www.jamesmolloy.co.uk

Home » JamesM's kernel development tutorials

6. Paging

In this chapter we're going to enable paging. Paging serves a twofold purpose - memory protection, and virtual memory (the two being almost inextricably interlinked).

6.1. Virtual memory (theory)

If you already know what virtual memory is, you can skip this section.

In linux, if you create a tiny test program such as

```
int main(char argc, char **argv)
{
  return 0;
}
```

, compile it, then run 'objdump -f', you might find something similar to this.

```
jamesmol@aubergine:~/test> objdump -f a.out

a.out:     file format elf32-i386
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x080482a0
```

Notice the start address of the program is at 0x80482a0, which is about 128MB into the address space. It may seem strange, then, that this program will run perfectly on machines with < 128MB of RAM.

What the program is actually 'seeing', when it reads and writes memory, is a virtual address space. Parts of the virtual address space are mapped to physical memory, and parts are unmapped. If you try to access an unmapped part, the processor raises a *page fault*, the operating system catches it, and in POSIX systems delivers a SIGSEGV signal closely followed by SIGKILL.

This abstraction is extremely useful. It means that compilers can produce a program that relies on the code being at an *exact* location in memory, every time it is run. With virtual memory, the process *thinks* it is at, for example, 0x080482a0, but actually it could be at physical memory location 0x1000000. Not only that, but

processes cannot accidentally (or deliberately) trample other processes' data or code.

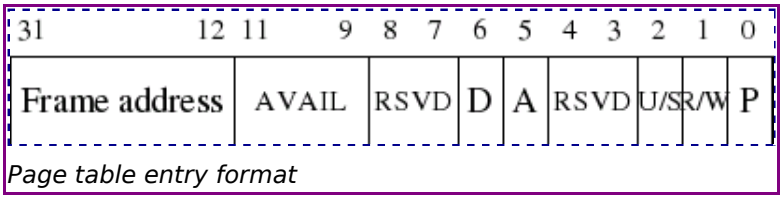
Virtual memory of this type is wholly dependent on hardware support. It cannot be emulated by software. Luckily, the x86 has just such a thing. It's called the MMU (memory management unit), and it handles all memory mappings due to segmentation and paging, forming a layer between the CPU and memory (actually, it's part of the CPU, but that's just an implementation detail).

6.2. Paging as a concretion of virtual memory

Virtual memory is an abstract principle. As such it requires *concretion* through some system/algorithm. Both segmentation (see chapter 3) and paging are valid methods for implementing virtual memory. As mentioned in chapter 3 however, segmentation is becoming obsolete. Paging is the newer, better alternative for the x86 architecture.

Paging works by splitting the virtual address space into blocks called *pages*, which are usually 4KB in size. Pages can then be mapped on to *frames* - equally sized blocks of physical memory.

6.2.1. Page entries



Page table entry format

Each process normally has a different set of page mappings, so that virtual memory spaces are independent of each other. In the x86 architecture (32-bit) pages are fixed at 4KB in size. Each page has a corresponding descriptor word, which tells the processor which frame it is mapped to. Note that because pages and frames must be aligned on 4KB boundaries (4KB being 0x1000 bytes), the least significant 12 bits of the 32-bit word are always zero. The architecture takes advantage of this by using them to store information about the page, such as whether it is present, whether it is kernel-mode or user-mode etc. The layout of this word is in the picture on the right.

The fields in that picture are pretty simple, so let's quickly go through them.

P
Set if the page is present in memory.
R/W
If set, that page is writeable. If unset, the page is read-only. This does not apply when code is running in kernel-mode (unless a flag in CR0 is set).
U/S
If set, this is a user-mode page. Else it is a supervisor (kernel)-mode page. User-mode code cannot write to or read from kernel-mode pages.
Reserved

These are used by the CPU internally and cannot be trampled.

