

# **Pintos Project 3: Threads**

담당 교수: 박성용

학번 / 이름: 20171643 / 박준혁

개발 기간: 10/31~11/12

## I. 개발 목표

이번 ‘pintOS Project 3 – Threads’에선 **Alarm Clock, Priority Scheduling, BSD Scheduler**라는 총 3가지 구현 목표를 설정한다. 지금까지의 Project Phase들을 통해 우리는 pintOS에서 User Program이 정상적으로 구동할 수 있는 환경을 구축했다. 그 과정에서 여태 우리의 pintOS는 Simple Round-Robin Scheduler를 기반으로 각 Thread에게 CPU Control을 부여했다. ‘Simple Round-Robin Scheduler’란, 말 그대로 가장 Naive한 RR (Round-Robin) 논리를 따르는 Scheduler로, 각 Thread(Process를 포함)의 Priority를 고려치 않고 단순히 Ready Queue에 들어온 순서대로 Thread를 Pop하고 그들에게 CPU 점유권한을 주는, 그러한 간단한 알고리즘을 의미한다.

본 Project Phase에서 우리는 이를 좀 더 개선된 Scheduler로 수정한다. 기존 FIFO(First-In, First-Out) 기반 RR Scheduler에 Priority를 도입하는 것이다. 모든 Thread에게 각각의 Priority를 부여하고, 현재 CPU Control을 점유하고 있는 Running Thread의 Priority보다 더 큰 Priority를 가진 Thread가 Ready Queue 내에 존재하면, 해당 Running Thread가 CPU Control을 Yield하도록 만들고, 더 높은 우선순위의 Thread가 돌아가도록 Re-Schedule하는 것이다.

또한, Priority에 가변성을 주어야 한다. 만일, Priority가 처음 할당된 값 그대로 고정된다면, 낮은 우선순위의 Thread는 CPU Control에 대한 Starvation 문제를 겪게 될 것이다. 따라서, 우리는 Priority Aging Technique를 적용한다. Nice, Load Average, Recent CPU와 같은 몇 가지 관련 Variable을 도입해 Aging을 구현한다. Fair Distribution이 일어나도록 말이다.

이러한 Aging Technique이 구현되면, 이를 토대로 BSD Scheduler도 구축한다. **BSD Scheduler**란 **MLFQ(Multi-Level Feedback Queue) Algorithm** 기반의 Scheduler로, 각 Priority에 대해 개별적인 Queue를 두어, 높은 우선순위의 Queue부터 Thread를 찾아 Schedule하는 방식이다. 이러한 Scheduler 구축 과정은 아래에서 좀 더 자세히 설명한다.

한편, 본 Phase에선 Alarm Clock의 구현도 요구한다. 특정 Thread에 대한 Timer Sleep 상황 시, 정해진 시간이 지나면 pintOS가 해당 Slept Thread를 Wake Up 할 수 있도록 만 들어주는 것이 핵심이다. 사실, 이는 Default pintOS에 이미 구현되어 있긴 하지만, 상당히 비효율적이다. 우리는 이에 대한 개선 작업을 해주어야 한다.

이제부터 본 Project의 수행 내용을 단계적으로 소개하도록 하겠다.

## II. 개발 범위 및 내용

### A. 개발 범위

상기 설명대로 본 'Project 3 – Thread'가 진행되는데, 우리는 이를 1) Alarm Clock, 2) Priority Scheduling, 3) Advanced Scheduler라는 세 항목으로 구분할 수 있다. 각 항목의 개발 필요성과 기대 결과는 다음과 같다.

#### 1) Alarm Clock

pintOS를 비롯한 많은 운영체제는 특정 Thread를 잠시 재웠다가 일정 시간이 지나면 해당 Thread를 다시 깨우는 기능이 존재한다. 마치 사람이 Alarm을 설정해두고 설정 시간이 되면 Alarm이 울려 깨는 것처럼 말이다. 이때, **기존의 Default pintOS에선 이 과정을 Busy Waiting 방식으로 처리한다.** Sleep의 Target Running Thread가 CPU Control을 Yield해 Ready Queue로 돌아가고, 다시 자신의 차례가 되어 CPU Control을 얻으면 '깨어날 시간이 되었는지' 확인해 시간이 다 됐으면 깨어나고, 그렇지 않으면 다시 CPU Control을 Yield하는, 이러한 행위가 계속 반복되는 식으로 처리한다.

허나, 이러한 기존의 방식은 Slept Thread에게 지속적으로 '실제 CPU 점유 권한'을 부여해 Tick(Time Count of pintOS)을 소모하게 만든다. 이는 상당한 비효율로, Slept Thread가 이름과 다르게 실제로는 자지 않고 있는 것이다. 따라서, 우리는 이를 개선해야 한다.

개선은 어렵지 않다. Slept Thread들을 특정 자료구조에 저장해놓은 후, 매 Time Interrupt마다 '깨어날 Thread가 있는지' 확인하면 된다. 그리고 그 자료구조에 Slept Thread를 넣을 때, Thread의 State를 BLOCKED로 지정해줌으로써 해당 Slept Thread가 실제로 CPU Control을 가지지 못하도록 유도해야 한다. 그것이 진정한 의미의 Sleeping이기 때문이다. 이를 통해 Alarm Clock 상황을 좀 더 효율적으로 처리할 수 있을 것이다.

#### 2) Priority Scheduling

상기 '개발 목표' 항목에서도 언급하였듯, Default pintOS의 기본 탑재 Scheduler는 'Simple Round Robin Scheduler'로, Ready Queue에 대해 Naive하게, Priority를 고려치 않고 삽입 순서대로 CPU Control을 부여하는 방식이다. 이는 '수행이 시급한 Thread'를 우선적으로 Schedule하지 못하기 때문에, 우리는 Priority 개념을 도입해야 한다. **Currently Running Thread의 Priority보다 더 높은 Priority를 가진 Thread가 Ready Queue에 존재한다면, 해당 Running Thread가 CPU Control을 Yield하고, 다시 Re-Schedule할 수 있도록 만들어준다.**

이때, 이 과정이 단순히 매 Context Switch 시마다 수행되는 것뿐만 아니라, 새로운 Thread를 생성할 때, Semaphore에 대한 V Operation을 수행할 때 등 다양한 상황 속에서 이 Context를 유지할 수 있도록 만들어주는 것이 중요하다. 이를 통해 “항상 현재 Running Thread가 모든 Threads 중 가장 Priority가 높다”가 지켜질 수 있도록 한다.

### 3) Advanced Scheduler

하나, 2)번 항목까지의 개발만 수행되면 **Lower Priority의 Threads가 CPU Control을 얻지 못하는 Starvation 문제**가 발생한다. 결국, 이를 방지하기 위해선 ‘Priority에 대한 가변성’, 즉, Priority에 대한 Aging Technique의 도입이 필요하다. Nice, Load Average, Recent CPU와 같은 Thread 관련 변수들을 도입하고, 이들에 대해 특정 주기마다 변화를 준 후, 이를 이용해 Priority를 재-계산하는 것이다. 그리고, 재-계산된 Priority를 기반으로 Priority Scheduling을 수행하는 것이다.

$$\text{priority} = \text{PRI\_MAX} - (\text{recent\_cpu} / 4) - (\text{nice} \times 2)$$

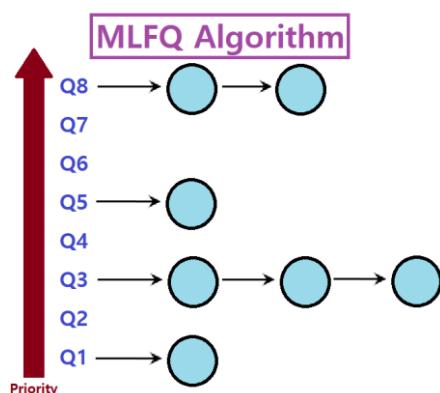
$$\text{recent\_cpu} = (2 \times \text{load\_avg}) / (2 \times \text{load\_avg} + 1) \times \text{recent\_cpu} + \text{nice}$$

$$\text{load\_avg} = (59/60) \times \text{load\_avg} + (1/60) \times \text{ready\_threads}$$

이때, 위와 같은 수식을 기반한 ‘재-계산’이 이뤄지는데, 수식을 보면 이것이 (전반적으로) I/O Bound Process에겐 Favor를, CPU Bound Process에겐 Non-favor를 주는 방식으로 진행될 것임을 짐작할 수 있다. Load Average Value는 Ready Queue 내 Threads의 개수를 기반으로 가중되고, 이 Value로 Recent CPU 값을 Weighting한다. 그리고, Priority는 Maximum Priority에서 이 Recent CPU 값을 빼서 재-계산되기 때문에, CPU를 덜 쓰는 Thread일수록 높은 우선순위를 가지는 경향이 있을 것이다.

결론적으로 이렇게 Aging을 도입해 Priority Scheduler를 완성한다. 각 변수에 대한 주기적인 재-계산이 핵심이며, 이 과정에서 본 pintOS가 Floating-Point Arithmetic을 지원하지 않기 때문에 Fixed-Point Arithmetic을 구현해야 한다.

이어서, 완성된 Aging 기법을 BSD Scheduler로 확장하는 작업까지 본 Step에 포함된다.



## B. 개발 내용

위에서 설명한 'Alarm Clock', 'Priority Scheduling', 'Advanced Scheduler' 항목 각각에서 핵심적인 개발 내용은 무엇이었는지, 구체적으로 어떤 개념을 토대로 그러한 개발이 이뤄지는지를 설명하겠다. 아래를 보자.

### 1) Blocked 상태의 Thread를 어떻게 깨울 수 있는지

'Alarm Clock'의 구현 시 우리는 Sleeping의 대상이 되는 Thread가 BLOCKED State로 변화하고, 그러한 Thread들을 특정 자료구조에 삽입해서 관리해야 한다고 언급했다. 이때, 이러한 BLOCKED State Thread를 어떻게 깨울 수 있을까. 본인은 이 '자료구조'에 집중했다. 그 내용은 다음과 같다.

