Systems Programming Assessed Component 2

Percentage of final grade: 35%

Date set: 27th October 2016

Submission date and time deadline: 9pm on Wednesday 9th November 2016

Introduction

You should use Step6 from Week 6 as your starting point for this assessment. If you wish to use your own code in this from previous stages, you may do so.

We now need to look at adding support for managing blocks of physical memory. It is not possible for an operating system to manage the allocation and deallocation of each individual byte of memory because trying to manage every byte would use up most of the RAM. So, we need to divide available memory into blocks of a specific size and manage the allocation and deallocation of blocks of memory.

The boot process in Step6 has been updated to obtain a memory map from the BIOS that tells us what different memory regions are available or reserved. We can use this memory map to determine the total addressable memory space, how much available RAM there is to use in total and which regions in the address space are available to use. A pointer to this memory map is passed as part of a structure to the main method of the kernel when it starts up.

Task

The rest of this document describes, in detail, what you need to produce and the required information you need in order to do it.

The boot process now passes a *BootInfo* structure to the kernel. This is defined in bootinfo.h. For this assessment, the key field in *BootInfo* is *MemoryRegions*. This is a pointer to a variable length array of *MemoryRegion* elements.

Each element in this array identifies the start address and size of a region of memory and how it is used. The *MemoryRegion* structure can be seen below:

```
typedef struct _MemoryRegion
{
      uint32_t    StartOfRegionLow;
      uint32_t    StartOfRegionHigh;
      uint32_t    SizeOfRegionLow;
      uint32_t    SizeOfRegionHigh;
      uint32_t    Type;
} MemoryRegion;
```

Since we are developing a 32-bit operating system, only the low 32-bits of the start of the region and the size of the region are used (*StartOfRegionHigh* and *SizeOfRegionHigh* are ignored). A value of 0 in *StartOfRegionLow* in any element other than the first element of the array indicates that there are no more elements in the array (there is always at least one element in the array).

The *Type* field indicates whether the memory is available, reserved or has some other use. For now, we are only interested in regions of memory with type 1 (*MEMORY_REGION_AVAILABLE*).

For example, using a standard Bochs configuration, if you displayed the contents of this array, you might see the following output:

Region 0: Start: 0x00, length: 0x09F000 bytes, type:Available
Region 1: Start: 0x09F000, length: 0x01000 bytes, type:Reserved
Region 2: Start: 0x0E8000, length: 0x018000 bytes, type:Reserved
Region 3: Start: 0x0100000, length: 0x01EF0000 bytes, type:Available
Region 4: Start: 0x01FF0000, length: 0x010000 bytes, type:ACPI Reclaim
Region 5: Start: 0x0FFFC0000, length: 0x040000 bytes, type:Reserved

If you look at kernel_main.c, you will see that a new function called *InitialisePhysicalMemory*() has been added and this function is called from *Initalise*(). However, this code is conditionally compiled out. Once you have implemented the functions required for this assessment (see below), you should uncomment the following line at the top of kernel_main.c to ensure that the physical memory manager is initialised correctly:

```
#define PMM
```

Your task for this assessment is to write the physical memory manager functions. It is recommended that you put these into a new source code file.

The specification for the functions is below. The file physicalmemorymanager.h includes definitions of all of these functions and is included in kernel_main.c

Implementation

The addressable memory range should be considered as a number of blocks of memory of a specific size, each of which can be marked as used or not used. Regions of memory that were listed in the memory map as not being available should be considered as being used. You need to maintain a memory usage bitmap that uses one bit for each block of memory to indicate whether it is used or not, i.e. the first bit in the bitmap is used to indicate whether the first block of memory is used, the second bit in the bitmap is used to indicate whether the second block in memory is used, and so on.

I would suggest that the size of a block of memory should be 4096 bytes (this will then ensure that it works with the virtual memory manager functions we will be creating later).

Functions

*uint32_t PMM_Initialise(BootInfo * bootInfo, uint32_t bitmap);*

Parameters:

bootinfo: A pointer to the BootInfo structure passed to the kernel by the boot process

bitmap: The memory address where the bitmap should be located.

