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 should be there for this data so we can put it in this specific free space. The process of telling that we need `n` bytes and we are going to get these bytes from a specific free memory region, this process 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 part of the run-time heap, which is a large region of process memory that is used

[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 and based on it more sophisticated concepts (e.g. Rust) can be built.

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 are used with 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 to store. 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 run-time heap starts from, and starting from this memory address we can allocate the needed bytes through the function `kalloc` which its prototype is presented here. As usual, with each subsystem in the kernel, there is an initialize 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 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 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`.

[3] Short for *kernel allocate*

```c
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. The second step of `kalloc` is adding the number of allocated bytes to `heap_base`, that means the next time `kalloc` is call, 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 contains 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 the base memory address of the region of run-time heap then it tells the memory allocator that a specific region of memory that was reserved once by using the allocation function, this region is now free and can be used for other allocations. In this way we can make sure that the run-time heap is not filled too soon, when an application fails to free up the memory region that is not used anymore, it causes a problem known as *memory leak*. In our current memory allocator, there is no way to know how many bytes to free up given the base address of a memory region in the run-time heap, 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 [4], whenever a new region needed and `kalloc` is called to allocate the region, a new entry is created and inserted into the linked-list, this entry can be stored right after the newly allocated region, this entry 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, after that, the freed up memory, 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 dynamically 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 prototype 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 process being created 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 notices that the function know 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`. As we have mentioned earlier, the name of the

[4] A data structure.

second 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 from the parameter we called the memory allocator to allocate from the kernel's run-time heap a region that has the same size of the type `process_t`, 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 );
```