**Pintos Project 3: Threads**

담당 교수 : 박성용 교수님

조 / 조원 : 20181703 황동준

개발 기간 : 2020-11-23 ~ 2020-12-06

1. **개발 목표**

이전에 구현하였던 userprog 부분을 완벽하게 구현 한 후, thread가 더욱 효율적으로 일을 처리하게 하기 위해서 thread의 Alarm-Clock과 Priority Schedule, BSD Scheduler를 구현할 수 있도록 한다.

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

Alarm clock을 구현하면, ticks에 따라서 조건에 맞으면 thread\_yield하게 되는데, 이 조건에 계속 만족을 해야 하는 비효율성이 생기고 이렇게 한다면 RUNNING State와 READY State 가 반복적으로 돌기 때문에 여기에 BLOCK State로 넘어가는 과정과 wakeup 하는 시간을 추가하여 thread 실행 과정에서의 연산을 조금 더 줄인다.

* 1. Priority Scheduling

Priority가 더 높은 thread가 먼저 실행되게 되며, 이를 통해서 빨리 실행이 되어야 효율적인 thread가 먼저 실행되고, System 자체에서 일을 처리하는 속도가 빨라질 것이다. Round-Robin 방법에서 Priority를 추가하여 우선순위가 높으면 먼저 그 thread가 실행될 것이다.

* 1. Advanced Scheduler (추가구현을 한 경우)

BSD Scheduler를 구현하면, Priority가 낮은 thread가 계속 실행이 안되면서 문제가 발생하는 것을 해결해 준다. 이를 개발하면 thread에서 우선순위가 낮은 것들도 모두 어느 정도의 새로운 우선순위를 가져 동등하게 실행된다.

* 1. **개발 내용**

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

Thread의 상태가 semphore나, timer에 의해 시간이 다 되어서 block에 들어갔다면, 그에 따른 wakeup 하는 시간을 thread마다 만들어주고 wakeup하는 시간이 되면 interrupt를 걸어주어 wakeup을 시키고 thread를 unblock 한다. 이때 상태는 block상태에서 ready 상태로 바꿔주게 되며, unblock을 실행할 때는 중간에 interrupt가 들어오지 않도록 interrupt를 disable 해주고 READY상태로 만든 후 다시 interrupt를 받을 수 있도록 set을 해준다.

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

만약 priority가 큰 thread가 들어온다면 (create) 그 즉시, 현재 실행되고 있는 thread와 비교를 하고 running thread를 다시 ready로 바꾸어 준 다음 다시 스케줄링을 하여 높은 priority가 running thread로 들어오게 된다.

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

Nice, Recent\_cpu, Load\_average 등의 변수가 필요한 데, nice는 각각 thread의 첫 priority를 정해주는 constant 값이고, recent cpu는 최근에 해당 thread가 cpu를 얼마나 잡아 먹었는지에 대한 지표이며, load\_average 변수는 최근 1분 동안 수행가능한 프로세스의 평균 개수이다. 이를 이용하여 priority를 update할 예정이다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**
   2. 2020-11-23 ~ 2020-11-24 Pintos Manual 확인 후 개발 범위와 구현 내용 이해
   3. 2020-11-25 ~ 2020-11-26 Alarm Clock 구현
   4. 2020-11-27 ~ 2020-11-30 Priority Scheduling 구현
   5. 2020-12-01 ~ 2020-12-02 Aging, BSD Scheduler 변수 구현
   6. 2020-12-03 ~ 2020-12-05 BSD Scheduler 구현
   7. **개발 방법**
      1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

이 부분은 Alarm Clock부분을 말하는 것인데, 우리는 timer\_sleep 과 timer\_interrupt 그리고 sleep list를 timer.c에 추가하여 block 된 thread들을 list에 넣어서 관리할 것이며, 당연히 우리는 thread의 자료구조에 wakeup의 시간을 재는 변수를 넣어서 각 thread마다 wakeup time이 존재할 수 있도록 만들 것이다. 추가해야 할 함수는 따로 없다.

