

Midterm II

March 31st, 2026

CS162: Operating Systems and Systems Programming

Your Name:	
SID:	
CS162 Login:	
TA Name/Section Time	
Person to Left, Right, Front, and Back of you:	

This is a **closed book** exam. You are allowed 1 page of notes (both sides). You have 170 minutes to complete as much of the exam as possible. Make sure to read all of the questions first, as some of the questions are substantially more time consuming.

Write all of your answers directly on this paper. *Make your answers as concise as possible.* On programming questions, we will be looking for performance as well as correctness, so think through your answers carefully. If there is something about the questions that you believe is open to interpretation, please ask us about it!

Problem	Possible	Score
1	20	
2	16	
3	18	
4	18	
5	16	
6	12	
Total	100	

[This page left for π]

3.14159265358979323846264338327950288419716939937510582097494459230781640628620899

Problem 1: True/False [20 pts]

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

Problem 1a[2pts]: In general, on systems that support inverted page tables, the size of the table is proportional to the amount of DRAM in the system.

True False

Explain:

Problem 1b[2pts]: If the Banker's algorithm finds that it's safe to allocate a resource to an existing thread, then all threads will eventually complete.

True False

Explain:

Problem 1c[2pts]: Even though most processors use a *physical address* to address the cache, it is possible to overlap the TLB lookup and cache access in hardware.

True False

Explain:

Problem 1d[2pts]: The lottery scheduler prevents CPU starvation by assigning at least one ticket to each scheduled thread.

True False

Explain:

Problem 1e[2pts]: For mainstream operating systems such as Linux, the x86 segmentation hardware is configured for user processes so that it will trigger a fault when a thread accesses an address that is outside the limit of the stack segment (SS). It is this fault that is used to automatically extend the stack when more space is needed.

True False

Explain:

Problem 1f[2pts]: The Shortest Remaining Time First (SRTF) algorithm is the best preemptive scheduling algorithm that can be implemented in an Operating System to optimize for average completion time.

True False

Explain:

Problem 1g[2pts]: The Rate Monotonic Scheduler (RMS) takes deadlines into account at runtime to help provide real-time guarantees.

True False

Explain:

Problem 1h[2pts]: In Linux CFS, sleeping threads do not accumulate vruntime, and as a result will be lower prioritized when they wake up.

True False

Explain:

Problem 1i[2pts]: Assume that, in a system, there are 10 jobs, each of which will run for 100 ticks total (and never voluntarily yield the CPU or block). If our system uses a round-robin (RR) scheduler, then increasing the quantum from 10 ticks to 20 ticks will always result in higher throughput.

True False

Explain:

Problem 1j[2pts]: It is possible for two *processes* to share information by reading and writing from a shared linked list.

True False

Explain:

Problem 2: Multiple Choice [16pts]

Problem 2a[2pts]: Which of the following is true about a multi-level feedback queue (MLFQ)? (choose all that apply):

- A: # Starvation is never a problem for a MLFQ.
 B: # Short-running tasks are given higher priority.
 C: # A MLFQ is an approximation of shortest remaining time first.
 D: # A MLFQ is less fair than round robin, where “fair” is with respect to CPU time.
 E: # One way to keep a job’s priority high in a MLFQ is to insert a bunch of short sleep operations.

Problem 2b[2pts]: Consider a system with a priority-based scheduler that provides priority donation. In this problem, a job with a *higher* priority number has *higher* priority. Suppose that the following events occur in this order:

1. Thread A is created with priority 1 and calls acquire on locks lock_1 and lock_2.
2. Thread B is created with priority 2 and calls acquire on lock_1.
3. Thread C is created with priority 3 and calls acquire on lock_2.
4. Thread A releases lock_1.
5. Thread A releases lock_2.

Which of the following sequences represent the possible evolution of Thread A’s effective priority? (choose one):

- A: 1, 1, 1, 1, 1
 B: # 1, 2, 3, 2, 1
 C: # 1, 1, 3, 2, 1
 D: # 1, 2, 5, 3, 1
 E: # 1, 2, 3, 3, 1

Problem 2c[2pts]: A processor which provides “Precise Exceptions” is one in which (choose one):

- A: Instructions after the one that caused the exception are allowed to progress while the exception is being handled, but they save a precise log of changes so that the operating system can merge updated instruction state with the post-exception state.
 B: Important exceptions are *synchronous* (always occurring at the same place in the instruction stream) as opposed to *asynchronous*. This helps during restart after the exception is handled.
 C: At the time that an exception handler begins execution, there is a well-defined instruction address in the instruction stream for which all prior instructions have committed their results and no following instructions (including the excepting one) have modified processor state.
 D: Allows the Operating System to disable out-of-order execution during critical sections of the user’s program, thereby preventing instructions following an exception from being partially executed.
 E: There is never any ambiguity as to *which* type of exception occurred in the user’s program.

