CS 570 - Introduction to Operating Systems

Gabriel Loewen

October 4th, 2011

Project 1: Threads
Design Document

Contents

1 Preliminaries

Before working on this project I reviewed the Pintos documentation[1] as well as the Pintos SIGCSE 2009 presentation[2].

2 Alarm Clock

The purpose of the modifications to the *timer.c* file is to remove the necessity for busy waiting.

2.1 Data Structures

Functions that I added:

1. `void start_sleep(struct thread *thread, int64_t ticks);`
2. `void wake_thread(struct thread *thread);`
3. `void check_thread(struct thread *thread, void *aux UNUSED);`

The *start_sleep* function sets the field *num_ticks_to_sleep* in the thread struct to the number specified in the parameter of the function and then the thread is blocked. When the *timer_interrupt* function is called, the tick count in incremented and each thread is checked by calling the *check_thread* function. The *check_thread* function decrements the threads tick count and checks if the thread has slept for the appropriate number of ticks (i.e. when $num\_ticks\_to\_sleep == 0$). If the thread has slept for the appropriate number of ticks *wake_thread* is called and is responsible for unblocking the thread. Modifications to *struct thread* include the addition of the field *num_ticks_to_sleep*.

2.2 Algorithms

Briefly describe what happens in a call to timer_sleep(),including the effects of the timer interrupt handler.
When the function *timer_sleep* is called the following takes place:

- Ensure that interrupts are turned on

[1] `http://www.stanford.edu/class/cs140/projects/pintos/pintos.pdf`
[2] `http://www.pintos-os.org/wp-content/files/SIGCSE2009-Pintos.pdf`

- If the number if ticks to sleep is 0 then return
- Call the function *start_sleep*

When the function *start_sleep* is called the following takes place:

- The number of ticks to sleep is set in the thread struct
- Interrupts are turned off
- The thread is blocked
- Interrupts are turned back on

Once the function *start_sleep* is called the thread sits in the *all_list* while sleeping. Once per second the timer interrupt handler is called and the following takes place:

- The tick count is incremented
- Each thread is checked using the *thread_foreach* function which is passed a pointer to the function *check_thread*.
- The threads tick count is decremented and then we evaluate the expression *num_ticks_to_sleep == 0* and if that is true the thread is unblocked.

What steps are taken to minimize the amount of time spent in the timer interrupt handler?
The timer interrupt handler is written to only check blocked threads. Since we only check blocked threads and ignore all other threads we can minimize the amount of time spent checking if threads need to be woken.

2.3 Synchronization

How are race conditions avoided when multiple threads call timer_sleep() simultaneously?
Interrupts are disabled when a thread is blocked, therefore race conditions are avoided.

How are race conditions avoided when a timer interrupt occurs during a call to timer_sleep()?
Interrupts are disabled during in the timer interrupt, so the interrupt cannot be interrupted by a call to timer_sleep().

2.4 Rationale

Why did you choose this design? In what ways is it superior to another design you considered?
I chose this design because it was the easiest to implement and seemed to be a reasonable choice. When a thread is initialized it is placed in the all_list, which is essentially a list of all processes. My original idea was to create a completely separate list for the sleeping threads but it seemed to be superfluous since the all_list contained all threads and it is easy to check if a thread is blocked or not. However, looking back it might have been a better choice to create a new list of sleeping threads just because it would remove the necessity to constantly check the threads running status since, presumably all threads in the sleeping list would be in the THREAD_BLOCKED state.

3 Priority Scheduling

Implement priority scheduling and priority donation.

3.1 Data Structures

Copy here the declaration of each new or changed 'struct' or 'struct' member, global or static variable, 'typedef', or enumeration. Identify the purpose of each in 25 words or less.
The following fields were added to the struct thread:

1. int priority;
2. int initial_priority;
3. struct list locks;
4. struct lock *blocking_lock;
5. bool donated_to;

The *priority* integer represents the priority (between 0 and 63) of the thread. If a thread donates priority to another thread then we swap priorities and store the original priority in the *initial_priority* integer and we set the *donated_to* boolean on the donating thread. The *blocking_lock* field is a pointer to the lock which is blocking the thread and the list of locks which the thread

holds. The list is always sorted by priority to ensure the lock with highest priority is given preference.

The following fields were added to the struct lock:

1. int priority;
2. bool initial_priority;
3. struct list_elem holderItem;

The integer *priority* keeps track of the priority of the lock. When priority is donated we track these donations using the integer *initial_priority* and we keep the list element *holderItem* which represents the thread which holds the lock.

Explain the data structure used to track priority donation.
I use the previously described fields of the struct thread and lock to keep track of donated priority. When a thread acquires a lock the lock will be put into the threads list of locks which is sorted by lock priority. When a lock is released it is removed from the threads list of locks. When a lock is acquired the thread acquiring the lock and the thread which holds the lock compare their priorities and if the priority of the acquiring thread is higher then the lock holders priority then priority is donated. The initial priority of the donating thread is preserved so that once the lock is released we can revert the threads priority to its initial value. We also set the boolean value *donated_to* which helps track threads which have donated priority. An example of the way this works comes from the Pintos testing suite:
Three threads are created: Thread A (priority 31), Thread B (priority 32), and Thread C (priority 33).
Thread A has lock a and Thread B has lock b.
Thread B acquires lock a and thread C acquires lock b. So, the donation process happens as follows:

Thread A:

- priority = 31
- initial_priority = 31

- donated_to = false
- locks = lock a
- blocking_lock = null

Thread B:

- priority = 32
- initial_priority = 32
- donated_to = false
- locks = lock b
- blocking_lock = null

Thread C:

- priority = 33
- initial_priority = 33
- donated_to = false
- locks = EMPTY
- blocking_lock = null

