

# 6.828 Fall 2006 Lab 4: Preemptive Multitasking

**Handed out Wednesday, October 11**

**Part A due Thursday, October 19**

**Part B due Thursday, October 26**

**Part C due Thursday, November 2**

## Introduction

In this lab you will implement preemptive multitasking among multiple simultaneously active user-mode environments. In part A you will first implement round-robin scheduling and the basic environment management system calls (calls that create and destroy environments, and allocate/map memory). In part B, you will implement a Unix-like `fork()`, which allows one user-mode environment to fork off other, "child" environments, which start off as "clones" of the parent but can subsequently execute independently of the parent. Finally, in part C you will add support for inter-process communication (IPC), allowing different user-mode environments to communicate and synchronize with each other explicitly. You will also add support for hardware clock interrupts and preemption.

## Getting Started

Download the lab 4 code (`lab4-handout.gz`) from labs section for this course, and unpack it into your 6.828 directory as before. If you downloaded the original lab handout (`lab4-handout.tar.gz`) then you'll need to update `grade.sh`; you can use this patch (see labs section) to do so. As usual, you will need to merge our new code for this lab into your source tree, or copy the changes that you have made in the previous labs over to our tree.

## Lab Requirements

This lab is divided into three parts, A, B, and C. We have allocated one week in the schedule for each part.

As before, you will need to do all of the regular exercises described in the lab and *at least one* challenge problem. (You do not need to do one challenge problem per part, just one for the whole lab.) Additionally, you will need to write up a brief description of the challenge problem that you implemented. If you implement more than one challenge problem, you only need to describe one of them in the write-up, though of course you are welcome to do more. Place the write-up in a file called `answers.txt` (plain text) or `answers.html` (HTML format) in the top level of your `lab4` directory before handing in your work.

## Part A: User-level Environment Creation and Cooperative Multitasking

In the first part of this lab, you will implement some new JOS kernel system calls to allow user-level environments to create additional new environments. You will also implement *cooperative* round-robin scheduling, allowing the kernel to switch from one environment to another when the current environment voluntarily relinquishes the CPU (or exits). Later in part C you will implement *preemptive* scheduling, which allows the kernel to re-take control of the CPU from an environment after a certain time has passed even if the environment does not cooperate.

### Round-Robin Scheduling

Your first task in this lab is to change the JOS kernel so that it does not always just run the environment in `envs[0]`, but instead can alternate between multiple environments in "round-robin" fashion. Round-robin scheduling in JOS works as follows:

- The first environment, in `envs[0]`, will from now on always be a special *idle* environment, which always runs the program `user/idle.c`. The purpose of this program is simply to "waste time" whenever the processor has nothing better to do - it just perpetually attempts to give up the CPU to another environment. Read the code and comments in `user/idle.c` for other useful details. We have modified `kern/init.c` for you to create this special idle environment in `envs[0]` before creating the first "real" environment in `envs[1]`.
- The function `sched_yield()` in the new `kern/sched.c` is responsible for selecting a new environment to run. It searches sequentially through the `envs[]` array in circular fashion, starting just after the previously running environment (or at the beginning of the array if there was no previously running environment), picks the first environment it finds with a status of `ENV_RUNNABLE` (see `inc/env.h`), and calls `env_run()` to jump into that environment. However, `sched_yield()` is aware that `envs[0]` is the special idle environment, and *never* picks it unless there are *no other* runnable environments.
- We have implemented a new system call for you, `sys_yield()`, which user environments can call to invoke the kernel's `sched_yield()` function and thereby voluntarily give up the CPU to a different environment. As you can see in `user/idle.c`, the idle environment does this routinely.
- Whenever the kernel switches from one environment to another, it must ensure the old environment's registers are saved so they can be restored properly later. Why? Where does this happen?

**Exercise 1.** Implement round-robin scheduling in `sched_yield()` as described above. Don't forget to modify `syscall()` to dispatch `sys_yield()`.

