Design notes

Mian Qin UIN:725006574

1 Overview

In this machine problem, we mainly implement two parts. First part is setup page table for the kernel memory using direct mapping. This is used for turning on paging from real mode. The second part is implement page fault handler for kernel demand paging beyond 4MB direct map memory.

For the first part setting up kernel page table, since we are still in the real mode (cpu issue physical address), we can access all physical memory directly. We use the kernel memory pool to allocate frames for page directory (1 frame) and page table (1 frame). This is enough for the 4MB kernel memory. After this, we can turn on paging by loading physical address of page directory to CR3 register and writing the CR0 register,

For the second part implementing page fault handler. We need to handle two kind of page faults. 1, protection fault which includes write privilege fault (write to a read only page) and access privilege fault (user access kernel page). 2, not present fault which we need to allocate a new frame for the demand paging. In the mean time we need to modify the page table and page directory and may also need to allocate from for the page table.

Note, the continuous memory allocator I used is from MP2. I have done more test to make sure the allocator for MP2 works perfectly.

2 Implementation

2.1 init_paging

Init_paging will setup all the global parameters (kernel memory pool, process memory pool, shared memory size) for the paging subsystem.

2.2 PageTable constructor

The constructor will first setup the page table and initialize the entries in the page directory and page table. In this machine problem, we setup the page directory and page table in the kernel memory using the kernel pool to allocate frames.

For the direct map of the first 4MB kernel memory, only the first entry of the page directory is valid (marked R/W and kernel page). The 1024 entries for the page table indirect by the first entry of page directory are all set as valid (marked R/W and kernel page). Page frame of pte are from 0-1023. That finish the page table setup.

2.3 load

Load will load the current page table pointer and load the physical address of page directory to the CR3.

2.4 enable_paging

enable_paging write CR0 register paging bit to enable paging.

2.5 handle_fault

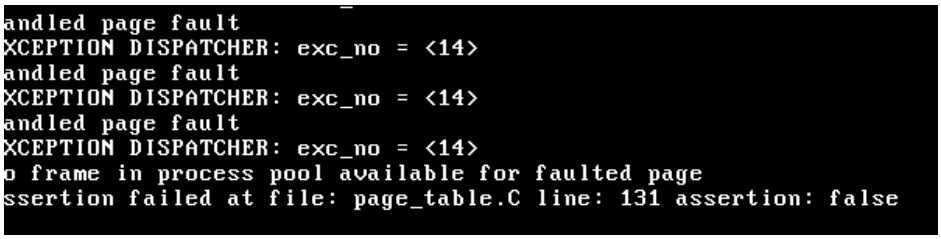
handle_fault handles two kinds of page faults present in the above section, protection fault and not present fault. For protection fault, it will print error messages to the console and hang the system by assert(false). For not present fault, it will allocate physical frames to the demand

page and setup the page table entry. Notice, if there is page table page missing ,we need to allocate frame for page table from the kernel pool (also need to setup entry in page directory to hook up with the page table). However, all the frames for the demand page are from process pool.

3 Debug and verification consideration

For debugging, I hook up with gdb to enable debug feature. This is already done in MP1.
For verification consideration, the original kernel.C gives only one test case to test the page fault handler by sequential write and read memory from 4MB to 5MB. I do more test to further verify the correctness of the page fault handler as follow:

1) More access, the total available memory for process pool is 27MB, I test with 27MB read/write and more than 27MB read/write, if memory access size beyond 27MB, it will print error and hang the system.



2) Different start address for test, since the start address is now virtual address, so it doesn't matter where it start, I test with start address at 400MB, it still works fine.
3) Different access pattern, I test with stride and random access pattern for memory test, all works fine. Here are some test code.

```
int random_offset[5] = {1025,1,10*1024+3,6*1024+1,6*1024+127};
Console::puts("RAMDOM ACCESS tesing...\n");
// random write
for(i=0;i<5;i++)
    foo[NACCESS+random_offset[i]] = random_offset[i];

for (i=0; i<5; i++) {
    if(foo[NACCESS+random_offset[i]] != random_offset[i]) {
        Console::puts("TEST FAILED for access number:");
        Console::putui(i);
        Console::puts("\n");
        break;
    }
}
if(i == 5) {
    Console::puts("RANDOM ACCESS TEST PASSED\n");
}
```