|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| General Tasklet Information:   * A Tasklet is a data structure used to wrap the handler for a soft-interrupt. * Generally scheduled by a hard interrupt handler and is placed into a list of other tasklets of the same priority * Are used to execute tasks that are not time sensitive – e.g. processing of data provided by a device. * Executed by do\_softIRQ. This is called when do\_IRQ() finishes or when a periodic **daemon** wakes up and checks to see if there are any tasklets that need to be executed. * Tasklet structures are generally declared statically. | | | | | Protection Model:   1. **Kernel –** level 0 | **User –** level 3 2. Idea: higher privileged code will never call less privileged code. And lower privileged code must pass through system calls to request services from kernel. 3. **RPL** is in the segment selector – part of logical address 4. **CPL** is in the current code/data segment registers 5. **DPL** is stored in the quad word GDT entry (segment descriptor) 6. If **(max(CPL, RPL)) > DPL**, then general protection fault. | | | | | | |
| VM Advantages (In order of importance):   1. **Protection -** VM prevents programs from accessing each other’s data 2. **Sharing** - sharing occurs through use of libraries. Library code is placed in physical memory and mapped into both program’s VM 3. **Prevents Memory Fragmentation -** By dividing memory of a program into 4kB pages, the memory of a program does not need to be contiguous in physical memory. 4. **Relocation -** Avoids having to change absolute addresses since the hardware will take care of that for us. 5. VM Disadvantage**:** Storage requirement for the paging structure and the time overhead to perform translations of VA to physical address. | | | | | | |
| Virtual Memory:   * Virtual memory is the insertion of a level of indirection between the memory address space seen by a program and the physical memory space of a system. * The hardware **(MMU or the processor)** is responsible for the translation from VM to physical memory. * X86 uses virtual memory regardless of the privilege level. * Some pages are mapped into a device’s memory instead of physical memory 🡪 **Memory Mapped I/O**   Translation occurs as shown in image below: | | | | | | | | | | Segmentation:   * Segmentation unit converts virtual or logical address to linear address. * Segments are described by the **Global Descriptor Table** and possibly **Local Descriptor Table** * **GDTR** holds the physical address to the GDT and a 16-bit limit (size -1 in bytes) * GDT stores segment descriptors which contain the base address, max offset of segment, and the DPL. * To convert to linear address, you take your VA and add the offset stored in the GDT segment descriptor. * Segment registers select the segment being referenced - CS, SS, DS, ES, FS, GS | |
| Task State Segment:   * Entries in the GDT can describe **TSSs** which hold information pertaining for a particular program. * Important components: **SS0 and ESP0** ---> used when we switch from user mode to kernel mode. * SS0 is the stack segment selector for the kernel * ESP0 is the offset within the segment to get to the start of the kernel stack. These combine to form the virtual address or the logical address. * Other components: ESP0 SS0 … CR3 EIP EFLAGS EAX ECX EDX EBX ESP EBP ESI EDI ES CS SS DS FS GS LDTR IOPB | | | | | | | | | |
|  | |  | | | | | | | | | Page Directory Entry:    Page Table Entry |
| Page Directory/Table Entries:   * Protection: each PTE has a privilege level bit (U) which is required for use in address translation. When U is set, everyone has access. When U is 0, privilege level check is made ---> max(CPL, RPL) < 3 to use translation * Performance: global flag (G). If set, the PT or page is present in all programs VA spaces in the space place. This allows retention of TLB translations when PDBR is changed. * 4kB and 4MB pages have separate TLBs. * Using larger pages reduces the number of TLBs required, but risks fragmentation due to larger amount of contiguous memory being used. * To use 4BM pages, the page size bit (S) needs to be set in the PDE. * PDE ---------------------------------- PTE | Paging:   * Page States: **Present**, **Swapped Out**, **Non-existent** * Present means that a program has access to allocated page * Swapped out means that a program has access to the page but the page has been moved to a **swap disk** * If a program tries to access a page that is non-existent, this will cause a **page fault**. * If a program tries to access a page that has been swapped out, the OS has the option to swap the page back in or send a signal to the program aka Segmentation Fault. * Motivation for having PD and PT is to save space and have **4kB consistent size and alignment** for PD, PT, and pages. * Registers associated with paging:   + **Cr3** – aka PDBR is used to point to the **physical address** of the page directory   + **Cr4** – used to enable 4MB pages ---> set 0x0010 bit   + **Cr0** – used to enable paging on the processor ---> set MSB | | | | | | | | | | |
