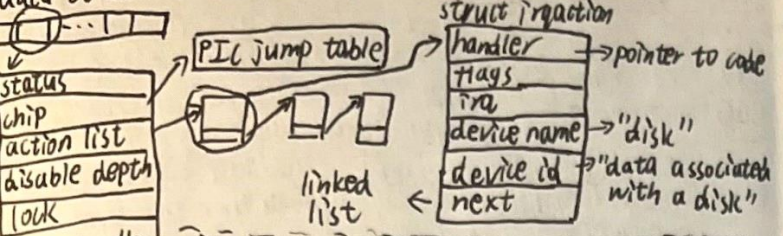
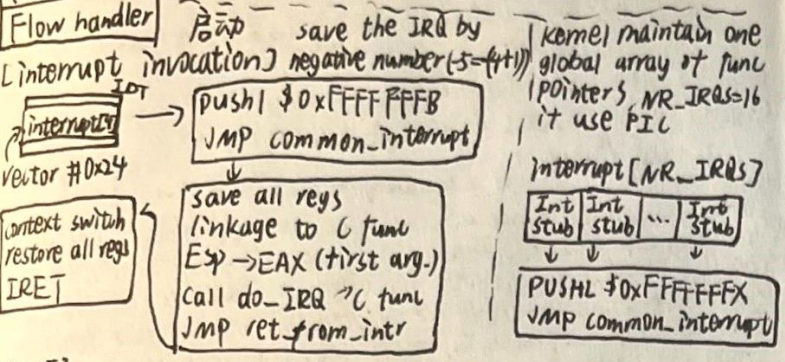
1. Basic of Linux interrupt → 记录interrupt vector数据

data structure: IRQ descriptor array (irq_desc)



启动 save the IRQ by negative number (-5=~(4+1))

[interrupt invocation]

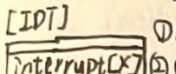


kernel maintain one global array of func pointers, NR_IRQS=16

it use PIC

interrupt[NR_IRQS]

[IDT]



① hardware interrupt: 0x20~0x2F: use interrupt gate

② exception: TRAP gate不清IF! 屏蔽中断, clear IF

③ initialize: 先用lidt指令把idtr reg指向IDT, 再用setup_idt() 设256个 entries with NULL ptr

2. 具体实现:

① init_IRQ(): 将IRQ descriptor的status设为IRQ_DISABLED, update IDT by replacing NULL interrupt gates with new one, 其为在interrupt[NR_IRQS] array中找到的地址.

② install an interrupt handler:

int request_irq(...)

I. sanity check

II. dynamically allocate (kmalloc) "action" for linked list

III. 把"action"加到linked list中用setup_irq(irq, action), 失败用kfree释放.

int setup_irq(...)

I. 计算对irq_desc位置, 用irqsave开critical section, 得action list head

II. 若head不为NULL, 且allow chaining (所有action都同意): walk to the end of list and append

III. 若head为NULL, 即list为空, 调用PIC startup func, critical section结束

IV. create a new directory: register_irq_proc(irq), add handler subdirectory: register_handler_proc(irq, new);

③ uninstalling an interrupt handler

void free_irq(irq, dev_id)

I. 计算对应irq_desc位置, 用irqsave开critical section

II. check the action list of that irq_desc找匹配dev_id, 中remove并从链表

III. 若为该action list唯一action: turn on software disablement 并调用desc→chip→shutdown(irq)

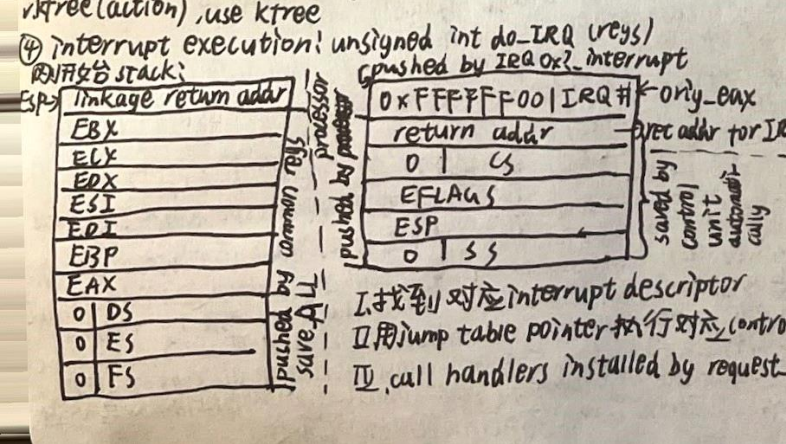
IV. remove handler subdirectory: unregister_handler_proc(irq, action)

synchronize_irq(irq); 需等其它程序完成这个irq

kfree(action), use kfree

④ interrupt execution: unsigned int do_IRQ (regs)

