Name: _____

Write your name and computing ID above. Write your computing ID at the top of each page in case pages get separated. Sign the honor pledge below.

Generally, we will not answer questions about the exam during the exam time. If you think a question is unclear and requires additional information to answer, please explain how in your answer. For multiple choice questions, write a * next to the relevant option(s) along with your explanation.

On my honor as a student I have neither given nor received aid on this exam.

_____

1. (10 points) Suppose an out-of-order processor, similar to the design we discussed in lecture and shown on the reference sheet, was running the following assembly snippet:

```
addq %r8, %r9
movq (%r9), %r12
subq %r8, %r10
xorq %r10, %r11
imulq %r8, %r12
movq %r12, (%r9)
```

If the processor has many (more than 10) functional units for executing arithmetic instructions that each have a latency of one cycle and a functional unit for doing data cache loads and stores that has a latency of three cycles, then answer below:

- what is the fastest (in cycles) the processor could do the arithmetic or loads/stores for the above instructions?; *and*
- how many arithmetic functional units would be used at the same time?

Assume that after a value is computed by a functional unit, it can be used for a calculation or memory access in the following cycle.

Solution: schedule: (cycle 1) add sub; (cycle 2) xorq, movq (load) cycle 1; (cycle 3) movq (load) cycle 2; (cycle 4) movq (load) cycle 3; (cycle 5) imul; (cycle 6) movq (store) cycle 1 (cycle 7) movq (store) cycle 2 (cycle 8) movq (store) cycle 3 — 8 cycles, 2 ALUs at once

2. A program that manages a calendar stored on a remote server from a client machine. The calendar consists of a series of *events*, each of which have a time, a unique name that consists of spaces and alphanumeric characters, and a status which is one of *pending*, *confirmed*, or *declined*.

   Clients send commands following one of the following templates to the server:

format	example	meaning
`confirm;`*`Name`*	`confirm;Big Meeting`	change an event's status
`decline;`*`Name`*	`decline;Big Meeting`	
`range;`*`StartDate`*`;`*`EndDate`*	`range;2024-08-01;2024-09-01`	request list of events

   When successful, servers respond to these command with a series of messages in the format "*`Date`*`;`*`Status`*`;`*`Name`*"
   indicating that event `Name` has status `Status` and date `Date`. After this series of messages the server will send the special message "`end of response`".

   If a command is unsuccessful because an event does not exist, the server will respond with the message in the format "does not exist;*`Name`*".

(a) (8 points) Suppose one wants to modify the protocol above to use cryptography to prevent untrusted entities operating routers from reading or manipulating calendar entries.

Consider a flawed attempt to do this:

- in advance, the client securely receives a public encryption key and a public signature verification key for the server and a unique passphrase it will use to verify its identity with the server
- whenever the client sends a command to the server, it also includes the passphrase and a unique identifier (which is different for each command), then it encrypts the resulting command (including the passphrase and identifier) using the server's public key before sending it to the server
- when the server sends back a response message for a command, it also includes the unique identifier with each response message, and it signs each response message with its private signing key (the 'range' command will still require multiple response messages, each of which will be signed separately)

Consider two types of malicious router:

- passive: one that forwards messages correctly, but records and analyzes all the messages
- active: one that replaces the client or server's messages with its own or with copies of messages that were sent earlier

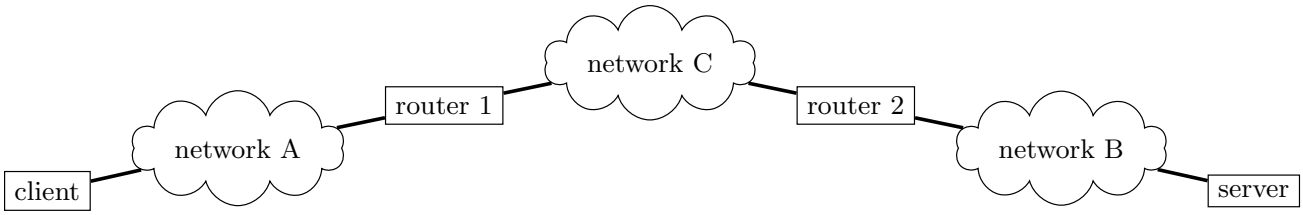
Fill in the table below by placing ✓s indicating which of the following are possible for each type of attacker to do.

