

Lecture Notes for CS347: Operating Systems

Mythili Vutukuru, Department of Computer Science and Engineering, IIT Bombay

8. Memory Management in xv6

8.1 Basics

- xv6 uses 32-bit virtual addresses, resulting in a virtual address space of 4GB. xv6 uses paging to manage its memory allocations. However, xv6 does not do demand paging, so there is no concept of virtual memory.
- xv6 uses a page size of 4KB, and a two level page table structure. The CPU register CR3 contains a pointer to the page table of the current running process. The translation from virtual to physical addresses is performed by the MMU as follows. The first 10 bits of a 32-bit virtual address are used to index into a page table directory, which points to a page of the inner page table. The next 10 bits index into the inner page table to locate the page table entry (PTE). The PTE contains a 20-bit physical frame number and flags. Every page table in xv6 has mappings for user pages as well as kernel pages. The part of the page table dealing with kernel pages is the same across all processes.
- In the virtual address space of every process, the kernel code and data begin from KERNBASE (2GB in the code), and can go up to a size of PHYSTOP (whose maximum value can be 2GB). This virtual address space of [KERNBASE, KERNBASE+PHYSTOP] is mapped to [0,PHYSTOP] in physical memory. The kernel is mapped into the address space of every process, and the kernel has a mapping for all usable physical memory as well, restricting xv6 to using no more than 2GB of physical memory. Sheets 02 and 18 describe the memory layout of xv6.

8.2 Initializing the memory subsystem

- The xv6 bootloader loads the kernel code in low physical memory (starting at 1MB, after leaving the first 1MB for use by I/O devices), and starts executing the kernel at `entry` (line 1040). Initially, there are no page tables or MMU, so virtual addresses must be the same as physical addresses. So the kernel entry code resides in the lower part of the virtual address space, and the CPU generates memory references in the low virtual address space only. The entry code first turns on support for large pages (4MB), and sets up the first page table `entryptpgdir` (lines 1311-1315). The second entry in this page table is easier to follow: it maps [KERNBASE, KERNBASE+4MB] to [0, 4MB], to enable the first 4MB of kernel code in the high virtual address space to run after MMU is turned on. The first entry of this page table maps virtual addresses [0, 4MB] to physical addresses [0, 4MB], to enable the entry code that resides in the low virtual address space to run. Once a pointer to this page table is stored in CR3, MMU is turned on, the entry code creates a stack, and jumps to the `main` function in the kernel's C code (line 1217). The C code is located in high virtual address space, and can run because of the second entry in `entryptpgdir`. So why was the first page table entry required? To enable the few instructions between turning on MMU and jumping to high address space to run correctly. If the entry page table only mapped high virtual addresses, the code that jumps to high virtual addresses of `main` would itself not run (because it is in low virtual address space).
- Remember that once the MMU is turned on, for any memory to be usable, the kernel needs a virtual address and a page table entry to refer to that memory location. When `main` starts, it is still using `entryptpgdir` which only has page table mappings for the first 4MB of kernel addresses, so only this 4MB is usable. If the kernel wants to use more than this 4MB, it needs to map all of that memory as free pages into its address space, for which it needs a larger page table. So, `main` first creates some free pages in this 4MB in the function `kinit1` (line 3030), which eventually calls the functions `freerange` (line 3051) and `kfree` (line 3065). Both these functions together populate a list of free pages for the kernel to start using for various things, including allocating a bigger, nicer page table for itself that addresses the full memory.
- The kernel uses the `struct run` (line 3014) data structure to address a free page. This structure simply stores a pointer to the next free page, and the rest of the page is filled with garbage. That is, the list of free pages are maintained as a linked list, with the pointer to the next page being stored within the page itself. Pages are added to this list upon initialization, or on freeing them up. Note that the kernel code that allocates and frees pages always returns the virtual address of the page in the kernel address space, and not the actual physical address. The `V2P` macro is used when one needs the physical address of the page, say to put into the page table entry.
- After creating a small list of free pages in the 4MB space, the kernel proceeds to build a bigger page table to map all its address space in the function `kvmalloc` (line 1857). This function in turn calls `setupkvm` (line 1837) to setup the kernel page table, and switches to it. The address space mappings that are setup by `setupkvm` can be found in the structure `kmap` (lines 1823-1833). `kmap` contains the mappings for all kernel code, data, and any free memory that the kernel wishes to use, all the way from KERNBASE to KERNBASE+PHYSTOP. Note that

the kernel code and data is already residing at the specified physical addresses, but the kernel cannot access it because all of that physical memory has not been mapped into any logical pages or page table entries yet.

- The function `setupkvm` works as follows. For each of the virtual to physical address mappings in `kmap`, it calls `mappages` (line 1779). The function `mappages` walks over the entire virtual address space in 4KB page-sized chunks, and for each such logical page, it locates the PTE using the `walkpgdir` function (line 1754). `walkpgdir` simply outputs the translation that the MMU would do. It uses the first 10 bits to index into the page table directory to find the inner page table. If the inner page table does not exist, it requests the kernel for a free page, and initializes the inner page table. Note that the kernel has a small pool of free pages setup by `kinit1` in the first 4MB address space—these free pages are used to construct the kernel’s page table. Once `walkpgdir` returns the PTE, `mappages` sets up the appropriate mapping using the physical address it has. (Sheet 08 has the various macros that are useful in getting the index into page tables from the virtual address.)
- After the kernel page table `kpgdir` is setup this way (line 1859), the kernel switches to this page table by storing its address in the CR3 register in `switchkvm` (line 1866). From this point onwards, the kernel can freely address and use its entire address space from KERNBASE to KERNBASE+PHYSTOP.
- Let’s return back to `main` in line 1220. The kernel now proceeds to do various initializations. It also gathers many more free pages into its free page list using `kinit2`, given that it can now address and access a larger piece of memory. At this point, the entire physical memory at the disposal of the kernel [0, PHYSTOP] is mapped by `kpgdir` into the virtual address space [KERNBASE, KERNBASE+PHYSTOP], so all memory can be addressed by virtual addresses in the kernel address space and used for the operation of the system. This memory consists of the kernel code/data that is currently executing on the CPU, and a whole lot of free pages in the kernel’s free page list. Now, the kernel is all set to start user processes, starting with the `init` process.

8.3 Creating user processes

- The function `userinit` (line 2502) creates the first user process. We will examine the memory management in this function. The kernel page table of this process is created using `setupkvm` as always. For the user part of the memory, the function `inituvvm` (line 1903) allocates one physical page of memory, copies the init executable into that memory, and sets up a page table entry for the first page of the user virtual address space. When the init process runs, it executes the init executable (sheet 83), whose main function forks a shell and starts listening to the user. Thus, in the case of init, setting up the user part of the memory was straightforward, as the executable fit into one page.
- All other user processes are created by the `fork` system call. In `fork` (line 2554), once a child process is allocated, its memory image is setup as a complete copy of the parent's memory image by a call to `copyuvvm` (line 2053). This function walks through the entire address space of the parent in page-sized chunks, gets the physical address of every page of the parent using a call to `walkpgdir`, allocates a new physical page for the child using `kalloc`, copies the contents of the parent's page into the child's page, adds an entry to the child's page table using `mappages`, and returns the child's page table. At this point, the entire memory of the parent has been cloned for the child, and the child's new page table points to its newly allocated physical memory.
- If the child wants to execute a different executable from the parent, it calls `exec` right after `fork`. For example, the init process forks a child and execs the shell in the child. The `exec` system call copies the binary of an executable from the disk to memory and sets up the user part of the address space and its page tables. In fact, after the initial kernel is loaded into memory from disk, all subsequent executables are read from disk into memory via the `exec` system call alone.
- `Exec` (line 6310) first reads the ELF header of the executable from the disk and checks that it is well formed. It then initializes a page table, and sets up the kernel mappings in the new page table via a call to `setupkvm` (line 6334). Then, it proceeds to build the user part of the memory image via calls to `allocuvvm` (line 6346) and `loaduvvm` (line 6348) for each segment of the binary executable. `allocuvvm` (line 1953) allocates physical pages from the kernel's free pool via calls to `kalloc`, and sets up page table entries. `loaduvvm` (line 1918) reads the memory executable from disk into the allotted page using the `readi` function. After the end of the loop of calling these two functions for each segment, the program executable has been loaded into memory, and page table entries setup to point to it. However, `exec` hasn't switched to this new page table yet, so it is still executing in the old memory image.
- Next, `exec` goes on to build the rest of its new memory image. For example, it allocates a user stack page, and an extra page as a guard after the stack. The guard page has no physical memory frame allocated to it, so any access beyond the stack into the guard page will cause a page fault. Then, the arguments to `exec` are pushed onto the user stack, so that the `exec` binary can access them when it starts.

- It is important to note that `exec` does not replace/reallocate the kernel stack. The `exec` system call only replaces the user part of the memory. And if you think about it, there is no way the process can replace the kernel stack, because the process is executing in kernel mode on the kernel stack itself, and has important information like the trap frame stored on it. However, it does make small changes to the trap frame on the kernel stack, as described below.
- Now, a process that makes the `exec` system call has moved into kernel mode to service the software interrupt of the system call. Normally, when the process moves back to user mode again (by popping the trap frame on the kernel stack), it is expected to return to the instruction after the system call. However, in the case of `exec`, the process doesn't have to return to the instruction after `exec` when it gets the CPU next, but instead must start executing in the new memory image containing the binary file it just loaded from disk. So, the code in `exec` changes the return address in the trap frame to point to the entry address of the binary (line 6392). Finally, once all these operations succeed, `exec` switches page tables to start using the new memory image, and frees up all the memory pointed at by the old page table. At this point, the process that called `exec` can start executing on the new memory image. Note that `exec` waits until the end to do this switch, because if anything went wrong in the system call, `exec` returns to the old process image and prints out an error.

Practice Problems: File systems

1. Provide one reason why a DMA-enabled device driver usually gives better performance over a non-DMA interrupt-driven device driver.

Ans: A DMA driver frees up CPU cycles that would have been spent copying data from the device to physical memory.

2. Which of the following statements is/are true regarding memory-mapped I/O?

- A. The CPU accesses the device memory much like it accesses main memory.
- B. The CPU uses separate architecture-specific instructions to access memory in the device.
- C. Memory-mapped I/O cannot be used with a polling-based device driver.
- D. Memory-mapped I/O can be used only with an interrupt-driven device driver.

Ans: A

3. Consider a file D1/F1 that is hard linked from another parent directory D2. Then the directory entry of this file (including the filename and inode number) in directory D1 must be exactly identical to the directory entry in directory D2. [T/F]

Ans: F (the file name can be different)

4. It is possible for a system that uses a disk buffer cache with FIFO as the buffer replacement policy to suffer from the Belady's anomaly. [T/F]

Ans: T

5. Reading files via memory mapping them avoids an extra copy of file data from kernel space buffers to user space buffers. [T/F]

Ans: T

6. A soft link can create a link between files across different file systems, whereas a hard link can only create links between a directory and a file within the same file system. [T/F]

Ans: T (because hard link stores inode number, which is unique only within a file system)

7. Consider the process of opening a new file that does not exist (obviously, creating it during opening), via the “open” system call. Describe changes to all the in-memory and disk-based file system structures (e.g., file tables, inodes, and directories) that occur as part of this system call implementation. Write clearly, listing the structure that is changed, and the change made to it.

Ans: (a) New inode allocated on disk (with link count=1), and inode bitmap updated in the process. (b) Directory entry added to parent directory, to add mapping from file name to inode number.

- (c) In-memory inode allocated. (d) System-wide open file table points to in-memory inode. (e) Per-process file descriptor table points to open file table entry.
8. Now, suppose the process that has opened the file in the previous question proceeds to write 100 bytes into the file. Assume block size on disk is 512 bytes. Assume the OS uses a write-through disk buffer cache. List all the operations/changes to various datastructures that take place when the write operation successfully completes.
- Ans:** (a) open file table offset is changed (b) in-memory and on-disk inode adds pointer to new data block, and last modified time is updated (c) a copy of the data block comes into the disk buffer cache (d) New data block is allocated from data block bitmap (e) new data block is filled with user provided data
9. Repeat the above question for the implementation of the “link” system call, when linking to an existing file (not open from any process) in a directory from another new parent directory.
- Ans:** (a) The link count of the on-disk inode of the file is incremented. (b) A directory entry is added to the new directory to create a mapping from the file name to the inode number of the original file (if the new directory does not have space in its data blocks for the new file, a new data block is allocated for the new directory entry, and a pointer to this data block is added from the directory’s inode).
10. Repeat the above question for the implementation of the ”dup” system call on a file descriptor.
- Ans:** To dup a file descriptor, another empty slot in the file descriptor table of the process is found, and this new entry is set to point to the same global open file table entry as the old file descriptor. That is, two FDs point to same system-wide file table entry.
11. Consider a file system with 512-byte blocks. Assume an inode of a file holds pointers to N direct data blocks, and a pointer to a single indirect block. Further, assume that the single indirect block can hold pointers to M other data blocks. What is the maximum file size that can be supported by such an inode design?
- Ans:** $(N+M)*512$ bytes
12. Consider a FAT file system where disk is divided into M byte blocks, and every FAT entry can store an N bit block number. What is the maximum size of a disk partition that can be managed by such a FAT design?
- Ans:** $2^N * M$ bytes
13. Consider a secondary storage system of size 2 TB, with 512-byte sized blocks. Assume that the filesystem uses a multilevel inode datastructure to track data blocks of a file. The inode has 64 bytes of space available to store pointers to data blocks, including a single indirect block, a double indirect block, and several direct blocks. What is the maximum file size that can be stored in such a file system?
- Ans:** Number of data blocks = $2^{41}/2^9 = 2^{32}$, so 32 bits or 4 bytes are required to store the number of a data block.

Number of data block pointers in the inode = $64/4 = 16$, of which 14 are direct blocks. The single indirect block stores pointers to $512/4 = 128$ data blocks. The double indirect block points to 128 single indirect blocks, which in turn point to 128 data blocks each.

So, the total number of data blocks in a file can be $14 + 128 + 128 \times 128 = 16526$, and the maximum file size is 16526×512 bytes.

14. Consider a filesystem managing a disk with block size 2^b bytes, and disk block addresses of 2^a bytes. The inode of a file contains n direct blocks, one single indirect block, one double indirect block, and one triple indirect block. What is the maximum size of a file (in bytes) that can be stored in this filesystem? Assume that the indirect blocks only store a sequence of disk addresses, and no other metadata.

Ans: Let $x = \text{number of disk addresses per block} = 2^{b-a}$. Then max file size is $2^b * (n + x + x^2 + x^3)$.

15. The `fork` system call creates new entries in the open file table for the newly created child process. [T/F]

Ans: F

16. When a process opens a file that is already being read by another process, the file descriptors in both processes will point to the same open file table entry. [T/F]

Ans: F

17. Memory mapping a file using the `mmap` system call adds one or more entries to the page table of the process. [T/F]

Ans: T

18. The `read` system call to fetch data from a file always blocks the invoking process. [T/F]

Ans: F (the data may be readily available in the disk buffer cache)

19. During filesystem operations, if the filesystem implementation ensures that changes to data blocks of a file are flushed to disk before changes to metadata blocks (like inodes and bitmaps), then the filesystem will never be in an inconsistent state after a crash, and a filesystem checker need not be run to detect and fix any inconsistencies. [T/F]

Ans: F (If there are multiple metadata operations, some may have happened and some may have been lost, causing an inconsistency. For example, a bitmap may indicate a data block is allocated but no inode points to it.)

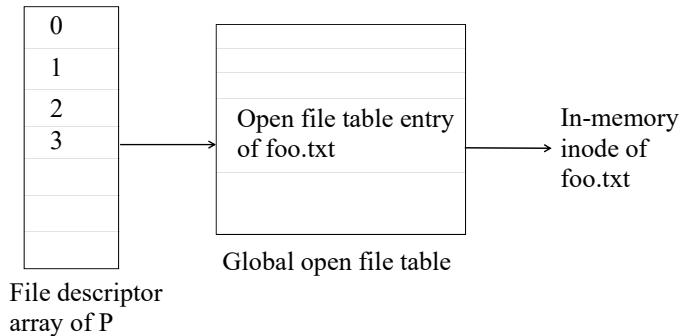
20. Interrupt-based device drivers give superior performance to polling-based drivers because they eliminate the time spent by the CPU in copying data to and from the device hardware. [T/F]

Ans: F

21. When a process writes a block to the disk via a disk buffer cache using the write-back policy, the process invoking the write will block until the write is committed to disk. [T/F]

Ans: F

22. Consider a process P that has opened a file `foo.txt` using the `open` system call. The figure below shows the file descriptor array of P and the global open file table, and the pointers linking these data structures.



- After opening the file, P forks a child C. Draw a figure showing the file descriptor arrays of P and C, and the global open file table, immediately after the fork system call successfully completes. It is enough to show the entries pertaining to the file `foo.txt`, as in the figure above.
- Repeat part (a) for the following scenario: after P forks a child C, another process Q also opens the same file `foo.txt`.

Ans: In (a), the file descriptor arrays of P and C are pointing to the same file table entry. In (b), the file descriptor array of Q is pointing to a new open file table entry, which points to the same inode of the file `foo.txt`.

Practice Problems: Concurrency

1. Answer yes/no, and provide a brief explanation.
 - (a) Is it necessary for threads in a process to have separate stacks?
 - (b) Is it necessary for threads in a process to have separate copies of the program executable?
2. Can one have concurrent execution of threads/processes without having parallelism? If yes, describe how. If not, explain why not.

Ans:

- (a) Yes, so that they can have separate execution state, and run independently.
(b) No, threads share the program executable and data.

2. Can one have concurrent execution of threads/processes without having parallelism? If yes, describe how. If not, explain why not.

Ans:

Yes, by time-sharing the CPU between threads on a single core.

3. Consider a multithreaded webserver running on a machine with N parallel CPU cores. The server has M worker threads. Every incoming request is put in a request queue, and served by one of the free worker threads. The server is fully saturated and has a certain throughput at saturation. Under which circumstances will increasing M lead to an increase in the saturation throughput of the server?

Ans: When $M < N$ and the workload to the server is CPU-bound.

4. Consider a process that uses a user level threading library to spawn 10 user level threads. The library maps these 10 threads on to 2 kernel threads. The process is executing on a 8-core system. What is the maximum number of threads of a process that can be executing in parallel?

Ans: 2

5. Consider a user level threading library that multiplexes $N > 1$ user level threads over $M \geq 1$ kernel threads. The library manages the concurrent scheduling of the multiple user threads that map to the same kernel thread internally, and the programmer using the library has no visibility or control on this scheduling or on the mapping between user threads and kernel threads. The N user level threads all access and update a shared data structure. When (or, under what conditions) should the user level threads use mutexes to guarantee the consistency of the shared data structure?

- (a) Only if $M > 1$.
- (b) Only if $N \geq M$.
- (c) Only if the M kernel threads can run in parallel on a multi-core machine.

- (d) User level threads should always use mutexes to protect shared data.

Ans: (d) (because user level threads can execute concurrently even on a single core)

6. Which of the following statements is/are true regarding user-level threads and kernel threads?

- (a) Every user level thread always maps to a separate schedulable entity at the kernel.
- (b) Multiple user level threads can be multiplexed on the same kernel thread
- (c) Pthreads library is used to create kernel threads that are scheduled independently.
- (d) Pthreads library only creates user threads that cannot be scheduled independently at the kernel scheduler.

Ans: (b), (c)

7. Consider a Linux application with two threads T1 and T2 that both share and access a common variable x . Thread T1 uses a pthread mutex lock to protect its access to x . Now, if thread T2 tries to write to x without locking, then the Linux kernel generates a trap. [T/F]

Ans: F

8. In a single processor system, the kernel can simply disable interrupts to safely access kernel data structures, and does not need to use any spin locks. [T/F]

Ans: T

9. In the pthread condition variable API, a process calling wait on the condition variable must do so with a mutex held. State one problem that would occur if the API were to allow calls to wait without requiring a mutex to be held.

Ans: Wakeup happening between checking for condition and sleeping causing missed wakeup.

10. Consider N threads in a process that share a global variable in the program. If one thread makes a change to the variable, is this change visible to other threads? (Yes/No)

Ans: Yes

11. Consider N threads in a process. If one thread passes certain arguments to a function in the program, are these arguments visible to the other threads? (Yes/No)

Ans: No

12. Consider a user program thread that has locked a pthread mutex lock (that blocks when waiting for lock to be released) in user space. In modern operating systems, can this thread be context switched out or interrupted while holding the lock? (Yes/No)

Ans: Yes

13. Repeat the previous question when the thread holds a pthread spinlock in user space.

Ans: Yes

14. Consider a process that has switched to kernel mode and has acquired a spinlock to modify a kernel data structure. In modern operating systems, will this process be interrupted by external hardware before it releases the spinlock? (Yes/No)

Ans: No

15. Consider a process that has switched to kernel mode and has acquired a spinlock to modify a kernel data structure. In modern operating systems, will this process initiate a disk read before it releases the spinlock? (Yes/No)

Ans: No

16. When a user space process executes the wakeup/signal system call on a pthread condition variable, does it always lead to an immediate context switch of the process that calls signal (immediately after the signal instruction)? (Yes/No)

Ans: No

17. Consider a process in kernel mode that acquires a spinlock. For correct operation, it must disable interrupts on its CPU core for the duration that the spinlock is held, in both single core and multicore systems. [T/F]

Ans. T

18. Consider a process in kernel mode that acquires a spinlock in a multicore system. For correct operation, we must ensure that no other kernel-mode process running in parallel on another core will request the same spinlock. [T/F]

Ans. F

19. Multiple threads of a program must use locks when accessing shared variables even when executing on a single core system. [T/F]

Ans: T

20. Recall that the atomic instruction compare-and-swap (CAS) works as follows:

CAS(&var, oldval, newval) writes newval into var and returns true if the old value of var is oldval. If the old value of var is not oldval, CAS returns false and does not change the value of the variable. Write code for the function to acquire a simple spinlock using the CAS instruction.

Ans: while(!CAS(&lock, 0, 1));

21. The simple spinlock implementation studied in class does not guarantee any kind of fairness or FIFO order amongst the threads contending for the spin lock. A ticket lock is a spinlock implementation that guarantees a FIFO order of lock acquisition amongst the threads contending for the lock. Shown below is the code for the function to acquire a ticket lock. In this function, the variables next_ticket and now_serving are both global variables, shared across all threads, and initialized to 0. The variable my_ticket is a variable that is local to a particular thread, and is not shared across threads. The atomic instruction fetch_and_increment(&var) atomically adds 1 to the value of the variable and returns the old value of the variable.

```
acquire() :  
    my_ticket = fetch_and_increment(&next_ticket)  
    while(now_serving != my_ticket); //busy wait
```

You are now required to write the code to release the spinlock, to be executed by the thread holding the lock. Your implementation of the release function must guarantee that the next contending

thread (in FIFO order) will be able to acquire the lock correctly. You must not declare or use any other variables.

```
release(): //your code here
```

Ans:

```
release(): //your code here  
now_serving++;
```

22. Consider a multithreaded program, where threads need to acquire and hold multiple locks at a time. To avoid deadlocks, all threads are mandated to use the function `acquire_locks`, instead of acquiring locks independently. This function takes as arguments a variable sized array of pointers to locks (i.e., addresses of the lock structure), and the number of lock pointers in the array, as shown in the function prototype below. The function returns once all locks have been successfully acquired.

```
void acquire_locks(struct lock *la[], int n);  
//i-th lock in array can be locked by calling lock(la[i])
```

Describe (in English, or in pseudocode) one way in which you would implement this function, while ensuring that no deadlocks happen during lock acquisition. Your solution must not use any other locks beyond those provided as input. Note that multiple threads can invoke this function concurrently, possibly with an overlapping set of locks, and the lock pointers can be stored in the array in any arbitrary order. You may assume that the locks in the array are unique, and there are no duplicates within the input array of locks.

Ans. Sort locks by address `struct lock *`, and acquire in sorted order.

23. Consider a process where multiple threads share a common Last-In-First-Out data structure. The data structure is a linked list of "struct node" elements, and a pointer "top" to the top element of the list is shared among all threads. To push an element onto the list, a thread dynamically allocates memory for the struct node on the heap, and pushes a pointer to this struct node in to the data structure as follows.

```
void push(struct node *n) {  
    n->next = top;  
    top = n;  
}
```

A thread that wishes to pop an element from the data structure runs the following code.

```
struct node *pop(void) {  
    struct node *result = top;  
    if(result != NULL) top = result->next;  
    return result;  
}
```

A programmer who wrote this code did not add any kind of locking when multiple threads concurrently access this data structure. As a result, when multiple threads try to push elements onto this structure concurrently, race conditions can occur and the results are not always what one would expect. Suppose two threads T1 and T2 try to push two nodes n1 and n2 respectively onto the data structure at the same time. If all went well, we would expect the top two elements of the data structure would be n1 and n2 in some order. However, this correct result is not guaranteed when a race condition occurs.

Describe how a race condition can occur when two threads simultaneously push two elements onto this data structure. Describe the exact interleaving of executions of T1 and T2 that causes the race condition, and illustrate with figures how the data structure would look like at various phases during the interleaved execution.

Ans: One possible race condition is as follows. n1's next is set to top, then n2's next is set to top. So both n1 and n2 are pointing to the old top. Then top is set to n1 by T1, and then top is set to n2 by T2. So, finally, top points to n2, and n2's next points to old top. But now, n1 is not accessible by traversing the list from top, and n1 remains on a side branch of the list.

24. Consider the following scenario. A town has a very popular restaurant. The restaurant can hold N diners. The number of people in the town who wish to eat at the restaurant, and are waiting outside its doors, is much larger than N. The restaurant runs its service in the following manner. Whenever it is ready for service, it opens its front door and waits for diners to come in. Once N diners enter, it closes its front door and proceeds to serve these diners. Once service finishes, the backdoor is opened and the diners are let out through the backdoor. Once all diners have exited, another batch of N diners is admitted again through the front door. This process continues indefinitely. The restaurant does not mind if the same diner is part of multiple batches.

We model the diners and the restaurant as threads in a multithreaded program. The threads must be synchronized as follows. A diner cannot enter until the restaurant has opened its front door to let people in. The restaurant cannot start service until N diners have come in. The diners cannot exit until the back door is open. The restaurant cannot close the backdoor and prepare for the next batch until all the diners of the previous batch have left.

Below is given unsynchronized pseudocode for the diner and restaurant threads. Your task is to complete the code such that the threads work as desired. Please write down the complete synchronized code of each thread in your solution.

You are given the following variables (semaphores and initial values, integers) to use in your solution. The names of the variables must give you a clue about their possible usage. You must not use any other variable in your solution.

```
sem (init to 0): entering_diners, exiting_diners, enter_done, exit_done  
sem (init to 1): mutex_enter, mutex_exit  
Integer counters (init to 0): count_enter, count_exit
```

All changes to the counters and other variables must be done by you in your solution. None of the actions performed by the unsynchronized code below will modify any of the variables above.

- (a) Unsynchronized code for the restaurant thread is given below. Add suitable synchronization in your solution in between these actions of the restaurant.

```
openFrontDoor()  
closeFrontDoor()  
serveFood()  
openBackDoor()  
closeBackDoor()
```

- (b) Unsynchronized code for the diner thread is given below. Add suitable synchronization in your solution around these actions of the diner.

```
enterRestaurant()  
eat()  
exitRestaurant()
```

Ans: Correct code for restautant thread:

```
openFrontDoor()
do N times: up(entering_diners)
down(enter_done)

closeFrontDoor()
serveFood()

openBackDoor()
do N times: up(exiting_diners)
down(exit_done)
closeBackDoor()
```

Correct code for the diner thread:

```
down(entering_diners)
enterRestaurant()

down(mutex_enter)
count_enter++
if(count_enter == N) {
    up(enter_done)
    count_enter = 0
}
up(mutex_enter)

eat()

down(exiting_diners)
exitRestaurant()

down(mutex_exit)
count_exit++
if(count_exit == N) {
    up(exit_done)
    count_exit = 0
}
up(mutex_exit)
```

An alternate to doing up N times in restaurant thread is: restaurant does up once, and every woken up diner does up once until N diners are done. This alternate solution is shown below. Correct code for restautant thread:

```
openFrontDoor()  
up(entering_diners)  
down(enter_done)  
  
closeFrontDoor()  
serveFood()  
  
openBackDoor()  
up(exiting_diners)  
down(exit_done)  
closeBackDoor()
```

Correct code for the diner thread:

```
down(entering_diners)  
enterRestaurant()  
  
down(mutex_enter)  
count_enter++  
  
if(count_enter < N)  
    up(entering_diners)  
else if(count_enter == N) {  
    up(enter_done)  
    count_enter = 0  
}  
up(mutex_enter)  
  
eat()  
  
down(exiting_diners)  
exitRestaurant()  
  
down(mutex_exit)  
count_exit++  
if(count_exit < N)  
    up(exiting_diners)  
else if(count_exit == N) {  
    up(exit_done)  
    count_exit = 0  
}  
up(mutex_exit)
```

25. Consider a scenario where a bus picks up waiting passengers from a bus stop periodically. The bus has a capacity of K . The bus arrives at the bus stop, allows up to K waiting passengers (fewer if less than K are waiting) to board, and then departs. Passengers have to wait for the bus to arrive and then board it. Passengers who arrive at the bus stop after the bus has arrived should not be allowed to board, and should wait for the next time the bus arrives. The bus and passengers are represented by threads in a program. The passenger thread should call the function `board()` after the passenger has boarded and the bus should invoke `depart()` when it has boarded the desired number of passengers and is ready to depart.

The threads share the following variables, none of which are implicitly updated by functions like `board()` or `depart()`.

```
mutex = semaphore initialized to 1.
bus_arrived = semaphore initialized to 0.
passenger_boarded = semaphore initialized to 0.
waiting_count = integer initialized to 0.
```

Below is given synchronized code for the passenger thread. You should not modify this in any way.

```
down(mutex)
waiting_count++
up(mutex)
down(bus_arrived)
board()
up(passenger_boarded)
```

Write down the corresponding synchronized code for the bus thread that achieves the correct behavior specified above. The bus should board the correct number of passengers, based on its capacity and the number of those waiting. The bus should correctly board these passengers by calling up/down on the semaphores suitably. The bus code should also update `waiting_count` as required. Once boarding completes, the bus thread should call `depart()`. You can use any extra local variables in the code of the bus thread, like integers, loop indices and so on. However, you must not use any other extra synchronization primitives.

Ans:

```
down(mutex)
N = min(waiting_count, K)
for i= 1 to N
    up(bus_arrived)
    down(passenger_boarded)
    waiting_count = waiting_count - N
up(mutex)
depart()
```

26. Consider a roller coaster ride at an amusement park. The ride operator runs the ride only when there are exactly N riders on it. Multiple riders arrive at the ride and queue up at the entrance of the ride. The ride operator waits for N riders to accumulate, and may even take a nap as he waits. Once N riders have arrived, the riders call out to the operator indicating they are ready to go on the ride. The operator then opens the gate to the ride and signals exactly N riders to enter the ride. He then waits until these N riders enter the ride, and then proceeds to start the ride.

We model the operator and riders as threads in a program. You must write pseudocode for the operator and rider threads to enable the behavior described above. Shown below is the skeleton code for the operator and rider threads. Complete the code to achieve the behavior described above. You can assume that the functions to open, start, and enter ride are implemented elsewhere, and these functions do what the names say they do. You must write the synchronization logic around these functions in order to invoke these functions at the appropriate times. You must use only locks and condition variables for synchronization in your solution. You may declare, initialize, and use other variables (counters etc.) as required in your solution.

```
//operator code, fill in the missing details  
....  
open_ride()  
....  
start_ride()  
....  
  
//rider thread, fill in the missing details  
....  
enter_ride()  
....
```

Ans:

```
//variables: int rider_count (initialized to 0)
//variables: int enter_count (initialized to 0)
//condvar cv_rider, cv_operator1, cv_operator2
//mutex

//operator
lock(mutex)
while(rider_count < N) wait(cv_operator1, mutex)

open_ride()
do N times: signal(cv_rider)
while(enter_count < N) wait(cv_operator2, mutex)

start_ride()
unlock (mutex)

//rider
lock(mutex)
rider_count++
if(rider_count == N) signal(cv_operator1)
wait(cv_rider, mutex) // all wait, even N-th guy

enter_ride()

enter_count++
if(enter_count == N) signal(cv_operator2)
unlock (mutex)
```

27. A host of a party has invited $N > 2$ guests to his house. Due to fear of Covid-19 exposure, the host does not wish to open the door of his house multiple times to let guests in. Instead, he wishes that all N guests, even though they may arrive at different times to his door, wait for each other and enter the house all at once. The host and guests are represented by threads in a multi-threaded program. Given below is the pseudocode for the host thread, where the host waits for all guests to arrive, then calls `openDoor()`, and signals a condition variable once. You must write the corresponding code for the guest threads. The guests must wait for all N of them to arrive and for the host to open the door, and must call `enterHouse()` only after that. You must ensure that all N waiting guests enter the house after the door is opened. You must use only locks and condition variables for synchronization.

The following variables are used in this solution: lock `m`, condition variables `cv_host` and `cv_guest`, and integer `guest_count` (initialized to 0). You must not use any other variables in the guest for synchronization.

```
//host
lock (m)
while(guest_count < N)
    wait (cv_host, m)
openDoor ()
signal (cv_guest)
unlock (m)
```

Ans:

```
//guest
lock (m)
guest_count++
if(guest_count == N)
    signal (cv_host)
wait (cv_guest, m)
signal (cv_guest)
unlock (m)
enterHouse ()
```

28. Consider the classic readers-writers synchronization problem described below. Several processes/threads wish to read and write data shared between them. Some processes only want to read the shared data (“readers”), while others want to update the shared data as well (“writers”). Multiple readers may concurrently access the data safely, without any correctness issues. However, a writer must not access the data concurrently with anyone else, either a reader or a writer. While it is possible for each reader and writer to acquire a regular mutex and operate in perfect mutual exclusion, such a solution will be missing out on the benefits of allowing multiple readers to read at the same time without waiting for other readers to finish. Therefore, we wish to have special kind of locks called reader-writer locks that can be acquired by processes/threads in such situations. These locks have separate lock/unlock functions, depending on whether the thread asking for a lock is a reader or writer. If one reader asks for a lock while another reader already has it, the second reader will also be granted a read lock (unlike in the case of a regular mutex), thus encouraging more concurrency in the application.

Write down pseudocode to implement the functions `readLock`, `readUnlock`, `writeLock`, and `writeUnlock` that are invoked by the readers and writers to realize reader-writer locks. You must use condition variables and mutexes only in your solution.

Ans: A boolean variable `writer_present`, and two condition variables, `reader_can_enter` and `writer_can_enter`, are used.

```

readLock:
lock (mutex)
while(writer_present)
    wait (reader_can_enter)
read_count++
unlock (mutex)

readUnlock:
lock (mutex)
read_count--
if (read_count==0)
    signal (writer_can_enter)
unlock (mutex)

writeLock:
lock (mutex)
while(read_count > 0 || writer_present)
    wait (writer_can_enter)
writer_present = true
unlock (mutex)

writeUnlock:
lock (mutex)
writer_present = false
signal (writer_can_enter)
signal_broadcast (reader_can_enter)
unlock (mutex)

```

29. Consider the readers and writers problem discussed above. Recall that multiple readers can be allowed to read concurrently, while only one writer at a time can access the critical section. Write down pseudocode to implement the functions readLock, readUnlock, writeLock, and writeUnlock that are invoked by the readers and writers to realize read/write locks. You must use **only** semaphores, and no other synchronization mechanism, in your solution. Further, you must avoid using more semaphores than is necessary. Clearly list all the variables (semaphores, and any other flags/counters you may need) and their initial values at the start of your solution. Use the notation down (x) and up (x) to invoke atomic down and up operations on a semaphore x that are available via the OS API. Use sensible names for your variables.

Ans:

```
sem lock = 1; sem writer_can_enter = 1; int readCount = 0;

readLock:
down(lock)
readCount++
if(readCount ==1)
    down(writer_can_enter) //don't coexist with a writer
up(lock)

readUnlock:
down(lock)
readCount--
if(readCount == 0)
    up(writer_can_enter)
up(lock)

writeLock:
down(writer_can_enter)

writeUnlock:
up(writer_can_enter)
```

30. Consider the readers and writers problem as discussed above. We wish to implement synchronization between readers and writers, while giving **preference to writers**, where no waiting writer should be kept waiting for longer than necessary. For example, suppose reader process R1 is actively reading. And a writer process W1 and reader process R2 arrive while R1 is reading. While it might be fine to allow R2 in, this could prolong the waiting time of W1 beyond the absolute minimum of waiting until R1 finishes. Therefore, if we want writer preference, R2 should not be allowed before W1. Your goal is to write down pseudocode for read lock, read unlock, write lock, and write unlock functions that the processes should call, in order to realize read/write locks with writer preference. You must use only simple locks/mutexes and conditional variables in your solution. Please pick sensible names for your variables so that your solution is readable.

Ans:

```

readLock:
lock (mutex)
while (writer_present || writers_waiting > 0)
    wait (reader_can_enter, mutex)
readcount++
unlock (mutex)

readUnlock:
lock (mutex)
readcount--
if (readcount==0)
    signal (writer_can_enter)
unlock (mutex)

writeLock:
lock (mutex)
writer_waiting++
while (readcount > 0 || writer_present)
    wait (writer_can_enter, mutex)
writer_waiting--
writer_present = true
unlock (mutex)

writeUnlock:
lock (mutex)
writer_present = false
if (writer_waiting==0)
    signal_broadcast (reader_can_enter)
else
    signal (writer_can_enter)
unlock (mutex)

```

31. Write a solution to the readers-writers problem with preference to writers discussed above, but using only semaphores.

Ans:

```
sem rlock = 1; sem wlock = 1;
sem reader_can_try = 1; sem writer_can_enter = 1;
int readCount = 0; int writeCount = 0;

readLock:
down(reader_can_try) //new sem blocks reader if writer waiting
down(rlock)
readCount++
if(readCount ==1)
    down(writer_can_enter) //don't coexist with a writer
up(rlock)
up(reader_can_try)

readUnlock:
down(rlock)
readCount--
if(readCount == 0)
    up(writer_can_enter)
up(rlock)

writeLock:
down(wlock)
writerCount++
if(writerCount==1)
    down(reader_can_try)
up(wlock)
down(writer_can_enter) //release wlock and then block

writeUnlock:
down(wlock)
writerCount--
if(writerCount == 0)
    up(reader_can_try)
up(wlock)

up(writer_can_enter)
```

32. Consider the famous dining philosophers' problem. N philosophers are sitting around a table with N forks between them. Each philosopher must pick up both forks on her left and right before she can start eating. If each philosopher first picks the fork on her left (or right), then all will deadlock while waiting for the other fork. The goal is to come up with an algorithm that lets all philosophers eat, without deadlock or starvation. Write a solution to this problem using condition variables.

Ans: A variable `state` is associated with each philosopher, and can be one of `EATING` (holding both forks) or `THINKING` (when not eating). Further, a condition variable is associated with each philosopher to make them sleep and wake them up when needed. Each philosopher must call the `pickup` function before eating, and `putdown` function when done. Both these functions use a `mutex` to change states only when both forks are available.

```

bothForksFree(i):
    return (state[leftNbr(i)] != EATING &&
            state[rightNbr(i)] != EATING)

pickup(i):
    lock(mutex)
    while (!bothForksFree(i))
        wait(condvar[i])
    state[i] = EATING
    unlock(mutex)

putdown(i):
    lock(mutex)
    state[i] = THINKING
    if(bothForksFree(leftNbr(i)))
        signal(leftNbr(i))
    if(bothForksFree(rightNbr(i)))
        signal(rightNbr(i))
    unlock(mutex)

```

33. Consider a clinic with one doctor and a very large waiting room (of infinite capacity). Any patient entering the clinic will wait in the waiting room until the doctor is free to see her. Similarly, the doctor also waits for a patient to arrive to treat. All communication between the patients and the doctor happens via a shared memory buffer. Any of the several patient processes, or the doctor process can write to it. Once the patient “enters the doctors office”, she conveys her symptoms to the doctor using a call to `consultDoctor()`, which updates the shared memory with the patient’s symptoms. The doctor then calls `treatPatient()` to access the buffer and update it with details of the treatment. Finally, the patient process must call `noteTreatment()` to see the updated treatment details in the shared buffer, before leaving the doctor’s office. A template code for the patient and doctor processes is shown below. Enhance this code to correctly synchronize between the patient and the doctor processes. Your code should ensure that no race conditions occur due to several patients overwriting the shared buffer concurrently. Similarly, you must ensure that the doctor accesses the buffer only when there is valid new patient information in it, and the patient sees the treatment only after the doctor has written it to the buffer. You must use **only semaphores** to solve this problem. Clearly list the semaphore variables you use and their initial values first. Please pick sensible names for your variables.

Ans:

- (a) Semaphores variables:

```
pt_waiting = 0
treatment_done = 0
doc_avlbl = 1
```

- (b) Patient process:

```
down(doc_avlbl)
consultDoctor()
up(pt_waiting)
down(treatment_done)
noteTreatment()
up(doc_avlbl)
```

- (c) Doctor:

```
while(1) {
    down(pt_waiting)
    treatPatient()
    up(treatment_done)
}
```

34. Consider a multithreaded banking application. The main process receives requests to transfer money from one account to the other, and each request is handled by a separate worker thread in the application. All threads access shared data of all user bank accounts. Bank accounts are represented by a unique integer account number, a balance, and a lock of type `mylock` (much like a `pthreads mutex`) as shown below.

```
struct account {
    int accountnum;
    int balance;
    mylock lock;
};
```

Each thread that receives a transfer request must implement the transfer function shown below, which transfers money from one account to the other. Add correct locking (by calling the `dolock(&lock)` and `unlock(&lock)` functions on a `mylock` variable) to the transfer function below, so that no race conditions occur when several worker threads concurrently perform transfers. Note that you must use the fine-grained per account lock provided as part of the account object itself, and not a global lock of your own. Also make sure your solution is deadlock free, when multiple threads access the same pair of accounts concurrently.

```
void transfer(struct account *from, struct account *to, int amount) {

    from->balance -= amount; // dont write anything...
    to->balance += amount; // ...between these two lines

}
```

Ans: The accounts must be locked in order of their account numbers. Otherwise, a transfer from account X to Y and a parallel transfer from Y to X may acquire locks on X and Y in different orders and end up in a deadlock.

```
struct account *lower = (from->accountnum < to->accountnum) ?from:to;
struct account *higher = (from->accountnum < to->accountnum) ?to:from;
dolock (&(lower->lock));
dolock (&(higher->lock));

from->balance -= amount;
to->balance += amount;

unlock (&(lower->lock));
unlock (&(higher->lock));
```

35. Consider a process with three threads A, B, and C. The default thread of the process receives multiple requests, and places them in a request queue that is accessible by all the three threads A, B, and C. For each request, we require that the request must first be processed by thread A, then B, then C, then B again, and finally by A before it can be removed and discarded from the queue. Thread A must read the next request from the queue only after it is finished with all the above steps of the previous one. Write down code for the functions run by the threads A, B, and C, to enable this synchronization. You can only worry about the synchronization logic and ignore the application specific processing done by the threads. You may use any synchronization primitive of your choice to solve this question.

Ans: Solution using semaphores shown below. The order of processing is A1–B1–C–B2–A2. All threads run in a forever loop, and wait as dictated by the semaphores.

```
sem aldone = 0; b1done = 0; cdone = 0; b2done = 0;
```

ThreadA:

```
get request from queue and process
up(aldone)
down(b2 done)
finish with request
```

ThreadB:

```
down(aldone)
//do work
up(b1done)
down(cdone)
//do work
up(b2done)
```

ThreadC:

```
down(b1done)
//do work
up(cdone)
```

36. Consider two threads A and B that perform two operations each. Let the operations of thread A be A1 and A2; let the operations of thread B be B1 and B2. We require that threads A and B each perform their first operation before either can proceed to the second operation. That is, we require that A1 be run before B2 and B1 before A2. Consider the following solutions based on semaphores for this problem (the code run by threads A and B is shown in two columns next to each other). For each solution, explain whether the solution is correct or not. If it is incorrect, you must also point out why the solution is incorrect.

```
(a) sem A1Done = 0;    sem B1Done = 0;
   //Thread A           //Thread B
   A1                  B1
   down(B1Done)        down(A1Done)
   up(A1Done)          up(B1Done)
   A2                  B2

(b) sem A1Done = 0;    sem B1Done = 0;
   //Thread A           //Thread B
   A1                  B1
   down(B1Done)        up(B1Done)
   up(A1Done)          down(A1Done)
   A2                  B2

(c) sem A1Done = 0;    sem B1Done = 0;
   //Thread A           //Thread B
   A1                  B1
   up(A1Done)          up(B1Done)
   down(B1Done)        down(A1Done)
   A2                  B2
```

Ans:

- (a) Deadlocks, so incorrect.
- (b) Correct
- (c) Correct

37. Now consider a generalization of the above problem for the case of N threads that want to each execute their first operation before any thread proceeds to the second operation. Below is the code that each thread runs in order to achieve this synchronization. `count` is an integer shared variable, and `mutex` is a mutex binary semaphore that protects this shared variable. `step1Done` is a semaphore initialized to zero. You are told that this code is wrong and does not work correctly. Further, you can fix it by changing it slightly (e.g., adding one statement, or rearranging the code in some way). Suggest the change to be made to the code in the snippet below to fix it. You must use only semaphores and no other synchronization mechanism.

```
//run first step

down(mutex);
count++;
up(mutex);
if(count == N)
    up(step1Done);
down(step1Done);

//run second step
```

Ans: The problem is that the semaphore is decremented N times, but is only incremented once. To fix it, we must do up N times when count is N . Or, add up after the last down, so that it is performed N times by the N threads.

38. The cigarette smokers problem is a classical synchronization problem that involves 4 threads: one agent and three smokers. The smokers require three ingredients to smoke a cigarette: tobacco, paper, and matches. Each smoker has one of the three ingredients and waits for the other two, smokes the cigar once he obtains all ingredients, and repeats this forever. The agent repeatedly puts out two ingredients at a time and makes them available. In the correct solution of this problem, the smoker with the complementary ingredient should finish smoking his cigar. Consider the following solution to the problem. The shared variables are three semaphores `tobacco`, `paper` and `matches` initialized to 0, and semaphore `doneSmoking` initialized to 1. The agent code performs `down` (`doneSmoking`) , then picks two of the three ingredients at random and performs `up` on the corresponding two semaphores, and repeats. The smoker with tobacco runs the following code in a loop.

```
down (paper)
down (matches)
//make and smoke cigar
up (doneSmoking)
```

Similarly, the smoker with matches waits for tobacco and paper, and the smoker with paper waits for tobacco and matches, before signaling the agent that they are done smoking. Does the code above solve the synchronization problem correctly? If you answer yes, provide a justification for why the code is correct. If you answer no, describe what the error is and also provide a correct solution to the problem. (If you think the code is incorrect and are providing another solution, you may change the code of both the agent and the smokers. You can also introduce new variables as necessary. You must use only semaphores to solve the problem.)

Ans: The code is incorrect and deadlocks. One fix is to add semaphores for two ingredients at a time (e.g., `tobaccoAndPaper`). The smokers wait on these and the agent signals these. So there is no possibility of deadlock.

39. Consider a server program running in an online market place firm. The program receives buy and sell orders for one type of commodity from external clients. For every buy or sell request received by the server, the main process spawns a new buy or sell thread. We require that every buy thread waits until a sell thread arrives, and vice versa. A matched pair of buy and sell threads will both return a response to the clients and exit. You may assume that all buy/sell requests are identical to each other, so that any buy thread can be matched with any sell thread. The code executed by the buy thread is shown below (the code of the sell thread would be symmetric). You have to write the synchronization logic that must be run at the start of the execution of the thread to enable it to wait for a matching sell thread to arrive (if none exists already). Once the threads are matched, you may assume that the function `completeBuy()` takes care of the application logic for exchanging information with the matching thread, communicating with the client, and finishing the transaction. You may use any synchronization technique of your choice.

```
//declare any variables here

buy_thread_function:
    //start of sync logic

    //end of sync logic
    completeBuy();
```

Ans:

```
sem buyer = 0; sem seller = 0;
```

Buyer thread:

```
up(buyer)
down(seller)
completeBuy()
```

40. Consider the following classical synchronization problem called the barbershop problem. A barbershop consists of a room with N chairs. If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop. If the barber is busy, but chairs are available, then the customer sits in one of the free chairs and awaits his turn. The barber moves onto the next waiting seated customer after he finishes one hair cut. If there are no customers to be served, the barber goes to sleep. If the barber is asleep when a customer arrives, the customer wakes up the barber to give him a hair cut. A waiting customer vacates his chair after his hair cut completes. Your goal is to write the pseudocode for the customer and barber threads below with suitable synchronization. You must use only semaphores to solve this problem. Use the standard notation of invoking up/down functions on a semaphore variable.

The following variables (3 semaphores and a count) are provided to you for your solution. You must use these variables and declare any additional variables if required.

```
semaphore mutex = 1, customers = 0, barber = 0;  
int waiting_count = 0;
```

Some functions to invoke in your customer and barber threads are:

- A customer who finds the waiting room full should call the function `leave()` to exit the shop permanently. This function does not return.
- A customer should invoke the function `getHairCut()` in order to get his hair cut. This function returns when the hair cut completes.
- The barber thread should call `cutHair()` to give a hair cut. When the barber invokes this function, there should be exactly one customer invoking `getHairCut()` concurrently.

Ans:

Customer:

```
down(mutex)
if(waiting_count == N)
    up(mutex)
    leave()
waiting_count++
up(mutex)

up(customers)
down(barber)

getHairCut()

down(mutex)
waiting_count--
up(mutex)
```

Barber:

```
up(barber)
down(customers)
cutHair()
```

41. Consider a multithreaded application server handling requests from clients. Every new request that arrives at the server causes a new thread to be spawned to handle that request. The server can provide service to only one request/thread at a time, and other threads that arrive when the server is busy must wait for service using a synchronization primitive (semaphore or condition variable). In order to avoid excessive waiting times, the server does not wish to have more than N requests/threads in the system (including the waiting requests and any request it is currently serving). You may assume that $N > 2$. Given this constraint, a newly arriving thread must first check if N other requests are already in the system: if yes, it must exit without waiting and return an error value to the client, by calling the function `thr_exit_failure()`. This function terminates the thread and does not return.

When a thread is ready for service, it must call the function `get_service()`. Your code should ensure that no more than one thread calls this function at any point of time. This function blocks the thread for the duration of the service. Note that, while the thread receiving service is blocked, other arriving threads must be free to join the queue, or exit if the system is overloaded. After a thread returns from `get_service()`, it must enable one of the waiting threads to seek service (if any are waiting), and then terminate itself successfully by calling the function `thr_exit_success()`. This function terminates the thread and does not return.

You are required to write pseudocode of the function to be run by the request threads in this system, as per the specification above. Your solution must use only locks and condition variables for synchronization. Clearly state all the variables used and their initial values at the start of your solution.

Ans

```
int num_requests=0;
bool server_busy = false
cv, mutex

lock(mutex)

if(num_requests == N)
    unlock(mutex)
    the_exit_failure()

num_requests++

if(server_busy)
    wait(cv, mutex)

server_busy = true
unlock(mutex)

get_service()

lock(mutex)
num_requests--
server_busy = false

if(num_requests > 0)
    signal(cv)

unlock(mutex)
thr_exit_success()
```

42. Consider the previous problem, but now assume that N is infinity. That is, all arriving threads will wait (if needed) for their turn in the queue of a synchronization primitive, get served when their turn comes, and exit successfully. Write the pseudocode of the function to be run by the threads with this modified specification. Your solution must only use semaphores for synchronization, and only the correct solution that uses the least number of semaphores will get full credit. Clearly state all the variables used and their initial values at the start of your solution.

Ans

```
sem waiting = 1

down(waiting)
get_service()
up(waiting)
thr_exit_success()
```

43. Consider the following synchronization problem. A group of children are picking chocolates from a box that can hold up to N chocolates. A child that wants to eat a chocolate picks one from the box to eat, unless the box is empty. If a child finds the box to be empty, she wakes up the mother, and waits until the mother refills the box with N chocolates. Unsynchronized code snippets for the child and mother threads are as shown below:

```
//Child
while True:
    getChocolateFromBox()
    eat()

//Mother
while True:
    refillChocolateBox(N)
```

You must now modify the code of the mother and child threads by adding suitable synchronization such that a child invokes `getChocolateFromBox()` only if the box is non-empty, and the mother invokes `refillChocolateBox(N)` only if the box is fully empty. Solve this question using only locks and condition variables, and no other synchronization primitive. The following variables have been declared for use in your solution.

```
int count = 0;
mutex m; // you may invoke lock and unlock
condvar fullBox, emptyBox; //you may perform wait and signal
                           //or signal_broadcast
```

- (a) Code for child thread
- (b) Code for mother thread

Ans:

```
//Child
while True:
    lock(m)
    while(count == 0)
        signal(emptyBox)
        wait(fullBox, m)
    getChocolateFromBox()
    eat()
    count--
    signal(fullBox) //optional
    unlock(m)

//Mother
while True:
    lock(m)
    if(count > 0)
        wait(emptyBox, m)
    refillChocolateBox(N)
    count += N
    signal(fullBox)
    unlock(m)
```

There are two ways of waking up sleeping children. Either the mother does a signal broadcast to all children. Or every child that eats a chocolate wakes up another sleeping child. You may also assume that signal by mother wakes up all children.

44. Repeat the above question, but your solution now must use only semaphores and no other synchronization primitive. The following variables have been declared for use in your solution.

```
int count = 0;  
semaphore m, fullBox, emptyBox;  
//initial values of semaphores are not specified  
//you may invoke up and down methods on a semaphore
```

- (a) Initial values of the semaphores
- (b) Code for child thread
- (c) Code for mother thread

Ans:

```
m = 1, fullBox = 0, emptyBox = 0
```

```
//Child  
while True:  
    down(m)  
    if(count == 0)  
        up(emptyBox)  
        down(fullBox)  
        count += N  
    getChocolateFromBox()  
    eat()  
    count--  
    up(m)
```

```
//Mother  
while True:  
    down(emptyBox)  
    refillChocolateBox(N)  
    up(fullBox)
```

Here the subtlety is the lock m. Mother can't get lock to update count after filling the box, as that will cause a deadlock. In general, if child sleeps with mutex m locked, then mother cannot request the same lock.

45. Consider the classic “barrier” synchronization problem, where N threads wish to synchronize with each other as follows. N threads arrive into the system at different times and in any order. The arriving threads must wait until all N threads have arrived into the system, and continue execution only after all N threads have arrived. We wish to write logic to synchronize the threads in the manner stated above using semaphores. Below are three possible solutions to the problem. You are told that one of the solutions is correct and the other two are wrong. Identify the correct solution amongst the three given options. Further, for each of the other incorrect solutions, explain clearly why the solution is wrong. The following shared variables are declared for use in each solution.

```

int count = 0;
sem mutex; //initialized to 1
sem barrier; //initialized to 0

(a) down(mutex)
    count++
    if(count == N) up(barrier)
    up(mutex)

    down(barrier)

    //wait done; proceed to actual task

(b) down(mutex)
    count++
    if(count == N) up(barrier)
    up(mutex)

    down(barrier)
    up(barrier)

    //wait done; proceed to actual task

(c) down(mutex)
    count++
    if(count == N) up(barrier)
    down(barrier)
    up(barrier)
    up(mutex)

    //wait done; proceed to actual task

```

Ans: In (a) up is done only once when many threads are waiting on down. In (c), down(barrier) is called when mutex held, so code deadlocks. (b) is correct answer.

46. Consider the barrier synchronization primitive discussed in class, where the N threads of an application wait until all the threads have arrived at a barrier, before they proceed to do a certain task. You are now required to write the code for a reusable barrier, where the N application threads perform a series of steps in a loop, and use the same barrier code to synchronize for each iteration of the loop. That is, your solution should ensure that all threads wait for each other before the start of each step, and proceed to the next step only after all threads have completed the previous step. Your solution must only use semaphores. The following functions can be invoked on a semaphore s used in this question: $\text{down}(s)$, $\text{up}(s)$, and $\text{up}(s, n)$. While the first two functions are as studied in class, the function $\text{up}(s, n)$ simply invokes $\text{up}(s)$ n times atomically.

We have provided you some code to get started. Shown below is the code to be run by each application thread, including the code to wait at the barrier. However, this is not the correct solution, as this code only works as a single-use barrier, i.e., it only ensures that the threads synchronize at the barrier once, and cannot be used to synchronize multiple times (can you figure out why?). You are required to modify this code to make it reusable, such that the threads can synchronize at the barrier multiple times for the multiple steps to be performed.

Your solution must only use the following variables: `int count = 0;` and semaphores (initial values as given): `sem mutex = 1; sem barrier1 = 0; sem barrier2 = 0;`

For each step to be executed by the threads, do:

```
//add code here if required to make barrier reusable

down(mutex)
count++
if(count == N) up(barrier1, N)
up(mutex)
down(barrier1)

... wait done, execute actual task of this step ...

//add code here if required to make barrier reusable for next step
```

Ans: The extra code to be added is at the end of completing a step, where you make all threads wait once again.

```
down(mutex)
count--
if(count==0) up(barrier2, N)
up(mutex)
down(barrier2)
```

47. Consider a web server that is supposed to serve a batch of N requests. Each request that arrives at the web server spawns a new thread. The arriving threads wait until N of them accumulate, at which point all of them proceed to get service from the server. Shown below is the code executed by each arriving thread, that causes it to wait until all the other threads arrive. The variable `count` is initialized to N . The code also uses `wait` and `signal` primitives on a condition variable; and you may assume that the `signal` primitive wakes up all waiting threads (not just one of them).

```

lock (mutex)
  count--;
unlock (mutex)

if(count > 0) {
    lock (mutex)
    wait (cv, mutex)
    unlock (mutex)
}
else {
    lock (mutex)
    signal (cv)
    unlock (mutex)
}

... wait done, proceed to server ...

```

You are told that the code above is incorrect, and can sometimes cause a deadlock. That is, in some executions, all N threads do not go to the server for service, even though they have arrived.

- (a) Using an example, explain the exact sequence of events that can cause a deadlock. You must write your answers as bullet points, with one event per bullet point, starting from threads arriving in the system until the deadlock.
- (b) Explain how you will fix this deadlock and correct the code shown above. You must retain the basic structure of the code. Indicate your changes next to the code snippet above.

Ans: The given incorrect solution may cause a missed wakeup. For example, some thread decides to wait and goes inside the if-loop, but is context switched out before calling `wait` (and before it acquires the lock). Now, if `count` hits 0 and `signal` happens before it runs again, it will wait with no one to wake it up, leading to deadlock. The fix is simply holding the lock all through the condition checking and waiting.

48. Consider an application that has $K + 1$ threads running on a Linux-like OS ($K > 1$). The first K threads of an application execute a certain task T1, and the remaining one thread executes task T2. The application logic requires that task T1 is executed $N > 1$ times, followed by task T2 executed once, and this cycle of N executions of T1 followed by one execution of T2 continue indefinitely. All K threads should be able to participate in the N executions of task T1, even though it is not required to ensure perfect fairness amongst the threads.

Shown below is one possible set of functions executed by the threads running tasks T1 and T2. You are told that this solution has two bugs in the code run by the thread performing task T2. Briefly describe the bugs in the space below, and suggest small changes to the corresponding code to fix these bugs (you may write your changes next to the code snippet). You must not change the code corresponding to task T1 in any way. All threads share a counter `count` (initialized to 0), a mutex variable `m`, and two condition variables `t1cv`, and `t2cv`. Here, the function `signal` on a condition variable wakes up only one of the possibly many sleeping threads.

```
//function run by K threads of task T1
while True {
    lock(m)
    if(count >= N) {
        signal(t2cv)
        wait(t1cv, m)
    }
    //.. do task T1 once ..
    count++
    unlock(m)
}
//function run by thread of task T2
while True {
    lock(m)
    wait(t2cv, m)
    // .. do task T2 once
    count = 0
    signal(t1cv)
    unlock(m)
}
```

Ans: (a) check `count < N` and only then wait (b) signal broadcast instead of signal

49. You are now required to solve the previous question using semaphores for synchronization. You are given the pseudocode for the function run by the thread executing task T2 (which you must not change). You are now required to write the corresponding code executed by the K threads running task T1. You must use the following semaphores in your solution: mutex, t1sem, t2sem. You must initialize them suitably below. The variable count (initialized to 0) is also available for use in your solution.

Ans:

```
//fill in initial values of semaphores
sem_init(mutex,      );
sem_init(t1sem,      );
sem_init(t2sem,      );
//other variables
int count = 0

//function run by thread executing T2
while True {
    down(t2sem)
    //.. do task T2 ..
    up(t1sem)
}

//function run by threads executing task T1
while True {

}
```

Ans:

```
mutex=1, t1sem=0, t2sem=0

down(mutex)
if(count == N)
    up(t2sem)
    down(t1sem)
    count = 0

do task T1 once
count++
up(mutex)
```

50. Multiple people are entering and exiting a room that has a light switch. You are writing a computer program to model the people in this situation as threads in an application. You must fill in the functions `onEnter()` and `onExit()` that are invoked by a thread/person when the person enters and exits a room respectively. We require that the first person entering a room must turn on the light switch by invoking the function `turnOnSwitch()`, while the last person leaving the room must turn off the switch by invoking `turnOffSwitch()`. You must invoke these functions suitably in your code below. You may use any synchronization primitives of your choice to achieve this desired goal. You may also use any variables required in your solution, which are shared across all threads/persons.
- (a) Variables and initial values
 - (b) Code `onEnter()` to be run by thread/person entering
 - (c) Code `onExit()` to be run by thread/person exiting

Ans:

```
variables: mutex, count
```

```
onEnter():
lock(mutex)
count++
if(count==1) turnOnSwitch()
unlock(mutex)

onExit():
lock(mutex)
count--
if(count==0) turnOffSwitch()
unlock(mutex)
```

Lectures on Operating Systems (Mythili Vutukuru, IIT Bombay)

Lab: Pthreads Synchronization

In this lab, you will learn the basics of multi-threaded programming, and synchronizing multiple threads using locks and condition variables. You will use the `pthreads` (POSIX threads) API in this assignment.

Before you begin

Please familiarize yourself with the `pthreads` API thoroughly. Many helpful tutorials and sample programs are available online. Practice writing simple programs with multiple threads, using locks and condition variables for synchronization across threads. Remember that you must include the header file `<pthread.h>` and compile code using the `-lpthread` flag when using this API.

Warm-up exercises

You may write the following simple programs before you get started with this lab.

1. Write a program that has a counter as a global variable. Spawn 10 threads in the program, and let each thread increment the counter 1000 times in a loop. Print the final value of the counter after all the threads finish—the expected value of the counter is 10000. Run this program first without using locking across threads, and observe the incorrect updation of the counter due to race conditions (the final value will be slightly less than 10000). Next, use locks when accessing the shared counter and verify that the counter is now updated correctly.
2. Write a program where the main default thread spawns N threads. Thread i should print the message “I am thread i ” to screen and exit. The main thread should wait for all N threads to finish, then print the message “I am the main thread”, and exit.
3. Write a program where the main default thread spawns N threads. When started, thread i should sleep for a random interval between 1 and 10 seconds, print the message “I am thread i ” to screen, and exit. Without any synchronization between the threads, the threads will print their messages in any order. Add suitable synchronization using condition variables such that the threads print their messages in the order 1, 2, ..., N . You may want to start with $N = 2$ and then move on to larger values of N .
4. Write a program with N threads. Thread i must print number i in a continuous loop. Without any synchronization between the threads, the threads will print their numbers in any order. Now, add synchronization to your code such that the numbers are printed in the order 1, 2, ..., N , 1, 2, ..., N , and so on. You may want to start with $N = 2$ and then move on to larger values of N .

Part A: Master-Worker Thread Pool

In this part of the lab, you will implement a simple master and worker thread pool, a pattern that occurs in many real life applications. The master threads produce numbers continuously and place them in a buffer, and worker threads will consume them, i.e., print these numbers to the screen. This simple program is an example of a master-worker thread pool pattern that is commonly found in several real-life application servers. For example, in the commonly used multi-threaded architecture for web servers, the server has one or more master threads and a pool of worker threads. When a new connection arrives from a web client, the master accepts the request and hands it over to one of the workers. The worker then reads the web request from the network socket and writes a response back to the client. Your simple master-worker program is similar in structure to such applications, albeit with much simpler request processing logic at the application layer.

You are given a skeleton program `master-worker-skeleton.c`. This program takes 4 command line arguments: how many numbers to “produce” (M), the maximum size of the buffer in which the produced numbers should be stored (N), the number of worker threads to consume these numbers (C), and the number of master threads to produce numbers (P). The skeleton code spawns P master threads, that produce the specified number of integers from 0 to $M - 1$ into a shared buffer array. The main program waits for these threads to join, and then terminates. The skeleton code given to you does not have any logic for synchronization.

You must write your solution in the file `master-worker.c`. You must modify this skeleton code in the following ways. You must add code to spawn the required number of worker threads, and write the function to be run by these threads. This function will remove/consume items from the shared buffer and print them to screen. Further, you must add logic to correctly synchronize the producer and consumer threads in such a way that every number is produced and consumed exactly once. Further, producers must not try to produce when the buffer is full, and consumers should not consume from an empty buffer. While you need to ensure that all C workers are involved in consuming the integers, it is not necessary to ensure perfect load balancing between the workers. Once all M integers (from 0 to $M - 1$) have been produced and consumed, all threads must exit. The main thread must call `pthread_join` on the master and worker threads, and terminate itself once all threads have joined. Your solution must only use `pthread`s condition variables for waiting and signaling: busy waiting is not allowed.

Your code can be compiled as shown below.

```
gcc master-worker.c -lpthread
```

If your code is written correctly, every integer from 0 to $M - 1$ will be produced exactly once by the master producer thread, and consumed exactly once by the worker consumer threads. We have provided you with a simple testing script (`test-master-worker.sh` which invokes the script `check.awk`) that checks this above invariant on the output produced by your program. The script relies on the two print functions that must be invoked by the producer and consumer threads: the master thread must call the function `print_produced` when it produces an integer into the buffer, and the worker threads must call the function `print_consumed` when it removes an integer from the buffer to consume. You must invoke these functions suitably in your solution. Please do not modify these print functions, as their output will be parsed by the testing script.

Please ensure that you test your case carefully, as tricky race conditions can pop up unexpectedly. You must test with up to a few million items produced, and with a few hundreds of master/worker threads. Test for various corner cases like a single master or a single worker thread or for a very small buffer size as well. Also test for cases when the number of items produced is not a multiple of the buffer size or the

number of master/worker threads, as such cases can uncover some tricky bugs. Increasing the number of threads to large values beyond a few hundred will cause your system to slow down considerably, so exercise caution.

Part B: Reader-Writer Locks

Consider an application where multiple threads of a process wish to read and write data shared between them. Some threads only want to read (let's call them "readers"), while others want to update the shared data ("writers"). In this scenario, it is perfectly safe for multiple readers to concurrently access the shared data, as long as no other writer is updating it. However, a writer must still require mutual exclusion, and must not access the data concurrently with any other thread, whether a reader or a writer. A reader-writer lock is a special kind of a lock, where the acquiring thread can specify whether it wishes to acquire the lock for reading or writing. That is, this lock will expose two separate locking functions, say, *ReaderLock()* and *WriterLock()*, and analogous unlock functions. If a lock is acquired for reading, other threads that wish to read may also be permitted to acquire the lock at the same time, leading to improved concurrency over traditional locks.

There are two flavors of the reader-writer lock, which we will illustrate with an example. Suppose a reader thread R1 has acquired a reader-writer lock for reading. While R1 holds this lock, a writer thread W and another reader thread R2 have both requested the lock. Now, it is fine to allow R2 also to simultaneously acquire the lock with R1, because both are only reading shared data. However, allowing R2 to acquire the lock may prolong the waiting time of the writer thread W, because W has to now wait for both R1 and R2 to release the lock. So, whether we wish to permit more readers to acquire the lock when a writer is waiting is a design decision in the implementation of the lock. When a reader-writer lock is implemented with *reader preference*, additional readers are allowed to concurrently hold the lock with previous readers, even if a writer thread is waiting for the lock. In contrast, when a reader-writer lock is implemented with *writer preference*, additional readers are not granted the lock when a writer is already waiting. That is, we do not prolong the waiting time of a writer any more than required.

In this part of the lab, you must implement both flavors of the reader-writer lock. You must complete the definition of the structure that captures the reader-writer lock in `rwlock.h`. The following functions to be supported by this lock are also defined in the header file:

- The function `InitializeReadWriteLock()` must initialize the lock suitably.
- The function `ReaderLock()` is invoked by a reader thread before entering a read-only critical section, and the function `ReaderUnlock()` is invoked by the reader when exiting the critical section.
- The function `WriterLock()` is invoked by a writer thread before entering a critical section with shared data updates, and the function `WriterUnlock()` is invoked by the writer when exiting the critical section.

You must write the code to implement these functions. You must write code that implements reader-writer locks with reader preference in `rwlock-reader-pref.cpp`. Similarly, you must implement reader-writer locks with writer preference in `rwlock-writer-pref.cpp`. Note that both implementations of the lock must share the same header file, including the definition of the reader-writer lock structure. Therefore, your lock structure may have some fields that are only used in one version of the code and not the other.

As part of the autograding scripts, you are given two tester programs that test each variant of your reader-writer lock, and an autograding script that runs both in one go. The tester program takes two command line arguments, say R and W . The program then spawns R reader threads, followed by W writer threads, followed by R additional reader threads. Each thread, upon creation, tries to acquire the same reader-writer lock as a reader or writer (as the case may be), holds the lock for a long period of time, and finally releases the lock. The program judges the correctness of your implementation by observing the relative ordering of the acquisitions and releases of these locks. By invoking this tester with different values of R and W , one can test the reader-writer lock code reasonably thoroughly. Of course, you are encouraged to write your own test programs that use the reader-writer lock as well.

Part C: Semaphores using `pthreads`

In this part of the lab, you will implement the synchronization functionality of semaphores using `pthreads` mutexes and condition variables. Let's call these new userspace semaphores that you implement as zemaphores, to avoid confusing them with semaphores provided by the Linux kernel. You must define your zemaphore structure in the file `zemaphore.h`, and implement the functions `zem_init`, `zem_up` and `zem_down` that operate on this structure in the file `zemaphore.c`. The semantics of these zemaphore functions are similar to those of the semaphores you have studied in class.

- The function `zem_init` initializes the specified zemaphore to the specified value.
- The function `zem_up` increments the counter value of the zemaphore by one, and wakes up any one sleeping thread.
- The function `zem_down` decrements the counter value of the zemaphore by one. If the value is negative, the thread blocks and is context switched out, to be woken up by an up operation on the zemaphore at a later point.

Once you implement your zemaphores, you can use the program `test-zem.c` to test your implementation. This program spawns two threads, and all three threads (the main thread and the two new threads) share a zemaphore. Before you implement the zemaphore logic, the new threads will print to screen before the main thread. However, after you implement the zemaphore correctly, the main thread will print first, owing to the synchronization enabled by the zemaphore. You must not modify this test program in any way, but only use it to test your zemaphore implementation.

Next, you are given a simple program with three threads in `test-toggle.c`. In its current form, the three threads will all print to screen in any order. Modify this program in such a way that the print statements of the threads are interleaved in the order `thread0, thread1, thread2, thread0, thread1, thread2, ...` and so on. You must only use your zemaphores to achieve this synchronization between the threads. You must not directly use the mutexes and condition variables of the `pthreads` library in the file `test-toggle.c`.

The script `test-zem.sh` will compile and run these test programs for you and can be used for testing. Please see the compilation commands in the script to understand how to compile code that uses `pthreads` using the `-lpthread` flag.

Part D: Your own synchronization problem

In this part, you will implement your own synchronization pattern, much like the producer-consumer (master-worker) pattern in part A or the reader-writer locks in part B. You may look through the practice problems, or the “Little Book of Semaphores” for inspiration. You may pick any of the existing problems from these sources (except producer-consumer and reader-writer locks, which are covered in parts A and B already, or the simple patterns included as test files in part C), or you can come up with your own toy/real-life problem as well.

Your problem should have at least two different agents/threads, e.g., producers and consumers, each doing different things, and requiring some kind of synchronization between them. The synchronization requirement should be more complex than simple patterns like “thread 2 must run after thread 1” which are given as test cases in part C. Your program should spawn multiple threads to create these different agents, with some randomness in their start times, in order to achieve different interleavings of threads during testing. Each agent/thread should do some dummy work in its start function and print some message when it does the work, e.g., producer prints something when it produces and consumer prints something when it consumes. Without proper synchronization, the threads may function incorrectly, e.g., consumer may consume from an empty buffer. But with correct synchronization added, your print statements should indicate that the threads are synchronized correctly as expected. You must define the expected correct behavior of the various threads and demonstrate in your solution that the correct behavior is indeed achieved by looking at the output.

You will provide two different solutions to your synchronization problem, one using condition variables and the other using semaphores. For condition variables, you will use the CV and mutex abstractions from the `pthreads` API. For semaphores, you will use your own semaphore (zemaphore) abstraction implemented by you in part C above. You should develop, test, and demonstrate the CV and sempahore solution in separate C/C++ files. You will use condition variables/mutexes/semaphores as shared global variables in your program, that are available for use by all threads. During your testing, you should run your programs multiple times, with different interleavings of the threads, to demonstrate that your solution is correct.

There is no starter/template code provided for this part of the assignment. You may use example code from other parts of this assignment to help you get started.

Submission instructions

- For part A, you must submit `master-worker.c`.
For part B, submit `rwlock.h`, `rwlock-reader-pref.cpp`, and `rwlock-writer-pref.cpp`.
For part C, submit `zemaphore.h`, `zemaphore.c`, and your modified `test-toggle.c`.
For part D, submit all code written by you.
- Place these files and any other files you wish to submit in your submission directory, with the directory name being your roll number (say, 12345678).
- Tar and gzip the directory using the command `tar -zcvf 12345678.tar.gz 12345678` to produce a single compressed file of your submission directory. Submit this tar gzipped file on Moodle.

Practice Problems: Memory

1. Provide one advantage of using the slab allocator in Linux to allocate kernel objects, instead of simply allocating them from a dynamic memory heap.

Ans: A slab allocator is fast because memory is preallocated. Further, it avoids fragmentation of kernel memory.

2. In a 32-bit architecture machine running Linux, for every physical memory address in RAM, there are at least 2 virtual addresses pointing to it. That is, every physical address is mapped at least twice into the virtual address space of some set of processes. [T/F]

Ans: F (this may be true of simple OS like xv6 studied in class, but not generally true)

3. Consider a system with N bytes of physical RAM, and M bytes of virtual address space per process. Pages and frames are K bytes in size. Every page table entry is P bytes in size, accounting for the extra flags required and such. Calculate the size of the page table of a process.

Ans: $M/K * P$

4. The memory addresses generated by the CPU when executing instructions of a process are called logical addresses. [T/F]

Ans: T

5. When a C++ executable is run on a Linux machine, the kernel code is part of the executable generated during the compilation process. [T/F]

Ans: F (it is only part of the virtual address space)

6. When a C++ executable is run on a Linux machine, the kernel code is part of the virtual address space of the running process. [T/F]

Ans: T

7. Consider a Linux-like OS running on x86 Intel CPUs. Which of the following events requires the OS to update the page table pointer in the MMU (and flush the changes to the TLB)? Answer “update” or “no update”.

- (a) A process moves from user mode to kernel mode.

Ans: no update

- (b) The OS switches context from one process to another.

Ans: update

8. Consider a process that has just forked a child. The OS implements a copy-on-write fork. At the end of the fork system call, the OS does not perform a context switch and will return back to the user mode of the parent process. Now, which of the following entities are updated at the end of a successful implementation of the fork system call? Answer “update” or “no update”.
- The page table of the parent process.
Ans: update because the parent's pages must be marked read-only.
 - The page table information in the MMU and the TLB.
Ans: update because the parent's pages must be marked read-only.
9. A certain page table entry in the page table of a process has both the valid and present bits set. Describe what happens on a memory access to a virtual address belonging to this page table entry.
- What happens at the TLB? (hit/miss/cannot say)
Ans: cannot say
 - Will a page fault occur? (yes/no/cannot say)
Ans no
10. A certain page table entry in the page table of a process has the valid bit set but the present bit unset. Describe what happens on a memory access to a virtual address belonging to this page table entry.
- What happens at the TLB? (hit/miss/cannot say)
Ans: miss
 - Will a page fault occur? (yes/no/cannot say)
Ans yes
11. Consider the page table entries within the page table of a process that map to kernel code/data stored in RAM, in a Linux-like OS studied in class.
- Are the physical addresses of the kernel code/data stored in the page tables of various processes always the same? (yes/no/cannot say)
Ans: yes, because there is only one copy of kernel code in RAM
 - Does the page table of every process have page table entries pointing to the kernel code/data? (yes/no/cannot say)
Ans: yes, because every process needs to run kernel code in kernel mode
12. Consider a process P running in a Linux-like operating system that implements demand paging. The page/frame size in the system is 4KB. The process has 4 pages in its heap. The process stores an array of 4K integers (size of integer is 4 bytes) in these 4 pages. The process then proceeds to access the integers in the array sequentially. Assume that none of these 4 pages of the heap are initially in physical memory. The memory allocation policy of the OS allocates only 3 physical frames at any point of time, to store these 4 pages of the heap. In case of a page fault and all 3 frames have been allocated to the heap of the process, the OS uses a LRU policy to evict one of these pages to make space for the new page. Approximately what percentage of the 4K accesses to array elements will result in a page fault?

- (a) Almost 100%
- (b) Approximately 25%
- (c) Approximately 75%
- (d) Approximately 0.1%

Ans: (d)

13. Given below are descriptions of different entries in the page table of a process, with respect to which bits are set and which are not set. Accessing which of the page table entries below will always result in the MMU generating a trap to the OS during address translation?

- (a) Page with both valid and present bits set
- (b) Page with valid bit set but present bit unset
- (c) Page with valid bit unset
- (d) Page with valid, present, and dirty bits set

Ans: (b), (c)

14. Which of the following statements is/are true regarding the memory image of a process?

- (a) Memory for non-static local variables of a function is allocated on the heap dynamically at run time
- (b) Memory for arguments to a function is allocated on the stack dynamically at run time
- (c) Memory for static and global variables is allocated on the stack dynamically at run time
- (d) Memory for the argc, argv arguments to the main function is allocated in the code/data section of the executable at compile time

Ans: (b)

15. Consider a process P in a Linux-like operating system that implements demand paging. Which of the following pages in the page table of the process will have the valid bit set but the present bit unset?

- (a) Pages that have been used in the past by the process, but were evicted to swap space by the OS due to memory pressure
- (b) Pages that have been requested by the user using mmap/brk/sbrk system calls, but have not yet been accessed by the user, and hence not allocated physical memory frames by the OS
- (c) Pages corresponding to unused virtual addresses in the virtual address space of the process
- (d) Pages with high virtual addresses mapping to OS code and data

Ans: (a), (b)

16. Consider a process P in a Linux-like operating system that implements demand paging. For a particular page in the page table of this process, the valid and present bits are both set. Which of the following are possible outcomes that can happen when the CPU accesses a virtual address in this page of the process? Select all outcomes that are possible.

- (a) TLB hit (virtual address found in TLB)
- (b) TLB miss (virtual address not found in TLB)
- (c) MMU walks the page table (to translate the address)
- (d) MMU traps to the OS (due to illegal access)

Ans: (a), (b), (c), (d)

17. Consider a process P in a Linux-like operating system that implements demand paging using the LRU page replacement policy. You are told that the i-th page in the page table of the process has the accessed bit set. Which of the following statements is/are true?

- (a) This bit was set by OS when it allocated a physical memory frame to the page
- (b) This bit was set by MMU when the page was accessed in the recent past
- (c) This page is likely to be evicted by the OS page replacement policy in the near future
- (d) This page will always stay in physical memory as long as the process is alive

Ans: (b)

18. Which of the following statements is/are true regarding the functions of the OS and MMU in a modern computer system?

- (a) The OS sets the address of the page table in a CPU register accessible to the MMU every time a new process is created in the system
- (b) The OS sets the address of the page table in a CPU register accessible to the MMU every time a new process is context switched in by the CPU scheduler
- (c) MMU traps to OS every time an address is not found in the TLB cache
- (d) MMU traps to OS every time it cannot translate an address using the page table available to it

Ans: (b), (d)

19. Consider a modern computer system using virtual addressing and translation via MMU. Which of the following statements is/are valid advantages of using virtual addressing as opposed to directly using physical addresses to fetch instructions and data from main memory?

- (a) One does not need to know the actual addresses of instructions and data in main memory when generating compiled executables.
- (b) One can easily provide isolation across processes by limiting the physical memory that is mapped into the virtual address space of a process.
- (c) Using virtual addressing allows us to hide the fact that user's memory is allocated non-contiguously, and helps provide a simplified view to the user.
- (d) Memory access using virtual addressing is faster than directly accessing memory using physical addresses.

Ans: (a), (b), (c)

20. Consider a process running on a system with a 52-bit CPU (i.e., virtual addresses are 52 bits in size). The system has a physical memory of 8GB. The page size in the system is 4KB, and the size of a page table entry is 4 bytes. The OS uses hierarchical paging. Which of the following statements is/are true? You can assume $2^{10} = 1\text{K}$, $2^{20} = 1\text{M}$, and so on.
- (a) We require a 4-level page table to keep track of the virtual address space of a process.
 - (b) We require a 5-level page table to keep track of the virtual address space of a process.
 - (c) The most significant 9 bits are used to index into the outermost page directory by the MMU during address translation.
 - (d) The most significant 40 bits of a virtual address denote the page number, and the least significant 12 bits denote the offset within a page.

Ans: (a), (d)

21. Consider the following line of code in a function of a process.

```
int *x = (int *)malloc(10 * sizeof(int));
```

When this function is invoked and executed:

- (a) Where is the memory for the variable *x* allocated within the memory image of the process? (stack/heap)
Ans: stack
- (b) Where is the memory for the 10 integer variables allocated within the memory image of the process? (stack/heap)
Ans: heap

22. Consider an OS that is not using a copy-on-write implementation for the `fork` system call. A process P has spawned a child C. Consider a virtual address *v* that is translated to physical address $A_p(v)$ using the page table of P, and to $A_c(v)$ using the page table of C.

- (a) For which virtual addresses *v* does the relationship $A_p(v) = A_c(v)$ hold?
Ans: For kernel space addresses, shared libraries and such.
- (b) For which virtual addresses *v* does the relationship $A_p(v) = A_c(v)$ not hold?
Ans: For userspace part of memory image, e.g., code, data, stack, heap.

23. Consider a system with paging-based memory management, whose architecture allows for a 4GB virtual address space for processes. The size of logical pages and physical frames is 4KB. The system has 8GB of physical RAM. The system allows a maximum of 1K (=1024) processes to run concurrently. Assuming the OS uses hierarchical paging, calculate the maximum memory space required to store the page tables of *all* processes in the system. Assume that each page table entry requires an additional 10 bits (beyond the frame number) to store various flags. Assume page table entries are rounded up to the nearest byte. Consider the memory required for both outer and inner page tables in your calculations.

Ans:

Number of physical frames = $2^{33}/2^{12} = 2^{21}$. Each PTE has frame number (21 bits) and flags (10 bits) ≈ 4 bytes. The total number of pages per process is $2^{32}/2^{12}=2^{20}$, so total size of inner page table pages is $2^{20} \times 4 = 4\text{MB}$.

Each page can hold $2^{12}/4 = 2^{10}$ PTEs, so we need $2^{20}/2^{10} = 2^0$ PTEs to point to inner page tables, which will fit in a single outer page table. So the total size of page tables of one process is 4MB + 4KB. For 1K processes, the total memory consumed by page tables is 4GB + 4MB.

24. Consider a simple system running a single process. The size of physical frames and logical pages is 16 bytes. The RAM can hold 3 physical frames. The virtual addresses of the process are 6 bits in size. The program generates the following 20 virtual address references as it runs on the CPU: 0, 1, 20, 2, 20, 21, 32, 31, 0, 60, 0, 0, 16, 1, 17, 18, 32, 31, 0, 61. (Note: the 6-bit addresses are shown in decimal here.) Assume that the physical frames in RAM are initially empty and do not map to any logical page.
- Translate the virtual addresses above to logical page numbers referenced by the process. That is, write down the reference string of 20 page numbers corresponding to the virtual address accesses above. Assume pages are numbered starting from 0, 1, ...
 - Calculate the number of page faults generated by the accesses above, assuming a FIFO page replacement algorithm. You must also correctly point out which page accesses in the reference string shown by you in part (a) are responsible for the page faults.
 - Repeat (b) above for the LRU page replacement algorithm.
 - What would be the lowest number of page faults achievable in this example, assuming an optimal page replacement algorithm were to be used? Repeat (b) above for the optimal algorithm.

Ans:

- For 6 bit virtual addresses, and 4 bit page offsets (page size 16 bytes), the most significant 2 bits of a virtual address will represent the page number. So the reference string is 0, 0, 1, 0, 1, 1, 2, 1, 0, 3 (repeated again).
 - Page faults with FIFO = 8. Page faults on 0, 1, 2, 3 (replaced 0), 0 (replaced 1), 1 (replaced 2), 2 (replaced 3), 3.
 - Page faults with LRU = 6. Page faults on 0, 1, 2, 3 (replaced 2), 2 (replaced 3), 3.
 - The optimum algorithm will replace the page least likely to be used in future, and would look like LRU above.
25. Consider a system with only virtual addresses, but no concept of virtual memory or demand paging. Define *total memory access time* as the time to access code/data from an address in physical memory, including the time to resolve the address (via the TLB or page tables) and the actual physical memory access itself. When a virtual address is resolved by the TLB, experiments on a machine have empirically observed the total memory access time to be (an approximately constant value of) t_h . Similarly, when the virtual address is not in the TLB, the total memory access time is observed to be t_m . If the average total memory access time of the system (averaged across all memory accesses, including TLB hits as well as misses) is observed to be t_x , calculate what fraction of memory addresses are resolved by the TLB. In other words, derive an expression for the TLB hit rate in terms of t_h , t_m , and t_x . You may assume $t_m > t_h$.

Ans: We have $t_x = h * t_h + (1 - h) * t_m$, so $t_h = \frac{t_m - t_x}{t_m - t_h}$

26. 4. Consider a system with a 6 bit virtual address space, and 16 byte pages/frames. The mapping from virtual page numbers to physical frame numbers of a process is (0,8), (1,3), (2,11), and (3,1). Translate the following virtual addresses to physical addresses. Note that all addresses are in decimal. You may write your answer in decimal or binary.

- (a) 20
- (b) 40

Ans:

- (a) $20 = 01\ 0100 = 11\ 0100 = 52$
- (b) $40 = 10\ 1000 = 1011\ 1000 = 184$

27. Consider a system with several running processes. The system is running a modern OS that uses virtual addresses and demand paging. It has been empirically observed that the memory access times in the system under various conditions are: t_1 when the logical memory address is found in TLB cache, t_2 when the address is not in TLB but does not cause a page fault, and t_3 when the address results in a page fault. This memory access time includes all overheads like page fault servicing and logical-to-physical address translation. It has been observed that, on an average, 10% of the logical address accesses result in a page fault. Further, of the remaining virtual address accesses, two-thirds of them can be translated using the TLB cache, while one-third require walking the page tables. Using the information provided above, calculate the average expected memory access time in the system in terms of t_1, t_2 , and t_3 .

Ans: $0.6*t_1 + 0.3*t_2 + 0.1*t_3$

28. Consider a system where each process has a virtual address space of 2^v bytes. The physical address space of the system is 2^p bytes, and the page size is 2^k bytes. The size of each page table entry is 2^e bytes. The system uses hierarchical paging with l levels of page tables, where the page table entries in the last level point to the actual physical pages of the process. Assume $l \geq 2$. Let v_0 denote the number of (most significant) bits of the virtual address that are used as an index into the outermost page table during address translation.

- (a) What is the number of logical pages of a process?
- (b) What is the number of physical frames in the system?
- (c) What is the number of PTEs that can be stored in a page?
- (d) How many pages are required to store the innermost PTEs?
- (e) Derive an expression for l in terms of v, p, k , and e .
- (f) Derive an expression for v_0 in terms of l, v, p, k , and e .

Ans:

- (a) 2^{v-k}
- (b) 2^{p-k}
- (c) 2^{k-e}
- (d) $2^{v-k} / 2^{k-e} = 2^{v+e-2k}$

- (e) The least significant k of v bits indicate offset within a page. Of the remaining $v - k$ bits, $k - e$ bits will be used to index into the page tables at every level, so the number of levels l
 $= \text{ceil } \frac{v-k}{k-e}$
- (f) $v - k - (l - 1) * (k - e)$
29. Consider an operating system that uses 48-bit virtual addresses and 16KB pages. The system uses a hierarchical page table design to store all the page table entries of a process, and each page table entry is 4 bytes in size. What is the total number of pages that are required to store the page table entries of a process, across all levels of the hierarchical page table?
- Ans:** Page size = 2^{14} bytes. So, the number of page table entries = $2^{48}/2^{14} = 2^{34}$. Each page can store $16\text{KB}/4 = 2^{12}$ page table entries. So, the number of innermost pages = $2^{34} / 2^{12} = 2^{22}$.
- Now, pointers to all these innermost pages must be stored in the next level of the page table, so the next level of the page table has $2^{22} / 2^{12} = 2^{10}$ pages. Finally, a single page can store all the 2^{10} page table entries, so the outermost level has one page.
- So, the total number of pages that store page table entries is $2^{22} + 2^{10} + 1$.
30. Consider a memory allocator that uses the buddy allocation algorithm to satisfy memory requests. The allocator starts with a heap of size 4KB (4096 bytes). The following requests are made to the allocator by the user program (all sizes requested are in bytes): $\text{ptr1} = \text{malloc}(500)$; $\text{ptr2} = \text{malloc}(200)$; $\text{ptr3} = \text{malloc}(800)$; $\text{ptr4} = \text{malloc}(1500)$. Assume that the header added by the allocator is less than 10 bytes in size. You can make any assumption about the implementation of the buddy allocation algorithm that is consistent with the description in class.
- (a) Draw a figure showing the status of the heap after these 4 allocations complete. Your figure must show which portions of the heap are assigned and which are free, including the sizes of the various allocated and free blocks.
 - (b) Now, suppose the user program frees up memory allocations of ptr2 , ptr3 , and ptr4 . Draw a figure showing the status of the heap once again, after the memory is freed up and the allocation algorithm has had a chance to do any possible coalescing.
- Ans:**
- (a) [512 B][256 B] 256 B free [1024 B][2048 B]
 - (b) [512 B] 512 B free, 1024 B free, 2048 B free. No further coalescing is possible.
31. Consider a system with 8-bit virtual and physical addresses, and 16 byte pages. A process in this system has 4 logical pages, which are mapped to 3 physical pages in the following manner: logical page 0 maps to physical page 6, 1 maps to 3, 2 maps to 11, and logical page 5 is not mapped to any physical page yet. All the other pages in the virtual address space of the process are marked invalid in the page table. The MMU is given a pointer to this page table for address translation. Further, the MMU has a small TLB cache that stores two entries, for logical pages 0 and 2. For each virtual address shown below, describe what happens when that address is accessed by the CPU. Specifically, you must answer what happens at the TLB (hit or miss?), MMU (which page table entry is accessed?), OS (is there a trap of any kind?), and the physical memory (which physical address is accessed?). You may write the translated physical address in binary format. (Note that it is not implied that the accesses below happen one after the other; you must solve each part of the question independently using the information provided above.)

- (a) Virtual address 7
- (b) Virtual address 20
- (c) Virtual address 70
- (d) Virtual address 80

Ans:

- (a) $7 = 0000$ (page number) + 0111 (offset) = logical page 0. TLB hit. No page table walk. No OS trap. Physical address $0110\ 0111$ is accessed.
 - (b) $20 = 0001\ 0100$ = logical page 1. TLB miss. MMU walks page table. Physical address $0011\ 0100$
 - (c) $70 = 0100\ 0110$ = logical page 4. TLB miss. MMU accesses page table and discovers it is an invalid entry. MMU raises trap to OS.
 - (d) $80 = 0101\ 0000$ = logical page 5. TLB miss. MMU accesses page table and discovers page not present. MMU raises a page fault to the OS.
32. Consider a system with 8-bit addresses and 16-byte pages. A process in this system has 4 logical pages, which are mapped to 3 physical frames in the following manner: logical page 0 maps to physical frame 2, page 1 maps to frame 0, page 2 maps to frame 1, and page 3 is not mapped to any physical frame. The process may not use more than 3 physical frames. On a page fault, the demand paging system uses the LRU policy to evict a page. The MMU has a TLB cache that can store 2 entries. The TLB cache also uses the LRU policy to store the most recently used mappings in cache. Now, the process accesses the following logical addresses in order: 7, 17, 37, 20, 40, 60.
- (a) Out of the 6 memory accesses, how many result in a TLB miss? Clearly indicate the accesses that result in a miss. Assume that the TLB cache is empty before the accesses begin.
Ans: 0,1,2, (miss) 1,2 (hit), 3 (miss)
 - (b) Out of the 6 memory accesses, how many result in a page fault? Clearly indicate the accesses that result in a page fault.
Ans: last access 3 result in a page fault
 - (c) Upon accessing the logical address 60, which physical address is eventually accessed by the system (after servicing any page faults that may arise)? Show suitable calculations.
Ans: $60 = 0011\ 1100$ = page 3. 3 causes page fault, replaces LRU page 0, and mapped to frame 2. So physical address = $0010\ 1100 = 44$
33. Consider a 64-bit system running an OS that uses hierarchical page tables to manage virtual memory. Assume that logical and physical pages are of size 4KB and each page table entry is 4 bytes in size.
- (a) What is the maximum number of levels in the page table of a process, including both the outermost page directory and the innermost page tables?
 - (b) Indicate which bits of the virtual address are used to index into each of the levels of the page table.
 - (c) Calculate the maximum number of pages that may be required to store all the page table entries of a process across all levels of the page table.

Ans

- (a) $\text{ceil}((64 - 12)/(12 - 2)) = 6$
- (b) 2, 10, 10, 10, 10, 10 (starting from most significant to least)
- (c) Innermost level has 2^{52} PTEs, which fit in 2^{42} pages. The next level has 2^{42} PTEs which require 2^{32} pages, and so on. Total pages = $2^{42} + 2^{32} + 2^{22} + 2^{12} + 2^2 + 1$
34. The page size in a system (running a Linux-like operating system on x86 hardware) is increased while keeping everything else (including the total size of main memory) the same. For each of the following metrics below, indicate whether the metric is *generally* expected to increase, decrease, or not change as a result of this increase in page size.
- (a) Size of the page table of a process
 - (b) TLB hit rate
 - (c) Internal fragmentation of main memory
- Ans:** (a) PT size decreases (fewer entries) (b) TLB hit rate increases (more coverage) (c) Internal fragmentation increases (more space wasted in a page)
35. Consider a process with 4 logical pages, numbered 0–3. The page table of the process consists of the following logical page number to physical frame number mappings: (0, 11), (1, 35), (2, 3), (3, 1). The process runs on a system with 16 bit virtual addresses and a page size of 256 bytes. You are given that this process accesses virtual address 770. Answer the following questions, showing suitable calculations.
- (a) Which logical page number does this virtual address correspond to?
 - (b) Which physical address does this virtual address translate to?
- Ans:** (a) $770 = 512 + 256 + 2 = 00000011\ 00000010$ = page 3, offset 2
- (b) page 3 maps to frame 1. physical address = $0000001\ 00000010 = 256 + 2 = 258$
36. Consider a system with 16 bit virtual addresses, 256 byte pages, and 4 byte page table entries. The OS builds a multi-level page table for each process. Calculate the maximum number of pages required to store all levels of the page table of a process in this system.
- Ans:** Number of PTE per process = $2^{16}/2^8 = 2^8$. Number of PTE per page = $2^8/2^2 = 2^6$. Number of inner page table pages = $2^8/2^6 = 4$, which requires one outer page directory. So total pages = $4+1 = 5$.
37. Consider a process with 4 physical pages numbered 0–3. The process accesses pages in the following sequence: 0, 1, 0, 2, 3, 3, 0, 2. Assume that the RAM can hold only 3 out of these 4 pages, is initially empty, and there is no other process executing on the system.
- (a) Assuming the demand paging system is using an LRU replacement policy, how many page faults do the 8 page accesses above generate? Indicate the accesses which cause the faults.
 - (b) What is the minimum number of page faults that would be generated by an optimal page replacement policy? Indicate the accesses which cause the faults.

Ans:

(a) 0 (M), 1 (M), 0(H), 2 (M), 3 (M), 3(H), 0(H), 2(H) = 4 misses

(b) Same as above

38. Consider a Linux-like operating system running on a 48-bit CPU hardware. The OS uses hierarchical paging, with 8 KB pages and 4 byte page table entries.

- (a) What is the maximum number of levels in the page table of a process, including both the outermost page directory and the innermost page tables?

Ans: $\text{ceil } (48 - 13)/(13 - 2) = 4$

- (b) Indicate which bits of the virtual address are used to index into each of the levels of the page table.

Ans: 2, 11, 11, 11

- (c) Calculate the maximum number of pages that may be required to store all the page table entries of a process across all levels of the page table.

Ans: Innermost level has 2^{35} PTEs. Each page can accommodate 2^{11} PTEs. Total pages = $2^{24} + 2^{13} + 2^2 + 1$

39. Consider the scenario described in the previous question. You are told that the OS uses demand paging. That is, the OS allocates a physical frame and a corresponding PTE in the page table only when the memory location is accessed for the first time by a process. Further, the pages at all levels of the hierarchical page table are also allocated on demand, i.e., when there is at least one valid PTE within that page. A process in this system has accessed memory locations in 4K unique pages so far. You may assume that none of these 4K pages has been swapped out yet. You are required to compute the minimum and maximum possible sizes of the page table of this process after all accesses have completed.

- (a) What is the minimum possible size (in pages) of the page table of this process?

Ans: Each page holds 2^{11} PTEs. So 2^{12} pages can be accommodated in 2 pages at the inner most level. Minimum pages in each of the outer levels is 1. So minimum size = $2 + 1 + 1 + 1 = 5$.

- (b) What is the maximum possible size (in pages) of the page table of this process?

Ans: The 2^{12} pages/PTEs could have been widely spread apart and in distinct pages at all levels of the page table. So maximum size = $2^{12} + 2^{12} + 2^2 + 1$.

40. In a demand paging system, it is intuitively expected that increasing the number of physical frames will naturally lead to a reduction in the rate of page faults. However, this intuition does not hold for some page replacement policies. A replacement policy is said to suffer from *Belady's anomaly* if increasing the number of physical frames in the system can sometimes lead to an increase in the number of page faults. Consider two page replacement policies studied in class: FIFO and LRU. For each of these two policies, you must state if the policy can suffer from Belady's anomaly (yes/no). Further, if you answer yes, you must provide an example of the occurrence of the anomaly, where increasing the number of physical frames actually leads to an increase in the number of page faults. If you answer no, you must provide an explanation of why you think the anomaly can never occur with this policy.

Hint: you may consider the following example. A process has 5 logical pages, and accesses them in this order: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5. You may find this scenario useful in finding an example of Belady's anomaly. Of course, you may use any other example as well.

(a) FIFO

Ans: Yes. For string above, 9 faults with 3 frames and 10 faults with 4 frames.

(b) LRU

Ans: No. The N most recently used frames are always a subset of N+1 most recently used frames. So if a page fault occurs with N+1 frames, it must have occurred with N frames also. So page faults with N+1 frames can never be higher.

Lectures on Operating Systems (Mythili Vutukuru, IIT Bombay)

Lab: Dynamic memory management

In this lab, you will understand the principles of memory management by building a custom memory manager to allocate memory dynamically in a program. Specifically, you will implement functions to allocate and free memory, that act as replacements for C library functions like `malloc` and `free`.

Before you begin

Understand how the `mmap` and `munmap` system calls work. In this lab, you will use `mmap` to obtain pages of memory from the OS, and allocate smaller chunks from these pages dynamically when requested. Familiarize yourself with the various arguments to the `mmap` system call.

Warm-up exercises

Do the following warm-up exercises before you start the lab.

1. Write a simple C/C++ program that runs for a long duration, say, by pausing for user input or by sleeping. While the process is active, use the `ps` or any other similar command with suitable options, to measure the memory usage of the process. Specifically, measure the virtual memory size (VSZ) of the process, and the resident set size (RSS) of the process (which includes only the physical RAM pages allocated to the process). You should also be able to see the various pieces of the memory image of the process in the Linux proc file system, by accessing a suitable file in the proc filesystem.
2. Now, add code to your simple program to memory map an empty page from the OS. For this program (and this lab, in general), it makes sense to ask the OS for an anonymous page (since it is not backed by any file on disk) and in private mode (since you are not sharing this page with other processes). Do not do anything else with the memory mapped page. Now, pause your program again and measure the virtual and physical memory consumed by your process. What has changed, and how do you explain it?
3. Finally, write some data into your memory mapped page and measure the virtual and physical memory usage again. Explain what you find.

Part A: Building a simple memory manager

In this part of the lab, you will write code for a memory manager, to allocate and deallocate memory dynamically. Your memory manager must manage 4KB of memory, by requesting a 4KB page via `mmap` from the OS. You must support allocations and deallocations in sizes that are multiples of 8 bytes. The header file `alloc.h` defines the functions you must implement. You must fill in your code in `alloc.c` or `alloc.cpp`. The functions you must implement are described below.

- The function `init_alloc()` must initialize the memory manager, including allocating a 4KB page from the OS via `mmap`, and initializing any other data structures required. This function will be invoked by the user before requesting any memory from your memory manager. This function must return 0 on success and a non-zero error code otherwise.
- The function `cleanup()` must cleanup state of your manager, and return the memory mapped page back to the OS. This function must return 0 on success and a non-zero error code otherwise.
- The function `alloc(int)` takes an integer buffer size that must be allocated, and returns a `char *` pointer to the buffer on a success. This function returns a NULL on failure (e.g., requested size is not a multiple of 8 bytes, or insufficient free space). When successful, the returned pointer should point to a valid memory address within the 4KB page of the memory manager.
- The function `dealloc(char *)` takes a pointer to a previously allocated memory chunk, and frees up the entire chunk.

It is important to note that you must NOT use C library functions like `malloc` to implement the `alloc` function; instead, you must get a page from the OS via `mmap`, and implement a functionality like `malloc` yourself. The memory manager can be implemented in many ways. So feel free to design and implement it in any way you see fit, subject to the following constraints.

- Your memory manager must make the entire 4KB available for allocations to the user via the `alloc` function. That is, you must not store any headers or metadata information within the page itself, that may reduce the amount of usable memory. Any metadata required to keep track of allocation sizes should be within data structures defined in your code, and should not be embedded within the memory mapped 4KB page itself.
- A memory region once allocated should not be available for future allocations until it is freed up by the user. That is, do not double-book your memory, as this can destroy the integrity of the data written into it.
- Once a memory chunk of size N_1 bytes has been deallocated, it must be available for memory allocations of size N_2 in the future, where $N_2 \leq N_1$. Further, if $N_2 < N_1$, the leftover chunk of size $N_1 - N_2$ must be available for future allocations. That is, your memory manager must have the ability to split a bigger free chunk into smaller chunks for allocations.
- If two free memory chunks of size N_1 and N_2 are adjacent to each other, a merged memory chunk of size $N_1 + N_2$ should be available for allocation. That is, you must merge adjacent memory chunks and make them available for allocating a larger chunk.

- After a few allocations and deallocations, your 4KB page may contain allocated and free chunks interspersed with each other. When the next request to allocate a chunk arrives, you may use any heuristic (e.g., best fit, first fit, worst fit, etc.) to allocate a free chunk, as long as the heuristic correctly returns a free chunk if one exists.

We have provided a sample test program `test_alloc.c` to test your implementation. This program runs several tests which initialize your memory manager, and invoke the `alloc` and `dealloc` functions implemented by you. Note that we will be evaluating your code not just with this test program, but with other ones as well. Therefore, feel free to write more such test programs to test your code comprehensively. It is important to note that none of the functionality or data structures required by your memory manager must be embedded within the test program itself. Your entire memory management code should only be contained within `alloc.c`.

You can compile and run the test program using the following commands (use `g++` for C++).

```
$gcc test_alloc.c alloc.c
$./a.out
```

Part B: Expandable heap

In this question, you will build a custom memory allocator over memory mapped pages, much like you did in the previous part of the lab. However, now your memory allocator should be “elastic”, i.e., it should memory map pages from the OS only on demand, as described below. You are given the header file `ealloc.h` that defines 4 functions that your elastic memory allocator should support. You must implement these functions in the file `ealloc.c` or `ealloc.cpp`.

- The function `init_alloc()` should initialize your memory manager. You can initialize any datastructures you may require in this function. However, you must NOT memory map any pages from the OS yet, because you are supposed to allocate memory only on demand.
- The function `cleanup()` should clean up any state of your memory manager. It is NOT required to unmap any pages you memory-mapped from the OS here. We assume in this question that your elastic memory allocator expands by invoking the `mmap` system call when allocations are made, but does not return memory back to the OS via `munmap`.
- The function `alloc(int)` should take an integer buffer size that must be allocated, and must return a `char *` pointer to the buffer on a success. This function should return `NULL` on failure. You can make the following assumptions to simplify the problem. Buffer sizes requested are multiples of 256 bytes, and never longer than 4KB (page size). The total allocated memory will not exceed 4 pages, i.e., 16KB. You need not worry about allocating chunks across page boundaries, i.e., you can assume that every allocated chunk fully resides in one of the 4 pages.

Upon receiving the `alloc` request, your memory allocator should check if it has a free chunk of memory to satisfy this request amongst its existing pages. If not, it must call `mmap` to allocate an anonymous private page from the OS, and use this to satisfy the allocation request. Memory must be requested from the OS on demand, and in the granularity of 4KB pages. However, the allocator should not memory map more pages than required from the OS, and should only request as many pages as required to satisfy the allocation request at hand. For example, suppose your memory allocator has been initialized, and the user of your memory allocator has invoked `alloc(1024)` to

allocate 1024 bytes. Your allocator should make its first `mmap` system call at this point, to memory map one 4KB page only. The next `mmap` system call to allocate a second page must happen only when there is no free space within the first memory mapped page to satisfy a subsequent allocation request. It is very important to note that successive calls to `mmap` may not return contiguous portions of virtual address spaces on all systems. Your code must not rely on this assumption, in order to be portable across systems. Therefore, please do not attempt to allocate chunks that cross page boundaries, and ensure that an allocated chunk is always fully within a page.

- The function `dealloc(char *)` takes a pointer to a previously allocated memory chunk (that was returned by an earlier call to `alloc`), and frees up the entire chunk. There is no requirement for your heap to shrink on deallocations, i.e., you need not ever give back freed up empty pages to the OS via the `munmap` system call.

Requirements of merging and splitting free chunks remain the same as in part A. We have provided a simple test program `test_ealloc.c` to check your implementation. This program performs multiple allocations and deallocations using your custom memory allocator, and checks the sanity of the allocated memory. The test script also checks that you are correctly splitting and merging existing free chunks to satisfy allocation requests. To check that you are only memory mapping pages from the OS on demand, as specified in the problem statement, the program also prints out the virtual memory size (VSZ) of the process periodically. The comments printed out by the test program should help you figure out how the VSZ of your program is expected to grow in a correct implementation.

(Note that we can only check correctness of the values of VSZ and not of the actual physical memory used by your process, because the physical memory allocation is out of your control and is fully handled by the OS demand paging policies.)

Submission instructions

- You must submit the files `alloc.c/alloc.cpp` in part A, and `ealloc.c/ealloc.cpp` in part B. You need not submit the testing code.
- Place these files and any other files you wish to submit in your submission directory, with the directory name being your roll number (say, 12345678).
- Tar and gzip the directory using the command `tar -zcvf 12345678.tar.gz 12345678` to produce a single compressed file of your submission directory. Submit this tar gzipped file on Moodle.

Practice Problems: Processes

1. Answer yes/no, and provide a brief explanation.
 - (a) Can two processes that are not parent/child be concurrently executing the same program executable?
 - (b) Can two running processes share the complete process image in physical memory (not just parts of it)?

Ans:

- (a) Yes, two processes can run the same program executable, which happens when we run the same executable from the terminal twice.
 - (b) No. In general, each process has its own memory image.
2. Consider a process P1 that is executing on a Linux-like OS on a single core system. When P1 is executing, a disk interrupt occurs, causing P1 to go to kernel mode to service that interrupt. The interrupt delivers all the disk blocks that unblock a process P2 (which blocked earlier on the disk read). The interrupt service routine has completed execution fully, and the OS is just about to return back to the user mode of P1. At this point in time, what are the states (ready/running/blocked) of processes P1 and P2?
 - (a) State of P1
 - (b) State of P2

Ans:

- (a) P1 is running (b) P2 is ready
3. Consider a process executing on a CPU. Give an example scenario that can cause the process to undergo:
 - (a) A voluntary context switch.
 - (b) An involuntary context switch.

Ans:

- (a) A blocking system call.
 - (b) Timer interrupt that causes the process to be switched out.
4. Consider a parent process P that has forked a child process C. Now, P terminates while C is still running. Answer yes/no, and provide a brief explanation.

- (a) Will C immediately become a zombie?
- (b) Will P immediately become a zombie, until reaped by its parent?

Ans:

- (a) No, it will be adopted by init.
- (b) Yes.

5. A process in user mode cannot execute certain privileged hardware instructions. [T/F]

Ans: True, some instructions in every CPU's instruction set architecture can only be executed when the CPU is running in a privileged mode (e.g., ring 0 on Intel CPUs).

6. Consider the following CPU instructions found in modern CPU architectures like x86. For each instruction, state if you expect the instruction to be privileged or unprivileged, and justify your answer. For example, if your answer is "privileged", give a one sentence example of what would go wrong if this instruction were to be executable in an unprivileged mode. If your answer is "unprivileged", give a one sentence explanation of why it is safe/necessary to execute the instruction in unprivileged mode.

- (a) Instruction to write into the interrupt descriptor table register.
- (b) Instruction to write into a general purpose CPU register.

Ans:

- (a) Privileged, because a user process may misuse this ability to redirect interrupts of other processes.
- (b) Unprivileged, because this is a harmless operation that is done often by executing processes.

7. Which of the following C library functions do NOT directly correspond to (similarly named) system calls? That is, the implementations of which of these C library functions are NOT straightforward invocations of the underlying system call?

- (a) `system`, which executes a bash shell command.
- (b) `fork`, which creates a new child process.
- (c) `exit`, which terminates the current process.
- (d) `strlen`, which returns the length of a string.

Ans: (a), (d)

8. Which of the following actions by a running process will *always* result in a context switch of the running process, even in a non-preemptive kernel design?

- (a) Servicing a disk interrupt, that results in another blocked process being marked as ready/runnable.
- (b) A blocking system call.
- (c) The system call `exit`, to terminate the current process.
- (d) Servicing a timer interrupt.

Ans: (b), (c)

9. Consider two machines A and B of different architectures, running two different operating systems OS-A and OS-B. Both operating systems are POSIX compliant. The source code of an application that is written to run on machine A must always be rewritten to run on machine B. [T/F]

Ans: False. If the code is written using the POSIX API, it need not be rewritten for another POSIX compliant system.

10. Consider the scenario of the previous question. An application binary that has been compiled for machine A may have to be recompiled to execute correctly on machine B. [T/F]

Ans: True. Even if the code is POSIX compliant, the CPU instructions in the compiled executable are different across different CPU architectures.

11. A process makes a system call to read a packet from the network device, and blocks. The scheduler then context-switches this process out. Is this an example of a voluntary context switch or an involuntary context switch?

Ans: Voluntary context switch.

12. A context switch can occur only after processing a timer interrupt, but not after any other system call or interrupt. [T/F]

Ans: False, a context switch can also occur after a blocking system call for example.

13. A C program cannot directly invoke the OS system calls and must always use the C library for this purpose. [T/F]

Ans: False, it is cumbersome but possible to directly invoke system calls from user code.

14. A process undergoes a context switch every time it enters kernel mode from user mode. [T/F]

Ans: False, after finishing its job in kernel mode, the OS may sometimes decide to go back to the user mode of the same process, without switching to another process.

15. Consider a process P in xv6 that invokes the wait system call. Which of the following statements is/are true?

- (a) If P does not have any zombie children, then the wait system call returns immediately.
- (b) The wait system call always blocks process P and leads to a context switch.
- (c) If P has exactly one child process, and that child has not yet terminated, then the wait system call will cause process P to block.
- (d) If P has two or more zombie children, then the wait system call reaps all the zombie children of P and returns immediately.

Ans: (c)

16. Consider a process P which invokes the default wait system call. For each of the scenarios described below, state the expected behavior of the wait system call, i.e., whether the system call blocks P or if P returns immediately.

- (a) P has no children at all.

- (b) P has one child that is still running.
- (c) P has one child that has terminated and is a zombie.
- (d) P has two children, one of which is running and the other is a terminated zombie.

Ans:

- (a) Does not block
- (b) Blocks
- (c) Does not block
- (d) Does not block

17. Consider the wait family of system calls (wait, waitpid etc.) provided by Linux. A parent process uses some variant of the wait system call to wait for a child that it has forked. Which of the following statements is always true when the parent invokes the system call?
- (a) The parent will always block.
 - (b) The parent will never block.
 - (c) The parent will always block if the child is still running.
 - (d) Whether the parent will block or not will depend on the system call variant and the options with which it is invoked.

Ans: (d)

18. Consider a simple linux shell implementing the command `sleep 100`. Which of the following is an accurate ordered list of system calls invoked by the shell from the time the user enters this command to the time the shell comes back and asks the user for the next input?
- (a) wait-exec-fork
 - (b) exec-wait-fork
 - (c) fork-exec-wait
 - (d) wait-fork-exec

Ans: (c)

19. Which of the following pieces of information in the PCB of a process are changed when the process invokes the exec system call?
- (a) Process identifier (PID)
 - (b) Page table entries
 - (c) The value of the program counter stored within the user space context on the kernel stack

Ans: (a) does not change. (b) and (c) change because the process gets a new memory image (and hence new page table entries pointing to the new image).

20. Which of the following pieces of information about the process are identical for a parent and the newly created child processes, immediately after the completion of the `fork` system call? Answer “identical” or “not identical”.

- (a) The process identifier.
- (b) The contents of the file descriptor table.

Ans: (a) is not identical, as every process has its own unique PID in the system. (b) is identical, as the child gets an exact copy of the parent's file descriptor table.

21. Consider a process P1 that forks P2, P2 forks P3, and P3 forks P4. P1 and P2 continue to execute while P3 terminates. Now, when P4 terminates, which process must wait for and reap P4?

Ans: init (orphan processes are reaped by init)

22. Consider a process P that executes the fork system call twice. That is, it runs code like this:

```
int ret1 = fork(); int ret2 = fork();
```

How many direct children of P (i.e., processes whose parent is P) and how many other descendants of P (i.e., processes who are not direct children of P, but whose grandparent or great grandparent or some such ancestor is P) are created by the above lines of code? You may assume that all fork system calls succeed.

- (a) Two direct children of P are created.
- (b) Four direct children of P are created.
- (c) No other descendant of P is created.
- (d) One other descendant of P is created.

Ans: (a), (d)

23. Consider the x86 instruction “int n” that is executed by the CPU to handle a trap. Which of the following statements is/are true?

- (a) This instruction is always invoked by privileged OS code.
- (b) This instruction causes the CPU to set its EIP to address value n.
- (c) This instruction causes the CPU to lookup the Interrupt Descriptor Table (IDT) using the value n as an index.
- (d) This instruction is always executed by the CPU only in response to interrupts from external hardware, and never due to any code executed by the user.

Ans: (c)

24. Consider the following scheduling policy implemented by an OS, in which a user can set numerical priorities for processes running in the system. The OS scheduler maintains all ready processes in a strict priority queue. When the CPU is free, it extracts the ready process with the highest priority (breaking ties arbitrarily), and runs it until the process blocks or terminates. Which of the following statements is/are true about this scheduling policy?

- (a) This scheduler is an example of a non-preemptive scheduling policy.
- (b) This scheduling policy can result in the starvation of low priority processes.
- (c) This scheduling policy guarantees fairness across all active processes.
- (d) This scheduling policy guarantees lowest average turnaround time for all processes.

Ans: (a), (b)

25. Consider the following scheduling policy implemented by an OS. Every time a process is scheduled, the OS runs the process for a maximum of 10 milliseconds or until the process blocks or terminates itself before 10 milliseconds. Subsequently, the OS moves on to the next ready process in the list of processes in a round-robin fashion. Which of the following statements is/are true about this scheduling policy?
- (a) This policy cannot be efficiently implemented without hardware support for timer interrupts.
 - (b) This scheduler is an example of a non-preemptive scheduling policy.
 - (c) This scheduling policy can sometimes result in involuntary context switches.
 - (d) This scheduling policy prioritizes processes with shorter CPU burst times over processes that run for long durations.

Ans: (a), (c)

26. Consider a process P that needs to save its CPU execution context (values of some CPU registers) on some stack when it makes a function call or system call. Which of the following statements is/are true?
- (a) During a system call, when transitioning from user mode to kernel mode, the context of the process is saved on its kernel stack.
 - (b) During a function call in user mode, the context of the process is saved on its user stack.
 - (c) During a function call in kernel mode, the context of the process is saved on its user stack.
 - (d) During a function call in kernel mode, the context of the process is saved on its kernel stack.

Ans: (a), (b), (d)

27. For each of the events below, state whether the execution context of a running process P will be saved on the user stack of P or on the kernel stack.
- (a) P makes a function call in user mode. **Ans:** user stack
 - (b) P makes a function call in kernel mode. **Ans:** kernel stack
 - (c) P makes a system call and moves from user mode to kernel mode. **Ans:** kernel stack
 - (d) P is switched out by the CPU scheduler. **Ans:** kernel stack
28. Which of the following statements is/are true regarding how the trap instruction (e.g., int n in x86) is invoked when a trap occurs in a system?
- (a) When a user makes a system call, the trap instruction is invoked by the kernel code handling the system call
 - (b) When a user makes a system call, the trap instruction is invoked by userspace code (e.g., user program or a library)
 - (c) When an external I/O device raises an interrupt, the trap instruction is invoked by the device driver handling the interrupt
 - (d) When an external I/O device raises an interrupt signaling the completion of an I/O request, the trap instruction is invoked by the user process that raised the I/O request

Ans: (b)

29. Which of the following statements is/are true about a context switch?

- (a) A context switch from one process to another will happen every time a process moves from user mode to kernel mode
- (b) For preemptive schedulers, a trap of any kind always leads to a context switch
- (c) A context switch will always occur when a process has made a blocking system call, irrespective of whether the scheduler is preemptive or not
- (d) For non-preemptive schedulers, a process that is ready/willing to run will not be context switched out

Ans: (c), (d)

30. When a process makes a system call and runs kernel code:

- (a) How does the process obtain the address of the kernel instruction to jump to?
- (b) Where is the userspace context of the process (program counter and other registers) stored during the transition from user mode to kernel mode?

Ans:

- (a) From IDT (interrupt descriptor table)
- (b) on kernel stack of process (which is linked from the PCB)

31. Which of the following operations by a process will definitely cause the process to move from user mode to kernel mode? Answer yes (if a change in mode happens) or no.

- (a) A process invokes a function in a userspace library.

Ans: no

- (b) A process invokes the `kill` system call to send a signal to another process.

Ans: yes

32. Consider the following events that happen during a context switch from (user mode of) process P to (user mode of) process Q, triggered by a timer interrupt that occurred when P was executing, in a Unix-like operating system design studied in class. Arrange the events in chronological order, starting from the earliest to the latest.

(A) The CPU program counter moves from the kernel address space of P to the kernel address space of Q.

(B) The CPU executing process P moves from user mode to kernel mode.

(C) The CPU stack pointer moves from the kernel stack of P to the kernel stack of Q.

(D) The CPU program counter moves from the kernel address space of Q to the user address space of Q.

(E) The OS scheduler code is invoked.

Ans:

B E C A D

33. Consider a system with two processes P and Q, running a Unix-like operating system as studied in class. Consider the following events that may happen when the OS is concurrently executing P and Q, while also handling interrupts.

- (A) The CPU program counter moves from pointing to kernel code in the kernel mode of process P to kernel code in the kernel mode of process Q.
- (B) The CPU stack pointer moves from the kernel stack of P to the kernel stack of Q.
- (C) The CPU executing process P moves from user mode of P to kernel mode of P.
- (D) The CPU executing process P moves from kernel mode of P to user mode of P.
- (E) The CPU executing process Q moves from the kernel mode of Q to the user mode of Q.
- (F) The interrupt handling code of the OS is invoked.
- (G) The OS scheduler code is invoked.

For each of the two scenarios below, list out the chronological order in which the events above occur. Note that all events need not occur in each question.

- (a) A timer interrupt occurs when P is executing. After processing the interrupt, the OS scheduler decides to return to process P.
- (b) A timer interrupt occurs when P is executing. After processing the interrupt, the OS scheduler decides to context switch to process Q, and the system ends up in the user mode of Q.

Ans:

- (a) C F G D
- (b) C F G B A E

34. Consider the following three processes that arrive in a system at the specified times, along with the duration of their CPU bursts. Process P1 arrives at time $t=0$, and has a CPU burst of 10 time units. P2 arrives at $t=2$, and has a CPU burst of 2 units. P3 arrives at $t=3$, and has a CPU burst of 3 units. Assume that the processes execute only once for the duration of their CPU burst, and terminate immediately. Calculate the time of completion of the three processes under each of the following scheduling policies. For each policy, you must state the completion time of all three processes, P1, P2, and P3. Assume there are no other processes in the scheduler's queue. For the preemptive policies, assume that a running process can be immediately preempted as soon as the new process arrives (if the policy should decide to preempt).

- (a) First Come First Serve
- (b) Shortest Job First (non-preemptive)
- (c) Shortest Remaining Time First (preemptive)
- (d) Round robin (preemptive) with a time slice of (atmost) 5 units per process

Ans:

- (a) FCFS: P1 at 10, P2 at 12, P3 at 15
- (b) SJF: same as above

- (c) SRTF: P2 at 4, P3 at 7, P1 at 15
 (d) RR: P2 at 7, P3 at 10, P1 at 15
35. Consider a system with a single CPU core and three processes A, B, C. Process A arrives at $t = 0$, and runs on the CPU for 10 time units before it finishes. Process B arrives at $t = 6$, and requires an initial CPU time of 3 units, after which it blocks to perform I/O for 3 time units. After returning from I/O wait, it executes for a further 5 units before terminating. Process C arrives at $t = 8$, and runs for 2 units of time on the CPU before terminating. For each of the scheduling policies below, calculate the time of completion of each of the three processes. Recall that only the size of the current CPU burst (excluding the time spent for waiting on I/O) is considered as the “job size” in these schedulers.
- (a) First Come First Serve (non-preemptive).
Ans: A=10, B=21, C=15. A finishes at 10 units. First run of B finishes at 13. C completes at 15. B restarts at 16 and finishes at 21.
- (b) Shortest Job First (non-preemptive)
Ans: A=10, B=23, C=12. A finishes at 10 units. Note that the arrival of shorter jobs B and C does not preempt A. Next, C finishes at 12. First task of B finishes at 15, B blocks from 15 to 18, and finally completes at 23 units.
- (c) Shortest Remaining Time First (preemptive)
Ans: A=15, B=20, C=11. A runs until 6 units. Then the first task of B runs until 9 units. Note that the arrival of C does not preempt B because it has a shorter remaining time. C completes at 11. B is not ready yet, so A runs for another 4 units and completes at 15. Note that the completion of B’s I/O does not preempt A because A’s remaining time is shorter. B finally restarts at 15 and completes at 20.
36. Consider the various CPU scheduling mechanisms and techniques used in modern schedulers.
- (a) Describe one technique by which a scheduler can give higher priority to I/O-bound processes with short CPU bursts over CPU-bound processes with longer CPU bursts, without the user explicitly having to specify the priorities or CPU burst durations to the scheduler.
Ans: Reducing priority of a process that uses up its time slice fully (indicating it is CPU bound)
- (b) Describe one technique by which a scheduler can ensure that high priority processes do not starve low priority processes indefinitely.
Ans: Resetting priority of processes periodically, or round robin.

37. Consider the following C program. Assume there are no syntax errors and the program executes correctly. Assume the fork system calls succeed. What is the output printed to the screen when we execute the below program?

```
void main(argc, argv) {  
  
    for(int i = 0; i < 4; i++) {  
        int ret = fork();  
        if(ret == 0)  
            printf("child %d\n", i);  
    }  
}
```

Ans: The statement “child i” is printed 2^i times for i=0 to 3

38. Consider a parent process P that has forked a child process C in the program below.

```
int a = 5;  
int fd = open(...); //opening a file  
int ret = fork();  
if(ret >0) {  
    close(fd);  
    a = 6;  
    ...  
}  
else if(ret==0) {  
    printf("a=%d\n", a);  
    read(fd, something);  
}
```

After the new process is forked, suppose that the parent process is scheduled first, before the child process. Once the parent resumes after fork, it closes the file descriptor and changes the value of a variable as shown above. Assume that the child process is scheduled for the first time only after the parent completes these two changes.

- What is the value of the variable a as printed in the child process, when it is scheduled next? Explain.
- Will the attempt to read from the file descriptor succeed in the child? Explain.

Ans:

5. The value is only changed in the parent.
- Yes, the file is only closed in the parent.

39. Consider the following pseudocode. Assume all system calls succeed and there are no other errors in the code.

```

int ret1 = fork(); //fork1
int ret2 = fork(); //fork2
int ret3 = fork(); //fork3
wait();
wait();
wait();

```

Let us call the original parent process in this program as P. Draw/describe a family tree of P and all its descendants (children, grand children, and so on) that are spawned during the execution of this program. Your tree should be rooted at P. Show the spawned descendants as nodes in the tree, and connect processes related by the parent-child relationship with an arrow from parent to child. Give names containing a number for descendants, where child processes created by fork "i" above should have numbers like "i1", "i2", and so on. For example, child processes created by fork3 above should have names C31, C32, and so on.

Ans: P has three children, one in each fork statement: C11, C21, C31. C11 has two children in the second and third fork statements: C22, C32. C21 and C22 also have a child each in the third fork statement: C33 and C34.

40. Consider a parent process that has forked a child in the code snippet below.

```

int count = 0;
ret = fork();
if(ret == 0) {
printf("count in child=%d\n", count);
}
else {
count = 1;
}

```

The parent executes the statement "count = 1" before the child executes for the first time. Now, what is the value of count printed by the code above?

Ans: 0 (the child has its own copy of the variable)

41. Consider the following sample code from a simple shell program.

```

command = read_from_user();
int rc = fork();
if(rc == 0) { //child
exec(command);
}
else { //parent
wait();
}

```

Now, suppose the shell wishes the redirect the output of the command not to STDOUT but to a file "foo.txt". Show how you would modify the above code to achieve this output redirection. You can indicate your changes next to the code above.

Ans:

Modify the child code as follows.

```
close( STDOUT_FILENO )
open( "foo.txt" )
exec( command )
```

42. What is the output of the following code snippet? You are given that the `exec` system call in the child does not succeed.

```
int ret = fork();
if(ret==0) {
    exec(some_binary_that_does_not_exec);
    printf(''child\n '');
}
else {
    wait();
    printf(''parent\n '');
}
```

Ans:

```
child
parent
```

43. Consider the following code snippet, where a parent process forks a child process. The child performs one task during its lifetime, while the parent performs two different tasks.

```
int ret = fork();
if(ret == 0) { do_child_task(); }
else { do_parent_task1();
       do_parent_task2(); }
```

With the way the code is written right now, the user has no control over the order in which the parent and child tasks execute, because the scheduling of the processes is done by the OS. Below are given two possible orderings of the tasks that the user wishes to enforce. For each part, briefly describe how you will modify the code given above to ensure the required ordering of tasks. You may write your answer in English or using pseudocode.

Note that you cannot change the OS scheduling mechanism in any way to solve this question. If a process is scheduled by the OS before you want its task to execute, you must use mechanisms like system calls and IPC techniques available to you in userspace to delay the execution of the task till a suitable time.

- (a) We want the parent to start execution of both its tasks only after the child process has finished its task and has terminated.

Ans: Parent does `wait()` until child finishes, and then starts its tasks.

- (b) We want the child process to execute its task after the parent process has finished its first task, but before it runs its second task. The parent must not execute its second task until the child has completed its task and has terminated.

Ans: Many solutions are possible. Parent and child share two pipes (or a socket). Parent writes to one pipe after completing task 1 and child blocks on this pipe read before starting its task. Child writes to pipe 2 after finishing its task, and parent blocks on this pipe read before starting its second task. (Or parent can use wait to block for child termination, like in previous part.)

44. What are the possible outputs printed from this program shown below? You may assume that the program runs on a modern Linux-like OS. You may ignore any output generated from “some_executable”. You must consider all possible scenarios of the system calls succeeding as well as failing. In your answer, clearly list down all the possible scenarios, and the output of the program in each of these scenarios.

```
int ret = fork();
if(ret == 0) {
    printf("Hello1\n");
    exec(''some_executable'');
    printf("Hello2\n");
} else if(ret > 0) {
    wait();
    printf("Hello3\n");
} else {
    printf("Hello4\n");
}
```

Ans: Case I: fork and exec succeed. Hello1, Hello3 are printed. Case II: fork succeeds but exec fails. Hello1, Hello2, Hello3 are printed. Case III: fork fails. Hello4 is printed.

45. Consider the following sample code from a simple shell program.

```
int rc1 = fork();
if(rc1 == 0) {
    exec(cmd1);
}
else {
    int rc2 = fork();
    if(rc2 == 0) {
        exec(cmd2);
    }
    else {
        wait();
        wait();
    }
}
```

- (a) In the code shown above, do the two commands cmd1 and cmd2 execute serially (one after the other) or in parallel? **Ans:** parallel.
- (b) Indicate how you would modify the code above to change the mode of execution from serial to parallel or vice versa. That is, if you answered “serial” in part (a), then you must change the code to execute the commands in parallel, and vice versa. Indicate your changes next to the code snippet above. **Ans:** move the first wait to before second fork.
46. Consider the following code snippet running on a modern Linux operating systems (with a reasonable preemptive scheduling policy as studied in class). Assume that there are no other interfering processes in the system. Note that the executable “good_long_executable” runs for 100 seconds, prints the line “Hello from good executable” to screen, and terminates. On the other hand, the file “bad_executable” does not exist and will cause the exec system call to fail.

```

int ret1 = fork();
if(ret1 == 0) { //Child 1
    printf("Child 1 started\n");
    exec("good_long_executable");
    printf("Child 1 finished\n");
}
else { //Parent
    int ret2 == fork();
    if(ret2 == 0) { //Child 2
        sleep(10); //Sleeping allows child 1 to begin execution
        printf("Child 2 started\n");
        exec("bad_executable");
        printf("Child 2 finished\n");
    } //end of Child 2
    else { //Parent
        wait();
        printf("Child reaped\n");
        wait();
        printf("Parent finished\n");
    }
}
}

```

Write down the output of the above program.

Ans:

Child 1 started
 Child 2 started
 (Some error message from the wrong executable)
 Child 2 finished
 Child reaped
 Hello from good executable
 Parent finished

47. What is the output printed by the following snippet of pseudocode? If you think there is more than one possible answer depending on the execution order of the processes, then you must list all possible outputs.

```
int fd[2];
pipe(fd);
int rc = fork();
if(rc == 0) { //child
    close(fd[1]);
    printf("child1\n");
    read(fd[0], bufc, bufc_size);
    printf("child2\n");
}
else { //parent
    close(fd[0]);
    printf("parent1\n");
    write(fd[1], bufp, bufp_size);
    wait();
    printf("parent2\n");
}
```

Ans: If child scheduled before parent: child1, parent1, child2, parent 2. If parent scheduled before child, parent1, child1, child2, parent2.

48. What is the output of the following program? Only relevant code snippets are shown, and you may assume that the code compiles and executes successfully.

```
int a = 2;
while(a > 0) {
    int ret = fork();
    if(ret == 0) {
        a++;
        printf("a=%d\n", a);
    }
    else {
        wait(NULL);
        a--;
    }
}
```

Ans: The code goes into an infinite while+fork loop, as every child that is forked will execute the same code with a higher value of “a” and hence the while loop never terminates. There will be an infinite number of child processes that will print a=3,4,5, and so on, until either fork fails or the system exhausts some other resource.

