1 Circle True or False

IMPORTANT RULES:

I happily provide you these T/F questions to guide you in preparing for the exam. There's a lot of information to digest from the lectures, so these questions guide you on understanding which part is important. You MAY NOT discuss these questions/answers in Piazza, nor ask me the direct answers. That being said, during office hours I welcome discussion leading to the answers (and if the discussion is leading to the right direction, most likely I give you the answers :). You may form study groups to answer these questions together.

Note that these are "systems" questions, which means they are not like "is it true that 1+1=2?" For example, if I ask you "deadlock means all jobs are not progressing," of course the answer is True, but I don't want to get into a debate like "well, it could be false, because 'all' jobs could be any processes running in this world that do not use locks." I hope this is clear – let's not argue on the language semantics. If 80% of your gut says the answer is true, then it's likely true. (Also, sorry for any grammar issues; I prefer to finish this quickly so I can give it to you not too late.

You might also ask me whether it is enough to just study the materials pertaining to these questions below. My answer is: it doesn't hurt to study more materials beyond the questions below. The exam will not be only true/false. There are many fill-in-the-blank questions that require you to think fast. Enjoy and good luck!

1 True / False I've completely read the Final Exam information on Piazza.

2 True / False To handle exceptional flow, hardware support is needed (*e.g.*, exception table).

3 True / False A long-running process will prevent other processes from writing data to disk.

4 True / False The exceptation table contains a list of function names (the exception handlers) in strings.

5 True / False If you only have one CPU core, a user code and the OS code can run at the same time, simultaneously.

6 True / False Regardless how many times your program is interrupted (by the timer interrupt), the correctness of your program will not be altered. (The only side effect your program sees is reduced performance).

7 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 call to the OS.

8 True / False To pass arguments to the OS' syscall handler, the C library puts the arguments to the syscall being called (e.g. write() or read()) in CPU registers and put the syscall number in the rax (eax) register.

9 True / False For shared hardware such as disks, network cards, and monitors that user-mode application cannot access directly, system calls are provided.

10 True / False In a web browser that creates a new process for every tab (e.g. Chrome), a buggy tab cannot

crash other tabs.

11 True / False The only way CPU enters kernel mode is through timer interruptions.

12 True / False System calls (user-kernel crossing) incur the same cost as library calls.

13 True / False The size of the exception table does *not* limit the number of syscalls an OS can provide.

14 True / False We should thank the OS for making our lives simple via abstraction.

15 True / False A program can put function arguments in the stack before making a call to the OS.

16 True / False Register values are part of the context of a process.

17 True / False Right after a process A spawns another process B via fork(), the OS will make an identical copy of A's stack content in B's memory space.

18 True / False After execve() completes successfully, the following line of code after the execve() call will be executed.

19 True / False Sending a signal creates a new thread in the destination process (to handle the signal).

20 True / False A process' main thread will stop executing when its signal handler executes.

21 True / False Sometimes, race condition is fine (*i.e.*, no need to be removed).

22 True / False System calls, signal handlers, and file descriptor tables are all built inside the hardware.

23 True / False When Process A sends an signal to process B, the OS will perform a context switch to give the control to process B and then B will run its signal handler.

24 True / False Processes by default cannot communicate, unless with inter-process communication (IPC).

25 True / False wait() waits for any of the childrens, and waitpid() waits for a specific child.

26 True / False The reason why fork() returns positive value to the parent process is to give the PID (name) of the child that was just created.

27 True / False After the forking, the child process gets a return value of 0, so the child does not know who the parent process is because the child usually doesn't care. But the OS is kind and provides "getppid()" system call if the child wants to know who the parent is (note: this is "getppid" which stands for "get parent pid").

28 True / False Two processes are not allowed to open the same file.

29 True / False Threads of the same process share the same file descriptor table.

30 True / False "close(stdout)" will always return an error.

31 True / False File descriptor structures themselves only describe "open status", but not the static file information (file metadata).

32 True / False In buffering, a larger buffer in general will result in a better performance.

33 True / False I promise I'll play the "fun with file descriptors" game multiple times, including the ones with dup2, dup, lseek, and fork.

34 True / False I promise that in this game, I will *draw a picture* of the situation of the FD tables, their pointers to FD structures, the current offsets, the file content, etc. just like what we did in class. Based on the picture I draw, I will try to answer the question.

35 True / False Every process has its own file descriptor (FD) table.

36 True / False But, after forking, the child process shares the same FD table as its parent.

37 True / False Every open() creates a new FD structure pointed by an unused FD entry with the smallest number.

38 True / False FD structure has "offset" (current position) information, identifying the next starting offset to be read when the user calls read().

39 True / False The first byte of a file is always called "file offset 0".

40 True / False read(N bytes), when successful, will move the current position (offset) by N (offset = offset + N).

41 True / False After dup2(7,13) syscall, the FD structure pointed by FDTable entry #7 will now also be pointed by FDTable entry #13.