Modify `kern/init.c` to create two (or more!) environments that all run the program `user/yield.c`. You should see the environments switch back and forth between each other five times before terminating, like this:

```
...  
Hello, I am environment 00001001.  
Hello, I am environment 00001002.  
Back in environment 00001001, iteration 0.  
Back in environment 00001002, iteration 0.  
Back in environment 00001001, iteration 1.  
Back in environment 00001002, iteration 1.  
...
```

After the `yield` programs exit, the idle environment should run and invoke the JOS kernel debugger. If all this does not happen, then fix your code before proceeding.

Question: In your implementation of `env_run()` you should have called `lcr3()`. Before and after the call to `lcr3()`, your code makes references (at least it should) to the variable `e`, the argument to `env_run`. Upon loading the `%cr3` register, the addressing context used by the MMU is instantly changed. But a virtual address (namely `e`) has meaning relative to a given address context--the address context specifies the physical address to which the virtual address maps. Why can the pointer `e` be dereferenced both before and after the addressing switch?

*Challenge!* Add a less trivial scheduling policy to the kernel, such as a fixed-priority scheduler that allows each environment to be assigned a priority and ensures that higher-priority environments are always chosen in preference to lower-priority environments. If you're feeling really adventurous, try implementing a Unix-style adjustable-priority scheduler or even a lottery or stride scheduler. (Look up "lottery scheduling" and "stride scheduling" in Google.)

Write a test program or two that verifies that your scheduling algorithm is working correctly (i.e., the right environments get run in the right order). It may be easier to write these test programs once you have implemented `fork()` and IPC in parts B and C of this lab.

*Challenge!* The JOS kernel currently does not allow applications to use the x86 processor's x87 floating-point unit (FPU), MMX instructions, or Streaming SIMD Extensions (SSE). Extend the `Env` structure to provide a save area for the processor's floating point state, and extend the context switching code to save and restore this state properly when switching from one environment to another. The `FXSAVE` and `FXRSTOR` instructions may be useful, but note that these are not in the old i386 user's manual because they were introduced in more recent processors.

Write a user-level test program that does something cool with floating-point.

## System Calls for Environment Creation

Although your kernel is now capable of running and switching between multiple user-level environments, it is still limited to running environments that the *kernel* initially set up. You will now implement the necessary JOS system calls to allow *user* environments to create and start other new user environments.

Unix provides the `fork()` system call as its process creation primitive. Unix `fork()` copies the entire address space of calling process (the parent) to create a new process (the child). The only differences between the two observable from user space are their process IDs and parent process IDs (as returned by `getpid` and `getppid`). In the parent, `fork()` returns the child's process ID, while in the child, `fork()` returns 0. By default, each process gets its own private address space, and neither process's modifications to memory are visible to the other.

You will provide a different, more primitive set of JOS system calls for creating new user-mode environments. With these system calls you will be able to implement a Unix-like `fork()` entirely in user space, in addition to other styles of environment creation. The new system calls you will write for JOS are as follows:

`sys_exofork`:

This system call creates a new environment with an almost blank slate: nothing is mapped in the user portion of its address space, and it is not runnable. The new environment will have the same register state as the parent environment at the time of the `sys_exofork` call. In the parent, `sys_exofork` will return the `env_id_t` of the newly created environment (or a negative error code if the environment allocation failed). In the child, however, it will return 0. (Since the child starts out marked as not runnable, `sys_exofork` will not actually return in the child until the parent has explicitly allowed this by marking the child runnable using....)

`sys_env_set_status`:

Sets the status of a specified environment to `ENV_RUNNABLE` or `ENV_NOT_RUNNABLE`. This system call is typically used to mark a new environment ready to run, once its address space and register state has been fully initialized.

`sys_page_alloc`:

Allocates a page of physical memory and maps it at a given virtual address in a given environment's address space.

`sys_page_map`:

Copy a page mapping (*not* the contents of a page!) from one environment's address space to another, leaving a memory sharing arrangement in place so that the new and the old mappings both refer to the same page of physical memory.

`sys_page_unmap`:

Unmap a page mapped at a given virtual address in a given environment.

For all of the system calls above that accept environment IDs, the JOS kernel supports the convention that a value of 0 means "the current environment." This convention is implemented by `envid2env()` in `kern/env.cc`.

