Isolation und Schutz in Betriebssystemen

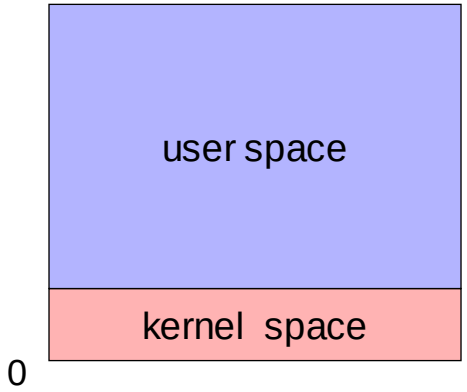
5. Kernel-Isolation bei x86-64

Michael Schöttner

Stand bisher

- Segmentierung:
  - User-Mode und Kernel-Mode Threads
  - Im User-Mode können keine privilegierten Befehle benutzt werden

- Paging:
  - Für jeden Prozess gibt es einen eigenen Adressraum
  - Der Kernel ist in jeden Adressraum eingeblendet, wird aber durch das U/S-Bit geschützt
    - Wir haben einen lower-half Kernel, d.h. er wird ab der Adresse 0 eingeblendet →



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

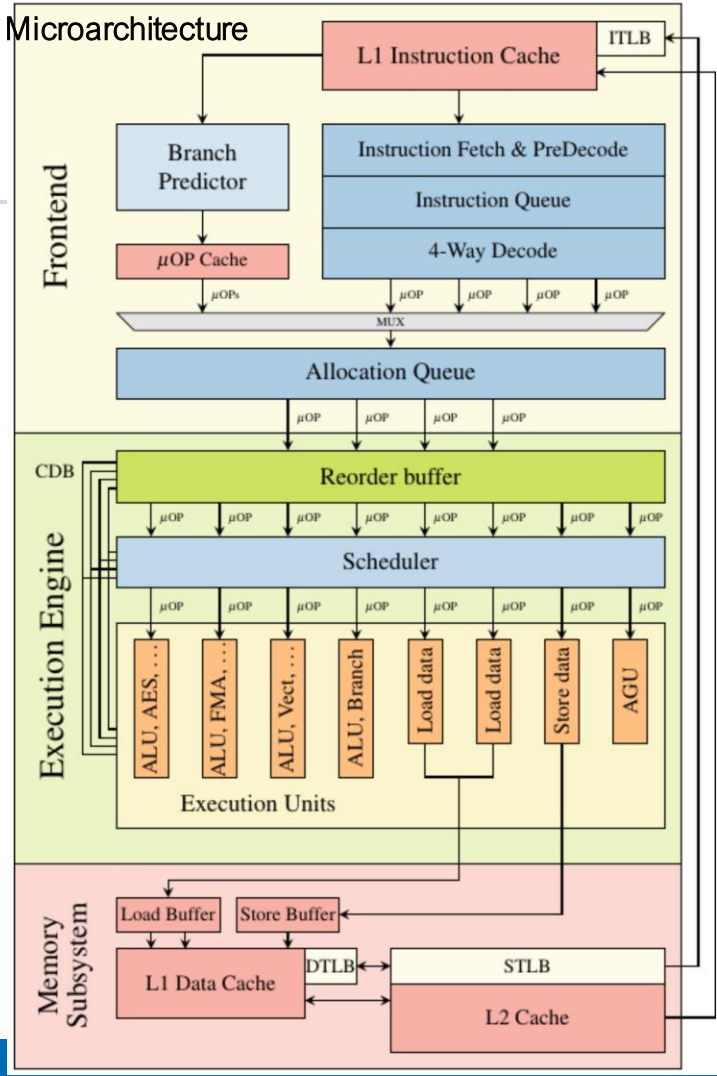
- Exploit basiert auf der Kombination verschiedener Beschleunigungs-funktionen in der Hardware und den Betriebssystemen
  - Parallele und spekulative Ausführung von Instruktionen
  - Caching & TLB
  - Adressräume

- Es handelt sich um einen sogenannten 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.