| Paging  **PLACE YOUR MP3 PAGING ENABLE CODE HERE** | | | | | | | | | | |
| Creating User Processes/Tasks:   * User-level: **fork**, **vfork**, and **clone** system calls * In Kernel: all map to **do\_fork** which calls copy­\_process() to set up the process descriptor and any other kernel data structures needed for child execution. * Creating processes at User-level: other programs have to start it. i.e. shell * First, **fork** is called to create a copy of current program and then **exec** is called which loads the new program and starts it. * Implementation Strategies of fork: **copy-on-fork or copy-on-write** * **Copy-on-fork**: instantly copies the writable portion of the original program’s address space. Address space is instantly disjoint. “Eager” approach. * **Copy-on-write**: Instead of copying data, fork creates a new page directory and creates copies of the page tables which point to the same pages. It also turns off write permission. At first, processes share the same physical address space. When one of the processes tries to write to a shared page, a private copy of the page is made for that process. “Lazy” approach. Example: **DEMAND PAGING, stack and heap are also shared between the child and parent.** * **vfork**: parent blocks while child uses the same address space. After child execs, control of address space returns to parent.   Clone: clone is used to implement threads. This allows multiple threads in a program to run concurrently in a shared memory space. Unlike fork, children created using clone share parts of the execution context with the calling process -----> such as memory and tgid | | | Processes/Tasks:   * Process/Task is a single running instance of a program and linux treats it as a unit of scheduling * **Process:** Each process provides the resources needed to execute a program. A process has a virtual address space, executable code, a unique process identifier, and at least one thread of execution. Each process is started with a single thread, often called the primary thread, but can create additional threads from any of its threads. * **Thread:** A thread is the entity within a process that can be scheduled for execution. All threads of a process share its virtual address space and system resources. In addition, each thread maintains exception handlers, a scheduling priority, thread local storage, a unique thread identifier, and a set of structures the system will use to save the thread context until it is scheduled. The thread context includes the thread's set of machine registers, the kernel stack, a thread environment block, and a user stack in the address space of the thread's process. * **User-level view** * each execution context that can be independently scheduled has its own process descriptor * traditional process id (pid, a field in task structure/process descriptor) * from 1 to 32,767 in Linux, used as task-unique identifier * tgid (thread group id) plays process id role for multithreaded applications (common id for all threads in process) * most processes belong to a thread group consisting of a single member * **Kernel view** * kernel must handle many processes at the same time * keeps two data structures in a single per-process area (8kB) * thread\_info structure (keeps pointer to task structure or process descriptor) * kernel stack * both dynamically allocated * architecture-dependent thread info shares space with kernel stack | | | | | | | | |
| **iret\_to\_user((unsigned long)entry\_point\_address, (unsigned long)USER\_CS, (unsigned long)new\_flags, (unsigned long)new\_esp, (unsigned long)USER\_DS); // pop ret address; iret** | | | | | | | | | | | |
|  | | | | | |  | | | | | |
