CS 154, Spring 2020: Exam 2

⟹ HONOR CODE (IMPORTANT): write your name at the top of the answer template txt to indicate that you:

1. have read and understood all the instructions below
2. agree to The University of Chicago's Academic Honesty and Plagiarism policy
   (a) I affirm that I will not give or receive any unauthorized help on this exam, and that this work is my own
   (b) I acknowledge that it is contrary to justice, academic integrity, and to the spirit of intellectual inquiry to submit another's statements or ideas as one's own work

Exam instructions (more details on Piazza posts)

- Exam duration: 24 hours, 9am May 29th - 9am May 30th, Chicago time (US Central Time)
- Deadline and how to submit: submit your answers in the provided ANSWER TEMPLATE TXT file THROUGH SVN by 9am on May 30th, Chicago time. All answers must be provided in the [answer.txt] file, following the given template, in the [exam2-answer] directory of the svn. Do NOT submit modified pdf files
- Interpret instructions, registers, and conventions in the exam questions in the context of the x86_64 ISA (unless specified otherwise).
- Closed everything, except that you can use the your own 1-page hand-written cheat sheet (8.5"x11", 2-sided). You may not use any other photocopied or printed notes
- Important note on the "close everything" policy: You need the internet and a computer to receive, view, answer, and submit the exam, and also use Piazza if you'd like to ask a clarification question. The above activities are not against the "close everything" policy. However, you're not allowed to use the internet/computer to help you answer exam questions. This means that, when you use Piazza, do NOT view prior messages; when you use svn, do NOT check contents in the other directories, and so on
- You cannot review course materials for the entire 24 hours even if you plan to take the exam later
- You don't have to complete the exam in one sitting
- For questions related to the exam, please send a PRIVATE post visible to all instructors on Piazza
- If we find an important update to the exam, we will make an announcement on Piazza, and force this message to be sent to your email. Please check Piazza and your email periodically just in case
- As such no further extension will be granted unless there are extenuating circumstances (e.g., sudden family/medical emergencies, etc.). In case of these circumstances please email the instructors ASAP
- Keep the exam confidential. **DO NOT SHARE or DISTRIBUTE**

Question	Topic	Grade	Max	Graded by
1	True or False		/ 40	
2	Fork		/ 12	
3	FD		/ 6	
4	Base and Bound		/ 10	
5	Segmentation		/ 10	
6	Malloc		/ 8	
7	Variable Sharing		/ 6	
8	Data Race		/ 8	
	Total		/ 100	

Last updated: May 29, 2020

Random version number: 444

1 Circle True or False (40 points)

1 True / False Are you readyyyy???

2 True / False A long-running process will prevent other processes (such as a browser) from receiving messages.

3 True / False Exception table contains the addresses of system calls in linux kernel.

4 True / False Because Adobe Reader creates a new process for each opened document, if one document is buggy, it cannot crash other opened documents.

5 True / False The only way CPU enters kernel mode is through system calls

6 True / False When making a system call that takes three arguments, the user program will set only three registers before making the system call.

7 True / False Specific x86 instructions that communicate with the disks, network cards, and the MMU, are catogerized as “privileged” instructions as they can only be run when the CPU is in kernel mode.

8 True / False To print some text on the computer screen, a user program must eventually invoke at least one user/kernel crossing.

9 True / False The OS code and user program share the same stack, so a program can put function arguments in the stack before making a system call.

10 True / False The value of the program counter (%rip) is part of the context of a process, so its value will be saved by the kernel during a context switch.

11 True / False Right after a process A spawns another process B by fork(), both processes share the stack segment in physical memory.

12 True / False When spawning a child process via fork(), the parent process will never see zero as the returned value.

13 True / False File descriptor structures are maintained by the disk hardware.

14 True / False System calls are implemented by hardware designers, not software developers.

15 True / False Sending a signal between two different processes must involve the OS.

16 True / False When receiving a signal, a signal handler always handles it in kernel mode.

17 True / False Linux provides the abstraction of “file” to allow access to different storage devices (disk, SSD, network card, etc).