- 첫째, Slept(Blocked) Thread를 '**Sleep List**'라는 이름의 **Thread Queue**에 삽입 한다.
- 둘째, Thread 삽입 시, '해당 Thread가 깨어날 시간'을 Tick을 기준으로 상정 해 이를 해당 Thread의 PCB에 기록해둔다.
- 그리고, 그 Thread를 BLOCKED State로 변화시킨다.
- 셋째, 매 **Time Interrupt** 발생 시마다 이 **Sleep List Queue**를 Search해, **Wake Up Time**이 지난 Thread들을 골라내어 이들을 다시 Ready State로 바꾸고 Ready Queue에 삽입한다.

즉, 새로운 Queue의 도입, 그리고 매 'Time Interrupt (Tick Counting)'마다 주기적인 Sleep Queue Search 및 Wake Up 작업을 통해 이 과정을 완성할 수 있는 것이다. 참고로, 이때 이 Search는 Iterative Search, Hash 기반 Search 등의 다양한 방식으로 수행할 수 있지만, 본인의 경우, 구현의 간편성 및 현실적인 Blocked Thread 개수가 그리 크지 않음 등을 고려해 단순 For문을 이용한 Search를 선택했다.

한편, Alarm Clock 이외의 방식으로 이뤄지는 Block & Unblock Routine은 이미 pintOS 상에서 Cover하고 있기에 추가 언급은 필요 없다고 판단한다.

### 2) Ready List에 Running Thread보다 높은 Priority를 가진 Thread가 들어올 경우 Priority Scheduling에 따르면 어떻게 해야 하는지

우선, 기본적으로 pintOS에서 Ready Queue를 관리하는 방식을 변경해주어야 한다. pintOS에선 List 자료구조에 대한 Sorted Insertion을 제공한다. 'list\_insert\_ordered'라는 이름의 함수로, 이 함수를 사용하면 List에 '특정 Order(높은 Priority의 Thread가 List Front

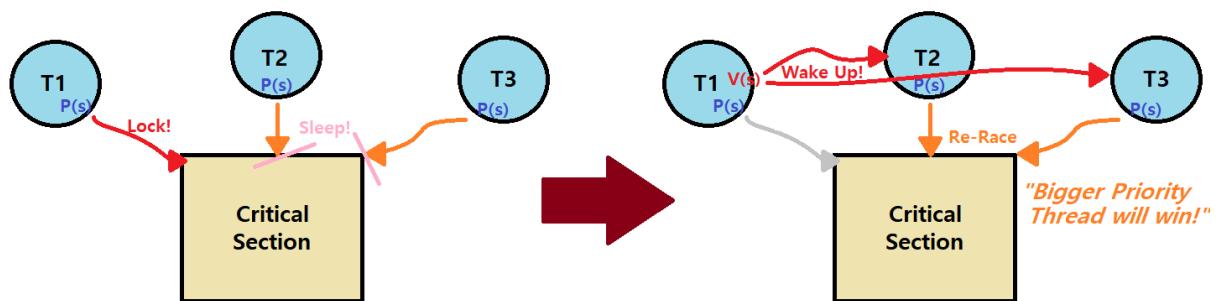
쪽으로 가도록 정렬되는 순서)'를 기준으로 List Insertion이 가능해진다. 이 함수를 언급하는 이유는, Currently Running Thread가 CPU Control을 Yield할 때, 기존에 Ready Queue에 Thread를 Push Back하던 것과 다르게, Sorted Insertion을 수행하는 것이다. 현재 수행 중이던 Thread가 CPU Control을 놓으면서 동시에 Ready Queue 내의 자신의 적절한 위치로 이동하는 것이다.

이와 함께, Blocked Thread를 다시 Ready State로 변경할 때에도 위와 같은 Sorted Insertion 작업을 해주어야 한다. 예를 들어, 앞선 항목의 Alarm Clock에서, 제 시간이 다되어 다시 깨어나야 하는 Slept Thread는 Unblock 과정을 거치게 되는데, 그 과정에서 Ready Queue에 삽입될 때 자신의 '적절한 위치(Priority를 고려)'로 이동하는 것이다.

즉, 'Ready State가 아닌 Thread가 Ready State로 변경되어 Ready Queue로 들어갈 때'를 주목해야 하는 것이다. 항상 **Ready Queue 삽입 시 Sorted Insertion을 수행해야 한다.** 이러한 상황에는 위 두 상황을 비롯해, Thread Creation 상황, 특정 Thread에 New Priority를 설정하는 상황, Semaphore를 두고 V Operation이 수행되어 Waiting Thread 중 하나가 깨어나는 상황 등이 포함된다.

Semaphore 예시를 조금 더 들여다보자. Semaphore에 대해 P Operation을 통해 Waiter가 된 Threads가 여러 개 있다고 하자. 특정 순간에 해당 Semaphore에 대해 V 연산이 수행되고, 이에 따라 복수의 Waiter들이 다시 Re-Race해야하는 상황이 되었다고 하자. 이 때, Re-Race의 Winner는 누가 되어야 하는가.

"그렇다. Re-Race 시 Waiting Threads 중 가장 Priority가 높은 Thread가 Winner가 되어 Wake Up 해야한다."



즉, OS 관점에선 Waiting Threads에 대해 특정 자료구조를 운용하고, V Operation 시 해당 자료구조를 순회해 Highest Priority Thread를 찾아 해당 Thread만 깨워야 한다. 해당 Thread에 대해 Un-Block Routine을 수행해, Ready Queue에 Sorted Insertion을 수행하는 것이다.

이렇게까지 구현하면 기본적인 Priority Scheduler가 완성된다. 왜냐면, Ready Queue의 Front Head에서 Pop을 수행해 특정 Thread에게 CPU Control을 부여하는 기존의 Simple

RR Scheduler 논리 일부는 여태까지 설명한 논리와 함께 조응하여 Priority Scheduler를 만들 수 있기 때문이다.

“Ready Queue에 Thread를 삽입할 때 (Descending Order in Priority) Sorted Insertion을 수행하면, 항상 Ready Queue의 Front에는 Ready State Threads 중 Highest Priority Thread가 위치하게 되고, 여기서 Pop하는 논리는 Priority Scheduler의 논리와 부합한다.”

이렇게 하면 Priority Scheduling을 완성할 수 있다.

### 3) Advanced Scheduler에서 Priority 계산에 필요한 각 요소

하나, 계속 언급한 것처럼, 위와 같은 Priority Scheduler는 Lower Priority Thread가 굶는 Starvation 문제를 일으킬 수 있다. 따라서, Fair Distribution을 위해선 Priority가 계속 변할 수 있어야 한다. CPU Burst Time이 큰 Thread는 Priority가 낮아지고, CPU Burst Time이 작은 Thread는 Priority를 높여 효율적이고 우수한 성능의 Scheduling을 도모해야 한다.

이를 위해 우리는 Aging Technique을 도입한다고 했다. 앞서 소개한 바와 같이 이러한 Aging에는 Nice, Load Average, Recent CPU라는 Variable들이 관여한다.

- Nice: Thread의 Niceness를 지정하는 Value로, 쉽게 말하면 ‘친절한가?’이다. 친절하다는 것은, 다른 Thread에게 CPU Control 점유 기회를 많이 주는지를 의미한다. 그래서, Nice 값이 크면 클수록 Priority 재-계산 시 Priority를 많이 낮추려고 유도한다. Thread가 하는 기능과 역할에 따라 Nice 값을 적절히 조절할 수 있겠다.
- Load Average: OS 내에서 System-Wide한 Value로, 말 그대로 System에 Thread가 얼마나 많이 Load되었는지를 나타낸다. 기존의 Load Average 값에 대한 History를 고려하며, 현재 Ready Queue 내에 Thread가 얼마나 있는지에 따라 그 값이 커질 수도, 작아질 수도 있다.
- Recent CPU: Thread가 이전 CPU Control 점유 상황에서 CPU를 얼마나 많이 사용했는가를 나타낸다. 역시나 History를 고려하며, 상기한 Nice와 Load Average Value를 적절히 혼합해 계산한다.

Unix와 같은 유명 운영체제에선 Thread의 Priority를 위와 같은 요소들을 이용해 계산하여 가변성을 부여한다. 우리의 pintOS 역시 마찬가지로 그럴 것이며, Priority는 Time Slice마다, Recent CPU와 Load Average는 1초마다 재-계산될 것이다. Nice의 경우 Thread Creation 시 지정되며, 중간에 변경할 수 있다. 이 과정에 아래와 같은 수식들이 이용됨을 앞서 확인한 바 있다.

```

'priority = PRI_MAX - (recent_cpu / 4) - (nice * 2)'

'recent_cpu = (2 * load_avg) / (2 * load_avg + 1) * recent_cpu + nice'

'load_avg = (59/60) * load_avg + (1/60) * ready_threads'

```

이를 실제로 구현하기 위해선 pintOS에서 지원하지 않는 실수 계산 기능을 만드는 것이 중요하다. 본 Phase에선 Fixed-Point Arithmetic을 이용한다. 본인은 이를 위해 우선적으로 fixed\_point.h라는 Header 파일을 만들어, 마치 함수를 사용하듯이 실수 계산을 할 수 있도록 했다. 위 수식에 사용되는 사칙 연산에 대해 Fixed Point Arithmetic을 구현하였으며, 자세한 Detail은 후술하겠다.

Fixed-Point Arithmetic이 준비되면, Aging의 구현은 어렵지 않다. OS에서 시간을 재는 기준은 무엇이라 하였는가? 그렇다. Timer Interrupt 마다 계수를 하면 쉽게 시간을 파악할 수 있다. 즉, 우리는 pintOS의 Timer Interrupt Handler에서 위에 명시한 주기마다 각각의 Variable을 재-계산하면 된다.

