CS 162 Summer 2021 Operating Systems and Systems Programming

Final Exam

INSTRUCTIONS

This is your exam. Complete it either at exam.cs61a.org or, if that doesn't work, by emailing course staff with your solutions before the exam deadline.

This exam is intended for the student with email address `<EMAILADDRESS>`. If this is not your email address, notify course staff immediately, as each exam is different. Do not distribute this exam PDF even after the exam ends, as some students may be taking the exam in a different time zone.

For questions with **circular bubbles**, you should select exactly *one* choice.

○ You must choose either this option

○ Or this one, but not both!

For questions with **square checkboxes**, you may select *multiple* choices.

☐ You could select this choice.

☐ You could select this one too!

You may start your exam now. Your exam is due at <DEADLINE> Pacific Time. Go to the next page to begin.

This is a **proctored, closed-book exam**. During the exam, you may **not** communicate with other people regarding the exam questions or answers in any capacity. If there is something in a question that you believe is open to interpretation, please use the "Clarifications" button to request a clarification. We will issue an announcement if we believe your question merits one.

Make sure you read through the exam completely before starting to work on the exam.

We will overlook minor syntax errors in grading coding questions. You do not have to add necessary `#include` statements.

(a) Name

(b) Student ID

(c) Please read the following honor code: "I understand that this is a closed book exam. I hereby promise that the answers that I give on the following exam are exclusively my own. I understand that I am allowed to use unlimited **handwritten** cheat-sheets of my own making, but otherwise promise not to consult other people, physical resources (e.g. textbooks), or internet sources in constructing my answers." **Type your full name below to acknowledge that you've read and agreed to this statement.**

1. **True/False**

   Please **EXPLAIN** your answer in **TWO SENTENCES OR LESS** (Answers longer than this may not get credit!). Answers without any explanation **GET NO CREDIT.**

   (a) (3 points)

   i. In the Nth chance algorithm (assume N=10), if a page is evicted, this means that the page has not been accessed in the past 10 memory accesses.

   ○ True

   [✓ False]

   ii. Explain your answer.

   False. If a page is evicted, this means that it has not been accessed after 10 full sweeps of the clock hand.

(b) (3 points)

i. In an RPC system with multiple servers, dynamic binding is generally more fault tolerant than static binding.

[✓ True]

◯ False

ii. Explain your answer.

True. With dynamic binding, if there is a failure in one server, the RPC client can select a different server at runtime. With static binding, since the endpoint is determined at compile time, this is not possible.

(c) (3 points)

i. If the following functions are run concurrently, deadlock may occur.

```
int val1 = 0;
int val2 = 0;

void functionA() {
  val1 = 1;
  while(test&set(&val2));
  val1 = 0;
  val2 = 0;
  printf(“Done!\n”);
}

void functionB() {
  val2 = 1;
  while(!compare&swap(&val1, 0, 1));
  val1 = 0;
  val2 = 0;
  printf(“Done!\n”);
}
```

[✓ True]

False

ii. Explain your answer.

True. In this problem, the first two lines of each function implement lock acquisition, while the second two lines of each function implement lock release.However, if both threads acquire different locks by running `val1 = 1` and `val2 = 1`, then there will be a circular wait, hold & wait, and mutual exclusion when each thread tries to acquire the lock it doesn't hold.

(d) (3 points)

i. If a system is using Round Robin scheduling and there are T threads running, each thread will have to wait exactly `[(T - 1) x Q]` ticks in between bursts of computation (assume that context switching is instantaneous).

○ True

[✓ False]

ii. Explain your answer.

False. Each thread will only wait *exactly* `[(T - 1) x Q]` ticks if none of the threads voluntarily yield the CPU or block before their quanta expires. Without this assumption, a thread may wait less than `[(T - 1) x Q]` ticks before running again.

(e) (3 points)

i. When a file is opened using `fopen`, a pointer to a `FILE` object is returned. A copy of this object is stored in kernel space.

○ True

[✓ False]

ii. Explain your answer.

False. `fopen` is a library function that runs in userspace. As a result, the `FILE` object is only allocated in userspace.

(f) (3 points)

i. Monty Mole is sending a packet from his computer to Mario's computer. However, the TCP header of the packet becomes corrupted before the packet is sent out. The packet might still reach the destination machine.

[✓ True]

○ False

ii. Explain your answer.

True. The IP network layer/packet header is responsible for routing a packet to a destination IP address. Even though the IP layer is responsible for best-effort delivery, which means that the packet may not reach its destination or it may be corrupted, there is still a good chance that the packet will be delivered to the destination machine.

(g) (3 points)

i. In the TCP protocol, if a packet is unacked, it has not reached its destination.

○ True

[✓ False]

ii. Explain your answer.

False. An unacked packet may have reached its destination, and the ACK message from the receiver may have been lost/corrupted.

(h) (3 points)

i. Monty Mole uses a Hard Disk Drive with multiple platters. Unfortunately, the arm on the HDD is broken and can no longer move. Now, Monty Mole will not be able to read from/write to more than a single sector on his disk.