Problem 2d[2pts]: Which of the following are potential ways to avoid cache misses? (choose all that apply):

- A: Increase cache size.
- B: # Increase cache associativity.
- C: # # Decrease cache associativity.
- D: # # Prefetching.
- E: # # Rearrange the layout of data structures.

Problem 2e[2pts]: Earliest Deadline First (EDF) scheduling has an important advantage over other scheduling schemes discussed in class because (choose all that apply):

- A: # # The EDF scheduler can be used to provide guaranteed fractions of the CPU to non-real-time tasks while still meeting deadlines for hard real-time tasks.
- B: # # It can allocate 100% of processor cycles to real-time periodic tasks while still providing a guarantee of meeting real-time deadlines.
- C: # # It can operate non-preemptively and is thus simpler than many other scheduling schemes.
- D: # # When combined with out-of-order execution, EDF can guarantee that deadlines are met even if the processor pipeline is overcommitted (more than 100% utilized), unlike the RMS scheduler.
- E: # # None of the above

Problem 2f[2pts]: Select all true statements about threads in Pintos (choose all that apply):

- A: An exited thread will free its contents only after a context switch from that thread
- B: When a kernel thread stack overflows, it will allocate more space to dynamically fit extra data.
- C: Semaphores contain an internal queue of threads waiting on the semaphore
- D: A user thread created with the `pthread_create` syscall will share the same kernel thread as its process's main thread
- E: None of the above

Problem 2g[2pts]: Suppose that during a page walk, two different virtual addresses in two different page directories resolve to the *same* physical frame. Which of the following could potentially cause this? (Choose all that apply.)

- A: `fork()` using copy on write
- B: `mmap()` system call
- C: `pipe()` system call
- D: `sbrk()` system call
- E: context switches
- F: none of the above

Problem 2h[2pts]: Consider a computer system with the following parameters:

Variable	Measurement	Value
P_{TLB}	Probability of TLB miss	App Dependent
P_{PF}	Probability of a page fault when a TLB miss occurs on user pages (assume page faults do not occur on page tables).	App Dependent
P_{L1}	Probability of a first-level cache miss for all accesses	App Dependent
P_{L2}	Probability of a second-level cache miss for all accesses	App Dependent
T_{TLB}	Time to access TLB (hit)	1 ns
T_{L1}	Time to access L1 cache (hit)	5ns
T_{L2}	Time to access L2 cache (hit)	20ns
T_M	Time to access DRAM	100ns
T_D	Time to transfer a page to/from disk	10 ms = 10,000,000 ns

Assume a physically indexed, physically tagged two-level cache with a TLB that *does not operate in parallel with cache lookup*. Further, assume that the TLB is refilled automatically by the hardware on a miss. Assume that *pages which hit in the TLB are definitely in memory*, but that pages which miss in the TLB may be either in memory or on disk.

The 2-level page tables are kept in physical memory and are cached like other accesses. Assume that the costs of the page replacement algorithm and updates to the page table are included in the T_D measurement. Also assume that no dirty pages are replaced and that pages retrieved from disk on a page fault are not cached.

Finally, assume that the full working set of the application is not able to fit into DRAM, thus $P_{PF} > 0$.

Which of the following expressions represents the AMAT for our application? (*Chose only one*):

A: $\bigcirc T_{L1} + P_{L1} \times (T_{L2} + P_{L2} \times T_M) + P_{TLB} \times (T_{TLB} + 2T_M + P_{PF} \times T_D)$

B: $\bigcirc T_{TLB} + [T_{L1} + P_{L1} \times (T_{L2} + P_{L2} \times T_M)] + P_{TLB} \times \{2[T_{L1} + P_{L1} \times (T_{L2} + P_{L2} \times T_M)] + P_{PF} \times T_D\}$

C: $\bigcirc T_{L1} + P_{L1} \times (T_{L2} + P_{L2} \times T_M) + P_{TLB} \times \{T_{TLB} + 2[T_{L1} + P_{L1} \times (T_{L2} + P_{L2} \times T_M)]\} + P_{PF} \times T_D$

