EE 461S – Spring 2022

PROJECT 2: USER PROGRAMS

DESIGN DOCUMENT

Malvika Badrinarayanan <malvika.badri@utexas.edu>

Abhijjith Venkkateshraj <abhijjith@utexas.edu>

ARGUMENT PASSING

DATA STRUCTURES

A1: Copy here the declaration of each new or changed `struct' or `struct' member, global or static variable, `typedef', or enumeration. Identify the purpose of each in 25 words or less.

/* Maximum Number of Tokens */

#define MAX_NUM_TOKENS 128

We assumed a max of 128 tokens of 1 byte each based on the Pintos documentation:

"There is a limit of 128 bytes on command-line arguments that the Pintos utility can

pass to the kernel".

ALGORITHMS

A2: Briefly describe how you implemented argument parsing. How do you arrange for the elements of argv[] to be in the right order? How do you avoid overflowing the stack page?

Step 1: Set the esp to *PHYS_BASE*

Step 2: Tokenize the command line using space as delimiter

Step 3: Push the arguments to stack from right to left by decrementing *esp according to the length of each token; Store location of the arguments to a pointer array argv[] in the correct order from argv[0] to argv[argc-1].

Step 4: Calculate the padding required and insert null characters (Padding is calculated based on the esp location after pushing arguments to stack).

Step 5: Push argv[argc] = *NULL* pointer to stack

Step 6: Push the pointer array from Step 3 on to stack from argv[argc-1] to argv[0].

Step 7: Push argv = &argv[0] to stack.

Step 8: Push argc to stack.

Step 9: Push return address to stack.

We can avoid overflowing of the stack page as follows:

a) Minimum size required before tokenizing is 12 bytes:

Return address: 4 bytes

argc: 4 bytes

argv: 4 bytes

b) In a loop:

- Add *strlen(token) + 1 (null character) + 4 bytes(argv[i])* to total size for every argument tokenized.

- If the total size at any point exceeds 1 page (4 kB), request for another page at *(PHYS_BASE – (n * PG_SIZE))*

RATIONALE

A3: Why does Pintos implement strtok_r() but not strtok()?

strtok_r can be called from multiple threads or points simultaneously. It takes an extra argument (context pointer) which stores the state between the calls. The non-reentrant versions use global state and invokes an undefined behavior if called from multiple threads.

SYSTEM CALLS

DATA STRUCTURES

B1: Copy here the declaration of each new or changed `struct' or `struct' member, global or static variable, `typedef', or enumeration. Identify the purpose of each in 25 words or less.

#define MAX_FNAME_LENGTH 16

char name[MAX_FNAME_LENGTH];

The use case of the name field in 'struct thread' has been modified to point to the name of the executable run by the thread. This is used in two places:

a) *sys_exit () in syscall.c*: To display the name of the executable.

b) *isRunning (char *fname) in thread.c*: To determine if 'fname' is a running executable.

B2: Describe how file descriptors are associated with open files. Are file descriptors unique within the entire OS or just within a single process?

- There is a separate file descriptor table maintained for *each process*, which keeps track of all open files for that process. Two fields were added in 'struct thread' to maintain file descriptors:

a) a file descriptor array - *struct file *fdtab[MAX_OPEN_FILES];*

b) and an index - *int nxtfd;*

The nxfd index keeps track of the file descriptor to be assigned to the next opened file.

- Whenever a file is opened, the file structure returned from filesys_open() is copied on to the fdtab[] array at the nxtfd index and nxtfd is incremented.

- When a file is closed, if the file struct element in the fdtab array is not NULL, it is closed using

file_close() and the corresponding entry is made NULL. This ensures that the file is closed only once even if user attempts to reclose.

ALGORITHMS

B3: Describe your code for reading and writing user data from the kernel.

To read user data:

When the user program makes a syscall, it pushes the arguments to

its stack and calls a TRAP instruction to make a call into the kernel which internally calls

the syscall_handler(). Inside the syscall_handler, we perform the following steps to read the user data from the stack:

a) The user stack pointer is obtained from the struct intr_frame f (f->esp).

b) The validity of the usp is checked.

c) The sys call number is obtained from the user stack.
d) Based on the call number, the number of arguments is read from the stack after validating their addresses.

e) The corresponding handler function is called which performs the requested operation.

To write to the user stack:

The value returned from the handler function is stored on the eax of the intr_frame f.

