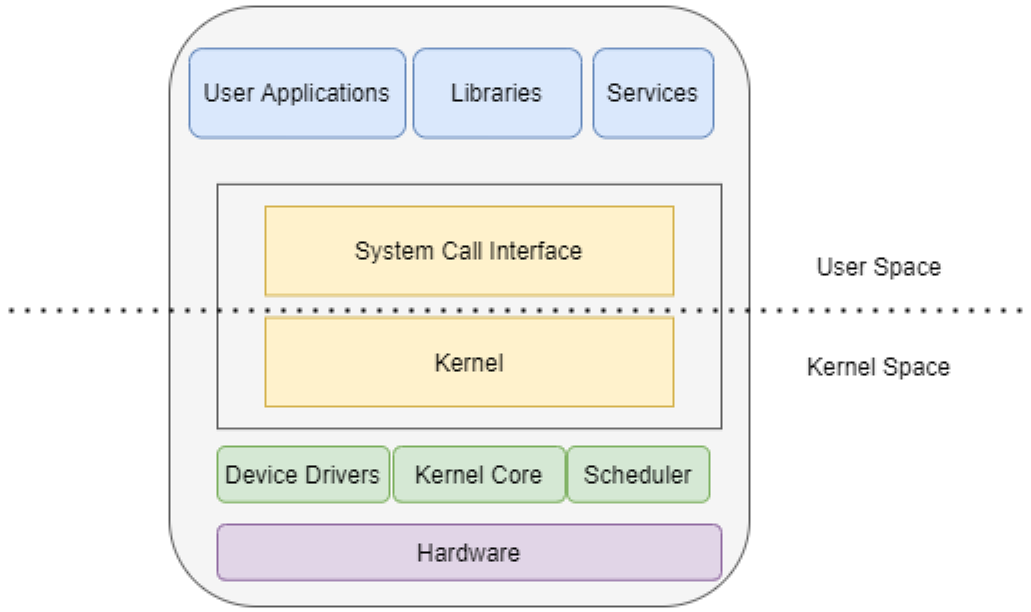
Module 1
Linux Operating System Architecture

Linux OS Architecture

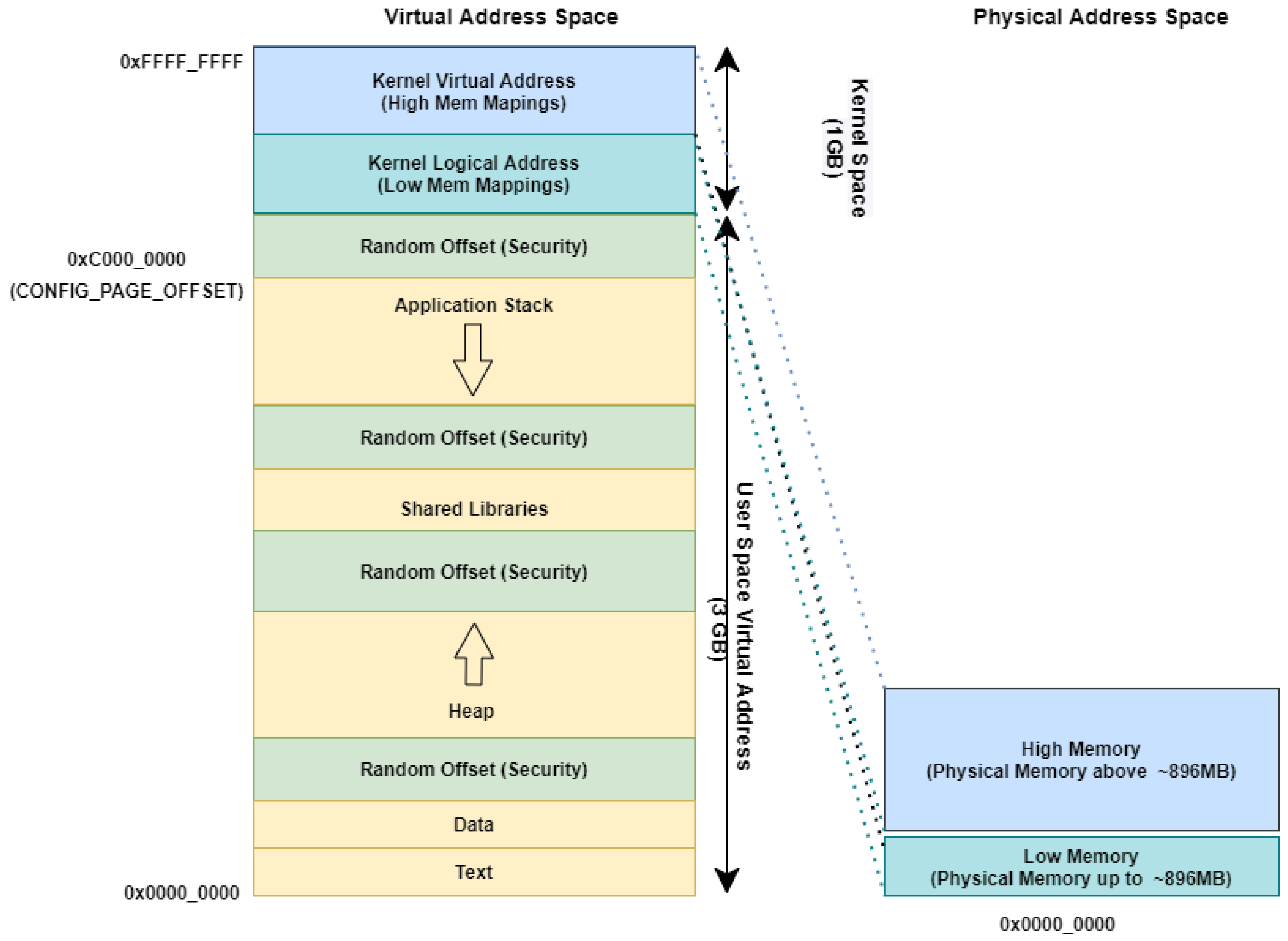
- Linux OS architecture is divided into two main parts.
  - User space
  - Kernel space
- User space is a protected virtual address space that hosts user application programs, system libraries and user level services.
- Kernel space is a privileged mode reserved for kernel functionalities like reading and writing to the hardware, interrupt (IRQ) handling, managing memory and other low level services.



User Mode/Kernel Mode

- The difference between user mode and kernel mode comes from the privileges available in each mode of execution.
- In kernel mode, the kernel is allowed to run all privileged operation such as interrupt handling, I/O, scheduling, process management.
- In user mode, an application is only allowed to perform a set of basic operations.
- A user application can request services from the kernel through system calls; these system calls allow the user space program to request services and certain privileged operations.