○ True

[✓ False]

ii. Explain your answer.

False. Because Monty Mole's HDD has multiple platters, the HDD arm will be able to read from/write to one sector on each platter.

(i) (3 points)

i. Assume that Thread A and Thread B are in the same process. If Thread A tries to access memory address 0xABCD and this triggers a page fault, then it is guaranteed that Thread B will also page fault if it attempts to access the same memory address immediately after.

○ True

[✓ False]

ii. Explain your answer.

False. Only one counterexample is necessary. Counterexample 1: Address 0xABCD may be read-only. If Thread A attempted to write to the address, while Thread B read from it, this could cause the behavior described in the problem. Counterexample 2: Thread A could have page faulted because the page was paged out to disk. Once the page was read into memory, Thread B would not page fault on the same access.

(j) (3 points)

i. In the context of storage devices and filesystems, blocks and sectors are not synonymous.

[✓ True]

False

ii. Explain your answer.

True. Sectors are a physical location on a storage device. Blocks are a group of sectors that the OS can address.

2. Multiple Choice

In the following multiple choice questions, **please select all options that apply**. Answering a question instead of leaving it blank will **NOT** lower your score (the minimum score for a single question is 0, not negative).

(a) (3 pt) Device drivers: (select all that apply)

- [✓ run in kernel mode]
- ☐ must poll the device hardware to determine if I/O has completed
- ☐ can only interface with block devices, such as Hard Disk Drives
- ☐ are part of an I/O device's hardware
- ☐ None of the above

(b) (3 pt) Monty Mole has created a new scheduling algorithm LJF, a non-preemptive algorithm that always schedules the longest available job and runs it to completion. Assuming that all jobs arrive at the same time and never block, which of the following scheduling algorithms are guaranteed to have the same throughput as LJF? Select all that apply. *Hint: do not assume that context switching is instantaneous.*

- [✓ SJF]
- ☐ Round Robin
- [✓ FCFS]
- ☐ MLFQS
- ☐ None of the above

(c) (3 pt) What are the possible outputs of this program? Select all that apply.

```
void *echo_1(void *arg) {
  char *cmd = "echo";
  char *argv[3];
  argv[0] = cmd;
  argv[1] = "CS";
  argv[2] = NULL;
  execvp(cmd, argv);
  return NULL;
}

void *echo_2(void *arg) {
  char *cmd = "echo";
  char *argv[3];
  argv[0] = cmd;
  argv[1] = "162";
  argv[2] = NULL;
  execvp(cmd, argv);
  return NULL;
}

int main() {
  pthread_t thread1, thread2;
  pthread_create(&thread1, NULL, echo_1, NULL);
  pthread_create(&thread2, NULL, echo_2, NULL);
  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);
}
```

☐ CS162

☐ 162CS

■ CS

■ 162

☐ None of the above

(d) **(3 pt)** Let's say that we are using Chord for a distributed data storage system. Each of the nodes and their associated IDs are displayed below.

A client wants to lookup a key that hashes to 3, but it is only aware of node 32. Which nodes will receive a lookup request? Select all that apply.

- ☐ 4
- ☐ 8
- ☐ 15
- ☐ 20
- ■ 32
- ■ 35
- ■ 44
- ■ 58
- ☐ None of the above

Assume that our system is using a journaling file system, and the current state of the log is as follows:

(e)
```
<OPERATION 1>
COMMIT
<OPERATION 2>
<OPERATION 3>
COMMIT
<OPERATION 4>
```

i. **(2 pt)** Which of the operations are guaranteed to have been applied already?

- ☐ `<OPERATION 1>`
- ☐ `<OPERATION 2>`
- ☐ `<OPERATION 3>`
- ☐ `<OPERATION 4>`
- ■ None of the above

ii. **(2 pt)** Which of the operations are guaranteed to eventually be applied?

- ■ `<OPERATION 1>`
- ■ `<OPERATION 2>`
- ■ `<OPERATION 3>`
- ☐ `<OPERATION 4>`
- ☐ None of the above

3. Terminator Protocol

(a) (4 pt) In 2PC, when a Worker has voted COMMIT but the Coordinator crashes before sending a GLOBAL-COMMIT/ABORT message, why can't the Worker abort without running a termination protocol?

The goal of 2PC is to ensure that all Workers eventually make the same decision when deciding whether or not to perform an action requested by the Coordinator. If Worker A votes COMMIT and is waiting for a message from the Coordinator, there is a chance that all the other Workers also voted COMMIT, and the Coordinator logged a GLOBAL-COMMIT. If this is the case, Worker A must commit and perform the requested action to remain consistent with all other Workers, so it cannot prematurely abort.

