CSCI 4730/6730 FINAL EXAM **File-System Management**: files organized into directories & access control determine who can access them (OS Activities include) creating/deleting files/directories, primitives to manipulate files/directories, mapping files onto secondary storage, backup files onto stable (non-volatile) storage media **Mass-Storage Management**: (OS Activities) (un)mounting, free-space management, storage allocation, disk scheduling, partitioning, protection Caching: information in use is copied from slower to faster storage temporarily. the cache is smaller than the storage being cached. **Cache management**: important design problem, Careful selection of the cache size and of a replacement policy can result in greatly increased performance **I/O Subsystem**: consist of several components: a memory-management component that includes buffering, caching, and spooling; a general device-driver interface; drivers for specific hardware devices. **Protection**: any mechanism for controlling the access of processes or users to the resources defined by a computer system. **Security**: defend the system from external and internal attacks. **Mobile computing**: refers to computing on handheld smartphone and tablet computers. **Virtualization**: is a technology that allows operating systems to run as applications within other operating systems. **Emulation**: is used when the source CPU type is different from the target CPU type. Emulation comes at a heavy price. **Interpretation**: A form of emulation. in that the high-level language code is translated to native CPU instructions, emulating not another CPU but a theoretical virtual machine on which that language could run natively. **Cloud computing**: is a type of computing that delivers computing, storage, and even applications as a service across a network. In some ways, it's a logical extension of virtualization, because it uses virtualization as a base for its functionality. **Embedded systems**: almost always run real-time operating systems They tend to have very specific tasks. Usually, they have little or no user interface, preferring to spend their time monitoring and managing hardware devices, such as automobile engines and robotic arms. **Compile to Assembly**: gcc -std=c17 -Wall -pedantic-errors -g -O0 -S filename.c **Assemble**: as -o filename.o filename.s **Linking**: gcc -o filename filename.o other.o **Preprocess**: gcc -E filename.c **Exam 1 Material:** **Anatomy of a process** As processes enter the system, put into the ready queue, where they are ready & waiting to execute on CPU. (generally stored as a linked list) processes waiting for an event to occur placed in a wait queue. **Process Termination** **exit**(3) - normal process termination **_exit**(2) - immediate process termination **atexit**(3) - register function to be called at normal process termination **on_exit**(3) - register a fancy function to be called at normal process termination **Process Creation** **fork**(2) - create a child process; separate address space than parent **vfork**(2) - create a child process; same address space as parent **wait**(2) - wait for process to change state **waitpid**(2) - suspend execution of the calling process until one of its children terminates. **function calls and system calls** - both in C and x86_64 assembly Three general methods are used to pass parameters to the operating system The simplest approach is to pass the parameters in registers. In some cases, however, there may be more parameters than registers In these cases, the parameters are generally stored in a block, or table in memory, and the address of the block is passed as a parameter in a register Parameters also can be placed, or pushed, onto the stack by the program and popped off the stack by the operating system. System calls can be grouped roughly into six major categories: **Process control**: A running program needs to be able to halt its execution either normally (exit()) or abnormally (abort()). **File manipulation**: We first need to be able to create() and delete() files. Once the file is created, we need to open() it and to use it. We may also read(), write(),or reposition()(rewind or skip to the end of the file, for example). Finally, we need to close() the file. **Device manipulation**: A process may need several resources to execute—main memory, disk drives, access to files, and so on. If the resources are available, they can be granted, and control can be returned to the user process (request device, release device, get device attributes, set device attributes) **information maintenance**: Many system calls exist simply for the purpose of transferring information between the user program and the operating system. For example a system call to return the current time() and date().(get system data, set system data) **Communications**: There are two common models of interprocess communication: the message passing model and the shared-memory model. In the message-passing model, the communicating processes exchange messages with one another to transfer information. The source of the communication, known as the client, and the receiving daemon, known as a server, then exchange messages by using read message() and write message() system calls. The close connection() call terminates the communication. In the shared-memory model, processes use shared memory create() and shared memory attach() system calls to create and gain access to regions of memory owned by other processes. normally, the operating system tries to prevent one process from accessing another process's memory. Shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas. Shared memory allows maximum speed and convenience of communication, since it can be done at memory transfer speeds when it takes place within a computer. **Protection**: provides a mechanism for controlling access to the resources provided by a computer system **facilitating context switches** - everything related to your uthreads project; Create; Setup thread; Mmap; Stack pointer; Stack size; Start timer; Was timer started by a different thread? If not, change default action of SIGPROF to custom timer handler; Set timer to fire every 5000 usec, sending SIGPROF; Return 0 or errno; Timer (fires every 5000 usec); Unblock signals; Check global pointer to current thread; Curr->state = running? Then we should save and enqueue; Dequeue next thread; If the state is waiting and not new, then it must've been run before; Longjmp; Else; If setjmp(curr->buf) == 0; Change stack pointer; Run function; If we get this far, call; uthread_exit; Join; While(curr->state != DONE); Exit; Curr->state = DONE; Enqueue; Block signals; Enqueue; Unblock signals; Dequeue; Block signals; Dequeue; Unblock signals; Self; Return curr; scheduling algorithms; **Exam 2 Material**: **library disassembly, tracing system calls** **ldd**(1) - print shared object dependencies. ldd prints the shared objects (shared libraries) required by each program or shared object specified on the command line. In the usual case, ldd invokes the standard dynamic linker (see ld.so(8)).This causes the dynamic linker to inspect the program's dynamic dependencies, and find (according to the rules described in ld.so(8)) and load the objects that satisfy those dependencies. For each dependency, ldd displays the location of the matching object and the (hexadecimal) address at which it is loaded. **vdso**(7) - overview of the virtual ELF dynamic shared object. The "vDSO" (virtual dynamic shared object) is a small shared library that the kernel automatically maps into the address space of all user-space applications. Why does the vDSO exist at all? There are some system calls the kernel provides that user-space code ends up using frequently, to the point that such calls can dominate overall performance. The vDSO (Virtual Dynamically linked Shared Objects) is a memory area allocated in user space which exposes some kernel functionalities at user space in a safe manner. mechanisms used to accelerate certain system calls in Linux. For instance, gettimeofday is usually invoked through this mechanism. The first mechanism introduced was vsyscall, which was added as a way to execute specific system calls which do not need any real level of privilege to run in order to reduce the system call overhead. Following the previous example, all gettimeofday needs to do is to read the kernel's the current time. There are applications that call gettimeofday frequently (e.g to generate timestamps), to the point that they care about even a little bit of overhead. To address this concern, the kernel maps into user space a page containing the current time and a fast gettimeofday implementation (i.e. just a function which

