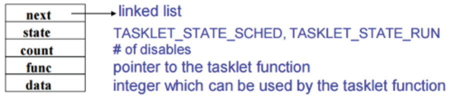
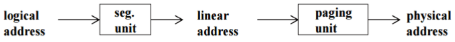
General Tasklet Information

•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.


next
state
count
func
data
linked list
TASKLET_STATE_SCHED, TASKLET_STATE_RUN
# of disables
pointer to the tasklet function
integer which can be used by the tasklet function

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



VM Advantages

•Protection - VM prevents programs from accessing each other's data
•Sharing - sharing occurs through use of libraries. Library code is placed in physical memory and mapped into both program's VM
•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.
•Relocation - Avoids having to change absolute addresses since the hardware will take care of that for us.

VM Disadvantage

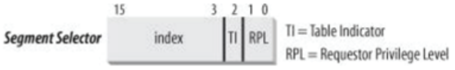
•Storage requirement for the paging structure and the time overhead to perform translations of VA to physical address.
•Individual programs face extra latency when they access a page for the first time.
•Memory management with page replacement algorithms becomes slightly more complex.
•Possible security risks, including vulnerability to timing attacks
•When updating page-directory and page-table entries, the processor uses locked cycles to set the accessed and dirty flag in the page-directory and page-table entries.

Protection Model

•Kernel – level 0 | User – level 3
•Idea: higher privileged code will never call less privileged code. And lower privileged code must pass through system calls to request services from kernel.
•RPL is in the segment selector – part of logical address
•CPL is in the current code/data segment registers
•DPL is stored in the quad word GDT entry (segment descriptor)
•If (max(CPL, RPL)) > DPL, then general protection fault.

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/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.

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

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

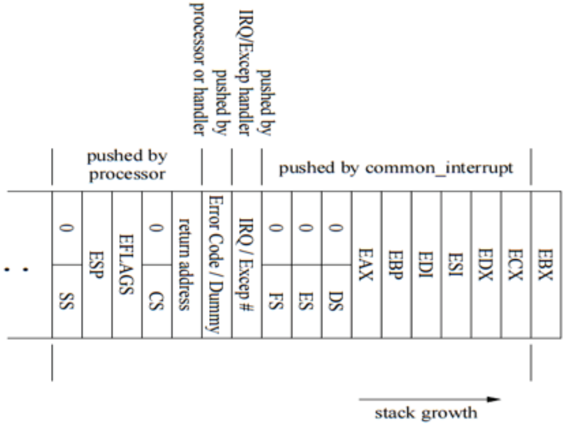
G - Ignored	G - Global
S - Page Size (0 for 4kb)	D - Dirty
A - Accessed	A - Accessed
D - Cache Disabled	C - Cache Disabled
W - Write Through	W - Write Through
U - User\Supervisor	U - User\Supervisor
R - Read\Write	R - Read\Write
P - Present	P - Present

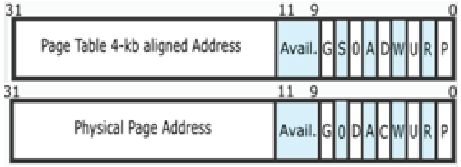
Processes/Tasks

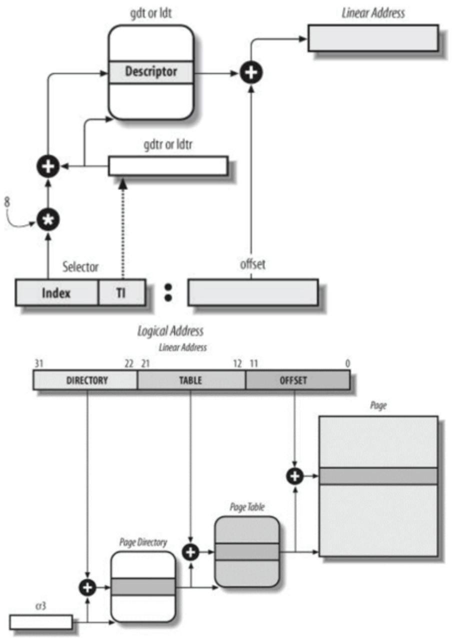
•Process/Task is a single running instance of a program and linux treats it as a unit of scheduling
•Process: Each process provides the resources needed to execute a program. A process has a virtual address space, executable code, a unique process identifier, and at least one thread of execution. Each 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