42 True / False When calling read() and write(), always check the return values for shortcounts (which may not cause an error) and errors.

43 True / False Buffering is a prevalent concept, it's everywhere, not just in C libraries but also in many applications like Microsoft Office and browsers, and disks, OS, etc.

44 True / False I promise to do all the address-translation examples (logical to physical, plus the error-bound

and r/w-bit checking).

45 True / False I'm ready with hex arithmetic and hex-to-bit conversation and vice versa.

46 True / False Virtual memory gives different processes the illusion that they have the same memory address space and the memory space is larger than the physical memory.

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

48 True / False Separate processes can share the same *data* segment.

49 True / False In dynamic relocation (base and bound/BB), the physical address is the sum of the logical address and the base address (and the logical address must be within the bound).

50 True / False If the logical address is beyond the bound, MMU will throw a segfault exception. This in turn will jump to the OS' segfault_exception_handler() which will kill the segfaulting process.

51 True / False With dynamic relocation (single base-bound pair), code segment cannot be shared.

52 True / False With segmentation, the top bits represent the index to the segment table. That is, the top bits tell us which segment (code/heap/stack) that the logical address is trying to go to.

53 True / False In segmentation, the width of the logical address is the width of the segment bits plus the width of the segment-offset bits.

54 True / False Using top 4 bits as the segment indexing is sufficient for OSes that want to support up to 8 segments per process.

55 True / False I promise I will be careful when chopping the virtual (logical) address. I will read the specification carefully because a wrong chop will lead to wrong answer.

56 True / False In this class, even for stack, I will do stackBase+offSet, not stackBase-offSet (as stated in the OSTEP book). The reason is that this class tries to make my life simpler.

57 True / False In this class, code is in the lowest segment of the process address space and stack is in the highest. This means the most significant bit of logical addresses within the code segment is "1". This also means the most significant bit of logical addresses of your local variables within the stack area is "0".

58 True / False An instruction that dereferences a bad/dangling pointer does not always lead to a segfault-/crash.

59 True / False Segmentation solves external fragmentation.

60 True / False Every thread within a process has its own segment table

61 True / False Sometimes processes need to share data or talk with each other. For this, they should use IPC mechanisms like shared memory (shmat() syscall), pipe, or signals.

62 True / False Today's OSes basically have solved the internal and external fragmentation problem by forcing everyone to use "small, fixed-sized luggages" (i.e. paging).

63 True / False When my Skype is open but I never use it for many days, the moment I click on it, things look slow because the Skype's heap, code, and stack have been "swapped out" to the disk's swap space area by the OS, and they must be "swapped in" back to the memory. This phenomenon is called "page faults" – i.e. the data (page) you want to access are not in the memory but in the disk, and must be brought up to the memory first before your Skype application can continue to run. The analogy is your luggage has been swapped out from the overhead bin (the memory) to the baggage compartment (the disk) and it will take some time for the flight attendant (the OS) to bring your luggage from the baggage compartment to the overhead bin (swap in).

64 True / False The major fragmentation problem that malloc() library faces is external fragmentation ...

65 True / False ... but mostly mallocing small spaces, *e.g.* malloc(1), leads to heavy internal fragmentation.

66 True / False I will remember that the "size" value in the block's header is the *total* size of the header+footer (8 bytes) *and* the payload (and perhaps some paddings). In other words, the "size" represents *the size of the block*, not just the payload.

67 True / False I will memorize the pros/cons of best/worst/first/next-fit policies.

68 True / False In malloc library, merging is all about knowing where your and left/right neighbor's heads and foots are, given only one information "p". From there, you will have all the information (e.g. block sizes n, m1, m2, and the allocation bits).

69 True / False So, given "p", do you know the pointer arithmetics to compute where your and left/right neighbor's heads and foots are?

70 True / False Explicit list in malloc library, is list of direct pointers to all free blocks.

71 True / False When malloc library maintains explicit list, it doesn't need to maintain the implicit list again.

72 True / False In 32-bit explicit list, the smallest block size possible is 16 bytes: header (4 bytes), next (4), prev (4) and footer (4).

73 True / False The "next" pointer contains the address of the first byte of the header of the next free block.

74 True / False The “prev” pointer also contains the address of the first byte of the header of the next free block.

75 True / False I have mastered the “C Pointer Primer” example (the “p” and “pp” example with all the *, **, & prefixes. This knowledge is important for updating the content of the “next” and “prev” memory lines.

76 True / False When understanding concurrency, it’s good to *draw* the lattice of the code execution (to see the happen-before relationship). From the lattice, I can know which actions are concurrent.

77 True / False sleep(N) means sleep for N seconds.

78 True / False In concurrent webserver code, the parent process sets up the file descriptors (the FD table) and simply calls fork(), which by implication makes the child process has its own FD table that inherits the same content (pointers to FD structures) in the parent’s FD table.

