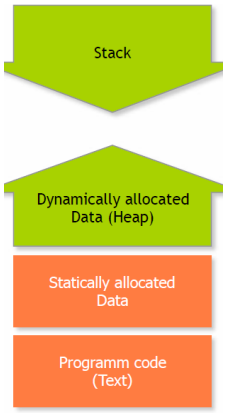
Systemsoftware Zusammenfassung

Adressraum



Die Menge der zur Verfügung gestellten Speicheradressen
Abhängig von Prozessorarchitektur (32bit = 4GB)
Physischer Adressraum: Adressen der Speicherbausteine, EA-Controller, o.Ä.
Sehr "löchrig", da viele Abstände zwischen belegten Adressen
Während der virtuelle Adressraum zusammenhängend ist, können Daten im physischen Adressraum fragmentiert vorliegen.
Virtueller/Logischer Adressraum: Aufbau auf Physischem Adressraum zu Vereinfachung der Nutzung.
Eine Anwendung bekommt einen virtuellen Adressraum
Swapping: Adressräume, die nicht auf den Hauptspeicher passen können auf externe Medien (HDD) ausgelagert werden.

Referenzstring

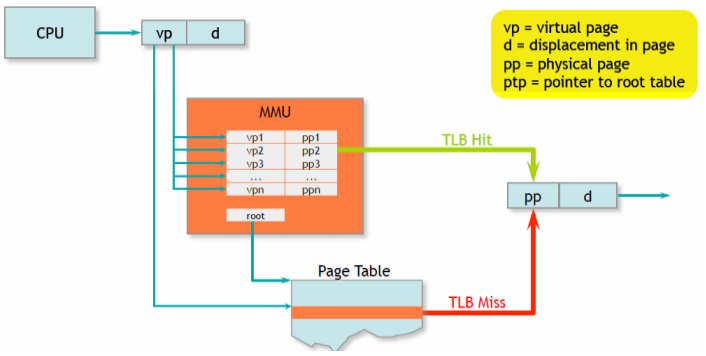
Serie von Adressen, auf die die CPU während der Ausführung zugreift
Enorme Datenmenge, da CPU Milliarden Referenzen pro Sekunden aufrufen kann

Referenzlokalität

Wenn eine Adresse a referenziert wird, ist es sehr wahrscheinlich, dass sie in der nächsten Zeit wieder referenziert wird. (Durch Aufbau von Programmen, z.B. Schleifen).
Wenn eine Adresse a referenziert wird, ist es wahrscheinlich, dass benachbarte Adressen referenziert werden.

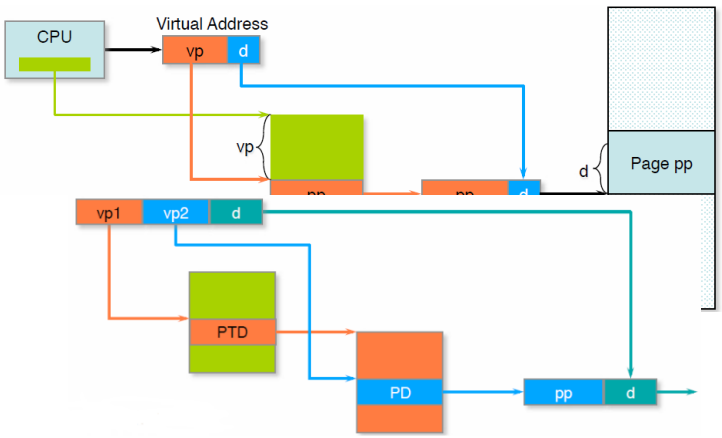
MMU (Memory management unit)

Bildet virtuelle Adressen auf physische ab
Bekommt virtuelle Adressen und benutzt Seitentabellen um auf physische abzubilden.
Cache für Seitentabellen, um den Speicherzugriff nicht weiter zu verlangsamen



Seitenbasierter Adressraum

Der virtuelle Speicher wird mithilfe von Seiten verwaltet.
Jede Seite hat die gleiche Größe und enthält zusätzliche Informationen wie z.B. Zugriffsrechte.
Eine Seite hat üblicherweise eine Größe von 4, 8 oder 16 KB
Die CPU greift auf eine Virtuelle Adresse zu, indem sie eine virtuelle Seite (vp) und ein Offset anfragt. Die MMU wandelt die virtuelle Seite in eine physische Seite (pp) um und hängt das Offset an.



