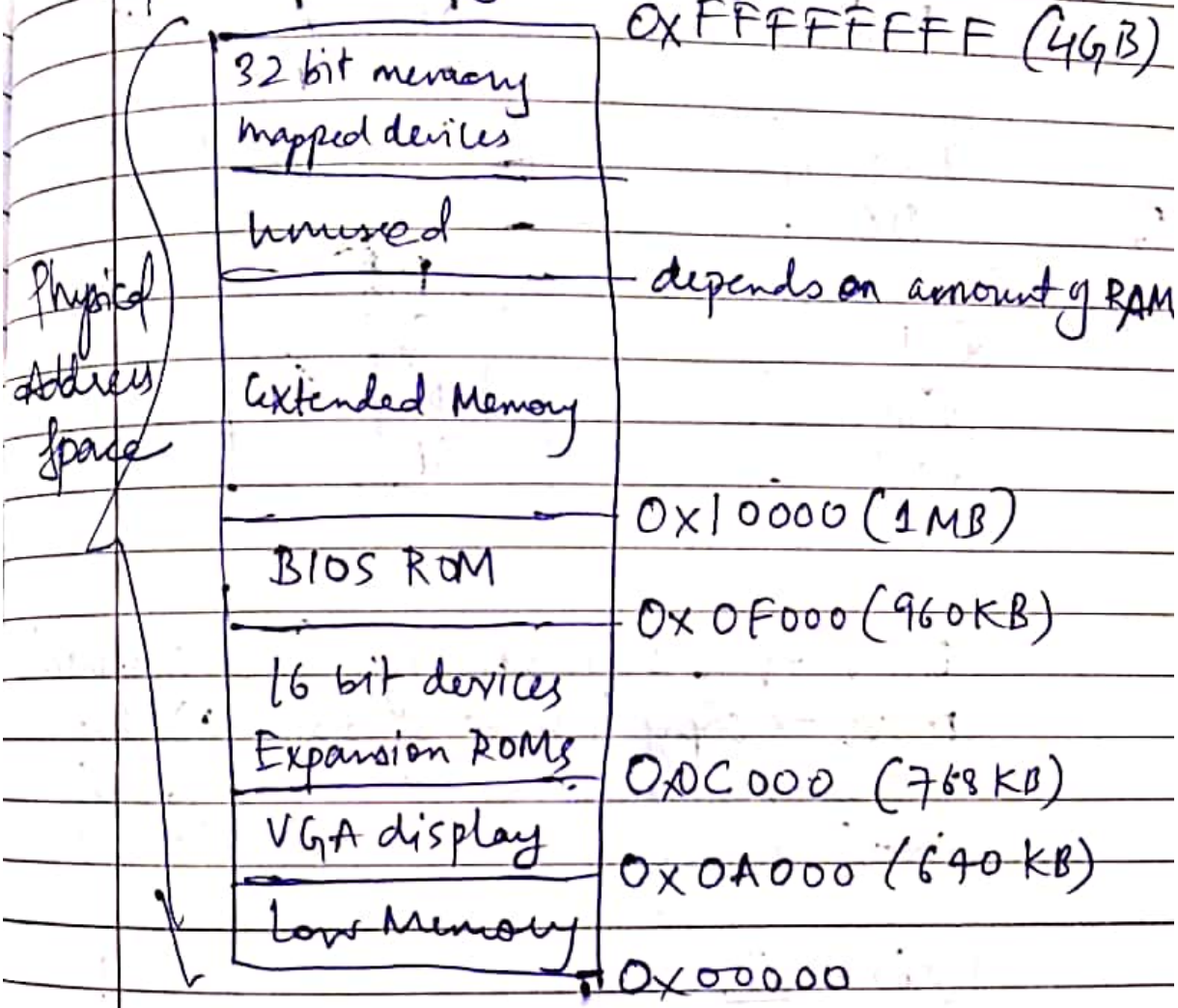
Page No.
Date

2/1/19 (Lec 1)

In 32 bit architecture, we have 4GB limit of RAM as addresses can ~~also~~ only be $2^{32} \simeq 4GB$. (Each address space points to 1 byte = 8 bits).



X86 has 8 32-bit registers :

EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP

EAX → return value generally in it

EDX → used for return value if 64bit

ESP → stack pointer

EBP → base pointer

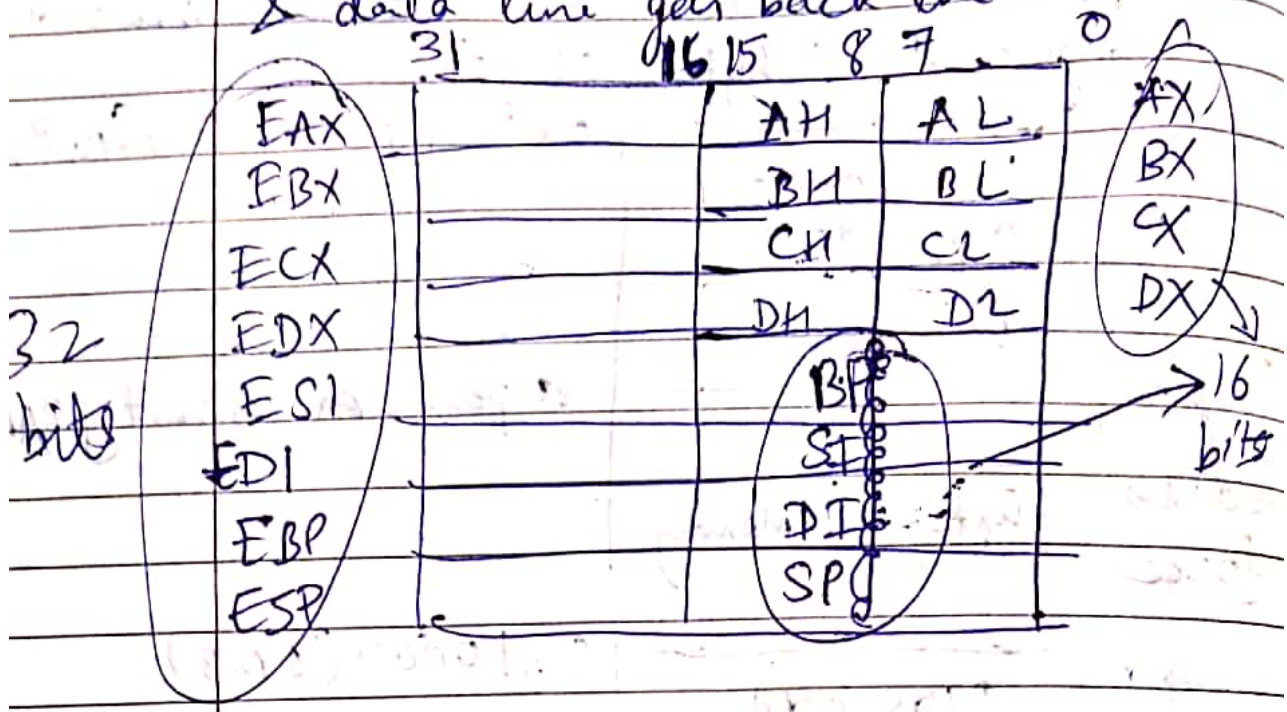
GCC follows ATAT syntax :

operation src, destination

add %eax, %ebx ⇒ %ebx = %ebx + %eax

Scanned by CamScanner

Applications need more data than the registers. So CPU ~~soo~~ sends address on data line to access main memory & data line gets back the data.



mov → if unambiguity (when reg. used)
movl → copy 4 byte of data (32 bits)
movw → copy 2 " " " (16 bits)
movb → copy 1 " " " (8 bits)

X86 AT&T	C-like
mov %eax, %edx	edx = eax
mov $0x123, %edx	edx = 0x123
mov 0x123, %edx	edx = *(int32*)0x123
mov (%ebx), %edx	edx = *(int32*)ebx
mov 4(%ebx), %edx	edx = *(int32*)(ebx+4)

~~Pause~~ ⑤

Pintos Stack size = 4KB → so don't allocate too much on stack like array of 1000 size in local fn. Instead allocate it dynamically

Linux Stack size = 8KB

```
foo (int a, int b)
{ int arr[1000];
    ↳ not recommended as local variables stored on stack and
}
```

→ instead do: int *arr;
arr = (int*) malloc (1000 * sizeof(int));

Don't forget to free if you malloc.

```
while (1) {
    len = length (instruc^n);
    execute (instruc^n);
    instruction + = len;
}
```

} move to next instruction

EIP → instruction pointer / PC
- ↳ points to next instr^n in memory
- ↳ incremented after every instr^n
- ↳ modified by call, ret, jmp & condj.

EFLAGS → flags register

Scanned by CamScanner

Page No.
Date

X86	X86 AT&T
Immediate operands	push $4
~~Regs~~	movl $0xd00a,
Register operands	preceded by %
Argument order / the eg. adds address of C var. foo to reg: EAX	Op. source, dest. eg addl $_foo, %eax
Single size operands	b, w, l use
address memory pointed by a register	(%eax)
address a C var. foo	_foo
address a var. offset by a value in reg.	_foo(%eax)
address value in array foo of 32 bit ints	_foo(, %eax, 4)
*(p+1)	1(%eax)

Segment registers: hold 16 bit segment selectors. It identify a special pointer that identifies a segment in memory.

CS : code segment selector

SS : stack " " .

ES, FS, GS, DS : data segment selectors, provide efficient ways to access data.

EFLAGS : CF, PF, ZF, SF, OF, AF
0 2 6 7 11 4

Scanned by CamScanner

Page No
Date

imm.

Memory Operand : Segment: disp [base, index, scale]

address = base of segment + disp + base + (index * scale)

)'s comp. offset.

base: any reg.
index: any reg. except EBP
scale: 1, 2, 4 or 8

① CS: code segment reg. for instruc^n fetches
② SS: stack seg. reg. for stack pushes & pops as well as references using ESP or EBP as a base reg.
③ DS: for all data references except when relative to stack / string destination
④ ES: for destination of string instruct^s.

Control transfer instr^ns:

① (l) jmp address
② (l) call address (save EIP on stack & jmp to add.)
③ ret (ret. to EIP loc^n saved by call)
④ leave (restore EBP from stack & pop off stack frame)
⑤ j{e, ne, l, le, g, ge, ...} address if =0
⑥ loop address (decrement ECX or CX; jump)
⑦ rep (repeat string op^n prefix)
⑧ int number (software interrupt)
⑨ iret (return from interrupt & pop EFLAGS from stack)

Inline Assembly Code

asm volatile ("assembly instr^n");
asm volatile ("cli"); to clear interrupts

Scanned by CamScanner

General Format: asm [volatile] ("statements": output-regs : input-regs : used-regs);

eg input : "a" (foo), "c" (bar)
so it assigns/loads for variable in eax & bar var. in ecx or "r" or "q" for compiler to choose.
We can let compiler choose o/p reg. for us by saying "=q" or "=r", so the first reg is accessed by "%0" & 2nd by "%1"...

7/1/18 (Lec-2)

For some variables, compiler manages the memory (RAM), whereas for some variables, the memory is managed by programmer.

Allocator → for memory management.

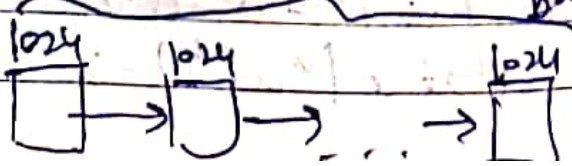
1. Buddy Allocator [like malloc/dynamic alloc]
→ It takes a range of memory addresses & manages it.

8 byte
16
32
64
128
256
512
1024

1499

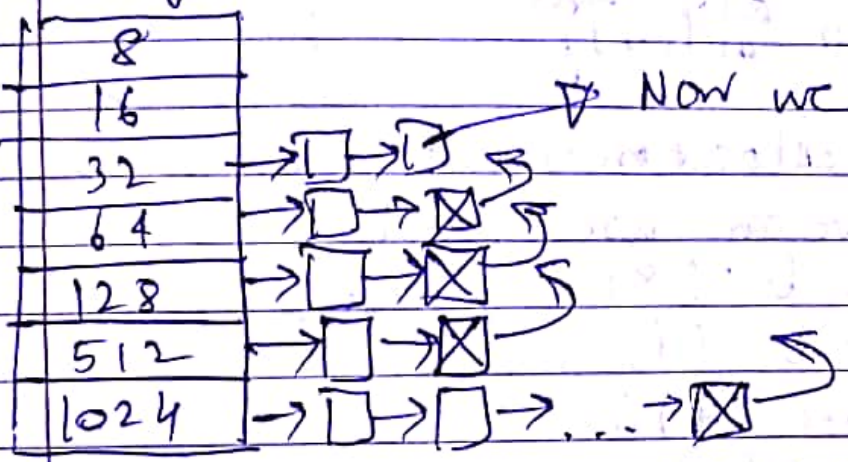
Say address range given is 1024 to 102400
So first the allocator adds 1024×100 blocks of 1024 size in chunk. When user demands it, it is taken out from list & given & when user frees it, it is put back to list.

99 blocks = based on memory range given

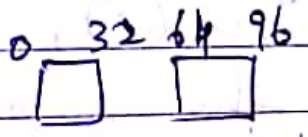


Scanned by CamScanner

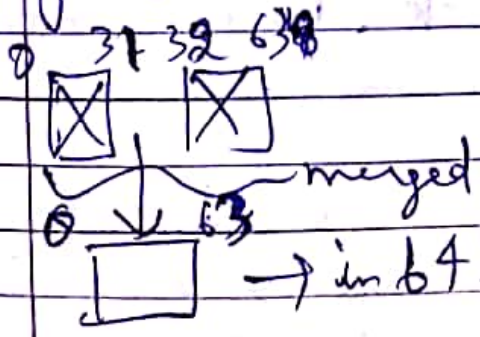
To allocate 32 byte chunk, we'll look just below it i.e. if we've 64 byte block. If we've it, then we divide it into 2 and remove it from 64 & put in 32 list & use one of these.
Else we move further down to take 64 bytes & repeat this recursively..
Only two contiguous address blocks can be merged with using free & then the merged block is moved down.



Eg: if in 32 block:



can't be merged as not contiguous



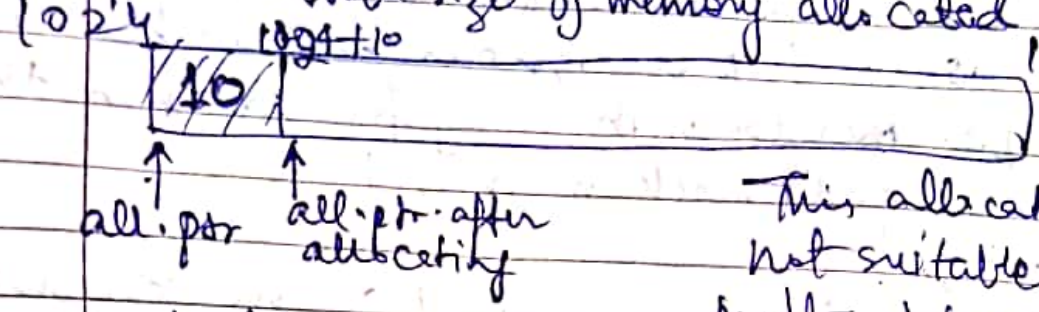
Merging takes place at free. If it was not the case, then we may have fragmentation prob. i.e. we may have a lot of 8 byte blocks in 8 byte chunk but these are not contig. So we can't merge to allocate 16 byte chunk.

Page No.
Date

local vars.

2. Bump Allocator (for stack / static alloc^n)

Take some memory range : 1024 to 102400
So this allocator has one allocation pointer.
When we allocate, we bump the allocator pointer with size of memory allocated



This allocator is not suitable for malloc & free as deallocation is not so easy in this.

We can use it as:

```
malloc(8);
malloc(4);
free(4);
free(8);
```

} used sequentially

So this is used for local variables in fact this is the stack. Last allocated gets freed first.

When an application is loaded on RAM, the loader also allocates space for buddy allocator (heap) & bump allocator (stack) say its (Bump) 10000 to 20000 & 1000-2000 (buddy)
Bump allocator is usually of small size.
Linux uses 8192 bytes & Pintos uses 4096 bytes
Bump allocator is implemented using stack & the

direction of alloc^n pointer is reversed to make use of faster x86 instr^{n}s.

On x86, esp reg. is reserved for alloc^n pointer. When the program is loaded, the loader sets esp to the end of space reserved for buddy allocator (2000). An allocation decrements the pointer.

```
main()
{ int a=0;
  int b=10;
  return a+b;
}
```

In this case esp is set to 2000.
After allocation of a, esp = 1996. So the address of a i.e. &a = 1996.
After allocation of b, esp = 1992, &b = 1992.
When a func^n returns, the program automatically deallocates the memory for local variable. In this case, first b will be deallocated by simply adding 4 to the alloc^n pointer. Then it deletes a by again adding 4 to the stack ptr.

<u>How are func^n arguments passed?</u>
Argument passing is also done via stack. When a func^n call another func^n, it allocates space for all the arguments & destroys them after returning from the func^n.

Scanned by CamScanner

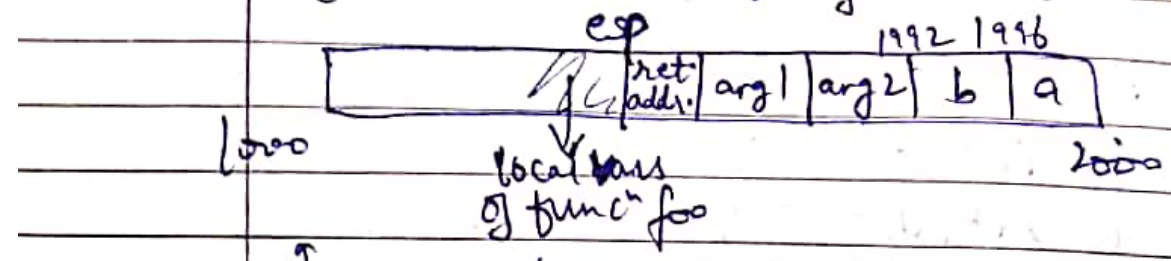
Doubt → Below, dealloc^n done by caller or callee?

Page No.
Date

```
main()                          foo (int a, int b)
{ int a=0;                      {
  int b=0;                        return a+b;
  foo (a,b);                    }
}
```

→ Pass by value by default.

→ Arguments simply pushed to stack in reverse order before call instr^n.



In pass by ref., address is passed.
Deallocation is done by Callee.

pushl %eax	:	subl $4, %esp movl %eax, (%esp)
popl %eax	:	movl (%esp), %eax addl $4, %esp
call 0x12345	:	push %eip movl $0x12345, %eip
ret	:	popl %eip

★★★
★★★

★ An entry to func^n foo (just after call)
→ %eip points to first instruc^n of foo
→ %esp+4 points to first arg.
→ %esp points to return address

Scanned by CamScanner

Page No.
Date

* After ret. instruc^n
→ %.eip contains ret address
→ %.esp points to arguments pushed by caller
→ Called func^n may have trashed arguments.
→ %.eax (& %.edx if 64 bit return) contains return value (or trash if func^n is void).
→ %.eax, %.edx, %.ecx may be trashed.
→ %.ebp, %.ebx, %.esi, %.edi must contain content from time of call.

Caller saved : %.eax, %.ecx, %.edx
Callee saved : %.ebp, %.ebx, %.esi, %.edi

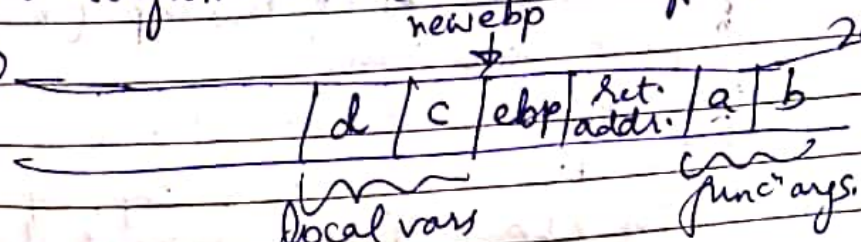
Frame pointer (ebp) : needed so that name of an argument, local var. doesn't change throught func^n body.

Func^n prologue : pushl %.ebp
movl %.esp, %.ebp } Save ebp & Start new ebp of new func^n

Func^n epilogue : movl %.ebp, %.esp
popl %.ebp } restore ebp after making esp point to frame pointer.

not strictly needed as compiler can calculate.

→ all arguments are at const. offset from ebp.



Page No.
Date

Implementing Backtrace.

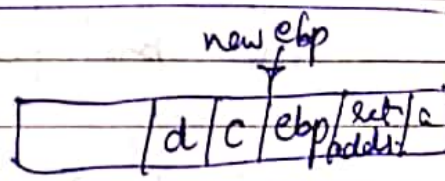
→ with the help of ebp as popping ebp, esp points to ret addr.

```
do {                    //(assuming ebp is unsigned)
    printf("%p\n", (char*)ebp[1]);
    ebp = ebp[0];
  } while (ebp)
```

OR

```
ebp = ebp_reg;
while (ebp) {
  ret_addr = *(ebp+4);
  printf("ret address %d", ret_addr);
  ebp = *ebp;
}
```

9/1/18 Lec3



foo(int a, int b)
{ int c, d ...