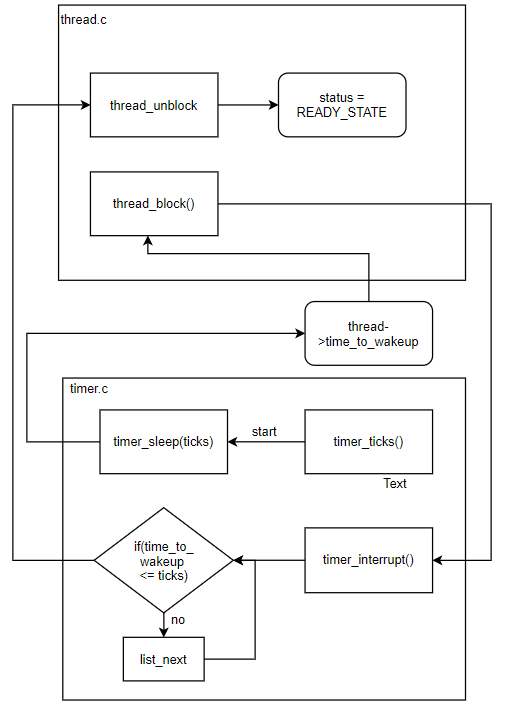
* + 1. Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 priority scheduling에 따르면 어떻게 해야 하는지 서술.

먼저 우리는 Priority Schedule을 구현하기 위해서 ready에 있는 thread들의 priority를 비교할 것이고, 이는 각각의 list들이 list의 라이브러리에 있는 함수를 통해 다시 sorting 될 것이다. 그래서 우리는 먼저 priority를 비교하는 함수를 만들 것이며, thread\_yield부분에서 원래는 list의 뒤에 thread를 넣었던 것을 list\_insert\_ordered를 실행하여, priority를 고려하여 list에 thread를 넣을 수 있도록 할 것이다.  
위에서 말한 높은 priority가 들어올 경우에는 thread\_create라는 함수에서 실행한 thread의 priority를 바로 running중인 thread의 priority와 비교하여 thread\_yield를 다시하여 재 스케줄링 할 수 있도록 한다. 위의 개발 내용을 구현하기 위해서는 thread\_create 부분과 thread\_yield 부분을 건드려야 할 것이다.

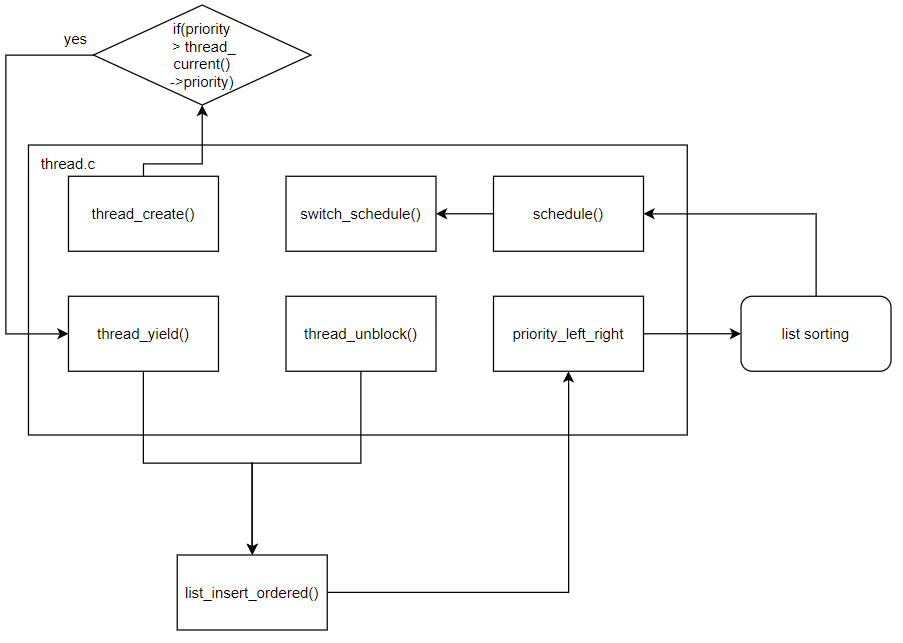
* + 1. Advanced Scheduler에서 priority 계산에 필요한 각 요소를 서술. (추가구현을 한 경우)

