

---

# **VE482 - Homework 6**

Introduction to Operating Systems

Kexuan Huang 518370910126

November 21, 2021



## Ex.1 - Simple Questions

1.

- First Fit
  - (i) 12 KB  $\rightarrow$  20 KB
  - (ii) 10 KB  $\rightarrow$  10 KB
  - (iii) 9 KB  $\rightarrow$  18 KB
- Best Fit:
  - (i) 12 KB  $\rightarrow$  12 KB
  - (ii) 10 KB  $\rightarrow$  10 KB
  - (iii) 9 KB  $\rightarrow$  9 KB
- Quick Fit:
  - (i) 12 KB  $\rightarrow$  12 KB
  - (ii) 10 KB  $\rightarrow$  10 KB
  - (iii) 9 KB  $\rightarrow$  9 KB

2.

$$\text{Time} = 10 + \frac{n}{k} \text{ nsec}$$

3.

- Page 0: 01101110
- Page 1: 01001001
- Page 2: 00110111
- Page 3: 10001011

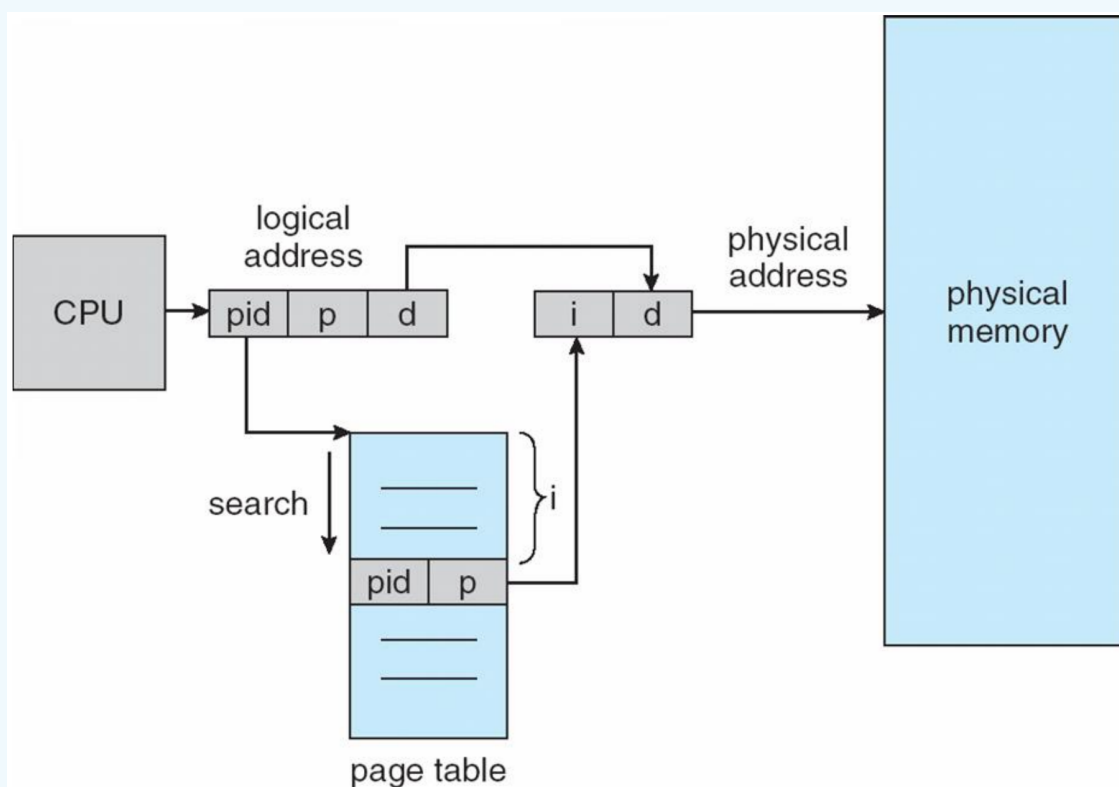
## Ex. 2 - Page tables

### Inverted page tables

Rather than each process having a page table and keeping track of all possible logical pages, the Inverted Page Table track all physical pages. It consists of one-page table entry for every frame of the main memory. So the number of page table entries in the Inverted Page Table reduces to the number of frames in physical memory and a single page table is used to represent the paging information of all the processes.

Through the inverted page table, the overhead of storing an individual page table for every process gets eliminated and only a fixed portion of memory is required to store the paging information of all the processes together. This technique is called as inverted paging as the indexing is done with respect to the frame number instead of the logical page number.

The Inverted Page Table decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs. TLB and hash table can be taken advantaged of to accelerate the access.

