# Isolation und Schutz in Betriebssystemen

## 5. Kernel-Isolation bei x86-64

Michael Schöttner

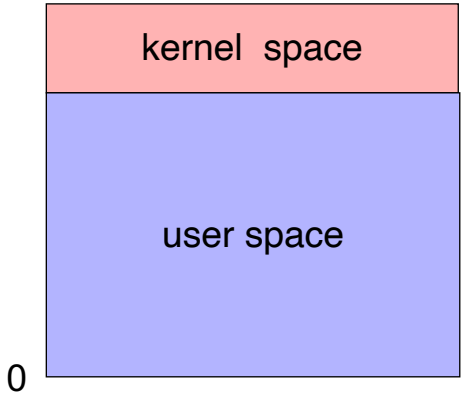
# Adressraumaufteilung vor Meltdown

Heinrich Heine
Universität Düsseldorf

- Für jeden Prozess gibt es einen eigenen Adressraum = user space
- Der Kernel ist in jeden Adressraum eingeblendet = kernel space

- Schutz durch Segmentierung = privilege level

- Schutz durch Paging = Kernel wird durch U/S Bit in Seitentabellen geschützt



# Meltdown



- Entdeckung Juli 2017
- Betrifft Prozessoren von Intel, AMD und ARM
- Im Prinzip unerkannt seit ca. 20 Jahren

- Was passiert hierbei?
  - Umgeht die Isolation zwischen Betriebssystem (kernel mode) und Anwendung (user mode)
  - Erlaubt im User-Mode den Zugriff auf geschützten Kernel-Speicher und den Speicher von anderen Prozessen

- Namensgebung → Hardware- und BS-Sicherheitsgrenzen schmelzen

# Meltdown

Heinrich Heine
Universität Düsseldorf

- Meltdown ist ein sogenannter Seitenkanal-Angriff
  - Der Angreifer kann die Daten nicht direkt lesen / senden.
  - Idee: Verwende unabhängige Ereignisse zur Kommunikation (z.B. Lampe an - Lampe aus)
  - Extraktion von Informationen über einen geheimen schmalbandigen Informationskanal. Ursprünglich im Bereich der eingebetteten Systeme.

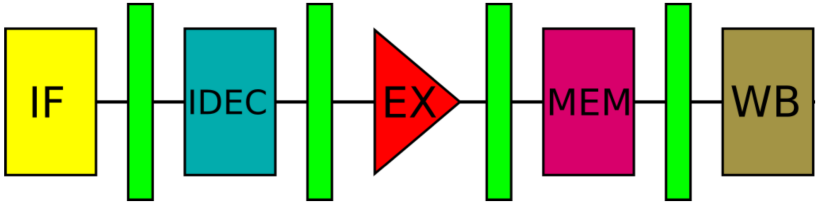
- Exploit basiert auf der Kombination verschiedener Beschleunigungsfunktionen in der Hardware und den Betriebssystemen
  - Parallele und spekulative Ausführung von Instruktionen
  - Caching & TLB
  - Adressräume