Eg: f->eax = sys_write((int)(*(usp + 1)), (char *)(*(usp + 2)), (int)(*(usp + 3)));

B4: Suppose a system call causes a full page (4,096 bytes) of data to be copied from user space into the kernel. What is the least and the greatest possible number of inspections of the page table (e.g. calls to pagedir_get_page()) that might result? What about for a system call that only copies 2 bytes of data? Is there room for improvement in these numbers, and how much?

TBD.

B5: Briefly describe your implementation of the "wait" system call and how it interacts with process termination.

The wait system call pushes the pid/tid of the child process/thread along with the SYS_WAIT call number onto the user stack and invokes a TRAP system call using INT 0x30. This invokes the syscall handler function where basic sanity checks are performed on the parameters read from the stack and the tid is passed onto the sys_wait function which calls the process_wait function.

We use 3 semaphores (part of the child's thread struct) to synchronize between the parent's wait and the child's exit process.

a. exit_started: The parent waits on this semaphore for the child to reach the process_exit stage.
b. exited: The child waits on this semaphore for the parent to reap its exit status.

c. removed: The parent waits on this semaphore for the child to be removed from the allThreads list. This ensures successive wait calls are handled in the right manner.

B6: Any access to user program memory at a user-specified address can fail due to a bad pointer value. Such accesses must cause the process to be terminated. System calls are fraught with such accesses, e.g. a "write" system call requires reading the system call number from the user stack, then each of the call's three arguments, then an arbitrary amount of user memory, and any of these can fail at any point. This poses a design and error-handling problem: how do you best avoid obscuring the primary function of code in a morass of error-handling? Furthermore, when an error is detected, how do you ensure that all temporarily allocated resources (locks, buffers, etc.) are freed? In a few paragraphs, describe the strategy or strategies you adopted for managing these issues. Give an example.

We have two functions for the error-handling

a. check_args: This is used as a sanity check on the user stack argument's addresses before calling any of the syscall handler functions. This internally calls the check_validity function. On a high level, this checks if the stack argument pointers are within user space.

Eg: case SYS_WRITE:

```
check_args(usp,3);

f->eax = sys_write((int)(*(usp + 1)), (char *)(*(usp + 2)), (int )(*(usp + 3)));

break;
```

In the above example, for every write system call, we do a validity check on the 3 user stack argument's addresses before we invoke the sys_write function. In a case of failure, a page fault occurs and resources are freed.

b. check_validity - Inside each syscall handler function, we individually check if a user pointer points to a valid location in user space. This also makes use of the get_user function which tries

to access the user pointer. If it is successful, it returns a byte value present at that location, else it causes a page fault (invoking the page_fault handler calling the sys_exit function).

SYNCHRONIZATION

B7: The "exec" system call returns -1 if loading the new executable fails, so it cannot return before the new executable has completed loading. How does your code ensure this? How is the load success/failure status passed back to the thread that calls "exec"?

A semaphore named *'loadStatusReaped'* and a boolean named *'loadStatus'* are declared as a part of the child's thread struct.

The success/failure of the load function call is copied into the *'loadStatus'* field inside the child's thread struct. The child waits on the '*loadStatusReaped'* semaphore inside the start_process function after the load function call is complete.

The parent extracts the child thread struct based on the child's tid returned from the thread_create inside the process_execute function. If the loadStatus is a failure, then the parent signals the semaphore and returns -1. If not, the parent signals the semaphore and returns the child tid.

B8: Consider parent process P with child process C. How do you ensure proper synchronization and avoid race conditions when P calls wait(C) before C exits? After C exits? How do you ensure that all resources are freed in each case? How about when P terminates without waiting, before C exits? After C exits? Are there any special cases?

a. If P calls wait(C) before C exits: We must ensure that P waits till C exists to reap its exit status. To do this, we use the 'exit_started' semaphore wherein the parent waits for the child to reach the process_exit stage.

b. If P calls wait after C exits: The child waits on the 'exited' semaphore for the parent to reap its exit status.
(In both the above cases a and b, the parent waits for the child to be removed from the

allThreads list using the ‘removed’ semaphore. This ensures successive wait calls are handled in the right manner)

c. If P exits before C exits without waiting: We ensure that the ‘exited’ semaphore that C would wait on, is incremented (sema_up) so that C doesn’t keep waiting forever for the parent’s wait call. To do so, we do a sema_up of ‘exited’ in ‘thread_exit()’ after process_exit is called.

d. If P exits after C exits without waiting: The child C remains active till P exits.

