운영체제

Pintos Project #2 – Alarm clock의 개선

****

컴퓨터과학전공

운영체제 2분반

201311154 이재민

**목차**

**Ⅰ. Alarm clock의 개선**

**1. 문제 정의**

**2. 해결**

(a) 해결 방법

(b) 구체적 개선 방안

**3. 소스 코드 수정 및 분석**

(a) /pintos/src/devices/timer.c 수정

(b) /pintos/src/threads/init.c 수정

**4. 테스트**

(a) “pintos -v -- run alarm-single” 실행

(b) “pintos -v -- run alarm-multiple” 실행

(c) “pintos -v -- run alarm-simultaneous” 실행

(d) “pintos -v -- run alarm-priority” 실행

(e) “pintos -v -- run alarm-zero” 실행

(f) “pintos -v -- run alarm-negative” 실행

Ⅰ. Alarm clock의 개선

1. 문제 정의 : Busy waiting 방식

|  |
| --- |
| void  timer\_sleep (int64\_t ticks)  {  **int64\_t start = timer\_ticks (); // (1)**  **ASSERT (intr\_get\_level () == INTR\_ON); // (2)**  **while (timer\_elapsed (start) < ticks) // (3)**  **thread\_yield ();**  } |

**<pintos/src/devices/timer.c>**

🡪 소스 코드 분석

(1) 타이머가 설정된 현재 시각을 start 변수에 저장한다.

(2) 인터럽트가 켜져 있는지 확인하고 켜져 있으면 계속 진행한다.

(3) 설정된 지연 시간이 안되었다면(아직 알람의 때가 아님), CPU를 점유 해제한다. (running 🡪 ready)

🡪 문제점 : Busy waiting 방식에서는 thread가 sleep된 이후부터 ticks 까지 단위 시간이 되지 않았다면 thread\_yield()를 반복 호출하게 된다. 알람의 때가 되지 않았다는 것인데, 이 과정에서도 계속 CPU를 점유하게 되므로 낭비가 발생한다.

2. 해결

(a) 해결 방법

🡪 timer\_sleep() 함수를 호출한 thread들이 대기할 수 있는 리스트를 만들고, 현재 시각 기준으로 ticks 시간이 경과할 때까지 block을 시킨다. 이후 ticks 이상이 경과하면 해당 thread를 리스트에서 꺼내고 unblock을 시켜준다.

(b) 구체적 개선 방안

- timer\_sleep() 함수 개선 : thread를 리스트에 매달고, thread\_block() 함수를 호출하여 해당 thread를 block 시킨다.

- timer\_interrupt() 함수 개선 : 인터럽트 처리 과정에서 리스트 중 시간이 만료된 것이 있는지 검사한다. 만료된 thread가 있다면 리스트에서 꺼내고 해당 thread는 thread\_unblock() 함수를 호출하여 thread를 깨운다.

3. 소스 코드 수정 및 분석

(a) /pintos/src/devices/timer.c 수정

- 변수 및 각종 함수, 구조체 선언

|  |
| --- |
| static bool alarm\_switch = false; // **(1)**  static bool less\_arr (const struct list\_elem \*a, const struct list\_elem \*b, void \*aux UNUSED); // **(2)**  static void thread\_compare (void); // **(3)**  void block\_init (void); // **(4)**  struct list alarms; // **(5)**  struct alarm { // **(6)**  int64\_t expiration; // **(7)**  struct thread \*th; // **(8)**  struct list\_elem elem; // **(9)**  }; |

