

Betriebssysteme Zusammenfassung

Aufgaben von Betriebssystemen:

- Zwischen Hard- und Software
- Effiziente Nutzung von vorhandenen Ressourcen
- Stelle unendlich virtuellen Speicher und unendliche CPUs bereit
- Verhindere Monopolisierung
- Isoliere Adressräume

Systemcalls:

Liegen im gefährlichen Bereich, haben alle Rechte

Dürfen nicht von Anwendungen direkt ausgeführt werden.

Systemcall Wrapper bieten Funktionen, die die Systemcalls aufrufen, liegt im Kernel

Systemcalls sind Software-Interrupts

Werden im privilegierten Modus ausgeführt, nur das Betriebssystem darf dies

Monolith:

Alle Funktionen des Betriebssystems liegen in einem großen Kernel

Jeder SystemCall wird also in diesem Kernel ausgeführt

Erweiterung durch Kernelmodule, die in den Kernel geladen werden

Microkernel:

Jede Funktionalität stellt einen eigenen Microkernel bereit

Anfragen werden u.U. durch alle Kernel geleitet, bis sie verarbeitet werden

Dauert lange, verursacht kalte Caches

Verbesserter Ansatz: Entwicklung in Microkernel, die beim Deploy in Monolith gesteckt werden

Virtuelle Maschinen:

VMM: Virtual Machine Manager (Hypervisor)

Läuft im privilegierten Modus

Gastsystem sendet an VMM, dieser sendet das erwartete Ergebnis zurück

Typ1: Direkt auf Hardware (VMM ist Betriebssystem)

Typ2: Usermode Anwendungen (VirtualBox etc.)

Paravirtualisierung: Angepasste Gast-Systeme arbeiten mit VMM zusammen

Sinnvoll, wenn Prozessor Virtualisierung nicht unterstützt

Basiert auf nativ, Quelltext muss vorliegen/verändert werden

Ausblick: Teilvirtualisierung (Docker/Managed Runtime)

Microsoft Singularity:

Komplett managed Betriebssystem

SIP: Software isolated processes -> Software verwaltet Adressräume

Nebeläufigkeit:

Virutelle Pozessoren: Jeder Thread hat eigenen Stack und CPU Zustand

Kontrollfluss:

Preemptive Scheduling:

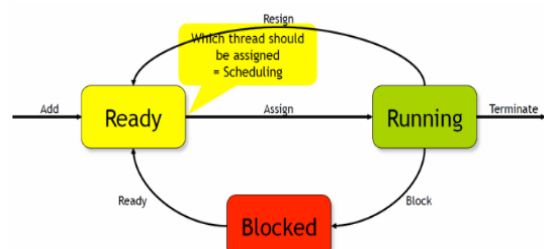
Regelmäßige Kontextwechsel

Keine CPU Monopole möglich

Zusätzliche Kontextwechsel

Non-Preemptive Scheduling:

Nur Kontextwechsel wenn Thread diesen selbst auslöst



CPU Monopol gut möglich

Realtime Berechnung

Competition: Threads wollen CPU behalten, egoistisch

Cooperation: Threads wissen, wann sie warten und geben CPU ab

Semaphoren:

```
public class Buffer
{
    public Buffer ( int n )
    {
        this.n = n;
        slots = new int[n];
        mutex_p = new Semaphore(1);
        mutex_c = new Semaphore(1);
        slots_available = new Semaphore(n);
        goods_available = new Semaphore(0);
    }

    ...

    private int n;
    private int [] slots;
    private int free = 0;
    private int used = 0;
    private Semaphore mutex_p, mutex_c;
    private Semaphore slots_available;
    private Semaphore goods_available;
}

public void Produce ( int good )
{
    slots_available.P();
    mutex_p.P();
    slots[free] = good;
    free = (free+1) % n;
    mutex_p.V();
    goods_available.V();
}

public int Consume ()
{
    goods_available.P();
    mutex_c.P();
    int good = slots[used];
    used = (used+1) % n;
    mutex_c.V();
    slots_available.V();
    return good;
}
```

