1 Overview

We will be extending DLXOS's memory management subsystem to support dynamic one-level paging and dynamic two-level paging, so there are 2 parts to this project.

DLXOS has been changed to fit the requirements for this lab. You will have to copy this new version of DLXOS (`/home/cs314/project4.tgz`) to see the changes.

1.1 Setup Instructions

As before, but unpacking project4.tgz will give you two directories this time: `project4_1` and `project4_2`. There are 2 parts to this project. Place your solution to each part in the respective directory.

Make sure you have `/home/cs314/dlxtools/bin/` in your PATH.
Compile as before: type `make` in `~/cs314/project4_1/src` or `~/cs314/project4_2/src` directory.
Run in `~/cs314/project4_1/execs` or `~/cs314/project4_2/execs` directory.

2 Assignment

DLXOS currently supports a static one-level page table, which allocates a single 64KB page (for all of the code, data and the stack segment) per process. This page is allocated when the process is first created. The total amount of physical memory present in DLXSIM is 2 MB.

2.1 Part 1: Dynamic One-Level Paging

Extend the current implementation to support dynamic one-level page tables. The system should support 256 pages of 8 KB each in total. The virtual address space of each process is limited to 512 KB. During process creation only the necessary physical pages (3 pages for text and data, and one page each for user stack and system stack) are allocated. The rest of the physical pages are dynamically allocated when a page fault occurs. Once a physical page is allocated, it is not freed. A process can use at most 16 physical pages during its execution. The kernel kills a process if it exceeds the 16 page physical memory limit and displays an informative error message.

Modify the constants `MEMORY_L1_PAGE_SIZE_BITS` and `MEMORY_L2_PAGE_SIZE_BITS` appropriately.

Your implementation should be in the `project4_1` directory. You may test your program by running the test cases given in this directory.

See the notes below for some useful details.

2.2 Part 2: Dynamic Two-Level Paging

Extend the one-level dynamic page table implementation to support a two-level dynamic page table. The virtual address space of each process is limited to 16 MB. Any virtual address referenced by a process is translated to a physical address using the two-level page table. Each L1 (Level-1) page table entry refers to a L2 (Level-2) page table and each L2 page table entry points to an 8 KB page. Each process can use at the most 1 MB of physical memory during its execution (this excludes the pages being used for L2 page tables). The kernel kills a process if it exceeds the 1MB physical memory limit and displays an informative error message.

```
-------------------------------------------------
|       L1    |     L2       | OFFSET            |
-------------------------------------------------
```

Each L1 page table entry corresponds to 512 KB of the total addressable virtual memory. There are a total of 32 entries in the L1 table (16 MB / 512 KB). The L1 part of the address is therefore 5 bits.

From the above we know that there are 32 L2 tables. Each L2 page table entry corresponds to 1 page (8KB) of the total addressable virtual memory. Hence, there are 64 entries in each L2 table (16 MB / (32 * 8 KB)) and each L2 part of the address is 6 bits.

The offset is designated by the remaining 13 bits.

In `DLX MEMORY_L1_PAGE_SIZE_BITS` is amount mapped by a single entry in the L1 table and `MEMORY_L2_PAGE_SIZE_BITS` is amount mapped by a single entry in the L2 table. Hence

```
#define MEMORY_L1_PAGE_SIZE_BITS 19 // each entry in L1 represents 512 KB
#define MEMORY_L2_PAGE_SIZE_BITS 13 // each entry in L2 is 8kb
```

If a particular entry in the L1 table does not exist, that entry should be zero. L2 PTEs point to real physical pages in memory. In addition to the physical page address, there are three special bits in each PTE: valid, dirty, and reference bits. The valid bit must be set by the operating system to indicate that the page in memory is valid. The dirty and reference bits are set by the "hardware" when an address in the page is modified and accessed (read or written), respectively. The bits are never reset by the "hardware", but they may be reset by the OS if desired.

