Lectures on Operating Systems (Mythili Vutukuru, IIT Bombay)

Practice Problems: Process Management in xv6

1. Consider the following lines of code in a program running on xv6.

```
int ret = fork();
if(ret==0) { //do something in child}
else { //do something in parent}
```

   (a) When a new child process is created as part of handling fork, what does the kernel stack of the new child process contain, after fork finishes creating it, but just before the CPU switches away from the parent?

   (b) How is the kernel stack of the newly created child process different from that of the parent?

   (c) The EIP value that is present in the trap frames of the parent and child processes decides where both the processes resume execution in user mode. Do both the EIP pointers in the parent and child contain the same logical address? Do they point to the same physical address in memory (after address translation by page tables)? Explain.

   (d) How would your answer to (c) above change if xv6 implemented copy-on-write during fork?

   (e) When the child process is scheduled for the first time, where does it start execution in kernel mode? List the steps until it finally gets to executing the instruction after fork in the program above in user mode.

   Ans:

   (a) It contains a trap frame, followed by the context structure.

   (b) The parent's kernel stack has only a trap frame, since it is still running and has not been context switched out. Further, the value of the EAX register in the two trap frames is different, to return different values in parent and child.

   (c) The EIP value points to the same logical address, but to different physical addresses, as the parent and child have different memory images.

   (d) With copy-on-write, the physical addresses may be the same as well, as long as both parent and child have not modified anything.

   (e) Starts at forkret, followed by trapret. Pops the trapframe and starts executing at the instruction right after fork.

2. Suppose a machine (architecture: x86, single core) has two runnable processes P1 and P2. P1 executes a line of code to read 1KB of data from an open file on disk to a buffer in its memory. The content requested is not available in the disk buffer cache and must be fetched from disk. Describe what happens from the time the instruction to read data is started in P1, to the time it completes (causing the process to move on to the next instruction in the program), by answering the following questions.

(a) The code to read data from disk will result in a system call, and will cause the x86 CPU to execute the `int` instruction. Briefly describe what the CPU's `int` instruction does.

(b) The `int` instruction will then call the kernel's code to handle the system call. Briefly describe the actions executed by the OS interrupt/trap/system call handling code before the read system call causes P1 to block.

(c) Now, because process P1 has made a blocking system call, the CPU scheduler context switches to some other process, say P2. Now, the data from the disk that unblocks P1 is ready, and the disk controller raises an interrupt while P2 is running. On whose kernel stack does this interrupt processing run?

(d) Describe the contents of the kernel stacks of P1 and P2 when this interrupt is being processed.

(e) Describe the actions performed by P2 in kernel mode when servicing this disk interrupt.

(f) Right after the disk interrupt handler has successfully serviced the interrupt above, and before any further calls to the scheduler to context switch from P2, what is the state of process P1?

Ans:

(a) The CPU switches to kernel mode, switches to the kernel stack of the process, and pushes some registers like the EIP onto the kernel stack.

(b) The kernel pushes a few other registers, updates segment registers, and starts executing the system call code, which eventually causes P1 to block.

(c) P2's kernel stack

(d) P2's kernel stack has a trapframe (since it switched to kernel mode). P1's kernel stack has both a context structure and a trap frame (since it is currently context switched out).

(e) P2 saves user context, switches to kernel mode, services the disk interrupt that unblocks P1, marks P1 as ready, and resumes its execution in userspace.

(f) Ready / runnable.

3. Consider a newly created process in xv6. Below are the several points in the code that the new process passes through before it ends up executing the line just after the fork statement in its user space. The EIP of each of these code locations exists on the kernel stack when the process is scheduled for the first time. For each of these locations below, precisely explain where on the kernel stack the corresponding EIP is stored (in which data structure etc.), and when (as part of which function or stage of process creation) is the EIP written there. Be as specific as possible in your answers.

(a) `forkret`

(b) `trapret`

(c) just after the `fork()` system call in userspace

Ans:

(a) EIP of forkret is stored in struct context by allocproc.

(b) EIP of trapret is stored on kernel stack by allocproc.

(c) EIP of fork system call code is stored in trapframe in parent, and copied to child's kernel stack in the fork function.

4. Consider a process that has performed a blocking disk read, and has been context switched out in xv6. Now, when the disk interrupt occurs with the data requested by the process, the process is unblocked and context switched in immediately, as part of handling the interrupt. [T/F]

   Ans: F

5. In xv6, state the system call(s) that result in new `struct proc` objects being allocated.

   Ans: fork

6. Give an example of a scenario in which the xv6 dispatcher / `swtch` function does NOT use a `struct context` created by itself previously, but instead uses an artificially hand-crafted `struct context`, for the purpose of restoring context during a context switch.

   Ans: When running process for first time (say, after fork).

