Project 1: Threads

Design Document

Design

Task 1: Efficient Alarm Clock

Data structures and functions

Add one attribute `sleep_ticks` to the `struct thread`

```
struct thread
  {
    /* ----- Omit Attributes Defined in Pintos Source Code ------ */
    /* --------------- NEW ATTRIBUTE ---------------- */
    int64_t sleep_ticks;                    /* ticks that a thread will sleep*/
  };
```

I have added a new attribute `sleep_ticks` at the end of the struct thead. It is `int64_t` because the function `timer_sleep` accepts a parameter which is also `int64_t`. By default, `sleep_ticks` has value 0.

Algorithms

Let's first clarify the **problem** existed in the current implementation of `timer_sleep`.

Assume that the current system has implemented the **priority scheduler**, which means the thread with the highest priority will always be executed first. The original implementation of `timer_sleep` uses `thread_yield()` in a loop to make a thread sleep for certain ticks, in which the major problem lies. `thread_yield()` does not put the current thread to sleep. Instead, it simply puts the current thread back into the **ready queue**. Then this thread will be scheduled again. This kind of action falls into a loop which is called "busy waiting".

In order to tackle this problem, we must put the thread into real sleep and wake it after the number of ticks set before. My algorithm is as follows.

Add another list called `sleep_list` to hold all the sleeping threads. When calling `timer_sleep` , first set the `sleep_ticks` of the thread to be the value of parameter `ticks`. Then call `thread_block()` to put the thread into sleep and push the thread to the `sleep_list`. At every timer interrupt, we will iterate the `sleep_list` , decrease the `sleep_ticks` by 1, and if `sleep_ticks` equals to 0, we will call `thread_unblock()` which unblocks the thread and put it into the ready queue.

Synchronization

`sleep_list`

The `sleep_list` is a resource which can be accessed concurrently by more than 1 thread. Thus, there is synchroniztion problem since the list implemention in pintos is not thread-safe.

Rough Solution:

Each thread should first acquire a lock before operating on the list. And when it finishes the operation, it should release the lock. This ensures that only one thread can operate on the list at one time. Therefore, guarantee the correctness of program.

Rationale

Alternative 1: Do not use `sleep_list` , add the sleeping thread into `all_list` directly. Call `thread_foreach()` to do the check at every timer interrupt.

Drawback: `thread_foreach()` needs to iterate through all threads and check whether its status is THREAD_BLOCK and check its `sleep_ticks` . It generates a large time waste. And the code that runs in interrupt handlers should be as fast as possible. So, I finally choose creating a `sleeping_list` and only do iteration and check on this shorter list each timer interrupt.

Task 2: Priority Scheduler

Data structures and functions

Modify `struct thread`

```
struct thread
  {
    /* ------- Omit Attributes Defined in Pintos Source Code ------- */
    /* --------------- NEW ATTRIBUTE ---------------- */
    int original_priority;
    static struct list lock_holding;            /* locks hold by the thread */
    struct lock *lock_waiting;          /* lock that the thread is waiting */
  };
```

Algorithms

1. **Higher-priority threads always run before lower-priority threads.**

   The key points to realize this function is to **maintain the order of the ready list.**

   To maintain the order of the ready list, we need to insert new element to their proper position instead simply push it at the end of the list. The list.c has provides us a function `list_insert_ordered()` to perform this opertion. There are 2 places in which we add new element to the `ready_list`. They are `thread_unblock()` , `thread_yield()` . Replace the `list_push_back()` of `thread_unblock()` attains our object. Now, our `ready_list` is always of descending order. And each time the scheduler will pick the first thread which has the highest priority to run.

