Betriebssysteme

Prozesse, Threads und Scheduling

Susanne Schaller, MMSc
Andreas Scheibenpflug, MSc

SE Bachelor (BB/VZ), 2. Semester

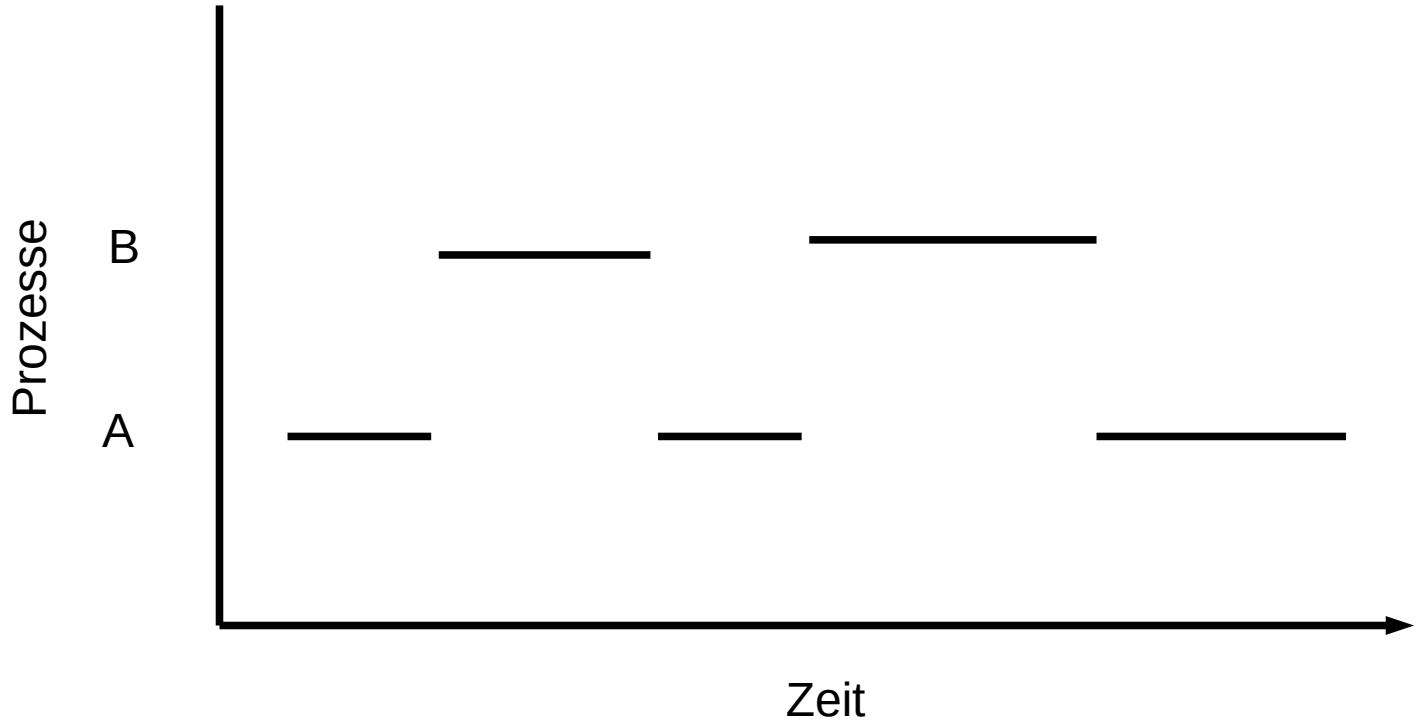
Prozess

- Abstraktion eines laufenden Programms
- Erlauben Pseudoparallelität wenn z.B. nur ein Prozessor vorhanden ist
- Gewöhnlich laufen auf einem Computer mehr Prozesse als Prozessoren/Kerne verfügbar sind
- OS kümmert sich darum, dass jeder Prozess CPU Zeit bekommt (Scheduling)
- Für den/die BenutzerIn sieht es so aus, als ob alle Prozesse gleichzeitig laufen

Prozessmodell

- Ein Prozess ist eine Instanz eines Programms zur Laufzeit und besteht aus
  - Code/Befehlszähler (Instruction Pointer)
    - Instruktion die als nächstes ausgeführt wird
  - Register
  - Variablen (Daten)
- Dem OS steht ein Prozessor zur Verfügung
- OS führt eine Menge an Prozessen pseudo-parallel aus
  - Jeder Prozess hat die CPU für eine begrenzte Zeit zur Verfügung
  - OS teil Prozessen die CPU zu und wechselt die Prozesse aus
  - Zu einem bestimmten Zeitpunkt wird immer nur ein Prozess ausgeführt

Prozessmodell



Prozesserzeugung

- Vier Ereignisse die zum Erzeugen eines neuen Prozesses führen
  - Initialisierung des OS
    - Beim Starten des OS werden Prozesse erzeugt
  - System Calls
    - Prozess, der einen anderen Prozess erzeugt (z.B. Linux: `fork()`, Windows: `CreateProcess()`)
  - Durch den/die BenutzerIn
    - z.B. Starten eines Programms in der Shell
  - Start eines Stapeljobs
    - z.B. Batchprocessing bei Großrechnern (Mainframes)
- Technisch gesehen meist Variante 2 (System Calls)

Prozesserzeugung - Beispiel

- `fork` dupliziert den Prozess inkl. Speicher
- Rückgabewert von `fork` ist
  - 0 im Kindprozess
  - PID des Kindes im Elternprozess
- `execve` erlaubt das Ausführen eines Programms
  - Ersetzt den aktuellen Prozess inkl. Speicher durch das auszuführende Programm

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

// gcc fork.c -o fork
// ./fork
int main(void)
{
        pid_t pid_new;
        int status;
        char *args[] = {"uname", "-a", NULL};
        char *env[] =  {NULL};

        pid_new = fork();

        if (pid_new == 0)
        {
                printf("Child process says Hello\n");
                execve("/usr/bin/uname", args, env);
                printf("Never executed code if execve is successful\n");
        }
        else
        {
                printf("Original process says Hello!\n");
                wait(&status);
                printf("Original process: Child finished, exiting...\n");
        }
        return 0;
}
```

```
Original process says Hello!
Child process says Hello
Linux ThinkPad-L390 5.4.0-31-generic #35-Ubuntu SMP Thu May 7 20:20:34 UTC 2020
x86_64 x86_64 x86_64 GNU/Linux
Original process: Child finished, exiting...
```

Prozessbeendigung

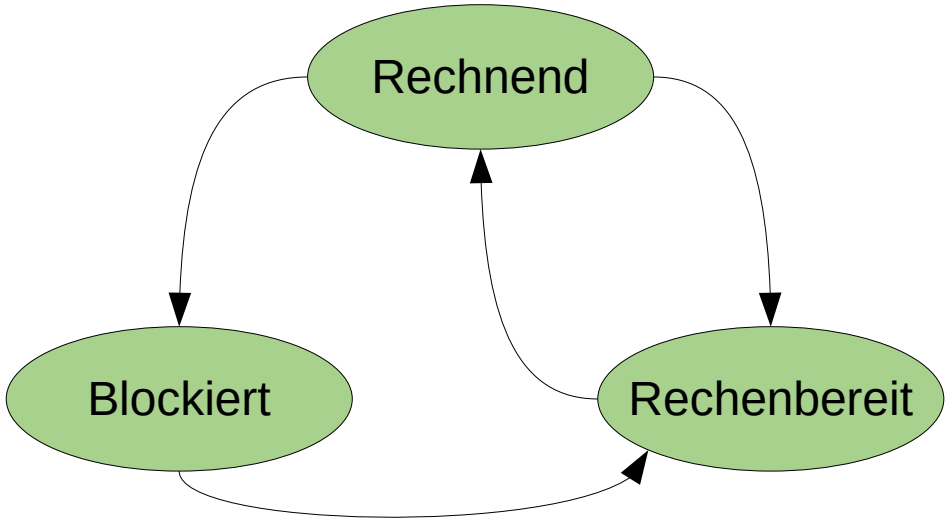
- Vier Ereignisse, die zum Beenden eines Prozesses führen
  - Normales Beenden (freiwillig)
    - Aufgabe beendet (z.B. Linux: `exit(0)`, Windows: `ExitProcess()`)
  - Beenden aufgrund eines Fehlers (freiwillig)
    - Programm stellt Fehler fest und beendet mit z.B. `exit(-1)`
  - Beenden aufgrund eines schwerwiegenden Fehlers (unfreiwillig)
    - z.B. Programmierfehler, Division durch Null, Zugriff auf ungültige Speicheradressen,...
  - Beenden durch einen anderen Prozess (unfreiwillig)
    - z.B. Linux mit `kill()` oder Windows mit `TerminateProcess()`

Prozesszustände

- Prozesse müssen oft warten auf
  - Ein- und Ausgabe auf Geräten
    - z.B. Lesen vom Dateisystem, `HTTP GET`, …
  - Andere Prozesse
    - z.B. `cat file.txt | grep Linux | sort`
- Dieser Zustand wird als „blockiert“ bezeichnet
- Im Gegensatz zum Stoppen eines rechenbereiten Prozesses weil das OS dies entscheidet
  - z.B. Umschalten zu einem anderen Prozess weil aktueller Prozess seinen Anteil an Rechenzeit verbraucht hat

Prozesszustände

- Drei Zustände
  - Rechnend
    - Prozess läuft auf der CPU und führt Befehle aus
  - Rechenbereit
    - Prozess ist bereit aber gestoppt, weil gerade ein anderer Prozess rechnet
  - Blockiert
    - Nicht lauffähig, da auf ein Ereignis (z.B. Benutzereingabe, Datei eingelesen,...) gewartet wird



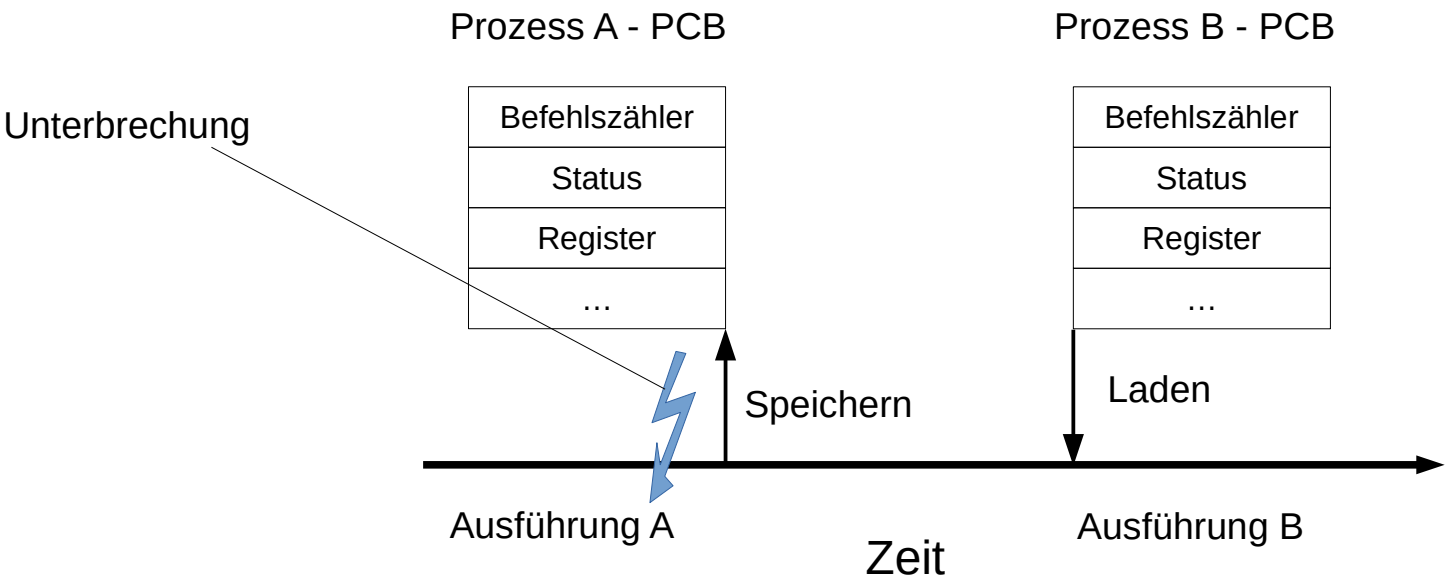
Prozesstabelle

- OS besitzt eine Prozesstabelle, welche eine Liste aller Prozesse enthält
- Ein Eintrag in dieser Liste wird als Prozesskontrollblock (PCB) bezeichnet
  - Enthält alle Informationen (Zustand) zu einem Prozess
  - Ermöglicht damit das Fortsetzen von Prozessen
  - Ermöglicht Scheduling Entscheidungen zu treffen

Prozessverwaltung	**Speicherverwaltung**	**Dateiverwaltung**
Register	Zeiger auf Textsegment	Arbeitsverzeichnis
Befehlszähler	Zeiger auf Datensegment	Dateideskriptor
Stackpointer	Zeiger auf Stacksegment	User Id
Programmstatuswort	...	Gruppen Id
Prozess ID (PID)		...
Benutzte CPU Zeit		
Prozesszustand		

Prozesswechsel

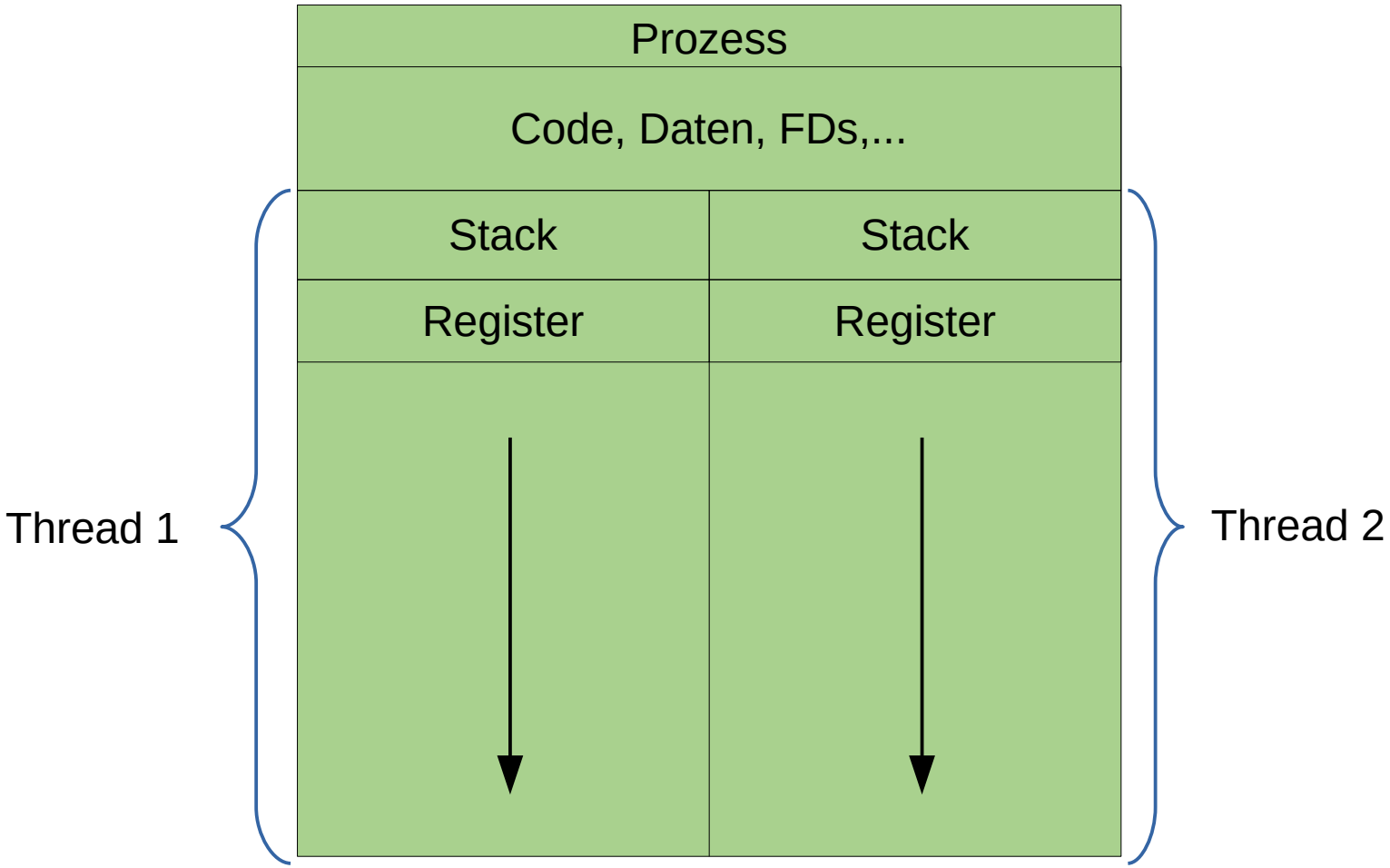
- Anzahl Prozesse > Anzahl CPUs (in unserem Modell == 1)
  - OS muss zwischen Prozessen wechseln, da jeder Prozess CPU Zeit zur Ausführung benötigt
- Prozesswechsel - Context Switch
  - Laufender Prozess wird unterbrochen
  - Scheduler wählt einen anderen, rechenbereiten Prozess aus und führt diesen aus
- Informationen zu unterbrochenem Prozess müssen gespeichert werden
  - Damit dieser später wieder ausgeführt werden kann
  - Zustand wird im PCB gespeichert



Threads

- Jeder Prozess hat seinen eigenen Adressraum („Speicher“)
  - CPU führt eine Instruktion nach der anderen sequentiell aus
- Ein Prozess kann mehrere Threads starten
  - Threads sind parallel laufende Ausführungspfade
  - Im selben Adressraum wie der Prozess
- Vorteile
  - Einfacheres Programmiermodell (im Vergleich zu z.B. IPC)
  - Leichtgewichtiger als Prozesse
  - Performance, z.B. 1 Thread rechnet während ein 2. auf I/O wartet
  - Performance bei Multicore Prozessoren

Threads


Prozess
Code, Daten, FDs,...
Stack
Stack
Register
Register
Thread 1
Thread 2

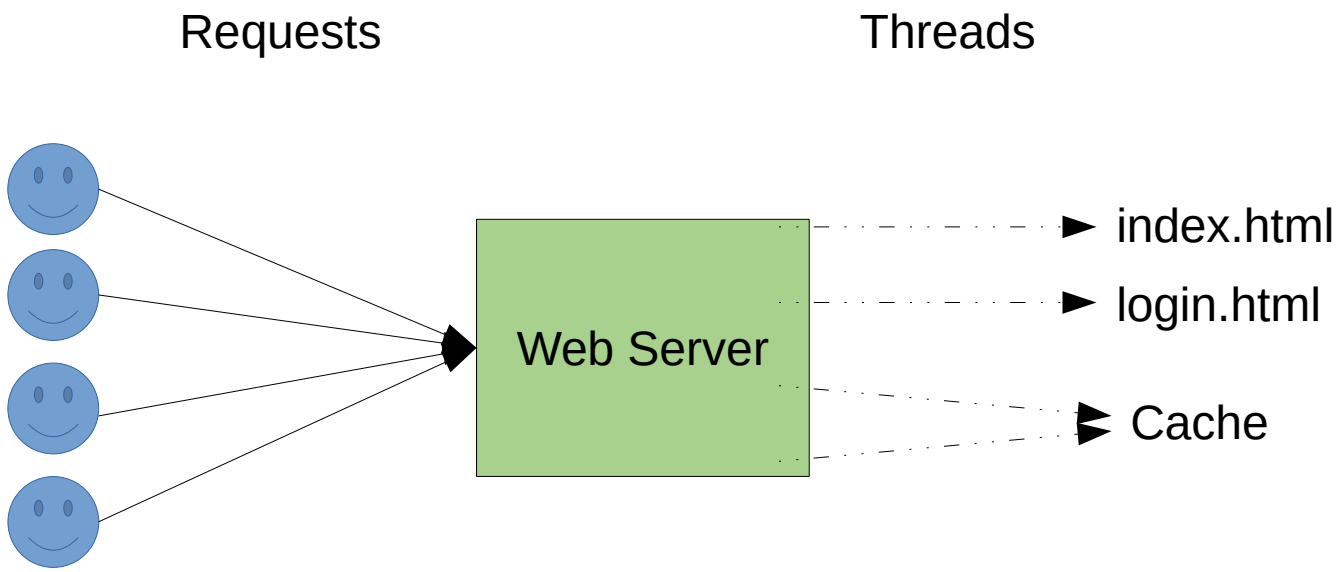
Threads

- Implementierung
  - Im OS: OS hält Thread Tabelle ähnlich zur Prozesstabelle mit z.B.:
    - Stack
    - Register
    - Befehlszähler
    - Threadzustand
  - Im User Space: Implementiert Funktionalität in Laufzeitsystem (Bibliothek)
    - Bessere Performance
    - Probleme bei Blocking I/O, Page Faults,… → andere Threads sind auch blockiert
  - Hybrid: Kombination aus OS und User Space Threads

Threads - Implementierungen

- Windows
  - „Echte“ OS Threads, Verwenden der `CreateThread(...)` Funktion
  - Jeder Prozess besteht aus mindestens einem Thread
  - Hybrid Threads mit User-Mode Scheduling
- Linux
  - Threads werden durch das OS verwaltet
  - Es wird aber die Implementierung der Prozessverwaltung verwendet
  - Das bedeutet beinahe kein Unterschied zwischen Prozessen und Threads aus Kernel Sicht
  - Verwendung der Funktion `clone(…)` und setzen der Flags für das Erlauben des Zugriffs auf die File Deskriptoren etc. im Prozessadressraum
  - PID bleibt allerdings gleich

Threads - Beispiel



Threads - Beispiel

- Diskussion: Implementierungsmöglichkeiten
  - 1 Prozess, sequenzielle Abarbeitung
  - Mehrere Prozesse, Parallelität
  - 1 Prozess, mehrere Threads (pro Request?)
  - 1 Prozess, Non-Blocking IO
  - ...

POSIX Threads - Beispiel

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

// gcc -pthread threads.c -o threads
// ./threads

void *say_hello_thread(void* id)
{
        printf("Hi! I'm thread %d\n", id);
        pthread_exit(NULL);
}

int main(void)
{
        static int nr_of_threads = 20;
        pthread_t threads[nr_of_threads];

        for(int i = 0; i < nr_of_threads; i++)
        {
                pthread_create(&threads[i], NULL, say_hello_thread, (void*) i);
        }

        for(int i = 0; i < nr_of_threads; i++)
        {
                pthread_join(threads[i], NULL);
        }

        return 0;
}
```

```
Hi! I'm thread 0
Hi! I'm thread 3
Hi! I'm thread 2
Hi! I'm thread 5
Hi! I'm thread 1
Hi! I'm thread 4
Hi! I'm thread 6
Hi! I'm thread 7
Hi! I'm thread 8
Hi! I'm thread 9
Hi! I'm thread 10
Hi! I'm thread 11
Hi! I'm thread 13
Hi! I'm thread 12
Hi! I'm thread 14
Hi! I'm thread 16
Hi! I'm thread 17
Hi! I'm thread 18
Hi! I'm thread 19
Hi! I'm thread 15
```