7. Give an example of a scenario in xv6 where a `struct context` is stored on the kernel stack of a process, but this context is never used or restored at any point in the future.

   Ans: When process has finished after exit, its saved context is never restored.

8. Consider a parent process P that has executed a fork system call to spawn a child process C. Suppose that P has just finished executing the system call code, but has not yet returned to user mode. Also assume that the scheduler is still executing P and has not context switched it out. Below are listed several pieces of state pertaining to a process in xv6. For each item below, answer if the state is identical in both processes P and C. Answer Yes (if identical) or No (if different) for each question.

   (a) Contents of the PCB (struct proc). That is, are the PCBs of P and C identical? (Yes/No)

   (b) Contents of the memory image (code, data, heap, user stack etc.).

   (c) Contents of the page table stored in the PCB.

   (d) Contents of the kernel stack.

   (e) EIP value in the trap frame.

   (f) EAX register value in the trap frame.

   (g) The physical memory address corresponding to the EIP in the trap frame.

   (h) The files pointed at by the file descriptor table. That is, are the file structures pointed at by any given file descriptor identical in both P and C?

   Ans:

   (a) No

   (b) Yes

   (c) No

   (d) No

   (e) Yes

(f) No

(g) No

(h) Yes

9. Suppose the kernel has just created the first user space "init" process, but has not yet scheduled it. Answer the following questions.

   (a) What does the EIP in the trap frame on the kernel stack of the process point to?

   (b) What does the EIP in the context structure on the kernel stack (that is popped when the process is context switched in) point to?

   Ans:

   (a) address 0 (first line of code in init user code)

   (b) forkret / trapret

10. Consider a process P that forks a child process C in xv6. Compare the trap frames on the kernel stacks of P and C just after the fork system call completes execution, and before P returns back to user mode. State one difference between the two trap frames at this instant. Be specific in your answer and state the exact field/register value that is different.

    Ans: EAX register has different value.

11. In xv6, the EIP within the `struct context` on the kernel stack of a process usually points to the `swtch` statement in the `sched` function, where the process gives up its CPU and switches to the scheduler thread during a context switch. Which processes are an exception to this statement? That is, for which processes does the EIP on the context structure point to some other piece of code?

    Ans: Newly created processes / processes running for first time

12. When an trap occurs in xv6, and a process shifts from user mode to kernel mode, which entity switches the CPU stack pointer from pointing to the user stack of the running program to its kernel stack? Tick one: x86 hardware instruction / xv6 assembly code

    Ans: x86 hardware

13. Consider a process P in xv6, which makes a system call, goes to kernel mode, runs the system call code, and comes back into user mode again. The value of the EAX register is preserved across this transition. That is, the value of the EAX register just before the process started the system call will always be equal to its value just after the process has returned back to user mode. [T/F]

    Ans: False, EAX is used to store system call number and return value, so it changes.

14. When a trap causes a process to shift from user mode to kernel mode in xv6, which CPU data registers are stored in the trapframe (on the kernel stack) of the process? Tick one: all registers / only callee-save registers

    Ans: All registers

15. When a process in xv6 wishes to pass one or more arguments to the system call, where are these arguments initially stored, before the process initiates a jump into kernel mode? Tick one: user stack / kernel stack

    Ans: User stack, as user program cannot access kernel stack

16. Consider the context switch of a CPU from the context of process P1 to that of process P2 in xv6. Consider the following two events in the chronological order of the events during the context switch: (E1) the ESP (stack pointer) shifts from pointing to the kernel stack of P1 to the kernel stack of P2; (E2) the EIP (program counter) shifts from pointing to an address in the memory allocated to P1 to an address in the memory allocated to P2. Which of the following statements is/are true regarding the relative ordering of events E1 and E2?

    (a) E1 occurs before E2.

    (b) E2 occurs before E1.

    (c) E1 and E2 occur simultaneously via an atomic hardware instruction.

    (d) The relative ordering of E1 and E2 can vary from one context switch to the other.

    Ans: (a)

17. Consider the following actions that happen during a context switch from thread/process P1 to thread/process P2 in xv6. (One of P1 or P2 could be the scheduler thread as well.) Arrange the actions below in chronological order, from earliest to latest.

    (A) Switch ESP from kernel stack of P1 to that of P2

    (B) Pop the callee-save registers from the kernel stack of P2

    (C) Push the callee-save registers onto the kernel stack of P1

    (D) Push the EIP where execution of P1 stops onto the kernel stack of P1.

    Ans: DCAB

18. Consider a process P in xv6 that invokes the wait system call. Which of the following statements is/are true?

    (a) If P does not have any zombie children, then the wait system call returns immediately.

    (b) The wait system call always blocks process P and leads to a context switch.

    (c) If P has exactly one child process, and that child has not yet terminated, then the wait system call will cause process P to block.

