REPORT

PART 1:

DESIGN:

Here we are adding a "*pgtPrint*" system to print the entries of page tables that belong to the current process. For this, we are parsing through the page directory. We wrote the code for this in the *proc.c*, wherein the *pgprint()* function. The pointer to the page directory is available in *myproc()*, and it can be accessed using *myproc()->pgdir*. Before going to the page table using each page directory entry, we check whether the entry is in user mode and valid by using an operator with *PTE_P* and *PTE_U*. There are *NPDENTRIES (2^10=1024)* in the page directory. Now, we will get the page table pointer using each entry of the page directory using *P2V (PTE_ADDR (pde[i]))*. Here *pde[i]* is the i^{th} entry in the page directory. We use the *PTE_ADDR* to remove the first 20 bits of the page directory entry, remove the 12-bit offset, and then *P2V* to get the pointer to the corresponding page table. After getting the corresponding page table of each page directory entry, we parse through the page table to print the entries. We print only those entries in user mode that are valid. We print the physical address as only the first 20 bits of the page table by removing the 12-bit offset. The virtual address is *the ((i << 10) | j) << 12*, (<< is the left-shift operator and | is or operator) where *i* is the page directory entry position and *j* is the page table entry position, we can find this using the diagram below. Finally, we call this sys-call using "*mypgtPrint*," which will print all valid and user entries of all page tables.

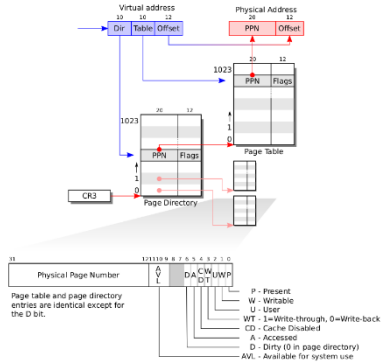

Figure 2-1. x86 page table hardware.

EXPERIMENT WITHOUT DECLARING ANY ARRAYS:

BEFORE IMPLEMENTING DEMAND PAGING:

pgdir entry num:0, Pgt entry number:0, Virtual addr:0, Physical addr: dee2000
pgdir entry num:0, Pgt entry number:2, Virtual addr:2000, Physical addr: dedf000

AFTER IMPLEMENTING DEMAND PAGING:

pgdir entry num:0, Pgt entry number:0, Virtual addr:0, Physical addr: dee2000
pgdir entry num:0, Pgt entry number:2, Virtual addr:2000, Physical addr: dedf000

EXPERIMENT BY DECLARING THE GLOBAL ARRAY:

In user program we created (*mypgtPrint.c*) declare a global array (*int arrGlobal[10000]*).

BEFORE IMPLEMENTING DEMAND PAGING:

pgdir entry num:0, Pgt entry number:0, Virtual addr:0, Physical addr: dee2000
pgdir entry num:0, Pgt entry number:1, Virtual addr:1000, Physical addr: dee0000
pgdir entry num:0, Pgt entry number:2, Virtual addr:2000, Physical addr: dedf000
pgdir entry num:0, Pgt entry number:3, Virtual addr:3000, Physical addr: dede000
pgdir entry num:0, Pgt entry number:4, Virtual addr:4000, Physical addr: dedd000
pgdir entry num:0, Pgt entry number:5, Virtual addr:5000, Physical addr: dedc000
pgdir entry num:0, Pgt entry number:6, Virtual addr:6000, Physical addr: dedb000

pgdir entry num:0, Pgt entry number:7, Virtual addr:7000, Physical addr: deda000
pgdir entry num:0, Pgt entry number:8, Virtual addr:8000, Physical addr: ded9000
pgdir entry num:0, Pgt entry number:9, Virtual addr:9000, Physical addr: ded8000
pgdir entry num:0, Pgt entry number:10, Virtual addr: a000, Physical addr: ded7000
pgdir entry num:0, Pgt entry number:12, Virtual addr:c000, Physical addr: ded5000

AFTER IMPLEMENTING DEMAND PAGING:

pgdir entry num:0, Pgt entry number:0, Virtual addr:0, Physical addr: dee2000
pgdir entry num:0, Pgt entry number:12, Virtual addr:c000, Physical addr: dedf000

EXPERIMENT BY DECLARING THE LOCAL ARRAY:

In user program we created (*mypgtPrint.c*) declare a local array (*int arrLocal[10000]*).

