Andrew ID:

Full Name:

Hint: This is an old school handwritten exam. There is no authenticated login. If we can't read your AndrewID, we won't easily know who should get credit for this exam. If we can't read either your AndrewID or Full Name, we're in real bind. Please write neatly :-)

18-213/18-613 Final Exam

Fall 2025

Instructions:

- Make sure that your exam is not missing any sheets (check page numbers at bottom)
- Write your Andrew ID and full name on this page (and we suggest on each and every page)
- This exam is closed book and closed notes.
- You may not use anything other than what we provide, except writing implements, such as pens and pencils, and a simple arithmetic calculator.
- Write your answers in the space provided for the problem.
- If you make a mess, clearly indicate your final answer.
- The exam has a maximum score of 100 points.
- The point value of each problem is indicated.
- **Good luck!**

Problem #	Scope	Max Points	Score
1	Malloc(), Free(), and User-Level Memory Allocation	15	
2	Virtual Memory, Paging, and the TLB	15	
3	Process Representation and Lifecycle	15	
4	Files and File System	15	
5	Signals	10	
6	Concurrency Control, Fundamentals	15	
7	Concurrency Control, Advances	15	
TOTAL	***Total points across all problems***	***100***	

Question : Malloc(), Free(), and User- Memory Allocation (15 points)

1(A) (2.5 points): How does a conventional segregated list allocator continue to search for an available block once it has searched the first list it tries without finding one?

It searches the next larger list. And repeats this process until it has exhausted all lists.

1(B) (2.5 points): How does it handle the case where no appropriate block can be found within any of the seg lists at all?

It grows the size of the heap to create at least one sufficiently large block.

1(C) (2.5 points): When optimizing a seglist allocator for a particular workload, how do you go about determining the size classes, i.e. the size range for each list.

Analyze the benchmark traces for the workload to determine heavily used sizes, set the seglist size classes to accommodate those heavily used allocation sizes, empirically measure the performance of the workload benchmarks, and tune from there. Wash. Rinse. Repeat until best performance is found.

1(D) (2.5 points): Having “too many” segregated lists negatively impacts performance. Please explain if it impacts throughput, utilization, or both, and why.

Throughput. The greater the number of lists, the more time it takes to determine the correct list and the less likely that list is to have any blocks at all, which would force the use of another list.

1(E) (3 points): Operating system kernels often have different memory allocators used for different purposes within the kernel. Consider an allocator designed to allocated and free buffers of specific fixed size that are known in advance. Reason about whether or not these allocators implement constant time coalescing and explain your answer.

In general, they do not. Buffers are placed back into the same list from which they came to be reused again for the same purpose. There is little to be gained by coalescing each time, and time to be lost doing it. Coalescing can be done as needed, for example, if a pool exceeds some high water mark.

Continued on next page.

2. Virtual Memory, Paging, and the TLB (15 points)

This problem concerns the way virtual addresses are translated into physical addresses. Imagine a system has the following parameters:

- Virtual addresses are 16 bits wide.
- Physical addresses are 12 bits wide.
- There are 8 offset bits within each virtual address
- The TLB is 2-way set associative with 4 total entries.
- The TLB may cache invalid entries
- A single level page table is used
- The replacement policy with a set of the TLB is to replace invalid entries before valid entries and to break ties by replacing the lower tag (Regardless of whether or not this is the smartest thing).

Part A: Interpreting addresses (3 points)

2(A)(1)(1 points): Please label the diagram below showing which bit positions are interpreted as each of the PPO and PPN. Leave any unused entries blank.

Bit	11	10	9	8	7	6	5	4	3	2	1	0
PPN/ PPO	*N*	*N*	*N*	*N*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*

2(A)(2)(1 points): Please label the diagram below showing which bit positions are interpreted as each of the VPO and VPN and each of the TLBI (TLB Index) and TLBT (TLB Tag). Leave any unused entries blank.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
VPO/ VPN	*N*	*N*	*N*	*N*	*N*	*N*	*N*	*N*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*
TLBI/ TLBT	*T*	*T*	*T*	*T*	*T*	*T*	*T*	*I*								

2(A)(3) (0.5 points): How many entries exist within each page table? *Hint:* This is the same as the total number of pages within each virtual address space.

256

2(A)(4) (0.5 points): How many sets are in the TLB?