1. Save args (caller)
2. save ret addr (by call instr.)
3. Save ebp (prev. ebp of caller) (push ebp) (by caller)
4. move ebp to new ebp (mov esp, ebp)
5. push local vars

ebp used to print chain of back traces with ret addr

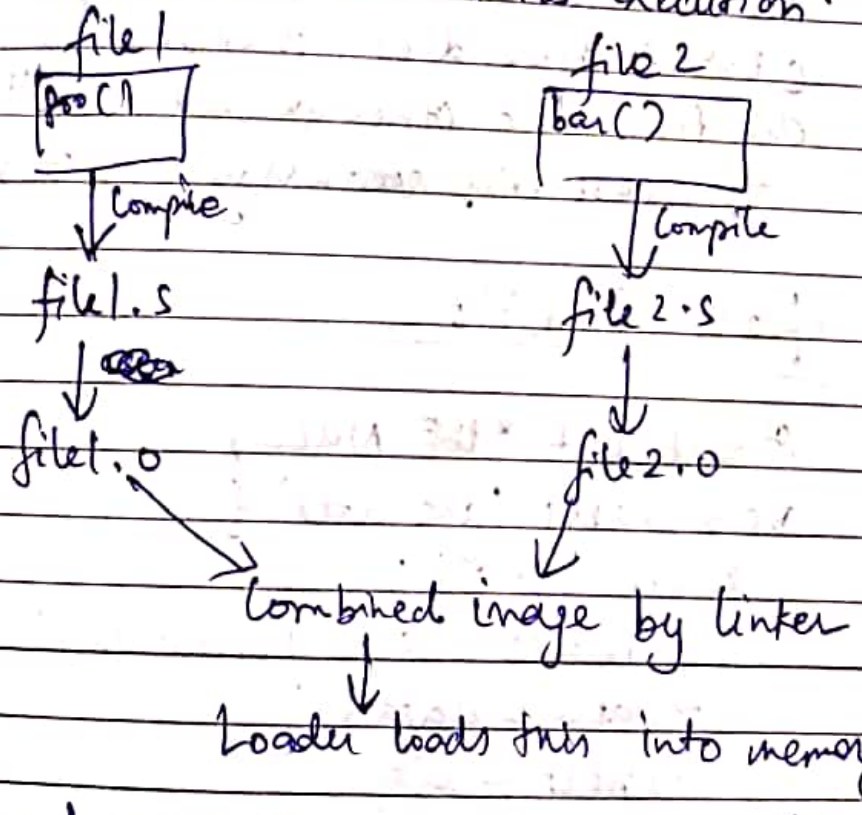
6. Just before return, mov ebp, esp
7. pop ebp (restore original ebp of caller)
8. ret (pops ret. addres to eip & so goes to caller)

Scanned by CamScanner

Page No.
Date

Compiling, linking & loading

1. Preprocessor takes C source code, expand #includes & #defines, produce C source code.
2. assembly from C source code by compiler
3. Assembly makes .o files from assembly file.
4. Linker takes multiple .o files, produces a single program image
5. Loader loads the program image into memory at run time & start its execution.



Thread → A stream of sequential instructs is called a thread. An application can have many such threads.

Eg:
```
int print_mul(int arr[], int len);
int print_avg(int arr[], int len);
int compute() {. . . .
    create_thread(printmul, arr1, 100);
}
```

The above application finishes when all the threads terminate.

** Whenever we make a new thread, he make a new stack for it & diff. registers, but they share the heap.

Quiz: That is why you should pass parameters to a threaded func with dynamically allocated memory because if you pass a ref. to a local var, like array, when the current thread dies, this variable will also die thus the other thread won't be able to access this array as stack destroyed.

Race Conditions

```
struct list *list = NULL;
void insert (int data) {
    struct list *l;
    l = malloc (sizeof (struct list));
    l->data = data;
    l->next = list;
    list = l;
}
```

Race Cond^n

Thread 1	Thread 2
l->next = NULL;	l->next = NULL;
list = l;	list = l;

Multiprocessor

Scanned by CamScanner

Page No.
Date

Single Processor : → Multiple threads on single processor machine by scheduling.

Thread 1 :
l→next = NULL ;
schedule() ;
Thread 2 :
l→next = NULL
list = l ;
schedule() ;
Thread 1 :
list = l /* eventually list has only one element */

→ implicit task scheduling by OS

To prevent this, we may use locks :
lock-acquire() ;
l→next = list ;
list = l ;
lock-release() ;

→ Prevents concurrent execution of a code

Interrupts (used for scheduling)

→ It is a signal to a processor.
→ Whenever a processor recieves an interrupt, it stops the current execution & jumps to a different routine.
→ Every interrupt has a vector no. from 0 to 255 as it may come from timer device or network device etc.
→ There is a table of these vector no's along with their corresponding handlers. This

Scanned by CamScanner

table is called IDT → ~~Instruction~~ Interrupt Descriptor Table. This table is set up by OS. Suppose vec. no. is 10 which corresponds to timer interrupt. Let the timer interrupt come every 10ms. Every vector no. has a handler. This handler has the address of the func^n to which processor has to go upon recieving interrupt with that particular vector no. This table is in RAM.

LIDT → x86 instruc^n which sets this table address (maybe) in RAM

int vec-no → throws interrupt. (x86)

→ Suppose we're in foo. Then timer interrupt comes & we go to timer interrupt handler. Then after exiting this handler, how do I return / go back to where I was?

So whenever interrupt occurs, the process pushes the flags and program counter to stack automatically.

iret instr^n pops these 2 things back.

We also need to save the reg. values. So we need to do this in software. So in the timer interrupt handler, all the reg. needs to be pushed to stack & before returning, we need to pop these values from stack.

Page No.
Date

Use of timer interrupt:

① implement sleep for 100 ms.

→ So we wait till 10 timer interrupts. i.e. cpu ticks.

```
tick = get_cpu_tick();
while(get_cpu_tick() - tick < 10)
{ thread_yield(); → to wake from sleep.
}
```

this is executed every time

How Scheduler works? (Round-Robin Scheduler)

Scheduler keeps the list of all threads called ready list in fifo list. And after every timer interrupt / after every n timer interrupt, it will remove current thread from running, add it to end of ready list & pop from ready list the thread at start to execute it.

```
timer_interrupt();
thread_yield();
```

What does thread_yield do?

① disables interrupts to avoid race condition during the manipulation of ready list & the entire schedule() operation.

② adds the current thread to the end of ready list.

③ Pops the next thread to run from the front of ready list (schedule() in Pintos)

Scanned by CamScanner

Page No.
Date

(4) saves all registers on stack (switch_threads() in pintos)
(5) saves the stack pointer of current thread (switch_threads())
(6) restores the stack pointer of target thread (switch_threads())
(7) restores reg. of target thread from the stack (switch_threads())

Threads can also voluntarily yield i.e. to wake from sleep (like prev. example)

E.I.P is automatically saved upon call to schedule();

How to implement locks?

(1) One way is to disable interrupts in critical section so that no other thread can change thread list at that time.

Single processor

```
void insert(int data)
{ struct list *l;
  l=malloc(sizeof(struct list));
  l->data = data;
  oldlevel = disable_interrupts();
  /* scheduler can't run when interrupts are
     disabled */
  l->next = list; list = l
  intr_set_level(oldlevel);
}
```

In multiprocessor, also acquire lock with disabling interrupts.

→ Do not disable interrupts for long time as. Unnecessarily blocking other threads is not suitable for an interactive application.

→ Interrupt handler are also very small so they don't disable interrupts for a long time. They also don't do any IO generally.

→ Also, we can't do IO if interrupts are disabled as most devices require interrupts to be enabled.

→ So disable interrupt when critical section is small.

What to do in other cases? (When big critical sec^n or can't disable interrupts)

Semaphore is used.

```
parent() {
  struct semaphore s;
  sema_init(&s, 0);
  create_thread(child, &s);
  sema_down(&s);
}

child(struct semaphore *s) {
  sema_up(s);
}
```

↳ Here parent thread wants to wait till child has finished execution.

Scanned by CamScanner

```
struct semaphore {
    int value;
    struct list waiters;
};
```

→ 0 or more value

→ list of all thread waiting on this semaphore (not on ready list).

```
sema_init (struct semaphore *s, int val) {
    s->value = val;   //Non-negative no.
}
```

```
sema_down (struct semaphore *s) {
    old_level = interrupts_disable();
    while (s->value == 0) {
        list_insert(&s->waiters, thread_current());
        schedule();
    }
    s->value--;
    interrupts_set_level(old_level);
}
```

remove current thread from ready list too.

→ puts thread in wait-list until the semaphore value is zero. So the thread is not scheduled until sema_up is called which puts in back in ready list and incrementing the s->value.

```
sema_up (struct semaphore *s) {
    old_level = interrupts_disable();
    if (!list_empty(&sema->waiters)) {
        thread = list_pop(&sema->waiters);
```

Scanned by CamScanner

```
        add-thread to ready list
    }
    s->value++;
    interrupt-set-label (old-level);
}
```

How to make locks with semaphore?

```
struct lock {
    struct semaphore s;
};
```

→ s->value = 1 → lock is available

→ s->value = 0 → lock is not available

```
lock-init (struct lock *l) {
    Sema-init (&(l->s), 1);
}
lock-acquire (struct lock *l) {
    Sema-down (&(l->s));
}
lock-release (struct lock *l) {
    Sema-up (&(l->s));
```

7/1/19 Lec-4

create_thread (func-name, args); → creates a new thread with new stack & new registers but same heap as that of parent. Also global variables are shared.

Scanned by CamScanner

Page No.
Date

How to run multiple threads on same process?

Ans: With the help of schedule func^n.

→ struct list *ready_list; ← list of all threads to be scheduled

→ struct thread {
    void *esp; → address of stack pointer of that thread.
};

→ Create_thread () {
    struct thread *t = malloc (sizeof...);
    t→esp = malloc (4096) + 4096;
    enqueue (ready_list, t);
}

then added as stack is down growing so we need top address.

→ schedule (struct thread * cur_thread) {
    enqueue (ready_list, cur_thread);
    next_thread = dequeue (ready_list);
    push %eax
    ...
    push %edi
    } all regs saved of current thread into its stack
    cur_thread→esp = cur_esp;
    esp = next_thread→esp;
    pop %edi
    ...
    pop %eax
    } pop reg. of next thread from its stack
}

Scanned by CamScanner

Page No.
Date

Q Why schedule() doesn't save the current pc (or eip)?

A We don't need to do it as schedule is called by a func^n call & in x86, func^n call automatically pushes return address to stack so that eip is saved effectively by default. So when we return there from schedule i.e. when the thread gets a chance to run, it starts ~~from the~~ from that return address.

eg

```
foo(){
    ...
    schedule()
    ...
}
```

```
bar(){
    ...
    schedule();
    ...
}
```

Race condition.

```
struct node * head = NULL;

insert (int val) {
    struct node * n = malloc(..);
    n -> val = val;
    n -> next = head;
    head = n;
}
```

let us refer to Thread1 as T1 & Thread2 as T2:

Race cond^n to loss of data

T1 : n -> next = NULL;
     schedule();
T2 : n -> next = NULL; head = n;
     schedule();
T1 : head = n;

Scanned by CamScanner

Page No.
Date

① So to prevent this let us apply locks:

```
insert (int val) {
    .....
    lock_acquire();
    n->next = head;   ] Critical
    head = n;         ] section
    lock_release();
}
```

So now only one thread can manipulate the head. The implementation of lock is:

```
lock_acquire(){
  status = interrupt_disable();
}

lock_release(int status) {
   set_interrupt_status(status);
}
```

Also won't work in multiprocessor

But this isn't a good implementation as this will not allow scheduling even if other thread doesn't ~~take~~ manipulate critical section so it will block.

②

```
struct lock{ int value; };
lock_acquire(struct lock *l){
    while (l->value == 0){ }
    l->value = 0;
}

lock_release(struct lock *l) {
    l->value = 1;
}
```

Scanned by CamScanner

Page No.
Date

Fault with this is the scenario :

T1: lock→value was one so it came out of while loop ~~and~~ but still hasn't executed lock→value = 0 to acquire it fully. Just then, schedule was called so we got to T2:

T2: This also sees lock unacquired as lock→value still one as prev thread was switched before it could make it zero. So this also comes out of while loop & acquires lock.

→ Thus both get lock!! (Race condition within lock implementation)

(3) To overcome above problem, let us disable interrupts before checking while loop & re-enable it after we've changed value to 0 so that no thread switching can take place in between.
But again, this has issue of infinite loop in the while loop since no other thread can run & thus value of lock will never get back to zero (in case this thread has to wait).

(4) To overcome above, we call schedule() within ~~wait~~ while loop so that while waiting, the other thread can free the lock so that the waiting thread can acquire it. This is

Scanned by CamScanner

Page No.
Date

Perfectly fine sol^n but this can waste some cpu cycles as this func^n scheduler may call this thread again which is waiting for some other thread to free the lock. So until some other thread frees it, no point in scheduling to this thread as it is in a while loop.

(5) The final sol^n is to use a waitlist.

```
struct list *wait_list;
lock_acquire (struct lock *l) {
  status = interrupts_disable();
  while(l->value == 0) {
    remove(ready_list, thread_current());
    enqueue(wait_list, thread_current());
    schedule();
  }
  l->value = 0;
  set_interrupts_status(status);
}
lock_release (struct lock *l) {
  struct thread *t = dequeue(waitlist);
  enqueue(ready_list, t);
  l->value = 1;
}
```

(1) Similar thing can be done by a semaphore.

Scanned by CamScanner

Page No.
Date

Monitor lock (conditional variables)

```
char buf [BUF_SIZE];  /* Buffer */
size_t n = 0;         /* # of chars in buffer */
size_t head = 0;      /* buf index of next char to write */
size_t tail = 0;      /* buf index of next char to read */
struct lock locks;    /* lock */
struct condition not_empty;  /* Signalled when buffer is not empty */
struct condition not_full;
```

(1)
```
void put (char ch) {
    while (n == BUF_SIZE) { }
    buf[head++ % BUF_SIZE] = ch;   /* add ch to buff */
    n++;
}

char get (void) {
    char ch;
    while (n == 0) { }
    ch = buf [tail++ % BUF_SIZE];
    n--;
}
```

Here the put func^n adds character to the buffer. The get func^n consumes a character from buffer. The buffer size is fixed. If the buffer is full, put routine waits until get routine consumes a character. Similarly if buffer is empty, get routine waits until put routine adds a char to buffer. The buffer is

implemented as a fixed size circular queue.
WKT spinning (waiting in loop) isn't good option for all operations.
To handle this we've an abstraction called cond-wait & cond-signal. The cond-wait waits until a given condition is true & cond-signal wakes up a thread which is waiting on a given condition. So the code becomes:

(2)
```
void put(char ch) {
    while (n == BUFF_SIZE)
        cond_wait(&notfull);
    buf[head++ % BUFF_SIZE] = ch;
    n++;
    cond_signal(&not_empty);
}

char get (void) {
    char ch;
    while (n==0)
        cond_wait(&notempty);
    ch = buf[tail++ % BUF_SIZE];
    n--;
    cond_signal(&not_full);
}
```

This code is still incorrect as multiple threads are accessing shared data structure "n". So we use locks.

Page No.
Date

(3)
```
void put (char ch) {
    lock_acquire (&lock);
    ;   //same code in b/w
    lock_release (&lock);
}

char get (void) {
    char ch;
    lock_acquire (&lock);
    ;   //same code in b/w
    lock_release (&lock);
}
```

This also has a problem. Since a thread maybe waiting on some condⁿ while acquiring the lock, nobody will be able to signal the thread since that other code won't be able to run as it also needs the lock. This is called deadlock. To avoid this:

(4)
```
void put(char ch){
    lock_acquire (&lock);
    while (n == BUF_SIZE) {
        lock_release (&lock);
        Cond_wait (&not_full);
        lock_acquire (&lock);
    }
    ;//same code
    lock_release (&lock);
}
```

Similarly in get also, lock release before condⁿ & lock acquire after it.

Scanned by CamScanner

But even this is incorrect. Consider the case when get routine found that n is zero and it released the lock. Let us say at this point, "put" was scheduled ~~and~~ which added a char to buffer & waked up a thread which was waiting on that cond": "not-empty" (which is currently none as get ~~as~~ hasn't called cond-wait yet). After that get was scheduled and it called cond-wait. So even if ∃ some data in buffer at this time, get is still waiting because it missed a call (wakeup call) from the put routine. This is called lost-wakeup problem. This is due to the fact that releasing a lock and conditional wait is not atomic. To prevent this, cond-wait also takes lock as an argument & ensures the atomicity of lock-release & cond-wait.

(5) In final code:

Replace: { lock-release(&lock);
cond-wait(&not-full);
lock-acquire(&lock);

with: { cond-wait(&not-full, &lock);

Page No.
Date

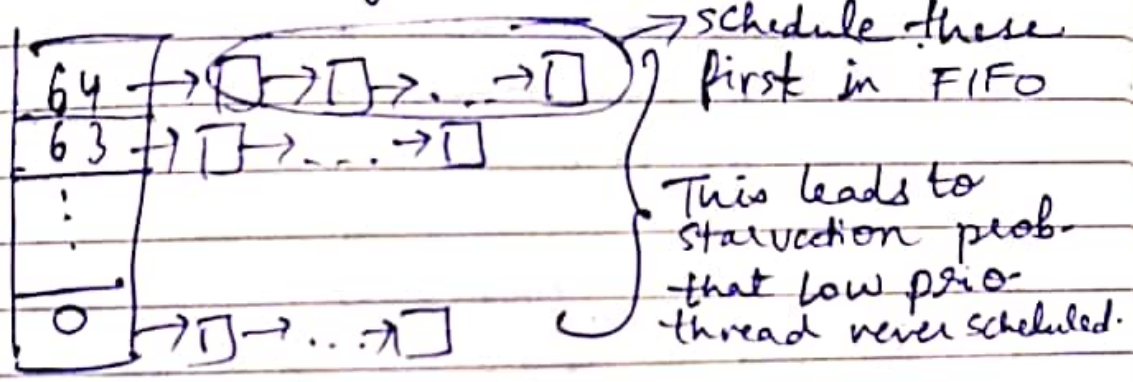
Priority Scheduling.

→ each thread is created with an initial priority.

→ let's say priority range is 0–64.
A higher value means high priority.

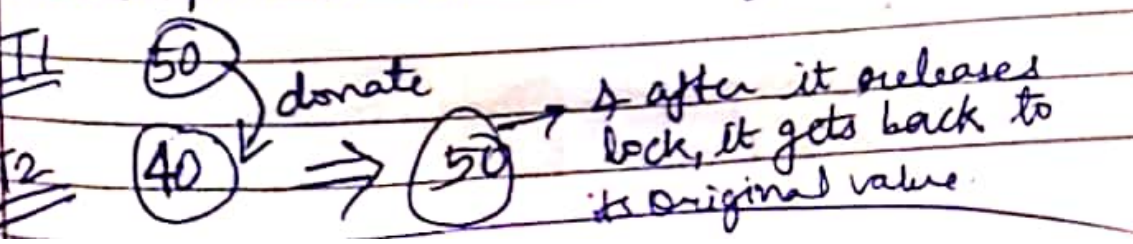
→ It will always run the high priority thread.

→ If threads have same priority then round robin scheduling i.e. fifo.



To prevent starvation, OS periodically inc. priority of thread (say 15 mins). ~~it is not scheduled~~ Also called aging.
So after it gets scheduled, the OS sets its priority to original value.

Priority Donation : If high priority thread is waiting for low prio. thread to release locks, the high prio. will donate its prio to low prio thread so that it gets scheduled fast.



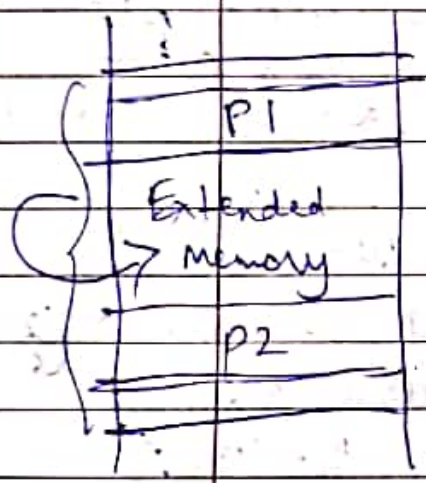
Scanned by CamScanner

16/1/19 Lec-5

Processes

→ They are containers of threads.
→ Each process start with one thread and can create more threads if needed.
→ Every process has its own quota in RAM.
→ A process can't see other process' data.

