1. **Preemptive dynamic priority scheduling**

~/pintos/src/threads/thread.c

**more\_priority() / thread\_unblock() / thread\_yield()**

void

thread\_unblock (struct thread \*t)

{

enum intr\_level old\_level;

ASSERT (is\_thread (t));

old\_level = intr\_disable ();

ASSERT (t->status == THREAD\_BLOCKED);

list\_insert\_ordered (&ready\_list, &t->elem, more\_priority, NULL); // (2)

t->status = THREAD\_READY;

intr\_set\_level (old\_level);

}

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

{

struct thread \*t1 = list\_entry(a,struct thread,elem);

struct thread \*t2 = list\_entry(b,struct thread,elem);

if(t1->priority <= t2->priority) // (1)

return false;

else return true;

}

스레드가 ready queue에 들어가는 경우는 thread\_unblock() 함수와 thread\_yield()를 호출할 때이다. 이 두 함수 안에서 스레드가 ready queue에 들어가게 되는 부분의 코드를 우선순위가 높은 스레드가 앞으로 올 수 있도록 함수 more\_priority를 정의하고, ready queue를 우선순위에 따라 유지할 수 있도록 수정하였다.

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, more\_priority, NULL); //(2)

cur->status = THREAD\_READY;

schedule ();

intr\_set\_level (old\_level);

}

1. list의 앞에 오는 스레드의 우선순위가 거나 같으면 false를 리턴하여 list의 앞에는 우선순위가 높은 스레드가 올 수 있도록 함수를 정의한다.
2. 스레드를 ready\_list에 삽입할 때, 위에서 정의한 함수 more\_priority의 조건에 따라 정렬하며 삽입하여 우선순위가 높은 스레드가 list의 앞에 올 수 있도록 ready\_list를 유지한다.

**thread\_create() / thread\_set\_priority() / run\_max\_priority()**

tid\_t thread\_create (const char \*name, int priority, thread\_func \*function, void \*aux)

{

(생략)

/\* Add to run queue. \*/

thread\_unblock (t);

**run\_max\_priority(); //(1)**

return tid;

}

void

thread\_set\_priority (int new\_priority)

{

enum intr\_level old\_level = intr\_disable();

int old\_priority = thread\_current ()->priority;

thread\_current()->init\_priority = new\_priority;

restore\_priority();

if (old\_priority < thread\_current()->priority)

donate\_priority();

if (old\_priority > thread\_current()->priority)

run\_max\_priority();

intr\_set\_level (old\_level);

}

preemptive dynamic priority 스케줄링을 구현하기 위해 실행중인 스레드의 우선순위를 설정할 수 있는 thread\_set\_priority()함수를 구현하였고, 선점형 스케줄링을 위해 스레드의 우선순위가 새로 설정되거나 새로 만들어진 스레드가 ready queue에 들어갈 때마다 실행중인 스레드의 우선순위를 ready\_list에서 가장 높은 우선순위를 가진 스레드와 비교하여 현재 실행중인 스레드의 우선순위가 더 낮을 경우 선점이 발생하고 스케줄링이 일어나도록 run\_max\_priority()함수를 구현하여 수정하였다. run\_max\_priority()는 가장 높은 우선순위를 가진 스레드가 실행할 수 있도록 해주는 함수이다.

void run\_max\_priority (void)

{

if (list\_empty(&ready\_list)) // (2)

return;

struct thread \*t = list\_entry(list\_front(&ready\_list),struct thread,elem); // (3)

if (thread\_current()->priority < t-> priority) // (4)

thread\_yield();

}

1. run\_max\_priority() 함수를 호출하여 ready\_list에서 우선순위가 제일 높은, 첫번째 스레드의 우선순위 보다 현재 스레드의 우선순위가 더 낮을 경우 선점이 발생하도록 하였다. 만약 새로 생성된 스레드의 우선순위가 현재 실행중인 스레드의 우선순위보다 높다면, ready\_list의 첫번째에 위치하게 되고, ready\_list의 첫번째 스레드가 현재 실행중이 스레드보다 우선순위가 높으므로 선점이 일어나게 된다. (새로 생성되는 스레드가 실행됨)
2. ready\_list가 비어있으면 리턴한다.
3. ready\_list의 맨 앞의 원소를 t에 저장한다. (ready\_list에서 우선순위가 가장 높은 스레드)
4. t와 현재 실행중인 스레드의 우선순위를 비교하여 현재 실행중인 스레드의 우선순위가 더 낮다면 thread\_yield() 함수를 호출하여 선점이 발생하도록 한다.