Besides, we must also consider `preempt scheduler` , which ensures a high-priority thread really preempts. It means when a newly created thread has higher priority or a thread sets its prority to be higher than the current thread, we should immediatly switch the thread. Re-schedule only happens when a new thread is add to the `ready_list` or a thread calls `thread_set_priority()` to change the priority. So we should first add a `thread_yield` at the end of the `thread_set_priority()` to re-schedule threads. At first, I want to add a `thread_yield` at the end of `thread_unblock()` because it adds a new element to the `ready_list`. However I noticed that the annotation said this function does not preempt the running thread. So I add a piece of code to reschedule threads at the end of `thread_create()` instead.

2. **Priority Donation**

    1. `Changing thread's priority`

        There are only two situations when the priority donation is triggerd. The one is `Acquire a Lock`, the other is `Release a Lock` .

        When acquring a lock, if the thread with higher priority (abbr. HP) wants to acquire a lock which is held by a lower priority (abbr. LP) thread, the HP thread will set the priority of the LP to be the same as HP. Then, our scheduler will choose the original LP thread to run. After the original release the lock, its priority will be set based on my design scheme which will be introduced later. Then, the HP thread can get the lock the continue to run.

    2. `Acquire a Lock`

        Key: Make sure the semaphore **waiter list** is in **descending order**

        When a thread acquires a lock, if the lock is available, the current thread get the lock, change the semaphore value of the lock to be 0. Then when other threads also calls `lock_acquire()` , they will be blocked and added to the semaphore waiter.

        ```
        /* Original design in the pintos source code */
        while (sema->value == 0)
        {
            list_push_back (&sema->waiters, &thread_current()->elem);
            thread_block();
        }
        ```

        Now, in my design, I need the semaphore `waiter` to be an ordered list. So I will call `list_insert_ordered()` instead of `list_push_back()`.

```
/* Now sema->waiters is a list with descending order */
while (sema->value == 0)
{
    list_insert_ordered (&sema->waiters, &thread_current()->elem);
    thread_block();
}
```

When a thread successfully gets the lock, the lock will be added into the thread's `lock_holding` list.

When a thread has to sleep to wait the lock being released, set the lock to be the `lock_waiting`

When acquring a lock, if the thread with higher priority (abbr. HP) wants to acquire a lock which is held by a lower priority (abbr. LP) thread, the HP thread will set the priority of the LP to be the same as HP. And the lock will be added into the thread `lock_waiting` list. After the donation, the original LP thread will run firstly. The original LP thread will first check if the `lock_waiting` attribute is NULL. If it isn't, the original LP shoud set the priority of the thread holding the `lock_waiting` to be the same as LP thread.

3. `Release a Lock`

   When a thread releasing a lock, we need to reset the thread priority based on elements in the `lock_holding`.

   First, remove the lock released from the `lock_holding` list.

   Then, Let's talk about 2 different situations.

   a. After removing, there is no element in the list.

   We simply set the priority of the thread to be the `original priority`.

   b. After removing, there are more than 1 element in the list.

   For each lock in the list, we can access the front value of the semaphore waiter list, which is the waiting thread with the highest priority. Thus, we can get the maximum among these front values. Set the thread priority to be the maximum value.

```
# pseudocode
max_priority = 0
for lock in lock_holding:
    thread = front(lock -> sema -> waiter)
    candidate = thread -> priority
    if candidate > max_priotity:
        max_priority = candidate
return max_priority
```

Synchronization

First Let's consider the new attributes added by my design. I only add `locks_holding` and `lock_waiting` to the struct thread. Because these attributes are in the thread struct, it can not be accessed by more than 2 accesses. So my modification won't generate synchronization issues.

There are list operations in `semaphore` . For example, `sema_down()` pushes a new thread to the waiter list. However, we could see that `sema_down()` calls `intr_disable()` to disable instruction which ensures synchronization.

Rationale

My design is clear and easy to implement.

Task 3: Multi-level Feedback Queue Scheduler

Data structures and functions

```
struct thread
  {
    /* ------ Omit Attributes Defined in Pintos Source Code ------ */
    /* --------------- NEW ATTRIBUTE ---------------- */
    fixed_t recent_cpu;         /* nice value of the thread */
    int nice;           /* recent_cpu of the thread */
  };
```

Algorithms

Multple parts of this scheduler require data to be updated after a certain number of timer ticks. Thus, we use timer interrupt handler `timer_interrupt()` to implement it.