# CPU: Pipelining

Heinrich Heine
Universität Düsseldorf

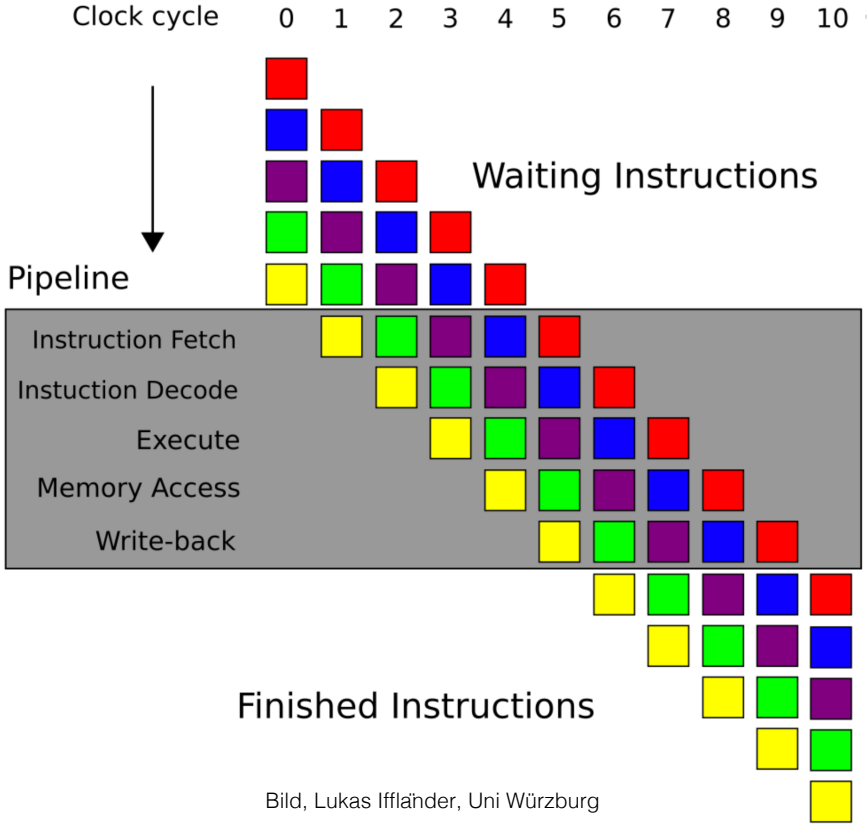
- Problem: Befehle sind komplex und Speicherzugriffe langsam

- Lösung: Pipelining
  - Befehle vorab in μOPs dekodieren und Daten vorab laden
  - Ablauf der Abarbeitung eines Befehls:
  - Instruction Fetch (IF)
  - Instruction Decode (IDEC)
  - Instruction Execute (EX)
  - Memory Access (MEM)
  - Result Writeback (WB)




Bild, Lukas Iffländer, Uni Würzburg

# CPU: Pipelining




Bild, Lukas Iffländer, Uni Würzburg

# CPU: Branch Prediction

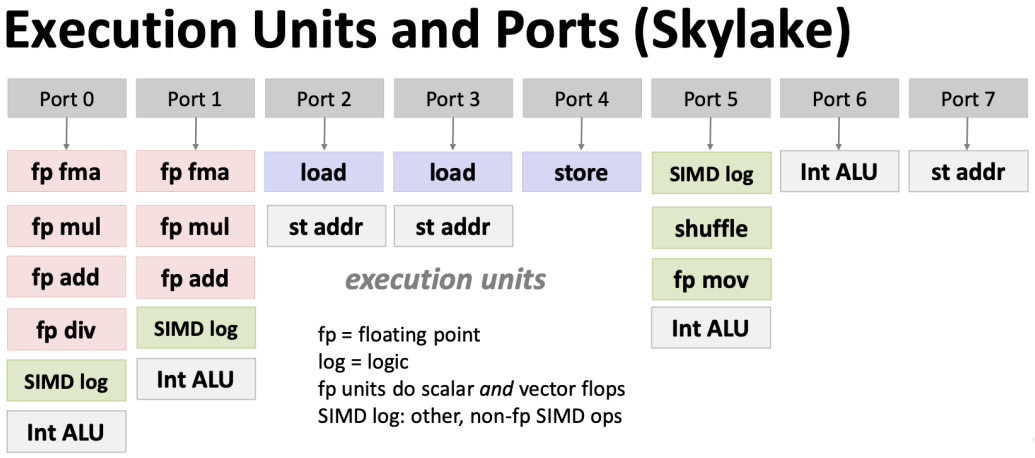
Heinrich Heine
Universität Düsseldorf

- Pipelining → Befehle vorab dekodieren und Daten vorab laden
- Problem: Bei einer Verzweigung im Code ist unklar, wo es weitergeht
- Lösung: Sprungvorhersage (engl. branch prediction) und spekulative Ausführung (engl. speculative execution)
  - Die Änderungen der Ausführung werden erst sichtbar, wenn die Spekulation richtig war, ansonsten werden sie verworfen
  - Funktioniert gut bei Schleifen
    → es ist wahrscheinlicher, dass noch eine Iteration kommt

# CPU: Out-of-Order Execution

Heinrich Heine
Universität Düsseldorf

- Problem: Taktraten können aufgrund physikalischer Grenzen nicht mehr viel erhöht werden. → Daher setzt man auf Parallelität
- Meiste CPUs sind seit 1998 superskalar:
  - Ein superskalarer Prozessor hat mehrere Execution-Units und kann daher in einem Taktzyklus mehrere Befehle ausführen



