**How to Understand Process :: steps followed**

* **Memory Management Algorithm** (specially on Intel architecture)
* Real Mode: - the general-purpose registers are truncated into 16-bit registers, as are error flag and instruction pointer registers.

**A** machine in real mode can address 1MB of DRAM. This implies that only 20 address lines are used in real mode. The address of a byte in memory, for a processor real mode, is formed by adding an offset address to a segment address. The result of the sum is always a **20-bit value**, which confirms our suspicion that there are 20 address lines.

AX : DS => AX[0] + DS , where AX and DS represents offset and segment address respectively.

**T**he address of byte in real mode maps directly to a “real” byte that is placed in processor’s address line in physical memory.

* Protected Mode: - EAX 32-bits offset : DS (segment selector). Segment selector is a 16-bits data structure containing three fields. The index filed, table type and privilege. The index field is responsible in saving descriptor table entry.

|  |  |  |
| --- | --- | --- |
| 15 Index Field 3 | Table Type 2 | 1 Privilege Level 0 |

**Q.N** if only 13 bit of index-field tells about GDT entry, is it possible; the maximum size of GDT entry is limited by index-field instead of GDT size limit which is found inside GDTR?

**QN.EXP**: - segment selector size \* GDT size == will be \* (2^16 -1 - 7) but less than (2^32 – 1- (2^20 -1))== maximum GDT size

**Q.N** how does the processor map a segment selector’s index to a descriptor?

**Ans** The processor takes the index, specified by the segment selector, multiplies the index by eight (as in 8 bytes because the descriptors are 64 bits (8 byte) in length), and then adds this product to the base address specified by GDTR or LDTR.

**T**he descriptor table might be LDT or GDT. Base address of GDT is stored inside GDTR 48-bits in length. 0-15 stores GDT size limit and the rest 16-47 saves GDT base address. **B**esides all the explanation earlier about GDTR, the first bit is the **SS** flag, which indicates whether the segment is a system segment or just a normal code/data segment. A **system segment** provides interrupt handling and multitasking services.

**S**egment descriptor store the base linear address of the memory segment they describe and other metadata

* Dirty bit which is found inside **PTE** will be cleared when a page is loaded in to DRAM for the first time. The processor will then set the dirty bit the first time that the page is updated. Read write flag (bit 1) and user-supervisor flag (bit 2). User-supervisor flag can be used to institute a two-ring security scheme. If a user application attempts to access pages that are in supervisor mode, the processor will generate a **page fault**.
* **CR0** is used to control the processor mode and state of the processor. First bit (the lowest-order bit) is a **PE** (protected enabled) flag. **PG** flag is set; the processor is enabled for paging.

**C**R2 is used to store the linear address that has caused a page fault.

**C**R3 holds the base address of the page directory. The other two flags (**PCD** and **PWT**) in this register are related to caching.

**Q**.**N**: - How can I relate this two flags with my current buffer caching mechanism?

**C**R4 is used to enable a couple of advanced mechanisms**. PAE** flag is enabled four extra address lines when it is set. This would bring the number of address lines to 36. PG flag must be set in CR0 in order for PAE to be effective. **PSE** is cleared to zero; the page size is 4KB. If PSE is set, the page size is 4MB. If both PAE and PSE are set, the page size is 2MB.

* On Intel hardware, there are three types of addresses (Physical, Logical and Linear).

**P**hysical address is the address of a byte in DRAM. It is the value that the processor places on its address lines in order to access a value in chip-based memory

**Logical address** is the address that is specified by the segment register and the general register. In real mode logical address correspond to a physical address.

**Linear address** is the 32-bit value that is formed using the base address value in the **segment descriptor** and the offset value in the general register.

* **There are two ways** that user applications can allocate memory: **Compiler-based** allocation and **heap** allocation.

**Compiler-based** allocation

* User application typically have their address space divided into four types of regions: Code section, Data section, Stack and Heap. **Compilers control** how memory is allocated, arranged, accessed, and updated in every section, **except the heap**.

**Q.N**:- According to the above statement, does it mean that **file linear address** can’t control the heap. If it is true, do we need to generate linear address for heap alone?

**M**anaging the heap is the domain of user libraries and virtual machines.

**My\_Unders**: - when user library requests memory for heap purpose, Virtual machines handles both the request and the mapping (linear address).

**D**ata section: - of an application traditionally supplies static memory and most compilers will construct data sections to serve as storage for global data.

**C**ode Section: - It is data as long as you don’t execute it.

**S**tack: - A stack is a sequence of bytes in memory that act like a first-in-last-out (FILO) data structure. There are two basic ways to manipulate a stack so memory can be allocated and freed: **PUSH**/**POP** instructions and integer arithmetic. The PUSH instruction causes the stack pointer to be decremented and the created space will be populated by the PUSH instruction’s operand. The reverse goes for POP instruction.