**thread\_tick()**

void

thread\_tick (void)

{

struct thread \*t = thread\_current ();

/\* Update statistics. \*/

if (t == idle\_thread)

idle\_ticks++;

#ifdef USERPROG

else if (t->pagedir != NULL)

user\_ticks++;

#endif

else

kernel\_ticks++;

/\* Enforce preemption. \*/

**/\*if (++thread\_ticks >= TIME\_SLICE) // (1)**

**intr\_yield\_on\_return ();\*/**

}

Round Robin scheduling의 time slice 만료시 preempt를 제거하기 위해 thread\_tick() 함수에서 현재 스레드의 실행시간 (thread\_ticks)이 TIME SLICE보다 클 경우 인터럽트 핸들러가 인터럽트에서 돌아오기 전에 새로운 프로세스를 실행하도록 하는, 즉 선점이 발생하도록하는 intr\_yield\_on\_return() 함수를 호출하는 부분을 삭제하였다.

~/pintos/src/threads/synch.c

**sema\_down()**

void

sema\_down (struct semaphore \*sema)

{

enum intr\_level old\_level;

ASSERT (sema != NULL);

ASSERT (!intr\_context ());

old\_level = intr\_disable ();

while (sema->value == 0)

{

list\_insert\_ordered (&sema->waiters, &thread\_current ()->elem, more\_priority, NULL); //(1)

thread\_block ();

}

sema->value--;

intr\_set\_level (old\_level);

}

1. semaphore의 wait queue 도 우선순위 순으로 유지하기 위해, 스레드를 위에서 정의한 함수 more\_priority의 조건에 따라 정렬하며 삽입하여 우선순위가 높은 스레드가 waiters의 앞에 올 수 있도록 하였다.

**sema\_up()**

void

sema\_up (struct semaphore \*sema)

{

enum intr\_level old\_level;

ASSERT (sema != NULL);

old\_level = intr\_disable ();

if (!list\_empty (&sema->waiters))

{

list\_sort(&sema->waiters, more\_priority, NULL); // (1)

thread\_unblock (list\_entry (list\_pop\_front (&sema->waiters),

struct thread, elem));

}

sema->value++;

run\_max\_priority(); // (2)

intr\_set\_level (old\_level);

}

1. semaphore의 wait queue를 정렬한 후 thread\_unblock()함수를 호출하여 wait queue를 정렬된 상태로 유지할 수 있도록 하였다.
2. run\_max\_priority() 함수를 호출하여 ready\_list에서 우선순위가 제일 높은, 첫번째 스레드의 우선순위 보다 현재 스레드의 우선순위가 더 낮을 경우 선점이 발생하도록 하였다.
3. **priority donation (inheritance)**

~/pintos/src/threads/thread.h , thread.c

**struct thread / init\_thread()**

우선순위 상속을 구현하기 위해 스레드 구조체에 몇가지 멤버 변수를 더하고, init\_thread()함수 안에 초기화 시키는 코드를 추가하였다.

init\_thread (struct thread \*t, const char \*name, int priority)

{

(생략)

t->init\_priority = priority; // (1)

t->wait\_on\_lock = NULL; // (2)

list\_init(&t->donations); // (3)

}

struct thread

{

(생략)

int init\_priority; // (1)

struct lock \*wait\_on\_lock; // (2)

struct list donations; // (3)

struct list\_elem donation\_elem; // (4)

};

1. 우선순위를 상속 받기 전 원래의 우선순위를 저장하는 필드이며 스레드 생성시에 주어지는 우선순위 값으로 초기화 한다.
2. 스레드가 얻고자하는 lock을 저장하는 필드이고, NULL로 초기화 한다.
3. lock을 이미 다른 스레드가 가지고 있는 경우에 lock을 얻기위해 시도한 스레드의 리스트이고 이 리스트를 초기화 한다.
4. donations에 스레드를 연결할 때 사용할 필드이다.