D: $\bigcirc T_{TLB} + T_{L1} + T_{L2} + T_M + T_D$

E: $\bigcirc T_{TLB} + (1 - P_{TLB} \times P_{PF}) \times [T_{L1} + P_{L1} \times (T_{L2} + P_{L2} \times T_M)] + P_{TLB} \times \{2[T_{L1} + P_{L1} \times (T_{L2} + P_{L2} \times T_M)]\} + (P_{TLB} \times P_{PF}) \times (T_{L1} + T_{L2} + T_M + T_D)$

[This Page Intentionally Left Blank]

Potpourri 3: Short Answer [18pts]

Problem 3a[3pts]: Suppose you are debugging the starting implementation of Project 0. Consider the following output from a GDB debugging session:

```

96 | /* Initialize interrupt frame and load executable. */
97 | if (success) {
98 |     memset(&if_, 0, sizeof if_);
99 |     if_.gs = if_.fs = if_.es = if_.ds = if_.ss = SEL_UDSEG;
100 |     if_.cs = SEL_UCSEG;
101 |     if_.eflags = FLAG_IF | FLAG_MBS;
102 |     success = load(file_name, &if_.eip, &if_.esp);
103 | }
104 |
105 | /* Handle failure with succesful PCB malloc. Must free the PCB */
106 | → if (!success && pcb_success) {
-----
(gdb) print/x if_
$1 = {edi = 0x0, esi = 0x0, ebp = 0x0, esp_dummy = 0x0, ebx = 0x0, edx = 0x0,
ecx = 0x0, eax = 0x0, gs = 0x23, fs = 0x23, es = 0x23, ds = 0x23, vec_no = 0x0,
error_code = 0x0, frame_pointer = 0x0, eip = 0x804890f, cs = 0x1b,
eflags = 0x202, esp = 0xc0000000, ss = 0x23}

```

Based on this output, what is the address of the `_start` function? Additionally, at what address will `_start` assume `argc` and `argv` are at? *Show some sort of work to justify your answer.*

`_start:` _____

`argc:` _____

`argv:` _____

Problem 3b[2pts]: Continuing from (3a) suppose you typed “continue” and get the following:

```

1 | Dump of assembler code for function intr0e_stub:
2 | → 0xc00224d2 <+0>:    push   (%esp)
3 |    0xc00224d5 <+3>:    mov    %ebp,0x4(%esp)
4 |    0xc00224d9 <+7>:    push   $0xe
5 |    0xc00224db <+9>:    jmp   0xc002242e <intr_entry>
6 | End of assembler dump.

```

What happened for this to show up? And where in the code did this stop?

Problem 3c[2pts]: Continuing further from (3b) on previous page, suppose that you now type “backtrace” and get the following result (you will tell us what the mystery commands are below):

```
(gdb) backtrace
#0  0xc00224d2 in intr0e_stub ()
#1  0x00000005 in ?? ()
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
(gdb) <mystery command A>
#0  0x08048915 in ?? ()
#1  0xf000ff53 in ?? ()
(gdb) <mystery command B>
add symbol table from file "tests/userprog/do-nothing" at
      .text_addr = 0x8048094
(gdb) <mystery command A>
#0  _start (argc=-268370093, argv=0xf000ff53) at ../../lib/user/entry.c:6
#1  0xf000ff53 in ?? ()
```

What are mystery commands A and B? Fill in the blanks:

Command A: (gdb) _____

Command B: (gdb) _____

Problem 3d[2pts]: Compute AMAT for a cache with a 90% hit rate, 10ns access time, and 100ns DRAM access time. *Show your work in distill your result to a single number.*

Problem 3e[2pts]: In our simple cache model, when we have a cache miss, we pay the DRAM fetch penalty and then move it into the cache, stalling the processor pipeline while we fetch from the DRAM. We then assume that any future accesses to that element will be cache hits. In reality, modern processors (with out-of-order execution) can make progress during a cache miss. In that case, what happens if we have a first cache miss, and during the 100ns delay to bring in the data from DRAM we have a second cache miss *to the same cache line*? This is called a *delayed hit*. We don't need to pay the full 100ns miss time, but we also don't get to count a 10ns hit time. Let's calculate the AMAT if 20% of all accesses are *delayed hits* and we now only have a 70% hit rate. Assume that *delayed hits* are uniformly distributed over the DRAM fetch window. *Show your work in distill your result to a single number.*

