report.txt

1
+---------------------+
| PROJECT : PintOS I |
|
DESIGN DOCUMENT
|
+---------------------+

Sanjoy Chowdhury <Roll no 20172123>
Ranajit Saha <Roll no 20172119>

---- PRELIMINARIES ----
>> Sources referred -
>> [1] https://web.stanford.edu/class/cs140/projects/pintos/pintos.html
>> [2] https://pintosiiith.wordpress.com/2012/09/13/install-pintos-with-
qemu/
>> [3] http://arpith.xyz/2016/01/getting-started-with-pintos/
SETUP IDIOSYNCRASIES
====================
>> [3] Ideally, QEMU should exit once PintOS powers off. It does not do
so and
>> gets stuck at "Powering Off...". To rectify the issue the following
changes
>> were made to src/devices/shutdown.c
@@ -99,6 +99,8 @@
printf ("Powering off...\n");
serial_flush ();
+
/* For ensuring proper shutdown on qemu */
+
outw(0x604, 0x0 | 0x2000);
/* This is a special power-off sequence supported by Bochs and
QEMU, but not by physical hardware. */
for (p = s; *p != '\0'; p++)
HELLO WORLD
===========
>> "hello.c" was added to the "src/tests/threads" directory
>> Contents of file "src/tests/threads/hello.c"
/* Tests hello, which should print "Hello Pintos" to console and exit */
#include
#include
#include
#include
#include
#include
<stdio.h>
"tests/threads/tests.h"
"threads/malloc.h"
"threads/synch.h"
"threads/thread.h"
"devices/timer.h"
void
test_hello (void)
{
printf("Hello Pintos\n");
pass ();
}

```
>> To ensure that "hello.c" was built with the kernel, we made changes to
>> "Make.tests", "tests.c" and "tests.h" in the "src/tests/threads"
directory
>> Changes to "Make.tests"
@@ -9,6 +9,7 @@
priority-fifo priority-preempt priority-sema priority-condvar
priority-donate-chain
\
\report.txt

2
mlfqs-load-1 mlfqs-load-60 mlfqs-load-avg mlfqs-recent-1 mlfqs-fair-2
+hello \
mlfqs-fair-20 mlfqs-nice-2 mlfqs-nice-10 mlfqs-block)
# Sources for tests.
@@ -36,6 +37,7 @@
tests/threads_SRC +=
tests/threads_SRC +=
tests/threads_SRC +=
+tests/threads_SRC +=
\
tests/threads/mlfqs-recent-1.c
tests/threads/mlfqs-fair.c
tests/threads/mlfqs-block.c
tests/threads/hello.c
MLFQS_OUTPUTS =
\
tests/threads/mlfqs-load-1.output
\
>> Changes to "tests.c"
@@ -38,6 +38,7 @@
{"mlfqs-nice-2", test_mlfqs_nice_2},
{"mlfqs-nice-10", test_mlfqs_nice_10},
{"mlfqs-block", test_mlfqs_block},
+
{"hello", test_hello},
};
static const char *test_name;
>> Changes to "tests.h"
@@ -32,6 +32,8 @@
extern test_func test_mlfqs_nice_2;
extern test_func test_mlfqs_nice_10;
extern test_func test_mlfqs_block;
+/* Prints Hello world */
+extern test_func test_hello;
void msg (const char *, ...);
void fail (const char *, ...);
ALARM CLOCK
===========
---- ADDITIONAL CHANGES ----
>> The following changes were made to "src/Makefile.build" in order to
ensure
>> that the source file "src/threads/priority.c" is build with the
kernel.
@@ -15,6 +15,7 @@
threads_SRC = threads/start.S
# Startup code.
threads_SRC += threads/init.c
# Main program.
threads_SRC += threads/thread.c
```

```
# Thread management core.
+threads_SRC += threads/pqueue.c
# Priority queue management
threads_SRC += threads/switch.S
# Thread switch routine.
threads_SRC += threads/interrupt.c # Interrupt core.
threads_SRC += threads/intr-stubs.S # Interrupt stubs.
---- DATA STRUCTURES ----
>> A generic priority-queue was implemented in "src/threads/pqueue.c"
>> (declarations in "src/threads/pqueue.h") to deal with the ordering of
>> thread wake up alarms.
