Assignment-1: Creating our first context

Description

At this stage, gemOS is in 64-bit mode executing itself as the first context (say the boot context). The boot context sets up the page table, stack, segment registers for itself. Further, it implements basic input output to a serial console. and puts itself into a basic shell. One of the commands un-implemented in gemOS shell is `launch`. In this assignment, you are required to implement the launch functionality. The `launch` functionality requires creating another context and execute a function before returning back to the basic shell. To make matters simpler, I have created the definition of a context and written code to execute the function in the new context. As part of the assignment, the paging structures for the new context as per its virtual address layout has to be built as per the specifications.

You are required to implement two functions in a C file named `context_mm.c`. The prototype of these two functions are as follows,

```
void prepare_context_mm (struct exec_context *ctx);
void cleanup_context_mm(struct exec_context *ctx);
```

Both the functions take an execution context as argument and do not return anything.

Execution context

A context is declared as a structure in `include/context.h`.

```
struct exec_context{
        u32 id;
        u8 type;
        u8 status;
        u16 used_mem;
        u32 pgd;
        struct mm_segment mms[MAX_MM_SEGS];
        char name[CNAME_MAX];
        u32 arg_pfn;
        u32 num_args;
};
```

The relevant fields for this assignment are,

`pgd` This is the physical frame number of the first level translation in the page table for a x86_64. This has to be provided by your code. The CR3 register will be loaded (by existing code) appropriately using this value.

`mms` This is an array of different virtual address segments, where each segment is defined by a structure named `mm_segment`. This is already populated, see the description below.

`arg_pfn` This is the physical frame number used to pass arguments to the the program, which is also to be mapped by the data segment virtual address (see below).

Virtual memory partitions (mm_segment)

Virtual memory area of the new context is partitioned into three segments—(i) stack, accessed as `ctx->mms[MM_SEG_STACK]`, (ii) code, accessed as `ctx->mms[MM_SEG_CODE]`, and, (iii) data segment, accessed as `ctx->mms[MM_SEG_DATA]`. You are required to allocate *one physical frame* for each segment (except for DATA which uses arg_pfn in the context parameter) and populate page tables for the first 4KB virtual address defined by each segment to point to the physical frame. The details of virtual memory segment structure (declared in `include/context.h`) is as follows,

```
struct mm_segment{
                unsigned long start;
                unsigned long end;
                u32 access_flags;   /*R=1, W=2, X=4, S=8*/
};
```

The start address, end address and access permissions for each segment is already populated. Only one virtual page at the starting address for each virtual segment is to be mapped using page tables. You are required to reflect the virtual address access permissions (`access_flags`, only the R and W are used for this assignment) in the page table structures and mark the entries accessible from user.

For each virtual address segment (except for `MM_SEG_DATA`), you are required to allocate user physical pages (data pages) and page table pages. For `MM_SEG_DATA`, you must use the `arg_pfn` as the data page and allocate the page table pages as necessary. To allocate physical pages required for page table and data pages, following API is provided.

```
u32 os_pfn_alloc (u32 region)
```

This function allocates a physical frame from a physical memory region and returns the frame number. The region parameter can be `OS_PT_REG` or `USER_REG`. For allocating physical memory used for page tables (page table pages), you should use `OS_PT_REG` and for physical pages containing the actual data, the region should be `USER_REG`. Before an allocated PFN can be used (addressed), the virtual address for the PFN for the boot context should be obtained by invoking `osmap(pfn)`. This is useful for accessing page table memory and create entries.

To implement the `cleanup_context_mm`, you are required to free all the physical pages used for page tables as well as data pages for all virtual address segments. To free physical pages, you need to use the following function,

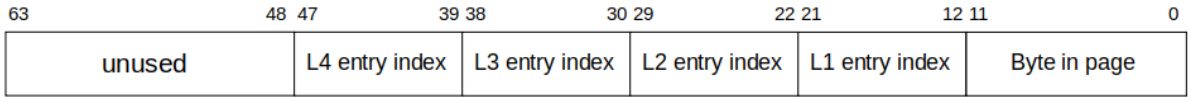
```
void os_pfn_free(u32 region, u64 pfn)
```

where, `region` is same as before and `pfn` is the physical frame number. Next, let us understand the page table structure for this assignment.

Paging

In x86_64 based systems, the processes are provided with 64-bit virtual addresses in which only the least significant 48 bits (47-0) are usable. The remaining 16 bits (63-48) are reserved and should be same as the MSB (47) of the usable bits. For this assignment lets keep the bits 63-47 as zeroes. Given a page of size 4KB,

the rightmost 12 bits (11-0) are used to point to a single byte in the page. The remaining 36 bits are used to find the frame number in RAM where the page belongs. A 4-depth radix tree is used for translating the page number to corresponding frame number. The 36 bits are further divided into 4 groups of 9 bits each as shown below (L4, L3, L2 and L1 entry indices). Each group is used for a different level of the radix tree.

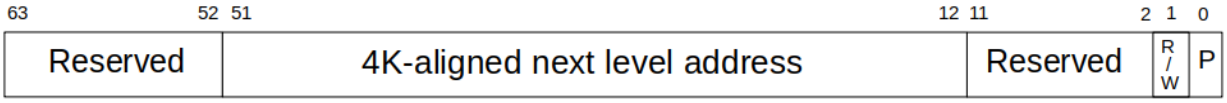


The radix tree used for translation are again made of pages of same size and starts with a single page. Lets call these pages as page table pages (PTP) and the actual pages of the process as data pages. The topmost level page table is called L4 PT and it goes down to L3, L2 and finally L1 pages. These pages store physical addresses of next level pages and the last level (L1) page tables store physical addresses of the data pages.

Each 9 bits group of the virtual address is used to index of 8 bytes of data called page table entry (PTE) in a page table page ($4KB/8bytes = 512 = 2^9$) that contains the physical address of next level page. Thus each page table page contains 512 page entries.

Structure of Page Table Entry

Now that we know each page table contains 512 entries, some entries are valid depending on the virtual address usage. Each page table entry contain some flag bits along with the location of next level physical frame number. In a 64-bit (8 byte) page table entry, 40 bits are used to store physical page number of next level page table or data page and some of the remaining bits are used for special purposes as detailed below.



- bit 0 - present

  Value 0 indicates the virtual memory range covered by this entry does not have any mapping. Value 1 indicates that the page table entry contains a valid next level address.

- bit 1 - read-only/write-enabled

  Value 0 indicates the virtual address range covered by this entry is read-only.

  Value 1 indicates the data under this virtual address range are write-enabled provided no page entries in the further level have this bit set to read-only.

- bits 8:2 are out of scope for this assignment and shall necessarily be kept as zeroes.

- bits 11:9 are ignored bits but for simplicity you are recommended to set them as 0.

- bits 51:12 - Frame number or 4KB aligned address of next level page

  Multiply by 4096 to get actual physical address of next level page.

- bit 63:52 - must be zeroes