reads the time saved into vsyscall). Using this virtual system call, the C library can provide a fast gettimeofday which does not have the overhead introduced by the context switch between kernel space and user space usually introduced by the classic system call model INT 0x80 or SYSCALL. **libc**(7) - overview of standard C libraries on Linux. The term "libc" is commonly used as a shorthand for the "standard C library", a library of standard functions that can be used by all C programs (and sometimes by programs in other languages). The pathname /lib/libc.so.6 (or something similar) is normally a symbolic link that points to the location of the glibc library, and executing this pathname will cause glibc to display various information about the version installed on your system **ld.so**(8) - dynamic linker/loader find and load the shared objects (shared libraries) needed by a program, prepare the program to run, and then run it. **nm**(1) - list symbols from object files nm displays the symbol table associated with an object, archive library of objects, or executable file. nm recognizes several different file types that may contain symbol tables: object files ending in .obj. These files may be Intel OMF (Object Module Format) files or COFF (Common Object File Format) files. library files, which normally end in .lib and contain one or more OMF or COFF files. Windows executable files, which normally end in .exe, that contain a symbol table compatible with those produced by PLINK86, the Microsoft Linker, the Watcom Linker, or the Borland Linker.**A symbol table** may be created from a .map file (such as those produced by Microsoft link) by using unstrip to create a symbol table compatible with PLINK86. 1. ld used for linking 2. ldd main used to check the library dependency from the executable file 3. vdso to make the program execute faster 4. ld-linux library in charge of loading the argv, load the shared libraries 5. nm main to list symbols from object file (nm /lib64/libc.so.6 | grep longjmp) **tracing system calls** **Strace**(1) - trace system calls and signals. strace is a useful diagnostic, instructional, and debugging tool. System administrators, diagnosticians and trouble-shooters will find it invaluable for solving problems with programs for which the source is not readily available since they do not need to be recompiled in order to trace them. Students, hackers and the overly-curious will find that a great deal can be learned about a system and its system calls by tracing even ordinary programs. And programmers will find that since system calls and signals are events that happen at the user/kernel interface, a close examination of this boundary is very useful for bug isolation, sanity checking and attempting to capture race conditions. **memory management - malloc(3), calloc(3), realloc(3)**, and free(3) **Malloc** The malloc() function allocates size bytes and returns a pointer to the allocated memory. Normally, malloc() allocates memory from the heap, and adjusts the size of the heap as required, using sbrk(2). When allocating blocks of memory larger than MAP_THRESHOLD bytes, the glibc malloc() implementation allocates the memory as a private anonymous mapping using mmap(2). Malloc(131049) will use mmap, while malloc(131048) will still use brk since 131048 + 24 = 128 kB = 128 * 1024 = 131072 bytes **Calloc** The calloc() function allocates memory for an array of nmemb elements of size bytes each and returns a pointer to the allocated memory. The memory is set to zero. **Realloc** The realloc() function changes the size of the memory block pointed to by ptr to size bytes. The contents will be unchanged in the range from the start of the region up to the minimum of the old and new sizes. If the new size is larger than the old size, the added memory will not be initialized.**Free** The free() function frees the memory space pointed to by ptr, which must have been returned by a previous call to malloc(), calloc(), or realloc(). **SBRK** brk() and sbrk() change the location of the program break, which defines the end of the process's data segment (i.e., the program break is the first location after the end of the uninitialized data segment). Increasing the program break has the effect of allocating memory to the process; decreasing the break deallocates memory. Calling sbrk(0) can be used to determine the current program break. **MMAP** mmap() creates a new mapping in the virtual address space of the calling process. The starting address for the new mapping is specified in addr. The length argument specifies the length of the mapping (which must be greater than 0). mmap() uses addresses outside your program's heap area, so heap fragmentation isn't a problem. mmap works by manipulating your process' page table, a data structure your CPU uses to map address spaces. The CPU will translate "virtual" addresses to "physical" ones, and does so according to the page table set up by your kernel. 131048 is the maximum ceiling for brk. 131049 is the minimum for mmap **synchronization - critical section problem, Peterson's solution, mutexes vs. semaphores** **Critical section**: Critical section is a part of the code responsible for changing data that must only be executed by one thread or process at a time to avoid RACE CONDITION. Each process/thread has a critical section. A solution for a critical section problem must satisfy three requirements: Mutual exclusion, progress and bounded waiting. **Mutual exclusion** - If there is one process executing its critical section, then no other process can executing in their critical section **Progress** - If no process is executing its critical section and some processes wish to enter their critical section, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely **Bounded waiting** - There is a limit on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted. Two general approaches are used to handle critical sections in OS: **Preemptive kernel**: allows a process to be preempted and **Nonpreemptive kernel**: does not allows preemption, a kernel mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU **Peterson's Solution:** It only uses shared memory for communication and its original formulation only worked with two processes, but the algorithm can be generalized for more than two. Peterson's solution requires the two processes to share two data items: Whose turn it is to enter the critical section (int turn;), A flag if its ready to enter critical section (bool flag[2] = {false, false}; while (true) {// entry section\n flag[i] = true; turn = j; while (flag[j] && turn == j); // spin\n// critical section\n// exit section\nflag[i] = false;// remainder section\n} **mutexes vs. semaphores Mutex**: Mutual Exclusion lock, Software tool for ensuring mutual exclusion, accessed through two standard atomic operations: acquire and release. It can be used to protect a critical section.acquire(m) - Blocks while m is locked, then locks m release(m) - unlocks m This is a implementation not limited to pthreads. It can synchronize processes as long the atomic_flag (mut_t) is in shared memory.typedef atomic_flag mut_t; volatile mut_t mut = ATOMIC_FLAG_INIT; // false; true = locked; false = unlocked // void acquire(mut_t * m) #define acquire(m) while (atomic_flag_test_and_set(m)) // void release(mut_t * m) #define release(m) atomic_flag_clear(m) The atomic_flag_test_and_set function checks the flag value. If the value is false, then it sets the flag to true (i.e., locks the mutex). This function is guaranteed to be atomic by <stdatomic.h>. Therefore, the only way to break the while loop is if the flag is false. Otherwise it will busy wait until it encounters the mutex in an unlocked state. Atomicity is important because we do not want threads to acquire the lock in between comparison. Pthreads comes with its own mutex implementation pthread_mutex_t mut; #define acquire(m) pthread_mutex_lock(m) #define release(m) pthread_mutex_unlock(m) **Semaphore** A semaphore is an integer variable that, apart from initialization, is accessed only through two standard atomic operations wait(s) - blocks while s<= 0, then decrements; signal(s) – increments s. A semaphore is used to limit access to a resource to a specific number of threads. The initial value of the semaphore denotes the limit. Just like a bouncer in a club. Once the bar is full (the number allowed {s} equals to zero) the bouncer will only let people get in, once some of the people get out. If the initial value of s is 1, it is called a binary semaphore. A mutex is a binary semaphore. The