RATIONALE

B9: Why did you choose to implement access to user memory from the kernel in the way that you did?

We implement access to user memory by first checking the validity of the address (if not NULL or below PHYS_BASE) and then trying to access the location using the get_user function given by the Pintos developers. If it is successful, it returns a byte value present at that location, else it causes a page fault (invoking the page_fault handler calling the sys_exit function). This is done at the beginning of all syscall function handlers. By doing so, we ensure that we proceed with handling the syscall only if all arguments and pointers in the user memory are valid. This is simple and doesn’t require any other complicated handling mechanisms to handle erroneous accesses.

B10: What advantages or disadvantages can you see to your design for file descriptors?

Advantage:

- We maintain a file descriptor table for each process. This ensures that we have a distributed way of handling the file operations i.e there is no need to track which processes are accessing the files.

Disadvantage:

- We do not reuse the file descriptors of closed files i.e only a total of 128 files can be opened by a process during its lifetime.

  Eg: If nxtfd is 5, and fd 3 is closed, the fd assigned to a newly opened file will be 5. 3 will not be assigned again.

B11: The default tid_t to pid_t mapping is the identity mapping. If you changed it, what advantages are there to your approach?

We haven't changed the mapping.

VM- Design Doc

+--------------------------+

| EE 461S |

| PROJECT 3: VIRTUAL MEMORY |

| DESIGN DOCUMENT |

+--------------------------+

Abhijjith Venkkateshraj <abhijjith@utexas.edu>

Malvika Badrinarayanan <malvika.badri@utexas.edu>

SUPPLEMENTAL PAGE TABLE MANAGEMENT

==========

---- DATA STRUCTURES ----

>> A1: Copy here the declaration of each new or changed `struct' or `struct' member, global or static variable, `typedef', or >> enumeration. Identify the purpose of each in 25 words or less.

1) page.h:

i) Structure for supplemental page table entry hash table (SPTE):

```
struct spte {
    struct hash_elem hash_elem;
    uint8_t *upage; /* upage used as a key */
    struct spte_payload payload; /* SPTE is the payload */
};
```

- Supplemental page table implemented as a hash table. The key is the upage and the value is

the spte of the upage.

ii) Structure for the spte payload:

```
struct spte_payload
{
    enum frame_location location; /* where the corresponding frame is present */
    /* Disk related info */
    struct file *file; /* File pointer */
    off_t ofs; /* Offset within the file */
    uint32_t read_bytes; /* No of bytes to read in the page */
    uint32_t zero_bytes; /* Page size - Read bytes */
    bool writable; /* Does the page have write permissions */
    /* SWAP_SPACE related info */
    uint32_t slot; /* Slot where the page is present in the swap space */
    /* PHY_MEM related info */
    uint8_t *kpage; /* Corresponding frame at which the SPTE is mapped
    */ /* VIRTUAL_MEM related info */
    uint8_t *upage; /* Corresponding upage at which the SPTE is mapped
    */ uint32_t *pd; /* Page directory of the process */
    uint32_t *pte; /* Page table entry for the correspoding SPTE */
    struct lock spte_lock; /* SPTE Lock for synchronization purposes */
};
```

- The payload contains the following:

  a) location of the upage (Memory/Swap/Disk)

  b) the file and allied details if its location

is in disk

c) slot if location is in swap space

d) kpage if location is in physical memory

e) upage details (useful when the spte is saved in FT)

f) page directory and page table entry of the upage

g) spte_lock used for locking the spte from getting evicted while file read/write operations are being performed

iii) Modifications to struct thread in thread.h

```
struct thread {
  ..
  /* VM */
  struct hash *spt; /* Supplemental page table implemented as a hash table */
  void *esp; //stack_pointer of thread during a syscall
};
```

- Fields for the supplemental page table (hash table) and esp of the thread added - The esp is used for storing the esp of the user thread during a system call. This is useful to check if the stack needs to be grown when a page fault occurs in the system call handler.

---- ALGORITHMS ----

>> A2: In a few paragraphs, describe your code for accessing the data stored in the SPT about a given page.

