EE469 Spring 2018 Lab 4: Memory Management

Due: Sunday, April 1, 2018 at 23:59:59

Notes

Get the examples to test paging from /home/shay/a/ee469/labs_2018 /Labs/example-paging.tar
(trace file shows the output of test2 with -D m flag)

Get the example to test heap-management from /home/shay/a/ee469 /labs_2018/Labs/heaptest-bestfit.tar

Heap management is simplified: For every process, allocate only 5-th page for the heap.

Introduction

The purpose of this lab is to become familiar with memory management techniques related to paging. In particular, you will accomplish the following:

- implement dynamic one-level paging in DLXOS,
- implement dynamic two-level paging in DLXOS,
- implement heap management on top of one-level paging in DLXOS.

The key to successfully finishing this lab is clear debugging print statements. Since you will be printing items from many different functions at the same time, a common coding standard is to prefix all your debugging messages with the name of the function they are printing from, and the processid of the current function. For instance, if you are printing the message "function started" inside the MemoryAllocPte function, your actual print statement should be: dbprintf('m', "MemoryAllocPte (%d): function started\n", GetCurrentPid());

Background and Review

Dynamic Memory Relocation

In order for multiple processes to run concurrently on shared hardware, modern operating systems work with hardware to relocate each process's address space into distinct places in memory at run-time. What this means is that when a user program runs, the addresses in the program, considered *virtual addresses*, must all be translated into physical locations whenever memory is accessed.

Note that in DLXOS, this translation is only performed on *user program instructions* (user mode), not on *operating system instructions* (kernel mode). Therefore, the hardware must be able to identify which mode it is running in so that it knows whether to do address translation or not.

Paging

Modern operating systems perform this run-time address translation via a method known as *paging*. This process is relatively simple to understand: the entire set of physical memory is broken into small chunks of equal size, called *pages*, and these chunks of memory are assigned to various processes as the processes need memory.

The virtual address space for a process is also broken into chunks (pages) of the same size. This means that the lower part of a virtual address indicates the offset within a given page (called the *page offset*), and the upper part of a virtual address indicates which page that offset corresponds to (the *page number*).

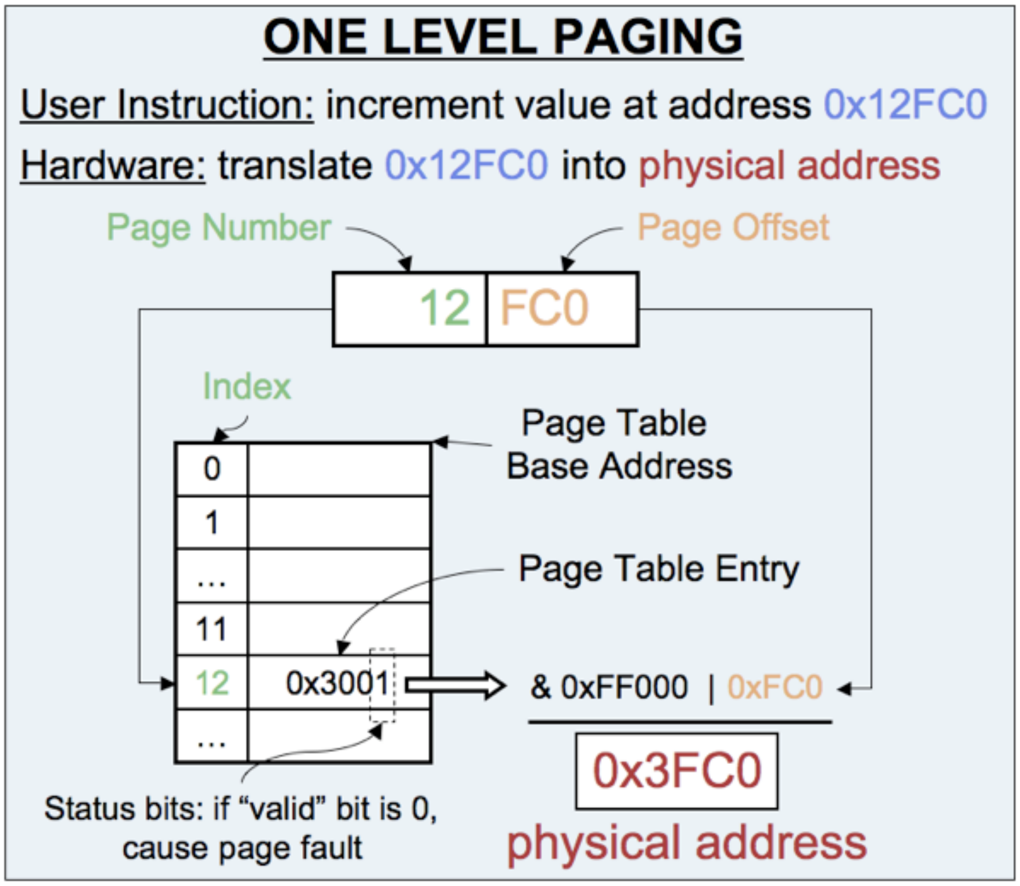
Processes maintain a *page table* in their process control block (PCB) which is simply an array of physical addresses of pages, with some status bits in the lowest bits of the page addresses. These status bits can be stored in the lowest bits of the page address only because the page addresses are always chosen to be a power of 2, thereby guaranteeing that the lowest bits of the address will always be zero. The combination of the physical page address and status bits is called a *page table entry* (PTE), because this is the value that is actually stored at each entry in the page table.