For the next couple of questions (Problems 3f-3h), we are going to examine the following student implementation for Homework 1 (as a reminder, you needed to implement word count with a Pintos list). Note that this code is known to be buggy!

```
1: typedef struct word_count {
2:     char* word;
3:     int count;
4:     struct list_elem elem;
5: } word_count_t;

6: word_count_t* find_word(struct list* wclist, char* word) {
7:     struct list_elem* e;
8:     for (e = list_head(wclist); e != list_tail(wclist); e = list_next(e))
9:         word_count_t* wc = list_entry(e, word_count_t, elem);
10:    if (wc->word == word)
11:        return wc;
12:    }
13:    return NULL;
14: }

15: word_count_t* add_word(struct list* wclist, char* word) {
16:    word_count_t* existing = find_word(wclist, word);
17:    if (existing != NULL) {
18:        existing->count++;
19:        return existing;
20:    }
21:    word_count_t* wc = malloc(sizeof(word_count_t));
22:    if (wc == NULL) return NULL;
23:    wc->word = word;
24:    wc->count = 1;
25:    list_push_back(wclist, &wc->elem);
26:    return wc;
27: }

28: static bool less_list(const struct list_elem* ewc1,
29:                       const struct list_elem* ewc2, void* aux) {
30:    bool (*less)(const word_count_t*, const word_count_t*) = aux;
31:    return less(ewc1, ewc2);
32: }

33: void wordcount_sort(struct list* wclist,
34:                    bool less(const word_count_t*, const word_count_t*)) {
35:    list_sort(wclist, less_list, less);
36: }

37: int main(int argc, char* argv[]) {
38:    struct list word_counts;
39:    char buf[256];
40:    FILE* infile = argc > 1 ? fopen(argv[1], "r") : stdin;
41:    while (fscanf(infile, "%255s", buf) == 1) {
42:        for (int i = 0; buf[i]; i++)
43:            buf[i] = tolower(buf[i]);
44:        add_word(&word_counts, buf);
45:    }
46:    if (infile != stdin) fclose(infile);
47:    wordcount_sort(&word_counts, less_count);
48:    return 0;
49: }
```

Using the code from the previous page, answer the following questions:

Problem 3f[3pts]: What implementation errors are there in `find_word()`? (There are 3 problems, one of them minor).

Problem 1: _____

Problem 2: _____

Problem 3: _____

Problem 3g[2pts]: What is the problem with `add_word()` and how can it be fixed?

Problem: _____

Solution: _____

Problem 3h[2pts]: The function `less_list()` is causing the code to not compile due to passing `list_elem` into `aux`, which is a `word_count_t` comparator. Please rewrite it here (you do not need to use all of the provided lines):

```
static bool less_list(const struct list_elem* ewc1,  
                    const struct list_elem* ewc2, void* aux) {
```

```
}
```

Problem 4: Scheduling [18pts]

Problem 4a[4pts]: Here is a table of processes and their associated arrival and running times:

Process ID	Arrival Time	CPU Running Time
Process A	0	2
Process B	1	5
Process C	3	3
Process D	6	3
Process E	8	3

Show the scheduling order for these processes under 3 policies: First Come First Serve (FCFS), Shortest-Remaining-Time-First (SRTF), Round-Robin (RR) with timeslice quantum = 1. *Assume that context switch overhead is 0, that new processes are available for scheduling as soon as they arrive, and that new processes are added to the **head** of the queue (to be considered first for next scheduling slot) except for FCFS, where they are added to the **tail**.*

Time Slot	FCFS	SRTF	RR
0			
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			

Problem 4b[2pts]: Explain why a chess program running against another program on the same machine might want to perform a lot of superfluous I/O operations.

Problem 4c[2pts]:

Five jobs are waiting to be run. Their expected running times are 10, 8, 3, 1, and X . In what order should they be run to minimize average completion time? State the scheduling algorithm that should be used AND the order in which the jobs should be run. *HINT: Your answer will consist of multiple possible orderings dependent on the value of X (will have expressions including X):*

Problem 4d[2pts]: In class, we introduced the “one-to-one” model of threads (sometimes called the “two-stack” model), in which every user-level thread has two separate stacks – one in user-space and one in kernel-space. Pintos has this model. Explain the advantage that the one-to-one model has over a “many-to-one” model in which multiple user-level threads, each with their own stacks in user-space, share a single stack in kernel space.

Problem 4e[2pts]: In Pintos, the code for `thread_unblock()` contains a comment that says “This function does not preempt the running thread”. Explain why you should not modify `thread_unblock()` in a way that could cause it to preempt the running thread.

Problem 4f[2pts]:

Suppose that we have the following resources: A, B, C and threads T1, T2, T3, T4. The total number of each resource is:

Total		
A	B	C
12	9	12

Further, assume that the processes have the following current allocations and maximum requirements:

Thread ID	Current Allocation			Maximum		
	A	B	C	A	B	C
T1	2	1	3	4	3	4
T2	1	2	3	5	3	3
T3	5	4	3	6	4	3
T4	2	1	2	4	1	2

Is the system in a *safe* state? In making this conclusion, you must show all your steps, intermediate matrices, etc.