**<pintos/src/devices/timer.c>**

🡪 소스 코드 분석

(1) 아무때나 thread\_compare() 를 실행하지 않도록 하기 위한 변수 설정

(2) list\_insert\_ordered 함수를 이용하기 위한 bool값 선언

(3) 리스트에 있는 스레드와 현재 시간을 비교, 만료된 스레드는 unblock하는 함수 선언

(4) 생성한 리스트를 초기화하는 함수 선언

(5) 스레드를 매달기 위한 리스트 선언

(6) 스레드의 내용을 저장할 구조체 선언

(7) 만료 시간 저장

(8) 알람을 요청한 스레드 저장

(9) 알람 리스트 연결 필드

- timer\_sleep 함수 개선

|  |
| --- |
| void  timer\_sleep (int64\_t ticks) // **(1)**  {  int64\_t start = timer\_ticks (); // **(2)**  int64\_t end = start + ticks; // **(3)**  struct alarm \*t; // **(4)**  t = (struct alarm\*)malloc (sizeof(struct alarm)); // **(5)**  t -> th = thread\_current ();// **(6)**  t -> expiration = end; // **(7)**  enum intr\_level old\_level; // **(8)**  old\_level = intr\_disable ();// **(9)**  list\_insert\_ordered (&alarms, &t -> elem, less\_arr, (void\*)NULL); // **(10)**  thread\_block(); // **(11)**  intr\_set\_level (old\_level); // **(12)**  } |

**<pintos/src/devices/timer.c>**

🡪 소스 코드 분석

(1) ticks : 타이머 인터럽트 발생 횟수를 지정하여 얼만큼 지연시킬지 파라미터로 넘겨 받음

(2) 현재 시각을 불러와서 start에 저장한다.

(3) 스레드 시작 시간에 sleep되어야 할 시간을 더해서 end에 저장한다.

(4) alarm 구조체를 가리키는 포인터 t를 생성한다.

(5) 구조체의 초기 메모리 할당을 선언한다.

(6) 현재 스레드를 참조한다.

(7) 스레드의 만료 시간을 참조한다.

(8) 인터럽트를 제어하기 위한 선언

(9) thread\_block() 함수를 위해 인터럽트를 끈다.

(10) alarms 리스트에 스레드를 매단다. (less\_alarm 방식으로 정렬함)

(11) 스레드를 block 시킨다.

(12) 인터럽트를 다시 활성화시킨다.

- less\_arr() 함수 생성

|  |
| --- |
| bool less\_arr (const struct list\_elem \*a, const struct list\_elem \*b, void \*aux UNUSED) { // **(1)**  struct alarm \*t1 = list\_entry (a, struct alarm, elem); // **(2)**  struct alarm \*t2 = list\_entry (b, struct alarm, elem);  if(t1 -> expiration >= t2 -> expiration) // **(3)**  return false;  else  return true;  } |

**<pintos/src/devices/timer.c>**

🡪 소스 코드 분석

(1) 리스트에 매달 때 만료 시간이 작은 순서대로 넣기 위한 함수

(2) 두 값을 비교하기 위해 각각의 스레드의 alarm 구조체를 선언한다.

(3) 스레드의 종료 시간을 비교하여 오름차순으로 정렬한다.

- thread\_compare() 함수 생성

|  |
| --- |
| void thread\_compare (void) { // **(1)**  if(alarm\_switch == true && list\_size(&alarms) != 0) { // **(2)**  struct alarm \*t; // **(3)**  enum intr\_level old\_level; // **(4)**  t = list\_entry (list\_front (&alarms), struct alarm, elem); // **(5)**  if((t -> expiration) <= timer\_ticks ()) { // **(6)**  old\_level = intr\_disable ();// **(7)**  list\_pop\_front (&alarms); // **(8)**  thread\_unblock ( t -> th); // **(9)**  intr\_set\_level (old\_level); // **(10)**  }  }  } |

**<pintos/src/devices/timer.c>**

🡪 소스 코드 분석

(1) 리스트의 첫 번째 스레드와 시간을 비교하여 시간이 지나면 스레드를 unblock 시키는 함수

(2) 리스트의 초기화가 되었고, 사이즈가 0일 아닐 경우에만 실행한다.

(3) alarm 구조체를 선언한다.

(4) 인터럽트를 제어하기 위해 선언한다.

(5) 리스트의 첫 번째, 만료 시간이 가장 작은 스레드를 불러온다.

(6) 불러온 스레드의 종료 시간과 현재 시간을 비교한다.

(7) 인터럽트를 비활성화시킨다.

(8) 첫 번째 스레드를 리스트에서 꺼낸다.

(9) 꺼낸 스레드를 unblock 시킨다.

(10) 인터럽트를 다시 활성화시킨다.

- block\_init() 함수 개선

|  |
| --- |
| void block\_init () { // **(1)**  list\_init (&alarms); // **(2)**  alarm\_switch = true; // **(3)**  } |

**<pintos/src/devices/timer.c>**

🡪 소스 코드 분석

(1) 생성한 리스트를 초기화한다.

