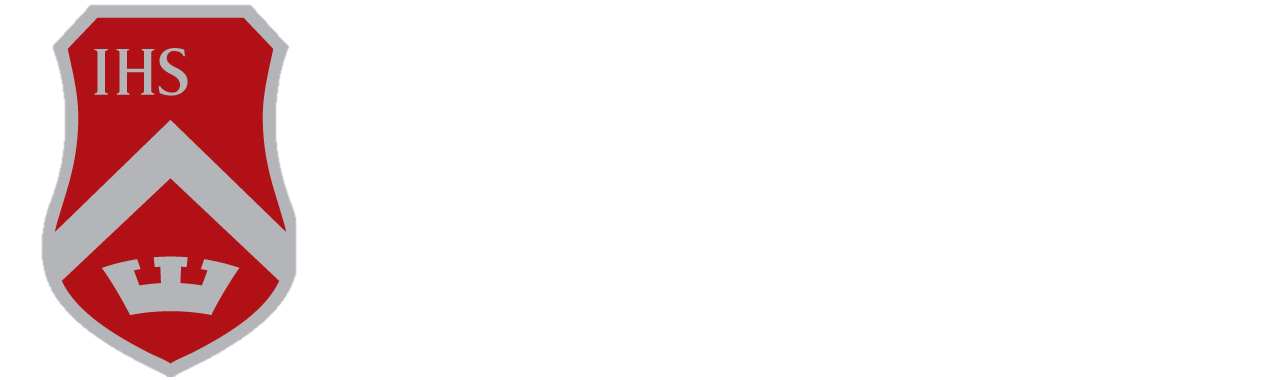
**2020 Fall- Operating System**

**Pintos Project 3**

**Threads**

****

|  |  |
| --- | --- |
| 담당 교수 / 분반 : | 김영재 교수 / 2분반 |
| 이름 / 학번 : | 김지수 / 20160366 |
| 개발 기간 : | 2020.11.30 ~ 12.06 |

목차

1. **개발 목표**
2. **개발 범위 및 내용**
   1. **개발 범위**
      * 1. Alarm Clock
        2. Priority Scheduling
        3. Advanced Scheduler
   2. **개발 내용**
      * 1. Alarm Clock
        2. Priority Scheduling
        3. Advanced Scheduler
3. **추진 일정 및 개발 방법**
   1. **추진 일정**
   2. **개발 방법**

Alarm Clock

Priority Scheduling

Advanced Scheduler

1. **연구 결과**
   1. **제작 내용**
2. Alarm Clock
3. Priority Scheduling
4. Advanced Scheduler
   1. **시험 및 평가 내용**

- priority-lifo 수행 결과 해석

- make check 수행 결과

1. **개발 목표**

현재 핀토스에는 Thread를 스케줄링 할 때 priority와 aging을 고려하지 않는 round-robin scheduling기법을 채택하고 있다. priority를 고려하여 새롭게 priority가 설정되는 경우 ready queue의 가장 높은 priority thread가 현재 스레드의 priority보다 높게 된다면 새롭게 scheduling되도록 만들어 준다. 이 때 낮은 priority의 스레드가 CPU자원으로부터 starvation을 겪지 않도록 aging도 구현해준다. 어떤 스레드가 일정 clock ticks동안 sleep하게 되는 경우 현재에는 ready state와 run state를 왔다 갔다 하며 CPU를 낭비하고 있다. 이 또한 sleep/awake기법으로 재 구현 하여 CPU utilization을 높인다. 더 나아가, Multi-Level Feedback Queue scheduling을 추가로 구현하여 주어진 공식에 맞게 priority가 계산되고 필요한 경우 rescheduling하도록 한다.

1. **개발 범위 및 내용**
   1. **개발 범위**
2. **Alarm Clock**

timer\_sleep을 호출한 thread를 busy-waiting이 아닌, 주어진 시간 동안 block된 상태로 sleep queue에 유지하다 필요한 시간에 awake하도록 구현한다.

1. **Priority Scheduling**

현재의 pintos scheduler는 우선순위와 aging을 고려하지 않는 단순한 round-robin scheduler이다. ready list에 추가된 새로운 스레드의 우선순위가 running중인 스레드보다 높다면 preemption이 일어나도록 한다. 여러 스레드가 semaphore를 기다리는 동안에도 우선순위가 높은 스레드가 CPU를 차지하도록 한다. 이 과정에서 starvation을 방지하기 위해 aging도 구현한다.

1. **Advanced Scheduler**

각 priority마다 queue를 가지고 높은 우선순위의 queue에 대해서 Round-Robin으로 스케줄링이 일어난다. 이때 priority계산은 이전의 것과 달리 (1) nice와 2) recent\_cpu값으로 보다 복잡하게 계산된다. 1초마다 load\_average(1분동안 수행가능한 프로세스의 평균 개수)와 모든 스레드에 대해 recent\_cpu값과 priority값이 재계산되고 현재 실행 중인 스레드는 4 ticks마다 priority가 재계산된다.

* 1. **개발 내용**

1. **Alarm Clock**

timer\_sleep의 호출로 sleep\_list에 스레드를 저장할 때 깨울 시간도 함께 저장한다. 매 tick마다 실행되는 timer\_interrupt 함수가 호출될 때 마다 thread\_awake함수를 통해 sleep list에서 깨울 시간이 된 스레드가 있는지 하나하나 확인하여, 존재한다면 sleep\_list에서 제거하고 thread를 unblock(=ready\_list에 추가)해준다.