In order to activate two-level page tables, appropriately modify the constants `MEMORY_L1_PAGE_SIZE_BITS` and `MEMORY_L2_PAGE_SIZE_BITS`.

To simulate 16MB of physical memory, you need to add the flag '-m 16777216' as shown below: dlxsim -m 16777216 -x os.dlx.obj -a -u userprog.dlx.obj.

See the notes below for some useful details.

Do your implementation in `project4_2` directory. You may test your program by running the test cases given in this directory.

2.3 Notes

- When a process is created it will need 3 pages for its code and data segment, one page for the user stack, and one page for the system stack. The user stack should always be initialized to the last page in the virtual address space of a process.
- The userprog1.c that is provided, is supposed to introduce 12 page faults. So, with initial 5 pages, this should end the program with an error(given that a process can have only 16 pages at a time). But do note that the page allocated for system stack is not counted in above mentioned math. Only the pages for code/data segments and user stack are referenced through the page table. As in the given code, you can see that the page allocated for system stack is not associated with the process's page table at all, thus making it not countable in the 16 allotted pages. So, 12 page faults would not result in an error but a 13 would as in userprog2.c.
- The highest virtual address space for a process is defined as `TOP_VIRTUAL_ADDRESS_SPACE` in memory.h. The user stack pointer initialization is dependent upon it. You can find the dependency by looking at ProcessFork().
- A `TRAP_PAGEFAULT` trap is generated when a process tries to access an invalid address. A "page fault handler" PageFaultHandler() is declared and called in the case of `TRAP_PAGEFAULT`. Implement the function. After loading the missing page and fixing the page tables, return from the page fault handler.
- A `TRAP_ACCESS` trap is generated when a process tries to access beyond its virutal address space limit. ProcessKill() is called to handle the trap in traps.c. Implement the function so that only the process is killed and the simulator does not exit.
- When a process exits, all physical pages allocated to a process need to be freed (in case of two-level paging free the pages used for L2 page tables also). ProcessKill() is called in the trap of `TRAP_EXIT` to clean the process.
- You may look at the function CPU:VaddrToPaddr in dlxsim.cc to see how virtual address to physical address translation is done.
- If the pagefault handler successfully allocates a physical page, make sure to print the page faulting address inside the handler. This should be of the following form: "Process id .., page fault address: 0x....., physical page number allocated: ....".

- When a process is killed or when a process exits, you should print all the physical page numbers being freed. This should be of the following form: "process id ..,, virtual page base address: 0x....., Freeing physical page number: .., "
- All virtual page addresses should be displayed in %x (hexadecimal) format. All physical page numbers should be displayed in %d (decimal) format.
- MemoryTranslateUserToSystem(): This procedure is used to translate user addresses into system addresses. The current implementation supports only static one-level page tables. Modify it to handle both one-level and two-level page tables.
- ProcessKill(PCB * pcb): This system call should release all the resources (i.e. all physical pages) held by the process and then remove its context. You will need to use the function ProcessFreeResources() already defined in process.c . You will need to change the implementation of ProcessFreeResources() for both parts of this project appropriately.

3 What to Turn In

Turn in 3 things as a group:

- group.txt containing information about your group.
- A .tgz of your project directory, as before. (Include a design.txt file in which you explain where you made your changes and what the changes are intended to do. Any additional information that would be useful for grading: testing, configuration, compiling, etc should go into a README file.)

Individually, you should turn in a text file containing a brief account of your group activity. Keeping in mind your group's dynamic, answer at least the following questions: What worked? What did not work? What were you responsible for in the project? What could be done differently next time?

4 FAQ

Q: Debugging is a nightmare, how can I tell what is going on?

