# Chapter 8: Paging and Dynamic Memory in 539kernel

## Introduction

The last result of this chapter is version `G` of 539kernel which contains the basic stuff that are related to the memory. Previously, we have seen that we have no way in 539kernel to allocate memory dynamically, due to that, the allocation of entries of processes table and the process control block was a static allocation. Making dynamic allocation possible is an important thing since a lot of kernel's objects need to be allocated dynamically. Therefore, the first memory-related thing to implement is a way to allocate memory dynamically. The other major part of version `G` is implementing paging by using x86 architecture's support. Since there is no way yet in 539kernel to access the hard disk, virtual memory cannot be implemented yet. However, basic paging can be implemented and this can be used as basis for further development.

## Dynamic Memory Allocation

In our normal process of developing applications by using programming languages that don't employ garbage collection, we are responsible for allocating spaces from memory. When we need to store data in memory, a free space in memory should be available for this data to put this data in. The process of telling that we need `n` bytes from memory to store some data is known as memory allocation. There are two possible ways to allocate memory, statically or dynamically.

Usually, a static memory allocation is used when we know the size of data at compile time, that is, before running the application that we are developing. Dynamic memory allocation is used when the size of data will be known at run time. Static memory allocation is the responsibility of the compiler of the language that we are using, while the dynamic memory allocation is the responsibility of the programmer [1], also, the regions that we have allocated dynamically should be freed manually [2].

As we have seen, there are multiple region of a running process's memory and each region has a different purpose, we already discussed run-time stack which is one of those region. The other data region of a process is known as *run-time heap*, or *heap* for short, but I prefer to use the long term to distinct the concept that we are discussing from a data structure also known as heap. When we allocate memory dynamically, the memory region that we have allocated is a

[1] Not in all cases though.

[2] This holds true in the case of programming languages like C. New system programming languages such as Rust for example may have different ways to deal with the matter. However, what we are discussing here is the basis, depending on this basis more sophisticated concepts (e.g. Rust) can be built.

part of the run-time heap, which is a large region of process memory that is used for dynamic allocation, in C, for example, the most well-known way to allocate bytes dynamically, that is, from the run-time heap is to use the function `malloc` which implements an algorithm known as *memory allocator*. The run-time heap need to be managed, due to that, this kind of algorithms use data structures that maintain information about the allocated space and free space.

A need of dynamic memory allocation have shown up previously in 539kernel. Therefore, in the current version 539kernel we are going to implement the most basic memory allocator possible. Through a new function `kalloc` [3], which works in a similar way to `malloc`, a bunch of bytes can be allocate from the kernel's run-time heap, the starting memory address of this allocated region will be returned by the function, after that, the region can be used to store whatever we wish. The stuff that are related to the kernel's run-time heap will be defined in a new file `heap.c` and its header file `heap.h`, let's start with the latter which is the following.

```
unsigned int heap_base;

void heap_init();
int kalloc( int );
```

A global variable known as `heap_base` is defined, this variable contains the memory address that the kernel's run-time heap starts from, and starting from this memory address we can allocate user's needed bytes through the function `kalloc` which its prototype is presented here.

As usual, with each subsystem in the kernel, there is an initialization function that sets the proper values and does whatever needed to make this subsystem ready to use, as you may recall, these functions are called right after the kernel starts in protected mode, in our current case `heap_init` is the initialization function of the kernel's run-time heap. We can now start with `heap.c`, of course, the header file `heap.h` is needed to be included in `heap.c`, and we begin with the code of `heap_init`.

```
#include "heap.h"

void heap_init()
{
    heap_base = 0x100000;
}
```

As you can see, the function `heap_init` is too simple. It sets the value `0x100000` to the global variable `heap_base`. That means that kernel's run-time heap starts from the memory address `0x100000`. In `main.c` we need to call this function in the beginning to make sure that dynamic memory allocation is ready and usable by any other subsystem, so, we first add `#include "heap.h"` in including

[3]Short for *kernel allocate*

section of `main.c`, then we add the call line `heap_init();` in the beginning of `kernel_main` function. Next is the code of `kalloc`.

```
int kalloc( int bytes )
{
    unsigned int new_object_address = heap_base;

    heap_base += bytes;

    return new_object_address;
}
```

Believe it or not! This is a working memory allocator that can be used for dynamic memory allocation. It's too simple, though, it has some disadvantages but in our case it is more than enough. It receives the number of bytes that the caller needs to allocate from the memory through a parameter called `bytes`.