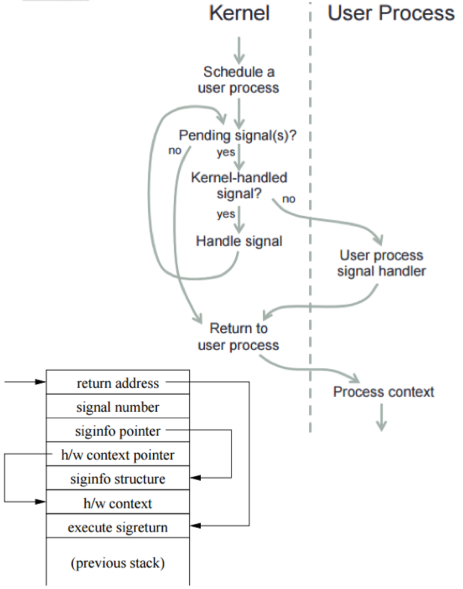
Signals

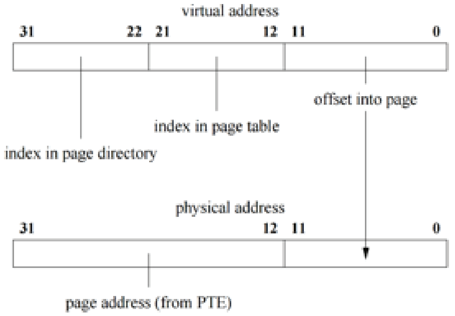
•Signals are user-level analogue of an IRQs -> asynchronous
•Each signal has a default action which can be changed by user prog.
•Signals can be ignored, blocked (masked), or caught (caused to execute a program-defined handler function).
•SIGKILL and SIGSTOP can neither be ignored nor caught and thus 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 on the right explain how the user level signal handlers work











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

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

Buddy Allocator

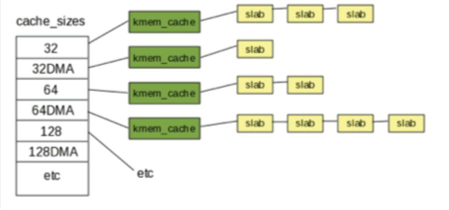
- Buddy system preferred because of 3 reasons:
- Contiguous page frames are sometimes really necessary
- Example: Buffers assigned to a DMA processor
- Leaves page tables unchanged if pages contiguous
- Higher page table modification leads to higher ave mem access times due to TLB flushing
- Large chunks 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 2order represents 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.

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.

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.
- 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.

```
#define __wait_event(wq, condition)       \
do {                                       \
       DEFINE_WAIT(__wait);                \
                                           \
       for (;;) {                          \
              prepare_to_wait(&wq, &__wait, TASK_UNINTERRUPTIBLE);   \
              if (condition)               \
                     break;                \
              schedule();                  \
       }                                   \
       finish_wait(&wq, &__wait);          \
} while (0)
```

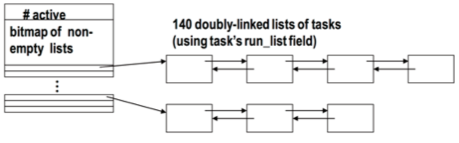
- wait_event("queue","condition") : The task will keep waiting on the queue as long as the condition does not become true.If put to sleep using this call, the task can not be interrupted.
- wait_event_interruptible("queue","condition") : similar to wait_event, but it can be interrupted by other signals 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.