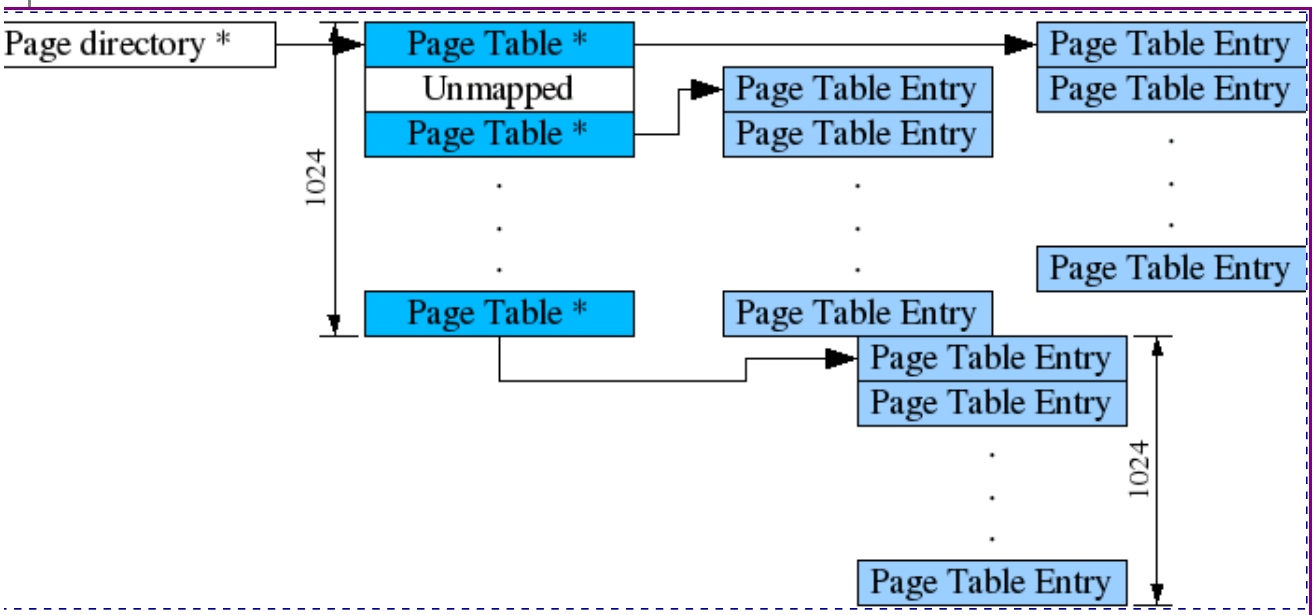
A
Set if the page has been accessed (Gets set by the CPU).

D
Set if the page has been written to (dirty).

AVAIL
These 3 bits are unused and available for kernel-use.

Page frame address
The high 20 bits of the frame address in physical memory.

6.2.2. Page directories/tables



er layout

Possibly you've been tapping on your calculator and have worked out that to generate a table mapping each 4KB page to one 32-bit descriptor over a 4GB address space requires 4MB of memory. Perhaps, perhaps not - but it's true.

4MB may seem like a large overhead, and to be fair, it is. If you have 4GB of physical RAM, it's not much. However, if you are working on a machine that has 16MB of RAM, you've just lost a quarter of your available memory! What we want is something progressive, that will take up an amount of space proportionate to the amount of RAM you have.

Well, we don't have that. But intel did come up with something similar - they use a 2-tier system. The CPU gets told about a *page directory*, which is a 4KB large table, each entry of which points to a *page table*. The page table is, again, 4KB large and each entry is a *page table entry*, described above.

This way, The entire 4GB address space can be covered with the advantage that if a page table has no entries, it can be freed and it's *present* flag unset in the page directory.

6.2.3. Enabling paging

Enabling paging is extremely easy.

1. Copy the location of your page directory into the CR3 register. This must, of course, be the **physical** address.
2. Set the *PG* bit in the CR0 register. You can do this by OR-ing with 0x80000000.