# CPU: Out-of-Order Execution

Heinrich Heine
Universität Düsseldorf

- Ziel: möglichst alle Execution-Units gleichzeitig nutzen
- Dazu werden die Befehle **eines** Threads aus einem sequentiellen Befehlsstrom automatisch auf mehrere Execution-Units verteilt = Instruction Level Parallelism (ILP)
- x86 CPUs verwenden intern eine RISC-Mikroarchitektur welche die Assemblerbefehle in Mikro-Operationen = µOps zerlegt
- Beispiel:

```
; Example 2.1. Out of order processing
mov eax, [mem1]
imul eax, 5
add eax, [mem2]
```

  - `ADD EAX, [MEM2]` wird in zwei µOps unterteilt
  - CPU holt `[MEM2]` während `imul` ausgeführt wird

# CPU: Out-of-Order Execution

Heinrich Heine
Universität Düsseldorf

- In der Mikroarchitektur werden die Register umbenannt = register renaming
  - Intern gibt es hierfür die Register Alias Table (RAT)
- Beispiel:
  - Hier wird [MEM1] mit 6 multipliziert
  - Und  auf [MEM3] 2 aufaddiert
  - Beides ist voneinander unabhängig und kann durch auto. umbennen der Register out of order ausgeführt werdenterteilt
  - CPU holt [MEM2] während imul ausgeführt wird

```
; Example 2.3. Register renaming
1: mov eax, [mem1]
2: imul eax, 6
3: mov [mem2], eax
4: mov eax, [mem3]
5: add eax, 2
6: mov [mem4], eax
```

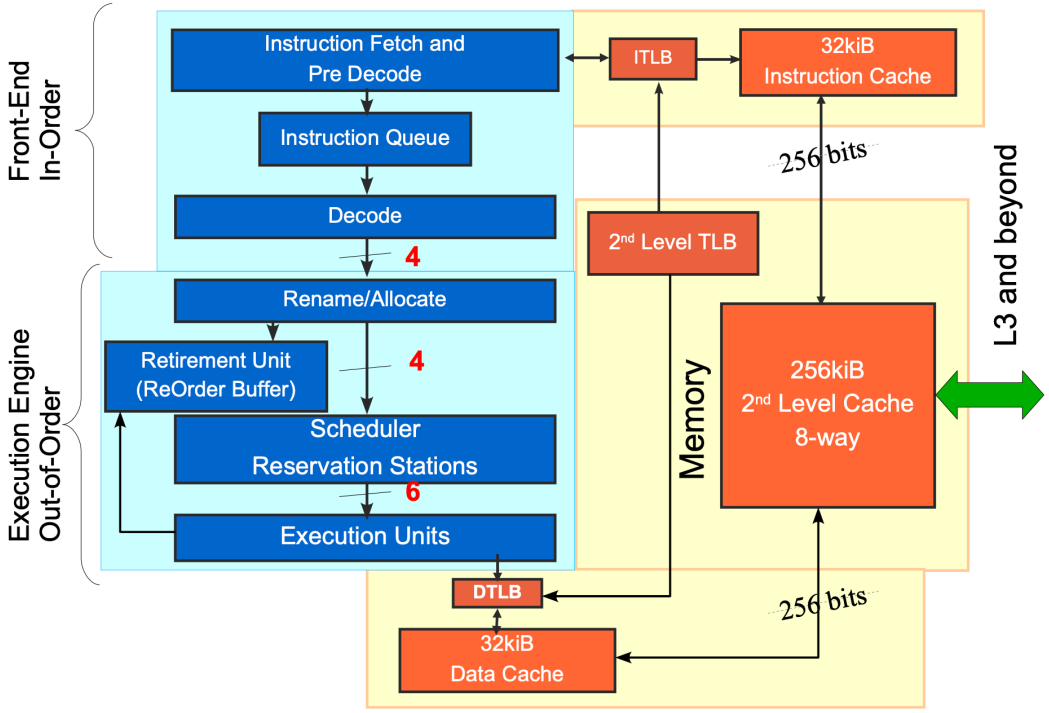
# CPU: Retire in Order

Heinrich Heine
Universität Düsseldorf

- Am Ende müssen die Änderungen der Assembler-Instruktionen in der Reihenfolge des Programms sichtbar werden → **retire in order**