49. What is the output of the following program? Only relevant code snippets are shown, and you may assume that the code compiles and executes successfully.

```
int a = 2;
while(a > 0) {
    int ret = fork();
    if(ret == 0) {
        a++;
        printf("a=%d\n", a);
        execl("/bin/ls", "/bin/ls", NULL);
    }
    else {
        wait(NULL);
        a--;
    }
}
```

Ans:

```
a=3
<output of ls>
a=2
<output of ls>
```

50. Consider an application that is composed of one master process and multiple worker processes that are forked off the master at the start of application execution. All processes have access to a pool of shared memory pages, and have permissions to read and write from it. This shared memory region (also called the request buffer) is used as follows: the master process receives incoming requests from clients over the network, and writes the requests into the shared request buffer. The worker processes must read the request from the request buffer, process it, and write the response back into the same region of the buffer. Once the response has been generated, the server must reply back to the client. The server and worker processes are single-threaded, and the server uses event-driven I/O to communicate over the network with the clients (you must not make these processes multi threaded). You may assume that the request and the response are of the same size, and multiple such requests or responses can be accommodated in the request buffer. You may also assume that processing every request takes similar amount of CPU time at the worker threads.

Using this design idea as a starting point, describe the communication and synchronization mechanisms that must be used between the server and worker processes, in order to let the server correctly delegate requests and obtain responses from the worker processes. Your design must ensure that every request placed in the request buffer is processed by one and only one worker thread. You must also ensure that the system is efficient (e.g., no request should be kept waiting if some worker is free) and fair (e.g., all workers share the load almost equally). While you can use any IPC mechanism of your choice, ensure that your system design is practical enough to be implementable in a modern multicore system running an OS like Linux. You need not write any code, and a clear, concise and precise description in English should suffice.

Ans: Several possible solutions exist. The main thing to keep in mind is that the server should be able to assign a certain request in the buffer to a worker, and the worker must be able to notify completion. For example, the master can use pipes or sockets or message queues with each worker. When it places a request in the shared memory, it can send the position of the request to one of the workers. Workers listen for this signal from the master, process the request, write the response, and send a message back to the master that it is done. The master monitors the pipes/sockets of all workers, and assigns the next request once the previous one is done.

Lectures on Operating Systems (Mythili Vutukuru, IIT Bombay)

Lab: Building a Shell

In this lab, you will build a simple shell to execute user commands, much like the `bash` shell in Linux. This lab will deepen your understanding of various concepts of process management in Linux.

Before you begin

- Familiarize yourself with the various process related system calls in Linux: `fork`, `exec`, `exit` and `wait`. The “man pages” in Linux are a good source of learning. You can access the man pages from the Linux terminal by typing `man fork`, `man 2 fork` and so on. You can also find several helpful links online.
- It is important to understand the different variants of these system calls. In particular, there are many different variants of the `exec` and `wait` system calls; you need to understand these to use the correct variant in your code. For example, you may need to invoke `wait` differently depending on whether you need to block for a child to terminate or not.
- Familiarize yourself with simple shell commands in Linux like `echo`, `cat`, `sleep`, `ls`, `ps`, `top`, `grep` and so on. To run these commands from your shell, you must simply “exec” these existing executables, and not implement the functionality yourself.
- Understand the `chdir` system call in Linux (see `man chdir`). This will be useful to implement the `cd` command in your shell.
- Understand the concepts of foreground and background execution in Linux. Execute various commands on the Linux shell both in foreground and background, to understand the behavior of these modes of execution.
- Understand the concept of signals and signal handling in Linux. Understand how processes can send signals to one another using the `kill` system call. Read up on the common signals (SIGINT, SIGTERM, SIGKILL, ..), and how to write custom signal handlers to “catch” signals and override the default signal handling mechanism, using interfaces such as `signal()` or `sigaction()`.
- Understand the notion of processes and process groups. Every process belongs to a process group by default. When a parent forks a child, the child also belongs to the process group of the parent initially. When a signal like Ctrl+C is sent to a process, it is delivered to all processes in its process group, including all its children. If you do not want some subset of children receiving a signal, you may place these children in a separate process group, say, by using the `setpgid` system call. Lookup this system call in the man pages to learn more about how to use it, but here is a simple description. The `setpgid` call takes two arguments: the PID of the process and the process

group ID to move to. If either of these arguments is set to 0, they are substituted by the PID of the process instead. That is, if a process calls `setpgid(0, 0)`, it is placed in a separate process group from its parent, whose process group ID is equal to its PID. Understand such mechanisms to change the process group of a process.

- Read the problem statement fully, and build your shell incrementally, part by part. Test each part thoroughly before adding more code to your shell for the next part.

Warm-up exercises

Below are some “warm-up” exercises you can do before you start implementing the shell.

1. Write a program that forks a child process using the fork system call. The child should print “I am child” and exit. The parent, after forking, must print “I am parent”. The parent must also reap the dead child before exiting. Run this program a few times. What is the order in which the statements are printed? How can you ensure that the parent always prints after the child?
2. Write a program that forks a child process using the fork system call. The child should print its PID and exit. The parent should wait for the child to terminate, reap it, print the PID of the child that it has reaped, and then exit. Explore different variants of the wait system call while you write this program.
3. Write a program that uses the exec system call to run the “ls” command in the program. First, run the command with no arguments. Then, change your program to provide some arguments to “ls”, e.g., “ls -l”. In both cases, running your program should produce the same output as that produced by the “ls” command. There are many variants of the exec system call. Read through the man pages to find something that suits your needs.
4. Write a program that uses the exec system call to run some command. Place a print statement just before and just after the exec statements, and observe which of these statements is printed. Understand the behavior of print statement placed after the exec system call statement in your program by giving different arguments to exec. When is this statement printed and when is it not?
5. Write a program where the fork system call is invoked N times, for different values of N . Let each newly spawned child print its PID before it finishes. Predict the number of child processes that will be spawned for each value of N , and verify your prediction by actually running the code. You must also ensure that all the newly created processes are reaped by using the correct number of wait system calls.
6. Write a program where a process forks a child. The child runs for a long time, say, by calling the sleep function. The parent must use the `kill` system call to terminate the child process, reap it, print a message, and exit. Understand how signals work before you write the program.
7. Write a program that runs an infinite while loop. The program should not terminate when it receives SIGINT (Ctrl+C) signal, but instead, it must print “I will run forever” and continue its execution. You must write a signal handler that handles SIGINT in order to achieve this. So, how do you end this program? When you are done playing with this program, you may need to terminate it using the SIGKILL signal.

Part A: A simple shell

We will first build a simple shell to run simple Linux commands. A shell takes in user input, forks a child process using the `fork` system call, calls `exec` from this child to execute the user command, reaps the dead child using the `wait` system call, and goes back to fetch the next user input. Your shell must execute *any* simple Linux command given as input, e.g., `ls`, `cat`, `echo` and `sleep`. These commands are readily available as executables on Linux, and your shell must simply invoke the corresponding executable, using the user input string as argument to the `exec` system call. It is important to note that you must implement the shell functionality yourself, using the `fork`, `exec`, and `wait` system calls. You must NOT use library functions like `system` which implement shell commands by invoking the Linux shell—doing so defeats the purpose of this assignment!

Your simple shell must use the string “\$ ” as the command prompt. Your shell should interactively accept inputs from the user and execute them. In this part, the shell should continue execution indefinitely until the user hits `Ctrl+C` to terminate the shell. You can assume that the command to run and its arguments are separated by one or more spaces in the input, so that you can “tokenize” the input stream using spaces as the delimiters. You are given starter code `my_shell.c` which reads in user input and “tokenizes” the string for you. For this part, you can assume that the Linux commands are invoked with simple command-line arguments, and without any special modes of execution like background execution, I/O redirection, or pipes. You need not parse any other special characters in the input stream. *Please do not worry about corner cases or overly complicated command inputs for now; just focus on getting the basics right.*

Note that it is not easy to identify if the user has provided incorrect options to the Linux command (unless you can check all possible options of all commands), so you need not worry about checking the arguments to the command, or whether the command exists or not. Your job is to simply invoke `exec` on any command that the user gives as input. If the Linux command execution fails due to incorrect arguments, an error message will be printed on screen by the executable, and your shell must move on to the next command. If the command itself does not exist, then the `exec` system call will fail, and you will need to print an error message on screen, and move on to the next command. In either case, errors must be suitably notified to the user, and you must move on to the next command.

A skeleton code `my_shell.c` is provided to get you started. This program reads input and tokenizes it for you. You must add code to this file to execute the commands found in the “tokens”. You may assume that the input command has no more than 1024 characters, and no more than 64 tokens. Further, you may assume that each token is no longer than 64 characters.

Once you complete the execution of simple commands, proceed to implement support for the `cd` command in your shell using the `chdir` system call. The command `cd <directoryname>` must cause the shell process to change its working directory, and `cd ..` should take you to the parent directory. You need not support other variants of `cd` that are available in the various Linux shells. For example, just typing `cd` will take you to your home directory in some shells; you need not support such complex features. Note that you must NOT spawn a separate child process to execute the `chdir` system call, but must call `chdir` from your shell itself, because calling `chdir` from the child will change the current working directory of the child whereas we wish to change the working directory of the main parent shell itself. Any incorrect format of the `cd` command should result in your shell printing `Shell: Incorrect command` to the display and prompting for the next command.

Your shell must gracefully handle errors. An empty command (typing return) should simply cause the shell to display a prompt again without any error messages. For all incorrect commands or any other erroneous input, the shell itself should not crash. It must simply notify the error and move on to prompt

the user for the next command.

For all commands, you must take care to terminate and carefully reap any child process the shell may have spawned. Please verify this property using the `ps` command during testing. When the forked child calls `exec` to execute a command, the child automatically terminates after the executable completes. However, if the `exec` system call did not succeed for some reason, the shell must ensure that the child is terminated suitably. When not running any command, there should only be the one main shell process running in your system, and no other children.

To test this lab, run a few common Linux commands in your shell, and check that the output matches what you would get on a regular Linux shell. Further, check that your shell is correctly reaping dead children, so that there are no extra zombie processes left behind in the system.

Part B: Background execution

Now, we will extend the shell to support background execution of processes. Extend your shell program of part A in the following manner: if a Linux command is followed by `&`, the command must be executed in the background. That is, the shell must start the execution of the command, and return to prompt the user for the next input, without waiting for the previous command to complete. The output of the command can get printed to the shell as and when it appears. A command not followed by `&` must simply execute in the foreground as before.

You can assume that the commands running in the background are simple Linux commands without pipes or redirections or any other special case handling like `cd`. You can assume that the user will enter only one foreground or background command at a time on the command prompt, and the command and `&` are separated by a space. You may assume that there are no more than 64 background commands executing at any given time. A helpful tip for testing: use long running commands like `sleep` to test your foreground and background implementations, as such commands will give you enough time to run `ps` in another window to check that the commands are executing as specified.

Across both background and foreground execution, ensure that the shell reaps all its children that have terminated. Unlike in the case of foreground execution, the background processes can be reaped with a time delay. For example, the shell may check for dead children periodically, say, when it obtains a new user input from the terminal. When the shell reaps a terminated background process, it must print a message `Shell: Background process finished` to let the user know that a background process has finished.

You must test your implementation for the cases where background and foreground processes are running together, and ensure that dead children are being reaped correctly in such cases. Recall that a generic `wait` system call can reap and return any dead child. So if you are waiting for a foreground process to terminate and invoke `wait`, it may reap and return a terminated background process. In that case, you must not erroneously return to the command prompt for the next command, but you must wait for the foreground command to terminate as well. To avoid such confusions, you may choose to use the `waitpid` variant of this system call, to be sure that you are reaping the correct foreground child. Once again, use long running commands like `sleep`, run `ps` in another window, and monitor the execution of your processes, to thoroughly test your shell with a combination of background and foreground processes. In particular, test that a background process finishing up in the middle of a foreground command execution will not cause your shell to incorrectly return to the command prompt before the foreground command finishes.

Part C: The exit command

Up until now, your shell executes in an infinite loop, and only the signal SIGINT (Ctrl+C) would have caused it to terminate. Now, you must implement the `exit` command that will cause the shell to terminate its infinite loop and exit. When the shell receives the `exit` command, it must terminate all background processes, say, by sending them a signal via the `kill` system call. Obviously, if the shell is receiving the command to exit, it goes without saying that it will not have any active foreground process running. Before exiting, the shell must also clean up any internal state (e.g., free dynamically allocated memory), and terminate in a clean manner.

Part D: Handling the Ctrl+C signal

Up until now, the Ctrl+C command would have caused your shell (and all its children) to terminate. Now, you will modify your shell so that the signal SIGINT does not terminate the shell itself, but only terminates the foreground process it is running. Note that the background processes should remain unaffected by the SIGINT, and must only terminate on the `exit` command. You will accomplish this functionality by writing custom signal handling code in the shell, that catches the Ctrl+C signal and relays it to the relevant processes, without terminating itself.

Note that, by default, any signal like SIGINT will be delivered to the shell and all its children. To solve this part correctly, you must carefully place the various children of the shell in different process groups, say, using the `setpgid` system call. For example, `setpgid(0, 0)` places a process in its own separate process group, that is different from the default process group of its parent. Your shell must do some such manipulation on the process group of its children to ensure that only the foreground child receives the Ctrl+C signal, and the background children in a separate process group do not get killed by the Ctrl+C signal immediately.

Once again, use long running commands like `sleep` to test your implementation of Ctrl+C. You may start multiple long running background processes, then start a foreground command, hit Ctrl+C, and check that only the foreground process is terminated and none of the background processes are terminated.

Part E: Serial and parallel foreground execution

Now, we will extend the shell to support the execution of multiple commands in the foreground, as described below.

- Multiple user commands separated by `&&` should be executed one after the other serially in the foreground. The shell must move on to the next command in the sequence only after the previous one has completed (successfully, or with errors) and the corresponding terminated child reaped by the parent. The shell should return to the command prompt after all the commands in the sequence have finished execution.
- Multiple commands separated by `&&&` should be executed in parallel in the foreground. That is, the shell should start execution of all commands simultaneously, and return to command prompt after all commands have finished execution and all terminated children reaped correctly.

Like in the previous parts of the assignment, you may assume that the commands entered for serial or parallel execution are simple Linux commands, and the user enters only one type of command (serial

or parallel) at a time on the command prompt. You may also assume that there are spaces on either side of the special tokens `&&` and `&&&`. You may assume that there are no more than 64 foreground commands given at a time. Once again, use multiple long running commands like `sleep` to test your series and parallel implementations, as such commands will give you enough time to run `ps` in another window to check that the commands are executing as specified.

The handling of the Ctrl+C signal should terminate all foreground processes running in serial or parallel. When executing multiple commands in serial mode, the shell must terminate the current command, ignore all subsequent commands in the series, and return back to the command prompt. When in parallel mode, the shell must terminate all foreground commands and return to the command prompt.

Submission instructions

- You must submit the shell code `my_shell.c` or `my_shell.cpp`.
- Place this file and any other files you wish to submit in your submission directory, with the directory name being your roll number (say, 12345678).
- Tar and gzip the directory using the command `tar -zcvf 12345678.tar.gz 12345678` to produce a single compressed file of your submission directory. Submit this tar gzipped file on Moodle.

Lectures on Operating Systems (Mythili Vutukuru, IIT Bombay)

Lab: Inter-process communication

In this lab, you will understand how to write programs using various Inter-Process Communication (IPC) mechanisms.

Before you begin

Familiarize yourself with various IPC mechanisms, including shared memory, named pipes, and Unix domain sockets.

Warm-up exercises

Do the following exercises before you begin the lab.

1. You are given two programs that use POSIX shared memory primitives to communicate with each other. The producer program in the file `shm-posix-producer-orig.c` creates a shared memory segment, attaches it to its memory using the `mmap` system call, and writes some text into that segment. You can see the shared memory file under `/dev/shm` after the producer writes to it. The consumer program in `shm-posix-consumer-orig.c` opens the same shared memory segment, reads the text written by the producer, and displays it to the screen. Read, understand and execute both programs to understand how POSIX shared memory works. These examples are from the famous OS textbook by Gagne, Galvin, Silberschatz. A sample run of this code is shown below. Note the library used during compilation.

```
$ gcc -o prod shm-posix-producer-orig.c -lrt
$ gcc -o cons shm-posix-consumer-orig.c -lrt
$ ./prod
$ cat /dev/shm/OS
Studying Operating Systems Is Fun!
$ ./cons
Studying Operating Systems Is Fun!
```

2. You are given two programs that communicate with each other using Unix domain sockets. The server program `socket-server-orig.c` opens a Unix domain socket and waits for messages. The client program `socket-client-orig.c` reads a message from the user, and sends it to the server over the socket. The server then displays it. The programs can be compiled as follows.

```
$ gcc -o client socket-client-orig.c  
$ gcc -o server socket-server-orig.c
```

You must first start the server:

```
$ ./server  
Server ready
```

Then, start the client, type a message, and check that it is displayed at the server.

```
$ ./client  
Please enter the message: hello  
Sending data...
```

3. Write two simple programs that communicate with each other using a named pipe or FIFO. Make one of the processes send a simple message to the other process via the pipe, and let the other process display the message on the screen.

Part A: Sharing strings using shared memory

In this part of the lab, you will write two programs, a producer `shm-posix-producer.c` and consumer `shm-posix-consumer.c`. The producer and consumer share a 4KB shared memory segment. The producer first fills the shared memory segment with 512 copies of the 8-byte string “freeeee” (7 characters plus null termination character) indicating that the shared memory is empty. Then, the producer repeatedly produces 8-byte strings, e.g., “OSisFUN”, and writes them to the shared memory segment. The consumer must read these strings from the shared memory segment, display them to the screen, and “erase” the string from the shared memory segment by replacing them with the free string. The consumer should also sleep for some time (say, 1 second) after consuming each string, in order to digest what it has consumed! The producer and consumer should exchange 1000 strings in this manner. Since there are only 512 slots in the shared memory segment, the producer will have to reuse previously used memory locations that have been consumed and freed up by the consumer as well.

You can use the starter code `shm-posix-producer-orig.c`/`shm-posix-consumer-orig.c` given to you to get started. But note that in the original programs, the shared memory segment is opened only for reading at the consumer, while this part of the lab requires the consumer to write to the shared memory as well when freeing it up. So you will have to change permission flags to the various system calls suitably.

How does the consumer know when and where the producer has written a string to the shared memory? The consumer can constantly keep reading the shared memory segment for a string that is different from the free string pattern, but this is inefficient. Instead, you must open another channel of communication between the producer and consumer, using named pipes or message queues or any other IPC mechanism. Whenever the producer writes a string to the shared memory segment, it sends a message to the consumer specifying the location (you can use byte offset or any other way to encode the location) of the string it has written. The consumer repeatedly reads messages from the producer on this channel, finds out the location of the string it must consume, and then consumes it. You must be careful in ensuring that the consumer reads the exact same number of bytes written by the producer on this channel.

How does the producer know when a string has been consumed, and the corresponding location freed-up, by the consumer? Once again, you can make the producer scan the shared memory segment to find empty slots to produce in, or the consumer can send a message to the producer via some channel to inform it about free slots. This design choice is left up to you, and you can use the inefficient method of scanning for free slots if you desire.

Once you write both programs, test them for a smaller number of iterations (instead of 1000) to check that the shared memory is being used correctly. You may also print out the contents of the shared memory segment for debugging purposes. You must also test your code for varying amounts of sleep time at the consumer. When the sleep time is small or 0, you will see that the producer and consumer finish quickly. However, for longer sleep times, and for more than 512 iterations, you will notice that the producer slows down while waiting for space to be freed up by the consumer. Play around with different sleep times to convince yourself that your code is working correctly.

Part B: File transfer using Unix domain sockets

In this part of the lab, you will write two programs, a client program `socket-client.c` and a server program `socket-server.c` which communicate with each other over Unix domain sockets to transfer a file. The client takes a filename as argument, opens and reads the file in chunks of some size (say, 256 bytes) from disk using open/read system calls, and sends this file data over the socket to the server. The server receives data from the client and displays it on screen. When you run the server in one window, and the client in another, you should see that the content of the file whose name you have given to the client is displayed in the server terminal. Ideally, the programs should also terminate when the file transfer is complete. though this needs a bit of work to achieve, so doing this is optional.

Submission instructions

- You must submit the files `shm-posix-producer.c` and `shm-posix-consumer.c` for part A, and the files `socket-client.c` and `socket-server.c` for part B.
- Place these files and any other files you wish to submit in your submission directory, with the directory name being your roll number (say, 12345678).
- Tar and gzip the directory using the command `tar -zcvf 12345678.tar.gz 12345678` to produce a single compressed file of your submission directory. Submit this tar gzipped file on Moodle.

Lectures on Operating Systems (Mythili Vutukuru, IIT Bombay)

Lab: Synchronization in xv6

The goal of this lab is to understand the concepts of concurrency and synchronization in xv6.

Before you begin

- Download, install, and run the original xv6 OS code. You can use your regular desktop/laptop to run xv6; it runs on an x86 emulator called QEMU that emulates x86 hardware on your local machine. In the xv6 folder, run make, followed by make qemu or make-qemu-nox, to boot xv6 and open a shell.
- We have modified some xv6 files for this lab, and these patched files are provided as part of this lab's code. Before you begin the lab, copy the patched files into the main xv6 code directory. The modified files are as follows.
 - The files `defs.h`, `syscall.c`, `syscall.h`, `sysproc.c`, `user.h` and `usys.S` have been modified to add the new system calls of this lab.
 - The programs `test_counters.c`, `test_toggle.c`, `test_barrier.c`, `test_waitpid.c` and `test_sem.c` are user test programs to test the synchronization primitives you will develop in this lab.
 - The new synchronization primitives you need to implement are declared in `uspinlock.h`, `barrier.h`, and `semaphore.h`. These functions must be implemented in `proc.c`, `uspinlock.c`, and `barrier.c`, `semaphore.c` in the placeholders provided. Some of the header files and test programs may also need to be modified by you.
 - An updated Makefile has been provided, to include all the changes for this lab. You need not modify this file.

Part A: Userspace locks in xv6

In the first part of the lab, we will implement userspace locks in xv6, to provide mutual exclusion in userspace programs. However, xv6 processes do not share any memory, and therefore a need for mutual exclusion does not arise in general. As a proxy for shared memory on which synchronization primitives like locks can be tested, we have defined shared counters within the xv6 kernel, that are available for user space programs.

Within the patched code for this lab, there are NCOUNTER (defined to be 10 in the modified `sysproc.c`) integers that are available to all userspace programs. These counters can all be initialized to 0 with the system call `ucounter_init()`. Further, one can set and get values of a particular shared counter using the following system calls: `ucounter_set(idx, val)` sets the value of the

counter at index `idx` to the value `val` and `ucounter_get(idx)` returns the value of the counter at index `idx`. Note that the value of `idx` should be between 0 and `N COUNTER - 1`.

We will now implement userspace spinlocks to protect access to these shared counters. You must support `NLOCK` (defined to be 10 in `uspinlock.h`) spinlocks for use in userspace programs. You will define a structure of `NLOCK` userspace spinlocks, and implement the following system calls on them. All your code must be in `uspinlock.c`.

- `uspinlock_init()` initializes all the spinlocks to unlocked state.
- `uspinlock_acquire(idx)` will acquire the spinlock at index `idx`. This function must busily spin until the lock is acquired, and return 0 when the lock is successfully acquired.
- `uspinlock_release(idx)` will release the spinlock at index `idx`. This function must return 0 when the lock is successfully released.

While not exposed as a system call, you must also implement the function `uspinlock_holding(idx)`, which must return 0 if the lock is free and 1 if the lock is held. You will find this function useful in implementing part B later. Do not worry about concurrent updates to the lock while this function is executing; it suffices to simply return the lock status.

You must use hardware atomic instructions to implement user level spinlocks, much like the kernel spinlock implementation of xv6. However, the userspace spinlock implementation is expected to be simpler, since you do not need to worry about disabling interrupts and such things with userspace spinlocks. The skeleton code for these system calls has already been provided to you in `sysproc.c`, and you only need to implement the core logic in `uspinlock.c`.

We have provided a test program `test_counters.c` for you to test your mutual exclusion functionality. This program forks a child, and the parent and child increment a shared counter concurrently multiple times. When you run this program in its current form, without any locks, you will see that an incorrect result is being displayed. You must add locks to this program to enable correct updates to the shared counter. After you implement locks and add calls to lock/unlock in the test program, you should be able to see that the shared counter is correctly updated as expected.

Part B: Userspace condition variables in xv6

In this part, you will implement condition variables for use in userspace programs. The following two system calls have been added in the patched code provided to you.

- `ucv_sleep(chan, idx)` puts the process to sleep on the channel `chan`. This call should be invoked with the userspace spinlock at index `idx` held. The code to parse the system call arguments has already been provided to you in `sysproc.c`, and you only need to implement the core logic of this system call in the function `ucv_sleep` defined in `proc.c`.
- `ucv_wakeup(chan)` wakes up all of the processes sleeping on the channel `chan`. This system call has already been implemented for you in `sysproc.c`, by simply invoking the kernel's `wakeup` function.

While the userspace `wakeup` function can directly invoke the kernel's `wakeup` function with a suitable channel argument, the `ucv_sleep` function cannot call the kernel's `sleep` function for the following reason: the kernel `sleep` function is invoked with a kernel spinlock held, while the userspace programs

will invoke sleep on the userspace condition variables with a userspace spinlock held. Therefore, you will need to implement the userspace sleep function yourself, along the lines of the kernel sleep function. Note that the process invoking this function must be put to sleep after releasing the userspace spinlock, and must reacquire the spinlock before returning back after waking up.

We have provided a simple test program `test_toggle.c` to test condition variable functionality. This program spawns a child, and the parent and child both print to the screen. You must use condition variables and modify this program in such a way that the parent and child alternate with each other in printing to the screen (starting with the child). After you correctly implement the `ucv_sleep` function, and add calls to sleep and wakeup to the test program, you should be able to achieve this desired behavior of execution toggling between the parent and child. Of course, you may also write other such test programs to test your condition variable implementation.

Part C: Waitpid system call

Implement the `waitpid` system call in xv6. This system call takes one integer argument: the PID of the child to reap. This system call must return the PID of the said child if it has successfully reaped the child, and must return -1 if the said child does not exist. Once a child is reaped with `waitpid`, the same child should not be returned by a subsequent `wait` system call. You must write your code in the placeholder provided in `proc.c`.

You have been provided a test program `test_waitpid.c` to test your code. The program forks a child and tries to reap it via `waitpid` and `wait`. If `waitpid` functionality is implemented correctly, the child will be reaped by the `waitpid` call and not by the subsequent `wait`.

Part D: Userspace barrier in xv6

In this part, you will implement a barrier in xv6. A barrier works as follows. The barrier is initialized with a count N , using the system call `barrier_init(N)`. Next, processes that wish to wait at the barrier invoke the system call `barrier_check()`. The first $N - 1$ calls to `barrier_check` must block, and the N -th call to this function must unblock all the processes that were waiting at the barrier. That is, all the N processes that wish to synchronize at the barrier must cross the barrier after all N of them have arrived. You must implement the core logic for these system calls in `barrier.c`.

You may assume that the barrier is used only once, so you need not worry about reinitializing it multiple times. You may also assume that all N processes will eventually arrive at the barrier.

You must implement the barrier synchronization logic using the `sleep` and `wakeup` primitives of the xv6 kernel, along with kernel spinlocks. Note that it does not make sense to use the userspace sleep/wakeup functionality or userspace spinlocks developed by you in this part of the assignment, because the code for these system calls must be written within the kernel using kernel synchronization primitives.

We have provided a test program `test_barrier.c` to test your barrier implementation. In this program, a parent and its two children arrive at the barrier at different times. With the skeleton code provided to you, all of them clear the barrier as soon as they enter. However, once you implement the barrier, you will find that the order of the print statements will change to reflect the fact that the processes block until all of them check into the barrier.

Part E: Semaphores in xv6

In this part, you will implement the functionality of semaphores in xv6. You will define an array of ($\text{NSEM} = 10$) semaphores in the kernel code, that are accessible across all user programs. User processes in xv6 will be able to refer to these shared semaphores by an index into the array, and perform up/down operations on them to synchronize with each other. Our patch adds three new system calls to the xv6 kernel:

- `sem_init(index, val)` initializes the semaphore counter at the specified `index` to the value `val`. Note that the index should be within the range [0, $\text{NSEM}-1$].
- `sem_up(index)` will increment the counter value of the semaphore at the specified index by one, and wakeup any one sleeping process.
- `sem_down(index)` will decrement the counter value of the semaphore at the specified index by one. If the value is negative, the process blocks and is context switched out, to be woken up by an up operation on the semaphore at a later point.

The scaffolding to implement these system calls has already been done for you, and you will need to only implement the system call logic in the files `semaphore.h` and `semaphore.c`. You must define the semaphore structure, and the array of NSEM semaphores, in the header file. You must also implement the functions that operate on the semaphores in the C file. Your implementation may need to invoke some variant of the `sleep` or `wakeup` functions of the xv6 kernel. For writing modified `sleep/wakeup` functions in xv6, you will need to modify the files `proc.c`, and `defs.h`.

Once you implement the three system calls, run the test program `test_sem` given to you to verify the correctness of your implementation. The test program spawns two children. When you run the test program before implementing semaphores, you will find that the children print to screen before the parent. However, once the semaphore logic has been correctly implemented by you in `semaphore.c` and `semaphore.h`, you will find that the parent prints to screen before the children, by virtue of the semaphore system calls. Do not modify this test program in any way, but simply use it to test your semaphore implementation.

Submission instructions

- For this lab, you will need to submit modified versions of the following files: `defs.h`, `proc.c`, `uspinlock.c`, `barrier.c`, `test_counters.c`, `test_toggle.c`, `semaphore.h`, `semaphore.c`.
- Place all the files you modified in a directory, with the directory name being your roll number (say, 12345678).
- Tar and gzip the directory using the command `tar -zcvf 12345678.tar.gz 12345678` to produce a single compressed file of your submission directory. Submit this tar gzipped file on Moodle.

Lectures on Operating Systems (Mythili Vutukuru, IIT Bombay)

Lab: Memory management in xv6

The goal of this lab is to understand memory management in xv6.

Before you begin

- Download, install, and run the original xv6 OS code. You can use your regular desktop/laptop to run xv6; it runs on an x86 emulator called QEMU that emulates x86 hardware on your local machine. In the xv6 folder, run `make`, followed by `make qemu` or `make-qemu-nox`, to boot xv6 and open a shell.
- We have modified some xv6 files for this lab, and these patched files are provided as part of this lab's code. Before you begin the lab, copy the patched files into the main xv6 code directory.
- For this lab, you will need to understand the following files: `syscall.c`, `syscall.h`, `sysproc.c`, `user.h`, `usys.S`, `vm.c`, `proc.c`, `trap.c`, `defs.h`, `mmu.h`.
 - The files `sysproc.c`, `syscall.c`, `syscall.h`, `user.h`, `usys.S` link user system calls to system call implementation code in the kernel.
 - `mmu.h` and `defs.h` are header files with various useful definitions pertaining to memory management.
 - The file `vm.c` contains most of the logic for memory management in the xv6 kernel, and `proc.c` contains process-related system call implementations.
 - The file `trap.c` contains trap handling code for all traps including page faults.
 - Understand the implementation of the `sbrk` system call that spans all of these files.
- Learn how to write your own user programs in xv6. For example, if you add a new system call, you may want to write a simple C program that calls the new system call. There are several user programs as part of the xv6 source code, from which you can learn. We have also provided a simple test program `testcase.c` as part of the code for this lab. This test program is compiled by our modified `Makefile` and you can run it on the xv6 shell by typing `testcase` at the command prompt. If you wish to include any other test programs in xv6, remember that the test program should be included in the `Makefile` for it to be compiled and executed from the xv6 shell. Understand how the sample testcase was included within the `Makefile`, and use a similar logic to include other test programs as well. Note that the xv6 OS itself does not have any text editor or compiler support, so you must write and compile the code in your host machine, and then run the executable in the xv6 QEMU emulator.

Part A: Displaying memory information

You will first implement the following new system calls in xv6.

- `numvp()` should return the number of virtual/logical pages in the user part of the address space of the process, up to the program size stored in `struct proc`. You must count the stack guard page as well in your calculations.
- `numpp()` should return the number of physical pages in the user part of the address space of the process. You must count this number by walking the process page table, and counting the number of page table entries that have a valid physical address assigned.

Because xv6 does not use demand paging, you can expect the number of virtual and physical pages to be the same initially. However, the next part of the lab will change this property.

Hint: you can walk the page table of the process by using the `walkpgdir` function which is present in `vm.c`. You can find several examples of how to invoke this function within `vm.c` itself. To compute the number of physical pages in a process, you can write a function that walks the page table of a process in `vm.c`, and invoke this function from the system call handling code.

Part B: Memory mapping with mmap system call

In this part, you will implement a simple version of the `mmap` system call in xv6. Your `mmap` system call should take one argument: the number of bytes to add to the address space of the process. You may assume that the number of bytes is a positive number and is a multiple of page size. The system call should return a value of 0 if any invalid inputs are provided. If a valid number of bytes is provided as input, the system call should expand the virtual address space of the process by the specified number of bytes, and return the starting virtual address of the newly added memory region. The new virtual pages should be added at the end of the current program break, and should increase the program size correspondingly. However, the system call should NOT allocate any physical memory corresponding to the new virtual pages, as we will allocate memory on demand. You can use the system calls of the previous part to print these page counts to verify your implementation. After the `mmap` system call, and before any access to the mapped memory pages, you should only see the number of virtual pages of a process increase, but not the number of physical pages.

Physical memory for a memory-mapped virtual page should be allocated on demand, only when the page is accessed by the user. When the user accesses a memory-mapped page, a page fault will occur, and physical memory should only be allocated as part of the page fault handling. Further, if you memory mapped more than one page, physical memory should only be allocated for those pages that are accessed, and not for all pages in the memory-mapped region. Once again, use the virtual/physical page counts to verify that physical pages are allocated only on demand.

We have provided a simple test program to test your implementation. This program invokes `mmap` multiple times, and accesses the memory-mapped pages. It prints out virtual and physical page counts periodically, to let you check whether the page counts are being updated correctly. You can write more such test cases to thoroughly test your implementation.

Some helpful hints for you to solve this assignment are given below.

- Understand the implementation of the `sbrk` system call. Your `mmap` system call will follow a similar logic. In `sbrk`, the virtual address space is increased and physical memory is allocated

within the same system call. The implementation of `sbrk` invokes the `growproc` function, which in turn invokes the `allocuvm` function to allocate physical memory. For your `mmap` implementation, you must only grow the virtual address space within the system call implementation, and physical memory must be allocated during the page fault. You may invoke `allocuvm` (or write another similar function) in order to allocate physical memory upon a page fault.

- The original version of xv6 does not handle the page fault trap. For this assignment, you must write extra code to handle the page fault trap in `trap.c`, which will allocate memory on demand for the page that has caused the page fault. You can check whether a trap is a page fault by checking if `tf->trapno` is equal to `T_PGFLT`. Look at the arguments to the `printf` statements in `trap.c` to figure out how one can find the virtual address that caused the page fault. Use `PGROUNDOWN(va)` to round the faulting virtual address down to the start of a page boundary. Once you write code to handle the page fault, do break or return in order to avoid the processing of other traps.
- Remember: it is important to call `switchuvm` to update the CR3 register and TLB every time you change the page table of the process. This update to the page table will enable the process to resume execution when you handle the page fault correctly.

Part C: Copy-on-Write Fork

In this part, you will implement the copy-on-write (CoW) variant of the `fork()` system call. Please begin this part with a clean installation of the original xv6 code.

You will begin by adding a new system call to xv6. The system call `getNumFreePages()` should return the total number of free pages in the system. This system call will help you see when pages are consumed, and can help you debug your CoW implementation. You must add code to maintain and track freepages in `kalloc.c`, and access this information when this system call is invoked.

Next, you will start the copy-on-write fork implementation. The current implementation of the fork system call in xv6 makes a complete copy of the parent's memory image for the child. On the other hand, a copy-on-write (CoW) fork will let both parent and child use the same memory image initially, and make a copy only when either of them wants to modify any page of the memory image. We will implement CoW fork in the following steps.

1. Begin with changes to `kalloc.c`. To correctly implement CoW fork, you must track reference counts of memory pages. A reference count of a page should indicate the number of processes that map the page into their virtual address space. The reference count of a page is set to one when a freepage is allocated for use by some process. Whenever an additional process points to an already existing page (e.g., when parent forks a child and both share the same memory page), the reference count must be incremented. The reference count must be decremented when a process no longer points to the page from its page table. A page can be freed up and returned to the freelist only when there are no active references to it, i.e., when its reference count is zero. You must add a datastructure to keep track of reference counts of pages in `kalloc.c`. You must also add code to increment and decrement these reference counts, with suitable locking.
2. Understand the various definitions and macros in `mmu.h`, e.g., to extract the page number from a virtual address. Feel free to add more macros here if required.

3. The main change to the `fork` system call to make it CoW fork will happen in the function `copyuvm` in `vm.c`. When you fork a child, you must not make a copy of the parent's pages for the child. Instead, the child should get a new page table, and the page tables of the parent and the child should both point to the same physical pages. This function is one place where you may have to invoke code in `kalloc.c` to increment the reference count of a kernel page, because multiple page tables will map the same physical page.
4. Further, when the parent and child are made to share the pages of the memory image as described above, these pages must be marked read-only, so that any write access to them traps to the kernel. Now, given that the parent's page table has changed (with respect to page permissions), you must reinstall the page table and flush TLB entries by republishing the page table pointer in the CR3 register. This can be accomplished by invoking the function `lcr3(v2p(pgd))` provided by `xv6`. Note that `xv6` already does this TLB flush when switching context and address spaces, but you may have to do it additionally in your code when you modify any page table entries as part of your CoW implementation.
5. Once you have changed the fork implementation as described above, both parent and child will execute over the same read-only memory image. Now, when the parent or child processes attempt to write to a page marked read-only, a page fault occurs. The trap handling code in `xv6` does not currently handle the `T_PGFLT` exception (that is defined already, but not caught). You must write a trap handler to handle page faults in `trap.c`. You can simply print an error message initially, but eventually this trap handling code must call the function that makes a copy of user memory.
6. The bulk of your changes will be in this new function you will write to handle page faults. This function can be written in `vm.c` and can be invoked from the page fault handling code in `trap.c`, because you cannot easily invoke certain static functions like `mappages` from `trap.c`. When a page fault occurs, the CR2 register holds the faulting virtual address, which you can get using the `xv6` function call `rcr2()`. You must now look at this virtual address and decide what must be done about this page fault. If this address is in an illegal range of virtual addresses that are not mapped in the page table of the process, you must print an error message and kill the process. Otherwise, if this trap was generated due to the CoW pages that were marked as read-only, you must proceed to make copies of the pages as needed.
7. Note that between the parent and the child processes, any process that attempts to write to the read-only memory image (whether parent or child) will trap to the kernel. At this stage, you must allocate a new page and copy its contents from the original page pointed to by the virtual address. However, you must make copies carefully. If N processes share a page, the first $N - 1$ processes that trap should receive a separate copy of the page in this fashion. After the $N - 1$ copies are made, the last process that traps is the only one that points to this page (as indicated by the reference count on the page). Therefore, this last process can simply remove the read-only restriction on its page and continue to use the original page. Make sure you modify the reference counts correctly, e.g., decrement the count when a process no longer points to a page by virtue of getting its own copy. Also remember to flush the TLB whenever you change page table entries.
8. Finally, think about how you will test the correctness of your CoW fork. Write test programs that print various statistics like the number of free pages in the system, and see how these statistics change, to test the correctness of your code. We have not provided any test cases and you can write your own.

Submission instructions

- For this lab, you may need to modify some subset of the following files: `syscall.c`, `syscall.h`, `sysproc.c`, `user.h`, `usys.S`, `vm.c`, `proc.c`, `trap.c`, `defs.h`. You may also write new test cases, and modify the `Makefile` to compile additional test cases.
- Place all the files you modified in a directory, with the directory name being your roll number (say, 12345678).
- Tar and gzip the directory using the command `tar -zcvf 12345678.tar.gz 12345678` to produce a single compressed file of your submission directory. Submit this tar gzipped file on Moodle.

Lectures on Operating Systems (Mythili Vutukuru, IIT Bombay)

Practice Problems: xv6 Filesystem

1. Consider two active processes in xv6 that are connected by a pipe. Process W is writing to the pipe continuously, and process R is reading from the pipe. While these processes are alternately running on the single CPU of the machine, no other user process is scheduled on the CPU. Also assume that these processes always give up CPU due to blocking on a full/empty pipe buffer rather than due to yielding on a timer interrupt. List the sequence of events that occur from the time W starts execution, to the time the context is switched to R, to the time it comes back to W again. You must list all calls to the functions `sleep` and `wakeup`, and calls to `sched` to give up CPU; clearly state which process makes these calls from which function. Further, you must also list all instances of acquiring and releasing `ptable.lock`. Make your description of the execution sequence as clear and concise as possible.

Ans:

- (a) W calls `wakeup` to mark R as ready.
 - (b) W realizes the pipe buffer is full and calls `sleep`.
 - (c) W acquires `ptable.lock`.
 - (d) W gives up the CPU in `sched`.
 - (e) The OS performs a context switch from W to R and R starts execution.
 - (f) R releases `ptable.lock`.
 - (g) R calls `wakeup` to mark W as ready.
 - (h) R realizes the pipe buffer is full and calls `sleep`.
 - (i) R gives up the CPU in `sched`.
 - (j) The OS performs a context switch from R to W and W starts execution
2. State one advantage of the disk buffer cache layer in xv6 besides caching. Put another way, even if there was zero cache locality, the higher layers of the xv6 file system would still have to use the buffer cache: state one functionality of the disk buffer cache (besides caching) that is crucial for the higher layers.
- Ans:** Synchronization - only one process at a time handles a disk block.
3. Consider two processes in xv6 that both wish to read a particular disk block, i.e., either process does not intend to modify the data in the block. The first process obtains a pointer to the struct `buf` using the function “`bread`”, but never causes the buffer to become dirty. Now, if the second process calls “`bread`” on the same block before the first process calls “`brelse`”, will this second call to “`bread`” return immediately, or would it block? Briefly describe what xv6 does in this case, and justify the design choice.

Ans: Second call to bread would block. Buffer cache only allows access to one block at a time, since the buffer cache has no control on how the process may modify the data

4. Consider a process that calls the log_write function in xv6 to log a changed disk block. Does this function block the invoking process (i.e., cause the invoking process to sleep) until the changed block is written to disk? Answer Yes/No.

Ans: No

5. Repeat the previous question for when a process calls the bwrite function to write a changed buffer cache block to disk. Answer Yes/No.

Ans: Yes

6. When the buffer cache in xv6 runs out of slots in the cache in the bget function, it looks for a clean LRU block to evict, to make space for the new incoming block. What would break in xv6 if the buffer cache implementation also evicted dirty blocks (by directly writing them to their original location on disk using the bwrite function) to make space for new blocks?

Ans: All writes must happen via logging for consistent updates to disk blocks during system calls. Writing dirty blocks to disk bypassing the log will break this property.

7. (a) Recall that buffer caches of operating systems come in two flavors when it comes to writing dirty blocks from the cache to the secondary storage disk: write through caches and write back caches. Consider the buffer cache implementation in xv6, specifically the bwrite function. Is this implementation an example of a write through cache or a write back cache? Explain your answer.
(b) If the xv6 buffer cache implementation changed from one mode to the other, give an example of xv6 code that would break, and describe how you would fix it. In other words, if you answered “write through” to part (a) above, you must explain what would go wrong (and how you would fix it) if xv6 moved to a write back buffer cache implementation. And if you answered “write back” to part (a), explain what would need to change if the buffer cache was modified to be write through instead.
(c) The buffer cache in xv6 maintains all the `struct buf` buffers in a fixed-size array. However, an additional linked list structure is imposed on these buffers. For example, each `struct buf` also has pointers `struct buf *prev` and `struct buf *next`. What additional functions do these pointers serve, given that the buffers can all be accessed via the array anyway?

Ans:

- (a) Write through cache
(b) If changed to write back, the logging mechanism would break.
(c) Helps implement LRU eviction
8. Consider a system running xv6. A process has the three standard file descriptors (0,1,2) open and pointing to the console. All the other file descriptors in its `struct proc` file descriptor table are unused. Now the process runs the following snippet of code to open a file (that exists on disk, but has not been opened before), and does a few other things as shown below. Draw a figure showing the file descriptor table of the process, relevant entries in the global open file table, and

the in-memory inode structures pointed at by the file table, after each point in the code marked parts (a), (b), and (c) below. Your figures must be clear enough to understand what happens to the kernel data structures right after these lines execute. You must draw three separate figures for parts (a), (b), and (c).

```
int fd;  
fd = open("foo.txt", O_RDWR); //part (a)  
dup(fd); //part (b)  
fd = open("foo.txt", O_RDWR); //part (c)
```

Ans:

- (a) Open creates new FD, open file table entry, and allocated new inode.
 - (b) Dup creates new FD to point to same open file table entry.
 - (c) Next open creates new open file table entry to point to the same inode.
9. Consider the execution of the system call `open` in xv6, to create and open a new file that does not already exist.
- (a) Describe all the changes to the disk and memory data structures that happen during the process of creating and opening the file. Write your answer as a bulleted list; each item of the list must describe one data structure, specify whether it is in disk or memory, and briefly describe the change that is made to this data structure by the end of the execution of the open system call.
 - (b) Suggest a suitable ordering of the changes above that is most resilient to inconsistencies caused by system crashes. Note that the ordering is not important in xv6 due to the presence of a logging mechanism, but you must suggest an order that makes sense for operating systems without such logging features.

Ans:

- (a)
 - A free inode on disk is marked as allocated for this file.
 - An inode from the in-memory inode cache is allocated to hold data for this new inode number.
 - A directory entry is written into the parent directory on disk, to point to the new inode.
 - An entry is created in the in-memory open file table to point to the inode in cache.
 - An entry is added to the in-memory per-process file descriptor table to point to the open file table entry.
 - (b) The directory entry should be added after the on-disk inode is marked as free. The memory operations can happen in any order, as the memory data structures will not survive a crash.
10. Consider the operation of adding a (hard) link to an existing file /D1/F1 from another location /D2/F2 in the xv6 OS. That is, the linking process should ensure that accessing /D2/F2 is equivalent to accessing /D1/F1 on the system. Assume that the contents of all directories and files fit within one data block each. Let $i(x)$ denote the block number of the inode of a file/directory, and let $d(x)$ denote the block number of the (only) data block of a file/directory. Let L denote the starting

block number of the log. Block L itself holds the log header, while blocks starting $L + 1$ onwards hold data blocks logged to the disk.

Assume that the buffer cache is initially empty, except for the inode and data (directory entries) of the root directory. Assume that no other file system calls are happening concurrently. Assume that a transaction is started at the beginning of the link system call, and commits right after the end of it. Make any other reasonable assumptions you need to, and list them down.

Now, list and explain all the read/write operations that the disk (not the buffer cache) sees during the execution of this link operation. Write your answer as a bulleted list. Each bullet must specify whether the operation is a read or write, the block number of the disk request, and a brief explanation on why this request to disk happens. Your answer must span the entire time period from the start of the system call to the end of the log commit process.

Ans:

- read $i(D1)$, read $d(D1)$ —this will give us the inode number of $F1$.
- read $i(F1)$. After reading the inode and bringing it to cache, its link count will be updated. At this point, the inode is only updated in the buffer cache.
- read $i(D2)$, read $d(D2)$ —we check that the new file name $F2$ does not exist in the directory. After this, the directory contents are updated, and a note is made in the log.
- Now, the log starts committing. This transaction has two modified blocks (the inode of $F1$ and the directory content of $D2$). So we will see two disk blocks written to the log: write to $L+1$ and $L+2$, followed by a write to block L (the log header).
- Next, the transactions are installed: a write to disk blocks $i(F1)$ and $d(D2)$.
- Finally, another write to block L to clear the transaction header.

11. Which of the following statements is/are true regarding the file descriptor (FD) layer in xv6?
- A. The FD returned by `open` is an index to the global `struct ftable`.
 - B. The FD returned by `open` is an index to the open file table that is part of the `struct proc` of the process invoking `open`.
 - C. Each entry in the global `struct ftable` can point to either an in-memory inode object or a pipe object, but never to both.
 - D. The reference count stored in an entry of the `struct ftable` indicates the number of links to the file in the directory tree.

Ans: BC

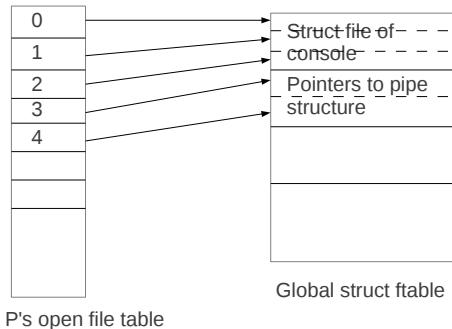
12. Consider the following snippet of the shell code from xv6 that implements pipes.

```

pcmd = (struct pipecmd*) cmd;
if(pipe(p) < 0)
    panic("pipe");
if(fork1() == 0){
close(1);
dup(p[1]); //part (a)
close(p[0]);
close(p[1]);
runcmd(pcmd->left);
}
if(fork1() == 0){
close(0);
dup(p[0]);
close(p[0]);
close(p[1]);
runcmd(pcmd->right);
}
close(p[0]);
close(p[1]); //part (b)
wait();
wait();

```

Assume the shell gets the command “echo hello | grep hello”. Let P denote the parent shell process that implements this command, and let CL and CR denote the child processes created to execute the left and right commands of the above pipe command respectively. Assume P has no other file descriptors open, except the standard input/output/error pointing to the console. Below are shown the various (global and per-process) open file tables right after P executes the `pipe` system call, but before it forks CL and CR.

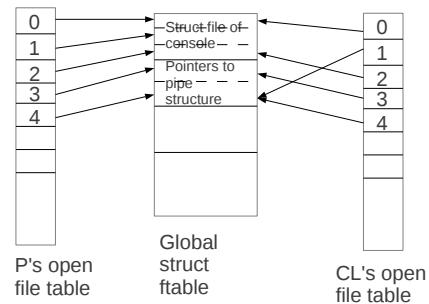


Draw similar figures showing the states of the various open file tables after the execution of lines marked (a) and (b) above. For each of parts (a) and (b), you must clearly draw both the global and per-process file tables, and illustrate the various pointers clearly. You may assume that all created child processes are still running by the time the line marked part (b) above is executed. You may

also assume that the scheduler switches to the child right after fork, so that the line marked part (a) in the child runs before the line marked part (b) in the parent.

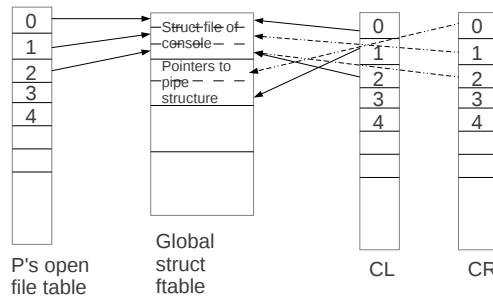
Ans:

- (a) The fd 1 of the child process has been modified to point to one of the pipe's file descriptors



by the close and dup operations.

- (b) CL and CR are now connected to either ends of the pipe. The parent has no pointers to the



pipe any more.

13. Consider a simple xv6-like filesystem, with the following deviations from the xv6 design. The filesystem uses a simple inode structure with only direct pointers (and no indirect blocks) to track file data blocks. Unlike xv6, this filesystem does not use logging or any other mechanism to guarantee crash consistency. The disk buffer cache follows the write-back design paradigm. Much like xv6, free data blocks and free inodes on disk are tracked via bitmaps. A process on this system performed a `write` system call on an open file, to add a new block of data at the end of the file. The successful execution of this system call changed several data and metadata blocks in the disk buffer cache. However, a power failure occurs before these changed blocks can be flushed to disk, causing changes to some disk blocks to be lost before being propagated to disk.

(a) Consider the following scenario after the crash. The user reboots the system, and opens the file he wrote to just before the crash. From the file size on disk, it appears that his last write completed successfully. However, upon reading back the data he had written in the last block, he finds the data to be incorrect (garbage). Can you explain how this scenario can occur? Specifically, can you state which changed blocks pertaining to this system call were propagated to disk, and which weren't?

Blocks correctly changed on disk:

Blocks whose changes were lost during the crash:

(b) Repeat the above question when the user finds himself in the following scenario. Immediately after reboot, the user finds that the data he had written to the file just before the crash did survive the crash, and he is able to read back the correct data from the file. However, after a few hours of using his system, he finds that the file now has incorrect data in the last block that he wrote before the crash.

Blocks correctly changed on disk:

Blocks whose changes were lost during the crash:

(c) Now, suppose the filesystem implementation is changed in such a way that, for any system call, changes to data blocks are always written to the disk before changes to metadata blocks. Which of the two scenarios of parts (a) and (b) could still have occurred after this change to the filesystem, and which could have been prevented?

Scenario(s) that still occur:

Scenario(s) prevented:

(d) Now, suppose the filesystem comes with a filesystem checker tool like `fsck`, which checks the consistency of filesystem metadata after a crash, and fixes any inconsistencies it finds. If this tool is run on the filesystem after the crash, which of the two scenarios of parts (a) and (b) can still happen, and which could have been prevented?

Scenario(s) that still occur:

Scenario(s) prevented:

Ans: (a) inode and bitmap changed, data block change lost (b) inode and data block changed, bitmap change lost (causing block to be reused) (c) part a wont happen, b will still occur (d) part a can still happen, b wont occur

14. Which of the following system calls in xv6, when executed successfully, always result in a new entry being added to the open file table?

- (A) `open` (B) `dup` (C) `pipe` (D) `fork`

Ans: AC

15. Consider an in-memory inode pointer `struct inode *ip` in xv6 that is returned via a call to `iget`. Which of the following statements about this inode pointer is/are true?
- (A) The pointer returned from `iget` is an exclusive pointer, and another process that requests a pointer to the same inode will block until the previous process releases it via `iput`
 - (B) The pointer returned by `iget` is non-exclusive, and multiple processes can obtain a pointer to the same inode via calls to `iget`
 - (C) The contents of the inode pointer returned by `iget` are always guaranteed to be correct, and consistent with the information in the on-disk inode
 - (D) The reference count of the inode returned by `iget` is always non-zero, i.e., `ip->ref > 0`

Ans: BD

16. When is an inode marked as free on the disk in xv6?
- (A) As soon as its link count hits 0, even if the reference count of the in-memory inode is non-zero
 - (B) As soon as the reference count of the in-memory inode hits 0, even if the link count of the inode is non-zero
 - (C) When both the link count and reference count of the in-memory inode are 0
 - (D) Cannot say; the answer depends on the exact sequence of system calls executed

Ans: C

Practice Problems: Synchronization in xv6

1. Modern operating systems disable interrupts on specific cores when they need to turn off preemption, e.g., when holding a spin lock. For example, in xv6, interrupts can be disabled by a function call `cli()`, and reenabled with a function call `sti()`. However, functions that need to disable and enable interrupts do not directly call the `cli()` and `sti()` functions. Instead, the xv6 kernel disables interrupts (e.g., while acquiring a spin lock) by calling the function `pushcli()`. This function calls `cli()`, but also maintains a count of how many push calls have been made so far. Code that wishes to enable interrupts (e.g., when releasing a spin lock) calls `popcli()`. This function decrements the above push count, and enables interrupts using `sti()` only after the count has reached zero. That is, it would take two calls to `popcli()` to undo the effect of two `pushcli()` calls and restore interrupts. Provide one reason why modern operating systems use this method to disable/enable interrupts, instead of directly calling the `cli()` and `sti()` functions. In other words, explain what would go wrong if every call to `pushcli()` and `popcli()` in xv6 were to be replaced by calls to `cli()` and `sti()` respectively.

Ans: If one acquires multiple spinlocks (say, while serving nested interrupts, or for some other reason), interrupts should be enabled only after locks have been released. Therefore, the push and pop operations capture how many times interrupts have been disabled, so that interrupts can be reenabled only after all such operations have been completed.

2. Consider an operating system where the list of process control blocks is stored as a linked list sorted by pid. The implementation of the wakeup function (to wake up a process waiting on a condition) looks over the list of processes in order (starting from the lowest pid), and wakes up the first process that it finds to be waiting on the condition. Does this method of waking up a sleeping process guarantee bounded wait time for every sleeping process? If yes, explain why. If not, describe how you would modify the implementation of the wakeup function to guarantee bounded wait.

Ans: No, this design can have starvation. To fix it, keep a pointer to where the wakeup function stopped last time, and continue from there on the next call to wakeup.

3. Consider an operating system that does not provide the `wait` system call for parent processes to reap dead children. In such an operating system, describe one possible way in which the memory allocated to a terminated process can be reclaimed correctly. That is, identify one possible place in the kernel where you would put the code to reclaim the memory.

Ans: One possible place is the scheduler code itself: while going over the list of processes, it can identify and clean up zombies. Note that the cleanup cannot happen in the exit code itself, as the process memory must be around till it invokes the scheduler.

4. Consider a process that invokes the `sleep` function in xv6. The process calling `sleep` provides a lock `lk` as an argument, which is the lock used by the process to protect the atomicity of its call to `sleep`. Any process that wishes to call `wakeup` will also acquire this lock `lk`, thus avoiding a call to `wakeup` executing concurrently with the call to `sleep`. Assume that this lock `lk` is not `ptable.lock`. Now, if you recall the implementation of the `sleep` function, the lock `lk` is released before the process invokes the scheduler to relinquish the CPU. Given this fact, explain what prevents another process from running the `wakeup` function, while the first process is still executing `sleep`, after it has given up the lock `lk` but before its call to the scheduler, thus breaking the atomicity of the `sleep` operation. In other words, explain why this design of xv6 that releases `lk` before giving up the CPU is still correct.

Ans: Sleep continues to hold `ptable.lock` even after releasing the lock it was given. And `wakeup` requires `ptable.lock`. Therefore, `wakeup` cannot execute concurrently with `sleep`.

5. Consider the `yield` function in xv6, that is called by the process that wishes to give up the CPU after a timer interrupt. The `yield` function first locks the global lock protecting the process table (`ptable.lock`), before marking itself as `RUNNABLE` and invoking the scheduler. Describe what would go wrong if `yield` locked `ptable.lock` AFTER setting its state to `RUNNABLE`, but before giving up the CPU.

Ans: If marked runnable, another CPU could find this process runnable and start executing it. One process cannot run on two cores in parallel.

6. Provide one reason why a newly created process in xv6, running for the first time, starts its execution in the function `forkret`, and not in the function `trapret`, given that the function `forkret` almost immediately returns to `trapret`. In other words, explain the most important thing a newly created process must do before it pops the trap frame and executes the return from the trap in `trapret`.

Ans: It releases `ptable.lock` and preserves the atomicity of the context switch.

7. Consider a process P in xv6 that acquires a spinlock L, and then calls the function `sleep`, providing the lock L as an argument to `sleep`. Under which condition(s) will lock L be released *before* P gives up the CPU and blocks?

- (a) Only if L is `ptable.lock`
- (b) Only if L is not `ptable.lock`
- (c) Never
- (d) Always

Ans: (b)

8. Consider a system running xv6. You are told that a process in kernel mode acquires a spinlock (it can be any of the locks in the kernel code, you are not told which one). While the process holds this spin lock, is it correct OS design for it to:

- (a) process interrupts on the core in which it is running?
- (b) call the `sched` function to give up the CPU?

For each question above, you must first answer Yes or No. If your answer is yes, you must give an example from the code (specify the name of the lock, and any other information about the scenario) where such an event occurs. If your answer is no, explain why such an event cannot occur.

Ans:

- (a) No, it cannot. The interrupt may also require the same spinlock, leading to a deadlock.
 - (b) Yes it is possible. Processes giving up CPU call sched with ptable.lock held.
9. Consider the following snippet of code from the sleep function of xv6. Here, lk is the lock given to the sleep function as an argument.

```
if(lk != &ptable.lock){  
    acquire(&ptable.lock);  
    release(lk);  
}
```

For each of the snippets of code shown below, explain what would happen if the original code shown above were to be replaced by the code below. Does this break the functionality of sleep? If yes, explain what would go wrong. If not, explain why not.

- (a) acquire(&ptable.lock);
 release(lk);
- (b) release(lk);
 acquire(&ptable.lock);

Ans:

- (a) This code will deadlock if the lock given to sleep is ptable.lock itself.
(b) A wakeup may run between the release and acquire steps, leading to a missed wakeup.
- 10. In xv6, when a process calls sleep to block on a disk read, suggest what could be used as a suitable channel argument to the sleep function (and subsequently by wakeup), in order for the sleep and wakeup to happen correctly.
Ans: Address of struct buf (can be block number also?)
- 11. In xv6, when a process calls wait to block for a dead child, suggest what could be used as a suitable channel argument in the sleep function (and subsequently by wakeup), in order for the sleep and wakeup to happen correctly.
Ans: Address of parent struct proc (can be PID of parent?)
- 12. Consider the exit system call in xv6. The exit function acquires ptable.lock before giving up the CPU (in the function sched) for one last time. Who releases this lock subsequently?
Ans: The process that runs immediately afterwards (or scheduler)

13. In xv6, when a process calls wakeup on a channel to wakeup another process, does this lead to an immediate context switch of the process that called wakeup (immediately after the wakeup instruction)? (Yes/No)

Ans: No

14. When a process terminates in xv6, when is the struct proc entry of the process marked as unused/free?

- (a) During the execution of exit
- (b) During the sched function that performs the context switch
- (c) In the scheduler, when it iterates over the array of all struct proc
- (d) During the execution of wait by the parent

Ans: (d)

15. In which of the following xv6 system call implementations is there a possibility of the process calling sched() to give up the CPU? Assume no timer interrupts occur during the system call execution. Tick all that apply: fork / exit / wait / none of the above

Ans: exit, wait

16. In which of the following xv6 system call implementations will a process *always* invoke sched() to give up the CPU? Assume no timer interrupts occur during the system call execution. Tick all that apply: fork / exit / wait / none of the above

Ans: exit

17. Under which conditions does the wait() system call in xv6 block? Tick all that apply: when the process has [no children / only one running child / only one zombie child / two children, one running and one a zombie]

Ans: Only one running child

18. Consider a system with two CPU cores (C0 and C1) that is running the xv6 OS. A process P0 running on core C0 is in kernel mode, and has acquired a kernel spinlock. Another process P1 on core C1 is also in kernel mode, and is busily spinning for the same spinlock that is held by P0. Which of the following best describes the set of cores on which interrupts are disabled?

- (A) C0 and C1
- (B) Only C0
- (C) Only C1
- (D) Neither C0 nor C1

Ans: A, interrupts need to be disabled before starting to spin also.

19. Consider a process in xv6 that invokes the wakeup function in kernel mode. Which of the following statements is/are true?

- (a) The wakeup function immediately causes a context switch to one of the processes sleeping on a particular channel value.
- (b) The wakeup function wakes up (marks as ready) all processes sleeping on a particular channel value.
- (c) The wakeup function wakes up (marks as ready) only the first process that is blocked on a particular channel value.

- (d) The wakeup function wakes up (marks as ready) only the last process that is blocked on a particular channel value.

Ans: (b)

20. Consider a process in kernel mode in xv6, which invokes the sleep function to block. One of the arguments to the sleep function is a kernel spinlock. Which of the following statements is/are true?

- (a) If the lock given to sleep is ptable.lock, then the lock is not released before the context switch.
- (b) If the lock given to sleep is not ptable.lock, then the lock is not released before the context switch.
- (c) If the lock given to sleep is not ptable.lock, then the lock is released before the context switch, but only after acquiring ptable.lock.
- (d) If the lock given to sleep is not ptable.lock, then the lock is released before the context switch and ptable.lock need not be acquired.

Ans: (a), (c)

Practice Problems: Memory Management in xv6

1. Consider a system with V bytes of virtual address space available per process, running an xv6-like OS. Much like with xv6, low virtual addresses, up to virtual address U , hold user data. The kernel is mapped into the high virtual address space of every process, starting at address U and upto the maximum V . The system has P bytes of physical memory that must all be usable. The first K bytes of the physical memory holds the kernel code/data, and the rest $P - K$ bytes are free pages. The free pages are mapped once into the kernel address space, and once into the user part of the address space of the process they are assigned to. Like in xv6, the kernel maintains page table mappings for the free pages even after they have been assigned to user processes. The OS does not use demand paging, or any form of page sharing between user space processes. The system must be allowed to run up to N processes concurrently.
 - (a) Assume $N = 1$. Assume that the values of V , U , and K are known for a system. What values of P (in terms of V , U , K) will ensure that all the physical memory is usable?
 - (b) Assume the values of V , K , and N are known for a system, but the value of P is not known apriori. Suggest how you would pick a suitable value (or range of values) for U . That is, explain how the system designer must split the virtual address space into user and kernel parts.

Ans:

- (a) The kernel part of the virtual address space of a process should be large enough to map all physical memory, so $V - U \geq P$. Further, the user part of the virtual address space of a process should fit within the free physical memory that is left after placing the kernel code, so $U \leq P - K$. Putting these two equations together will get you a range for P .
 - (b) If there are N processes, the second equation above should be modified so that the combined user part of the N processes can fit into the free physical pages. So we will have $N * U \leq P - K$. We also have $P \leq V - U$ as before. Eliminating P (unknown), we get $U \leq \frac{V-K}{N+1}$.
2. Consider a system running the xv6 OS. A parent process P has forked a child C , after which C executes the `exec` system call to load a different binary onto its memory image. During the execution of `exec`, does the kernel stack of C get reinitialized or reallocated (much like the page tables of C)? If it does, explain what part of `exec` performs the reinitialization. If not, explain why not.

Ans: The kernel stack cannot be reallocated during `exec`, because the kernel code is executing on the kernel stack itself, and releasing the stack on which the kernel is running would be disastrous. Small changes are made to the trap frame however, to point to the start of the new executable.

3. The xv6 operating system does not implement copy-on-write during fork. That is, the parent's user memory pages are all cloned for the child right at the beginning of the child's creation. If xv6 were to implement copy-on-write, briefly explain how you would implement it, and what changes need to be made to the xv6 kernel. Your answer should not just describe what copy-on-write is (do not say things like "copy memory only when parent or child modify it"), but instead concretely explain *how* you would ensure that a memory page is copied only when the parent/child wishes to modify it.

Ans: The memory pages shared by parent and child would be marked read-only in the page table. Any attempt to write to the memory by the parent or child would trap the OS, at which point a copy of the page can be made.

4. Consider a process P in xv6 that has executed the `kill` system call to terminate a victim process V. If you read the implementation of `kill` in xv6, you will see that V is not terminated immediately, nor is its memory reclaimed during the execution of the `kill` system call itself.
 - (a) Give one reason why V's memory is not reclaimed during the execution of `kill` by P.
 - (b) Describe when V is actually terminated by the kernel.

Ans: Memory cannot be reclaimed during the kill itself, because the victim process may actually be executing on another core. Processes are periodically checked for whether they have been killed (say, when they enter/exit kernel mode), and termination and memory reclamation happens at a time that is convenient to the kernel.

5. Consider the implementation of the `exec` system call in xv6. The implementation of the system call first allocates a new set of page tables to point to the new memory image, and switches page tables only towards the end of the system call. Explain why the implementation keeps the old page tables intact until the end of `exec`, and not rewrite the old page tables directly while building the new memory image.

Ans: The `exec` system call retains the old page tables, so that it can switch back to the old image and print an error message if exec does not succeed. If exec succeeds however, the old memory image will no longer be needed, hence the old page tables are switched and freed.

6. In a system running xv6, for every memory access by the CPU, the function `walkpgdir` is invoked to translate the logical address to physical address. [T/F]

Ans: F

7. Consider a process in xv6 that makes the `exec` system call. The EIP of the `exec` instruction is saved on the kernel stack of the process as part of handling the system call. When and under what conditions is this EIP restored from the stack causing the process to execute the statement after `exec`?

Ans: If some error occurs during exec, the process uses the eip on trap frame to return to instruction after exec in the old memory image.

8. Consider a process in an xv6 system. Consider the following statement: "All virtual memory addresses starting from 0 to $N - 1$ bytes, where N is the process size (`proc->sz`), can be read by the process in user mode." Is the above statement true or false? If true, explain why. If false, provide a counter example.

Ans: False, the guard page is not accessible by user.

9. When an xv6 process invokes the `exec` system call, where are the arguments to the system call first copied to, before the system call execution begins? Tick one: user heap / user stack / trap frame / context structure

Ans: user stack

10. When a process successfully returns from the `exec` system call to its new memory image in xv6, where are the commandline arguments given to the new executable to be found? Tick one: user heap / user stack / trap frame / context structure

Ans: user stack

11. After the `exec` system call completes successfully in xv6, where is the EIP of the starting instruction of the new executable stored, to enable the process to start execution at the entry of the new code? Tick one: user heap / user stack / trap frame / context structure

Ans: trap frame

12. Consider the list of free memory frames maintained by xv6 in the free list. Whenever a process in kernel mode requests memory via the function `kalloc()`, xv6 extracts and returns free frames from this list. Which of the following things can be stored in the pages thus allocated from the free list?

- (a) New page table created for child process during fork
- (b) New memory image (code/data/stack/heap) created for child during fork
- (c) Kernel stack of new process created in fork
- (d) New page table created for the new memory image in exec

Ans: (a), (b), (c), (d)

13. Consider a process P in xv6 that executes the `sbrk` system call to increase the size of the user part of its virtual address space from N1 bytes to N2 bytes. Assume N1 and N2 are multiples of page size, N2 > N1, and the difference between N1 and N2 is K pages. Which of the following statements is/are true about the actions that occur during the execution of the system call?

- (a) The OS assigns K free physical frames to the process and adds the frame numbers into the page table.
- (b) The OS does not assign any new physical frames to the process, but updates the page table.
- (c) The OS updates (N2-N1) page table entries in the page table of P.
- (d) The OS updates K page table entries in the page table of P.

Ans: (a), (d)

14. Consider a process P running in xv6. The high virtual addresses in the address space of P are assigned to OS code/data. Consider a virtual address V assigned to OS code/data. Which of the following statements is/are true?

- (a) If the CPU accesses address V in user mode, the MMU raises a trap to the OS.
- (b) The address V is translated to the same physical address by the page tables of all processes in the system.

- (c) The address V can be translated to different physical address by the page tables of different processes in the system.
- (d) The page table entry that translates address V to a physical address is present in the page table of P only when it is running in kernel mode.

Ans: (a), (b)

15. Consider a process running in xv6. The page table of the process maps virtual address V to physical address P. Which of the following statements is/are true? Assume KERNBASE is set to 2GB in xv6.

- (a) $V = P + \text{KERNBASE}$ always
- (b) $V = P - \text{KERNBASE}$ always
- (c) $V = P + \text{KERNBASE}$ only for $V \geq \text{KERNBASE}$
- (d) $V = P + \text{KERNBASE}$ only for $V < \text{KERNBASE}$

Ans: (c)

Practice Problems: Process Management in xv6

1. Consider the following lines of code in a program running on xv6.

```
int ret = fork();
if(ret==0) { //do something in child}
else { //do something in parent}
```

- When a new child process is created as part of handling fork, what does the kernel stack of the new child process contain, after fork finishes creating it, but just before the CPU switches away from the parent?
- How is the kernel stack of the newly created child process different from that of the parent?
- The EIP value that is present in the trap frames of the parent and child processes decides where both the processes resume execution in user mode. Do both the EIP pointers in the parent and child contain the same logical address? Do they point to the same physical address in memory (after address translation by page tables)? Explain.
- How would your answer to (c) above change if xv6 implemented copy-on-write during fork?
- When the child process is scheduled for the first time, where does it start execution in kernel mode? List the steps until it finally gets to executing the instruction after fork in the program above in user mode.

Ans:

- It contains a trap frame, followed by the context structure.
- The parent's kernel stack has only a trap frame, since it is still running and has not been context switched out. Further, the value of the EAX register in the two trap frames is different, to return different values in parent and child.
- The EIP value points to the same logical address, but to different physical addresses, as the parent and child have different memory images.
- With copy-on-write, the physical addresses may be the same as well, as long as both parent and child have not modified anything.
- Starts at forkret, followed by trapret. Pops the trapframe and starts executing at the instruction right after fork.

2. Suppose a machine (architecture: x86, single core) has two runnable processes P1 and P2. P1 executes a line of code to read 1KB of data from an open file on disk to a buffer in its memory.

The content requested is not available in the disk buffer cache and must be fetched from disk. Describe what happens from the time the instruction to read data is started in P1, to the time it completes (causing the process to move on to the next instruction in the program), by answering the following questions.

- (a) The code to read data from disk will result in a system call, and will cause the x86 CPU to execute the `int` instruction. Briefly describe what the CPU's `int` instruction does.
- (b) The `int` instruction will then call the kernel's code to handle the system call. Briefly describe the actions executed by the OS interrupt/trap/system call handling code before the read system call causes P1 to block.
- (c) Now, because process P1 has made a blocking system call, the CPU scheduler context switches to some other process, say P2. Now, the data from the disk that unblocks P1 is ready, and the disk controller raises an interrupt while P2 is running. On whose kernel stack does this interrupt processing run?
- (d) Describe the contents of the kernel stacks of P1 and P2 when this interrupt is being processed.
- (e) Describe the actions performed by P2 in kernel mode when servicing this disk interrupt.
- (f) Right after the disk interrupt handler has successfully serviced the interrupt above, and before any further calls to the scheduler to context switch from P2, what is the state of process P1?

Ans:

- (a) The CPU switches to kernel mode, switches to the kernel stack of the process, and pushes some registers like the EIP onto the kernel stack.
 - (b) The kernel pushes a few other registers, updates segment registers, and starts executing the system call code, which eventually causes P1 to block.
 - (c) P2's kernel stack
 - (d) P2's kernel stack has a trapframe (since it switched to kernel mode). P1's kernel stack has both a context structure and a trap frame (since it is currently context switched out).
 - (e) P2 saves user context, switches to kernel mode, services the disk interrupt that unblocks P1, marks P1 as ready, and resumes its execution in userspace.
 - (f) Ready / runnable.
3. Consider a newly created process in xv6. Below are the several points in the code that the new process passes through before it ends up executing the line just after the `fork` statement in its user space. The EIP of each of these code locations exists on the kernel stack when the process is scheduled for the first time. For each of these locations below, precisely explain where on the kernel stack the corresponding EIP is stored (in which data structure etc.), and when (as part of which function or stage of process creation) is the EIP written there. Be as specific as possible in your answers.

- (a) `forkret`
- (b) `trapret`
- (c) just after the `fork()` system call in userspace

Ans:

- (a) EIP of forkret is stored in struct context by allocproc.
 - (b) EIP of trapret is stored on kernel stack by allocproc.
 - (c) EIP of fork system call code is stored in trapframe in parent, and copied to child's kernel stack in the fork function.
4. Consider a process that has performed a blocking disk read, and has been context switched out in xv6. Now, when the disk interrupt occurs with the data requested by the process, the process is unblocked and context switched in immediately, as part of handling the interrupt. [T/F]

Ans: F

5. In xv6, state the system call(s) that result in new `struct proc` objects being allocated.

Ans: fork

6. Give an example of a scenario in which the xv6 dispatcher / `swtch` function does NOT use a `struct context` created by itself previously, but instead uses an artificially hand-crafted `struct context`, for the purpose of restoring context during a context switch.

Ans: When running process for first time (say, after fork).

7. Give an example of a scenario in xv6 where a `struct context` is stored on the kernel stack of a process, but this context is never used or restored at any point in the future.

Ans: When process has finished after exit, its saved context is never restored.

8. Consider a parent process P that has executed a fork system call to spawn a child process C. Suppose that P has just finished executing the system call code, but has not yet returned to user mode. Also assume that the scheduler is still executing P and has not context switched it out. Below are listed several pieces of state pertaining to a process in xv6. For each item below, answer if the state is identical in both processes P and C. Answer Yes (if identical) or No (if different) for each question.

- (a) Contents of the PCB (`struct proc`). That is, are the PCBs of P and C identical? (Yes/No)
- (b) Contents of the memory image (code, data, heap, user stack etc.).
- (c) Contents of the page table stored in the PCB.
- (d) Contents of the kernel stack.
- (e) EIP value in the trap frame.
- (f) EAX register value in the trap frame.
- (g) The physical memory address corresponding to the EIP in the trap frame.
- (h) The files pointed at by the file descriptor table. That is, are the file structures pointed at by any given file descriptor identical in both P and C?

Ans:

- (a) No
- (b) Yes
- (c) No

- (d) No
 - (e) Yes
 - (f) No
 - (g) No
 - (h) Yes
9. Suppose the kernel has just created the first user space “init” process, but has not yet scheduled it. Answer the following questions.
- (a) What does the EIP in the trap frame on the kernel stack of the process point to?
 - (b) What does the EIP in the context structure on the kernel stack (that is popped when the process is context switched in) point to?

Ans:

- (a) address 0 (first line of code in init user code)
 - (b) forkret / trapret
10. Consider a process P that forks a child process C in xv6. Compare the trap frames on the kernel stacks of P and C just after the fork system call completes execution, and before P returns back to user mode. State one difference between the two trap frames at this instant. Be specific in your answer and state the exact field/register value that is different.

Ans: EAX register has different value.

11. In xv6, the EIP within the `struct context` on the kernel stack of a process usually points to the `switch` statement in the `sched` function, where the process gives up its CPU and switches to the scheduler thread during a context switch. Which processes are an exception to this statement? That is, for which processes does the EIP on the context structure point to some other piece of code?

Ans: Newly created processes / processes running for first time

12. When a trap occurs in xv6, and a process shifts from user mode to kernel mode, which entity switches the CPU stack pointer from pointing to the user stack of the running program to its kernel stack? Tick one: x86 hardware instruction / xv6 assembly code

Ans: x86 hardware

13. Consider a process P in xv6, which makes a system call, goes to kernel mode, runs the system call code, and comes back into user mode again. The value of the EAX register is preserved across this transition. That is, the value of the EAX register just before the process started the system call will always be equal to its value just after the process has returned back to user mode. [T/F]

Ans: False, EAX is used to store system call number and return value, so it changes.

14. When a trap causes a process to shift from user mode to kernel mode in xv6, which CPU data registers are stored in the trapframe (on the kernel stack) of the process? Tick one: all registers / only callee-save registers

Ans: All registers

15. When a process in xv6 wishes to pass one or more arguments to the system call, where are these arguments initially stored, before the process initiates a jump into kernel mode? Tick one: user stack / kernel stack

Ans: User stack, as user program cannot access kernel stack

16. Consider the context switch of a CPU from the context of process P1 to that of process P2 in xv6. Consider the following two events in the chronological order of the events during the context switch: (E1) the ESP (stack pointer) shifts from pointing to the kernel stack of P1 to the kernel stack of P2; (E2) the EIP (program counter) shifts from pointing to an address in the memory allocated to P1 to an address in the memory allocated to P2. Which of the following statements is/are true regarding the relative ordering of events E1 and E2?

- (a) E1 occurs before E2.
- (b) E2 occurs before E1.
- (c) E1 and E2 occur simultaneously via an atomic hardware instruction.
- (d) The relative ordering of E1 and E2 can vary from one context switch to the other.

Ans: (a)

17. Consider the following actions that happen during a context switch from thread/process P1 to thread/process P2 in xv6. (One of P1 or P2 could be the scheduler thread as well.) Arrange the actions below in chronological order, from earliest to latest.

- (A) Switch ESP from kernel stack of P1 to that of P2
- (B) Pop the callee-save registers from the kernel stack of P2
- (C) Push the callee-save registers onto the kernel stack of P1
- (D) Push the EIP where execution of P1 stops onto the kernel stack of P1.

Ans: DCAB

18. Consider a process P in xv6 that invokes the wait system call. Which of the following statements is/are true?

- (a) If P does not have any zombie children, then the wait system call returns immediately.
- (b) The wait system call always blocks process P and leads to a context switch.
- (c) If P has exactly one child process, and that child has not yet terminated, then the wait system call will cause process P to block.
- (d) If P has two or more zombie children, then the wait system call reaps all the zombie children of P and returns immediately.

Ans: (c)

19. Consider a process P in xv6 that executes the exec system call successfully. Which of the following statements is/are true?

- (a) The exec system call changes the PID of process P.
- (b) The exec system call allocates a new page table for process P.

- (c) The exec system call allocates a new kernel stack for process P.
- (d) The exec system call changes one or more fields in the trap frame on the kernel stack of process P.

Ans: (b), (d)

20. Consider a process P in xv6 that executes the exec system call successfully. Which of the following statements is/are true?

- (a) The arguments to the exec system call are first placed on the user stack by the user code.
- (b) The arguments to the exec system call are first placed on the kernel stack by the user code.
- (c) The arguments (argc, argv) to the new executable are placed on the kernel stack by the exec system call code.
- (d) The arguments (argc, argv) to the new executable are placed on the user stack by the exec system call code.

Ans: (a), (d)

21. Consider a newly created child process C in xv6 that is scheduled for the first time. At the point when the scheduler is just about to context switch into C, which of the following statements is/are true about the kernel stack of process C?

- (a) The top of the kernel stack contains the context structure, whose EIP points to the instruction right after the fork system call in user code.
- (b) The bottom of the kernel stack has the trapframe, whose EIP points to the forkret function in OS code.
- (c) The top of the kernel stack contains the context structure, whose EIP points to the forkret function in OS code.
- (d) The bottom of the kernel stack contains the trap frame, whose EIP points to the trapret function in OS code.

Ans: (c)

22. Consider a trapframe stored on the kernel stack of a process P in xv6 that jumped from user mode to kernel mode due to a trap. Which of the following statements is/are true?

- (a) All fields of the trapframe are pushed onto the kernel stack by the OS code.
- (b) All fields of the trapframe are pushed onto the kernel stack by the x86 hardware.
- (c) The ESP value stored in the trapframe points to the top of the kernel stack of the process.
- (d) The ESP value stored in the trapframe points to the top of the user stack of the process.

Ans: (d)

23. Consider a process P that has made a blocking disk read in xv6. The OS has issued a disk read command to the disk hardware, and has context switched away from P. Which of the following statements is/are true?

- (a) The top of the kernel stack of P contains the return address, which is the value of EIP pointing to the user code after the read system call.
- (b) The bottom of the kernel stack of P contains the trapframe, whose EIP points to the user code after the read system call.
- (c) The top of the kernel stack of P contains the context structure, whose EIP points to the user code after the read system call.
- (d) The CPU scheduler does not run P again until after the disk interrupt that unblocks P is raised by the device hardware.

Ans: (b), (d)

24. In the implementation of which of the following system calls in xv6 are new ptable entries allocated or old ptable entries released (marked as unused)?

- (a) fork
- (b) exit
- (c) exec
- (d) wait

Ans: (a), (d)

25. A process has invoked exit() in xv6. The CPU has completed executing the OS code corresponding to the exit system call, and is just about to invoke the swtch() function to switch from the terminated process to the scheduler thread. Which of the following statements is/are true?

- (a) The stack pointer ESP is pointing to some location within the kernel stack of the terminated process
- (b) The MMU is using the page table of the terminated process
- (c) The state of the terminated process in the ptable is RUNNING
- (d) The state of the terminated process in the ptable is ZOMBIE

Ans: (a), (b), (d)

Lecture Notes for CS347: Operating Systems

Mythili Vutukuru, Department of Computer Science and Engineering, IIT Bombay

6. Scheduling and Synchronization in xv6

6.1 Locks and (equivalents of) conditional variables

- Locks (i.e., spinlocks) in xv6 are implemented using the `xchg` atomic instruction. The function to acquire a lock (line 1574) disables all interrupts, and the function that releases the lock (line 1602) re-enables them.
- xv6 provides `sleep` (line 2803) and `wakeup` (line 2864) functions, that are equivalent to the wait and signal functions of a conditional variable. The sleep and wakeup functions must be invoked with a lock that ensures that the sleep and wakeup procedures are completed atomically. A process that wishes to block on a condition calls `sleep(chan, lock)`, where `chan` is any opaque handle, and `lock` is any lock being used by the code that calls sleep/wakeup. The sleep function releases the lock the process came with, and acquires a specific lock that protects the scheduler's process list (`ptable.lock`), so that it can make changes to the state of the process and invoke the scheduler (line 2825). Note that the function checks that the lock provided to it is not this `ptable.lock` to avoid locking it twice. All calls to the scheduler should be made with this `ptable.lock` held, so that the scheduler's updating of the process list can happen without race conditions.
- Once the sleep function calls the scheduler, it is context switched out. The control returns back to this line (of calling the scheduler) once the process has been woken up and context switched in by the scheduler again, with the `ptable.lock` held. At that time, the sleep function releases this special lock, reacquires the original lock, and returns back in the woken up process. Note that the sleep function holds at least one of the two locks—the lock given to it by the caller, or the `ptable.lock`—at any point of time, so that no other process can run wakeup while sleep is executing, because a process will lock one or both of these locks while calling wakeup. (Understand why lines 2818 and 2819 can't be flipped.)
- A process that makes the condition true will invoke `wakeup(chan)` while it holds the lock. The wakeup call makes all waiting processes runnable, so it is indeed equivalent to the signal broadcast function of `pthreads` API. Note that a call to `wakeup` does not actually context switch to the woken up processes. Once `wakeup` makes the processes runnable, these processes can actually run when the scheduler is invoked at a later time.
- xv6 does not export the sleep and wakeup functionality to userspace processes, but makes use of it internally at several places within its code. For example, the implementation of pipes (sheet 65) clearly maps to the producer-consumer problem with bounded buffer, and the implementation uses locks and conditional variables to synchronize between the pipe writer and pipe reader processes.

6.2 Scheduler and context switching

- xv6 uses the following names for process states (line 2350): UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE. The `proc` structures are all stored in a linked list.
- Every CPU has a scheduler thread that calls the `scheduler` function (line 2708) at the start, and loops in it forever. The job of the scheduler function is to look through the list of processes, find a runnable process, set its state to RUNNING, and switch to the process.
- All actions on the list of processes are protected by `ptable.lock`. Any process that wishes to change this data structure must hold the lock, to avoid race conditions. What happens if the lock is not held? It is possible that two CPUs find a process to be runnable, and switch it to running state simultaneously, causing one process to run at two places. So, all actions of the `scheduler` function are always done with the lock held.
- Every process has a `context` structure on its stack (line 2343), that stores the values of the CPU registers that must be saved during a context switch. The registers that are saved as part of the context include the EIP (so that execution can resume at the instruction where it stopped), ESP (so that execution can continue on the stack where it left off), and a few other general purpose registers. To switch the CPU from the current running process P1 to another process P2, the registers of P1 are saved in its context on its stack, and the registers of P2 are reloaded from its context (that would have been saved in the past when P2 was switched out). Now, when P1 must be resumed again, its registers are reloaded from its context, and it can resume execution where it left off.
- The `swtch` function (line 2950) does the job of switching between two contexts, and old one and a new one. The step of pushing registers into the old stack is exactly similar to the step of restoring registers from the new stack, because the new stack was also created by `swtch` at an earlier time. The only time when the `context` structure is not pushed by `swtch` is when a process is created for the first time. For a new process, `allocproc` writes the context onto the new kernel stack (lines 2488-2491), which will be loaded into the CPU registers by `swtch` when the process executes for the first time. All new processes start at `forkret`, so `allocproc` writes this address into the EIP of the context it is creating. Except in this case of a new process, the EIP stored in context is always the address at which the running process invoked `swtch`, and it resumes at the same point at a later time. Note that `swtch` does not explicitly store the EIP to point to the address of the `swtch` statement, but the EIP is automatically stored on the stack as part of making the function call to `swtch`.
- In xv6, the scheduler runs as a separate thread with its own stack. Therefore, context switches happen from the kernel mode of a running process to the scheduler thread, and from the scheduler thread to the new process it has identified. (Other operating systems can switch directly between kernel modes of two processes as well.) As a result, the `swtch` function is called at two places: by the scheduler (line 2728) to switch from the scheduler thread to a new process, and by a running process that wishes to give up the CPU in the function `sched` (line 2766). A running process always gives up the CPU at this call to `swtch` in line 2766, and always

resumes execution at this point at a later time. The only exception is a process running for the first time, that never would have called `swtch` at line 2766, and hence never resumes from there.

- A process that wishes to relinquish the CPU calls the function `sched`. This function triggers a context switch, and when the process is switched back in at a later time, it resumes execution again in `sched` itself. Thus a call to `sched` freezes the execution of a process temporarily. When does a process relinquish its CPU in this fashion? (i) When a timer interrupt occurs and it is deemed that the process has run for too long, the trap function calls `yield`, which in turn calls `sched` (line 2776). (ii) When a process terminates itself using `exit`, it calls `sched` one last time to give up the CPU (line 2641). (iii) When a process has to block for an event and sleep, it calls `sched` to give up the CPU (line 2825). The function `sched` simply checks various conditions, and calls `swtch` to switch to the scheduler thread.
- Any function that calls `sched` must do so with the `ptable.lock` held. This lock is held all during the context switch. During a context switch from P1 to P2, P1 locks `ptable.lock`, calls `sched`, which switches to the scheduler thread, which again switches to process P2. When process P2 returns from `sched`, it releases the lock. For example, you can see the lock and release calls before and after the call to `sched` in `yield`, `sleep`, and `exit`. Note that the function `forkret` also releases the lock for a process that is executed for the first time, since a new process does not return in `sched`. Typically, a process that locks also does the corresponding unlock, except during a context switch when a lock is acquired by one process and released by the other.
- Note that all interrupts are disabled when any lock is held in xv6, so all interrupts are disabled during a context switch. If the scheduler finds no process to run, it periodically releases `ptable.lock`, re-enables interrupts, and checks for a runnable process again.

6.3 Implementation of sleep, wakeup, wait, exit

- With a knowledge of how scheduling works, it may be worth revisiting the sleep and wakeup functions (sheet 28), especially noting the subtleties around locks. The `sleep` function (line 2803) must eventually call `sched` to give up the CPU, so it must acquire `ptable.lock`. The function first checks that the lock held already is not `ptable.lock` to avoid deadlocks, and releases the lock given to it after acquiring `ptable.lock`. Is it OK to release the lock given to `sleep` before actually calling `sched` and going to sleep? Yes it is, because `wakeup` also requires `ptable.lock`, so there is no way a `wakeup` call can run while `ptable.lock` is held. Is it OK to release the lock given to `sleep` before acquiring `ptable.lock`? No, it is not, as `wakeup` may be invoked in the interim when no lock is held.
- When a parent calls `wait`, the `wait` function (line 2653) acquires `ptable.lock`, and looks over all processes to find any of its zombie children. If none is found, it calls `sleep`. Note that the lock provided to `sleep` in this case is also `ptable.lock`, so `sleep` must not attempt to re-lock it again. When a child calls `exit` (line 2604), it acquires `ptable.lock`, and wakes up its parent. Note that the `exit` function does not actually free up the memory of the process. Instead, the process simply marks itself as a zombie, and relinquishes the CPU by calling `sched`. When the parent wakes up in `wait`, it does the job of cleaning up its zombie child, and frees up the memory of the process. The `wait` and `exit` system calls provide a good use case of the `sleep` and `wakeup` functions.

Lecture Notes for CS347: Operating Systems

Mythili Vutukuru, Department of Computer Science and Engineering, IIT Bombay

10. The xv6 filesystem

10.1 Device Driver

- The data structure `struct buf` (line 3750) is the basic unit of storing a block's content in the xv6 kernel. It consists of 512 bytes of data, device and sector numbers, pointers to the previous and next buffers, and a set of flags (valid, dirty, busy).
- Sheets 41 and 42 contain the device driver code for xv6. The disk driver in xv6 is initialized in `ideinit` (line 4151). This function enables interrupts on last CPU, and waits for disk to be ready. It also checks if a second disk (disk 1) is present.
- The list `idequeue` is the queue of requests pending for the disk. This list is protected by `idelock`.
- The function `iderw` (line 4254) is the entry point to the device driver. It gets requests from the buffer cache / file system. Its job is to sync a specified buffer with disk. If the buffer is dirty, it must be written to the disk, after which the dirty flag must be cleared and the valid flag must be set. If the buffer is not dirty, and not valid, it indicates a read request. So the specified block is read into the buffer from the disk, and valid flag is set. If the buffer is valid and not dirty, then there is nothing to do. This function does not always send the requests to the disk, because the disk may be busy. Instead, it adds request to queue. If this is the first request in the queue, then it starts the requested operation and blocks until completion.
- The function `idestart` (line 4175) starts a new disk request—the required information to start the operation (block number, read/write instructions) are all provided to the driver. The process calling `idestart` will eventually sleep waiting for the operation to finish.
- The function `ideintr` (line 4202) is called when the requested disk operation completes. It updates flags, wakes up the process waiting for this block, and starts the next disk request.

10.2 Buffer cache

- The code for the buffer cache implementation in xv6 can be found in sheets 43 and 44. The buffer cache is simply an array of buffers initialized at startup in the main function, and is used to cache disk contents.
- Any code that wishes to read a disk block calls `bread`. This function first calls the function `bget`. `bget` looks for the particular sector in the buffer cache. There are a few possibilities.
 - (i) If the given block is in cache and isn't being used by anyone, then `bget` returns it. Note that this can be clean/valid or dirty/valid (i.e., someone else got it from disk, and maybe even wrote to it, but is not using it).
 - (ii) If it is in cache but busy, `bget` sleeps on it. After waking up, it tries to lock it once again by checking if the buffer corresponding to that sector number exists in cache.(Why can't it assume that the buffer is still available after waking up? Because someone else could have reused that slot for another block, as described next.)
 - (iii) If the sector number is not in cache, it must be fetched from disk. But for this, the buffer cache needs a free slot. So it starts looking from the end of the list (LRU), finds a clean, non busy buffer to recycle, allots it for this block, marks it as invalid (since the contents are of another block), and returns it. So when `bread` gets a buffer from cache that is not valid, it will proceed to read from disk.
- A process that locked the buffer in the cache may make changes to it. If it does so, it must flush the changes to disk using `bwrite` before releasing the lock on the buffer. `bwrite` first marks the buffer as dirty (so that the driver knows it is a write operation), and then calls `iderw` (which adds it to the request queue). Note that there is no guarantee that the buffer will be immediately written, since there may be other pending requests in the queue. Also note that the process still holds the lock on the buffer, as the busy flag is set, so no other process can access the buffer.
- When a process is done using a disk block, it calls `brelse`. This function releases the lock by clearing the busy flag, so other processes sleeping on this block can start using this disk block contents again (whether clean or dirty). This function also moves this block to the head of the buffer cache, so that the LRU replacement algorithm will not replace it immediately.

10.3 Logging

- Sheets 45–47 contain the code for transaction-based logging in xv6. A system call may change multiple data and metadata blocks. In xv6, all changes of one system call must be wrapped in a transaction and logged on disk, before modifying the actual disk blocks. On boot, xv6 recovers any committed but uninstalled transactions by replaying them. In the boot process, the function `initlog` (line 4556) initializes the log and does any pending recovery. Note that the recovery process ignores uncommitted transactions.
- A special space at the end of the disk is dedicated for the purpose of storing the log of uncommitted transactions, using the data structure `struct log` (line 4537). The `committing` field indicates that a transaction is committing, and that system calls should wait briefly. The field `outstanding` denotes the number of ongoing concurrent system calls—the changes from several concurrent system calls can be collected into the same transaction. The `struct logheader` indicates which buffers/sectors have been modified as part of this transaction.
- Any system call that needs to modify disk blocks first calls `begin_op` (line 4628). This function waits if there is not enough space to accommodate the log entry, or if another transaction is committing. After beginning the transaction, system call code can use `bread` to get access to a disk buffer. Once a disk buffer has been modified, the system call code must now call `log_write` instead of `bwrite`. This function updates the block in cache as dirty, but does not write to disk. Instead, it updates the transaction header to note this modified log. Note that this function absorbs multiple writes to the same block into the same log entry. Once the changes to the block have been noted in the transaction header, the block can be released using `brelse`. This process of `bread`, `log_write`, and `brelse` can be repeated for multiple disk blocks in a system call.
- Once all changes in a transaction are complete, the system call code must call `end_op` (line 4653). This function decrements the outstanding count, and starts the commit process once all concurrent outstanding system calls have completed. That is, the last concurrent system call will release the lock on the log, set the commit flag in the log header, and start the long commit process. The function `commit` (line 4701) first writes all modified blocks to the log on disk (line 4683), then writes the log header to disk at the start of the log (4604), then installs transactions from the log one block at a time. These functions that actually write to disk now call `bwrite` (instead of the system call code itself calling them). Once all blocks have been written to their original locations, the log header count is set to zero and written to disk, indicating that it has no more pending logged changes. Now, when recovering from the log, a non-zero count in the log header on disk indicates that the transaction has committed, but hasn't been installed, in which case the recovery module installs them again. If the log header count is zero, it could mean that only part of the transactions are in the log, or that the system exited cleanly, and nothing needs to be done in both cases.

10.4 Inodes

- Sheets 39 and 40 contain the data structures of the file system, the layout of the disk, the inode structures on disk and in memory, the directory entry structure, the superblock structure, and the `struct file` (which is the entry in the open file table). The inode contains 12 direct blocks and one single indirect block. Also notice the various macros to compute offsets into the inode and such. Sheets 47–55 contain the code that implements the core logic of the file system.
- In xv6, the disk inodes are stored in the disk blocks right after the super block; these disk inodes can be free or allocated. In addition, xv6 also has an in-memory cache of inodes, protected by the `i_cache.lock`. Kernel code can maintain C pointers to active inodes in memory, as part of working directories, open files etc. An inode has two counts associated with it: `nlink` says how many links point to this file in the directory tree. The count `ref` that is stored only in the memory version of the inode counts how many C pointers exist to the in-memory inode. A file can be deleted from disk only when both these counts reach zero.
- The function `readsbs` (4777) reads superblock from disk, and is used to know the current state of disk. The function `balloc` (line 4804) reads the free block bit map, finds a free block, fetches it into the buffer cache using `bread`, zeroes it, and returns it. The function `bfree` (line 4831) frees the block, which mainly involves updating the bitmap on disk. Note that all of these functions rely on the buffer cache and log layers, and do not access the device driver or disk directly.
- The function `ialloc` (line 4953) allocates an on-disk inode for a file for the first time. It looks over all disk inodes, finds a free entry, and gets its number for a new file. The function `iget` (line 5004) tries to find an in-memory copy of an inode of a given number. If one exists (because some one else opened the file), it increments the reference count, and returns it. If none exists, it finds an unused empty slot in the inode cache and returns that inode from cache. Note that `iget` is not guaranteeing that the in-memory inode is in sync with the disk version. That is the job of the function `ilock` (line 5053), which takes an in memory inode and updates its contents from the corresponding disk inode. So, to allocate a new inode to a file, one needs to allocate an on-disk inode, an in-memory inode in the cache, and sync them up.
- Why do we need two separate functions for the logic of `iget` and `ilock`? When a process locks an inode, it can potentially make changes to the inode. Therefore, we want to avoid several processes concurrently accessing an inode. Therefore, when one process has locked an inode in the cache, another process attempting to lock will block. However, several processes can maintain long term pointers to an inode using `iget`, without modifying the inode. As long as a C pointer is maintained, the inode won't be deleted from memory or from disk: the pointers serve to keep the inode alive, even if they do not grant permission to modify it.
- When a process locks an inode, and completes its changes, it must call `iunlock` (line 5085) to release the lock. This function simply clears the busy lock and wakes up any processes blocking on the lock. At this point after unlocking the inode, the changes made to the inode are in the

inode cache, but not on disk. To flush to the disk, one needs to call `iupdate` (line 4979). Finally, to release C pointer reference, a process must call `iput` (line 5108). Note that one must always call `iunlock` before `iput`, because if the reference is decremented and goes to zero, the inode may no longer be valid in memory.

- When can a file be deleted from disk? In other words, when is an inode freed on disk? When it has no links or C references in memory. So the best place to check this condition is in the `iput` function. When releasing the last memory reference to an inode, `iput` checks if the link count is also zero. If yes, it proceeds to clean up the file. First, it locks the inode by setting it to busy, so that the in-memory inode is not reused due to a zero reference count. Then, it releases all data blocks of the the file by calling `itrunc` (line 5206), clears all state of the inode, and updates it on disk. This operation frees up the inode on disk. Next, it clears the flags of the in-memory inode, freeing it up as well. Since the reference count of the in-memory inode is zero, it will be reused to hold another inode later. Note how `iput` releases and reacquires its lock when it is about to do disk operations, so as to not sleep with a spinlock.
- Notice how all the above operations that change the inode cache must lock `icache.lock`, because multiple processes may be accessing the cache concurrently. However, when changing blocks on disk, the buffer cache handles locking by giving access to one process at a time in the function `bread`, and the file system code does not have to worry about locking the buffers itself.
- Finally, the functions `readi` and `writei` are used to read and write data at a certain offset in a file. Given the inode, the data block of the file corresponding to the offset is located, after which the data is read/written. The function `bmap` (line 5160) returns the block number of the n-th block of an inode, by traversing the direct and indirect blocks. If the n-th block doesn't exist, it allocates one new block, stores its address in the inode, and returns it. This function is used by the `readi` and `writei` functions.

10.5 Directories and pathnames

- A directory entry (line 3950) consists of a file name and an inode number. A directory is also a special kind of file (with its own inode), whose content is these directory entries written sequentially. Sheets 53–55 hold all the directory-related functions.
- The function `dirlookup` (line 5361) looks over each entry in a directory (by reading the directory “file”) to search for a given file name. This function returns only an unlocked inode of the file it is looking up, so that no deadlocks possible, e.g., look up for “.” entry. The function `dirlink` (line 5402) adds a new directory entry. It checks that the file name doesn’t exist, finds empty slot (inode number 0) in the directory, and updates it.
- Pathname lookup involves several calls to `dirlookup`, one for each element in the path name. Pathname lookup happens via two functions, `namei` and `nameiparent`. The former returns the inode of the last element of the path, and the latter stops one level up to return the inode of the parent directory. Both these functions call the function `namex` to do the work. The function `namex` starts with getting a pointer to the inode of the directory where traversal must begin (root or current directory). For every directory/file in the path name, the function calls `skipelem` (line 5465) to extract the name in the path string, and performs `dirlookup` on the parent directory for the name. It proceeds in this manner till the entire name has been parsed.
- Note: when calling `dirlookup` on a directory, the directory inode is locked and updated from disk, while the inode of the name looked up is not locked. Only a memory reference from `iget` is returned, and the caller of `dirlookup` must lock the inode if needed. This implementation ensures that only one inode lock (of the directory) is held at a time, avoiding the possibility of deadlocks.

10.6 File descriptors

- Every open file in xv6 is represented by a `struct file` (line 4000), which is simply a wrapper around inode/pipe, along with read/write offsets and reference counts. The global open file table `ftable` (line 5611-5615) is an array of file structures, protected by a global `ftable.lock`. The per-process file table in `struct proc` stores pointers to the `struct file` in the global file table. Sheets 56–58 contain the code handling `struct file`.
- The functions `filealloc`, `filedup`, and `fileclose` manipulate the global filetable. When an open file is being closed, and its reference count is zero, then the in-memory reference to the inode is also released via `iput`. Note how the global file table lock is released before calling `iput`, in order not to hold the lock and perform disk operations.
- The functions `fileread` and `filewrite` call the corresponding read/write functions on the pipe or inode, depending on the type of object the file descriptor is referring to. Note how the `fileread/filewrite` functions first lock the inode, so that its content is in sync with disk, before calling `readi/writei`. The `struct file` is one place where a long term pointer to an inode returned from `iget` is stored. The inode is only locked when the file has to be used for reading, writing, and so on. The function `filewrite` also breaks a long write down into smaller transactions, so that any one transaction does not exceed the maximum number of blocks that can be written per transaction.

10.7 System calls

- Once a `struct file` is allocated in the global table, a pointer to it is also stored in the per-process file table using the `fdalloc` function (line 5838). Note that several per-process file descriptor entries (in one process or across processes) can point to the same `struct file`, as indicated by the reference count in `struct file`. The integer file descriptor returned after `file open` is an index in the per-process table of pointers, which can be used to get a pointer to the `struct file` in the global table. All file-system related system calls use the file descriptor as an argument to refer to the file.
- The file system related system calls are all implemented in sheets 58-62. Some of them are fairly straightforward: they extract the arguments from the stack, and call the appropriate functions of the lower layers of the file system stack. For example, the `read/write` system calls extract the arguments (file descriptor, buffer, and the number of bytes) and call the `fileread/filewrite` functions on the file descriptor. It might be instructive to trace through the `read/write` system calls through all the layers down to the device driver, to fully understand the file system stack.
- The `link` system call (line 5913) creates a pointer (directory entry) from a new directory location to an old existing file (inode). If there were no transactions, the implementation of the `link` system call would have resulted in inconsistencies in the system in case of system crashes. For example, a crash that occurs after the old file's link count has been updated but before the new directory entry is updated would leave a file with one extra link, causing it to be never garbage cleaned. However, using transactions avoids all such issues. The `unlink` system call (line 6001) removes a path from a file system, by erasing a trace of it from the parent directory.
- The function `create` (line 6057) creates a new file (inode) and adds a link from the parent directory to the new file. If the file already exists, the function simply returns the file's inode. If the file doesn't exist, it allocates and updates a new inode on disk. Then it adds a link from the parent to the child. If the new file being created is a directory, the “.” and “..” entries are also created. The function finally returns the (locked) inode of the newly created file. Note that `create` holds two locks on the parent and child inodes, but there is no possibility of deadlock as the child inode has just been allocated.
- The `open` system call (line 6101) calls the `create` function above to get an inode (either a new one or an existing one, depending on what option is provided). It then allocates a global file table entry and a per-process entry that point to the inode, and returns the file descriptor. The function `create` is also used to create a directory (line 6151) or a device (line 6167).
- The `chdir` system call simply releases the reference of the old working directory, and adds a reference to the new one.
- To create a pipe, the function `pipealloc` (line 6471) allocates a new pipe structure. A pipe has the buffer to hold data, and counters for bytes read/written. Two `struct file` entries are created that both point to this pipe, so that read/write operations on these files go to the pipe. Finally file descriptors for these file table entries are also allocated and returned.

Lecture Notes on Operating Systems

Mythili Vutukuru, Department of Computer Science and Engineering, IIT Bombay

Process Management in xv6

We will study xv6 process management by walking through some of the paths in the code. Specifically, we will understand how traps are handled in xv6, how processes are created, how scheduling and context switching works, and how the xv6 shell starts up at boot time.

0. Background to understand x86 assembly code in xv6

- We will review the basics of x86 assembly language that are required to understand some simple assembly language code in the xv6 OS. While none of the xv6 programming assignments require you to write assembly code, some basic understanding will help you follow the concepts better. It is not required to read and understand all of this background right away; you can keep coming back to this section as and when you encounter assembly language code in xv6. We will begin with a review of the common x86 registers and instructions used in xv6 code.
- One may wonder: **why does an OS need to have assembly code?** Why can't everything be written in a high-level language like C? Below is a small explanation of the relationship between high-level language code, assembly code, and hardware instructions, specifically in the context of operating systems.
 - Every CPU comes with its own architecture: a set of instructions which do specific tasks (mov, add, load, store, compare, jump, and so on) and a set of registers which are used during computations. That is, when the CPU executes the instruction “add register1 register2”, what happens in the CPU hardware is defined by the CPU manufacturers. For example, this instruction will read the value in register1, and add it to register2. How does the CPU hardware accomplish this task? The CPU designers would have written code to implement this addition functionality in a hardware description language (HDL) like Verilog. You would have learnt about this in a previous course.
 - Now, given that you have CPU hardware capable of executing a bunch of instructions, software running on this hardware will tell the CPU what instructions exactly to execute in what order to accomplish a useful task. For example, you can write an assembly language program to read a few numbers from the terminal input, add them, and print them out to the screen. This software program will comprise of a sequence of instructions that the underlying hardware can understand and execute. That is, assembly software code consists of hardware instructions that tell the hardware to do certain tasks for you. Obviously, if you change the underlying hardware, the set of instructions will also change, so your assembly code must suitably change too.
 - But of course, writing assembly code is not everyone's cup of tea. This is where high-level languages like C come in. You can write code in more understandable language and the C compiler will translate this code into assembly code that your hardware will

understand and execute. The C compiler changes the translated assembly code depending on the underlying hardware architecture, because different architectures have different sets of instructions. So, your C program will compile to different assembly code on different architectures, but you as a C programmer do not have to worry about this. The C compiler will use some standard techniques to translate high level code into assembly.

- For example, if you write a statement like “int c = a + b;” in your code, it will use some standard template to translate this into assembly code, say like this: “load the integer from memory address of a into register R1, load b into register R2, add R1 and R2 and store it into R3, store R3 back into the memory location of c”. Thus, your one line of C code will be translated into one or more lines of assembly code by the compiler, and when you run your executable a.out, all of these instructions will be executed by the hardware to accomplish the task you have written in your C program.
 - Sometimes, you may not like the way the compiler translates a given statement into assembly. For example, maybe you wanted to use registers R4, R5, and R6 in the example above (for whatever reason). Sometimes, you may also want to execute certain instructions that are not easily expressible through a high level language. For example, you may want to switch your stack pointer from one value to another, but there is no C language expression that lets you do this. In such cases, you can write your own assembly code, instead of relying on the assembly code output by the C compiler. Though, for most users, compiler generated assembly will be good enough for most of us.
 - Now, operating system is also a large piece of software. For convenience, OS developers write most OS code in a high level language like C. However, for some parts of the OS code, the developers need more control on the underlying assembly code, and cannot rely on C code. Such small parts of an OS are written in assembly language. This part of the OS must be rewritten for every underlying architecture the OS needs to run on, while the C code can work across all architectures (because it is translated suitably by the compiler). Thus, every OS has an architecture-independent part of its code written in a high level language like C, and a small architecture-dependent part written in assembly. The same holds for xv6 as well.
- **x86 registers.** Several CPU registers are referenced in the discussion of operating system concepts. While the names and number of registers differ based on the CPU architecture, here are some names of x86 registers that you will find useful. (Note: this list of registers is by no means exhaustive.)
 1. EAX, EBX, ECX, EDX, ESI, and EDI are general purpose registers used to store variables during computations.
 2. The general purpose registers EBP and ESP store pointers to the base and top of the current stack frame. That is, they contain the (virtual) memory address of the base/top of the current stack frame of a process. (More on this later.)
 3. The program counter (PC) is also referred to as EIP (instruction pointer). It stores the (virtual) memory address of the instruction that is about to run next on the CPU.

4. The segment registers CS, DS, ES, FS, GS, and SS store pointers to various segments (e.g., code segment, data segment, stack segment) of the process memory. We will not study segments in detail, since xv6 does not use segmentation much. It is enough for you to understand that segment registers are changed when moving from user mode to kernel mode and vice versa.
5. The control registers like CR0 hold control information. For example, the CR3 register holds the address of the page table of the current running process. You can think of the page table as collection of pointers to (physical addresses of) memory locations in RAM where the memory image of a process is stored. This information is used by the CPU to translate virtual addresses to physical addresses.

- **x86 instructions.** The x86 ISA (instruction set architecture) has several instructions. Some common ones you will encounter when reading xv6 code are described below at a high level.

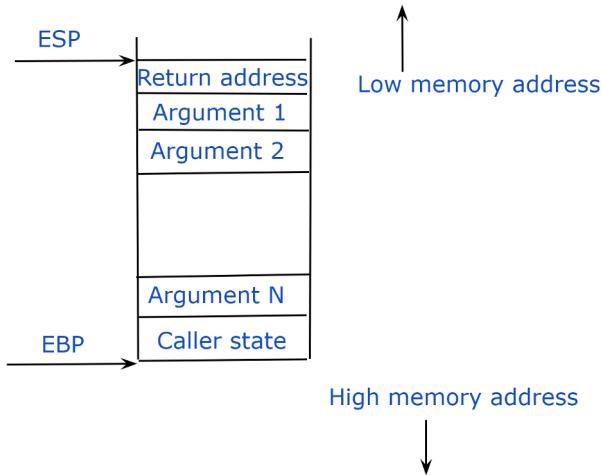
1. `mov src dst` instruction moves contents of `src` to `dst`.¹ The source/destination can be a constant value, content of a register (e.g., `%eax`), data present at a memory location whose address is stored in a register (e.g., `(%eax)`), or even data present at a memory address that is located at an offset from the address stored in a register (e.g., `4(%eax)` refers to data at a memory location that is 4 bytes after the memory address stored in `eax`).
2. `pop` pops the value at the top of the stack (pointed at by `ESP`) and stores it into the register provided as an argument to `pop`. Similarly, `push` pushes the argument provided onto the top of the stack. Both these instructions automatically update `ESP`. `pusha` and `popa` push/pop all general purpose registers in one go.
3. `jmp` jumps or changes PC to the instruction address provided.
4. `call` makes a function call and jumps to the address of the function provided as an argument, after storing the return address on the stack. The `ret` instruction returns from a function to the return address stored at the top of the stack.
5. Finally, some of the above instruction names have a letter appended at the end (e.g., `movw`, `pushl`) to indicate the different sizes of registers to be used in different architectures.

- **What happens on a function call.** We will now understand some basic C language calling conventions in x86, i.e., what happens on the stack when you make a function call. The C compiler does several things on the stack for you, the details of which are beyond the scope of this course, but below is a rough outline.

1. Some basics first. Processes usually allocate an empty area for the stack, and make `ESP/EBP` point to the bottom of this memory region, i.e., to a high memory address. As things are pushed onto the stack, the stack grows “upwards”, i.e., it starts at a high memory address and moves towards lower (empty) memory addresses. In other words, pushing something onto the stack decreases the value of the memory address stored in `ESP`. The `EBP` stays at the bottom of the stack unless explicitly changed.

¹This is called the AT&T syntax, and this is followed in xv6. A different syntax called the Intel syntax places the destination before the source.

2. You will also need to understand some terminology. Of the various x86 registers, EAX, ECX, and EDX are called caller-save registers, and EBX, ESI, EDI, EBP are called callee-save registers. This classification denotes an agreement between caller and callee during function calls in languages like C, on how to preserve context across function calls. The caller-save registers are saved by the caller, or the entity that makes the function call. The caller does not make any assumptions about the callee preserving the values of these registers, and saves them itself. In contrast, the callee-save registers must be saved by the callee, and restored to their previous values when returning to the caller.
3. To make a function call, the caller (the code making the function call) first saves the caller-save registers onto the stack. Then the caller pushes all the arguments passed to the function onto the stack in the reverse order (from right to left), which also updates the ESP. Now, when you read the contents of the stack from ESP downwards, you will find all the arguments from the top of the stack in left-to-right order. Finally, the `call` instruction is invoked. This instruction pushes the return address onto the stack, and the PC jumps to the function's instructions. Your function call has begun. The stack of the process looks like this now.



4. At this point, the control has been transferred to the callee, or the function that was called. The callee can now choose to push a new “stack frame” onto the stack, to hold all its data that it wishes to store on the stack during the function call. Before pushing the new stack frame, the EBP register points to the base of the previous caller’s stack frame, and the ESP register points to the top of the stack. A new stack frame is pushed onto the stack by the callee in the following manner: the old base of stack is saved on the stack (push EBP onto stack), and the new base of the stack is initialized to the old top of the stack ($EBP = ESP$). At this time, both EBP and ESP point to the same address, which is the top of the old stack frame / base of a new stack frame. Now, anything that is pushed onto the stack changes the value of ESP, while EBP remains at the bottom of this new stack frame.
5. Next, the callee stores local variables, callee-save registers, and whatever else it wishes to store on its stack frame, and begins its execution. The function can access the arguments passed to it by looking below the base of its new stack frame.

6. When the function completes, all the above actions are reversed. The callee pops the things it put on the stack, and invokes the `ret` instruction. This instruction uses the return address at the top of the stack to return to the caller. The caller then pops the things it put on the stack as well before resuming execution of code after the function call.
7. One important thing to note is that the callee places the return value of the function in the `EAX` register (by convention) before returning to the caller. The caller reads the function return value from this register.
8. All of these actions are done under the hood for you by the C compiler when translating your C code to assembly. However, if you directly write assembly code of a function, you will have to do these things yourself. A high level understanding of this process will be useful when we read assembly code in xv6, as you will see various things getting pushed and popped off the stack.

1. Handling Interrupts

- Let us begin by looking at the `proc` data structure (line 2353), that corresponds to the PCB structure studied in class. Especially note the fields corresponding to the kernel stack pointer, the trapframe, and the context structure. Every process in xv6 has a separate area of memory called the kernel stack that is allocated to it, to be used instead of the userspace stack when running in kernel mode. This memory region is in the kernel part of the RAM, is not accessible when in usermode, and is hence safe to use in kernel mode. When a process moves from usermode to kernel mode, the fields of the trapframe data structure are pushed onto the kernel stack to save user context. On the other hand, when the kernel is switching context from one process in kernel mode to another during a context switch, the fields of the context structure are pushed onto the kernel stack. These two structures are different, because one needs to store different sets of registers in these two situations. For example, the kernel may want to store many more registers in the trapframe to fully understand why the trap occurred, and may choose to only save a small subset of registers that really need to be saved in the context structure. While the trapframe and context structure are already on the kernel stack, the `struct proc` has explicit pointers to the trapframe and context structures on the stack, in addition to the pointer to the whole kernel stack itself, in order to easily locate these structures when needed.
- xv6 maintains an array of PCBs in a process table or `ptable`, seen at lines 2409–2412. This process table is protected by a lock—any function that accesses or modifies this process table must hold this lock while doing so. We will study the use of locks later.
- We will now study how a process handles interrupts and system calls in xv6. Interrupts are assigned unique numbers (called IRQ) on every machine. For example, an interrupt from a keyboard will get an IRQ number that is different from the interrupt from a network card. All system calls are considered as software interrupts and get the same IRQ. The interrupt descriptor table (IDT) stores information on what to do for every interrupt number, and is setup during initialization (line 3317). You need not understand the exact structure of the IDT, or how it is constructed, but it is enough to know that in a simple OS like xv6, the IDT entries for all interrupts point to a generic function to handle all traps (line 3254) that we will examine soon. That is, no matter which interrupt occurs, the kernel will start executing at this common function.
- Now, how does control shift from the user code to this all traps function in the kernel? When an interrupt / program fault / system call (henceforth collectively called a **trap**) occurs in xv6, the CPU stops whatever else it is doing and executes the `int n` instruction, with a specific interrupt number (IRQ) as argument. For example, a signal from an external I/O device can cause the CPU to execute this instruction. In the case of system calls, the user program explicitly invokes this `int n`—you may never have realized this because users almost never call system calls directly, and only call C library functions. The C library function internally invokes this `int n` instruction for you to make a system call. So, someone, either an external hardware device, or the user herself, must invoke the `int n` instruction to begin the processing of a trap event.
- What happens to the CPU when it runs this trap instruction? This special trap instruction moves

the CPU to kernel mode (if it was in user mode previously). The CPU has a *task state segment* for the current running task, that lets the CPU find the kernel stack for the current process and switch to it. That is, the ESP moves from the old stack (user or kernel) to the kernel stack of the process. As part of the execution of the `int` instruction, the CPU also saves some registers on the kernel stack (pointers to the old code segment, old stack segment, old program counter etc.), which will eventually form a part of the trapframe. Next, the CPU fetches the IDT entry corresponding to the interrupt number, which has a pointer to the kernel trap handling code. Now, the control is transferred to the kernel, and the CPU starts executing the kernel code that is responsible for handling traps.² You will not find the code for what was described in the previous few sentences in xv6. Why? Because all of this is done by the CPU hardware as part of the `int n` instruction, and not by software. That is, this logic is built into your CPU processor hardware, to be run when the special trap instruction is executed.

- It is important to note that the change in EIP (from user code to the kernel code of interrupt handler) and the change in ESP (from whatever was the old user stack to the kernel stack) must necessarily happen as part of the hardware CPU instruction, and cannot be done by the kernel software, because the kernel can start executing only on the kernel stack and once the EIP points to its code. Therefore, it is imperative that some part of the trap handling should be implemented as part of the hardware logic of the `int` instruction. Someone has to switch control to kernel before it can run, and that “someone” is the CPU here.
- The kernel code pointed at by the IDT entry (sheet 32) pushes the interrupt number onto the stack (e.g., line 3233), and completes saving various CPU registers (lines 3256-3260), with the result that the kernel stack now contains a trapframe (line 0602). Note that the trapframe has been built collaboratively by the CPU hardware and the kernel. Some parts of the trapframe (e.g., EIP) would have been pushed onto the stack even before the kernel code started to run, and the rest of the fields are pushed by this `alltraps` function in the kernel. Look at the trapframe structure on sheet 6, and understand how the bottom parts of the trapframe would have been pushed by the CPU and the top parts by the kernel. The trapframe serves two purposes: it saves the execution context that existed just before the trap (so that the process can resume where it left off), and it provides information to the kernel to handle the trap. In addition to creating a trapframe on the stack, the kernel does a few other things like setting up various segment registers to execute correctly in kernel mode (lines 3263-3268). Then the main C function to handle traps (defined at line 3350) is invoked at line 3272. But before this function is called, the stack pointer (which is also a pointer to the trapframe, since the trapframe is at the top of the stack) is pushed onto the stack, as an argument to this trap handling function.
- The main logic to handle all traps is in the C function `trap` starting at line 3350. This function takes the trapframe as an argument. This function looks at the IRQ number of the interrupt in the trapframe, and executes the corresponding action. For example, if it is a system call, the corresponding system call function is executed. If the trap is from the disk, the disk device driver is called, and so on. We will keep coming back to this function often in the course.

²Note that this switch to the kernel code segment and kernel stack need not happen if the process was already in kernel mode at the time the interrupt occurred.

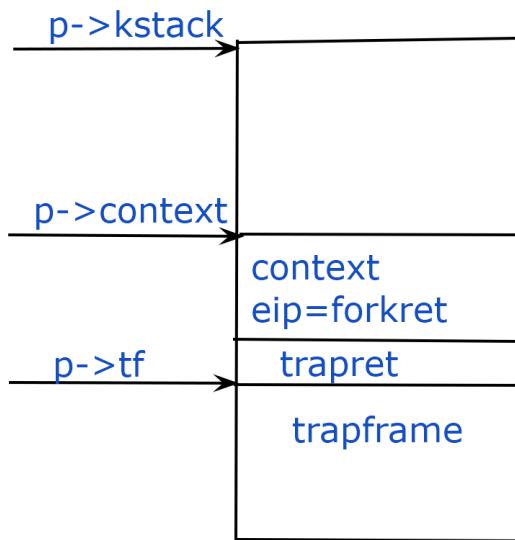
After handling the trap suitably, this trap function returns back to line 3272 from where it was invoked.

- After trap returns, it executes the logic at `trapret`, (line 3277). The registers that were pushed onto the stack are popped by the OS. Then the kernel invokes the `iret` instruction which is basically the inverse of the `int` instruction, and pops the registers that were pushed by the `int` instruction. Thus the context of the process prior to the interrupt is restored, and the userspace process can resume execution. Trap handling is complete!
- It is important to understand one particular aspect of the main trap handling function. In particular, pay attention to what happens if the interrupt was a timer interrupt. In this case, the trap function tries to yield the CPU to another process (line 3424). If the scheduler does decide to context switch away from this process, the return from trap to the userspace of the process does not happen right away. Instead, the kernel context of the process is pushed onto the kernel stack by the code that does a context switch (we will read it later), and another process starts executing. The return from trap happens after this process gets the CPU again. We will understand context switches in detail later on.
- Note that if the trap handled by a process was an interrupt, it could have lead to some blocked processes being unblocked (e.g., some process waiting for disk data). However, note that the process that serviced the interrupt to unblock a process does not immediately switch to the unblocked process. Instead, it continues execution and gives up the CPU upon a timer interrupt or when it blocks itself. It is important to note that unblocked processes do not run right away, and will wait in the ready state to be scheduled by the scheduler.
- **The contents of the kernel stack are key to understanding the workings of xv6.** When servicing an interrupt, the kernel stack has the trapframe, as we have just seen. When the process is switched out by the scheduler after servicing an interrupt, we will study later that the kernel stack has the trapframe followed by the context structure that stores context during the context switch.
- Interrupt handling in most Unix systems is conceptually similar, though more complicated. For example, in Linux, interrupt handling is split into two parts: every interrupt handler has a *top half* and a *bottom half*. The top half consists of the basic necessities that must be performed on receiving an interrupt (e.g., copy packets from the network card's memory to kernel memory), while the bottom half that is executed at a later time does the not-so-time-critical tasks (e.g., TCP/IP processing). Linux also has a separate interrupt context and associated stack that the kernel uses while servicing interrupts, instead of running on the kernel stack of the interrupted process itself.

2. Process creation via the fork system call

- xv6 supports several system calls, which are handled by the trap function we just studied. You can see the complete set of system calls on sheets 35–36. What happens during a system call? The system calls that user programs in xv6 can use are defined in the file `user.h`, which is the header file for the userspace C library of xv6 (xv6 doesn't have the standard glibc). You can find this file in the xv6 code tarball (it is not part of the kernel code PDF document because it is part of the user library of xv6, and not the kernel). User programs can invoke these library functions defined in `user.h` to make system calls. Next, you will find that the code in `usys.S` of the tarball converts the user library function calls to system calls by invoking the `int` instruction with a suitable argument indicating that the trap is due to a system call. Now, getting back to the kernel code, the main trap handling function at line 3350 looks at this argument passed to `int`, realizes that this trap is a system call (line 3353), and calls the common system call processing function `syscall1` (line 3624) in the kernel. How does the kernel know which specific system call was invoked? If you would have carefully noticed the user library code in `usys.S`, you would have seen that the system call number is pushed into the EAX register. The common `syscall` function looks at this `syscall` number and invokes the specific function corresponding to that particular system call. Note that the system calls themselves could be implemented in other places throughout the code; the entry function `syscall` is responsible for locating the suitable system call functions by storing the function pointers in an array.
- How are arguments passed to system calls? The user library stores arguments on the userspace stack before invoking the `int` instruction. The system call functions locate the userspace stack (from the ESP stored in the trapframe), and fetch the system call arguments from there. You can also find the various helper functions to parse system call arguments on sheet 35.
- We will now study the system call used for process creation. When a process calls `fork`, the generic trap handling function calls the specific system call `fork` function (line 2554). `Fork` calls the `allocproc` function (line 2455) to allocate an unused proc data structure. These two functions will now be discussed in detail.
- The `allocproc` function allocates a new unused `struct proc` data structure, and initializes it (lines 2460-2465). It also allocates a new kernel stack for the process (line 2473). On the kernel stack of a new process, it pushes various things, beginning with a trapframe. We will first see what goes into the trapframe of the new process. Once `allocproc` returns, the `fork` function copies the parent's trapframe onto the child's trapframe (line 2572). As a result, both child and the parent have the same user context stored at the top of the stack during trapret, and hence return to the same point in the user program, right after the `fork` system call. The only difference is that the `fork` function zeroes out the EAX register in the child's trapframe (line 2575), which is the return value from `fork`, causing the child to return with the value 0. On the other hand, the system call in the parent returns with the pid of the child (line 2591).
- In addition to the trapframe, the `allocproc` function pushes the address of trapret on the stack (line 2486), followed by the context structure (line 2488). That is, the function starts at the bottom of the kernel stack in the beginning (line 2477), and gradually moves upwards to make

space for a trapframe, a return address of trapret, and a context structure on the stack. So, when you start popping the kernel stack from top, you will first find the context structure, followed by a return address of trapret, followed by the trapframe. Below is how the kernel stack of the child will look like after allocproc.



- Let us first understand the purpose of the context structure on the kernel stack of the child. We will study more on this structure later, but for now, you should know that processes have this context data structure pushed onto the kernel stack when they are being switched out by the CPU scheduler, so that the contents in this data structure can be used to restore context when the process has to resume again. For example, this context structure saves the EIP at which a process stopped execution just before the context switch, so that it can resume execution again from the same EIP where it stopped. For a new-born process that was never context switched out, a context structure does not make sense. However, allocproc creates an artificial context structure on the kernel stack to make this process appear like it was context switched out in the past, so that when the CPU scheduler sees this process, it can start scheduling this new process like any other process in the system that it had context switched out in the past. This neat little trick means that once a new process is created and added to the list of active processes in the system, the CPU scheduler can treat it like any other process.
- Now, what should the value of the EIP in this context structure be set to? Where do we want our new process to begin execution when the CPU scheduler restores this context and resumes its execution? The instruction pointer in this (artificially created) context data structure is set (line 2491) to point to a piece of code called `forkret`, which is where this process starts executing when the scheduler swaps it in. The function `forkret` (line 2783) does a few things which we will study later, and returns. When `forkret` returns, the process continues to execute at `trapret` because the return address of `trapret` is now present at the top of the stack (after the context structure has been popped off the stack when restoring context by the CPU scheduler),

and we know that the address at the top of the stack is used as the return address when returning from a function call. That is, in effect, a newly created process, when it is scheduled on the CPU for the first time, simply behaves as if it is returning from a trap.

- We have seen earlier that the trapret function simply pops the trapframe from the kernel stack, restores userspace context and returns the process to usermode again. The child's trapframe is an exact copy of the parent's trapframe, and the parent's trapframe stores the userspace context of the process that stopped execution at the fork system call. So, when the child process resumes, it returns to the exact same instruction after the fork system call, much like the parent, with only a different return value. Now you should be able to understand why the parent and child processes resume execution at the same line after a fork system call, and the effort that has gone in to make this appear so easy.
- The fork system call does many other things like copying the user space memory image from parent to child and other initializations. We will study these later.

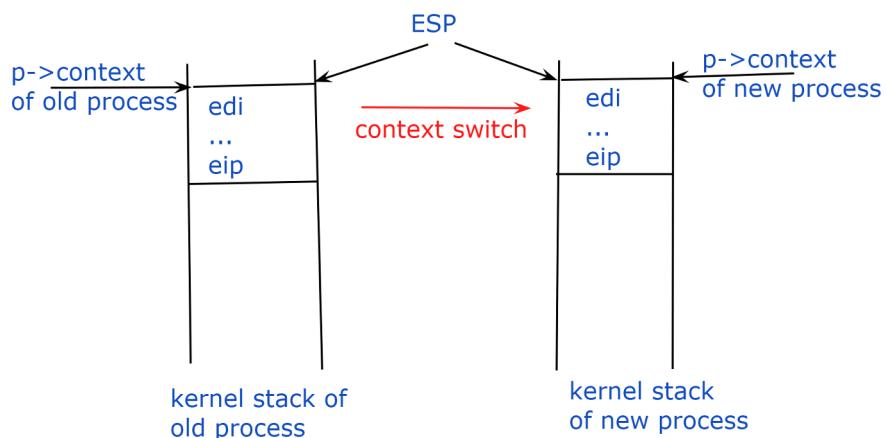
3. Process Scheduling

- xv6 uses the following names for process states (line 2350): UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE. The `proc` structures are all stored in the `ptable` array.
- We will now study how the CPU scheduler works in xv6 (pages 27–29). Every CPU in the machine starts a scheduler thread (think of it as a special process) that finishes some basic bootup activities and calls the `scheduler` function (line 2708) when the machine comes up. This scheduler thread loops in this function forever, finding active processes to run and context switching to them. The job of the scheduler function is to look through the list of processes, find a runnable process, set its state to RUNNING, and switch to the process. We will understand this process in detail.
- First, note that all actions on the list of processes are protected by `ptable.lock`. Any process that wishes to change this data structure must hold the lock, to avoid race conditions (which we will study later). What happens if the lock is not held? It is possible that two CPUs find a process to be runnable, and switch it to running state simultaneously, causing one process to run at two places. So, all actions of the `scheduler` function are always done with this lock held. Always remember: one must acquire the lock, change the process table, and release the lock.
- During a context switch, every process has a `context` structure on its stack (line 2343), that stores the values of the CPU registers that must be saved during a context switch. The registers that are saved as part of the context include the EIP (so that execution can resume at the instruction where it stopped), and a few other general purpose registers. To switch the CPU from the current running process P1 to another process P2, the registers of P1 are saved in its context structure on its stack, and the registers of P2 are reloaded from its context structure (that would have been saved in the past when P2 was switched out). Now, when P1 must be resumed again, its registers are reloaded from its context, and it can resume execution where it left off.
- The `swtch` function (line 2950) does this job of switching between two contexts, and old one and a new one. Let us carefully study this process of context switching. Understanding this function needs some basic knowledge of x86 assembly.
 - For example, consider the scenario when the scheduler finds a process to run and decides to context switch to it (line 2728). At this point, it calls the `swtch` function, providing to it two arguments: a pointer to the context structure of the scheduler thread itself (where context must be saved), and a pointer to the context structure of the new process to run (from where context should be restored), which is available on the kernel stack of the new process.
 - Next, look at the code of `swtch` starting at line 2950. This code is in assembly language, so we need to understand the C calling conventions to understand what is on the stack when this function starts. The top of the stack will have the return address from where `swtch` was invoked, and below this return address would be the two arguments: the old context pointer and the new context pointer. The function first saves the old context pointer and

the new context pointer into two registers (lines 2959, 2960). That is, EAX has the old context pointer and EDX has the new context pointer.

