The Design of *OrionOS* Operating System

Students' Name

November 5, 2013

Abstract

A short description of this report content.

Take into account that a design document is a complete high-level solution to the problem presented. It should be detailed enough that somebody who already understands the problem could go out and code the project without having to make any significant decisions. Further, if this somebody happens to be an experienced coder, they should be able to use the design document to code the solution in a few hours (not necessarily including debugging).

Chapter 1

General Presentation

1.1 Working Team

1. Pană Alexandru
   (a) Threads: dealt with designing
   (b) Threads: dealt with BBB
   (c) Threads: dealt with CCC
2. Soucup Adrian
   (a) Threads: dealt with designing
   (b) Threads: dealt with BBB
   (c) Threads: dealt with CCC
3. Vultur Horațiu
   (a) Threads: dealt with designing
   (b) Threads: dealt with BBB
   (c) Threads: dealt with CCC
4. Zene Andrei
   (a) Threads: dealt with designing

(b) Threads: dealt with BBB

(c) Threads: dealt with CCC

Chapter 2

Design of Module *threads*

2.1 Alarm clock

2.1.1 Initial Functionality

At the beginning of this project the function *sleep_timer* is implemented as a busy wait. We want to reimplement it to avoid the busy wait.

2.1.2 Data Structures and Functions

```
struct thread
{

      ...
  /* the time (in number of ticks) at which a
      sleeping thread should wake up */
      int64_t wakeup_time;
      ...

};

/* a sorted list containing all sleeping
(blocked) threads. Sorting is done using
thread_wakeup_time_comparison function*/
```

```
static struct list sleep_list;

/* comparison function to order the sleeping
threads ascending by their wakeup time */
list_less_func thread_wakeup_time_comparison;

/* sets the time at which the thread should
wake up and puts the thread in the sleeping
threads list */
void thread_sleep (int64_t wakeup_time);

/*this function checks if there are sleeping
threads that should wake up at this moment
and calls the function thread_unblock for
each of those threads.*/
void handle_sleeping_threads();
```

2.1.3 Functionality

When the function *sleep_timer* is called, this function calculates the time (in ticks) at which the thread should wake up, sets this time in the thread structure, inserts the thread in a sleeping list and then calls *thread_block*, in order to set the status THREAD_BLOCKED, and to call the scheduler. The sleeping list is sorted ascending according to the time at which the threads should wake up (the first thread in this list is the thread that should wake up the earliest). At every timer interrupt, the function *handle_sleeping_threads* is called which checks if there are threads that can wake up at the current moment of time. If there are, these threads are removed from the sleeping list, and *thread_unblock* is called on them, which sets their status to THREAD_READY and puts them in the ready list.

In order to avoid race conditions, the interrupts are disabled during the execution of *sleep_timer* and *timer_interrupt*.

2.1.4 Design Decisions

This solution has the advantage that the time spent in the *timer_interrupt* function is constant, because the list is ordered. On the other hand the time spent for inserting is O(n), because we use a list to keep the threads. A better solution would be to use a heap as a data structure to keep the sleeping threads. In this way, the insertion timewould be only O(log n).

However, this solution is better than the solution in which at every *timer_interrupt* the whole sleeping list is traversed to see if there is a thread that should wake up even though the insertion is done in constant time, (without sorting the list) because *timer_interrupt* is called more often than the function *sleep_timer*.

2.1.5 Tests

alarm-single, alarm-multiple, alarm-negative, alarm-simultaneous, alarm-zero

2.2 Priority scheduler

2.2.1 Initial Functionality

At the beginning of this project the scheduler is implented as a Round-Robin scheduler. We want to reimplement it as a priority scheduler.

2.2.2 Data Structures and Functions

```
struct thread
{
        ...
    /*the fixed priority of the thread,
    given at creation */
    int priority;

    /* the priority at a given moment of
    the thread. This can be either the
    thread's fixed priority or the
    priority inherited by donation */
    int current_priority;
      ...
};

/* ready_list[i] contains the threads of
priority i having the status THREAD_READY. */
static struct list ready_list[PRI_MAX + 1];

/* the priority scheduler */
static struct thread * next_thread_to_run (void);

/* promotes a thread to current thread's priority */
void thread_promote (struct thread *);

/* forces the current thread to return to it's
default fixed priority. */
void thread_lessen ();

/* reimplementation of thread_unblock which checks
if the new READY thread is more prioritary than the
current thread, and if so, it forces current thread
to yield the cpu. */
void thread_unblock (struct thread *);

struct lock
{
    struct thread *holder;    /* Thread holding lock */
    unsigned value;           /* Current value. */
    struct list waiters;      /* List of waiting threads. */
};

/* reimplementation of old lock functions according to
the new structure of the lock */
```

```
void lock_init (struct lock *);
void lock_acquire (struct lock *);
bool lock_held_by_current_thread (const struct lock *);

/* reimplementation of old lock_acquire, with priority
donation. When the lock is hold by a less prioritary
thread, that thread is promoted to current thread's
priority. */
bool lock_try_acquire (struct lock *);

/* reimplementation of old lock_release, with priority
donation. If current thread was promoted, it gives up
to it's inherited priority after releasing the lock. */
void lock_release (struct lock *);
```

2.2.3 Functionality

The ready list is organized as a vector of lists, in which, every list contains threads that have the same priority. The scheduler calls the function *next_thread_to_run* which pops the most prioritary thread from its list and returns it; if there are more threads with the same priority, a round-robin algorithm is used. In order to avoid priority inversion, the *lock_aquire* is rewritten, so that whenever a thread with a greater priority waits for a lock holded by a thread with a lower priority, the function *lock_aquire* calls the function *thread_promote (holder)*. The function *thread_promote* sets the new current priority for the holder to the priority of the current thread and then moves the thread in the ready lists vector from its list to the beginning of the list corresponding to its new priority. At the next call of the scheduler, the thread that holds the lock will receive cpu, and will be able to release the lock. In order to bring things back to previous situation, the function *lock_release* is rewritten, so that it calls the function *thread_lessen*, which forces the current thread to go back to its fixed priority, and to yield the cpu. At the call of *thread_yield* the next thread will put itself in the ready list corresponding to its previous priority.

To be able to give the cpu to a new more prioritary thread when it occurs, the *thread_unblock* function is rewritten to do this test, and force current thread to yield the cpu if necessary.

In order to avoid race conditions, the interrupts are disabled during the execution of *thread_promote* and *thread_lessen*, and of course during the execution of functions where it was previously disabled.

2.2.4 Design Decisions

why promoting/lessening is done in *lock_aquire* / *lock_release*?

why vector of lists?

2.2.5 Tests

alarm-priority, priority-change, priority-condvar, priority-donate-chain, priority-donate-lower, priority-donate-multiple, priority-donate-nest, priority-donate-one, priority-donate-sema, priority-fifo, priority-preempt, priority-sema