Scheduling

Required reading: Eliminating receive livelock

Notes based on prof. Morris's lecture on scheduling (6.824, fall'02).

Overview

- What is scheduling? The OS policies and mechanisms to allocates resources to entities. A good scheduling policy ensures that the most important entitity gets the resources it needs. This topic was popular in the days of time sharing, when there was a shortage of resources. It seemed irrelevant in era of PCs and workstations, when resources were plenty. Now the topic is back from the dead to handle massive Internet servers with paying customers. The Internet exposes web sites to international abuse and overload, which can lead to resource shortages. Furthermore, some customers are more important than others (e.g., the ones that buy a lot).
- Key problems:
  - Gap between desired policy and available mechanism. The desired policies often include elements that not implementable with the mechanisms available to the operation system. Furthermore, often there are many conflicting goals (low latency, high throughput, and fairness), and the scheduler must make a trade-off between the goals.
  - Interaction between different schedulers. One have to take a systems view. Just optimizing the CPU scheduler may do little to for the overall desired policy.
- Resources you might want to schedule: CPU time, physical memory, disk and network I/O, and I/O bus bandwidth.
- Entities that you might want to give resources to: users, processes, threads, web requests, or MIT accounts.
- Many polices for resource to entity allocation are possible: strict priority, divide equally, shortest job first, minimum guarantee combined with admission control.
- General plan for scheduling mechanisms
  1. Understand where scheduling is occuring.
  2. Expose scheduling decisions, allow control.

3. Account for resource consumption, to allow intelligent control.

- Simple example from 6.828 kernel. The policy for scheduling environments is to give each one equal CPU time. The mechanism used to implement this policy is a clock interrupt every 10 msec and then selecting the next environment in a round-robin fashion.

  But this only works if processes are compute-bound. What if a process gives up some of its 10 ms to wait for input? Do we have to keep track of that and give it back?

  How long should the quantum be? is 10 msec the right answer? Shorter quantum will lead to better interactive performance, but lowers overall system throughput because we will reschedule more, which has overhead.

  What if the environment computes for 1 msec and sends an IPC to the file server environment? Shouldn't the file server get more CPU time because it operates on behalf of all other functions?

  Potential improvements for the 6.828 kernel: track "recent" CPU use (e.g., over the last second) and always run environment with least recent CPU use. (Still, if you sleep long enough you lose.) Other solution: directed yield; specify on the yield to which environment you are donating the remainder of the quantuam (e.g., to the file server so that it can compute on the environment's behalf).

- Pitfall: Priority Inversion

```
Assume policy is strict priority.
Thread T1: low priority.
Thread T2: medium priority.
Thread T3: high priority.
T1: acquire(l)
context switch to T3
T3: acquire(l)... must wait for T1 to release(l)...
context switch to T2
T2 computes for a while
T3 is indefinitely delayed despite high priority.
Can solve if T3 lends its priority to holder of lock it
  So T1 runs, not T2.
[this is really a multiple scheduler problem.]
[since locks schedule access to locked resource.]
```

- Pitfall: Efficiency. Efficiency often conflicts with fairness (or any other policy). Long time quantum for efficiency in CPU scheduling versus low delay. Shortest seek versus FIFO disk scheduling. Contiguous read-ahead vs data needed now. For example, scheduler swaps out my idle emacs to let gcc run faster with more phys mem. What happens when I type a key? These don't fit well into a "who gets to go next" scheduler framework. Inefficient scheduling may make *everybody* slower, including high priority users.
- Pitfall: Multiple Interacting Schedulers. Suppose you want your emacs to have priority over everything else. Give it high CPU priority. Does that mean nothing else will run if emacs wants to run? Disk scheduler might not know to favor emacs's disk I/Os. Typical UNIX disk scheduler favors disk efficiency, not process prio. Suppose emacs needs more memory. Other processes have dirty pages; emacs must wait. Does disk scheduler know these other processes' writes are high prio?
- Pitfall: Server Processes. Suppose emacs uses X windows to display. The X server must serve requests from many clients. Does it know that emacs' requests should be given priority? Does the OS know to raise X's priority when it is serving emacs? Similarly for DNS, and NFS. Does the network know to give emacs' NFS requests priority?

In short, scheduling is a system problem. There are many schedulers; they interact. The CPU scheduler is usually the easy part. The hardest part is system structure. For example, the *existence* of interrupts is bad for scheduling. Conflicting goals may limit effectiveness.

Case study: modern UNIX

Goals:

- Simplicity (e.g. avoid complex locking regimes).
- Quick response to device interrupts.
- Favor interactive response.