Problem 4g[2pts]:

Assuming the allocations in (4f), suppose that T1 asks for 2 more copies of A. Can the system grant this or not? Explain.

Problem 4h[2pts]:

Assuming the allocations in (4f), what is the maximum number of additional copies of resources (A, B, and C) that T1 can be granted in a single request without risking deadlock? Explain.

Problem 5: Virtual Memory Mapping [16 pts]

Consider the x86 multi-level memory management scheme for 32-bit mode. The top-level of the translation table is called a “page directory” (PD) and the second level is a “page table” (PT). Virtual addresses are formatted as:

Directory (10 bits)	Page Table (10 bits)	Offset (12 bits)
------------------------	-------------------------	---------------------

Virtual addresses are translated into physical addresses of the following form:

Physical Page # (20 bits)	Offset (12 bits)
------------------------------	---------------------

The table-entries at the top level (Page Directory Entries or PDEs) and second level (Page Table Entries or PTEs) are formatted identically. They are 32-bits, *stored in big-endian form* in memory (i.e. the MSB is first byte in memory), interpreted as follows:

Physical Page # (20 bits)	OS Defined (3 bits)	0	0	Dirty	Accessed	Nocache	Write Through	User	Writeable	Valid
------------------------------	------------------------	---	---	-------	----------	---------	---------------	------	-----------	-------

Here, “Valid” means that a translation is valid, “Writeable” means that the page is writeable, “User” means that the page is accessible by the User (rather than only by the Kernel).

Problem 5a[2pts]: Draw a picture to illustrate the process of walking a page in the x86 translation table. Make sure to include the CR3 register, at least one page directory and at least one page table, involved in translating a particular address. (i.e. Show where the information in the physical address comes from). We show the starting (virtual) and ending (physical) addresses:

VA:

10 bits	10 bits	12 bits
---------	---------	---------

PA:

20 bits	12 bits
---------	---------

Problem 5b[2pts]: What is the total size of the translation table (in bytes) that maps two pages at the top of the virtual address space and two pages at the bottom of the virtual address space? **Explain.** We want the size of the complete data structure that maps virtual addresses to physical addresses.

Problem 5c[2pts]: What is the maximum number of DRAM accesses that might be involved in accessing data using a virtual address in the x86 32-bit mode? What mechanisms would be involved in reducing the access time of a virtual address to a single first-level cache access time? Explain.

Problem 5d[10pts]: Assuming the memory translation scheme given above (5a). Suppose that the base table pointer for the current *user level process* (i.e. in CR3) points at `0x00200000`. Using the memory contents shown on the next page, predict results for the following instructions. Addresses in braces “[]” are virtual and are in hexadecimal. The return value for a load is an 8-bit data value or an error, while the return value for a store is either “ok” or an error. Possible errors are: **invalid, read-only, kernel-only**. (Hint: Don’t forget that hexadecimal digits contain 4 bits!)

Feel free to use the “scratch page” given at the end of the exam, but keep in mind that no answers will be taken from any place other than the boxes, below.

We have given you 3 results to start with:

Instruction	Result
Load [0x00001047]	0x04
Store [0x0000145F]	ERROR: read-only
Store [0x018015FF]	ok
Load [0x01801047]	

Instruction	Result
Test-And-Set [0x02005047]	
Load [0xFF800032]	
Load [0xFFFFE042]	
Store [0xFFFFE043]	

Physical Memory contents for (5d) in Bytes [All Values are in Hexadecimal]

