The Design of *OrionOS* Operating System

Pană, A., Soucup A., Vultur H., Zene, A.

December 3, 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 synchronisation mechanisms implementation & priority donation
   (c) Threads: dealt with file system syscalls
2. Soucup Adrian
   (a) Threads: dealt with designing
   (b) Threads: dealt with advanced scheduler
   (c) Threads: dealt with wait syscall
3. Vultur Horațiu
   (a) Threads: dealt with designing
   (b) Threads: dealt with priority scheduler
   (c) Threads: dealt with argument passing, process termination messages & exec syscall
4. Zene Andrei

(a) Threads: dealt with designing

(b) Threads: dealt with alarm clock & fixed-point arithmetic

(c) Threads: dealt with exit syscall, process data structures & denying writes to executables

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

```
"In␣thread.c/thread.h:"
  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-negative
Source: tests/threads/alarm-negative.c
Purpose: Checks that the OS doesn't crash when a thread is put to sleep for a negative ammount of ticks.
Description: The current thread is put to sleep for -100 ticks.

alarm-zero
Source: tests/threads/alarm-zero.c *Purpose:* Checks that the OS doesn't put the thread to sleep at all if timer_sleep is called with a value of 0 ticks.
Description: The current thread is put to sleep for 0 ticks. No checks are made.

Solution: In function timer_sleep, do nothing if sleep_time is $<=$ 0.

alarm-simultaneous
Source: tests/threads/alarm-simultaneous.c
Purpose: Creates N threads, each of which sleeps a different, fixed duration, M times. Records the wake-up order and verifies that it is valid.
Description: Each thread is put to sleep 5 times and scheduled to wake up at the same time as all the other threads. The test records the wake times for each iteration relative to the first thread. The first thread must wake up

10 ticks later (relative to it's own previous wakeup time). The rest of the treads must wake up during the same tick as the first (0 ticks relative to the first).

alarm-multiple

Source: tests/threads/alarm-wait.c

Purpose: Creates N threads, each of which sleeps a different, fixed duration, M times. Records the wake-up order and verifies that it is valid.

Description: Starts 5 threads, each sleeping for thread_id * 10 ticks each iteration for 7 iterations. The wakeup time for each thread is given by sleep_duration * iteration for each iteration. The test checks that the wake up times are in ascending order (the threads did indeed wake up at the right times).

alarm-single

Source: tests/threads/alarm-wait.c

Purpose: Creates N threads, each of which sleeps a different, fixed duration, M times. Records the wake-up order and verifies that it is valid.

Description: Uses the same test as alarm-multiple, but with only one iteration per thread.

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

```
"In thread.c/thread.h:"
  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 ();

/* checks if new priority is not the higest priority
anymore and if so yields the cpu. */
void thread_set_priority(int);

"In␣synch.c/synch.h:"
/* 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? Another idea is to handle this in thread_tick procedure, but we found out that it's generally a bad idea to load the interrupt handling code. If we would choose to do that the running time on each thread_tick interrupt could degenerate to O(n) where n is the lock count of the system. So for each thread that is running and waiting for a lock to edit the lock holder's priority and let the schedule decide. This also solves the locking stack problem.

Why vector of lists? O(1) insertion/removal. We could alternatively use sorted data structures.

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.

2.3 Advanced scheduler

2.3.1 Initial Functionality

At the beginning of this project the scheduler is implented as a priority scheduler. We want to reimplement it as an advanced scheduler(4.4BSD).

