ECE 469 Spring 2019 Laboratory #4: Memory Management

Due Tuesday, April 9, 2019 at 23:59 EST

Changelog

3/29/20: Minor revisions for clarification. Formatted as ~~deletion~~ and addition .

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 UNIX-style fork() with copy-on-write,
- 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 id of the current process. 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, 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),

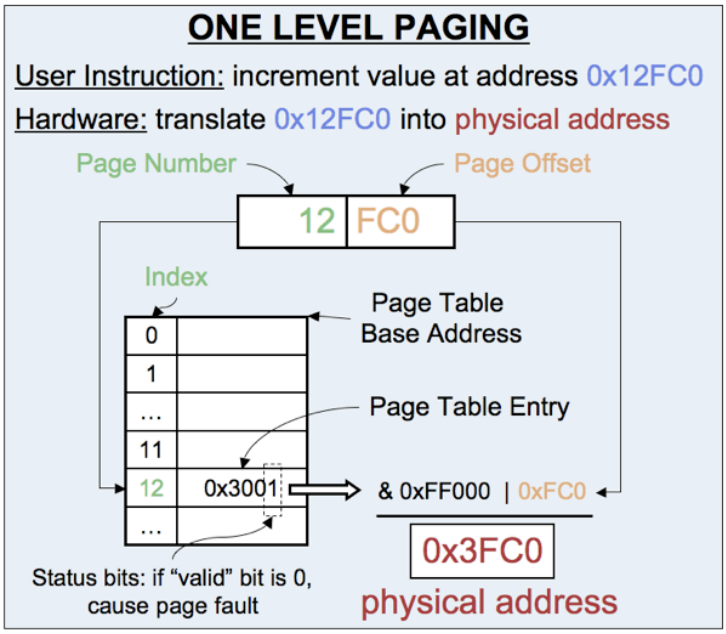
because this is the value that is actually stored at each entry in the page table.

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

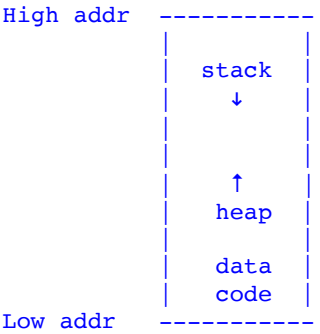
The simplest form of paging is called *one-level* paging. In one-level paging, there is a single page table for each process, stored in the PCB. The procedure for translating an address in one-level paging is illustrated in the figure below.



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~~ towards the highest address 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~~ the highest address (i.e. 0xFFFFC, 4-byte aligned) of the virtual address space, since stacks grow ~~upward~~ from higher addresses towards lower addresses.



Finally, a single page for the system stack should be allocated, and its address stored in its own special register that identifes the system stack area. The system stack pointer should then be set to the ~~bottom~~ highest virtual address (4-byte aligned, see above) 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. Recall that the stack pointer grows from higher addresses to lower addresses. 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.

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~~ highest address of that page since the system stack grows ~~up from the bottom~~ from higher to lower addresses . 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 _intrhandler 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. In this lab, we will not deal with level 2 page tables
- 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 when implementing fork later, so be sure to set it to zero for now.
- 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 in which 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 implementated 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. 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. In this lab we will be implementing Buddy Memory Allocation technique which is explained below (copied from Wikipedia):

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 six pages already allocated at process creation (one for system stack, one for user stack, and four for user code and global data).** For simplicity, in this lab, we assume that the physical page for the heap is preallocated. In reality, a heap is backed up by multiple physical pages and they are allocated dynamically, on demand, just like those for the stack. In summary, we will implement the following system calls:

- `void *malloc(int memsize)` - allocate a memory block of size `memsize` in the heap space using the buddy memory allocation technique 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 check whether the buddy to this newly-freed heap block is also free and combine them recursively.

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.

