# Threads, processes, and context switching

Required reading: proc.c (focus on scheduler() and sched()), setjmp.S, and sys_fork (in sysproc.c)

## Overview

Big picture: more programs than processors. How to share the limited number of processors among the programs?

Observation: most programs don't need the processor continuously, because they frequently have to wait for input (from user, disk, network, etc.)

Idea: when one program must wait, it releases the processor, and gives it to another program.

Mechanism: thread of computation, an active active computation. A thread is an abstraction that contains the minimal state that is necessary to stop an active and an resume it at some point later. What that state is depends on the processor. On x86, it is the processor registers (see setjmp.S).

Address spaces and threads: address spaces and threads are in principle independent concepts. One can switch from one thread to another thread in the same address space, or one can switch from one thread to another thread in another address space. Example: in xv6, one switches address spaces by switching segmentation registers (see setupsegs). Does xv6 ever switch from one thread to another in the same address space? (Answer: yes, v6 switches, for example, from the scheduler, proc[0], to the kernel part of init, proc[1].) In the JOS kernel we switch from the kernel thread to a user thread, but we don't switch kernel space necessarily.

Process: one address space plus one or more threads of computation. In xv6 all *user* programs contain one thread of computation and one address space, and the concepts of address space and threads of computation are not separated but bundled together in the concept of a process. When switching from the kernel program (which has multiple threads) to a user program, xv6 switches threads (switching from a kernel stack to a user stack) and address spaces (the hardware uses the kernel segment registers and the user segment registers).

xv6 supports the following operations on processes:

- fork; create a new process, which is a copy of the parent.
- exec; execute a program
- exit: terminte process
- wait: wait for a process to terminate
- kill: kill process

- sbrk: grow the address space of a process.

This interfaces doesn't separate threads and address spaces. For example, with this interface one cannot create additional threads in the same threads. Modern Unixes provides additional primitives (called pthreads, POSIX threads) to create additional threads in a process and coordinate their activities.

Scheduling. The thread manager needs a method for deciding which thread to run if multiple threads are runnable. The xv6 policy is to run the processes round robin. Why round robin? What other methods can you imagine?

Preemptive scheduling. To force a thread to release the processor periodically (in case the thread never calls sleep), a thread manager can use preemptive scheduling. The thread manager uses the clock chip to generate periodically a hardware interrupt, which will cause control to transfer to the thread manager, which then can decide to run another thread (e.g., see trap.c).

# xv6 code examples

Thread switching is implemented in xv6 using setjmp and longjmp, which take a jumpbuf as an argument. setjmp saves its context in a jumpbuf for later use by longjmp. longjmp restores the context saved by the last setjmp. It then causes execution to continue as if the call of setjmp has just returned 1.

- setjmp saves: ebx, exc, edx, esi, edi, esp, ebp, and eip.
- longjmp restores them, and puts 1 in eax!

Example of thread switching: proc[0] switches to scheduler:

- 1359: proc[0] calls iget, which calls sleep, which calls sched.
- 2261: The stack before the call to setjmp in sched is:
- `CPU 0:`
- `eax: 0x10a144   1089860`
- `ecx: 0x6c65746e 1818588270`
- `edx: 0x0        0`
- `ebx: 0x10a0e0   1089760`
- `esp: 0x210ea8   2166440`
- `ebp: 0x210ebc   2166460`
- `esi: 0x107f20   1081120`
- `edi: 0x107740   1079104`
- `eip: 0x1023c9`
- `eflags 0x12`
- `cs:  0x8`
- `ss:  0x10`
- `ds:  0x10`
- `es:  0x10`
- `fs:  0x10`