32-bit Linux Address Space

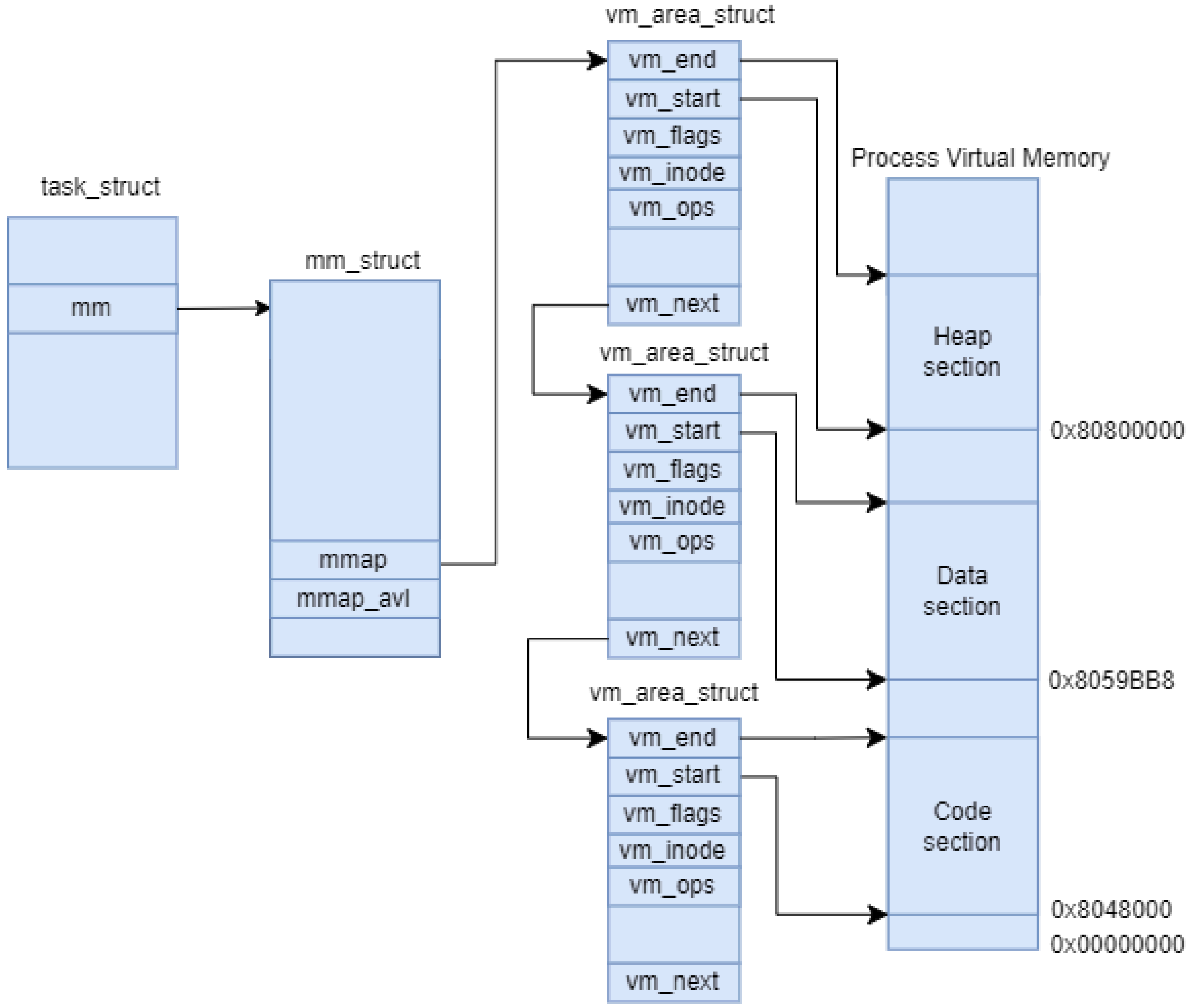


32-bit User Space Addressing

- In a 32 bit system each process has the access to the 3 GB virtual address space.
    - task_struct is a process descriptor and represents a process or a thread in the Linux kernel.The memory layout of a process, including its page tables and memory mapings are defined in mm_struct.
- By default all user mappings are randomized to minimize the possibility of attack (Base of heap, stack, text, data etc.)
- Due to randomization multiple processes could have different address spaces.
- The kernel 'norandmaps' command line option can be used to disable randomization.
    - This is the equivalent to:

```
echo 0 > /proc/sys/kernel/randomize_va_space
```

Virtual address mapping



Virtual Memory Areas

- VMAs are the actual memory zones in a process which are setup by Kernel upon initialization of the process.
    - task_struct->mm-> chain of vm_area_struct
- These zones are tagged by specific attribute (R/W/X).
- A segmentation fault can happen when a program tries to access non-existing VMA or existing VMA in a different way as defined by its attribute.
    - Execute data in non-executable segment
    - Write data in read only segment
- Per application maps is located in `/proc/{PID}/maps` .

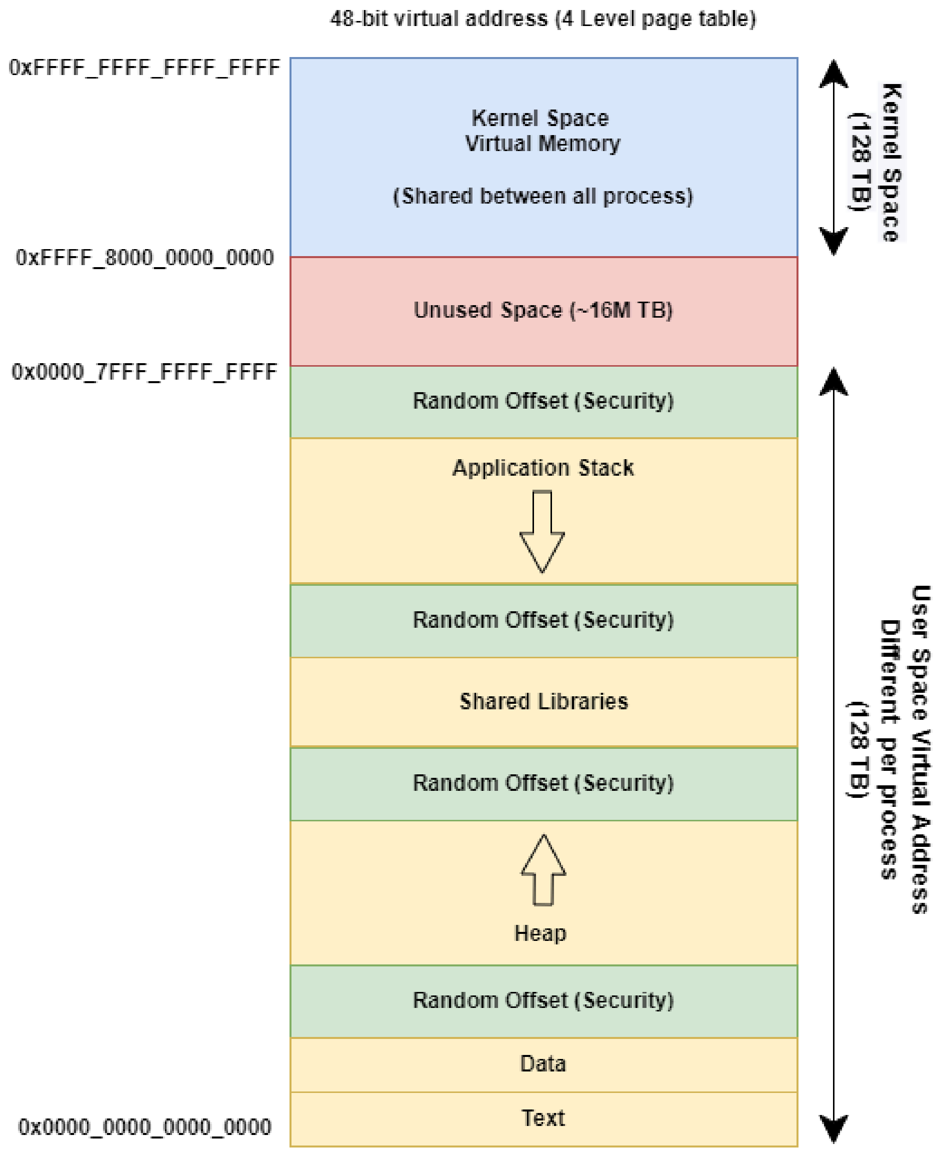
Kernel Logical Addressing

- Kernel Logical Addressing (KLA) - also called Low Mem - is directly mapped to kernel space.
- CONFIG_PAGE_OFFSET defines the offset for KLA. The logical address is calculated at a fixed offset from the physical address.
    - Logical address 0xC0000000 (Page offset) => 0x00000000 Physical address
    - Conversion: The macros __pa(x) and __va(x) can be used to translate between physical and virtual addresses in each direction.
- The Kernel Logical Address range is in physically contiguous memory and suitable for allocating kernel per process stack space, memory for DMA and memory requested through kmalloc.

Kernel Virtual Address

- Kernel Virtual Addresses (also called High Mem because it maps to memory beyond the 896 MB boundary), resides at the top of the Kernel Logical address space. High memory is a region of physical memory that is not directly mapped into the lower portion of a system's physical address space. Instead, it's accessed through various mechanisms, such as paging or memory-mapped I/O.
- These virtual address are suitable for large buffer allocation in the kernel, for example:
    - Memory mapped I/O (SOC IP Block)
    - Insmod
    - Ioremap, kmap
    - vmalloc
- Kernel Virtual Addresses are not physically contiguous, so these are not suitable for some operations like DMA.

64-bit Linux Address Space



64-bit User Space Address

- In a 64 bit system each process has access to 128 TB of virtual address space.
- The basic layout of memory sections such as heap, stack, text, data, shared libraries are same as in 32-bit system.
- The primary difference in user space layout between 32-bit and 64-bit addressing spaces is the significantly larger addressable range in 64-bit architectures.
- This larger space provides more flexibility for memory allocation, mapping, and shared libraries, and it allows for more efficient memory management and utilization in modern systems.
- For x86 64 bit systems the kernel/user space split is at: 0x800000000000000. For ARM64 it is at 0xFFFF880000000000.