The Stack is predictable and the **heap** is chaotic, since a stack enforces a certain degree of regularity. In stack you always know where the next memory chunk starts.

Stacks are used to implement high-level features like recursion and variable scope.

**M**anaging the stack to facilitate a function call is the responsibility of both the procedure that is invoking the function and the function being invoked

:: **Steps** which will be used to invoke a procedure and pass its arguments

: - Push the current function’s state onto the stack

: - Push the return value onto the stack

: - Push function arguments onto the stack this is similar step inside **pintos**

: - Push the return address onto the stack Check start\_process inside

: - Jump to the location of the procedure process.c

:: **Steps** which shall be followed by the function being invoked

: - Push EBP on to the stack (to save its value).

: - Copy the current ESP value into EBP.

: - Execute the function’s instructions.

:: **After** the function has done its thing and is ready to return, it will do the following

: - Reclaim local storage.

: - Pop EBP off the stack.

: - Pop the return address off the stack.

: - Jump to the return address. **The** Intel RET instruction will usually take care of the last two steps.

:: **Once** the invoked function has returned, the invoking function will do the following

: - Pop the function arguments off the stack

: - Pop the return value off the stack

: - Pop the saved program state off the stack

**Q.N**: - 1. in the above steps, there is no notification which talks about using return value or assigning a value to it, so where is it? 2. Is there a no relationship between return value and return address?

My\_Unders: - The invoked function will refer the return address to assign the return value to it.

: -

**HEAP allocation**

* **The Structure of Process (UNIX)**
* The state of process defined in **process table entry**, **U area**

**P**rocess state: process is executing in user mode, process is executing in kernel mode, process is ready but not running …etc

**T**wo kernel data structures **process table entry** and the **u area** describe the state of the process. The process table contains fields that must always be accessible to the kernel, but the u area contains fields that need to be accessible only to the running process.

Process table:**????????????????**

* The principles of memory management for processes and for kernel and how the OS and HW cooperate to do virtual memory address translation.
* Context of a process
* How kernel saves context of a process during interrupt occurrence.
* Various System call algorithms that manipulate the process address space.
* **Assembly 8086 chip**
* **Intel** VS **AT&T** syntax(check out the documentation which is found inside ASM folder)

NOTE: if a program could run on the 8086 chip, they will all run on any IBM compatible PC, as the 80386, 80486 and Pentium (I, II, III …) chips all are designed to run 8086 code.

* Segment Registers : **CS** (code segment), **DS** (data segment), **ES** (Extra Segment pointer), **SS** (Points to start to stack segment)
* Pointer Register: **IP** (Instruction Pointer), **SP** (Stack Pointer), **BP** (Base Pointer)
* Data transfer Register: **SI** (Source Index), **DI** (Destination Index) EFLAGS ( set of flags)
* The INT (interrupt) instruction clears a road for keyboard, disk, mouse, or screen usage. Each interrupt though, has a number of sub-functions which select the individual task that the function has to do.
* Hello world program written in c and converted to assembly

#include <stdio.h>; -> int main() -> printf (“hello world!”); -> retrun 0;

^ ^ ^ ^ ^

| | | | | **After converting to assembly**

.file “hello.c” //**.file** directive states the original source file name

.section .rodata //.**rodata** section used for read-only data variables

.LC0:

.string “Hello World!”

.text //.**text** section, which is used for the code of the program

.globl main //.**globl** globally visible

.type main, @function

main: // from the **main**: label till the **ret** instruction is the actual code of the program

pushl %ebp

movl %esp, %ebp

andl $-16, %esp //optimization, multiple of 16 bytes

subl $16, %esp //subtracting 16 bytes from the current ESP for local variables

movl $.LC0, (%esp) //coping string to the top of stack

call printf // OR you can use (puts)

movl $0, %eax

leave

ret

.size main, .-main // -main holds the exact size of function main

**Q.N**:- is it possible that during andl, esp locates other address instead of stack?

* **X86 calling conventions**

Registers EAX, ECX, and EDX are caller-saved, and the rest are callee-saved.

* **ELF file format**
* **Linux Kernel development 3rd edition**
* process Management

The process descriptor (struct task\_struct) contains the data that describes the executing program, open files, the process’s address space, pending signals, the process state …etc

Process descriptor is stored at the end of the kernel stack of each process.