Die Seitentabellen können auch verschachtelt

werden, dazu werden Seitentabellen Deskriptoren (PTD) und Seitendeskriptoren (PD) benötigt. PTDs verweisen auf weitere Seitentabellen, PDs auf physikalische Seiten.

Page Table Descriptor (PTD):

T-Bit: Gibt an ob es PTD oder PD ist

P-Bit: 1= Seite liegt im Hauptspeicher

0= Seite muss von externem Speicher geladen werden

Page Descriptor (PD):

T, P wie PTD

R-Bit: 1 = Seite wurde referenziert bevor R-Bit zurückgesetzt wurde

D-Bit: 1= Seite wurde verändert bevor R-Bit zurückgesetzt wurde

C-Bit: Disable Cache, benötigt für Eingabe-/Ausgabegeräte, diese sollten nicht in den Cache

Access: Zugriffsrechte

Page Fault

Falls eine Seite, die gerade referenziert wird nicht im Hauptspeicher liegt, wird ein Page Fault ausgelöst.

Falls auf eine ungültige Seite zugegriffen wurde beendet das Betriebssystem üblich das Programm.

Falls die Seite auf einem externen Speicher liegt:

Suche freie Seite im RAM (oder lagere eine "alte" aus)

Gebe Auftrag, Seite in den RAM zu laden

Update der Page Table (P=1, neue physikalische Adresse)

Signalisiere CPU die Anweisung zu wiederholen

Seitenverdängungsverfahren

Falls keine freie Seite im RAM muss eine "alte" Seite verdrängt werden.

Referenzstring erstellen:

Großer Aufwand

Großer zusätzlicher Speicherbedarf

Belady-Verfahren:

Diejenige Seite auslagern, die in der Zukunft am längsten nicht mehr referenziert wird.

Nur schwer umsetzbar

Fifo-Verfahren:

Verdränge die Seite, die am längsten im RAM liegt

Einfach zu implementieren

Beachtet Referenzlokalität nicht

Least Recently Used (LRU):

Verdränge die Seite, die am längsten nicht benutzt wurde

Beachtet Referenzlokalität

Beste Annährung an Belady

Schwer umzusetzen

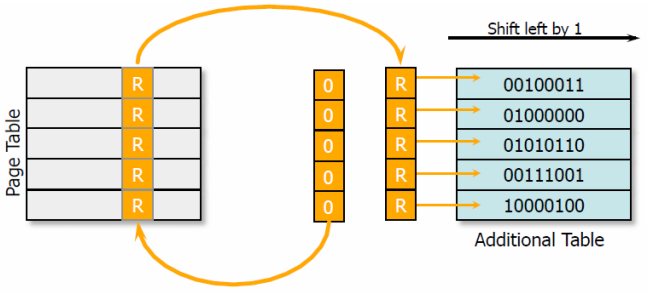
Annährungen an LRU (counter):

Füge jedem Seitendeskriptor einen Zähler hinzu

Erhöhe diesen Zähler immer, wenn eine CPU Clock tickt
Die Seite mit der niedrigsten Zahl fliegt raus

Annährung an LRU (stack):
Verwalte einen Stack der zuletzt referenzierten Seiten
Wird eine Seite referenziert, kommt diese auf den Stack
Das unterste Element wird ausgelagert

Annährung an LRU (Tanenbaum):
Verwalte eine Tabelle in der periodisch das R-Bit der Seiten gespeichert wird
Die Tabelle beinhaltet die R-Bits der letzten n Zeitpunkte
Die Seite mit der kleinsten Zahl wird entfernt