CPU: Out-of-Order Execution

- Problem: Taktraten können aufgrund physikalischer Grenzen nicht mehr viel erhöht werden.
  → Daher setzt man auf Parallelität

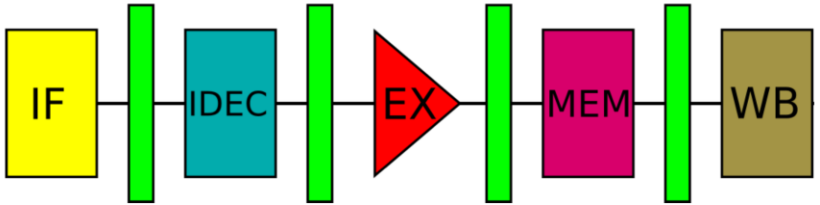
- x86 CPUs arbeiten intern mit einem Mikrocode und verschiedenen Funktionseinheiten
  - Die CPU sortiert die Instruktionen um, sodass diese Funktionseinheiten möglichst gut genutzt werden
  - Hierbei werden auch Instruktionen eines Threads automatisch umsortiert, sofern diese voneinander unabhängig sind. → out-of-order execution
  - Ergebnisse werden erst gültig gesetzt, sobald alle vorherigen Instruktionen abgearbeitet sind und keine Fehler aufgetreten sind, ansonsten werden die Ergebnisse verworfen



CPU: Pipelining

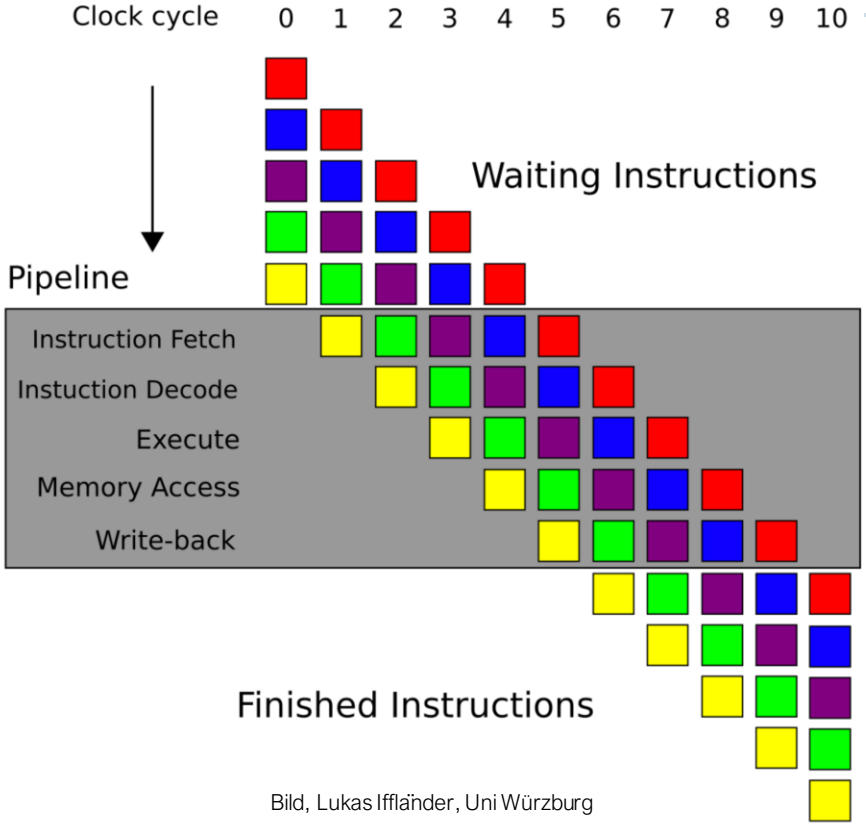
- Problem: Befehle sind komplex und Speicherzugriffe langsam

- Lösung: Pipelining
  - Befehle vorab 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

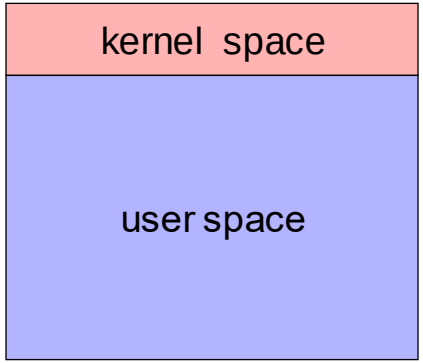
- Pipelining → Befehle vorab dekodieren und Daten vorab laden
- Problem: Bei einer Verzweigung im Code ist unklar, wo es weitergeht
- Lösung: Sprungvorhersage (engl. branch prediction) und spekulative Ausführung (engl. speculative execution)
  - Die Änderungen der Ausführung werden erst sichtbar, wenn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