- Right now, we are still operating on the stack of the old process from which we wish to move away. Next, the function pushes some registers onto this old stack (lines 2963-2966). Note that because this function is the callee, we are only saving the callee-save registers (the other caller-save registers would have been saved by the compiled C code of the caller). The registers pushed by this code, along with the EIP (i.e., the return address pushed on the stack when the function call to swtch was made), together will form a context structure on the stack.
- After pushing the context onto the old stack, the stack pointer still points to the top of the old stack, and this top of the stack contains the context structure. The pointer to the old context (which was saved in EAX) is now updated to point to this saved context at the top of the stack (line 2969). Notice the subtlety around the indirect addressing mode in line 2969 (there are parentheses around EAX indicating that you don't store anything directly into EAX but rather at the address pointed by EAX). That is, you are not writing anything into EAX, rather, you are going to the memory location pointed at by EAX (the old context pointer), and overwriting it with the updated context pointer at the top of the stack.
- Having saved the old context, the stack pointer now shifts to point to the top of the stack of the new process, which has a new context structure to restore. Recall that we saved this new context structure pointer in EDX. So, this value of EDX is moved to ESP (line 2970). The stack pointer has now switched from the stack of the old process to that of the new process. A context switch has happened!
- Next, we pop the registers of the context structure from the top of the new stack (lines 2973-2976), and return from swtch into the kernel mode of a new process.

The process of this context switch is illustrated in this figure below.



- Note that, in the swtch function, the step of pushing registers into the old stack is exactly similar to the step of restoring registers from the new stack, because the new stack was also created by

`swtch` at an earlier time. The only time when we switch to a `context` structure that was not pushed by `swtch` is when we run a newly created process. For a new process, `allocproc` writes the context onto the new kernel stack (lines 2488-2491), which will be loaded into the CPU registers by `swtch` when the process executes for the first time. All new processes start at `forkret`, so `allocproc` writes this address into the EIP of the context it is creating. Except in this case of a new process, the EIP stored in `context` is always the address at which the running process invoked `swtch`, and it resumes at the same point at a later time. Note that `swtch` does not explicitly store the EIP to point to the address of the `swtch` statement, but the EIP (return address) is automatically pushed on the stack as part of making the function call to `swtch`.

- In xv6, the scheduler runs as a separate thread with its own stack. Therefore, context switches happen from the kernel mode of a running process to the scheduler thread, and from the scheduler thread to the new process it has identified. (Other operating systems can switch directly between kernel modes of two processes as well.) As a result, the `swtch` function is called at two places: by the scheduler (line 2728) to switch from the scheduler thread to a new process, and by a running process that wishes to give up the CPU in the function `sched` (line 2766) and switch to the scheduler thread. A running process always gives up the CPU at this call to `swtch` in line 2766, and always resumes execution at this point at a later time. The only exception is a process running for the first time, that never would have called `swtch` at line 2766, and hence never resumes from there.
- A process that wishes to relinquish the CPU calls the function `sched` (line 2753). This function triggers a context switch, and when the process is switched back in at a later time, it resumes execution again in `sched` itself. Thus a call to `sched` freezes the execution of a process temporarily. When does a process relinquish its CPU in this fashion? For example, when a timer interrupt occurs and it is deemed that the process has run for too long, the trap function calls `yield`, which in turn calls `sched` (line 2776). The other cases are when a process exits or blocks on a system call. We will study these cases in more detail later on.

4. Boot procedure

- We will now understand how xv6 boots up. The first part of the boot process concerns itself with setting up the kernel code in memory, setting up suitable page tables to translate the kernel’s memory addresses, and initializing devices (sheet 12), and we will study this in detail later. After the kernel starts running, it starts setting up user processes, starting with the first `init` process in the `userinit` function (line 2502). We will understand the boot procedure from the creation of init process onwards.
- To create the first process, the `allocproc` function is used to allocate a new proc structure, much like in `fork` (line 2507). The memory image of the new process is initialized with the compiled code of the `init` program in line 2511 (we will study this later). We have seen earlier that `allocproc` builds the kernel stack of this new process in such a way that it runs `forkret`, and then begins returning from trap. What will happen when this new process returns from trap? The trapframe of this `init` process is hand-created (lines 2514-2520) to look like the process encountered a trap right on the first instruction in its memory (i.e., the EIP saved in the trapframe is address 0). As a result, when this process returns from trap, its context is restored from the trapframe, and it starts executing at the start of the userspace `init` program (sheet 83).
- The `init` process sets up the first three file descriptors (`stdin`, `stdout`, `stderr`), and all point to the console. All subsequent processes that are spawned inherit these descriptors. Next, it forks a child, and `exec`’s the shell executable in the child. While the child runs the shell (and forks various other processes), the parent waits and reaps zombies.
- The shell program (sheets 83–88, main function at line 8501) gets the user’s input, parses it into a command (`parsecmd` at line 8718), and runs it (`runcmd` at line 8406). For commands that involve multiple sub-commands (e.g., list of commands or pipes), new children are forked and the function `runcmd` is called recursively for each subcommand. The simplest subcommands will eventually call `exec` when the recursion terminates. Also note the complex manipulations on file descriptors in the pipe command (line 8450).

Lecture Notes on Operating Systems

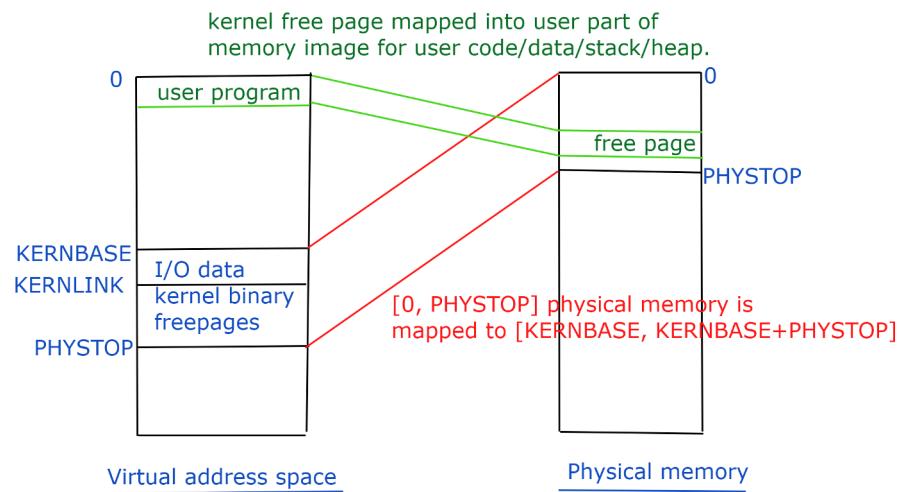
Mythili Vutukuru, Department of Computer Science and Engineering, IIT Bombay

Memory Management in xv6

1. Basics

- xv6 uses 32-bit virtual addresses, resulting in a virtual address space of 4GB. xv6 uses paging to manage its memory allocations. However, xv6 does not do demand paging, so there is no concept of virtual memory. That is, all valid pages of a process are always allocated physical pages.
- xv6 uses a page size of 4KB, and a two level page table structure dictated by the underlying x86 hardware. The CPU register CR3 contains a pointer to the outer page directory of the current running process. The translation from virtual to physical addresses is performed by the x86 MMU as follows. The first 10 bits of a 32-bit virtual address are used to index into the page table directory, which provides an address of the inner page table. The next 10 bits index into the inner page table to locate the page table entry (PTE). The PTE contains a 20-bit physical frame number and flags. Every page table in xv6 has mappings for user pages as well as kernel pages. The part of the page table dealing with kernel pages is the same across all processes. Sheet 8 contains various definitions pertaining to the page table and page table entries. You can find macros to extract specific bits (e.g., index into page table) from a virtual address here, which will be useful when understanding code. You can also find the various flags used to set permissions in the page table entry defined here.
- Sheets 02 and 18 describe the memory layout of xv6. In the virtual address space of every process, the user code+data, heap, stack and other things start from virtual address 0 and extend up to KERNBASE. The kernel is mapped into the address space of every process beginning at KERNBASE. The kernel code and data begin from KERNBASE, and can go up to KERNBASE+PHYSTOP. This virtual address space of [KERNBASE, KERNBASE+PHYSTOP] is mapped to [0,PHYSTOP] in physical memory. In the current xv6 code, KERNBASE is set to 2GB and PHYSTOP is set to 224MB. Because KERNBASE+PHYSTOP can go to a maximum of 4GB in 32-bit architectures, and KERNBASE is 2GB, the maximum possible value of PHYSTOP is 2GB, so xv6 can use a maximum of 2GB physical memory. You can also find here various macros like V2P that translates from a virtual address to a physical address (by simply subtracting KERNBASE from the virtual address).
- The kernel code doesn't exactly begin at KERNBASE, but a bit later at KERNLINK, to leave some space at the start of memory for I/O devices. Next comes the kernel code+read-only data from the kernel binary. Apart from the memory set aside for kernel code and I/O devices, the remaining memory is in the form of free pages managed by the kernel. When any user process requests for memory to build up its user part of the address space, the kernel allocates memory to the user process from this free space list. That is, most physical memory can be mapped twice, once into the kernel part of the address space of a process, and once into the user part.

- The memory layout described above is illustrated below.



2. Initializing the memory subsystem

- The xv6 bootloader loads the kernel code in low physical memory (starting at 1MB, after leaving the first 1MB for use by I/O devices), and starts executing the kernel at `entry` (line 1040). Initially, there are no page tables or MMU, so virtual addresses must be the same as physical addresses. So this kernel entry code resides in the lower part of the virtual address space, and the CPU generates memory references in the low virtual address space only, which are passed onto the memory hardware as-is and used as physical addresses also. Now, the kernel must turn on MMU and start executing at high virtual addresses, in order to make space for user code at low virtual addresses. We will first see how the kernel jumps to high virtual addresses.
- The entry code first turns on support for large pages (4MB), and sets up the first page table `entryptpgdir` (lines 1311-1315). The second entry in this page table is easier to follow: it maps [KERNBASE, KERNBASE+4MB] to [0, 4MB], to enable the first 4MB of kernel code in the high virtual address space to run after MMU is turned on. The first entry of this page table maps virtual addresses [0, 4MB] to physical addresses [0,4MB], to enable the entry code that resides in the low virtual address space to run. Once a pointer to this page table is stored in CR3, MMU is turned on. From this point onwards, virtual addresses in the range [KERNBASE, KERNBASE+4MB] are correctly translated by MMU. Now, the entry code creates a stack, and jumps to the `main` function in the kernel's C code (line 1217). This function and the rest of the kernel is linked to run at high virtual addresses, so from this point on, the CPU fetches kernel instructions and data at high virtual addresses. All this C code in high virtual address space can run because of the second entry in `entryptpgdir`. So why was the first page table entry required? To enable the few instructions between turning on MMU and jumping to high address space to run correctly. If the entry page table only mapped high virtual addresses, the code that jumps to high virtual addresses of `main` would itself not run (because it is still in low virtual address space).
- Remember that once the MMU is turned on, all memory accesses must go through the MMU. So, for any memory to be usable, the kernel must assign a virtual address for that memory and a page table entry to translate that virtual address to a physical address must be present in the page table/MMU. That is, for any physical memory address N to be usable by the kernel, there must exist a page table entry that translates virtual address KERNBASE+ N to physical address N . When `main` starts, it is still using `entryptpgdir` which only has page table mappings for the first 4MB of kernel addresses, so only this 4MB is usable. If the kernel wants to use more than this 4MB, it needs a larger page table to hold more entries. So, the `main` function of the kernel first creates some free pages in this 4MB in the function `kinit1` (line 3030), and uses these freepages to allocate a bigger page table for itself. This function `kinit1` in turn calls the functions `freerange` (line 3051) and `kfree` (line 3065). Both these functions together populate a list of free pages for the kernel to start using for various things, including allocating a bigger, nicer page table for itself that addresses the full memory.
- The kernel uses the `struct run` (line 3014) data structure to track a free page. This structure simply stores a pointer to the next free page, and is stored within the page itself. That is, the list of free pages are maintained as a linked list, with the pointer to the next page being stored

within the page itself. The kernel keeps a pointer to the first page of this free list in the structure `struct kmem` (lines 3018-3022). Pages are added to this list upon initialization, or on freeing them up. Note that the kernel code that allocates and frees pages always returns the virtual address of the page in the kernel address space, and not the actual physical address. The `V2P` macro is used when one needs the physical address of the page, say to put into the page table entry.

- After creating a small list of free pages in the 4MB space, the kernel main function (sheet 12) proceeds to build a bigger page table to map all its address space in the function `kvmalloc` (line 1857). This function in turn calls `setupkvm` (line 1837) to setup the kernel page table, and switches to it (by writing the address of the page table into the CR3 register). The address space mappings that are setup by `setupkvm` can be found in the structure `kmap` (lines 1823-1833). `kmap` contains the mappings from virtual to physical address for various virtual address ranges: the small space at the start of memory for I/O devices, followed by kernel code+read-only data (kernel binary), followed by other kernel data, all the way from `KERNBASE` to `KERNBASE+PHYSTOP`. Note that the kernel code/data and other free physical memory is already lying around at the specified physical addresses in RAM, but the kernel cannot access it because all of that physical memory has not been mapped into any page tables yet and there are no page table entries that translate to these physical addresses at the MMU.
- The function `setupkvm` works as follows. It first allocates an outer page directory. Then, for each of the virtual to physical address mappings in `kmap`, it calls `mappages`. The function `mappages` (line 1779) is given a virtual address range and a physical address range it should map this to. It then walks over the virtual address range in 4KB page-sized chunks, and for each such logical page, it locates the PTE corresponding to this virtual address using the `walkpgdir` function (line 1754). `walkpgdir` simply emulates the page table walking that the MMU would do. It uses the first 10 bits to index into the page table directory to find the inner page table. If the inner page table does not exist, it requests the kernel for a free page, and initializes the inner page table. Note that the kernel has a small pool of free pages setup by `kinit1` in the first 4MB address space—these free pages are returned here and used to construct the kernel’s page table. Once the inner page table is located (either existing already or newly allocated), `walkpgdir` uses the next 10 bits to index into it and return the PTE. Once `walkpgdir` returns the PTE, `mappages` writes the appropriate mapping in the PTE using the physical address range given to it. (Sheet 08 has the various macros that are useful in getting the index into page tables from the virtual address.)
- Note that `walkpgdir` simply returns the page table entry corresponding to a virtual address in a page table. That is, it uses the first 10 bits of a virtual address as an index in the page directory to find the inner page table, then uses the next 10 bits as an index in this inner page table to find the actual PTE. What if the inner page table corresponding to the address does not exist? The last argument to the function specifies if a page table should be allocated if one doesn’t exist. That is, this function `walkpgdir` serves two purposes. It can simple be used to look up a virtual address in an existing page table and return whatever PTE exists. It can also be used to construct the page table entry if one doesn’t exist. In this case, since we are initializing the page table, we use `walkpgdir` to allocate inner page tables. When it allocates

an inner page table, the PTE returned by `walkpgdir` doesn't hold any valid information as such. The function `mappages` takes this empty PTE returned by `walkpgdir` and writes the correct physical address into it.

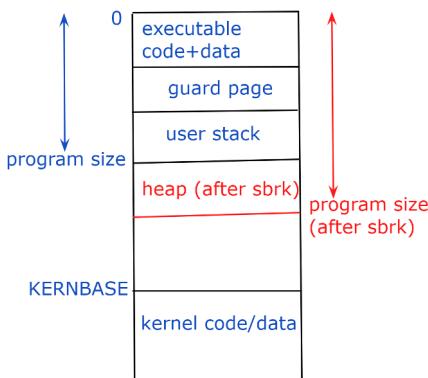
- After the kernel page table `kpgdir` is setup this way (line 1859), the kernel switches to this page table by storing its address in the CR3 register in `switchkvm` (line 1866). From this point onwards, the kernel can freely address and use its entire address space from KERNBASE to KERNBASE+PHYSTOP.
- Let's return back to main in line 1220. The kernel now proceeds to do various initializations. It also gathers many more free pages into its free page list using `kinit2`, given that it can now address and access a larger piece of memory. At this point, the entire physical memory at the disposal of the kernel. All usable physical memory [0, PHYSTOP] is mapped by `kpgdir` into the virtual address space [KERNBASE, KERNBASE+PHYSTOP], so all memory can be addressed by virtual addresses and translated by MMU. This memory consists of the kernel code/data that is currently executing on the CPU, and a whole lot of free pages in the kernel's free page list. Now, the kernel is all set to start user processes, starting with the init process (line 1239).

3. Memory management of user processes

- The function `userinit` (line 2502) creates the first user process. We will examine the memory management in this function. The kernel page table of this process is created using `setupkvm` as always. For the user part of the memory, the function `inituvm` (line 1903) allocates one physical page of memory, copies the init executable into that memory, and sets up a page table entry for the first page of the user virtual address space. When the init process runs, it executes the init executable (sheet 83), whose main function forks a shell and starts listening to the user. Thus, in the case of init, setting up the user part of the memory was straightforward, as the executable fit into one page.
- All other user processes are created by the `fork` system call. In `fork` (line 2554), once a child process is allocated, its memory image is setup as a complete copy of the parent's memory image by a call to `copyuvm` (line 2053). This function first sets up the kernel part of the page table. Then it walks through the entire address space of the parent in page-sized chunks, gets the physical address of every page of the parent using a call to `walkpgdir`, allocates a new physical page for the child using `kalloc`, copies the contents of the parent's page into the child's page, adds an entry to the child's page table using `mappages`, and returns the child's page table. At this point, the entire memory of the parent has been cloned for the child, and the child's new page table points to its newly allocated physical memory.
- Next, look at the implementation of the `sbrk` system call (line 3701). This system call can be invoked by a process to grow/shrink the userspace part of the memory image. For example, the heap implementation of `malloc` can use this system call to grow/shrink the heap when needed. This system call invokes the function `growproc` (line 2531), which uses the functions `allocuvm` or `deallocuvm` to grow or shrink the virtual memory image. The function `allocuvm` (line 1953) walks the virtual address space between the old size and new size in page-sized chunks. For each new logical page to be created, it allocates a new free page from the kernel, and adds a mapping from the virtual address to the physical address by calling `mappages`. The function `deallocuvm` (line 1982) looks at all the logical pages from the (bigger) old size of the process to the (smaller) new size, locates the corresponding physical pages, frees them up, and zeroes out the corresponding PTE as well.
- Next, let's understand the `exec` system call. If the child wants to execute a different executable from the parent, it calls `exec` right after `fork`. For example, the init process forks a child and execs the shell in the child. The `exec` system call copies the binary of an executable from the disk to memory and sets up the user part of the address space and its page tables. In fact, after the initial kernel is loaded into memory from disk and the init process is setup, all subsequent executables are read from disk into memory via the `exec` system call alone.
- `Exec` (line 6310) first reads the ELF header of the executable from the disk and checks that it is well formed. (There is a lot of disk I/O related code here that you can ignore for now.) It then initializes a page table, and sets up the kernel mappings in the new page table via a call to `setupkvm` (line 6334). Then, it proceeds to build the user part of the memory image via calls to `allocuvm` (line 6346) and `loaduvm` (line 6348) for each part of the binary executable (an

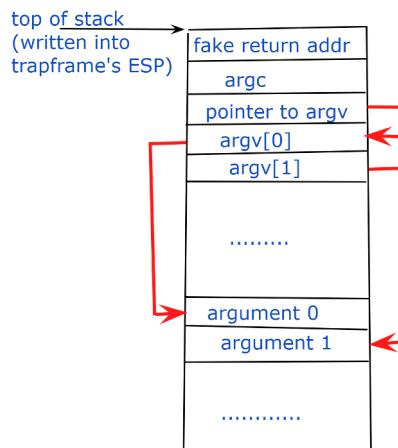
ELF binary is composed of many segments). We have already seen that `allocuvm` (line 1953) allocates physical pages from the kernel's free pool via calls to `kalloc`, and sets up page table entries, thereby expanding the virtual/physical address space of the process. `loaduvm` (line 1918) reads the memory executable from disk into the newly allotted memory page using the disk I/O related `readi` function. After the end of the loop of calling these two functions for each segment, the complete program executable has been loaded from disk into memory, and page table entries have been setup to point to it. However, `exec` is still using the old page table, and hasn't switched to this new page table yet. We have only constructed a new memory image and a page table pointing to it so far, but the process that called `exec` is still executing on the old memory image.

- The new memory image so far has only the code/data present in the executable. Next, `exec` goes on to build the rest of its new memory image. For example, it allocates two pages for the userspace stack of the process. The second page is the actual stack and the first page serves as a guard page. The guard page's page table entry is modified to make it as inaccessible by user processes, so any access beyond the stack into the guard page will cause a page fault. (Recall that the stack grows upwards towards lower memory addresses, so it will overflow into the page before it.) After the stack in the memory image is where the heap should be located. However, xv6 doesn't allocate any heap memory upfront. The user library code that handles `malloc` will call `sbrk` to grow the memory image, and use this new space as the heap, as and when memory is requested by user programs. The program size includes all the memory from address zero until the end of the stack now, and the size will increase to include any memory allocated by `sbrk` when the currently empty heap expands. The new memory image constructed by `exec` is shown below.



- After allocating the user stack, the arguments to `exec` are pushed onto the user stack (lines 6363-6380). The arguments passed to the `exec` system call are already made available as arguments to the `exec` function on sheet 63. We must now copy these arguments onto the newly created userspace stack of the process. Why? Because when the main function of the new executable starts, it expects to find arguments `argc` and `argv` on the user stack.

- If you are curious, below is an explanation of this process of preparing the user stack (lines 6363-6380) in more detail. Recall the structure of the stack when any C function is called: the top of the stack has the return address, followed by the arguments passed to the function below it. We must now prepare this new userspace in this manner for the main function as well. The top of the user stack has a fake return address (line 6374) because main doesn't really return anywhere. Next, the stack has the number of arguments (argc), followed by a pointer to the argv array. Next on the stack are the actual contents of the argv array (which is of size argc). The element i of the array argv contains a pointer to the i -th argument (which can be any random string). Finally, after the array of pointers to arguments, the actual arguments themselves are also present on the stack. We will now see how this structure is constructed. We start at the bottom of the stack, and start pushing the actual arguments on the stack (lines 6363-6371). In the process, we also remember where the argument was pushed, i.e., we store the pointers to the arguments in the array ustack. Finally, we will write out the other things that go above the arguments on the stack: the return PC, argc, pointer to argv, as well as the contents of argv (stored pointers to the arguments). Note that all of these things have to be written into the user stack of the new memory image, not on the current memory image where the process is running. (How does a process access another memory image that is not its own? Note the use of the function copyout which simply copies specific content at the specified virtual address of the new memory image defined by the new page table.) This user stack structure is illustrated below.



- It is important to note that `exec` does not replace/reallocate the kernel stack. The `exec` system call only replaces the user part of the memory image, and does nothing to the kernel part. And if you think about it, there is no way the process can replace the kernel stack, because the process is executing in kernel mode on the kernel stack itself, and has important information like the trap frame stored on it. However, it does make small changes to the trap frame on the kernel stack, as described below.
- Recall that a process that makes the `exec` system call has moved into kernel mode to service the software interrupt of the system call. Normally, when the process moves back to user mode

again (by popping the trap frame on the kernel stack), it is expected to return to the instruction after the system call. However, in the case of `exec`, the process doesn't have to return to the instruction after `exec` when it gets the CPU next, but instead must start executing the new executable it just loaded from disk. So, the code in `exec` changes the return address in the trap frame to point to the entry address of the binary (line 6392). It also sets the stack pointer in the trap frame to point to the top of the newly created user stack. Finally, once all these operations succeed, `exec` switches page tables to start using the new memory image (line 6394). That is, it writes the address of the new page table into the CR3 register, so that it can start accessing the new memory image when it goes back to userspace. Finally, it frees up all the memory pointed at by the old page table, e.g., pages containing the old userspace code, data, stack, heap and so on. At this point, the process that called `exec` can start executing on the new memory image when it returns from trap. Note that `exec` waits until the end to do this switch of page tables, because if anything went wrong in the system call, `exec` returns from trap into the old memory image and prints out an error.

The `.bochsrc` file in the xv6 code is a configuration file used by the Bochs IA-32 emulator. Bochs is a software emulation program that emulates an entire PC system, including the CPU, memory, disk, and other components. The `.bochsrc` file specifies the settings for the Bochs emulator, such as the amount of memory, the hard drive image to use, the boot order, and so on.

In the xv6 code, the `.bochsrc` file is used to specify the configuration for booting the xv6 operating system in the Bochs emulator. The file contains several options and settings, such as:

- `megs: 32` - This specifies the amount of memory to allocate for the virtual machine. In this case, 32MB of memory is allocated for the xv6 operating system.
- `romimage: file=/usr/local/share/bochs/BIOS-bochs-latest` - This specifies the BIOS image to use. In this case, it points to the latest BIOS image for the Bochs emulator.
- `vgaromimage: /usr/local/share/bochs/VGABIOS-Igpl-latest` - This specifies the VGA BIOS image to use.
- `ata0: enabled=1, ioaddr1=0x1f0, ioaddr2=0x3f0, irq=14` - This specifies the configuration for the first ATA device, which is used to boot the xv6 operating system.
- `boot: disk` - This specifies that the xv6 operating system should be booted from the disk image.

In summary, the `.bochsrc` file in the xv6 code is used to specify the configuration for booting the xv6 operating system in the Bochs emulator, including the amount of memory, the BIOS and VGA BIOS images, the ATA device configuration, and the boot order.

2.

`xv6. img` is a disk image file that contains the xv6 operating system. A disk image file is a complete copy of a disk that contains an operating system or other data. It is typically used to boot a virtual machine or to restore a system to a previous state.

In the case of xv6, `xv6. img` is a pre-built disk image file that contains a fully functional version of the xv6 operating system. This file is used to boot a QEMU virtual machine that runs the xv6 operating system. The file is typically created using tools like `mkfs` and `dd`, which are used to format and copy the files onto a virtual disk.

When the virtual machine is started using the `qemu` command with the `-drive` option that points to `xv6. img`, QEMU loads the xv6 operating system from the disk image file and boots it up as if it were running on a physical machine. This allows users to test, modify, and experiment with the xv6 operating system in a safe and isolated environment.

3.

The commands you provided are used to create the `xv6. img` disk image file, which is used to boot the xv6 operating system. Here's a breakdown of each command:

1. `dd if=/dev/zero of=xv6. img count=10000`: This command creates a new disk image file called `xv6. img` and fills it with zeros. The `count=10000` option specifies that the file should be 10000 blocks (or 5MB) in size.
2. `dd if=bootblock of=xv6. img conv=notrunc`: This command writes the `bootblock` file to the beginning of the `xv6. img` disk image file. The `if=bootblock` option specifies the input file to use, while the `of=xv6. img` option specifies the output file. The `conv=notrunc` option tells `dd` not to truncate the output file, which means that any existing data in the file will not be overwritten.
3. `dd if=kernel of=xv6. img seek=1 conv=notrunc`: This command writes the `kernel` file to the `xv6. img` disk image file, starting from the second block (since the first block is already occupied by the `bootblock`). The `if=kernel` option specifies the input file to use, while the `of=xv6. img`

option specifies the output file. The `seek=1` option tells `dd` to skip the first block and start writing from the second block. The `conv=notrunc` option tells `dd` not to truncate the output file, as before.

Overall, these commands create a disk image file called `xv6.img` and write the `bootblock` and `kernel` files to it. The resulting file is a complete copy of a disk that contains the xv6 operating system, ready to be booted by a virtual machine or a physical machine.

4. BOOTBLOCK (BOOTASM, BOOTMAIN)

These lines of code are part of a Makefile rule that compiles a boot loader program for a computer system. Here's what each line does:

1. `bootblock: bootasm.S bootmain.c` - This line specifies the target "bootblock" and its dependencies, which are the source code files "bootasm.S" and "bootmain.c". This means that in order to create the "bootblock" binary, these source files must be compiled.
2. `$(CC) $(CFLAGS) -fno-pic -O -nostdinc -I. -c bootmain.c` - This line invokes the C compiler (`$(CC)`) with the specified flags (`$(CFLAGS)`) to compile the "bootmain.c" source file. The flags used are `-fno-pic` (don't use position-independent code), `-O` (optimize the code), `-nostdinc` (don't include standard system headers), `-I.` (add the current directory to the include search path), and `-c` (generate an object file, rather than an executable).
3. `$(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c bootasm.S` - This line is similar to the previous line, but compiles the "bootasm.S" assembly source file.
4. `$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o bootblock.o bootasm.o bootmain.o` - This line invokes the linker (`$(LD)`) with the specified flags (`$(LDFLAGS)`) to link the object files generated in the previous steps into a single binary file called "bootblock.o". The flags used are `-N` (set the text and data sections to be writable and executable), `-e start` (set the entry point to the "start" label), `-Ttext 0x7C00` (set the starting address of the program to 0x7C00), and `-o bootblock.o` (output the linked binary to a file called "bootblock.o").

5. `$(OBJDUMP) -S bootblock.o > bootblock.asm` - This line disassembles the "bootblock.o" binary into assembly code and saves it to a file called "bootblock.asm".
6. `$(OBJCOPY) -S -O binary -j .text bootblock.o bootblock` - This line copies the "text" section of the "bootblock.o" binary into a new binary file called "bootblock".
7. `./sign.pl bootblock` - This line runs a script called "sign.pl" with the "bootblock" binary as an argument. The purpose of this script is not clear from the given code, but it may be used to add a digital signature to the binary or perform some other post-processing step.

5. Kernel

The code you provided is a Makefile rule for building a kernel program. Here's what each line does:

1. `kernel: $(OBJS) entry.o entryother initcode kernel.1d` - This line specifies the target "kernel" and its dependencies, which are the object files listed in the variable `$(OBJS)`, as well as the "entry.o", "entryother", "initcode", and "kernel.1d" files. This means that in order to create the "kernel" binary, these files must be linked together.
2. `$(LD) $(LDFLAGS) -T kernel.1d -o kernel entry.o $(OBJS) -b binary initcode entryother` - This line invokes the linker (`$(LD)`) with the specified flags (`$(LDFLAGS)`) to link the object files generated in previous compilation steps into a single binary file called "kernel". The flags used are `-T kernel.1d` (use the linker script file "kernel.1d"), `-o kernel` (output the linked binary to a file called "kernel"), `-b binary` (treat the "initcode" and "entryother" files as binary files), and `entry.o $(OBJS)` (specify the object files to link).
3. `$(OBJDUMP) -S kernel > kernel.asm` - This line disassembles the "kernel" binary into assembly code and saves it to a file called "kernel.asm".
4. `$(OBJDUMP) -t kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > kernel.sym` - This line generates a symbol table for the "kernel" binary and saves it to a file called "kernel.sym". The `$(OBJDUMP)` command generates a full symbol table, but the `sed` command filters out unnecessary information and formatting. Specifically, it

deletes everything before the "SYMBOL TABLE" header, removes the second column of each line (which contains unnecessary information), and deletes any empty lines. The resulting symbol table lists the names and addresses of all global symbols defined in the "kernel" binary.

6. fs.img, xv6.img

`fs. img` and `xv6. img` are disk images used by the XV6 operating system.

`xv6. img` is the bootable disk image that contains the XV6 kernel and file system. It is created by combining the bootloader (bootblock) and kernel (kernel) into a single image. This disk image is loaded into a virtual machine or written to a physical disk in order to boot the XV6 operating system.

`fs. img` is a separate disk image used to store the file system data for XV6. It is created separately from `xv6. img` and contains the initial file system used by the operating system. The file system contains the root directory and a few other initial directories and files, which can be modified and extended by user programs.

Both `xv6. img` and `fs. img` are binary files that represent the content of a disk, including its partition table, boot sector, and file system data. They can be modified using disk editing tools, such as `dd`, to change the content of the disk, create new partitions, or copy data from one disk to another.

7. compiling userland programs

The code you provided is a makefile rule that describes how to build an executable binary file from a C source file, with the help of some object files (`ulib.o`, `usys.o`, `printf.o`, `umalloc.o`).

Here is a breakdown of what each line does:

- `_%: %.o $(ULIB)`: This line specifies a target that matches any file name ending with an underscore. The dependencies of this target

are the corresponding `.o` file (the C source file compiled into an object file) and the `ULIB` object files.

- `$(LD) $(LDFLAGS) -N -e main -Ttext 0 -o $@ $^`: This line links the object files into a binary executable using the linker (`ld`). The `-N` option specifies that no default values should be used for uninitialized data, `-e main` specifies that the entry point of the program should be the `main` function, `-Ttext 0` specifies the starting address of the program, and `-o $@` specifies the output file name. The `$^` variable expands to a list of all the dependencies (the `.o` files and `ULIB` object files), and `$@` expands to the name of the target.
- `$(OBJDUMP) -S $@ > $*.asm`: This line creates a disassembly listing of the executable using the `objdump` tool, and writes it to a file with the same name as the target, but with a `.asm` extension.
- `$(OBJDUMP) -t $@ | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' > $*.sym`: This line creates a symbol table of the executable using the `objdump` tool, pipes it to the `sed` command to remove unnecessary information, and writes it to a file with the same name as the target, but with a `.sym` extension.

In summary, this makefile rule describes how to build an executable binary file from a C source file and some object files, and also creates a disassembly listing and a symbol table of the resulting executable.

8. Compiling cat command

This set of commands compiles and links the `cat` command in the xv6 operating system. Here is what each command does:

- `gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -c -o cat.o cat.c`: This compiles the `cat.c` source code file into an object file called `cat.o`. The options provided to `gcc` specify various compilation options, such as disabling position-independent code (`-fno-pic`), using a static link (`-static`), disabling certain compiler optimizations (`-O2`), and generating debugging information (`-ggdb`).
- `ld -m elf_i386 -N -e main -Ttext 0 -o _cat cat.o ulib.o usys.o printf.o umalloc.o`: This links the `cat.o` object file with several other object files (`ulib.o`, `usys.o`, `printf.o`, `umalloc.o`) into an executable file called

`_cat`. The options provided to `ld` specify the target architecture (`-m elf_i386`), specify the entry point of the program (`-e main`), specify the starting address of the program (`-Ttext 0`), and specify the output file name (`-o _cat`).

- `objdump -S _cat > cat.asm`: This creates a disassembly listing of the `_cat` executable using the `objdump` tool, and writes it to a file called `cat.asm`.
- `objdump -t _cat | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^[$/d' > cat.sym`: This creates a symbol table of the `_cat` executable using the `objdump` tool, pipes it to the `sed` command to remove unnecessary information, and writes it to a file called `cat.sym`.

1.struct proc

This code defines a structure called `proc` which represents the state of a single process in an operating system context. The `proc` structure has several fields:

- `sz`: an unsigned integer representing the size of the process's address space in bytes.
- `pgdir`: a pointer to a page directory which is used by the process to map its virtual address space to physical memory.
- `kstack`: a pointer to the process's kernel stack, which is used when the process is running in kernel mode.
- `state`: an enumerated type representing the current state of the process, such as running, sleeping, waiting for I/O, or exiting.
- `pid`: an integer representing the process ID (PID) of the process.
- `parent`: a pointer to the `proc` structure of the parent process.
- `tf`: a pointer to the trapframe structure containing the saved register state of the process when it was interrupted.
- `context`: a pointer to the saved processor context for the process.
- `chan`: a pointer to a synchronization object that the process is waiting on, such as a semaphore or mutex.
- `killed`: a flag indicating whether the process has been killed.
- `ofile`: an array of pointers to the `file` structures representing the process's open file descriptors.
- `cwd`: a pointer to the `inode` structure representing the process's current working directory.
- `name`: a character array representing the name of the process. This is typically a human-readable string that identifies the process.

2.setting up IDT entries

This code appears to be initializing the Interrupt Descriptor Table (IDT) and the system call interrupt.

The IDT is a data structure used by the CPU to handle various types of interrupts, such as hardware interrupts (e.g., keyboard input) and software interrupts (e.g., system calls). In this code, the IDT is being initialized by setting each of its 256 entries with the SETGATE macro.

The SETGATE macro takes four arguments:

1. A pointer to an IDT entry.
2. A value indicating the type of gate (0 for interrupt gates, 1 for trap gates).
3. The segment selector for the code segment that the gate should use when handling the interrupt.
4. A pointer to the interrupt handler function.

The code is using `SEG_KCODE << 3` to set the code segment selector to the kernel code segment. The `vectors` array contains pointers to the interrupt handler functions.

After initializing the IDT, the code sets up the system call interrupt by calling `SETGATE` with the `T_SYSCALL` constant. The `T_SYSCALL` constant is likely defined elsewhere in the codebase and represents the interrupt number for the system call interrupt.

The second argument of `SETGATE` is set to 1, indicating that interrupts should not be disabled when handling the system call interrupt. The third argument is set to `SEG_KCODE << 3`, indicating that the kernel code segment should be used to handle the system call interrupt.

The final argument of `SETGATE` is a pointer to the system call interrupt handler function, which is likely defined elsewhere in the codebase.

Finally, the code initializes the tickslock semaphore by calling the `initlock` function, passing in a pointer to the semaphore and a string containing its name. The purpose of this semaphore is not clear from this code snippet, but it is likely used to synchronize access to some shared resource involving timekeeping.

3.define setgate

This code defines a macro called `SETGATE` that is used to set the contents of an Interrupt Gate Descriptor in the Interrupt Descriptor Table (IDT) on x86-based systems.

The macro takes five arguments:

1. `gate`: A pointer to the Interrupt Gate Descriptor to be initialized.
2. `istrap`: A boolean value indicating whether the gate should be an Interrupt Gate (0) or a Trap Gate (1).
3. `sel`: A 16-bit value representing the segment selector of the segment that the gate should use.
4. `off`: A 32-bit value representing the offset of the interrupt handler function.

5. d: An 8-bit value representing the privilege level required to execute the interrupt handler.

The macro works by setting the fields of the Interrupt Gate Descriptor based on the arguments passed to it.

Specifically, it sets the 16 least significant bits of the offset field (off_15_0) to the lower 16 bits of the given offset value. It sets the code segment selector field (cs) to the given sel value. It sets the arguments count field (args) to 0, and the reserved field (rsv1) to 0 as well.

The type field (type) is set based on the value of istrap. If istrap is true, then STS_TG32 is used, otherwise STS_IG32 is used. These constants represent the different types of Interrupt Gates and Trap Gates, respectively.

The s field is set to 0 to indicate that this is a system gate. The dpl field is set to the given d value, representing the privilege level required to execute the interrupt handler. The p field is set to 1 to indicate that the gate is present.

Finally, the most significant 16 bits of the offset field (off_31_16) are set to the upper 16 bits of the given offset value.

Overall, this macro provides a convenient way to initialize Interrupt Gate Descriptors in the IDT with the necessary values to handle interrupts and exceptions on x86-based systems.

4.vectors.s

This code defines four global labels: alltraps, vector0, vector1, and the jmp instruction to alltraps function.

The alltraps function is likely a handler function for various types of traps and interrupts on the system, such as page faults or system calls. It is defined elsewhere in the codebase and has been made globally accessible by declaring it with the ".globl" directive.

The vector0 and vector1 labels are used to specify the addresses of the interrupt handler functions for interrupt vectors 0 and 1, respectively. An interrupt vector is an index into the Interrupt Descriptor Table (IDT), which specifies the interrupt handler function to be called when the interrupt occurs.

The code following the vector0 and vector1 labels pushes two values onto the stack and then jumps to the alltraps function.

The first pushl instruction pushes a value of 0 onto the stack, and the second pushl instruction pushes a value of 0 or 1 onto the stack depending on which vector is being initialized. These values are likely used as arguments to the alltraps function to help identify which interrupt occurred.

The jmp instruction then jumps to the address of the alltraps function, which will handle the interrupt based on the values pushed onto the stack.

Overall, this code sets up interrupt handlers for two interrupt vectors (0 and 1) by specifying their addresses and pushing values onto the stack before jumping to the interrupt handler function.

5.trapasm.S

This code defines a global label called "alltraps" and begins the implementation of a trap handler function.

The code first saves the values of the segment registers %ds, %es, %fs, and %gs onto the stack using the "pushl" instruction. This is done to preserve these registers as they may be changed by the exception or interrupt that caused the trap.

The "pushal" instruction is then used to push the values of all general-purpose registers (%eax, %ecx, %edx, %ebx, %esp, %ebp, %esi, and %edi) onto the stack in their current state. This creates a trap frame, which is a data structure that contains the state of the machine at the time of the trap. The trap frame can be used by the trap handler function

to examine the state of the machine and to restore the state of the machine after the trap has been handled.

The overall effect of this code is to save the current state of the machine onto the stack, creating a trap frame that can be used by the trap handler function.

OS Questionnaire

Q. List all the blocks/data-structures on the ext2 partition (e.g., superblock, group descriptor, block bitmap, directory data block, inode table, block groups etc) with block size 4KB, that MAY be updated (means changed) when a file of size 8 KB is created.

Sure, here is a detailed explanation of how each of the data structures/blocks listed above may be updated when a file of size 8 KB is created in a 4 KB block size ext2 partition:

- Superblock - When a file is created in the ext2 file system, the superblock may be updated to reflect the new file system size. The superblock is located at a fixed position on the file system and contains important information about the file system, such as the total number of blocks, the number of free blocks, and the block size. When a new file is created, the block count in the superblock needs to be increased to account for the new blocks that are allocated to the file.
- Block Bitmap - The block bitmap keeps track of which data blocks in the file system are currently in use and which are free. When a new file is created, the file system needs to allocate new data blocks to store the file's contents. The block bitmap is updated to indicate that the newly allocated blocks are now in use.
- Inode Bitmap - The inode bitmap keeps track of which inodes in the file system are currently in use and which are free. When a new file is created, the file system needs to allocate a new inode to store information about the file. The inode bitmap is updated to indicate that the newly allocated inode is now in use.
- Inode Table - The inode table contains information about each file and directory in the file system. When a new file is created, the file system needs to allocate a new inode to store information about the file. The inode table is updated to reflect the new inode allocation and to initialize the inode with information about the new file, such as its size and ownership.
- Directory Data Block - A directory data block contains information about the files and directories in a directory. When a new file is created in a directory, an entry for the new file needs to be added to the directory data block. The directory data block is updated to include a new entry for the newly created file.
- Group Descriptor - The group descriptor contains information about each block group in the file system, such as the location of the block bitmap, inode bitmap, and inode table. When a new file is created, the file system needs to allocate new data blocks and an inode for the file, which may require updates to the group descriptor. For example, the group descriptor may need to be updated to indicate that the block bitmap and inode bitmap in a block group have been updated to allocate new blocks and inodes to the new file.

Q. What are the 3 block allocation schemes, explain in detail.

The three block allocation schemes used in file systems are:

1. Contiguous allocation
2. Linked allocation.
3. Indexed allocation.

Here is a detailed explanation of each of these schemes:

- Contiguous allocation: In this scheme, each file is allocated a contiguous set of data blocks on the disk. This means that all the blocks that belong to a file are stored together in a contiguous sequence on the disk. The advantage of this scheme is that it is simple and fast, as the file can be read or written to disk in one go. However, it suffers from external fragmentation, where the disk becomes fragmented and there are no contiguous blocks large enough to hold a new file. Another disadvantage is that the size of the file is limited by the size of the largest contiguous block available on the disk.
- Linked allocation: In this scheme, each data block contains a pointer to the next data block in the file. The last data block of the file contains a special pointer that indicates the end of the file. This scheme eliminates external fragmentation, as each block can be allocated anywhere on the disk, and the file size can be as large as the number of blocks available on the disk. However, it suffers from the problem of disk overhead, as each block needs to store a pointer to the next block, which increases the amount of space required on disk. Moreover, it is relatively slow to access a file, as the system needs to follow a chain of pointers to read or write to the file.
- Indexed allocation: In this scheme, each file has an index block that contains pointers to all the data blocks that belong to the file. This scheme eliminates external fragmentation and reduces disk overhead, as the index block contains only one pointer to each data block, rather than having each data block contain a pointer to the next block. This scheme also allows the file size to be large, as the number of data blocks that can be allocated to a file is limited only by the size of the index block. However, it suffers from the problem of internal fragmentation, where a partially filled data block may be allocated to a file, resulting in wasted space. Also, it is relatively slower than the contiguous allocation scheme, as the system needs to read the index block to locate the data blocks that belong to the file.

Q. List any 3 problems of each of the block allocation schemes (continuous, linked, indexed).

Sure, here are three problems of each of the three block allocation schemes:

Contiguous Allocation:

- External Fragmentation: Over time, contiguous allocation leads to external fragmentation where there may be no contiguous free blocks large enough to hold a new file, even if there are enough free blocks in total.
- Fixed File Size: The size of the file is limited by the size of the largest contiguous block available on the disk.
- Poor Disk Utilization: If the size of a file changes, the entire file needs to be moved to another location on the disk to make room for the expanded file, which may result in unused blocks of space.

Linked Allocation:

- Disk Overhead: Each block needs to store a pointer to the next block, which increases the amount of space required on disk and reduces the number of blocks available for file data.

- Slow Access: It is relatively slow to access a file, as the system needs to follow a chain of pointers to read or write to the file.
- No Random Access: Linked allocation makes it difficult to access data randomly, as the system needs to start at the beginning of the chain and traverse all blocks in the chain to reach the desired block.

Indexed Allocation:

- Internal Fragmentation: A partially filled data block may be allocated to a file, resulting in wasted space.
- Overhead: Index blocks require space on disk to store the index entries, which can take up a significant amount of space if there are many small files.
- Slower Access: Indexed allocation is relatively slower than contiguous allocation, as the system needs to read the index block to locate the data blocks that belong to the file. Moreover, if the index block is not in memory, it may need to be read from disk, which can cause additional delays.

It's worth noting that file systems typically use a combination of these allocation schemes to optimize performance, such as a combination of contiguous and linked allocation.

Q. What is a device driver? Write some 7-8 points that correctly describe need, use, placement, particularities of device drivers.

A device driver is a software program that allows the operating system to communicate with a hardware device. Some key points that describe the need, use, placement, and particularities of device drivers include:

1. **Need:** Device drivers are necessary because hardware devices have their own unique languages, which are often not compatible with the operating system. Device drivers translate these languages into a language that the operating system can understand and use.
2. **Use:** Device drivers are used to control a wide variety of hardware devices such as printers, keyboards, mice, network adapters, and video cards.
3. **Placement:** Device drivers are usually installed on the operating system as a separate module or driver file, which is loaded into the kernel during the boot process.
4. **Particularities:** Device drivers are unique to each hardware device and must be specifically designed to work with that device. They must be optimized for efficiency and speed, as they often operate in real-time and must process large amounts of data quickly.
5. **Device Configuration:** Device drivers are responsible for configuring the hardware device to work with the operating system. This includes setting up interrupts, I/O ports, and memory mapping.
6. **Error Handling:** Device drivers must be designed to handle errors and exceptions, such as device timeouts or data transmission errors.

7. Security: Device drivers must be designed with security in mind, as they have access to sensitive system resources and can be used to gain unauthorized access to a system.
8. Compatibility: Device drivers must be compatible with the operating system and hardware platform. They must be designed to work with different versions of the operating system and hardware configurations.

Device drivers are software programs that facilitate communication between the operating system and hardware devices. In other words, a device driver is a translator that allows the operating system to interact with the physical hardware components, such as printers, scanners, keyboards, mice, network adapters, and other peripherals.

When the computer needs to interact with a hardware device, it sends a request to the device driver, which then communicates with the device to carry out the requested action. For example, if a user wants to print a document, the computer sends a print request to the printer driver, which then translates the request into a language that the printer can understand.

Device drivers are typically developed by the hardware manufacturer or a third-party developer who specializes in writing device drivers. They are written in low-level programming languages, such as C or C++, and they need to be compatible with the specific operating system and version they are intended for.

Device drivers work in kernel mode, which is a privileged mode of operation in the operating system. This means that device drivers have direct access to hardware resources, such as memory and input/output ports, and can interact with them without going through the user mode.

Device drivers can be classified into different categories, such as input drivers, output drivers, storage drivers, and network drivers, based on the type of hardware device they are designed to control.

In summary, device drivers are essential software programs that allow the operating system to communicate with hardware devices. They are responsible for translating the requests from the operating system into actions that can be understood by the hardware device, and they play a critical role in optimizing the performance and reliability of hardware devices on a computer.

The correct answers are: A process becomes zombie when it finishes, and remains zombie until parent calls wait() on it. A process can become zombie if it finishes, but the parent has finished before it. A zombie process occupies space in OS data structures. If the parent of a process finishes before the process itself, then after finishing the process is typically attached to 'init' as parent, init() typically keeps calling wait() for zombie processes to get cleaned up

A zombie process is a process that has completed its execution but still has an entry in the process table. A zombie process occurs when a parent process does not call the "wait" system call to retrieve the exit status of its terminated child process. As a result, the child process is not fully terminated and remains in a "zombie" state.

Yes, a zombie process does occupy space in the operating system's data structures, specifically in the process table. When a process terminates, its entry in the process table is marked as "zombie" until

the parent process retrieves the exit status of the child process using the "wait" system call. While the process is in a zombie state, it continues to occupy an entry in the process table and consumes some system resources, such as memory and process ID. However, the resources consumed by a zombie process are typically minimal and do not have a significant impact on the overall system performance. Once the parent process retrieves the exit status of the child process, the zombie process is fully terminated and its resources are released.

An orphan process, on the other hand, is a process that has lost its parent process. This can happen if the parent process terminates before the child process, or if the parent process crashes unexpectedly. Orphan processes are adopted by the init process, which is the first process that is started during the system boot-up process and is responsible for starting all other processes.

Orphan processes can become zombie processes if the init process does not call the "wait" system call to retrieve the exit status of the terminated child process. However, orphan processes that are adopted by the init process are not typically a problem, as the init process will eventually clean up any zombie processes that are left over.

In summary, a zombie process is a terminated process that has not been fully cleaned up by its parent process, while an orphan process is a process that has lost its parent process. The main difference between the two is that zombie processes are not fully terminated and remain in the process table, while orphan processes are adopted by the init process and are eventually cleaned up.

Q. Which state changes are possible for a process, which changes are not possible?

A process can go through various states during its lifetime, and different operating systems may use different terms for these states. However, the most common process states are:

- New: When a process is first created, it is in the "new" state.
- Ready: In the "ready" state, the process is waiting to be assigned to a processor.
- Running: When the process is assigned to a processor, it is in the "running" state.
- Waiting: In the "waiting" state, the process is waiting for some event, such as an input or output operation or the completion of another process.
- Terminated: When the process completes its execution, it is in the "terminated" state.

It is possible for a process to transition between all these states. For example, a process in the "new" state can transition to the "ready" state when it is ready to be executed. A process in the "ready" state can transition to the "running" state when it is assigned to a processor. A process in the "running" state can transition to the "waiting" state if it needs to wait for some event to occur. Finally, a process in any state can transition to the "terminated" state when it completes its execution.

However, there are some state transitions that are not possible or are not allowed in some operating systems. For example, it is not possible for a process to transition directly from the "terminated" state to any other state. Similarly, in some operating systems, it is not allowed for a process to transition directly from the "running" state to the "new" state without first going through the "terminated" state.

Q. What is mkfs? what does it do? what are different options to mkfs and what do they mean?

mkfs is a command in Unix-like operating systems that is used to create a new file system on a disk partition or storage device. It stands for "make file system".

When you run the mkfs command, it creates a new file system on the specified partition or storage device by writing a new file system structure to the disk. This includes things like the superblock, block group descriptors, block bitmap, inode bitmap, and the inode table.

The mkfs command supports various options that allow you to customize the file system that it creates. Some of the most used options include:

- -t or --type: This option specifies the type of file system to create. For example, you can use mkfs.ext4 to create an ext4 file system.
- -b or --block-size: This option specifies the size of the blocks that the file system will use. The default block size is usually 4KB, but you can specify a different value if you want.
- -i or --inode-size: This option specifies the size of the inodes that the file system will use. Inodes are data structures that are used to represent files and directories on the file system.
- -L or --label: This option allows you to give a label to the file system. The label is used to identify the file system, and it can be up to 16 characters long.
- -O or --features: This option specifies any optional file system features that you want to enable. For example, you can use -O dir_index to enable directory indexing, which can improve performance on large directories.
- -F or --force: This option forces mkfs to create the file system even if there are warnings or errors.

These are just a few of the options that are available with the mkfs command. The full list of options may vary depending on the type of file system that you are creating.

Q. What is the purpose of the PCB? which are the necessary fields in the PCB?

The Process Control Block (PCB) is a data structure used by an operating system to store information about a running process. The PCB serves as a central repository of information about the process, and the operating system can use this information to manage the process and allocate system resources.

The PCB contains a variety of fields that provide information about the process, including:

1. Process ID (PID): A unique identifier assigned to the process by the operating system.
2. Process state: The current state of the process (e.g. running, waiting, etc.).
3. Program counter (PC): The memory address of the next instruction to be executed by the process.

4. CPU registers: The values of the CPU registers that are being used by the process.
5. Memory management information: Information about the process's memory usage, including the base and limit registers.
6. Priority: The priority level of the process, which determines the amount of CPU time that it will receive.
7. Open files: A list of the files that the process has opened, along with their current state and file pointers.
8. Process accounting information: Information about the resources used by the process, such as CPU time, disk I/O, and memory usage.

These are some of the most important fields that are typically found in a PCB. However, the specific fields and their contents may vary depending on the operating system and the specific requirements of the system.

Process accounting information is a set of data that tracks the resource usage of a process. The operating system can use this information to monitor the performance of the system, to bill users for the resources that they consume, or to identify processes that are using excessive resources.

The process accounting information typically includes data such as:

1. CPU time: The amount of time that the process has spent running on the CPU.
2. Memory usage: The amount of memory that the process is using.
3. Disk I/O: The amount of data that the process is reading from or writing to the disk.
4. Network I/O: The amount of data that the process is sending or receiving over the network.
5. Number of page faults: The number of times that the process has requested a page of memory that is not currently in physical memory.
6. Number of system calls: The number of times that the process has made a system call to request a service from the operating system.

This information can be used by system administrators to identify processes that are consuming excessive resources, or to monitor the overall performance of the system. It can also be used for billing purposes, to charge users for the resources that they consume.

Q. Write a C program that uses globals, local variables, static local variables, static global variables, and malloced memory. Compile it. Dump the object code file using objdump -S. Can you see in the output, the separation into stack, heap, text, etc?

Here is an example C program that uses globals, local variables, static local variables, static global variables, and malloced memory:

c

 Copy code

```
#include <stdio.h>
#include <stdlib.h>

// Global variable
int global_var = 5;

// Static global variable
static int static_global_var = 10;

int main() {
    // Local variable
    int local_var = 15;

    // Static local variable
    static int static_local_var = 20;

    // Malloced memory
    int *ptr = malloc(sizeof(int));
    *ptr = 25;

    printf("Global variable: %d\n", global_var);
    printf("Static global variable: %d\n", static_global_var);
    printf("Local variable: %d\n", local_var);
    printf("Static local variable: %d\n", static_local_var);
    printf("Malloced memory: %d\n", *ptr);

    free(ptr);
    return 0;
}
```

When compiled with `gcc -g -o program program.c`, the object code file can be dumped using `objdump -S program`. The output will show the separation into different sections, such as `.text` (code), `.data` (global and static variables), and `.bss` (uninitialized global and static variables).

Here is an example of the output:

The screenshot shows a terminal window with the title 'yaml'. The content of the terminal is as follows:

```
program: file format elf64-x86-64

Disassembly of section .text:

000000000400526 <main>:
400526: 55                      push  %rbp
400527: 48 89 e5                mov    %rsp,%rbp
40052a: 48 83 ec 10             sub    $0x10,%rsp
40052e: c7 45 fc 0f 00 00 00    movl   $0xf,-0x4(%rbp)
400535: 8b 05 b9 0c 20 00       mov    0x200cb9(%rip),%eax      # 6011f4
40053b: 89 45 f8                mov    %eax,-0x8(%rbp)
40053e: b8 05 00 00 00           mov    $0x5,%eax
400543: 89 45 f4                mov    %eax,-0xc(%rbp)
400546: c7 45 f0 14 00 00 00    movl   $0x14,-0x10(%rbp)
40054d: 8b 05 a6 0c 20 00       mov    0x200ca6(%rip),%eax      # 6011f8
400553: 89 45 ec                mov    %eax,-0x14(%rbp)
400556: 48 8d 45 f0             lea    -0x10(%rbp),%rax
40055a: 48 89 c7                mov    %rax,%rdi
40055d: e8 1e ff ff ff           callq  400580 <malloc@plt>
400562: 48 89 45 e8             mov    %rax,-0x18(%rbp)
400566: 48 8b 45 e8             mov    -0x18(%rbp),%
```

When we run the command `objdump -S <executable_file>`, we get the assembly code generated by the compiler. The output of `objdump` contains several sections, including the text, data, bss, rodata, and comment sections.

- The text section contains the machine code for the program's executable instructions. This section is typically marked as read-only.
- The data section contains initialized global and static variables. These variables are stored in memory and are accessible throughout the entire program.
- The bss section contains uninitialized global and static variables. This section is typically initialized to zero by the operating system when the program is loaded into memory.
- The rodata section contains read-only data, such as string literals.
- The comment section contains any comments that were present in the original source code.

The stack and heap are runtime memory areas used by the program.

- The stack is used for storing local variables and function call frames. It is a last-in-first-out (LIFO) data structure that grows downwards in memory.
- The heap is used for dynamic memory allocation, typically using functions such as `malloc`, `calloc`, and `realloc`. The heap grows upwards in memory.

The stack and heap are not explicitly visible in the `objdump` output, as they are created and destroyed dynamically at runtime. However, we can get an idea of how they are used by looking at the assembly code and observing the instructions that modify the stack and heap pointers.

In summary, the objdump output provides a detailed view of the compiled program's executable instructions, initialized and uninitialized global/static variables, and read-only data. The stack and heap are runtime memory areas that are not explicitly visible in the objdump output, but their usage can be inferred by analysing the assembly code.

Q. Which parts of a C program ((typedef, #define, #include, functions, local vars, statics, globals, #ifdef, ...) etc occupy RAM when it's a process, and where (code, data, stack, ..)?

When a C program is compiled and run as a process, it occupies different parts of the RAM memory. Here's where each part typically resides:

- **typedef:** Typedefs don't take up memory during runtime. They are simply aliases for existing types and are resolved during compilation.
- **#define:** Like typedefs, #defines don't take up memory during runtime. They are simply macros that are resolved during compilation.
- **#include:** The code included by a #include directive is compiled into the program's text section.
- **Functions:** Function code is compiled into the program's text section.
- **Local variables:** Local variables are stored on the stack. Each function call creates a new stack frame, which includes the function's local variables.
- **Static variables:** Static variables that are defined inside a function are stored on the data section of the program, just like global variables. Static variables that are defined outside of functions are stored in the BSS section of the program.
- **Global variables:** Global variables are stored in the data section of the program. Initialized global variables are stored in the data section, while uninitialized global variables are stored in the BSS section.
- **#ifdef:** Preprocessor directives like #ifdef don't take up memory during runtime. They are simply evaluated during compilation.

In summary, the text section contains the compiled code of the program, the data section contains initialized global and static variables, the BSS section contains uninitialized global and static variables, and the stack contains local variables and function call frames.

When a C program is compiled and run as a process, it is divided into several parts that occupy different areas of memory.

Here are the essential parts of a C program and where they typically reside in memory:

- **Preprocessor directives (#define, #include, #ifdef, etc.):** These directives are processed by the preprocessor before compilation and don't take up memory during runtime.
- **Type definitions (typedef):** These are aliases for existing types and don't take up memory during runtime. They are resolved during compilation.

- Global variables: Global variables are stored in the data section of the program. Initialized global variables are stored in the data section, while uninitialized global variables are stored in the BSS (Block Started by Symbol) section.
- Static variables: Static variables that are defined inside a function are stored on the data section of the program, just like global variables. Static variables that are defined outside of functions are stored in the BSS section of the program.
- Function code: Function code is compiled into the program's text section. This section contains the executable code of the program.
- Local variables: Local variables are stored on the stack. Each function call creates a new stack frame, which includes the function's local variables.
- Heap memory: Dynamically allocated memory using functions like malloc() and calloc() is stored on the heap. This memory is managed by the program at runtime.
- Program arguments and environment variables: These are stored on the stack, along with local variables.

When the C program is compiled, the compiler generates an object file, which contains the code and data of the program. The linker then combines the object file with any libraries and generates an executable file.

During runtime, the operating system loads the program into memory and sets up the stack, heap, and other necessary data structures. The program then begins executing from the main() function, and any global variables are initialized before the main() function is called.

In summary, a C program is divided into several parts, each of which occupies a different area of memory. Global variables and static variables are stored in the data and BSS sections of the program, function code is stored in the text section, local variables are stored on the stack, and dynamically allocated memory is stored on the heap.

Q. Describe the role of CPU (MMU), kernel, compiler in the overall task of "memory management" for a process/program.

Memory management is the process of allocating and deallocating memory for a process or program. This process is essential to ensure that the program runs smoothly and that the system resources are used efficiently. The main components involved in memory management are the CPU's MMU, the kernel, and the compiler.

CPU (MMU): The Memory Management Unit (MMU) is a component of the CPU responsible for virtual memory management. It maps virtual addresses used by a process into physical memory addresses. When a process accesses memory, the MMU checks the virtual address to see if it is valid and maps it to the appropriate physical address. This allows the operating system to manage the physical memory more efficiently by allowing multiple processes to access the same physical memory without interfering with each other.

Kernel: The kernel is the core component of the operating system and is responsible for managing system resources, including memory. It provides the processes with virtual address spaces and controls the allocation and deallocation of physical memory. The kernel also provides mechanisms for inter-process communication and synchronization, which may involve sharing of memory between processes.

The kernel manages memory through a set of data structures, such as the page table and the buddy allocator. The page table maps the virtual memory of a process to the physical memory of the system. It keeps track of the page frame number, the physical address of the page in memory, and other information related to the page. The buddy allocator is used to manage the physical memory, which is divided into fixed-sized chunks of pages. The allocator maintains a free list of memory chunks and allocates chunks to processes on demand.

Compiler: The compiler is responsible for generating machine code from the program's source code. It analyses the program's memory usage and generates instructions that use the memory efficiently. For example, the compiler may optimize memory usage by reusing memory for variables or by storing variables in CPU registers rather than in memory.

The compiler also performs several optimization techniques, such as dead code elimination and loop unrolling, to reduce the program's memory footprint. It may also use data structures such as arrays and linked lists to manage the program's data efficiently.

In summary, the CPU's MMU, the kernel, and the compiler work together to manage memory for a process or program. The MMU maps virtual addresses to physical addresses, the kernel manages the allocation and deallocation of physical memory and provides virtual address spaces to processes, and the compiler generates efficient code that uses memory efficiently. The efficient use of memory ensures that the program runs smoothly and that the system resources are used effectively.

Q. What is the difference between a named pipe and un-named pipe? Explain in detail.

A pipe is a method of interprocess communication that allows one process to send data to another process. A named pipe and an unnamed pipe are two types of pipes that differ in their features and usage. Here is a detailed explanation of their differences:

1. **Naming:** An unnamed pipe, also known as an anonymous pipe, has no external name, whereas a named pipe, also known as a FIFO (First In First Out), has a name that is visible in the file system.
2. **Persistence:** A named pipe persists even after the process that created it has terminated, and can be used by other processes. In contrast, an unnamed pipe exists only as long as the process that created it is running.
3. **Communication:** An unnamed pipe is used for communication between a parent process and its child process or between two processes that share a common ancestor. On the other hand, a named pipe can be used for communication between any two processes that have access to the same file system.

4. Access: A named pipe can be accessed by multiple processes simultaneously, allowing for one-to-many communication. In contrast, an unnamed pipe can only be accessed by the two processes that share it, allowing for one-to-one communication.
5. Synchronization: Named pipes provide a method for synchronizing the flow of data between processes, as each write to a named pipe is appended to the end of the pipe and each read retrieves the oldest data in the pipe. In contrast, unnamed pipes do not provide any built-in synchronization mechanism.

In summary, a named pipe is a named, persistent, two-way, interprocess communication channel that can be accessed by multiple processes, while an unnamed pipe is a temporary, one-way, communication channel that can only be accessed by the two processes that share it.

An unnamed pipe, also known as an anonymous pipe, is a temporary, one-way communication channel that is used for interprocess communication between a parent process and its child process or between two processes that share a common ancestor.

An unnamed pipe is created using the `pipe()` system call, which creates a pair of file descriptors. One file descriptor is used for reading from the pipe, and the other is used for writing to the pipe. The file descriptors are inherited by the child process when it is created by the parent process.

An unnamed pipe has a fixed size buffer that is used to store data that is written to the pipe. When the buffer is full, further writes to the pipe will block until space is available in the buffer. The buffer is emptied when data is read from the pipe, creating more space in the buffer for new data to be written.

An unnamed pipe is useful for implementing simple communication between a parent and child process. For example, a parent process may create a child process to perform some work and use an unnamed pipe to receive the results from the child process. However, unnamed pipes have several limitations, such as being limited to one-way communication, having a fixed buffer size, and not providing any synchronization mechanism. These limitations can be overcome by using other interprocess communication methods, such as named pipes, sockets, or message queues.

A named pipe, also known as a FIFO (First-In-First-Out), is a named, one-way communication channel that can be used for interprocess communication between two or more unrelated processes.

A named pipe is created using the `mkfifo()` system call, which creates a special file in the file system that can be opened for reading and writing by multiple processes. Once created, a named pipe can be used like any other file, but the data that is written to the file is not stored in the file system, but is passed directly to the process that is reading from the pipe.

Named pipes are useful for implementing more complex interprocess communication scenarios than can be achieved with unnamed pipes. For example, named pipes can be used to implement a server-client architecture, where multiple clients can connect to a server using a named pipe, and the server can respond to requests from the clients by writing data to the pipe. Named pipes can also be used to implement message passing between unrelated processes, where one process writes messages to a named pipe, and other processes read the messages from the pipe.

In general, named pipes can be used for bidirectional communication between processes, although they can also be used for unidirectional communication if needed. The direction of communication

depends on how the named pipe is implemented and used by the processes that are communicating through it.

Named pipes provide a simple way for two or more processes to communicate with each other by reading and writing data to a common pipe. Each named pipe has two ends, one for reading and one for writing, which can be used by different processes for communication.

For example, process A could open a named pipe for writing and write data to it, while process B could open the same named pipe for reading and read the data written by process A. In this case, communication is unidirectional from A to B. However, if process A also opens the named pipe for reading and process B also opens the named pipe for writing, then bidirectional communication can occur between the two processes.

It's important to note that the implementation of named pipes may differ between operating systems and programming languages, and this can affect whether bidirectional communication is possible or not. Some implementations may restrict the direction of communication, while others may allow bidirectional communication by default.

Named pipes provide several benefits over unnamed pipes, such as being accessible by multiple processes, providing a reliable means of communication, and not requiring a parent-child relationship between the communicating processes.

Q. What are privileged instructions? Why are they required? What is their relationship to 2 modes of CPU execution?

Privileged instructions are instructions in a computer's instruction set architecture (ISA) that can only be executed in privileged mode. Privileged mode is a mode of operation in the CPU that allows certain instructions and actions to be performed that are not allowed in user mode.

Privileged instructions are required to ensure the security and integrity of the operating system and the system as a whole. These instructions are used to perform critical functions such as managing system resources, controlling interrupts and exceptions, accessing privileged hardware devices, and modifying system control registers. Without privileged instructions, an application or user could potentially modify critical system resources and cause system instability or security breaches.

The two modes of CPU execution are user mode and privileged mode. User mode is the normal mode of operation, in which most applications and processes run. In user mode, only a subset of instructions is available, and certain instructions that could cause harm to the system or other processes are prohibited. In privileged mode, all instructions are available, including privileged instructions, and the CPU has access to system resources that are not available in user mode.

The relationship between privileged instructions and the two modes of CPU execution is that privileged instructions can only be executed in privileged mode, which is reserved for the operating system kernel and trusted system processes. By limiting access to privileged instructions to only trusted processes running in privileged mode, the operating system can ensure the security and stability of the system as a whole.

To resolve the path name /a/b/c on an ext2 filesystem, the following steps are involved:

1. The root directory is accessed by the kernel, as all path names start at the root directory.
2. The kernel searches the root directory for the directory entry for the directory "a".
3. The kernel reads the inode for directory "a", which contains a list of directory entries.
4. The kernel searches this list for the directory entry for the directory "b".
5. The kernel reads the inode for directory "b", which contains a list of directory entries.
6. The kernel searches this list for the directory entry for the directory "c".
7. The kernel reads the inode for directory "c", which contains the file data or another list of directory entries.

If any of the directories or files in the path name are not found, the kernel will return an error. The kernel uses the file system's inode and directory data structures to navigate the file system and locate the files and directories specified in the path name. This process is repeated for every path name that the kernel encounters when accessing files on the file system.

Q. Explain what happens in pre-processing, compilation and linking and loading.

Pre-processing, compilation, linking, and loading are the four stages involved in the process of compiling and running a program. Each stage plays a crucial role in the final outcome of the program. Let's discuss each stage in detail:

1. Pre-processing: The first stage in compiling a program is pre-processing. In this stage, the pre-processor reads the source code and performs certain operations on it. These operations include:
 - Removing comments from the code
 - Expanding macros
 - Including header files
 - Defining constants

The pre-processor generates an intermediate code after performing these operations, which is used by the compiler in the next stage.

2. Compilation: The second stage is compilation. In this stage, the compiler reads the intermediate code generated by the pre-processor and translates it into assembly language or machine code. The compiler performs various optimizations to make the code more efficient. The output of the compilation stage is an object file, which contains machine code in binary format.

3. Linking: The third stage is linking. In this stage, the linker combines the object files generated by the compiler and resolves any external references between them. External references are the references to functions or variables defined in other files. The linker generates an executable file after resolving all the external references.
4. Loading: The final stage is loading. In this stage, the operating system loads the executable file into memory and starts executing it. The loader allocates memory for the program and maps the code and data sections of the program into memory. It also sets up the environment for the program to run, such as the stack and heap.

Q. Why is the kernel called an event-drive program? Explain in detail with examples.

In summary, pre-processing, compilation, linking, and loading are the four stages involved in compiling and running a program. Each stage has its own set of tasks and produces a specific output, which is used as an input for the next stage. The final output is an executable file, which can be run by the operating system.

The kernel is called an event-driven program because it responds to various events that occur in the system. Events may include interrupts generated by hardware devices such as disk drives, network adapters, or input/output devices. The kernel must handle these events and take appropriate actions, such as scheduling processes to run, allocating memory to processes, or responding to system calls from user-space applications.

The kernel is constantly monitoring the system for events, and when an event occurs, the appropriate handler function is called to handle the event. The kernel must be able to handle multiple events simultaneously and prioritize them according to their importance and urgency.

The event-driven nature of the kernel is what allows it to be responsive to the needs of the system and provide a stable and reliable operating environment for user-space applications.

The kernel is called an event-driven program because it responds to various events or interrupts that occur in the system. These events can be generated by hardware devices or by software running on the system. The kernel is designed to handle these events and take appropriate actions to ensure the system functions correctly.

Here are some examples of events that the kernel can handle:

1. Interrupts: When a hardware device generates an interrupt, the kernel will pause the current task and handle the interrupt. For example, if the user presses a key on the keyboard, an interrupt is generated, and the kernel will handle it by updating the appropriate data structures and waking up any processes waiting for input.

2. Signals: When a process sends a signal to another process, the kernel will handle the signal and take appropriate action. For example, if a process receives a SIGINT signal (generated by pressing Ctrl+C), the kernel will terminate the process.
3. System calls: When a process makes a system call, the kernel will handle the call and execute the appropriate code. For example, if a process calls the open() system call to open a file, the kernel will handle the call by checking permissions and allocating file descriptors.
4. Memory management: When a process requests memory, the kernel will handle the request and allocate the appropriate amount of memory. For example, if a process requests more memory using malloc(), the kernel will handle the request by allocating the memory and updating the process's memory map.

Overall, the kernel is responsible for handling events and ensuring that the system functions correctly. This requires a lot of coordination between different parts of the kernel and the various hardware and software components in the system.

Q. What are the limitations of segmentation memory management scheme?

Segmentation memory management scheme has the following limitations:

1. External Fragmentation: As memory is allocated in variable-sized segments, the unused memory between two allocated segments may be too small to hold another segment. This leads to external fragmentation and a loss of memory.
2. Internal Fragmentation: In segmentation, memory is allocated in variable-sized segments. Therefore, there is a possibility of having unused memory within a segment, which is called internal fragmentation.
3. Difficulty in Implementation: Segmentation memory management is more complex than other memory management schemes. It requires a sophisticated hardware support and a more complex software to manage it.
4. Limited Sharing: In a segmentation memory management scheme, it is difficult to share segments between processes, because each process has its own segment table. This makes it difficult for processes to share data and code.
5. Security Issues: Segmentation memory management requires more security checks than other memory management schemes because it needs to ensure that one process does not access the memory of another process. This can lead to increased overhead and slower performance.

Overall, while segmentation can be useful for managing memory in certain situations, it has several limitations that must be taken into consideration.

Q. How is the problem of external fragmentation solved?

The problem of external fragmentation can be solved in various ways:

1. Compaction: Compaction involves moving all the allocated memory blocks to one end of the memory and freeing up the unallocated space at the other end. This requires copying the contents of the allocated blocks to a new location, which can be time-consuming and can cause delays in the program execution.
2. Paging: Paging involves dividing the memory into fixed-size pages and allocating the memory on a page-by-page basis. This reduces external fragmentation by eliminating the need for contiguous memory allocation. However, it can lead to internal fragmentation as not all the space in a page may be used.
3. Buddy allocation: Buddy allocation involves dividing the memory into blocks of fixed sizes and allocating them in powers of two. When a block is freed, it is combined with its buddy (a block of the same size) to create a larger block, which can then be allocated to another process. This reduces external fragmentation by ensuring that only blocks of the same size are combined.
4. Segmentation with paging: Segmentation with paging involves dividing the memory into variable-sized segments and then dividing each segment into fixed-sized pages. This combines the benefits of both segmentation and paging and reduces external fragmentation.

Overall, the solution to external fragmentation depends on the specific requirements of the system and the trade-offs between memory utilization and program performance.

Q. Does paging suffer from fragmentation of any kind?

Paging does not suffer from external fragmentation because the pages are of fixed size and can be easily allocated or deallocated without leaving any holes in the memory. However, it can suffer from internal fragmentation, which occurs when a page is allocated to a process, but not all of the space in the page is used. In such cases, some space in the page is wasted, leading to internal fragmentation. This can be minimized by choosing the page size carefully to balance the trade-off between internal fragmentation and page table overhead.

Internal fragmentation in paging occurs when the memory allocated to a process is slightly larger than the actual size required by the process. As a result, some portion of the allocated page remains unused. This unused portion of the page is referred to as internal fragmentation.

For example, if a process requires only 3KB of memory but is allocated an entire page of 4KB, the remaining 1KB of memory will be unused and wasted, resulting in internal fragmentation.

Internal fragmentation can occur in both dynamic and static memory allocation schemes. In dynamic allocation, it may occur due to the allocation of memory blocks that are slightly larger than required, while in static allocation, it may occur due to the allocation of fixed-size memory blocks that are larger than required by some processes.

Segmentation with paging is a memory management technique used by operating systems to handle the virtual memory of a computer system. It is a hybrid method that combines the benefits of both segmentation and paging, while overcoming some of their individual limitations.

In this technique, the logical address space of a process is divided into variable-length segments, each representing a different type of memory region such as code, data, stack, heap, etc. Each segment has its own base address and length and is further divided into fixed-size pages of a uniform size.

The translation of a logical address into a physical address is performed in two stages. First, the segment number is used to index into a segment table, which contains the base address and length of each segment. Then, the page number is used to index into a page table, which maps the page to a physical frame in memory. The physical address is obtained by combining the base address of the segment and the physical frame number.

The segmentation with paging technique provides several benefits:

1. It allows each process to have its own logical address space, which is protected from other processes.
2. It provides flexibility in allocating and managing memory, as segments can be dynamically resized and relocated.
3. It reduces external fragmentation by allowing the operating system to allocate pages from any free frame in memory.
4. It allows for sharing of memory between processes by mapping the same physical page to multiple logical addresses.

However, this technique also has some limitations:

1. It can suffer from internal fragmentation, as a segment may not exactly fit into an integral number of pages.
2. It requires more complex hardware support for the segment and page tables.
3. It can result in slower memory access times due to the additional level of indirection in the address translation process.

Overall, segmentation with paging is a powerful memory management technique that provides a balance between the flexibility and protection of segmentation and the efficiency and simplicity of paging.

Q. What is segmentation fault? why is it caused? who detects it and when? how is the process killed after segmentation fault?

A segmentation fault, also known as a segfault, is an error in the memory management system of a computer. It occurs when a program tries to access a memory location that it is not authorized to access, causing the operating system to terminate the program.

Segmentation faults are typically caused by programming errors such as dereferencing a null pointer or accessing an array out of bounds. They can also be caused by hardware errors or software bugs.

The operating system detects the segmentation fault when the program tries to access an unauthorized memory location. At this point, the operating system terminates the program and returns an error message to the user.

After a segmentation fault, the process is killed by the operating system. The operating system deallocates any resources that were being used by the process, such as memory and open files. The error message is typically written to a log file or displayed to the user, depending on the configuration of the system.

Q. What is the meaning of "core dumped"?

A core dump is a file that gets automatically generated by the Linux kernel after a program crashes. This file contains the memory, register values, and the call stack of an application at the point of crashing. In computing, a core dump, memory dump, crash dump, storage dump, system dump, or ABEND dump consists of the recorded state of the working memory of a computer program at a specific time, generally when the program has crashed or otherwise terminated abnormally.

"Core dumped" is a message displayed in the terminal or console after a program crashes due to a segmentation fault or other fatal error. It means that the program attempted to access memory that it was not allowed to access, and as a result, the operating system terminated the program and saved a snapshot of the program's memory at the time of the crash, known as a "core dump". The core dump can be analysed by a programmer or system administrator to determine the cause of the crash and fix any bugs or issues in the program.

Q. in this program: int main() { int a[16], i; for(i = 0; ; i++) printf("%d\n", a[i]); } why does the program not segfault at a[16] or some more values?

The reason why the program does not segfault at a[16] or some more values is because the array a is a local variable and is allocated on the stack. When the program tries to access a[16] or beyond, it goes beyond the allocated memory of the array and accesses some other part of the stack, which may or may not cause a segmentation fault.

However, since the stack is typically quite large, the program may be able to access many more elements beyond the end of the array before a segmentation fault occurs. Additionally, the behavior of accessing memory beyond the bounds of an array is undefined, which means that the program could produce unpredictable results or crash at any point.

Q. What is voluntary context switch and non-voluntary context switch on Linux? Name 2 processes which have lot of non-voluntary context switches compared to voluntary.

In Linux, voluntary context switch occurs when a process explicitly yields the CPU, for example by calling a blocking system call like sleep(). On the other hand, non-voluntary context switch occurs when the running process is forcibly pre-empted by the kernel scheduler due to some other process becoming runnable or due to the expiration of the process's time slice.

Two processes that have a lot of non-voluntary context switches compared to voluntary are:

1. I/O-bound processes: These processes spend most of their time waiting for I/O operations to complete, and hence are often pre-empted by the kernel when other processes become runnable. Examples include network servers, database servers, and file servers.
2. Real-time processes: These processes have strict timing constraints and need to be executed within a certain time frame. If the kernel scheduler fails to schedule the process in time, it may be pre-empted forcefully to avoid violating the real-time constraints. Examples include multimedia applications, flight control systems, and medical equipment.

1. Context switching

This code snippet appears to be implementing a "context switch" operation, which is a fundamental component of multitasking operating systems.

The purpose of a context switch is to save the current state of a process (its register values and program counter) and then restore the state of a different process so that it can resume execution from where it last left off.

Let's break down the code step by step:

```
movl 4(%esp), %eax
```

```
movl 8(%esp), %edx
```

These instructions load the first two arguments passed to the function into the `eax` and `edx` registers. Assuming this is a function implementing a context switch, we can assume that these arguments represent the addresses of the old and new stacks that will be used to save the register state of the old process and load the state of the new process.

```
# Save old callee-saved registers
```

```
pushl %ebp
```

```
pushl %ebx
```

```
pushl %esi
```

```
pushl %edi
```

These instructions save the values of the callee-saved registers onto the stack. These registers are typically saved by the callee (i.e., the function being called) so that they can be modified without interfering with the caller's code. In this case, the function implementing the context switch is acting as the callee, so it saves these registers to ensure that they are not accidentally overwritten when switching contexts.

```
# Switch stacks
```

```
movl %esp, (%eax)
```

```
movl %edx, %esp
```

These instructions perform the actual context switch. The current value of the stack pointer (`esp`) is saved into the memory location pointed to by the `eax` register (which contains the address of the old stack), and the `esp` register is loaded with the value in `edx` (which contains the address of the new stack). This effectively switches the stack that the processor will use for subsequent instructions.

```
# Load new callee-saved registers
```

```
popl %edi
```

```
popl %esi
```

```
popl %ebx
```

```
popl %ebp
```

```
ret
```

These instructions restore the callee-saved registers from the new stack. Since the function implementing the context switch is acting as the caller, it needs to restore these registers before returning to the newly switched-to process. The `ret` instruction then returns control to the newly switched-to process, which will resume execution from where it last left off.

Overall, this code implements a basic context switch operation by saving the callee-saved registers, switching stacks, restoring the callee-saved registers, and returning control to the new process.

1.Diff between process and threads

In computing, a process is an instance of a computer program that is being executed. A process is a self-contained execution environment that includes the program code, memory, data, and system resources like CPU time, I/O resources, and network access. Each process has its own virtual memory space and runs in a separate address space, which means that a process cannot access the memory of another process directly.

On the other hand, a thread is a lightweight execution unit within a process. A thread shares the same memory and resources as the process that it belongs to. Threads are used to perform multiple tasks concurrently within a single process, which improves the overall performance and responsiveness of the application.

In summary, the key difference between a process and a thread is that a process is an independent execution environment that consists of one or more threads, while a thread is a lightweight execution unit within a process that shares the same memory and resources as the process. Processes are isolated from each other and communicate through inter-process communication mechanisms, while threads share memory and resources and communicate through shared variables and synchronization primitives.

2.no kernel threads one process then many to one mapping how?

In a system without kernel threads, there is only one execution context, which is provided by the process. This means that the process is responsible for managing all of its threads and allocating CPU time to each thread. In this scenario, a user-level thread library can be used to implement concurrency within the process.

A user-level thread library provides a set of functions that allow the programmer to create, schedule, and manage threads within the process. The library maintains a table of user-level thread structures, which contain information about each thread, such as its state, CPU time used, and priority. The library also provides a scheduler that determines which

thread should be executed next, based on the scheduling algorithm implemented by the programmer.

Since there is no kernel support for threads in this scenario, the threads are implemented entirely in user space. This means that the operating system is not aware of the threads, and cannot allocate CPU time to them directly. Instead, the user-level thread library must use blocking system calls or other mechanisms to ensure that each thread gets a fair share of CPU time.

Overall, the many-to-one mapping of user-level threads to the single process is possible because the threads are not directly managed by the kernel. Instead, the user-level thread library provides an abstraction layer that allows multiple threads to be managed within a single process. However, this approach has some limitations, such as the inability to take advantage of multiple CPUs or to run threads in kernel mode.

3. 2 level model in threading

The two-level model in threading refers to a model that includes both user-level threads and kernel-level threads. In this model, user-level threads are created and managed by a user-level thread library, while kernel-level threads are created and managed by the operating system's kernel.

The two-level model provides a number of benefits compared to a single-level model, where all threads are managed at the same level. For example, it allows user-level threads to be implemented efficiently, since they can be scheduled and switched without requiring kernel intervention. This reduces the overhead associated with thread management and improves the performance of the application.

At the same time, the two-level model also provides the benefits of kernel-level threads, such as the ability to take advantage of multiple processors and to execute in kernel mode. Kernel-level threads are managed by the operating system's scheduler, which can assign threads to different processors and optimize their execution to improve performance.

Overall, the two-level model in threading provides a balance between the efficiency of user-level threads and the flexibility and performance of kernel-level threads. It allows applications to take advantage of the benefits of both types of threads, and provides a flexible framework for implementing concurrent and parallel programming.

4. difference between many to many and 2 level model

The many-to-many model and the two-level model are two different approaches to threading, each with its own advantages and disadvantages.

The many-to-many model allows multiple user-level threads to be mapped to multiple kernel-level threads, providing a greater degree of concurrency and parallelism than the two-level model. In this model, the thread library schedules user-level threads on a pool of kernel-level threads, which can be assigned to different processors by the operating system's scheduler. This allows multiple threads to run in parallel on multiple processors, providing better performance and scalability than the two-level model.

However, the many-to-many model can also be more complex to implement than the two-level model, since it requires coordination between the user-level thread library and the operating system's kernel. The thread library must manage a pool of kernel-level threads, and ensure that each user-level thread has access to a sufficient number of kernel-level threads to achieve good performance. Additionally, since the user-level threads are not directly associated with kernel-level threads, there can be additional overhead involved in context switching and synchronization.

In contrast, the two-level model provides a simpler and more predictable threading model. User-level threads are created and managed by a user-level thread library, and are mapped directly to kernel-level threads by the operating system's scheduler. This provides good performance and scalability, especially when running on a system with a small number of processors.

Overall, the choice between the many-to-many model and the two-level model for threading depends on the specific requirements of the application. If the application requires a high degree of concurrency and parallelism, the many-to-many model may be more appropriate. If the application is simpler and requires good performance on a small number of processors, the two-level model may be a better choice.

5. fork duplicates threads?

The `fork()` system call duplicates the entire process, including all threads that are currently running within that process. This means that after the `fork()` call, the child process has an exact copy of the parent process's entire memory space, including all threads that were running at the time of the `fork()` call.

However, it's worth noting that the child process does not inherit the thread IDs of the parent process. Instead, each thread in the child process is given a new thread ID. This means that the child process has a new set of threads that are identical to the threads in the parent process, but with different IDs.

It's also important to keep in mind that after the `fork()` call, each process (the parent and the child) has its own copy of the memory space. This means that any modifications made to the memory space in one process do not affect the memory space in the other process. Similarly, any changes made to a thread in one process do not affect the corresponding thread in the other process.

Overall, the `fork()` system call duplicates the entire process, including all threads, but creates a new set of thread IDs for the child process.

6. pthreads functions

`pthread_key_create()`, `pthread_key_delete()`, `pthread_setspecific()`, and `pthread_getspecific()` are functions in the POSIX threads (pthreads) library that are used to manage thread-specific data. These functions allow a program to associate thread-specific data with a specific key, which can be used to retrieve the data later from any thread.

Here's a brief explanation of each of these functions:

1. `pthread_key_create()` - This function creates a new thread-specific data key, which can be used to store thread-specific data. The

function takes two parameters: a pointer to a `pthread_key_t` variable, which will hold the key value, and an optional destructor function, which will be called automatically when a thread exits and the data associated with the key is no longer needed. The function returns zero on success, or an error code if the key could not be created.

2. `pthread_key_delete()` - This function deletes a previously created thread-specific data key. The function takes one parameter: the key to be deleted. Any data associated with the key in all threads will also be deleted. The function returns zero on success, or an error code if the key could not be deleted.
3. `pthread_setspecific()` - This function associates a value with a specific key for the current thread. The function takes two parameters: the key to associate the value with, and the value to associate with the key. The value can be a pointer to any data type. Each thread can associate a different value with the same key. The function returns zero on success, or an error code if the value could not be set.
4. `pthread_getspecific()` - This function retrieves the value associated with a specific key for the current thread. The function takes one parameter: the key to retrieve the value for. The function returns the value associated with the key, or `NULL` if no value has been associated with the key in the current thread.

Overall, these functions allow a program to create, manage, and access thread-specific data, which can be useful in situations where data needs to be shared between multiple threads without the risk of interference or synchronization issues.

7. upcall handler

This code snippet outlines a simple thread library implementation that uses lightweight processes (LWPs) to schedule threads. Here's a brief explanation of each of the functions:

1. `upcall_handler()` - This is a callback function that will be invoked by the operating system whenever a blocking system call completes. The purpose of this function is to create a new LWP and schedule

any waiting threads onto the new LWP, so that they can continue executing.

2. `th_setup(int n)` - This function initializes the thread library with a maximum number of LWPs (specified by the parameter `n`). The `max_LWP` variable is set to `n`, and the `curr_LWP` variable is initialized to 0. The `register_upcall(upcall_handler)` function is also called to register the `upcall_handler()` function with the operating system, so that it will be called whenever a blocking system call completes.
3. `th_create(...., fn,)` - This function creates a new thread, with the specified function `fn`. If there are available LWPs (i.e. `curr_LWP < max_LWP`), a new LWP is created using `create_LWP`. The thread is then scheduled onto one of the available LWPs, using `schedule fn on one of the LWP`.

Overall, this thread library implementation uses LWPs to schedule threads, and employs an upcall mechanism to handle blocking system calls. When a system call blocks, the `upcall_handler()` function is invoked, and a new LWP is created to handle the waiting threads. This approach allows for efficient and scalable thread scheduling, while minimizing the risk of thread interference and synchronization issues.

1. KINIT1:

The function initlock is called to initialize a lock on a structure called kmem. This lock is used to ensure that multiple threads cannot access the memory allocator (i.e., kmem) at the same time, which could cause problems like data corruption.

The kmem.use_lock variable is set to 0, which means that the lock is not currently in use. This variable is used to track whether the lock is currently being used, so that threads can avoid trying to access kmem at the same time.

The freerange function is called to free a range of memory from vstart to vend. This function is used to initialize the kernel memory allocator, which manages the allocation and deallocation of memory in the kernel.

Overall, this code is used to initialize the kernel memory allocator and ensure that it can be safely accessed by multiple threads.

2. KINIT2:

The freerange function is called to free a range of memory from vstart to vend. This function is used to initialize the kernel memory allocator, which manages the allocation and deallocation of memory in the kernel. This is similar to what happens in kinit1.

The kmem.use_lock variable is set to 1, which means that the lock should now be used to protect access to the memory allocator. This variable is used to track whether the lock is currently being used, so that threads can avoid trying to access kmem at the same time.

By setting kmem.use_lock to 1, the kernel memory allocator is now protected by a lock, which ensures that only one thread can access it at any given time. This is important to avoid race conditions, where multiple threads might try to allocate or deallocate memory at the same time and cause conflicts.

Overall, kinit2 is used to finalize the initialization of the kernel memory allocator and enable thread-safe access to it.

3. FREE RANGE:

The freerange function takes two void pointers vstart and vend, which represent the start and end addresses of a range of memory that needs to be freed.

The PGROUNDUP function is called to round up vstart to the nearest page boundary, which is a multiple of PGSIZE. PGSIZE is a constant that represents the size of a page of memory. The resulting address is stored in a pointer p.

A loop is executed that iterates over the range of memory from p to vend, with a step of PGSIZE each time. The loop body calls the kfree function on each page in the range. kfree is a function that frees a page of memory in the kernel.

Once the loop has finished, all pages in the specified range have been freed.

Overall, this code is used to free a range of memory in the kernel. It works by iterating over the memory range one page at a time and calling kfree on each page to free it. This is typically used during the initialization of the kernel memory allocator to mark a range of memory as free and available for future allocations.

Practice Problems: Process Management in xv6

1. Consider the following lines of code in a program running on xv6.

```
int ret = fork();
if(ret==0) { //do something in child}
else { //do something in parent}
```

- When a new child process is created as part of handling fork, what does the kernel stack of the new child process contain, after fork finishes creating it, but just before the CPU switches away from the parent?
- How is the kernel stack of the newly created child process different from that of the parent?
- The EIP value that is present in the trap frames of the parent and child processes decides where both the processes resume execution in user mode. Do both the EIP pointers in the parent and child contain the same logical address? Do they point to the same physical address in memory (after address translation by page tables)? Explain.
- How would your answer to (c) above change if xv6 implemented copy-on-write during fork?
- When the child process is scheduled for the first time, where does it start execution in kernel mode? List the steps until it finally gets to executing the instruction after fork in the program above in user mode.

Ans:

- It contains a trap frame, followed by the context structure.
- The parent's kernel stack has only a trap frame, since it is still running and has not been context switched out. Further, the value of the EAX register in the two trap frames is different, to return different values in parent and child.
- The EIP value points to the same logical address, but to different physical addresses, as the parent and child have different memory images.
- With copy-on-write, the physical addresses may be the same as well, as long as both parent and child have not modified anything.
- Starts at forkret, followed by trapret. Pops the trapframe and starts executing at the instruction right after fork.

2. Suppose a machine (architecture: x86, single core) has two runnable processes P1 and P2. P1 executes a line of code to read 1KB of data from an open file on disk to a buffer in its memory.

The content requested is not available in the disk buffer cache and must be fetched from disk. Describe what happens from the time the instruction to read data is started in P1, to the time it completes (causing the process to move on to the next instruction in the program), by answering the following questions.

- (a) The code to read data from disk will result in a system call, and will cause the x86 CPU to execute the `int` instruction. Briefly describe what the CPU's `int` instruction does.
- (b) The `int` instruction will then call the kernel's code to handle the system call. Briefly describe the actions executed by the OS interrupt/trap/system call handling code before the read system call causes P1 to block.
- (c) Now, because process P1 has made a blocking system call, the CPU scheduler context switches to some other process, say P2. Now, the data from the disk that unblocks P1 is ready, and the disk controller raises an interrupt while P2 is running. On whose kernel stack does this interrupt processing run?
- (d) Describe the contents of the kernel stacks of P1 and P2 when this interrupt is being processed.
- (e) Describe the actions performed by P2 in kernel mode when servicing this disk interrupt.
- (f) Right after the disk interrupt handler has successfully serviced the interrupt above, and before any further calls to the scheduler to context switch from P2, what is the state of process P1?

Ans:

- (a) The CPU switches to kernel mode, switches to the kernel stack of the process, and pushes some registers like the EIP onto the kernel stack.
 - (b) The kernel pushes a few other registers, updates segment registers, and starts executing the system call code, which eventually causes P1 to block.
 - (c) P2's kernel stack
 - (d) P2's kernel stack has a trapframe (since it switched to kernel mode). P1's kernel stack has both a context structure and a trap frame (since it is currently context switched out).
 - (e) P2 saves user context, switches to kernel mode, services the disk interrupt that unblocks P1, marks P1 as ready, and resumes its execution in userspace.
 - (f) Ready / runnable.
3. Consider a newly created process in xv6. Below are the several points in the code that the new process passes through before it ends up executing the line just after the `fork` statement in its user space. The EIP of each of these code locations exists on the kernel stack when the process is scheduled for the first time. For each of these locations below, precisely explain where on the kernel stack the corresponding EIP is stored (in which data structure etc.), and when (as part of which function or stage of process creation) is the EIP written there. Be as specific as possible in your answers.

- (a) `forkret`
- (b) `trapret`
- (c) just after the `fork()` system call in userspace

Ans:

- (a) EIP of forkret is stored in struct context by allocproc.
 - (b) EIP of trapret is stored on kernel stack by allocproc.
 - (c) EIP of fork system call code is stored in trapframe in parent, and copied to child's kernel stack in the fork function.
4. Consider a process that has performed a blocking disk read, and has been context switched out in xv6. Now, when the disk interrupt occurs with the data requested by the process, the process is unblocked and context switched in immediately, as part of handling the interrupt. [T/F]

Ans: F

5. In xv6, state the system call(s) that result in new `struct proc` objects being allocated.

Ans: fork

6. Give an example of a scenario in which the xv6 dispatcher / `swtch` function does NOT use a `struct context` created by itself previously, but instead uses an artificially hand-crafted `struct context`, for the purpose of restoring context during a context switch.

Ans: When running process for first time (say, after fork).

7. Give an example of a scenario in xv6 where a `struct context` is stored on the kernel stack of a process, but this context is never used or restored at any point in the future.

Ans: When process has finished after exit, its saved context is never restored.

8. Consider a parent process P that has executed a fork system call to spawn a child process C. Suppose that P has just finished executing the system call code, but has not yet returned to user mode. Also assume that the scheduler is still executing P and has not context switched it out. Below are listed several pieces of state pertaining to a process in xv6. For each item below, answer if the state is identical in both processes P and C. Answer Yes (if identical) or No (if different) for each question.

- (a) Contents of the PCB (`struct proc`). That is, are the PCBs of P and C identical? (Yes/No)
- (b) Contents of the memory image (code, data, heap, user stack etc.).
- (c) Contents of the page table stored in the PCB.
- (d) Contents of the kernel stack.
- (e) EIP value in the trap frame.
- (f) EAX register value in the trap frame.
- (g) The physical memory address corresponding to the EIP in the trap frame.
- (h) The files pointed at by the file descriptor table. That is, are the file structures pointed at by any given file descriptor identical in both P and C?

Ans:

- (a) No
- (b) Yes
- (c) No

- (d) No
 - (e) Yes
 - (f) No
 - (g) No
 - (h) Yes
9. Suppose the kernel has just created the first user space “init” process, but has not yet scheduled it. Answer the following questions.
- (a) What does the EIP in the trap frame on the kernel stack of the process point to?
 - (b) What does the EIP in the context structure on the kernel stack (that is popped when the process is context switched in) point to?

Ans:

- (a) address 0 (first line of code in init user code)
 - (b) forkret / trapret
10. Consider a process P that forks a child process C in xv6. Compare the trap frames on the kernel stacks of P and C just after the fork system call completes execution, and before P returns back to user mode. State one difference between the two trap frames at this instant. Be specific in your answer and state the exact field/register value that is different.

Ans: EAX register has different value.

11. In xv6, the EIP within the `struct context` on the kernel stack of a process usually points to the `switch` statement in the `sched` function, where the process gives up its CPU and switches to the scheduler thread during a context switch. Which processes are an exception to this statement? That is, for which processes does the EIP on the context structure point to some other piece of code?

Ans: Newly created processes / processes running for first time

12. When a trap occurs in xv6, and a process shifts from user mode to kernel mode, which entity switches the CPU stack pointer from pointing to the user stack of the running program to its kernel stack? Tick one: x86 hardware instruction / xv6 assembly code

Ans: x86 hardware

13. Consider a process P in xv6, which makes a system call, goes to kernel mode, runs the system call code, and comes back into user mode again. The value of the EAX register is preserved across this transition. That is, the value of the EAX register just before the process started the system call will always be equal to its value just after the process has returned back to user mode. [T/F]

Ans: False, EAX is used to store system call number and return value, so it changes.

14. When a trap causes a process to shift from user mode to kernel mode in xv6, which CPU data registers are stored in the trapframe (on the kernel stack) of the process? Tick one: all registers / only callee-save registers

Ans: All registers

15. When a process in xv6 wishes to pass one or more arguments to the system call, where are these arguments initially stored, before the process initiates a jump into kernel mode? Tick one: user stack / kernel stack

Ans: User stack, as user program cannot access kernel stack

16. Consider the context switch of a CPU from the context of process P1 to that of process P2 in xv6. Consider the following two events in the chronological order of the events during the context switch: (E1) the ESP (stack pointer) shifts from pointing to the kernel stack of P1 to the kernel stack of P2; (E2) the EIP (program counter) shifts from pointing to an address in the memory allocated to P1 to an address in the memory allocated to P2. Which of the following statements is/are true regarding the relative ordering of events E1 and E2?

- (a) E1 occurs before E2.
- (b) E2 occurs before E1.
- (c) E1 and E2 occur simultaneously via an atomic hardware instruction.
- (d) The relative ordering of E1 and E2 can vary from one context switch to the other.

Ans: (a)

17. Consider the following actions that happen during a context switch from thread/process P1 to thread/process P2 in xv6. (One of P1 or P2 could be the scheduler thread as well.) Arrange the actions below in chronological order, from earliest to latest.

- (A) Switch ESP from kernel stack of P1 to that of P2
- (B) Pop the callee-save registers from the kernel stack of P2
- (C) Push the callee-save registers onto the kernel stack of P1
- (D) Push the EIP where execution of P1 stops onto the kernel stack of P1.

Ans: DCAB

18. Consider a process P in xv6 that invokes the wait system call. Which of the following statements is/are true?

- (a) If P does not have any zombie children, then the wait system call returns immediately.
- (b) The wait system call always blocks process P and leads to a context switch.
- (c) If P has exactly one child process, and that child has not yet terminated, then the wait system call will cause process P to block.
- (d) If P has two or more zombie children, then the wait system call reaps all the zombie children of P and returns immediately.

Ans: (c)

19. Consider a process P in xv6 that executes the exec system call successfully. Which of the following statements is/are true?

- (a) The exec system call changes the PID of process P.
- (b) The exec system call allocates a new page table for process P.

- (c) The exec system call allocates a new kernel stack for process P.
- (d) The exec system call changes one or more fields in the trap frame on the kernel stack of process P.

Ans: (b), (d)

20. Consider a process P in xv6 that executes the exec system call successfully. Which of the following statements is/are true?

- (a) The arguments to the exec system call are first placed on the user stack by the user code.
- (b) The arguments to the exec system call are first placed on the kernel stack by the user code.
- (c) The arguments (argc, argv) to the new executable are placed on the kernel stack by the exec system call code.
- (d) The arguments (argc, argv) to the new executable are placed on the user stack by the exec system call code.

Ans: (a), (d)

21. Consider a newly created child process C in xv6 that is scheduled for the first time. At the point when the scheduler is just about to context switch into C, which of the following statements is/are true about the kernel stack of process C?

- (a) The top of the kernel stack contains the context structure, whose EIP points to the instruction right after the fork system call in user code.
- (b) The bottom of the kernel stack has the trapframe, whose EIP points to the forkret function in OS code.
- (c) The top of the kernel stack contains the context structure, whose EIP points to the forkret function in OS code.
- (d) The bottom of the kernel stack contains the trap frame, whose EIP points to the trapret function in OS code.

Ans: (c)

22. Consider a trapframe stored on the kernel stack of a process P in xv6 that jumped from user mode to kernel mode due to a trap. Which of the following statements is/are true?

- (a) All fields of the trapframe are pushed onto the kernel stack by the OS code.
- (b) All fields of the trapframe are pushed onto the kernel stack by the x86 hardware.
- (c) The ESP value stored in the trapframe points to the top of the kernel stack of the process.
- (d) The ESP value stored in the trapframe points to the top of the user stack of the process.

Ans: (d)

23. Consider a process P that has made a blocking disk read in xv6. The OS has issued a disk read command to the disk hardware, and has context switched away from P. Which of the following statements is/are true?

- (a) The top of the kernel stack of P contains the return address, which is the value of EIP pointing to the user code after the read system call.
- (b) The bottom of the kernel stack of P contains the trapframe, whose EIP points to the user code after the read system call.
- (c) The top of the kernel stack of P contains the context structure, whose EIP points to the user code after the read system call.
- (d) The CPU scheduler does not run P again until after the disk interrupt that unblocks P is raised by the device hardware.

Ans: (b), (d)

24. In the implementation of which of the following system calls in xv6 are new ptable entries allocated or old ptable entries released (marked as unused)?

- (a) fork
- (b) exit
- (c) exec
- (d) wait

Ans: (a), (d)

25. A process has invoked exit() in xv6. The CPU has completed executing the OS code corresponding to the exit system call, and is just about to invoke the swtch() function to switch from the terminated process to the scheduler thread. Which of the following statements is/are true?

- (a) The stack pointer ESP is pointing to some location within the kernel stack of the terminated process
- (b) The MMU is using the page table of the terminated process
- (c) The state of the terminated process in the ptable is RUNNING
- (d) The state of the terminated process in the ptable is ZOMBIE

Ans: (a), (b), (d)

DESIGN QUESTIONS

Explain how you would implement a basic file system journaling feature in xv6.

Answer:

1. Journal structure: Define a journal structure in fs.h to log file system operations before they are committed.
2. Modify file system operations: Update file system functions in fs.c to log each operation to the journal before executing it.
3. Commit changes: Implement a function to commit changes logged in the journal to the file system in a transactional manner.
4. Recovery: Write a recovery mechanism in fs.c to replay journal entries in case of system crashes or failures.
5. Test: Write test cases to verify that the journaling feature ensures file system consistency and integrity, even in the event of crashes.

Question 2: Describe the steps to implement a basic process checkpointing and restoration feature in xv6.

Answer:

1. Checkpointing: Implement a function in proc.c to save the state of a process, including its registers, memory, and file descriptors, to disk.
2. Restore: Write a function to restore a process from a checkpointed state, loading its saved state from disk back into memory.
3. User-level interface: Define system calls in syscall.h and implement corresponding functions in sysproc.c to checkpoint and restore processes.
4. Cleanup: Implement mechanisms to release resources associated with checkpointed processes when they are no longer needed.
5. Test: Write test cases to ensure that processes can be successfully checkpointed and restored without loss of state or functionality.

Question 3: Explain how you would implement a basic memory protection feature to prevent processes from accessing unauthorized memory regions in xv6.

Answer:

1. Page table protection: Modify the page table structure in vm.h to include permission bits for each page, indicating whether it is readable, writable, or executable.
2. Memory access checks: Update memory access functions in vm.c to check permissions in the page table before allowing processes to read, write, or execute memory.
3. Fault handling: Modify the page fault handler in trap.c to handle access violations by terminating processes that attempt to access unauthorized memory regions.
4. User-space interface: Define system calls in syscall.h and implement functions in sysproc.c to allocate and protect memory regions with specified permissions.
5. Test: Write test cases to verify that memory protection prevents unauthorized memory access and that processes are terminated appropriately when violations occur.

Describe the steps to implement a basic signal handling mechanism in xv6.

Answer:

1. Define signal numbers: Assign unique numbers to each type of signal in signal.h.
2. Modify process structure: Add a signal mask and signal handlers to the proc structure in proc.h.
3. Signal sending: Implement functions in proc.c to send signals to specific processes or all processes.
4. Signal handling: Modify the trap handling code in trap.c to invoke signal handlers when signals are received.
5. Test: Write test cases to ensure that signals can be sent and received correctly, and that signal handlers are invoked as expected.

Question 2: Explain how you would implement file permissions and access control in xv6.

Answer:

1. Modify file structure: Add fields for permissions (read, write, execute) to the file structure in file.h.
2. Check permissions: Modify file system functions in fs.c to check permissions before allowing file operations like reading, writing, or executing.
3. User and group ownership: Implement functions in fs.c to track the owner and group of each file, and check permissions accordingly.

4. Setuid and setgid: Implement mechanisms to temporarily change the effective user or group ID of a process when executing files with the setuid or setgid bits set.
5. Test: Write test cases to ensure that file permissions are enforced correctly for different users and groups, and that setuid and setgid functionality works as expected.

Question 3: Describe the steps to implement a basic networking stack in xv6.

Answer:

1. Network device support: Add support for network devices by writing device drivers in dev.c.
2. Network protocol implementation: Implement network protocols such as TCP/IP or UDP/IP in net.c to handle packet transmission and reception.
3. Socket API: Define socket system calls in syscall.h and implement socket functions in syssocket.c to create, bind, connect, send, and receive data over network sockets.
4. Packet handling: Write functions in net.c to handle incoming and outgoing network packets, including packet fragmentation and reassembly.
5. Test: Write test cases to ensure that network communication works correctly, including connecting to remote hosts, sending and receiving data, and handling network errors.

Question 1: Explain how you would implement a basic virtual memory system in xv6.

Answer:

1. **Modify page table structure:** Introduce a page table structure in **vm.h** to map virtual addresses to physical addresses.
2. **Page fault handling:** Implement page fault handling in **trap.c** to handle page faults by loading data from disk into memory.
3. **Memory allocation:** Update memory allocation functions like **kalloc()** and **kfree()** in **kalloc.c** to manage both physical and virtual memory.
4. **User-space memory mapping:** Implement functions in **vm.c** to map virtual memory to physical memory for user-space processes.
5. **Test:** Write test cases to ensure that virtual memory management works correctly, including memory mapping, page fault handling, and memory allocation.

Question 2: Describe the steps to implement a basic file system cache in xv6.

Answer:

1. **Introduce cache structure:** Define a cache structure in **fs.h** to store recently accessed file system blocks.
2. **Read/write caching:** Modify file system read and write functions in **fs.c** to check the cache for requested blocks before accessing disk.

3. **Cache eviction policy:** Implement a cache eviction policy to replace old blocks with new ones when the cache is full.
4. **Synchronization:** Ensure that the cache is thread-safe by using locks to prevent race conditions.
5. **Test:** Write test cases to verify that the file system cache improves performance by reducing disk access.

Question 3: Explain how you would implement inter-process communication (IPC) using shared memory in xv6.

Answer:

1. **Allocate shared memory:** Implement functions in **vm.c** to allocate shared memory regions that can be accessed by multiple processes.
2. **Map shared memory:** Modify process creation functions in **exec.c** to map shared memory regions into the address space of new processes.
3. **Synchronization:** Use synchronization primitives like semaphores or locks to coordinate access to shared memory between processes.
4. **Cleanup:** Implement mechanisms to release shared memory when processes exit to prevent memory leaks.
5. **Test:** Write test cases to ensure that shared memory regions can be successfully accessed and modified by multiple processes without data corruption or race conditions.

Explain how you would implement a basic file compression feature in xv6.

Answer:

1. **Compression algorithm:** Choose a compression algorithm such as LZ77 or Huffman coding.
2. **Modify file system:** Implement functions in **fs.c** to compress and decompress files before reading from or writing to disk.
3. **User-level interface:** Define system calls in **syscall.h** and implement functions in **sysfile.c** to enable compression and decompression of files.
4. **Compression metadata:** Store metadata in the file system to indicate whether a file is compressed and which compression algorithm was used.
5. **Test:** Write test cases to ensure that files can be compressed and decompressed correctly without loss of data or functionality.

Question 2: Describe the steps to implement a basic process migration feature in xv6.

Answer:

1. **Checkpoint process state:** Implement functions in **proc.c** to save the state of a process, including its memory contents and execution context, to disk.
2. **Transfer process state:** Write functions to transfer a process's state from one machine to another over the network.
3. **Restore process state:** Implement functions to restore a process's state on a remote machine, loading its saved state from disk back into memory.
4. **Synchronization:** Ensure that process migration is synchronized to prevent race conditions and inconsistencies.
5. **Test:** Write test cases to verify that processes can be successfully migrated between machines without loss of state or functionality.

Question 3: Explain how you would implement a basic file system encryption feature in xv6.

Answer:

1. **Encryption algorithm:** Choose an encryption algorithm such as AES or RSA.
2. **Key management:** Implement functions in **fs.c** to generate and manage encryption keys for each file.
3. **Encrypt file contents:** Modify file system functions to encrypt file contents before writing them to disk.
4. **Decrypt file contents:** Implement functions to decrypt file contents when reading from disk.
5. **Test:** Write test cases to ensure that files can be encrypted and decrypted correctly, and that encrypted files cannot be accessed without the appropriate decryption key.

These questions delve into more advanced topics such as file compression, process migration, and file system encryption, providing a broader understanding of operating system design and implementation in xv6.

Question 1: Explain how you would implement a basic multi-threading support in xv6.

Answer:

1. **Thread structure:** Define a thread control block structure to manage thread-specific information such as register state and stacsk pointer.
2. **Thread creation:** Implement functions in **thread.c** to create and initialize threads within a process.
3. **Thread scheduling:** Modify the scheduler in **proc.c** to include support for scheduling multiple threads within a process.
4. **Synchronization:** Implement synchronization primitives such as locks and condition variables to coordinate access to shared resources between threads.
5. **Test:** Write test cases to ensure that multiple threads can execute concurrently within a process and that synchronization primitives work as expected.

Question 2: Describe the steps to implement a basic file versioning feature in xv6.

Answer:

1. Version control metadata: Modify the file system to store metadata such as timestamps and version numbers for each file.
2. File versioning: Implement functions in fs.c to create and manage multiple versions of a file, allowing users to revert to previous versions if needed.
3. User-level interface: Define system calls in syscall.h and implement functions in sysfile.c to enable versioning of files.
4. Version history: Implement functions to track and display the history of file versions, including the ability to compare different versions.
5. Test: Write test cases to ensure that file versioning works correctly, including creating, reverting, and comparing versions of files.

Question 3: Explain how you would implement a basic file system quota management feature in xv6.

Answer:

1. User quotas: Define quotas for each user or group in the file system metadata.
2. Quota enforcement: Modify file system functions to track and enforce quotas when users or groups attempt to allocate disk space or create files.
3. Quota accounting: Implement functions to update quota usage when users or groups perform file system operations such as creating, modifying, or deleting files.
4. Quota reporting: Write functions to report quota usage and notify users or administrators when quotas are exceeded.
5. Test: Write test cases to verify that quotas are enforced correctly and that users are notified when their quota limits are reached.

The `.bochsrc` file in the xv6 code is a configuration file used by the Bochs IA-32 emulator. Bochs is a software emulation program that emulates an entire PC system, including the CPU, memory, disk, and other components. The `.bochsrc` file specifies the settings for the Bochs emulator, such as the amount of memory, the hard drive image to use, the boot order, and so on.

In the xv6 code, the `.bochsrc` file is used to specify the configuration for booting the xv6 operating system in the Bochs emulator. The file contains several options and settings, such as:

- `megs: 32` - This specifies the amount of memory to allocate for the virtual machine. In this case, 32MB of memory is allocated for the xv6 operating system.
- `romimage: file=/usr/local/share/bochs/BIOS-bochs-latest` - This specifies the BIOS image to use. In this case, it points to the latest BIOS image for the Bochs emulator.
- `vgaromimage: /usr/local/share/bochs/VGABIOS-Igpl-latest` - This specifies the VGA BIOS image to use.
- `ata0: enabled=1, ioaddr1=0x1f0, ioaddr2=0x3f0, irq=14` - This specifies the configuration for the first ATA device, which is used to boot the xv6 operating system.
- `boot: disk` - This specifies that the xv6 operating system should be booted from the disk image.

In summary, the `.bochsrc` file in the xv6 code is used to specify the configuration for booting the xv6 operating system in the Bochs emulator, including the amount of memory, the BIOS and VGA BIOS images, the ATA device configuration, and the boot order.

2.

`xv6. img` is a disk image file that contains the xv6 operating system. A disk image file is a complete copy of a disk that contains an operating system or other data. It is typically used to boot a virtual machine or to restore a system to a previous state.

In the case of xv6, `xv6. img` is a pre-built disk image file that contains a fully functional version of the xv6 operating system. This file is used to boot a QEMU virtual machine that runs the xv6 operating system. The file is typically created using tools like `mkfs` and `dd`, which are used to format and copy the files onto a virtual disk.

When the virtual machine is started using the `qemu` command with the `-drive` option that points to `xv6. img`, QEMU loads the xv6 operating system from the disk image file and boots it up as if it were running on a physical machine. This allows users to test, modify, and experiment with the xv6 operating system in a safe and isolated environment.

3.

The commands you provided are used to create the `xv6. img` disk image file, which is used to boot the xv6 operating system. Here's a breakdown of each command:

1. `dd if=/dev/zero of=xv6. img count=10000`: This command creates a new disk image file called `xv6. img` and fills it with zeros. The `count=10000` option specifies that the file should be 10000 blocks (or 5MB) in size.
2. `dd if=bootblock of=xv6. img conv=notrunc`: This command writes the `bootblock` file to the beginning of the `xv6. img` disk image file. The `if=bootblock` option specifies the input file to use, while the `of=xv6. img` option specifies the output file. The `conv=notrunc` option tells `dd` not to truncate the output file, which means that any existing data in the file will not be overwritten.
3. `dd if=kernel of=xv6. img seek=1 conv=notrunc`: This command writes the `kernel` file to the `xv6. img` disk image file, starting from the second block (since the first block is already occupied by the `bootblock`). The `if=kernel` option specifies the input file to use, while the `of=xv6. img`

option specifies the output file. The `seek=1` option tells `dd` to skip the first block and start writing from the second block. The `conv=notrunc` option tells `dd` not to truncate the output file, as before.

Overall, these commands create a disk image file called `xv6.img` and write the `bootblock` and `kernel` files to it. The resulting file is a complete copy of a disk that contains the xv6 operating system, ready to be booted by a virtual machine or a physical machine.

4. BOOTBLOCK (BOOTASM, BOOTMAIN)

These lines of code are part of a Makefile rule that compiles a boot loader program for a computer system. Here's what each line does:

1. `bootblock: bootasm.S bootmain.c` - This line specifies the target "bootblock" and its dependencies, which are the source code files "bootasm.S" and "bootmain.c". This means that in order to create the "bootblock" binary, these source files must be compiled.
2. `$(CC) $(CFLAGS) -fno-pic -O -nostdinc -I. -c bootmain.c` - This line invokes the C compiler (`$(CC)`) with the specified flags (`$(CFLAGS)`) to compile the "bootmain.c" source file. The flags used are `-fno-pic` (don't use position-independent code), `-O` (optimize the code), `-nostdinc` (don't include standard system headers), `-I.` (add the current directory to the include search path), and `-c` (generate an object file, rather than an executable).
3. `$(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c bootasm.S` - This line is similar to the previous line, but compiles the "bootasm.S" assembly source file.
4. `$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o bootblock.o bootasm.o bootmain.o` - This line invokes the linker (`$(LD)`) with the specified flags (`$(LDFLAGS)`) to link the object files generated in the previous steps into a single binary file called "bootblock.o". The flags used are `-N` (set the text and data sections to be writable and executable), `-e start` (set the entry point to the "start" label), `-Ttext 0x7C00` (set the starting address of the program to 0x7C00), and `-o bootblock.o` (output the linked binary to a file called "bootblock.o").

5. `$(OBJDUMP) -S bootblock.o > bootblock.asm` - This line disassembles the "bootblock.o" binary into assembly code and saves it to a file called "bootblock.asm".
6. `$(OBJCOPY) -S -O binary -j .text bootblock.o bootblock` - This line copies the "text" section of the "bootblock.o" binary into a new binary file called "bootblock".
7. `./sign.pl bootblock` - This line runs a script called "sign.pl" with the "bootblock" binary as an argument. The purpose of this script is not clear from the given code, but it may be used to add a digital signature to the binary or perform some other post-processing step.

5. Kernel

The code you provided is a Makefile rule for building a kernel program. Here's what each line does:

1. `kernel: $(OBJS) entry.o entryother initcode kernel.1d` - This line specifies the target "kernel" and its dependencies, which are the object files listed in the variable `$(OBJS)`, as well as the "entry.o", "entryother", "initcode", and "kernel.1d" files. This means that in order to create the "kernel" binary, these files must be linked together.
2. `$(LD) $(LDFLAGS) -T kernel.1d -o kernel entry.o $(OBJS) -b binary initcode entryother` - This line invokes the linker (`$(LD)`) with the specified flags (`$(LDFLAGS)`) to link the object files generated in previous compilation steps into a single binary file called "kernel". The flags used are `-T kernel.1d` (use the linker script file "kernel.1d"), `-o kernel` (output the linked binary to a file called "kernel"), `-b binary` (treat the "initcode" and "entryother" files as binary files), and `entry.o $(OBJS)` (specify the object files to link).
3. `$(OBJDUMP) -S kernel > kernel.asm` - This line disassembles the "kernel" binary into assembly code and saves it to a file called "kernel.asm".
4. `$(OBJDUMP) -t kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > kernel.sym` - This line generates a symbol table for the "kernel" binary and saves it to a file called "kernel.sym". The `$(OBJDUMP)` command generates a full symbol table, but the `sed` command filters out unnecessary information and formatting. Specifically, it

deletes everything before the "SYMBOL TABLE" header, removes the second column of each line (which contains unnecessary information), and deletes any empty lines. The resulting symbol table lists the names and addresses of all global symbols defined in the "kernel" binary.

6. fs.img, xv6.img

`fs. img` and `xv6. img` are disk images used by the XV6 operating system.

`xv6. img` is the bootable disk image that contains the XV6 kernel and file system. It is created by combining the bootloader (bootblock) and kernel (kernel) into a single image. This disk image is loaded into a virtual machine or written to a physical disk in order to boot the XV6 operating system.

`fs. img` is a separate disk image used to store the file system data for XV6. It is created separately from `xv6. img` and contains the initial file system used by the operating system. The file system contains the root directory and a few other initial directories and files, which can be modified and extended by user programs.

Both `xv6. img` and `fs. img` are binary files that represent the content of a disk, including its partition table, boot sector, and file system data. They can be modified using disk editing tools, such as `dd`, to change the content of the disk, create new partitions, or copy data from one disk to another.

7. compiling userland programs

The code you provided is a makefile rule that describes how to build an executable binary file from a C source file, with the help of some object files (`ulib.o`, `usys.o`, `printf.o`, `umalloc.o`).

Here is a breakdown of what each line does:

- `_%: %.o $(ULIB)`: This line specifies a target that matches any file name ending with an underscore. The dependencies of this target

are the corresponding `.o` file (the C source file compiled into an object file) and the `ULIB` object files.

- `$(LD) $(LDFLAGS) -N -e main -Ttext 0 -o $@ $^`: This line links the object files into a binary executable using the linker (`ld`). The `-N` option specifies that no default values should be used for uninitialized data, `-e main` specifies that the entry point of the program should be the `main` function, `-Ttext 0` specifies the starting address of the program, and `-o $@` specifies the output file name. The `$^` variable expands to a list of all the dependencies (the `.o` files and `ULIB` object files), and `$@` expands to the name of the target.
- `$(OBJDUMP) -S $@ > $*.asm`: This line creates a disassembly listing of the executable using the `objdump` tool, and writes it to a file with the same name as the target, but with a `.asm` extension.
- `$(OBJDUMP) -t $@ | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' > $*.sym`: This line creates a symbol table of the executable using the `objdump` tool, pipes it to the `sed` command to remove unnecessary information, and writes it to a file with the same name as the target, but with a `.sym` extension.

In summary, this makefile rule describes how to build an executable binary file from a C source file and some object files, and also creates a disassembly listing and a symbol table of the resulting executable.

8. Compiling cat command

This set of commands compiles and links the `cat` command in the xv6 operating system. Here is what each command does:

- `gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -c -o cat.o cat.c`: This compiles the `cat.c` source code file into an object file called `cat.o`. The options provided to `gcc` specify various compilation options, such as disabling position-independent code (`-fno-pic`), using a static link (`-static`), disabling certain compiler optimizations (`-O2`), and generating debugging information (`-ggdb`).
- `ld -m elf_i386 -N -e main -Ttext 0 -o _cat cat.o ulib.o usys.o printf.o umalloc.o`: This links the `cat.o` object file with several other object files (`ulib.o`, `usys.o`, `printf.o`, `umalloc.o`) into an executable file called

`_cat`. The options provided to `ld` specify the target architecture (`-m elf_i386`), specify the entry point of the program (`-e main`), specify the starting address of the program (`-Ttext 0`), and specify the output file name (`-o _cat`).

- `objdump -S _cat > cat.asm`: This creates a disassembly listing of the `_cat` executable using the `objdump` tool, and writes it to a file called `cat.asm`.
- `objdump -t _cat | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^[$/d' > cat.sym`: This creates a symbol table of the `_cat` executable using the `objdump` tool, pipes it to the `sed` command to remove unnecessary information, and writes it to a file called `cat.sym`.

1.struct proc

This code defines a structure called `proc` which represents the state of a single process in an operating system context. The `proc` structure has several fields:

- `sz`: an unsigned integer representing the size of the process's address space in bytes.
- `pgdir`: a pointer to a page directory which is used by the process to map its virtual address space to physical memory.
- `kstack`: a pointer to the process's kernel stack, which is used when the process is running in kernel mode.
- `state`: an enumerated type representing the current state of the process, such as running, sleeping, waiting for I/O, or exiting.
- `pid`: an integer representing the process ID (PID) of the process.
- `parent`: a pointer to the `proc` structure of the parent process.
- `tf`: a pointer to the trapframe structure containing the saved register state of the process when it was interrupted.
- `context`: a pointer to the saved processor context for the process.
- `chan`: a pointer to a synchronization object that the process is waiting on, such as a semaphore or mutex.
- `killed`: a flag indicating whether the process has been killed.
- `ofile`: an array of pointers to the `file` structures representing the process's open file descriptors.
- `cwd`: a pointer to the `inode` structure representing the process's current working directory.
- `name`: a character array representing the name of the process. This is typically a human-readable string that identifies the process.

2.setting up IDT entries

This code appears to be initializing the Interrupt Descriptor Table (IDT) and the system call interrupt.

The IDT is a data structure used by the CPU to handle various types of interrupts, such as hardware interrupts (e.g., keyboard input) and software interrupts (e.g., system calls). In this code, the IDT is being initialized by setting each of its 256 entries with the SETGATE macro.

The SETGATE macro takes four arguments:

1. A pointer to an IDT entry.
2. A value indicating the type of gate (0 for interrupt gates, 1 for trap gates).
3. The segment selector for the code segment that the gate should use when handling the interrupt.
4. A pointer to the interrupt handler function.

The code is using `SEG_KCODE<<3` to set the code segment selector to the kernel code segment. The `vectors` array contains pointers to the interrupt handler functions.

After initializing the IDT, the code sets up the system call interrupt by calling `SETGATE` with the `T_SYSCALL` constant. The `T_SYSCALL` constant is likely defined elsewhere in the codebase and represents the interrupt number for the system call interrupt.

The second argument of `SETGATE` is set to 1, indicating that interrupts should not be disabled when handling the system call interrupt. The third argument is set to `SEG_KCODE<<3`, indicating that the kernel code segment should be used to handle the system call interrupt.

The final argument of `SETGATE` is a pointer to the system call interrupt handler function, which is likely defined elsewhere in the codebase.

Finally, the code initializes the tickslock semaphore by calling the `initlock` function, passing in a pointer to the semaphore and a string containing its name. The purpose of this semaphore is not clear from this code snippet, but it is likely used to synchronize access to some shared resource involving timekeeping.

3.define setgate

This code defines a macro called `SETGATE` that is used to set the contents of an Interrupt Gate Descriptor in the Interrupt Descriptor Table (IDT) on x86-based systems.

The macro takes five arguments:

1. `gate`: A pointer to the Interrupt Gate Descriptor to be initialized.
2. `istrap`: A boolean value indicating whether the gate should be an Interrupt Gate (0) or a Trap Gate (1).
3. `sel`: A 16-bit value representing the segment selector of the segment that the gate should use.
4. `off`: A 32-bit value representing the offset of the interrupt handler function.

5. d: An 8-bit value representing the privilege level required to execute the interrupt handler.

The macro works by setting the fields of the Interrupt Gate Descriptor based on the arguments passed to it.

Specifically, it sets the 16 least significant bits of the offset field (off_15_0) to the lower 16 bits of the given offset value. It sets the code segment selector field (cs) to the given sel value. It sets the arguments count field (args) to 0, and the reserved field (rsv1) to 0 as well.

The type field (type) is set based on the value of istrap. If istrap is true, then STS_TG32 is used, otherwise STS_IG32 is used. These constants represent the different types of Interrupt Gates and Trap Gates, respectively.

The s field is set to 0 to indicate that this is a system gate. The dpl field is set to the given d value, representing the privilege level required to execute the interrupt handler. The p field is set to 1 to indicate that the gate is present.

Finally, the most significant 16 bits of the offset field (off_31_16) are set to the upper 16 bits of the given offset value.

Overall, this macro provides a convenient way to initialize Interrupt Gate Descriptors in the IDT with the necessary values to handle interrupts and exceptions on x86-based systems.

4.vectors.s

This code defines four global labels: alltraps, vector0, vector1, and the jmp instruction to alltraps function.

The alltraps function is likely a handler function for various types of traps and interrupts on the system, such as page faults or system calls. It is defined elsewhere in the codebase and has been made globally accessible by declaring it with the ".globl" directive.

The vector0 and vector1 labels are used to specify the addresses of the interrupt handler functions for interrupt vectors 0 and 1, respectively. An interrupt vector is an index into the Interrupt Descriptor Table (IDT), which specifies the interrupt handler function to be called when the interrupt occurs.

The code following the vector0 and vector1 labels pushes two values onto the stack and then jumps to the alltraps function.

The first pushl instruction pushes a value of 0 onto the stack, and the second pushl instruction pushes a value of 0 or 1 onto the stack depending on which vector is being initialized. These values are likely used as arguments to the alltraps function to help identify which interrupt occurred.

The jmp instruction then jumps to the address of the alltraps function, which will handle the interrupt based on the values pushed onto the stack.

Overall, this code sets up interrupt handlers for two interrupt vectors (0 and 1) by specifying their addresses and pushing values onto the stack before jumping to the interrupt handler function.

5.trapasm.S

This code defines a global label called "alltraps" and begins the implementation of a trap handler function.

The code first saves the values of the segment registers %ds, %es, %fs, and %gs onto the stack using the "pushl" instruction. This is done to preserve these registers as they may be changed by the exception or interrupt that caused the trap.

The "pushal" instruction is then used to push the values of all general-purpose registers (%eax, %ecx, %edx, %ebx, %esp, %ebp, %esi, and %edi) onto the stack in their current state. This creates a trap frame, which is a data structure that contains the state of the machine at the time of the trap. The trap frame can be used by the trap handler function

to examine the state of the machine and to restore the state of the machine after the trap has been handled.

The overall effect of this code is to save the current state of the machine onto the stack, creating a trap frame that can be used by the trap handler function.

OS Questionnaire

Q. List all the blocks/data-structures on the ext2 partition (e.g., superblock, group descriptor, block bitmap, directory data block, inode table, block groups etc) with block size 4KB, that MAY be updated (means changed) when a file of size 8 KB is created.

Sure, here is a detailed explanation of how each of the data structures/blocks listed above may be updated when a file of size 8 KB is created in a 4 KB block size ext2 partition:

- Superblock - When a file is created in the ext2 file system, the superblock may be updated to reflect the new file system size. The superblock is located at a fixed position on the file system and contains important information about the file system, such as the total number of blocks, the number of free blocks, and the block size. When a new file is created, the block count in the superblock needs to be increased to account for the new blocks that are allocated to the file.
- Block Bitmap - The block bitmap keeps track of which data blocks in the file system are currently in use and which are free. When a new file is created, the file system needs to allocate new data blocks to store the file's contents. The block bitmap is updated to indicate that the newly allocated blocks are now in use.
- Inode Bitmap - The inode bitmap keeps track of which inodes in the file system are currently in use and which are free. When a new file is created, the file system needs to allocate a new inode to store information about the file. The inode bitmap is updated to indicate that the newly allocated inode is now in use.
- Inode Table - The inode table contains information about each file and directory in the file system. When a new file is created, the file system needs to allocate a new inode to store information about the file. The inode table is updated to reflect the new inode allocation and to initialize the inode with information about the new file, such as its size and ownership.
- Directory Data Block - A directory data block contains information about the files and directories in a directory. When a new file is created in a directory, an entry for the new file needs to be added to the directory data block. The directory data block is updated to include a new entry for the newly created file.
- Group Descriptor - The group descriptor contains information about each block group in the file system, such as the location of the block bitmap, inode bitmap, and inode table. When a new file is created, the file system needs to allocate new data blocks and an inode for the file, which may require updates to the group descriptor. For example, the group descriptor may need to be updated to indicate that the block bitmap and inode bitmap in a block group have been updated to allocate new blocks and inodes to the new file.