We have provided a very primitive implementation of a Unix-like `fork()` in the test program `user/dumbfork.c`. This test program uses the above system calls to create and run a child environment with a copy of its own address space. The two environments then switch back and forth using `sys_yield` as in the previous exercise. The parent exits after 10 iterations, whereas the child exits after 20.

**Exercise 2.** Implement the system calls described above in `kern/syscall.c`. You will need to use various functions in `kern/pmap.c` and `kern/env.cc`, particularly `envid2env()`. For now, whenever you call `envid2env()`, pass 1 in the `checkperm` parameter. Be sure you check for any invalid system call arguments, returning `-E_INVALID` in that case. Test your JOS kernel with `user/dumbfork` and make sure it works before proceeding.

*Challenge!* Add the additional system calls necessary to *read* all of the vital state of an existing environment as well as set it up. Then implement a user mode program that forks off a child environment, runs it for a while (e.g., a few iterations of `sys_yield()`), then takes a complete snapshot or *checkpoint* of the child environment, runs the child for a while longer, and finally restores the child environment to the state it was in at the checkpoint and continues it from there. Thus, you are effectively "replaying" the execution of the child environment from an intermediate state. Make the child environment perform some interaction with the user using `sys_cgetc()` or `readline()` so that the user can view and mutate its internal state, and verify that with your checkpoint/restart you can give the child environment a case of selective amnesia, making it "forget" everything that happened beyond a certain point.

This completes Part A of the lab; hand it in using `make handin` as usual.

## Part B: Copy-on-Write Fork

As mentioned earlier, Unix provides the `fork()` system call as its primary process creation primitive. The `fork()` system call copies the address space of the calling process (the parent) to create a new process (the child).

xv6 Unix implements `fork()` by copying the parent's entire data segment into a new memory region allocated for the child. This is essentially the same approach that `dumbfork()` takes. The copying of the parent's address into the child is the most expensive part of the `fork()` operation.

However, a call to `fork()` is frequently followed almost immediately by a call to `exec()` in the child process, which replaces the child's memory with a new program. This is what the shell typically does, for example. In this case, the time spent copying the parent's address space is largely wasted, because the child process will use very little of its memory before calling `exec()`.

For this reason, later versions of Unix took advantage of virtual memory hardware to allow the parent and child to *share* the memory mapped into their respective address spaces until one of the processes actually modifies it. This technique is known as *copy-on-write*. To do this, on `fork()` the kernel would copy the address space *mappings* from the parent to the child instead of the contents of the mapped pages, and at the same time mark the now-shared pages read-only. When one of the two processes tries to write to one of these shared pages, the process takes a page fault. At this point, the Unix kernel realizes that the page was really a "virtual" or "copy-on-write" copy, and so it makes a new, private copy of the page for the faulting process. In this way, the contents of individual pages aren't actually copied until they are actually written to. This optimization makes a `fork()` followed by an `exec()` in the child much cheaper: the child will probably only need to copy one page (the current page of its stack) before it calls `exec()`.

In the next piece of this lab, you will implement a "proper" Unix-like `fork()` with copy-on-write, as a user space library routine. Implementing `fork()` and copy-on-write support in user space has the benefit that the kernel remains much simpler and thus more likely to be correct. It also lets individual user-mode programs define their own semantics for `fork()`. A program that wants a slightly different implementation (for example, the expensive always-copy version like `dumbfork()`, or one in which the parent and child actually share memory afterward) can easily provide its own.

## User-level page fault handling

A user-level copy-on-write `fork()` needs to know about page faults on write-protected pages, so that's what you'll implement first. Copy-on-write is only one of many possible uses for user-level page fault handling.

It's common to set up an address space so that page faults indicate when some action needs to take place. For example, most Unix kernels initially map only a single page in a new process's stack region, and allocate and map additional stack pages later "on demand" as the process's stack consumption increases and causes page faults on stack addresses that are not yet mapped. A typical Unix kernel must keep track of what action to take when a page fault occurs in each region of a process's space. For example, a fault in the stack region will typically allocate and map new page of physical memory. A fault in the program's BSS region will typically allocate a new page, fill it with zeroes, and map it. In systems with demand-paged executables, a fault in the text region will read the corresponding page of the binary off of disk and then map it.