(2) list로 선언되어 있는 alarms 리스트를 초기화한다.

(3) thread\_compare() 함수를 사용하기 위해 true로 변경한다.

- timer\_interrupt() 함수 개선

|  |
| --- |
| static void  timer\_interrupt (struct intr\_frame \*args UNUSED)  {  ticks++; // **(1)**  thread\_compare (); // **(2)**  thread\_tick ();  } |

**<pintos/src/devices/timer.c>**

🡪 소스 코드 분석

(1) 인터럽트 한 번 발생 시마다 ticks를 증가시킨다.

(2) 매 tick마다 리스트의 첫 번째 스레드와 현재 시간을 비교한다.

(b) /pintos/src/threads/init.c 수정

- run\_actions() 함수 개선

|  |
| --- |
| static void  run\_actions (char \*\*argv)  {  block\_init (); // **(1)**  struct action  {  char \*name;  int argc;  void (\*function) (char \*\*argv);  };  … (생략) |

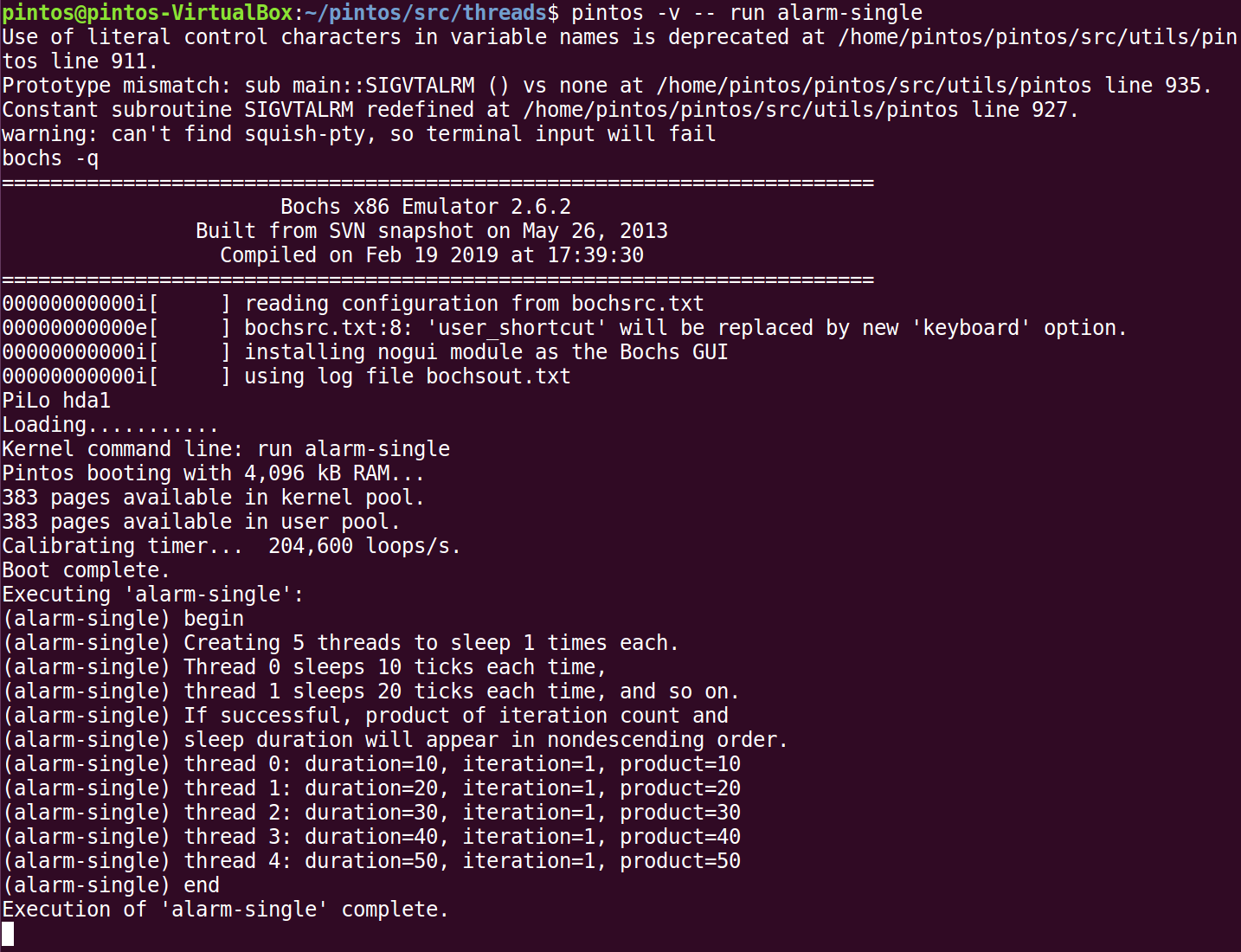
**<pintos/src/threads/init.c>**

🡪 소스 코드 분석

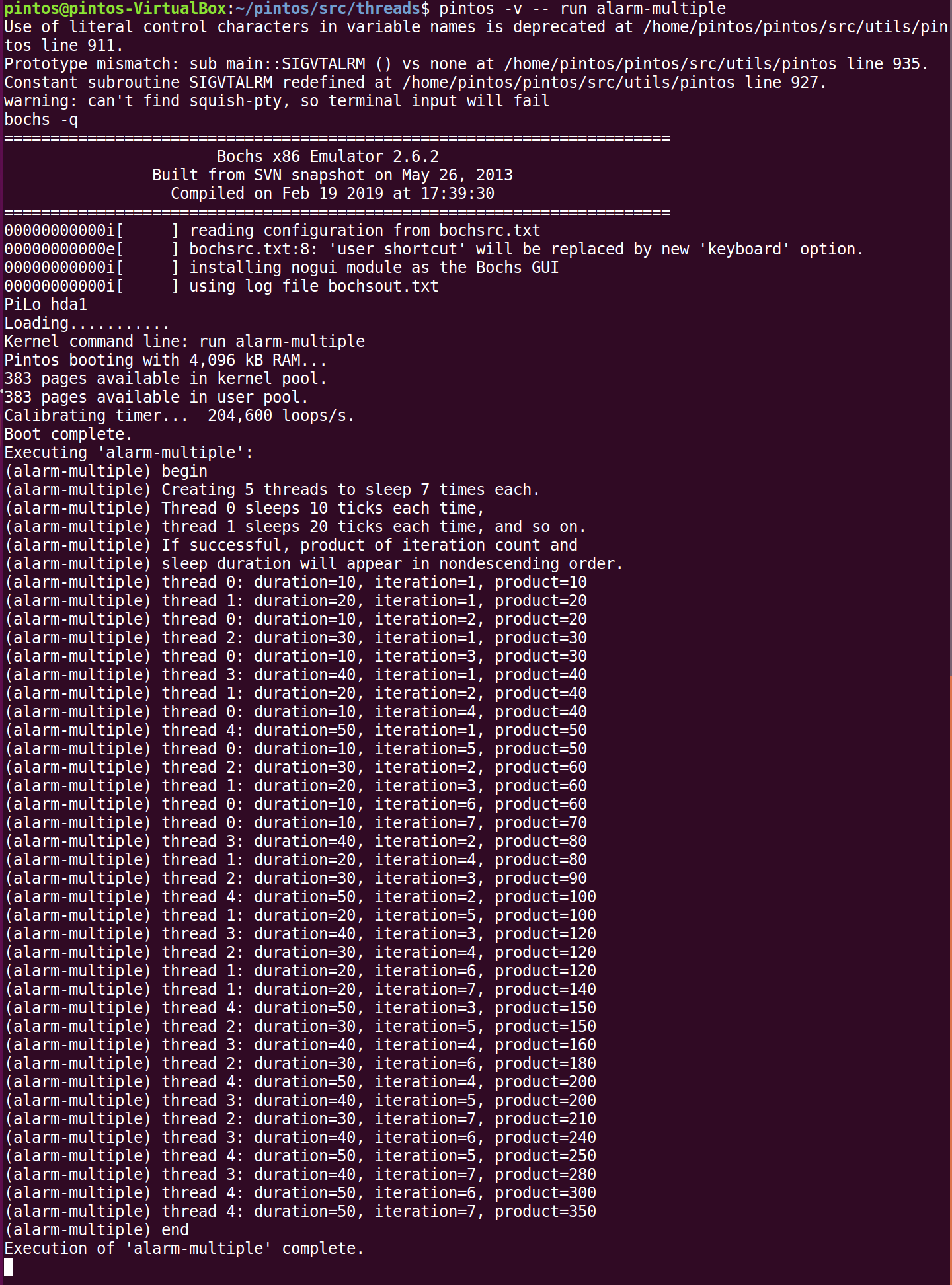
(1) 생성한 리스트를 초기화한다.

