IMPLEMENTING FEATURES TO PINTOS

Karan Jain, Kethireddy Harshith Reddy, Lakshya Kumawat
B20AI016, B20AI018, B20AI019
Indian Institute Of Technology Jodhpur
Principles Of Computer Systems - 2 (CSL2090)
Course Project

Pintos is a platform for operating systems based on the 80x86 architecture. It has kernel threads, user programme loading and execution, and a file system, but it does it in a very simple way. It might take a long time to run Pintos on bare metal machines each time. Instead, we used an x86 emulator, specifically QEMU, which allows us to run a number of guest operating systems.

Pintos already has a simple threading system, a basic programme loader, and a simple file system, however it has a few limitations which as follows:

1) The alarm clock function isn't really effective. This project simply requires interrupt handlers to access a little amount of thread information. The timer interrupt is required to wake up sleeping threads for the alarm clock. The timer interrupt in the advanced scheduler requires access to a few global and per-thread variables.

2) Pintos have priority inversion and no priority scheduler. When a high priority job is indirectly preempted by a low priority activity, this is known as priority inversion. The low priority process, for example, retains a mutex that the high priority activity must wait for in order to continue running.

3) Pintos also lacks effective schedulers to keep processes organized.

4) We allow many processes to execute at the same time. Each process is connected by a single thread (multithreaded processes are not supported). User programmes are written as though they had complete control over the computer. To preserve this illusion, we must manage memory, scheduling, and other state effectively while loading and running numerous processes at the same time.

5) Normal C programmes can run on Pintos as long as they fit in memory and utilise just the system calls you implement. Because none of the system methods necessary for this project allow for memory allocation, malloc() cannot be implemented. Pintos also can't execute floating-point applications because the kernel doesn't preserve and restore the floating-point unit of the CPU when switching threads.

We tried to resolve these issues by implementing the following tasks:

Task 1: Efficient Alarm Clock

We used *thread_bock()* and *thread_unblock()*, rather than introducing a new lock synchronization.
We used the original *elem* for usage in *sleeping_list* and introduced a comparator function for removing threads from *sleeping_list.*

Task 2: Priority Scheduler

To implement the ready and waiting list queues, follow these steps: We may just preserve the current way of popping the lists to find the thread with the highest effective priority by keeping the ready/waiting lists in sorted order. We can maintain a round robin schedule by simply popping from the sorted list, while avoiding saving runtimes and other such things, because we consider insertion as a LIFO queue, where threads of the same priority are internally ordered by last run time. Another approach we investigated was to create a 64-bit array of pointers to linked lists, with the i'th linked list being a LIFO queue of threads of the i'th priority, in round-robin order. However, simply for keeping the pointers, which in the case of semaphores might be mostly empty (wasted space), a minimum of 256 bytes of memory would be required, and there would be one of these for each semaphore, which might take up a significant amount of space. If the resource is in the kernel stack or static memory, this may be unsatisfactory due to memory constraints.

Another option discussed was identical to the preceding, except we'd use a linked list with a maximum size of 64 nodes that would link to "priority nodes." There would be no node for a given priority if there were no threads of that priority. The existing nodes would point to their own linked lists, forming a round-robin LIFO queue. This might solve the problem of wasted space, but it would be difficult to implement. Our chosen solution is accurate, but it adds to the time complexity of traversing the list for insertion and necessitates a bespoke implementation of insert in sorted order ().

We investigated a recursive solution for implementing layered priority donation, but recognised that the stack would grow unbounded in the case of a long chain of lock dependents, so we decided on an iterative alternative. The proposed approach of saving dependent thread pointers was simple and basic, but it needed us to construct a reference to the lock that a thread is waiting on within the thread struct in order to always have a reference to the next child while following the dependency chain (since the lock already has a reference to its owner).

Task 3: Multi-Level Feedback Queue Scheduler (MLFQS)

Multilevel Feedback Queue Scheduling (MLFQS) keeps analyzing the behavior (time of execution) of processes and according to which it changes its priority. MLFQS is implemented to avoid the starvation of lower priority threads/processes.

We update the thread MLFQS priority void *thread_update_mlfqs_priority*(*void*). Update the priority for each and every thread, like *priority = PRI_MAX - (recent_cpu / 4) - (nice * 2)*.

We update the *recent_cpu* void *increment_recent_cpu(void)* by increasing *recent_cpu* by 1 for the running threads, unless the idle thread is running.