`Overview`

Initialize:

Iterate all threads, put threads with same priority into the same queue.

```
# persudocode
for thread in ready_list:
    queues[thread->priority].push_back(thread)
```

Then, at every timer interrupt, update the priority of every thread. And also update queues.

```
# persudocode
for thread in ready_list:
    original_priority = thread -> priority
    update_priority(thread)
    if original_priority != thread -> priority:     # if the priority has changed
        queues[original_priority].delete(thread)
        queues[thread->priority].push_back(thread)
```

Choosing the queue with highest priority the pop the front thread to run.

- Update `priority`

  The `priority` of each thread will be updated **every fourth clock tick** accouding to the following fomula.

  ```
  priority = PRI_MAX - (rencent_cpu/4) - (nice*2)
  ```

- Update `recent_cpu`

  `recent_cpu` is incremented by 1 **every tick** for the running thread only, unless the idle thread is running.

  `recent_cpu` is recalculated based on the following fomular **once per second** (100 timer ticks).

  ```
  recent_cpu = (2*load_avg)/(2*load_avg+1)*recent_cpu + nice
  ```

- Update `load_avg`

  It is updated according to the following formular **once per second**.

  ```
  load_avg = (59/60)*load_avg + (1/60)*ready_threads
  ```

  Since `load_avg` is system-wide, not thread-specific, I don't put it into the struct thread. Instead, I add a static variable called `load_avg` in the thread.c.

Synchronization

`queues`

`queues` is an array holds all the priority list. The priority list can be accessed by more than 2 threads concurrently. Therefore, one thread must accquire a lock when operating on the list.

`recent_cpu` and `nice` are inside one thread. Thus there are no synchronization problems.

`load_avg` is updated inside the timer interrupt handler. Thus there are no synchronization problems.

Rationale

In my design, update queues is a little inefficient. I need O(n) time to find the thread to perform delete action. The better solution is that I can implement a real priority queue using heap. Then O(log n) insertion and O(logn) deletion.

The MLFQS problem

timer ticks	R(A)	R(B)	R(C)	P(A)	P(B)	P(C)	thread to run
0	0	0	0	63	61	59	A
4	4	0	0	62	61	59	A
8	8	0	0	61	61	59	B
12	8	4	0	61	60	59	A
16	12	4	0	60	60	59	B
20	12	8	0	60	59	59	A
24	16	8	0	59	59	59	C
28	16	8	4	59	59	58	B
32	16	12	4	59	58	58	A
36	20	12	4	58	58	58	C

- Did any ambiguities in the scheduler specification make values in the table (in the previous question) uncertain? If so, what rule did you use to resolve them?

  Yes. there are always threads of the same priority.

  My solution is to follow the FIFO principle. The thread first enters the priority queue (of a certain value) will be executed first.

Questions About Pintos Source Code

- **Tell us about how pintos start the first thread in its thread system (only consider the thread part).**

In the **init.c** which is located at the thread folder, funtion `thread_init()` has been called to initialize a thread which is exactly the first thread in the pintos thread system.

The function `thread_init()` is defined at **thread.c**. Before `thread_init()`, a static variable called `initial_thread` has been declared. The detailed declaration is as follows:

```
static struct thread *initial_thread;
```

In the `thread_init()` , it sets up a **thread structure** for the intial_thread.

```
intial_thread = running_thread(); // returns a running thread
/*Does basic initialization of initial_thread having default priority with name
"main"*/
init_thread(initial_thread, "main", PRI_DEFAULT);
initial_thread->status = THREAD_RUNNING; // change the thread status to
THREAD_RUNNING
initial_thread->tid = allocate_tid();   // allocate tid for the initial_thread
```

- **Consider priority scheduling, how does pintos keep running a ready thread with highest priority after its time tick reaching TIME_SLICE?**

