Lecture Notes on Operating Systems

Practice Problems: Process Management in xv6

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

```c
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. Vague answers like “during process creation” will not get credit.
(a) forkret
(b) trapret
(c) just after the fork() statement 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. Consider a process in xv6 that has been context-switched out because of making a blocking
   read system call. The EIP of the user program instruction that made the system call is stored
   in the struct context on the kernel stack. [T/F]
   
   Ans: F

6. In xv6, state the system call(s) that result in new struct proc objects being allocated.
   
   Ans: fork

7. 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).

8. 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.

9. 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.
10. 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

11. 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.

12. 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
13. 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

14. 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.

15. 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

16. 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