Threads/IPC

- Interprocess Communication
- Drei Themen
  - Informationsweitergabe
    - Austausch von Informationen zwischen Prozessen
  - Synchronisation
    - Warten auf andere Prozesse
  - Abhängigkeiten/Reihenfolgen
    - Mehrere Prozesse, die in einer definierten Reihenfolge eine Aufgabe abarbeiten sollen (→ Race Conditions)
- Nur der erste Punkt betrifft Threads nicht →selber Adressraum
- Daraus ergeben sich eine Reihe von Problemen und Lösungen: Race Conditions, Critical Regions, Mutex, Semaphoren → LVA VPS

Scheduling

- Bei PCs / mobilen Geräten laufen mehr Prozesse als CPUs zur Verfügung stehen
- Die Prozesse müssen sich daher die Ressource CPU teilen
- Scheduler ist Teil des OS
  - Entscheidet welcher Prozess als nächstes ausgeführt wird
  - Diese Berechnung der Entscheidung wird auch als Scheduling bezeichnet und durch den Scheduling Algorithmus, den der Scheduler implementiert, getroffen
  - Das Resultat der Berechnung führt zu einem Context Switch

Ziele

- Wir werden uns auf interaktive Systeme fokusieren
  - Im Gegensatz zu Batchverarbeitungssystemen oder Echtzeitsystemen
- Ziele
  - Fairness
    - Vergleichbare Prozesse sollten auch den gleichen Anteil an Rechenzeit bekommen
  - Policy Enforcement
    - Unterstützt das OS Policies (z.B. Prioritäten, Deadlines), sollten diese auch eingehalten werden
  - Balance
    - Optimale Ausnutzung der HW Ressourcen
    - Gleichmäßige Auslastung der CPU und der I/O Geräte ist zu bevorzugen
  - Response Time
    - Minimierung der Zeit zwischen einer Benutzereingabe und einer dementsprechenden Rückmeldung
    - Vorrang gegenüber z.B. Hintergrundtasks (Daemons/Services)

Things to consider

- Context Switch ist nicht gratis
  - Wechsel in den Kernel Space, Speichern des PCBs,…
  - Viele Context Switches kosten Performance
    - Steht im Gegensatz zu Ziel Response Time
- Verhalten von Prozessen
  - CPU-bound: Verbringen die meiste Zeit mit Berechnungen
  - I/O-bound: Verbringen die meiste Zeit mit Ein- und Ausgabe
  - Scheduler muss auf das Verhalten von Prozessen Rücksicht nehmen
    - z.B. Bevorzugung von I/O-bound Prozessen, damit diese die Festplatte „beschäftigen“, während CPU-bound Prozesse die CPU beschäftigen → Ziel Balance

Zeitpunkte

- Es gibt mehrere Ereignisse die einen Context Switch auslösen
  - Wenn ein neuer Prozess erzeugt wird
    - Ausführung des Elternprozesses oder des neuen Prozesses
  - Beendigung eines Prozesses
  - Wenn ein Prozess blockiert
    - Blockierendes I/O (System Call), Semaphore, Mutex,…
  - Auftreten eines I/O Interrupts
    - Wurde eine I/O Operation fertig, kann ein Prozess, der auf diese wartet, weiter ausgeführt werden (blockierend → rechenbereit)
    - Oder es könnte trotzdem der aktuelle Prozess weiter ausgeführt werden weil er z.B. eine höhere Priorität hat

Context Switch – Am Beispiel eines Interrupts

- Ein IRQ tritt auf
  - CPU prüft nach der Ausführung der aktuellen Instruktion ob ein IRQ aufgetreten ist
- CPU stoppt Ausführung des aktuellen Prozesses und legt Instruction Pointer etc. auf einen Stack
- CPU lädt mit Hilfe der Interrupt Descriptor Table (IDT) den Interrupt Handler
  - Sichert Register des zuvor ausgeführten Prozesses und die Daten die zuvor von der CPU auf einen Stack gelegt wurden
  - Richtet Stack ein
  - Führt das Interrupt Handling aus (z.B. Lesen von Daten in einen Puffer)
  - Bereinigt Stack/Register
- Aufruf des Schedulers
  - Selektion des nächsten Prozesses
- Ausführen des selektierten Prozesses
  - Laden des Instruction Pointers, Registers,...