UNIX has a number of execution environments. We care about scheduling transitions among them. Some transitions aren't possible, some can't be be controlled. The execution environments are:

- Process, user half

- Process, kernel half
- Soft interrupts: timer, network
- Device interrupts

The rules are:

- User is pre-emptible.
- Kernel half and software interrupts are not pre-emptible.
- Device handlers may not make blocking calls (e.g., sleep)
- Effective priorities: intr > soft intr > kernel half > user

Rules are implemented as follows:

- UNIX: Process User Half. Runs in process address space, on per-process stack. Interruptible. Pre-emptible: interrupt may cause context switch. We don't trust user processes to yield CPU. Voluntarily enters kernel half via system calls and faults.
- UNIX: Process Kernel Half. Runs in kernel address space, on per-process kernel stack. Executes system calls and faults for its process. Interruptible (but can defer interrupts in critical sections). Not pre-emptible. Only yields voluntarily, when waiting for an event. E.g. disk I/O done. This simplifies concurrency control; locks often not required. No user process runs if any kernel half wants to run. Many process' kernel halfs may be sleeping in the kernel.
- UNIX: Device Interrupts. Hardware asks CPU for an interrupt to ask for attention. Disk read/write completed, or network packet received. Runs in kernel space, on special interrupt stack. Interrupt routine cannot block; must return. Interrupts are interruptible. They nest on the one interrupt stack. Interrupts are not pre-emptible, and cannot really yield. The real-time clock is a device and interrupts every 10ms (or whatever). Process scheduling decisions can be made when interrupt returns (e.g. wake up the process waiting for this event). You want interrupt processing to be fast, since it has priority. Don't do any more work than you have to. You're blocking processes and other interrupts. Typically, an interrupt does the minimal work necessary to keep the device happy, and then call wakeup on a thread.
- UNIX: Soft Interrupts. (Didn't exist in xv6) Used when device handling is expensive. But no obvious process context in which to run. Examples include IP forwarding, TCP input processing. Runs in kernel space, on

interrupt stack. Interruptable. Not pre-emptable, can't really yield. Triggered by hardware interrupt. Called when outermost hardware interrupt returns. Periodic scheduling decisions are made in timer s/w interrupt. Scheduled by hardware timer interrupt (i.e., if current process has run long enough, switch).

Is this good software structure? Let's talk about receive livelock.

Paper discussion

- What is application that the paper is addressing: IP forwarding. What functionality does a network interface offer to driver?
    - Read packets
    - Poke hardware to send packets
    - Interrupts when packet received/transmit complete
    - Buffer many input packets
- What devices in the 6.828 kernel are interrupt driven? Which one are polling? Is this ideal?
- Explain Figure 6-1. Why does it go up? What determines how high the peak is? Why does it go down? What determines how fast it goes does? Answer:

```
(fraction of packets discarded)(work invested in discarde
           -------------------------------------------
              (total work CPU is capable of)
```

- Suppose I wanted to test an NFS server for livelock.

```
  Run client with this loop:
    while(1){
      send NFS READ RPC;
      wait for response;
    }
```

  What would I see? Is the NFS server probably subject to livelock? (No--offered load subject to feedback).
- What other problems are we trying to address?
    - Increased latency for packet delivery and forwarding (e.g., start disk head moving when first NFS read request comes)
    - Transmit starvation

    - User-level CPU starvation
- Why not tell the O/S scheduler to give interrupts lower priority? Non-preemptible. Could you fix this by making interrupts faster? (Maybe, if coupled with some limit on input rate.)
- Why not completely process each packet in the interrupt handler? (I.e. forward it?) Other parts of kernel don't expect to run at high interrupt-level (e.g., some packet processing code might invoke a function that sleeps). Still might want an output queue
- What about using polling instead of interrupts? Solves overload problem, but killer for latency.
- What's the paper's solution?
    - No IP input queue.
    - Input processing and device input polling in kernel thread.
    - Device receive interrupt just wakes up thread. And leaves interrupts *disabled* for that device.
    - Thread does all input processing, then re-enables interrupts.

  Why does this work? What happens when packets arrive too fast? What happens when packets arrive slowly?

- Explain Figure 6-3.
    - Why does "Polling (no quota)" work badly? (Input still starves xmit complete processing.)
    - Why does it immediately fall to zero, rather than gradually decreasing? (xmit complete processing must be very cheap compared to input.)
- Explain Figure 6-4.
    - Why does "Polling, no feedback" behave badly? There's a queue in front of screend. We can still give 100% to input thread, 0% to screend.
    - Why does "Polling w/ feedback" behave well? Input thread yields when queue to screend fills.
    - What if screend hangs, what about other consumers of packets? (e.g., can you ssh to machine to fix screend?) Fortunately screend typically is only application. Also, re-enable input after timeout.
- Why are the two solutions different?
    1. Polling thread *with quotas*.
    2. Feedback from full queue.

(I believe they should have used #2 for both.)

- If we apply the proposed fixes, does the phenomemon totally go away? (e.g. for web server, waits for disk, &c.)
    - Can the net device throw away packets without slowing down host?
    - Problem: We want to drop packets for applications with big queues. But requires work to determine which application a packet belongs to Solution: NI-LRP (have network interface sort packets)
- What about latency question? (Look at figure 14 p. 243.)
    - 1st packet looks like an improvement over non-polling. But 2nd packet transmitted later with poling. Why? (No new packets added to xmit buffer until xmit interrupt)
    - Why? In traditional BSD, to amortize cost of poking device. Maybe better to poke a second time anyway.
- What if processing has more complex structure?
    - Chain of processing stages with queues? Does feedback work? What happens when a late stage is slow?
    - Split at some point, multiple parallel paths? No so great; one slow path blocks all paths.
- Can we formulate any general principles from paper?
    - Don't spend time on new work before completing existing work.
    - Or give new work lower priority than partially-completed work.