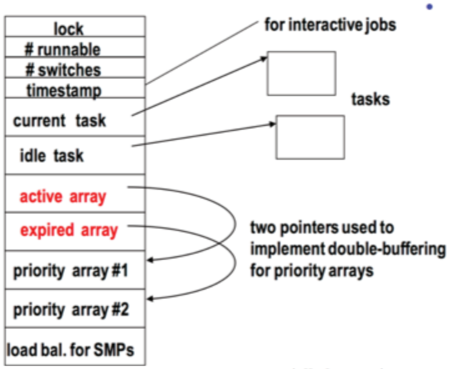
I-Mail

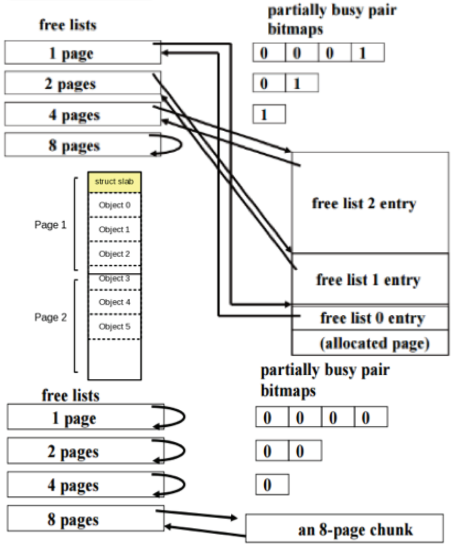
- DESIGN CONCEPTS STEPS below outline the process of designing a device driver :
- Contemplate security. Security is hard or impossible to add in correctly as an afterthought.
- Write descriptions of all operations in terms of the visible interface (the file operations structure).
- Design the data structures.
- 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.

Kernel Role

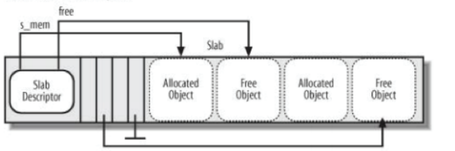
- Process Management - process creation, scheduling
- Memory Management - virtual memory
- File system - almost everything can be treated as file
- Device control - almost every operation externally maps to physical device, perform by code specific to the device
- Networking


# active
bitmap of non-empty lists
140 doubly-linked lists of tasks (using task's run_list field)

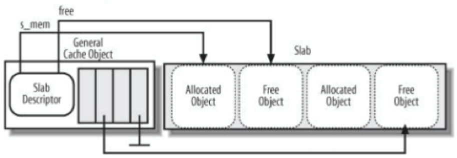




Slab with Internal Descriptors



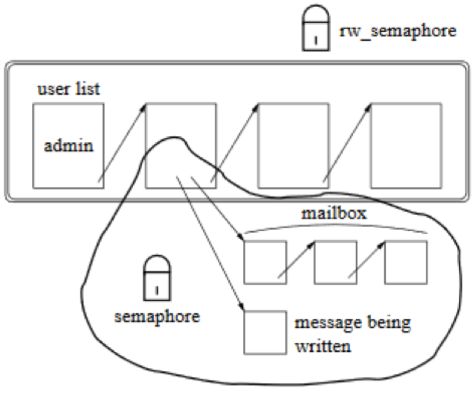
Slab with External Descriptors

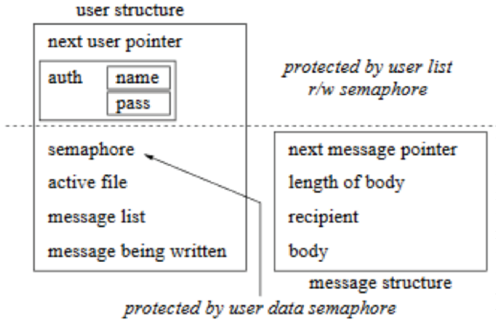


Syntax

- `pde_t page_directory[PD_ENTRIES] __attribute__((aligned(4096));`
- `pte_t page_table[PT_ENTRIES] __attribute__((aligned(4096));`