This is a lot of information for the kernel to keep track of. Instead of taking the traditional Unix approach, you will decide what to do about each page fault in user space, where

bugs are less damaging. This design has the added benefit of allowing programs great flexibility in defining their memory regions; you'll use user-level page fault handling later for mapping and accessing files on a disk-based file system.

## Setting the Page Fault Handler

In order to handle its own page faults, a user environment will need to register a *page fault handler entrypoint* with the JOS kernel. The user environment registers its page fault entrypoint via the new `sys_env_set_pgfault_upcall` system call. We have added a new member to the `Env` structure, `env_pgfault_upcall`, to record this information.

**Exercise 4.** Implement the `sys_env_set_pgfault_upcall` system call. Be sure to enable permission checking when looking up the environment ID of the target environment, since this is a "dangerous" system call.

## Normal and Exception Stacks in User Environments

During normal execution, a user environment in JOS will run on the *normal* user stack: its `ESP` register starts out pointing at `USTACKTOP`, and the stack data it pushes resides on the page between `USTACKTOP-PGSIZE` and `USTACKTOP-1` inclusive. When a page fault occurs in user mode, however, the kernel will restart the user environment running a designated user-level page fault handler on a different stack, namely the *user exception* stack. In essence, we will make the JOS kernel implement automatic "stack switching" on behalf of the user environment, in much the same way that the *x86 processor* already implements stack switching on behalf of JOS when transferring from user mode to kernel mode!

The JOS user exception stack is also one page in size, and its top is defined to be at virtual address `UXSTACKTOP`, so the valid bytes of the user exception stack are from `UXSTACKTOP-PGSIZE` through `UXSTACKTOP-1` inclusive. While running on this exception stack, the user-level page fault handler can use JOS's regular system calls to map new pages or adjust mappings so as to fix whatever problem originally caused the page fault. Then the user-level page fault handler returns, via an assembly language stub, to the faulting code on the original stack.

Each user environment that wants to support user-level page fault handling will need to allocate memory for its own exception stack, using the `sys_mem_alloc()` system call introduced in part A.

## Invoking the User Page Fault Handler

You will now need to change the page fault handling code in `kern/trap.c` to handle page faults from user mode as follows. We will call the state of the user environment at the time of the fault the *trap-time* state.

If there is no page fault handler registered, the JOS kernel destroys the user environment with a message as before. Otherwise, the kernel sets up a trap frame on the exception stack that looks like a `struct UTrapframe` from `lib/trap.h`:

```

                                <-- UXSTACKTOP
trap-time esp
trap-time eflags
trap-time eip
trap-time eax          start of struct PushRegs
trap-time ecx
trap-time edx
trap-time ebx
trap-time esp
trap-time ebp
trap-time esi
trap-time edi          end of struct PushRegs
tf_err (error code)
fault_va              <-- %esp when handler is run
```

The kernel then arranges for the user environment to resume execution with the page fault handler running on the exception stack with this stack frame; you must figure out how to make this happen. The `fault_va` is the virtual address that caused the page fault.

If the user environment is *already* running on the user exception stack when an exception occurs, then the page fault handler itself has faulted. In this case, you should start the new stack frame just under the current `tf->tf_esp` rather than at `UXSTACKTOP`. You should first push an empty 32-bit word, then a `struct UTrapframe`.

To test whether `tf->tf_esp` is already on the user exception stack, check whether it is in the range between `UXSTACKTOP-PGSIZE` and `UXSTACKTOP-1`, inclusive.

**Exercise 5.** Implement the code in `kern/trap.c` required to dispatch page faults the user-mode handler. Be sure to take appropriate precautions when writing into the exception stack. (What happens if the user environment runs out of space on the exception stack?)

## User-mode Page Fault Entrypoint

Next, you need to implement the assembly routine that will take care of calling the C page fault handler and resume execution at the original faulting instruction. This assembly routine is the handler that will be registered with the kernel using `sys_env_set_pgfault_upcall()`.