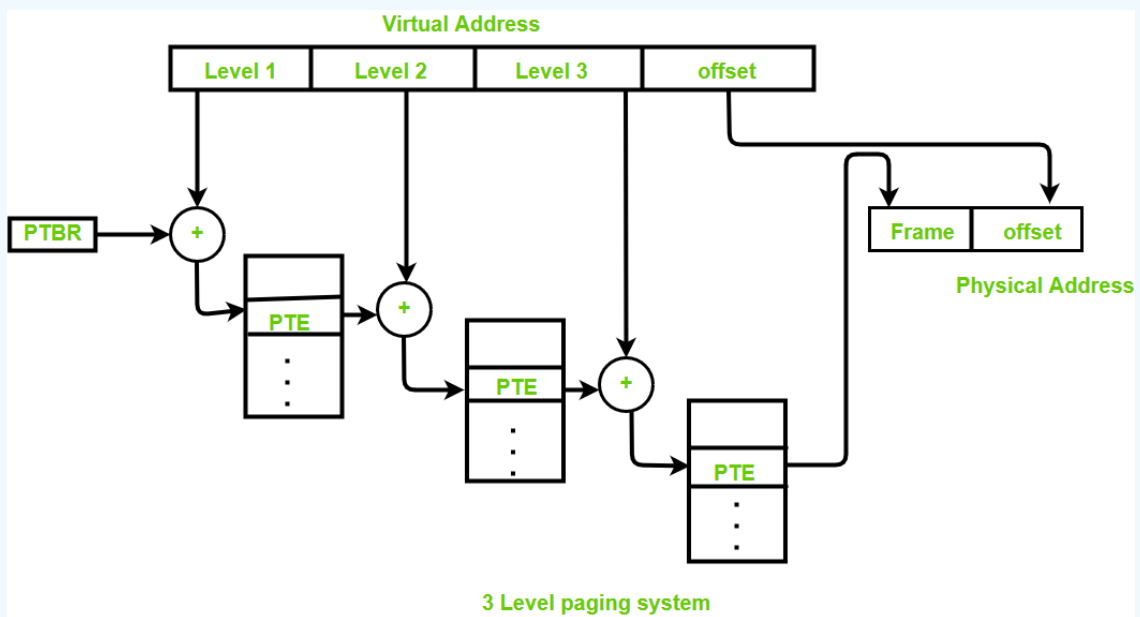


**Figure 1:** Inverted Page Table Architecture

### Multilevel Page Table

Multilevel Paging is a paging scheme which consist of two or more levels of page tables in a hierarchical manner. It is also known as hierarchical paging. The entries of the level 1 page table are pointers to a level 2 page table and entries of the level 2 page tables are pointers to a level 3 page table and so on. The entries of the last level page table are stores actual frame information. Level 1 contain single page table and address of that table is stored in PTBR (Page Table Base Register).

In multilevel paging whatever may be levels of paging all the page tables will be stored in main memory. So it requires more than one memory access to get the physical address of page frame. One access for each level needed. Each page table entry except the last level page table entry contains base address of the next level page table. Extra memory references to access address translation tables can slow programs down by a factor of two or more. Use translation look aside buffer (TLB) to speed up address translation by storing page table entries.



**Figure 2:** Multilevel Page Table

## Research

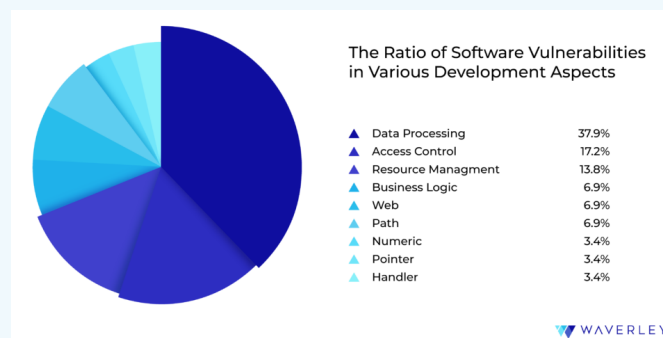
### 1. Security Holes<sup>1 2</sup>

The most common software holes are caused by software flaws - configuration or coding errors. Security bugs generally fall into a fairly small number of broad categories that include:

- Memory safety
- Race condition
- Secure input and output handling
- Faulty use of an API
- Improper use case and exception handling
- Resource leaks, often but not always due to improper exception handling
- Preprocessing input strings before they are checked for being acceptable