### III. 추진 일정 및 개발 방법

#### A. 추진 일정

앞선 Project Phase 때와 다르게 이번 Phase에선 '요구사항 분석'에 시간이 많이 들지 않았다. 아무래도 두 번 Phase를 통해 User Program을 구축하는 과정에서 pintOS에 대한 이해가 깊어졌기 때문에 요구사항을 이해하는 것이 어렵지 않았다. 사실상 이번 Phase에선 오히려 Code-Level Implementation에서 시간이 많이 소요되었으며, 특히나 Aging 구현 과정에서 수식이 정확한 동작을 하는지 파악하기가 까다로웠다. 이 과정에서 겪었던 Troubleshooting 과정은 아래 항목에서 소개하겠다. 결과적으로 개발 일정을 요약하면 다음과 같다.

##### 1) Understanding the Requirements (10/31-11/02)

Project 안내 PDF와 pintOS Manual을 토대로 본 Phase에서 요구하는 Alarm Clock, Priority Scheduling, BSD Scheduler의 이론적 개념을 이해한다. 중간 시험 준비 과정에서 Scheduling Algorithm에 대한 학습을 마친 상태라 이해에 별다른 어려움이 존재하지 않았다. 약 3일 간 가볍게 관련 자료를 읽어보며 Implementation 준비를 마쳤다.

## 2) Alarm Clock (11/02-11/04)

Alarm Clock Step을 개발하는 과정이다. 상기 서술 내용대로 어렵지 않게 구축하였으나, Sleep List의 정의 및 초기화 위치와 관련해서는 약간의 난해한 부분이 있긴 했다. pintOS에선 Timer 관련 함수가 devices/timer.c 파일에, Thread 관련 함수가 threads/thread.c 파일에 각각 따로 존재하는데, Timer Sleep으로 인해 BLOCKED된 Thread들이 Sleep List에 들어간 후, 이 공통의 자료구조를 양 파일에서 동시에 접근할 수 있어야 했다. 이것을 맞춰주는 작업이 생소했다. 결과적으로, **extern** Keyword를 이용해 해결할 수 있었다.

## 3) Priority Scheduling (11/04-11/06)

Priority Scheduler 설계 역시 순조롭게 진행하였다. 상기 내용을 토대로, Ready Queue의 삽입과 관련한 함수 및 Context를 적절히 찾아, 해당 함수의 List Insertion을 변경하는 식의 작업을 통해 개발할 수 있었다. 자세한 Code-Level Detail은 후술한다.

## 4) Advanced Scheduler (11/06-11/09)

가장 까다로운 지점이었다. 상술한 것처럼, Aging의 핵심 수식들을 Fixed-Point로 다루어 계산하는 과정이 상당히 복잡했다. 이게 직관적인 확인 어려웠는데, 그 이유는, 본 Phase의 make check Test 과정이 상당히 길었기 때문이다. 수식을 조금만 변형하더라도 오랜 시간이 소요되는 Test 과정이 필요했기 때문에 절대적인 시간 자체가 많이 소요되었다. 다행히, 수식의 안정성이 파악된 후부터는, 재-계산 함수의 배치 과정은 그다지 복잡하지 않았다.

모든 Test Case의 통과를 확인한 시점은 11/09일었으며, 이후 사흘 정도 (11/09-11/12) 보고서 작성 및 코드 정리를 수행하였다.

## B. 개발 방법

지금까지 설명한 개발 내용을 실제 Code로 구현하기 위해선 어떠한 작업이 필요한지를 항목 별로 소개하겠다.

### i) Alarm Clock

Alarm Clock Step은 devices/timer.c 파일에서 구현할 수 있다. 초기엔, Blocked State Thread 전체에 대해 Blocked Queue와 같은 자료구조를 운용해야 하는 것 아닌가 고민했다. threads/thread.c에서 Blocking 관련 Routine에서 모조리 이 자료구조에 접근해 Context를 유지 관리해야 하는 것이라 판단했다. 하지만, 잠시 기존 thread.c 파일 코드를 분석한 결과, 그렇지 않아도 됨을 알 수 있었다. 기본적인 Scheduling 과정에서

Blocked Thread를 배제하는 방법은 너무나도 간단했다. 그저, State Flag를 BLOCKED로 변경하고, Schedule 관련 Routine에서 이를 체크해 제외하면 되는 것이었다.

따라서, Alarm Clock Step에서 Blocked Thread를 관리하는 것만 신경 쓰면 된다. 우선, Timer Sleep을 통해 생성되는 **Slept Thread**들을 OS가 추적/관리 할 수 있어야 한다. 이를 위해 ‘Sleep List’라는 이름의 새로운 Queue를 도입한다. 매 Timer Sleep이 이뤄질 때마다 Target Thread를 State 변경 후 이 Queue에 삽입하는 것이다. (timer\_sleep 함수 수정)

이어, Queue에 들어간 Thread는 ‘제 시간(깨어나야 하는 시간)’이 되면 Wake Up 해야 한다. 이를 위해 우리는 Thread Structure에 한 가지 추가 Field를 마련해야 한다. 일어날 시간을 기록하는 것이다. 본인은 ‘**wakeup\_time**’이라고 명명하였다. (이는 threads/thread.h 파일의 Thread Structure 변경을 통해 수행)

그렇다면, 깨우는 과정은 어떻게 구현할 것인가? 이미 언급한 것과 같이, 매 Timer Interrupt마다 확인하면 된다. Sleep List Queue를 매 Timer Interrupt마다 Search하여, List 내 Thread 중 현재 ‘Time Count (Tick)’ 이전의 ‘wakeup\_time’을 가진, 즉, 깨어날 시간이 된 Thread가 발견되면, Block 시켰던 것을 Unblock 시키는 것이다. (이 Block & Unblock Routine은 thread.c의 함수를 그대로 옮겨 사용)

참고로, timer.c의 timer\_interrupt 함수는 변경 없이 그대로 유지한다. 사실, 이는 Design Dependent한 요소인데, 본인은 thread\_tick 함수의 존재 이유를 살리기 위해 ‘**Wake Up Routine**’을 timer.c가 아닌 thread.c 파일의 thread\_tick 함수에서 수행하였다. thread\_tick 함수가 thread\_wake\_up이라는 새로운 함수를 호출해 이 작업을 위탁한다. 자세한 Code-Level Detail은 후술하겠다.

## ii) Priority Scheduling

Priority Scheduler의 설계는 이미 여러 차례 언급한 것처럼, Ready Queue Insertion 과정을 수정하면 된다. 먼저, **Ready Queue Insertion**이 기존에 어디서 이뤄지는지 파악해야 한다. threads/thread.c 파일을 보자.

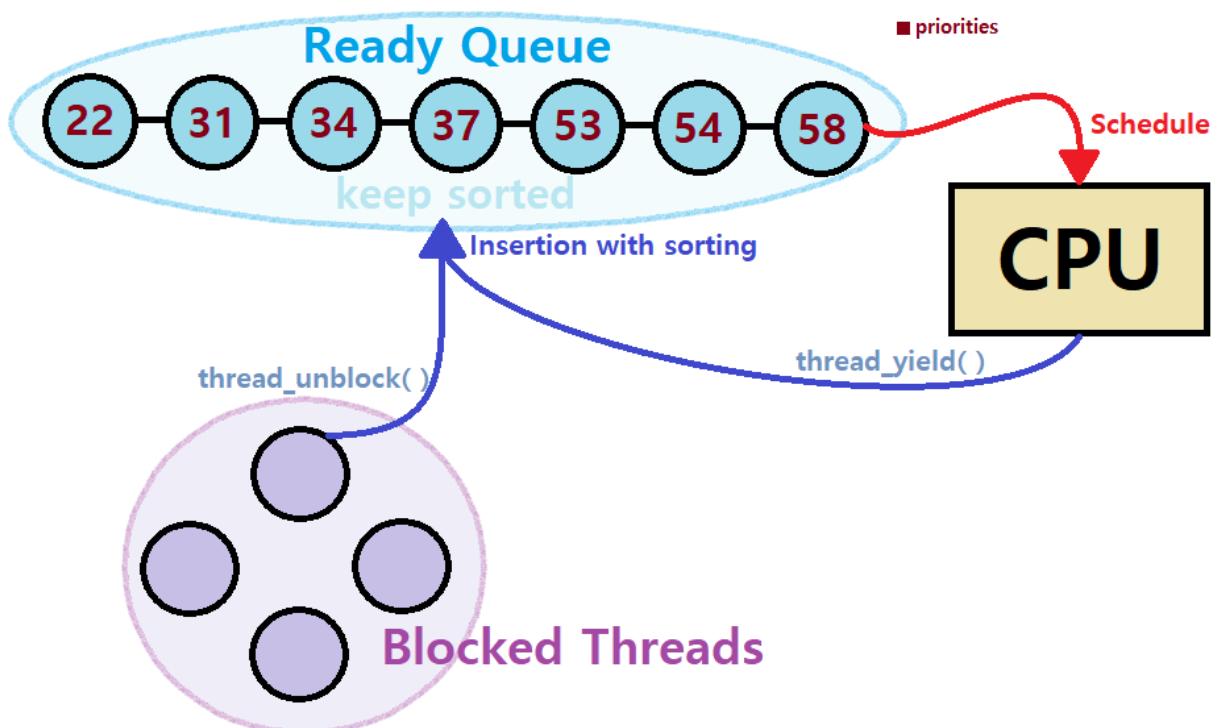
- **thread\_unblock:** BLOCKED State Thread를 Ready State로 변경하고 Ready Queue에 다시 삽입하는 과정이 존재한다.
- **thread\_yield:** Currently Running Thread가 CPU Control을 놓으면 Running State에서 Ready State로 바뀌고 Queue에 삽입되어야 한다.

