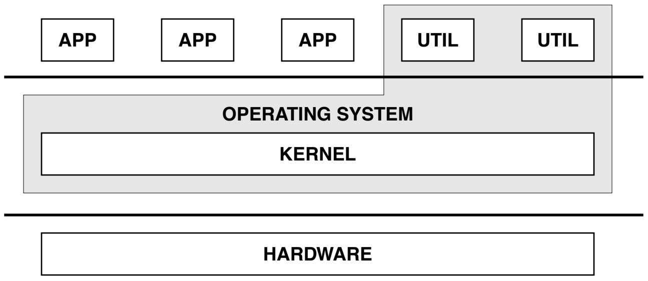
The OS and Multitasking: An Example

ENEE 446: Digital Computer Design, Spring 2006
Prof. Bruce Jacob



The diagram to the right illustrates the simplistic view of what goes on in a typical system. Applications run in the context of the operating system. However, the operating system is actually broken down further into the *OS kernel* and a set of *OS utilities*. The utilities are things like the shell, the windowing system, compilers, linkers, loaders, etc. They often run in user-mode (i.e., they are like APPS).

What does "user-mode" mean? It means that there are special instructions that directly affect the state of the machine and perform powerful operations. Normal applications are not allowed access to these instructions; if a normal application executed one of these instructions, the operating system would kill it. An example is the instruction that sets the ASID register (address-space identifier). This register identifies the process that is currently running, so that different processes do not interfere with each other. The instruction that sets a value in the ASID register is protected, because if a normal application could set the value in the ASID register, then it could masquerade as any other process on the system.

To provide protection against such abuse, the hardware typically has at least two *modes* of operation. We will concern ourself with a simple, common model of two modes: USER and PRIVILEGED. If the machine is in privileged mode, then privileged instructions are allowed. Otherwise, their use causes a special interrupt. The kernel is a big block of code that runs in privileged mode. Moreover, it is the *only* block of code that runs in privileged mode.

Applications cannot have direct access to all of the hardware all the time, else anarchy. Otherwise, you have to force them to cooperate (which is similar to anarchy if done poorly, or similar to the above diagram if done well).

In reality, an application is made to *think* that it has direct access to the hardware, but that access is moderated by the operating system, which can take over at any time. Here is an example that demonstrates what actually goes on in a real system.

We will look at three processes that execute "simultaneously" on a single processor.

APP	references a data structure for the first time and is going to cause a TLB miss, then a page fault
CAT	is reading a large portion of a file to the disk (which succeeds)
NET	is sending a large network packet out

So. We have three processes: APP, CAT, and NET, and we have the KERNEL code. The following depicts an interval of time on the machine. We begin *in media res*, with all of the processes having run for a while. Note that this example is very stylized, it assumes that each process only has a single thread of control (from the kernel's point of view), and it was written in stream-of-consciousness off the top of my head. It is intended to present an impression of what goes on between the hardware and operating system, and not necessarily depict a perfectly accurate (or even self-consistent) OS implementation.

ASID	USER-CODE	KERNEL-CODE	HARDWARE
APP	APP is running executes load/store, causes TLB miss (say the TLB is software-managed)		
APP			TLBMISS interrupt save EPC/EPC+1 turn on privs. vector to TLBMISS
APP		TLBMISS entry point: build address for PTE load PTE insert into TLB jump to EPC + turn off privileges	
APP	Retries load/store instruction (we jumped to EPC, not EPC+1) this time, it succeeds APP keeps running, oblivious		
APP			TIMER interrupt save EPC/EPC+1 turn on privs. vector to TIMER
APP		TIMER entry point: Evidently, APP has exceeded its quota call scheduler()	
APP		function *scheduler*: check RunQ for another process: is there another? yes — CAT copy CPU state to u.state (registers, etc.) put EPC+1 into u.nextPC put CAT into ASID register	
CAT		copy u.state into CPU + reset timer put u.nextPC into register jump to register + turn off privileges	

CAT	CAT begins executing from where it last left off
	calls read(fd, &buf, 64K)
	read() is a library routine:
	   puts address of BUF into u.base_addr
	   puts 64K into u.count
	   puts SYSCALL_READ into reg1
	   puts FD into reg2
	   calls SYSCALL -- interrupts machine		

