**Pintos Project 3: Threads**

담당 교수 : 서강대학교 컴퓨터공학과 김영재 교수

학번 / 이름 : 20181657 오정연

개발 기간 : 21.10.15 ~ 21.11.06

1. **개발 목표**

프로젝트 3에서 구현해야 할 내용은, Alarm Clock, Priority Scheduling, Advanced Scheduler(BSD Scheduler)이다. Pintos는 기본적으로 busy-waiting 방식으로 대기하며 Round Robin 스케쥴링 방식을 사용해서 thread를 실행한다. 따라서 Alarm Clock방식을 구현하여, thread에 wakeup tick을 추가하여 해당 tick 이상의 시간이 지나면 unblock을 해주는 방식을 구현하였다. 그리고 스케쥴링에서 우선순위를 고려하는 방식을 구현한다. 추가적으로 가장 우선순위가 높은 큐부터 thread를 선택하는 BSD 스케쥴러를 구현한다.

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

Pintos는 기본적으로 timer\_sleep 함수를 통해 thread를 sleep하게 했다. 이 함수 내에서 계속적으로 시간이 얼마나 지났는지 확인하며, thread\_yield()함수를 호출하는데, 이는 thread의 상태가 RUNNING과 READY state를 오가야 했기 때문에, 비효율적이다. 따라서 일정 시간(tick)이 지난 후에는 BLOCKED state로 바꿔주는 방식을 구현한다. 여기서 sleeping\_list라는 큐 자료구조를 통해, wake up time의 정보와 함께 삽입한다. 따라서 timer\_interrupt()함수를 통해 현재까지 지난 시각(tick)과 wake up time을 비교하여 적절히 wake up 시킨다.

* 1. Priority Scheduling

기존의 thread를 Round-robin 스케쥴링 했던 방식은, 우선순위를 고려하지 않았다. 따라서 이를 반영하기 위해서는, ready\_list queue에 thread를 우선순위에 따라 내림차순으로 삽입해주는 방식을 구현한다. 따라서, thread간의 우선순위를 반영한 스케쥴링이 가능해지고, 새롭게 생성된 스레드의 우선순위가 실행되고 있는 스레드보다 높을 경우 thread\_yield()도 가능해진다.

* 1. Advanced Scheduler

BSD 스케쥴러는 MLFQ(multi-level feedback queue), 또는 MLRQ(multi-level ready queue)와 같이, 우선순위가 높은 큐에서 thread를 우선적으로 선택한다. 여기서 각 ready\_queue는 Round-robin 스케쥴링을 따르고, MLFQ는 큐 64개 혹은 1개의 큐를 선택할 수 있는데, 이번에는 한 개의 큐로 구현하였다.

* 1. **개발 내용**

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

스레드가 BLOCKED 상태가 되면 timer\_sleep()함수에 의해 sleeping list에 삽입된다. 그 후, 매 tick 마다 timer\_interrupt()가 호출이 된다. 이때, 함수 내에서 sleeping list의 요소들을 탐색하며 깨울 스레드를 찾는다. 깨워야 할 대상은 현재 tick보다 저장된 wake up tick이 이하일 경우에 해당한다. 따라서, 이 대상자는 sleeping\_list에서 삭제하고, thread\_unblock()함수를 통해 스레드를 깨우는 다음 단계로 넘어간다. Thread\_unblock() 함수에서는 ready\_list로 해당 스레드를 삽입한다.

1. Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 priority scheduling에 따르면 어떻게 해야 하는지 서술.

실행중인 스레드보다 우선순위가 높은 스레드가 삽입된다면, thread\_yield()함수를 실행한다. 이 함수 내에서는 실행중인 스레드가 idle\_thread가 아닌 경우, 다시 ready\_list 큐에 우선순위에 맞게 삽입한다. 그 후에, schedule()를 호출하여 다음 차례로 실행되어야 할 스레드를 새롭게 찾아서 해당 프로세스를 실행한다.

1. Advanced Scheduler에서 priority 계산에 필요한 각 요소를 서술.

스레드는 생성될 때 우선순위가 초기화(31)된 후, 4 tick 마다 우선순위를 갱신해주어야 한다. 우선순위를 계산하는 식은 다음과 같다.

priority = PRI\_MAX – (recent\_cpu / 4) – (nice \* 2)

여기서 PRI\_MAX는 priority의 최댓값(63)이다. recent\_cpu의 경우 스레드가 가장 최근에 쓰인 CPU time을 계산한다. 처음에 선언되었을 때는 0으로 초기화되고, 자식 스레드라면 부모 스레드의 값을 상속받는다. 그리고 time\_interrupt()함수 내에서 다시 계산되는데, idle\_thread가 아니면서 RUNNING 상태의 스레드라면 recent\_cpu 값이 1씩 증가한다. 이를 계산하는 수식은 다음과 같다.

recent\_cpu = (2 \* load\_avg) / (2 \* load\_avg + 1) \* recent\_cpu + nice

여기서 load\_avg는 READY 상태에 있는 스레드 개수의 평균을 말한다. 다음과 같은 수식으로 계산된다.

load\_avg = (59/60) \* load\_avg + (1/60) \* ready\_threads

ready\_threads는 READY나 RUNNING 상태에 있는 스레드의 개수다. 위에서 계속 쓰이는 nice 변수는 스레드의 구조체 내에 선언되어 있으며, -20 과 20 사이의 범위를 가진다. nice 값이 양수이면 우선순위를 낮추고 다른 스레드가 CPU를 사용할 수 있게 한다. 따라서 값이 0이 되면 우선순위에 영향을 미치지 않는다. 따라서 초기값이 0으로 설정된다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

10/15 ~ 10/18 : 강의자료와 핀토스 메뉴얼을 읽으며 구현내용 파악

10/19 ~ 10/24 : Alarm Clock 구현

10/25 ~ 10/28 : Priority Scheduling 구현

10/29 ~ 10/30 : 소수점 연산 구현

10/31 ~ 11/03 : Advanced Scheduler 구현

11/04 ~ 11/05 : make check 및 디버깅

11/06 : 보고서 작성

* 1. **개발 방법**
  + 수정하거나 추가해야 하는 자료구조

thread.h의 struct thread 내부에 wakeup\_tick, nice, recent\_cpu 변수를 추가한다.

thread.c에서 새롭게 추가된 함수들 ( thread\_aging, cal\_recent\_cpu, cal\_priority )를 선언해준다.

* + 수정해야하는 소스코드

- /src/devices : timer.c

timer\_sleep() : 현재 스레드의 wakeup\_tick을 저장하고, BLOCKED 스레드들을 저장하는 자료구조(큐)인 sleeping\_list에 삽입한다. 그 후에 thread\_block()을 호출한다.

timer\_interrupt() : 매 tick마다 sleeping\_list에서 깨워야 할 대상이 되는 스레드가 있는지 wakeup\_tick과 비교한다. 그리고 priority aging을 위해 recent\_cpu와 load\_avg, priority를 다시 계산하는 작업도 해준다.