Each thread can see data of other thread of same process.
OS prevents read/write access to other process.



Application

→ OS is shared library
→ The library exports some interfaces that application developers can call to use OS services.
→ The OS enforces that an application can only call these exported routines to access data so that they can't interfere with other application's data. OS library prevents this.

→ Application is partitioned into 2 comp.:
• Application code (user programs)
• OS library
→ User programs are untrusted – there might be bugs that hacker can use to leak data.

Scanned by CamScanner

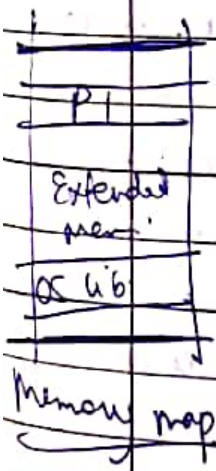
→ OS library is trusted.

How user programs use OS library?

```
create-thread {
    struct thread *t = malloc (sizeof(struct thread));
    t→esp = malloc(4096) + 4096;
    status = interrupt-disable();
    add-to-ready-list(t);
    set-interrupt-status(status);
```

→ OS func^n which is trusted to add to ready list.

→ Any programmer can't call add-to-ready-list func^n because it might lead to some failure. Because if add-to-ready-list is called by user program without disabling interrupts, then it might lead to something wrong. So how OS enforces this restricted access to user program?



→ User program & OS (kernel) lives in diff address space.

→ User programs cannot directly access kernel memory

→ Kernel can access entire memory.

→ User prog. have specific memory allocated to them which they can access (only that memory can be accessed).

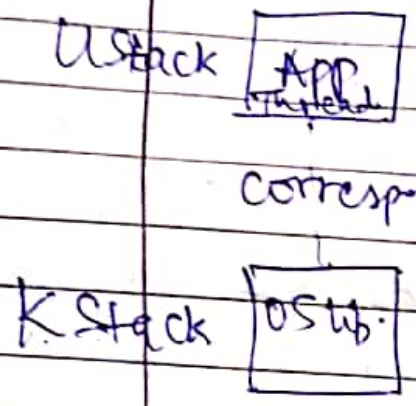
So how can a user prog. call create_thread? We'll discuss soon.

Scanned by CamScanner

Page No.
Date

Interrupt Handling (for user programs)

~~every interrupt~~ The kernel creates a kernel thread corresponding to every user thread. On every interrupt, the hardware automatically switches to kernel thread.



So kernel stack first pushes, esp (ustack + something ka address), eip, and eflags. iret pushes it back.

So why do we need this new stack? Why can't we push to user stack?

User can move $8, esp. So at this point say interrupt can come. And thus if we use user stack, then we'll push eflags & eip to there. But it might be that esp points to the loc^n where there is ready list or something so that user is not supposed to access it but with this interrupt, it can access.

Scanned by CamScanner

Software interrupt

int $100 → 0 to 255 → call interrupts with vec. no. 100.
Linux uses vec. 128 for sys. call. The syscall id is passed in some register or user stack.

Sys.call → to call a kernel routine stored at IVT table. (sys-call-handler)
So OS has many func^n to export so it assigns a number/sys call id with each func^n as sys call & pass it to user so user can call it.
Sys. call id → id to each func^n in sys call handler so that user can call proper func^n. So user passes sys call id in EAX reg & call the sys_vec. no. interrupt. So interrupt will look into EAX reg to see what func^n do user wants. So there can be more than 255 sys call ids.

Why? as we can't directly access kernel OS.

```
//idt[128] = syscall_handler;
int $128 → mov $0, eax → reg. has syscall id.
→ Sys_call_handler (int id)
{   switch (id) {
        case 0: create thread();
                break;
        case 1: create_process();
                break;
    }
}
```

Page No.
Date

Unix operating system

→ "shell" is the first user prog. created by Linux.

→ "shell" can create more processes.

Sys call eg → fork() → create new process which is replica of current proc.

```
main() {
    int pid;
    pid = fork();
    if (pid == 0) {
        /* child process */
    }
    else if (pid > 0) {
    /* parent process */
    }
}
```

Parent process does fork sys call & creates process called Child process. Every process has unique identifier. If id is zero, then it is a child process else parent process.

Let P1's PID = 100 & P2's PID = 200. So if we're in child then it returns 0 & if we're in parent, it returns 200.

Child inherits curr. working directory from parent.

Scanned by CamScanner

Page No.
Date

Exec() → command will be name of executable.
So this runs the new process/executable and it never returns & it runs on top of current prog. i.e. current prog. now becomes ls i.e. ls will overwrite current prog.

System calls

* int creat(pathname, mode) → open filename & it returns us a file descriptor to that file which can be used to read/write.
* write(fd, buf, len)
* read(fd, buf, len)
* close(fd)

mode i.e rwx?

```
fd = creat("temp.txt", 0666);
write(fd, "Hello World", sizeof("Hel..."));
read(fd, buf, len)
char buf[64];
close(fd)
```

read(fd, buf, len) → read len no. of chars into buf.
it also returns # of chars read from file

close(fd) // closes this fd & thus the particular fd is free.

* fds 0, 1, 2 have special meaning
  - 0 points to std. input eg keyboard
  - 1 " - " output eg console/terminal
  - 2 " - - error eg terminal/console

Scanned by CamScanner

Pseudo code of shell :

```
while (1) {        → no space after dollersign
    write (1, "$", 2);
    readcommand (0, command, args);
    if ((pid = fork()) == 0) {
        exec (command, args);
    } else if (pid > 0) {
        wait (0);
    } else
        printf("Failed to fork\n");
}
```

This fork a new shell & executes the command parsed on the child shell. Meanwhile parent waits.

exit(int status) → the process terminates with a given status.
→ wipe out all the memory occupied by the process.
→ release all resources.

int wait() → wait system call waits until the child terminates
→ the returnvalue is the exit status of the child.

Status = -1 usually means crash.
If OS killed some process, then OS exits with some specific status that'll tell parents.

Scanned by CamScanner

File descriptor table

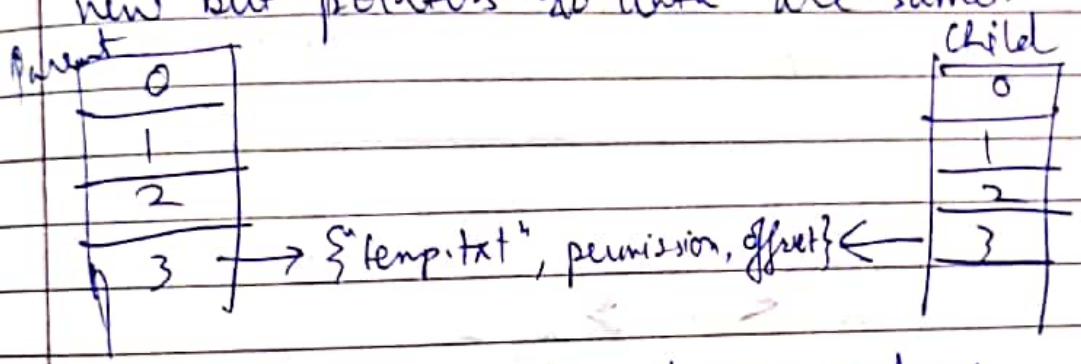
→ OS creates fd table for every process.

→ fd table are inherited by the child process.

→ So whenever we call open sys call, then it finds an empty value in fd table & puts the first empty value as the fd of new file opened.

→ Fd table holds name of file, permission to the file, and offset (needed in case of write).

Fd table is shared. As in, the tables are new but pointers to data are same.



So after making child, if parent closes fd = 3, then only parent's fd is closed but child will still have the entry in fd table in child.

→ OS always try to find lowest fd to allocate to open a new file.

→ Exec also inherits fd table as it is the same process. Same process doesn't change it's fd table. It does not create new table unlike child case. Because process is same as it overwrites.

Scanned by CamScanner

How does shell implements ls > tmpl ?

→ It closed fd=0 and then opens tmpl so fd=0 corresponds to tmpl & not stdout. So then it will write to tmpl instead of stdout because fd=0 pts to it.

```
pid = fork()
if (pid == 0)
{   close(1);
    1 = creat("tmpl", ...);
    execl(command) → ls.;
}
```

21/1/19 Lec-6

sh <script> tmp 1 → redirection operator

```
close(0);
open(script_file);
close(1);
creat("tmpl");
```

} run script in shell.

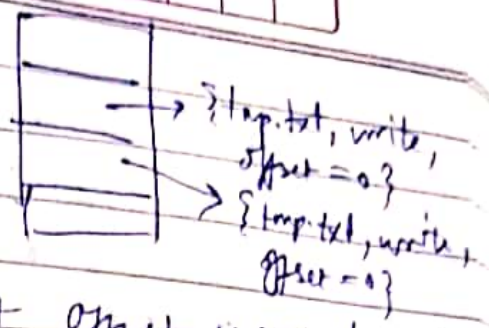
Shell command/ Program → reads from fd 0 & writes to fd 1

ls f1 f2 non-existant-f3 > tmpl 2>&1 → redirect stderr to fd 1 which tmpl

```
close(1)
creat("tmpl.txt");
```

Scanned by CamScanner

close (2)
creat ("temp2.txt")



Problem with this is that offset is not shared! So it will overwrite instead of appending.
So we've <u>dup</u> system call to copy fd.

dup(1); → points both to same entry.

Inter process communication – Pipe

→ Pipe has two ends. It is like a buffer managed by OS
→ Data written to input end.
→ Data are fetched from o/p end.

```
int fd[2];                  fd[1] → write end
char buf[512];              fd[0] → read end.
int n;
pipe(fd);
write(fd[1], "hello", 5);
n = read(fd[0], buf, sizeof buf);
// buf[] now has 'h', 'e', 'l', 'l', 'o'.
```

→ read waits till someone writes to pipe
It blocks till the writing fd is open.
After reading, data is gone from pipe.

Scanned by CamScanner

Page No.
Date

Inter process:

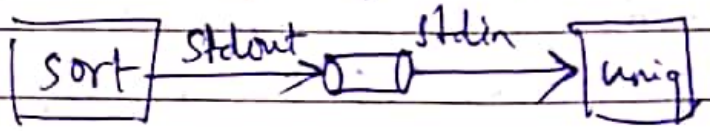
```
pipe(fd);
pid = fork();
if (pid > 0) {
    write(fd[1], "hello", 5);
}
else {
    n = read(fd[0], buf, sizeof buf);
}
```

How to run a series of programs?

```
sort < file.txt > tmp1
uniq tmp1 > tmp2
wc tmp2
rm tmp1 tmp2
```

→ puting all unique words from sorted o/p to tmp2 i.e. removing duplicate

This has too many file I/O & so this is very slow as we've to read write to disk.
So instead of doing this, we can use pipe.




```
sort < file.txt | uniq | wc
```

Scanned by CamScanner

Page No.
Date

```
pipe(fd);
if ((pid = fork()) == 0) {
    close(1);
    tmp = dup(fd[1]);
    close(fd[0]);
    close(fd[1]);
    exec(command1, args, 0);
}
else if (pid > 0) {
    close(0);
    tmp = dup(fd[0]);
    close(fd[0]);
    close(fd[1]);
    exec(command2, args2, 0);
}
```

→ Sort & uniq in above case are running at same time. So as soon as sort give o/p, uniq consumes it.

→ We need to close all writing fd's because even if one writing fd to pipe is open, then read call will be a blocking call, waiting for i/p to read.

Scanned by CamScanner

Memory Isolation

→ Processes have their own quota of RAM also called address space of the process.

→ If we only allow physical addresses, the compiler is unnecessarily forced to generate relocatable code as the compiler doesn't know actual RAM addresses in RAM

(Memory Map)

move $100, 0xF00 → This address might not be ~~able to~~ accessible to our process.

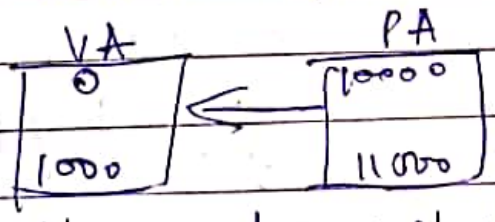
This isn't relocatable code.

→ Because of this restriction, compiler may not generate the most efficient code.

Virtual addresses

→ uniform among all processes

→ MMU (Mem. management unit) translates physical addresses to virtual addresses :



→ Virtual addresses always start at 0 & end at the size of memory located to process

→ Hardware converts VA to PA during mem. access

→ PA = base of process address + VA.

So compiler now knows to move $100, 0x

Something that & size of memory allocated to process

Page No.
Date

Segmentation

→ MMU does address translation in CPU.

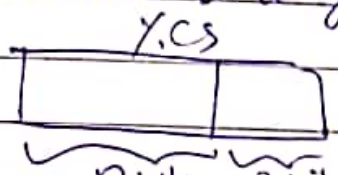
→ In addition to gen. purpose reg., x86 also have six segment reg. namely %cs, %ds, %ss, %es, %fs, %gs.

→ Every memory operand is prefixed by some segment register. (default is %ds)

→ Let us assume for now that %cs segment reg. is used for all mem. operands.

movl $100, (%eax) ≡ movl $100, %ds:(%eax).

→ Segment reg. contain 16 bit value that is used to index in a global descriptor table (GDT).



→ used to reference field of array

→ GDT is an array of structures of following type:

```
struct {
    unsigned base, limit, perm;
};
```

base is like a 32 bit address.

%cs : (%eax) → address of eax + base of %cs.

limit checks if VA is within the limit of VA available to the processor. Gen. protection fault is generated if process exceeds the limit.

size of reserved data

Protection rings

→ Why process cannot create their own GDT?

→ Protection rings

- some x86 instr are allowed or have a dif meaning in lower privilege mode
- OS runs in high privilege mode

23/1/19 Lec-7 Segmentation

```
movl $100, %cs:100
```

```
index = cs >> 3;
base = gdt[index].base;
unsigned *phy = base + 100 //phys. address
assert(100 < gdt[index].limit);
*phy = 100;
```

```
mov 16, %ax
mov %ax, %ds
```

index	base	limit
0	1000	10000
1	30000	31000
2	200	256
3		
⋮		

Every processor can have its own gdt.

Who creates GDT?

→ OS creates it

→ OS creates GDT in RAM and executes

Scanned by CamScanner

lgdt instruc" to load the address of GDT (in RAM) in GDTR register.

→ why can't user prog. do the same thing?
Not allowed by protec" ring

Protection rings.

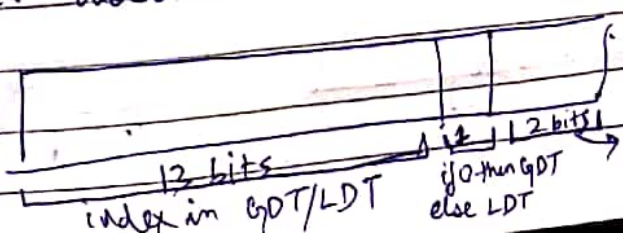
→ x86 has 4 protec" ring (0, 1, 2, 3)
→ 0 is most priv. And 3 is least priv.
→ OS executes in most priv. mode.
→ User prog. executes in least priv. mode.

High priv. means more instr" will be allowed to execute. So user prog. can't create gdt table as its not allowed to use lgdt instr".

Q How does hardware knows the priv. level?
→ The last 2 bits of CS segment selector contains CPL. (Current priv. level)

Q What prevents application from modifying the GDT entries?
→ GDT lives in OS memory. So user prog' can't access it.

Segment reg:



Q How does OS access entire memory?
→ OS segment selectors point to a GDT entry with base & limit set to 0 and 2^{32} resp.
0 to 0xffffffff.

Q Why user prog. can't set their segment reg. to OS GDT entries.
→ Supervisor (System) flag in the GDT entry.

base	limit	perm/Supervisor flag
1000	2000	0
0	0xffffffff	1
⋮	⋮	⋮

} Supervisor flag.

Context switch

(LDT) called Local Descr. Table

A → A dedicated GDT for every process.
→ Load the target process' GDT on every context switch.
→ execute lgdt.
→ Why this? Becaus if not, then one process can make its cs point to gdt index of some other process as supervisor flag will be zero so it'll be allowed.

P1

1000	2000	0
⋮		
0	0xffffffff	1
⋮		

P2

3000	4000	0
⋮		
0	0xffffffff	1
⋮		

Scanned by CamScanner

Page No.
Date

(B) Other option is to modify the GDT for each process so that ∃ only one GDT. Overwrite the GDT entries with target process' entries on every context switch. (Save GDT of prev. on stack)

Default Segment Registers

→ %cs is default for EIP
→ %ss is default for stack (push, pop)
→ %ds is default for most memory operands (eg movl $100, (%eax))
→ %es is default for string instruc^ns
→ %ds can be overriden by other segment reg: eg movl $100, %fs:(%eax)

Exceptions

→ Divide by zero
→ General protection fault (process tries to access a VA outside its limit)
→ 0-31 interrupt vec are reserved for exceptions. (look in intel manual)

Interrupt handler

→ IDT is ~~array~~ structure of 256 entries
→ IDT contains cs:eip pair of interrupt handlers. cs is cs of OS & eip is ip to the int. handler.

Scanned by CamScanner

→ OS creates IDT.
→ OS creates IDT in its own address space so that user prog. can't modify it and executes lidt instr^n to load the address of IDT in IDTR reg.
lidt is 0 level protec^n instr^n.

Task State segment (TSS)

Has state of kernel stack

→ on interrupt in user prog. the hardware automatically switches to kernel stack.
→ a special structure TSS contains the address of kernel stack & kernel stack segment.

Interrupt/exception in user mode

If prog. executing in user mode & interrupt comes

Switch to kernel stack (SS:esp)
push old-SS
push old-esp
push old-eflags
push old-CS
push old-eip

When interrupt comes in kernel mode, it will push values on kernel stack only & no new stack will be formed.

Scanned by CamScanner

Page No.
Date

---

push old-eflags
push old-cs
push old-eip

} for interrupt in kernel mode.

For every user thread, there is a kernel thread.

<u>Returning to user mode/kernel mode</u>

iret

---

pop old-eip
pop old-cs → CPL = 3
pop old-eflags
pop old-esp
pop old-ss

---

this tells whether it will return to user mode or kernel mode by CPL flag. So if CPL = 0 then it will pop 3 values else it will pop 5 values.

iret

---

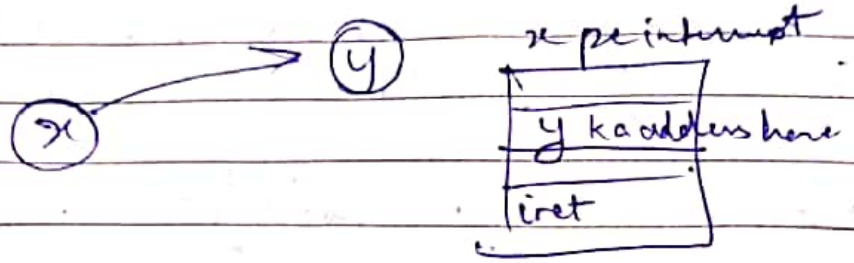
pop old-eip
pop old-cs → CPL = 0
pop old-eflags

CPL 1 & 2 are hardly used. mode for device drivers.

Scanned by CamScanner

fork & exec

When a new program/process starts, then it will put some random stack and change cip to the eip of the new process to start.



So upon executing iret, it returns to new prog.

Software interrupts

- int 3 → generates interrupt of vec no. 3
- into → generates int. of vec. no. 4, if the overflow condition is set.

This is used to set breakpoints. So at breakpoint, put this instruc" over there

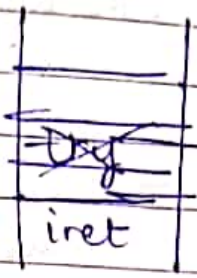
GDB

main()

[mov %eax, %ecx] → breakpoint here so replace it by int 3.

How to call user defined handler?

```
try {
    }
catch {
    }
```



→ Change this to catch before returning from handler.

This is better than calling a func.n as call itself may take 5 bytes & so we can't replace it on a 2 byte or 1 byte instr.n. O/W it may overwrite

28/1/19 Lec 8

Process Control block

```
struct process {
    int pid;
    unsigned base;
    unsigned limit;
    struct fd *fdtable;
    struct list threads;
};
```

} maintained by OS

Thread Control Block

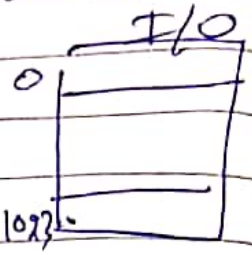
```
struct thread {
    int tid;
    struct process *pcb;
    void *esp;
    unsigned regs[8];
    int state;
}
```

Q Any reason that this instead of saved on stack?

