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 `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 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 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 `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 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.

- 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.

Lectures on Operating Systems (Mythili Vutukuru, IIT Bombay)

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 (becasue 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*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 = 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.)

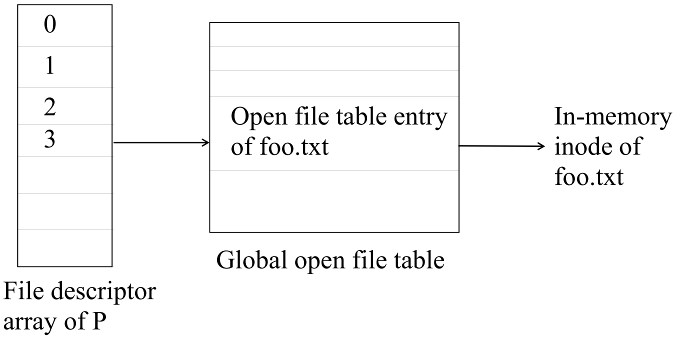
20. Interrupt-based device drivers give superior performance to polling-based drivers because they eliminate the time spent by the CPU in copying data to and from the device hardware. [T/F]

    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.


0
1
2
3
Open file table entry of foo.txt
In-memory inode of foo.txt
Global open file table
File descriptor array of P

(a) 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.

(b) 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.

Lectures on Operating Systems (Mythili Vutukuru, IIT Bombay)

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?

   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 aquire 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 indefinitey. 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 write-Unlock 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 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 tranfer 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 tranfer 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 a1done = 0; b1done = 0; cdone = 0; b2done = 0;

ThreadA:
  get request from queue and process
  up(a1done)
  down(b2 done)
  finish with request

ThreadB:
  down(a1done)
  //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 succesfully 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: `down(s)`, `up(s)`, and `up(s,n)`. While the first two functions are as studied in class, the function `up(s,n)` simply invokes `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 `pthreads` 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 `InitalizeReadWriteLock()` 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.

Lectures on Operating Systems (Mythili Vutukuru, IIT Bombay)

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".

   (a) The page table of the parent process.

   Ans: update because the parent's pages must be marked read-only.

   (b) 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.

   (a) What happens at the TLB? (hit/miss/cannot say)

   Ans: cannot say

   (b) 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.

    (a) What happens at the TLB? (hit/miss/cannot say)

    Ans: miss

    (b) 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.

    (a) Are the physical addresses of the kernel code/data stored in the page tables of various process always the same? (yes/no/cannot say)

    Ans: yes, because there is only one copy of kernel code in RAM

    (b) 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 the 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} = 1K, 2^{20} = 1M, 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) $\approx$ 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 = 4MB.

Each page can hold $2^{12}/4$ = 2^{10} PTEs, so we need $2^{20}/2^{10}$ 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 process, 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.

    (a) 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, ...

    (b) Calculate the number of page faults genrated 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.

    (c) Repeat (b) above for the LRU page replacement algorithm.

    (d) 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:

    (a) 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).

    (b) Page faults with FIFO = 8. Page faults on 0,1,2,3 (replaced 0), 0 (replaced 1), 1 (replaced 2), 2 (replaced 3), 3.

    (c) Page faults with LRU = 6. Page faults on 0, 1, 2, 3 (replaced 2), 2 (replaced 3), 3.

    (d) 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: t1 when the logical memory address is found in TLB cache, t2 when the address is not in TLB but does not cause a page fault, and t3 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 t1,t2, and t3.

    Ans: 0.6*t1 + 0.3*t2 + 0.1*t3

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 = 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 16KB/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): ptr1 = malloc(500); ptr2 = malloc(200); ptr3 = malloc(800); ptr4 = 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) 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 sytem 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: 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 anynomous 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.

Lectures on Operating Systems (Mythili Vutukuru, IIT Bombay)

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 implentations 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 premempt).

    (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.

(a) What is the value of the variable `a` as printed in the child process, when it is scheduled next? Explain.

(b) Will the attempt to read from the file descriptor succeed in the child? Explain.

Ans:

(a) 5. The value is only changed in the parent.

(b) 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 descendents (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 descendents 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 descendents, 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 exchausts 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 comand 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 coresponding 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 NCOUNTER-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 (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, 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 or 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 provied 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 cprintf statements in trap.c to figure out how one can find the virtual address that caused the page fault. Use PGROUNDDOWN(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, becasue 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(pgdir))* 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 asusmptions 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.

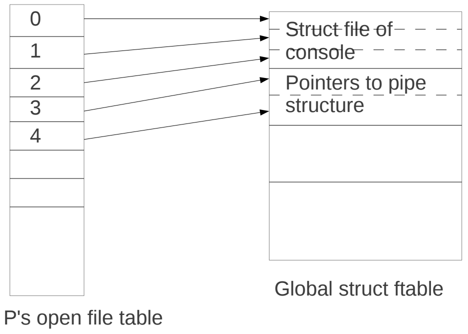
D. The reference count stored in an entry of the `struct ftable` indicates the number of links to the file in the directory tree.

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();
```

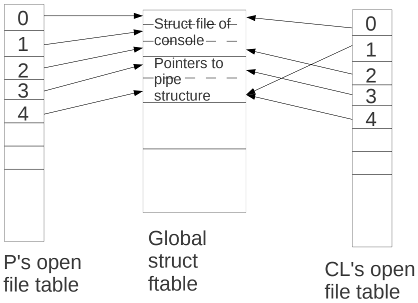
Assume the shell gets the command “echo hello | grep hello”. Let P denote the parent shell process that implements this command, and let CL and CR denote the child processes created to execute the left and right commands of the above pipe command respectively. Assume P has no other file descriptors open, 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.



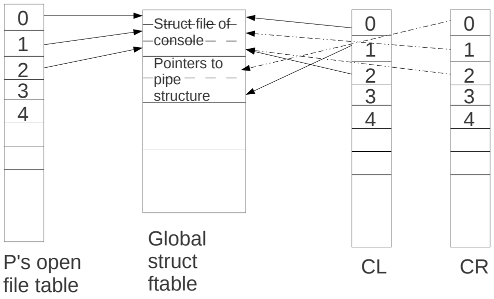
Draw similar figures showing the states of the various open file tables after the execution of lines marked (a) and (b) above. For each of parts (a) and (b), you must clearly draw both the global and per-process file tables, and illustrate the various pointers clearly. You may assume that all created 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

Lectures on Operating Systems (Mythili Vutukuru, IIT Bombay)

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 of 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)

Lectures on Operating Systems (Mythili Vutukuru, IIT Bombay)

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 + KERNBASE always

(b) V = P – KERNBASE always

(c) V = P + KERNBASE only for V ¿= KERNBASE

(d) V = P + KERNBASE only for V ¡ KERNBASE

Ans: (c)

Lectures on Operating Systems (Mythili Vutukuru, IIT Bombay)

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}
```

   (a) 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?

   (b) How is the kernel stack of the newly created child process different from that of the parent?