이어, 새롭게 Ready Queue 삽입 코드를 추가해야 하는 부분이 어디인지 파악하자. 새롭게 **Priority**가 Set될 때, 해당 **Priority**가 그 순간에 CPU를 사용하는 **Running**

Thread의 Priority보다 높을 경우 Re-Schedule되어야 한다.

- **thread\_create**: 새로운 Thread를 생성할 때, 생성된 Thread의 Priority가 현재 수행 Thread의 Priority보다 높을 수 있다.
- **sema\_up**: Counting Semaphore에 대한 V Operation 수행 시 Waiter 중 가장 Priority가 높은 Thread를 Ready Queue로 복권(復權)시켜야 한다. 이 과정에서, 만약 이 Thread가 현재 수행 Thread보다 우선순위가 높다면, Re-Schedule해야 한다. (threads/synch.c 파일에서 작업)
- **thread\_set\_priority**: 당연히 특정 Thread의 Priority를 변경한 경우, 위와 같은 작업을 수행해야 한다.
- **thread\_set\_nice**: Nice 값의 변경은 Priority 재-계산 수식의 변경을 의미한다. 따라서, 여기서도 새 Nice 값에 대한 Priority를 토대로 Re-Schedule이 필요한지를 확인해야 한다.

위와 같은 위치에 각각 Sorted Insertion을 수행한다. 앞의 두 함수에선 'list\_insert\_ordered' 함수를 호출해 이를 구현하고, 뒤의 네 함수에선 `thread_yield` 함수를 호출해 구현한다. 참고로, 'list\_insert\_ordered' 함수는 삽입뿐만 아니라, List 정렬도 수행한다. Parameter로 넘긴 Comparison 기준 함수를 토대로 말이다. (그 말은 즉, Comparison 기준 함수도 따로 마련해야 한다는 것임. 본인은 `priority_compare`라는 이름으로 작성)



### iii) Advanced Scheduler

우선 Aging Technique을 구현하자. 앞서 언급한 바와 같이, Recalculation 수식의 구현을 위해선 **Fixed-Point Arithmetic**을 먼저 구현해야 한다. 이를 위해 fixed\_point.h라는 헤더 파일을 threads Directory에 마련한다. 해당 헤더 내에는 실수 사칙 연산을 수행하는 매크로 함수들을 정의한다.

예를 들어, 실수 변수 real과 정수 변수 integer를 더한다고 하자. 실수 덧셈 상황이다. real과 integer 모두 32Bits Type이다. integer에 대해 Left Shift 연산으로 14Bits 이동하면 Fixed-Point Real Type에 맞는 수가 도출될 것이다. 이 값에 기존 real 값을 더하면 될 것이다. 뺄셈도 유사한 논리로 구현하자.

곱셈과 나눗셈은 더 간단하다. Real Type과 Integer Type의 곱셈은 그냥 둘을 곱하면 된다. 왜냐? **Binary Multiplication**은 Bit 단위로 Partial Sum 논리에 따라 수행되는데, 이를 고려하면 Real Type을 Integer만큼 더하면 되기 때문이다. 같은 이유로, Real Type 두 개를 곱할 땐 단순히 곱하고, 나눌 땐 단순히 나누면 된다. 다만, 자료형의 초과를 고려해 64Bits로 잠시 Casting해두는 작업이 필요할 것이다. 값의 손실을 방지하기 위해 말이다.

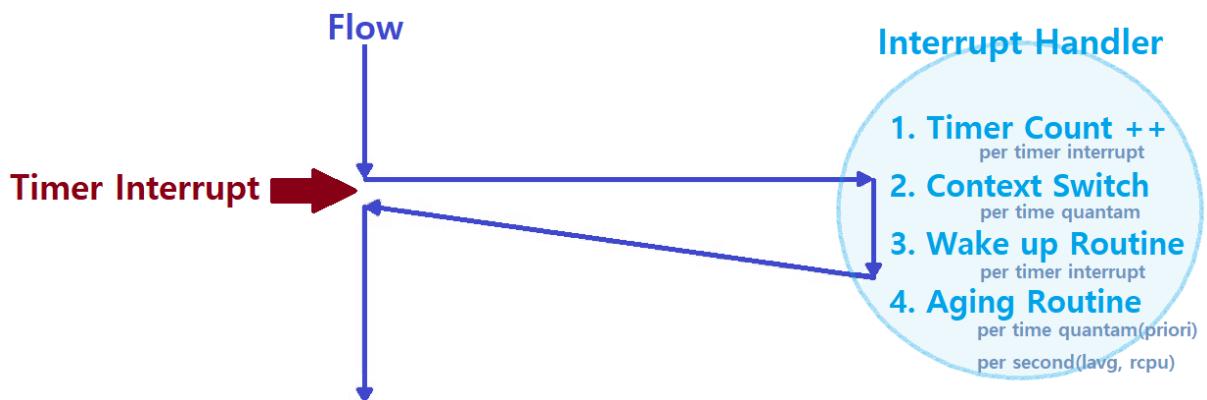
이렇게 Fixed-Point Arithmetic을 간단히 구현하였으면, 이어 이를 토대로 Aging 관련 Variable의 Recalculation Function을 마련하자. 우선, Variable의 도입이 필요한데, Recent CPU와 Nice Value는 Thread마다 하나씩 존재하는 값이므로 threads/thread.h의 Thread Structure에 새로운 변수를 도입해 마련한다. Load Average의 경우, OS-Wide Value이므로, threads/thread.c에 Global Variable로 마련하면 될 것이다. (초기화도 적절히 할 것)

이를 토대로 재계산 함수를 작성한다. recalculate\_variable의 형식으로 세 가지 함수를 만든다. 세 함수는 모두 위에서 소개한 수식을 Fixed-Point Arithmetic으로 계산만 하면 되는 간단한 함수이다. 다만, Recent CPU와 Priority의 경우 Thread마다 하나씩 처리해야 하므로 pintOS에서 모든 Thread를 관리하는 Queue인 'all\_list'에 대한 순회가 필요할 것이다. 참고로, recalculate\_priority의 경우, 새롭게 계산된 Priority들 중 Thread 가 Ready Queue에 있으면서 그 값이 가장 큰 Thread에 대해, Currently Running Thread 보다 우선순위가 높으면 Re-Schedule하도록 Yield Routine을 추가해야 한다.

참고로, `thread_yield` 함수 대신 `interrupt.c` 파일의 `intr_yield_on_return` 함수를 사용해야 한다. Recalculation은 Timer Interrupt Handling 중간에 수행되기 때문이다. timer.c의 `timer_interrupt` 함수가 Timer Interrupt Handler로서, 매 Timer Interrupt마다 수행되고, 해당 함수에선 Time Count인 Tick에 대한 Increment를 수행한 후 `thread_tick` 함수를 호출한다. 그리고, `thread_tick` 함수에선 Aging을 관리하는 `thread_aging`(새롭게 정의)라는 함수를 호출하고, 그 함수는 내부에서 이 `recalculate_variable` 함수들을 호출하지 않는가. 즉, `recalculate_priority` 함수도 결국 Interrupt Handling 과정 중간에 수

행되는 작업이기 때문에, 바로 Yield하지 않고, Handling이 끝나면 그제서야 Yield하도록 유도하는 것이다.

이렇게, Recalculation 함수가 완성되면, 앞 문단에서 미리 언급한 것처럼, thread\_tick 함수(앞 전에 thread\_wake\_up 함수를 호출하던 함수)에 thread\_aging 함수도 호출해 Aging을 수행하도록 만들어준다. thread\_aging에선 세 Recalculation 함수들을 호출해 Aging을 실질적으로 수행한다. 이후, 몇 가지 미완의 함수(thread\_set\_nice, thread\_get\_priority와 같은 함수들)를 구현하고 나면 Aging Technique 구현이 마무리된다.



Aging Technique까지 끝나면, BSD Scheduler의 구현은 일도 아니다. 처음 '개발 범위' 항목에서 소개한 바와 같이, BSD Scheduler는 MLFQ Algorithm이다. 자, 잠시 생각해보자. 다음과 같은 사실이 보이는가?

