * name;
    int p_kop;
    int maiz;
    int timer;
    int core_kop;
```

```
};
```

Ikus daitekeen bezala identifikadoreaz aparte, hari bakoitzak ere izena, prozesu kopurua, maiztasuna, timer-eko maiztasuna eta core kopurua gorde ditzake.

3 Bigarren Fasea

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. Gure kasuan prozesu bakoitzak bi egoera izango ditu itxaroten eta exekuzioan. Honako hau izango litzateke lortu nahi dugun egituraren itxura. Irudiaren ezkerraldean ikusi daitekeen bezala,



Irudia 2

itxarote ilara bat dago bertan prozesuek, corean 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 nolabait jaitxi behar da, bestela etengabe egongo litzateke prozesu berdina core-an. Aipatutako balio hori jaitxiarazteko, 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:

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

3.1 Process Generator

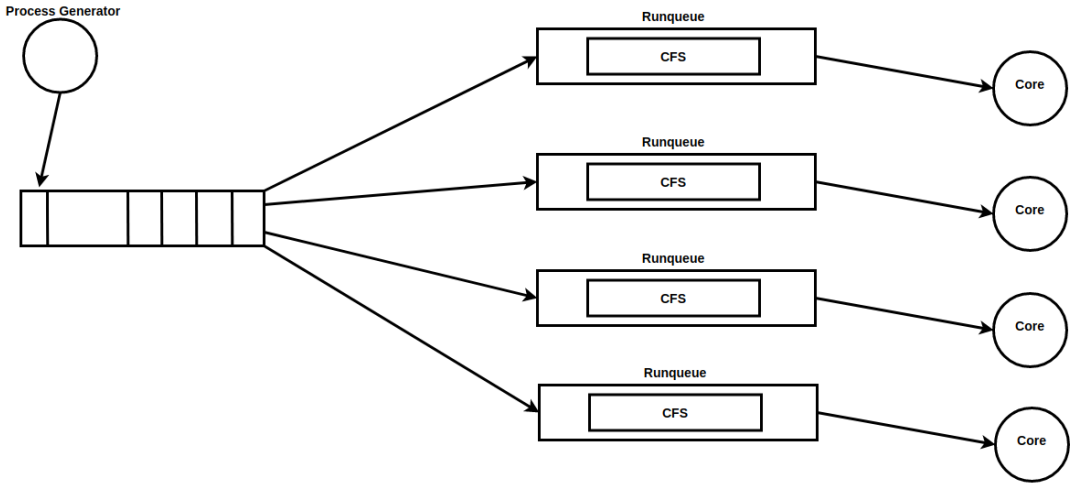
Aurretik esan bezala, prozesuek informazio berria gorde behar dute, beraz prozesuak sortzeaz arduratzen den harian aldaketak egin beharko ditugu. Lehenik eta behin, prozesu baten parametro berriei balio bat eman behar diegu, horretarako ausaz zenbaki bat esleituko zaio bai *vruntime* eta bai *weight* parametroei. Prozesuak sortzeko haria eta schedulerraren haria, ez daudenez berdin sinkronizatuta, sortzen diren prozesu guztiak array batean gorde beharko dira, horrela scheduler bakoitza prozesu bat sartzeko agindua jasotzen duenean array horretatik jasoko ditu beharrezko datua. Era honetan ez dira datuak galduko. Azkenik honako itxura izango luke hari honen funtzioak:

```
void *process_generator(void *hari_par){

    int k, j, i, p_kop, p, tam;
    int lower = 0, upper = 40, minrt = 30, maxrt = 250;
    struct hari_param *param;
    struct process_control_block pcb;

    //Hasieraketak
    param = (struct hari_param *)hari_par;
    p_kop = param->p_kop;
    i = 0;
    p = 0;
    tam = p_kop;

    //Hariaren hasierako informazioa pantailaratu
    printf("[PROCESS GENERATOR:    id = %d   name = %s  ]\n", param->id,
        param->name);

    //Funtzioko loop-a
    while(i <= p_kop){
        //Ausazko zenbakiak sortzeko
        srand(tick*time(NULL));

        sem_wait(&semp);

        DEBUG_WRITE("[PROCESS GENERATOR] tick read! %d\n", tick);
        //Prozesu nulua sortu prozesu kopurura iristen bada.
```

```
        if(i == p_kop){
            pcb.pid = 555;
            pcb.weight = 555;
            pcb.vruntime = 555;
        }else{
            pcb.pid = rand() % 100;
            pcb.weight = (rand() % (upper - lower + 1)) + lower;
            pcb.vruntime = (rand() % (maxrt - minrt + 1)) + minrt;
        }
        DEBUG_WRITE("[PROCESS GENERATOR] id: %d vruntime: %d \n", pcb.pid,
            pcb.weight);

        //Prozesuak array-ean sartu
        sch_arr[i] = pcb;
        i++;
    }
    pthread_exit(NULL);
}
```

3.2 Scheduler

Bigarren fase honetan, lehenengoan ez bezala core kopuruaren arabera schedulerrak sortuko dira. Beraz, core bakoitzak bere schedulerra izango du. Irudian ikusi daitekeen bezala,



Irudia 3

array-tik lehenengo prozesua aterako da schedulerretako batera. Hori egiteko politika, beti prozesu gutxien dituen planifikatzailera prozesua bidaltzea izango da. Horretarako core egitura bakoitzak bakoitzak informazio hau gorde behar du.