| Signals:   * Signals are user-level analogue of an IRQs 🡪 asynchronous * Each signal has a default action which can be changed by user prog. * Signals can be **ignored**, **blocked** (masked), or **caught** (caused to execute a program-defined handler function). * SIGKILL and SIGSTOP can neither be ignored nor caught and thus always execute the default actions. * User program generate signals though the use of **sys\_kill** system call.   + This checks for calling programs permissions to generate signals for the targeted pid. Permission is granted only if the caller is owned by the same user as the target, is in the same login session as the target, or is owned by the machine’s super-user. * Send\_sig() and force\_send\_sig() are used to send sigs. Force version primarily used to deliver signals generated by exception. * Images explain how the user level signal handlers work | | | | | | https://lh6.googleusercontent.com/GOUje8VrooYAB4ODxadPWWZrXDiz7D0DtYx2Cy3ZW7s0VdDngDcr0XKhdsXbQlaVxdzxofgGugxAH1CZ9bexqVSJXfyja6XdLqWDZO8eiHRvtwh6GgXvURQ3vt_R2_dLDStnLjAI  Random Coding Syntax Stuff: | | | | | |
| Scheduling:   * **I/O Bound** means the rate at which a process progresses is limited by the speed of the I/O subsystem. A task that processes data from disk, for example, counting the number of lines in a file is likely to be I/O bound. * **CPU Bound** means the rate at which process progresses is limited by the speed of the CPU. A task that performs calculations on a small set of numbers, for example multiplying small matrices, is likely to be CPU bound. * Linux checks for rescheduling after each interrupt, system call, and exception. * Time broken into **epochs** which contains **quantums** * The **swapper process** is the idle process, swap in when there is nothing to do | | | | | | Tasks:   * TASK\_RUNNING: task is executing or waiting to execute; in a run queue on some processor * TASK\_INTERRUPTIBLE: task is sleeping on a semaphore/condition/signal or in a wait queue, can be made runnable by delivery of signal * TASK\_UNINTERRUPTIBLE: task is busy with something that can’t be stopped (e.g. device will stay in unrecoverable state without further task interaction, cannot be made runnable by delivery of signal) * TASK\_STOPPED: task is stopped; not in a queue and must be woken by signal * TASK\_ZOMBIE: task has terminated; task state retained until parent collects exit status information. | | | | | |
| Run Queues:   * Each processor has a **run queue** which has 2 **priority arrays**. * These arrays are lists of tasks of each priority. They are double buffered to implement epochs. * The priority array has 100 real-time priorities and 40 regular ones * Real-time tasks will preempt normal tasks. They can also preempt within themselves since they exist from priority 0 to 99. * An **epoch** can be defined as one run through an entire active array (active run queue) before your switch pointers and have the expired array become active. * Linux Scheduling Policies * SCHED\_FIFO: real-time; retain CPU until preempted or yields * SCHED\_RR: real-time; take turns with tasks at same priority * SCHED\_NORMAL: not real-time; basically round robin without accounting for the static priorities. **Nice** value of task is used to decide its time slice. * Shortest Job First (SJF): a non-preemptive scheduling algorithm. In this algorithm the shortest job is picked form the queue and is run first. After that the next shortest job is picked and so on. * Shortest Remaining Time Next (SRTN): preemptive form of Shortest Job First * Turnaround time: Time finish – Time arrival 🡪 add up all of the time slots that the job has been in the schedulers | | | ../Desktop/Screen%20Shot%202016-12-09%20at%209.19.41%20AM.png  ../Desktop/Screen%20Shot%202016-12-09%20at%209.19.54%20AM.pngTo the right is the **run queue** and below there is an image of the **priority array**. | | | | | | | | |
| Buddy Allocator:   * Buddy system preferred because of 3 reasons: * Contiguous page frames are sometimes really necessary * Example: Buffers assigned to a DMA processor * Leaves page tables unchanged if pages contiguous * Higher page table modification leads to higher ave mem access times due to TLB flushing * Large chunks of contiguous memory can be accessed by the kernel through 4 MB pages. * This reduces TLB misses * Two blocks are considered buddies if: * Both blocks have same size b * They are located in contiguous physical addresses * The physical address of the first page frame of the first block is a multiple of 2\*b\*212 or 2\*b\*size of block * Example: if two block are at order 1 where 2orderrepresents the number of page represented by a block, then we would decide to merge one block with another if the start index of the first block is a multiple of 22. For example, if block id 2 and 6 are free and you want to free block 4, you will decide to merge 4 with 6 since 4 is a multiple of 22 and 2 is not. So basically if divisible by 2order+1 , then current block and next block are buddies. | | | | | | | | | To the left we have the initial sate. Once we have allocated stuff we end up with the image above. | | |
