Project Threads Report

Group 36

Name	Autograder Login	Email
Sidd Shashi	student63	sshashi@berkeley.edu
Matthew Nguyen	student161	matthew.t.nguyen@berkeley.edu
Aman Doshi	student457	aman.doshi@berkeley.edu
Pritish Parmar	student251	pritishparmar@berkeley.edu

Changes

Efficient Alarm Clock

To implement the efficient alarm clock, we made 2 major changes compared to our original design. Both of these changes were inspired by our design review with our TA.

1. Initially, we planned to create a global Pintos list, where each entry was malloced and stored wakeup information for a thread. However, as pointed out to us by our TA, we cannot use **malloc()** while in a disabled interrupt context because **malloc()** requires the use of locks. Instead, we stored the wake-up time as a field inside our thread struct. Our global **sleeping_threads** Pintos list stored thread structs, instead of malloced elements.
2. In the original design, there existed an edge case where a lower priority thread wakes up a higher priority thread in a timer interrupt, but the lower priority thread continues to run after interrupt completion. However, by rules of strict priority scheduling, the higher priority thread should run after interrupt completion. To resolve this issue, we check if any woken up thread has a higher effective priority than the current thread. If so, we call **intr_yield_on_return()** so the current thread can yield and the higher priority thread can be scheduled.

Strict Priority Scheduler

To implement the strict priority scheduler, we made 1 major change compared to our original design. This change was inspired by our design review with our TA.

1. In the original design, there existed an edge case where calling **sema_up()** or **cond_signal()** by a lower priority thread wakes up a higher priority thread, but the lower priority thread continues execution after the respective function is called. However, by rules of strict priority scheduling, the higher priority thread should run when woken up. To resolve this issue, we check if a woken up thread has a higher effective priority than the current thread. If so, we yield the current thread (by calling **intr_yield_on_return()** if in an interrupt context, and **thread_yield()** otherwise), so the current thread can yield and a higher priority thread can be scheduled.

User Threads

In our original design, for pthread user stack page allocation, we planned to assign each pthread a user stack **page_id**, which corresponds to a mapped page in user space (top page = 0, next page = 1, ...). When allocating a new user stack to a new pthread, we would find the next available **page_id** and allocate the corresponding page in user space. This algorithm has a linear runtime. However, this system required extra book-keeping of **page_id** in the pthread struct.

1. Instead, in **setup_thread()**, we iterated through user pages to find the first unset page in the PCB's page directory and assign it to a new pthread. The runtime of the approach is still linear, but does not require the extra bookkeeping of **page_id**. The code is also easier to implement than the previous approach.

When implementing code for creating a new pthread in **pthread_execute()** and **start_pthread()**, we realized two synchronization bugs.

2. In our original design, the **pthread_execute()** function could return before a new pthread is initialized inside **start_pthread()**. If a thread calls **pthread_execute()** and **pthread_join()** consecutively, the **pthread_join()** on the new pthread may fail since the new pthread has not been fully initialized. To resolve this issue, we added a join semaphore initialized to 0. The pthread in **pthread_execute()** downs the semaphore after calling **start_pthread()**. The new pthread in **start_pthread()** ups the semaphore after the new pthread is initialized. Hence, the original pthread exits from **pthread_execute()** after the new pthread has been initialized.
3. In our original design, we shared variables between **pthread_execute()** and **pthread_join()** using a pointer to malloced memory. However, we did not synchronize the freeing of memory between the two pthreads, which caused errors. To resolve this issue, we added a reference count (initialized to 2) and a lock (to synchronize updates to reference count). When a pthread is done with

the shared variables, it decrements the reference count in a critical section. After decrement, if the reference count goes to 0, then the thread must free the malloced memory.

In our original design, to free a pthread's user stack, we simply called **palloc_free_page()** in **pthread_exit()**. However, this led to errors in **process_exit()** when **pagedir_destroy()** was called, since the original user page was still registered inside the PCB's page directory.

4. To resolve this issue, in **pthread_exit()**, we frist called **pagedir_clear_page()** to clear the page from the PCB's page directory before calling **palloc_free_page()**.