   (c) 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.

   (d) How would your answer to (c) above change if xv6 implemented copy-on-write during fork?

   (e) 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:

   (a) It contains a trap frame, followed by the context structure.

   (b) 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.

   (c) The EIP value points to the same logical address, but to different physical addresses, as the parent and child have different memory images.

   (d) With copy-on-write, the physical addresses may be the same as well, as long as both parent and child have not modified anything.

   (e) 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 `swtch` 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 an 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. It's 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). Thus 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 computes 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 `icache.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 `readsb` (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 is 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 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, 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 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 to 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.

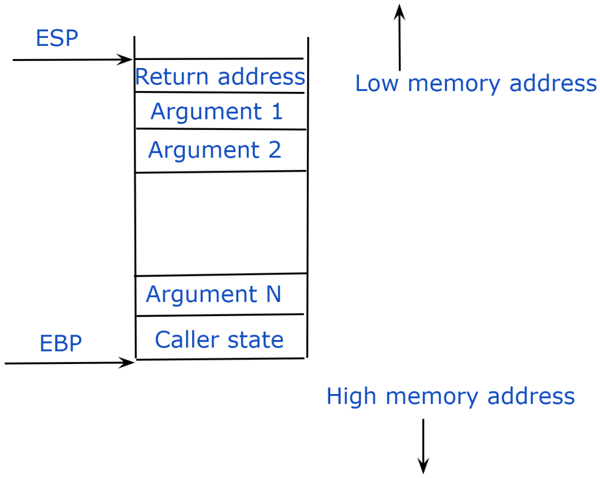
  1. `mov src dst` instruction moves contents of `src` to `dst.` [1] The source/destination can be a constant value, content of a register (e.g., `%eax`), data present at a memory location whose address is stored in a register (e.g., `(%eax)`), or even data present at a memory address that is located at 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.

[1] This 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.

[2] Note 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

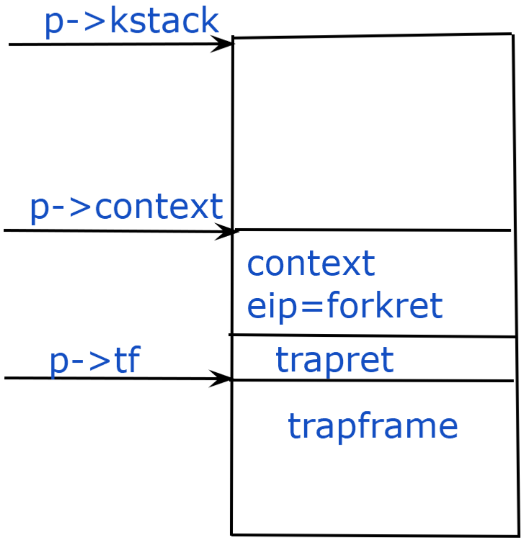
- xv6 supports several system calls, which are handled by the trap function we just studied. You can see the complete set of system calls on sheets 35–36. What happens during a system call? The system calls that user programs in xv6 can use are defined in the file `user.h`, which is the header file for 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 `syscall` (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 thorughout 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.

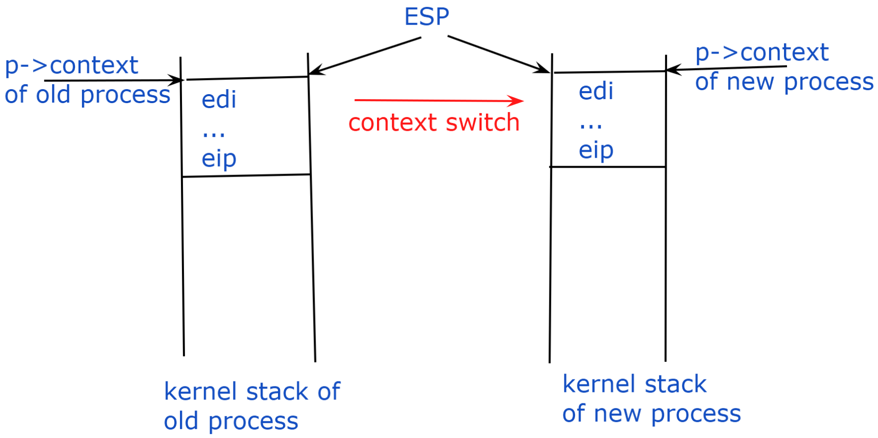
- The `swtch` function (line 2950) does this job of switching between two contexts, and old one and a new one. Let us carefully study this process of context switching. Understanding this function needs some basic knowledge of x86 assembly.

  - 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 asembly 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 funtion 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 parantheses 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

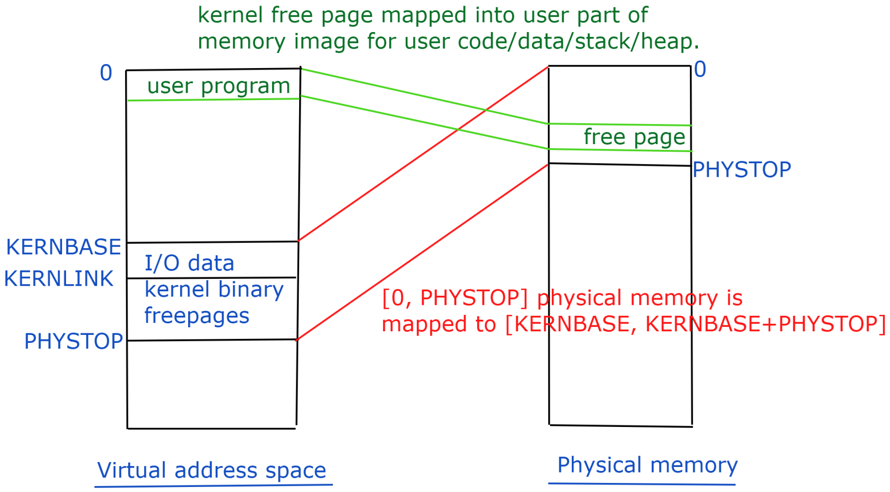
- xv6 uses 32-bit virtual addresses, resulting in a virtual address space of 4GB. xv6 uses paging to manage its memory allocations. However, xv6 does not do demand paging, so there is no concept of virtual memory. That is, all valid pages of a process are always allocated physical pages.

- 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 addres (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.


kernel free page mapped into user part of memory image for user code/data/stack/heap.
0
user program
KERNBASE
KERNLINK
I/O data
kernel binary
freepages
PHYSTOP
0
free page
PHYSTOP
[0, PHYSTOP] physical memory is mapped to [KERNBASE, KERNBASE+PHYSTOP]
Virtual address space
Physical memory

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 `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 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 `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 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 `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 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.