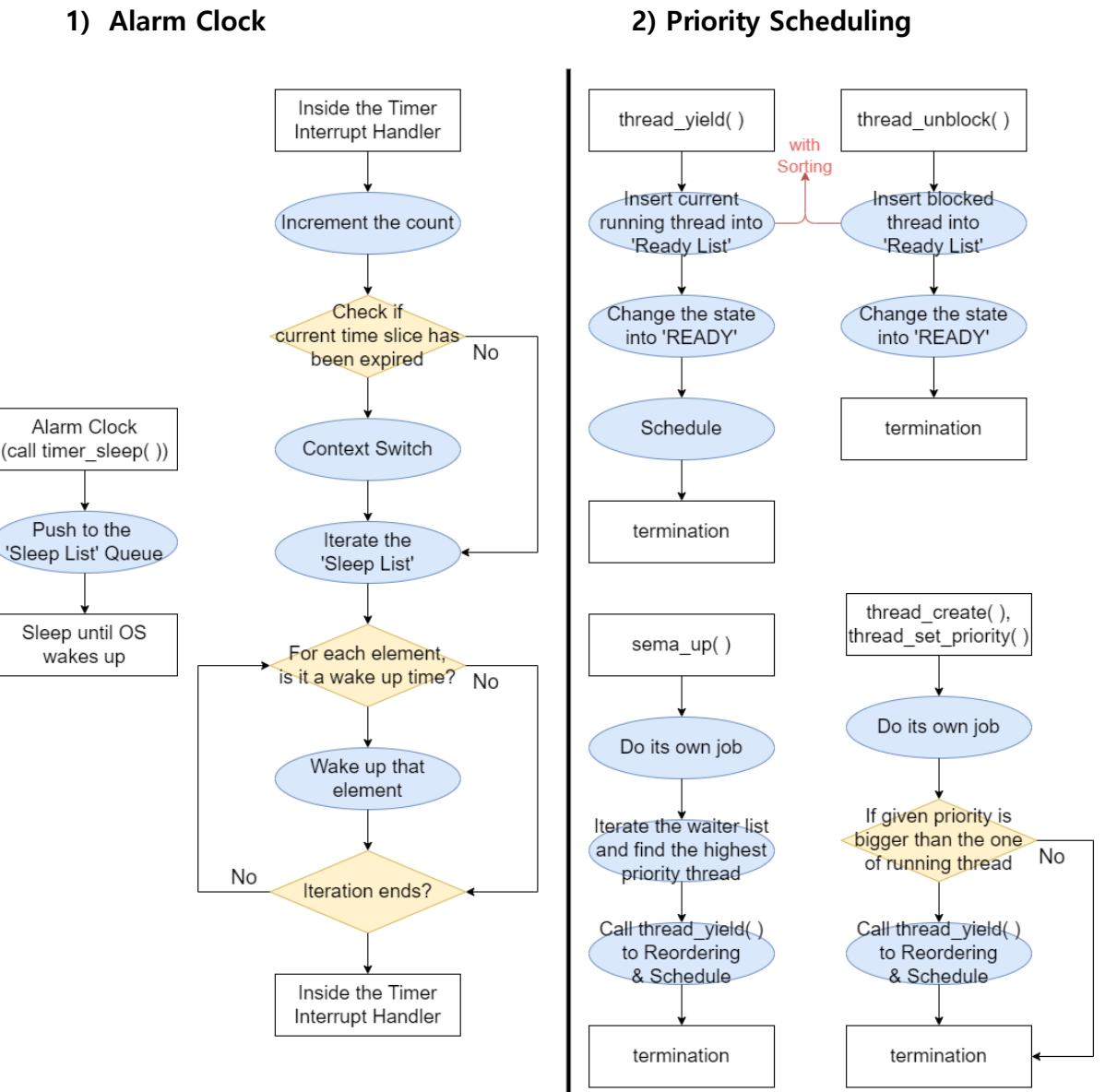
**"Aging Technique이 적용된 Priority Scheduler 자체가 곧 BSD Scheduler이다."**

왜냐? Manual에 따르면, BSD Scheduler는 하나의 Queue로 구현해도 문제가 없다. 그리고, Aging Technique이 적용된 Priority Scheduler는 하나의 Ready Queue를 가지고 MLFQ 논리를 수행한다. 즉, 이 자체가 바로 BSD Scheduler인 것이다. 다만, Manual에 따르면 BSD Scheduler에선 thread\_set\_priority 함수가 Scheduler가 정해놓은 Priority를 그대로 반환해야 한다고 하므로 여기서 MLFQ일 경우 처리하지 않도록 지정해주는 작업만 하면 될 것이다.

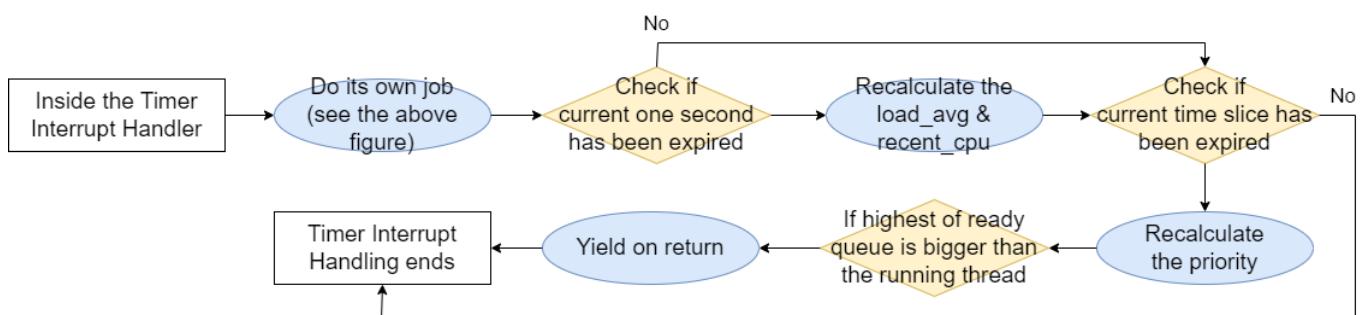
물론, Queue 60개를 운영해 BSD Scheduler를 구현할 수 있으나, 이 작업은 단순히 자료구조 확장 작업만 하면 되므로 굳이 일부러 수행할 필요는 없다 판단하여 생략하였다. 이렇게 해서 Advanced Scheduler 추가 구현을 마무리한다.

## IV. 연구 결과

### A. Flow Chart



### 3) Aging Technique



## B. 제작 내용

### 1) Alarm Clock

일단 여태 설명한 내용대로 Timer Sleep으로 인한 Slept Thread들을 기록하는 'Sleep List'의 구현이 선행되어야 한다. 본인은 devices/timer.c 파일에 Global Data Structure로서 sleep\_list라는 List를 정의하였다. 아래와 같이 말이다.

```
/* List of processes that are slept by 'timer_sleep()' function
   in the timer.c. It means, all the threads in here are BLOCKED */
struct list sleep_list;
```

이러한 Sleep List는 아래와 같이 Timer Device가 시작함에 따라 바로 초기화된다.

```
/* Sets up the timer to interrupt TIMER_FREQ times per second,
   and registers the corresponding interrupt. */
void
timer_init (void)
{
    pit_configure_channel (0, 2, TIMER_FREQ);
    intr_register_ext (0x20, timer_interrupt, "8254 Timer");
    list_init(&sleep_list);
}
```

그리고 이어서 Timer Sleep 명령 시에는 아래와 같이 동작하도록 설계한다. 상기 항목들에서 설명한 내용 그대로인데, 현재(호출시점) tick(Timer Count)을 start라는 변수에 저장한 후, 넘겨 받은 '인자(잘 시간)'을 start에 더한 '(tick을 이용한) 상대적 시간'을 Thread의 Field 변수 wakeup\_time에 저장한다(이를 위해서 threads/thread.h 파일의 Thread Structure에 해당 Field 추가). 이어, 해당 Target Thread를 Sleep List에 넣고 Block 시켜줌으로써 진정한 Sleeping을 실현한다.

```
/* Sleeps for approximately TICKS timer ticks. Interrupts must
   be turned on. */
void
timer_sleep (int64_t ticks)
{
    int64_t start = timer_ticks ();
    struct thread *cur_thread = thread_current ();
    enum intr_level old_level;

    ASSERT (intr_get_level () == INTR_ON);

    old_level = intr_disable ();

    /* Set the wake up time and push to the 'sleep_list'.
       And then, block it. */
    cur_thread->wakeup_time = (start + ticks);
    list_push_back (&sleep_list, &(cur_thread->elem));
    thread_block ();

    intr_set_level (old_level);
}
```

이렇게 Sleep List Pushing이 끝나면, 매 Timer Interrupt Handling 시에 '아까 재운 녀석을 포함해, 지금 깨어날 녀석이 있는지'를 확인하면 된다. 아래의 Timer Interrupt Handler 프로시저를 보자. threads/thread.c의 thread\_tick 함수를 호출해 Timer Interrupt 시 '해야 할 일'들을 위탁하고 있다.

```
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
    /* Increment the ticks. */
    ticks++;

    /* Crucial jobs such as waking up the expired slept threads
       or performing the aging techniques are done in 'thread_tick'
       function in the thread.c file, for every time interrupt. */
    thread_tick ();
}
```

아래의 좌측 코드는 thread\_tick 함수이다. Context Switch를 비롯해 Timer Interrupt 시 일어날 일들을 수행하고 있음을 어렵지 않게 확인할 수 있다. 본 Project3에선, 이 부분에 아래 코드와 같이 Wake Up Routine과 Aging Routine을 추가한다. Aging은 잠시 제쳐두고, 먼저 Wake Up Routine에 주목해보자.

```
/* Called by the timer interrupt handler at each timer tick.
   Thus, this function runs in an external interrupt context. */
void
thread_tick (void)
{
    struct thread *t = thread_current ();

    /* Update statistics. */
    if (t == idle_thread)
        idle_ticks++;
#ifdef USERPROG
    else if (t->pagedir != NULL)
        user_ticks++;
#endif
    else
        kernel_ticks++;

    /* Enforce preemption. */
    if (++thread_ticks >= TIME_SLICE)
        intr_yield_on_return ();

    /* Wake up all the expired slept threads. */
    thread_wake_up ();

    /* Performs the aging technique of priority scheduler. */
    if (thread_prior_aging == true || thread_mlfqs == true)
        thread_agging ();
}

/* Wake up all the threads whose sleeping times are already
   expired. That is, it iterates the 'sleep_list'.
   Meanwhile, this function is called for every time interrupt */
void
thread_wake_up (void)
{
    struct list_elem *iter; struct thread *entry;

    /* Iterate the 'sleep_list'. */
    for (iter = list_begin (&sleep_list);
         iter != list_end (&sleep_list); )
    {
        entry = EACH_ENTRY(iter);

        /* Check if the sleeping time is expired */
        if (timer_ticks() >= (entry->wakeup_time))
        {
            iter = list_remove (iter);           // pop that entry
            thread_unblock (entry);           // and unblock it
        }
        else iter = list_next (iter);
    }
}
```

Wake Up Routine을 수행하는 thread\_wake\_up 함수는 위 우측 코드와 같다. Sleep List를 단순 Iteration을 통해 Search하면서, 각 Element에 대해 wakeup\_time이 지났는지를 확인한다. 시간이 지났으면 해당 Thread를 Sleep List에서 Pop하고, Unblock하여 Ready Queue에 삽입하고 있음을 주목하자. 참고로, 위 우측 코드에서 EACH\_ENTRY는 매크로 함수로서, List의 Entry를 뽑아내는 기존의 매크로를 조금 더 가독성 있게 변형한 것이다.