Clock Interrupts

- HW Uhr stellt periodische (I/O) Interrupts zur Verfügung
  - Preemptive Scheduler verwenden diese (auch), um über Context Switches zu entscheiden
- Non-preemptive Scheduling
  - Context Switch nur, wenn ein Prozess blockiert oder die CPU „freiwillig hergibt“
- Preemptive Scheduling
  - Prozesse bekommen eine maximale Ausführungszeit (Zeitscheibe, „Quantum“) zugeordnet
  - Wird diese überschritten, wird der Prozess unterbrochen, auch wenn dieser noch rechenbereit ist, und ein anderer Prozess gestartet
  - Standard bei interaktiven Systemen

Schedulingalgorithmen

- Unterschiedliche OS implementieren unterschiedliche Algorithmen
  - Und Speichern auch unterschiedliche Daten im PCB aufgrund welcher eine Entscheidung zur Prozessselektion getroffen wird
- Hängt von dem Einsatzgebiet und der Zielsetzung ab
- Ein Batchsystem verfolgt andere Ziele als ein Echtzeitsystem oder ein interaktives System
  - Daher ist auch der Schedulingalgorithmus zwischen diesen Systemen anders
  - Ein Batchsystem kann z.B. auch mit non-preemptivem Scheduling auskommen
- Wir sehen uns im Folgenden einige Beispiele von Schedulingalgorithmen an

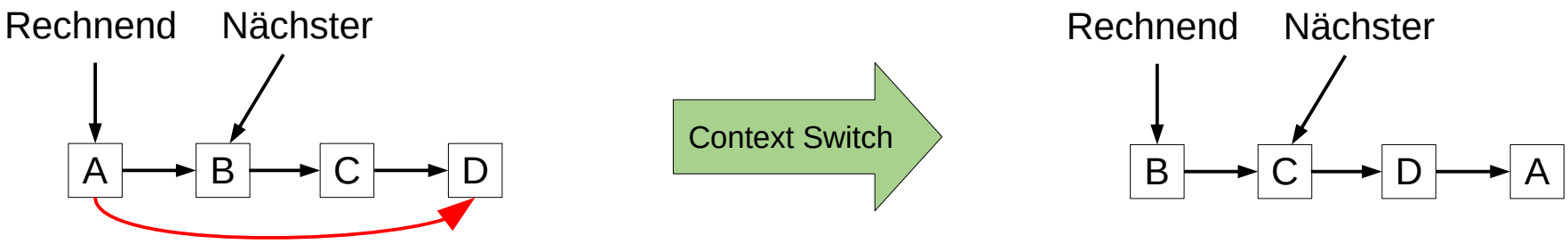
First Come, First Served (FIFO)

- Verwendung z.B. bei Batchsystemen
  - Non-preemptive
  - Neue Prozesse werden in eine Warteschlange (Queue) eingereiht
  - Prozesse werden der Reihe nach entnommen und abgearbeitet
  - Nachteil: Ziel der Balance wird nicht gewährleistet bei I/O-bound Prozessen

Round Robin

- Jeder Prozess bekommt das gleiche Quantum
  - Läuft er noch nach Ablauf des Quantums, wird er unterbrochen und ein anderer Prozess wird fortgesetzt (preemptive)
  - Blockiert der Prozess früher, wird ebenfalls gewechselt
- Daraus resultiert: Jeder Prozess ist gleich wichtig
  - Vorteil: Fair
  - Nachteil: Vielleicht nicht ganz richtig
- Weiterer Nachteil – Wahl des Quantums:
  - Kleines Quantum → Verschwendung von CPU Zeit
  - Großes Quantum → Beeinträchtigung der Interaktivität

Round Robin



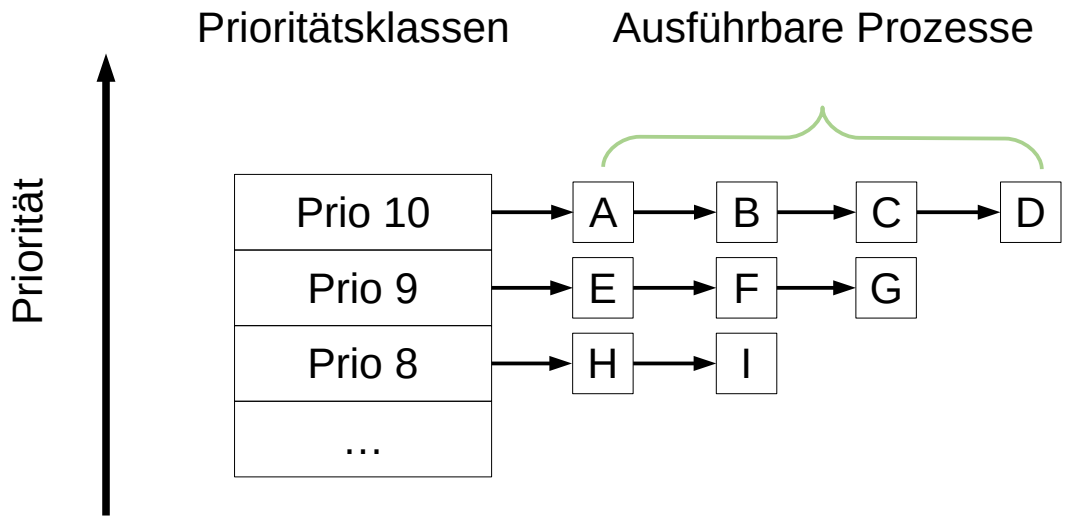
- Scheduler verwaltet eine Liste von Prozessen
- Nach der Ausführung wird der Prozess an das Ende der Liste gereiht und der nächste Prozess in der Liste ausgeführt
- Einfach zu implementieren
  - Wurde von Linux in frühen Versionen (0.x – 1.x) verwendet

Prioritäts-Scheduling

- Prozesse erhalten eine Priorität
  - Höhere Priorität bedeutet, dass diese Prozesse öfter ausgeführt werden und damit auch mehr Rechenzeit bekommen
- Statisch
  - Priorität wird festgelegt und bleibt unverändert
  - z.B. Webbrowser vs. periodischer E-Mail Check
  - z.B. Multiuser Systeme: General vs. Soldat
- Dynamisch
  - Priorität wird vom OS dynamisch angepasst
  - Setzt voraus, dass das OS z.B. verbrauchte Quantumzeit aufzeichnet
    - I/O-bound Prozesse bekommen eine höhere Priorität → Werden schneller/häufiger ausgewählt weil sie die CPU nur kurz belegen → Ziel Balance

Prioritäts-Scheduling - Beispiel

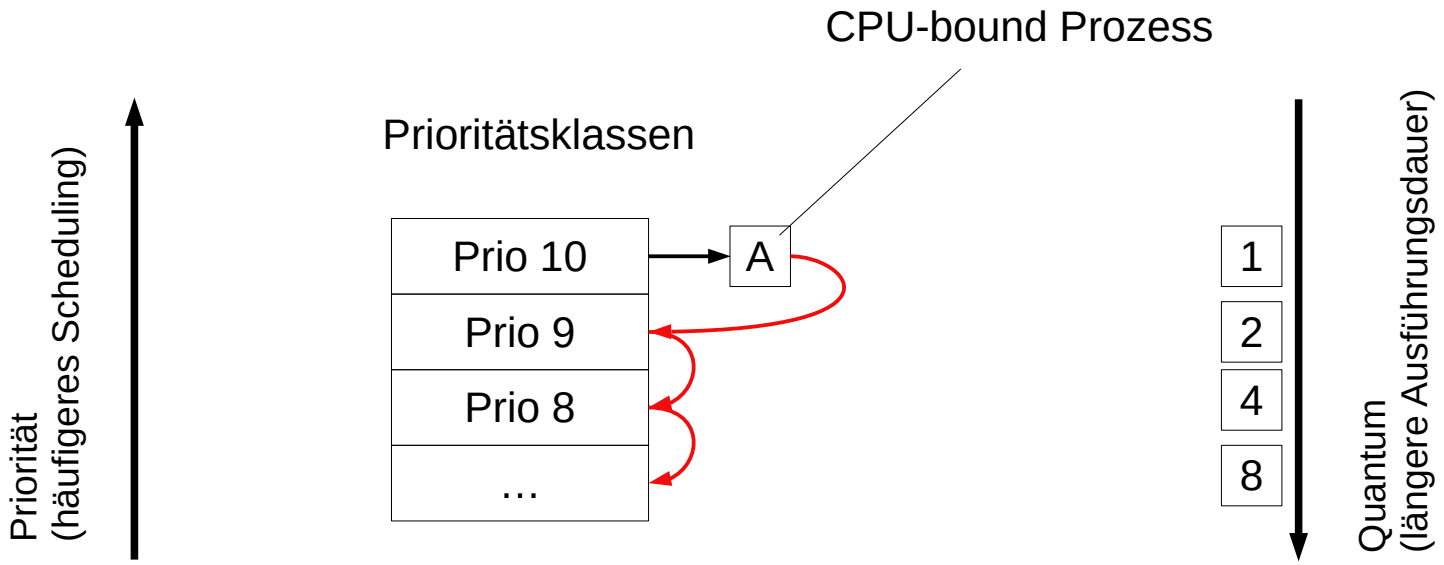
- f: Zeit (anteilig) des letzten Quantums, die für die Berechnung verwendet wurde
  - z.B. Quantum: 100; I/O-bound: 10/100 = 0,1; CPU-bound: 100/100 = 1
- 1/f: Priorität
- CPU-bound: Kleine Priorität, z.B. 1/1 = 1
- I/O-bound: Hohe Priorität, z.B. 1/0,1 = 10
- Gruppierung in Prioritätsklassen
  - Round Robin innerhalb dieser Klassen solange die Prozesse der Klasse rechenbereit sind
  - Ignorieren der anderen Klassen, Wechsel in die niederpriore Klasse erst, wenn keine Prozesse mehr rechenbereit sind
  - Nachteil: Verhungern von niederprioren Prozessen → Dynamische Anpassung der Priorität notwendig



Multilevel Feedback Queues (MLFQ)

- Bis jetzt haben wir festgestellt, dass I/O-bound Prozesse eher eine höhere Priorität bekommen sollen
- Erweiterung nun bei CPU-bound Prozessen
  - Besser diese für längere Zeit rechnen zu lassen, als oft für kurze Zeit
  - Minimierung der Context Switches
- Daher: Prozessen in unterschiedlichen Prioritätsklassen wird eine unterschiedliche Anzahl an Quanten zugeteilt
  - Höchste Prio Klasse: 1 Quantum
  - Nächste Prio Klasse: 2 Quanten
  - Nächste Prio Klasse: 4 Quanten
  - …
- Je nachdem wie viel ein Prozess rechnet oder auf I/O wartet, wird dieser zwischen den Klassen verschoben und
  - damit die Priorität geändert
  - die Menge an CPU Zeit, die der Prozess erhält, dynamisch angepasst

MLFQ - Beispiel



- Beispiel: Prozess A benötigt 20 Quanten und kein I/O
  - Startet in Prioritätsklasse 10 und wird unterbrochen → kann nicht fertig rechnen
  - A wird daher nach unten gereiht (CPU-bound)
  - A wird das nächste Mal in Prioritätsklasse 9 gestartet
  - A bekommt daher weniger häufig die CPU
  - A bekommt dafür die CPU für eine längere Zeit
- Bsp: Was würde passieren wenn A plötzlich beginnt I/O Operationen durchzuführen?
- Bsp: Vergleich Round Robin: 5 vs. 20 Context Switches

Implementierungen

- OS verwenden oft Varianten/Mischformen von MLFQ
  - Mac OS X, (Net|Free)BSD + Prioritäten
- Windows
  - 3.1x: Non-preemptive Scheduler
  - 95: Preemptive Scheduler
  - >= Vista: MLFQ + Magic (Closed Source)
    - Windows Scheduler arbeitet mit Threads
    - Threads besitzen zusätzlich eine statische Priorität (kann von BenutzerIn oder im Code gesetzt werden)
    - Priorität kann aber auch vom OS angepasst werden
    - Ausnahme: Threads der Real Time Prioritätsklasse

Implementierungen - Linux

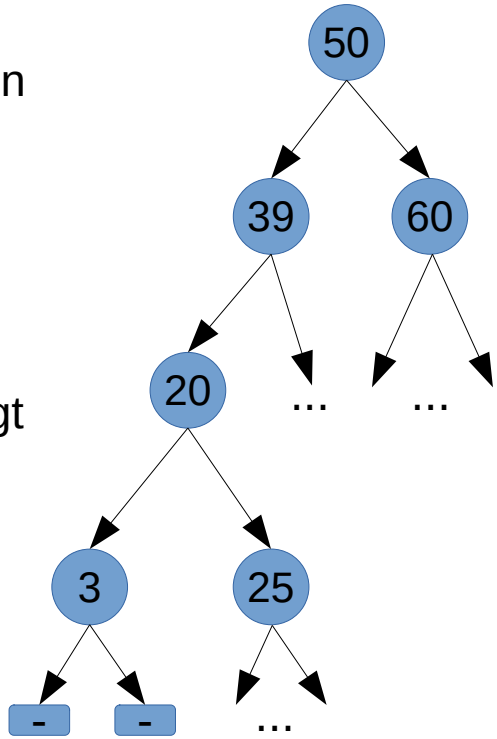
- Linux 0.x – 1.x: Round Robin
- Linux 2.2: Erweiterungen Scheduling Klassen und SMP Support
- Linux 2.4: O(n) Scheduler
  - Iteration über alle Prozesse der Run Queue → O(n)
  - Auswahl des nächsten Prozesses aufgrund einer Heuristik
  - Mitnahme von nicht verbrauchtem Quantum (wenn Prozess blockiert)
- Linux 2.6: O(1) Scheduler
  - Auswahl des nächsten Prozesse aus der Run Queue in O(1)
  - Allerdings viele Heuristiken zur Feststellung ob ein Prozess CPU-bound oder I/O-bound ist → komplex
- > Linux 2.6.23: Completely Fair Scheduler (CFS)
  - Verhalten kann durch Scheduling Policies verändert werden (neben dem standard, interaktiven Scheduling gibt es auch noch FIFO, Round Robin und Deadline)
  - Deadline Policy ist für Echtzeitsysteme ausgelegt
- Alternative: Brain Fuck Scheduler (BFS) / MuQSS (Con Kolivas)

Completely Fair Scheduler

- *CFS basically models an 'ideal, precise multitasking CPU' on real hardware. CFS's design is quite radical: it does not use runqueues, it uses a time-ordered rbtree to build a 'timeline' of future task execution“* - Ingo Molnár
- „Fair“ weil jeder Prozess Anspruch auf gleich viel Prozessorzeit hat
  - Jeder Prozess bekommt also den selben Anteil an CPU Zeit unter Berücksichtigung der anderen Prozesse (Maximum Execution Time)
  - Maximum Execution Time für einen Prozess wächst mit der Dauer, die er wartend verbringt („Sleeper Fairness“)
  - Die Zeit vergeht allerdings für low-prio Prozesse schneller
  - Da sich die Anzahl an Prozessen laufend ändert, gibt es kein fixes Quantum für einen Prozess
- Für jeden Prozess wird die Zeit, die er ausgeführt wird, aufgezeichnet (`vruntime`)
- Es gibt keine spezielle Behandlung oder Heuristik zur Feststellung der Interaktivität eines Prozess (→ einfacher als O(1) Scheduler)

Completely Fair Scheduler

- Die Datenstruktur (Run Queue) für die Prozesse ist ein Rot-Schwarz Baum, der nach der `vruntime` sortiert ist
  - Knoten mit der kleinsten `vruntime` befinden sich im Baum links unten
- Es wird immer der linkeste Knoten (Prozess) entnommen und als nächstes ausgeführt
  - Der Prozess wird für Maximum Execution Time ausgeführt
  - Die `vruntime` des Prozesses wird aktualisiert und der Prozess nach der Ausführung wieder im Baum nach der `vruntime` sortiert eingefügt
  - Während ein Prozess ausgeführt wird, wächst die Maximum Execution Time der anderen Prozesse
  - Damit gibt es einen neuen linken Knoten welcher als nächstes entnommen und ausgeführt wird
- CFS wurde über die Jahre weiterentwickelt
  - Auto-group von Prozessen, die logisch zusammengehören, um Fairness zu steigern
  - Verbesserungen im Bereich Multicore Performance



A Decade of Wasted Cores[1]

- Eine Run Queue pro Prozessor/Core
  - Bei einer einzigen globalen Run Queue wäre Locking notwendig → Performance
- CPU Affinity für Prozesse
  - Prozesse werden einer CPU/Core zugeordnet
    - Um Cache Misses zu reduzieren
    - Das Verschieben eines Prozesses von einer Run Queue in eine andere ist eine teure Operation
- Load Balancing zwischen CPUs/Cores
  - Ziel ist, alle CPUs/Cores möglichst gleichmäßig auszunutzen
  - Daher wird periodisch ein Load Balancing Algorithmus ausgeführt, der eine Umverteilung der Prozesse, falls benötigt, durchführt