You may assume that the attacker can guess what events are present in the calendar and which operation the client is most likely attempting a significant amount of the time, and you should consider an attack possible if it would succeed when that guess is correct. You may assume that either type of malicious router has access to all relevant public keys, but has no access to or ability to guess any passphrases or private keys. *-1 point per disagreement, ignoring when passive is correct and checked and active is not checked*

passive	active	action
✓	✓	learn the names of events on the client's calendar if the client lists events in a range
	✓	make a list of events in a time range falsely appear to be empty
✓	✓	verify that an event with a particular date and time is included in what the server sent
✓	✓	identify, if they could not guess already, which event was manipulated by a confirm or decline command *also accepted no/no, for those interpreting this as the attacker needing to distinguish an accept/decline from a read-out-range command*
	accept either	if the client tries to confirm event A and decline event B, make the client end up declining event A and confirming event B

for last item, yes if we can manipulated encrypted data; no otherwise

(b) (8 points) Suppose the server and client are connected as shown in the diagram below: The server and client are on two different wired networks A and B. The client's network A is connected to a router 1, which is connected to a wired network C. The server's network B is connected to a router 2, which is also connected to wired network C. Wired network C and router 1 and router 2 are the only connection between the client and sever.



Suppose the client successfully sends "`confirm;The Meeting`" to the server. (For this question, assume no cryptography is in use.)

For each blank below identify of the options (A-G) below would complete it. (You may use the same option multiple times.)

For its part in sending this message, the client will send a (link-layer) frame whose source address represents A and whose destination address represents C. That frame will contain an (IP) packet whose source address represents A and whose destination address represents B.

Then, C will copy the packet to a new frame, whose destination address will represent D.

A. the client machine
B. the server machine
C. the router 1
D. the router 2
E. a router not associated with networks A and B on network C
F. the client program as opposed to other programs on the client machine
G. the server program as opposed to other programs on the server machine

3. Consider the following sequence of events on a single-core Unix-like system:
   1. an `ls` command finishes running, and the shell displays a prompt for the next command
   2. the shell starts waiting for input, and while it waits, a simulation program does some computation
   3. a human enters the command '`editor file.c`', and the shell starts processing this command
   4. the program `editor` starts running as the shell starts waiting for it to finish
   5. the editor program tries to open `file.c`, but finds that it does not have permission to access it
   6. the editor displays an error message and prompts whether to exit or retry
   7. a human enters the command to exit the editor, and the editor processes this command
   8. the shell resumes running and displays the prompt for the next command

(a) (4 points) Which items above will require a process context switch? Identify them by their numbers.

Solution: 1, 2, 3, 4, 8

(b) (6 points) Which items above (identify by number) will require one or more exceptions that is not a system call? For each such item, identify it by number and also *briefly* identify the causes of the non-system-call exceptions.

Solution: 3 / keyboard input ; allow 5 / disk I/O or omitting 5; 7 / keyboard input or omitting 7 (assuming system call catches keyboard input without exception notifying); allow page fault exceptions for loading program, allow timer exceptions for 2

(c) (4 points) To implement step 5, suppose the editor calls the C standard library function `fopen`. The implementation of this function will most likely (directly or indirectly) _____.