1. **Priority Scheduling**

ready\_list가 업데이트 되면 늘 priority checking을 통해 우선순위대로 선점이 일어날 수 있도록 한다. ready\_list가 업데이트 되는 경우는 크게 **(1) thread\_yield가 일어날 때**와 **(2) thread\_unblock이 일어날 때**로 볼 수 있다.

(1) thread\_yield가 일어나면 schedule함수가 내부에서 호출되어 ready\_list의 다음 스레드와 context switch가 일어나므로, thread\_yield에서 현재 스레드를 ready\_list로 이동시킬 때 우선순위에 맞게 추가해주는 코드만 추가하면 된다.

(2) thread\_unblock은 대표적으로 thread\_create을 통해 새로운 스레드가 만들어 졌을 경우와 timer\_interrupt 도중에 awake할 스레드를 발견한 경우에 호출된다. 전자의 경우 thread\_unblock을 통해 새로운 스레드를 ready list에 우선순위에 맞게 추가한 후 test\_max\_priority함수를 호출해, 현재 실행 중인 스레드의 우선순위보다 ready\_list 맨 앞 스레드의 우선순위가 높은 경우에 thread\_yield가 일어나도록 한다. 후자의 경우 interrupt 도중이므로 thread\_yield의 호출이 불가하기 때문에, 하나의 thread라도 awake한 경우 intr\_yield\_on\_return을 호출하여 timer\_interrupt를 끝낸 후 intr\_handler에서 thread\_yield가 호출되도록 한다.

1. **Advanced Scheduler**

priority 계산에 필요한 요소는 총 3가지가 있다: PRI\_MAX, nice, recent\_cpu. (1) **PRI\_MAX**는 priority가 가질 수 있는 가장 큰 값으로 현재 핀토스에서는 63으로 define되어있다. (2) **nice**는 -20 부터 20까지의 숫자로, 음수일수록 우선순위가 증가하고 양수일수록 덜 중요 해지며 초기에는 0으로 설정된다. (3) **recent\_cpu**는 최근에 얼마나 많은 CPU time을 썼는지를 나타내며 init 스레드의 초기값은 0이고, 나머지는 부모의 것을 상속받는다. 현재 running thread의 경우 매 1tick마다 1씩 이 값이 증가하고 4ticks마다 이를 기반으로 한 priority가 재계산되며, 1초마다는 모든 스레드의 recent\_cpu가 재계산되는데 이 때 재계산할 때 load\_average의 변화도 중요하다. **load\_average**는 1분동안 평균 수행할 수 있는 스레드의 개수로, **ready\_thread**(ready\_list의 스레드 및 (idle 이 아니라면) 현재 스레드의 개수)와 load\_average의 EWMA를 통해 계산한다. 이 값은 1초마다 계산되는 system-wide value이며, 이 변한 load\_avg값과 1초 동안 실행하며 변한 thread들의 recent\_cpu값, nice값들을 기반으로 recent\_cpu가 계산되고, 이 recent\_cpu과 nice값을 기반으로 priority가 계산되는 것이다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 11.29 | 11.30 | 12.01 | 12.02 | 12.03 | 12.04 | 12.05 | 12.06 |
| Pintos Manual 읽기 | | | Alarm Clock, Priority scheduling 구현 및 디버깅 | | Advanced Scheduling 구현 및 디버깅 | | 보고서 작성 |

* 1. **개발 방법**

1. **Alarm Clock**

src/thread/thread.h **struct thread**에 깨워야 할 tick정보를 담는 field(**wakeup\_tick**)를 추가한다.

src/thread/thread.c 또한 **sleep list**를 추가하고, sleep list에서 대기중인 스레드들의 wakeup\_tick 값 중 가장 작은 값(**next\_tick\_to\_awake**)도 전역 변수로 추가한다. timer\_sleep에서 호출하여 해당 thread를 sleep 상태로 만드는 **thread\_sleep**함수, timer\_interrupt에서 호출하여 매 tick마다 깨울 스레드를 찾는 **thread\_awake**함수, next\_tick\_to\_awake를 업데이트 하는 함수(**update\_next\_tick\_to\_awake**), next\_tick\_to\_awake를 반환하는 함수 (**get\_next\_tick\_to\_awake**)를 추가한다. **thread\_init**에 **sleep\_list**를 초기화하는 코드를 추가한다.

src/devices/timer.c **timer\_sleep**에서 busy waiting으로 wait하던 것을 **지우고**, sleep\_list에 thread를 깨울 시간과 함께 저장하는 함수(**thread\_sleep**)를 호출한다. **timer\_interrupt**에 sleep list에서 깨울 시간이 된 스레드가 있는지 확인하여 sleep\_list에서 제거하고 thread를 unblock해주는 함수(**thread\_awake**)를 호출한다.

