INSTRUCTIONS

Please do not open this exam until instructed to do so. Do not discuss exam questions for at least 24 hours after the exam ends, as some students may be taking the exam at a different time.

GENERAL INFORMATION

This is a **closed book** exam. You are allowed 2 pages of notes (both sides). You have 80 minutes to complete as much of the exam as possible.

Write all of your answers directly on this paper. *Make your answers as concise as possible.* If there is something about the questions that you believe is open to interpretation, please ask us about it!

Name

Student ID

Please read the following honor code: "I understand this is a closed-book exam. I promise the answers I give on this exam are my own. I understand that I am allowed to use one 8.5x11, double-sided, handwritten cheat-sheets of my own making, but otherwise promise not to consult other people, physical resources (e.g. textbooks), or the internet in constructing my answers."

Write your full name below to acknowledge that you've read and agreed to this statement.

Problem	Possible
1	12
2	12.5
3	22.5
5	8
4	15
6	10
7	1
Total	81

This page is intentionally left blank, with the exception of this sentence, the header, and two jokes.

A programmer walks to the butcher shop and buys a kilo of meat.

An hour later he comes back upset that the butcher shortchanged him by 24 grams.

1. **(12.0 points) True/False**

   Please explain your answer in **TWO SENTENCES OR FEWER**. Answers without an explanation or that just rephrase the question will **not receive credit**.

   (a) **(2.0 pt)** The highest priority thread can be starved in a strict priority scheduler.

   ● True. Explain/mention one such scenario **as well as** a mechanism that addresses the issue.

   > **Priority inversion. Fix it with priority donation.**

   ○ False. Explain why the highest priority thread cannot be starved.

   (b) **(2.0 pt)** SJF is at least as good as Round Robin (RR) in terms of minimizing average completion time.

   ○ True. Briefly explain why this is the case.

   ● False. Give an example where Round Robin would have a smaller average completion time than SRTF.

   > **SJF is not preemptive, so consider a case where there is one extremely long task present in the system at the beginning. SJF will suffer from convoy effect while RR completes quickly.**

   (c) **(2.0 pt)** Lower quantas for round robin generally decrease throughput.

   ● True. Explain why this is the case.

   > **Lower quantas mean more context switching $\rightarrow$ more overhead**

   ○ False. Explain why lower quantas for round robin increase throughput.

(d) **(2.0 pt)** Banker's algorithm allows a system to recover from deadlock once it occurs.

○ True. Explain how Banker's algorithm implements deadlock recovery.

⬤ False. Explain the purpose of Banker's algorithm and describe how it works.

> **Banker's is an algorithm for deadlock avoidance that evaluates each request and grants it if some ordering of threads is still deadlock free afterward**

(e) **(2.0 pt)** Multi-level paging works better under the assumption that address spaces are sparse.

⬤ True. True. Explain the main benefit of having a multi-level paging scheme with a sparse address space.

> **With multi level page tables only the pages needed for virtual pages are allocated -> less memory than single level which allocates space for all possible virtual pages**

○ False. Explain why **single-level** paging is better for sparse address spaces.

(f) **(2.0 pt)** With demand paging, fork may lazily copy memory from the parent.

⬤ True. Explain when the OS will choose to allocate separate memory for the parent and the child.

> **Demand paging will simply mark the pages as read-only and allocate copies when one attempts to write to it.**

○ False. Explain how the child process copies memory from the parent.

2. **(12.5 points)   Multiple Select**

   (a) **(2.5 pt)** Select all true statements.

      ■ Users can manipulate MLFQ schedulers by strategically inserting I/O requests.

      Note: Credit was awarded for both True or False.

      ■ MLFQ reduces the priority of a process that fails to use up a given time slice.

      Note: Credit was awarded for both True or False.

      ■ Lottery scheduling suffers from a lack of predictability in performance.

      ☐ Stride scheduling gives priority to threads by giving them larger strides.

      ■ EEVDF is the default scheduling policy in the latest versions of Linux.

   (b) **(2.5 pt)** Select all of the following statements that are true regarding EEVDF.

      ■ The system virtual time represents the index of the current round in a weighted bit-by-bit round robin.

      ■ The rate of virtual time slows down when additional threads enter the system.

      ■ EEVDF schedules threads based on when they would start and finish in a fluid flow system.

      ☐ When a new thread enters the system, all existing threads' virtual eligible times and deadlines are immediately recalculated.

      ■ A thread can force an earlier virtual deadline by requesting a shorter time slice.

   (c) **(2.5 pt)** Which of these schedulers/scheduling policies are prone to starvation?

      ☐ Lottery

      ☐ Round robin

      ☐ EEVDF

      ■ SJF

      ■ STCF

   (d) **(2.5 pt)** Select all statements that **must** be true if there is a deadlock.