6.3. Page faults

When a process does something the memory-management unit doesn't like, a page fault interrupt is thrown. Situations that can cause this are (not complete):

- Reading from or writing to an area of memory that is not mapped (page entry/table's 'present' flag is not set)
- The process is in user-mode and tries to write to a read-only page.
- The process is in user-mode and tries to access a kernel-only page.
- The page table entry is corrupted - the reserved bits have been overwritten.

The page fault interrupt is number 14, and looking at chapter 3 we can see that this throws an error code. This error code gives us quite a bit of information about what happened.

Bit 0
If set, the fault was **not** because the page wasn't present. If unset, the page wasn't present.
Bit 1
If set, the operation that caused the fault was a write, else it was a read.
Bit 2
If set, the processor was running in user-mode when it was interrupted. Else, it was running in kernel-mode.
Bit 3
If set, the fault was caused by reserved bits being overwritten.
Bit 4
If set, the fault occurred during an instruction fetch.

The processor also gives us another piece of information - the address that caused the fault. This is located in the CR2 register. Beware that if your page fault hander itself causes another page fault exception this register will be overwritten - so save it early!

6.4. Putting it into practice

We're almost ready to start implementing. We will, however, need a few assistant functions first, the most important of which are memory management functions.

6.4.1. Simple memory management with placement malloc

If you come from a C++ background, you may have heard of 'placement new'. This is a version of new that takes a parameter. Instead of calling malloc, as it normally would, it creates the object at the address specified. We are going to use a very similar concept.

When the kernel is sufficiently booted, we will have a kernel heap active and operational. The way we code heaps, though, usually requires that virtual memory is enabled. So we need a simple alternative to allocate memory before the heap is active.

As we're allocating quite early on in the kernel bootup, we can make the assumption that nothing that is kmalloc()'d will ever need to be kfree()'d. This simplifies things greatly. We can just have a pointer (*placement address*) to some free memory that we pass back to the requestee then increment. Thus:

```
u32int kmalloc(u32int sz)
{
  u32int tmp = placement_address;
  placement_address += sz;
  return tmp;
}
```

That will actually suffice. However, we have another requirement. When we allocate page tables and directories, they *must be page-aligned*. So we can build that in:

```
u32int kmalloc(u32int sz, int align)
{
  if (align == 1 && (placement_address & 0xFFFFF000)) // If the address is not
already page-aligned
  {
    // Align it.
    placement_address &= 0xFFFFF000;
    placement_address += 0x1000;
  }
  u32int tmp = placement_address;
  placement_address += sz;
  return tmp;
}
```

Now, unfortunately, we have one more requirement, and I can't really explain to you why it is required until later in the tutorials. It has to do with when we clone a page directory (when fork()ing processes). At this point, paging will be fully enabled, and kmalloc will return a virtual address. But, we also (bear with me, you'll be glad we did later) need to get the *physical* address of the memory allocated. Take it on faith for now - it's not much code anyway.

```
u32int kmalloc(u32int sz, int align, u32int *phys)
```

```
{
  if (align == 1 && (placement_address & 0xFFFFF000)) // If the address is not
already page-aligned
  {
    // Align it.
    placement_address &= 0xFFFFF000;
    placement_address += 0x1000;
  }
  if (phys)
  {
    *phys = placement_address;
  }
  u32int tmp = placement_address;
  placement_address += sz;
  return tmp;
}
```

Great. This is all we need for simple memory management. In my code I have actually (for aesthetic purposes) renamed *kmalloc* to *kmalloc_int* (for kmalloc_internal). I then have several wrapper functions:

```
u32int kmalloc_a(u32int sz);  // page aligned.
u32int kmalloc_p(u32int sz, u32int *phys); // returns a physical address.
u32int kmalloc_ap(u32int sz, u32int *phys); // page aligned and returns a physical
address.
u32int kmalloc(u32int sz); // vanilla (normal).
```

I just feel this interface is nicer than specifying 3 parameters for every kernel heap allocation! These definitions should go in kheap.h/kheap.c.