2

Continued on next page.

Part B: Hits and Misses (12 points)

This question asks you to determine the result of each memory access in a memory trace, i.e. a list of sequential memory access. The tables on this page provide the relevant *initial* state of two key structures, the TLB and the Page Table. The table containing the memory trace and in which you mark the result of each access within the trace is provided on the next page.

TLB (X=INvalid, V=VALID, R=READ, W=WRITE, N=Not Resident, e.g. swapped):

Set	Tag	PPN	BITS	Scratch space for you
0	0x79	0xF	V-R	
0	0x12	0x2	V-RW	**Replaced by 0xBA = 0x5D, 0x4, V-RW**
1	0x6B	0xE	V-R	
1	0x64	0x6	VN-RW	

Page Table (X=INvalid V=VALID, R=READ, W=WRITE, N=Not Resident, e.g. swapped):

Index/VPN	PPN	BITS	Scratch space for you
0x24	0x2	V-RW	
0xAD	0x9	V-R	
0xBA	0x4	V-RW	
0xC9	0x6	V-RW	
0xD7	0xE	V-R	
0xF2	0xF	V-R	
0xFC	0x1	V-RW	

Continued on next page.

2. Virtual Memory, Paging, and the TLB (15 points), *cont.*
Part B: Hits and Misses (12 points), *cont.*

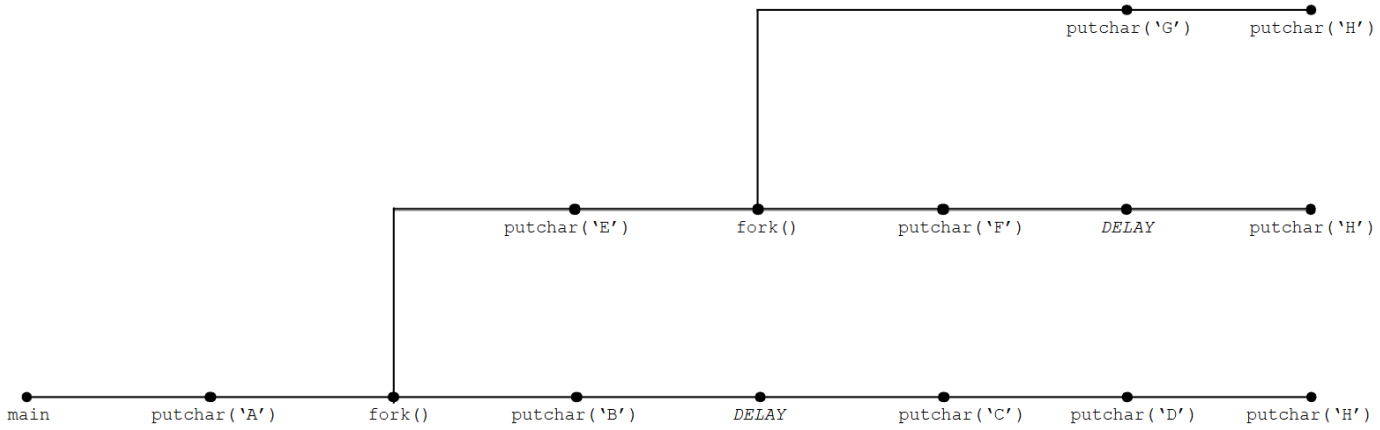
Consider the following memory access trace i.e. sequence of memory operations listed in order of execution, as shown in the first two columns (operation, virtual address). It begins with the TLB and page table in the state shown on the page above.

Please complete the remaining columns by circling the correct answer in the 3rd, 4th, and 5th columns and, if knowable, writing the physical address in hexadecimal into the 6th column.

