www.jamesmolloy.co.uk

Home » JamesM's kernel development tutorials

9. Multitasking

Eventually most people want to have their OS run two things (seemingly) at once. This is called multitasking, and is in my opinion one of the final hurdles before you can call your project an 'operating system' or 'kernel'.

9.1. Tasking theory

Firstly a quick recap; A CPU (with one core) cannot run multiple tasks simultaneously. Instead we rely on switching tasks quickly enough that it seems to an observer that they are all running at the same time. Each task gets given a "timeslice" or a "time to live" in which to use the CPU and memory. That timeslice is normally ended by a timer interrupt which calls the scheduler.

It should be noted that in more advanced operating systems a process' timeslice will normally also be terminated when it performs a synchronous I/O operation, and in such operating systems (all but the most trivial) this is the normal case.

When the scheduler is called, it saves the stack and base pointers in a task structure, restores the stack and base pointers of the process to switch to, switches address spaces, and jumps to the instruction that the new task left off at the last time it was swapped.

This relies on several things:

1. *All the general purpose registers are already saved.* This happens in the IRQ handler, so is automatic.
2. *The task switch code can be run seamlessly when changing address spaces.* The task switch code should be able to change address spaces and then continue executing as if nothing happened. This means that the kernel code must be mapped in at the same place in all address spaces.

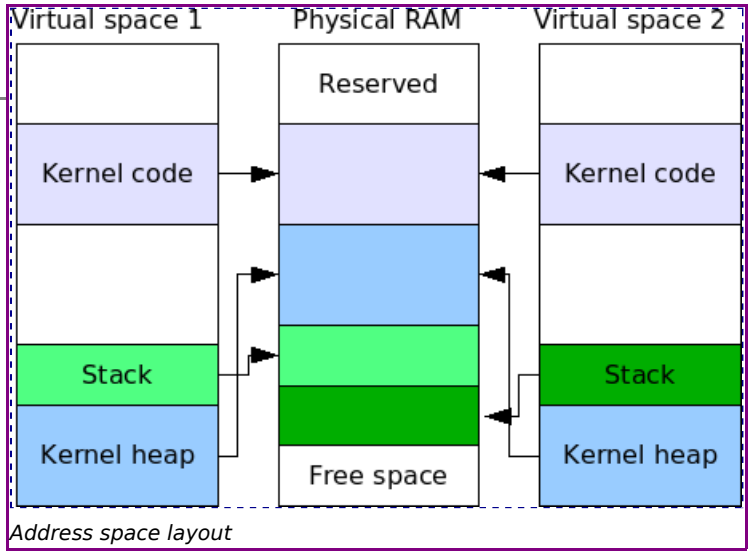
9.1.1. Some notes about address spaces

A lot of the complication in implementing multitasking is not just the context switching - a new address space must be created for each task. The complication is that some parts of the address space must be copied, and others must be linked. That is, two pages point to the same frame in physical memory. Take the example layout on the right - The picture shows two virtual address spaces and how areas are mapped to an example physical RAM layout.

The stack is indicative of most areas in a virtual address space: it is copied when a new process is forked, so that if the new process changes data the old process doesn't see the change. When we load executables, this will also be the case for the executable code and data.

The kernel code and heap areas are slightly different - both areas in both virtual memory spaces map to the same two areas of physical memory. Firstly there is no point in copying the kernel code as it will never change, and secondly it is important that the kernel heap is consistent in all address spaces - if task 1 does a system call and causes some data to be changed the kernel must be able to pick that up in task 2's address space.

9.2. Cloning an address space



Address space layout

So, as mentioned above, one of the most complex things we need to do is to create a copy of an address space - so let's get that over with first.