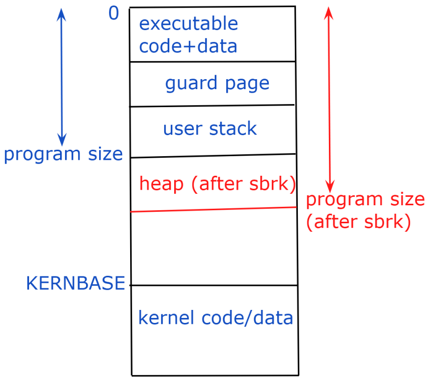
- Next, look at the implementation of the sbrk system call (line 3701). This system call can be invoked by a process to grow/shrink the userspace part of the memory image. For example, the heap implementation of malloc can use this system call to grow/shrink the heap when needed. This system call invokes the function growproc (line 2531), which uses the functions allocuvm or deallocuvm to grow or skrink 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

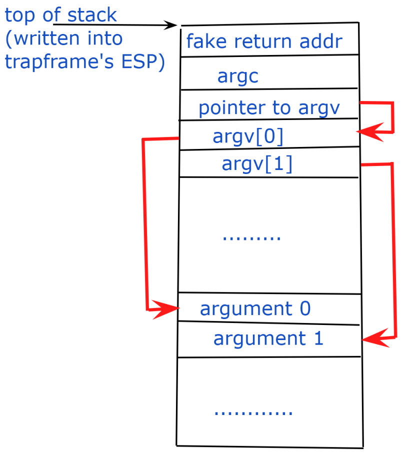
ELF binary is composed of many segments). We have already seen that allocuvm (line 1953) allocates physical pages from the kernel's free pool via calls to kalloc, and sets up page table entries, thereby expanding the virtual/physical address space of the process. loaduvm (line 1918) reads the memory 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.

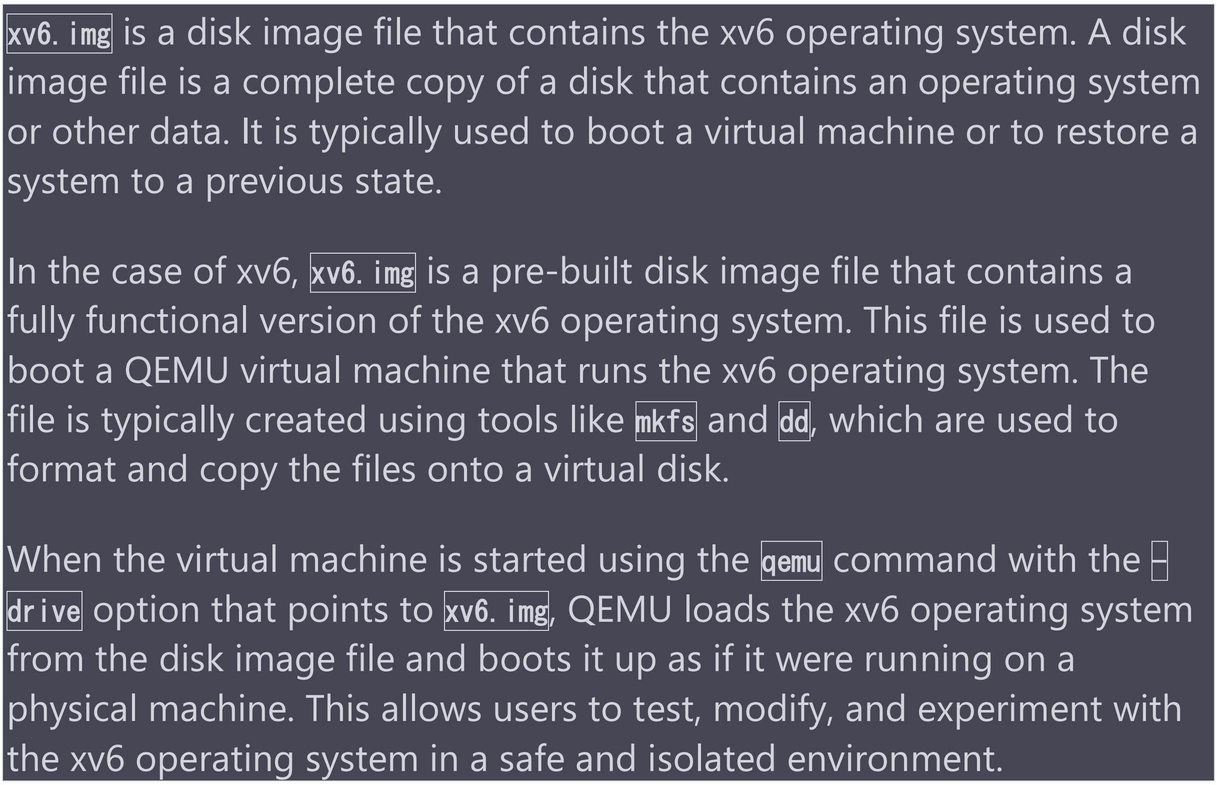
The `.bochsrc` file in the xv6 code is a configuration file used by the Bochs IA-32 emulator. Bochs is a software emulation program that emulates an entire PC system, including the CPU, memory, disk, and other components. The `.bochsrc` file specifies the settings for the Bochs emulator, such as the 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-lgpl-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.

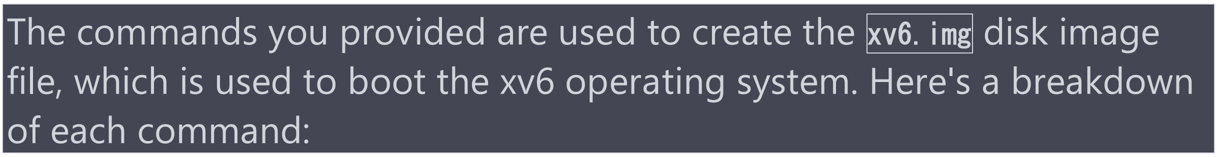


`xv6.img` is a disk image file that contains the xv6 operating system. A disk image file is a complete copy of a disk that contains an operating system or other data. It is typically used to boot a virtual machine or to restore a system to a previous state.