Operation	User List R/W Semaphore	User Data Semaphore	Comments
read message	no	yes	
write message	no	yes	
wait for message	no	yes	
send message/fsync	(no) read	sender recipient	for removing outgoing message for message delivery
end session/release	(no) *	yes *	for removing associated file from user data see “send message” if message delivery is necessary
authenticate	read	yes	only need user data semaphore if successful
set password	write	no	only changes authentication data—these are protected by user list R/W semaphore
start writing message	read *	yes *	only need user data semaphore if recipient exists see “send message” if message delivery is necessary
delete first message	no	yes	
add new user	write	no	no need for new user’s user data semaphore: no task can authenticate while we hold write lock on user list
delete user	write	yes	user data semaphore necessary to ensure that deletion synchronizes with all other operations on the user





Malloc Strategy

- a few small items - kmalloc contigious
- a lot of items, repeatedly - slab cache
- a big, physically contiguous region - free pages
- a big area of virtual memory - vmalloc (not necessarily physically contiguous)

Managing a private cache of objects (slab cache)

- frequent allocations/deallocations
- one cache per item type
- physical memory is contiguous
- Protocol
  - Creation returns a page handle
  - Use handle to allocate/deallocate objects or to destroy the slab cache when done

Devices

- Kernel interacts with I/O devices by means of device drivers
- operate at the kernel level
- include data structures and functions that control one or more devises, e.g., keyboard, monitors, network interfaces
- Advantages:
  - Make a HW device to respound to well-defined interface
  - device code can be encapsulated in a module, hide details
  - easy addition of new devices, no need to know the kernel source code
  - dynamic load/unload of device drivers at runtime
  - User activities performed by a set of standard calls that are independent of the specific driver
  - Driver can build seperately from the rest of the kernel
- Block devices (underlies file systems)
  - data accessible only in blocks of fixed size, with size determined by device
  - can be addressed randomly
  - transfers to/from device are usually buffered (to) and cached (from) for performance
  - examples: disks, CD ROM, DVD
- Character device:
  - Almost everything else, except network cards
  - Network cards are not directly associated with device files : keyboard, terminal, printers
- Devices
  - identified by major and minor numbers (traditionally both 8-bit)
  - major number is device type (e.g., specific model of disk, not just “disk”)
  - minor number is instance number (if driver allows you to have more than one of a given model attached to your computer)
- File Operations:
  - flush is called each time a file is closed, release is called after the last close, lock call for file locking, readv and writev for read and write vector operations

Scheduling

- General strategy:break time into slices, allow interactive jobs to preempt the current job based on interrupts
- Time broken into epochs: each task given a quantum of time (in ticks of 10 milliseconds), run until no runnable task has time left, then start a new epoch
- Real-time jobs: always given priority over non-real-time jobs, prioritized amongst themselves
- Interactive jobs handled with heuristics • heuristic estimate job interactiveness •an interactive job can continue to run after running out of time / takes turns with other interactive jobs •philosophy is that they don’t usually use up quantum •heuristic ensures that job can’t use lots of CPU and still be “interactive”
- 100 real time priorities • 40 regular priorities • 1 list per priority and a bitmap to make it fast
- Real time r-r tasks take turns by placing themselves at the end of list of their priority
- If time slice expires, take it out of run Q, mark it as need to be removed from processor, give it a new time slice, normal tasks go into expired Q, interactive job put back at end (make sure expired tasks are not starved)
- If the average quantum duration is too short, the system overhead caused by process switches becomes excessively high. For instance, suppose that a process switch requires 5 milliseconds; if the quantum is also set to 5 milliseconds, then at least 50 percent of the CPU cycles will be dedicated to process switching.[*]
- If the average quantum duration is too long, processes no longer appear to be executed concurrently. For instance, let’s suppose that the quantum is set to five seconds; each runnable process makes progress for about five seconds, but then it stops for a very long time (typically, five seconds times the number of runnable processes).