CAT			SYSCALL interrupt save EPC/EPC+1 turn on privs. vector to SYSCALL
CAT		SYSCALL entry point: looks in reg1: SYSCALL_READ looks in reg2: FILEDES looks in u.filedes[FILEDES] for state of device call function devices[DEV].entrypoint[SYSCALL-READ] with args: OUT: u.base_addr, SIZE: u.count, DISKBLOCK: u.filedes[FILEDES].curblock	
[this sets up transfers from DISK to internal buffer pool, then copies data from the buffers (once they are full) into user space, one buffer at a time, each time incrementing u.base_addr and decrementing u.count]			
CAT		function *devices[DEV].entrypoint[SYSCALL-READ]*: sends request to DISK: get block u.filedes[FILEDES].curblock goes to sleep on u.filedes[FILEDES].curblock	

ASID	USER-CODE	KERNEL-CODE	HARDWARE
CAT		function *sleep* (sleep acts something like a context switch): save PC of instruction after sleep() in u.kernPC take CAT off RunQ & put on SleepQ call scheduler()	
CAT		function *scheduler*: check RunQ for another process: is there another? yes — NET copy CPU state to u.state (registers, etc.) put EPC+1 into u.nextPC put NET into ASID register	
NET		copy u.state into CPU + reset timer put u.nextPC into register jump to register + turn off privileges	

NET	NET begins executing from where it last left off
	calls send(sockfd, buf, siz)
	send() is a library routine:
	    puts BUF into u.base_addr
	    puts SIZ into u.count
	    puts SYSCALL_WRITE into reg1
	    puts SOCKFD into reg2
	    calls SYSCALL -- interrupts machine		

NET			SYSCALL interrupt save EPC/EPC+1 turn on privs. vector to SYSCALL
NET		SYSCALL entry point: looks in reg1: SYSCALL_WRITE looks in reg2: FILEDES looks in u.filedes[FILEDES] for state of device call function devices[DEV].entrypoint[SYSCALL_WRITE] with args: IN: u.base_addr, SIZE: u.count PORT: u.filedes[FILEDES].portnum	
[assume buffer space available in the driver]			
NET		function *devices[DEV].entrypoint[SYSCALL_WRITE]:* copy u.count bytes: u.base_addr -> local buffer update u.status_of_syscall == DONE send msg to device: WAKEUP! sending you u.count bytes on PORT goes to sleep on PORT (or some corresponding addr) save PC after sleep() in u.kernPC	
[THIS TIME, sleep() doesn't take NET off RunQ, because the data is safely in the kernel, and as far as NET knows, the packet has gone out onto network. The kernel can either go directly back to NET (by jumping to EPC+1) or switch to another process]			
NET		copy u.nextPC into register jump to register + turn off privileges	
NET	NET returns from send(), continues processing		
NET			TIMER interrupt save EPC/EPC+1 turn on privs. vector to TIMER
NET		TIMER entry point: Evidently, NET has exceeded its quota call scheduler()	
NET		function *scheduler*: check RunQ for another process: is there another? yes — APP copy CPU state to u.state (registers, etc.) put EPC+1 into u.nextPC put APP into ASID register	

ASID	**USER-CODE**	**KERNEL-CODE**	**HARDWARE**
APP		copy u.state into CPU + reset timer put u.nextPC into register jump to register + turn off privileges	
APP	APP begins executing from where it last left off		
APP			DEVICE interrupt save EPC/EPC+1 turn on privs. vector to device[DEV].intr()

APP		device[DEV].intr entry point:
		happens to be DEV = disk: block BLOCKNUM is here
		wakeup(BLOCKNUM)
		    anyone sleeping on BLOCKNUM?
		    yes -- this is what CAT was waiting for
		awaken() sleeping kernel thread	

APP		copy CPU state to u.state put EPC+1 into u.nextPC put CAT into ASID register (to get access to CAT's u. struct & VM space)	
CAT		copy u.state into CPU + reset timer put **u.kernPC** into register jump to register + turn **on** privileges … jumps to 1st instruction after sleep()	

CAT		… in function *devices[DEV].entrypoint[SYSCALL-READ]*:
		copy block BLOCKNUM from disk to internal buffer
		copyout(u.base_addr, block, blocksize)
		u.base_addr += blocksize;
		u.count -= blocksize;
		if (u.count == 0) {
		    make CAT active again
		} else {
		    get next block (or portion thereof)
		}	