**Exercise 6.** Implement the `_pgfault_upcall` routine in `lib/pfentry.S`. The interesting part is returning to the original point in the user code that caused the page fault. You'll return directly there, without going back through the kernel. The hard part is simultaneously



switching stacks and re-loading the EIP.

Finally, you need to implement the C user library side of the user-level page fault handling mechanism.

**Exercise 7.** Finish `set_pgfault_handler()` in `lib/pgfault.c`.

## Testing

Change `kern/init.c` to run `user/faultread`. Build your kernel and run it. You should see:

```
[00000000] new env 00001000
[00000000] new env 00001001
[00001001] user fault va 00000000 ip 0080003a
TRAP frame ...
[00001001] free env 00001001
```

Change `kern/init.c` to run `user/faultdie`. Build your kernel and run it. You should see:

```
[00000000] new env 00001000
[00000000] new env 00001001
i faulted at va deadbeef, err 6
[00001001] exiting gracefully
[00001001] free env 00001001
```

Change `kern/init.c` to run `user/faultalloc`. Build your kernel and run it. You should see:

```
[00000000] new env 00001000
[00000000] new env 00001001
fault deadbeef
this string was faulted in at deadbeef
fault cafebffe
fault cafec000
this string was faulted in at cafebffe
[00001001] exiting gracefully
[00001001] free env 00001001
```

If you see only the first "this string" line, it means you are not handling recursive page faults properly.

Change `kern/init.c` to run `user/faultallocbad`. Build your kernel and run it. You should see:

```
[00000000] new env 00001000
[00000000] new env 00001001
[00001001] user_mem_check assertion failure for va deadbeef
TRAP frame ...
```

[00001001] free env 00001001  
(Your ip may differ from ours but should begin f01.)

Make sure you understand why `user/faultalloc` and `user/faultallocbad` behave differently.

*Challenge!* Extend your kernel so that not only page faults, but *all* types of processor exceptions that code running in user space can generate, can be redirected to a user-mode exception handler. Write user-mode test programs to test user-mode handling of various exceptions such as divide-by-zero, general protection fault, and illegal opcode.

## Implementing Copy-on-Write Fork

You now have the kernel facilities to implement copy-on-write `fork()` entirely in user space.

We have provided a skeleton for your `fork()` in `lib/fork.c`. Like `dumbfork()`, `fork()` creates a new environment, then scans through the parent environment's entire address space and sets up corresponding page mappings in the child. The key difference is that, while `dumbfork()` copied *pages*, `fork()` will initially only copy page *mappings*. `fork()` will copy each page only when one of the environments tries to write it.

The basic control flow for `fork()` is as follows:

1. The parent installs `pgfault()` as the C-level page fault handler, using the `set_pgfault_handler()` function you implemented above.
2. The parent calls `sys_exofork()` to create a child environment.
3. For each writable or copy-on-write page in its address space below `UTOP`, the parent maps the page copy-on-write into the address space of the child and then *remaps* the page copy-on-write in its own address space. The parent sets both PTEs so that the page is not writeable, and to contain `PTE_COW` in the "avail" field to distinguish copy-on-write pages from genuine read-only pages.

The exception stack is *not* remapped this way, however. Instead you need to allocate a fresh page in the child for the exception stack. Since the page fault handler will be doing the actual copying and the page fault handler runs on the exception stack, the exception stack cannot be made copy-on-write: who would copy it?

4. The parent sets the user page fault entrypoint for the child to look like its own.
5. The child is now ready to run, so the parent marks it runnable.

Each time one of the environments writes a copy-on-write page that it hasn't yet written, it will take a page fault. Here's the control flow for the user page fault handler:

1. Kernel propagates page fault to `_pgfault_upcall`, which calls `fork()`'s `pgfault()` handler.
2. `pgfault()` checks that the fault is a write (check `FEC_WR`) and that the PTE for the page is marked `PTE_COW`. If not, panic.
3. `pgfault()` allocates a new page mapped at a temporary location and copies the contents of the faulting page contents into it. Then the fault handler maps the new page at the appropriate address with read/write permissions, in place of the old read-only mapping.