(b) Monty Mole wants to throw a party, but he'll only host the party if a majority of his friends commit to attending. He decides to use a modified version of 2PC to decide what to do. In this version of 2PC, he (the Coordinator) will send a VOTE-REQ message to his N friends (the Workers). If at least N/2 friends send a VOTE-COMMIT message back, Monty Mole will host the party and send a GLOBAL-COMMIT message to everyone. Otherwise, he will send a GLOBAL-ABORT message and cancel the party. Everything else about the 2PC algorithm remains the same.

i. (4 pt) Recall the termination protocol from lecture, in which a Worker node A is waiting for a GLOBAL-COMMIT/ABORT message from the Coordinator. In this protocol, the Worker contacts a friendly Worker P, and does the following:

- Case 1: If P has decided COMMIT/ABORT, A makes the same decision.
- Case 2: If P has not decided, P votes to ABORT and tells A to abort.
- Case 3: If P has voted COMMIT, P also needs to wait and can't help A make a decision.

Explain why this termination protocol might fail given Monty Mole's modified 2PC scheme.

Case 2 will fail in Monty Mole's modified scheme. This part of the termination protocol assumes that a single ABORT vote will cause the Coordinator to send a GLOBAL-ABORT, which is true in the original 2PC scheme. However, in the modified scheme, even with a single ABORT vote, a majority of Workers can still vote to COMMIT, causing the Coordinator to send a GLOBAL-COMMIT. As a result, Worker P's vote alone cannot determine the outcome of the system.

ii. (5 pt) Describe why Monty Mole's modified 2PC may be slower than regular 2PC in certain cases.

In the regular 2PC, the GLOBAL-ABORT state can be reached quicker. This is because the Coordinator would only need to wait for a single Worker to vote ABORT. On the other hand, in modified 2PC, the Coordinator would need to receive at least N/2 ABORT votes.

4. Broken Clocks

Monty Mole bought a faulty machine! According to the vendor, the machine uses a single-level paging scheme for address translation and the Clock Algorithm for page eviction. The vendor promised that the processor had hardware-supported "use" and "modified" bits. However, after doing some testing, Monty Mole has discovered that the "use" bit is only set during read accesses, not during write accesses. In addition, the "modified" bit is never set by the MMU during a write access.

Monty Mole has asked for your help in implementing a version of the Clock Algorithm that will function correctly on this faulty hardware. **For this problem, you can assume that all pages are writable (i.e. not read-only).**

Your solution should be the fastest possible implementation of the Clock Algorithm, given these constraints.

(a) **(2 pt)** Each Page Table Entry (PTE) consists of the following fields:

`Physical Page Number (PPN) [23 bits]`	`Valid [1 bit]`	`Writable [1 bit]`	`Use [1 bit]`	`Modified [1 bit]`	`OS Metadata [5 bits]`

Currently, when a page is first loaded into memory, the page's PTE is initialized with Valid=1, Use=0, Modified=0, and Writable=1. What change(s) do you need to make to these initial values to implement a functioning version of the Clock Algorithm on Monty Mole's faulty hardware?

Set Writable=0.

(b) **(7 points)**

The data structure below is a C representation of the PTE described earlier. The numbers associated with each `struct` member (e.g. `uint32 ppn : **23**`) represent how many bits are used to store each value in memory, and are based on the table above.

```
typedef struct pte {
  uint32 ppn : 23;
  bool present : 1;
  bool writable : 1;
  bool use : 1;
  bool modified : 1;
  uint32 os : 5; // You will not need to use these bits in your solution.
} pte_t;
```

Every time a page fault occurs, the function `handle_page_fault`, shown below, is invoked. Every time a page needs to be evicted, the function `run_clock_algo`, shown below, is invoked to perform the Clock Algorithm. Note that `handle_page_fault` calls `handle_page_not_present`, which in turn calls `run_clock_algo` if eviction is necessary.

```
/* This function is only invoked if a page is not present in memory. In
   addition to normal page fault logic (e.g. running the clock algorithm using
   `run_clock_algo`), this function will initialize the PTE as you specified in
   Part 1 for any new pages loaded into memory. */
void handle_page_not_present(pte_t *pte);

/* This function takes in a single argument `pte`, which is a pointer to the PTE
   of the page that caused the page fault. Any changes made to `pte` will modify
   the actual page table entry. */
void handle_page_fault(pte_t *pte) {
  handle_page_not_present(pte);
}

/* Runs the clock algorithm, and returns the VPN (i.e. the array index) of the
   page that should be evicted. `pte_arr` is a pointer to the page table
   (represented as an array of PTEs). `num_pages` is the number of entries in the
   page table. `clock_hand` is a value from 0 to `num_pages`.*/
size_t run_clock_algo(pte_t *pte_arr, size_t num_pages, size_t *clock_hand) {
  while (ptr_arr[*clock_hand].use) {
    pte_arr[clock_hand].use = 0;
    *clock_hand = (*clock_hand + 1) % num_pages;
  }

  size_t old_clock_hand = *clock_hand;
  *clock_hand = (*clock_hand + 1) % num_page;
  return old_clock_hand;
}
```

Your task is to modify the `handle_page_fault` and `run_clock_algo` functions. You will add any logic that is necessary for the Clock Algorithm to function correctly on Monty Mole's faulty hardware. In addition, you need to make sure that the faulty `modified` bit is set manually when a page becomes dirty. To this end, there is an additional boolean `write_access` passed into `handle_page_fault`, specifying whether the memory access that triggered the page fault was a write operation or not. You can assume that whatever change(s) you specified in Part 1 have already been made (see `handle_page_not_present`).