Buddy Memory Allocation

The buddy memory allocation technique is a memory allocation algorithm that divides memory into partitions to try to satisfy a memory request as suitably as possible. This system makes use of splitting memory into halves to try to give a best-fit. According to Donald Knuth, the buddy system was invented in 1963 by Harry Markowitz, who won the 1990 Nobel Memorial Prize in Economics, and was first described by Kenneth C. Knowlton (published 1965). Buddy memory allocation is relatively easy to implement. It supports limited but efficient splitting and coalescing of memory blocks.

How it works

There are various forms of the buddy system, but binary buddies, in which each block is subdivided into two smaller blocks, are the simplest and most common variety. Every memory block in this system has an order, where the order is an integer ranging from 0 to a specified upper limit. The size of a block of order n is proportional to 2n, so that the blocks are exactly twice the size of blocks that are one order lower. Power-of-two block sizes make address computation simple, because all buddies are aligned on memory address boundaries that are powers of two. When a larger block is split, it is divided into two smaller blocks, and each smaller block becomes a unique buddy to the other. A split block can only be merged with its unique buddy block, which then reforms the larger block they were split from.

Starting off, the size of the smallest possible block is determined, i.e. the smallest memory block that can be allocated. If no lower limit existed at all (e.g., bit-sized allocations were possible), there would be a lot of memory and computational overhead for the system to keep track of which parts of the memory are allocated and unallocated. However, a rather low limit may be desirable, so that the average memory waste per allocation (concerning allocations that are, in size, not multiples of the smallest block) is minimized. Typically the lower limit would be small enough to minimize the average wasted space per allocation, but large enough to avoid excessive overhead. The smallest block size is then taken as the size of an order-0 block, so that all higher orders are expressed as power-of-two multiples of this size.

The programmer then has to decide on, or to write code to obtain, the highest possible order that can fit in the remaining available memory space. Since the total available memory in a given computer system may not be a power-of-two multiple of the minimum block size, the largest block size may not span the entire memory of the system. For instance, if the system had 2000K of physical memory and the order-0 block size was 4K, the upper limit on the order would be 8, since an order-8 block (256 order-0 blocks, 1024K) is the biggest block that will fit in memory. Consequently it is impossible to allocate the entire physical memory in a single chunk; the remaining 976K of memory would have to be allocated in smaller blocks.

In practice

The following is an example of what happens when a program makes requests for memory. Let's say in this system, the smallest possible block is 64 kilobytes in size, and the upper limit for the order is 4, which results in a largest possible allocatable block, 24 times 64K = 1024K in size. The following shows a possible state of the system after various memory requests

Step	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K
1	2^4															
2.1	2^3								2^3							
2.2	2^2				2^2				2^3							
2.3	2^1		2^1		2^2				2^3							
2.4	2^0	2^0	2^1		2^2				2^3							
2.5	A: 2^0	2^0	2^1		2^2				2^3							
3	A: 2^0	2^0	B: 2^1		2^2				2^3							
4	A: 2^0	C: 2^0	B: 2^1		2^2				2^3							
5.1	A: 2^0	C: 2^0	B: 2^1		2^1		2^1		2^3							
5.2	A: 2^0	C: 2^0	B: 2^1		D: 2^1		2^1		2^3							
6	A: 2^0	C: 2^0	2^1		D: 2^1		2^1		2^3							
7.1	A: 2^0	C: 2^0	2^1		2^1		2^1		2^3							
7.2	A: 2^0	C: 2^0	2^1		2^2				2^3							
8	2^0	C: 2^0	2^1		2^2				2^3							
9.1	2^0	2^0	2^1		2^2				2^3							
9.2	2^1		2^1		2^2				2^3							
9.3	2^2				2^2				2^3							
9.4	2^3								2^3							
9.5	2^4															

