Project 1 Design Doc

Group 4

July 7 2020

Contents

1 Task 1

1.1 Data Structures and Functions

We create a new function, `parse_file_name()`, in `process.c` that uses `strtok_r()` to tokenize the filename passed into it.

1.2 Algorithms

- Step 1: Take the input string `file_name`, a complete command-line argument, and parse it using `parse_file_name()`, delimiting by whitespace. Store the name of the executable into its own buffer. This buffer is used as the input to `load()`.
- Step 2: Calculate the length of the `file_name` string using `strlen()` and call `load()` to initialize the value of the `if_.esp` variable.
- Step 3: Decrement the `if_.esp` variable by the smallest multiple of 16 that is still larger than the length of the input string (e.g.: if the `file_name` string is 17 bytes long, decrement `if_.esp` by 32 bytes). This is done in order to preserve stack alignment.
- Step 4: Create a `char` buffer at the new memory address stored in `if_.esp` and fill it with the contents of the `file_name` string, replacing every whitespace with a null terminator. In the same loop, increment the value of the local `argc` variable in order to keep track of the number of arguments.
- Step 5: Decrement the value of `if_.esp` by the smallest multiple of 16 that is still larger than 4 * (`argc` + 1). This space will be used to store distinct pointers to the different arguments that will be found in the user processes' `char *argv[]`.
- Step 6: Fill in the newly allocated space with pointers to the start of each command line argument stored in step 4.
- Step 7: Decrement the value of `if_.esp` by 12 to create space for the user processes' `int argc` and `char *argv[]` variables. Store a pointer to the zeroth pointer that was created in step 5 in `if_.esp + 8` (this will be the user's `char *argv[]`). Store the value of `argc` computed in step 4 in `if_.esp + 4`. The last 4 bytes allocated are used as a dummy saved `eip`—a fake return address that won't be used.
- Step 8: Begin execution as normal by simulating a return from interrupt.

Failure checks and edge cases: If there is ever a `load()` failure, the function frees the page memory and terminates before doing any of this work. In the case where the command line argument is formatted oddly, (e.g.: spaces before the first argument or multiple spaces between arguments) the null terminators negate any weird spacing. Invalid arguments for the user process should be handled by the user process itself.

1.3 Synchronization

Synchronization isn't a concern for this task. Any process can start any process at any time.

1.4 Rationale

Since this task is relatively straightforward, we simply went with the first approach that we came up with and didn't find it necessary to consider alternatives. This design for completing task 1 requires multiple passes over the arguments, but it is the most conceptually intuitive approach. Building the stack from top to bottom incrementally and in multiples of 16 ensures stack alignment and keeps the model consistent. This approach has a runtime of $\Theta(n + c)$, where n is the number of arguments and c is the number of characters in the command. This approach also has spatial complexity $\Theta(n + c)$. This design is generalized to work for all user processes.

2 Task 2

2.1 Data Structures and Functions

For this task, we create a new data structure, `struct wait_status`, in `/threads/thread.h`.

```
struct wait_status {
    struct semaphore wait_sema;
    int exit_code;
    int reference_count;
    struct lock count_lockula;
    pid_t child;
    struct list_elem elem;
};
```

This `wait_status` struct works as an intermediary between parent and child processes. We include a semaphore (to signal between parent and child), the exit code of the child process, a reference count to ensure that the struct gets freed exactly once, a lock to protect the reference count in case the parent and child exit concurrently, the `pid` of the child process, and a `struct list_elem` to facilitate creating a linked list of `wait_status` structs.

Additionally, we modify `struct thread` in `/threads/thread.h` to include `struct list waits`, a `list` of `wait_status` structs so that we can keep track of parent-child relationships. We also add an `int exit_status` initially set to `-1` under `#ifdef USERPROG`. We `typedef pid_t` to `tid_t` for consistency.

Lastly, we add the following process syscall implementations to `/userprog/syscall.c`, using if statements to implement each syscall:

```
int practice(int i)
void halt(void)
pid_t exec(const char *cmd_line)
int wait(pid_t pid)
```

2.2 Algorithms

- `int practice(int i)`: Increment `int i` by 1 and return by setting `f->eax` to the incremented value.
- `void halt(void)`: Call `shutdown_power_off()`
- `pid_t exec(const char *cmd_line)`:
  - Step 1: Call `sema_down()` in `process_execute()` to wait for `thread_create()` to be successful.
  - Step 2: Within `process_execute()`, marshall `fn_copy`, which is passed into `thread_create()`, using the `parse_file_name()` function, and use the `argument_check()` function to test the validity of the pointers. If the pointers are invalid, return `-1` and kill the process.
  - Step 3: In the parent process, instantiate a `struct wait_status`, initializing `wait_sema` to `0`, `reference_count` to 2, `count_lockula` using `lock_init`, and `child` to the `pid` that we obtained from `thread_create()` in `process_execute()`, and push it onto the initialized `struct list waits`.
  - Step 4: In the child process, traverse the `list` such that the child process points to its corresponding `wait_status` struct, and call `sema_up()` to signal to `process_execute()` that it is done setting up.
  - Step 5: Return by setting `f->eax` to the `pid` of the child process.

- `int wait(pid_t pid)`:
  - Step 1: Find the child process whose `pid` matches the one passed into the function by traversing the `wait_status` list and comparing `wait_status->child` to the given `pid`.
  - Step 2: Run failure checks as detailed in the failure check section below. If the checks fail, or if `wait()` has been called more than once, return an error and kill the calling process.
  - Step 3: Pass the validated `pid` into `process_wait()`, calling `sema_down()` on `wait_status->wait_sema` to cause the parent process to wait until the child has finished executing.
  - Step 4: The child process updates the `exit_code` of the `struct wait_status` it is pointing to in the list of `wait_status` structs with its exit code.
  - Step 5: Acquire `count_lockula`, and decrement `reference_count` by 1. If `reference_count` is `0`, deallocate the `struct wait_status`, and remove it from the `list` of `wait_status` structs.
  - Step 6: If `reference_count` is `0` when the child process finishes executing, the parent is dead and the *child* deallocates the `struct wait_status`, removing it from the `list` of `wait_status` structs.
  - Step 7: Call `sema_up()` on `wait_status->wait_sema`.
  - Step 8: Return the exit code of the child process by setting `f->eax` to `wait_status->exit_code`.

Failure checks and edge cases: We use the second method detailed in the project spec to verify pointer validity. We use the `argument_check()` function defined in task 3 for this.

2.3 Synchronization

The `wait_status` structs are shared data structures, and as such, access to them must be synchronized. We implement synchronization by using locks and semaphores. In particular, we use a lock to protect `wait_status->reference_count` in case both the parent and child process exit concurrently, and a semaphore to ensure that a parent continues executing after its child processes have exited.

2.4 Rationale

For this task, the primary challenge was figuring out a way to track the relationships between parent and child processes. We used a `list` of the `wait_status` structs that Sam talked about in lecture for this. We went with the most intuitive solution that we could think of for this task and tried to minimize extraneous changes to the kernel. We considered some alternatives to the design we came up with for tracking parent-child relationships, including using a `struct wait_status` concurrently with a list of `struct maps` to track parent-child relationships, but eventually settled on the current design. It is fairly easy to extend this design to accommodate more syscalls, as we simply have to add an `if` statement to `/userprog/syscall.c` and a `switch` statement to the `argument_check()` function to implement each new syscall.

3 Task 3

3.1 Data Structures and Functions

We create a new data structure in /threads/thread.h, struct fd_map, defined as follows:

```
struct fd_map {
    int fd;
    struct file *file;
    struct list_elem elem;
};
```

We also add 3 new fields to struct thread in /threads/thread.h: (1) struct list fd_map_list, which is a list of file descriptor tables, (2) struct file *executable, which stores the opened and running executable file for the current thread, and (3) int allo_fd = 3, which keeps track of the number of currently opened file descriptors. 0, 1, 2 default to stdin, stdout, stderr. New opened files will acquire fds starting from 3.