We Initialized *initial_thread's recent_cpu* to 0 and *nice* to 0, and *load_avg* to 0 in *thread_init(void)*. Inherit *recent_cpu* and nice in *thread_create()* in *thread_create()*.

Task 4: Argument Passing

In the current version of pintos, *process_execute*() does not support argument passing to the new processes. So, we implemented this functionality by extending *process_execute*() so that instead of simply taking a program file name as its argument, it divides it into words at spaces. In which the first word is the file name and the second word is the first argument, the third word is the second argument and so on.
We implement a functionality to acquire the file system lock for the file system function call in *load()* and *setup_stack()*.

Task 5: Syscall

We implement the system call handler in "userprog/syscall.c". We implemented the following system calls.

- System Call *void* ***halt****(void)*: This function terminates the pintos by calling *power_off()*.

- System Call *void* ***exit****(void)*: This syscall function terminates the current user program, returning status to the kernel. If the process's parent waits for it, this is the status that will be returned, if it returns 0 means the task is successful and if it returns some non-zero value, it means there is some error.

- System Call: *pid_t* ***exec*** *(const char *cmd_line)*: This syscall function runs the executable whose name is given in *cmd_line*, passing any given arguments, and returns the new process's program id (pid). It returns the value pid -1, otherwise it is not a valid pid, if the program cannot load or run for any reason. We use synchronization to ensure that the parent process cannot return from the exec until it knows if the child process successfully loaded its executable or not.

- System Call: *int* ***wait*** *(pid_t pid)*: If process pid is still alive, waits until it dies. Then, returns the status that pid passed to exit, or -1 if pid was terminated by the kernel. If pid does not refer to a child of the calling thread, or if wait has already been successfully called for the given pid, returns -1 immediately, without waiting.

- System Call: *bool* ***create*** *(const char *file, unsigned initial_size)*: This syscall function creates a file by taking two arguments, first is file name(*file*) and second is size of file in bytes (*initial_size*). It will reuters true if the file is created successfully and false if not. But it will not open the file.

- System Call: *bool* ***remove*** *(const char *file)*: This syscall function will delete the file and take one argument which is the name of the file. It will return true if the file is successfully deleted and return false if not.

- System Call: *int* ***open*** *(const char *file)*: This syscall function will open a file by taking filename(*file*) as argument. It will return a nonnegative integer handle called a "file descriptor" (fd), or -1 if the file could not be opened.

- System Call: *int* ***filesize*** *(int fd)*: This syscall function will return the size of the file open as fd.

- System Call: *int* ***read*** *(int fd, void *buffer, unsigned size)*: This syscall function will read the bytes of *size* from buffer to the open file *fd.* It will return the number of bytes read or -1 if any error occurred.

- System Call: *int* ***write*** *(int fd, const void *buffer, unsigned size)*: This syscall function writes size byte from buffer to open a file in *fd*, returns the number of bytes which are actually written to the buffer or -1 if any error occurred.

- System Call: *unsigned* ***tell*** *(int fd)*: This syscall function returns the position of the next byte to be read or written in open file fd, expressed in bytes from the beginning of the file.

- System Call: *void* ***close*** *(int fd)*: This syscall function will close the *fd*.

Results and Test Cases

All the parts according to the documentation given in the Stanford documentation of Pintos and then in order to access our implementation, we tried running our Stanford test cases on them. After multiple improvements, the best result we got was that 105 out of 107 test cases passed.

```
SUMMARY BY TEST SET

Test Set                                Pts Max  % Ttl  % Max
--------------------------------------- --- --- ------ ------
tests/userprog/Rubric.functionality     102/108  33.1%/ 35.0%
tests/userprog/Rubric.robustness         88/ 88  25.0%/ 25.0%
tests/userprog/no-vm/Rubric               1/  1  10.0%/ 10.0%
tests/filesys/base/Rubric                30/ 30  30.0%/ 30.0%
--------------------------------------- --- --- ------ ------
Total                                            98.1%/100.0%
```

```
SUMMARY BY TEST SET

Test Set                                Pts Max  % Ttl  % Max
--------------------------------------- --- --- ------ ------
tests/threads/Rubric.alarm               18/ 18  20.0%/ 20.0%
tests/threads/Rubric.priority            38/ 38  40.0%/ 40.0%
tests/threads/Rubric.mlfqs               37/ 37  40.0%/ 40.0%
--------------------------------------- --- --- ------ ------
Total                                           100.0%/100.0%
```