Project Threads Report

Group 48

Name	Autograder Login	Email
Ben Plate	student219	bplate@berkeley.edu
Jeffrey Liang	student50	jeffreyliang@berkeley.edu
Teddy Cho	student139	teddycho@berkeley.edu
Theophilus Pedapolu	student270	theopedapolu@berkeley.edu

Changes

Efficient Alarm Clock

For our implementation, we kept our algorithms the same but changes some of the structs used to store the sleeping threads. In fact, we decided to remove the `sleeping_thread` struct in favor of storing `wake_ticks` as outlined in our design document in the TCB. One reason we decided to do this was that the list of sleeping threads was now a list containing actual threads, meaning that to put a thread to sleep and wake it up, we just have to call `thread_block` and `thread_unblock` on the list component itself. Furthermore, we also no longer need the `wake_wait` semaphore as thread blocking achieves this functionality. These factors reduced the complexity of our solution, making the implementation much easier to write and understand.

Strict Priority Scheduler

Our implementation of the strict priority scheduler was largely the same as we outlined in our design document. The main change is that the current thread should yield when a higher priority thread is created or woken up. This happens at the end of the functions `thread_create`, `sema_up`, and `cond_signal`.

A very minor change is that we reversed the `list_less_func`'s in our design doc because we anticipated sorting our waiting queues in reverse order. However, this added complexity, so we instead decided to use the function `list_max` to select the highest priority waiting thread.

User Threads

The data structures we outlined in the design document were mostly the same in our final implementation of user threads, except we changed our `join_lock` to be named `pthread_lock` because we changed our synchronization strategy as outlined below. We also introduced a new function `destroy_process` and made specific changes to the algorithms in `setup_thread`, (the helper function for `pthread_create`), `pthread_join`, and `pthread_exit`, `process_exit`

For `setup_thread`, instead of malloc'ing a page for each new user thread which is an evasive and incorrect way to get memory, we instead call `setup_stack` which uses `palloc_get_page` to get a page from the user pool and `pagedir_set_page` to put it in the process's page directory. Additionally, we modified `setup_stack` so that it gets the first available virtual page below `PYHS_BASE` and installs it. This is in case some previous user thread has already terminated; we can use the space left behind from freeing it's userspace stack so that fragmentation is minimized.

For `pthread_join`, we modified it so that the calling thread frees the corresponding `user_thread` struct of the thread it waited on and removes it from the list of `user_threads` in the PCB. This logic happens after the thread calls `sema_down`, i.e. after the waited thread has exited, and it conserves heap space as we don't need the waited thread's `user_thread` struct after it has terminated.

We modified `process_exit` so that only the main thread is allowed to exit the process and any non-main thread that calls process_exit designates the main thread as the exiter and exits the thread normally. To do this, upon a call to `process_exit`, we first set the exit flags in the PCB, `pcb->exit_status->exited = true` and `pcb->exit_status->status = status`. Then, if the caller is the main thread, we simply invoke `pthread_exit_main`. Otherwise, we invoke `pthread_exit`. The logic for `pthread_exit` remains the same. However, for `pthread_exit_main`, after the main thread has finished joining on all threads, we check to see if any thread has called process_exit (i.e. the PCB exit flag is set). If so, we call a separate function `destroy_process` which contains the logic for our original `process_exit` function, freeing all necessary resources and exiting with the status code set in the PCB. If the

PCB exit flag is not set, we first set the exit flag to true and the status code to 0, then call `destroy_process`. The last thing `pthread_exit_main` does is call `thread_exit()` This change was necessary because we ran into a triple page fault error in the case that the main thread is stuck joining on a user thread in `pthread_exit_main` and the user thread then calls `process_exit`. Because the PCB is destroyed, when the main thread was scheduled again, this caused memory errors.

Additionally, we changed our synchronization strategy by putting a process-level lock (called `pthread_lock` in the PCB) on all pthread syscalls. As discussed in the design review with our TA, this is necessary since the PCB's page directory and user threads list might change between pthread syscalls, and we also need synchronization to ensure that if a thread exits and frees its userspace stack page, that page is used when we create a new user thread with `pthread_execute`Furthermore, we make sure to release `pthread_lock` before we `sema_down` on a thread in `pthread_join` and `pthread_exit_main` so that the thread waited on can acquire that lock when it calls `pthread_exit`. Synchronization for user-level locks and semaphores was unchanged.

Reflection

Each of our group members contributed to the following parts of the project.

- Ben helped write the Efficient Alarm Clock section and wrote the rationale for the Strict Priority Scheduler in the design doc in addition to writing the additional test cases, both for question 5 on the design doc as well as the testing section for the report. This also includes the code implementations in the PintOS tests. Ben also implemented the Efficient Alarm Clock in PintOS and helped debug edge cases in `process_exit` with multiple user threads.
- Jeffrey helped write the Efficient Alarm Clock and Strict Priority Scheduler sections of our design doc, contributed ideas to the User Threads design, answered questions 1-4 of the concept check, implemented the strict priority scheduler, wrote the Strict Priority Scheduler section of this document, and changed Theo's implementation of user threads to pass the `join-exit-2` test.
- Teddy contributed ideas to the strict priority scheduler and participated in the design review.

- Theo wrote the User Threads portion of the design doc, contributed ideas to the strict priority scheduler, implemented the pthread syscalls and the user-level locks and semaphores syscalls, and wrote the User Threads section of this final report

Our working environment was generally productive, and we were able to comfortably meet the deadlines for this project. We met a couple of times throughout this project: twice to discuss our ideas for the design, once to coordinate the implementation, and once to debug the last test that we were failing. We usually used these meetings to discuss ideas so that we had a general approach to each part of the project and then divide work so that we could work offline more efficiently. During the implementation phase, we each worked on separate branches and merged our changes to the main branch when they were complete. This worked effectively for this project because each of its parts could be completed largely in isolation. Because the user threads implementation was significantly more involved than other parts of this project, it might have been helpful to split up work or pair program for that section, but we did work together to modify our design and implementation to pass the final test we were failing.

Testing

We created the `join-exit-3` test in `tests/userprog/multithreading` to test how the kernel handles a process exiting with another thread (that isn't main) joining on the thread calling `exit`. The purpose of this is to ensure that all threads belonging to a process are killed and freed when `exit` is called, including ones that are blocked because they are joining on another thread.

The test works by creating two threads from main: thread A, and thread B. Thread A joins with the main thread and thread B joins with thread A. After creating the two threads, the main thread calls `pthread_exit`, which should begin running thread A, which is now no longer blocked (while thread B is blocked waiting on thread A). Thread A calls `exit(162)`, which should immediately exit the process and, because thread B is waiting to join on thread A, stop thread B from running any more code. This means that anything supposed to run after thread B joins on thread A should not run. We also call `exit(9583)` from thread B after the `pthread_check_join`, which again should not be run and the exit status of this process should be 162.

The output of the test should be as follows:

```
(join-exit-3) begin
(join-exit-3) Main starting
(join-exit-3) Thread starting
join-exit-3: exit(162)
```

This captures that the main thread starts, thread A starts (detected by printing "Thread starting"), and then the process exits with status 162. We have `fail` test functions after `pthread_check_join` in thread B and `exit(9583)`, which should not affect the exit status of the process.

One bug that could occur is that thread B tries to finish what it's doing when thread A calls `exit`. This is not intended behavior, and the test would call `fail` if this were to happen. Another bug that is tested is the case of two calls to `exit` being made in different threads. When thread A calls `exit(162)`, this should not allow any other threads in the process to call `process_exit` and the status should not change if any other threads call `exit`. If this were the case, then we might see `join-exit-3: exit(9583)` in our output, since thread B calls `exit(9583)` after thread A.

The test-writing experience for PintOS is easy and and we feel that there isn't much overhead in creating tests. However, one negative experience we did come across was that when we tried to add one of our tests, the autograder rejected the entire project, saying that we should not have edited the file we had to add the test to for it to run. This wasn't much of a problem as we could still run the test locally, but it did mean that we couldn't commit the `test.c` and `test.h` files that told `pintos-test` to register the tests.