Assignment 3 Report ::: PINTOS

Aditya Tewari 2018201082

Sandeep Gupta 2018201076

Installing Pintos on Ubuntu

:: Refer --> https://web.stanford.edu/class/cs140/projects/pintos/pintos_1.html

##

Understanding how Pintos program flow works::

1.1 Understanding GDB debugger to track programs:

:: Refer --> http://math.hws.edu/eck/cs431/f16/lab4/index.html

(GDB)- We will need to work in two terminals.

- In the first terminal execute the pintos --run function:
- Eg: pintos --gdb -v -- run alarm_multiple
- In the second terminal make sure you're in the ~/src/threads/build/ directory
- Execute:: pintos-gdb kernel.o
- To use GDB always use the second terminal now::
- To debug execute :: debugpintos
- Some instances of GDB are as follows::
    - "break function_name" to apply the breakpoint
    - "continue" or "c" { to continue till one loop}
    - "next" or "n" for the next line
    - "p variable_name " to print the variable

Detailed explanation is given in the link mentioned...

-- Understand makefiles -> see all the makefiles in the directories and understand the flow of the programs.

-- For changes try changing /src/utils/Makefile . Also

-- Some tips: you may change the CC to make compatibility accordingly set the FLAGS.

* Makefile clean operation here comes in handy.
-- It clears all the unnecessary .o files created while testing.

##

Test cases for part 2 : preemption of threads

------How to run a program for checking the outputs ?

Note:: Path to all the testcases we need :: ***~/pintos/src/threads/build/tests/threads/*.***

To test them individually we can use ::

--->> ***$ make***
~/pintos/src/threads/build/tests/threads/tests/threads/test-name-from-list.result

Moreover, understanding the flow of program and the structure of the list_elem was the trickiest.

--Refer list_elem structure in **list.h** and **list.c** for functions {/src/lib/kernel/list.h}

--Refer thread structure in threads.h and thread.c for functions {/src/threads/threads.h}

--Refer list_entry() macro from **list.c** ---- the hardest and the most generic function of the OS.

```
#define list_entry(LIST_ELEM, STRUCT, MEMBER)           \
        ((STRUCT *) ((uint8_t *) &(LIST_ELEM)->next     \
                     - offsetof (STRUCT, MEMBER.next)))
```

--Implementation of List.c and use of functions like list_insert_ordered(...) {} .

--Know and understand what thread_yield() , schedule() does. {}

--Understand the drawback of the function such as "list_pushback(...)" over list_insert_ordered(...)

Structures of Data_Types :

```
struct list_elem
  {
    struct list_elem *prev;     /* Previous list element. */
    struct list_elem *next;     /* Next list element. */
  };
```

```
struct list
 {
   struct list_elem head;      /* List head. */
   struct list_elem tail;      /* List tail. */
 };

struct thread
 {
   tid_t tid;                          /* Thread identifier. */
   enum thread_status status;          /* Thread state. */
   char name[16];                      /* Name (for debugging purposes). */
   uint8_t *stack;                     /* Saved stack pointer. */
   int priority;                       /* Priority. */
   struct list_elem allelem;           /* List element for all threads list. */
   /* Shared between thread.c and synch.c. */
   struct list_elem elem;              /* List element. */
#ifdef USERPROG
   /* Owned by userprog/process.c. */
   uint32_t *pagedir;                  /* Page directory. */
#endif
   /* Owned by thread.c. */
   unsigned magic;                     /* Detects stack overflow. */
   //Wakeup_ticks for inserting in wakeup queue
  // Added new
   int wakeup_ticks;
 };
```

##

Test cases for part 2 : preemption of threads

1.1 Alarm-single and Alarm-zero

--->> $ make ~/pintos/src/threads/build/tests/threads/tests/threads/alarm-single.result

-

It passes without making any change. It gives a brief idea of how to check for the program flow in Pintos.

We used GDB and putting several breakpoints on functions.

1.2 Alarm-multiple, alarm-negative, alarm-simultaneous

-Understand what these cases do.

-Refer to the output expected from their corresponding test-case.ck file

---> Refer {/src/devices/timer.c} to understand what it does,

- the function timer_sleep() is called.

- Most important is to understand what busy_wait means in this context and how is it different from busy_wait from what we refer in semaphore (Galvin)

- We have to make changes in timer_sleep().

-How?

- In {./thread.h} structure we insert an element to keep track of the wakeup_ticks.

- Refer what timer_ticks() , ticks variables are used for...

- Insted of using the busy_wait while loop for thread yield, we use the added extra variable in thread structure
(**int wakeup_ticks**) to keep track of when to wakeup the thread.

- Every thread is initialized with **wakeup_ticks = timer_ticks() + ticks;**

- Use this to insert list in ordered fashion using -- list_insert_ordered(...) .