- Dies erledigt der Reorder buffer (ROB):
  - Überwacht richtige Ordnung der Assembler-Instruktionen
  - Und kümmert sich auch um die Abarbeitung von Exceptions (in der richtigen Reihenfolge)

# Beispiel: Nehalem Core Pipeline




High-level diagram of a Nehalem core pipeline, Michael E. Thomadakis, Ph.D., https://progforperf.github.io/nehalem.pdf

# Beispiel: Out-of-Order Execution

Heinrich Heine
Universität Düsseldorf

- Wir verdrängen den kompletten Cache → lesen ein sehr großes Array (nicht `array`!)
- Dann wird folgendes Programm ausgeführt:

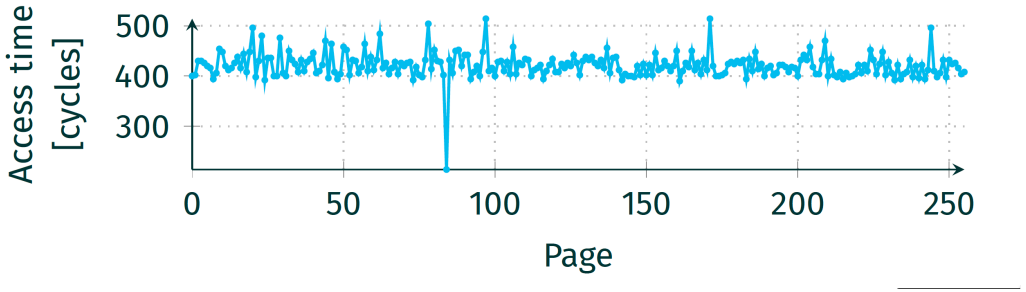
```
1: *(volatile char*) 0;  // raise_exception();
2: array[84 * 4096] = 0; // access 84 page from array start
```

- Die 2. Zeile ist offensichtlich unabhängig von Zeile 1

# Beispiel: Out-of-Order Execution

Heinrich Heine
Universität Düsseldorf

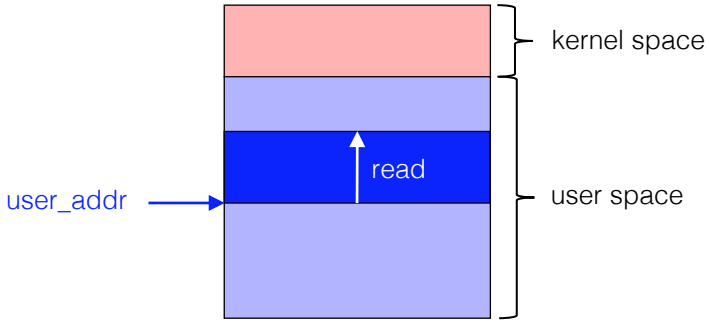
- Nun lesen wir die Elemente von `array` und messen jeweils wie lange ein Lesezugriff dauert




- Ein Zugriff ist viel schneller
  → Instruktion in Zeile 2 wurde ausgeführt
  → Exception wurde erst danach geworfen
  → der Cache wird durch die Exception nicht zurückgesetzt
  → wir können dadurch indirekt die Ausführung einer flüchtigen Instruktion beobachten

# Meltdown Attack (Prinzip)

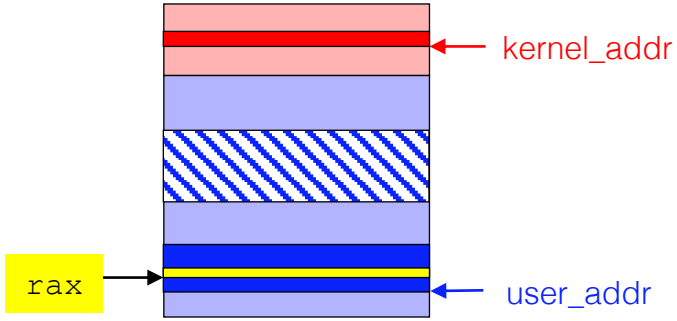
- Schritt 1: Lösche den Cache, indem ein großes Array um User-Space alloziert (im Bild ab `user_addr`)
  und komplett gelesen wird



