Project File System Design

Group 36

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

Buffer Cache

Data Structures and Functions

Structures (filesys/inode.c):

```
struct buffer_cache_entry { // Element of available_cache list
  uint8_t block[BLOCK_SECTOR_SIZE]; // Cache block
  block_sector_t block_id; // Unique to each block
  bool valid; // Indicate if block is valid
  bool dirty; // Indicate if block has been written to
  struct list_elem elem; // Element of available_cache list
};
struct arriving_cache_entry { // Element of arriving_cache list
  block_sector_t block_id; // Unique to each block
  struct condition cond;
  int ref_cnt; // Free struct when 0, for synchronization
  struct list_elem elem; // Element of arriving_cache list
};
```

Global data (filesys/inode.c):

```
struct buffer_cache_entry buffer_cache[64]; // 64 cache blocks
struct list available_cache; // Available blocks in buffer cache
struct list arriving_cache; // Cache blocks being fetched from disk
struct lock buffer_cache_lock; // Synchronize buffer cache
struct condition arriving_cond; // If evicting, wait if no blocks can be evicte
d
```

Cache functions (filesys/inode.c):

These helper functions replace previous calls to block_read and block_write in filesys/inode.c, and are used to interface with the buffer cache

```
void buffer_cache_access(block_sector_t block_id, uint8_t* buffer, int block_offset, int num_bytes, bool write, void* aux);
void buffer_cache_write_memcpy(uint8_t* block, uint8_t* buffer, int block_offset, int num_bytes);
void buffer_cache_write_memset(uint8_t* block, uint8_t* buffer, int block_offset, int num_bytes);
void buffer_cache_read(uint8_t* block, uint8_t* buffer, int block_offset, int num_bytes);
```

Algorithms

Overview:
The available_cache Pintos list stores available cache blocks for reading/writing. The arriving_cache Pintos list stores cache blocks being fetched from the block device (i.e., disk).

Cache initialization (filesys/inode.c):
In the inode_init function,

- Initialize Pintos lists available_cache and arriving_cache. Also, initialize the buffer_cache_lock lock and arriving_cond condition variable.
- Set all buffer_cache_entry blocks in the buffer_cache as invalid.

Cache access (filesys/inode.c):
The buffer_cache_access helper function interfaces with the cache for all read/write operations with the block device. This helper function replaces the bounce buffer and previous calls to block_read and block_write in filesys/inode.c file.

Cache search:
When searching for a block in the buffer cache, the associated block ID is searched in available_cache list or arriving_cache list. This takes linear time.

Pseudocode of buffer_cache_access:

```
Acquire buffer_cache_lock
Search VALID block in cache. If not found, evict cache block + fetch new block.
  Release buffer_cache_lock before cache block write-back + fetch.
  Reacquire buffer_cache_lock after cache block write-back + fetch.
Read/write to buffer cache
Release buffer_cache_lock
```

The aux parameter of buffer_cache_access is a function that does the read/write operation on a cache block. There are 3 aux functions.

1. buffer_cache_write_memcpy: function copies num_bytes bytes from pointers buffer to block + block_offset.

2. buffer_cache_write_memset: function sets num_bytes bytes at block + block_offset to a byte store in buffer.
3. buffer_cache_read: copies num_bytes from pointers block + block_offset to buffer.

In filesys/inode.c, the buffer_cache_access function replaces previous calls to block_read and block_write, as follows.

- In functions inode_open and inode_read_at, replace calls to block_read with buffer_cache_access, with parameters write set to false and aux set to buffer_cache_read.
- In functions inode_create and inode_write_at, replace calls to block_write with buffer_cache_access, with parameter write set to true.
  - If data is written from a buffer, set aux to buffer_cache_write_memcpy and set the buffer parameter.
  - If data is set to a specific value (e.g., 0), set aux to buffer_cache_write_memset and set the buffer parameter to a uint8_t value (e.g., 0).
- In function inode_close, write the inode_disk to the block device, by calling buffer_cache_access on the inode→data buffer, with aux set to buffer_cache_write_memcpy.