* Using Fork

**D**uplicate the parent process. Code, data, stack(Which one exactly?)

**T**he parent process continues executing the program from the point that fork was called

**T**he child process, too, executes the same program from the same place.

* Using Exec

Replace the running program in a process with another program. Because of this reason, it never returns unless an error occurs.

* **Register Usage in MIPS ABI**
* **The** ABI gives well-understood functions to each of the registers in the general purpose register set.

**Return address** (r a) register is assigned the return address when a function call is made. Software will put this value on the stack if the called function itself calls further function.

**Frame pointer** (fp) register points to the base of the stack frame for the current function.

**Global pointer** (gp) register used to point to a poll of global data that can be commonly referenced by all functions.

A function can use registers t0-t9 freely, but if it calls another function they may be overwritten. A function may not overwrite the contents of s0-s7, and must preserve their original contents if it wants to use them. Hence, s0-s7 is callee-saved, whereas t0-t9 are caller-saved registers.

* Call return sequencing

**Step 1 C**all sequence: - save caller-saved registers, copy arguments to stack or regs (incoming args), call the function

**Step 1.1** Function prologue: - Allocate callee’s stack frame, Reposition frame pointer, and Save callee-saved registers

**Step 1.2** Execute body of function

**Step 1.3** Function Epilogue: - Restore callee-saved registers, restore frame pointer, de-allocated callee’s stack frame and return to caller

**Step 2** Return sequence: - Restore caller-saved registers

* **Pintos code understanding (thread , process)**
* Each **thread structure** is stored in its own 4kb page. The thread structure itself sits at the very bottom of the page (at offset 0). In Linux concept **Program descriptor**.

**T**he rest of the page is reserved for the **thread’s kernel stack**, which grows downward from the top of the page (at offset 4kb).

NOTE: Since we don’t have independent ‘kernel stacks’, there is a limitation in ‘**struct thread’** size and also **kernel stack** is not allowed to grow too large, because both information found at the same page. The **magic** member detects stack overflow.

**P**intos dfn:- eip (next instruction to be executed by the interrupted thread OR return address), esp (the interrupted thread’s stack pointer)

* **HOW TO PREPARE** **THREAD (process)**

**Thread\_create()**Creates a new kernel thread named NAME with the given initial PRIORITY, which executes FUNCTION passing AUX as the argument, and adds it to the *ready queue*. Returns the thread identifier for the new thread, or TID\_ERROR if creation fails (**fork** concept).

**I**f thread\_start() has been called, then the new thread may be scheduled before thread\_create() returns. It could even exit before thread\_create() returns. Contrariwise, the original thread may run for any amount of time before the new thread is scheduled. Use **semaphore** or some other form of synchronization if you need to ensure ordering.

* **Contents** (kernel\_thread\_frame, switch\_entry\_frame and switch\_threads\_frame)

**s**truct switch\_threads\_frame

{

uint32\_t edi; /\* 0: Saved %edi. \*/

uint32\_t esi; /\* 4: Saved %esi. \*/

uint32\_t ebp; /\* 8: Saved %ebp. \*/

uint32\_t ebx; /\* 12: Saved %ebx. \*/

void (\*eip) (void); /\* 16: Return address. \*/

struct thread \*cur; /\* 20: switch\_threads()'s CUR argument. \*/

struct thread \*next; /\* 24: switch\_threads()'s NEXT argument. \*/

};

**s**truct switch\_entry\_frame

{

void (\*eip) (void);

};

**s**truct kernel\_thread\_frame

{

void \*eip; /\* Return address. \*/

thread\_func \*function; /\* Function to call. \*/

void \*aux; /\* Auxiliary data for function. \*/

};

NOTE: asm ("mov %%esp, %0" : "=g" (esp));

/\* Copy the CPU's stack pointer into `esp', and then round that down to the start of a page. Because `struct thread' is always at the beginning of a page and the stack pointer is somewhere in the middle, this locates the curent thread. \*/

==

*Kernel\_thread\_frame \*kf;*

*Switch\_entry\_frame \*ef;*

*Switch\_threads\_frame \*sf;*

*t = palloc\_get\_page (PAL\_ZERO);*

*t->stack = (uint8\*)t + PGSIZE; ( this is taken from* ***init\_thread****())*

*kf = t->stack = t->stack – sizeof (\*kf);*

*kf->eip = NULL;*

*kf->function = function;*

*kf->aux = aux;*

*ef = t->stack = t->stack – sizeof (\*ef);*

*ef->eip = (void (\*) (void)) kernel\_thread; I explanation (kernel\_thread)*

*sf = t->stack = t->stack – sizeof (\*sf);*

*sf->eip = switch\_entry; II explanation (switch\_entry)*

*sf->ebp = 0;*

*thread\_unblock (t); III explanation (thread\_unblock (\*t))*

==

**Explanation**

**I kernel\_thread (thread\_func \*function, void \*aux)** {

function (aux);

thread\_exit ();

