xv6 kill

(sheet 26)

sets p->killed for the pid after taking `ptable.lock`.

What does setting p->killed do? Nothing, for now. But the process must be in the middle of something when this was done --- e.g., running in user mode, running in kernel mode, sleeping, etc. Whenever the process reaches a *safe* boundary, `p->killed` will be checked and `exit()` would be called. Notice that if the process was sleeping, kill() marks it RUNNABLE.

Where is killed checked? At syscall entry and exit (see trap function at lines 3104 and 3108). At other places where the process may have woken up from a sleep, and it is no longer correct to go back to sleep again.

Why not destroy the process right away? It is in the middle of something, destroying it will leave the kernel in an inconsistent state.

Why is it safe to mark the process RUNNABLE if it was SLEEPING? The programmer has ensured that the waiting loops that call sleep() within them, also check `p->killed` wherever necessary. e.g., the `wait()` functions waiting loop checks `proc->killed`.

Wouldn't it be okay to let the sleeping process keep sleeping, even if it was killed? Won't it anyways exit whenever it wakes up? This is true, and if you expect the process to wakeup in some bounded amount of time, then this may be okay. However, if the process could wait for an unbounded amount of time (e.g., parent waiting on child to exit), then this will not be acceptable. In general, all sleeping processes are woken up. Assuming that all calls to `sleep` are nested inside a `while` loop that checks the condition, it is possible that some of these processes go back to sleep again (where the sleep time is bounded anyways).

For example, the sleep loop inside `wait()` checks `p->killed` (because this sleep loop can wait for an unbounded time). However, the sleep loop inside the IDE device driver `iderw` does not check `p->killed`, which means that if it is woken up by `kill()`, it will go back to sleep again till the IDE request is not completed (see following discussion for more clarity on this).

ide device driver

(sheet 39)

The `iderw` function is called with an argument of type `struct buf *`, which represents a buffer in memory that will store the contents of the disk block. This function is used to read/write from/to the disk which has an IDE interface. The API for `iderw` specifies that if the B_DIRTY flag is set in `buf`, then the buffer needs to be written to the disk. Similarly, if the `B_VALID` flag is not set in `buf`, then the buffer needs to be read from the disk.

Multiple processes may be trying to access the disk through system calls (e.g., read/write). Also, disk requests could be made due to virtual memory's demand paging logic. All these accesses to the disk need to be mutually exclusive. So, xv6 uses a lock, called `idelock`.

Also, as we know, disk accesses are likely to be very slow. So, it is quite likely that multiple processes are simultaneously waiting for the disk. The right thing to do is to make the processes sleep, while they are waiting for the disk. However, before they go to sleep, they must register their request. These outstanding requests are maintained in a FIFO list, called `idequeue`.

The disk driver processes one request at a time, starting from the head of the list.

Let's look at how the IDE driver uses sleep and wakeup. ide_rw() starts a disk operation and calls sleep(). ide_intr() calls wakeup() when disk interrupts to say that it's done.

Why does it work? What if the disk finishes and interrupts just before ide_rw() calls sleep()? What if the disk interrupts during the execution of the interrupt handler?

What's the sleep channel convention? Why does it make sense?

Let's look at the IDE disk driver as an example of xv6's use of locks. Sheet 39. The file system kernel code calls ide_rw() to read or write a block of data from/to the disk. ide_rw() should return when the read or write has completed. There may be multiple calls to ide_rw() outstanding because multiple processes may be reading files at the same time. The disk can only perform one operation at a time, so the IDE driver keeps a queue of requests that are waiting their turn.

The IDE driver has just one lock (ide_lock). This one lock helps enforce multiple invariants required for correct operation:

- Only one thread should be inserting or deleting from ide_queue at a time.
- Only one thread should be commanding the IDE hardware (via inb/outb instructions) at a time.
- The disk hardware can only execute one read or write at a time.

The software (and hardware) for these activities was all designed for one-at-a-time use, which is why it needs locking.

What happens if two processes call ide_rw() at the same time from different CPUs? [3954]

ide_intr() deletes a request from ide_queue (line 3902). What happens if an IDE interrupt occurs when a process is adding a block to the queue?

Why does idestart() want the caller to hold the lock? Why doesn't it just acquire the lock itself?

recursive locks are a bad idea
ideintr should certainly not use that instead of disabling interrupts!

lock ordering
iderw: sleep acquires ptable.lock
never acquire ptable.lock and then ide_lock
Also never acquire ide_lock and then some other lock
So, `ptable.lock` has least priority (will always be the inner-most lock), ide_lock will have higher priority than `ptable.lock` but lower priority than all other locks in xv6.

Why not check p->killed in the ide_rw() loop? As also discussed earlier, this sleep loop is guaranteed to finish in bounded time. Also, the caller of `iderw` may be in an inconsistent state and may not expect `iderw` to return without actually reading the buffer. In this case, it is okay for some function higher in the stack to check `p->killed` and exit.

Demand Paging

Now, let's move back to our discussion on virtual memory. Recall that paging divides the address space into page-sized segments and allows a more flexible mapping between VA space and PA space. Also, the TLB acts as a cache for page table mappings and usually has high hit rates, that allow paging to be used without excessive overhead.

Further, previously we said that while loading a process, the executable file in `a.out` format is parsed, and all its memory contents are loaded into physical memory (from disk) and corresponding mappings created in the virtual address space (through the page table). In general, a loaded process may not necessarily access all its code/data and so, many disk reads can be avoided if the code/data pages are loaded on-demand.

Here, at program load time, you could load some pages in physical memory and create corresponding mappings in the page table. For others, you may mark them not-present, but also store somewhere that these pages are present at a location on the disk (along with the details of the location). Notice that now, some mappings in the page table are marked not-present, even if the program believes that it has loaded them. In other words, the OS is playing tricks with the program under the carpet, without the program's knowledge.

Previously, if a program tried to access a virtual address that is not currently mapped (i.e., the corresponding present bit is zero in the page table), an exception would get generated. This exception is also called a *page fault*. The corresponding OS exception handler (also called `page fault handler`) would get to execute and would likely kill the process. With demand paging however, the page fault handler will additionally check if this address is logically present (but physically not present due to its demand paging optimization), and if so, will load it from the disk to the physical memory on-demand.

After the OS loads the page from disk to physical memory, it creates a mapping in the page table, such that a page fault is not generated again on that address, and restarts the faulting instruction. Notice that in doing so, the OS assumes that it is safe to restart a faulting instruction. On the x86 platform, the hardware guarantees that if an instruction causes an exception, the machine state is left unmodified (as though the instruction never executed) before transferring control to the exception handler. This property of the hardware is called *precise exceptions*. If the hardware guarantees precise exceptions, it is safe for the OS to return control to the faulting instruction, thus causing it to execute again without changing the semantics of the process logic.