Block replacement policy (LRU):

- Cache access: cache block is pushed to head of available_cache list every block read/write operation.
- Write: cache block is marked as dirty when written to.
- Block eviction: cache block at tail of available_cache list is popped. If block is dirty, it is written to block device, following write-back policy.
- Block fetch: cache block fetched from the fs_device block device for read/write is marked as valid and not dirty, and pushed to head of available_cache list.

Block search + eviction + fetch:

Overview:

- Blocks are searched by block_id, which is a block's index in the block device.
- For synchronization, a block search occurs while the buffer_cache_lock is held.
- For concurrency, release buffer_cache_lock during cache block write-back + fetch, reacquire lock afterwards
- During cache eviction and block fetch, a buffer_cache_entry is popped from tail of available_cache list. If the popped block is dirty, it is written back to the block device. The buffer_cache_entry is updated with the fetched bock from the block device, and pushed to the head of available_cache list.

Pseudocode:

```
While true:
  Search available_cache list. If VALID block found, DONE.
  Search arriving_cache list. If block ID found:
    Cond_wait on condition variable in corresponding arriving_cache_entry
    Continue // Repeat entire search, going to line 1
  Otherwise:
    Malloc + set arriving_cache_entry, add to arriving_cache list
    While empty(available_cache), cond_wait on arriving_cond
    Evict block from available_cache list, fetch requested block from block dev
ice
```

```
    Push new block to head of available_cache list
    Pop arriving_cache_entry from arriving_cache list
    Cond_broadcast to condition variable in arriving_cache_entry
    Cond_signal to arriving_cond condition variable
    Done.
```

Since arriving_cache_entry is a shared malloc struct, the free must be synchronized.

- When initialized, the ref_cnt is set to 1. After cond_broadcast (line 12), decrement ref_cnt. If 0, free the struct.
- Before cond_wait (line 4), increment the ref_cnt. After cond_wait (line 4), decrement ref_cnt. If 0, free the struct.

System shutdown (filesys/filesys.c):
During the system shutdown, the dirty, valid cache blocks in the available_cache list must be written back to the block device. This will be implemented in the filesys_done function.

- In the filesys_done function, search the available_cache list. For each dirty, valid cache block, write it back to the block device, using calls to the block_write function.

Synchronization

- Access to buffer cache is serial, as threads hold the buffer_cache_lock during cache block search in available_cache and arriving_cache + read/write operation to found cache block. This ensures updates to cache are synchronized.
- During cache access, a block write/read with block device may occur. When this happens, the buffer_cache_lock is released to increase concurrency (IO operation is blocking). For synchronization, the evicted buffer_cache_entry is popped from available_cache list while buffer_cache_lock is held, so no other thread can access it while an eviction occurs.
- If all 64 blocks are being evicted simultaneously (ie, available_cache list is empty), a thread waits on arriving_cond. When an eviction completes and a cache block is added to the available_cache list, the evicting thread calls cond_signal on arriving_cond to wake up any sleeping threads. This ensures only 64 blocks can be evicted at a time.
- When a cache block is being fetched from the block device, an arriving_cache_entry, storing the block ID and a condition variable, is added to arriving_cache list. If a thread requires data from a block in arriving_cache list, it waits on the associated condition variable. The fetching thread calls cond_broadcast on the condition variable after the cache block is fetched and added to available_cache list, and pops the associated arriving_cache_entry from arriving_cache list. The woken up threads re-search the buffer cache for the same block, and then execute a read/write operation.
- The arriving_cache_entry is malloced and shared by multiple threads. To synchronize memory free, a ref_cnt is tracked. The ref_cnt is synchronized by buffer_cache_lock. When a thread accesses the struct, it increments the ref_cnt. When a thread is finished with the struct, it decrements the ref_cnt. When the ref_cnt is 0, the struct can be deallocated (freed).

Rationale

- Our proposed solution is simple to conceptualize and requires a moderate quantity of code. The use of helper functions makes the design easy to extend for future functionality. For example, the use of the aux function parameter in buffer_cache_access function allows additional cache operations to be easily added.