**Exercise 8.** Implement `fork` and `pgfault` in `lib/fork.c`.

Test your code with the `forktree` program. It should produce the following messages, with interspersed 'new env', 'free env', and 'exiting gracefully' messages. The messages may not appear in this order, and the environment IDs may be different.

```
1001: I am ''
1802: I am '0'
2801: I am '00'
3802: I am '000'
2003: I am '1'
5001: I am '11'
4802: I am '10'
6801: I am '100'
5803: I am '110'
3004: I am '01'
8001: I am '011'
7803: I am '010'
4005: I am '001'
6006: I am '111'
7007: I am '101'
```

*Challenge!* Implement a shared-memory `fork()` called `sfork()`. This version should have the parent and child *share* all their memory pages (so writes in one environment appear in the other) except for pages in the stack area, which should be treated in the usual copy-on-write manner. Modify `user/forktree.c` to use `sfork()` instead of regular `fork()`. Also, once you have finished implementing IPC in part C, use your `sfork()` to run `user/pingpongs`. You will have to find a new way to provide the functionality of the global `env` pointer.

*Challenge!* Your implementation of `fork` makes a huge number of system calls. On the x86, switching into the kernel has non-trivial cost. Augment the system call interface so that it is possible to send a batch of system calls at once. Then change `fork` to use this interface.

How much faster is your new `fork`?

You can answer this (roughly) by using analytical arguments to estimate how much of an improvement batching system calls will make to the performance of your `fork`: How expensive is an `int 0x30` instruction? How many times do you execute `int 0x30` in your `fork`? Is accessing the `TSS` stack switch also expensive? And so on...

Alternatively, you can boot your kernel on real hardware and *really* benchmark your code. See the `RDTSC` (read time-stamp counter) instruction, defined in the IA32 manual, which counts the number of clock cycles that have elapsed since the last processor reset. `Bochs` doesn't emulate this instruction faithfully.

This ends part B. As usual, you can grade your submission with `gmake grade` and hand it in with `gmake handin`.

## Part C: Preemptive Multitasking and Inter-Process communication (IPC)

In the final part of lab 4 you will modify the kernel to preempt uncooperative environments and to allowing environments to pass messages to each other explicitly.

### Clock Interrupts and Preemption

Modify `kern/init.c` to run the `user/spin` test program. This test program forks off a child environment, which simply spins forever in a tight loop once it receives control of the CPU. Neither the parent environment nor the kernel ever regains the CPU. This is obviously not an ideal situation in terms of protecting the system from bugs or malicious code in user-mode environments, because any user-mode environment can bring the whole system to a halt simply by getting into an infinite loop and never giving back the CPU. In order to allow the kernel to *preempt* a running environment, forcefully retake control of the CPU from it, we must extend the JOS kernel to support external hardware interrupts from the clock hardware.

### Interrupt discipline

External interrupts (i.e., device interrupts) are referred to as IRQs. There are 16 possible IRQs, numbered 0 through 15. The mapping from IRQ number to IDT entry is not fixed. `Pic_init` in `picirq.c` maps IRQs 0-15 to IDT entries `IRQ_OFFSET` through `IRQ_OFFSET+15`.

In `kern/picirq.h`, `IRQ_OFFSET` is defined to be decimal 32. Thus the IDT entries 32-47 correspond to the IRQs 0-15. For example, the clock interrupt is IRQ 0. Thus, `IDT[32]` contains the address of the clock's interrupt handler routine in the kernel. This `IRQ_OFFSET` is chosen so that the device interrupts do not overlap with the processor

exceptions, which could obviously cause confusion. (In fact, in the early days of PCs running MS-DOS, the `IRQ_OFFSET` effectively *was* zero, which indeed caused massive confusion between handling hardware interrupts and handling processor exceptions!)

In JOS, we make a key simplification compared to xv6 Unix. External device interrupts are *always* disabled when in the kernel (and, like xv6, enabled when in user space). External interrupts are controlled by the `FL_IF` flag bit of the `%eflags` register (see `inc/mmu.h`). When this bit is set, external interrupts are enabled. While the bit can be modified in several ways, because of our simplification, we will handle it solely through the process of saving and restoring `%eflags` register as we enter and leave user mode.