Shared Data

```
Semaphore Sanctum = new Semaphore(1);
Semaphore RMutex = new Semaphore(1);
Semaphore WMutex = new Semaphore(1);
Semaphore PreferWriter = new Semaphore(1);
Semaphore ReaderQueue = new Semaphore(1);
int readers_inside = 0;
int writers_interested = 0;
```

```
while (true) {
    WMutex.P();
    if (writers_interested == 0)
        PreferWriter.P();
    writers_interested++;
    WMutex.V();
    Sanctum.P();
    // Change data
    Sanctum.V();
    WMutex.P();
    writers_interested--;
    if (writers_interested == 0)
        PreferWriter.V();
    WMutex.V();
}
```

Writer

```
while (true) {
    ReaderQueue.P();
    PreferWriter.P();
    RMutex.P();
    if (readers_inside == 0)
        Sanctum.P();
    readers_inside++;
    RMutex.V();
    PreferWriter.V();
    ReaderQueue.V();
    // Read data
    RMutex.P();
    readers_inside--;
    if (readers_inside == 0)
        Sanctum.V();
    RMutex.V();
}
```

Reader

Barrier:

```
1 # rendezvous
2
3 mutex.wait()
4     count += 1
5     if count == n:
6         turnstile2.wait()    # lock the second
7         turnstile.signal()   # unlock the first
8     mutex.signal()
9
10 turnstile.wait()            # first turnstile
11 turnstile.signal()
12
13 # critical point
14
15 mutex.wait()
16     count -= 1
17     if count == 0:
18         turnstile.wait()    # lock the first
19         turnstile2.signal() # unlock the second
20     mutex.signal()
21
22 turnstile2.wait()           # second turnstile
23 turnstile2.signal()
```

Spinlocks: Busy waiting, bis Freigabevariable den richtigen Wert hat, nur wechselseitig
Semaphore Blockade ist langsamer, Spinlock direkter

TAS: Test and set: gibt den ursprünglichen Wert zurück und aktualisiert Wert atomar

Gang-Threads: Threads die gemeinsam laufen wollen. Wird von aktuellen BS nicht unterstützt

Deadlock: Es geht nicht weiter, alle Beteiligten sind blockiert

Lifelock: Es geht nicht weiter, alle Beteiligten laufen auf 100%

Echtzeitbetriebssysteme:

Benötigen bestimmten Scheduler

Voraussetzung: Alle Anwendungen müssen einen Zeitrahmen angeben, in dem sie erfüllt werden müssen, und wie lange sie im worst case brauchen .

Voraussetzung: Interrupt Zeit und Kontextwechsel müssen nach oben abschätzbar sein
Anwendungen haben Deadline und Ready time (wann angefangen werden darf), sowie maximale Ausführzeit, Phase (Initiales offset/wann das erste mal gerechnet wird), Periode (Frequenz in der diese Aufgabe abgearbeitet werden muss)

Echtzeitanwendungen dürfen nicht blockieren -> nicht mehr abschätzbar

Static/Dynamic Scheduling: Bei Static sind alle Faktoren bekannt

Explicit/Implicit: Implicit (Prioritäten für Anwendungen), Explicit (Definitiver Scheduler Plan der im Voraus schon feststeht)

Preemption: Eine Ausführung kann unterbrochen werden (Achtung: Kontextwechsel kostet)

EDF: Earliest Deadline First: Nicht preemptives EDF ist schlecht, preemptives ist optimal

Rate Monotonic Scheduling: Höhere Frequenz = Höhere Priorität

Je höher die Frequenz, desto kleiner ist die Ausführzeit

ZFS:

Integriertes Volume Management

Alle Speichersysteme in einem Pool verwaltet

Prüfsummen getrennt von Daten

Automatische Selbstheilung bei redundantem Aufbau

Copy-on-Write überschreibt alte Daten nicht

Erlaubt snapshots