Scanned by CamScanner

Thread state can be either:
- Running
- Ready
- Blocked → waiting for something like bus

I/O space → used for external devices like keyboards



I/O port
→ Use dedicated I/O space
→ Only 1024 I/O addresses
→ accessed using in/out instruc^n

Physical address space is 2^{32} (RAM)

→ Printer might be connected to one of these ports.
→ x86 in & out instr^n used:

port 100 ← in $100 → read in eax reg.
out $100 → writes value of eax reg in port 100

Line printer (Port I/O)

```
void lpt_putc(int c){
  while ((inb(STATUS_PORT) & BUSY)==0); /* wait for printer to consume prev. byte
  outb(DATA_PORT, 'c'); /* put the byte on parallel line */
  outb(CONTROL_PORT, STROBE); /* tell printer to look at it
  outb(CONTROL_PORT, 0);
}
```

Scanned by CamScanner

```
#define DATA-PORT 0x378
"       STATUS_PORT 0x379
"       BUSY 0x80
"       CONTROL-PORT 0x37A
"       STROBE 0x01
```

The above is device driver code in OS/kernel address space, User can't access these port but OS gives corr. System call to use I/O devices.

Memory Mapped I/O (as I/O space is very small = 1024)

→ Some address in the physical address space are reserved for devices.

→ Software talks to these devices using reading/writing to that address.

eg Line printer (MMIO)

```
volatile int * DATA_REG = 0xffff00;
             * STATUS-REG = 0xffff04;
             * CONTROL-REG = 0xffff0C;
#define BUSY 0x80
#define STROBE 0x01
void lpt-putc (int c) {
while((STATUS-REG[0] & BUSY) == 0);
DATA_REG[0] = 'c';
CONTROL_REG[0] = STROBE;
CONTROL-REG[0] = 0;
}
```

Scanned by CamScanner

→ Most new devices uses MMIO.

Volatile Vs Non-Volatile

```
while ((STATUS_REG[0] & BUSY)==0);
- - - - - -
int reg = STATUS_REG[0];
while ((reg & BUSY)==0);
```

volatile used to access data from memory & prevent compiler optimisation as compiler tries to optimise to reduce memory access time as it is costly.

Direct memory access

```
#define TX_STATUS_PORT  0x400
        TX_ADDR_PORT    0x401
        RX_STATUS_PORT  0x402
        RX_ADDR_PORT    0x403
tx_packet (char * buf) {    // transmit buf to network.
    outl (TX_STATUS_PORT, 0);
    outl (TX_ADDR_PORT, virt-to-phys(buf));
    while (inl (TX_STATUS_PORT)==0); // poll until buf is sent
}
```

We are doing TX addr PORT = actual physical address of buffer

Devices accesses the memory address in TX_ADDR_PORT.

→ This prevents CPU cycles.
→ Similar for recieving packets:

```
rx_packet (char *buf) {   //recieve network packet in buf.
    outl (RX_STATUS_PORT, 0)
    outl (RX_ADDR_PORT, virt_to_phys(buf));
    while(inl(RX_STATUS_PORT)==0); //wait until data is recieved in buf.
}
```

Polling → periodic check

→ If recieving rate is quite low or the event is very rare to happen so waiting in loop is not a good idea.
→ We periodically check for the occurence of an event. Eg after every 100ms network device checks for the arrival of a packet.
→ Polling is not useful in all cases:
especially, when the devicevents are not frequent eg.:
→ the keyboard input.
→ low throughput network channel.

Interrupts (instead of polling)

Interrupt vec. corr. to each device. So whenever device wants to send a value, it interrupts.

Scanned by CamScanner

instead of rx packet

```
/*called after packet is recieved */
rx_intr() {
    addr_t phys = inb(RX_ADDR_PORT);
    char *buf = phys_to_virt(phys);
    return buf;
}

/* called after every 100 ms */
rx_poll() {
    : same code
}
```

Polling is better when the throughput is very high i.e. the data rate is high as we may be in the interrupt of network & it interrupts again.

So it switches to polling when throughput becomes high & interrupt when throughput is less.

Disk driver

→ Chapter 3 of xv6 has real driver.

Shell

→ What if shell doesn't call wait sys. call is part. Then it can execute another command even if prev. command (child) hasn't finished

This is called a background process.

ls &

→ run ls in background
→ shell doesn't wait for ls to complete before reading another command.
→ We remove wait(0); line in parent to achieve this.

Problem with not calling wait?

→ The child process is not completely destroyed until the parent process calls the wait sys. call.
→ The wait sys. call returns the exit status of child process.
→ The child process becomes a zombie process after doing an exit sys. call until the parent process does the wait sys. call.

Parent can call wait any time after child is ended. So OS has the return value (exit status ~~error code~~) of child and parent can do wait & get the value.

→ So if parent dies, the child process becomes orphan process. ~~So they~~ hold some memory resources.

Scanned by CamScanner

→ init process adopts all the orphan processes.

→ Zombie process is a ~~p~~ resource leak
The shell somehow needs to call wait sys. call for this background process.

Signals

→ SIGINT (ctrl + c) → keyboard interrupt
→ SIG STOP (ctrl + z)
→ SIG CHILD (child process terminates)
→ SIG TRAP (process executes int 3)
→ SIG SEGV (memory access outside limit)
and many more.

There are some default behaviour of these signals like SIGCHILD will do wait().
But OS allows to change the handling of signals.

Signal system call

```
/*sighandler is called everytime ctrl+c is pressed*/
void sighandler(int signal) {
    printf("recieved %d signal\n", signal);
}
main() {
    signal(SIGINT, sighandler);
    ...
}
```

These system calls are called in user address space. Handlers are called. User can change these handlers too.

How OS does this?

OS does this signal sys. call using iret by making iret frame & making ret point to the ~~add~~ address of handler.

to move to user space from kernel space. as kernel recieves ctrl C using keyboard interrupt.

So user process was running & keyboard interrupt comes and we shift to kernel space. We know now that ctrl C was pressed so we have to call sighandler routine in user space so to go from ~~user s~~ kernel space to user space, we use iret by pushing frame for iret. It can also read user program state (as when keyboard interrupt came, it must have saved state of user process) from the stack.

Kill system call.

→ int kill (pid_t pid, int sig) ~~with pid = pid~~.

→ Send sig signal to another process

→ ~~eg~~ a process can send SIGKILL signal to kill target process.

→ It can only kill process of same user.

Scanned by CamScanner

30/1/19 Lec 9

Background processes in shell

```
childExit() {
    wait(0);
}
signal(SIGCHILD, childExit);
while(1) {
    write(1, "$", 2)
    readcommand(0, command, argv);
    if ((pid=fork())==0) {
        exec(command, argv, 0);
    } else if (pid>0) {
        //status = wait(0);
    } else
        printf("failed to fork");
}
```

childExit will be called whenever SIGCHILD signal comes (when child exits).

Fragmentation issue in segmentation

→ Suppose the RAM of size 10000 bytes is mapped to loc^n 0–10000.

→ And we've 10 processes (P1–P10) each of size ~~10000~~ 1000 bytes.

Scanned by CamScanner

Problem with ~~the approach~~ Segmentation?

→ Let's say P1 & P10 have finished exec^n.
→ At this point, the OS wants to load a new process P11 of size of 200 s.
→ The OS needs to realocate either P2 or P9 to accomodate the new process.
- need to copy entire processes, and change segment registers and base & limit.

Problem 2: Similarly if a process needs more RAM during its execution & consecutive addresses are not available, the OS may need to realocate the entire process (in case of malloc → dynamic alloc^n so not known before hand).
→ This can be done by inc. its limit & if cons. addresses are not available to shift & then extend limit.

Prob 3: Process address space is limited by the amount of RAM available on the system.

Sol^n → Paging

→ Due to limitations of segmentation the x86 hardware has an additional memory management hardware called paging hardware.
→ In paging scheme, the process virtual address space is $(0 - 2^{32} - 1)$ for every process

Scanned by CamScanner

In segmentation we can have a pointer to another segment so that same process could run there but there are only limited segment registers so this is issue.

→ In the paging scheme, the process address space is not limited by the address of RAM.
→ Each process virtual address is 0 - 0xffffffff
→ OS creates a lookup table that is walked by the paging hardware during runtime to convert the virtual addr. into the physical address.
→ Corr. to each virtual addr., there is a phys addr. in lookup table.

VA	Page Table	PA
0	5	0
1	3	1
2	4	2
3	1	3
4	2	4
5	6	5
6	7	6
7	0	7

↳ like hashtable as hashes are fast

Virtual addr. is used as index in the table. Corr. to each byte there is a mapping.

Scanned by CamScanner

Page No
Date

```
unsigned pagetable[8];
phys_addr = pagetable[0];
phys_addr = pagetable[4]
```
→ V.A.

→ Prob: If for every byte we use this hashtable then size of this table will be very large i.e. $2^{32}-1$ so its not feasible to create such a table for each process. (4GB for every process)

→ The virtual & phys. address spaces are divided into pages.

→ Pages are the continuous area of memory of size 2^k.

→ Page addresses are 2^k byte aligned (i.e. the page is always div. by 2^k).

So we are mapping from one page to another.

→ Total no. of virt. & phys. pages $= 2^{32}/2^k$.

→ The page table keeps a mapping from virtual page no. (VPN) to the phys. page no. (PPN)

$VPN = VA/2^k$: VA = virt. addr.

$PPN = PA/2^k$

So for 2^k bytes we have now one entry in page table instead of each 1 byte.

Scanned by CamScanner

→ Because the hardware walks the page tables at every memory access a single hash table is used to convert PPN to VPN.

→ VPN is key / index.

How large is page table?

- k == 0 (page size = 1 byte)
  - No. of entries in page table = 2^{32}
  - req. lot of space
- k == 26 (page size = 64 mb)
  - # of entries = 2^{26} = 64 bytes
  - fragmentation prob as this size is very large per process.
- Typical page size = 4096 = 4kB

Total # of virt. & phy. pages = $2^{32} / 2^{12} = 2^{20}$

VA to PA

VA = 0x10100

VPN = VA / 2^{12} = VA >> 12 = 0x10

offset = VA % 2^{12} = VA & $(2^{12}-1)$ = 0x100

PPN = hash (VPN) = 0x40 (say 0x40 = hash(10))

PA = (0x40 x 2^{12}) + offset.

Page table size = $2^{20} \times 2^{2}$ (4mb is size occupied by)

Thus 4mb is too much for a process of 64 kb.

size of address = 32 bits = 4 bytes.

64 Kb process

→ Only 16 out of 2^{20} entries are used → as 4MB page table

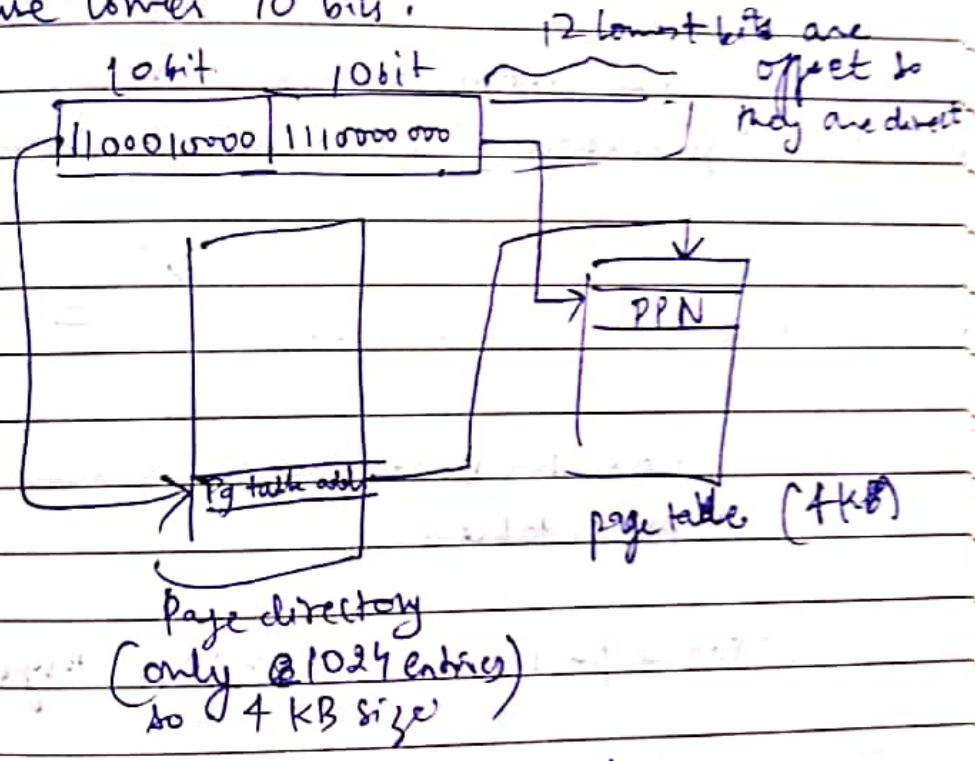
→ One dimensional page table causes fragmentation.

2-D page table (to reduce size of page table)

→ VPN is 20 bytes

→ lookup is done in 2 steps.

→ The higher 10 bits are used to index into a table & the value at the table points to other table. And in the other table we use lower 10 bits.



Page table are only created by page directory when they are to be used/ on demand.

So to use 4MB we need only 8 KB page table (page dir. + page table) memory.

→ So at first OS creates this page table in advanced as it knows how much will be req.
→ On demand page can also occur (malloc).

X86 page table

→ Paging can be disabled/enabled (using reg like CR0)
→ Segmentation can never be disabled.

Cons of page table: extra time to look up. (2 addl memory access)

Paging + segmentation how?
→ Segmentation will generate linear addresses

%ds : 0x100
base of ds say is 0x10000 } → final addr by segmentation = 0x101000 (linear address)

now this linear address is used to index page table.

Without paging, 0x101000 is physical address.

→ Compiler generates virtual address.

To eliminate effect of segment reg → set base to 0 and limit to $2^{32}-1$. This what most OS do. This disables effect of segment reg.

Scanned by CamScanner

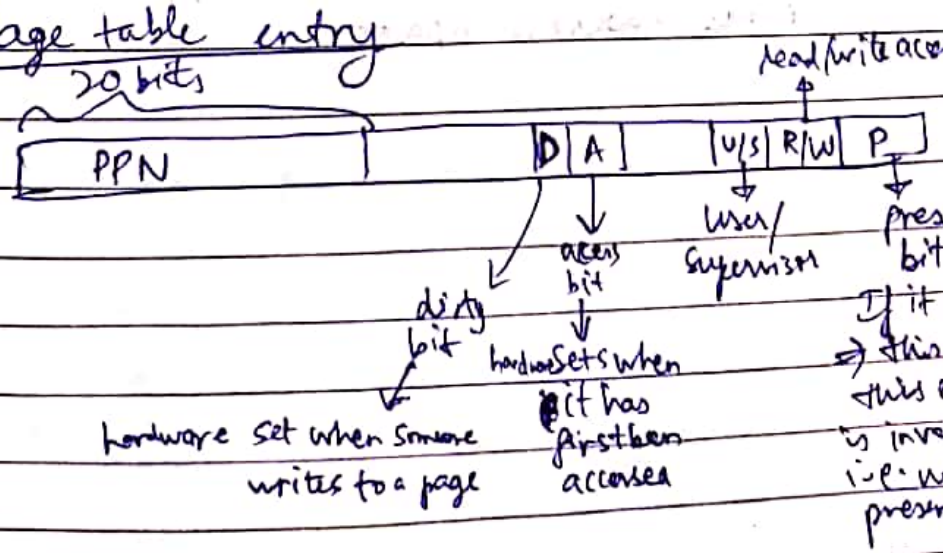
base of %ds = 0x10000
%ds : 0x100
linear addr = 0x101000
VPN = 0x10
offset = 0x100
PPN = pagetable(0x10) = 0x40 by 2D lookup
PA = (0x40 << 12) | 0x100 = 0x40100

→ Page tables are stored in RAM.

How does processor finds page directory?
→ %cr3 reg. contains the base addr. of page directory.
→ The OS creates the page table in memory & Sets the %cr3 reg. to the bases of page directory

Why can't user prog. load their own page tables?
→ ~~mov to~~ %cr3 is a privileged register.

Page table entry



Scanned by CamScanner

Page
Date

Memory Isolation

→ OS creates a page table for every process
→ OS uses diff VA to PA mapping for diff processes.

Page sharing

→ This can be simply done by using the same VA to PA mapping in diff page tables.

Where does OS live?

→ OS maps itself into each process page tables.
→ OS pages are shared among all the processes.
→ So same OS maps into all processes (like scheduler list is common for all).
→ This is done using page sharing.
→ To prevent user to access it, we won't set the user bit i.e. we set supervisor bit.

Why can't user applications not modify the page table entries?

→ Page table pages are only mapped in the kernel address space.

Scanned by CamScanner

Page No.
Date

1/8/19 Lec-10

On interrupt, control shifts to kernel. So we need to map entire kernel to process' address space as interrupt handler is in kernel space and if we don't do this, then we need to switch page tables.
OS in user space is good & user can't access it because of user/supervisor bit

Can OS be mapped at diff virtual addresses in diff processes?
→ NO. As pointer values will then be diff for diff processes say for ready list. Also, address of diff functions & IDT and handlers will be changed.

Other alternative?
→ Seperate page table for kernel.
→ Only map kernel stacks & interrupt handlers wrappers in user prog.
→ The interrupt handler switch to kernel page table before jumping to original handler & ~~return~~ after returning, switch back to user space page table
Overhead: Significant overhead of switching page tables.

Scanned by CamScanner

Adv. of this: 1. More address.
2. See below:

Pros & Cons of seperate page tables for kernel

Cons
→ Kernel can't directly access user pointers
→ The user buffers are always copied to the kernel address space during system calls.
read (int fd, char *buf, size_t buf);

Pros
If kernel share page table of user process, then it can directly access the buffer.
→ Adv. is while debugging as in other case, kernel can write to user data directly.

Overhead of page tables
→ Hardware walks the pagetable on every memory access. Two additional memory access on every mem. access
→ To reduce overhead of extra mem. access, on chip translation lookaside buffer (TLB) is used.
→ TLB caches VPN — PPN mappings.
→ Before walking the page table, hardware first looks into TLB.
→ TLB accesses are very fast as its in ~~the cat~~ cache (fully associated cache)
Cache of just 512 entries.

Scanned by CamScanner

→ It is useful as most of the applic. access the data in same VPN so not many pages accessed at same time so this reduces the overheads very much as access time becomes very less.

→ All this depends on TLB hit rate i.e if we have good TLB hit rate, we don't have to pay overhead much. e.g. TLB hitrate for most appl. is 99.9%.

What happens if OS modifies a page table entry which is cached in TLB?

→ TLB entries are not updated to the new value.

So the hardware uses the cached entries in TLB even though the actual entries are modified.

The software can use "invlpg" instruc" to invalidate a TLB entry. It takes VPN as arg.

The entire TLB is flushed when the OS reloads the %cr3 register.

The addresses of Page table & Page directory are physical addresses & not linear addresses o.w who will map them? (Recursion prob.)

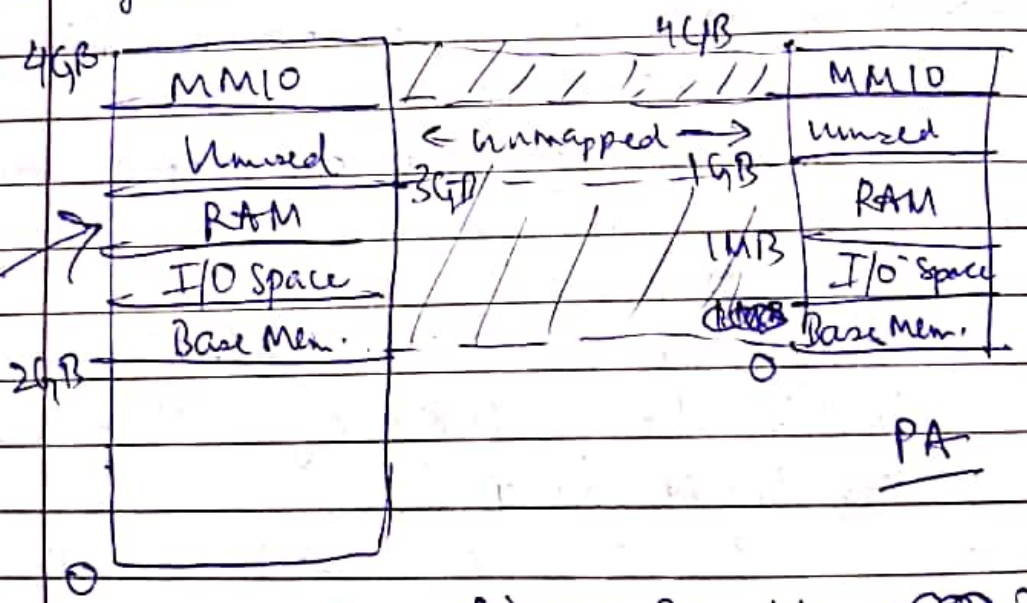
TLB flush mov %cr3 %eax
mov %eax %cr3

Large pages (4MB pages in x86)

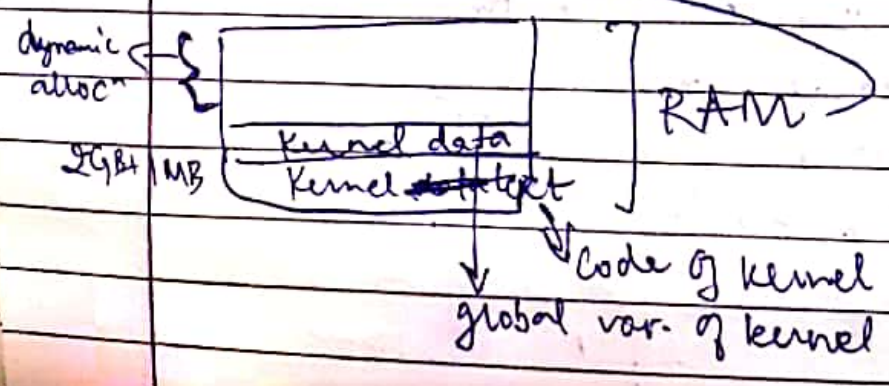
Large pages reduce the TLB pressure.
Hardware knows whether 4MB page or not.
So bits req. $= 2^2 \times 2^{20} = 2^{22} = 22$ bits.
So if we use them, then hardware only traverses page directory & not page table.