A supplemental page table hash map is created for every process. The SPT hash table has key value as the upage and the value is a payload struct containing the following fields:

a) location: Location of the upage (Memory/Swap/Disk)

b) struct file: The file structure if the location of the page is in disk

c) read_bytes: The bytes to read from the file for the given page

d) zero_bytes: The bytes to zero out in the given page post read_bytes

e) writable: Whether or not the page is writable. Used to ensure that code pages are given read-only access

f) slot: slot number if location is in swap space

g) kpage: Pointer to the kpage if location is in physical memory

h) upage: upage number

i) pd: Pointer to page directory of the upage

j) pte: Pointer to page table entry of the upage

k) spte_lock: Used for locking the spte from getting evicted while file read/write operations are being performed

---

Creation and Insertion of SPTE into SPT:

The SPTE for a given page is created and inserted in the SPT of the process in the following scenarios:

i) load_segment() in process.c - The SPTE is created for the upage with location marked as DISK and the corresponding file details (read_bytes, zero_bytes, writable) are updated in the SPTE; pd and pte pointers are updated, spte_lock is initialized and the SPTE is inserted into the SPT.

ii) page_fault() in exception.c - If it is determined that the stack needs to be grown in the page

fault handler, a new SPTE is created for the upage of the fault address (Fault address & ~PGMASK), its location is marked as PHY_MEM, pd and pte pointers are updated, spte_lock is initialized and the SPTE is inserted into the SPT

Accessing the SPTE data

The SPTE data is accessed in the following functions:

i) page_fault() in exception.c - If the user access (or kernel access of user memory) caused a page fault, the upage corresponding to the fault address is looked up in the SPT hash table. If found, according to the location field in the payload, the page is read and installed in the memory as a frame.

ii) pin_frames() and unpin_frames() in syscall.c: The upages of the buffer accessed in sys_write() and sys_read() are pinned to the memory by locking the spte_lock which is a part of the SPTE of the upages. When the swap_evict() in swap.c spte_lock identifies one of these locked frames as possible candidate for eviction, it ignores it when it realizes that the lock is held by the thread which pinned these upages to memory.

iii) process_exit() in process.c: All the SPTEs in the SPT of the process is traversed, and if found to be mapped to a kpage is freed from the frame table mapping.

---- SYNCHRONIZATION ----

>> A3: When two user processes both need a new frame at the same time, how are races avoided?

We use a global lock called the ‘vm_lock’ which is acquired by a thread that requires a frame (acquired before frame_get() is invoked). The clock algorithm we have implemented works on a single clock handle and by locking it, we ensure that only the process which has the lock can move the clock handle i.e at a particular time only one process can obtain a frame. The other process waits on the vm_lock until it is released by the previous process after acquiring a frame.

---- RATIONALE ----

>> A4: Why did you choose the data structure(s) that you did for representing virtual-to-physical mappings?

We used a hash table for the Supplemental Page Table and also the Frame Table. We used a hash table for the following reasons:

i) We do not need to know how big the table should be, as the hashtable can grow automatically as you add entries.

ii) Easy to look up the mapping by using the key of interest. Need not iterate over the elements and implement our own search functions.

PAGING TO AND FROM DISK

==========

---- DATA STRUCTURES ----

>> B1: Copy here the declaration of each new or changed `struct' or

>> `struct' member, global or static variable, `typedef', or >>

enumeration. Identify the purpose of each in 25 words or less.

i) Frame Location Enumeration

```
enum frame_location {
DISK, /* Disk */
SWAP_SPACE, /* Swap space */
PHY_MEM /* Memory */
};
```

- The upage can either be on disk, swap space or physical memory

b) frame.h

i) struct fte {

```
struct hash_elem hash_elem;
uint8_t *kpage; /* kpage used as a key */
```

struct fte_payload payload; /* FTE is the payload */

};

- Frame table implemented as a hash table. The key is the kpage and the value is the fte of the kpage.

ii) struct fte_payload

{

bool isFree; /* Indicates if the frame entry is free for allocation to a upage of a process */

struct spte *spt_entry; /* The frame entry is associated to a unique SPT entry of a process

*/ };

- The frame table entry payload contains the following:

a) Whether or not the kpage (physical frame) is free.

b) If the kpage is not free, the spt_entry points to the supplement page table entry of the

upage which is mapped to the kpage.

c) swap.h

i) struct bitmap *swap_space_bitmap

- Free slots in the swap space is kept track as a bitmap

ii) struct block *swap_block_p;

- Pointer to the BLOCK_SWAP (obtained by using block_get_role()). This is the handle used for all swap block device operation

#define NUM_SWAP_SLOTS 2048

#define NUM_SECTORS_IN_SLOT 8

---- ALGORITHMS ----

>> B2: When a frame is required but none is free, some frame must be

>> evicted. Describe your code for choosing a frame to evict.