Once Thread B acquires lock a the following changes are made:
Thread A:

- priority = 31
- initial_priority = 32
- donated_to = true
- locks = lock a
- blocking_lock = null

Thread B:

- priority = 32
- initial_priority = 32
- donated_to = false
- locks = lock b
- blocking_lock = lock a

Thread C:

- priority = 33
- initial_priority = 33
- donated_to = false
- locks = EMPTY
- blocking_lock = null

Once Thread C acquires lock b the following changes are made:
Thread A:

- priority = 31
- initial_priority = 32
- donated_to = true
- locks = lock a
- blocking_lock = null

Thread B:

- priority = 32
- initial_priority = 33
- donated_to = false
- locks = lock b

- blocking_lock = lock a

Thread C:

- priority = 33
- initial_priority = 33
- donated_to = false
- locks = EMPTY
- blocking_lock = lock b

Thread A will release lock a:
Thread A:

- priority = 31
- initial_priority = 32
- donated_to = false
- locks =
- blocking_lock = null

Thread B:

- priority = 32
- initial_priority = 33
- donated_to = true
- locks = lock b, lock a
- blocking_lock = null

Thread C:

- priority = 33
- initial_priority = 33

- donated_to = false
- locks = EMPTY
- blocking_lock = lock b

Thread B will release lock b:
Thread A:

- priority = 31
- initial_priority = 31
- donated_to = false
- locks =
- blocking_lock = null

Thread B:

- priority = 32
- initial_priority = 32
- donated_to = true
- locks = lock a
- blocking_lock = null

Thread C:

- priority = 33
- initial_priority = 33
- donated_to = false
- locks = lock b
- blocking_lock = lock b

3.2 Algorithms

The idea of priority scheduling is that each thread has a priority within the range of 0 and 63. In my implementation, when a thread is created the following takes place:

- The thread is inserted into the all_list.
- Once the thread enters the THREAD_READY state it is inserted into the ready list in order of priority where the thread with highest priority sits at the head of the list.
- When the time comes to schedule the thread the scheduler simply pops the first element off of the ready list
- If a new thread is created with a higher priority than the current thread the current thread is immediately pre-empted and put back into the ready list in sorted order and the new thread is scheduled.

How do you ensure that the highest priority thread waiting for a lock, semaphore, or condition variable wakes up first?
The list of threads are always sorted by priority such that the thread which has the highest priority will be first to be selected and if the priority of the thread holding the lock is higher than the acquiring thread priority then priority is donated.

3.3 Synchronization

Describe a potential race in thread_set_priority() and explain how your implementation avoids it. Can you use a lock to avoid this race?
It is possible that during a call to thread_set_priority() the priority of the lock holder might be modified by the priority donor thread and at the same time the thread might set its priority which could cause unpredictable behavior. However, interrupts are disabled during this so it is highly unlikely that this condition would happen. This could be fixed using a lock, however I have not implemented it in that manner.

3.4 Rationale

Why did you choose this design? In what ways is it superior to another design you considered?
Priority donation is a feature that must be easily tracked within between multiple threads. This is why I have implemented priority scheduling in a way which makes tracking donations easy using sorted lists. When a thread donates priority fields within the struct thread are set which indicate that priority has been donated. When the thread with donated priority is released it is easy to select the donated thread and reset its priority to its initial priority.
Threads waiting on a semaphore are treated similarly, the list of waiting threads is sorted by priority such that the thread with highest priority is given preference.

4 Advanced Scheduler

Implement BSD type multi-level feedback queue scheduling

4.1 Data Structures

The following were added to thread.h:
#define NICE_MAX 20
#define NICE_DEFAULT 0
#define NICE_MIN -20
The struct thread was modified to include:

- int nice;
- int recent_cpu;

And in thread.c I have added the fields:

- int load_average;

4.2 Algorithms

Suppose threads A, B, and C have nice values 0, 1, and 2. Each has a recent_cpu value of 0. Fill in the table below showing the

scheduling decision and the priority and recent_cpu values for each thread after each given number of timer ticks:

	recent_cpu			priority			
timer ticks	A	B	C	A	B	C	thread to run
0	0	1	2	63	61	59	A
4	4	1	2	62	61	59	A
8	7	2	4	61	61	58	B
12	6	6	6	61	59	58	A
16	9	6	7	60	59	57	A
20	12	6	8	60	59	57	A
24	15	6	9	59	59	57	B
28	14	10	10	59	58	57	A
32	16	10	11	58	58	56	B
36	15	14	12	59	57	56	A

Did any ambiguities in the scheduler specification make values in the table uncertain? If so, what rule did you use to resolve them? Does this match the behavior of your scheduler?
No, not particularly. The behavior of my scheduler should match this behavior.

4.3 Rationale

Briefly critique your design, pointing out advantages and disadvantages in your design choices. If you were to have extra time to work on this part of the project, how might you choose to refine or improve your design?
My design is implemented as a priority sorted ready list in which threads are added to the list in order of priority. On every fourth tick it is required that the scheduler recalculate the threads priority and if the priorities have changed resort the ready list. The idea is that the highest priority job should always be at the head of the list such that the scheduler only needs to pop the head of the list in order to retrieve the next job.
The assignment explains arithmetic for fixed-point math in detail, but it leaves it open to you to implement it. Why did you decide to implement it the way you did? If you created an abstraction layer for fixed-point math, that is, an abstract data type and/or a

set of functions or macros to manipulate fixed-point numbers, why did you do so? If not, why not?
Because load_average and recent_cpu are floating point numbers I calculated these values using the functions found in *fixed-point.c*. These functions allow me to calculate values which require floating point operations. I didn't create macros for these calculations because I am more comfortable with the implementation using functions.