| Slab Allocator:   * Slab allocators seek to reduce internal fragmentation of kernel * Largest alignment factor by slab allocator is 4096, size of a page frame * A slab is a set of one or more contiguous pages of memory set aside by the slab allocator for an individual cache. This memory is further divided into equal segments the size of the object type that the cache is managing. * As an example, assume a file-system driver wishes to create a cache of inodes that it can pull from. Through the **kmem\_cache\_create**() call, the slab allocator will calculate the optimal number of pages (in powers of 2) required for each slab given the inode size and other parameters. A **kmem\_cache\_t** pointer to this new inode cache is returned to the file-system driver. * When the file-system driver needs a new inode, it calls **kmem\_cache\_alloc**() with the **kmem\_cache\_t** pointer. The slab allocator will attempt to find a free inode object within the slabs currently allocated to that cache. If there are no free objects, or no slabs, then the slab allocator will grow the cache by fetching a new slab from the free page memory and returning an inode object from that. * When the file-system driver is finished with the inode object, it calls **kmem\_cache\_free**() to release the inode. The slab allocator will then mark that object within the slab as free and available. * If all objects within a slab are free, the pages that make up the slab are available to be returned to the free page pool if memory becomes tight. If more inodes are required at a later time, the slab allocator will re-grow the cache by fetching more slabs from free page memory. All of this is completely transparent to the file-system driver. | | | | | | | | |  | | |
| Slab Continued:   * **slabs\_full**, **slabs\_partial**, and **slabs\_free** are lists of slabs associated with this cache * There are times when a kernel module or driver needs to allocate memory for an object that doesn't fit one of the uniform types of the other caches, for example string buffers, one-off structures, temporary storage, etc. For those instances drivers and kernel modules use the **kmalloc**() and **kfree**() routines. **The Linux kernel ties these calls into the slab allocator too**. * **On initialization, the kernel asks the slab allocator to create** **several caches of varying sizes for this purpose.** Caches for generic objects of 32, 64, 128, 256, all the way to 131072 bytes are created for both the GFP\_NORMAL and GFP\_DMA zones of memory. * When a kernel module or driver needs memory, the **cache\_sizes** array is searched to find the cache with the size appropriate to fit the requested object. For example, if a driver requests 166 bytes of **GFP\_NORMAL** memory through **kmalloc**(), an object from the 256 byte cache would be returned. * When **kfree**() is called to release the object, the page the object resides in is calculated. Then the page struct for that page is referenced from **mem\_map** (which was set up to point to our **kmem\_cache\_t** and **slab\_t** pointers when the slab was allocated). Since we now have the slab and cache for the object, we can release it with **\_\_kmem\_cache\_free**(). | | | | | | | | kmalloc vs vmalloc   * Kmalloc is physically and virtually contagious * Due to this, this can fail sometimes if the memory is too fragmented * Vmalloc modifies page table entries to map physically fragmented memory to virtual memory. Slower than kmalloc. | | | |
| Wait Queues:  while (1) {  /\* add self to wait queue \*/  prepare\_to\_wait(&readQ, &wait, TASK\_UNINTERRUPTIBLE);  if (size > 0) break;  /\* go to sleep (potentially) \*/  schedule();  }   * Doubly linked list of tasks waiting for some event * Inside while(1) because of spurious wakeups: when a process sleeping for a condition to be active is woken up to a false alarm, the programmer needs to check if it’s a false-positive after all. * Consider: * You put a job on a queue. * You signal the condition variable, waking thread A. * You put a job on a queue. * You signal the condition variable, waking thread B. * Thread A gets scheduled, does the first job. * Thread A finds the queue non-empty and does the second job. * Thread B gets scheduled, having been woken, but finds the queue still empty. | | | | Linux Code for Wait Queues:  [178](http://lxr.free-electrons.com/source/include/linux/wait.h?v=2.6.32#L178) #define [\_\_wait\_event](http://lxr.free-electrons.com/ident?v=2.6.32;i=__wait_event)([wq](http://lxr.free-electrons.com/ident?v=2.6.32;i=wq), condition)        \ [179](http://lxr.free-electrons.com/source/include/linux/wait.h?v=2.6.32#L179) do {                                                                     \ [180](http://lxr.free-electrons.com/source/include/linux/wait.h?v=2.6.32#L180)        [DEFINE\_WAIT](http://lxr.free-electrons.com/ident?v=2.6.32;i=DEFINE_WAIT)(\_\_wait);                          \ [181](http://lxr.free-electrons.com/source/include/linux/wait.h?v=2.6.32#L181)                                                                          \ [182](http://lxr.free-electrons.com/source/include/linux/wait.h?v=2.6.32#L182)         for (;;) {                                                      \ [183](http://lxr.free-electrons.com/source/include/linux/wait.h?v=2.6.32#L183)                [prepare\_to\_wait](http://lxr.free-electrons.com/ident?v=2.6.32;i=prepare_to_wait)(&[wq](http://lxr.free-electrons.com/ident?v=2.6.32;i=wq), &\_\_wait, [TASK\_UNINTERRUPTIBLE](http://lxr.free-electrons.com/ident?v=2.6.32;i=TASK_UNINTERRUPTIBLE));    \ [184](http://lxr.free-electrons.com/source/include/linux/wait.h?v=2.6.32#L184)                 if (condition)                                      \ [185](http://lxr.free-electrons.com/source/include/linux/wait.h?v=2.6.32#L185)                         break;                                          \ [186](http://lxr.free-electrons.com/source/include/linux/wait.h?v=2.6.32#L186)                [schedule](http://lxr.free-electrons.com/ident?v=2.6.32;i=schedule)();                                         \ [187](http://lxr.free-electrons.com/source/include/linux/wait.h?v=2.6.32#L187)         }                                                                \ [188](http://lxr.free-electrons.com/source/include/linux/wait.h?v=2.6.32#L188)        [finish\_wait](http://lxr.free-electrons.com/ident?v=2.6.32;i=finish_wait)(&[wq](http://lxr.free-electrons.com/ident?v=2.6.32;i=wq), &\_\_wait);                 \ [189](http://lxr.free-electrons.com/source/include/linux/wait.h?v=2.6.32#L189) } while (0)   |  |  | | --- | --- | | void wake\_up(wait queue head t\* wq); | Wake up all tasks waiting in a wait queue. | | void wake\_up\_interruptible           (wait queue head t\* wq); | Wake up all interruptible tasks  (with task state **TASK INTERRUPTIBLE**) waiting in a wait queue. | | | | | | | | |
| Wait Queues APIs:   * **wait\_event**("queue","condition") : The task will keep waiting on the queue as long as the condition does not become true.If put to sleep using this call, the task can not be interrupted. * **wait\_event\_interruptible**("queue","condition") : similar to wait\_event, but it can be interrupted by other signals too. It is always preferable to use this interruptible way of sleeping so that the task can be stopped in case the condition never becomes true. * **wait\_event\_timeout**("queue","condition","timeout") : The task will sleep on the queue until the condition becomes true or the timeout mentioned expires, which ever occurs first. The timeout is expressed in jiffies. Task can not be interrupted before the timeout if the condition does not become true. * **wait\_event\_interruptible\_timeout**("queue","condition","timeout") : Similar to wait\_event\_timeout but it can be interrupted. * **wake\_up**(queue) : In case the task has been put to non interruptible sleep. * **wake\_up\_interruptible** (queue) : In case the task has been put to an interruptible sleep. | | | | | | | https://lh6.googleusercontent.com/akUnYFyU4cJm87nDLoYwdFL5LZm5qMUbMpQDLywanBlEKNpgELTp7QkgGf70hbJDmu4KI70tk_wzWhvMygEOf1FepzxHi-0O3aSo7lo-NJBv5S3kjdfHlrWvFxGhjWvMGeqZDTCp https://lh3.googleusercontent.com/GgRPdRwoP1FOa7N39x8ZnzDG-z3hdbVSNwCJ0y6qJrPUmdG6cxo0_YLaEUxPCdiqsWUTBehAr83t3wJY5LtY5wQlZ7b-UPrwzKTHhNHnYV4iRA4tU0ZpLj8Ft5JxTwzbl7vusvo9 | | | | |
| https://lh5.googleusercontent.com/N4pwHxNAPNHBZii4W_hrhpbnKU4QzquIN_t2f1gpQhGcX7Mxime6ky7Gkqggfqi-iggECdtbj8uLh1pHCL2rjydRdKnUeIuQBUibSREZiztxVhsl32jN54n70vft677k3U2uLzpT | | | | | | | | | | | |
| I-Mail:   * **DESIGN CONCEPTS STEPS** below outline the process of designing a device driver : * Contemplate security. Security is hard or impossible to add in correctly as an afterthought. * Write descriptions of all operations in terms of the visible interface (the file operations structure). * Design the data structures. * Pick a locking strategy: organize data into sets protected by locks. * Determine what types of locks should be used and where they are stored, then define a lock ordering. * Identify blocking conditions and events that cause blocked tasks to wake up. * Consider dynamic allocation issues and hazards. * Write the code. * Write subfunctions and synchronization rules for them (in comments). * Return to Step 3 or Step 4 if Step 7 or Step 8 fails. * Write unit tests for the driver. | | | | | | | | | | | |
| I-Mail Read Code: | | | | | | | | | | | |