-/src/threads : thread.c

thread\_create(): 실행중인 스레드보다 생성된 스레드의 우선순위가 높을 경우 thread\_yield()를 수행한다.

thread\_unblock(): 스레드를 ready\_list에 우선순위에 맞게 넣는다. (priority scheduling / advanced scheduler )

thread\_yield(): 실행 중인 스레드보다 높은 우선순위의 스레드가 존재하므로, 실행 중인 스레드는 ready\_list에 우선순위에 맞게 넣어주고, 다시 scheduling한다.

thread\_set\_priority(): 현재 스레드의 우선순위를 새로운 우선순위로 설정한다.

thread\_get\_priority(): 현재 스레드의 우선순위를 리턴한다.

thread\_set\_nice(): 현재 스레드의 nice 값을 갱신하고, 우선순위도 다시 계산해준다.

thread\_get\_nice(): 현재 스래드의 nice 값을 리턴한다.

thread\_get\_load\_avg(): 전역변수로 선언된 load\_avg 값을 리턴한다.   
thread\_get\_recent\_cpu(): 현재 스레드의 recent\_cpu 값을 리턴한다.

- /src/threads : synch.c

sema\_down(): sema 구조체 안에는 waiters 리스트가 존재한다. sema 구조체에는 우선순위 개념이 없으므로, 직접 thread의 우선순위를 비교하며 해당 요소를 삽입한다.

sema\_up(): sema->waiters 리스트 안에서 가장 높은 우선순위를 갖는 요소를 watiers리스트에서 삭제한 후, 이를 thread\_unblock()한다.

* + 수정하거나 추가해야 하는 함수

-/src/threads : thread.c

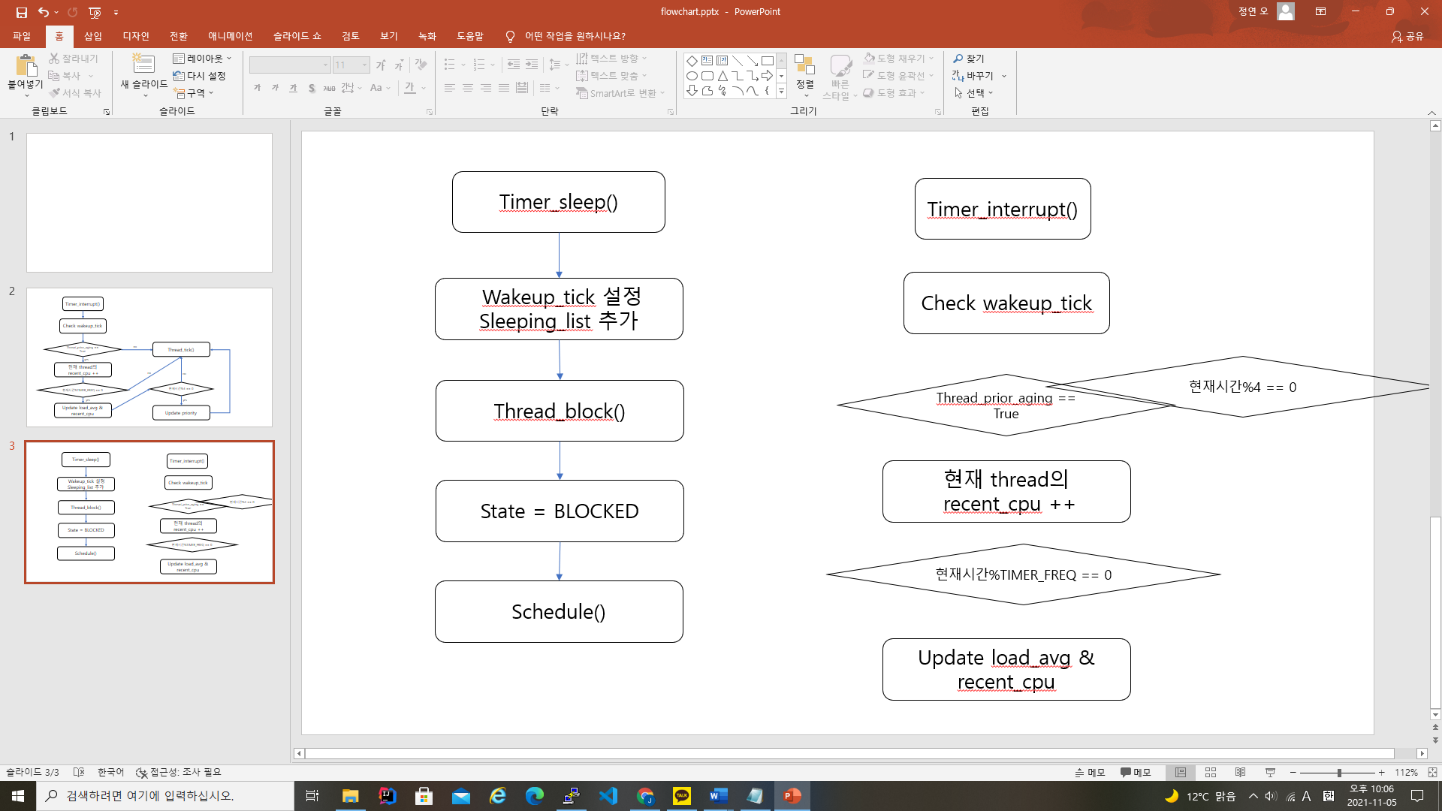
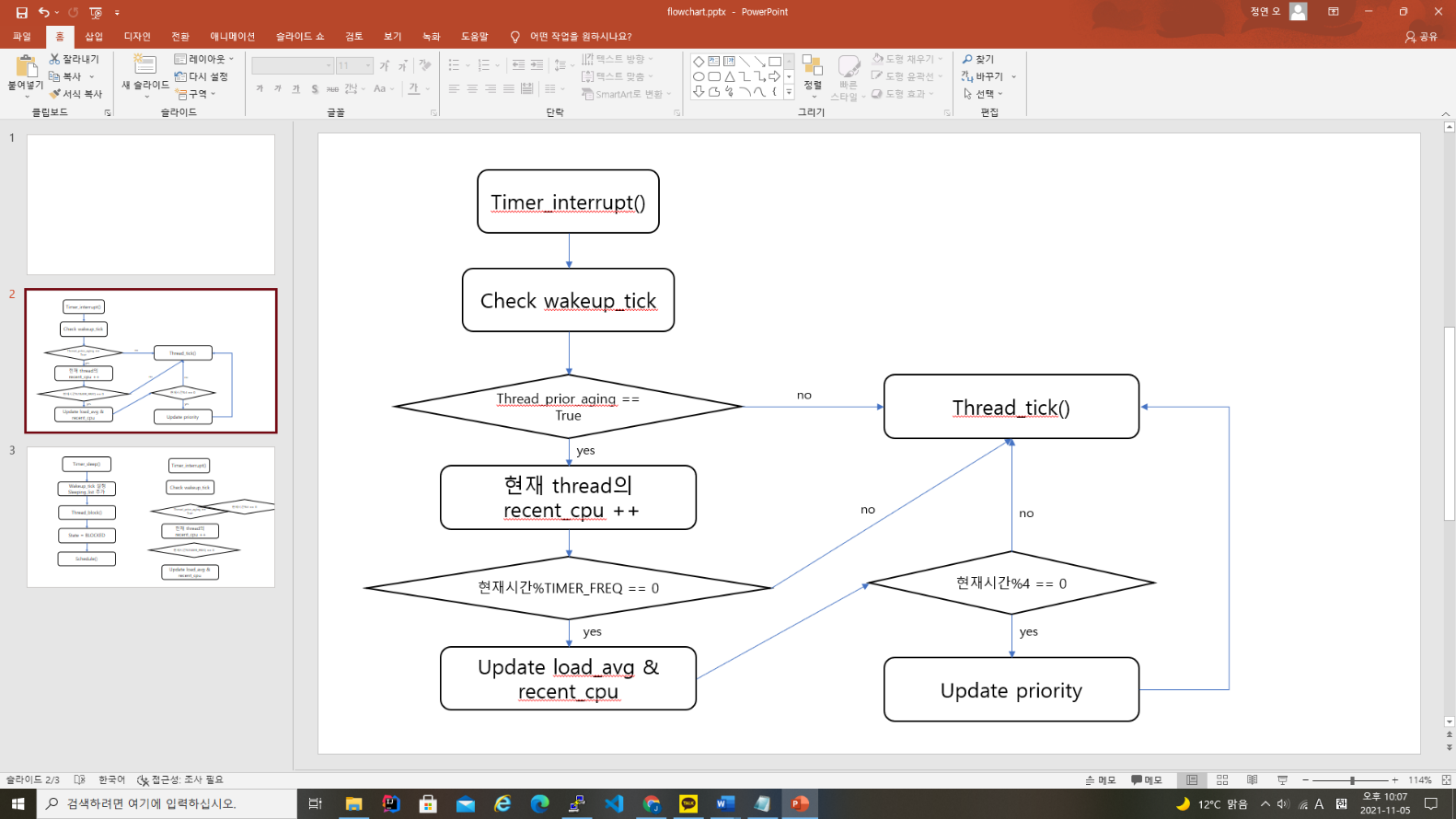
cal\_recent\_cpu(): 모든 스레드의 recent\_cpu값을 다시 계산하여 저장하고, 전역변수로 선언된 load\_avg값을 갱신한다.