We have defined a lot of terms thus far, all of which are subtly different. Understanding the differences between these terms is critical to being able to code a dynamic memory relation implementation.

- **page:** a chunk of memory of some size.
- **virtual address:** any address used in a user program that needs to be translated to a physical address.
- **physical address:** any address used in the operating system that points to a real place in memory.
- **page table:** an array of page table entries, usually held in the PCB.
- **page table base address:** the address of entry 0 in the page table.
- **page table size:** the number of PTE's that the page table can hold.
- **page offset:** lower part of an address which indicates the offset from the base of a page. Note that there is no difference between the offset of a virtual address and the offset of a physical address.
- **physical page address:** the address of the beginning of a physical page in memory
- **virtual page address:** the address of the beginning of a chunk of virtual address space that will be mapped to a chunk of physical memory. This is found by taking the bit-wise AND of the virtual address and a mask which zero's out the offset bits.
- **page number:** the index into the page table of a virtual page. This is found by right-shifting the virtual address to get rid of the offset bits.
- **page table entry:** the value stored at each position in the page table. This is the logical "or" between the physical page address (which has zeros at the lower offset bit positions) and a set of status bits that communicate information about that page between the OS and the hardware.

One-Level Paging

The simplest form of paging is called *one-level* paging. In one-level paging, there is a single page table for each process, stored in the PCB. The procedure for translating an address in one-level paging is illustrated in the figure below.


ONE LEVEL PAGING
User Instruction: increment value at address 0x12FC0
Hardware: translate 0x12FC0 into physical address
Page Number
Page Offset
12
FC0
Index
Page Table Base Address
0
1
...
11
12
0x3001
...
Page Table Entry
& 0xFF000 | 0xFC0
0x3FC0
physical address
Status bits: if "valid" bit is 0, cause page fault

A user program runs an instruction that requires a value from memory to be read or written. This value is identified in the user-mode instruction as a virtual address. The hardware right-shifts the virtual address to get the page number. It then uses its knowledge of the page table base address, and adds the page number * sizeof(int) to the page table base address to find the PTE. It then checks the "valid" bit in the PTE to see if this PTE represents a valid physical page. If the page is not valid, a page fault exception is thrown (i.e. a hardware interrupt occurs). If the page is valid, it masks off the status bits from the PTE, or's in the offset from the virtual address, and then goes to memory to get the value at the resulting physical address.

