一. x86 位数小的存在小内存

0x78	小内存
0x56	↓
0x34	
0x12	大内存

1.储存方式: little end first
0x12345678 → 0x78 ··· 0x12 (+1, −1)

immediate number: "$"号
$0x__hex; $0__octal
$53decimal

2. 操作: INC, DEC, XOR(每一位都执行)(同0得1), SHL(左移), SAR(算术右移,补符号位), SHR(逻辑右移,补0), ROL(循环左移), ROR(循环右移)

ORL %ECX %EBX → destination and first source
operation, data type, second source
#EBX ← EBX OR ECX
L: long (32 bit) W: word (16 bit) B: byte (8 bit)

default ① 2, 4, 8 | 可以只有1个reg: (, %ecx, 4)

②memory operand: displacement (SR1, SR2, scale) ← SR2*scale + SR1 + displacement
偏移值 除了ESP

MOV src, dst 将"1"移到"2"中, "dst"是reg/memory
LEA src, dst "src"是memory, "dst"是reg. 把src这个数移入"dst"中

③CMPL %EAX, %EBX #flags ← EBX−EAX "2"−"1" 只改变flag, reg不变!
TESTL %EAX, %EBX #flags ← (EBX AND EAX) "2" AND "1"

jne jb jbe je jae ja ← unsigned
≠ < ≤ = ≥ >
jne jl jle je jge jg ← signed

④data size conversion: MOVsize from to
MOVSBL %AH %ECX # ECX ← sign extend to 32bit (AH)
MOVZBL 4(%EBP), %EAX # EAX ← zero extend to 32bit (M[EBP+4])

⑤input, output independent I/O
IN port, dest.reg   OUT src.reg, port
all data to/from AL, AX, EAX
port: 8bit imm 或 DX
INB $0X40, %AL #AL ← P[0X40] INW (%DX), %AX #AL ← P[DX], AH ← P[DX+1]
OUTB %AL, (%DX) #P[DX] ← AL OUTW %AX, 68 #P[68] ← AL, P[69] ← AH

3. calling convention

local变量 ← EBP, ESP
old EBP
ret. add
int key
int* array
int size

argument (不能reuse: 可能会变!)

①参数(argument), 从右往左入栈
int fun(int key, int* array, int size)
why: I. 参数内存地址fixed: 1st arg is 8(%ebp) II. 允许variable number of 参数

②caller saved reg(主程序保存): EAX, ECX, EDX, EFLAGS
callee saved reg(子程序保存, 出去还原): ESP, EBP; EBX, EDI, ESI

[caller sequence]:
I. save caller-saved reg (push) II. argument入栈 (push) III. call子程序
IV. argument出栈 (add ··· %esp) V. restore caller-saved reg (pop)

[callee sequence]: save old base pointer, get a new one
I. pushl %ebp; movl %esp, %ebp II. save callee-saved reg (push)
III. make space for local变量 (subl $4, %esp) IV. 正常执行
V. tear down local变量 (addl $4, %esp) VI. restore callee-saved reg (pop)
VII. load old base pointer, return (leave; ret)

[eg]. 主程序
XORL %ESI, %ESI (初始化reg) #ESI ← 0
push ··· argument入栈
call func 调用子程序
addl $8 %esp

子程序:
pushl %ebp 固定流程
movl %esp, %ebp
subl $4, %esp local变量
movl %eax, −4(%ebp) 存入local变量
movl −4(%ebp), %eax 返回值
leave: ESP ← EBP+4; EBP ← M[EBP]
ret: EIP ← M[ESP], ESP ← ESP+4

二. Synchronization

why we need: ①protect shared resources ②prevent interrupts from corrupting data ③race condition ④critical section

1. system software's role: ①virtualization (the illusion of multiple/practically unlimited resource) ②protection ③abstraction 隐藏底层细节, 提供simpler接口

2. I/O: ①independent I/O: using distinct instructions to separates I/O ports from memory address IN, OUT ← keyboard (device)
②memory-mapped I/O: no new instructions, has a region of memory address that is set aside for I/O VGA

3. shared data/structure
interrupt不能改变reg中内容
①interrupt handler: preserve the contents of regs, avoid overwriting memory location used by interrupted program interrupt: preserve all regs used
I. volatile: this value may change at any time so compiler won't optimize it
volatile int busy = 0 [不要什么变量都volatile]: always reload 变量 ⇒ more memory ops → slower program.
critical section should be short: avoid delaying device; long delays crash system
②critical section: 用IF实现
CLI: clear IF
(code): atomically: the entire critical section either excuted or none excuted, (no moving memory ops in/out critical section)
STI: set IF

4. multiprocessors, SMP (symmetric multiprocessor)
all processors have equal latency to all memory banks