We make use of the Clock algorithm to evict a frame.

We define the following global iterator -> struct hash_iterator clock_hand; /* Clock hand used in the Clock algorithm for eviction */

We move the clock hand to the beginning of the frame table hashmap inside frame_init().

The clock algorithm is performed using the following steps. At each and every step, we check for the rollover condition until we have iterated over num_user_pages (keeps track of the number of user pages available from the user pool at startup) and initializes the clock hand back to point to the beginning of the frame table if rollover occurs.

1. First Pass- Loop over all the frame table entries and check if any frames are free in the frame table and can be assigned to a upage
2. Second Pass- Check the Accessed Bit in the PTE for each kpage that is associated in the frame table. If the accessed bit is set, clear it and move the clock hand to the next entry. If the accessed bit is not set, then evict that page.
3. Third Pass- Check the Dirty Bit in the PTE for each kpage that is associated in the frame table. If the dirty bit is set, move the clock hand to the next entry. If the dirty bit is not set, then evict that page.
4. Final Pass- If all pages are accessed and dirty, then we can evict any page. In this case, we evict the page that is pointed by the clock hand initially.

>> B3: When a process P obtains a frame that was previously used by

a >> process Q, how do you adjust the page table (and any other data

>> structures) to reflect the frame Q no longer has?

We make use of the following function to obtain a kpage (frame) for the Process P.

uint8_t frame_get(struct spte *spt_entry)*

The spt_entry that is passed as the argument is that of the Process P.

The frame_get function invokes the swap_evict function with the Process Q's spt_entry. (This is found while iterating over all the frame table entries and figuring out which frame to evict via the clock's algorithm)
*int swap_evict(struct spte *spt_entry)*

If the Process Q's page is dirty, it is put onto the swap space, else, it is put onto the disk. In either case, the kpage associated with the spt_entry is made NULL and the corresponding locations are updated. Also, the corresponding page table entry is cleared using the pd and upage from the spt_entry.

Once Process Q's bookkeeping structures are invalidated, the frame table entry (for which the pages were evicted) will be updated with the Process P's spt_entry and the corresponding kpage is updated. Finally, this mapping is added to the page table entry via the install_page function using the Process P's upage and the kpage obtained from the eviction algorithm. If the frame is read from the swap space, the dirty bit is set in the PTE. The new location of Process P's spt_entry is made to PHY_MEM.

>> B4: Explain your heuristic for deciding whether a page fault for an

>> invalid virtual address should cause the stack to be extended into

>> the page that faulted.

Stack Growth conditions:

1) *esp - fault address < = 32 bytes (PUSH instruction)

2) Fault address within PHYS_BASE and *esp

If any of the above conditions are satisfied

- We create a new SPT entry. Fill the SPT entry with the corresponding information and insert it into the SPT of the current process.
- Get a frame of memory using the frame_get() and the obtained kpage will be stored in the upage's spte.
- Finally, this mapping is added to the page table entry via the install_page function using the upage (from input) and the kpage obtained from the frame_get().

---- SYNCHRONIZATION ----

>> B5: Explain the basics of your VM synchronization design. In
>> particular, explain how it prevents deadlock. (Refer to the

>> textbook for an explanation of the necessary conditions for

>> deadlock.)

We have two locks for the synchronization

1. struct lock vm_lock; /* Global VM lock used for synchronization purposes */ 2. struct lock spte_lock; /* Local SPTE Lock within each process for synchronization purposes */

We have used these two locks in a way that “Circular wait” will be completely eliminated. The locks are carefully placed such that one process that has acquired vm_lock and is waiting for spte_lock will never come in contact with another process that has acquired spte_lock and waits for the vm_lock.

>> B6: A page fault in process P can cause another process Q's frame

>> to be evicted. How do you ensure that Q cannot access or modify

>> the page during the eviction process? How do you avoid a race

>> between P evicting Q's frame and Q faulting the page back in?