1. **Priority Scheduling**

src/thread/thread.h **struct thread**에 ready list에 있던 ticks 정보를 담는 field(**ticks\_in\_ready**)를 추가한다.

src/thread/thread.c **thread\_create**에 **thread\_unblock**이후 우선순위대로 선점이 발생하도록 코드를 추가한다. **thread\_unblock**과 **thread\_yield**에 ready\_list에 우선순위대로 정렬되게 삽입하도록 코드를 수정한다. **thread\_awake**에 thread\_unblock이 일어났다면 intr\_yield\_on\_return을 호출하도록 코드를 수정한다. **thread\_set\_priority**를 통해 현재 스레드의 우선순위가 변경된 경우 우선순위대로 선점이 발생하도록 코드를 추가한다. 우선순위를 체크하여 필요한 경우 선점을 일으키는 함수 **test\_max\_priority**를 구현한다. list에 우선순위대로 삽입하도록 하는 비교함수 **cmp\_priority**를 구현한다.

**thread\_unblock**와 **thread\_yield**에서 스레드가 ready상태로 바뀔 때 ticks\_in\_ready 값도 0으로 업데이트 한다. **thread\_tick**에서 **thread\_prior\_aging**이 true인 경우에 **thread\_aging**을 호출하도록 한다. **thread\_aging**이 호출되면 ready\_list의 ticks\_in\_ready값을 모두 업데이트 하고 그 값에 비례하여 priority를 aging시킨 후, 우선순위에 따른 선점이 필요하다면 intr\_yield\_on\_return을 호출하도록 구현한다.

src/thread/synch.c **sema\_down**에 semaphore를 기다리는 waiter리스트에 우선순위대로 삽입되도록 코드를 수정한다. **sema\_up**에 우선순위대로 한번 더 정렬한 후 다음 semaphore를 이용할 thread를 unblock하여 ready list로 이동시킨다. 이때도 ready\_list가 업데이트 되었기 때문에 우선순위대로 선점이 일어나도록 코드를 추가한다. 또한 waiter list에 우선순위대로 삽입하도록 하는 비교함수 **cmp\_sema\_priority**를 구현한다.

1. **Advanced Scheduler**

src/thread/fixed\_point.h recent\_cpu와 load\_avg는 실수이나, pintos에서는 부동소수점 연산을 지원하지 않기 위해 **고정소수점 연산**을 위한 파일을 하나 만든다. fixed point간, 또는 fixed point와 integer간의 연산을 정의 하였다. pintos manual의 appendix에 따라 17.14 fixed-point number representation을 이용한다.

src/thread/thread.h **struct thread**에 각 스레드의 **nice**와 **recent\_cpu** field를 추가한다

src/thread/thread.c **init\_thread**에 추가한 field를 초기화하는 코드를 추가한다. system wide 변수인 **load\_ avg**를 전역변수로 선언하고 **thread\_init**에서 초기화한다. BSD 스케줄링을 위해 priority, recent\_cpu, load\_avg를 각각 계산할 **mlfqs\_priority**, **mlfqs\_recent\_cpu**, **mlfqs\_load\_avg**와, 1 tick마다 현재 스레드의 recent\_cpu를 증가 시켜주는 **mlfqs\_increment**, 1초마다 recent\_cpu와 priority를 재계산해주는 **mlfqs\_recalc**를 추가한다. 또한 뼈대만 존재하는 **thread\_get\_nice**, **thread\_set\_nice**, **thread\_get\_recent\_cpu**, **thread\_get\_load\_avg**를 각 기능을 하도록 구현한다

src/devices/timer.c **timer\_interrupt**에 thread\_mlfqs가 true인 경우 1 ticks마다, 4 tick마다, 1초(=100tikcs)마다 필요한 업데이트 함수를 호출하도록 코드를 추가한다.

1. **연구 결과**
   1. **Flow Chart**

|  |  |
| --- | --- |
| **Alarm Clock** | **Priority scheduling** |
|  |  |

* 1. **제작 내용**

1. **Alarm Clock**

(1) src/thread/thread.h/struct thread

|  |
| --- |
| **struct thread**  **{**  **︙**  /\* Alarm clock. \*/  **int64\_t wake\_up\_ticks;**  /\* Ticks for waking up from sleeping. \*/  **︙**  **};** |

스레드 마다 sleep에 들어가게 되는 경우 깨어날 tick정보를 저장하는 변수 filed를 만들어 준다.

(2-1) src/thread/thread.c

|  |
| --- |
| **︙**  /\* List for sleeping processes. \*/  **static struct list sleep\_list;**  /\* Tick for storing smallest wakeup\_tick in sleep list. \*/  **int64\_t next\_tick\_to\_awake;**  **︙** |

block state가 되어 sleep에 들어가 있는 동안 스레드들이 연결될 리스트와, 깨울 tick의 비교를 위해 가장 작은 next\_tick\_to\_awake값을 저장하는 전역 변수를 선언한다.

(2-2) src/thread/thread.c/void thread\_init(void)

|  |
| --- |
| **void thread\_init(void)**  **{**  **︙**  **list\_init(&sleep\_list);**  **︙**  **};** |

새롭게 추가한 sleep\_list를 초기화하는 코드를 thread\_init 함수에 추가한다. 따라서 최초의 kernel thread가 생성될 때 sleep list 또한 초기화 될 수 있다.