Operation	Virtual Address	TLB Hit or Miss?	Page Table Hit or Miss?	Page Fault? Yes or No?	Physical Address If Knowable
Read	0xF2A	**Hit** Miss Not knowable	Hit Miss **N/A**	Yes **No** Not knowable	**0xFA**
Write	0xD7C	**Hit** Miss Not knowable	Hit Miss **N/A**	**Yes** No Not knowable	**0xEC**
Write	0xBAA	Hit **Miss** Not knowable	**Hit** Miss N/A	Yes **No** Not knowable	**0x4A**
Read	0xC9D	**Hit** Miss Not knowable	Hit Miss **N/A**	**Yes** No Not knowable	**(Backing Store)**
Write	0xADD	Hit **Miss** Not knowable	**Hit** Miss N/A	**Yes** No Not knowable	**0x9D**
Write	0x24A	Hit **Miss** Not knowable	**Hit** Miss N/A	Yes **No** Not knowable	**0x2A**

Continued on next page.
Question 3: Process Representation and Lifecycle + Signals and Files (10 points)

Please consider the following following process tree:



3(A) (10 points): Please fill in the main() method below such that it implements the specification provided by the tree above with as few calls to putchar() as possible.

```
void main(){
  // Your code here

  putchar('A');
  if (fork())
    putchar('B');
    wait(NULL);
    putchar('C');
    putchar('D');
  } else {
    putchar('E');
    if (fork())
      putchar('F');
      wait(NULL);
    } else {
      putchar('G');
    }
  }
  putchar('H');
}
```

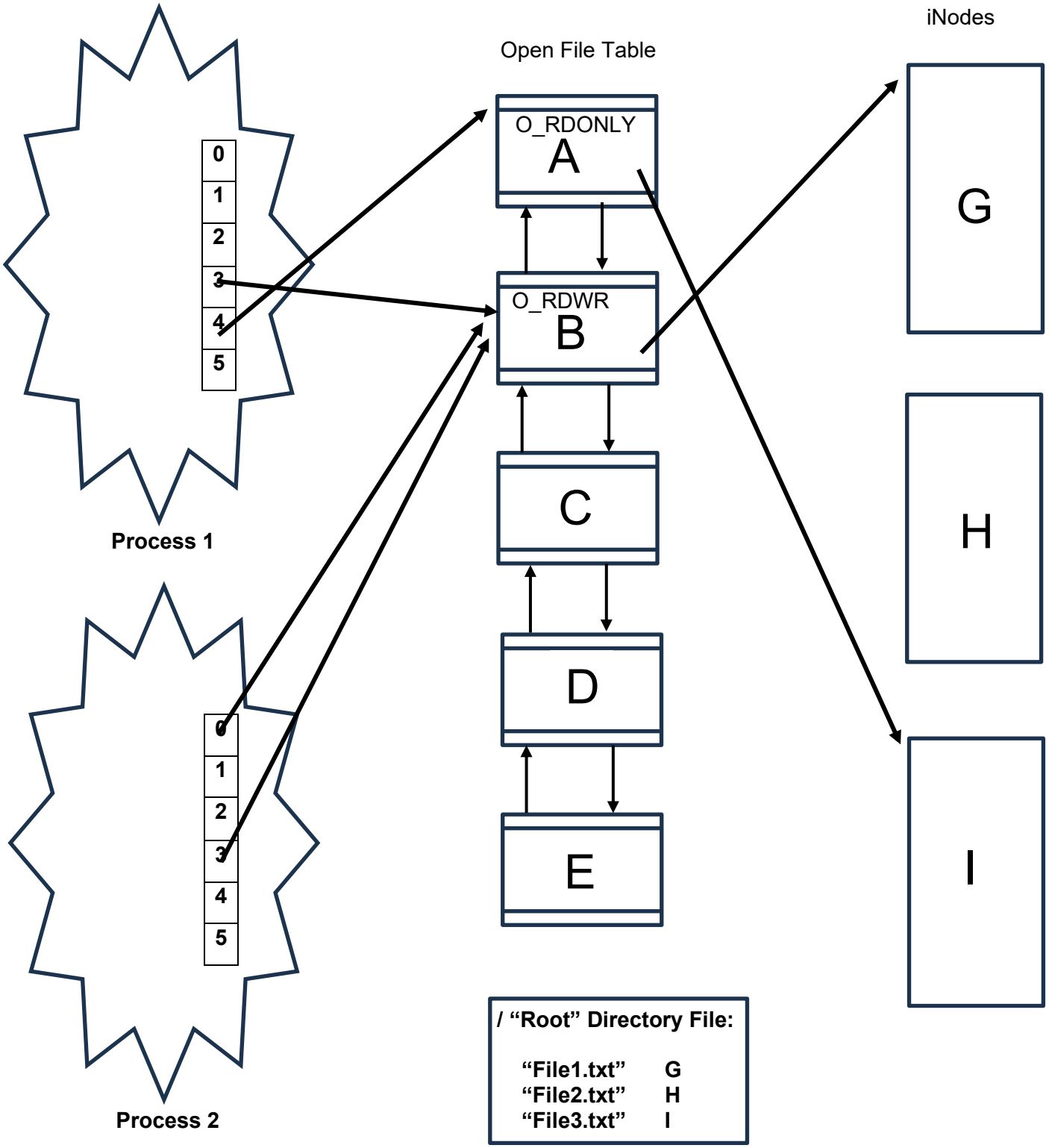
3(B) (5 points): Consider only the last three characters of the output. Please list all possibilities.