The majority of the work for this task is done in /userprog/syscall.c for error checking and dispatch jobs to functions in /filesys/file.c. We reference /lib/syscall-nr.h for syscall numbers. If all error checks are passed, we invoke the functions in /filesys/file.c.

We define and initialize a global struct lock filesys_lock in /userprog/filesys.c. Each time we enter a syscall handle case, we acquire the lock and release it at the end of the case. In syscall.c, in addition to all the file syscalls, the exec() syscall also acquires the global filesys_lock upon entry and releases the lock at the end, in order to protect file_deny_write() and multiple calls to exec().

Additionally, we create a new function bool argument_check(const uint32_t *args) in /userprog /syscall.c that checks if the correct number of arguments for a syscall have been pushed correctly onto the stack. This function also checks the validity of the pointers.

3.2 Algorithms

- Step 1: /lib/user/syscall.c is the user program entry point for syscalls. The signals are passed to syscall_handler(). syscall_handler() receives the interrupt frame. The arguments (syscall numbers) can be accessed from args[i], starting with i = 0, per the x86 calling convention. Pass these args into argument_check().
  - Perform unique checks for each syscall in its respective if block. If all the checks pass, dispatch syscalls to functions in /filesys/file.c and emit error messages per the results, putting the return values in f->eax. In this case, the calling processes are responsible for handling error cases.
- Step 2: Handling file descriptors:
  - open() adds a new node to fd_map_list with fd = allo_fd, and runs thread_current()-> allo_fd++, or returns an error if allo_fd < 3.
  - close() closes the file with the fd passed into it, removes the corresponding node from fd_map_list, and runs thread_current()->allo_fd--, or returns an error if (fd < 3 || fd >= allo_fd).

Failure checks and edge cases: argument_check() uses a switch to check each pointer in the arguments to the function against PHYS_BASE by invoking get_user(). get_user() sets eax to -EFAULT upon encountering a page fault, which provides a way to to inform the kernel that a page fault has occurred while at the same time preventing the kernel from crashing. If this check fails, syscall_handler() emits an error message and kills the calling process. We use the following approach for argument_check():

- Step 1: Validate args[0] and &args[0] first.

- Step 2: Parse `args[0]` and decide the expected number of arguments for the syscall using a `switch`.
- Step 3: Validate the remaining `&args[i]`.
- Step 4: Try to dereference and validate each `args[i]`, and return an error and kill the process if it fails. Return true if all the checks pass.
- Note: Each syscall has certain unique conditions that must be checked; these are checked in their respective `if` blocks. `argument_check()` only tests the validity of the pointers.

3.3 Synchronization

Syscall handling cases are shared processes. We put a lock around each syscall handle case in order to synchronize access to these processes. We deallocate `filesys` resources in `process_exit()` by traversing `fd_map_list` and closing each open file, and then deallocating `fd_map_list` itself. For `file_deny_write()` and `file_allow_write()`, we make the following changes in `load` in `/userprog/process.c`:

- Use the `struct file *executable` we defined in `/threads/thread.h`, set `struct file *file` to `thread_current()->executable`.
- Call `file_deny_write()` just after `file = filesys_open(file_name)`
- Instead of calling `file_close()` at the end of `load`, call `filesys_close(thread_current->executable)`, which implicitly calls `file_allow_write()`.

3.4 Rationale

We attempted to minimize the amount of code written for this task. We use a single function for pointer validation of all syscalls, using separate `if` statements to handle each syscall number. We make a small modification to the `pagefault` handler, and include two lines of file access protection code. This implementation simplifies adding support for additional syscalls, as we can simply add an `if` statement and a `switch` case to the error handler.

4 Additional Questions

4.1 Question 1

`/tests/userprog/child-bad.c` uses an invalid stack pointer and invokes a syscall. This test checks to see if the syscall handler is equipped to handle invalid pointers.

In line `12`, in the call to `asm volatile()`, the `movl $0x20101234` clause sets the stack pointer to an invalid address.

4.2 Question 2

`/tests/userprog/exec-bound-2.c` uses a valid stack pointer when making a syscall, but it is too close to a page boundary, which causes some syscall arguments to be located in invalid memory. This test checks to see if the syscall handler kills the calling process when invalid pointers are passed into a syscall.

In line `13`, the function sets a the address of `char *p` to just under a page boundary, such that only the first byte of `p`, which is set to `SYS_EXIT` in line `14`, is valid.

4.3 Question 3

The test suite as it is doesn't provide full coverage for testing file-related syscalls. The test suite can be extended by adding tests for file related syscalls, e.g.:

- Invoking `seek()` on an invalid file.
- Concurrently invoking two or more file-related syscalls to check if the syscall handler disallows such an attempt.

Project 1 Final Report

Group 4

July 14 2020

Contents

1 Changes

1.1 Task 1

We made no changes to the approach we detailed in our design doc for this task.

1.2 Task 2

We created a new data structure, `struct exec_info`, in `process.c` after consulting with Sam about some problems we were running into for this task:

```
struct exec_info {
    struct wait_status *wait_status
    struct semaphore load_sema;
    bool success_exec;
    char *fn_copy;
};
```

This struct contains a pointer to the `wait_status` struct between parent and child, a `load_sema` semaphore that we use to signal successful `load`s between processes, a `bool success_exec` that we use to determine if a `wait_status` struct is ready to be pushed into the list of `wait_status` structs after successful execution, and a `char *fn_copy` which we use to set the `file_name` in `start process()`.

We used this struct to link `process_execute()` and `start_process()`, as the current threads in each of these functions are distinct. We needed this struct in order to successfully push the child's `wait_status` struct to the parent's `waits` list. We modified `start_process` to accept an `exec_info` struct as its argument instead of a `file_name` string.

We also added a new field to `/threads/thread.c`, `struct lock *intr_lock_ref`, which we used variously in `/userprog/process.c` and `/userprog/syscall.c` to solve synchronization issues that were preventing the `wait-kill` test from passing.

Additionally, we added a few lines of code to `page_fault()` in `/userprog/exception.c` to handle test cases that were causing page faults.

1.3 Task 3

We altered the `argument_check()` function that we used to validate pointers. This function now returns an `int` instead of a `bool`. It is now an all-in-one checker for all syscalls. It returns `-1` if the process is to be terminated, `0` if the syscall fails (in this case, the process isn't terminated but the syscall jobs are not dispatched), and `1` if the tests pass and the syscalls jobs are eligible to be dispatched.

2 Reflection

We started working on this project early, which granted us ample time to robustly sort out the problems we ran into before the deadline. Post the design review with our TA, Kevin, our design needed only minor tweaking in order to get everything working. In particular, we found it necessary to make some changes to our design for task 2, and one change to our design for task 3, as detailed above. We divided up the work for this project as follows:

Jenna: Task 2, debugging
Jacob: Task 1, debugging
Taoxi: Task 3, debugging, test cases
Angad: Task 2, design doc, final report

We were able to work cohesively as a team and didn't run into any disagreements about the design and implementations for this project. Despite the vast time zone differences between us, we were able to convene frequently and work on the project together in real-time. Overall, while challenging, this project was engaging and fun to figure out.

3 Student Testing Report

We wrote 2 new tests for this project: `seek-bad.c` and `tell-simple.c`:

3.1 Test 1

3.1.1 Feature Tested and Test Overview

Our first test, `seek-bad.c`, tests various ways in which `SYS_SEEK` might fail. This syscall must either fail silently or terminate the process with exit code `-1`. This test attempts to (1) run `seek()` on a file descriptor which does not exist, which must cause it to fail silently, and (2) run `seek()` with a negative position passed into it, which must cause it to fail, kill the process, and return `-1`.

