COL331/CSL633 Minor1 Exam
Operating Systems
Sem II, 2016-17

Answer all 6 questions Max. Marks: 30

1. A new compiler 'hcc' is being developed. The developers of 'hcc' would like to ensure that the object files compiled with 'hcc' can be linked with the object files compiled with 'gcc'. However, the developers of 'hcc' would like to implement more optimizations, i.e., produce faster code, than 'gcc'. Answer true/false, with brief explanation:
    a. The code generated by 'hcc' should be identical to the code generated by 'gcc', to be able to link with 'gcc' object files. [2]

False. ---------- 0 marks if Correct True answer

2 marks for Correct Explanation
The code generated by 'hcc' need not to be identical to the code generated by 'gcc'. Header files are included in source code and while linking, gaps created in a file are filled by object code of included files. The function declaration in header file should be matching with the function definition.

    b. 'hcc' should obey the same calling conventions as 'gcc'. If false, explain why. If true, list the 'gcc' calling conventions. [3]

True. ---------- 0 marks for Correct True/False answer

'Gcc' Calling Conventions

- at entry to a function (i.e. just after call): ---------- 1.5 marks
    - %eip points at first instruction of function
    - %esp+4 points at first argument, %esp+8 points at second argument, ...
    - %esp points at return address
    - registers %eax, %edx and %ecx may be trashed

- after ret instruction: ---------- 1.5 marks
    - %eip contains return address
    - %esp points at arguments pushed by caller
    - called function may have trashed arguments

    - %ebp, %ebx, %esi, %edi must contain contents from time of call

2. Among all the system calls that have been discussed in class, list the system calls that are invoked (with their arguments), when you execute the following command: [5]

$ cat a.out | wc

10 system calls --- 0.4 marks for each correct system call. Wrong order will be counted as wrong system call.
1 marks for who has written these optional system calls also

---
pipe(fd_array);
fork();
---
close(1);
dup(fd_array[1]);
close(fd_array[0]); ---------- Optional
close(fd_array[1]); ---------- Optional
exec("cat", a.out);
---
fork();
---
close(0);
dup(fd_array[0]);
close(fd_array[0]); ---------- Optional
close(fd_array[1]); ---------- Optional
exec("wc");
---
wait(0);

Okay if the student also writes the system calls executed by "cat" and "wc":
Cat:
fd = open("a.out")
len = read(fd, buf, sizeof buf)
write(1, buf, len)
len = read(fd, buf, sizeof buf)
write(1, buf, len)
…
…

Wc:
read(0, buf, sizeof buf)
read(0, buf, sizeof buf)
….
…..

write(1, wc_result, ...)

3. Signal handling : A signal handler is specified as a function, e.g.,

```
void sigint_handler(int signum);
```

When a signal is “delivered” to a process, this function gets executed, as an “asynchronous function call”. Do all the registers need to be saved before transferring control to the signal handler function? Do only caller-save registers need to be saved? Or only callee-save registers need to be saved? Where should these registers be saved? After, the signal handler has finished executing, what should happen? [5]

- When a signal is delivered, the process is interrupted, and all the registers are saved. Saving only caller-save or callee-save registers is not possible, because the interrupted function could be in the middle of an execution, and may be using all registers. The operating system cannot assume that the program has been compiled using a certain compiler
    - If the OS was sure that the process has been compiled using a gcc (for example), then it would only need to save the caller-save registers.
    - Full marks if correct answer. Full marks if clearly specify that gcc is the compiler that is assumed for all processes, and mention that only caller-save registers need to be saved.
- The registers can be saved in the process address space. One convenient location to store them, is through pushing them to the process stack. Notice that if the process stack is malformed, the process will crash, without affecting the correctness of the whole system
- After the signal handler has finished executing, all the saved registers need to be restored before resuming execution at the interrupted user instruction. Here is one way to implement this:

```
Signal_handling_stub:
    push %eax
    push %ecx
    ….  (push all registers)
    call signal_handler
    …. (pop all registers)
    pop %ecx
    pop %eax
    ret
```

To deliver the signal to the process, the OS emulates an asynchronous function call (without saving any registers) to “signal_handling_stub”. The signal_handling_stub can save/restore all registers before transferring control to the actual “signal_handler”.