4. 테스트

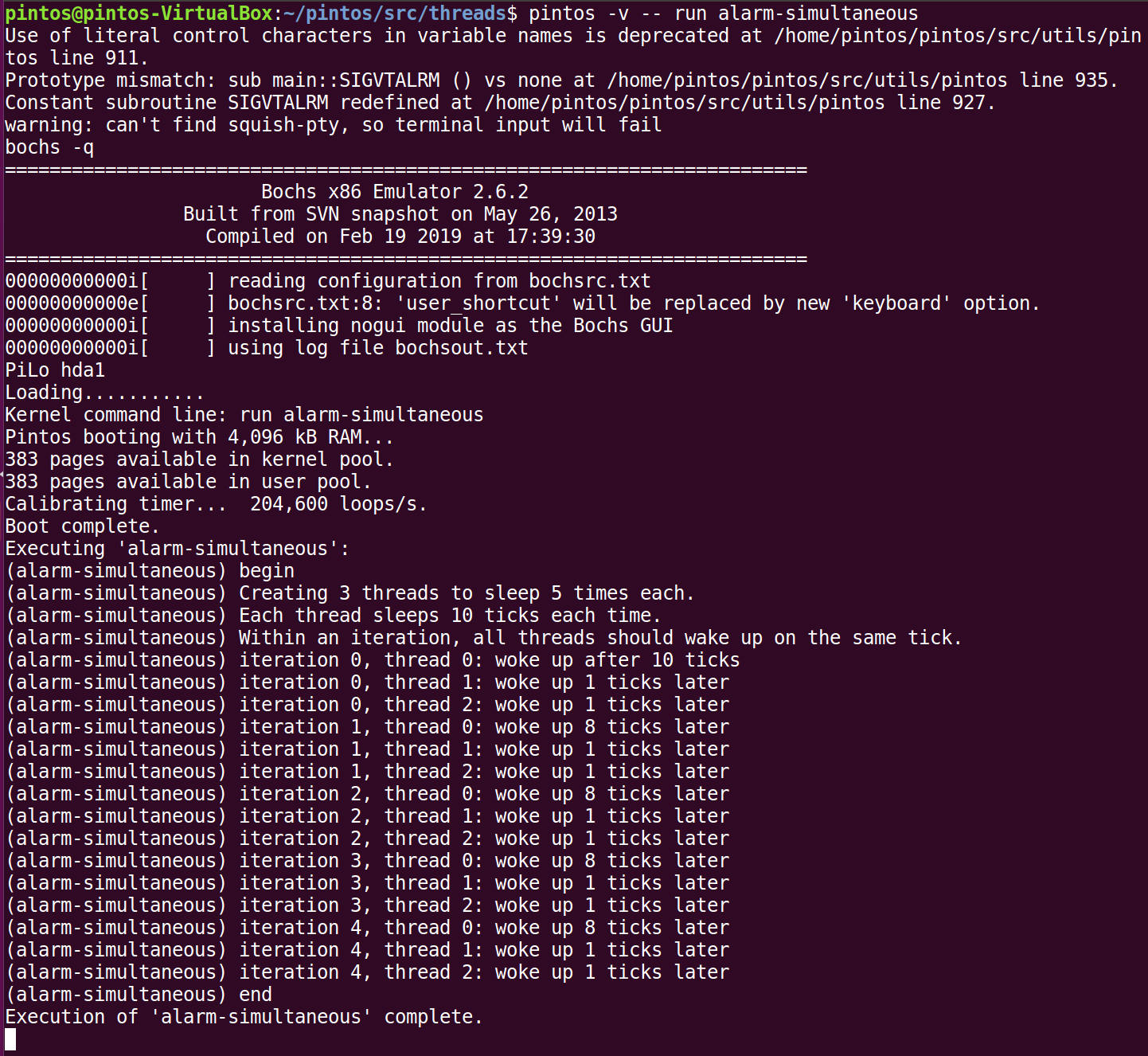
(a) “pintos -v -- run alarm-single” 실행