}

**II (switch\_entry) ---- check switch.S** for full description.

**III thread\_unblock (struct thread \*t)**{

Old\_level = intr\_disable();

List\_push\_back (&ready\_list, &t->elem);

intr\_set\_level (old\_level);

}

* What the above code did is

|  |  |
| --- | --- |
| Stack 4KB (1 page) | location |
| \*aux | 4095 B |
| \*function |  |
| \*eip (return address) |  |
| **Kernel thread frame (entry)** |  |
| void (\*eip) (void) |  |
| **Switch entry frame** **(entry)** |  |
| struct thread \*next |  |
| struct thread \*cur |  |
| void (\*eip) (void) |  |
| ebx |  |
| ebp |  |
| esi |  |
| edi |  |
| **Switch threads frame (entry)** |  |
| \* |  |
| \* |  |
| \* |  |
| Magic number(data check up) |  |
| struct thread (entry) | 0 B |

* When do **schedule ()** function is called?

During thread\_block(), thread\_exit() or thread\_yield().

schedule()

* **O**bjective is to switch current thread, which must be running thread, to next thread, which must also be running **switch\_threads**(), returning current thread in next’s context.

# struct thread switch\_threads (struct thread\* cur, struct thread \*next);

* Switch function works by assuming that the thread we’re switching into is also running switch\_threads(). Thus, all it has to do is preserve a few registers on the stack, then switch stacks and restore the registers. As part of switching stacks we record the current stack pointer in CUR’s thread structure.
* **Switch\_threads**(+)

**: 1-** saves registers on the stack

**: 2-** save the CPU’s current stack pointer in the current struct thread’s stack member

**: 3-** restore the new thread’s stack into the CPU’s stack pointer

**: 4-** restore registers from the stack, and returns.

**Those 4 steps in detail**

**switch\_entry:**

**N**ote: According to x86 calling convention, eax, ecx, and edx are caller-saved registers but the rest are callee-saved.

: 1- Discards switch\_threads () arguments. **addl $8. %esp.**

Q.N How do we discard by adding 8 to the address of esp?

A.N. addl simply navigates the current esp address by 64bit or 2 arguments (\*cur and \*next) to discard from using it.

: 2- Call thread\_schedule\_tail (prev). **pushl %eax**

: 3- **call thread\_schedule\_tail**

: 3.1- **addl $4, %esp**

: 4- Start thread proper. **Ret**

|  |  |  |
| --- | --- | --- |
| \* | | |
| 3  **addl $4,esp** | 1  **addl $8, %esp**| \*next | 2  **pushl %eax** (thread\_schedule\_tail (prev) |
|  | 1  \*cur |  |
| \* | | |

**schedule**(): - schedules a new process by disabling interrupt and changing the running process’s state from running to some other state. **Calls** switch\_threads (cur, next);

**switch\_threds:**

: **1**- pushes those callee-saved (ebx, ebp, esi, edi) registers to current thread stack (Yonas)

**Q.N** In which stack (current thread or new one)?

**A.N** it is inside caller (current thread) stack. Including those caller-saved register values will be handled by the caller itself.

: **2**- get offsetof (struct thread, stack). **.globl thread\_stack\_ofs**

**2.1**- **move thread\_stack\_ofs, %edx** (Description: offset of stack locates switch\_threads\_frame **entry**)

: **3**- save current stack pointer to old thread’s stack, if any

**3.0**- **movl SWITCH\_CUR (%esp), %eax** (Description: SWITCH\_CUR is 20, refers offset of thread \*cur from switch\_threads\_frame structure.)

**3.1**- **movl %esp, (%eax, %edx, 1)**

: **4**- Restore stack pointer from new thread’s stack.

**4.0**- **movl SWITCH\_NEXT (%esp), %ecx**

**4.1**- **movl (%ecx, %edx, 1), %esp** (Description: again we use **edx**, our previous thread stack switch\_threads\_frame entry offset, to assign new esp value. This is possible b/c both process have a similar structure (thread)).

: **5**- Restore caller’s register state.