3.1.2 Output

Raw output:

```
Copying tests/userprog/seek-bad to scratch partition...
Copying ../../tests/userprog/sample.txt to scratch partition...
qemu-system-i386 -device isa-debug-exit -hda /tmp/dk0kFiEDLx.dsk -m 4 -net none -nographic
-monitor null
PiLo hda1
Loading...........
Kernel command line: -q -f extract run seek-bad
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer...  532,480,000 loops/s.
hda: 5,040 sectors (2 MB), model "QM00001", serial "QEMU HARDDISK"
hda1: 181 sectors (90 kB), Pintos OS kernel (20)
hda2: 4,096 sectors (2 MB), Pintos file system (21)
hda3: 106 sectors (53 kB), Pintos scratch (22)
filesys: using hda2
scratch: using hda3
Formatting file system...done.
Boot complete.
Extracting ustar archive from scratch device into file system...
Putting 'seek-bad' into the file system...
Putting 'sample.txt' into the file system...
Erasing ustar archive...
Executing 'seek-bad':
(seek-bad) begin
(seek-bad) create "test.txt"
(seek-bad) open "test.txt"
seek-bad: exit(-1)
Execution of 'seek-bad' complete.
Timer: 73 ticks
Thread: 6 idle ticks, 64 kernel ticks, 3 user ticks
hda2 (filesys): 135 reads, 224 writes
hda3 (scratch): 105 reads, 2 writes
Console: 966 characters output
Keyboard: 0 keys pressed
Exception: 0 page faults
Powering off...
```

Raw results:

```
PASS
```

3.1.3 Potential Kernel Bugs

Two non-trivial potential kernel bugs that would affect the output of this test case are:

- If the kernel passes an invalid file descriptor (negative or unopened), the test will fail.
- If the kernel contains an incorrect implementation of `write()` or `tell()`, it would cause `seek()` to fail.

3.2 Test 2

3.2.1 Feature Tested and Test Overview

Our second test, `tell-simple.c`, tests the functionality of the `SYS_TELL` syscall. It (1) writes a few bytes to an open test file, (2) checks the position by running `tell()`, (3) runs `seek()` to some point in the file, and (3) runs `tell()` again to check if we are at the expected position. If any of these steps fail, the test outputs an error.

3.2.2 Output

Raw output:

```
Copying tests/userprog/tell-simple to scratch partition...
Copying ../../tests/userprog/sample.txt to scratch partition...
qemu-system-i386 -device isa-debug-exit -hda /tmp/s1rX1pJMge.dsk -m 4 -net none -nographic
-monitor null
PiLo hda1
Loading...........
Kernel command line: -q -f extract run tell-simple
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer...  543,129,600 loops/s.
hda: 5,040 sectors (2 MB), model "QM00001", serial "QEMU HARDDISK"
hda1: 181 sectors (90 kB), Pintos OS kernel (20)
hda2: 4,096 sectors (2 MB), Pintos file system (21)
hda3: 106 sectors (53 kB), Pintos scratch (22)
filesys: using hda2
scratch: using hda3
Formatting file system...done.
Boot complete.
Extracting ustar archive from scratch device into file system...
Putting 'tell-simple' into the file system...
Putting 'sample.txt' into the file system...
Erasing ustar archive...
Executing 'tell-simple':
(tell-simple) begin
(tell-simple) create "test.txt"
(tell-simple) open "test.txt"
(tell-simple) file position at 10
(tell-simple) file position at 100
(tell-simple) end
tell-simple: exit(0)
Execution of 'tell-simple' complete.
Timer: 70 ticks
Thread: 9 idle ticks, 59 kernel ticks, 2 user ticks
hda2 (filesys): 135 reads, 224 writes
hda3 (scratch): 105 reads, 2 writes
Console: 1076 characters output
Keyboard: 0 keys pressed
Exception: 0 page faults
Powering off...
```

Raw results:

```
PASS
```

3.2.3 Potential Kernel Bugs

Two non-trivial potential kernel bugs that would affect the output of this test case are:

- If the kernel passes an invalid file descriptor (negative or unopened), the test will fail.
- If the kernel contains an incorrect implementation of `write()` or `tell()` it would cause the test to fail (e.g.: if we move the current position twice and pass `EOF`).

3.3 Experience Writing Tests

There is plenty of room for improvement in the current testing suite for Pintos. We found that there are many file system related syscalls that aren't tested robustly, which is something we pointed out in our design doc. We built on that observation for the test cases that we provided for this project. In writing our test cases, we learned (besides the trivial case of learning how to write test cases), that it is essential to be paranoid and consider as many edge cases as possible when trying to come up with test cases for various aspects of an OS. One of the essential features of an OS is reliability, which can only be maintained when the system programmer covers all bases.

Project 2 Design Doc

Group 4

July 21 2020

Contents

1 Task 1: Efficient Alarm Clock

1.1 Data Structures and Functions

We create a new data structure, `struct sleep_elem`, in `timer.h`:

```
struct sleep_elem {
    struct semaphore sleep_sema;
    int64_t raw_wake_time;
    struct list_elem elem;
};
```

We create a linked list of this `struct` in order to keep track of the sleep times for threads, sorting the list by wake time, and define a global `struct lock list_lock` which we use to make list operations thread-safe, in `timer.c`.

Additionally, we create a comparator function in `timer.c` which we use to sort the `list` of `sleep_elems` and to insert nodes into it:

```
bool sleep_elem_comp(const struct list_elem *e1, const struct list_elem *e2, void *aux);
bool wake_time_comp(struct sleep_elem *s1, struct sleep_elem *s2);
```

We pass `wake_time_comp()` as the `aux` argument in `list_insert_ordered()`.

1.2 Algorithms

In `timer_sleep(int64_t ticks)`:

- Step 1: If `ticks <= 0`, return immediately.
- Step 2: Instantiate a `sleep_elem`, initializing `sleep_sema` to 0 and `raw_wake_time` to `start + ticks`.
- Step 3: Acquire the global list lock.
- Step 4: Add this `sleep_elem` to the `list` of `sleep_elem` structs using `list_insert_ordered()` and our custom comparator function to insert it in the appropriate place in the `list` according to its `raw_wake_time`.
- Step 5: Release the global list lock.
- Step 6: Call `sema_down()` on `sleep_sema`.
- Step 7: Free the previously instantiated `sleep_elem`.

In `timer_interrupt(struct intr_frame *args UNUSED)`:

- Step 1: Disable interrupts.
- Step 2: While the `sleep_elem` list is not empty:
  - If the `raw_wake_time` of the `sleep_elem` is less than or equal to `ticks`, pop the front of the list and call `sema_up()` on `sleep_sema`.
  - Else, break.
- Step 3: Enable interrupts by resetting to the old interrupt level.

1.3 Synchronization

We use a global lock in `timer.c` in order to make the list operations in `timer_sleep()` thread-safe. Additionally, we use a semaphore `sleep_sema` to signal between `timer_sleep()` and `timer_interrupt()`. Since this task is relatively straightforward, we did not need to use too many synchronization primitives.

1.4 Rationale

Since `timer_interrupt()` is called at every tick, we are able to use it to check the `sleep_elem`s and insert nodes into the `list` of `sleep_elem`s. We attempted to make minimal modifications to the kernel for this task, and our design limits significant modifications to `timer.c`. We considered using a `list` of `struct thread_wake_up_times` which would contain the TCB and wake up time for threads, but settled on using a `list` of `sleep_elem`s and maintaining a `list` of waiting threads in `timer.c` instead. We also initially considered adding a semaphore to `thread.h` but decided to move it to `struct sleep_elem` instead.

2 Task 2: Priority Scheduler

2.1 Data Structures and Functions

We add two new fields, `int prev_priority` and `priority_sema`, to the TCB in `thread.h` and rename the existing `priority` field to `cur_priority` as it now holds the current priority of the thread. We also modify `thread_get_priority()` to return the effective priority and `thread_set_priority()` to swap the values of `prev_priority` and `cur_priority`.

We also use a slightly different version of our custom comparator function, which now compares thread priority values:

```
bool thread_comp(const struct list_elem *e1, const struct list_elem *e2, void *aux);
bool priority_comp(struct thread *t1, struct thread *t2);
```

We pass `priority_comp()` as the `aux` argument in `list_insert_ordered()`.