      ■ There exists circular wait amongst some set of threads.

      ■ At least one thread is holding some resource and waiting to acquire another.

      ☐ Threads cannot be preempted, i.e. they cannot be taken off the CPU once scheduled.

      ■ There exists some set of threads that are being starved.

      ☐ There exists at least one mutex (mutual exclusion) lock in the system.

   (e) **(2.5 pt)** Select all true statements about virtual memory.

      ■ Base-and-bound is prone to external fragmentation.

      ■ With segmentation, virtual addresses can include segment identifiers.

      ■ The page table base pointer is stored in a special register within the CPU.

      ■ A page table entry can point to a data page or another page table.

      ■ TLBs can reduce the number of main memory accesses.

3. **(22.5 points) Short Answer**

(a) **(10.0 pt)** Suppose the following threads arrive in the ready queue at the clock ticks shown below. Assume threads **arrive** in the queue at the **start** of the time slices below (i.e. you may update the ready queue with the newly arrived thread before you execute that clock tick). **Higher numerical values** correspond to **higher priority**. *Note*: if a thread is executed 1 time tick after its arrival, it has waited for 1 time tick.

Thread	Arrival Time	Execution Time	Priority
A	0	3 ticks	1
B	1	3 ticks	3
C	3	1 ticks	4

Fill in the table to indicate which thread would be scheduled at each tick.

Time →	0	1	2	3	4	5	6
STCF	A	A	A	C	B	B	B
FCFS	A	A	A	B	B	B	C
Priority	A	B	B	C	B	A	A

Fill out the table to complete the average waiting time for each policy. Recall that the waiting time is the total time it spends waiting for the CPU.

Scheduling Policy	Average Waiting Time (ms)
STCF	$(0 + 3 + 0)/3 = 1$
FCFS	$(0 + 2 + 3) / 3 = 5/3$
Priority	$(4 + 1 + 0) / 3 = 5/3$

(b) **(6.5 pt)** EEVDF

i. **(5.0 pt)** Assume we now wish to run EEVDF. Thread F has request length `R=1 tick` and weight `w=1`. Thread G has request length `R=1 ticks` and weight `w=4`. Fill out the following table to determine the scheduling decisions using EEVDF.

A. Assume each thread **continuously submits requests**. v_e represents the virtual eligible time of that thread, and v_d represents its virtual deadline.

B. You may also assume that we allocate CPU time for each thread in increments of a fixed time quanta of 1 tick. Assume each thread takes up exactly its requested service time for each request.

Real time (ticks)	Virtual time	F		G		Scheduled Thread
t	**V(t)**	v_e	v_d	v_e	v_d	— —
0	0	0	1	0	0.25	G
1	0.2	0	1	0.25	0.5	F
2	0.4	1	2	0.25	0.5	G
3	0.6	1	2	0.5	0.75	G

ii. **(1.5 pt)** Assume another Thread F with weight `w=1` enters at time t=2. What virtual time corresponds to real time t=5? Assume Threads D, E, and F do not leave the system. **Show your work.**

$$V(t) = 0.4 + 1/6 * (5\text{-}2) = 0.4 + 0.5 = 0.9$$

(c) **(3.0 pt)** Answer the questions below regarding the Pintos scheduler and the run queue.

    i. **(0.5 pt)** What is the appropriate mechanism for synchronizing concurrent access to the global scheduler queue? (5 words or fewer)

> Disable interrupts

    ii. **(2.5 pt)** Your answer to the previous question should not have been "locking". Explain why you cannot use a lock to synchronize access to the global scheduler queue. (3 sentences or fewer.)

    Note: Recall that Pintos is a uniprocessor OS.

> Say you service an interrupt in the middle of modifying a doubly-linked queue. And after you're done you want to schedule a new thread, but the run queue is potentially in an inconsistent state.

(d) **(3.0 pt)** Assume that our system uses a 3-level page table for address translation, in addition to a TLB.

    TLB Hit Rate: 80%
    TLB Lookup Time: 10ns
    Memory Access Time: 30ns

Assume each page table is stored in memory. Calculate the **Average Memory Access Time (AMAT)** for this system.