18 True / False The same file on disk can be associated with multiple file descriptors.

19 True / False A file descriptor table itself includes static information of a file (such as file metadata).

20 True / False Two processes can share a file descriptor (FD) table.

21 True / False After calling call dup2(4,3), FD3 and 4 point at the same file descriptor structure.

22 True / False Each peer threads of the same process has a separate copy of the same file descriptor table.

23 True / False When read() has a shortcount, it is an error and an exception handler must be called.

24 True / False Virtual memory gives the illusion of a memory larger than the physical memory.

25 True / False User program should not directly use physical addresses (for privacy and fault isolation).

26 True / False If a logical address is beyond the bound, MMU will throw a segfault exception.

27 True / False The base and bound in MMU are shared by different processes, so they should not be changed during context switch.

28 True / False In segmentation, using top 3 bits as the segment indexing is sufficient for OSes that want to support up to 15 segments per process.

29 True / False De-referencing a bad/dangling pointer always leads to a segfault.

30 True / False Every thread within a process has its own segment/page table.

31 True / False When searching for an unused block for malloc, the worst-fit policy is generally slower than next-fit policy.

32 True / False Explicit list in malloc library is a list of pointers to all blocks in the heap (allocated and free blocks).

33 True / False A file descriptor that has not been used for a long time will be cleaned up by the OS.

34 True / False Sharing data between two threads in the same process is easier than sharing data between two processes.

35 True / False Race condition/non-determinism will never happen if you don't use threads.

36 True / False Threads can run concurrently with other threads.

37 True / False If a process is running `pthread_join` to wait for a thread, the process *cannot* be terminated before `pthread_join` returns.

38 True / False A variable in a thread's stack *cannot* be modified by the peer threads.

39 True / False Every one-line C instruction (such as i++) is atomic.

40 True / False If critical sections (locks) are not used carefully, multiple concurrent threads can run almost as slowly as a single thread.

Just a gentle note: If you try to "game the system" by answering all the questions above with True (or with False), hoping to get a 50% probability of correct answers, we will give you a zero.

2 Fork, Threads, Sleep, and Wait (12 points)

Consider this program and **please write down below *all possible* outputs that this program might print.**

"Random()" is a random integer generator between 0 and 9. That is, each time it is called, it returns a random integer value 0, 1, 2, . . . , 9.

```
void *func(void *argp){
    sleep(Random()); // randomly generate an integer between 0 and 9
    printf("2");        <----------
    sleep(Random());
    exit(0);
}

int main(){
    pthread_t tid;
    int status, pid;
    printf("1");        <----------
    if ((pid = fork()) == 0){
        sleep(2);
        printf("3");        <----------
        exit(0);
    }
    else {
        pthread_create(&tid, NULL, func, NULL);
        wait(&status);
    }
    sleep(2);
    printf("4");        <----------
    sleep(2);
    exit(0);
}
```

Notes: To make sure you don't miss the `printf()` statements, they are marked with arrows. "sleep(x)" implies the thread will sleep for x seconds.

Some lines can be left blank (*e.g.*, if you believe the program only has 6 possible outputs, then please just fill the first 6 lines and leave the rest blank). *Hint: to answer this correctly, try drawing a time lattice diagram like we did in the class.*

____________ , ____________ , ____________ , ____________ , ____________ , ____________ ,

____________ , ____________ , ____________ , ____________ , ____________ , ____________

3 File Descriptors (6 points)

If the content of file.txt is "abcdef", what will be output of each of the print statements in the following code?

```c
int fd1, fd2; char c1, c2;
fd1 = open("file.txt", O_RDONLY, 0);
fd2 = open("file.txt", O_RDONLY, 0);

read(fd1, &c1, 1);
read(fd1, &c2, 1);
printf("First output: %c,%c\n", c1, c2); // Q1
if (fork() == 0) {
    read(fd1, &c1, 1);
    read(fd2, &c2, 1);
    printf("Second output: %c,%c\n", c1, c2); // Q2
}
else {
    sleep(2);
    dup2(fd1, fd2);
    read(fd2, &c2, 1);
    printf("Third output: %c,%c\n", c1, c2); //Q3
}
```