CPU: Paging

Heinrich Heine
Universität Düsseldorf

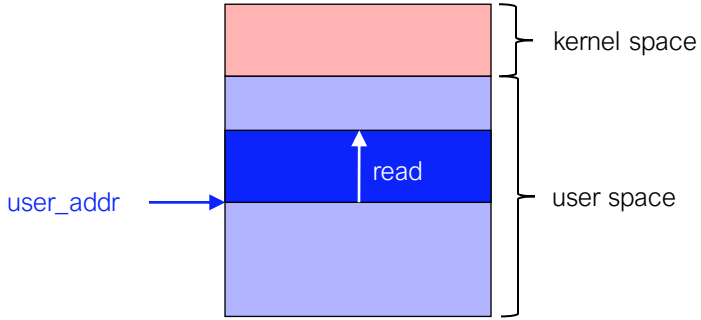
- Bisher Betriebssystem in jeden Adressraum eingeblendet →
  - I.d.R. im oberen Teil des logischen Adressbereichs
  - Dadurch ist der Zugriff auf Kernel-Funktionen schneller, da der Adressraum nicht umgeschaltet werden muss

- Zugriffe auf Kernel-Space sind geschützt durch die MMU
  - In den Page-Table-Einträgen gibt es das U/S Bit (User-Mode / Supervisor-Mode)
  - Greift ein Anwendungsprozess auf Kernel-Pages zu, so wird er terminiert
    - x86-CPU löst Protection-Fault aus
    - Bewirkt Signal SIGSEGV



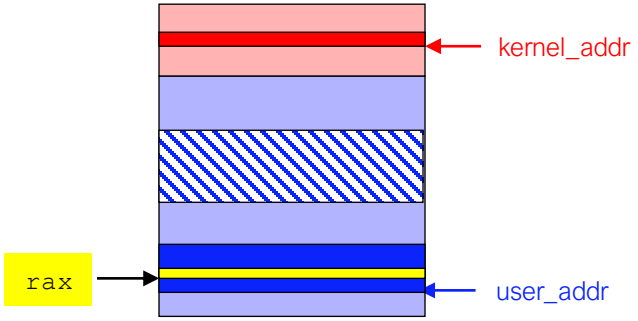
Meltdown Attack (Prinzip)

- 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)