①spin_lock, lock是"1", unlock是"0"
program等待解锁而非执行其他
用于interrupt handler (短!)
locking must be atomic with respect to other processors! (不然出现多个processor共用lock)

why we need lock: IF not cleared on other processor
spin_lock_irqsave: 先CLI再lock (避免deadlock)
spin_unlock_irqstore: 先unlock再STI

[deadlock]: a thread always hold the lock is waiting for an interrupt to finish, But the interrupt is tring to obtain the lock and fail, thus the program can't work. Lock always belong to the thread. 单个processor上发生!
[livelock]: multiple threads all need to acquire some same locks, but none of them can get all locks. They will release the locks they acquired and try for new, but never get all. The locks never belong to one thread. 多个processor!
解决办法: 按照同一顺序acquire locks ← lock ordering

②semaphore: 允许fixed number的processor访问同一个critical section (FIFO)
P(down): decrement 尝试获取 V(up): increment 用完释放
MUTEX: thread = 1
当一个程序等待semaphore时, 允许其他程序在processor上运行
用于system call, 尤其是long critical section
↳不能用于interrupt handler: 要求快速执行; 可能deadlock
↳先acquire semaphore (mutex) 再acquire spin lock: tring acquire semaphore may put calling process to sleep, shouldn't happen while holding spinlock (deadlock)

③reader/writer problem: 允许无数个reader, 但writer只有1个, 且在无reader下进行
I. reader/writer spin_lock: 用于short critical sections/interrupt handler
↳admit writer starvation (一直有reader请求, 不能写)
II. reader/writer semaphore: 用于system call
↳不admit writer starvation (writer请求时, reader排writer后面)

三. task和linkage

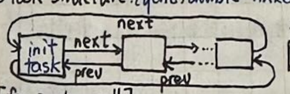
1. process/task: unit of scheduling ⇒ 独立提供resources来run program (有virtual address space, executable code, process descriptor and at least one thread.
thread: 在process中, scheduled for execution. 1个process可以有多个thread, 它们共享virtual addr space, system resources, 但有独立优先级和scheduled时要用的数据, 如kernel stack和user stack.

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

②kernel view: 同时处理多个process, 在single per-process area (8kB) 中存:

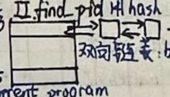
8kB
thread_info → pointer to task_struct或process descriptor (PCB)
↑ → dynamically allocated
kernel stack 不要在kernel中用recursion: overflow the kernel stack

③task structure: cyclic, double-linked list



init_task: 还有ksoftirqd: soft interrupt daemon. kernel thread: boot时创造, 一直到shutdown/重启
pid=1 → init_mm/fs/files/signal ···
map kernel pages

[for systemcall].
I. 传pid*: small structure refering task
II. find_pid 用hash table来把large, sparse space map到small, dense space上

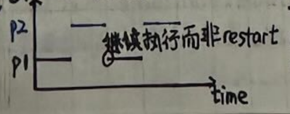


双向链表: back link 用于删除

2. 创造process/task
I. user-level: 先fork再exec
fork ← create a copy of current program
exec ← load a new program and start it
fork: 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address space.
↳在早期系统中很慢, 且在exec过程中most data discarded.
↳父进程中返回子进程pid, 子进程中返回0.
还有个clone: 没学过
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, 父进程拿回the control of address space.
"lazy approach" → copy-on-write: 子进程复制page tables, 去掉write权限, 只有当子进程write (fork中) 才给private copy ↳作用: avoid work that won't be useful.

II. kernel view: fork, vfork, clone 三个system call都会调用do_fork (在kernel中)
int do_fork (clone_flags, 如CLONE_PARENT: 创造sibling task而不是child
stack_start, new task's ESP (user level; kernel 为0)
regs, regs value for the new task, stack_size, parent/child_tidptr);
返回new pid 或error (负数)
x86只用这两个参数
do_fork先call copy_process来设置process descriptor和child要用的kernel data struture
kernel stack, thread_info
[kernel thread]: 没有对应的address space, 从last user stack to excute中继承address space (page mapping)

3. process/task switch

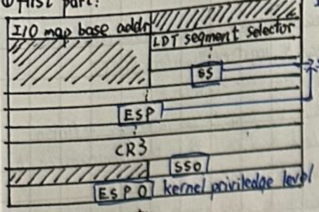


继续执行而非restart
(每个process) 开始执行时: TSS里有ESP (user stack), ESP0 (kernel stack)
执行完time slice时: 存esp和esp0, 下次唤醒process时, 可以restoration (会存到process descriptor中)

4. x86 task state segment TSS

TSS descriptor中segment limit定义了I/O bitmap长度

① first part:



kernel priviledge level

save value for user priviledge level.

