**Pintos Project 3: Threads**

담당 교수 : 문의현 교수님

학번 / 이름 : 20191112 / 심은서

개발 기간 : 2022. 12. 02 ~ 2022. 12. 05

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술.

현재 pintos는 각 process의 우선순위를 고려하지 않고 round-robin 방식의 스케줄링을 사용하고 있다. 따라서 이를 개선하기 위해 process에 우선순위를 부여하고 우선순위에 따라서 process가 작동 하도록 alarm clock, priority scheduling, advanced scheduler를 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* 아래 각 항목 개발의 필요성 또는 개발 시 기대되는 결과를 간략히 서술
  1. Alarm Clock

기존의 timer\_sleep() 함수에서는 thread가 running state와 ready state를 오가며 반복하게 하도록 함으로써 busy-waiting 방식으로 작동하고 있는데, 이는 cpu 자원 낭비를 초래하여 비효율적이므로 이 방식을 개선한다.

* 1. Priority Scheduling

기존의 process 스케줄링 방식은 round-robin으로, thread간의 우선순위가 고려되지 않는다. 따라서 thread가 각자의 우선순위를 가지도록 하고 그 우선순위에 맞게 실행되게 하여 효율성을 높인다. 더불어, 그저 우선순위에 맞게만 진행하면 낮은 우선순위의 thread가 starvation을 겪을 수 있으므로, 이를 막기 위해 thread가 ready list에 머무르는 시간에 비례하여 우선순위를 높이는 aging을 구현한다.

* 1. Advanced Scheduler (추가구현을 한 경우)

BSD scheduler에서는 각 우선순위가 각자의 ready queue를 가지는 multi level feedback queue를 이용하여 구현한다. 각 ready queue는 round-robin 방식으로 구현되며, 우선 순위가 높은 queue부터 스케줄링을 진행한다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

block 상태인 thread를 관리하기 위하여 list를 만들고, 이들이 언제 깨어나야 할지를 저장하는 변수를 만든다. 매 틱마다 이 list에 들어있는 thread 중에서 깨어나야 할 시간이 된 thread가 있는지 조사하여 있다면 이들을 ready list에 넣고 sleep list에서는 제거한다.

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

현재 실행중인 thread보다 높은 우선순위를 갖는 thread가 새로 들어온다면 현재 실행중인 thread와 새로운 thread사이에 switching을 하여 다시 스케줄링을 해야 한다. 따라서 우선순위에 맞게 queue가 관리되도록 하기 위하여, 우선순위 순으로 queue에 thread를 집어넣을 수 있도록 기존의 list\_push\_back()이 아닌 list\_inserted\_ordered()를 사용한다.

1. Advanced Scheduler에서 priority 계산에 필요한 각 요소를 서술. (추가구현을 한 경우)

priority는 다음의 식과 같이 계산된다.

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

* + - PRI\_MAX : 우선순위의 최댓값
    - recent cpu : thread의 최근 cpu 사용량으로, 이 값이 높으면 우선순위를 낮춘다.
    - nice : -20~20의 범위를 가지며, 이 값이 높을 수록 다른 thread에게 더 cpu를 양보한다.

recent cpu는 다음과 같이 계산된다.

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

* + - load avg : ready 상태에 있는 thread 수의 평균값이며, 다음과 같이 계산된다.

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

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성
* 12.02 : 프로젝트 내용 분석과 명세서 이해
* 12.03 : alarm clock 및 priority scheduling 구현
* 12.04 : advanced scheduler 구현
* 12.05 : 보고서 작성
  1. **개발 방법**
* II. B.의 개발 내용을 구현하기 위해 각각에 대해 다음 사항들을 포함하여 설명
  + 수정해야하는 소스코드
  + 수정하거나 추가해야 하는 자료구조
  + 수정하거나 추가해야 하는 함수

1. alarm clock

alarm clock을 개선하기 위해서는 block 상태의 thread들을 관리해야 한다. 따라서 thread.c에서 sleep\_list를 만들고 이를 thread\_init()에서 초기화한다. 또한 이 thread들이 언제 깨어나야할 지를 저장하기 위해 thread.h에서 thread struct에 wakeup\_time변수를 추가하고 init\_thread()에서 이를 초기화한다.

time.c의 timer\_sleep()에서 기존의 비효율적인 busy waiting 방식을 지우고, thread를 sleep 시키는 방식을 sleep\_list와 wakeup\_time을 이용하여 다시 구현한다. 이 thread가 일어나야 할 tick을 wakeup\_time에 저장하고 sleep\_list에 추가하고, 다시 스케줄링한다.