For each blank, you may use as many lines as you need.

```
void handle_page_fault(pte_t *pte, bool write_access) {
  ________[A]________
  handle_page_not_present(pte);
}

size_t run_clock_algo(pte_t *pte_arr, size_t num_pages, size_t *clock_hand) {
  while (ptr_arr[*clock_hand].use) {
    ________[B]________
    pte_arr[clock_hand].use = 0;
    *clock_hand = (*clock_hand + 1) % num_pages;
  }

  size_t old_clock_hand = *clock_hand;
  *clock_hand = (*clock_hand + 1) % num_page;
  return old_clock_hand;
}
```

i. [A]

```
if (pte->present) {
  if (write_access) pte->modified = 1;
  pte->writable = 1;
  pte->use = 1;
}
```

ii. [B]

```
pte_arr[clock_hand].writable = 0;
```

(c) **(4 pt)** For this question, we assumed that all pages will be writable. Why would you need to employ a slower solution if some pages were read-only?

If some pages were read-only, this information would be encoded in the Writable bit of the PTE. One solution would be to store the read-only status of a page in the OS and page fault on every memory access to determine if a page is read-only. Another solution is to page fault on every memory access when the use bit is 0 (by setting the present bit equal to the use bit) to check if a memory access is a write operation and update the use bit if necessary. Either solution is slower.

5. Notorious TLB

Monty Mole is experimenting with a new type of TLB designed specifically for multi-level page tables. Recall that multi-level page tables break up virtual addresses into three components instead of two:

Virtual P1 Index	Virtual P2 Index	Offset

where the Virtual P1 Index and Virtual P2 Index make up the Virtual Page Number.

However, instead of mapping Virtual Page Numbers to Physical Page Numbers, Monty Mole's TLB will map a Virtual P1 Index to the corresponding second-level page table. In other words, Monty Mole's TLB only caches the first level of address translation in a multi-level paging scheme.

For this problem, you will assume that Monty Mole's machine uses a two-level paging scheme. Each page is 1KiB, physical memory is 1 GiB large, and each process has a 32-bit virtual address space. You should also assume that each page table fits exactly within a single page.

(a) **(3 pt)** If Monty Mole's TLB is fully associative and contains 32 blocks, how large is the TLB in bits? Assume that each cache line of the TLB only contains the cache tag, the cache data, and a valid bit (no other metadata is necessary).

1 KiB page -> 10 bit offset 32 bit virtual address - 10 bit offset = 22 bit VPN VPN / 2 = Virtual P1/P2 Index; 22 bits / 2 = 11 bits
1 GiB physical address space / 1 KiB page = 2^20 pages -> 20 bit PPN
Cache tag = Virtual P1 Index = 11 bits Cache data = Second-Level Page Table PPN = 20 bits Valid bit = 1 bit 11 + 20 + 1 = 32 bits 32 blocks * 32 bits = 1024 bits

(b) **(5 pt)** Given the two-level paging scheme described earlier, explain what kinds of memory access patterns would cause Monty Mole's TLB to perform better than a traditional TLB.

If Monty Mole accesses many distinct pages that share the same Virtual P1 Index, but doesn't use each page frequently, this can result in better performance on his custom TLB.

6. Synchronization

(a) Wait

Consider the functions below. Assume a single thread of control, the main thread, is running `int main(void)`, and the main thread creates a helper thread.

```
int main(void) {
  pthread_t thread;
  printf("Main thread start\n");
  pthread_create(&thread, NULL, func, NULL);
  printf("Main thread end\n");
}

void* func(void* arg) {
  printf("Helper thread start\n");
  printf("Helper thread end\n");
}
```

Your job is to use synchronization primitives to ensure the output of the program is:

```
Main thread start
Helper thread start
Helper thread end
Main thread end
```

without using a semaphore.

i. (4 points)

Implement the desired behavior using only locks OR explain why it is not possible.

For each blank, you may use as many lines as you need.