2.2 Algorithms

- Initialize `priority_sema` to 1.
- Use `list_insert_ordered()`, using our priority comparator function as the `aux` argument, in place of `list_push_back()` in `thread_unblock()`, `thread_yield()`, and `init_thread()`.
- Sort `ready_list` by priority after every call to `thread_set_priority()`, and then call `thread_yield()`.
  - Call `sema_down()` on `priority_sema`.
  - Change the thread priority values.
  - Call `sema_up()` on `priority_sema`.
- In `lock_acquire()`:
  - Step 1: Find `lock->holder`'s priority value.
  - Step 2: Call `sema_down()` on `lock->holder->priority_sema`.
  - Step 3: Save `lock->holder->prev_priority` in a `temp` variable.
  - Step 4: Set `lock->holder->prev_priority` to `lock->holder->cur_priority`.
  - Step 5: Set `lock->holder->cur_priority` to `thread_current()->cur_priority`.
  - Step 6: Call `sema_up()` on `lock->holder->priority_sema`.
  - Step 7: Call `sema_down()` on `list_lock`.
  - Step 8: Reset all the priority values.
  - Step 9: Call `sema_down()` on the previous lock holder's `priority_sema`.
  - Step 10: Set the previous lock holder's `cur_priority` to its `prev_priority`.
  - Step 11: Set the previous lock holder's `prev_priority` to the `temp` variable we saved in Step 3.
  - Step 12: Call `sema_up()` on the previous lock holder's `priority_sema`.

2.3 Synchronization

We use a global `struct list_lock` in `timer.c` in order to synchronize access to `ready_list` for each call to `list_insert_ordered()`. This lock also allows us to prevent race conditions like different threads competing to modify the same priority value. Additionally, we use a semaphore `priority_sema` that allows us to make any operations that change priorities atomic.

2.4 Rationale

We considered creating a `struct donation` to keep track of the threads and priority values involved in a donation, but settled on the current design as we believe it is not as memory-intensive. Sorting `ready_list` by priority simplifies the rest of the task, and we are able to limit significant modifications to `lock_acquire()` in `synch.h`.

3 Additional Questions

3.1 Question 1

The program counter remains the same between two threads when one is switched to the other as both threads are running the same method, `thread_switch()`, at the time of switching. The stack pointer of the current thread is saved in `%eax`, and is later switched into the new thread's stack pointer. The registers are stored in `%ebx`, `%ebp`, `%esi`, and `%edi` by pushing them onto the stack.

3.2 Question 2

The page containing the kernel thread's stack is freed when `thread_schedule_tail()` is called in the `schedule()` method if the status of that thread is set to `THREAD_DYING` (i.e., if that thread made a call to `thread_exit()`). We cannot just free that page of memory within `thread_exit()` because `thread_exit()` is being run in that kernel thread's stack, so calling `palloc_free_page()` would deallocate the kernel thread's page before it has finished running. We can instead free that page in the scheduler because the kernel thread within that page is not running.

3.3 Question 3

The `thread_tick()` function executes in the main stack frame. It is called by `timer_interrupt()`, which is called by the interrupt handler for the main thread.

3.4 Question 4

Assuming the process with the smallest priority is most urgent, we create 3 processes in our test case:

- Process `A` with priority `60` which is very quick, but needs resources from process `C`.
- Process `B` with priority `10`, which does a short CPU burst.
- Process `C` with priority `1`, which does a very long CPU burst.

With priority inversion, the processes should execute in the order `C->A->B`. Without priority inversion, the processes should execute in the order `B->C->A`, where `A` gets starved by `B`.

Note: This test does not provide any information if the scheduler falls back to round-robin scheduling. Basic tests such as `priority-donate-one.ck` and `priority-donate-lower.ck` must pass to show that priority scheduling is in effect.

Test code:

```c
struct lock test_lock;

// long CPU burst that lasts for a few seconds
void
consume_cpu ()  // takes a few seconds
{
  volatile unsigned long long i;  // volatile disables gcc optimizations
  for (i = 0; i < 1000000000ULL; ++i);
}

// call consume_cpu() 10 times, very long CPU burst
void
consume_cpu_long ()
{
  for (int i = 0; i < 10; i++)
```

```
    consume_cpu ();
}

void
priority_1_start ()
{
  test_lock.acquire ();
  consume_cpu_long ();  // very long CPU burst, but should finish before priority_10
  test_lock.release ();
  msg ("priority_1 finished\n");
}

void
priority_60_start ()
{
  test_lock.acquire ();
  msg ("priority_60 ran\n");  // should be printed before priority_10 ran."
  test_lock.release ();
}

void
priority_10_start ()
{
  // require no resource, just run if priority donation fails.
  consume_cpu ();  // much shorter than priority_1
  msg ("priority_10 finished\n");
}

void
test_priority_donate_simple ()
{
  lock_init (&test_lock);
  int pid_priority_1 = thread_create ("priority_1", 1, priority_1_start);
  int pid_priority_60 = thread_create ("priority_60", 60, priority_1_start);
  int pid_priority_10 = thread_create ("priority_10", 10, priority_10_start);
}
```

Expected output:

```
priority_1 finished
priority_60 ran
priority_10 finished
```

Actual output:

```
priority_10 finished
priority_1 finished
priority_60 ran
```

Any other output patterns would indicate that something is seriously wrong.

Project 2 Final Report

Group 4

July 29 2020

Contents

1 Changes

1.1 Task 1: Efficient Alarm Clock

We made the following changes to our initial design for this task based on the advice our TA, Kevin, gave us during the design review:

- We removed the new data structure we introduced in the design doc, `struct sleep_elem` in `timer.h`, and moved its 3 fields into the TCB in `thread.h`. We thus used `struct thread`s themselves instead of frivolously creating new data structures at the cost of memory.
- We replaced the two calls to `malloc()` that we had in our code with stack allocations, as Kevin informed us that allocating memory on the heap can cause unintended behaviour in this context.
- Instead of acquiring and releasing locks in our implementation, we instead enabled/disabled interrupts for critical sections.

1.2 Task 2: Priority Scheduler

We made significant changes to our initial design for this task based on the advice that Kevin gave us during the design review:

- Instead of sorting the ready list by priority after each call to `thread_set_priority()`, which would have caused significant slowdown, we took the max of `thread_schedule()` to find the next highest priority ready thread.
- We issued calls to `thread_yield()` within `thread_create()`, `sema_up()`, and `thread_set_priority()`.
- We did not use any synchronization primitives besides disabling/enabling interrupts for this task.
- We removed any synchronization we added on the ready list since it is always interacted with in a disabled interrupt context.
- We added two new fields to the TCB in `thread.h`: `struct list locks`, `struct lock waiting_for`. The list of locks was used to keep track of all the locks that the currently executing thread had in its possession, and `waiting_for` was used if a thread was trying to acquire a lock that was already held.
- We created a new function, `void priority_update(struct thread *t)`, which we used to recursively handle priority donation/updation, which worked as follows:
  - Iterate through all the locks that the current thread is holding and iterate through the waiters on each lock to find the maximum effective priority (if the maximum effective priority in the list was greater than the current thread's base priority, then we set the effective priority of the current thread to that value).
  - Recursively call `priority_update()` on the function that owns the lock that the current thread is waiting for by passing `t->waiting_for->holder` as its argument.
- On calls to `sema_up()`, we popped the max of the priorities of the `waiters` list.
- We created a new function, `struct semaphore_elem *max_priority_sema(struct list *list)`, which we used to determine the `semaphore elem` with the highest priority from the given list.
- We added a new field, `int *priority`, to `struct semaphore_elem` in `synch.c`, which points to the effective priority of the thread that called `cond_wait()`.
- In `cond_signal()`, we popped the `semaphore_elem` with the highest priority from the ready list using `max_priority_sema()`.
- We called `priority_update()` on the current thread in `thread_set_priority()`, `sema_down()`, `lock_acquire()`, and `lock_release()`.