Q. What are the 3 block allocation schemes, explain in detail.

The three block allocation schemes used in file systems are:

1. Contiguous allocation
2. Linked allocation.
3. Indexed allocation.

Here is a detailed explanation of each of these schemes:

- Contiguous allocation: In this scheme, each file is allocated a contiguous set of data blocks on the disk. This means that all the blocks that belong to a file are stored together in a contiguous sequence on the disk. The advantage of this scheme is that it is simple and fast, as the file can be read or written to disk in one go. However, it suffers from external fragmentation, where the disk becomes fragmented and there are no contiguous blocks large enough to hold a new file. Another disadvantage is that the size of the file is limited by the size of the largest contiguous block available on the disk.
- Linked allocation: In this scheme, each data block contains a pointer to the next data block in the file. The last data block of the file contains a special pointer that indicates the end of the file. This scheme eliminates external fragmentation, as each block can be allocated anywhere on the disk, and the file size can be as large as the number of blocks available on the disk. However, it suffers from the problem of disk overhead, as each block needs to store a pointer to the next block, which increases the amount of space required on disk. Moreover, it is relatively slow to access a file, as the system needs to follow a chain of pointers to read or write to the file.
- Indexed allocation: In this scheme, each file has an index block that contains pointers to all the data blocks that belong to the file. This scheme eliminates external fragmentation and reduces disk overhead, as the index block contains only one pointer to each data block, rather than having each data block contain a pointer to the next block. This scheme also allows the file size to be large, as the number of data blocks that can be allocated to a file is limited only by the size of the index block. However, it suffers from the problem of internal fragmentation, where a partially filled data block may be allocated to a file, resulting in wasted space. Also, it is relatively slower than the contiguous allocation scheme, as the system needs to read the index block to locate the data blocks that belong to the file.

Q. List any 3 problems of each of the block allocation schemes (continuous, linked, indexed).

Sure, here are three problems of each of the three block allocation schemes:

Contiguous Allocation:

- External Fragmentation: Over time, contiguous allocation leads to external fragmentation where there may be no contiguous free blocks large enough to hold a new file, even if there are enough free blocks in total.
- Fixed File Size: The size of the file is limited by the size of the largest contiguous block available on the disk.
- Poor Disk Utilization: If the size of a file changes, the entire file needs to be moved to another location on the disk to make room for the expanded file, which may result in unused blocks of space.

Linked Allocation:

- Disk Overhead: Each block needs to store a pointer to the next block, which increases the amount of space required on disk and reduces the number of blocks available for file data.

- Slow Access: It is relatively slow to access a file, as the system needs to follow a chain of pointers to read or write to the file.
- No Random Access: Linked allocation makes it difficult to access data randomly, as the system needs to start at the beginning of the chain and traverse all blocks in the chain to reach the desired block.

Indexed Allocation:

- Internal Fragmentation: A partially filled data block may be allocated to a file, resulting in wasted space.
- Overhead: Index blocks require space on disk to store the index entries, which can take up a significant amount of space if there are many small files.
- Slower Access: Indexed allocation is relatively slower than contiguous allocation, as the system needs to read the index block to locate the data blocks that belong to the file. Moreover, if the index block is not in memory, it may need to be read from disk, which can cause additional delays.

It's worth noting that file systems typically use a combination of these allocation schemes to optimize performance, such as a combination of contiguous and linked allocation.

Q. What is a device driver? Write some 7-8 points that correctly describe need, use, placement, particularities of device drivers.

A device driver is a software program that allows the operating system to communicate with a hardware device. Some key points that describe the need, use, placement, and particularities of device drivers include:

1. **Need:** Device drivers are necessary because hardware devices have their own unique languages, which are often not compatible with the operating system. Device drivers translate these languages into a language that the operating system can understand and use.
2. **Use:** Device drivers are used to control a wide variety of hardware devices such as printers, keyboards, mice, network adapters, and video cards.
3. **Placement:** Device drivers are usually installed on the operating system as a separate module or driver file, which is loaded into the kernel during the boot process.
4. **Particularities:** Device drivers are unique to each hardware device and must be specifically designed to work with that device. They must be optimized for efficiency and speed, as they often operate in real-time and must process large amounts of data quickly.
5. **Device Configuration:** Device drivers are responsible for configuring the hardware device to work with the operating system. This includes setting up interrupts, I/O ports, and memory mapping.
6. **Error Handling:** Device drivers must be designed to handle errors and exceptions, such as device timeouts or data transmission errors.

7. Security: Device drivers must be designed with security in mind, as they have access to sensitive system resources and can be used to gain unauthorized access to a system.
8. Compatibility: Device drivers must be compatible with the operating system and hardware platform. They must be designed to work with different versions of the operating system and hardware configurations.

Device drivers are software programs that facilitate communication between the operating system and hardware devices. In other words, a device driver is a translator that allows the operating system to interact with the physical hardware components, such as printers, scanners, keyboards, mice, network adapters, and other peripherals.

When the computer needs to interact with a hardware device, it sends a request to the device driver, which then communicates with the device to carry out the requested action. For example, if a user wants to print a document, the computer sends a print request to the printer driver, which then translates the request into a language that the printer can understand.

Device drivers are typically developed by the hardware manufacturer or a third-party developer who specializes in writing device drivers. They are written in low-level programming languages, such as C or C++, and they need to be compatible with the specific operating system and version they are intended for.

Device drivers work in kernel mode, which is a privileged mode of operation in the operating system. This means that device drivers have direct access to hardware resources, such as memory and input/output ports, and can interact with them without going through the user mode.

Device drivers can be classified into different categories, such as input drivers, output drivers, storage drivers, and network drivers, based on the type of hardware device they are designed to control.

In summary, device drivers are essential software programs that allow the operating system to communicate with hardware devices. They are responsible for translating the requests from the operating system into actions that can be understood by the hardware device, and they play a critical role in optimizing the performance and reliability of hardware devices on a computer.

The correct answers are: A process becomes zombie when it finishes, and remains zombie until parent calls wait() on it. A process can become zombie if it finishes, but the parent has finished before it. A zombie process occupies space in OS data structures. If the parent of a process finishes before the process itself, then after finishing the process is typically attached to 'init' as parent, init() typically keeps calling wait() for zombie processes to get cleaned up

A zombie process is a process that has completed its execution but still has an entry in the process table. A zombie process occurs when a parent process does not call the "wait" system call to retrieve the exit status of its terminated child process. As a result, the child process is not fully terminated and remains in a "zombie" state.

Yes, a zombie process does occupy space in the operating system's data structures, specifically in the process table. When a process terminates, its entry in the process table is marked as "zombie" until

the parent process retrieves the exit status of the child process using the "wait" system call. While the process is in a zombie state, it continues to occupy an entry in the process table and consumes some system resources, such as memory and process ID. However, the resources consumed by a zombie process are typically minimal and do not have a significant impact on the overall system performance. Once the parent process retrieves the exit status of the child process, the zombie process is fully terminated and its resources are released.

An orphan process, on the other hand, is a process that has lost its parent process. This can happen if the parent process terminates before the child process, or if the parent process crashes unexpectedly. Orphan processes are adopted by the init process, which is the first process that is started during the system boot-up process and is responsible for starting all other processes.

Orphan processes can become zombie processes if the init process does not call the "wait" system call to retrieve the exit status of the terminated child process. However, orphan processes that are adopted by the init process are not typically a problem, as the init process will eventually clean up any zombie processes that are left over.

In summary, a zombie process is a terminated process that has not been fully cleaned up by its parent process, while an orphan process is a process that has lost its parent process. The main difference between the two is that zombie processes are not fully terminated and remain in the process table, while orphan processes are adopted by the init process and are eventually cleaned up.

Q. Which state changes are possible for a process, which changes are not possible?

A process can go through various states during its lifetime, and different operating systems may use different terms for these states. However, the most common process states are:

- New: When a process is first created, it is in the "new" state.
- Ready: In the "ready" state, the process is waiting to be assigned to a processor.
- Running: When the process is assigned to a processor, it is in the "running" state.
- Waiting: In the "waiting" state, the process is waiting for some event, such as an input or output operation or the completion of another process.
- Terminated: When the process completes its execution, it is in the "terminated" state.

It is possible for a process to transition between all these states. For example, a process in the "new" state can transition to the "ready" state when it is ready to be executed. A process in the "ready" state can transition to the "running" state when it is assigned to a processor. A process in the "running" state can transition to the "waiting" state if it needs to wait for some event to occur. Finally, a process in any state can transition to the "terminated" state when it completes its execution.

However, there are some state transitions that are not possible or are not allowed in some operating systems. For example, it is not possible for a process to transition directly from the "terminated" state to any other state. Similarly, in some operating systems, it is not allowed for a process to transition directly from the "running" state to the "new" state without first going through the "terminated" state.

Q. What is mkfs? what does it do? what are different options to mkfs and what do they mean?

mkfs is a command in Unix-like operating systems that is used to create a new file system on a disk partition or storage device. It stands for "make file system".

When you run the mkfs command, it creates a new file system on the specified partition or storage device by writing a new file system structure to the disk. This includes things like the superblock, block group descriptors, block bitmap, inode bitmap, and the inode table.

The mkfs command supports various options that allow you to customize the file system that it creates. Some of the most used options include:

- -t or --type: This option specifies the type of file system to create. For example, you can use mkfs.ext4 to create an ext4 file system.
- -b or --block-size: This option specifies the size of the blocks that the file system will use. The default block size is usually 4KB, but you can specify a different value if you want.
- -i or --inode-size: This option specifies the size of the inodes that the file system will use. Inodes are data structures that are used to represent files and directories on the file system.
- -L or --label: This option allows you to give a label to the file system. The label is used to identify the file system, and it can be up to 16 characters long.
- -O or --features: This option specifies any optional file system features that you want to enable. For example, you can use -O dir_index to enable directory indexing, which can improve performance on large directories.
- -F or --force: This option forces mkfs to create the file system even if there are warnings or errors.

These are just a few of the options that are available with the mkfs command. The full list of options may vary depending on the type of file system that you are creating.

Q. What is the purpose of the PCB? which are the necessary fields in the PCB?

The Process Control Block (PCB) is a data structure used by an operating system to store information about a running process. The PCB serves as a central repository of information about the process, and the operating system can use this information to manage the process and allocate system resources.

The PCB contains a variety of fields that provide information about the process, including:

1. Process ID (PID): A unique identifier assigned to the process by the operating system.
2. Process state: The current state of the process (e.g. running, waiting, etc.).
3. Program counter (PC): The memory address of the next instruction to be executed by the process.

4. CPU registers: The values of the CPU registers that are being used by the process.
5. Memory management information: Information about the process's memory usage, including the base and limit registers.
6. Priority: The priority level of the process, which determines the amount of CPU time that it will receive.
7. Open files: A list of the files that the process has opened, along with their current state and file pointers.
8. Process accounting information: Information about the resources used by the process, such as CPU time, disk I/O, and memory usage.

These are some of the most important fields that are typically found in a PCB. However, the specific fields and their contents may vary depending on the operating system and the specific requirements of the system.

Process accounting information is a set of data that tracks the resource usage of a process. The operating system can use this information to monitor the performance of the system, to bill users for the resources that they consume, or to identify processes that are using excessive resources.

The process accounting information typically includes data such as:

1. CPU time: The amount of time that the process has spent running on the CPU.
2. Memory usage: The amount of memory that the process is using.
3. Disk I/O: The amount of data that the process is reading from or writing to the disk.
4. Network I/O: The amount of data that the process is sending or receiving over the network.
5. Number of page faults: The number of times that the process has requested a page of memory that is not currently in physical memory.
6. Number of system calls: The number of times that the process has made a system call to request a service from the operating system.

This information can be used by system administrators to identify processes that are consuming excessive resources, or to monitor the overall performance of the system. It can also be used for billing purposes, to charge users for the resources that they consume.

Q. Write a C program that uses globals, local variables, static local variables, static global variables, and malloced memory. Compile it. Dump the object code file using objdump -S. Can you see in the output, the separation into stack, heap, text, etc?

Here is an example C program that uses globals, local variables, static local variables, static global variables, and malloced memory:

c

 Copy code

```
#include <stdio.h>
#include <stdlib.h>

// Global variable
int global_var = 5;

// Static global variable
static int static_global_var = 10;

int main() {
    // Local variable
    int local_var = 15;

    // Static local variable
    static int static_local_var = 20;

    // Malloced memory
    int *ptr = malloc(sizeof(int));
    *ptr = 25;

    printf("Global variable: %d\n", global_var);
    printf("Static global variable: %d\n", static_global_var);
    printf("Local variable: %d\n", local_var);
    printf("Static local variable: %d\n", static_local_var);
    printf("Malloced memory: %d\n", *ptr);

    free(ptr);
    return 0;
}
```

When compiled with `gcc -g -o program program.c`, the object code file can be dumped using `objdump -S program`. The output will show the separation into different sections, such as `.text` (code), `.data` (global and static variables), and `.bss` (uninitialized global and static variables).

Here is an example of the output:

The screenshot shows a terminal window with the title 'yaml'. The content of the terminal is as follows:

```
program: file format elf64-x86-64

Disassembly of section .text:

000000000400526 <main>:
400526: 55                      push  %rbp
400527: 48 89 e5                mov    %rsp,%rbp
40052a: 48 83 ec 10             sub    $0x10,%rsp
40052e: c7 45 fc 0f 00 00 00    movl   $0xf,-0x4(%rbp)
400535: 8b 05 b9 0c 20 00       mov    0x200cb9(%rip),%eax      # 6011f4
40053b: 89 45 f8                mov    %eax,-0x8(%rbp)
40053e: b8 05 00 00 00          mov    $0x5,%eax
400543: 89 45 f4                mov    %eax,-0xc(%rbp)
400546: c7 45 f0 14 00 00 00    movl   $0x14,-0x10(%rbp)
40054d: 8b 05 a6 0c 20 00       mov    0x200ca6(%rip),%eax      # 6011f8
400553: 89 45 ec                mov    %eax,-0x14(%rbp)
400556: 48 8d 45 f0             lea    -0x10(%rbp),%rax
40055a: 48 89 c7                mov    %rax,%rdi
40055d: e8 1e ff ff ff          callq  400580 <malloc@plt>
400562: 48 89 45 e8             mov    %rax,-0x18(%rbp)
400566: 48 8b 45 e8             mov    -0x18(%rbp),%
```

When we run the command `objdump -S <executable_file>`, we get the assembly code generated by the compiler. The output of objdump contains several sections, including the text, data, bss, rodata, and comment sections.

- The text section contains the machine code for the program's executable instructions. This section is typically marked as read-only.
- The data section contains initialized global and static variables. These variables are stored in memory and are accessible throughout the entire program.
- The bss section contains uninitialized global and static variables. This section is typically initialized to zero by the operating system when the program is loaded into memory.
- The rodata section contains read-only data, such as string literals.
- The comment section contains any comments that were present in the original source code.

The stack and heap are runtime memory areas used by the program.

- The stack is used for storing local variables and function call frames. It is a last-in-first-out (LIFO) data structure that grows downwards in memory.
- The heap is used for dynamic memory allocation, typically using functions such as malloc, calloc, and realloc. The heap grows upwards in memory.

The stack and heap are not explicitly visible in the objdump output, as they are created and destroyed dynamically at runtime. However, we can get an idea of how they are used by looking at the assembly code and observing the instructions that modify the stack and heap pointers.

In summary, the objdump output provides a detailed view of the compiled program's executable instructions, initialized and uninitialized global/static variables, and read-only data. The stack and heap are runtime memory areas that are not explicitly visible in the objdump output, but their usage can be inferred by analysing the assembly code.

Q. Which parts of a C program ((typedef, #define, #include, functions, local vars, statics, globals, #ifdef, ...) etc occupy RAM when it's a process, and where (code, data, stack, ..)?

When a C program is compiled and run as a process, it occupies different parts of the RAM memory. Here's where each part typically resides:

- **typedef:** Typedefs don't take up memory during runtime. They are simply aliases for existing types and are resolved during compilation.
- **#define:** Like typedefs, #defines don't take up memory during runtime. They are simply macros that are resolved during compilation.
- **#include:** The code included by a #include directive is compiled into the program's text section.
- **Functions:** Function code is compiled into the program's text section.
- **Local variables:** Local variables are stored on the stack. Each function call creates a new stack frame, which includes the function's local variables.
- **Static variables:** Static variables that are defined inside a function are stored on the data section of the program, just like global variables. Static variables that are defined outside of functions are stored in the BSS section of the program.
- **Global variables:** Global variables are stored in the data section of the program. Initialized global variables are stored in the data section, while uninitialized global variables are stored in the BSS section.
- **#ifdef:** Preprocessor directives like #ifdef don't take up memory during runtime. They are simply evaluated during compilation.

In summary, the text section contains the compiled code of the program, the data section contains initialized global and static variables, the BSS section contains uninitialized global and static variables, and the stack contains local variables and function call frames.

When a C program is compiled and run as a process, it is divided into several parts that occupy different areas of memory.

Here are the essential parts of a C program and where they typically reside in memory:

- **Preprocessor directives (#define, #include, #ifdef, etc.):** These directives are processed by the preprocessor before compilation and don't take up memory during runtime.
- **Type definitions (typedef):** These are aliases for existing types and don't take up memory during runtime. They are resolved during compilation.

- Global variables: Global variables are stored in the data section of the program. Initialized global variables are stored in the data section, while uninitialized global variables are stored in the BSS (Block Started by Symbol) section.
- Static variables: Static variables that are defined inside a function are stored on the data section of the program, just like global variables. Static variables that are defined outside of functions are stored in the BSS section of the program.
- Function code: Function code is compiled into the program's text section. This section contains the executable code of the program.
- Local variables: Local variables are stored on the stack. Each function call creates a new stack frame, which includes the function's local variables.
- Heap memory: Dynamically allocated memory using functions like malloc() and calloc() is stored on the heap. This memory is managed by the program at runtime.
- Program arguments and environment variables: These are stored on the stack, along with local variables.

When the C program is compiled, the compiler generates an object file, which contains the code and data of the program. The linker then combines the object file with any libraries and generates an executable file.

During runtime, the operating system loads the program into memory and sets up the stack, heap, and other necessary data structures. The program then begins executing from the main() function, and any global variables are initialized before the main() function is called.

In summary, a C program is divided into several parts, each of which occupies a different area of memory. Global variables and static variables are stored in the data and BSS sections of the program, function code is stored in the text section, local variables are stored on the stack, and dynamically allocated memory is stored on the heap.

Q. Describe the role of CPU (MMU), kernel, compiler in the overall task of "memory management" for a process/program.

Memory management is the process of allocating and deallocating memory for a process or program. This process is essential to ensure that the program runs smoothly and that the system resources are used efficiently. The main components involved in memory management are the CPU's MMU, the kernel, and the compiler.

CPU (MMU): The Memory Management Unit (MMU) is a component of the CPU responsible for virtual memory management. It maps virtual addresses used by a process into physical memory addresses. When a process accesses memory, the MMU checks the virtual address to see if it is valid and maps it to the appropriate physical address. This allows the operating system to manage the physical memory more efficiently by allowing multiple processes to access the same physical memory without interfering with each other.

Kernel: The kernel is the core component of the operating system and is responsible for managing system resources, including memory. It provides the processes with virtual address spaces and controls the allocation and deallocation of physical memory. The kernel also provides mechanisms for inter-process communication and synchronization, which may involve sharing of memory between processes.

The kernel manages memory through a set of data structures, such as the page table and the buddy allocator. The page table maps the virtual memory of a process to the physical memory of the system. It keeps track of the page frame number, the physical address of the page in memory, and other information related to the page. The buddy allocator is used to manage the physical memory, which is divided into fixed-sized chunks of pages. The allocator maintains a free list of memory chunks and allocates chunks to processes on demand.

Compiler: The compiler is responsible for generating machine code from the program's source code. It analyses the program's memory usage and generates instructions that use the memory efficiently. For example, the compiler may optimize memory usage by reusing memory for variables or by storing variables in CPU registers rather than in memory.

The compiler also performs several optimization techniques, such as dead code elimination and loop unrolling, to reduce the program's memory footprint. It may also use data structures such as arrays and linked lists to manage the program's data efficiently.

In summary, the CPU's MMU, the kernel, and the compiler work together to manage memory for a process or program. The MMU maps virtual addresses to physical addresses, the kernel manages the allocation and deallocation of physical memory and provides virtual address spaces to processes, and the compiler generates efficient code that uses memory efficiently. The efficient use of memory ensures that the program runs smoothly and that the system resources are used effectively.

Q. What is the difference between a named pipe and un-named pipe? Explain in detail.

A pipe is a method of interprocess communication that allows one process to send data to another process. A named pipe and an unnamed pipe are two types of pipes that differ in their features and usage. Here is a detailed explanation of their differences:

1. **Naming:** An unnamed pipe, also known as an anonymous pipe, has no external name, whereas a named pipe, also known as a FIFO (First In First Out), has a name that is visible in the file system.
2. **Persistence:** A named pipe persists even after the process that created it has terminated, and can be used by other processes. In contrast, an unnamed pipe exists only as long as the process that created it is running.
3. **Communication:** An unnamed pipe is used for communication between a parent process and its child process or between two processes that share a common ancestor. On the other hand, a named pipe can be used for communication between any two processes that have access to the same file system.

4. Access: A named pipe can be accessed by multiple processes simultaneously, allowing for one-to-many communication. In contrast, an unnamed pipe can only be accessed by the two processes that share it, allowing for one-to-one communication.
5. Synchronization: Named pipes provide a method for synchronizing the flow of data between processes, as each write to a named pipe is appended to the end of the pipe and each read retrieves the oldest data in the pipe. In contrast, unnamed pipes do not provide any built-in synchronization mechanism.

In summary, a named pipe is a named, persistent, two-way, interprocess communication channel that can be accessed by multiple processes, while an unnamed pipe is a temporary, one-way, communication channel that can only be accessed by the two processes that share it.

An unnamed pipe, also known as an anonymous pipe, is a temporary, one-way communication channel that is used for interprocess communication between a parent process and its child process or between two processes that share a common ancestor.

An unnamed pipe is created using the `pipe()` system call, which creates a pair of file descriptors. One file descriptor is used for reading from the pipe, and the other is used for writing to the pipe. The file descriptors are inherited by the child process when it is created by the parent process.

An unnamed pipe has a fixed size buffer that is used to store data that is written to the pipe. When the buffer is full, further writes to the pipe will block until space is available in the buffer. The buffer is emptied when data is read from the pipe, creating more space in the buffer for new data to be written.

An unnamed pipe is useful for implementing simple communication between a parent and child process. For example, a parent process may create a child process to perform some work and use an unnamed pipe to receive the results from the child process. However, unnamed pipes have several limitations, such as being limited to one-way communication, having a fixed buffer size, and not providing any synchronization mechanism. These limitations can be overcome by using other interprocess communication methods, such as named pipes, sockets, or message queues.

A named pipe, also known as a FIFO (First-In-First-Out), is a named, one-way communication channel that can be used for interprocess communication between two or more unrelated processes.

A named pipe is created using the `mkfifo()` system call, which creates a special file in the file system that can be opened for reading and writing by multiple processes. Once created, a named pipe can be used like any other file, but the data that is written to the file is not stored in the file system, but is passed directly to the process that is reading from the pipe.

Named pipes are useful for implementing more complex interprocess communication scenarios than can be achieved with unnamed pipes. For example, named pipes can be used to implement a server-client architecture, where multiple clients can connect to a server using a named pipe, and the server can respond to requests from the clients by writing data to the pipe. Named pipes can also be used to implement message passing between unrelated processes, where one process writes messages to a named pipe, and other processes read the messages from the pipe.

In general, named pipes can be used for bidirectional communication between processes, although they can also be used for unidirectional communication if needed. The direction of communication

depends on how the named pipe is implemented and used by the processes that are communicating through it.

Named pipes provide a simple way for two or more processes to communicate with each other by reading and writing data to a common pipe. Each named pipe has two ends, one for reading and one for writing, which can be used by different processes for communication.

For example, process A could open a named pipe for writing and write data to it, while process B could open the same named pipe for reading and read the data written by process A. In this case, communication is unidirectional from A to B. However, if process A also opens the named pipe for reading and process B also opens the named pipe for writing, then bidirectional communication can occur between the two processes.

It's important to note that the implementation of named pipes may differ between operating systems and programming languages, and this can affect whether bidirectional communication is possible or not. Some implementations may restrict the direction of communication, while others may allow bidirectional communication by default.

Named pipes provide several benefits over unnamed pipes, such as being accessible by multiple processes, providing a reliable means of communication, and not requiring a parent-child relationship between the communicating processes.

Q. What are privileged instructions? Why are they required? What is their relationship to 2 modes of CPU execution?

Privileged instructions are instructions in a computer's instruction set architecture (ISA) that can only be executed in privileged mode. Privileged mode is a mode of operation in the CPU that allows certain instructions and actions to be performed that are not allowed in user mode.

Privileged instructions are required to ensure the security and integrity of the operating system and the system as a whole. These instructions are used to perform critical functions such as managing system resources, controlling interrupts and exceptions, accessing privileged hardware devices, and modifying system control registers. Without privileged instructions, an application or user could potentially modify critical system resources and cause system instability or security breaches.

The two modes of CPU execution are user mode and privileged mode. User mode is the normal mode of operation, in which most applications and processes run. In user mode, only a subset of instructions is available, and certain instructions that could cause harm to the system or other processes are prohibited. In privileged mode, all instructions are available, including privileged instructions, and the CPU has access to system resources that are not available in user mode.

The relationship between privileged instructions and the two modes of CPU execution is that privileged instructions can only be executed in privileged mode, which is reserved for the operating system kernel and trusted system processes. By limiting access to privileged instructions to only trusted processes running in privileged mode, the operating system can ensure the security and stability of the system as a whole.

To resolve the path name /a/b/c on an ext2 filesystem, the following steps are involved:

1. The root directory is accessed by the kernel, as all path names start at the root directory.
2. The kernel searches the root directory for the directory entry for the directory "a".
3. The kernel reads the inode for directory "a", which contains a list of directory entries.
4. The kernel searches this list for the directory entry for the directory "b".
5. The kernel reads the inode for directory "b", which contains a list of directory entries.
6. The kernel searches this list for the directory entry for the directory "c".
7. The kernel reads the inode for directory "c", which contains the file data or another list of directory entries.

If any of the directories or files in the path name are not found, the kernel will return an error. The kernel uses the file system's inode and directory data structures to navigate the file system and locate the files and directories specified in the path name. This process is repeated for every path name that the kernel encounters when accessing files on the file system.

Q. Explain what happens in pre-processing, compilation and linking and loading.

Pre-processing, compilation, linking, and loading are the four stages involved in the process of compiling and running a program. Each stage plays a crucial role in the final outcome of the program. Let's discuss each stage in detail:

1. Pre-processing: The first stage in compiling a program is pre-processing. In this stage, the pre-processor reads the source code and performs certain operations on it. These operations include:
 - Removing comments from the code
 - Expanding macros
 - Including header files
 - Defining constants

The pre-processor generates an intermediate code after performing these operations, which is used by the compiler in the next stage.

2. Compilation: The second stage is compilation. In this stage, the compiler reads the intermediate code generated by the pre-processor and translates it into assembly language or machine code. The compiler performs various optimizations to make the code more efficient. The output of the compilation stage is an object file, which contains machine code in binary format.

3. Linking: The third stage is linking. In this stage, the linker combines the object files generated by the compiler and resolves any external references between them. External references are the references to functions or variables defined in other files. The linker generates an executable file after resolving all the external references.
4. Loading: The final stage is loading. In this stage, the operating system loads the executable file into memory and starts executing it. The loader allocates memory for the program and maps the code and data sections of the program into memory. It also sets up the environment for the program to run, such as the stack and heap.

Q. Why is the kernel called an event-drive program? Explain in detail with examples.

In summary, pre-processing, compilation, linking, and loading are the four stages involved in compiling and running a program. Each stage has its own set of tasks and produces a specific output, which is used as an input for the next stage. The final output is an executable file, which can be run by the operating system.

The kernel is called an event-driven program because it responds to various events that occur in the system. Events may include interrupts generated by hardware devices such as disk drives, network adapters, or input/output devices. The kernel must handle these events and take appropriate actions, such as scheduling processes to run, allocating memory to processes, or responding to system calls from user-space applications.

The kernel is constantly monitoring the system for events, and when an event occurs, the appropriate handler function is called to handle the event. The kernel must be able to handle multiple events simultaneously and prioritize them according to their importance and urgency.

The event-driven nature of the kernel is what allows it to be responsive to the needs of the system and provide a stable and reliable operating environment for user-space applications.

The kernel is called an event-driven program because it responds to various events or interrupts that occur in the system. These events can be generated by hardware devices or by software running on the system. The kernel is designed to handle these events and take appropriate actions to ensure the system functions correctly.

Here are some examples of events that the kernel can handle:

1. Interrupts: When a hardware device generates an interrupt, the kernel will pause the current task and handle the interrupt. For example, if the user presses a key on the keyboard, an interrupt is generated, and the kernel will handle it by updating the appropriate data structures and waking up any processes waiting for input.

2. Signals: When a process sends a signal to another process, the kernel will handle the signal and take appropriate action. For example, if a process receives a SIGINT signal (generated by pressing Ctrl+C), the kernel will terminate the process.
3. System calls: When a process makes a system call, the kernel will handle the call and execute the appropriate code. For example, if a process calls the open() system call to open a file, the kernel will handle the call by checking permissions and allocating file descriptors.
4. Memory management: When a process requests memory, the kernel will handle the request and allocate the appropriate amount of memory. For example, if a process requests more memory using malloc(), the kernel will handle the request by allocating the memory and updating the process's memory map.

Overall, the kernel is responsible for handling events and ensuring that the system functions correctly. This requires a lot of coordination between different parts of the kernel and the various hardware and software components in the system.

Q. What are the limitations of segmentation memory management scheme?

Segmentation memory management scheme has the following limitations:

1. External Fragmentation: As memory is allocated in variable-sized segments, the unused memory between two allocated segments may be too small to hold another segment. This leads to external fragmentation and a loss of memory.
2. Internal Fragmentation: In segmentation, memory is allocated in variable-sized segments. Therefore, there is a possibility of having unused memory within a segment, which is called internal fragmentation.
3. Difficulty in Implementation: Segmentation memory management is more complex than other memory management schemes. It requires a sophisticated hardware support and a more complex software to manage it.
4. Limited Sharing: In a segmentation memory management scheme, it is difficult to share segments between processes, because each process has its own segment table. This makes it difficult for processes to share data and code.
5. Security Issues: Segmentation memory management requires more security checks than other memory management schemes because it needs to ensure that one process does not access the memory of another process. This can lead to increased overhead and slower performance.

Overall, while segmentation can be useful for managing memory in certain situations, it has several limitations that must be taken into consideration.

Q. How is the problem of external fragmentation solved?

The problem of external fragmentation can be solved in various ways:

1. Compaction: Compaction involves moving all the allocated memory blocks to one end of the memory and freeing up the unallocated space at the other end. This requires copying the contents of the allocated blocks to a new location, which can be time-consuming and can cause delays in the program execution.
2. Paging: Paging involves dividing the memory into fixed-size pages and allocating the memory on a page-by-page basis. This reduces external fragmentation by eliminating the need for contiguous memory allocation. However, it can lead to internal fragmentation as not all the space in a page may be used.
3. Buddy allocation: Buddy allocation involves dividing the memory into blocks of fixed sizes and allocating them in powers of two. When a block is freed, it is combined with its buddy (a block of the same size) to create a larger block, which can then be allocated to another process. This reduces external fragmentation by ensuring that only blocks of the same size are combined.
4. Segmentation with paging: Segmentation with paging involves dividing the memory into variable-sized segments and then dividing each segment into fixed-sized pages. This combines the benefits of both segmentation and paging and reduces external fragmentation.

Overall, the solution to external fragmentation depends on the specific requirements of the system and the trade-offs between memory utilization and program performance.

Q. Does paging suffer from fragmentation of any kind?

Paging does not suffer from external fragmentation because the pages are of fixed size and can be easily allocated or deallocated without leaving any holes in the memory. However, it can suffer from internal fragmentation, which occurs when a page is allocated to a process, but not all of the space in the page is used. In such cases, some space in the page is wasted, leading to internal fragmentation. This can be minimized by choosing the page size carefully to balance the trade-off between internal fragmentation and page table overhead.

Internal fragmentation in paging occurs when the memory allocated to a process is slightly larger than the actual size required by the process. As a result, some portion of the allocated page remains unused. This unused portion of the page is referred to as internal fragmentation.

For example, if a process requires only 3KB of memory but is allocated an entire page of 4KB, the remaining 1KB of memory will be unused and wasted, resulting in internal fragmentation.

Internal fragmentation can occur in both dynamic and static memory allocation schemes. In dynamic allocation, it may occur due to the allocation of memory blocks that are slightly larger than required, while in static allocation, it may occur due to the allocation of fixed-size memory blocks that are larger than required by some processes.

Segmentation with paging is a memory management technique used by operating systems to handle the virtual memory of a computer system. It is a hybrid method that combines the benefits of both segmentation and paging, while overcoming some of their individual limitations.

In this technique, the logical address space of a process is divided into variable-length segments, each representing a different type of memory region such as code, data, stack, heap, etc. Each segment has its own base address and length and is further divided into fixed-size pages of a uniform size.

The translation of a logical address into a physical address is performed in two stages. First, the segment number is used to index into a segment table, which contains the base address and length of each segment. Then, the page number is used to index into a page table, which maps the page to a physical frame in memory. The physical address is obtained by combining the base address of the segment and the physical frame number.

The segmentation with paging technique provides several benefits:

1. It allows each process to have its own logical address space, which is protected from other processes.
2. It provides flexibility in allocating and managing memory, as segments can be dynamically resized and relocated.
3. It reduces external fragmentation by allowing the operating system to allocate pages from any free frame in memory.
4. It allows for sharing of memory between processes by mapping the same physical page to multiple logical addresses.

However, this technique also has some limitations:

1. It can suffer from internal fragmentation, as a segment may not exactly fit into an integral number of pages.
2. It requires more complex hardware support for the segment and page tables.
3. It can result in slower memory access times due to the additional level of indirection in the address translation process.

Overall, segmentation with paging is a powerful memory management technique that provides a balance between the flexibility and protection of segmentation and the efficiency and simplicity of paging.

Q. What is segmentation fault? why is it caused? who detects it and when? how is the process killed after segmentation fault?

A segmentation fault, also known as a segfault, is an error in the memory management system of a computer. It occurs when a program tries to access a memory location that it is not authorized to access, causing the operating system to terminate the program.

Segmentation faults are typically caused by programming errors such as dereferencing a null pointer or accessing an array out of bounds. They can also be caused by hardware errors or software bugs.

The operating system detects the segmentation fault when the program tries to access an unauthorized memory location. At this point, the operating system terminates the program and returns an error message to the user.

After a segmentation fault, the process is killed by the operating system. The operating system deallocates any resources that were being used by the process, such as memory and open files. The error message is typically written to a log file or displayed to the user, depending on the configuration of the system.

Q. What is the meaning of "core dumped"?

A core dump is a file that gets automatically generated by the Linux kernel after a program crashes. This file contains the memory, register values, and the call stack of an application at the point of crashing. In computing, a core dump, memory dump, crash dump, storage dump, system dump, or ABEND dump consists of the recorded state of the working memory of a computer program at a specific time, generally when the program has crashed or otherwise terminated abnormally.

"Core dumped" is a message displayed in the terminal or console after a program crashes due to a segmentation fault or other fatal error. It means that the program attempted to access memory that it was not allowed to access, and as a result, the operating system terminated the program and saved a snapshot of the program's memory at the time of the crash, known as a "core dump". The core dump can be analysed by a programmer or system administrator to determine the cause of the crash and fix any bugs or issues in the program.

Q. in this program: int main() { int a[16], i; for(i = 0; ; i++) printf("%d\n", a[i]); } why does the program not segfault at a[16] or some more values?

The reason why the program does not segfault at a[16] or some more values is because the array a is a local variable and is allocated on the stack. When the program tries to access a[16] or beyond, it goes beyond the allocated memory of the array and accesses some other part of the stack, which may or may not cause a segmentation fault.

However, since the stack is typically quite large, the program may be able to access many more elements beyond the end of the array before a segmentation fault occurs. Additionally, the behavior of accessing memory beyond the bounds of an array is undefined, which means that the program could produce unpredictable results or crash at any point.

Q. What is voluntary context switch and non-voluntary context switch on Linux? Name 2 processes which have lot of non-voluntary context switches compared to voluntary.

In Linux, voluntary context switch occurs when a process explicitly yields the CPU, for example by calling a blocking system call like sleep(). On the other hand, non-voluntary context switch occurs when the running process is forcibly pre-empted by the kernel scheduler due to some other process becoming runnable or due to the expiration of the process's time slice.

Two processes that have a lot of non-voluntary context switches compared to voluntary are:

1. I/O-bound processes: These processes spend most of their time waiting for I/O operations to complete, and hence are often pre-empted by the kernel when other processes become runnable. Examples include network servers, database servers, and file servers.
2. Real-time processes: These processes have strict timing constraints and need to be executed within a certain time frame. If the kernel scheduler fails to schedule the process in time, it may be pre-empted forcefully to avoid violating the real-time constraints. Examples include multimedia applications, flight control systems, and medical equipment.

1. Context switching

This code snippet appears to be implementing a "context switch" operation, which is a fundamental component of multitasking operating systems.

The purpose of a context switch is to save the current state of a process (its register values and program counter) and then restore the state of a different process so that it can resume execution from where it last left off.

Let's break down the code step by step:

```
movl 4(%esp), %eax
```

```
movl 8(%esp), %edx
```

These instructions load the first two arguments passed to the function into the `eax` and `edx` registers. Assuming this is a function implementing a context switch, we can assume that these arguments represent the addresses of the old and new stacks that will be used to save the register state of the old process and load the state of the new process.

```
# Save old callee-saved registers
```

```
pushl %ebp
```

```
pushl %ebx
```

```
pushl %esi
```

```
pushl %edi
```

These instructions save the values of the callee-saved registers onto the stack. These registers are typically saved by the callee (i.e., the function being called) so that they can be modified without interfering with the caller's code. In this case, the function implementing the context switch is acting as the callee, so it saves these registers to ensure that they are not accidentally overwritten when switching contexts.

```
# Switch stacks
```

```
movl %esp, (%eax)
```

```
movl %edx, %esp
```

These instructions perform the actual context switch. The current value of the stack pointer (`esp`) is saved into the memory location pointed to by the `eax` register (which contains the address of the old stack), and the `esp` register is loaded with the value in `edx` (which contains the address of the new stack). This effectively switches the stack that the processor will use for subsequent instructions.

```
# Load new callee-saved registers
```

```
popl %edi
```

```
popl %esi
```

```
popl %ebx
```

```
popl %ebp
```

```
ret
```

These instructions restore the callee-saved registers from the new stack. Since the function implementing the context switch is acting as the caller, it needs to restore these registers before returning to the newly switched-to process. The `ret` instruction then returns control to the newly switched-to process, which will resume execution from where it last left off.

Overall, this code implements a basic context switch operation by saving the callee-saved registers, switching stacks, restoring the callee-saved registers, and returning control to the new process.

1.Diff between process and threads

In computing, a process is an instance of a computer program that is being executed. A process is a self-contained execution environment that includes the program code, memory, data, and system resources like CPU time, I/O resources, and network access. Each process has its own virtual memory space and runs in a separate address space, which means that a process cannot access the memory of another process directly.

On the other hand, a thread is a lightweight execution unit within a process. A thread shares the same memory and resources as the process that it belongs to. Threads are used to perform multiple tasks concurrently within a single process, which improves the overall performance and responsiveness of the application.

In summary, the key difference between a process and a thread is that a process is an independent execution environment that consists of one or more threads, while a thread is a lightweight execution unit within a process that shares the same memory and resources as the process. Processes are isolated from each other and communicate through inter-process communication mechanisms, while threads share memory and resources and communicate through shared variables and synchronization primitives.

2.no kernel threads one process then many to one mapping how?

In a system without kernel threads, there is only one execution context, which is provided by the process. This means that the process is responsible for managing all of its threads and allocating CPU time to each thread. In this scenario, a user-level thread library can be used to implement concurrency within the process.

A user-level thread library provides a set of functions that allow the programmer to create, schedule, and manage threads within the process. The library maintains a table of user-level thread structures, which contain information about each thread, such as its state, CPU time used, and priority. The library also provides a scheduler that determines which

thread should be executed next, based on the scheduling algorithm implemented by the programmer.

Since there is no kernel support for threads in this scenario, the threads are implemented entirely in user space. This means that the operating system is not aware of the threads, and cannot allocate CPU time to them directly. Instead, the user-level thread library must use blocking system calls or other mechanisms to ensure that each thread gets a fair share of CPU time.

Overall, the many-to-one mapping of user-level threads to the single process is possible because the threads are not directly managed by the kernel. Instead, the user-level thread library provides an abstraction layer that allows multiple threads to be managed within a single process. However, this approach has some limitations, such as the inability to take advantage of multiple CPUs or to run threads in kernel mode.

3. 2 level model in threading

The two-level model in threading refers to a model that includes both user-level threads and kernel-level threads. In this model, user-level threads are created and managed by a user-level thread library, while kernel-level threads are created and managed by the operating system's kernel.

The two-level model provides a number of benefits compared to a single-level model, where all threads are managed at the same level. For example, it allows user-level threads to be implemented efficiently, since they can be scheduled and switched without requiring kernel intervention. This reduces the overhead associated with thread management and improves the performance of the application.

At the same time, the two-level model also provides the benefits of kernel-level threads, such as the ability to take advantage of multiple processors and to execute in kernel mode. Kernel-level threads are managed by the operating system's scheduler, which can assign threads to different processors and optimize their execution to improve performance.

Overall, the two-level model in threading provides a balance between the efficiency of user-level threads and the flexibility and performance of kernel-level threads. It allows applications to take advantage of the benefits of both types of threads, and provides a flexible framework for implementing concurrent and parallel programming.

4. difference between many to many and 2 level model

The many-to-many model and the two-level model are two different approaches to threading, each with its own advantages and disadvantages.

The many-to-many model allows multiple user-level threads to be mapped to multiple kernel-level threads, providing a greater degree of concurrency and parallelism than the two-level model. In this model, the thread library schedules user-level threads on a pool of kernel-level threads, which can be assigned to different processors by the operating system's scheduler. This allows multiple threads to run in parallel on multiple processors, providing better performance and scalability than the two-level model.

However, the many-to-many model can also be more complex to implement than the two-level model, since it requires coordination between the user-level thread library and the operating system's kernel. The thread library must manage a pool of kernel-level threads, and ensure that each user-level thread has access to a sufficient number of kernel-level threads to achieve good performance. Additionally, since the user-level threads are not directly associated with kernel-level threads, there can be additional overhead involved in context switching and synchronization.

In contrast, the two-level model provides a simpler and more predictable threading model. User-level threads are created and managed by a user-level thread library, and are mapped directly to kernel-level threads by the operating system's scheduler. This provides good performance and scalability, especially when running on a system with a small number of processors.

Overall, the choice between the many-to-many model and the two-level model for threading depends on the specific requirements of the application. If the application requires a high degree of concurrency and parallelism, the many-to-many model may be more appropriate. If the application is simpler and requires good performance on a small number of processors, the two-level model may be a better choice.

5. fork duplicates threads?

The `fork()` system call duplicates the entire process, including all threads that are currently running within that process. This means that after the `fork()` call, the child process has an exact copy of the parent process's entire memory space, including all threads that were running at the time of the `fork()` call.

However, it's worth noting that the child process does not inherit the thread IDs of the parent process. Instead, each thread in the child process is given a new thread ID. This means that the child process has a new set of threads that are identical to the threads in the parent process, but with different IDs.

It's also important to keep in mind that after the `fork()` call, each process (the parent and the child) has its own copy of the memory space. This means that any modifications made to the memory space in one process do not affect the memory space in the other process. Similarly, any changes made to a thread in one process do not affect the corresponding thread in the other process.

Overall, the `fork()` system call duplicates the entire process, including all threads, but creates a new set of thread IDs for the child process.

6. pthreads functions

`pthread_key_create()`, `pthread_key_delete()`, `pthread_setspecific()`, and `pthread_getspecific()` are functions in the POSIX threads (pthreads) library that are used to manage thread-specific data. These functions allow a program to associate thread-specific data with a specific key, which can be used to retrieve the data later from any thread.

Here's a brief explanation of each of these functions:

1. `pthread_key_create()` - This function creates a new thread-specific data key, which can be used to store thread-specific data. The

function takes two parameters: a pointer to a `pthread_key_t` variable, which will hold the key value, and an optional destructor function, which will be called automatically when a thread exits and the data associated with the key is no longer needed. The function returns zero on success, or an error code if the key could not be created.

2. `pthread_key_delete()` - This function deletes a previously created thread-specific data key. The function takes one parameter: the key to be deleted. Any data associated with the key in all threads will also be deleted. The function returns zero on success, or an error code if the key could not be deleted.
3. `pthread_setspecific()` - This function associates a value with a specific key for the current thread. The function takes two parameters: the key to associate the value with, and the value to associate with the key. The value can be a pointer to any data type. Each thread can associate a different value with the same key. The function returns zero on success, or an error code if the value could not be set.
4. `pthread_getspecific()` - This function retrieves the value associated with a specific key for the current thread. The function takes one parameter: the key to retrieve the value for. The function returns the value associated with the key, or `NULL` if no value has been associated with the key in the current thread.

Overall, these functions allow a program to create, manage, and access thread-specific data, which can be useful in situations where data needs to be shared between multiple threads without the risk of interference or synchronization issues.

7. upcall handler

This code snippet outlines a simple thread library implementation that uses lightweight processes (LWPs) to schedule threads. Here's a brief explanation of each of the functions:

1. `upcall_handler()` - This is a callback function that will be invoked by the operating system whenever a blocking system call completes. The purpose of this function is to create a new LWP and schedule

any waiting threads onto the new LWP, so that they can continue executing.

2. `th_setup(int n)` - This function initializes the thread library with a maximum number of LWPs (specified by the parameter `n`). The `max_LWP` variable is set to `n`, and the `curr_LWP` variable is initialized to 0. The `register_upcall(upcall_handler)` function is also called to register the `upcall_handler()` function with the operating system, so that it will be called whenever a blocking system call completes.
3. `th_create(...., fn,)` - This function creates a new thread, with the specified function `fn`. If there are available LWPs (i.e. `curr_LWP < max_LWP`), a new LWP is created using `create_LWP`. The thread is then scheduled onto one of the available LWPs, using `schedule fn on one of the LWP`.

Overall, this thread library implementation uses LWPs to schedule threads, and employs an upcall mechanism to handle blocking system calls. When a system call blocks, the `upcall_handler()` function is invoked, and a new LWP is created to handle the waiting threads. This approach allows for efficient and scalable thread scheduling, while minimizing the risk of thread interference and synchronization issues.

1. KINIT1:

The function initlock is called to initialize a lock on a structure called kmem. This lock is used to ensure that multiple threads cannot access the memory allocator (i.e., kmem) at the same time, which could cause problems like data corruption.

The kmem.use_lock variable is set to 0, which means that the lock is not currently in use. This variable is used to track whether the lock is currently being used, so that threads can avoid trying to access kmem at the same time.

The freerange function is called to free a range of memory from vstart to vend. This function is used to initialize the kernel memory allocator, which manages the allocation and deallocation of memory in the kernel.

Overall, this code is used to initialize the kernel memory allocator and ensure that it can be safely accessed by multiple threads.

2. KINIT2:

The freerange function is called to free a range of memory from vstart to vend. This function is used to initialize the kernel memory allocator, which manages the allocation and deallocation of memory in the kernel. This is similar to what happens in kinit1.

The kmem.use_lock variable is set to 1, which means that the lock should now be used to protect access to the memory allocator. This variable is used to track whether the lock is currently being used, so that threads can avoid trying to access kmem at the same time.

By setting kmem.use_lock to 1, the kernel memory allocator is now protected by a lock, which ensures that only one thread can access it at any given time. This is important to avoid race conditions, where multiple threads might try to allocate or deallocate memory at the same time and cause conflicts.

Overall, kinit2 is used to finalize the initialization of the kernel memory allocator and enable thread-safe access to it.

3. FREE RANGE:

The freerange function takes two void pointers vstart and vend, which represent the start and end addresses of a range of memory that needs to be freed.

The PGROUNDUP function is called to round up vstart to the nearest page boundary, which is a multiple of PGSIZE. PGSIZE is a constant that represents the size of a page of memory. The resulting address is stored in a pointer p.

A loop is executed that iterates over the range of memory from p to vend, with a step of PGSIZE each time. The loop body calls the kfree function on each page in the range. kfree is a function that frees a page of memory in the kernel.

Once the loop has finished, all pages in the specified range have been freed.

Overall, this code is used to free a range of memory in the kernel. It works by iterating over the memory range one page at a time and calling kfree on each page to free it. This is typically used during the initialization of the kernel memory allocator to mark a range of memory as free and available for future allocations.

Practice Problems: Process Management in xv6

1. Consider the following lines of code in a program running on xv6.

```
int ret = fork();
if(ret==0) { //do something in child}
else { //do something in parent}
```

- When a new child process is created as part of handling fork, what does the kernel stack of the new child process contain, after fork finishes creating it, but just before the CPU switches away from the parent?
- How is the kernel stack of the newly created child process different from that of the parent?
- The EIP value that is present in the trap frames of the parent and child processes decides where both the processes resume execution in user mode. Do both the EIP pointers in the parent and child contain the same logical address? Do they point to the same physical address in memory (after address translation by page tables)? Explain.
- How would your answer to (c) above change if xv6 implemented copy-on-write during fork?
- When the child process is scheduled for the first time, where does it start execution in kernel mode? List the steps until it finally gets to executing the instruction after fork in the program above in user mode.

Ans:

- It contains a trap frame, followed by the context structure.
- The parent's kernel stack has only a trap frame, since it is still running and has not been context switched out. Further, the value of the EAX register in the two trap frames is different, to return different values in parent and child.
- The EIP value points to the same logical address, but to different physical addresses, as the parent and child have different memory images.
- With copy-on-write, the physical addresses may be the same as well, as long as both parent and child have not modified anything.
- Starts at forkret, followed by trapret. Pops the trapframe and starts executing at the instruction right after fork.

2. Suppose a machine (architecture: x86, single core) has two runnable processes P1 and P2. P1 executes a line of code to read 1KB of data from an open file on disk to a buffer in its memory.

The content requested is not available in the disk buffer cache and must be fetched from disk. Describe what happens from the time the instruction to read data is started in P1, to the time it completes (causing the process to move on to the next instruction in the program), by answering the following questions.

- (a) The code to read data from disk will result in a system call, and will cause the x86 CPU to execute the `int` instruction. Briefly describe what the CPU's `int` instruction does.
- (b) The `int` instruction will then call the kernel's code to handle the system call. Briefly describe the actions executed by the OS interrupt/trap/system call handling code before the read system call causes P1 to block.
- (c) Now, because process P1 has made a blocking system call, the CPU scheduler context switches to some other process, say P2. Now, the data from the disk that unblocks P1 is ready, and the disk controller raises an interrupt while P2 is running. On whose kernel stack does this interrupt processing run?
- (d) Describe the contents of the kernel stacks of P1 and P2 when this interrupt is being processed.
- (e) Describe the actions performed by P2 in kernel mode when servicing this disk interrupt.
- (f) Right after the disk interrupt handler has successfully serviced the interrupt above, and before any further calls to the scheduler to context switch from P2, what is the state of process P1?

Ans:

- (a) The CPU switches to kernel mode, switches to the kernel stack of the process, and pushes some registers like the EIP onto the kernel stack.
 - (b) The kernel pushes a few other registers, updates segment registers, and starts executing the system call code, which eventually causes P1 to block.
 - (c) P2's kernel stack
 - (d) P2's kernel stack has a trapframe (since it switched to kernel mode). P1's kernel stack has both a context structure and a trap frame (since it is currently context switched out).
 - (e) P2 saves user context, switches to kernel mode, services the disk interrupt that unblocks P1, marks P1 as ready, and resumes its execution in userspace.
 - (f) Ready / runnable.
3. Consider a newly created process in xv6. Below are the several points in the code that the new process passes through before it ends up executing the line just after the `fork` statement in its user space. The EIP of each of these code locations exists on the kernel stack when the process is scheduled for the first time. For each of these locations below, precisely explain where on the kernel stack the corresponding EIP is stored (in which data structure etc.), and when (as part of which function or stage of process creation) is the EIP written there. Be as specific as possible in your answers.

- (a) `forkret`
- (b) `trapret`
- (c) just after the `fork()` system call in userspace

Ans:

- (a) EIP of forkret is stored in struct context by allocproc.
 - (b) EIP of trapret is stored on kernel stack by allocproc.
 - (c) EIP of fork system call code is stored in trapframe in parent, and copied to child's kernel stack in the fork function.
4. Consider a process that has performed a blocking disk read, and has been context switched out in xv6. Now, when the disk interrupt occurs with the data requested by the process, the process is unblocked and context switched in immediately, as part of handling the interrupt. [T/F]

Ans: F

5. In xv6, state the system call(s) that result in new `struct proc` objects being allocated.

Ans: fork

6. Give an example of a scenario in which the xv6 dispatcher / `swtch` function does NOT use a `struct context` created by itself previously, but instead uses an artificially hand-crafted `struct context`, for the purpose of restoring context during a context switch.

Ans: When running process for first time (say, after fork).

7. Give an example of a scenario in xv6 where a `struct context` is stored on the kernel stack of a process, but this context is never used or restored at any point in the future.

Ans: When process has finished after exit, its saved context is never restored.

8. Consider a parent process P that has executed a fork system call to spawn a child process C. Suppose that P has just finished executing the system call code, but has not yet returned to user mode. Also assume that the scheduler is still executing P and has not context switched it out. Below are listed several pieces of state pertaining to a process in xv6. For each item below, answer if the state is identical in both processes P and C. Answer Yes (if identical) or No (if different) for each question.

- (a) Contents of the PCB (`struct proc`). That is, are the PCBs of P and C identical? (Yes/No)
- (b) Contents of the memory image (code, data, heap, user stack etc.).
- (c) Contents of the page table stored in the PCB.
- (d) Contents of the kernel stack.
- (e) EIP value in the trap frame.
- (f) EAX register value in the trap frame.
- (g) The physical memory address corresponding to the EIP in the trap frame.
- (h) The files pointed at by the file descriptor table. That is, are the file structures pointed at by any given file descriptor identical in both P and C?

Ans:

- (a) No
- (b) Yes
- (c) No

- (d) No
 - (e) Yes
 - (f) No
 - (g) No
 - (h) Yes
9. Suppose the kernel has just created the first user space “init” process, but has not yet scheduled it. Answer the following questions.
- (a) What does the EIP in the trap frame on the kernel stack of the process point to?
 - (b) What does the EIP in the context structure on the kernel stack (that is popped when the process is context switched in) point to?

Ans:

- (a) address 0 (first line of code in init user code)
 - (b) forkret / trapret
10. Consider a process P that forks a child process C in xv6. Compare the trap frames on the kernel stacks of P and C just after the fork system call completes execution, and before P returns back to user mode. State one difference between the two trap frames at this instant. Be specific in your answer and state the exact field/register value that is different.

Ans: EAX register has different value.

11. In xv6, the EIP within the `struct context` on the kernel stack of a process usually points to the `switch` statement in the `sched` function, where the process gives up its CPU and switches to the scheduler thread during a context switch. Which processes are an exception to this statement? That is, for which processes does the EIP on the context structure point to some other piece of code?

Ans: Newly created processes / processes running for first time

12. When a trap occurs in xv6, and a process shifts from user mode to kernel mode, which entity switches the CPU stack pointer from pointing to the user stack of the running program to its kernel stack? Tick one: x86 hardware instruction / xv6 assembly code

Ans: x86 hardware

13. Consider a process P in xv6, which makes a system call, goes to kernel mode, runs the system call code, and comes back into user mode again. The value of the EAX register is preserved across this transition. That is, the value of the EAX register just before the process started the system call will always be equal to its value just after the process has returned back to user mode. [T/F]

Ans: False, EAX is used to store system call number and return value, so it changes.

14. When a trap causes a process to shift from user mode to kernel mode in xv6, which CPU data registers are stored in the trapframe (on the kernel stack) of the process? Tick one: all registers / only callee-save registers

Ans: All registers

15. When a process in xv6 wishes to pass one or more arguments to the system call, where are these arguments initially stored, before the process initiates a jump into kernel mode? Tick one: user stack / kernel stack

Ans: User stack, as user program cannot access kernel stack

16. Consider the context switch of a CPU from the context of process P1 to that of process P2 in xv6. Consider the following two events in the chronological order of the events during the context switch: (E1) the ESP (stack pointer) shifts from pointing to the kernel stack of P1 to the kernel stack of P2; (E2) the EIP (program counter) shifts from pointing to an address in the memory allocated to P1 to an address in the memory allocated to P2. Which of the following statements is/are true regarding the relative ordering of events E1 and E2?

- (a) E1 occurs before E2.
- (b) E2 occurs before E1.
- (c) E1 and E2 occur simultaneously via an atomic hardware instruction.
- (d) The relative ordering of E1 and E2 can vary from one context switch to the other.

Ans: (a)

17. Consider the following actions that happen during a context switch from thread/process P1 to thread/process P2 in xv6. (One of P1 or P2 could be the scheduler thread as well.) Arrange the actions below in chronological order, from earliest to latest.

- (A) Switch ESP from kernel stack of P1 to that of P2
- (B) Pop the callee-save registers from the kernel stack of P2
- (C) Push the callee-save registers onto the kernel stack of P1
- (D) Push the EIP where execution of P1 stops onto the kernel stack of P1.

Ans: DCAB

18. Consider a process P in xv6 that invokes the wait system call. Which of the following statements is/are true?

- (a) If P does not have any zombie children, then the wait system call returns immediately.
- (b) The wait system call always blocks process P and leads to a context switch.
- (c) If P has exactly one child process, and that child has not yet terminated, then the wait system call will cause process P to block.
- (d) If P has two or more zombie children, then the wait system call reaps all the zombie children of P and returns immediately.

Ans: (c)

19. Consider a process P in xv6 that executes the exec system call successfully. Which of the following statements is/are true?

- (a) The exec system call changes the PID of process P.
- (b) The exec system call allocates a new page table for process P.

- (c) The exec system call allocates a new kernel stack for process P.
- (d) The exec system call changes one or more fields in the trap frame on the kernel stack of process P.

Ans: (b), (d)

20. Consider a process P in xv6 that executes the exec system call successfully. Which of the following statements is/are true?

- (a) The arguments to the exec system call are first placed on the user stack by the user code.
- (b) The arguments to the exec system call are first placed on the kernel stack by the user code.
- (c) The arguments (argc, argv) to the new executable are placed on the kernel stack by the exec system call code.
- (d) The arguments (argc, argv) to the new executable are placed on the user stack by the exec system call code.

Ans: (a), (d)

21. Consider a newly created child process C in xv6 that is scheduled for the first time. At the point when the scheduler is just about to context switch into C, which of the following statements is/are true about the kernel stack of process C?

- (a) The top of the kernel stack contains the context structure, whose EIP points to the instruction right after the fork system call in user code.
- (b) The bottom of the kernel stack has the trapframe, whose EIP points to the forkret function in OS code.
- (c) The top of the kernel stack contains the context structure, whose EIP points to the forkret function in OS code.
- (d) The bottom of the kernel stack contains the trap frame, whose EIP points to the trapret function in OS code.

Ans: (c)

22. Consider a trapframe stored on the kernel stack of a process P in xv6 that jumped from user mode to kernel mode due to a trap. Which of the following statements is/are true?

- (a) All fields of the trapframe are pushed onto the kernel stack by the OS code.
- (b) All fields of the trapframe are pushed onto the kernel stack by the x86 hardware.
- (c) The ESP value stored in the trapframe points to the top of the kernel stack of the process.
- (d) The ESP value stored in the trapframe points to the top of the user stack of the process.

Ans: (d)

23. Consider a process P that has made a blocking disk read in xv6. The OS has issued a disk read command to the disk hardware, and has context switched away from P. Which of the following statements is/are true?

- (a) The top of the kernel stack of P contains the return address, which is the value of EIP pointing to the user code after the read system call.
- (b) The bottom of the kernel stack of P contains the trapframe, whose EIP points to the user code after the read system call.
- (c) The top of the kernel stack of P contains the context structure, whose EIP points to the user code after the read system call.
- (d) The CPU scheduler does not run P again until after the disk interrupt that unblocks P is raised by the device hardware.

Ans: (b), (d)

24. In the implementation of which of the following system calls in xv6 are new ptable entries allocated or old ptable entries released (marked as unused)?

- (a) fork
- (b) exit
- (c) exec
- (d) wait

Ans: (a), (d)

25. A process has invoked exit() in xv6. The CPU has completed executing the OS code corresponding to the exit system call, and is just about to invoke the swtch() function to switch from the terminated process to the scheduler thread. Which of the following statements is/are true?

- (a) The stack pointer ESP is pointing to some location within the kernel stack of the terminated process
- (b) The MMU is using the page table of the terminated process
- (c) The state of the terminated process in the ptable is RUNNING
- (d) The state of the terminated process in the ptable is ZOMBIE

Ans: (a), (b), (d)

HOMEWORK QUESTIONS

1. Explain how paging works. In this explain the role of PTBR. Which parameters determine the size of page table in bytes, and number of entries in it? How is a 32-bit address split into parts, and how is each part used?
2. Explain the code of setupkvm(). Which logical addresses are mapped by setupkvm() and to which physical addresses?

-> 1. Kalloc //gets 4b mem and allocates
2. Memset //make full pg 0 so prev data is erased
3. Kvmalloc //Iterate over kmap array. iterate 4 times. Creates pgtable for each separately Since each has diff permissions.
4. Kmap
5. map pages // creates pages
6. Readi //reads and puts in mem
7. Error checks
8. Allocuvvm //creates user pddir, table
9. Loaduvvm //loads data in pages

3. How does the xv6 kernel obtain the parameters passed by user-application to the system call? E.g. how does kernel obtain the paramters to read(fd, &ch, 1); ?

Ans: Accesses them off of **user** stack (lol).

Detailed Ans: User programs push the args on the process' user stack. Execution of system call is done in the kernel mode hence the stack is changed to kernel stack. In order to access the arguments the user stack's esp is accessed through the trapframe. Rest you can read in the code.

4. Explain the meaning of each field in struct proc. Draw a nice diagram of struct proc with as many details as possible. What are the contents of the trap frame - draw a neat detailed diagram. What is the 'chan' field?

-> 1. size
2. Pgdir //points to page dir of process. Page comes from kalloc. Gives pointers to both user and kernel stack
3. Kstack //points to page obtained from kalloc. This is the kernel stack. Trap frame hosted on kernel stack
4. Proc * parent
5. Tf //trapframe pointer. Points to specific location on kernel stack
6. Context //pointer stores context [eip=forkret(),ebp,ebx,esi,edi]
7. Chan // used when process is in wait queue for some i/o operation. Basically an array which stores address of variables indicating what the proc is waiting for.
8. Int killed //=1 if process is killed
9. File * ofile //points to array of open file descriptors outside struct proc
10. Inode * cwd //pointer to inode of cwd of processes. Getcwd setcwd Manipulate inode
11. Name //array with name of processes

5. Explain the mappages and walkpgdir functions.

->

1. Map pages calls walkpgdir
2. Walkpgdir creates page table mapping (first address is kernbase i.e. 2GB)
3. After pg table allocated, exec pushes args on stack
4. Sets up args to main (argc, argv)
5. Saves context of prev process //refer line 97 of exec code in xv6
6. Calls switchuvvm() //switches cr3 here

6. What are the different usages of a semaphore? Give one example each.

Semaphores are a fundamental synchronization mechanism used in operating systems for coordinating access to shared resources among multiple processes or threads. They help prevent race conditions and ensure that critical sections of code are executed atomically.

1. Binary Semaphore: A binary semaphore is a semaphore with only two possible states, often used for mutual exclusion or as a signaling mechanism.

Example: In a producer-consumer problem, where multiple producers produce items and multiple consumers consume those items, a binary semaphore can be used to synchronize access to the shared buffer. Producers wait on the semaphore when the buffer is full, and consumers wait when the buffer is empty.

2. Counting Semaphore: A counting semaphore is a semaphore with an integer value that can range over an unrestricted domain.

Example: In a scenario where a limited number of resources (e.g., printers, database connections) are available, a counting semaphore can be used to control access to these resources. Each time a resource is acquired, the semaphore's count decreases, and when a resource is released, the count increases. Processes or threads can block if the count reaches zero, indicating that no resources are available.

3. Mutex Semaphore: A mutex (short for mutual exclusion) semaphore is a specialized binary semaphore used to enforce mutual exclusion on a critical section of code.

Example: In a multi-threaded environment, if multiple threads need to access a shared resource (e.g., a global variable), a mutex semaphore can be employed to ensure that only one thread can access the resource at a time. Threads attempting to acquire the mutex will block if it's already held by another thread, preventing concurrent access and potential data corruption.

4. Named Semaphore: Named semaphores are semaphores that have a unique name in the operating system's namespace, allowing processes to share synchronization primitives.

Example: In inter-process communication (IPC), named semaphores can be used to coordinate access to shared memory regions or file resources. Processes can create or open a named semaphore by its unique name and use it to control access to the shared resource, enabling synchronization across multiple processes.

7. Explain the code of exec(). Which are the major functions called by exec() and what do they do? What does allocproc() do?

->

1. Exec -> setupkvm -> kvmalloc -> map pages -> walkpgdir -> readi -> allocuvm -> loaduvm.
2. If page is not present at a given location then map pages calls alloc().
3. Allocuvm only allocates pgdir, table (not data)

4. Allocproc finds the first unused process in ptable and allocs it a pid and space for context and trapframe

8. Write one program that deadlocks, and one that livelocks.

9. Which data structures on disk are modified if you delete a file, say /a/b, on an ext2 file system? Try to enlist considering the worst possibilities.

1. Inode Table: The inode corresponding to the file /a/b will be modified to mark it as free. If the inode is part of an inode table block, that block might need to be modified to reflect the change in the inode's state.

2. Directory Entry: The directory entry for file b in directory /a will be modified to remove the reference to the inode of b. If the directory block containing this entry becomes empty after the deletion, it might be marked as free, and the corresponding data block might be modified to reflect this change.

3. Bitmaps: The block bitmap and inode bitmap might be updated to mark the blocks and inodes that were previously allocated to file b as free for reuse.

4. Data Blocks: If file b has any data blocks allocated to it, these blocks will be marked as free in the block bitmap, and the corresponding data blocks might be modified to reflect this change.

10. How will you write code to read a directory from an ext2 partition ? How will the entries in the directory be read by your code so that you can list all files/folders in a directory ? What is rec_len and how it is used in creating/deleting files/folders?

11. Compare concurrency and parallelism.

Concurrent and parallel programming are related but not synonymous concepts. Concurrent programming is more about the logical structure and behavior of programs, while parallel programming is more about the physical execution and optimization of programs. Concurrent programs can run on single-core or multicore processors, while parallel programs require multicore or distributed systems. Concurrent programs can be parallel, but not all concurrent programs are parallel. For example, a web server that handles multiple requests concurrently may not run them in parallel if it has only one processor. Similarly, parallel programs can be concurrent, but not all parallel programs are concurrent. For example, a matrix multiplication that runs on multiple cores may not be concurrent if it does not have any coordination or synchronization between them.

12. Which are the different symbols in the 'kernel' ELF file, used in the xv6 code ? (e.g. 'end')

In the xv6 kernel ELF file, there are several symbols used to mark different parts of the code. Some of the common symbols found in the kernel ELF file include:

1. **_start**: The entry point for the kernel code.
2. **_etext**: Marks the end of the text (code) section.
3. **_edata**: Marks the end of the data section.
4. **_end**: Marks the end of the kernel image.
5. **_binary_obj_name_start**, **_binary_obj_name_end**: Symbols used to denote the start and end of binary objects included in the kernel image, such as initcode, bootblock, etc.
6. **_rodata_start**, **_rodata_end**: Marks the start and end of the read-only data section.

These symbols help in understanding the layout and organization of the kernel image in memory. They are often used in linker scripts and other parts of the build process to properly place code and data sections in memory.

13. Which memory violations are detected in xv6 ? Which violations are not detected?

In xv6, a simple Unix-like operating system developed for educational purposes, memory violations are not extensively detected by default. However, some common memory violations that might be detected include:

1. **Null Pointer Dereference**: Accessing or dereferencing a null pointer. This can sometimes lead to a kernel panic, but xv6 does not always detect this explicitly.
2. **Out-of-Bounds Memory Access**: Attempting to access memory outside the bounds of an allocated region, such as accessing an array element beyond its size.
3. **Invalid Memory Access**: Trying to access memory that is not mapped or accessible.
4. **Double Free**: Freeing memory that has already been freed, leading to potential memory corruption.

However, xv6 does not provide comprehensive protection against all types of memory violations. For example, it may not detect:

1. ****Use-after-Free****: Accessing memory after it has been freed. This can lead to undefined behavior but may not always be explicitly detected by the system.
2. ****Buffer Overflows****: Writing more data into a buffer than its allocated size. xv6 does not typically have built-in protections against buffer overflows.
3. ****Memory Leaks****: Failing to deallocate memory after it's no longer needed. xv6 does not have built-in mechanisms to detect or handle memory leaks.

Overall, while xv6 may catch some basic memory violations, it is not designed to be a robust environment for detecting and handling all possible memory-related issues. It's primarily intended for educational purposes and to provide a basic understanding of operating system principles.

14. explain in 1 line these concepts: mutex, spinlock, peterson's solution, sleeplock, race, critical section, entry-section, exit-section.

15. How do sched() and scheduler() functions work? Trace the sequence of lines of code which run one after another, when process P2 gets scheduled after process P1 encounters timers interrupt.

16. Explain how the spinlock code in xv6 works.

17. Explain the meaning of as many possible fields as possible in superblock, group descriptor and inode of ext2.

-> In the ext2 filesystem, the superblock, group descriptor, and inode contain crucial information about the file system structure and organization:

1. ****Superblock****: The superblock is a data structure at the beginning of the filesystem that contains essential information about the filesystem, including:

- Filesystem type (ext2)
- Total number of inodes and blocks
- Block size and fragment size
- Blocks per group and inodes per group
- First non-reserved inode and inode size
- Mount time and last write time
- Filesystem state (clean or dirty)
- Error handling information

2. ****Group Descriptor****: The group descriptor is a data structure that exists for each block group in the filesystem. It contains information specific to each group, such as:

- Block bitmap location and size
- Inode bitmap location and size
- Inode table location and size
- Free block and inode counts
- Directory count
- Last mounted time
- Filesystem state
- Group checksum (in newer versions like ext3 and ext4)

3. ****Inode****: An inode is a data structure that represents a file or directory in the filesystem. Each file or directory has a unique inode, which contains metadata about the file, such as:

- File type and access permissions
- Owner and group
- File size and timestamps (creation, modification, and access)

- Pointers to data blocks (direct, indirect, and doubly indirect)
- Number of hard links
- File flags and attributes
- File system-specific attributes (e.g., extended attributes)
- Access control lists (ACLs) and capabilities

These structures are crucial for the functioning and organization of the ext2 filesystem, providing information about the layout of data on disk, allocation of blocks and inodes, and metadata associated with files and directories.

18. How can deadlocks be prevented?

-> Write code in such a way that it will invalidate one of the 4 conditions necessary for deadlock i.e. mutual exclusion, hold & wait, no preemption, cyclic wait (most used).

1. Locking hierarchy (used in kernels): keep an eye on all locks code is holding at any given point in time.
2. Lock ordering (used in xv6): priority given in ppt 16, slide 77.

19. Explain the code of swtch() by drawing diagrams.

LAB TASKS

1. What is a Zombie process? is there a difference between an orphan and a zombie process?

2. Does the Demo given by Abhijit always create a Zombie process?

3. True/False? Every dead process first become a Zombie.

-> True

4. How is a Zombie process "cleared"? (Observe first that it gets cleared)

5. Write a C code and commands to show use of dup() using /proc?

6. How is a Zombie process different from Zombies in movies ?

-> init adopts zombie processes and eventually they finish.

7. Write a small code and commands, to show use of pipe() using /proc.

8. List the names of application programs that are part of the xv6 git repo.

9. Which is the complete "qemu" command used to run xv6, by the Makefile?

10. Suppose you want to run xv6 by specifying one more extra disk. How will you do that ?

11. What happens when you run xv6 (using qemu) with 2G RAM? Anything special or just normal execution?

15. Can you list all commands used to compile the "_ls" program in xv6 project?

16. What are all those options provided to gcc, to compile an application like _ls? what is -static? what is -m32?

17. What will happen if you change the entry in "gdtdesc" to be base=0 and limit = 2 GB?

Ans: It works fine as kernel has absolute addressing from 0x7c00 onwards, hence 2Gb limit is enough

18. Is the line movw %ax, %es really necessary in bootasm.S ?

Ans: no, as es register is never used in xv6 (see the output of objdump on bootasm.S). Similarly mov ax,dx is also not needed as dx is by default 0

19. Which all parts of the code actually push something on the stack starting at 0x7c00 and what do they push ?

20. How many program headers are there in the kernel and what do they mean?

Ans: 3 program headers in the Kernel. Maybe : First one is CS, second is Data seg and last is stack seg

21. readseg((uchar*)elf, 4096, 0); will read the first 4k bytes from the xv6.img into memory. This should include the bootloader also. Isn't it? If yes, how is "elf" pointer correct?

Ans: No readseg, gives an offset to make the sector read from =1 whilst translating bytes to sectors . Refer bootmain.c - readseg function

22. Why V2P_WO is used in entry.S and not V2P ?

Ans: Dumb answer but: No typecasting is required in entry.s. Perhaps due to the fact that in assembly no explicit typecast to uint is required. In fact during pre-processor directives the 'uint' symbol is unrecognized by the assembler.

23. What will be the effect if we remove the 0th entry from entrypgdir Array ?

Ans: The computer would have crashed trying to execute the instruction after the one that enabled paging. Mostly due to the fact that you will not be setting the flags to indicate that this 0th entry is present, writable & 4Mb in size. Accessing the 0th entry will hence cause a page fault to occur. - Refer mmu.h and main.c

24. Exactly how many page frames are created by knit1?

Ans: According to last T1 attempt, value of the variable 'end' passed to kinit1 is: 801154a8=KERNBASE+1135784. P2V(4*1024*1024) is KERNBASE+(4*1024*1024). Hence Diff between vstart and vend is (4*1024*1024)-1135784=3058520. Hence total pages=3058520/PGSIZE = 746 pages. (unless i have to further divide by 8 as this may be in terms of bytes and I used bits...)

25. Draw a diagram of the data structure changes done by mappage(), walkpgdir(), allocuvm(), deallocuvm(), freeuvm(), setupkvm(). Write 3-4 line description of each of these functions.

26. Take an application, for example cat.c ; Examine it's object code file using objdump. Obtain all the addresses uses for code (text), data, bss, etc. Then draw the page table that will be setup for this application, showing exactly the indices in page-directory and page-table that will be used.

27.

List down all changes required to xv6 code, in order to add the system call chown().

Every change should be mentioned in terms of either of the following:

- (a) pseudo-code of new function to be added
- (b) prototype of any new function or new system call to be added
- (c) pseudo-code of changes to an existing function, describing lines to be removed, and lines to be added
- (d) **precise** declaration of new data structures to be added in C, or changes to the existing data structure
- (e) Name and a one-line description of new userland functionality to be added
- (f) Changes to Makefile
- (g) Any other change in a maximum of 20 words per change.

Ans: Added pseudo code of system call in sysfile.c

```
int
sys_chown(void)
{
    //takes two parameters path and owner
    char *path;
    int owner;
    // checking parameter was given or not and retuen to chown() code
    return chown(path, owner);
}
```

MORE Specific:

```
int sys_chown(void)
{
    char* path;
    int uid, gid;

    if(argstr(0, &path) < 0 || argint(1, &uid)<0 || argint(2, &gid)<0)
        return -1;
```

```
    struct proc* currProc=myproc();
    struct inode* ip=namei(path);
    if(currProc->uid!=ROOT)
        return -1;
```

```
    chown(uid, gid, ip);
    return 1;
}
```

```
void chown(int uid, int gid, struct inode* ip)
{
    ilock(ip);
    ip->uid=uid; —> need some changed in struct INode
    ip->gid=gid;
    iunlock(ip);
}
```

So add uid and gid fields in the struct inode too.

Add a prototype for the new system call to the file syscall.h

```
int chown(char *path, int uid, int gid);
```

add implementation in fs.c

Changes to old functions:

1. lupdate : set appropriate uid and gid of dip
2. Set uid of root user as 0
3. Every proc also must have a user id (uid)
4. In userinit by default set the UID as ROOT uid
5. In fork() make sure new proc uid= old parent proc uid

Changing system call number in syscall.h by adding the following line:

```
#define SYS_chown 22
```

Update system call jump table in syscall.c :

```
[SYS_chown] sys_chown,
```

```
\
```

Add a test case for the new system call to the file usertests.c. The test case should verify that chown() changes the owner of a file successfully.

Modify the Makefile to include the new source files created for the new system call:

```
_\chown
```

28.

Write all changes required to xv6 to add a buddy allocator.

Every change should be mentioned in terms of either of the following:

- (a) pseudo-code of new function to be added
- (b) prototype of any new function or new system call to be added
- (c) pseudo-code of changes to an existing function, describing lines to be removed, and lines to be added
- (d) **precise** declaration of new data structures to be added in C, or changes to the existing data structure
- (e) Name and a one-line description of new userland functionality to be added
- (f) Changes to Makefile
- (g) Any other change in a maximum of 20 words per change.

Critical Section Problem: The critical section problem arises in concurrent programming when multiple processes or threads access shared resources. A critical section is a part of the code where shared resources are accessed, and if multiple processes access it simultaneously, it can lead to data inconsistency or race conditions. To solve this problem, we need mechanisms to ensure that only one process can execute the critical section at a time. Common solutions include the use of locks or semaphores to enforce mutual exclusion. These mechanisms prevent multiple processes from simultaneously accessing the critical section, thereby ensuring data integrity.

Hardware Support for Mutual Exclusion: Some hardware architectures provide built-in support for mutual exclusion through special instructions or mechanisms. For example, processors may offer atomic operations, which are indivisible instructions that execute without interruption. Examples include Test-and-Set or Compare-and-Swap instructions, which can be used to implement locks efficiently. These hardware mechanisms are essential for building efficient synchronization primitives like locks and semaphores in concurrent programming.

Semaphores: Semaphores are a synchronization primitive introduced by Dijkstra. They are used to control access to a common resource by multiple processes or threads. A semaphore is essentially a non-negative integer counter. There are two main types of semaphores:

Binary Semaphore: Can only take on the values 0 and 1. It can be used for simple signaling or mutual exclusion.

Counting Semaphore: Can take on any non-negative integer value. It is often used for resource counting or synchronization among multiple processes.

Semaphores support two primary operations:

Wait (P) Operation: Decrements the semaphore value. If the value becomes negative, the process is blocked until it becomes positive.

Signal (V) Operation: Increments the semaphore value. If there are blocked processes waiting on the semaphore, one of them is unblocked.

Semaphores are a versatile synchronization primitive and can be used to implement various synchronization patterns, including mutual exclusion, producer-consumer, and reader-writer synchronization.

Deadlock Principle: Deadlock occurs when two or more processes are unable to proceed because each is waiting for the other to release a resource. Deadlock arises when four necessary conditions are met: mutual exclusion, hold and wait, no preemption, and circular wait.

Mutual Exclusion: Resources cannot be simultaneously shared; only one process can use a resource at a time.

Hold and Wait: Processes hold resources while waiting for others.

No Preemption: Resources cannot be forcibly taken from a process.

Circular Wait: A cycle of waiting exists, where each process waits for a resource held by another process in the cycle.

To prevent deadlock, one or more of these conditions must be eliminated.

Deadlock Detection, Prevention, and Avoidance:

Deadlock Detection: Involves periodically checking the system for deadlock. If detected, recovery strategies such as killing processes, preempting resources, or rolling back transactions can be employed.

Deadlock Prevention: Aims to prevent one of the necessary conditions for deadlock. Techniques include resource allocation policies, ensuring that at least one of the conditions is not met.

Deadlock Avoidance: Dynamically allocate resources in a way that avoids the possibility of deadlock. Techniques like Banker's algorithm or resource allocation graphs are used to ensure safe resource allocation.

Classical Problems in Concurrent Programming:

Producer-Consumer Problem: Involves one or more producer threads generating data and one or more consumer threads consuming the data. Synchronization is required to ensure that producers do not produce data too quickly for consumers to consume and vice versa.

Reader-Writer Problem: Involves multiple readers and writers accessing a shared resource. Variations include with and without a bounded buffer, where the buffer has a finite capacity. Synchronization is needed to ensure data consistency and avoid race conditions between readers and writers.

Design of Locking Primitives:

Spinlock: A lock that repeatedly checks for the availability of a resource in a tight loop until it becomes available. Spinlocks are suitable for short critical sections or when the wait time is expected to be short.

Semaphore: A synchronization primitive that controls access to a shared resource through a counter. Semaphores can be used to enforce mutual exclusion or synchronize multiple processes.

Read-Write Locks: Allows multiple readers to access a resource simultaneously but exclusive access for writers. They come in two main variants: reader-preference and writer-preference locks

HOMEWORK QUESTIONS

1. Explain how paging works. In this explain the role of PTBR. Which parameters determine the size of page table in bytes, and number of entries in it? How is a 32-bit address split into parts, and how is each part used?
2. Explain the code of setupkvm(). Which logical addresses are mapped by setupkvm() and to which physical addresses?

-> 1. Kalloc //gets 4b mem and allocates

2. Memset //make full pg 0 so prev data is erased
3. Kvmalloc //Iterate over kmap array. iterate 4 times. Creates pgtble for each separately
Since each has diff permissions.

4. Kmap
5. map pages // creates pages
6. Readi //reads and puts in mem
7. Error checks
8. Allocuvm //creates user pdmdir, table
9. Loaduvm //loads data in pages

3. How does the xv6 kernel obtain the parameters passed by user-application to the system call?
E.g. how does kernel obtain the paramters to read(fd, &ch, 1); ?

Ans: Accesses them off of stack (lol).

Detailed Ans: User programs push the args on the process' user stack. Execution of system call is done in the kernel mode hence the stack is changed to kernel stack. In order to access the arguments the user stack's esp is accessed through the trapframe. Rest you can read in the code.

4. Explain the meaning of each field in struct proc. Draw a nice diagram of struct proc with as many details as possible. What are the contents of the trap frame - draw a neat detailed diagram. What is the 'chan' field?

-> 1. size

2. Pgdir //points to page dir of process. Page comes from kalloc. Gives pointers to both user

and kernel stack

3. Kstack //points to page obtained from kalloc. This is the kernel stack. Trap frame hosted on kernel stack

4. Proc * parent

5. Tf //trapframe pointer. Points to specific location on kernel stack

6. Context //pointer stores context [eip=forkret(),ebp,ebx,esi,edi]

7. Chan // used when process is in wait queue for some i/o operation. Basically an array which stores address of variables indicating what the proc is waiting for.

8. Int killed //=1 if process is killed

9. File * ofile //points to array of open file descriptors outside struct proc

10. Inode * cwd //pointer to inode of cwd of processes. Getcwd setcwd Manipulate inode

11. Name //array with name of processes

5. Explain the mappages and walkpgdir functions.

->

1. Map pages calls walkpgdir

2. Walkpgdir creates page table mapping (first address is kernbase i.e. 2GB)

3. After pg table allocated, exec pushes args on stack

4. Sets up args to main (argc, argv)

5. Saves context of prev process //refer line 97 of exec code in xv6

6. Calls switchuvvm() //switches cr3 here

6. What are the different usages of a semaphore? Give one example each.

Semaphores are a fundamental synchronization mechanism used in operating systems for coordinating access to shared resources among multiple processes or threads. They help prevent race conditions and ensure that critical sections of code are executed atomically

1. Binary Semaphore: A binary semaphore is a semaphore with only two possible states, often used for mutual exclusion or as a signaling mechanism.

Example: In a producer-consumer problem, where multiple producers produce items and multiple consumers consume those items, a binary semaphore can be used to synchronize access to the shared buffer. Producers wait on the semaphore when the buffer is full, and consumers wait when the buffer is empty.

2. Counting Semaphore: A counting semaphore is a semaphore with an integer value that can range over an unrestricted domain.

Example: In a scenario where a limited number of resources (e.g., printers, database connections) are available, a counting semaphore can be used to control access to these resources. Each time a resource is acquired, the semaphore's count decreases, and when a resource is released, the count increases. Processes or threads can block if the count reaches zero, indicating that no resources are available.

3. Mutex Semaphore: A mutex (short for mutual exclusion) semaphore is a specialized binary semaphore used to enforce mutual exclusion on a critical section of code.

Example: In a multi-threaded environment, if multiple threads need to access a shared resource (e.g., a global variable), a mutex semaphore can be employed to ensure that only one thread can access the resource at a time. Threads attempting to acquire the mutex will block if it's already held by another thread, preventing concurrent access and potential data corruption.

4. Named Semaphore: Named semaphores are semaphores that have a unique name in the operating system's namespace, allowing processes to share synchronization primitives.

Example: In inter-process communication (IPC), named semaphores can be used to coordinate access to shared memory regions or file resources. Processes can create or open a named semaphore by its unique name and use it to control access to the shared resource, enabling synchronization across multiple processes

7. Explain the code of exec(). Which are the major functions called by exec() and what do they do? What does allocproc() do?

->

1. Exec -> setupkvm -> kvmalloc -> map pages -> walkpgdir -> readi -> allocuvm -> loaduvm.
2. If page is not present at a given location then map pages calls alloc().
3. Allocuvm only allocates pgdir, table (not data)
4. Allocproc finds the first unused process in ptable and allocs it a pid and space for context and trapframe

8. Write one program that deadlocks, and one that livelocks.

9. Which data structures on disk are modified if you delete a file, say /a/b, on an ext2 file system? Try to enlist considering the worst possibilities.

1. Inode Table: The inode corresponding to the file /a/b will be modified to mark it as free. If the inode is part of an inode table block, that block might need to be modified to reflect the change in the inode's state.

2. Directory Entry: The directory entry for file b in directory /a will be modified to remove the reference to the inode of b. If the directory block containing this entry becomes empty after the deletion, it might be marked as free, and the corresponding data block might be modified to reflect this change.

3. Bitmaps: The block bitmap and inode bitmap might be updated to mark the blocks and inodes that were previously allocated to file b as free for reuse.

4. Data Blocks: If file b has any data blocks allocated to it, these blocks will be marked as free in the block bitmap, and the corresponding data blocks might be modified to reflect this change.

10. How will you write code to read a directory from an ext2 partition ? How will the entries in the directory be read by your code so that you can list all files/folders in a directory ? What is rec_len and how it is used in creating/deleting files/folders?

11. Compare concurrency and parallelism.

Concurrent and parallel programming are related but not synonymous concepts. Concurrent programming is more about the logical structure and behavior of programs, while parallel programming is more about the physical execution and optimization of programs. Concurrent programs can run on single-core or multicore processors, while parallel programs require multicore or distributed systems. Concurrent programs can be parallel, but not all concurrent programs are parallel. For example, a web server that handles multiple requests concurrently may not run them in parallel if it has only one processor. Similarly, parallel programs can be concurrent, but not all parallel programs are concurrent. For example, a matrix multiplication that runs on multiple cores may not be concurrent if it does not have any coordination or synchronization between them.

12. Which are the different symbols in the 'kernel' ELF file, used in the xv6 code ? (e.g. 'end')

In the xv6 kernel ELF file, there are several symbols used to mark different parts of the code. Some of the common symbols found in the kernel ELF file include:

1. **__start**: The entry point for the kernel code.
2. **__etext**: Marks the end of the text (code) section.
3. **__edata**: Marks the end of the data section.
4. **__end**: Marks the end of the kernel image.
5. **__binary_obj_name_start**, **__binary_obj_name_end**: Symbols used to denote the start and end of binary objects included in the kernel image, such as initcode, bootblock, etc.
6. **__rodata_start**, **__rodata_end**: Marks the start and end of the read-only data section.

These symbols help in understanding the layout and organization of the kernel image in memory. They are often used in linker scripts and other parts of the build process to properly place code and data sections in memory.

13. Which memory violations are detected in xv6 ? Which violations are not detected?

In xv6, a simple Unix-like operating system developed for educational purposes, memory violations are not extensively detected by default. However, some common memory violations that might be detected include:

1. **Null Pointer Dereference**: Accessing or dereferencing a null pointer. This can sometimes lead to a kernel panic, but xv6 does not always detect this explicitly.
2. **Out-of-Bounds Memory Access**: Attempting to access memory outside the bounds of an allocated region, such as accessing an array element beyond its size.
3. **Invalid Memory Access**: Trying to access memory that is not mapped or accessible.
4. **Double Free**: Freeing memory that has already been freed, leading to potential memory corruption.

However, xv6 does not provide comprehensive protection against all types of memory violations. For example, it may not detect:

1. **Use-after-Free**: Accessing memory after it has been freed. This can lead to undefined behavior but may not always be explicitly detected by the system.

2. **Buffer Overflows**: Writing more data into a buffer than its allocated size. xv6 does not typically have built-in protections against buffer overflows.

3. **Memory Leaks**: Failing to deallocate memory after it's no longer needed. xv6 does not have built-in mechanisms to detect or handle memory leaks.

Overall, while xv6 may catch some basic memory violations, it is not designed to be a robust environment for detecting and handling all possible memory-related issues. It's primarily intended for educational purposes and to provide a basic understanding of operating system principles.

14. explain in 1 line these concepts: mutex, spinlock, peterson's solution, sleeplock, race, critical section, entry-section, exit-section.

15. How do sched() and scheduler() functions work? Trace the sequence of lines of code which run one after another, when process P2 gets scheduled after process P1 encounters timers interrupt.

16. Explain how the spinlock code in xv6 works.

17. Explain the meaning of as many possible fields as possible in superblock, group descriptor and inode of ext2.

-> In the ext2 filesystem, the superblock, group descriptor, and inode contain crucial information about the file system structure and organization:

1. **Superblock**: The superblock is a data structure at the beginning of the filesystem that contains essential information about the filesystem, including:

- Filesystem type (ext2)
- Total number of inodes and blocks
- Block size and fragment size
- Blocks per group and inodes per group
- First non-reserved inode and inode size
- Mount time and last write time
- Filesystem state (clean or dirty)
- Error handling information

2. **Group Descriptor**: The group descriptor is a data structure that exists for each block group in the filesystem. It contains information specific to each group, such as:

- Block bitmap location and size
- Inode bitmap location and size
- Inode table location and size
- Free block and inode counts
- Directory count
- Last mounted time
- Filesystem state
- Group checksum (in newer versions like ext3 and ext4)

3. **Inode**: An inode is a data structure that represents a file or directory in the filesystem. Each file or directory has a unique inode, which contains metadata about the file, such as:

- File type and access permissions
- Owner and group
- File size and timestamps (creation, modification, and access)
- Pointers to data blocks (direct, indirect, and doubly indirect)
- Number of hard links
- File flags and attributes
- File system-specific attributes (e.g., extended attributes)
- Access control lists (ACLs) and capabilities

These structures are crucial for the functioning and organization of the ext2 filesystem, providing information about the layout of data on disk, allocation of blocks and inodes, and metadata associated with files and directories.

18. How can deadlocks be prevented?

-> Write code in such a way that it will invalidate one of the 4 conditions necessary for deadlock i.e. mutual exclusion, hold & wait, no preemption, cyclic wait (most used).

1. Locking hierarchy (used in kernels): keep an eye on all locks code is holding at any given point in time.
2. Lock ordering (used in xv6): priority given in ppt 16, slide 77.

19. Explain the code of swtch() by drawing diagrams.

LAB TASKS

1. What is a Zombie process? is there a difference between an orphan and a zombie process?

2. Does the Demo given by Abhijit always create a Zombie process?

3. True/False? Every dead process first become a Zombie.

-> True

4. How is a Zombie process "cleared"? (Observe first that it gets cleared)

5. Write a C code and commands to show use of dup() using /proc?

6. How is a Zombie process different from Zombies in movies ?

-> init adopts zombie processes and eventually they finish.

7. Write a small code and commands, to show use of pipe() using /proc.

8. List the names of application programs that are part of the xv6 git repo.

9. Which is the complete "qemu" command used to run xv6, by the Makefile?

10. Suppose you want to run xv6 by specifying one more extra disk. How will you do that ?

11. What happens when you run xv6 (using qemu) with 2G RAM? Anything special or just normal execution?

15. Can you list all commands used to compile the "_ls" program in xv6 project?

16. What are all those options provided to gcc, to compile an application like _ls? what is -static? what is -m32?

17. What will happen if you change the entry in "gdtdesc" to be base=0 and limit = 2 GB?

Ans: It works fine as kernel has absolute addressing from 0x7c00 onwards, hence 2Gb limit is enough

18. Is the line movw %ax, %es really necessary in bootasm.S ?

Ans: no, as es register is never used in xv6 (see the output of objdump on bootasm.S). Similarly mov ax,dx is also not needed as dx is by default 0

19. Which all parts of the code actually push something on the stack starting at 0x7c00 and what do they push ?

20. How many program headers are there in the kernel and what do they mean?

Ans: 3 program headers in the Kernel. Maybe : First one is CS, second is Data seg and last is stack seg

21. readseg((uchar*)elf, 4096, 0); will read the first 4k bytes from the xv6.img into memory. This should include the bootloader also. Isn't it? If yes, how is "elf" pointer correct?

Ans: No readseg, gives an offset to make the sector read from =1 whilst translating bytes to sectors . Refer bootmain.c → readseg function

22. Why V2P_WO is used in entry.S and not V2P ?

Ans: Dumb answer but: No typecasting is required in entry.s. Perhaps due to the fact that in assembly no explicit typecast to uint is required. In fact during pre-processor directives the 'uint' symbol is unrecognized by the assembler.

23. What will be the effect if we remove the 0th entry from entrypgdir Array ?

Ans: The computer would have crashed trying to execute the instruction after the one that enabled paging. Mostly due to the fact that you will not be setting the flags to indicate that this 0th entry is

present, writable & 4Mb in size. Accessing the 0th entry will hence cause a page fault to occur. → Refer mmu.h and main.c

24. Exactly how many page frames are created by knit1?

Ans: According to last T1 attempt, value of the variable 'end' passed to kinit1 is: 801154a8= KERNBASE+1135784. P2V(4*1024*1024) is KERNBASE+(4*1024*1024). Hence Diff between vstart and vend is (4*1024*1024)-1135784=3058520. Hence total pages=3058520/PGSIZE = 746 pages. (unless i have to further divide by 8 as this may be in terms of bytes and I used bits...)

25. Draw a diagram of the data structure changes done by mappage(), walkpgdir(), allocuvm(), deallocuvm(), freeuvm(), setupkvm(). Write 3-4 line description of each of these functions.

26. Take an application, for example cat.c ; Examine it's object code file using objdump. Obtain all the addresses uses for code (text), data, bss, etc. Then draw the page table that will be setup for this application, showing exactly the indices in page-directory and page-table that will be used.

27.

List down all changes required to xv6 code, in order to add the system call chown().

Every change should be mentioned in terms of either of the following:

- (a) pseudo-code of new function to be added
- (b) prototype of any new function or new system call to be added
- (c) pseudo-code of changes to an existing function, describing lines to be removed, and lines to be added
- (d) **precise** declaration of new data structures to be added in C, or changes to the existing data structure
- (e) Name and a one-line description of new userland functionality to be added
- (f) Changes to Makefile
- (g) Any other change in a maximum of 20 words per change.

Ans: Added pseudo code of system call in sysfile.c

```
int
sys_chown(void)
{
    //takes two parameters path and owner
    char *path;
    int owner;
    // checking parameter was given or not and retuen to chown() code
    return chown(path, owner);
```

}

Add a prototype for the new system call to the file syscall.h

```
int chown(char *path, int owner);
```

add implementation in fs.c

Changing system call number in syscall.h by adding the following line:

```
#define SYS_chown 22
```

Update system call jump table in syscall.c :

```
[SYS_chown] sys_chown,  
\
```

Add a test case for the new system call to the file usertests.c. The test case should verify that chown() changes the owner of a file successfully.

Modify the Makefile to include the new source files created for the new system call:

```
_\chown
```

28.

Write all changes required to xv6 to add a buddy allocator.

Every change should be mentioned in terms of either of the following:

- (a) pseudo-code of new function to be added
- (b) prototype of any new function or new system call to be added
- (c) pseudo-code of changes to an existing function, describing lines to be removed, and lines to be added
- (d) **precise** declaration of new data structures to be added in C, or changes to the existing data structure
- (e) Name and a one-line description of new userland functionality to be added
- (f) Changes to Makefile
- (g) Any other change in a maximum of 20 words per change.

Started on Saturday, 20 February 2021, 2:51 PM

State Finished

Completed on Saturday, 20 February 2021, 3:55 PM

Time taken 1 hour 3 mins

Grade 7.30 out of 20.00 (37%)

Question 1

Partially correct

Mark 0.80 out of 1.00

Select all the correct statements about the state of a process.

- a. A process can self-terminate only when it's running ✓
- b. Typically, it's represented as a number in the PCB ✓
- c. A process that is running is not on the ready queue ✓
- d. Processes in the ready queue are in the ready state ✓
- e. It is not maintained in the data structures by kernel, it is only for conceptual understanding of programmers
- f. Changing from running state to waiting state results in "giving up the CPU" ✓
- g. A process in ready state is ready to receive interrupts
- h. A waiting process starts running after the wait is over ✗
- i. A process changes from running to ready state on a timer interrupt ✓
- j. A process in ready state is ready to be scheduled ✓
- k. A running process may terminate, or go to wait or become ready again ✓
- l. A process waiting for I/O completion is typically woken up by the particular interrupt handler code ✓
- m. A process waiting for any condition is woken up by another process only
- n. A process changes from running to ready state on a timer interrupt or any I/O wait

Your answer is partially correct.

You have selected too many options.

The correct answers are: Typically, it's represented as a number in the PCB, A process in ready state is ready to be scheduled, Processes in the ready queue are in the ready state, A process that is running is not on the ready queue, A running process may terminate, or go to wait or become ready again, A process changes from running to ready state on a timer interrupt, Changing from running state to waiting state results in "giving up the CPU", A process can self-terminate only when it's running, A process waiting for I/O completion is typically woken up by the particular interrupt handler code

Question 2

Incorrect

Mark 0.00 out of 1.00

For each line of code mentioned on the left side, select the location of sp/esp that is in use

`jmp *%eax`
in entry.S

0x7c00 to 0x10000



`ljmp $(SEG_KCODE<<3), $start32`
in bootasm.S

0x10000 to 0x7c00



`call bootmain`
in bootasm.S

0x7c00 to 0x10000



`cli`
in bootasm.S

0x7c00 to 0



`readseg((uchar*)elf, 4096, 0);`
in bootmain.c

The 4KB area in kernel image, loaded in memory, named as 'stack'



Your answer is incorrect.

The correct answer is: `jmp *%eax`

`in entry.S` → The 4KB area in kernel image, loaded in memory, named as 'stack', `ljmp $(SEG_KCODE<<3), $start32`

`in bootasm.S` → Immaterail as the stack is not used here, `call bootmain`

`in bootasm.S` → 0x7c00 to 0, `cli`

`in bootasm.S` → Immaterail as the stack is not used here, `readseg((uchar*)elf, 4096, 0);`

`in bootmain.c` → 0x7c00 to 0

Question 3

Correct

Mark 0.25 out of 0.25

Order the following events in boot process (from 1 onwards)

Boot loader	2	✓
Shell	6	✓
BIOS	1	✓
OS	3	✓
Init	4	✓
Login interface	5	✓

Your answer is correct.

The correct answer is: Boot loader → 2, Shell → 6, BIOS → 1, OS → 3, Init → 4, Login interface → 5

Question 4

Partially correct

Mark 0.30 out of 0.50

Consider the following command and its output:

```
$ ls -lht xv6.img kernel
-rw-rw-r-- 1 abhijit abhijit 4.9M Feb 15 11:09 xv6.img
-rwxrwxr-x 1 abhijit abhijit 209K Feb 15 11:09 kernel*
```

Following code in bootmain()

```
readseg((uchar*)elf, 4096, 0);
```

and following selected lines from Makefile

```
xv6.img: bootblock kernel
dd if=/dev/zero of=xv6.img count=10000
dd if=bootblock of=xv6.img conv=notrunc
dd if=kernel of=xv6.img seek=1 conv=notrunc
```

```
kernel: $(OBJS) entry.o entryother initcode kernel.ld
$(LD) $(LDFLAGS) -T kernel.ld -o kernel entry.o $(OBJS) -b binary initcode entryother
$(OBJDUMP) -S kernel > kernel.asm
$(OBJDUMP) -t kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' > kernel.sym
```

Also read the code of bootmain() in xv6 kernel.

Select the options that describe the meaning of these lines and their correlation.

- a. Although the size of the kernel file is 209 Kb, only 4Kb out of it is the actual kernel code and remaining part is all zeroes.
- b. The kernel is compiled by linking multiple .o files created from .c files; and the entry.o, initcode, entryother files ✓
- c. The kernel.ld file contains instructions to the linker to link the kernel properly ✓
- d. The bootmain() code does not read the kernel completely in memory
- e. readseg() reads first 4k bytes of kernel in memory
- f. Although the size of the xv6.img file is ~5MB, only some part out of it is the bootloader+kernel code and remaining part is all zeroes.
- g. The kernel.asm file is the final kernel file
- h. The kernel disk image is ~5MB, the kernel within it is 209 kb, but bootmain() initially reads only first 4kb, and the later part is not read as it is user programs.
- i. The kernel disk image is ~5MB, the kernel within it is 209 kb, but bootmain() initially reads only first 4kb, and the later part is read using program headers in bootmain(). ✓

Your answer is partially correct.

You have correctly selected 3.

The correct answers are: The kernel disk image is ~5MB, the kernel within it is 209 kb, but bootmain() initially reads only first 4kb, and the later part is read using program headers in bootmain(), readseg() reads first 4k bytes of kernel in memory, The kernel is compiled by linking multiple .o files created from .c files; and the entry.o, initcode, entryother files, The kernel.ld file contains instructions to the linker to link the kernel properly, Although the size of the xv6.img file is ~5MB, only some part out of it is the bootloader+kernel code and remaining part is all zeroes.

Question 5

Partially correct

Mark 0.50 out of 1.00

```
int f() {  
    int count;  
    for (count = 0; count < 2; count++) {  
        if (fork() == 0)  
            printf("Operating-System\\n");  
    }  
    printf("TYCOMP\\n");  
}
```

The number of times "Operating-System" is printed, is:

Answer:

The correct answer is: 7.00

Question 6

Partially correct

Mark 0.40 out of 0.50

Select Yes/True if the mentioned element must be a part of PCB

Select No/False otherwise.

Yes	No	
<input checked="" type="radio"/>	<input type="radio"/> X	PID
<input checked="" type="radio"/>	<input type="radio"/> X	Process context
<input checked="" type="radio"/>	<input type="radio"/> X	List of opened files
<input checked="" type="radio"/>	<input type="radio"/> X	Process state
<input type="radio"/> X	<input checked="" type="radio"/>	Parent's PID
<input type="radio"/> X	<input checked="" type="radio"/>	Pointer to IDT
<input type="radio"/> X	<input checked="" type="radio"/>	Function pointers to all system calls
<input checked="" type="radio"/>	<input type="radio"/> X	Memory management information about that process
<input checked="" type="radio"/>	<input checked="" type="radio"/>	Pointer to the parent process
<input checked="" type="radio"/>	<input type="radio"/> X	EIP at the time of context switch

PID: Yes

Process context: Yes

List of opened files: Yes

Process state: Yes

Parent's PID: No

Pointer to IDT: No

Function pointers to all system calls: No

Memory management information about that process: Yes

Pointer to the parent process: Yes

EIP at the time of context switch: Yes

Question 7

Incorrect

Mark 0.00 out of 1.00

Select all the correct statements about code of bootmain() in xv6

```

void
bootmain(void)
{
    struct elfhdr *elf;
    struct proghdr *ph, *eph;
    void (*entry)(void);
    uchar* pa;

    elf = (struct elfhdr*)0x10000; // scratch space

    // Read 1st page off disk
    readseg((uchar*)elf, 4096, 0);

    // Is this an ELF executable?
    if(elf->magic != ELF_MAGIC)
        return; // let bootasm.S handle error

    // Load each program segment (ignores ph flags).
    ph = (struct proghdr*)((uchar*)elf + elf->phoff);
    eph = ph + elf->phnum;
    for(; ph < eph; ph++){
        pa = (uchar*)ph->paddr;
        readseg(pa, ph->filesz, ph->off);
        if(ph->memsz > ph->filesz)
            stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
    }

    // Call the entry point from the ELF header.
    // Does not return!
    entry = (void(*)(void))(elf->entry);
    entry();
}

```

Also, inspect the relevant parts of the xv6 code. binary files, etc and run commands as you deem fit to answer this question.

- a. The kernel file gets loaded at the Physical address 0x10000 +0x80000000 in memory. ✗
- b. The elf->entry is set by the linker in the kernel file and it's 0x80000000 ✗
- c. The kernel ELF file contains actual physical address where particular sections of 'kernel' file should be loaded ✓
- d. The kernel file in memory is not necessarily a continuously filled in chunk, it may have holes in it. ✓
- e. The kernel file has only two program headers ✓
- f. The elf->entry is set by the linker in the kernel file and it's 0x80000000 ✗
- g. The readseg finally invokes the disk I/O code using assembly instructions ✓
- h. The elf->entry is set by the linker in the kernel file and it's 8010000c ✓
- i. The kernel file gets loaded at the Physical address 0x10000 in memory. ✓
- j. The condition if(ph->memsz > ph->filesz) is never true. ✗
- k. The stosb() is used here, to fill in some space in memory with zeroes ✓

Your answer is incorrect.

The correct answers are: The kernel file gets loaded at the Physical address 0x10000 in memory., The kernel file in memory is not necessarily a continuously filled in chunk, it may have holes in it., The elf->entry is set by the linker in the kernel file and it's 8010000c, The readseg finally invokes the disk I/O code using assembly instructions, The stosb() is used here, to fill in some space in memory with zeroes, The kernel ELF file contains actual physical address where particular sections of 'kernel' file should be loaded, The kernel file has only two program headers

Question 8

Partially correct

Mark 0.13 out of 0.25

Which of the following are NOT a part of job of a typical compiler?

- a. Check the program for logical errors ✓
- b. Convert high level language code to machine code
- c. Process the # directives in a C program
- d. Invoke the linker to link the function calls with their code, extern globals with their declaration
- e. Check the program for syntactical errors
- f. Suggest alternative pieces of code that can be written

Your answer is partially correct.

You have correctly selected 1.

The correct answers are: Check the program for logical errors, Suggest alternative pieces of code that can be written

Question 9

Correct

Mark 0.25 out of 0.25

Rank the following storage systems from slowest (first) to fastest(last)

Cache	6	✓
Hard Disk	3	✓
RAM	5	✓
Optical Disks	2	✓
Non volatile memory	4	✓
Registers	7	✓
Magnetic Tapes	1	✓

Your answer is correct.

The correct answer is: Cache → 6, Hard Disk → 3, RAM → 5, Optical Disks → 2, Non volatile memory → 4, Registers → 7, Magnetic Tapes → 1

Question 10

Partially correct

Mark 0.21 out of 0.50

Which of the following parts of a C program do not have any corresponding machine code ?

- a. local variable declaration
- b. global variables
- c. function calls ✗
- d. #directives ✓
- e. expressions
- f. pointer dereference
- g. typedefs ✓

Your answer is partially correct.

You have correctly selected 2.

The correct answers are: #directives, typedefs, global variables

Question 11

Correct

Mark 0.25 out of 0.25

Match a system call with it's description

pipe	create an unnamed FIFO storage with 2 ends - one for reading and another for writing	✓
dup	create a copy of the specified file descriptor into smallest available file descriptor	✓
dup2	create a copy of the specified file descriptor into another specified file descriptor	✓
exec	execute a binary file overlaying the image of current process	✓
fork	create an identical child process	✓

Your answer is correct.

The correct answer is: pipe → create an unnamed FIFO storage with 2 ends - one for reading and another for writing, dup → create a copy of the specified file descriptor into smallest available file descriptor, dup2 → create a copy of the specified file descriptor into another specified file descriptor, exec → execute a binary file overlaying the image of current process, fork → create an identical child process

Question 12

Correct

Mark 0.25 out of 0.25

Match the register with the segment used with it.

eip	cs	✓
edi	es	✓
esi	ds	✓
ebp	ss	✓
esp	ss	✓

Your answer is correct.

The correct answer is: eip → cs, edi → es, esi → ds, ebp → ss, esp → ss

Question 13

Correct

Mark 0.25 out of 0.25

What's the trapframe in xv6?

- a. A frame of memory that contains all the trap handler code
- b. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware only
- c. The IDT table
- d. A frame of memory that contains all the trap handler code's function pointers
- e. A frame of memory that contains all the trap handler's addresses
- f. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware + code in trapasm.S ✓
- g. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by code in trapasm.S only

Your answer is correct.

The correct answer is: The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware + code in trapasm.S

Question 14

Incorrect

Mark 0.00 out of 0.50

Select all the correct statements about linking and loading.

Select one or more:

- a. Continuous memory management schemes can support dynamic linking and dynamic loading. ✗
- b. Loader is last stage of the linker program ✗
- c. Continuous memory management schemes can support static linking and dynamic loading. (may be inefficiently) ✓
- d. Dynamic linking and loading is not possible without demand paging or demand segmentation. ✓
- e. Dynamic linking essentially results in relocatable code. ✓
- f. Continuous memory management schemes can support static linking and static loading. (may be inefficiently) ✓
- g. Loader is part of the operating system ✓
- h. Static linking leads to non-relocatable code ✗
- i. Dynamic linking is possible with continuous memory management, but variable sized partitions only. ✗

Your answer is incorrect.

The correct answers are: Continuous memory management schemes can support static linking and static loading. (may be inefficiently), Continuous memory management schemes can support static linking and dynamic loading. (may be inefficiently), Dynamic linking essentially results in relocatable code., Loader is part of the operating system, Dynamic linking and loading is not possible without demand paging or demand segmentation.

Question 15

Incorrect

Mark 0.00 out of 0.25

In bootasm.S, on the line

```
ljmp    $(SEG_KCODE<<3), $start32
```

The SEG_KCODE << 3, that is shifting of 1 by 3 bits is done because

- a. The value 8 is stored in code segment
- b. The code segment is 16 bit and only upper 13 bits are used for segment number
- c. The code segment is 16 bit and only lower 13 bits are used for segment number ✗
- d. While indexing the GDT using CS, the value in CS is always divided by 8
- e. The ljmp instruction does a divide by 8 on the first argument

Your answer is incorrect.

The correct answer is: The code segment is 16 bit and only upper 13 bits are used for segment number

Question 16

Partially correct

Mark 0.07 out of 0.50

Order the events that occur on a timer interrupt:

Change to kernel stack

1	✗
---	---

Jump to a code pointed by IDT

2	✗
---	---

Jump to scheduler code

5	✗
---	---

Set the context of the new process

4	✗
---	---

Save the context of the currently running process

3	✓
---	---

Execute the code of the new process

6	✗
---	---

Select another process for execution

7	✗
---	---

Your answer is partially correct.

You have correctly selected 1.

The correct answer is: Change to kernel stack → 2, Jump to a code pointed by IDT → 1, Jump to scheduler code → 4, Set the context of the new process → 6, Save the context of the currently running process → 3, Execute the code of the new process → 7, Select another process for execution → 5

Question 17

Incorrect

Mark 0.00 out of 1.00

Consider the two programs given below to implement the command (ignore the fact that error checks are not done on return values of functions)

```
$ ls . /tmp/asdfksdf >/tmp/ddd 2>&1
```

Program 1

```
int main(int argc, char *argv[]) {
    int fd, n, i;
    char buf[128];

    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
    close(1);
    dup(fd);
    close(2);
    dup(fd);
    execl("/bin/ls", "/bin/ls", ".", "/tmp/asldjfaldfs", NULL);
}
```

Program 2

```
int main(int argc, char *argv[]) {
    int fd, n, i;
    char buf[128];

    close(1);
    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
    close(2);
    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
    execl("/bin/ls", "/bin/ls", ".", "/tmp/asldjfaldfs", NULL);
}
```

Select all the correct statements about the programs

Select one or more:

- a. Both programs are correct ✗
- b. Program 2 makes sure that there is one file offset used for '2' and '1' ✗
- c. Only Program 2 is correct ✗
- d. Program 2 does 1>&2 ✗
- e. Program 2 ensures 2>&1 and does not ensure >/tmp/ddd ✗
- f. Program 1 makes sure that there is one file offset used for '2' and '1' ✓
- g. Program 1 is correct for >/tmp/ddd but not for 2>&1 ✗
- h. Program 1 does 1>&2 ✗
- i. Both program 1 and 2 are incorrect ✗
- j. Program 2 is correct for >/tmp/ddd but not for 2>&1 ✗
- k. Only Program 1 is correct ✓
- l. Program 1 ensures 2>&1 and does not ensure >/tmp/ddd ✗

Your answer is incorrect.

The correct answers are: Only Program 1 is correct, Program 1 makes sure that there is one file offset used for '2' and '1'

Question 18

Correct

Mark 0.25 out of 0.25

Select the option which best describes what the CPU does during its powered ON lifetime

- a. Ask the user what is to be done, and execute that task
- b. Ask the OS what is to be done, and execute that task
- c. Fetch instructions specified by location given by PC, Decode and Execute it, during execution increment PC or change PC as per the instruction itself, Ask the User or the OS what is to be done next, repeat
- d. Fetch instructions specified by location given by PC, Decode and Execute it, during execution increment PC or change PC as per ✓ the instruction itself, repeat
- e. Fetch instruction specified by OS, Decode and execute it, repeat
- f. Fetch instructions specified by location given by PC, Decode and Execute it, during execution increment PC or change PC as per the instruction itself, Ask OS what is to be done next, repeat

The correct answer is: Fetch instructions specified by location given by PC, Decode and Execute it, during execution increment PC or change PC as per the instruction itself, repeat

Question 19

Partially correct

Mark 0.86 out of 1.00

Consider the following code and MAP the file to which each fd points at the end of the code.

```
int main(int argc, char *argv[]) {
    int fd1, fd2 = 1, fd3 = 1, fd4 = 1;

    fd1 = open("/tmp/1", O_WRONLY | O_CREAT, S_IRUSR|S_IWUSR);
    fd2 = open("/tmp/2", O_RDONLY);
    fd3 = open("/tmp/3", O_WRONLY | O_CREAT, S_IRUSR|S_IWUSR);
    close(0);
    close(1);
    dup(fd2);
    dup(fd3);
    close(fd3);
    dup2(fd2, fd4);
    printf("%d %d %d %d\n", fd1, fd2, fd3, fd4);
    return 0;
}
```

1	closed	✗
fd4	/tmp/2	✓
fd2	/tmp/2	✓
fd1	/tmp/1	✓
2	stderr	✓
0	/tmp/2	✓
fd3	closed	✓

Your answer is partially correct.

You have correctly selected 6.

The correct answer is: 1 → /tmp/3, fd4 → /tmp/2, fd2 → /tmp/2, fd1 → /tmp/1, 2 → stderr, 0 → /tmp/2, fd3 → closed

Question 20

Incorrect

Mark 0.00 out of 2.00

Following code claims to implement the command

```
/bin/ls -l | /usr/bin/head -3 | /usr/bin/tail -1
```

Fill in the blanks to make the code work.

Note: Do not include space in writing any option. x[1][2] should be written without any space, and so is the case with [1] or [2]. Pay attention to exact syntax and do not write any extra character like ';' or = etc.

```
int main(int argc, char *argv[]) {
```

```
    int pid1, pid2;
```

```
    int pfd[
```

```
    x ] [2];
```

```
    pipe(
```

```
    x );
```

```
    pid1 =
```

```
    x ;
```

```
    if(pid1 != 0) {
```

```
        close(pfd[0]
```

```
    x );
```

```
        close(
```

```
    x );
```

```
        dup(
```

```
    x );
```

```
        execl("/bin/ls", "/bin/ls", "
```

```
    x ", NULL);
```

```
    }
```

```
    pipe(
```

```
    x );
```

```
    x = fork();
```

```
    if(pid2 == 0) {
```

```
        close(
```

```
        x ;
```

```
        close(0);
```

```
        dup(
```

```
        x );
```

```
        close(pfd[1]
```

```
✗ );
close(
  
✗ );
dup(
  
✗ );
execl("/usr/bin/head", "/usr/bin/head", "  
  
✗ ", NULL);
} else {
close(pfd
  
✗ );
close(
  
✗ );
dup(
  
✗ );
close(pfd
  
✗ );
execl("/usr/bin/tail", "/usr/bin/tail", "  
  
✗ ", NULL);
}  
}
```

Question 21

Partially correct

Mark 0.11 out of 1.00

Select all the correct statements about calling convention on x86 32-bit.

- a. Return address is one location above the ebp ✓
- b. Parameters may be passed in registers or on stack ✓
- c. Space for local variables is allocated by subtracting the stack pointer inside the code of the called function ✓
- d. The ebp pointers saved on the stack constitute a chain of activation records ✓
- e. The two lines in the beginning of each function, "push %ebp; mov %esp, %ebp", create space for local variables ✗
- f. Parameters may be passed in registers or on stack ✓
- g. The return value is either stored on the stack or returned in the eax register ✗
- h. Parameters are pushed on the stack in left-right order
- i. during execution of a function, ebp is pointing to the old ebp
- j. Space for local variables is allocated by subtracting the stack pointer inside the code of the caller function ✗
- k. Compiler may allocate more memory on stack than needed ✓

Your answer is partially correct.

You have selected too many options.

The correct answers are: Compiler may allocate more memory on stack than needed, Parameters may be passed in registers or on stack, Return address is one location above the ebp, during execution of a function, ebp is pointing to the old ebp, Space for local variables is allocated by subtracting the stack pointer inside the code of the called function, The ebp pointers saved on the stack constitute a chain of activation records

Question 22

Correct

Mark 1.00 out of 1.00

Match the program with its output (ignore newlines in the output. Just focus on the count of the number of 'hi')

main() { int i = fork(); if(i == 0) execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }

hi ✓

main() { fork(); execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }

hi hi ✓

main() { int i = NULL; fork(); printf("hi\n"); }

hi hi ✓

main() { execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }

hi ✓

Your answer is correct.

The correct answer is: main() { int i = fork(); if(i == 0) execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); } → hi, main() { fork(); execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); } → hi hi, main() { int i = NULL; fork(); printf("hi\n"); } → hi hi, main() { execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); } → hi

Question 23

Incorrect

Mark 0.00 out of 0.50

Some part of the bootloader of xv6 is written in assembly while some part is written in C. Why is that so?

Select all the appropriate choices

- a. The code in assembly is required for transition to protected mode, from real mode; but calling convention was applicable all the time ✗
- b. The setting up of the most essential memory management infrastructure needs assembly code ✓
- c. The code for reading ELF file can not be written in assembly ✗
- d. The code in assembly is required for transition to protected mode, from real mode; after that calling convention applies, hence code can be written in C ✓

Your answer is incorrect.

The correct answers are: The code in assembly is required for transition to protected mode, from real mode; after that calling convention applies, hence code can be written in C, The setting up of the most essential memory management infrastructure needs assembly code

Question 24

Incorrect

Mark 0.00 out of 0.50

```
xv6.img: bootblock kernel
dd if=/dev/zero of=xv6.img count=10000
dd if=bootblock of=xv6.img conv=notrunc
dd if=kernel of=xv6.img seek=1 conv=notrunc
```

Consider above lines from the Makefile. Which of the following is incorrect?

- a. The size of the kernel file is nearly 5 MB ✓
- b. The kernel is located at block-1 of the xv6.img ✗
- c. The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies 10,000 blocks on the disk. ✗
- d. The size of xv6.img is exactly = (size of bootblock) + (size of kernel) ✗
- e. The bootblock is located on block-0 of the xv6.img ✗
- f. The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies upto 10,000 blocks on the disk. ✓
- g. The bootblock may be 512 bytes or less (looking at the Makefile instruction) ✗
- h. The xv6.img is the virtual disk that is created by combining the bootblock and the kernel file. ✗
- i. The size of the xv6.img is nearly 5 MB ✗
- j. xv6.img is the virtual processor used by the qemu emulator ✓
- k. Blocks in xv6.img after kernel may be all zeroes. ✗

Your answer is incorrect.

The correct answers are: xv6.img is the virtual processor used by the qemu emulator, The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies upto 10,000 blocks on the disk., The size of the kernel file is nearly 5 MB, The size of xv6.img is exactly = (size of bootblock) + (size of kernel)

Question 25

Incorrect

Mark 0.00 out of 1.00

Select the sequence of events that are NOT possible, assuming a non-interruptible kernel code

Select one or more:

a. P1 running

P1 makes system call

timer interrupt

Scheduler

P2 running

timer interrupt

Scheuler

P1 running

P1's system call return

b. P1 running

P1 makes sytem call and blocks

Scheduler

P2 running

P2 makes sytem call and blocks

Scheduler

P1 running again



c. P1 running

P1 makes system call

system call returns

P1 running

timer interrupt

Scheduler running

P2 running

d. P1 running

P1 makes sytem call and blocks

Scheduler

P2 running

P2 makes sytem call and blocks

Scheduler

P3 running

Hardware interrupt

Interrupt unblocks P1

Interrupt returns

P3 running

Timer interrupt

Scheduler

P1 running



e.

P1 running

P1 makes sytem call

Scheduler

P2 running

P2 makes sytem call and blocks

Scheduler

P1 running again

f. P1 running

keyboard hardware interrupt

keyboard interrupt handler running

interrupt handler returns

P1 running

P1 makes sytem call

system call returns



P1 running
timer interrupt
scheduler
P2 running

Your answer is incorrect.

The correct answers are: P1 running

P1 makes system call and blocks

Scheduler

P2 running

P2 makes system call and blocks

Scheduler

P1 running again, P1 running

P1 makes system call

timer interrupt

Scheduler

P2 running

timer interrupt

Scheuler

P1 running

P1's system call return,

P1 running

P1 makes system call

Scheduler

P2 running

P2 makes system call and blocks

Scheduler

P1 running again

Question 26

Correct

Mark 0.25 out of 0.25

Which of the following are the files related to bootloader in xv6?

- a. bootasm.s and entry.S
- b. bootasm.S and bootmain.c ✓
- c. bootasm.S, bootmain.c and bootblock.c
- d. bootmain.c and bootblock.S

Your answer is correct.

The correct answer is: bootasm.S and bootmain.c

Question 27

Correct

Mark 0.25 out of 0.25

Match the following parts of a C program to the layout of the process in memory

Instructions	Text section	✓
Local Variables	Stack Section	✓
Dynamically allocated memory	Heap Section	✓
Global and static data	Data section	✓

Your answer is correct.

The correct answer is:

Instructions → Text section, Local Variables → Stack Section,
Dynamically allocated memory → Heap Section,
Global and static data → Data section

Question 28

Incorrect

Mark 0.00 out of 0.50

What will this program do?

```
int main() {  
    fork();  
    execl("/bin/ls", "/bin/ls", NULL);  
    printf("hello");  
}
```

- a. one process will run ls, another will print hello
- b. run ls once ✗
- c. run ls twice
- d. run ls twice and print hello twice
- e. run ls twice and print hello twice, but output will appear in some random order

Your answer is incorrect.

The correct answer is: run ls twice

Question 29

Correct

Mark 0.25 out of 0.25

What is the OS Kernel?

- a. The code that controls hardware, abstracts access to hardware resources using system calls, creates an environment for processes to be created and run ✓ correct
- b. The set of tools like compiler, linker, loader, terminal, shell, etc.
- c. Only the system programs like compiler, linker, loader, etc.
- d. Everything that I see on my screen

The correct answer is: The code that controls hardware, abstracts access to hardware resources using system calls, creates an environment for processes to be created and run

Question 30

Correct

Mark 0.50 out of 0.50

Which of the following is/are not saved during context switch?

- a. Program Counter
- b. General Purpose Registers
- c. Bus ✓
- d. Stack Pointer
- e. MMU related registers/information
- f. Cache ✓
- g. TLB ✓

Your answer is correct.

The correct answers are: TLB, Cache, Bus

Question 31

Partially correct

Mark 0.10 out of 0.25

Select the order in which the various stages of a compiler execute.

Linking	3	
Syntactical Analysis	2	
Pre-processing	1	
Intermediate code generation	does not exist	
Loading	4	

Your answer is partially correct.

You have correctly selected 2.

The correct answer is: Linking → 4, Syntactical Analysis → 2, Pre-processing → 1, Intermediate code generation → 3, Loading → does not exist

Question 32

Partially correct

Mark 0.08 out of 0.50

Order the sequence of events, in scheduling process P1 after process P0

context of P0 is saved in P0's PCB	2	
context of P1 is loaded from P1's PCB	3	
Process P1 is running	5	
timer interrupt occurs	6	
Process P0 is running	1	
Control is passed to P1	4	

Your answer is partially correct.

You have correctly selected 1.

The correct answer is: context of P0 is saved in P0's PCB → 3, context of P1 is loaded from P1's PCB → 4, Process P1 is running → 6, timer interrupt occurs → 2, Process P0 is running → 1, Control is passed to P1 → 5

Question 33

Not answered

Marked out of 1.00

Select the correct statements about interrupt handling in xv6 code

- a. On any interrupt/syscall/exception the control first jumps in vectors.S
- b. The trapframe pointer in struct proc, points to a location on user stack
- c. Each entry in IDT essentially gives the values of CS and EIP to be used in handling that interrupt
- d. xv6 uses the 64th entry in IDT for system calls
- e. The CS and EIP are changed only after pushing user code's SS,ESP on stack
- f. The trapframe pointer in struct proc, points to a location on kernel stack
- g. The function trap() is called only in case of hardware interrupt
- h. The CS and EIP are changed only immediately on a hardware interrupt
- i. All the 256 entries in the IDT are filled

- j. On any interrupt/syscall/exception the control first jumps in trapasm.S
- k. The function trap() is called irrespective of hardware interrupt/system-call/exception
- l. xv6 uses the 0x64th entry in IDT for system calls
- m. Before going to alltraps, the kernel stack contains upto 5 entries.

Your answer is incorrect.

The correct answers are: All the 256 entries in the IDT are filled, Each entry in IDT essentially gives the values of CS and EIP to be used in handling that interrupt, xv6 uses the 64th entry in IDT for system calls, On any interrupt/syscall/exception the control first jumps in trapasm.S, Before going to alltraps, the kernel stack contains upto 5 entries., The trapframe pointer in struct proc, points to a location on kernel stack, The function trap() is called irrespective of hardware interrupt/system-call/exception, The CS and EIP are changed only after pushing user code's SS,ESP on stack

[◀ \(Assignment\) Change free list management in xv6](#)

Jump to...

Started on Thursday, 18 March 2021, 2:46 PM

State Finished

Completed on Thursday, 18 March 2021, 3:50 PM

Time taken 1 hour 4 mins

Grade 10.36 out of 20.00 (52%)

Question 1

Partially correct

Mark 0.57 out of 1.00

Mark True, the actions done as part of code of swtch() in swtch.S, in xv6

True

False

<input checked="" type="radio"/>	<input checked="" type="radio"/>	Restore new callee saved registers from kernel stack of new context	✓
<input checked="" type="radio"/>	<input checked="" type="radio"/>	Save old callee saved registers on kernel stack of old context	✓
<input checked="" type="radio"/>	<input checked="" type="radio"/>	Save old callee saved registers on user stack of old context	✓
<input checked="" type="radio"/>	<input checked="" type="radio"/>	Switch from old process context to new process context	✗
<input checked="" type="radio"/>	<input checked="" type="radio"/>	Switch from one stack (old) to another(new)	✗
<input checked="" type="radio"/>	<input checked="" type="radio"/>	Restore new callee saved registers from user stack of new context	✓
<input checked="" type="radio"/>	<input checked="" type="radio"/>	Jump to code in new context	✗

Restore new callee saved registers from kernel stack of new context: True

Save old callee saved registers on kernel stack of old context: True

Save old callee saved registers on user stack of old context: False

Switch from old process context to new process context: False

Switch from one stack (old) to another(new): True

Restore new callee saved registers from user stack of new context: False

Jump to code in new context: False

Question 2

Partially correct

Mark 0.17 out of 0.50

For each function/code-point, select the status of segmentation setup in xv6

bootmain()	gdt setup with 3 entries, right from first line of code of bootloader	✗
kvmalloc() in main()	gdt setup with 5 entries (0 to 4) on one processor	✗
after startothers() in main()	gdt setup with 5 entries (0 to 4) on all processors	✓
after seginit() in main()	gdt setup with 5 entries (0 to 4) on all processors	✗
bootasm.S	gdt setup with 3 entries, right from first line of code of bootloader	✗
entry.S	gdt setup with 3 entries, at start32 symbol of bootasm.S	✓

Your answer is partially correct.

You have correctly selected 2.

The correct answer is: bootmain() → gdt setup with 3 entries, at start32 symbol of bootasm.S, kvmalloc() in main() → gdt setup with 3 entries, at start32 symbol of bootasm.S, after startothers() in main() → gdt setup with 5 entries (0 to 4) on all processors, after seginit() in main() → gdt setup with 5 entries (0 to 4) on one processor, bootasm.S → gdt setup with 3 entries, at start32 symbol of bootasm.S, entry.S → gdt setup with 3 entries, at start32 symbol of bootasm.S

Question 3

Partially correct

Mark 0.38 out of 1.00

Compare paging with demand paging and select the correct statements.

Select one or more:

- a. The meaning of valid-invalid bit in page table is different in paging and demand-paging. ✓
- b. Demand paging requires additional hardware support, compared to paging. ✓
- c. Paging requires some hardware support in CPU
- d. With paging, it's possible to have user programs bigger than physical memory. ✗
- e. Both demand paging and paging support shared memory pages. ✓
- f. Demand paging always increases effective memory access time.
- g. With demand paging, it's possible to have user programs bigger than physical memory. ✓
- h. Calculations of number of bits for page number and offset are same in paging and demand paging. ✓
- i. TLB hit ration has zero impact in effective memory access time in demand paging.
- j. Paging requires NO hardware support in CPU

Your answer is partially correct.

You have correctly selected 5.

The correct answers are: Demand paging requires additional hardware support, compared to paging., Both demand paging and paging support shared memory pages., With demand paging, it's possible to have user programs bigger than physical memory., Demand paging always increases effective memory access time., Paging requires some hardware support in CPU, Calculations of number of bits for page number and offset are same in paging and demand paging., The meaning of valid-invalid bit in page table is different in paging and demand-paging.

Question 4

Partially correct

Mark 0.44 out of 0.50

Suppose a processor supports base(relocation register) + limit scheme of MMU.

Assuming this, mark the statements as True/False

True	False	
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The OS may terminate the process while handling the interrupt of memory violation
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The hardware detects any memory access beyond the limit value and raises an interrupt
<input type="radio"/> <input checked="" type="checkbox"/>	<input type="radio"/>	The hardware may terminate the process while handling the interrupt of memory violation
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The OS sets up the relocation and limit registers when the process is scheduled
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The compiler generates machine code assuming continuous memory address space for process, and calculating appropriate sizes for code, and data;
<input type="radio"/> <input checked="" type="checkbox"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The process sets up its own relocation and limit registers when the process is scheduled
<input type="radio"/> <input checked="" type="checkbox"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The OS detects any memory access beyond the limit value and raises an interrupt
<input type="radio"/> <input checked="" type="checkbox"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The compiler generates machine code assuming appropriately sized segments for code, data and stack.

The OS may terminate the process while handling the interrupt of memory violation: True

The hardware detects any memory access beyond the limit value and raises an interrupt: True

The hardware may terminate the process while handling the interrupt of memory violation: False

The OS sets up the relocation and limit registers when the process is scheduled: True

The compiler generates machine code assuming continuous memory address space for process, and calculating appropriate sizes for code, and data;: True

The process sets up its own relocation and limit registers when the process is scheduled: False

The OS detects any memory access beyond the limit value and raises an interrupt: False

The compiler generates machine code assuming appropriately sized segments for code, data and stack.: False

Question 5

Correct

Mark 0.50 out of 0.50

Consider the following list of free chunks, in continuous memory management:

10k, 25k, 12k, 7k, 9k, 13k

Suppose there is a request for chunk of size 9k, then the free chunk selected under each of the following schemes will be

Best fit:

9k



First fit:

10k



Worst fit:

25k

**Question 6**

Partially correct

Mark 0.50 out of 1.00

Select all the correct statements about MMU and its functionality

Select one or more:

- a. MMU is a separate chip outside the processor
- b. MMU is inside the processor ✓
- c. Logical to physical address translations in MMU are done with specific machine instructions
- d. The operating system interacts with MMU for every single address translation ✗
- e. Illegal memory access is detected in hardware by MMU and a trap is raised ✓
- f. The Operating system sets up relevant CPU registers to enable proper MMU translations
- g. Logical to physical address translations in MMU are done in hardware, automatically ✓
- h. Illegal memory access is detected by operating system

Your answer is partially correct.

You have correctly selected 3.