>> Contents of "pqueue.h"
#ifndef PQUEUE_H_INCLUDED
#define PQUEUE_H_INCLUDED
#include <stdio.h>report.txt

3
#include <stdbool.h>
#include <inttypes.h>
#include "threads/malloc.h"
typedef struct heap64_elem_t
{
void *data;
int64_t key;
} heap64_elem_t;
typedef struct heap64_t
{
heap64_elem_t *heap;
int n_elem;
int max_nelem;
} heap64_t;
extern heap64_t *
heap64_init (heap64_t *h, int max_nelem);
extern bool
heap64_is_empty (heap64_t *h);
extern bool
heap64_is_full (heap64_t *h);
#define HEAP_PARENT(i) ((i)>>1)
#define HEAP_LCHILD(i) ((i)<<1)
#define HEAP_RCHILD(i) (((i)<<1)+1)
extern void
heap64_minpq_heapify (heap64_t *h, int ind);
extern void
heap64_maxpq_heapify (heap64_t *h, int ind);
extern void *
heap64_peek (heap64_t *h);
extern void
heap64_minpq_insert (heap64_t *h, void* elem);
extern void
heap64_maxpq_insert (heap64_t *h, void *elem);
extern void
heap64_minpq_pop (heap64_t *h);
extern void
heap64_maxpq_pop (heap64_t *h);
extern void
pr_heap (heap64_t *h, void (*pr_data)(void *));
#endif
>> Contents of "pqueue.c"
#include "threads/pqueue.h"
/* Allocates space for heap types’s data and initializes it
```

```
* Caller must ensure/check that:
* h is not NULL
* h->heap is a valid memory address (i.e. NOT NULL) after the call
*/
heap64_t *
heap64_init (heap64_t *h, int max_nelem)
{report.txt

4
ASSERT( h != NULL);
h->heap = malloc (max_nelem * sizeof(heap64_elem_t) );
h->max_nelem = max_nelem;
h->n_elem = 0;
return h;
}
/* Returns true if heap has no elements
* Caller must ensure that:
* h is not NULL
*/
bool
heap64_is_empty (heap64_t *h)
{
ASSERT (h != NULL);
return (h->n_elem <= 0);
}
/* Returns true if heap is full
* Caller must ensure that:
* h is not NULL
*/
bool
heap64_is_full (heap64_t *h)
{
ASSERT (h != NULL);
return (h->n_elem >= h->max_nelem);
}
/* Min heapifies the heap in h->heap from index ind (1 based)
* Caller must ensure that
* h is not NULL
*/
void
heap64_minpq_heapify (heap64_t *h, int ind)
{
ASSERT (h != NULL);
int smallest;
while (1) {
int l = HEAP_LCHILD(ind);
int r = HEAP_RCHILD(ind);
if (l <= h->n_elem && h->heap[l].key < h->heap[ind].key)
smallest = l;
else
smallest = ind;
if (r <= h->n_elem && h->heap[r].key < h->heap[smallest].key)
smallest = r;
if (smallest == ind)
break;
else {
heap64_elem_t temp = h->heap[ind];
h->heap[ind] = h->heap[smallest];
h->heap[smallest] = temp;
ind = smallest;
```

```
}
}
}
void
heap64_maxpq_heapify (heap64_t *h, int ind)
{report.txt

5
ASSERT (h != NULL);
int largest;
while (1) {
int l = HEAP_LCHILD(ind);
int r = HEAP_RCHILD(ind);
if (l <= h->n_elem && h->heap[l].key > h->heap[ind].key)
largest = l;
else
largest = ind;
if (r <= h->n_elem && h->heap[r].key > h->heap[largest].key)
largest = r;
if (largest == ind)
break;
else {
heap64_elem_t temp = h->heap[ind];
h->heap[ind] = h->heap[largest];
h->heap[largest] = temp;
ind = largest;
}
}
}
/* Returns pointer to top element in heap
* Caller must ensure/check that:
* h is not NULL
* heap has at least one element
* Caller must cast the returned void * to their expected type
*/
void *
heap64_peek (heap64_t *h)
{
ASSERT (h != NULL);
return h->heap + 1;
}
/* Inserts into min heap
* Caller must ensure that
* h is not NULL
* heap has at least one element
* Caller must pass pointer to the element they are inserting
*/
void
heap64_minpq_insert (heap64_t *h, void* elem)
{
ASSERT (h != NULL);
ASSERT (elem != NULL);
int i, p;
i = ++(h->n_elem);
h->heap[i] = *((heap64_elem_t *)elem);
while (i > 1 && h->heap[p = HEAP_PARENT(i)].key > h->heap[i].key) {
heap64_elem_t temp = h->heap[p];
h->heap[p]= h->heap[i];
h->heap[i] = temp;
i = p;
```

```
}
}
voidreport.txt

6
heap64_maxpq_insert (heap64_t *h, void *elem)
{
ASSERT (h != NULL);