Paging in xV6 (Kernel mapping to VA)

xV6 works with 2GB. But let us generalize it to 4GB.



Since RAM b/w 2GB-4GB
so xV6 assumes ram < 2GB



Page No.
Date

x is something in kernel

P2V & V2P

```
# define KERNELBASE 0x80000000 //2GB
  P2V (x)      (KERNELBASE + x)
  V2P (x)      (x - KERNEL BASE)
```

kalloc() : allocates one virtual page (4KB) in kernel address space

kfree() : frees the virtual page allocated by kalloc.

kinit2(vstart, vend) : frees all the virtual pages b/w vstart to vend.

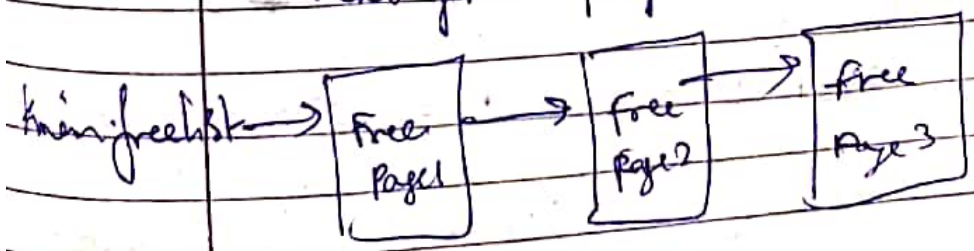
First thing that kernel does is to call kinit.

- kinit2 (P2V(4 * 1024 * 1024), P2V(PHYSTOP)) → after kernel data till top this all
- calls kfree for each virtual page in this range.

from 4MB to end of RAM free all pages. Till 4MB is kernel data so it can't be freed.

kfree

kmem.freelist → points to first free page. Every free page contains address of next free page.



Scanned by CamScanner

```
kfree (VA)
{       VA.next = kmem.freelist ;
        kmem.freelist = VA ;
}
```

kfree memsets the page with all bytes=1 so that by setting some garbage value, kernel can't use this page and gets some fault on access.

walkpgdir (unsigned *pgdir, unsigned va)

→ expects the valid page directory
→ Returns the address of the page table entry for a given VA
→ Creates page table if not already present.

```
walkpgdir (unsigned *pgdir, ...) {
  int pgd_index = (va >> 22);
  unsigned pgd_entry = pgd[pgd_index];
  unsigned *pgtable;
  if (!(pgd_entry & PTE_P)) {
        pgtable = kalloc();
        memset (pgtable, 0, PAGE_SIZE);
        pgd[pgd_index] = V2P(pgtable) | PTE_
                         PTE_W | PTE_U;
  } else {
        pgtable = P2V (PTE_ADDR(pgd_en
  }
  unsigned pgtable_idx = (va >> 12) & DX
  return &pgtable [pgtable_idx];
}
```

top 10 bits remain

Physical address of pgtable or addr of pgtable

/* create a page table entry */

returns VA of pg table

setting lowest 12 bits to zero as they're flags

top 20 bits given so AND

1024 means 1KB

But every entry is 4bytes (32 bit)

= 4KB

Page No.
Date

```
map~~user~~kernel page (unsigned *pgdir, unsigned va, unsigned pa)
{ unsigned* pte = walkpgdir(va);
  *pte = pa | PTE_P | PTE_W
}
```

PTE_U bit not set as.

Create a page table & map all phys pages to kernel pages in VA

```
#define KERNEL_BASE 0x80000000
#define PHYSEND RAM_SIZE (1GB)
create_page_table ( ){
    unsigned* pgdir = kalloc( );
    unsigned va;
    memset (pgdir, 0, PAGE_SIZE);
    for (va = KERNEL_BASE; va < PHYSEND;
        va += PAGE_SIZE) {
        map kernel page (pgdir, va, V2P(va));
    }
```

Loading new page table into memory.

```
load_page_table (unsigned *pgdir) {
    load_cr3 (V2P(pgdir));
}
```

CR3 also contains PA.

User appl. can't modify page table entries as page tables are in kernel space & we didn't set PTE_U bit.

Scanned by CamScanner

Fork sys call

→ It has to make new page for child & map every data from parent to child. But this is too much ~~wato~~ work & most of the time we don't even use it as we do exec & wipe it all to load new program.

→ COPY ON WRITE Optimisation is used

So on forking, it ~~use~~ doesn't copy the page table of parent to itself. But it uses the same page table till it does not write. Otherwise if it writes then it can corrupt values of parent & there won't be memory isolation. So to do this PTE-W flag is not set. So the process is only allowed to read.

So upon forking, it revokes the write access of both parent & child, And whenever one writes, then it copies to its own page table. This is useful if we do just exec in fork.

Scanned by CamScanner

Page No.
Date

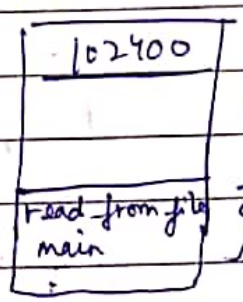
Lec 11 6/2/19

Executable & Linkable Format (ELF)

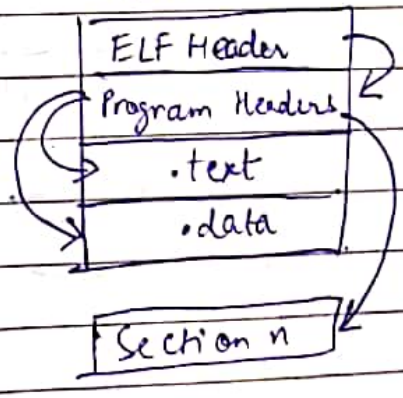
```
int num_entries = 102400;
int buf[102400];
int main() {
    read_from_file(buf, num_entries);
    sort(buf, num_entries);
    print_data(buf, num_entries);
    return 0;
}
```

data → variables (global)
text → code instructions

a.out



a.out



- .text
  cpu instr. of main, read_from_file, sort, print_data, etc.
- .data
  num_entries, buf
- stack

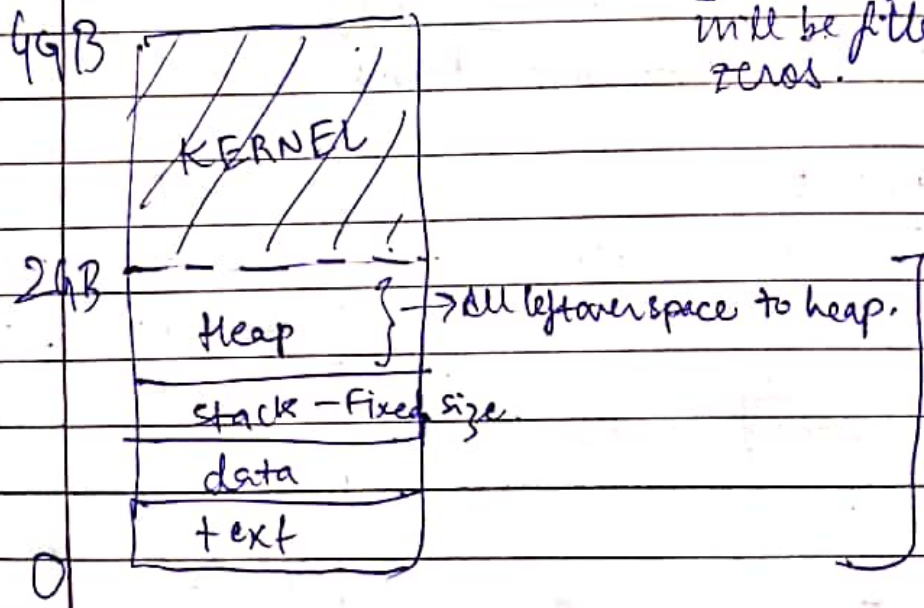
See elfhdr : 1005 in xV6 code listing

Filesize → how much bytes of data is initialised in file/need to be read from file

Memsize → Total memory needed for the file i.e. total space needed for variables.

If memsize > filesize, then our file has some uninitialised variables. So ~~was~~ they may have some garbage value. So we need not pre initialise everything as it will inc. file size.

So memsize - filesize will be filled with zeros.



Read Goto exec : 6310 of xV6 code listing

Setupkvm : 1837 → loading 2GB to 4GB with kernel for VA

Page No.
Date

Setup kvm()
→ creates page dir.
→ map I/O space
→ {0, 1MB} → {KERN-BASE, KERNBASE+1MB}
→ map kernel text + read only data
→ {1MB, textsize} → {KERN-BASE+1MB, KERNBASE+1MB+textsize}
→ map rest of RAM ie data+memory
→ {1MB+textsize, ramsize} → {Kernbase+1MB+textsize,
→ map more devices. (Devspace but size=0)

steps it does in that the permissions

no write perm here

exec (char *path, char **argv)
→ read program headers from file
→ for every prog. header
→ fetch the VA & mem. size of the section from prog header
→ call allocuvm to map phys. pages at the VA range {vaddr, vaddr+memsz}
→ call loaduvm to load sections from exe.
See allocuvm : 1953
See slides
See mappages : 1779 → Creates PTE for va for size=sz to map to phys addr
After loading text, data it has to load stack. So it allocates 2 pages for stack & revokes user access for 2nd page first
So that in case stack overflow occurs, we can get some error before actually corrupting text/data
This is done as this is faster than checking esp before every push. Obviously we can fail to detect stack overflow in case the data in stack directly jumps one page.

Scanned by CamScanner

After this we've to call main.
So we need to copy arguments of main stack

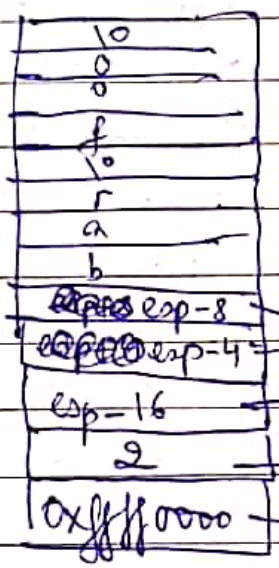
exec (char * path, char ** argv)

main(int argc, char * argv[]) argc → # of args.

char * argv[] = { "foo", "bar", NULL }

argc = 2.

exec takes name of executable & list of args



esp-8, esp-4 → each of 4 bytes

esp-16 → ref. to arguments i.e (esp-4) address of argv

2 → argc

0xffff0000 → fake return address as main will do explicit exit() instead of returning

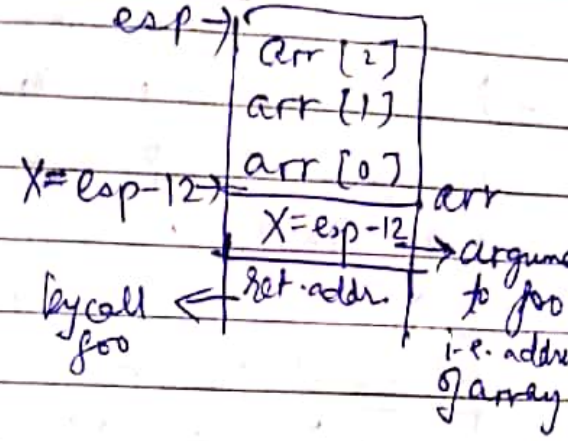
loaduvm can't be directly done by loading in VA as cr3 doesn't have the pgdir address i.e. page table is not active.
So we've to know phys page addr & then go to kernel address corr. to phys page address. (kernel addr = VA in kernel space)
uva2kva does this translation.

Page No.
Date

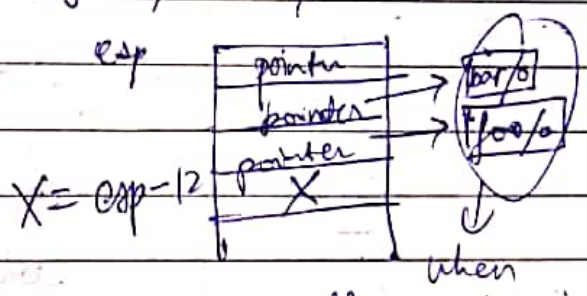
12/19 Lec 12

Local var. & args in gcc

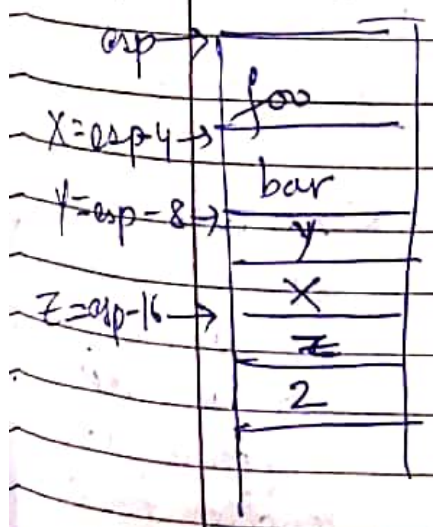
```
main()
{ int arr[3];
    foo(arr);
}
```




```
main( )
{
  char *arr[3] = {"foo", "bar", NULL}
  foo(arr);
}
```



So for main args:



exec sys call calls main by rewriting value on interrupt frame & do iret.
by doing cs, eip, ss, esp and random value to ret. address.

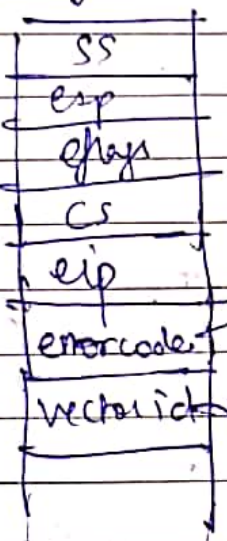
Scanned by CamScanner

Dynamic Memory Allocation

→ Pages above stack & below Kernel-base can be used for malloc.

→ So each process maintains process size which is equal to posn above stack (VA of top of stack).

On every interrupt/trap:



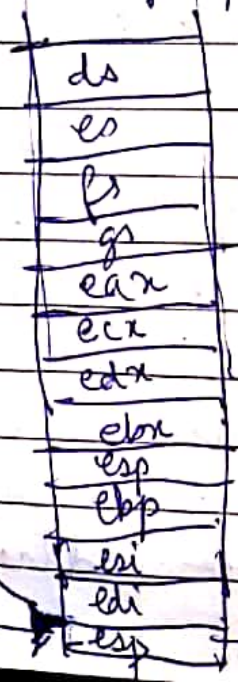
→ due to exceptions like divide by zero. if error code is not there, then software pushes fake error code on stack.

→ vector id of interrupt

trapframe

After pushing above values, it jumps to routine alltraps : 3254

So trap pushes on the same stack:



trap frame

→ Then we push esp at the end to pass pointer to this trap frame to another func. Then it'll call trap

Page No.
Date

trap : 3351

→ It stores proc→tf i.e. the current trap frame on sys call.

→ So exec sets proc→tf→eip to new executable main at 6396 line.

→ exec sets proc→tf→esp to new stack at 6397.

→ exec then calls switchuvm to load new page table in cr3 reg.

switchuvm : 1873

→ disable interrupts

→ setup TSS to point to the process kernel stack.

→ load the new page table.

→ the it calls freevm to free all the user pages in the old page directory, & user free pg table & pg dir.

Malloc

→ It uses sbrk sys. call to grow space

→ sys_sbrk (int n) : 3701

↳ +ve to malloc i.e. inc. size

↳ -ve to free i.e. dec. size

→ growproc (int n) : 2535

↳ adjust the process size by n bytes

↳ n > 0 : call allocuvm

↳ n < 0 : call deallocuvm

Scanned by CamScanner

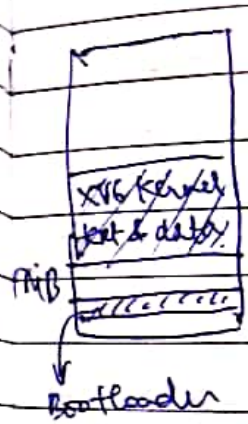
Page
Date

13/2/19

We have called switchuvm in growproc so that in case we do free, we need to flush TLB so that we may not access it still after freeing it.

A race condition in dealloc uvm:
Consider a multithreaded application, so when we are in deallocuvm, then the other thread (in other core) can still access TLB. So to prevent this we need to do TLB flush before kfree.

Does OS have to map the entire RAM in the kernel address space?
→ How to implement loaduvm: 1918 if the entire RAM is not mapped in kernel addr. space.
To do this, we can make a new api called kernelmap to do on demand mapping to kernel phys page & then unmap the page.



Page fault → page not present / page not mapped.

Scanned by CamScanner

4MB page = 2^{22} offset and 10 bit entry in pg dir = pg table

Page No.
Date

12/19 Lec-13

Bootloader is in ROM. ROM code loads it. ROM code can be directly executed.

Boot Loader

→ When PC starts, it loads the boot loader into a predefined physical memory in RAM.

→ The bootloader loads the KERNEL text & data at the physical address 0x100000 (1MB) (as 0-1 MB is bootloader & device).
We load at 1MB as we don't know how much big is our actual memory so can't load at 2GB.

→ The kernel code is compiled to be loaded at location 0x80100..0.

Page tables are not there at this point.

entry:1144 is the assembly func^n which is run after loading kernel by bootloader.
It calls entrypgdir:1411

So we map 0 to 4MB and 2GB to 2GB+4MB (both) to same physical page b/w 0 to 4MB.
If we don't do this, then we won't be able to execute instruc^n after eip as eip doesn't change, it is still b/w 0-4MB. So it must be mapped to same phys page. After this we jump to kernel VA.

So entry loads entrypgdir in cr3 with the above mapping. So at this point

paging hardware becomes active. But we're still in 0-4MB range & not above 2GB. So the virtual pages in range 0 : 0x400000 & 2GB ← (0x80000000 , 0x80400000) → 2GB+4MB refer to the same physical pages.

→ The entire XV6 kernel & data can fit into 4MB.

→ Let us call 0x400000 range as low range.

→ Let us call 0x80000000 : 0x80400000 range as high address.

So even after loading the page table the current EIP is still at low address.

→ entry uses global var. stack as kernel stack

→ entry loads the address (stack + KSTACKSIZE) which is a high address in register %esp

→ entry loads the address of main in %eax which is a high addr.

→ entry does an indirect jump through %eax reg to jump to main

→ After this step all addresses are high addresses.

Scanned by CamScanner

Page No.
Date

Creating the first process

→ userinit : 2502

→ allocproc : 2455

↳ allocate a process kernel block (PCB)

- size, kernel stack, pgdir, context, pid, trap frame, state, open files etc.
- allocate kernel stack
- allocate space on kernel stack for trap frame & process context.

to goto usermode by change set value of esp, eip.

has all callee saved reg & eip i.e. eip, ebp, ebx, esi, edi. It sets eip to forkret func^n.

trapret() routine is also stored on this stack.

After returning from allocproc, userinit does setupkvm(). At offset 0-4096 (VA) it loads initcode.S & this prog. is so small that it is stored in only 1 page.

So in trap frame, we need to do eip=0 & esp=4096 and when we do iret on trap frame, then it goes to initcode.S

It doesn't take any argument.

initcode.S → 8408. It only does exec system call to load an executable called init.

Scanned by CamScanner

Page No.
Date

Schedule

→ After userinit at some point main calls mpmain that eventually calls the scheduler:2708.

→ scheduler:2708 finds a runnable process, switch the page table & call switch:2758

using switchuvm()

(so it schedules init which was execed)

→ switch (scheduler, target process).
Switches the context of above two.
Context is like stack pointer.
So we go to forkret as eip=forkret[1].
Here we just release scheduler locks & return. So it will return to trapret routine as it pops the value ~ top of stack & jumps to it & above forkret we've trapret in stack.