- The cache stores 64 blocks, each of size 512B. Thus, the cache occupies 64KiB of space in memory, plus some metadata. The cache is large, but significantly reduces disk operations, which incur large access times.
- The current design results in serial cache access, which limits concurrency. A few changes can be made to improve concurrency.
  - A read-write lock could replace the buffer_cache_lock to improve concurrency of read operations.
  - Each cache block could have its own lock (e.g., read-write lock), instead of a global cache lock. This allows threads to access different cache blocks concurrently.
- As the cache is fully associative, searching the available_cache list is a linear time operation. However, as the list size is at most 64, searching the list is fast.

Extensible Files

Data Structures and Functions

Data structures:

```
struct inode_disk { // Fits in 1 disk block
  off_t length; // File size in bytes
  block_sector_t dp[122]; // Direct pointer
  block_sector_t ip; // Indirect pointer
  block_sector_t dip; // Double indirect pointer
  unsigned magic; // Magic number, identifies inode
  ...
};
struct inode {
  struct lock extension_lock; // For serialized file extension
  struct lock access_lock; // For inode access synchronization
  // Other attributes ...
};
static struct lock free_map_lock; // filesys/free-map.c, global lock
static struct lock open_inodes_lock; //filesys/inode.c, global lock
```

Additional functions

```
void syscall_inumber(struct intr_frame* f, int fd); // userprog/syscall.c
bool inode_file_resize(struct inode* inode, off_t size); // filesys/inode.c
```

Algorithms

Inode structure:

Pointer type	Total bytes

122 Direct pointers	$122 \times 512 = 62464$
1 Indirect pointer	$128 \times 512 = 65536$
1 Double indirect pointer	$128^2 \times 512 = 8388608$

$$\text{maximum file size} = 122 \times 512 + 128 \times 512 + 128^2 \times 512 = 62464 + 65536 + 8388608 = 8,516,608 > 8MiB$$

System call overview:
To add a new syscall, complete the following steps in the syscall_handler function.
1. Identify system call type with a switch case statement.
2. Validate user memory of syscall arguments, then copy as needed to kernel space.
3. Call a syscall helper function with validated arguments to execute the respective syscall.

System call inumber:
In syscall_inumber function,
1. Call get_fdt_entry(fd) to get corresponding fdt_entry.
2. Call inode_get_inumber(fdt_entry->file->inode) to get the inumber. Store result in f->eax.

File resize
Function signature:

```
bool inode_file_resize(struct inode* inode, off_t size);
```

The inode_file_resize function is structured into 3 stages. Stage 1 handles direct pointers, stage 2 handles indirect pointers, and stage 3 handles double indirect pointers.
- Disk access is not detailed in pseudocode. For any disk access (read/write), use the buffer_cache_access function, which interacts with the disk's buffer cache and reads/writes from disk when necessary.

Stage 0:
This stage checks if the resize is possible, by checking if the requested size does not exceed the maximum file size.

```
if (size < 0 || size > (122 + 128 + 128*128) * BLOCK_SECTOR_SIZE)
  return false;
```

Stage 1 (direct pointers):
This stage checks all direct pointers in the inode, and grows/shrinks the file as necessary.

```
block_sector_t *dp = inode->data.dp; // Direct pointer (DP)
for (int i = 0; i < 122; i++) { // Iterate over all DP
  if (size <= i * BLOCK_SECTOR_SIZE && dp[i] != 0) { // Shrink file condition
    free_map_release(dp[i], 1); // Free block assoicated with dp[i]
    inode->data->dp[i] = 0; // Mark dp[i] as unallocated
  } else if (size > i * BLOCK_SECTOR_SIZE && dp[i] == 0) { // Grow file conditi
on
    if (!free_map_allocate(1, &dp[i])) // Allocate new block for DP
      return false; // Return false if allocation fails
    memset(*dp[i], 0, BLOCK_SECTOR_SIZE); // Initialize block to all zeros
  }
```

```
}
```