(2-3) src/thread/thread.c/void thread\_sleep(void)

|  |
| --- |
| **void thread\_sleep(void)**  **{**  **enum intr\_level old\_level;**  **struct thread \* t = thread\_current();**  **ASSERT(is\_thread(t));**  **old\_level = intr\_disable();**  **t->wakeup\_ticks = ticks;**  **list\_push\_back(&sleep\_list,&t->elem);**  **thread\_block();**  **if(get\_next\_tick\_to\_awake( )>ticks)**  **update\_next\_tick\_to\_awake(ticks);**    **intr\_set\_level(old\_level);**  **};** |

현재 스레드가 idle이 아닌 경우, 이 스레드가 깨워질 wakeup\_ticks 정보를 업데이트, sleep\_list에 해당 스레드를 추가하고 thread\_block을 호출한다. thread\_block에서는 현재 스레드의 state를 BLOCKED로 바꾼 후 schedule을 호출해 ready list 중 새로운 한 스레드가 running상태가 된다. 새롭게 sleep에 들어가게 되는 스레드의 tick정보를 현재 next\_tick\_to\_awake와 비교하여 더 작다면 업데이트 해주는 과정도 필요하다. 또한 이 과정에서 interrupt가 걸리지 않도록 막아준다.

(2-4) src/thread/thread.c/void thread\_awake(int64\_t ticks)

|  |
| --- |
| **void thread\_awake(int64\_t ticks)**  **{**  **struct list\_elem \*e;**  **for (e = list\_begin(&sleep\_list); e != list\_end(&sleep\_list); )**  **{**  **struct thread \*t = list\_entry(e, struct thread, elem);**  **if (t->wakeup\_ticks <= ticks)**  **{**  **e = list\_remove(e);**  **thread\_unblock(t);**  **}**  **else e = list\_next(e);**  **}**  **};** |

sleep\_list를 순회하며 각 스레드의 wakeup\_ticks가 현재 깨워야 할 가장 작은 tick값 보다 작거나 같다면 그 스레드를 깨운다. 즉, 그 스레드를 sleep\_list에서 제거하고 thread\_unblock함수를 호출하게 되는데, thread\_unblock함수에서는 해당 스레드를 ready\_list에 추가하고 state를 READY로 바꿔준다.

(2-5) src/thread/thread.c/void update\_next\_tick\_to\_awake(int64\_t ticks)

|  |
| --- |
| **void update\_next\_tick\_to\_awake(int64\_t ticks)**  **{**  **next\_tick\_to\_awake = ticks;**  **};** |

인자로 전달받은 ticks값을, 현재의 가장 작은 awake tick값으로 업데이트 한다. 새로운 스레드가 sleep\_list에 추가될 때 해당 전역변수가 업데이트되어야 하는 경우에 호출되는 함수이다.

(2-6) src/thread/thread.c/int64\_t get\_next\_tick\_to\_awake(void)

|  |
| --- |
| **int64\_t get\_next\_tick\_to\_awake(void)**  **{**  **return next\_tick\_to\_awake;**  **};** |

현재의 가장 작은 awake tick값을 반환한다. 새로운 스레드가 sleep\_list에 추가될 때 해당 전역변수가 업데이트되어야 하는 경우를 판단하기 위해 호출되는 함수이다.

(3-1) src/devices/timer.c/void timer\_sleep(int64\_t ticks)

|  |
| --- |
| **void timer\_sleep(int64\_t ticks)**  **{**  **int64\_t start = timer\_ticks();**  **ASSERT(intr\_get\_level() == INTR\_ON);**  **thread\_sleep(start+ticks);**  **};** |

기존에 loop를 돌며 run state와 ready state를 오가던 busy waiting코드를 삭제하고, thread\_sleep함수를 호출하여 현재의 ticks값에 총 sleep에 있을 tick을 더한 awake time을 전달하며 sleep/awake 방법을 구현한다.

(3-2) src/devices/timer.c/void timer\_interrupt(struct intr\_frame \* args UNUSED)

|  |
| --- |
| **void timer\_interrupt(struct intr\_frame \* args UNUSED)**  **{**  **ticks++;**  **thread\_awake(ticks);**  **thread\_tick();**  **};** |

매 1 tick마다 실행되는 timer\_interrupt에 sleep\_list에 있는 스레드들 중 깨울 스레드가 있는지 확인하기 위한 thread\_awake 함수를 호출하도록 코드를 추가한다. timer\_interrupt에서 매 1tick마다 증가시키는, OS에서 부팅된 이후로의 현재의 ticks를 인자로 전달한다.

1. **Priority Scheduling**

(1) src/thread/thread.h/struct thread

|  |
| --- |
| **struct thread**  **{**  **︙**  /\* Cumulated Ticks in Ready list. \*/  **int64\_t ticks\_in\_ready;**  /\* Update value when it is in ready list. \*/  **︙**  **};** |

스레드 마다 ready list에 머문 ticks 정보를 담을 field를 선언해준다. 이는 ready\_list에 머문 정도를 체크해서 aging을 구현하는 데 사용된다.

(2-1) src/thread/thread.c/tid\_t thread\_create(const char \*name, int priority, thread\_func \*function, void \*aux)

|  |
| --- |
| **tid\_t thread\_create(const char \*name, int priority, thread\_func \*function, void \*aux**)  **{**  **︙**  /\* Add to run queue. \*/  **thread\_unblock(t);**  **test\_max\_priotity();**  **return tid;︙**  **};** |

새로 만든 스레드를 thread\_unblock을 통해 ready\_list에 추가한다. ready\_list가 업데이트 되었으니 test\_max\_priority함수를 호출하여 필요한 경우 높은 우선순위의 스레드의 CPU 선점이 일어나도록 한다.

