An .exe file as created for the emulator contains no special formatting or extra information, it is just an *image* of what should be in memory when your program starts. Unless you have done anything strange, it should be completely *position independent*, which means that although the system expects programs to start at address 0x400, you should be able to load them starting at any address, and just set the PC (i.e. `JUMP`) to that address.

Your file system should be able to load real unix files (such as the .exe from a BCPL program) into itself, so that your programs have direct access to them as files on your emulated disc. If your file system isn't working, you can still read the .exe files directly using the devctl() function's dc_tape_load and dc_tape_read operations.

Try it. Load an .exe file created by the BCPL compiler into memory at a location that you know exists but isn't already in use, and then jump to it. Remember that you can embed assembly language in BCPL programs like this `assembly { jump 0x1000 }`.

Every .obj or .exe file created by the BCPL compiler starts with these exact four words:

```
push  0
call  g_start
add   SP, 1
halt
```

This arranges for the `start()` function to be called the same way as any zero parameter function is called, and stops the processor when `start()` returns. This means that even if your code start the loaded program with a `CALL` instruction instead of the `JUMP`, control will not return to it when the loaded program ends.

However, you know that the second word of an loaded .exe file will be a `CALL` instruction that jumps to the true address of the `start()` function, and that can be exploited. The BCPL compiler uses relative addressing for all jumps. This means that the `CALL` instruction will be in this form:

```
call  [PC + 123]
```

where 123 is replaced by the number of words between the next instruction and the beginning of the start function. Remembering that an instruction's numeric operand (e.g. 123) is stored in the least significant 16 bits of the instruction word, it can easily be extracted. If `loadaddr` is the address where you loaded an .exe file into memory, then

```
startaddr := loadaddr + 2 +
             ((loadaddr ! 1) bitand 0xFFFF);
startaddr();
out("I'm back!\n");
```

will call the loaded program as though it were a function.
Being able to call loaded programs as though they were functions is useful for understanding, but not of great practical importance. When you create a new process, it is usually allowed to run to completion in user mode and then halt.

The halt in user mode causes an interrupt, and it is that interrupt that allows the operating system to regain control. Being able to treat a loaded program as a function doesn't really add much.

The 16 bit limitation on the distance over which a call or jump can be made (i.e. -32768 to +32767 words) is important to remember. It makes the use of *jump tables*, as with the SYSCALL instruction or in the discussion of dynamically loaded libraries very important. Jump tables are arrays of actual addresses in full 32 bit words, not just 16 bit distances, and a pointer to a jump table is an ordinary 32 bit value stored in a variable. Jump tables mean you can get from anywhere to anywhere.

This might be a good time to set up some system calls. You will need them soon, and you want to be confident that you know how to make them before things get more complicated. The call gate base register CGBR works just like INTVEC, it holds the physical address of the beginning of the call gate vector, which holds the virtual addresses of the system calls. Unlike with interrupts, there can be any number of call gates, so you must also set CGLEN. The syscall instruction causes an error if the function code is <0 or >=CGLEN.

Just a few simple syscalls will get you started, maybe one that returns the time of day, and one that shuts down the whole system (so that when you include your file system, it'll have a chance to save all the important things).

This sample shows how syscalls can be made reasonably convenient. The first function provides a BCPL interface to any established syscall. If you want the result of syscall number 7 when given the parameters 11, 22, and 33, you would say

```
x := callsysc(7, 11, 22, 33);
```

and think about what would be on the stack when callsysc starts:

33	last parameter
22	
11	
7	the syscall code, first parameter
8	2 * numbargs()
return address	
saved frame pointer	

```
let callsysc(code, arg1) be
{ let thecode = code,
      ptr = @code;
  ptr ! 0 := numbargs() - 1;
  assembly
  { load r1, [<ptr>]
    syscall r1, [<thecode>] }
  resultis ptr ! 0 }
```

The SYSCALL instruction pushes an enormous number of things on the stack, and that makes accessing callsysc's useful parameters a nuisance. So the plan here is to take a pointer to the first parameter, @code, which will serve as a pointer to the vector of parameters, and give that directly to SYSCALL.

First, the value of the `code` parameter is kept safe in a local variable (`thecode`), and then replaced in the parameter array by the number of useful parameters the system call will receive. If the system call needs to return a result, it simply puts it in that same place so that `callsysc` can return it to the caller.