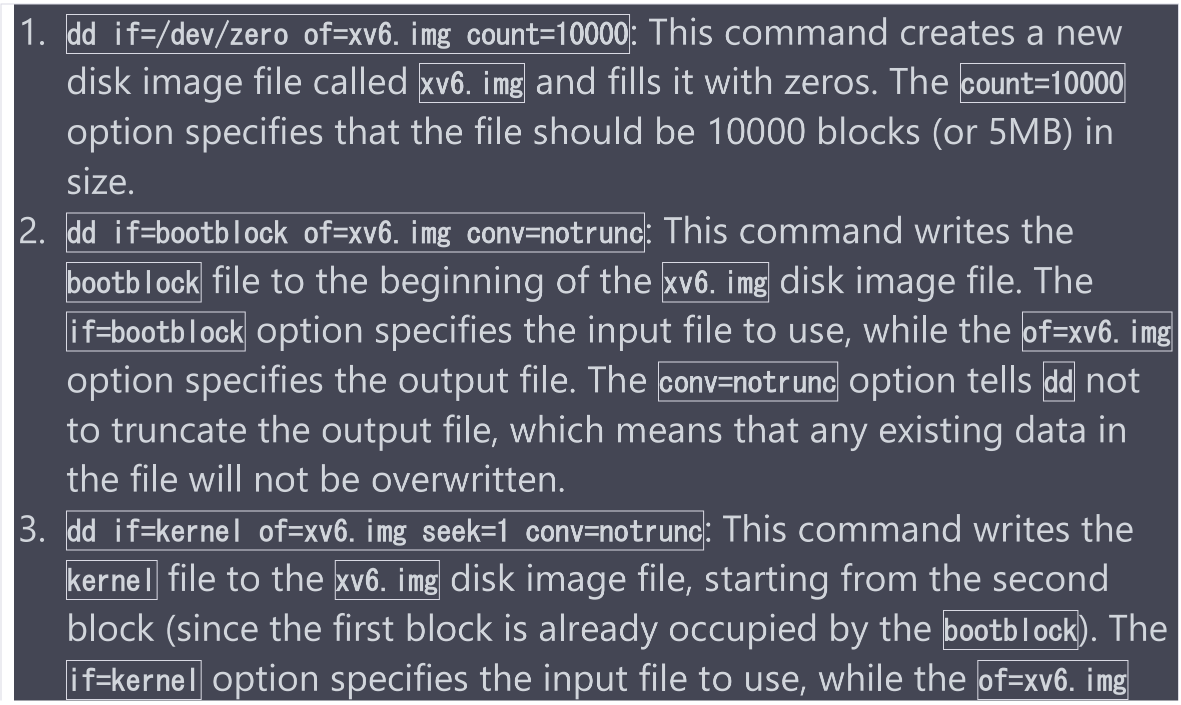
In the case of xv6, `xv6.img` is a pre-built disk image file that contains a fully functional version of the xv6 operating system. This file is used to boot a QEMU virtual machine that runs the xv6 operating system. The file is typically created using tools like `mkfs` and `dd`, which are used to format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

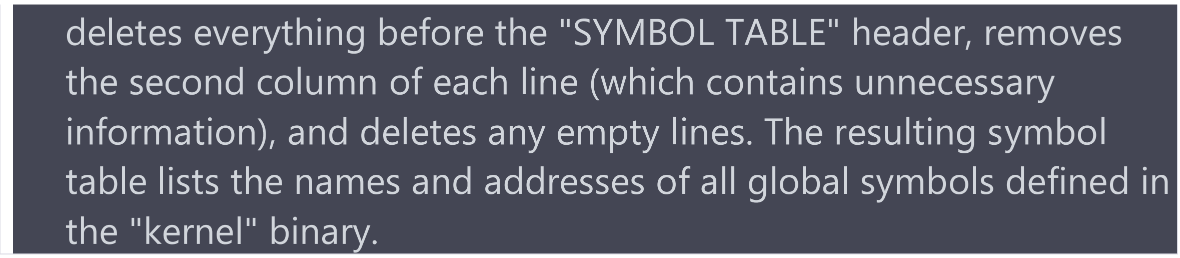
1. `bootblock: bootasm.S bootmain.c` - This line specifies the target "bootblock" and its dependencies, which are the source code files "bootasm.S" and "bootmain.c". This means that in order to create the "bootblock" binary, these source files must be compiled.
2. `$(CC) $(CFLAGS) -fno-pic -O -nostdinc -I. -c bootmain.c` - This line invokes the C compiler (`$(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

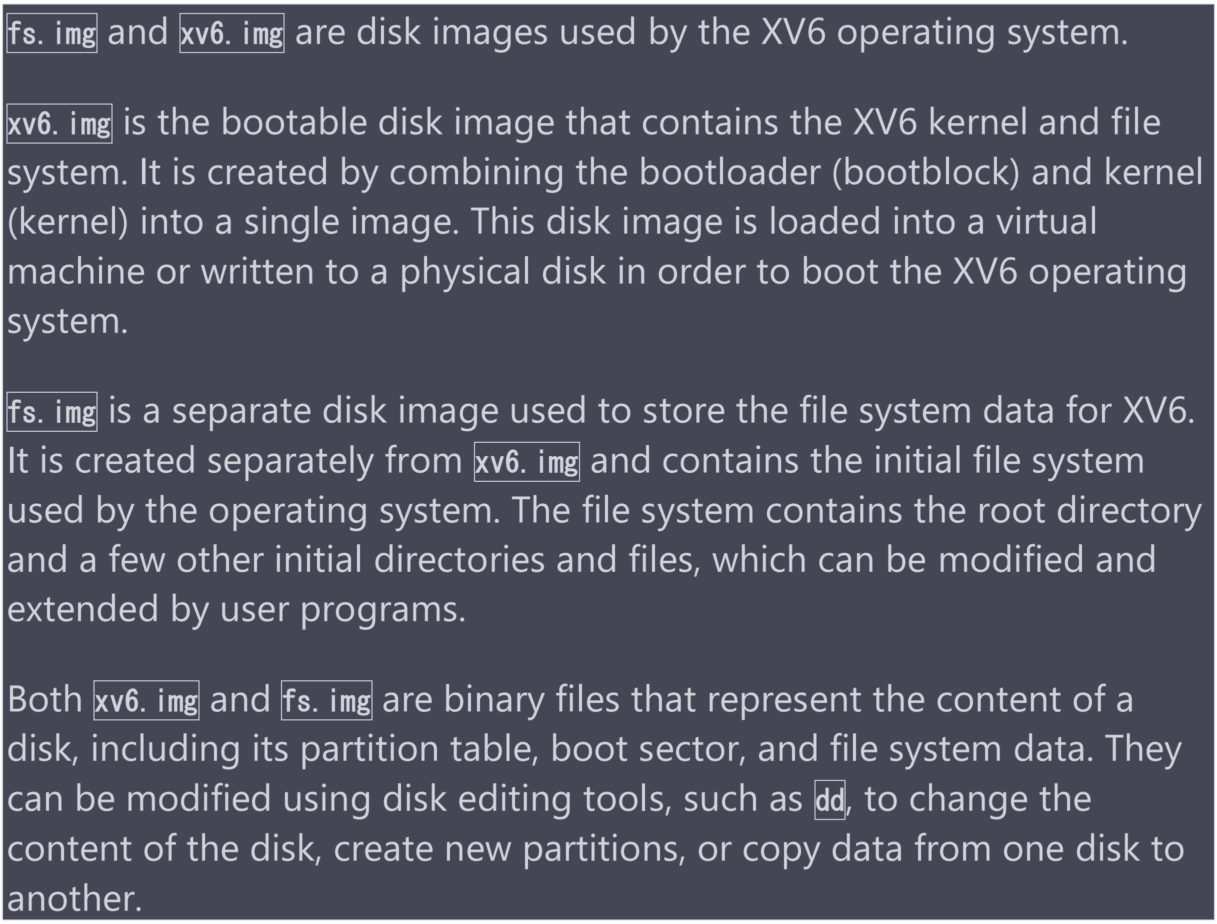
The code you provided is a Makefile rule for building a kernel program. Here's what each line does:

1. `kernel: $(OBJS) entry.o entryother initcode kernel.ld` - This line specifies the target "kernel" and its dependencies, which are the object files listed in the variable `$(OBJS)`, as well as the "entry.o", "entryother", "initcode", and "kernel.ld" files. This means that in order to create the "kernel" binary, these files must be linked together.
2. `$(LD) $(LDFLAGS) -T kernel.ld -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.ld` (use the linker script file "kernel.ld"), `-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



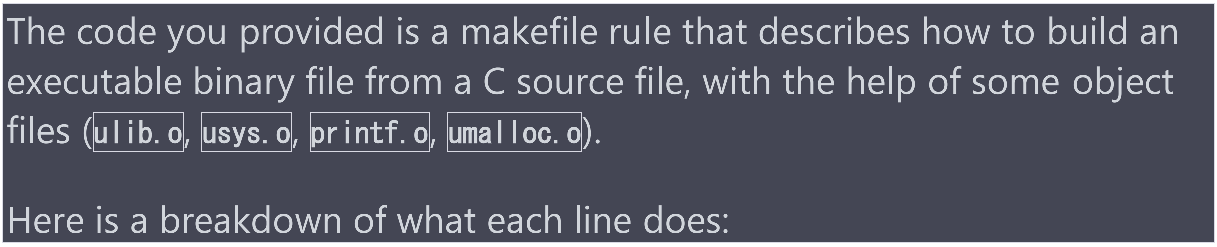
`fs.img` and `xv6.img` are disk images used by the XV6 operating system.

`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.

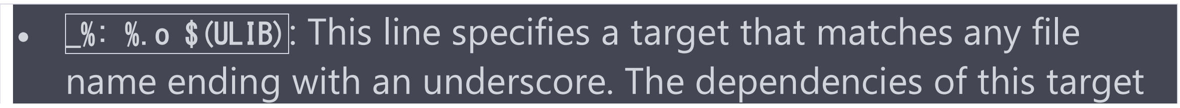
Both `xv6.img` and `fs.img` are binary files that represent the content of a disk, including its partition table, boot sector, and file system data. They can be modified using disk editing tools, such as `dd`, to change the content of the disk, create new partitions, or copy data from one disk to another.

7.compiling userland programs

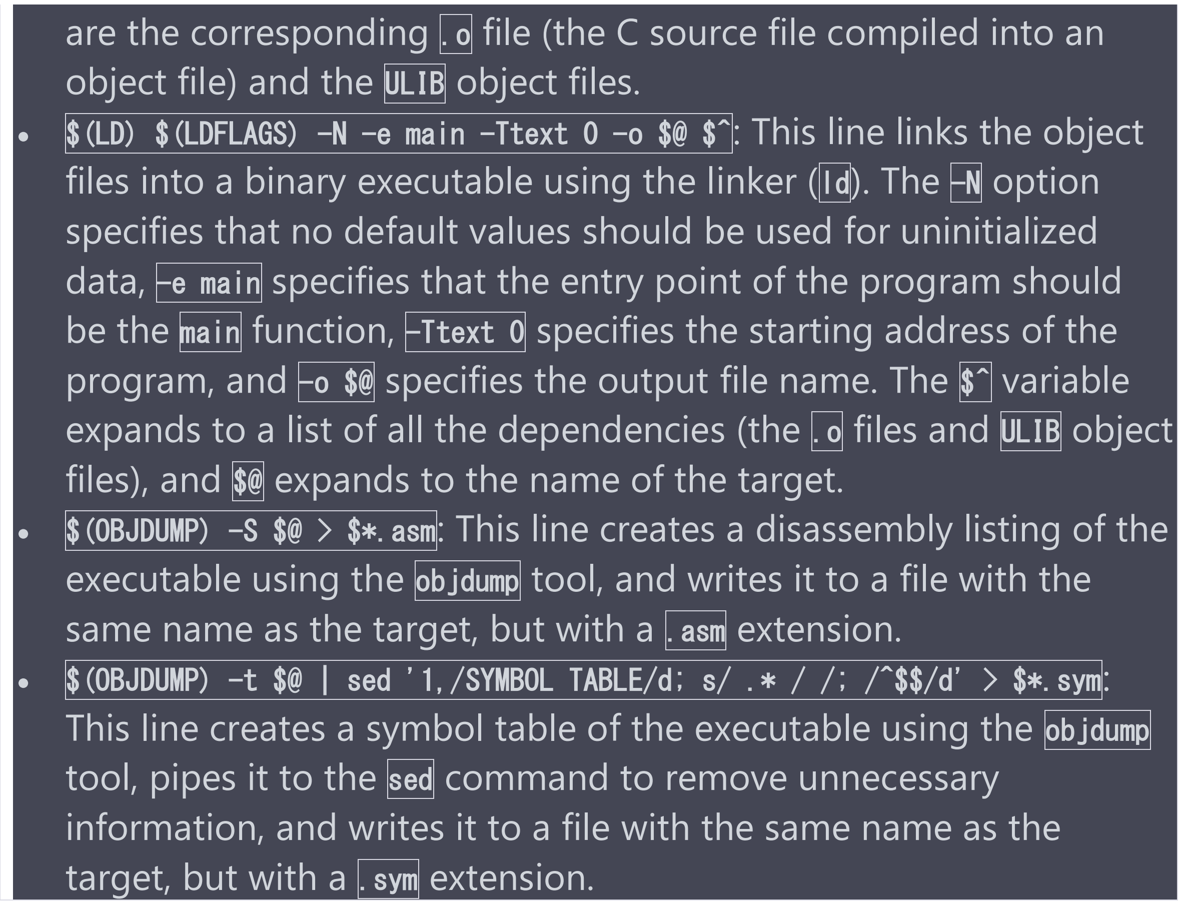


The code you provided is a makefile rule that describes how to build an executable binary file from a C source file, with the help of some object files (`ulib.o`, `usys.o`, `printf.o`, `umalloc.o`).

Here is a breakdown of what each line does:

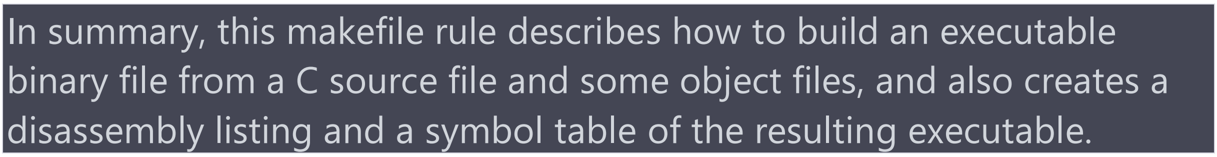


- `_%: %.o $(ULIB)`: This line specifies a target that matches any file name ending with an underscore. The dependencies of this target



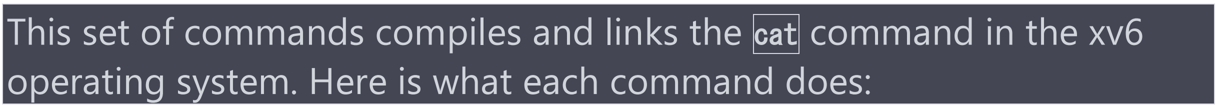
are the corresponding `.o` file (the C source file compiled into an object file) and the `ULIB` object files.

