**Pintos Project 3: Threads**

담당 교수 : 김영재 교수님

조 / 조원 : 김하늘(20181618)

개발 기간 : 2020.11.24 ~ 2020.12.06

1. **개발 목표**

* Pintos에서 OS의 scheduling 기법들을 이해하고 적용하여 thread scheduling 방식들을 개선한다. 즉 기본적인 priority scheduler를 구현하고 multi-level feedback queue를 이용한 BSD scheduler를 구현하여 scheduling에 대한 이해도를 높인다.

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

변경 전 프로그램의 sleep 상태는 busy wating 방식으로 구현되어있었다. 이는 CPU의 효율을 낮추기때문에 sleep queue를 만들고 Alarm clock 부분을 변경한다. thread를 block 상태로 만든 후 timer interrupt 함수 내에서 특정 시간이 지나면 해당 thread를 깨우는 방식으로 구현한다.

* 1. Priority Scheduling

우선순위에 따라 thread를 실행시키는 priority scheduling을 구현한다. 이떄 priority가 바뀌면 가장 높은 thread가 수행되도록 preemptive 형식으로 구현하며, 모든 thread들이 최소한의 cpu 제어권을 가질 수 있도록 aging 기법을 사용하여 priority를 조정해준다.

* 1. Advanced Scheduler (추가구현을 한 경우)

thread에 nice, recent cpu time 정보를 사용해 priority를 특정 시간마다 계산해 더 효과적으로 실행시킬 수 있는 BSD scheduler을 구현한다.

* 1. **개발 내용**

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

스레드를 sema\_down을 사용하여 block시킨 뒤 sleep list에 추가하여 해당 스레드를 sleep 상태로 만든다. 이때 해당 스레드에 wake up 시간을 저장해 둔 뒤, timer interrupt를 할때마다 sleep list를 돌면서 wake up 시간인지 확인한 뒤 깨워야 할 경우 sleep list에서 지우고 sema\_up을 호출한다.

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

thread를 깨울때 호출하는 thread\_unblock()과 thread\_yield()함수에서 thread를 ready list에 추가할때, 기존에는 리스트의 끝에 붙이는 형태였지만 list\_insert\_ordered()를 사용하여 priority 순서대로 정렬될 수 있도록 하였다. 이를 위해 priority를 비교 연산하는 함수를 만들어 파라미터에 사용하였다. 만약 실행중인 thread보다 높은 priority를 가질 경우 다시 스케쥴링해주어야한다.

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

thread간의 서로다른 scheduling need를 균형있게 충족시키기 위해 nice와 recent cpu time을 이용하여 priority를 계산한다. nice는 해당 thread가 다른 threads에게 얼마나 CPU yield에 영향을 미치는 지에 대한 값이다. niceness가 작을수록 priority는 높아지게 된다. recent cpu time은 thread가 최근에 사용한 cpu time이며 이 값을 계산하는 공식은 다음과 같다.

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

이때 load\_avg는 ready list에 존재하는 thread의 moving average 값이다. 이는

load\_avg = (59/60) \* load\_avg + (1/60) \* (sleep상태가 아닌 thread 수)

이며, 이를 이용하여

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

와 같이 priority를 계산할 수 있다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* 2020.11.24 ~ 2020.11.26 메뉴얼 정독 및 구현 계획 수립
* 2020.11.26 ~ 2020.11.28 Alarm clock 구현
* 2020.11.29 ~ 2020.12.3 priority scheduler, priority aging 구현
* 2020.12.3 ~ 2020.12.5 BSD scheduler 추가 구현
* 2020.12.5 ~ 2020.12.6 보고서 작성
  1. **개발 방법**
* Alarm Clock

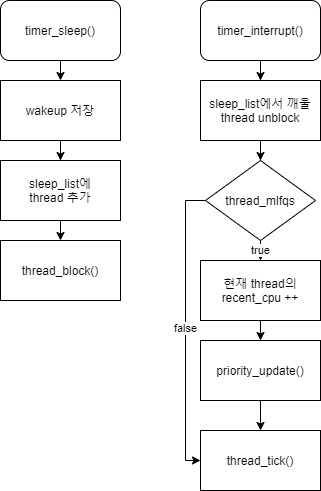
기존의 코드는 thread를 sleep시킬때 단순 while문만 사용한 busy waiting방식으로 구현되어있다. 이는 cpu 효율을 낮추므로, sema를 사용하여 thread를 block시킬 수 있도록 하였다. thread.h의 thread구조체에 int wakeup 변수를 추가하여 timer.c의 timer\_sleep()에서 사용할 수 있도록 하였다. 또한 sleep\_list를 선언하여 sleep 상태인 thread를 다룰 수 있도록 하였다. 그리고 time interrupt가 발생할때마다 이 리스트를 검색하여 깨울 thread가 있다면 sleep list에서 지우고 block 상태를 풀어주도록 하였다. 이는 void timer\_interrupt()에서 구현하였다.