[1] http://www.ece.ubc.ca/~sasha/papers/eurosys16-final29.pdf

Zusammenfassung

- Prozesse
- Threads
- Prozesszustände
- Prozessverwaltung
- Scheduling
- Schedulingalgorithmen

Betriebssysteme

Speicherverwaltung

Susanne Schaller, MMSc
Andreas Scheibenpflug, MSc

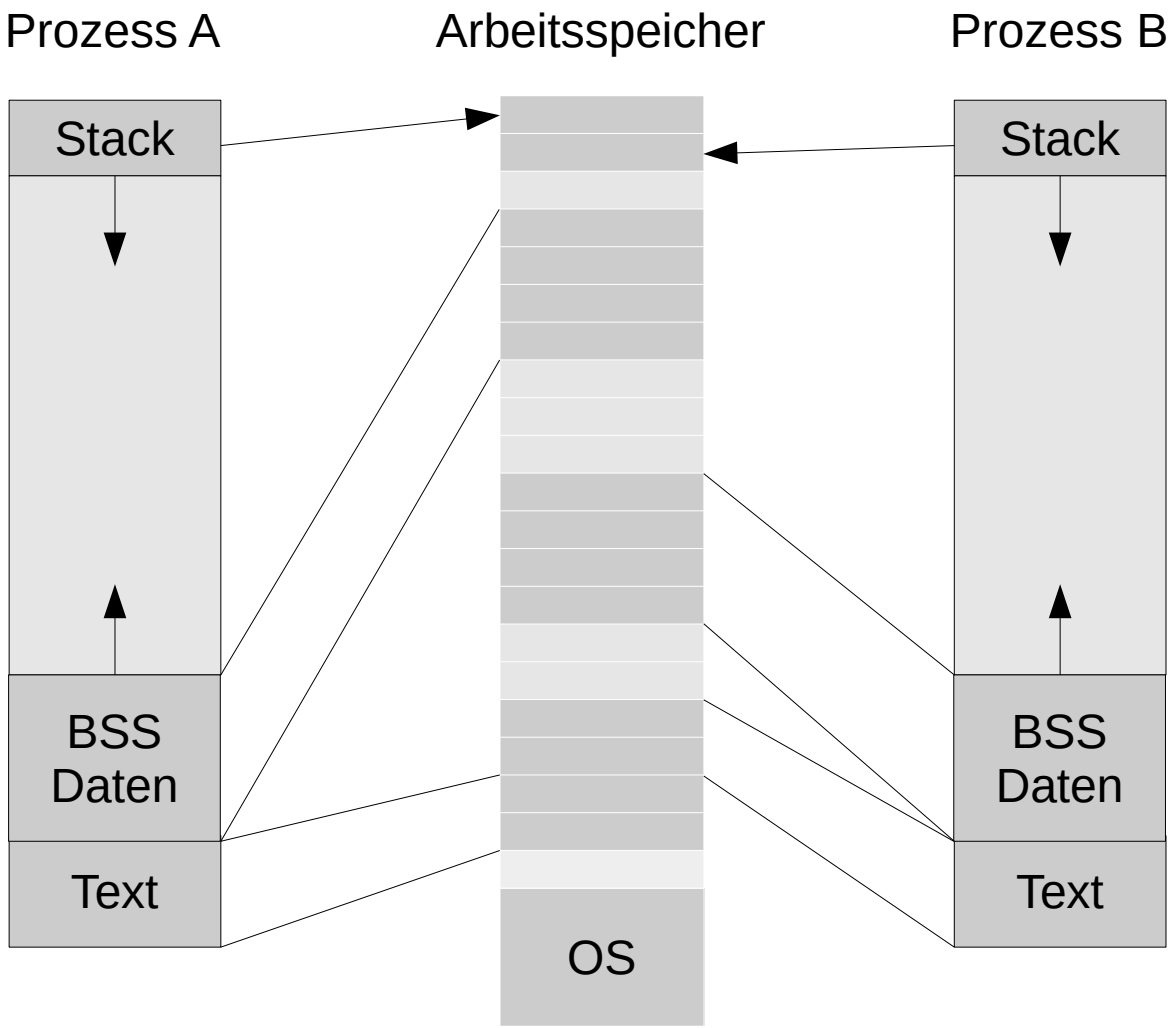
SE Bachelor (BB/VZ), 2. Semester

Speicherverwaltung

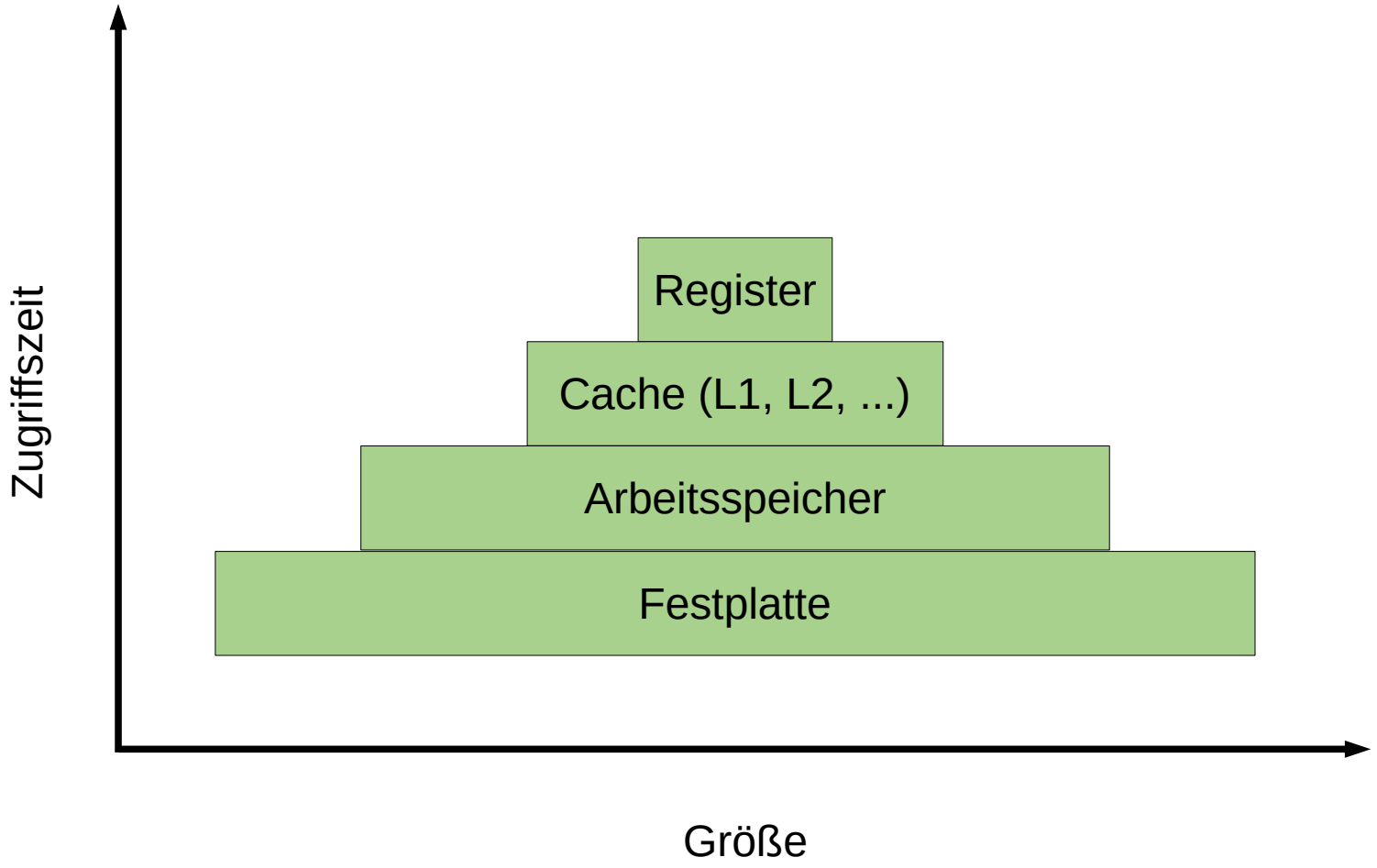
- Prozesse benötigen Arbeitsspeicher für
  - Code (Instruktionen)
  - Daten (Variablen, Heap, Stack)
- OS verwaltet den Hauptspeicher (RAM)
  - Zuteilung von Speicher zu Prozessen
  - Allokieren / Freigeben von Speicher
  - Auslagern (Paging) auf die Festplatte
  - Optimale Ausnutzung des Speichers

Speicherlayout (Linux)

- Text Segment
  - Instruktionen/Code
- Daten
  - Initialisierte Daten, z.B. statische Variablen, globale Variablen
- BSS
  - „Block Started by Symbol“
  - Nicht initialisierte Variablen
  - Heap
- Stack
  - Lokale Variablen, Rücksprungadressen, Argumente
  - Stack Pointer zeigt auf die Startadresse dieses Bereichs



Speicherhierarchien



Abstraktion

- Ein OS stellt auch mit der Speicherverwaltung eine Abstraktionsschicht für Programme im User Space zur Verfügung
- Virtual Memory ist dabei die gebräuchlichste Technik
- Wir sehen uns zur Einleitung aber auch noch weitere Möglichkeiten an
  - Keine Abstraktion
  - Adressräume

Keine Abstraktion

- Fand Anwendung bei z.B. Mainframe Computer vor 1960 oder PCs vor 1980
- Findet noch Anwendung bei manchen Eingebetteten Systemen
- Jede Operation in Bezug auf Arbeitsspeicher, die eine Anwendung durchführt, wird direkt 1:1 auf dem physischen Speicher ausgeführt
  - z.B. JMP 28 springt an die tatsächliche Speicheradresse 28
- Konsequenz: Es kann nur ein Prozess im Speicher gehalten werden da
  - Speicherzugriffsverletzungen (Segfault) nicht festgestellt bzw. verhindert werden können
  - Adressierung bei mehreren Prozessen im Speicher nicht mehr funktionieren würde
- Mögliche Lösungsansätze
  - Anpassung von Adressen relativ zur Startspeicheradresse
  - Auslagern des Speicherabbilds von inaktiven Prozessen auf die Festplatte

Keine Abstraktion - Beispiel

Prozesse	Speicheradressen
Prozess 2	
JMP 8	1016
	1012
	1008
	1004
	1000
	...
	...
Prozess 1	
	20
JMP 8	16
	12
	8
	4
	0

Adressräume

- Jeder Prozess hat seinen eigenen Adressraum
  - d.h. jedem Prozess wird eine Menge an Adressen zugewiesen, die dieser verwenden darf
- Dynamic Relocation
  - Base Register ist die Startadresse des Programms
  - Limit Register ist die Programmlänge
  - Greift das Programm auf Speicheradressen zu, wird die Adresse mit dem Base Register addiert und überprüft, ob die Endadresse des Programms nicht überschritten wird
  - Nachteil: Zusätzliche Operationen (Addition + Vergleich) bei jedem Speicherzugriff

Dynamic Relocation - Beispiel

Prozesse	Speicheradressen
Prozess 2	
JMP 8	1016
	1012
	1008
	1004
	1000
	...
	...
Prozess 1	
	20
JMP 8	16
	12
	8
	4
	0

1000 + 8 &&
1008 <= 1016
→ JMP 1008

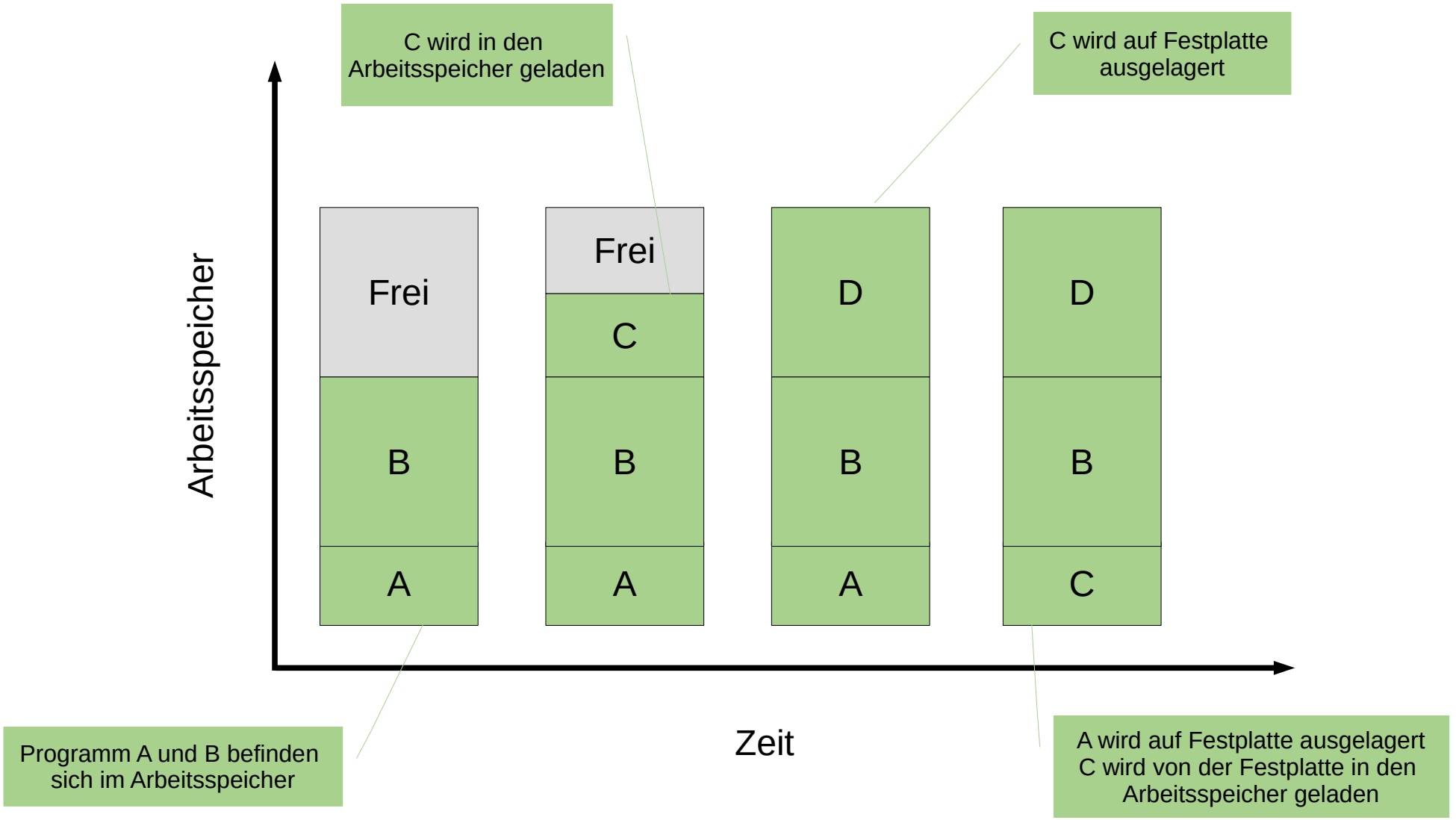
Limit Register

Base Register

Swapping

- Es ist unter Umständen nicht möglich, alle Prozesse im Arbeitsspeicher zu halten
  - Wenn viele Prozesse gleichzeitig laufen und/oder Prozesse große Mengen an Arbeitsspeicher benötigen
- Swapping bedeutet,
  - den Speicher des kompletten Prozesses auf die Festplatte auszulagern, wenn dieser nicht läuft
  - den Speicher des kompletten Prozesses wieder von der Festplatte in den Arbeitsspeicher zu laden, wenn der Prozess fortgesetzt wird
- Damit wird Arbeitsspeicher für andere laufende Prozesse frei

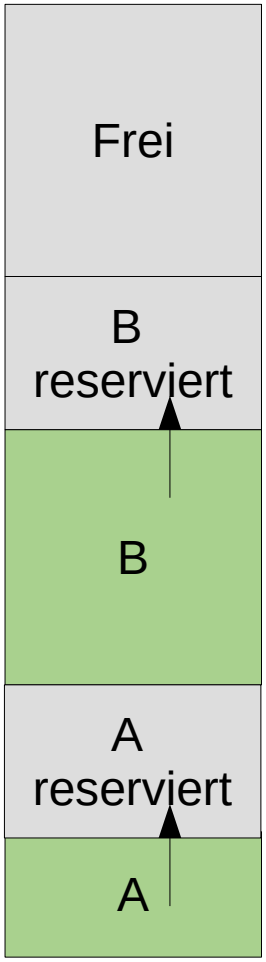
Swapping - Beispiel



malloc anyone?

