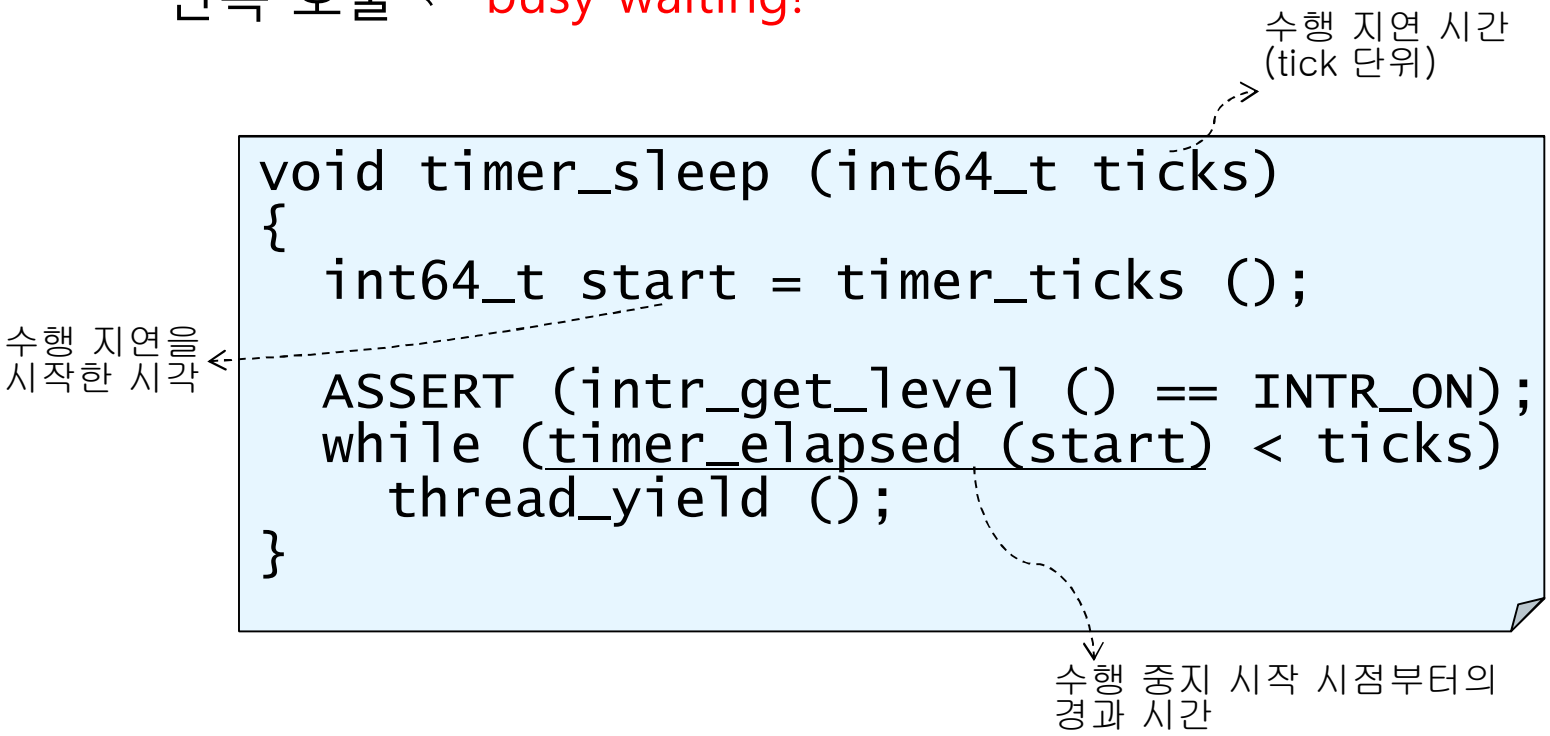
Project 1: Threads

손성훈

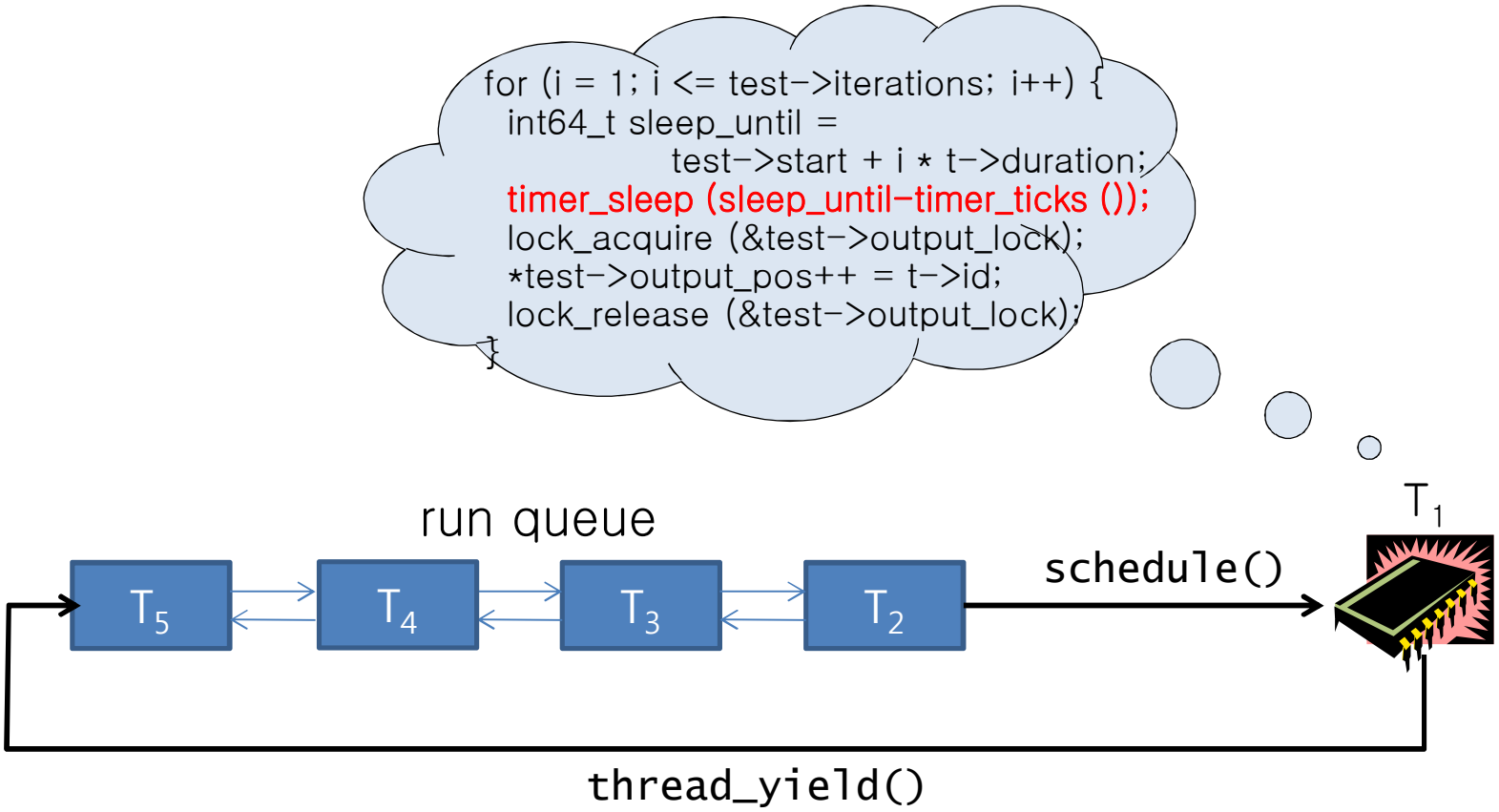
1. ALARM CLOCK 개선

Alarm Clock

- `void timer_sleep (int64_t ticks);`
  - 이를 호출한 쓰레드의 수행을 호출 시점부터 `ticks` 틱 동안 지연하는데 사용
  - 현재의 구현: `ticks` 틱이 경과할 때까지 `thread_yield()` 를 반복 호출 ← busy waiting!