We have one global lock named “vm_lock” used for synchronization purposes.

Any page fault that occurs results in pagefault_handler() getting invoked in exception.c. Before any major operation related to VM is performed in this handler, we acquire the global vm_lock so that any process that gets page faulted later has to wait before the previous page fault is handled and completed.

In this case, the process P evicts the Process Q's frame via the eviction algorithm using the frame_get and the swap_evict functions and the corresponding spt_entry's are updated for both Process P and Q. Now, when Process Q tries to access the page, it gets page faulted and calls the page fault handler where it waits on the global vm_lock that Process P had acquired previously. This global lock is released only when Process P has completed the entire page fault handling process.

>> B7: Suppose a page fault in process P causes a page to be read from

>> the file system or swap. How do you ensure that a second process Q

>> cannot interfere by e.g. attempting to evict the frame while it is

>> still being read in?

We have one global lock named "vm_lock" used for synchronization purposes.

Any page fault that occurs results in pagefault_handler() getting invoked in exception.c. Before any major operation related to VM is performed in this handler, we acquire the global vm_lock so that any process that gets page faulted later has to wait before the previous page fault is handled and completed.

In this case, the process P is page faulted, invokes the pagefault_handler() and reads the page from the disk or swap space. The process P acquires the global vm_lock in this process. Now, when Process Q gets page faulted and calls the page fault handler, it waits on the global vm_lock that Process P had acquired previously. This global lock is released only when Process P has completed the entire page fault handling process. Hence, Process Q cannot interfere by attempting to evict the frame while it is still being read in by process P.

>> B8: Explain how you handle access to paged-out pages that occur

>> during system calls. Do you use page faults to bring in pages (as

>> in user programs), or do you have a mechanism for "locking" frames

>> into physical memory, or do you use some other design? How do you

>> gracefully handle attempted accesses to invalid virtual addresses?

The spte_lock is initialized whenever an SPT entry is added to the process's SPT. The spte_lock's are acquired to provide the frame's pinning functionality and the spte_lock's are released to provide the frame's unpinning functionality.

The pinning and unpinning functionalities are invoked inside the sys_read and sys_write functions. (i.e) When a process P is exercising the read and write operations for a certain number of pages/frames, we bring in those pages to memory if not present through page fault invoked in get_user() functionality and once the pages are brought in, the corresponding frames/pages are excluded from the clock's eviction algorithm via the spte_lock. During the eviction algorithm exercised by another process in parallel, it tried to acquire the spte_lock associated with the choice of frame to be evicted. If it acquires the spte_lock, then it means that the frame is not currently used by any other active process in the system.

---- RATIONALE ----

>> B9: A single lock for the whole VM system would make

>> synchronization easy, but limit parallelism. On the other hand,

>> using many locks complicates synchronization and raises the

>> possibility for deadlock but allows for high parallelism. Explain

>> where your design falls along this continuum and why you chose to

>> design it this way.

We have two locks for the synchronization

> 1. struct lock vm_lock; /* Global VM lock used for synchronization purposes */ 2. struct lock spte_lock; /* Local SPTE Lock within each process for synchronization purposes */

We have used these two locks in a way that "Circular wait" will be completely eliminated. The locks are carefully placed such that one process that has acquired vm_lock and is waiting for spte_lock will never come in contact with another process that has acquired spte_lock and waits for the vm_lock.

We chose the above design so that we allow concurrency and parallelism among the processes. For example, with the current design, we allow one process to do the system reads and writes while another process is performing the eviction algorithm (by either bringing in a page from the disk or swap space).

EE 461S – Spring 2022

PROJECT 4: FILE SYSTEM

DESIGN DOCUMENT

---- GROUP ----

>> Fill in the names and email addresses of your group members.

Malvika Badrinarayanan <malvika.badri@utexas.edu>

Abhijjith Venkkateshraj <abhijjith@utexas.edu>

---- PRELIMINARIES ----

>> If you have any preliminary comments on your submission, notes for the

>> TAs, or extra credit, please give them here.

>> Please cite any offline or online sources you consulted while

>> preparing your submission, other than the Pintos documentation, course

>> text, lecture notes, and course staff.

INDEXED AND EXTENSIBLE FILES

==========

---- DATA STRUCTURES ----

>> A1: Copy here the declaration of each new or

changed `struct' or `struct' member, global or static variable, `typedef', or enumeration. Identify the purpose of each in 25 words or less.