6.4.2. Required definitions

paging.h should contain some structure definitions that will make our life easier.

```
#ifndef PAGING_H
#define PAGING_H

#include "common.h"
#include "isr.h"

typedef struct page
{
   u32int present    : 1;   // Page present in memory
   u32int rw         : 1;   // Read-only if clear, readwrite if set
   u32int user       : 1;   // Supervisor level only if clear
   u32int accessed   : 1;   // Has the page been accessed since last refresh?
   u32int dirty      : 1;   // Has the page been written to since last refresh?
   u32int unused     : 7;   // Amalgamation of unused and reserved bits
   u32int frame      : 20;  // Frame address (shifted right 12 bits)
} page_t;
```

```
typedef struct page_table
{
   page_t pages[1024];
} page_table_t;

typedef struct page_directory
{
   /**
      Array of pointers to pagetables.
   **/
   page_table_t *tables[1024];
   /**
      Array of pointers to the pagetables above, but gives their *physical*
      location, for loading into the CR3 register.
   **/
   u32int tablesPhysical[1024];
   /**
      The physical address of tablesPhysical. This comes into play
      when we get our kernel heap allocated and the directory
      may be in a different location in virtual memory.
   **/
   u32int physicalAddr;
} page_directory_t;

/**
  Sets up the environment, page directories etc and
  enables paging.
**/
void initialise_paging();

/**
  Causes the specified page directory to be loaded into the
  CR3 register.
**/
void switch_page_directory(page_directory_t *new);

/**
  Retrieves a pointer to the page required.
  If make == 1, if the page-table in which this page should
  reside isn't created, create it!
**/
page_t *get_page(u32int address, int make, page_directory_t *dir);

/**
  Handler for page faults.
**/
void page_fault(registers_t regs);
```

Note the *tablesPhysical* and *physicalAddr* members of page_table_t. What are they doing there?

The physicalAddr member is actually only for when we clone page directories (not until later in the tutorials). Remember that at that point, the new directory will have

an address in virtual memory that is not the same as physical memory. We will need the physical address to tell the CPU if we ever want to switch directories.

The tablesPhysical member is similar. It is a solution to a problem: How do you access your page tables? It may seem simple, but remember that a page directory must hold *physical* addresses, not virtual ones. And the only way you can read/write to memory is using *virtual* addresses!

One solution to this problem is to never access your page tables directly, but to map one page table to point back to the page directory, so that by accessing memory at a certain address you can see all your page tables as if they were pages, and all your page table entries as if they were normal integers. The diagram on the right should help to explain. This method is a little counter-intuitive in my opinion and it also wastes 256MB of addressable space, so I prefer another method.

The second method is to, for every page directory, keep 2 arrays. One holding the physical addresses of it's page tables (for giving to the CPU), and the other holding the virtual ones (so we can read/write to them). This only gives us an extra overhead of 4KB per page directory, which is not much.

6.4.3. Frame allocation

If we want to map a page to a frame, we need some way of finding a free frame. Of course, we could just maintain a massive array of 1's and 0's, but that would be extremely wasteful - we don't need 32-bits just to hold 2 values, we can do that with 1 bit. So if we use a bitset, we will be using 32 times less space!