    (d) If P has two or more zombie children, then the wait system call reaps all the zombie children of P and returns immediately.

    Ans: (c)

19. Consider a process P in xv6 that executes the exec system call successfully. Which of the following statements is/are true?

    (a) The exec system call changes the PID of process P.

    (b) The exec system call allocates a new page table for process P.

(c) The exec system call allocates a new kernel stack for process P.

(d) The exec system call changes one or more fields in the trap frame on the kernel stack of process P.

Ans: (b), (d)

20. Consider a process P in xv6 that executes the exec system call successfully. Which of the following statements is/are true?

    (a) The arguments to the exec system call are first placed on the user stack by the user code.

    (b) The arguments to the exec system call are first placed on the kernel stack by the user code.

    (c) The arguments (argc, argv) to the new executable are placed on the kernel stack by the exec system call code.

    (d) The arguments (argc, argv) to the new executable are placed on the user stack by the exec system call code.

    Ans: (a), (d)

21. Consider a newly created child process C in xv6 that is scheduled for the first time. At the point when the scheduler is just about to context switch into C, which of the following statements is/are true about the kernel stack of process C?

    (a) The top of the kernel stack contains the context structure, whose EIP points to the instruction right after the fork system call in user code.

    (b) The bottom of the kernel stack has the trapframe, whose EIP points to the forkret function in OS code.

    (c) The top of the kernel stack contains the context structure, whose EIP points to the forkret function in OS code.

    (d) The bottom of the kernel stack contains the trap frame, whose EIP points to the trapret function in OS code.

    Ans: (c)

22. Consider a trapframe stored on the kernel stack of a process P in xv6 that jumped from user mode to kernel mode due to a trap. Which of the following statements is/are true?

    (a) All fields of the trapframe are pushed onto the kernel stack by the OS code.

    (b) All fields of the trapframe are pushed onto the kernel stack by the x86 hardware.

    (c) The ESP value stored in the trapframe points to the top of the kernel stack of the process.

    (d) The ESP value stored in the trapframe points to the top of the user stack of the process.

    Ans: (d)

23. Consider a process P that has made a blocking disk read in xv6. The OS has issued a disk read command to the disk hardware, and has context switched away from P. Which of the following statements is/are true?

(a) The top of the kernel stack of P contains the return address, which is the value of EIP pointing to the user code after the read system call.

(b) The bottom of the kernel stack of P contains the trapframe, whose EIP points to the user code after the read system call.

(c) The top of the kernel stack of P contains the context structure, whose EIP points to the user code after the read system call.

(d) The CPU scheduler does not run P again until after the disk interrupt that unblocks P is raised by the device hardware.

Ans: (b), (d)

24. In the implementation of which of the following system calls in xv6 are new ptable entries allocated or old ptable entries released (marked as unused)?

(a) fork

(b) exit

(c) exec

(d) wait

Ans: (a), (d)

25. A process has invoked exit() in xv6. The CPU has completed executing the OS code corresponding to the exit system call, and is just about to invoke the swtch() function to switch from the terminated process to the scheduler thread. Which of the following statements is/are true?

(a) The stack pointer ESP is pointing to some location within the kernel stack of the terminated process

(b) The MMU is using the page table of the terminated process

(c) The state of the terminated process in the ptable is RUNNING

(d) The state of the terminated process in the ptable is ZOMBIE

Ans: (a), (b), (d)

26. Answer the following questions about the context switching code in xv6.

(a) The `swtch` function takes two arguments: the address of the context pointer of the old process, say `struct context **oldc`, and a pointer to the context of the new process, say, `struct context *newc`. After the registers are pushed on the stack of the old process, two steps happen: the old context pointer is updated with the latest value of `esp`, and the stack pointer shifts to the context of the new process. Write down the code for these two steps using C-like syntax. Your answer must have two assignment statements using the variables `oldc` and `newc` declared above, and the stack pointer `esp`.

Ans: `*oldc = esp; esp = newc;`

(b) Why does the `swtch` function push and pop only some registers, and not all registers, while saving and restoring context?

Ans: Only callee-save registers need to be saved inside the function, as caller-save registers would have been saved before entering the function code, by the caller.

(c) During the `swtch` function, does the `esp` contain addresses from high virtual address space (belonging to the kernel) or from the low address space?

Ans: high virtual address

(d) A process in xv6 can invoke the `sched` function (which in turn calls `swtch`) from 3 different places. Two of these are from the `sleep` function which blocks a process, and the `exit` system call which terminates the process. What is the third place?

Ans: yield (timer interrupt)

(e) A process in xv6 can return from `swtch` and resume execution in three different places in the kernel code. One of these is the `forkret` function where new processes start. List the other two.

Ans: yield and sleep