(2-2) src/thread/thread.c/void thread\_unblock(struct thread \*t)

|  |
| --- |
| **void thread\_unblock(struct thread \*t)**  **{**  **︙**  **list\_insert\_ordered(&ready\_list, &t->elem, \*cmp\_priority, NULL);**  **t->status = THREAD\_READY;**  **t->ticks\_in\_ready = 0;**  **︙**  **};** |

인자로 넘어온 스레드를 ready 상태로 만들어 ready\_list에 넣을 때 cmp\_priority 비교함수를 통해 우선순위 내림차순으로 정렬되도록 list\_insert\_ordered함수를 사용한다. 또한, aging을 구현하기 위해 ready\_list에 있는 동안의 tick을 세는 변수(ticks\_in\_ready)를 0으로 세팅해준다.

(2-3) src/thread/thread.c/void thread\_yield(void)

|  |
| --- |
| **void thread\_yield(void)**  **{**  **struct thread \*cur = thread\_current();**  **︙**  **if (cur != idle\_thread)**  **list\_insert\_ordered(&ready\_list, &cur->elem, \*cmp\_priority, NULL);**  **cur->status = THREAD\_READY;**  **cur->ticks\_in\_ready = 0;**  **︙**  **};** |

현재 실행중인 스레드를 ready\_list에 넣을 때 cmp\_priority 비교함수를 통해 우선순위 내림차순으로 정렬되도록 list\_insert\_ordered함수를 사용한다. 또한, aging을 구현하기 위해 ready\_list에 있는 동안의 tick을 세는 변수(ticks\_in\_ready)를 0으로 세팅해준다.

(2-4) src/thread/thread.c/void thread\_awake(int64\_t ticks)

|  |
| --- |
| **void thread\_awake(int64\_t ticks)**  **{**  **int flag = 0;**  **struct list\_elem \*e;**  **for (e = list\_begin(&sleep\_list); e != list\_end(&sleep\_list); )**  **{**  **struct thread \*t = list\_entry(e, struct thread, elem);**  **if (t->wakeup\_ticks <= ticks)**  **{**  **e = list\_remove(e);**  **thread\_unblock(t);**  **flag = 1;**  **}**  **else e = list\_next(e);**  **}**  **if(flag) intr\_yield\_on\_return();**  **};** |

awake를 통해 ready list가 업데이트 되었다면, test\_max\_priority함수를 호출하여 필요한 경우 높은 우선순위의 스레드의 CPU 선점이 일어나도록 해야 하나, thread\_awake는 timer\_interrupt도중에 호출된 것으로 인터럽트 중에는 thread\_yield가 일어날 수 없다. 따라서, flag를 만들어 한 스레드라도 thread\_unblock이 되어 ready list가 업데이트 되었다면 intr\_yield\_on\_return을 통해 interrupt가 끝나고 thread\_yield가 일어날 수 있도록 한다.

(2-5) src/thread/thread.c/void thread\_set\_priority(int new\_priority)

|  |
| --- |
| **void thread\_set\_priority(int new\_priority)**  **{**  **thread\_current()->priority = new\_priority;**  **test\_max\_priotity();**  **};** |

현재 실행 중인 스레드의 priority가 변경되는 경우에도, test\_max\_priority함수를 호출하여 높은 우선순위의 스레드의 CPU 선점이 일어나도록 한다.

(2-6) src/thread/thread.c/void test\_max\_priority(void)

|  |
| --- |
| **void** **test\_max\_priority(void)**  **{**  **if (!list\_empty(&ready\_list))**  **{**  **struct list\_elem \*e = list\_begin(&ready\_list);**  **struct thread \*t = list\_entry(e, struct thread, elem);**  **if (thread\_current()->priority < t->priority)**  **thread\_yield();**  **}**  **};** |

ready\_list가 업데이트 되거나, 현재 스레드의 priority가 변경된 경우 preemption이 필요한지 테스트하는 함수이다. 우선순위로 정렬되어 있기 때문에 ready\_list 맨 앞에는 가장 우선순위가 큰 스레드가 존재하므로 그 우선순위 값과 현재 스레드의 우선순위를 비교한 후 전자가 더 크다면 thread\_yield를 호출한다.

(2-7) src/thread/thread.c/bool cmp\_priority(const struct list\_elem \*a, const struct list\_elem \*b, void \*aux UNUSED)

|  |
| --- |
| **bool cmp\_priority(const struct list\_elem \*a, const struct list\_elem \*b, void \*aux UNUSED)**  **{**  **struct thread \*t1 = list\_entry(a, struct thread, elem);**  **struct thread \*t2 = list\_entry(b, struct thread, elem);**  **if (t1->priority > t2->priority)**  **return true;**  **else**  **return false;**  **};** |

list\_insert\_ordered를 통해 ready\_list에 정렬되게 삽입할 때 사용하는 비교함수이다. 두 스레드의 우선순위를 비교해 우선순위가 더 큰 값이 먼저 오도록 Boolean값을 반환한다.

(2-8) src/thread/thread.c/void thread\_tick(void)

|  |
| --- |
| **void** **thread\_tick(void)**  **{**  **#ifndef USERPROG**  **if(thread\_prior\_aging==true)**  **thread\_aging();**  **#endif**  **};** |

aging 테스트를 위해 init.c에서 thread\_prioir\_aging이 true가 된 경우, thread\_aging을 호출하여 aging을 구현한다.