9.2.1. Cloning a directory

First off we need to create a new directory. We use our kmalloc_ap function to obtain an address aligned on a page boundary and to also retrieve the physical address. We then have to ensure that it is completely blank (each entry is initially zero).

```
page_directory_t *clone_directory(page_directory_t *src)
{
   u32int phys;
   // Make a new page directory and obtain its physical address.
   page_directory_t *dir = (page_directory_t*)kmalloc_ap(sizeof(page_directory_t), &phys);
   // Ensure that it is blank.
   memset(dir, 0, sizeof(page_directory_t));
```

We now have a new page directory, and the physical address at which it is located. However, for loading into the CR3 register, we need the physical address of the *tablesPhysical* member (remember that the physical address of a directory's page tables are held in *tablesPhysical*. See chapter 6). In order to do this, we perform a simple calculation. We get the offset of the *tablesPhysical* member from the start of the page_directory_t struct, then add that to the obtained physical address.

```
   // Get the offset of tablesPhysical from the start of the page_directory_t structure.
   u32int offset = (u32int)dir->tablesPhysical - (u32int)dir;

   // Then the physical address of dir->tablesPhysical is:
   dir->physicalAddr = phys + offset;
```

Now we're ready to copy each page table. If the page table is zero, we don't need to bother copying anything.

```
   int i;
   for (i = 0; i < 1024; i++)
   {
       if (!src->tables[i])
           continue;
```

Now we need a method of working out whether we should *link* a page table or *copy* it. Remember that we want to link the kernel code and heap, and copy everything else. Luckily, we already have a very simple method of finding out. The global variable *kernel_directory* is the first page directory we create. We identity map the kernel code and data, and map in the kernel heap all in this directory. Up until now, we've finished off the *initialise_paging* function with this:

```
current_directory == kernel_directory;
```

But, if instead we set current_directory to a *clone* of the kernel_directory, kernel_directory will remain constant, just containing the kernel code/data and the kernel heap. All modifications will be made to the clone, and not the original. This means that in our *clone* function we can compare page tables with the kernel_directory. If a page table in the directory we are cloning is *also in the kernel_directory*, we can assume that that page table should be linked. If not, it should be copied. Simple!

```
        if (kernel_directory->tables[i] == src->tables[i])
        {
           // It's in the kernel, so just use the same pointer.
           dir->tables[i] = src->tables[i];
           dir->tablesPhysical[i] = src->tablesPhysical[i];
        }
        else
        {
           // Copy the table.
           u32int phys;
           dir->tables[i] = clone_table(src->tables[i], &phys);
           dir->tablesPhysical[i] = phys | 0x07;
        }
```

Let's quickly go through that code segment. If the current page table is the same in the kernel directory and in the current directory, we link it - that is, in the new directory, we set the page table pointer to be the same as in the source directory. We also copy the physical address of this page table (very important - this is the address that matters to the processor). If, instead, we need to copy the table, we use an (as yet) undefined function called *clone_table*, which returns a virtual pointer to a page table, and stores its physical address in a passed-in argument. When setting the tablesPhysical pointer, we bitwise-OR the physical address with 0x07, which means "Present, Read-write, user-mode".

Let's just quickly end this function, then we can go on to define *clone_table*.

```
   }
   return dir;
}
```

9.2.2. Cloning a table

To clone a page table, we have to do something similar to above, with some changes. We never have to choose whether to copy or link table entries - we always copy. We also have to copy the data in the page table entries.

```
static page_table_t *clone_table(page_table_t *src, u32int *physAddr)
{
   // Make a new page table, which is page aligned.
   page_table_t *table = (page_table_t*)kmalloc_ap(sizeof(page_table_t), physAddr);
   // Ensure that the new table is blank.
   memset(table, 0, sizeof(page_directory_t));

   // For every entry in the table...
   int i;
   for (i = 0; i < 1024; i++)
   {
     if (!src->pages[i].frame)
       continue;
```

The preamble for this function is exactly the same as in *clone_directory*.