The correct answers are: MMU is inside the processor, Logical to physical address translations in MMU are done in hardware, automatically, The Operating system sets up relevant CPU registers to enable proper MMU translations, Illegal memory access is detected in hardware by MMU and a trap is raised

Question 7

Incorrect

Mark 0.00 out of 0.50

Assuming a 8- KB page size, what is the page numbers for the address 874815 reference in decimal :
(give answer also in decimal)

Answer: 2186



The correct answer is: 107

Question 8

Incorrect

Mark 0.00 out of 0.25

Select the compiler's view of the process's address space, for each of the following MMU schemes:
(Assume that each scheme,e.g. paging/segmentation/etc is effectively utilised)

Segmentation, then paging	Many continuous chunks each of page size	
Relocation + Limit	Many continuous chunks of same size	
Segmentation	one continuous chunk	
Paging	many continuous chunks of variable size	

Your answer is incorrect.

The correct answer is: Segmentation, then paging → many continuous chunks of variable size, Relocation + Limit → one continuous chunk, Segmentation → many continuous chunks of variable size, Paging → one continuous chunk

Question 9

Incorrect

Mark 0.00 out of 0.50

Suppose the memory access time is 180ns and TLB hit ratio is 0.3, then effective memory access time is (in nanoseconds);

Answer: 192



The correct answer is: 306.00

Question 10

Correct

Mark 0.50 out of 0.50

In xv6, The struct context is given as

```
struct context {
    uint edi;
    uint esi;
    uint ebx;
    uint ebp;
    uint eip;
};
```

Select all the reasons that explain why only these 5 registers are included in the struct context.

- a. The segment registers are same across all contexts, hence they need not be saved ✓
- b. esp is not saved in context, because context{} is on stack and it's address is always argument to swtch() ✓
- c. xv6 tries to minimize the size of context to save memory space
- d. esp is not saved in context, because it's not part of the context
- e. eax, ecx, edx are caller save, hence no need to save ✓

Your answer is correct.

The correct answers are: The segment registers are same across all contexts, hence they need not be saved, eax, ecx, edx are caller save, hence no need to save, esp is not saved in context, because context{} is on stack and it's address is always argument to swtch()

Question 11

Partially correct

Mark 0.83 out of 1.50

Arrange the following events in order, in page fault handling:

Disk interrupt wakes up the process

7	✓
---	---

The reference bit is found to be invalid by MMU

1	✓
---	---

OS makes available an empty frame

6	✗
---	---

Restart the instruction that caused the page fault

9	✓
---	---

A hardware interrupt is issued

3	✗
---	---

OS schedules a disk read for the page (from backing store)

5	✓
---	---

Process is kept in wait state

4	✗
---	---

Page tables are updated for the process

8	✓
---	---

Operating system decides that the page was not in memory

2	✗
---	---

Your answer is partially correct.

You have correctly selected 5.

The correct answer is: Disk interrupt wakes up the process → 7, The reference bit is found to be invalid by MMU → 1, OS makes available an empty frame → 4, Restart the instruction that caused the page fault → 9, A hardware interrupt is issued → 2, OS schedules a disk read for the page (from backing store) → 5, Process is kept in wait state → 6, Page tables are updated for the process → 8, Operating system decides that the page was not in memory → 3

Question 12

Incorrect

Mark 0.00 out of 0.50

Suppose a kernel uses a buddy allocator. The smallest chunk that can be allocated is of size 32 bytes. One bit is used to track each such chunk, where 1 means allocated and 0 means free. The chunk looks like this as of now:

00001010

Now, there is a request for a chunk of 70 bytes.

After this allocation, the bitmap, indicating the status of the buddy allocator will be

Answer: 11101010



The correct answer is: 11111010

Question 13

Incorrect

Mark 0.00 out of 0.25

The complete range of virtual addresses (after main() in main.c is over), from which the free pages used by kalloc() and kfree() is derived, are:

- a. end, 4MB
- b. P2V(end), P2V(PHYSTOP)
- c. end, P2V(4MB + PHYSTOP)
- d. P2V(end), PHYSTOP ✗
- e. end, (4MB + PHYSTOP)
- f. end, PHYSTOP
- g. end, P2V(PHYSTOP)

Your answer is incorrect.

The correct answer is: end, P2V(PHYSTOP)

Question 14

Partially correct

Mark 0.33 out of 0.50

Match the pair

Hashed page table	Linear search on collision done by OS (e.g. SPARC Solaris) typically	✓
Inverted Page table	Linear/Parallel search using frame number in page table	✗
Hierarchical Paging	More memory access time per hierarchy	✓

Your answer is partially correct.

You have correctly selected 2.

The correct answer is: Hashed page table → Linear search on collision done by OS (e.g. SPARC Solaris) typically, Inverted Page table → Linear/Parallel search using page number in page table, Hierarchical Paging → More memory access time per hierarchy

Question 15

Partially correct

Mark 0.29 out of 0.50

After virtual memory is implemented

(select T/F for each of the following) One Program's size can be larger than physical memory size

True	False	
<input checked="" type="radio"/>	<input type="radio"/> ✗	Code need not be completely in memory
<input checked="" type="radio"/>	<input type="radio"/> ✗	Cumulative size of all programs can be larger than physical memory size
<input type="radio"/> ✗	<input checked="" type="radio"/>	Virtual access to memory is granted
<input checked="" type="radio"/>	<input type="radio"/> ✗	Logical address space could be larger than physical address space
<input type="radio"/> ✗	<input checked="" type="radio"/>	Virtual addresses are available
<input checked="" type="radio"/>	<input checked="" type="radio"/> ✗	Relatively less I/O may be possible during process execution
<input checked="" type="radio"/>	<input type="radio"/> ✗	One Program's size can be larger than physical memory size

Code need not be completely in memory: True

Cumulative size of all programs can be larger than physical memory size: True

Virtual access to memory is granted: False

Logical address space could be larger than physical address space: True

Virtual addresses are available: False

Relatively less I/O may be possible during process execution: True

One Program's size can be larger than physical memory size: True

Question 16

Partially correct

Mark 0.64 out of 1.00

W.r.t. Memory management in xv6,

xv6 uses physical memory upto 224 MB only
Mark statements True or False**True False**

<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The switchkvm() call in scheduler() is invoked after control comes to it from sched(), thus demanding execution in kernel's context	✓
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The stack allocated in entry.S is used as stack for scheduler's context for first processor	✓
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The switchkvm() call in scheduler() changes CR3 to use page directory kpgdir	✓
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The free page-frame are created out of nearly 222 MB	✗
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The kernel code and data take up less than 2 MB space	✓
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The switchkvm() call in scheduler() changes CR3 to use page directory of new process	✗
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The switchkvm() call in scheduler() is invoked after control comes to it from swtch() scheduler(), thus demanding execution in new process's context	✓
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	PHYSTOP can be increased to some extent, simply by editing memlayout.h	✓
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	xv6 uses physical memory upto 224 MB only	✗
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The process's address space gets mapped on frames, obtained from ~2MB:224MB range	✓
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The kernel's page table given by kpgdir variable is used as stack for scheduler's context	✗

The switchkvm() call in scheduler() is invoked after control comes to it from sched(), thus demanding execution in kernel's context: True

The stack allocated in entry.S is used as stack for scheduler's context for first processor: True

The switchkvm() call in scheduler() changes CR3 to use page directory kpgdir: True

The free page-frame are created out of nearly 222 MB: True

The kernel code and data take up less than 2 MB space: True

The switchkvm() call in scheduler() changes CR3 to use page directory of new process: False

The switchkvm() call in scheduler() is invoked after control comes to it from swtch() scheduler(), thus demanding execution in new process's context: False

PHYSTOP can be increased to some extent, simply by editing memlayout.h: True

xv6 uses physical memory upto 224 MB only: True

The process's address space gets mapped on frames, obtained from ~2MB:224MB range: True

The kernel's page table given by kpgdir variable is used as stack for scheduler's context: False

Question 17

Incorrect

Mark 0.00 out of 1.50

Consider the reference string

6 4 2 0 1 2 6 9 2 0 5

If the number of page frames is 3, then total number of page faults (including initial), using LRU replacement is:

Answer: ✖

#6# 6,4# 6,4,2 # 0,4,2#0,1,2#6,1,2#6,9,2#0,9,2#0,5,2

The correct answer is: 9

Question 18

Partially correct

Mark 0.31 out of 0.50

Consider the image given below, which explains how paging works.

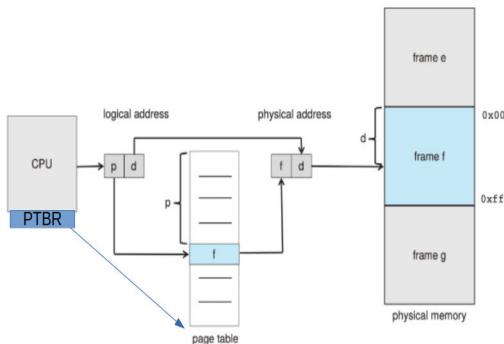


Figure 9.8 Paging hardware.

Mention whether each statement is True or False, with respect to this image.

True	False	
<input checked="" type="radio"/>	<input type="radio"/>	The PTBR is present in the CPU as a register
<input type="radio"/>	<input checked="" type="radio"/>	The page table is indexed using frame number
<input checked="" type="radio"/>	<input type="radio"/>	The page table is indexed using page number
<input type="radio"/>	<input checked="" type="radio"/>	The locating of the page table using PTBR also involves paging translation
<input type="radio"/>	<input checked="" type="radio"/>	Size of page table is always determined by the size of RAM
<input checked="" type="radio"/>	<input type="radio"/>	The page table is itself present in Physical memory
<input checked="" type="radio"/>	<input type="radio"/>	Maximum Size of page table is determined by number of bits used for page number
<input checked="" type="radio"/>	<input type="radio"/>	The physical address may not be of the same size (in bits) as the logical address

The PTBR is present in the CPU as a register: True

The page table is indexed using frame number: False

The page table is indexed using page number: True

The locating of the page table using PTBR also involves paging translation: False

Size of page table is always determined by the size of RAM: False

The page table is itself present in Physical memory: True

Maximum Size of page table is determined by number of bits used for page number: True

The physical address may not be of the same size (in bits) as the logical address: True

Question 19

Correct

Mark 2.00 out of 2.00

Given below is shared memory code with two processes sharing a memory segment.

The first process sends a user input string to second process. The second capitalizes the string. Then the first process prints the capitalized version.

Fill in the blanks to complete the code.

// First process

```
#define SHMSZ 27

int main()
{
    char c;
    int shmid;
    key_t key;
    char *shm, *s, string[128];
    key = 5679;
    if ((shmid =
        shmget
        ✓ (key, SHMSZ, IPC_CREAT | 0666)) < 0) {
        perror("shmget");
        exit(1);
    }
    if ((shm =
        shmat
        ✓ (shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
        exit(1);
    }
    s = shm;
    *s = '$';
    scanf("%s", string);
    strcpy(s + 1, string);
    *s =
        @
        ✓ ';' //note the quotes
    while(*s != '
        $
        ')
        sleep(1);
        printf("%s\n", s + 1);
        exit(0);
}
```

//Second process

```
#define SHMSZ 27

int main()
{
    int shmid;
    key_t key;
    char *shm, *s;
    int i;
    char string[128];
    key =
        5679
```

```

✓ ;
if ((shmid = shmget(key, SHMSZ, 0666)) < 0) {
    perror("shmget");
    exit(1);
}
if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
    perror("shmat");
    exit(1);
}
s =

✓ ;
while(*s != '@')
    sleep(1);
for(i = 0; i < strlen(s + 1); i++)
    s[i + 1] = toupper(s[i + 1]);
*s = '$';
exit(0);
}

```

Question 20

Partially correct

Mark 0.25 out of 0.50

Map the functionality/use with function/variable in xv6 code.

return a free page, if available; 0, otherwise

Create page table entries for a given range of virtual and physical addresses; including page directory entries if needed

Array listing the kernel memory mappings, to be used by setupkvm()

Setup kernel part of a page table, mapping kernel code, data, read-only data, I/O space, devices

Return address of page table entry in a given page directory, for a given virtual address; creates page table if necessary

Setup kernel part of a page table, and switch to that page table

 kinit1()

 mappages()

 kmap[]

 kvmalloc()

 walkpgdir()

 setupkvm()

Your answer is partially correct.

You have correctly selected 3.

The correct answer is: return a free page, if available; 0, otherwise → kalloc(), Create page table entries for a given range of virtual and physical addresses; including page directory entries if needed → mappages(), Array listing the kernel memory mappings, to be used by setupkvm() → kmap[], Setup kernel part of a page table, mapping kernel code, data, read-only data, I/O space, devices → setupkvm(), Return address of page table entry in a given page directory, for a given virtual address; creates page table if necessary → walkpgdir(), Setup kernel part of a page table, and switch to that page table → kvmalloc()

Question 21

Partially correct

Mark 1.53 out of 2.50

Order events in xv6 timer interrupt code

(Transition from process P1 to P2's code.)

P2 is selected and marked RUNNING

12 ✓

Change of stack from user stack to kernel stack of P1

3 ✓

Timer interrupt occurs

2 ✓

alltraps() will call iret

17 ✗

change to context of P2, P2's kernel stack in use now

13 ✓

P2's trap() will return to alltraps

16 ✗

jump in vector.S

4 ✓

P2 will return from sched() in yield()

14 ✗

yield() is called

8 ✓

trap() is called

7 ✓

Process P2 is executing

18 ✗

P1 is marked as RUNNABLE

9 ✓

P2's yield() will return in trap()

15 ✗

Process P1 is executing

1 ✓

sched() is called,

11 ✗

change to context of the scheduler, scheduler's stack in use now

10 ✗

jump to alltraps

5 ✓

Trapframe is built on kernel stack of P1

6 ✓

Your answer is partially correct.

You have correctly selected 11.

The correct answer is: P2 is selected and marked RUNNING → 12, Change of stack from user stack to kernel stack of P1 → 3, Timer interrupt occurs → 2, alltraps() will call iret → 18, change to context of P2, P2's kernel stack in use now → 13, P2's trap() will return to alltraps → 17, jump in vector.S → 4, P2 will return from sched() in yield() → 15, yield() is called → 8, trap() is called → 7, Process P2 is executing → 14, P1 is marked as RUNNABLE → 9, P2's yield() will return in trap() → 16, Process P1 is executing → 1, sched() is called, → 10, change to context of the scheduler, scheduler's stack in use now → 11, jump to alltraps → 5, Trapframe is built on kernel stack of P1 → 6

Question 22

Incorrect

Mark 0.00 out of 1.00

Given that the memory access time is 200 ns, probability of a page fault is 0.7 and page fault handling time is 8 ms,
The effective memory access time in nanoseconds is:

Answer: ✖

The correct answer is: 5600060.00

Question 23

Correct

Mark 0.25 out of 0.25

Select the state that is not possible after the given state, for a process:

- New: Running ✓
- Ready : Waiting ✓
- Running: None of these ✓
- Waiting: Running ✓

Question 24

Partially correct

Mark 0.63 out of 1.00

Select the correct statements about sched() and scheduler() in xv6 code

- a. scheduler() switches to the selected process's context ✓
- b. When either sched() or scheduler() is called, it does not return immediately to caller ✓
- c. After call to swtch() in sched(), the control moves to code in scheduler()
- d. Each call to sched() or scheduler() involves change of one stack inside swtch() ✓
- e. After call to swtch() in scheduler(), the control moves to code in sched()
- f. When either sched() or scheduler() is called, it results in a context switch ✓
- g. sched() switches to the scheduler's context ✓
- h. sched() and scheduler() are co-routines

Your answer is partially correct.

You have correctly selected 5.

The correct answers are: sched() and scheduler() are co-routines, When either sched() or scheduler() is called, it does not return immediately to caller, When either sched() or scheduler() is called, it results in a context switch, sched() switches to the scheduler's context, scheduler() switches to the selected process's context, After call to swtch() in scheduler(), the control moves to code in sched(), After call to swtch() in sched(), the control moves to code in scheduler(), Each call to sched() or scheduler() involves change of one stack inside swtch()

Question 25

Correct

Mark 0.25 out of 0.25

The data structure used in kalloc() and kfree() in xv6 is

- a. Doubly linked circular list
- b. Singly linked circular list
- c. Double linked NULL terminated list
- d. Singly linked NULL terminated list



Your answer is correct.

The correct answer is: Singly linked NULL terminated list

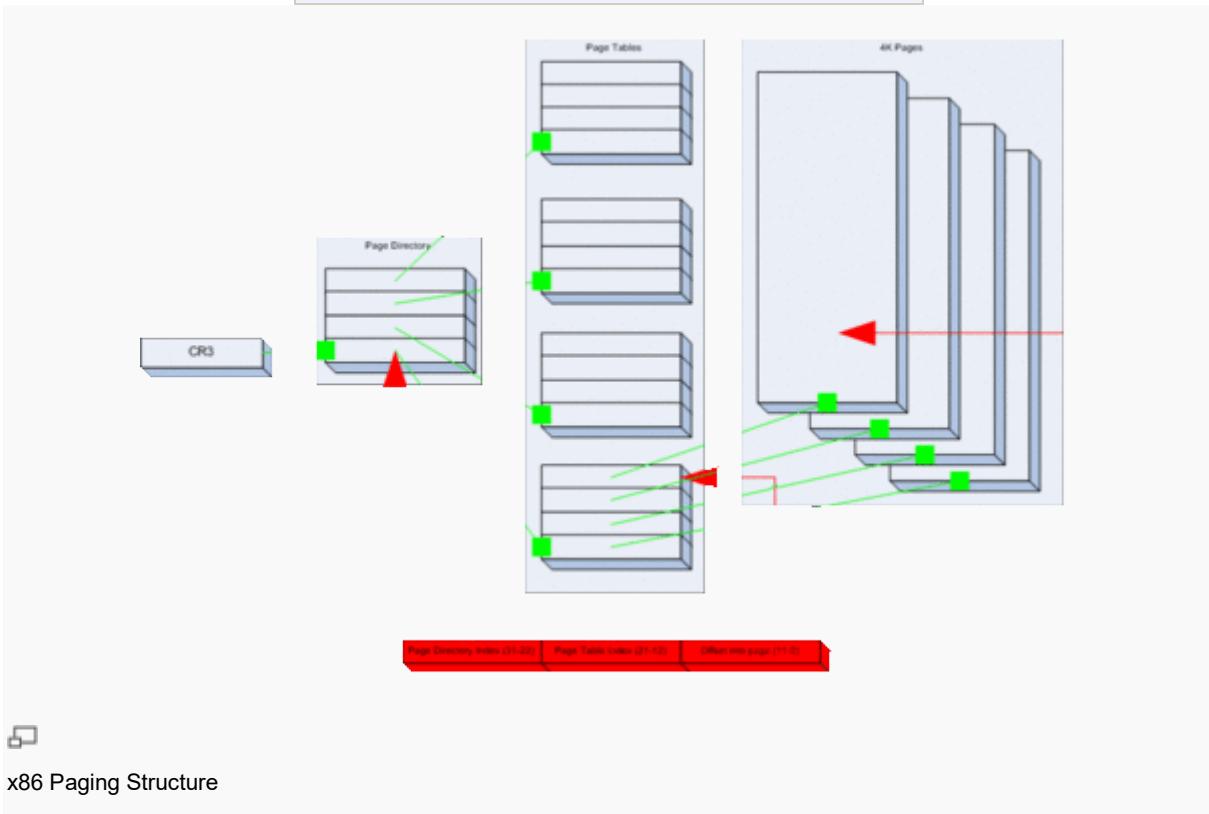
[◀ \(Assignment\) lseek system call in xv6](#)

Jump to...

Paging

The factual accuracy of this article or section is [disputed](#).

Please see the relevant discussion on the [talk page](#).



Paging is a system which allows each process to see a full virtual address space, without actually requiring the full amount of physical memory to be available or present. 32-bit x86 processors support 32-bit virtual addresses and 4-GiB virtual address spaces, and current 64-bit processors support 48-bit virtual addressing and 256-TiB virtual address spaces. Intel has released [documentation](#) for a extension to 57-bit virtual addressing and 128-PiB virtual address spaces. Currently, implementations of x86-64 have a limit of between 4 GiB and 256 TiB of physical address space (and an architectural limit of 4 PiB of physical address space).

In addition to this, paging introduces the benefit of page-level protection. In this system, user processes can only see and modify data which is paged in on their own address space, providing hardware-based isolation. System pages are also protected from user processes. On the x86-64 architecture, page-level protection now completely supersedes [Segmentation](#) as the memory protection mechanism. On the IA-32 architecture, both paging and segmentation exist, but segmentation is now considered 'legacy'.

Once an Operating System has paging, it can also make use of other benefits and workarounds, such as linear framebuffer simulation for memory-mapped IO and paging out to disk, where disk storage space is used to free up physical RAM.

Contents

[\[hide\]](#)

- [1_32-bit Paging \(Protected Mode\)](#)
 - [1.1_MMU](#)
 - [1.2_Page Directory](#)
 - [1.3_Page Table](#)
 - [1.4_Example](#)
- [2_64-Bit Paging](#)
 - [2.1_Page Map Table Entries](#)
 - [2.2_Process Context Identifiers](#)
- [3_Enabling](#)
 - [3.1_32-bit Paging](#)
 - [3.2_64-bit Paging](#)
- [4_Physical Address Extension](#)
- [5_Usage](#)
 - [5.1_Virtual Address Spaces](#)
 - [5.2_Virtual Memory](#)
- [6_Manipulation](#)
- [7_Page Faults](#)
 - [7.1_Handling](#)
- [8_INVLPG](#)
- [9_Paging Tricks](#)
- [10_PAT](#)
- [11_See Also](#)
 - [11.1_Articles](#)
 - [11.2_External Links](#)

32-bit Paging (Protected Mode)

MMU

Paging is achieved through the use of the [Memory Management Unit](#) (MMU). On the x86, the MMU maps memory through a series of [tables](#), two to be exact. They are the paging directory (PD), and the paging table (PT).

Both [tables](#) contain 1024 4-byte entries, making them 4 KiB each. In the page directory, each entry points to a page table. In the page table, each entry points to a 4 KiB physical page frame. Additionally, each

entry has bits controlling access protection and caching features of the structure to which it points. The entire system consisting of a page directory and page tables represents a linear 4-GiB virtual memory map.

Translation of a virtual address into a physical address first involves dividing the virtual address into three parts: the most significant 10 bits (bits 22-31) specify the index of the page directory entry, the next 10 bits (bits 12-21) specify the index of the page table entry, and the least significant 12 bits (bits 0-11) specify the page offset. The then MMU walks through the paging structures, starting with the page directory, and uses the page directory entry to locate the page table. The page table entry is used to locate the base address of the physical page frame, and the page offset is added to the physical base address to produce the physical address. If translation fails for some reason (entry is marked as not present, for example), then the processor issues a page fault.

Page Directory

The topmost paging structure is the page directory. It is essentially an array of page directory entries that take the following form.

Page Directory Entry (4 MB)

31	...	22	21	20	...	13	12	11	...	9	8	7	6	5	4	3	2	1	0
		R S V D (0)		Bits 31-22 of address	P A T	Bits 39-32 of address	AVL	G S (1)	D A	P C W D T	P P U S T S W	R /	P						

Page Directory Entry

31	...	12	11	...	8	7	6	5	4	3	2	1	0
		Bits 31-12 of address		AVL	P S (0) L	A V A	P C W D T	P P U S T S W	R /	P			

P: Present	D: Dirty
R/W: Read/Write	PS: Page Size
U/S: User/Supervisor	G: Global
PWT: Write-Through	AVL: Available
PCD: Cache Disable	PAT: Page Attribute Table
A: Accessed	

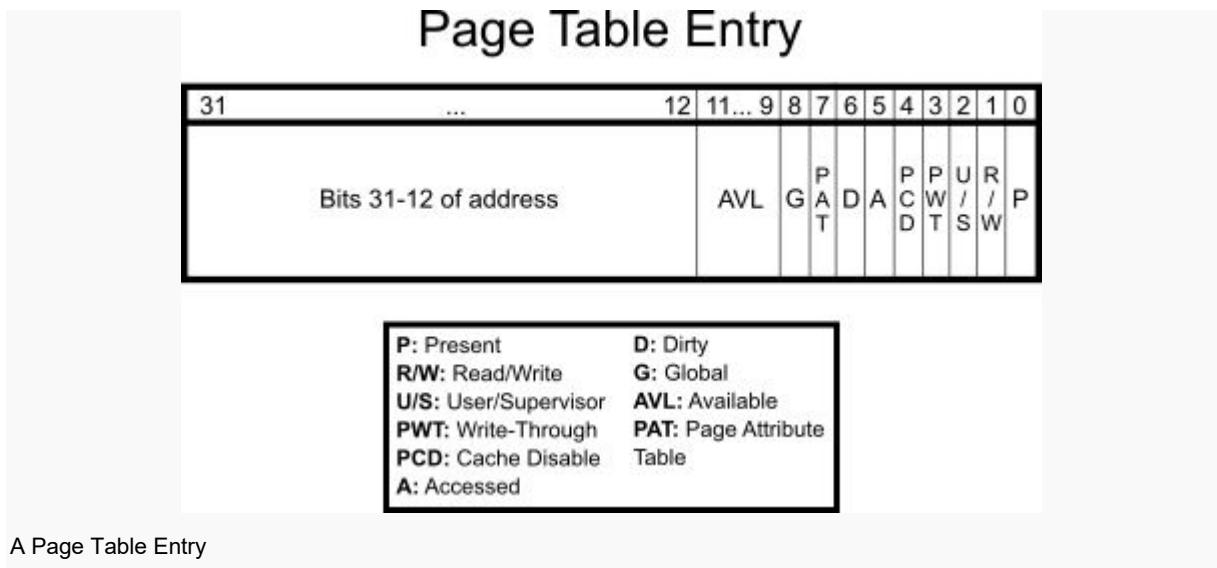
A Page Directory Entry

When PS=0, the page table address field represents the physical address of the page table that manages the four megabytes at that point. Please note that it is very important that this address be 4-KiB aligned. This is needed, due to the fact that the last 12 bits of the 32-bit value are overwritten by access bits and such. Similarly, when PS=1, the address must be 4-MiB aligned.

- PAT, or '**P**age **A**ttribute **T**able'. If PAT is supported, then PAT along with PCD and PWT shall indicate the memory caching type. Otherwise, it is reserved and must be set to 0.
- G, or '**G**lobal' tells the processor not to invalidate the TLB entry corresponding to the page upon a MOV to CR3 instruction. Bit 7 (PGE) in CR4 must be set to enable global pages.
- PS, or '**P**age **S**ize' stores the page size for that specific entry. If the bit is set, then the PDE maps to a page that is 4 MiB in size. Otherwise, it maps to a 4 KiB page table. Please note that 4-MiB pages require PSE to be enabled.
- D, or '**D**irty' is used to determine whether a page has been written to.
- A, or '**A**ccessed' is used to discover whether a PDE or PTE was read during virtual address translation. If it has, then the bit is set, otherwise, it is not. Note that, this bit will not be cleared by the CPU, so that burden falls on the OS (if it needs this bit at all).
- PCD, is the 'Cache Disable' bit. If the bit is set, the page will not be cached. Otherwise, it will be.
- PWT, controls Write-Through' abilities of the page. If the bit is set, write-through caching is enabled. If not, then write-back is enabled instead.
- U/S, the '**U**ser/**S**upervisor' bit, controls access to the page based on privilege level. If the bit is set, then the page may be accessed by all; if the bit is not set, however, only the supervisor can access it. For a page directory entry, the user bit controls access to all the pages referenced by the page directory entry. Therefore if you wish to make a page a user page, you must set the user bit in the relevant page directory entry as well as the page table entry.
- R/W, the '**R**ead/**W**rite' permissions flag. If the bit is set, the page is read/write. Otherwise when it is not set, the page is read-only. The WP bit in CR0 determines if this is only applied to userland, always giving the kernel write access (the default) or both userland and the kernel (see Intel Manuals 3A 2-20).
- P, or '**P**resent'. If the bit is set, the page is actually in physical memory at the moment. For example, when a page is swapped out, it is not in physical memory and therefore not 'Present'. If a

page is called, but not present, a page fault will occur, and the OS should handle it. (See below.)

The remaining bits 9 through 11 (if PS=0, also bits 6 & 8) are not used by the processor, and are free for the OS to store some of its own accounting information. In addition, when P is not set, the processor ignores the rest of the entry and you can use all remaining 31 bits for extra information, like recording where the page has ended up in swap space. When changing the accessed or dirty bits from 1 to 0 while an entry is marked as present, it's recommended to invalidate the associated page. Otherwise, the processor may not set those bits upon subsequent read/writes due to TLB caching.



Setting the PS bit makes the page directory entry point directly to a 4-MiB page. There is no paging table involved in the address translation. Note: With 4-MiB pages, whether or not bits 20 through 13 are reserved depends on PSE being enabled and how many PSE bits are supported by the processor (PSE, PSE-36, PSE-40). [CPUID](#) should be used to determine this. Thus, the physical address must also be 4-MiB-aligned. Physical addresses above 4 GiB can only be mapped using 4 MiB PDEs.

Page Table

In each page table, as it is, there are also 1024 entries. These are called page table entries, and are **very** similar to page directory entries.

The first item, is once again, a 4-KiB aligned physical address. Unlike previously, however, the address is not that of a page table, but instead a 4 KiB block of physical memory that is then mapped to that location in the page table and directory. Note that the PAT bit is bit 7 instead of bit 12 as in the 4 MiB PDE.

Example

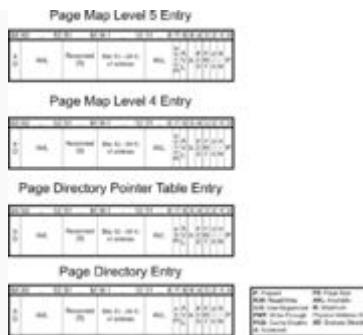
Say the kernel is loaded to 0x100000. However, it needed to be remapped to 0xC0000000. After loading the kernel, it'll initiate paging, and set up the appropriate tables. (See [Higher Half Kernel](#)) After [Identity Paging](#) the first megabyte, it'll need to create a second table (ie. at entry #768 in the paging directory.) to map 0x100000 to 0xC0000000. The code may be like:

```

mov eax, 0x0
mov ebx, 0x100000
.fill_table:
    mov ecx, ebx
    or ecx, 3
    mov [table_768+eax*4], ecx
    add ebx, 4096
    inc eax
    cmp eax, 1024
    jne .fill_table

```

64-Bit Paging

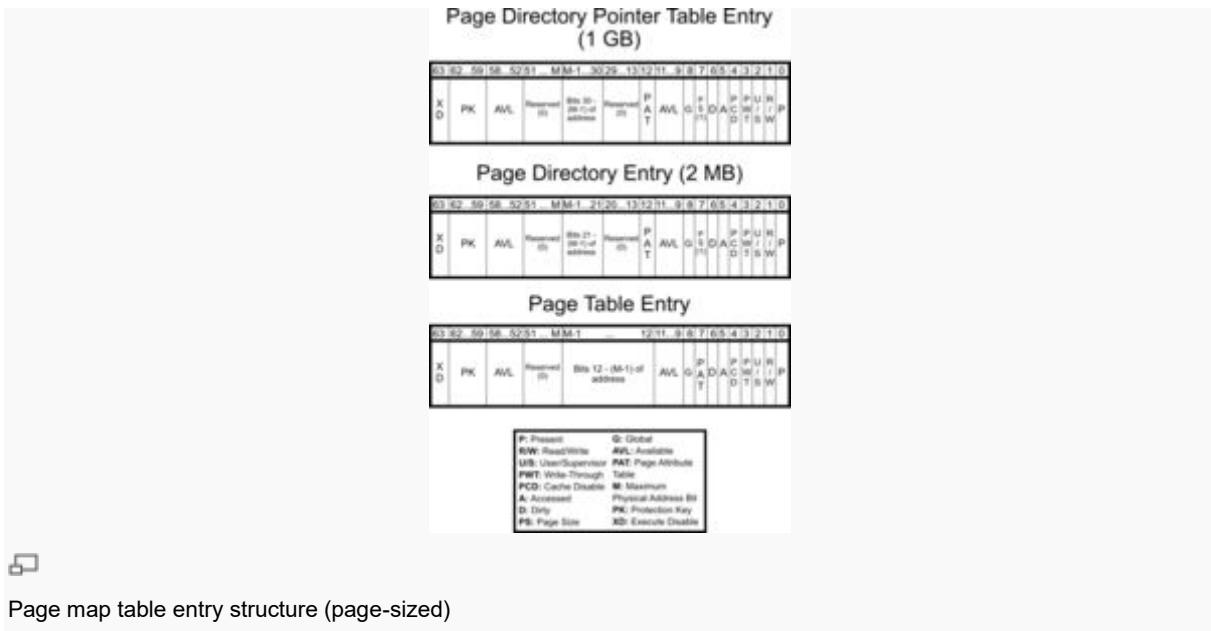


Page map table entry structure (non-page-sized)

Paging in [long mode](#) is similar to that of 32-bit paging, except [Physical Address Extension](#) (PAE) is required. Registers CR2 and CR3 are extended to 64 bits. Instead of just having to utilize 3 levels of page maps: page directory pointer table, page directory, and page table, a fourth page-map table is used: the level-4 page map table (PML4). This allows a processor to map 48-bit virtual addresses to 52-bit physical addresses. If level-5 page maps are supported and enabled, then a fifth page-map table, the level-5 page map table (PML5), allows the processor to map 57-bit virtual addresses to 52-bit physical addresses. Both the PML4 and PML5 contain 512 64-bit entries of which each may point to a lower-level page map table. Do note that with each additional level of paging, virtual addressing becomes slower, especially in the case of TLB cache misses.

Virtual addresses in 64-bit mode must be **canonical**, that is, the upper bits of the address must either be all 0s or all 1s. For systems supporting 48-bit virtual address spaces, the upper 16 bits must be the same, and for systems supporting 57-bit virtual addresses, the upper 7 bits must match. Although 32-bit code running in [long mode](#) (compatibility mode) is still limited to 32-bit virtual addresses, they can still map to a 52-bit physical addresses.

Page Map Table Entries



New bits have been added to page map table entries for long-mode paging:

- XD, or 'Execute Disable'. If the NXE bit (bit 11) is set in the [EFER register](#), then instructions are not allowed to be executed at addresses within the page whenever XD is set. If EFER.NXE bit is 0, then the XD bit is reserved and should be set to 0.
- PK, or 'Protection Key'. The protection key is a 4-bit corresponding to each virtual address that is used to control user-mode and supervisor-mode memory accesses. If the PKE bit (bit 22) in CR4 is set, then the PKRU register is used for determining access rights for user-mode based on the protection key. If the PKS bit (bit 24) is set in CR4, then the PKRS register is used for determining access rights for supervisor-mode based on the protection key. A protection key allows the system to enable/disable access rights for multiple page entries across different address spaces at once.

M signifies the physical address width supported by a processor using PAE. Currently, up to 52 bits are supported, but the actual supported width may be less.

Bits marked as reserved must all be set to 0, otherwise, a page fault will occur with a reserved error code.

Support for 1 GiB pages, (NX) execute disable, (PKS/PKU) protection keys for supervisor-mode and user-mode pages, shadow stack pages, (M) physical address width, virtual address width, (PAT) page attribute table, (PCID) process context identifiers, and (LA57) 5-level paging can be determined with the [CPUID](#) instruction (EAX:0x01; EAX:0x07, ECX=0x00; EAX:0x80000001; EAX:0x80000008).

Process Context Identifiers

If process context ids (PCID) are supported, then bits 0-11 of CR3 specify the process context id.

Otherwise, bit 3 is PWT for PML4, and bit 4 is PCD for PML4. PCIDs are used to control TLB caching across multiple address spaces. The INVPcid instruction uses PCIDs to allow more control over page invalidation.

Enabling

32-bit Paging

Enabling paging is actually very simple. All that is needed is to load CR3 with the address of the page directory and to set the paging (PG) and protection (PE) bits of CR0.

```
mov eax, page_directory  
mov cr3, eax  
  
mov eax, cr0  
or eax, 0x80000001  
mov cr0, eax
```

Note: setting the paging flag when the protection flag is clear causes a [general protection exception](#). Also, once paging has been enabled, any attempt to enable long mode by setting LME (bit 8) of the [EFER register](#) will trigger a [GPF](#). The CR0.PG must first be cleared before EFER.LME can be set.

If you want to set pages as read-only for both userspace and supervisor, replace 0x80000001 above with 0x80010001, which also sets the WP bit.

To enable PSE (4 MiB pages) the following code is required.

```
mov eax, cr4  
or eax, 0x00000010  
mov cr4, eax
```

64-bit Paging

Enabling paging in long mode requires a few more additional steps. Since it is not possible to enter long mode without paging with PAE active, the order in which one enables the bits are important. Firstly, paging must not be active (i.e. CR0.PG must be cleared.) Then, CR4.PAE (bit 5) and EFER.LME (bit 8 of MSR 0xC0000080) are set. If 57-bit virtual addresses are to be enabled, then CR4.LA57 (bit 12) is set. Finally, CR0.PG is set to enable paging.

```
; Skip these 3 lines if paging is already disabled  
mov ebx, cr0  
and ebx, ~(1 << 31)  
mov cr0, ebx  
  
; Enable PAE  
mov edx, cr4
```

```

or edx, (1 << 5)
mov cr4, edx

; Set LME (long mode enable)
mov ecx, 0xC0000080
rdmsr
or eax, (1 << 8)
wrmsr

; Replace 'pml4_table' with the appropriate physical address (and flags, if applicable)
mov eax, pml4_table
mov cr3, eax

; Enable paging (and protected mode, if it isn't already active)
or ebx, (1 << 31) | (1 << 0)
mov cr0, ebx

; Now reload the segment registers (CS, DS, SS, etc.) with the appropriate segment selectors...
mov ax, DATA_SEL
mov ds, ax
mov es, ax
mov fs, ax
mov gs, ax

; Reload CS with a 64-bit code selector by performing a long jmp

jmp CODE_SEL:reloadCS

[BITS 64]
reloadCS:
hlt ; Done. Replace these lines with your own code
jmp reloadCS

```

Once paging has been enabled, you cannot switch from 4-level paging to 5-level paging (and vice-versa) directly. The same is true for switching to legacy 32-bit paging. You must first disable paging by clearing CR0.PG before making changes. Failure to do so will result in a [general protection fault](#).

Physical Address Extension

All Intel processors since Pentium Pro (with exception of the Pentium M at 400 Mhz) and all AMD since the Athlon series implement the [Physical Address Extension](#) (PAE). This feature allows you to access up to 4 PiB (2^{52}) of RAM. You can check for this feature using [CPUID](#). Once checked, you can activate this feature by setting bit 5 in CR4.

For legacy 32-bit PAE, the CR3 register points to a page directory pointer table (PDPT) of 4 64-bit entries, each one pointing to a page directory made of 4096 bytes (like in normal paging), divided into 512 64-bit entries, each pointing to a 4096-byte page table, divided into 512 64bit page entries. Keep in mind that virtual addresses are still limited to 4 GiB (2^{32} bytes).

For 4-level and 5-level PAE, as used in compatibility mode and [long mode](#), the CR3 register points to the top-level page map table: the PML4 table and PML5 table, respectively. Each of the page map tables: PML5 table, PML4 table, page directory pointer table, page directory, page table, contain 512 64-bit entries.

If paging is enabled then PAE must also be enabled before entering long mode. Attempting to enter long mode with CR0.PG set and CR4.PAE cleared will trigger a general protection fault.

Usage

Due to the simplicity in the design of paging, it has many uses.

Virtual Address Spaces

In a paged system, each process may execute in its own area of memory, without any chance of affecting any other process's memory, or the kernel's. Two or more processes may opt to share memory by mapping the same physical page(s) to addresses in their own address spaces. The virtual address of each mapping do not need to be the same. Consequently, a virtual address in one address space won't point to the same data in other address spaces, in general.

Physical Memory		Process A		Process B	
		Page Table	Virtual Memory	Page Table	Virtual Memory
00x	H E L L	00x 00	00x H E L L	00x 03	00x H A V E
01x	R L D !	01x 02	01x O W O	01x 05	01x L O T
02x	O W O	02x 01	02x R L D !	02x 06	02x S O F
03x	H A V E	03x n.a.	03x #####	03x 04	03x F U N
04x	F U N	04x n.a.	04x #####	04x n.a.	04x #####
05x	L O T	05x 07	05x ; -)	05x 07	05x ; -)
06x	S O F				
07x	; -)				

paging illustrated: two process with different views of the same physical memory

Virtual Memory

Because paging allows for the dynamic handling of unallocated page tables, an OS can swap entire pages, not in current use, to the hard drive where they can wait until they are called. In the mean time, however, the physical memory that they were using can be used elsewhere. In this way, the OS can manipulate the system so that programs actually seem to have more RAM than there actually is.

More...

Manipulation

The CR3 value, that is, the value containing the address of the page directory, is in physical form. Once, then, the computer is in paging mode, only recognizing those virtual addresses mapped into the paging tables, how can the tables be edited and dynamically changed?

Many prefer to map the last PDE to itself. The page directory will look like a page table to the system. To get the physical address of any virtual address in the range 0x00000000-0xFFFFF000 is then just a matter of:

```
void *get_physaddr(void *virtualaddr) {
    unsigned long pdindex = (unsigned long)virtualaddr >> 22;
    unsigned long ptindex = (unsigned long)virtualaddr >> 12 & 0x03FF;

    unsigned long *pd = (unsigned long *)0xFFFFF000;
    // Here you need to check whether the PD entry is present.

    unsigned long *pt = ((unsigned long *)0xFFC00000) + (0x400 * pdindex);
    // Here you need to check whether the PT entry is present.

    return (void *)((pt[ptindex] & ~0xFFF) + ((unsigned long)virtualaddr &
0FFF));
}
```

To map a virtual address to a physical address can be done as follows:

```
void map_page(void *physaddr, void *virtualaddr, unsigned int flags) {
    // Make sure that both addresses are page-aligned.

    unsigned long pdindex = (unsigned long)virtualaddr >> 22;
    unsigned long ptindex = (unsigned long)virtualaddr >> 12 & 0x03FF;

    unsigned long *pd = (unsigned long *)0xFFFFF000;
    // Here you need to check whether the PD entry is present.
    // When it is not present, you need to create a new empty PT and
    // adjust the PDE accordingly.

    unsigned long *pt = ((unsigned long *)0xFFC00000) + (0x400 * pdindex);
    // Here you need to check whether the PT entry is present.
    // When it is, then there is already a mapping present. What do you do now?

    pt[ptindex] = ((unsigned long)physaddr) | (flags & 0xFFF) | 0x01; // Present

    // Now you need to flush the entry in the TLB
    // or you might not notice the change.
}
```

Unmapping an entry is essentially the same as above, but instead of assigning the `pt[ptindex]` a value, you set it to 0x00000000 (i.e. not present). When the entire page table is empty, you may want to remove it

and mark the page directory entry 'not present'. Of course you don't need the 'flags' or 'physaddr' for unmapping.

Page Faults

A [page fault](#) exception is caused when a process is seeking to access an area of virtual memory that is not mapped to any physical memory, when a write is attempted on a read-only page, when accessing a PTE or PDE with the reserved bit or when permissions are inadequate. A [page fault](#) can either be pure, which occurs when the faulting process has permission to access the page, or invalid, which is due to a protection violation. Pure [page faults](#) aren't errors, but are resolved through the page fault handler by performing the appropriate map operation and/or page swap.

Handling

The CPU pushes an error code on the stack before firing a [page fault exception](#). The error code must be analyzed by the exception handler to determine how to handle the exception. The following bits are the only ones used, all others are reserved.

```
Bit 0 (P) is the Present flag.  
Bit 1 (R/W) is the Read/Write flag.  
Bit 2 (U/S) is the User/Supervisor flag.  
Bit 3 (RSVD) indicates whether a reserved bit was set in some page-  
structure entry  
Bit 4 (I/D) is the Instruction/Data flag (1=instruction fetch, 0=data  
access)  
Bit 5 (PK) indicates a protection-key violation  
Bit 6 (SS) indicates a shadow-stack access fault  
Bit 15 (SGX) indicates an SGX violation
```

The combination of these flags specify the details of the page fault and indicate what action to take:

US	RW	P	Description
0	0	0	- Supervisory process tried to read a non-present page entry
0	0	1	- Supervisory process tried to read a page and caused a protection fault
0	1	0	- Supervisory process tried to write to a non-present page entry
0	1	1	- Supervisory process tried to write a page and caused a protection fault
1	0	0	- User process tried to read a non-present page entry
1	0	1	- User process tried to read a page and caused a protection fault
1	1	0	- User process tried to write to a non-present page entry
1	1	1	- User process tried to write a page and caused a protection fault

When the CPU fires a page-not-present exception the CR2 register is populated with the linear address that caused the exception. The upper 10 bits specify the page directory entry (PDE) and the middle 10 bits specify the page table entry (PTE). First check the PDE and see if its present bit is set, if not setup a page table and point the PDE to the base address of the page table, set the present bit and iretd. If the PDE is present then the present bit of the PTE will be cleared. You'll need to map some physical memory to the page table, set the present bit and then iretd to continue processing.

INVLPG

INVLPG is an instruction available since the i486 that invalidates a single page in the TLB. Intel notes that this instruction may be implemented differently on future processes, but that this alternate behavior must be explicitly enabled. INVLPG modifies no flags.

NASM example:

```
invlpg [0]
```

Inline assembly for GCC (from Linux kernel source):

```
static inline void __native_flush_tlb_single(unsigned long addr) {
    asm volatile("invlpg (%0)" ::"r" (addr) : "memory");
}
```

This only invalidates the page on the current processor. If you're using SMP, you'll need to send an IPI to the other processors so that they can also invalidate the page (this is called a TLB shootdown; it's very slow), making sure to avoid any nasty race conditions. You may only want to do this when removing a mapping, and just make your page fault handler invalidate a page if it didn't invalidate a mapping addition on that processor by looking through the page directory, again avoiding race conditions.

When you modify an entry in the page directory, rather than just a page table, you'll need to invalidate each page in the table. Alternatively, you could reload CR3 which will invalidates the whole directory, but this may be slower. (TODO time this)

Paging Tricks

The processor always fires a page fault exception when the present bit is cleared in the PDE or PTE regardless of the address. This means the contents of the PTE or PDE can be used to indicate a location of the page saved on mass storage and to quickly load it. When a page gets swapped to disk, use these entries to identify the location in the paging file where they can be quickly loaded from then set the present bit to 0. Similarly, blocks from disk can be mapped to memory this way. When a process accesses the memory-mapped region, a page fault occurs. The fault handler reads the appropriate tables, loads the disk block(s) into a page, and maps it. The process can then read/write to memory as if it were accessing the device directly. The contents of the page would then be written back to disk to save the changes.

For memory efficiency, two or more processes can share pages as read-only. If one process were to write to its page, then a page fault would occur and the system could duplicate the page and then mark it as

read-write. This is known as copy-on-write (COW). Copy-on-write allows the system to delay memory allocation until a process actually requires it, preventing unnecessary copying.

PAT

The Page Attribute Table determines caching attributes on a page granularity. This is similar to [MTRRs](#), but those apply to physical addresses and are more limited.

The PAT is set via the IA32_PAT_MSR [MSR](#) (0x277). It has 8 entries, taking the low order 3 bits of each byte, in standard little endian order. So the high byte is PAT7, low byte is PAT0.

The following are the different caching types.

Number	Name	Description
0	UC — Uncacheable	All accesses are uncacheable. Write combining is not allowed. Speculative accesses are not allowed.
1	WC — Write-Combining	All accesses are uncacheable. Write combining is allowed. Speculative reads are allowed.
4	WT — Writethrough	Reads allocate cache lines on a cache miss. Cache lines are not allocated on a write miss. Write hits update the cache and main memory.
5	WP — Write-Protect	Reads allocate cache lines on a cache miss. All writes update main memory. Cache lines are not allocated on a write miss. Write hits invalidate the cache line and update main memory.
6	WB — Writeback	Reads allocate cache lines on a cache miss, and can allocate to either the shared, exclusive, or modified state. Writes allocate to the modified state on a cache miss.
7	UC- — Uncached	Same as uncacheable, <i>except</i> that this can be overridden by Write-Combining MTRRs.

The PAT has a reset value of 0x0007040600070406. This ensures compatibility with non-PAT usage. This corresponds to the following:

UC	UC-	WT	WB	UC	UC-	WT	WB
----	-----	----	----	----	-----	----	----

The PAT is indexed by the three page table bits:

PAT	PCD	PWT
-----	-----	-----

The PAT bit is reserved when there isn't a PAT, and the default value of the MSR ensures backwards compatibility with the PCD and PWT bit.

You will need to modify the PAT if you want Write-Combining cache, which is very useful for framebuffers.

Ext2

[Filesystems](#)

Virtual Filesystems

[VFS](#)

Disk Filesystems

[FAT 12/16/32](#), [VFAT](#), [ExFAT](#)

[Ext 2/3/4](#)

[LEAN](#)

[HPFS](#)

[NTFS](#)

[HFS](#)

[HFS+](#)

[MFS](#)

[ReiserFS](#)

[FFS \(Amiga\)](#)

[FFS \(BSD\)/UFS](#)

[BeFS](#)

[BFS](#)

[XFS](#)

[SFS](#)

[ZDSFS](#)

[ZFS](#)

[USTAR](#)

CD/DVD Filesystems

[ISO 9660](#)

[Joliet](#)

[UDF](#)

[Network Filesystems](#)

[NFS](#)

[RFS](#)

[AFS](#)

Flash Filesystems

[JFFS2](#)

[YAFFS](#)

The **Second Extended Filesystem (ext2fs)** is a rewrite of the original *Extended Filesystem* and as such, is also based around the concept of "inodes." Ext2 served as the de facto filesystem of Linux for nearly a decade from the early 1990s to the early 2000s when it was superseded by the journaling file systems [ext3](#) and [ReiserFS](#). It has native support for UNIX ownership / access rights, symbolic- and hard-links, and other properties that are common among UNIX-like operating systems. Organizationally, it divides disk space up into groups called "block groups." Having these groups results in distribution of data across the disk which helps to minimize head movement as well as the impact of fragmentation. Further, some (if not all) groups are required to contain backups of important data that can be used to rebuild the file system in the event of disaster.

Note: Most of the information here is based off of work done by Dave Poirier on the ext2-doc project (see the [links section](#)) which is graciously released under the [GNU Free Documentation License](#). Be sure to buy him a beer the next time you see him.

Contents

[\[hide\]](#)

- [1_Basic Concepts](#)
 - [1.1_What is a Block?](#)
 - [1.2_What is a Block Group?](#)
 - [1.3_What is an Inode?](#)
- [2_Superblock](#)

- [2.1 Locating the Superblock](#)
- [2.2 Determining the Number of Block Groups](#)
- [2.3 Base Superblock Fields](#)
 - [2.3.1 File System States](#)
 - [2.3.2 Error Handling Methods](#)
 - [2.3.3 Creator Operating System IDs](#)
- [2.4 Extended Superblock Fields](#)
 - [2.4.1 Optional Feature Flags](#)
 - [2.4.2 Required Feature Flags](#)
 - [2.4.3 Read-Only Feature Flags](#)
- [3 Block Group Descriptor Table](#)
 - [3.1 Locating the Block Group Descriptor Table](#)
 - [3.2 Block Group Descriptor](#)
- [4 Inodes](#)
 - [4.1 Determining which Block Group contains an Inode](#)
 - [4.2 Finding an inode inside of a Block Group](#)
 - [4.3 Reading the contents of an inode](#)
 - [4.4 Inode Data Structure](#)
 - [4.4.1 Inode Type and Permissions](#)
 - [4.4.2 Inode Flags](#)
 - [4.4.3 OS Specific Value 1](#)
 - [4.4.4 OS Specific Value 2](#)
 - [4.5 Directories](#)
 - [4.6 Directory Entry](#)
 - [4.6.1 Directory Entry Type Indicators](#)
- [5 Quick Summaries](#)
 - [5.1 How To Read An Inode](#)
 - [5.2 How To Read the Root Directory](#)
- [6 See Also](#)
 - [6.1 External Links](#)

Basic Concepts

Important Note: All values are little-endian unless otherwise specified

What is a Block?

The Ext2 file system divides up disk space into logical blocks of contiguous space. The size of blocks need not be the same size as the sector size of the disk the file system resides on. The size of blocks can be determined by reading the field starting at byte 24 in the [Superblock](#).

What is a Block Group?

Blocks, along with inodes, are divided up into "block groups." These are nothing more than contiguous groups of blocks.

Each block group reserves a few of its blocks for special purposes such as:

- A bitmap of free/allocated blocks within the group
- A bitmap of allocated inodes within the group
- A table of inode structures that belong to the group
- Depending upon the revision of Ext2 used, some or all block groups may also contain a backup copy of the [Superblock](#) and the [Block Group Descriptor Table](#).

What is an Inode?

An inode is a structure on the disk that represents a file, directory, symbolic link, etc. Inodes do not contain the data of the file / directory / etc. that they represent. Instead, they link to the blocks that actually contain the data. This lets the inodes themselves have a well-defined size which lets them be placed in easily indexed arrays. Each block group has an array of inodes it is responsible for, and conversely every inode within a file system belongs to one of such tables (and one of such block groups).

Superblock

The first step in implementing an Ext2 driver is to find, extract, and parse the superblock. The Superblock contains all information about the layout of the file system and possibly contains other important information like what optional features were used to create the file system. Once you have finished with the Superblock, the next step is to look at the [Block Group Descriptor Table](#)

Locating the Superblock

The Superblock is always located at byte 1024 from the beginning of the volume and is exactly 1024 bytes in length. For example, if the disk uses 512 byte sectors, the Superblock will begin at LBA 2 and will occupy all of sector 2 and 3.

Determining the Number of Block Groups

From the Superblock, extract the size of each block, the total number of inodes, the total number of blocks, the number of blocks per block group, and the number of inodes in each block group. From this information we can infer the number of block groups there are by:

- Rounding up the total number of blocks divided by the number of blocks per block group
- Rounding up the total number of inodes divided by the number of inodes per block group

- Both (and check them against each other)

Base Superblock Fields

These fields are present in all versions of Ext2

Starting Byte	Ending Byte	Size in Bytes	Field Description
0	3	4	Total number of inodes in file system
4	7	4	Total number of blocks in file system
8	11	4	Number of blocks reserved for superuser (see offset 80)
12	15	4	Total number of unallocated blocks
16	19	4	Total number of unallocated inodes
20	23	4	Block number of the block containing the superblock (also the starting block number, NOT always zero.)
24	27	4	\log_2 (block size) - 10. (In other words, the number to shift 1,024 to the left by to obtain the block size)
28	31	4	\log_2 (fragment size) - 10. (In other words, the number to shift 1,024 to the left by to obtain the fragment size)
32	35	4	Number of blocks in each block group
36	39	4	Number of fragments in each block group
40	43	4	Number of inodes in each block group
44	47	4	Last mount time (in POSIX time)
48	51	4	Last written time (in POSIX time)
52	53	2	Number of times the volume has been mounted since its last consistency check (fsck)
54	55	2	Number of mounts allowed before a consistency check (fsck) must be done
56	57	2	Ext2 signature (0xef53), used to help confirm the presence of Ext2 on a volume
58	59	2	File system state (see below)
60	61	2	What to do when an error is detected (see below)
62	63	2	Minor portion of version (combine with Major portion below to construct full version field)
64	67	4	POSIX time of last consistency check (fsck)

68	71	4	Interval (in POSIX time) between forced consistency checks (fsck)
72	75	4	Operating system ID from which the filesystem on this volume was created (see below)
76	79	4	Major portion of version (combine with Minor portion above to construct full version field)
80	81	2	User ID that can use reserved blocks
82	83	2	Group ID that can use reserved blocks

File System States

Value	State Description
1	File system is clean
2	File system has errors

Error Handling Methods

Value	Action to Take
1	Ignore the error (continue on)
2	Remount file system as read-only
3	Kernel panic

Creator Operating System IDs

Value	Operating System
0	Linux
1	GNU HURD
2	MASIX (an operating system developed by Rémy Card, one of the developers of ext2)
3	FreeBSD

4	Other "Lites" (BSD4.4-Lite derivatives such as NetBSD , OpenBSD , XNU/Darwin , etc.)
---	--

Extended Superblock Fields

These fields are only present if Major version (specified in the base superblock fields), is greater than or equal to 1.

Starting Byte	Ending Byte	Size in Bytes	Field Description
84	87	4	First non-reserved inode in file system. (In versions < 1.0, this is fixed as 11)
88	89	2	Size of each inode structure in bytes. (In versions < 1.0, this is fixed as 128)
90	91	2	Block group that this superblock is part of (if backup copy)
92	95	4	Optional features present (features that are not required to read or write, but usually result in a performance increase. see below)
96	99	4	Required features present (features that are required to be supported to read or write. see below)
100	103	4	Features that if not supported, the volume must be mounted read-only see below)
104	119	16	File system ID (what is output by blkid)
120	135	16	Volume name (C-style string: characters terminated by a 0 byte)
136	199	64	Path volume was last mounted to (C-style string: characters terminated by a 0 byte)
200	203	4	Compression algorithms used (see Required features above)
204	204	1	Number of blocks to preallocate for files
205	205	1	Number of blocks to preallocate for directories
206	207	2	(Unused)
208	223	16	Journal ID (same style as the File system ID above)
224	227	4	Journal inode
228	231	4	Journal device
232	235	4	Head of orphan inode list
236	1023	X	(Unused)

Optional Feature Flags

These are optional features for an implementation to support, but offer performance or reliability gains to implementations that do support them.

Flag Value	Description
0x0001	Preallocate some number of (contiguous?) blocks (see byte 205 in the superblock) to a directory when creating a new one (to reduce fragmentation?)
0x0002	AFS server inodes exist
0x0004	File system has a journal (Ext3)
0x0008	Inodes have extended attributes
0x0010	File system can resize itself for larger partitions
0x0020	Directories use hash index

Required Feature Flags

These features if present on a file system are required to be supported by an implementation in order to correctly read from or write to the file system.

Flag Value	Description
0x0001	Compression is used
0x0002	Directory entries contain a type field
0x0004	File system needs to replay its journal
0x0008	File system uses a journal device

Read-Only Feature Flags

These features, if present on a file system, are required in order for an implementation to write to the file system, but are not required to read from the file system.

Flag Value	Description
0x0001	Sparse superblocks and group descriptor tables
0x0002	File system uses a 64-bit file size
0x0004	Directory contents are stored in the form of a Binary Tree

Block Group Descriptor Table

The Block Group Descriptor Table contains a descriptor for each block group within the file system. The number of block groups within the file system, and correspondingly, the number of entries in the Block Group Descriptor Table, is described [above](#). Each descriptor contains information regarding where important data structures for that group are located.

Locating the Block Group Descriptor Table

The table is located in the block immediately following the Superblock. So if the block size (determined from a field in the superblock) is 1024 bytes per block, the Block Group Descriptor Table will begin at block 2. For any other block size, it will begin at block 1. Remember that blocks are numbered starting at 0, and that block numbers don't usually correspond to physical block addresses.

Block Group Descriptor

A Block Group Descriptor contains information regarding where important data structures for that block group are located.

Starting Byte	Ending Byte	Size in Bytes	Field Description
0	3	4	Block address of block usage bitmap
4	7	4	Block address of inode usage bitmap
8	11	4	Starting block address of inode table
12	13	2	Number of unallocated blocks in group
14	15	2	Number of unallocated inodes in group

16	17	2	Number of directories in group
18	31	X	(Unused)

Inodes

Like blocks, each inode has a numerical address. It is extremely important to note that unlike block addresses, **inode addresses start at 1**.

With Ext2 versions prior to Major version 1, inodes 1 to 10 are reserved and should be in an allocated state. Starting with version 1, the first non-reserved inode is indicated via a field in the Superblock. Of the reserved inodes, number 2 subjectively has the most significance as it is used for the root directory.

Inodes have a fixed size of either 128 for version 0 Ext2 file systems, or as dictated by the field in the Superblock for version 1 file systems. All inodes reside in inode tables that belong to block groups. Therefore, looking up an inode is simply a matter of determining which block group it belongs to and indexing that block group's inode table.

Determining which Block Group contains an Inode

From an inode address (remember that they start at 1), we can determine which group the inode is in, by using the formula:

```
block group = (inode - 1) / INODES_PER_GROUP
```

where INODES_PER_GROUP is a field in the Superblock

Finding an inode inside of a Block Group

Once we know which group an inode resides in, we can look up the actual inode by first retrieving that block group's inode table's starting address (see [Block Group Descriptor](#) above). The index of our inode in this block group's inode table can be determined by using the formula:

```
index = (inode - 1) % INODES_PER_GROUP
```

where % denotes the [Modulo operation](#) and INODES_PER_GROUP is a field in the Superblock (the same field which was used to determine which block group the inode belongs to).

Next, we have to determine which block contains our inode. This is achieved from:

```
containing block = (index * INODE_SIZE) / BLOCK_SIZE
```

where INODE_SIZE is either fixed at 128 if VERSION < 1 or defined by a field in the Superblock if VERSION >= 1.0, and BLOCK_SIZE is defined by a field in the Superblock.

Finally, mask and shift as necessary to extract only the inode data from the containing block.

Reading the contents of an inode

Each inode contains 12 direct pointers, one singly indirect pointer, one doubly indirect block pointer, and one triply indirect pointer. The direct space "overflows" into the singly indirect space, which overflows into the doubly indirect space, which overflows into the triply indirect space.

Direct Block Pointers: There are 12 direct block pointers. If valid, the value is non-zero. Each pointer is the block address of a block containing data for this inode.

Singly Indirect Block Pointer: If a file needs more than 12 blocks, a separate block is allocated to store the block addresses of the remaining data blocks needed to store its contents. This separate block is called an indirect block because it adds an extra step (a level of indirection) between an inode and its data. The block addresses stored in the block are all 32-bit, and the capacity of stored addresses in this block is a function of the block size. The address of this indirect block is stored in the inode in the "Singly Indirect Block Pointer" field.

Doubly Indirect Block Pointer: If a file has more blocks than can fit in the 12 direct pointers and the indirect block, a double indirect block is used. A double indirect block is an extension of the indirect block described above only now we have two intermediate blocks between the inode and data blocks. The inode structure has a "Doubly Indirect Block Pointer" field that points to this block if necessary.

Triply Indirect Block Pointer: Lastly, if a file needs still more space, it can use a triple indirect block. Again, this is an extension of the double indirect block. So, a triple indirect block contains addresses of double indirect blocks, which contain addresses of single indirect blocks, which contain address of data blocks. The inode structure has a "Triply Indirect Block Pointer" field that points to this block if present.

[This image from Wikipedia](#) illustrates what is described above pretty well.

Inode Data Structure

Starting Byte	Ending Byte	Size in Bytes	Field Description
0	1	2	Type and Permissions (see below)
2	3	2	User ID
4	7	4	Lower 32 bits of size in bytes
8	11	4	Last Access Time (in POSIX time)
12	15	4	Creation Time (in POSIX time)
16	19	4	Last Modification time (in POSIX time)
20	23	4	Deletion time (in POSIX time)

24	25	2	Group ID
26	27	2	Count of hard links (directory entries) to this inode. When this reaches 0, the data blocks are marked as unallocated.
28	31	4	Count of disk sectors (not Ext2 blocks) in use by this inode, not counting the actual inode structure nor directory entries linking to the inode.
32	35	4	Flags (see below)
36	39	4	Operating System Specific value #1
40	43	4	Direct Block Pointer 0
44	47	4	Direct Block Pointer 1
48	51	4	Direct Block Pointer 2
52	55	4	Direct Block Pointer 3
56	59	4	Direct Block Pointer 4
60	63	4	Direct Block Pointer 5
64	67	4	Direct Block Pointer 6
68	71	4	Direct Block Pointer 7
72	75	4	Direct Block Pointer 8
76	79	4	Direct Block Pointer 9
80	83	4	Direct Block Pointer 10
84	87	4	Direct Block Pointer 11
88	91	4	Singly Indirect Block Pointer (Points to a block that is a list of block pointers to data)
92	95	4	Doubly Indirect Block Pointer (Points to a block that is a list of block pointers to Singly Indirect Blocks)
96	99	4	Triply Indirect Block Pointer (Points to a block that is a list of block pointers to Doubly Indirect Blocks)
100	103	4	Generation number (Primarily used for NFS)
104	107	4	In Ext2 version 0, this field is reserved. In version >= 1, Extended attribute block (File ACL).
108	111	4	In Ext2 version 0, this field is reserved. In version >= 1, Upper 32 bits of file size (if feature bit set) if it's a file, Directory ACL if it's a directory
112	115	4	Block address of fragment
116	127	12	Operating System Specific Value #2

Inode Type and Permissions

The type indicator occupies the top hex digit (bits 15 to 12) of this 16-bit field

Type value in hex	Type Description
0x1000	FIFO
0x2000	Character device
0x4000	Directory
0x6000	Block device
0x8000	Regular file
0xA000	Symbolic link
0xC000	Unix socket

Permissions occupy the bottom 12 bits of this 16-bit field

Permission value in hex	Permission value in octal	Permission Description
0x001	00001	Other—execute permission
0x002	00002	Other—write permission
0x004	00004	Other—read permission
0x008	00010	Group—execute permission
0x010	00020	Group—write permission
0x020	00040	Group—read permission
0x040	00100	User—execute permission
0x080	00200	User—write permission
0x100	00400	User—read permission
0x200	01000	Sticky Bit
0x400	02000	Set group ID

0x800	04000	Set user ID
-------	-------	-------------

Inode Flags

Flag Value	Description
0x00000001	Secure deletion (not used)
0x00000002	Keep a copy of data when deleted (not used)
0x00000004	File compression (not used)
0x00000008	Synchronous updates—new data is written immediately to disk
0x00000010	Immutable file (content cannot be changed)
0x00000020	Append only
0x00000040	File is not included in 'dump' command
0x00000080	Last accessed time should not be updated
...	(Reserved)
0x00010000	Hash indexed directory
0x00020000	AFS directory
0x00040000	Journal file data

OS Specific Value 1

Operating System	How they use this field
Linux	(reserved)

HURD	"translator"?
MASIX	(reserved)

OS Specific Value 2

Operating System	How they use this field			
	Starting Byte	Ending Byte	Size in Bytes	Field Description
Linux	116	116	1	Fragment number
	117	117	1	Fragment size
	118	119	2	(reserved)
	120	121	2	High 16 bits of 32-bit User ID
	122	123	2	High 16 bits of 32-bit Group ID
	124	127	4	(reserved)
HURD	Starting Byte	Ending Byte	Size in Bytes	Field Description
	116	116	1	Fragment number
	117	117	1	Fragment size
	118	119	2	High 16 bits of 32-bit "Type and Permissions" field
	120	121	2	High 16 bits of 32-bit User ID
	122	123	2	High 16 bits of 32-bit Group ID
	124	127	4	User ID of author (if == 0xFFFFFFFF, the normal User ID will be used)
MASIX	Starting Byte	Ending Byte	Size in Bytes	Field Description
	116	116	1	Fragment number

	117	117	1	Fragment size	
	118	127	X	(reserved)	

Directories

Directories are inodes which contain some number of "entries" as their contents. These entries are nothing more than a name/inode pair. For instance the inode corresponding to the root directory might have an entry with the name of "etc" and an inode value of 50. A directory inode stores these entries in a linked-list fashion in its contents blocks.

The root directory is Inode 2.

The total size of a directory entry may be longer than the length of the name would imply (The name may not span to the end of the record), and records have to be aligned to 4-byte boundaries. Directory entries are also not allowed to span multiple blocks on the file-system, so there may be empty space in-between directory entries. Empty space is however not allowed in-between directory entries, so any possible empty space will be used as part of the preceding record by increasing its record length to include the empty space. Empty space may also be equivalently marked by a separate directory entry with an inode number of zero, indicating that directory entry should be skipped.

Directory Entry

Starting Byte	Ending Byte	Size in Bytes	Field Description
0	3	4	Inode
4	5	2	Total size of this entry (Including all subfields)
6	6	1	Name Length least-significant 8 bits
7	7	1	Type indicator (only if the feature bit for "directory entries have file type byte" is set, else this is the most-significant 8 bits of the Name Length)
8	8+N-1	N	Name characters

Directory Entry Type Indicators

Value	Type Description
0	Unknown type

1	Regular file
2	Directory
3	Character device
4	Block device
5	FIFO
6	Socket
7	Symbolic link (soft link)

Quick Summaries

How To Read An Inode

1. Read the Superblock to find the size of each block, the number of blocks per group, number Inodes per group, and the starting block of the first group (Block Group Descriptor Table).
2. Determine which block group the inode belongs to.
3. Read the Block Group Descriptor corresponding to the Block Group which contains the inode to be looked up.
4. From the Block Group Descriptor, extract the location of the block group's inode table.
5. Determine the index of the inode in the inode table.
6. Index the inode table (taking into account non-standard inode size).

Directory entry information and file contents are located within the data blocks that the Inode points to.

How To Read the Root Directory

The root directory's inode is defined to always be 2. Read/parse the contents of inode 2.

Synchronization Primitives

All the techniques presented here are basic building blocks to address the problem of *process synchronization*. E.g. given programs that are running independently from each other on the same machine, how can one ensure some properties about what combination of operations are allowed and what combinations are not.

Among other examples of real-world problems, we're looking for technique that can grant:

- Mutual exclusion of processes: a portion of code cannot be executing by two process simultaneously.
- Rendezvous: one process must not perform one operation (e.g. generating a summary) before other processes have completed their operations.
- Shared readers/Single writer approach of resource locking: many process may be reading a table at the same time, but only one can write at a time and it should prevent readers to access the table until the table has returned to a consistent state.

Note: A good synchronization implementation should not only guarantee correctness, but also fairness (all process have equal chance to get the access) and non-starvation (any waiting process will eventually have the resource).

Contents

[\[hide\]](#)

- [1_Semaphores](#)
- [2_Mutexes](#)
- [3_Spinlocks](#)
- [4_See Also](#)
 - [4.1_Threads](#)
 - [4.2_External Links](#)

Semaphores

Semaphores are one of the oldest and most widely used methods of ensuring [Mutual Exclusion](#) between two or more processes. A semaphore is a special integer variable which is (usually) initialized to 1,

and can only be altered by a pair of functions. Each of these functions, historically called *p* and *v* (from the Dutch words *proberen*, to try, and *verhogen*, to increment), must be an [Atomic operation](#). Each semaphore has an associated queue for processes waiting on the resource it guards.

- The function *p*, also called `wait()` (or `test()`), decrements the value of the semaphore, and if the semaphore is negative, puts the process on the waiting queue until the semaphore is released by the process holding it.
- The function *v*, also called `signal()` (or `release()`), increments the semaphore and, if it is still negative, indicates to the scheduler to wake the next waiting process in the queue.

Note that a general semaphore can do much more than just guaranteeing mutual exclusion. Some FIFO queue (single reader and single writer) can for instance be implemented by using one semaphore counting "how many messages are available" and another one counting "how many free slots are available"

```
Message queue[N];
Semaphore slots=new Semaphore(N);
Semaphore messages=new Semaphore(0);
int last_read=0, last_written=0;

Message get() {
    Message m;
    messages.wait();
    m=queue[last_read]; last_read=(last_read+1)%N;
    slots.signal();
    return m;
}

void put(Message m) {
    slots.wait();
    queue[last_written]=m;
    last_written=(last_written+1)%N;
    messages.signal();
}
```

Mutexes

A variant on this, called a *binary semaphore* uses a boolean value instead of an integer. In that case, *p* tests the value of the semaphore, and if it is true, sets it to false, and if false, waits. The binary *v* function checks the waiting queue, and if it is empty, set the semaphore to true; otherwise, it indicates to the scheduler to wake the next queued process.

In either form, it is important that a process release a semaphore once it has finished using the resource it guards, otherwise the resource could be left inaccessible.

Note that while "semaphore" is a globally-unique semantic items, "mutex" is a fuzzy name and system designers tends to have "their own mutex" which may look more like a spinlock or like a binary semaphore or like a general semaphore...

Spinlocks

[Spinlocks](#) try to address the same problem of [Mutual Exclusion](#), but without relying on a scheduler infrastructure to make the process sleep if the resource is busy. Instead, a spinlock will keep checking the value until it has changed and usually relies on some atomic `test_and_set` instruction on the CPU to perform its task (See [Intel Manuals](#) to see how `xchg` can be used to mimic `test_and_set` virtual operation).

While poorly used spinlocks will lead to severe performance penalty in single-cpu systems, wise use on multi-cpu may achieve higher throughput.

If you are using GCC or Clang, you have access to [the freestanding C11 header stdatomic.h](#), which can be used to implement a spinlock. The following example ensures a "full barrier" and can therefore be used as a simple lock for multiprocessor code. The `atomic_flag_test_and_set` is an atomic instruction that simply sets a flag and returns the prior value of the flag. If the flag was already set, another processor holds the lock, and our processor keeps spinning.

```
#include <stdatomic.h>

void acquire_mutex(atomic_flag* mutex)
{
    while(atomic_flag_test_and_set(mutex))
    {
        __builtin_ia32_pause();
    }
}
```

```

        }
    }

void release_mutex(atomic_flag* mutex)
{
    atomic_flag_clear(mutex);
}

```

Note for IA-32 programmers: If you consider to use spinlocks, be aware that the P4 / Xeon CPUs will falsely detect a possible memory order violation as the spinloop finishes, resulting in a large additional performance penalty. Place a PAUSE instruction into the spinlock to avoid this undesirable behavior. Refer to the [Intel Manuals](#) for more information.

See Also

Threads

- [Userland only Semaphores](#)
- [Spinlocks that disable interrupts](#)
- [SMP compatibility](#)
- [Mutex Implementation](#)
- [Mutexes, Spinlocks and all that jazz](#)
- [Spinlocks & Semaphores](#)

External Links

- [Spinlocks and Read-Write locks](#) - Shows basic code for some kinds of spinlocks and read-write locks

Categories:

- [IPC](#)
- [Synchronization](#)

- [Log in](#)
- [Page](#)
- [Discussion](#)
- [Read](#)
- [View source](#)
- [View history](#)

Navigation

- [Main Page](#)
- [Forums](#)
- [FAQ](#)

- [OS Projects](#)
 - [Random page](#)
- About
- [This site](#)
 - [Joining](#)
 - [Editing help](#)
 - [Recent changes](#)
- Toolbox
- [What links here](#)
 - [Related changes](#)
 - [Special pages](#)
 - [Printable version](#)
 - [Permanent link](#)
- This page was last modified on 1 March 2023, at 23:54.
 - This page has been accessed 106,951 times.
- [Privacy policy](#)
 - [About OSDev Wiki](#)
 - [Disclaimers](#)



Page Frame Allocation

Contents

[\[hide\]](#)

- [1_Physical Memory Allocators](#)
 - [1.1_Bitmap](#)
 - [1.2_Stack/List of pages](#)
 - [1.3_Sized Portion Scheme](#)
 - [1.4_Buddy Allocation System](#)
 - [1.5_Hybrid scheme](#)
 - [1.6_Hybrid scheme #2](#)
- [2_Virtual Addresses Allocator](#)
 - [2.1_Flat List](#)
 - [2.2_Tree-based approach](#)

- [3_See Also](#)
 - [3.1_Articles](#)
 - [3.2_Threads](#)
 - [3.3_External Links](#)

Physical Memory Allocators

These are the algorithms that will provide you with a new page frame when you need it. The client of this algorithm is usually indifferent to which frame is returned, and especially, a request for n frames doesn't need to return contiguous frames (unless you are allocating memory for DMA operations like network packet buffers).

N will be the size of the memory in pages in the following text.

Bitmap

A large array of N/8 bytes is used as a large bit map of the memory usage (that is, bit #i in byte #n define the status of page #n*8+i). Setting the state of a given page is fast ($O(1)$), allocating a page may take time ($O(N)$).

- an `uint32_t` comparison can test up to 32 bits at once and thus speed up allocations
- keeping a pointer to the last allocated bit may improve the performance of the next search (keeping information about the fact all the previous bytes were searched unsuccessfully)

Stack/List of pages

The address of each available physical frame is stored in a [stack](#)-like dynamic structure. Allocating a page is fast ($O(1)$), freeing a page too but checking the state of a page is not practical, unless additional metadata is sorted by physical address.

Sized Portion Scheme

You split each area of, say 16kb into (for example) chunks of 1 8kb, and 2 4kb's. Then you hand out each chunk. By doing this you can find closer fits to exact sizes. That means less waste. So say that you have an area of 32kb



You can even have 1, 2, even 3 or 4 (or more!) types of layouts for each portion. This way you have even more sizes to choose from.

Buddy Allocation System

This is the physical memory allocator of Linux kernel. Note that linux has several buddies depending on whether the memory is suitable for ISA DMA, or is coming from 'high physical memory' or just 'normal'. Each buddy contains k bitmaps, each indicating the availability of 2^i -sized and 2^i aligned blocks of free pages. Usually, linux uses from 4K to 512K blocks.

```

          0   4   8   12  16  20  24  28  32  36
          #####.#. .... #.....#....##...#....####.... real memory
pattern

buddy[0]---> #####.#.xx.#xxxxxxxxx#.xx###.xx#######xxxx 5  free 4K , 5-
byte bitmap
buddy[1]---> # # # . # . x x . # . # # . # # # x x 5  free 8K , 20-
bits map
buddy[2]---> #   #   #   .   #   #   #   #   #   .   2  free 16K, 10-
bits map
buddy[3]---> #           #           #           #           #           0  free 32K, 5-
bits map

```

A buddy for N pages is about twice the size of a bitmap for the same area, but it allows a faster location of collections of pages. The figure above shows a 4-buddy with free pages/blocks denoted as . and used pages/blocks denoted as #. When a block contains at least one used sub-block, it is itself marked as used and sub-blocks that are part of a larger block are also marked as used (x in the figure). Say we want to allocate a 12-K region on this buddy, we'll look up the bitmap of free 16K blocks (which says we have one such starting at page #12 and another starting at page #36). buddy[2]->bit[4] is then set to 'used'. Now we only want 3 pages out of the 4 we got, so the remaining page is returned to the appropriated buddy bitmap (e.g. the single pages map). The resulting buddy is

```

          0   4   8   12  16  20  24  28  32  36
          #####.#. .... #..##....#....##....#....####.... real memory
pattern

buddy[0]---> #####.#.xx.#xx###.xx#.xx###.xx#######xxxx 6  free 4K , 5-
byte bitmap
buddy[1]---> # # # . # . # # . # . # # . # # # x x 5  free 8K , 20-
bits map
buddy[2]---> #   #   #   #   #   #   #   #   #   .   1  free 16K, 10-
bits map
buddy[3]---> #           #           #           #           #           0  free 32K, 5-
bits map

```

Note that initially, only the largest regions are available, so if buddy[0] is apparently empty, we need to check buddy[1], then buddy[2] etc. for a free block to be split.

Hybrid scheme

Allocators may be chained so that (for instance) a [stack](#) only covers the last operations and that the 'bottom' of the stack is committed to a bitmap (for compact storage). If the stack lacks pages, it can scan the bitmap to find some (possibly in a background job).

Hybrid scheme #2

Instead of keeping track of just bits representing pages, or just page numbers on a [stack](#), use a big array of structs to represent the memory. In these page frame structs, store a single link to a next page (pointer-sized) and a 8-16 bit information block indicating its status. Also include the virtual page pointer and the TCB to which the page number belongs. Keep pointers to each type of page, to both the start and the end of their lists. This way, you can easily display information about their content, the amount of pages for each type available, mix types, allow dynamic cleaning threads, do copy-on-write fairly easily and keep clear & concise overviews of the pages. It functions as a reverse page mapping table that lists types of pages too.

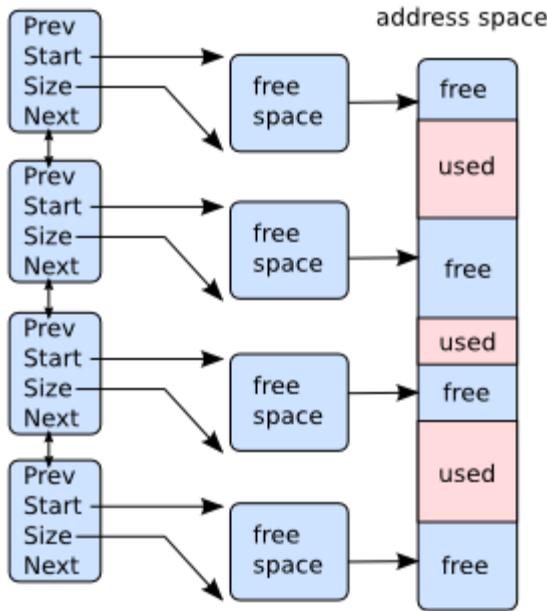
For an example implementation see AtlantisOS 0.0.2 or higher.

Virtual Addresses Allocator

Flat List

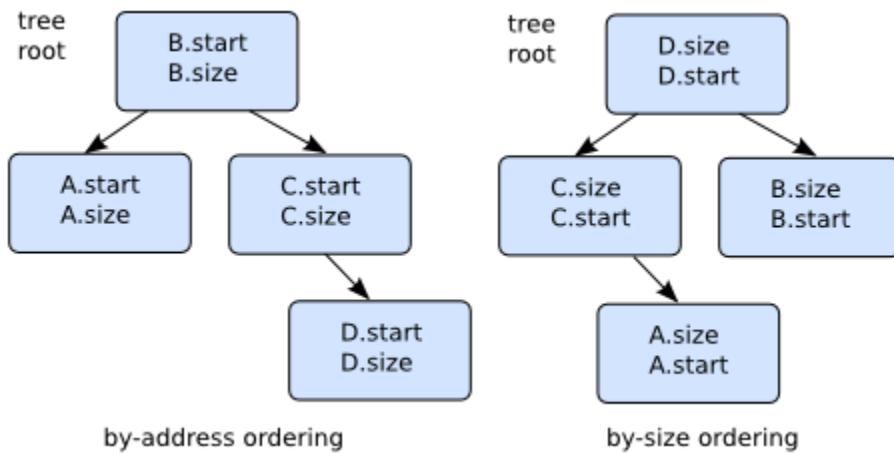
One straightforward way to manage big areas of addresses space is a linked-list (as depicted below). Each "free" region is associated with a descriptor giving its size and its base address. When some space needs to be allocated, the list is scanned for a region being large enough with a "first fit" or "best fit" (or whatever) algorithm. This was e.g. the way memory was managed by MS-DOS. When memory is allocated, the suitable entry is either removed (the whole region is allocated) or resized (only a portion of the region is allocated).

Note that with flat linked-lists, both "is memory at address XXX free" or "where can i get a block of size YYY" questions may require a complete traversal of the list to get answered. If virtual memory gets fragmented and the list gets longer, that may become an issue. "Is memory at address XXX free?" is mainly used to merge two free zone into a new (bigger) one when a block is released, and it is easier to deal with if the list is kept ordered by growing addresses.



Tree-based approach

Since it is frequent that the list is searched for a given address or a given size, it may be interesting to use more efficient data structures. One of them that still keeps the ability of traversing the whole list is the AVL Tree. Each "node" in the AVL tree will describe a memory region and has pointer to the subtree of lower nodes and to the subtree of higher nodes.



While insertion/deletion in such a balanced tree requires more complex operations than linked list manipulation, searching the tree is usually achieved with $O(\log_2(N))$ rather than $O(N)$ for linked lists -- that is, if you have 1000 entries, it requires 1000 iterations to scan the list against 10 iterations to find a given interval in the tree.

Linux has used AVL trees for virtual addresses management for quite a while. Note however that it uses it for regions (like what you find in `/proc/xxxx/maps`), not for a malloc-like interface.

7. Memory Management in xv6 8.1 Basics • xv6 uses 32-bit virtual addresses, resulting in a virtual address space of 4GB. xv6 uses paging to manage its memory allocations. However, xv6 does not do demand paging, so there is no concept of virtual memory. • xv6 uses a page size of 4KB, and a two level page table structure. The CPU register CR3 contains a pointer to the page table of the current running process. The translation from virtual to physical addresses is performed by the MMU as follows. The first 10 bits of a 32-bit virtual address are used to index into a page table directory, which points to a page of the inner page table. The next 10 bits index into the inner page table to locate the page table entry (PTE). The PTE contains a 20-bit physical frame number and flags. Every page table in xv6 has mappings for user pages as well as kernel pages. The part of the page table dealing with kernel pages is the same across all processes. • In the virtual address space of every process, the kernel code and data begin from KERNBASE (2GB in the code), and can go up to a size of PHYSTOP (whose maximum value can be 2GB). This virtual address space of [KERNBASE, KERNBASE+PHYSTOP] is mapped to [0,PHYSTOP] in physical memory. The kernel is mapped into the address space of every process, and the kernel has a mapping for all usable physical memory as well, restricting xv6 to using no more than 2GB of physical memory. Sheets 02 and 18 describe the memory layout of xv6.
- 8.2 Initializing the memory subsystem • The xv6 bootloader loads the kernel code in low physical memory (starting at 1MB, after leaving the first 1MB for use by I/O devices), and starts executing the kernel at entry (line 1040). Initially, there are no page tables or MMU, so virtual addresses must be the same as physical addresses. So the kernel entry code resides in the lower part of the virtual address space, and the CPU generates memory references in the low virtual address space only. The entry code first turns on support for large pages (4MB), and sets up the first page table entrypgdir (lines 1311-1315). The second entry in this page table is easier to follow: it maps [KERNBASE, KERNBASE+4MB] to [0, 4MB], to enable the first 4MB of kernel code in the high virtual address space to run after MMU is turned on. The first entry of this page table maps virtual addresses [0, 4MB] to physical addresses [0,4MB], to enable the entry code that resides in the low virtual address space to run. Once a pointer to this page table is stored in CR3, MMU is turned on, the entry code creates a stack, and jumps to the main function in the kernel's C code (line 1217). The C code is located in high virtual address space, and can run because of the second entry in entrypgdir. So why was the first page table entry required? To enable the few instructions between turning on MMU and jumping to high address space to run correctly. If the entry page table only mapped high virtual addresses, the code that jumps to high virtual addresses of main would itself not run (because it is in low virtual address space). • Remember that once the MMU is turned on, for any memory to be usable, the kernel needs a virtual address and a page table entry to refer to that memory location. When main starts, it is still using entrypgdir which only has page table mappings for the first 4MB of kernel addresses, so only this 4MB is usable. If the kernel wants to use more than this 4MB, it needs to map all of that memory as free pages into its address space, for which it needs a larger page table. So, main first creates some free pages in this 4MB in the function kinit1 (line 3030), which eventually calls the functions freerange (line 3051) and kfree (line 3065). Both these functions together populate a list of free pages for the kernel to start using for various things, including allocating a bigger, nicer page table for itself that addresses the full memory. • The kernel uses the struct run (line 3014) data structure to address a free page. This structure simply stores a pointer to the next free page, and the rest of the page is filled with garbage. That is, the list of free pages are maintained as a linked list,

with the pointer to the next page being stored within the page itself. Pages are added to this list upon initialization, or on freeing them up. Note that the kernel code that allocates and frees pages always returns the virtual address of the page in the kernel address space, and not the actual physical address. The V2P macro is used when one needs the physical address of the page, say to put into the page table entry. • After creating a small list of free pages in the 4MB space, the kernel proceeds to build a bigger page table to map all its address space in the function kvmalloc (line 1857). This function in turn calls setupkvm (line 1837) to setup the kernel page table, and switches to it. The address space mappings that are setup by setupkvm can be found in the structure kmap (lines 1823-1833). kmap contains the mappings for all kernel code, data, and any free memory that the kernel wishes to use, all the way from KERNBASE to KERNBASE+PHYSTOP. Note that 2 the kernel code and data is already residing at the specified physical addresses, but the kernel cannot access it because all of that physical memory has not been mapped into any logical pages or page table entries yet. • The function setupkvm works as follows. For each of the virtual to physical address mappings in kmap, it calls mappages (line 1779). The function mappages walks over the entire virtual address space in 4KB page-sized chunks, and for each such logical page, it locates the PTE using the walkpgdir function (line 1754). walkpgdir simply outputs the translation that the MMU would do. It uses the first 10 bits to index into the page table directory to find the inner page table. If the inner page table does not exist, it requests the kernel for a free page, and initializes the inner page table. Note that the kernel has a small pool of free pages setup by kinit1 in the first 4MB address space—these free pages are used to construct the kernel’s page table. Once walkpgdir returns the PTE, mappages sets up the appropriate mapping using the physical address it has. (Sheet 08 has the various macros that are useful in getting the index into page tables from the virtual address.) • After the kernel page table kpgdir is setup this way (line 1859), the kernel switches to this page table by storing its address in the CR3 register in switchkvm (line 1866). From this point onwards, the kernel can freely address and use its entire address space from KERNBASE to KERNBASE+PHYSTOP. • Let’s return back to main in line 1220. The kernel now proceeds to do various initializations. It also gathers many more free pages into its free page list using kinit2, given that it can now address and access a larger piece of memory. At this point, the entire physical memory at the disposal of the kernel [0, PHYSTOP] is mapped by kpgdir into the virtual address space [KERNBASE, KERNBASE+PHYSTOP], so all memory can be addressed by virtual addresses in the kernel address space and used for the operation of the system. This memory consists of the kernel code/data that is currently executing on the CPU, and a whole lot of free pages in the kernel’s free page list. Now, the kernel is all set to start user processes, starting with the init process. 3.8.3 Creating user processes • The function userinit (line 2502) creates the first user process. We will examine the memory management in this function. The kernel page table of this process is created using setupkvm as always. For the user part of the memory, the function inituvm (line 1903) allocates one physical page of memory, copies the init executable into that memory, and sets up a page table entry for the first page of the user virtual address space. When the init process runs, it executes the init executable (sheet 83), whose main function forks a shell and starts listening to the user. Thus, in the case of init, setting up the user part of the memory was straightforward, as the executable fit into one page. • All other user processes are created by the fork system call. In fork (line 2554), once a child process is allocated, its memory image is setup as a complete copy of the parent’s memory image by a call to copyuvvm (line 2053). This function walks through the entire address space of the parent in page-sized chunks, gets the physical address of every page of the parent using a call to walkpgdir, allocates a new physical page

for the child using kalloc, copies the contents of the parent's page into the child's page, adds an entry to the child's page table using mappages, and returns the child's page table. At this point, the entire memory of the parent has been cloned for the child, and the child's new page table points to its newly allocated physical memory. • If the child wants to execute a different executable from the parent, it calls exec right after fork. For example, the init process forks a child and execs the shell in the child. The exec system call copies the binary of an executable from the disk to memory and sets up the user part of the address space and its page tables. In fact, after the initial kernel is loaded into memory from disk, all subsequent executables are read from disk into memory via the exec system call alone. • Exec (line 6310) first reads the ELF header of the executable from the disk and checks that it is well formed. It then initializes a page table, and sets up the kernel mappings in the new page table via a call to setupkvm (line 6334). Then, it proceeds to build the user part of the memory image via calls to allocuvm (line 6346) and loaduvm (line 6348) for each segment of the binary executable. allocuvm (line 1953) allocates physical pages from the kernel's free pool via calls to kalloc, and sets up page table entries. loaduvm (line 1918) reads the memory executable from disk into the allotted page using the readi function. After the end of the loop of calling these two functions for each segment, the program executable has been loaded into memory, and page table entries setup to point to it. However, exec hasn't switched to this new page table yet, so it is still executing in the old memory image. • Next, exec goes on to build the rest of its new memory image. For example, it allocates a user stack page, and an extra page as a guard after the stack. The guard page has no physical memory frame allocated to it, so any access beyond the stack into the guard page will cause a page fault. Then, the arguments to exec are pushed onto the user stack, so that the exec binary can access them when it starts. 4 • It is important to note that exec does not replace/reallocate the kernel stack. The exec system call only replaces the user part of the memory. And if you think about it, there is no way the process can replace the kernel stack, because the process is executing in kernel mode on the kernel stack itself, and has important information like the trap frame stored on it. However, it does make small changes to the trap frame on the kernel stack, as described below. • Now, a process that makes the exec system call has moved into kernel mode to service the software interrupt of the system call. Normally, when the process moves back to user mode again (by popping the trap frame on the kernel stack), it is expected to return to the instruction after the system call. However, in the case of exec, the process doesn't have to return to the instruction after exec when it gets the CPU next, but instead must start executing in the new memory image containing the binary file it just loaded from disk. So, the code in exec changes the return address in the trap frame to point to the entry address of the binary (line 6392). Finally, once all these operations succeed, exec switches page tables to start using the new memory image, and frees up all the memory pointed at by the old page table. At this point, the process that called exec can start executing on the new memory image. Note that exec waits until the end to do this switch, because if anything went wrong in the system call, exec returns to the old process image and prints out an error. 5

Hello, world.

In this document, we'll attempt to explain the actual code of our beloved xv6.

Not all of it, but some of the interesting parts.

God have mercy on us.

`1217 main(void)`

The entry point of the kernel.

Sets up kernel stuff and starts running the first process.

****1219**:** set up first bunch of pages, for kernel to work with (minimal, because old hardware has little memory)

****1220**:** set up all kernel pages

****1223**:** set up segment tables and per-CPU data

****1238**:** set up the rest of the pages for general use (because until now we had just minimal, because other CPUs might not handle high addresses)

****1239**:** set up the First Process

****1241**:** run the scheduler (and the First Process)

`2764 struct run`

Represents a memory page.

A `run` points to the next available page/`run` (which is actually the *previous* page, because the first available is the last in the memory).

`2768 struct kmem`

Points to the head of a list of free (that is, available) pages of memory.

`2780 kinit1(void *vstart, void *vend)`

Frees a bunch of pages.

Also does some locking thing (I'll elaborate once we actually learn this stuff).

Used only when kernel starts up.

Called by [`main`](#1217-mainvoid).

`2788 kinit2(void *vstart, void *vend)`

Frees a bunch of pages.

Also does some locking thing (I'll elaborate once we actually learn this stuff).

Used only when kernel starts up.

Called by [`main`](#1217-mainvoid).

`2801 freerange(void *vstart, void *vend)`

Frees a bunch of pages.

2804: use PGROUNDUP because `kinit1` in called to start where the kernel finished (which is not likely to end **exactly** at a page end).

Called by:

* [`kinit1`](#2780-kinit1void-vstart-void-vend)

* [`kinit2`](#2788-kinit2void-vstart-void-vend)

`2815 kfree(char *v)`

Frees the (single!) page that `v` points at.

2819-2820: address validity checks

2823: fill page with 1s, to help in case of bugs

2827-2829: insert our page into the beginning of `kmem` (a linked list with all available pages)

Called by:

* `deallocuvvm`

- * `freevm`
- * `fork`
- * `wait`
- * [`freerange`](#2801-freerangevoid-vstart-void-vend)
- * `pipealloc`
- * `pipeclose`

`2838 kalloc(void)`

Removes a page from `kmem`, and returns its (virtual!) address.

****2844-2846**:** remove first free page from `kmem`

****2849**:** return address

Called by:

- * `startothers`
- * [`walkpgdir`](#1654-walkpgdirpde_t-pgdir-const-void-va-int-alloc)
- * [`setupkvm`](#1737-setupkvmvoid)
- * [`inituvvm`](#1803-inituvmpde_t-pgdir-char-init-uint-sz)

- * `allocuvm`
- * `copyuvm`
- * [`allocproc`](#2205-allocprocvoid)

- * `pipealloc`

`1757 kvmalloc(void)`

Builds new page table and makes `%CR3` point to it.

1759: make table and get its address

1760: make `%CR3` point to returned address

Called by [`main`](#1217-mainvoid).

`1728 kmap[]`

Contains data of how kernel pages should look.

Used by `setupkvm` for mapping.

Column 0: Virtual addresses

Column 1: Physical addresses start

Column 2: Physical addresses end

****Column 3**: Pages permissions**

****Note**:** The `data` variable (used in lines 1730-1731) is where the kernel's data start (*data* is plural, by the way).

We do not know during compilation where this will be.

`1737 setupkvm(void)`

Sets up kernel virtual pages.

****Returns**** page table address if successful, 0 if not.

****1742**:** create outer `pgdir` page table

****1744**:** clear `pgdir`

****1745-1746**:** make sure we didn't map illegal address

****1747-1750**:** loop over `kmap` and map pages using `mappages`

Called by:

* [`kvmalloc`](#1757-kvmallocvoid)

* `copyuvvm`

* [`userinit`](#2252-userinitvoid)

* [`exec`](#5910-execchar-path-char-argv)

`1679 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)`

Creates translations from *`va`* (virtual address) to *`pa`* (physical address) in existing page table `pgdir`.

Returns 0 if successful, -1 if not.

1684: get starting address

1685: get ending address (which is starting address if `size`=1)

1686: for each page...

1687: get i1 row address (using `walkpgdir`)

1689: make sure i1 row not used already

1691: write *`pa`* in i1 and mark as valid, with required permissions

Called by:

* [`setupkvm`](#1737-setupkvmvoid)

* [`inituvm`](#1803-inituvmpde_t-pgdir-char-init-uint-sz)

* `allocuvm`

* `copyuvm`

```
### `1654 walkpgdir(pde_t *pgdir, const void *va, int alloc)'
```

Looks at virtual address `va`,
finds where it should be mapped to according to page table `pgdir`,
and returns the **virtual** address of the the *index* i1.
Returns address if successful, 0 if not.

If there is no mapping, then:

```
if `alloc`=1, mapping is created (and address is returned);  
if `alloc`=0, return 0
```

Some constants and macros used here:

PDX - zeroes offset bits

PTX - uh... some more bit manipulations

PTE_P - "valid" bit

PTE_W - "can write" bit

PTE_U - "available in usermode" bit

1659: get i0 index address

1660: check if i0 is valid

1661: put address of subtable in `pgtab`

1663: (i0 not valid) create new subtable, `pgtab`

1666: (i0 not valid) clear `pgtab` rows

1670: (i0 not valid) point i0 to `pgtab` and mark **i0** as valid, writable, usermodeable.

****1672**:** return address of appropriate row in `pgtab`

Called by:

* [`mappages`](#1679-mappagespde_t-pgdir-void-va-uint-size-uint-pa-int-perm)

* `loaduvm`

* `deallocuvm`

* `clearpteu`

* `copyuvm`

* `uva2ka`

`1616 seginit(void)`

Sets segmentation table so that it doesn't get in the way, for each CPU.

Adds extra row to each segmentation table in order to guard CPU-specific data, makes `%gs` register point to it, and makes `proc` and `cpu` actually point to `%gs`.

****1624-1628**:** set up regular rows in segmentation table

****1631-1634**:** set up special row and `%gs` register

****1637-1638**:** set up initial `proc` and `cpu` data

Called by [`main`](#1217-mainvoid).

```
### `2252 userinit(void)`
```

Creates and sets up The First Process.

2255: data and size of First Process code. Filled by script during compilation.

2257: allocate `proc` structure and set up data on kernel stack

2258: save proc in `initproc`, so we'll always remember who the First Process is

2259-2260: create page table with kernel addresses mapped

2261: allocate free page, copy process code to page, map user addresses in page table

2262-2275: fix data on kernel-stack, *as if* it were stored there because of an interrupt

- **2264**: make sure we'll be in usermode when process starts

- **2270**: make sure process will start in address 0

Called by [`main`](#1217-mainvoid).

```
### `2205 allocproc(void)`
```

Allocates `proc` structure and sets up data on kernel stack.

Returns proc if succeeds, 0 if not.

****2211-2213**:** find first unused `proc` structure

****2217-2219**:** set to EMBRYO state and give `pid`

****2223-2226**:** allocate and assign kernel stack for process

****2227**:** set stack pointer to bottom of stack (stack bottom is highest address in stack)

****2230-2231**:** make room on stack for `trapframe`

****2235-2239**:** make room on stack for `context`

****2240-2241**:** set `context`, setting `context.eip` to function `forkret`

Called by:

* [`userinit`](#2252-userinitvoid)

* `fork`

`1803 inituvm(pde_t *pgdir, char *init, uint sz)`

Allocates and maps single page (4KB), and fills it with with program code.

****1807-1808**:** make sure entire program code fits in single page

****1809**:** allocate page

****1810**: clear page**

****1811**: map pages in page table `pgdir` (using `v2p`, because we know what memory this is, because this is the First Process, which the kernel always creates on startup)**

****1812**: copy the code**

Called by [`userinit`](#2252-userinitvoid).

`2458 scheduler(void)`

Loops over all processes (in each CPU), finds a runnable process, and runs it.

Loops for ever and ever.

****2467**: lock process table, to prevent multiple CPUs from grabbing same process**

****2468-2470**: find first available process**

****2475**: set *per-CPU* variable `proc` to point to current running process**

****2476**: set up process's kernel stack, and switch to its page table**

****2477**: mark process as running**

****2478**: save current registers (including where to continue on scheduler) and load process's registers, handing the stage over to the process**

(NOTE: the running process is responsible to release the process table lock (to enable interrupts) and later re-lock it.)

****2479**:** now that process is switching back to scheduler, switch back to kernel registers and Page Table

****2483**:** set per-CPU variable `proc` back to 0

****2485**:** release process table, allowing other CPUs to grab processes (just in case

Called by `mpmain`.

`1773 switchuvvm(struct proc *p)`

Perpares kernel-stack of process (that is, makes `%tr` register indirectly point to it), and loads process's Page Table to `%cr3`.

****1776-1779**:** set up `%tr` register and `SEG_TSS` section in GDT end up magically (don't ask how) referring us to top of process's kernel stack

Called by:

* `growproc`

* [`scheduler`](#2458-schedulervoid)

* [`exec`](#5910-execchar-path-char-argv)

`2708 swtch(struct context **old, struct context *new)`

Saves current register context in `old`, then loads the register context from `new`.

Basically gives control to new process.

2709: set `%eax` to contain address of `old` context

2710: set `%edx` to contain `new` context

2713-2716: push `%ebp`, `%ebx`, `%esi`, `%edi` onto current stack (which happens to be `old` stack)

2719: copy value of `%esp` to address held in `%eax`, which is the `old` stack address (see line **2709**)

2720: set current stack pointer (`%esp`) to value of `%edx`, which is the `new` stack address (see line **2710**)

2723-2726: pop `%edi`, `%esi`, `%ebx`, `%ebp` from `new` stack onto the actual stack

Called by:

* [`scheduler`](#2458-schedulervoid)

* [`sched`](#2504-schedvoid)

`2503 sched(void)`

Switches back `scheduler` to return from a process that had enough running.

Called by:

* `exit`

* [`yield`](#2522-yieldvoid)

* `sleep`

`2522 yield(void)`

Gives up the CPU from a running process.

2524: re-lock the process table for scheduler

2525: make self as not running

2526: switch back to scheduler

2527: after scheduler re-ran process, re-elease process table to enable interrupts

Called by `trap`.

`2553 sleep(void *chan, struct spinlock *lk)`

Makes process sleep until `chan` event occurs.

2568: lock process table in order to set sleeping state safely

2569: now that process table is locked, release `lk`

****2573-2574**:** set up sleeping state (and alarm clock)

****2575**:** return to scheduler until the event manager marks process as runnable

****2578**:** clean up

****2582**:** release process table

****2583**:** lock `lk` once again

`1555 pushcli(void)`

Saves state of `%eflags` register's `IF` bit (that is, the current state of "listen to interrupts?" bit), increments the "how many times did we choose to ignore interrupts" counter, and clears the "listen to interrupts" bit.

****1561-1562**:** save initial state of bit in `interna` var (FL_IF is the location of our bit)

Called by:

* [`acquire`](#1474-acquirestuct-spinlock-lk)

* [`switchuvvm`](#1773-switchuvvmstruct-proc-p)

`1566 popcli(void)`

Decrement the "how many times did we choose to ignore interrupts" counter,

and if it reaches 0 then sets the "listen to interrupts" bit to whatever it was before the very first `pushcli` was ever called.

1572-1573: only set our bit if `interna` (initial bit value) was set

Called by:

* [`release`](#1502-releasestruct-spinlock-lk)

* [`switchuvvm`](#1773-switchuvmstruct-proc-p)

`1474 acquire(struct spinlock *lk)`

Loops over spinlock until lock is acquired (exclusively by current CPU).

1476: disable interrupts (which will be enabled in `release`)

1483-1484: loop over lock until it has a zero (and write "1" in it)

`1502 release(struct spinlock *lk)`

Releases spinlock from being held by current CPU.

1519: write "0" in lock

1521: re-enable interrupts (which were disabled in `acquire`)

`3004 alltraps`

Catches and prepares all interrupts for `trap`.

Pushes register data on stack, calls `trap` with the `stack` as a `trapframe` argument, pops register data from stack, and finally calls `iret`.

3005-3010: store registers and build `trapframe`

3013-3018: set up data and per-CPU segments (?)

3021-3023: call `trap`, using stack as argument

- **3023**: skip over top of frame (`%esp` address) without popping it into anything

3027-3034: pop registers and call `iret`

- **3033**: skip over data and per-CPU segments (without popping it into anything)

`3101 trap(struct trapframe *tf)`

Handles all interrupts.

3103-3111: handle system call and return

- **3106**: save `tf` to `proc->tf`, so that we don't need to start passing it around during `syscall`

3113-3143: handle controller interrupts (keyboard, timer, etc.)

****3150-3163**: handle unexpected interrupt**

- ****3151**: check if there is no current process (i.e. during `scheduler`) or if we were in kernel-mode during interrupt**
- ****3158-3162**: print error and kill buggy process**

****3168-3178**: finish up non-system calls**

- ****3168-3169**: if process is user-process, and killed, and is not in the middle of a system-call, exit (and don't return to `alltraps`)**
- ****3173-3174**: if process is running, and we had a timer-interrupt, and the process ran for long enough already, yield CPU back to `scheduler`**
- ****3177-3178**: after previous yield, if process was killed (and not in middle of system-call), exit**

Called by [`alltraps`](#3004-alltraps)

`3067 tvinit(void)`

Initializes the IDT table.

Called by [`main`](#1217-mainvoid).

`3079 idtinit(void)`

Makes `%IDTR` point at existing IDT table.

Called by `mpmain`

`3375 syscall(void)`

Handles system-calls from user-code.

3379: get system-call number

3380: make sure number is valid

3381: execute system-call and store return value in process's trapframe's `eax` field (which will afterward be popped to `%eax`)

3382-3385: if bad system-call number, print error and store -1 (error) in trapframe's `eax` field

Called by [`trap`](#3101-trapstruct-trapframe-tf).

`3465 sys_sleep(void)`

System call for sleeping a certain amount of ticks.

3470: get number of required ticks (and validate that the user supplied a valid address as an argument)

3472: lock tickslock

3473: store current (initial) value of `ticks` in `ticks0`

3474-3480: while required number of ticks didn't pass, loop

- ***3479**: wait for event #`&ticks` (we don't really need the lock in this case, but usually in `sleep` call we need to lock because other cases we `sleep` for disc or something else where we don't want two process's to grab the resource simultaneously)

Can be called by user code.

```
### `2614 wakeup(void *chan)`
```

Locks process table, finds all sleeping processes that are waiting for `chan`, makes them runnable, and unlocks process table.

Called by a lot of different functions.

```
### `2603 wakeup1(void *chan)`
```

Finds all sleeping processes that are waiting for `chan`, and makes them runnable.

Called by:

* `exit`

* [`wakeup`](#2614-wakeupvoid-chan)

```
### `3295 argint(int n, int *ip)`
```

Gets the `n`th *integer* argument pushed onto the user-stack by user code before user asked for system-call.

`3267 fetchint(uint addr, int *ip)`

Gets the integer argument in address `addr`, and sets it in `ip`.

Returns 0 if successful, -1 otherwise.

3267: Validates that neither "edge" of integer-containing address space goes beyond valid proc memory.

Called by:

* [`argint`](#3295-argintint-n-int-ip)

* `sys_exec`

`2304 fork(void)`

Creates new process, copying lots from its parent, and set stack as if returning from a system-call.

2310: allocate new proc. Proc now contains kernel-stack, context (with trapret address), trapframe and pid

2314-2319: copy memory

- **2314**: copy memory

- **2315-2318**: if error, free kernel stack

2320: copy `sz`

2321: set parent

2322: *copy* trapframe struct to new kernel-stack

2325: clear `%eax` so `fork` will return 0 for child process

2327-2330: do file stuff

2333: make new proc RUNNABLE (at this point, `scheduler` can grab child process before parent)

1953 copyuv(pde_t *pgdir, uint sz)`

Creates copy of parent memory for child process.

Returns address of new page table.

1960: set up kernel virtual pages

1962: loop over all pages:

- **1963**: get address+flags of parent process's page

- **1965**: make sure page is actually present

- **1967**: get physical address of parent process's page

- ****1968**: allocate new page**
- ****1970**: copy memory from old physical page to new physical page**
- ****1971**: add-n-map new page to new page table**

****1974**: if no errors, return address of new page table**

****1977-1978**: if there were any errors, release all memory and return 0**

Called by [`fork`] (#2304-forkvoid)

`1910 freevm(pde_t *pgdir)`

Frees a page table and all the physical memory pages (in its user part).

****1916**: free user-mode pages**

****1917-1921**: free internal page tables**

****1923**: free external page table**

Called by:

* [`copyuvm`] (#1953-copyuvmpde_t-pgdir-uint-sz)

* `wait`

* [`exec`](#5910-execchar-path-char-argv)

`1882 deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)`

Deallocates user pages to bring the process size from `oldsz` to `newsz`.

1891: loop over extra pages we want to deallocate:

- **1892**: get virtual address of internal table entry that points to current page-to-remove

- **1896**: get physical address of page

- **1899**: get virtual address of page

- **1900**: free page

- **1901**: mark internal page table entry as "pointing at no page"

Called by:

* [`allocuvm`](#1853-allocuvmpde_t-pgdir-uint-oldszi-uint-newszi)

* [`freevmpde_t-pgdir`](#1910-freevmpde_t-pgdir)

* `growproc`

`1853 allocuvm(pde_t *pgdir, uint oldsz, uint newsz)`

Allocate page tables and physical memory to grow process from `oldsz` to `newsz`. returns `newsz` if succeeded, 0 otherwise.

Called by:

- * `growproc`
- * [`exec`](#5910-execchar-path-char-argv)

`5910 exec(char *path, char **argv)`

Replaces current process with new one.

****5920-5949**: load da code**

- ****5920**: open file**

- ****5926**: read ELF header**

- ****5936-5947**: loop over sections in `proghdr`:**

- ****5937**: read section from file**

- ****5943**: allocate memory**

- ****5945**: load code and data**

- ****5948**: close file**

****5952-5956**: allocate user-stack and guard page**

- ****5952**: round up address in order to add stack and guard-page at new page**

- ****5953**: allocate two pages, for stack and guard page**

- ****5955**: remove user-mode bit from guard page**

- ****5956**: set stack pointer to point to stack page**

****5959-5967**: push arguments to new process user-stack**

- ****5959**: loop over arguments:**

- ****5962**: move new process stack pointer so there's room for current argument**

- ****5963**: copy argument to actual new process user-stack**

- ****5965**: make `argv` vector entry (which is still in temporary `ustack` variable!) point to current argument that we just pushed to stack**

- ****5967**: put 0 in last entry of `argv` vector (which is still in temporary `ustack` variable!), as is expected by convention**

- ****5969**: put -1 as return address in appropriate spot in temporary `ustack` variable**

- ****5970**: put `argc` in appropriate spot in temporary `ustack` variable**

- ****5971**: put address of where `argv` will be in new user-stack (but isn't there yet) in appropriate spot in temporary `ustack` variable**

- ****5974****: now that all arguments are copied to new user-stack, and we know where to place return address & `argc` & `argv`, copy `ustack` variable to new process user-stack

****5978-5981****: copy new process name

- ****5978-5980****: get part of the name after all the slashes

****5984-5991****: switch address space and fix trapframe

- ****5984****: save old page table for freeing later

- ****5985****: give proc new page table!

- ****5986****: give proc new size

- ****5987****: set trapframe's `eip` to new process entry point

- ****5988****: set trapframe's `sp` to new process user-stack

- ****5989****: make `%CR3` point to new page table (which doesn't harm our current running!)

- ****5990****: free old page table

- ****5991****: return to syscall, which returns to alptraps, which returns to popall, which returns to iret, which pops a bunch of values to actual registers, which include `%eip`, which makes us actually continue with the new process

`1818 loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)`

Loads a `sz`-sized program section from `ip` file (in `offset` offset) to address `addr` (which is already mapped in `pgdir`).

Returns 0 if successful, -1 otherwise.

****1825-1835**:** loop page by page (because the pages mapped in `pgdir` are scattered (and we can't rely on the Paging Unit to handle this, because `pgdir` is not our current page table))

- ****1826**:** validate that page is already mapped

- ****1828**:** get physical address

- ****1829-1832**:** take care of case where what's left to read is less than page

- ****1833**:** copy code-n-data

Called by [`exec`](#5910-execchar-path-char-argv)

`2002 uva2ka(pde_t *pgdir, char *uva)`

Returns the kernel virtual address of a user virtual address.

Only works for addresses of pages (and not for middle of page).

****2006**:** get `pte` entry

****2011**:** get offset, do `p2v` to it, and return result

`2018 copyout(pde_t *pgdir, uint va, void *p, uint len)`

Copies `len` bytes from `p` address to `pgdir->va` address.

2029: get `va` offset within its page

2032: copy data

`2354 exit(void)`

Exists current process.

2359-2360: make sure we're not the First Process

2363-2368: close all files opened by user-code

2370-2371: close current working-directory

2376: let parent know we're exiting (in case parent called `wait` for child to exit) (The `chan` the parent is waiting for is the parent's `proc` address)

2379-2385: pass abandoned children to the First Process

- **2382-2383**: if child did `exit`, let the First Process know

2388: become zombie

2389: awaken the `scheduler`

`5225 filealloc(void)`

Finds the first free slot in the global file table, and returns its **address**.

If there are none free, returns 0.

`5438 fdalloc(struct file *f)`

Finds the first free slot in the process's file table, points it to `f`, and returns the index (AKA the file descriptor).

If no room, returns -1.

`5252 filedup(struct file *f)`

Increments the reference count of `f`.

(Used as part of the file duplication process)

`5451 sys_dup(void)`

Duplicates proc's reference to file.

****5458**: actual duplication**

****5460**: increment ref count**

```
### `5419 argfd(int n, int *pf, struct file **pf)`
```

Gets the `n`th argument sent to the system call, as a file descriptor.

Returns descriptor and the struct file it points to.

(The only reason we need the file descriptor `pf` is in case we're closing the file and need to make `ofile` point to null.)

```
### `5315 fileread(struct file *f, char *addr, int n)`
```

Reads from `f` to `addr`.

****5319**:** make sure can read

****5321-5322**:** handle case when file is pipe

****5323-5329**:** handle case when file is inode:

- ****5324**:** lock the inode (because we must)

- ****5325**:** read

- ****5326**:** update offset

- ****5327**:** unlock the inode

Called by `sys_read`.

```
### `5352 fwrite(struct file *f, char *addr, int n)`
```

Writes from `addr` to `f`.

****5358-5359**:** handle case when file is pipe

****5360-5386**:** handle case when file is inode:

- ****5367-5372**:** break up data to manageable chunks (for transactions)

- ****5374-5379**:** write chunk:

- ****5374**:** begin transaction

- ****5376-5377**:** write

- ****5379**:** end transaction

```
### `5264 fclose(struct file *)`
```

Decrements file reference count.

When no references left, actually close file.

****5271-5273**:** decrement ref count

****5275**:** keep backup of struct file, because we're about to release the lock on the file table (and anything can happen after *that*)

****5278**:** release file table because closing file on disk can take a long time

****5280-5281**:** handle case when file is pipe (needs to happen once for `read` and once for `write` descriptors of pipe for it to actually close)

****5282-5286**:** handle case when file is inode (using transaction because this can require writing on device)

`5851 sys_pipe(void)`

Allocates two files (read pipe and write pipe).

Expects a vector with two entries (from the input), in order to return the descriptors in.

****5859**:** allocate pipe (creates 2 file structs)

****5862**:** allocate slots in `ftable`, and point them to the pipe

****5863-5867**:** remove files in case of failure

****5869-5870**:** put descriptors in vector, for user code

`5701 sys_open(void)`

Opens or creates inode.

****5710-5715**:** create inode (on disk!) - can only create file (not directory)

****5716-5724**:** open inode

- ****5720-5722****: if tried opening directory in *write* mode, close-n-error

****5726****: create struct for inode, add it to `ftable`

****5734-5738****: set file struct data

`5011 dirlookup(struct inode *dp, char *name, uint *poff)`

Finds an inode *under* `dp` with name that's equal to `name`.
(`poff` is an optional pointer to the offset of the found inode.)

****5020****: read single `struct dirent` in `dp`

****5022****: check whether `inum` of current `dirent` is active

****5024****: check whether name of current `dirent` equals `name`

If we found the droid we're looking for:

****5026-5027****: store offset in `poff` (if we supplied the optional pointer)

****5028-5029****: get actual inode

...And if we did not find it:

****5033****: return 0

```
### `5115 skipelem(char *path, char *name)`
```

A helper function that helps us take apart "long/path/names".

****Returns**** the value of `path` ***without*** the first part, and sets `name` to equal the chopped off head.

(Ignores the first instance of "/" in the path: "***/**a/b/c" acts the same as "a/b/c".)

```
### `5189 namei(char *path)`
```

Returns the inode with the matching path.

(If `path` begins with "/", looks in root inode. Else, looks in current working directory inode.)

****5192**:** ask `namex` to do the work

```
### `5196 nameiparent(char *path, char *name)`
```

Returns the inode with the matching path ***without the last part***.

(If `path` begins with "/", looks in root inode. Else, looks in current working directory inode.)

****5198**:** ask `namex` to do the work

```
### `5154 namex(char *path, int nameiparent, char *name)`
```

Returns the inode with the matching path.

(If `nameiparent` is 0, finds actual inode. Else, ignores last part (for case when we want to *create* a *new* inode).)

****5158-5161**:** check if we need to look in root or current working directory

****5163-5180**:** loop over parts in path:

- ****5164**:** `ilock` current inode, because we must

- ****5165-5168**:** make sure we're looking at a *directory*

- ****5169-5173**:** if we're looking for all but last (and found it), return it

- ****5174-5177**:** try to get the next part of the path under the current inode

- ****5179**:** set current inode to be the found path part inode

****5181-5184**:** if looking for all but last (and haven't found id before), return 0

****5185**:** happily return our found inode

`5052 dirlink(struct inode *dp, char *name, uint inum)`

Writes a new directory entry (name, inum) into the inode pointed at by `dp` .

****5059-5062**:** make sure name doesn't already exist in inode

****5065-5070**:** find empty slot in inode

5072: prepare to write name

5073: prepare to write inum

5074: actually write the data (if we reached the end of the inode, `writei` will increase its size)

`5657 create(char *path, short type, short major, short minor)`

Creates and returns inode supplied in `path` (expecting `path` to exist up to its last part, which is the inode we're creating).

If inode already exists, opens it.

5663: get parent of requested inode

5667-5674: check if inode exists (and is a file!)

If so, return it.

If exists but is not file, return 0.

5676: get an actual inode from the disk

5679: lock the new inode (because we must)

5680-5682: set some values. nlink is 1, because parent is linked to new inode

5683: updates new values in the disk

5689: add the two links every directory has:

- "." self

- ".." parent

5693: add the new inode to parent

`4654 iget(uint dev, uint inum)`

Opens (and returns) inode.

4661-4670: loop over `icache`, looking if inode was already loaded once

- **4663-4667**: check if the inode is already in the cache

- **4668-4669**: find first empty slot, in case we'll need to load inode to it

4673-4674: if cache is full (and inode not there), panic (although we could have also gone to sleep till there's room)

4676-4680: load some of the inode data to `icache`. Valid bit is set to 0, because we haven't yet read the data from the disk

`4689 idup(struct inode *ip)`

Increments reference count of inode (and returns it).

4691: lock **`icache`** before use, because we access inode field

```
### `4756 iput(struct inode *ip)`
```

Decrements reference count of inode, and closes it on the disk if this it will no longer be referenced at all.

****4759**:** check if:

1. We're about to close the inode's last reference
2. We're closing a valid inode
3. There are no more names of / links to the inode (so it must be destroyed)

If so:

- ****4761**:** mark inode as busy (since there's still one reference!), just in case

- ****4765**:** actually delete inode on disk

- ****4767**:** update the inode (????)

- ****4770**:** do something that isn't really needed

****4772**:** finally decrement the ref count

```
### `4603 ialloc(uint dev, short type)`
```

Allocates a new inode ***on the disk***, and then ***in the memory***.

****4610**:** read super-block from disk.

****4612**:** loop over all inodes on disk:

- ****4613**:** read block of current inode

- ****4614**:** calculate pointer to current inode (using casting so that `+ inum%IPB` will add the correct size)

- ****4615**:** check if inode is free. If so:

- ****4616**:** clear inode data

- ****4617**:** set type

- ****4618**:** re-write entire block, marking new inode as used (because we set type)

- ****4619**:** close current block

- ****4620**:** allocate inode in memory and return the *memory* inode

- ****4622**:** close current block

****4624**:** no more inodes on disk?! Panic!

`4629 iupdate(struct inode *ip)`

Updates inode-on-disk from inode-on-memory `ip`.

****4634**:** read entire block from disk

****4635**:** calculate pointer to inode-on-disk (using casting so that `+ inum%IPB` will add the correct size)

****4636-4641**:** set inode data to copy of block that is on the memory now

****4642**:** re-write entire block on disk

****4643**:** close block

`4703 ilock(struct inode *ip)`

Locks inode, without spinning or preventing interrupts.

****4711-4715**:** sleep over inode until it's not busy:

- ****4712-4713**:** while inode is busy, go back to sleep

- ****4714**:** mark inode as busy

****4717-4730**:** make sure inode is still valid - affects only in-memory inode

- ****4718**:** read entire block

- ****4719**:** calculate pointer to inode

- ****4720-4725**:** set data (just in case it changed meanwhile

- ****4726**:** release block

- ****4727****: mark as valid

`4735 iunlock(struct inode *ip)`

Unlocks inode.

****4741****: remove "busy" flag

****4742****: wakeup all procs waiting for inode lock.

`5751 sys_mkdir(void)`

Creates a new directory.

`5513 sys_link(void)`

Creates a new name (or *shortcut*) for a file.

(But not for a directory!)

`5601 sys_unlink(void)`

Destroys a name of a file or directory.

```
### `4902 readi(struct inode *ip, char *dst, uint off, uint n)`
```

Actually reads data from the disk.

****4907-4911**:** (weird stuff beyond the scope of this course)

****4913-4914**:** offset validation

****4915-4916**:** uh...

****4918**:** for each block from offset till block where we want to finish reading:

- ****4919**:** read entire current block

- ****4920**:** figure out how much to read (entire block, or just part)

- ****4921**:** copy the how-much-to-read data to memory (buffer)

- ****4922**:** close current block

```
### `4952 writei(struct inode *ip, char *src, uint off, uint n)`
```

Actually reads data from the disk.

****4957-4961**:** (weird stuff beyond the scope of this course)

****4963-4966**: validation**

****4968**: for each block from offset will block where we want to finish writing:**

- ****4969**: read entire current block (we actually don't really need to do this when we're writing a whole block, but whatever)**

- ****4970**: figure out how much to write (entire block, or just part)**

- ****4971**: copy the how-much-to-write data to memory (buffer)**

- ****4972**: copy buffer to disk**

- ****4972**: close current block**

****4976-4979**: if the file grew because of the write, update inode's *size***

`4856 itrunc(struct inode *ip)`

Destroys inode on disk!

(Must be called only when inode is no longer referenced or held open by anyone.)

****4862-4867**: free direct blocks (if they're allocated) and mark them as such on inode**

****4869**: if there are also indirect blocks:**

- ****4870**: read the block with the indirect pointers**

- ****4871**: cast block to int vector, for convenience**

- **4872-4875**: free indirect blocks (if they're allocated) - no need to mark them on indirect vector, because we'll destroy him soon

- **4876**: close the block with the indirect pointers

- **4877**: destroy the block with the (now freed) indirect pointers

- **4878**: mark the indirect pointer as free on the inode

4881-4882: update inode

4810 bmap(struct inode *ip, uint bn)

Returns the physical block number of `ip`'s `bn`th block.

If block doesn't exist, the block is allocated.

4815-4819: handle case when block is direct

- **4816**: try to get physical address

- **4817**: if no physical address, allocate one

If we reached here, then we know block is indirect

4824-4825: allocate indirect block if it doesn't exist yet

4826: read contents of indirect block

****4827**:** cast the contents as a vector of numbers

****4828-4831**:** try to get physical address

- ****4829**:** allocate new address if needed

- ****4830**:** write new address on the indirect block on the disk

****4832**:** close indirect block

`4102 bread(uint dev, uint sector)`

Gets a block from the disk.

****4106**:** try to get buffer from cache

****4107-4108**:** if buffer is not valid, ask for the actual buffer from the driver (which is the layer that *really* reads from the disk)

`4114 bwrite(struct buf *b)`

Writes a block to the disk.

****4116-4117**:** make sure buffer is marked as busy

****4118**:** mark buffer as dirty (that's our way to tell the driver to *write*)

****4119**:** ask driver layer to write to disk.

`4038 binit(void)

Initialize `bcache` buffer cache.

`4066 bget(uint dev, uint sector)`

Gets a buffer from the cache.

If it's not there, allocate it there.

****4070**:** lock buffer cache

****4072-4084**:** search for buffer in cache

- ****4075**:** buffer found!

- ****4076-4080**:** if buffer is not busy, unlock the buffer cache, mark buffer as busy, and return it

- ****4081-4082**:** if buffer is busy, go to sleep till it's unlocked (but will need to search again for case buffer was changed)

Got here?

Buffer not found; allocate new buffer

****4087**:** loop from end of list to beginning

- ****4088****: check if current buff is not busy and not dirty

- ****4089-4093****: fill data, release lock, return buff

`4125 brelse(struct buf *b)`

Release buffer from being BUSY and move to head of linked list.

****4130****: lock cache

****4132-4137****: reposition buff to list head

****4139****: mark buff as not busy

****4140****: wake up anyone who might be waiting for buff

****4142****: release cache lock

`4454 balloc(uint dev)`

Allocates and zeroes a block on the disk.

****4461**** read super block

`bfree`

`idewait`

`idestart`

`3954 iderw(struct buf *b)`

Handles a queue of blocks to write

(because it can receive many requests during the long time it takes the IDE to actually write stuff to the disk).

3968-3971: append `b` to end of `idequeue`

3974-3975: if our `b` is at the head of the list, read/write it to IDE

3978-3980: sleep until `b` is VALID and NOT DIRTY

`3902 ideintr(void)`

Handles the interrupt from the IDE (which means the disk finished reading/writing).

Manages `idequeue`.

3907-3913: get current head of `idequeue`, and move `idequeue` to next guy.

- **3908-3912**: handle false alarms (which happen sometimes)

- **3913**: "increment" queue

3916-3917: read data (if that's what current buff wanted)

3920-3922: clear buff flags and wakeup all those waiting for buff

3925-3926: tell IDE to start running on next guy in queue

The point of this document:

I'm actually not so sure.

I think the point is to explain why xv6 does stuff, in broad strokes, as a complement to the explanations of the actual code of xv6.

Important-ish note:

Often, I'll refer to things that haven't been explained yet.

Just keep in mind that not everything is explained at first, and that's okay.

These explanations are not full. They may not even be accurate.

I'm just jotting these down during class, hoping it more or less gives a picture of what's going on.

Anyways, here we go.

Kernel

When I say "kernel", I mean the operating system.

This is the mother of all programs that hogs the computer to itself and manages all the other programs and how they access the computer resources (such as memory, registers, time (that is, who runs when), etc.).

The kernel also keeps control over the "mode" bit (in the `%cs` register), which marks whether current commands are running in user-mode (and therefore can't do all kinds of privileged stuff) or kernel-mode (and therefore can do whatever it wants).

Boot

The processor (that is, the actual computer) does not know what operating system is installed, what it looks like, or how large it is.

So how can the processor load xv6 into the memory?

The processor instruction pointer `%eip` register points - by default - to a certain memory place in the ROM, which contains a simple program that:

1. Copies the very first block (512 bytes. AKA the boot block) from the disk to the memory
2. Sets the instruction pointer `%eip` to point to the beginning of the newly-copies data

So what?

Every operating system needs to make sure that its first 512 bytes are a small program (it has to be small; it's only 512 bytes!)

that loads the rest of the operating system to the memory and sets the PC to whatever place it needs to be in.

If 512 bytes aren't enough for this, the program can actually call a slightly larger program that can load and set up more.

In short:

1. `%eip` points to hard-coded ROM code
2. ROM code loads beginning of OS code
3. Beginning of OS code loads the rest of the code
4. Now hear the word of the Lord!

Processes

A process is the running of a program, including the program's state and data. The state includes such things as:

- Memory the program occupied
- Memory contents
- Register values
- Files
- Kernel structures

This is managed by the kernel. The kernel has a simple data structure for each process, organized in some list. The kernel juggles between the processes, using a context switch, which:

1. Saves state of old process to memory
2. Loads state of new process from memory

Context switch can be preemptive (i.e. the kernel decides "next guy's turn!", using the scheduler (there'll be a whole lot of talk about this scheduler guy later on)) or non-preemptive (i.e. the hardware itself decides). In non-preemptive, it is asked of the programmers to make calls to the kernel once in a while, in order to let the kernel choose to let the next guy run.

Xv6 is preemptive.

Processes are created by the kernel, after another process asks it to. Therefore, the kernel needs to run the first process itself, in order to create someone who will ask for new processes to be created.

`fork()`

Every process has a process ID (or pid for short).

A process can call the kernel to do `fork()`, which creates a new process, which is entirely identical to the parent process (registers, memory and everything). The only differences are:

- The pid
- The value returned from fork:
- In the new process - 0
- In the parent process - the new pid
- In case of failure - some negative error code

`exec()`

`Fork()` creates a new process, and leaves the parent running. `Exec()`, on the other hand, replaces the process's program with a new program. It's still the same process, but with new code (and variables, stack, etc.). Registers, pid, etc. remain the same.

It is common practice to have the child of `fork` call `exec` after making sure it is the child. So why not just make a single function that does both fork and exec together? The exec system call replaces the entire memory of the parent process except for the open files. This allows a parent process to decide what the stdin, stdout and stderr for the child process. This trick is used in the /init process in xv6 and the sh to pipe outputs to other processes.

Process termination

Trigger warning: sad stuff ahead. And also zombies.

A process will be terminated if (and only if) one of the following happens:

1. The process invokes `exit()`
2. Some other process invokes `kill()` with its pid
3. The process generates some exception

Note that `kill` does not actually terminate the process. What it does is leave a mark of "You need to kill yourself" on the process, and it'll be the process itself that commits suicide (after it starts running again, when the scheduler loads it).

Note also that not any process can `kill` any other process. The kernel makes sure of that.

Once `kill`ed, the process's resources (memory, etc.) are not released yet, until its parent (that is, the process which called for its creation) allows this to happen.

A process that `kill`ed itself but whose parent did not acknowledge this is called a zombie.

In order for a parent to "acknowledge" its child's termination, it needs to call `wait()`.

When it calls `wait()`, it will not continue until one of its children exits, and then it will continue.

If there are a few children, the parent will need to call `wait` once for each child process.

`Wait` returns the pid of the exited process.

What happens if a parent process `exit`s before its children?

Its children become orphans, and the The First Process (whose pid is 1) will make them His children.

System calls

Many commands will only run if the "mode" bit is set to kernel-mode.

However, all processes run on user-mode only; xv6 makes sure of that.

In order to run a privileged command, a process must ask the kernel (which runs in kernel-mode, of course) to carry out the command.

This "asking" is called a system call.