Trapret: 3277. So it will go to eip=0 & execute initcode.S to execute exec("init")

Scanned by CamScanner

Page No.
Date

Locking (Chapter 5 - XV6)

→ Assume acq. & rel. for multiprocessor.

→ **Sleep**
- waiting for I/O
- waiting for an event
- sleep sys. call

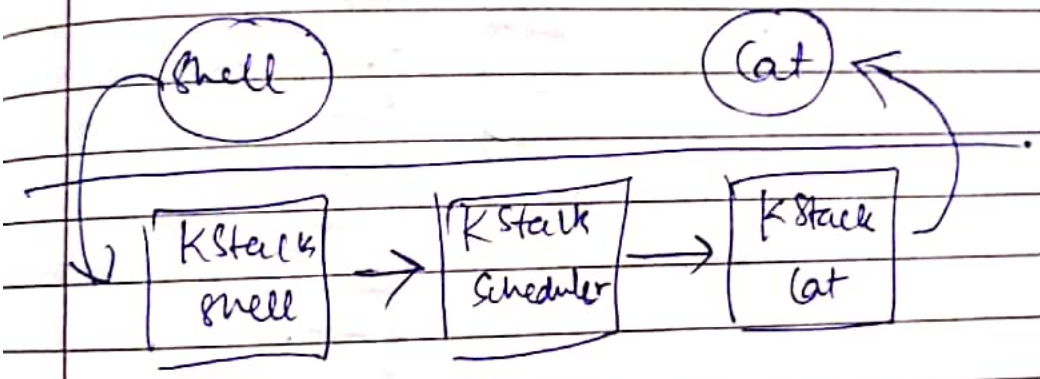
} → Scheduling happens

or a process has completed its time slice

In xV6 we've 1 thread per process.

Q So when to do context-switch?
- timer interrupt
- sleep.

Q How to support concurrent calls to scheduler?
- xV6 uses locks to prevent multiple processor's scheduling the same thread.



So 2 context switches happen. First it switches to kernel stack of scheduler & then to kernel stack of scheduled thread.

Scheduler → maintains a list of all processes

→ In xV6, all processes are single threaded.

→ Scheduler maintains a list of processes control block (xV6)

→ PCB contains a pointer to the process context.

Scanned by CamScanner

→ Process context is the info that we need to save when we want to do context switches.

→ Each thread has its own set of CPU reg, stack & instruc^n pointer.

→ But processor has only one set of CPU reg.

→ So processor needs to multiplex these among diff threads.

→ The reg. are multiplexed by saving them on the process context.

→ The EIP is also saved in process context.

→ Space is allocated for this on PCB.

values saved : ~~eip~~ → call inst^n.

ebp
ebx
esi
edi

only these saved by switch as others already on stack.

all are on Kstack

~~ecx~~
~~edx~~
~~eax~~

→ caller saved

~~esp~~ → = process context

Switch : 2958

Scheduler : 2708

↳ called on scheduler stack

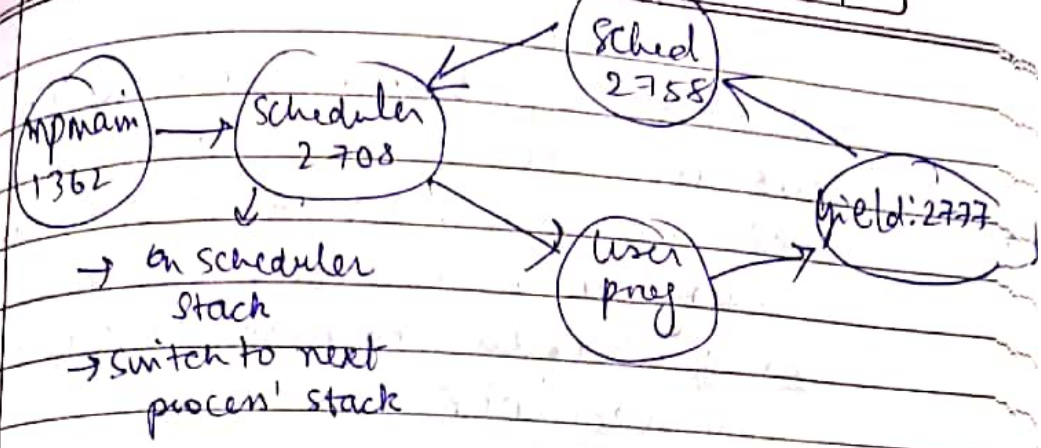
↳ scheduler is called when the first process is created. mpmain calls scheduler.

Yield : 2777

→ called after timeslice expired to save context of current process & load scheduler's context.

Scanned by CamScanner

current process' stack to scheduler stack



→ on scheduler stack
→ switch to next process' stack

PCB list

→ scheduler maintains a list of processes
→ scheduler walks to PCB list to find a runnable process for scheduling.
→ fork : 2556 creates a new process & adds it to PCB list.
↳ It is the only way to create new process.

ptable.lock → shared among all cores,
↳ used to protect the pcb,
↳ concurrent calls to yield & scheduler may try to schedule & yield the same process concurrently.
lock released in yield.

cpu → scheduler → It is a per cpu variable & not shared among processors.
We can't share this b/c in one processor, scheduler might be in a diff context (eg at index i) & other scheduler might be at diff index (eg index j) so we need to have diff states & memory.

Scanned by CamScanner

→ acquire (&ptable.lock) internally disables interrupts because if timer interrupt again comes, it'll again trying to acquire lock but since it is already acquired by scheduler so deadlock.
Interrupts disabled only for that cpu.

→ for loop in scheduler is infinite as we need to schedule always. But we have release outside inside for loop to prevent the scenario:
- no process is runnable
- we have a process waiting for I/O
- I/O comes & wants to change its state to runnable but for that it needs ptable.lock. But since scheduler won't release, so it won't be able to schedule the thread. So kind of deadlock.

I/O interrupt

Sleep & Wakeup

1. Busy waiting → inefficient if sender is rare
2. Lost wakeup prob

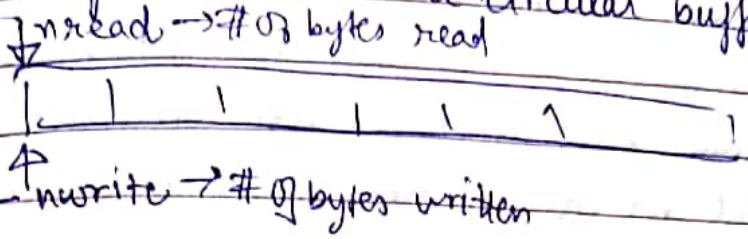
Sleep: 2809
Wakeup: 2864

Scanned by CamScanner

21/2/19

Pipe

→ pipe is a circular buffer of size PIPESIZE
→ implemented as a circular buffer



- pipe_is_empty : nread == nwrite
- pipe_is_full : nwrite − nread == PIPESIZE

(5530) → pipe read (struct pipe *p, char *addr, int n);
- tries to read n bytes from pipe to buffer (addr)
- returns the no. of byte read
-

→ pipewrite (struct pipe *p, char *addr, int n);
- write n bytes from the buffer (addr) to pipe (p)
- if the pipe is full, waits until a reader consumes some bytes

Exit & Wait

(5504) Exit : → marks the status of current process as ZOMBIE
→ If the parent has already exited, change the state to ZOMBIE

Scanned by CamScanner

Q Why exit don't free its pgdir & kstack?
Because it is executing in that pgdir & that kstack, so it can't free those o.w it'll crash. So parent frees it when this proc is not running.
But we can free user pages though as this process now won't go to its user pages to execute user code.
All fd are also closed in exit syscall.

It acquires ptable lock to remove concurrency prob at line 2640 & at same time parent frees.

Wait

Q.

Scanned by CamScanner

Page No.
Date

kill : 2875

→ kill sys call allows a process to kill another proc.
→ Killing a proc at arbitary pts. is tricky because the target process might be holding in the middle of something
a holding kernel lock.
→ kill : 2875 simply sets a field "killed" in the target process at PCB
→ trap : 2251 checks the killed flag before returning to user mode
• if the current process was killed, trap calls exit.
(before returning to user mode we are sure that it won't hold kernel lock so at that point we call exit).

Files

→ Files are stored on disk.
→ OS provides read/write interfaces for reading/writing as paging hardware doesn't map IO hardware to VA i.e. no isolation.

Memory Mapped files

→ On open system call, the OS checks the permissions & map the entire file in the VA space of the appl.
→ The application can then directly read/write to file through reading/writing to memory
• no system call.

Scanned by CamScanner

→ When ~~any~~ appl. does a close sys call to OS, then OS can write the pages back to disk.

Page fault

→ exception raised when we try to access a page which is not ~~loaded~~ mapped/present in a page table. It could also come if we try to write to a read only permissions page.

Precise exceptions

→ Hardware rollbacks all the changes made by a partially executed instruct^n which causes an exception.

eg xadd %eax, (%ecx)

→ Exchange the values of src & dest op. & load the sum of these 2 values in the dest. operand.

Implemented using micro opcodes.

C code: t1 = %eax

%eax += *%ecx

*%ecx = t1

*%ecx += %eax

Exception can occur if we don't have ~~read~~ write perm. but have read perm. So to prevent half changes, x86 rollbacks.

Page Fault

→ Page fault is used for copy on write optimisation to whenever parent or child try to write to

CR2 → VA of page which causes page fault

Page No.
Date

shared page after doing a fork sys call, then this will generate page fault, as they don't have write access. So when OS recieves this page fault i.e is in its page fault handler, then it know that this page (in CR2) is mapped as read only in both parent & child, so it can map this page with write perm. in the VA of anyone who causes page fault. Then it can do iret and iret will execute the same instruc" again (due to rollback) & won't get page fault.

%cr2 = VA
kalloc()
copy
map page with write perm. in child
enable write access in parent
iret.

This won't work without precise exceptions.

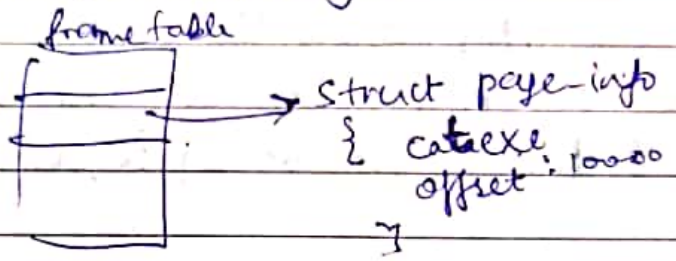
Demand Paging

→ Lazily loading program text & data in memory.

→ Memory acts as a cache between the prog. on disk & processor.

For this, we can have another new software frame table where we can keep addition inf. for each page so that whenever page fault comes, we goto ~~from~~ frame table

& load the memory from which file from which to which VA.
So initially we can just ~~perform~~ make frame table & don't add any page to page table. So as soon as process will start to execute, it'll give a page fault & then we'll load corr. data using frame table entry.



→ Corresponding to every page table, maintain another data structure called frame table.
→ Corr. to every virtual page, store the file descriptor & offset in the frame table.
→ On a page ~~table~~ fault, allocate a physical page, copy data from disk to memory, & map the page into page table.

What if no physical page is available to map in the page table?
→ We can have swap area on hard disk to load & store temp. state of RAM.
Eg VA : 0 – 409600 bytes : 100 virtual pgs (VP)
PA : 0 – 4096 bytes : 1 physical pages (PP)
User program accesses : VP1
Page fault
OS maps VP1 – PP1.

ecx bytes

Page No.
Date

Normally, mov & normal instructions have only one memory access.
So we need at max 3 pages → 2 for mem. access as it maybe on boundary of a page, and 1 for EIP.

instruc^n:
97 puts ecx bytes of value in esi to loc^n in edi.
We assume only 1 access to page for one instruc^n:

Q. 100 VP, 1PP
User prog. accesses VP1
page fault.
OS maps VP1 - PP1
User prog access VP2
pg fault
save PP1 to swap space
OS maps VP2 - PP1.

5/19

Swapping

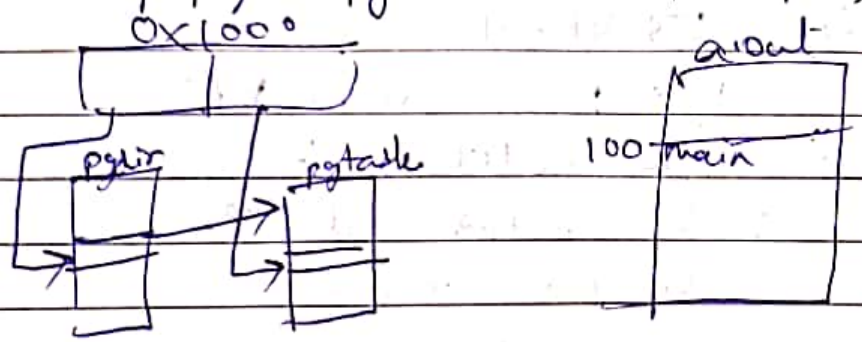
→ Some area of the disk is reserved for saving contents of phys. memory temporarily.

Scanned by CamScanner

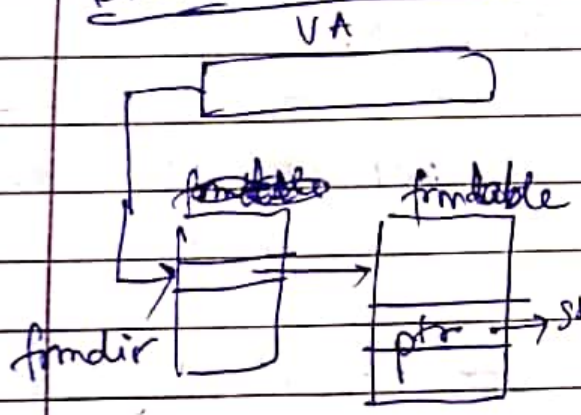
Exec - without DP (demand paging)

ELF → load from a.out at offset 100
at VA 0x1000, size = 4096.

```
char *ptr = kalloc();
fd = open("a.out");
seek(fd, 100, 0);
read(fd, ptr, 4096);
map_pages(pgdir, 0x1000, V2P(ptr));
```



Exec — with DP



exec: load from offset 100 from a.out at VA 0x1000, size = 4096.

```
fd = open("a.out");
page_info.fd = fd;
page_info.offset = 100;
```

On page fault: walk frame dir:

```
ptr = kalloc()
seek(page_info.fd, page_info.offset);
read(page_info.fd, ptr, 4096);
map_page(pgdir, 0x1000, V2P(ptr));
```

Scanned by CamScanner

memory mapped file is just in RAM.

Page No.
Date

also.

→ Same thing for memory mapped file: Keep this info that at which offset the file is at.

Frame dir. is also in kernel space.

→ In the struct, to swap from swap space, we might also store some flags whether it is swapped or not & the location where it is in swap memory.

Disk: Seq. of sectors/blocks

512 bytes	512	. . .		

So for swap, we can use disk interface i.e. store the sector no. to swap. OS can directly write to disk sector but not user process as it is malicious. So user uses file system.

Code to swap:

```
if (flag & SWAPPED)
{ ptr = kalloc();
  read_swap_page(ptr, sector_id)
  map_page(pgdir, 0x1000, V2P(ptr))
  free_swap_space(sector_id);
}
```

Check ptr != null.
If yes, then swap some page to swap memory & do kalloc again.

Swap(VP1);

Fork bump: doing fork in an infinite loop.

See Slide: On a page fault:

(malloc → kernel doesn't map any page)

sbrk syscall ?

Unused stack & heap pgs filled by zero on access (specified by ELF)

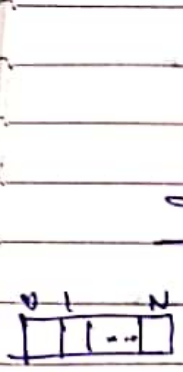
Scanned by CamScanner

memory mapped files

MMF → file in VA space of user so no read/write, simply write to memory.

Disk latency

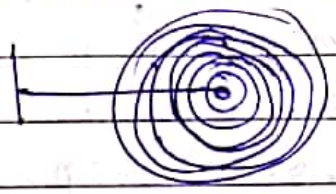
→ Disk interface allows programmers to read/write a block of data (512 bytes)



We can either read an entire sector or write an entire sector.

→ Latency depends on rotate time and ~~access time~~ move arm time

time to issue & serve real req.



→ move arm
→ rotate disc

Latency b/w 1ms – 100ms

But memory latency – 1 ns – 100ns

Sequential access is faster as arm & disk has already been moved & rotated. Random access is slower.

Q Why not page table for disk?

Disk latency is already too high so no point in it. The sys. call is anyway very less as compared to disk latency, so we have read/write sys calls instead of page tables. So even if we allowed user prog. to directly write to that sector using disk pg table, it won't much as time nearly same.

Page No.
Date

Q. How does page size affect swapping?
More time to swap large pages.
So need to copy atleast page size of data.
But benefit is these sectors are contiguous so disk read/write time is faster as compared to writing many small pages in disk as they mayn't be contiguous.

Temporal locality
→ It states that the data which is currently getting accessed has higher chance of getting accessed again.

Spatial locality
→ If we are accessing a memory loc^n, it is very possible that soon we may access nearby address.

If pgsize is 4MB → Spatial locality is not preserved as very far loc^n also brought into memory i.e. stale & not used.

4096 bytes
Disk blocks

So 4KB pages preserved spatial locality as it brings only hot data.
But 4MB pgs access time is less as compared to 1024 4KB pages (as disk sequential I/O is less).

What if spatial locality not preserved?
→ A lot of cold data lives in memory which may be used for hot data.

Scanned by CamScanner

Page
Date

→

Average latency.

→ Say the disk is 10ms & memory latency is 10μs.
→ 90% of pnc.

Throughput → Determines good page size

~~Page~~

Page replacement

→ A good ~~a~~ DP algorithm should minimize the no. of page faults.
→ A page replacement algo decides which page is going to be swapped i.e. hot pgs should remain & cold should not.

Scanned by CamScanner

Page No.
Date

→ Random : not good as may swap hot page but easy to implement.

→ FIFO : replace the oldest page (in terms of age)
Add pgs to FIFO queue when they are brought in memory.

Consider 3 PP & 7 VP :

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

7*	7*	7*	7*	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
	0	0	0	0*	0*	0*	0*	0*	0*	0*	0*	0*	0*	0*	0*	0*	0*	0*	0*
			1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1*

no page fault

2*	4																		
3	3*																		
0	0																		

→ Prob in FIFO : if we have 3 PP, then we may get 5 pg faults, but if we have >3PP, then we may have >5 pg fault that ideally shouldn't happen.

eg → 1, 2, 3, 4, 1, 2, 5, 1, 1, 2, 3, 4, 5

1*	1*	1*	4	4	4*							
	2	2	2*	1	1							
		3	3	3*	2							

9 pg fault.

Belady's Anomaly

1*	1*	1*	1*	5	5	5	5*	4	4			
	2	2	2	2*	1	1	1	1*	5			
		3	3	3	3*	2	2	2	2*			
			4	4	4	4*	3	3	3			

10 pg faults

Scanned by CamScanner

Optimal: Know future asked pages.

Optimal page replacement

→ Replace page i.e will be "not" accessed for longest ~~time~~ period of time.

eg 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

7 → 7 → 7 → 2 → 2 → 2 → 2 → 2 → 7

0 0 0 0 4 0 0 0

1 1 3 3 3 1 1

Optimal but not practical

9

LRU replacement

Replace pg ~~[illegible]~~ that is least recently used.

eg 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

7 → 7 → 7* → 2 → 2 → 2* → 4 → 4 → 4* → 0 ...

0 0 0* 0 0 0 0* 3 3

1 1 1* 3 3* 2 2 2

no pg fault

12 pg faults

LRU can be implemented using timestamp. We update timestamp of page to current timestamp on every reference.

It can be implemented using stack. Put the newly referenced page to the top of stack. If the page already ~~exists~~ resides on stack, remove from stack & put on top.

→ It doesn't have belady anomaly as if we've ~~to~~ more no. of ~~to~~ page frames (PF) then stack will be larger than the one with less PF so stack will be superset of stack with less pf.

Scanned by CamScanner

Page No.
Date

Global vs local pg replacement → wrt processes.

LRU

→ We need some hardware support to update the timestamp on every access.
→ x86 maintains an access (reference) bit set by hardware
→ So whenever we read/write to memory, hardware automatically sets the access bit to 1. If mov $100, 0x10101 will set access bit of 0x10 page table entry.
In this case we're writing so dirty bit will also be set.
So we can clear all access bit after every 100 ms say to get pages accessed in last 100 ms.

Additional n-bit algo.

→ keeps n bit timestamp for each page in memory.
→ So at every 100ms, we right shift the n-bit timestamp & put value of access bit to MSB of this & then clear access bit.
If initially → [1|0|1|0|1] → 5 bit
after 100ms if access bit = 0 ⇒ [0|1|0|1|0]
= 1 ⇒ [1|0|0|1|0] } Right shifted

If multiple pages have same ⇒ so many
But this is very expensive so OS don't use it.