This allocation could have occurred in the following manner

1. The initial situation.
2. Program A requests memory 34K, order 0.
    1. No order 0 blocks are available, so an order 4 block is split, creating two order 3 blocks.
    2. Still no order 0 blocks available, so the first order 3 block is split, creating two order 2 blocks.
    3. Still no order 0 blocks available, so the first order 2 block is split, creating two order 1 blocks.
    4. Still no order 0 blocks available, so the first order 1 block is split, creating two order 0 blocks.
    5. Now an order 0 block is available, so it is allocated to A.
3. Program B requests memory 66K, order 1. An order 1 block is available, so it is allocated to B.
4. Program C requests memory 35K, order 0. An order 0 block is available, so it is allocated to C.
5. Program D requests memory 67K, order 1.
    1. No order 1 blocks are available, so an order 2 block is split, creating two order 1 blocks.
    2. Now an order 1 block is available, so it is allocated to D.
6. Program B releases its memory, freeing one order 1 block.
7. Program D releases its memory.
    1. One order 1 block is freed.
    2. Since the buddy block of the newly freed block is also free, the two are merged into one order 2 block.
8. Program A releases its memory, freeing one order 0 block.
9. Program C releases its memory.
    1. One order 0 block is freed.
    2. Since the buddy block of the newly freed block is also free, the two are merged into one order 1 block.
    3. Since the buddy block of the newly formed order 1 block is also free, the two are merged into one order 2 block.
    4. Since the buddy block of the newly formed order 2 block is also free, the two are merged into one order 3 block.
    5. Since the buddy block of the newly formed order 3 block is also free, the two are merged into one order 4 block.

As you can see, what happens when a memory request is made is as follows:

- If memory is to be allocated

1. Look for a memory slot of a suitable size (the minimal 2^k block that is larger or equal to that of the requested memory)
    1. If it is found, it is allocated to the program
    2. If not, it tries to make a suitable memory slot. The system does so by trying the following:
        1. Split a free memory slot larger than the requested memory size into half

2. If the lower limit is reached, then allocate that amount of memory
3. Go back to step 1 (look for a memory slot of a suitable size)
4. Repeat this process until a suitable memory slot is found

- If memory is to be freed

1. Free the block of memory
2. Look at the neighboring block - is it free too?
3. If it is, combine the two, and go back to step 2 and repeat this process until either the upper limit is reached (all memory is freed), or until a non-free neighbour block is encountered

Implementation and efficiency

In comparison to other simpler techniques such as dynamic allocation, the buddy memory system has little external fragmentation, and allows for compaction of memory with little overhead. The buddy method of freeing memory is fast, with the maximal number of compactions required equal to log2(highest order). Typically the buddy memory allocation system is implemented with the use of a binary tree to represent used or unused split memory blocks. The "buddy" of each block can be found with an exclusive OR of the block's address and the block's size.

However, there still exists the problem of internal fragmentation — memory wasted because the memory requested is a little larger than a small block, but a lot smaller than a large block. Because of the way the buddy memory allocation technique works, a program that requests 66K of memory would be allocated 128K, which results in a waste of 62K of memory. This problem can be solved by slab allocation, which may be layered on top of the more coarse buddy allocator to provide more fine-grained allocation.

One version of the buddy allocation algorithm was described in detail by Donald Knuth in volume 1 of The Art of Computer Programming. The Linux kernel also uses the buddy system, with further modifications to minimise external fragmentation, along with various other allocators to manage the memory within blocks.

Fork and Exec

The fork() system call in UNIX creates a new process that is an exact copy of the calling process. In practice, fork() is often used with another system call exec() to provide a means of spawning a new process that runs a different program.

The prototype of fork() looks like this:

```
int fork();
```

When a process calls fork(), a duplicate process, referred to as the "child" process, is created. The process that called fork() is referred to as the "parent" process. The parent process continues executing its program right after the point that fork() was called. The child process, too, executes the same program starting from the same place. The only difference between the two processes is that they have different process IDs. The child process is a new process and therefore has a new process ID, distinct from its parent's process ID.