Returns: The size of the bitmap. This should be a multiple of 4.

Initialise the physical memory manager. It needs to perform the following tasks:

- Determine the optimum size of the bitmap and locate it at the address specified in *bitmap*.
- Initialise the bitmap to indicate that all blocks in the address space are used (this will ensure that any regions not included in the memory map are flagged as used, avoiding the situation where you might try to allocate some memory that does not exist).
- Use the memory map pointed to by the BootInfo structure to specify which blocks of memory are available and mark them as unused.

void PMM_MarkRegionAsAvailable(uint32_t base, size_t size);

Parameters:

base: The base address of the region of memory to mark as unused

size: The size of the region of memory (in bytes).

This function should mark the region of memory specified as being unused. Only complete blocks should be marked. If the base address is not the start of a block, then it should be adjusted to the start of the block that contains that address. If the size does not indicate a complete number of blocks, it should be adjusted to the number of blocks needed to include the region.

void PMM_MarkRegionAsUnavailable(uint32_t base, size_t size);

Parameters:

base: The base address of the region of memory to mark as unavailable.

size: The size of the region of memory (in bytes).

This function should mark the region of memory specified as being used. Only complete blocks should be marked. If the base address is not the start of a block, then it should be adjusted to the start of the block that contains that address. If the size does not indicate a complete number of blocks, it should be adjusted to the number of blocks needed to include the region.

void PMM_AllocateBlock();*

Returns: A pointer to the block of memory that has been allocated.

Allocate a block of memory and mark it as used.

void PMM_FreeBlock(void p);*

Parameter:

p: The memory address of a block of memory previously allocated.

Free the block of memory at the specified address and mark it as unused.

*void * PMM_AllocateBlocks(size_t size);*
Parameter:

size: The number of blocks of memory to be allocated.

Returns: A pointer to the block of memory that has been allocated.

Allocates *size* blocks of memory and mark them as being used..

void PMM_FreeBlocks(void p, size_t size);*
Parameter:

p: The memory address of the blocks of memory previously allocated.

size: The number of blocks to free

Free the blocks of memory at the specified address and mark them as unused.

size_t PMM_GetAvailableMemorySize();
Returns: The amount of available memory (in K).

Return the total amount of memory that can be used by the system in K (i.e. the total amount of memory that was indicated as being available in the memory map).

uint32_t PMM_GetAvailableBlockCount();
Returns: The total number of blocks that are available (both used and unused).

uint32_t PMM_GetUsedBlockCount();
Returns: The number of blocks that are currently used.

uint32_t PMM_GetFreeBlockCount();
Returns: The number of blocks that are currently unused.

uint32_t PMM_GetBlockSize();
Returns: The size of a block (should be 4096).

uint32_t PMM_GetMemoryMap();
Returns: The address of the bitmap.

Testing

A big part of the challenge of this assignment is how you test your memory management functions. I want to see a detailed plan for how you test that what you have written meets the requirements and is robust and the results of that testing. This should be included in a Microsoft Word (.doc or .docx) or PDF file.

Submission

You must submit a zip file containing your submission to submission point “Assessment 2 Submission Point” by the due date and time. Any submission that Course Resources indicates as being submitted late will not be marked and a grade of 0 given for this component.

The zip file must contain:

- The complete folder for the submission, including the files you used as a starting point.
- The folder must be cleaned before you create the zip file. This means that it should be as if you had run ‘make clean’ on the folder. Your submission will be tested by using make to build the disk image containing your code and then it will be executed in Bochs (using the .img file) and under VMWare Workstation 12 (using the .iso file).
- Your testing document..

The name of the zip file should be Assessment2_xxxxxxxxx.zip where xxxxxxxxx is your student number. For example, if your student number was 123456789, your zip file would be called Assessment2_123456789.zip. Any file that is not named using this convention or if something other than a zip file is submitted will result in the assessment not being marked and a grade of 0 for this assessment being given for this component.