BEFORE IMPLEMENTING DEMAND PAGING:

pgdir entry num:0, Pgt entry number:0, Virtual addr:0, Physical addr: dee2000
pgdir entry num:0, Pgt entry number:2, Virtual addr:2000, Physical addr: dedf000

AFTER IMPLEMENTING DEMAND PAGING:

pgdir entry num:0, Pgt entry number:0, Virtual addr:0, Physical addr: dee2000
pgdir entry num:0, Pgt entry number:2, Virtual addr:2000, Physical addr: dedf000

OBSERVATIONS:

- Without any arrays, it always allocates the same memory without any difference after or before implementing demand paging
- With a global array declared without any demand Paging, we see a large memory is allocated for 12-page entries.
- With a global array declared with demand Paging, we find only 2-page entries, but the second-page access is at the 12th entry position of the page table.
- With local array declared with or without any demand Paging, we find only 2-page entries.

REASONS:

- Without an array declaration, there won't be any change with or without demand Paging. But we can give a reason for why 2-page entries. The user program will have a default program size of *sz*, and the one entry in the middle is missing because it belongs to the kernel, and we are only printing user program entries.
- Now, with a global array, the program size increases and gets allocated without any demand paging, and here it reaches 12-page entries. We see only two entries with demand paging, but the second one is at the 12th position because it refers to the program being given the memory between 0 to 12 but not allocated. So, there are no valid page entries. This is because we declared the array but were not initialised.
- All local variables are stored in a stack with the local array. We can't see them in user page table entries as they are temporarily stored at the kernel level. We are getting the same number of page table entries every time, so we can't see the difference between with and without demand paging.

On repeated execution of the user program, the number of entries doesn't change, and even the virtual address does not change, but the physical address is changed. The virtual address is identical because the page directory is different for repeated processes. So page tables start from the beginning again, but the physical address is different from the previous repetition because *kalloc()* finds and gets a separate memory space.

PART 2:

DESIGN:

Here we modify the xv6 to implement a simplified demand page version.
We implemented a simplified version of demand Paging, where we assumed that the memory available was enough and neglected the swapping. We modified two files, *exec.c* and *trap.c*.

In *exec.c*, we removed dynamic data allocation as we allocate that on-demand at the initial state. We give only the memory starting from *vaddr* to *filesz (allocuvm(..., ...+ ph.filesz)*, where *vaddr* is the virtual address where the segment should be loaded. But we keep the total program size, *sz*, as fixed. That is the complete program size(*memsz*) from *vaddr*, and we keep updated. The *filesz* is the size of the program segment in the file, and *memsz* is the size of the program segment in memory (includes read-only and dynamic data). We are doing this because the *sz* is being used in the next iteration of for loop to allocate the memory as the old size. If we don't do this, we go into a situation where we think that *vaddr + filesz* is itself the old program size. So, we add the remaining *memsz - filesz* to the allocuvm() returned value and then assigning to *sz*. Due to this, we get a page fault, and the control is passed to the *trap ()* function in a *trap.c*.
So, in the *trap ()* function, we catch the page fault using the switch-case already present. We add a case *T_PDFLT*, where *T_PGFLT = 14*. 14 is the trap id for a page fault. In this case, we allocate the page to the address which caused the page fault. We first get a physical address in memory using the *kalloc()* function and store it in the mem variable. We clear the physical address to 0 using the *memset()* function. Finally, we map this new physical address to a page in the page table using a new function called *map_pgflt(...)*. This function creates a page table entry for the virtual address and maps it to the physical address. This function is like the *mappages* function in *vm.c*, which creates a page table entry to map to the virtual address. After this, we allocate the missing page, and the process resumes until another page fault occurs.

OBSERVATIONS AND REASONS:

After using the *mydemadPage* call, page faults occur after every 1024 values assigned. The page size is 4 KB, and the integer size is 4 B, which shows that we can fit 1024 integers in a one-page table entry. So, the page fault occurs when the current page table entry is filled and demands a new page table entry for the following values to be stored.

LEARNING:

- how the OS accesses the page table entry from the page directory level.
- The flow of the program memory allocation.
- The flow of the trap to catch the errors.
- How the OS boots.
- How a new physical address is created and mapped to the *PTE*