**donate\_priority()**

lock을 가지고 있는 스레드 보다 lock을 요청한 스레드의 우선순위가 더 높을 경우 lock을 가지고 있는 스레드에게 lock을 요청한 스레드(현재 실행중인 스레드)의 우선순위만큼 상속해 주는 함수이다.

1. 현재 스레드가 얻고자 하는 lock을 변수 l에 저장한다.

void

donate\_priority (void)

{

struct thread \*t = thread\_current();

struct lock \*l = t->wait\_on\_lock; // (1)

while (l != NULL) // (2)

{

if (!l->holder) // (3)

return;

if (l->holder->priority >= t->priority) // (4)

return;

l->holder->priority = t->priority; // (5)

l = l->holder->wait\_on\_lock; // (6)

}

}

1. 얻고자하는 lock이 없을 때까지 반복한다.
2. lock을 가지고 있는 thread가 없으면 리턴한다.
3. lock을 가지고 있는 스레드의 우선순위가 더 높거나 같으면 리턴한다.
4. (3),(4)의 경우가 아닌 경우는 lock을 가지고 있는 스레드의 우선순위가 lock을 요청한 스레드(현재 실행중인 스레드)의 우선순위보다 낮은 경우이다. 이 경우에는 우선순위를 상속받아야 하므로, lock을 가지고 있는 스레드의 우선순위에 현재 스레드의 우선순위 값을 저장한다.
5. l에 lock을 가지고 있는 스레드의 wait\_on\_lock의 값을 저장하고 l의 값이 NULL이 될 때 까지 반복하여 lock을 가지고 있는 스레드가 또 다른 lock을 얻고자 했을 경우에 필요하다면 우선순위를 상속할 수 있도록 한다. (test 중 priority-donate-nest와 관련)

**restore\_priority()**

void

restore\_priority(void)

{

struct thread \*t = thread\_current();

t->priority = t->init\_priority; // (1)

if (list\_empty(&t->donations)) // (2)

return;

struct thread \*s = list\_entry(list\_front(&t->donations), struct thread, donation\_elem); // (3)

if (s->priority > t->priority) // (4)

t->priority = s->priority;

}

우선순위를 상속받기 전 원래의 우선순위로 돌려주는 함수이다.

1. 현재 실행중인 스레드의 우선순위를 우선순위를 상속받기 전 원래 우선순위로 설정한다.
2. 현재 실행중인 스레드가 가지고 있는 lock을 요청한 스레드가 없다면 리턴한다.
3. (2)의 경우가 아니라면 현재 실행중인 스레드의 donations의 첫번째 원소를 변수 s에 저장한다.
4. 만약 s의 우선순위가 현재 실행중인 스레드의 우선순위보다 높다면 현재 실행중인 스레드의 우선순위를 s의 우선순위 값으로 설정한다.

~/pintos/src/threads/synch.c

**remove\_lock()**

void

remove\_lock (struct lock \*lock)

{

struct list\_elem \*e = list\_begin(&thread\_current()->donations);

struct list\_elem \*next;

while (e != list\_end(&thread\_current()->donations)) // (1)

{

struct thread \*t = list\_entry(e, struct thread, donation\_elem); // (2)

if (t->wait\_on\_lock == lock) // (3)

list\_remove(e);

e = list\_next(e); // (4)

}

}

현재 실행중인 스레드가 가지고 있는 lock을 요청한 스레드를 donations에서 제거하는 함수이다.

1. 현재 실행중인 스레드의 donations를 확인한다. (현재 실행중인 스레드가 가지고 있는 lock을 요청한 스레드의 리스트를 확인한다.)
2. donations의 첫번째 원소를 변수 t에 저장한다.
3. 만약 t가 요청한 lock이 매개변수로 들어온 lock과 같을 경우 donations에서 t에 해당하는 스레드를 제거한다.
4. donations의 다음원소를 확인한다.

**lock\_acquire()**

void

lock\_acquire (struct lock \*lock)

{

ASSERT (lock != NULL);

ASSERT (!intr\_context ());

ASSERT (!lock\_held\_by\_current\_thread (lock));

enum intr\_level old\_level = intr\_disable();

if (lock->holder != NULL) // (1)

{

thread\_current()->wait\_on\_lock = lock; // (2)

list\_insert\_ordered(&lock->holder->donations,&thread\_current()->donation\_elem, more\_priority, NULL); // (3)

donate\_priority(); // (4)

}

sema\_down (&lock->semaphore); // (5)

thread\_current()->wait\_on\_lock = NULL; // (6)

lock->holder = thread\_current (); // (7)

intr\_set\_level(old\_level);

}

1. 요청한 lock을 다른 스레드가 가지고 있을 경우에 우선순위 상속이 실행된다.
2. 현재 스레드의 wait\_on\_lock에 매개변수로 전달받은 lock을 저장한다. 현재 스레드가 요청한 lock을 wait\_on\_lock 필드에 저장한다.
3. lock을 가지고 있는 스레드의 donations에 현재 스레드를 연결한다. donations에는 해당 스레드가 가지고 있는 lock을 요청한 스레드들이 연결되어 있다.
4. donate\_priority()함수를 호출하여 lock을 가지고 있는 스레드보다 현재 스레드의 우선순위가 더 높을 경우 우선순위를 상속할 수 있도록 한다.
5. 현재 스레드가 sema\_down()함수를 통해 lock을 얻는다.
6. 현재 스레드가 lock을 얻은 경우 , 현재 스레드의 wait\_on\_lock 필드를 NULL로 설정해준다.
7. lock을 가지고 있는 스레드를 현재 스레드로 바꿔준다.

**lock\_release()**

void

lock\_release (struct lock \*lock)

{

ASSERT (lock != NULL);

ASSERT (lock\_held\_by\_current\_thread (lock));

enum intr\_level old\_level = intr\_disable();

lock->holder = NULL; // (1)

remove\_lock(lock); // (2)

restore\_priority(); // (3)

sema\_up (&lock->semaphore);

intr\_set\_level(old\_level);

}

가지고 있던 lock을 해제하는 함수이다,

1. lock을 가지고 있는 스레드를 NULL로 설정한다.
2. remove\_lock()함수를 호출하여 현재 스레드의 donations에서 해당 lock을 요청했던 스레드를 제거한다.
3. restore\_priority()함수를 호출하여 상속받았던 우선순위를 복구한다.
4. **thread\_get\_priority() / thread\_set\_priority()**

void

thread\_set\_priority (int new\_priority)

{

enum intr\_level old\_level = intr\_disable();

int old\_priority = thread\_current ()->priority; // (1)

thread\_current()->init\_priority = new\_priority; // (2)

restore\_priority(); // (3)

if (old\_priority < thread\_current()->priority) // (4)

donate\_priority();

if (old\_priority > thread\_current()->priority) // (5)

run\_max\_priority();

intr\_set\_level (old\_level);

}

int

thread\_get\_priority (void)

{

return thread\_current ()->priority; // (6)

}

~/pintos/src/threads/thread.c

1. 새로운 우선순위를 설정하기 전 원래의 우선순위를 변수 old\_priority에 저장한다.
2. 현재 스레드의 init\_priority를 새로운 우선순위로 설정한다.
3. restore\_priority()함수를 호출하여 우선순위를 새롭게 설정한다. 새로운 우선순위가 init\_priority가 되어서 현재 스레드의 우선순위를 새로운 우선순위로 설정하고, 현재 스레드가 우선순위를 상속받는 중이라면 우선순위를 낮출 수 없게 하기 위해서이다.
4. 기존의 우선순위보다 새롭게 설정한 우선순위가 더 높을 경우 donate\_priority() 함수를 호출하여 우선순위 상속이 다시 이루어지도록 한다. 현재 실행중인 스레드가 lock을 요청한 상태일 경우 lock을 가지고 있는 스레드의 우선순위를 현재 스레드의 새로 설정한 우선순위만큼 더 높여줘야 하기 때문이다.
5. 기존의 우선순위보다 새롭게 설정한 우선순위가 더 낮을 경우 run\_max\_priority() 함수를 호출하여 ready\_list에서 우선순위가 제일 높은, 첫번째 스레드의 우선순위 보다 현재 스레드의 우선순위가 더 낮아졌을 경우 선점이 발생하도록 하였다.
6. 현재 스레드의 우선순위를 반환한다.
7. **test 실행 결과**

**텍스트이(가) 표시된 사진

매우 높은 신뢰도로 생성된 설명**