- ○ make a system call to retrieve the file permissions, then compare the current process's user and group ID to those permissions, then return an error based on the result of that comparison, skipping code that would make another system call to open the file
- ■ make a system call to open the file, then check the return value of that system call, and decode it to realize that there was a permissions problem
- ○ make a system call to open the file, which will trigger a segmentation fault signal, which the C standard library handles to indicate the permissions problem
- ○ make a system call to set the user ID and group ID of the program to ensure permissions to be checked properly, then make a system call to try to open the file, then discover that that system call fails
- ○ none of the above, explain *briefly*: _____

4. Consider the following Makefile:

```
1  util.o: util.c util.h
2          clang -g -Og -Wall -c util.c -o util.o
3
4  server-main.o: server-main.c server.h
5          clang -g -Og -Wall -c server-main.c -o server-main.o
6
7  server.o: server.c server.h util.h
8          clang -g -Og -Wall -c server.c -o server.o
9
10 server: server.o server-main.o util.o
11         clang -g -Og -o server server.o server-main.o util.o
12
13 client-main.o: client-main.c client.h
14         clang -g -Og -Wall -c client-main.c -o client-main.o
15
16 client.o: client.c client.h util.h
17         clang -g -Og -Wall -c client.c -o client.o
18
19 client: client.o client-main.o util.o
20         clang -g -Og -o client client.o client-main.o util.o
```

(a) (6 points) Suppose files have the following modification times:

filename	time
client.h	10 hours ago
client.c	9 hours ago
client.o	8 hours ago
client-main.c	4 hours ago
client-main.o	5 hours ago

filename	time
server.h	11 hours ago
server.c	4 hours ago
server.o	3 hours ago
server-main.c	2 hour ago
server-main.o	1 hour ago

filename	time
util.h	7 hours ago
util.c	6 hours ago
util.o	5 hours ago

and the files `client` and `server` do not exist. Then, running `make client` and then `make server` will cause make to run which commands? *-1 per disagreement* **Select all that apply.**

- ☐ `clang -g -Og -Wall -c util.c -o util.o`
- ☐ `clang -g -Og -Wall -c server-main.c -o server-main.o`
- ☐ `clang -g -Og -Wall -c server.c -o server.o`
- ■ `clang -g -Og -o server server.o server-main.o util.o` *-Wall was included in this command on the exam as printed; so the command as printed was not run, so also accepted this not being checked*
- ■ `clang -g -Og -Wall -c client-main.c -o client-main.o`
- ■ `clang -g -Og -Wall -c client.c -o client.o`
- ■ `clang -g -Og -o client client.o client-main.o util.o` *-Wall was included in this command in the exam as printed, so the command as printed was not run, so also accepted this not being checked*

5. Consider the following code:

```
int secret;
int table1[4096];
...
int table2[4096];
int FunctionThatUsesSecret(int x, int y) {
    if (x < 2048 && y < 4096) {
        return table2[table1[(secret + x) % 4096]] + y;
    } else if (x < 4096) {
        return table2[table1[x]];
    }
}
```

Suppose this is run on a system with a 16384-byte (2^{14} byte) direct-mapped data cache with 256-byte blocks that does not use virtual memory (so all addresses are physical).

Assume that `table1` and `table2` are each assigned addresses `0x100000` and `0x120000` respectively. (Since these addresses are multiples of `16384`, this means that `table1[0]` maps to set index 0, offset 0 in the cache, and the same for `table2[0]`.)

This function can be used as part of side-channel attacks where, by detecting what is evicted from the cache when the function is run on an out-of-order processor, an attacker can detect information about either `secret` or arbitrary locations in memory (depending how the attacker arranges for the function to be run).

(a) (6 points) The function is called with `x=0` and `y=0` and with the contents of `table1` and `table2` not initially stored in the data cache. It returns `158` and evicts from the cache sets with index 4 and index 10. Based on this information, give **a** possible value of `secret`. You may leave your answer as an unsimplified arithmetic expression.

Note that the arrays are a arrays of **4-byte ints**, not chars.

Solution: 256 / 4 * 4 + (zero to 255 / 4) OR 256 / 4 * 10 + (zero to 255 / 4) OR these answers + multiples of 4096; -1 point for missing divide by 4

(b) (4 points) Suppose we want to use the cache eviction side-channel to get information about values in memory from `0x200000`. In order for this attack to succeed, how should the branch predictor predict the conditions in if statements in the code above?