```
struct core_t{
    struct node *root;
    int core_num;
    int tamaina;
    struct process_control_block hari1[1];
    struct process_control_block hari2[1];
};
```

Parametroen arteaan tamaina ikusi dezakegu, balio hori array batean gordeko da, core-aren identifikadorearen araberako posizioan. Core guztietan prozesu kopuru minimoa zeinek daukan jakiteko *get_min* funtzioa erabiliko da. Parametro moduan array bat eta array horren tamaina jasoko ditu. Funtzio honi esker prozesuak era orekatu batean core bakoitzean sartzea lortuko dugu.

```
int getMin(int *arr, int kop){
    int i, id, min = 10000000;

    for(i = 0; i < kop; i++){
        if(arr[i] <= min){
            min = arr[i];
            id=i;
        }
    }
    return id;
}
```

Scheduler bakoitza, prozesu bat sartzea tokatzen zaionean, prozesu berria zuhaitzean sartuko du, eta tamaina inkrementatuko du, gero *get_min* exekutatuko du, hurrengo prozesua non sartu behar den jakiteko. Azkenik prozesuaren parametroak eguneratuko ditu, zatiaren hasieran aipatu bezala. Kodeak honako itxura izango luke:

```
void *scheduler_dispatcher(void *hari_par){

    struct hari_param *param;
    param = (struct hari_param *)hari_par;
    int core_num, i1 = 1, weightval, vrunt, i, sch_tam, min;
    struct process_control_block nulua;
    struct core_t core;
    struct node *root, *exec, *lag;

    //Hasieran sortutako schedulerraren informazioa
    printf("[SCHEDULER/DISPATCHER: id = %d   name = %s]\n", param->id,
        param->name);
    sleep(1);


    //Hasieraketak
    core_num = param->id - 3;

    nulua.pid = -1;
    nulua.weight = -1;
```

```
nulua.vruntime = -1;

core.core_num = core_num;
core.root = root;
core.tamaina = 0;
sch_tam = sch_arr_tam;
initArray(tam_arr);

//Funtzioko loop-ean sartu, mutex baldintzatua erabiliz.
pthread_mutex_lock(&mutex);
while(1){
    done++;
    if(i1 == 1){
        i1 = 0;
    }else{
        min = minimum;
        //Array-ko datua 555 identifikadorea edo beste gauza badauka prozesu
            nulua sartzen da exekuzioan.
        if(sch_arr[0].pid != 555 && minimum == core_num){
            //Zuhaitza hutsik badago nodo berria sortzen da eta root moduan
                jarri. Bestela insertatu zuhaitzean
            if(core.tamaina == 0){
                root = new_node(sch_arr[0]);
                exec = root;
                lag = exec;
                core.tamaina++;
            }else{
                insert(root, sch_arr[0]);
                exec = find_minimum(root);
                lag = exec;
                core.tamaina++;
            }
            tam_arr[core_num] = core.tamaina;
            for(i = 0; i < sch_tam; i++){
                sch_arr[i] = sch_arr[i+1];
            }
            sch_tam--;
```

```
                minimum = getMin(tam_arr, param->core_kop);
            }else{
                exec = new_node(nulua);
            }

            //Corera exekutatzera joan den prozesua kendu egiten da zuhaitzetik
            if(min == core_num && exec->data.vruntime <= 0){
                delete(root, lag->data);
                exec = find_minimum(root);
                core.tamaina--;
            }
            DEBUG_WRITE("[SCHEDULER/DISPATCHER %d] tick read!\n", param->id);
            printf("---------core%d--------- thread 1: [ id: %d vruntime: %d
                ]\n", core_num, exec->data.pid, exec->data.vruntime);
            inorder(root);
            printf("\n");

            //Exekuzioan dagoen vruntime balioa txikitu
            vrunt = exec->data.vruntime;
            vrunt = vrunt - param->timer;
            exec->data.vruntime = vrunt;

        }
        pthread_cond_signal(&cond);
        pthread_cond_wait(&cond2, &mutex);
        pthread_mutex_unlock(&mutex);
    }
    pthread_exit(NULL);
}
```

4 Konpilazioa eta exekuzioa

4.1 Makefile

Proiektuan zehar aldaketak probatzerakoan behin eta berriro konpilatzeko *gcc* komandoa erabili eta gero exekutatu beharrean, Makefile delako fitxategia sor daiteke. Fitxategi mota honetan Makefileen sintaxia erabiliz, zenbait atal sortzen dira. Atal bakoitzean agindu bat exekutatuko da, eta normalean all, exec eta clean dira tipikoki ikusten diren atal oinarrizkoak.