Annährung an LRU (Second Chance):
Verwalte eine FIFO mit den referenzierten Seiten
Durchsuche die FIFO in Richtung abnehmender Einlagerungsdauer
Falls R=1, setze R=0 (Second Chance)
Falls R=0, ersetze Tabelle

Annährung an LRU (Clock Algorithmus):
FIFO teuer zu verwalten und durchsuchen
Benutze zyklische Liste mit zwei Zeigern
Ein Zeiger setzt R-Bit auf 0, ein Zeiger prüft das R-Bit
Beide Zeiger laufen synchron um eins weiter
Abstand der Zeiger bestimmt den Zeitraum für die zweite Chance

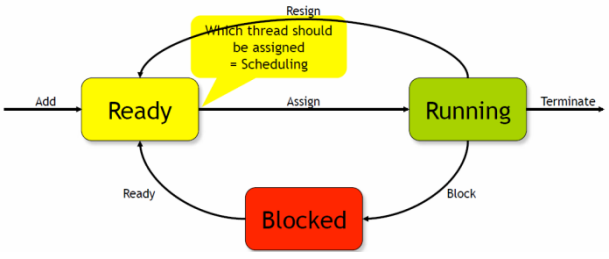
Thrashing

Falls Anwendungen so viel Speicher reservieren, dass nicht mehr alles in den Hauptspeicher passt kann es sein, dass sich die Anwendungen wechselseitig die Pages auslagern. Diese Situation nennt sich Thrashing oder Seitenflattern. Das gesamte System wird dadurch ausgebremst, dass immer wieder Seiten ausgelagert und neu geladen werden.

Um Thrashing zu verhindern:
Zu viele Page Faults:
Erstelle neue Seiten, lagere viele Seiten aus

Threads

Annahme: Es gibt unendlich viele (virtuelle) Prozessoren
Jeder Thread hat einen eigenen Addressraum, Stack und Register
Context Switch (Kontextwechsel): Speichern des Status des aktuellen Threads, laden des Status eines anderen
Kontextwechsel werden periodisch, bei Hardware-Interrupts (Pagefault) oder blockierenden Anweisungen ausgeführt.
CPU burst: Thread läuft aktiv auf der CPU, dauert nur sehr kurz
IO burst: Thread wartet auf IO Operation, dauert mitunter lange (HDD/Eingabe)
Nur CPU Burst kann nicht parallelisiert werden
Thread states: Ready (Wartet auf CPU zuweisung), Running, Blocked



Terminate: Selbst terminieren, durch Job Control oder andere Prozesse beendet
Block: Page Fault, IO, Synchronisation
Ready: Page fault aufgelöst, IO abgeschlossen, Synchronistationsbedingung gegeben
Resign: Kontextwechsel selbst auslösen (sleep(0)), Zeitüberschreitung, Scheduler

Scheduling

Kriterien: CPU Gebrauch, Wartezeit, Kontextwechselzeit, Interruptlänge

Preemptive Scheduling:

- Regelmäßige Kontextwechsel
- Keine CPU Monopole möglich
- Zusätzliche Kontextwechsel und dadurch geringer overhead

Non-Preemptive Scheduling:

- Nur Kontextwechsel wenn Thread diesen selbst auslöst
- CPU Monopol gut möglich
- Realtime Berechnung

First-Come, First-Serverd (FCFS):

Non-Preemptive, einfache Queue
Durchschnittliche Wartezeit hängt stark von Anmeldereihenfolge ab

- Wenn kurze Aufgaben nach großen ausgeführt werden erhöht sich durchschnittliche Wartezeit drastisch

Priority based:

Jeder Thread bekommt vom System oder Benutzer eine Priorität zugeordnet
Der Prozess mit der höchsten Priorität wird aktiviert
Preemptive: Wenn ein Thread höherer Priorität bereit wird, wird ein Kontextwechsel ausgelöst
Nachteil: Prozesse mit niedriger Priorität werden wohlmöglich nie ausgeführt

Round-Robin (RR):