- ○ it does not matter
- ○ it should predict the `x < 2048 && y < 4096` as true (and it doesn't matter how `x < 4096` is predicted)
- ○ it should predict the `x < 2048 && y < 4096` as false (and it doesn't matter how `x < 4096` is predicted)
- ■ it should predict the `x < 2048 && y < 4096` as false and `x < 4096` as true
- ○ it should predict the `x < 2048 && y < 4096` as false and `x < 4096` as false
- ○ something else, explain: _____

6. Consider a processor with a **seven-stage pipeline** such that:

   - instructions are fetched from the instruction cache in the first stage
   - register file values are read in the second stage
   - arithmetic performed by an instruction is computed in the third and fourth stage; arithmetic operands must be ready by near the beginning of the third stage and results will be available by near the end of fourth stage
   - accesses to memory (that is, the data cache) need the addresses available near the start of the fifth stage and have their result (if a read) available near the end of the sixth stage
   - register file writes are performed in the seventh stage and need the value to be written available near the beginning of the seventh stage
   - when branches are mispredicted, the processor spends three cycles fetching instructions that will need to be squashed
   - the processor uses a combination of stalling and forwarding to resolve data hazards

(a) (7 points) Give an example of assembly snippet with at most three instructions that would require **exactly** 4 cycles of stalling (that is, would take an extra 4 cycles in total to run compared to no stalling) to execute on the processor.

(Assume there is no stalling from cache misses.)

Solution: multiple answers, example: mov (%r8), %r9; add %r9, %r10; add %r10, %r11

(b) (5 points) Consider if the processor is running the following sequence of instructions:

```
                          0 1 2 3 4 5 6 7 8 9 X
addq  %r8, %rbx           F D E E M M W
imulq %rbx, %r8             F D D E E M M W
subq  %rbx, %r9               F F D E E M M W
xorq  %r8, %rbx                   F D E E M M W
```

(If you don't remember operand order, etc. in x86-64 assembly, see the reference sheet.)

Identify where forwarding will be used when the above instructions are run. (Show any relevant work above. It is possible that not all necessary forwarding is included in the options below.) **Select all that apply.**

- ■ from `addq` to `imulq` for the value `%rbx`
- ☐ from `addq` to `imulq` for the value `%r8`
- ■ from `addq` to `subq` for the value `%rbx`
- ■ from `addq` to `xorq` for the value `%rbx`
- ☐ from `addq` to `xorq` for the value `%r8`

7. Consider the following C code:

```
1  pid_t temp_pid;
2  void handler(int ignored_signum) {
3      temp_pid = getpid();
4      write(STDOUT_FILENO, "X", 1);
5  }
6  int main() {
7      pid_t orig_pid = getpid();
8      pid_t new_pid;
9      struct sigaction sa;
10     sa.sa_handler = handler;
11     sigemptyset(&sa.sa_mask); sa.sa_flags = SA_RESTART;
12     sigaction(SIGUSR1, &sa, NULL);
13     int fds[2];
14     if (pipe(fds) < 0) handle_error();
15     new_pid = fork();
16     if (new_pid == 0) {
17         close(fds[0]);
18         write(fds[1], "A", 1);
19         kill(orig_pid, SIGUSR1);
20         write(fds[1], "B", 1);
21         exit(0);
22     } else if (new_pid > 0) {
23         close(fds[1]);
24         char c, d;
25         read(fds[0], &c, 1);
26         write(STDOUT_FILENO, c, 1);
27         read(fds[0], &d, 1);
28         write(STDOUT_FILENO, d, 1);
29     } else { handle_error(); }
30     return 0;
31 }
```

Assume the calls to `handle_error` are never reached and needed header files are `#included`.

(a) (4 points) When the `handler` function is run by the above code, the value assigned to `temp_pid` (on line 3) will be _____.