[assume we're done ... u.count == 0]			
CAT		u.status_of_syscall = DONE move CAT from SleepQ to RunQ	
[at this point, we have two choices. we can either go back to APP, who was preempted by the disk I/O, or we can restart CAT. perhaps we want to look at the timing logs -- if CAT had previously eaten up very little of its quantum, then maybe we jump straight to it. ... there is room for choices ... assume that the copyin & copyout took a while ... CAT doesn't have much time left to it (it would execute very few instructions before ending its quantum). so we return to APP]			
CAT		function *scheduler*: check RunQ for another process: is there another? yes — APP copy CPU state to u.state (registers, etc.) — [*may not be necessary*] put EPC+1 into u.nextPC put APP into ASID register	
APP		copy u.state into CPU + reset timer put u.nextPC into register jump to register + turn off privileges	
APP	APP begins executing from where it last left off		
APP			DEVICE interrupt save EPC/EPC+1 turn on privs. vector to device[DEV].intr()

ASID	USER-CODE	KERNEL-CODE	HARDWARE
APP		device[DEV].intr entry point: happens to be NUM = network controller: ready for data on PORTNUM wakeup(PORTNUM) anyone sleeping on PORTNUM? yes -- NET was waiting for this awaken( ) sleeping kernel thread	
APP		copy CPU state to u.state put EPC+1 into u.nextPC put NET into ASID register	
NET		copy u.state into CPU + reset timer put **u.kernPC** into register jump to register + turn **on** privileges … jumps to 1st instruction after sleep()	
NET		… in function *devices[DEV].entrypoint[SYSCALL_WRITE]:* copies bytes from buffer to network controller if it all fits, we can stop if the network controller can take only a portion, we go to sleep again	
[if there had not been room in the driver to copy bytes in, we also would have to sleep, but at a different place.]			
NET		assume we are done. call scheduler()	
NET		function *scheduler*: check RunQ for another process: is there another? yes — APP copy CPU state to u.state (registers, etc.) put EPC+1 into u.nextPC put APP into ASID register	
APP		copy u.state into CPU + reset timer put u.nextPC into register jump to register + turn off privileges	
APP	APP begins executing from where it last left off, again. This time, it performs another load/store that causes a TLB miss		
APP			TLBMISS interrupt save EPC/EPC+1 turn on privs. vector to TLBMISS
APP		TLBMISS entry point: build address for PTE load PTE	
[oops -- the PTE says that it is currently not a valid translation -- that the data is not in memory but on disk. here is a design choice: do we actually CHECK the PTE or do we blindly put it into the TLB? checking will increase overhead of the common case by 20-30%. **MIPS solution**: put it blindly into TLB]			
APP		insert into TLB jump to EPC + turn off privileges	
APP	Retries load/store instruction (we jumped to EPC, not EPC+1) it fails again, but this time, with a different interrupt type: this time, the mapping is in the TLB, so we don't miss, but the mapping is INVALID, so we get a PAGE FAULT.		
APP			PAGEFAULT interrupt save EPC/EPC+1 turn on privs. vector to PAGEFAULT
APP		PAGEFAULT entry point: look at PTE -- what are its flags? says that page is on disk, not in memory	
		THIS IS WHERE LIFE GETS WEIRD.	

ASID	USER-CODE	KERNEL-CODE	HARDWARE
[now, we (potentially) have to involve the filesystem. up to now, there have been strict boundaries between devices, allowing us to have strict boundaries between the drivers — no overlap of duties, no contention for resources within the kernel. however, NOW, we mix the virtual memory system with the filesystem/disk-I/O ... this is an issue that is implemented differently in virtually EVERY operating system — the interplay between VM and FILESYSTEM. this is one reason why so many people are suggesting we merge the two (as in Multics, the original OS). this is one of the things the SASOS guys talk about. For now, let's just say we INITIATE DISK XFER into the application's address space — just like the read() call that happened earlier in CAT.]			
APP		we put APP to sleep(), take it off runQ when the data comes back, we copy it out APP's address space and put APP back on RunQ	

One of the main questions that is glossed over COMPLETELY by this discussion is: WHICH STACK? when the operating system is executing, which stack does it use?

The way I've set it up, the ASID corresponds roughly to whose stack you're operating on.