- To support priority donation in `lock_acquire()`, if the lock is already held, we set the current thread's `waiting_for` field to that lock. Once the lock is acquired, we add it to the current thread's `locks` list, set `waiting_for` to `NULL`, and call `priority_update()`.
- In `lock_release()`, we removed the lock that is to be released from the `locks` list and called `priority_update()`.

2 Reflection

As with project 1, we started early on this project which gave us time to sort out our design flaws and come up with a working implementation. We found that the workload for this project was not as intense as for project 1. There were some subtleties about scheduling and synchronization that escaped our notice initially, but with the help of our TA we were able to get a better understanding of these subtleties. We divided up the work for this project as follows:

Jenna: Task 2, debugging, scheduling lab
Jacob: Task 1, task 2, debugging, scheduling lab
Taoxi: Task 2, debugging, scheduling lab
Angad: Task 1, debugging, scheduling lab, design doc, final report

As before, we were able to work cohesively as a group and build off each other's ideas. This project went more smoothly with much less of a time crunch than project 1.

Project 2 Scheduling Lab

Group 4

July 29 2020

Contents

1 Scheduling Simulator Implementation

a.
```
0: Arrival of Task 12 (ready queue length = 1)
0: Run Task 12 for duration 2 (ready queue length = 0)
1: Arrival of Task 13 (ready queue length = 1)
2: Arrival of Task 14 (ready queue length = 2)
2: IO wait for Task 12 for duration 1
2: Run Task 14 for duration 1 (ready queue length = 1)
3: Arrival of Task 15 (ready queue length = 2)
3: Wakeup of Task 12 (ready queue length = 3)
3: IO wait for Task 14 for duration 2
3: Run Task 12 for duration 2 (ready queue length = 2)
5: Wakeup of Task 14 (ready queue length = 3)
5: Run Task 14 for duration 1 (ready queue length = 2)
6: Run Task 15 for duration 2 (ready queue length = 1)
8: Run Task 15 for duration 1 (ready queue length = 1)
9: Run Task 13 for duration 2 (ready queue length = 0)
11: Run Task 13 for duration 2 (ready queue length = 0)
13: Run Task 13 for duration 2 (ready queue length = 0)
15: Run Task 13 for duration 1 (ready queue length = 0)
16: Stop
```

b.
```
0: Arrival of Task 12 (ready queue length = 1)
0: Run Task 12 for duration 2 (ready queue length = 0)
1: Arrival of Task 13 (ready queue length = 1)
2: Arrival of Task 14 (ready queue length = 2)
2: IO wait for Task 12 for duration 1
2: Run Task 13 for duration 2 (ready queue length = 1)
3: Arrival of Task 15 (ready queue length = 2)
3: Wakeup of Task 12 (ready queue length = 3)
4: Run Task 14 for duration 1 (ready queue length = 3)
5: IO wait for Task 14 for duration 2
5: Run Task 15 for duration 2 (ready queue length = 2)
7: Wakeup of Task 14 (ready queue length = 3)
7: Run Task 12 for duration 2 (ready queue length = 3)
9: Run Task 14 for duration 1 (ready queue length = 2)
10: Run Task 13 for duration 4 (ready queue length = 1)
14: Run Task 15 for duration 1 (ready queue length = 1)
15: Run Task 13 for duration 1 (ready queue length = 0)
16: Stop
```

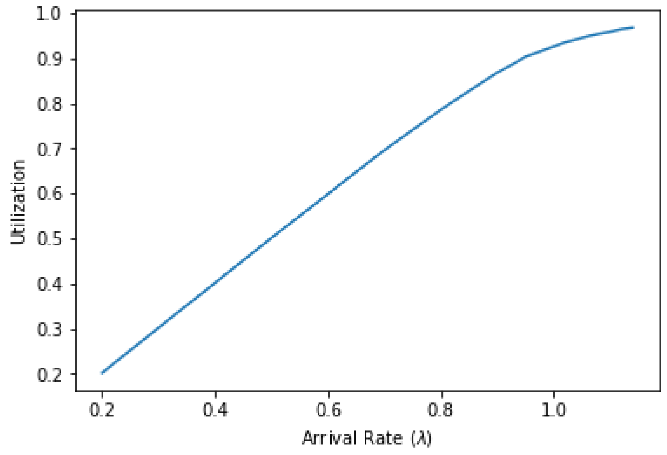
2 Approaching 100% Utilization

a. The system is open. The tasks we generate do not depend on whether the previous tasks have finished, and there is no feedback loop of any kind. Note that this is true only because X_i is exponential (and thus memoryless) so later arrivals do not depend on previous ones.

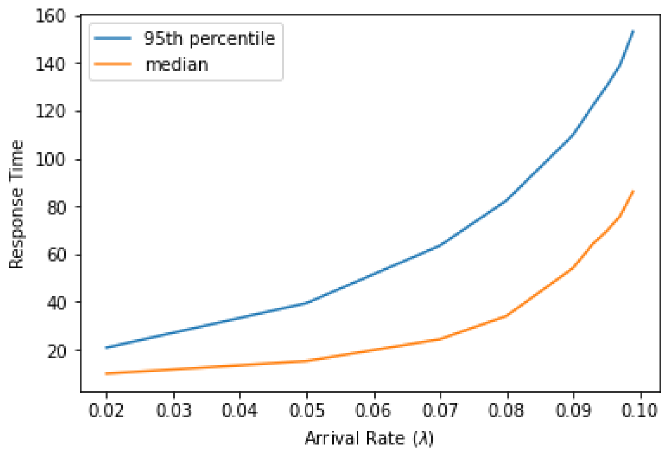
b. The mean arrival time is given by $\frac{1}{\lambda}$. We want a λ such that T_s = mean arrival time, i.e., such that $T_s = \frac{1}{\lambda}$. Thus, the required value of λ is $\frac{1}{T_s}$

c. On average, we let the CPU run for time T_s and idle for time T_s. If we want the system to run at 50% utilization on average, the mean arrival time must be increased to $2T_s$. Thus, the required value of λ is $\frac{1}{2T_s}$

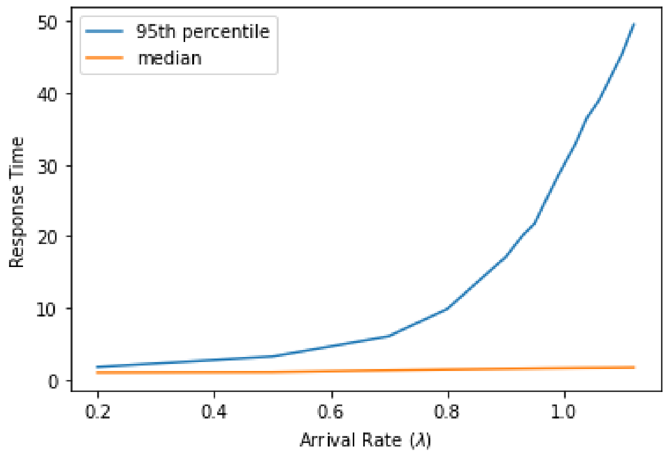
d. The CPU utilization increases linearly with λ and levels out past $\lambda = 1$:



e. The response time for each CPU burst increases exponentially with λ:



f. Using SRTF causes the median response time to become constant. However, the 95th percentile response time grows exponentially with λ as in FIFO:



g. When the arrival time λ approaches the service rate μ, the ready queue grows without bound which causes the latency to increase.