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

int main(void) {
  pthread_t thread;
  printf("Main thread start\n");
  ________[A]________
  pthread_create(&thread, NULL, func, NULL);
  ________[B]________
  printf("Main thread end\n");
}

void* func(void* arg) {
  ________[C]________
  printf("Helper thread start\n");
  printf("Helper thread end\n");
  ________[D]________
}
```

A. If it is **possible**, write your implementation here.

B. [A]

C. [B]

D. [C]

E. [D]

F. If it is **not possible**, explain why here.

ii. (6 points)

Implement the desired behavior using condition variables OR explain why it is not possible.

For each blank, you may use as many lines as you need.

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
________[A]________

int main(void) {
  pthread_t thread;
  printf(‘‘Main thread start\n’’);
  ________[B]________
  pthread_create(&thread, NULL, func, NULL);
  ________[C]________
  printf(‘‘Main thread end\n’’);
}

void* func(void* arg) {
  ________[D]________
  printf(‘‘Helper thread start\n’’);
  printf(‘‘Helper thread end\n’’);
  ________[E]________
}
```

A. If it is **possible**, write your implementation here.

B. [A]

C. [B]

D. [C]

E. [D]

F. [E]

G. If it is **not possible**, explain why here.

(b) (6 points) Flip Flop

Consider the functions below. Assume a single thread of control, the main thread, is running `int main(void)`, and the main thread creates a helper thread.

```
int main(void) {
  pthread_t thread;
  pthread_create(&thread, NULL, func, NULL);
  printf(“1”);
  printf(“2”);
  printf(“5”);
  printf(“6”);
}

void* func(void* arg) {
  printf(“3”);
  printf(“4”);
  printf(“7”);
  printf(“8\n”);
}
```

Your job is to use synchronization primitives to ensure the output of the program is:

```
12345678
```

only using semaphores. You may not interchange the relative ordering of statements we have written. You may add statements wherever you like, but they can only be related to synchronization.

For each blank, you may use as many lines as you need.

```
________[A]________

int main(void) {
  pthread_t thread;
  ________[B]________
  pthread_create(&thread, NULL, func, NULL);
  printf(“1”);
  printf(“2”);
  ________[C]________
  printf(“5”);
  printf(“6”);
  ________[D]________
}

void* func(void* arg) {
  ________[E]________
  printf(“3”);
  printf(“4”);
  ________[F]________
  printf(“7”);
  printf(“8\n”);
  ________[G]________
}
```

i. [A]

ii. [B]

iii. [C]

iv. [D]

v. [E]

vi. [F]

vii. [G]

(c) (6 points) Concurrent Binary Tree

Consider the following binary tree implementation of a map which supports two operations: `put` and `get`. Both the keys and values of the map are integers.

```
struct node {
  int key;
  int val;
  struct node* left;
  struct node* right;
};

/* Returns the value for a key, or -1 if not found */
int get(struct node* node, int key) {
  // Base Case: Not Found
  if (node == NULL)
    return -1;

  // Recurse
  if (key == node->key)
    return node->val;
  else if (key < node->key)
    return get(node->left, key);
  else
    return get(node->right, key);
}

/* Initializes a new node with a <key, val> pairing */
struct node* init_node(int key, int val) {
  struct node* new_node = (struct node*) malloc(sizeof(struct node));
  new_node->key = key;
  new_node->val = val;
  new_node->left = NULL;
  new_node->right = NULL;
  return new_node;
}

/* Updates map to include <key, val> pairing */
void put(struct node** root, int key, int val) {
  *root = put_helper(*root, key, val);
}

struct node* put_helper(struct node* node, int key, int val) {
  if (node->key == key)
    node->val = val;
  else if (key < node->key) {
    if (node->left == NULL) {
      node->left = init_node(key, val);
    } else {
      node->left = put_helper(node->left, key, val);
    }
  } else {
    if (node->right == NULL) {
      node->right = init_node(key, val);
    } else {
      node->right = put_helper(node->right, key, val);
    }
  }

  return node;
}
```

Your job is to make read and write operations to the tree concurrent and sufficiently granular:

- Two read operations should always be able to proceed concurrently
- A read and a write should be able to proceed concurrently
- Two write operations should be able to proceed concurrently unless they both need to read or write the same node

You may assume that memory reads and writes are atomic, and you should use this fact to limit the amount of synchronization code you write.

Below, we've created a modified concurrent tree, which contains a per-node lock that is properly initialized. To synchronize the binary tree, you will specify where we should add `pthread_mutex_lock(&node->lock)` and `pthread_mutex_unlock(&node->lock)` calls in the existing `put` function. Line numbers have been provided so you can identify where to insert these function calls.

```
struct node {
  int key;
  int val;
  struct node* left;
  struct node* right;
  pthread_mutex_t lock; // NEW
};

/* Returns the value for a key, or -1 if not found */
int get(struct node* node, int key) {
  // Base Case: Not Found
  if (node == NULL)
    return -1;

  // Recurse
  if (key == node->key)
    return node->val;
  else if (key < node->key)
    return get(node->left, key);
  else
    return get(node->right, key);
}

/* Initializes a new node with a <key, val> pairing */
struct node* init_node(int key, int val) {
  struct node* new_node = (struct node*) malloc(sizeof(struct node));
  new_node->key = key;
  new_node->val = val;
  new_node->left = NULL;
  new_node->right = NULL;
  new_node->lock = PTHREAD_MUTEX_INITIALIZER; // NEW
  return new_node;
}

/* Updates map to include <key, val> pairing */
void put(struct node** root, int key, int val) {
  *root = put_helper(*root, key, val);
}

struct node* put_helper(struct node* node, int key, int val) {
  if (node->key == key) {                               // 1
    node->val = val;                                    // 2
  }                                                     // 3
  else if (key < node->key) {                           // 4
    if (node->left == NULL) {                           // 5
      node->left = init_node(key, val);                 // 6
    } else {                                            // 7
      node->left = put_helper(node->left, key, val);   // 8
    }                                                   // 9
  } else {                                              // 10
    if (node->right == NULL) {                          // 11
      node->right = init_node(key, val);                // 12
    } else {                                            // 13
      node->right = put_helper(node->right, key, val); // 14
    }                                                   // 15
  }                                                     // 16
  return node;                                          // 17
}                                                       // 18
```