You will have to ensure that the `FL_IF` flag is set in user environments when they run so that when an interrupt arrives, it gets passed through to the processor and handled by your interrupt code. Otherwise, interrupts are *masked*, or ignored until interrupts are re-enabled. Interrupts are masked by default after processor reset, and so far we have never gotten around to enabling them.

**Exercise 9.** Modify `kern/trapentry.S` and `kern/trap.c` to initialize the appropriate entries in the IDT and provide handlers for IRQs 0 through 15. Then modify the code in `env_alloc()` to ensure that user environments are always run with interrupts enabled.

The processor never pushes an error code or checks the Descriptor Privilege Level (DPL) of the IDT entry when invoking a hardware interrupt handler. You might want to re-read section 9.2 of the 80386 Reference Manual, or section 5.8 of the IA-32 Intel Architecture Software Developer's Manual, Volume 3 (see readings), at this time.

After doing this exercise, if you run your kernel with any test program that runs for a non-trivial length of time (e.g., `dumbfork`), you should see a kernel panic shortly into the program's execution. This is because our code has set up the clock hardware to generate clock interrupts, and interrupts are now enabled in the processor, but JOS isn't yet handling them.

## Handling Clock Interrupts

In the `user/spin` program, after the child environment was first run, it just spun in a loop, and the kernel never got control back. We need to program the hardware to generate clock interrupts periodically, which will force control back to the kernel where we can switch control to a different user environment.

The calls to `pic_init` and `kclock_init` (from `i386_init` in `init.c`), which we have written for you, set up the clock and the interrupt controller to generate interrupts. You now need to write the code to handle these interrupts.

**Exercise 10.** Modify the kernel's `trap_dispatch()` function so that it calls `sched_yield()` to find and run a different environment whenever a clock interrupt takes place.

You should now be able to get the `user/spin` test to work: the parent environment should fork off the child, `sys_yield()` to it a couple times but in each case regain control of the CPU after one time slice, and finally kill the child environment and terminate gracefully.

Make sure you can answer the following questions:

1. How many instructions of user code are executed between each interrupt?
  2. How many instructions of kernel code are executed to handle the interrupt?
- Hint: use the `vb` command mentioned earlier.

This is a great time to do some *regression testing*. Make sure that you haven't broken any earlier part of that lab that used to work (e.g. `forktree`) by enabling interrupts. Run `gmake grade` to see for sure. You should now get 55/65 points on the lab.

## Inter-Process communication (IPC)

(Technically in JOS this is "inter-environment communication" or "IEC", but everyone else calls it IPC, so we'll use the standard term.)

We've been focusing on the isolation aspects of the operating system, the ways it provides the illusion that each program has a machine all to itself. Another important service of an operating system is to allow programs to communicate with each other when they want to. It can be quite powerful to let programs interact with other programs. The Unix pipe model is the canonical example.

There are many models for interprocess communication. Even today there are still debates about which models are best. We won't get into that debate. Instead, we'll implement a simple IPC mechanism and then try it out.

### IPC in JOS

You will implement a few additional JOS kernel system calls that collectively provide a simple interprocess communication mechanism. You will implement two system calls, `sys_ipc_recv` and `sys_ipc_can_send`. Then you will implement two library wrappers `ipc_recv` and `ipc_send`.

The "messages" that user environments can send to each other using JOS's IPC mechanism consist of two components: a single 32-bit value, and optionally a single page mapping. Allowing environments to pass page mappings in messages provides an

efficient way to transfer more data than will fit into a single 32-bit integer, and also allows environments to set up shared memory arrangements easily.

## **Sending and Receiving Messages**

To receive a message, an environment calls `sys_ipc_recv`. This system call de-schedules the current environment and does not run it again until a message has been received. When an environment is waiting to receive a message, *any* other environment can send it a message - not just a particular environment, and not just environments that have a parent/child arrangement with the receiving environment. In other words, the permission checking that you implemented in Part A will not apply to IPC, because the IPC system calls are carefully designed so as to be "safe": an environment cannot cause another environment to malfunction simply by sending it messages (unless the target environment is also buggy).