- Bisher haben wir einem Programm eine fixe Größe im Arbeitsspeicher zugewiesen
- Es muss aber auch dynamische Speicherallokierung beachtet werden
  - Hat einen Einfluss auf die Aufteilung des Arbeitsspeichers
  - Anders gesagt: Das Datensegment eines Prozesses kann zur Laufzeit wachsen und schrumpfen
- Wenn also ein Prozess über seine Grenzen hinauswachsen würde, muss er im Speicher umgesiedelt werden
  - Schlecht für die Performance
  - Daher ist es eine bessere Idee, Lücken zwischen den Speicherbereichen von Prozessen zu schaffen
  - Beim Swappen wird aber nur der tatsächlich belegte Teil des Speichers auf die Festplatte geschrieben

Swapping mit Lücken



- Wenn reservierter Speicher ausgeht:
  - Relocation in einen Speicherbereich der groß genug ist
  - Swapping und Pausieren, bis dass ein größerer Speicherbereich frei ist
  - Beenden (Out Of Memory Exception)

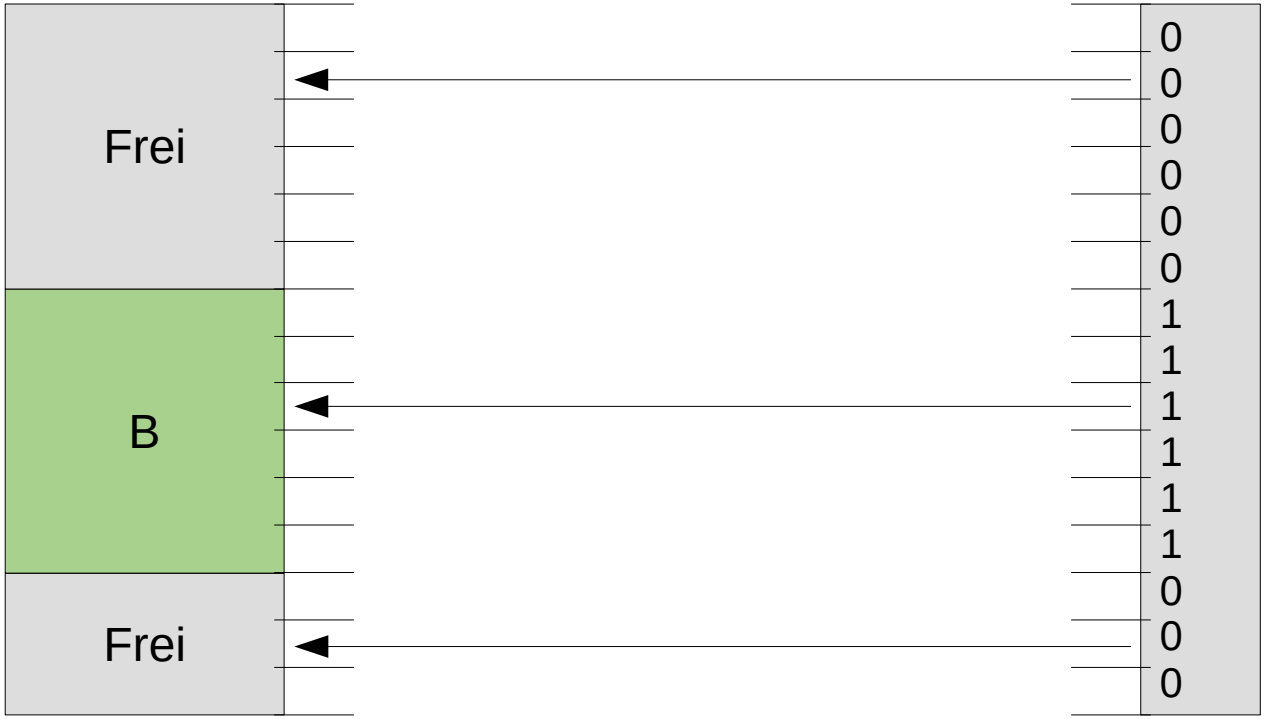
Keeping Track of Memory

- Wie wir im Swapping Beispiel gesehen haben, trifft das OS Entscheidungen, an welche Stellen im Arbeitsspeicher Programme geladen werden
- Voraussetzung dafür ist, dass das OS weiß,
  - wo wie viel Speicher frei ist
  - wie viel Speicher als Buffer (Lücke) freigehalten werden sollte
- Wir sehen uns im Folgenden zwei Strategien zum Verwalten der Speicherbelegung an
  - Bitmaps
  - Verkettete Listen

Bitmaps

- Jedes Bit in einer Bitmap ist einer Allocation Unit zugeordnet
  - Allocation Unit: Adressierbare Speichereinheit
  - 0 bedeutet Allocation Unit ist frei
  - 1 bedeutet Allocation Unit ist belegt
- Größe einer Allocation Unit hat einen Einfluss auf die Größe der Bitmap
  - Umso kleiner die Allocation Unit, umso größer wird die Bitmap
  - Bsp: AU 4 Bytes, d.h. 1 Bit pro 4 Bytes, bei 2 GB RAM würde also die Bitmap 64 MB groß sein

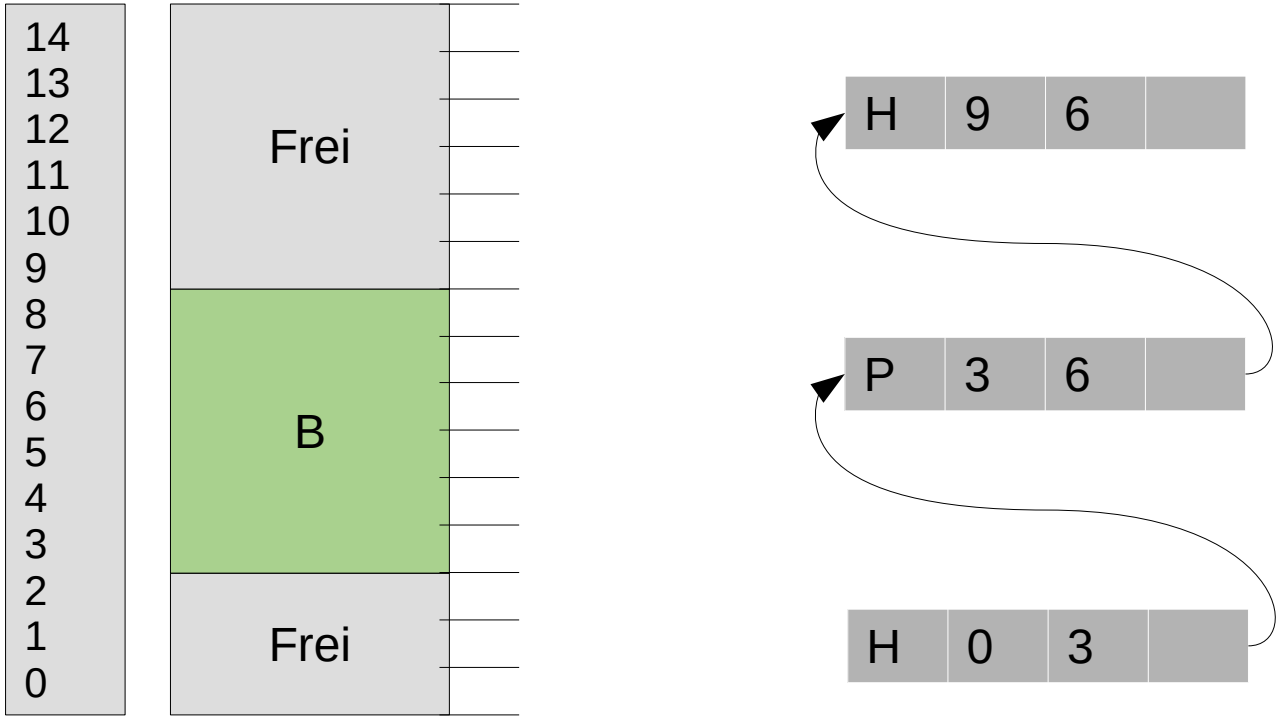
Bitmaps - Beispiel



Verkettete Listen

- Anstelle einer Bitmap wird eine verkettete Liste mit 2 Knotentypen (Process, Hole) verwendet
- Knoten speichern
  - Typ
    - P – Process: Speicherbelegung eines Prozesses
    - H – Hole: Block aus freiem Speicher
  - Startadresse
  - Länge
- Länge der Liste und der Speicherbedarf der Liste hängt damit von der Anzahl an Prozessen und freien Speicherblöcken ab
  - Im Gegensatz zur Bitmap, die eine konstante Größe (abhängig von der Größe des Arbeitsspeichers) hat

Verkettete Listen - Beispiel



Verkettete Listen - Varianten

- Wird ein neuer Prozess in den Speicher geladen, muss entschieden werden, welcher freie Platz verwendet werden soll
  - First Fit: Durchlaufen der Liste und Verwenden des ersten passenden Platzes
  - Best Fit: Durchlauf der Liste und Verwenden des kleinsten freien Platzes
  - Separate Listen für Prozesse und Holes als Optimierung
- Jede Variante hat Auswirkung auf Performance und die Menge an verschwendetem Arbeitsspeicher
  - Best Fit tendiert z.B. überraschender Weise dazu, kleine Speicherlöcher zu produzieren, ist aber schneller beim Speicherzuweisen als First Fit

Virtual Memory

- Jeder Prozess hat seinen eigenen Adressraum
- Der Adressraum ist unterteilt in Seiten (Pages)
  - Eine Page ist eine aufsteigende Folge von Adressen
  - Pages werden auf den physischen Speicher gemappt
- Während ein Prozess ausgeführt wird, müssen sich nicht alle Pages des Prozesses im Hauptspeicher befinden
- Hardware führt das Adressmapping durch, wenn sich eine referenzierte Speicheradresse schon im physischen Speicher befindet (siehe Base/Limit Register)
- Befindet sich die referenzierte Speicheradresse nicht im Hauptspeicher, wird das OS benachrichtigt (Page Fault, wird vom Prozessor (MMU) ausgelöst) und lädt die Page, die die Adresse enthält
- Vorteil: Es kann eine größere Anzahl an Prozessen gleichzeitig im Speicher gehalten werden
  - Ist z.B. das Laden einer Page für einen Prozess notwendig, kann in der Zwischenzeit ein anderer, wartender Prozess ausgeführt werden

Virtual Memory

- Programme referenzieren virtuelle Adressen (vom Compiler generiert) in einem virtuellen Adressraum
- Die MMU (Memory Management Unit, Teil des Prozessors)
  - Mappt die virtuelle Adresse auf die physische Adresse
  - Stellt fest, ob die Page, die eine virtuelle Adresse enthält, sich im Hauptspeicher befindet (ob die Page einen korrespondierenden Page Frame hat)
  - Löst einen Page Fault aus, wenn sich die Adresse nicht im Hauptspeicher befindet
- Paging ist
  - das Laden von benötigten Pages von der Festplatte in den Arbeitsspeicher
  - das Auslagern von nicht benötigten Pages aus dem Arbeitsspeicher auf die Festplatte
  - Aufgabe des OS ist es, dass
    - benötigte Pages nachgeladen (eingelagert) werden
    - Page Frames auf die Festplatte ausgelagert werden, um Arbeitsspeicher freizugeben

Virtual Memory - Beispiel

Page (Größe: 2 kB)