64-bit User Space Address Example

The following dump represents the 64-bit user space virtual address map of memory example.

```
manas@sandbox:~/work$ cat /proc/324/maps

**<Address Start>-<AddressEnd>**
        |          |     **<mode>**
        |          |       |    **<offset>**
        |          |       |       |  **<Major ID:Minor ID>**
        |          |       |       |      |  **<inode id>**               **<file path>**
5645ad827000-5645ad828000 r--p 00000000 08:20 438673                     /home/manas/work/memory --> **Read Only, Private Segment (Contains Globals, Constants etc)**
5645ad828000-5645ad829000 r-xp 00001000 08:20 438673                     /home/manas/work/memory --> **Executable Code segment**
5645ad829000-5645ad82a000 r--p 00002000 08:20 438673                     /home/manas/work/memory
5645ad82a000-5645ad82b000 r--p 00002000 08:20 438673                     /home/manas/work/memory
5645ad82b000-5645ad82c000 rw-p 00003000 08:20 438673                     /home/manas/work/memory
5645af7e8000-5645af809000 rw-p 00000000 00:00 0                          [heap]                 --> **Heap section of process**
7f9014ef0000-7f9014ef3000 rw-p 00000000 00:00 0
7f9014ef3000-7f9014f1b000 r--p 00000000 08:20 6488                       /usr/lib/x86_64-linux-gnu/libc.so.6 -> **libc [Shared Lib] read only section**
7f9014f1b000-7f90150b0000 r-xp 00028000 08:20 6488                       /usr/lib/x86_64-linux-gnu/libc.so.6 -> **Libc executable section**
7f90150b0000-7f9015108000 r--p 001bd000 08:20 6488                       /usr/lib/x86_64-linux-gnu/libc.so.6
7f9015108000-7f901510c000 r--p 00214000 08:20 6488                       /usr/lib/x86_64-linux-gnu/libc.so.6
7f901510c000-7f901510e000 rw-p 00218000 08:20 6488                       /usr/lib/x86_64-linux-gnu/libc.so.6
7f901510e000-7f901511b000 rw-p 00000000 00:00 0
7f9015121000-7f9015123000 rw-p 00000000 00:00 0
7f9015123000-7f9015125000 r--p 00000000 08:20 6292                       /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f9015125000-7f901514f000 r-xp 00002000 08:20 6292                       /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f901514f000-7f901515a000 r--p 0002c000 08:20 6292                       /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f901515b000-7f901515d000 r--p 00037000 08:20 6292                       /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f901515d000-7f901515f000 rw-p 00039000 08:20 6292                       /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7ffde39a8000-7ffde39c9000 rw-p 00000000 00:00 0                          [stack]                  --> **Stack section of the process**
7ffde39d2000-7ffde39d6000 r--p 00000000 00:00 0                          [vvar]                   --> "Virtual VDSO Variable"
7ffde39d6000-7ffde39d8000 r-xp 00000000 00:00 0                          [vdso]                   --> "virtual Dynamic Shared Object".
                                                                          This is a small shared library which is exported by the kernel and mapped into user space.
```

64-bit Kernel Space Address

```
                    Kernel-space virtual memory, shared between all processes
                          48-bit virtual address (4 Level page table)
____________________________________________________________________________________________________________________
                  |             |                  |         |
 ffff800000000000 | -128    TB | ffff87ffffffffff |    8 TB | ... guard hole, also reserved for hypervisor
 ffff880000000000 | -120    TB | ffff887fffffffff |  0.5 TB | LDT remap for PTI
 ffff888000000000 | -119.5  TB | ffffc87fffffffff |   64 TB | direct mapping of all physical memory (page_offset_base)
 ffffc88000000000 |  -55.5  TB | ffffc8ffffffffff |  0.5 TB | ... unused hole
 ffffc90000000000 |  -55    TB | ffffe8ffffffffff |   32 TB | vmalloc/ioremap space (vmalloc_base)
 ffffe90000000000 |  -23    TB | ffffe9ffffffffff |    1 TB | ... unused hole
 ffffea0000000000 |  -22    TB | ffffeaffffffffff |    1 TB | virtual memory map (vmemmap_base)
 ffffeb0000000000 |  -21    TB | ffffebffffffffff |    1 TB | ... unused hole
 ffffec0000000000 |  -20    TB | fffffbffffffffff |   16 TB | KASAN shadow memory
                  |             |                  |         |
 fffffc0000000000 |   -4    TB | fffffdffffffffff |    2 TB | ... unused hole
                  |             |                  |         | vaddr_end for KASLR
 fffffe0000000000 |   -2    TB | fffffe7fffffffff |  0.5 TB | cpu_entry_area mapping
 fffffe8000000000 |   -1.5  TB | fffffeffffffffff |  0.5 TB | ... unused hole
 ffffff0000000000 |   -1    TB | ffffff7fffffffff |  0.5 TB | %esp fixup stacks
 ffffff8000000000 | -512    GB | ffffffeeffffffff |  444 GB | ... unused hole
 ffffffef00000000 |  -68    GB | fffffffeffffffff |   64 GB | EFI region mapping space
 ffffffff00000000 |   -4    GB | ffffffff7fffffff |    2 GB | ... unused hole
 ffffffff80000000 |   -2    GB | ffffffff9fffffff |  512 MB | kernel text mapping, mapped to physical address 0
 ffffffff80000000 |-2048    MB |                  |         |
 ffffffffa0000000 |-1536    MB | fffffffffeffffff | 1520 MB | module mapping space
 ffffffffff000000 |  -16    MB |                  |         |
    FIXADDR_START | ~-11    MB | ffffffffff5fffff | ~0.5 MB | kernel-internal fixmap range, variable size and offset
 ffffffffff600000 |  -10    MB | ffffffffff600fff |    4 kB | legacy vsyscall ABI
 ffffffffffe00000 |   -2    MB | ffffffffffffffff |    2 MB | ... unused hole
__________________|_____________|__________________|_________|______________________________________________________

* https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt
```

64-bit Kernel Space Address

- **Direct Mapping of All Physical Memory (page_offset_base)**:
  A region in the kernel space where the entire physical memory is directly mapped for efficient access, allowing kernel code to directly access physical addresses.

- **vmalloc/ioremap Space (vmalloc_base)**:
  This region is used for dynamically allocated kernel data structures and for mapping memory-mapped I/O (MMIO) regions from device drivers.

- **Virtual Memory Map (vmemmap_base)**:
  A mapping of the physical memory's page frames into kernel virtual space, used to access and manage physical memory. It allows the kernel to reference physical memory addresses as kernel virtual addresses.

- **cpu_entry_area Mapping**:
  This area holds per-CPU data structures and is mapped for each CPU, providing a separate area for certain CPU-specific operations and data.