int i, p;
i = ++(h->n_elem);
h->heap[i] = *((heap64_elem_t *)elem);
while (i > 1 && h->heap[p = HEAP_PARENT(i)].key < h->heap[i].key) {
heap64_elem_t temp = h->heap[p];
h->heap[p]= h->heap[i];
h->heap[i] = temp;
i = p;
}
}
void
heap64_minpq_pop (heap64_t *h)
{
ASSERT (h != NULL);
h->heap[1] = h->heap[h->n_elem--];
heap64_minpq_heapify(h, 1);
}
void
heap64_maxpq_pop (heap64_t *h)
{
ASSERT (h != NULL);
h->heap[1] = h->heap[h->n_elem--];
heap64_maxpq_heapify(h, 1);
}
void
pr_heap (heap64_t *h, void (*pr_data)(void *))
{
ASSERT (h != NULL);
int i;
for (i = 1; i <= h->n_elem; i++) {
printf ("(%d)[ key = %"PRId64" ; { ", i, h->heap[i].key);
pr_data (h->heap[i].data);
printf (" } ]\n");
}
}
>> The declarations for our priority queue of timer alarms was added to
>> "threads.h".
>> Changes to "thread.h"
@@ -4,6 +4,7 @@
#include <debug.h>
#include <list.h>
#include <stdint.h>
+#include "threads/pqueue.h"
/* States in a thread's life cycle. */
enum thread_status
@@ -138,4 +139,25 @@
int thread_get_recent_cpu (void);
int thread_get_load_avg (void);
+/* Timer-alarm structure */
+typedef struct alarm_t {
+ struct thread *t;
/* Thread setting the alarm */
```

```
+ int64_t wk_tm;
/* Wake-Up time (units: ticks since boot) */report.txt

7
+} alarm_t;
+
+/* Priority queue of Alarms: Earliest alarm first */
+extern heap64_t timer_pq;
+
+/* Ticks since boot (used for waking up threads) */
+extern int64_t glob_tm;
+
+/* Return true if first thread has a higher priority, false otherwise */
+bool priority_check(const struct list_elem *, const struct list_elem *,
void *);
+
+/* Maximum size of Timer-alarm Priority queue */
+#define TIMERPQ_MAXSZ 50
+
+/* Maximum no. of processes that can be released per tick */
+#define MAX_UNBLOCKS_PER_TICK 10
+
#endif /* threads/thread.h */
---- ALGORITHMS ----
>> Changes to "timer.c"
@@ -90,10 +90,23 @@
timer_sleep (int64_t ticks)
{
int64_t start = timer_ticks ();
-
+ struct thread *t = thread_current();
+
ASSERT (intr_get_level () == INTR_ON);
- while (timer_elapsed (start) < ticks)
-
thread_yield ();
+
+ enum intr_level old_level = intr_disable ();
+
+ /* Set alarm for the required wakeup time */
+ alarm_t alm;
+ alm.t = t;
+ alm.wk_tm = start + ticks;
+
+ /* Push alarm to the timer priority queue and block */
+ heap64_minpq_insert(&timer_pq, &alm);
+ thread_block();
+
+ /* Sleep is over. Wakeup */
+ intr_set_level (old_level);
}
/* Sleeps for approximately MS milliseconds. Interrupts must be
@@ -171,6 +184,7 @@
timer_interrupt (struct intr_frame *args UNUSED)
{
ticks++;
+ glob_tm = ticks;
thread_tick ();
}
>> Changes to "thread.c"
```

```
@@ -50,6 +50,10 @@
static long long kernel_ticks;
static long long user_ticks;
+/* Priority Queue of alarms */
+heap64_t timer_pq;
+int64_t glob_tm;
/* # of timer ticks in kernel threads. */
/* # of timer ticks in user programs. */report.txt

8
+
/* Scheduling. */
#define TIME_SLICE 4
static unsigned thread_ticks;
/* # of timer ticks to give each thread. */
/* # of timer ticks since last yield. */
@@ -134,8 +149,31 @@
else
kernel_ticks++;
-
-
+
+
+
+
+
+
+
+
+
+
+
+
+
+
+
+
+
+
+
+
+
+
+
+
/* Enforce preemption. */
if (++thread_ticks >= TIME_SLICE)
int n_unblocked = 0; /* No. of threads unblocked in this run */
/* While there're threads to ublock and
* we haven't unblocked too many */
while (heap64_is_empty (&timer_pq) == false &&
n_unblocked < MAX_UNBLOCKS_PER_TICK) {
/* Get the thread with the earliest wake time */
alarm_t alm = *((alarm_t *)heap64_peek (&timer_pq));
/* If it's before wake-up time break,
* otherwise wake the thread up */
if (alm.wk_tm > glob_tm)
break;
else {
```

```
heap64_minpq_pop (&timer_pq);
thread_unblock (alm.t);
/* Wake the thread up */
n_unblocked++;