I/O bitmap privilege checking:

IOPL (I/O优先级level field, 在EFLAG中)

IOPL set 3, 执行I/O instruction, 否则

check I/O port在I/O permission bitmap中对应的bit, cleared则执行I/O instruction, 否则general protection exception

② second two parts: I/O permission bitmap (I/O map base addr指向start)

interrupt redirection: I/O permission bitmap

四. scheduling → goal: fair, efficient, responsive (对于keyboard反应, response no wait)

1. jobs分类:

① interactive: 互动, respond quickly如editors, GUI → 用heuristics处理, 保证jobs的"互动"

实现: 把时间分成time slice, each task run for a slice, 一直执行到no runnable task has time left

且can't use lots of CPU

② batch: 主要考虑time to completion如compilation

③ real-time: 按时完成即可, periodic dealine → always比non-real-time job优先级高

2. task state (存在task_struct/processor descriptor中)

① TASK_RUNNING: 正在执行/等待执行: task被放在run queue中, 只有这个状态会被执行

② TASK_INTERRUTIBLE: sleeping on semaphore/condition/signal, 在wait queue中, 可被signal唤醒 ③ TASK_UNINTERRUPTIBLE: 不可中断的休眠状态, device等待外部事件发生(如I/O, network data), 不能被signal唤醒 (task is busy with sth. that can't stopped)

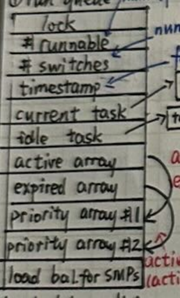
④ TASK_STOPPED: not in a queue, must be woken by signal (被主动停止)

⑤ TASK_ZOMBIE: task已经结束, 不再执行, 但data structure还在

3. scheduling data structure swapper process is always runnable (idle task)

处理内存交换 (unactive 用程或page放到disk)

① run queue

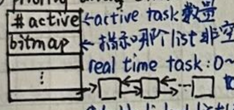


active process: runnable process且未超时

expired process: runnable process且已超时

active跑完后, change pointer (active指向#2) 实现double-buffering better performance, 减少图像更新时的闪烁

② priority array structure



4. scheduler policies

① SCHED_FIFO (first in, first-out real-time process): 一直执行直到relinguish the CPU voluntarily (自愿放弃CPU) no pre-empted! 进程执行过程中被系统中断, 切换到另一进程

② SCHED_RR (Round Robin real-time process): 给process分配time slice, 跑完后放到run queue末端

③ SCHED_NORMAL (conventional, time-shared process): 用predefined formula重新计算task priority, 相同priority的用round robin policy.

[rescheduling and yielding (放弃)]:

① task change (context switch): I. current task yield by calling scheduler (主动放弃CPU) II. run out of time ② other place task may yeld: semaphore/wake_up process

③ 在every time tick (由IRQ0上发interrupt (IF=0)): reduce current task's time

五. memory allocation

1. 4个函数

① kmalloc(size, flag):

[flag]:

- GFP_ATOMIC: does not sleep (不用于大内存request)
- may sleep to wait for page:
  - GFP_KERNEL: kernel, driver…
  - GFP_NOFS: no file sys call (不会push pages to disk)
  - GFP_NOIO: no I/O ops → 等待内存空间可用 (page在内存中出现)
- GFP_USER: user (low priority)
- GFP_DMA: DMA accessible memory — direct memory access: 直接访问physical memory
- GFP_HIGHMEM: high memory (PAE) acceptable — 允许32位系统访问超过4GB物理

用exponentially-sized slab caches (8B到4MB), allocation在physical memory上连续 用于a few small items

② slab cache: 用于a lot of items, repeatedly → reduce internal fragnentation

I. slab最大是4kB (page大小), slab是连续physical memory, 被slab allocator用作individual cache的, 会被提前分成32, 64, 256 … 大小去分配

[eg] I. file system想allocate一个inode structure: 先调用kmem_cache_create(), 计算最优分配并返回kmem_cache_t pointer (指向new inode cache) II. 当file system需要用时, 调用kmem_cache_alloc() 通过kmem_cache_t pointer). slab allocator会在slab中找free inode object, 没找到 (或slab内存不足): fetch a new slab from page and return an inode object. III. 当file system用完这个inode后, 调用kmem_cache_free() 去release the inode并标记这个在slab中的inode为free. IV. 如果一个slab中所有object都是free, 在memory不足时会将slab返回free page.

③ free pages: 用于big, physically contiguous region

只能allocate multiples of page size

get_free_page(flags) ↔ free_page(long)

get_free_page(flags, order) ↔ free_page(long, order) ← order是log(2为底数) of num of pages requested

order要match!