Scheduling Continue

- If no tasks are runnable, run the idle task, reset forced expiration of interactive tasks
- If nothing is left in the active run Q, the epoch is over, swap the priority arrays, and reset forced expiration of interactive tasks
- Find the next task to run: Find first bit selects the priority, then take the first task at that priority

```
/* * Imail_read
 * DESCRIPTION: Implements the read system call for Imail. Data are
 * read from the first message in the user’s list (only).
 * Copies data to buffer and advances file pointer.
 * INPUTS: f -- the file structure used for reading
 * buf -- the user’s buffer
 * len -- the length of the buffer
 * offp -- pointer to the file offset
 * OUTPUTS: buf -- buffer with data copied from first message
 * offp -- updated file pointer
 * RETURN VALUE: number of bytes read on success
 * -EPERM if user not authenticated or deleted after
 * authentication
 * -EAGAIN when no message available (non-blocking I/O only)
 * -EINTR when signal received
 * -EFAULT if buffer is bad
 * SIDE EFFECTS: advances file pointer; may sleep
 * SYNCHRONIZATION: Obtains the associated user’s user data semaphore.
 * May sleep on user data’s read_queue waiting for a
 * message to arrive.
 */
static ssize_t
Imail_read (struct file* f, char* buf, size_t len, loff_t* offp) {
  Imail_user_data_t* udata; /* user data structure */
  Imail_msg_t* msg; /* message to be read */
  ssize_t n_read; /* bytes read */
  loff_t off; /* offset */
  /* If user has not authenticated yet, permission is denied. */
  if (NULL == (udata = f->private_data)) return -EPERM;
  /* Loop to wait for data to be available for reading. */
  while (1) {
    /* Start by obtaining user data lock and checking for user deletion. */
    down (&udata->sem);
    if (NULL == udata->active_file) {
      up (&udata->sem);
      return -EPERM;
    }
    /* Have data to read? Break out of this loop. */
    if (NULL != (msg = udata->msg_list)) break;
    /* * No data yet. Release semaphore and either return failure (for
     * non-blocking I/O) or wait for a message or user deletion. */
    up (&udata->sem);
    if (0 != (f->f_flags & O_NONBLOCK))
      return -EAGAIN;
    if (wait_event_interruptible
        (udata->read_queue,
        (NULL == udata->active_file || NULL != udata->msg_list)))
      return -ERESTARTSYS; /* got a signal */
    /* * ERESTARTSYS prevents use of signals to break out of system calls.
     * The semantics are taken from BSD. To change this property, use
     * sigaction and turn off SA_RESTART for the signal used to kick
     * system calls out of blocking.
     */
  }

  /** We have an undeleted user, a message to read, and a semaphore to
   * protect us. Note that the default kernel seek functions do not
   * allow file offsets < 0.
   */
  n_read = 0; off = *offp;
  if (msg->length > off) {
    /* There are bytes left to read--read as many as fit in buffer. */
    if (len < (n_read = msg->length - off))
      n_read = len;
    /* Copy the bytes into the user’s buffer. */
    if (copy_to_user (buf, msg->body + off, n_read))
      n_read = -EFAULT;
    else
      (*offp) = off + n_read;
  }
  /** Release the semaphore and return the number of bytes read
   * (or error number).
   */
  up (&udata->sem);
  return n_read;
}
```