- For the comparator function write your own. { first get to observe what this function actually does. Now the 3rd argument is
comparator function }

- The function compares the wakeup_ticks someway like this---

****

*int insert_sleep_at (struct list_elem *x , struct list_elem *y , void* aux UNUSED)*
{

*struct thread *first = list_entry(x , struct thread , elem);   ------ list entry usage*
*struct thread *last = list_entry(y , struct thread , elem);*

```
  if(first->wakeup_ticks < last->wakeup_ticks)  ---- using wakeup_ticks to push_back the threads
in a sorted order...
    return 1;

  return 0;
```
****

##

Test cases for part 3 : priority scheduling

1.Alarm-priority

→ $ make ~/pintos/src/threads/build/tests/threads/tests/threads/alarm-priority.result

- use the testing command as mentioned above to look at the way it's meant to be implemented.

- we make changes in /src/threads/thread.c

- a similar helper functions as we used in inserting thread according to wakeup time is (we can observe this by looking at the desired output)

- in thread_unblock() use this function to pushback according to the thread priority.

- Here the function list_insert_ordered is useful..

****

```
int priority_accordance ( struct list_elem* x  , struct list_elem* y , void* aux UNUSED)
{
  struct thread *first = list_entry( x , struct thread , elem );
  struct thread *last = list_entry( y , struct thread , elem );

  if(first->priority > last->priority)
    return 1;
  return 0;
}
```

2. Priority-change

→ $ make ~/pintos/src/threads/build/tests/threads/tests/threads/alarm-change.result

- use the testing command as mentioned above to look at the way it's meant to be implemented.

- we make changes in /src/threads/thread.c

- the main observation is the usage of the function "thread_set_priority(new_priority)"

- it dynamically changes the priority~
-

- after "creating thread" check if ready list changed and effects preemption of currently running thread

- so in thread create use a function which does so.

- we implemented **void priority_ordering();**

- checking if current thread priority is less than front of ready list(max priority thread).

- if so then we yield the thread.

```
*********************************************************************************
void priority_ordering()
{

  struct thread *t = thread_current();
  enum intr_level intt = intr_disable ();

  if(!list_empty(&ready_list))
  {
    struct thread *y = list_entry( list_front(&ready_list) , struct thread , elem );

    if(t->priority <= y->priority)
    {
      thread_yield();
    }
  }
  intr_set_level(intt);
}
```

3. 4. Priority-fifo and Priority-sema

→ $ make ~/pintos/src/threads/build/tests/threads/tests/threads/priority-fifi.result
→ $ make ~/pintos/src/threads/build/tests/threads/tests/threads/priority-sema.result

- use the testing command as mentioned above to look at the way it's meant to be implemented.
- we make changes in /src/threads/sync.c and sync.h

- Understand the modules of semaphores. The struct semaphore, struct lock and struct condition.

- Try to analyze how semaphores and locks are implemented practically, (we have read the theory)

- notice we have a concept like counting semaphore implemented in priority-sema...

- **sync.c** is the major function we tries to debug and atlast made some additions there.

- notice the function sema_down(...)

- here for priority-sema and fifo just one addition is enough.

- use the comparator function used in thread to insert the elements in the list based on threads priority.

- add the thread.c header in #include of the code to refer to the function... {}

- use the list_insert_ordered using the comparator function priority_accordance(...) mentioned above!

- after this check if the thread needs to be replaced with the higher_priority thread...

- this can be done using thread_block()

- another way is to use thread_yield().

5. Priority-condvar

→ $ make ~/pintos/src/threads/build/tests/threads/tests/threads/priority-condvar.result

- understand the structure of struct semaphore_elem and how this test is different from priority-sema

- this uses Binary semaphore instead of Counting semaphore.

- the function in sync.c {cond_wait()} is used here.

- function cond_signal() working understanding is important

- main observation is to see that the list in struct condition; --> waiters is needed to be sorted.

- the list contains entry of **structure - > semaphore_elem**

****

```
struct semaphore_elem
 {
   int priority_semaphore;          // Added to the structure
   struct list_elem elem;           /* List element. */

   struct semaphore semaphore;        /* This semaphore. */
 };
```

****

- instead of sorting we use an indirect approach. This exploits and reuses the entire code

- in the structure semaphore_elem; { as shown above }

- to keep track of priority of track we insert an element { **int priority_semaphore** }

- We assign it with the current thread's priority; in the function **void cond_wait(...)**

```
waiter.priority_semaphore = thread_current()->priority;
```

****

```
int priority_sema ( struct list_elem* x, struct list_elem* y,void* aux)
{
  struct semaphore_elem *f = list_entry(x,struct semaphore_elem,elem);
  struct semaphore_elem *l = list_entry(y,struct semaphore_elem,elem);

  if(f->priority_semaphore > l->priority_semaphore)
    return 1;
  return 0;
}
```

- we used this comparator function to insert list ordered fashion in the function { void codn_wait(...) }

MLFQS

-- read theory ..