In the first step of `kalloc`, the value of `heap_base` is copied to a local variable named `new_object_address` which represents the starting memory address of newly allocated bytes, this value will be returned to the caller so the latter can start to use the allocated memory region starting from this memory address.

The second step of `kalloc` adds the number of allocated bytes to `heap_base`, that means the next time `kalloc` is called, it starts with a new `heap_base` that contains a memory address which is right after the last byte of the memory region that has been allocated in the previous call. For example, assume we called `kalloc` for the first time with `4` as a parameter, that is, we need to allocate four bytes from kernel's run-time heap, the base memory address that will be returned is `0x100000`, and since we need to store four bytes, we are going to store them on the memory address `0x100000`, `0x100001`, `0x100002` and `0x100003` respectively. Just before returning the base memory address, `kalloc` added `4`, which is the number of required bytes, to the base of the heap `heap_base` which initially contained the value `0x100000`, the result is `0x100004` which will be stored in `heap_base`. Next time, when `kalloc` is called, the base memory address of the allocated region will be `0x100004` which is, obviously, right after `0x100003`.

As you can see from the allocator's code, there is no way to implement `free` function, usually, this function takes a base memory address of a region in run-time heap and tells the memory allocator that the region which starts with this base address is free now and can be used for other allocations. Freeing memory regions when the applications finishes from using the helps in ensuring the the run-time heap is not filled too soon, when an application doesn't free up the memory regions that are not needed anymore, it causes a problem known as *memory leak*.

In our current memory allocator, the function `free` cannot be implemented because there is no way to know how many bytes to free up given the base address of a memory region, returning to the previous example, the region of run-time heap which starts with the base address `0x100000` has the size of `4` bytes, if we want to tell the memory allocator to free this region, it must know what is the size of this region which is requested to be freed, that of course means that the memory allocator needs to maintain a data structure that can be used at least when the user needs to free a region up, one simple way to be able to implement `free` in our current memory allocator is to modify `kalloc` and make it uses, for example, a linked-list, whenever `kalloc` is called to allocate a region, a new entry is created and inserted into the linked-list, this entry can be stored right after the newly allocated region and contains the base address of the region and its size, after that, when the user request to free up a region by giving its base memory address, the `free` function can search in this linked-list until it finds the entry of that region and put on the same entry that this region is now free and can be used for future allocation, that is, the memory which was allocated once and freed by using `free` function, can be used later somehow.

Our current focus is not implementing a full memory allocator, so, it is up to you as a kernelist to decide how your kernel's memory allocator works, of course, there are a bunch of already exist algorithm as we have mentioned earlier.

## Using The Allocator with Process Control Block

To make sure that our memory allocator works fine, we can use it when a new process control block is created. It also can be used for processes table, as you may recall, the processes table from version `T` is an array which is allocated statically and its size is `15`, instead, the memory allocator can be used to implement a linked-list to store the list of processes. However, for the sake of simplicity, we will stick here with creating PCB dynamically as an example of using `kalloc`, while keeping the processes table for you to decide if it should be a dynamic table or not and how to design it if you decide that it should be dynamic.

The first thing we need to do in order to allocate PCBs dynamically is to change the parameters list of the function `process_create` in both `process.h` and `process.c`. As you may recall, in version `T`, the second parameter of this function called `process` and it was the memory address that we will store the PCB of the new process on it. We had to do that since dynamic memory allocation wasn't available, so, we were creating local variables in the caller for each new PCB, then we pass the memory address of the local variable to `process_create` to be used for the new PCB. This second parameter is not needed anymore since the region of the new PCB will be allocated dynamically by `kalloc` and its memory address will be returned by the same function. So, the prototype of the function `process_create` will be in `process.h` and `process.c` respectively as the following.

```
process_t *process_create( int * );
```

```
process_t *process_create( int *base_address )
```