Address	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+A	+B	+C	+D	+E	+F
00000000	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
00000010	1E	1F	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D
...																
00001010	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F
00001020	40	03	41	01	30	01	31	03	00	03	00	00	00	00	00	00
00001030	00	11	22	33	44	55	66	77	88	99	AA	BB	CC	DD	EE	FF
00001040	10	01	11	02	31	03	13	04	14	01	15	03	16	01	17	00
...																
00002030	10	01	11	00	12	03	67	03	11	03	00	00	00	00	00	00
00002040	02	20	03	30	04	40	05	50	01	60	03	70	08	80	09	90
00002050	10	00	31	01	10	03	31	01	12	03	30	00	10	00	10	01
...																
00004000	00	00	11	01	11	01	33	03	34	01	35	00	43	38	32	79
00004010	50	28	84	19	71	69	39	93	75	10	58	20	97	49	44	59
00004020	23	03	20	03	00	01	62	08	99	86	28	03	48	25	34	21
...																
00100000	00	00	10	65	00	00	20	67	00	00	20	65	00	00	40	07
00100010	00	00	50	03	00	00	00	00	00	00	20	67	00	00	00	00
...																
00102000	00	00	20	01	00	00	00	00	00	00	30	03	00	00	40	07
...																
00103000	11	22	00	05	55	66	77	88	99	AA	BB	CC	DD	EE	FF	00
00103010	22	33	44	55	66	77	88	99	AA	BB	CC	DD	EE	FF	00	67
...																
001FE000	00	00	10	67	48	59	70	7B	8C	9D	AE	BF	D0	E1	F2	03
001FE010	10	15	00	67	10	15	10	67	10	15	20	67	10	15	30	67
...																
001FF000	00	00	20	00	00	00	10	65	00	00	10	67	00	00	20	65
001FF010	00	00	20	67	00	00	30	67	00	00	40	65	00	00	50	07
...																
001FFFF0	00	00	00	00	00	00	00	00	00	00	20	67	00	00	40	65
...																
00200000	00	1F	F0	07	00	10	10	07	00	10	20	07	00	10	30	07
00200010	00	10	20	07	00	10	50	07	00	10	00	07	00	10	70	07
00200020	00	10	00	07	00	10	00	07	00	00	00	00	00	10	30	07
...																
00200FF0	00	00	00	00	00	00	00	00	00	1F	E0	07	00	1F	F0	07
...																

[This Page Intentionally Left Blank]

Problem 6: Copy-On-Write for Fork [12pts]

In this problem, we design a version of Copy-on-Write (COW) `fork()` for Pintos. Recall that in a standard `fork()`, all of the data in the parent's address space is copied into the child's address space. With COW, we instead share physical pages between parent and child and only copy a page when one of them tries to write it.

Since we need to track the situation in which the same page is present in multiple translation tables, we are going to use modified versions of the Pintos page-allocation functions that support **reference counting** for user-pool pages:

```
// Allocate a page from the user or kernel pool.
// For user-pool pages (PAL_USER), the frame's reference count => 1
// Possible flags are PAL_USER, PAL_ZERO, and PAL_ASSERT
void *palloc_get_page(enum palloc_flags flags);

// Decrement the reference count of a user-pool page.
// If the reference count reaches 0, the page is freed.
// For kernel-pool pages, the page is freed immediately.
void palloc_free_page(void *kpage);

// Increment the reference count of a user-pool page.
// Used when an additional page table maps an existing frame.
void palloc_add_page(void *kpage);

// Returns the current reference count of a user-pool page.
int palloc_get_refcnt(void *kpage);
```

Example usage:

```
void *kpage = palloc_get_page(PAL_USER); /* Allocate; ref_count = 1 */
palloc_add_page(kpage);                 /* ref_count is now 2 */
palloc_free_page(kpage);                /* ref_count is back to 1 */
palloc_free_page(kpage);                /* ref_count hits 0 -> freed*/
```

The following PTE flag definitions are to be used when writing code (feel free to look back at the picture of the PTE at the beginning of Problem 5).

```
/* PTE flag bits (from threads/pte.h) */
#define PTE_FLAGS 0x0000fff /* Flag bits. */
#define PTE_ADDR 0xfffff000 /* Address bits. */
#define PTE_AVL 0x0000e00 /* Bits 9-11: available for OS use. */
#define PTE_P 0x1 /* 1=present, 0=not present. */
#define PTE_W 0x2 /* 1=read/write, 0=read-only. */
#define PTE_U 0x4 /* 1=user/kernel, 0=kernel only. */
#define PTE_A 0x20 /* 1=accessed. */
#define PTE_D 0x40 /* 1=dirty (PTes only). */

/* COW marker -- uses bit 9 of the PTE (one of the OS-available bits). */
#define PTE_COW 0x200
```

Remember that Intel calls the top-level of a two-level page table a “page directory” and the second level a “page table”. Since, the page directory entries (PDEs) have the same format as PTEs, we will use the same constants, above.