So, for every page table entry in the table, we need to:

- Allocate ourselves a new frame to hold the copied data.
- Copy the flags - read/write, present, user-mode etc.
- Physically copy the data.

```
      // Get a new frame.
      alloc_frame(&table->pages[i], 0, 0);
      // Clone the flags from source to destination.
      if (src->pages[i].present) table->pages[i].present = 1;
      if (src->pages[i].rw)      table->pages[i].rw = 1;
      if (src->pages[i].user)    table->pages[i].user = 1;
      if (src->pages[i].accessed)table->pages[i].accessed = 1;
      if (src->pages[i].dirty)   table->pages[i].dirty = 1;
      // Physically copy the data across. This function is in process.s.
      copy_page_physical(src->pages[i].frame*0x1000, table->pages[i].frame*0x1000);
```

All fairly simple. We use a function which (again) is as yet undefined, called copy_page_physical. We'll define that in a second, just after we end this function.

```
   }
   return table;
}
```

9.2.3. Copying a physical frame

copy_page_physical is really a misnomer. What we actually want to do is copy the contents of one *frame* into another *frame*. This, unfortunately, involves disabling paging (so we can access all of physical RAM), so we write this as a pure assembler function. This should go in a file called 'process.s'.

```
[GLOBAL copy_page_physical]
copy_page_physical:
   push ebx              ; According to __cdecl, we must preserve the contents of EBX.
   pushf                 ; push EFLAGS, so we can pop it and reenable interrupts
                         ; later, if they were enabled anyway.
   cli                   ; Disable interrupts, so we aren't interrupted.
                         ; Load these in BEFORE we disable paging!
   mov ebx, [esp+12]     ; Source address
   mov ecx, [esp+16]     ; Destination address

   mov edx, cr0          ; Get the control register...
   and edx, 0x7fffffff   ; and...
   mov cr0, edx          ; Disable paging.

   mov edx, 1024         ; 1024*4bytes = 4096 bytes to copy

.loop:
   mov eax, [ebx]        ; Get the word at the source address
   mov [ecx], eax        ; Store it at the dest address
   add ebx, 4            ; Source address += sizeof(word)
   add ecx, 4            ; Dest address += sizeof(word)
   dec edx               ; One less word to do
   jnz .loop

   mov edx, cr0          ; Get the control register again
   or  edx, 0x80000000   ; and...
   mov cr0, edx          ; Enable paging.
```

```
   popf                    ; Pop EFLAGS back.
   pop ebx                 ; Get the original value of EBX back.
   ret
```

Hopefully the comments should make it clear what is happening. Anyway, that's a directory successfully cloned! We should now add a call to this in *initialise_paging*, as mentioned above.

```
void initialise_paging()
{
   // The size of physical memory. For the moment we
   // assume it is 16MB big.
   u32int mem_end_page = 0x1000000;

   nframes = mem_end_page / 0x1000;
   frames = (u32int*)kmalloc(INDEX_FROM_BIT(nframes));
   memset(frames, 0, INDEX_FROM_BIT(nframes));

   // Let's make a page directory.
   u32int phys; // ********** ADDED ***********
   kernel_directory = (page_directory_t*)kmalloc_a(sizeof(page_directory_t));
   memset(kernel_directory, 0, sizeof(page_directory_t));
   // *********** MODIFIED ************
   kernel_directory->physicalAddr = (u32int)kernel_directory->tablesPhysical;

   // Map some pages in the kernel heap area.
   // Here we call get_page but not alloc_frame. This causes page_table_t's
   // to be created where necessary. We can't allocate frames yet because they
   // they need to be identity mapped first below, and yet we can't increase
   // placement_address between identity mapping and enabling the heap!
   int i = 0;
   for (i = KHEAP_START; i < KHEAP_END; i += 0x1000)
       get_page(i, 1, kernel_directory);

   // We need to identity map (phys addr = virt addr) from
   // 0x0 to the end of used memory, so we can access this
   // transparently, as if paging wasn't enabled.
   // NOTE that we use a while loop here deliberately.
   // inside the loop body we actually change placement_address
   // by calling kmalloc(). A while loop causes this to be
   // computed on-the-fly rather than once at the start.
   // Allocate a lil' bit extra so the kernel heap can be
   // initialised properly.
   i = 0;
   while (i < placement_address+0x1000)
   {
       // Kernel code is readable but not writeable from userspace.
       alloc_frame( get_page(i, 1, kernel_directory), 0, 0);
       i += 0x1000;
   }

   // Now allocate those pages we mapped earlier.
   for (i = KHEAP_START; i < KHEAP_START+KHEAP_INITIAL_SIZE; i += 0x1000)
       alloc_frame( get_page(i, 1, kernel_directory), 0, 0);

   // Before we enable paging, we must register our page fault handler.
   register_interrupt_handler(14, page_fault);

   // Now, enable paging!
   switch_page_directory(kernel_directory);

   // Initialise the kernel heap.
   kheap = create_heap(KHEAP_START, KHEAP_START+KHEAP_INITIAL_SIZE, 0xCFFFF000, 0, 0);
```

```
   // ******** ADDED *********
   current_directory = clone_directory(kernel_directory);
   switch_page_directory(current_directory);
}

void switch_page_directory(page_directory_t *dir)
{
   current_directory = dir;
   asm volatile("mov %0, %%cr3":: "r"(dir->physicalAddr)); // ******** MODIFIED *********
   u32int cr0;
   asm volatile("mov %%cr0, %0": "=r"(cr0));
   cr0 |= 0x80000000; // Enable paging!
   asm volatile("mov %0, %%cr0":: "r"(cr0));
}
```