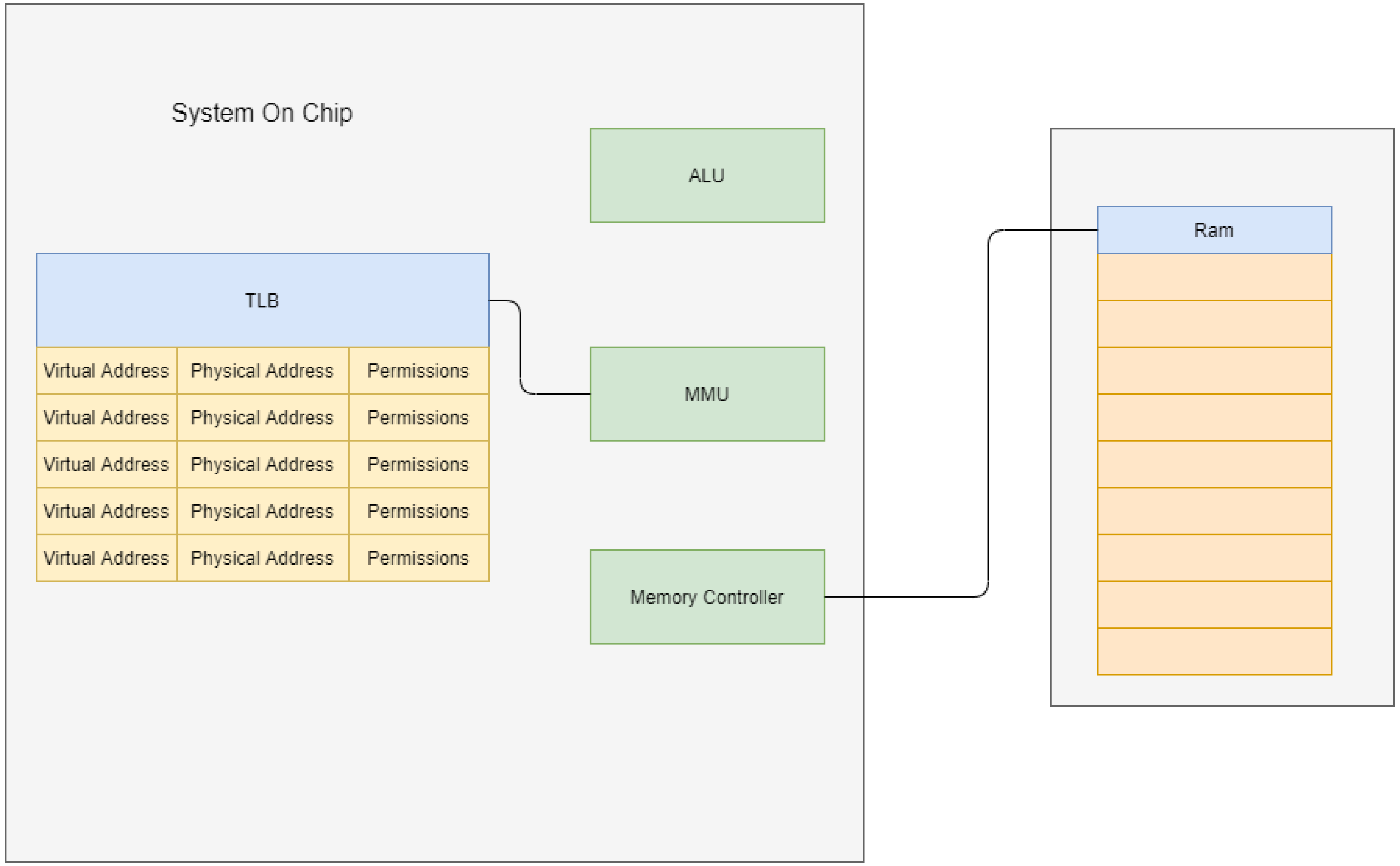
64-bit Kernel Space Address

- **%esp Fixup Stacks**:
  Stacks used to handle exceptions during context switches and when the kernel starts execution. These stacks are used to ensure proper handling of CPU state during such operations.
- **Kernel Text Mapping, Mapped to Physical Address 0**:
  The virtual memory mapping of the kernel's text section (code), which is mapped to the physical address 0 to enable direct access to kernel instructions.
- **Module Mapping Space**:
  A region reserved for dynamically loaded kernel modules, allowing the kernel to load and manage modules separately from its main code.
- **Kernel-Internal Fixmap Range, Variable Size and Offset**:
  A region with variable size and offset used for mapping kernel-internal data structures and hardware register addresses, ensuring efficient access.

Summary of Virtual Addressing in Linux

- **User Virtual Addresses**: These are addresses used by user-level processes. They are an abstraction that allows each process to have its own dedicated virtual memory space which is mapped to physical memory by an MMU.
- **Kernel Logical Addresses**: Kernel logical addresses are a mapping from kernel-space virtual addresses to physical address (Low Mem on 32 bit systems). They use a constant offset that provides a linear, one-to-one mapping to a physical addresses. This memory, allocated with functions like kmalloc, is contiguous and cannot be swapped out. Because of this these allocations are suitable for operations like DMA.
- **Kernel Virtual Addresses**: Kernal virtual addresses map to the High Mem part of physical memory (memory beyond 896MB) on 32 bit systems. These are not physically contiguous but they are virtually contiguous. Memory is allocated with vmalloc and this area is suitable for insmod.
- **Note:** When dealing with addresses in Linux we are **always** dealing with virtual addresses.

Memory Management/ MMU (part 1)



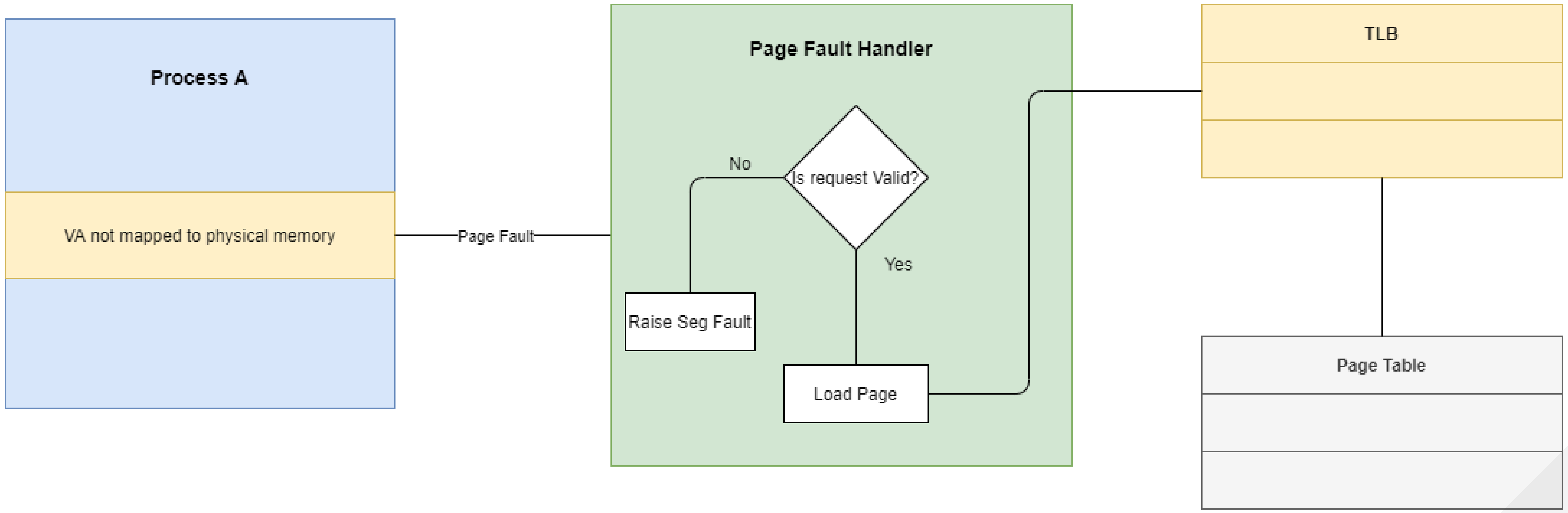
TLB		
Virtual Address	Physical Address	Permissions
Virtual Address	Physical Address	Permissions
Virtual Address	Physical Address	Permissions
Virtual Address	Physical Address	Permissions
Virtual Address	Physical Address	Permissions

- An MMU facilitates the translation from virtual to physical addresses with the help of a Translation Lookaside Buffer (TLB).

Memory Management/ MMU (part 2)

- A user space process runs in a virtual address space.
- The MMU facilitates the translation from virtual to physical addresses.
- The Base unit of MMU is a ‘page’ which is fixed in size and that size depends on the underlying architecture.
- The MMU hardware uses mappings in a page table (PT) for address translation.
- In a multilevel page table, it is more efficient to have first and second level PTs in memory while other levels can reside on the disk.

Page Fault Handling (part 1)