FCFS mit periodischem Interrupt (10-20ms)
Problem: Threads mit IO Burst nutzen die Zeitspanne nicht -> ineffizient
Lösung: Multi-Level: Zusätzliche Queue für Threads, die keine komplette Zeitspanne brauchen

Multi-Level Scheduling:

1. Level: Auswahl der Queue meist durch Prioritäten
2. Level: Reihenfolge innerhalb der Queue - hier werden verschiedene Strategien benutzt

Gesetz von Amdahl:

Meist ist Software nur zu einem geringen Teil parallelisierbar, der Speedup durch mehr Prozessoren wird immer geringer

Synchronisation

Konkurrenz: Kampf um Ressourcen, OS Synchronisiert implizit
Kooperation: Teilen sich Ressourcen, dadurch Geschwindigkeitsvorteil, beliebig komplexe Synchronisation
Vererbung: Blockt ein Thread hoher Priorität und wartet auf das Beenden eines Threads mit niedriger Priorität, bekommt dieser Thread für die Dauer die Priorität des ersten Threads.
Interprocesscommunication (IPC): Synchronisation (Bedingtes Warten, kein Datenaustausch),

- Kommunikation (Datenaustausch)
- Kann im selben Addressraum, selben PC oder entfernter PC sein

Mutual Exclusion (Mutex): Wechselseitiger Ausschluss

Semaphore:

P(): Blockiere bis freigegeben, falls bereits blockiert (request entry)

Vermindert internen Zähler um 1, falls Wert < 0 wird der Thread geblockt

V(): Gebe Blockade frei

Erhöht den internen Zähler um 1, falls Wert < 1 wird ein Thread durchgelassen

Wird eine Semaphore mit n initialisiert, dürfen maximal n Threads gleichzeitig ausgeführt werden.

Barrier:

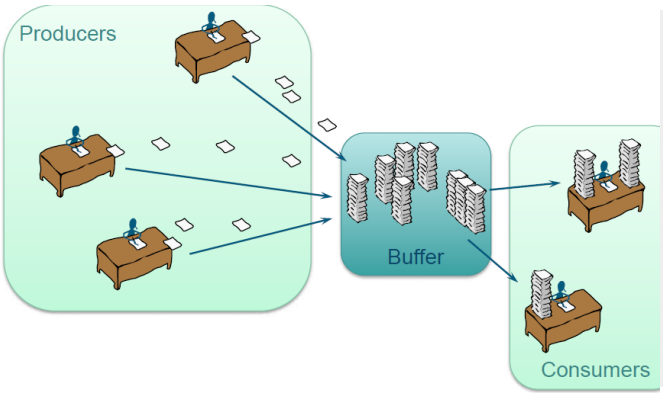
Warte bis alle n Threads die Barrier erreicht haben, lasse dann alle durch.

```
1  rendezvous
2
3  mutex.wait()
4      count = count + 1
5  mutex.signal()
6
7  if count == n: barrier.signal()
8
9  barrier.wait()
10 barrier.signal()
11
12 critical point
```

Wiederverwendbare 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()
```

Producer-Consumer Problem




```
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;
}
```

Reader-Writer Problem

Ineffiziente Version:

Shared Data

```
Semaphore Sanctum = new Semaphore(1);
```

Writer

```
while (true) {
  Sanctum.P();
  // Change data
  Sanctum.V();
}
```

Reader

```
while (true) {
  Sanctum.P();
  // Read data
  Sanctum.V();
}
```

Reader Preference (Writer warten u.U. unendlich):

Shared Data

```
Semaphore Sanctum = new Semaphore(1);
Semaphore RMutex = new Semaphore(1);
int readers_inside = 0;
```

Writer

```
while (true) {
  Sanctum.P();
  // Change data
  Sanctum.V();
}
```

Reader

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

Writer Preference (Writer warten bis alle Reader fertig sind, keine neuen Reader erlaubt):

Shared Data

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

Writer

```
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();
}
```

Reader

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

Writer Preference (Reader Queue, sodass Writer nur auf maximal einen Reader warten muss):

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;
```