Reflection

Overall Working Environment

Very similar to the working environment in project 1, our group was extremely organized and diligent. We consistently worked towards project deadlines by scheduling group meetings on a regular basis. We understood this project was more difficult than the first and accordingly increased our effort to ensure steady progress. Our group text chat was also very active and responsive, and we regularly communicated our availability and progress in code development. Additionally, we wrote most of our code collaboratively in VSCode Live Share and not in isolation. This helped with bouncing ideas off each other and debugging in GDB. The overall group dynamic was positive.

What Went Well

Continuing our clear, respectful channel of communication from project 1 was key to our progress. We always clearly communicated our individual confusion and ideas in group meetings, and felt comfortable working with each other. We also spent nearly all of the debugging and designing time together, making things more efficient than working in isolation.

Moreover, our success in this project attributed to our regular group meetings. During the design document week, our group met every day for a few hours and had productive work sessions. We took breaks afterwards to allow preparation for the midterm, then continued frequent group meetings for code development, allowing us to finish all the code before spring break.

Areas of Improvement
During the design section, we often brainstormed various ideas in our group meetings but rarely wrote them down until we started writing up the final design document. Inevitably, our group failed to detail key ideas in our final design document, despite discussing these ideas in previous group meetings. Moving forward, we aim to draft a summary of our ideas after each meeting, to ensure all of our ideas are represented inside our final design document.

Additionally, when starting to write code for user threads, we first aimed to write enough code so the create-simple Pintos test could pass. To achieve this, we wrote a lot of code that contained various bugs. As a result, we spent a considerable amount of time debugging in GDB before we could pass the create-simple test. Instead, we should have written our own tests in Pintos for user threads, which could pass with fewer lines of new code needed. This would have allowed us to identify bugs faster by testing smaller amounts of code.

Sidd Shashi:
Sidd worked with the rest of the group on the design for each of the three sections, particularly in the efficient alarm clock and user threads sections, and wrote up much of the design doc's efficient alarm clock, user threads, and concept check sections.

He worked with the group to write code for and debug efficient alarm clock, strict priority scheduler, user threads. For efficient alarm clock, he worked on timer_wakeup() and timer_sleep(). For strict priority scheduler, he worked on sema_up() and cond_wait(). For user threads, he worked in varying capacities on pthread_exit(), pthread_exit_main(), start_pthread(), pthread_execute(), setup_thread(), and pthread_join().

Matthew Nguyen:
Matthew worked with the rest of the group to design each of the three sections. However, most of his contributions were made in the design of efficient alarm clock and the pthread syscalls (particularly sys_pthread_create).

For writing code, he worked with the rest of the group to implement all three sections. For efficient alarm clock, Matthew helped implement timer_interrupt. For strict priority scheduler, Matthew helped implement thread_get_effective_priority,

sema_down, and cond_signal. For user threads, Matthew helped implement sys_pthread_create, syscall_sema_up, syscall_sema_down, and syscall_lock_release. Matthew also worked with Aman to write the priority-donate-condvar test case which tests priority donation and if condition variables properly wake up threads with higher effective priority.

Aman Doshi:
In the design document, Aman worked with the rest of the group to design all three sections. For strict priority scheduling, Aman helped develop an algorithm to compute the effective priority of a thread. Additionally, Aman made most of his contributions to the overall design of user threads and user synchronization primitives. Aman also provided support for writing up answers to the concept check questions.

For writing code, Aman worked with the rest of the group to implement all three sections. For efficient alarm clock, Aman helped implement timer_sleep() and timer_wakeup(). For strict priority scheduling, Aman primarily worked on helper functions, sema_up(), and sema_down(). For user threads, Aman primarily worked on pthread-related syscalls. He worked extensively on pthread_execute(), pthread_exit(), pthread_exit_main(), and start_pthread(). He also helped write skeleton code for the user semaphore and user lock syscalls. For testing, Aman worked alongside Mathew to write the Pintos test priority-donate-condvar, which tests if cond_signal() wakes up the thread with the highest effective priority.

Pritish Parmar:
Pritish worked alongside the entire group for the design of all three sections of the project. With special emphasis on efficient alarm clock. As for the coding portion also contributed alongside rest of group during synchronous work sessions through live share with the group on all three sections of the project.

