**Pintos Project 3: Threads**

담당 교수 : 김영재

학번 / 이름 : 20161564 곽우철

개발 기간 : ~11/6

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술.

Alarm clock이 running, ready를 왔다 갔다 하는 비효율적인 방식으로 되어있는데 이를 효율적으로 바꾼다. 또한, scheduling의 방식도 round-robin만 사용하고 있는데 여기에 priority 요소를 넣어서 priority scheduling이 되도록 한다. 마지막으로 multi-level feedback queue를 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

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

위에서 말한 것과 같이 thread들이 running과 ready를 오고 가고 있기 때문에 새로운 queue(sleep\_check\_list)를 하나 만들고 매 tick마다 wakeup 해야 하는 thread가 어떤 것들인지 확인하는 alarm과 같은 역할을 하도록 한다(unblock).

* 1. Priority Scheduling

이전에는 priority를 고려하지 않고 thread가 ready\_list의 끝에 들어갔지만 priority를 고려해서 넣어주게 되면 우선순위가 높은 thread가 preemptive하게 실행된다. 또한, 여기서 생기는 starvation문제를 aging을 통해서 해결해준다.

* 1. Advanced Scheduler (추가구현을 한 경우)

thread마다 cpu 점유 시간이 다르기 때문에 priority값을 조정을 해주는 작업을 하는 advanced scheduler를 구현한다. 이는 이전 작업 때 사용한 cpu 점유율 등이 고려되는데 cpu를 많이 점유하면 priority가 낮아지게 된다. load\_avg의 update는 매 tick마다 이루어지고 priority의 update는 4tick마다 이루어지도록 한다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

block된 thread들을 관리하는 queue를 만들고 여기에 넣어줄 때 wakeup time도 같이 지정한다(start+ticks로). 그러면 ticks를 매번 1씩 증가시키는 timer\_interrupt 함수가 queue를 돌면서 현재 ticks와 wakeup time을 비교하여 시간이 지났으면 해당 thread를 unblock함으로써 깨우는 것이다. 물론 이 queue에서 제거도 해야 할 것이다.

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

preemptive한 것이기 때문에 더 높은 priority의 thread가 들어오면 이것이 먼저 scheduling되도록 해야 한다. 그러므로 current thread의 priority와 create되는 thread의 priority를 thread\_create 함수에서 비교하여 만약 후자가 더 높으면 thread\_yield 함수를 통해 바뀌도록 한다. test case에 sema의 경우도 있으므로 sema\_up에서도 비슷하게 priority를 고려하도록 한다.

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

recentcpu, nice 요소들이 필요한데 이를 thread.h에 추가하면 된다. 각각을 살펴보면 recentcpu의 경우는 앞에서 말한 것과 같이 thread의 priority 결정을 위해 cpu 점유 시간과 비례하는 값이다. 다음과 같은 식으로 매초 update 된다.

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

nice는 각 thread마다 가지는 -20~20 사이의 정수 값이고 처음에 0 혹은 parent의 nice값을 initialize 된다. 계산에 이용되는 load\_avg는 ready state에 load된 평균 thread 개수로 매초 update되는데 다음식과 같다.

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

최종적으로 priority는 4ticks마다 update 되고 다음 식과 같다.

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

PRI\_MAX는 pintos에서 thread가 가질 수 있는 최대 priority 값으로 63이고 PRI\_MIN은 반대로 0이다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

alarm clock- 중간 끝나고 10/27일 정도에 시작해서 과제 이해하고 alarm clock 구현 완료까지.

priority clock, aging및 추가구현- 그 주 주말까지 완료.(~10/31)

보고서 작성- 제출 주에 내용 정리하고 하루 잡아서 작성.

* 1. **개발 방법**
* II. B.의 개발 내용을 구현하기 위해 각각에 대해 다음 사항들을 포함하여 설명
  + 수정해야하는 소스코드/함수

alarm clock의 경우 device쪽에서 timer.c의 timer\_sleep()에서 wakeup time 지정, queue에 넣어주는 작업. timer\_interrupt에서 wake 돼야할 thread 선정하고 unblock하는 작업.

priority schedule의 경우 thread.c에서 priority 비교해주는 함수 추가(comp\_priority), thread\_unblock과 thread\_yield에서 list\_insert\_ordered로 정렬해주기, thread/synch.c에서 sema\_up도 마찬가지로 해주기, thread\_create에서 현재 priority와 비교해주고 yield 조건적으로 하는 코드 추가, thread\_set\_priority 함수도 새로 set한 것의 priority 비교 후 조건적으로 yield.

aging/ advanced scheduler에서 강의자료 나와있는 것과 같이 bool thread\_prior\_aging thread.h와 thread.c에 추가하기. threads/init.c에 name이 -aging인경우 thread\_prior\_aging이 true되도록 수정, recentcpu 및 nice thread.h에 추가하기 및 thread.c에서 초기화. thread.c에서 load\_avg global 변수로 선언, thread.c에서 thread\_get\_recent\_cpu, thread\_get\_load\_avg, thread\_get\_nice, thread\_set\_nice등의 함수 구현, priority와 load\_avg, recentcpu update해주는 함수 구현. 그리고 마지막으로 devices/timer.c에서 timer\_interrupt 함수 안에 update되는 함수 넣어주기.

* + 수정하거나 추가해야 하는 자료구조

block된 thread들 관리해주는 queue(list) 추가

static struct list sleep\_check\_list;

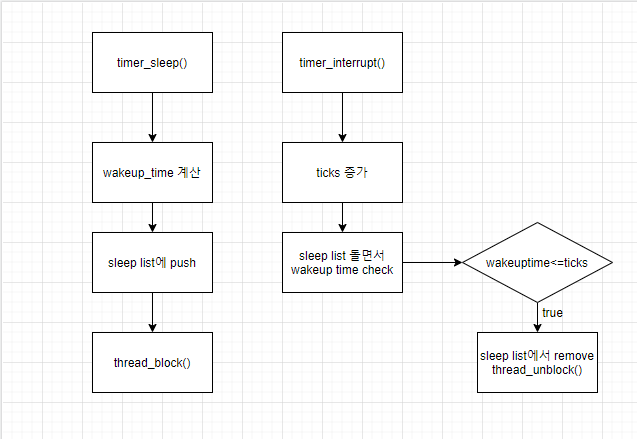
* + 수정하거나 추가해야 하는 함수

(위에)

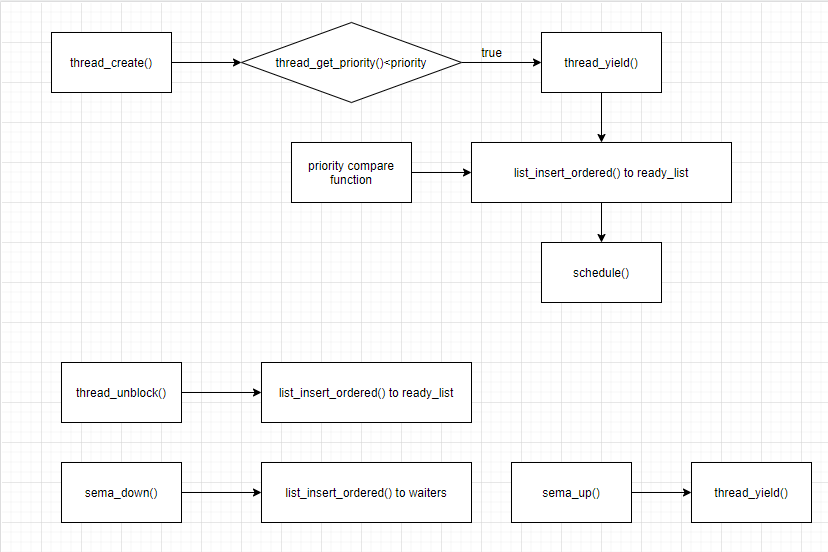
1. **연구 결과**
   1. **Flow Chart**

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

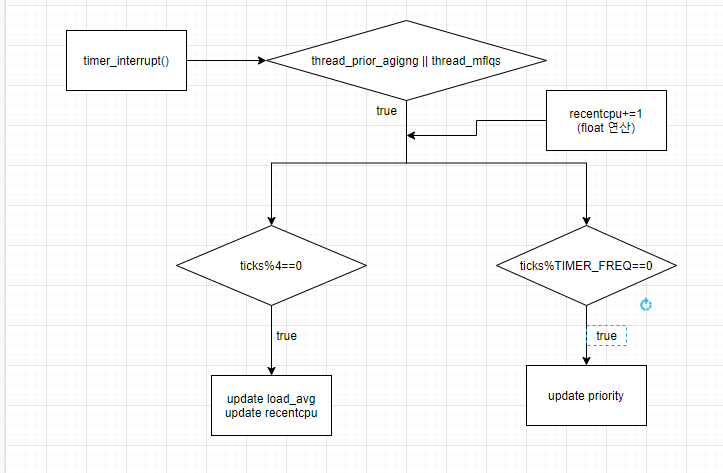
**alarm clock**



**priority scheduling**



**advanced scheduler/aging**



* 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

static struct list sleep\_check\_list;로 block된 thread들을 관리해주는 queue 생성한다.