用内核stack:



I. 找到对应interrupt descriptor

II. 用jump table pointer执行对应controller

III. call handlers installed by request_irq

3. why tunnel all IRQs into one piece of code insted of one function per IRQ? Kernel size versus speed trade off:

① the function is not entirely trivial, making a large number of copies can bloat the kernel. ② APIC有256 vectors.

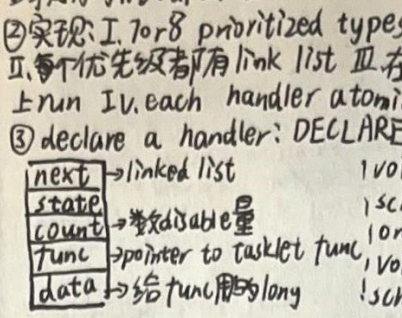
4. software interrupt: 用tasklet数据结构实现

① 执行时机: after a hard interrupt completes; periodically by a daemon in kernel

② 实现: I. 10 or 8 prioritized types, 有high和low

II. 每个优先级都有link list III. 在interrupt scheduled的processor上run IV. each handler atomic to itself (only)

③ declare a handler: DECLARE_TASKLET(name, func, data);



void tasklet_schedule(...)

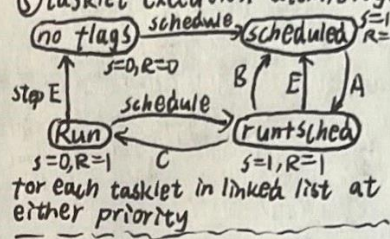
schedules tasklet at low priority on the excuting processor

void tasklet_hi_schedule(...)

schedules at high priority

④ do_softing: I. check per-processor bit vector of pending priorities II 执行action for each priority [softing_vec] III 其中tasklet_action遍历链表, tasklet_hi_action遍历高优先级链表. IV. 重复10次或直到无~~softing~~ softings are raised.

⑤ tasklet execution atomicity:



for each tasklet in linked list at either priority

A. set TASKLET_STATE_RUN atomically stop if already set

B. check if tasklet is software disabled (看count), 是的话清空TASKLET_STATE_RUN, 把tasklet放到此优先级链表, 并set pending bit for this priority, daemon会在这后try again.

C. clear TASKLET_STATE_SCHED

D. 执行 handler

E. 清空TASKLET_STATE_RUN

S: TASKLET_STATE_SCHED目的: ① a tasklet has been scheduled but yet to excute. ② a tasklet is only scheduled on one CPU at one time

R: TASKLET_STATE_RUN: serialize execution of the tasklet on scheled processor

二、virtual memory

1. virtual memory: indirection between logical address memory address seen by software and those used by hardware → physical address

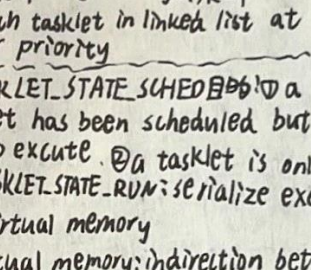
[why use virtual memory]:

① protection: one program can't accidentally or deliberately destory another's data. the memory is simply not accessible.

② more effective sharing: 多个程序sharing library code可以share a single copy of the code in physical memory; not actively used的code/data可以放disk中为别的程序的active data让位置

③ provide illusion of a much larger physical memory.

③ little to none ~~external~~ fragmentation:



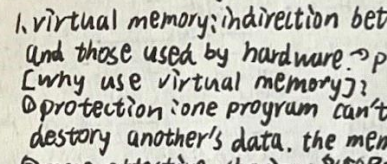
seg. 易发external fragmen...

paging易引发internal fragme...