3 Fairness for CPU Bursts

a. The length of the FCFS queue never exceeds 2 because at any given time it is either the case that one task is running, or that both tasks are waiting.

b. For two independent exponential random variables S_1 and T_1 of parameter λ_S and λ_T,

$$\mathbb{P}[S_1 < T_1] = \frac{\lambda_S}{\lambda_S + \lambda_T} = \frac{1}{2} \quad (for \quad \lambda_S = \lambda_T)$$

c. By the Central Limit Theorem, we have that:

$$\lim_{m\to\infty} \frac{CPUTime(S) - m\mathbb{E}(S_i)}{\sqrt{mVar(S_i)}} \to \mathcal{N}(0,1)$$

$$\therefore \quad CPUTime(S) \sim \sqrt{mVar(S_i)}\mathcal{N}(0,1) + m\mathbb{E}(S_i)$$
$$\implies \quad CPUTime(S) \sim \mathcal{N}(0, mVar(S_i)) + m\mathbb{E}(S_i)$$
$$\implies \quad CPUTime(S) \sim \mathcal{N}(m\mathbb{E}(S_i), mVar(S_i))$$

d.

$$Let \quad x \sim \mathcal{N}(m\mathbb{E}(S_i), mVar(S_i))$$
$$Note \quad \mathbb{E}(S_i) = \mathbb{E}(T_i), \quad and \quad Var(S_i) = Var(T_i)$$

$$\implies \mathbb{P}[n \cdot CPUTime(S) < CPUTime(T)] = \mathbb{P}[nx < x] = \mathbb{P}[nx - x < 0]$$

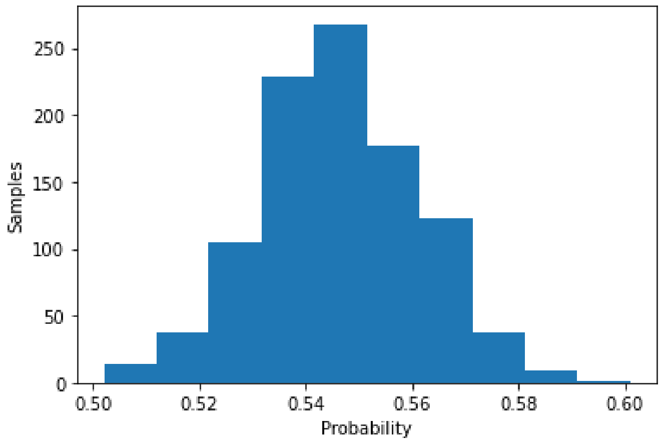
$$n \cdot \mathcal{N}(m\mathbb{E}(S_i), mVar(S_i)) - \mathcal{N}(m\mathbb{E}(S_i), mVar(S_i)) =$$
$$\mathcal{N}(nm\mathbb{E}(S_i), n^2 mVar(S_i)) + \mathcal{N}(-m\mathbb{E}(S_i), mVar(S_i)) =$$
$$\mathcal{N}((n-1)m\mathbb{E}(S_i), (n^2+1)mVar(S_i))$$

$$\implies Zscore = \frac{0 - (n-1)m\mathbb{E}(S_i)}{\sqrt{(n^2+1)mVar(S_i)}}$$

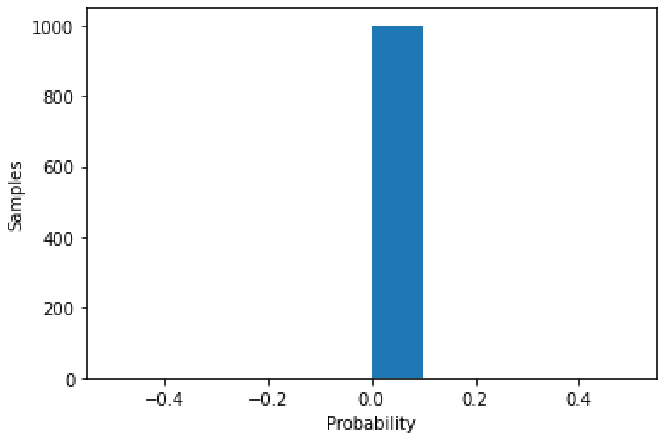
$$\therefore \quad \mathbb{P}[n \cdot CPUTime(S) < CPUTime(T)] = \Phi\left(\sqrt{m}\frac{(1-n)\mathbb{E}(S_i)}{\sqrt{(n^2+1)Var(S_i)}}\right)$$

e. For $m = 100$, the probability that one task receives at least 10% more CPU time than the other is 0.55. For $m = 10000$, this probability is 0. Sam's reasoning that the CPU allocation will be fair with FCFS is correct.

f. With a confidence of 95%, we can say that the average probability that one task gets at least 10% more CPU time than the other for $m = 100$ lies in the interval $[0.52, 0.58]$:



With a confidence of 95%, we can say that the average probability that one task gets at least 10% more CPU time than the other for $m = 10000$ is exactly 0:



We set up the simulation by creating a function that generated a task based on the number of CPU bursts for that task (m) and the average burst length (λ).

The tasks were created by sampling an exponential distribution of parameter λ m times and setting the result to be the length of each burst. These tasks were then allowed to be simulated on an FCFS scheduler and the total time that one task was being serviced was divided by the total time of the other.

If the calculated ratio was greater than our simulation parameter n (= 1.1 in our simulation to test if one task receives 10% more time), the number of unfair CPU instances was incremented. After repeating this process 1000 times, we divided the number of unfair instances by 1000 to get the experimental probability that either task would run for n times longer than the other.

We repeated this process 1000 times each for $m = 100$ and $m = 10000$ to obtain 1,000 sample probabilities for each, allowing us to construct a 95% confidence interval based on the distribution of the samples.

Project 3 Design Doc

Group 4

August 5 2020

Contents

1 Task 1: Buffer Cache

1.1 Data Structures and Functions

We create two new files in `filesys/`, `buffer_cache.c` and `buffer_cache.h`, which we use to implement the buffer cache.

In `buffer_cache.h`, we create a new `struct cache_block`:

```
struct cache_block {
    block_sector_t sector;
    bool is_dirty;
    char buffer[BLOCK_SECTOR_SIZE];
    int active_readers;
    int active_writers;
    struct list_elem elem;
};
```

In `buffer_cache.c`, we initialize a linked list of `cache_blocks` of size 64 each and define the following:

```
static struct lock buffer_cache_lock;
struct condition ok_to_read, ok_to_write;
void cache_read(struct block *block, block_sector_t sector, void *buffer);
void cache_write(struct block *block, block_sector_t sector, void *buffer);
void evict_block(void);
```

1.2 Algorithms

We modify `filesys_done()`, setting `is_dirty` in each cache block that needs to be written on eviction or shutdown.

`void cache_read(struct block *block, block_sector_t sector, void *buffer):`

1. Check if `block` is already in the buffer cache:
   - If it is already in the buffer cache:
     i. Remove `block` from the `cache_blocks` list.
     ii. Push `block` to the front of the `cache_block` list.
   - If it is not in the buffer cache:
     i. Read `block` in from disk.
     ii. Push `block` to the front of the `cache_block` list.
2. Copy the data into `buffer`.

`void cache_write(struct block *block, block_sector_t sector, void *buffer):`

1. Check if `block` is already in the buffer cache:
   - If it is already in the buffer cache:
     i. Set `is_dirty = true`.
     ii. Remove `block` from the `cache_block` list.
     iii. Push `block` to the front of the `cache_block` list.
   - If it is not in the buffer cache:
     i. Read `block` in from disk.
     ii. Write `block` to `buffer`.

   iii. Set `is_dirty = true`.
   iv. Remove `block` from the `cache_block` list.
   v. Push `block` to the front of the `cache_block` list.

`void evict_block(void):`

1. Call `list_pop_front()` on the `cache_block` list.
2. Deallocate the `block` that was popped.

1.3 Synchronization

We use a global lock and condition variables in order to synchronize access to shared structures. We acquire the global lock before performing any operations on the buffer cache and release it before performing any blocking disk I/O operations.

We synchronize reads and writes using condition variables. A reader will wait until there are no active or waiting writers before accessing a cache block, and a writer will wait until there are no active readers or writers before accessing a cache block.

1.4 Rationale

This design is the simplest and most straightforward way to implement a buffer cache for this task. Using LRU as the eviction policy allows us to update the list in simple $O(1)$ operations whenever we access a cache block element.

2 Task 2: Extensible Files

2.1 Data Structures and Functions

We make the following changes to data structures and functions in `inode.c`:

We add the following fields to `struct inode_disk`:

```
{
    bool is_dir;
    block_sector_t parent;
    block_sector_t direct[3];
    block_sector_t indirect[2];
    block_sector_t double_indirect;
    char unused[475];
}
```

Additionally, we add the following fields to `struct inode`:

```
{
    bool is_dir;
    struct lock ref_lock;
    struct inode_disk data;
}
```

Lastly, we add an argument, `bool dir`, to `inode_create()`.