현재 alarm clock 동작 – alarm-multiple의 예



thread_block() and thread_unblock()

```
/* This function must be
   called with interrupts
   turned off */

Void thread_block (void)
{
  ASSERT (!intr_context ());
  ASSERT (intr_get_level ()
          == INTR_OFF);

  thread_current()->status
    = THREAD_BLOCKED;
  schedule ();
}
```

```
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);
}
```

Semaphore – thread_block() / thread_unblock()과 wait queue의 예

```
struct semaphore
{
  /* Current value. */
  unsigned value;
  /* List of waiting threads. */
  struct list waiters;
};
```

```
struct thread
{
  …

  struct list_elem elem;
  …
};
```

```
void sema_down (struct semaphore *sema)
{
  …

  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);
}
```

/* wait(S) or P(S) */

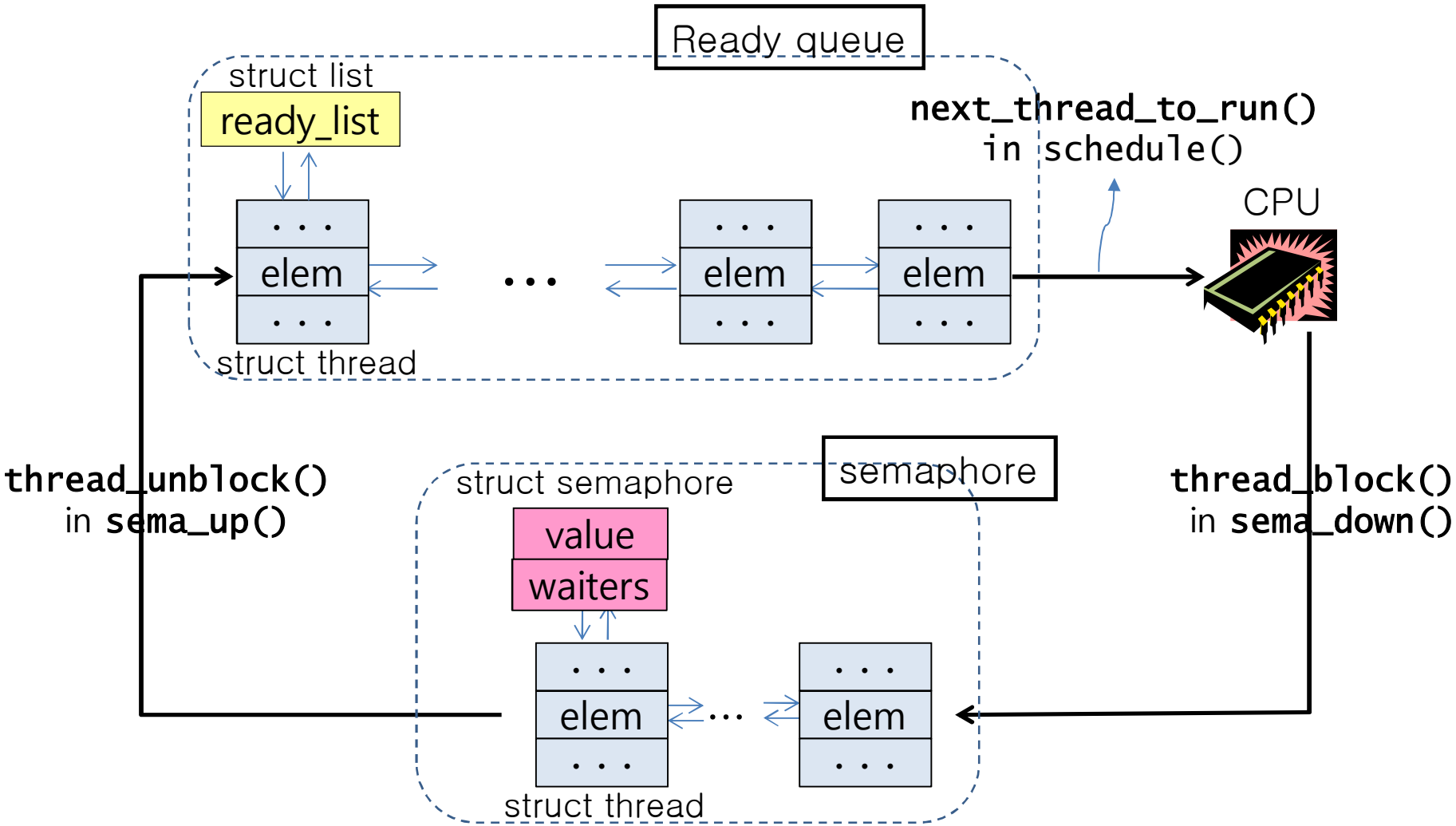
```
void sema_up (struct semaphore *sema)
{
  enum intr_level old_level;

  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);
}
```