/* Increment the count of unblocks */
}
}
/* Enforce preemption upon time-slice expiry and
* preempt idle thread if any other threads were unblocked*/
if (++thread_ticks >= TIME_SLICE ||
(n_unblocked > 0 && t == idle_thread))
intr_yield_on_return ();
}
```
>>
>>
>>
>>
>>
>>
>>
>>
>>
>>
>>
>>
When timer_sleep() is called we create an alarm object which stores a pointer to the
calling thread and the time that it wants to wake up at (units - ticks since OS boot)
We push this object in a min priority queue and block the calling thread.
When the timer_interrupt() is called we update the glob_tm variable with the current
time (units ticks since OS boot). timer_interrupt then calls thread_tick(). Inside
thread_tick() we peek for the thread with the earliest wake up time (O(1) operation
in the min heap). If the current time is past or equal to its wake up time, we
iteratively wake up as many threads as we can upto the limit specified by MAX_UNBLOCKS_PER_TICK
These threads can be scheduled only when the scheduler gets the chance to run again
which might be after the current thread expires its time slice. However if the idle
thread is currently running and we have unblocked atleast one thread we preempt the
idle thread.
---- SYNCHRONIZATION ----
>>
>>
>>
>>
In order to ensure that no race conditions occur during simultaneous access to
thread_sleep() we disable interrupts before any action statements occur. This gives
us the chance to insert the alarm to the min heap and block the calling thread
```
```

atomically. Interrupts get enabled again by the next thread to be scheduled.
---- RATIONALE ----
>>
>>
>>
>>
>>
Race conditions were avoided by disabling interrupts instead of using another
mechanism like semaphores, locks, etc. because cooperation between different
interacting threads could not be guaranteed in this case.
We had also considered maintaining a sleep interval variable inside each thread structure which would get decremented at each timer tick allowing
thereport.txt

9
>> thread to wake up when that variable becomes zero. However, this approach required
>> us to perform operations linear in the number of threads at each timer tick (O(n)).
>> Our current approach ONLY takes amortized constant time per tick.
PRIORITY SCHEDULING A
=====================
---- DATA STRUCTURES ----
>>
>>
>>
>>
Ready list is now maintained as an ordered list on the basis of thread priority. The highest priority thread will reside at the front of the list.
This was done by utilising the functions list_insert_ordered() provided in
"src/libs/kernel/list.c" by providing a comparator function priority_check()
---- ALGORITHMS ----
>> A comparator function priority_check() was defined to ensure sorted ordering
>> within the ready list.
>> Changes to "thread.c"
@@ -71,6 +75,17 @@
void thread_schedule_tail (struct thread *prev);
static tid_t allocate_tid (void);
+/* Return true if first thread has a higher priority, false otherwise */
+bool priority_check(const struct list_elem * first,
const struct list_elem * second, void *aux)
+{
+ struct thread * first_thread = list_entry (first, struct thread, elem);
+ struct thread * second_thread = list_entry (second, struct thread, elem);
+ if (first_thread->priority > second_thread->priority)
+
return true;
+ else
+
return false;
+}
+

```
/* Initializes the threading system by transforming the code
that’s currently running into a thread. This can’t work in
general and it is possible in this case only because loader.S
>>
>>
>>
>>
Changes were made to ready list manipulation statements inside thread.c
in order
to ensure that the list remains sorted and threads are scheduled
according to
their priorities by the scheduler.
