Exam 3

601.229 Computer Systems Fundamentals

December 16, 2021

Complete all questions.

Time: 90 minutes.

I affirm that I have completed this exam without unauthorized assistance from any person, materials, or device.

Signed: ______________________________

Print name: ______________________________

Date: ______________________________

Reference

Powers of 2 ($y = 2^x$):

x	0	1	2	3	4	5	6	7	8	9	10	11	12
y	1	2	4	8	16	32	64	128	256	512	1,024	2,048	4,096

x	13	14	15	16
y	8,192	16,384	32,768	65,536

Note that in all questions concerning C:

- `uint8_t` is an 8-bit unsigned integer type
- `uint16_t` is a 16-bit unsigned integer type
- `uint32_t` is a 32-bit unsigned integer type
- `int8_t` is an 8-bit signed two's complement integer type
- `int16_t` is a 16-bit signed two's complement integer type
- `int32_t` is a 32-bit signed two's complement integer type

x86-64 registers:

Callee-saved: `%rbx`, `%rbp`, `%r12`, `%r13`, `%r14`, `%r15`

Caller-saved: `%r10`, `%r11`

Return value: `%rax`

Arguments: `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`

Note that argument registers and return value register are effectively caller-saved.

Registers and sub-registers:

Register	Low 32 bits	Low 16 bits	Low 8 bits
`%rax`	`%eax`	`%ax`	`%al`
`%rbx`	`%ebx`	`%bx`	`%bl`
`%rcx`	`%ecx`	`%cx`	`%cl`
`%rdx`	`%edx`	`%dx`	`%dl`
`%rbp`	`%ebp`	`%bp`	`%bpl`
`%rsi`	`%esi`	`%si`	`%sil`
`%rdi`	`%edi`	`%di`	`%dil`
`%r8`	`%r8d`	`%r8w`	`%r8b`
`%r9`	`%r9d`	`%r9w`	`%r9b`
`%r10`	`%r10d`	`%r10w`	`%r10b`
`%r11`	`%r11d`	`%r11w`	`%r11b`
`%r12`	`%r12d`	`%r12w`	`%r12b`
`%r13`	`%r13d`	`%r13w`	`%r13b`
`%r14`	`%r14d`	`%r14w`	`%r14b`
`%r15`	`%r15d`	`%r15w`	`%r15b`

Stack alignment: `%rsp` must contain an address that is a multiple of 16 when any `call` instruction is executed.

Operand size suffixes: **b** = 1 byte, **w** = 2 bytes, **l** = 4 bytes, **q** = 8 bytes (Examples: `movb`, `movw`, `movl`, `movq`)

Question 1. [20 points] On the 32-bit x86 architecture, the page size is 2^{12} = 4096 bytes, and there are two levels of page tables. Each page table (both the root page table and the second-level page tables) has 1024 page table entries. Virtual addresses are 32 bits, and each virtual page in the entire 32-bit address space can be mapped to a physical page. Assume the bits in a virtual address are numbered 0–31, with 0 being the least significant bit, and 31 being the most significant bit.

(a) Which bits of a virtual address are the *page offset*?

(b) Which bits of a virtual address are used as the index in the "root" (level 1) page table, in order to find the page table entry leading to the level 2 page table?

(c) Which bits of a virtual address are the index in the level 2 page table, in order to find the page table entry leading to the mapped physical page?

(d) How many virtual pages are there in the overall address space? You may express this as a power of 2 or sum of powers of 2.

(e) Assume that a virtual address space maps every virtual page to a corresponding physical page. How many page tables (at both levels) are needed? You may express this as a power of 2 or sum of powers of 2.

Question 2. [10 points] Consider the following server loop which uses processes to allow concurrent client connections:

```
1:    while (1) {
2:      int clientfd = accept(serverfd, NULL, NULL);
3:      pid_t pid = fork();
4:      if (pid == 0) {
5:        chat_with_client(clientfd);
6:        exit(0);
7:      }
8:      close(clientfd);
9:    }
```

Briefly explain the reason why the call to `close` is needed at line 8.