/* signal(S) or V(S) */

Semaphore – thread_block() / thread_unblock()과 list 사용 예 (cont'd)



Alarm Clock의 개선 – No Busy-wait Alarm Clock

- 자료구조
  - 스레드들이 요청한 알람을 유지하는 linked list
  - 알람을 위한 구조체의 예

```
struct list alarms;

struct alarm
{
  int64_t expiration; // 알람 만료 시간
  struct thread *th;  // 알람 요청한 스레드
  struct list_elem elem;  // 알람 리스트 연결 필드
  …
};
```

Alarm Clock의 개선 – No Busy-wait Alarm Clock (cont'd)

- `timer_sleep()` 에서는
  - 알람을 해당 스레드와 함께 위 리스트에 매단 후,
  - `thread_block()`을 호출, 해당 스레드를 블록시킴

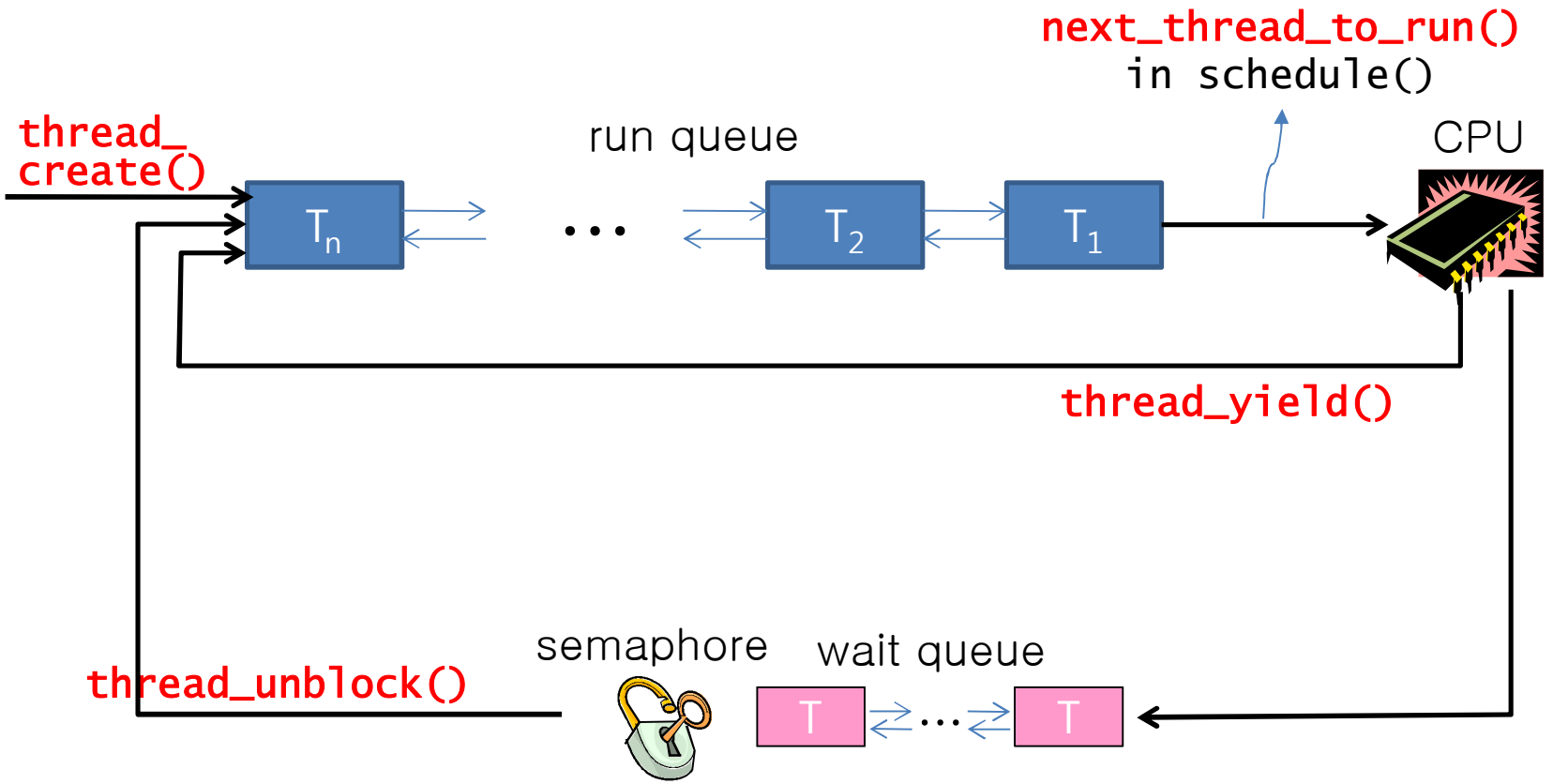
- 타이머 인터럽트 핸들러 `timer_handler()` 에서는
  - 매 틱 발생 시마다 인터럽트 처리 과정에서 위 리스트 중 시간이 만료된 알람이 있는지 검사하고,
  - 만료된 알람이 있는 경우, 알람은 리스트에서 제거하고 해당 스레드는 `thread_unblock()`을 호출, 스레드를 깨움