④ vmalloc: 用于allocate virtual memory

[与kmalloc对比]: kmalloc更快, 但可能会因memory too fragmented而失败

vmalloc更慢: 要modify page table entry to map physical memory and virtual mem

[memory fragmentation]:

I. external fragmentation: small blocks of free pages are scattered inside blocks of allocated pages thus can't used together

II. internal fragmentation: mismatch between the size of the memory request and the size of the memory allocated to satisfy the request

2. buddy system

① why buddy system prefored:

I. contiguous page frames

II. page contiguous → page table unchanged → less TLB flush

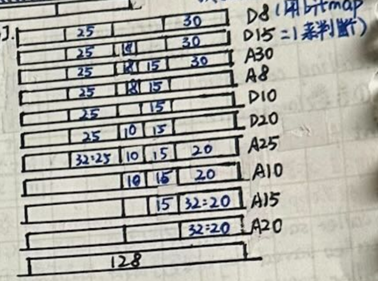
III. access by kernel through 4MB pages → less TLB miss

② partially busy bit: "1" if one buddy in use, "0" if both/neither in use [two blocks are buddy 时]: I. same size $b = 2^{order}$ 个page

II. allocated in contiguous 物理内存

III. physical address of first block 是 2·b·(page size) 的倍数

[eg].



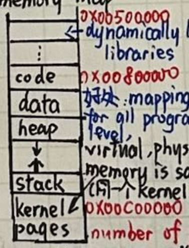
六. memory map

1. allocating memory to user mode process

① process requests for dynamic memory被kernel认为non-urgent: I. 在executable file load后, 不可能马上用all code pages II. 当一个program调用malloc()时, process不会马上access all memory obtained

② user mode process获得dynamic memory时, 意味着address space加入了new range of virtual addr, 但kernel会defer allocating dynamic memory to user mode process.

2. memory map

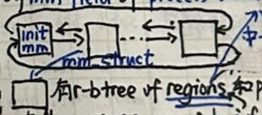


not necessary in the same place in virtual memory (mapping can change)

好处: mapping is the same for all program at kernel level, virtual, physical memory is same (同一个kernel) TLB no need to flush

number of pages in map (十进制)

process addr space被存在memory descriptor中, data structure: mm_struct, is referenced by mm field of process descriptor



virtual memory中一块连续区域

有r-b tree of regions和page directory pointer

存flags, range, ops, file info

[eg]. cat和tac process的memory map

permission: read/write/execute

mapping region: 开始和结束的virtual memory

inode of the file

the file used to provide data

用file路径判断shared

08048000 - 0804c000	r-x	4	2286660	/bin/cat	
	rw-	1		/bin/cat	
	rw-	33	0 [heap]		
	r-x	312	⋮	/lib/libc-2.7.so	shared library
	r--	1		/lib/libc-2.7.so	
RW data region	rw-	2		/lib/libc-2.7.so	
	r-x	26		/lib/libc-2.7.so	
	rw-	2		/lib/libc-2.7.so	
	rw-	21	0 [stack]		
0x8048000 - 0804c000	r-x	4	12127494	/usr/bin/tac	
⋮	rw-	1	⋮	/usr/bin/tac	
	rw-	33	0 [heap]		
"shared library"					
bfff000 - bff6b4000	rw-	21	0 [stack]		

[Q]: how much memory would be used by cat and tac process together?

[A]: cat process: 402 pages tac process: 402 pages shared library: 343 pages

solution while excluding RW data regions with RW permissions:

$2 \cdot 402 - 343 + 4_{RW\,data} = 465$

solution while including all regions that contain shared data:

$2 \cdot 402 - 343 = 461$

七. signal 相当于 user-level 的 interrupt

1. 与interrupt异同

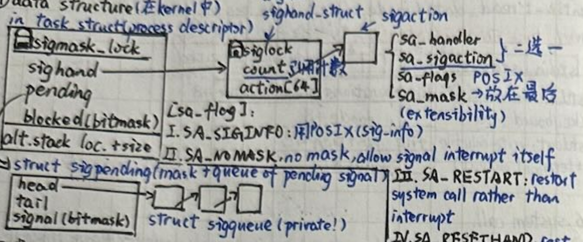
①同: asynchronous(异步); not queued (send twice, handler可能只执行1次); can be ignored, blocked (masked)或caught (执行handler func); 有default action (kernel决定), can be changed by user program; 只传递signal和这个data 给handler; signal blocked while handler execute

②异: software生成 (no device associated). only software with permission can send

③signal default action: ignore signal, terminate program, terminate program and dump core file to disk, stop program's execution, let program continue execution. NMI (default action can't change): SIGKILL, SIGSTOP

2. control of signal behavior:

①data structure (在kernel中)



[sa_flag]:
I. SA_SIGINFO: 用POSIX (sig-info)
II. SA_NOMASK: no mask, allow signal interrupt itself
III. SA_RESTART: restart system call rather than interrupt
IV. SA_RESETHAND: rest signal handler to default

②函数: sigaction(signum, newact, oldact) ← read, replace signal behavior [how]: I. SIG_BLOCK (add) II. SIG_UNBLOCK (rm)
sigprocmask(how, set, oldset); set bit vector of blocked signal III. SIG_SETMASK

3. signal generation: 而用sys_kill system call实现

①permission: sysadmin (kernel/priviledged call), process with same user id can always send, process with same login session can send SIGCONT ← continue.