You can also notice that the function now returns a pointer to the newly created PCB, in version `T` it was returning nothing. The next changes will be in the code of `process_create`. The name of the eliminated parameter of `process_create` was `process` and it was a pointer to the type `process_t`. We substitute it with the following line which should be in the beginning of `process_create`.

```
process_t *process = kalloc( sizeof( process_t ) );
```

Simply, we used the same variable name `process` but instead of getting it as a parameter, we called the memory allocator to allocate a region that has the same size of the type `process_t` from the kernel's run-time heap, exactly as we do in user-space applications development, so, the new memory region can be used to store the new PCB. In the last of `process_create` we should add the line `return process;` to return the memory address for the newly created PCB for the new process.

In version `T` we have called `process_create` in `main.c` to create four processes, we need to change the calls by omitting the second parameter, also the line `process_t p1, p2, p3, p4;` in `main.c` which was allocating memory for the PCBs can be removed since we don't need them anymore. The calls of `process_create` will be as the following.

```
process_create( &processA );
process_create( &processB );
process_create( &processC );
process_create( &processD );
```

## Paging

In this section we are going to implement a basic paging for 539kernel. To do that, a number of steps should be performed. A valid page directory should be initialized and its address should be loaded in the register `CR3`. Also, paging should be enabled by modifying the value of `CR0` to tell the processor to start using paging and translate linear memory addresses by using the page tables instead of consider those linear addresses as physical addresses. We have mentioned earlier, for each process we should define a page table, however, in this section we are going to define the page table of the kernel itself since this is the minimum requirement to enable paging.

The page size in 539kernel will be `4KB`, that means we need a page directory that can point to any number of page tables up to `1024` page table. The mapping itself will be *one-to-one mapping*, that is, each linear address will be translated to a physical address and both are identical. For example, in one-to-one mapping the linear address `0xA000` refers to the physical address `0xA000`. This choice has been made to make things simple, more advanced designs can be used instead. We already know the concept of page frame, when the page size is `4KB` that means page frame `0` is the memory region that starts from the memory address `0` to `4095d`. One-to-one mapping is possible, we can simply define the first entry of the first page table [4] to point to page frame `0` and so on. The memory allocator will be used when initializing the kernel's page directory and page tables, we can allocate them statically as we have done with `GDT` for example, but that can increase the size of kernel's binary file.

Before getting started with the details two new files are needed to be created: `paging.h` and `paging.c` which will contain the stuff that are related to paging. The content of `paging.h` is the following.

```
#define PDE_NUM 3
#define PTE_NUM 1024

extern void load_page_directory();
extern void enable_paging();

unsigned int *page_directory;

void paging_init();
int create_page_entry( int, char, char, char, char, char, char, char, char );
```

The part `PDE` in the name of the macro `PDE_NUM` means page directory entries, so this macro represents the number of the entries that will be defined in the kernel's page directory. Any page directory may hold `1024` entries but in our case not all of these entries are needed so only `3` will be defined instead, that means only three page tables will be defined for the kernel. How many entries will be defined in those page tables is decided by the macro `PTE_NUM` which `PTE` in its name means page table entries, its value is `1024` which means there will be `3` entries in the kernel's page directory and each one of them points to a page table which has `1024` entries. The total entries will be `3 * 1024 = 3072` and we know that each of these entries map a page frame of the size `4KB` then `12MB` of the physical memory will be mapped in the page table that we are going to define, and since our mapping will be one-to-one, that means the reachable physical memory addresses start at `0` and ends at `12582912`, any region beyond this range, based on our setting, will not be reachable by the kernel and it is going to cause a page fault exception. It is your choice to set the value of `PDE_NUM` to the maximum (`1024`), this will make a `4GB` of memory addressable.

Getting back to the details of `paging.h`, both `load_page_directory` and `enable_paging` are external functions that will be defined in assembly and will be used in `paging.c`. The first function loads the address of the kernel's page directory in the register `CR3`, this address can be found in the global variable `page_directory` but of course, its value will be available after allocating the needed space by `kalloc`. The second function is the one that modifies the register `CR0` to enable paging in x86, this should be called after finishing the initialization of kernel's page directory and loading it.

[4]The first page table is the one which is pointed to by the first entry in the page directory.