Note: We use chevrons ("<" and ">") to separate directory entries.

2.2 Algorithms

`static block_sector_t byte_to_sector(const struct inode *inode, off_t pos):`

1. Call `cache_read(fs_device, inode->sector, &inode->data)`.
2. If the `length` of the on-disk inode is greater than `pos`, return `-1`.
3. If `pos < 512*3` return `inode->data.direct[pos/512]`.
4. If `pos < 512*128*2`, return `inode->data.indirect[(pos - 512*3)/(512*128)]`.
5. Otherwise, return `inode->data.double_indirect`.

`bool inode_create (block_sector_t sector, off_t length, bool dir):`

1. Replace calls to `block_write()` with calls to `cache_write()`.
2. When setting the disk inode metadata, set `disk_inode->parent = thread_current()->cwd`, where `cwd` is the current thread's current working directory.
3. Set `disk_inode->is_dir = dir`.

`off_t inode_write_at(struct inode *inode, const void *buffer_, off_t size, off_t offset):`

1. In order to implement file growth, we update the length of the disk inode containing the address to be written to.
2. We do not add support for "sparse files", and instead write data blocks for implicitly zeroed blocks.

```
void block_write(struct block *block, block_sector_t sector, void *buffer):
```

1. Check if a sector is a data sector by checking the metadata in the buffer.
2. If it is a data sector:
   i. Find a free sector in `free_map`.
   ii. Write `buffer` to the free sector.
   iii. Find the file inode that references this data sector.
   iv. Rewrite the file inode to a new data sector, copying over all the fields and changing only the pointer to this data sector (this may require multiple disk/cache reads if `block` is referenced with indirect pointers).
   v. Go to the `parent` of the file inode (directory), and write the duplicate file entry into directory data with the new sector to disk. (On the next write to this sector, we delete the prior entry). If the duplicate entry is missing a closing chevron on the next load, delete the later entry. If no more sectors are available, load the old `free_map` back out of disk and make no changes to the parent directory.
3. If it is a file inode, execute steps (iii) to (v).
4. If it is a directory data structure, execute steps (i), (ii), and (v).
5. If it is a directory inode, do nothing apart from the usual procedure for `block_write()`.

Note:

- Within `filesys_close()`, we acquire `ref_lock`, update `open_cnt`, and release `ref_lock`. If `open_cnt == 0`, we deallocate the memory block.
- Within `filesys_open()`, we acquire `ref_lock`, update `open_cnt`, and release `ref_lock`.

2.3 Synchronization

We use a `struct lock ref_lock` to prevent concurrent accesses to the `open_cnt` field of `struct inode`.

2.4 Rationale

We considered various options for the number of direct, indirect, and doubly-indirect pointers for the inode structure. We initially elected to use a single doubly-indirect pointer and no direct or indirect pointers but later decided to use 3 direct, 2 indirect, and one doubly-indirect pointers in order to accommodate small, medium, and large files and maximize seek time for each.

We also initially considered using a log-structured file system, which involved using a journal to keep track of sequences of operations and apply a sequence of operations atomically. We later decided to use copy-on-write instead, which we believe will be easier to implement than a log-structured file system.

3 Task 3: Subdirectories

3.1 Data Structures and Functions

In `filesys.c`, we create a new `struct busy_disk_sector`:

```
struct busy_disk_sector {
    block_sector_t sector;
    struct block *block;
    struct lock sector_lock;
    struct list_elem elem;
};
```

We create a linked list of this `struct` in `filesys.c` and initialize it on startup.

Additionally, we add two new fields to the TCB in `thread.h`:

```
{
    char[MAX_PATH_LEN] cwd;
    struct lock cwd_lock;
}
```

Where `MAX_PATH_LEN = 1024`.

Lastly, we create a new function `bool resolver(char *path, char *abspath)`, which resolves a path into the absolute path of that file/directory and returns `true` on success.

3.2 Algorithms

`bool resolver(char *path, char *abspath):`

1. Check if `path` contains valid characters (only letters of the alphabet, /, _, and whitespace are considered valid).
2. Check the first character:
   - If the first character is /, we do absolute path resolution:
     i. Break the absolute path into chunks using the provided `get_next_part()` function.
     ii. While chunks remain to be processed, access the inode of root and go through each direct, indirect, and doubly-indirect pointer in order to check the directory name of the subsequent level in the directory tree (we go through every block, including regular file blocks, but only check blocks with `is_dir` set to true, until we hit what we want).
     iii. If we can't find the next chunk, raise a `"directory not found"` error.
     iv. If we find a chunk, continue until the `while` loop exits. We now have the required directory data block along with its absolute path.
   - If the first character is not /, we do relative path resolution:
     i. Get the current working directory from the running thread and copy it into a new string, `char *cwd_cp`
     ii. Parse `".."` to remove chunks backwards from `cwd_cp`.
     iii. Concatenate `cwd_cp` with the relative path, omitting `"."` and `".."`
     iv. Do absolute path resolution on `cwd_cp`.

To obtain files and directories from file descriptors:

In our implementation for Project 1, we could obtain the absolute path of a file from its file descriptor (since we had a linked list for each process to store the `fd` table). We can simply call `readdir()` and `open()` on this absolute path to get the corresponding file or directory. Since we don't know whether the path leads to a file or directory before we obtain the inode, we call both `readdir()` and `open()` on the path. If the path is valid, one of these syscalls will return an error, which we can silently handle, and we will then know whether the path leads to a directory or a file.

System calls:

- `chdir`:
  1. First, do a `readdir()`-esque logic.
  2. Do a path resolution on the directory we want to change to.
  3. If the directory we want to change to exists, acquire `thread_current()->cwd_lock`, set `thread_current()->cwd` to that directory, and release `thread_current()->cwd_lock`.
- `mkdir`:
  1. First, do a `readdir()`-esque logic.
     - If the directory at the specified path does not exist, we determine the level of failure within the `readdir()` logic (i.e. which level of the directory is missing).
  2. Add an inode to the inode structure and allocate information into the corresponding data block.
- `readdir`:
  - We described the design for `readdir` in the algorithms section. In our implementation, we will break `readdir` up into separate functions, which will perform path resolution and obtain the data block of the directory.
- `isdir`:
  - We implement `isdir` using logic abstracted away from the design we described for `readdir` in section 3.2.
- `open`:
  1. First, resolve the specified path.
  2. Obtain the data block of the file/directory at the specified path.
  3. If `is_dir == false`, move it to the buffer cache if possible. Else, raise an error.
- `close`:
  1. Obtain the absolute path from the `fd`.
  2. Obtain the inode at the file path.
  3. Traverse the buffer cache and remove the corresponding inode.
  4. Return `true` if successful.
- `exec`:
  - Resolve the specified path to ensure that the arguments passed into `exec` are absolute paths.
- `remove`:
  1. Use the same logic as `open`, without loading blocks into the buffer cache.
  2. Obtain the data block and check if it is a directory.
  3. If it is a directory:
     i. Check inode pointers to ensure that all the inodes to be removed are marked for deletion.

        ii. If so, mark itself for deletion, call `free_map_release()` on all the inodes it points to and on the directory data block.

    4. If it is a file:
        – Mark itself for deletion and call `free_map_release()` on the sector.

- `inumber`:
    1. Resolve the specified path.
    2. Obtain the inode from the file path.
    3. Obtain the inumber from the inode, where the inumber is the sector number for the corresponding inode.