또한 매 틱마다 깨워야할 thread가 있는지 확인하고 깨우기 위하여 time.c의 timer\_interrupt에서 깨울 thread가 있다면 sleep\_list에서 이를 삭제하는 작업을 구현한다.

1. priority scheduling

priority scheduling을 위해서는 말그대로 우선순위를 고려해야 한다. 이말인 즉슨 thread의 우선순위가 높은 thread부터 실행되도록 해야하며, 우선순위에 따라서 queue가 정렬되어야 한다. queue가 뒤바뀌어야 하는 상황에서 우선순위를 잃지 않도록 한다. 따라서 새로운 thread가 생성되는 thread.c의 thread\_create(), thread가 unblock되는 thread\_unblock(), thread가 cpu를 양보하는 thread\_yield(), 우선순위를 변경하는 thread\_set\_priority()에서 모두 우선순위에 따라서 thread를 재스케줄링하도록 변경한다. thread\_create()과 thread\_set\_priority()에서는 현재 실행중인 thread와 새로운 priority를 비교하고, thread\_unblock()과 thread\_yield()에서는 queue에 thread를 넣을 때 list\_insert\_ordered()를 사용함으로써 이를 구현할 수 있다. 또한 list\_insert\_ordered()를 사용하기 위해서는 비교함수를 새로 구현하도록 한다. 추가로, sync.c의 sema\_down()에서도 waiters에 thread를 삽입하고 block하므로 이 과정에서 priority를 고려하여 삽입되기 위해 list\_insert\_ordere()를 사용한다. sema\_up()에서는 thread가 waiters에 있는 동안 priority가 변경되었을 수 있기 때문에, list를 다시 정렬한다. 또한 priority가 높은 thread에게 양보하도록 한다.

여기까지 하면 priority scheduling이 구현되기는 하나, 우선순위가 낮은 process는 starvation상태에 빠질 수 있으므로 이를 개선하기 위하여 aging기법을 사용한다. 이를 위해 thread.h에 thread\_prior\_aging 변수를 추가하고 thread\_tick()에서는 thread\_prior\_aging flag가 true라면 매 틱마다 모든 thread의 priority를 1씩 높여주도록 한다.

1. advanced scheduler

recent cpu, nice, load avg 값을 이용하여 priority를 계산하고 업데이트 해야 한다. 따라서 thread 구조체에 nice와 recent\_cpu를 추가하고 thread\_init()에서 초기화한다. thread.c에 load\_avg를 추가하고 tread\_start()에서 초기화한다. 이 때 recent cpu와 load avg 값은 실수인데 pintos에서는 부동 소수점 연산을 지원하지 않으므로 fixed float point를 사용하여 연산할 수 있도록 실수 연산을 위한 함수들을 만들어야 한다.

advanced scheduler에서는 매 4tick마다 모든 thread의 priority가 다시 계산된다. 또한 running state인 thread의 recent\_cpu값은 idle thread가 아닐 경우 매 초마다 1씩 증가하도록 한다. load\_avg 값 또한 식에 맞게 매 초마다 업데이트 되며, 이렇게 priority가 달라짐에 따라 thread의 선점도 달라져야 한다.

따라서 thead.c에 load\_avg, priority, recent\_cpu를 계산하고 업데이트하는 함수를 만든다. 그리고 이를 매 틱마다 또는 4틱마다 실행되도록 하기 위하여 thread\_tick()에서 알맞게 함수를 호출한다.

