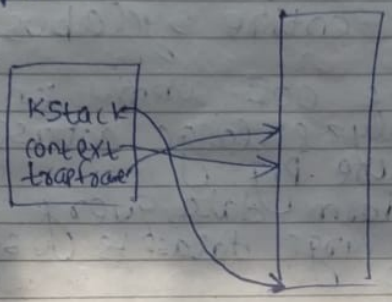
Prof Rebello - IIT Chennai } Both use
IIT Delhi } xv6

All exams will be online.

* struct proc



ushort → 2 bytes

* struct trapframe:

→ The lowermost esp and ss are the pointers to older application stack

* All read write codes are present in usys.S.

After INT instruction hardware does some pushes & jumps control to vector64. The stack change from user → kernel process stack happens in hardware

* uint err in trapframe is the push $0 in usys.S
* See argstr(), argint() in sys_ codes
4+4*n → calling convention

22/2/2022

Threads & Signal

Thread - independent flow of code execution

→ fact() divided into 2 parts:
$1 \to n/2$, $n/2+1 \to n$

pthread_join is like wait. It waits for each thread to be completed

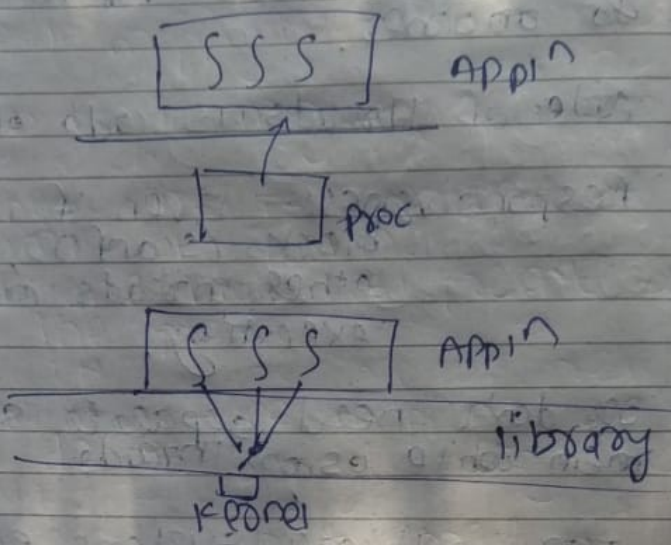
* gcc -lpthread
* Thread - flow of code which uses CPU
* Concurrency: progress at same time does not mean exec.
  Parallel: execution at same time

  For one processor, parallelism not possible.
  Parallelism needs multiprocessing
  Scheduler also schedules threads.
* fork() → heavy-weight. As PCB needs to be setup
  Thread creation takes less effort
* Youtube keeps running on 1 tab due to multi-threading.

→ when we run a function on a thread that function may call another fn. so each thread needs a separate stack.

* we need to do a context switch (save registers) to switch from 1 thread to another.
* code of threads is code of process
* Responsiveness - Even if 1 thread blocks (scanf()), then other threads can keep executing.
* we don't need separate shared mem while using threads.
* For multicore systems each core has different interrupts (eg timer)
* xv6 gives us only processes.
* user threads - The applications handle them
* How to do scheduling in user land?
* If OS doesn't provide threads it is possible that userland lib can create & manage threads.
* Even if kernel gives us threads a user library has to map those user threads → kernel, otherwise appln codes cannot access those threads.

* If there are no kernel threads



→ If u have no kernel Threads then kernel thread means process

* pthread on linux is 1:1 mapped
* In Many → one if 1 thread does blocking entire process is blocked. As kernel understands only processes
* Disadvantage of one-one thread: For one user thread one kernel thread is needed. Hence more resources.
* Two level model - Hybrid.
* clone() → creates a kernel thread

Linux → understands only Threads!

flags argument to clone() gives choice of what part of process needs to be duplicated.

fork() is a fn on top of clone()

* Learn clone() to understand threads.
* Follow syntax of pthread lib for doing project as it is POSIX standard.
* what happens when fork() or exec() are called in a thread
* Thread cancellation - terminate a thread
*

```
f() {
}
main() {
    pthread_create(f);
}
```

```
pthread_create() {
    clone(f)
}
```

lib.