nice라는 변수를 추가하기 위해서는 기존에 있던 thread\_get\_nice, thread\_set\_nice함수를 구현해야 할 것이며, thread\_get\_recent\_value, thread\_get\_load\_avg함수 또한 구현해야 할 것이다. (return 값을 추가) 또한, priority나 recent\_cpu, load\_avg를 update하는 함수도 생성해야 할 것이고, 이를 계산하기 위해서는 실수를 계산하는 pintos 안에서의 유일한 방법인 fixed\_point를 이용해서 실수를 계산해야 한다. 당연히 추가할 자료구조는 thread에 nice, recent\_cpu, load\_avg를 각각 추가해야 할 것이다.

1. **연구 결과**
   1. **Flow Chart**
      1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.



* + 1. Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 priority scheduling에 따르면 어떻게 해야 하는지 서술.



* 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. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

|  |
| --- |
| void  timer\_sleep (int64\_t ticks) |
| void  timer\_sleep (int64\_t ticks)  {  struct thread \*cur = thread\_current();  int64\_t start = timer\_ticks ();  enum intr\_level old\_level;  ASSERT (intr\_get\_level () == INTR\_ON);  old\_level = intr\_disable();  cur->time\_to\_wakeup = start + ticks;  //if(cur != idle\_thread)  list\_push\_back(&sleep\_list, &cur->elem);  thread\_block();  intr\_set\_level(old\_level);  //while (timer\_elapsed (start) < ticks)  // thread\_yield ();  } |
| 먼저 기존의 thread\_yield를 지우고 sleep list를 만들어 ticks를 통해 sleep에 들어간다면 바로 ready state로 가는게 아니라 block을 할 수 있도록 구현하였다. 만약 시간이 다 된다면 interrupt 함수에서 unblock을 해주는데, 이는 timer\_interrupt에서 구현되었다. List는 아래와 같이 init 되고 선언되었다. |

|  |
| --- |
| /\* user \*/  static struct list sleep\_list;  void  timer\_init (void)  ..  list\_init(&sleep\_list); |
| 다음과 같이 block list를 선언하고 이를 초기화 하였다. |

|  |
| --- |
| static void  timer\_interrupt (struct intr\_frame \*args UNUSED) |
| static void  timer\_interrupt (struct intr\_frame \*args UNUSED)  {  struct thread \*t;  struct list\_elem \*e;  ticks++;  for(e=list\_begin(&sleep\_list); e != list\_end(&sleep\_list);){  t=list\_entry(e, struct thread, elem);  if(t->time\_to\_wakeup <= ticks){  e = list\_remove(e);  thread\_unblock(t);  }  else e = list\_next(e);  }  if (thread\_prior\_aging || thread\_mlfqs){  thread\_current()->recent\_cpu = add\_float\_int(thread\_current()->recent\_cpu, 1);  if(timer\_ticks() % TIMER\_FREQ == 0)  change\_load\_avg\_recent\_cpu();  if(timer\_ticks() % 4 == 0)  change\_priority();  }  thread\_tick ();  } |
| thread\_prior\_aging과 나머지 부분은 아래의 priority와 BSD를 구현할 때 추가 설명할 것이다. 이 함수에서는 ticks를 매번 높여주고, (interrupt가 올때마다) 이를 높일 때마다 시간상 wakeup 시간이 된 thread를 block list에서 찾아내서 thread를 unblock 시켜준 후 READY STATE로 thread의 상태를 바꾸고 priority list에 thread를 추가하게 된다. |

구현 중 문법오류를 제외하고는 추가적인 에러가 발생하지 않았다.