Writer

```
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();
}
```

Reader

```
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();
}
```

Dining Philosophers

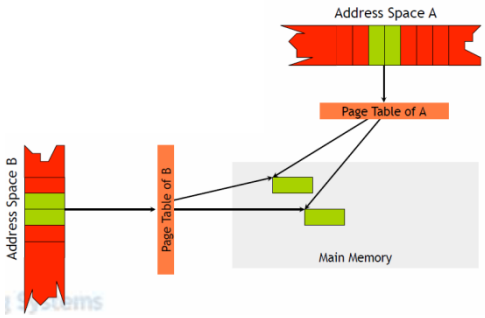
```
public int join ( int id ) {
    while (true) {
        mutex.P();
        for (int s = 0; s < seats.length; s++) {
            if (!seats[s].occupied) {
                seats[s].occupied = true;
                seats[s].id = id;
                mutex.V();
                return s;
            }
        }
        in_queue++;
        mutex.V();
        queue.P();
    }
}
```

```
public void printState () {
    mutex.P();
    System.out.print("Table: ");
    for (Seat s: seats) {
        if (s.stick.value() == 0)
            System.out.print(". ");
        else
            System.out.print("| ");
        if (s.occupied)
            System.out.format("<%3d> ", s.id);
        else
            System.out.print(" ---  ");
    }
    mutex.V();
}
```

Essenstäbchen (Sticks) sind eigene Semaphoren, das Aufnehmen eines Stäbchens ist s.P()

IPC

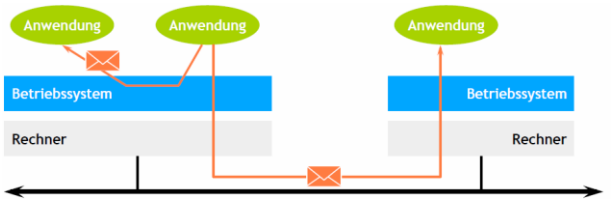
- Innerhalb eines Prozesses (gleicher Addressraum)
- Innerhalb eines Systems (shared Memory)
  - Die Seitentabellen der Prozesse zeigen auf einen gemeinsamen Speicherbereich (shared Memory)
  - **Achtung**: Werden im shared Memory Pointer auf den Speicherbereich eines Prozesses gespeichert, kann der andere Prozess beim Zugriff darauf abstürzen
  - Vorteile**:** Effizient und schell
  - Nachteile**:** Synchronisation, keine Rückmeldung, wann kommuniziert wurde
- Zwischen verschiedenen Systemen (message passing/distributed shared memory)



Message-based Communication:

Es werden Nachrichten ausgetauscht, dafür gibt es zwei Operationen (send, receive)

Der Austausch von Nachrichten innerhalb eines Systems und eines verteilten Systems unterscheidet sich nur kaum.



Vorteile: Übergreifendes Konzept

Nachteile: Ineffizient

Synchrone Kommunikation: Sender blockiert, bis Nachricht empfangen wurde

Asynchrone Kommunikation: Sender blockt nur, bis Nachricht kopiert wurde (Betriebssystem schickt diese dann ab)

Kommunikationsarten: Notification (einfache Nachricht an Empfänger), Service Invokation (Request mit Reply),

Multicast (Mehrere Empfänger), Bradcast (sendet an gesamtes Netzwerk)

	Asynchronous	Synchronous
Notification	Datagram	Rendezvous
Service Invocation	Asynchronous Service Invocation	Synchronous Service Invocation

Datagram: OS muss Nachrichten Puffern, Sender und Empfänger unabhängig

Bsp.: UDP, Signals

Rendezvous: Sender und Empfänger blockieren, bis Nachricht empfangen wurde.