Stage 2 (indirect pointers):
This stage checks all direct pointers contained in the indirect pointer in the inode, and grows/shrinks the file as necessary.

```
block_sector_t *ip = &inode->data.ip; // Indirect pointer (IP)
if (ip[0] == 0 && size <= 122 * BLOCK_SECTOR_SIZE) { // Check if IP needed
  inode->data.length = size;
  return true;
}
if (ip[0] == 0) { // Allocate IP if unallocated
  free_map_allocate(1, &ip[0]);
}
for (int i = 0; i < 128; i++) { // Iterate over all DP in IP
  dp = ip[i];
  // Use DP logic with new conditions
  // Shrink if (size <= (122 + i) * BLOCK_SECTOR_SIZE && ip[i] != 0)
  // Grow if (size > (122 + i) * BLOCK_SECTOR_SIZE && ip[i] == 0)
}
if (size <= 122 * BLOCK_SECTOR_SIZE) { // Unallocate IP if not needed
  free_map_release(ip[0], 1);
  ip[0] = 0;
}
```

Stage 3 (double indirect pointers):
This stage checks all indirect pointers and direct pointers contained in the double indirect pointer in the inode, and grows/shrinks the file as necessary.

```
block_sector_t *dip = &inode->data.dip; // Double indirect pionter (DIP)
if (dip[0] == 0 && size <= (122 + 128) * BLOCK_SECTOR_SIZE) { // Check if DIP n
eeded
  inode->data.length = size;
  return true;
}
if (dip[0] == 0) { // Allocate DIP if unallocated
  free_map_allocate(1, &dip[0]);
}
ip = *dip[0]; // Indirect pointers in DIP
for (int i = 0; i < 128; i++) { // Iterate over all IP in DIP
  if (ip[i] == 0 && size <= (122 + 128 + 128*i) * BLOCK_SECTOR_SIZE)
```

```
    break; // Exit loop if no more IP's need updates
  dp = *ip[i]; // Direct pointers in IP
  for (int j = 0; j < 128; j++) { // Iterate over all DP in IP
    // Use DP logic with new conditions
    // Shrink if (size <= (122 + 128 + 128*i + j) * BLOCK_SECTOR_SIZE && dp[j]
!= 0)
    // Grow if (size > (122 + 128 + 128*i + j) * BLOCK_SECTOR_SIZE && dp[j] ==
0)
  }
  if (size <= (122 + 128 + 128*i) * BLOCK_SECTOR_SIZE) { // Free IP not needed
    free_map_release(ip[i], 1);
    ip[i] = 0;
  }
}
if (size <= (122 + 128) * BLOCK_SECTOR_SIZE) { // Unallocate DIP if not needed
  free_map_release(dip[0], 1);
  dip[0] = 0;
}
inode->data.length = size;
return true;
```

Rollback on disk space exhaustion:
If there is not enough disk space when extending the size of a file (when free_map_allocate function returns false), then we must rollback by freeing all the sectors that were just allocated. Rollback is achieved by calling the inode_file_resize function again on the inode's **pre-extension** file size.
Pseudocode for rollback on disk space exhaustion:

```
if (!inode_file_resize(inode, size)) { // Extend file
  // Resize to original length if extension fails.
  // inode->data.length is not updated if inode_file_resize() fails
  inode_file_resize(inode->data.length);
}
```

Byte to sector function (filesys/inode.c):
The byte_to_sector function returns the block device sector that contains the byte offset pos within the inode. With the new inode file structure, this function must be updated to find the correct sector block from a direct pointer, indirect pointer, or double indirect pointer in the inode.