## Initializing Kernel's Page Directory and Tables

From our previous encounter with the structure of page directory/table entry, we know that the size of this entry is 4 bytes and has a specific arrangement of the bits to indicate the properties of the entry being pointed to. The function `create_page_entry` helps in building a value to be stored in a page directory/table entry based on the properties that should be enabled and disabled, this value will be returned to the caller. As you can see from `paging.h`, it returns an integer and that makes sense, as we know, the size of integer in `32-bit` architecture C is 4 bytes, exactly same as the size of an entry. The following is the code of `create_page_entry` that should be defined in `paging.c`.

```
int create_page_entry( int base_address, char present, char writable, char privilege_level,
{
    int entry = 0;

    entry |= present;
    entry |= writable << 1;
    entry |= privilege_level << 2;
    entry |= write_through_cache << 3;
    entry |= cache_enabled << 4;
    entry |= accessed << 5;
    entry |= dirty << 6;
    entry |= page_size << 7;

    return base_address | entry;
}
```

As you can see, each parameter of `create_page_entry` represents a field in the entry of page directory/table, the possible values of all of them but `base_address` are either `0` or `1`, the meaning of each value depends on the flag itself and we already have covered them. By using bitwise operations we put each flag in its correct place.

The base address represents the base memory address of a page table in case we are creating a page directory entry, while it represents the base memory address of a page frame in case we are creating a page table entry. This base address will be `OR`red with the value that is generated to represent the properties of the entity that the current entry is pointing to, we will discuss more details about the base memory address when we start talking about page-aligned entries.

Now we can use `create_page_entry` to implement the function `paging_init` which should reside in `paging.c`. This function will be called when the kernel switches to protected-mode, as the usual with initialization functions, its job is creating the kernel's page directory and kernel's page tables that implement one-to-one map based on the sizes that defined in the macros `PDE_NUM` and `PTE_NUM`. The code of `paging_init` is the following.

```c
void paging_init()
{
    // PART 1:

    unsigned int curr_page_frame = 0;

    page_directory = kalloc( 4 * 1024 );

    for ( int currPDE = 0; currPDE < PDE_NUM; currPDE++ )
    {
        unsigned int *pagetable = kalloc( 4 * PTE_NUM );

        for ( int currPTE = 0; currPTE < PTE_NUM; currPTE++, curr_page_frame++ )
            pagetable[ currPTE ] = create_page_entry( curr_page_frame * 4096, 1, 0, 0, 1, 1,

        page_directory[ currPDE ] = create_page_entry( pagetable, 1, 0, 0, 1, 1, 0, 0, 0 );
    }

    // ... //

    // PART 2

    load_page_directory();
    enable_paging();
}
```

For the sake of simpler discussion, I have divided the code of the function into two parts and each part is indicated by a heading comment. The job of the first part is to create the page directory and the page tables, based on the default values of `PDE_NUM` and `PTE_NUM`, three entries will be defined in the page directory, each one of them points to a page table that contains `1024` entries.

First, we allocate `4 * 1024` from the kernel's heap for the page directory, that's because the size of each entry is `4` bytes, but as you can see, while we need only three entries for the page directory, we are allocating memory for `1024` entries instead, the reason of that is the following: the base memory address of a page table should be page-aligned, also, the base memory address of a page frame should be page-aligned. When the page size is `4KB`, then a memory address that we can describe as a *page-aligned memory address* is the one that is a multiple of `4KB`, that is, a multiple of `4096`. In other words, it should be dividable by `4096` with no remainder. The first six multiples of `4KB` are `0 = 4096 * 0`, `4096 = 4096 * 1`, `8192 = 4096 * 2` (`8KB`), `12288 = 4096 * 3` (`12KB`), `16384 = 4096 * 4` (`16KB`), `20480 = 4096 * 5` (`20KB`) and so on. Each one of those value can be considered as a page-aligned memory address when the page size is `4KB`.