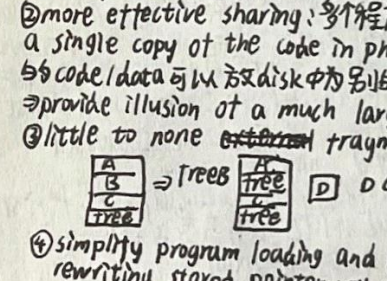
二者结合即可little to none frag...

④ simplify program loading and execution: no relocation of code, rewriting stored pointer values, etc.

[实现逻辑]



2. x86 protection model



kernel (ring 0): 高优先级; user (ring 3) 低优先级

kernel从不call/trust user, user通过narrow interface requestor 来call kernel.

CPL: 现在执行的code的优先级

RPL: requestor's priviledge level

DPL: descriptor priviledge level, 执行code/data必需的优先级

it MAX(CPL, RPL) > DPL, processor generate an exception (general protection fault)

当user code调用system call, ⇒ CPL=0, RPL=3

level 1, 2 not used by Linux

b: bit, B: byte 1B=2³ bit

1kB=2¹⁰B, 1MB=2¹⁰kB, 1GB=2¹⁰MB

segmentation: x86 ~~must~~ must always use

segment: a contiguous portion of a linear address space

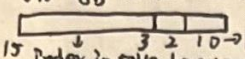
① GDT: global descriptor table 可被所有程序使用

I. GDTR: 48 bit register points to the GDT and hold table size (16~47b, 0~15b size-1 in byte)

I. each segment descriptor has 8B, 包括 base addr, size, DPL & some other bits.

GDT占64kB, 元素为8B ⇒ 可有 $\frac{64kB}{8B}=8192$ 个descriptor, 其中GDT[0]不使用

I. segment register:

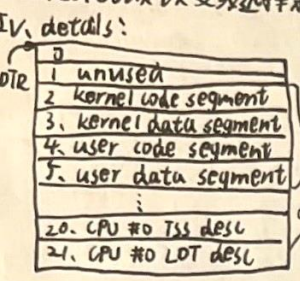


CS (code segment) 16-bit selector visible, 64-bit invisible descriptor

DS (data segment) ES (extra segment), SS (stack segment)

64 bit shadow 可用来cache the value stored in GDT/LDT. 要reload 如果GDT/LDT的项改变或选择新的LDT

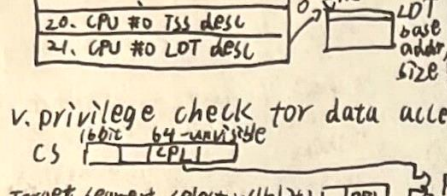
IV. details:



segment	Base	G	Limit	DPL
user code	0x0	1	0xfffff	3
user data	0x0	1	0xfffff	3
kernel code	0x0	1	0xfffff	0
kernel data	0x0	1	0xfffff	0

G (granularity flag): ⇒ { 0, seg size in B; 1, seg size in multiple of 4kB }

V. privilege check for data access



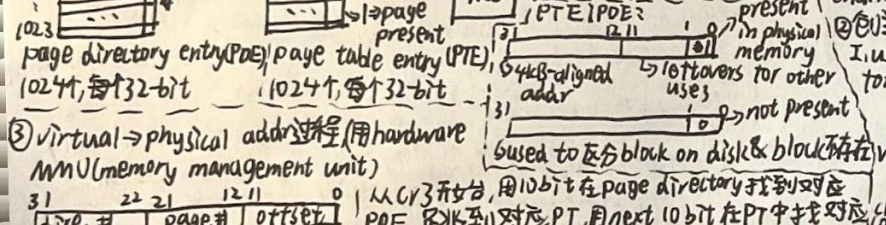
在 selector for the descriptor of the target segment is loaded into DS 时发生

4. Paging:

① Page state: present: exist in physical memory; nonexist: 试图访问 ⇒ page fault

swapped out: exist but in the disk rather than physical memory ⇒ seg. fault

② page directory (4kB) → page table (4kB) → Page: 4kB



PDBR (cr3): page directory base pointer

page directory entry (PDE) 1024个, 每个32-bit

page table entry (PTE) 1024个, 每个32-bit

③ virtual → physical addr 过程, 用 hardware MMU (memory management unit)

31	22 21	12 11	0
dire. #	page #	offset	
10bit 记录1个page table	10bit 记录1个page	12bit: page中offset	

从cr3开始, 用10bit在page directory找到对应PDE, 跳转到对应PT, 用next 10 bit 在PT中找对应PTE, 跳转到对应page, 加上last 12bit offset

若实现4MB分页, 把最后22bit合并为offset

这样 too slow! (involve too many memory access)

⇒ TLBs: translation lookaside butters: keep translutions of first 20 bits and reuse them (存对应PDE和PTE, 只有需要变时才更换)

TLB miss: TLB未存对应转换, 但要用对个转换的page

TLBs flushed (重新加载) when cr3 is reloaded when context is switched.

④ 11 leftovers in PDE|PTE: protect / optimize to improve performance

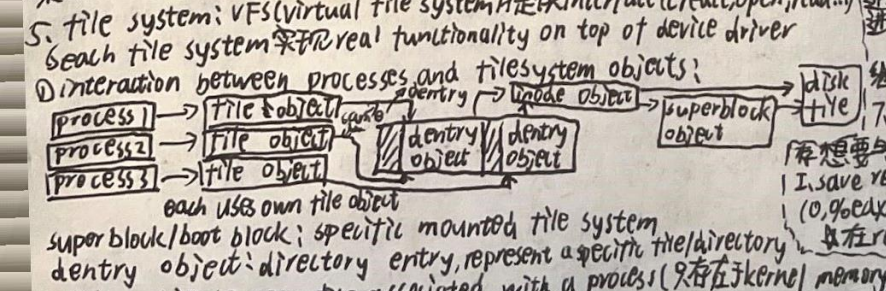
I. protect: PTE, PDE中 U/S: user/supervisor 要求 MAX(CPL, RPL) < 3

II. optimize: PTE中 G flag (global): TLB not flushed when 改cr3 change, 通常在kernel page (translation不变), PDE中 PS: PS=1 ⇒ PDE指向4MB page, 剩下22bit 表示offset.

5. file system: VFS (virtual file system) 提供interface (create, open, read...)

each file system 实现 real functionality on top of device driver

① interaction between processes and filesystem objects:

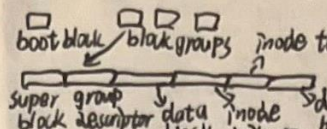


superblock/boot block: specific mounted file system

dentry object: directory entry, represent a specific file/directory

file object: open file associated with a process (只存在于kernel memory)

② ext2 (second extended filesystem) on disk, 创造时固定block size



superblock: 1kB大小, 存① file system 的block size ② check infor.. ③ file system state: 0 (mounted/uncleanly dismounted), 1 (cleanly dismounted), 2 (存在error)

group descriptor: 32B大小, shortcut for block/inode bitmap, 和inode table/data block, free block count

冗余每个blockgroup前复制: reliability

inode: 128B, 最重要的是i_block[15] 存file对应的data block

① 前12个, 直接指向data block (共12b Byte) b表示block size

② 第13个: indirect pointer ($\frac{b}{4}$ Pointers, $\frac{b^2}{4}$ Byte)

③ 第14个: double indirect pointer ($\frac{b^3}{16}$ data byte)

④ 第15个: tribly indirect pointer ($\frac{b^4}{64}$ data byte)

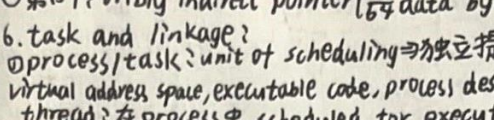
6. task and linkage:

① process/task: unit of scheduling ⇒ 独立提供 resources 来run程序 (有virtual address space, executable code, process descriptor 和至少一个thread)

thread: 在process中, scheduled for execution, 1个process可以有多个thread, 它们共享virtual address space和system resources, 但有独立的优先级和scheduled时要用的数据, 如kernel stack和user stack.

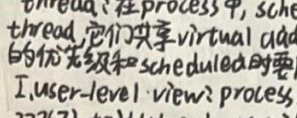
I. user-level view: process 有 process descriptor task_struct, pid (1~32767), tgid (thread group id) 在multithread process中充当pid作用

II. kernel view: 必须能同时处理多个process, 在 single per-process area (8kB) 中存两种data struct: both dynamically allocated



不要在kernel里用recursion

III. task struct: cyclic, doubly-linked list



init task { first task created by kernel at boot time; persist until machine shutdown/reboot; pid=1 }

[for system call]:

I. translate pid to struct pid*, 变成指针 (small structure referencing task)

II. find_pid 用 hash table 来把 large, sparse space map 到 small dense space上, pid_hash里装pid*, use chaining to handle colliding PIDs

chaining 为双向链表, 便于用删除pid

② 创造process/task

I. user-level: 先fork再exec

fork: 单创造一个子进程, 子进程和父进程同时进行, 子进程复制父进程的address space: 在早期系统中缓慢, 而在exec, most data discarded

vfork: 创造一个子进程, 父进程blocks, 子进程用父进程的pid和address space, 直接共享数据, 子进程exit后, 父进程会回 control of addr space

clone: copy-on-write: 子进程复制page tables, 关闭write权限, 只有子进程write时才给private copy.

"lazy" approach (要用时才创造), avoid work that won't be useful

II. kernel view: fork, vfork, clone 都除调用 do_fork

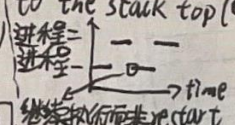
do_fork calls copy_process (same file) to set up the process descriptor and any other kernel data structures necessary for child execution (e.g. thread_info structure and Kernel Mode stack)

[kernel thread]: 没有对应的addr space, 从last user task to execute 中inherit addr space.

③ x86 task state segment (TSS)

I. TSS中ESP, SS 存 value for user privilege level

II. TSS中最重要的是 stack segment descriptor (SS0) 和 pointer to the stack top (ESP0)



被抢占执行时, TSS里有 user stack ESP, kernel stack ESP0 两是 执行完一段时, 存esp和esp0, 下次换回时 process时可以restoration (存到process descriptor)

7. system call: ① systemcall 要用 assembly linkage, INT 0x80, eax

存想要的system call #, 结束后eax存return value (error时为 -1 ~ -4095)

I. save regs to stack II. check for valid system call # III call *sys_call_table (0, %%eax, 4) IV. 执行完eax存return value V. restore all regs, IRET

在relocatable library 转到 static data (errno) 的地址, 核心是知道 现在的位置和代码中errno相对位置不变

1. MP2

① mode X: 256color (8 bits, 6bit for RGB), 320×200 resolution | video memory: 32bit physical memory 0xA0000 到 0xAffff

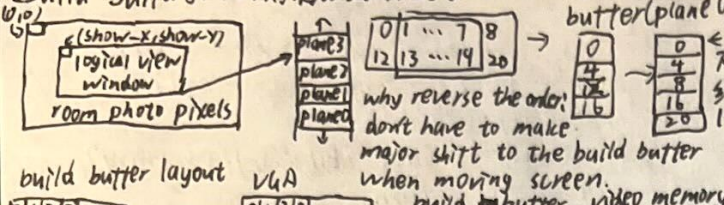
4 pixel at 14: 0 1 2 3 | 0 1 2 3 | 0 1 2 3 (0xA03, 0xA04, 0xA25)
addr: 0 1 2 3 → 0xA123+80

e 写时，用4bits of VGA reg 来表示哪个 plane
原因: changing a VGA reg value 很麻烦

可以同时向多个 plane 写入: 用 set_write_mask() → 注意同步, 0-index

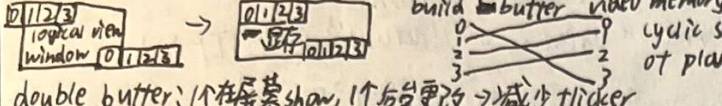
split the screen: 改 line compare register: 182-2=0x01b8, 高位放 max scan line reg(a), 低位放 line compare reg.(24)

(build buffer): 两个则有 width 为 256 的 memory tence



why reverse the order: don't have to make major shift to the build buffer when moving screen.

build buffer layout



cyclic shift of planes

double buffer: 1个在屏幕show, 1个后台更改 → 减少 flicker

i = y3 + ~~logical view window~~ X/4 + ~~logical view window~~ Y * SCROLL_X_WIDTH
→ 指向 build buffer 中 plane 3 开头

```
draw_horiz_line:
y += show_y;
(*horiz_line_fn)(show_x, y, buf);
addr = img3 + (show_x/4) + y*SCROLL_X_WIDTH;
p_off = (3 - (show_x & 3));
for (i=0; i < SCROLL_X_DIM; i++){
    addr[p_off * SCROLL_SIZE] = buf[i];
    if (--p_off < 0){
        p_off = 3;
        addr++;
    }
}
```

```
draw_vert_line:
x += show_x;
(*vert_line_fn)(x, show_y, buf);
addr = img3 + x/4 + show_y*SCROLL_X_WIDTH;
p_off = (3 - (x & 3));
for (i=0; i < SCROLL_Y_DIM; i++){
    addr[p_off * SCROLL_SIZE] = buf[i];
    addr += SCROLL_X_WIDTH;
}
```

② TUX: response from controler to PC: 3个8bit的packets,
I. TUX_INIT: 初始化, return 0, 所有user-level code 先调用这个函数
II. TUX_SET_LED: 参数32bit, 低16bit是 hex value of the number displayed on 7-seyment displays (4bit 1位数, 0~F), bit 20~23 mask 哪个 LED 亮, bit 24~27 表示哪个小数点该亮
如 LED_buffer[0] = MTCP_LED_SET, LED_buffer[1] = 0x F, 接下来传对应 LED 数字
要用 ACK_flag 来判断是否有别的 LED 在用, 因为 MTCP_LED_SET 执行完会返回 MTCP_ACK, 以应对 spamming input
III. TUX_BUTTONS: 参数是指向 32-bit 的指针, 返回 -EINVAL 如果指针无效
要用 lock: interrupt handler 可能会写 button
IV. tuxctl_handle_packet: handle packets received by the PC from TUX controller: byte0: MTCP_BIOC_EVENT, byte1: 1 x x x | C | B | A | START
byte2: 1 x x x | right | down | left | up

③ palette: photo 里是 16 bit: 5 6 5 RGB, 总共 256 个 color, 前 64 个 reserved ⇒ 192 个可用, octree 用 RGB 的 MSB 来区分, 在 level 4 中找最常见的 128 个, 剩下的用 level 2 的 64 个 color 凑和. 逻辑: 先计数, 然后用 qsort 排序找前 128 个放 level 4, 然后回到 level 2 算剩下对应平均, 注意 R, B 需先打到 6 位, 最低位补 0.

④ 注意事项: pthread_cond_wait(lock, CV)
I. synchronization: status bar 要用 lock, tux thread 先等到 lock 然后 wait 等 game loop 中 button 按下后再 cond_signal (及 tux_cv) condition variables.
II. interrupt/polling: polling 一直问问 driver, interrupt 是 driver → PC 优点是 better performance, no need to continuously asking. (TUX is slow, polling will slow computer processor to TUX's speed)

2. MP3

① GDT: ~~gdt~~ gdt_desc(48B): .word 的 limit, .long 的 base

② idt: exception 用 trap gate, dpl=0, interrupt 用 INTR gate, dpl=0
system call 用 trap gate, dpl=3, INTR gate 会 clear IF flag, TRAP gate 不会

③ PIC: I. master PIC command (0x20), data (0x21); slave PIC command (0x40), data (0xA1)
II. idt 中 master 对应 0x20~0x27, slave 对应 0x28~0x2F, slave 接 master irq2
III. initialization: 先 mask interrupt: outb(0xff, 0x21) 再; outb(0x11, ICW1: 开始初始化, edge-triggered input, cascade mode, 4ICW? 0x20)
ICW2: high bit of vector #: outb_p(0x20+0, 0x21), outb_p(0x20+8, 0x21)
ICW3: { primary: bit vector of secondary: outb_p(0x04, 0x21)
secondary: input pin of primary: outb_p(0x02, 0xA1)
ICW4: ISA = x86, normal/auto EOI
(eg 2 set IF to 1 ⇒ double fault exception: PIC 初始化错误, 生成 wrong vector number)

IV. send_EOI: if (irq_num & 8) { outb((irq_num-8)|EOI, SLAVE_PORT); → outb(2|EOI, MASTER_PORT); }
一定要向主PIC发送 不然后续只有主PIC
else { outb(irq_num|EOI, MASTER_PORT)
后的irq0,1能正常工作
如 interrupt 在执行完后要 send_EOI! ← handler 要 cli(), sti()

④ keyboard: PS/2 controller, 从 0x60 获取, 连到 irq1

⑤ RTC: 用 port 0x70 (指定用哪个 reg), 0x71 (data), 选 regs 可 mask NMI
每次读写完都会清零, 需重新指定 regs, 然后 enable_irq(8)
每次 handler 要加: outb(0x70, 0x0C), inb(0x71), (reg 7)

⑥ paging: 0x138000~0x138FFF: video memory (4kB), 0x400000~0x7FFFFF: kernel (4MB), 其他地方: page fault

⑦ filesystem: I. file name 在 dir_entry 中, 好处: search the file faster: only look in the boot block for the file rather than looping in data block.
如为什么用 linkage: C function 结束后默认 ret, interrupt 需要从 kernel 返回 user, 要用 iret ⇒ 用 asm linkage 实现
IRET! 没有的话 kernel crashed after interrupt handler 执行完.

补充:
1. memory tence tradeoff: performance overhead 与 protection
2. 用 palette 而不是直接填颜色: reduce memory overhead
memory space to keep the palette: 256·3B, without: 320·200·3B
video memory for implementation with a palette: 320·200·1B
3. MP2 synchronization:
① use main thread: 不能 concurrent
② use new thread only: waiting for TUX, wasting potential CPU power 或 CPU 多次响应
③ use new thread and condition variable: 当 button 按下时再唤醒 tux thread.
4. purpose of having more than 4GB physical memory
if only 4GB virtual memory can be addressed: Multiple programs can map virtual address to their own physical memory so having more physical memory allows less disk swapping to occur when you are running many programs
5. 画 4x4 buttom 直接到显存:

```
set_mask(0xf);
for (j=0; j < BUTTON_DIM; j++){
    video[button_offsets[i] + j*SCREEN_PLANE_WIDTH] = c;
}
```

② change the LED color without redrawing the LED segments: change the corresponding RGB color of that index in the color palette.

③ button 和 LED segment 之间 synchronization: set mask is the main synchronization concern. If one thread is waiting to the video memory while the other wants to start it may overwrite the mask desired by the other thread and cause pixels to be in incorrect page. Another answer: both of them could set palette at the same time, but the TUX controller hasn't replied with Ack.

6. 不能在 kernel stack 中 allocate large objects: kernel is is limited to 8kB per-process in virtual memory; large objects may destory thread id; kernel's memory is not pageable; kernel usually want physically continuous memory.

7. TLB flush when switch between process: The TLB must be flushed when switching processes to ensure a process can't access data stored in memory pages of another process. (isolates the processes' data/memory from each other)

8. 用 paging 不用 segmentation 的优点: I. protection, II. less external fragmentation. 缺点: take more memory to hold page directory/table (overhead), need dedicated hardware (TLB, MMU).

9. ext2 file system 缺点: I. lack of journaling. II. high data fragmentation. limitation: max file size, max number of inodes

10. reserved byte 在 boot block 中作用: I. align each block to 4kB, align dentry to 64B, pad things to power of 2 so index just shift, add, yields better performance. II. need reserved bit in the future 什么东西的能.

11. 在 mp3 filesystem 中实现 write 会很不高效: don't know which data block has been used and which is free. 改进: use a data block bitmap to show which data block has been used.

12. purpose of TLB: TLB is a memory cache that's part of the CPU that stores recent translations from virtual memory addr to physical addr. It significantly reduces the time penalty for virtual memory access compared to physical memory addressing.

13. Difference between process and thread: The process is a whole program in execution, thread is a unit of execution and it's usually part of/contained within the process.

14. I. doesn't send EOI to master PIC so it disable interrupt. II. master PIC not reliant on slave to operate. III. >=8, slave interrupts will be influenced.

15. The one with the 20 levels of paging has a ton of overhead: it use tons of memory with little portion of actual usable memory. It is also too slow because we need to traverse the page table tree 20 levels, meaning 21 memory accesses to read 1 byte.