Pseudocode:

```
static block_sector_t byte_to_sector(const struct inode* inode, off_t pos) {
  int dp_index = pos / BLOCK_SECTOR_SIZE; // Calculate the DP index
```

```
  if (dp_index < 122) { // Check direct pointers
    if (dp[dp_index] != 0) return dp[dp_index]; // Return block sector if alloc
ated
  } else if (dp_index < 122 + 128) { // Check indirect pointer
    if (ip[0] != 0 && ip[0][dp_index - 122] != 0)
      return ip[0][dp_index - 122]; // Return block sector if allocated
  } else if (dp_index < 122 + 128 + 128*128) { // Check double indirect pointer
    int ip_index = (dp_index - 122 - 128) / 128;
    dp_index = (dp_index - 122 - 128) % 128;
    if (dip[0] != 0 && dip[0][ip_index] != 0 && dip[0][ip_index][dp_index] !=
0)
      return dip[0][ip_index][dp_index]; // Return block sector if allocated
  }
  return -1; // If DP/IP/DIP block not allocated, return error value
}
```

<u>File extension on write (filesys/inode.c):</u>
If a file is written beyond the EOF point, the file must be extended. To achieve this, a check can be added at the beginning of the inode_write_at function. For synchronization, file extensions are serialized with a lock.

```
Acquire inode's extension_lock
If write_offset + write_size > file_size: // Check if extension is necessary
  If !inode_file_resize(inode, write_offset + write_size): // Increase file len
gth
    inode_file_resize(inode->data.length) // Rollback if extension fails
    return 0; // No bytes written to
}
Release inode's extension_lock
```

<u>File extension on create (filesys/inode.c):</u>
In the inode_create function, the initial file block size of an inode is set. This function must be updated to accommodate the new inode file structure.

- The inode_file_resize function will be used to set the new file size, instead of manually allocating blocks with free_map_allocate and setting blocks to zero with block_write.

<u>File removal on close (filesys/inode.c):</u>
When a file is closed, it could be removed. With the new inode file structure, call inode_file_resize on size 0 to deallocate the entire file before the disk inode is deallocated.

<u>File read past EOF (filesys/inode.c):</u>
In inode_read_at function, if the inode is read past the EOF point, then no bytes should be read. This is achieved by adding a check at the beginning of inode_read_at.

```
if (read_offset > file_size) return 0; // Return 0 if no bytes can be read
if (read_offset + read_size > file_size)
  read_size = file_size - read_offset; // Only read upto EOF
```

File seek/write/read past EOF:

- In the current implementation, Pintos file seek beyond EOF without extending the file.
- When writing beyond EOF, inode_file_resize is called before disk blocks are written to. The inode_file_resize zeros out newly allocated blocks.
- When reading beyond EOF, 0 bytes are read (as detailed in the previous section).

Synchronization

As per the Project 3 spec, we will remove our global file system lock. As a result, we need to use other synchronization techniques to ensure proper synchronization. More specifically, since each process will only have one thread, we need to synchronize things that are **shared between processes** and do not need to synchronize things that are **per process**.

- We will synchronize things **shared between processes:**
    - inode metadata
        - We will synchronize access to an inode's metadata because inodes are shared between processes. To do this, we will create a new lock, access_lock, in the struct inode to synchronize access to the inode metadata. Whenever we need to access the inode metadata, we will acquire the inode's access_lock. This will mainly be done in the inode functions (e.g. inode_open, inode_read_at, inode_write_at) and any other relevant functions that need to access inode metadata (e.g. isdir).
    - open_inodes list
        - We will synchronize access to the open_inodes list. To do this, we will create a global lock open_inodes_lock**,** which will be acquired whenever we need to access the open_inodes list to read/modify it.
    - free map
        - We will synchronize access to the free map by creating a global lock in free-map.c, which we will acquire whenever we need to access the free map (e.g. when calling free_map_allocate and free_map_release). The lock should be released when the free_map_file is being updated (e.g., when bitmap_write is called), to prevent holding the lock when performing blocking IO operations.
    - File extensions should be serialized. To achieve this, each struct inode stores an extension_lock. File extensions occur while the inode's corresponding extension_lock is held, making the operation serial.
- We do **NOT** need to synchronize things that are **per process**:
    - For example, we do not need to implement any new synchronization techniques for accessing file descriptor tables, file structs, or dir structs because each process has its own file descriptor table.

