Systems II
Project 4
Paged virtual memory

1 Overview

A substantial change in the virtual address spaces for processes (and the kernel!) via paging.

2 Implementing paged virtual memory

Fivish uses the exact paged virtual memory arrangement that we have described in class: The entire 2^{32}-bit (4 GB) address space is divided into 2^{20} *pages*, each of which is 2^{12} (4,096) bytes.

Furthermore, the *Fivish* simulated processor defines each virtual address space via a *two-level page table*. The *upper page table (UPT)* contains 2^{10} entries, each of which leads to one *lower page table (LPT) segment*. Each such LPT segment itself contains 2^{10} *page table entries (PTEs)*, each of which map a specific *virtual page* to a *physical page frame*. Since each entry in the UPT and each LPT segment contains 1,024 entries, each of which is a 4 byte (32 bit) address, then each such piece of a page table occupies 4 KB.

Simplifying the address space layout: As a simplifying step, we will invert the usual division of the virtual address space:

- `0x00000000` to `0x7fffffff` (the lower half) will be *kernel space*. That is, only the kernel will be allowed to use this range of addresses.
- `0x80000000` to `0xffffffff` (the upper half) will be *user space*. Each process will have access to this range of addresses.

The purpose of this inversion is to make the transition from physical to virtual (paged) addressing simpler. With the kernel space overlapping with the occupied portion of the physical devices, the virtual address spaces can map that entire range of devices with the *identity mapping*: Each virtual address p in that range maps to the physical address p. Transitioning from physical addressing to paged-virtual addressing modes becomes simple, since the addresses will look identical before and after.

3 A suggested approach

The following steps provide a sketch of the changes that need to be made:

- Modify `mm_alloc()`: Change this allocator to divide memory into page frames (whose base addresses must be *page-aligned*)—that is, it should allocate 4 KB blocks instead of the larger blocks we previously used for process spaces.
- Create a *mapping function*:

```
bool map_page (pte_t** upt, address_t virtual_page,
               address_t physical_page, int permissions);
```

  Given a pointer to an *upper page table* (`upt`), map a given `virtual_page` to the provided `physical_page`. This function must index into the UPT. If the entry there is `NULL`, then it must allocate and initialize a new *lower page table (LPT)* and then update the UPT to point to it.

  Given that the UPT entry is (now) non-null, this function should follow its pointer to the LPT. There it should index into the LPT to find the *page table entry (PTE)* that maps this virtual page. If that entry is non-null, the function should return *false* (indicating that it is already mapped); otherwise, it should map the given `physical_page` into that PTE.

  The function must also be sure to set the appropriate bits in the PTE's metadata so that the page is properly accessible. The metadata bits (the 12 least significant bits) are:

  - `[0]` – *mapped*: Whether the mapping is valid.
  - `[1]` – *resident*: Whether the page is resident in RAM.
  - `[4:2]` – *supervisor permissions*: Whether the page is *executable*, *readable*, and/or *writeable* (respectively) when the processor is in *supervisor* mode.
  - `[7:5]` – *user permissions*: Whether the page is *executable*, *readable*, and/or *writeable* (respectively) when the processor is in *user* mode.
  - `[8]` – *referenced*: Whether the page has been referenced (accessed). This bit is set automatically by the MMU.
  - `[9]` – *dirty*: Whether the page has been written. This bit is set automatically by the MMU.
  - `[11:10]` – *unassigned*: These bits are available for other uses.

  The `permissions` argument to this function can be exactly the bits from `[7:2]` of the metadata correspond to the permission bits for the page. That is, we can define the following constants…

```
#define PTE_SREAD  0x04
#define PTE_SWRITE 0x08
```

```
#define PTE_SEXEC  0x10
#define PTE_UREAD  0x20
#define PTE_UWRITE 0x40
#define PTE_UEXEC  0x80
```

…and then *bitwise*-OR them to create a permissions value. For example, if we want a page to be readable and writeable in both user and supervisor mode:

```
int permissions = PTE_SREAD | PTE_SWRITE | PTE_UREAD | PTE_UWRITE;
```

This value can then be passed into `map_page()`, and the function can then use that value (which is a *bit-mask*) to set those bits in the PTE by OR'ing the PTE with the permission value itself.

- Modify the kernel's `main()` function so that, before calling `run_init()`, it calls a new function `init_vm()`, that transitions the system into paged virtual addressing mode. Specifically, it should:
  - Create the *kernel page table* (to which you should keep a pointer in the statics region). That is, create the first UPT.
  - Use the `map_page()` function (above) to map every page from `0x0` to the final page of the last device listed in the device table.
  - Transition by setting the `pt` CSR and then changing the `md` CSR to enable paged virtual addressing.
- Modify `do_run()` so that the creation of a new process involves allocating a new UPT for it. The lower half of each UPT should be a copy of the kernel page table's UPT's lower half (to share the kernel space). The upper half should be assigned mappings sufficient to load its executable image and provide it a stack. This UPT should replace the *base* and *limit* values kept in each process object with a pointer to its UPT.
- Modify `cpu_scheduler()` and `userspace_jump()` so that they set the `pt` CSR of the process about to run.
- Create an `INVALID_ADDRESS` interrupt handler. This function needs to traverse the page table to find the PTE for the virtual address that caused the interrupt. (That address will be stored in the `eai` CSR.) If the access is a permission violation, then the process should be aborted. If the access is to a page that does not yet have a mapping, then a page frame should be allocated and a virtual mapping established with that PTE. The process should then resume, attempting the memory access again.

4 How to submit your work

Copy into the `project-4/` directory in your shared Google Drive folder **all** of your source code: Every `.asm`, `.h`, and `.c` file. Include test programs and data files (if any) that those programs might use. It should be straightfoward to build and run your kernel and test programs.

This assignment is due at **11:59 pm** on **Tuesday, May** 5^{th}**.**