Let's first look at the function `thread_tick()`. It is called by the timer interrupt handler at each timer tick. Look at the following slice of code.

```
if (++thread_ticks >= TIME_SLICE)
    intr_yield_on_return();
```

When the thread tick reaches TIME_SLICE, `thread_tick()` will call `intr_yield_on_return()` , which will set the variable `yield_on_return` to be true. `yield_on_return == true` means that we could request that a new process be scheduled just before the interrupt returns. The following slice of code in the `intr_handler()` reveals the secret of `yield_on_return`.

```
if (yield_on_return)
    thread_yield();
```

The function `thread_yield` calls `schedule()` to decide the next thread to run. In the context of priority scheduling, the thread with the highest priority will be chosen. Therefore, pintos keeps running a ready thread with highest priority after its time tick reaching TIME_SLICE.

- **What will pintos do when switching from one thread to the other? By calling what functions and doing what?**

1. By calling `switch_threads`
2. It saves registers on the stack, saves the CPU's current stack pointer in the current struct thread's stack member, restores the new thread's stack into the CPU's stack pointer, restores registers from the stack and returns.

   Ref: CS302-OS-project1-guide

How does pintos implement floating point number operation

Basic idea: treat the rightmost 16 bits of an integer to represent fraction.

1. Convert a value to fixed-point value.

   Left shift the integer for 16 bits. (the fraction bits are all 0)

2. Add 2 fixed-point value or substract 2 fixed-point value.

   simply do A+B and A-B

3. Add a fixed-point value A and an int value B (Substract is similar).

   first convert the int value to fixed-point value, then add them together.

4. Multiply a fixed-point value A by an int value B. (Divide is similar)

   multiply directly

5. Multiply two fixed-point value (simliar to Divide)

   To avoid overflow, it first converts A to `int_64`. After multiplying, right shift 16 bits to set the number of fraction bits to be 16.

6. Get the integer part of a fixed-point value

   Right shift the fixed-point for 16 bits

7. Get rounded integer of a fixed-point value

   If A >= 0, add 0.5 to the value and do shift. If A < 0, substract A by 0.5 and do shift. The final result is rounded it the nearest integer.

What do priority-donation test cases(priority-donate-chain and priority-donate-nest) do and illustrate the running process

Priority-donate-chain

What does it do ?

It constructs a lock chain. Each thread[i] holds the lock[i] and attempts to accquire lock[i-1] which is held by thread[i-1], except for thread[0], which is held by the main thread. Because the lock is held, thread[i] donates its priority to thread[i-1], which donates to thread[i-2], and so on until the main thread receives the donation.

What does it illustrate ?

1. We should add some data structure to record locks one thread holds and locks one thread is waiting for.
2. If one thread holds no locks after releasing, it should be set to the original priority directly.

Priority-donate-nest

What does it do ?

Low-priority main thread L acquires lock A. Medium-priority thread M then acquires lock B then blocks on acquiring lock A. High-priority thread H the blocks on acquiring lock B. Thus, thread H donates its priority to M which in turn donates it to thread L.

What does it illustrate ?

1. Priority donation can be in a chain form. So we need to add some data structure to record locks one thread holds and locks one thread is waiting for.

Project 2: User Programs

Final Report

Group member

- 11611908 龚玥
- 11612611 贺启旸

Task 1: Argument passing

Data structure

No new data structure is needed.

Algorithm

Use `strtok_r` to split the input argument string, and got each argument. Then push them to the process stack by the order according to the stack structure below

Address	Name	Data	Type
0xbffffffc	argv[3][...]	bar\0	char[4]
0xbffffff8	argv[2][...]	foo\0	char[4]
0xbffffff5	argv[1][...]	-l\0	char[3]
0xbfffffed	argv[0][...]	/bin/ls\0	char[8]
0xbfffffec	word-align	0	uint8_t
0xbfffffe8	argv[4]	0	char *
0xbfffffe4	argv[3]	0xbffffffc	char *
0xbfffffe0	argv[2]	0xbffffff8	char *
0xbfffffdc	argv[1]	0xbffffff5	char *
0xbfffffd8	argv[0]	0xbfffffed	char *
0xbfffffd4	argv	0xbfffffd8	char **
0xbfffffd0	argc	4	int
0xbfffffcc	return address	0	void (*) ()

Synchronization

No synchronization operation needed.