Before any of this will work for a user process, the operating system must setup the page table. When a process is created, pages are allocated for the initial memory required for the process: this means allocating several pages starting at page number 0 (virtual address 0x0), to be used for code and global variables. Also, the initial page for the user stack must be allocated at the top of the virtual address space, i.e. the maximum page number. The user stack pointer (located at register r29) should then be initialized to the bottom of the virtual address space, since stacks grow upward. Finally, a single page for the system stack should be allocated, and its address stored in its own special register that identifies the system stack area. The system stack pointer should then be set to the bottom of the system stack page.

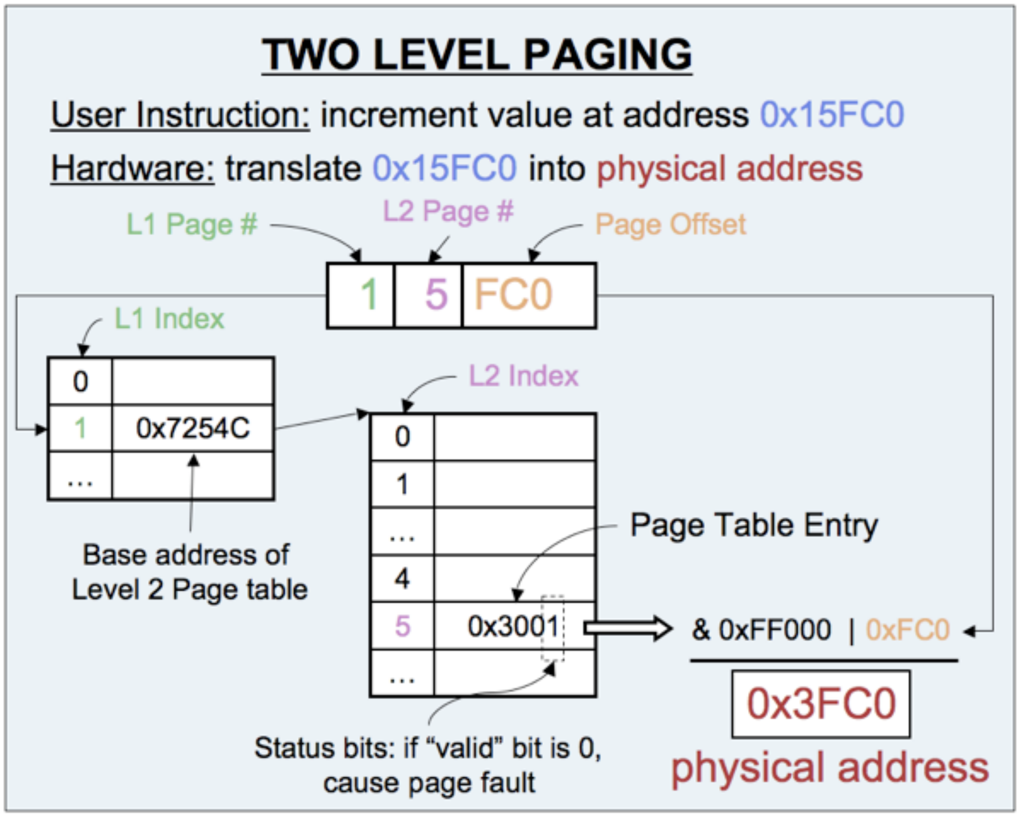
To be clear, the system stack is where registers are stored by the operating system when an interrupt occurs. In DLXOS, you can access the system stack pointer as part of the PCB. Since the system stack is only dealt with by the operating system, it always uses physical addresses. The user stack, by contrast, is accessed by both user programs and the operating system. Therefore, since user programs cannot deal with physical addresses, the user stack must use only virtual addresses. When the operating system wants to access values on the user stack, it must first translate the virtual addresses manually into physical addresses.

As the user stack grows beyond the single page allocated, it will eventually cause a page fault. When this occurs, the operating system should automatically allocate a new page. This sounds easy enough: just write a page fault handler which reads the faulting virtual address, figures out its page number, and allocates a new physical page for that page number. However, as I'm sure any Computer Engineering student can attest, sometimes when a user program tries to access memory that hasn't been allocated, it should kill the program and print a message about a SEGFAULT. So, the question becomes, how can we support legitimately allocating new pages when the user stack causes a page fault, but still kill a program that otherwise tries to access memory that it shouldn't?

The answer lies in the fact that user programs always move the stack pointer before copying items onto the stack. Therefore, the page fault handler can simply compare the address that caused the page fault with the user stack pointer. If the fault address is greater than or equal to the stack pointer, then it was the user stack that caused the fault and a new page should be allocated. Otherwise, the current process should be killed due to a segmentation fault.

Two-Level Paging

The problem with one-level paging is that as the virtual address space becomes large, the size of the page table becomes large as well. This is troublesome since the one-level page table has to be preallocated in a contiguous chunk of the physical memory. To help alleviate this problem, another level of page table is added. Another level of **Indirection**! The process of converting a virtual address to a physical address for two-level page tables is illustrated in the figure below.


TWO LEVEL PAGING
User Instruction: increment value at address 0x15FC0
Hardware: translate 0x15FC0 into physical address
L1 Page #
L2 Page #
Page Offset
1
5
FC0
L1 Index
0
1
0x7254C
...
Base address of Level 2 Page table
L2 Index
0
1
...
4
5
0x3001
...
Page Table Entry
& 0xFF000 | 0xFC0
0x3FC0
physical address
Status bits: if "valid" bit is 0, cause page fault

The first level page table, contained in the PCB, now contains the physical addresses of level 2 page tables, rather than PTEs. Each level 2 page table contains the PTEs for the pages. This setup means that the virtual address is partitioned into 3 parts:

- the level 1 page number, which locates the address of a level 2 page table in the level 1 page table,
- the level 2 page number, which locates the PTE for a page in the level 2 page table,
- and the page offset, which is the same as the page offset in the one-level paging scheme.

DLXOS Hardware Simulator

As mentioned earlier, paging requires hardware support. This means that the hardware has to know the following information:

- page table base address
- page table size
- which bits in the virtual address represent the offset field of the address
- which bits in the virtual address represent the index into the page table (the *page number*)
- which bits in the PTE are status bits

The DLX hardware simulator uses CPU registers to store this information. When you are setting up a new process, you will need to set the various registers appropriately in order for addresses to be translated properly. However, you will not be putting values directly into registers. This is because when your new process gets switched in by ProcessSchedule the first time, it will load all the CPU registers with values from the system stack, as you learned in lab 3. Therefore, if you put values directly into CPU registers when a process is created, they will simply be overwritten. Therefore, the register values must instead be set in the saved set of register values on the system stack.

Recall that you will need to allocate a physical page for the system stack, and store that value in the PCB, in the "sysStackArea" field. Then, the actual system stack pointer, stored in the PCB as "sysStackPtr", must be set to the bottom of that page since the system stack grows up from the bottom. There is one caveat for this, however: addresses in DLXOS must be 4-byte aligned, a common practice in almost all 32-bit hardware. If you set the system stack pointer to the bottom address of the page, it will not be 4-byte aligned. For instance, if your page size is 4KB, and your system stack page starts at address 0x3000, then the maximum address in that page is 0x3FFF. This is not 4-byte aligned. You must subtract 3 to make it 4-byte aligned, or simply mask off the lower 2 bits. Therefore, the actual maximum address in the page is 0x3FFC.

Once you have allocated the system stack area and setup the system stack pointer, you can decrement sysStackPtr by PROCESS_STACK_FRAME_SIZE to make it appear that a full set of registers have been saved on the system stack. Note that PROCESS_STACK_FRAME_SIZE is in units of 4-byte words, so be sure to use pointer arithmetic correctly. You can then set the "currentSavedFrame" field of the PCB to the same thing as the sysStackPtr. Once you have currentSavedFrame set properly, you can use it like an array to set all the register values you need, which will then be loaded into the CPU by the interrupt handler when the process gets switched in.

The indices from currentSavedFrame that you need to set are (#define-d in process.h):

- PROCESS_STACK_PTBASE: base address of the level 1 page table
- PROCESS_STACK_PTSIZE: maximum number of entries in the level 1 page table
- PROCESS_STACK_PTBITS: this register contains two different pieces of information, one in the upper 16 bits, and one in the lower 16 bits. The number in the lower 16 bits indicates the bit position of the least significant bit of the level 1 page number field in a virtual address. The number in the upper 16 bits indicates the bit position of the least significant bit of the level 2 page table field in a virtual address. As a special case, if the two numbers are the same, then the hardware assumes that there are no level 2 page tables and uses only one-level address translation.
- PROCESS_STACK_USER_STACKPOINTER: this is the initial stack pointer for the user process. This should be set to the highest 4-byte-aligned address in the virtual memory space.

Recall that the hardware also has to agree with the operating system as to which bits in a PTE are status bits. The DLX hardware simulator supports 3 status bits:

- MEMORY_PTE_READONLY: 0x4. If this bit is set in a PTE, it means that the page should be marked readonly. We will ONLY set this bit if we use fork. But we are not implementing fork in this lab, so be sure to set it to zero.
- MEMORY_PTE_DIRTY: 0x2. This is set by the hardware when a page has been modified. This is only used for caching, and we will not deal with this bit in this lab.
- MEMORY_PTE_VALID: 0x1. If this bit is set to 1, then the PTE is considered to contain a valid physical page address.

It is important to note that a physical page is only marked as read-only or valid in a PTE, which resides in a page table. It is entirely possible for one process to mark a page as read-only in its page table, and another process could try to use the same physical page as read-write in its page table. This will become important when dealing with the system stack, because its page does not reside in a page table, so it therefore has no PTE to store readonly or valid bits.

Also, when a page fault occurs, the hardware must tell the operating system which virtual page caused the fault. In the DLX hardware simulator, it puts this address into a register, which is pushed onto the system stack when the page fault interrupt occurs. You can access this address by accessing the "PROCESS_STACK_FAULT" index of the currentSavedFrame pointer.

The Freemap

In order for the OS to allocate and deallocate physical pages, it must keep track of which pages are in use and which are not. To accomplish this, many modern operating systems use the concept of a "freemap" which is just an array of bits in memory, where each bit corresponds to 1 physical page. The freemap is typically implemented as an array of integers, where each integer represents 32 pages. To find an available physical page, you simply need to loop through the freemap integers until you find one entry that is not zero, and then start looking at each bit one-at-a-time until you find a bit that is a one. You can then compute the physical page number by multiplying the freemap index by 32 and adding the bit position within that 32-bit number.

One very important point about the freemap is how it is initialized when the OS starts up. Since the OS takes up some physical memory, and the freemap has one bit for every page-sized chunk of physical memory, then the bits in the freemap where the OS resides need to be marked as "in use". To accomplish this, recognize that the OS is loaded at the top of physical memory, i.e. starting at address 0x0. To find the last address of the OS, DLXOS has a nifty assembly trick that defines a global variable at the end of the operating system (in os/osend.s). The value of this variable is written at compile time by the compiler once the size of the entire OS has been computed by the linker. You can simply declare in memory.h an external global variable named "lastosaddress", and then read that value to find the end of the operating system. So at the outset, mark all the pages from 0x0 to lastosaddress as inuse in the freemap, and all other pages as not inuse.

Heap Management

The heap is generally used for user land memory management in a given process. A portion of a process' address space is pre-allocated as the process heap. The heap always grows from a lower address to a higher address (different from the stack). When a process performs a `malloc()` call, the O/S scans through all free space in the heap and attempts to find an area at least as large as the requested size. Freeing any allocated space in the heap is the responsibility of the process. i.e., an explicit call to `mfree()` must be made.

A typical malloc implementation works by requesting a consecutive chunk of addresses in the virtual address space, and returning the starting virtual address. Under paging, this boils down to set of consecutive virtual pages. To keep track of which areas in the virtual address space are free at any given time, a malloc implementation must maintain metadata about the size and location of each allocated address block in use and any free address block between used blocks. As the program requires more memory, the malloc implementation requests more virtual memory pages, increasing the application's "memory footprint". The main design challenge is to reduce external fragmentation, while keeping the Allocation/Deallocation algrorithms reasonably fast.

There are many heuristic algorithms to perform memory allocation and its associated garbage collection (best fit, first fit, and so on). No single technique is optimal for all programs.

On UNIX, malloc() is implemented as a standard C library function in libc. Since there is no notion of libc, in DLXOS, for this part of the lab, we will implement the malloc and free functionality in the kernel, i.e., as system calls. In addition, we assume the heap size is one page. In other words, we will implement heap management in DLXOS by pre-allocating one page for the heap during process creation. This page should be in addition to the four pages already allocated at process creation. In summary, we will implement the following system calls:

- `void *malloc(int memsize)` - allocate a memory block of size `memsize` in the heap space and return a pointer corresponding to the starting virtual address of the allocated block. Blocks *must* be allocated in multiples of 4 bytes. This call should round the size up if it is not already a multiple of four. Eg, if `memsize` is 7 bytes, the allocated heap block will have a size of 8 bytes.

  In the event of a failure, no memory allocation should be performed and the call should return NULL. Failure occurs when: `memsize` is less than or equal to zero; `memsize` is greater than the total heap size; there is no free heap block large enough to accommodate `memsize` bytes.

  If the call is successful, you should output a string in the following format:

  ```
  Created a heap block of size <memsize> bytes: virtual address <vaddress>, physical address <paddress>.
  ```
- `int mfree(void *ptr)` - free the heap block identified by `ptr`. The caller does not specify the size of the heap block to be freed. HINT: `malloc()` must keep track of the size of each allocated heap block! After freeing the block, `mfree` should determine whether there are any free block(s) adjacent to this newly-freed heap block. If such block(s) exist, the newly-freed heap block should be merged with adjacent free block(s) to create a larger free block.

  If the call is successful, the number of bytes freed should be returned. If a failure occurs, -1 should instead be returned. A failure occurs when `ptr` is NULL, or `ptr` does not belong to the heap space. Additionally, when `mfree()` succeeds it should display the following:

  ```
  Freeing heap block of size <memsize> bytes: virtual address <vaddress>, physical address <paddress>.
  ```

Additional Notes

- The limit of 32 physical pages per process includes the page allocated for the heap.
- When a process exits or is killed, all physical pages belonging to the process should be freed. This includes the page allocated for the heap.
- You will need to keep track of the location and size of free blocks in the heap. The free blocks can be maintained as a linked list.

Bitwise tricks

There are several bitwise operations that can make your life easier in this lab. These tips assume that you have #define-d the following constants:

- MEM_L1FIELD_FIRST_BITNUM: bit position of the least significant bit of the level 1 page number field in a virtual address.
- MEM_L2FIELD_FIRST_BITNUM: bit position of the least significant bit of the level 2 page number field in a virtual address.
- MEM_MAX_VIRTUAL_ADDRESS: the maximum allowable address in the virtual address space. Note that this is not the 4-byte-aligned address, but rather the actual maximum address (it should end with 0xF).
- MEM_PTE_READONLY: 0x4
- MEM_PTE_DIRTY: 0x2
- MEM_PTE_VALID: 0x1

With those constants in mind, here are a few useful items that you can compute. Note that most of these operations only work because we're assuming that the page size is an integer power of 2.

- Finding the size of a page (MEM_PAGESIZE): `0x1 << MEM_L2FIELD_FIRST_BITNUM`. Note that this is the actual size of the page: to get the maximum possible offset within a page you would need to subtract 1.
- Finding the level 1 pagetable size: `(MEM_MAX_VIRTUAL_ADDRESS + 1) >> MEM_L1FIELD_FIRST_BITNUM`. Note that this is the size of the level 1 page table: to get the maximum allowable index into the level 1 page table, you would need to subtract 1.
- Finding the level 2 pagetable size: `(0x1 << (MEM_L1FIELD_FIRST_BITNUM - MEM_L2FIELD_FIRST_BITNUM)`. Not that this is the size of the level 2 page table: to get the maximum allowable index into the level 2 page table, you would need to subtract 1.
- Finding the page offset mask: `(MEM_PAGESIZE - 1)`
- Finding a mask to convert from a PTE to a page address: `~(MEM_PTE_READONLY | MEM_PTE_DIRTY | MEM_PTE_VALID)`

Changes to DLXOS Source

Since you will be modifying the operating system in three different ways, we have created 3 directories for you this week: one-level/, two-level/, and heap-mgmt/. The only directory with code in it when you start is one-level/. Once you are done getting your one-level page tables working, copy that code to the two-level/ and heap-mgmt directories

Since you will be working very closely with the DLX simulator hardware this week, we have included one of the main source files from the simulator for you to browse. The "VaddrToPaddr" function in simulator_source/dlxsim.cc will show you how the hardware does the page-based address translation.

Up to this point, you have only turned on debugging messages in your DLXOS implementation, not in the simulator itself. This week, you're going to need the simulator to tell you information about its address translation process. To do that, you simply need to pass the "-D m" flag to the simulator. Be careful to observe that passing "-D m" to the simulator is not the same as passing "-D m" to your operating system. A typical call to dlxsim will now look like this:

```
$ dlxsim -D m -x os.dlx.obj -a -D m -u makeprocs.dlx.obj
```

The first "-D m" tells the hardware simulator to turn on debugging messages labeled with 'm', and the second "-D m" tells the OS to turn on debugging messages labeled with 'm'.

We have cleaned out almost all the previous code in memory.c. The code that is left you will need to update as you implement one-level and two-level paging. The rest of the code was removed because it is more confusing than helpful. You are still welcome to look back at previous labs' memory.c if you really want to see what was there before.

We have also removed all the code in process.c that related to memory management. You can search through process.c for the word "STUDENT" to find most of the places that you need to write code. The primary location is in ProcessFork, where it creates a new process. Note that the OS will not compile until you have fixed ProcessFork with new code for your paging implementation.

Since memory.c needs to use PCB's as arguments in some of it's functions, it must #include process.h at the top. However, process.h needs to use the #define-d page table size from memory.h in order to know how big the PCB should be, so it needs to #include memory.h at the top. This recursive structure doesn't work in C, so we split memory.h into two files: memory_constants.h which contains all #define's dealing with memory, and memory.h which holds everything else for the memory header file. This way, process.h now #include's memory_constants.h at the top instead of memory.h.

When creating your freemap and reference counters, you will need to know at compile time how large your arrays should be. This size depends on the number of physical pages in the system, which is simply computed as `num_pages = size_of_memory / size_of_one_page`. You can read the size of memory at runtime from a special register at DLX_MEMSIZE_ADDRESS. The problem with this computation is that the size of memory is not known at compile time (you can run the simulator with various memory sizes using command-line flags). Therefore, you will need to set a maximum allowed memory size at compile time in order to create your freemap and reference counters, and then only use the values that pertain to the size of memory at runtime.

We have not included all the traps for you automatically this week. You will need to implement the trap handlers for any traps that you find missing.

Assignment

Download and untar /home/shay/a/ee469/labs_2018/Labs/lab4.tar. Untar using tar -xf. This will create a `lab4/` directory for you. Put all your work in this directory structure.