Rationale

- The inode_file_resize function makes it simple to implement file extension and roll back. The proposed design requires a moderate quantity of code and is a little difficult to comprehend; however, this is to be expected as disk management is a complex process.
- The inode structure involves 122 direct pointers, 1 indirect pointer, and 1 double indirect pointer. The purpose of 122 direct pointers and 1 indirect pointer allows fast access for small files by minimizing disk accesses. The 1 double indirect pointer allows support for larger files, upto 8MiB. The proposed system follows the Linux's FFS scheme, which aims to provide fast access to small files while supporting large files. 1 inode fits into 1 block sector on disk.

- The inode_file_resize function runs in linear time of file size, for both file shrink and file growth. This is unavoidable, as an indexed inode has block data dispersed across the disk, meaning each file block must be checked/allocated.
- For the roll back algorithm on file extension, we originally considered going through all recently allocated sectors and calling free_map_release. However, a simpler solution was to call inode_file_resize a second time, with size equal to the **pre-extension** file size. This approach utilizes less code and is simpler to comprehend.

Subdirectories

Data Structures and Functions

Structures:

```
struct process { // userprog/process.h
  struct dir* cwd; // Current working directory
  // Other attributes ...
};
struct fdt_entry { // userprog/process.h
  struct dir* dir; // Directory associated with a file descriptor
  // Other attributes ...
};
struct inode_disk { // filesys/inode.c
  uint32_t is_dir; // 4B bool, indicates if file is directory
  block_sector_t parent_dir; // Block address of parent directory
  // Other attributes ...
};
```

Path resolution functions (filesys/directory.c)

```
struct inode* dir_resolve_path(char* filepath);
static int get_next_part(char part[NAME_MAX + 1], const char** srcp);
```

System call functions (userprog/syscall.c):

```
void syscall_chdir(struct intr_frame* f, const char* dir);
void syscall_mkdir(struct intr_frame* f, const char* dir);
void syscall_readdir(struct intr_frame* f, int fd, char* name);
void syscall_isdir(struct intr_frame* f, int fd);
```

Algorithms

Current working directory (userprog/process.c):

Each process tracks its current working directory (cwd) in its PCB.

- In userprog_init function, set cwd of first process to the root directory, by calling dir_open_root function.

- A child process inherits its parent process's cwd. In process_execute function, the parent process copies its cwd by calling dir_reopen function, then passes the cwd copy into start_process function. Inside start_process function, the child process sets the copy cwd as its own cwd.
- In process_exit function, call dir_close on the current process's cwd before the PCB is destroyed.

Adding to a directory:

- When we add to a directory and it is already full, our logic for inode_write_at() already uses resizing when writing past the end of a file, so there's no need to change dir_add().

Path Resolution:

Given a path (absolute or relative), the dir_resolve_path function will return the inode (file or directory) that a path corresponds to. The implementation uses the get_next_part helper function provided in the project spec. This helper function parses a path and serially extracts file/directory names.

Pseudocode:

```
struct inode* dir_resolve_path(char* filepath) {
  struct inode* current_inode = inode_open(root if absolute, CWD if relative);
  char current_part[NAME_MAX + 1];
  int result;
  while ((result = get_next_part(current_part, &filepath)) > 0) {
    if (!current_inode->data.is_dir) // Inode not directory, but not at end of
path
      inode_close(current_inode), return NULL;
    dir = dir_open(current_inode); // Open directory to lookup current part
    if (current_part is ".") current_inode = inode_reopen(dir->inode);
    else if (current_part is "..")
      current_inode = inode_open(dir->inode->data.parent_dir);
    else { // Look up inside directory
      if (!dir_lookup(dir, current_part, &current_inode)
        dir_close(dir), return NULL; // Current part not in directory, invalid
path
    }
    dir_close(dir); // Close directory after lookup of current part
    }
  }
  if (result == -1) inode_close(current_inode), return NULL; // Invalid path
  return inode;
}
```

Filesys Functions (filesys/filesys.c):