1) Modified struct inode_disk- removed start and magic fields, added type, direct pointer array, single indirect pointer and double indirect pointer fields

```
struct inode_disk {
      off_t            length;     /* File size in bytes. */
      uint32_t         fd_type;    /* File/Directory */
      block_sector_t   dptr[124];   /* Direct Pointer */
      block_sector_t   siptr;      /* Single Indirect Pointer */
      block_sector_t   diptr;      /* Double Indirect Pointer */
};
```

>> A2: What is the maximum size of a file supported by your inode structure? Show your work.

Excluding the one sector used for the inode diskstruct storage,

We have 124 direct pointers: 124 * 512 = 63488 bytes

A single pointer which in turn can hold 128 direct pointers = 128 * 512 = 65536 bytes

A double pointer which in turn has 128 single pointers = 128 * 128 * 512 = 8388608 bytes

Total = 8,517,632 bytes ~= 8MB

---- RATIONALE ----

>> A6: Is your inode structure a multilevel index? If so, why did you choose this particular combination of direct, indirect, and doubly indirect blocks? If not, why did you choose an alternative inode >> structure, and what advantages and disadvantages does your structure have, compared to a multilevel index?

We used a combination of direct pointers, single indirect pointers and double indirect pointers.

The direct pointers are the fastest to look up and works for most of the files which are within 62kB. For slightly larger files (upto 126kB) we look up using the single direct pointer. For much larger files, we use a doubly indexed pointer with a trade off for lookup speed. Using this sort of a multi level indexing gives us the best performance.

SUBDIRECTORIES

==========

---- DATA STRUCTURES ----

>> B1: Copy here the declaration of each new or changed `struct' or

>> `struct' member, global or static variable, `typedef', or

>> enumeration. Identify the purpose of each in 25 words or less.

1) Changed the fd table in struct thread to hold structures of type fd_entry:

```
struct fd_entry *fdtab[MAX_OPEN_FILES]; /* File Desciptor Table */
```

2) Created struct fd_entry:

```
/* Struct for file descriptor table entries */

struct fd_entry {

      struct file *f;   /* File Struct of File/Directory */

      uint32_t dir_pos;    /* Current position of directory */

      bool isOpen;      /* Open Status of the fd */

};
```

3) Added working_dir to struct thread

```
struct dir *working_dir;
```

4) Created #defines for fd type- FILE OR DIRECTORY

```
#define FD_TYPE_FILE         0
#define FD_TYPE_DIRECTORY    1
```

---- ALGORITHMS ----

>> B2: Describe your code for traversing a user-specified path. How

>> do traversals of absolute and relative paths differ?

In order to support the traversal of relative paths and for the chdir system call, we keep track of the current working directory inside the thread struct.

```
struct dir *working_dir;
```

Absolute paths are detected when the path starts with '\', otherwise it is assumed to be a relative

path (where the traversal starts from the current working directory).

Once the relative/absolute paths are sorted, we tokenize the provided user path by having ‘\’ as the delimiter. We then iterate through the path to get the parent directory of the directory (last before token).. The last token indicates the directory/path to be created.

Eg: For the mkdir system call, the following is perfomed:

- Allocate a free sector for the directory, create a directory and add it to the parent directory's entry list
- Set inode type as directory
- Write modified inode to disk for persistence
- Add '.' and '..' as entries into the new directory

---- RATIONALE ----

>> B6: Explain why you chose to represent the current directory of a

>> process the way you did.

In order to support the traversal of relative paths and for the chdir system call, we keep track of the current working directory inside the thread struct.

```
struct dir *working_dir;
```

Absolute paths are detected when the path starts with ‘\’, otherwise it is assumed to be a relative path (where the traversal starts from the current working directory).

```
    /* Get Start Directory from path */
```

```
      if(dir[0] == '/')

      {

      /* Absolute path */

      cur_dir = dir_open_root();

      }

      else

      {

      /* Relative path */

      if(thread_current()->working_dir == NULL)

      {

            /* Pintos thread has no current working dir */

            cur_dir = dir_open_root();

      }

      else

      {

            /* Current working dir is the start directory */

            cur_dir =
dir_open(inode_open(thread_current()->working_dir->inode->sector));

      }

      }
```