The return value of fork() is set to 0 in the child process, and it is set to the processid of the new child in the parent process. This manifests itself in code like this:

```
int child_pid;

printf ("the main program process ID is %d\n", (int) getpid ());
child_pid = fork();
if (child_pid != 0) {
    printf ("this is the parent process, with id %d\n", (int) getpid ());
    printf ("the child's process ID is %d\n", child_pid);
} else {
    printf ("this is the child process, with id %d\n", (int) getpid ());
}
```

When a process calls exec(), an existing process is replaced with a different program. The primary feature of exec() is that it does not allocate any new resources (memory pages, PCB, etc.): it uses the existing resources of the process, and free's up any existing resources that aren't needed by the new process. The prototype for exec looks like this:

```
void exec(char *filename_to_run, ...);
```

where the "..." is the set of command-line arguments for the new process.

When using fork and exec together to spawn a new process, the code generally looks like this:

```
if (fork() != 0) {
  printf("Hi, I'm the parent.\n");
} else {
  exec("new_program_execuatable.dlx.obj", arg1, arg2);
  printf("This line will never run\n");
}
```

Copy-on-write

At this point, implementing fork seems simple enough: just grab a new PCB, copy the parent's PCB into the new PCB byte-by-byte, and then go through the page table and system stack, allocating new pages for the child process, and copy all the data from the parent pages to the child pages byte-by-byte.

Obviously, this "brute-force" approach to cloning a process is very inefficient, especially if the child process calls exec() soon after fork(), which will just replace all the existing data in its pages anyway. Also, even if the child never calls exec(), the code sections of the parent and child will never change, so it was really a waste of time and resources to copy them.

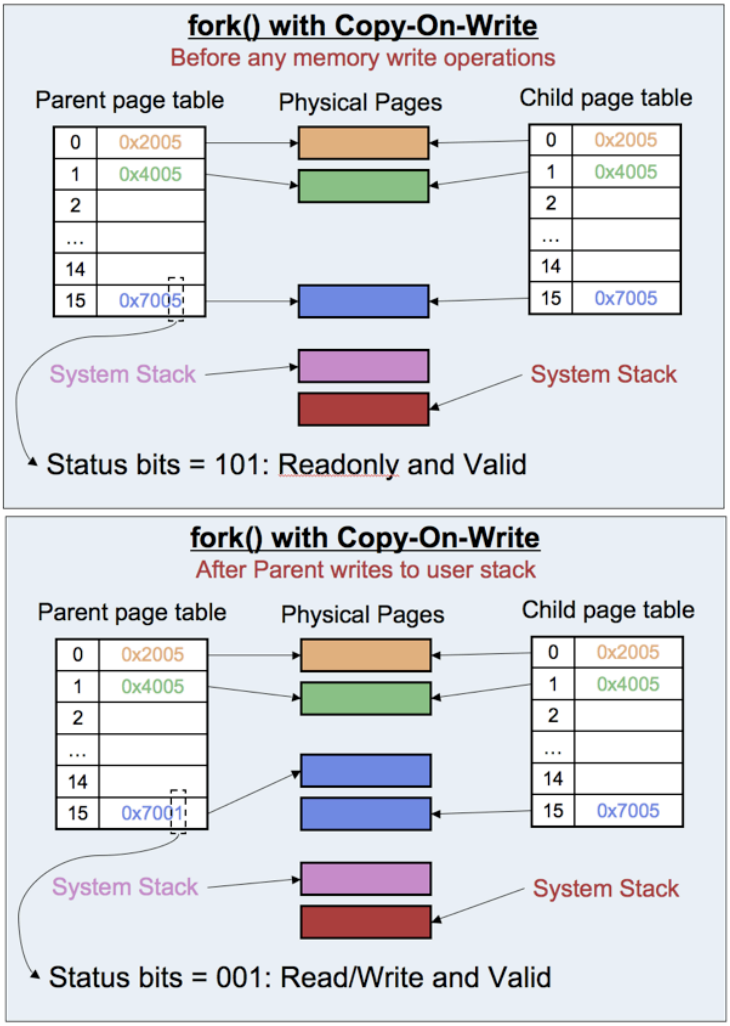
To alleviate these problems, fork() is implemented with the "copy-on-write" technique. In this technique, when a process forks, no new pages are allocated for the child. Instead, the child's page table points to the same physical pages as the parent's page table. All the valid PTE's in the parent and the child are marked as readonly by setting the MEMORY_PTE_READONLY bit. When either the parent or child tries to write to one of the shared pages, the hardware will throw a TRAP_ROP_ACCESS exception (trap number 0x8 in DLXOS). The page which caused the exception will be stored in the PROCESS_STACK_FAULT register in the currentSavedFrame of the PCB.