---

clone()

By default duplicates stack

Linux Kernel

* If for every pthread_create() we call clone(), then it's 1-1 mapping.

* How to do many-one mapping

This one is created at the time of fork().
So one flow is present, which is the process.

For many→one mapping, we have to do job without calling clone.

* we need struct thread {
    char stack[];
    context {
        fn pointer;
    }
}

→ context → depends on calling convention

pthread_create (fn)

    t = malloc (thread)
    threads[i] = t
    swtch ()
}

In linux we have userland lib fns which get context & switch the context

→ First write project, using existing lib fns.

* change context → change stack.

Q → what is the context

swapcontext() → take context of 1 thread put on context (stack) of library then switch to new stack

* See fns getcontext() setcontext() makecontext() longjmp() setjmp()

* To get more marks - write getcontext() setcontext()

24/2/2022

Signals

Segfault - event
synchronous - predictable
system calls to make signals - synchronous way
signal() → system call
signal.h
SIG_INT - Macro for signal

Kernel has default signal handlers

SIGINT occurs when we press ctrl + C

ctrl + Z - SIGSTOP

* Synchronous way of sending signal - kill()

→ kill is not always used to kill a process.
So one process can send signal to another. This is synchronous way

* man 7 signal

* SIGKILL is 9.
kill -9 pid will DEFINITELY kill the process

kill -s SIGINT pid
→ sends signal to pid.

shows all syscalls that command is making

* strace kill -s SIGINT 8179
* kill -s SIGSTOP pid

fg → brings stopped process into foreground
i.e. process was paused → it has continued.

SIGCONT – continue

* signal() is no longer preferred choice.

sigaction() is POSIX standard

SIGCHLD → send by kernel to parent when child dies

Imp SIGALRM → Timer!

* Signal handler can only run when process is scheduled
The scheduler must now check if any pending signals are there
How to jump to signal handler from scheduler?

* Signal handler doesn't work acc to LIFO stack

* Thread pool
→ normally when we create a thread it starts running immediately with a fr

→ Thread pool is created so that it is ensured that threads will be available when they will be needed.

* global - visible in all files
static global - visible in only this file.
Hence some fns are also marked as static

* __thread is specific to gcc. Not c

* one-one mapping : Huge resources required

* How does appl? know that kernel threads are blocked?
→ Kernel has to communicate with user

upcalls → backward call. Kernel calls user code.

* How does a kernel call user fn?

* LWP → data structure given by kernel to user library
LWP part of kernel DS
user can access LWP
so our library is a wrapper around LWP

* Linux doesn't distinguish bet" process & thread

* strace / fork program

→ see that it internally calls clone

1 Mar 2022 O.S.

XV6 - paging code

We started with bootasm.S → bootmain.c → entry.S

entry creates a page table with a single entry.

Goto a main.c:

(1) kinit1(end, P2V(4*1024*1024))

4 MB

Physical to virtual

adds KERNBASE

→ We can see 'end' in kernel.sym. 'end' is last address of ELF file.

Range bet^n vstart & vend is less than 4 MB.

Ignore the lock.
Go to freerange()

PGSIZE is 4K

PGROUNDUP - Rounds up addr to a page boundary (4K, 16K, etc.)

So for every address on a page boundary, kfree() is called.

kfree() is adding a chunk to a linked list.

kfree() is passed a ptr which is aligned on a page boundary.

Upto now kernel code uses only 224 MB - PHYSTOP

panic() is a fn which will make the kernel stop & enters in an infinite loop.

* kmem is global structure
* A certain part of the node of linked list (4 byte ptr) is used to link
* Only the free chunks are chained together
* Rn: logical address = Physical address bcoz segmentation setup is done.

  Any address 2GB+ is mapped to 0 → 4 MB
* Each page in xv6 is 4KB
* So kinit1() creates a freelist.
  why free pages created? → Bcoz all next fns need this stop
* kinit2(P2V(4MB), P2V(PHYSTOP))
* Rn we are living in a world where there are 3 entries in page directory. 0 → 4MB only
  Hence kinit1(end, 4MB)
  ↑ max is 4MB only

* kvmalloc() → does mem setup for scheduler

* lcr3 → present in x86.h
In x86.h → these are wrappers around assembly code

cr3 → base of page directory

* setupkvm() returns new kernel page directory address.