- gs:  0x10
- 00210ea8 [00210ea8]  10111e
- 00210eac [00210eac]  210ebc
- 00210eb0 [00210eb0]  10239e
- 00210eb4 [00210eb4]  0001
- 00210eb8 [00210eb8]  10a0e0
- 00210ebc [00210ebc]  210edc
- 00210ec0 [00210ec0]  1024ce
- 00210ec4 [00210ec4]  1010101
- 00210ec8 [00210ec8]  1010101
- 00210ecc [00210ecc]  1010101
- 00210ed0 [00210ed0]  107740
- 00210ed4 [00210ed4]  0001
- 00210ed8 [00210ed8]  10cd74
- 00210edc [00210edc]  210f1c
- 00210ee0 [00210ee0]  100bbc
- 00210ee4 [00210ee4]  107740
- 2517: stack at beginning of setjmp:
- CPU 0:
- eax: 0x10a144   1089860
- ecx: 0x6c65746e 1818588270
- edx: 0x0        0
- ebx: 0x10a0e0   1089760
- esp: 0x210ea0   2166432
- ebp: 0x210ebc   2166460
- esi: 0x107f20   1081120
- edi: 0x107740   1079104
- eip: 0x102848
- eflags 0x12
- cs:  0x8
- ss:  0x10
- ds:  0x10
- es:  0x10
- fs:  0x10
- gs:  0x10
- 00210ea0 [00210ea0]  1023cf   <--- return address (sched)
- 00210ea4 [00210ea4]  10a144
- 00210ea8 [00210ea8]  10111e
- 00210eac [00210eac]  210ebc
- 00210eb0 [00210eb0]  10239e
- 00210eb4 [00210eb4]  0001
- 00210eb8 [00210eb8]  10a0e0
- 00210ebc [00210ebc]  210edc
- 00210ec0 [00210ec0]  1024ce
- 00210ec4 [00210ec4]  1010101
- 00210ec8 [00210ec8]  1010101
- 00210ecc [00210ecc]  1010101
- 00210ed0 [00210ed0]  107740
- 00210ed4 [00210ed4]  0001

- `00210ed8 [00210ed8]  10cd74`
- `00210edc [00210edc]  210f1c`
- 2519: What is saved in jmpbuf of proc[0]?
- 2529: return 0!
- 2534: What is in jmpbuf of cpu 0? The stack is as follows:
- `CPU 0:`
- `eax: 0x0        0`
- `ecx: 0x6c65746e 1818588270`
- `edx: 0x108aa4   1084068`
- `ebx: 0x10a0e0   1089760`
- `esp: 0x210ea0   2166432`
- `ebp: 0x210ebc   2166460`
- `esi: 0x107f20   1081120`
- `edi: 0x107740   1079104`
- `eip: 0x10286e`
- `eflags 0x46`
- `cs:  0x8`
- `ss:  0x10`
- `ds:  0x10`
- `es:  0x10`
- `fs:  0x10`
- `gs:  0x10`
- `00210ea0 [00210ea0]  1023fe`
- `00210ea4 [00210ea4]  108aa4`
- `00210ea8 [00210ea8]  10111e`
- `00210eac [00210eac]  210ebc`
- `00210eb0 [00210eb0]  10239e`
- `00210eb4 [00210eb4]  0001`
- `00210eb8 [00210eb8]  10a0e0`
- `00210ebc [00210ebc]  210edc`
- `00210ec0 [00210ec0]  1024ce`
- `00210ec4 [00210ec4]  1010101`
- `00210ec8 [00210ec8]  1010101`
- `00210ecc [00210ecc]  1010101`
- `00210ed0 [00210ed0]  107740`
- `00210ed4 [00210ed4]  0001`
- `00210ed8 [00210ed8]  10cd74`
- `00210edc [00210edc]  210f1c`
- 2547: return 1! stack looks as follows:
- `CPU 0:`
- `eax: 0x1        1`
- `ecx: 0x108aa0   1084064`
- `edx: 0x108aa4   1084068`
- `ebx: 0x10074    65652`
- `esp: 0x108d40   1084736`
- `ebp: 0x108d5c   1084764`
- `esi: 0x10074    65652`
- `edi: 0xffde     65502`
- `eip: 0x102892`