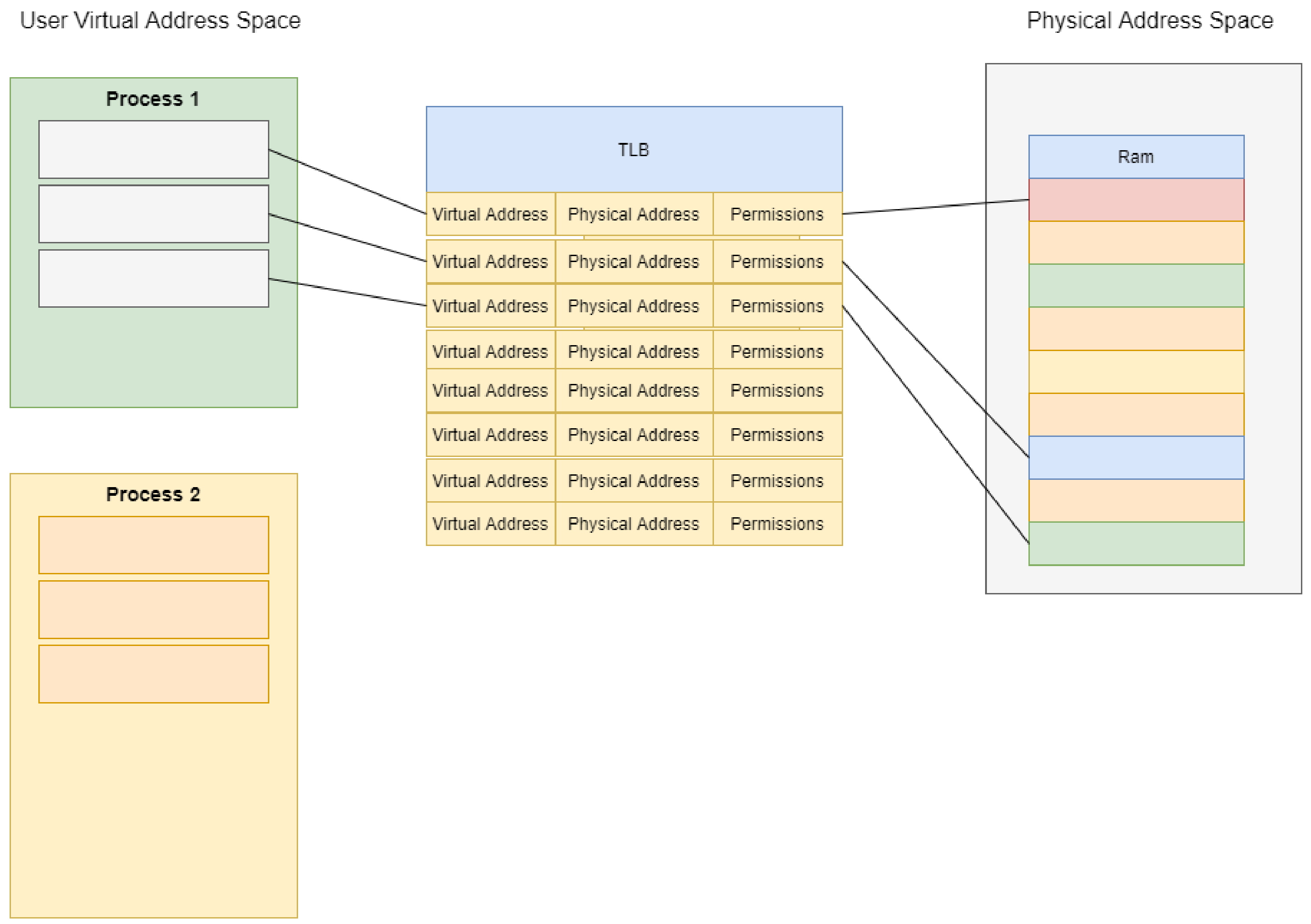
Page Fault Handling (part 2)

- If a user application tries to access a virtual memory address which is not mapped to a physical address, the MMU triggers a page fault.
- The Kernel on receiving the page fault interrupt performs following operations.
    - Puts the user space process to sleep.
    - Finds the mapping for offending address in the PT.
    - Selects and removes existing TLB entry and copies the frame from disk to RAM.
    - Creates a TLB entry for the page containing the address.
    - Finally it wakes up the user space process.

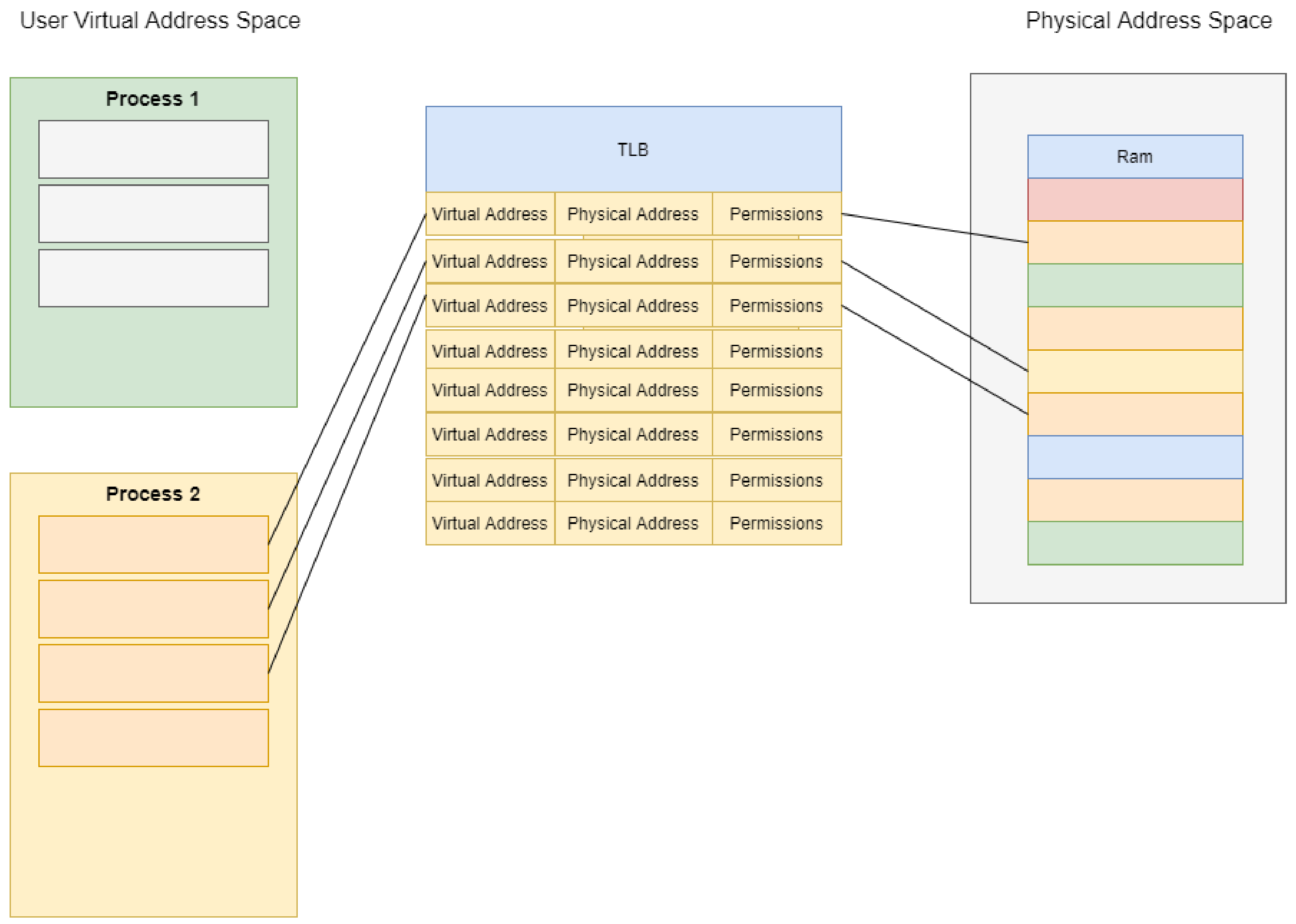
User Processes Mappings (part 1)

- Each process in user virtual address space has its own mapping, and this mapping is changed during a context switch.
- The memory map for a user process will have many mappings.
- Each mapping can cover multiple page frames in physical memory.
- The same virtual address can be mapped to different physical addresses for different processes.
- Because the TLB is a limited resource, far more mappings can be made that exist in the TLB at any one time, so the kernel must keep track of all mappings and it stores this is page tables in struct_mm and vm_area_struct.
- A mapping to a virtually contiguous space does not have to be physically contiguous for user space processes and this makes allocation easier.

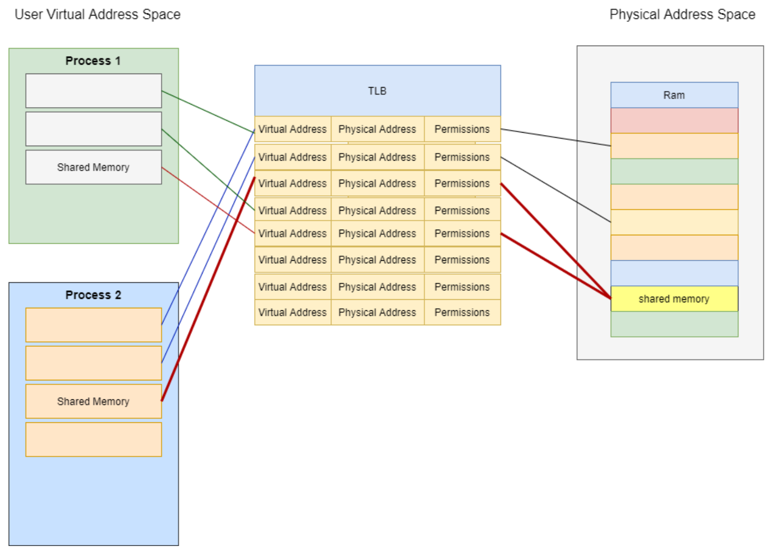
User Processes Mappings (part 2)



User Processes Mappings (part 3)



Shared memory in user space

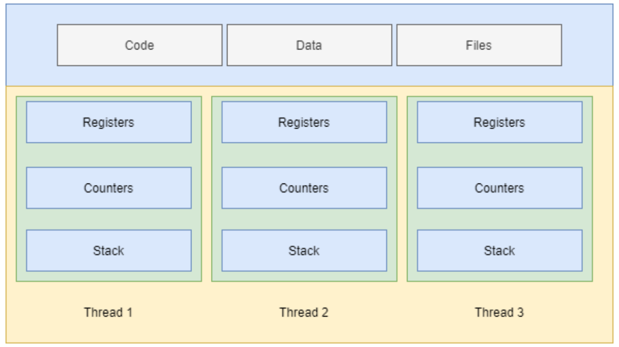


- Shared memory in user space refers to a technique used in operating systems to allow multiple processes to access a region of memory that is shared between them.
- MMU maps the same physical frame into two or more processes.
- The shared memory will have different virtual addresses in each process, but they will be mapped to the same physical memory location.
- mmap() allows us to request specific virtual address to map the shared region.
- Refer to the Shared Memory code example.