timer\_sleep함수에서 수정 사항은 다음과 같다.

int64\_t wake\_time=(start+ticks);

  thread\_current()->waketime=wake\_time;

  list\_push\_back(&sleep\_check\_list,&((thread\_current())->elem));

  thread\_block();

wake해야 할 시간을 계산하고 이 값을 현재 thread의 waketime 요소(thread.h에 미리 추가)에 넣는다. 그리고 sleep\_check\_list에 elem을 넣고 block 상태로 만든다.

다음은 timer\_interrupt 함수의 수정사항이다.

ticks++;

  // find the threads that need to be woken up

  struct list\_elem\*temp=list\_begin(&sleep\_check\_list);

  struct thread\*tempt=NULL;

  while(1)

  {

  if(temp==list\_end(&sleep\_check\_list))

    break;

  tempt=list\_entry(temp,struct thread,elem);

  if(tempt->waketime<=ticks)

  {

    temp=list\_remove(temp);

    thread\_unblock(tempt);

    temp=list\_prev(temp);

  }

  temp=list\_next(temp);

  }

temp로 list\_elem에 접근해서 sleep\_check\_list를 돌면서 wakeuptime이 지난 것들을 list에서 제거하고 unblock시켜주는 작업이다.

priority scheduling을 위해 thread.c에서 다음과 같이 함수들을 수정한다. 먼저 thread\_create에서 현재 priority와 create하는 것의 priority를 비교하고 후자가 클 시 thread\_yield()를 호출한다.

  int run\_priority=thread\_get\_priority();

  if(priority>run\_priority)

    thread\_yield();

thread\_yield 함수는 다음과 같은데 list\_insert\_orderd로 정렬해서 ready list에 넣어주는 것을 확인할 수 있다.

  old\_level = intr\_disable ();

  if (cur != idle\_thread)

    list\_insert\_ordered(&ready\_list, &cur->elem,comp\_priority,NULL);

  cur->status = THREAD\_READY;

  schedule ();

여기서 정렬에 이용되는 함수가 comp\_priority로 다음과 같다.

bool comp\_priority(const struct list\_elem\*a,const struct list\_elem\*b,void\* aux)

{

  bool result=true;

  struct thread\* x=list\_entry(a,struct thread,elem);

  struct thread\* y=list\_entry(b,struct thread,elem);

  if((x->priority)<=(y->priority))

    result=false;

  return result;

}

그리고 나서 thread\_unblock와 sema\_down에서 앞에서 말한 것과 같이 각각 ready list와 waiters에 list\_insert\_ordered로 넣어줘서 priority라는 요소를 가지게 한다.

advanced scheduler와 aging을 위해서 priority와 load\_avg, recentcpu를 update하는 함수들을 timer.c에서 다음과 같이 호출한다. 각각 1ticks, 그리고 4ticks의 추기로 update되도록 한 것을 확인할 수 있다. (thread.h에 recentcpu, nice값 추가 및 thread.h, thread.c에 bool thread\_prior\_aging 추가하는 코드 생략)

if(thread\_prior\_aging||thread\_mlfqs)

  {

    thread\_current()->recentcpu+=FRACTION;

    if(timer\_ticks()%TIMER\_FREQ==0)

      update\_sec();

    if(timer\_ticks()%4==0)

      update\_4sec();

  }

load\_avg와 recentcpu를 update하는 코드는 다음과 같다.

void update\_sec(void)

{

  int ready;

  if(thread\_current()!=idle\_thread)

    ready=list\_size(&ready\_list)+1;

  else

    ready=list\_size(&ready\_list);

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

  int temp=(59\*load\_avg);

  temp=temp+ready\*FRACTION;

  load\_avg=temp/60;

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

  struct list\_elem\*e=list\_begin(&all\_list);

  struct thread\*temp2;

  while(1)

  {

    if(e==list\_end(&all\_list))

      break;

    temp2=list\_entry(e,struct thread,allelem);

    if(temp2!=idle\_thread)