If you don't know what a bitset (also called a bitmap) is, you should read the link above. There are only 3 functions a bitset implements - set, test and clear. I have also implemented a function to efficiently find the first free frame from the bitmap. Have a look at it and work out why it is efficient. My implementation of these is below. I'm not going to go through explaining it - this is a general concept and is not kernel related. If you're confused, search google for bitset implementations, and if worst comes to the worst post on the osdev.net forums.

```
// A bitset of frames - used or free.
u32int *frames;
u32int nframes;

// Defined in kheap.c
extern u32int placement_address;

// Macros used in the bitset algorithms.
#define INDEX_FROM_BIT(a) (a/(8*4))
#define OFFSET_FROM_BIT(a) (a%(8*4))

// Static function to set a bit in the frames bitset
static void set_frame(u32int frame_addr)
{
```

```
   u32int frame = frame_addr/0x1000;
   u32int idx = INDEX_FROM_BIT(frame);
   u32int off = OFFSET_FROM_BIT(frame);
   frames[idx] |= (0x1 << off);
}

// Static function to clear a bit in the frames bitset
static void clear_frame(u32int frame_addr)
{
   u32int frame = frame_addr/0x1000;
   u32int idx = INDEX_FROM_BIT(frame);
   u32int off = OFFSET_FROM_BIT(frame);
   frames[idx] &= ~(0x1 << off);
}

// Static function to test if a bit is set.
static u32int test_frame(u32int frame_addr)
{
   u32int frame = frame_addr/0x1000;
   u32int idx = INDEX_FROM_BIT(frame);
   u32int off = OFFSET_FROM_BIT(frame);
   return (frames[idx] & (0x1 << off));
}

// Static function to find the first free frame.
static u32int first_frame()
{
   u32int i, j;
   for (i = 0; i < INDEX_FROM_BIT(nframes); i++)
   {
       if (frames[i] != 0xFFFFFFFF) // nothing free, exit early.
       {
           // at least one bit is free here.
           for (j = 0; j < 32; j++)
           {
               u32int toTest = 0x1 << j;
               if ( !(frames[i]&toTest) )
               {
                   return i*4*8+j;
               }
           }
       }
   }
}
```

Hopefully that code shouldn't cause too many surprises. It just fancy bit twiddling. We then come to functions to allocate and deallocate frames. Now that we have an efficient bitset implementation, these functions total just a few lines!

```
// Function to allocate a frame.
void alloc_frame(page_t *page, int is_kernel, int is_writeable)
{
   if (page->frame != 0)
```

```
    {
        return; // Frame was already allocated, return straight away.
    }
    else
    {
        u32int idx = first_frame(); // idx is now the index of the first free frame.
        if (idx == (u32int)-1)
        {
            // PANIC is just a macro that prints a message to the screen then hits
an infinite loop.
            PANIC("No free frames!");
        }
        set_frame(idx*0x1000); // this frame is now ours!
        page->present = 1; // Mark it as present.
        page->rw = (is_writeable)?1:0; // Should the page be writeable?
        page->user = (is_kernel)?0:1; // Should the page be user-mode?
        page->frame = idx;
    }
}

// Function to deallocate a frame.
void free_frame(page_t *page)
{
    u32int frame;
    if (!(frame=page->frame))
    {
        return; // The given page didn't actually have an allocated frame!
    }
    else
    {
        clear_frame(frame); // Frame is now free again.
        page->frame = 0x0; // Page now doesn't have a frame.
    }
}
```

Note that the PANIC macro just calls a global function called *panic*, with arguments of the message given and the __FILE__ and __LINE__ it occurred on. *panic* prints these out and enters an infinite loop, stopping all execution.

6.4.4. Paging code, finally!

```
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.
```

```
    kernel_directory = (page_directory_t*)kmalloc_a(sizeof(page_directory_t));
    memset(kernel_directory, 0, sizeof(page_directory_t));
    current_directory = 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.
    int i = 0;
    while (i < placement_address)
    {
        // Kernel code is readable but not writeable from userspace.
        alloc_frame( get_page(i, 1, kernel_directory), 0, 0);
        i += 0x1000;
    }
    // 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);
}

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

page_t *get_page(u32int address, int make, page_directory_t *dir)
{
    // Turn the address into an index.
    address /= 0x1000;
    // Find the page table containing this address.
    u32int table_idx = address / 1024;
    if (dir->tables[table_idx]) // If this table is already assigned
    {
        return &dir->tables[table_idx]->pages[address%1024];
    }
    else if(make)
    {
        u32int tmp;
        dir->tables[table_idx] = (page_table_t*)kmalloc_ap(sizeof(page_table_t), &tmp);
        memset(dir->tables[table_idx], 0, 0x1000);
        dir->tablesPhysical[table_idx] = tmp | 0x7; // PRESENT, RW, US.
        return &dir->tables[table_idx]->pages[address%1024];
    }
    else
```

```
    {
        return 0;
    }
}
```

