**Pintos Project 3: Threads**

1. **개발 목표**

기존 Pintos에서는 Round-Robin 방식으로 각 thread를 스케줄링한다. 이러한 스케줄링 알고리즘을 수정하여 Pintos가 우선순위에 따라 thread를 스케줄링하고 실행하도록 하는 것이 목표이다. 또한 Busy waiting방식으로 구현된 thread\_sleep의 비효율성을 해결하고 프로세스의 starvation문제가 발생하는 것을 방지하기 위해 우선순위에 aging을 구현하는 것이 목표이다. 추가적으로 각 thread의 특성을 반영하여 우선순위를 업데이트하는 BSD스케줄러를 구현한다.

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

Pintos의 thread\_sleep()은 일정시간 동안 thread의 실행을 중지시킬 때 사용된다. 기존에 Pintos에 구현되어 있는 thread\_sleep()은 thread를 READY와 RUNNING state를 번갈아 가지는 busy waiting방식으로 구현되어 있다. 실행이 중지되어야 할 thread가 스케줄러에 의해 선택된다면 이를 ready queue가장 뒤에 삽입한다. 이러한 busy waiting방식은 CPU의 utilization을 방해한다. 이러한 비효율성을 해결하기 위해 thread\_sleep()가 호출된 thread는 일정 시간동안 다른 queue에 삽입하여 스케줄링되지 않도록 한다. 또한 sleep하는 thread가 일정시간이 지난 후에 다시 ready queue에 삽입될 수 있도록 한다. 이러한 Alarm Clock의 구현은 sleep하고 있는 thread를 따로 리스트로 관리하여 이들이 스케줄링 되지 않도록 하여 기존의 busy waiting방식의 비효율성을 해결한다.

* 1. Priority Scheduling

기존에 구현되어 있는 Pintos의 스케줄러는 Round-Robin방식으로 thread을 선택하고 실행한다. 이러한 구현은 thread의 우선순위를 무시하고 FIFO방식으로 thread을 ready queue에 넣고 일정시간마다 차례대로 실행시킨다. 각 thread의 우선순위를 고려하는 스케줄러를 구현하기 위해 각 thread에 priority를 부여하고 우선순위가 높은 thread가 먼저 실행되도록 한다. 이러한 Priority Scheduler에서는 우선순위가 낮은 thread기 오랫동안 실행되지 않는 기아 현상(starvation)이 발생할 수 있다. starvation문제를 해결하기 위해 READY상태의 thread의 우선순위를 시간이 지남에 따라 올려주는 aging기능을 구현한다. 이러한 우선순위 스케줄러를 구현하면 각 thread의 우선순위에 따라서 선택되고 실행될 수 있다.

* 1. Advanced Scheduler

앞서 설명한 우선순위 스케줄러의 aging방식은 단지 시간이 지남에 따라 우선순위를 높여준다. 하지만 I/O를 담당하는 thread는 짧은 cpu time을, computing을 위주로 담당하는 thread는 긴 cpu time을 요구하는 등 각기 다른 특성을 가진다. 이렇게 BSD 스케줄러는 시간 이외에도 CPU time이나 nice value등 thread의 특성을 고려하여 우선순위를 업데이트 하여 aging보다 더 효율적으로 thread를 스케줄링 해준다. 다양한 요소들을 고려하여 thread의 우선순위를 계산하기 때문에 복잡한 실수 연산이 요구된다. 기존의 Pintos에서는 floating-point의 실수 연산을 지원하지 않기 때문에 fixed-point실수 연산을 구현하여 BSD스케줄러를 구현한다. 이러한 BSD 스케줄러의 구현은 보다 더 체계적으로 thread들의 우선순위를 변경하여 CPU에서 효율적으로 thread를 실행할 수 있도록 해준다.

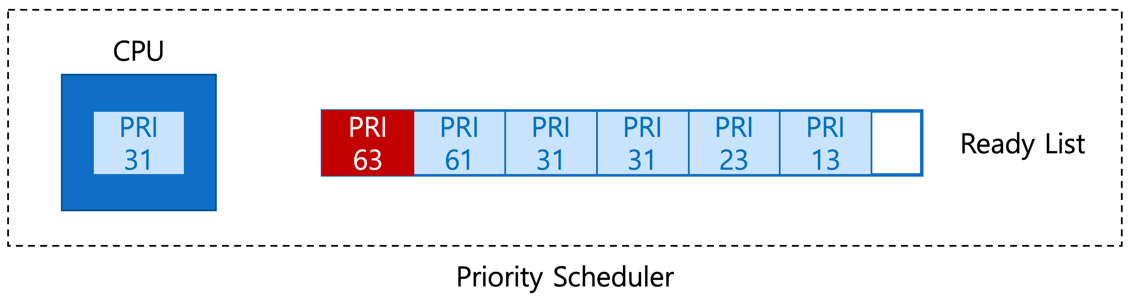
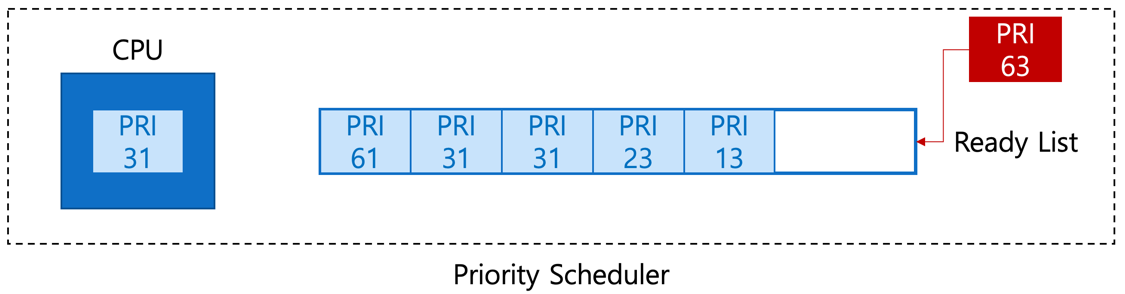
* 1. **개발 내용**