이렇게 해서 Alarm Clock의 구현을 마칠 수 있다. 본 구현 과정에서 가장 까다로웠던 지점은, sleep\_list를 서로 다른 위치에 있는 timer.c와 thread.c가 공유해야 하는 것이었는데, 이는 C에서 제공하는 extern Keyword를 통해 해결할 수 있었다. sleep\_list의 선언 및 초기화, 삽입

은 timer.c에서 수행하고, thread.c에선 extern을 통해 이 sleep\_list를 참조해, 순회 및 삭제만 수행하는 것이다.

## 2) Priority Scheduling

Priority Scheduler는 상기 설명대로 'Insertion with Sorting'이 핵심이다. 따라서, 우선 특정 Thread(Running Thread or Blocked Thread)를 Ready Queue에 삽입하는 과정이 존재하는 Thread 관련 함수들을 찾아가, Ready Queue 삽입 과정을 Sorted Insertion으로 수정한다. 아래와 같이 말이다. (언급했듯, Schedule 과정에서 Next Running Thread를 항상 Queue의 Front에서 Pop해 사용한다면, 이러한 Sorted Insertion만으로도 충분히 Priority Scheduling이 가능하다. 다만, 이를 위해 Comparison Function이 필요할 것)

```
/* Yields the CPU. The current thread is not put to sleep and
   may be scheduled again immediately at the scheduler's whim. */
void
thread_yield (void)
{
    struct thread *cur = thread_current ();
    enum intr_level old_level;

    ASSERT (!intr_context ());

    old_level = intr_disable ();

    /* Insert the new node to the proper position in the list. */
    if (cur != idle_thread)
        list_insert_ordered(&ready_list, &cur->elem, priority_compare, NULL);

    /* Change the state of running thread. */
    cur->status = THREAD_READY;

    /* Re-schedule, expecting new thread get the control. */
    schedule ();

    intr_set_level (old_level);
}

/* Transitions a blocked thread T to the ready-to-run state.
   This is an error if T is not blocked. (Use thread_yield() to
   make the running thread ready.)

   This function does not preempt the running thread. This can
   be important: if the caller had disabled interrupts itself,
   it may expect that it can atomically unblock a thread and
   update other data. */
void
thread_unblock (struct thread *t)
{
    enum intr_level old_level;

    ASSERT (is_thread (t));

    old_level = intr_disable ();
    ASSERT (t->status == THREAD_BLOCKED);

    /* Insert new ready node to the proper position in the list. */
    list_insert_ordered(&ready_list, &t->elem, priority_compare, NULL);

    /* Change the state of blocked thread. */
    t->status = THREAD_READY;

    intr_set_level (old_level);
}
```

이 과정에서 사용되는 priority\_compare 함수는 List Sorting 시 Comparison의 기준이 되는 함수로, 아래와 같이 간단히 구성하면 된다. Priority를 비교해 Priority가 크면 좌측으로 가도록!

```
/* This function indicates the condition used for sorting
   in the linked list. It means, we will use this function
   with 'list_insert_ordered' routine. */
bool
priority_compare (const struct list_elem* left, const struct list_elem* right)
{
    struct thread *left_thread, *right_thread;

    left_thread = list_entry(left, struct thread, elem);
    right_thread = list_entry(right, struct thread, elem);

    return ((left_thread->priority) > (right_thread->priority));
}
```

이렇게 해서 Priority에 대해 Descending Order로 정렬된 Ready Queue가 만들어진다. 우리는 이제 위 두 함수를 이용해 Priority Scheduling의 논리를 안정화시키는 노력을 해야 한다.

앞선 항목들에서 여러 차례 언급했듯, 이 과정에는 다음과 같은 작업들이 포함된다.

- (1) Thread Creation 시 New Thread의 Priority가 Running Thread의 그것보다 더 큰 경우
- (2) Semaphore에 대한 V Operation 시 Highest Priority Waiter가 Ready Queue로 들어가고, 그렇게 되면 Priority Scheduling 논리가 위험을 받으므로 다시 정렬 및 Scheduling!
- (3) 특정 Thread에 New Priority를 Setting할 경우, 당연히 재-정렬 및 Scheduling이 필요!
- (4) Nice 값 수정 시 Priority 변화 가능성으로 인한 재-정렬 및 Scheduling 필요! (이 부분은 Aging Technique 관련이므로 여기서 설명 생략)

이 점을 고려해 다음과 같이 `thread_create`, `sema_up`(in `threads/synch.c`), `thread_set_priority` 함수를 수정하자.

```
tid_t
thread_create (const char *name, int priority,
               thread_func *function, void *aux)
{
    struct thread *t;
    struct kernel_thread_frame *kf;
    struct switch_entry_frame *ef;
    struct switch_threads_frame *sf;
    tid_t tid;

    ASSERT (function != NULL);

    /* Allocate thread. */
    t = palloc_get_page (PAL_ZERO);
    if (t == NULL)
        return TID_ERROR;

    /* Initialize thread. */
    init_thread (t, name, priority);
    tid = t->tid = allocate_tid ();

    /* Stack frame for kernel_thread(). */
    kf = alloc_frame (t, sizeof *kf);
    kf->eip = NULL;
    kf->function = function;
    kf->aux = aux;

    /* Stack frame for switch_entry(). */
    ef = alloc_frame (t, sizeof *ef);
    ef->eip = (void *) (void) kernel_thread;

    /* Stack frame for switch_threads(). */
    sf = alloc_frame (t, sizeof *sf);
    sf->eip = switch_entry;
    sf->ebp = 0;

    /* Add to run queue. */
    thread_unblock (t);

    /* If the priority of newly created thread is bigger than
       the priority of currently running thread, then yield it. */
    if (priority > thread_get_priority ())
        thread_yield ();

    return tid;
}

/* Up or "V" operation on a semaphore. Increments SEMA's value
   and wakes up one thread of those waiting for SEMA, if any.

   This function may be called from an interrupt handler. */
void
sema_up (struct semaphore *sema)
{
    struct thread *entry;
    enum intr_level old_level;

    ASSERT (sema != NULL);

    old_level = intr_disable ();

    /* When the V operation for the counting semaphore is done,
       unblock only the thread that has the highest priorities
       in the waiting list. */
    if (!list_empty (&sema->waiters))
    {
        entry = pop_highest_priority_thread(&sema->waiters);
        thread Unblock (entry);
    }
    sema->value++;
    intr_set_level (old_level);

    /* Re-schedule. */
    thread_yield ();
}

/* Sets the current thread's priority to NEW_PRIORITY. */
void
thread_set_priority (int new_priority)
{
    int old_priority;
    if (thread_mlfqs == true) return;

    /* Set the new priority. */
    old_priority = thread_current ()->priority;
    thread_current ()->priority = new_priority;

    /* If new priority is lower than the old one, then
       re-schedule, cause it's priority scheduler. */
    if (old_priority > new_priority)
        thread_yield ();
}
```

위 함수들을 자세히 살펴보자. 먼저 `thread_create`이다. 최-하단 `return` 명령 바로 위 `if`문만 추가된 부분인데, 새롭게 생성된 Thread의 Priority가 현재 CPU Control을 점유하는 Running Thread의 Priority보다 더 큰지 확인해, 더 크다면 해당 Running Thread가 CPU를 놓고 Ready Queue로 들어가 Ready Queue가 다시 정렬되고 Re-Schedule될 수 있도록 명령이 구성되어 있음을 알 수 있다.

`sema_up`도 확인하자. Semaphore에 대해 P Operation을 호출해놓고 대기하고 있던 Waiting Threads 중 Priority가 가장 높은 Thread를 `pop_highest_priority_thread`라는 함수를 통해 뽑아내고 있다. 이 함수는 본인이 직접 작성한 함수로, 아래와 같이 구성되어 있다. Parameter로 넘겨 받은 List Structure에 대해 단순 Iteration을 돌아, List의 소속 Thread 중 Priority가 가장 높은 Thread를 List에서 Pop하는 함수이다.

```
/* Pops the highest-priority thread in the given linked list */
struct thread *
pop_highest_priority_thread (struct list *_list_)
{
    struct list_elem *iter; struct thread *entry;
    struct list_elem *h_iter; struct thread *highest;
    int highest_priority = PRI_MIN;

    if (list_empty(_list_) == true) return NULL;

    for (ITERATE_LIST((*_list_)))
    {
        entry = EACH_ENTRY(elem);
        if ((entry->priority) >= highest_priority)
        {
            highest = entry; h_iter = iter;
            highest_priority = entry->priority;
        }
        list_remove (h_iter); // pop it!
    }
    return highest;
}
```

참고로, 위 함수와 똑같이 작동하는데 다만 Pop 연산을 수행하지 않는 `top_highest_priority_thread`라는 함수도 정의해놓았다. 이하 설명에서 등장할 것이다.

이렇게 Highest Priority Thread를 Waiter들 중 뽑아내어 Unblock Routine을 호출해 Ready Queue로 넣어준다. Ready Queue로 들어가고 나면, 현재 Running Thread와 새롭게 정렬된 Ready Queue에서의 Highest Thread 중 누가 더 우선순위가 높은지 불확실해진다. 따라서 이를 조정하기 위해 `thread_yield` 함수를 호출하고 있음을 주목하자.

