

Section: Intro to I/O

1 Announcements

- Project 3 ... and the ball keeps on rolling

Disclaimer: These notes aren't substitutes for attending lecture or reading the book. Those have been created with careful revision and review. These notes are mainly intended to be a more thorough index of the lectures and reading so that you can go off and read the full treatment on the topic and be able to ask the right questions while reading those two sources. If you have comments or suggestions please send me email and I'll fix them up.

2 Nachos Project 3

Since these notes are a bit delayed this section is no longer useful since most of you already understand the project. However i think it'll be good to document what live lock is so i'll just go through that.

2.1 Livelock

Livelock is this interesting case where you could get into where a process or a thread is not waiting on any other thread but it can't make progress for another very insidious problem. Lets imagine that we have a really small system with exactly one physical page. Lets trace through what happens when we try to run a load instruction. First the instruction fetch part of the pipeline will miss because the instruction page is not resident and brings it in and then restarts the instruction, however then we want to load another data page and since that page is not resident it'll try to load in the data page. However it needs to kick out the instruction page to make room and then restarts the instruction. However when we want to run the instruction it is no longer there and we have locked ourselves in an infinite loop trying to execute an instruction but making no forward progress. Its often not the case that we have one physical page but this situation could easily arise when the number of processes is on the same order of magnitude as the number of physical pages. The short answer for nachos is don't worry about it but its still an interesting case to consider.

3 Recap and final thoughts on paging

Last week was mainly about page tables and paging. One of the key ideas was picking out what physical page to replace and what algorithm to use to conduct that replacement. We saw that there were different implications of each but we kinda brushed the details of what

physical pages (also called page frames) we can choose from. There are two policies that we can use.

- **Global replacement** Select replacement frame from all available frame. Processes can steal
- **Local replacement** Each process can select a page to replace from what it only owns

In addition to the replacement we can also control how the frames are initially allocated to the different processes and there are a couple of methods for that as well.

- **Equal allocation** Every process gets an equal share of physical memory
- **Proportional allocation** Every process gets pages based on the number of pages it says it needs up front.
- **Priority allocation** Proportional using priority instead of size. That way kernel processes are much more likely to be paged in.
- **Some combination of all of the above** As in any OS we never have either this nor that, we usually combine all the techniques.

3.1 The working set

Professor Kubi had a great definition of what working set actually is in the lecture with mathematical symbols that I won't reproduce here. Instead I'll give you some high-level thoughts on what it is from my experiences. The working set of an algorithm or an application is the set of memory that is accessed during a given period of time. Notice that if our "given period of time" is too small then we don't capture the right model of working set. If our "given period of time" is too large then we will count too much as the working set. In scientific applications (think very regular computation) the working set is usually defined as the time and memory access from the inner loops. For databases and other applications it's a bit harder but you can do it. The reason it's important to us is that it captures information about locality which is directly applicable to replacement strategies. However the problem with analyzing working sets is that it is a very application dependent idea and there is obviously no magic oracle that will tell you whether you are right or not. If you are more curious about this idea, I can point you to a lot of papers. The study of working sets has been the topic of a lot of systems research.

4 I/O

One of the main results of paging that we saw above is that we have to write out pages to disk. In addition we have just assumed some magic way of the data and programs just showing up into our system. All these things allude to a complex I/O system. We will now dive into that section a bit more to understand what is happening there. One of the fundamental roles of an OS is to manage the I/O subsystem so that different devices (i.e. usb keys, hard drives, sound cards, graphics cards, cd rom drives) can all talk together to produce a full featured system. Otherwise you have a very expensive system that can calculate pi or some other obscure thing that few people really care about.

4.1 Classifications of I/O

We have come a long way from the time of little switches and leds telling you whether or not the program was correct. But there are many different I/O devices and the goal of any OS programmer is how do we manage the complexity of the variety of devices out there. Before we can see how to manage this complexity lets see what the different types of devices are.

