INF333 2024-2025 Spring Semester

Pintocchio
Sude Melis Pilaz <22401992@ogr.gsu.edu.tr>
Ali Burak Saraç <21401932@ogr.gsu.edu.tr>

Homework I
Design Document

Please provide answers inline in a `quote` environment.

1 Preliminaries

Q1: If you have any preliminary comments on your submission, notes for the TAs, or extra credit, please give them here.

> On our initial plans, we aimed to implement a heap for ordering our sleeping threads by their wake-up times. Thus, as a sketch, we designed our own list structure to keep it separated from other lists. However, we realized that heap structure could not dynamically reallocate and we had to change our design to a simple linked list. At this stage, it would cause a lot of complexities to switch to the already implemented list structure. Therefore, we decided to keep our design as it is and implement the linked list structure. We are aware that our design is not the most efficient one, but we believe that it is the most suitable one for our current implementation.

Q2: Please cite any offline or online sources you consulted while preparing your submission, other than the Pintos documentation, course text, and lecture notes.

- Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating System Concepts* (10th ed.). Wiley.
- Stanford University. (2024). *CS212 Operating Systems Lecture Notes.* https://www.scs.stanford.edu/24wi-cs212/notes/

- University of California, Berkeley. (n.d.).
  Operating Systems and Systems Programming Webcast Lectures.
  https://archive.org/details/ucberkeley-webcast
- jenson.gitbooks.io
  https://jeason.gitbooks.io/pintos-reference-guide-sysu/content/priority_donating.html

2 Sleep

2.1 Data Structures

Q3: 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.

static struct ordered_list sleeping: ordered list consisting of sleeping threads and their wake up times

struct ordered_list: custom ordered list containing wake up time information for each thread
struct ordered_list_elem *head: Pointer to the first element (earliest wake-up time)

struct ordered_list_elem: list element for ordered list
struct thread *t : pointer to the sleeping thread
int64_t wake_up_time: time when the thread should wake up
struct ordered_list_elem *next : pointer to the next element in the list

2.2 Algorithms

Q4: Briefly describe your implementation of `thread_join()` and how it interacts with thread termination.

In Pintos, we did not use a thread_join() function. Instead, parent thread blocks itself until its child finishes execution. This blocking is done using thread_block() and later unblocking with thread_unblock() when the child terminates. We implemented a sleep list to keep track of sleeping threads. When a thread calls thread_sleep, the thread is added to the sleep list with its wake-up time. The thread gets blocked and starts waiting. The

timer_interrupt function checks the sleep list each tick and wakes up the threads when their time comes. The thread is then removed from the sleep and waiting lists. The thread is then unblocked and added to the ready list. This method avoids busy waiting and allows the thread to sleep until its wake-up time.

Q5: What steps are taken to minimize the amount of time spent in the timer interrupt handler?

By using an ordered list, the interrupt handler efficiently checks only the thread with the shortest wake-up time, avoiding unnecessary comparisons. Any unnecessary or complex operation is avoided to keep timer interrupt as simple as possible.

2.3 Synchronization

Q6: Consider parent thread `P` with child thread `C`. How do you ensure proper synchronization and avoid race conditions when `P` calls `wait(C)` before `C` exits? After `C` exits? How do you ensure that all resources are freed in each case? How about when `P` terminates without waiting, before `C` exits? After `C` exits? Are there any special cases?

When P calls wait(C) before C exits, P will be blocked until C exits. C wakes up normally when its wake-up time is reached. When C wakes up, it will signal its parent thread to wake up.

When P calls wait(C) after C exits, C would be already exited and the parent thread would read the exit status of C and continue its execution. C's resources would be freed when P reads the exit status of C.

When P terminates without waiting, before C exits, C detects that its parent thread has exited. C will clean up its resources itself.

When P terminates without waiting, after C exits, P reads and frees C's exit status and continues its execution.

Q7: How are race conditions avoided when multiple threads call `timer_sleep()` simultaneously?

Race conditions are avoided by disabling the interrupts in critical operations. With this approach it's ensured that no other thread or interrupt can modify the sleeping list or the state of the current thread during operations. Inserting threads to sleep list and blocking thread operations can be performed safely. This way, the list integrity is maintained while multiple threads are calling timer_sleep() simultaneously.

Q8: How are race conditions avoided when a timer interrupt occurs during a call to `timer_sleep()`?

During critical operations, such as modifying the sleep list, interrupts are disabled to prevent race conditions during timer_sleep(). This ensures that the timer interrupt handler does not interfere with the sleep list or change the state of the sleeping threads. The thread is blocked only after the list update is complete, ensuring the timer interrupt can properly wake it later. By disabling interrupts inside thread_unblock which is called by timer_interrupt, we ensure that atomicity of both functions are preserved.

2.4 Rationale

Q9: Critique your design, pointing out advantages and disadvantages in your design choices.

As stated above, we initially aimed to implement a heap structure for the ordered list. However, we had to change our design to a simple linked list due to memory overflow. This change caused our design to be more difficult to understand and decreased readability. If we hadn't planned to implement a heap structure from the beginning, we could have used the already imple mented list structure in Pintos. In result, our wake_up_time also became a separate variable, rather than a property of the thread, being different from other elements. Besides this, in thread_sleep function, we preferred to disable interrupts, while the alternative would be to use a lock. We believe that disabling interrupts is more efficient and less complex than using a lock.

An advantage of our function is that timer_interrupt function is kept simple and efficient. By using an ordered list, the interrupt handler efficiently checks only the thread with the shortest wake-up time, avoiding unnecessary comparisons, resulting in O(1) time complexity.

3 Priority Scheduling

3.1 Data Structures

Q10: 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

Our code for reading and writing user data from the kernel is primarily implemented in the `sys_io` function, which handles both read and write

operations with a unified approach:

1. **Parameter handling**: The function takes a file descriptor (`fd`), a buffer pointer, a size, and a boolean flag indicating whether this is a write operation.

2. **Special case for console output**: If the file descriptor is 1 (stdout) and the operation is write, we directly call `putbuf()` to output to the console after acquiring the file system lock.

3. **File descriptor lookup**: For other file operations, we: - Get the current thread - Acquire the file system lock to ensure thread safety - Iterate through the thread's open files list to find the matching file descriptor - Release the lock after finding the file (or not finding it)

4. **Error handling**: If no matching file descriptor is found, we return -1 to indicate failure.

5. **Performing I/O**: After finding the file: - We reacquire the file system lock - Based on the `is_write` flag, we either: - Call `file_write()` to write data from user space to the file - Call `file_read()` to read data from the file into user space - Release the lock after the operation

6. **Memory safety**: Before accessing user memory, the `syscall_handler` validates all user-provided addresses using `check_user_address()`, which verifies both that the address is in user space and that it's properly mapped in the page directory.

This implementation ensures thread-safe file operations while properly handling the transition between kernel and user memory spaces.

Q11: Describe the sequence of events when a call to `lock_acquire()` causes a priority donation. How is nested donation handled?

When lock_acquire is called and the lock is already held by another thread, and the current thread has a higher priority than the holder, donate_priority is called. donate_priority is a function that traverses the chain of locks that the current thread is waiting on and donates priority as necessary. Nested donation is handled using this loop in donate_priority. Priority donation is handled for the current thread and then the thread that is holding the lock. This process continues until the maximum depth is reached. handle_priority function is called before donation to ensure that the priority is recalculated correctly.

Q12: Describe the sequence of events when `lock_release()` is called on a lock that a higher-priority thread is waiting for.

When lock_release is called, the lock is removed from the held locks list of the current thread. The priority of the current thread is recalculated using the

handle_priority function. handle_priority function recalculates the priority by iterating through the locks that the current thread holds and finding the maximum priority of the threads waiting on the locks. The waiters list of the semaphore is used for finding the waiting threads. The priority of the current thread is then updated. The lock holder is set to NULL and the semaphore is upped.

3.2 Synchronization

Q13: Describe a potential race in `thread_set_priority()` and explain how your implementation avoids it. Can you use a lock to avoid this race?

A potential race occurs when two different threads try to set their priority at the same time. If both threads are trying to access shared data like the ready_list their operations can interfere with each other.

The solution we implemented is to disable interrupts during the critical section of the thread_set_priority function. This way, we ensure that no other thread or interrupt can modify the shared data while the priority is being set.

We can't use a lock to avoid this race. Because when a thread which helds helds locks calls thread_set_priority but a lower priority thread is already holding priority lock, the thread_set_priority function will be blocked. This will cause a deadlock.

3.3 Rationale

Q14: Why did you choose this design? In what ways is it superior to another design you considered?

Our design uses held_locks list and waiting_on_lock in each thread to keep track of the priority donations. At first we considered having a waiting threads list but even then we needed the locks associated with the threads. But having a separated waiting threads list would duplicate the information and make the implementation more complex. Our design is superior to this approach because it is more efficient and simple.

4 Advanced Scheduler

4.1 Data Structures

Q15: 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.

int nice: determines how "nice" the thread should be to other threads. Decreases the priority of a thread and causes it to give up some CPU time it would otherwise receive.
fixed recent_cpu: measure how much CPU time each process has received "recently."
static fixed load_avg: moving average of the number of threads ready to run.
typedef int64_t fixed: type definition for fixed point numbers

4.2 Algorithms

Q16: 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:

priority, recent_cpu and load_avg are calculated by formulas below:
priority = PRI_MAX - ROUND((recent_cpu / 4) - (nice * 2))
recent_cpu = (2 * load_avg) / (2 * load_avg + 1) * recent_cpu + nice
load_avg = (59/60) * load_avg + (1/60) * ready_threads

Priority is updated every 4 ticks, and recent_cpu and load_avg is updated every second meaning every 100 ticks. The initial value of load_avg and recent_cpu is 0. Besides these formulas, recent_cpu is incremented by 1 for the running thread every timer tick.

timer ticks	recent_cpu A	recent_cpu B	recent_cpu C	priority A	priority B	priority 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

Q17: 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?

> The scheduler did not specify how to handle situations where priorities of two threads are equal, but we resolved them by assuming round-robin scheduling. This matches the behavior of our scheduler. Another ambiguity was that the priority calculation should be made every 4th tick and recent_cpu calculation formula should be applied every tick, but we have no directives on doing it before or after scheduling and also which one to update first at every 4th tick. We decided to update the recent_cpu first and then calculate the priority for that ticks and update these values before the next scheduling.

Q18: How is the way you divided the cost of scheduling between code inside and outside interrupt context likely to affect performance?

> The implementation of the scheduler divides scheduling costs by handling the lighter operations in the interrupt context, such as updating the recent_cpu, tracking ticks and making preemptions, while leaving heavier operations outside the interrupt context. However, some expensive computations like recalculating recent_cpu and load_avg, and sorting the ready list are done within the interrupt context. This doesn't affect the performance for the small number of threads, but it may cause performance issues when the number of threads increases. With more time and experience with Pintos, we could optimize the scheduler to improve performance, moving these operations outside the interrupt context.

4.3 Rationale

Q19: 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?

> We designed our project for simplicity and modularity. We implemented fixed-point math as a dedicated abstraction layer using macros. This implementation is more modular, easier to maintain and debug, and accessible from all the files. Also, it prevents code duplication and makes the code more readable. However, we could improve our design by optimizing the scheduler to improve performance. We could move some operations outside the interrupt context to reduce the time spent in the interrupt handler. We could also optimize the priority calculation and thread sorting to improve the performance of the scheduler. With more time and experience with Pintos, maybe with some guidance from the TAs, we could refine our design to make it more efficient and optimized.