* + 1. Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 priority scheduling에 따르면 어떻게 해야 하는지 서술.

|  |
| --- |
| thread.c |
| bool priority\_left\_right(const struct list\_elem \*left, const struct list\_elem \*right, void \*aux){  struct thread \*tl = list\_entry(left, struct thread, elem);  struct thread \*tr = list\_entry(right, struct thread, elem);  return tl->priority > tr->priority;  }  int max\_prior(void){  int priority = -1;  struct thread \*t;  if(!list\_empty(&ready\_list)){  t = list\_entry(list\_front(&ready\_list), struct thread, elem);  priority = t->priority;  }  return priority;  } |
| 해당 부분은 list\_insert\_ordered에 넣을 함수로 priority\_left\_right는 각각 들어온 두개의 list element들을 받고 둘 중 어느 thread가 priority 가 큰지 비교하고 list에 priority 순서대로 삽입되게 된다. Max\_prior은 priority가 큰 순서대로 정렬된 list의 가장 앞 thread를 뽑아내면서, 가장 큰 priority를 return 하게 되는 함수이다. 이 함수는 만약 ready list에서 가장 큰 priority 를 가진 thread가 지금 실행중인 thread보다 priority가 높을 경우 thread rescheduling이 이루어져야 하기 때문에, 구현한 함수이다. 이 함수들은 새로 구현한 함수들이다. |

|  |
| --- |
| thread.c |
| void  thread\_unblock (struct thread \*t)  &  void  thread\_yield (void)  …  list\_insert\_ordered(&ready\_list, &cur->elem, priority\_left\_right, NULL);  tid\_t  thread\_create (const char \*name, int priority,  thread\_func \*function, void \*aux)  …  if(priority > thread\_get\_priority()) thread\_yield();  void  thread\_set\_priority (int new\_priority)  …  if(new\_priority < priority\_current) thread\_yield(); |
| thread\_unblock과 thread\_yield에서 thread가 block이 풀려서 ready state로 들어가거나 바로 thread가 yield되서 rescheduling에 들어갈 경우 list\_insert\_ordered함수에 위에서 구현한 priority\_left\_right 함수를 이용하여 priority에 따라 정렬을 한 후, 가장 높은 priority를 가진 thread가 switch 된다.  또한 thread\_create부분에서는 위에서 말한 만약 새로운 priority가 높은 thread가 생성되었을 때의 경우를 처리한 부분이다. Thread\_create가 되자마자 priority를 비교하여 만약에 높다면 thread\_yield를 통해 재 스케줄링 되도록 함수를 작성했다.  thread\_set\_priority에서는 만약 running thread보다 priority가 높은 thread가 들어왔다면 rescheduling이 일어날 수 있도록 만든 함수부분이다. |

새로 구현한 함수는 위에서 적어 놓았다.

에러는 priority\_sema에서 발견되었다.

|  |
| --- |
| synch.c |
| void  sema\_up (struct semaphore \*sema)  {  enum intr\_level old\_level;  struct thread \*t, \*pt; // max priority thread  struct list\_elem \*e, \*pe; // max priority element  ASSERT (sema != NULL);  old\_level = intr\_disable ();  if (!list\_empty (&sema->waiters)) {  e = list\_begin(&sema->waiters);  pe = list\_begin(&sema->waiters);  pt = list\_entry(e, struct thread, elem);  for(e=list\_next(e); e!=list\_end(&sema->waiters); e = list\_next(e)){  t = list\_entry(e, struct thread, elem);  if(t->priority > pt->priority){  pt = t;  pe = e;  }  }  list\_remove(pe);  thread\_unblock(pt);  }//find max priority if semaphore up! If semaphore action not consider priority, the priority of thread is mixed.  sema->value++;  intr\_set\_level (old\_level);  thread\_yield(); // always scheduling  } |
| 주석에서도 설명했지만, priority\_sema 부분이 해결되지 않아서 sema\_up 부분에 sema->waiter을 확인하고 여기에서 또한 priority를 따라서 가장 큰 priority를 지정하여, 해당 thread는 block을 해제하는 방식으로 진행하였다. sema에서 이를 처리해 주지 않아서 만약 thread에서 semaphore을 up 했을 때 priority가 꼬여서 running thread와 ready list간의 priority가 꼬여버리는 현상이 발생했다. 따라서 이러한 조치를 취하도록 코드를 작성하였다. |