1. **연구 결과**
   1. **Flow Chart**

* II. B. 개발 내용의 각 항목에 대하여 Flow Chart 작성  
  (추가구현에 대해서는 flow chart를 작성하지 않아도 됨)
* Alarm clock
* Diagram

  Description automatically generated
* Priority scheduling

Diagram

Description automatically generated

* 1. **제작 내용**
* II. B. 개발 내용의 각 항목에 대하여 실질적으로 구현한 코드의 관점에서 작성 (구현 내용, 알고리즘 등을 명확히 서술할 것)
  + 구현에 있어 Pintos에 내장된 라이브러리나 자체 제작한 함수를 사용한 경우 이에 대해서도 설명
* 개발 중 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결한 방식에 대해 설명

1. alarm clock

block 상태의 thread를 저장하고 올바른 때에 깨우기 위하여 sleep\_list와 wakeup\_time을 각각 thread.h에 추가했다.

Text

Description automatically generated



time.c의 timer\_sleep()에서 기존의 비효율적인 busy waiting 방식을 지우고, wakup\_time에 thread가 일어나야 할 시간을 저장한 뒤 sleep\_list에 thread를 push하고, 해당 thread를 block시키는 방식으로 재구현한다.

Text

Description automatically generated

이렇게 sleep\_list에서 block state인 thread가 관리된다면, time.c에서는 이를 확인하여 적절한 때에 깨우는 thread\_wake\_up() 함수를 새로 만들었다. sleep\_list에 있는 thread를 탐색하며 해당 thread의 wakeup\_time이 현재 tick보다 작다면 sleep\_list에서 해당 thread를 지우고 unblock처리한다.

A screenshot of a computer

Description automatically generated with medium confidence

매 틱마다 실행되는 timer\_interrupt()에서 이 함수를 호출한다.

Text

Description automatically generated

1. priority scheduling

thread.c의 thread\_create()에서 새로 만들어지는 thread의 우선순위가 현재 thread의 우선순위보다 높으면 우선순위를 양보하도록 수정한다.

Text

Description automatically generated

priority scheduling을 구현하려면 thread가 queue에서 우선순위 순으로 정렬이 되어야 한다. 따라서 thread가 우선순위에 맞게 정렬되어 queue에 넣어지기 위하여, thread\_unblock()에서 기존의 list\_push\_back()이 아닌 list\_inert\_ordered()를 사용한다.

A screenshot of a computer

Description automatically generated with medium confidence

이때, 우선순위를 비교하는 함수가 필요한데, 이를 위해 두개의 thread의 priority를 비교하는 thread\_priority\_compare() 함수를 만든다.

Text

Description automatically generated

thread\_yield()에서 또한 thread의 순서가 뒤바뀌므로, 우선순위 순서대로 정렬되도록 기존의 list\_push\_back()이 아닌 list\_inert\_ordered()를 사용한다.

Text

Description automatically generated

우선순위를 변경하는 thread\_set\_priority()에서도 이를 고려해야 한다. 우선순위가 변경되면 다시 스케줄링되어야하기 때문이다. 따라서 현재 thread보다 새로 설정되는 우선순위가 더 높다면 양보하도록 한다. advanced scheduler에서는 우선순위를 임의로 변경할 수 없으므로, mlfqs 모드일 때는 그냥 return해준다.

Text

Description automatically generated

sync.c의 sema\_down에서 또한 waiters에 thread를 삽입하므로, 우선순위에 맞게 삽입할 수 있도록 list\_insert\_ordered()를 사용하는 것으로 수정한다.

Text

Description automatically generated

sema\_up에서는 또한 우선순위 순으로 다시 정렬하기 위해 list\_sort()를 사용하고, 우선순위가 높은 thread가 cpu를 선점할 수 있도록 priority를 비교한 후 다시 스케줄링한다.

Text

Description automatically generated

추가로 aging을 적용해야 한다. 이를 위해 thread.h에 thread\_prior\_aging 변수를 추가하고, init.c에 다음과 같은 내용을 추가하였다.

Text

Description automatically generated

Text

Description automatically generated

thread.c에 매 틱마다 모든 thread의 priority를 1씩 증가하도록 하는 thread\_aging()함수를 만들었다. 이 때 priority가 PRI\_MAX를 초과하지 않도록 한다.