(2-9) src/thread/thread.c/void thread\_aging(void)

|  |
| --- |
| **void** **thread\_aging(void)**  **{**  **struct list\_elem \*e;**  **struct thread \*t;**  **int max\_pri;**  **for (e = list\_begin(&ready\_list); e != list\_end(&ready\_list); e=list\_next(e))**  **{**  **t = list\_entry(e, struct thread, elem);**  **t->ticks\_in\_ready++;**  **int pri = t->priority + 4\*(t->ticks\_in\_ready);**  **if(pri>PRI\_MAX)**  **pri = PRI\_MAX;**  **t->priority = pri;**  **}**  **if (!list\_empty(&ready\_list))**  **{**  **list\_sort(&ready\_list, cmp\_priority, NULL);**  **t = list\_entry(list\_front(&ready\_list), struct thread, elem);**  **max\_pri = t->priority;**  **}**  **if (thread\_current()->priority < max\_pri)**  **intr\_yield\_on\_return();**  **};** |

매 1tick마다 호출되는 timer\_interrupt 🡪 thread\_tick 에서 thread\_prioir\_aging이 true라면 이 함수가 호출된다. ready\_list들 순회하며 각 스레드의 ticks\_in\_ready값을 +1 해주어 ready\_list에서 머문 tick 값을 업데이트 한다. 또한 그 머문 시간에 비례하여 priority를 aging해주기 위해 임의로 4라는 값을 머문 tick에 곱하여 기존 priority에 더해준다(MAX를 넘게 되면 MAX값으로).

ready list가 업데이트 되었으나, thread\_awake는 timer\_interrupt도중에 호출된 것으로 인터럽트 중에는 thread\_yield가 일어날 수 없다. 따라서 ready\_list를 새로 업데이트된 priority들을 고려하여 sort를 한 후, 가장 큰 priority값이 현재의 스레드의 priority값보다 크다면 intr\_yield\_on\_return을 통해 interrupt가 끝나고 thread\_yield가 일어날 수 있도록 한다.

(3-1) src/thread/synch.c/void sema\_down(struct semaphore \* sema)

|  |
| --- |
| **void sema\_down(struct semaphore \* sema)**  **{**  **︙**  **while (sema->value == 0)**  **{**  **list\_insert\_ordered (&sema->waiters, &thread\_current()->elem, \*cmp\_priority, NULL);**  **thread\_block ();**  **}**  **︙**  **};** |

semaphore를 얻고 스레드가 waiter list에 들어갈 때 우선순위대로 삽입되도록 list\_insert\_ordered를 호출한다. 이때 우선순위 비교함수 cmp\_priority를 사용한다.

(3-2) src/thread/synch.c/void sema\_up(struct semaphore \* sema)

|  |
| --- |
| **void sema\_up(struct semaphore \* sema)**  **{**  **︙**  **if (!list\_empty(&sema->waiters))**  **{**  **list\_sort(&sema->waiters, cmp\_priority, NULL);**  **thread\_unblock (list\_entry (list\_pop\_front (&sema->waiters),**  **struct thread, elem));**  **}**  **︙**  **test\_max\_priority();**  **};** |

semaphore를 다음 기다리는 스레드에게 할당하기 전에 우선순위대로 정렬을 한번 해준 후 다음으로 우선순위가 높은 thread를 unblock하여 ready\_list에 넣어준다. 그러고는 우선순위에 따른 선점이 필요한지 테스트하여 yield하는 test\_max\_priority를 호출한다.

1. **Advanced Scheduler**

(1) src/thread/fixed\_point.h

**#define F (1<< 14)** //fixed point 1

|  |  |
| --- | --- |
| **int int\_to\_fp(int n)**  **{**  **return n\*F;**  **}** /\* Transforms Integer n into Fixed point. \*/ | **int fp\_to\_int\_round(int x)**  **{**  **return x/F;**  **}** /\* Transforms Fixed point X into Integer rounding up to nearest integer. \*/ |
| **int fp\_add\_int(int x, int n)**  **{**  **return x+n\*F;**  **}** /\* Adds one fixed point and one integer.\*/ | **int fp\_sub\_fp(int x, int y)**  **{**  **return x-y;**  **}**/\* Subtracts one fixed point X by another fp.\*/ |
| **int fp\_mult\_fp(int x, int y)**  **{**  **return ((int64\_t)x)\*y/F;**  **}**/\* Multiplies x by y (two fixed points).\*/ | **int fp\_mult\_int(int x, int n)**  **{**  **return x\*n;**  **}** /\* Multiplies x by n (one fixed point & integer).\*/ |
| **int fp\_div\_fp(int x, int y)**  **{**  **return ((int64\_t)x)\*F/y;**  **}**/\* Divides x by y (two fixed points).\*/ | **int fp\_div\_int(int x, int n)**  **{**  **return x/n;**  **}**/\* Divides x by n (one fixed point & integer).\*/ |

17.14로 표현한 1을 이용해 integer를 변형시킨 후 각 연산을 진행한다.

(2) src/thread/thread.h/struct thread

|  |
| --- |
| **struct thread**  **{**  **︙**  /\* Additional Data Structure for BSD scheduler. \*/  **int nice;**  **int recent\_cpu;**  **};** |

스레드 마다 BSD scheduler에서 priority 계산에 필요한 정보를 저장하는 변수 filed를 만들어 준다.