2. PRIORITY SCHEDULER

Round-robin Scheduling

- Ready queue의 초기화
  - thread_init()에서 초기화
- 스레드가 ready queue에 들어가는 경우
  - 최초 스레드 생성 시 thread_create()에서 thread_unblock() 호출
  - sema_up()에서 thread_unblock() 호출
  - Time slice 만료 시 또는 timer_sleep() 함수 등에서 thread_yield() 호출
- 스레드가 ready queue에서 나가는 경우
  - 스케줄러 schedule()에서 next_thread_to_run() 호출

Round-robin Scheduling (cont'd)



Priority Scheduling

- Priority
  - `struct thread`의 `priority` 필드; 현재는 사용되지 않고 있음
  - 스레드 생성 시 `thread_create()`의 인자로 주어짐
  - `PRI_MIN` (0) ~ `PRI_MAX` (63) (default 값은 `PRI_DEFAULT` (32) 임)

1. Preemptive dynamic priority scheduling
   - Ready queue에서 priority를 기준으로 선택; lock, semaphore, condition variable에서 기다릴 때도 priority 기준으로 깨움 (priority 값이 같은 스레드가 여럿인 경우에는 RR)
   - Preemptive : 현재 수행 중인 스레드보다 높은 priority를 가지는 스레드가 ready queue에 들어오면 preempt
   - Dynamic : 스레드 동작 중 priority 값이 변경될 수 있음
2. Prioirty inversion 현상의 해결: priority inheritance (donation)
3. 스레드가 자신의 priority 값을 얻거나 변경하는 인터페이스
   - `void thread_set_priority (int new_priority)`
   - `int thread_get_priority (void)`