Note:

- We mention "`readdir()`-esque logic" in several places because we don't want to make a syscall inside another syscall, especially when the callee is heavy-duty. That is, the logic in these parts of the code will be very similar to `readdir` with unnecessary parts removed. It is very possible we abstract away some of the logic therein and put it into new functions, but the only way to figure out how we break up these functions is by actually implementing them, which we cannot state with certainty at the design stage. Nevertheless, the overall logic is fixed: the syscalls are going to behave exactly as specified. We're only missing volatile implementation details that cannot be stated with certainty at this point.
- When initializing the first thread in `main()`, we set `cwd` to the root directory.
- "Will a user process be allowed to delete a directory if it is the `cwd` of a running process?" No. We prevent this by traversing the global thread list once within the `remove` syscall.

3.3 Synchronization

We remove the global filesys lock that we used in Project 1, but use a global lock on other, non-filesys-related syscalls. We also use a `sector_lock`, which we use to synchronize access to the `busy_disk_sector` list.

3.4 Rationale

The `resolver()` function allows us to easily obtain absolute paths from relative paths without repetition.

4 Additional Questions

4.1 Question 1

- Write-behind: In `timer_interrupt()`, if `ticks % 20 == 0`, write the data in the buffer cache to disk.
- Read-ahead: For sufficiently large files, for each megabyte of filesize, load the following 8 sectors in addition to the sector requested by the user.

Project 3 Final Report

Group 4

August 16 2020

Contents

1 Changes

1.1 Task 1

We closely followed Sam's design for this task. Our initial design coincided fairly closely with Sam's design, so we didn't have to change much in the way of data structures and functions.

1.2 Task 2

We modified our initial design slightly for this task. In particular, we created two new functions in `inode.c`:

```
void num_blocks(off_t length, off_t offset, size_t *retlist):
```

We used this function to calculate the number of direct, indirect, and doubly-indirect blocks to be allocated upon file extension, and populate `retlist`, a 7-tuple, with these values.

```
bool file_extend(struct inode *inode, off_t offset):
```

We used this function to extend the file pointed to by `inode`.

Besides these functions, we incorporated a few design ideas from Sam's example design doc and followed the relaxed synchronization requirement for this task. We abandoned our copy-on-write idea as we realized that we were not required to handle unexpected shutdowns.

1.3 Task 3

Apart from incorporating a few design tips from Sam's example design doc, we stuck to our initial design for this task. We abandoned the idea of a magic byte for our directories as we realized that we were not required to handle the case of unexpected shutdowns.

2 Reflection

This project was definitely the most time-consuming out of the three we've had to do for this class. The example design doc that Sam released, together with the relaxed synchronization requirement for task 2, was very helpful in completing the project on time. We divided up the work for this project as follows:

Jenna: Task 1, debugging
Jacob: Task 2, task 3, tests, debugging
Taoxi: Task 3, tests, debugging
Angad: Task 1, task 2, tests, debugging, design doc, final report

3 Student Testing Report

We wrote 2 new tests for this project: `cache-hitrate.c` and `cache-coalesce.c`:

3.1 Test 1

3.1.1 Feature Tested and Test Overview

Our first test, `cache-hitrate.c`, tests the hit rate of the buffer cache. We accomplish this by following the scheme mentioned in the project spec. We create syscalls which allow us to access the buffer cache functions in userspace, and introduce 2 new variables in `bufcache.c`, `hits` and `misses`, which we access using the aforementioned syscalls to calculate and compare cache hits on a cold cache and an in-use cache.

3.1.2 Output

Raw output:

```
Copying tests/filesys/base/cache-hitrate to scratch partition...
Copying ../../tests/filesys/base/sample.txt to scratch partition...
qemu-system-i386 -device isa-debug-exit -hda /tmp/LHyf1ATMOb.dsk -m 4 -net none -nographic
-monitor null
WARNING: Image format was not specified for '/tmp/LHyf1ATMOb.dsk' and probing guessed raw.
         Automatically detecting the format is dangerous for raw images, write operations
         on block 0 will be restricted.
         Specify the 'raw' format explicitly to remove the restrictions.
PiLo hda1
Loading...........
Kernel command line: -q -f extract run cache-hitrate
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer...  472,678,400 loops/s.
hda: 5,040 sectors (2 MB), model "QM00001", serial "QEMU HARDDISK"
hda1: 200 sectors (100 kB), Pintos OS kernel (20)
hda2: 4,096 sectors (2 MB), Pintos file system (21)
hda3: 132 sectors (66 kB), Pintos scratch (22)
filesys: using hda2
scratch: using hda3
Formatting file system...done.
Boot complete.
Extracting ustar archive from scratch device into file system...
Putting 'cache-hitrate' into the file system...
Putting 'sample.txt' into the file system...
Erasing ustar archive...
Executing 'cache-hitrate':
(cache-hitrate) begin
(cache-hitrate) end
cache-hitrate: exit(0)
Execution of 'cache-hitrate' complete.
Timer: 148 ticks
Thread: 9 idle ticks, 70 kernel ticks, 70 user ticks
hda2 (filesys): 39 reads, 299 writes
hda3 (scratch): 131 reads, 2 writes
Console: 961 characters output
Keyboard: 0 keys pressed
```

```
Exception: 0 page faults
Powering off...
```

Raw results:

```
PASS
```

3.1.3 Potential Kernel Bugs

Two non-trivial potential kernel bugs that would affect the outcome of this test are:

- If the kernel is unable to correctly execute the `open` syscall (invalid filename, etc.), the test would fail.
- If the kernel is unable to correctly execute the `read` syscall (read short, etc.), the test would fail.

3.2 Test 2

3.2.1 Feature Tested and Test Overview

Our second test, `cache-coalesce.c`, tests the buffer cache's ability to coalesce writes to the same sector. We accomplish this by following the scheme mentioned in the project spec. We create a syscall which allows us to access the block device's `write_cnt` in userspace. We write a >64 KB file byte-by-byte and then read it byte-by-byte to ensure that the total number of writes is on the order of 128 writes.

3.2.2 Output

Raw output:

```
Copying tests/filesys/base/cache-coalesce to scratch partition...
qemu-system-i386 -device isa-debug-exit -hda /tmp/8A0pm4cZAD.dsk -m 4 -net none -nographic
-monitor null
WARNING: Image format was not specified for '/tmp/8A0pm4cZAD.dsk' and probing guessed raw.
         Automatically detecting the format is dangerous for raw images, write operations
         on block 0 will be restricted.
         Specify the 'raw' format explicitly to remove the restrictions.
PiLo hda1
Loading............
Kernel command line: -q -f extract run cache-coalesce
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer...  467,763,200 loops/s.
hda: 5,040 sectors (2 MB), model "QM00001", serial "QEMU HARDDISK"
hda1: 200 sectors (100 kB), Pintos OS kernel (20)
hda2: 4,096 sectors (2 MB), Pintos file system (21)
hda3: 109 sectors (54 kB), Pintos scratch (22)
filesys: using hda2
scratch: using hda3
Formatting file system...done.
Boot complete.
Extracting ustar archive from scratch device into file system...
Putting 'cache-coalesce' into the file system...
Erasing ustar archive...
Executing 'cache-coalesce':
(cache-coalesce) begin
(cache-coalesce) end
cache-coalesce: exit(0)
Execution of 'cache-coalesce' complete.
Timer: 563 ticks
Thread: 33 idle ticks, 61 kernel ticks, 469 user ticks
hda2 (filesys): 294 reads, 609 writes
hda3 (scratch): 108 reads, 2 writes
Console: 926 characters output
Keyboard: 0 keys pressed
Exception: 0 page faults
Powering off...
```

Raw results:

```
PASS
```

3.2.3 Potential Kernel Bugs

Two non-trivial potential kernel bugs that would affect the outcome of this test are:

- If the kernel is unable to correctly create a buffer file with the `create` syscall (not enough memory, etc.), the test would fail.
- If the kernel is unable to correctly execute the `seek` syscall, the test would fail.

3.3 Experience Writing Tests

There are no tests for the buffer cache in the current testing suite. In addition to this, we noticed that reading from and writing to directories can be tested more rigorously. In writing these tests, we initially ran into some problems correctly linking the syscalls we introduced for these tests. After some careful reasoning, we were able to identify the issues that were preventing our tests from behaving as expected.