Let's get back to the reason of allocating `4 * 1024` bytes for the page directory instead of `4 * 3` bytes. We know that memory allocator sets the base of the heap from the memory address `0x100000`, also, we know, based on the code order of the kernel that `paging_init` will be the first code ever that calls `kalloc`, that is, `kalloc` will be called the first time in 539kernel when we allocate a region for kernel's page directory in the line `page_directory = kalloc( 4 * 1024 );` which means that the memory address of kernel's page directory will be `0x100000` (`1048576d`) which is a page-aligned memory address since `1048576 / 4096 = 256` [5] with no remainders.

When we allocate `4 * 1024` bytes for the page directory, the next memory address that will be used by the memory allocator for the next allocation will be `1048576 + ( 4 * 1024 ) = 1052672` (`0x101000`) which is also a page-aligned memory address, let's call this the first case. The second case is when we allocate `4 * 3` bytes for the page directory instead, the next memory address that the memory allocator will use for the next allocation will be `1048576 + ( 4 * 3 ) = 1048588` (`0x10000C`) which is not a page-aligned memory address which cannot be used as a base memory address for a page table.

If you continue reading the function `paging_init` you will see that the next thing that will be allocated via `kalloc` after that page directory is the first page table which should be in a page-aligned memory address, due to that, we have used the first case which ensures that the next call of `kalloc` is going to return a page-aligned memory address instead of the second case which will not.

Getting back to the first part of `paging_init`, as you can see, it is too simple, it allocates regions from the kernel's heap for the page directory and the entries of the three page tables. Then each entry in both page table and page directory is being filled by using the function `create_page_entry`. Let's start with the line which defines entries in a page table.

```
create_page_entry( curr_page_frame * 4096, 1, 0, 0, 1, 1, 0, 0, 0 )
```

Given that the size of a page is `4KB`, then, page frame number `0` which is the first page frame starts at the physical memory address `0` and ends at physical memory address `4095`, in the same way, page frame `1` starts at the physical memory address `4096` and ends at the physical memory address `8191` and so on. In general, given `n` is the number of a page frame and the page size is `4KB`, then `n * 4096` is the physical memory address that this page frame starts at. We use this equation in the first parameter that we pass to `create_page_entry` when we create the entries that point to the page frames, that is, page tables entries. The local variable `curr_page_frame` denotes the current page frame that we are defining an entry for, and this variable is increased by `1` with each new page table entry. In this way we can ensure that the page tables that we are defining use a one-to-one map.

As you can see from the rest of the parameters, for each entry in the page table, we set that the page frame is present, its cache is enabled and write-through

[5] In hexadecimal: 0x100000 / 0x1000 = 0x100

policy is used. Also, the page frame belongs to supervisor privilege level and the page size is `4KB`.

The code which define a new entry in the page directory is similar to the one which define an entry in a page table, the main different is, of course, the base address which should be the memory address of the page table that belongs to the current entry of the page directory. When we allocate a memory region for the current page table that we are defining, its base memory address will be returned by `kalloc` stored in the local variable `pagetable` which is used as the first parameter when we define an entry in the page directory.

### The Need of Page-aligned Memory Addresses

In the previous section we have discussed the meaning of a page-aligned memory address, and we stated the fact that any base memory address that is defined in a page directory/table entry should be a page-aligned memory address. Why? You may ask.

Recalling the structure of page directory/table entry, it is known that the number of bits that are dedicated for the base memory address are `20` bits (`2.5` bytes), also, we know that in `32-bit` architecture, the size of the largest memory address (`0xFFFFFFFF`) is of size `32` bits.

Now, assume that we want to define a page table entry that points to the last possible page frame which its base address is `0xFFFFF000`. To store this full address `32` bits are needed [6] but only `20` bits are available for base memory address in the page table entry, so, how can we point to this page frame since we can't store its full address in the entry?

The numbers that we have defined previously as page-aligned numbers, in other words, the multiples of `4096`, have an interesting property when they are represented in hexadecimal format, they always end with three zeros! [7] In our current example of the last possible page frame, we need to store `0xFFFFF000` as a base memory address, you can see that it ends with three zeros which means that this number is a page-aligned number. Removing the last three zeros of the example memory address gives us the value `0xFFFFF` which exactly needs `20` bits to be stored, so, due to that the base address the is stored in page directory/table should be a page-aligned memory address which makes it possible to remove the last three zeros from it and make its size `20` bits and later on it will be possible to get the correct base address from the entry, simply, by appending three zeros to it. In `create_page_entry` the place of these three zeros were used to store the properties of the entry when we `OR`red the base address with the value that has been built to represent the properties.