Text

Description automatically generated

thread\_aging()함수를 매 틱마다 실행되는 thread\_tick()함수에서 호출한다.

A picture containing text, orange

Description automatically generated

1. advanced scheduling

먼저 priority 계산을 위해 필요한 nice, recent\_cpu 변수를 thread 구조체에 추가하고 초기화하였다.

A picture containing text, orange, dark

Description automatically generated Text

Description automatically generated

load\_avg변수 또한 새로 선언하고 초기화하였다.

Text

Description automatically generated

그리고 다음과 같이 priority 계산에 필요한 실수 연산 함수들을 구현하였다.



**A screenshot of a computer

Description automatically generated with medium confidence** **Text

Description automatically generated**

이제 load\_avg, recent\_cpu, priority 각각을 계산하고 업데이트 하는 함수를 구현한다.

먼저 load\_avg는 다음과 같은 식으로 계산되므로 이에 적절한 실수 연산 함수를 사용한다.

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

이 식대로 update\_load\_avg()함수를 구현한다.

Text

Description automatically generated

다음으로는 recent\_cpu를 계산하는 update\_recent\_cpu()이다.

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

Text

Description automatically generated

다음은 update\_priority()함수이다.

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

Text

Description automatically generated

세가지 함수를 구현함에 있어서 실수와 정수를 구분하는 것이 굉장히 헷갈렸다.

이제 매 틱마다 세가지가 변동되고 적용되도록 하기 위하여 세 함수를 thread\_tick() 함수에서 적절하게 호출하여야 한다.

/ Text

Description automatically generated먼저 매 틱마다 idle thread가 아니라면 recent\_cpu는 1씩 증가하고, 위에서 보인 식에 의해 업데이트 되므로 idle thread가 아닌 모든 thread에 대해 update\_recent\_cpu()를 호출한다. 그리고 매 틱마다 load\_avg또한 update되므로 idle thread가 아니라면 update\_load\_avg()를 호출한다.

또한 매 4틱 (TIME\_SLICE)마다 priority를 업데이트해야한다. 따라서 모든 thread에 대하여 update\_priority()를 호출한다.

추가로 thread.c 에서 int thread\_get\_nice (void), void thread\_set\_nice (int), int thread\_get\_recent\_cpu (void), int thread\_get\_load\_avg (void)를 구현해야 한다.

먼저 thread\_get\_nice는 현재 thread의 nice값을 반환한다.

Graphical user interface, text

Description automatically generated with medium confidence

thread\_set\_nice()는 현재 thread의 nice값을 인자로 받은것으로 변경한 후 우선순위를 업데이트 한다. 따라서 nice값을 갱신한 뒤에 update\_priority()를 호출한다.

Text

Description automatically generated

thread\_get\_recent\_cpu()는 100과 현재 thread의 recent\_cpu를 곱한 값을 반환한다. 따라서 아까 구현한 실수 연산 함수를 이용하여 100을 곱한다. 이 때, 계산값과 가장 가까운 정수를 반환하도록 rounding한다.

A screenshot of a computer

Description automatically generated with medium confidence

thread\_get\_load\_avg() 또한 100과 현재 load\_avg를 곱한 값을 반환하며, 라운딩한다.

Text

Description automatically generated

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

Text

Description automatically generated

priority-lifo.c에서는 1부터 15까지의 priority를 갖는 thread을 생성하여 실행시키는데, 같은 priority를 같은 thread들이 존재한다. 따라서 priority가 15인 (가장 큰) thread들은 모두 가장 가장 나중에 ready list로 들어갔는데도 가장 먼저 수행되고 종료되었다. 반대로 가장 낮은 priority를 갖는 0은 가장 먼저 ready list로 들어갔는데도 가장 나중에 실행되고 있다. priority에 따라 실행 순서가 달라지는 priority scheduling이 구현된 것을 볼 수 있다.

* make check 수행 결과를 캡처하여 첨부

Text

Description automatically generated

advanced scheduling 구현을 시도하였으나, 몇개의 mlfqs test는 통과하지 못하였다.