i. BEFORE which lines should we insert `pthread_mutex_lock(&node->lock);`. Select all that apply.

- ☐ 1
- ☐ 2
- ☐ 3
- ☐ 4
- ☐ 5
- ☐ 6
- ☐ 7
- ☐ 8
- ☐ 9
- ☐ 10
- ☐ 11
- ☐ 12
- ☐ 13
- ☐ 14
- ☐ 15
- ☐ 16
- ☐ 17
- ☐ 18
- ☐ None of the above

ii. BEFORE which lines should we insert `pthread_mutex_unlock(&node->lock);`. Select all that apply.

- ☐ 1
- ☐ 2
- ☐ 3
- ☐ 4
- ☐ 5
- ☐ 6
- ☐ 7
- ☐ 8
- ☐ 9
- ☐ 10
- ☐ 11
- ☐ 12
- ☐ 13
- ☐ 14
- ☐ 15
- ☐ 16
- ☐ 17
- ☐ 18
- ☐ None of the above

(d) (8 points) Semaphore with Futex

Implement a semaphore with the futex system call. Your implementation should not trap to the kernel on a down call where the value of the semaphore is positive and the semaphore is uncontested, but is allowed to trap to the kernel on every up call.

We have given you the definition of a semaphore and a function `atomic_decr_if_pos`, which you should use in your implementation.

For each blank, you may use as many lines as you need.

```
typedef struct semaphore {
  int val;
} sema_t;

/* Atomically decrements val if it is positive. Returns true if the value
   was positive and decremented, false if it was not. */
bool atomic_decr_if_pos(int* val) {
  int curr_val;
  do {
    if ((curr_val = *val) <= 0)
      return false;
  } while (!compare&swap(val, curr_val, curr_val - 1));
  return true;
}

void sema_down(sema_t* sema) {
  _________[A]_________
}

void sema_up(sema_t* sema) {
  _________[B]_________
}
```

i. [A]

ii. [B]

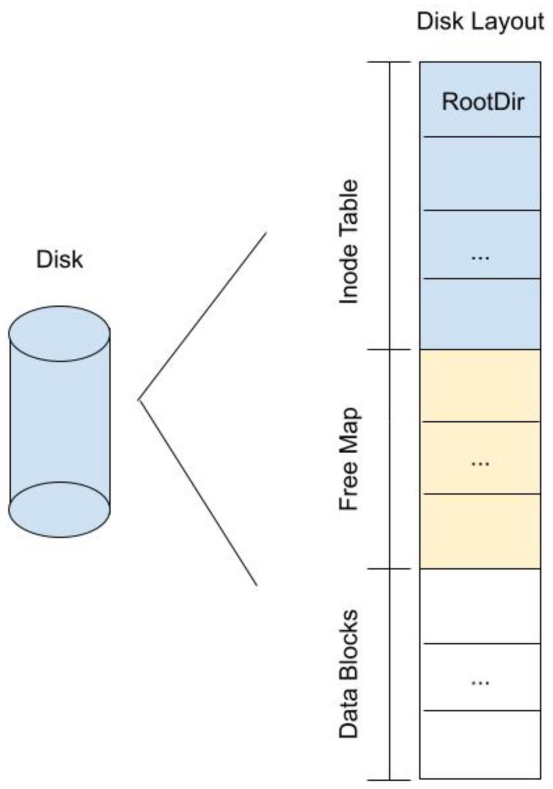
7. Filesystems

(a) Overview

Monty Mole wants to create a new operating system Tacos, which is heavily based on Pintos. Tacos is exactly the same as Pintos, except it has a different file system, outlined below (Monty Mole doesn't quite understand that plagiarism). The architecture is as follows:

The disk has 32 bit block addresses and a block size of 512 B (like Pintos, blocks are one sector long). The first 512 MB of disk are dedicated to the inode table, which is a fixed size (does not expand). Each file is given an inode, and its inumber is its index into the inode table. Directories are considered files (as always), and the root directory is hardcoded to be the first inode (in the 0th block of disk). The next segment of disk is dedicated to persistently storing the free map, and the last segment of disk is for data blocks.

Monty Mole uses a global file system lock when accessing the file system, because he was too lazy to reason about finer-grained synchronization. (He did not do very well on his design doc). In other words, you do not have to worry about synchronization here. You also do not have to worry about implementing rollback features; we can assume allocation always succeeds.