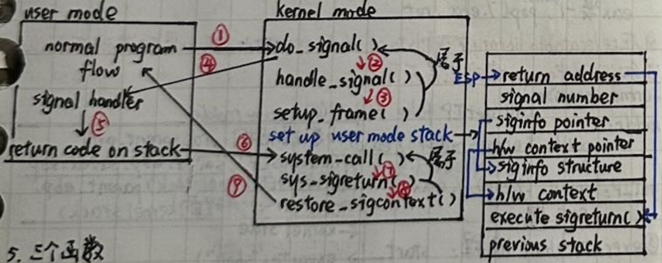
②generated signal not ignored: 加到pending中, woken up sleeping recipient (可能唤醒进程)

③通过send_sig()和force_send_sig()函数send signal, force意味着forcibly reset an ignored signal to default behavior, unblock the signal. forced signal always priviledged (在kernel中生成) → 一般用于exception ← 如果exception 被block的话, program can't execute next instruction, kernel can't deliver signal → deadlock.

④info版本多传一个info structure pointer, (void*)0表示traditional signals from users, (void*)1表示traditional signals from kernel, (void*)2表示forced signal.

4. signal delivery ← 从interrupt, exception和system call返回时触发

★ only deliver to currently executing process.



5. 三个函数

①signal(sig, handler): 给对应signal赋上handler

②exit(0): 终止程序执行并退出,

③alarm(t): 在定时"t"结束后, 发出"SIGALRM"

wait_event_interruptible(wq, condition): return 0 if condition is true.

八. device driver

1. kernel role

①process management: creating/destroying/scheduling process

②memory management: 支持virtual memory

③filesystem: treat almost everything as a file

④device control: 支持system interactions with hardware devices, device control operations are defined/implemented by device driver.

2. device driver

①kernel interacts with I/O devices by device drivers ← 在kernel level, 包括 data structure和functions. 好处是: I. encapsulated in a module II. h/w device 有well-defined interface III. hide the detail IV. dynamic load/unload device driver

treated as "device file" → 这样, same system call 可以使用, 用file descriptor abstraction处理

②block device → 用于实现file systems
I. data accessible only in blocks of fixed size
II. addressed randomly (随机访问)
III. transfer to/from device are buffered (to) 或cached (from) 如CD ROM, DVD, disk

character device
I. contiguous space of bytes
II. 有的random access (magnetic tape driver) 有的sequentially (顺序读写) (sound card)
III. not buffered when transfer
eg. keyboard, terminal, printers

③kernel abstractions for device
I. device file store as real file (inode中有表示是block/character device的地方)
II. device 由major/minor numbers 区分: major number是device type, minor number是 instance number (如果driver允许more than one of a given model attached to computer)

④registration: statically compiled in the kernel ⇒ register during kernel init phase
compiled as a kernel module ⇒ register when module loaded
[eg]. insmod ./hello.ko rmmod ./hello

⑤使用的data structure (kernel): file operations, file object, inode

九. I-mail (example of device driver)

1. security: 用I-mail administrator (user with priviledge) 优点是sysadmin可以hand off I-mail rights; without giving away other right.

2. user data/operation/lock

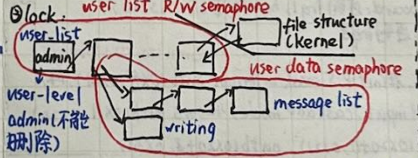
①data: authentication data (认证数据), message list, writing message (正在写的), association with a program (one file a time)

②

operation	user list R/W	user data	comment
read	x	✓	
write	x	✓	
poll	x	✓	wait for message ← 用"*"表示需要delivery
fsync *	read (x)	✓	[message delivery]: 需要user list R/W 和收message的recipient's user data semaphore. 不能同时持有两个user data semaphore! ⇒ 避免deadlock (考虑同时多个delivery发生)
release *	x	✓	
authenticate	read	✓	
set password	write	x	
start writting *	read	✓	
delete message	x	✓	
add new user	write	☒	→ hold writer lock, 不允许别的程序读写的user.
delete user	write	☑	→ sychronization (其他执行完再删)

ioctl

③lock:



lock order:
I. user list r/w semaphore (因user list中read多write少) (希望有concurrent authorization)
II. user data semaphore (lots of read, write) (by one user process)