The operating system now has a decision to make inside its ROP_ACCESS handler.

- If there is more than one process using this page, then the page should be copied byte-by-byte to a new page, and the corresponding PTE of the current process should be replaced with the new PTE, with the new PTE marked as read/write (i.e. the readonly bit is cleared).
- If there is exactly one process using this page, it should be simply marked as read/write. No copying is necessary.

This of course raises the question, how is the operating system supposed to know how many processes are using a particular physical page? The brute-force method for determining this number is to loop through all valid PCB's, and then loop through all of their valid PTE's, and see which ones have this page address marked as valid in any PTE. This is horribly inefficient. A much better way is for the operating system to keep a global array of reference counters that are associated with each physical page. Whenever a physical page is put into any process's page table, the counter should be incremented. Whenever a physical page is freed from a process's page table, the counter should be decremented. Once the counter for a particular page reaches 0, it should be marked as free in the global freemap.

The fork with copy-on-write process is illustrated in the figures below.



Copying the System Stack

Another confusing component of implementing fork() is the copying of the system stack. The system stack page is not in a page table, so it has no PTE that can be marked as readonly. Therefore, the copy-on-write technique will not work for the system stack and it must be copied byte-by-byte manually into the child process when fork() is called. This does not actually introduce any inefficiencies, since the return value for a trap is stored in the system stack for a process. Since fork() is supposed to return different values to each process, then the "copy-on-write" technique would have required the system stack to be copied anyway.

It is simple enough to just allocate a new page for the system stack of the child process, and copy the system stack page from the parent process onto the child's new page. There is a problem, however, since all system stack addresses are physical addresses rather than virtual addresses. If we relocate the system stack to a new page, any addresses that were supposed to point back into the system stack have to be manually fixed to point to the new system stack.

For example, consider the initial scenario where a process calls fork, and its system stack page starts at 0x4000 with its sysStackPointer set to 0x4EA8, which of course points into the system stack page. Now we copy the PCB for the parent onto a new PCB for the child, which will set the sysStackPtr for the child to 0x4EA8. We then allocate a new page for the child's system stack at physical address 0x7000 and copy all the bytes from the page at 0x4000 (the parent's system stack page) to the page at 0x7000 (the child's system stack page). When the call to fork() returns, the system stack pointer in the child's PCB still points at the parent's system stack! Therefore, all the CPU registers will be loaded with the values from the parent's stack instead of the child's stack.

To fix this problem, recognize that any addresses that point into the system stack can be broken down as: address =

(page_base_address + offset), in the same way that one-level paging has a page base address and an offset. Therefore, when copying the system stack, as long as you know which addresses to fix that point into the system stack, you can fix them by finding the offset portion of those addresses in the parent, and adding it to the new page base address in the child.

The items that need to be fixed for this lab in DLXOS are:

- the sysStackPtr field in the child's PCB
- the currentSavedFrame field in the child's PCB
- the PROCESS_STACK_PTBASE field within the current saved frame (this is not computed by finding an offset, but rather is simply set to the base address of the child's level 1 page table).

Note that if the system stack contains more than one stack frame, you would also need to iterate through the other stack frames and fix these values as well. In this lab, since the priority of the fork() trap is relatively high, it is unlikely that the system stack will have any other frames on it, therefore you can ignore this step in your implementation.