2.3.2 Data Structures and Functions

```
"In␣thread.c:"
  struct thread
  {
        ...
      /* */
      int nice;
      /* */
      int64_t recent_cpu;
        ...
  };

  /* load average of the whole system */
  int64_t load_avg;

/* add a new function that recompute the new priority
    if the priority has changed than the thread is
    pop from the thread and inserted again with the
    new priority */
    void thread_recompute_priority( struct thread );


  /* create new files fixed_point.c and fixed_point.h*/

"In␣fixed_point.h/fixed_point.c:"
  /* add the following header functions in the file
```

```
fixed_point.h. These function represents operation
between fixed-points and integer values. The
implementation of the files are in the file
fixed_point.c.*/
int64_t fp_from_int(int n);
int fp_to_int_rz(int64_t fp);
int fp_to_int_rn(int64_t fp);
int64_t fp_add(int64_t x, int64_t y);
int64_t fp_subtract(int64_t x, int64_t y);
int64_t fp_mult(int64_t x, int64_t y);
int64_t fp_div(int64_t x, int64_t y);
int64_t fp_int_add(int64_t x, int y);
int64_t fp_int_subtract(int64_t x, int y);
int64_t fp_int_mult(int64_t x, int y);
int64_t fp_int_div(int64_t x, int y);
```

```
"In timer.c:"
  /* modify the function timer_interrupt in which we
    we calculate for each the recent_cpu using the
    function thread_for_each */
    void timer_interrupt();
```

2.3.3 Functionality

The timer_interrupt function modification:

proc timer_interrupt() $\equiv$

$recent\_cpu[running\_thread] := recent\_cpu[running\_thread] + 1;$

if $TIMER\_FREQ \% TIMER\_TICKS \equiv 0$

then $thread\_for\_each(all\_threads\_list, thread\_recompute\_priority);$

else $thread\_recompute\_priority(running\_thread);$

fi;

end;

proc thread_recompute_priority($thread$) $\equiv$

$ready\_threads = count(ready\_list);$

Use functions from fixed-point.h lib to compute the next expressions;
$load\_avg := (59/60) * load\_avg + (1/60) * ready_threads$;
$recent\_cpu[thread] := (2 * load\_avg)/(2 * load\_avg + 1) + recent\_cpu[t] + nice[t]$;
$new\_priority := clamp(PRI\_MAX - (recent\_cpu/4) - (nice * 2))$;
$old\_priority := priority[thread]$;
if $new\_priority != old\_priority$
then $remove(ready\_list[old\_priority], thread.elem)$;
$push\_back(ready\_list[new\_priority], thread.elem)$;
fi;
end;

We keep the vector V, of queues that we used when building the priority scheduler. V[i] represents the queue with priority i. By convention when the schedule is called, the next_thread_to_run() procedure will remove the thread from the V[i].front() (where i represents the index with the highest priority) and will push it back in V[newPriority] in thread_yield() proc and thread_unblock().

It is important that computations for recent_cpu and load_avg are done with functions from fixed_point.h, because these two variables are real numbers.

In order to be able to choose between the MLFQS scheduler and the Priority Scheduler when running the tests, the thread_mlfqs flag is used. If this flag is set, the functions

- *thread_create* ignores the priority given as a parameter, and creates a thread with priority PRI_DEFAULT,
- *thread_set_priority*, does not change the priority anymore
- *lock_aquire* and *lock_release* don't do priority donation anymore

2.3.4 Design Decisions

We chose the vector of queues because of O(1) constant time retrieval and insertion. This means that the scheduler will run with the same speed regardless of the numbers of threads that are currently in the system. The choice of editing the functions thread_yield(), thread_unblock() and next_thread_to_run() is good because all the synchronization mechanisms and sleeps already depend on them so we don't need to worry about how they are implemented.

2.3.5 Tests

mlfqs-block, mlfqs-fair, mlfqs-load-1, mlfqs-load-60 mlfqs-load-avg, mlfqs-recent-1

Chapter 3

Design of Module *userprog*

3.1 Process Termination Messages & Argument passing

3.1.1 Initial Functionality

At the beginning of this project ... .

3.1.2 Data Structures and Functions

```
"In␣thread.h␣:"

  struct thread {
      /* The process id to which the thread
         belongs to. */
         int pid;
  };

  /* comments */
  variable definition;

  /* comments */
```

```
function definition;
```

3.1.3 Functionality

Function x works like this ... : (picture with sequence diagram or algorithm. See commented examples for including images or describing algorithms below)

3.1.4 Design Decisions

This solution has the advantage that ...

On the other hand it is not so good that ... .

A better solution would be ... .

However, this solution is better than ...

3.1.5 Tests

name of the test

Source: path/to/test.c

Purpose: What does it check?