Processes and Threads (part 1)

- A process is an instance of a running program that has its own memory space and system resources, such as file descriptors, network sockets, and environment variables.
- Each process is assigned a unique process ID (PID) and information related to a process can be accessed from `/proc/{PID}` file.
- The Kernel keep tracks of all the user space processes through a task vector which is an array of task_struct pointer.
    - The 'current' pointer points to currently running process.
    - This can be accessed through get_current() function.
- Refer to the Process Observer Example.

Processes and Threads (part 2)

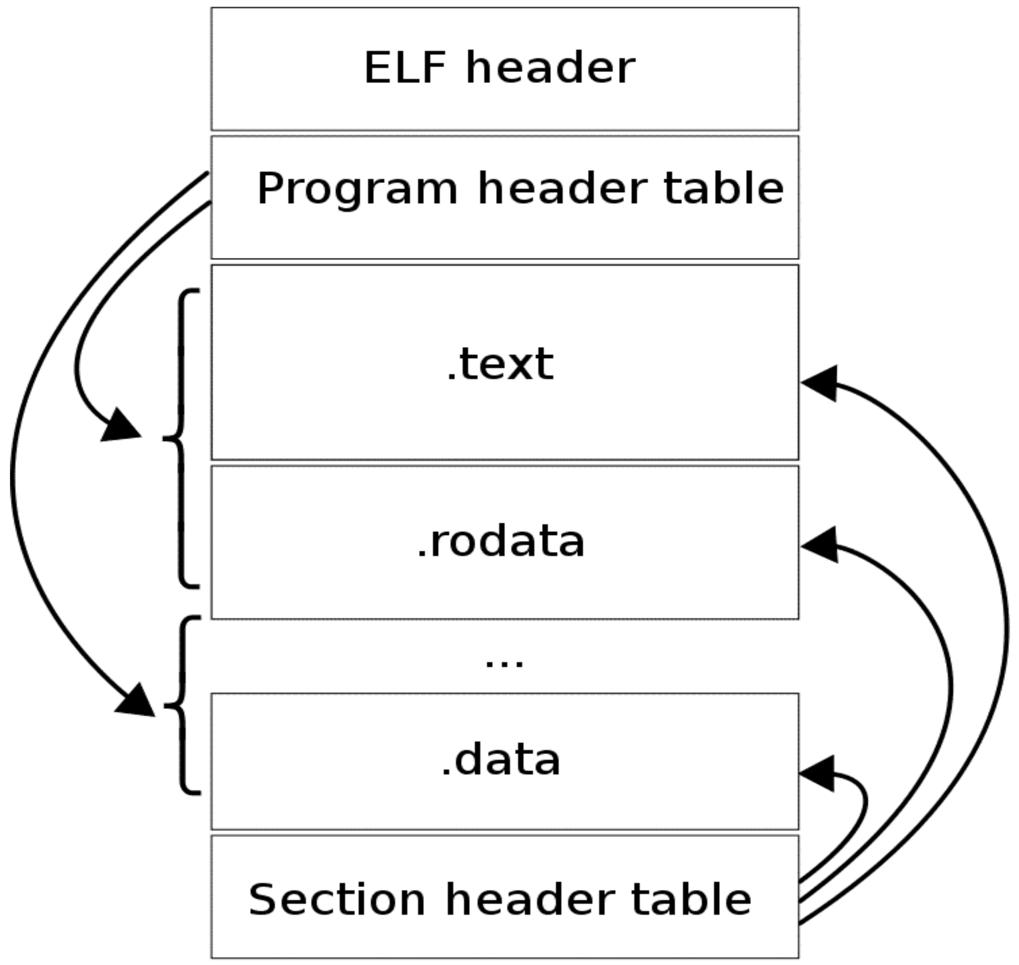


- A thread is a lightweight execution unit within a process.
- Threads share the same memory space and system resources as their parent process, but they have their own registers, stack and program counter.
- Threads can run concurrently and independently within the same process, allowing for parallel execution of code.
- A process is created using the fork() system call whereas a thread is created using the pthread_create() function. Both of these use the clone() system call internally.
- Information related to threads in a process can be found in `/proc/{PID}/task` .
- Refer to the Process and Threads tutorial for an example.

ELF - Executable Format

- ELF is a standard binary format, supported on various platforms including Linux.
- The same format is used by application binaries, shared libraries, Kernel modules, Kernel (vmlinux) and core dump files.
- The ELF file contains an ELF header along with program related segments
    - .text section: Code
    - .data section: Data
    - .rodata section: Read-only Data
    - .debug_info section: debug information
- The kernel uses the run time information from the program header to create the process and map the segments into memory.

ELF Format


ELF header
Program header table
.text
.rodata
...
.data
Section header table

Shared Libraries (part 1)

- A Shared library (.so) file is a collections of pre-compiled code that can be shared among multiple programs at runtime. It binds with the application at runtime.
- The kernel loads the application binary into user space virtual program space as defined in ELF file and parses the .interp section to find the dynamic linker.
    - ld-linux.so is the dynamic linker/loader for the Linux operating system. It is responsible for resolving dependencies between shared libraries and executable files at runtime.

Shared Libraries (part 2)

Shared Libraries are searched in the following path:

- rpath (built into the binary)
- LD_LIBRARY_PATH (an environment variable)
- runpath
- Directories listed in the file /etc/ld.so.conf
- Default system libs (/lib, /usr/lib)

Shared Library Example

- Shared Library Example
- Executable using the shared library
- For more detailed information on Shared Libraries and how they are linked to an executable please refer to this article.

Scheduling (part 1)

- The scheduler is a core component of the Linux Kernel responsible for allocating CPU time to processes.
- It is responsible for deciding which process or thread gets to run on a CPU core at any given time.
- It does this by assigning a priority to each process based on various factors such as the amount of time it has already used, the amount of memory it is using, and its scheduling policy.

Scheduling (part 2)

- The scheduler in the Linux Kernel can be called by various events that require a change in the execution context.
    - Time quantum expiration: The process time slice expires.
    - Process blocking: A process becomes blocked waiting for an event, such as I/O completion, a signal, or a lock acquisition. This causes the scheduler to select another process to run.
    - Process termination: A process completes or is terminated.
    - Interrupts: A software or hardware interrupt occurs which causes the scheduler to run and possibly select a different process to run.
    - Fork and exec: When a process is forked or a new program is executed the scheduler may be called to select a new process based on priority.