Right, let's analyse that. First of all, the utility functions.

switch_page_directory does exactly what it says on the tin. It takes a page directory, and switches to it. It does this by moving the address of the tablesPhysical member of that directory into the CR3 register. Remember that the tablesPhysical member is an array of physical addresses. After that it first gets the contents of CR0, then OR-s the *PG* bit (0x80000000), then rewrites it. This enables paging and flushes the page-directory cache as well.

get_page returns a pointer to the page entry for a particular address. It can optionally be passed a parameter - make. If make is 1, and the page table that the requested page entry should reside in hasn't been created, then it will be created. Otherwise, the function would just return 0. So, if the table has already been assigned, it will look up the page entry and return it. If it hasn't (and make == 1), it will attempt to create it.

It uses our *kmalloc_ap* function to retrieve a memory block which is page-aligned, and *also gets given it's physical location*. The physical location gets stored in 'tablesPhysical' (after several bits have been set telling the CPU that it is present, writeable, and user-accessible), and the virtual location is stored in 'tables'.

initialise_paging firstly creates the frames bitset, and sets everything to zero using memset. Then it allocates space (which is page-aligned) for a page directory. After that, it allocates frames such that any page access will map to the frame with the same linear address, called identity-mapping. This is done for a small section of the address space, so the kernel code can continue to run as normal. It registers an interrupt handler for page faults (below) then enables paging.

6.4.5. The page fault handler

```
void page_fault(registers_t regs)
{
   // A page fault has occurred.
   // The faulting address is stored in the CR2 register.
   u32int faulting_address;
   asm volatile("mov %%cr2, %0" : "=r" (faulting_address));

   // The error code gives us details of what happened.
   int present   = !(regs.err_code & 0x1); // Page not present
   int rw = regs.err_code & 0x2;           // Write operation?
   int us = regs.err_code & 0x4;           // Processor was in user-mode?
   int reserved = regs.err_code & 0x8;     // Overwritten CPU-reserved bits of page
entry?
   int id = regs.err_code & 0x10;          // Caused by an instruction fetch?
```

```
    // Output an error message.
    monitor_write("Page fault! ( ");
    if (present) {monitor_write("present ");}
    if (rw) {monitor_write("read-only ");}
    if (us) {monitor_write("user-mode ");}
    if (reserved) {monitor_write("reserved ");}
    monitor_write(") at 0x");
    monitor_write_hex(faulting_address);
    monitor_write("\n");
    PANIC("Page fault");
}
```

All this handler does is print out a nice error message. It gets the faulting address from CR2, and analyses the error code pushed by the processor to glean some information from it.