CDH

Continued on next page.

Question 4: Files and File System (15 points)

Consider the following representation of file-related state:



Continued on next page.

Question 4: Files and File System (15 points), cont.

4(A) (2 points) What is the reference count at open file table entry “B”? Why?

3. One file descriptor from Process 1 and two file descriptors from Process 2.

4(B) (2 points) What is the reference count at iNode “G”? Why?

1. Only one file session has it in use, the one from OFT Entry B.

4(C) (2 points) If Process 2 reads 100 bytes from it’s fd #3, which other file descriptors would see their associated offset change, if any? Why?

Process #2 FD=0 and Process #1 FD=3. They reference the same OFT entry.

4(D) (2 points) How many file descriptors would need to be closed for the reference count at iNode “G” to decrease by 1? Why?

3. There are three different FDs referencing the only OFT entry referencing iNode G.

Continued on next page.

Question 4: Files and File System (15 points), cont.

4(E) (7 points) Write a main() function, containing not more than one fork() that creates state equivalent to what is shown in the figure for each processes' file descriptor table, open file table entries A and B, and iNodes G and I. The specific file descriptor numbers within each process are arbitrary, you don't need to match exactly those numbers, but you do need to preserve the relationships shown in the diagram.

Each process should end with a "sleep (500);". The state described is observed at a time during which all processes are concurrently with this sleep.

```
void main() {

  if (fork()) {
    // Parent
    int fd1, fd2;

    fd1 = open ("/File1.txt", O_RDWR);
    fd2 = open ("/File3.txt", O_RDONLY);
  } else {
    // Child
    close (fd2);
    dup2(fd1, 0);
  }

  sleep (500);
}
```

Continued on next page.

Question 5: Signals and Signal Management (10 points)

5(A) (4 points): Is signal context a per-process, per-thread, or global state? Why does this make sense?

It would be easy to imagine that signals would be per-process, since we many times treat them as a way of asynchronously signaling an activity about a change in its environment and often might like them to signal a specific thread, e.g. sigsegv. But, processes predate threads and backward compatibility is a thing. Also, many signals related to processes, e.g. SIGCHLD, SIGTSTP, SIGSTOP, SIGCONT, SIGHUP, etc. They could have either been kept per-process, or a new type of signal could have bveen added, but simply making them per-thread wouldn't solve the problem, even if we'd ignore backward compatibility.

As for global? Um. They are really communicating to a specific process about events that process needs to be aware of. There isn't anything global about them.

5(B) (3 points): Imagine that a signal is received while the receiving process blocks it. What happens when the receiving process subsequently unblock the signal? And, why does this make sense?

It will be handled upon the next return from supervisor mode to user mode, as if it was freshly received. It makes sense because ti allows us to defer signals – we can ignore them (do nothing handler, SIGIGN) if we prefer they never be handled at all.

5(C) (3 points) In class we gave example code similar to the following. Please explain a potential problem that might arise in some executions, but not others:

Answer Here ↓

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void signal_handler(int signum) {
    printf("Caught signal %d\n", signum);
}

int main() {
    signal(SIGINT, signal_handler);

    printf("Press Ctrl+C to trigger signal handler\n");

    while(1) {
        sleep(1);
    }

    return 0;
}
```

Should the signal be handled in the middle of main's printf, the printfs could deadlock as they share a libc buffer, which is a critical section and is protected by a mutex. The thread would wait for main to release the mutex, and main would wait for the thread to complete to continue.

Continued on next page.

Question 6: Concurrency Control, Fundamentals (15 points)

Please consider the following code:

```
#include <stdio.h>
#include <pthread.h>

#define NUM_THREADS 4
#define ITERATIONS 100000

int counter = 0;
pthread_mutex_t lock;

void *increment(void *arg) {
    for (int i = 0; i < ITERATIONS; i++) {
        counter++;
    }
    return NULL;
}

int main() {
    pthread_t threads[NUM_THREADS];

    // Initialize mutex
    pthread_mutex_init(&lock, NULL);

    // Create threads
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_create(&threads[i], NULL, increment, NULL);
    }

    // Wait for threads to finish
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    printf("Final counter value: %d\n", counter);
    printf("Expected value: %d\n", NUM_THREADS * ITERATIONS);

    pthread_mutex_destroy(&lock);
    return 0;
}
```

LOCK

UNLOCK

6(A) (2 points) Please circle the critical section of code above

6(B) (3 points) Please annotate the code above the show where you would lock and unlock the critical resource.

Continued on next page.

Question 6 : Concurrency Control, Fundamentals (15 points), *cont.*

6(C) (5 points) A friend suggests augmenting the posix semaphores' interface to provide a function that exposes the count of resources currently available from within a semaphore. Is this a potentially useful addition? Why or why not?

This is useless. Becase of the type of concurrency that semaphores are designed to protect against, the count could change by any amount before this value could be used. A random guess would be as useful.

6(D) (5 points) Can local variables behave as critical resources in threaded code? Why or why not?

Hint: Do not consider aliasing them via global variables: Focus on local variable as local variables.

No. There is one instance per function activation, stored in the activation's stack or register context. Each thread gets its own stack and register context.

7. Concurrency Control, Advanced (15 points)

Please consider the following code that simulates many bank transactions.

```
#define NUM_ACCOUNTS 3
#define NUM_THREADS 5
#define NUM_TRANSFERS 1000

typedef struct {
    int balance;
    pthread_mutex_t lock; // Hint #1: Note this
} Account;

Account accounts[NUM_ACCOUNTS];  // Hint #2: Note this

void transfer(int from, int to, int amount) {
    Account *first = accounts[from];
    Account *second = accounts[to];

   // Hint #3 Maybe stuff goes here?

    if (accounts[from].balance >= amount) {
        accounts[from].balance -= amount;
        usleep(1);  // Simulate processing delay
        accounts[to].balance += amount;
        printf("Thread %ld: Transferred $%d from Account %d to %d\n",
               pthread_self() % 1000, amount, from, to);
    } else {
        printf("Thread %ld: Insufficient funds in Account %d\n",
               pthread_self() % 1000, from);
    }

  // Hint #4: Maybe stuff goes here?
}

void *worker(void *arg) {
    for (int i = 0; i < NUM_TRANSFERS; i++) {
        int from = rand() % NUM_ACCOUNTS;
        int to = rand() % NUM_ACCOUNTS;
        int amount = (rand() % 50) + 1;

        if (from != to) {
            transfer(from, to, amount);
        }
    }
    return NULL;
}
```

```
// Lock in consistent order to avoid deadlock
Account *first = (from < to) ? &accounts[from] : &accounts[to];
Account *second = (from < to) ? &accounts[to] : &accounts[from];

pthread_mutex_lock(&first->lock);
pthread_mutex_lock(&second->lock);
```

```
pthread_mutex_unlock(&second->lock);
pthread_mutex_unlock(&first->lock);
```

Continued on next page.

7. Concurrency Control, Advanced (15 points), *cont.*

```
int main() { // Hint #5 Something goes somewhere in main()

    pthread_t threads[NUM_THREADS];

    // Initialize accounts
    int initial_total = 0;
    for (int i = 0; i < NUM_ACCOUNTS; i++) {
        accounts[i].balance = 1000;
        initial_total += accounts[i].balance;
    }

    printf("Initial total: $%d\n\n", initial_total);

    // Create worker threads
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_create(&threads[i], NULL, worker, NULL);
    }

    // Wait for completion
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }


    return 0;
}
```

```
pthread_mutex_init(&accounts[i].lock, NULL);
```

Please annotate the code above to

- **7(A) (5 points):** Initialize all mutexes,
- **7(B) (5 points):** Protect all critical sections, and
- **7(C) (5 points):** Prevent deadlock. Think carefully here.

  If you can't prevent deadlock your code, we'll give 50% credit if you explain, specifically, how it can occur within your code.

All done! Great work! Enjoy winter break! You've earned it!