- ■ the same as `orig_pid`
- ○ the same as `new_pid`
- ○ either the same as `new_pid` or the same as `orig_pid`, depending on timing
- ○ none of the above because `handler` will not be run by the above code
- ○ none of the above because `getpid()` would return something else
- ○ something else, explain: _____

(b) (6 points) What are ***two*** possible outputs of the above code?

Solution: two of: AB, AXB, ABX, XAB

8. (20 points) Complete the following C code by filling in the blanks and following the description below. (Not all lines may be necessary.)

   In the code below, the `Player` struct tracks up to one accepted team and up to one offering team. If a player has no offering team or accepted team, the corresponding fields of the struct are NULL.

   A thread running on behalf of a player can call `GetNextOffer` to wait for an offering team to be set, or `AcceptOffer` or `RejectOffer` to accept or reject the current offering team.

   A thread running on behalf of a team can run `MakeOfferAndWaitForAcceptOrReject`. This function should first wait for a player to have no team offering or accepted. Then, if the player has accepted an offer already, it returns false. Otherwise, it should edit the `Player` struct to indicate the current team is offering, then wait for the player to accept or reject the offer, then return true if the offer was accepted and return false otherwise.

```
pthread_mutex_t lock;
struct Team {
    const char *name;
    int num_players;
};
struct Player {
    const char *name;
    pthread_cond_t offer_change_cv;
    struct Team *offering_team;  // or NULL, if none
    struct Team *accepted_team;  // or NULL, if none
};

struct Team *GetNextOffer(struct Player *player) {
    struct Team *result;
    pthread_mutex_lock(&lock);
    while (player->offering_team == NULL) {
        pthread_cond_wait(&player->offer_change_cv, &lock);
    }
    result = player->offering_team;
    pthread_mutex_unlock(&lock);
    return result;
}

void AcceptOrRejectOffer(bool accepted, struct Player *player) {
    pthread_mutex_lock(&lock);
    if (accepted) {
        player->accepted_team = player->offering_team;
    } else {
        player->offering_team = NULL;
    }
    /* BLANK 1 */
    pthread_cond_broadcast(&player->offer_change_cv);
        /* also accept signal */
    pthread_mutex_unlock(&lock);
}

void RejectOffer(struct Player *player) {
```

```
    AcceptOrRejectOffer(false, player);
}
void AcceptOffer(struct Player *player) {
    AcceptOrRejectOffer(true, player);
}

bool MakeOfferAndWaitForAcceptOrReject(sturct Team* team, struct Player
 ↪  *player) {
    pthread_mutex_lock(&lock);
    while (player->offering_team != NULL && player->accepted_team == NULL)
     ↪  {
        pthread_cond_wait(&player->offer_change_cv, &lock);
    }
    bool successful = false;
    if (player->accepted_team == NULL) {
        player->offering_team = team;
        pthread_cond_broadcast(&player->offer_change_cv);
        while (/* BLANK 2 */ player->offering_team == team &&
         ↪  player->accepted_team == NULL) {
            /* some notes on common alternatives for BLANK 2:
               - checking player->offering_team != NULL instead of == team
                 will wait too long if the player rejects and another team
                  ↪  offers very quickly
               - failing to check accepted_team somehow will wait
                ↪  indefinitely
                 if the player accepts
               - using || will wait indefinitely if the player rejects the
                ↪  offering
             */
            /* BLANK 3 */ pthread_cond_wait(&player->offer_change_cv,
             ↪  &lock);
        }
        successful = (player->accepted_team == team);
    }
    pthread_mutex_unlock(&lock);
    return sucessful;
}
```

9. Consider a system with:
   - 2-level page tables, with 8192 (2^{13}) entries in page tables at each level
   - 16-bit page offsets
   - 8-byte page table entries
   - a 4-way TLB with a total of 128 entries (split across the 4 ways) an LRU replacement policy
   - a 2-way 8192 (2^{13}) byte L1 data cache with 256-byte blocks that uses physical addresses, an LRU replacement policy, and write-back and write-allocate policies
   - a 2-way 8192 (2^{13}) byte L1 instruction cache with 256-byte block that uses physical addresses and an LRU replacement policy
   - an 4-way 1048576 (2^{20}) byte L2 cache, with 256-byte blocks shared between instructions and data, with an LRU replacement policy, and write-back and write-allocate policies

(a) (12 points) Suppose virtual address `0x123456789` maps to physical address `0x4FFFF6789`.