**5.0**- pops out (edi, esi, ebp, and ebx) values from \*next thread stack)

: **6**- **ret**

* **thread\_schedule\_tail** ():- completes a thread switch by activating the new thread’s page tables, and, if the previous thread is dying, destroying it. This function is normally invoked by thread\_schedule() as its final action before returning, but the first time a thread is scheduled it is called by switch\_entry().
* **running\_thread()** :- Copy the CPU’s stack pointer into ‘esp’, and then round that down to the start of a page. Because `struct thread` is always at the beginning of a page. **asm (“mov %%esp, %0” : “=g” (esp)); pg\_round\_down (esp);**
* **init\_thread()** :- t->stack = (uint8\_t \*) t + PGSIZE; list\_push\_back (&all\_list, &t->allelem);
* **Schedule ()**: - records the current thread in local variable cur, determines the next thread to run as local variable next (by calling **next\_thread\_to\_run()**), and then calls switch\_threads() to do the actual thread switch. The thread we switched to was also running inside switch\_threads(), as are all the threads not currently running, so the new thread now returns our of switch\_threads(), returning the previously running thread.
* **Completes a thread switch** by activating the new thread’s page tables, and, if the previous thread is dying, destroying it. At this function’s invocation (**thread\_schedule\_tail**), we just switched from thread PREV, the new thread is already running, and interrupts are still disabled. This function is normally invoked by thread\_schedule() as its final action before returning, but the first time a thread is scheduled it is called by switch\_entry()

**I**t’s not safe to call printf() until the thread switch is complete. In practice that means that printf()s should be added at the end of the function. **A**fter this function and its caller returns, the thread switch is complete.

* /\* Interrupt stack frame. \*/

struct intr\_frame

{

/\* Pushed by intr\_entry in intr-stubs.S.

These are the interrupted task's saved registers. \*/

uint32\_t edi; /\* Saved EDI. \*/

uint32\_t esi; /\* Saved ESI. \*/

uint32\_t ebp; /\* Saved EBP. \*/

uint32\_t esp\_dummy; /\* Not used. \*/

uint32\_t ebx; /\* Saved EBX. \*/

uint32\_t edx; /\* Saved EDX. \*/

uint32\_t ecx; /\* Saved ECX. \*/

uint32\_t eax; /\* Saved EAX. \*/

uint16\_t gs, :16; /\* Saved GS segment register. \*/

uint16\_t fs, :16; /\* Saved FS segment register. \*/

uint16\_t es, :16; /\* Saved ES segment register. \*/

uint16\_t ds, :16; /\* Saved DS segment register. \*/

/\* Pushed by intrNN\_stub in intr-stubs.S. \*/

uint32\_t vec\_no; /\* Interrupt vector number. \*/

/\* Sometimes pushed by the CPU,

otherwise for consistency pushed as 0 by intrNN\_stub.

The CPU puts it just under `eip', but we move it here. \*/

uint32\_t error\_code; /\* Error code. \*/

/\* Pushed by intrNN\_stub in intr-stubs.S.

This frame pointer eases interpretation of backtraces. \*/

void \*frame\_pointer; /\* Saved EBP (frame pointer). \*/

/\* Pushed by the CPU.

These are the interrupted task's saved registers. \*/

void (\*eip) (void); /\* Next instruction to execute. \*/

uint16\_t cs, :16; /\* Code segment for eip. \*/

uint32\_t eflags; /\* Saved CPU flags. \*/

void \*esp; /\* Saved stack pointer. \*/

uint16\_t ss, :16; /\* Data segment for esp. \*/

};

**C**heck Out: - if the system segment is enabled inside GDTR first bit (**SS**).

**NOTE for pintos**

**Init\_ram\_pages**: - amount of physical memory, in 4 KB pages.

**paging\_init (**void):- Populates the base page directory and page table with the kernel virtual mapping, and then sets up the CPU to use the new page directory. Points **init\_page\_dir** to the page directory it creates.

**pagedir\_create**: - creates a new page directory that has mappings for kernel virtual addresses, but none for user virtual addresses.

**init\_page\_dir**: - page director with kernel mapping only. Example inside paging\_init(), pd=init\_page\_dir = palloc\_get\_page (PAL\_ZERO); , asm volatile (“movl %0, %%cr3” : : “r” (vtop (init\_page\_dir)));

**TSS:** stack switching for interrupts that occur in user mode. If interrupt occurs in user mode (ring 3), the processor consults the ss0 and esp0 members of the current TSS to determine the stack to use for handling the interrupt.

**How to Handle Interrupt** or trap gate works

**: 1-** If the code interrupted by the interrupt is in the same ring as the interrupt handler, then no stack switch takes place. **TSS is irrelevant in this case**.

**: 2-** If the interrupted code is in a different ring from the handler, then the processor switches to the stack specified in the TSS for the new ring. This is the case for interrupts that happen when we are in **user space**. Because we are in user space, we know that the current kernel stack is not in use, so we can always use that. Thus, when the scheduler switches threads, it also changes the TSS’s stack pointer to point to the new thread’s kernel stack.