- `$(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.

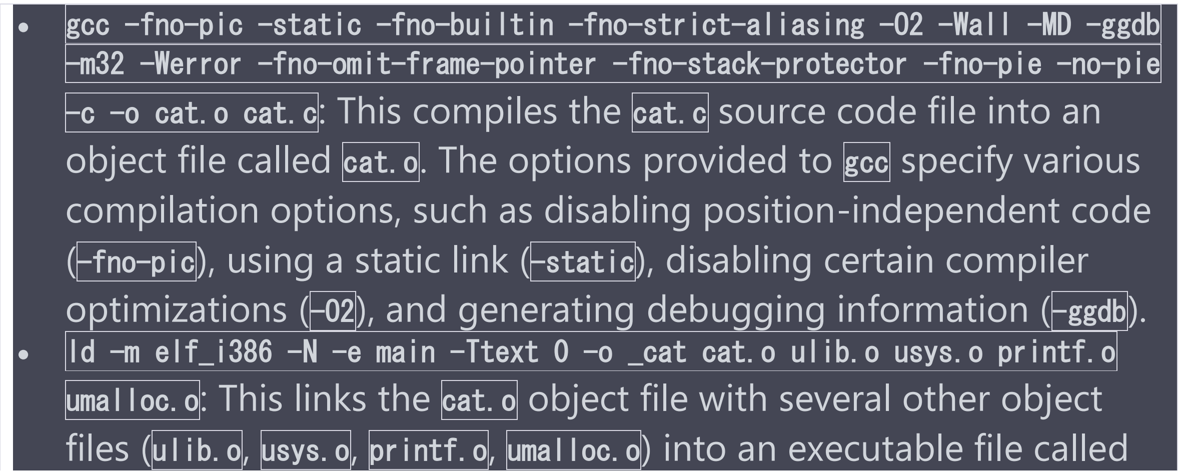


In summary, this makefile rule describes how to build an executable binary file from a C source file and some object files, and also creates a disassembly listing and a symbol table of the resulting executable.

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.

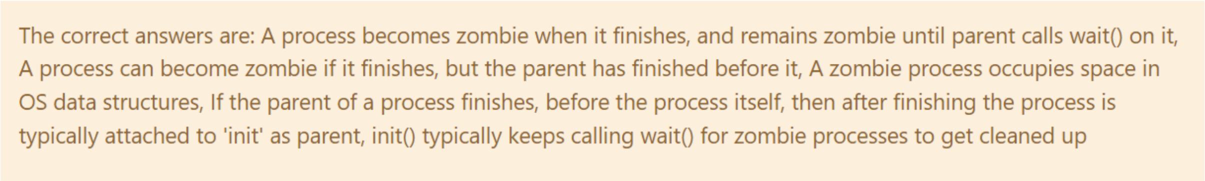
When the computer needs to interact with a hardware device, it sends a request to the device driver, which then communicates with the device to carry out the requested action. For example, if a user wants to print a document, the computer sends a print request to the printer driver, which then translates 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.

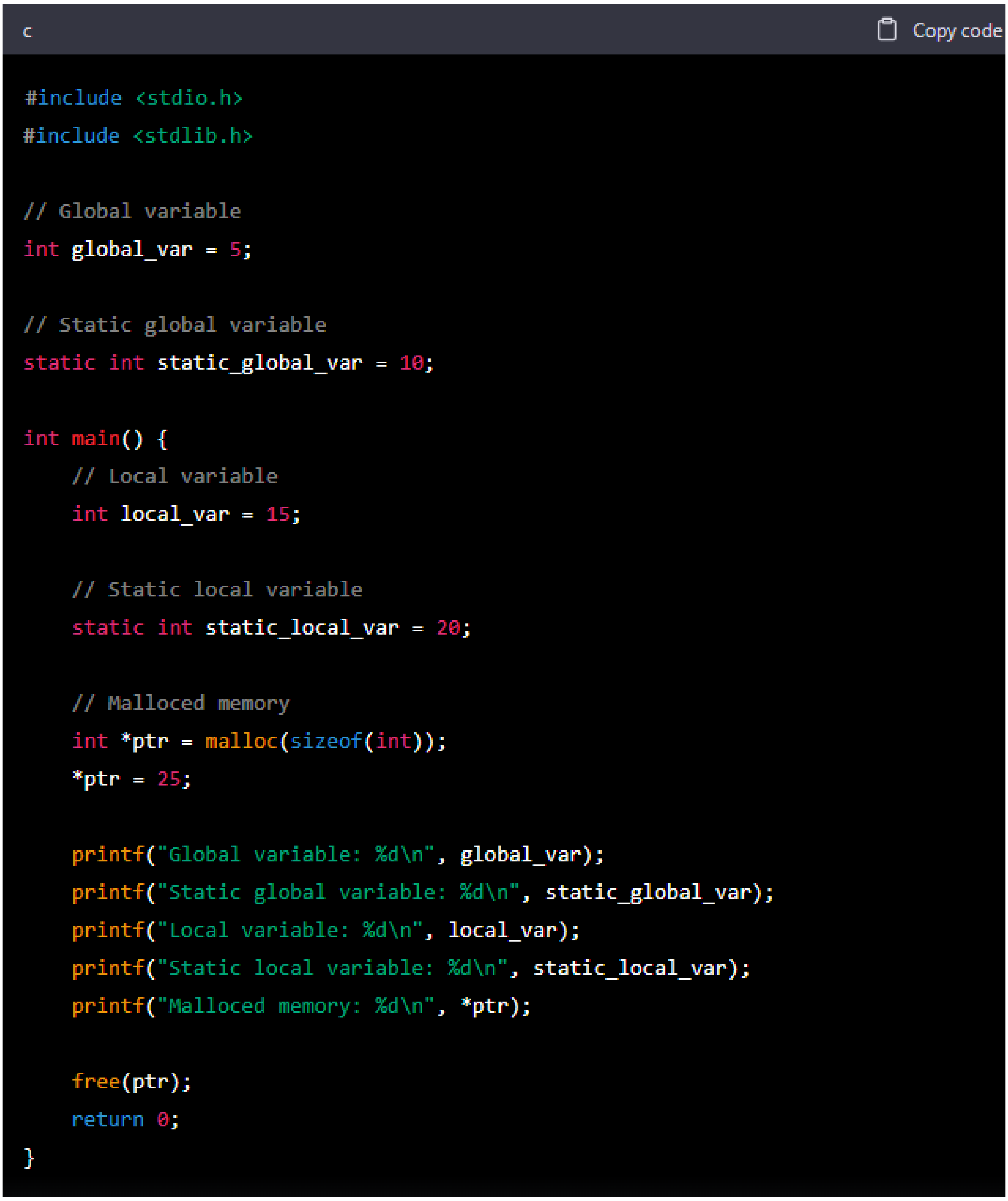
The process accounting information typically includes data such as:

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?