Description: Short description. *Solution:(if necessary)* Here, if what the test does is something that was not explicitly asked in the requirements, it would be good to say that we already treat that situation and where. If it's something that should be solved only by the implementation of the requirements you can say: Solved by requirements fulfillment.

3.2 System calls

3.2.1 Initial Functionality

At the beginning of this project the pintOS does not provide the posibility of calling system calls from the user interface. Besides this, there is no data structure to support processes. We want to implement the data structures to support processes and filesystem syscalls and the syscalls that will allow user programs to use this functionality.

3.2.2 Data Structures and Functions

```
"In␣process.h␣:"

  /* maximum number of processes running at the same
  time. May need some adjustments. */
  #define MAX_PROCESSES 1024

  /* possible states of a process.
      ALIVE   is a process that has still not finished
              it's execution.
      KILLED  is a process killed by the kernel.
      DEAD    is a process that was killed at user's
              request by calling exit. */
  enum process_status_type {ALIVE, KILLED, DEAD}

  struct process {
      /* pid (process id) of this process's parent */
      int ppid;

      /* status of the process: one of ALIVE, KILLED,
         DEAD. */
      process_status_type status;

      /* the code returned by the process at exit */
      int exit_code;

      /* The thread that is waiting after this thread. */
      struct thread* waiting_thread;

      /* In construction ... Come back soon! */
      void *etc;
  };
```

```
"In␣process.c:␣"

/* The processes table */
static struct processes_table[MAX_PROCESSES];

"In␣userprog/syscalls.h:␣(kernel␣side)"

/* synchronisation lock in order to execute
   only one syscall at a time */
static struct lock syscall_lock;

/* Reads a byte at user virtual address UADDR.
  UADDR must be below PHYS_BASE.
  Returns the byte value if successful, -1 if a
  segfault occurred.
  PROBLEM!!! WHAT HAPPENS IF BYTE AT UADDR IS -1?*/
static int get_user (const uint8_t *uaddr);

/* Writes BYTE to user address UDST.
   UDST must be below PHYS_BASE.
   Returns true if successful, false if a segfault
   occurred. */
static bool put_user (uint8_t *udst, uint8_t byte);
```

3.2.3 Functionality

Function x works like this ... : (picture with sequence diagram or algorithm. See commented examples for including images or describing algorithms below)

Exit sycall

$syscall\_lock.aquire();$
$proc\_crt = processes\_table[current\_thread.pid];$
$proc\_crt.exit\_code = get_user(status);$
$thread\_exit();$
$proc\_crt.status = DEAD;$

if $proc\_crt.waiting\_thread! = NULL$
then $sycall\_lock.release();$
$thread\_unblock(proc\_crt.waiting_thread);$
fi

3.2.4 Design Decisions

This solution has the advantage that ...

On the other hand it is not so good that ... .

A better solution would be ... .

However, this solution is better than ...

3.2.5 Tests

name of the test
Source: path/to/test.c
Purpose: What does it check?
Description: Short description. *Solution:(if necessary)* Here, if what the test does is something that was not explicitly asked in the requirements, it would be good to say that we already treat that situation and where. If it's something that should be solved only by the implementation of the requirements you can say: Solved by requirements fulfillment.

3.3 Denying writes to executables

3.3.1 Initial Functionality

At the beginning of this project ... .

3.3.2 Data Structures and Functions

```
"In␣...␣:"
```

```
struct structure_def {
    type field;
};

/* comments */
variable definition;

/* comments */
function definition;
```

3.3.3 Functionality

Function x works like this ... : (picture with sequence diagram or algorithm. See commented examples for including images or describing algorithms below)

3.3.4 Design Decisions

This solution has the advantage that ...

On the other hand it is not so good that ... .

A better solution would be ... .

However, this solution is better than ...

3.3.5 Tests

name of the test

Source: path/to/test.c

Purpose: What does it check?

Description: Short description. *Solution:(if necessary)* Here, if what the test does is something that was not explicitly asked in the requirements, it would be good to say that we already treat that situation and where. If it's something that should be solved only by the implementation of the requirements you can say: Solved by requirements fulfillment.