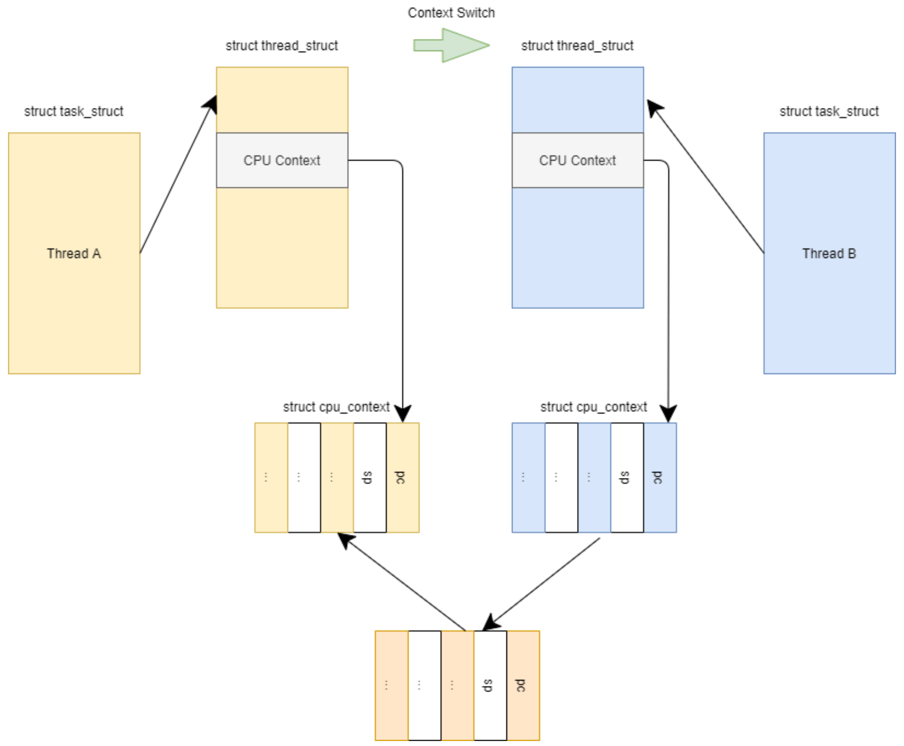
Scheduling (part 3)

- A process can also voluntarily yield the CPU by calling a scheduling function like sched_yield().
- A program can set its own scheduling policy with the sched_setscheduler() call.
- We can query and set CPU scheduling paramaters with tools like schedtool.
- We can set the CPU affinity of a process with taskset.
- Refer to the Scheduling tutorial.

Context Switching (part 1)

- Context switching is a mechanism used by the operating system to switch between different execution contexts in order to allow multiple processes or threads to run concurrently on a single CPU.
- The context of a process includes its CPU registers, program counter, stack pointer, and other relevant information.
- The scheduler selects next process from runqueue and updates its scheduling information.
- The current execution context of running process is saved on kernel stack (located in KLA).

Context Switching (part 2)



- The scheduler restores execution context of selected process from task_struct data structure.
- The selected process's scheduling information is updated, such as priority and virtual runtime.
- The control is returned to restored process, allowing it to continue execution from where it left off.

Scheduling Algorithms (part 1)

- The scheduler uses various scheduling policies and algorithms to make scheduling decisions based on the state of the system.
- Completely Fair Scheduler (CFS): Uses a red-black tree to maintain a sorted list of processes, assigns virtual runtime to each process to calculate priority.
- Real-time scheduling: Provides strict timing guarantees, includes FIFO and Round Robin scheduling

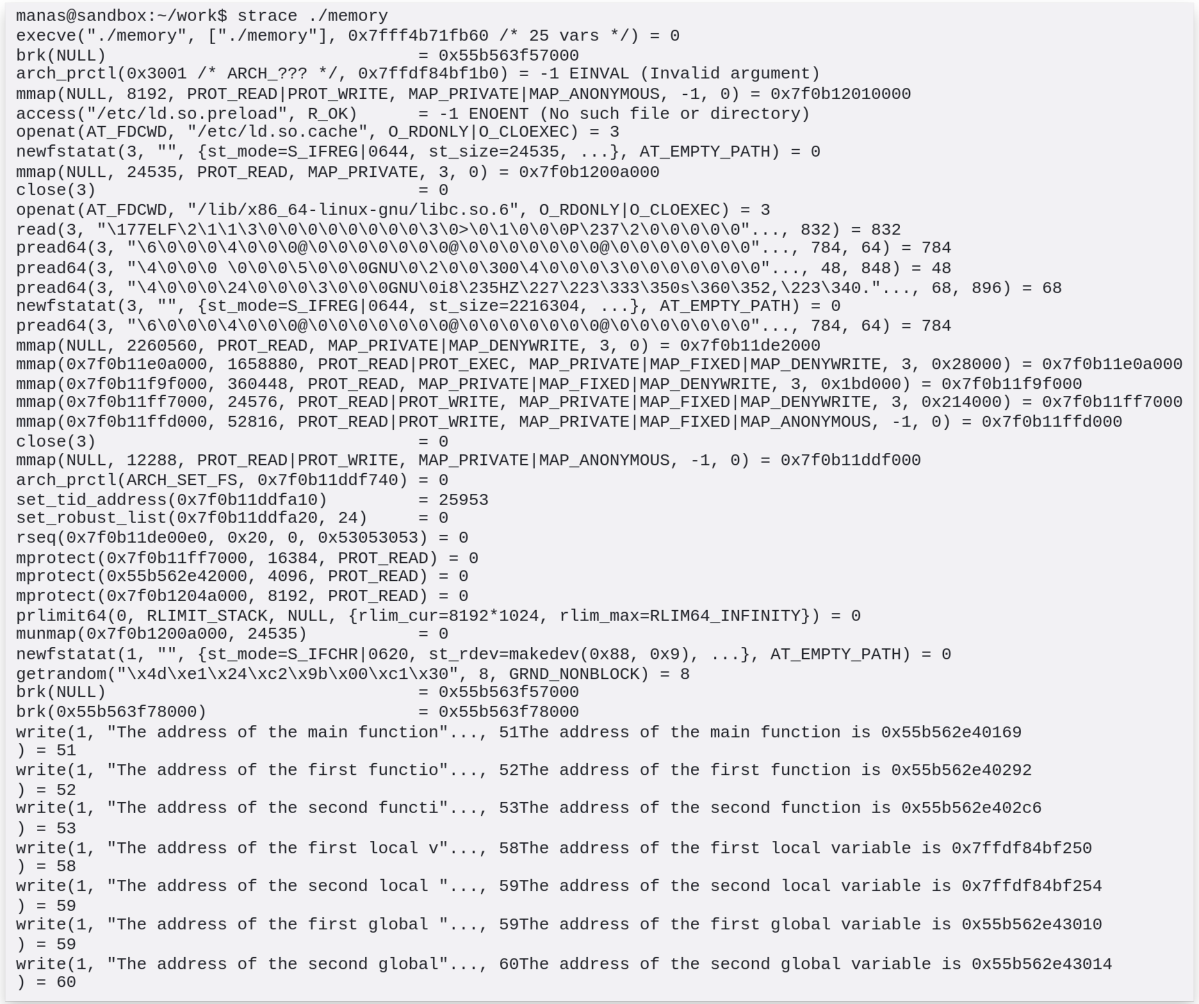
Scheduling Algorithms (part 2)

- Deadline scheduling: Provides soft real-time guarantees by ensuring processes meet their deadline
- Multi-Level Feedback Queue (MLFQ) scheduling: Divides runqueue into multiple priority queues, promotes or demotes processes between queues based on CPU usage
- Round Robin (RR): This scheduler assigns a time quantum to each process, and each process is scheduled to run for its allotted time quantum. Once the time quantum is exhausted, the process is preempted and moved to the back of the run queue.

System Calls

- A system call is an interface which allows user space to request kernel services.
    - read, write, lseek are some of the common system calls.
- On X86 systems a special instruction (0x80) is executed from the system call to generate an exception and switch the execution mode from the user mode to the kernel mode.
- On Arm, executing a SVC (Supervisor Call) instruction generates an Supervisor Call exception, to fullfil a similar function.
- System calls are vectored and identified by their numbers ($\_NR$).
    - Example: __NR__read, defined as 63 in unistd.h.