* Priority Scheduling & Advanced Scheduler (BSD sheduler)

thread\_yield() 함수는 현재 thread를 ready queue에 추가하는 rescheduling 함수이다. 우선순위에 따라 list에 추가해야하므로 list\_insert\_ordered()함수를 사용하여 추가하고, priority 값을 비교하기위한 함수를 만들어 파라미터로 넘겨주어 사용한다. 또한 thread\_unblock()함수에서도 sleep 상태인 thread를 깨워 ready queue에 넣어주어야하므로, 위와 같은 방식을 사용하여 추가한다. block된 thread의 sema의 경우에도 priority를 부여하기 위해 synch.c의 sema\_up에 가장 높은 priority를 가진 thread가 unblock될 수 있도록 수정하였다.

또한 aging을 구현하고 BSD scheduling 방식을 사용하기 위해서 위에서 서술한 식을 사용해 priority를 계산해야한다. 이때 pintos는 floating point가 아닌 fixed point를 사용하여 연산을 처리하기때문에 이와 관련한 계산 함수를 compute\_i\_f()를 추가하여 구현하였다. thread 구조체에 recent\_cpu와 nice 변수를 추가하여 적절히 초기화하고, thread.c에 load\_avg 변수를 선언하였다. 이 외에도 ppt에 나와있는 bool thread\_prior\_aging 변수와 같이 aging을 위해 사용하는 계산 식과 관련한 함수들을 thread.c에 수정 및 추가하였다. aging flag 처리는 init.c의 parse\_option()함수를 수정하여 구현하였다.

1. **연구 결과**
   1. **Flow Chart**

****

* 1. **제작 내용**
* Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

|  |
| --- |
| timer.c |
| static struct list sleep\_list;  ...  void  timer\_sleep (int64\_t ticks)  {  int64\_t start = timer\_ticks ();  ASSERT (intr\_get\_level () == INTR\_ON);    struct thread \*t;  enum intr\_level old\_level = intr\_disable();  t = thread\_current();  t->wakeup = start + ticks;  list\_push\_back(&sleep\_list, &t->elem);  thread\_block();  intr\_set\_level(old\_level);  }  static void  timer\_interrupt (struct intr\_frame \*args UNUSED)  {  ticks++;  struct thread \*t;  struct list\_elem \*e;  for(e = list\_begin(&sleep\_list); e!=list\_end(&sleep\_list);){  t = list\_entry(e, struct thread, elem);  if(t->wakeup <= ticks){  e = list\_remove(e);  thread\_unblock(t);  }  else{  e = list\_next(e);  }  }  if(thread\_prior\_aging || thread\_mlfqs){  thread\_current()->recent\_cpu = compute\_i\_f(thread\_current()->recent\_cpu, 1, 3);  if(timer\_ticks() % TIMER\_FREQ == 0)  update\_load\_avg\_and\_recent\_cpu();  if(timer\_ticks() % 4 == 0)  update\_priority();  }  thread\_tick();  } |

기존의 busy waiting 방식이 아닌 sleep list를 이용하여 thread를 block했다. wakeup 변수를 thread 구조체에 추가하고 timer\_sleep()함수에서 현재 thread에 대한 wakeup을 계산하여 저장한 뒤 sleep\_list에 추가하고 thread\_block()을 사용하여 block시켰다. 그리고 time interrupt가 발생할때마다 깨워야 할 thread가 있는지 검사하기 위해 timer\_interrupt()함수를 호출할 때 sleep 중인 thread들을 for문을 wakeup 값을 체크한다.

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

|  |
| --- |
| thread.c |
| void  thread\_yield (void)  {  struct thread \*cur = thread\_current ();  enum intr\_level old\_level;    ASSERT (!intr\_context ());  old\_level = intr\_disable ();  if (cur != idle\_thread)  list\_insert\_ordered(&ready\_list, &cur->elem, priority\_less, NULL);  cur->status = THREAD\_READY;  schedule ();  intr\_set\_level (old\_level);  }  ...  tid\_t  thread\_create (const char \*name, int priority,  thread\_func \*function, void \*aux)  {  struct thread \*t;    ...  /\* Add to run queue. \*/  thread\_unblock (t);  /////////////////////  if(priority > thread\_get\_priority())  thread\_yield();  return tid;  }  ...  bool priority\_less(const struct list\_elem\* left, const struct list\_elem \*right, void \*aux){  struct thread \*t\_left = list\_entry(left, struct thread, elem);  struct thread \*t\_right = list\_entry(right, struct thread, elem);  if(t\_left->priority > t\_right->priority) return true;  else return false;  } |

새로운 thread가 들어와 스케쥴링을 해야할 때 thread\_create()가 호출된다. 이때 추가되는 thread의 우선순위가 현재 실행중인 thread의 우선순위보다 높을 경우 rescheduling하는 함수인 thread\_yield()를 호출하여 현재 thread를 ready\_list의 priority 순서대로 삽입한다. list\_insert\_ordered()함수를 사용하는데, priority의 대소비교를 위해 이전 프로젝트 0\_2에서 만든 것 처럼 bool 형태의 대소비교 함수를 만들어서 적용시켰다.

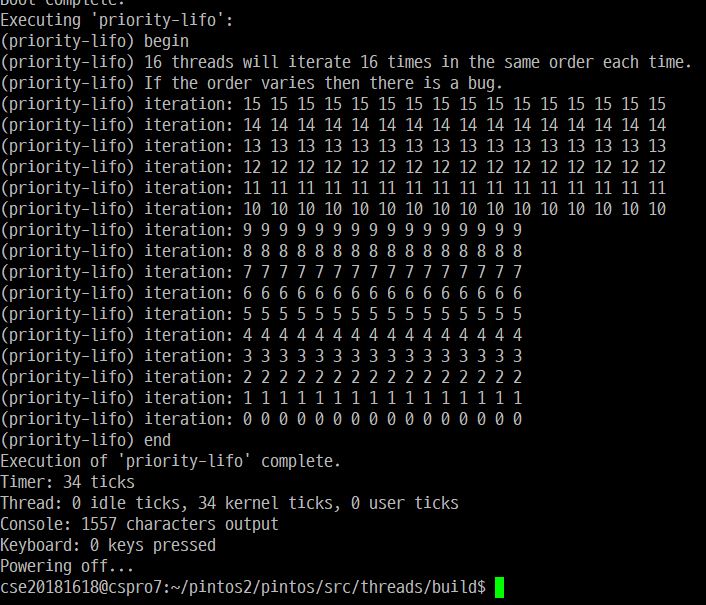
* Advanced Scheduler에서 priority 계산에 필요한 각 요소를 서술. (추가구현을 한 경우)

|  |
| --- |
| thread.c |
| void  thread\_set\_priority (int new\_priority)  {  int cur\_priority = thread\_current()->priority;  if(thread\_mlfqs) return;  thread\_current()->priority = new\_priority;  if(new\_priority < cur\_priority)  thread\_yield();  }  int  thread\_get\_priority (void)  {  return thread\_current ()->priority;  }  void  thread\_set\_nice (int nice UNUSED)  {  struct thread \*t = thread\_current();  t->nice = nice;  t->priority = compute\_i\_f(compute\_i\_f(compute\_i\_f(0, PRI\_MAX, 3), compute\_i\_f(t->recent\_cpu, 4, 4), 6), compute\_i\_f( 2, compute\_i\_f(0, t->nice, 3), 2), 6) / FRACTION;  if(t->priority > PRI\_MAX)  t->priority = PRI\_MAX;  if(t->priority < PRI\_MIN)  t->priority = PRI\_MIN;  if(t->priority < max\_priority())  thread\_yield();  }  int  thread\_get\_nice (void)  {  /\* Not yet implemented. \*/  return thread\_current()->nice;  }  int thread\_get\_load\_avg(void){  return compute\_i\_f(100, load\_avg, 2) / FRACTION;  }  int thread\_get\_recent\_cpu(void){  return compute\_i\_f(100, thread\_current()->recent\_cpu, 2) / FRACTION;  }  int compute\_i\_f(int a, int b, int key){  int ret = 0;  int64\_t t;  switch(key){  case 1 : // i - f  ret = a \* FRACTION - b;  break;  case 2 : // i \* f  ret = a \* b;  break;  case 3 : // f + i  ret = a + b \* FRACTION;  break;  case 4 : // f / i  ret = a / b;  break;  case 5 : //f + f  ret = a + b;  break;  case 6 : //f - f  ret = a - b;  break;  case 7 : // f \* f  t = a;  t = t \* b / FRACTION;  return (int)t;  break;  case 8 : // f / f  t = a;  t = t \* FRACTION / b;  return (int)t;  break;  default :;  }  return ret;  }  int max\_priority(){  int prior = -1;  struct thread \*t;  if(!list\_empty(&ready\_list)){  t = list\_entry(list\_front(&ready\_list), struct thread, elem);  prior = t->priority;  }  return prior;  }  void update\_load\_avg\_and\_recent\_cpu(void){  int ready\_threads = list\_size(&ready\_list);  struct thread \*t;  struct list\_elem \*e;  if(thread\_current() != idle\_thread)  ready\_threads++;  load\_avg = compute\_i\_f(compute\_i\_f(compute\_i\_f(59, load\_avg, 2), ready\_threads, 3), 60, 4);  for(e = list\_begin(&all\_list);e!= list\_end(&all\_list);e=list\_next(e)){  t = list\_entry(e, struct thread, allelem);  if(t != idle\_thread)  t->recent\_cpu = compute\_i\_f(compute\_i\_f(compute\_i\_f(compute\_i\_f(2, load\_avg, 2), compute\_i\_f(compute\_i\_f(2,load\_avg, 2), 1, 3), 8), t->recent\_cpu,7) ,t->nice, 3);    }  }  void update\_priority(){  struct thread \*t;  struct list\_elem \*e;  for(e = list\_begin(&all\_list);e != list\_end(&all\_list); e=list\_next(e)){  t = list\_entry(e, struct thread, allelem);  t->priority = compute\_i\_f(compute\_i\_f(compute\_i\_f(0, PRI\_MAX, 3), compute\_i\_f(t->recent\_cpu, 4, 4), 6), compute\_i\_f(2, compute\_i\_f(0, t->nice, 3), 2), 6)/FRACTION;  if(t->priority > PRI\_MAX)  t->priority = PRI\_MAX;  if(t->priority < PRI\_MIN)  t->priority = PRI\_MIN;  }  if(thread\_current()->priority < max\_priority())  intr\_yield\_on\_return();  } |