Nachricht wird direkt kopiert, nicht gepuffert

Sender bekommt Rückmeldung, dass Nachricht angekommen

Bsp.: Ada, QNX

Synchronous Service Invocation: Send, Receive, Reply

Sender bekommt Empfangsbestätigung und Ergebnis

Bsp.: RPC (Remote Procedure Call)

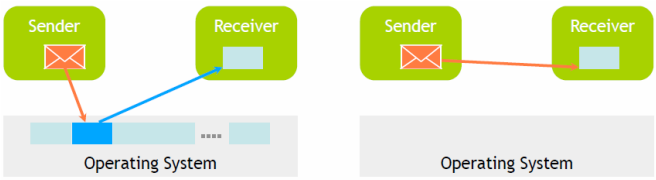
Asynchronous Service Invocation: Nicht blockierendes Senden

Nachrichtenaustausch zwischen Adderssräumen:

Synchron: Sender und Empfänger blockierend

Nachricht wird direkt zum Empfänger kopiert

Geringe Parallelität



Asynchron: Sender kopiert in OS Buffer, OS kopiert Nachricht von Buffer zum Receiver, 2 Kopien der Nachricht

Buffer muss verwaltet werden, evtl. Overflow

Direct/Indirect Communikation:

Direct: Empfänger wird z.B. per Prozess ID direkt kontaktiert

Effizient, schwer umzusetzen

Indirect: Empfänger erreichbar über Port oder Mailbox

Flexibel, Viele Empfänger möglich, mehr Arbeitsaufwand

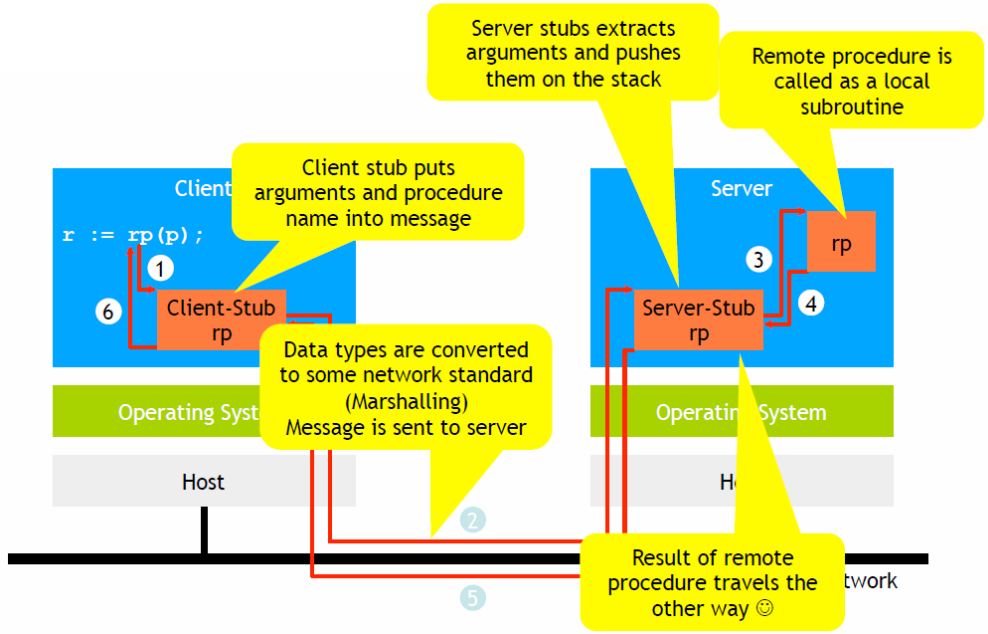
Pipes/Named-Pipes:

FIFO Buffer bestimmter Länge

Empfänger blockiert, wenn Buffer leer

Sender blockiert, wenn Buffer voll

RPC:



File Systems:

Persistenter Speicher

Abstraktion: Datei (Sequenz von Daten inklusive Name und Metainformation)

Name, Typ, Position auf Datenträger, Größe, Rechte, Timestamps etc.

Directories: Dateien, die Dateien enthalten

Die meistern Dateien sind sehr klein oder sehr groß, es wird meistens gelesen

Moderne Dateisysteme versuchen die Kopfbewegung der Platte gering zu halten

Update in Place vs Log: