**Pintos Project 3: Threads**

담당 교수 : 김영재 교수님

조 / 조원 : 조현지

개발 기간 : 11/16 – 11/25

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술.

**스케줄링 알고리즘을 구현한다. 기존의 Round Robin 스케줄러에 Alarm Clock 알고리즘을 더해준다. 이후, 스레드의 우선순위를 기반으로 작업이 진행되도록 구현한다. 마지막으로, MLFQ 스케줄러를 구현한다.**

1. **개발 범위 및 내용**
   1. **개발 범위**

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

스레드가 매 ticks마다 Ready와 Running 상태를 반복 전환하는 비효율성을 줄이고자 Alarm clock을 도입하고자 한다. 이를 통해, 스레드는 지정된 ticks가 지나기 전까지는 ready 상태에 머물러있다 지정된 ticks가 지나고 나면 woke up 된다. 스레드의 상태가 ticks마다 전환되지 비효율성을 감소시킬 수 있다.

* 1. Priority Scheduling

하나의 Ready 리스트에 들어간 스레드는 우선순위에 상관없이 FIFO 형식으로 작업이 처리된다. 우선순위가 높은 스레드가 CPU를 빠르게 확보할 수 있도록, Ready 리스트에 스레드를 추가할 때 우선순위를 고려한다. 이를 통해, 우선순위가 낮은 스레드가 CPU 자원이 선점하는 경우를 방지할 수 있다.

* 1. Advanced Scheduler (추가구현을 한 경우)
  2. **개발 내용**
* 아래 항목의 내용만 서술

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

Timer\_interrupt() 함수를 이용한다.

* + - Ready list 외에 추가로 wait queue list를 만들어, block 상태의 스레드를 기억해둔다. 또한, block 상태의 스레드는 woke up할 시간을 스레드에 저장해둔다. Timer\_interrupt() 함수는 매 ticks마다 wait queue list에 저장된 모든 스레드의 woke up 시간을 확인한다. 만약 woke up 시간이 지났으면, thread\_unblock() 함수를 이용해 스레드를 ready 상태로 바꿔준다.

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

Thread\_yield() 함수를 이용한다.

* + - 현재 running 상태인 스레드와 새롭게 들어온 thread의 우선순위를 비교한다. 만약 새롭게 들어온 thread의 우선순위가 더 높다면, thread\_yield() 함수를 이용해 즉시 CPU를 양보해주도록 한다.

1. Advanced Scheduler에서 priority 계산에 필요한 각 요소를 서술. (추가구현을 한 경우)
2. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

|  |  |
| --- | --- |
| 2024.11.16 ~ 2024.11.20 | Alarm Clock 구현 |
| 2024.11.20 ~ 2024.11.25 | Priority Scheduling 구현 |

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

|  |  |
| --- | --- |
| **Thread.h** | |
| struct thread  {  /\* Owned by thread.c. \*/  tid\_t tid; /\* Thread identifier. \*/  enum thread\_status status; /\* Thread state. \*/  char name[16]; /\* Name (for debugging purposes). \*/  uint8\_t \*stack; /\* Saved stack pointer. \*/  int priority; /\* Priority. \*/  struct list\_elem allelem; /\* List element for all threads list. \*/  /\* Shared between thread.c and synch.c. \*/  struct list\_elem elem; /\* List element. \*/  }; | **[Alarm Clock - 수정 사항]**  - woke up 시간을 저장할 정수 변수 추가하기  - block 상태의 스레드를 담아둘 리스트 변수 추가하기 (외부 참조 가능)  - wait 리스트 내에서의 우선순위순 정렬을 위한 comparator 함수 정의하기 |
| **Thread.c** | |
| /\* List of processes in THREAD\_READY state, that is, processes  that are ready to run but not actually running. \*/  static struct list ready\_list;  /\* List of all processes. Processes are added to this list  when they are first scheduled and removed when they exit. \*/  static struct list all\_list;  /\* Idle thread. \*/  static struct thread \*idle\_thread; | **[Priority Scheduling - 수정 사항]**  - 우선순위 스케줄러를 위해 ready list를 1개에서 64개로 늘리기  - block 상태의 스레드를 담아둘 리스트 변수 추가하기 (외부 참조 가능)  - wait 리스트 내의 우선순위순 정렬을 위한 함수 구현하기  - schedule 함수 내의 next\_thread\_to\_run 함수를 64개의 우선순위 큐에 따라 움직이게 구현하기 |
| 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\_push\_back (&ready\_list, &t->elem);  t->status = THREAD\_READY;  intr\_set\_level (old\_level);  } | **[Alarm Clock - 수정 사항]**  - 스레드의 우선순위에 해당하는 ready 리스트에 스레드를 리스트에 삽입한다. 이때, ready\_list에 삽입된 스레드가 현재 running 상태인 스레드보다 우선순위가 높다면 thread\_yield()를 통해 running 스레드가 CPU를 양보할 수 있게 한다. |
| **Timer.c** | |
| /\* Sleeps for approximately TICKS timer ticks. Interrupts must  be turned on. \*/  void  timer\_sleep (int64\_t ticks)  {  int64\_t start = timer\_ticks ();  ASSERT (intr\_get\_level () == INTR\_ON);  while (timer\_elapsed (start) < ticks)  thread\_yield ();  } | **[Alarm Clock - 수정 사항]**  - 현재 실행 중인, 스레드를 wait 리스트에 밀어넣는다. 이때, thread\_block() 함수를 이용한다. |
| \* Timer interrupt handler. \*/  static void  timer\_interrupt (struct intr\_frame \*args UNUSED)  {  ticks++;  thread\_tick ();  } | **[Alarm Clock - 수정 사항]**  - wait 리스트 속의 모든 스레드가 가리키는 woke up 시간과 ticks를 비교하여, woke up 시간이 지났으면 스레드의 상태를 blocked에서 ready 상태로 바꾼다. 이때, thread\_unblock() 함수를 이용한다. |
| **Synch.c** | |
| 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\_push\_back (&sema->waiters, &thread\_current ()->elem);  thread\_block ();  }  sema->value--;  intr\_set\_level (old\_level);  } | **[Priority Scheduling - 수정 사항]**  - 스레드에서 만들어둔 comparator 함수를 이용하여, sema의 wait 리스트에 들어가는 스레드를 우선순위 순으로 정렬 삽입한다. 이후, thread\_block()을 통해 스레드의 상태를 변환한다. |
| void  sema\_up (struct semaphore \*sema)  {  enum intr\_level old\_level;  ASSERT (sema != NULL);  old\_level = intr\_disable ();  if (!list\_empty (&sema->waiters))  thread\_unblock (list\_entry (list\_pop\_front (&sema->waiters),  struct thread, elem));  sema->value++;  intr\_set\_level (old\_level);  } | **[Priority Scheduling - 수정 사항]**  - wait 리스트의 가장 앞에 있는 스레드를 뽑아낸다. Thread\_unblock() 함수를 이용해서, 스레드의 상태를 ready로 변환시킨다. |

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

|  |  |
| --- | --- |
| 1 | Blocked 상태로 변환하기 위해, Timer\_sleep() 함수를 실행한다. |
| 2 | Timer\_sleep() 함수가 woke up 해야할 시간을 스레드의 woke up 변수에 저장한 뒤, 스레드를 wait 리스트에 추가한다. |
| 3 | Timer\_interrupt() 함수는 매 ticks마다 wait 리스트 내 모든 스레드의 woke up 시간을 확인한다. |
| 4 | 만약 woke up 시간이 ticks를 지났다면, wait 리스트에서 제거한 뒤 thread\_unblock() 함수를 실행한다. |
| 5 | 만약 woke up 시간이 지나지 않았다면, 다음 스레드로 넘어간다. |

Priority Scheduling

|  |  |
| --- | --- |
| 1 | 스레드의 우선순위에 따라 ready list에 넣어준다. |
| 2 | Sema\_down을 했을 경우에는, sema의 wait 리스트에 우선순위에 맞춰 삽입한다. |
| 3 | Sema\_up을 했을 경우에는, sema의 가장 앞에 있는 우선순위가 가장 높은 스레드를 wait 리스트에서 뽑아낸 뒤 thread\_unblock() 함수를 이용한다. 스레드의 우선순위에 맞는 ready 리스트의 제일 마지막에 스레드를 삽입한다. |
| 4 | 3번에서 삽입하기 전에, 현재 실행중인 스레드보다 우선순위가 큰 스레드인지 확인한다. |

* 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

|  |  |
| --- | --- |
| **Thread.h** | |
| **[Alarm Clock - 수정 사항]**  - woke up 시간을 저장할 정수 변수 추가하기  - block 상태의 스레드를 담아둘 리스트 변수 추가하기 (외부 참조 가능)  - wait 리스트 내에서의 우선순위순 정렬을 위한 comparator 함수 정의하기 | - int wake\_ticks  - struct list wait\_queue  - bool thread\_priority\_comprator  : 두 개의 스레드를 가져온 뒤, 왼쪽 스레드가 오른쪽 스레드보다 큰 지에 대한 True or False 값을 반환한다.  -> 이후에, sema의 wait 리스트를 정렬 위해 list\_insert\_ordered에서 사용된다. |
| **Thread.c** | |
| **[Priority Scheduling - 수정 사항]**  - 우선순위 스케줄러를 위해 ready list를 1개에서 64개로 늘리기  - block 상태의 스레드를 담아둘 리스트 변수 추가하기 (외부 참조 가능)  - wait 리스트 내의 우선순위순 정렬을 위한 함수 구현하기  - schedule 함수 내의 next\_thread\_to\_run 함수를 64개의 우선순위 큐에 따라 움직이게 구현하기 | - static struct list ready\_list[64]  - struct list wait\_queue  - bool thread\_priority\_comparator( const struct list\_elem \*a, const struct list\_elem \*b, void \*aux UNUSED);  - next\_thread\_to\_run (void)  : 64개의 우선순위 ready 리스트를 우선순위가 가장 높은 순서대로 for문을 통해 순회한다. 만약 우선순위 ready 리스트가 비어있다면 바로 다음 우선 순위 ready 리스트로 넘어간다. 만약 ready 리스트에 스레드가 있다면, 가장 앞에 있는 스레드를 ready 리스트에서 뽑아낸 뒤, 반환한다. 같은 우선순위 리스트 내에서는 RR 스케줄링이 적용되기 때문에 리스트에 들어간 순서대로 처리된다. |
| **[Alarm Clock - 수정 사항]**  - 스레드의 우선순위에 해당하는 ready 리스트에 스레드를 리스트에 삽입한다. 이때, ready\_list에 삽입된 스레드가 현재 running 상태인 스레드보다 우선순위가 높다면 thread\_yield()를 통해 running 스레드가 CPU를 양보할 수 있게 한다. | - thread\_unblock (struct thread \*t)  : 스레드의 우선순위에 맞는 ready 리스트에 스레드를 삽입한다. 만약, 삽입한 스레드의 우선순위가 현재 실행중인 스레드보다 크다면 thread\_yield() 함수를 실행한다. |
| **Timer.c** | |
| **[Alarm Clock - 수정 사항]**  - 현재 실행 중인, 스레드를 wait 리스트에 밀어넣는다. 이때, thread\_block() 함수를 이용한다. | - timer\_sleep()  : 현재 실행중인 스레드를 wait 리스트의 제일 마지막에 삽입한다. 이후, 스레드의 상태를 변경하기 위해 thread\_block() 함수에 해당 스레드를 매개변수로 전달한다. |
| **[Alarm Clock - 수정 사항]**  - wait 리스트 속의 모든 스레드가 가리키는 woke up 시간과 ticks를 비교하여, woke up 시간이 지났으면 스레드의 상태를 blocked에서 ready 상태로 바꾼다. 이때, thread\_unblock() 함수를 이용한다. | - 매 ticks마다 wait\_queue 전체를 for문을이용해서 순회한다.  1) 만약, ticks가 스레드에 정의해둔 woke up 시간보다 작다면 스레드의 상태 변환 없이 다음 스레드로 넘어간다.  2) 만약, ticks가 스레드에 정의해둔 woke up 시간보다 크거나 같다면, wait 스레드에서 해당 스레드를 제거한다. 이후, 스레드의 상태를 ready로 변경하기 위해 thread\_unblock 함수의 매개변수에 해당 스레드를 넘긴다. |
| **Synch.c** | |
| **[Priority Scheduling - 수정 사항]**  - 스레드에서 만들어둔 comparator 함수를 이용하여, sema의 wait 리스트에 들어가는 스레드를 우선순위 순으로 정렬 삽입한다. 이후, thread\_block()을 통해 스레드의 상태를 변환한다. | - sema\_down (struct semaphore \*sema)  : thread\_priority\_comparator 함수를 이용해, sema의 wait 리스트가 우선순위가 높은 스레드가 가장 앞에 있게 정렬한다. List\_insert\_ordered 함수를 이용한다. 이후, 스레드의 상태를 변환하기 위해 thread\_ block() 함수를 실행한다. |
| **[Priority Scheduling - 수정 사항]**  - wait 리스트의 가장 앞에 있는 스레드를 뽑아낸다. Thread\_unblock() 함수를 이용해서, 스레드의 상태를 ready로 변환시킨다. | - sema\_up (struct semaphore \*sema)  : sema의 wait 리스트가 비어있지 않다면, wait 리스트의 가장 앞에 있는 스레드를 뽑아낸다. 이후, thread\_unblock() 함수의 매개변수로 해당 스레드를 넘겨준다. |

* 1. **시험 및 평가 내용**
* priority-lifo.c 코드 및 priority-lifo 테스트 결과 분석

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

**[코드]**

: 각각의 우선순위에 대해 THREAD\_CNT개 만큼의 스레드를 생성한다. 우선순위가 높은 스레드가 먼저 처리된다. 락이 걸린 상태에서는 thread\_set\_priority를 통한 스레드의 우선순위 변화가 적용되지 않는다. 락이 풀리고 나면, 스케줄러가 변동된 우선순위를 반영한다. 락이 걸린 스레드보다 우선순위가 더 높은 스레드가 대기하게 되는 경우가 발생하더라도, 락이 풀리기 전까지는 락이 걸린 스레드를 우선적으로 처리한다. 또한, output 배열에 대해 한번에 하나의 스레드만 접근할 수 있다.

**[테스트]**

: 각기 다른 16개의 우선순위에 대해 우선순위가 가장 높은 순서대로 스레드가 실행되었다. 또한 반복 횟수가 16번이기에, 같은 스레드에 대해 16번 연속으로 실행된다. 락은 output 배열을 보호한다. 따라서, 데이터 손상 없이 배열에 16개의 스레드의 id를 출력할 수 있게 된다.

* make check 수행 결과를 캡처하여 첨부