The system call itself (`sysc1`, below) receives from the `SYSCALL` instruction three particularly useful parameters: the syscall code, the register number that appeared in the instruction, and the value of that register. So `rv` will be the pointer to the vector of parameters originally given to `callsysc`.

This example system call just lists all of its parameters, then returns as its result the product of the first two.

```
let sysc1(code, rn, rv) be
{ let n = rv ! 0;
  out("sysc1: %d arguments:", n);
  for i=1 to n do
    out(" %d", rv!i);
  rv ! 0 := rv ! 1 * rv ! 2;
  out("\n       returning %d\n", rv ! 0);
  ireturn }
```

It is certainly a good idea to give the syscall codes meaningful names, and if you want you can create wrapper functions so that they can be called in exactly the same way as a normal function:

```
manifest { sys_multiply = 1 }

let multiply(a, b) = callsysc(sys_multiply, a, b)
```

This sets up the call gate vector and installs all (one) of the syscall functions:

```
let setup_syscalls(cgv) be
{ cgv ! 0 := 0;
  cgv ! sys_multiply := sysc1;
  assembly
  { load  r1, [<cgv>]
    setsr r1, $cgbr
    load  r1, 2
    setsr r1, $cglen } }
```

and this uses it.

```
let start() be
{ let cgv = vec 10, x;
  setup_syscalls(cgv);
  x := callsysc(sys_multiply, 123, 10001);
  out("123 * 10001 was %d\n", x);
  x := multiply(9, 8);
  out("9 * 8 was %d\n", x) }
```

Now remember all those experiments in classes 18, 19, and 22 that led to running a program in user mode under virtual memory. Make a simple adaptation. Create an entirely new virtual address space, with its page directory and all of its pages and page tables completely fresh. Load the .exe file where it is expected, starting at 0x400 in this new virtual address space. Then turn on virtual memory with a simultaneous jump to 0x400.

With that, the loaded .exe file will be running almost as a real process. Unfortunately, when it finishes there is no operating system to regain control, so everything will just stop. Or more likely crash.

It is quite likely that your file system makes use of `newvec` and `freevec`, in which case you'll have to add a few extra pages to allow a heap to be created. Remember that the values in `0x100` and `0x101` will not be meaningful any more, so you'll need to think of a new way to tell your processes how much memory they've got.

Never mind about everything crashing when the program ends, it is still a major step forwards. Your main program can now be made into a very sad and limited shell (command processor). It can print a prompt and wait for you to type a command. You can put in some simple commands such as one to tell the user what time it is, and one that uses your file system to produce a directory listing. More importantly, you can have a command for loading and running any .exe file on your disc, then crashing.

The next step then is to convert your little command shell program into a mini operating system, and that isn't too difficult. Recall that a process is expected to have four separate regions of virtual memory: user static (code + globals + heap), user stack, system static, and system stack.

Make your command shell program map itself as the system static region of the new process and give it a bit of system stack space. Put in the interrupt handlers (especially for halts), and make it happen. You will have to do some careful thinking to determine which virtual address to jump to for continuity when virtual memory is turned on, but it isn't difficult.

With all that working, it is time to make a slow test program, one that will run for long enough to notice, without swallowing up all of the CPU time:

```
import "io"

let idle(n) be
  for i=1 to n do
    assembly { pause }

let start() be
{ out("start\n");
  for i=1 to 20 do
  { idle(20);
    out("%d\n", i) }
  out("done\n") }
```

A `pause` instruction causes a delay of just over 50mS, so you should definitely see a slow progress through the numbers 1 to 20. Slow enough that you can set up a timer interrupt that occurs four or five times between each number being printed. Make sure that your timer interrupt handler is in the system code area (i.e. part of your mini-OS), and remember that `INTVEC` holds the physical address of the interrupt vector, but the entries in the vector are the virtual addresses of the interrupt handlers.

To start with, the timer interrupt handler can be as simple as just printing a dot. You want to be sure that the basics are working. But then it will be time to make the interrupt handler do something useful so you can really be sure your OS is taking over.

This would be a good time to set up some more system calls. something nice and simple is to make your OS keep track of the real time of day, in an efficient way. The advanced hardware documentation tells you about two `peri` operations: `SECONDS` and `DATETIME`. These tell you the real time of day. It would be inefficient to use one of them on every timer interrupt. A much better plan is to find out the time once at start up, and just periodically update it. If your interrupt handler is ticking five times per second, it might be sensible to just update the time on every fifth tick. The time information would be kept in system memory, and your new `syscall` would be to give the user program a copy of that information.