# Meltdown Attack (Prinzip)

Heinrich Heine
Universität Düsseldorf

- Schritt 2: Führe folgende Instruktionen aus
  - Der Zugriff auf `kernel_addr` ist verboten
  - Der Zugriff auf `user_addr` ist aber erlaubt



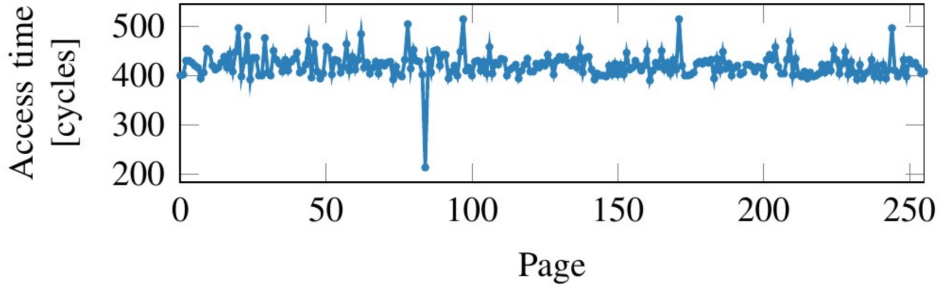

```
; user-mode code

1: mov rax, [kernel_addr]        ; access address in kernel space
2: and rax, 0xff                 ; use lowest byte
3: mov rbx, [rax*128+user_addr] ; as index in an array in user-space
                                 ; align the access with cache lines
                                 ; here 128 bytes per cache line
```

# Meltdown Attack (Prinzip)

Heinrich Heine Universität Düsseldorf

- Schritt 3: Lese ab `user_addr` in einer Schleife jeweils ein Byte im Abstand von 128 Byte (Cache-Line-Größe)
  - Dabei wird ein Zugriff viel schneller als alle anderen sein
  - Das ist genau die Cache-Line, welche geladen wurde, nachdem der unerlaubte Zugriff stattgefunden hat
  - Damit können wir die Nummer der Cache-Line bestimmen und damit den Inhalt des Index, also das Byte in `rax`
  - Wir haben also ein Byte aus dem Kernel-Space gelesen
  - Dies kann man jetzt natürlich wiederholen und so beliebige viel auslesen



# Meltdown Attack (Prinzip)

Heinrich Heine
Universität Düsseldorf

- Schritt 4: verhindern, dass der Prozess terminiert wird, wenn der unerlaubte Speicherzugriff erkannt wird
  - Einfach einen Signalhandler für SIGSEGV registrieren
  - Und damit eine Terminierung verhindern
  - Und darin dann den Index per Cache-Zeitmessung ermitteln (siehe Schritt 3)
  - So kann Byte für Byte gelesen werden

# Meltdown Attack (Prinzip)

Heinrich Heine
Universität Düsseldorf

- Warum funktioniert das?
  - Zeile 2 & 3 hängen ja von Zeile 1 ab

```
1: mov rax, [kernel_addr]
2: and rax, 0xff
3: mov rbx, [rax*128+user_addr]
```

- Spekulative Ausführung
  - Die Wahrscheinlichkeit ist hoch, dass µOPs für die Zeilen 2 und 3 bereits in der Reservierungsstation warten, bis die Daten von Zeile 1 vorliegen
  - Sobald die Daten von [kernel_addr] auf dem internen Datenbus sichtbar sind, werden die wartenden µOPs angestossen.
  - Parallel dazu wird das Ergebnis von Zeile 1 in der Retirement-Unit abgearbeitet, wo dann die Exception ausgelöst wird.
  - Bis das passiert wurde aber der Cache bereits für die µOPs für Zeile 3 befüllt
  - Durch die Exception wird der CPU-Zustand zurückgesetzt, nicht aber der Cache-Inhalt.