> 0.8 * (10ns + 30ns) + 0.2 * (10ns + 30ns*4) = 40*0.8 + 0.2*(130)

4. **(8.0 points)   Is this an omen for Project 3?**

Let's implement dynamic stack growth in Pintos, where you add a new stack page on demand when a program needs more stack memory. We want to support a maximum of 1 MiB (0x100000 bytes) for the stack.

```c
static void GROWTH_FUNCTION(struct intr_frame* f) {
  bool not_present = (f->error_code & PF_P) == 0;
  bool user = (f->error_code & PF_U) != 0; // True: access by user, false: access by kernel.
  void* fault_addr;
  asm("movl %%cr2, %0" : "=r"(fault_addr)); // Obtain faulting address.

  uint32_t* pagedir = thread_current()->pcb->pagedir; // Not needed, but use if necessary.

  if (not_present && user && fault_addr >= PHYS_BASE - 0x100000 && fault_addr < PHYS_BASE) {

    bool success = true;

    void* curr_page = fault_addr & ~(PGSIZE - 1); // Rounds down to the nearest PGSIZE.
    void* kpage;

    kpage = palloc_get_page(PAL_USER|PAL_ZERO);


    success = kpage != NULL;

    success = success && install_page(curr_page, kpage, true);
      // Hint 1: C lazily evaluates && expressions.


    if (success) {
      return;
    }
    else if (kpage) {

      palloc_free_page(kpage);
    }
  }
  kill(f);
}
```

(a) **(2.0 pt)** Examine the code for the body of `GROWTH_FUNCTION`. Identify what this function is with a description of 5 words or fewer. (Hint: It is not a stack growth function.)

> **This function is the page fault handler.**

(b) **(6.0 pt)** Complete the implementation of the code block above. Relevant PintOS functions may be found on your reference sheet.

5. **(15.0 points)   The Return of the Dining Philosophers**

In the Dining Philosophers problem, N philosophers sit around a table, identified by Philosopher 1 through N, in a clockwise manner. There are a total of N chopsticks on the table, each of which is in between two philosophers.

Ana wants to code a `PintOS` program that allows the philosophers to dine together. Each philosopher thread is identified by an integer `i` and runs the function `philosopher(i)`. Assume `eat` runs for a finite amount of time.

```
/* Lines 1 and 2 are called before any thread calls the philosopher function.
   Note that each semaphore is initialized to 1. */

1.  struct semaphore chopsticks[N];
2.  for (int i = 0; i < N; i++) sema_init(&chopsticks[i], 1);

3.  void philosopher(int i) {
4.
5.    sema_down(&chopsticks[i]); // pick up left chopstick
6.    sema_down(&chopsticks[(i+1) % N]); // pick up right chopstick
7.
8.    eat();

9.    sema_up(&chopsticks[i]);
10.   sema_up(&chopsticks[(i+1) % N]);
11. }
```

(a) **(2.0 pt)** Select which of the following statements are true regarding Ana's code above.

☐ If all `N` threads run concurrently, Ana's code **always** deadlocks.

■ If the `Nth` philosopher ran **line 6 before line 5**, deadlock would **not** occur.

■ If each philosopher only runs line 6 after their right neighbor runs line 5, deadlock occurs.

■ If `N-1` threads run concurrently with `N` chopsticks, deadlock will not occur.

(b) **(4.0 pt)** Tarun has another idea to solve the Dining Philosophers problem. He proposes that we wrap lines 5 and 6 from **part (a)** in a critical section by using a global lock `mutex`. Assume the lock is initialized. (*Hint*: each philosopher may only acquire the chopsticks to their immediate left and right).

```
4.    lock_acquire(&mutex);
5.    sema_down(&chopsticks[i]);
6.    sema_down(&chopsticks[(i+1) % N]);
7.    lock_release(&mutex);
```

i. **(2.0 pt)** Does Tarun's code still deadlock? Why or why not? **Explain in 1-2 sentences.**

> **No, this removes the circular waiting because only one philosopher can pick up chopsticks at once.**

ii. **(2.0 pt)** Suppose that only one philosopher, Socrates, eats indefinitely. **Assume N > 2.** Select all true statements given this information, regarding **Tarun's solution.**

■ N-1 philosophers may starve because Socrates's **neighbor** may hold the `mutex` indefinitely.

☐ N-1 philosophers starve because **Socrates** will hold the `mutex` indefinitely.

■ At most one philosopher can try to **acquire** chopsticks at any given time.

☐ No philosophers starve.

(c) **(9.0 pt)** Diana wants to fix this issue, so she poses the Dining Philosophers problem for `N` philosophers with the following constraints. **Please read all constraints carefully.**

  i. In this problem, we represent the state of each **philosopher** instead of each **chopstick**. Each `philosopher_state` variable should be properly synchronized across many philosopher threads.

  ii. A philosopher requires **both** their left and right chopstick to eat, and should release both chopsticks immediately after eating.

  iii. A philosopher should **sleep** while they are waiting to eat, and **wake up** once they are allowed to eat.