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_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_L1FIELD_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 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/, fork/, 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 fork/ 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 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 funtions, 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 ~ee469/labs_2019/Labs/lab4.tar.gz. 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 1024KB (note that 1024KB is not exactly 1,024,000 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, MAX_VIRTUAL_ADDRESS, MEM_MAX_SIZE, MEM_PTE_READONLY, MEM_PTE_DIRTY, and MEM_PTE_VALID. All other memory-related constants must be computed from those 6 in code. MEM_MAX_SIZE is the maximum physical memory size.
    - 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 user 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. (5 points) **Write user test program for the following scenarios**:
    1. Print "Hello World" and exit.

2. Access memory beyond the maximum virtual address.
3. Access memory inside the virtual address space, but outside of currently allocated pages.
4. Cause the user function call stack to grow larger than 1 page.
5. Call the "Hello World" program 100 times to make sure you are rightly allocating and freeing pages.
6. 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.

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 fork() with one-level page tables using copy-on-write.** Copy your one-level/ directory into fork/ once your one-level page tables are working. When implementing fork(), note that there is an existing function in process.c called "ProcessFork". This function actually is where new processes are created, rather than where you should be implementing fork(). You should create a new function, ProcessRealFork(), that implements the actual call to fork().

   When forking a new process with level 1 (L1) page tables, you will need to explictly allocate new L1 page tables for the child process and copy all the data from the parent's L1 tables into the child's L1 tables. Of course, you will need to duplicate the PCB first. You can use *bcopy((char *)currentPCB, (char *)childpcb, sizeof(PCB));* Use ProcessSetResult() to return values. Don't forget to put child PCB onto the end of the run queue.

   You must implement reference counters for physical pages in this question.

   You must implement a handler for the trap number 0x8 for readonly page access violations.

   As mentioned earlier, do not forget to fix the various system stack pointers and values on the system stack when creating the new system stack page.

4. (5 points) **Write test code to verify that fork() with copy-on-write works as planned.** Some things that you must include are:
   - after fork(), print the valid page table entries for both the parent and the child.
   - after fork(), generate a TRAP_ROP_ACCESS exception and print the valid page table entries for both the parent and the child. Recall that when either the parent or child tries to write to one of the shared pages, the hardware will throw a TRAP_ROP_ACCESS exception (trap number 0x8 in DLXOS).

5. (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 (using the buddy memory allocation technique), and free and merge the buddy blocks when memory is freed. Note that the heap size is 4KB (one page of 4KB). The smallest possible block (order 0) that can be allocated by your buddy memory allocation implementation is 32 bytes (hint: there can be 128 such blocks at max).

   Your OS should print the following messages (please use printf and note that your numbers will be different; order means the order of the node/block, addr is the address of the block inside heap (relative to heap), size is the block size in bytes).

   After allocating a block (should be one message per malloc() call):

   ```
   Allocated the block: order = 1, addr = 320, requested mem size = 40, block size = 64
   ```

   After creating a child node: (there can be multiple such messages per malloc() call, please specify if it is the left child or right)

   ```
   Created a right child node (order = 2, addr = 384, size = 128) of parent (order = 3, addr = 256, size = 256)
   ```

```
Created a left child node (order = 1, addr = 256, size = 64) of parent (order = 2, addr = 256, size = 128)
```

After freeing a block (should be one message per mfree() call):

```
Freed the block: order = 1, addr = 256, size = 64
```

After coalescing buddy nodes: (there can be multiple such messages per mfree() call, use two print statements to print the two lines)

```
Coalesced buddy nodes (order = 2, addr = 256, size = 128) & (order = 2, addr = 384, size = 128)
into the parent node (order = 3, addr = 256, size = 256)
```

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,
- **NOTE: please mention the files that you have modified for each of the questions**,
- 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, so it will be lab4-1 or lab4-2 or lab4-3 depending on which lab session you are in. Wednesday, 2:30pm = 1, Friday, 11:30am = 2, Friday, 2:30pm = 3.

---

ee469@ecn.purdue.edu