1. (30 points) **Implement dynamic one-level paging in DLXOS**. All your code should go in the one-level/ directory. You will need to write code in process.c, memory.c, process.h, memory.h, and memory_constants.h. Your implementation must abide by the following specifications:
    - Use a page size of 4KB (note that 4KB is not exactly 4,000 bytes).
    - Use a virtual memory size of 4MB (note that 4MB is not exactly 4,000,000 bytes).
    - Use a maximum physical memory size of 2MB (again, 2MB is not exactly 2,000,000 bytes).
    - You can only hard-code values for MEM_L1FIELD_FIRST_BITNUM, MEM_L2FIELD_FIRST_BITNUM, MAX_VIRTUAL_ADDRESS, MEM_PTE_READONLY, MEM_PTE_DIRTY, and MEM_PTE_VALID. All other memory-related constants must be computed from these 6 values in your code.

  - When creating a process, initially allocate 4 pages for code and global data, 1 page for the user stack, and one page for the system stack. Assume that the system stack will never grow larger than 1 page.
  - You will need to implement a page fault handler which allocates a new page if the system stack causes a page fault, and kills a process for any other page faults.
  - DO NOT EXIT THE SIMULATOR IF SOMETHING GOES WRONG. You should kill the process that caused the problem with ProcessKill(), and then continue running the OS.

2. (10 points) **Write user test program for the following scenarios**:
   1. Print "Hello World" and exit.
   2. Access memory inside the virtual address space, but outside of currently allocated pages.
   3. Cause the user function call stack to grow larger than 1 page.
   4. Call the "Hello World" program 100 times to make sure you are rightly allocating and freeing pages.
   5. Spawn 30 simultaneous processes that print a message, count to a large number in a for loop, and then print another message before exiting. You should choose a number large enough for counting that all 30 processes end up running at the same time. You should not run out of memory with 30 processes.
   6. Access memory beyond the maximum virtual address.

   These tests should all be written in the one-level/apps/example/ directory. They should all be controlled by the makeprocs process which will run them all in succession. You must print informative messages, either from the OS or from your user programs, that will clearly indicate to the grader that you have performed the tests and they succeeded.

3. (30 points) **Implement dynamic two-level paging in DLXOS.** Copy your one-level/ directory into the two-level/ directory, and modify it to use two-level paging instead of one-level paging. This means that you will need to have memory available somewhere in which to allocate and free the level 2 page tables as they are needed. For simplicity in this lab, you can statically allocate a global array of 256 level 2 page tables to use as they are needed. You can use the same testing code from the one-level paging to test your two-level paging implementation. You should use the same specifications as you did for one-level paging, except that you should use 2 bits for the level 1 page table field, and 8 bits for the level 2 page table field.

4. (30 points) **Implement heap management in DLXOS.** Copy your one-level/ directory into heap-mgmt/ once your one-level page table is working. Implement the malloc and mfree functions in memory.c

   The user program can malloc a chunk of memory from the heap. The user can free the block after using. Your job is to implement the heap allocation/free functionalities, i.e., to search for the best-fit block of sufficient size, and to update the free list when memory is freed.

Turning in your solution

You should make sure that your `lab4/` directory contains all of the files needed to build your project source code. You should include a README file that explains:

- how to build your solution,

- anything unusual about your solution that the TA should know,
- and **a list of any external sources referenced while working on your solution**.

DO NOT INCLUDE OBJECT FILES in your submission! In other words, make sure you run "make clean" in all of your application directories, and in the os directory. Every file in the turnin directory that could be generated automatically by the DLX compiler or assembler will result in a 5-point deduction from your over all project grade for this lab.

When you are ready to submit your solution, change to the directory containing the `lab4` directory and execute the following command:

```
turnin -c ee469 -p lab4-Y lab4
```

where `Y` stands for your lab session. Wednesday, 2:30pm = 1, Friday, 11:30am = 2, Friday, 2:30pm = 3.