  Hint: A *sleeping* philosopher cannot wake themselves.

```
/* Helper functions to find the left or right neighbor of philosopher i. */
#define left(i) (i-1+N) % N
#define right(i) (i+1) % N

/* Each philosopher is THINKING by default. A philosopher is HUNGRY if they're waiting to eat */
#define THINKING 0
#define HUNGRY 1
#define EATING 2

struct lock mutex;

struct semaphore philosopher[N];
int philosopher_state[N];

void eat(i) { /* Implementation not shown. */ }

/* Main function of this Dining Philosophers Problem. */
void main(int N) {
    lock_init(&mutex);

    for (int i = 0; i < N; i++) {
        sema_init(&philosopher[i], 0);
        philosopher_state[i] = THINKING;
    }

    /* Spawn all N philosopher threads and waits for them to run to completion.
       Implementation not shown. */
}

// Main function for each philosopher thread.
void philosopher(int i) {
    start_eating(i);
    eat(i);
    finish_eating(i);
}
```

Fill out the coding implementation on the following page.

(c) **(9.0 pt) Fill out the following coding implementation in the lines below.**
Relevant PintOS functions may be found on your reference sheet.

```
// Philosopher i calls this before eating.
void start_eating(int i) {

    lock_acquire(&mutex);
    philosopher_state[i] = HUNGRY;
    test_and_wake(i);

    lock_release(&mutex);

    sema_down(&philosopher[i]);

}


// Wake philosopher j if they are allowed to eat.
void test_and_wake(j) {
    if (philosopher_state[j] == HUNGRY

    && philosopher_state[left(j)] != EATING

    && philosopher_state[right(j)] != EATING {
        philosopher_state[j] = EATING;
        sema_up(&philosopher[j]);
    }
}

// Philosopher i calls this after eating.
void finish_eating(i) {

    lock_acquire(&mutex);
    philosopher_state[i] = THINKING;

    test_and_wake(left(i));

    test_and_wake(right(i));

    lock_release(&mutex);
}
```

6. (10.0 points)   One Page at a Time

(a) **(4.0 pt)** Assume we're operating in a 64-bit system with 64 GiB of physical memory. Pages are 4KiB in size.

   i. **(1.0 pt)** What is the maximum number of pages that we can store in physical memory?

   > $2^{36-12} = 2^{24}$

   ii. **(1.0 pt)** What are the number of offset bits needed to support the system described above?

   > 12

   iii. **(2.0 pt)** Assume we want to fit each page table into a single page, and that the upper 16 bits of each virtual address are unused. Assume each PTE is 8 bytes in size.

   What are the number of page table levels required to support each virtual address given these requirements?

   > $2^{12}/2^3 = 2^9$ PTEs per page table. 64 bit virtual address - (16 bits unused + 12 bits offset) = 36 bits for VPNs. 36 / 9 = 4 VPNs.

(b) **(6.0 pt) Note that this subpart is independent of part (a).** Now, assume we are on a 32-bit system with 4 KiB pages and the following virtual address format.

VPN 1 (10 bits)	VPN 2 (10 bits)	Offset (12 bits)

Each page table entry is structured as follows:

PPN (20 bits)	Miscellaneous (9 bits)	User (1 bit)	Writable (1 bit)	Valid (1 bit)

Fill out the following table of memory **addresses** to perform a page table walk. Note that all necessary contents of memory are **provided.** If we notice an access violation, do **NOT** translate the address; instead write access failures (either **"KERNEL_ONLY"**, **"NOT_WRITABLE"**, or **"INVALID"**). If there is no violation, write the byte loaded or **"OK"** if the store was successful.

Hint: `0x2B4 * 0x4 = 0xAD0, 0x255 * 0x4 = 0xFFC`

The Page Table Base Register (CR3) holds the address 0xCF3D 2000.

	Virtual Address	PTE 1 Address	PTE 1	PTE 2 Address	PTE 2	Physical Address (or Violation)
Load	0x0060 2104	0xCF3D 2004	0xDEAD B025	0xDEAD B808	0x1235 A025	0x1235 A104
Store	0xAD00 6434	0xCF3D 2AD0	0xDEAD BEE7	0xDEAD B018	0x8BAD 5025	NOT_WRITABLE

7. (1.0 pt) Estimate each staff member's height (including all professors, TAs, and tutors).

Name	Height
Aarin	6'
Ashwin	6'
Diana	6'
Jacob	6'
Jason	6'
Jeremy	6'
Luca	6'
Matei	6'
Natacha	6'
Richard	6'
Rohan	6'
Sean	6'
Shamith	6'
Sriram	6'
Tushar	6'

8. (0.0 pt) If you found any questions ambiguous, please address your interpretation of the question with its question number here. We will take a look at this during regrades and may award points if we consider it valid.

.