Tacos Disk Layout

i. (2 pt) What is the total size of the disk?

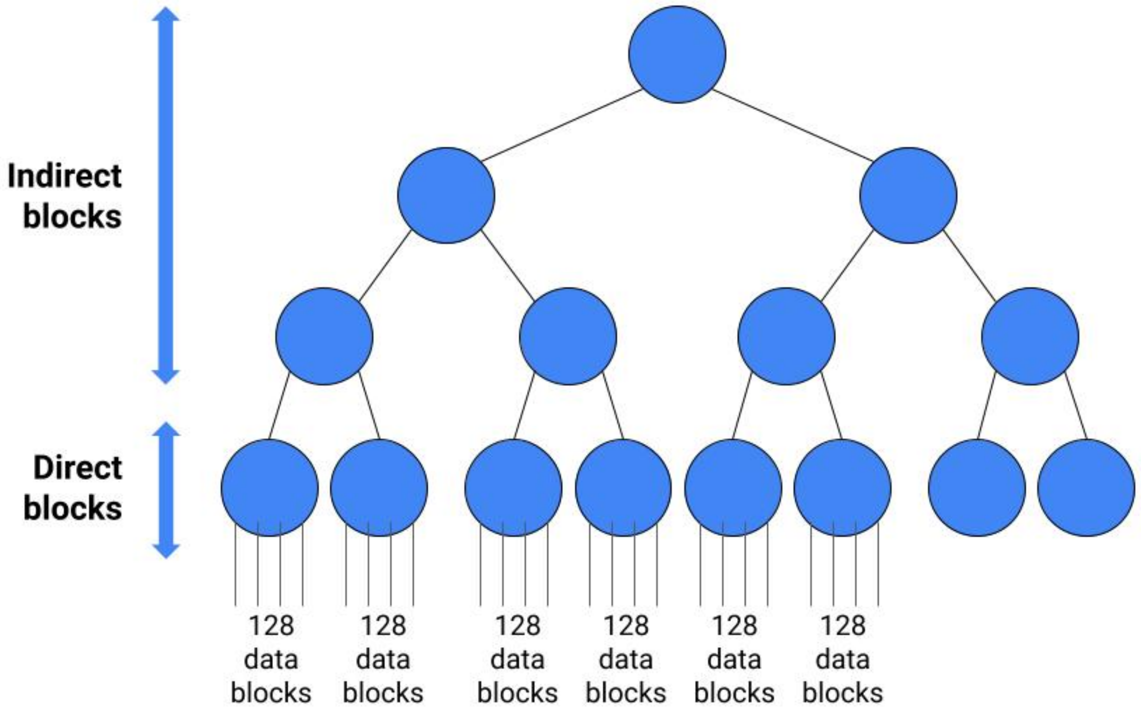
ii. (2 pt) What is the maximum number of files on the system?

iii. (2 pt) How many blocks are required to store the free map?

(b) Extensible Files

Extensible Files Monty Mole has decided to implement extensible files as follows: each file's inode points to a perfect binary tree (note: a perfect binary tree is a binary tree in which every level of the tree is completely filled). The tree consists of indirect blocks (i.e. blocks that simply contain pointers to other nodes in the tree) and direct blocks (i.e. blocks that contain pointers to data blocks). All leaves of the tree must be direct blocks (the data blocks they point to are not considered part of the tree). These inode trees should be able to grow without bound. In addition, the inode tree must always have at least one node.

See the image below for an example of an inode tree.



Tacos Inode

We give the definition of `struct inode_disk` below, which should hold the disk format of inodes in the inode table. Note that defining the length of the file as an `int` implicitly means the max file size is 4 GB. Your code should not be dependent on this limit; i.e. if we change `int length` to `long length`, your code should still work. The length field represents the true length of the file in bytes, not the number of blocks allocated to the file. The depth field represents the depth of the tree, where we consider depth to be the *number of edges* from root to leaf. Again, using an integer for depth implicitly limits depth, but your code should not depend on this limit.

We also give a minimal definition of `struct inode`, the in-memory representation of an inode. You can assume that the `struct` is populated properly when an inode is brought in from disk into memory, and written to disk correctly when an inode is evicted from memory. More fields may need to be added to `struct inode` later in this question.

You also have access to helper functions, `get_length`, `get_depth`, `set_length`, and `set_depth`. Given a `struct inode`, these functions will read in the associated inode from disk and return or update the desired attributes of the inode.

```
typedef uint32_t block_sector_t;

struct inode_disk {
    int length;                    // Length of file
    int depth;                     // Depth of the tree
    block_sector_t tree;         // Pointer to tree
    block_sector_t unused[125];  // Unused space
};

struct inode {
    block_sector_t sector;  // Pointer to inode_disk
};

int get_length(struct inode *inode);
int get_depth(struct inode *inode);
int set_length(struct inode *inode, int new_length);
int set_depth(struct inode *inode, int new_depth);
```

i. (2 pt) Fill in the following definition for `struct indirect_block` (the disk format of the interior nodes of the inode tree) below. `struct direct_block` has been defined for you.