Q1: *First* output: _______ ,_______

Q2: *Second* output: _______ ,_______

Q3: *Third* output: _______ ,_______

4 Base and Bound (10 points.)

This question concerns internal and external fragmentation in a simple computer with 12MB of memory, using Dynamic Relocation (Base and Bound). Memory is used only with the granularity of megabytes, so for this question we can consider the physical addresses to be 0, 1, . . ., 11. Processes P1, P2, P3 require only 1MB, 2MB, and 3MB, respectively. For a new to-be-run process, a contiguous span of $(P + 1)$MB (not P) will be allocated in the lowest possible addresses, where P is the amount of memory required by the process. Memory at address 3 and 8 are already in use (address ranges 0–2, 4–7, and 9–11 are free). There is no process migration. Consider the question parts below as a sequence (i.e. part **B** assumes that the OS has dealt with the allocation requested for part **A**). Concisely explain your answers; Write "n/a" if there is no meaningful answer.

A. The user wants to run P1. Is there memory for it, and if so, in what address range will P1 be loaded?

Yes No.

From address _____ to address _____

B. The user next wants to run P2. Can P2 be loaded, and if so, in what address range?

Yes No.

From address _____ to address _____

C. The user next wants to run P3. Can P3 be loaded, and if so, in what address range?

Yes No.

From address _____ to address _____

D. After the OS tries to run P1, P2, and P3, what is the internal fragmentation for each of the processes (P1, P2, and P3), and what is the total internal fragmentation?

_____MB.

E. After the OS tries to run P1, P2, and P3, what is the external fragmentation?

_____MB.

5 Segmentation (10 points.)

Consider a computer that only provides **8-bit addressing and 3 segments (2 segment bits)**. The segments are code, heap, and stack. The *code*'s segment number is “0”; the *heap*'s segment number is “1”; the *stack*'s segment number is “2”.

Let's suppose a process P1 is running and the segmentation table of P1 is as follows. **You do not need to worry about bounds in this question.**

Segment	Base	R	W
Stack (2)	0x74	1	1
Heap (1)	0x58	1	1
Code (0)	0x12	1	0

A. Please convert the following memory accesses. Access can be read (r) or write (w). In the last column, please put “S” for success and “E” for error.

```
Access &                          Physical   Success
logical      Base       Offset    address      or
address     (in hex)   (in hex)  (in hex)     Error


w 0x34   =   0x________ + 0x________ = 0x________   (_____)


w 0x92   =   0x________ + 0x________ = 0x________   (_____)
```

B. Given the **architecture** above, what is the largest possible size of a segment?

_____bytes

6 Malloc: Where's My Head and Foot? (8 points)

In class, we discussed *32-bit* malloc library. The library must locate the header and footer locations of each block.

Let's assume there are three blocks in heap (`A, B, C`). Each block is created by a malloc call. The pointer returned by the malloc that creates `A` is `p`, the pointer returned by the malloc that creates `B` is `q`, and the pointer returned by the malloc that creates `C` is `r`. Let's say the size of block **A** is **32** bytes, the size of block **B** is **16** bytes, and the size of block **C** is **24** bytes. Their locations in the memory are as follows (this is illustrative and the block sizes shouldn't be identical):

```
                ---------------- --------------- ---------------
low address |        A        |        B       |       C       | high address
                ---------------- --------------- ---------------
```

Now, the value of pointer `q` is **0x2020**. *Please fill out the following blanks (note the values must be in hexdecimal)*