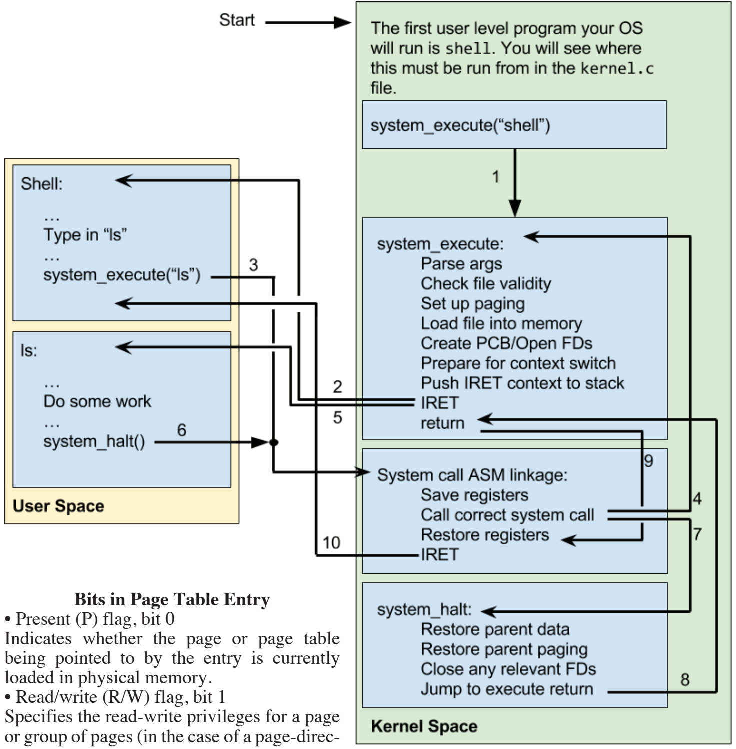
```
system_call_wrapper:
    cmpl $1, %eax
    jl fail
    cmpl $10, %eax
jg fail
pushal (except %eax)
    pushl %edx
    pushl %ecx
pushl %ebx
call *system_call_jump_table(, %eax, 4)
addl $12, %esp
popal (except %eax)
    iret
fail:
    movl $-1, %eax
    iret
system_call_jump_table:
    .long 0, halt, execute, read, write,
open, close, getargs, vidmap, set_han-
dler, sigreturn
```

Signal delivery

Signals can be delivered to the program currently running on the processor whenever the system returns from an interrupt, exception, or system call.

Signal generation inside the kernel typically occurs in response to an interrupt or exception, but can also happen when another program performs a system call other than sys kill. For example, one program may send data to another program using sys write; if the receiving program has requested signals when data are ready on the file descriptor from which it will read those data, a signal is generated.

So when are the forcing functions useful? They are used primarily to deliver signals generated by exceptions. A program that generates an exception cannot make forward progress, since the processor does not know how to proceed. For example, the processor cannot simply keep executing a program that contains an illegal instruction, as the processor cannot execute the illegal instruction. In this case, the operating system can either terminate the program or make a last-ditch effort to get its attention, which is the purpose of the signal forcing functions.



Bits in Page Table Entry

- Present (P) flag, bit 0

Indicates whether the page or page table being pointed to by the entry is currently loaded in physical memory.

- Read/write (R/W) flag, bit 1

Specifies the read-write privileges for a page or group of pages (in the case of a page-directory entry that points to a page table).

- User/supervisor (U/S) flag, bit 2

Specifies the user-supervisor privileges for a page or group of pages (in the case of a page-directory entry that points to a page table).

- Accessed (A) flag, bit 5

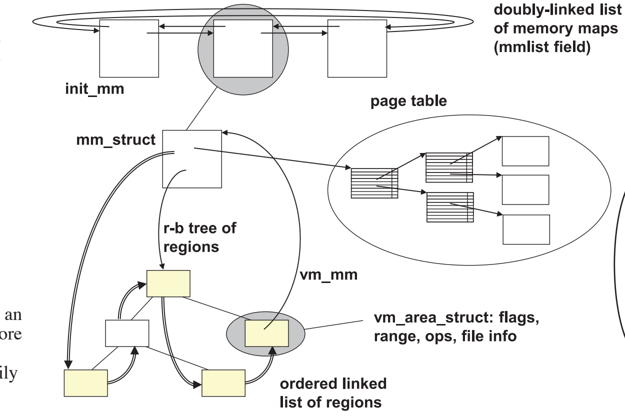
Indicates whether a page or page table has been accessed (read from or written to) when set

- Dirty (D) flag, bit 6

Indicates whether a page has been written to when set.