### 3) Advanced Scheduler

앞선 항목에서 언급한 것처럼 Advanced Scheduler의 핵심은 Priority Aging Technique의 구현이다. 그리고 이를 위해선 Fixed-Point Arithmetic 기능이 pintOS에 포함되어야 한다 했다. 본인은 이를 threads Directory 내에 `fixed_point.h` 헤더파일을 만들었으므로써 처리했다. 해당 파일에는 다음과 같이 Fixed-Point Arithmetic을 지원하는 매크로 함수들이 정의되어 있다.

```

#ifndef THREADS_FIXED_POINT_H
#define THREADS_FIXED_POINT_H

#include <stdint.h>

/* Fixed-Point Real Arithmetic

The format of the fixed-point real type in this project(#3)
is just like a figure below.

-----
|   0   /  00000000000000000000000000000000  /  00000000000000000000000000000000 |
| Sign(1)/      Integer(17)      /      Fraction(14)      |
-----

* Total 4 bytes (32 bits) for representing reals
* MSB indicates the sign of a real value
* Next 17 bits are indicating the integer part of real
* Last 14 bits are indicating the fraction part of real

pintOS will use this 'real' type to implement the calculation
of priority aging technique, especially in 'recent_cpu', 'load_avg'
*/
/* Type definitions of 'REAL' type. Users are recommended to
use only 'real_t' for indicating this fixed-point type */
typedef int32_t real32_t;
typedef int64_t real64_t;
typedef real32_t real_t;

/* Macros for readability (meaning) */
#define FRACTION (1 << 14)
// It means the fraction part of real
#define MAKE_REAL(num) num * FRACTION
// Type casting (up_to_real)
#define MAKE_REAL_DOWN(num) num / FRACTION
// Type casting (down_to_real)
#define MAKE_INT(num) num / FRACTION
// Type casting (real_to_integer)

/** Add operations */
/* (1) real_t + int */
#define R_ADD_I(real, integer) (real32_t)(real + MAKE_REAL(integer))
/* (2) int + real_t */
#define I_ADD_R(integer, real) (R_ADD_I(real, integer))
/* (3) real_t + real_t */
#define R_ADD_R(real1, real2) (real32_t)(real1 + real2)

/** Sub operations */
/* (1) real_t - int */
#define R_SUB_I(real, integer) (real32_t)(real - MAKE_REAL(integer))
/* (2) int - real_t */
#define I_SUB_R(integer, real) (real32_t)(MAKE_REAL(integer) - real)
/* (3) real_t - real_t */
#define R_SUB_R(real1, real2) (real32_t)(real1 - real2)

/** Mul operations */
/* (1) int * real_t */
#define I_MUL_R(integer, real) (real32_t)(integer * real)
/* (2) real_t * int */
#define R_MUL_I(real, integer) (I_MUL_R(integer, real))
/* (3) real_t * real_t */
#define R_MUL_R(real, integer) (real32_t)(MAKE_REAL_DOWN((real64_t)real * (real64_t)integer))

/** Div operations */
/* (1) real_t / int */
#define R_DIV_I(real, integer) (real32_t)(real / integer)
/* (2) int / real_t */
#define I_DIV_R(integer, real) (real32_t)(MAKE_REAL(integer) / real)
/* (3) real_t / real_t */
#define R_DIV_R(real1, real2) (real32_t)(MAKE_REAL((real64_t)real1) / (real64_t)real2)

#endif /* threads/fixed_point.h */

```

매크로 함수 정의 형태에 대한 설명은 이미 앞선 항목에서 했기 때문에 생략한다. 핵심은, 'Real Type(본인이 fixed\_point.h에서 정의한 32비트 단위 자료형. int형과 동일함)'과 'Integer Type'이 피-연산자일 때, 덧셈/뺄셈/나눗셈에선 Integer Type에 대한 14 Bits Left Shifting이 이뤄져 연산 위치가 맞춰진다는 것이다. 그 외 나머지 Case에선 일반적인 Binary Operation을 그대로 적용하면 되며, 곱셈/나눗셈 시 Real Type 두 개가 피-연산자인 경우엔 Overflow로 인한 데이터 손실을 방지하기 위한 임시 Casting 작업이 이뤄지고 있음을 기억하자.

이렇게 정의한 Fixed-Pointer Arithmetic을 토대로 아래와 같이 Load Average, Recent CPU, Priority 재-계산 함수를 thread.c에 정의할 수 있다. 먼저 Load Average부터 보자. (이에 앞서 thread.h의 Thread Structure에 Recent CPU와 Priority, Nice Field를 마련하자. 이 세 Variable은 Thread마다 따로 존재해야 한다. Load Average의 경우, System-Wide Value이므로 thread.c 내에 전역 변수로 선언하고, 두 개의 init 함수(thread\_init, init\_thread)에서 초기화하면 된다)

```
/* Recalculate the system-wide load average value based on
   the equation introduced in the pdf file. */
void
recalculate_load_avg (void)
{
    real_t old_load_avg = load_avg, new_load_avg;
    int ready_threads = list_size(&ready_list) +
                        (thread_current() != idle_thread);

    /* Recalculation of load average. */
    new_load_avg = /// "(59*load_avg + ready_threads) / 60" ///
                  R_DIV_I( R_ADD_I( I_MUL_R( 59, old_load_avg ), ready_threads ), 60 );

    load_avg = new_load_avg;
}
```

안내 PDF에 명시된 수식을 코드화한 것에 불과하다. ready\_threads 초기화 시 idle\_thread (CPU 점유 Thread가 없을 때 CPU를 차지하는 Thread)는 Thread 개수로 포함하지 않고 있음

```
/* Recalculate the recent_cpu value of all threads based on
   the equation introduced in the pdf file. */
void
recalculate_recent_cpu (void)
{
    struct list_elem *iter; struct thread *entry;
    real_t old_recent_cpu, new_recent_cpu;
    int nice;

    /* For all the threads in the system. */
    for (ITERATE_LIST(all_list))
    {
        entry = EACH_ENTRY(all elem);
        if (entry == idle_thread)                      // except for the idle thread
            continue;

        old_recent_cpu = entry->recent_cpu;
        nice = entry->nice;

        /* Recalculation of recent_cpu. */
        new_recent_cpu = /// "(2*ld_avg) / (2*ld_avg + 1) * rct_cpu + nice" ///
                      R_ADD_I( R_MUL_R( R_DIV_R( I_MUL_R( 2, load_avg ),
                                              R_ADD_I( I_MUL_R( 2, load_avg ), 1 ) ), old_recent_cpu ), nice );

        entry->recent_cpu = new_recent_cpu;
    }
}
```

을 주목하자.

Recent CPU 재-계산 함수는 좌측과 같다. System 내의 모든 Thread를 추적 관리하는 all\_list를 순회하여, 각 Element(Thread)에 대해 Recent CPU를 수식에 따라 재-계산하고 있음을 주목하자. 참고로, ITERATE\_LIST는 매크로 함수로서, 가독성을 높이기 위한 용도이다. for문 내부를 구성한다.

```

/* Recalculate the priority of all threads in the system
based on the equation introduced in the pdf file. */
void
recalculate_priority (void)
{
    struct list_elem *iter; struct thread *entry, *highest;
    int new_priority, nice;
    real_t recent_cpu;

    /* For all the threads in the system. */
    for (ITERATE_LIST(all_list))
    {
        entry = EACH_ENTRY(allelem);
        recent_cpu = entry->recent_cpu;
        nice = entry->nice;

        /* Recalculation of priority. */
        new_priority = /// " PRI_MAX - (recent_cpu / 4) - (nice * 2) " ///
                      MAKE_INT( R_SUB_I( I_SUB_R( PRI_MAX,
                      R_DIV_I( recent_cpu, 4 ) ), 2 * nice ) );

        if (new_priority > PRI_MAX)
            new_priority = PRI_MAX;
        if (new_priority < PRI_MIN)
            new_priority = PRI_MIN;

        entry->priority = new_priority;
    }

    /* If the highest of recalculated priorities is bigger than
     the priority of currently running thread, re-schedule. */
    highest = top_highest_priority_thread (&ready_list);
    if (highest == NULL) return;

    if ((thread_current ()->priority) < (highest->priority))
        intr_yield_on_return ();
}

```

Priority 재-계산 함수는 좌측과 같다. Recent CPU와 마찬가지로 Priority도 각 Thread마다 개별적으로 존재하는 속성이기 때문에 all\_list를 순회하여 모든 Thread에 대해 일일이 재-계산을 수행하고 있음을 주목하자.

계산된 Priority가 Boundary Case에 걸리면 그것을 조정하는 루틴도 포함되어 있다.

이 함수에서 핵심은 하단 부분이다. 재-계산 이후, Ready Queue 내의 Highest Priority를 뽑아내 재-계산으로 인해 Running Thread보다 우선순위가 높은 Thread가 Ready List에 생겼는지를 확인한다.

만약 그렇다면, Return 이후에 Yield하도록 함수를 설계한다. 참고로, 상기 항목에서도 한 번 언급하였듯, 본 Recalculation Routine은 Timer Interrupt Handling 과정에서 수행되기 때문에 Interrupt 보호를 위해 thread\_yield 대신 pintOS 제공 intr\_yield\_on\_return 함수를 호출하였음을 주목하자. 이 함수는 Thread Yield Routine을 바로 수행하지 않고, Return이 발생하고 나서 수행하도록 Synchronization을 제공한다. 이 점은 본 Step 구현 과정에서 가장 발견이 어려운 포인트 중 하나였다. Thread Scheduling Flow를 명확히 이해하지 못하면 이 포인트를 찾아내기 어려울 것이다.

이렇게 Recalculation 함수 작성이 완료되면 앞서 첨부한 코드와 같이 '(매 Timer Interrupt마다 수행되는) thread\_tick 함수'가 thread\_aging 함수를 호출하도록 설계한다. thread\_aging은 Aging을 실질적으로 수행하는 함수로, 우측과 같이 구성되어 있다. 1초마다 Load Average와 Recent CPU를 계산하고, Time Slice마다 Priority 계산을 함을

```

/* Performs an aging technique.
- recent_cpu of running thread is incremented for every time intr
- recent_cpu and load_avg values are recalculated for every sec
- priorities of all threads in system are recalculated for every q */
void
thread_aging (void)
{
    struct thread *cur_thread = thread_current ();

    cur_thread->recent_cpu = R_ADD_I(cur_thread->recent_cpu, 1);

    /* Recalculation per second. */
    if ((timer_ticks() % TIMER_FREQ) == 0)
    {
        recalculate_load_avg ();
        recalculate_recent_cpu ();
    }

    /* Recalculation per time slice. */
    if ((timer_ticks() % TIME_SLICE) == 0)
        recalculate_priority ();
}

```

주목하자. 또한, 매 Timer Interrupt마다 Currently Running Thread의 Recent CPU가 Increment 되도록 함도 기억하자.