(It should be noted that a function prototype for *clone_directory* should be put in the 'paging.h' header file.)

9.3. Creating a new stack

Currently, we have been using an undefined stack. What does that mean? well, GRUB leaves us, stack-wise, in an undefined state. The stack pointer could be anywhere. In all practical situations, GRUB's default stack location is large enough for our startup code to run without problems. However, it is in lower memory (somewhere around 0x7000 physical), which causes us problems as it'll be 'linked' instead of 'copied' when a page directory is changed (because the area from 0x0 - approx 0x150000 is mapped in the kernel_directory). So, we really need to move the stack.

Moving the stack is not particularly difficult. We just memcpy() the data in the old stack over to where the new stack should be. However, there is a problem. When a new stack frame is created (for example, when entering a function) the EBP register is pushed onto the stack. This base pointer is used by the compiler to work out how to reference local variables. If we plainly copy the stack over, these pushed EBP values will point to locations on the *old* stack, not the new one! So we need to change them manually.

Unfortunately, first, we need to know exactly where the current stack starts! To do this, we have to add an instruction right at the start, in boot.s:

```
 ; Add this just before "push ebx".
 push esp
```

This passes another parameter to main() - the initial stack pointer. We need to modify main() to take this extra parameter also:

```
u32int initial_esp; // New global variable.
```

```
int main(struct multiboot *mboot_ptr, u32int initial_stack)
{
   initial_esp = initial_stack;
```

Good. Now we have what we need to start moving the stack. The following function should be in a new file, "task.c".

```
void move_stack(void *new_stack_start, u32int size)
{
  u32int i;
  // Allocate some space for the new stack.
```

```
  for( i = (u32int)new_stack_start;
       i >= ((u32int)new_stack_start-size);
       i -= 0x1000)
  {
    // General-purpose stack is in user-mode.
    alloc_frame( get_page(i, 1, current_directory), 0 /* User mode */, 1 /* Is writable */ );
  }
```

Now, we've changed a page table. So we need to inform the processor that a mapping has changed. This is called "Flushing the TLB (translation lookaside buffer)". It can be done partially, using the "invlpg" instruction, or fully, by simply writing to cr3. We choose the simpler latter option.

```
  // Flush the TLB by reading and writing the page directory address again.
  u32int pd_addr;
  asm volatile("mov %%cr3, %0" : "=r" (pd_addr));
  asm volatile("mov %0, %%cr3" : : "r" (pd_addr));
```

Next, we read the current stack and base pointers, and calculate an offset to get from an address on the old stack to an address on the new stack, and use it to calculate the new stack/base pointers.

```
 // Old ESP and EBP, read from registers.
 u32int old_stack_pointer; asm volatile("mov %%esp, %0" : "=r" (old_stack_pointer));
 u32int old_base_pointer;  asm volatile("mov %%ebp, %0" : "=r" (old_base_pointer));
```

```
 u32int offset            = (u32int)new_stack_start - initial_esp;
```

```
 u32int new_stack_pointer = old_stack_pointer + offset;
 u32int new_base_pointer  = old_base_pointer  + offset;
```