To implement absolute/relative path use in the file system, the following filesys functions are updated to use the dir_resolve_path function. Pseudocode is provided below.

```
bool filesys_create(const char* name, off_t initial_size) {
  // Split name into directory path and file name
```

```
  // Use dir_resolve_path() to extract inode corresponding to directory path
  // Call dir_open() on returned inode to get the directory
  // If dir_open() doesn't return NULL
    // Allocate and create a new inode with free_map_allocate() and inode_creat
e()
    // Add to the directory with dir_add()
    // Close the directory and return true
  // Else
    // Release allocated memory and return false
}
```

```
struct file* filesys_open(const char* name) {
  // Split name into directory path and file name
  // Use dir_resolve_path() to extract inode corresponding to directory path
  // Call dir_open() on returned inode to get the directory
  // If dir_open() doesn't return NULL
    // Use dir_lookup() to get the corresponding inode to the file
  // Close the directory
  // Return file_open() called on the inode corresponding to the file
}
```

For filesys_remove, our proposed implementation prevents removing a directory if it is open by a process (e.g., is a current working directory).

```
bool filesys_remove(const char* name) {
  // Split name into directory path and file name
  // Use dir_resolve_path() to extract inode corresponding to filepath
  // If inode represents a file, call dir_remove on file, return true.
  // Else (inode represents a directory)
    // If inode's open_cnt == 1 && dir_readdir() returns false (dir is empty)
      // Call dir_remove() on the directory
      // Return true
    // Close the inode
  // Return false
}
```

System call overview:

To add a new syscall, the below steps are completed in the syscall_handler function.

1. Identify system call type with a switch case statement.
2. Validate user memory of syscall arguments, then copy as needed to kernel space.
3. Call a syscall helper function with validated arguments to execute the respective syscall.

System call open (userprog/syscall.c):

In the syscall_open function, after a file is opened by calling filesys_open, determine if file represents directory by checking file→inode→data.is_dir.

- If directory, store dir_open(inode_reopen(file→inode)) in fdt_entry→dir, and call file_close on the file. Set fdt_entry→file to NULL. If dir_open returns NULL value, set f→eax to -1 and return early.
- If not directory, store file in fdt_entry→file and set fdt_entry→dir to NULL.

System call close (userprog/syscall.c):

In syscall_close function, after getting the fdt_entry:

- If fdt_entry→dir != NULL (file is a directory), call dir_close on fdt_entry→dir.
- Otherwise (file is not a directory), call file_close on fdt_entry→file.

System call read (userprog/syscall.c):

In syscall_read function,

- In the default case: (not reading from STD_IN or STD_OUT)
  - If fdt_entry→dir != NULL (is a directory), set f→eax to -1 and return. This prevents user programs from reading a directory using read syscall.

System call write (userprog/syscall.c):

In syscall_write function,

- In the default case: (not writing to STD_IN or STD_OUT)
  - If fdt_entry→dir != NULL (is a directory), set f→eax to -1 and return. This prevents user programs from writing to a directory file using a write syscall.

System call exec (userprog/syscall.c):

This system call does not need to be updated. The parent process passes its cwd as an argument to start_process from process_execute, so child process can set its cwd to its parent cwd.

System call remove (userprog/syscall.c):

- Our edited filesys_remove() handles removing directories and files, so there is no need to change anything for this syscall.

System call is directory (userprog/syscall.c):

In syscall_isdir function,

- Call get_fdt_entry(fd) to get the corresponding fdt_entry. Set f→eax Boolean value to fdt_entry != NULL and fdt_entry→dir != NULL and return.

System call inumber (userprog/syscall.c):

In syscall_inumber function,

- Call get_fdt_entry(fd) to get the corresponding fdt_entry.
- If fdt_entry→dir != NULL (is a directory), set f→eax to inode_get_inumber(fdt_entry→dir→inode) and return.
- Otherwise (not a directory), set f→eax to inode_get_inumber(fdt_entry→file→inode) and return.