(3-1) src/thread/thread.c/void init\_ thread(struct thread \*t, const char \*name, int priority)

|  |
| --- |
| **void init\_ thread(struct thread \*t, const char \*name, int priority)**  **{**  **︙**  **t->nice = (t==initial\_thread)? 0 : thread\_current()->nice;**  **t->recent\_cpu =(t==initial\_thread)? 0 : thread\_current()->recent\_cpu;**  **︙**  **};** |

새롭게 추가한 nice와 recent\_cpu field를 초기화하는 코드를 init\_thread 함수에 추가하여 스레드 생성시 initial thread라면 0을, 아니라면 부모의 스레드가 가진 값을 갖도록 한다.

(3-2) src/thread/thread.c

|  |
| --- |
| /\* Load average value for BSD scheduler.\*/  **static int load\_avg;**  **void thread\_init(void)**  **{**  **︙**  **load\_avg = int\_to\_fp(0);**  **︙**  **};** |

system wide인 load\_avg를 전역변수로 선언하고, 최초의 kernel thread가 생성될 때 load\_avg 변수를 0으로 초기화한다.

(3-3) src/thread/thread.c/void mlfqs\_priority(struct thread \*t)

|  |
| --- |
| **void mlfqs\_priority(struct thread \*t)**  **{**  **if (t != idle\_thread)**  **{**  **int tmp = fp\_to\_int\_round(fp\_sub\_fp(fp\_sub\_fp(int\_to\_fp(PRI\_MAX), fp\_div\_int(t-> recent\_cpu, 4)), fp\_mult\_int(int\_to\_fp(t->nice), 2)));**    **t->priority = (tmp>PRI\_MAX)? PRI\_MAX : (tmp<PRI\_MIN)? PRI\_MIN : tmp;**  **}**  **}** |

**priority = PRI\_MAX – (recent\_cpu / 4) – (nice\*2)**의 수식에 맞게 인자로 넘어오는 스레드의 priority를 재계산하는 함수이다. 이 함수는 4 tick마다 현재 스레드의 priority를 계산할 때, 1초 마다 모든 스레드의 priority를 계산할 때, thread\_set\_nice에서 현재 스레드의 nice값이 변했을 때 그에 맞게 priority를 업데이트 해줄 때 호출한다. priority의 최대/최솟값을 벗어나지 않게 주의한다.

(3-4) src/thread/thread.c/void mlfqs\_recent\_cpu(struct thread \*t)

|  |
| --- |
| **void mlfqs\_recent\_cpu(struct thread \*t)**  **{**  **if (t != idle\_thread)**  **t->recent\_cpu = fp\_add\_int(fp\_mult\_fp(fp\_div\_fp(fp\_mult\_int(load\_avg, 2), fp\_add\_int(fp\_mult \_int(load\_avg, 2, 1)), t-> recent\_cpu), t->nice);**  **}** |

**recent\_cpu = (2 \* load\_avg) / (2 \* load\_avg +1 )\*recent\_cpu + nice** 에 맞게 인자로 넘어오는 스레드의 recent\_cpu를 재계산하는 함수이다. 이 함수는 1초 마다 모든 스레드의 recent\_cpu를 재계산할 때 호출된다.

(3-5) src/thread/thread.c/void mlfqs\_load\_avg(void)

|  |
| --- |
| **void mlfqs\_load\_avg(void)**  **{**  **int number\_of\_ready\_threads = list\_size(&ready\_list);**  **if(thread\_current() != idle\_thread)**  **number\_of\_ready\_threads++;**  **load\_avg = fp\_div\_int(fp\_add\_int(fp\_mult\_int(load\_avg, 59), number\_of\_ready\_threads), 60);**  **if(load\_avg<0) load\_avg=0;**  **}** |

**load\_avg = (59/60) \* load\_avg + (1/60) \* ready\_threads** 에 맞게 전역 변수 load\_avg를 재계산하는 함수이다. 이 함수는 1초 마다 load\_avg를 업데이트 해줄 때 호출된다.

(3-6) src/thread/thread.c/void mlfqs\_increment(void)

|  |
| --- |
| **void mlfqs\_increment(void)**  **{**  **struct thread \* t = thread\_current();**  **if(t != idle\_thread)**  **t->recent\_cpu = fp\_add\_int(t->recent\_cpu, 1);**  **}** |

매 1tick마다 현재 실행중인 스레드의 recent\_cpu를 1씩 증가시키는 함수이다. 이 함수는 1tick마다 실행되는 timer\_interrupt에서 호출된다.

(3-7) src/thread/thread.c/void mlfqs\_recalc(void)

|  |
| --- |
| **void** **mlfqs\_recalc(void)**  **{**  **struct list\_elem \*e;**  **struct thread \*t;**  **int max\_pri;**  **for (e = list\_begin(&all\_list); e != list\_end(&all\_list); e=list\_next(e))**  **{**  **t = list\_entry(e, struct thread, allelem);**  **mlfqs\_recent\_cpu(t);**  **mlfqs\_priority(t);**  **}**  **if (!list\_empty(&ready\_list))**  **{**  **list\_sort(&ready\_list, cmp\_priority, NULL);**  **t = list\_entry(list\_front(&ready\_list), struct thread, elem);**  **max\_pri = t->priority;**  **}**  **if (thread\_current()->priority < max\_pri)**  **intr\_yield\_on\_return();**  **};** |

매 1tick마다 호출되는 timer\_interrupt에서 thread\_mlfqs라 true라면 1초(100 ticks)마다 이 함수가 호출된다. all\_list를 순회하며 모든 스레드의 recent\_cpu와 priority를 업데이트 한다. 이 함수가 불리기 전에 timer\_interrupt에서 load\_avg가 먼저 업데이트 되므로, 변경된 load\_avg에 따라 recent\_cpu가 변하고, 이 변한 recent\_cpu에 근거해 priority도 업데이트가 된다(물론 nice도 고려함).

ready list의 내용들도 함께 업데이트 되었을 테니, 새로 업데이트된 priority들을 고려하여 ready list를 sort를 한 후, 가장 큰 priority값이 현재의 스레드의 priority값보다 크다면 intr\_yield\_on\_return을 통해 interrupt가 끝나고 thread\_yield가 일어날 수 있도록 한다.