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




```
; user-mode code

mov rax, [kernel_addr]       ; access kernel space
and rax, 0xff                ; use lowest byte
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, als der erlaubt 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)

- 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

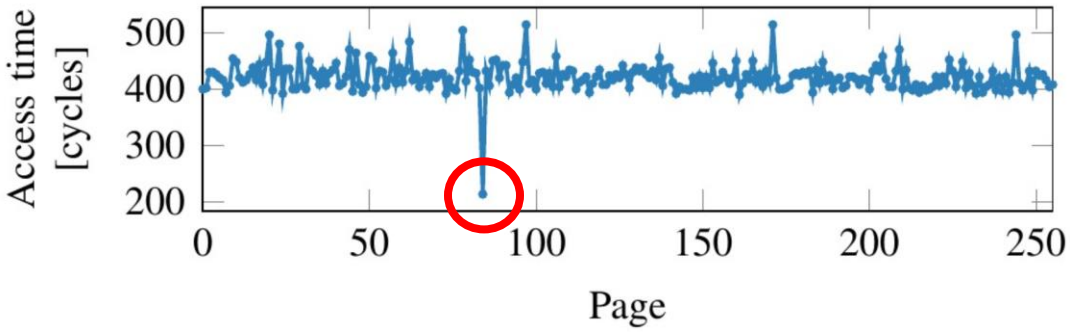


Bild aus der Publikation zu Meltdown

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?

- Aufgrund der spekulativen Ausführung und da das Betriebssystem und die Anwendung im gleichem Adressraum laufen
- Spekulative Ausführung
  - Die Instruktionen werden ausgeführt, bis die CPU mithilfe der MMU erkennt, dass der Zugriff auf den Kernel-Space nicht erlaubt war.
    - Exceptions werden zurückgestellt bis feststeht, ob Ausführung richtig oder falsch war
    - Dadurch wird der unerlaubte Speicherzugriff nicht sofort angebrochen (das ist wieder eine Performance-Optimierung)
  - Dann wird der CPU-Zustand zurückgesetzt, nicht aber der Cache-Inhalt.