Scanned by CamScanner

Second chance algo.

→ A simplification of the prev. algo is this algo in which we combine fifo and access bit. we maintain fifo to remove page but if a page comes with access bit = 1 then it is given a second chance.

eg 7, 0, 1, 2, 3, 4, 5, 9, 10, 11, 12

Clock algo.

① | ~~7~~ | 0 | 1 | 2 | 3 |

② 4

→ Clock ptr.

③ 4 5

④ 4 5 9

⑤ 4 5 9 10

⑥ 4 5 9 10 11

⑦ 12 5 9 10 11

Second chance eg:- 1 2 3 4 5 6 7 8 9

| 1 | 2 | 3 | 4 | 5 |

① We've to replace pg 1 but its access bit is set. So we reset its access bit & give it second chance i.e don't remove it & move ptr.

② Same for 2, 3, 4, 5 will be done as all of them have access bit is set. So now all page's access bit is reset. So now when we come back to 1, then we'll remove 1 as its access bit is zero.

Scanned by CamScanner

Page No.
Date

~~Pls to prove page~~

```
select_victim(struct queue_elem *clock) {
  while (1) {
      page = clock -> page;
      if (!was_accessed(page))
          return page;
      clear_access_bit(page);
      clock = clock -> next;
  }
}
```

Replacement only happen when we don't have any physical pages left.
Page replacement is only done for user pgs.

<u>Dirty bit</u> → Set by hardware when we write to a page

mov $100, 0x10101

⇒ pte = pte(0x10) } by hardware.
pte → dirty bit = 1

<u>Enhanced 2nd chance algo</u>

→ In case of memory mapped file, a page need not be saved to be written to disk if it is not modified. So if we have dirty bit not set, then we don't have to write to disk. So we can skip it and directly remove from VA.

→ The enhanced algo gives preference to dirty page over clean page.

Scanned by CamScanner

Page No.
Date

VA

```
char *addr = mmap(ls.txt);
```

munmap(ls.txt); → write changes in the file back to disk,

→ It avoids swapping pages whose dirty bit is set.

```
select_victim (struct queue_elem *clock){
  while(1) {
      page = clock->page;
      if (!was_accessed(page)) {
          if (!was_written(page)||page->given_
              return page
          page->given_adv = true;
      }
      else {
          clear_access_bit(page);
          page->given_adv = false;
      }
      clock = clock->next;
  }
}
```

There can be cases when dirty bit is set but access bit is not set as due to algo above we're clearing access bit when we're giving it 2nd chance.
We can't clear dirty bit unless we sync it with disk.

Scanned by CamScanner

Page No.
Date

Allocation of frames

→ An OS must decide how page frames are allocated for a process.

→ Two allocation policies:

- Local
- Global.

fixed quota for each process.

Local prob → under utilisation of memory.
Benefit is that we get same performance each time we run the same process. Its performance doesn't depend on work load.
Depending on binary (exe) size OS can decide each process' local quota. But this is not very good/accurate. Also we can't estimate heap.

Global processes given PP from set of all PP. But here performance depends also on other processes which are running i.e. workload, so no deterministic performance of process.
Benefit → max throughput/utilisation of memory.

Working Set → working set of a process is the set of pages accessed in last Δ page references
for eg if Δ = 10 :

eg 2, 6, 1, 5, 7, 7, 7, 7, 5, 1, 6, 2, 3, 4, 1, 2, 4, 4, 4, 3, 4, 4, 1, 3, 2

← Δ → t1 ← Δ → t2

Working set at t1: {1, 6, 5, 7} t2: {1, 2, 3, 4}

Scanned by CamScanner

Page No.
Date

Thrashing: If the sum of working sets of active applications is larger than the total no. of physical frames, then the appl. will spend most of the time swapping.
This results in very low CPU utilisation because most of the time will be spent in disk I/O. This is called thrashing.
Eg if working set of P1 = 2 & P2 = 8.
And if PP = 8, then 8+2 = 10 > 8
So we have thrashing prob.

To overcome this, we mayn't schedule all processes immediately. So we let one process run for some more time.
But appl. won't be very interactive as we'll schedule less often (and thus less disk I/O).
If a single process' working set > total P.P., then best way is to never schedule this process.

How to avoid thrashing prob?

To avoid thrashing, the scheduler maintains 2 lists: active list & inactive list.
Appl. may move b/w active & inactive list.
The scheduler picks process from active list.
Process are added to the active list in such a way that the sum of working sets of active process is less than total no. of physical frames. This is the invariant. Pages of process in inactive list are in swap memory.

Scanned by CamScanner

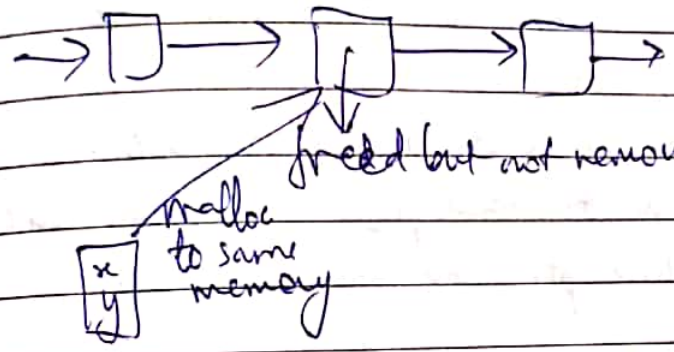
Page No.
Date

Page Protection.

5 Temporal safety. (use after free bug)

Dangling ptr → a ptr that has already been freed

eg in delete-node, we just free but don't remove from ~~memory~~ linked list. So the same memory can be malloced again and so we'll have 2 alias to same memory (as we didn't update memory).



We won't even get segmentation fault even if no one malloced after free (consider free as nop). If free actually frees, we might get segfault.

So to ensure such safety:

→ never ~~use~~ reuse a VA. malloc always returns a new VA. (free is nop i.e. does nothing).

But we can run ~~of~~ out of VA.

So for every allocated thing, we align it to page size. So in free, we can remove PP corr. to VP. So we get a page fault on accessing dangling ptr.

Scanned by CamScanner

11/3/19

Q. On free take back physical page?

A. Won't work in case the malloc done is not page aligned and equal to page size as some other object be there on the same VP.

Soln → ~~Always~~ On malloc → allocate virtual page for every allocation irrespective of size malloced. Like malloc(20) & malloc(30) will both give new VP. But in this, there is memory wastage. This scheme is good in case live objects are not very large.

New VA each time we'll get.

So now on freed page, we can catch dangling ptr by page fault.

Problem with this scheme → can't support many live objects due to large mem. wastage.

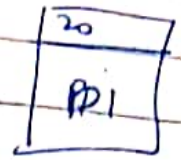
We might run out of VA in 32 bit sys but not in 64 bit system as it has very large ~~memory~~ address range.

Other way is to keep extra data structures & instrumentation to catch use after free bugs but this makes the prog. 5-10X slow.

Adv. of using hardware is that hardware is very fast as compared to writing software instr. for same task in case we don't have hardware support.

Scanned by CamScanner

Share phys. page ?



VP1 = malloc(20)

VP2 = VP2+20 = malloc(30)

This works but →

So here we're freeing only a portion of PP corr. to that VP ① But this can cause fragmentation problem.

② Consider that we're using too many VP as compared to PP. So here we might have prob. of TLB pressure.

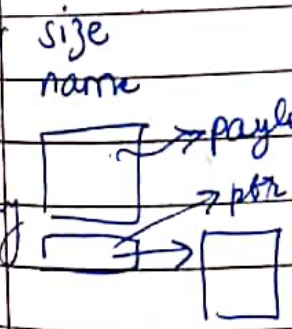
Each disc sector of 512 bytes

File System

→ read, write sys call.

→ Directories are stored on disk sector.

How to lookup "/usr/rtm/xv6/fs.c" ?



In file, the payload could just have content of file.

Drawback → to write at specif loc^n in file, need to traverse all the links & thus more time.

For every file/dir there is first block called inode which has address of all sectors of the file. If file size = 512 bytes then it need 1 sector for its content + 1 block for inode. So if file has 10 sectors, then we will have 10 addresses in the inode table with addr. of all sectors. So inode has metadata.
Directory is also similiar to a file, it is like a structured file.

/ a.c
b.c
c.c

a.c
100
b.c
101
c.c
102

} Dir. also like a file.

To look up /a.c : we need to look up in the root directory's inode.

```
struct inode {
  uint num_sectors;
  uint sectors[16];
}
```

```
lookup(filename)
{ for(i=0; i<num_sect
    read(sector[i], filenam
}
```

Buffer Cache

→ Reading/writing to disk is very slow.
→ Cache popular disk blocks so that they don't need to be re-read from the disk.
→ Need to synchronize access to disk blocks.

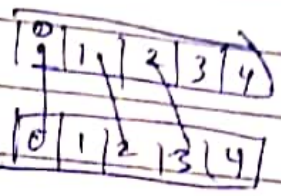
Scanned by CamScanner

Sector → PA

Crash Recovery

→ What happens if a partial disk update leaves the filesystem in inconsistent state.

9/12 **Block to sector :**



Block-no = offset / block-size

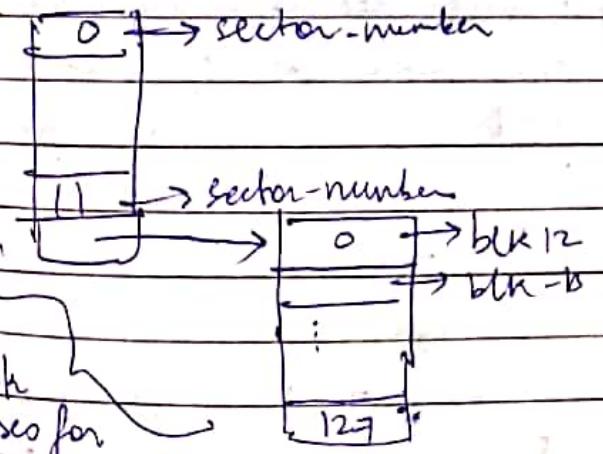
inode contains this mapping.

```
struct dinode {
    short type;
    short major;
    short minor;

    int addr[12+1];
```

→ 12 direct addresses & 1 indirect address

→ Indirect address is address to sector containing these mappings.



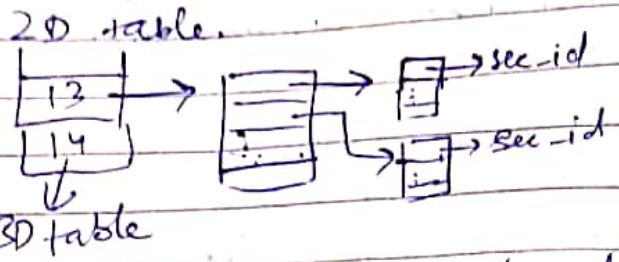
Max file size : 12 × 512 = 6 KB

128 × 512 = 64 KB

Scanned by CamScanner

To support more size, do something like paging with 2D table.

So



So this way we can support large files (512 GB in 3D Table).

To reduce this metadata (of mapping), we may increase the block size, But this may lead to internal fragmentation.

Another advantage: Disk access time decrease as now we're reading a continuous data so we only rotate & not move arm.

As compared to smaller blocks, we need more random access than sequential access, which takes more time.

A file sys. must decide a block size that preserves the spatial locality of common workloads.

NTFS : 4 KB (windows)

FAT32 : 512 bytes - 4 KB (Old windows)

EXT.3 : 1 KB - 4 KB

XV6 : 512 bytes

→ content of directory i.e. each file.

```
struct dirent {
    ushort inum
    char name[14];   //max. length of filename
                     ↳ 14 chars
};
```

16 bytes long.

So max entries = max-file-sz / 16

Page No.
Date

Path to inode

/x/y/z

How to reach inode corr. to z?

root-inode → given.

inode = find (root_inode, x);
inode = find (inode, y);
inode = find (inode, z);

struct dinode {
⋮
short nlink; → no. of links to this file. A file can be refered from diff directories. Like same file in diff directory.
⋮
}

link sys call

fd = open ("x/y", O_CREATE);
link ("x/y", "x/z"); // y & z point to same file
unlink ("x/y")

We can delete after this till we unlink from z also. But we also need to close the file descriptor before deleting. It has the place in struct ~~work name~~ like fd references. So a file can be deleted only if there is no open fd. and no nlink. (i.e nlink = 0)
So the no. of references can be stored on memory not disk as all files will be closed when system shuts down.

struct dinode {
⋮
int ref; → shared among all open files to increment open fds.
}

Scanned by CamScanner

XV6 File sys :

for every sector bitmap contains a bit which tells it is already allocated or not

boot	super	log	inodes	bitmap	data... data

not used be file system

```
struct superblock {
    uint size;
    uint nblocks;      // no. of blocks
    uint ninodes;
    uint log;
    uint logstart;
    uint inodestart;
    uint bmapstart;
}
```

→ Limit on max. no. of inodes as they are assigned mem. sequentially. It's good design in terms of when we want to sequentially access all inodes. But this imposes restric^n on no. of inodes.

→ bitmap contain info whether a block is present or not. So for each block we're 1 bit as 0 or 1. So to check if sector 128 is mapped or not, we need to check value at bmapstart + 128 & if it is one then that mean, the sector is mapped

So overhead of bitmap for 512MB is 1MB.

Scanned by CamScanner

Page No.
Date

balloc: allocates a new disk block
→ Iterates through the free bitmap to obtain a free block
→ mark the block as allocated in the bitmap.

bfree: free a block
→ find the right bitmap bit corr. to the sector & clears it.

Creating a file

echo > a (creates file named a)
→ allocate an inode (update type i.e. its file)
→ update nlink, size, direct, indirect block fields in the inode
→ add a new entry to the parent directory
→ update the size of the parent inode (dir.)

Race condition: Not while giving block as same lock must be used. But can happen while adding new entry to parent dir. like for creating "b/a" & "b/c" in dir b.

Use Coarse lock: Take whole lock to make any change.

Per file lock: lock the inode before accessing file/directory so only one process is allowed to read/write data.

So we can keep this locks in memory & not disk. So in struct inode:
struct sleeplock lock;

Scanned by CamScanner

updated at same time to underlying storage

(Write through cache)

Buffer cache

→ Synchronize access to disk block
• Only one thread can access a disk block at a given time.
→ Cache popular block such that they don't need to be re read from disk.
→ But while writing, we write data to disk.

25/3/19

Inode

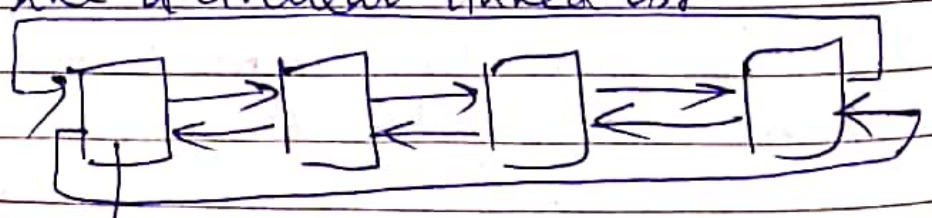
→ contains metadata corr. to file or directory
→ in-memory inode structure also contains lock.
→ Before read/write to a file xv6 acquires the lock
→ Multiple threads can't read/write to file simultaneously

Disk accesses are slow

To eliminate the cost of slow disk accesses, an in memory buffer cache is maintained by xv6. We can read from cache if data is cached but whenever we have to write, we have to write to disk.

Buffer Cache

like a circular linked list



next, prev, sectorID, char data[512]

Scanned by CamScanner

bread → buffer read
bwrite → buffer write → write disk also

Page No.
Date

So first we check buffer & if it is present, we read, else go to hard disk.

Synchronize access to buffer cache by locks.
This lock is not needed because we've taken lock on the file. So no lock needed on lower level/fine granularity.

This lock is needed because 1 block of buffer may have many inodes, so if 1 thread has updated inode1 & wants to write to disk & another thread 2 is updating inode 2 in that buffer, so partial updates of inode2 will be written as soon as inode1 is updated completely. So this is why we need buffer locks to prevent partial ~~wr~~ writes on disk.

Buffer Cache Replacement

→ LRU replacement policy is used in xv6

```
                    todo locking   after bwrite, set
                                   from dirty to clean
struct buf {
    int flags       //BUSY or DIRTY or CLEAN
    uint blockno
    :
    prev
    next
    :
    uchar data[512]
}
```

Page No.
Date

```
buf = bread (dev, . . . )
:
bwrite (
brelease (
```

Can we avoid double copy? No

Double copy happens as first from hard disk to buffer & buffer to memory.

```
Applic^n : read (fd, buf, size)
Kernel   : sec_id = offset_to_sector(fd);
           kbuf = bread (dev, sector_id);
           memcpy (buf, kbuf, ...)
```

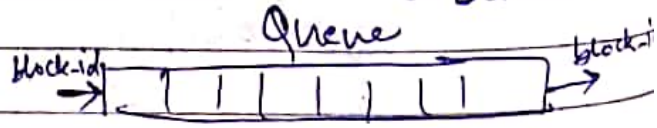
device.

We can't simply return kbuf as its in kernel memory & as it is not read only so processes may write to it so we may have trouble synchronizing.

As locks are not shared b/w processes we can do this by sharing locks in shared memory but for this we'll have to trust user.

← Device driver

Idestart



→ Wait until disk is ready.

→ Configure disk device to read/write contiguous sectors.

→ Disk interface takes sector-id of the first

Page No
Date

sector & the # of sectors to read/write data.
→ Data is written to the port 0x1F0
→ Idestart take requests from queue one by one.

Ideintr
Disk device generates an interrupt when the writing is complete or the data is ready for reading.
Data is read from port 0x1F0
In the interrupt handler, xv6 pops the buffer from the queue & initiate next req. if present in the queue.

What is disk scheduling policy in xv6?
→ FIFO (not round robin) → as its circular
Better policy would be if somehow we minimise seek time i.e. move arm
As we need to read/write contig. sectors to minimise time.

*FCFS (first come first serve)

queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53
53 - 98 - 183 - 37 - 122 - .... - 65 - 67
Total head movement = 640
Problem → not the best algorithm for performance

Page No.
Date

SSTF scheduling

→ shortest seek time first.

Queue = 98, 183, 37, 122, 14, 124, 65, 67

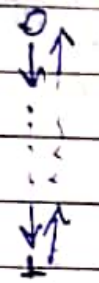
head starts at 53

53 - 65 - 67 - 37 - 14 - 98 - 122 - 124 - 183

Total head movement = 236

Problem → starvation

Elevator Algorithm (Mostly used)



- Starts from one end of the disk & moves forward to the other end of the disk, servicing in f/w req.
- After reaching other end, we'll start ~~again at this end~~ going back in opp. dir.

0-1000
1000-0

Queue = 98, 183, 37, 122, 14, 124, 65, 67

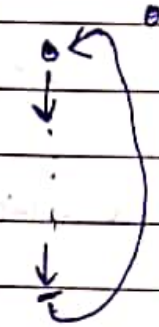
head start = 53

Say head is moving towards zero

53 - 37 - 14 - 65 - 67 - 98 - 122 - 124 - 183

No starvation!

Circular SCAN Algo.



- After reaching the other end, reset head to beginning of the disk.

Queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53

53 - 65 - 67 - 98 - 122 - 124 - 183 - 14 - 37

0-1000
0-1000

This is useful if requests closer to zero are dense!!.

Scanned by CamScanner

Page No.
Date

Crash recovery

- A file sys. must be aware that a power failure may happen at any time.

eg echo > a/b (Creating file b in directory a)

1. allocate node "x" for file b
2. Create an entry which pts to "x" in dir "a"
3. initialise inode "x".

A power failure after step 2 will leave a dangling ptr. in dir. "a". Very bad → as we may modify anything

If instead the steps are:

1. 1.
2. 3.
3. 2.

A power failure after step will create space leak. not so bad!

Assumption: • Writes to individual sector on disk are atomic.

• Even if a power failure happens in b/w the sector write, the disk has enough energy to complete the write (This is true).

Unlink (rm a/b)

1. Free data blocks of file "b"
2. Free inode b
3. Remove directory entry corr. to "b" from "a"

If a crash happens after step 1 → inode will pt. to dangling ptr.

~~If a crash happens after step 2~~ → directory will pt. to dangling ptr.

Scanned by CamScanner

So instead : 1. remove directory entry
2. free node
3. free data blocks

A space leak is not as bad as dangling references. To recover from space leaks:
→ Walk all blocks in file system which are allocated but have no references. This is called fsck in Linux. But this is very slow.

Synchronous metadata update
• Always initialize the disk before creating references

Another prob. with fsck :
mov a/b c/d
1. remove "b" (inode x) from "a"
2. add entry "d" (inode x) to "c"
After crash after step 1, the file would be deleted.
So change to : 1. add entry "d" (inode x) to "c"
2. remove "b" (inode x) from "a"
A crash after step 1 would leave the inode x with multiple references to it.