Also worked with group during debugging process using gdb, and particularly worked on setup_stack, syscall_sema and syscall_lock functions like lock acquire, release, sema init, up, down etc.

Testing

Description of Test:

The Pintos test **priority-donate-condvar** tests our implementation of cond_signal to confirm it correctly handles priority donation. The test simulates a priority donation situation and confirms that cond_signal wakes up sleeping threads with the highest **effective** priority.

Overview of Test Mechanics and Expected Output:

The Main thread will create 3 threads, Low, Medium, and High which have low, medium, and high priorities respectively. The Main thread has the lowest overall priority.

Here is a chronological overview of the test.

1. Main: Runs and initializes a resource lock called resource_lock, a condition variable, and a lock that is used by the condition variable.
2. Main: Creates a low priority thread and the low priority thread runs.
3. Low: Acquires resource_lock and calls cond_wait.
4. Main: Creates a medium priority thread and the medium priority thread runs.
5. Medium: Calls cond_wait.
6. Main: Creates a high priority thread and the high priority thread runs.
7. High: Tries to acquire resource_lock and sleeps because it is held by the low priority thread. The high priority thread should donate its priority to the low priority thread.
8. Main: Calls cond_signal(), which should wake up the low priority thread.
9. Low: Releases resource_lock, which wakes up the high priority thread.
10. High: Acquires resource_lock, outputs “Thread H Finished” and exits.
11. Low: Outputs “Thread L finished” and exits.
12. Main: Calls cond_signal() a second time, which wakes up the medium priority thread.
13. Medium: Outputs “Thread M finished” and exits.
14. Main: Outputs “Main thread finished” and exits.

At each step, we will output a message describing what new action occurred and by what thread. For example, we will have messages indicating when threads are created, when threads acquire a lock, when threads call cond_wait, when threads call cond_signal, and when threads exit. We expect the output of our test to align with the order of steps outlined above. Most importantly, we expect the output of our test to have “Thread H finished”, “Thread L finished”, “Thread M finished”, and “Main thread finished” in that order (with additional messages in between). If the output of

our test is not in order, then we know that our implementation of condition variables does not properly prioritize threads with higher effective priority.

Output and Results:

priority-donate-condvar.result

src > threads > build > tests > threads > priority-donate-condvar.result

```
1  PASS
2
```

priority-donate-condvar.result

priority-donate-condvar.output

src > threads > build > tests > threads > priority-donate-condvar.output

```
1  qemu-system-i386 -device isa-debug-exit -hda /tmp/FCW6SUucvE.dsk -m 4 -net none -nographic -monitor null
2  ESC c ESC [?7l ESC [2J ESC [0mSeaBIOS (version 1.15.0-1)
3  Booting from Hard Disk...
4  PPiiLLoo  hhddaa1
5  1
6  LLooaaddiinngg.....................................
7  Kernel command line: -q -sched=prio -f rtkt priority-donate-condvar
8  Pintos booting with 3,968 kB RAM...
9  367 pages available in kernel pool.
10 367 pages available in user pool.
11 Calibrating timer...  133,120,000 loops/s.
12 ide0: unexpected interrupt
13 hda: 5,040 sectors (2 MB), model "QM00001", serial "QEMU HARDDISK"
14 hda1: 311 sectors (155 kB), Pintos OS kernel (20)
15 hda2: 4,096 sectors (2 MB), Pintos file system (21)
16 ide1: unexpected interrupt
17 filesys: using hda2
18 Formatting file system...done.
19 Boot complete.
20 Executing 'priority-donate-condvar':
21 (priority-donate-condvar) begin
22 (priority-donate-condvar) Thread L created.
23 (priority-donate-condvar) Thread L acquired resource_lock.
24 (priority-donate-condvar) Thread L acquired cond_lock and sleeps.
25 (priority-donate-condvar) Thread M created.
26 (priority-donate-condvar) Thread M acquired cond_lock and sleeps.
27 (priority-donate-condvar) Thread H created.
28 (priority-donate-condvar) Main thread calls cond_signal.
29 (priority-donate-condvar) Thread L releases cond_lock after waking up.
30 (priority-donate-condvar) Thread L releases resource_lock.
31 (priority-donate-condvar) Thread H acquired resource_lock.
32 (priority-donate-condvar) Thread H finished.
33 (priority-donate-condvar) Thread L finished.
34 (priority-donate-condvar) Main thread calls cond_signal a second time.
35 (priority-donate-condvar) Thread M releases cond_lock after waking up.
36 (priority-donate-condvar) Thread M finished.
37 (priority-donate-condvar) Main thread finished.
38 (priority-donate-condvar) end
39 Execution of 'priority-donate-condvar' complete.
40 Timer: 62 ticks
41 Thread: 30 idle ticks, 30 kernel ticks, 2 user ticks
42 hda2 (filesys): 3 reads, 6 writes
43 Console: 1651 characters output
44 Keyboard: 0 keys pressed
45 Exception: 0 page faults
46 Powering off...
47
```