Default actions for signals

- ignoring the signal
- terminating the program
- terminating the program and dumping an image of the program's memory—called a core file—to disk
- stopping the program's execution temporarily
- letting the program continue execution.

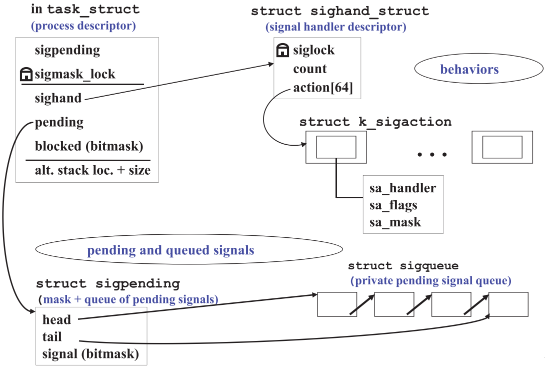

doubly-linked list of memory maps (mmlist field)
init_mm
mm_struct
page table
r-b tree of regions
vm_mm
vm_area_struct: flags, range, ops, file info
ordered linked list of regions

IMail Part 3

- IMail administrator as a user with privileges
  - allows sysadmin to handof Imail rights
  - without giving away other rights
- Use rwsem for the user list because
  - lots of reads, few writes
  - want concurrent authorization and management
- Use sem for user data because
  - lots of reads and writes
  - typically by one user process
- What can block?
  - read - wait for new msg • poll - wait for new msg
  - write - not in imail
- When are readers or pollers awoken?
  - new msg delivered / user is deleted
- What if user is using Imail when the user data is deleted?
  - On deletion, remove from user list, but don't free the structure.
  - Give ownership to file structure
  - it can delete atomically on release call
  - signal by changing active_file to NULL
- How do we know that user doesn't authorize after we check?
  - Auth needs read lock, but delete user op holds write lock on user list.

Signal Part 2

- Similarities:
  - Asnchronous • Not queued • Can be ignored, blocked, or caught • NMIs: SIGKILL, SIGSTOP • handler can be changed from default • Only data given to handler is signal # • Signal is by default blocked while handler executes
- Difference:
  - Generated by software: kernel or program • No device associated with signal; only software with permission can send a signal, signle permission allows any signal to be sent
- POSIX model:
  - new real-time signals are queued • can be sent to threads or processes • siginfo structure contains additional information about the signal
- What's the point of force signal? Why allow a program to block an asynchronous event and then override the block?
  - Used to deliver exceptions
  - Instead of just letting the program hang, try to do something slightly more useful
- What would happen if user program blocked signal from an exception?
  - Program can't execute next instruction • Kernel can't deliver signal -> deadlock
- Who can send?
  - Sysadmin • process with same user id • process with same login session can send SIGCONT
- When are signals delivered?
  - check sigpending when returing from any interrupt, exception, sys call • only deliver to currently executing process
- Action depend on current sigaction. Defaults handled by kernel, handlers are executed by • add a new stack frame to the user stack • transferring machine state context to the user stack • returning to user level
- Why copy hw context from kernel stack? • Avoid kernel stack overflow • Allow user to modify context • Switch threads in signal handler by swapping context with that on stack
- Copy of sigreturn: avoid stack execution, which is pathway for bufferr overrun attack
- sigreturn: • dump stack frame • check if signal was delivered during syscall 1. If so, check whether system call should be restarted 1.1If not returns EINTR, 1.2 If so, sets EAX to previous value and changes PC to reexecute INT x80 2. If not, simply continue the process
- What happen if sigreturn is not called? • no state left on kernel stack, so has no impact on kernel
- can only affect user level stack for that process