    {

      int a=2\*load\_avg;

      int b=2\*load\_avg;

      b=b+1\*FRACTION;

      int64\_t tempp=(int64\_t)a;

      tempp=tempp\*FRACTION/b;

      int c=(int)tempp;

      tempp=(int64\_t)c;

      tempp=tempp\*(temp2->recentcpu)/FRACTION;

      c=(int)tempp;

      temp2->recentcpu=c+(temp2->nice)\*FRACTION;

    }

    e=list\_next(e);

  }

}

floating point 연산을 허용하지 않기 때문에 간단한 사칙연산 코드가 위와 같이 길어졌다. 여기서 FRACTION이란 1<<14로 32bit중에 하위 14bit가 fraction이기 때문에 연산을 위해 미리 define 해 두었다. floating point연산 중 정수-실수=정수\*FRACTION-실수로, 정수+실수=정수\*FRACTION+실수로 계산하면 된다. 실수끼리의 곱셈, 나눗셈은 overflow를 고려하여 int64\_t로 바꿔서 마찬가지로 FRACTION을 이용해서 하면 된다.

다음으로는 priority를 update하는 함수이다.

void update\_4sec(void)

{

  //priority update

  // priority=PRI\_MAX-(recentcpu/4)-(nice\*2)

  struct list\_elem\*e=list\_begin(&all\_list);

        struct thread\*temp2;

  while(1)

  {

    if(e==list\_end(&all\_list))

                        break;

    temp2=list\_entry(e,struct thread,allelem);

    int a=(temp2->recentcpu)/4;

    a=PRI\_MAX\*FRACTION-a;

    int b=(temp2->nice)\*FRACTION\*2;

    temp2->priority=(a-b)/FRACTION;

    if(temp2->priority>PRI\_MAX)

      temp2->priority=PRI\_MAX;

    else if(temp2->priority<PRI\_MIN)

      temp2->priority=PRI\_MIN;

    e=list\_next(e);

  }

  if(!list\_empty(&ready\_list))

  {

    temp2=list\_entry(list\_begin(&ready\_list),struct thread,elem);

  }

  if(temp2->priority>thread\_current()->priority)

    intr\_yield\_on\_return();

}

마찬가지로 실수 연산을 해주고 PRI\_MAX, PRI\_MIN으로 범위 체크를 해준다. 그리고 현재 priority와 ready\_list의 priority 최대를 비교하여 더 클 시 yield한다(interrupt 도중이니 intr\_yield\_on\_return()사용). thread\_get\_load\_avg(), thread\_get\_nice(), thread\_set\_priority(), thread\_set\_nice() 등의 함수는 그저 값을 가져오거나 지정한 값을 넣어주면 되는 것이기에 생략한다.

**시험 및 평가 내용**

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

priority-lifo.c에서 test\_priority\_lifo 함수는 다음과 같다.

* 텍스트이(가) 표시된 사진

  자동 생성된 설명텍스트이(가) 표시된 사진

  자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명lifo는 last in first out이므로 마지막에 들어간 것이 제일 먼저 나와야 한다. 위 코드를 살펴보면 처음에 thread\_set\_priority로 PRI\_DEFAULT+THREAD\_CNT(16)+1로 한다. 그리고 for문으로 16개의 thread들을 더 create하는데 각각의 priority를 PRI\_DEFAULT+i+1로 하므로 나중일수록 커지도록 한 것이다. for 문 이후로 처음 thread를 다시 그냥 PRI\_DEFAULT로 priority를 설정하면 생성한 것 들보다 낮아지므로 list에 있는 priority 높은 것(즉 마지막에 넣은 ~+15)가 schedule되는 것이다. 그 다음부터는 simple\_thread\_func()를 수행하면서 두번째 for문에서 data id를 print하는 것이다. 여기서 함수 안의 for문이 ITER\_CNT(16)만큼 있으므로 결과에 한 줄 당 16개씩 print되는 것이고 함수 끝에 있는 thread\_yield()의 경우 어차피 그 thread가 priority가 높기 때문에 그냥 같은 줄에 같은 숫자가 나오는 것이다. 그래서 결과를 보면(아래 사진) 15부터 0까지 한 줄에 16개씩 반복되는 것을 볼 수 있다.

* make check 수행 결과를 캡처하여 첨부

**make check시 mlfqs부분에서 시간이 좀 오래 걸리지만 pass는 됨!**

* 텍스트이(가) 표시된 사진

  자동 생성된 설명