79 True / False If an FD structure is pointered by at least file descriptor, the OS will not clean it up.

80 True / False In a long/infinitely-running code, it is important to close() files that you no longer use, otherwise the corresponding FD entries cannot be reused.

81 True / False Thus, in general, memory leak is dangerous in long-running programs (*e.g.*, server), but not in short ones.

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

83 True / False You will get the benefit of parallelism (make your program run faster) by running multiple threads on a single CPU core.

84 True / False A bug in one thread can potentially impact other threads of the same process.

85 True / False A variable in a thread’s stack cannot be modified by the peer threads.

86 True / False i++ is a cute one-line C instruction, I bet it is atomic.

87 True / False A critical section implies instructions that read/write shared data, hence must be protected with locks.

88 True / False Adding too many locks or simply wrapping critical sections with locks without reorganizing your code can cause performance problems (a.k.a. “lock step” performance).

89 True / False When using threads and locks, it’s a good practice to create embarassingly parallel tasks that do not need to synchronize too often. (See the “Parallelizing a job” slides).

90 True / False I have practiced the "Which data is shared?" slides. To answer this type of question, the basic building blocks are:
- Draw, draw, draw, i.e. draw the memory layout, e.g. an integer is a box, a pointer is a box that has an edge pointing to another box.
- Understand that a local variable can have multiple instances (more than one boxes), e.g. when the function is called recursively or the function is called by multiple threads.
- When asked if "ptr" is shared, don't just see who owns "ptr", but rather go to the data being pointed and ask whether the data's memory line can be accessed by which threads.
- When asked if "x" (data) is shared, don't just see who owns "x", but rather checks all the pointers pointing to the x's memory line and check who can access those pointers.

91 True / False Do you know the output of this code below? [In this class, adding sleep(n) with n being predetermined is enough to remove non-determinism. (Although in reality, please use locks or wait calls for proper synchronization).]

```
int main(...)
{
    int a=1, b=2, c=3;
    printf(" A \n");
    pthread_create(&tid1, NULL, tfunc, &a);
    pthread_create(&tid2, NULL, tfunc, &b);
    pthread_create(&tid3, NULL, tfunc, &c);
    printf(" C \n");
    pthread_join(tid1, NULL);
    printf(" D \n");
    sleep(100);
}
void *tfunc (void *argp) {
    int x =  *((int*)argp);
    sleep(x);
    printf(" S-%d \n", x);
    if (x == 2) {
      exit(0);
    }
}
```

92 True / False How about this one?

```
int main(...)
{
    int a=1;
    printf(" A \n");
    pthread_create(&tid1, NULL, tfunc, NULL);
    sleep(1);
    printf(" C \n");
    pthread_join(tid1, NULL);
    printf(" D \n");
    sleep(1);
}
void *tfunc (void *argp) {
    sleep(Random(3)); // Random(n) randomly picks an integer from 0, 1, ..., n-1.
    printf("B \n");
}
```

93 True / False How about this one?

```
int main(...)
{
  printf(" A \n");
  ret = fork();
  if (ret == 0) {
    pthread_create(&tid, NULL, tfunc, NULL);
    sleep(2);
    printf(" B \n");
  }
  else {
    printf(" C \n");
    waitpid(ret, NULL, 0);
    printf(" D \n");
  }
}
void *tfunc (void *argp) {
  sleep(1);
  printf(" E \n");
  exit(0);
}
```

94 True / False In the code below, can you tell the difference among the three ways of passing arguments to a thread?

```
int main(...)
{
    int a=1, b=1, c=1;
    int *p = malloc(sizeof(int));
    *p = c;
    printf(" A \n");
    pthread_create(&tid1, NULL, tfunc1, &a);
    pthread_create(&tid2, NULL, tfunc2, b);
    pthread_create(&tid3, NULL, tfunc3, p);
    printf(" C \n");
    pthread_join(tid1, NULL);
    printf(" D \n");
    sleep(100);
}
void *tfunc1 (void *argp) {
    int x =  *((int*)argp);
    sleep(x);
    printf(" S-%d \n", x);
}
void *tfunc2 (void *argp) {
    int x =  (int)argp;
    sleep(x);
    printf(" S-%d \n", x);
}
void *tfunc3 (void *argp) {
    int x =  *((int*)argp);
    sleep(x);
    printf(" S-%d \n", x);
}
```

95 True / False Now, have you memorized all the concepts above? The exam will require you to answer

fast, and answering fast is possible if you have a firm understanding of the basic OS foundations above.

96 True / False I am happy that I don't have to memorize:
- precise definitions (e.g. critical section, atomicity), as long as I understand general concept,
- lists of system call numbers/functions, signal numbers, exception numbers/functions, etc.,
- every line of the echo server code, the rioBuffer code, the semaphore code , (as long as I understand what the code is trying to achieve),
- signal number and names,
- how paging and page table works,
- every instruction in rio_read()

97 True / False Okay, I think I'm ready for the exam now.