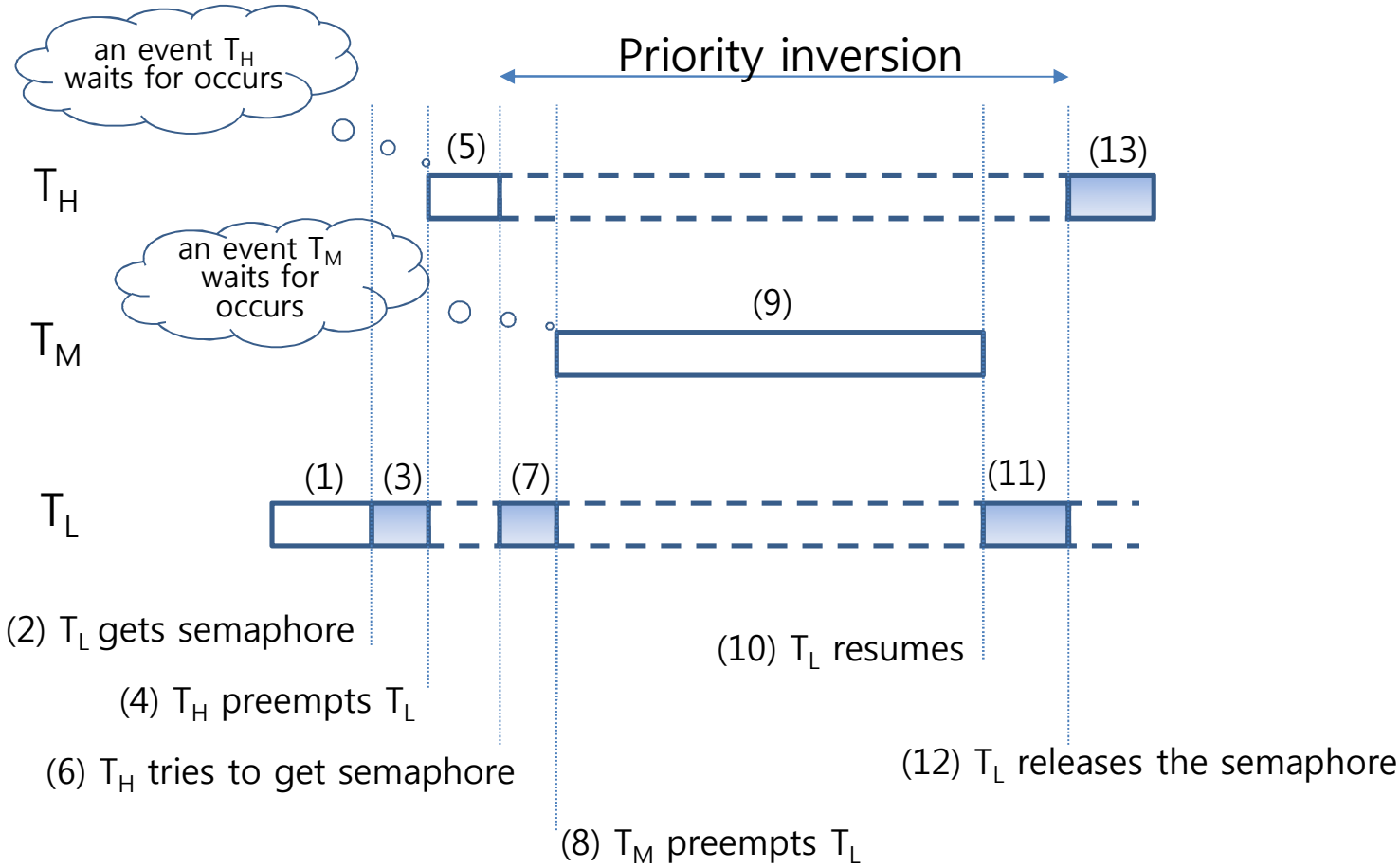
Priority Scheduling 구현

- Ready queue의 운영
  - Ready queue를 스레드의 priority 순으로 유지
  - priority 값이 같은 스레드들이 여럿인 경우 ready queue에 도착한 순서로 유지
- Preemption 시점의 변경
  - RR의 'time slice 만료 시 preempt' 제거
  - 대신 스레드가 ready queue에 들어오는 경우, 현재 running 중인 thread와 priority를 비교, 그 결과에 따라 preempt 결정
- Wait queue의 운영
  - Lock, semaphore, cond. Variable의 wait queue도 priority 순으로 유지

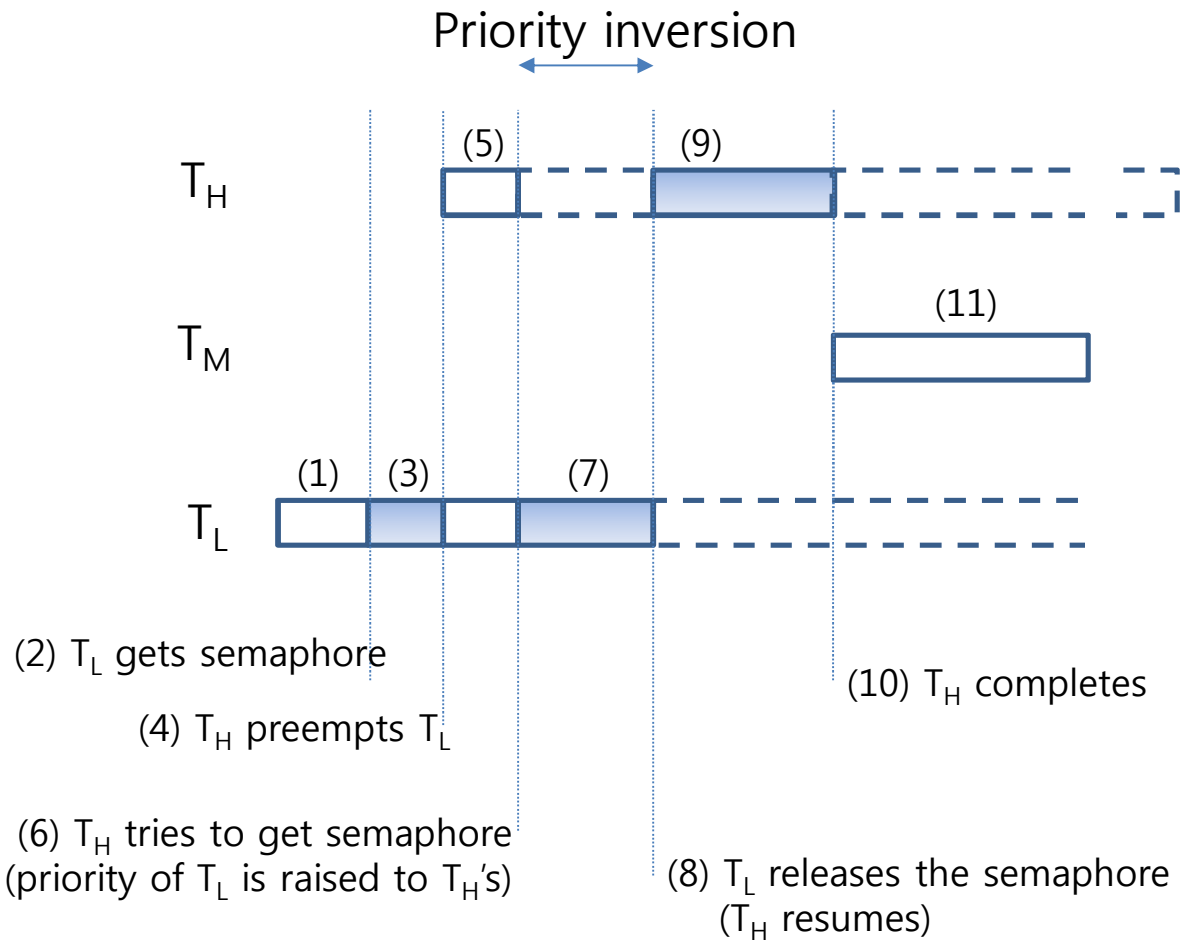
Priority Donation 구현

- Priority inheritance (donation) – priority inversion 의 해결책
  - 높은 우선순위 스레드 T_H(우선순위 값 H)가 낮은 우선순위 스레드 T_L(우선순위 값 L, L < H)이 이미 가지고 있는 lock을 요청하는 경우, T_L의 priority를 일시적으로 T_H와 같아지도록 높여줌 (즉, T_L의 우선순위 값을 임시로 H로 설정)
  - 이후 T_L 이 lock을 해제하는 순간, T_L의 우선순위를 원래의 우선순위 값 L로 환원; 이후 T_H이 lock을 가지게 됨
- Multiple donation의 처리
  - T_L가 T_{H1} 과 T_{H2} (단 우선 순위는 T_{H1} < T_{H2})로부터 donation 받는 경우
- Nested donation의 처리
  - T_M 이 T_H로부터 donation 받고, 다시 T_L 이 T_M 로부터 donation 받는 경우
- Priority inheritance 는 lock, semaphore, … 중 semaphore에 대해서만 구현하면 됨

Priority Inversion



Priority Donation



Priority 조작 인터페이스 구현

- 스레드가 자신의 priority 값을 얻거나 변경하는 인터페이스
  - `void thread_set_priority (int new_priority)`
  - `int thread_get_priority (void)`
  - 주) priority scheduling에서는 (blocked 상태의 thread들을 제외하면) 항상 running thread의 우선순위가 가장 높다; 이 호출은 항상 현재 수행 중인 running thread가 호출하며, 특히 `thread_set_priority()`를 호출하는 경우, running thread의 priority 값이 바뀔 수도 있음을 주의할 것

- `int thread_get_priority (void)`
  - Running thread의 priority 값을 리턴

- `void thread_set_priority (int new_priority)`
  - Running thread의 priority 값을 변경
  - Prioiry 가 낮아지는 경우 context switch 가 발생할 수도 있음

LINKED LIST IN PINTOS

Linked list in Pintos

- Doubly linked list implementation in Pintos
  - lib/kernel/list.h, lib/kernel/list.c /* 코드에 포함된 커멘트들을 반드시 읽어볼 것 */
  - Ready queue 나 semaphore의 wait queue 등의 구현에 사용
- Declaration

```
struct foo {
    struct list_elem elem;
    int bar;
    ...other members...
};
```

- Initialization

```
struct list foo_list;
list_init (&foo_list);
```

- Eg) Iteration

```
struct list_elem *e;
for (e = list_begin (&foo_list);
  e != list_end (&foo_list);
  e = list_next (e)) {
    struct foo *f = list_entry (e, struct foo, elem);
    ...do something with f...
}
```

List Operations

- List initialization