2 marks for saying that “all registers need to be saved” with proper reason. 0 marks if reason not correct.

1.5 marks for saying that the registers need to be saved/restored from process address space.

0.5 marks for saying that the registers can be saved to user stack

1 mark for saying, how this can be implemented exactly (e.g., through pseudo-code, or through explanation).

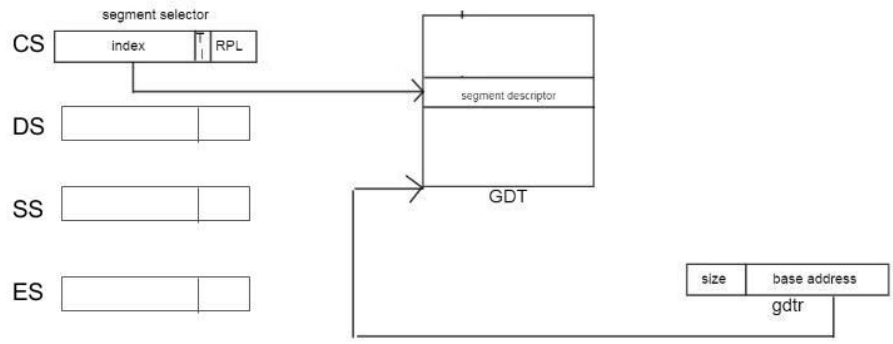
4. Segmentation
   a. What is segmentation, and why is it useful? [1]
   b. How does GDT help in implementing segmentation? [2]
   c. How is GDT protected from untrusted user processes? [3]

a. Segmentation is division of single Physical memory into multiple segments, so that private address space for different processes can co exist in memory in case of a multiprocessing system. It uses a base/limit mechanism, with hardware support. [deduct 0.5 marks if do not mention base/limit mechanism].

b. ----------1 mark for Diagram, 1 mark for explanation

Global descriptor Table stores the valid segment values and CS, DS(Segment Registers/Selectors) etc store the index for the GDT.
Each entry in GDT store Base, limit and each memory access is dereferenced using Segment selector as follows:

Pa = va + base, assert(va ≤ limit)



c. ----------1.5 marks for each point

Base address for GDT in memory is stored in GDTR and instruction to load GDTR i.e. lgdt is a privileged instruction and user processes cannot execute it and hence modify GDT. [1 mark]

GDTR points to a structure containing gdtstruct = (gdtbase, gdtlimit). Intersection(gdtstruct, user application's address space) = empty [1 mark]

Also intersection(GDT, User application's address space) = empty [1 mark]

5. Firefox uses a thread per tab, but Chrome uses a process per tab. What are the trade-offs of the two different approaches? [5]

Firefox = Kernel level thread per Tab [if wrong, deduct 2 marks]
Chrome = Process per Tab

Trade offs:

Advantages of Firefox over Chrome:

- Context switch is much faster
- Sharing of data(e.g. URLs) among various tabs. In case of Firefox shared address space, so no specific interprocess communication required. For chrome, required.
- Creation of thread is much faster and lightweight than process.

Disadvantages of Firefox over Chrome:

- If 1 tab in Firefox does some malicious activity (e.g., due to a security exploit), all threads(tabs) are affected.

---------- If mention all four points, full marks. Deduct 1 mark for each missing argument. If extra arguments are given, that are incorrect, deduct 1.5 marks for each incorrect extra argument.

6. What happens if the parent dies before the child exits? Is the child doomed to become a ZOMBIE forever? How can the 'init' process help; write pseudo-code for the init process to handle such orphaned child processes. [4]

- When exit is called in Child, then also OS will not clear the complete information of the child process, as exit code needs to be collected by Parent. If the parent dies before the child exits, then the child process will become orphan process. No, Child won't become a ZOMBIE forever. All the orphan processes will become children of init process. [2 marks]

- How can the 'init' process help: 'init' will keep on calling wait() after a regular interval to clear the zombie/orphan processes. pseudo-code for the init process to handle such orphaned child processes. [2 marks]

Option 1:

```
Int main() {
        while(1)
        {
                wait(0);
        }
}
```

Option 2: Install a sigchild handler.

```
Int main() {
  signal(SIGCHLD, sigchld_handler);
}

Void sigchld_handler(int signum) {
  wait(0);
}
```