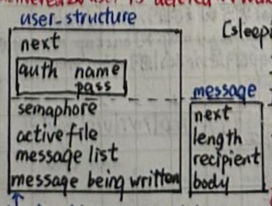
3. wait queue: 在一个task不能make progress immediately (等待semaphore, page时), 把task sleep: 放到wait queue中等待唤醒, 让scheduler跑别的程序

★race condition: task checking sleep condition, sleep和 other task search the queue, wake it up之间 wait queue是double-linked list

①. if (!(condition)) { while (1) { 开始critical section (wq→lock);
add to wait queue; set task state to TASK_INTERRUPTIBLE; 结束critical section
if (condition) break; ← 满足条件, wake up.
if (!signal_pending(current)) {schedule(); continue; } ← 继续sleep.
break; }} ← deliver signal, maybe return.

[函数] wake_up() 唤醒1个task wake_up_all() 唤醒wait_queue中全部task
wake_up interruptible() 唤醒1个 interruptible task

②在I-mail中: read和poll要wait for a new message, 有可能sleep, 在new message delivered或user is deleted时wake up!



加一个wait_queue_head_t

[sleeping流程]. I. release all locks
II. check condition without locks ⇒ condition must be atomically safe
III. go to sleep
IV. awoken → reacquire all locks, recheck all validty requirments for waking (不然是false alarm)

★sleeper doesn't have locks!

[check condition例子]: I. 一定可行的: read an integer (并比较另个常数); add integers together; read a pointer and 与NULL比较 II. 不一定可行的: dereferencing a pointer; 遍历pointer-based data structure

[reconstructure]: 在critical section中recalculate condition 并用integer存, woken后读这个integer.

③ I-mail read:

I. 先check是否认证: unauthenticated state → throw it out

II. while(1){ acquire user data lock;

check the state of file ⇒ NULL: return -EPERM; ← 在loop中check: 因为可能sleep, 导致delete在loop中发生

同时check 3 user是否deleted: 放在lock中进行, 避免race condition.

find the message: leave the loop with the lock;

otherwise, release the lock and sleep; derefrence 可以: user data只在release被free, 而release不可能被call

if(wait_event_interruptible(·· , (NULL == udata->active_file || NULL != udata->msg_list))

}

III. read the data ← { user not deleted / user has a message / hold the user data semaphore

release the semaphore, return.

4. dynamic allocation: 在critical section外发生, 避免deadlock, 用GFP_KERNEL

[Q] user is using I-mail, 但user data is deleted?

[A]. delete只remove from user list, 不free the structure, 由active_file的值决定是否delete

$\boxed{\text{file}}$ → NULL unauthenticated (NOT_AUTH)

$\boxed{\text{file}}$ → $\boxed{\text{user}}$ → NULL deleted (DELETED)

$\boxed{\text{file}}$ ⇄ $\boxed{\text{user}}$ authenticated (GOOD)

NOT_AUTH $\xrightarrow{\text{authenticate}}$ good $\xrightarrow{\text{delete user}}$ DELETED

why user doesn't authorize after check? auth need read lock, delete ops hold write locks on user list.

5. sub-operations 会看会不会与delete有影响

① find by name: acquire the user list lock before calling the function.

② extract message being written: acquire the user data lock before calling function

③ deliver mail: 允许caller optionally hold user list lock (not held: acquire read user list lock), acquire recipient's user data lock, release user list lock if acquired within the function.

十. MP3 ← 存了TSS, LDT desc.

1. GDT: gdt_desc (48B): .word 的limit, .long的base → 在boot.s中用 lgdt gdt_desc载入

2. idt: exception 用TRAP gate (reserved 3 = 1), dpl = 0, interrupt 用TRAP gate (~ = 0), dpl = 0, system call 用TRAP gate, dpl = 3 只有interrupt是asychronous

INTR gate会清空 IF flag (interrupt mask), TRAP gate不会

0X00 ~ 0X1F (32个)	exception
0X20 ~ 0X27 primary PIC	IRQ0 ~ IRQ7
0X28 ~ 0X2F secondary PIC	IRQ8 ~ IRQ15
⋮	⋮
0X80	system call INT 0X80

IF: interrupt enable flag (set: 有interrupt; clear: mask ~)

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

没有IRET的话: kernel crashed after interrupt handler执行完.

keyboard: 连到 irq1, handler要cli(), sti().

rtc: 连到irq8

3. PIC: slave 接到master 的 irq2. 在init中, 先mask interrupt: outb(0Xff, 0X21), 再: 先master再slave

ICW1: 开始初始化, edge-triggered input, cascade mode, 4ICWs outb(0X11, 0X20)