- **Byte/Block.** some I/O devices provide byte access to data while others provide block access to data so that you can only read blocks at a time.
- **Sequential/Random.** Some devices must be accessed sequentially and there is no concept of random access (sounds weird doesn't it) while others allow you to access data anywhere on the device at roughly the same speed. "Roughly" here is used very loosely.
- **Polling/Interrupts.** Some devices require that you constantly ask it if there is new data while other devices generate interrupts telling you when service is required

Question: Take a look at all the things attached to a computer or any other device. What types of I/O classification can we place on it?

We also want to be able to provide a standardized interface to these devices. In Unix it is all the files in the /dev folder and it varies from OS to OS. The idea is that there is a common interface with calls such as `open()`, `close()`, `read()`, `write()`, etc that allow common access to the device. However the results of these calls and what the arguments mean varies from device to device. The timing of a device is referred to as the results of those calls and what the data read/write model is. There are a few interesting models with timing.

- **Blocking interface** or "wait": the syscalls for read and write are put to sleep until the data is ready to be read or written and thus the entire thread of control stops in its tracks. (this is what we mean by waiting for I/O in the early part of the semester)
- **Non-blocking interface** or "don't wait": The system calls return write away with how much they managed to read or write and often times it maybe 0.
- **Asynchronous Interface** or "tell me later": The syscall tells the system that a read or write needs to be performed on a chunk of memory and the call returns a handle. You can then use the handle to peak into the status of the operation. Its a cool interface on the surface but the main problem is that the buffer space can not be overwritten or reused until we have confirmation that the syscall has finished. This trick is used quite a bit to overlap computation with communication or data transfer but it often requires more memory.

Question: What types of application would each interface be useful for?

Now that we have listed the characteristics lets talk a bit about how to actually talk to the various devices. All the devices are usually connected on what is called a bus. A

bus is simply a combination of physical wires and a protocol that connects many different devices. A common example of a bus in your home machine is a PCI bus or the IDE bus. If you want to find out more there is tons of dry reading available on the web about and how they actually work. There are two ways to read that bus. One is through an explicit I/O instruction and the other is a new concept of memory mapped I/O.

Memory mapped I/O provides the processor the illusion that part of a devices registers and local memory is part of the process's address space. Thus with simple load or store instructions we can write to the memory system. The underlying bus then takes these addresses and then translates them and actually forwards them to the right device. The reason this is really cool is that it provides an intuitive framework for writing applications, however keeping everything straight is hard. For example on a graphics card, in order to render a frame we can send the graphics card a set of triangles and write them directly into the graphic's cards memory and then set a bit in a control register. Once that bit is set the graphics cards reads that memory and pipes the information to the screen. Obviously I have brushed over details but realize the power. We are controlling devices and memory all through a load and store and letting the processor take care of it. Because we are doing loads and stores we can even set up the page tables to properly handle the permissions and not worry about random processes messing up address spaces.

Question: What are the drawbacks of memory mapped I/O? How do you deal with context switches and switching the device amongst different processes?

4.2 Devices talking to the CPU

Until now we have focused on how the CPU can send data and receive data, but how do we know when we are done? There are a couple of things different I/O devices do in order to tell the CPU when they are done.

4.2.1 Interrupts

In an interrupt based device the device actually signals an interrupt when a piece of data is ready and its up to the application and the OS to properly handle these interrupts. The obvious benefit is that it handles unpredictable events really well (such as the first network packet in a stream of network packets). However processing these interrupts can lead to high overheads.

4.2.2 Polling

In this model the OS is constantly looking over the shoulder of the device asking "are you done yet or do you have anything new for me?" The problem with this approach is that you could waste a lot of time asking these questions but it does lead to a lower overhead than interrupts. An example of good polling devices are keyboards and mouse where we don't have to ask the question "do you have anything new for me?" very often (in the processor's timescale)

4.2.3 Combination

As in anything in this course, the actual solutions use a combination of both polling and interrupts to leverage the best of both worlds. The example in class was the network card could interrupt the CPU indicating that a stream of network packets is arriving, and then the CPU could poll for each subsequent network packet since we know that they are right behind the first one.