Here's an example of system call on Linux with an x86 processor:

```assembly

```
movl flags(%esp), %ecx
lea name(%esp), %ebx
movl $5,%eax ; loads the value 5 into eax register, which is the command "open" in Linux
int $128 ; invokes system call. Always 128!
; eax should now contain success code
...
```

The kernel has a vector with a bunch of pointers to functions (Yay, pointers!).

### ### \*Addresses\*

This one's a biggie. Hold on to your seatbelts, kids.

Programs refer to memory addresses. They do this when they refer to a variable (that's right; once code's compiled, all mentions of the variable are turned into the variable's address). They do this in every `if` or loop. When the good old `%eip` register holds the address of the next instruction, it's referring to a memory address.

There are two issues that arise from this:

- \* The compiled code does not know where in the memory the program is going to be, and therefore these addresses must be relative to the program's actual address. (This is a problem with loops and ifs, not with the `%eip`.)
- \* We'll want to make sure no evil program tries to access the memory of another program.

So, each process has to have its "own" addresses, which it thinks are the actual addresses. It has **\*nothing\*** to do with the actual RAM, just with the **\*addresses\*** that the process knows and refers to. (\*\*Process\*\*, not \*\*program\*\*; this includes the kernel.)

Behold! A sketch of what a process's addresses looks like in xv6:

Address | Who uses this

--- | ---

```
`[0xFFFF FFFF]` | Kernel
`[0xFFFF FFFE]` | Kernel
... | Kernel
`[0x8000 0000]` | **Kernel**
`[0x7FFF FFFF]` | **Process**
... | Process
`[0x0000 0000]` | Process
```

In order to pull off this trick, we use a hardware piece called the Address Translation Unit, which actually isn't officially called that.

Its real name is the MMU (Memory Management Unit), for some reason.

The MMU is actually comprised of two units:

1. Segmentation Unit
2. Paging Unit.

The MMU sits on the address bus between the CPU and the memory, and decides which actual addresses the CPU accesses when it reads and writes.

Each of the smaller units (segmentation and paging) can be turned on or off. Note that Paging can only be turned on if Segmentation is on. (We actually won't really use the Segmentation Unit in xv6. LATER EDIT: This is a lie. A LIE! We actually use it for all kinds of unwholesome tricks.)

Addresses coming from CPU are called \*\*virtual/logical addresses\*\*. Once through the Segmentation Unit, they're called \*\*linear addresses\*\*. Once through the Paging Unit, they're \*\*physical addresses\*\*.

In short: CPU [virtual] -> Segmentation [linear] -> Paging [physical] -> RAM

Here's a bunch of 16-bit registers that are used by the Segmentation Unit:

- \* `%cs` - \*\*Code. This guy actually messes with our `%eip` register (that's the guy who points to the next command!).
- \* `%ds` - \*\*Data. By default, messes with all registers except `%eip`, `%esp` and `%ebp`. (In assembly, we can override the choice of messing register.)

- \* `%ss` - \*\*Stack. By default, messes with `%esp` and `%ebp` registers.
- \* `%es`
- \* `%fs`
- \* `%gs`

The address `%eip` points to after going through `%cs` is written as `CS`:`EIP`.

In the Segmentation Unit there is a register called `%GDTR`. The kernel uses this to store the address of a table called GDT (Global Descriptor Table). Every row is 8 bytes, and row #0 is not used. It can have up to 8192 rows, but no more.

The first two parts of each row are \*\*Base\*\* and \*\*Limit\*\*.

When the Segmentation Unit receives an address, the CPU gives it an \*index\*. This index is written in one of the segmentation registers (`%cs`, `%ds`, ... `%gs`), thusly:

- \* Bits 0-1: permissions
- \* Bit 2: "use GDT or LDT?" (Let's pretend this doesn't exist, because it does not interest us at all.)
- \* Bits 4-15: The index

The Segmentation Unit receives a logical address and uses the index to look at the GDT. Then:

- \* If the logical address is greater than the `Limit`, crash!
- \* Else, the Segmentation adds the `Base` to the logical address, and out comes a linear address.

Note: In the past, the Segmentation Unit was used in order to make sure different processes had their own memory. For example, they could do this by making sure that each time the kernel changes a process, its segmentation registers would all point to an index that "belongs" to that process (and each row in the GDT would contain appropriate data). Another way this could be done would be by maintaining a separate GDT for each process. Or maintaining a single row in the GDT and updating it each time we switch a process. There is no single correct way.

Note that all the addresses used by each process must be consecutive (along the physical memory).

In xv6, we don't want any of this.

Therefore, we will make sure that the GDT 'Limit' is set to max possible, and the 'Base' is set to 0.

In order to allow consecutive virtual addresses to be mapped to different areas in the physical memory, we use **\*\*paging\*\***.

In the Paging Unit, there is a register named `%CR3` (Control Register 3), which points to the **\*\*physical\*\*** address of the Page Table (kinda like GDT). A row in the Page Table has a whole bunch of data, such as page address, "is valid", and some other friendly guys.

When the Paging Unit receives a linear address, it acts thusly:

- \* The left-side bits are used as an **\*index\*** (AKA "page number") for the Page Table. (There are 20 of these.)
- \* In the matching row, if the "is valid" bit = 0, crash!
- \* (There are also "permission" bits, but let's ignore them for now.)
- \* Those "page number" bits from the linear address are replaced by the actual page in our row (which is already **\*part\*** of the actual real live physical address)
- \* The page is "glued" to the right-side bits of the linear address (you know, those that aren't the page number. There are 12 of these.)
- \* Voila! We have in our hands a physical address.

Note that each page can be in a totally different place in the physical memory. The pages can be scattered (in page-sized chunks) all along the RAM.

Also note: Hardware demands that the 12 right-most bits of `%CR3` be 0. (If not, the hardware'll zero 'em itself.)

**\*\*Uh oh\*\*:**

Each row in the Page Table takes up 4KB (that's 12 bits).

The Page Table has 1024 rows.

4KB \* 1024 = 4MB. That's 4 whole consecutive MBs. That's quite a large area in the memory, which kind of defeats the whole purpose of the Page Table. Well, not the **\*whole\*** purpose, but definitely some of it.

**\*\*The solution\*\*:** The Page Table gets its very own Page Table!

- \* First (small) page table contains - in each row - the (physical) address of another (small) page table.
- \* Each of the (small) 2nd-level page tables (which are scattered) contain actual page addresses.
- \* So: instead of 20 bits for single index, we have 10 bits for 1st-level table index and 10 bits for 2nd-level table index.

Thus, a virtual address can be broken down like so:

`[i0][i1][offset]`

- \* `i0` is the index of the First Table
- \* `i1` is the index of a second table
- \* `offset` is the offset

### \*Addresses - double mapping\*

Every single physical address is mapped by the kernel to virtual address by adding KERNBASE to it.

When a process is given a virtual address, this new virtual address is *\*in addition\** to the kernel mapping.

So when we want to get the physical address of a virtual address:

1. If the user-code is asking, it needs to access the page tables.
2. If the kernel is asking, it can simply subtract KERNBASE from the address.

Likewise, the kernel can determine *\*its\** virtual address of *\*any\** physical address by adding KERNBASE.

KERNBASE = 0x80000000.

```
`1217 main`
```

In the beginning, we know that from `[0x0000 0000]` till `[0x0009 FFFF]` there are 640KB RAM.

From `[0x000A 0000]` till `[0x000F FFFF]` is the "I/O area" (384KB), which contains ROM and stuff we must not use (it belongs to the hardware).

From `[0x0010 0000]` (1MB) till `[0xFF00 0000]` (4GB - 1MB in total) there is, once again, usable RAM.

After that comes "I/O area 2".

(Why the 640KB, the break, and then the rest? Because in the olden days they thought no one would ever use more than 640KB.)

Address | Who uses this

--- | ---

`[Who cares]` | I/O

... | I/O

`[0xFF00 0001]` | I/O

`[0xFF00 0000]` | Usable RAM

... | Usable RAM

`[0x0010 0000]` | Usable RAM

`[0x000F FFFF]` | I/O

... | I/O

`[0x000A 0000]` | I/O

`[0x0009 FFFF]` | Usable RAM

... | Usable RAM

`[0x0000 0000]` | Usable RAM

Remember Mr. Boot? He loads xv6 to `[0x0010 0000]`.

By the time xv6 loads (and starts running `main`), we have the following setup:

1. `%esp` register is pointing to a stack with 4KB, for xv6's use. (That's not a lot.)

2. Segmentation Unit is ready, with a (temporaray) GDT that does no damage (first row inaccessible, and another two with 0 `Base` and max `Limit`).

3. Paging Unit is ready, with a temporary page table. The paging table works thusly:

- \* Addresses from `[0x800- ----]` till `[0x803- ----]` are mapped to `[0x000- ----]` through `[0x003- ----]` repectively. (That is, the left-most bit is simply zeroed.)

- \* ALL the the above addresses have 1000000000b as their 10 left-most bits, so our 1st-level page table has row 512 (that's 1000000000b) as "valid", and all the rest marked as "not valid".

- \* Row 512 points to a single 2nd-level table.

- \* The 2nd-level table uses all 1024 of its rows (that's exactly the next 10 bits of our virtual addresses), so they're all marked as valid.

- \* Each row in 2nd-level table contains a value which - coincidentally - happens to be the exact same number as the row index.

Let's look at some sample virual address just to see how it all works:

`[0x8024 56AB]` -> `[1000 0000 0010 0100 0101 0110 1010 1011]` -> `[1000 0000 00` (that's row 512) `10 0100 0101` (that's row 581) `0110 1010 1011` (and that's the offset)`]` -> row 581 in the 2nd-level table will turn `[1000 0000 0010 0100 0101...]` to `[0000 0000 0010 0100 0101...]`, which is exactly according to the mapping rule we mentioned a few lines ago.

### \*Available pages (free!)\*

Processes need pages to be mapped to. Obviously, we want to make sure that we keep track of which page are available.

The free pages are managed by a \*\*linked list\*\*. This list is held by a global variable named `kmem`. Each item is a `run` struct, which contains only a pointer to the next guy.

`kmem` also has a lock, which makes sure (we'll learn later how) that different processes don't work on the memory in the same time and mess it up. (An alternative would be to give each processor its own pages. However, that could cause some processors to run out of memory while another has spare. (There are ways to work around it.))

`main` calls `kinit1` and `kinit2`, which call `freerange`, which calls `kfree`.

In `kfree`, we perform the following 3 sanity checks:

\* We're not in the middle of some page

- \* We're not trying to free part of the kernel
- \* We're not pushing beyond the edge of the physical memory

### ### \*Building a page table\*

Let's examine what needs to be done (not necessarily in xv6) in order to make our very own paging table.

Our table should be able to "translate" virtual address `*va*` to physical address `*vp*` according to some rule.

Note that we'll be using `'v2p'`, which is a hard-coded mother-function that translates **\*\*virtual\*\*** addresses to **\*\*physical\*\*** by subtracting `'KERNBASE'` (which equals `'0x8000 0000'`) from the virtual addresses.

**\*\*Step 1\*\*:** Call `'kalloc'` and get a page for our First Table. (Save (virtual!) address of new table in `'pgdir'` variable)

**\*\*Step 2\*\*:** Call `'memset'` to clear entire page (thus marking all rows as invalid).

**\*\*Step 3\*\*:** Do the following for **\*\*every single \_va\_\*\*** we want to map:

- **\*\*Step 3.1\*\*:** Create and clear subtable, and save address in `'pgtab'` (similar to what we did in steps 1 and 2). (SEE NOTE AFTER THIS LIST)

- **\*\*Step 3.2\*\*:** Figure out `i0` (index in `'pgdir'`) (using `*va*` & `'v2p'` function), write `'pgtab'` there, mark as valid.

- **\*\*Step 3.3\*\*:** Figure out `i1` (index in `'pgtab'`) (using `*va*` & `'v2p'` function), write `*pa*` there, mark as valid.

**\*\*Step 4\*\*:** Set `'%CR3'` to point at `'pgdir'`, using `'v2p'`.

THE NOTE MENTIONED EARLIER: After we already have some subtables, we only need to create new subtables if the requested subtable does not exist yet.

How do we know whether it exists already? Simply by looking at the current iO and seeing whether it's already marked as valid.

ANOTHER NOTE: What if two different `*va*->*vp*` rules clash? xv6 will crash (deliberately).

\*\*For more details about how the kernel builds its mapping tables, please refer to [xv6 Code Explained.md] (`'kvmalloc'`).\*\*

### \*Moar GDT stuff!\*

Remember how we said we'd make sure GDT has `'base=0'` (so it won't alter addresses) and `'limit=max'` (so it won't interfere)?

Well, it turns out the hardware still uses the GDT to check various user-mode/kernel-mode stuff.

We need four rows in the GDT:

1. Kernel-mode, can only execute and read
2. Kernel-mode, can write
3. User-mode, can only execute and write
4. User-mode, can write.

There is a macro named `'SEG'`, which helps us build all these annoying bits.

Okay, that's enough of this.

Don't pretend you understand, because you don't and it doesn't matter.

### \*Per-CPU variables\*

We've got an array of CPU data, `'cpus'`.

We can access specific CPU stuff via `'cpus[SOME_CPU_IDENTIFIER]'`.

In order to get current CPU identifier, we call `'getcpu()'`, which is slow.

When we do this, we **\*MUST\*** stop interrupts from happening, to make sure we stay within the same CPU.

**\*\*Problem\*\*:** Calling `getcpu()` is slow, and stopping (and then resuming) interrupts can be extremely slow.

**\*\*Solution\*\*:** Instead of using `getcpu()`, we can use a special register in each CPU!

**\*\*Problem\*\*:** Registers can be overriden by users.

**\*\*Solution\*\*:** Special register that only kernel can use!

**\*\*Problem\*\*:** \*There are no such registers.\*

**\*\*Solution\*\*:** Cheating, using the GDT!

So we have a list of GDTs, one per CPU.

When we initialize a CPU (just once, yeah?), we set up its very own GDT.

In this GDT, we add a **\*fifth\*** row (remember, we have four rows for kernel/user x read/write).

In this new row, we set the base to point at the place in the memory where the CPU data sits, and set the limit to 8 (because we have two important variables of CPU data, and each one take 4).

We do this in `seginit()`.

```C

```
struct cpus *c = &cpus[getcpu()]; // costly, but used just once!
```

```
// Set first four rows...
```

```
c->gdt[SEG_KCODE]=SEG(STA_X|STA_R, 0, 0xffffffff, 0);
c->gdt[SEG_KDATA]=SEG(STA_S, 0, 0xffffffff, 0);
c->gdt[SEG_UCODE]=SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
```

```

c->gdt[SEG_UDATA]=SEG(STA_S, 0, 0xffffffff, DPL_USER);

// Set our special row

c->gdt[SEG_KCPU]=SEG(STA_W, &c->proc, 8, 0);

// Load the GDT

lgdt(c->gdt, sizeof(c->gdt));

// Set %gs register to point at our fifth row in the GDT.

loadgs(SEG_KCPU << 3); // shift 3 left so we get value and not permission bits.

// Declare out two variables,
// telling code generator to use gs register
// whenever we access proc or cpu

extern struct proc *proc asm("%gs:0");
extern struct proc *cpu asm("%gs:4");

proc = NULL;
cpu = c;
...

```

So how does this affect our accessing per-CPU data?

Normally, if we do `proc = 3;`, then it would be complied to `movl \$3, proc`.

However, now that we did that weird `extern struct proc *proc asm("%gs:0");` part, it is compiled to `movl \$3, %gs:0`.

Therefore, whenever in the code we call `proc` or `cpu`, we will automagically be referring to the address held in `GS:0` or `GS:4`, which happens to be the `proc` or `cpu` belonging to the specific CPU in which the code is executed.

Note that variable `cpu` points to the variables `cpu` and `proc`, so calling `proc` is the same as (yet faster than) calling `cpu->proc`.

Note also that this means that calling `cpu->cpu->cpu->cpu` is the same as calling `cpu`.

****Important note**:** `GS` can be changed by code in user-mode.

So, every time there is an interrupt, we reload our value to `%gs` using `loadgs(SEG_KCPU << 3);`. (Don't worry, we back up all user-mode's registers beforehand.)

Processes

Every process has a struct `proc` that holds a bunch of its data:

- * `sz` - memory size
- * `pgdir` - its very own page table
- * `kstack` - small **kernel** stack (4KB), for data we don't want the process to touch and break (such as return address from system call)
- * `state`
- * `pid` - process ID
- * `parent` - parent process
- * `tf` - pointer to place where register values are saved during interrupt
- * `context` = another interrupt thing
- * `chan` - channel, marks what event process is waiting for (while sleeping). If none, then set to 0.
- * `killed` - 0 if not killed yet
- * `ofile` - vector of open files
- * `cwd` - current working directory (like when doing `cd...` in command)
- * `name`

A process can move through its processes thusly:

Embryo -> Runnable -> Running

Running -> Zombie

Running -> Sleeping (waiting for whatever event is marked in `chan`)

Running -> Runnable

The processes are held in a struct named `ptable`, who has a vector of processes named `proc`.

Preparing the First Process

When running The First Process, we do the following in `userinit`:

- * `allocproc`: allocate `proc` structure
 - Allocate process kernel stack
 - Set up data on kernel stack (in the "trapframe" area) **as if the process already existed and went into kernel mode**. As a result, the data will be loaded to proc in due time
- * `setupkvm`: create page table (without knowing yet where to map user addresses (only kernel addresses))
- * `inituvm`:
 - Allocate free page
 - Copy code to page
 - Update page table to map user memory
- * Fix the data on the kernel stack for some reason.

Trapframe contains all register data.

When we return from an interrupt (or, in our case, start the process for the first time), we do the following:

1. Pop the first 4 values to their appropriate registers
2. Pop `eip` field to `%eip` register (now `%eip` register is pointing to `forkret` function)
3. Goto `forkret` function, which (somehow) takes us to the next guy in the stack...
4. ...Trapret which pops another whole bunch of values from the stack to the registers
5. And so on. We end up with all registers holding good values, and `%eip` pointing to 0.

The First Process

All it does is execute the program "init" (with the arguments {"/init", 0} (the 0 is to mark the end of the list)).

This is how the code ***would*** have looked like if it were written in c:

```
```c
char *argv[] = {"/init", 0};
main() {
 exec(argv[0], argv);
 for (;;) exit();
}
```
```

```

...But it's not written in c.

It's written in assembly.

Here's the arguments code:

```
```Assembly
# these are the arguments:

init:
.string "/init\0"

.p2align 2

argv:
.long init
.long 0
```
```

```

...And here's the ***code*** code:

```
```Assembly
```

```
.globl start

start:
 pushl $argv ; push second argument
 pushl $init ; push first argument
 pushl $0
 movl $SYS_exec, %eax ; load the canons (perpare to call "exec" system-call)
 int $T_SYSCALL ; FIRE!
```

```
once end up back here, we need to quit.
```

```
exit:
```

```
 movl $SYS_exit, %eax
 int $T_SYSCALL
 jmp exit
```

```
...
```

So we have this code, but we need to load it during `inituvvm`.

This is done in the script in the `**initcode.S**` file, which compiles that code to binary and shoves it all into the data section, with the exact same labels that are used in `inituvvm`.

```
Actually running the First Process
```

This is done in `mpmain`, which calls the scheduler, which loops over process tables and runs processes.

Since at this point there's only one process at this point, it's the only one that'll run.

```
Accessing process's kernel stack during interrupt
```

When we're in user-mode and there's an interrupt, the kernel needs its own stack on the process to push real data onto it (before leaving the process).

We need this kernel-stack to be accessible only by the kernel.

The hardware supports this, with a magnificent Rube Goldberg machine:

- \* `%tr` register can only be accessed in kernel-mode
- \* `%tr` contains `SEG\_TSS`, which points to special place in GDT
- \* special place in GDT contains address and size of some weird struct
- \* weird `TSS` struct contains field `esp0`, which actually (finally) points to top of process's kernel stack (and `ss0` with stack size)

So during `switchuvm` we set up this whole thing, so that in due time the kernel (and only the kernel) can access its stack on the proc.

#### #### \*Locks - the problem\*

Because we have multiple CPUs, we need to make sure two CPUs don't both grab the same data (such as the same `proc` struct during `allocproc`, or the same page during `kalloc`).

This could (theoretically) even happen with a single CPU, because of interrupts.

#### #### \*Locks - a possible solution (just for interrupts)\*

We can block interrupts (!) from happening by setting the `IF` bit in `%eflags` register to 0.

This can be controlled (in kernel-mode, yes?) using the assembly commands:

- \* `cli` (clears bit)
- \* `sti` (sets bit to 1)

(Also, the entire `eflags` register can be pushed into `eax` using some other assembly command.)

So each time we want to do something unsafe, we can do:

```
```C
cli();
something_dangerous();
sti();
```

```

...But this is still dangerous, because of cases such as the following:

```
```C
function b() {
    cli();
    // ...
    sti();
}

function a() {
    cli();
    b(); // at the end of this we set sti()!
    // ... Uh oh. We're open to interrupts!
    sti();
}
```

```

### \*Locks - an actual solution (just for interrupts)\*

\*\*Solution\*\*: Keep track of how many times we called `cli()`, so that `sti()` only releases if down to 0. We can do this by wrapping all `cli()`'s with `pushcli()` and `sti()`'s with `popcli()`.  
In `popcli()`, if our count reaches 0, we don't automatically call `sti()`, but revert to the \*\*previous setting\*\*.

```
```C

```

```

function b() {
    pushcli();
    // ...
    popcli();
}

function a() {
    pushcli();
    b(); // at the end of this we DO NOT set sti()
    // ... the counter is still 1
    popcli(); // NOW the counter became 0, so now we call cli().
}
```

```

(Note that each CPU runs `pushcli()` and `popcli()` separately for each CPU.)

### \*Locks - an actual solution (for multiple CPUs)\*

There is a magical assembly word, `lock`.

Usage example:

```Assembly

lock ; xchgl mem, reg

...

If `lock` is added to different commands run by different CPUs, then the `lock`ed commands will execute ***serially***.

Using this, the following c call:

```c

```
old = xchg(&lock, b);
```

```
...
```

...Does the following:

1. Sets old = lock
2. Sets lock = b (but only if not locked by someone else!)

How on earth does this help us?

I'll tell you how!

We have a struct called `spinlock`, which holds all kinds of locking data.

In the function `acquire`, we do the following:

```
```C
```

```
acquire(struct spinlock *lk) {  
    pushcli(); // disable interrupts in this CPU, to avoid deadlocks  
    while(xchg(&lk->locked, 1) != 0);  
}
```

```
...
```

`xchg` will always return the **existing** value of `&lk->locked`, so if it's already locked we'll loop until it's released.

In `release`, we do the following:

```
```C
```

```
release (struct spinlock*lk) {
 xchg(&lk->locked, 0);
 popcli();
```

```
}
```

```
...
```

### ### \*How locks are managed during `scheduler`\*

When `scheduler` searches for RUNNABLE procs, it `acquire`s a lock on the process table.

This is in order to make sure that `scheduler`s on **\*other\*** CPUs don't grab the same process we just did.

The lock is held even as we switch to the process; it's the process's responsibility to release the lock (in order to allow interrupts).

Likewise, the process must re-lock the process table before returning to `scheduler`.

Each lock is re-released by the next process that runs, and then re-locked.

The final release is done by `scheduler` once there are no runnable processes left.

### ### \*Sleep\*

When a process needs a resource in order to continue (such as disk, input or whatever), it must call `sleep` where it marks itself (`proc->chan`) as waiting for the wanted event, sets its `state` to SLEEPING, and returns to `scheduler`.

When the Event Manager (whom we never heard of yet) sees that the event occurs, it marks the proc as RUNNABLE once again.

Note that the process needs to see that its resource is still available (and hasn't been made unavailable again by the time it woke up) before continuing. If the resource is unavailable again, it should go back to sleep (if it knows what's good for it).

`sleep` also demands a locked `spinlock` for a weird reason we don't know yet.

It replaces the locking from **\*that\*** lock to the process table (unless they are the same lock, of course).

Once done, it releases the process table and re-locks the supplied `spinlock`.

We'll get back to you folks on this one once we understand why on earth this is needed.

#### ### \*Interrupts - CPU\*

There are two basic types of interrupts: internal (thrown by CPU) and external (like keyboard strokes).

We can decide whether to listen to external interrupts using kernel-mode functions `sti()` and `cli()` which sets or clears the `IF` bit on the `%eflags` register.

We cannot ignore internal interrupts.

Interrupts can only occur *\*between\** commands.

During an interrupt, the CPU needs to know what kind of interrupt it is. These range between 0-255, when 0-31 are reserved by Intel for all kinds of hardware stuff.

In order to handle these interrupts, the CPU needs a table with pointers to functions that handle 'em (with the index being the interrupt number).

This table is called IDT, and the register that points to it is `%IDTR`.

Each row has a whole descriptor of 64 bits.

These include:

- \* `offset` - the actual address of the function (this guy is loaded to `%eip`)
- \* `selector` - loaded to `%cs`
- \* `type` - 1 for trap gate, 0 for interrupt gate (if interrupt, we disable other interrupts)

Before an interrupt call, the CPU must do a bunch of stuff to rescue our registers before they are overriden by the IDT guy.

Reminder: the kernel stack's address is saved in a `tss` structure, which is saved in the GDT, in the index held by `%tr` register.

Therefore: the kernel needs to set this before an interrupt occurs.

**\*\*SO\*\*:** The CPU uses the kernel stack to store our registers.

(And after the interrupt handling is over, the CPU needs to pop these guys again. It does this with assembly `iret` command.)

If the interrupts occurs during \*\*kernel-mode\*\*, the CPU performs the following:

- \* push `%eflags`, `%cs` and `%eip` to the \*\*kernel\*\* stack (so we'll have them again after the interrupt).

If the interrupts occurs during \*\*user-mode\*\*, the CPU performs the following:

- \* Store `%ss` in `%ss3`
- \* Store `\$esp` in `%esp3`
- \* Store `tss.ss0` in `%ss`
- \* Store `tss.esp0` in `%esp`
- \* Push `%ss3`, `%esp3`, `%eflags`, `%cs` and `%eip` to \*\*kernel\*\* stack

**\*\*Question\*\*:** Since we push a different number of registers for kernel-mode and user-mode, how do we know how many to pop back out?

**\*\*Answer\*\*:** In both cases, we need to first pop `%eip`, `%cs` and `%eflags`.

Having done that, we can look at the two right-most bits of `%cs` to see whether we were in user- or kernel-mode before the interrupt! Hurray!

### ### \*Interrups - xv6\*

In xv6, all interrupts are handled by a **\*single\*** C function `trap`.

However:

1. This function needs to end with `iret` (that's the guy who handles all the register poppin' at the end of the call), while C functions end with `ret`!
2. Different interrupts need to send different data to the function (such as interrupt number, error number (which not every interrupt needs)).

In order to accomplish this, we wrap the call to `trap` in assembly code that does the following:

1. Push registers to kernel-stack (except those which were already pushed by CPU (see previous section))
2. Push address of kernel-stack to kernel-stack (so the address is sent as a parameter to `trap`; I'll elaborate more on this in a moment)
3. Call `trap`
4. Pop the registers
5. Do `iret`

The function `trap` receives a pointer to a `trapframe` struct, which is actually the kernel-stack.

The fields in `trapframe` are the actual register values, so `trap` can use them in order to determine all kinds of stuff (such as "were we in user-mode when the interrupt occurred?"). Additionally, we have the interrupt number and error-code (where applicable).

The assembly code that wraps the call to trap is mostly the same for each interrupt, with just the slightest difference:

Which interrupt number to push, and do we push an error-number (and if so, which).

So, the shared code is under a label `alltraps` (which, as it says on the label, is for all traps).

Each interrupt does its own little thing and then jumps to `alltraps`.

The code parts for the different interrupts are labelled `vector0`, `vector1`, ... `vector255`, and are generated by a script (in vectors.pl file).

After generating the vectors, the script creates an array (called - surprise, surprise - "vectors").

Later, during `main`, we call `tvinit` which loops over the vectors array and writes the data in the IDT table.

```
syscall
```

When user-code calls a system-call, the system-call number is stored in `%eax`.

So in `syscall`, we can access the system-call number via `proc->tf->eax`.

After running the system-call, we put the return value in `proc->tf->eax`; this will cause `alltraps` to pop it back to `%eax`, and then the user-code will know whether the system-call succeeded.

The system-call numbers are saved as constants in syscall.h file.

Then we have a vector that maps each constant to its appropriate function. This vector is used in `syscall` function in order to execute the system-call.

```
`sleep` system-call
```

There is a global variable called `ticks`.

Whenever there is a timer interrupt occurs, `ticks` increments by 1.

We use the \*address\* of `ticks` as the \*\*event number\*\* for the timer.

So...

When a process calls `sleep`, it sets its channel to equal `&ticks`, and stores the current value of `ticks`.

Whenever a timer event occurs, if the current value of `ticks` minus the value of `ticks` that the process stored \*reaches\* the number of ticks the process asked to sleep for, it becomes RUNNABLE.

Here's a sample of what a call to sleep looks like in assembly (for 10 ticks) :

```
```assembly
pushl $10
subl $4, %esp ; empty word. Some weird convention
movl $SYS_sleep, %eax
int $64
```

```

```
Getting arguments in system-calls
```

In the example of `sleep`, the argument (how many ticks to wait) is stored on the \*\*user\*\*-stack.

This is accessed via `proc->tf->esp + 4` (the +4 is to skip the empty word, which is on the stack because of some weird convention). We do this in `argint` function.

Because this the data is placed on the stack by \*user code\* (which is pointed at by `%esp`), we need to check that the user didn't set `%esp` to point at some invalid address! We do this in `fetchint` function.

```
fork() #2
```

When we `fork` a proc, we need to copy a bunch of its data.

We want to map \*new\* physical addresses, but copy the old data.

We want to create a \*new\* kernel stack.

We want to copy context.

We want the state to be RUNNABLE (and not RUNNING like parent).

We want a new `pid`.

A lot of this stuff is already handled by `allocproc`.

The memory stuff is handled by `copyuvvm`.

```
exec() #2
```

`exec` replaces the current process with new one.

Because this may fail, we don't want to destroy the current memory data until we know we succeeded.

In order to keep the current memory until the end, we need to create a \*new\* memory mapping, and only switch to it (and destroy the old mapping) at the very end.

The four stages of `exec`:

1. Load file
2. Allocate user stack (and a guard page to protect data from being overwritten by stack)
3. Pass argv[] arguments
4. Fix trapframe and switch context

In order for `exec` to run the file it's told to run, the file needs to be in ELF format.

### \*ELF\*

Executable and Linkable Format.

In xv6 we only use **\*static\*** ELF, but in real life there are also **\*dynamic\*** ones but we don't care about that here.

ELF file have:

1. ELF header at the very start
2. `proghdr` vector
3. Program sections

The program sections include the actual code, as well as external code linked by the linker.

There can be many program sections, scattered along different parts of the file (but not at the beginning).

Each program section needs a **\*program header\***, which says where it is in the file and where it is in the RAM.

All the **\*program headers\*** are held in the `proghdr` vector.

The location of `proghdr` is held in the ELF header.

ELFHDR (that's our ELF header) has a bunch of fields, but we'll only look here at a few:

- \* `uint magic` - must equal `ELF\_MAGIC`. Just an assurance all's good.
- \* `ushort machine` - the type of processor. xv6 actually doesn't pay attention to this, so I don't know why we bothered mentioning this one.
- \* `uint entry` - virtual address of `main`
- \* `uint phoff` - PH offset, location of `proghdr`
- \* `uint phnum` - PH number, length of `proghdr`
- \* `uint flags` - uh... flags.

Since the ELF is created by user-code, the kernel doesn't trust it.

The kernel treats the data with caution, and fails the `exec` if there are any errors.

`proghdr` has the following important fields in each vector:

- \* `uint off` - where section starts in file
- \* `uint filesz` - where sections ends in file (so last byte is in `off+filesz-1`)
- \* `uint vaddr` - virtual address to which section is loaded
- \* `uint memsz` - size section takes in memory. May be larger than `filesz`, because section may include lots of zeros (but not the other way around).

The kernel needs to loop and copy the memory, allocating new memory if either 1) we need more memory or 2) `vaddr` is beyond what is already allocated. This is done using `allocuvm`.

### \*`exec` - guard page\*

After `exec` loads da codes, it allocates another page for the user-stack.

**\*\*Problem\*\*:** Stack goes to \*lower\* addresses as it fills up; eventually, it'll overrun the code-n-data!

**\*\*Solution\*\*:** Add guard page, with user-mode bit cleared. That way, user-code will get an error if StackOverflow.

### \*`exec` - pushing arguments to new process\*

New process expects the following to be on the user-stack:

|                                                                                   |
|-----------------------------------------------------------------------------------|
| variable   what it holds                                                          |
| ---   ---                                                                         |
| `argv`   pointer to vector of pointers to string arguments (with 0 in last place) |
| `argc`   number of arguments                                                      |
| return address                                                                    |

...And this is how `exec` fills the user-stack:

|                                 |
|---------------------------------|
| user-stack                      |
| ---                             |
| argument                        |
| ...                             |
| argument                        |
| 0                               |
| pointer to argument             |
| ...                             |
| pointer to argument             |
| `argv` (pointing to this ↑ guy) |
| `argc`                          |
| -1                              |

Note that the `pointers to arguments` don't really need to be on the stack; we just put 'em there because we have to put them \*somewhere\* so why not.

Note also that we don't know in advance how many arguments there are and how long they'll be.

Therefore, our code must first place the arguments stuff, and only afterwards copy `argc`, `argv` and return address.

\*Therefore\*, we save `argc` & `argv` & return address to a temporary variable `ustack`, which we copy to actual stack at the end of the argument pushing.

### \*I/O\*

xv6 supplies the following system-call functions for files:

- \* `open(char \*name, int flag)` - opens file and returns file descriptor (i.e. index in file array)
- \* `close(int fd)` - closes file
- \* `pipe(int pipefd[2])` - kinda like a file, for sending messages between processes
- \* `dup(int fd)` - duplicates opening to file (so if we move the cursor/offset of one (such as by reading), it'll affect the other)
- \* `read`
- \* `write`
- \* `stat`

Reading/writing can be much different if we're dealing with keyboard, file, etc., but xv6 strives to make them "act" the same.

We've got the following read/writables:

- \* pipe
- \* inode
  - file
  - directory
  - device

Each has its own internal functions for reading, writing, etc., but - as mentioned above - xv6 wraps 'em and hides 'em from user code.

xv6 has an abstract struct `file`, which it uses for all the above read/writables.

xv6 has functions that deal with this `file`, but they accept a **\*pointer\*** instead of a file descriptor.

Why? Because:

\* xv6 needs a pointer to an actual struct, because it needs to handle data.

\* User code isn't trusted to handle pointers; it gives us a file descriptor, and the \*kernel\* accesses the actual pointer.

So how do we maintain this translation between file descriptors and pointers?

With an array "ofile", that's how.

These `file` guys have basic data that all read/writables contains, such as "type".

\*\*Inode\*\*'s implement their "inheritance" by having their "type" be "FD\_INODE", and containing a pointer (in "inode" field) to a `struct inode` that has its very own data (such as "type", which could be "T\_FILE" for a file or "T\_DEV" for device).

OK, this explanation is pretty lame :(

You should really just see the "I/O" presentation on Carmi's site, which he changes every semester.

Currently, the presentation is at <http://www2.mta.ac.il/~carmi/Teaching/2016-7A-OS/Slides/Lec%20io.pdf> (slides 13-23).

### \*Pipe stuff\*

A `pipe` is pointed to by a \*reader\* `file struct` and a \*writer\* `file struct`.

(Both of which exist \*once\*, and may be pointed to by many different procs.)

The `pipe` has `readopen` and `writeopen` fields, which contain "1" as long as the \*reader\* and \*writer\* guys are still holding him open.

When both are "0", the `pipe` is closed.

### \*Moar file stuff\*

The kernel has an array `ftable`, which holds all the open files (and a lock to make sure two procs don't fight over a slot in the array).

Files are added to `ftable` in the function `filealloc`.

### \*Transactions\*

When writing to blocks on disk (or deleting), we really don't want the machine to crash or restart, or else there'll be corrupted data.

In order to alleviate our fears, we can wrap our \*writing\* in a \*transaction\* (by calling `begin\_trans()` before and `commit\_trans()` after).

A transaction ensures that the write will either be \*complete\* or won't be at all.

A transaction is limited to a certain size, so during `writefile` we need to divide the data to manageable chunks.

Therefore, if there's a crash in the middle of the write operation, some blocks might be written and the rest not. However, each block will be fully-written or not written at all (but no hybrid mish-mash).

### ### \*inode Stuff\*

xv6 considers a directory to be a file like any other.

Every `struct dirent` has an inode number, and a name. (If the directory is not in use, the number is 0.)

The inodes are managed by the inode layer, which we will talk about later.

`struct inode` has the following amongst its fields:

- \* `uint dev` - device number
- \* `uint inum` - inode number
- \* `int ref` - reference count
- \* `int flags` - flags, such as 0x1 for I\_BUSY (that is, \*locked\*) and 0x2 for I\_VALID
- \* `short nlink` - the number of names ('links') the inode has on the disk (kinda like a shortcut in Windows)
- \* `uint size` - the size of the inode on the disk

In order to open an inode, we have `namei()`.

In order to search for an inode, we have `dirlookup()`.

Note that `dirlookup` can be supplied with an optional pointer `poff` that gets the offset of the found inode.

Why?

In case of renaming or deleting, we'll need to update the row in our parent inode. (In case of name change, change the num; in case of deletion, change `inum` to 0.)

So... when we open an inode, we need to supply the inode to search in.

Conveniently, we actually \*don't\* supply an inode pointer, because we have THE CURRENT WORKING DIRECTORY which is "supplied" automatically.

...Actually, that's not exactly true.

It *\*is\** true if we're looking for some "a/b/c" path, but *\*\*not\*\** for "/a/b/c".

In the latter case, we use the *\*\*root inode\*\** instead of the current working directory.

As you may or may not have guessed, if we have a path with a few parts (such as "a/b/c/d"), we need to loop over the parts and for each part find the matching inode.

In order to split the path into parts, we use `skipelem`.

### ### \*inode Stuff - The Inode Layer\*

The inode layer supplies a bunch of funcs:

- \* `iget` - open
- \* `iput` - close
- \* `idup` - increments ref count
- \* `ilock` - must lock inode whenever we want to access any field
- \* `iunlock` - unlocks inode
- \* `ialloc` - allocates inode both on disk and in memory
- \* `iupdate` - update disk with whatever changes we made to inode
- \* `readi`

\* `writei`

Our inode structs are saved in `icache`.

They are saved there the first time they are opened (but not before; they are loaded on-demand).

The inodes are added to `icache` in `iget`.

When added, we add them \*without the actual data from the disk\*.

Why?

Because often we call `iget` just to see if the inode exists, and actually reading from the disk is expensive (so we do it just if we need it).

### ### \*The Disk - Reading and Writing\*

The disk is divided to blocks.

Every block on our disk is of 512 bytes; no more, no less.

Reading and writing to and from a block is done \*just\* in the beginning of a block.

So how do we write (or read) only a few bytes?

We have a buffer layer that takes a whole block to memory, writes (or reads) our few bytes \*in memory\*, then - if we're writing - re-writes the entire block to the disk.

The buffer layer contains the functions:

\* `readsb` - reads super-block. Never mind.

\* `bread(dev, sector)` reads an entire block. (function returns a `struct buf` which contains a field `data` with the actual 512 bytes of data)

\* `log\_write`

\* `brlse`

\* `balloc`

\* `IBLOCK` - we'll explain in a moment:

Remember `struct inode`?

Well, there's also `struct dinode` which represents an inode *\*on the disk\**.

It's similar to the inode, but without the meta-data (such as ref, inum, etc.).

`IBLOCK` is a macro that gives us the block number of an inode.

In order to tell *\*where\** within the block is our inode, we can calculate  $(\text{inum} \% \text{IPB})$ . IPB is the number of inodes in a block.

### \*`ilock`\*

As mentioned before, before accessing inode data we need to `ilock` it.

We don't want to use our regular locks, because they:

- \* disable interrupts

- \* when trying to acquire, "spin" over lock till lock is open

This wastes a lot of time.

`ilock` is a *\*soft\** lock, that doesn't do this stuff.

We don't disable interrupts, and instead of spinning, we do `sleep` (so we're not hogging CPU while waiting for acquiring).

### \*Reading from the disk\*

Every inode on the disk has a vector of `addrs`.

Every entry points at a single block on the disk.

The first 12 entries are *\*direct\** pointers.

The 13th points to a block that serves as vector of pointers.

That is, the last pointer is a pointer to pointers.

It turns out there's actually a reason for having the inode data be partially direct and partially indirect:

- \* We have the direct blocks because reading from the indirect blocks cost an extra read for each block.
- \* We have the indirect blocks because the direct blocks are a waste of space for small files.

### ### \*The buffer layer\*

The buffer layer supplies the following functions:

- \* `bread`
- \* `bwrite`
- \* `balloc`
- \* `bfree`

`struct buf` has the following fields:

- \* `flags` - such as BUSY or DIRTY
- \* `dev`
- \* `sector` - device and block num
- \* `struct buf \* prev` - for linked list
- \* `struct buf \* next` - for linked list
- \* `struct bref \* qnext`
- \* `uchar data[512]`

All our buffers are kept in a cache, `bcache`.

This cache contains a spinlock (obviously), and a linked list of buffers (held in field `head`).

The linked list of buffers is maintained so that the most recently used is in the head, and the least in the tail.

Bufs are never removed from the cache!

If they're not used, they're marked as \*not\* busy and \*not\* dirty.

If we need a new buff (such as calling `bget` for a block that hasn't been read yet), we find a `struct buff` at the end of the list that is neither BUSY nor DIRTY, and mark it BUSY.

When we call `brelease` and release a buffer, it is moved to the head of the list (and its BUSY flag is turned off).

`bcache` is initialized in `binit()`.

### ### \*The superblock and the bitmap\*

There is a block on the disk called the \*superblock\*.

This block is located right after the boot block, and contains meta data regarding the other blocks.

There is another area on the disk called the \*bitmap\*.

For every block on the disk, there is a bit that marks whether the block is free or not.

### ### \*The driver layer\*

Ah, the driver layer. The guy who actually does all the dirty work.

In order to actually access the disk (as well as other peripherals, such as keyboard), we have controllers.

Our guy is the IDE.

The IDE has a bunch of \*ports\*, which are actually numbers (but "port" sounds a lot more computer-y than "number").

The processor can send/receive values to/from these ports using IN/OUT commands.

The IDE decides what to do for different values it is given to different ports.

If we want the IDE to read (or write) from the disk, we need to tell it:

1. Which disk to read from (turns out there are two)
2. Where on the disk to read (28 bits to tell us which block)
3. How many blocks to read

Each of these needs to be sent to different OUT ports, before we send the command "read" to the port that accepts the command:

- \* 0x1f2 - number of blocks
- \* 0x1f3, 0x1f4, 0x1f5 and 0x1f6 - block number (28 bits split to 8 bits per port (0x1f6 gets only 3 bits and device number))
- \* 0x1f7 - the command

The IDE can only handle one single command at a time; so before we do anything, we need to check its IN 0x1f7 port (the STATUS port).

The left-most bit tells us if the IDE is busy (so shouldn't use).

The second bit tells us if the IDE is ready (so can use).

The IDE has its very own RAM attached to it, which it uses to read/write.

(NOTE: We don't know how much is in this RAM. It depends on the controller we chose to put in our PC.)

So how do we access this RAM from our code?

### \*Accessing this RAM from our code\*

In order to do this, we use OUT port 0x1f0.

For example:

```C

```
long *a (long*)b->data;  
for (int i = 0; i < 128; i++)
```

```
outl(0x1f0, a[i]);
```

```
...
```

Although it looks like every `long` is being written to the same place, it isn't.

Same goes for reading:

```
```C
```

```
long *a (long*)b->data;
```

```
for (int i = 0; i < 128; i++)
```

```
 a[i] = inl(0x1f0);
```

```
...
```

Another useful parameter to send is `0` to port 0x3f6.

This tells the IDE to raise an interrupt when it finishes the command.

(Otherwise, the kernel can't tell when the read/write from/to the disk is complete.)

#### \*Functions provided by the driver layer\*

- `idewait` - loops over IDE port 0x1f7 until status is READY

- `idestart` - \*starts\* requesting to read/write (whether to read or write depends on input)

- `iderw` - the \*real\* read/write function. Handles a queue of blocks to write, because it can receive many requests during the long time it takes the IDE to actually write stuff to the disk.

In order to handle the queue, the buffer layer has a variable `idequeue`.

This is a linked list of bufs, which are actually read/written by `ideint`, which is run whenever there is an interrupt from IDE due to IDE finishing its previous command.

- `ideint` - handles "I'm finished!" interrupt from IDU and managed `idequeue`.

**\*\*main\*\* \*(kernel entry point)\***

- **\*\*kinit1\*\* \*(frees pages)\***

- **\*\*freerange\*\* \*(frees pages)\***

- **\*\*kfree\*\* \*(frees single page)\***

- **\*\*kvmalloc\*\* \*(builds new page table)\***

- **\*\*setupkvm\*\* \*(sets up kernel page table)\***

- **\*\*mappages\*\* \*(adds translations to page table)\***

- **\*\*walkpgdir\*\* \*(allocates and maps page)\***

- **\*\*kalloc\*\* \*(allocates page)\***

- **\*\*memset\*\* \*(clean new page)\***

- **\*\*seginit\*\* \*(sets up segmentation table)\***

- **\*\*tvinit\*\* \*(initializes interrupt table)\***

- **\*\*kinit2\*\* \*(frees pages)\***

- **\*\*freerange\*\* \*(frees pages)\***

- **\*\*kfree\*\* \*(frees single page)\***

- **\*\*userinit\*\* \*(initialize the First Process)\***

- **allocproc** \*(allocates new proc)\*
- **setupkvm** \*(sets up kernel page table)\*
- **mappages** \*(adds translations to page table)\*
- **walkpgdir** \*(allocates and maps page)\*
  - **kalloc** \*(allocates page)\*
  - **memset** \*(clean new page)\*
- **inituvm** \*(allocates and maps single page, and copies First Process code to it)\*
- **mpmain**\*
- **idtinit** \*(sets %IDTR to point at existing interrupt table)\*
- **scheduler** \*(runs runnable processes)\*
- **acquire** \*(locks process table)\*
- **pushcli** \*(makes us ignore interrupts)\*
- **switchuvm** \*(prepares proc's kernel stack and makes TSS point to it)\*
- **swtch** \*(saves current context on proc, and switch to new proc)\*
- **switchkvm** \*(switches back to kernel page table)\*

- **release** \*(unlocks process table)\*

- **popcli** \*(makes us stop ignoring interrupts)\*

---

**fork** \*(creates child process)\*

- **allocproc** \*(allocates new proc)\*

- **copyuvm** \*(copies memory)\*

- **setupkvm** \*(sets up kernel page table)\*

- **mappages** \*(adds translations to page table)

- **walkpgdir** \*(allocates and maps page)\*

- **kalloc** \*(allocates page)\*

- **memset** \*(clean new page)\*

- **walkpgdir** \*(validate that page mapping exists, without allocating or cleaning)\*

- **kalloc** \*(allocate new page for user-code)\*

- **memmove** \*(copy page data)\*

- **mappages** \*(add user-code page)\*

- **walkpgdir** \*(maps page, without allocating or cleaning)\*

- **freevm** \*(free page table in case of error)\*

- **deallocuvm** ()

- **walkpgdir** \*(get entry of internal page)\*

- **kfree** \*(free actual page)\*

- **kfree** \*(free inner table)\*

- **kfree** \*(free outer table)\*

- **kfree** \*(if error, free kernel-stack)\*

- **filedup**\*

- **idup**\*

- **safestrcpy**\*

---

# It's a Trap!

The last post introduced the mechanisms that xv6 uses for scheduling and context switches. User processes can transfer control to kernel code with system calls, potentially switching into the scheduler with `sleep()` or `exit()` to find another process to run. But there are many other system calls besides those two. Kernel code can also be invoked during hardware interrupts or software exceptions; these three together are collectively referred to as traps.

We'll go over traps now to understand them more generally. First, about the terminology: depending on the source, interrupts might mean hardware interrupts specifically or any trap generally; similarly, exceptions might mean errors arising from the code, or traps in general. It's super frustrating because it makes it really hard to know what's meant by a word like "interrupt" or "exception" in whatever specification or source you happen to be reading. So I'm gonna try my best to save you that kind of pain in this post by sticking to "interrupt" for the hardware interrupts only, "exception" for software errors, and "trap" for those two combined with system calls.

## Interrupt Descriptor Table

Imagine if, after every single time some user code carried out a division, the processor stopped, context switched into the kernel, and asked the kernel to check if there was a division by zero and handle it if necessary. Or every time a hardware interrupt happened, the kernel had to start polling all the devices to figure out which one just yelled. No. Just no. Running kernel code for all this would be way too slow.

So it's the processor that will have to detect traps and decide how to handle them. But what exactly it should do for a specific trap depends on all kinds of particulars about that OS, e.g. a disk saying it's done reading from a file might require updating some file system data or storing the disk data in a specific buffer or something. That's too much responsibility for the processor.

Okay, so the kernel will set up a bunch of handler functions for every possible type of trap. Then it tells the hardware, "Okay, so if you get a disk interrupt, here are my instructions to handle that. For timer interrupts, use these instructions. If a process tries to access an invalid page, do this..." From then on, the processor can handle the traps without further input from the kernel by looking up the interrupt number in a big table to get the trap handler function that the kernel set up, then just running it.

In the x86 architecture, that table is called the *interrupt descriptor table* or IDT. I know, I'm sorry, I promised I'd say "trap" for the general case, but the x86 specs give it the official name of IDT even though it handles all the traps. Sigh. It has 256 entries (so that's the maximum number of distinct traps we can define); each one specifies a segment descriptor (ugh segmentation again, you know what that means: opaque code) and an instruction pointer (%eip) that tell the processor where it can find the corresponding trap handler function.

xv6 won't use all 256 entries; it'll mostly use trap numbers 0-31 (software exceptions), 32-63 (hardware interrupts), and 64 (system calls), all defined in [traps.h](#). But we do have to stick all 256 in the IDT anyway, so we're the unlucky fools who get to write 256 functions' worth of assembly code by hand. Nah, just kidding: xv6 uses a script in a high-level language to do that for us and spit out the entries into an assembly file.

Unfortunately for us, that high-level language is Perl. Sigh. Perl is infamous as a "write-only" language, so I guess instead we're just the unlucky fools who get to try reading Perl.

## vectors.pl

Okay, I'm not gonna assume you know Perl, and either way I really don't wanna go over every single line of this file. The syntax is similar enough to C's (except that somehow they managed to make it even worse than C), so you can read it on your own if you want.

Now, no script will be able to generate 256 completely unique assembly functions with enough detail to handle each trap correctly, so each function in the script has to be pretty generic. They're all gonna call the same assembly helper function, which will call a C function where we can more comfortably code up how to handle each interrupt.

The gist of this Perl script is that it prints a bunch of stuff using a for loop with 256 iterations. The xv6 [Makefile](#) will run it from the command line with `./vectors.pl > vectors.S` so that the output gets saved in an assembly file, which will then get assembled together with all the other kernel code in `OBJS`.

The resulting assembly file will look like this:

```
.globl alltraps

.globl vector0
vector0:
 pushl $0
 pushl $0
 jmp alltraps

.globl vector1
vector1:
 pushl $0
 pushl $1
 jmp alltraps

.globl vector2
vector2:
 pushl $0
 pushl $2
 jmp alltraps

...

...
```

Except that a handful of entries (8, 10 through 14, and 17) will skip one line (I'll explain why below):

```
...

.globl vector8
vector8:
 pushl $8
 jmp alltraps

...
```

Then at the end, it defines an array `vectors` with each of those entries above:

```
...
```

```
.data
.globl vectors

vectors:
 .long vector0
 .long vector1
 .long vector2
 # ...
```

Okay, so those are all the handler functions; the `vectors` array holds a pointer to each one. They're all more or less the same: most of them push zero onto the stack, then all they push a *trap number* to indicate which trap just happened, and then they jump to a point in the code called `alltraps`; that's the assembly helper function I mentioned earlier.

A handful of the entries don't push zero on the stack: these are trap numbers 8 (a double fault, which happens when the processor encounters an error while handling another trap), 10 (an invalid task state segment), 11 (segment not present), 12 (a stack exception), 13 (a general protection fault), 14 (a page fault), and 17 (an alignment check). These are special because the processor will actually push an error code on the stack before calling into the corresponding handler function in `vectors`. It doesn't push any error codes on the stack for the others, so we just push 0 ourselves to make them all match up.

## trapasm.S

### alltraps

The processor needs to run the trap handler in kernel mode, which means we have to save some state for the process that's currently running so we can return to it later (similar to the `struct context` we saw before), then set things up to run in kernel mode. The `alltraps` routine does just that.

Remember how we said the IDT holds segment selectors for `%cs` and `%ss`, plus and instruction pointer `%eip`? (I know we haven't seen the code to create the IDT and store the entries of `vectors` in it yet; we'll get to that below.) The processor will start using those segments (and save the old ones) before running the trap handler function. Each trap handler function in `vectors` above pushed an error code (or 0) followed by a trap number. Now we have to push all the other segment selectors on the stack one at a time, then push all the general-purpose registers at once with the x86 instruction `pushal`.

```
.globl alltraps
alltraps:
 pushl %ds
 pushl %es
 pushl %fs
 pushl %gs
 pushal
 # ...
```

Cool, all the registers are saved now. So now we'll set up the `%ds` and `%es` registers for kernel mode (`%cs` and `%ss` were already done by the processor).

```
...
movw $(SEG_KDATA<<3), %ax
movw %ax, %ds
movw %ax, %es
...
```

Now we're ready to call the C function `trap()` that's gonna do most of the work. That function expects a single argument: a pointer to the process's saved register contents. Well, we just pushed them all on the stack, so we can just use `%esp` as that pointer.

```
...
pushl %esp
call trap
...
```

That function will return back here when it's done, so let's ignore the return value by moving the stack pointer just above it (essentially popping it off the stack).

```
...
addl $4, %esp
...
```

## trapret

We've talked about this function before; when we create a new process, it starts executing in `forkret()`, which then returns into `trapret()`. More generally, any call to `trap()` will return here as well.

This function just restores everything back to where it was before, popping stored registers off the stack in reverse order. We can skip the trap number and error code; we won't need them anymore. Then we use the `iret` or "interrupt return" (though you should read that as "trap return") instruction to close out, return to user mode, and start executing the process's instructions again.

```
.globl trapret
trapret:
 popal
 popl %gs
 popl %fs
 popl %es
 popl %ds
 addl $0x8, %esp # skip the trap number and error code
 iret
```

## trap.c

Okay, on to the main part of the code! We have to do two things here: stick the trap handler functions in `vectors` into an IDT, and figure out what to do with each interrupt type.

At the top, we've got four global variables. The IDT is represented as an array of `struct gatedescs`, defined in [mmu.h](#). It's worth taking a look at because it uses an obscure C feature (bit fields); we'll do that in the next section.

Then we declare the `vectors` array of trap handler (with an `extern` keyword, since it's defined in an assembly file), a global counter `ticks` that tracks the number of timer interrupts so far (basically a rough timer), and a lock to use with `ticks`.

```
struct gatedesc idt[256];
extern uint vectors[];
struct spinlock tickslock;
uint ticks;
// ...
```

## Bit Fields

This section will get *deep* into the weeds, so feel free to skip it if you're having a nice day and don't want to spoil it by reading about a bunch of C standards.

So far, we've used bit flags with regular integers by manually doing some bit arithmetic to set one bit at a time. For example, the flags for page table and page directory entries are defined as powers of 2 (e.g., PTE\_P is 0x1, PTE\_W is 0x2, PTE\_U is 0x4, etc.) so that we can set a specific bit using a bitwise-OR like `pte |= PTE_U` or test whether it's set with a bitwise-AND like `pte & PTE_P`.

But sometimes that can get annoying and hard to keep track of; wouldn't it be nice if we could just have variables that represent a single bit? Or two bits, or any number of bits we want?

The trouble is that most computer architectures don't work with a single bit at a time; they operate on bytes, words (2 bytes), long/double words (4 bytes), or quad words (8 bytes), so it would be nontrivial to compile a line of C like `a = 1` if `a` is a nonstandard size.

In fact, accessing variables that aren't aligned to a standard size (4 bytes on x86 or 8 bytes on x86\_64) is much slower than when they are aligned. Compilers often optimize code to correct for this by padding `structs` so that they'll line up along those standard sizes. For example, one like

```
struct nopadding {
 int n;
};
```

is probably left the same on x86, but one like this:

```
struct padding {
 char a;
 int n;
 char b;
};
```

is probably converted by the compiler into this:

```
struct padding {
 char a;
 char pad0[3];
 int n;
 char b;
 char pad1[3];
};
```

**WARNING:** We're entering the dark arts of C's unspecified and implementation-defined behavior here. Note that these are different from *undefined* behavior: undefined behavior means you did something BAD BAD BAD like dereferencing a null pointer, freeing a memory region twice, using a variable after freeing it, accessing an out-of-bounds index in a buffer, or overflowing a signed data

type. Implementation-defined and unspecified behavior aren't as dangerous as undefined behavior is, but they can cause portability issues.

The C standard is a huge document with a bunch of legalese rules about what makes C, well, C. People who write C compilers need to know exactly how C code should behave under all kinds of different circumstances, so the C standard spells most of it out. But there are some parts it intentionally leaves out.

*Implementation-defined* behavior means the C standard doesn't set any fixed requirements about how a compiler should handle some behavior or feature; the developers of a C compiler get to decide how to write that part of the code with total freedom. One example is the number of bits in a byte; we've been assuming it's 8, but there are some (dumb) architectures where it's different.

*Unspecified behavior*, on the other hand, means that the C Standard provides some specific options, and compiler developers have to choose from those options for *each instance* of the behavior in the code they're compiling (that means, don't assume it's always gonna be the same, even with the same compiler).

Structure padding is implementation-defined, and there are often implementation-defined ways to modify it or disable it altogether (i.e., to *pack* the `struct` instead of *padding* it), usually with stuff like `__attribute__`s or `#pragma` directives for the preprocessor.

Wait weren't we gonna talk about bit manipulation? Why are we talking about `structs`? Well, C does have a workaround to make bit manipulation a little easier by avoiding that slightly-annoying bit arithmetic you have to do to set or clear flags in an `int` or `unsigned int`: it's called a *bit field*, and it takes advantage of `struct` padding.

You can specify the number of bits that a field of a `struct` should occupy by adding a colon and a size after the field name:

```
struct bitfield_example {
 unsigned char a : 1;
 unsigned char b : 7;
};
```

This way, you can set the single-bit flag `a` with simple variable assignments like `var.a = 1`, and the compiler will figure out any necessary magic similar to structure padding to make that happen. Awesome, right? So why haven't we been using it all the time instead of all that opaque bit arithmetic with arcane operators like `<<`, `>>`, `|`, and `&`?

Well, there are some big downsides to bit fields. First, the C standard sets some strict rules on their use to make sure that compilers can figure out how to handle them. Bit fields are only allowed inside of structures. You're not allowed to create arrays of bit fields or pointers to bit fields.

Functions aren't allowed to return a bit field. You're not allowed to get the address of a bit field with the `&` operator. You can only operate on a single bit field at a time in any statement; that means you can't set one bit field to equal another, and you can't compare the values of two bit fields.

Second, they're *extremely* implementation-defined. Each implementation (read: compiler + architecture combo) determines what data types and sizes are allowed to be used in bit fields. The data types you *can* use might have different signedness rules from the usual ones for signed and unsigned types. How they're laid out, ordered, and padded in memory can differ. In short: the low-

level details are a total black box that you can probably only figure out by reading *deep* into the compiler's specifications.

Now imagine trying to do something that requires specific protocols like sending data over a network, and you come across a bit field. Lolwut. Who knows what you'd have to do. Bit fields make it impossible to port your code.

BUT! Bit arithmetic is annoying, so let's use bit fields anyway!

Okay, so back to `struct gatedesc`. IDT entries have to contain a 16-bit code segment selector (%cs), 16 low bits and 16 high bits for an offset in that segment, the number of arguments for the handler function, a type, a system/ application flag, a descriptor privilege level (0 for kernel, 3 for user), and a "present" flag. And x86 is very particular about how it's all laid out, so we have to set up `struct gatedesc` in the exact right order.

```
struct gatedesc {
 uint off_15_0 : 16;
 uint cs : 16;
 uint args : 5;
 uint rsv1 : 3;
 uint type : 4;
 uint s : 1;
 uint dpl : 2;
 uint p : 1;
 uint off_31_16 : 16;
};
```

Well, okay, that's it for now.

## tvinit

This function loads all the assembly trap handler functions in `vectors` into the IDT. The `SETGATE()` macro in [mmu.h](#) will organize each entry correctly. We said before that the IDT needs a code segment selector, an instruction pointer (from `vectors`), and a privilege level (0 for kernel mode), so we'll stick those in.

```
void tvinit(void)
{
 for (int i = 0; i < 256; i++) {
 SETGATE(idt[i], 0, SEG_KCODE << 3, vectors[i], 0);
 }
 // ...
}
```

We're basically done now, but there's one last hiccup: user code needs to be able to generate system calls, but we just set all the privilege levels so only the kernel and processor can generate traps. So we'll fix the entry for system calls as a special case.

```
void tvinit(void)
{
 // ...
 SETGATE(idt[T_SYSCALL], 1, SEG_KCODE << 3, vectors[T_SYSCALL], DPL_USER);
 // ...
}
```

Oh and while we're at it, let's just go ahead and initialize the lock for the tick counter.

```

void tvinit(void)
{
 // ...
 initlock(&tickslock, "time");
}

```

## idtinit

The last function stored all the trap vectors in the IDT, so now we need to tell the processor where to find the IDT. There's a special assembly instruction for that in x86 called `lidt`.

```

void idtinit(void)
{
 lidt(idt, sizeof(idt));
}

```

## trap

This last function is the one that gets called by the assembly code in `alltraps`; it's responsible for figuring out what to do based on the trap number we pushed on the stack before. Heads up: it's gonna do that by calling a bunch of other functions, many of which we haven't seen yet. I'll just give a quick summary when we come across them, and we'll get to them later on.

The only argument is a pointer to a `struct trapframe`. Wait, hang on. Up above in the assembly code, the argument we pushed on the stack was `%esp`, the stack pointer, not a pointer to any `struct trapframe`. What's up with that? Did we pass the wrong kind of argument in?

Let's check out the definition for `struct trapframe`, found in [x86.h](#). It's got a bunch of fields, starting off with the general purpose registers (those are the fields from `%edi` to `%eax`). Then it has four segment registers (fields `%gs` through `%ds`), plus some unused padding bits in between them to round the 16-bit segment registers up to 32 bits. The next two fields are a trap number and an error code.

All that should sound familiar. Take another look at [trapasm.S](#): so far, those are the exact same things we pushed on the stack! The other fields are what the processor pushed on the stack before calling the handler function in the IDT. So basically, we're never gonna construct a `struct trapframe` in C code; we already constructed it manually in assembly. It just describes everything that's already on the stack by the time this `trap()` function gets called. In that sense, the `%esp` we pushed as an argument really *is* a pointer to a `struct trapframe`. It's a clever way to read values off the stack.

So we said we're gonna check the trap number and decide which kernel function to call based on that, right? Let's start by checking if the trap number indicates this is a system call (trap number 64, or `T_SYSCALL`).

```

void trap(struct trapframe *tf)
{
 if (tf->trapno == T_SYSCALL) {
 // ...
 }
 // ...
}

```

Well how should we handle system calls? xv6 will have several, and we don't even know what they all are yet. So let's procrastinate again and just call some other function `syscall()` to handle the work of figuring out which system call to execute. Now we'll store the pointer to the `struct trapframe` in that process's `struct proc`, obtained with a call to `myproc()`. Also, processes need to be killed once they're done, or if they cause an exception; that happens by setting a `killed` flag in the `struct proc`. So we'll check for that before and after carrying out the system call and close the process out with `exit()` if it's due to be killed.

```
void trap(struct trapframe *tf)
{
 if (tf->trapno == T_SYSCALL) {
 if (myproc()->killed) {
 exit();
 }
 myproc()->tf = tf;
 syscall();
 if (myproc()->killed) {
 exit();
 }
 return;
 }
 // ...
}
```

Okay, now we have all the other trap numbers to think about. We could do them with a ton of `if` statements, but that would be a pain; we'll use a `switch` statement instead. If you haven't seen `switch` statements, they replace big `if-else` blocks with cases instead. The cases can only be indexed by integers, and you have to stick a `break` statement at the end or else you'll fall through to the next case and execute the code found there as well. (To be honest, I don't see a reason why the system call case wasn't just included in this same switch statement; if you see a reason for that, let me know.)

```
void trap(struct trapframe *tf)
{
 // ...
 switch (tf->trapno) {
 // cases go here
 }
 // ...
}
```

First up is the trap number for timer interrupts; the main function of timer interrupts is to schedule a new process, but that will come further down in this function. For now, we'll just increment the `ticks` counter then call `wakeup()`, which checks if any processes went to sleep until the next tick; it'll switch to running any process it finds. There's one detail to deal with here: the system may have multiple processors, each with their own timer and interrupts. We want to use the `ticks` counter as a rough timer, but we don't know whether all the CPU timers will be synchronized, so we'll only update `ticks` using the first CPU to avoid those issues.

If you read the post on interrupt controllers then you'll be familiar with `lapiceoi()`; if you didn't (or you forgot), it just tells the local interrupt controller that we've read and acknowledged the current interrupt so it can clear it and get ready for more interrupts.

```
void trap(struct trapframe *tf)
```

```

{
 // ...
 switch (tf->trapno) {
 case T_IRQ0 + IRQ_TIMER:
 if (cpuid() == 0) {
 acquire(&tickslock);
 ticks++;
 wakeup(&ticks);
 release(&tickslock);
 }
 lapiceoi();
 break;
 // ...
 }
 // ...
}

```

Later on, we'll see some interrupt handler functions for various devices: `ideintr()` handles disk interrupts, `kbdintr()` for key presses and releases, and `uartintr()` for serial port data. We'll direct the corresponding interrupts to those functions, then acknowledge and clear them with `lapiceoi()`. Also, devices occasionally generate spurious interrupts due to hardware malfunctions; we'll either ignore them (if they're coming from the Bochs emulator) or print a message about it to the console.

```

void trap(struct trapframe *tf)
{
 // ...
 switch (tf->trapno) {
 // ...
 case T_IRQ0 + IRQ_IDE: // disk interrupt
 ideintr();
 lapiceoi();
 break;
 case T_IRQ0 + IRQ_IDE + 1: // spurious Bochs disk interrupt
 break;
 case T_IRQ0 + IRQ_KBD: // keyboard interrupt
 kbdintr();
 lapiceoi();
 break;
 case T_IRQ + 7: // spurious interrupt-no break, FALL THROUGH
 case T_IRQ + IRQ_SPURIOUS: // spurious interrupt
 cprintf("cpu%d: spurious interrupt at %x:%x\n",
 cpuid(), tf->cs, tf->eip);
 lapiceoi();
 break;
 // ...
 }
 // ...
}

```

Okay, so now we've dealt with system calls and hardware interrupts, so any other trap must be a software exception. `switch` statements allow a catch-all case with `default`, so we'll use that to catch the rest of the trap numbers. Now, this may have come from a kernel error or a misbehaving user process. We can check with `myproc()`, which returns a null pointer if we were running kernel code or a pointer to a `struct proc` if we were in user space, or by checking the current privilege level in the code segment selector. Depending on the source, we'll print out an appropriate error message and either panic (if in the kernel) or mark the process so it gets killed soon.

```

void trap(struct trapframe *tf)
{
 // ...
 switch (tf->trapno) {
 // ...
 default:
 if (myproc() == 0 || (tf->cs & 3) == 0) {
 // Kernel code exception
 cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
 tf->trapno, cpuid(), tf->eip, rcr2());
 panic("trap");
 }
 // User process exception
 cprintf("pid %d %s: trap %d err %d on cpu %d "
 "eip 0x%xx addr 0x%xx--kill proc\n",
 myproc()->pid, myproc()->name, tf->trapno,
 tf->err, cpuid(), tf->eip, rcr2());
 myproc()->killed = 1;
 }
 // ...
}

```

The reason we don't kill it immediately is because the process might be executing some kernel code right now; for example, system calls allow other interrupts and exceptions to occur while they're being handled. Killing it now might corrupt whatever it's doing. So instead we just give it the kiss of death for now and come back to finish the job later.

So next up, we'll check if this trap was generated by a user process that's due to be killed, and that process is running in ring 3. If so, we finally do the deed with `exit()`; otherwise if it's running in ring 0, it'll live for now and get killed the next time it generates a trap instead.

```

void trap(struct trapframe *tf)
{
 // ...
 if (myproc() && myproc()->killed && (tf->cs & 3) == DPL_USER) {
 exit();
 }
 // ...
}

```

Up above, the only thing a timer interrupt did was increment `ticks`. But we know a really important function of timer interrupts is to force a process to let go of the CPU and let someone else run. It's time to do that. We'll check if the process's state is `RUNNING` and the trap was a timer interrupt; if so, we call `yield()` to let another process get scheduled on this CPU.

```

void trap(struct trapframe *tf)
{
 // ...
 if (myproc() && myproc()->state == RUNNING &&
 tf->trapno == T_IRQ0 + IRQ_TIMER) {
 yield();
 }
 // ...
}

```

Now we have one last check: a process that yielded, then got picked up again later might have been marked as killed in the meantime, so if it was, we need to finish it off now. So we do the exact same check as above again, and then we're done.

```

void trap(struct trapframe *tf)
{
 // ...
 if (myproc() && myproc()->killed && (tf->cs & 3) == DPL_USER) {
 exit();
 }
}

```

Note that this function will return into `trapret` in the assembly code, which will then send it back to user mode.

## Summary

Let's take a moment to assess how much of xv6 we've already covered. Remember, the xv6 kernel has four main functions: (1) finishing the boot process that the boot loader started, (2) virtualizing resources in order to isolate processes from each other, (3) scheduling processes to run, and (4) interfacing between user processes and hardware devices. Let's take that as a checklist and go through those items now.

We've already seen some of the initialization routines that get run on boot in `main()`; most of the code there sets up virtual memory and all the hardware devices. We still have a few more devices to talk about: the keyboard, serial port, console, and disk; each of those has its own boot function that we'll need to go over in order to wrap up point (1).

On the other hand, we're already done with (2) and (3): we spent a lot of time going over virtual memory and paging, and the last post on scheduling showed us how xv6 virtualizes the CPU as well as it runs processes.

The code we saw in this post was our introduction to point (4). Traps are the primary mechanism for user processes to communicate with the hardware; the kernel coordinates that communication by setting up trap handler functions. The code we've seen here basically acts like an usher, directing traps to the right trap handler function depending on its type.

When a trap occurs (x86 instruction `int`), the processor will stop executing code, find the IDT, and looks up the entry for that trap number. The script that xv6 uses to generate the IDT entries just makes them all point to the same function `alltraps()`, which saves all the process's registers, switches into kernel mode, and calls `trap()`. Then that function uses the trap number to figure out how the kernel wants it to respond to this particular trap. So any hardware interrupt, software exception, or user system call will get funneled into the functions here before getting dispatched to some other appropriate kernel code that will know what to do with it.

We haven't finished point (4) yet, though: we have to actually see what each of those trap handler functions does. But we did see some of them: for example, we saw that a software exception either kills the process that caused it or panics if it occurred in kernel code. That already takes care of one of the three types of traps, so we're left with hardware interrupts and system calls. All the system calls got redirected to a `syscall()` function which we haven't seen yet.

We have seen how some of the hardware interrupts are dealt with: a timer interrupt increments a `ticks` counter (if it's on CPU 0), then calls `yield()` to force a process to give up the CPU until the next scheduling round. Spurious interrupts either get ignored or print a message to the console. But we've procrastinated some of the others: disk interrupts call an `ideintr()` function to handle

them, keyboard interrupts call `kdbintr()`, and serial port interrupts call `uartintr()`, none of which we've gone over.

So in order to wrap up the xv6 kernel, we still have to understand how system calls are routed in general, as well as how devices are initialized at boot and how the kernel responds to specific system calls that require use of those devices. The general system call routing mechanism is up next.

# System Calls: Routing

We said in the last post that system calls are the primary means for user processes to request some action by the kernel; system calls mediate processes' access to hardware resources.

If a user process wants to generate a system call, it starts a trap with the trap number for system calls. Then it identifies which of the various xv6 system calls it wants to do and passes any required arguments. The processor will then handle the trap instruction using the code we saw in the last post. Eventually, it'll get to the `trap()` function, which will recognize the trap number as a system call and pass it on to the `syscall()` function.

`syscall()` is itself a routing function like `trap()`; it'll figure out which system call the process created and redirect it again to the appropriate kernel code.

## `syscall.c`

All system calls use the same trap number: 64, or `T_SYSCALL`, but xv6 has multiple system calls, so we need another number for a process to identify which system call it wants to run. The convention on x86 is to use a system call number which the calling process should put in the `%eax` register, which usually holds return values. Then the kernel's handler function (here, `syscall()`) can just check `%eax` to figure out which system call to run. The system call numbers are defined in [`syscall.h`](#). There you can see that, e.g. `SYS_fork` is defined as 1, `SYS_exit` is 2, and so on.

All the system call functions are defined in other files, so we'll have to import their declarations with the `extern` keyword:

```
// ...
extern int sys_chdir(void);
extern int sys_close(void);
extern int sys_dup(void);
extern int sys_exec(void);
extern int sys_exit(void);
extern int sys_fork(void);
extern int sys_fstat(void);
extern int sys_getpid(void);
extern int sys_kill(void);
extern int sys_link(void);
extern int sys_mkdir(void);
extern int sys_mknod(void);
extern int sys_open(void);
extern int sys_pipe(void);
extern int sys_read(void);
extern int sys_sbrk(void);
extern int sys_sleep(void);
extern int sys_unlink(void);
extern int sys_wait(void);
extern int sys_write(void);
extern int sys_uptime(void);
// ...
```

Now we've got the numbers and the functions. Note that the numbers start with uppercase `SYS_` and the functions start with lowercase `sys_`, so make sure your kernel hacking adventures don't do anything like `SYS_fork()`; use `sys_fork()` instead.

We'll also need a way to map the numbers to those system call functions so that `syscall()` can call the right one depending on the number. We could use another `switch` statement like we did in `trap()`, but there are 21 system calls here, so that would get pretty long; also, each number will just call the specific function, unlike the different trap numbers which required different responses (e.g., the timer interrupt trap number didn't call any function at all). xv6 does something else this time that's much simpler and more elegant, but it uses some slightly-obscure C features, so we'll go over it carefully.

Remember function pointers from way back in the boot loader? Functions are just a set of instructions in order, loaded somewhere in the kernel's code segment, so C lets us use the function's name as a pointer to the beginning of its code in memory. So if we have a C function like `int func(char c)`, then `func` is its function pointer. We could even assign it to a variable; that variable's type would be a pointer to a function of argument type `char` and return type `int`; then we could call the function using the new pointer too. Here's an example that would print "Match!" to the screen:

```
int m = func('a');

int (*func_ptr)(char) = &func;
int n = (*func_ptr)('a');

if (m == n) {
 printf("Match!\n");
}
```

So instead of a big old `switch` statement, the `syscall()` function will use a static, global array of pointers to all the system call functions we just imported above. (Remember that the `static` keyword in front of a variable means it always occupies the same fixed place in memory.) It'll work because all the functions have the same argument type (`void`) and return type (`int`), so their pointers all have the same type and can fit inside a single array. Then we can get the right function by just using the system call number to index into the array of function pointers.

Now, we'd have to be super careful to add the function pointers into the array in the right order so that the indices match up. Even worse, there is no system call with number zero, so we'd have to skip that entry of the array. This could get complicated. Luckily, even though humans are bad at this kind of thing, computers are *really* good at it. So instead of trying to line them up by hand, we can use the array notation from `procfdump()` in the post on processes where we specified the value of each entry of an array like this with the index in square brackets, like this:

```
int arr[] = { [2] 5, [0] 1, [4] -2 };
```

The C compiler will use the indices we wrote there to figure out that the array needs 5 entries (indices 0 to 4), and entry 0 is 1, entry 2 is 5, and entry 4 is -2. Entries 1 and 3 will just be initialized to zero.

So at the end of the day, our array of pointers to system call functions looks like this:

```
// ...
static int (*syscalls[])(void) = {
 [SYS_fork] sys_fork,
 [SYS_exit] sys_exit,
 [SYS_wait] sys_wait,
 [SYS_pipe] sys_pipe,
```

```

[SYS_read] sys_read,
[SYS_kill] sys_kill,
[SYS_exec] sys_exec,
[SYS_fstat] sys_fstat,
[SYS_chdir] sys_chdir,
[SYS_dup] sys_dup,
[SYS_getpid] sys_getpid,
[SYS_sbrk] sys_sbrk,
[SYS_sleep] sys_sleep,
[SYS_uptime] sys_uptime,
[SYS_open] sys_open,
[SYS_write] sys_write,
[SYS_mknod] sys_mknod,
[SYS_unlink] sys_unlink,
[SYS_link] sys_link,
[SYS_mkdir] sys_mkdir,
[SYS_close] sys_close,
};

// ...

```

Okay great, now we're ready to route system calls to the right function.

## syscall

The first thing we need to do is get the system call number so we can figure out which function to call. We said above that the x86 convention is to store it in the `%eax` register, but we might have a problem: by the time we get to `syscall()`, the processor has already executed the code in the trap handler function for trap number `T_SYSCALL`, which sent it to `alltraps()`, which replaced all the register contents with those of `trap()`, so the system call number is probably long gone from `%eax`.

But wait, all is not lost! `alltraps()` saved all the registers in a `struct trapframe` for the current process. So we can just read the value of `%eax` from there. Whew, that was some good forward-thinking.

```

void syscall(void)
{
 struct proc *curproc = myproc();
 int num = curproc->tf->eax;
 // ...
}

```

Now we just need to do one more thing: call the function that corresponds to that number. We're gonna use the array of function pointers above, but we have to be careful: this number was given to us by a user process. A malicious user process might pass in an invalid number in the hopes of getting the kernel to carry out some undefined behavior which might lead to an easy exploit. So in order to keep up good security practices, the kernel should *always* distrust anything originating from user code and handle it carefully, preferably with three-inch-thick lead-lined gloves. So let's think about it: what might go wrong?

First of all, any entries that weren't explicitly initialized above (including the 0 entry) will have been automatically initialized to zero, i.e. a null pointer. Also, a number that's bigger than the highest system call number will make us do an out-of-bounds read from the array, thus possibly executing some arbitrary kernel code that's stored after the array in memory. So we should check that (1) the

number is greater than 0, (2) it's smaller than the number of elements in the array, and (3) the entry it points to is not a null pointer.

Finally, the `%eax` register is usually used in x86 to store return values, so we'll put the return value of the system call function there. If any of the above checks failed, we'll just print a message to the console and return -1 to indicate failure.

```
void syscall(void)
{
 // ...
 if (num > 0 && num < NELEM(syscalls) && syscalls[num]) {
 curproc->tf->eax = syscalls[num]();
 } else {
 // Invalid system call number
 cprintf("%d %s: unknown sys call %d\n", curproc->pid, curproc->name,
num);
 curproc->tf->eax = -1;
 }
}
```

The system call handler function will store its return value in `%eax`; after that, `syscall()` will return to the line below where it was called in `trap()`. After executing the rest of the code there, `trap()` will return into `trapret()`, which ends with an `iret` (interrupt return) instruction to tell the processor to switch to user mode and resume executing the process's code.

## fetchint

Take a look at the `sys_` functions we imported above: they all have argument type `void`. But if you think about it, many system calls need an argument: for example, `open()` needs to know which file to open, `chdir()` needs to know which directory to open, `kill()` needs a PID to know which process to kill, etc. So why did we make them all have argument type `void`?

The trouble is that until we get the system call number in `syscall()` above, we have no way of knowing which function we'll need. And each function takes arguments with different types, e.g. `open()` might need a string for the file to open but `kill()` might need an integer for the PID. So there's no way for the kernel to know which arguments to expect in `syscall()`, even though the arguments were already pushed on the stack. The task of recovering the arguments from the stack will have to fall to each of the `sys_` functions. But let's go ahead and make their lives a little easier by setting up some nice helper functions now.

The system call functions might take integers, strings, or pointers, so we'll need functions to fetch each of those types. `fetchint()` is one example; it takes a user virtual address (an integer argument's location in memory) and a pointer to an integer where we can store the integer we find. Then it returns 0 if it was able to find it, or -1 if it failed.

Just like `syscall()` above, we need to treat anything passed from user space with extreme caution. A user process that tries to read or write memory outside its address space will cause a segmentation fault or page fault and be killed, but the kernel has free reign over memory, so a malicious process might try to trick the kernel into doing that *for* it by putting its "argument" outside of the user's address space. So we have to start by checking that the entire 4 bytes of the integer is inside the process's address space.

```

int fetchint(uint addr, int *ip)
{
 struct proc *curproc = myproc();

 if (addr >= curproc->sz || addr + 4 > curproc->sz) {
 return -1;
 }
 // ...
}

```

Now we can just cast the address to a pointer, dereference it, and store the value in `*ip`.

```

int fetchint(uint addr, int *ip)
{
 // ...
 *ip = *(int *)(addr);
 return 0;
}

```

Note that we can use an address like `addr` which will be in the lower half of memory because traps don't perform a full context switch, so we're still using the process's page directory even though we're in kernel mode (ring 0). If we had switched to a kernel page directory, we'd have to call `walkpgdir()` or `uva2ka()` to figure out the corresponding kernel virtual address for `addr`.

Now hopefully, if you've taken anything away from my past rants about undefined behavior in C, you noticed something wrong with this function. If you didn't, take another look; I'll wait.

Did you see it? We're dereferencing `addr` without checking that it's not null, so if the user passed in a null address, we'd dereference a null pointer! We also dereference `ip` without a similar check, but at the very least `ip` is passed in by the kernel.

This could be very dangerous -- in general, it's undefined behavior in C, but now that we've seen the code for handling traps, we're actually at a point where we can figure out what would happen in xv6 if a null pointer gets dereferenced, so let's take the opportunity to think about it for a bit.

First, what would happen if the kernel dereferenced a null pointer? Well, if the kernel is currently using `kpgdir` as a page directory, the address 0 isn't mapped to anything, so when the paging hardware goes to figure out which physical address corresponds to the kernel virtual address 0, it would fail and generate a "General Protection Fault" (trap number 13, or `T_GPFLT`). That would start running the trap handler code, which would eventually get to the `switch` statement in `trap()` (see the last post). Trap number 13 would fall under the `default` case, and the `if` statement there would recognize that it originated in the kernel. So it would print an error message to the console, then panic.

Okay, what if we're using a process's page directory, e.g. during a system call? Address 0 is in the lower half of memory, so it's a user virtual address. The result will depend on whether that page and its page table are mapped in the process's page directory. If they are, then dereferencing a null pointer might be fine after all. But if they're not mapped, dereferencing a null pointer will cause a General Protection Fault. This time, `trap()` would print an error message to the console, then mark the process to be killed.

Now, killing a process or causing a kernel panic might not sound like a huge deal. In fact, xv6 does a great job here by killing a process that might have dereferenced a null pointer or caused the kernel

to do so. A kernel panic would be much worse -- think about how annoying it would be if that PDF you downloaded from that one sketchy website installed some malware that made your kernel panic all the time -- the OS would become unusable. In fact, this is an example of a "denial of service" vulnerability -- a malicious process might not be able to read or write arbitrary memory or execute arbitrary code, but it can still keep you from using your machine the way you expect to.

Just like `uva2ka()`, this function will only get called by one other function (we'll see it soon), so it just so happens that under the current xv6 code, it'll all be okay because it should never get passed a null pointer. But everything from my rant about `uva2ka()` applies here: if you add any kernel code that calls this function, be *VERY* careful and add your own null checks.

Okay, deep breath now. /rant.

## fetchstr

Fetching a string argument is tricky too; strings in C are just pointers to an array of characters that ends in nul, i.e. '\0', so this time we have to make sure that both the pointer *and* the entire string are in the user's address space; otherwise, we could unwittingly read from some arbitrary memory location and pass the data back to the user process.

So we'll start by making sure the pointer itself is in a valid address:

```
int fetchstr(uint addr, char **pp)
{
 struct proc *curproc = myproc();

 if (addr >= curproc->sz) {
 return -1;
 }
 // ...
}
```

Now we'll store the string pointer in `*pp`. We'll also get a pointer to the end of the process's virtual address space so we can make sure the entire string is inside its bounds.

```
int fetchstr(uint addr, char **pp)
{
 // ...
 *pp = (char *) addr;
 char *ep = (char *) curproc->sz;
 // ...
}
```

How can we check if the entire string is inside user memory? Well, a string ends with a nul byte, '\0', so we just have to start scanning the memory starting from `*pp` up to `ep` until we find a zero byte. If we find one in that range, then the entire string is in user memory and we can return its length to indicate success; otherwise the string overflows past the end of the process's virtual address space, so we should return -1 to indicate failure.

```
int fetchstr(uint addr, char **pp)
{
 // ...

 // Scan for a nul byte inside process's address space
 for (char *s = *pp; s < ep; s++) {
 if (*s == 0) {
```

```

 // If nul byte found, return the length
 return s - *pp;
 }
}
// String is not nul-terminated inside process's memory, so report failure
return -1;
}

```

Note that again, we're dereferencing `pp` and `addr` without any null checks (where `addr` is definitely the bigger concern, since it's user-generated), and again, it's gonna work out okay (a misbehaving process will just get killed), but once more: be careful if you use this function for your own kernel hacks.

## **argint**

This is the main function that the `sys_` system call functions will use to recover an integer argument; it's basically just a wrapper for `fetchint()`. The arguments are an integer `n` to say we want the `n`th integer argument, and a pointer `ip` to store the recovered argument in. We have to call `fetchint()` with an address argument, so the main task now is to figure out where in memory the `n`th integer argument should be.

We're gonna have to use the x86 function call conventions again. Remember how whenever we call a function in x86, its arguments get pushed onto the stack in reverse order (i.e., from right to left), so that the first argument is at the top of the stack (i.e., lowest memory address)? Then we push a return address (`%eip`) and the old stack base pointer `%ebp`. Normally, the stack pointer would just keep going on to the next slot on the callee's stack, but in this case the code in `alltraps()` saved all the registers (including the stack pointer `%esp`) in a `struct trapframe` before calling `trap()` or `syscall()`.

That means we can recover the old value of `%esp` from the trap frame and look one spots below that on the stack (i.e., 4 bytes higher in memory, since `ints` are 4 bytes) to get the first (`n = 0`) argument. The second argument (`n = 1`) would be 8 bytes higher than `%esp`, and so on. Pretty neat.

Okay, now that we've got that down, the code for this function is pretty straightforward.

```

int argint(int n, int *ip)
{
 return fetchint((myproc()->tf->esp) + 4 + 4*n, ip);
}

```

## **argptr**

Some of the system call functions will have pointer arguments, so this function recovers them. Pointers are 4 bytes in x86, so we can use `argint()` to get the pointer itself before performing some additional checks to make sure the pointer and the address it points to are valid.

The arguments are `n` (to retrieve the `n`th function argument), a pointer `pp` to an address where we can store the retrieved pointer, and the size of the block of memory that the retrieved pointer points to.

Let's start off by just retrieving the value of the pointer as an integer using `argint()`; that'll make sure that the number `n` is valid.

```
int argptr(int n, char **pp, int size)
{
 struct proc *curproc = myproc();

 int i;
 if (argint(n, &i) < 0) {
 return -1;
 }
 // ...
}
```

Now we have to make sure that the pointer we just retrieved is itself valid, i.e. that the size is nonnegative and the beginning and end of the memory block it points to are both within the process's address space.

```
int argptr(int n, char **pp, int size)
{
 // ...
 if (size < 0 || (uint) i >= curproc->sz || (uint) i + size > curproc->sz) {
 return -1;
 }
 // ...
}
```

Finally, we can store the pointer in `*pp` and return 0.

```
int argptr(int n, char **pp, int size)
{
 // ...
 *pp = (char *) i;
 return 0;
}
```

## argstr

A string is just a pointer in C, so we can recover the pointer's value using `argint()` again, then pass it to `fetchstr()`. The former will make sure `n` is valid, and the latter will make sure the string is nul-terminated and resides entirely in the process's address space.

```
int argstr(int n, char **pp)
{
 int addr;
 if (argint(n, &addr) < 0) {
 return -1;
 }
 return fetchstr(addr, pp);
}
```

## sysproc.c

So now we know how `syscall()` will route a system call trap to the right `sys_` function, and we've seen how those functions can recover arguments from the process's stack. Let's see some examples in action; most of these will be simple wrapper functions.

## sys\_fork

All the hard work here is gonna be done by `fork()`, which will create a new child process by cloning the parent process's virtual address space. We don't need any arguments for this, so we'll just call `fork()`.

```
int sys_fork(void)
{
 return fork();
}
```

## sys\_exit

`exit()` closes out a process, but it puts it in the ZOMBIE state so that the parent process can call `wait()` to find out it's done running. `exit()` should never return, so we'll add a return value here to make the compiler happy, but it should never get executed.

```
int sys_exit(void)
{
 exit();
 return 0;
}
```

## sys\_wait

This system call is the parent process's counterpart to `exit()`; it'll do as its name says and wait until the child process exits.

```
int sys_wait(void)
{
 return wait();
}
```

## sys\_kill

The `kill()` system call sounds like a more aggressive version of `exit()`: after all, we're killing another process against its will, right? But in reality it would be way too complicated to do that: the process might be running on another CPU, midway through updating some kernel data structure, or about to wake up another process that's asleep. Killing it by force might screw up a lot of other things.

So instead `kill()` just tags it with the `killed` field in its `struct proc`; eventually either the process will call `exit()` on its own, or it'll generate another trap, at which point the code in `trap()` will call `exit()` on it.

`kill()` needs an integer argument: the process ID for the process we wish to kill. So now we can see the payoff of writing those functions above.

```
int sys_kill(void)
{
 int pid;
 if (argint(0, &pid) < 0) {
 return -1;
 }
 return kill(pid);
}
```

## sys\_getpid

The `getpid()` system call is so simple that it doesn't even have another function for this `sys_getpid()` to call. We'll just return the PID for the current process.

```
int sys_getpid(void)
{
 return myproc()->pid;
}
```

## sys\_sbrk

If you're not familiar with system calls like `brk()` and `sbrk()` on Unix systems, here's what they do: they grow or shrink the virtual address space of a process. `brk()` sets its new size to a specific maximum address; `sbrk()` grows or shrinks the process by a certain size in bytes and returns its old size. They're mostly used to implement higher-level memory management functions like `malloc()`. Heh, "high-level" probably isn't high on your mind when you think of adjectives for `malloc()`, right? Anyway, xv6 only has `sbrk()`, so let's check out its `sys_` wrapper function.

We'll need an integer argument (the number of bytes to grow or shrink by), so let's grab that.

```
int sys_sbrk(void)
{
 int n;
 if (argint(0, &n) < 0) {
 return -1;
 }
 // ...
}
```

Now we can use `growproc()` from our posts on paging to grow the process by `n` bytes. But we want to return the old size, so we'll have to grab that before we change it with the call to `growproc()`.

```
int sys_sbrk(void)
{
 // ...
 int addr = myproc()->sz;

 if (growproc(n) < 0) {
 return -1;
 }

 return addr;
}
```

## sys\_sleep

The `sleep()` function is pretty interesting; we'll get to the implementation details later, but let's talk about the broad strokes now. You might be familiar with the `sleep()` system call in Unix systems; you pass it an integer (usually in milliseconds) and it puts your process to sleep (i.e., leaves it inactive or not running) for that amount of time.

However, `sleep()` plays a dual role in xv6: the kernel will call `sleep()` for processes that need to wait while something else happens, e.g. waiting for a disk to read or write data. That way the processes don't end up idly spinning in a loop or something and wasting valuable CPU time.

Implementing that is tricky; there's no way to know how long it would take for whatever condition the process is waiting on to be satisfied, so it's not like we can just stick in a random amount of time in the call to `sleep()` and hope the condition is satisfied by then. So instead the `sleep()` function will just "put a process to sleep" (read: make its state `SLEEPING` so it can't be run by the scheduler) on a *channel*, which is just an arbitrary integer. Then later on the kernel can wake up any processes sleeping on that channel. So for example, the kernel can put a process waiting on the disk to sleep using a specific channel that's assigned to the disk; then when the next disk interrupt occurs it can wake up any processes that might be sleeping on the disk channel.

Okay so that's all well and good for the kernel's use of `sleep()`. But what about the regular old `sleep()` system call? The argument is an integer that represents the number of ticks to sleep for; how are we gonna turn that into a channel to sleep on?

The answer is pretty neat (at least I think so): we'll set the channel to the address of the `ticks` counter. Remember, `ticks` is a global variable that gets incremented with every timer interrupt. Go check out the code in `trap()` again: each timer interrupt sends a wakeup call to any processes that might be sleeping on the `&ticks` channel. That should wake the process at every timer interrupt. Then we'll just stick that inside a for loop so it keeps sleeping forever until the right amount of ticks have passed.

Let's start by retrieving the integer argument, which is the number of ticks to sleep for.

```
int sys_sleep(void)
{
 int n;
 if (argint(0, &n) < 0) {
 return -1;
 }
 // ...
}
```

That argument `n` is a relative count, since a user process won't necessarily know how many ticks have already gone by. So let's get the current tick count before we put the process to sleep.

```
int sys_sleep(void)
{
 // ...
 acquire(&tickslock);
 uint ticks0 = ticks;
 // ...
}
```

Now we just have to write that while loop I mentioned above to put the process to sleep until `n` ticks have passed. Since we started counting at `ticks0`, the condition should be satisfied when `ticks - ticks0 == n`.

Two more details: first, we'll add a check inside the while loop to see if the current process has been tagged to be killed; if so, we'll just return -1 so we can hasten the process's actual death by letting it run more code so the kernel will call `exit()` on it at the next trap. Second, the function `sleep()`

takes another argument in addition to the channel: a lock. It'll release the lock for us and reacquire it before waking up so that a sleeping process doesn't hog a lock when it doesn't need it.

```
int sys_sleep(void)
{
 // ...
 while (ticks - ticks0 < n) {
 if (myproc()->killed) {
 release(&tickslock);
 return -1;
 }
 sleep(&ticks, &tickslock);
 }
 release(&tickslock);
 return 0;
}
```

## sys\_uptime

The `uptime()` system call just returns the amount of ticks that have passed since the system started. This is another one that's so simple it doesn't need another function, so we'll take care of it all here.

We just acquire the lock for `ticks`, get its current value, release the lock, and return the value we got.

```
int sys_uptime(void)
{
 acquire(&tickslock);
 uint xticks = ticks;
 release(&tickslock);
 return xticks;
}
```

## Running System Calls from User Code

We have system calls now! Well, not quite -- we still have to check out the actual functions like `exit()`, `sleep()`, `kill()`, etc. Plus, we only saw the `sys_` wrapper functions for *some* of the system calls here; the rest are in [sysfile.c](#), which we'll get to after we understand the xv6 file system.

But let's pause for a second and think about how a user process will send a system call. Like let's say you're writing some C code for a user program that will run on xv6 and you want to create a child process with `fork()`. What should you do?

Well, if you were coding for a Unix system like Linux or macOS, you'd just write a call to `fork()` in your code. But that can't be right in xv6, can it? After all, `fork()` is a kernel function, to be run in kernel mode with a current privilege level of 0. Plus, isn't it supposed to be called by `sys_fork()`? So should we call that?

None of these options will work. Well, yes, you do end up just calling `fork()`, but it's *not* the kernel function `fork()`, so if you're expecting that one, you'll be surprised when it doesn't behave the way you want it to. You won't be able to use any kernel code at all in your user program for xv6. This is a mistake I've seen a *lot* of people make in their xv6 OSTEP projects, so bear with me for a

second while I explain why you can't do it; feel free to skip the next section on the Makefile if you already know why.

## Makefile

To see why, let's check out the xv6 [Makefile](#) to see how xv6 is actually compiled, built, and run. There's a ton of stuff in there, but take a second to think about this: how do you usually run xv6? I bet it's a command like `make qemu` or `make qemu-nox`, right?

If you're not familiar with Makefiles, here's a quick primer: each command like `make qemu`, `make clean`, etc. is specified in the Makefile with a rule that looks like this:

```
mycmd: dependency1 dependency2 ...
 build_cmd1
 build_cmd2
...
```

So if I run `make mycmd`, the `make` program will check that `dependency1`, `dependency2`, etc. are up to date; if they're not, it'll update them by looking up *their* rules and executing those to update them. Then it'll execute `build_cmd1` on the shell, followed by `build_cmd2`, etc.

Okay, I know that might be confusing, so let me simply the `make qemu` command a bit to make it more readable (note that I cut a lot of stuff out here, so don't try to run xv6 with what I wrote below).

```
...
qemu: fs.img xv6.img
 qemu -drive file=fs.img,index=1 -drive file=xv6.img,index=0
...
```

This just says that in order to run `make qemu` when you type it on the terminal, the `make` program first has to make sure that both `fs.img` and `xv6.img` are fully up to date. Then once they are, it can just run the shell command `qemu` with the options `-drive file=fs.img, index=1` and `-drive file=xv6.img, index=0`. Those options are just regular flags like the ones you're probably used to with stuff like `ls -a` or `rm -rf`. In this case, they tell `qemu` to use the files `fs.img` and `xv6.img` as virtual hard drives, with `xv6.img` as disk number 0 and `fs.img` as disk number 1.

Okay, let's check out the `make` command for `xv6.img` next.

```
...
xv6.img: bootblock kernel
 # some dd commands here
...
```

Hey, that's interesting, we already saw `bootblock` in a prior post. That's the one we get when we compile the boot loader. `kernel` is, well, all the kernel code. The `dd` command is often used in Unix systems to format and set up disks; the details aren't important here, so I left them out for now. The point is that the boot loader got compiled separately from the kernel code, remember? But their machine code files get smushed into the same (virtual) disk together as `xv6.img`, which will be disk 0 when we run in `qemu`.

Not let's check out the (slightly simplified) `make` command for `fs.img`.

```

...
UPROGS = cat echo forktest grep init kill ln ls # ...

fs.img: mkfs README $(UPROGS)
 ./mkfs fs.img README $(UPROGS)
...

```

Okay, so `UPROGS` is just a list of all the user programs. Each of those gets compiled separately; e.g. if you look in their source code, you'll see each one has its own `main()` function. Then the shell command says to run `mkfs` to create a file system called `fs.img` with `README` and all the user programs as files.

The point of this detour is this: the boot loader gets compiled as a single unit, as does the entire kernel code. But the user programs are compiled one at a time. So if you write a user program for xv6, you should add it to the list in `UPROGS` (as well as in `EXTRA`) and expect it to get compiled individually and stuck onto the `fs.img` disk.

That means there's no way for a user program to call into any kernel code; the linker wouldn't even be able to match up the call to the right function. So no user program will ever be able to call functions like (the kernel's) `fork()`. Think about it: if you write a program in C and compile it to run on Linux, do you expect to have to recompile the entire Linux kernel just to run your one little program? No, right?

But certainly we can't just expect every single program ever to be totally self-contained. You also don't have to rewrite and recompile all of `malloc()` every time you write a C program. So operating systems provide libraries for users to include and call in their programs. Aha! So all we need to do in order for user processes to execute system calls is to provide a library. That library is [usys.S](#).

## usys.S

Let's trace back to the beginning of a trap. In order to execute a system call, we're supposed to send the processor an `int` instruction with a specific trap number; that would be `int 64` for system calls on xv6. We're also supposed to stick the system call number in the `%eax` register. Let's say we want to call `fork()`. According to [syscall.h](#), the system call number for `fork` is `SYS_fork`, or 1. In order to send a specific x86 instruction and manipulate individual registers, we'll have to write our system call library in assembly. Here's what it would look like for the `fork()` system call:

```

.globl fork
fork:
 movl $1, %eax
 int $64
 ret

```

Okay, that's easy enough, but we have 21 of these to write, and it would be pretty easy to make a mistake or write the wrong system call number. Let's automate it instead with a C preprocessor macro. We've seen plenty of examples of defining simple constants with `#define` directives for the preprocessor, but we haven't looked at them too closely until now.

The C preprocessor is a piece of software that edits C (or assembly) code before it's compiled. Preprocessor directives like `#define A 5` create macros that are expanded to replace every instance of `A` in the code with the number 5; directives like `#include "header.h"` expand

such that they essentially copy- paste all the code in the file header .h. We can also create function-like macros like the P2V( ) and V2P( ) macros we've used often by adding a parameter inside parentheses; unlike functions, these will be expanded *before* compilation to paste the code into every instance of its use, thus avoiding the usual overhead associated with a function call. Function-like macros are also generic, in a sense, since they don't require specifying parameter types or return types (as long as it works within the places where the macro will be used). Note that there are some drawbacks: macros aren't type-checked, they can evaluate their arguments more than once, we can't use pointers to them like we can with functions, and they can result in larger code.

We're gonna use a function-like macro here to create the assembly code for each system call function so that it gets expanded before the code is assembled. We'll use T\_SYSCALL instead of 64 in the code above, and SYS\_fork (or its equivalent for each system call) for the system call number. We'll have to replace the part after the underscore in SYS\_ with the name of the system call function; we can do that with the token-pasting operator ##, which glues two tokens together to form a single token. Also, macros must be defined on a single line, so we'll escape the newline characters with \ and end each assembly line with a semicolon.

```
#include "syscall.h"
#include "traps.h"

#define SYSCALL(name) \
.globl name; \
name: \
 movl $SYS_##name, %eax; \
 int $T_SYSCALL; \
 ret
```

Now we can just invoke the macro on the name of each function we want to create:

```
...
SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
and so on ...
```

After the preprocessor runs on the file, the result will look like this:

```
.globl fork
fork:
 movl $1, %eax
 int $64
 ret

.globl exit
exit:
 movl $2, %eax
 int $64
 ret

.globl wait
wait:
 movl $3, %eax
 int $64
 ret

and so on ...
```

Great! Now we have 21 functions for the system calls, all written in assembly. All user programs for xv6 will be compiled together with the code for these functions: see **ULIB** in the Makefile. So now, a user program can execute a system call by calling these functions, e.g. `fork()`.

## Summary

After all the preparations are handled by the trap handler functions in the IDT, `alltraps()`, and `trap()`, system calls get routed to the `syscall()` function, which uses a system call number to pick the right function out of an array. That function will have to recover any arguments to the system call before passing it on to the real system call function later on.

Next up, we'll take a look at some of those system calls; we'll leave the rest until after we go over xv6's file system.

# System Calls: Processes

In a previous post, I pointed out some of the most important functions a kernel has to fulfill. System calls take care of two of these: virtualizing resources via virtual memory and processes, and mediating communication between user-mode processes and the hardware. We'll wrap up the former now by looking at the system call functions relating to processes and scheduling.

## proc.c

### fork

Unlike some of the other functions we'll talk about in this post, `fork()` is used almost exclusively by user code as a system call; the kernel never calls it. That said, it has an extremely important role: after the first process has started, it's the only way to create more processes. It does that by copying the parent process's virtual address space into a new page directory. We haven't talked about the file system yet, but hopefully you're familiar with file I/O in Linux, so you know each process has its own list of open files and a current working directory; `fork()` will clone those as well for the child process.

Let's start off by getting a pointer to the parent process and creating a slot in the process table for the child process with `allocproc()`. Remember, that function returns a pointer to the new process's `struct proc`, but it can fail and return null (e.g., if there is no available slot in the process table, or if its call to `kalloc()` fails), so we'll need to check for that.

```
int fork(void)
{
 // Parent process
 struct proc *curproc = myproc();

 // Allocate process table slot for child process
 struct proc *np;
 if ((np = allocproc()) == 0) {
 return -1;
 }
 // ...
}
```

`allocproc()` also sets up the new process's stack so that it'll return into `forkret()`, then `trapret()`, before context switching into user mode, and sets the process's state to `EMBRYO`.

Next we need a page directory for the new child process; it should be a copy of the parent process's page directory. Luckily, we already did the hard work for this back in the virtual memory posts, so we can just use `copyuvm()` now. That function can also fail, in which case we'll free the stack that `allocproc()` created and set the child process's state back to `UNUSED`.

```
int fork(void)
{
 // ...
 if ((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0) {
 kfree(np->kstack);
 np->kstack = 0;
 np->state = UNUSED;
```

```

 return -1;
 }
 // ...
}

```

Next we'll copy the parent process's size and trap frame; the latter will make sure the child starts executing after `trapret()` with the same register contents as the parent. We'll set the child process's parent to, well, its parent (the current process).

```

int fork(void)
{
 // ...
 np->sz = curproc->sz;
 np->parent = curproc;
 *np->tf = *curproc->tf;
 // ...
}

```

The two processes will be nearly identical, so we need a way to distinguish them from user space so that a user program can give different instructions to each. xv6 follows the Unix convention that `fork()` should return the child process's PID to the parent and return 0 for the child. The parent's return value is easy; we'll just literally return the child's PID at the end. But the child didn't actually call `fork()`, so how can we set a return value that it will see?

Well, the x86 convention is for return values to be passed in the `%eax` register, right? And that register will be restored from the trap frame before switching into user mode. So we'll just store the value 0 there.

```

int fork(void)
{
 // ...
 np->tf->eax = 0;
 // ...
}

```

Next we'll copy all the parent process's open files and its current working current working directory. The files are stored in a per-process file array `curproc->ofile` of size `NOFILE`, so we can copy them over with the function `filedup()` (which we'll see later). The current working directory is in `curproc->cwd` and can be copied with `idup()`.

```

int fork(void)
{
 // ...
 for (int i = 0; i < NOFILE; i++) {
 if (curproc->ofile[i]) {
 np->ofile[i] = filedup(curproc->ofile[i]);
 }
 }
 np->cwd = idup(curproc->cwd);
 // ...
}

```

Then we'll copy the parent process's name with `safestrncpy()`, defined in [string.c](#). You might be familiar with the C standard library function `strncpy()`; this function is almost identical, except that unlike `strncpy()` it's guaranteed to nul- terminate the string it copies. If you haven't seen this kind of thing before, it's a fairly common practice to write your own safe wrappers for some of

the C standard library functions, especially the ones in `string.h` which are so often error-prone and dangerous.

```
int fork(void)
{
 // ...
 safestrcpy(np->name, curproc->name, sizeof(curproc->name));
 // ...
}
```

Finally, we'll set the child process's state to `RUNNABLE` and return its PID for the parent.

```
int fork(void)
{
 // ...
 int pid = np->pid;

 acquire(&ptable.lock);
 np->state = RUNNABLE;
 release(&ptable.lock);

 return pid;
}
```

## kill

This is one of the functions that can get called both by the kernel and as a system call. The kernel will use it to terminate malicious or buggy processes, and user code can use it as a system call to kill another process too.

We said before that killing a process immediately would present all kinds of risks (e.g. corrupting any kernel data structures it might be updating, etc.), so all we're gonna do is give it the ominous mark of death with the `p->killed` field. Then the code in `trap()` will handle the actual murder the next time the process passes through there.

The argument is a process ID number, so let's just iterate over the process table until we find a process with a matching PID; we'll return `-1` if we don't find any.

```
int kill(int pid)
{
 acquire(&ptable.lock);

 for (struct proc *p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
 if (p->pid == pid) {
 // ...
 }
 }

 release(&ptable.lock);
 return -1;
}
```

If we do find a matching process, then we'll set `p->killed`. Also, some of the calls to `sleep()` will occur inside a while loop that checks if `p->killed` has been set since the process started sleeping, so let's hasten the process's death a little by setting its state to `RUNNABLE` so it'll wake up and encounter those checks faster. There's no risk of screwing up by waking up a process too early,

since each call to `sleep()` should be in a loop that will just put it back to sleep if it's not ready to wake up yet.

```
int kill(int pid)
{
 // ...
 for (struct proc *p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
 if (p->pid == pid) {
 p->killed = 1;

 if (p->state == SLEEPING) {
 p->state = RUNNABLE;
 }

 release(&ptable.lock);
 return 0;
 }
 }
 // ...
}
```

## sleep

The last post went over the basics of `sleep()` and `wakeup()`; they act as mechanisms for *sequence coordination* or *conditional synchronization*, which allows processes to communicate with each other by sleeping while waiting for conditions to be fulfilled and waking up other processes when those conditions are satisfied.

Processes can go to sleep on a channel or wake up other processes sleeping on a channel. In many operating systems, this is achieved via channel queues or even more complex data structures, but xv6 makes it as simple as possible by simply using pointers (or equivalently, integers) as channels; the kernel can just use any convenient address as a pointer for one process to sleep on while other processes send a wakeup call using the same pointer.

This does mean that multiple processes might be sleeping on the same channel, either because they are waiting for the same condition before resuming execution or because two different `sleep()`/`wakeup()` pairs accidentally used the same channel. The result would be that a process might be woken up before the condition it's waiting for has been fulfilled. We can solve that problem by requiring every call to `sleep()` to occur inside a loop that checks the condition; that way, if a process receives a spurious wakeup call before it really should have been woken up, the loop will put it right back to sleep anyway. We saw one example of this in the `sys_sleep()` function, in which the while loop checked if the right number of ticks had passed.

A common concurrency danger with conditional synchronization in any operating system is the problem of missed wakeup calls: if the process that's supposed to send the wakeup call runs *before* the process that's supposed to sleep, it's possible that the sleeping process will never be woken up again. The problem is more general than just processes; it applies to devices too.

Imagine this scenario: a process tries to read from the disk; it'll check whether the data is ready yet and go to sleep (inside a while loop) until it is. If the disk gets to run first, then the process will just find the data ready and waiting for it, so it can continue on to use the data. If the process runs before the disk does, then it'll see the data isn't ready yet and sleep in a loop until it is; the disk will wake the process up once the data is ready.

But suppose they run at the same time, or in between each other. The process does its check and finds the data isn't ready, but before it can go to sleep, a timer interrupt or some other trap goes off and the kernel switches processes. *Then* the disk finishes reading and starts a disk interrupt that sends a wakeup call to any sleeping processes, but the process isn't sleeping yet. When the process starts running again later on, it'll go to sleep -- having already missed its wakeup call.

The problem is that the process can get interrupted between checking the condition and going to sleep, right? So why don't we just disable interrupts there with `pushcli()` and `popcli()`? add a lock there? Ah, but there's another problem: what if the disk driver is running simultaneously on another CPU? Disabling interrupts on the process's CPU wouldn't stop the other CPU from sending the disk's wakeup call too early.

Okay fine, so let's use a lock instead. The process will hold the lock while it checks the condition and sleeps, and the disk driver will have to acquire the lock before it can send its wakeup call... Can you see the problem here? If the process holds the lock while it's sleeping, the disk driver will never be able to acquire the lock in order to wake it up. That's a deadlock.

HEAD. DESK.

Ugh, okay, fine, you got me. So let's use a lock, but let's have `sleep()` release it right away, then reacquire it before waking up; that way the lock will be free while the process is sleeping so the disk driver can acquire it. Done, right? Everybody's happy?

Nope. Now we're back to the original problem: if the lock gets released inside `sleep()` before the process is actually sleeping, then the wakeup call might happen in between those and get missed.

@\*#&@#\$\*\*&@%\$!!!

So we need a lock. And we can't hold the lock while sleeping, or we'd get a deadlock. But we also can't release it before sleeping, or we might miss a wakeup call. So... ???

See, I told you: concurrency is your worst nightmare. Ever since we decided we'd like our operating systems to do more than run a single basic process at a time, we introduced all *kinds* of problems we have to reason through. Let's check out how xv6 actually writes the `sleep()` function and think through it ourselves and try to understand if it manages to solve this problem.

We'll start by making sure of two things: (1) this CPU is currently running a process and not the scheduler (which can't ever go to sleep), and (2) the caller passed in a lock (which can be any arbitrary lock).

```
void sleep(void *chan, struct spinlock *lk)
{
 struct proc *p = myproc();
 if (p == 0) {
 panic("sleep");
 }
 if (lk == 0) {
 panic("sleep without lk");
 }
 // ...
}
```

Next we need to release the lock and put the process to sleep. That will require modifying its state, so we should now acquire the lock for the process table. But if the lock that the process is already

holding *is* the process table lock, then trying to acquire it again would cause a panic, so let's add a check for that; if we're already holding it then we'll keep using it and we don't need to release it.

```
void sleep(void *chan, struct spinlock *lk)
{
 // ...
 if (lk != &ptable.lock) {
 acquire(&ptable.lock);
 release(lk);
 }
 // ...
}
```

Okay, now it's nap time for this process. We just update its channel to `chan` and its state to `SLEEPING`, then call `sched()` to perform a context switch into the scheduler so it can run a new process. We *have* to be holding the process table lock before calling `sched()`, remember?

```
void sleep(void *chan, struct spinlock *lk)
{
 // ...
 p->chan = chan;
 p->state = SLEEPING;
 sched();
 // ...
}
```

When the process wakes up later on (if indeed it turns out that the code here works and doesn't miss any wakeup calls), it'll eventually be run by the scheduler, at which point it will context switch back here. So at that point we'll reset its channel and reacquire the original lock before returning.

```
void sleep(void *chan, struct spinlock *lk)
{
 // ...
 p->chan = 0;
 if (lk != &ptable.lock) {
 release(&ptable.lock);
 acquire(lk);
 }
}
```

Okay, well I don't know about you, but I'm still not convinced that this implementation won't miss any wakeup calls. After all, we release the original lock before putting the process to sleep, right? We're holding the process table lock at that point, which at least means that interrupts are disabled, but the process that will wake this one up might already be running on another CPU and might send the wakeup signal in between releasing the original lock and updating this process's channel and state. Hmm... Well, as always, xv6 is brilliant, so we'll see how this gets solved in the code for `wakeup()`.

But wait! Before we move on, I have a warning for you about using this function in your own code when you start hacking away at xv6. Remember that when we first talked about deadlocks, we saw we can cause a deadlock if two processes acquire two locks in opposite orders? If process 1 tries to acquire lock A, then lock B, and process 2 simultaneously tries to acquire lock B, then lock A, then the end result is that process 1 will acquire lock A and process 2 will acquire lock B, but neither will be able to acquire the other lock since it's already being held.

If you look at the code above, the process that called `sleep()` must have already been holding a lock `lk`, then `sleep()` acquires `ptable.lock` before releasing `lk`. You know what that means: there's potential for a deadlock. So in order to avoid that, you should make sure that *any* lock you pass in to `sleep()` must *always* get acquired before `ptable.lock`. If any other function (or chain of function calls) could potentially acquire `ptable.lock` before `lk`, then you might end up with a deadlock. As always, the xv6 authors have been extremely careful to make sure that that never happens in the existing code, so you'll have to do the same thing for any code you add.

## wakeup

This function is short and sweet because it procrastinates all the work it has to do by pushing it off to a helper function, `wakeup1()`. It just acquires the process table lock, calls `wakeup1()`, then releases the process table lock. It has to grab that lock since it's gonna modify the process's state in the process table.

```
void wakeup(void *chan)
{
 acquire(&ptable.lock);
 wakeup1(chan);
 release(&ptable.lock);
}
```

xv6 has to use this kind of a wrapper function for the real wakeup function `wakeup1()` in order to let processes that are already holding the process table lock send wakeup calls too.

Okay, now before we go look at `wakeup1()`, let's get back to figuring out whether xv6's implementation of `sleep()` and `wakeup()` can lead to missed wakeup calls. Take a look at the code in `sleep()` again where the original lock gets released -- we have to acquire the process table lock *before* we can release the other lock. So now there are always two locks in play whenever we use `sleep()` and `wakeup()`.

Let's go back to the example of a process waiting on a disk read. The process acquires some disk-related lock first, then checks to see if the disk is done reading; if not, it'll call `sleep()` inside a while loop. If the disk driver runs now before the process gets to call `sleep()`, that's okay: the disk driver also has to acquire the same lock before calling `wakeup()`, so the disk would just end up spinning idly. Eventually, the process runs again and gets to call `sleep()`; there, it will first acquire the process table lock before releasing the original disk-related lock.

So what happens if the disk driver's code runs now? Now the disk would be able to acquire the original lock, so there's nothing stopping it from calling `wakeup()`. But the very first thing it has to do there is acquire the process table lock, which the process is already holding, so it just spins idly again! There's no way the disk driver could ever beat the process to acquiring this second lock, because the process already held the first (disk-related) lock before acquiring the second one (the process table lock). Now the process can finish going to sleep and switch into the scheduler, which will eventually release the process table lock. So then the disk driver can acquire it, release the first lock, and finally send its wakeup call.

Moral of the story? There's no way for xv6 to ever have any missed wakeup calls! The trick was to use two locks, and acquire the second before releasing the first. But coming up with that solution isn't as easy as saying "oh, just use two locks!" The solution only works because of the way the

process table lock is already being handled by so many other parts of the kernel code. For example, if the context switch into the scheduler wasn't guaranteed to release the process table lock, then the disk driver in the example would never be able to acquire it after the process goes to sleep, resulting in a deadlock. The solution works because of all the design decisions in xv6 up to this point.

## wakeup1

Okay, I'll stop fawning over the intricacies of xv6 concurrency management now so we can look at how wakeup calls actually happen. Remember, this is a separate function from `wakeup()` because sometimes the scheduler needs to send a wakeup call while it's already holding the process table lock. So we're gonna assume that every function that ever calls this is already holding it.

The implementation here is actually pretty simple now: we'll just iterate over the process table and set every single process that's sleeping on channel `chan` to `RUNNABLE`.

```
static void wakeup1(void *chan)
{
 for (struct proc *p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
 if (p->state == SLEEPING && p->chan == chan) {
 p->state = RUNNABLE;
 }
 }
}
```

Now, there might be multiple processes sleeping on this channel, so this will wake them all up. For some of those processes, this might be a spurious wakeup, so again, we should always make sure to call `sleep()` in a loop that checks for some condition to be satisfied. Even if multiple processes do have their sleep conditions satisfied, they'll have to reacquire their original lock before returning out of `sleep()`, so only one of them will do so and the others will spin until the first one is done.

Why not just wake up the first process we find that's sleeping on `chan`? Then we could avoid the extra overhead of a bunch of processes waking up, checking a condition, and going back to sleep, or even spinning idly waiting to reacquire the lock before returning. The issue is that the channels may not be unique, so there's no way to know which of all the sleeping processes is the one whose sleep condition has just been fulfilled. If we wake up the wrong process, it'll just go back to sleep, but the right process didn't wake up, so that means we've lost a wakeup call.

## exit

Okay, so we saw above that `kill()` doesn't really kill a process immediately; it shirks that responsibility and lets `exit()` handle it instead... except even `exit()` won't really fully kill a process. Whew, a process's death just keeps getting dragged out forever, doesn't it? It's starting to feel like a cheesy death scene in a tragedy; I bet the process is tired of suffering the slings and arrows of outrageous fortune by now.

But it does make sense. Think about what we have to do in order to wrap up a process and recycle its slot in the process table: we have to close out any open files and reset its current working directory, free its kernel stack and its entire page directory, then notify the parent that it's done running.

The trouble comes with freeing the kernel stack and process page directory. This function runs in kernel mode, so while the user stack in the lower half of memory will be unused now, the kernel

stack is still needed in order to keep executing the instructions for `exit()`. Also, with the exception of the times when it's running the scheduling algorithm, the kernel uses the page directory of the current process. The moment we free that page directory, the very next memory access will be to an invalid page; the CPU would trigger an exception then. That exception would eventually get routed to `exit()` again, except, oh wait, we can't even run any instructions without generating another exception, because the entire page directory and stack have been freed; that's a double fault. So then the CPU would try to handle *that* exception, which would cause the dreaded boogeyman of OS devs around the world: a triple fault. After a fault triggers a second exception, which itself triggers a third exception, the CPU just decides that the kernel in its current state doesn't have its shit together enough to keep running, so it takes over and reboots the whole system. Oops.

Okay, so let's not do that. That means we can't free the kernel stack nor the page directory until we're running on a different stack/page directory combo. That could happen in `scheduler()` while we're using the page directory `kpgdir`, or it could happen while we're running another process. xv6 does it while it's running the parent process, in the `wait()` system call. If you haven't used that in Linux before, `wait()` lets a parent process sleep until a child process is done running. xv6 will use `wait()` to finish cleaning up after an exited child process too.

Now, the very first process that starts running in xv6 (`initproc`, which loads and runs the shell) obviously has no parent process, but that's okay because that one should never exit as long as the system is up. So let's start this function off by making sure that the process that's exiting isn't the initial process.

```
void exit(void)
{
 struct proc *curproc = myproc();
 if (curproc == initproc) {
 panic("init exiting");
 }
 // ...
}
```

Next we'll close all open files and clear the current working directory; again, we haven't seen the file system functions used here, but we'll get to them soon.

```
void exit(void)
{
 // ...

 // Close all open files
 for (int fd = 0; fd < NOFILE; fd++) {
 if (curproc->ofile[fd]) {
 fileclose(curproc->ofile[fd]);
 curproc->ofile[fd] = 0;
 }
 }

 // Clear the current working directory
 begin_op();
 iput(curproc->cwd);
 end_op();
 curproc->cwd = 0;

 // ...
}
```

Now we only have one thing left to do: notify the parent process that this process has exited. If the parent process is currently sleeping in `wait()`, then we'll need to wake it up. But maybe the parent process is currently in the middle of executing other code before it gets to `wait()`; we don't want it to miss the wakeup call... oh wait, but that's okay, remember? The implementations of `sleep()` and `wakeup()/wakeup1()` guarantee that we can't miss a wakeup call as long as we're holding the right lock; `wait()` will use the process table lock for that. So let's acquire it now and send a wakeup call.

```
void exit(void)
{
 // ...
 acquire(&ptable.lock);
 wakeup1(curproc->parent);
 // ...
}
```

Now, remember that a sleeping process needs to check some condition in a loop; how can the parent process know that the child has exited? Hmm, okay, let's set the child's state to `ZOMBIE`. That'll also prevent the scheduler from trying to run it again.

Ah, but hang on a sec... what if the parent process has itself been killed, i.e. the current process has been orphaned? (Again with the melodrama...) A process can't run any more user code after `exit()`, so an undead parent process would never get to call `wait()` to clean up after its children. In that case, we'd have to find another process that could adopt a child.

So let's just solve that problem now: this process is about to shuffle off its mortal coil, so let's figure out if it has any children and pass them off to another process that can keep raising them as its own. But which process is guaranteed to live long enough to clean up after those children once they die? Ah, `initproc`, of course! That first process is immortal, so it should be able to look after any children that this process might leave behind after it makes its quietus with a bare bodkin.

So we'll iterate over the process table, looking for any processes with parent process equal to `curproc`; if we find any, we'll have `initproc` adopt them. If any of our now-abandoned children has already exited before we did, we'll send a wakeup signal to `initproc` too in case it's sleeping in `wait()`.

```
void exit(void)
{
 // ...
 for (struct proc *p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
 if (p->parent == curproc) {
 p->parent = initproc;
 if (p->state == ZOMBIE) {
 wakeup1(initproc);
 }
 }
 }
 // ...
}
```

Okay, now it's finally time for this process to find out what dreams may come in that sleep of death. We'll set its state to `ZOMBIE` and context-switch into the scheduler, never to return; if something goes wrong and the scheduler *does* return, we'll panic in order to keep this function from returning into user code again.

```

void exit(void)
{
 // ...
 curproc->state = ZOMBIE;
 sched();

 panic("zombie exit");
}

```

## wait

Like we said above, this system call lets a parent process wait for a child process to exit; it also cleans up after the child process has exited.

First, we don't even know if this process has any children, so we'll have to check by iterating through the process table and checking each process's parent to see if it matches the current process. If it does, then we'll check if it's a zombie, in which case we can clean it up and return its process ID.

We should also deal with two edge cases: first, if the process has no children at all, and second, if the process does have children but none of them are dead yet. In the first case, we'll just return -1 to report failure; in the second case we'll put the current process to sleep until one of its children exits. The `sleep()` call means we'll have to do these checks inside an infinite loop.

Alright, let's get started by getting the current process and acquiring the process table lock, then starting an infinite loop.

```

int wait(void)
{
 struct proc *curproc = myproc();

 acquire(&ptable.lock);

 for (;;) {
 // ...
 }
}

```

Inside the loop, we'll use a variable `havekids` as a boolean to track whether we've found any child processes. Then we can iterate over the process table, skipping any processes for which the current process is not the parent. If we find any children, we'll set `havekids` to 1.

```

int wait(void)
{
 // ...
 for (;;) {
 int havekids = 0;

 for (struct proc *p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
 if (p->parent != curproc) {
 continue;
 }
 havekids = 1;
 // ...
 }
 // ...
 }
}

```

If we did find a child process, we should check if it's a zombie, in which case it's time to finish its clean-up. That means freeing its kernel stack and its page directory and recycling its `struct proc` so that it can be reallocated to another process later on.

```
int wait(void)
{
 // ...
 for (;;) {
 // ...
 for (struct proc *p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
 // ...
 if (p->state == ZOMBIE) {
 int pid = p->pid;

 // Free child's kernel stack
 kfree(p->kstack);
 p->kstack = 0;

 // Free child's page directory
 freevm(p->pgdir);

 // Recycle child's struct proc
 p->pid = 0;
 p->parent = 0;
 p->name[0] = 0;
 p->killed = 0;
 p->state = UNUSED;

 release(&ptable.lock);
 return pid;
 }
 }
 // ...
 }
}
```

Now, if `havekids` is still zero by the time we finish the for loop, that means the process doesn't have any children, so we should report failure. We'll also check if the process has been marked as killed in the meantime.

```
int wait(void)
{
 // ...
 for (;;) {
 // ...
 if (!havekids || curproc->killed) {
 release(&ptable.lock);
 return -1;
 }
 // ...
 }
}
```

Finally, if it *does* have children, but none of them have exited yet, we'll put the process to sleep. It'll get woken up when a child exits, at which point it'll restart the outer for loop at the top and start looking through the process table again.

```
int wait(void)
{
 // ...
```

```
for (;;) {
 // ...
 sleep(curproc, &ptable.lock);
}
}
```

## Summary

By now, we've looked at a good chunk of the system calls available in xv6. These system calls wrap up the mechanisms that xv6 uses to create and exit processes with `fork()`, `kill()`, `exit()`, and `wait()`, and introduced `sleep()` and `wakeup()` as a means for (limited) inter-process communication.

So what's left now? The rest of the kernel code we're gonna look at will just focus on communicating with various hardware devices like the serial port, console, and keyboard. Those drivers are relatively short, but there's one device that will require a lot more work: the disk. Storing files on disk and making sure they persist across reboots require careful planning, and making files conveniently accessible to users requires an entire system of abstractions layered on top of each other, along with a whole host of file-related system calls.

# Detour: Spin-Locks

So I know I said I wasn't expecting you to have finished the OSTEP section on concurrency, but xv6 uses locks all over the place, so we're gonna have to get comfortable with them right away. Luckily, xv6 primarily uses spin-locks, which are super simple and work on bare metal; a lot of the more complex/more awesome locks that OSTEP talks about require an OS beneath them.

I'll give a brief intro to concurrency first in case you haven't made it to that part in OSTEP; then we'll turn to the spin-lock implementation in xv6.

## A Very Brief, Poor-Man's Intro to Concurrency

TL;DR: Concurrency is your worst nightmare. It'll cause bugs in the places where you least expect it, and they won't even be consistent: your code might work 95% of the time, but every once in a while it'll randomly fail and you'll have no idea why. The good news: xv6 handles it in a super-simple way, so we'll get to appreciate it as we go along. If you're like me, you might also see the code use locks when you wouldn't have thought they were needed, and then you'll come to appreciate just how clever the xv6 authors are.

First off, stop reading this and go watch the discussion of data races and locks in [the last few minutes of the CS 50 2021 lecture on SQL](#). I'm serious, go watch it right now; this post will still be here.

Okay, I'm gonna assume you've seen it now; you should have a decent sense of the main issues with data races and how locks solve them. But the CS 50 lecture skipped some details about locks: (1) what Brian (the TA) does when he finds a locked fridge, (2) how locks are implemented in code, and (3) deadlocks.

### What Does Brian Do?

Let's say process `david` is running on one thread, and it needs to use some resource (a global variable maybe, or an I/O device like the disk or console) that other threads might want to use too, so `david` acquires the lock for that resource. Then process `brian` comes along and wants to use the same resource at the same time. This could cause a data race, but luckily we've thought ahead and used a lock, so `brian` can't access it until `david` is done with it and releases the lock.

First of all, we better hope `david` remembers to release the lock; otherwise `brian` (and all other processes, even the kernel) will *never* be able to use that resource. But assuming we're smart and remembered to release it, what does the `brian` process do in the meantime?

Well, maybe `brian` has some other work to do that he can get started on in the meantime. But what would that mean for an OS? How would we know, in general, whether the lines of code that follow the use of a shared resource can be safely executed if we haven't used that resource yet? That sounds impossible to figure out without knowing ahead of time what the resource is and how it's used, so let's just go ahead and skip that idea.

Another option that's actually used often in the real world is for `brian` to stop trying and go to sleep. Maybe he can put a note on himself asking `david` to wake him up when he gets back with

the milk. So in code, that might look like `brian` signaling the OS and letting it run a different process until the lock is released. That sounds nice and all, but at this early stage in our kernel, we don't even have processes or a scheduler yet, let alone a notion of sleeping.

Okay, another option: what if `brian` just spins around in circles, or twiddles his thumbs, or does jumping jacks or whatever until `david` releases the lock? In code, that means looping over and over forever until the lock is released. That would be horribly inefficient; think of all the CPU time wasted when one process just loops over and over again while another process does something slow while holding a lock! But it's also the approach that xv6 is gonna take, because at the end of the day, our kernel is still in baby stages and beggars can't be choosers. So xv6 uses *spin-locks* with loops that only stop when we acquire a lock.

This means we should be careful when using locks to acquire them only at the last possible moment when they're absolutely needed, and release them as soon as they're no longer required, in order to limit the amount of wasted CPU cycles.

## Implementing Locks

We can implement locks as a simple boolean variable: if it's true, then someone else is using the resource behind the lock. If it's false, then it's unused and you can go ahead and take it. So an `acquire()` function sets the lock to `true` and a `release()` function sets it back to `false`. Done!

But it's not so simple: there's actually a race condition hidden in the very idea of a lock. Think about it for a second: a lock protects some shared resource, right? And a shared resource is something that more than one process wants to use? But a lock is itself a thing that more than one process wants to use... so we haven't actually gotten rid of the race condition. (FLIPS TABLE.)

We have another Catch-22 on our hands, but this time we can't get rid of it with a clever software trick like we did with the `entrypgdir`. The issue is that no matter how well we write our code, it will always require more than one step: first we have to check whether the lock is `true`, then we have to set it to `true`. But if someone else is doing the same thing at the same time, our instructions might get executed in parallel and then we'd both acquire the lock at the same time -> RACE CONDITION.

The solution will require hardware support, using *atomic* instructions -- these are hardware instructions that are indivisible; no other code can execute in between ours. One example is the x86 instruction `xchg`, which atomically reads a value from memory, updates it to a new value, and returns the old value.

Now we're good! A lock can still be a boolean variable but now `acquire` has to use `xchg`: it should get the old value while simultaneously updating it to `true`.

Atomic instructions have more overhead than regular ones, so we should only use them when they're required, like in locks, but otherwise we can stick to the regular instructions we've always used.

There's one other detail we should be careful about: a lot of the locks in xv6 protect resources that are needed by both interrupt handlers and kernel or user code. For example, we might use a process table lock to protect the list of all currently running processes; suppose some kernel code has

acquired the lock in order to run a new process. What happens if a timer interrupt goes off at that moment? The timer interrupt handler function might need to acquire the lock in order to switch processes, but it's already being held by the kernel thread. But the timer interrupt might take priority over the kernel thread and refuse to return to the kernel until it finishes executing. The result: that CPU comes to a total halt as the timer interrupt handler function spins forever, never to get the lock it so desperately needs to move on. So sad. :(

xv6 avoids this issue in a really simple way: every time we acquire a lock, we'll just disable interrupts altogether. Problem solved: now a thread can't get interrupted until it's done using the lock and releases it. This does mean that a process which grabs locks often might stick around longer than it should, since we won't have timer interrupts to tell the scheduler to swap it out with another process, but we're just gonna cross our fingers and hope that doesn't happen too often.

## Deadlocks

The last concurrency issue we need to be aware of is the problem of deadlocks. Suppose two threads each need locks A and B; this happens often, e.g. when loading a user program the kernel will need to hold a lock for the disk and another for the process table, or a process might be reading from disk and printing to the console at the same time.

Suppose they're running at the same time, and one process acquires lock A while the other one acquires lock B. If they each need the other lock to keep going, they'd spin forever waiting for it. This is a deadlock.

The way to avoid these is to make sure that, if we use more than one lock, we *always* acquire them in the same order. That way, one process would acquire lock A, the second one would be unable to acquire it and would spin, then the first process acquires lock B with no issues. When it's done, it releases both locks and the second process can continue.

This can get complicated though: if we ever acquire a lock in a function, we'd have to check any functions that that function calls to see whether they use any locks, and so on. If they do, and if the order conflicts with another chain of function calls, we'd have to refactor the code until the orders match. xv6 has been carefully written so that the lock acquisition order is always consistent.

## spinlock.c

xv6's spin-locks are set up as a `struct spinlock`, defined in [spinlock.h](#). The `locked` field acts as the boolean variable to determine whether the lock is held; the other fields are for debugging, since we can expect concurrency issues to be the one of the most common causes of bugs in the kernel code because, again, concurrency is your worst nightmare.

Note that `locked` is an `unsigned int` instead of a `bool`; C requires the standard library header `stdbool.h` in order to use the `bool` type, but on bare metal we can't assume we have a standard library to use.

## initlock

```
void initlock(struct spinlock *lk, char *name)
{
 lk->name = name;
 lk->locked = 0;
```

```

 lk->cpu = 0;
}

```

This function is pretty straightforward; it just stores the string `name` in the lock and starts it off as unlocked; the `cpu` field is 0 because no CPU is holding it yet. Next.

## pushcli and popcli

For reasons mentioned above, we need to disable interrupts whenever we're using a lock and re-enable them when we release a lock. But if we're not careful, we could end up enabling interrupts too early when we release one lock while still holding another; or if interrupts were already disabled when we acquired a lock, we could unintentionally re-enable them upon releasing it.

xv6 uses paired functions `pushcli()` and `popcli()`.