Rationale

We use smart method to implement word-align (bit operation => ptr&0xfffffffc).

Task 2: Process Control Syscalls

Data structure

Add a children list in `struct thread`

Add a struct to record the information of child process

```
struct child_process
  {
    tid_t tid;                          /* Thread identifier. */
    struct list_elem childelem;         /* List element for children list. */
    bool exited;                        /* Whether it has exited*/
    bool waited;                        /* Whether it has waited for some
child. */
    struct semaphore sema;              /* Wait semaphore. */
    int exit_status;                    /* Status when it exits. */
  };
```

Algorithm

The syscall stack is as below

```
f->esp   ----------------------------
        |           SYS_CODE          |
         ----------------------------
        |  arg[0] ptr -OR- int_value  |
         ----------------------------
        |  arg[1] ptr -OR- int_value  |
         ----------------------------
        |  arg[2] ptr -OR- int_value  |
         ----------------------------
```

We will first get the `SYS_CODE` to decide which syscall is called. The we will read the arguments and do validation. The major challange here is to valify whether the argument pointer is valid. We should not only check whether the address of the pointer is a valid user address, but also check whether it is mapped in the kenerl address.

The most difficult part is to implement `exec` and `wait`. When `exec`, the parent process must wait until the child process finished load operation. `wait` is to wait for a child process to finish, then the parent process can proceed executing.

Synchronization

When `exec`, the parent process must wait until the child process finished load operation. `wait` is to wait for a child process to finish, then the parent process can proceed executing.

Task 3: File Operation Syscalls

Data Stucture

In the `struct thread`, add a file list which records files the process opens.

```
struct file_control_block
{
  struct file *process_file;
  int fd;
  struct list_elem file_elem;
};
```

Algorithm

The implementaion of file operation is straightforward because the basic implementation is given in the `filesys/file.c`. The only point we need to pay attention to is the synchronization. We set a global lock named `filesys_lock`. Every time a process wants to user file operation syscall, it must firstly acquire the `filesys_lock`.

Synchronization

The only point we need to pay attention to is the synchronization. We set a global lock named `filesys_lock`. Every time a process wants to user file operation syscall, it must firstly acquire the `filesys_lock`.

Questions

What exactly did each member do? What went well, and what could be improved?

Qiyang he is mainly responsible for task1 and task2

Yue Gong is mainly responsible for task3.

We corporate well, finished each function step by step. We could furthur refactor our code to make it more readable. Currently, the arguments check part is too complex. Some part of the code can be more elegant.

Does your code exhibit any major memory safety problems (especially regarding C strings), memory leaks, poor error handling, or race conditions?

No. We will free every page that was allocated before. And we use semaphore and lock to solve race conditions.

Did you use consistent code style? Your code should blend in with the existing Pintos code. Check your use of indentation, your spacing, and your naming conventions.

Yes. We follow the pintos coding style.

Is your code simple and easy to understand?

Yes. We had no redundant code and each line of our codes has clear puporse.

If you have very complex sections of code in your solution, did you add enough comments to explain them?

Yes. For complex section, we always split it into small and simple sections. For nearly every code block, we have clear comment. You can easily get the idea of our implementation.

Did you leave commented-out code in your final submission?

Yes. Our final submission has clear and detailed comments.

Did you copy-paste code instead of creating reusable functions?

We encapsulate every common procedure into reusable functions.

Are your lines of source code excessively long? (more than 100 characters)

No.

Did you re-implement linked list algorithms instead of using the provided list manipulation

No. We use the original linked list algorithm.