* **Struct tss**

struct tss

{

uint16\_t back\_link, :16;

void \*esp0; /\* Ring 0 stack virtual address. \*/

uint16\_t ss0, :16; /\* Ring 0 stack segment selector. \*/

void \*esp1;

uint16\_t ss1, :16;

void \*esp2;

uint16\_t ss2, :16;

uint32\_t cr3;

void (\*eip) (void);

uint32\_t eflags;

uint32\_t eax, ecx, edx, ebx;

uint32\_t esp, ebp, esi, edi;

uint16\_t es, :16;

uint16\_t cs, :16;

uint16\_t ss, :16;

uint16\_t ds, :16;

uint16\_t fs, :16;

uint16\_t gs, :16;

uint16\_t ldt, :16;

uint16\_t trace, bitmap;

};

**tss\_update ():**- tss->esp0 = (uint8\_t \*) thread\_current() + PGSIZE;

**start\_process (...):** - loads a user process and starts it running. 1- Initialize interrupt frame and load executable (.gs .fs .es .ds .ss = user data selector (0x23), .cs = user code selector (0x1B), .eflags = Interrupt flag (0x00000200) | 0x00000002 (must be set)) 2- Fill the stack space accordingly and control the esp

|  |  |
| --- | --- |
| Stack | |
| Address | Name | | Data | Type |
| 0xbffffff8 | argv[2] […] | | ‘foo\0’ | Char [4] |
| 0xbffffff5 | argv[1] […] | | ‘-l\0’ | Char [3] |
| 0xbfffffed | argv[0] […] | | ‘/bin/ls\0’ | Char [8] |
| 0xbfffffec | Word-align | | 0 | Uint8\_t |
| 0xbfffffe8 | argv[3] | | 0 | Char \* |
| 0xbfffffe4 | argv[2] | | 0xbffffff8 | Char \* |
| 0xbfffffe0 | argv[1] | | 0xbffffff5 | Char \* |
| 0xbfffffdc | argv[0] | | 0xbfffffed | Char \* |
| 0xbfffffd8 | argv | | 0xbfffffe8 | Char \*\* |
| 0xbfffffd4 | argc | | 3 | Int |
| 0xbfffffd0 | Return address | | 0 | Void (\*) () |

3- **Start the user process by simulating a return from an interrupt, implemented by intr\_exit.** Intr\_exit takes all of its arguments on the stack in the form of a `struct intr\_frame`, we just point the stack pointer (%esp) to our stack frame and jump to it. (asm volatile (“movl %0, %%esp; jmp intr\_exit” : : “g” (&if\_) : “memory”); ‘***if\_*** *is an interrupt structure object*’

**Q.N**: - if the above step is to start a process, how do we handle two processes switching and run one of them properly?

**intr\_entry**: - sa ve caller’s registers, set up kernel environment, and finally call interrupt handler.

**Intr\_exit**: - restores the caller’s registers, discards extra data on the stack, and returns to the caller.

**intrNN\_stub**: - is in which an internal or external interrupt starts, which push the `struct intr\_frame’ frame pointer, error\_code, and vec\_no members on the stack, then jump here.

**load** (..) :- loads an ELF executable from FILE\_NAME into the current thread. Stores the executable’s entry point into \*EIP and its initial stack pointer into \*ESP. L-1:- it will create page directory for the current thread. L-2:- it will open the file structure. L-3:- it will read the ELF header. L-4:- it will read the executable header (Program header) L-5:- it will read the file **offset** and **file linear address**. L-6:- loads the file into memory based on the offset and linear address. (The last step mainly done by **load\_segment**() function)

**3**.1 **Qn**: - According to Intel architecture, **linear address** is formed from offset which is found from general register (EAX) and Segment descriptor (GDT OR LDT). From this statement, if we look it from the reverse, **file linear address** is used to load and used as a linear address for the program file. Is it possible to generate EAX and GDT value from **file linear address?**

**ANS**: - Yonas said to me, file linear address means EAX offset address indirectly without including segment descriptor. **Justify** more on this topic!!!

**load\_segment:**- Loads a segment starting at offset OFS in file at address UPAGE.

**Process\_activate** (void):- Sets up the CPU for running user code in the current thread. It will do its job by **activating thread’s page table** and **set thread’s kernel stack** for use in processing interrupts.

Pagedir\_create (void):- creates a new page directory that has mappings for kernel virtual address, but none for user virtual addresses.

Pintos Questions

**1**- Inside paging\_init (), there is a statement which says **uintptr\_t paddr =** page \* PGSIZE and char \*vaddr = ptov (paddr). According to the above statement it just converted random integer number by assuming like a physical address which starts from zero up-to some max value. How is this possible?