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

thread\_sleep()이 호출되었을 때 thread는 일정시간동안 실행이 되지 않아야 하기 때문에 이를 BLOCKED상태로 변경하고 독립된 blocked queue에서 관리한다. 하지만 일정시간 후에는 BLOCKED된 thread가 깨어나서 다시 READY 상태로 변경되고 ready queue에 삽입되어야 한다. 이를 위해서 각 thread가 자신이 깨어나야 할 시간을 저장할 수 있도록 thread 구조체에 변수를 추가한 뒤 thread\_sleep()이 호출되었을 때 이 시간을 저장한다. 이후 매 tick마다 호출되는 timer\_interrupt()에서 blocked queue에 저장된 thread들이 깨어나야 할 시간과 현재 시간을 비교한다. 만약 blocked queue의 thread 중 깨어나야 할 thread가 있다면 이를 blocked. queue에서 제거하고 READY상태로 변경 한 후에 ready queue에 삽입한다.

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

현재 RUNNING상태인 thread보다 높은 우선순위를 가진 thread가 ready queue로 들어올 경우에는 현재 RUNNING하는 thread의 실행을 멈춘 후에 ready queue를 다시 스케줄링해 새롭게 선택된 thread를 실행해야 한다. 이를 위해서 먼저 thread가 생성되고 ready queue에 삽입될 때 running thread의 우선순위와 이 thread의 우선순위를 비교한다. 새로운 thread의 우선순위가 높을 경우에는 thread\_yield()를 호출하여 running thread의 실행을 멈춘 후에 이를 다시 ready queue에 삽입한다. 이후 이를 다시 스케줄링해서 가장 높은 우선순위의 thread가 선택되도록 한다. 이런 경우에는 running thread가 기존에 가장 높은 우선순위를 가지는 thread이기 때문에 이보다 우선순위가 높은 thread가 들어왔으니 새 thread가 선택되어 실행될 것이다.



1. Advanced Scheduler에서 priority 계산에 필요한 각 요소를 서술.

각 thread는 정수 값인 nice value를 가진다. 이는 해당 thread가 다른 thread에게 얼마나 “친절한지”, 즉 CPU time을 얼마나 양보해주는 지에 대한 값을 가진다. nice value의 범위는 -20~20으로 양수의 nice value는 우선순위를 낮추어 다른 thread에게 CPU time을 양보한다. 반면 음수의 nice value는 우선순위를 높여 다른 thread들도 부터 CPU time을 가져온다. nice value가 0일 때는 우선순위에 아무런 영향을 미치지 않는다. 처음 생성된 thread는 0의 nice value를 가지고 이외의 thread는 부모 thread의 nice value를 상속받는다.

recent\_cpu값 또한 각 thread가 가지는 값으로 해당 thread가 최근에 할당받은 CPU time을 계산한다. 이를 계산할 때는 최근에 사용한 CPU time이 그 전에 사용한 CPU time보다 더 크게 반영된다. recent\_cpu를 계산하는 식은 아래와 같다.

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

인터럽트가 일어날 때마다 RUNNING thread의 recent\_cpu 값은 1씩 증가한다. 또한 매 초마다(TIMER\_FREQ) 모든 thread의 recent\_cpu값은 위 식을 토대로 새로 계산된다. load\_avg는 ready queue에서 CPU time을 할당받기 위해 경쟁하는 thread의 개수이다. load\_avg는 시스템에서 하나의 값을 가지고 있고 매 초마다 아래의 식에 따라서 계산된다. ready\_threads는 현재 READY 또는 RUNNING 상태인 thread의 개수이다.

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

BSD 스케줄러의 우선순위는 0(PRI\_MIN)부터 63(PRI\_MAX)까지 64개의 단계를 가진다. 모든 thread는 생성될 때 초기의 우선순위 값을 배정받는다. 이후 우선순위 값은 아래 식에 따라 새로 계산된다.

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

4 tick마다 우선순위는 위 값에 따라 새로 계산된다. 이러한 우선순위 값은 0~63사이의 값이 되도록 조정된다. BSD 스케줄러의 우선순위는 최근에 CPU time을 할당받은 thread에게는 낮은 우선순위를, 최근에 CPU time을 받지 못한 thread에게는 높은 우선순위를 준다. 이러한 방식으로 기아 현상을 해결하고 각 thread의 특성을 고려하여 이들을 스케줄링한다.

BSD 스케줄러에서 사용되는 recent\_cpu값과 load\_avg값은 실수이다. 하지만 Pintos에서는 floating-point 연산을 지원하지 않기 때문에 fixed-point연산을 구현하여 사용해야 한다. fixed-point format은 실수를 32 비트로 나타내는 것으로 첫번째 비트는 sign, 그 다음 17개의 비트는 integer값, 그 다음 14개 비트는 소수점 아래 값을 나타낸다. 위의 식을 계산하기 위해서 필요한 연산은 다음과 같다.

|  |  |  |  |
| --- | --- | --- | --- |
| 실수 + 실수  실수 + 정수 | 실수 - 실수  정수 - 실수 | 정수 \* 실수  실수 \* 실수 | 실수 / 실수  실수 / 정수 |

총 8가지의 연산에 대해서 fixed-point연산을 구현하여 우선순위를 계산할 수 있다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

|  |  |
| --- | --- |
| 날짜 | 수행 내용 |
| 11.14 ~ 11.20 | 이론 이해 및 프로젝트 계획 |
| 11.20 ~ 11.30 | Alarm Clock 및 Priority Scheduler 구현 |
| 11.30 ~ 12.08 | Aging, BSD Scheduler 구현, Documentation |