Present/Absent Bit

Virtueller Adress-raum	Mapping
16	X
14	X
12	1
10	2
8	X
6	X
4	X
2	3
0	0

Page Frame	Physische Speicher-adresse
	6
	4
	2
	0

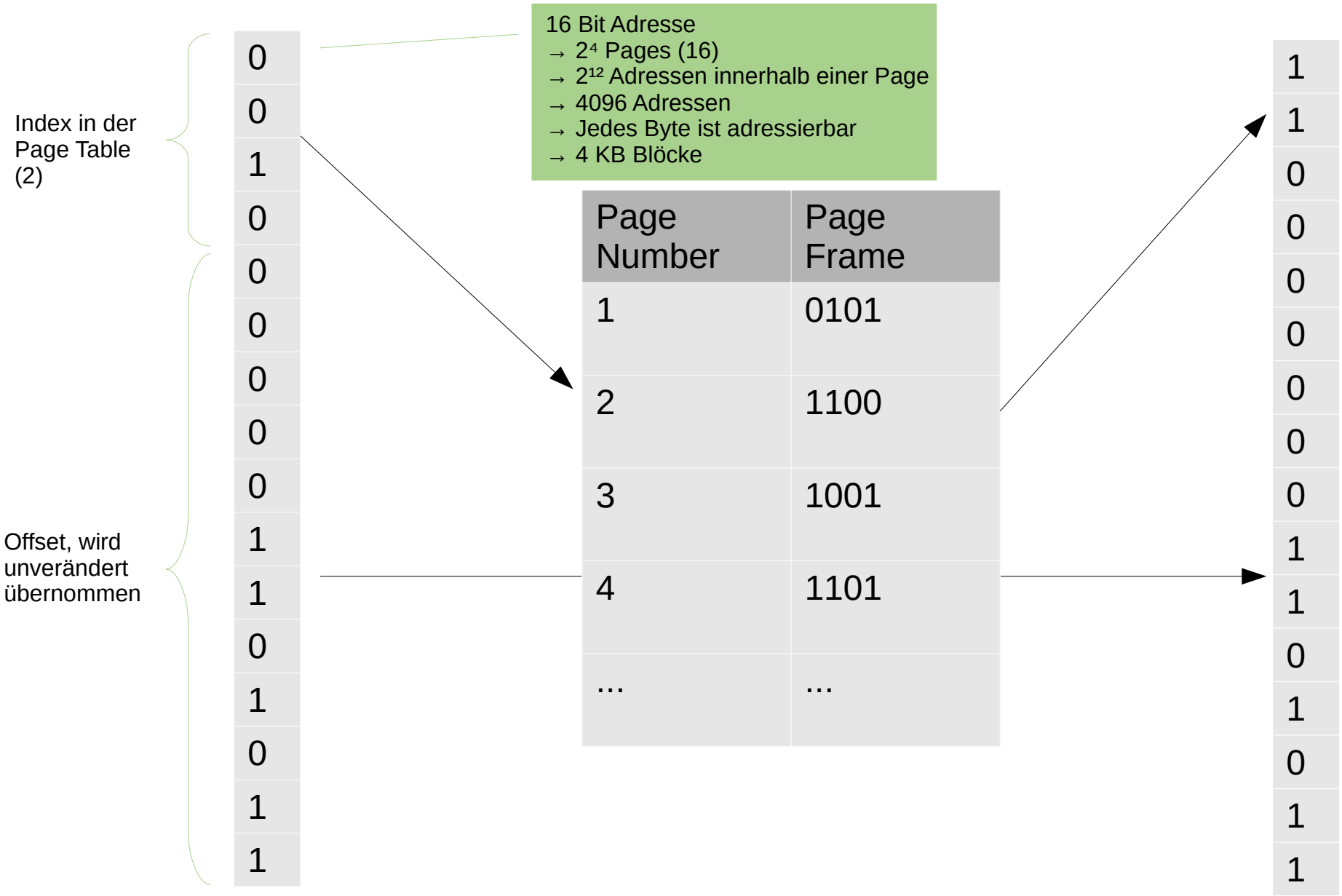
Größe: 2 kB
Page Größe == Page Frame Größe

Physischer Speicher < virtueller Adressraum

Page Table

- Page Table enthält
  - das Mapping zwischen Page und Page Frame (virtuellen und physischen Adresse)
  - Zusatzinformationen (z.B. Present/Absent Bit,...)
- Adresse besteht aus 2 Teilen
  - Page Number: Wird in der Page Table als Index für den Eintrag der Page Frame Nummer verwendet
  - Offset: Für die Adressierung innerhalb des Frames, wird unverändert übernommen

MMU – Vereinfachtes Beispiel



Page Table Einträge

- Tatsächlich enthält ein Eintrag in der Page Table mehr Information
  - Page Frame Number
  - Present/Absent Bit:
    - 0: Page ist nicht gemappt → Page Fault
    - 1: Page ist gemappt und es kann zugegriffen werden
  - Protection: Lese- und oder Schreibzugriff erlaubt
  - Modified: Wurde in die Page geschrieben, wird das Modified Bit gesetzt. Wurde die Page verändert, muss sie beim Freigeben auf die Platte zurückgeschrieben werden. Andernfalls ist keine Aktion des OS nötig
  - Referenced: Wird gesetzt, wenn eine Adresse von einer Aktion referenziert wird. Damit weiß das OS, dass diese Page ein schlechter Kandidat zum Freigeben ist, da sie noch, oder bald, verwendet wird
- Aber keine Information zu z.B. Speicherort auf der Festplatte, da dies Aufgabe des OS ist

Performance Aspekte

- Auflösen von virtuellen (auf physische) Adressen muss schnell sein
  - Wird bei jedem Speicherzugriff benötigt
  - Sollte daher HW-unterstützt werden, da schnell
- Virtueller Adressraum ist groß
  - Die Page Tables können sehr groß werden
    - Mit 32 Bit können ~1 Million Pages (2^{20}) adressiert werden
    - Mit 64 Bit können sehr sehr viele Pages adressiert werden
  - Nicht wirtschaftlich, HW Register für komplette Page Tables anzubieten
- Lösung ist ein Kompromiss: Translation Lookaside Buffer

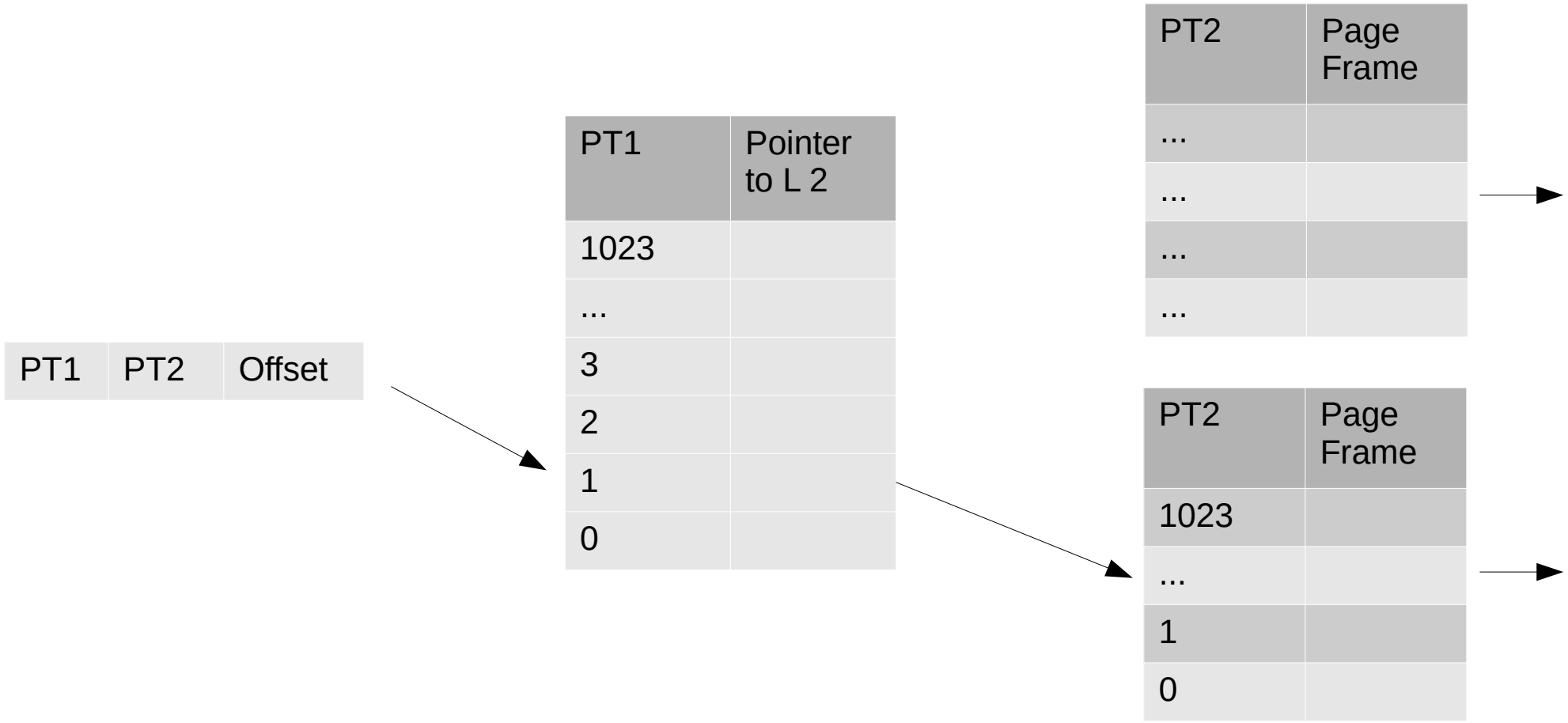
Translation Lookaside Buffer

- Technik zur Beschleunigung von Page Lookups
- Page Table ist im Arbeitsspeicher, TLB ist Teil der MMU und in HW implementiert
- Idee: Die meisten Programme greifen nur auf eine kleine Anzahl an Pages oft zu
- TLB ist Teil der MMU und ist eine kleine Mapping Tabelle, die den Umweg über die Page Table und damit den Arbeitsspeicher vermeidet
- Wird beim Anfordern einer Page das Mapping nicht im TLB gefunden, wird der Eintrag aus der Page Table geladen und der TLB aktualisiert

Multilevel Page Tables

- Technik für große virtuelle Adressräume
- Es müssen damit nicht immer alle Page Tables im Speicher gehalten werden
  - z.B. jene Page Tables für Bufferblöcke eines Programms, die für wachsenden Speicher reserviert sind, müssen nicht im Speicher gehalten werden
- Beispiel 32 Bit Adressen
  - Ersten 10 Bit für die Level 1 Page Table
    - Referenziert 4 MB Blöcke (→ 4 GB adressierbarer Speicher in 32 Bit Architekturen)
  - Nächsten 10 Bit für die Level 2 Page Table
  - Verbleibenden 12 Bit für den Offset
- z.B.
  - x86 (32 Bit): 2^{10} x 2^{10} x 2^{12} → 4294967296 Bytes → 4 GB
  - x68_64, 4 Levels à 512 Byte → 2^{9} x 2^{9} x 2^{9} x 2^{9} x 2^{12} → 2^{48} → 256 TB

Multilevel Page Tables



Inverted Page Tables

- „Inverted“, weil die Tabelle ein Mapping von Frame auf Page ist
  - Eine Zeile der Tabelle entspricht einem Frame des Arbeitsspeichers und der zugeordneten Page
- Damit ist die Größe der Tabelle konstant
- Problem: Suche nach virtuellen Adressen ist langsam
- Lösung
  - Verwendung des TLB
  - Hashing der virtuellen Adressen um Suchen zu beschleunigen
- z.B. Itanium, UltraSPARC, PowerPC

Ersetzungsstrategien

- Wird eine Page benötigt, die sich nicht im Arbeitsspeicher befindet
  - Muss dass OS diese von der Festplatte laden
  - Eventuell eine Page aus dem Speicher entfernen
  - Die entfernte Page eventuell auf die Festplatte schreiben, wenn diese verändert wurde
    - Oder keine Aktion durchführen, wenn der Eintrag auf der Festplatte aktuell ist weil die Page nicht verändert wurde
- Der Page Replacement Algorithmus führt die Auswahl einer geeigneten Page zum Ersetzen mit der nachzuladenden Page durch
  - Im Optimalfall wird jene Page, die zeitlich möglichst weit entfernt verwendet wird, ersetzt
  - Das Problem ist, dass diese Information nicht verfügbar ist

Not recently used (NRU)

- Voraussetzung: Zu jeder Page in der Page Table wird folgende Information gespeichert
  - Referenced (R Bit): Wird gesetzt, wenn eine Page gelesen oder verändert wird
  - Modified (M Bit): Wird gesetzt, wenn eine Page verändert wird
- Wird im Normalfall von der Hardware aktuell gehalten (auch Verwaltung durch OS möglich)
- Zu bestimmten Zeitpunkten (z.B. Timer Interrupt) wird das R Bit zurückgesetzt
  - Länger nicht verwendete Pages werden damit als nicht referenziert markiert

Not recently used (NRU)

- Daraus ergeben sich folgende Klassen
  - Klasse 0: nicht referenziert, nicht modifiziert
  - Klasse 1: nicht referenziert, modifiziert
  - Klasse 2: referenziert, nicht modifiziert
  - Klasse 3: referenziert, modifiziert
- NRU entfernt zufällig Seiten, beginnend mit der niedrigsten, nicht leeren Klasse
  - Besser nicht referenzierte Seiten zu entfernen, auch wenn diese modifiziert sind
- Vorteil: Einfach zu implementieren, für viele Fälle ausreichend

FIFO / Second Chance

- Pages werden in eine verkettete Liste eingefügt
  - Bei einem Seitenfehler wird der älteste Eintrag einfach entfernt
  - Nachteil: Eintrag könnte noch benötigt werden → untauglich in der Praxis
- Second Chance Algorithmus
  - Es wird zusätzlich das R Bit verwendet, um zu überprüfen, ob die Seite noch referenziert ist
  - Ist das der Fall, wird die Page übersprungen, das R Bit gelöscht und die Page an das Ende der Liste gestellt
  - Danach wird der nächsten Eintrag in der FIFO Liste überprüft
  - Sind alle R Bits gesetzt, wird aus Second Chance FIFO und der älteste Eintrag entfernt
  - Vorteil: Vernünftiger Algorithmus, praktische Implementierungen vermeiden Umhängen von Knoten in der Liste aus Performance Gründen (Clock Algorithmus)

Least Recently Used (LRU)

- Idee: Eine Seite, die in letzter Zeit häufig benutzt wurde, wird wahrscheinlich auch in Zukunft häufig benutzt
- Benötigt wird eine Liste aus allen Pages, geordnet nach Benutzungshäufigkeit
- Problem dabei: Aufwändig zu implementieren bzw. langsam
  - Liste müsste bei jedem Speicherzugriff aktualisiert werden
  - Benötigt für jede Page einen Zähler (Counter) und dieser muss auch bei jedem Zugriff aktualisiert werden
- Es würde also zumindest HW Unterstützung benötigt werden, um dies effizient umsetzen zu können → selbst dann nicht praxistauglich
- Daher Annäherung über Algorithmus Not Frequently Used (NFU)

Not Frequently Used (NFU)

- Für jede Page gibt es einen (Software) Zähler
  - Ist zu Beginn 0
  - Bei einem Timer Interrupt werden alle Seiten durchlaufen und bei jenen, die das R Bit gesetzt haben, der Zähler inkrementiert
  - Die Seite mit dem niedrigsten Zähler wird zur Ersetzung ausgewählt
- Nebeneffekt: Wurde auf eine Seite oft zugegriffen, wird diese unter Umständen sehr lange im Speicher gehalten obwohl sie nicht mehr benötigt wird → zeitliche Komponente fehlt
- Erweiterung
  - Zähler werden 1 Bit nach rechts geschoben
  - Inkrementieren erfolgt durch Addieren des R Bits zum höchstwertigen (linken) Bit des Zählers
  - Wird auch als „Aging“ bezeichnet
- Dadurch „vergessen“ Zähler ihre Werte nach einer bestimmten Zeit