Great. Now we can actually copy the stack.

```
// Copy the stack.
memcpy((void*)new_stack_pointer, (void*)old_stack_pointer, initial_esp-old_stack_pointer);
```

Now we try and go through the new stack, looking for base pointers to change. Here we use an algorithm which is not fool-proof. We assume that any value on the stack which is in the range of the stack (old_stack_pointer < x < initial_esp) is a pushed EBP. This will, unfortunately, completely trash any value which isn't an EBP but just happens to be in this range. Oh well, these things happen.

```
// Backtrace through the original stack, copying new values into
// the new stack.
for(i = (u32int)new_stack_start; i > (u32int)new_stack_start-size; i -= 4)
{
   u32int tmp = * (u32int*)i;
   // If the value of tmp is inside the range of the old stack, assume it is a base pointer
   // and remap it. This will unfortunately remap ANY value in this range, whether they are
   // base pointers or not.
   if (( old_stack_pointer < tmp) && (tmp < initial_esp))
   {
     tmp = tmp + offset;
     u32int *tmp2 = (u32int*)i;
     *tmp2 = tmp;
   }
}
```

Lastly we just need to actually change the stack and base pointers.

```
  // Change stacks.
  asm volatile("mov %0, %%esp" : : "r" (new_stack_pointer));
  asm volatile("mov %0, %%ebp" : : "r" (new_base_pointer));
}
```

9.4. Actual multitasking code

Now that we've got the necessary support functions written, we can actually start writing some tasking code.

Firstly in task.h, we'll need some definitions.

```
//
// task.h - Defines the structures and prototypes needed to multitask.
// Written for JamesM's kernel development tutorials.
//

#ifndef TASK_H
#define TASK_H

#include "common.h"
#include "paging.h"

// This structure defines a 'task' - a process.
typedef struct task
{
   int id;                // Process ID.
   u32int esp, ebp;       // Stack and base pointers.
   u32int eip;            // Instruction pointer.
   page_directory_t *page_directory; // Page directory.
   struct task *next;     // The next task in a linked list.
} task_t;

// Initialises the tasking system.
void initialise_tasking();

// Called by the timer hook, this changes the running process.
void task_switch();

// Forks the current process, spawning a new one with a different
// memory space.
int fork();

// Causes the current process' stack to be forcibly moved to a new location.
void move_stack(void *new_stack_start, u32int size);

// Returns the pid of the current process.
int getpid();

#endif
```

We define a task structure, which contains a task ID (known as a PID), some saved registers, a pointer to a page directory, and the next task in the list (this is a singly linked list).

In task.c, we'll need some global variables, and we have a small initialise_tasking function that just creates one, blank, task.

```
//
```

```
// task.c - Implements the functionality needed to multitask.
// Written for JamesM's kernel development tutorials.
//

#include "task.h"
#include "paging.h"

// The currently running task.
volatile task_t *current_task;

// The start of the task linked list.
volatile task_t *ready_queue;

// Some externs are needed to access members in paging.c...
extern page_directory_t *kernel_directory;
extern page_directory_t *current_directory;
extern void alloc_frame(page_t*,int,int);
extern u32int initial_esp;
extern u32int read_eip();

// The next available process ID.
u32int next_pid = 1;

void initialise_tasking()
{
```

```
   asm volatile("cli");

   // Relocate the stack so we know where it is.
   move_stack((void*)0xE0000000, 0x2000);

   // Initialise the first task (kernel task)
   current_task = ready_queue = (task_t*)kmalloc(sizeof(task_t));
   current_task->id = next_pid++;
   current_task->esp = current_task->ebp = 0;
   current_task->eip = 0;
   current_task->page_directory = current_directory;
   current_task->next = 0;

   // Reenable interrupts.
   asm volatile("sti");
}
```

Right. We only have two more functions to write - fork(), and switch_task(). Fork() is a UNIX function to create a new process. It clones the address space and starts the new process running at the same place as the original process is currently at.

```
int fork()
{
   // We are modifying kernel structures, and so cannot be interrupted.
   asm volatile("cli");

   // Take a pointer to this process' task struct for later reference.
   task_t *parent_task = (task_t*)current_task;

   // Clone the address space.
   page_directory_t *directory = clone_directory(current_directory);
```