A: Make good use of the debugging printfs, they can be enabled individually by using the tag associated with each dbprint. You can also enable all of them with '+'. Should look something like this: `dlxsim -x os.dlx.obj -a -D + -u userprog.dlx.obj`

Q: When the userprog.dlx.obj starts it displays the usage message: `Usage: [case id]` then it is followed by repetitive error lines: `Got an open with parameters (' [case id] ',0x0)`

It seems that userprog isn't recieving the arguments when I run it.

A: Don't mess with the sysStackArea. The code that allocates the page for sysStackArea should not be touched. Also, the "Got an Open" messages usually come from when dlxos tries to run a process that is done or not runnable. Have you implemented the ProcessKill function yet? That is what is called when a process exits now instead of ProcessDestroy, but it currently does not do anything. Hope one of those prompts points you in the right direction.

Q : In the assignment it says that ProcessKill() is implemented so that the simulator does not exit. Does this include the case when there are no more processes to run?

A: Call ProcessSchedule() at the end of ProcessKill(). If there is no runable process, let the system exit. Print whatever reasonable messages you think. Make sure to give yourself, and me, a clue what's going on from the message.

Q: In the slides it says that we need to modify MemoryTranslateUserToSystem(), but in the code the comments say "This works for simple one-level page tables, but will have to be modified for two-level page tables." Does this meen that we do not have to modify MemoryTranslateUserToSystem()?

A: "simple one-level page table" means it works for dlxos with a single 64kb page. You need to think about what to change when moving to multiple pages.

Q: What does MemorySetFreemap () do?

A: It's called to flag a physical page is free. You shouldn't call it directly. Instead, use MemoryFreePage().

Q: In memory.h, do we need to change the `memory_max_pages`?

A: You don't change `memory_max_pages`. That's for physical space.

Q: I am getting a page fault at location 0. This is very weird. Is this normal?

A: Most likely you didn't set table entry properly. Remember to use MemorySetupPte().

Q: What value should `TOP_VIRTUAL_ADDRESS_SPACE` be?

A: It determines the top of virtual address. Valid virtual address should be smaller than it. For part 1 it should be: `#define TOP_VIRTUAL_ADDRESS_SPACE  0x40 * MEMORY_PAGE_SIZE`
For part 2 it should be:
`#define TOP_VIRTUAL_ADDRESS_SPACE  0x800*MEMORY_PAGE_SIZE`

Q: In addition to the physical page address, there are three special bits in each PTE: valid, dirty, and reference bits. The valid bit must be set by the operating system to indicate that the page in memory is valid. The dirty and reference bits are set by the "hardware" when an address in the page is modified and accessed (read or written), respectively. The bits are never reset by the "hardware", but they may be reset by the OS if desired. Its not clear to me which bits we will be

setting/changing and which bits will be set by the system. Also, it seems to me that if the PTE will keep track of these bits, then we need to know the format of where these bits will be in the physical address (ie at the end, beginning) I imagine that the address will be shifted to the left 3 bits and this area will store the three above bits, but since the "hardware" will be using these bits as well I'd like to know where/how they're expected to be implemented.

A: The bit you need to handle is valid bit. You should set it when allocating a page, reset it after freeing the page. The mask for valid bit is `MEMORY_PTE_VALID` in memory.h

Q: Does the `MEMORY_MAX_PAGES` value need to change in lab4_2? It seems like we would since we're addressing 16MB instead of 2MB.

A: Don't worry about it. We won't use that much space.

Q: I'd like to know more about L1 and L2 in part 2.

A: In part 2, you should only store L1 table in PCB and allocate page for L2 table dynamically when needed. Here's an outline:

1. Allocate a page for L2 page table.
2. Setup L1 entry for L2 page table using MemmorySetupPte()
3. Calculate L2 page address: `L2 = L1[L1_page_no] & (MEMORY_PAGE_MASK)`
4. Allocate pages for actual text, data, or stack.
5. Use L2 page address to get pointer to L2 table and setup L2 entries for text, data, or stack pages using MemorySetupPte(). e.g. L2[3] = ...