To try to send a value, an environment calls `sys_ipc_can_send` with both the receiver's environment id and the value to be sent. If the named environment is actually receiving (it has called `sys_ipc_recv` and not gotten a value yet), then the send delivers the message and returns 0. Otherwise the send returns `-E_IPC_NOT_RECV` to indicate that the target environment is not currently expecting to receive a value.

A library function `ipc_recv` in user space will take care of calling `sys_ipc_recv` and then looking up the information about the received values in the current environment's `struct Env`.

Similarly, a library function `ipc_send` will take care of repeatedly calling `sys_ipc_can_send` until the send succeeds.

## **Transferring Pages**

When an environment calls `sys_ipc_recv` with a nonzero `dstva` parameter, the environment is stating that it is willing to receive a page mapping. If the sender sends a page, then that page should be mapped at `dstva` in the receiver's address space. If the receiver already had a page mapped at `dstva`, then that previous page is unmapped.

When an environment calls `sys_ipc_can_send` with a nonzero `srcva`, it means the sender wants to send the page currently mapped at `srcva` to the receiver, with permissions `perm`. After a successful IPC, the sender keeps its original mapping for the page at `srcva` in its address space, but the receiver also obtains a mapping for this same physical page at the `dstva` originally specified by the receiver, in the receiver's address space. As a result this page becomes shared between the sender and receiver.

If either the sender or the receiver does not indicate that a page should be transferred, then no page is transferred. After any IPC the kernel sets the new field `env_ipc_perm` in the receiver's `Env` structure to the permissions of the page received, or zero if no page was received.

## Implementing IPC

**Exercise 11.** Implement `sys_ipc_recv` and `sys_ipc_can_send` in `kern/syscall.c`. When you call `envid2env` in these routines, you should set the `checkperm` flag to 0, meaning that any environment is allowed to send IPC messages to any other environment, and the kernel does no special permission checking other than verifying that the target `envid` is valid.

Then implement the `ipc_recv` and `ipc_send` functions in `lib/ipc.c`.

Use the `user/pingpong` and `user/primes` functions to test your IPC mechanism. You might find it interesting to read `user/primes.c` to see all the forking and IPC going on behind the scenes.

*Challenge!* The `ipc_send` function is not very fair. Run three copies of `user/fairness` and you will see this problem. The first two copies are both trying to send to the third copy, but only one of them will ever succeed. Make the IPC fair, so that each copy has approximately equal chance of succeeding.

*Challenge!* Why does `ipc_send` have to loop? Change the system call interface so it doesn't have to. Make sure you can handle multiple environments trying to send to one environment at the same time.

*Challenge!* The prime sieve is only one neat use of message passing between a large number of concurrent programs. Read C. A. R. Hoare, "Communicating Sequential Processes," *Communications of the ACM* 21(8) (August 1978), 666-667, and implement the matrix multiplication example.

*Challenge!* Probably the most impressive example of the power of message passing is Doug McIlroy's power series calculator, described in [M. Douglas McIlroy, "Squinting at Power Series," \*Software--Practice and Experience\*, 20\(7\) \(July 1990\), 661-683.](#) Implement his power series calculator and compute the power series for  $\sin(1+x^2)$ .

*Challenge!* Make JOS's IPC mechanism more efficient by applying some of the techniques from Liedtke's paper, "Improving IPC by Kernel Design", or any other tricks you may think of. Feel free to modify the kernel's system call API for this purpose, as long as your code is backwards compatible with what our grading scripts expect. -->

*Challenge!* Generalize the JOS IPC interface so it is more like L4's, supporting more complex message formats.

This ends part C. As usual, you can grade your submission with `gmake grade` and hand it in with `gmake handin`. If you are trying to figure out why a particular test case is failing,



run `sh grade.sh -v`, which will show you the output of the kernel builds and Bochs runs for each test, until a test fails. When a test fails, the script will stop, and then you can inspect `bochs.out` to see what the kernel actually printed.

---