So firstly we disable interrupts, because we're changing kernel structures and could cause problems if we're interrupted half way through. We then clone the current page directory.

```
   // Create a new process.
   task_t *new_task = (task_t*)kmalloc(sizeof(task_t));
   new_task->id = next_pid++;
   new_task->esp = new_task->ebp = 0;
   new_task->eip = 0;
   new_task->page_directory = directory;
   new_task->next = 0;

   // Add it to the end of the ready queue.
   // Find the end of the ready queue...
   task_t *tmp_task = (task_t*)ready_queue;
   while (tmp_task->next)
       tmp_task = tmp_task->next;
   // ...And extend it.
   tmp_task->next = new_task;
```

Here we create a new process, just like in *initialise_tasking*. We add it to the end of the ready queue (the queue of tasks that are ready to run). If you don't understand this code, I suggest you look up a tutorial on working with Singly Linked Lists.

We have to tell the task where it should start executing. For this, we need to read the current instruction pointer. We need a quick read_eip() function to do this - this is in process.s:

```
[GLOBAL read_eip]
read_eip:
  pop eax
  jmp eax
```

This is a rather clever way of reading the current instruction pointer. When read_eip is called, the current instruction location is pushed onto the stack. Normally, we use "ret" to return from a function. This instruction pops the value from the stack and jumps to it. Here, however, we pop the value ourselves, into EAX (remember that EAX is the 'return value' register for the __cdecl calling convention), then jump to it.

```
// This will be the entry point for the new process.
u32int eip = read_eip();
```

Important to note is that because (later) we set the new task's starting address to "eip", after the call to read_eip we could be in one of two states.

1. We just called read_eip, and are the parent task.
2. We are the child task, and just started executing.

To try and distinguish between the two cases, we check if "current_task == parent_task". In switch_task(), we will add code which updates "current_task" to always point to the currently running task. So, if we are the child task, current_task will not be the same as parent_task, else, it will.

```
   // We could be the parent or the child here - check.
   if (current_task == parent_task)
   {
       // We are the parent, so set up the esp/ebp/eip for our child.
       u32int esp; asm volatile("mov %%esp, %0" : "=r"(esp));
       u32int ebp; asm volatile("mov %%ebp, %0" : "=r"(ebp));
       new_task->esp = esp;
       new_task->ebp = ebp;
       new_task->eip = eip;
       // All finished: Reenable interrupts.
```

```
        asm volatile("sti");
```

```
        return new_task->id;
    }
    else
    {
        // We are the child - by convention return 0.
        return 0;
    }
}
```

Let's just run through that code. If we are the parent task, we read the current stack pointer and base pointer values and store them into the new task's task_struct. We also store the instruction pointer we found earlier in there, and reenable interrupts (because we've finished). Fork(), by convention, returns the PID of the child task if we are the parent, or zero if we are the child.

9.4.1. Switching tasks

Firstly we need to get the timer callback to call our scheduling function.

In timer.c

```
static void timer_callback(registers_t regs)
{
   tick++;
   switch_task();
}
```

Now we just need to write it! ;)

```
void switch_task()
{
   // If we haven't initialised tasking yet, just return.
   if (!current_task)
       return;
```

Because this function will be called whenever the timer fires, it is very possible that it will be called before *initialise_tasking* has been called. So we check that here - if the current task is NULL, we haven't set up tasking yet, so just return.

Next, lets just quickly grab the stack and base pointers - we'll need them in a minute.

```
// Read esp, ebp now for saving later on.
u32int esp, ebp, eip;
asm volatile("mov %%esp, %0" : "=r"(esp));
asm volatile("mov %%ebp, %0" : "=r"(ebp));
```

Now it's time for some cunning logic. *Make sure you understand this piece of code. It's very important*. We read the instruction pointer, using our read_eip function again. We'll put this value into the current task's "eip" field, so the next time it is scheduled, it picks up again at exactly the same location. However, just like in fork(), after the call we could be in one of two states:

1. We just called read_eip, and it returned us the current instruction pointer.
2. We just switched tasks, and execution started just after the read_eip function.