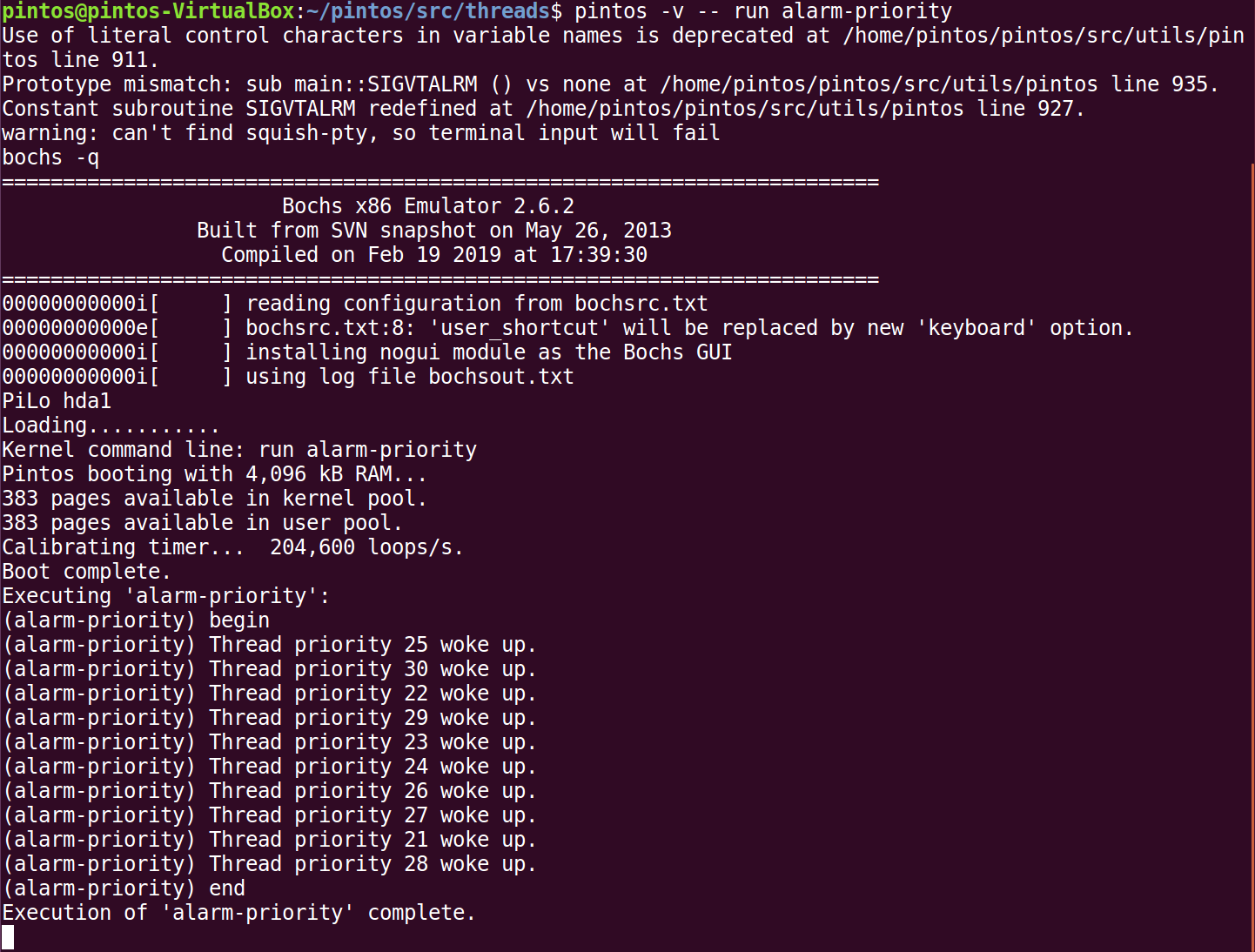
(b) “pintos -v -- run alarm-multiple” 실행