* The lcr3() won't get called. It is basically an inline fn. It's code is copy pasted directly

* setupkvm() is called from many places.

* In kvmalloc we are going to map certain regions of logical memory to certain regions of physical mem

* In XV6:
For process: 0 → 2GB
For kernel: 2GB onwards

* PHYSTOP → limit on physical memory currently (224MB)

* kmap[] tells that KERNBASE should be mapped from 0 to EXTMEM

* ending virtual address
= physical end = physical start + vir

* come back to setupkvm

Goto kalloc() → it gives us a 4K frame.
kalloc() picks up first frame from freemem list

kfree() → puts a frame on free frame list

* Remember PDE and PTE from x86 mem management.

Every PDE entry points to a page table

* memset(pgdir, 0, 4K)
as everything is set to 0
the PS bit in PDE is also 0.
So it says that it's 4K page

* mappages() → creates entries within page dir & page table

mappages() job is to set up entries in page directory & also the page tables. 2 way paging

~~mappages~~ Go to mappages

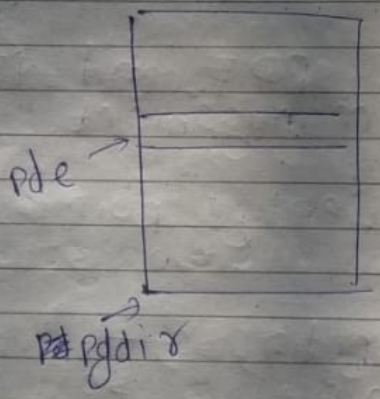
PGROUNDDOWN(a) aligns address downwards on a page size

virtual addr is a
Physical addr is pa

* walkpgdir() returns a pointer to a page table

If alloc = 1 then creates new page tables

PDX( ) → gives page Directory Index (10 MSB bits)



*pde → particular entry

* Entries in a Page directory & Page tables are PHYSICAL address

PTE_ADDR → gives address of page table entry

* Initially *pde & PTE_P is false as we memsetted to 0.

Go in else part
pgtab takes 1 page using Kalloc( ).

Now we will set up page table

* walkpgdir() checked if va existed in pgdir

* After kvmalloc() in main we switched from 4 MB pages to 4K pages & switched to 2 level paging.

* After kvmalloc() we can access memory upto PHYSTOP
∴ kinit2() uses 2nd arg as PHYSTOP

* Goto seginit() from main

see struct cpu

In seginit() again mapping is same 0 → 4GB

only diff is that here UCODE has privilege as DPL_USER
UDATA has privilege level DPL_USER

So now the processes will use only the UCODE & UDATA

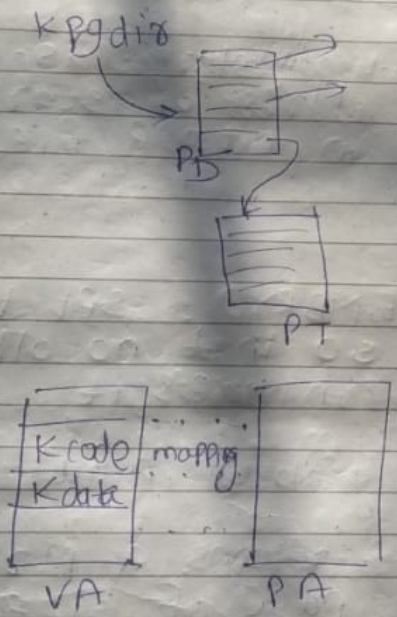
2/3/2022 O.S.

kinit1 } create free frame list
kinit2 }

kalloc, kfree } managing free frame list

KVmalloc → setupKvm

setupKvm():



Q. Can the seginit() be moved to line 35?
As userinit() starts first user proces

* Goto userinit() → creates init process by hand

extern char _binary_initcode_start[];

→ This _binary_init_code can be seen in kernel.sym
→ initcode.S, see in Makefile.