How do we distinguish between the two? Well, we can cheat. When we actually do the assembly to switch tasks (in a minute), we can plant a dummy value (I've used 0x12345) into EAX. Because C

uses EAX as the return value of a function, in the second case the return value of read_eip will *seem* to be 0x12345! So we can use that to distinguish between them.

```
   // Read the instruction pointer. We do some cunning logic here:
   // One of two things could have happened when this function exits -
   // (a) We called the function and it returned the EIP as requested.
   // (b) We have just switched tasks, and because the saved EIP is essentially
   // the instruction after read_eip(), it will seem as if read_eip has just
   // returned.
   // In the second case we need to return immediately. To detect it we put a dummy
   // value in EAX further down at the end of this function. As C returns values in EAX,
   // it will look like the return value is this dummy value! (0x12345).
   eip = read_eip();

   // Have we just switched tasks?
   if (eip == 0x12345)
       return;
```

Next, we write the new ESP, EBP and EIP into the current task's task struct.

```
   // No, we didn't switch tasks. Let's save some register values and switch.
   current_task->eip = eip;
   current_task->esp = esp;
   current_task->ebp = ebp;
```

Then, we switch tasks! Advance through the current_task listed list. If we fall off the end (if current_task ends up being zero, we just start again).

```
   // Get the next task to run.
   current_task = current_task->next;
   // If we fell off the end of the linked list start again at the beginning.
   if (!current_task) current_task = ready_queue;
```

```
   esp = current_task->esp;
   ebp = current_task->ebp;
```

The last three lines are just to make the assembly that follows a bit easier to understand.

The comments in this function really should explain everything. We change all the registers we need, then jump to the new instruction location.

```
   // Here we:
   // * Stop interrupts so we don't get interrupted.
   // * Temporarily put the new EIP location in ECX.
   // * Load the stack and base pointers from the new task struct.
   // * Change page directory to the physical address (physicalAddr) of the new directory.
   // * Put a dummy value (0x12345) in EAX so that above we can recognise that we've just
   // switched task.
   // * Restart interrupts. The STI instruction has a delay - it doesn't take effect until after
   // the next instruction.
   // * Jump to the location in ECX (remember we put the new EIP in there).
   asm volatile("         \
     cli;                 \
     mov %0, %%ecx;       \
     mov %1, %%esp;       \
     mov %2, %%ebp;       \
     mov %3, %%cr3;       \
     mov $0x12345, %%eax; \
     sti;                 \
```

```
    jmp *%%ecx               "
                : : "r"(eip), "r"(esp), "r"(ebp), "r"(current_directory->physicalAddr));
}
```

Sorted! That's us finished! Let's test it out!

9.5. Testing

Let's change our main() function:

```
int main(struct multiboot *mboot_ptr, u32int initial_stack)
{
   initial_esp = initial_stack;
   // Initialise all the ISRs and segmentation
   init_descriptor_tables();
   // Initialise the screen (by clearing it)
   monitor_clear();
   // Initialise the PIT to 100Hz
   asm volatile("sti");
   init_timer(50);

   // Find the location of our initial ramdisk.
   ASSERT(mboot_ptr->mods_count > 0);
   u32int initrd_location = *((u32int*)mboot_ptr->mods_addr);
   u32int initrd_end = *(u32int*)(mboot_ptr->mods_addr+4);
   // Don't trample our module with placement accesses, please!
   placement_address = initrd_end;

   // Start paging.
   initialise_paging();

   // Start multitasking.
   initialise_tasking();

   // Initialise the initial ramdisk, and set it as the filesystem root.
   fs_root = initialise_initrd(initrd_location);

   // Create a new process in a new address space which is a clone of this.
   int ret = fork();

   monitor_write("fork() returned ");
   monitor_write_hex(ret);
   monitor_write(", and getpid() returned ");
   monitor_write_hex(getpid());
   monitor_write("\n================================================================================\n");

   // The next section of code is not reentrant so make sure we aren't interrupted during.
   asm volatile("cli");
   // list the contents of /
   int i = 0;
   struct dirent *node = 0;
   while ( (node = readdir_fs(fs_root, i)) != 0)
   {
       monitor_write("Found file ");
       monitor_write(node->name);
       fs_node_t *fsnode = finddir_fs(fs_root, node->name);

       if ((fsnode->flags&0x7) == FS_DIRECTORY)
       {
           monitor_write("\n\t(directory)\n");
       }
```