Take Buffer Overflow as example, top one (37.9%) coding errors occur in the data processing aspect in C and C++. These failures happen when the memory buffer is set to receive more data than its capacity permits. During program execution, read or write operations can be conducted from a location outside the buffer limits. An unauthorized user can exploit this security vulnerability fairly easily and execute malicious code, read sensitive data, or change the control flow in the program.

```
1 #include <string.h>
2 int main(int argc, char** argv) {
3     char buf[256];
4     strcpy(buf, argv[1]);
5 }
```



**Figure 3:** The Ratio of Software Vulnerabilities in Various Development Aspects

<sup>1</sup>[https://en.wikipedia.org/wiki/Security\\_bug](https://en.wikipedia.org/wiki/Security_bug)

<sup>2</sup><https://waverleysoftware.com/blog/top-software-vulnerabilities/>

## 2. Meltdown<sup>3</sup> and Spectre<sup>4</sup>

Meltdown and Spectre exploit critical vulnerabilities in modern processors. These hardware vulnerabilities allow programs to steal data which is currently processed on the computer. While programs are typically not permitted to read data from other programs, a malicious program can exploit Meltdown and Spectre to get hold of secrets stored in the memory of other running programs. The key difference between Spectre and Meltdown is that due to Spectre you can read or trick other processes to leak memory on the same privilege level, using Meltdown you can read memory you have no privileges to access.

**Meltdown** exploits the way these features interact to bypass the CPU's fundamental privilege controls and access privileged and sensitive data from the operating system and other processes. The simple illustration can be written in following steps:

- The CPU encounters an instruction accessing the value, A, at an address forbidden to the process by the virtual memory system and the privilege check. Because of speculative execution, the instruction is scheduled and dispatched to an execution unit. This execution unit then schedules both the privilege check and the memory access.
- The CPU encounters an instruction accessing address Base+A, with Base chosen by the attacker. This instruction is also scheduled and dispatched to an execution unit.
- The privilege check informs the execution unit that the address of the value, A, involved in the access is forbidden to the process (per the information stored by the virtual memory system), and thus the instruction should fail and subsequent instructions should have no effects. Because these instructions were speculatively executed, however, the data at Base+A may have been cached before the privilege check – and may not have been undone by the execution unit (or any other part of the CPU). If this is indeed the case, the mere act of caching constitutes a leak of information in and of itself. At this point, Meltdown intervenes.
- The process executes a timing attack by executing instructions referencing memory operands directly. To be effective, the operands of these instructions must be at addresses which cover the possible address, Base+A, of the rejected instruction's operand. Because the data at the address referred to by the rejected instruction, Base+A, was cached nevertheless, an instruction referencing the same address directly will execute faster. The process can detect this timing difference and determine the address, Base+A, that was calculated for the rejected instruction – and thus determine the value A at the forbidden memory address.

<sup>3</sup>[https://en.wikipedia.org/wiki/Meltdown\\_\(security\\_vulnerability\)](https://en.wikipedia.org/wiki/Meltdown_(security_vulnerability))

<sup>4</sup>[https://en.wikipedia.org/wiki/Spectre\\_\(security\\_vulnerability\)](https://en.wikipedia.org/wiki/Spectre_(security_vulnerability))

**Spectre** is a vulnerability that tricks a program into accessing arbitrary locations in the program's memory space. An attacker may read the content of accessed memory, and thus potentially obtain sensitive data. The Spectre paper displays the attack in four essential steps:

- First, it shows that branch prediction logic in modern processors can be trained to reliably hit or miss based on the internal workings of a malicious program.
- It then goes on to show that the subsequent difference between cache hits and misses can be reliably timed, so that what should have been a simple non-functional difference can in fact be subverted into a covert channel which extracts information from an unrelated process's inner workings.
- Thirdly, the paper synthesizes the results with return-oriented programming exploits and other principles with a simple example program and a JavaScript snippet run under a sandboxing browser; in both cases, the entire address space of the victim process (i.e. the contents of a running program) is shown to be readable by simply exploiting speculative execution of conditional branches in code generated by a stock compiler or the JavaScript machinery present in an existing browser. The basic idea is to search existing code for places where speculation touches upon otherwise inaccessible data, manipulate the processor into a state where speculative execution has to touch that data, and then time the side effect of the processor being faster, if its by-now-prepared prefetch machinery indeed did load a cache line.
- Finally, the paper concludes by generalizing the attack to any non-functional state of the victim process. It briefly discusses even such highly non-obvious non-functional effects as bus arbitration latency.

Since the fundamental vulnerability is from the hardware, the complete avoidance requires redesign of CPU architecture. But there're patches against Meltdown for Linux (KPTI<sup>a</sup>), Windows, and OS X. For KPTI, it stands for "Kernel page-table isolation", and it works by better isolating user space and kernel space memory. There is also work to harden software against future exploitation of Spectre, respectively to patch software after exploitation through Spectre.