System call trace of memory example.



```
manas@sandbox:~/work$ strace ./memory
execve("./memory", ["./memory"], 0x7fff4b71fb60 /* 25 vars */) = 0
brk(NULL)                               = 0x55b563f57000
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffdf84bf1b0) = -1 EINVAL (Invalid argument)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f0b12010000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=24535, ...}, AT_EMPTY_PATH) = 0
mmap(NULL, 24535, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f0b1200a000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\237\2\0\0\0\0\0"..., 832) = 832
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
pread64(3, "\4\0\0\0 \0\0\0\5\0\0\0GNU\0\2\0\0\300\4\0\0\0\3\0\0\0\0\0\0\0"..., 48, 848) = 48
pread64(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0i8\235HZ\227\223\333\350s\360\352,\223\340."..., 68, 896) = 68
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=2216304, ...}, AT_EMPTY_PATH) = 0
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
mmap(NULL, 2260560, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f0b11de2000
mmap(0x7f0b11e0a000, 1658880, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x28000) = 0x7f0b11e0a000
mmap(0x7f0b11f9f000, 360448, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1bd000) = 0x7f0b11f9f000
mmap(0x7f0b11ff7000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x214000) = 0x7f0b11ff7000
mmap(0x7f0b11ffd000, 52816, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f0b11ffd000
close(3)                                = 0
mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f0b11ddf000
arch_prctl(ARCH_SET_FS, 0x7f0b11ddf740) = 0
set_tid_address(0x7f0b11ddfa10)         = 25953
set_robust_list(0x7f0b11ddfa20, 24)     = 0
rseq(0x7f0b11de00e0, 0x20, 0, 0x53053053) = 0
mprotect(0x7f0b11ff7000, 16384, PROT_READ) = 0
mprotect(0x55b562e42000, 4096, PROT_READ) = 0
mprotect(0x7f0b1204a000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) = 0
munmap(0x7f0b1200a000, 24535)           = 0
newfstatat(1, "", {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0x9), ...}, AT_EMPTY_PATH) = 0
getrandom("\x4d\xe1\x24\xc2\x9b\x00\xc1\x30", 8, GRND_NONBLOCK) = 8
brk(NULL)                               = 0x55b563f57000
brk(0x55b563f78000)                     = 0x55b563f78000
write(1, "The address of the main function"..., 51The address of the main function is 0x55b562e40169
) = 51
write(1, "The address of the first functio"..., 52The address of the first function is 0x55b562e40292
) = 52
write(1, "The address of the second functi"..., 53The address of the second function is 0x55b562e402c6
) = 53
write(1, "The address of the first local v"..., 58The address of the first local variable is 0x7ffdf84bf250
) = 58
write(1, "The address of the second local "..., 59The address of the second local variable is 0x7ffdf84bf254
) = 59
write(1, "The address of the first global "..., 59The address of the first global variable is 0x55b562e43010
) = 59
write(1, "The address of the second global"..., 60The address of the second global variable is 0x55b562e43014
) = 60
```

Exception

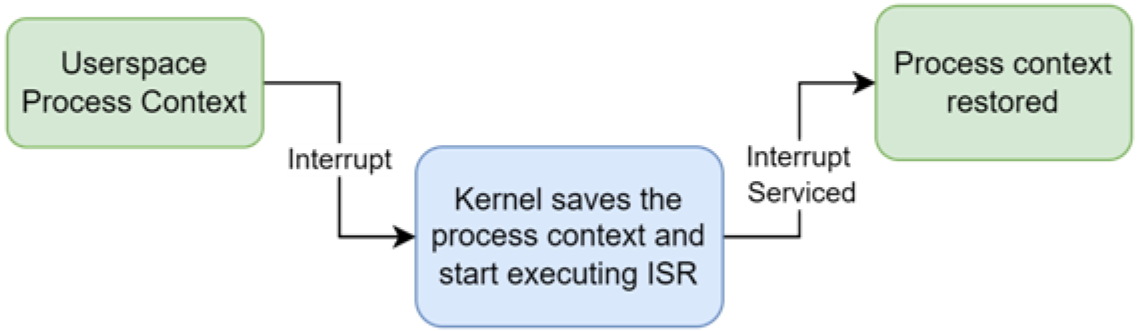
- Exceptions are events that occur within the processor itself that require the attention of the operating system or kernel.
- Examples of software exceptions include invalid memory access, divide-by-zero errors, syscall execution,and other errors that occur as a direct result of the currently executing instruction.
- When an exception occurs, the processor stops executing the current task and transfers control to the kernel, which can then handle the exception appropriately.

Interrupts

- Interrupts are external events that can happen anytime, like data received from a network card or a key pressed on the keyboard.
- Interrupts are asynchronous and can happen anytime, irrespective of what the processor is doing.
- When an interrupt happens, the processor stops the current task and hands over control to the kernel to execute the corresponding ISR.
- The kernel handles interrupts by saving the current task's state, services the interrupt, and then returns to the interrupted task.
- Linux provides information on interrupts through `/proc/interrupts` interface.

Interrupt Context

- The interrupt handler runs in "interrupt context", which is a restricted execution context.
- In interrupt context, certain operations like blocking or sleeping are not allowed because these operations could lead to deadlocks or other issues.
- Interrupt handlers usually run quickly and perform only the essential operations necessary to service the interrupt.
- Nested IRQs are not supported.


Userspace Process Context
Interrupt
Kernel saves the process context and start executing ISR
Interrupt Serviced
Process context restored

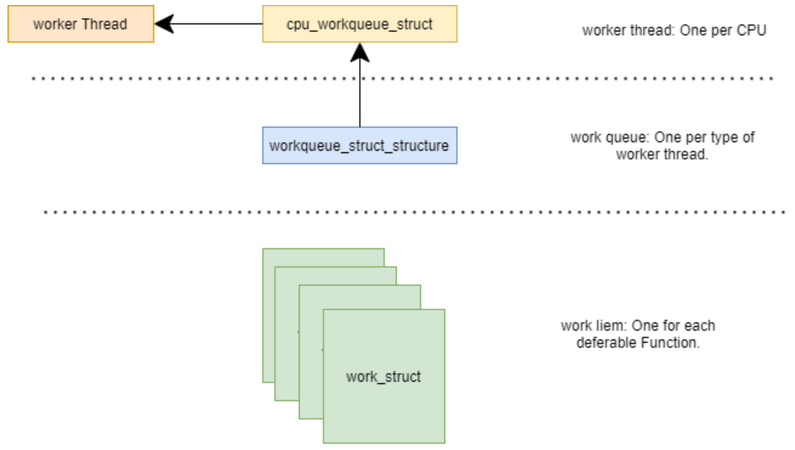
Deferred Interrupts

- Deferred interrupt handling is a technique used to handle interrupts that cannot be serviced immediately by the interrupt handler.
- It allows the processor to return to executing the interrupted task as quickly as possible while still ensuring that the interrupt is eventually serviced.
- The mechanisms provided in the kernel to handle deferred interrupt handling include:
    - Work Queues
    - Softirqs
    - Tasklets

Kernel Thread

- Kernel threads in the Linux kernel are lightweight processes that operate independently of user space processes.
- Kernel threads can be created using kthread_create(), which clones the thread from kthread process.
    - task_struct->mm = NULL
- Kernel threads can perform blocking I/O operations without affecting other processes or threads.
- Examples of kernel threads in the Linux kernel include kworker, ksoftirqd, and kswapd.

Work Queue


worker Thread
cpu_workqueue_struct
worker thread: One per CPU
workqueue_struct_structure
work queue: One per type of worker thread.
work_struct
work liem: One for each deferable Function.

- Work queues in Linux kernel execute non-time-critical tasks asynchronously via a dedicated kernel thread called a worker thread.
    - They run in process context and blocking calls (sleep) are allowed.
    - Interrupts are enabled in workqueues.
- A work item, on the other hand, is a unit of work that is submitted to a work queue.
- Work items are represented by a struct work_struct and contain a callback function pointer.
- Execution of work items is queued and managed by the kernel thread.

Soft IRQs

- SoftIRQ (software interrupt) is a mechanism in the Linux kernel used for handling deferred interrupt processing.
- SoftIRQs run in interrupt context and sleep is not allowed.
- Interrupts are enabled while running softIRQ.
- SoftIRQ are statically defined at compile time and cannot be changed dynamically.
- NR_SOFTIRQS is a constant in the Linux kernel that defines the total number of software interrupts (softirqs) available in the system.
- Soft IRQs are reserved for most time critical task such as networking and block devices.

Tasklets

- Tasklets are a type of SoftIRQ handler used in the Linux kernel for handling **non-time-critical** tasks. They run in a Soft IRQ context.
- Tasklets are implemented on top of SoftIRQs.
- HI_SOFTIRQ: HI_SOFTIRQ is a high-priority softirq in the Linux kernel that runs on every CPU when it is scheduled. It is designed for high-priority work (like tasklets) that needs to be done as soon as possible.
- When a tasklet is scheduled, it is executed in the TASKLET_SOFTIRQ context, which is a type of softirq designed specifically for handling tasklets.
- Tasklets can be dynamically allocated and initialized at runtime using the tasklet_init() function.
- Tasklets of the **same type** are always serialized: in other words, the **same type** of tasklet cannot be executed by two CPUs at the same time. However, tasklets of **different types** can be executed concurrently on several CPUs.

References

- Introduction to memory management in Linux, Matt porter, Konsulko Group
  - Video: https://www.konsulko.com/portfolio-item/introduction-to-memory-management-in-linux-matt-porter-video
  - Slides: https://www.konsulko.com/portfolio-item/intro-to-memory-management
- Processes in Linux: https://tldp.org/LDP/tlk/kernel/processes.html
- Linux memory management: https://tldp.org/LDP/tlk/mm/memory.html
- Debugging Shared Libraries
  - https://medium.com/@johnos3747/shared-libraries-in-c-programming-ab149e80be22
  - https://amir.rachum.com/shared-libraries/