- ` eflags 0x6`
- ` cs:  0x8`
- ` ss:  0x10`
- ` ds:  0x10`
- ` es:  0x10`
- ` fs:  0x10`
- ` gs:  0x10`
- `    00108d40 [00108d40]  10231c`
- `    00108d44 [00108d44]  10a144`
- `    00108d48 [00108d48]  0010`
- `    00108d4c [00108d4c]  0021`
- `    00108d50 [00108d50]  0000`
- `    00108d54 [00108d54]  0000`
- `    00108d58 [00108d58]  10a0e0`
- `    00108d5c [00108d5c]  0000`
- `    00108d60 [00108d60]  0001`
- `    00108d64 [00108d64]  0000`
- `    00108d68 [00108d68]  0000`
- `    00108d6c [00108d6c]  0000`
- `    00108d70 [00108d70]  0000`
- `    00108d74 [00108d74]  0000`
- `    00108d78 [00108d78]  0000`
- `    00108d7c [00108d7c]  0000`
- 2548: where will longjmp return? (answer: 10231c, in scheduler)
- 2233:Scheduler on each processor selects in a round-robin fashion the first runnable process. Which process will that be? (If we are running with one processor.) (Ans: proc[0].)
- 2229: what will be saved in cpu's jmpbuf?
- What is in proc[0]'s jmpbuf?
- 2548: return 1. Stack looks as follows:
- ` CPU 0:`
- ` eax: 0x1        1`
- ` ecx: 0x6c65746e 1818588270`
- ` edx: 0x0        0`
- ` ebx: 0x10a0e0   1089760`
- ` esp: 0x210ea0   2166432`
- ` ebp: 0x210ebc   2166460`
- ` esi: 0x107f20   1081120`
- ` edi: 0x107740   1079104`
- ` eip: 0x102892`
- ` eflags 0x2`
- ` cs:  0x8`
- ` ss:  0x10`
- ` ds:  0x10`
- ` es:  0x10`
- ` fs:  0x10`
- ` gs:  0x10`
- `    00210ea0 [00210ea0]  1023cf   <--- return to sleep`
- `    00210ea4 [00210ea4]  108aa4`

- 00210ea8 [00210ea8]  10111e
- 00210eac [00210eac]  210ebc
- 00210eb0 [00210eb0]  10239e
- 00210eb4 [00210eb4]  0001
- 00210eb8 [00210eb8]  10a0e0
- 00210ebc [00210ebc]  210edc
- 00210ec0 [00210ec0]  1024ce
- 00210ec4 [00210ec4]  1010101
- 00210ec8 [00210ec8]  1010101
- 00210ecc [00210ecc]  1010101
- 00210ed0 [00210ed0]  107740
- 00210ed4 [00210ed4]  0001
- 00210ed8 [00210ed8]  10cd74
- 00210edc [00210edc]  210f1c

Why switch from proc[0] to the processor stack, and then to proc[0]'s stack? Why not instead run the scheduler on the kernel stack of the last process that run on that cpu?

- If the scheduler wanted to use the process stack, then it couldn't have any stack variables live across process scheduling, since they'd be different depending on which process just stopped running.
- Suppose process p goes to sleep on CPU1, so CPU1 is idling in scheduler() on p's stack. Someone wakes up p. CPU2 decides to run p. Now p is running on its stack, and CPU1 is also running on the same stack. They will likely scribble on each others' local variables, return pointers, etc.
- The same thing happens if CPU1 tries to reuse the process's page tables to avoid a TLB flush. If the process gets killed and cleaned up by the other CPU, now the page tables are wrong. I think some OSes actually do this (with appropriate ref counting).

How is preemptive scheduling implemented in xv6? Answer see trap.c line 2905 through 2917, and the implementation of yield() on sheet 22.

How long is a timeslice for a user process? (possibly very short; very important lock is held across context switch!)