Scanned by CamScanner

maybe when cache entry is evicted.

So fsck won't fix it as it's a valid state wrt some other call.

27/3/19

Write Back Cache

faster { → Doesn't do update to disk/mem. at each write but updates at random points.

→ But this makes recovery harder.

So order of unlink might not be preserved:

1. remove dir. entry
2. free inode
3. free data blocks.

So even if we change data in cache in this order but 2 may be written to disk before 1 as it depends on write back cache.

→ So somehow we need to remember dependencies b/w writes before puting them on disk.

Logging

→ for write back cache issues

→ ~~and for mov and issues.~~

→ To eliminate the shortcomings of synchronous meta data update, xv6 implements logging.

Operation

mov a/b c/d

Changes written asynchronously but no issue after end_op. { → begin_op()

1. add an entry d ("inode x") to "c"
2. remove "b" (inode "x") from "a"

end_op()

These write are made to temp. location instead of disk, And one the logging is done & everything succeeded then we copy to disk.

XV6 guarantees the atomicity of writes enclosed with begin-op & end-op. In this case, after crash, the file sys will see ~~neither~~ either both the changes or none of them.

But a prob is → If multiple writes are done asynchronously, then it can clobber.

Syscall :

```
begin-op():
...
bp = bread(_);
bp->data[..] = ...;
log-write(bp);
brelse(bp);
...
end-op();
```

remembering that I have to do write to temp. loc^n - then to original loc^n but asynchronously.

Write to log file

instead of bwrite

as bwrite doesn't guarantee atomicity but log-write guarantee atomicity.

Page No.
Date

Logging

→ multiple sys. calls can log in parallel.
→ The data is saved on the disk & log asynchronously.
• write back cache.
→ Faster than synchronous writes.

Log file is also written asynchronously.

Group Commit

→ multiple operations

ensure no other operation is executing in parallel. As this way when we commit one buffer, it may also commit

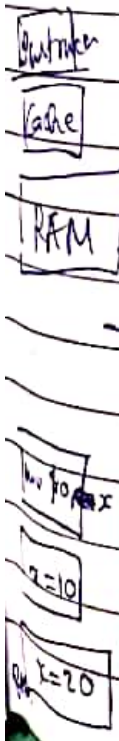
See slides. (Copy buffer to ~~another buffer~~ memory)

Copy on write optimisation with this

1/4/19

Cache → 10x faster than RAM

Hardware first looks in the cache and if it is not present then it brings to cache from RAM.



→ Cache line size is 64 bytes.
→ An entire cache line is brought to the cache when a memory byte is accessed.

Write back cache: On every update no need to write back to RAM but only when we're reading new ~~block~~ cache line.

Scanned by CamScanner

Page No.
Date

In buffer cache, write back doesn't mean replacement but it means this in cache.

Write through cache: Everytime we write in cache, we also write in RAM immediately, so every write is of speed of RAM.

$\boxed{x=10}$

$\boxed{x=10}$
RAM

Adv. with this is that other CPUs will be able to see change imm.
Another adv. is for MMIO that those MMIO will directly see the change as they observe RAM.

Cache coherence hardware is simple.
MMIO therefore have to use write through cache only. Its pte has a bit corr. to write through cache.

For write back { MESI → Cache coherence protocol
A cache line can be in one of these 4 states.

- Modified (M): doesn't match main mem. & only present in current cache.
- Exclusive (E): Matches main mem. & only present in current cache.
- Shared (S): matches main mem & present in more than 1 ~~current~~ CPU cache.
- Invalid (I): this cache line is invalid

Scanned by CamScanner

	M	E	S	I
M	x	x	x	y
E	x	x	x	y
S	x	x	y	y
I	y	y	y	y

For same cache line in 2 diff CPU caches.

REQ	P1	P2	P3
Initially	NP	NP	NP
R1	E	NP	NP
W1	M	NP	NP
R3	S	NP	S
W3	I	NP	M
R1	S	NP	S
R3	S	NP	S
R2	S	S	S

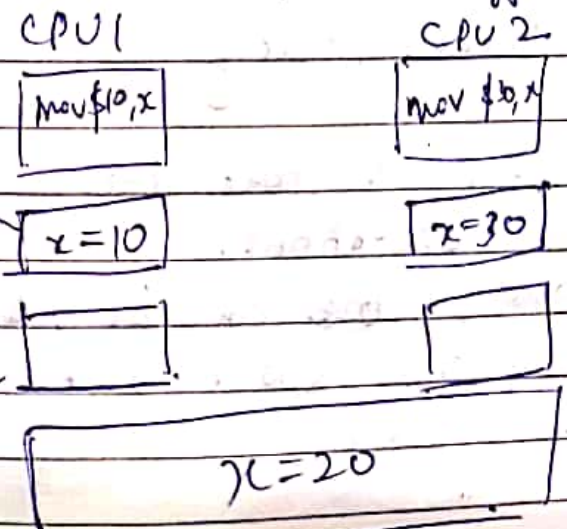
not present

Write back so it'll go to excl. mode

write back & then read from RAM for R1

only for the same data in a cache line. If cache line diff then none of this shit.

Store buffer → No cache coherency among store buffer.



Computation will proceed with this till data can be fetched to cache line from RAM.

Scanned by CamScanner

Page No.
Date

Store buffers are private to ~~each~~ CPUs.
Only the CPU can see its own store buffer.

Memory ordering

→ load & store are atomic on x86.

→ memory ordering refers to the order in which the processor issues read & write requests to the system memory.

Initially x, y are 0;

P0	P1
x = 1	y = 1
r1 = y	r2 = x

write 1 to x

read given register r1.

So order can be:

x = 1	x = 1
r1 = y	y = 1
y = 1	r1 = y
r2 = x	r2 = x

r1 = y
x = 1
y = 1
r2 = x

← reordered instruction of a CPU, which is kind of not valid as we want seq. order in one CPU.

If some hardware allows this, it is called weak memory model.

→ On x86, loads & store are not reordered with like operations.
i.e. load can't be reordered with a load and store can't be reordered with a store.

Scanned by CamScanner

P0	P1
x=1	r1=y
y=1	r2=y

So state: r1==1 && r2==0 is not possible after termination of prog.

load means loading from mem. to reg.

→ Stores are not reordered with earlier loads

P0	P1
r1=x	r2=y
y=1	x=1

So r2==1 && r1==1 is not possible.

→ loads may be reordered with earlier stores to diff. memory loc^n.

Only case where x86 reorders

P0	P1
x=1	y=1
r1=y	r2=x

So r1==0 && r2==0 is possible.

If earlier store is a locked instruc^n, only then load won't be allowed reorder!!

How to prevent reordering?

→ mfence

see slide

Page No.
Date

Locking in multiprocessor.

Peterson's sol^n (Uniprocessor)

without disabling interrupts.

```
volatile int turn;
volatile boolean flag[2];
```

For only 2 threads.

acquire:

```
flag[i] = TRUE
turn = j;
while (flag[j] && turn==j);
```

→ Put mfence here.

release:

```
flag[i] = FALSE
```

If $i=0$ then $j=1$ & vice versa.

In case of reordering:

turn = j

can be reordered with

flag[j] → in while(flag[j] && turn==j)

flag[j] can also be reorder with flag[i]=TRUE

Store flag[i]		load j		load j
load j	⟷	store flag[i]	⟷	load flag[j]
Store turn		store turn		
load ~~load~~ flag[j]		load flag[j]		
load turn				
load j				

This can be avoided by using mfence before while statement. — sync_synchronize()

in x86 in mfence.

Scanned by CamScanner

Page No
Date

Spinlock

```
struct spinlock {
    volatile unsigned int locked;
};
void acquire (struct spinlock *lk) {
    while (lk->locked == 1);
    lk->locked = 1;
}
```

Prob → 2 processes can read lock = 0 at same time on 2 diff CPUs.

lock prefix

↳ certain instruc^n can be prefixed with lock
↳ lock prefix ensures the atomicity of instruc^n
• x86 instruc^n are complex & a single instruc^n can do multiple things → read/writes.

Logwrite vs bwrite

logwrite → atomicity of multiple op.
bwrite → atomicity of single op.

End-op imm. commit ? → No, we have to wait for other op. to commit or we can do end op after 5 secs.

Can we commit just once, before the program termination ? No user mode & application mode → so user will not be displayed data. We don't display until commit.

Scanned by CamScanner

We can talk to users only through cons. or program termination.

Cache lines

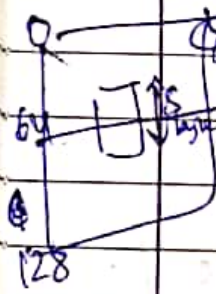
VA → PA
[4096-8192] → [0-4096]

Q Which cache line would be brought in cache when appl. accesses 4096+8 VA

Ans → 0-64 (as cache line size is 64 byte & 64 byte aligned).

Q Can one instruct need 2 cache lines?

Yes. If data is on cache line boundary say 8 byte data, half in one cache line & other half in other cache line.



Q Why does load & store reorder?

→ To increase efficiency.

→ Store buffer ki wajah se hoti hai.

Store buffer is needed to invalidate cache line in other cores to update same memory so we can update in store buffer till the time everything else is invalidated on other cores & then we can sync with cache.

Scanned by CamScanner

Page No.
Date

Q Do we need mfence in a single threaded application?

A NO as we don't need to read buffer from other caches. as no other thread can change data.

Lock prefix

→ Flushes all store buffers
→ lock cache line which has to read/write/execute locked instruc^n
lock add $1, 0x1000.
→ Load/store can't be reordered across lock instruc^n.

Invalidate all other data cache line with same data

Disadv.

→ very slow.
→ lock cache lines (slow the execution of other cores.

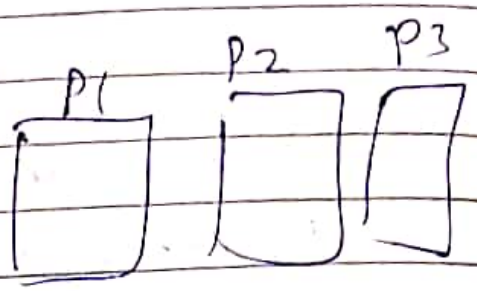
(slides)

Scanned by CamScanner

Page No.
Date

Cache line bouncing

```
label 1:
  lock add $1, lockvar
  cmp $1, lockvar
  je label2
  lock sub $1, lockvar
  jmp label1
label2:
```



→ On every update, it will invalidate cache line

→ On every read, it will fetch cache line in shared mode.

So there will be multiple bouncing of cache line.

This problem is worse in more no. of cores like 32 cores. This is because more invalidation is needed so time increases.

If other threads are accessing diff data on diff cache then there is no cache bouncing prob.

False sharing prob → Cache line bouncing for diff data on same cache line.

Scanned by CamScanner

Page No.
Date

locked xchg.

Spinlock

```
struct spinlock {
    volatile unsigned int locked;
}

void acquire (struct spinlock *lk){
    pushcli();
    while (atomic_xchg(&lk->locked,1)!=0);
    __sync_synchronize();
}
```

memory barrier. That all loads & stores must be completed before going forward. (mfence)

```
void release(struct spinlock *lk){
    __sync_synchronize();
    lk->locked=0;
    popcli();
}
```

Improvization ?

```
void acquire
    pushcli();
    while (atomic_xchg(&lk->locked,1)!=0){
        while(lk->locked!=0);
    }
    __sync_synchronize();
}
```

Prob → No fairness. Starvation.

Scanned by CamScanner

Spinlocks → Used in interrupt handlers as we can't sleep there.

Ticket Spin Lock

→ Maintain order in which they've arrived.

```
struct lock {
    volatile unsigned head; // initially 0
    " " tail; // " 0
}
```

acquire :

```
Oldtail : atomic_xadd (&lockvar->tail, 1);
while (oldtail != lockvar->head);
```

release :

```
lockvar->head++;
```

(Even if integer overflows i.e. becomes zero, even then it is not equal).

So this ensures fairness.

8/4/15

Reader-Writer Locks

→ Multiple readers can concurrently execute in the critical section

→ Only single writer is allowed in the critical section.

Scanned by CamScanner

Page No
Date

```
volatile unsigned lockvar = 10000;
read-acquire :
    while(atomic_sub(&lockvar,1) <0){
        atomic_add (&lockvar,1);
        while(lockvar<0);
```

read-release :

```
    atomic_add(&lockvar,1);
```

write-acquire :

```
    while (atomic_sub (&lockvar,10000)!=0){
        atomic_add(&lockvar,10000);
        while(lockvar != 10000);
    }
```

write-release :

```
    atomic_add(&lockvar,10000);
```

Prob → Starvation.

atomic_sub returns final value.

Soln : whenever a writer arrives, no new readers are allowed.

Another prob → Even in case of just readers, cache line bouncing problem.

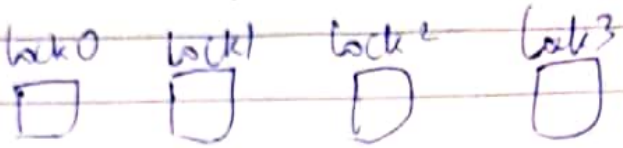
Soln Big reader lock

Scanned by CamScanner

Page No.
Date

Big Reader Lock

→ each CPU has a private lock.
→ locks on diff CPUs belong to diff cache line.
→ reader acquires the lock corr. to current CPU.
→ writer acquires locks of all CPUs.



reader:

```
acquire(&lock of curr cpu)
//critical sect.
release(&lock of curr cpu)
```

} assumption → thread is not scheduled on diff cpu / new thread not scheduled on this.

So we do:

```
preempt_disable();
```

↓ interrupts are allowed, but thread can't be preempted.

} no sleeplock here

```
preempt_enable();
```

Alignment → locks of cpu on diff cache line

```
struct spinlock {
    unsigned long val;
} __attribute__(aligned(4096))
```

Page No.
Date

struct spinlock lock;
& lock is guaranteed to be div. by 4096.
size of (struct spinlock) would be nearest multiple of 4096.

To be cache line aligned → aligned(64)

This BigReader lock is only for kernel as user can take preempt-disable forever.

Disadv → (1) Locks for each CPU
(2) Lock size increased to cache line size.

Struct spin-lock brlocks [NUM-CPUS]; // this should be cache-line size aligned.

read-acquire:
preempt_disable();
spinlock(&brlocks[cur-cpuid]);
acquire

read after doing preempt disable.

read-release:
spinlock-release (&brlocks[cur-cpuid]);
preempt-enable();

write-acquire:
for (i=0; i<num-cpus; i++) {
spin-lock

no preemption

Page No.
Date

Adv: cache friendly, no bounces.
Disadv: acquisition of writer lock is slow.
space of lock var. is prop. to # of CPUs
require information regarding cache line size.

Can we use reader lock in user mode?
No (Preemption ki wajah se).

Read Copy Update.

→ no reader lock
only for special cases
→ only for OS kernel
→ only works when the writer is compatible with the lock free reader.

eg In slides.
Search & delete routine for linked list.
So we won't free the old node unless we're sure that no one is using it/ has searched it.
With assert it mayn't work as due to partial updates.

RCU Lock
→ Preemption is disabled in read critical secⁿ
→ frees are delayed in the write critical secⁿ.

Scanned by CamScanner

Page No.
Date

A point in prog. where it is guaranteed that a thread can't be in read critical section.

Schedule is a ~~crescent~~ quiescent point as in read critical sec^n, we have preempt disabled so if schedule is called that mean that thread is out of read critical sec^n.

So free schedules itself a thread on all cores & after that, it will actually free once it has been scheduled in all cores.

wait_for_rcu : It waits for quiescent state on all CPUs.

rcu_free calls wait_for_rcu.

```
wait_for_rcu() {
  /*cpus_allowed field controls the scheduling
    of a thread on specific sets of CPUs */
  cpus_allowed = current -> cpus_allowed;
  current -> cpus_allowed = all_cpus;
  while (remove_curr_cpu(&current->cpus_allowed) &&
        current -> cpus_allowed) {
      schedule();
  }
  current -> cpus_allowed = cpus_allowed;
}
```

that for cpu:
1 2 3 4
1 4
2

If we're in interrupts, we can't do this. So we need to do this after coming out of interrupts. So we can put all nodes to be freed in a queue and periodically we can free the whole queue together (so this will even dec. time).

We won't be able to do RCU locks on tree. But if we allow copy of tree, we will be able to do it:

→ copy tree.

→ modify copied tree.

→ Change the root : oldroot = newroot

RCU locks used when reads are more than writes.

TLB Invalidation

→ How to invalidate TLB on other cores when page table entries are modified.

→ Polling a global memory flag. ~~Have~~ Has latency.

To remove latency → IPI.

IPI → Inter processor interrupt

A processor can send IPI on other cores.

The corr. interrupt handler is called when an IPI.

Scanned by CamScanner

Page No
Date

Scheduling

Multilevel feedback queue scheduler

- has multiple queues.
- each queue has diff priority
- I/O intensive & interactive workloads are given highest priority
- compute-intensive appl. have lower priority.
- priority of low priority process is inc. over time if it is not getting scheduled (to avoid starvation)
- We need to see dynamically whether an appl. is IO intensive or CPU intensive. So it's nature can change.
- we can see whether application is IO intensive or not by seeing that whether it yields itself before certain threshold (as it needs to wait for IO). → time quantum

See slides.

- Protec. rings?
  - → middle of instruc. by fn ptr.
  - → ret. addr. changed to middle by array out of bounds access

- Java.
  Can we do w/o protec. rings?
  No arbitrary type casts allowed.
  No free in java.

Scanned by CamScanner

Page No.
Date

Q How does java detects free objects?
Mark all objects that can be reached from the current place & free the unmarked vars. This is done infreq. or when there is memory pressure. They try to do it concurrently with the exec^n of prog. This is time consuming as we've to traverse all memory occupied.
Disadv: is that it produces latency in the ~~code~~ prog. as it stops it & goes to mark vars & free them. so it has to take snapshot.

eg struct list *node = null
~ ~ *ptr = list
node = ptr
ptr = null

Concurrent exec^n mayn't traverse the list at all to mark it as node & ptr can be null.

Memory Isolation: Java ensures this b/w 2 processes if we trust the allocator so appli can't access arbit. memory, it can only access the object & that too within its limit. So VA not needed. So no paging, TLB

So context switch for sys call to OS will be just a func^n call. We need to copy args. in the OS. (from usr proc. to kernel proc as both have diff garbage collectors)

Scanned by CamScanner

Page No.
Date

Similarly for IPC, we'll have send, receive APIs & we'll context switch. Context switch API just copies args & does func call.

Device drivers can still access OS objects in linux.
But in java, we can have diff. processes for device drivers.
This will ensure that even if there's bug in a device driver, it is isolated & only this will crash & not others.
Device drivers can do sys call to access OS & sys call isn't expensive.

But java is not efficient (fast) due to mainly garbage collection.
Small part can be written in unsafe APIs (not in java) like allocator & garbage collector (like in C/assembly) but amount of this code is very small.

Can we design a a relatively secure OS in C.

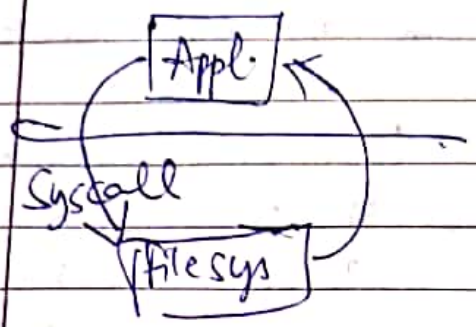
Microkernel

→ In C/C++. OS should do the minimal work. Everything else can be run in user mode. like device drivers, file sys.

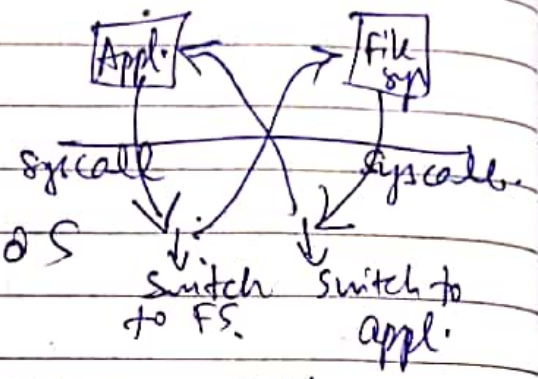
Scanned by CamScanner

OS does only:
- address space
- threads
- IPC (send & recieve)

Normal



Microkernel



→ 2 page table changes.
⇒ Error recovery is possible from file sys.

Adv.
- Fault tolerance: OS can restart the file sys process if it crashes.
- Security.

Disadv:
- Sys calls are slow
- page table switch
- addition 2 transac^n b/w diff priv. level.

Scanned by CamScanner

Page No.
Date

Cloud Security

Linux kernel isn't scalable on 80 cores due to cache coherance & cache line bouncing.

Scanned by CamScanner