Aging Beispiel

Zeit	R-Bit Page 0	R-Bit Page 1	Page 0	Page 1
0	1	0	10000000	00000100
1	1	0	11000000	00000010
2	1	0	11100000	00000001
3	1	0	11110000	00000000
4	0	1	01111000	10000000
5	0	1	00111100	11000000
6	0	0	00011110	01100000

Working Set Page Replacement Algorithmus

- Die Menge an Pages, die ein Prozess zu einem bestimmten Zeipunkt verwendet, wird als Working Set bezeichnet
  - Ist eine Teilmenge aller Pages, die ein Prozess über seine Lebensdauer verwendet
  - Prozesse verwenden meist nur einen kleinen Teil der theoretisch benötigten Pages
  - Working Set ändert sich über die Lebensdauer des Prozesses
  - Sind alle theoretisch benötigten Pages im Speicher, gibt es keine Seitenfehler
  - Trashing: Muss zwischen Instruktion eingelagert werden, weil Pages nicht im Speicher vorhanden sind, dauert das Einlagern länger als das tatsächliche Ausführen der Instruktion → schlecht für Performance
- Idee
  - OS merkt sich das Working Set eines Prozesses
  - Stellt sicher, dass das Working Set eingelagert ist, bevor ein Prozess durch den Scheduler fortgesetzt wird

Working Set Page Replacement Algorithmus

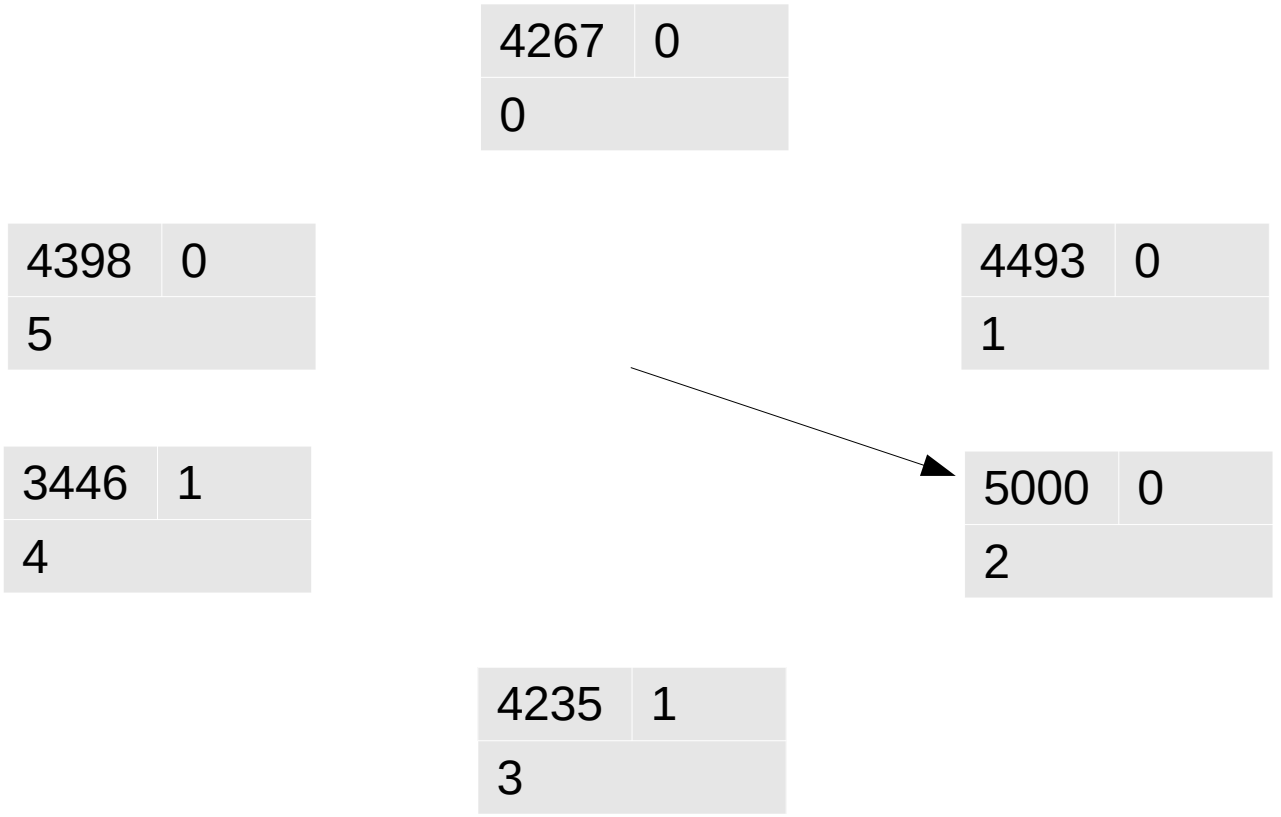
- Working Set: Menge an Pages, die in den letzten k Speicherzugriffen benutzt wurden
  - Protokollieren dieser Information würde Performance negativ beeinflussen
- Daher Vereinfachung: Menge an Pages, die in den letzten t Zeiteinheiten benutzt wurden
- Beim Ersetzen von Pages werden immer jene ersetzt, die nicht zum Working Set zählen
- Es wird zusätzlich zu den R/M Bits der Zeitpunkt des letzten Zugriffs (Alter) gespeichert
- Wir nehmen auch wieder an, dass das R Bit periodisch zurückgesetzt wird
- Seitenauslagerung wenn
  - R == 0 ist und
  - Alter > t
- Wenn R == 0 und Alter <= t wird Alter mit höchstem Alter ersetzt
- Tabelle wird immer komplett durchlaufen und das Alter aller Einträge aktualisiert

WSClock

- Problem des Working Set Page Replacement Algorithmus ist das Durchlaufen und Aktualisieren der kompletten Page Table bei einem Seitenfehler
- Daher: Kombination des Clock Algorithmus mit dem Working Set Page Replacement Algorithmus
- Datenstruktur ist eine ringförmige Liste (→ „Clock“)
  - Pages werden diesem Ring, wenn benötigt, hinzugefügt
  - Zusätzlich enthalten die Einträge den Zeitstempel der letzten Verwendung (und R / M Bit)
  - Es gibt einen Zeiger, der immer auf die älteste Page zeigt
    - Bei ihr wird bei einem Page Fault mit der Suche nach zu ersetzenden Pages begonnen
    - Ist R == 0 && M == 0 && Alter > t wird
      - die Page entfernt
      - die neue Page an diese Stelle geladen
      - Der Zeiger weiterbewegt
    - Ist R == 0 && M == 1 wird
      - Aus Performance Gründen das Schreiben der Page eingeplant
      - Der Zeiger aber weiterbewegt und die Suche fortgesetzt
    - Ist R == 1, wird R auf 0 gesetzt und der Zeiger weiterbewegt

WSClock - Beispiel

- Aktuelle Zeit: 5100
- t: 200
- Zeiger steht auf Page 2
  - R Bit ok, Alter nicht ok
- Zeiger wird auf Page 3 gestellt
  - R Bit nicht ok, Alter ok
  - Reset R Bit
- Zeiger wird auf Page 4 gestellt
  - Gleich zu Page 3
- Zeiger wird auf Page 5 gestellt
  - R Bit ok, Alter ok
  - Page wird ersetzt, vorausgesetzt M == 0

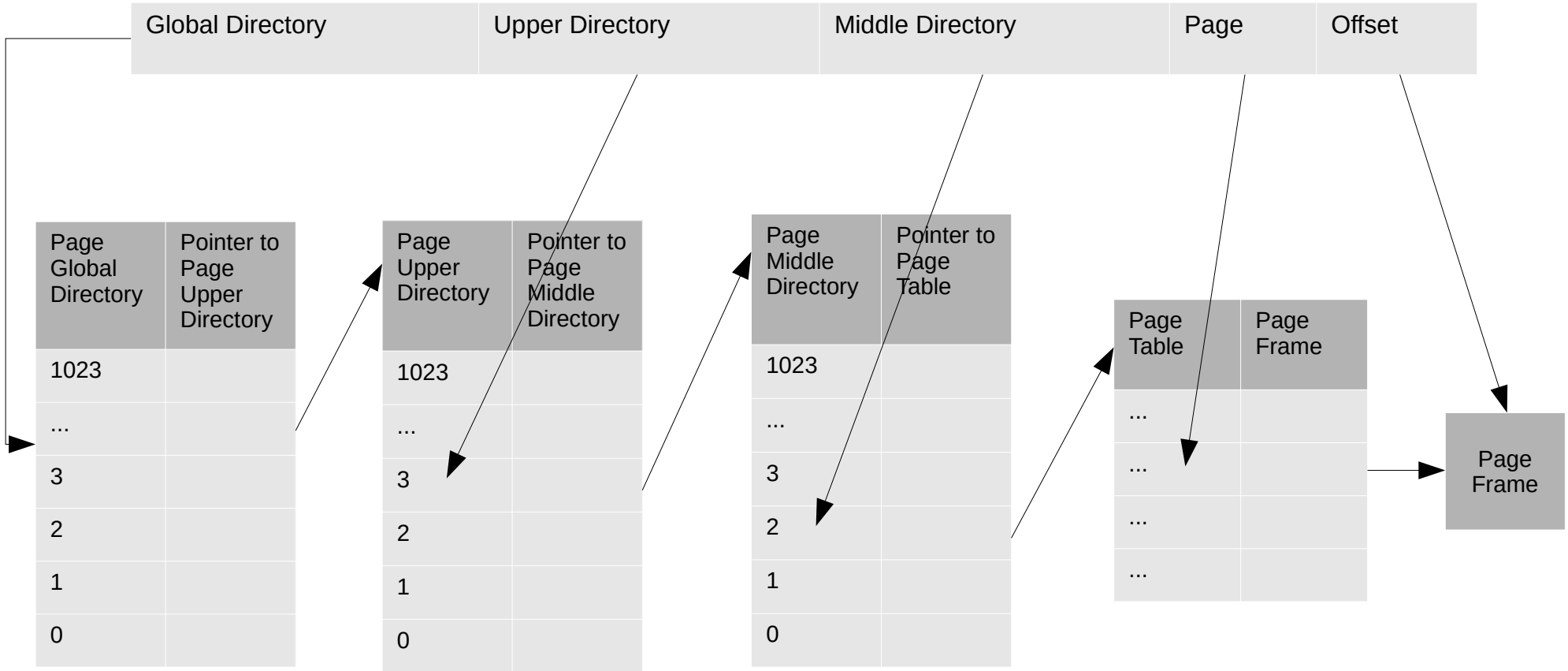


Implementierung - Linux

- Linux verwaltet Speicher mit einem Array bestehenden aus Page Deskriptoren (32 Byte → benötigt < 1% des Speichers)
  - Mapping Page → Page Frame
  - Enthält Zeiger zum Adressraum
  - Zeiger zu anderen Deskriptoren → Verlinkte Liste für z.B. freie Blöcke
- Speicher ist in Zonen unterteilt
  - Werden in einer Liste verwaltet (DMA, Normal, HighMem)
  - Enthält Informationen
    - Über Belegung, freie Bereiche
    - Active/Inactive Pages
    - Watermarks für Page Replacement Algorithmus
- Zusätzlich wird der Speicher in Nodes unterteilt
  - Für NUMA (Non-Uniform Memory Access) Systeme
  - Bei mehreren CPUs kann der Zugriff auf einen bestimmten Speicher je nach CPU unterschiedlich lange dauern
    - Muss beim Allokieren von Speicher beachtet werden
    - Es sollte nach Möglichkeit der schnellste Speicher der CPU, an die ein Prozess (CPU Affinity) gebunden ist, ausgewählt werden

Linux – Multilevel Page Tables

- Linux verwendet Multilevel Page Tabels mit 4 Ebenen (Levels)
- Update auf 5 Level für 64 Bit Systeme möglich



Linux - Allocators

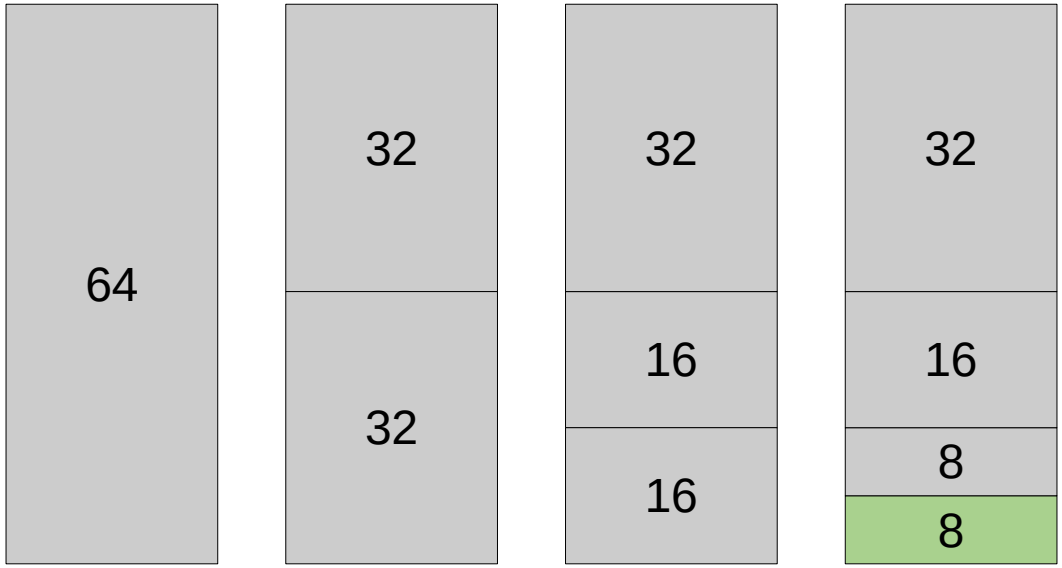
- Bis jetzt haben wir komplett ignoriert, welche Strategie ein OS bei der Zuteilung (Allokierung) von Arbeitsspeicher zu Prozessen verfolgen kann
  - Neu gestartete Prozesse müssen in den Speicher geladen werden (→ `fork()`)
  - Prozesse benötigen über ihre Lebensdauer unterschiedlich große Blöcke an Speicher (→ `malloc()`)
  - Prozesse geben unterschiedliche große Blöcke an Speicher über ihre Lebensdauer frei (→ `free()`)
- Linux verwendet einen leicht modifizierten Buddy Algorithmus in Kombination mit Slab Allocation

Exkurs: Allokierung und Fragmentierung

- Nehmen wir an, wir stellen Speicher in 4 KB Blöcken zur Verfügung
  - Was wenn wir nur 4 Byte allokieren?
    - Verschwendung → Interne Fragmentierung
  - Was wenn wir 2 MB allokieren
    - Dann benötigen wir 512 4 KB Blöcke
    - Nehmen wir an, dass es sich um das Working Set eines Prozesses handelt und der TLB 64 Adressen speichern kann → Performance Problem → TLB Trashing
  - Über die Zeit wird Speicher freigegeben, dadurch entstehen Lücken und eine Durchmischung aus belegtem und freiem Speicher
    - z.B. sind noch viele kleine, nicht durchgängige Blöcke an Speicher frei, können diese aber bei einer Anforderung nach einem großen Block nicht mehr verwendet werden, d.h. theoretisch wäre noch Speicher verfügbar, dieser ist aber nicht mehr verwendbar → Externe Fragmentierung
    - D.h. es müsste der Speicher neu angeordnet werden und die freien Blöcke zu einem Bereich zusammengefasst werden (Compaction)
- Es sollte also möglich sein, Blöcke mit variabler Größe zu allokieren
  - x86 CPUs und Linux unterstützen zusätzlich variable Page Größen

Linux - Buddy Algorithms

- Speicherblock, bestehend aus Pages, wird immer weiter unterteilt, bis die benötigte Blockgröße erreicht ist
- Bei Speicherfreigaben werden Blöcke wieder vereint
- Erweiterung Linux: Zusätzliche Listen mit unterschiedlichen Größen (1-Page Liste, 2-Page Liste, 4-Page Liste,…)
- Beispiel:
  - 64 Page Block
  - 8-Page Block wird benötigt
  - Was passiert wenn ein 16-Page Block benötigt wird?
  - Was passiert wenn danach der 8-Page Block freigegeben wird?
  - Was passiert wenn ein 33-Page Block benötigt wird?