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

|  |
| --- |
| thread.c thread.h (fixed point 구현) |
| /\* user manage fixed pointer (aging) \*/  #define FRACTION (1<<14)  #define sub\_int\_float(x,y) ((x)\*FRACTION - (y))  #define mul\_int\_float(x,y) ((x)\*(y))  #define add\_float\_int(x,y) ((x)+(y)\*FRACTION)  #define add\_float\_float(x,y) ((x)+(y))  #define sub\_float\_float(x,y) ((x)-(y))  #define div\_float\_int(x,y) ((x)/(y))  int mul\_float\_float(int x, int y){  int64\_t temp = x;  temp = temp \* y / FRACTION;  return (int)temp;  }  int div\_float\_float(int x, int y){  int64\_t temp = x;  temp = temp \* FRACTION / y;  return (int)temp;  } |
| FRACTION을 위와 같이 정의한 이유는 floating point가 소수점 아래로 2의 14승 까지 범위를 포함하기 때문이다. 이를 기반으로 정수와 실수사이의 계산을 구현하였고 이는 형의 변환의 불완전성을 극복하기 위해서 만든 함수를 제외하고는 모두 매크로로 정의하였다. 이 함수와 매크로를 이용해서 아래의 nice, recent\_cpu, load\_avg를 구현하여 priority aging을 할 수 있도록 하였다. |

|  |
| --- |
| thread.c |
| void change\_load\_avg\_recent\_cpu(void){  int ready\_threads = list\_size(&ready\_list);  struct thread \*t;  struct list\_elem \*e;  if(thread\_current() != idle\_thread) ready\_threads += 1;  load\_avg = div\_float\_int(add\_float\_int(mul\_int\_float(59, load\_avg), ready\_threads),60);  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 = add\_float\_int(mul\_float\_float(div\_float\_float(mul\_int\_float(2, load\_avg), add\_float\_int(mul\_int\_float(2, load\_avg), 1)), t->recent\_cpu), t->nice);  }  }  }  void change\_priority(void){  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 = sub\_float\_float(sub\_float\_float(add\_float\_int(0,PRI\_MAX), div\_float\_int(t->recent\_cpu, 4)), mul\_int\_float(2, add\_float\_int(0, t->nice))) / FRACTION;    if(t->priority > PRI\_MAX)  t->priority = PRI\_MAX;    if (t->priority < PRI\_MIN)  t->priority = PRI\_MIN;  }  if(thread\_current()->priority < max\_prior())  intr\_yield\_on\_return();  }  void  thread\_set\_nice (int nice)  {  /\* Not yet implemented. \*/  struct thread \*t = thread\_current();  t->nice = nice;  t->priority = sub\_float\_float(sub\_float\_float(add\_float\_int(0,PRI\_MAX), div\_float\_int(t->recent\_cpu, 4)), mul\_int\_float(2, add\_float\_int(0, t->nice))) / FRACTION;  if(t->priority > PRI\_MAX) t->priority = PRI\_MAX;  if(t->priority < PRI\_MIN) t->priority = PRI\_MIN;  if(t->priority < max\_prior()) thread\_yield();  } |
| 여러가지 변수들은 다음과 같은 pintos manual에 나온 방법대로 구현을 완료하였다.   * + *recent\_cpu* = (2 \* load\_avg) / (2 \* load\_avg + 1 ) \* *recent\_cpu* + nice   + *load\_avg* : (59/60) \* *load\_avg* + (1/60) \* *ready\_threads*     - *priority* = PRI\_MAX – (*recent\_cpu* / 4) - (*nice* \* 2)     - *nice:*  nice value of the thread   이를 통해서 priority와 nice, load\_avg와 recent\_cpu를 update할 수 있다. Thread\_set\_nice 같은 경우 pintos 프로그램에서 일부 쓰이는 함수(처음에 thread를 만들 때 constant값인 nice를 넣어주게 된다.)라 자동으로 업데이트가 되지만 다른 함수들 같은 경우 직접 만들어 준 함수이므로, timer\_interrupt에 추가하여 ticks가 하나 증가할 때마다 thread\_prior\_aging(ppt에 추가하라고 나온 함수)이 true이면 aging을 시도하도록 만들고, 이때 위의 변수들을 모두 update해주도록 한다. 이 부분은 아래에서 설명할 것이다.  따라서 priority와 recent\_cpu, load\_avg는 interrupt가 걸릴때마다 호출되는 것이다. |