[6] Remember, each hexadecimal digit represents a nibble. One byte consists of two nibbles.

[7] And that makes sense, the first one of them after zero is `0x1000` (`4096d`) and to get the next one you need to add `0x1000` on the previous one and so on.

### Loading Kernel's Page Directory and Enabling Paging

The second part of the function `paging_init` performs two operations, the first one is loading the content of the global variable `page_directory` in the register `CR3`, that is, loading the kernel's page directory so that the processor can use it when the second operation, which enables the paging, is performed.

Because both of these functions need to access the registers directly, they will be written in assembly in the file `starter.asm`. Till now, it is the first time that we define a function in assembly and use it in C code, to do that we need to add the following lines in the beginning of `starter.asm`.

```
extern page_directory

global load_page_directory
global enable_paging
```

There is nothing new in the first line. We are telling NASM that there is a symbol named `page_directory` that will be used in the assembly code, but it isn't defined in it, instead it's defined in a place that the linker is going to tell you about in the future. As you know, `page_directory` is the global variable that we have defined in `paging.h` and holds the memory address of the kernel's page directory, it will be used in the code of `load_page_directory`.

The last two lines are new, what we are telling NASM here that there will be two labels named `load_page_directory` and `enable_paging`, both of them should be global, that is, they are reachable by places other than the current assembly code, in our case, it's the C code of the kernel. The following is the code of those functions, they reside in `starter.asm` below the line `bits 32` since they are going to run in `32-bit` environment.

```
load_page_directory:
    mov eax, [page_directory]
    mov cr3, eax

    ret

enable_paging:
    mov eax, cr0
    or eax, 80000000h
    mov cr0, eax

    ret
```

There is nothing new here. In the first function we load the content of `page_directory` into the register `CR3` and in the second function we use bit-wise operation to modify bit `31` in `CR0` and sets its value to `1` which means enable paging. Finally, `paging_init` should be called by `kernel_main` right after `heap_init`, the full list of calls in the proper order is the following.

```
heap_init();
paging_init();
screen_init();
process_init();
scheduler_init();
```

## Finishing up Version `G`

And now version `G` of 539kernel is ready. It contains a basic memory allocator and a basic paging. The following is its `Makefile` which adds the new files to the compilation list.

```
ASM = nasm
CC = gcc
BOOTSTRAP_FILE = bootstrap.asm
SIMPLE_KERNEL = simple_kernel.asm
INIT_KERNEL_FILES = starter.asm
KERNEL_FILES = main.c
KERNEL_FLAGS = -Wall -m32 -c -ffreestanding -fno-asynchronous-unwind-tables -fno-pie
KERNEL_OBJECT = -o kernel.elf

build: $(BOOTSTRAP_FILE) $(KERNEL_FILE)
    $(ASM) -f bin $(BOOTSTRAP_FILE) -o bootstrap.o
    $(ASM) -f elf32 $(INIT_KERNEL_FILES) -o starter.o
    $(CC) $(KERNEL_FLAGS) $(KERNEL_FILES) $(KERNEL_OBJECT)
    $(CC) $(KERNEL_FLAGS) screen.c -o screen.elf
    $(CC) $(KERNEL_FLAGS) process.c -o process.elf
    $(CC) $(KERNEL_FLAGS) scheduler.c -o scheduler.elf
    $(CC) $(KERNEL_FLAGS) heap.c -o heap.elf
    $(CC) $(KERNEL_FLAGS) paging.c -o paging.elf
    ld -melf_i386 -Tlinker.ld starter.o kernel.elf screen.elf process.elf scheduler.elf heap
    objcopy -O binary 539kernel.elf 539kernel.bin
    dd if=bootstrap.o of=kernel.img
    dd seek=1 conv=sync if=539kernel.bin of=kernel.img bs=512 count=8
    dd seek=9 conv=sync if=/dev/zero of=kernel.img bs=512 count=2046
    #bochs -f bochs
    qemu-system-x86_64 -s kernel.img
```