Linux - Slab Allocation

- Buddy Algorithmus vermeidet externe Fragmentierung (leere Speicherlücken zwischen allokiertem Speicher), leidet jedoch unter interner Fragmentierung
- Zusätzlicher Slab Allocation Algorithmus reserviert mit dem Buddy Algorithmus Blöcke, welche für Objekt Caches verwendet werden
  - 1 Slab kann mehrere Objekte des selben Typs speichern, z.B. File Deskriptoren, Prozess Deskriptoren,...
  - Wird z.B. eine Datei geöffnet, wird aus einem Slab ein bereits allokierter (aber nicht benutzter) File Deskriptor verwendet
- Slab Allocation reduziert daher die Anzahl an Allokierungs- und Freigabe-Operationen für bestimmte Typen von Objekten

Linux – Ersetzungsstrategie

- Linux verwendet kein Preloading von Pages oder Working Sets
- Linux führt Demand Paging durch → es werden nur jene Pages geladen, die auch benötigt werden
- Linux verwaltet einen Pool an freien Pages und hält diesen auf einem vernünftigen Level
- Ein Daemon (kswapd) untersucht periodisch die Watermarks von Zonen und überprüft, ob noch genug freier Speicher in den Zonen vorhanden ist
  - Ist das nicht der Fall, wird der Page Frame Reclaiming Algorithmus ausgeführt
- Eigener Daemon (pdflush) schreibt
  - periodisch alte Pages mit gesetztem w Bit auf die Festplatte
  - Oder schreibt Pages auf die Festplatte, wenn der Arbeitsspeicher knapp wird (Watermark erreicht)

Linux - Page Frame Reclaiming Algorithm

- Jede Page besitzt 2 Bits
  - Active
  - Referenced
- 2 Listen (Ringe)
  - Active Pages
  - Inactive Pages
- Pages werden zwischen den Listen aufgrund ihrer gesetzten Bits verschoben
  - Unterschiedliche Arten von allokiertem Speicher werden unterschiedlich behandelt
  - Es wird eine Mischung aus LRU und Second Chance Clock Algorithmus verwendet
    - Referenced Bit wird bei jedem Durchlauf zurückgesetzt
- kswapd gibt zuerst inactive und unreferenced Pages frei
  - Erst wenn diese Möglichkeit ausgeschöpft ist, versucht er „schwierigere“ Pages freizugeben (z.B. Shared Pages zwischen Prozessen,...)

Zusammenfassung

- Memory Abstraktionen
- Virtual Memory
- Page Tables
- MMU & TLB
- Ersetzungsstrategien
- Linux & Allokierung

Betriebssysteme

OS Architekturen

Susanne Schaller, MMSc
Andreas Scheibenpflug, MSc

SE Bachelor (BB/VZ), 2. Semester

Architekturen

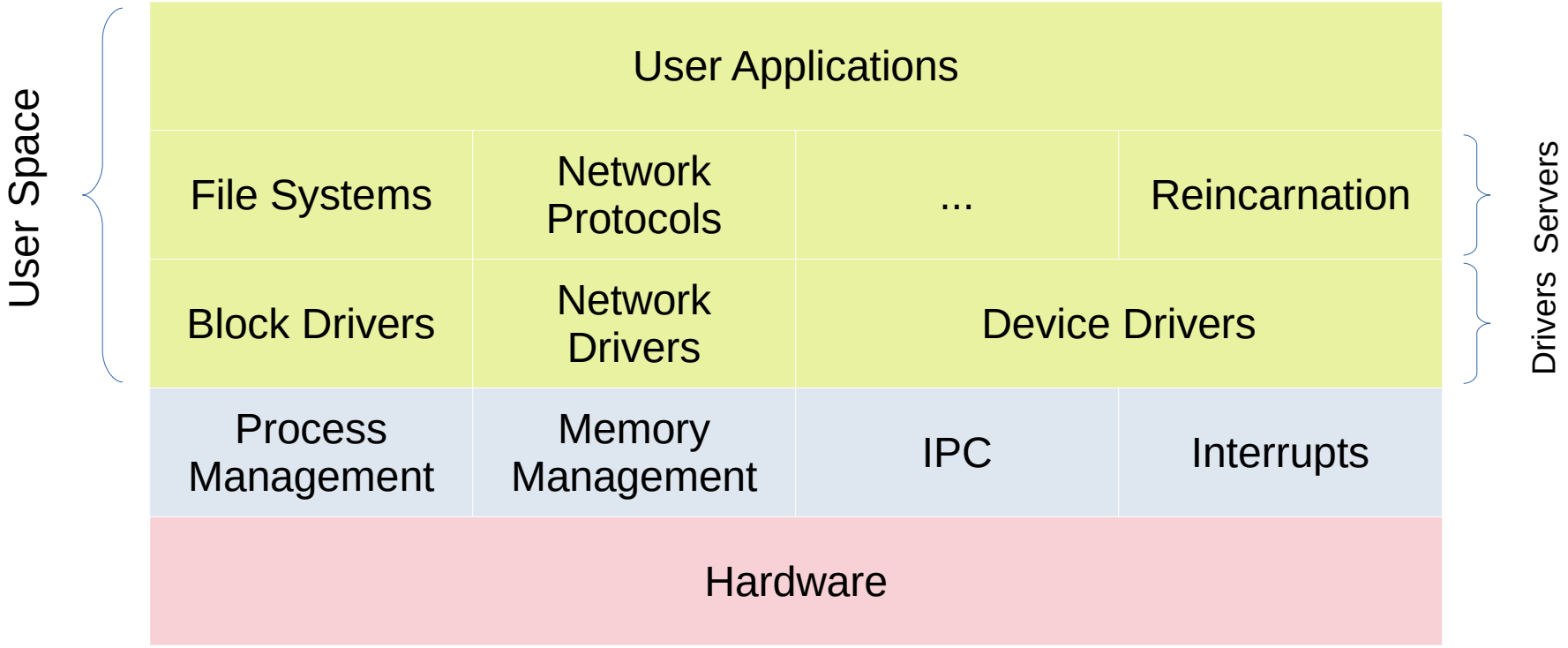
- Bis jetzt haben wir Betriebssysteme als eine große Komponente im Kernel Space (Ring 0) betrachtet
  - Es gibt auch andere Möglichkeiten
- Architekturen
  - Monolith
    - Alle OS Komponenten im Kernel Space
  - Microkernel
    - Nur die minimalen, notwendigsten Komponenten im Kernel Space
    - Alle anderen Komponenten befinden sich im User Space als Service/Server
  - Hybride
    - Mischung aus Monolith und Microkernel
    - Manche Komponenten laufen im User Space
    - Großteil der Komponenten befindet sich im Kernel Space

Monolith - Linux

<table>
<tr><td colspan="5">User Space</td></tr>
<tr><td colspan="5">System Calls / Device Files</td></tr>
<tr><td colspan="3">Virtual File System (VFS)</td><td>Memory Management</td><td>Process Management</td></tr>
<tr><td rowspan="2">Device Drivers</td><td rowspan="3">Sockets</td><td>File Systems (ext4,...)</td><td rowspan="3">Virtual Memory</td><td>Signals / IPC</td></tr>
<tr><td rowspan="2">Network Protocols (TCP/IP,...)</td><td rowspan="2">Process Handling</td></tr>
<tr><td rowspan="2">Interrupts</td><td>I/O Schedulers</td><td rowspan="2">Page Replacement</td><td rowspan="2">Scheduler</td></tr>
<tr><td>Network Device Drivers</td><td>Block Device Drivers</td></tr>
<tr><td colspan="5">Hardware</td></tr>
</table>

- Vorteil
  - Performance
- Nachteil
  - Bugs können Stabilität des ganzen OS gefährden

Microkernel



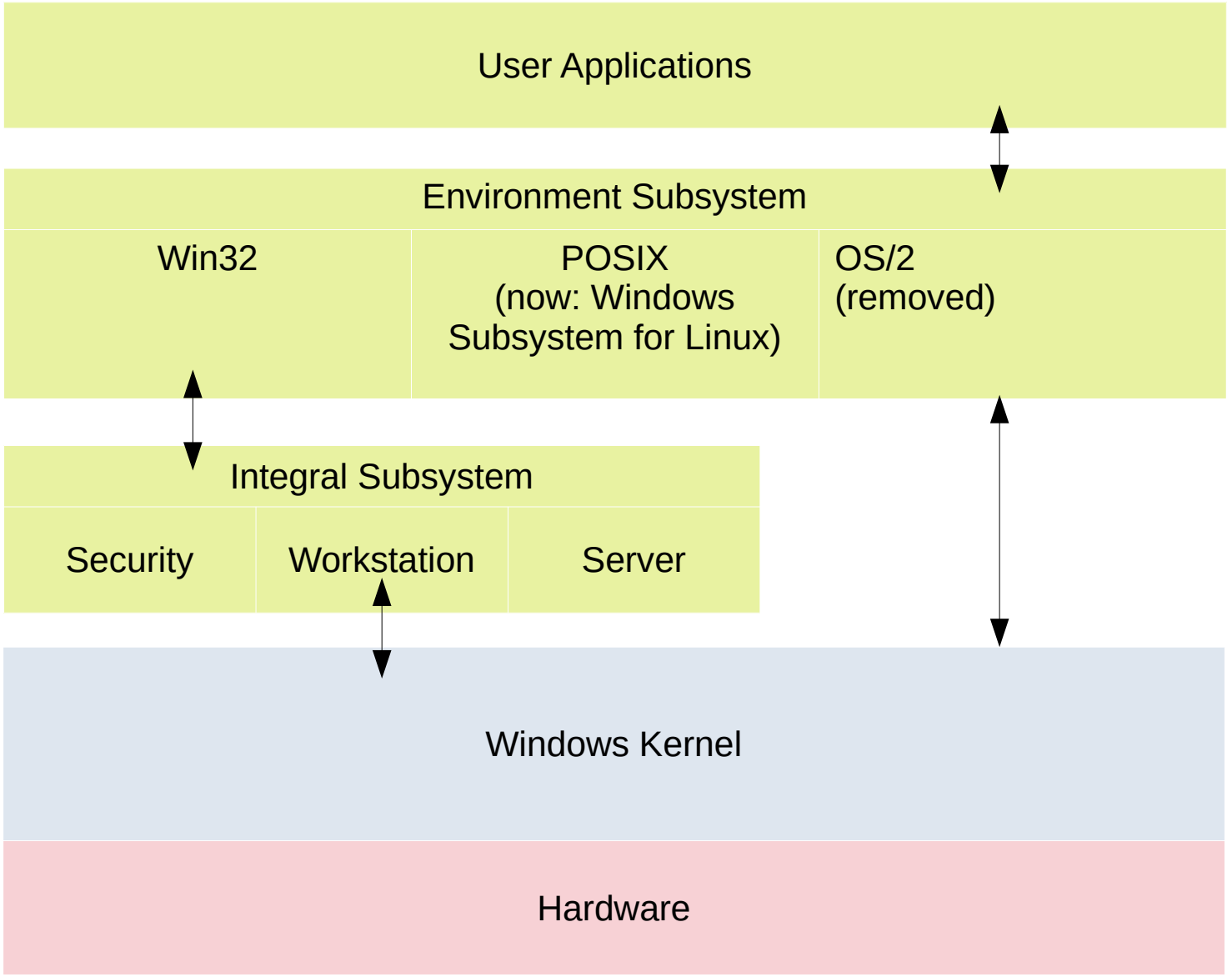
Microkernel

- Im Kernel Space wird nur die notwendigste Funktionalität implementiert
  - Geräte Treiber werden im User Space ausgeführt
  - Andere OS Funktionalität wird durch eigene Prozesse („Server“) im User Space abgebildet
  - Anwendungsprogramme nutzen die Funktionalität der Server
  - Kommunikation meist mit IPC/Message Passing
- Bekanntester Vertreter: Mach
  - Forschungsprojekt
  - Probleme mit Performance aufgrund des IPC Overheads zwischen Servern und OS
  - Mitarbeiter aus dem Projekt arbeiteten später bei Next/Apple und Microsoft

Hybrid

- Mischung aus Monolith und Microkernel Architektur
  - Teile der OS Komponenten werden in den User Space ausgelagert
  - Teilweise werden Gerätetreiber im User Space unterstützt
  - Großteil der Komponenten ist im Kernel Space implementiert
  - Damit werden mögliche Performance Probleme von Microkernel Architekturen abgeschwächt/vermieden
- Bekannte Vertreter
  - Windows >= NT
  - macOS

Windows Architektur



macOS Architektur

- Mit der Rückkehr von Steve Jobs zu Apple und dem Kauf von NeXT Computer wurden viele Technologien und Konzepte für macOS übernommen
  - NeXTSTEP
  - Objective-C
  - OO Anwendungs API (Kits)
  - Gerätetreiber Framework
- macOS basiert auf Mach
  - Implementiert allerdings viele Funktionalitäten im Kernel Space
  - Leiht Network Stack, Sockets, IPC, Filesysteme, div. Bibliotheken von (Free|Net|Open)BSD
  - Stellt mit Driver Kit ein Framework zur Treiberentwicklung im User Space zur Verfügung
- Das Kern OS wird als Darwin bezeichnet
  - Mach + *BSD + I/O Kit + Dateisysteme
- MacOS ist Darwin mit
  - Grafischen Benutzeroberfläche (Aqua)
  - Diverse Bibliotheken/Kits/Services (z.B. Quarz, Metal, Core Audio, Core Image, Foundation, Cocoa,...)

macOS Architektur