* we are going to create a process called initcode.
The initcode creates init.

* _binary_initcode_start : is starting address of start fn in initcode file.

* allocproc():

p->state = EMBRYO : set to embryo so that no other process can access it

nextpid is global initialized to 1

Kalloc() → returns a ptr to a page of size 4 KB

* we have to create a trapframe on kstack once we return from kernel we want to simulate returning from a trap

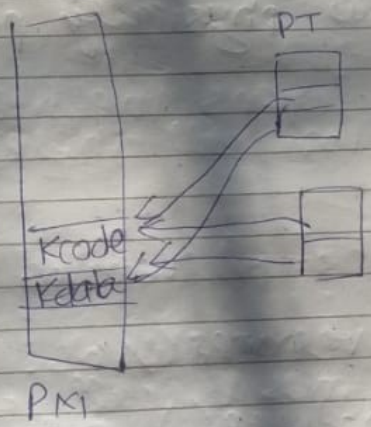
* trapret fn is present in trapasm.S

* Draw diagram of what allocproc() is dng

* initproc is global pointer to init process

* setupkvm() maps kernel code & kernel data & creates page tables

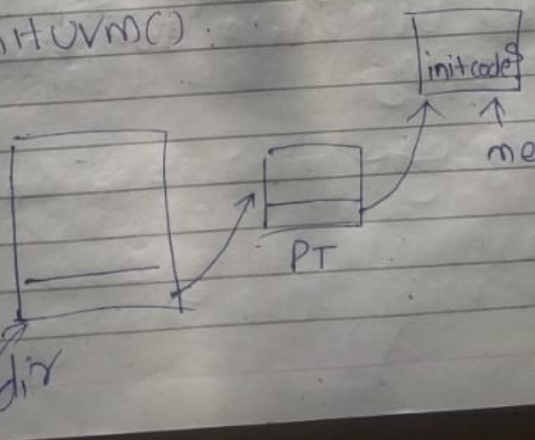
→ The kernel code & data already exists in Physical mem. only the mapping is done in setupkvm



* inituvm → init uses virtual memory

* mappages() → create page table mappings.

* In inituvm():

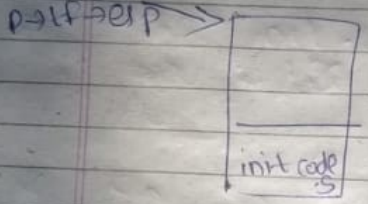


p→sz is PGSIZE But actually size stored is 0x002C

* SEGUCODE & SEG_UDATA are 3 and 4 hex in gdt the 3rd & 4th entries correspond to user code & data

* p → tf → esp = PGSIZE

so esp now physically points to

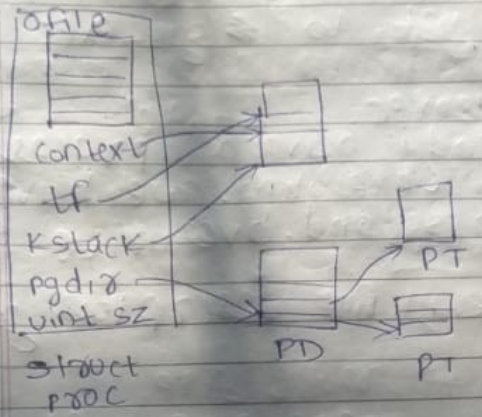


* Skip p→cwd = namei("/")

* After scheduler() ~~schedul~~ calls swtch() on initcode we goto forkret.

03/ Mar/2022 O.S.

Code of exec :

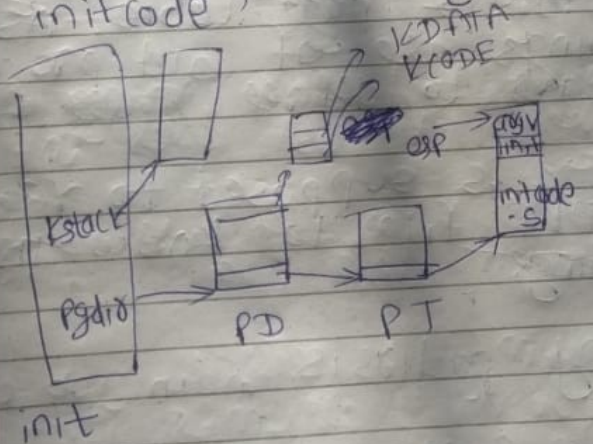


* Goto initcode.S

→ push $argv
push $init
push $0

→ see what is argv and init

where is this pushing happening in initcode?