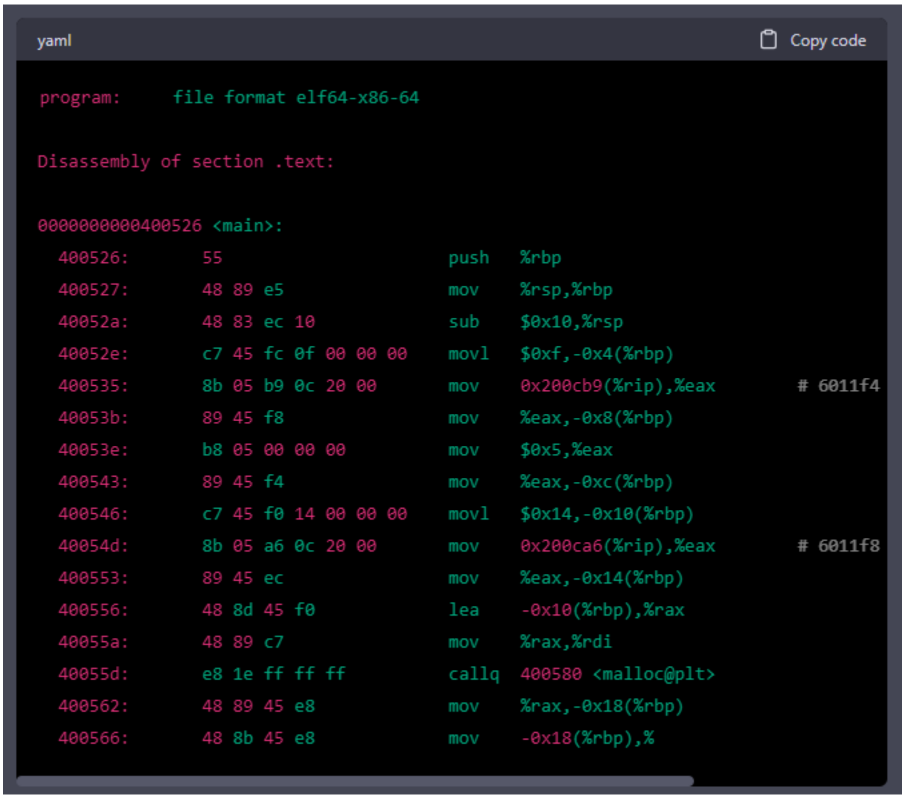
Here is an example C program that uses globals, local variables, static local variables, static global variables, and malloced memory:



```c
#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:



```
program:     file format elf64-x86-64

Disassembly of section .text:

0000000000400526 <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.

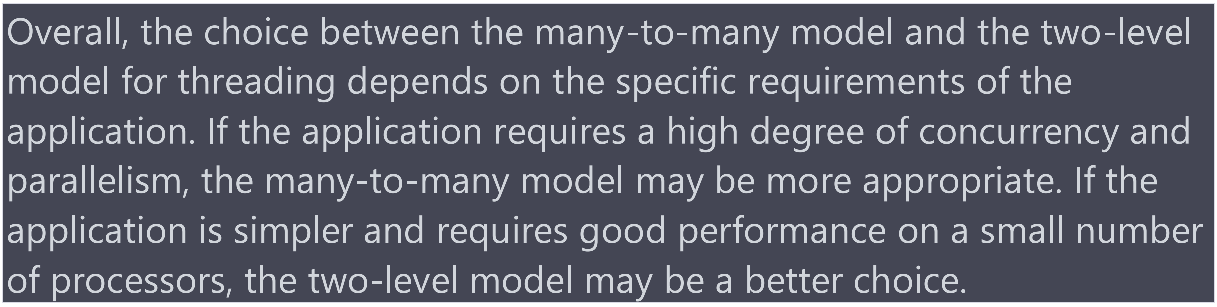
4. difference between many to many and 2 level model

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.

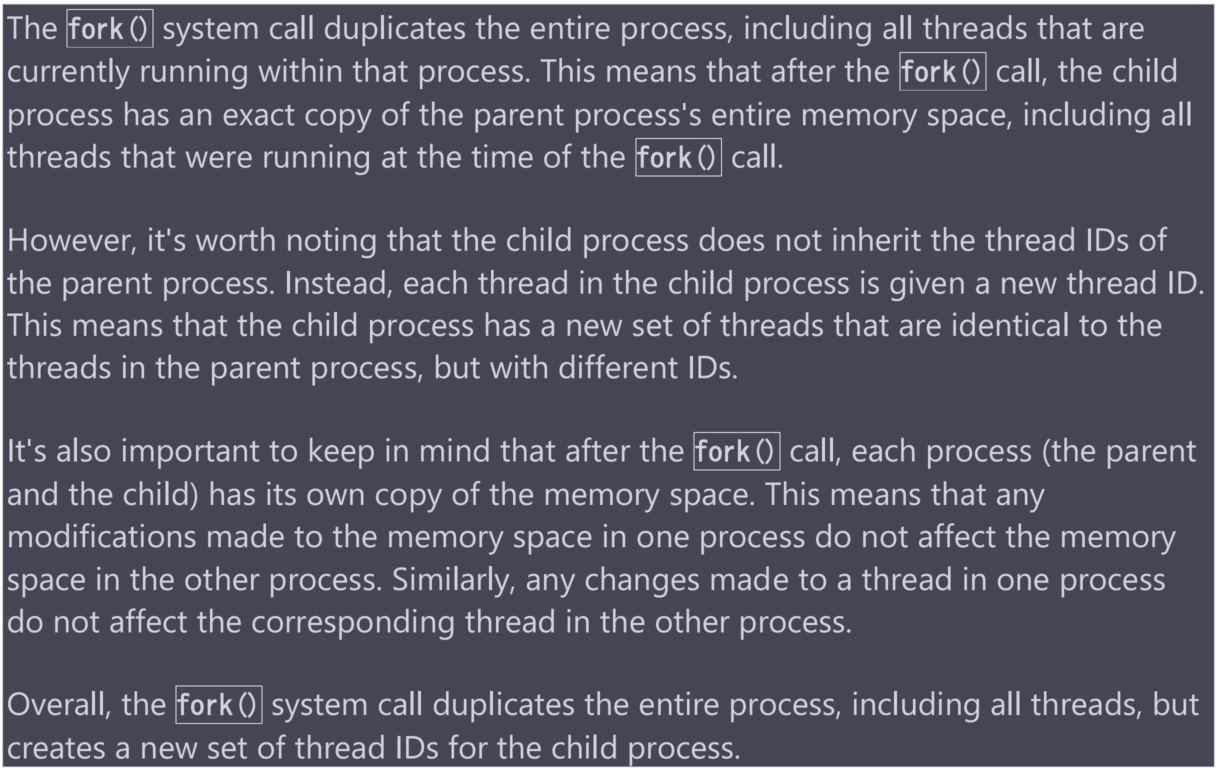
However, the many-to-many model can also be more complex to implement than the two-level model, since it requires coordination between the user-level thread library and the operating system's kernel. The thread library must manage a pool of kernel-level threads, and ensure that each user-level thread 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?



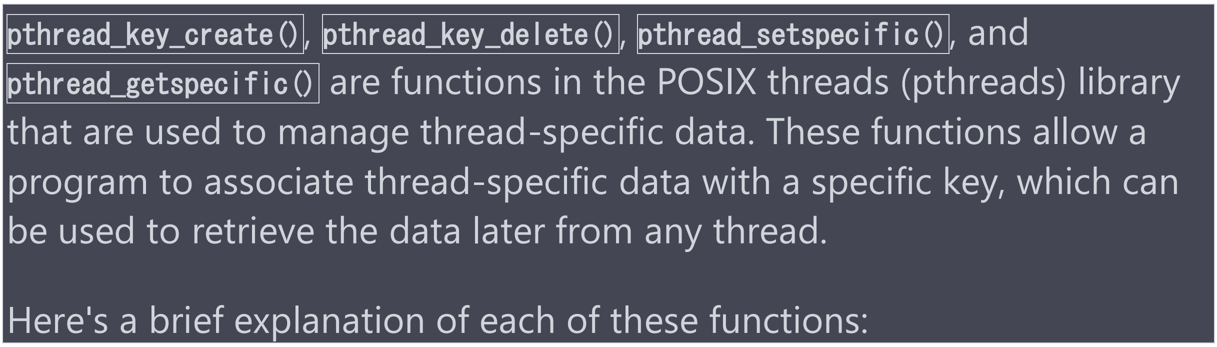
The `fork()` system call duplicates the entire process, including all threads that are currently running within that process. This means that after the `fork()` call, the child process has an exact copy of the parent process's entire memory space, including all threads that were running at the time of 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.

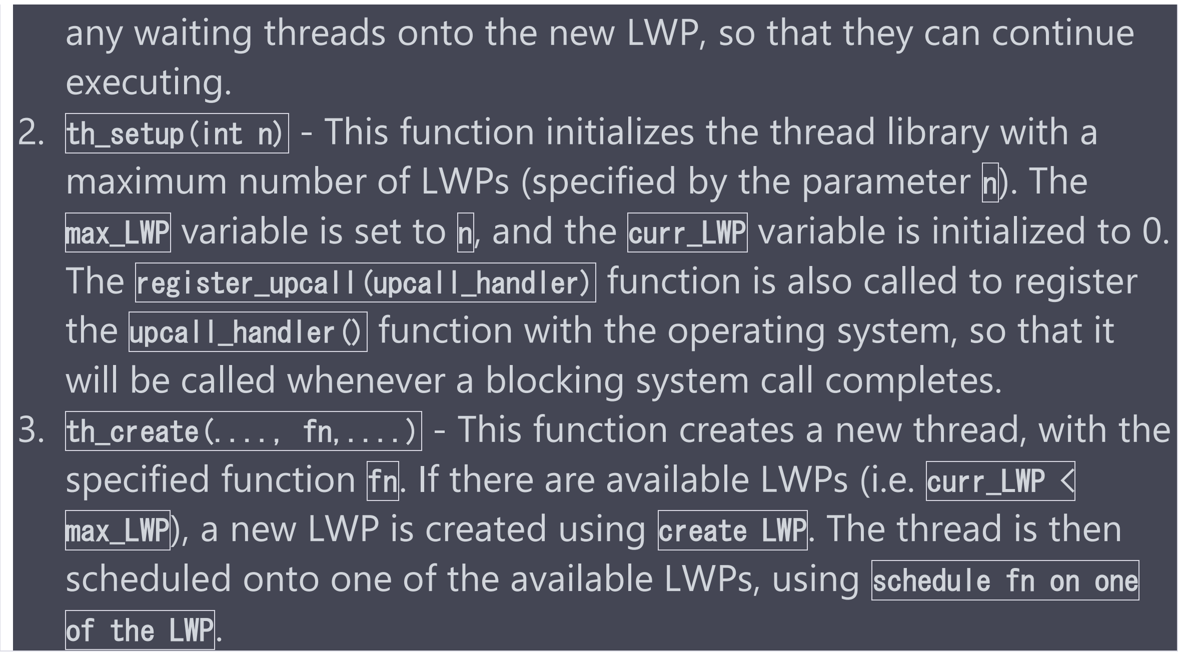
2. `pthread_key_delete()` - This function deletes a previously created thread-specific data key. The function takes one parameter: the key to be deleted. Any data associated with the key in all threads will also be deleted. The function returns zero on success, or an error code if the key could not be 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

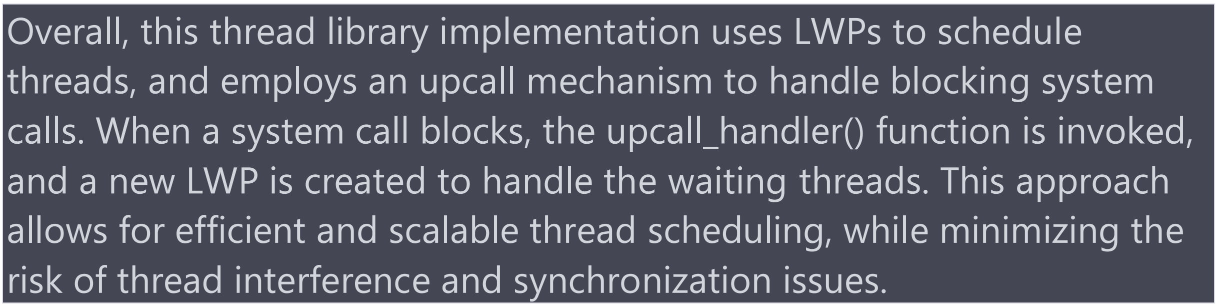
This code snippet outlines a simple thread library implementation that uses lightweight processes (LWPs) to schedule threads. Here's a brief explanation of each of the functions:

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.

4.

Lectures on Operating Systems (Mythili Vutukuru, IIT Bombay)

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}
   ```

   (a) 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?

   (b) How is the kernel stack of the newly created child process different from that of the parent?

   (c) 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.

   (d) How would your answer to (c) above change if xv6 implemented copy-on-write during fork?

   (e) 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:

   (a) It contains a trap frame, followed by the context structure.

   (b) 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.

   (c) The EIP value points to the same logical address, but to different physical addresses, as the parent and child have different memory images.

   (d) With copy-on-write, the physical addresses may be the same as well, as long as both parent and child have not modified anything.

   (e) 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 `swtch` 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 an 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.