Q20: 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?

> We decided to implement fixed-point math as a dedicated abstraction layer. Using macros and a fixed_point.h file containing the necessary definitions. We decided to use macros instead of functions because macros are faster and fixed point operations are generally handled when timer_interrupt handler is called so we need to make it as fast as possible. This way, we can easily change the implementation of fixed-point math if needed. This implementation is more modular, easier to maintain and debug and accessible from all the files. Also it prevents code duplication and makes the code more readable.

5 Survey Questions

Answering these questions is optional, but it will help us improve the course in future quarters. Feel free to tell us anything you want–these questions are just to spur your thoughts. You may also choose to respond anonymously in the course evaluations at the end of the quarter.

Q1: In your opinion, was this assignment, or any one of the three problems in it, too easy or too hard? Did it take too long or too little time?

> We sincerely believe that the assignment was very challenging. The problems were complex and required a deep understanding of the Pintos operating system. The assignment took a lot of time to complete, and many problems we encountered were because of our lack of experience with Pintos. The main problem for this project was the lack of guidance and examples. Not only the assignment but also the Pintos documentation was not very helpful. We believe that the assignment would be more manageable if there were more examples and guidance on how to approach the problems.

Q2: Did you find that working on a particular part of the assignment gave you greater insight into some aspect of OS design?

> Interestingly enough, we found that working on the priority scheduling part of the assignment gave us greater insight into the synchronization and scheduling aspects of OS design. We learned how to handle priority donations and how to implement a priority scheduler. However, probably speaking on behalf of all groups, we believe that the assignment would be more beneficial if it were divided into smaller parts and more guidance was provided on how to approach the problems. More hints on the problems we were expected to encounter and how to solve them would be very helpful and let us focus on the main problems of the assignment rather than getting stuck on the details.

Q3: Is there some particular fact or hint we should give students in future quarters to help them solve the problems? Conversely, did you find any of our guidance to be misleading?

It would be very helpful if the lab projects we do in the course were more aligned with the assignment. We believe that the labs should be designed to prepare us for the assignment and help us understand the concepts better. For example, the labs could be designed to help us understand the Pintos operating system better and how to approach the problems in the assignment. For example, installing Pintos or writing Git commands were helpful for the project. But the timing for the git commands lab was not very good. If it were done before, say the first week of the project, we could benefit from the git more effectively.

Q4: Do you have any suggestions for the TAs to more effectively assist students, either for future quarters or the remaining projects?

Please provide more examples and guidance on how to approach the problems. Sparing some time to talk, debate on the project in the classroom could be an effective way to help students understand the concepts better, making the lessons more interactive. Having short conversations about the common problems, key aspects of the project before or after the lessons could prove to be very beneficial. This is a popular method in many universities and it is proven to be very effective.

Sparing some time to talk, debate on the project in the classroom could be an effective way to help students understand the concepts better, making the lessons more interactive. Having short conversations about the common problems, key aspects of the project before or after the lessons could prove to be very beneficial. This is a popular method in many universities and it is proven to be very effective.

Q5: Any other comments?

Thank you for taking an interest in our feedback. We believe that the course could be steered in a more beneficial direction if the suggestions we provided above were taken into consideration. If the difficulty of the assignments could be balanced, we believe we can learn many things from this course.

Out of topic, but another feedback we would like to give is about the lab sessions. Currently, the lab sessions try to cover different topics in a short time. These topics are very important and should be covered in detail. The topics we discovered so far (Linux, Assembly, Git, etc.) are huge and require a lot of time to understand clearly, they are fundamentals of many other topics. We believe that the lab sessions should be more focused on only a few of them and should be more detailed. This way, we can understand the topics better and benefit more from the labs. And it would be wonderful if our lab sessions were targeting the problems we would encounter in the projects. This way, we can understand the concepts better and apply them to the projects more effectively.