|  |
| --- |
| timer.c |
| static void  timer\_interrupt (struct intr\_frame \*args UNUSED)  {  …  if (thread\_prior\_aging || thread\_mlfqs){  thread\_current()->recent\_cpu = add\_float\_int(thread\_current()->recent\_cpu, 1);  if(timer\_ticks() % TIMER\_FREQ == 0)  change\_load\_avg\_recent\_cpu();  if(timer\_ticks() % 4 == 0)  change\_priority();  }  thread\_tick ();  } |
| 위에서 말했던 것처럼 interrupt가 걸릴 때마다 thread\_prior\_aging이 true라면 우선순위와 나머지 변수들이 모두 update되도록 만들어야 한다. 하지만 interrupt가 걸릴 때마다 update를 하는건 연산이 너무 많이 소모 되므로, 우리는 pintos manual에 나온 방식대로 timer\_ticks () % TIMER\_FREQ == 0 일 때 load\_avg와 recent\_cpu를 update 하였고, priority는 4 ticks에 한 번씩 update 하도록 하였다. |

|  |
| --- |
| thread.c |
| int  thread\_get\_load\_avg (void)  {  /\* Not yet implemented. \*/  return mul\_int\_float(100, load\_avg) / FRACTION;  }  /\* Returns 100 times the current thread's recent\_cpu value. \*/  int  thread\_get\_recent\_cpu (void)  {  /\* Not yet implemented. \*/  return mul\_int\_float(100, thread\_current()->recent\_cpu) / FRACTION;  } |
| 이 부분은 ppt에 나온 부분으로 설명한 방식 대로 구현하였다. 이는 test 파일에서 실수 부분으로 계산을 처리하기 위해 만들어 진 것으로 설명된다. |

새로 구현한 함수와 이용한 라이브러리에 대한 내용은 위에서 설명했으며, 이 또한 사소한 에러를 제외하고는 큰 실수를 하지않고 프로그램을 작성하였다.

* 1. **시험 및 평가 내용**

|  |
| --- |
| priority-lifo.c |
| void  test\_priority\_lifo (void) {  thread\_set\_priority (PRI\_DEFAULT + THREAD\_CNT + 1);  for (i = 0; i < THREAD\_CNT; i++)  {  char name[16];  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);  }  cnt = 0;  for (; output < op; output++)  {  struct simple\_thread\_data \*d;  ASSERT (\*output >= 0 && \*output < THREAD\_CNT);  d = data + \*output;  if (cnt % THREAD\_CNT == 0)  printf ("(priority-lifo) iteration:");  printf (" %d", d->id);  if (++cnt % THREAD\_CNT == 0)  printf ("\n");  d->iterations++;  }  }  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 ();  }  } |
| 먼저 위 코드를 보면 일정한 수의 자식 thread를 만든다는 것을 알 수 있다. 먼저 부모 thread를 만들고 simple\_thread\_func를 통해 자식 thread를 만들게 된다. 이때 lock을 걸고 data->op에 id를 저장하여 child를 ITER\_CNT 만큼 만들어 주게 된다. 위에서 define 한 것을 보면, ITER\_CNT, THREAD\_CNT 모두 16 개이므로 총 16 \* 16 개의 자식 thread가 만들어 졌다는 것을 알 수 있다.  그리고 thread\_create을 보면 PRI\_DEFAULT + 1 + i라는 것을 알 수 있는데 이 말인 즉슨 i가 커질수록 해당 thread의 자식들은 모두 priority가 커진다는 것을 알 수 있었다. I는 증가하는 순서로 for문이 돌게 되므로 늦게 생성한 thread가 더 priority가 높다는 것을 알 수 있다.  다음 ouput과 관련된 for문이 돌 때는 priority가 높게 잡힌 thread가 먼저 실행되어야 하기 때문에 thread의 id가 가장 높은 15부터 thread가 16개 실행 될 것이고, 이 thread가 모두 실행되면 다음인 14 도 16개, 쭉 나아가서 0번째 thread까지 출력이 되되 될 것이다. 따라서 이러한 코드를 돌리고 나서 실행결과는 다음과 같아야 한다. |

|  |
| --- |
| priority-lifo 실행결과 |
|  |

Make check 수행결과 아래 첨부

|  |
| --- |
| make check |
|  |