```

void pushcli(void)
{
 int eflags = readeflags();
 cli();
 if (mycpu()->ncli == 0) {
 mycpu()->intena = eflags & FL_IF;
 }
 mycpu()->ncli += 1;
}

```

`readeflags()` is a C wrapper for some x86 assembly code that reads from the `eflags` register; the 9th bit is the interrupt flag, which is set whenever interrupts are enabled. `cli` is another x86 instruction that clears that flag, thus disabling interrupts.

`mycpu()` returns a pointer to a `struct cpu` with information about the CPU running this code; we'll go over these when we talk about processes; here we increment the `ncli` field in every call to `pushcli()`. If this is the first call, we save the value of the interrupt flag in the `intena` field.

```

void popcli(void)
{
 if (readeflags() & FL_IF) {
 panic("popcli - interruptible");
 }
 if (--mycpu()->ncli < 0) {
 panic("popcli");
 }
 if (mycpu()->ncli == 0 && mycpu()->intena) {
 sti();
 }
}

```

`popcli()` first checks to make sure interrupts aren't already enabled and we're not popping without having pushed. Then it decrements the `ncli` field of the `struct cpu` for this CPU. If this is the last call to `popcli()`, it checks the `intena` field; if it was set (i.e., interrupts were enabled before the first `popcli()`), then it enables interrupts again.

Check out how these two functions are carefully written so that they're matched: it takes two calls to `popcli()` to undo two calls to `pushcli()`. Also, if interrupts were already off before the first call to `pushcli()`, they'll stay off after the last `popcli()`. Pretty neat, right?

## holding

This function checks whether this CPU is holding the lock.

```
int holding(struct spinlock *lock)
{
 pushcli();
 int r = lock->locked && lock->cpu == mycpu();
 popcli();
 return r;
}
```

Not much to talk about here; it just checks (inside calls to `pushcli()` and `popcli()`) whether the lock is being held and this is the CPU holding it. If both conditions are true it'll return 1; otherwise 0.

## acquire

The first step in this function is to disable interrupts to avoid deadlocks. We also make sure we're not already holding the lock; otherwise we'd deadlock ourselves.

```
void acquire(struct spinlock *lk)
{
 pushcli();

 if (holding(lk)) {
 panic("acquire");
 }

 // ...
}
```

Next up, we've gotta acquire the lock using the atomic `xchg` instruction, defined in [x86.h](#). Like we said before, the trick is to atomically set `locked` to 1 while returning the old value. If the returned old value is 1, that means it was already 1 before we got to it, so it's currently being held and we can't acquire it yet -- gotta spin. But if the returned old value is 0, that means the lock was free before we got to it, and our `xchg` just updated it to 1, so we've successfully acquired it. No other instruction can occur between checking the old value and updating it to the new one, so we can be confident that no one else will be holding the lock at the same time.

```
void acquire(struct spinlock *lk)
{
 // ...
 while (xchg(&lk->locked, 1) != 0)
 ;
 // ...
}
```

We do have to be careful about one other thing: compiler optimizations can get pretty wild nowadays, so the order of code on the page isn't necessarily the order it'll get compiled to or executed in. This is a critical section of code, so we need to make sure acquiring the lock forms a barrier between the code that comes before it and the code after it so any reordering doesn't cross the lock acquisition point. We can do that with a special compiler instruction:

```
void acquire(struct spinlock *lk)
{
 // ...
```

```

 __sync_synchronize();
 // ...
}

```

Finally, we'll record some info about the CPU and process holding the lock for debugging purposes. Don't worry about `mycpu()` for now, but we'll talk about `getcallerpcs()` below.

```

void acquire(struct spinlock *lk)
{
 // ...
 lk->cpu = mycpu();
 getcallerpcs(&lk, lk->pcs);
}

```

## release

Releasing a lock is a little easier than acquiring it: to acquire it, we need to check whether it's already held and update its value, with both steps together as an atomic instruction. To release it, we only have to set the value to false. That's only one instruction, so it's automatically atomic!

Well, almost, but not quite. The compiler works some serious magic behind the scenes, so there's no guarantee that a single C operation like `lk->locked = 0` will actually get compiled down to a single assembly instruction. So we're gonna have to make sure it does by writing it directly in assembly.

We start off by making sure we are already holding the lock before releasing a lock held by someone else. Then we clear the debug info stored in the lock, and tell the compiler and processor not to reorder code past the lock release.

```

void release(struct spinlock *lk)
{
 if (!holding(lk)) {
 panic("release");
 }

 lk->pcs[0] = 0;
 lk->cpu = 0;

 __sync_synchronize();

 // ...
}

```

Next we need to release the lock, i.e. an assembly instruction equivalent to `lk->locked = 0` in C. C allows in-line assembly code using the `asm` keyword. We mark it as `volatile`, which prevents the compiler from optimizing the write away and ensures it'll get written to memory. Finally, we call `popcli()` to enable interrupts again.

```

void release(struct spinlock *lk)
{
 // ...
 asm volatile("movl $0, %0" : "+m" (lk->locked) :);
 popcli();
}

```

## getcallerpcs

This function exists to store information about the current process in the lock for use in debugging. In particular, we want to record the program counters of the last 10 functions on the call stack so we can try to figure out which functions were called in which order when concurrency issues inevitably bring our world crashing down with data races, or to a grinding halt with deadlocks.

In order to get the program counters, we're gonna have to know a bit about how x86 handles function calls. The %eip register (or instruction pointer) holds the program counter, which tracks the next instruction to be executed. The %ebp register (or base pointer) holds the address of the base of the stack (i.e., its highest address, since it grows down).

When a function gets called all its arguments are pushed on the stack in reverse order, so that the first argument is at the top (lowest address) of the stack. Then the previous function's %eip is pushed on the stack, followed by its %ebp:

```
<- low addresses high addresses ->
... [new function's data] [old %ebp] [old %eip] [new arg1] [new arg2] ...
<- top of stack bottom of stack ->
```

Anyway, the point is that if we have the address of the first argument to the current function, then we can recover the contents of the previous function's %ebp and %eip registers: %eip is one spot below it on the stack and %ebp is two spots below it.

```
void getcallerpcs(void *v, uint pcs[])
{
 uint *ebp = (uint *) v - 2;
 // ...
}
```

Note the type casts here -- `v` is a pointer to the first argument, which can be of any type and size, so we use a `void *`. But both of the %eip and %ebp registers hold 32-bit pointers, so `ebp` is declared as a pointer to a `uint` (a type alias for `unsigned int`, remember?), which makes the pointer arithmetic work out nicely so that subtracting 2 returns a pointer to the right spot on the stack.

Now, what we really want is the program counter %eip, not the pointer to the stack base %ebp. But we can use the address of %ebp to make sure we haven't gone too far back in the function call history. Remember, we wanna get the program counters for the last 10 functions in the call stack, then save them in the `pcs` array.

```
void getcallerpcs(void *v, uint pcs[])
{
 // ...
 int i;
 for (i = 0; i < 10; i++) {
 // Stop if the %ebp pointer is null or out of range
 if (ebp == 0 || ebp < (uint *) KERNBASE || ebp == (uint *) 0xffffffff) {
 break;
 }
 pcs[i] = ebp[1];
 ebp = (uint *) ebp[0];
 }
 // ...
}
```

Let's talk about those last two lines: the `ebp` pointer in the code holds the location of the saved `%ebp` register, so `ebp[0]` is the value at that address (i.e., the actual value of the saved `%ebp` register) and `ebp[1]` is the value stored one spot above that, i.e. the value of the saved `%eip` register. So each iteration of the loop will get one `%eip` and store it in a `pcs` entry.

Then we update `ebp` to the actual value at the address it points to, which means `ebp` will now point to the address of the saved `%ebp` register for the function one step further back in the call chain. Okay sorry, I know that's confusing, but basically each iteration of the for loop moves us back to the function that called this function, then the function that called that one, and so on.

Okay, whew. So what happens if we break out of the for loop early because we went all the way back in the call stack? The other entries of `pcs` might hold some garbage values, so let's just make them null pointers so we know to ignore them when debugging.

```
void getcallerpcs(void *v, uint pcs[])
{
 // ...
 for (; i < 10; i++) {
 pcs[i] = 0;
 }
}
```

One last little trick: the previous for loop declared the loop variable `i` before the loop -- this means `i` will be in scope for the rest of the function body. If it had been declare inside the for loop like `for (int i = 0; ...)`, it would fall out of scope at the end of the loop. So we can keep using the same `i` in this second for loop (without an initialization statement) and know it'll hold the value it had after finishing the first for loop. If we finished all the iterations, that value will be 10; otherwise it'll be less. So we use that to clear any remaining entries of `pcs`.

## Summary

You'll learn to hate concurrency issues in C; newer languages like Rust make data races a thing of the past, though deadlocks can still rear their ugly heads. But for now, the xv6 authors have done all the dirty work for us, so we can just sit back and watch. Note, though, that even the xv6 authors say it's totally possible that something has slipped past them and the thousands of other students and instructors that have looked at xv6, so it's probable that xv6 still has some lingering race conditions. See, even the masters struggle with it. -\_-

Anyway, we saw that locks have to be implemented with hardware support using atomic instructions. C and most languages provide high-level atomics that real-world operating systems use, but the point of xv6 is elegance in simplicity, not being a total show-off, so the xv6 spin-locks just use the basic `xchg`.

We took this detour into spin-locks to make sure we all understand some basic details because we're gonna be seeing a lot of them in the rest of the kernel code. They're inefficient (because the processor just spins around waiting for the lock to be released, WHEEEEE), but we gotta make do with the machinery we've built up so far. xv6 will also use some fancier locks called sleep-locks, but we'll cross that bridge when we get to it.

# Sleep Locks

We've used plenty of spin-locks, and a previous post looked at their implementation in xv6. Spin-locks have pretty harsh performance costs: a process that's waiting to acquire a lock will just spin idly in a while loop, wasting valuable CPU time that could be used to run other processes. So far, we've only seen locks for kernel resources like the process table, page allocator, and console, for which all operations should be relatively fast, on the order of a few dozen CPU cycles at most.

Now it's time to look at the disk driver and file system implementation, and we'll need some locks there too. But disk operations are *slow* -- reading from and writing to disk might take milliseconds, which is a literal eternity for a CPU. Imagine a process hogging a spin-lock for the disk while other processes spin around and around waiting *forever* for the disk to finish writing. It would be an enormous waste!

Spin-locks were the best we could do at the time, since we didn't have any infrastructure to support more complex locks, but now we really do need a better alternative. We also have some more kernel building blocks in place relating to processes, including a bunch of system calls.

For example, we've seen the `sleep()` and `wakeup()` system calls, which let a process give up the CPU until some condition is met. Well, hang on a second -- what if that condition is that a lock is free to acquire? Then a process could sleep while another process holds the lock, and wake up when it's ready to be acquired; that would let other processes run instead of forcing a process to spin and spin. xv6 calls these *sleep-locks*, and it's time to find out how they work.

## `sleeplock.h`

If we want a process holding a sleep-lock to give up the processor in the middle of a critical section, then sleep-locks have to work well when held across context switches. They also have to leave interrupts enabled. This couldn't happen with spin-locks: it was important that they disable interrupts to prevent deadlocks and ensure a kernel thread can't get rescheduled in the middle of updating some important data structure.

Leaving interrupts on adds some extra challenges. First, we have to make sure the lock can still be acquired atomically; second, we have to make sure that any operations in the critical section can safely resume after being interrupted.

Let's solve the first problem: how can we make sure a sleep-lock will always be acquired atomically? Well, if we want to do something atomically, we already have a solution: spin-locks! So rather than reinventing the wheel, we'll just make each sleep-lock a two-tiered deal with a spin-lock to protect its acquisition.

We'll use a `locked` field just like the one all spin-locks have, but then we'll add a spin-lock to protect it. We'll also make debugging a little easier by adding a name for the lock and a field for a PID to identify which process is holding it.

```
struct sleeplock {
 uint locked;
 struct spinlock lk;
 char *name;
 int pid;
```

```
};
```

## sleeplock.c

### initsleeplock

We can initialize a sleep-lock by initializing its guard spin-lock, then adding a name for it, setting `locked` to false, and the `pid` field to zero.

```
void initsleeplock(struct sleeplock *lk, char *name)
{
 initlock(&lk->lk, "sleep lock");
 lk->name = name;
 lk->locked = 0;
 lk->pid = 0;
}
```

### acquiresleep

In order to make sure sleep-lock acquisition is atomic, we'll bookend this function by acquiring and releasing a spin-lock. This will also make sure that interrupts are disabled during this function but re-enabled when it's done. It does add some overhead in the form of spinning until this lock is free, but the code here should be relatively short and fast to execute. What we really want is to avoid spinning once the sleep-lock is acquired, i.e. spinning *after* this function is done. So we'll tolerate a little waste here.

```
void acquiresleep(struct sleeplock *lk)
{
 acquire(&lk->lk);
 // ...
 release(&lk->lk);
}
```

Okay, now we have to do the actual acquisition. We said above that we'd use the `sleep()` function to avoid wasting processor time. Hopefully you remember one important detail about `sleep()`: it must always be called inside a while loop in order to make sure that we don't miss any wakeup calls. So let's check if the sleep-lock is already being held and go to sleep if it is. We'll need a channel and a lock for `sleep()` to release, so let's use the pointer to this lock `lk` as the channel, and the outer spin-lock `lk->lk` as the lock to be released.

```
void acquiresleep(struct sleeplock *lk)
{
 // ...
 while (lk->locked) {
 sleep(lk, &lk->lk);
 }
 // ...
}
```

It's important to keep the two locks separate in your head right now: `lk` is the sleep-lock, and `lk->lk` is the spin-lock it uses to protect the sleep-lock's acquisition. Note that we're checking `lk->locked` here, *not* the spin-lock `lk->lk` -- this process is already holding `lk->lk`, but we need to acquire `lk` itself by updating `lk->locked`. Phew, try saying that ten times fast.

Now the process will go to sleep and yield the CPU until the sleep-lock is free. If multiple processes are sleeping waiting on the same sleep-lock, they will all wake up at the same time, but all of them have to reacquire `lk->lk` before returning from sleep, so only one will get to return here and complete the sleep-lock acquisition. The others will spin a bit longer, then return here only to find that `lk->locked` is already being held by another process, so the while loop will put them to sleep again.

Once the sleep-lock is free, the process can exit the while loop and claim the sleep-lock for itself.

```
void acquiresleep(struct sleeplock *lk)
{
 // ...
 lk->locked = 1;
 lk->pid = myproc()->pid;
 // ...
}
```

We don't need fancy atomic operations like `xchg` anymore, since the guarding spin-lock has already made sure that interrupts are disabled and all operations are effectively atomic. So that's all we need! Now we just release the spin-lock and return.

## releasesleep

Now that we've seen how a process acquires a sleep-lock, releasing it is easy, we just do the opposite. We'll set `lk->locked` to zero and clear the `lk->pid` field. And what's the opposite of `sleep()`? Well, `wakeup()`, of course! That will check whether there are any processes sleeping on this channel and let them know they can attempt to acquire the sleep-lock now.

```
void releasesleep(struct sleeplock *lk)
{
 acquire(&lk->lk);

 lk->locked = 0;
 lk->pid = 0;

 wakeup(lk);

 release(&lk->lk);
}
```

## holdingsleep

This function is even more simple: it just checks whether a sleep-lock is being held, and if so, whether it's being held by the current process. The first is done by just checking `lk->locked`; the second is done by checking that `lk->pid` matches the current process's PID. The result is a boolean stored in a temporary variable so we can release the guarding spin-lock before returning the result.

```
int holdingsleep(struct sleeplock *lk)
{
 acquire(&lk->lk);

 int r = lk->locked && (lk->pid == myproc()->pid);

 release(&lk->lk);
 return r;
```

}

## Summary

Okay, that wasn't too bad! It makes sense why we couldn't use sleep-locks in a kernel without system calls like `sleep()` and `wakeup()`. But xv6 already has those, so why not use them everywhere? If sleep-locks really do cut down on the wasted CPU time, can we just go back and replace all the spin-locks with sleep-locks? Then the only use for spin-locks would be as a guard for the more- sophisticated sleep-locks.

Hold your horses! It's not that easy. Sleep-locks leave interrupts enabled, so they can't be used in interrupt handler functions, or inside a critical section where a spin-lock is being used, since interrupts will be disabled (though spin- locks can be used inside sleep-lock critical sections). They also can't be used by kernel threads like the scheduler, since those aren't processes and thus can't be put to sleep.

Finally, there are some situations in which a sleep-lock might actually add *more* overhead than a spin-lock: it takes some time to put a process to sleep, schedule another process, send a wakeup call, schedule the first process again, and so on, and the process will hold the sleep-lock the entire time. If another process is waiting on the sleep-lock, it might actually end up waiting longer than with a spin-lock, although it'll wait in a sleeping state instead of a running state where it just spins in a loop.

Additionally, sleep-locks can only be used when it's safe to interrupt a process in the middle of a critical section and wake it up later. Sure, no other process can acquire the sleep-lock in the meantime, but it's still not great for time- sensitive operations like getting the current number of ticks.

So sleep-locks are great, but their applications are more limited than spin- locks. The perfect use for them is when a process needs to complete an operation atomically, but that operation itself might take a very long time. A great example of that is disk I/O, and we'll see next how xv6 puts them to use in its file system implementation.

# Scheduling

We've done a lot of talking about context switching and scheduling, but we've procrastinated looking at the code for those. It's time to fix that.

There are all kinds of advanced schedulers out there, but as we've said before, the name of the game in xv6 is simplicity, so xv6 just uses a round-robin scheduling algorithm in which it loops through the existing processes in order. Each timer interrupt will force the current process to yield the processor and perform a context switch back into the scheduler so it can run the next available process.

## swtch.S

The `struct context` we talked about in the last post is gonna be key here, so let's just look at its fields again:

```
struct context {
 uint edi;
 uint esi;
 uint ebx;
 uint ebp;
 uint eip;
};
```

The context switch function is `swtch()`; it's gonna need to save and restore processor registers, so that means it's gonna have to be written in assembly. But let's pretend it's just a C function for a second and talk about what it's going to do.

This function will save the contents of the registers on the stack as a `struct context`, then save that location as the old context. Then it'll load a new context, switch to the new stack, and restore the registers of the new context. Its declaration would look like this in C:

```
void swtch(struct context **old, struct context *new);
```

The first argument is a pointer to a pointer to a `struct context`. That double indirection might be confusing, but there's a method to this madness: C passes arguments by value, so if we used `struct context *old` and changed `old` to point to the saved context, it would be lost as soon as we returned from this function. So instead we have to use this kind of double pointer so we can set `*old` to point to the saved context. This way `old` will be lost anyway, but `*old` was changed and will persist beyond this function's return.

Note that, as we've said before, those arguments will be pushed on the stack before `swtch()` is called. So at the beginning of `swtch()`, the stack pointer `%esp` points to a return address; the argument `old` is one space (4 bytes) above that in the stack, and `new` is one space higher than that.

Okay, let's check out the assembly code now. We're gonna start by saving those arguments into registers. We can't just use any old registers here, or we might overwrite some of the data we're trying to save. But in the last post, I said x86 has a convention that the caller has to save the contents of the `%eax`, `%ecx`, and `%edx` registers, so that means we're free to overwrite them all we want since they've already been saved.

```
.globl swtch
swtch:
 movl 4(%esp), %eax
 movl 8(%esp), %edx
 # ...
```

We haven't seen this number-parenthesis notation in assembly yet, so in case you're not familiar with x86 assembly, it's just a way to add a number to the contents of a register, then treating it as a pointer and dereferencing it. So `4(%esp)` in assembly is the same as `*(esp + 4)` in C. So at this point, `%eax` holds the `struct context **old` pointer, and `%edx` holds the `struct context *new` pointer.

Now it's time to save all the fields in a `struct context` on the stack. The stack grows from high addresses to low ones, but C `structs` expect their fields to be from low to high, so we'll save them in reverse order. Oh, and hang on -- remember what's at the bottom of the stack right now, after the arguments? That's right, a return address. That's just a saved `%eip`, so that one's already done for us! We just need to save the others.

```
swtch:
 # ...
 pushl %ebp
 pushl %ebx
 pushl %esi
 pushl %edi
 # ...
```

Next we have to save a pointer to this old `struct context` into `*old`. Well, we pushed them on the stack in reverse order, right? So `%esp` already *is* pointing to it, so that's our pointer; we'll just copy it into `*old` (remember it's stored in `%eax`, and we dereference it in assembly with parentheses).

```
swtch:
 # ...
 movl %esp, (%eax)
 # ...
```

Now it's time to switch stacks to the `new` context, which we saved in `%edx`. That context must have been saved by a previous call to `swtch()`, so it also happens to be a stack pointer as well.

```
swtch:
 # ...
 movl %edx, %esp
 # ...
```

At this point, we're using the stack from `new`, which will already have its saved context at the top. So we can load the new context by popping it off the stack in reverse order into the corresponding registers. And again, just like the `call` instruction had already saved `%eip` on the stack as the return address, the `ret` (return) instruction will pop it off and restore it into `%eip` for us.

```
swtch:
 # ...
 popl %edi
 popl %esi
 popl %ebx
 popl %ebp
```

```
 ret
```

And that's it! That's a context switch in xv6.

## proc.c

And now, finally, we can look at the scheduling code. Once the kernel is done setting itself up, initializing all the devices and drivers, etc., the very last function that `main()` calls is `scheduler()`. Interrupts were disabled in the boot loader and haven't been enabled yet, so it's also the scheduler's job to enable them for the first time in xv6.

`scheduler()` never returns; it's an infinite loop that just keeps searching through the process table for a `RUNNABLE` process, then runs it. So from that point on, with the exception of interrupts and system calls, the kernel will only ever do one thing: schedule processes to run.

## scheduler

A CPU that's running the scheduler isn't running its own process. So we'll start off by setting this CPU's process pointer to null. Note that `mycpu()` requires interrupts to be disabled before it's called, but that's okay here because interrupts were disabled in the boot loader and haven't been re-enabled before the scheduler is called.

```
void scheduler(void)
{
 struct cpu *c = mycpu();
 c->proc = 0;
 // ...
}
```

The order of the next few steps is tricky, and the authors of xv6 had to be extremely careful to do them in the right order to avoid concurrency problems. We need to (1) re-enable interrupts, (2) acquire the process table's lock, and (3) create an infinite loop to iterate over the process table forever, scheduling processes along the way. To see why this is nontrivial, let's check out some different orders (with a `fake_scheduler()` function) and see what problems we get.

ATTEMPT #1: interrupts -> lock -> loop. Let's try it out.

```
void fake_scheduler1(void)
{
 // ...
 sti(); // enable interrupts
 acquire(&ptable.lock); // acquire lock
 for (;;) { // infinite scheduling loop
 // ...
 }
}
```

Interrupts have been disabled since the boot loader used `cli`, so when we call `sti()` here they'll be turned on for the first time in the kernel. At that point we'll find out if there were any interrupts waiting to be acknowledged, and possibly jump into some handler function to take care of it. Then when that's done, we'll come back here and acquire the process table's lock. Acquiring a lock disables interrupts, remember? So they're disabled again in the infinite scheduling loop (but not forever; we'll release the lock before switching to a user process). That sounds okay, right?

Not so fast! There's a hidden problem: suppose we had a situation in which none of the current processes are **RUNNABLE** -- maybe they're all blocked (or **SLEEPING**) waiting for I/O or something, which is not unlikely. In that case, the scheduler would just keep idly looping through the process table until one of them becomes **RUNNABLE** again. But if interrupts are always disabled in the loop, then this processor will never find out about, e.g., a disk interrupt saying it's done reading data which would allow a blocked process to become **RUNNABLE**. That means the process will never find out the condition it's waiting for has already happened, which means the scheduler will never find any **RUNNABLE** processes. It'll just get stuck in an infinite loop, repeatedly and desperately searching every entry of the process table. So basically, the system would freeze while the CPU pointlessly spins at top speed.

Okay okay, so that doesn't work. We'll have to periodically re-enable interrupts before disabling them again. So let's try moving the call to `sti()` inside the infinite loop so interrupts get re-enabled every once in a while.

ATTEMPT #2: lock -> loop -> interrupts.

```
void fake_scheduler2(void)
{
 // ...
 acquire(&ptable.lock); // acquire lock
 for (;;) { // infinite scheduling loop
 sti(); // temporarily enable interrupts
 // ...
 }
}
```

Problem solved, right? Actually... this one turns out to be just as bad. The call to `acquire()` disables interrupts, only for `sti()` to enable them again. There's a reason that locks disable interrupts, remember? If an interrupt occurs that switches away from `scheduler()`, then it might call a handler function that needs to access the process table lock, which is already held by `scheduler()`, so that function would spin forever in a deadlock.

So now we arrive at the correct order: we'll call *both* `sti()` and `acquire()` inside the loop, in that order. That means we'll also need a call to `release()` at the end of the loop before we try to `acquire()` again in the next iteration. We had already said we'd have to release the lock before running a process; now we'll have to acquire it again before context-switching back into the loop.

ATTEMPT #3 (the right one): loop -> interrupts -> lock. This will give us a chance to detect any outstanding interrupts in each iteration of the for loop, but before we've acquired the lock again and thus, before doing so could cause a deadlock.

```
void scheduler(void)
{
 // ...
 for (;;) {
 sti();
 acquire(&ptable.lock);
 // ... pick a process and run it ...
 release(&ptable.lock);
 }
}
```

Whew, okay. Basically, we've learned that concurrency bugs can be hard to predict and can turn seemingly-fine code into impossible-to-diagnose system crashes or freezes.

Okay, so now let's fill in the part of the loop where the scheduling algorithm goes. We'll add an inner for loop to iterate over the process table entries and stop when we find a **RUNNABLE** process.

```
void scheduler(void)
{
 // ...
 for (;;) {
 // ...
 for (struct proc *p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
 if (p->state != RUNNABLE) {
 continue;
 }
 // ... run that process ...
 }
 // ...
 }
}
```

Next, if we found a process, then we need to switch to that process's virtual address space; that is, we need to start using its page directory.

```
void scheduler(void)
{
 // ...
 for (;;) {
 // ...
 for (struct proc *p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
 // ...
 c->proc = p;
 switchuvvm(p);
 // ...
 }
 // ...
 }
}
```

Now, if we just switched to an arbitrary page directory in the middle of running other code, we might cause a bunch of problems: all the virtual addresses we're currently using for variables, functions, instructions, etc. might suddenly become invalid and point to random other places in memory. But this is where we can see some of the earlier design decisions in xv6 start to pay off: remember how `setupkvm()` made sure every single process would have the exact same mappings for the upper half of the address space, starting at `KERNBASE`? That means that if we're running in kernel mode, we can arbitrarily switch to any process's page directory and know that all of our mappings will be exactly the same. The user mappings in the lower half might be different, but the kernel side will never change. Nice!

Now we can run the process using `swtch()`.

```
void scheduler(void)
{
 // ...
 for (;;) {
 // ...
 for (struct proc *p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
 // ...
 }
 }
}
```

```

 p->state = RUNNING;
 swtch(&(c->scheduler), p->context);
 // ...
 }
 // ...
}

```

`swtch()` will *not* return here immediately; instead, it'll pick up execution wherever the process last left off, which will be in kernel mode -- if it stopped running before, it must have been due to a system call, interrupt, or exception, which would have been handled in kernel mode before calling the scheduler.

Note that this process will still be holding the process table lock when it starts running again. For example, that's the main reason for the existence of the `forkret()` function we mentioned before. This is another dangerous detail we'll have to remember, so I'm just gonna go ahead and hope you remember THIS BIG GIANT GLARING WARNING FLAG RIGHT HERE: if you do any xv6 kernel hacking, and you want to add a new system call that will let go of the CPU, then your code *must* release the process table lock at the point at which it starts executing after switching to it from the scheduler.

This is pretty dangerous; if xv6 were a big project, it would be really easy to forget that when adding more features later on. But in this case, there's no easy way to get around it; for example, we can't just release the process table lock before calling `swtch()` and reacquire it after. The problem becomes apparent if you think of locks as protecting some invariant; that invariant might be temporarily violated while you hold the lock, but it should be restored before the lock is released.

The process table protects invariants related to the process's `p->state` and `p->context` fields, e.g. that the CPU registers must hold the process's register values, that a `RUNNABLE` process must be able to be run by any idle CPU's scheduler, etc. These don't hold true while executing in `swtch()`, so we need to hold the lock then; otherwise another CPU might decide to run the process before `swtch()` is done executing.

Now, at some point that process will be done running and will give up the CPU again. Before it switches back into the scheduler, it has to acquire the process table lock again. So here's ONE MORE GIANT WARNING for good measure: you should make sure to do that too if you add your own scheduling-related system call.

Eventually, it'll switch back here with a call with the arguments in reverse, like `swtch(&(p->context), c->scheduler)`. At the point, execution of the scheduler will resume right here, so we need to switch back to using the kernel page directory `kpgdir`.

```

void scheduler(void)
{
 // ...
 for (;;) {
 // ...
 for (struct proc *p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
 // ...
 switchkvm();
 c->proc = 0;
 }
 // ...
 }
}

```

```
}
```

After that, the outer for loop just releases the lock before looping to the top again to temporarily re-enable interrupts, then acquire the lock again and check for another process to run.

## forkret

Let's take a quick look at one example of where a process might start to execute after being scheduled. All processes (whether the very first process, or any others created later through calls to `fork()`) will start running code in `forkret()`, then return from here into `trapret()`.

Most of the time, this function does just one thing: it releases the process table lock. However, there are two kernel initialization functions that have to be run from user mode, so we can't just call them from `main()` and be done with it. We need a place for a process to call them, and `forkret()` is as good a place as any. So the very first call to `forkret()` will run these two start-up functions, and the rest will ignore them.

The two functions are `iinit()` and `initlog()`, which are part of xv6's file system code; we'll get to them later on. For now, we'll just use a `static int` as a boolean and set it to false after we've run those functions once on our first pass through `forkret()`.

```
void forkret(void)
{
 static int first = 1;

 release(&phtable.lock);

 // Only gets run once, on the first call
 if (first) {
 first = 0;
 iinit(ROOTDEV);
 initlog(ROOTDEV);
 }
 // Returns into `trapret()`
}
```

Any other kernel code that switches into the scheduler (e.g., `sleep()` and `yield()`) will have a similar lock release right after returning from the scheduler.

## sched

We saw one example of code that runs after switching *away* from the scheduler, but what about code that runs before switching *to* the scheduler? Any functions that need to call into the scheduler can't just call `scheduler()`, since the scheduler probably left off last time halfway through the loop and should resume in the same place. So `sched()` handles the task of picking up the scheduler wherever it last left off.

`sched()` should be called *after* acquiring the process table lock and without holding any other locks (lest we cause a deadlock somewhere). Also, the process should not be in the `RUNNING` state anymore since we're about to stop running it. So we'll start off by checking that those are all true and that interrupts are disabled.

```
void sched(void)
{
```

```

 struct proc *p = myproc();

 if (!holding(&ptable.lock)) {
 panic("sched ptable.lock");
 }
 if (mycpu() ->ncli != 1) {
 panic("sched locks");
 }
 if (p->state == RUNNING) {
 panic("sched running");
 }
 if (readeflags() & FL_IF) {
 panic("sched interruptible");
 }
 // ...
}

```

Next, remember when the `pushcli()` and `popcli()` functions checked whether interrupts were enabled before turning them off while holding a lock? That's really a property of this kernel thread, not of this CPU, so we need to save that now. Then we can call `swtch()` to pick up where the scheduler left off (the line right after its own call to `swtch()`). This process will resume executing after that line eventually, at which point we'll restore the data about whether interrupts were enabled and let it run again.

```

void sched(void)
{
 // ...

 // Save whether interrupts were enabled before acquiring the lock
 int intena = mycpu() ->intena;

 // Perform context switch into the scheduler
 swtch(&p->context, mycpu() ->scheduler);
 // Execution will eventually resume here

 // Restore whether interrupts were enabled before
 mycpu() ->intena = intena;
}

```

## yield

Okay, let's see an example of how this all comes together now! The `yield()` function forces a process to give up the CPU for one scheduling round. For example, this will be used to handle timer interrupts later on. Now that we know how scheduling works in xv6, `yield()` is easy. We just acquire the process table lock, set the current process's state to `RUNNABLE` so it can get picked up again in the next scheduling round, and call `sched()` to switch into the scheduler. When we eventually return here, we'll just release the lock again.

```

void yield(void)
{
 acquire(&ptable.lock);

 myproc() ->state = RUNNABLE;
 sched();

 release(&ptable.lock);
}

```

## Summary

We've now seen how xv6 handles process scheduling with a super-simple round-robin algorithm. The `scheduler()` function had plenty of concurrency pitfalls, but luckily the xv6 authors took care of all the careful coding for us, so we just get to sit back and admire their work.

We also saw how context switches occur in xv6, so now we can understand how, in the previous post, `allocproc()` set up a new process with a context that would result in it starting execution in `forkret()`.

Next up, we'll look at the way xv6 handles interrupts, system calls, and software exceptions.

# Processes

It's time to turn our attention to processes in xv6! [proc.c](#) is another huge file, so I'm gonna split it up into a few posts. This one will focus on the basic functions we'll need in order to create new processes; later posts will go over scheduling and system calls.

## proc.h

I haven't spent much time on the header files in xv6, but [proc.h](#) defines some important structures we're gonna be using often, so let's just get those out of the way first.

Let's start off with the definition for `struct context`. The processor will have to switch between different processes during interrupts, system calls, exceptions, etc.; these *context switches* will require saving the contents of some of the CPU registers so that it can reload them when it switches back and resume execution where it left off. It'll save the process's context by pushing those register contents on the stack; that way the stack pointer is effectively a pointer to the context. So the fields of a `struct context` will just list all the registers that were saved on the stack.

Now, which registers do we need to save? Let's look at the full list on the [OSDev Wiki](#). We've got some general-purpose registers, the instruction pointer register `%eip`, segment registers, a flags register, control registers, and the GDT and IDT registers (x86 doesn't use the debug, test, or LDT registers).

The flags register, control registers, and GDT/IDT registers shouldn't change between processes, so we don't need to save those. What about the segment registers like `%cs`? Back when we set up segmentation, we made the segments be identity maps that would always stay the same for all processes. There are separate segments for user mode and kernel mode, but context switches will always occur in kernel mode, so the segment registers shouldn't change, and we don't need to save them either.

We should definitely save the program counter (AKA instruction pointer `%eip`), since that will point to the place in the code where we should resume execution.

The only ones left now are the general-purpose registers: the stack base pointer `%ebp` and stack pointer `%esp`, along with `%eax`, `%ebx`, `%ecx`, `%edx`, `%esi`, and `%edi`. We said above that the stack pointer `%esp` would tell us where to find the context, so that must mean we'll already have it through some other means in order to find the rest of the context, so we don't need to save it again (we'll see how we end up getting it later on). But we do need to save `%ebp`.

There's an x86 convention that the caller of a function has to save `%eax`, `%ecx`, and `%edx`, so those are already taken care of. So we'll just save the others: `%edi`, `%esi`, and `%ebx`.

We end up with this list of saved registers as the fields for `struct context`:

```
// ...
struct context {
 uint edi;
 uint esi;
 uint ebx;
 uint ebp;
```

```

 uip eip;
};

// ...

```

Next up: we might end up with a bunch of processes, some of which are currently running while others aren't. Let's set up some labels to note that. We'll definitely need a **RUNNING** label; we'll also use one called **RUNNABLE** for processes that are ready to be run the next time there's a free CPU. We also need a label for processes that are blocked waiting for something else to happen (e.g., I/O); xv6 calls this **SLEEPING**. Processes that don't exist yet will be called **UNUSED**.

There are two special moments in a process's lifecycle that we should be careful with: birth and death. When we create a new process, we'll have to do a bunch of setup before it's **RUNNABLE**; killing a process requires clean-up before it goes back to **UNUSED**. We'll call those **EMBRYO** and **ZOMBIE**, respectively.

We could use bit flags for these states or just regular integers, except then we'd have to do annoying bit arithmetic or keep track of which number represents which state. And yes, we could use a bunch of `#define` directives for the preprocessor for that, but there's a better way to do it. C lets us create data types for labels using `enums`. These don't have fields like `structs` do; they're basically just a mapping between integers and what the labels those integers represent. So it's pretty similar to using a bunch of `#define` directives, except that they're all defined neatly in a single place, so it helps us remember they're all representing the same idea. So we'll use an `enum` like this:

```

// ...
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
// ...

```

Now it's time to look at how we'll represent processes themselves together with their metadata. Let's see... what kind of unique data does each process have? We just talked about `struct contexts` and `enum procstates`; each process will have both of those.

We also talked about virtual memory for processes in a previous post, so it should also have its own page directory and stack for the kernel to use, plus a way to track the size of its virtual address space. We said then that processes are created using `fork()`, so let's add a field to point to the parent process.

We'll need a way for the kernel to refer to a process, so let's give it a unique process ID. That's not super helpful when it comes to debugging, so let's also add a name for it as a string.

The rest of the fields are for aspects we haven't seen yet but will talk about soon: a *trap frame* for interrupts and system calls, a boolean to track whether a process should be killed soon, a *channel* to be able to wake up a sleeping process, an array of open files, and a current working directory.

```

// ...
struct proc {
 uint sz; // size (in bytes) of virtual address space
 pde_t *pgdir; // page directory
 char *kstack; // kernel stack for this process
 enum procstate state; // process state
 int pid; // process ID
 struct proc *parent; // parent process
 struct trapframe *tf; // trap frame for current system call
 struct context *context; // saved register contents for context switches
 void *chan; // channel that process is sleeping on, if any
};

```

```

 int killed; // boolean: should process be killed soon?
 struct file *ofile[NFILE]; // array of open files
 struct inode *cwd; // current working directory
 char name[16]; // process name
}
// ...

```

Okay, next we'll add another structure for metadata representing each CPU.

If you read the previous post, then you know each CPU has its own local interrupt controller with a unique ID, so we'll write that down. The post about process paging talked about the TSS, so we'll need one of those per CPU, plus a GDT too.

At any point in time, a processor will be running one of: its own initialization routine (only once while the kernel is setting up), a user process (or any interrupts or system calls that come up), or a scheduler routine to run the next process. So let's add a pointer to a `struct proc`, which will be null if it's not running a process; a boolean `started` will be false until the CPU finishes its own set-up. The scheduler isn't itself a process; it uses the `kpgdir` page directory and has its own context, so we'll store that context in a field here.

Finally: remember how the spin-lock post talked about nested calls to `pushcli()` and `popcli()` tracking whether interrupts were enabled before the first call to `pushcli()`, and only enabling interrupts after the last call to `popcli()` if they were enabled before? Those were tracked with per-CPU fields `ncli` and `intena`, so we need those too.

```

struct cpu {
 uchar apicid; // ID of this CPU's local interrupt controller
 struct context *scheduler; // scheduler's context
 struct taskstate ts; // task state segment
 struct segdesc gdt[NSEGS]; // global descriptor table
 volatile uint started; // boolean: has this CPU been initialized yet?
 int ncli; // depth of pushcli() nesting
 int intena; // were interrupts enabled before pushcli()?
 struct proc *proc; // currently running process
};
// ...

```

Last but not least, we'll add declarations for the global array of CPUs and the number of CPUs actually present on this machine; these were defined in [mp.c](#)

Okay, on to the functions now!

## proc.c

xv6 uses a global process table with an array of processes to store all the `struct procs` in; this means we'll never be able to create more processes than the number of entries in the array, `NPROC`, defined in [param.h](#) as 64. We'll need a lock too to prevent data races while accessing the process table. The process table's definition does that thing again where you simultaneously define a `struct type` and define a variable using that type in a single statement.

```

struct {
 struct spinlock lock;
 struct proc proc[NPROC];
} ptable;
// ...

```

Then we define a global static variable to point to the first process that gets run on xv6, so that other files can set it up.

```
// ...
static struct proc *initproc;
// ...
```

Finally, we're gonna need to assign unique process IDs, so we'll use a global counter to know which one we should use next.

```
// ...
int nextpid = 1;
// ...
```

## pinit

This function only does one thing: initializes the lock in the process table.

```
void pinit(void)
{
 initlock(&ptable.lock, "ptable");
}
```

## mycpu

This function will return a pointer to the `struct cpu` for the current CPU. There's a potential concurrency bug with this function: if it gets interrupted before it returns, then it might get rescheduled on a different CPU, and end up returning an incorrect `struct cpu`. So we need to make sure that interrupts are disabled when we call it. Normally we'd do that with `pushcli()` and `popcli()`, but those functions actually call this one, so we'd get an infinite recursion. So instead we're just gonna have to remember to disable interrupts *before* calling this function.

If you're reading this because you're gonna do some xv6 kernel hacking for an OSTEP project or something, you should read that as "DANGER DANGER DANGER!". If your code calls this function, or calls any other functions that in turn call this one, you *have* to make sure you've disabled interrupts first.

Concurrency bugs are a nightmare because they're not deterministic: for example, if you forget to disable interrupts before calling this function, it might work just fine most of the time until the one unlucky moment when it gets interrupted and rescheduled on a different CPU. So let's make this easier to debug by starting off with a check that interrupts are disabled and panic if they're not. We can check whether the interrupt flag `FL_IF` is set in the `eflags` register.

```
struct cpu *mycpu(void)
{
 if (readeflags() & FL_IF) {
 panic("mycpu called with interrupts enabled\n");
 }
 // ...
}
```

Okay so how do we figure out which CPU we're on? Well, the previous post talked about interrupt controllers; each CPU has a local interrupt controller with a unique ID which we can get with

`lapicid()`. Once we have that, we can iterate over the CPU array `cpus` until we find an entry with a matching `apicid`; we'll just panic if none of them match.

```
struct cpu *mycpu(void)
{
 // ...
 int apicid = lapicid();

 for (int i = 0; i < ncpu; ++i) {
 if (cpus[i].apidid == apicid) {
 return &cpus[i];
 }
 }
 panic("unknown apicid\n");
}
```

## cpuid

Those local interrupt controller IDs aren't guaranteed to start from 0, so we'll need another way to identify CPUs. We can just use its entry number in the global `cpus` array for that; `cpus` is an array of `struct cpus`, which in C means it's really a pointer to the entry with index 0. `mycpu()` returns a pointer to the entry for the current CPU, so we can just subtract those pointers to get the index.

```
int cpuid(void)
{
 return mycpu() - cpus;
}
```

## myproc

This function returns a pointer to the `struct proc` running on this CPU. We're gonna call `mycpu()` here, so we'll be good and remember to disable interrupts first with `pushcli()` and reenable them at the end with `popcli()`. Then we'll get the current process from the `struct cpu`'s field.

```
struct proc *myproc(void)
{
 pushcli();

 struct cpu *c = mycpu();
 struct proc *p = c->proc;

 popcli();
 return p;
}
```

## allocproc

Okay, we're finally at the code to create a new process! Whew, it's been a long journey.

This is a `static` function, which means it can only be called by functions defined in this same file. Creating a new process will require modifying the process table, so we need to grab the lock so that other threads can't mess with it while we're using it.

```
static struct proc *allocproc(void)
```

```

{
 acquire(&ptable.lock);
 // ...
}

```

Now we need to look through the table and find a slot that's **UNUSED**; if we find one, then great, we'll assign that slot to the new process after the **found** label below. But if none of them are free, we'll have to return a null pointer to indicate that. You know what that means, right? Yup, we're gonna have to add null checks every time we call this function! Wooooo!

```

static struct proc *allocproc(void)
{
 // ...
 struct proc *p;

 // Look through process table looking for an UNUSED slot
 for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
 if (p->state == UNUSED) {
 goto found;
 }
 }
 // If none is found, return null pointer
 release(&ptable.lock);
 return 0;

 // ...
}

```

Check out that for loop too: **p** is a pointer to a **struct proc** that starts off pointing to **ptable.proc**; that means it points to the entry at index 0. Then it gets incremented by 1 each iteration; since it's a **struct proc**, the pointer arithmetic will work out so that it points to the next entry in the process table.

Okay now let's check out the **found** label and see what happens if we did find an unused slot. First we set its state to **EMBRYO** (instead of **RUNNABLE**, since we're not done setting it up) and give it a PID. That state means it's neither **UNUSED** nor **RUNNABLE**, so we can be confident that any other threads wouldn't try messing with it right now; they can't allocate the slot to another process, and they can't try to run it yet. So we can stop hogging the process table now and let other threads take a turn.

```

static struct proc *allocproc(void)
{
 // ...
found:
 p->state = EMBRYO;
 p->pid = nextpid++;

 release(&ptable.lock);

 // ...
}

```

Now we need to allocate a page for this process's kernel thread to use as a stack. Remember, **kalloc()** can return null, so we need a null check here.

```

static struct proc *allocproc(void)
{

```

```

// ...
found:
// ...
if ((p->kstack = kalloc()) == 0) {
 p->state = UNUSED;
 return 0;
}
// ...
}

```

Now, we're not gonna set up its page directory yet; that'll happen in `fork()`, which we'll see later on. But we do need to set up the process so that it'll start executing code somewhere. It needs to start off in kernel mode, then it'll context-switch back into user mode and start running its code.

We haven't looked at the mechanics of context switches yet, so I'll spoil it a little now (I know, I'm sorry). When a process is already running, it can send a system call to ask for the kernel's attention to do whatever it needs, like a baby crying until it gets fed or changed or burped or whatever. Then it'll switch into kernel mode to run the system call, then switch back to where it left off and pick up from there.

Well, xv6 is all about simplicity, right? And what's more simple and elegant than treating a special case (creating a new process and starting it off running some code) the same as the general case (returning from a system call)? So xv6 will set up every new process to start off by "returning" from a (non-existent) system call. That way the context switch code can be reused for new processes too.

New processes are created via `fork()`, so we'll return into a function called `forkret()`. Then that has to return into the function `trapret()`, which closes out a *trap* (interrupt, system call, or exception) by restoring saved registers and switching into user mode. We'll get to `forkret()` and `trapret()` soon.

But first, the challenge: how do we "return" into a function that never called us in the first place? We talked about function calls in x86 in the post on spin-locks with the `getcallerpcs()` function, so make sure to read that now if you need a refresher.

To summarize: when a function `f()` calls another function `g()`, it pushes the arguments of `g()` on the top of its stack. Then it pushes a return address to know where it should continue running the code of `f()` after `g()` returns; that's just the `%eip` register. Then it pushes the base address of the stack for `f()`, i.e. the current `%ebp` register. That's where `g()`'s stack will start off.

When the scheduler first runs the new process, it'll check its context via `p->context` to get its register contents, including the instruction pointer `%eip`. So if we want it to start executing the code in `forkret()`, the `eip` field of its context should point to the beginning of `forkret()`. Then we can trick it into thinking that the previous caller was `trapret()` by setting up arguments and a return address in its stack.

Let's start off by getting a pointer to the bottom of the stack. We had just allocated a new stack page at `p->kstack`, but the stack grows from high to low addresses, so the base of the stack is really at `p->kstack + KSTACKSIZE`. We'll make it a `char *` so we can move around one byte at a time using pointer arithmetic.

```

static struct proc *allocproc(void)
{

```

```

// ...
found:
// ...
char *sp = p->stack + KSTACKSIZE;
// ...
}

```

Now we should push any arguments for `trapret()` on the stack; it takes a `struct trapframe` (which we'll go over later), so we'll leave some room for it and make the process point to it with `p->tf`.

```

static struct proc *allocproc(void)
{
 // ...
found:
 // ...
 sp -= sizeof(*p->tf);
 p->tf = (struct trapframe *) sp;
 // ...
}

```

Then we add a "return address" to the beginning of `trapret()` after that.

```

static struct proc *allocproc(void)
{
 // ...
found:
 // ...
 sp -= 4;
 *((uint *) sp) = (uint) trapret;
 // ...
}

```

The last thing we need is to save some space for the process's context on the stack and point `p->context` to it. Then we'll zero it all out, except for the `eip` field, which will point to the beginning of `forkret()`. And that's it! We just return the pointer to the process now.

```

static struct proc *allocproc(void)
{
 // ...
found:
 // ...
 sp -= sizeof(*p->context);
 p->context = (struct context *) sp;

 memset(p->context, 0, sizeof(*p->context));
 p->context->eip = (uint) forkret;

 return p;
}

```

We can create new processes now!

## growproc

What about growing or shrinking the size of a process's address space? We already did most of the hard work with `allocuvm()` and `deallocuvm()` from the post on process paging, so let's take a beat to thank past us for that.

Okay, so first we have to get the current process's size.

```
int growproc(int n)
{
 struct proc *curproc = myproc();
 uint sz = curproc->sz;
 // ...
}
```

Depending on the size of n, we'll either grow the process or shrink it by n bytes. Both `allocuvm()` and `deallocuvm()` can fail and return zero, so let's add some checks for those and return -1 if they fail.

```
int growproc(int n)
{
 // ...
 if (n > 0) {
 if ((sz = allocuvm(curproc->pgdir, sz, sz + n)) == 0) {
 return -1;
 }
 } else if (n < 0) {
 if ((sz = deallocuvm(curproc->pgdir, sz, sz + n)) == 0) {
 return -1;
 }
 }

 curproc->sz = sz;
 // ...
}
```

Finally, we need to tell the hardware that there's a new page directory in town with a different size than the old one, so we'll use `switchuvm()` to update the page directory and TSS stored by the hardware to reflect the changes. Then we return 0 to indicate everything went okay.

```
int growproc(int n)
{
 // ...
 switchuvm(curproc);
 return 0;
}
```

## procdump

This function is for debugging purposes: it'll print a complete listing of any processes in the process table. Quick spoiler: the keyboard interrupt handler function will set things up so that pressing ^P runs this function. Go ahead, load up xv6 and try it out!

We want to print out the state for each process, but the states in `enum procstate` are just integers, which isn't very debug-friendly. So let's map them all to strings first with a static array of strings.

```
void procdump(void)
{
 static char *states[] = {
 [UNUSED] "unused",
 [EMBRYO] "embryo",
 [SLEEPING] "sleep ",
 [RUNNABLE] "runble",
 [RUNNING] "run ",
```

```

 [ZOMBIE] "sombie",
};

// ...
}

```

This array notation might be a little unusual if you haven't seen it before: C lets you initialize arrays by specifying the value of each entry. If you leave any entries out, then they'll get initialized to zero. You can even write the entries out of order by adding their index before them in square brackets. So `{ [1] 5, [0] 2 }` is the same thing as `{2, 5}`. The `enum` turns the states into integers, so they work as indices here.

Now we'll just iterate over the process table to get all the processes, skipping over any `UNUSED` ones.

```

void procdump(void)
{
 // ...
 for (struct proc *p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
 if (p->state == UNUSED) {
 continue;
 }
 // ...
 }
}

```

Next we'll get the process's state (or just use `"???"` if something went wrong and the state isn't recognized).

```

void procdump(void)
{
 // ...
 for (struct proc *p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
 // ...
 char *state;
 if (p->state >= 0 && p->state < NELEM(states) && states[p->state]) {
 state = states[p->state];
 } else {
 state = "???";
 }
 // ...
 }
}

```

Then we can print out its PID, state, and name to the console.

```

void procdump(void)
{
 // ...
 for (struct proc *p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
 // ...
 cprintf("%d %s %s", p->pid, state, p->name);
 // ...
 }
}

```

Finally, we'll see later on that the `sleep()` and `wakeup()` system calls involve some lock trickery, so sleeping processes could be a common cause of concurrency issues like deadlocks. So if a process is sleeping, we'll print out its call stack using the `getcallerpcs()` function.

```

void procdump(void)
{
 // ...
 for (struct proc *p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
 // ...
 if (p->state == SLEEPING) {
 uint pc[10];
 getcallerpcs((uint *) p->context->ebp + 2, pc);

 for (int i = 0; i < 10 && pc[i] != 0; i++) {
 cprintf(" %p", pc[i]);
 }
 }
 cprintf("\n");
 }
}

```

## Summary

Whew, we're making good progress. The most important part of this code was how xv6 creates new processes and sets them up to start running: basically, it uses some stack and function call trickery to make the scheduler start running a new process with the code in `forkret()`, then `trapret()`, before switching context into user mode.

We haven't talked about those two functions yet; we'll hold off on that until we do traps and system calls. Next up is scheduling processes!

# More Paging: The User Side

It's almost time to turn to interrupts and processes so we can figure out how to work that sweet multiprocessing magic, but unfortunately we have some last pieces of paging to wrap up before we can get there.

I know, we've been talking about virtual memory for what feels like a century now, but so far everything we've done has been on the kernel side, allocating pages and creating new page directories with the same kernel mapping. But what about the lower half of the virtual address space, where user processes live?

This post will go through the rest of [vm.c](#) and set up the paging-related machinery we'll need to run processes later on.

## Detour: Starting a New Process

When xv6 runs a new process, it will create a brand new virtual memory space for it with a fresh page directory. We haven't talked about processes in xv6 yet, so you might wonder how a process gets started up in the first place.

Let's forget all about xv6 for a second and think about another Unix-like OS: Linux. How do we start a process there? Okay, we also have to forget about GUI applications there. Let's just say you want to run some C code (xv6 maybe?) that you've just compiled; what happens when you run it from the terminal?

Hopefully, you've done the OSTEP project called [processes-shell](#) by now, so you know the answer; if you haven't, I recommend doing that one right now before I give it away. (It's not strictly required, but are you really the kind of person who loves getting movies spoiled for them?)

Okay, are you done?

The answer: it's just an `exec()` system call! The shell finds the executable file in the file system, calls `fork()` to create a new child process, which then calls `exec()` to transform itself into the program you want to run.

We'll get to these system calls later, so for now let's just go over the broad strokes as they relate to virtual memory. `fork()` works by taking the parent process's virtual memory space and making a copy of it for the child process.

`exec()` allocates a new page directory, figures out how much memory the new program will need when it runs, then grows the virtual memory space allocated in that new page directory to the required size. Then it loads the program into memory in the new page directory.

Next, `exec()` skips a page, leaving it mapped but user-inaccessible; then the next page becomes the process's stack. Why that empty page? It's an important one for protection: that way, user programs that blow their stack will trigger a page fault or a general protection fault instead of possibly overwriting random code.

Then `exec()` copies some arguments into the stack before it switches to using the new page directory and gets rid of the old one it had before.

Whew, okay, that's a lot of code to go over later, and that's only the virtual memory part of the story. So let's just make it easier by doing all the work we can right now. According to the above, we have to understand how xv6 does all of the following:

- Makes a copy of a whole page directory,
- Creates a new page directory,
- Grows (or shrinks) the virtual memory space of a page directory,
- Loads program code into a page directory,
- Makes a page inaccessible to users,
- Copies stuff into a page in a page directory,
- Switches to a new process page directory, and
- Gets rid of an unused page directory.

Finally, there's one edge case to think about: running the very first process. We obviously need to start running a shell at some point, so we need a special way to get that started too, so it can in turn run other processes.

## vm.c, Again

We're gonna need some new functions! Actually, we already finished one of the requirements -- `setupkvm()` can allocate a new page directory and set up the kernel portion too. `switchkvm()` lets us switch to using `kpgdir` as a page directory, but now we need to switch *away* from that to a page directory for a process, so that'll be `switchuvm()`.

`copyuvvm()` creates a copy of an entire page directory for a child process. `allocuvvm()` and `deallocuvvm()` grow and shrink the virtual memory space that's allocated in a page directory, and `freeuvvm()` clears a page directory we no longer need.

`loaduvvm()` will load program code into a page directory; `clearpteu` makes a page inaccessible to users, and `copyout()` copies data into a page in a page directory. `inituvvm()` handles the special case of setting up the page directory for the very first process that xv6 will run.

The rest of this post will go over those functions one by one so we can be done with virtual memory, but I know it's a little strange to go through a million helper functions when we haven't seen the code that's gonna use them yet, so if you'd prefer, you can come back to this after reading about processes and system calls.

### deallocuvvm

The arguments for this function are a page directory, the process's old size, and the new size we want to shrink it down to; it'll return the process's new size. By "shrinking" a virtual memory space, we really mean making sure that the page directory only allocates up to `newsz` worth of pages. So if we think of the sizes as virtual addresses, then the page directory currently maps the virtual space from 0 to `oldsz`, so we should free everything between `newsz` and `oldsz`, leaving behind the space from 0 to `newsz`.

First, we should make sure the new size is actually smaller than the old one; otherwise trying to "shrink" down to the new size might cause integer overflow. There; the sizes are both unsigned integers here, so at least it wouldn't be that scary boogeyman of undefined behavior, but it could still

be bad: 0 would wrap around to  $2^{32} - 1$ , so "shrinking" to the new size would actually grow the process way beyond what physical memory could handle.

```
int deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
 if (newsz >= oldsz) {
 return oldsz;
 }
 // ...
}
```

We're gonna shrink the physical memory allocated to this page directory by freeing pages until we reach the new size. Let's start with the first page above `newsz`; we can get its virtual address by rounding up `newsz` to a page boundary.

```
int deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
 // ...
 uint a = PGROUNDUP(newsz);
 // ...
}
```

Now we'll just iterate over the pages between `a` and `oldsz` one at a time and free them. This is a little tricky: `kfree()` takes a virtual address (cast to a `char *`), but it should be a *kernel* virtual address in the higher half, not a user virtual address. Luckily, we already have `walkpgdir()`, which can take an arbitrary virtual address and return its page table entry, so that's a good start.

```
int deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
 // ...
 for (; a < oldsz; a += PGSIZE) {
 pte_t *pte = walkpgdir(pgdir, (char *) a, 0);
 // ...
 }
 // ...
}
```

The page table entry contains the page's physical address, plus some flags to determine whether it's mapped and what permissions are set for it.

Now, a virtual address space isn't laid out contiguously. Think about it: if you sit back and imagine a user process hanging out in memory, what does that address space look like? You're probably imagining the stack at one end of memory and the heap at the other, with each growing toward the center, right? so there will be some pages in the center that aren't mapped; some of the page tables might not exist either, in which case `walkpgdir()` would return a null pointer.

Remember we agreed to never dereference null pointers? Yeah, so we'll have to skip all those unmapped pages. If we got a null pointer, then that means the entire page table doesn't exist, so we need to skip forward to the next page directory entry (and thus the next page table). We'll have to move `a` to the virtual address that corresponds to that next page directory entry.

We can get the page directory index from `a` with the `PDX()` macro we've seen before, and then just add 1 to get the next entry in the page directory. Now we need to turn that back into a virtual address. We'll use a new macro, `PGADDR()` (also from [mmu.h](#)), to do that. So then we'll continue to the next loop iteration, which will get the page table entry for this new virtual address.

Wait wait wait, one last thing! After all that, `a` should now be the first virtual address in the page table for the new page directory entry... except it's get `PGSIZE` added to it because of the for loop's update statement.

Ugh, okay, fine, this is annoying. Let's just fix it with a hack: subtract `PGSIZE` from it now, so that it gets incremented to the right value in the next iteration. Okay, that's it, I swear!

```
int deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
 // ...
 for (; a < oldsz; a += PGSIZE) {
 pte_t *pte = walkpgdir(pgdir, (char *) a, 0);

 if (!pte) {
 a = PGADDR(PDX(a) + 1, 0, 0) - PGSIZE;
 } else {
 // ...
 }
 }
 // ...
}
```

Okay, now the else branch: if we don't get a null pointer then at least the page table exists, but that doesn't mean the page itself is mapped. If it's not, then we don't need to do anything else, but if it is mapped, then we need to free it. We can get the page's physical address out of the page table entry with the `PTE_ADDR` macro then make sure it's not null.

```
int deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
 // ...
 for (; a < oldsz; a += PGSIZE) {
 // ...
 if (!pte) {
 // ...
 } else if ((*pte & PTE_P) != 0) {
 uint pa = PTE_ADDR(*pte);
 if (pa == 0) {
 panic("kfree");
 }
 // ...
 }
 }
 // ...
}
```

The whole point of this was to be able to call `kfree()`, remember? So let's convert `pa` to a kernel virtual address as a `char *` and free it. Then after the loop is done, we'll return the new size.

```
int deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
 // ...
 for (; a < oldsz; a += PGSIZE) {
 // ...
 if (!pte) {
 // ...
 } else if ((*pte & PTE_P) != 0) {
 // ...
 char *v = P2V(pa);
 kfree(v);
 *pte = 0;
```

```

 }
 }
 return newsz;
}

```

## allocuvvm

This is the reverse of `deallocuvvm()`: instead of freeing pages with `kfree()`, we'll allocate them with `kalloc()`. Here too, we start by checking for integer overflow by making sure `newsz` really is larger than `oldsz`. But now we also have to check that we're not gonna grow the process's size into the region where it could access kernel memory; otherwise it might read or modify arbitrary physical memory.

```

int allocuvvm(pde_t *pgdir, uint oldsz, uint newsz)
{
 if (newsz >= KERNBASE) {
 return 0;
 }
 if (newsz < oldsz) {
 return oldsz;
 }
 // ...
}

```

We're gonna start adding new pages right after `oldsz`, so we have to align that to a page boundary:

```

int allocuvvm(pde_t *pgdir, uint oldsz, uint newsz)
{
 // ...
 uint a = PROUNDUP(oldsz);
 // ...
}

```

The for loop is easier this time around because we already know that the pages aren't mapped. First we allocate a new page. Any call to `kalloc()` needs two things after, remember? We have to check for null, in which case we print an error message to the console (that's `cprintf()`; we'll get to that in the devices section), then undo any allocations we made and return 0. Then we have to zero the page because we filled it with 1s when it was freed.

```

int allocuvvm(pde_t *pgdir, uint oldsz, uint newsz)
{
 // ...
 for (; a < newsz; a += PGSIZE) {
 char *mem = kalloc();
 if (mem == 0) {
 cprintf("allocuvvm out of memory\n");
 deallocuvvm(pgdir, newsz, oldsz);
 return 0;
 }
 memset(mem, 0, PGSIZE);
 // ...
 }
 // ...
}

```

We have a page now, but it's not yet mapped in the page directory. We can do that with `mappages( )`; that might fail too (because it needs to allocate more pages for the page tables), in which case we do the same as before. Then after the for loop is done, we return the new size.

```
int allocuvvm(pde_t *pgdir), uint oldsz, uint newsz) {
 // ...
 for (; a < newsz; a += PGSIZE) {
 // ...

 if (mappages(pgdir, (char *) a, PGSIZE, V2P(mem), PTE_W | PTE_U) < 0) {
 cprintf("allocuvvm out of memory (2)\n");
 deallocuvvm(pgdir, newsz, oldsz);
 kfree(mem);
 return 0;
 }
 }
 return newsz;
}
```

## freevm

This function will get rid of a user page directory that we no longer need. Now that we have `deallocuvvm( )`, it's easy: we just "shrink" the process to a size of zero. Oh and we'll remember the lessons our ancestors have taught us and make sure the pointer to the page directory isn't null before dereferencing it.

```
void freevm(pde_t *pgdir)
{
 if (pgdir == 0) {
 panic("freevm: no pgdir");
 }
 deallocuvvm(pgdir, KERNBASE, 0);
 // ...
}
```

Great, so all pages are freed, and we're done!

Now hang on a sec... The page directory itself resides in memory; so do the page tables. We have to free those too. We'll start with the page tables; freeing the page directory first would be a use-after-free vulnerability because we'd need to use it to get to the page tables.

We'll iterate over the page directory's entries, checking whether each one has the "present" flag set (`NPDENTRIES` is defined as 1024 in [mmu.h](#)). If it does, we'll get the page table's physical address from it with the `PTE_ADDR( )` macro, then convert that to a virtual address as a `char *` to make `kfree( )` happy. We don't have to worry about clearing the "present" flag in the page directory because it's about to be freed anyway.

```
void freevm(pde_t *pgdir)
{
 // ...
 for (uint i = 0; i < NPDETRIES; i++) {
 if (pgdir[i] & PTE_P) {
 char *v = P2V(PTE_ADDR(pgdir[i]));
 kfree(v);
 }
 }
 // ...
}
```

We wrap up by freeing the page directory itself.

```
void freevm(pde_t *pgdir)
{
 // ...
 kfree((char *) pgdir);
}
```

## copyuvvm

The `fork()` system call will need to "clone" a process, which includes its virtual address space. This function takes a pointer to the parent process's page directory and the size of the parent process's address space and returns a pointer to a fresh new page directory with everything set up exactly the same.

We start by creating a new page directory and taking care of the kernel's half of the address space with `setupkvm()`. That might fail if it can't allocate a new page, so we have to check for null. Sigh. C code is approximately 40% checking for null return values.

```
pde_t *copyuvvm(pde_t *pgdir, uint sz)
{
 pde_t *d;
 if ((d = setupkvm()) == 0) {
 return 0;
 }
 // ...
}
```

Now we'll iterate over the user portion of the parent process's address space from 0 to `sz`, copying everything over as we go. Say we want to copy a page from the parent's virtual address `i` to the child's address `i` (note that they'll map to different physical addresses). We'll have to figure out the corresponding kernel virtual address for the parent's `i` in order to do that, so we use `walkpgdir()` to get the page table entry, then get the page's physical address.

```
pde_t *copyuvvm(pde_t *pgdir, uint sz)
{
 // ...
 for (uint i = 0; i < sz; i += PGSIZE) {
 pte_t *pte;
 if ((pte = walkpgdir(pgdir, (void *) i, 0)) == 0) {
 panic("copyuvvm: pte should exist");
 }
 if (!(pte & PTE_P)) {
 panic("copyuvvm: page not present");
 }

 uint pa = PTE_ADDR(*pte);
 uint flags = PTE_FLAGS(*pte);
 // ...
 }
 // ...
}
```

In this case we know the parent process is already set up, so we don't really have to worry about `walkpgdir()` failing and returning null, but it's bad C juju to ignore a possibly-null return value, so we just panic if it does fail or if the page isn't present.

Next we allocate a page for the child process (checking for null again...) and copy everything from the parent's page to the new child page.

```
pde_t *copyuvvm(pde_t *pgdir, uint sz)
{
 // ...
 for (uint i = 0; i < sz; i += PGSIZE) {
 // ...
 char *mem;
 if ((mem = kalloc()) == 0) {
 goto bad;
 }
 memmove(mem, (char *) P2V(pa), PGSIZE);
 // ...
 }
 // ...
}
```

You might recognize `memmove()` as a C standard library function that copies the contents of one memory address into another, but we can't use those, remember? So xv6 provides its own implementation of it in [string.c](#).

If you haven't seen a `goto` statement before, it's basically a holdover from ye olde days before Edsger Dijkstra preached the gospel of structured programming to the world and invented the `if` statement. It does exactly what it sounds like: you make a label somewhere in code and it takes you there.

Next we stick that new page into the child's page directory, checking for null again. If `mappages()` fails, then the new page won't be in the page directory, so we have to free it here or else we'll never be able to find it again: a memory leak.

```
pde_t *copyuvvm(pde_t *pgdir, uint sz)
{
 // ...
 for (uint i = 0; i < sz; i += PGSIZE) {
 // ...
 if (mappages(d, (void *) i, PGSIZE, V2P(mem), flags) < 0) {
 kfree(mem);
 goto bad;
 }
 }
 // ...
}
```

If none of the allocations failed, we just return a pointer to the new page directory. But if something went wrong, then one of those `goto` statements will send us to the time out corner of `bad`, where we undo all our work by freeing the page directory and returning a null pointer.

```
pde_t *copyuvvm(pde_t *pgdir, uint sz)
{
 // ...
 return d;

bad:
 freevm(d);
 return 0;
}
```

Great, another function we'll have to check for null.

## switchuvvm

Okay, we've got a way to create a new process page directory. We also have a way to switch to using the kernel page directory `kpgdir` with `switchkvm()`. But we need a way to switch to using the process page directory too. Enter `switchuvvm()`.

I'll warn you -- `switchkvm()` was nice and short, but `switchuvvm()` is an ugly one for sure.

The argument to this function is a pointer to a `struct proc`, which represents a process. We'll talk about that more when we get to processes; two fields are important now: `p->kstack` which holds a pointer to the kernel stack for that process, and `p->pgdir`, which points to that process's page directory.

Okay, well let's start with some sanity checks to make sure that the process `p` actually exists (the pointer is non-null) and its kernel stack and page directory pointers are non-null too.

```
void switchuvvm(struct proc *p)
{
 if (p == 0) {
 panic("switchuvvm: no process");
 }
 if (p->kstack == 0) {
 panic("switchuvvm: no kstack");
 }
 if (p->pgdir == 0) {
 panic("switchuvvm: no pgdir");
 }
 // ...
}
```

The main function of loading the process's page directory will be the same as in `switchkvm()`: just an `lcr3` instruction. But the difference now is that the x86 architecture requires some additional bookkeeping for processes.

See, when the kernel runs a new process, the CPU will start executing different instructions. But it needs a way to keep track of where it left off in the kernel code so that it can pick the thread back up after the process is done executing. Similarly, interrupts and system calls might change the running process, so the CPU needs to record some metadata about the process's state too before switching to another one. x86 does that by means of a structure called a *Task State Segment*, or TSS.

The TSS holds information like the current state of certain registers (e.g., `%esp`, `%eip`, `%cr3`, etc.), segment descriptors (`%cs`, `%ss`, `%ds`, etc.), the current privilege level, and I/O privilege levels -- in other words, the process's *context*. It can be located anywhere in memory, but the processor needs to find it, so it uses an entry in the GDT called the TSS segment descriptor that points to the TSS. Remember the GDT from way back when we were talking about segmentation? Good times. The CPU holds a pointer to the GDT's TSS entry in a special register called the task register.

Back in the segmentation days of our youth, we stored the GDT in a `struct cpu` that held information about the current processor. We got that `struct cpu` by calling a `mycpu()` function. We're gonna do the same thing here in order to update the GDT with a segment for the TSS. Getting interrupted in the middle of this might be disastrous: the TSS would be half-updated,

so who knows what would happen when the CPU tried to resume execution where it last left off. So we'll use the `pushcli()` and `popcli()` functions we saw with spin-locks to temporarily disable interrupts.

```
void switchuvm(struct proc *p)
{
 // ...
 pushcli();

 mycpu()->gdt[SEG_TSS] = SEG16(STS_T32A, &mycpu()->ts, sizeof(mycpu()->ts)-1,
 0);
 mycpu()->gdt[SEG_TSS].s = 0;
 // ...

 popcli();
}
```

Whoa okay what is this?

We've seen the `SEG()` and `SEG_ASM()` macros before; they created GDT segments. `SEG16()` does the same with 16 bits (it's defined in [mmu.h](#)). `STS_T32A` is a flag that sets the segment's type as an available 32-bit TSS. Then we pass in a pointer to the task state with `&mycpu()->ts`, its size, and a descriptor privilege level of 0 (which means ring 0, the kernel level). The GDT's `.s` field is a one-bit flag to determine whether this is a system or application segment, so we set it to system.

Okay, so now the GDT points to the task state. Next we need to update the task state, then load it into the CPU. We'll start by storing a segment selector and the stack pointer in the task state; these should look familiar from the boot loader and `seginit()`.

```
void switchuvm(struct proc *p)
{
 // ...
 mycpu()->ts.ss0 = SEG_KDATA << 3;
 mycpu()->ts.esp0 = (uint) p->kstack + KSTACKSIZE;
 // ...
}
```

The TSS can also specify permissions for accessing I/O ports: for example, setting the I/O privilege level to 0 in the `eflags` register *and* setting a part of the TSS called the I/O map base address to an address beyond the TSS segment forbids I/O instructions like `inb` and `outb` from user space. So we'll set the I/O map base address next.

```
void switchuvm(struct proc *p)
{
 // ...
 mycpu()->ts.iomb = (ushort) 0xFFFF;
 // ...
}
```

So now we have a GDT entry pointing to the TSS, which is now updated. Now we just load it into the task register with the x86 instruction `ltr`; here we use a C wrapper for that assembly instruction, defined in [x86.h](#).

```
void switchuvm(struct proc *p)
{
 // ...
 ltr(SEG_TSS << 3);
```

```
// ...
}
```

Finally, the last thing we do before re-enabling interrupts is to load the process's page directory into the `%cr3` register so we can start using it.

```
void switchuvm(struct proc *p)
{
 // ...
 lcr3(V2P(p->pgdir));
 // ...
}
```

## loaduvm

Okay, this is another function that's gonna require extra info we haven't seen yet, but I'm gonna make it a bit easier by waving my hands around and glossing over the details. It's gonna read a program from a file into memory at virtual address `addr` using page directory `pgdir`. The part we want to read has size `sz` and is located at position `offset` within the file.

Now, what about the file? We'll talk more when we get to the file system code, but for now let's just say that files are represented in xv6 as `struct inodes`, and we can read from them with the function `readi()`.

We're gonna run the program from this code, so the address it's stored in needs to be page-aligned.

```
int loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
{
 if ((uint) addr % PGSIZE != 0) {
 panic("loaduvm: addr must be page aligned");
 }
 // ...
}
```

Next we're gonna iterate over pages starting from `addr`, reading from the file in `ip` into that page. As usual, we'll need to get the kernel virtual address from the user address `addr`, so we start by getting the page table entry via `walkpgdir()`, checking for a null pointer if the corresponding page table doesn't exist. Then we can turn that into a physical address.

```
int loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
{
 // ...
 for (uint i = 0; i < sz; i += PGSIZE) {
 // Get the page table entry
 pte_t *pte;
 if ((pte = walkpgdir(pgdir, addr + i, 0)) == 0) {
 panic("loaduvm: address should exist");
 }
 // Get the page's physical address
 uint pa = PTE_ADDR(*pte);

 // ...
 }
 // ...
}
```

Now we want to read from the file one page at a time using `readi()`, which takes a pointer to an inode (here, `ip`), a kernel virtual address (`P2V(pa)`), the location within the file of the segment we want to read (`offset + i`), and the segment's size.

Now we want to read from the file one page at a time using `readi()`. We have to specify a size in bytes to read; if the remaining unread part of the segment is larger than a page, then the size we pass to `readi()` should be `PGSIZE`, but otherwise it'll be less. So we'll compare `sz` to `i` and define `n` accordingly.

```
int loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
{
 // ...
 for (uint i = 0; i < sz; i += PGSIZE) {
 // ...
 uint n;
 if (sz - i < PGSIZE) {
 n = sz - i;
 } else {
 n = PGSIZE;
 }
 // ...
 }
 // ...
}
```

The other arguments to `readi()` are a pointer to an inode (`ip`), a kernel virtual address (`P2V(pa)`), and the location within the file of the segment we want to read (`offset + i`). It returns the number of bytes read, so if it's not `n` we'll report an error by returning `-1`. Otherwise we return `0` after the for loop is done.

```
int loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
{
 // ...
 for (uint i = 0; i < sz; i += PGSIZE) {
 // ...
 if (readi(ip, P2V(pa), offset + i, n) != n) {
 return -1;
 }
 }
 return 0;
}
```

## inituvm

Okay, the next three are nice and easy! This next one is pretty similar to `loaduvm()`, except instead of loading program code from disk, it copies it in from memory. We'll take `sz` bytes from a source address of `init` and stick it in address 0 of the process's page directory `pgdir`.

This function is also easier because we're only gonna call it for programs that are less than one page in size, so we don't have to worry about looping over pages or anything like that. I like it when xv6 keeps things simple.

```
void inituvm(pde_t *pgdir, char *init, uint sz)
{
 if (sz >= PGSIZE) {
 panic("inituvm: more than a page");
 }
```

```
// ...
}
```

Next we allocate a fresh page of memory, zero it to clear the garbage values, and stick it into `pgdir` at address 0.

```
void inituvm(pde_t *pgdir, char *init, uint sz)
{
 // ...
 char *mem = kalloc();
 memset(mem, 0, PGSIZE);
 mappages(pgdir, 0, PGSIZE, V2P(mem), PTE_W | PTE_U);
 // ...
}
```

And we wrap up by actually loading the code from `init` into the new page.

```
void inituvm(pde_t *pgdir, char *init, uint sz)
{
 // ...
 memmove(mem, init, sz);
}
```

## clearpteu

This function takes a page directory and a user virtual address and clears the "user-accessible" flag so that the process can't touch it. It's used to create an inaccessible page below a new process's stack to guard against stack overflows; this way, a stack overflow will cause a page fault instead of silently overwriting memory.

The `PTE_U` flag is in the page table entry, so we'll have to get that, then set the flag.

```
void clearpteu(pde_t *pgdir, char *uva)
{
 // Get the page table entry
 pte_t *pte = walkpgdir(pgdir, uva, 0);
 if (pte == 0) {
 panic("clearpteu");
 }
 // Clear the user permission flag
 *pte &= ~PTE_U;
}
```

Here `&` is a bitwise-AND and `~` is a bitwise-NOT; for reference, `|` is bitwise-OR and `^` is bitwise-XOR. Contrast these with their logical versions, `&&`, `!`, and `||` (XOR has no logical version). C also has corresponding assignment operators (similar to `+=`, `-=`, `*=`, etc.) for each of them. So the last line of code is equivalent to `*pte = *pte & (~PTE_U)`.

## uva2ka

We often need to convert user virtual addresses to kernel ones; `uva2ka()` is a short helper function that does that while checking that the page is actually present and has the user permission flag set.

We'll call `walkpgdir` to get the page table entry, then check both permission bits before recovering the page address with `PTE_ADDR( )` and converting it to a kernel virtual address. We'll return the kernel virtual address as a `char *`, or null if either flag is not set.

```
char *uva2ka(pde_t *pgdir, char *uva)
{
 pte_t *pte = walkpgdir(pgdir, uva, 0);

 if ((*pte & PTE_P) == 0) { // check that it's present
 return 0;
 }
 if ((*pte & PTE_U) == 0) { // check that it's user-accessible
 return 0;
 }
 return (char *) P2V(PTE_ADDR(*pte));
}
```

Let me ask you a weird question: how are you feeling right now?

Okay, that was a test of your C coding practices, because if you took those null checks to heart, you should be *really* uncomfortable right about now.

Check it out: `walkpgdir()` returns a pointer to the page table entry. *Any* time a function returns a pointer, you should immediately ask yourself whether that function can return a null pointer. Tons of C functions report an error by returning null. In this case, we *know* `walkpgdir()` can fail and report null if the page table doesn't exist, so we *know* we might get a null pointer out of it -- it'll happen whenever a page table doesn't exist. So what do we do with that knowledge?

Why, we go right ahead and dereference that pointer. WKBW;NQ39Q2A4T8YHMFGRW!!!

Dereferencing a null pointer is undefined behavior. There's literally no telling what might happen. It can cause all kinds of bugs from segmentation faults to security vulnerabilities.

All those null checks in the other functions serve a purpose: if something goes wrong and a function returns a null pointer, they catch it before it gets dereferenced, then either handle it gracefully or simply propagate the error by returning null (or some other error code) and let the caller figure out what to do with it.

Omitting a check for a null pointer like `uva2ka()` does is bad practice in C because it means the programmer has to *guarantee* -- by manually checking -- that no call to this function could *ever possibly* cause a null return value. Except humans are dumb, dumb creatures who make mistakes all the time, especially in big projects: there's no way you'd be able to remember that tiny little detail two years later when you decide to refactor your code or add a new feature or something.

But maybe you can note that in the comments? Okay yeah, but think about it: how often do you go and look up the source code for every single function you call? Yeah, I thought so.

This is why C is so dangerous: there are hundreds of such problems that you need to be aware of and remember to add stuff like null pointer checks to your code. If you don't because you're a normal human who forgets things sometimes, then you'll need to remember that you forgot to do it before and manually check every single call to your code and think about every possible edge case that a malicious adversary might exploit.

Good thing no one ever makes these mistakes in C, or we'd see enormous security vulnerabilities being reported every single day in all kinds of critical software. Oh wait...

So if you ever find yourself looking at C during code review and you come across a function that returns a pointer, you should stop what you're doing and look up the documentation for that function. If that function has any chance of returning a null pointer, then you should yell and kick and scream until somebody adds a null check and figures out how they want to handle it if it's null. Is this annoying? Yes. Hard to remember? Yes. But that's C. (*cough cough use Rust instead cough cough...*)

Now, the xv6 authors are so awesome that I'm gonna give them the benefit of the doubt and assume they left it off because they hand-checked every call to make sure it would never be an issue. But you and me? Nah.

The point of my rant is this: if you're reading this, then you're probably gonna find yourself hacking away at xv6 for a project sooner or later. When you do that, you should treat this function as VERBOTEN. You're not allowed to touch it or call it, at least until you add a null check to it yourself.

The same goes for any functions that call this one, because maybe all the existing calls to `uva2ka()` are fine right now, but then you make some tiny change and now it's no longer guaranteed to never be null. For reference, this function currently only gets called by `copyout()`, and that one only gets called by `exec()`. `exec()` gets called by `sys_exec()`, the shell, and the initial user-space program `init`. So be careful if you touch any of those.

Whew, okay, /rant.

## copyout

This function copies `len` bytes of data from a kernel virtual address `p` to a user virtual address `va` using page directory `pgdir`. `exec()` will use this to copy command-line arguments to the stack for a program it's about to run.

You might be wondering why it's needed -- doesn't `memmove()` do the same thing? Almost, but the difficulty is that `pgdir` may not be the current page directory, so we'll have to manually translate the virtual address `va`. That's where `uva2ka()` comes in, plus it ensures that the page for `va` has the right flags set. *Then* we can use `memmove()`.

First, `p` will be the source address, but `memmove()` requires a `char *` in order to copy data byte-by-byte, so let's convert it now:

```
int copyout(pde_t *pgdir, uint va, void *p, uint len)
{
 char *buf = (char *) p;
 // ...
}
```

Next we need to get the kernel virtual address corresponding to `va`, but there's a challenge: what if the data crosses a page table boundary? It might be spread across separate locations in physical memory (and thus in kernel virtual memory too). So we'll need a loop in which each iteration gets the next kernel virtual address and copies whatever part of the data is in this page.

```

int copyout(pde_t *pgdir, uint va, void *p, uint len)
{
 // ...
 while (len > 0) {
 // ...
 len -= n;
 buf += n;
 // ...
 }
 // ...
}

```

So we'll start each iteration by making `va0` the base address of the page `va` is on and `pa0` the kernel address of `va0`, converted with `uva2ka()`. I... honestly don't know why they used `pa0` as an identifier here. It makes it look like it should be a physical address, but it's not; it's a kernel virtual address. Sigh. Anyway, the call to `uva2ka()` might fail if the page isn't present or it doesn't have a user permission bit, so we have to check for a null pointer and return -1 if we find one.

```

int copyout(pde_t *pgdir, uint va, void *p, uint len)
{
 // ...
 while (len > 0) {
 uint va0 = (uint) PGROUNDDOWN(va);
 char *pa0 = uva2ka(pgdir, (char *) va0);
 if (pa0 == 0) {
 return -1;
 }

 // ...
 va = va0 + PGSIZE;
 }
 // ...
}

```

Now `va` is in between `va0` and the next page, so the length of the data within this page is `PGSIZE` - (`va` - `va0`), unless it's the last page, in which case we should pick the lesser of this value and `len` (since `len` gets decremented on each iteration through the loop).

```

int copyout(pde_t *pgdir, uint va, void *p, uint len)
{
 // ...
 while (len > 0) {
 // ...
 uint n = PGSIZE - (va - va0);
 if (n > len) {
 n = len;
 }
 // ...
 }
 // ...
}

```

Finally, we copy the data from `buf` into the target kernel virtual address for `va`. Hmm, we don't have that yet. Oh wait, `pa0` is the kernel virtual address for `va0`, and `va` is just `va - va0` bytes after that, so we'll use it.

```

int copyout(pde_t *pgdir, uint va, void *p, uint len)
{

```

```
// ...
while (len > 0) {
 // ...
 memmove(pa0 + (va - va0), buf, n);
 // ...
}
return 0;
}
```

We return 0 if everything went okay.

## Summary

Okay, that was a lot of helper functions, but we're ALL DONE with virtual memory! From now on, we have all the tools we'll need to manage memory and set up virtual address spaces for new processes.

# More Paging: The Kernel Side

We've already talked *plenty* about virtual memory, and I bet you're probably so over `entrypgdir` by now; let's wrap up its story and get rid of it!

The `vm.c` file is HUGE; only `proc.c` and `sysfile.c` match its length. Some parts deal with the general paging implementation; we'll look at those here. The rest handles the details of paging for processes and user code, we'll need to know a bit more about processes in xv6 for that.

## vm.c

After the include directives for the preprocessor, we have a declaration for an external symbol defined in `kernel.ld`. This one is the beginning of the data section for the kernel.

```
extern char data[];
```

Next we have a definition for a pointer to a global page directory: this is the fancy new one that's gonna replace `entrypgdir`. Note that `pde_t` is a type for page directory entries defined in `types.h`; it's just a type alias for `int`.

```
pde_t *kpgdir;
```

## seginit

This first function gets called directly by the kernel's `main()`; it sets up the segment descriptors in the GDT as identity maps to all of memory so that we can ignore them from now on. Wait, didn't we already do that in the boot loader?

Yes, kind of, but that was before the kernel took over, so back then we had no notion of kernel space versus user space. Now that we do, we want to set the permission flags for each segment so that we can use the privilege ring levels, with the kernel in ring 0 and user code in ring 3. That way any misbehaving user code will get slapped with a segmentation fault the way we've all come to know and love in C.

We also have some permission flags for protection in the page directory and page table entries, so maybe we could get away without it? I mean, both kernel code and user code are read-only anyway, so maybe they could both have a Descriptor Privilege Level of 3. But no, x86 is gonna shut that right down by forbidding interrupts that take you from ring level 0 to ring level 3, so all the interrupt handler functions have to be in kernel space with a kernel code segment selector at ring level 0.

So we're just gonna have to do it all over again. Great. Well, maybe it's not too bad, let's take a look... oh god, it's awful. Okay, deep breath.

Each processor has its own GDT, so we're gonna need to call this function once per CPU. First we figure out which CPU we're on with the `cpuid()` function that we'll see later on; for now it... (drumroll)... gets the CPU's ID. Then we look that up in a global table of CPUs (there's an `extern` declaration for this in the included `proc.h`) and store it in a `struct cpu`; we saw that before in the spin-lock code, but we'll get around to talking about it more later.

```

void seginit(void)
{
 struct cpu *c = &cpus[cpuid()];
 // ...
}

```

That `struct cpu` has a field to hold the GDT, so we're gonna add entries for the kernel code, kernel data, user code, and user data segment descriptors; those entries are `SEG_KCODE`, `SEG_KDATA`, `SEG_UCODE`, and `SEG_UDATA`, respectively. Recall that the permission bits are `STA_X` (executable), `STA_R` (readable), and `STA_W` (writeable); now we're gonna pile on the descriptor privilege levels for the kernel (0) and user (3, or `DPL_USER`) on top. Besides those ring levels, we want to ignore segmentation, so each segment should be an identity map for all virtual memory from 0 to 4 GB (`0xffff_ffff`).

```

void seginit(void)
{
 // ...
 c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
 c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
 c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
 c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
 // ...
}

```

The only difference between the `SEG` macro used here and the `SEG_ASM` one from the boot loader is that this one is for C code and the other is for assembly.

Finally, we load up the new GDT into the processor with a C wrapper for the x86 instruction `lgdt`.

```

void seginit(void)
{
 // ...
 lgdt(c->gdt, sizeof(c->gdt));
}

```

Done with segmentation, now on to more paging.

## walkpgdir

A page directory lets the paging hardware convert virtual addresses to physical ones, but we're gonna need those mappings in the kernel too while we set up the page directory, so this function does the conversion manually. Wait, but aren't we setting up paging so that all of physical memory is mapped in the higher half of the virtual address space? Can't we just add or subtract `KERNBASE` to do the conversion? Well, that would work for kernel virtual addresses, but user virtual addresses actually will use page directories and page tables in a non-obvious way, so if we want to figure out where those go, we'll need a function for it.

In C, using the `static` keyword before a function limits its scope and makes it visible only within its own file. The function returns a `pte_t *`, a pointer to a page table entry (the type is defined in [`mmu.h`](#) as a type alias for `uint`).

Its arguments are a pointer to a page directory, a virtual address, and `alloc` (a boolean variable, but as an `int` instead of `bool`). This `alloc` lets the function play a dual role: if it's set, the function will allocate a page table if needed; otherwise it reports failure if the page table doesn't

exist. The `const` keyword lets the compiler know a variable shouldn't be mutated so it'll throw an error if we do. Here, `const void *va` is a pointer to a constant value of any type; the address the pointer holds might change, but we can never write to that address. The opposite is a `void *const va`: the address being pointed to will never change, but we can overwrite the contents of that address all we want. You can combine the two with `const void *const va`. What's that I hear? C syntax is the worst? No, never...

```
static pte_t *walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
 // ...
}
```

Remember way back when, when we talked about how "linear" addresses are set up and converted to physical ones? The first 10 bits are an index for the page directory to pick a page directory entry, which points to a page table; the next 10 bits pick a page table entry that points to a page, and the last 12 bits are an offset within that page; the `PDX()` and `PTX()` macros get first 10 bits and the next 10 bits from a linear address, respectively. So we start by getting the page directory index and using that to get the page directory entry.

```
static pte_t *walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
 pde_t *pde = &pgdir[PDX(va)];
 // ...
}
```

Okay, so now `pde` points to a page directory entry which has two parts: a pointer to the physical address of a page table, and some flags. But who knows if this page table even exists; most page directory (and page table) entries aren't mapped in order to save space. So we have to check whether `*pde` has the `PTE_P` (present) flag set.

```
static pte_t *walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
 // ...
 pte_t *pgtab;
 if (*pde & PTE_P) {
 // ...
 } else {
 // ...
 }
 // ...
}
```

If the page table exists, we should get rid of the flags and recover the pointer to the page table using the `PTE_ADDR()` macro. But the hardware uses physical addresses for these pointers, so we need to convert it to a virtual address first, which is what this function does... recursion? Bootstrap problem? No, it's actually easy because we can access the page table from within the kernel's virtual address space in the higher half by adding `KERNBASE` to the physical address with the `P2V()` macro.

```
static pte_t *walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
 // ...
 pte_t *pgtab;
 if (*pde & PTE_P) {
 pgtab = (pte_t *) P2V(PTE_ADDR(*pde));
```

```

} else {
 // ...
}
// ...
}

```

Now for the else clause, which happens if the page directory entry doesn't have the PTE\_P bit set. Well, if the boolean `alloc` is false (zero), then we're done and we should just report failure by returning a null pointer. On the other hand, if it's true, we just allocate a page for the page table. But wait, remember how page allocation might fail and return a null pointer if we're out of free pages in the free list? And remember how I said we should always check for that? Okay well let's check for that; if allocation fails, we also return a null pointer. Oh, and because this is C, we're gonna do a jillion things at once in a single line: check if `alloc` is false, try to allocate a page table, and check if that allocation failed. C lets us assign to a variable and then test that variable's value in a single statement.

```

static pte_t *walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
 // ...
 pte_t *pgtab;
 if (*pde & PTE_P) {
 // ...
 } else {
 if (!alloc || (pgtab = (pte_t *) kalloc()) == 0) {
 return 0;
 }
 // ...
 }
 // ...
}

```

Okay, so now suppose: (1) the page table wasn't present, (2) `alloc` was set, and (3) we successfully allocated a page. Now what? Remember how we filled all free pages with garbage in `kfree()` using `memset()`? Let's undo that now by zeroing it.

```

static pte_t *walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
 // ...
 pte_t *pgtab;
 if (*pde & PTE_P) {
 // ...
 } else {
 // ...
 memset(ptab, 0, PGSIZE);
 // ...
 }
 // ...
}

```

Now we'll update the page directory entry to point to this new page table and add the PTE\_P flag so it knows it's present. Wait, while we're at it, what other permissions will it need? Is it writeable? Can users access it? Hmm, we'd have to know whether we're looking up a user virtual address or a kernel one, and whether it's gonna be used for code or data. Ah, screw it, we'll just throw all the flags on there at once. Either way, the page table entries will have their own flags too, so we can restrict the page's permissions there instead of here at the page directory entry.

```

static pte_t *walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
 // ...
 pte_t *pgtab;
 if (*pde & PTE_P) {
 // ...
 } else {
 // ...
 *pde = V2P(ptab) | PTE_P | PTE_W | PTE_U;
 }
 // ...
}

```

This probably isn't the safest thing ever, because we're saying that only the page table will restrict permissions, so we're throwing all that responsibility over there, but hey, xv6 is supposed to be simple, not ultra-secure. Just don't do this at home, kids.

Finally, we return the address of the corresponding page table entry using the index from the middle bits of `va`:

```

static pte_t *walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
 // ...
 return &ptab[PTX(va)];
}

```

## mappages

Okay, so `walkpgdir()` returns a pointer to a page table entry and can even create a page table if it doesn't exist. That's not quite enough to add new mappings for pages though; the page itself might not be mapped, and if we just created a new page table, then certainly none of the pages are mapped yet. `mappages()` will finish the job by installing mappings in page tables (possibly newly-allocated ones) for a range of virtual addresses.

The arguments are a page directory, a virtual address for the beginning of the range, the size of the range, a physical address to map it to, and the flags for permissions we want to set. We start off by rounding the virtual address down to the nearest page boundary and getting a pointer to the end of the range, also page-aligned.

```

static int mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
 char *a = (char *) PGROUNDDOWN((uint) va);
 char *last = (char *) PGROUNDDOWN((uint) va) + size - 1;
 // ...
}

```

Now we're gonna iterate over the pages in that range; `for (;;)` is a common C idiom for an infinite loop. In this case, we need to increment `a` and `pa` by `PGSIZE` each time, and we'll break out of the loop when `a` reaches `last`. To be completely honest, I'm not really sure why the authors chose to write this as an infinite loop with the condition/break statement and update statements inside the loop rather than as a regular old for loop; I think the latter would be more clear, but oh well, I didn't write this.

```

static int mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
 // ...
}

```

```

for (;;) {
 // ...
 if (a == last) {
 break;
 }
 a += PGSIZE;
 pa += PGSIZE;
}
// ...
}

```

Inside the for loop, we'll start each iteration by looking up the right page table entry with `walkpgdir()`, with `alloc` set to true. Remember how that function called `kalloc()`, which might fail, in which case it returns a null pointer? Well that means we've gotta check for a null pointer here too. This time however, we'll return -1 for failure and 0 for success, because why not?

```

static int mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
 // ...
 for (;;) {
 pte_t *pte;
 if ((pte = walkpgdir(pgdir, a, 1)) == 0) {
 return -1;
 }
 // ...
 }
 return 0;
}

```

We're supposed to be allocating brand-new pages for this range of addresses, so if a page has already been allocated, we'll just flip out in rage and panic.

```

static int mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
 // ...
 for (;;) {
 // ...
 if (*pte & PTE_P) {
 panic("remap");
 }
 // ...
 }
 // ...
}

```

The last thing before checking the loop condition and updating `a` and `pa` is to install the mapping to the right physical address with the right permissions in the page table. Then we're done!

```

static int mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
 // ...
 for (;;) {
 // ...
 *pte = pa | perm | PTE_P;
 // ...
 }
 // ...
}

```

Cool, now we have a way to map new pages into a page directory. We're well on our way to leaving poor old `entrypgdir` behind for the shiny new `kpgdir`.

## kmap

Each process is gonna have its own page directory, so its mappings in the lower half of the virtual address space might be totally different from those of another process. But the mappings in the higher half (where the kernel lives) will always be the same -- that way, the kernel can always use the existing page directory for whatever process it happens to be running. We'll only use `kpgdir` when the kernel isn't currently running a process, e.g. while it's running the scheduler.

So when we create a new process, we'll need to copy in all the mappings that the kernel expects to find into a fresh page directory for that process. Those are: memory-mapped I/O device space from physical address 0 to 0x10\_0000 (the boot loader is also here, but we don't need it any more), kernel code and read-only data from 0x10\_0000 to the physical address of `data` (one of the symbols defined in [kernel.ld](#)), kernel data the rest of physical memory from there to `PHYSTOP`, and more I/O devices from 0xFE00\_0000 and up. Each of these ranges needs its own permissions too.

We'll represent each of these mappings with a `struct kmap`, which has fields for the starting virtual address, the starting and ending physical addresses, and the permissions; then the mappings will get stored in a static global variable `kmap`... oh come on, what fresh hell is THIS?

```
static struct kmap {
 void *virt;
 uint phys_start;
 uint phys_end;
 int perm;
} kmap[] = {
 { (void *)KERNBASE, 0, EXTMEM, PTE_W },
 { (void *)KERNLINK, V2P(KERNLINK), V2P(data), 0 },
 { (void *)data, V2P(data), PHYSTOP, PTE_W },
 { (void *)DEVSPACE, DEVSPACE, 0, PTE_W }.
};
```

Okay, there are a few things going on here. First, the `static` keyword for a variable means that variable has a single fixed location in memory that it's never gonna move out of.

Then it does that thing again where we simultaneously define a `struct` type and define a variable of that type. So the type is

```
struct kmap {
 void *virt;
 uint phys_start;
 uint phys_end;
 int perm;
};
```

So then the static global variable `kmap` is an array of `struct kmaps`. I guess we ran out of names or something. The array has four entries, and since each one is a `struct`, it needs curly braces around it.

The first entry (for the lower of the two memory-mapped I/O device regions) has a `virt` field of `KERNBASE`, a `phys_start` field of 0, a `phys_end` field of `EXTMEM` (defined as `0x10_0000`), and permission flag `PTE_W`. So it maps a virtual address range starting at `KERNBASE` to the

physical address range from 0x0 to 0x10\_0000 and makes it writeable so we can communicate with the devices there. The next two entries are similar, except that the kernel code isn't writeable.

The last entry has `phys_start` of 0xFE00\_0000 and a `phys_end` of 0. That's a little strange, but it's because we want to map all the way up to the end of the virtual address space at 0xFFFF\_FFFF. The end should be one byte past that, but it's impossible to represent 0x1\_0000\_0000 with 32 bits. Setting the end to 0 makes the size calculation (`phys_end - phys_start`) work out nicely: it'll just overflow to the right number. This is okay since we're using unsigned integers, but note that *signed* integer overflow is undefined behavior and thus VERY BAD and the cause of many security vulnerabilities.

Okay, back to getting rid of `entrypgdir`!

## setupkvm

This function sets up a fresh new page directory with all the mappings in `kmap` in order to please the kernel when it encounters the page directory. So needy, right?

It takes no arguments and returns a pointer to the new page directory. First, let's allocate a page of memory to hold the new directory. We'll be good and remember to check for null (in which case we return null too) and clear the page of the garbage values we wrote when we freed it.

```
pde_t *setupkvm(void)
{
 pde_t *pgdir;
 if ((pgdir = (pde_t *) kalloc()) == 0) {
 return 0;
 }
 memset(pgdir, 0, PGSIZE);
 // ...
}
```

The upper end of virtual memory after `DEVSPACE` has I/O devices, so `PHYSTOP` should be below that; this is as good a place as any to make sure.

```
pde_t *setupkvm(void)
{
 // ...
 if (P2V(PHYSTOP) > (void *) DEVSPACE) {
 panic("PHYSTOP too high");
 }
 // ...
}
```

Finally, we'll add all the mappings in `kmap` above into this page directory so the kernel is happy. We'll use `mappages()`, which returns -1 if it fails, so we should check for that. The `freevm()` function is defined below, and we'll get to it soon, but for now just know that it gets rid of all the mappings we just made, in case any of them fails.

```
pde_t *setupkvm(void)
{
 // ...
 struct kmap *k;
 for (k = kmap; k < &kmap[NELEM(kmap)]; k++) {
 if (mappages(pgdir,
 k->virt,
```

```

 k->phys_end - k->phys_start,
 (uint) k->phys_start,
 k->perm) < 0) {
 freevm(pgdir);
 return 0;
}
}
return pgdir;
}

```

Let's check out that for loop: `k` is a pointer to a `struct kmap`, and `kmap` is an array of `struct kmaps`; in C, arrays decay to pointers, so they have the same type. `k` starts off pointing to the first (zero) entry of `kmap`. Then incrementing it with `k++` shifts its value by the size of a `struct kmap`, so it'll point to the next entry. The loop stops when `k` points beyond the last entry of `kmap`, as determined by the `NELEM()` macro which counts the number of entries in an array. Note that array element-counting only works in C if the array is defined in the same function or as a global variable in the same file, which is why it's so easy to do an out-of-bounds read or write in C (yet another common security vulnerability).

Finally, if everything worked out okay, we return a pointer to the new page directory.

## switchkvm

We said above that the kernel would usually just use the page directory of the currently-running process, but it'll use `kpgdir` when no process is running, i.e. during the kernel setup and while it's scheduling a new process. So we need a way to tell the paging hardware to load `kpgdir` into register `%cr3`, which holds a pointer to the page directory. That's this function.

It's a one-liner: get the physical address of `kpgdir` and stick it in `%cr3` with the assembly instruction `lcr3`.

```

void switchkvm(void)
{
 lcr3(V2P(kpgdir));
}

```

## kvmalloc

FINALLY, we're here! We're gonna get rid of `entrypgdir`! The kernel's `main()` calls this function right after `kinit1()`.

We already did all the hard work, so this one's a breeze: we call `setupkvm()` to allocate a new page directory and fill it with the kernel's mappings, then call `switchkvm()` to load it into the paging hardware.

```

void kvmalloc(void)
{
 kpgdir = setupkvm();
 switchkvm();
}

```

And we're DONE! Take that, `entrypgdir`, we don't need you anymore. We're big kids now.

## Summary

So far, it's been a serious odyssey just to move from no paging in the boot loader, to super basic paging with `entrypgdir` in [entry.S](#), to `kpgdir` now. Along the way, we've looked at code to allocate and free pages and install new mappings in page directories and page tables. That'll come in handy when we look at processes next; the virtual memory story still isn't over.

Also, note that `kpgdir` still isn't at the height of its powers: at the point when `main()` calls `kvmalloc()`, the free list only contains pages for physical memory between 0 and 4 MB. The rest will have to wait until `kinit2()` unleashes its full potential. (Maybe some self-actualization seminars would help...)

# Page Allocation

When we left off before the lock detour, the boot loader had set up a GDT to ignore segmentation, and the entry code set up some barebones paging with an `entrypgdir`. But that initial page directory is too limiting to keep for long; it only mapped the first 4 MB of physical memory. So we want a new one, but we have to set it up and allocate pages in it before we can actually use it. And until we switch to it, everything has to happen in those first 4 MB.

## kalloc.c

We start off in this file by declaring the function `freerange()`, which will be defined below. We have to do this in C in order to call a function in the code before the compiler has actually seen the function's definition, which comes below, or maybe in another file. A *declaration* tells the C compiler "I know I haven't shown you this symbol before, but don't worry; it's just a function that takes this number of arguments with these types and has a return value of this type." That lets the compiler keep calm and carry on with its usual type-checks (weak as they may be in C). A *definition* tells the compiler that this is the function (or variable) we were talking about, so it'll reserve some space in memory for it; it also tells the compiler how to evaluate that function whenever it's called (for variables, an *initialization* will have to tell the compiler what the value the variable should hold). The linker will take care of matching function calls (and variable uses) to their definitions, possibly across files.

Usually you'd stick declarations in a C header file and tell the preprocessor to copy-paste the header into your code with an `#include` directive; then other files could `#include` that header too. So header files should really be more of an API kind of thing, for functions that you want other code to be able to call. This one is just a local helper function, so we'll declare it here instead of in a header so other code can't use it.

```
void freerange(void *vstart, void *vend);
```

Okay okay, I know function declarations are like 101-level C, but I wanted to mention them because we're about to see something similar but a little off next when we declare `end` as a global array of characters.

```
extern char end[];
```

The C keyword `extern` lets you define a global variable or function in one file and use it in another, so in that sense it's similar to the function declaration above. In fact, the compiler implicitly assumes there's an `extern` before each function declaration. The difference is that an explicit `extern` lets us do the same thing for global variables: we tell the compiler and linker "hey, I'm gonna use a variable of this type with symbol `end`, but don't worry about reserving a spot in memory for it; that already happened elsewhere."

The really cool thing about `extern` is that the function or variable might not even be defined in C -- it could come from any other language! We just pass the compiled object files from the other language together with the C object files to the linker and it'll match up the definitions and calls.

In this case if you try looking for the place where `end` is defined in the C or assembly code, you're gonna be disappointed. Turns out it's actually defined in `kernel.ld`, remember? Back then, we said it was gonna be located at the very first memory address right after the end of the kernel code and data in memory. We're about to see why it's needed.

Next up, we define a new `struct` type:

```
struct run {
 struct run *next;
};
```

Hmm, the only member of this `struct run` is a pointer to another `struct run`. Hopefully, you've seen some singly-linked lists before so you can recognize it as one of those. Usually it would have another member to hold the data in the list, but we won't need any extra data here; we'll find out why soon enough.

Last thing before we get to the functions: we define another `struct` type and declare the global variable `kmem` to be of that type.

```
struct {
 struct spinlock lock;
 int use_lock;
 struct run *freelist;
} kmem;
```

The syntax here is the usual C thing where we say the type of a variable, then an identifier, like `int i`; it just looks more confusing because we're also defining the type at the same time. This `struct` type doesn't get a name like `struct run` did because we're only gonna need it this one time. The fields are a spin-lock (hence the detour before coming here), a `use_lock` variable that we'll treat as a boolean, and a pointer to a `struct run` called `freelist`.

I'm just gonna go ahead and spoil the next two functions for you: we want to use a better page directory than `entrypgdir`, right? Well then we need to assign a page of memory for it, plus a page for each of its page tables, plus a page for each entry in those page tables that's mapped. That means we'll need some bookkeeping to track which pages have already been assigned. We're gonna use a linked list of free pages (that's what `struct run` is for); we'll allocate a page by popping one off the free list, and we'll free a page by pushing it onto the top of the list.

Note that `kfree()` here is *not* supposed to be a kernel version of the usual C standard library function `free()`, nor is `kalloc()` supposed to be a kernel version of `malloc()`. We have no concept of a heap yet, so heap allocation wouldn't make sense. These functions allocate and free *whole physical pages* to be added to the current page directory and its page tables.

## kfree

This function will free a single page (4096 bytes, or `PGSIZE`) of memory by adding it to the front of the free list. It takes an argument `char *v` which is a virtual address; we're using `char *` here instead of `uint *` or `void *` or whatever so that the pointer arithmetic increments by a single byte instead of 4 bytes for `uint` or whatever.

First, some sanity checks: `v` should be page-aligned (because we're freeing a whole page), it should be above `end` (because we don't want to accidentally overwrite the kernel code), and its corresponding physical address should be below `PHYSTOP` (because the only addresses we'll use above the top of physical memory are for memory-mapped I/O devices and we shouldn't be freeing those pages anyway).

```
void kfree(char *v)
{
 if ((uint) v % PGSIZE || v < end || V2P(v) >= PHYSTOP) {
 panic("kfree");
 }
 // ...
}
```

Now, if you've programmed in C, you might have come across the dreaded (but oh-so-common) bug known as a *use-after-free*. This means you called `free()` on some variable (hopefully one you had `malloc()`-ed before), and then used it again. Hmm, very naughty! The problem is that that memory might have been re-allocated to some other variable or even another process, so you might read the wrong values or overwrite something important. This is a *very common cause of security vulnerabilities* in C and C++ to this day; it's also not always easy to spot because huge projects might have you call `malloc()` in one file, then use the variable somewhere else thousands of lines of code later in some other file, then call `free()` in yet another file -- plus it's unlikely that all of these pieces were written by the same person. So let's make this a little easier on ourselves by filling the freed page with junk (a bunch of 1s everywhere) in the hope that a use-after-free leads to a crash (and thus debugging and detection) sooner than it would otherwise.

```
void kfree(char *v)
{
 // ...
 memset(v, 1, PGSIZE);
 // ...
}
```

You might be familiar with `memset()` from the C standard library in `string.h`, but we can't risk using standard library functions here because they assume the code will be provided by the OS, and the implementation might require any of a million features we haven't implemented yet. So we have to make our own version for the kernel in [string.c](#). We'll get around to looking at that code later on in an optional detour, but for now just know that it sets the memory starting at `v` and continuing for `PGSIZE` bytes to hold a bunch of repeated 1s.

Now let's talk concurrency. At any time, multiple threads might want to allocate or free pages simultaneously; if we're not careful we might accidentally use the same page twice, which would cause bugs in addition to security vulnerabilities, because all the per-process isolation that paging gets us would be lost. So much work down the drain! This is why `kmem` has a lock, which we should use any time we push to or pop from the free list.

But in the early stages of the kernel we only use a single CPU and interrupts are disabled, so there's nothing to fear. Plus, locks add overhead, and the `acquire()` function needs to call `mycpu()`, which we haven't even defined yet, so let's just go ahead and skip them in the beginning. So `kmem.use_lock` is a boolean that will tell us whether we need a lock right now or not.

```
void kfree(char *v)
```

```
{
 // ...
 if (kmem.use_lock) {
 acquire(&kmem.lock);
 }
 // ...
}
```

Okay, we're finally at the point where we can free the page. We'll make a `struct run *r` that points to virtual address `v`, then make its `next` point to the first entry of the free list. Then we'll update the head of the list to point at the newly-freed page. This is the standard C idiom to add to the front of a singly-linked list.

```
void kfree(char *v)
{
 // ...
 struct run *r = (struct run *) v;
 r->next = kmem.freelist;
 kmem.freelist = r;
 // ...
}
```

There's something interesting here: where are we storing this entry for the free list? Why, in the free page itself! So each unused page will hold the address of the next one in its first few bytes.

Finally, we're out of the critical section where we updated the free list, so we can release the lock.

```
void kfree(char *v)
{
 // ...
 if (kmem.use_lock) {
 release(&kmem.lock);
 }
}
```

## kalloc

Allocating a page means popping off the head of the free list. We acquire the lock first, if we need one.

```
char *kalloc(void)
{
 if (kmem.use_lock) {
 acquire(&kmem.lock);
 }
 // ...
}
```

Next, we get a pointer to the first free page in the list and update the head to point to the next one in the list. But what if the list is empty? In that case, the head would be a null pointer, and dereferencing a null pointer (like we do here in `r->next`) is undefined behavior in C, which means BAD THINGS HAPPEN. I'm serious -- there are absolutely no restrictions on what might happen, so the compiler could literally set your computer on fire if it wanted to. In the real world, that usually means either a segmentation fault or security vulnerability, or both if you're unlucky. So we should check whether `r` is null (i.e. zero). If it's nonzero then we can update `r->next`; otherwise we should just return `r` and hope whoever called us checks whether it's null. Moral of the

story: any call to `kalloc()`, just like any call to `malloc()` in regular C code, should always be followed by checking whether the returned pointer is null.

```
char *kalloc(void)
{
 // ...
 struct run *r = kmem.freelist;
 if (r) {
 kmem.freelist = r->next;
 }
 // ...
}
```

Okay, so now we just release the lock, and we're done!

```
char *kalloc(void)
{
 // ...
 if (kmem.use_lock) {
 release(&kmem.lock);
 }
}
```

## freerange

`kalloc()` and `kfree()` both handle only one page at a time, which can get annoying if we're trying to free tons of pages at once; also, they can only use page-aligned virtual addresses, which have to be typecast to `char *`. Let's simplify our lives with a simple wrapper function to free multiple pages between two virtual memory addresses `vstart` and `vend` that may not be page-aligned.

Let's assume that `vstart` is the first address after some other data in an already-allocated page; we don't want to free that page, but the next one, so we align it to a page boundary by rounding up, then cast that to a `char *`.

```
void freerange(void *vstart, void *vend)
{
 char *p = (char *) PGROUNDUP((uint) vstart);
 // ...
}
```

Now we can iterate over the pages, starting at `p` and incrementing by `PGSIZE` until we reach or pass `vend`, freeing pages as we go.

```
void freerange(void *vstart, void *vend)
{
 // ...
 for (; p + PGSIZE <= (char *) vend; p += PGSIZE) {
 kfree(p);
 }
}
```

Done, next.

## kinit1 and kinit2

Both of these functions get called by the kernel's `main()`. Quick reminder: we've got an `entrypgdir` that maps two virtual address ranges (0 to 4 MB and KERNBASE to KERNBASE + 4 MB) to the physical addresses range from 0 to 4 MB. We want to leave this baby page directory behind for a grown-up page directory that maps all of physical memory, but first we needed to figure out how to allocate pages.

Okay cool, we already did that. But allocation needs a free list, which for now is just sitting around chilling as an empty list. But we can't free pages if they're not already allocated, right? Ahh, bootstrap problems! This one's not an issue; we'll just cheat this one time and free all the memory between `end` (the end of the kernel code and data in memory) and `PHYSTOP`, even though we didn't get it from a call to `kalloc()`. Sounds good, right?

I hate to burst your bubble, but kernel development *loves* bursting bubbles. Turns out there's yet another bootstrap problem: each page has to store the pointer to the next free page, which means we have to write to that page, which means that page must already be mapped... but we can't map all of memory until we initialize the free list by freeing all of memory...

HEAD. DESK. We're screwed.

Okay, obviously the xv6 authors figured this out already. The trick is that we do have *some* physical memory we can write to: everything between `end` and 4 MB. So we can free that part for now, allocate some of those pages for a fresh page directory and some pages, then use those pages to map the rest of physical memory, then come back later and free those pages.

So we'll have to split up the work of setting up the new page directory into two very similar functions, `kinit1()` and `kinit2()`. The first one will initialize the lock for the free list but make `kmem.use_lock` false so we don't use a lock in the early stages of kernel setup. The second one will set it to true so we start using a lock to allocate and free pages once we have multiple CPUs, a scheduler, interrupts, etc.

Both of them will use `freerange()` to free the pages in a section of physical memory. `main()` calls `kinit1()` with arguments to free the range from `end` to 4 MB, and calls `kinit2()` with arguments for the range from 4 MB to `PHYSTOP`.

```
void kinit1(void *vstart, void *vend)
{
 initlock(&kmem.lock, "kmem");
 kmem.use_lock = 0;
 freerange(vstart, vend);
}

void kinit2(void *vstart, void *vend)
{
 freerange(vstart, vend);
 kmem.use_lock = 1;
}
```

## Summary

This whole file was just to set up page allocation for the new page directory we're gonna replace `entrpgdir` with. It uses a free list in `kmem`; freeing a page adds it to the front of the list and

allocation pops a page off the front. We have to populate the free list with pages for all of physical memory, but we do that in two steps to avoid some bootstrap issues.

Again, this is a *page* allocator, not a *heap* allocator like `malloc()`, but many heap allocator implementations use linked lists of free heap regions in the same way. We talked about use-after-free bugs above, but now we can also see why *double-frees* (in which you free the same memory region more than once) can cause bugs and security vulnerabilities: they add the same region to it twice, which then might get allocated to two different variables or processes, which might ruin the per-process isolation that virtualization is supposed to provide. In addition, our page allocator handles fixed-size regions, but a heap allocator needs to use variable regions, so when a memory region gets allocated twice after a double-free, it might get split up into differently-sized pieces, of which some parts get allocated to other processes, etc... It's just a nightmare.

Next up, we'll see the full story of virtual memory.

# The Beginning: Entry and Paging

## xv6's Memory Layout

The whole point of virtualizing memory is to give users the illusion that they can roam freely across a limitless field of memory without worrying their pretty little heads about such boring details as how much physical memory their machine actually has, or where kernel code is stored, or the fact that their seemingly-continuous heap space is actually shattered into tons of tiny pages spread out in possibly random parts of physical memory. As long as user code is well-behaved, that illusion should hold up; if they do a no-no we'll just smack them with a segmentation fault.

One downside is that the kernel also has to use virtual memory, so we're faced with the potentially-complicated challenge of setting things up in physical memory without knowing where anything is actually located in physical memory! So xv6 does something that a lot of OSes do: it sets itself up as a higher-half kernel. That means that in the virtual address space (from 0 to 4 GB), the kernel will reside in the upper half starting at 2 GB, i.e. address 0x8000\_0000 and up; user code will start at 0 and end at 2 GB. Because of this, KERNBASE is defined in [memlayout.h](#) as 0x8000\_0000.

Then it sets up paging so that all of physical memory is identity-mapped to virtual memory starting at 0x8000\_0000. This makes it really convenient for the kernel to figure out the physical address of a virtual address it's using; just subtract KERNBASE and you're done. The V2P and V2P\_WO macros defined in [memlayout.h](#) do just that, and the P2V and P2V\_WO add KERNBASE to a physical address to get the kernel virtual address.

Note that I said "kernel virtual address", not just any old virtual address. Users don't get these kinds of fancy privileges, because they shouldn't be worrying about where anything is in physical memory. They're running through a limitless field of virtual memory, remember? So user virtual addresses between 0 and 2 GB will get mapped to totally arbitrary locations in physical memory.

One consequence of this is that xv6 is limited to no more than 2 GB of physical memory (instead of the 4 GB that 32-bit addresses allow for) in order to map it all into the top 2 GB of virtual memory. In reality, it's even less, for two reasons: (1) we also need to map device I/O regions into virtual memory, so it'll be a little less than 2 GB, and (2) it's hard and annoying to figure out how much physical memory is actually present on any given machine, so xv6 just says to hell with all that and picks the totally arbitrary value of a puny 224 MB as the amount of available physical memory (that's PHYSTOP, defined in [memlayout.h](#)).

## Paging

Remember when we talked about segmentation, and how we said we'd come back to paging later? Guess what? It's later.

So all virtual addresses are really "logical addresses", and segmentation turns those into "linear addresses". In xv6, the boot loader set up the segmentation hardware to use an identity map, so virtual addresses are the same as logical addresses are the same as linear addresses. Now paging has to turn those linear addresses into physical addresses. Just like segmentation uses a GDT and the segment registers for its mapping, paging uses a page directory, page tables, and the %cr3 register.

First, imagine a world where every single time some user code throws up an address (maybe it looks up a variable, or it calls a function, or it simply needs to execute the next instruction), the CPU has to stop what it's doing, save all the user's register contents, load up some kernel code, restore its register contents, find out where its stack is, get it running, and then ask the OS where that virtual address is actually located in physical memory. That would be so slow. We don't want that. We want the hardware to do all the address conversions by itself, and involve the OS only minimally to set up a new page directory when it starts a new process.

Instead, the x86 hardware uses one of its control registers, `%cr3`, to store a pointer to a page directory in memory. Then every time it needs to map a linear address to a physical one, it goes to that page directory and grabs the relevant entry. That entry is a pointer to a page *table* somewhere else in memory, so the processor grabs the right entry from there, which points to a 4096-byte page in some other location.

A linear address has a three-part structure: the 10 most significant bits are an index that picks an entry from the page directory, the next 10 bits are an index to pick an entry from whatever page table we've been directed to, and the last 12 bits are an offset that determines where to look in the page that the page table entry pointed to.

For example, let's say we have a virtual address like `0x9C4A_02BF`. If we convert to binary, split it up, and convert back to hex, we can see that the 10 most significant bits are `0x271`, the next 10 are `0x0A0`, and the last 12 are `0x2BF`. So the paging hardware would look at wherever `%cr3` is pointing to find the page directory; let's just call it `pgdir`. Then it would take entry `pgdir[0x271]` and go look wherever that's pointing to find the right page table; let's call that `pgtab271`. Then it would take entry `pgtab271[0x0A0]` and look wherever that's pointing to find the right page, `pg`. Then it would finally know that the corresponding physical address is `pg + 0x2BF`. Whew.

This still sounds super slow, so the paging hardware uses a cache called the Translation Lookaside Buffer (TLB) to store recently-used mappings and make them faster in the future. Since pages are 4096 bytes, it only needs to map a new page if the addresses some code is asking for crosses a page boundary.

xv6 provides two macros, `PDX` and `PTX` defined in [`mmu.h`](#), to recover just the page directory index bits or the page table index bits, respectively, from a virtual address.

Finally: an important aspect of virtual memory is that each process should be isolated from the others, and the kernel should be isolated from user processes. So each process will get its own page directory, and each entry of that page directory will say whether it's present (i.e., mapped) or not. If it's present, then it points to a page table for that process; if it's not present and we try to access it, we'll get a page fault or a general protection fault. Each entry in a page table will also say whether that page is present and what kinds of permissions it has. The bit flags for the permissions are (in order from least to most significant bit):

- Bit 0: present.
- Bit 1: read/write.
- Bit 2: user (otherwise only the kernel can access it).
- Bit 3: write-through.
- Bit 4: cache disabled.

- Bit 5: accessed (for the TLB).
- Bit 6: page size (we'll talk about this later).
- Bit 7: (unused).

This way, since each process has its own page directory, page tables, and pages, and each level has specific permissions set, they should never be able to interfere with each other.

Again, most of the time, the kernel will just happily ignore all this and use the mapping in the higher half of virtual memory for simplicity. Each user process's page directory will have the same mapping in the higher half so that the kernel can keep doing what it's doing no matter which user process is currently running.

Anyway, back to the code! We left off after the boot loader had finished loading the kernel into memory; it ended by calling an `entry()` function in the kernel. We haven't set up paging yet, so that's next on our to-do list. But first, the kernel is compiled and linked using a *linker script*, so we'll have to look at that to understand how that sets up memory the way we want it.

## kernel.ld

The gory details of linker scripts as a whole are outside the scope of these posts, so I'm gonna gloss over a lot of the parts of this file and focus on the important pieces.

It's important to understand what a linker does in a rough sense, so I'll just generalize and wave my hands around and say that a compiler takes code in a high-level language and converts it to assembly, an assembler takes that assembly code and turns it into machine code, and a linker takes a whole bunch of machine code files (including any code for library functions) and links them all together into a single executable file.

Linking involves three steps that are important for us here: first, the linker has to assign each piece of code a location in memory, so that different variables, functions, etc. don't end up colliding; then it replaces references to that object with its address. Second, it has to resolve any outstanding symbols (variables, functions, etc.) in each file by looking them up in all the other files and replacing them with those addresses; the linker can define its own symbols too. Third, it has to create an output file in a format that the OS can use, like ELF.

xv6 has decided that command-line flags are too basic for it, so instead it'll use a linker script [`kernel.ld`](#) for the GNU linker.

We start off by specifying the output format (32-bit ELF), the architecture (x86, also known as i386), and the entry point to start executing code. The convention is to call the entry point `_start`; the ELF header will include its address, which is how we were able to call it from the boot loader.

```
OUTPUT_FORMAT("elf32-i386", "elf32-i386", "elf32-i386")
OUTPUT_ARCH(i386)
ENTRY(_start)
```

Next up come the sections. Remember the ELF sections `text`, `rodata`, `data`, `bss`, and `stab?` Well we've gotta tell the linker where to set them up in memory, using commands like `. = address`. These are virtual addresses, so since we want to set up our kernel in the higher half of virtual memory, we'll tell it to link the code start at `0x8010_0000`. Again, we use that address

instead of 0x8000\_0000 (which maps to physical address 0) because we have to avoid the address spaces of the boot loader and the memory-mapped I/O devices.

We can also tell the linker where in physical memory the code should be placed (in linker script lingo, its "load address") using the AT( address ) command. We'll use the physical address 0x0010\_0000, since that maps to virtual address 0x8010\_0000.

```
SECTIONS {
 . = 0x80100000;

 .text : AT(0x100000) {
 /* this part tells the linker which files to include in this section */
 }

 /* more sections here... */
}
```

There's one other detail we should check out: the linker can create its own symbols using the PROVIDE(symbol = .) command. If the code happens to declare its own variable `symbol`, then the linker will just throw away its own version of it, but if the code uses `symbol` without defining it, then the linker will replace those references with the contents of that memory location.

```
SECTIONS {
 /* virtual address and text sections are defined as above */

 PROVIDE(etext = .); /* etext will be at the address right after the end
 of the text section */

 /* rodata, stab, and stabstr sections defined here */

 PROVIDE(data = .); /* data will be at the address at the very beginning
 of the data section */

 /* data section defined here */

 PROVIDE(edata = .); /* edata will be at the address right after the end
 of the data section */

 /* bss section defined here */

 PROVIDE(end = .); /* end will be at the very last address at the end
 of the entire kernel code */
}
```

Those variables will be used later in the kernel code; not so much for their contents but for their addresses, as pointers to the virtual addresses of specific parts of the kernel's code in memory. On to the kernel!

## entry.S

I have bad news. That `entry()` function that the boot loader called? It's in assembly again. :(

## Multiboot Header

Okay, so first off, we've got some more hideous specs to deal with for a bit in the form of a multiboot header. Multiboot is a specification that lets boot loaders load up kernel code in a

standardized way; the GNU boot loader GRUB uses it. So this part is mostly here in case you want to run xv6 on real hardware using GRUB; feel free to skip to **entry()** below.

The original Multiboot specification has since been replaced with Multiboot 2, but again, it's 1995, so we don't know about that yet.

Multiboot helps compliant kernels and boot loaders identify each other using a special header. The header must be completely contained in the first 8192 bytes of the kernel's image, and it must be 32-bit aligned. The header contains three things: (1) a magic number used for mutual identification and recognition (0x1BADB002 for kernels, 0x2BADB002 for boot loaders), (2) some flags for the kernel to inform the boot loader what the kernel requires in order to run successfully, and (3) a 32-bit unsigned checksum which when added to the other two fields must have a 32-bit unsigned sum of zero. Depending on the flags that are set, there may be other components to the Multiboot header.

So we'll start by creating a `multiboot_header` label at the beginning of the file (and thus, the beginning of the kernel image) and making sure it's aligned to 32 bits.

```
.p2align 2 # Force 4-byte alignment
.text
.globl multiboot_header
multiboot_header:
...
```

Now we'll just add the magic number, set the flags to 0 to indicate no special requirements, and add the checksum.

```
#define magic 0x1badboo2
#define flags 0
.long magic
.long flags
.long (-magic-flags)
```

And that's it!

## entry

Back in [kernel.ld](#), we said that the linker would set up the kernel's ELF header to specify the kernel's entry point using `_start`, but `_start` itself wasn't actually defined there, so we have to do that first. We don't know where this code will end up in memory, so we'll define an `entry` label and set `_start` to the address of `entry`. Note that the linker script used virtual addresses in the higher half, but we haven't set up paging yet, so we'll have to convert it to a physical address using one of the macros we mentioned earlier.

```
.globl _start
_start = V2P_W0(entry)
.globl entry
```

Next up we want to finish setting up virtual memory by enabling paging, but that's all kinds of complicated, so we're gonna start off with a super simple version of paging. Part of that difficulty is that there's a bootstrap problem: we need to allocate pages to hold the page tables themselves, but we can't use pages without page tables... uhh...

We'll solve that by starting off with a basic, super-simple page directory where only two entries are mapped: the first entry maps virtual addresses 0 to 4 MB to physical addresses 0 to 4 MB, and the

second entry maps virtual addresses KERNBASE to KERNBASE + 4MB to physical addresses 0 to 4 MB. One consequence is that the entire kernel code and data has to fit in 4 MB.

Why the two entries pointing to the same place? It's to solve another bootstrap problem. The kernel is currently running in physical addresses close to 0. Once we enable paging and start using virtual addresses in the higher half, the stack pointer %esp, instruction pointer %eip, even the pointer in %cr3 to the page directory itself will all still point to low addresses until we update them. But updating them requires executing instructions, which would require accessing low addresses a few more times. If we left out the low addresses, we'd get a page fault, and since we don't have exception handlers set up yet, that would cause a double fault, which would turn into the dreaded **TRIPLE FAULT**, in which the processor enters an infinite reboot loop. So yeah, point is, we need both the low and high mappings for now; we'll get rid of the low mappings once we're done setting up.

But wait! Aren't page directory entries supposed to point to page tables? How can they point directly to pages here? It turns out that x86 can skip that second layer altogether if we use so-called "huge" pages of 4 MB in size instead of the usual 4 KB. In the long run, this could lead to internal fragmentation, but it does cut down on the overhead and allows a faster set-up. Plus we're only gonna use them for a minute while we get ready for the full paging ordeal.

To use 4 MB pages, we have to enable x86's Page Size Extension (PSE) by setting the fourth bit in the %cr4 register. CR4\_PSE is defined in [mmu.h](#) as 0x10, or 00010000 in binary.

```
entry:
 movl %cr4, %eax
 orl $(CR4_PSE), %eax
 movl %eax, %cr4
```

We need a page directory before we can set up paging; again, basic version now, full glorious page directory later. We're gonna do the same thing we did in the boot loader where we tell the processor to load the page directory now but then procrastinate actually writing it; this time, we'll write it in C and call it `entrypgdir`. Then we'll load its physical address into register %cr3.

```
 movl $(V2P_W0(entrypgdir)), %eax
 movl %eax, %cr3
```

Now we can enable (a basic version of) paging! We tell the CPU to start using the page directory in %cr3 by setting bit 31 (paging) of register %cr0; we can also set bit 16 (write protect) of the same register to prevent writing to any pages that the page directory and page tables have marked as read-only. CR0\_PG and CR0\_WP are defined in [mmu.h](#) to set these bits.

```
 movl %cr0, %eax
 orl $(CR0_PG|CR0_WP), %eax
 movl %eax, %cr0
```

Now remember how the processor is still running at low addresses? Yeah, let's fix that. First we'll make a new kernel stack in the higher half that will still be valid even after we get rid of the lower address mappings. We'll have the linker save some space for us under the symbol `stack` and set it up there; KSTACKSIZE is defined in [param.h](#) as 4096 bytes. So we just set the stack pointer register %esp to the top of that section in order to let the stack grow down toward the address of `stack`. Again, we'll procrastinate actually defining `stack`.

```
 movl $(stack + KSTACKSIZE), %esp
```

Now we want to call into the `main()` function, but we don't just want to do that the usual assembly way of `call main`. That would generate a jump relative to the current value of `%eip`, which is still in low addresses. We'll use an indirect jump instead.

```
 mov $main, %eax
 jmp *%eax
```

Finally, we need to get around to reserving space for the stack. We can do that with the assembler instruction `.comm symbol, size`:

```
.comm stack, KSTACKSIZE
```

## main.c

Awesome, back to C code now! Remember how we procrastinated actually defining `entrypgdir`? Let's do that now; it's at the bottom of [main.c](#).

### entrypgdir

What in the world is this?!