For each blank, you may use as many lines as you need.

```
struct indirect_block {
    ____________________
};

struct direct_block {
    block_sector_t data_blocks[128];
};
```

ii. (3 pt) Monty Mole creates 2^20 - 2 subdirectories, named `s1, s2, ...` and so on, where each subdirectory is nested under the last. For example, the file path to `s5` is `RootDir/s1/s2/s3/s4/s5`. Each of these directories only requires a single data block of storage. In addition, Monty Mole adds a file, 10 data blocks long, to the deepest subdirectory.

How many disk accesses does it take to read the file in its entirety?

iii. (3 pt) After allocating all the directories and files above, Monty Mole has run out of space in the inode table. Because of this, he argues that he can neither add new files nor expand the current file. Is Monty Mole correct? Why?

iv. (8 points)

Finish the implementation of the function `byte_to_sector` below, which finds the block address of the block that contains that byte of the file

- You can use the functions `void block_read(block_sector_t sector, uint32_t* buf)` and `block_write(block_sector_t sector, uint32_t* buf)` which read and write data to a block on disk from a buffer in memory
- Do not worry about the size of the kernel stack when running these functions (**hint**: this means you can use recursion!)
- Please use the `#define`'d constants provided below to help make your code cleaner

For each blank, you may use as many lines as you need.

```
#define BLOCK_SIZE 512;
#define NUM_POINTERS (BLOCK_SIZE / 4);
#define DIR_BLK_CAP NUM_POINTERS * BLOCK_SIZE;

block_sector_t byte_to_sector(struct inode* inode, int byte) {
    if (byte < 0 || byte >= get_length(inode))
        return -1;

    struct inode_disk id;
    block_read(inode->sector, &id);

    // Base Case
    if (________[A]________) {
        struct direct_block;
        block_read(id.tree, &direct_block);
        ________[B]________
    }

    struct indirect_block;
    block_read(id.tree, &indirect_block);
    int num_leaves = (get_length(inode) - 1) / DIR_BLOCK_CAP + 1;
    int max_leaves = 1 << get_depth(inode);
    int max_cap_at_cur_depth = max_leaves * DIR_BLOCK_CAP;

    // Copy the inode, and recurse on an updated version of it
    struct new_inode = *inode;
    ________[C]________

    if (________[D]________) {
        ________[E]________
    } else {
        _________[F]_________
    }

    _________[G]_________
}
```

A. [A]

B. [B]

C. [C]

D. [D]

E. [E]

F. [F]

G. [G]

v. (8 points)

Write a function `void inode_grow_block(struct inode* inode)` to extend the size of the file by a single disk block. We call this function in the provided `inode_resize` function below, which extends an inode and updates its length (note: you are not required to implement shrinking). As in Pintos, `inode_resize` is called when a user tries to write beyond the end of a file.

You should fill in the helper function `alloc_tree` first.

You can assume we provide a function `void inode_grow_block_same_depth(struct inode* inode)` that will perform the function of `inode_grow_block` when the depth of the tree stays the same. The implementation of this function is roughly the same as the implementation of `byte_to_sector`, except it always searches for the byte `inode->length + BLOCK_SIZE` and instead of simply returning the corresponding sector, it allocates one there.

In your implementation, you may use the `block_sector_t block_allocate(void)` function, which allocates a new block on disk and returns its sector. Again, you can assume this function never fails when writing your code

Hint: it may help to update the `inode->length` field in `inode_grow_block` to help you keep track of which pointer to allocate next; the field will be updated to the proper value in `inode_resize` anyway

For each blank, you may use as many lines as you need.

```
void inode_resize(struct inode* inode, int new_size) {
    if (new_size <= inode->length)
        return;

    int curr_num_data_blocks = (inode->length - 1) / BLOCK_SIZE + 1;
    int max_size_cur_data_blocks = BLOCK_SIZE * curr_num_blocks;

    while (max_size_cur_data_blocks < new_size) {
        inode_grow_block(inode);
        max_size_cur_data_blocks += BLOCK_SIZE;
    }

    inode->length = new_size;
}

/* Allocates an on-disk perfect binary tree of depth DEPTH, and
   returns the block_sector_t of the root of the tree. The leaves of the
   tree should remain uninitialized. */
block_sector_t alloc_tree(int depth) {
    if (depth == 0) {
       ________[A]________
    }
    ________[B]________
}

/* Grows the inode by a single block. May require increasing depth */
void inode_grow_block(struct inode* inode) {
    int num_leaves = (inode->length - 1) / DIR_BLOCK_CAP + 1;
    int max_leaves = 1 << get_depth(inode);
    struct inode_disk id;

    if (________[C]________) {
        ________[D]________
    }
    inode_grow_block_same_depth(inode);
    inode->length += BLOCK_SIZE;
}
```

A. [A]

B. [B]

C. [C]

D. [D]

No more questions.