<sup>a</sup>[https://en.wikipedia.org/wiki/Kernel\\_page-table\\_isolation](https://en.wikipedia.org/wiki/Kernel_page-table_isolation)

## Dirty COW<sup>5</sup>

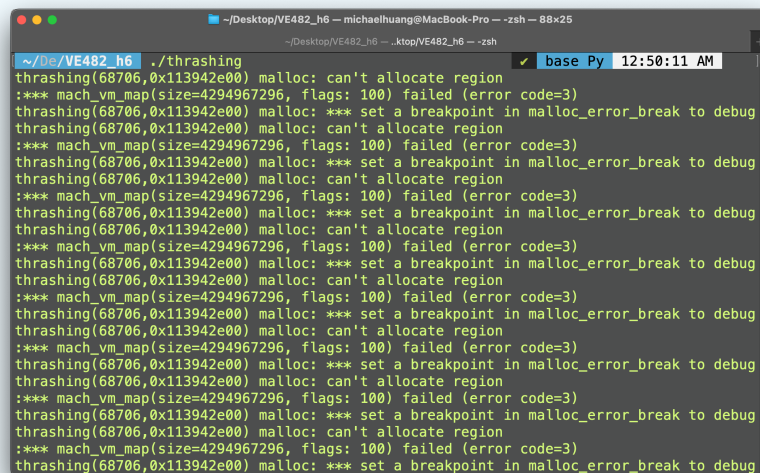
Dirty COW (Dirty copy-on-write) is a computer security vulnerability for the Linux kernel that affected all Linux-based operating systems that used older versions of the Linux kernel created before 2018. It is a local privilege escalation bug that exploits a race condition in the implementation of the copy-on-write mechanism in the kernel's memory-management subsystem. Computers and devices that still use the older kernels remain vulnerable.

The Dirty COW vulnerability has many perceived use cases including proven examples, such as obtaining root permissions in Android devices. When privileges are escalated, someone else can modify unmodifiable binaries and files, so that so that such binaries perform additional, unexpected functions, bringing loss to the computer or users.

## Ex.4 - Linux

Write a very short C program that leads to thrashing.

Please refer to [thrashing.c](#) and [README.md](#).



```
~/Desktop/VE482_h6 — michaelhuang@MacBook-Pro — zsh — 88x25
~/Desktop/VE482_h6 — ktop/VE482_h6 — zsh
~/De/VE482_h6 ./thrashing
thrashing(68706,0x113942e00) malloc: can't allocate region
:*** mach_vm_map(size=4294967296, flags: 100) failed (error code=3)
thrashing(68706,0x113942e00) malloc: *** set a breakpoint in malloc_error_break to debug
thrashing(68706,0x113942e00) malloc: can't allocate region
:*** mach_vm_map(size=4294967296, flags: 100) failed (error code=3)
thrashing(68706,0x113942e00) malloc: *** set a breakpoint in malloc_error_break to debug
thrashing(68706,0x113942e00) malloc: can't allocate region
:*** mach_vm_map(size=4294967296, flags: 100) failed (error code=3)
thrashing(68706,0x113942e00) malloc: *** set a breakpoint in malloc_error_break to debug
thrashing(68706,0x113942e00) malloc: can't allocate region
:*** mach_vm_map(size=4294967296, flags: 100) failed (error code=3)
thrashing(68706,0x113942e00) malloc: *** set a breakpoint in malloc_error_break to debug
thrashing(68706,0x113942e00) malloc: can't allocate region
:*** mach_vm_map(size=4294967296, flags: 100) failed (error code=3)
thrashing(68706,0x113942e00) malloc: *** set a breakpoint in malloc_error_break to debug
thrashing(68706,0x113942e00) malloc: can't allocate region
:*** mach_vm_map(size=4294967296, flags: 100) failed (error code=3)
thrashing(68706,0x113942e00) malloc: *** set a breakpoint in malloc_error_break to debug
thrashing(68706,0x113942e00) malloc: can't allocate region
:*** mach_vm_map(size=4294967296, flags: 100) failed (error code=3)
thrashing(68706,0x113942e00) malloc: *** set a breakpoint in malloc_error_break to debug
thrashing(68706,0x113942e00) malloc: can't allocate region
:*** mach_vm_map(size=4294967296, flags: 100) failed (error code=3)
thrashing(68706,0x113942e00) malloc: *** set a breakpoint in malloc_error_break to debug
```

Figure 4: Thrashing

<sup>5</sup>[https://en.wikipedia.org/wiki/Dirty\\_COW](https://en.wikipedia.org/wiki/Dirty_COW)