waiting: int wait(mysem_t * s) {_acquire(&s->mut); while (atomic_load(&s->val) <= 0); atomic_fetch_sub(&s->val, 1); release(&s->mut); return 0;} // wait The signal [ADDED]: int signal(mysem_t * s) { return atomic_fetch_add(&s->val, 1);} // signal While load and fetch_sub are guaranteed to be atomic, we can't guarantee that both lines involving these functions will execute atomically as a block, so we treat this section as its own critical section, and use a mutex to ensure mutual exclusion Linux comes with POSIX Semaphores <semaphore.h> sem_t sem; #define wait(s) sem_wait(s) #define signal(s) sem_post(s) deadlock, deadlock prevention vs. avoidance, banker's algorithm A deadlock it is a state in which each process is waiting for another process (including itself) A deadlock can arise if all of the following four conditions hold simultaneously true. **Mutual exclusion**: At least one resource must be held in a non-sharable mode; only one thread at a time can use the resource. If another thread request that resource, the requesting thread must wait until the resource has been released. **Hold and wait**: A thread must be holding at least one resource and waiting to acquire additional resources that are currently being held by other threads. **No preemption**: Resources cannot be preempted; the resource can only be released voluntarily by the thread holding it. **Circular wait**: A set {T0, T1, ..., Tn} of waiting threads must exist such that T0 is waiting for a resource held by T1, T1 is waiting for a resource held by T2, ..., Tn−1 is waiting for a resource held by Tn, and Tn is waiting for a resource held by T0. (If a resource can be acquired by more than one thread, a cycle in the graph is a necessary but not a sufficient condition for a deadlock) **Deadlock Prevention** imposes restrictions on the overall system to prevent DL from ever occurring, **Deadlock Avoidance** keeps track of the state of a system and take measures to avoid DL on a case-by-case basis. Prevention: To prevent Deadlock to occur, we just need to ensure that at least one of the DL condition is prevented. Disallow Mutual Exclusion - Sharable resources that does not require exclusive access, for example read only files. that several threads can be granted permission to access the file simultaneously. However we cannot prevent DL by denying mutual-exclusion because some resources are intrinsically non-sharable. Disallow Hold and Wait: We can use one protocol that requires each thread to request all its resources before it begins executing. This removes the dynamic nature of requesting resources. Alternatively a protocol that forces a thread to request resources only when it has none, it must release all of its resources before requesting new resources. This leads to low utilization. Disallow preemption: Preemption cannot generally be applied to such resources as mutex locks and semaphores, precisely the types of resources where deadlock occurs most commonly. Disallow Circular Wait: Impose a total ordering theoretically avoids the cycle, developing an ordering/hierarchy does not in itself prevent deadlock, it is up to the app developers to write programs that follow the ordering. Deadlock avoidance can have some overhead. And it has two states, safe and unsafe. The system is in a safe state if it can allocate resources to each thread up to its maximum in some order and still avoid a deadlock. **Resource-Allocation-Graph**, if each resource has only one instance, then we can build this graph to check if adding a request edge forms a cycle. However this is not applicable to a system with multiple instances of each resource, since a cycle its a necessary but not sufficient condition. **Banker's Algorithm** is a deadlock avoidance algorithm. When a new thread enters the system, it must declare a maximum number of instances of each resource that it may need. This number may not exceed the total number of resources in the system. When a user requests a set of resources, the system must determine whether it will leave the system in a safe state. If so, it is allocated, if not, the thread must wait until some other thread releases enough resources. **virtual memory paging** (including hierarchical) 1. Virtual/logical vs Physical a. **Virtual address**: address generated by CPU that will be translated to physical address before CPU use it b. **Virtual address space**: set of all the virtual addresses c. **Physical address**: actual address in computer memory d. **Physical address space**: set of all physical addresses 2. **MMU: memory management** unit a. CPU passes virtual address b. MMU translates to physical address and access physical memory c. Relocation-based MMU d. Memory protection **Post-Exam 2 Material** demand paging and page replacement - FIFO, OPTIMAL, and LRU **Demand paging**: only load pages that are actually going to be used (demanded) into physical memory **Valid-invalid bit scheme**: 1 if page is legal and in memory 0 if it is not valid or is valid but in secondary storage **Page fault**: attempting to access a page marked invalid handling page faults 1. We check an internal table to determine if it was valid or invalid 2. invalid -> terminate the process. If valid, -> page it in 3. find a free frame 4. read the desired page into the newly allocated frame 5. modify the internal table kept with the process and the page table to indicate page is in memory 6. Restart the instruction that was interrupted **Extreme page fault**: every line accesses a page not in memory, therefore every line must wait to have a page brought in **Locality of reference**: Tendency for processes to access memory near where they have accessed recently. **Secondary memory (swap space) effective access time** ((1 - p) * mem. access time) + (p * page fault time) where p is the probability of page fault if p is probability of page fault, 1 - p is the probability that there isn't a page fault **Page replacement**: if there are no free frames, find a page not in use, swap it out for the needed page using a page replacement algorithm. 1. Change the page table 2. write the victim to secondary storage, 3. write in the desired page in the victims frame **Victim frame**: frame killed by page-replacement algorithm **FIFO page replacement**: replace the oldest page; simple to implement, but many page faults **Optimal page replacement**: Replace page that will not be used for longest period of time **LRU Page replacement**: use the recent past as an approximation of the future, replace the page that hasn't been used for the longest period of time, or the least recently used page **Filesystems Mount**: a process by which the os makes files and directories on a medium available for users via access to the file system **Mount point**: the location within the file structure where the file system is to be attached. **Unmount**: a process by which os cuts off user access to the mount point, writes remaining queue of user data, refreshes the file systems metadata, and relinquishes access to the device **File system**: provides efficient and convenient access to the disk by allowing data to be stored, located, and retrieved easily. **I/O control level**: consists of device drivers and interrupt handlers to transfer information between the main memory and the disk system **Device driver**: translator between device and OS inputs are high level commands, output is low level, hardware specific instructions **Basic file system**: Issues generic commands to the appropriate device driver to read and write physical blocks on the disk, also concerned with I/O request scheduling, managing memory buffers and caches **File organization module** - knows about files and their logical blocks, as well as physical blocks. - logical blocks are numbered from 0 to N - includes a free space manager **Logical file system**: manages metadata information, everything but actual data, manages directory structure, contains inodes **Boot control block**: contains info needed by system to boot OS from that volume, first block of a volume **Volume control block**: super block, contains volume details like number of blocks in the volume, size of the blocks, free block count, and free block pointers **Contiguous allocation**: each file occupies a set of contiguous blocks **Linked allocation**: each file is a linked list of blocks**demand paging hardware support** - page table secondary memory (swap space) **operating system case study – linux Basic history** Linus Torvalds created Linux in 1991 for the 80386 processor. It was free and cross developed on Minix. Up until version 3, even numbered releases were stable production releases, odd numbered were development releases i.e. 1.1 and 2.5 are development releases, 1.2 and 1.4 are stable production releases **Completely Fair Scheduler** Handles CPU resource allocation for executing processes. Attempts to