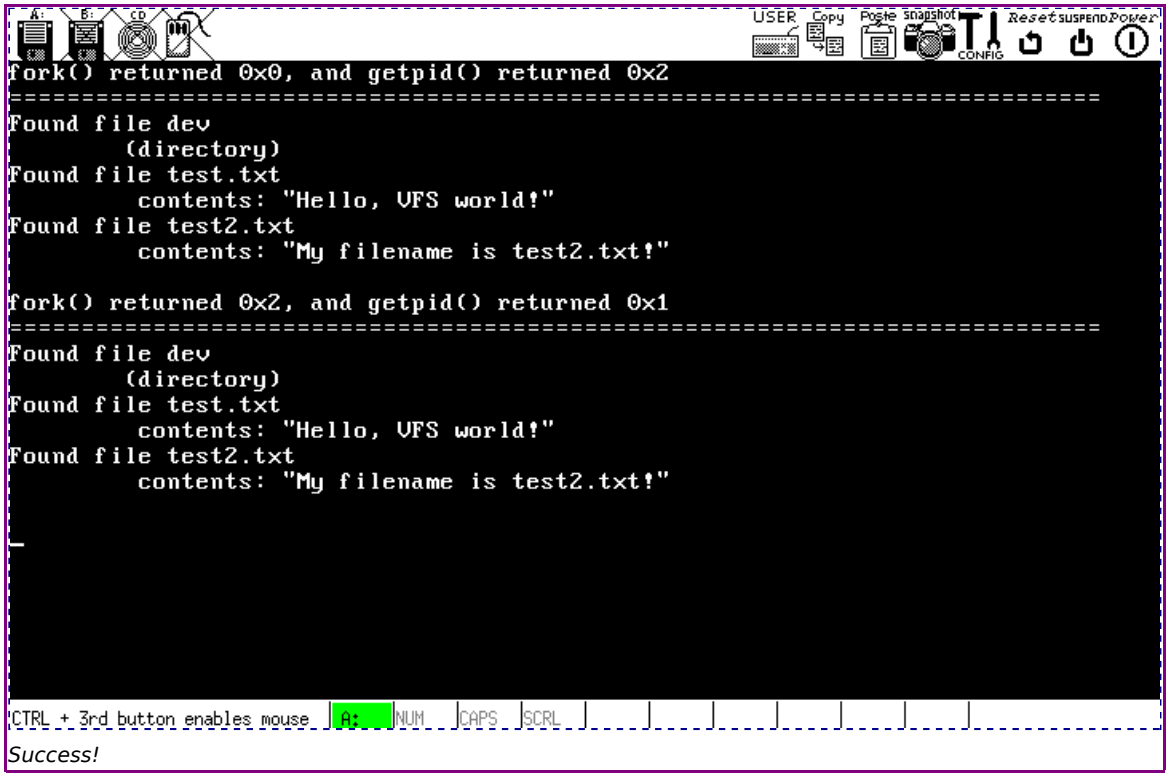
```
        else
        {
            monitor_write("\n\t contents: \"");
            char buf[256];
            u32int sz = read_fs(fsnode, 0, 256, buf);
            int j;
            for (j = 0; j < sz; j++)
                monitor_put(buf[j]);

            monitor_write("\"\n");
        }
        i++;
    }
    monitor_write("\n");

    asm volatile("sti");

    return 0;
}
```

9.6. Summary



Success!

Multitasking is really one of the final hurdles to creating a "proper" kernel. Giving the user the appearance of being able to run multiple things concurrently is essential to any modern OS.

Full source code is available here.