Bemerkungen

Heinrich Heine
Universität Düsseldorf

- Einige 64-Bit Systeme blenden kompletten physikalischen Adressraum zusätzlich im Kernel-Space ein

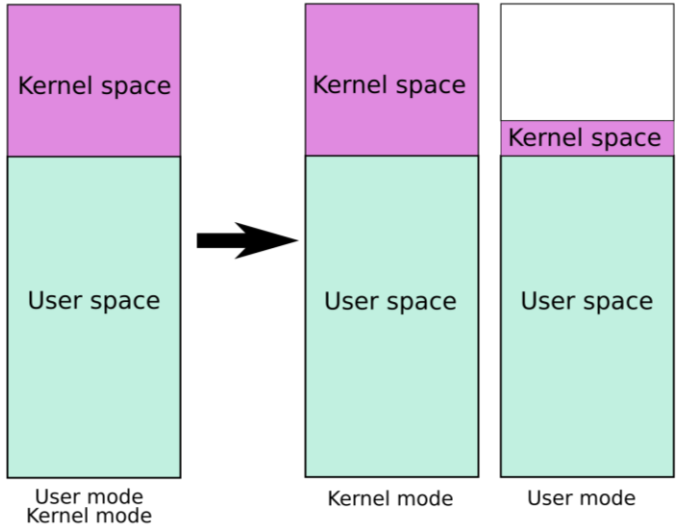
- Damit ist es mit Meltdown möglich den gesamten Speicher auszulesen
- Teilweise wurden in NTFS auch private Schlüssel von Dateisystemen im Kernel-Space gehalten, wodurch diese auch auslesbar sind und damit verschlüsselte Dateisysteme angreifbar sind

- 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

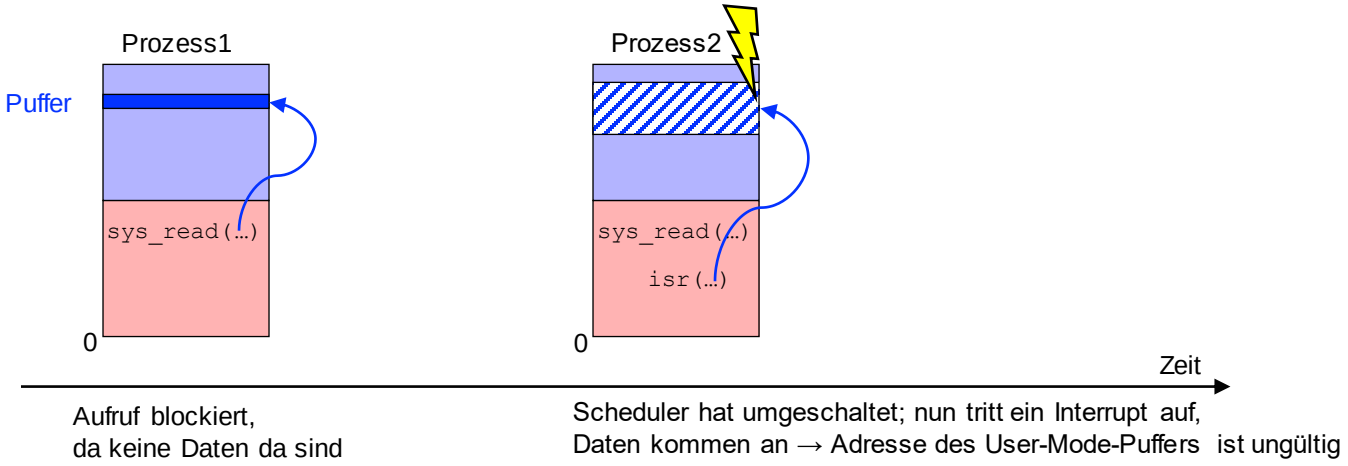
Datenaustausch bei Systemaufrufen

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)
- Bei vielen Systemaufrufen wird ein Pointer auf einen Puffer im Heap übergeben
- 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

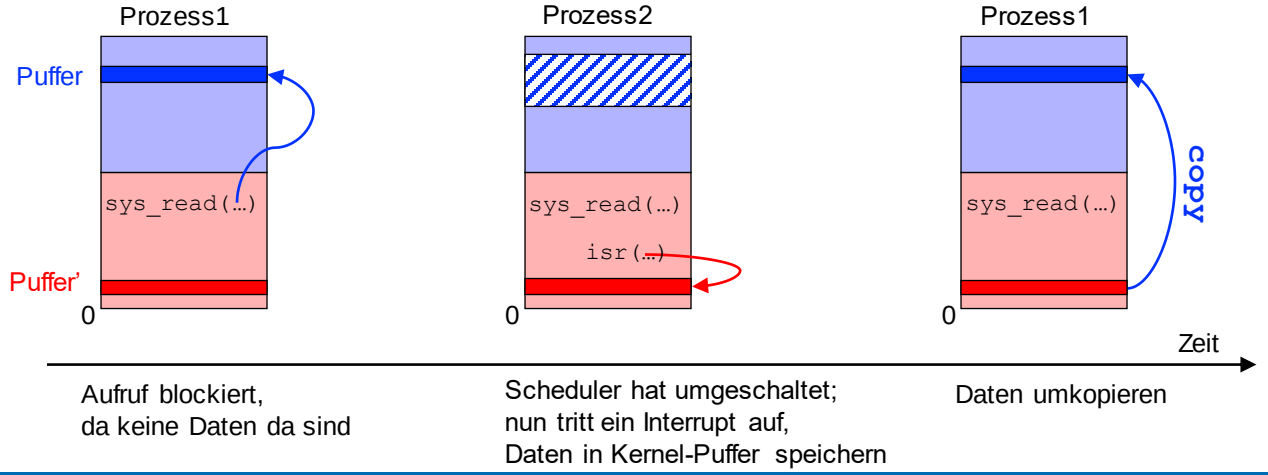
Systemaufruf: Lesen

- 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:



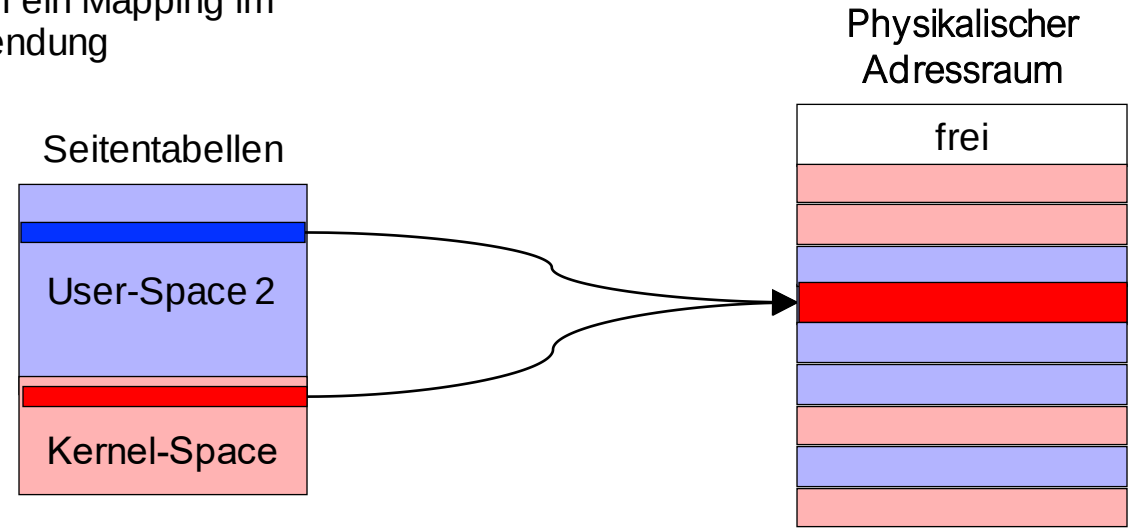
Systemaufruf: Schreiben

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

- 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