maximize overall CPU utilization and maximize interactive performance. It is pre-emptive, priority based. Priority has 2 values: a real time range from 0-99, and a nice value from -20 to 19 The higher the nice value, the nicer the process is by letting other processes go ahead of it. Threads run as a function of total number of runnable threads. **Target Latency**: an interval of time during which every task runs once **Minimum granularity**: minimum amount of time any thread is allotted **Page Replacement** Uses a modified second-chance FIFO Each frame has a reference bit. If the bit is 1, the frame is given a second chance and the bit is set to 0 If the bit is 0, the frame is chosen as the victim and replaced This degenerates to FIFO if all bits are set. Ext3 VFS; Journaled filesystem; Contains from ext2:; Blocks; Inodes; Block groups; Directories; Bitmaps; Superblocks; **Ext3 New Additions Journal**: circular log that keeps track of changes not yet committed to the file system **Online filesystem growth**: a mounted filesystem can be enlarged **H-tree indexing**: constant depth hashed B-trees that don't need rebalancing **Levels of journaling: Journal** - low risk - both metadata and file contents are written to journal before being written to the main filesystem **Ordered** - medium risk - only metadata is written to the journal, but file contents are guaranteed to be written beforehand **Write back** - high risk - only metadata is journaled

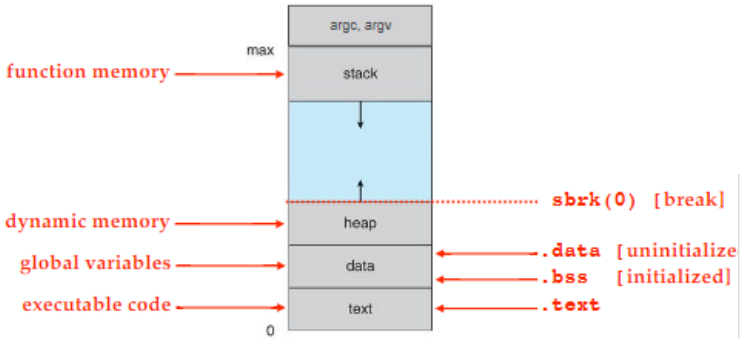
To invoke a system call:

1. Set `%rax` to the system call number (see Full Table below);
2. Set `%rdi`, `%rsi`, `%rdx`, `%r8`, `%r10`, `%r9` (in order) according to the `glibc` prototype given in the manual; then
3. Perform `syscall` instruction.

The order of the parameter registers is different for system calls!

Parameter	Function Call	System Call
1	`%rdi`	`%rdi`
2	`%rsi`	`%rsi`
3	`%rdx`	`%rdx`
4	`%rcx`	`%r8`
5	`%r8`	`%r10`
6	`%r9`	`%r9`

Process Memory



Intrusive Linked List Exa

```
struct list_node {
  struct list_node * next;
};

struct data_item {
  unsigned int x;
  struct list_node list1;
  struct list_node list2;
};
```

You can use `call` and `ret` to transfer control between functions.

Instruction	Example	Pretend it's C	Notes
`call`	`callq foo`	`foo();`	automatic `pushq %rip`
`ret`	`retq`	`return;`	automatic `popq %rip;`

Instruction	Example	Pretend it's C	Notes
`mov`	`movq x, %rax`	`rax = x;`	
`inc`	`incq %rax`	`++rax;`	
`dec`	`decq %rax`	`--rax;`	
`add`	`addq %rbx, %rax`	`rax = rax + rbx;`	
`sub`	`subq %rbx, %rax`	`rax = rax - rbx;`	
`imul`	`mulq %rbx, %rax`	`rax = rax * rbx;`	
`and`	`andq %rbx, %rax`	`rax = rax & rbx;`	
`xor`	`xorq %rbx, %rax`	`rax = rax ^ rbx;`	
`shr`	`shrq $4, %rax`	`rax = rax >> 4;`	unsigned
`shl`	`shlq $5, %rax`	`rax = rax << 5;`	unsigned
`sar`	`shrq $4, %rax`	`rax = rax >> 4;`	signed
`sal`	`shlq $5, %rax`	`rax = rax << 5;`	signed
`imul`	`imul $0x10, %rax`	`rax = rax * 16;`