ICW2: high bit of vector # outb(0X20+0, 0X21), outb(0X20+8, 0XA1)

ICW3: primary 是 bit vector of secondary outb(0X04, 0X21)

secondary 是 input pin of primary outb(0X02, 0XA1)

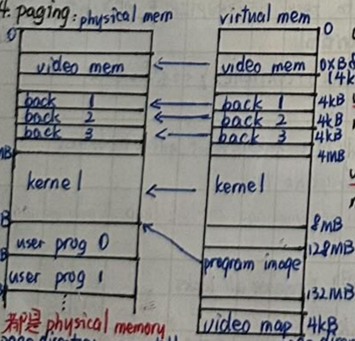
ICW4: ISA = X86, normal/auto EOI outb(0X01, 0X21)

enable_irq 和 disable_irq: 将对应bit的设成1来忽略对应irq. (初始值: 0Xff)

send_eoi: 在interrupt 执行完后发送, 要向master 和 slave两个PIC发 (如果irq在slave)

↳ 不然 当前 irq这个口被认为在工作, 往这个irq # 发的interrupt会被blocked.

4. paging:



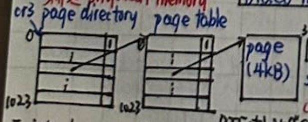
0 ~ 8MB, virtual 与 physical 相同

user_prog_paging: 将program image map 到对应的 user program上

vidmap_paging: 将program的 video page map 到video memory上

multi_vidmem_mapping: 如果传入的terminal是正在展示的, map到video memory, 否则 map 到 backup memory # 中. (map的是virtual的video memory)

都是 physical memory



page directory entry

31 ... 12	11 ~ 8	7	6	5	4	3	2	1	0
address	AVL	PS	AVL	A	PCD	PWT	U/S	R/W	P

set(1): PS → pagesize (4MB); U/S → user; R/W → R/W

cr3: 存pointer to page directory table

PTE中bit 8 "G": set(1): TLB not flush when cr3 change

[virtual memory].

31 ... 22	21 ... 12	11 ... 0
directory #	page #	offset

从cr3 (PDBR) 出发, 用10MSB在page directory中找对应PDE, 跳到对应page table, 用next 10 bit 在page table中找对应PTE, 跳到对应page, 用last 12 bit作为page中offset.

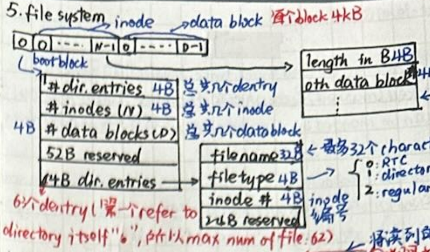
too slow (involving too many memory ops)

TLB: keep translation of first 20 bits and reuse them

TLB flushed when cr3 is reloaded {movl %%cr3, %%eax; movl %%eax, %%cr3;}

只要mapping 变了就要flush TLB!

5. file system



每个block 4kB

总共几个dentry

总共几个inode

总共几个data block

最多32个character

0: RTC / 1: directory / 2: regular file

inode 编号

4B, data block 编号

最多1023个data block

64个dentry (第一个refer to directory itself ".", 所以max num of file: 62)

```
int32_t read_dentry_by_name(fname, dentry)
int32_t read_dentry_by_index(index, dentry)
int32_t read_data(inode, offset, buf, length)
```

将读到的装入dentry中

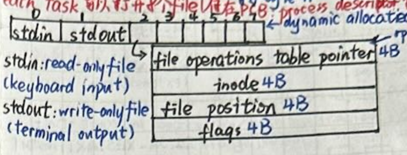
都是指针!

dentry index

按byte算

读的长度

each task 可以打开8个file (存在PCB: process descriptor中)



← dynamic allocated.

open, read, write, close 4B

stdin: read-only file (keyboard input)

stdout: write-only file (terminal output)

6. system call.

① system call 要protection boundary (user to kernel mode) ⇒ 用asm linkage:

INT $0X80 调用, eax存想要的system call #, 结束后 eax 存return value, (error时是 -1 ~ -4095, 可以取正在error table中找对应error)

[assemble linkage]: 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

ret_addr	← ESP
EBX	argument 0
ECX	argument 1
EDX	argument 2
ESI	
⋮	

[IRET context]:

USER_DS	
ESP	(128MB + 4MB − 4) → user stack
EFFLAG	
CS	USER_CS
EIP	first intruction of user program (file中byte 24-27)