단일 queue를 사용하여 추가 구현을 진행하였으며, priority가 높은 순으로 thread가 수행되게 하기 위해 여러 부분을 수정 및 추가하였다. 그리고 aging 기법 및 BSD scheduling을 구현하기 위해, 위에서 서술한 load\_avg, recent\_cpu, nice에 대한 함수들을 수정하였고 계산 식을 이용하기 위해 fixed point 연산을 하는 compute\_i\_f() 함수를 추가하였다.

|  |
| --- |
| timer.c |
| static void  timer\_interrupt (struct intr\_frame \*args UNUSED)  {  ticks++;  struct thread \*t;  struct list\_elem \*e;  ...  if(thread\_prior\_aging || thread\_mlfqs){  thread\_current()->recent\_cpu = compute\_i\_f(thread\_current()->recent\_cpu, 1, 3);  ...  }  thread\_tick();  } |

aging 구현과 더불어 BSD scheduling에서, -mlfqs 옵션이 있을 때 thread\_mlfqs가 true가 되어 관련 함수들을 호출한다. load\_avg, recent\_cpu, priority는 1ticks마다 계산되고, 4ticks마다 priority가 다시 계산되어 update를 한다.

* 1. **시험 및 평가 내용**
* 
* priority\_lifo test는 THREAD\_CNT개의 동일한 자식 thread를 생성하는데, 이 자식 thread들은 실행되면 data->op에 INTE\_CNT번 data->id를 채워 넣는다. data->id는 thread가 실행된 순서와 같다. (처음 실행된 thread가 0, 마지막 thread가 THREAD\_CNT) 각 iteration마다 data->op를 수정할 때 data->lock를 걸었다가 푼다. 따라서 한 순간에 data->op를 수정할 수 있는 것은 하나의 thread이다. 모든 자식 thread의 수행이 끝나고 나면, data->op의 내용을 출력한다. 결과적으로는 data->op의 내용은 자식 thread가 실행된 순서이다. priority\_lifo에서는 priority가 PRI\_DEFAULT+1+i로 처음에는 비교적 낮은 priority의 thread가, 나중에 실행된 thread일수록 priority가 높은 thread가 생성된다. 따라서 가장 높은 priority의 마지막 thread가 연속해서 ITER\_CNT번 data->op에 쓰고, THREAD\_CNT-1번째 thread가 ITER\_CNT번 쓰고, ..., 0번 thread가 ITER\_CNT번 쓰게 되어야 맞게 수행된 것이다. 유사한 구조의 테스트인 priority\_fifo와 비교해보면, fifo 테스트는 생성되는 thread의 priority가 모두 같다. 따라서 rount-robin에 의해 thread들이 1번씩, iteration마다 같은 순서로 등장한다. 각 thread가 ITER\_CNT번씩, 높은 priority부터 출력되어, 올바르게 수행됨을 알 수 있다.