Question 3. [10 points] Consider the following function, which is meant to write the contents of a buffer in memory to a file descriptor:

```
// Returns 1 if successful, 0 if unsuccessful
int send_data(const void *buf, unsigned num_bytes, int fd) {
  ssize_t bytes_sent = write(fd, buf, num_bytes);
  if (bytes_sent >= 0 && (unsigned)bytes_sent == num_bytes) {
    return 1;
  } else {
    return 0;
  }
}
```

Briefly explain the most important flaw in this function, and how to fix it. (You don't need to show code for a fixed version.) Hint: consider that `fd` might refer to a TCP socket.

Question 4. [20 points] Consider the following partially-specified multithreaded server implementation (note that error handling is omitted, and assume appropriate system headers are #included):

```
void chat_with_client(int fd); // defined elsewhere

struct ConnInfo {
  HERE 1
};

void *worker(void *arg) {
  HERE 2
}

int main(int argc, char **argv) {
  int serverfd = Open_listenfd(argv[1]);
  while (1) {
    int clientfd = accept(serverfd, NULL, NULL);
    struct ConnInfo *info = malloc(sizeof(struct ConnInfo));
    HERE 3
    pthread_t thr;
    pthread_create(&thr, NULL, worker, HERE 4);
  }
}
```

Indicate what code should be substituted for the missing code labeled `HERE 1`, `HERE 2`, `HERE 3`, and `HERE 4`. Assume that the `chat_with_client` function implements sending data to and receiving data from the remote client.

Question 5. [15 points] Assume that the message format for a network protocol is defined as follows. Each message is a single line of text terminated by a newline (`'\n'`) character. The content of a line is a *code* specified by an upper case letter (`'A'` through `'Z'`), immediately followed by an integer *value* specified as a sequence of 1 or more digit characters (`'0'` through `'9'`).

Examples of messages:

```
Q9
B55
Y90125
```

Struct data type to represent a message:

```
struct Message {
  char code;
  int value;
};
```

Implement the following `recv_msg` function so that it reads a single message from the specified file descriptor and uses the received data to fill in the contents of the `struct Message` instance pointed-to by the parameter `p`.

Hints and specifications:

- The function should read *only* data that is part of *one* message
- It will probably be easiest to read one character at a time
- The function prototype for the `read` system call is
  `int read(int fd, void *buf, size_t n);`
- The function prototype for the `atoi` function (to convert a NUL-terminated string of digits to an `int` value) is
  `int atoi(const char *str);`
- You may use the `isalpha` and/or `isdigit` functions
- You may assume that the received data is properly formatted and that no errors will occur

```
void recv_msg(int fd, struct Message *p) {
```

Question 6. [10 points] Consider the following operations performed by two different threads without synchronization:

```
// Thread 1                    // Thread 2
foo = x                        bar = x
foo = foo * 2                  bar = bar + 1
x = foo                        x = bar
```

Assume that `x` is a shared variable accessible by both threads, and that its initial value (before either thread starts) is 1. What possible final value(s) could `x` have after both threads finish their operations? Explain briefly.

Question 7. [15 points] Consider the following C data type and functions:

```
struct SharedVec3 {

  float data[3];

};

// Initialize the SharedVec3 object before threads start using it
void svec3_init(struct SharedVec3 *sv) {

  for (int i = 0; i < 3; i++) { sv->data[i] = 0.0f; }

}

void svec3_addto(struct SharedVec3 *sv, int index, float val) {

  sv->data[index] += val;

}

float svec3_get(struct SharedVec3 *sv, int index) {

  float value = sv->data[index];

  return value;

}
```

Show how to add synchronization to the data type and functions so that it is safe for concurrent use by multiple threads. Indicate your changes above.

[Extra page for answers and/or scratch work.]

[Extra page for answers and/or scratch work.]

[Extra page for answers and/or scratch work.]