6.1 The value of pointer `p` is 0x______

6.2 The value of pointer `r` is 0x______

6.3 The address of `A`'s header is 0x______

6.4 The address of `A`'s footer is 0x______

6.5 The address of `B`'s header is 0x______

6.6 The address of `B`'s footer is 0x______

6.7 The address of `C`'s header is 0x______

6.8 The address of `C`'s footer is 0x______

7 Variable Sharing (6 points.)

Consider the following code:

```
#define N 2
void *thread(void *vargp);

char *msgs[N] = {
    "Hello from foo",
    "Hello from bar"
};  /* global variable */

int main()
{
    int i;
    pthread_t tid;

    for (i = 0; i < N; i++)
        pthread_create(& tid, NULL, thread, (void *)&i);
    pthread_exit(NULL);
}

void *thread(void *vargp)
{
    int myid = *((int*)vargp);
    int cnt = 0;
    printf("[%d]: %s (cnt=%d)\n", myid, msgs[myid], ++cnt);
    return NULL;
}
```

Fill each entry in the following table with "Yes" or "No" for the code above, to indicate whether the variable is referenced by different threads. In the first column, the notation `v.t` denotes an instance of variable `v` residing on the local stack for thread `t`, where `t` is either `m` (main thread), `t0` (peer thread 0), or `t1` (peer thread 1).

Variable instance	Referenced by main thread?	Referenced by peer thread 0?	Referenced by peer thread 1?
`cnt.t0`	__________	__________	__________
`i.m`	__________	__________	__________
`msgs`	__________	__________	__________
`myid.t1`	__________	__________	__________

8 Data Race (8 points)

Let's say we want to calculate the product of all odd numbers between 1 and 10 (that is, $1 \times 3 \times 5 \times 7 \times 9 = 945$). Now, consider the following four programs. The first program saves the product in `prod1`, second in `prod2`, third in `prod3`, and fourth in `prod4`. Some of them are buggy (may not produce the right answer). For ease of comparison, the important lines that differentiate these programs are highlighted in bold font.

```
int prod1 = 1;
void *thread1(void *argp){
    prod1 *= *((int *)argp);
}
void main(int argc, char **argv){
    int i = 1;
    while (i <= 10) {
        int *p = malloc(sizeof(int));
        *p = i;
        pthread_t tid;
        if (i % 2 != 0)
            pthread_create(&tid, NULL,
                           thread1, p);
        else
            printf("skipping %d\n", i);
        i++;
    }
    printf("prod1 = %d\n", prod1);
    exit(0);
}
```

```
int prod2 = 1;
void *thread2(void *argp){
    prod2 *= *((int *)argp);
}
void main(int argc, char **argv){
    int i = 1;
    while (i <= 10) {
        pthread_t tid;
        if (i % 2 != 0)
            pthread_create(&tid, NULL,
                           thread2, &i);
        else
            printf("skipping %d\n", i);
        i++;
    }
    printf("prod2 = %d\n", prod2);
    exit(0);
}
```

```
int prod3 = 1;
void *thread3(void *argp){
    prod3 *= (int)argp;
}
void main(int argc, char **argv){
    int i = 1;
    pthread_t tid_arr[10];
    while (i <= 10) {
        pthread_t tid;
        if (i % 2 != 0)
            pthread_create(&tid, NULL,
                           thread3, i);
        else
            printf("skipping %d\n", i);
        tid_arr[i] = tid;
        i++;
    } for (i = 1; i < 10; i++) {
        pthread_join(tid_arr[i], NULL);
    }
    printf("prod3 = %d\n", prod3);
    exit(0);
}
```

```
int prod4 = 1;
pid_t pids[10];
void main(int argc, char **argv){
    int i;
    pid_t pid;
    for (i = 1; i <= 10; i++) {
        if ((pid = fork()) == 0) {
            if (i%2 != 0) {
                prod4 *= i;
            }
            exit(0);
        }
        else {pids[i-1] = pid;}
    }
    i = 0;
    while(waitpid(pids[i++],NULL,0)> 0){}
    printf("prod4 = %d\n", prod4);
    exit(0);
}
```

Based on the possible outcomes (the prod1-prod4 values when each program finishes), please fill out the following table. Each cell should be either "Yes" or "No".

	is always odd	is always 945
prod1		
prod2		
prod3		
prod4		

(Scratch page)

(This is the end of the exam)