→ From initcode.S we made int 64
So now stack changed.

How esp points to kstack

How do we get to sys_exec?

→ Goto sysexec.
→ We already have a process which is executing. We have to clear this and overwrite everything

* On a fork, file descriptors get copied.

* The init.c opens the files 0, 1, 2 So after fork all these will be duplicated

* So, on exec: pid remains same.

* exec() never returns. So exec() must call sched().

* We have to change the page directory & page table. As the mappings are of old process. We have to kfree() all frames.

exec() is supposed to read the ELF file to know the mappings

* Changing pgdir is main job of exec

* Read code of sys_exec

* see how exec() is called in sh.c.

* In syscalls we have fns like argstr() argint(). Bcoz currently we are in kernel stack. And our args are pushed on ~~sys fn~~ appln stack

we can get older esp from trapframe This was pushed by hardware

argstr(0, &path) means fetch 0th argument in path

Args are pushed onto the stack right to left

* Each process needs to map kernel pages also as kernel needs to access user pages

* argstr(0, &path): This path will point to the user stack. This is a bad practice.

* exec() does the actual work.

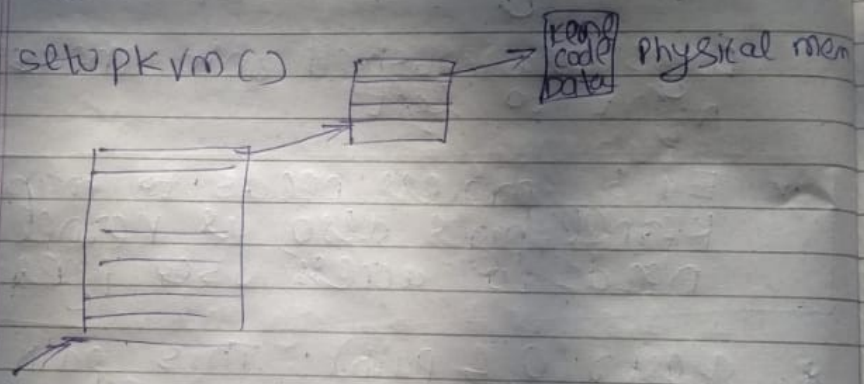
* Ignore begin_op()

* namei() finds file on the disk & creates an inode data structure

inode is unique repr of file.

* readi() → reads file

* setupkvm()



* mappages() doesn't ~~bring~~ allocate memory! It just does the mapping

* The offset in elf file is byte-wise offset in the file

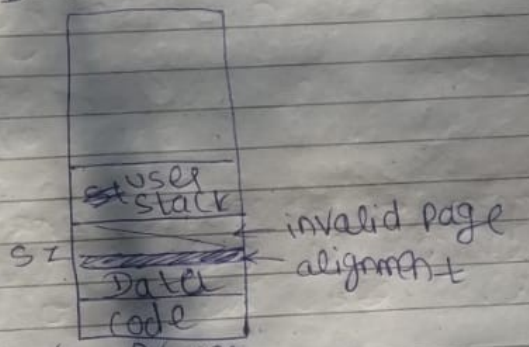
* allocuvm() is called
  → allocuvm() assumes that process is of some size & increases the memory

loaduvm() → loads elf contents into memory allocated by allocuvm()
ip → inode pointer

* what does walkpgdir do:
Now the argument given to walkpgdir is '0'.

* P2V() is done bcoz the address in page table is physical address.

* PGROUNDUP()



VA Process map

→ The invalid page is added only for detecting user stack violation

* exec() also pushes the argc and argv on the user stack. Bcoz they must be made available to main() of new process