**Ans Q1**: - it is not actually used as a physical address; instead it is **used like virtual address**

**2**- Check out this. pt = palloc\_get\_page (PAL\_ASSERT | PAL\_ZERO); -> pd[pde\_idx] = pde\_create (pt); and pde\_create retrun vtop (pt) | PTE\_U | PTE\_P | PTE\_W; The question is, where does the palloc\_get\_page address comes from? Is it truly returns a physical address collection?

**PARTIAL Ans Q2:**- Free memory starts at 1 MB and runs to the end of RAM. free\_start = ptov (1024 \* 1024); and free\_end = ptov (init\_ram\_pages \* PGSIZE); p->base = base + bm\_pages \* PGSIZE; **base** is free\_start and **bm\_pages** is the space needed for the bitmap.

**According** to **P**ans Q2, all the pool address is just consecutive integer number.

**2.1** Is there any mechanism that changes this random number to real physical address? Don’t forget that the current running program is in **protected mode**.

**3**- Check out install\_page () function. It adds a mapping from user virtual address UPAGE to kernel virtual address KPAGE to the page table. **UPAGE must not already be mapped!** **Q.N** What if UPAGE, (PHYS\_BASE – PGSIZE) OR file linear address, is already mapped?

**Ans** (1/2): - UPAGE represents linear address for one file, and in pintos one thread corresponds to on program not two. So the same linear address can be mapped with different thread page directory, problem solved.

**Q.N** From the above question we take UPAGE by decrementing page size from PHYS\_BASE, but according to user pool definition, it takes the left memory after the kernel takes his side, I misunderstood this one, may be check how user poll is organized? If **setup\_stack**() function always takes the same UPAGE, what will be our choice if we need to run multiple processes at the same time?

**Future Solution for Pintos**

**@** Kernel pool memory shortage can be solved by giving user pool address whenever request comes. But still user pool memory will continue to work by enabling swapping future not for kernel memory. And whenever the given user pool memory is free in the hands

INDEXS

* ARM (Advanced RISC Machine) -> RISC (Reduced Instruction Set Computing)
* MIPS (Millions of Instruction Per Second)
* SIMD (Single instruction, Multiple data)
* EABI (Embedded Application Binary Interface)
* RTFM (Read the Friendly Manual)
* LFB (Local Function Begin) LFE (Local Function End) lables
* CFI (Call Frame Information)
* CFA (Canonical Frame Address)
* API (Application Program Interface)
* Real-time Operating System (RTOS):- is an OS intended to serve real-time application process data as it comes in, typically without buffering delays. A key characteristic of RTOS is the level of its consistency the amount of time it takes to accept and complete an application’s task.
* UUID (universally unique identifier) is an identifier standard used in a software construction). It is a 128-bit value.
* System V Release 4 (SAV4) Application Binary Interface (ABI)?
* PIE on ELF stands for Position independent executable
* TSL (Test and set lock)
* MAR (memory address register):- is a CPU register that either stores the memory address from which data will be fetched to the CPU or the address to which data will be sent and stored. In other words, MAR holds the memory location of data that needs to be accessed. When reading from memory, data addressed by MAR is fed into the MDR (memory data register) and then used by the CPU. When writing to memory, the CPU writes data from MDR to the memory location whose address is stored in MAR.
* MDR || MBR (Memory data register or Memory buffer register) is the register of a computer’s control unit that contains the data to be stored in the computer register (e.g RAM), or the data after a fetch from the computer storage. It acts like a buffer and holds anything that is copied from the memory ready for the processor to use it.
* Program counter (PC) – an incrementing counter that keeps track of the memory address of the instruction that is to be executed next or in other words, holds the address of the instruction to be executed next.
* Instruction register (IR) – a temporary holding ground for the instruction that has just been fetched from memory
* MAC (media access control address), a hardware address that uniquely indentifies each node of a network.

MEANINGS

* Little endian VS Big endian: - In big endian, you store the most significant byte in the smallest address. In little endian, you store the least significant byte in the smallest address.
* Garbage collector (reclaim garbage, memory occupied by objects that are no longer in use by the program.
* Recursion: - an expression such that each term is generated by repeating a particular mathematical operation.
* Assembly language is a low-level programming language for a computer, or other programmable device, in which there is a very strong (generally one-to-one) correspondence between the language and the machine architecture. Each assembly language is specific to particular computer architecture, but requires interpreting or compiling.

