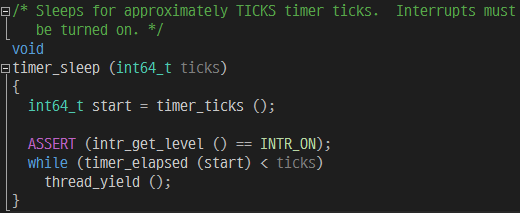
**Project 1 Design Report**

성해빈, 송창근

**1. 문제 해결**

1. Alarm Clock
   1. 현재 구현의 문제

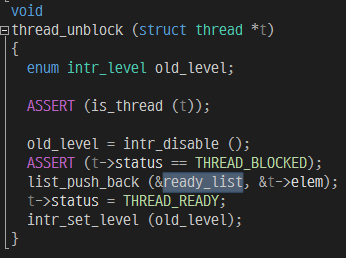


현재 timer\_sleep 함수는 매 틱마다 정한 틱이 지났는지 확인하여 지나지 않았으면 thread\_yield 함수를 호출해 자신에게 할당된 시간을 다른 스레드에게 돌린다. sleep 함수는 해당 스레드를 처리하지 않기 위해 존재하는 기능인데, 이와 같이 구현하면 스레드를 처리하지 않기 위해 계속해서 스레드에 접근해야 하는 모순이 생긴다. 이는 당연하게도 매우 비효율적이며, 성능에 악영향을 미친다.

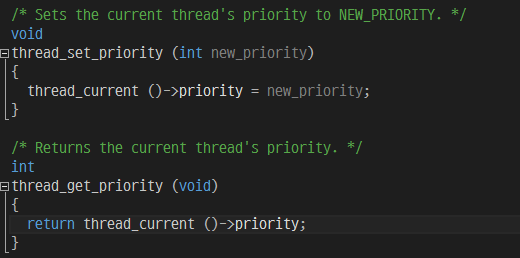
* 1. 해결 방법 및 알고리즘

thread\_yield가 아니라 thread\_block을 이용하여 스레드를 아예 정지시키고, 시간이 되면 thread\_unblock을 이용하여 풀어주도록 한다. Thread\_yield 는 계속 자동으로 ready\_list에 추가하기 때문에 상당히 비효율적이다. Sleep\_list를 따로 만들어 일정 시간이 지날 동안 ready\_list에 넣지 않고 시간이 지나야 ready\_list에 넣는다. 이를 위해서는 정지한 스레드와 깨어날 시간을 저장해야 한다. timer\_sleep 함수가 호출되면 스레드 번호와 깨어나야 할 ticks을 저장한다. 여러 스레드가 동시에 sleep 될 수 있으니 깨어날 시간이 가까운 순서로 정렬하여 리스트에 삽입한다. 이후 해당 스레드를 block한다. thread\_tick 에 현재 시스템의 ticks을 확인하여 리스트의 가장 앞에 있는 스레드의 시간이 지났으면 해당 스레드에 thread\_unblock을 하는 함수를 넣는다.

1. Priority Scheduling
   1. 현재 구현의 문제



스레드는 ready\_list에 저장된 순서로 실행된다. 그런데 현재 구현은 스레드를 ready\_list에 넣을 때, list\_push\_back 함수를 이용해서 그냥 리스트의 맨 마지막에 넣는다. 즉, 스레드는 계속 순서대로 실행된다. 그러나 실제 OS에서는 먼저 처리해야 하는 스레드들이 있다. 예를 들어, 작업 관리자 등은 우선적으로 처리해야 할 것이다. 이러한 기능을 위해 스레드에 우선순위가 있다. 우선 순위를 처리할 때 다양한 문제들이 생길 수 있는데, 이를 전부 해결해야 한다. 예를 들어, 높은 우선순위 작업을 처리하기 위해 필요한 자원을 낮은 우선순위 작업이 가지고 있다면(lock했을 때), 그 때는 낮은 우선순위 작업의 우선도를 일시적으로 올려줘야 한다.(donation)



또한, 우선순위를 제대로 구현하기 위해 우선순위를 설정하는 함수와 우선순위를 알아내는 함수가 있는데, 우선순위를 설정하는 함수의 경우, 우선순위가 바뀜으로 인해 현재 실행 중인 스레드가 최고 우선순위가 아니게 된 경우 그 스레드를 yield해야 하고, 우선순위를 받아오는 함수의 경우 donation까지 고려하게 함수를 수정해야 한다.