System call change directory (userprog/syscall.c):

In the syscall_chdir function,

- Get the inode from the dir string by calling dir_resolve_path(dir). If inode == NULL or inode→data.is_dir == false, the set f→eax to false, close inode if open, and return early.
- Call dir_close on the current PCB's cwd. Set the PCB's new cwd to dir_open(inode). Set f→eax to true and return.

System call read directory (userprog/syscall.c):

In the syscall_readdir function,

- Call get_fdt_entry(fd) to get the corresponding fdt_entry. If fdt_entry == NULL or fdt_entry→dir == NULL, set f→eax to false and return early.
- Set f→eax to dir_readdir(fdt_entry→dir, name) and return.

System call make directory (userprog/syscall.c):
In the syscall_mkdir function,

- If string dir == NULL, set f→eax to false and return early.
- Split the string dir into a directory path (i.e., prefix) and directory name (i.e., suffix). Get dir_inode by calling dir_resolve_path on directory path.
  - If dir_inode == NULL or dir_inode→data.is_dir == false, then set f→eax to false, close dir_inode if open, and return.
  - Otherwise, call dir_open on dir_inode.
- Determine directory name does not exist by checking dir_lookup on resolved directory and directory name returns false. If true, close the returned inode, set f→eax to false, and return.
- Call free_map_allocate to get a block sector to store the directory inode. Then call dir_create on the inode sector, with initial_size set to 16. Call dir_add on the inode_sector, directory name, and the resolved directory.
  - If any function fails, set f→eax to false, free resources (e.g., inodes, directories) and return.
  - Otherwise, set f→eax to true and return.

Synchronization

- The inode metadata (e.g., open_cnt, data, removed) is referenced across multiple threads in system calls (e.g., syscall_isdir, syscall_mkdir). Anytime the inode metadata is accessed, the inode's access_lock must be held, to ensure read/write operations are atomic to the inode.
- The data inside struct dir and struct file is **per process**. As there is only one thread **per process**, these structs' data does not need to be synchronized.
- In the proposed solution, a directory can only be removed when no other references to it exist and it is empty. To ensure no thread updates the directory while it is being removed, threads should be prevented from opening the directory (inside inode_open) once the inode is marked for removal.

Rationale

- Our proposed implementation is relatively simple to conceptualize and implement in code. By design, our code is modularized across many functions. This ensures functions can be implemented independently, making debugging and code development easier.
- We implemented a helper function to resolve relative/absolute paths. This helper function is used across the filesys functions. Accordingly, this function reduces code repetition and complexity. Moreover, it creates an abstraction layer when dealing with long file paths.
- Our dir_resolve_path() function runs linearly with respect to the number of items in the filepath, but runs rather slowly because of all the necessary pointer accesses needed for going from one directory to the next. An alternative would be to cache commonly used file paths so that the accessing a common file paths occurs quickly.

Concept check

1. Write-Behind Functionality:
   - To implement write-behind functionality, we can create a kernel thread which runs in the "background" and flushes the buffer cache periodically, e.g. every 10 seconds. First, we can create a helper function called buffer_cache_flush() that flushes the dirty blocks in the buffer cache and writes them to the file system block device. Then, we can create another function

buffer_cache_flush_thread() which runs in a loop, calls buffer_cache_flush(), and calls a non-blocking implementation of timer_sleep(10s). We will then call thread_create() to create a new kernel thread which executes the buffer_cache_flush_thread() function.

2. Read-Ahead Functionality:
    - To implement read-ahead functionality, we can take advantage of spatial locality. When we fetch a requested block sector from disk (which may be stored in the buffer), we can also fetch the next block sector. To make the second fetch be asynchronous, a new thread can handle the fetching of the next block sector.
    - More specifically, when reading a block from the block device (which may be in the buffer), we create a thread to handle the block fetch for the next block. A new thread is created by calling thread_create. The new thread will call buffer_cache_access (detailed in the buffer cache section) with aux set to buffer_cache_read. This function call will fetch the next block from the block device if it is not already in the buffer cache. This ensures that read-ahead/pre-fetching happens in the "background", as the main thread is not blocked on fetching the next block.