```
#SE makefile

CC = gcc
CFLAGS = -g -pthread
OBJ = binaries/seso

all:
	$(CC) src/main.c src/sched_disp.c src/timer.c src/clock.c
	    src/p_generator.c src/rbtree.c -o $(OBJ) $(CFLAGS)
exec:
	./binaries/seso -p 10 -m 1 -t 40 -c 1
clean:
	rm -rf binaries/seso
	rm -rf *.o
```

Adibidean ikus daitekeenez, all atalean konpilatzeko agindua ematen da, *gcc* komandoa eta honi jartzen zaizkion flag guztiak aldagaietan daude definituta. Exec funtzioak exekutatu egiten du konpilazioan sortutako exekutagarria, eta clean-ek utzitako arrasto guztia garbitzeko balio du, hau da ".o"fitxategiak eta exekutagarriak ezabatu egiten ditu. Makefileak nahiko malguak dira eta funtzio asko sor daitezke. Guzti hau esanda, Makefilea erabiltzeko *make* komando lerroan jarriz Makefileko all funtzioa exekutatuko du eta *makeexec* jarri ezgero exec funtzioa.

4.2 Lmake

Aurretik azaldutako prozesu hau bera egiteko beste aukerak daude horien artean Lmake[2]. Lmake edo Luamake maila baxuko eraikuntza sistema da, eta malgutasun handia dauka fitxategi bitarrak sortzeko zatiak aldatzerako orduan. Konfigurazio fitxategiak ez dira Makefilean bezain sinpleak, baina ondo ulertzen dira. Hau izango litzateke aurreko Makefilearen bertsioa Lmake-n:

```
lmake_compatibility_version(1)
function build()
    lmake_set_compiler("/bin/gcc")
    lmake_set_compiler_flags("-pthread -Wall -Iinclude")
    lmake_set_compiler_out("binaries/%.o")
    lmake_compile("src/main.c src/sched_disp.c src/timer.c src/clock.c
        src/p_generator.c src/rbtree.c")
    lmake_set_linker("/bin/gcc")
    lmake_set_linker_flags("-pthread")
    lmake_set_linker_out("binaries/seso")
    lmake_link("binaries/main.c.o binaries/sched_disp.c.o binaries/timer.c.o
        binaries/clock.c.o binaries/p_generator.c.o binaries/rbtree.c.o")
end
function exec()
    build()
    lmake_exec("binaries/seso -p 10 -m 1 -t 40 -c 1")
end
function clean()
    lmake_exec("rm binaries/main.c.o binaries/sched_disp.c.o binaries/timer.c.o
        binaries/clock.c.o binaries/p_generator.c.o binaries/rbtree.c.o")
end
```

Fitxategia ".lua"letrekin bukatu behar da.

4.3 Exekuzioa

Programa exekutatzen dugunean, hasieran sortu diren hari guztien informazioa ageri da, ondoren core bakoitzaren egoera agertuko da. Core bakoitzaren eskuin aldean exekuzioan dagoen prozesuaren identifikadorean eta bere *vruntime* agertuko dira, azpialdean aldiz zuhaitzean dauden prozesuak.

```
[+] Linking seso


Parametroak:
-----------------------------------------------------------------
[Maiztasuna: 10 MHz  Periodoa: 100 ns]


Creating threads...
-----------------------------------------------------------------
[PROCESS GENERATOR:     id = 0   name = Process Generator ]
[TIMER:                 id = 1   name = Timer             ]
[CLOCK:                 id = 2   name = Clock             ]
[SCHEDULER/DISPATCHER:  id = 3   name = Scheduler/Dispatcher]
-----------------------------------------------------------------



Clock, timer and process signals:
-----------------------------------------------------------------

[TIMER] seinalea bidalita

---------core0--------- thread 1: [ id: 83 vruntime: 158 ]
 (83, 158)

[TIMER] seinalea bidalita

---------core0--------- thread 1: [ id: 83 vruntime: 118 ]
 (83, 118) (29, 216)
```

```
[TIMER] seinalea bidalita

---------core0--------- thread 1: [ id: 83 vruntime: 78 ]
 (83, 78) (69, 165) (29, 216)

[TIMER] seinalea bidalita

---------core0--------- thread 1: [ id: 83 vruntime: 38 ]
 (83, 38) (74, 115) (69, 165) (29, 216)
```

Bibliografia

[1] U. Fernandez. Seso. [Online]. Available: https://github.com/UnaiFernandez/SE_PRAKTIKA

[2] I. Galardi. Lmake. [Online]. Available: https://github.com/IkerGalardi/LMake