Try it out. Make the loop in start use the system call each time so that it prints the current time instead of the value of `i`, it will look like an actual achievement!

The next step is quite simple, but makes things look much more real. Put in an interrupt handler for user mode `HALT`s. When a user program halts, that is a clear signal that it should stop existing. That is easy to handle: just collect up all the pages of memory that were allocated for its user mode page tables and stack and code, and put them back into the system's free page list, then get back to running your system mode shell. The shell can accept another command to run another program, over and over again. Then it will really look like a real operating system.

But you might not want to do it exactly that way. If a program fails, it is very useful to be able to see what it was doing. It is easy to build in shell commands for looking at memory locations and so on, but if the memory isn't there any more it won't help. Perhaps instead you should wait until the next command to run a program, or an explicit clear command, before deallocating the user memory area.

At this stage there are a number of different things that could be done next. Getting input from the user is the easiest to get out of the way.

Although this isn't multi-processing yet, there are now two distinct active entities that you need to communicate with: the mini-OS, and the user-mode program that may or may not be running at any given time. Obviously commands to each of them should not get mixed up.

You have already made an interrupt based keyboard input system. Incorporate it into your mini-OS. But adapt it so that it has more than one character buffer: one for communicating with the OS, and one for each of the user processes.

Pick on some of the under-achievers on the keyboard, such as F1, F2, ... or ctrl-A, ctrl-B, ... and give them special meanings. As soon as one of them is received, the keyboard interrupt system takes it as a signal to change which buffer it is putting received characters in, and the problem almost solves itself. Be careful with the special characters you choose though, because some control characters already have essential functions (ctrl-C=stop, ctrl-H= backspace, ctrl-J=newline, ctrl-M sometimes=enter, ctrl-Z=suspend). And you will almost certainly want some quick way of telling the OS that it should stop a running program immediately.

By now, all the groundwork is done, and it is time to go for real multi-processing. Think about what the OS needs to remember for each process, and how it should be stored. It won't be very much yet.

- The page directory takes up half a page (1024 words), and that tells you where all the page tables are, and they tell you which physical pages are in use.
- When a process isn't actively running, all of its volatile state must be saved somewhere. On an interrupt, all the volatile state is automatically pushed onto the system stack, and that only amounts to 20 words. Of course, they can't be left cluttering up the system stack.
- Each process needs to have its own character buffer, and its own index variables to make the buffer work as a queue.
- Each process needs to know its own identity (a PID perhaps), and its own state: is it really ready to run, or is it waiting for something?
- You'll probably discover a few more essentials.

but all of that doesn't really add up to very much. You can probably get away with giving each process one whole 2048 word page to act as its Process Control Block, and make the OS keep a short vector of pointers to all existing PCBs.

So how do you create a new process? Easy. Allocate a new PCB page, construct the new process' page directory in that page, not altering the PDBR yet, prefill all the other entries with appropriate values (initial register values, etc), and as the final step, set its state to "runnable".

What happens when the timer interrupt is detected? Remember the 20 words of volatile state will automatically be on the system stack. Look to see if there are any other runnable processes, if there aren't then just do the usual thing (updating the time, etc) and allow the interrupted process to continue.

If there is another runnable process, then copy the 20 words of volatile state from the system stack into the current process' PCB, and copy the new process' 20 words from its PCB back into the same locations on the system stack, and change over the PDBR values. Now an `IRET` or `ireturn` will will "return" back to executing the new process as though nothing had happened.

With the right preparation it was quite easy in the end.

Of course, there are a few loose ends to be tied up.

`Newvec` and `freevec` will need a very slight modification. A newly created process will not have all of memory available to make its heap out of. Instead, you can give each new process just a few extra pages where its heap should be, just enough for it to set up its essential structures. When a call to `newvec` finds there isn't enough heap to satisfy it, it must ask the OS to add a few new free pages to the end of its heap area so that it can enlarge. This request would of course be made through a new syscall.

It is not reasonable to continue to use PAUSE instructions to slow down programs, it is just a waste of time when there may be other programs wanting to run. PAUSEs should be replaced by a new syscall that allows a process to voluntarily surrender its turn on the CPU before the timer interrupt occurs. The process' status would be set to "not runnable" until the appropriate number of timer ticks have elapsed.

Even more importantly, waits for user input must also be dealt with differently. If a process tries to read keyboard input when none is available, it must be put to sleep, and automatically reawoken when input becomes available.