이 밖에도, 본 Phase에서 구현을 요구하는 thread\_(set/get)\_(nice/priority/recent\_cpu/load\_avg)류 함수들도 정의하도록 하자. 다 가볍게 설계할 수 있는데, 딱 하나 예외로, thread\_set\_nice 함수의 경우 아래와 같이 신경을 써야 한다.

```
/* Sets the current thread's nice value to NICE. */
void
thread_set_nice (int nice UNUSED)
{
    struct thread *cur = thread_current (), *highest;
    int cur_nice, new_priority;
    real_t recent_cpu;

    /* Set the new nice value. */
    cur->nice = cur_nice = nice;
    recent_cpu = cur->recent_cpu;

    /* Re-calculate the priority based on the new NICE. */
    new_priority = /// " PRI_MAX - (recent_cpu / 4) - (nice * 2) " ///
                  MAKE_INT( R_SUB_I( I_SUB_R( PRI_MAX,
                                              R_DIV_I( recent_cpu, 4 ) ), 2 * cur_nice ) );

    if (new_priority > PRI_MAX)
        new_priority = PRI_MAX;
    if (new_priority < PRI_MIN)
        new_priority = PRI_MIN;

    cur->priority = new_priority;

    /* If the new priority is lower than the highest
       of current ready list, then re-schedule. */
    highest = top_highest_priority_thread(&ready_list);
    if (highest == NULL) return;

    if ((cur->priority) < (highest->priority))
        thread_yield();
}
```

Nice 값을 Parameter로 넘겨 받은 값으로 설정 후, 해당 New Nice 값을 토대로 다시 Priority를 계산한다. 왜냐? Priority 계산 과정에 Nice가 관여하기 때문이다. 그리고, 이렇게 Priority가 변화하면 무엇을 해야 할까? 그렇다. Ready Queue Re-Ordering 및 Re-Schedule을 해야 한다. 따라서 위 코드 하단부와 같이 top\_highest\_priority\_thread(&ready\_list)를 호출해 Ready Queue의 Highest Priority Thread를 뽑아내고, 그 Thread가 현재 Running Thread보다 우선순위가 큰지를 확인해, 만약 크다면 Thread Yield Routine이 수행되도록 만들어준다.

이렇게 본 Project Phase 구현을 마친다. 이렇게 구현해놓고 init.c와 일부 파일을 PDF 안내 대로 수정하면 최종 목표를 달성할 수 있다. 그 결과는 바로 다음의 항목에서 확인할 수 있다.

## C. 시험 및 평가 내용

### ① make check 수행 결과

위와 같은 설계를 거쳐 완성된 본인의 pintOS make check 수행 결과는 아래와 같다.

```
pass tests/threads/alarm-single
pass tests/threads/alarm-multiple
pass tests/threads/alarm-simultaneous
pass tests/threads/alarm-priority
pass tests/threads/alarm-zero
pass tests/threads/alarm-negative
pass tests/threads/priority-change
pass tests/threads/priority-change-2
FAIL tests/threads/priority-donate-one
FAIL tests/threads/priority-donate-multiple
FAIL tests/threads/priority-donate-multiple2
FAIL tests/threads/priority-donate-nest
FAIL tests/threads/priority-donate-sema
FAIL tests/threads/priority-donate-lower
pass tests/threads/priority-fifo
pass tests/threads/priority-preempt
pass tests/threads/priority-sema
pass tests/threads/priority-aging
FAIL tests/threads/priority-condvar
FAIL tests/threads/priority-donate-chain
pass tests/threads/mlfqs-load-1
pass tests/threads/mlfqs-load-60
pass tests/threads/mlfqs-load-avg
pass tests/threads/mlfqs-recent-1
pass tests/threads/mlfqs-fair-2
pass tests/threads/mlfqs-fair-20
pass tests/threads/mlfqs-nice-2
pass tests/threads/mlfqs-nice-10
pass tests/threads/mlfqs-block
8 of 29 tests failed.
```

본 Project Phase에서 요구하는 모든 Test Case를 정상적으로 pass하였음을 알 수 있다.

### ② priority-lifo.c 코드 및 priority-lifo 테스트 결과 분석

priority-lifo.c Program은 다음과 같이 설계되어 있다. (핵심 부분만 첨부)

```
thread_set_priority (PRI_DEFAULT + THREAD_CNT + 1);
for (i = 0; i < THREAD_CNT; i++)
{
    char name[10];
    struct simple_thread_data *d = data + i;
    snprintf (name, sizeof name, "%d", i);
    d->id = i;
    d->iterations = 0;
    d->lock = &lock;
    d->op = &op;
    thread_create (name, PRI_DEFAULT + 1 + i, simple_thread_func, d);
}
thread_set_priority (PRI_DEFAULT);
```

처음의 Priority Setting은 이 lifo 프로그램이 Thread를 생성하는 동안에는 자신이 생성하는 Thread들보다 무조건 우선순위가 높게 만들어, 중간에 Yield되지 않도록 만들어준다.

이어, THREAD\_CNT(16)만큼 순회를 돌면서 총 16개의 Thread를 순차적으로 생성한다. 단, 이때, 이 Thread들은 한 순회마다 Priority가 높아진다. 그 말은 즉, 마지막에 생성된 Thread일수록 Priority가 높아 먼저 수행될 수 있도록 유도한다.

그래서 LIFO(Last-In, First-Out)이라 하는 것이다. 마지막에 생성된 Thread가 Ready List에 먼저 들어가는 형국이기 때문이다. 그렇게 Schedule된 Thread들은 아래와 같은 Routine을 수행한다. (참고로, 각 Thread의 data->id는 자신의 순회 내 생성 순서임)

```
static void
simple_thread_func (void *data_)
{
    struct simple_thread_data *data = data_;
    int i;

    for (i = 0; i < ITER_CNT; i++)
    {
        lock_acquire (data->lock);
        (*data->op)++ = data->id;
        lock_release (data->lock);
        thread_yield ();
    }
}
```

Child Thread는 단순한 일을 한다. 단순한 할당 연산만 ITER\_CNT(16)번 수행하고 바로 Yield한다. 이때, 할당의 대상이 되는 **공유 자료구조 'data'에 대해 Binary Semaphore를 통한 Mutual Exclusion**이 제공되기 때문에 무조건 먼저 Ready Queue에 들어간 순서대로, 즉, 나중에 생성된 Thread가 먼저 접근해 data 자료구조에 자신의 id를 16번씩 적는 것이다.

그렇다면, 본 Program의 목적은 무엇일까? 그렇다. Priority Scheduling이 정상적으로 이루어져있다면(Aging 및 MLFQs는 없는 상태를 가정한 Program임. 그래서 두 Flag가 올라갈 일이 없음), 가장 나중에 생성된 15번 Thread부터 Descending Order로 각각 16번씩 자신의 id를 화면에 찍어가는, 그러한 결과가 출력될 것이다. Aging이 없는 기초 Priority Scheduler에선 Higher Priority Thread가 무조건 우선 수행되므로!

이러한 점을 인지한 채, 실제 서버에서 'pintos -v -- -q run priority-lifo'를 수행해 테스트 해보면 아래와 같은 결과가 출력된다.

```
SeaBIOS (version 1.13.0-1ubuntu1.1)
Booting from Hard Disk...
PPiLoo hhddaa1
1
Llooaddiinngg.....
Kernel command line: -q run priority-lifo
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer... 337,100,800 loops/s.
Boot complete.
Executing 'priority-lifo':
(priority-lifo) begin
(priority-lifo) 16 threads will iterate 16 times in the same order each time.
(priority-lifo) If the order varies then there is a bug.
(priority-lifo) iteration: 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15
(priority-lifo) iteration: 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14
(priority-lifo) iteration: 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13
(priority-lifo) iteration: 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12
(priority-lifo) iteration: 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11
(priority-lifo) iteration: 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
(priority-lifo) iteration: 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9
(priority-lifo) iteration: 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8
(priority-lifo) iteration: 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7
(priority-lifo) iteration: 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
(priority-lifo) iteration: 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
(priority-lifo) iteration: 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
(priority-lifo) iteration: 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
(priority-lifo) iteration: 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
(priority-lifo) iteration: 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
(priority-lifo) iteration: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
(priority-lifo) end
Execution of 'priority-lifo' complete.
Timer: 29 ticks
Thread: 0 idle ticks, 29 kernel ticks, 0 user ticks
Console: 1557 characters output
Keyboard: 0 keys pressed
Powering off...
```

예상대로 결과가 출력되었음을 확인할 수 있다. 즉, 본인의 pintOS는 **Priority Scheduling이 정상적으로 수행되고 있는 것이다**. Program에 의해 생성된 Thread가, 가장 나중에 생성된 Thread부터 수행되어 15, 14,..., 1, 0으로 각각 16번씩 화면에 출력되고 있음을 보자.

한편, 본 Program의 수행은 총 29개의 Timer Interrupt의 시간이 걸렸다고 나오는데, 다른 Test Case에 비해서 훨씬 적은 시간이 걸리는 Program임을 알 수 있다. Idle Thread가 수행되는 상황도 없음을 알 수 있다. idle ticks 값이 0이기 때문이다.

이렇게 해서 본 'Project 3 – Threads' 개발을 마친다. 다음 Phase도 열심히 수행하도록 하겠다.