cal\_priority(): 모든 스레드의 우선순위를 다시 계산하여 저장한다.

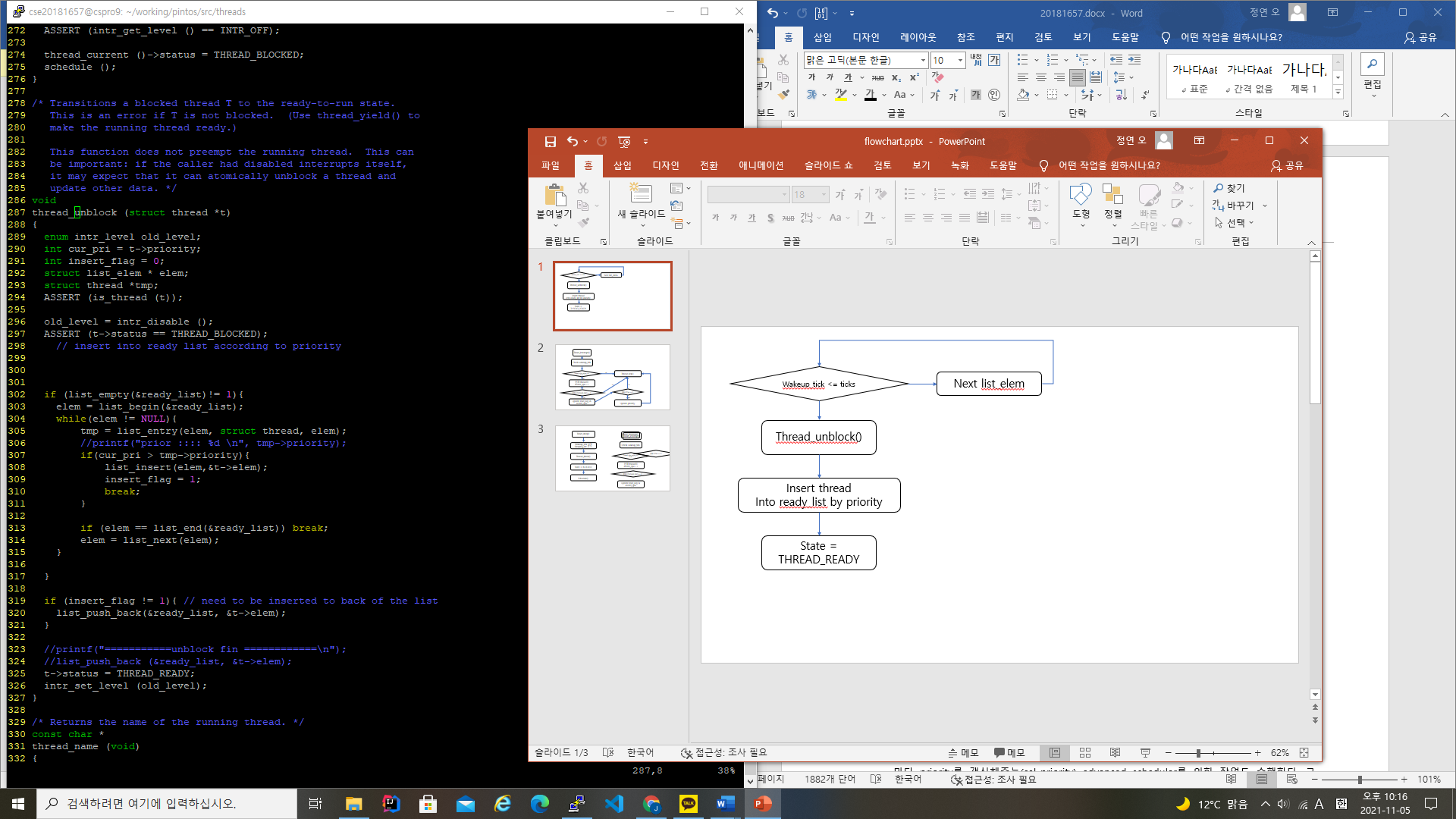
thread\_aging(): 시간이 지남에 따라 스레드의 우선순위를 증가시켜준다.

float\_mul\_float(): 소수점 연산이 지원되지 않는 핀토스에서 fixed-point number 표현법으로 실수끼리의 곱을 연산하는 함수이다.

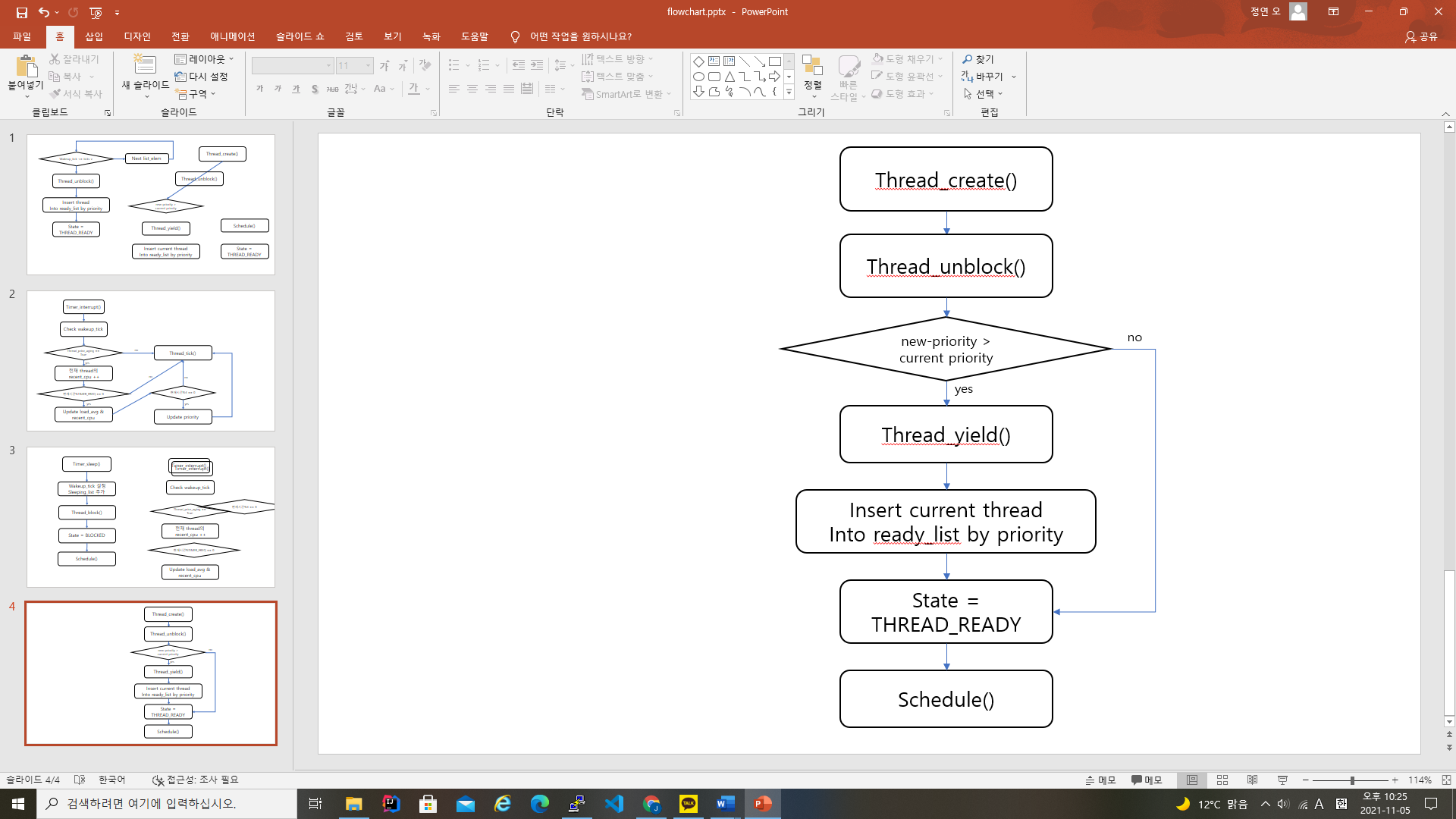
float\_div\_float(): 소수점 연산이 지원되지 않는 핀토스에서 fixed-point number 표현법으로 실수끼리의 나눗셈을 연산하는 함수이다.

1. **연구 결과**
   1. **Flow Chart**
      1. Alarm Clock
   2. Timer\_sleep
   3. Timer\_interrupt()
      1. Priority Scheduling

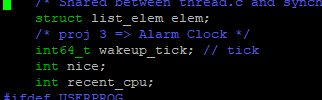
2-1. Thread wake up



2-2. Thread yield



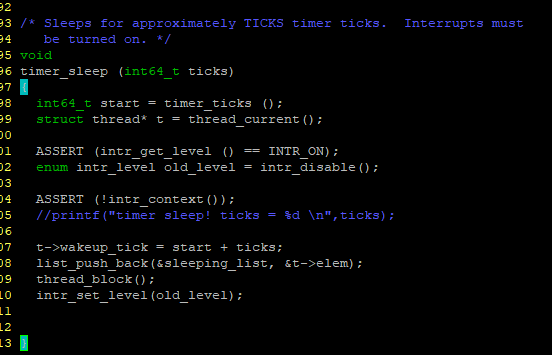
* 1. **제작 내용**
  2. **프로젝트를 위해 추가된 변수들**



스레드 구조체 내부에 이번 프로젝트에서 필요한 변수들을 선언해주었다.

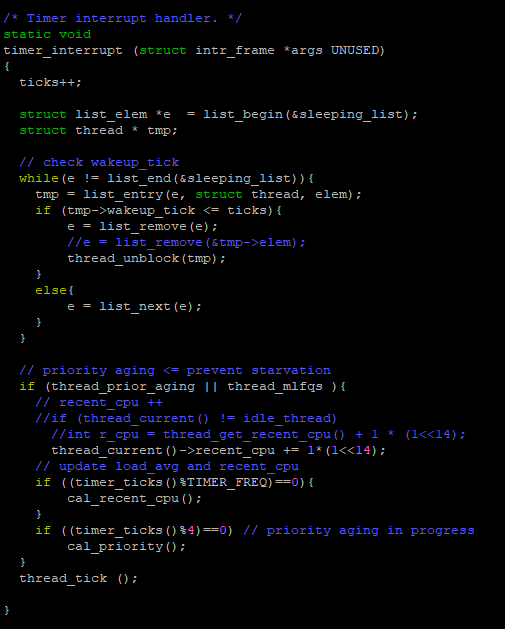


Timer.c 내부에 전역적으로 선언한 리스트이다. BLOCKED 상태인 스레드를 담고 있다.

* 1. **Timer.c 내부 함수들**
* timer\_sleep()

스레드를 BLOCKED state로 변경하는 함수이다. 따라서 스레드의 wakeup\_tick을 현재 시각(tick) + 해당 스레드의 wake up tick 을 해주어서 저장한다. 그 이유는. Timer-interrupt()에서 비교할 때 현재 시각(tick)과 비교하기 때문이다. 그 후에 해당 스레드의 elem을 sleeping\_list에 삽입한다. 해당 스레드의 block작업을 위해 thread\_block()을 호출한다.

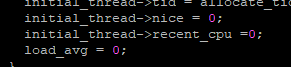
* timer\_interrupt()



이 함수 내에서는 두 가지 작업을 수행한다. 첫번째로는, BLOCKED 상태인 스레드 중 깨어날 시간이 된 스레드를 찾아 unblock하는 것이다. 두번째로는, priority aging을 위하여 현재 스레드의 recent\_cpu를 1씩 증가한다. 그리고 timer.h에 정의된 TIMER\_FREQ에 따라 recent\_cpu와 load\_avg 값을 갱신하는 함수(cal\_recent\_cpu)를 호출한다. 그 다음 4 ticks 마다 priority를 갱신해주는(cal\_priority) advanced scheduler를 위한 작업도 수행한다. 그 후에 thread\_tick()을 호출해준다.

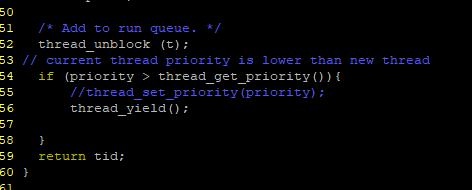
2) Thread.c 내부 함수들

* thread\_init()



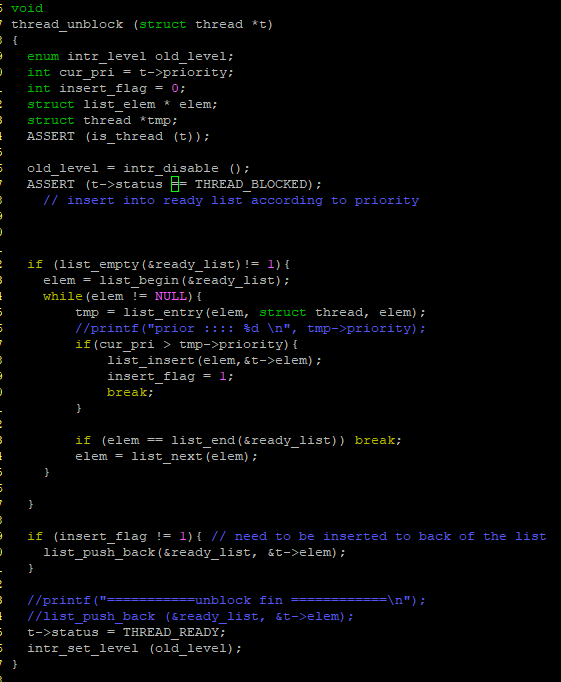
새롭게 추가된 변수들을 초기화한다.

* thread\_create()



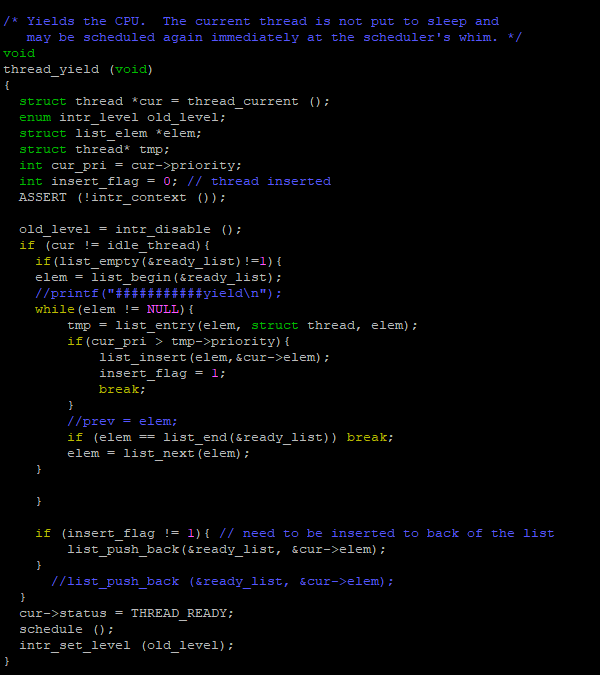
현재 스레드의 우선순위(Thread\_get\_priority)와 새롭게 생성된 스레드의 우선순위(priority)를 비교하여 새로운 스레드의 우선순위가 높을 경우, thread\_yield()를 통해 priority scheduling을 수행한다.

* Thread\_unblock()



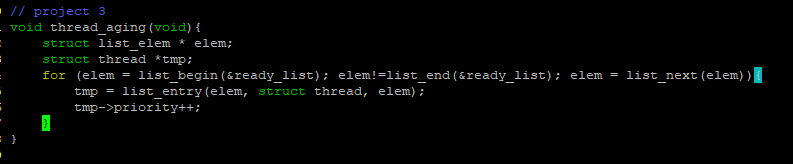
기존에는 Round-Robin 스케쥴링을 수행하였으므로 list\_push\_back()만을 수행하고 있었다. 그러나 priority scheduling을 수행하기 위해 우선순위 내림차순으로 저장할 수 있도록 다음과 같이 리스트를 순회하여 자신보다 우선순위가 작은 스레드를 만나면 그 앞에 넣을 수 있도록 list\_insert()함수(list.c 내장함수)를 호출하도록 설계하였다. 스레드가 정상적으로 ready\_list에 삽입되면 insert\_flag를 set하여 그렇지 않은 경우에는 list\_push\_back()함수(list.c 내장함수)를 통해 맨 뒤에 (우선순위가 가장 낮은 스레드에 해당하므로) 삽입한다. 위의 작업이 수행되었다면 스레드의 status를 THREAD\_READY상태로 변경해준다.

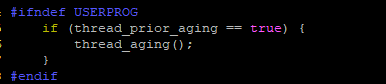
* Thread-yield()



해당 함수는 현재 실행중인 스레드보다 우선순위가 높은 스레드가 있을 때 CPU를 yield하는 작업을 수행한다. 이를 위해서는 현재 스레드를 다시 ready\_list에 삽입하기 위해 우선순위를 고려하여 적절한 위치를 찾아 삽입한다. 이 과정은 위의 thread\_unblock()에서 우선순위를 고려하여 삽입하는 과정과 동일하다. 다만, 차이점은 schedule()함수를 호출하여 우선순위가 가장 높은 스레드를 RUNNING 할 수 있게 다시 스케쥴링 한다.

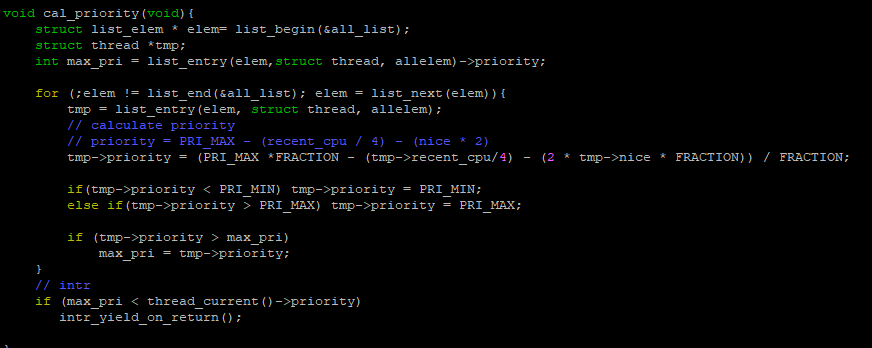
* Thread\_aging



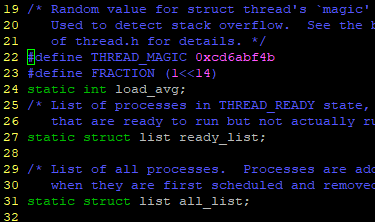


우선순위를 고려한 스케쥴링을 할 때, 우선순위가 상대적으로 낮은 스레드의 경우에는 계속 실행할 기회가 주어지지 않는 starvation 현상이 발생한다. 이를 방지하기 위해 thread\_aging()함수 내에서 일정 시간(tick)이 지나면 ready\_list에 들어 있는 스레드의 우선순위를 높여 이를 방지한다.

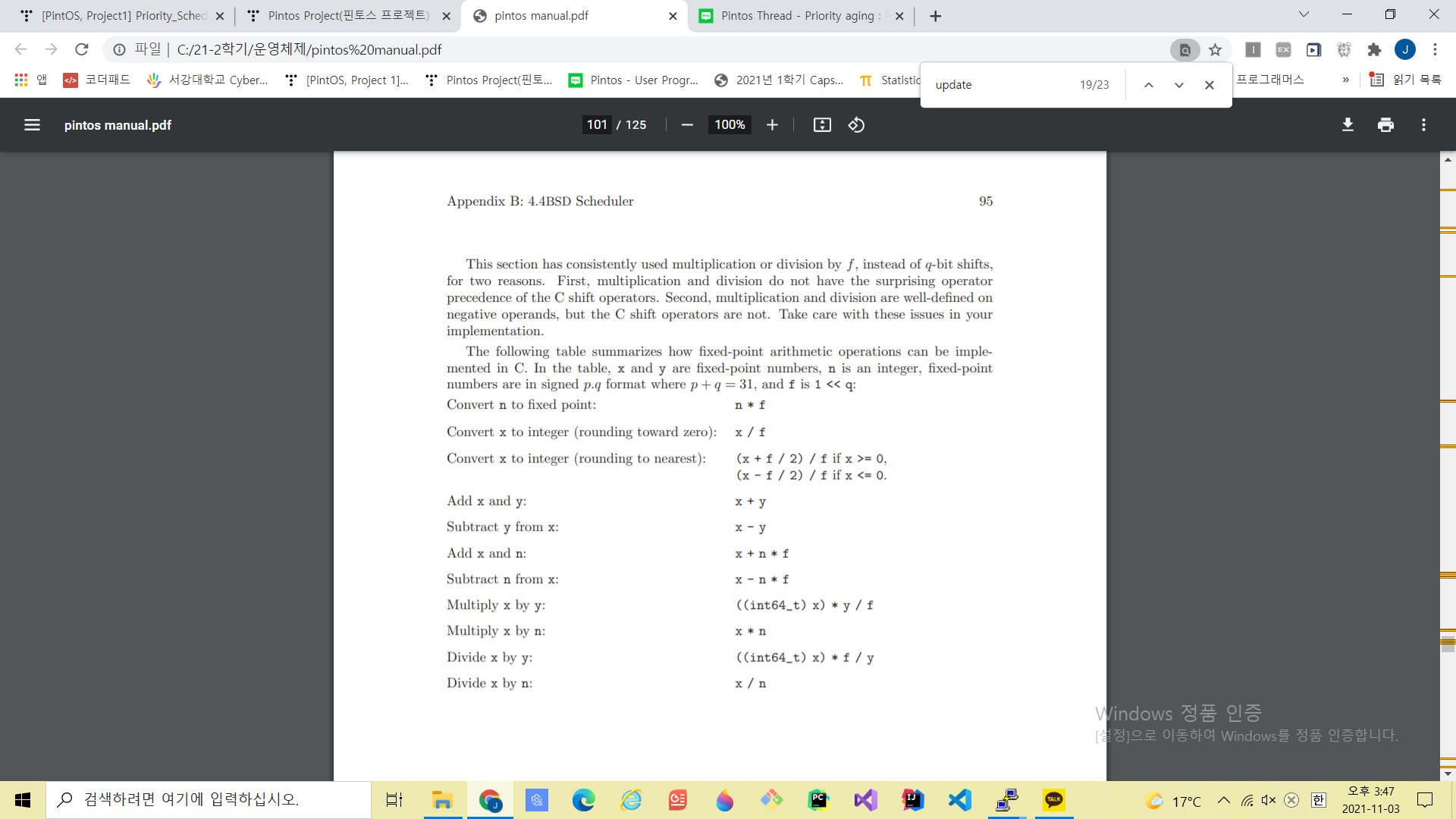
* Cal\_priority : 스레드의 priority를 갱신하는 함수



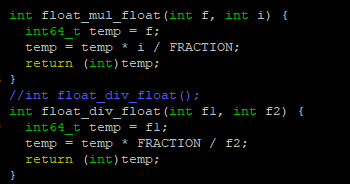
강의자료에 따르면, advanced scheduler의 경우 매 4 tick마다 모든 스레드는 priority를 다시 계산해주어야 한다. 따라서 이를 계산하는 함수로, all\_list에 들어있는 스레드의 우선순위를 주어진 수식에 따라 계산한다. 만약 계산한 우선순위가 사전에 정의한 우선순위의 범위(0 ~ 63)를 초과할 경우, 적절하게 PRI\_MIN 혹은 PRI\_MAX 값으로 설정하였다. 그리고 저장된 스레드 중 우선순위 최댓값을 찾아서 현재 스레드의 우선순위보다 높을 경우 intr\_yield\_on\_return(intr 도중의 yield)을 수행하였다. 여기서 주의할 점은 recent\_cpu와 load\_avg는 실수로 표현되는데, 핀토스에서는 소수점 연산을 지원하지 않기 때문에, fixed-point real arithmetic을 수행해야 한다.



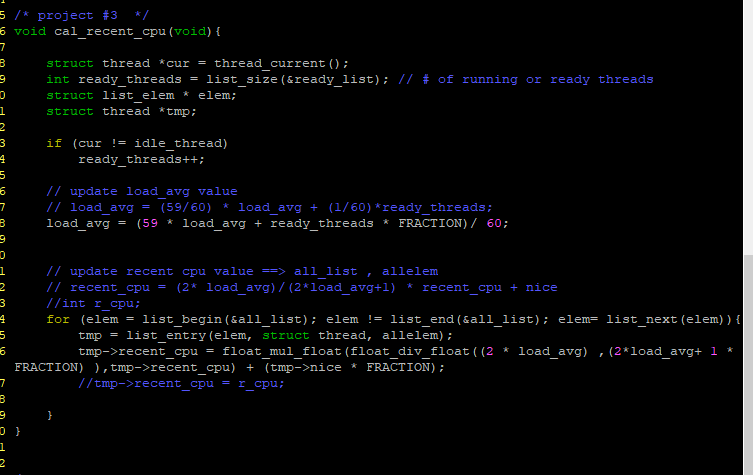
따라서 fixed-point 포맷을 살펴보면 32bit로 첫번째 비트는 sign bit, 중간의 17비트는 정수부분, 나머지 14비트는 소수점을 표현한다. 따라서 이 소수점 부분을 추출하기 위해 14bit shift left 연산을 해야 한다. 자세한 내용은 핀토스 매뉴얼을 참고하였다.



부동소수점 연산을 지원하지 않는 핀토스에서 계산을 하기 위해, 다른 연산은 직접 코드로 작성(FRACTION를 곱하는 방식)하였으나, 이와 같이 실수를 이용한 곱셈 및 나눗셈 연산은 함수로 구현해보았다. 후에 든 생각이지만 다른 연산들도 함수로 선언해도 상관은 없을 것 같다.

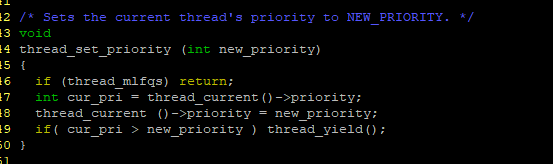


* Cal\_recent\_cpu : recent\_cpu와 load\_avg 값을 갱신하는 함수



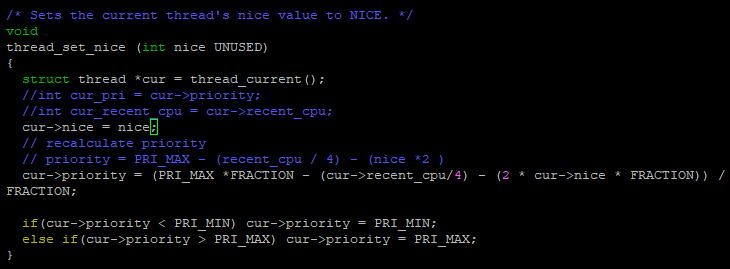
해당 함수는 recent\_cpu 뿐만 아니라 전역변수 load\_avg 값도 계산하는 함수이다. Recent\_cpu의 수식에 load\_avg와 ready\_threads 값이 필요하므로 앞부분에 계산한다. Load-avg를 계산할 때에도 소수점 연산을 해야 하므로 FRACTION(1<<14)으로 바꾸어서 계산해준다. Recent\_cpu는 스레드의 최근 CPU time을 계산하므로, 매 tick마다 RUNNING 상태라면 1씩 증가하는 작업은 timer\_interrupt()함수에서 수행하였다. 따라서 이 함수 내에서는 강의자료에서 명시된 수식에 따라 recent\_cpu의 값을 갱신한다.

* Thread\_set\_priority



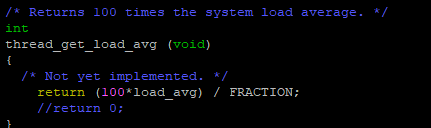
주석에 나와있는 설명대로 현재 스레드의 priority를 새롭게 입력으로 받은 priority로 갱신한다. 이때 실행중인 스레드의 우선순위보다 높다면 preempt를 통해 thread\_yield()를 수행한다.

* Thread\_get\_priority : 현재 스레드의 우선순위를 리턴하는 함수로, 코드가 간단하므로 생략한다.
* Thread\_set\_nice



현재 스레드의 nice 값을 new\_nice로 바꾸어지는 함수이다. 새로운 값에 의해 현재 스레드의 우선순위도 다시 계산해 값을 바꾸어 준다. 이를 실행중인 스레드의 우선순위와 비교하여 새로운 우선순위의 값이 더 클 경우 thread\_yield()를 수행한다.

* Thread\_get\_nice : 현재 스레드의 nice 값을 리턴하는 함수로, 코드가 간단하므로 생략한다.
* Thread\_get\_load\_avg

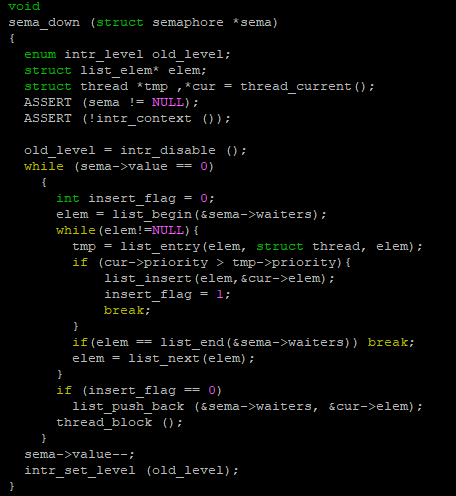


Load\_avg의 100배를 리턴하는데, load\_avg 값이 실수이므로 FRACTION으로 나누어 주었다.

* Thread\_get\_recent\_cpu

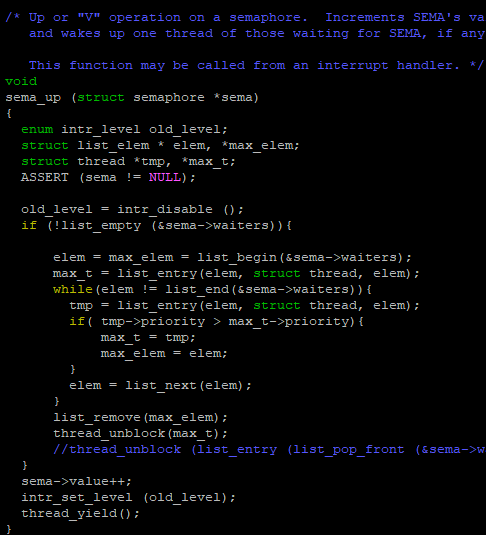
위의 thread\_get\_load\_avg와 동일하게 recent\_cpu의 100배 값을 리턴한다. 이 값도 실수이므로 동일하게 작성하였다. 코드는 생략한다.

* + 1. **Synch.c 내부 함수들**
* Sema\_down()



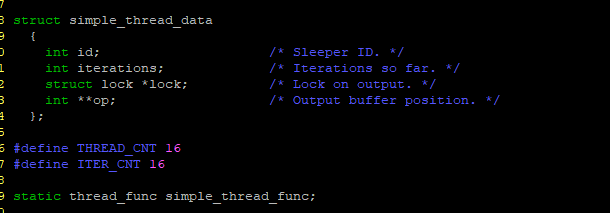
Semaphore 구조체 내에는 priority 변수가 없으므로, sema안의 waiters 리스트에서 직접 스레드의 우선순위를 비교하며 적절한 위치에 요소를 삽입한다. 해당 스레드가 제대로 waiters 리스트에 삽입된 후에는 thread\_block()을 수행한다.

* Sema\_up()

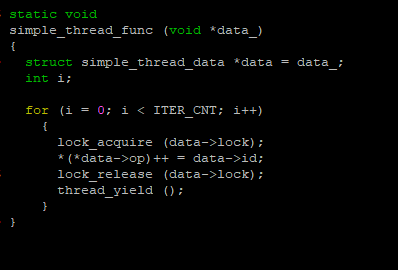


Sema\_up을 수행하기 위해서 우선순위가 제일 높은 스레드를 찾아야 하므로 waiters리스트를 순회하며 max priority를 가지는 스레드와 요소를 찾는다. 최종적으로 찾은 요소를 waiters 리스트에서 삭제한 후 thread\_unblock()을 수행한다. Sema\_down()에서 우선순위에 맞게 리스트를 삽입했지만 sema\_up에서도 우선순위를 비교하도록 이중작업을 진행하였다.

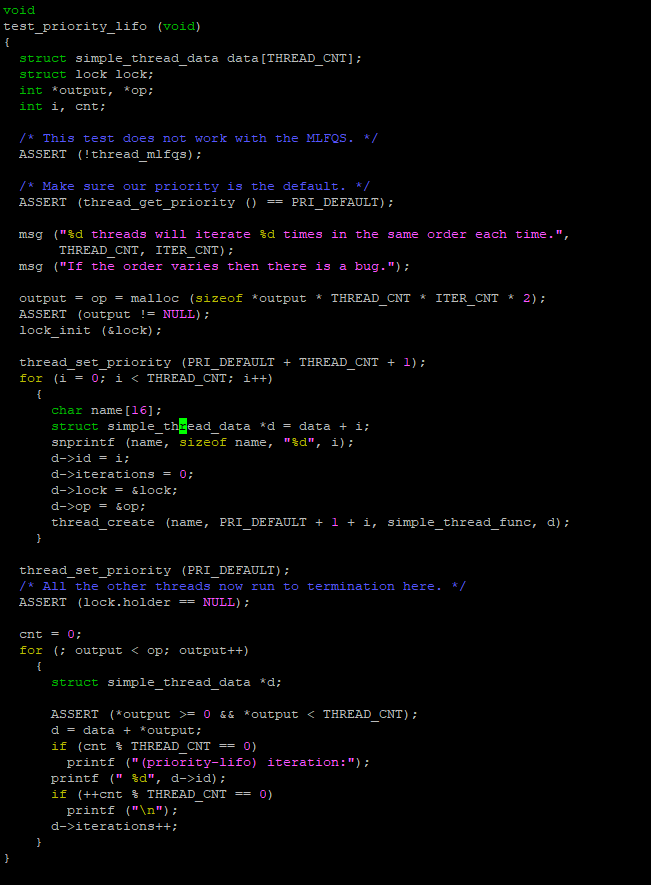
* 1. **시험 및 평가 내용**
* priority-lifo.c 코드 분석



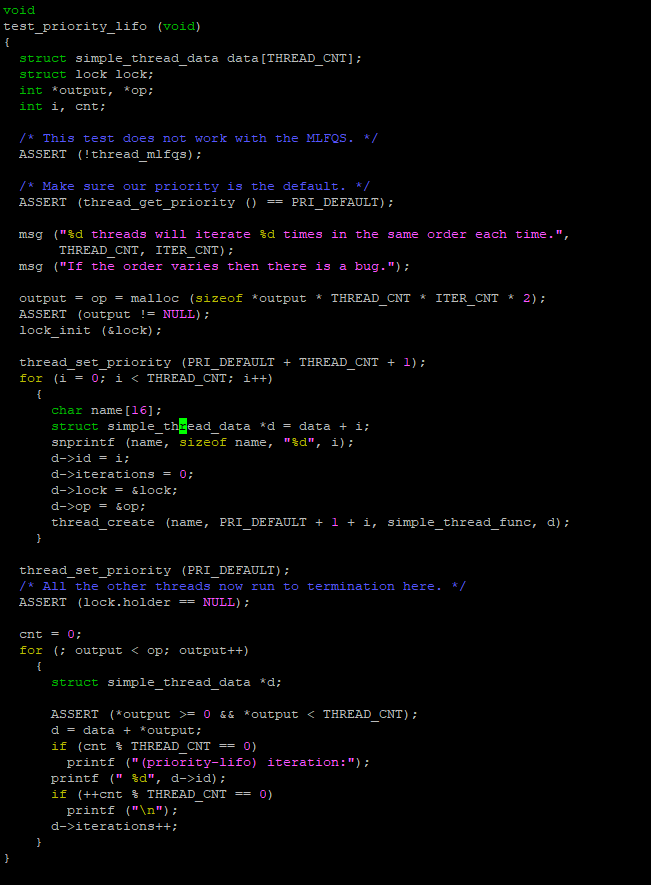
해당 파일 안에서는 simple\_thread\_data라는 구조체가 사용될 예정으로 보인다. 관련 설명은 옆에 주석으로 달려있는 것을 확인할 수 있다.



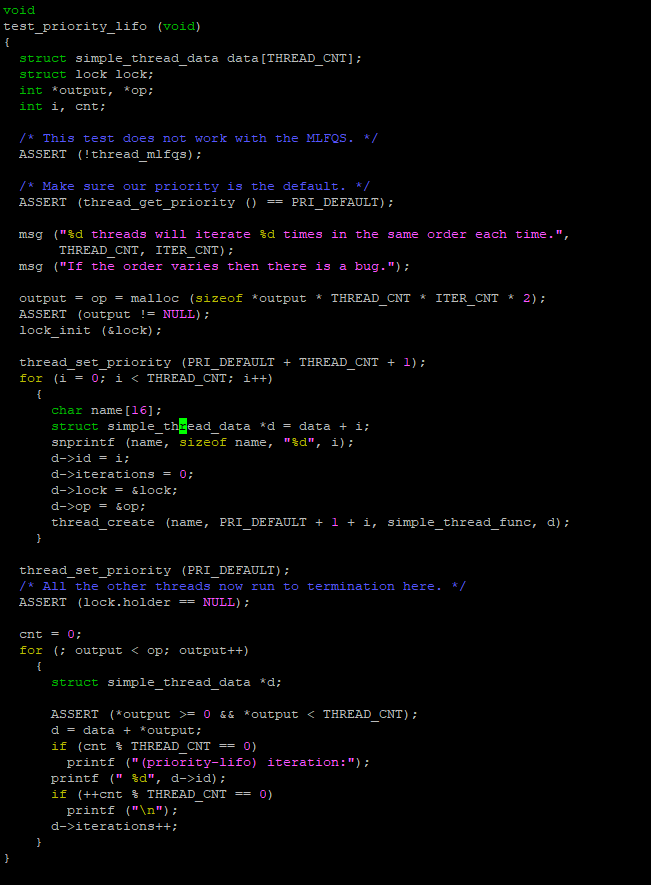
해당 함수는 simpe\_thread\_data 구조체를 사용하여 구조체 포인터를 \*data로 선언하였다. 그리고 ITER\_CNT(16)번 동안 data->id를 op에다가 저장해주었는데 이는 구조체의 output buffer에 해당한다. 따라서 \*data->op에는 data->id가 ITER\_CNT(16)번 작성되었다. 이 과정에서 다른 스레드의 접근을 막기 위해 락(lock)을 사용하였고, 이 이후에 thread\_yield()를 호출하여 새롭게 리스트의 우선순위를 정리하고 다시 스케쥴링한다.



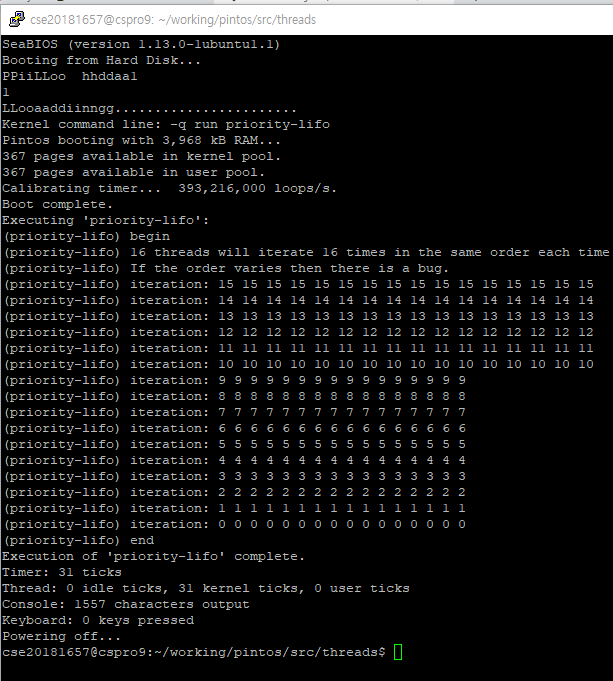
코드를 살펴보면 MLFQS에서는 작동이 되지 않는다는 것을 알 수 있고, 16개의 스레드가 16번의 iteration동안 같은 오더에서 반복됨을 알려준다. Output과 op 변수는 int형의 THREAD\_CNT(16)\*ITER-CNT(16)\*2 만큼의 메모리를 동적 할당해주었다. 그 후에 락(lock)을 걸어주었다.



그리고는 현재 스레드를 PRI\_DEFAULT+THREAD\_CNT +1로 초기화 해주었다(thread\_set\_priority). 그 다음 for문을 통해 16개의 스레드에 대하여 각 구조체 변수들을 초기화 해주고, thread\_create를 실행한다. 여기서 앞서 설명했던 simple\_thread\_func를 넣어주는 것으로 보아, output buffer에 16번 반복적으로 해당 data->id가 저장될 것을 짐작할 수 있다. 그리고 생성되는 스레드의 우선순위는 PRI\_DEFAULT + I = 1 이므로 1개씩 우선순위가 커지는 방식으로 생성될 것이다. 이렇게 16개의 스레드를 생성했다면(for 반복문 종료) 현재 스레드를 다시 PRI\_DEFAULT로 우선순위를 강등한다(thread\_set\_priority).

그 다음 Lock.holder == NULL이므로, 주석의 설명과 같이 현재 스레드보다 우선순위가 높은 다른 스레드가 RUNNING할 수 있게 된다. 따라서 우선순위가 높은 스레드부터 실행될 텐데, for문도 ASSERT(\*output >= 0 && \*output < THREAD\_CNT)를 통해 16번의 반복을 할 것을 알게 된다. 그런데 simple\_thread\_data 구조체 포인터 d에 data+\*output;을 통해 해당 구조체 리스트의 주소를 담고 있다는 것을 알 수 있다. 따라서 해당 스레드에 대하여 id는 ITER\_CNT(16)번 만큼 출력된 후 새로운 스레드로 넘어가게 된다. 따라서 (priority-lifo) iteration도 THREAD\_CNT(16)만큼 출력된다. 따라서 아래와 같은 출력결과를 얻게 된다.

* priority-lifo 테스트 결과



* make check 수행 결과