16. The 1GB paging can't be used for video memory and shared libraries. The 4B pages allow for a ton of granularity when allocating memory to userspace and shared data, less internal fragmentation.

17. internal fragmentation: wasted memory that has been allocated within page

external fragmentation: memory blocks scattered but can't use together.

18. every context switch flushes the TLB, so everything becomes a TLB miss.

---

8. system call wrapper: 以 open 为例

open: pushl %ebx　　装 system call #

装 arg　movl 0x08(%esp), %ebx … 或 movl $0x05 %eax

int $0x80 → 看返回值: cmpl $0xFFFFF001, %eax

error 情况: xorl %edx, %edx, subl %eax %edx　存正的 error number　pushl %edx

call _errno_location ← eax 装 pointer to errno

popl %ecx　movl %ecx, (%eax) 把取正的 error num 放到 errno 中

eax 装 -1, popl %ebx, ret

_errno_location: call getIP　take call, 目的 eax ← raddr

raddr: addl $errno-raddr, %eax → offset 不变

ret

getIP: movl (%esp), %eax

ret

19. open_wrapper(x86) — user stack; system_call(x86) — kernel stack; sys_open(C) — kernel stack

save callee-saved reg.
move parameter into reg.
move the system call num to a reg.
throw the 0x80 interrupt

save all regs.
call 此 system call number and system call table
restore all regs.
put return value in proper place
IRET

return

pop callee-saved regs.
ret.

20.
```
uint32_t* PD_to_PT1(uint32_t* PDaddr, uint32_t v_addr){
    if(PD_addr==NULL) return NULL;
    uint32_t pd_entry=v_addr>>22;
    int32_t actual=((int32_t*)PD_addr)[pd_entry];
    if(actual%2==0) return;
    return (uint32_t*)(actual & 0xFFFFF00);
}
uint32_t* PT2_to_page(uint32_t* PT2_addr, uint32_t v_addr){
    if(PT2_addr==NULL) return NULL;
    uint32_t pd_entry=((v_addr)>>10)&0x3F;
    int32_t actual=((int32_t*)PT2_addr)[pd_entry];
    if(actual%2==0) return NULL;
    return(uint32_t*)(actual & 0xFFFFF00);
}
```

21.
```
void draw_horizontal_line(x,y,length,color){
    end_x=x+length-1; start_plane=x%4; end_plane=end_x%4;
    start_addr=x14 + SCROLL_X_WIDTH*y; end_addr=...;
    if(start_addr==end_addr){ set_mask((0xf<<start_plane)&
    (0xf>>(3-end_plane))) mem_img[start_addr]=color; } else{
    set_mask(0xf<<start_plane)) mem_img[start_addr]=color;
    set_mask(0xf); for(中间 addr), ...;
    set_mask(0xf>>(3-end_plane)) mem_img[end_addr]=color;
}
```

22.
```
① void create_dir_entry(filename){ Fs_start->dir_ent_num++;
strcpy(Fs_start->dir_ent_table[12].filename, filename);
...filetype=2;}  ② create_inode(int32_t filelength){
dentry_t* new_dentry=&(Fs_start->dir_ent_table[12]);
new_dentry->inode=24; (inode_t*)(Fs_start+12+1)->filelength
=filelength; int block num=0; if(filelength%4096==0){
blocknum=filelength/4096; }else{ blocknum=filelength/4096+1
}; int i=36, int j=0, while(blocknum>0){ blocknum--;
(inode_t*)(Fs_start+12+1)->data_block_table[j++]=i++;
③ write_data_blocks(const void* buf,
int32_t filelength){ void* block_addr=(void*)(Fs_start+1
+Fs_start->inode_number+36); memcpy(block_addr,
buf, filelength); }
```

23.

# levels	size of pages	# page offset bits	# index bits (1 level)
2	4kB	12	10
20	4kB	12	1
1	1GB	30	2
n	s	$\log_2 s$	$\frac{32-\log_2 s}{n}$

# levels	Size of pages	Size of One entry	# Entries/level	Size of Paging data (1 level)
2	4kB	4B	1024	4kB
20	4kB	4B	2	8B
1	1GB	4B	4	16B
n	s	4B	$2^{\frac{32-\log_2 s}{n}}$	4B
2	4B	4B	2^{15}	2^{17}B