```
void list_init (struct list *);
```

- List traversal

```
struct list_elem *list_begin (struct list *);
struct list_elem *list_next (struct list_elem *);
struct list_elem *list_end (struct list *);

struct list_elem *list_rbegin (struct list *);
struct list_elem *list_prev (struct list_elem *);
struct list_elem *list_rend (struct list *);

struct list_elem *list_head (struct list *);
struct list_elem *list_tail (struct list *);
```

- List insertion

```
void list_insert (struct list_elem *,
                  struct list_elem *);
void list_splice (struct list_elem *before,
                  struct list_elem *first,
                  struct list_elem *last);
void list_push_front (struct list *, struct
list_elem *);
void list_push_back (struct list *, struct
list_elem *);
```

- List removal

```
struct list_elem *list_remove (struct list_elem *);
struct list_elem *list_pop_front (struct list *);
struct list_elem *list_pop_back (struct list *);
```

- List elements

```
struct list_elem *list_front (struct list *);
struct list_elem *list_back (struct list *);
```

- List properties

```
size_t list_size (struct list *);
bool list_empty (struct list *);
```

- Miscellaneous

```
void list_reverse (struct list *);
```

- Compares the value of two list elements

```
typedef bool list_less_func (const struct
list_elem *a, const struct list_elem *b, void
*aux);
```

- Operations on lists with ordered elements

```
void list_sort (struct list *, list_less_func *,
void *aux);
void list_insert_ordered (struct list *, struct
list_elem *, list_less_func *, void *aux);
void list_unique (struct list *, struct list
*duplicates,list_less_func *, void *aux);
```

- Max and min

```
struct list_elem *list_max (struct list *,
list_less_func *, void *aux);
struct list_elem *list_min (struct list *,
list_less_func *, void *aux);
```