priority-donate-condvar.output

Potential Kernel Bugs:

1. If the kernel woke up the sleeping thread with the highest **base** priority instead of the highest **effective** priority, the medium priority thread (instead of the low priority thread) would wake up when the main thread calls cond_signal() for the first time. Consequently, the test would fail. A portion of the failed output is shown below, where the red-highlighted messages are out of order. Note: all messages before this are the same as the expected output.

   "Main thread calls cond_signal"
   "Thread M releases cond_lock after waking up."
   "Thread M finished."
   "Main thread calls cond_signal a second time."
   "Thread L releases resource_lock."
   "Thread H acquired resource_lock."
   "Thread H finished."
   "Thread L finished."
   "Main thread finished."
   "end"

   As observed in the output, the messages are out of order since the medium priority thread is woken up first instead of the low priority thread. This output indicates priority donation from the high priority thread to the low priority thread was not taken into consideration.

2. When the low priority thread releases the resource_lock in step 9, the high priority thread is woken up and is able to acquire the resource_lock. If the kernel does not immediately run the high priority thread after waking it up (by strict priority scheduling), then the test case would fail. A portion of the failed output is shown, where the red-highlighted messages are out of order. Note: all messages before this are the same as the expected output.

   "Thread L releases cond_lock after waking up."
   "Thread L releases resource_lock."
   "Thread L finished."
   "Thread H acquired resource_lock."
   "Thread H finished."
   "Main thread calls cond_signal a second time."
   "Thread M releases cond_lock after waking up."

"Thread M finished."
"Main thread finished."
"end"

As observed in this output, the low priority thread continues execution after releasing the resource_lock, even though the high priority thread has been woken and is ready to run.

Overall Testing Experience

Writing Pintos Tests

We had a good experience writing Pintos tests for this project. This is because writing Pintos tests seems more simple and intuitive after having implemented two tests in Project 1: User Programs. Furthermore, the Pintos specification provides a clear outline on how to write Pintos tests, which helped make the test writing process seamless for us.

Suggested Improvements

Currently, students either run tests individually or run **make check** to run all tests at the same time. However, this testing process takes a lot of time. As a result, we suggest that the Pintos testing system implement a way to easily group specific tests together and allow students to run a specific group of tests. This would significantly decrease the time it takes to test our project.

Additionally, we reiterate the same suggested improvements we outlined in Project 1. Writing tests in Pintos is laborious compared to other testing suits our group has used in other classes (e.g., JUnit in CS 61B). For an improved testing experience, the Pintos testing suite should allow the programmer to embed the test and expected result into one C file, instead of splitting the test across 2 files. This change leads to 2 main benefits.

1. There are less files cluttering the test directory, making it easier to locate tests when bugs occur.
2. Writing and updating tests is easier, since all the work is done in a single C file.

What We Learned

Like in Project 1, we learned a lot from writing tests. For example, we learned that there are several potential bugs that can be very hard to identify during the coding process. This taught us the importance of having good test cases to easily find and address bugs that may easily slip our minds when we are coding. Furthermore, we

learned that there are numerous edge cases that we have to account for when writing a large system like Pintos. This taught us the importance of having a variety of test cases to test these edge cases so that we don’t miss anything. Lastly, we learned about the importance of having test cases that provide useful, detailed feedback/information that will allow us to understand the bugs in our code and address them appropriately.