(3-8) src/thread/thread.c/void thread\_set\_nice(int new\_nice)

|  |
| --- |
| **void thread\_set\_nice(int new\_nice)**  **{**  **enum intr\_level old\_level = intr\_disable();**  **struct thread \* t = thread\_current();**  **t->nice = new\_nice;**  **mlfqs\_priority(t);**  **intr\_set\_level(old\_level);**    **test\_max\_priotity();**  **}** |

현재 스레드의 nice값을 변경하는 함수이다. 변경 시 interrupt는 비활성화 해야 하며, 새로운 nice값을 반영하여 현재 스레드의 priority를 계산하고, 우선순위에 따른 선점이 필요한지 테스트하여 필요하면 yield를 한다.

(3-9) src/thread/thread.c/int thread\_get\_nice(void)

|  |
| --- |
| **int thread\_get\_nice(void)**  **{**  **return thread\_current()->nice;**  **}** |

현재 스레드의 nice값을 반환한다.

(3-10) src/thread/thread.c/int thread\_get\_load\_avg(void)

|  |
| --- |
| **int thread\_get\_load\_avg(void)**  **{**  **return fp\_to\_int\_round(fp\_mult\_int(load\_avg, 100));**  **}** |

현재의 load\_avg를 반환하는데, 100배 후 정수로 올림 하여 반환한다.

(3-11) src/thread/thread.c/int thread\_get\_recent\_cpu(void)

|  |
| --- |
| **int thread\_get\_recent\_cpu(void)**  **{**  **return fp\_to\_int\_round(fp\_mult\_int(thread\_current()->recent\_cpu, 100));**  **}** |

현재 스레드의 recent\_cpu값에 100배 후 정수로 올림 하여 반환한다

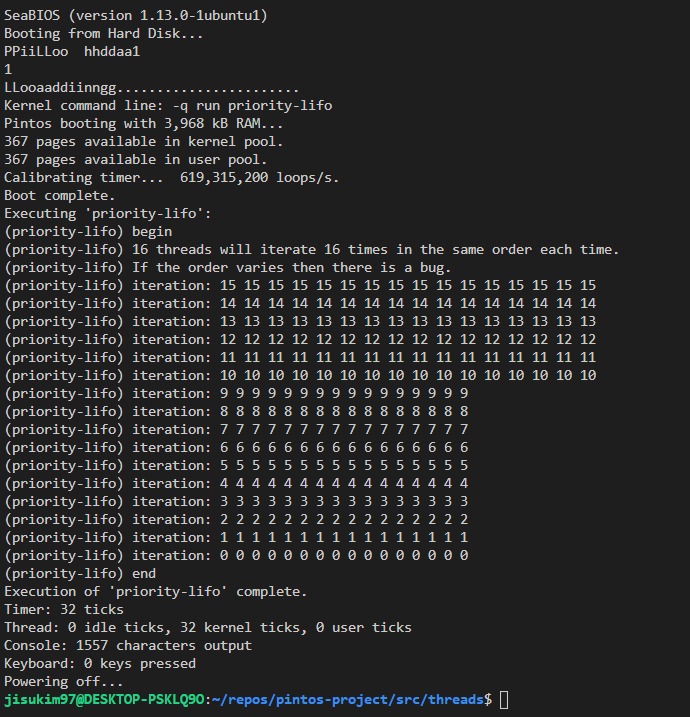
(4) src/devices/timer.c/void timer\_interrupt(struct intr\_frame \* args UNUSED)

|  |
| --- |
| **void timer\_interrupt(struct intr\_frame \* args UNUSED)**  **{**  **ticks++;**  **thread\_awake(ticks);**  **if(thread\_mlfqs)**  **{**  **mlfqs\_increment();** // every 1 tick  **if(timer\_ticks() % TIMER\_FREQ==0)** // every 1 sec  **{**  **mlfqs\_load\_avg();**  **mlfqs\_recalc();**  **}**  **if(timer\_ticks() % 4==0)** // every 4 ticks  **mlfqs\_priority(thread\_current());**  **}**  **thread\_tick();**  **};** |

매 1 tick마다 실행되는 timer\_interrupt에 thread\_mlfqs가 true일 때 매 1 tick마다 현재 스레드의 recent\_cpu를 증가시키는 함수(mlfqs\_increment)를 호출하고, 매 4 tick마다 현재 스레드의 priority를 재계산하는 함수(mlfqs\_priority) 호출한다.

* 1. **시험 및 평가 내용**

1. priority-lifo.c 코드 및 priority-lifo 테스트 결과 분석



1. make check 결과 캡쳐