When this virtual to physical address translation is performed using page tables directly (as on a TLB miss), a first-level page table at physical address `0x400000` and a second-level page table at physical address `0x800000` is used.

Fill in the blanks below. You may leave answers as unsimplified arithmetic expressions.

If this virtual to physical address translation is performed using the TLB (as on a TLB hit), then an entry in the TLB with set index 0x5 will be used and contain page number 0x4FFFF. This page number was previously derived from the page table entry at address 0x800000 + 0x345 times 8.

(b) (10 points) How the operating system sets up the page tables for and locates a program's data can affect the hit rate for that data in the L2 cache.

Suppose a program's accesses to the L2 cache are only to a global array:

```
unsigned char array[65536 * 8];
```

(65536 is 2^{16}.) and the program repeatedly accesses the array sequentially as in:

```
for (int j = 0; j < LARGE_NUMBER; j += 1) {
    for (int i = 0; i < 65536 * 8; i += 1) {
        array[i] += i;
    }
}
```

How could the page tables for the array and the location of the array be setup to ***minimize*** the hit rate for the accesses to the array? Explain *briefly* why your setup minimizes the hit rate.

Solution: the lower 2 (18-16) bits of the *physical* page numbers for the pages that make up the array should be idenntical; for example, placing the physical pages of the array at 0x10000000, 0x20000000, 0x3000000, etc. This will make the memory of the array map to as few cache sets as possible, by reducing the number of set index bit vlaues Also, the array should ideally be located at a non-multiple-of-256 address so it spans more cache blocks.

(c) (4 points) What is the maximum space that could be used to store the page tables for a single process on this system? You may leave your answer as a unsimplified arithmetic expression.

Solution: $(1 + 2^{13}\text{tables}) \times (2^{13}\text{entries} \times 8\text{bytes per entry})$

(d) (6 points) Suppose a process on this system has the following memory layout:

from	to	usage	permissions
`0x000000`	`0x00FFFF`	unused	inaccessible
`0x010000`	`0x03FFFF`	code	read, execute
`0x040000`	`0x05FFFF`	globals	read, write
`0x060000`	`0x06FFFF`	heap	read, write
`0x070000`	`0x0FFFFF`	unused	inacessible
`0x100000`	`0x11FFFF`	stack	read, write
`0x11FFFF`	(highest possible)	unused	inaccessible

Suppose the system

- uses the allocate- and/or load-on-demand strategy where the page tables initially are entirely invalid
- as part of the allocate-on-demand strategy, page table entries are filled in response to page faults (a type of exception) as the process tries to use memory that is not mapped in the page table
- the operating system fills in the minimum number of page table entries required to make the memory access work each time

Suppose when the program starts it runs the following operations in the order shown below. For each of them, in the *faults* column, identify how many page faults or any other hardware exceptions resulting from a memory access will occur. *-1 per wrong*

#	operation	faults	
1	using code at `0x10400`, store 0 in `%rdi`	1	
2	using code at `0x10408`, push the value 1 on the stack at address `0x117FF8`	1	
3	using code at `0x10410`, push the value 2 on the stack at address `0x117FF0`	0	
4	using code at `0x10418`, jump to address `0x20800`	0	*accept 1*
5	using code at `0x20800`, copy an integer from `0x3FFF8` *was written 0x3FFFF8 on exam* to `%rsi`	2	
6	using code at `0x20808`, copy an integer from `%rsi` to `0x117FE0`	0	*dropped*