* 1. **개발 방법**
  + 수정해야하는 소스코드

- thread.h thread 구조체에 변수 추가 및 전역 변수 추가 및 추가된 함수의 prototype 작성

- devices/timer.c timer\_init(), timer\_sleep(), timer\_interrupt() 함수 수정 및 전역변수 추가하여 Alarm Clock 구현

- thread.c thread\_yield(), thread\_unblock(), thread\_create() 함수 수정 및 전역변수 추가해 우선순위 스케줄러 구현, 함수 추가 및 전역변수 추가해 BSD 스케줄러 구현

- threads/synch.c sema\_up() 함수 수정하여 우선순위 스케줄러 구현

- threads/ init.c aging 옵션 추가

* + 수정하거나 추가해야 하는 자료구조

threads/thread.h의 thread 구조체에 wake\_up\_time, recent\_cpu, nice 추가

threads/thread.c의 전역 변수로 load\_avg, thread\_prior\_aging 추가

devices/timer.cd의 전역 변수로 blocked\_threads list 추가

* + 수정하거나 추가해야 하는 함수

- devices/timer.c의 timer\_init(), timer\_sleep(), timer\_interrupt() 수정

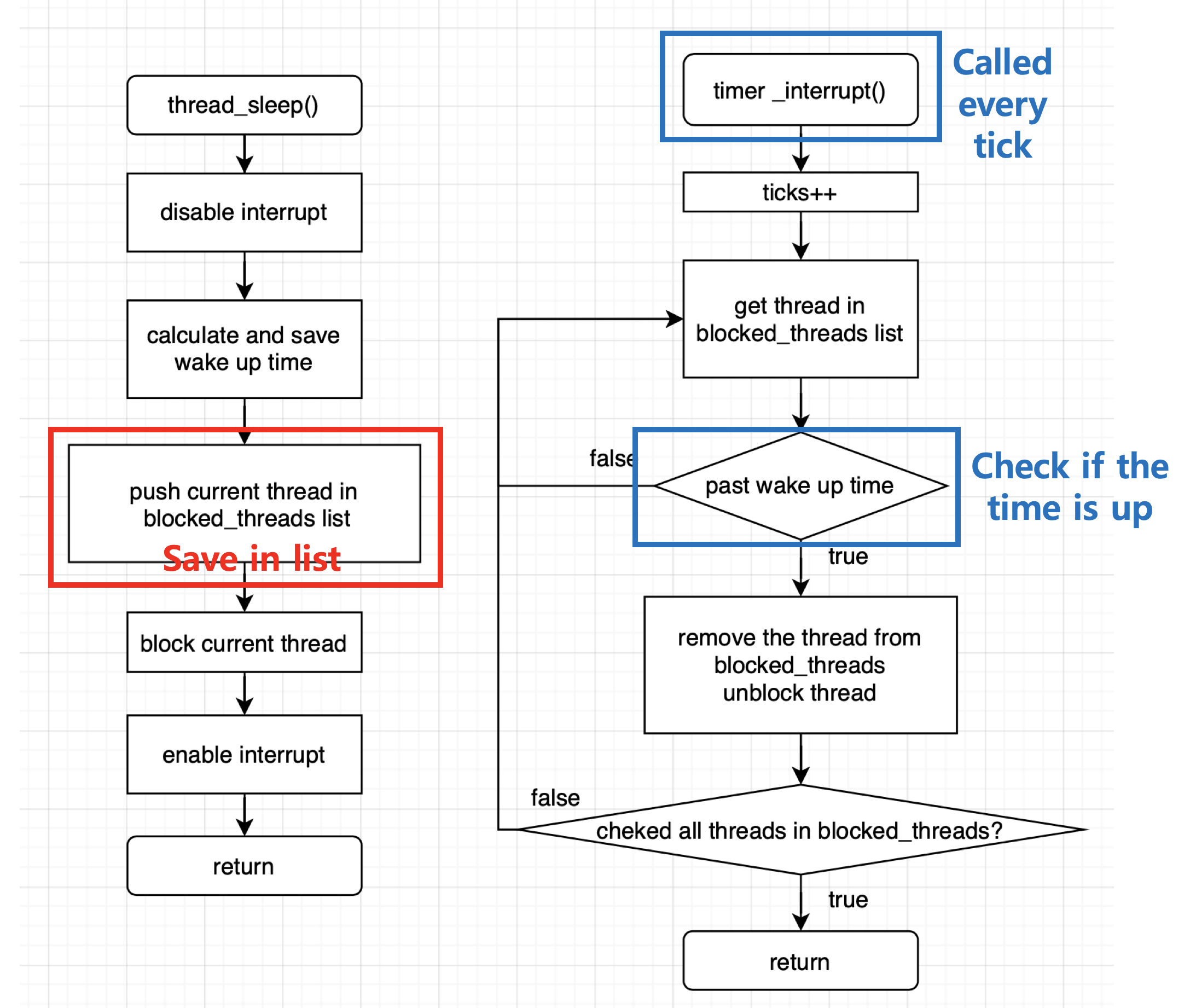
- threads/synch.c의 sema\_up() 수정

- thread.c의 thread\_yield(), thread\_unblock(), thread\_create(), thread\_tick() 수정