# Bemerkungen

Heinrich Heine
Universität Düsseldorf

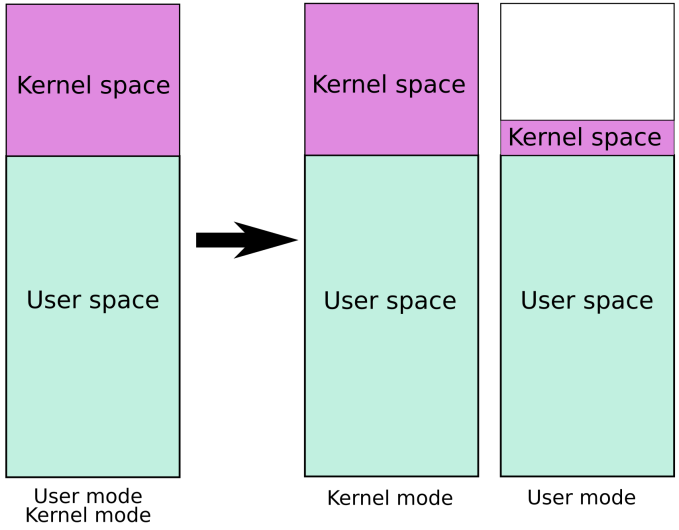
- Die meisten Betriebssysteme blenden kompletten physikalischen Adressraum zusätzlich im Kernel-Space ein
- Damit ist es mit Meltdown möglich den gesamten Speicher auszulesen
- Teilweise wurden in NTFS auch private Schlüssel von Dateisystemen im Kernel-Space gehalten, wodurch diese auch auslesbar sind und damit verschlüsselte Dateisysteme angreifbar sind

- Der SIGSEGV ist teuer, aber auf schnellen Systemen kann mit Meltdown der Kernel-Speicher mit einer Rate von ca. 500 kb/s gelesen werden.
  → insgesamt äußerst kritisches Problem

# Lösung: Kernel page-table isolation (KPTI)

Heinrich Heine Universität Düsseldorf

- KPTI (vormals KAISER) realisiert eine getrennten Adressraum für den Linux Kernel →
  - Wenn die Anwendung aktiv ist, wird nur ein minimaler Teil für den Einsprung ins System eingeblendet
  - Bei einem Systemaufruf wird der Kernel komplett eingeblendet
    - Dies geschieht durch Setzen von Einträgen im Page-Directory
  - Beim Rücksprung aus dem Kernel muss der TLB geflusht werden
    - Damit die Kernel-Seiten nicht mehr zugreifbar sind
    - Das ist teuer, bis 30% langsamer




https://de.wikipedia.org/wiki/Kernel_page-table_isolation

# TLB mit PCIDs

Heinrich Heine
Universität Düsseldorf

- Neuere Prozessoren bieten Process Context Identifiers (PCIDs)
  - 12 Bit → 4096 PCIDs möglich
  - Kann in CR4 aktiviert werden (falls vorhanden)
  - Die aktuelle PCID steht in Bit 11:0 im CR3
- Erlaubt es der MMU im TLB gleichzeitig Einträge von verschiedenen Adressräumen zu speichern und zu unterscheiden
- Beim Zugriff auf der TLB werden nur die Einträge verwendet, deren PCID der aktuellen PCID entspricht
- Deswegen muss beim Rücksprung von einem Systemaufruf nicht mehr der ganze TLB gespült werden!

# Literatur




- Lipp et.al., „Meltdown: Reading Kernel Memory from User Space“, USENIX Security Symposium, USA, 2018.
- Gruss et.al., „KASLR is Dead: Long Live KASLR”, Engineering Secure Software and Systems, Germany, 2017.
- Gruss et.al., „Kernel Isolation - From an Academic Idea to an Efficient Patch for Every Computer“, USENIX, 2018, Vol. 43, No. 4
- Umfassende Infos zu Meltdown und Spectre: https://meltdownattack.com