IMail Part 2

- release function is called after all files have been closed and all system calls using the file have returned. The release function is thus atomic with
- respect to all other operations on the file.
- I-mail must thus provide this routine and must remove the link from the
- user data structure to the file whenever release is called
- obtain the user list R/W semaphore first, otherwise may deadlock
- cannot sleep if holding a semaphore, otherwise may deadlock
- user data semaphores are not ordered with respect to one another
- delivery must hold both locks
- I-mail first checks the recipient, then atomically swaps the old message with a new one under the protection of the user data semaphore
- Putting a task to sleep while waiting for a device is fairly common, but the steps necessary to avoid race conditions in this code are somewhat error-prone.
- freeing the user data as part of deletion must wait for all outstanding operations on the user data to finish, including half-completed operations by sleeping tasks.
- wait queue head t handle the tasks waiting on an empty mailbox, a spin lock embedded in the head, so I-mail can make use of this queue without obtaining either type of I-mail semaphore.
- The pseudo-code below illustrates the general protocol for using a wait queue to put a task to sleep:

1. Release all locks.
2. Check sleeping conditions without locks. (must be atomically safe)
3. Go to sleep by calling wait event interruptible.
4. When awoken, reacquire necessary locks.
5. Recheck all validity requirements.
6. If the sleeping condition still holds, it was a false alarm: start this process over.

- Delete a user:
  - The first step is to remove the user from the user list, which prevents tasks from authenticating as the deleted user.
  - If the user is not active at that point, they can be safely deleted and the structure freed
  - If the user is active, we defer the deallocation until all user activity finishes.
  - user deletion leaves the pointer in the file structure unchanged
  - The user data structure's pointer to the file structure is under the protection of the user data semaphore, so user deletion can safely change it to NULL
  - kfree?no. We can't guarantee that such a task is the last, and we must wait for the last before deallocating the structure. However, that guarantee is provided by release.

```
void init_paging(void) {
  unsigned int i, j;
  /* Flush page directory */
  for (i = 0; i < PAGE_DIRECTORY_SIZE; i++) {
    page_directory[i] = 0x00000000;
  }
  /* Flush page table 0 */
  for (j = 0; j < PAGE_TABLE_SIZE; j++) {
    page_table_0[j] = 0x00000000;
  }
  /* Points to the page_table_0 */
  page_directory[0] = ((unsigned long)(page_table_0)
& PTBA_MASK) | PRESENT_MASK | R_W_MASK;
  /* Points to the start of Video memory */
  page_table_0[VIDEO_VIRTUAL] = VIDEO_START |
R_W_MASK | PRESENT_MASK;
  /*3 terminal memory*/
  page_table_0[TERMINAL1_VIRTUAL] = TERMINAL1_START
| R_W_MASK | PRESENT_MASK;
  page_table_0[TERMINAL2_VIRTUAL] = TERMINAL2_START
| R_W_MASK | PRESENT_MASK;
  page_table_0[TERMINAL3_VIRTUAL] = TERMINAL3_START
| R_W_MASK | PRESENT_MASK;
  /* Kernel 4MB page, present, R/W on */
  page_directory[1] = KERNEL_START | PAGE_SIZE_MASK
| R_W_MASK | PRESENT_MASK;
  /* Set CR4 Flags */
  clear_in_cr4(CR4_PAE_FLAG);/*PAE set to 0 */
  set_in_cr4(CR4_PSE_FLAG); /* PSE set to 1 */
  /* Set CR3 */
  /* This instruction flushed the tlb */
  write_cr3((unsigned long)page_directory);
  /* Set CR0 bit 31*/
  /* Assert Page Table is turned on */
  set_in_cr0(CR0_PG_FLAG);
}
```