6.4.6. Testing

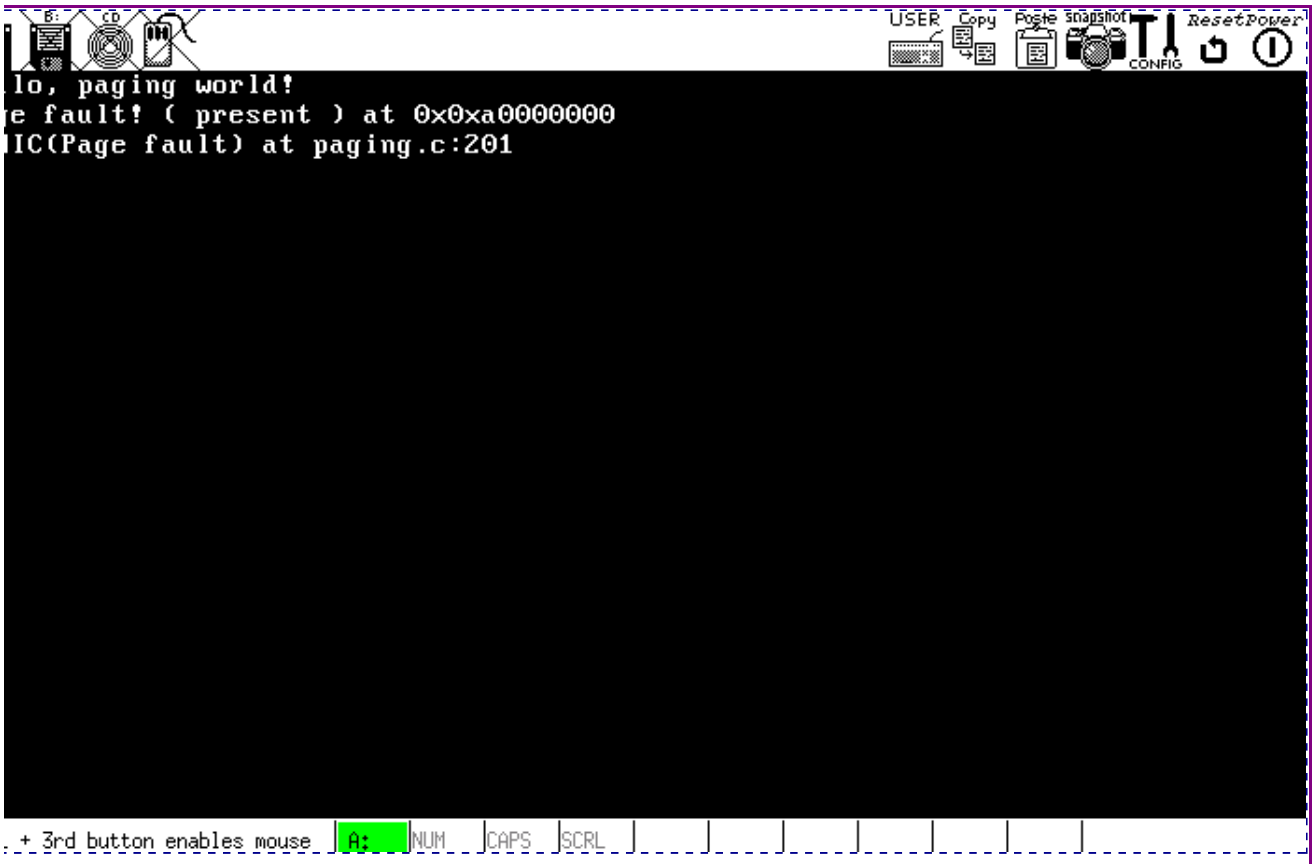
Awesome! you now have code that enables paging and handles page faults! Let's just check it actually works, shall we ...?

main.c

```
int main(struct multiboot *mboot_ptr)
{
    // Initialise all the ISRs and segmentation
    init_descriptor_tables();
    // Initialise the screen (by clearing it)
    monitor_clear();

    initialise_paging();
    monitor_write("Hello, paging world!\n");

    u32int *ptr = (u32int*)0xA0000000;
    u32int do_page_fault = *ptr;

    return 0;
}
```

This will, obviously, initialise paging, print a string to make sure it's set up right and not faulting when it shoudn't, and then force a page fault by reading location 0xA0000000.

Congrats! you're all done! you can now move on to the next tutorial - making a working kernel heap :D. The source code for this tutorial is available here.



hs, enabling paging then forcing a page fault