# Datenaustausch bei Systemaufrufen

Heinrich Heine
Universität Düsseldorf

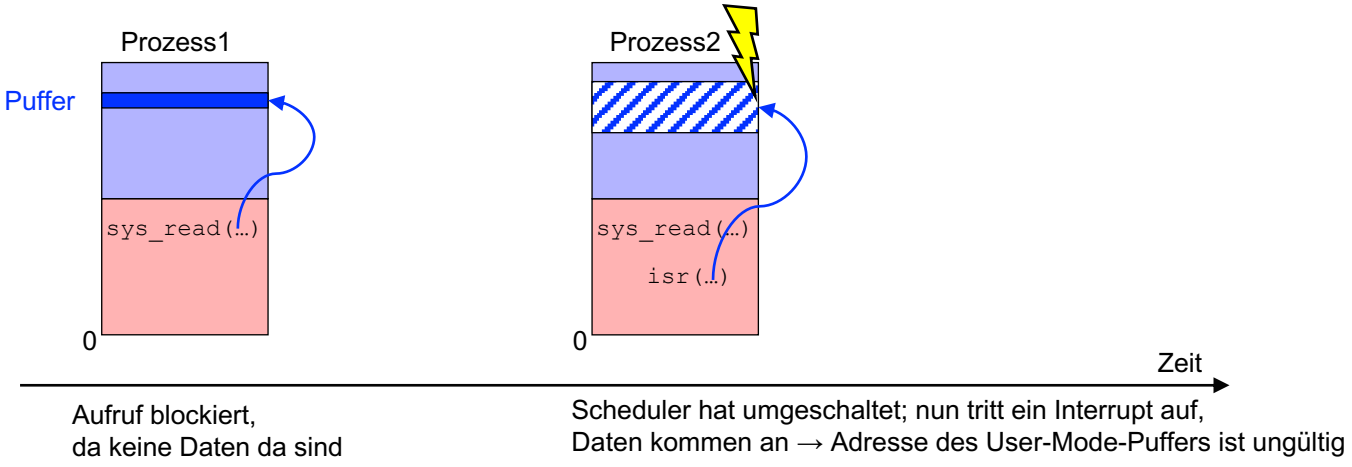
- Bei Systemaufrufen wird oft ein Pointer auf einen Puffer im Heap übergeben
  - Beim Lesen: Eingangspuffer (leer, für neue Daten)
  - Beim Schreiben: Ausgangspuffer (bereits mit Daten befüllt)

- Hierbei muss im Kernel ein gleich großer Puffer angelegt und die Daten müssen umkopiert werden (siehe nächste Seiten)

- Grund: zum Zeitpunkt des Zugriffs auf den User-Mode-Puffer ist evt. ein anderer Prozess aktiv, sodass dann die User-Mode-Adressen ungültig sind

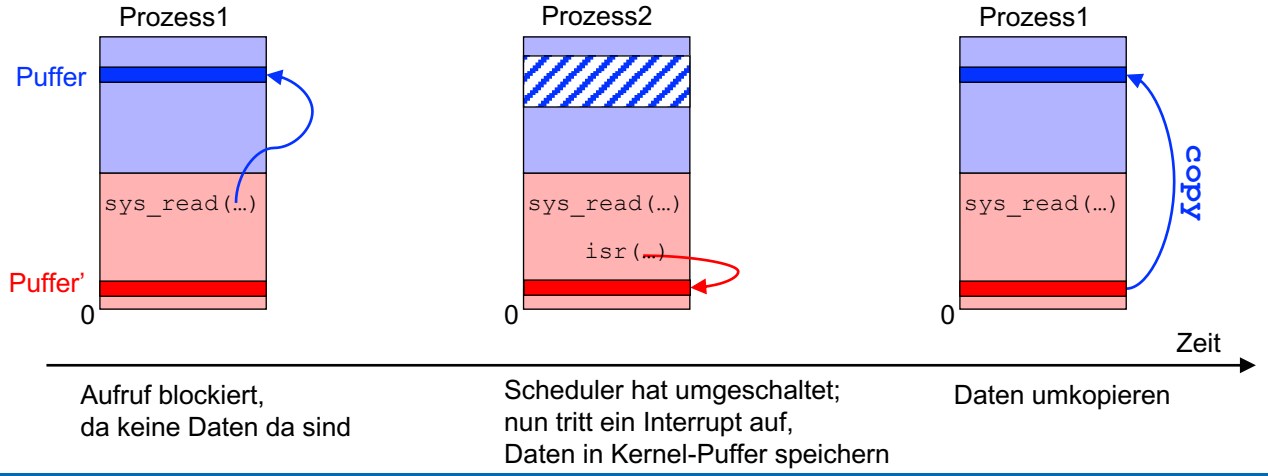
# Systemaufruf: Lesen

- Falls keine Daten vorhanden sind, so wird der Aufrufer blockiert
- Später kommt ein Interrupt im Treiber, wenn neue Daten ankommen und der wartende Thread wird deblockiert
- Zum Zeitpunkt, wenn der Interrupt auftritt, ist evt. ein anderer Prozessadressraum aktiviert, weswegen die User-Mode-Adressen ungültig sind:



# Systemaufruf: Lesen

- Falls keine Daten vorhanden sind, so wird der Aufrufer blockiert
- Später kommt ein Interrupt im Treiber, wenn neue Daten ankommen und der wartende Thread wird deblockiert
- Zum Zeitpunkt, wenn der Interrupt auftritt, ist evt. ein anderer Prozessadressraum aktiviert, weswegen die User-Mode-Adressen ungültig sind:



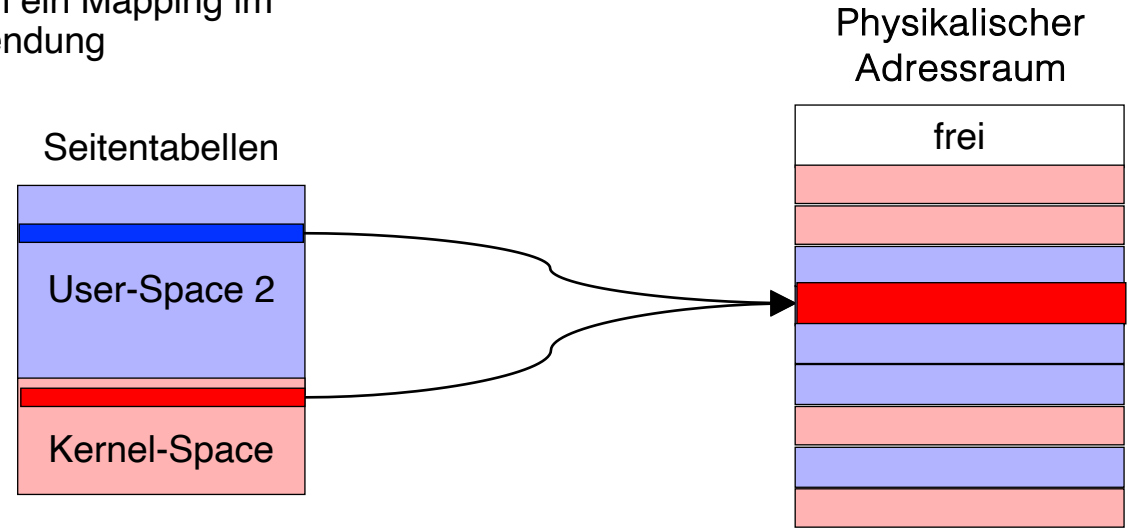
# Systemaufruf: Schreiben

Heinrich Heine
Universität Düsseldorf

- Wie beim Lesen besteht auch hier das Problem, dass die übergebenen Daten evt. nicht direkt an das Gerät geschrieben werden können
  - Beispielsweise ist das Gerät gerade noch beschäftigt
- Daher wird auch hier umkopiert, d.h. im Kernel wird ein Puffer erzeugt und dann werden die Daten aus dem User-Mode-Puffer umkopiert.
- Dadurch ist es egal, ob der Prozess noch aktiv ist, wenn der eigentlich Schreibvorgang stattfindet

# Doppelpufferung vermeiden

- Lösung 1: Zwei Mappings (UNIX/Linux)
  - Treiber realisiert die `mmap`-Funktion
  - Falls eine Anwendung `mmap` im Treiber aufruft, so alloziert der Treiber einen Puffer im Kernel-Space und erzeugt zusätzlich ein Mapping im User-Space der Anwendung



# Doppelpufferung vermeiden

Heinrich Heine
Universität Düsseldorf

- Lösung 2: Kernel verwendet physikalische Adressen (Windows DirectIO)
  - I/O-Manager erzeugt eine Memory Description List:
    - Liste mit Kacheln, die Puffer belegt
  - Treiber arbeitet dann direkt auf den physikalischen Adressen
    - I.d.R. ist der gesamte physikalische Adressraum im Kernel-Space ein 1:1 Mapping
    - Somit ist die Adress-Umrechnung einfach möglich, ohne Seitentabellen:
      - Bei einem lower-half Kernel gilt: physische Adresse = virtuelle Adresse
      - Bei einem higher-half Kernel gilt: physische Adresse = virtuelle Adresse – kernel_offset