Unclear Concept reference

* Original booting concept
* Process switch
* System call (Linux is easy instead of pintos(Yonas))
* Signed VS Unsigned and compliment VS 2’s compliment

GOALS

* Changing system call
* Creating VFS

BONUS CONCEPTS

* How simple .c file will be converted into assembly format and why it should follow the regular steps
* The **call frame** is identified by an address on the stack. *The typical call stack is used for the return address, locals, and parameters (known as a call frame)*. We refer to this address as the Canonical Frame Address or CFA. Typically, the CFA is defined to be the value of the stack pointer at the call site in the previous frame (which may be different from its value on entry to the current frame)

**M**ain() is called from somewhere else (in the **libc** C runtime support code), and, at the time the **call** instruction is executed, %rsp will point to the top of the stack (which is the lowest address the stack grows downwards)

1. The value of %rsp at this point is the “value of the stack pointer at the call site”, i.e. the CFA as defined by the spec.
2. As the call instruction is executed, it will push a 64-bit (8 byte) ‘ this is 64-bit OS case’ return address onto the stack
3. Now we are running the code at main, which executes **subq** $8, %rsp to reserve another 8 bytes of stack for itself.
4. The change of stack pointer is declared in the debugging information using the .cfi\_def\_cfa\_offset directive, and you can see that the CFA is now at an offset of X bytes (16 Byte in our case) from the current stack pointer.
5. At the end of the function, the **addq** $8, %rsp instruction changes the stack pointer again, so another .cfi\_def\_cfa\_offset directive is inserted to indicate that the CFA is now at an offset of only 8 bytes from the stack pointer.
6. Those numbers which is used inside labels is just an arbitrary value (E.g 22). Commonly the labels are numbered sequentially (regarding **function scope**), here we may be seeing only those (in our case 22) because the optimizer removed the other labels.
7. What is **floating-point register**?
8. How caller and callee use same register without interference? By saving registers value on stack. On which stack exactly? If they use the same stack, what will be the result if other function is involved (like f1 calls f2 and f2 calls f3 and f3 calls f1. Can f1 continue from where it left off)?

* Static VS Dynamic memory allocation
* Static memory is reserved from the moment a program starts until the program exits. It can‘t change its size. Its use and position relative to other application components is typically determined when the source code for the application is compiled.
* Dynamic memory is memory that is requested and managed while the program is running. **It looks like heap behavior.**
* **#**define offfsetof (TYPE, MEMBER) ((size\_t) &((TYPE \*) 0)->MEMBER)

0 // Everything starts with number 0

((TYPE \*)0) //**low**, we are making the compiler believe that there is such a struct at the begging of the program data segment (which is generally dangerous, but not in this case since we don’t actually **access it**)

( ((TYPE \*)0)->MEMBER ) // Then member with this struct will be referenced

&( ((TYPE \*)0)->MEMBER ) //Its contents are not used, but its address taken (with &)

( (size\_t) &( ((TYPE \*)0)->MEMBER )// this address is converted to numeric type (**offset of the member**)

* **#d**efine list\_entry (LIST\_ELEM, STRUCT, MEMBER) ((STRUCT \*) ((uint8\_t \*) &(LIST\_ELEM)->next – offsetof (STRCUT, MEMBER.next))) …………….??????????
* **Bitwise** compliment operator VS 2’s Complement

**B**itwise compliment changes the corresponding bit of the operand to opposite bit,i.e, 0 to 1 and 1 to 0. It is denoted by ~. ~35 will bring 220 if it’s represented by ‘unsigned char’ data type and it will give -36 results for those ‘signed’ data type representation. How?

For any signed integer n, bitwise compliment of n will be – (n+1), which is **2’s** compliment.

* **Accessing the S**tack

Coordinate stack operands with pointer concept to fetch how it works easily.

Take int \*P = &Y;

P == movl %eax, (contents of eax as a source)

\*P == movl %(eax), (fetches source operand from eax address)

P == movl $i, (address denoted by I as source)

\*P == movl i, (source operand from memory at address i)

P == movl $5, (5 as source operand)

* Signed number representation: - setting the most significant bit to 0 is for a positive number or positive zero, and setting it to 1 is for negative number or negative zero.

TEMPORARY PROBLEMS

* Inside thread.c at line 337: - list\_entry (e, struct thread, **allelem**); but I can’t find allelem variable in related files. Similarly, at the same file thread.c at line 587: - thread\_stack\_ofs = offsetof (struct thread, **stack**); but still I can’t find stack variable in related locations. **R.Q.N**: - is it possible? (if you can’t find it in related files, you should find it inside linker script (Yonas))

WHAT IS THE OBJECTIVE OF UNDERSTANDING PROCESS

* It helps me to understand OS better.
* >> >> >> >> >> the effects of interrupt and TSS
* >> >> >> >> >> stack management
* It helps me to manipulate current OS and Process creation as well as context switch of process