Pintos provides the following functions for manipulating multi-level page tables:

(PD \equiv Page Directory, PDE \equiv Page Directory Entry, PT \equiv Page Table, PTE \equiv Page Table Entry):

```

/* Page directory (PD) / page table (PT) helpers */
uintptr_t pd_no(const void *va);          // PD index from VA
unsigned  pt_no(const void *va);          // PT index from VA
uint32_t *pde_get_pt(uint32_t pde);       // Get page table from PDE
void      *pte_get_page(uint32_t pte);    // Get kpage from PTE
uint32_t  pte_create_user(void *page, bool writable); // Build a user PTE
uint32_t  pde_create(void *pt);           // Build a PDE

/* Page directory operations (from userprog/pagedir.h) */
uint32_t *pagedir_create(void); // Empty user pagedir with kernel AS filled
void      pagedir_destroy(uint32_t *pd);
bool      pagedir_set_page(uint32_t *pd, void *upage,
                           void *kpage, bool writable);
void      *pagedir_get_page(uint32_t *pd, const void *uaddr);
void      pagedir_clear_page(uint32_t *pd, void *upage);
void      pagedir_activate(uint32_t *pd); /* Flushes TLB */

/* Address helpers (from threads/vaddr.h) */
#define PGSIZE 4096
#define PHYS_BASE ((void *) 0xC0000000)
bool  is_user_vaddr(const void *);
void  *pg_round_down(const void *);

/* other useful function */
void *memcpy(void *dst, const void *src, size_t n);
struct thread *thread_current(void);

```

Problem 6a[2pts]: Fill in the missing blanks in the existing Pintos function for *freeing* a translation table, such as when a process exits. Use constants and functions above and on previous page (*Hint: we are freeing up the USER portion, and only present pages (PTE_P) should be freed*):

```

void pagedir_destroy(uint32_t *pd) {
    uint32_t *pde;
    if (pd == NULL) return;
    for (pde = pd; pde < pd + pd_no(PHYS_BASE); pde++)

        if (_____ ) {
            uint32_t *pt = pde_get_pt(*pde);
            uint32_t *pte;
            for (pte = pt; pte < pt + _____; pte++)

                if (_____ ) {
                    // Free up page
                    palloc_free_page(_____);
                }
            palloc_free_page(pt); // Free up PT
        }
    palloc_free_page(pd); // Free up PD
}

```

Problem 6b[5pts]: Fill in the blanks to complete `cow_fork_pagedir()`, which creates a new page directory for a child process for copy-on-write. The incoming `parent_pd` is a pointer to the parent's translation table (i.e. the "page directory" at the top level). You should return a freshly created translation table for the child, in addition to potentially modifying the parent's translation table. Whenever possible, use functions and constants that we gave you on the previous two pages. Don't forget to manage the reference counts on shared pages. *Share each PRESENT user page with the child, after marking writeable pages as read-only so that writes invoke COW. Use the bit PTE_COW to keep track of the fact that you did this.*

```
uint32_t *cow_fork_pagedir(uint32_t *parent_pd) {

    uint32_t *child_pd = _____;
    // Walk Page Directory (PD)
    for (uint32_t pdi = 0; pdi < pd_no(PHYS_BASE); pdi++) {

        if (_____ ) {
            child_pd[pdi] = parent_pd[pdi]; // Replicate any OS info in PDE
            continue;
        }
        uint32_t *parent_pt = pde_get_pt(parent_pd[pdi]);
        uint32_t *child_pt = palloc_get_page(PAL_ZERO);
        child_pd[pdi] = pde_create(child_pt);

        // Walk Page Table (PT)
        for (uint32_t pti = 0; pti < _____; pti++) {
            uint32_t pte = parent_pt[pti];
            child_pt[pti] = pte; // Replicate PTE as base case

            if (_____ )
                continue;

            // Register shared page

            _____;

            _____;

            // Set up for later COW if needed

            if (_____ ) {

                uint32_t COW_pte = _____;

                child_pt[pti] = _____;

                parent_pt[pti] = _____;

            }
        }
    }
    pagedir_activate(child_pd); // Flush TLBs
    return child_pd;
}
```

Problem 6c[5pts]: Add COW support to page fault handler, below. When a write fault occurs on a PRESENT page, we check whether the page is marked COW. If the current process is the *sole* owner of the physical frame, we can simply make the page writable again. Otherwise, we must copy the frame and update the PTE. Make sure to use helper functions that we gave you on the previous page when possible. *Hint: Don't forget to manage PTE_COW flag and reference counts!*

```

static void handle_page_fault(struct intr_frame *f) {
    bool not_present, write, user;
    void *fault_addr;

    asm ("movl %%cr2, %0" : "=r" (fault_addr)); // Get fault address
    intr_enable();
    not_present = (f->error_code & PF_P) == 0;
    write       = (f->error_code & PF_W) != 0;
    user        = (f->error_code & PF_U) != 0;

    /* --- COW handling --- (conditions right to possibly handle COW) */

    if (_____ ) {
        struct process *pcb = thread_current()->pcb;
        uint32_t *pd = pcb->pagedir;
        void *fault_page = pg_round_down(fault_addr);

        /* Mini page walk to find the PTE */
        uint32_t *pde = pd + pd_no(fault_page);
        uint32_t *pt = pde_get_pt(*pde);

        uint32_t *pte = _____;

        if (*pte & PTE_COW) {
            // Get current page
            void *old_kpage = _____;

            if (_____ ) {
                /* We are the sole owner */
                _____;
            }

            } else {
                /* Must copy the page */
                void *new_kpage = palloc_get_page(PAL_USER);

                _____;

                _____;

                _____;
            }
            pagedir_activate(pd); // Flush TLB
            return;
        }
    }
}
/* Handle normal page fault (which in Pintos results in a Panic())! */
}