Changes to "thread.c"
@@ -209,6 +247,20 @@
/* Add to run queue. */
thread_unblock (t);
+
+
+
+
+
+
+
+
/* If the newly created process has higher priority it may execute before
this function finishes */
old_level=intr_disable();
if (!list_empty(&ready_list)){
struct thread * current_thread = thread_current();
struct thread * list_front_thread = list_entry (list_front (&ready_list),
struct thread, elem);
if (current_thread->priority < list_front_thread->priority)
thread_yield();
}
+
+
+
+
+
+
+
intr_set_level (old_level);
return tid;
}report.txt

10
@@ -245,7 +297,7 @@
-
+
old_level = intr_disable ();
ASSERT (t->status == THREAD_BLOCKED);
list_push_back (&ready_list, &t->elem);
list_insert_ordered (&ready_list, &t->elem, priority_check, NULL);
t->status = THREAD_READY;
intr_set_level (old_level);
}
@@ -316,7 +368,7 @@
old_level = intr_disable ();
if (cur != idle_thread)
-
list_push_back (&ready_list, &cur->elem);
```

```
+
list_insert_ordered (&ready_list, &cur->elem, priority_check, NULL);
cur->status = THREAD_READY;
schedule ();
intr_set_level (old_level);
@@ -339,11 +391,25 @@
}
}
-/* Sets the current thread’s priority to NEW_PRIORITY. */
+/* Sets the current thread’s priority to NEW_PRIORITY.
+ * Yield if the current thread no longer has the
+ * highest priority among all the ready threads */
void
thread_set_priority (int new_priority)
{
- thread_current ()->priority = new_priority;
+ enum intr_level old_level = intr_disable();
+
+ struct thread * current_thread = thread_current();
+ int current_priority = current_thread->priority;
+ current_thread->priority = new_priority;
+
+ if (new_priority < current_priority && !list_empty ( &ready_list)){
+
struct thread * list_front_thread = list_entry (list_front (&ready_list),
struct thread, elem);
+
if (current_thread->priority < list_front_thread->priority)
+
thread_yield();
+ }
+
+ intr_set_level (old_level);
}
/* Returns the current thread’s priority. */
---- SYNCHRONIZATION ----
>> No changes were made to the synchronization arrangements within the
functions
>> modified because we were already operating with interrupts disabled
---- RATIONALE ----
>> The approach taken by us allows us to implement the desired operations
in a
>> very simple yet efficient manner, allowing us to reuse the
functionality already
>> available with the list implementation.
PRIORITY SCHEDULING B
=====================
---- ALGORITHM ----
>> Changes to "synch.c"report.txt

11
@@ -68,7 +68,7 @@
old_level = intr_disable ();
while (sema->value == 0)
{
-
list_push_back (&sema->waiters, &thread_current ()->elem);
+
list_insert_ordered (&sema->waiters, &thread_current ()->elem,
priority_check, NULL);
```

```
thread_block ();
}
sema->value--;
@@ -113,10 +113,17 @@
ASSERT (sema != NULL);
-
+
+
+
-
+
+
+
+
+
+
old_level = intr_disable ();
if (!list_empty (&sema->waiters))
if (!list_empty (&sema->waiters)) {
struct thread * current_thread = thread_current();
struct thread * list_front_thread = list_entry (list_front (&sema-
>waiters),
struct thread, elem);
thread_unblock (list_entry (list_pop_front (&sema->waiters),
struct thread, elem));
sema->value++;
sema->value++;
if (current_thread->priority < list_front_thread->priority)
thread_yield();
}
else
sema->value++;
intr_set_level (old_level);
---- RATIONALE ----
>> Similar to the approach used in thread priority scheduling we
converted
>> the semaphore waiting list to an ordered list based on the priority of
the threads
>> waiting on the semaphore.
-----------------------------------------END-------------------------------
-------------
```