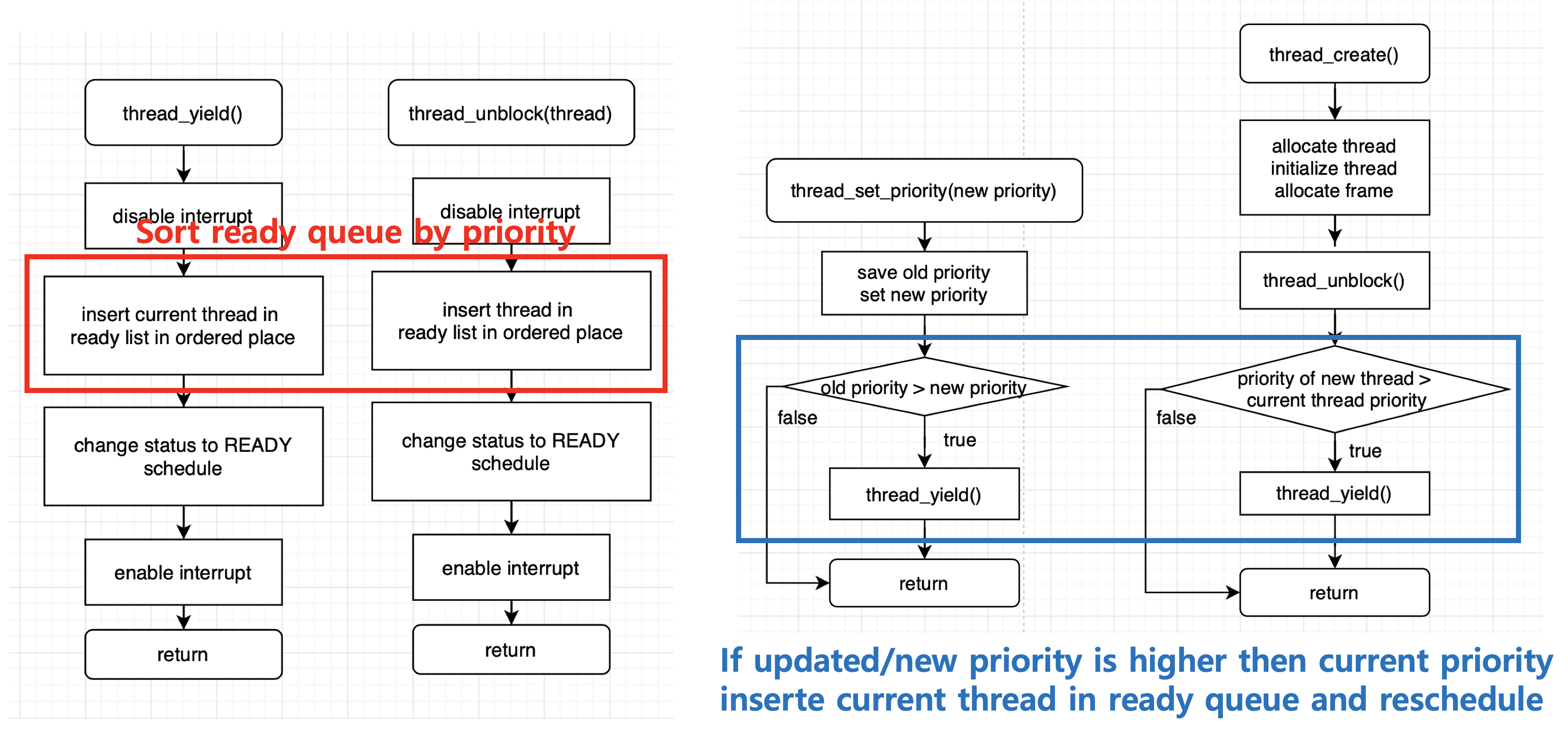
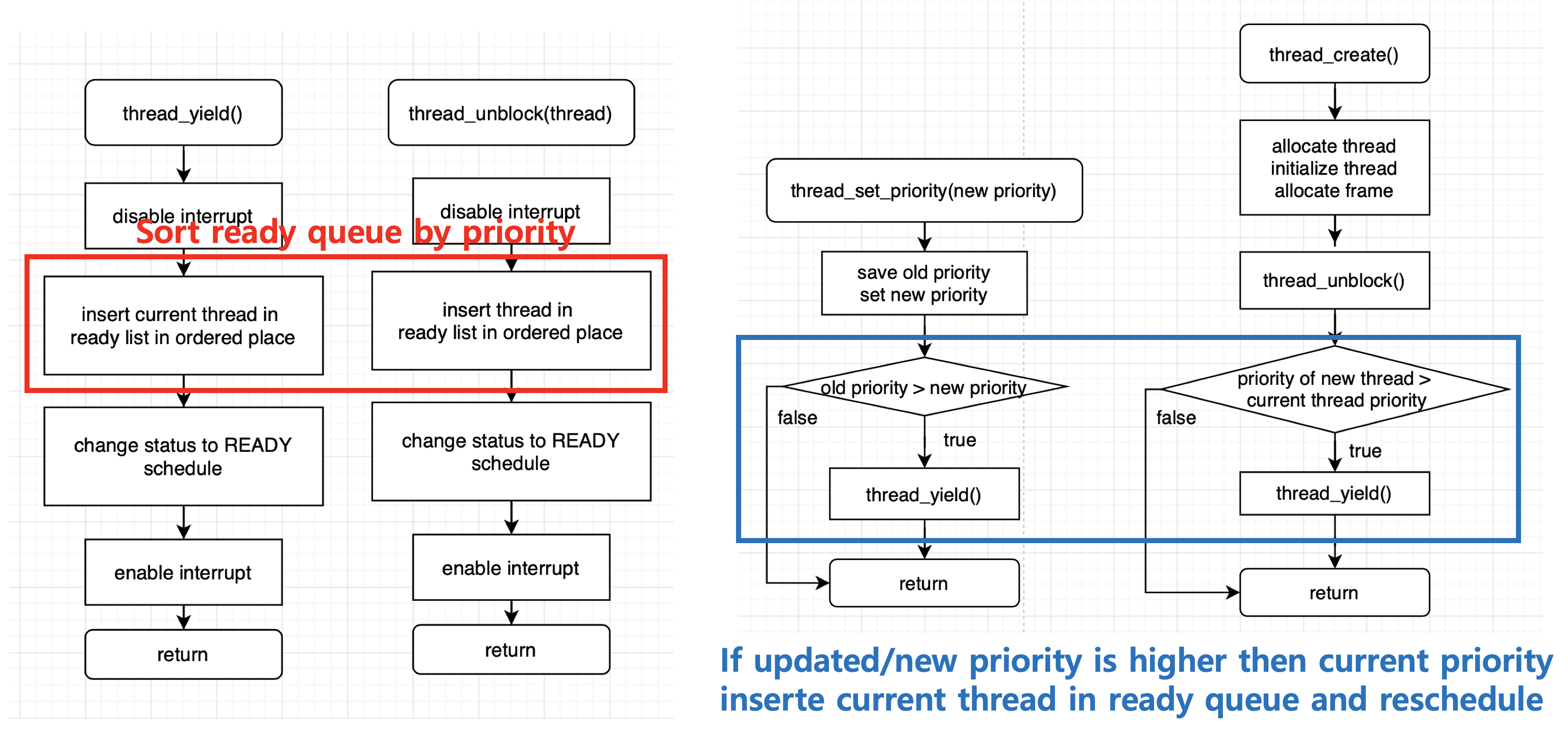
-threads/thread.c의 compare\_pri() thread\_set\_priority(), thread\_get\_priority(), thread\_set\_nice(), thread\_get\_nice(), thread\_get\_load\_avg(), thread\_set\_load\_avg(), thread\_get\_recent\_cpu(), thread\_set\_recent\_cpu(), thread\_aging(), f\_plus\_i(), f\_plus\_f(), i\_minus\_f(), f\_minus\_f(), i\_times\_f(), f\_times\_f(), f\_divide\_i(), f\_divide\_f() 추가

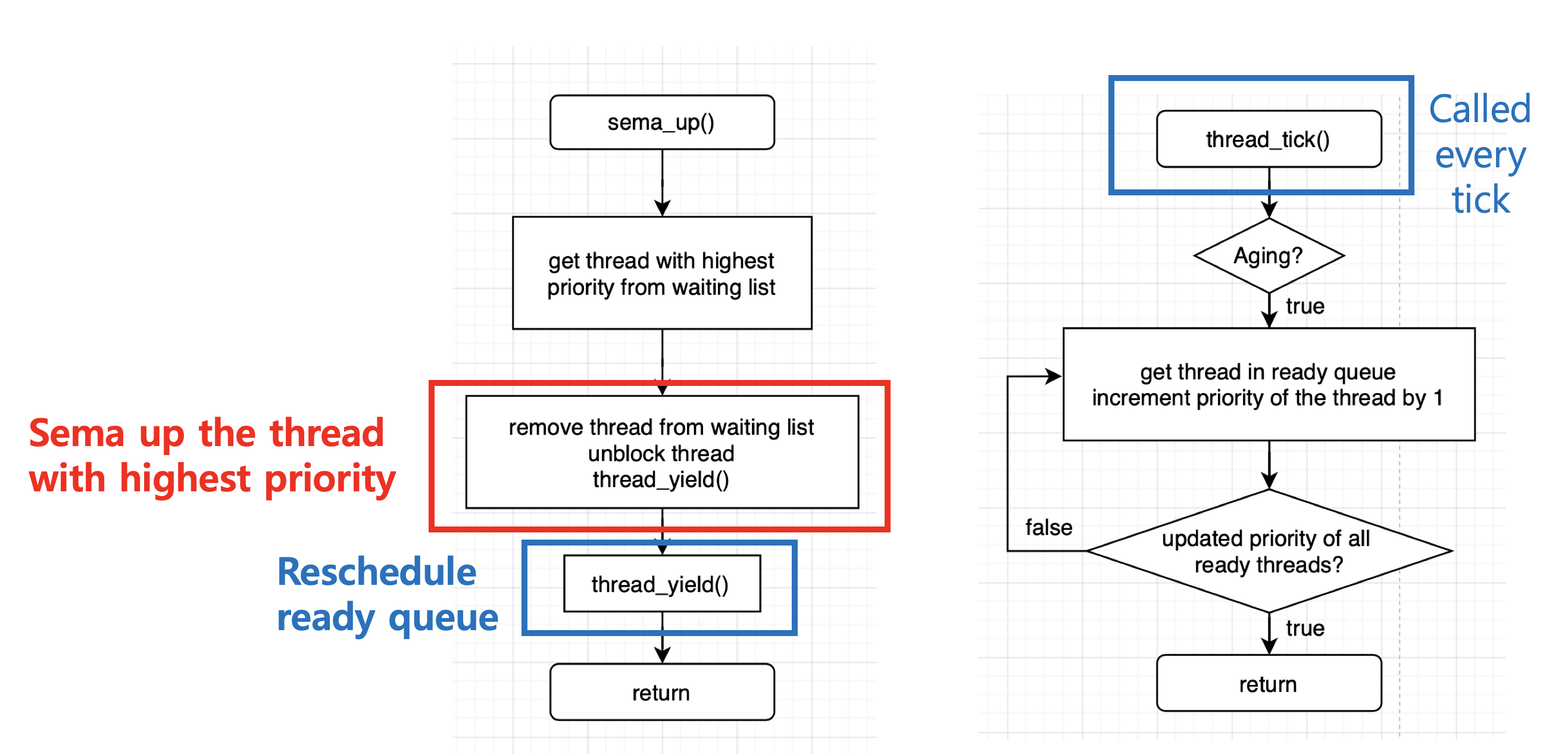
1. **연구 결과**
   1. **Flow Chart**

1. Alarm Clock



2. Priority Scheduler & Aging





* 1. **제작 내용**

1. Alarm Clock

|  |  |
| --- | --- |
| threads/thread.h |  |
| struct thread {  …  int64\_t wake\_up\_time  …  } | -각 thread가 BLOCKED되었을 때 깨어날 시간을 정할 수 있도록 thread구조체에 wake\_up\_time 추가 |
| devices/timer.c |  |
| static struct list blocked\_threads;  void timer\_init (void) {  …  list\_init(&blocked\_threads);}  void timer\_sleep (int64\_t ticks) {  int64\_t start = timer\_ticks ();  struct thread \*cur = thread\_current();  enum intr\_level old\_level;  ASSERT (intr\_get\_level () == INTR\_ON);  ASSERT (!intr\_context());  old\_level = intr\_disable();    cur->wake\_up\_time = start + ticks;  list\_push\_back(&blocked\_threads, &cur->elem);  thread\_block();  intr\_set\_level(old\_level);  }  static void timer\_interrupt (struct intr\_frame \*args UNUSED){  struct thread\* tmp;  struct list\_elem \*e;    ticks++;  for (e = list\_begin(&blocked\_threads); e!=list\_end(&blocked\_threads); ){  tmp = list\_entry(e, struct thread, elem);  if(tmp->wake\_up\_time <= ticks){  e = list\_remove(e);  thread\_unblock(tmp);  continue; }  e = list\_next(e);}  thread\_tick (); } | -BlOCKED된 thread들을 따로 관리하는 list blocked\_threads 추가  -blocked\_threads list 초기화  -thread가 sleep할 때 깨어날 시간이 될 때까지 thread\_yield()를 하는 대신 독립적인 list에 삽입했다가 일어나야할 시간이 깨워준다.  -wake up time 계산해서 thread에 저장  -blocked\_threads list에 thread삽입  -thread를 BLOCK상태로 변경  -매 tick마다 호출되어 시간을 재는 함수  -tick을 올려준다  -blocked\_threads list에 있는 thread들의 wake\_up\_time을 모두 확인한다.  -일어날 시간이 된 thread는 blocked\_thread에서 제거한 후에 READY상태로 바꾸고 ready queue에 삽입한다. |

2. Priority Scheduler

|  |  |
| --- | --- |
| threads/thread.c |  |
| bool compare\_pri(const struct list\_elem \*a, const struct list\_elem \*b){  return list\_entry(a, struct thread, elem)->priority > list\_entry(b, struct thread, elem)->priority; }  void thread\_unblock (struct thread \*t) {  …  ASSERT (t->status == THREAD\_BLOCKED);  list\_insert\_ordered (&ready\_list, &t->elem, compare\_pri, NULL);  …  }  void thread\_yield (void) {  struct thread \*cur = thread\_current ();  …  if (cur != idle\_thread)  list\_insert\_ordered(&ready\_list, &cur->elem, compare\_pri, NULL);  …  }  void thread\_set\_priority (int new\_priority) {  struct thread \*cur = thread\_current();  int pre\_priority = cur->priority;  …  cur->priority = new\_priority;  if(new\_priority < pre\_priority)  thread\_yield();  }  tid\_t thread\_create (const char \*name, int priority, thread\_func \*function, void \*aux) {  …  if (priority > thread\_get\_priority())  thread\_yield();  …} | -ready queue에서 priority의 내림차순으로 thread가 정렬될 수 있도록 비교함수를 생성한다.  -BLOCKED 상태의 thread가 READY상태로 변경되고 ready queue에 삽입될 비교함수에 따라 정렬될 수 있게 삽입한다.  -RUNNING상태의 thread의 실행을 멈추고 이를 다시 ready queue에 삽일할 때 비교함수에 따라 정렬될 수 있게 삽입한다.  -현재 thread의 우선순위를 변경하기 전 이전 우선순위를 저장한다.  -thread의 우선순위를 새 우선순위로 변경해준다.  -만약 새 우선순위가 이전 우선순위보다 낮다면 실행을 중단하고 다시 스케줄링해서 가장 높은 우선순위를 가지는 thread를 찾는다.  -새 thread가 생성될 때 그 우선순위가 현재 실행되고 있는 thread의 우선순위보다 높다면 현재 thread의 실행을 멈춘다. 또한 다시 스케줄링하여 우선순위가 가장 높은 thread가 실행될 수 있도록 한다. |
| threads/synch.c |  |
| void sema\_up (struct semaphore \*sema) {  enum intr\_level old\_level;  int max\_pri=-1;  struct thread \*tmp, \*max\_thrd;  struct list\_elem \*e, \*max\_elem;  …  if (!list\_empty (&sema->waiters)) {  for (e=list\_begin(&sema->waiters); e!=list\_end(&sema->waiters); e = list\_next(e)){  tmp = list\_entry(e, struct thread, elem);  if(tmp->priority > max\_pri){  max\_pri = tmp->priority;  max\_thrd = tmp;  max\_elem = e; } }  list\_remove(max\_elem);  thread\_unblock(max\_thrd);  }  sema->value++;  intr\_set\_level (old\_level);  thread\_yield(); } | -sema\_up()이 호출되면 가장 높은 우선순위를 가진 thread가 깨어나 실행되도록 해야한다.  -sema\_down()이 호출되면 이를 기다리고 있는 thread들을 탐색한다.  -가장 높은 우선순위를 가지고 있는 thread의 우선순위, thread, element를 저장한다.  -가장 높은 우선순위를 가진 thread를 waiting list에서 제거한 후에 깨워준다.  -ready queue의 thread가 정렬되지 않을 수도 있기에 다시 스케줄링 해준다. |

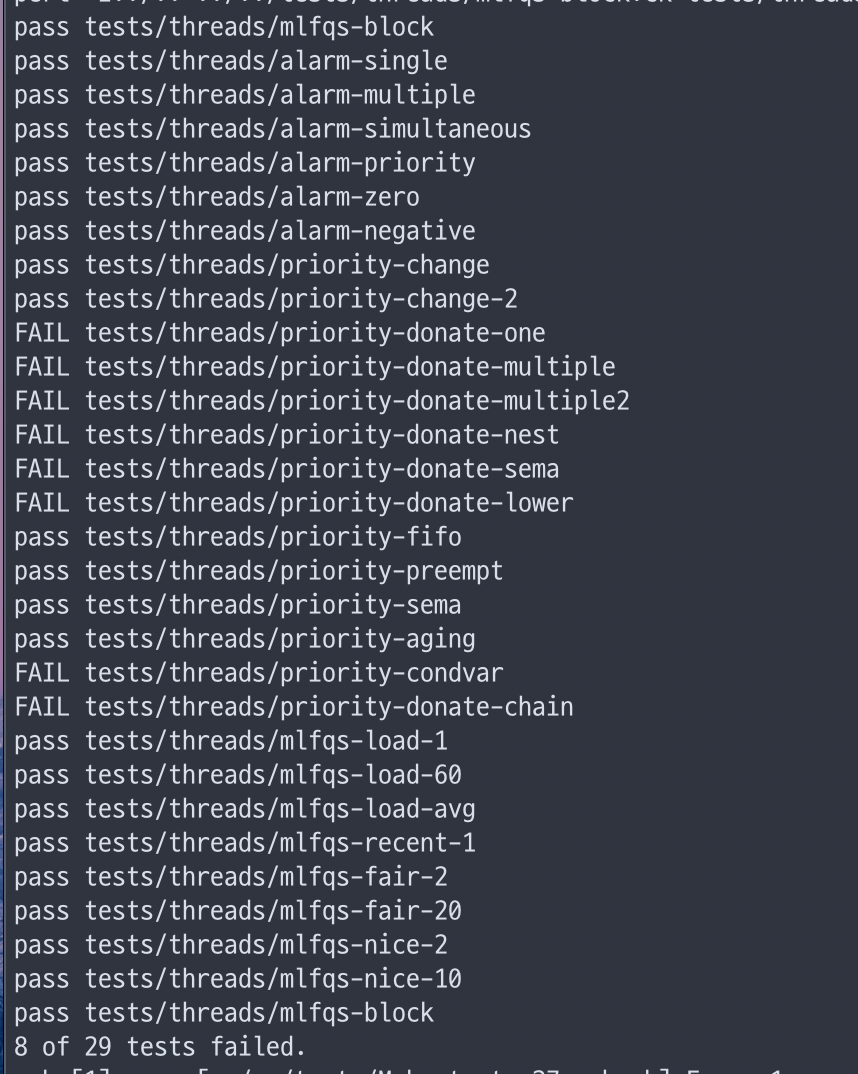
3. Aging

|  |  |
| --- | --- |
| threads/thread.c |  |
| bool thread\_prior\_aging;  void thread\_aging(void) {  struct thread\* tmp;  struct list\_elem \*e;  for (e = list\_begin(&ready\_list); e!=list\_end(&ready\_list); e = list\_next(e)){  tmp = list\_entry(e, struct thread, elem);  if(tmp->priority < PRI\_MAX)  tmp->priority++; }}  void thread\_tick (void) {  ….  #ifndef USERPROG  if (thread\_prior\_aging == true)  thread\_aging();  …  #endif } | -aging flag변수를 추가해서 실행 프로그램의 옵션으로 aging이 들어오면 이를 true로 변경한다.  -thread\_aging()함수를 추가하여 Aging을 구현한다.  -ready queue에서 실행되기를 기다리고 있는 thread를 차례대로 탐색한다.  -thread 의 우선순위가 최대값이 아닐 경우 1씩 올려준다.  -매 tick마다 호출되는 thread\_tick()에서 thread\_prior\_aging이 true라면 thread\_aging()을 호출하여 매 tick마다 READY상태의 우선순위를 높여준다. |

4. Advanced Scheduler – BSD

|  |  |
| --- | --- |
| threads/thread.h |  |
| struct thread{  …  int recent\_cpu;  int nice;  }  #define FRACTION (1<<14) | -우선순위를 계산하기 위해서 각 thread가 recent\_cpu, nice값을 가지도록 한다.  -fixed-point number연산에 사용될 FRACTION 선언 |
| threads/thread.c |  |
| void thread\_init (void) {  ASSERT (intr\_get\_level () == INTR\_OFF);  …  initial\_thread->nice = 0;  initial\_thread->recent\_cpu = 0;  }  static void init\_thread (struct thread \*t, const char \*name, int priority) {  …  t->recent\_cpu = running\_thread()->recent\_cpu;  t->nice = running\_thread()->nice; }  int f\_plus\_i(int x, int y){  return x + y \* FRACTION; }  int f\_plus\_f(int x, int y){  return x + y; }  int i\_minus\_f(int x, int y){  return x \* FRACTION - y; }  int f\_minus\_f(int x, int y){  return x - y; }  int i\_times\_f(int x, int y){  return x \* y; }  int f\_times\_f(int x, int y){  int64\_t temp = x;  return temp\*y/FRACTION; }  int f\_divide\_i(int x, int y){  return x / y; }  int f\_divide\_f(int x, int y){  int64\_t temp = x;  return temp\*FRACTION/y; }  void thread\_set\_nice (int nice UNUSED) {  thread\_current()->nice = nice;  }  int thread\_get\_nice (void) {  return thread\_current()->nice;  }  int thread\_get\_load\_avg (void) {  return i\_times\_f(100, load\_avg)/FRACTION;  }  void thread\_set\_load\_avg (void){  int ready\_threads = list\_size(&ready\_list);  if (thread\_current()!= idle\_thread)  ready\_threads++;  load\_avg = f\_divide\_i(f\_plus\_i(i\_times\_f(59, load\_avg), ready\_threads), 60);  }  int thread\_get\_recent\_cpu (void) {  return i\_times\_f(100, thread\_current()->recent\_cpu)/FRACTION;  }  void thread\_set\_recent\_cpu (void) {  struct thread \*tmp;  struct list\_elem \*e;  for (e = list\_begin (&all\_list); e != list\_end (&all\_list); e = list\_next (e)){  tmp = list\_entry (e, struct thread, allelem);  if( tmp != idle\_thread ){  tmp->recent\_cpu = f\_plus\_i(f\_times\_f(f\_divide\_f(i\_times\_f(2, load\_avg), f\_plus\_i(i\_times\_f(2, load\_avg), 1)), tmp->recent\_cpu), tmp->nice);  }}}  void thread\_set\_priority (int new\_priority) {  struct thread \*cur = thread\_current();  int pre\_priority = cur->priority;  struct list\_elem \*e;  struct thread \*tmp;    if(thread\_mlfqs){  for (e = list\_begin(&all\_list); e != list\_end (&all\_list); e = list\_next (e)){  tmp = list\_entry (e, struct thread, allelem);  tmp->priority = f\_minus\_f(f\_minus\_f(f\_plus\_i(0, PRI\_MAX), f\_divide\_i(tmp->recent\_cpu, 4)), i\_times\_f(2, f\_plus\_i(0, tmp->nice))) / FRACTION; }  return; }  … }  void thread\_tick (void) {  struct thread \*t = thread\_current ();  #ifndef USERPROG  …  if (thread\_mlfqs){  if( t != idle\_thread )  t->recent\_cpu = f\_plus\_i( t->recent\_cpu, 1);  if( timer\_ticks() % TIME\_SLICE == 0 )  thread\_set\_priority(0);  if( timer\_ticks() % TIMER\_FREQ == 0 ){  thread\_set\_load\_avg(); thread\_set\_recent\_cpu();  } } #endif } | -첫 thread의 nice 값과 recent\_cpu값을 0으로 초기화한다.  -thread가 생성될 때 부모의 recent\_cpu값과 nice값을 상속받도록 한다.  -우선순위 값을 계산하기 위해서 필요한fixed-point 연산을 추가한다.  -실수에 정수를 더하는 함수 추가, 정수를 fixed-point number로 변환하여 계산한다.  -실수에 실수를 더하는 함수 추가  -정수에서 실수를 빼는 함수 추가, 정수를 fixed-point number로 변환하여 계산한다.  -실수에서 실수를 빼는 함수 추가  -정수에서 실수를 곱하는 함수 추가  -실수에서 실수를 곱하는 함수 추가  -실수를 정수로 나누는 함수 추가  -실수를 실수로 나누는 함수 추가  -thread가 새 nice value를 가지도록 이를 setting해주는 함수 추가  -현재 thread의 nice value를 반환해주는 함수 추가  -현재 load\_avg값에 100을 곱한 값을 반환하는 함수 추가  -load\_avg값을 “*load\_avg* = (59/60) \* *load\_avg* + (1/60) \* ready\_threads”식에 따라 계산해주는 함수 추가  -현재 READY상태인 thread개수 계산  -RUNNING상태인 thread 개수를 더한다.  -fixed-point 연산으로 load\_avg값을 계산한다.  -현재 thread의 recent\_cpu값에 100을 곱한 값을 반환하는 함수 추가  -모든 thread의 recent\_cpu값을”*recent\_cpu* = (2\*load\_avg) / (2\*load\_avg+1) \* *recent\_cpu* + nice”식에 따라 계산하는 함수 추가  -모든 thread를 탐색한다.  -idle이 아닌 thread의 recent\_cpu값을 fixed-point연산으로 계산한다.  -BSD 스케줄러를 사용할 경우 “priority = PRI\_MAX – (recent\_cpu / 4) – (nice \* 2)”식에 따라서 우선순위를 계산하도록 수정  -flag 변수로 BSD 스케줄러를 사용하는 지 판단  -모든 thread를 탐색한다.  -thread의 우선순위를 fixed-point 연산을 사용해서 계산하고 정수로 변환한다.  -매 tick마다 호출되는 thead\_tick()에서 매tick마다 RUNNING thread의 recent\_cpu값을 올리고, 매 초마다 recent\_cpu, load\_avg값을 계산하고 매 4 tick마다 우선순위를 계산한다.  - BSD 스케줄러를 사용할 때만 작동하도록 한다.  -RUNNING thread의 recent\_cpu를 매 tick마다 1씩 올린다.  -TIME\_SLICE(4 ticks)마다 우선순위를 다시 계산한다.  -TIMER\_FREQ(1 second)마다 recent\_cpu와 load\_avg값을 다시 계산한다. |

* 1. **시험 및 평가 내용**



모든 구현을 마친 후 threads 디렉토리에서 make check를 수행한 결과는 위와 같다. 프로젝트에서 요구된 모든 testcase를 통과한 것을 알 수 있다. 또한 추가로 구현한 BSD 스케줄러 역시 구현이 성공적으로 됨을 알 수 있다.

따로 확인을 해야하는 테스트 케이스 priority-lifo는 우선순위가 모두 다른 thread 16개를 생성한 후에 자신의 번호를 output으로 16번 출력한다. 각 thread의 이름은 정수 값으로 0번부터 15번까지 있다. 0번 thread의 우선순위는 32이며 1번은 33, 2번은 34로 차례대로 우선순위가 높은 thread가 생성된다. 결국 15번 thread의 우선순위가 47로 가장 높은 값을 가지게 된다. 우선순위 스케줄러가 정상적으로 구현되었다면 15번 thread부터 차례대로 스케줄되어 실행되어야 한다. 아래의 결과와 마찬가지로 15번 thread부터 우선순위에 따라서 thread번호를 출력하는 것을 알 수 있다. 따라서 priority-lifo testcase 또한 통과했음을 알 수 있다.