② system call wrapper 以open 为例:

```
open:   pushl %ebx              callee save regs
        ⋮
        movl 0x08(esp), %ebx    装argument
        movl $0x05 %eax         装system call number #
        int $0x80               system call
        cmpl $0XFFFFF001, %eax; jb done
error情况: xorl %edx, %edx    subl %eax, %edx    pushl %edx
        call _errno_location     eax装pointer to errno
        pop %ecx   movl %ecx (%eax)   把取正的error number 装errno中
        eax装-1
done:   popl %ebx   ret
```

← 看返回值 (成不成功)

存取正的error number

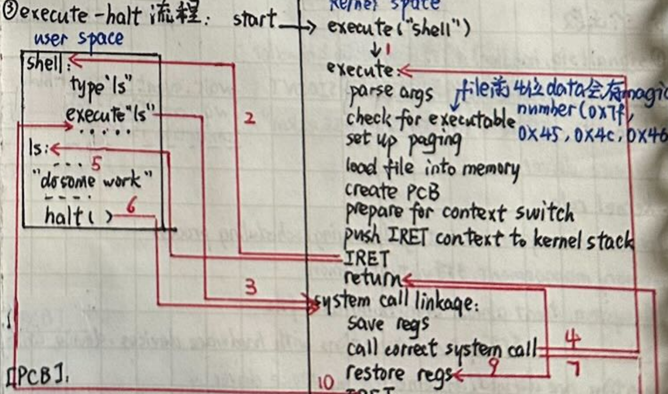
在relocatable library中找到static data的地址, 核心是知道现在的位置和代码中errno相对位置不变.

```
_errno_location:
        call getIP        fake call, 目的: eax ← raddr
raddr:
        addl $errno-raddr, %eax
        ret
getIP:  movl (%esp), %eax
        ret
```

offset (不变)

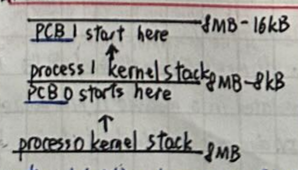
pcb数据结构: pid, parent_pid, file_dscs[8], saved_ebp, saved_esp (自己程序的kernel stack) parent_ebp, parent_esp (父进程的kernel stack)

③ execute-halt 流程:



file 前4位data会有magic number (0x7f, 0x45, 0x4c, 0x46)

[PCB].



get_pcb(pid): return (pcb_t*)(8MB − (pid+1) * 8kB)

[拿cur_pcb方法]: asm volatile("movl %%esp, %0"
:"=r(esp));
current_pid = (8MB - esp) / 8kB; cur_pcb = get_pcb(pid);

[execute "context switch"之后]: kernel stack(system call, exception, interrupt 要用)
tss.ss0 = kernel_ds, tss_esp0 = 8MB - pid*8kB - 4
/*save parent esp, ebp*/
/*IRET*/

[halt "context switch"之后]:
tss 操作一样(此时 pid 是 parent_pid)
eax 装 return value, ebp 装 parent_ebp, esp 装 parent_esp
leave, return.

④scheduler: SCHED_TASKS [3]: 一开始是 -2, 里面装每个 terminal 里跑的 pid.
为什么用pit: I. pit 用 irq0, priority 高, 不会被 interrupt II. frequency 不会像 rtc 被 user program 改变

pit_handler: 先 send_eoi(1), 然后 scheduler()

[scheduler()]:
I. 先存 ebp, esp 到 cur_pcb 的 saved_ebp/esp 中 注意判断有无process!
II. 如果是"-2"的话, multi_vidmem_mapping(cur_index), execute "shell"
已经(~+1)%3了
III. get the pcb of cur_index, user_prog_mapping
IV. multi_vidmem_mapping(pid), 改变 mapping!
V. change TSS: ss0 = kernel_ds, esp0 = 8MB - pid*8kB - 4
VI. context switch, 用当前这个 pcb 的 esp, ebp 来跳转! leave, ret 到 pit handler(asm) 的 iret, 再 switch 到 next process 的 user space.

[switch_terminal]: (从1切换到2为例)
I. video map 到 current terminal: 让 virtual 的 0XB8000 (video mem) 指到 physical video mem
II. 将 video memory 复制到 back 1 上, 将 back 2 复制到 video memory 上
III. video map 当前正在 schedule 的 process, multi_vidmem_mapping(cur_index)
此时 keyboard 和 terminal 的 write 有区别. keyboard write 到当前正在展示的 terminal 上, 而 terminal write 要写到正在 schedule 的 process 中.

十一. 补充

page table entry:

31–12	11–9	8	7	6	5	4	3	2	1	0
address	AVL	G	PAT	D	A	PCD	PWT	U/S	R/W	P

page attribute table
cache disable
global (set 1): TLB not flushed when cr3 change (在kernel中用)

D (dirty flag): set "1" 表示 page 内容被修改
A (accessed flag): set "1" 表示 page 内容被访问 (R/W)