```
__attribute__((__aligned__(PGSIZE)))
pde_t entrypgdir[NPDENTRIES] = {
 [0] = (0) | PTE_P | PTE_W | PTE_PS,
 [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
};
```

Okay, bear with me; I promise it's not too bad.

First, the `__attribute__` tells the compiler and linker that the page directory should be placed in memory at an address that's a multiple of `PGSIZE` (4096 bytes); that's just a requirement of the paging hardware.

Next, we define `entrypgdir` as an array of `NPDENTRIES` (1024, according to [mmu.h](#)), each of type `pde_t` (a type alias for `unsigned int`, according to [types.h](#)).

Then we initialize the entries: in C, you're allowed to initialize an array by specifying the values of specific entries; all other entries become zero. You specify an entry by putting its index in square brackets before its value, so `[2] 5` will set the entry with index 2 to be 5. Here we initialize the entries with indices 0 and `KERNBASE >> PDXSHIFT`, which is the same thing as `PDX(KERNBASE)`, AKA the page directory index corresponding to the virtual address `KERNBASE`, AKA `0x8000_0000`. So basically, we've initialized the page directory entries corresponding to the low virtual address 0 and the high virtual address `KERNBASE`.

We set their value to 0, because we want them to map to physical addresses from 0 up to 4 MB. Oh, and remember how page directories and page tables can also hold permission flags? We want to set flags to say that these pages are present (so that accessing them doesn't cause a page fault), writeable, and 4 MB in size; those are defined in [mmu.h](#) as `PTE_P`, `PTE_W`, and `PTE_PS`. We can combine them all together by bitwise-ORing them.

And we're done!

## main

The code in [entry.S](#) finished up by calling into the C function `main()`, which is where the core set-up happens before we can start running processes. It calls into basically every single part of the xv6 kernel, so we can't go through all the functions line-by-line yet; instead I'll just give you an overview of what they do.

- `kinit1()` solves another bootstrap problem around paging: we need to allocate pages in order to use the rest of memory, but we can't allocate those pages without first freeing the rest of memory, which requires allocating them... You see what I mean. This function will free the rest of memory between the `end` of the kernel code (defined in [kernel.ld](#), remember?) and 4 MB.
- `kmalloc()` allocates a page of memory to hold the fancy full-fledged page directory, sets it up with mappings for the kernel's instructions and data, all of physical memory, and I/O space, then switches to that page directory (leaving poor old `entrypgdir` in the trash).
- `mpinit()` detects hardware components like additional CPUs, buses, interrupt controllers, etc. Then it determines whether this machine supports this crazy new idea where you can have multiple CPU cores. Wow, 1995 is crazy.
- `lapicinit()` programs this CPU's local interrupt controller so that it'll deliver timer interrupts, exceptions, etc. when we're ready for them later.
- `seginit()` sets up this CPU's kernel segment descriptors in its GDT; we still won't really use segmentation, but we'll at least use the permission bits.
- `picinit()` disables the *ancient* PIC interrupt controller that literally no one has ever used since the APIC was introduced in 1989. I don't even know what to say. I guess I was mistaken when I assumed it was 1995; I don't know.
- `ioapicinit()` programs the I/O interrupt controller to forward interrupts from the disk, keyboard, serial port, etc., when we're ready for them later. Each device will have to be set up to send its interrupts to the I/O APIC.
- `consoleinit()` initializes the console (display screen) by adding it to a table that maps device numbers to device functions, with entries for reading and writing to the console. It also sets up the keyboard to send interrupts to the I/O APIC.
- `uartinit()` initializes the serial port to send an interrupt if we ever receive any data over it. xv6 uses the serial port to communicate with emulators like QEMU and Bochs.
- `pinit()` initializes an empty process table so that we can start allocating slots in it to processes as we spin them up.
- `tvinit()` sets up and interrupt descriptor table (IDT) so that the CPU can find interrupt handler functions to deal with exceptions and interrupts when they come.
- `binit()` initializes the buffer cache, a linked list of buffers holding cached copies of disk data for more efficient reading and writing.
- `fileinit()` sets up the file table, a global array of all the open files in the system. There are other parts of the file system that need to be initialized like the logging layer and inode layer, but those might require sleeping, which we can only do from user mode, so we'll do that in the first user process we set up.
- `ideinit()` initializes the disk controller, checks whether the file system disk is present (because both the kernel and boot loader are on the boot disk, which is separate from the disk with user programs), and sets up disk interrupts.

- `startothers()` loads the entry code for all other CPUs (in [entryothers.S](#)) into memory, then runs the whole setup process again for each new CPU.
- `kinit2()` finishes initializing the page allocator by freeing memory between 4 MB and `PHYSTOP`.
- `userinit()` creates the first user process, which will run the initialization steps that have to be done in user space before spinning up a shell.
- `mpmain()` loads the interrupt descriptor table into the CPU so that we're finally completely ready to receive interrupts, then calls the `scheduler()` function in [proc.c](#), which enables interrupts on this CPU and starts scheduling processes to run. `scheduler()` never returns, so at that point we're completely done with setup and we're running the OS proper.

## Summary

The entry code in the xv6 kernel had one job: to set up paging. It kind of failed at that job, but not for lack of trying! There are just all kinds of Catch-22s when it comes to paging, so at least it got us partway there by making a temporary page directory to tide us over until we can throw it away and never look back.

We also took a sneak peek at all the setup code in `main()`; we're gonna end up going through it all, but at least now you should have enough of an idea of what's going on that you can more or less skip around and look at what you need.

# Devices: Disk Driver

At this point, we've seen how xv6 virtualizes memory and the processor to give each user process the illusion of a contiguous, near-infinite memory space and a dedicated CPU to run it; we've also seen how xv6 mediates interactions between most of a computer's hardware components and user processes via system calls. But there's one more piece of hardware that's critically important for an OS that we haven't looked at yet: the disk. All that's left in the kernel code for us to look at is how xv6 manages data storage on the disk and how it presents that data to users in a simplified way.

The function of a disk is to provide *persistence* for an operating system. RAM is volatile memory: it gets erased when the machine is turned off, so any data stored there is fleeting. A disk allows an OS to store and retrieve data across shut-offs. The disk driver we'll go over in this post allows the xv6 kernel direct access to that device so it can read and write data to it.

But unlike other devices, a simple driver isn't enough here. We don't just need to be able to read and write data; we'd like to present users with a simplified, accessible framework to navigate that data. Imagine using a computer where you had to specify which byte of the disk to read or write, then remember that yourself in order to access it again later. It's madness! Enter file systems; "files" don't really exist in any real sense on a disk, but the OS can provide the illusion of discrete, individual files in order to simplify access to data.

We also need to make sure concurrent accesses of the same file don't risk corrupting the file (or even the entire file system). We need to separate out kernel data (like the kernel code itself) from user data on the disk, so that a malicious user process can't just overwrite arbitrary kernel code. Finally, there's that oh-so-famous line about Unix systems, "everything is a file". We'll need a way to present "everything" in the elegant abstraction of a file.

All of these abstractions and security checks will require far more code than a simple driver to implement them, so before we go on to the driver, let's check out how xv6 will organize its file system to get a preview of what's ahead.

## File System Organization

Laying the abstraction of a complete file system on top of a physical disk will require several steps. xv6 does this using seven layers. From bottom (direct hardware interaction) to top (user-facing code), they are:

- Disk driver: reads and writes blocks on an IDE hard drive.
- Buffer cache: caches disk blocks in memory and synchronizes access to them.
- Logging: provides atomic disk writes to mitigate the risk of a crash.
- Inodes: turns disk blocks into individual files that the OS can manipulate.
- Directories: creates a tree of named directories that contain other files.
- Path names: provides hierarchical, human-readable path names in the directory tree structure.
- File descriptors: abstracts OS resources like pipes and devices as files to provide a unified API for user programs.

That's a lot of work to do now, but it'll pay off! The kernel will do all this labor so that users are free to be lazy later on and can live in blissful ignorance of the fact that their precious little files actually exist as nothing but ones and zeroes in totally arbitrary locations on the disk.

Note that hard drives are usually divided into *sectors*, which are physical divisions (originally referring to literal geometric sectors), traditionally of 512 bytes. Operating systems can then collect these into larger *blocks* which are multiples of the sector size. xv6 uses 512-byte blocks for simplicity so that the sector and block sizes match up; I'll use the two terms interchangeably.

On the disk, block 0 usually contains the boot sector, so it's not used by xv6 (but remember the Makefile -- xv6 actually stores the boot loader and kernel code on an entirely separate physical disk). Block 1 is called the *superblock* because it contains metadata about the file system like its total size, the size of the log, the number of files, and their location on the disk. Then the log starts at block 2 and on.

## buf.h

If you've read any of the previous optional posts on device drivers, you know that interacting directly with the hardware means all kinds of opaque code with seemingly-arbitrary port I/O and cryptic magic numbers. Drivers are also specific to the actual (or virtual) hardware in the machine that xv6 will run on, so it tends to be less useful for showing general OS concepts -- hence why all the other device driver posts were optional. That being said, the disk driver nicely rounds out the rest of the file system code, so I recommend checking it out, but if you're short on time or bored with all the talk about hardware specs, feel free to skip to the summary section below.

Reading and writing disk data is super slow, so the second layer in the file system is the buffer cache, which will store copies of disk blocks in memory for faster access. But we still have to read from the disk to create that buffer, and we still have to write any modified data to the disk once we're done, so we still need a layer below the buffer cache to do that. That layer is the disk driver; its purpose is to copy data from the disk to the in-memory cache and vice versa. A single block is represented in the cache as a `struct buf`, defined in [buf.h](#).

```
struct buf {
 int flags;
 uint dev; // device number
 uint blockno; // block number (same as sector number)
 struct sleeplock lock; // sleep-lock to protect buffer reads and writes
 uint refcnt; // how many processes are using this buffer
 struct buf *prev; // for use with buffer cache doubly-linked list
 struct buf *next; // for use with buffer cache doubly-linked list
 struct buf *qnext; // for use with disk driver queue
 uchar data[BSIZE]; // data stored in the buffer
};

#define B_VALID 0x2
#define B_DIRTY 0x4
```

The two constants defined at the bottom are used in the `flags` field; `B_VALID` indicates that a buffer has been read from disk and should accurately reflect the sector's contents on the disk, and `B_DIRTY` says we've modified the buffer but haven't yet updated the on-disk version of a file, so we need to write the buffer to disk soon.

We'll see later on that the buffer cache uses a doubly-linked list of buffers; the `prev` and `next` fields are used there. However, the disk driver also maintains its own queue of buffers that are waiting to be read from or written to the disk; that's implemented as a singly-linked list using the `qnext` field.

## ide.c

We've already seen some code to read and write disk data in the [boot loader](#); I know it's been a while, so you can check that out again if you want. We can't reuse the code there for a few reasons, though: (1) the boot loader has to be compiled separately from the kernel, so we can't access any of the functions there, and (2) we need to store data in the buffer cache, so we can't even copy-paste the code we used before since the boot loader barely even knows what memory is, let alone a buffer cache.

### ATA Programmed I/O Mode

Modern disk drivers usually talk to the disk via direct memory access (DMA), but to keep things simple xv6 is just gonna talk to it with port I/O. That's much, much slower, and it requires active participation by the CPU (which means it can't do anything else at the same time), but hey, xv6 thinks it's 1995, remember? So PIO mode is still (relatively) cutting edge. Either way, extreme performance isn't the goal here, so we'll just have to suck it up.

Okay, let's do a super-quick summary. `inb` is a C wrapper for an x86 assembly instruction that reads a single byte of data from a port; `outb` writes a byte to a port. The disk controller chip has primary and secondary buses; the primary bus sends data on port 0x1F0 and has control registers on ports 0x1F1 through 0x1F7. Port 0x1F7 doubles as a command register and a status port with some useful flags we can check in order to know what the disk is up to; we saw some of those before, but I'll give you the full list now.

- Bit 0 (0x01) - ERR (indicates an error occurred)
- Bit 1 (0x02) - IDX (index; always set to zero)
- Bit 2 (0x04) - CORR (corrected data; always set to zero)
- Bit 3 (0x08) - DRQ (drive has data to transfer or is ready to receive data)
- Bit 4 (0x10) - SRV (service request)
- Bit 5 (0x20) - DF (drive fault error)
- Bit 6 (0x40) - RDY (ready; clear when drive isn't running or after an error and set otherwise)
- Bit 7 (0x80) - BSY (busy; drive is in the middle of sending/receiving data)

The disk driver defines some of these with preprocessor macros at the top of the file.

```
#define SECTOR_SIZE 512
#define IDE_BSY 0x80
#define IDE_DRDY 0x40
#define IDE_DF 0x20
#define IDE_ERR 0x01
// ...
```

We also saw one command example in the boot loader: sending 0x20 to port 0x1F7 tells the disk to read a sector and send it to us through data port 0x1F0. Now we'll also use commands to write a sector, as well as to read or write multiple sectors at once.

```
// ...
#define IDE_CMD_READ 0x20
#define IDE_CMD_WRITE 0x30
#define IDE_CMD_RDMUL 0xc4
#define IDE_CMD_WRMUL 0xc5
// ...
```

If, for some reason beyond mortal comprehension, you decide you want to know more about the eldritch secrets of ancient hard drives, you can read [this resource on ATA disks](#).

After those constants, we find three static global variables: a spin-lock for accessing the disk, the queue of buffers waiting to be synchronized with their on-disk counterparts, and a boolean to track whether xv6 is running with only disk 0 (boot loader and kernel) or with disk 1 (user file system) as well.

```
// ...
static struct spinlock idelock;
static struct buf *idequeue;
static int havedisk1;
// ...
```

## idewait

This function takes an integer `checkerr` argument that should be a boolean and waits for the disk to be ready to receive more commands. If `checkerr` is true, it'll also check whether the status port includes any error flags.

It starts by reading from the disk's status port and looping until the busy flag is not set but the ready flag is. The bitwise-OR `IDE_BSY | IDE_DRDY` combines both flags, and the bitwise-AND tests whether either one is set in `r`.

```
static int idewait(int checkerr)
{
 int r;
 while (((r = inb(0x1f7)) & (IDE_BSY | IDE_DRDY)) != IDE_DRDY)
 ;
 // ...
}
```

Now if `checkerr` is nonzero we have to check that neither the error nor the drive failure flag is set in the status port. If either one is set, we'll return -1; we'll return 0 otherwise.

```
static int idewait(int checkerr)
{
 // ...
 if (checkerr && (r & (IDE_DF | IDE_ERR)) != 0) {
 return -1;
 }
 return 0;
}
```

## ideinit

This function is called by the kernel's `main()` during set-up to initialize the disk. We start by initializing the disk lock, then tell the I/O interrupt controller to forward all disk interrupts to the last

CPU. We talked about the `ioapicenable()` function in detail in the post on interrupt controllers.

```
void ideinit(void)
{
 initlock(&idelock, "ide");
 ioapicenable(IRQ_IDE, ncpu - 1);
 // ...
}
```

Then we wait for the disk to be ready to accept commands (ignoring any error flags that may be present).

```
void ideinit(void)
{
 // ...
 idewait(0);
 // ...
}
```

We said above that disk 0 should contain the boot loader and kernel, so we can assume any machine running xv6 should have that present. However, we need to make sure disk 1 is present; the [Makefile](#) includes some configurations like `make qemu-memfs` under which xv6 can run without a dedicated disk for the file system, storing files in memory instead.

Port 0x1F6 is used to select a drive. Bits 5 and 7 should always be set, and bit 6 picks the right mode we need to indicate a disk. Bit 4 determines whether we want to select disk 0 or disk 1. So we can select drive 1 by setting bits 5-7 (0xE0 when combined), then bit 4 (`1 << 4`).

```
void ideinit(void)
{
 // ...
 outb(0x1f6, 0xe0 | (1 << 4));
 // ...
}
```

Now we need to wait for disk 1 to be ready; we need to handle this as a special case since `waitdisk()` can't check a specific disk for us, and because an absent disk 1 would make the while loop there continue forever. So we'll check the status register 1000 times; if it ever reports that it's ready, we'll set `havedisk1` to true and break, but otherwise we'll assume disk 1 isn't present and leave `havedisk1` as zero (i.e., false).

```
void ideinit(void)
{
 // ...
 for (int i = 0; i < 1000; i++) {
 if (inb(0x1f7) != 0) {
 havedisk1 = 1;
 break;
 }
 }
 // ...
}
```

Finally, we'll switch back to using disk 0 by changing the fourth bit of the register at port 0x1F6.

```
void ideinit(void)
{
```

```

 // ...
 outb(0x1f6, 0xe0 | (0 << 4));
}

```

## idestart

This is the core function that will read or write a buffer to or from the disk. It's a `static` function, so it can only be called by other functions in this file; `ideintr()` and `iderw()` will both use it as a helper function. It takes a pointer to a buffer, so the first thing to do is make sure that pointer isn't null. We'll also make sure the buffer's block number is within the maximum limit set by `FSSIZE`, defined in [param.h](#) as 1000.

```

static void idestart(struct buf *b)
{
 if (b == 0) {
 panic("idestart");
 }
 if (b->blockno >= FSSIZE) {
 panic("incorrect blockno");
 }
 // ...
}

```

Next we need to figure out which disk sector to read from or write to. Since xv6 uses blocks that are the same size as a sector, this should just be `b->blockno`, but we'll add a conversion here in case that gets changed later on (especially if we want higher disk throughput).

```

static void idestart(struct buf *b)
{
 // ...
 int sector_per_block = BSIZE / SECTOR_SIZE;
 int sector = b->blockno * sector_per_block;
 // ...
}

```

If each block fits exactly one sector, then we'll need to use the single-sector read and write commands; otherwise we should use the multi-sector versions of those commands. We'll set `read_cmd` and `write_cmd` to the right versions. We'll also make sure that there are no more than 7 sectors per block.

```

static void idestart(struct buf *b)
{
 // ...
 int read_cmd = (sector_per_block == 1) ? IDE_CMD_READ : IDE_CMD_RDMUL;
 int write_cmd = (sector_per_block == 1) ? IDE_CMD_WRITE : IDE_CMD_WRMUL;
 if (sector_per_block > 7) {
 panic("idestart");
 }
 // ...
}

```

Now let's wait for the disk to be ready, ignoring any error flags.

```

static void idestart(struct buf *b)
{
 // ...
 idewait(0);
 // ...
}

```

```
}
```

Okay, now it's time to brace yourself, because this next part is a hot mess of port I/O operations with lots of magic numbers. First we'll tell the disk controller to generate an interrupt once it's done reading or writing by setting the device control register at 0x3F6 to zero. Then we'll tell it how many total sectors we want to read or write by writing that number (AKA `sector_per_block`) to port 0x1F2.

```
static void idestart(struct buf *b)
{
 // ...
 outb(0x3f6, 0); // generate interrupt when done
 outb(0x1f2, sector_per_block); // number of sectors to read/write
 // ...
}
```

Before sending the read or write command, we have to tell the disk which sector to read from, using our `sector` variable from above. Let's take a second to talk about hard drive geometry. A hard drive consists of a bunch of stacked circular surfaces, where each surface has a corresponding *head* that changes its position to read or write from the right place on the disk. Each surface has a number of *tracks*: concentric circles that contain data. If you pick a track number (i.e. pick a distance from the center of the surfaces) and collect all those tracks from all the surfaces, you get a *cylinder*.

A sector number acts as a kind of address with each part specifying a different geometric component, similar to how linear addresses contain a page directory index, page table index, and offset. The eight most significant bits (24 through 31) identify the drive and/or head that the sector is located on (plus some flags); bits 8 through 23 identify the cylinder, and bits 0 through 7 pick a sector within that cylinder. Altogether, these define a 3D coordinate system that uniquely identifies all sectors on a machine's disks.

Port 0x1F3 is the sector number register, ports 0x1F4 and 0x1F5 are the cylinder low and high registers, and port 0x1F6 is the drive/head register. We can write the sector number as `sector & 0xFF`; the cylinder low and high numbers can be recovered by bitshifting `sector` down by 8 and 16, respectively.

```
static void idestart(struct buf *b)
{
 // ...
 outb(0x1f3, sector & 0xff); // sector number
 outb(0x1f4, (sector >> 8) & 0xff); // cylinder low
 outb(0x1f5, (sector >> 16) & 0xff); // cylinder high
 // ...
}
```

Now for the drive/head register, we'll use `b->dev` to get the block's device and (`sector >> 24`) to get the head it's on. Finally, we'll set bits 5-7 as required (and as mentioned above in `ideinit()`) with 0xE0. Then we can bitwise-OR all of these together and write them to port 0x1F6.

```
static void idestart(struct buf *b)
{
 // ...
 outb(0x1f6, 0xe0 | ((b->dev & 1) << 4) | ((sector >> 24) & 0x0f));
 // ...
}
```

```
}
```

Okay, that was the worst of it! Deep breath now. The last part is just sending the actual read or write command. But how do we know which one we're supposed to do? The only argument is a pointer to a buffer `b`, not any sort of boolean that might tell us which to carry out. Well, remember the buffer flag `B_DIRTY`? That one indicates that a buffer has been modified and needs to be written to disk. If that flag is set, reading from the disk would overwrite any changes, which probably isn't what we want. So let's just assume that the `B_DIRTY` flag means we should write to disk, and the absence of that flag means we should read from disk.

```
static void idestart(struct buf *b)
{
 // ...
 if (b->flags & B_DIRTY) {
 outb(0x1f7, write_cmd);
 outsl(0x1f0, b->data, BSIZE / 4);
 } else {
 outb(0x1f7, read_cmd);
 }
}
```

Here `outsl()` is another C wrapper for an x86 instruction; this one writes data from a string, four bytes at a time.

That's it! This is by far the most cryptic function in the disk driver; the last two are relatively easy now.

## ideintr

We saw in `idestart()` that we set up the disk to send an interrupt whenever it's done reading or writing data. Back when we looked at [trap.c](#), we saw that the `trap()` function directs all disk interrupts to the handler function `ideintr()`. It's time to check that one out now.

We'll start by acquiring the disk's spin-lock; note that we don't use a sleep- lock because this is an interrupt handler function, so interrupts should be disabled while it runs.

```
void ideintr(void)
{
 acquire(&idelock);
 // ...
 release(&idelock);
}
```

If we got an interrupt, then it usually means the disk is done with the most recent request. Those requests are stored in the global `idequeue` linked list, with the current request at the front of the queue. So we'll get the head of the queue as `b`, then set `idequeue` to point to the next buffer in the queue. If the head is null, then we'll just return early.

```
void ideintr(void)
{
 // ...
 struct buf *b;
 if ((b = idequeue) == 0) {
 release(&idelock);
 return;
 }
```

```

 idequeue = b->next;
 // ...
}

```

The read command in `idestart()` didn't specify where to read the data to, so we do that now. We'll check if the `B_DIRTY` flag was set; if it wasn't (i.e. the operation was a disk read), then we'll wait for the disk to be ready (without any errors, using `idewait(1)` instead of `idewait(0)` as we have before) and read the data into `b->data`.

```

void ideintr(void)
{
 // ...
 if (!(b->flags & B_DIRTY) && idewait(1) >= 0) {
 insl(0x1f0, b->data, BSIZE / 4);
 }
 // ...
}

```

Next, we set the `B_VALID` flag with a bitwise-OR and clear any `B_DIRTY` flag with a bitwise-AND and a bitwise-NOT. Then we'll wake up any user process that went to sleep on a channel for this buffer after requesting a disk I/O operation.

```

void ideintr(void)
{
 // ...
 b->flags |= B_VALID;
 b->flags &= ~B_DIRTY;
 wakeup(b);
 // ...
}

```

Finally, we'll get the disk started on the next operation, for the next buffer in the queue.

```

void ideintr(void)
{
 // ...
 if (idequeue != 0) {
 idestart(idequeue);
 }
 // ...
}

```

## iderw

The `idestart()` function is `static`, so it can't be called by anything outside of this file; we need to provide a mechanism for both kernel and user threads to read and write disk data. That's what `iderw()` does. Note that processes should never call this function directly; it only gets called by the code for the buffer cache layer of the file system. In other words, processes will use system calls like `open()`, `read()`, `write()`, `close()`, etc., which in turn will use functions from higher layers of abstraction, which in turn call functions from lower layers, and so on, until they reach the buffer cache, which calls `iderw()` to finally read/write directly from/to the disk.

By the time a process gets to `iderw()`, it should already be holding a sleep-lock `b->lock` for the buffer `b` it wants to read or write, and either the `B_DIRTY` flag should be set (to write to disk) or the `B_VALID` flag should be absent (to read from disk). We'll start off with some sanity checks

for those, and make sure that we're not trying to read from disk 1 if it's not present on this machine. Then we'll acquire the disk's spin-lock.

```
void iderw(struct buf *b)
{
 if (!holdingsleep(&b->lock)) {
 panic("iderw: buf not locked");
 }
 if ((b->flags & (B_VALID | B_DIRTY)) == B_VALID) {
 // B_VALID is set, so we don't need to read it; B_DIRTY is not set, so
 // we don't need to write it
 panic("iderw: nothing to do");
 }
 if (b->dev != 0 && !havedisk1) {
 panic("iderw: ide disk 1 not present");
 }

 acquire(&idelock);
 // ...
 release(&idelock);
}
```

There may be other buffers waiting in line in the disk queue, so we have to append this buffer `b` to the end of `idequeue`. We can do that by setting `b->qnext` to null, then creating a variable `pp` to traverse the entire queue. When `pp` points to the last element, we'll set its `qnext` field to point to `b`.

```
void iderw(struct buf *b)
{
 // ...
 b->qnext = 0;

 // Traverse the queue
 struct buf **pp;
 for (pp = &idequeue; *pp; pp = &(*pp)->qnext)
 ;

 // Append b to end of queue
 *pp = b;

 // ...
}
```

That traversal might look confusing as all hell, so let's take a closer look. It defines `pp` as a double pointer: a pointer to a pointer to a `struct buf`. (If you've seen the interview of Linus Torvalds where he talks about good style with linked lists, it's similar to the code there; there's a nice summary [here](#).) `pp` starts off equal pointing to `idequeue`, i.e. the head of the linked list. Each iteration checks that `pp` points to a valid (non-null) pointer, i.e. the loop will end when we reach the end of the list. The body of the loop is empty, so none of the iterations actually do anything; the purpose of the for loop is just to update `pp` several times. At the end of each iteration, `pp` is updated to point to a pointer to the next buffer in the queue.

Suppose the last buffer in the queue is `end`. At the end of the for loop, `pp` will hold the address of `end->qnext`, so `*pp = b` sets `end->qnext = b`. The double indirection makes it easy to update the last buffer in the queue; without it, we would have to stop the loop one step earlier when `pp` points to `end` instead of `end->qnext` then be careful to update the actual buffer at the end of

the queue instead of just updating the local variable `pp`. All in all, it's just an elegant way to write a linked list traversal in a single line.

Okay, so now our buffer `b` is at the end of the queue. If there are others in front of it, then `ideintr()` will make sure that each disk interrupt starts the disk on the next operation. But what if `b` is actually the only buffer in the queue? In that case, the disk isn't running yet, so we need to get it started ourselves.

```
void iderw(struct buf *b)
{
 // ...
 if (idequeue == b) {
 idestart(b);
 }
 // ...
}
```

At this point, we can be confident that the disk will either start our request now or get to it eventually (if there are other requests in the queue). This process just has to wait for the disk to finish, so we'll put it to sleep until the buffer has been synchronized with the disk. We'll check that by making sure the `B_VALID` flag is present but `B_DIRTY` is not set. The call to `sleep()` will release `idelock` and reacquire it before returning.

```
void iderw(struct buf *b)
{
 // ...
 while ((b->flags & (B_VALID | B_DIRTY)) != B_VALID) {
 sleep(b, &idelock);
 }
 // ...
}
```

## Summary

The disk driver handles direct communication with the hard drive, issuing orders to read or write sectors. It exposes two API functions, `ideintr()` and `iderw()`. The former is called by `trap()` to handle disk interrupts, while the latter is called by the code for the buffer cache layer of the file system to update blocks in the buffer cache with their corresponding sectors on disk. Next up we'll look at the buffer cache itself, as well as the logging layer, which provides crash recovery.

# DAS BOOT

First things first: in order for a computer to run xv6, we need to load it from disk into memory and tell the processor to start running it. So how does this all happen?

## The Boot Process

When you press the power button, the hardware gets initialized by a piece of firmware called the BIOS (Basic Input/Output System) that comes pre-installed on the motherboard on a ROM chip. Nowadays, your computer probably uses UEFI loaded from flash memory, but xv6 pretends like it's 1995 and sticks with BIOS. Since xv6 runs on x86 hardware, we're gonna have to satisfy all the janky requirements that come with that architecture, in addition to the BIOS's requirements.

Now the BIOS has to load some *other* code called the boot loader from disk; then it's the boot loader's job to load the OS and get it running. The boot loader has to act as a middle-man because the BIOS has no idea where on the disk you decided to put the OS.

The BIOS will look for the boot loader in the very first sector (512 bytes) of whatever mass storage device you told it to boot from, which we'll call the boot disk. The processor will execute the instructions it finds there. This means you have to make a choice: either your boot loader has to be less than 512 bytes or you can split it up into smaller parts and have each part load the next one. xv6 takes the first approach.

The BIOS loads the boot loader into memory at address 0x7C00, then sets the processor's %ip register to that address and jumps to it. Remember that %eip is the instruction pointer on x86? Okay cool. But why did I write %ip instead of %eip? Well, the BIOS assumes we're gonna be using 16 bits because of the hellscape known as backwards-compatibility, so we've gotta pretend like it's 1975 before we can pretend it's 1995. The irony here is that this initial 16-bit mode is called "real mode". So on top of loading the OS, the boot loader will also have to shepherd the processor from real mode to 32-bit "protected mode".

One last detail: we'll look at the Makefile and linker script later on, but for now just keep in mind that the boot loader will be compiled separately from the kernel, which will be compiled separately from all the user-space programs. This makes it easier to make sure that the entire boot loader will fit in the first 512 bytes on disk. Eventually, the boot loader and the kernel will be stored on the same boot disk together, and the user-space programs will be on a separate disk that holds the file system.

## bootasm.S

Boot loader space is tight, and we want to make sure our instructions are exact, so we're gonna start off in assembly. The ".S" file extension means it's gonna be assembled by the GNU assembler `as`, and we're allowed to use C preprocessor directives like `#include` or `#define` or whatever in the assembly code. Also, xv6 uses AT&T syntax, so if you read CS:APP or took the online course then it'll be familiar; if you don't know what that means, then don't worry about it.

## Getting Started

First we include some header files to use some constants; I'll point them out later. Next up, we gotta tell the assembler to generate 16-bit code, and set a global label to tell the BIOS where to start executing code.

```
.code16 # Tell compiler to generate 16-bit code
.globl start
start:
```

Next up: you know how sometimes you can press a special key to tell the BIOS to stop what it's doing and let you pick a disk to boot from? Or you move your mouse around in the BIOS menu and you see the pointer moving? Yeah, that needs hardware interrupts in order to work, but right now, we don't have the faintest clue how to handle those if they happen, so let's go ahead and turn those off. There's an x86 instruction to disable them by clearing the interrupt flag in the CPU's flags register.

```
cli
```

Now we've gotta handle some of x86's quirks. First off, we're gonna need 20-bit memory addresses, but we only have 16 bits to work with. x86 uses six segment registers %cs (code segment), %ds (data segment), %ss (stack segment), %es (extra segment), %fs and %gs (general-purpose segments) to create 20-bit addresses from 16-bit ones; we're gonna need the first four. The BIOS guarantees that %cs will be set to zero, but it doesn't make any promises about the others, so we have to clear them ourselves. We're not using %eax for anything yet, so we'll use that to clear the others. The w at the end of xorw and movw means we're operating on 16-bit words.

```
xorw %ax,%ax
movw %ax,%ds # Data segment
movw %ax,%es # Extra segment
movw %ax,%ss # Stack segment
```

This next part is a total hack for backwards-compatibility: sometimes a virtual address might get converted to a 21-bit physical address, and oh no, what are we gonna do? Well, some hardware can't deal with 21 bits, so it just ignores it, but it's 1995, so we've got fancy hardware that can use that extra bit. Wow, you really know we're in the future when you've got a whole 2 MB of RAM to work with! So we have to tell the processor not to throw away that 21st bit. The way we do that is by setting the second bit of the keyboard controller's output port to line high. I don't know. Don't ask me why. The output ports are 0x64 and 0x60, so we're gonna wait until they're not busy, then set the magic values that will make this all work.

```
seta20.1:
 inb $0x64,%al # Wait for not busy
 testb $0x2,%al
 jnz seta20.1

 movb $0xd1,%al # 0xD1 -> port 0x64
 outb %al,$0x64

seta20.2:
 inb $0x64,%al # Wait for not busy
 testb $0x2,%al
 jnz seta20.2

 movb $0xdf,%al # 0xDF -> port 0x60
```

```
outb %al,$0x60
```

## Segmentation

Now it's time to switch to 32-bit "protected mode". Up until now, the processor has been converting virtual addresses to physical ones using those segment registers which we cleared, so the mapping has been an identity map. But let's talk about how x86 converts 32-bit virtual addresses to physical ones; this is important for the rest of the boot loader code as well as the OS, so you're gonna have to bear with me for this maelstrom of x86-specific details.

The x86 architecture does the conversion in two steps: first segmentation, then paging. A virtual address starts off life as a *logical address*. Segmentation converts that to a *linear address*, and paging converts that to a physical one.

A logical address consists of a 20-bit *segment selector* and a 12-bit offset, with the segment bits before the offset bits, like `segment:offset`. The CPU's segmentation hardware uses those segment bits to pick one of those four segment registers we cleared earlier, which acts as an index into a *Global Descriptor Table* or GDT. Each entry of this GDT tells you where that segment is found in memory using a base physical address and a virtual address for the maximum or limit.

The GDT entry also has some permission bits for that segment; the segmentation hardware will check whether each address can be written to and whether the process generating the virtual address has the right permissions to access it. These checks compare the GDT entry's *Descriptor Privilege Levels*, also known as *ring levels*, against the *Current Privilege Level*. x86 has four privilege levels (0-3), so if you've ever heard of the kernel operating in ring 0 or user code in ring 3, this is where it comes from.

Okay, so the GDT entry will give us the first 20 bits of the new linear address; the offset bits stay the same. After that, the linear address is ready to be converted to a physical address by the paging hardware. We'll go over this second half of the story in the virtual memory section. For now, the point is this: xv6 is mostly gonna say no thank you to segmentation and stick to paging alone for memory virtualization.

So we're gonna set up our GDT to map all segments the exact same way: with a base of zero and the maximum possible limit (with 32 bits, that works out to a grand total of 4 GB, wow so much RAM, I can't imagine ever needing more). We have to stick this GDT somewhere in our code so we can point the CPU to it, so we'll put it at the end and throw a `gdtdesc` label on it. Now we can tell the CPU to load it up with a special x86 instruction for that.

```
lgdt gdtdesc
```

## Protected Mode

Good news, everyone! We're finally ready to turn on protected mode, which we do by setting the zero bit of the `%cr0` control register. Note that the `l` at the end of the instructions here means we're now using long words, i.e. 32 bits; `CR0_PE` is defined in the [mmu.h](#) header file as `0x1`.

```
movl %cr0, %eax # Copy %cr0 into %eax
orl $CR0_PE, %eax # Set bit 0
movl %ax, %cr0 # Copy it back
```

Oh wait, I lied. Enabling protection mode like we just did doesn't change how the processor translates addresses. We have to load a new value into a segment register to make the CPU read the GDT and change its internal segmentation settings. We can do that by using a long jump instruction, which lets us specify a code segment selector. We're just gonna jump to the very next line anyway, but in doing so we'll force the CPU to start using the GDT, which describes a 32-bit code segment, so now we're finally in 32-bit mode! Here, `SEG_KCODE` is a constant defined in [mmu.h](#) as segment 1, for `%CS`; we bitshift it left by 3.

```
ljmp $(SEG_KCODE<<3), $start32
```

First we signal the compiler to start generating 32-bit code. Then we initialize the data, extra, and stack segment registers to point to the `SEG_KDATA` entry of the GDT; that constant is defined in [mmu.h](#) as the segment for the kernel data and stack. We're not required to set up `%fs` and `%gs`, so we'll just zero them.

```
.code 32 # Tell assembler to generate 32-bit code now
start32:
 movw $(SEG_KDATA<<3), %ax # Our data segment selector
 movw %ax, %ds # Data segment
 movw %ax, %es # Extra segment
 movw %ax, %ss # Stack segment
 movw $0, %ax # Zero the segments not ready for use
 movw %ax, %fs
 movw %ax, %gs
```

## The Kernel Stack

Okay, last step in the assembly code now: we have to set up a stack in an unused part of memory. In x86, the stack grows downwards, so the "top" of the stack-- that is, the most-recently-added byte--is actually at the bottom of the stack in physical memory. It's annoying, but we're gonna have to keep track of that. The `%ebp` register points to the base of the stack (i.e., the first byte we pushed onto the stack), and the `%esp` register holds the address of the top of the stack (most-recently-pushed byte).

But where should we put the stack? The memory from `0xA_0000` to `0x10_0000` is littered with memory regions that I/O devices are gonna be checking, so that's out. The boot loader starts at `0x7C00` and takes up 512 bytes, so that means it ends at `0x7E00`. So xv6 is gonna start the stack at `0x7C00` and have it grow down from there, toward `0x0000` and away from the boot loader. Remember how back in the beginning, we started off the assembly code with a `start` label? That means that `start` is conveniently located at `0x7C00`.

```
movl $start, %esp
```

And we're done with assembly! Time to move on to C code for the rest of the boot loader. We'll take over with a C function called `bootmain()`, which should never return. The linker will take care of connecting the call here to its definition in [bootmain.c](#).

```
call bootmain
```

## Handling Errors

Wait, what? There's more assembly code after this? Why?

Well, if something goes wrong in `bootmain()`, then the function will return, so we have to handle that here. Since we usually run OSes we're developing in an emulator like Bochs or QEMU, we'll trigger a breakpoint and loop. Bochs listens on port 0x8A00, so we can transfer control back to it there; this wouldn't do anything on real hardware.

```

movw $0x8a00, %ax # 0x8a00 -> port 0x8a00
movw %ax, %dx
outw %ax, %dx
movw $0x8ae0, %ax # 0x8ae0 -> port 0x8a00
outw %ax, %dx
spin:
 jmp spin # loop forever

```

## The Global Descriptor Table

Oh, and remember when we promised the hardware that we were gonna give it a GDT? We even told it to load it from address `gdtdesc`, remember? Well, we have to deliver on that promise now by defining the GDT here.

x86 expects that the GDT will be aligned on a 32-bit boundary, so we tell the assembler to do that. Then we use the macros `SEG_NULLASM` and `SEG_ASM` defined in [asm.h](#) to create three segments: a null segment, a segment for executable code, and another for writeable data. The null segment has all zeroes; the first argument to `SEG_ASM` has the permission bits, the second is the physical base address, and the third is the maximum virtual address. As we said before, xv6 relies mostly on paging, so we set the segments to go from 0 to 4 GB so they identity-map all the memory.

```

.p2align 2 # force 4-byte alignment
gdt:
 SEG_NULLASM # null segment
 SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff) # code segment
 SEG_ASM(STA_W, 0x0, 0xffffffff) # data segment

gdtdesc:
 .word (gdtdesc - gdt - 1) # sizeof(gdt) - 1
 .long gdt # address of gdt

```

## bootmain.c

Okay, the rest of the boot loader is in C now! Most of the code here is just to interact with the disk in order to read the kernel from disk and load it into memory. Let's start off by looking at `waitdisk()`.

### waitdisk

```

void waitdisk(void)
{
 while ((inb(0x1F7) & 0xC0) != 0x40)
 ;
}

```

HEAD. DESK. Why all the magic numbers? At least we're lucky that the name makes it obvious what this function does; this won't always be true in xv6. Okay, so this function does only one thing: it loops until the disk is ready. Disk specs are boring as all hell, so feel free to skip to the next section if you don't care about the particulars (I don't blame you).

The usual way to talk to the disk is with Direct Memory Access (DMA), in which devices are hooked up directly to RAM for easy communication. But we haven't initialized the disk at all or set up any drivers for it; that's the OS's responsibility, not the boot loader's. Even if we could ask the disk to give us some data through memory-mapped I/O, we disabled all interrupts, so we wouldn't know when it's ready. So instead, we have to go back to assembly code (ugh, I know) to access the disk directly.

Storage disks have all kinds of standardized specifications, among them IDE (Integrated Drive Electronics) and ATA (Advanced Technology Attachment). The ATA specs include a Programmed I/O mode where data can be transferred between the disk and CPU through I/O ports. This is usually a huge waste of resources because every byte has to be transferred through a port and the CPU is busy the entire time, but right now beggars can't be choosers.

Each disk controller chip has two buses (primary and secondary) for use with ATA PIO mode; the primary bus sends data on port 0x1F0 and has control registers on ports 0x1F1 through 0x1F7. In particular, port 0x1F7 is the status port, which will have some flags to let us know what it's up to. The sixth bit (or 0x40 in hex) is the RDY bit, which is set when it's ready to receive more commands. The seventh bit (i.e., 0x80) is the BSY bit, which if set says the disk is busy.

Since interrupts are disabled, we'll have to manually poll the status port in an infinite loop until the BSY bit is not set but the RDY bit is: `inb( )` is a C wrapper (defined in [x86.h](#)) for the x86 assembly instruction `inb`, which reads from a port. We don't care about any of the other status flags, so we'll get rid of them by bitwise-ANDing the result with  $0xC0 = 0x40 + 0x80$ . If the result of that is 0x40, then only the RDY bit is set and we're good to go.

Phew. That was a lot for just one line of code.

## readsect

```
void readsect(void *dst, uint offset)
{
 // Issue command
 waitdisk();
 outb(0x1F2, 1);
 outb(0x1F3, offset);
 outb(0x1F4, offset >> 8);
 outb(0x1F5, offset >> 16);
 outb(0x1F6, (offset >> 24) | 0xE0);
 outb(0x1F7, 0x20);

 // Read data
 waitdisk();
 insl(0x1F0, dst, SECTSIZE/4);
}
```

If you skipped the last section: this function reads a sector (which in the current-year-according-to-xv6 of 1995 is 512 bytes) from disk. Good to see you again, on to the next section for you!

If you powered through the pain and read about ATA PIO mode above, some of the magic numbers here might be familiar. First we call `waitdisk( )` to wait for the RDY bit, then we send some stuff over ports 0x1F2 through 0x1F7, which we know are the command registers for the primary ATA bus.

Note that `uint` is just a type alias for C's `unsigned int`, defined in the header file [types.h](#). The `offset` argument is in bytes, and determines which sector we're gonna read; sector 0 has to hold the boot loader so the BIOS can find it, and in xv6 the kernel will start on disk at sector 1.

`outb()` is another C wrapper for an x86 instruction from [x86.h](#); this one's the opposite of `inb()` because it sends data out to a port. The disk controller register at port 0x1F2 determines how many sectors we're gonna read. Ports 0x1F3 through 0x1F6 are where the sector's address goes. If you *really* must know (why?) they're the sector number register, the cylinder low and high registers, and the drive/head register, in order. Port 0x1F7 was the status port above, but it also doubles as the command register; we send it command 0x20, aka READ SECTORS.

Then we wait for the RDY bit again before reading from the bus's data register at port 0x1F0, into the address pointed to by `dst`. Once again, `insl()` is a C wrapper for the x86 instruction `insl`, which reads from a port into a string. The `l` at the end means it reads one long-word (32 bits) at a time.

## readseg

```
void readseg(uchar *pa, uint count, uint offset)
{
 uchar *epa = pa + count;

 // Round down to sector boundary
 pa -= offset % SECTSIZE;

 // Translate from bytes to sectors; kernel starts at sector 1
 offset = (offset / SECTSIZE) + 1;

 // If this is too slow, we could read lots of sectors at a time. We'd write
 // more to memory than asked, but it doesn't matter -- we load in increasing
 // order.
 for (; pa < epa; pa += SECTSIZE, offset++) {
 readsect(pa, offset);
 }
}
```

Okay, finally, we're done with assembly and disk specs. We're gonna read `count` bytes starting from `offset` into physical address `pa`. Note that `uchar` is another type alias for `unsigned char` from [types.h](#); this means that `pa` is a pointer (which is 32 bits in x86) to some data where each piece is 1 byte.

`epa` will point to the end of the part we want to read. Now, `count` might not be sector-aligned, so we fix that. Declaring `pa` as a `uchar *` lets us do this pointer arithmetic easily because we know that adding 1 to `pa` makes it point at the next byte; if it were a `void *` like in `readsect()`, pointer arithmetic would be undefined. (Actually, GCC lets you do it anyway, but GCC lets you get away with a lot of crazy stuff, so let's not go there.)

Now that we've got everything set up, we just call `readsect()` in a for loop to read one sector at a time, and that's it!

Some people have asked about the structure of some of the for loops in xv6, because they don't always use obvious index variables like `int i`. There are plenty of reasons to hate C, but I think the way it structures for loops is by far one of its most powerful features:

```
for (initialization; test condition; update statements) {
 code
}
```

When evaluating the for loop, C first executes anything in the initialization. Then it checks whether the test condition is true; if so, it executes the code inside the loop. Then it carries out the update statements before checking the test condition again and running the code if it's still true.

In the for loop above, the initialization is just an empty statement; all the variables we want to use have already been set up, so we don't need it and C will just move on to the next step. The test condition is simple enough. But the update statement actually increments both `pa` and `offset` at once before going through the loop again.

Okay great, so now we can read from the disk into memory, so we're all set up to load the kernel and start running it!

## ELF Files

Before we move on to the star of the show, `bootmain()`, we need to talk about how a computer can actually recognize a file as executable. When you compile some code, the result gets spit out in a format that your machine can recognize, load into memory, and run; it's usually the linker's job to do this. Most Unix and Unix-like systems use the standardized Executable and Linkable Format, or ELF, for this purpose.

ELF divides the executable file into sections: `text` (the code's instructions), `data` (initialized global variables), `bss` (statically-allocated variables that have been declared but not initialized), `stab` and `stabstr` (debugging symbols and similar info), `rodata` (read-only data, usually stuff like string literals).

An ELF file starts with a header which has a magic number: 0x7F followed by the letters "ELF" represented as ASCII bytes; an OS can use this to recognize an ELF file. The header also tells you the file's type: it could be an executable, or a library to be linked with executables, or something else. There's a whole bunch of other info in the header, like the architecture it's made to run on, version, etc., but we're gonna ignore most of that.

The most important parts of the header are the part where it tells us where in the file the processor should start executing instructions and the part that describes the number of entries, on-disk offset, and size of the program header table.

The program header table is an array that has one entry for each of the file sections above that's found in this program. It describes the offset in the file where each section can be found along with the physical and virtual address at which that section should be loaded into memory and the size of the section, both in the file and in memory; these might differ if, e.g. the program contains some uninitialized variables which don't need to be stored in the file but do need to have space in memory.

The kernel (along with all the user-space programs) will be compiled and linked as ELF files, so `bootmain()` will have to parse the ELF header to find the program header table, then parse that to load each section into memory at the right address. xv6 uses a `struct elfhdr` and a `struct proghdr`, both defined in [elf.h](#), for this purpose.

Okay, back to the boot loader to finish up now!

## bootmain

This is the C function that gets called by the first part of the boot loader written in assembly. Its job will be to load the kernel into memory and start running it at its entry point, a program called `entry()`.

Next up, we're gonna use `readseg()` to load the kernel's ELF header into memory at physical address `0x1_0000`; the number isn't too important because the header won't be used for long; we just need some scratch space in some unused memory away from the boot loader's code, the stack, and the device memory-mapped I/O region. We'll read 4096 bytes first at offset 0; `readseg()` turns that offset into sector 1. Remember that we have to convert `elf` into a `uchar *` so that the pointer arithmetic in `readseg()` works out the way we want it to.

```
void bootmain(void)
{
 struct elfhdr *elf = (struct elfhdr *) 0x10000;
 readseg((uchar *) elf, 4096, 0);
 // ...
}
```

While we're at it, let's go ahead and make sure that what we're loading really is an ELF file and not some random other garbage because any of a million things went wrong during the compilation process, or we got some rootkit that totally corrupted the kernel or something. It's not really the most robust of checks, but *eh*. If something went wrong we'll just return, since we know that the code in `bootasm.S` is ready to handle that with some Bochs breakpoints.

```
void bootmain(void)
{
 // ...
 if (elf->magic != ELF_MAGIC) {
 return;
 }
 // ...
}
```

Now we have to look at the program header table to know where to find each of the kernel's segments. The `elf->phoff` field tells us the program header table's offset from the start of the ELF header, so we'll set `ph` to point to that and `eph` to point to the end of the table.

```
void bootmain(void)
{
 // ...
 struct proghdr *ph = (struct proghdr *) ((uchar *) elf + elf->phoff);
 struct proghdr *eph = ph + elf->phnum;
 // ...
}
```

Each entry in the program header table tells us where to find a segment, so we'll iterate over the entries, reading each one from disk and loading it up. In this for loop, note that `ph` is a `struct proghdr *`, so incrementing it with `ph++` increments it by the size of a `struct proghdr` and not by one byte; this makes it automatically point at the next entry in the table.

```
void bootmain(void)
```

```

{
 // ...
 for (; ph < eph; ph++) {
 uchar *pa = (uchar *) ph->paddr; // address to load section into
 readseg(pa, ph->filesz, ph->off); // read section from disk

 // Check if the segment's size in memory is larger than the file image
 if (ph->memsz > ph->filesz) {
 stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
 }
 }
 // ...
}

```

That if statement at the end checks if the section's size in memory should be larger than its size in the file, in which case it calls `stosb()`, which is yet another C wrapper from [x86.h](#) for the x86 instruction `rep stosb`, which block loads bytes into a string. It's used here to zero the rest of the memory space for that section. Okay, but why would we want to do that? Well, if the reason it's larger is because it has some uninitialized static variables, then we want to make sure those start off holding zero (as the C standard requires) and not whatever garbage value may have been there before.

Last part of the bootloader: let's call the kernel's entry point, `entry()`, and get it running! But remember how the boot loader is compiled and linked separately from the kernel? Yeah, that means we can't just call `entry()` as a function, because then the linker would go "Huh? What entry function? I don't have any `entry` function here in your symbol table. REJECTED." And then it would throw a huge error.

Luckily, the ELF header tells us where to find the entry point in memory, so we could get a pointer to that address. That means... function pointers! If you've never used function pointers in C before, then this won't be the last time you'll see them in xv6, so check it out.

A C function is just a bunch of code to be executed in order, right? That means it shows up in the ELF file's `text` section, which will end up in memory. When you call a regular old C function, the compiler just adds some extra assembly instructions to throw a return address on the stack and update the registers `%ebp` and `%esp` to point to the new function's stack on top of the old one. If the function getting called has any arguments or local variables, they'll get pushed onto the stack too. Then the instruction register `%eip` gets updated to point to the new function section, and that's it. After the compiler is done, the linker will replace the function's name with its memory address in the `text` section, and voila, a function call.

The point of all this is that in C we can use pointers to functions; they just point to the beginning of that function's instructions in memory, where the `%eip` register would end up pointing if the function gets called. So in this case, even though we're not linking with the kernel, we can still call into the entry point by getting its address from the ELF header, creating a function pointer to that address, then calling the function pointer. The compiler will still add all the usual stack magic, but instead of the linker determining where `%eip` should point, we'll do that ourselves.

The first line below declares `entry` as a pointer to a function with argument type `void` and return type `void`. Then we set `entry` to the address from the ELF header, then we call it.

Again, this shouldn't return, but if it does then it's the last part of this function, so this function will return back into the assembly boot loader code.

```
void bootmain(void)
{
 // ...
 void (*entry)(void);
 entry = (void(*)(void)) (elf->entry);
 entry();
}
```

That's it! Starting from `entry()`, we're officially out of the boot loader and into the kernel.

## Summary

To summarize, the assembly part of the boot loader (1) disabled interrupts, (2) set up the GDT and segment registers so the segmentation hardware is happy and we can ignore it later, (3) set up a stack, and (4) got us from 16-bit real mode to 32-bit protected mode.

Then the C part of the boot loader just loaded the kernel from disk and called into its entry point.

ELF headers will continue to haunt us in the kernel's linker script and when we load user programs from disk in `exec()`, and function pointers will make another appearance when we get around to handling interrupts. The good news: the boot loader is one of the most opaque parts of the xv6 code, full of boring hardware specs and backwards-compatibility requirements, so if you made it this far, it does get better!

(But it also gets worse... looking at you, [mp.c](#) and [kbd.c...](#))

## **The point of this document:**

I'm actually not so sure. I think the point is to explain why xv6 does stuff, in broad strokes, as a complement to the explanations of the actual code of xv6.

## **Important-ish note:**

Often, I'll refer to things that haven't been explained yet. Just keep in mind that not everything is explained at first, and that's okay.

These explanations are not full. They may not even be accurate. I'm just jotting these down during class, hoping it more or less gives a picture of what's going on.

Anyways, here we go.

---

## ***Kernel***

When I say "kernel", I mean the operating system. This is the mother of all programs that hogs the computer to itself and manages all the other programs and how they access the computer resources (such as memory, registers, time (that is, who runs when), etc.).

The kernel also keeps control over the "mode" bit (in the %CS register), which marks whether current commands are running in user-mode (and therefore can't do all kinds of privileged stuff) or kernel-mode (and therefore can do whatever it wants).

## ***Boot***

The processor (that is, the actual computer) does not know what operating system is installed, what it looks like, or how large it is.

So how can the processor load xv6 into the memory?

The processor instruction pointer %eip register points - by default - to a certain memory place in the ROM, which contains a simple program that:

1. Copies the very first block (512 bytes. AKA the boot block) from the disk to the memory
2. Sets the instruction pointer %eip to point to the beginning of the newly-copies data

So what?

Every operating system needs to make sure that its first 512 bytes are a small program (it has to be small; it's only 512 bytes!) that loads the rest of the operating system to the memory and sets the PC to whatever place it needs to be in. If 512 bytes aren't enough for this, the program can actually call a slightly larger program that can load and set up more.

In short:

1. %eip points to hard-coded ROM code
2. ROM code loads beginning of OS code
3. Beginning of OS code loads the rest of the code
4. Now hear the word of the Lord!

## **Processes**

A process is the running of a program, including the program's state and data. The state includes such things as:

- Memory the program occupied
- Memory contents
- Register values
- Files
- Kernel structures

This is managed by the kernel. The kernel has a simple data structure for each process, organized in some list. The kernel juggles between the processes, using a context switch, which:

1. Saves state of old process to memory
2. Loads state of new process from memory

Context switch can be preemptive (i.e. the kernel decides "next guy's turn!", using the scheduler (there'll be a whole lot of talk about this scheduler guy later on)) or non-preemptive (i.e. the hardware itself decides). In non-preemptive, it is asked of the programmers to make calls to the kernel once in a while, in order to let the kernel choose to let the next guy run.

Xv6 is preemptive.

Processes are created by the kernel, after another process asks it to. Therefore, the kernel needs to run the first process itself, in order to create someone who will ask for new processes to be created.

## **fork( )**

Every process has a process ID (or pid for short).

A process can call the kernel to do **fork( )**, which creates a new process, which is entirely identical to the parent process (registers, memory and everything). The only differences are:

- The pid
- The value returned from fork:
- In the new process - 0
- In the parent process - the new pid
- In case of failure - some negative error code

## **exec( )**

**Fork( )** creates a new process, and leaves the parent running. **Exec( )**, on the other hand, replaces the process's program with a new program. It's still the same process, but with new code (and variables, stack, etc.). Registers, pid, etc. remain the same.

It is common practice to have the child of **fork** call **exec** after making sure it is the child. So why not just make a single function that does both fork and exec together? The exec system call replaces the entire memory of the parent process except for the open files. This allows a parent process to decide what the stdin, stdout and stderr for the child process. This trick is used in the /init process in xv6 and the sh to pipe outputs to other processes.

## **Process termination**

Trigger warning: sad stuff ahead. And also zombies.

A process will be terminated if (and only if) one of the following happens:

1. The process invokes `exit()`
2. Some other process invokes `kill()` with its pid
3. The process generates some exception

Note that `kill` does not actually terminate the process. What it does is leave a mark of "You need to kill yourself" on the process, and it'll be the process itself that commits suicide (after it starts running again, when the scheduler loads it).

Note also that not any process can `kill` any other process. The kernel makes sure of that.

Once `killed`, the process's resources (memory, etc.) are not released yet, until its parent (that is, the process which called for its creation) allows this to happen. A process that `killed` itself but whose parent did not acknowledge this is called a zombie.

In order for a parent to "acknowledge" its child's termination, it needs to call `wait()`. When it calls `wait()`, it will not continue until one of its children exits, and then it will continue. If there are a few children, the parent will need to call `wait` once for each child process.

`Wait` returns the pid of the exited process.

What happens if a parent process `exits` before its children? Its children become orphans, and the The First Process (whose pid is 1) will make them His children.

## **System calls**

Many commands will only run if the "mode" bit is set to kernel-mode. However, all processes run on user-mode only; xv6 makes sure of that.

In order to run a privileged command, a process must ask the kernel (which runs in kernel-mode, of course) to carry out the command. This "asking" is called a system call.

Here's an example of system call on Linux with an x86 processor:

```
movl flags(%esp), %ecx
lea name(%esp), %ebx
movl $5,%eax ; loads the value 5 into eax register, which is the command
"open" in Linux
int $128 ; invokes system call. Always 128!
; eax should now contain success code
```

The kernel has a vector with a bunch of pointers to functions (Yay, pointers!).

## **Addresses**

This one's a biggie. Hold on to your seatbelts, kids.

Programs refer to memory addresses. They do this when they refer to a variable (that's right; once code's compiled, all mentions of the variable are turned into the variable's address). They do this in

every **if** or loop. When the good old %eip register holds the address of the next instruction, it's referring to a memory address.

There are two issues that arise from this:

- The compiled code does not know where in the memory the program is going to be, and therefore these addresses must be relative to the program's actual address. (This is a problem with loops and ifs, not with the %eip.)
- We'll want to make sure no evil program tries to access the memory of another program.

So, each process has to have its "own" addresses, which it thinks are the actual addresses. It has *nothing* to do with the actual RAM, just with the *addresses* that the process knows and refers to. (**Process**, not **program**; this includes the kernel.)

Behold! A sketch of what a process's addresses looks like in xv6:

Address	Who uses this
[0xFFFF FFFF]	Kernel
[0xFFFF FFFE]	Kernel
...	Kernel
[0x8000 0000]	<b>Kernel</b>
[0x7FF FFFF]	<b>Process</b>
...	Process
[0x0000 0000]	Process

In order to pull off this trick, we use a hardware piece called the Address Translation Unit, which actually isn't officially called that.

Its real name is the MMU (Memory Management Unit), for some reason.

The MMU is actually comprised of two units:

1. Segmentation Unit
2. Paging Unit.

The MMU sits on the address bus between the CPU and the memory, and decides which actual addresses the CPU accesses when it reads and writes. Each of the smaller units (segmentation and paging) can be turned on or off. Note that Paging can only be turned on if Segmentation is on. (We actually won't really use the Segmentation Unit in xv6. LATER EDIT: This is a lie. A LIE! We actually use it for all kinds of unwholesome tricks.)

Addresses coming from CPU are called **virtual/logical addresses**. Once through the Segmentation Unit, they're called **linear addresses**. Once through the Paging Unit, they're **physical addresses**.

In short: CPU [virtual] -> Segmentation [linear] -> Paging [physical] -> RAM

Here's a bunch of 16-bit registers that are used by the Segmentation Unit:

- %cs - **C**ode. This guy actually messes with our %eip register (that's the guy who points to the next command!).
- %ds - **D**ata. By default, messes with all registers except %eip, %esp and %ebp. (In assembly, we can override the choice of messing register.)
- %ss - **S**tack. By default, messes with %esp and %ebp registers.
- %es

- %fs
- %gs

The address %eip points to after going through %cs is written as CS:EIP.

In the Segmentation Unit there is a register called %GDTR. The kernel uses this to store the address of a table called GDT (Global Descriptor Table). Every row is 8 bytes, and row #0 is not used. It can have up to 8192 rows, but no more.

The first two parts of each row are **Base** and **Limit**.

When the Segmentation Unit receives an address, the CPU gives it an *index*. This index is written in one of the segmentation registers (%cs, %ds, ... %gs), thusly:

- Bits 0-1: permissions
- Bit 2: "use GDT or LDT?" (Let's pretend this doesn't exist, because it does not interest us at all.)
- Bits 4-15: The index

The Segmentation Unit receives a logical address and uses the index to look at the GDT. Then:

- If the logical address is greater than the **Limit**, crash!
- Else, the Segmentation adds the **Base** to the logical address, and out comes a linear address.

Note: In the past, the Segmentation Unit was used in order to make sure different processes had their own memory. For example, they could do this by making sure that each time the kernel changes a process, its segmentation registers would all point to an index that "belongs" to that process (and each row in the GDT would contain appropriate data). Another way this could be done would be by maintaining a separate GDT for each process. Or maintaining a single row in the GDT and updating it each time we switch a process. There is no single correct way.

Note that all the addresses used by each process must be consecutive (along the physical memory).

In xv6, we don't want any of this.

Therefore, we will make sure that the GDT **Limit** is set to max possible, and the **Base** is set to 0.

In order to allow consecutive virtual addresses to be mapped to different areas in the physical memory, we use **paging**.

In the Paging Unit, there is a register named %CR3 (Control Register 3), which points to the **physical** address of the Page Table (kinda like GDT). A row in the Page Table has a whole bunch of data, such as page address, "is valid", and some other friendly guys.

When the Paging Unit receives a linear address, it acts thusly:

- The left-side bits are used as an *index* (AKA "page number") for the Page Table. (There are 20 of these.)
- In the matching row, if the "is valid" bit = 0, crash!
- (There are also "permission" bits, but let's ignore them for now.)
- Those "page number" bits from the linear address are replaced by the actual page in our row (which is already part of the actual real live physical address)

- The page is "glued" to the right-side bits of the linear address (you know, those that aren't the page number. There are 12 of these.)
- Voila! We have in our hands a physical address.

Note that each page can be in a totally different place in the physical memory. The pages can be scattered (in page-sized chunks) all along the RAM.

Also note: Hardware demands that the 12 right-most bits of %CR3 be 0. (If not, the hardware'll zero 'em itself.)

### **Uh oh:**

Each row in the Page Table takes up 4KB (that's 12 bits).

The Page Table has 1024 rows.

$4\text{KB} * 1024 = 4\text{MB}$ . That's 4 whole consecutive MBs. That's quite a large area in the memory, which kind of defeats the whole purpose of the Page Table. Well, not the *whole* purpose, but definitely some of it.

**The solution:** The Page Table gets its very own Page Table!

- First (small) page table contains - in each row - the (physical) address of another (small) page table.
- Each of the (small) 2nd-level page tables (which are scattered) contain actual page addresses.
- So: instead of 20 bits for single index, we have 10 bits for 1st-level table index and 10 bits for 2nd-level table index.

Thus, a virtual address can be broken down like so:

[*i0*][*i1*][*offset*]

- *i0* is the index of the First Table
- *i1* is the index of a second table
- *offset* is the offset

### **Addresses - double mapping**

Every single physical address is mapped by the kernel to virtual address by adding KERNBASE to it.

When a process is given a virtual address, this new virtual address is *in addition* to the kernel mapping.

So when we want to get the physical address of a virtual address:

1. If the user-code is asking, it needs to access the page tables.
2. If the kernel is asking, it can simply subtract KERNBASE from the address.

Likewise, the kernel can determine *its* virtual address of *any* physical address by adding KERNBASE.

KERNBASE = 0x80000000.

## 1217 main

In the beginning, we know that from [0x0000 0000] till [0x0009 FFFF] there are 640KB RAM.

From [0x000A 0000] till [0x000F FFFF] is the "I/O area" (384KB), which contains ROM and stuff we must not use (it belongs to the hardware).

From [0x0010 0000] (1MB) till [0xFF00 0000] (4GB - 1MB in total) there is, once again, usable RAM.

After that comes "I/O area 2".

(Why the 640KB, the break, and then the rest? Because in the olden days they thought no one would ever use more than 640KB.)

Address	Who uses this
[Who cares]	I/O
...	I/O
[0xFF00 0001]	I/O
[0xFF00 0000]	Usable RAM
...	Usable RAM
[0x0010 0000]	Usable RAM
[0x000F FFFF]	I/O
...	I/O
[0x000A 0000]	I/O
[0x0009 FFFF]	Usable RAM
...	Usable RAM
[0x0000 0000]	Usable RAM

Remember Mr. Boot? He loads xv6 to [0x0010 0000].

By the time xv6 loads (and starts running `main`), we have the following setup:

1. %esp register is pointing to a stack with 4KB, for xv6's use. (That's not a lot.)
2. Segmentation Unit is ready, with a (temporaray) GDT that does no damage (first row inaccessible, and another two with 0 Base and max Limit).
3. Paging Unit is ready, with a temporary page table. The paging table works thusly:
  - Addresses from [0x800- ----] till [0x803- ----] are mapped to [0x000- ----] through [0x003- ----] repectively. (That is, the left-most bit is simply zeroed.)
  - ALL the the above addresses have 1000000000b as their 10 left-most bits, so our 1st-level page table has row 512 (that's 1000000000b) as "valid", and all the rest marked as "not valid".
  - Row 512 points to a single 2nd-level table.
  - The 2nd-level table uses all 1024 of its rows (that's exactly the next 10 bits of our virtual addresses), so they're all marked as valid.
  - Each row in 2nd-level table contains a value which - coincidentally - happens to be the exact same number as the row index.

Let's look at some sample virual address just to see how it all works:

[0x8024 56AB] -> [1000 0000 0010 0100 0101 0110 1010 1011] -> [1000 0000 00 (that's row 512) 10 0100 0101 (that's row 581) 0110 1010 1011 (and that's the offset)] -> row 581 in the 2nd-level table will turn [1000 0000 0010 0100 0101...] to [0000 0000 0010 0100 0101...], which is exactly according to the mapping rule we mentioned a few lines ago.

## **Available pages (free!)**

Processes need pages to be mapped to. Obviously, we want to make sure that we keep track of which page are available.

The free pages are managed by a **linked list**. This list is held by a global variable named `kmem`. Each item is a `run` struct, which contains only a pointer to the next guy.

`kmem` also has a lock, which makes sure (we'll learn later how) that different processes don't work on the memory in the same time and mess it up. (An alternative would be to give each processor its own pages. However, that could cause some processors to run out of memory while another has spare. (There are ways to work around it.))

`main` calls `kinit1` and `kinit2`, which call `freerange`, which calls `kfree`.

In `kfree`, we perform the following 3 sanity checks:

- We're not in the middle of some page
- We're not trying to free part of the kernel
- We're not pushing beyond the edge of the physical memory

## **Building a page table**

Let's examine what needs to be done (not necessarily in xv6) in order to make our very own paging table.

Our table should be able to "translate" virtual address `va` to physical address `vp` according to some rule.

Note that we'll be using `v2p`, which is a hard-coded mother-function that translates **virtual** addresses to **physical** by subtracting `KERNBASE` (which equals `0x8000 0000`) from the virtual addresses.

**Step 1:** Call `kalloc` and get a page for our First Table. (Save (virtual!) address of new table in `pgdir` variable)

**Step 2:** Call `memset` to clear entire page (thus marking all rows as invalid).

**Step 3:** Do the following for **every single `va`** we want to map:

- **Step 3.1:** Create and clear subtable, and save address in `pgtab` (similar to what we did in steps 1 and 2). (SEE NOTE AFTER THIS LIST)
- **Step 3.2:** Figure out `i0` (index in `pgdir`) (using `va` & `v2p` function), write `pgtab` there, mark as valid.
- **Step 3.3:** Figure out `i1` (index in `pgtab`) (using `va` & `v2p` function), write `pa` there, mark as valid.

**Step 4:** Set %CR3 to point at `pgdir`, using v2p.

THE NOTE MENTIONED EARLIER: After we already have some subtables, we only need to create new subtables if the requested subtable does not exist yet.

How do we know whether it exists already? Simply by looking at the current i0 and seeing whether it's already marked as valid.

ANOTHER NOTE: What if two different `va->vp` rules clash? xv6 will crash (deliberately).

**For more details about how the kernel builds its mapping tables, please refer to [xv6 Code Explained.md] (`kvmalloc`).**

## ***Moar GDT stuff!***

Remember how we said we'd make sure GDT has `base=0` (so it won't alter addresses) and `limit=max` (so it won't interfere)?

Well, it turns out the hardware still uses the GDT to check various user-mode/kernel-mode stuff.

We need four rows in the GDT:

1. Kernel-mode, can only execute and read
2. Kernel-mode, can write
3. User-mode, can only execute and write
4. User-mode, can write.

There is a macro named `SEG`, which helps us build all these annoying bits.

Okay, that's enough of this.

Don't pretend you understand, because you don't and it doesn't matter.

## ***Per-CPU variables***

We've got an array of CPU data, `cpus`.

We can access specific CPU stuff via `cpus[SOME_CPU_IDENTIFIER]`.

In order to get current CPU identifier, we call `getcpu()`, which is slow.

When we do this, we *MUST* stop interrupts from happening, to make sure we stay within the same CPU.

**Problem:** Calling `getcpu()` is slow, and stopping (and then resuming) interrupts can be extremely slow.

**Solution:** Instead of using `getcpu()`, we can use a special register in each CPU!

**Problem:** Registers can be overridden by users.

**Solution:** Special register that only kernel can use!

**Problem:** *There are no such registers.*

**Solution:** Cheating, using the GDT!

So we have a list of GDTs, one per CPU.

When we initialize a CPU (just once, yeah?), we set up its very own GDT.

In this GDT, we add a *fifth* row (remember, we have four rows for kernel/user x read/write).

In this new row, we set the base to point at the place in the memory where the CPU data sits, and set the limit to 8 (because we have two important variables of CPU data, and each one take 4).

We do this in `seginit()`.

```
struct cpus *c = &cpus[getcpu()]; // costly, but used just once!

// Set first four rows...
c->gdt[SEG_KCODE]=SEG(STA_X|STA_R, 0, 0xffffffff, 0);
c->gdt[SEG_KDATA]=SEG(STA_S, 0, 0xffffffff, 0);
c->gdt[SEG_UCODE]=SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
c->gdt[SEG_UDATA]=SEG(STA_S, 0, 0xffffffff, DPL_USER);

// Set our special row
c->gdt[SEG_KCPU]=SEG(STA_W, &c->proc, 8, 0);

// Load the GDT
lgdt(c->gdt, sizeof(c->gdt));

// Set %gs register to point at our fifth row in the GDT.
loadgs(SEG_KCPU << 3); // shift 3 left so we get value and not permission bits.

// Declare out two variables,
// telling code generator to use gs register
// whenever we access proc or cpu
extern struct proc *proc asm("%gs:0");
extern struct proc *cpu asm("%gs:4");

proc = NULL;
cpu = c;
```

So how does this affect our accessing per-CPU data?

Normally, if we do `proc = 3;`, then it would be complied to `movl $3, proc`.

However, now that we did that weird `extern struct proc *proc asm("%gs:0");` part, it is compiled to `movl $3, %gs:0`.

**Therefore**, whenever in the code we call `proc` or `cpu`, we will automagically be referring to the address held in `GS:0` or `GS:4`, which happens to be the `proc` or `cpu` belonging to the specific CPU in which the code is executed.

Note that variable `cpu` points to the variables `cpu` and `proc`, so calling `proc` is the same as (yet faster than) calling `cpu->proc`.

Note also that this means that calling `cpu->cpu->cpu->cpu` is the same as calling `cpu`.

**Important note:** GS can be changed by code in user-mode.

So, every time there is an interrupt, we reload our value to `%gs` using `loadgs(SEG_KCPU << 3);`. (Don't worry, we back up all user-mode's registers beforehand.)

## Processes

Every process has a struct `proc` that holds a bunch of its data:

- `SZ` - memory size
- `pgdir` - its very own page table

- **kstack** - small *kernel* stack (4KB), for data we don't want the process to touch and break (such as return address from system call)
- **state**
- **pid** - process ID
- **parent** - parent process
- **tf** - pointer to place where register values are saved during interrupt
- **context** = another interrupt thing
- **chan** - channel, marks what event process is waiting for (while sleeping). If none, then set to 0.
- **killed** - 0 if not killed yet
- **ofile** - vector of open files
- **cwd** - current working directory (like when doing `cd . . .` in command)
- **name**

A process can move through its processes thusly:

Embryo -> Runnable -> Running

Running -> Zombie

Running -> Sleeping (waiting for whatever event is marked in **chan**)

Running -> Runnable

The processes are held in a struct named **ptable**, who has a vector of processes named **proc**.

## ***Preparing the First Process***

When running The First Process, we do the following in **userinit**:

- **allocproc**: allocate **proc** structure
- Allocate process kernel stack
- Set up data on kernel stack (in the "trapframe" area) *as if the process already existed and went into kernel mode*. As a result, the data will be loaded to **proc** in due time
- **setupkvm**: create page table (without knowing yet where to map user addresses (only kernel addresses))
- **inituvm**:
  - Allocate free page
  - Copy code to page
  - Update page table to map user memory
- Fix the data on the kernel stack for some reason.

Trapframe contains all register data.

When we return from an interrupt (or, in our case, start the process for the first time), we do the following:

1. Pop the first 4 values to their appropriate registers
2. Pop **eip** field to **%eip** register (now **%eip** register is pointing to **forkret** function)
3. Goto **forkret** function, which (somehow) takes us to the next guy in the stack...

4. ...Trapret which pops another whole bunch of values from the stack to the registers
5. And so on. We end up with all registers holding good values, and %eip pointing to 0.

## **The First Process**

All it does it execute the program "init" (with the arguments {"/init", 0} (the 0 is to mark the end of the list)).

This is how the code *would* have looked like if it were written in c:

```
char *argv[] = {"/init", 0};
main() {
 exec(argv[0], argv);
 for (;;) exit();
}
```

...But it's not written in c.

It's written in assembly.

Here's the arguments code:

```
these are the arguments:
```

```
init:
 .string "/init\0"
 .p2align 2
argv:
 .long init
 .long 0
```

...And here's the *code* code:

```
.globl start
start:
 pushl $argv ; push second argument
 pushl $init ; push first argument
 pushl $0
 movl $SYS_exec, %eax ; load the canons (perpare to call "exec"
system-call)
 int $T_SYSCALL ; FIRE!

once end up back here, we need to quit.
exit:
 movl $SYS_exit, %eax
 int $T_SYSCALL
 jmp exit
```

So we have this code, but we need to load it during `inituvm`.

This is done in the script in the `initcode.S` file, which compiles that code to binary and shoves it all into the data section, with the exact same labels that are used in `inituvm`.

## **Actually running the First Process**

This is done in `mpmain`, which calls the scheduler, which loops over process tables and runs processes.

Since at this point there's only one process at this point, it's the only one that'll run.

## ***Accessing process's kernel stack during interrupt***

When we're in user-mode and there's an interrupt, the kernel needs its own stack on the process to push real data onto it (before leaving the process).

We need this kernel-stack to be accessable only by the kernel.

The hardware supports this, with a magnificent Rube Goldberg machine:

- %tr register can only be accessed in kernel-mode
- %tr contains SEG\_TSS, which points to special place in GDT
- special place in GDT contains address and size of some weird struct
- weird TSS struct contains field esp0, which actually (finally) points to top of process's kernel stack (and ss0 with stack size)

So during `switchuvm` we set up this whole thing, so that in due time the kernel (and only the kernel) can access its stack on the proc.

## ***Locks - the problem***

Because we have multiple CPUs, we need to make sure two CPUs don't both grab the same data (such as the same `proc` struct during `allocproc`, or the same page during `kalloc`).

This could (theoretically) even happen with a single CPU, because of interrupts.

## ***Locks - a possible solution (just for interrupts)***

We can block interrupts (!) from happening by setting the IF bit in `%eflags` register to 0.

This can be controlled (in kernel-mode, yes?) using the assembly commands:

- `cli` (clears bit)
- `sti` (sets bit to 1)

(Also, the entire `eflags` register can be pushed into `eax` using some other assembly command.)

So each time we want to do something unsafe, we can do:

```
cli();
something_dangerous();
sti();
```

...But this is still dangerous, because of cases such as the following:

```
function b() {
 cli();
 // ...
 sti();
}

function a() {
 cli();
 b(); // at the end of this we set sti()!
 // ... Uh oh. We're open to interrupts!
 sti();
}
```

## ***Locks - an actual solution (just for interrupts)***

**Solution:** Keep track of how many times we called `cli()`, so that `sti()` only releases if down to 0.

We can do this by wrapping all `cli()`s with `pushcli()` and `sti()`s with `popcli()`.

In `popcli()`, if our count reaches 0, we don't automatically call `sti()`, but revert to the **previous setting**.

```
function b() {
 pushcli();
 // ...
 popcli();
}

function a() {
 pushcli();
 b(); // at the end of this we DO NOT set sti()
 // ... the counter is still 1
 popcli(); // NOW the counter became 0, so now we call cli().
}
```

(Note that each CPU runs `pushcli()` and `popcli()` separately for each CPU.)

## ***Locks - an actual solution (for multiple CPUs)***

There is a magical assembly word, `lock`.

Usage example:

```
lock ; xchgl mem, reg
```

If `lock` is added to different commands run by different CPUs, then the `locked` commands will execute *serially*.

Using this, the following c call:

```
old = xchg(&lock, b);
```

...Does the following:

1. Sets `old = lock`
2. Sets `lock = b` (but only if not locked by someone else!)

How on earth does this help us?

I'll tell you how!

We have a struct called `spinlock`, which holds all kinds of locking data.

In the function `acquire`, we do the following:

```
acquire(struct spinlock *lk) {
 pushcli(); // disable interrupts in this CPU, to avoid deadlocks
 while(xchg(&lk->locked, 1) != 0);
}
```

`xchg` will always return the *existing* value of `&lk->locked`, so if it's already locked we'll loop until it's released.

In `release`, we do the following:

```
release (struct spinlock*lk) {
 xchg(&lk->locked, 0);
 popcli();
}
```

### ***How locks are managed during scheduler***

When `scheduler` searches for `RUNNABLE` procs, it **acquires** a lock on the process table. This is in order to make sure that `schedulers` on *other* CPUs don't grab the same process we just did.

The lock is held even as we switch to the process; it's the process's responsibility to release the lock (in order to allow interrupts).

Likewise, the process must re-lock the process table before returning to `scheduler`.

Each lock is re-released by the next process that runs, and then re-re-locked.

The final release is done by `scheduler` once there are no runnable processes left.

### ***Sleep***

When a process needs a resource in order to continue (such as disk, input or whatever), it must call `sleep` where it marks itself (`proc->chan`) as waiting for the wanted event, sets its `state` to `SLEEPING`, and returns to `scheduler`.

When the Event Manager (whom we never heard of yet) sees that the event occurs, it marks the proc as `RUNNABLE` once again.

Note that the process needs to see that its resource is still available (and hasn't been made unavailable again by the time it woke up) before continuing. If the resource is unavailable again, it should go back to sleep (if it knows what's good for it).

`sleep` also demands a locked `spinlock` for a weird reason we don't know yet.

It replaces the locking from *that* lock to the process table (unless they are the same lock, of course). Once done, it releases the process table and re-locks the supplied `spinlock`.

We'll get back to you folks on this one once we understand why on earth this is needed.

### ***Interrupts - CPU***

There are two basic types of interrupts: internal (thrown by CPU) and external (like keyboard strokes).

We can decide whether to listen to external interrupts using kernel-mode functions `sti()` and `cli()` which sets or clears the `IF` bit on the `%eflags` register.

We cannot ignore internal interrupts.

Interrupts can only occur *between* commands.

During an interrupt, the CPU needs to know what kind of interrupt it is. These range between 0-255, when 0-31 are reserved by Intel for all kinds of hardware stuff.

In order to handle these interrupts, the CPU needs a table with pointers to functions that handle 'em (with the index being the interrupt number).

This table is called IDT, and the register that points to it is %IDTR.

Each row has a whole descriptor of 64 bits. These include:

- **offset** - the actual address of the function (this guy is loaded to %eip)
- **selector** - loaded to %cs
- **type** - 1 for trap gate, 0 for interrupt gate (if interrupt, we disable other interrupts)

Before an interrupt call, the CPU must do a bunch of stuff to rescue our registers before they are overridden by the IDT guy.

Reminder: the kernel stack's address is saved in a **tss** structure, which is saved in the GDT, in the index held by %tr register.

Therefore: the kernel needs to set this before an interrupt occurs.

**SO:** The CPU uses the kernel stack to store our registers.

(And after the interrupt handling is over, the CPU needs to pop these guys again. It does this with assembly **iret** command.)

If the interrupt occurs during **kernel-mode**, the CPU performs the following:

- push %eflags,%cs and %eip to the **kernel** stack (so we'll have them again after the interrupt).

If the interrupt occurs during **user-mode**, the CPU performs the following:

- Store %ss in %ss3
- Store \$esp in %esp3
- Store tss.ss0 in %ss
- Store tss.esp0 in %esp
- Push %ss3, %esp3, %eflags, %cs and %eip to **kernel** stack

**Question:** Since we push a different number of registers for kernel-mode and user-mode, how do we know how many to pop back out?

**Answer:** In both cases, we need to first pop %eip, %cs and %eflags.

Having done that, we can look at the two right-most bits of %cs to see whether we were in user- or kernel-mode before the interrupt! Hurray!

## **Interrups - xv6**

In xv6, all interrupts are handled by a *single* C function **trap**.

However:

1. This function needs to end with **iret** (that's the guy who handles all the register poppin' at the end of the call), while C functions end with **ret**!
2. Different interrupts need to send different data to the function (such as interrupt number, error number (which not every interrupt needs)).

In order to accomplish this, we wrap the call to **trap** in assembly code that does the following:

1. Push registers to kernel-stack (except those which were already pushed by CPU (see previous section))
2. Push address of kernel-stack to kernel-stack (so the address is sent as a parameter to `trap`; I'll elaborate more on this in a moment)
3. Call `trap`
4. Pop the registers
5. Do `iret`

The function `trap` receives a pointer to a `trapframe` struct, which is actually the kernel-stack. The fields in `trapframe` are the actual register values, so `trap` can use them in order to determine all kinds of stuff (such as "were we in user-mode when the interrupt occurred?"). Additionally, we have the interrupt number and error-code (where applicable).

The assembly code that wraps the call to `trap` is mostly the same for each interrupt, with just the slightest difference:

Which interrupt number to push, and do we push an error-number (and if so, which).

So, the shared code is under a label `alltraps` (which, as it says on the label, is for all traps). Each interrupt does its own little thing and then jumps to `alltraps`.

The code parts for the different interrupts are labelled `vector0`, `vector1`, ... `vector255`, and are generated by a script (in `verctors.pl` file).

After generating the vectors, the script creates an array (called - surprise, surprise - "vectors"). Later, during `main`, we call `tvinit` which loops over the vectors array and writes the data in the IDT table.

## ***syscall***

When user-code calls a system-call, the system-call number is stored in `%eax`.

So in `syscall`, we can access the system-call number via `proc->tf->eax`.

After running the system-call, we put the return value in `proc->tf->eax`; this will cause `alltraps` to pop it back to `%eax`, and then the user-code will know whether the system-call succeeded.

The system-call numbers are saved as constants in `syscall.h` file.

Then we have a vector that maps each constant to its appropriate function. This vector is used in `syscall` function in order to execute the system-call.

## ***sleep system-call***

There is a global variable called `ticks`.

Whenever there is a timer interrupt occurs, `ticks` increments by 1.

We use the *address* of `ticks` as the **event number** for the timer.

So...

When a process calls `sleep`, it sets its channel to equal `&ticks`, and stores the current value of `ticks`.

Whenever a timer event occurs, if the current value of `ticks` minus the value of `ticks` that the process stored *reaches* the number of ticks the process asked to sleep for, it becomes RUNNABLE.

Here's a sample of what a call to sleep looks like in assembly (for 10 ticks) :

```
pushl $10
subl $4, %esp ; empty word. Some weird convention
movl $SYS_sleep, %eax
int $64
```

### ***Getting arguments in system-calls***

In the example of `sleep`, the argument (how many ticks to wait) is stored on the **user**-stack.

This is accessed via `proc->tf->esp + 4` (the +4 is to skip the empty word, which is on the stack because of some weird convention). We do this in `argint` function.

Because this the data is placed on the stack by *user code* (which is pointed at by `%esp`), we need to check that the user didn't set `%esp` to point at some invalid address! We do this in `fetchint` function.

### ***fork() #2***

When we `fork` a proc, we need to copy a bunch of its data.

We want to map *new* physical addresses, but copy the old data.

We want to create a *new* kernel stack.

We want to copy context.

We want the state to be RUNNABLE (and not RUNNING like parent).

We want a new `pid`.

A lot of this stuff is already handled by `allocproc`.

The memory stuff is handled by `copyuvm`.

### ***exec() #2***

`exec` replaces the current process with new one.

Because this may fail, we don't want to destroy the current memory data until we know we succeeded.

In order to keep the current memory until the end, we need to create a *new* memory mapping, and only switch to it (and destroy the old mapping) at the very end.

The four stages of `exec`:

1. Load file
2. Allocate user stack (and a guard page to protect data from being overwritten by stack)
3. Pass `argv[]` arguments
4. Fix trapframe and switch context

In order for `exec` to run the file it's told to run, the file needs to be in ELF format.

## **ELF**

Executable and Linkable Format.

In xv6 we only use *static* ELF, but in real life there are also *dynamic* ones but we don't care about that here.

ELF file have:

1. ELF header at the very start
2. `proghdr` vector
3. Program sections

The program sections include the actual code, as well as external code linked by the linker. There can be many program sections, scattered along different parts of the file (but not at the beginning).

Each program section needs a *program header*, which says where it is in the file and where it is in the RAM.

All the *program headers* are held in the `proghdr` vector.

The location of `proghdr` is held in the ELF header.

ELFHDR (that's our ELF header) has a bunch of fields, but we'll only look here at a few:

- `uint magic` - must equal `ELF_MAGIC`. Just an assurance all's good.
- `ushort machine` - the type of processor. xv6 actually doesn't pay attention to this, so I don't know why we bothered mentioning this one.
- `uint entry` - virtual address of `main`
- `uint phoff` - PH offset, location of `proghdr`
- `uint phnum` - PH number, length of `proghdr`
- `uint flags` - uh... flags.

Since the ELF is created by user-code, the kernel doesn't trust it.

The kernel treats the data with caution, and fails the `exec` if there are any errors.

`proghdr` has the following important fields in each vector:

- `uint off` - where section starts in file
- `uint filesz` - where sections ends in file (so last byte is in `off+filesz-1`)
- `uint vaddr` - virtual address to which section is loaded
- `uint memsz` - size section takes in memory. May be larger than `filesz`, because section may include lots of zeros (but not the other way around).

The kernel needs to loop and copy the memory, allocating new memory if either 1) we need more memory or 2) `vaddr` is beyond what is already allocated. This is done using `allocuvm`.

### **`exec` - guard page**

After `exec` loads da codes, it allocates another page for the user-stack.

**Problem:** Stack goes to *lower* addresses as it fills up; eventually, it'll overrun the code-n-data!

**Solution:** Add guard page, with user-mode bit cleared. That way, user-code will get an error if StackOverflow.

## **exec - pushing arguments to new process**

New process expects the following to be on the user-stack:

variable	what it holds
argv	pointer to vector of pointers to string arguments (with 0 in last place)
argc	number of arguments
	return address

...And this is how **exec** fills the user-stack:

### **user-stack**

argument  
...  
argument  
0  
pointer to argument  
...  
pointer to argument  
argv (pointing to this ↑ guy)  
argc  
-1

Note that the **pointers to arguments** don't really need to be on the stack; we just put 'em there because we have to put them *somewhere* so why not.

Note also that we don't know in advance how many arguments there are and how long they'll be. Therefore, our code must first place the arguments stuff, and only afterwards copy **argc**, **argv** and return address.

*Therefore*, we save **argc** & **argv** & return address to a temporary variable **ustack**, which we copy to actual stack at the end of the argument pushing.

## **I/O**

xv6 supplies the following system-call functions for files:

- **open(char \*name, int flag)** - opens file and returns file descriptor (i.e. index in file array)
- **close(int fd)** - closes file
- **pipe(int pipefd[2])** - kinda like a file, for sending messages between processes
- **dup(int fd)** - duplicates opening to file (so if we move the cursor/offset of one (such as by reading), it'll affect the other)
- **read**
- **write**
- **stat**

Reading/writing can be much different if we're dealing with keyboard, file, etc., but xv6 strives to make them "act" the same.

We've got the following read/writables:

- pipe
- inode
  - file
  - directory
  - device

Each has its own internal functions for reading, writing, etc., but - as mentioned above - xv6 wraps 'em and hides 'em from user code.

xv6 has an abstract struct `file`, which it uses for all the above read/writables.

xv6 has functions that deal with this `file`, but they accept a *pointer* instead of a file descriptor.

Why? Because:

- xv6 needs a pointer to an actual struct, because it needs to handle data.
- User code isn't trusted to handle pointers; it gives us a file descriptor, and the *kernel* accesses the actual pointer.

So how do we maintain this translation between file descriptors and pointers?

With an array "ofile", that's how.

These `file` guys have basic data that all read/writables contains, such as "type".

**Inodes** implement their "inheritance" by having their "type" be "FD\_INODE", and containing a pointer (in "inode" field) to a `struct inode` that has its very own data (such as "type", which could be "T\_FILE" for a file or "T\_DEV" for device).

OK, this explanation is pretty lame :(

You should really just see the "I/O" presentation on Carmi's site, which he changes every semester. Currently, the presentation is at <http://www2.mta.ac.il/~carmi/Teaching/2016-7A-OS/Slides/Lec%20io.pdf> (slides 13-23).

## **Pipe stuff**

A `pipe` is pointed to by a *reader file struct* and a *writer file struct*.

(Both of which exist *once*, and may be pointed to by many different procs.)

The `pipe` has `readopen` and `writeopen` fields, which contain "1" as long as the *reader* and *writer* guys are still holding him open.

When both are "0", the `pipe` is closed.

## **Moar file stuff**

The kernel has an array `f_table`, which holds all the open files (and a lock to make sure two procs don't fight over a slot in the array).

Files are added to `f_table` in the function `filealloc`.

## **Transactions**

When writing to blocks on disk (or deleting), we really don't want the machine to crash or restart, or else there'll be corrupted data.

In order to alleviate our fears, we can wrap our *writing* in a *transaction* (by calling `begin_trans()` before and `commit_trans()` after).

A transaction ensures that the write will either be *complete* or won't be at all.

A transaction is limited to a certain size, so during `writefile` we need to divide the data to manageable chunks.

Therefore, if there's a crash in the middle of the write operation, some blocks might be written and the rest not. However, each block will be fully-written or not written at all (but no hybrid mish-mash).

## **inode Stuff**

xv6 considers a directory to be a file like any other.

Every `struct dirent` has an inode number, and a name. (If the directory is not in use, the number is 0.)

The inodes are managed by the inode layer, which we will talk about later.

`struct inode` has the following amongst its fields:

- `uint dev` - device number
- `uint inum` - inode number
- `int ref` - reference count
- `int flags` - flags, such as 0x1 for `I_BUSY` (that is, *locked*) and 0x2 for `I_VALID`
- `short nlink` - the number of names ('links') the inode has on the disk (kinda like a shortcut in Windows)
- `uint size` - the size of the inode on the disk

In order to open an inode, we have `namei()`.

In order to search for an inode, we have `dirlookup()`.

Note that `dirlookup` can be supplied with an optional pointer `poff` that gets the offset of the found inode.

Why?

In case of renaming or deleting, we'll need to update the row in our parent inode. (In case of name change, change the num; in case of deletion, change `inum` to 0.)

So... when we open an inode, we need to supply the inode to search in.

Conveniently, we actually *don't* supply an inode pointer, because we have THE CURRENT WORKING DIRECTORY which is "supplied" automatically.

...Actually, that's not exactly true.

It *is* true if we're looking for some "a/b/c" path, but **not** for "/a/b/c".

In the latter case, we use the **root inode** instead of the current working directory.

As you may or may not have guessed, if we have a path with a few parts (such as "a/b/c/d"), we need to loop over the parts and for each part find the matching inode.

In order to split the path into parts, we use `skip elem`.

## ***inode Stuff - The Inode Layer***

The inode layer supplies a bunch of funcs:

- `iget` - open
- `iput` - close
- `idup` - increments ref count
- `ilock` - must lock inode whenever we want to access any field
- `iunlock` - unlocks inode
- `ialloc` - allocates inode both on disk and in memory
- `iupdate` - update disk with whatever changes we made to inode
- `readi`
- `writei`

Our inode structs are saved in `icache`.

They are saved there the first time they are opened (but not before; they are loaded on-demand).

The inodes are added to `icache` in `iget`.

When added, we add them *without the actual data from the disk*.

Why?

Because often we call `iget` just to see if the inode exists, and actually reading from the disk is expensive (so we do it just if we need it).

## ***The Disk - Reading and Writing***

The disk is divided to blocks.

Every block on our disk is of 512 bytes; no more, no less.

Reading and writing to and from a block is done *just* in the beginning of a block.

So how do we write (or read) only a few bytes?

We have a buffer layer that takes a whole block to memory, writes (or reads) our few bytes *in memory*, then - if we're writing - re-writes the entire block to the disk.

The buffer layer contains the functions:

- `readsb` - reads super-block. Never mind.
- `bread(dev, sector)` reads an entire block. (function returns a `struct buf` which contains a field `data` with the actual 512 bytes of data)
- `log_write`
- `brlse`
- `balloc`
- `IBLOCK` - we'll explain in a moment:

Remember `struct inode`?

Well, there's also `struct dinode` which represents an inode *on the disk*.

It's similar to the inode, but without the meta-data (such as ref, inum, etc.).

**IBLOCK** is a macro that gives us the block number of an inode.

In order to tell *where* within the block is our inode, we can calculate (inum % IPB). IPB is the number of inodes in a block.

## **ilock**

As mentioned before, before accessing inode data we need to **ilock** it.

We don't want to use our regular locks, because they:

- disable interrupts
- when trying to acquire, "spin" over lock till lock is open

This wastes a lot of time.

**ilock** is a *soft* lock, that doesn't do this stuff.

We don't disable interrupts, and instead of spinning, we do **sleep** (so we're not hogging CPU while waiting for acquiring).

## **Reading from the disk**

Every inode on the disk has a vector of **addrs**.

Every entry points at a single block on the disk.

The first 12 entries are *direct* pointers.

The 13th points to a block that serves as vector of pointers.

That is, the last pointer is a pointer to pointers.

It turns out there's actually a reason for having the inode data be partially direct and partially indirect:

- We have the direct blocks because reading from the indirect blocks cost an extra read for each block.
- We have the indirect blocks because the direct blocks are a waste of space for small files.

## **The buffer layer**

The buffer layer supplies the following functions:

- **bread**
- **bwrite**
- **balloc**
- **bfree**

**struct buf** has the following fields:

- **flags** - such as BUSY or DIRTY
- **dev**
- **sector** - device and block num
- **struct buf \* prev** - for linked list
- **struct buf \* next** - for linked list
- **struct buf \* qnext**

- uchar data[512]

All our buffers are kept in a cache, **bcache**.

This cache contains a spinlock (obviously), and a linked list of buffers (held in field **head**).

The linked list of buffers is maintained so that the most recently used is in the head, and the least in the tail.

Buffs are never removed from the cache!

If they're not used, they're marked as *not* busy and *not* dirty.

If we need a new buff (such as calling **bget** for a block that hasn't been read yet), we find a **struct buff** at the end of the list that is neither BUSY nor DIRTY, and mark it BUSY.

When we call **brelse** and release a buffer, it is moved to the head of the list (and its BUSY flag is turned off).

**bcache** is initialized in **binit()**.

### ***The superblock and the bitmap***

There is a block on the disk called the *superblock*.

This block is located right after the boot block, and contains meta data regarding the other blocks.

There is another area on the disk called the *bitmap*.

For every block on the disk, there is a bit that marks whether the block is free or not.

### ***The driver layer***

Ah, the driver layer. The guy who actually does all the dirty work.

In order to actually access the disk (as well as other peripherals, such as keyboard), we have controllers.

Our guy is the IDE.

The IDE has a bunch of *ports*, which are actually numbers (but "port" sounds a lot more computer-y than "number").

The processor can send/receive values to/from these ports using IN/OUT commands.

The IDE decides what to do for different values it is given to different ports.

If we want the IDE to read (or write) from the disk, we need to tell it:

1. Which disk to read from (turns out there are two)
2. Where on the disk to read (28 bits to tell us which block)
3. How many blocks to read

Each of these needs to be sent to different OUT ports, before we send the command "read" to the port that accepts the command:

- 0x1f2 - number of blocks
- 0x1f3, 0x1f4, 0x1f5 and 0x1f6 - block number (28 bits split to 8 bits per port (0x1f6 gets only 3 bits and device number))
- 0x1f7 - the command

The IDE can only handle one single command at a time; so before we do anything, we need to check its IN 0x1f7 port (the STATUS port).

The left-most bit tells us if the IDE is busy (so shouldn't use).

The second bit tells us if the IDE is ready (so can use).

The IDE has its very own RAM attached to it, which it uses to read/write.

(NOTE: We don't know how much is in this RAM. It depends on the controller we chose to put in our PC.)

So how do we access this RAM from our code?

### ***Accessing this RAM from our code***

In order to do this, we use OUT port 0x1f0.

For example:

```
long *a (long*)b->data;
for (int i = 0; i < 128; i++)
 outl(0x1f0, a[i]);
```

Although it looks like every `long` is being written to the same place, it isn't.

Same goes for reading:

```
long *a (long*)b->data;
for (int i = 0; i < 128; i++)
 a[i] = inl(0x1f0);
```

Another useful parameter to send is 0 to port 0x3f6.

This tells the IDE to raise an interrupt when it finishes the command.

(Otherwise, the kernel can't tell when the read/write from/to the disk is complete.)

### ***Functions provided by the driver layer***

- `idewait` - loops over IDE port 0x1f7 until status is READY
- `idestart` - starts requesting to read/write (whether to read or write depends on input)
- `iderw` - the *real* read/write function. Handles a queue of blocks to write, because it can receive many requests during the long time it takes the IDE to actually write stuff to the disk.

In order to handle the queue, the buffer layer has a variable `idequeue`.

This is a linked list of bufs, which are actually read/written by `ideint`, which is run whenever there is an interrupt from IDE due to IDE finishing its previous command.

- `ideint` - handles "I'm finished!" interrupt from IDU and managed `idequeue`.

## **main** (*kernel entry point*)

- **kinit1** (*frees pages*)
  - **freerange** (*frees pages*)
    - **kfree** (*frees single page*)
- **kvmalloc** (*builds new page table*)
  - **setupkvm** (*sets up kernel page table*)
    - **mappages** (*adds translations to page table*)
      - **walkpgdir** (*allocates and maps page*)
        - **kalloc** (*allocates page*)
        - **memset** (*clean new page*)
  - **seginit** (*sets up segmentation table*)
  - **tvinit** (*initializes interrupt table*)
  - **kinit2** (*frees pages*)
    - **freerange** (*frees pages*)
      - **kfree** (*frees single page*)
  - **userinit** (*initialize the First Process*)
    - **allocproc** (*allocates new proc*)
    - **setupkvm** (*sets up kernel page table*)
      - **mappages** (*adds translations to page table*)
        - **walkpgdir** (*allocates and maps page*)
          - **kalloc** (*allocates page*)
          - **memset** (*clean new page*)
      - **inituvm** (*allocates and maps single page, and copies First Process code to it*)
  - **mpmain**
    - **idtinit** (*sets %IDTR to point at existing interrupt table*)
    - **scheduler** (*runs runnable processes*)
      - **acquire** (*locks process table*)
        - **pushcli** (*makes us ignore interrupts*)
      - **switchuvm** (*prepares proc's kernel stack and makes TSS point to it*)
      - **swtch** (*saves current context on proc, and switch to new proc*)
      - **switchkvm** (*switches back to kernel page table*)
      - **release** (*unlocks process table*)
        - **popcli** (*makes us stop ignoring interrupts*)

---

**fork** (*creates child process*)

- **allocproc** (*allocates new proc*)
- **copyuvm** (*copies memory*)
  - **setupkvm** (*sets up kernel page table*)
    - **mappages** \*(*adds translations to page table*)
      - **walkpgdir** (*allocates and maps page*)
        - **kalloc** (*allocates page*)
        - **memset** (*clean new page*)
      - **walkpgdir** (*validate that page mapping exists, without allocating or cleaning*)
    - **kalloc** (*allocate new page for user-code*)
    - **memmove** (*copy page data*)
    - **mappages** (*add user-code page*)
      - **walkpgdir** (*maps page, without allocating or cleaning*)
    - **freevm** (*free page table in case of error*)
      - **deallocuvm** ()
        - **walkpgdir** (*get entry of internal page*)
        - **kfree** (*free actual page*)
      - **kfree** (*free inner table*)
      - **kfree** (*free outer table*)
  - **kfree** (*if error, free kernel-stack*)
  - **filedup**
  - **idup**
  - **safestrcpy**

---

Hello, world.

In this document, we'll attempt to explain the actual code of our beloved xv6.  
Not all of it, but some of the interesting parts.

God have mercy on us.

---

## **1217 main(void)**

The entry point of the kernel.

Sets up kernel stuff and starts running the first process.

**1219**: set up first bunch of pages, for kernel to work with (minimal, because old hardware has little memory)

**1220**: set up all kernel pages

**1223**: set up segment tables and per-CPU data

**1238**: set up the rest of the pages for general use (because until now we had just minimal, because other CPUs might not handle high addresses)

**1239**: set up the First Process

**1241**: run the scheduler (and the First Process)

---

## **2764 struct run**

Represents a memory page.

A **run** points to the next available page/run (which is actually the *previous* page, because the first available is the last in the memory).

---

## **2768 struct kmem**

Points to the head of a list of free (that is, available) pages of memory.

---

## **2780 kinit1(void \*vstart, void \*vend)**

Frees a bunch of pages.

Also does some locking thing (I'll elaborate once we actually learn this stuff).

Used only when kernel starts up.

Called by [main](#).

---

## **2788 kinit2(void \*vstart, void \*vend)**

Frees a bunch of pages.

Also does some locking thing (I'll elaborate once we actually learn this stuff).

Used only when kernel starts up.

Called by [main](#).

---

## **2801 freerange(void \*vstart, void \*vend)**

Frees a bunch of pages.

**2804:** use PGROUNDUP because `kinit1` in called to start where the kernel finished (which is not likely to end *exactly* at a page end).

Called by:

- [kinit1](#)
  - [kinit2](#)
- 

## **2815 kfree(char \*v)**

Frees the (single!) page that `v` points at.

**2819-2820:** address validity checks

**2823:** fill page with 1s, to help in case of bugs

**2827-2829:** insert our page into the beginning of `kmem` (a linked list with all available pages)

Called by:

- `deallocuvm`
  - `freevm`
  - `fork`
  - `wait`
  - [freerange](#)
  - `pipealloc`
  - `pipeclose`
- 

## **2838 kalloc(void)**

Removes a page from `kmem`, and returns its (virtual!) address.

**2844-2846:** remove first free page from `kmem`

**2849:** return address

Called by:

- `startothers`
  - [walkpgdir](#)
  - [setupkvm](#)
  - [inituvm](#)
  - `allocuvm`
  - `copyuvm`
  - [allocproc](#)
  - `pipealloc`
- 

## 1757 `kvmalloc(void)`

Builds new page table and makes %CR3 point to it.

**1759:** make table and get its address

**1760:** make %CR3 point to returned address

Called by [main](#).

---

## 1728 `kmap[]`

Contains data of how kernel pages should look.

Used by `setupkvm` for mapping.

**Column 0:** Virtual addresses

**Column 1:** Physical addresses start

**Column 2:** Physical addresses end

**Column 3:** Pages permissions

**Note:** The `data` variable (used in lines 1730-1731 is where the kernel's data start (*data* is plural, by the way)).

We do not know during compilation where this will be.

---

## 1737 `setupkvm(void)`

Sets up kernel virtual pages.

**Returns** page table address if successful, 0 if not.

**1742:** create outer `pgdir` page table

**1744:** clear `pgdir`

**1745-1746:** make sure we didn't map illegal address

**1747-1750:** loop over kmap and map pages using mappages

Called by:

- [kvmalloc](#)
  - [copyuvm](#)
  - [userinit](#)
  - [exec](#)
- 

**1679 mappages(pde\_t \*pgdir, void \*va, uint size, uint pa, int perm)**

Creates translations from *va* (virtual address) to *pa* (physical address) in existing page table *pgdir*.

**Returns** 0 if successful, -1 if not.

**1684:** get starting address

**1685:** get ending address (which is starting address if *size*=1)

**1686:** for each page...

**1687:** get i1 row address (using *walkpgdir*)

**1689:** make sure i1 row not used already

**1691:** write *pa* in i1 and mark as valid, with required permissions

Called by:

- [setupkvm](#)
  - [inituvm](#)
  - [allocuvm](#)
  - [copyuvm](#)
- 

**1654 walkpgdir(pde\_t \*pgdir, const void \*va, int alloc)**

Looks at virtual address *va*,

finds where it should be mapped to according to page table *pgdir*,  
and returns the **virtual** address of the the *index* i1.

**Returns** address if successful, 0 if not.

If there is no mapping, then:

if *alloc*=1, mapping is created (and address is returned);  
if *alloc*=0, return 0

Some constants and macros used here:

PDX - zeroes offset bits

PTX - uh... some more bit manipulations

PTE\_P - "valid" bit

PTE\_W - "can write" bit

PTE\_U - "available in usermode" bit

**1659:** get i0 index address

**1660:** check if i0 is valid

**1661:** put address of subtable in pgtab

**1663:** (i0 not valid) create new subtable, pgtab

**1666:** (i0 not valid) clear pgtab rows

**1670:** (i0 not valid) point i0 to pgtab and mark **i0** as valid, writable, usermodeable.

**1672:** return address of appropriate row in pgtab

Called by:

- [mappages](#)
  - [loaduvm](#)
  - [deallocuvm](#)
  - [clearpteu](#)
  - [copyuvm](#)
  - [uva2ka](#)
- 

## **1616 seginit(void)**

Sets segmentation table so that it doesn't get in the way, for each CPU. Adds extra row to each segmentation table in order to guard CPU-specific data, makes %gs register point to it, and makes proc and cpu actually point to %gs.

**1624-1628:** set up regular rows in segmentation table

**1631-1634:** set up special row and %gs register

**1637-1638:** set up initial proc and cpu data

Called by [main](#).

---

## **2252 userinit(void)**

Creates and sets up The First Process.

**2255:** data and size of First Process code. Filled by script during compilation.

**2257:** allocate proc structure and set up data on kernel stack

**2258:** save proc in `initproc`, so we'll always remember who the First Process is

**2259-2260:** create page table with kernel addresses mapped

**2261:** allocate free page, copy process code to page, map user addresses in page table

**2262-2275:** fix data on kernel-stack, *as if* it were stored there because of an interrupt

- **2264:** make sure we'll be in usermode when process starts
- **2270:** make sure process will start in address 0

Called by [main](#).

---

## **2205 allocproc(void)**

Allocates `proc` structure and sets up data on kernel stack.

**Returns** proc if succeeds, 0 if not.

**2211-2213:** find first unused `proc` structure

**2217-2219:** set to EMBRYO state and give pid

**2223-2226:** allocate and assign kernel stack for process

**2227:** set stack pointer to bottom of stack (stack bottom is highest address in stack)

**2230-2231:** make room on stack for `trapframe`

**2235-2239:** make room on stack for `context`

**2240-2241:** set `context`, setting `context.eip` to function `forkret`

Called by:

- [userinit](#)
  - `fork`
- 

## **1803 inituvm(pde\_t \*pgdir, char \*init, uint sz)**

Allocates and maps single page (4KB), and fills it with program code.

**1807-1808:** make sure entire program code fits in single page

**1809:** allocate page

**1810:** clear page

**1811:** map pages in page table `pgdir` (using v2p, because we know what memory this is, because this is the First Process, which the kernel always creates on startup)

**1812:** copy the code

Called by [userinit](#).

---

## **2458 scheduler(void)**

Loops over all processes (in each CPU), finds a runnable process, and runs it.

Loops for ever and ever.

**2467:** lock process table, to prevent multiple CPUs from grabbing same process

**2468-2470:** find first available process

**2475:** set *per-CPU* variable **proc** to point to current running process

**2476:** set up process's kernel stack, and switch to its page table

**2477:** mark process as running

**2478:** save current registers (including where to continue on scheduler) and load process's registers, handing the stage over to the process

(NOTE: the running process is responsible to release the process table lock (to enable interrupts) and later re-lock it.)

**2479:** now that process is switching back to scheduler, switch back to kernel registers and Page Table

**2483:** set per-CPU variable **proc** back to 0

**2485:** release process table, allowing other CPUs to grab processes (just in case

Called by **mpmain**.

---

## **1773 switchuvm(struct proc \*p)**

Perpares kernel-stack of process (that is, makes %tr register indirectly point to it), and loads process's Page Table to %cr3.

**1776-1779:** set up %tr register and SEG\_TSS section in GDT end up magically (don't ask how) referring us to top of process's kernel stack

Called by:

- **growproc**
  - [scheduler](#)
  - [exec](#)
- 

## **2708 swtch(struct context \*\*old, struct context \*new)**

Saves current register context in **old**, then loads the register context from **new**.

Basically gives control to new process.

**2709:** set %eax to contain address of **old** context

**2710:** set %edx to contain **new** context

**2713-2716:** push %ebp, %ebx, %esi, %edi onto current stack (which happens to be old stack)  
**2719:** copy value of %esp to address held in %eax, which is the old stack address (see line 2709)  
**2720:** set current stack pointer (%esp) to value of %edx, which is the new stack address (see line 2710)  
**2723-2726:** pop %edi, %esi, %ebx, %ebp from new stack onto the actual stack

Called by:

- [scheduler](#)
  - [sched](#)
- 

## **2503 sched(void)**

Switches back scheduler to return from a process that had enough running.

Called by:

- exit
  - [yield](#)
  - sleep
- 

## **2522 yield(void)**

Gives up the CPU from a running process.

**2524:** re-lock the process table for scheduler

**2525:** make self as not running

**2526:** switch back to scheduler

**2527:** after scheduler re-ran process, re-elease process table to enable interrupts

Called by trap.

---

## **2553 sleep(void \*chan, struct spinlock \*lk)**

Makes process sleep until chan event occurs.

**2568:** lock process table in order to set sleeping state safely

**2569:** now that process table is locked, release lk

**2573-2574:** set up sleeping state (and alarm clock)

**2575:** return to scheduler until the event manager marks process as runnable

**2578:** clean up

**2582:** release process table

**2583:** lock lk once again

---

## **1555 pushcli(void)**

Saves state of %eflags register's IF bit (that is, the current state of "listen to interrupts?" bit), increments the "how many times did we choose to ignore interrupts" counter, and clears the "listen to interrupts" bit.

**1561-1562:** save initial state of bit in interna var (FL\_IF is the location of our bit)

Called by:

- [acquire](#)
  - [switchuvm](#)
- 

## **1566 popcli(void)**

Decrements the "how many times did we choose to ignore interrupts" counter, and if it reaches 0 then sets the "listen to interrupts" bit to whatever it was before the very first **pushcli** was ever called.

**1572-1573:** only set our bit if **interna** (initial bit value) was set

Called by:

- [release](#)
  - [switchuvm](#)
- 

## **1474 acquire(struct spinlock \*lk)**

Loops over spinlock until lock is acquired (exclusively by current CPU).

**1476:** disable interrupts (which will be enabled in **release**)

**1483-1484:** loop over lock until it has a zero (and write "1" in it)

---

## **1502 release(struct spinlock \*lk)**

Releases spinlock from being held by current CPU.

**1519:** write "0" in lock

**1521:** re-enable interrupts (which were disabled in **acquire**)

---

## **3004 alltraps**

Catches and prepares all interrupts for `trap`.

Pushes register data on stack, calls `trap` with the `stack` as a `trapframe` argument, pops register data from stack, and finally calls `iret`.

**3005-3010:** store registers and build `trapframe`

**3013-3018:** set up data and per-CPU segments (?)

**3021-3023:** call `trap`, using stack as argument

- **3023:** skip over top of frame (%esp address) without popping it into anything

**3027-3034:** pop registers and call `iret`

- **3033:** skip over data and per-CPU segments (without popping it into anything)
- 

## **3101 trap(struct trapframe \*tf)**

Handles all interrupts.

**3103-3111:** handle system call and return

- **3106:** save `tf` to `proc->tf`, so that we don't need to start passing it around during `syscall`

**3113-3143:** handle controller interrupts (keyboard, timer, etc.)

**3150-3163:** handle unexpected interrupt

- **3151:** check if there is no current process (i.e. during `scheduler`) or if we were in kernel-mode during interrupt
- **3158-3162:** print error and kill buggy process

**3168-3178:** finish up non-system calls

- **3168-3169:** if process is user-process, and killed, and is not in the middle of a system-call, exit (and don't return to `alltraps`)
- **3173-3174:** if process is running, and we had a timer-interrupt, and the process ran for long enough already, yield CPU back to `scheduler`
- **3177-3178:** after previous yield, if process was killed (and not in middle of system-call), exit

Called by [alltraps](#)

---

## **3067 tvinit(void)**

Initializes the IDT table.

Called by [main](#).

---

## **3079 idtinit(void)**

Makes %IDTR point at existing IDT table.

Called by `mpmain`

---

## **3375 syscall(void)**

Handles system-calls from user-code.

**3379:** get system-call number

**3380:** make sure number is valid

**3381:** execute system-call and store return value in process's trapframe's `eax` field (which will afterward be popped to `%eax`)

**3382-3385:** if bad system-call number, print error and store -1 (error) in trapframe's `eax` field

Called by [trap](#).

---

## **3465 sys\_sleep(void)**

System call for sleeping a certain amount of ticks.

**3470:** get number of required ticks (and validate that the user supplied a valid address as an argument)

**3472:** lock tickslock

**3473:** store current (initial) value of `ticks` in `ticks0`

**3474-3480:** while required number of ticks didn't pass, loop

- **3479:** wait for event `#&ticks` (we don't really need the lock in this case, but usually in `sleep` call we need to lock because other cases we `sleep` for disc or something else where we don't want two process's to grab the resource simultaneously)

Can be called by user code.

---

## **2614 wakeup(void \*chan)**

Locks process table, finds all sleeping processes that are waiting for `chan`, makes them runnable, and unlocks process table.

Called by a lot of different functions.

---

## **2603 wakeup1(void \*chan)**

Finds all sleeping processes that are waiting for `chan`, and makes them runnable.

Called by:

- `exit`
  - wakeup
- 

### **3295 `argint(int n, int *ip)`**

Gets the nth *integer* argument pushed onto the user-stack by user code before user asked for system-call.

---

### **3267 `fetchint(uint addr, int *ip)`**

Gets the integer argument in address `addr`, and sets it in `ip`.

Returns 0 if successful, -1 otherwise.

**3267:** Validates that neither "edge" of integer-containing address space goes beyond valid proc memory.

Called by:

- `argint`
  - `sys_exec`
- 

### **2304 `fork(void)`**

Creates new process, copying lots from its parent, and set stack as if returning from a system-call.

**2310:** allocate new proc. Proc now contains kernel-stack, context (with trapret address), trapframe and pid

**2314-2319:** copy memory

- **2314:** copy memory
- **2315-2318:** if error, free kernel stack

**2320:** copy sz

**2321:** set parent

**2322:** copy trapframe struct to new kernel-stack

**2325:** clear %eax so `fork` will return 0 for child process

**2327-2330:** do file stuff

**2333:** make new proc RUNNABLE (at this point, `scheduler` can grab child process before parent)

---

## **1953 copyuvm(pde\_t \*pgdir, uint sz)**

Creates copy of parent memory for child process.

Returns address of new page table.

**1960:** set up kernel virtual pages

**1962:** loop over all pages:

- **1963:** get address+flags of parent process's page
- **1965:** make sure page is actually present
- **1967:** get physical address of parent process's page
- **1968:** allocate new page
- **1970:** copy memory from old physical page to new physical page
- **1971:** add-n-map new page to new page table

**1974:** if no errors, return address of new page table

**1977-1978:** if there were any errors, release all memory and return 0

Called by [fork](#)

---

## **1910 freevm(pde\_t \*pgdir)**

Frees a page table and all the physical memory pages (in its user part).

**1916:** free user-mode pages

**1917-1921:** free internal page tables

**1923:** free external page table

Called by:

- [copyuvm](#)
  - [wait](#)
  - [exec](#)
- 

## **1882 deallocuvm(pde\_t \*pgdir, uint oldsz, uint newsz)**

Deallocates user pages to bring the process size from `oldsz` to `newsz`.

**1891:** loop over extra pages we want to deallocate:

- **1892:** get virtual address of internal table entry that points to current page-to-remove
- **1896:** get physical address of page
- **1899:** get virtual address of page
- **1900:** free page

- **1901**: mark internal page table entry as "pointing at no page"

Called by:

- [allocuvm](#)
  - [freevm](#)
  - growproc
- 

### **1853 allocuvm(pde\_t \*pgdir, uint oldsz, uint newsz)**

Allocate page tables and physical memory to grow process from `oldsz` to `newsz`.  
returns `newsz` if succeeded, 0 otherwise.

Called by:

- growproc
  - [exec](#)
- 

### **5910 exec(char \*path, char \*\*argv)**

Replaces current process with new one.

**5920-5949**: load da code

- **5920**: open file
- **5926**: read ELF header
- **5936-5947**: loop over sections in `proghdr`:
  - **5937**: read section from file
  - **5943**: allocate memory
  - **5945**: load code and data
- **5948**: close file

**5952-5956**: allocate user-stack and guard page

- **5952**: round up address in order to add stack and guard-page at new page
- **5953**: allocate two pages, for stack and guard page
- **5955**: remove user-mode bit from guard page
- **5956**: set stack pointer to point to stack page

**5959-5967**: push arguments to new process user-stack

- **5959**: loop over arguments:
  - **5962**: move new process stack pointer so there's room for current argument
  - **5963**: copy argument to actual new process user-stack

- **5965:** make `argv` vector entry (which is still in temporary `ustack` variable!) point to current argument that we just pushed to stack
- **5967:** put 0 in last entry of `argv` vector (which is still in temporary `ustack` variable!), as is expected by convention
- **5969:** put -1 as return address in appropriate spot in temporary `ustack` variable
- **5970:** put `argc` in appropriate spot in temporary `ustack` variable
- **5971:** put address of where `argv` will be in new user-stack (but isn't there yet) in appropriate spot in temporary `ustack` variable
- **5974:** now that all arguments are copied to new user-stack, and we know where to place return address & `argc` & `argv`, copy `ustack` variable to new process user-stack

**5978-5981:** copy new process name

- **5978-5980:** get part of the name after all the slashes

**5984-5991:** switch address space and fix trapframe

- **5984:** save old page table for freeing later
- **5985:** give proc new page table!
- **5986:** give proc new size
- **5987:** set trapframe's `eip` to new process entry point
- **5988:** set trapframe's `sp` to new process user-stack
- **5989:** make `%CR3` point to new page table (which doesn't harm our current running!)
- **5990:** free old page table
- **5991:** return to syscall, which returns to alltraps, which returns to popall, which returns to iret, which pops a bunch of values to actual registers, which include `%eip`, which makes us actually continue with the new process

**1818 `loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)`**

Loads a `sz`-sized program section from `ip` file (in `offset` offset) to address `addr` (which is already mapped in `pgdir`).

Returns 0 if successful, -1 otherwise.

**1825-1835:** loop page by page (because the pages mapped in `pgdir` are scattered (and we can't rely on the Paging Unit to handle this, because `pgdir` is not our current page table))

- **1826:** validate that page is already mapped
- **1828:** get physical address
- **1829-1832:** take care of case where what's left to read is less than page

- 1833: copy code-n-data

Called by [exec](#)

---

## **2002 uva2ka(pde\_t \*pgdir, char \*uva)**

Returns the kernel virtual address of a user virtual address.

Only works for addresses of pages (and not for middle of page).

2006: get pte entry

2011: get offset, do p2v to it, and return result

---

## **2018 copyout(pde\_t \*pgdir, uint va, void \*p, uint len)**

Copies len bytes from p address to pgdir->va address.

2029: get va offset within its page

2032: copy data

---

## **2354 exit(void)**

Exits current process.

2359-2360: make sure we're not the First Process

2363-2368: close all files opened by user-code

2370-2371: close current working-directory

2376: let parent know we're exiting (in case parent called `wait` for child to exit) (The chan the parent is waiting for is the parent's `proc` address)

2379-2385: pass abandoned children to the First Process

- 2382-2383: if child did `exit`, let the First Process know

2388: become zombie

2389: awaken the scheduler

---

## **5225 filealloc(void)**

Finds the first free slot in the global file table, and returns its *address*.

If there are none free, returns 0.

---

## **5438 fdalloc(struct file \*f)**

Finds the first free slot in the process's file table, points it to **f**, and returns the index (AKA the file descriptor).

If no room, returns -1.

---

## **5252 filedup(struct file \*f)**

Increments the reference count of **f**.

(Used as part of the file duplication process)

---

## **5451 sys\_dup(void)**

Duplicates proc's reference to file.

**5458:** actual duplication

**5460:** increment ref count

---

## **5419 argfd(int n, int \*pf, struct file \*\*pf)**

Gets the **n**th argument sent to the system call, as a file descriptor.

Returns descriptor and the struct file it points to.

(The only reason we need the file descriptor **pf** is in case we're closing the file and need to make **ofile** point to null.)

---

## **5315 fileread(struct file \*f, char \*addr, int n)**

Reads from **f** to **addr**.

**5319:** make sure can read

**5321-5322:** handle case when file is pipe

**5323-5329:** handle case when file is inode:

- **5324:** lock the inode (because we must)
- **5325:** read
- **5326:** update offset
- **5327:** unlock the inode

Called by **sys\_read**.

---

## **5352 filewrite(struct file \*f, char \*addr, int n)**

Writes from addr to f.

**5358-5359:** handle case when file is pipe

**5360-5386:** handle case when file is inode:

- **5367-5372:** break up data to manageable chunks (for transactions)
  - **5374-5379:** write chunk:
    - **5374:** begin transaction
    - **5376-5377:** write
    - **5379:** end transaction
- 

## **5264 fileclose(struct file \*f)**

Decrements file reference count.

When no references left, actually close file.

**5271-5273:** decrement ref count

**5275:** keep backup of struct file, because we're about to release the lock on the file table (and anything can happen after *that*)

**5278:** release file table because closing file on disk can take a long time

**5280-5281:** handle case when file is pipe (needs to happen once for `read` and once for `write` descriptors of pipe for it to actually close)

**5282-5286:** handle case when file is inode (using transaction because this can require writing on device)

---

## **5851 sys\_pipe(void)**

Allocates two files (read pipe and write pipe).

Expects a vector with two entries (from the input), in order to return the descriptors in.

**5859:** allocate pipe (creates 2 file structs)

**5862:** allocate slots in `ftable`, and point them to the pipe

**5863-5867:** remove files in case of failure

**5869-5870:** put descriptors in vector, for user code

---

## **5701 sys\_open(void)**

Opens or creates inode.

**5710-5715:** create inode (on disk!) - can only create file (not directory)

**5716-5724:** open inode

- **5720-5722:** if tried opening directory in *write* mode, close-n-error

**5726:** create struct for inode, add it to `ftable`

**5734-5738:** set file struct data

---

### **5011 `dirlookup(struct inode *dp, char *name, uint *poff)`**

Finds an inode *under* dp with name that's equal to name.

(`poff` in an optional pointer to the offset of the found inode.)

**5020:** read single `struct dirent` in dp

**5022:** check whether `inum` of current `dirent` is active

**5024:** check whether name of current `dirent` equals name

If we found the droid we're looking for:

**5026-5027:** store offset in `poff` (if we supplied the optional pointer)

**5028-5029:** get actual inode

...And if we did not find it:

**5033:** return 0

---

### **5115 `skipelem(char *path, char *name)`**

A helper function that helps us take apart "long/path/names".

**Returns** the value of `path` *without* the first part, and sets `name` to equal the chopped off head.

(Ignores the first instance of "/" in the path: "\*\*\*/\*\*a/b/c" acts the same as "a/b/c".)

---

### **5189 `namei(char *path)`**

Returns the inode with the matching path.

(If `path` begins with "/", looks in root inode. Else, looks in current working directory inode.)

**5192:** ask `namex` to do the work

---

### **5196 `nameiparent(char *path, char *name)`**

Returns the inode with the matching path *without the last part*.

(If `path` begins with "/", looks in root inode. Else, looks in current working directory inode.)

**5198:** ask `namex` to do the work

---

## **5154 namex(char \*path, int nameiparent, char \*name)**

Returns the inode with the matching path.

(If `nameiparent` is 0, finds actual inode. Else, ignores last part (for case when we want to *create* a new inode).)

**5158-5161:** check if we need to look in root or current working directory

**5163-5180:** loop over parts in path:

- **5164:** i\_lock current inode, because we must
- **5165-5168:** make sure we're looking at a *directory*
- **5169-5173:** if we're looking for all but last (and found it), return it
- **5174-5177:** try to get the next part of the path under the current inode
- **5179:** set current inode to be the found path part inode

**5181-5184:** if looking for all but last (and haven't found id before), return 0

**5185:** happily return our found inode

---

## **5052 dirlink(struct inode \*dp, char \*name, uint inum)**

Writes a new directory entry (`name`, `inum`) into the inode pointed at by `dp`.

**5059-5062:** make sure name doesn't already exist in inode

**5065-5070:** find empty slot in inode

**5072:** prepare to write name

**5073:** prepare to write inum

**5074:** actually write the data (if we reached the end of the inode, `writeli` will increase its size)

---

## **5657 create(char \*path, short type, short major, short minor)**

Creates and returns inode supplied in `path` (expecting `path` to exist up to its last part, which is the inode we're creating).

If inode already exists, opens it.

**5663:** get parent of requested inode

**5667-5674:** check if inode exists (and is a file!)

If so, return it.

If exists but is not file, return 0.

**5676:** get an actual inode from the disk

**5679:** lock the new inode (because we must)

**5680-5682:** set some values. nlink is 1, because parent is linked to new inode

**5683:** updates new values in the disk

**5689:** add the two links every directory has:

- "." self
- ".." parent

**5693:** add the new inode to parent

---

#### **4654 iginode(uint dev, uint inum)**

Opens (and returns) inode.

**4661-4670:** loop over **icache**, looking if inode was already loaded once

- **4663-4667:** check if the inode is already in the cache
- **4668-4669:** find first empty slot, in case we'll need to load inode to it

**4673-4674:** if cache is full (and inode not there), panic (although we could have also gone to sleep till there's room)

**4676-4680:** load some of the inode data to **icache**. Valid bit is set to 0, because we haven't yet read the data from the disk

---

#### **4689 idup(struct inode \*ip)**

Increments reference count of inode (and returns it).

**4691:** lock **icache** before use, because we access inode field

---

#### **4756 iput(struct inode \*ip)**

Decrements reference count of inode, and closes it on the disk if this it will no longer be referenced at all.

**4759:** check if:

1. We're about to close the inode's last reference
2. We're closing a valid inode
3. There are no more names of / links to the inode (so it must be destroyed)

If so:

- **4761:** mark inode as busy (since there's still one reference!), just in case
- **4765:** actually delete inode on disk
- **4767:** update the inode (????)
- **4770:** do something that isn't really needed

**4772:** finally decrement the ref count

---

## **4603 ialloc(uint dev, short type)**

Allocates a new inode *on the disk*, and then *in the memory*.

**4610:** read super-block from disk.

**4612:** loop over all inodes on disk:

- **4613:** read block of current inode
- **4614:** calculate pointer to current inode (using casting so that + `inum%IPB` will add the correct size)
- **4615:** check if inode is free. If so:
  - **4616:** clear inode data
  - **4617:** set type
  - **4618:** re-write entire block, marking new inode as used (because we set type)
  - **4619:** close current block
  - **4620:** allocate inode in memory and return the *memory* inode
- **4622:** close current block

**4624:** no more inodes on disk?! Panic!

---

## **4629 iupdate(struct inode \*ip)**

Updates inode-on-disk from inode-on-memory `ip`.

**4634:** read entire block from disk

**4635:** calculate pointer to inode-on-disk (using casting so that + `inum%IPB` will add the correct size)

**4636-4641:** set inode data to copy of block that is on the memory now

**4642:** re-write entire block on disk

**4643:** close block

---

## **4703 ilock(struct inode \*ip)**

Locks inode, without spinning or preventing interrupts.

**4711-4715:** sleep over inode until it's not busy:

- **4712-4713:** while inode is busy, go back to sleep
- **4714:** mark inode as busy

**4717-4730:** make sure inode is still valid - affects only in-memory inode

- **4718:** read entire block
  - **4719:** calculate pointer to inode
  - **4720-4725:** set data (just in case it changed meanwhile)
  - **4726:** release block
  - **4727:** mark as valid
- 

### **4735 iunlock(struct inode \*ip)**

Unlocks inode.

**4741:** remove "busy" flag

**4742:** wakeup all procs waiting for inode lock.

---

### **5751 sys\_mkdir(void)**

Creates a new directory.

---

### **5513 sys\_link(void)**

Creates a new name (or *shortcut*) for a file.

(But not for a directory!)

---

### **5601 sys\_unlink(void)**

Destroys a name of a file or directory.

---

### **4902 readi(struct inode \*ip, char \*dst, uint off, uint n)**

Actually reads data from the disk.

**4907-4911:** (weird stuff beyond the scope of this course)

**4913-4914:** offset validation

**4915-4916:** uh...

**4918:** for each block from offset till block where we want to finish reading:

- **4919:** read entire current block
- **4920:** figure out how much to read (entire block, or just part)
- **4921:** copy the how-much-to-read data to memory (buffer)

- 4922: close current block
- 

## **4952 writei(struct inode \*ip, char \*src, uint off, uint n)**

Actually reads data from the disk.

4957-4961: (weird stuff beyond the scope of this course)

4963-4966: validation

4968: for each block from offset will block where we want to finish writing:

- 4969 read entire current block (we actually don't really need to do this when we're writing a whole block, but whatever)
- 4970: figure out how much to write (entire block, or just part)
- 4971: copy the how-much-to-write data to memory (buffer)
- 4972: copy buffer to disk
- 4972: close current block

4976-4979: if the file grew because of the write, update inode's size

---

## **4856 itrunc(struct inode \*ip)**

Destroys inode on disk!

(Must be called only when inode is no longer referenced or held open by anyone.)

4862-4867: free direct blocks (if they're allocated) and mark them as such on inode

4869: if there are also indirect blocks:

- 4870: read the block with the indirect pointers
- 4871: cast block to int vector, for convenience
- 4872-4875: free indirect blocks (if they're allocated) - no need to mark them on indirect vector, because we'll destroy him soon
- 4876: close the block with the indirect pointers
- 4877: destroy the block with the (now freed) indirect pointers
- 4878: mark the indirect pointer as free on the inode

4881-4882: update inode

---

## **4810 bmap(struct inode \*ip, uint bn)**

Returns the physical block number of ip's bnth block.

If block doesn't exist, the block is allocated.

**4815-4819:** handle case when block is direct

- **4816:** try to get physical address
- **4817:** if no physical address, allocate one

If we reached here, then we know block is indirect

**4824-4825:** allocate indirect block if it doesn't exist yet

**4826:** read contents of indirect block

**4827:** cast the contents as a vector of numbers

**4828-4831:** try to get physical address

- **4829:** allocate new address if needed
- **4830:** write new address on the indirect block on the disk

**4832:** close indirect block

---

## **4102 bread(uint dev, uint sector)**

Gets a block from the disk.

**4106:** try to get buffer from cache

**4107-4108:** if buffer is not valid, ask for the actual buffer from the driver (which is the layer that *really* reads from the disk)

---

## **4114 bwrite(struct buf \*b)**

Writes a block to the disk.

**4116-4117:** make sure buffer is marked as busy

**4118:** mark buffer as dirty (that's our way to tell the driver to *write*)

**4119:** ask driver layer to write to disk.

---

## **4038 binit(void)**

Initialize bcache buffer cache.

---

## **4066 bget(uint dev, uint sector)**

Gets a buffer from the cache.

If it's not there, allocate it there.

**4070:** lock buffer cache

**4072-4084:** search for buffer in cache

- **4075:** buffer found!
  - **4076-4080:** if buffer is not busy, unlock the buffer cache, mark buffer as busy, and return it
  - **4081-4082:** if buffer is busy, go to sleep till it's unlocked (but will need to search again for case buffer was changed)

Got here?

Buffer not found; allocate new buffer

**4087:** loop from end of list to beginning

- **4088:** check if current buff is not busy and not dirty
    - **4089-4093:** fill data, release lock, return buff
- 

### **4125 brelse(struct buf \*b)**

Release buffer from being BUSY and move to head of linked list.

**4130:** lock cache

**4132-4137:** reposition buff to list head

**4139:** mark buff as not busy

**4140:** wake up anyone who might be waiting for buff

**4142:** release cache lock

---

### **4454 balloc(uint dev)**

Allocates and zeroes a block on the disk.

**4461** read super block

---

### **bfree**

---

### **idewait**

---

### **idestart**

---

## **3954 iderw(struct buf \*b)**

Handles a queue of blocks to write

(because it can receive many requests during the long time it takes the IDE to actually write stuff to the disk).

**3968-3971:** append b to end of idequeue

**3974-3975:** if our b is at the head of the list, read/write it to IDE

**3978-3980:** sleep until b is VALID and NOT DIRTY

---

## **3902 ideintr(void)**

Handles the interrupt from the IDE (which means the disk finished reading/writing).  
Manages idequeue.

**3907-3913:** get current head of idequeue, and move idequeue to next guy.

- **3908-3912:** handle false alarms (which happen sometimes)
- **3913:** "increment" queue

**3916-3917:** read data (if that's what current buff wanted)

**3920-3922:** clear buff flags and wakeup all those waiting for buff

**3925-3926:** tell IDE to start running on next guy in queue