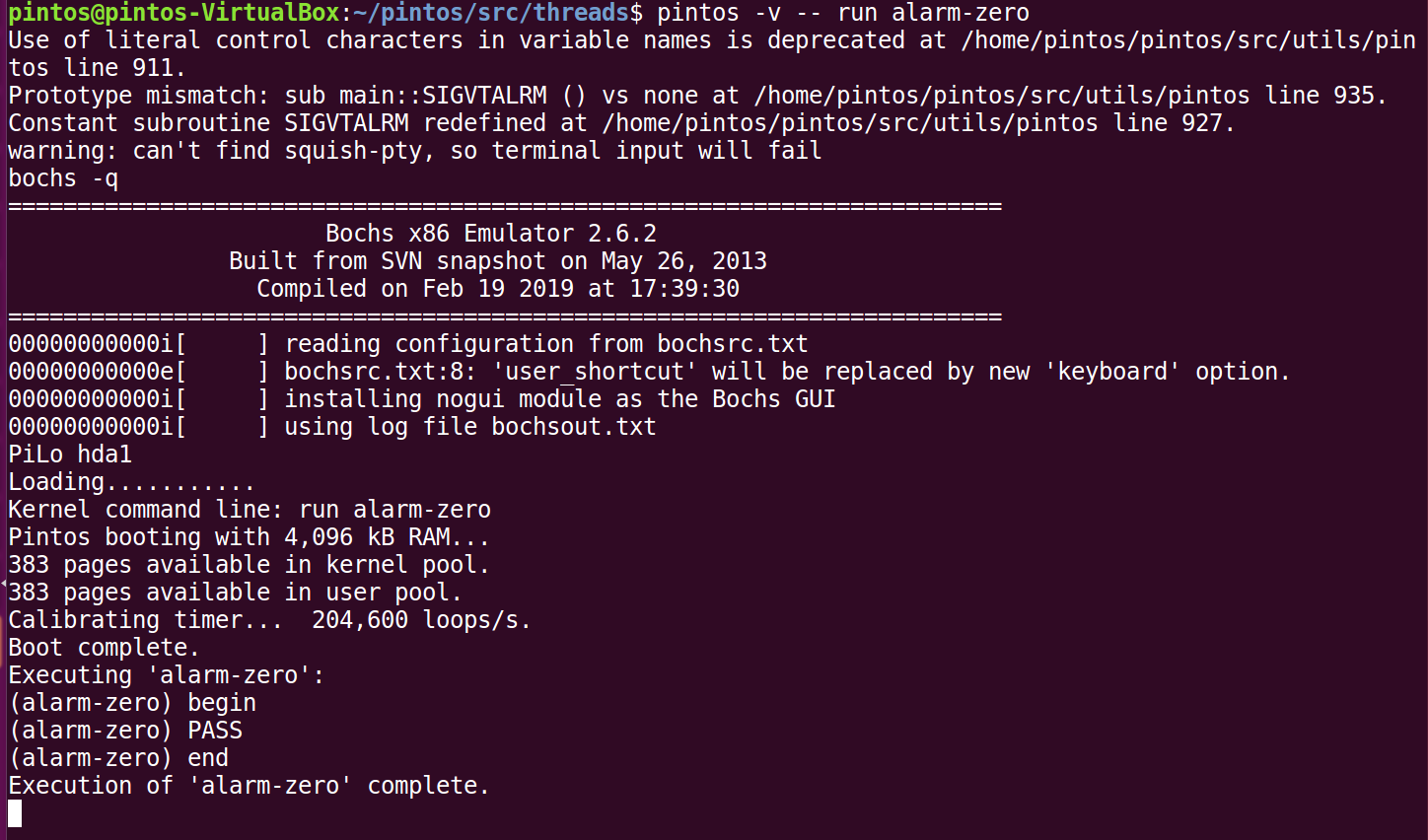
(c) “pintos -v -- run alarm-simultaneous” 실행



(d) “pintos -v -- run alarm-priority” 실행



(e) “pintos -v -- run alarm-zero” 실행



(f) “pintos -v -- run alarm-negative” 실행