```

[Function Signature Cheat Sheet]

```

/***** pThreads *****/
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
void *(*start_routine) (void *), void *arg);
int pthread_join(pthread_t thread, void **retval);
int pthread_mutex_init(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);

int pthread_cond_init(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);

/***** Processes *****/
pid_t fork(void);
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
int execv(const char *path, char *const argv[]);

/***** High-Level I/O *****/
FILE *fopen(const char *path, const char *mode);
FILE *fdopen(int fd, const char *mode);
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
int fclose(FILE *stream);

/***** Sockets *****/
int socket(int domain, int type, int protocol);
int bind(int sockfd, struct sockaddr *addr, socklen_t addrlen);
int listen(int sockfd, int backlog);
int accept(int sockfd, structure sockaddr *addr, socklen_t *addrlen);
int connect(int sockfd, struct sockaddr *addr, socklen_t addrlen);
ssize_t send(int sockfd, const void *buf, size_t len, int flags);

/***** Low-Level I/O *****/
int open(const char *pathname, int flags);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
off_t lseek(int fd, off_t offset, int whence);
    whence=>SEEK_SET, SEEK_CUR, or SEEK_END
int dup(int oldfd);
int dup2(int oldfd, int newfd);
int pipe(int pipefd[2]);
int close(int fd);

```

[Function Signature Cheat Sheet (con't)]

```

/***** Pintos *****/
void list_init(struct list *list);
struct list_elem *list_head(struct list *list)
struct list_elem *list_tail(struct list *list)
struct list_elem *list_begin(struct list *list);
struct list_elem *list_next(struct list_elem *elem);
struct list_elem *list_end(struct list *list);
struct list_elem *list_remove(struct list_elem *elem);
bool list_empty(struct list *list);
#define list_entry(LIST_ELEM, STRUCT, MEMBER) ...
void list_insert(struct list_elem *before, struct list_elem *elem);
void list_push_front(struct list *list, struct list_elem *elem);
void list_push_back(struct list *list, struct list_elem *elem);
struct list_elem *list_pop_front(struct list *list);
struct list_elem *list_push_front(struct list *list);

void sema_init(struct semaphore *sema, unsigned value);
void sema_down(struct semaphore *sema);
void sema_up(struct semaphore *sema);
void lock_init(struct lock *lock);
void lock_acquire(struct lock *lock);
void lock_release(struct lock *lock);

/* Page directory (PD) / page table (PT) helpers */
uintptr_t pd_no(const void *va); // PD index from VA
unsigned pt_no(const void *va); // PT index from VA
uint32_t *pde_get_pt(uint32_t pde); // Get page table from PDE
void *pte_get_page(uint32_t pte); // Get kpage from PTE
uint32_t pte_create_user(void *page, bool writable); // Build a user PTE
uint32_t pde_create(void *pt); // Build a PDE

/* Page directory operations (from userprog/pagedir.h) */
uint32_t *pagedir_create(void); // Empty user pagedir with kernel AS filled
void pagedir_destroy(uint32_t *pd);
bool pagedir_set_page(uint32_t *pd, void *upage,
                      void *kpage, bool writable);
void *pagedir_get_page(uint32_t *pd, const void *uaddr);
void pagedir_clear_page(uint32_t *pd, void *upage);
void pagedir_activate(uint32_t *pd); /* Flushes TLB */

/* Address helpers (from threads/vaddr.h) */
#define PGSIZE 4096
#define PHYS_BASE ((void *) 0xC0000000)
bool is_user_vaddr(const void *);
void *pg_round_down(const void *);

/* other useful function */
void *memcpy(void *dst, const void *src, size_t n);
struct thread *thread_current(void);

```

[Scratch Page: Do not put answers here!]

[Scratch Page: Do not put answers here!]