* 1. 해결 방법 및 알고리즘

스레드가 가질 수 있는 상태는 RUNNING, READY, BLOCK, DYING이다. 여기서 DYING는 스레드가 삭제되므로 READY 될 수 없음을 감안하면 ready\_list에 스레드가 삽입되는 경우는 thread\_yield와 thread\_unblock 때임을 알 수 있다. 해당 함수들의 list\_push\_back 부분을 list\_insert\_ordered로 새로 만들어준다. 이를 위해서는 대소비교를 하는 함수(list\_less\_func)를 만들어주어야 하는데, 스레드의 우선순위를 비교하게 하면 된다.

Donation의 경우, 우선 sema\_down을 수정한다. 이것 역시 대기열에 스레드를 집어넣을 때 list\_push\_back으로 집어넣는데, 우선순위별로 넣도록 list\_insert\_ordered를 이용하여 넣는다. 그 뒤 lock\_acquire를 수정한다. sema\_down을 시행하기 전에 현재 스레드의 번호와 우선순위를 따로 저장해두고, sema\_down이 끝났을 때 스레드 번호와 저장해 둔 스레드 번호를 비교한다. 만약 다르다면, down할 수 없었다는 의미이니 donation을 진행해야 한다. 해당 lock의 holder의 우선순위를 아까 저장해 둔 스레드 우선순위로 바꾼다. 이를 위해 set\_priority\_new(가칭) 함수를 만들어야 한다. 이 함수는 현재 스레드가 아니라 임의의 스레드의 우선순위를 바꾸고 현재 스레드를 yield한다. 이렇게 되면 lock을 하고 있는 함수가 실행되고, donation이 성공적으로 일어난다.

1. Advanced Scheduler
   1. 현재 구현의 문제

새로운 스케줄러(MLFQS)를 구현한다. 현재 구현으로는 우선순위가 자동으로 고쳐지지 않는다. 또한 –mlfqs 커널 옵션을 통해 스케줄러를 선택할 수 있도록 한다.

* 1. 해결 방법 및 알고리즘

priority = PRI\_MAX - (recent\_cpu / 4) - (nice \* 2) 가 우선순위를 결정하는 공식이다. recent\_cpu는 해당 스레드가 최근에 사용한 CPU 시간이며, nice는 해당 스레드의의 nice값이다. 자세한 것은 부록에 적혀있지만, MLFQS에서는 우선순위가 4틱마다 자동으로 계산되기에 임의로 set\_priority를 쓸 수 없고, nice 값을 조정함으로서 간접적으로 우선순위를 수정한다. 이 기능을 위해 스레드 구조체에 nice 값을 추가하고, set\_nice와 get\_nice 함수를 구현한다. 또한, thread\_tick 함수에 우선순위를 변경하는 함수를 만들어 넣는다. donation이 없는 구현도 다시 만들어야 한다..

**2. 현재 시스템 분석**

1. Thread system

스레드 시스템은 thread.c 에 정의되어 있다. 스레드 구조체에는 다음과 같은 내용이 들어있다.

* tid: 스레드를 구별하기 위한 번호
* status: 스레드의 상태. RUNNING, READY, BLOCK, DYING의 4가지.
* name: 이름(디버깅용)
* stack: 스택 포인터. 스택이 너무 커지면 오버플로우가 일어날 수 있다.
* priorty: 스레드의 우선순위
* allelem: 스레드 리스트를 만들기 위한 리스트용 원소
* elem: 두 가지 목적으로 쓰인다. thread.c에서는 실행대기 큐(ready\_list), synch.c에서는 세마포어 대기 리스트(waiters)의 원소로 쓰인다. 이는 각각 THREAD\_READY와 THREAD\_BLOCKED로 구분되기에 충돌하지 않는다.
* magic: 스택 오버플로우가 일어났는지 확인하는 용도

스레드는 다음과 같은 함수들이 처리한다.(미구현 함수 제외)

* thread\_init: 스레드를 초기화한다.
* thread\_start: 스케줄링을 시작하고 idle 스레드를 만든다. idle 스레드는 실행중인 스레드가 없을 때 실행되며, 아무 것도 하지 않는다.
* thread\_tick: 틱마다 타이머에서 호출하는 함수.
* thread\_print\_stats: 스레드 사용 통계를 출력한다.
* thread\_create: 새로운 스레드를 만든다.
* thread\_block: 스레드를 정지하고, schedule을 불러 (ready\_list의 front에 있는 thread를 running 상태로 만든다. : schedule의 기능) 다만 지금의 구현은 current thread를 따로 관리하지 않는데, 그래서 block 을 한 thread pointer를 다시 사용할 예정이라면 관리해주자.
* thread\_unblock: 인자로 받은 스레드의 정지를 해제하고 ready\_list의 끝에 넣는다.
* thread\_current: 현재 실행중인 스레드를 리턴한다.
* thread\_tid: 현재 실행중인 스레드의 번호를 리턴한다.
* thread\_name: 현재 실행중인 스레드의 이름을 리턴한다.
* thread\_exit: 현재 스레드를 삭제한다.
* thread\_yield: 현재 스레드의 CPU 점유를 종료하고 스레드를 ready\_list의 끝에 넣는다.
* thread\_get\_priority: 현재 스레드의 우선순위를 리턴한다.
* thread\_set\_priority: 현재 스레드의 우선순위를 조정한다.
* thread\_foreach: 모든 스레드에 (내가 지정하는)함수를 적용한다.
* schedule: ready\_list의 front에 있는 thread를 running 상태로 만든다. switch\_threads를 부른다.
* switch\_threads: 첫 인자의 thread에서 두번째 인자의 thread로 cpu 점유를 전환한다. RETURN VALUE는 원래 실행되고 있었던 첫 인자의 thread pointer!!!

스레드는 기본적으로 만들어지면 (우리가 수정할 방향으로는 우선순위에 따라) ready\_list에 삽입되며, 때가 되면 실행되었다가 yield되면 다시 ready\_list의 끝에 삽입되는 식으로 계속 돌아간다. block과 unblock으로 스레드의 실행을 잠시 정지시킬 수 있으며, 일을 다 끝내면 마지막에는 exit로 삭제된다.

1. synchronization

동시에 서로 다른 스레드가 접근해서는 안 되는 코드 구역을 임계 구역이라고 하는데, 이러한 임계 구역의 처리를 하기 위한 처리가 동기화이다. 스레드가 임계 구역에 진입하면 lock을 걸고, 임계 구역을 나오면 unlock을 하는데, 이 과정에서 세마포어를 사용한다. 세마포어는 두 개의 원자적 함수로 조작되는 정수 변수로, sema\_down(P)은 임계 구역에 들어갈 때, sema\_up(V)은 임계 구역에서 나올 때 실행된다. 이 때 세마포어의 초기 value는 공유할 수 있는 자원의 총 수로, 세마포어의 value가 0이면 더 이상 사용할 수 있는 자원이 남아있지 않음을 의미한다. 이 경우 스레드는 해당 임계 구역에 접근할 수 없다. 다른 스레드가 사용하고 sema\_up을 해 줄 때까지 기다려야 한다.