Betriebssystem 4

内存虚拟化 : Illusion von privatem Speicher

- Abstraktion: Adressraum
- Mechanismus: Adress-Übersetzung, Paging, Swapping
- Poicy: Seitenersetzung

1. Structure:
   - Aufbau des Adressraums eines Prozesses
     - Stack: LIFO, dynamischer Speicher nach festem Muster
     - Heap: Verwaltung von dynamischem Speicher
   - Virtualisierung des Speichers(everything is about how to dealing with the Adressraum)
     - Zeitliche Aufteilung
     - Statische Verschiebung
     - Dynamische Verschiebung
     - Segmentierung
2. Definition:
   - Uniprogrammierung: Genau ein Prozess läuft zur gleichen Zeit und had Vollzugrigg auf Speicher
     - ```
       #Physischer Speicher
       OS
       .
       .
       User Prozess #Adressraum
       .             |Code |
       .             |Heap |
       .             |...  |
       .             |...  |
       .             |Stack|
       ```
     - Nachteile:
       - Es läuft immer nur ein Prozess gleichzeitig
       - Kein Schutz. Prozess kann BS überschreiben/übernehmen
   - Multiprogrammierung
     - Transparenz:
       - The process doesn't know that the memory is shared with other processes
       - The OS works independetly regardless of how many processes are running
     - Schutz
     - Effizienz:
       - verschwendet keine Speicherressourcec
     - Sharing:
       - Kooperierende Prozesse Können Teile des Adressraums gemeinsam nutzen
   - Adressraum
     - Definition: eine Reihe der Adressen von Prozesse
     - statisch: Code und einige globale Variablen
     - dynamisch: Stack und Heap
3. Dynamischer Speicher
   - **Stack** und **Heap**
   - Benutzung: wenn die Menge des benötigten Speicher zur Compile-Zeit nicht bekannt ist
       ```

    - statischer Speicherzuweisung: muss man perssimistisch die maximale Menge an Speicher reservieren(meistens ineffizient)
    - rekursive Prozeduren
    - komplexe Datenstrukturen: Listen und Bäume

4. Stack (LIFO): a region of memeory
    - `1. The stack grows and shrinks dynamically as functions are called and return.`
      `2. Each time a function is called, a new stack frame is pushed onto the stack.`
      `3. When the function execution completes, its stack frame is popped off the stack.`
    - Daten:
        - `push 1; push 2; push 3;`
          `pop -> 3; pop -> 2; pop -> 1;`
    - Speicher:
        - `a = alloc(20); b = alloc(10);`
          `free(b); free(a)`
    - Stack Frame: a single unit within stack
        - A data structure created on the stack to hold information which is necessary and specific to a function call.
    - Stack Pointer: ist ein einzelnes Register, trennt zugewisenen und freigegebenen Speicherplatz
        - Allokieren: Pointer dekrementieren. Speicher wird reserviert wenn die Funktion aufgeruhen wurde.
        - Freigeben: Pointer inkrementieren. Speicher wird freigegeben, wenn die Funktion zurückkehrt.
        - Freigegebener Speicher wird nicht sofort überschriben, sondern erst bei Bedarf, wenn ein späteres Stack Frame diesen Bereich wieder nutzt.
    - Verwendung: um lokalen und parameter zu speichern.
        - Erlaubt einfaches Management vom lokalen Zustand eiener Funktion
            - rekursive Aufrufe derselben Funktion erstellen eigene Versionen der Variablen in ihrem Stack Frame (
            - im stack liegen die StackFrames der aktiven Funktionen in der aktuellen Aufrufkette

5. Heap
    - besteht aus Objekten(Variablen, Datenstrukturen), die unregelmäßig und statisch unbekannter Größe allokiert werden
    - Life time: Für Heap-Objekte steht nicht fest, wie lange sie benötigt werden. (Gegensatz zu lokalen Variablen im Stack, deren Lifetime durch den Funktionsaufruf begrenzt ist)
        - können nach bedarf vom Programmcode freigegeben werden
    - Freier Speicher im Heap muss wiederverwendent werden können
    - `malloc`: erzeugt ein Speicherobjekt gegebener Größe in Bytes und gibt Pointer zurück
    - `free`: gibt ein Speicherobjekt wieder frei, nimmt Pointer als Parameter
    - Vorteil: funktioniert für ale Datenstrukturen
    - Nachteil:
        - die Zuweisung kann langsam sein
        - nach einiger Zeit kann es zu externer Fragmentierung des freien Speicherplatzes kommen
    - Rolle des Betriebssystems:
        - gibt einen großen Teil des freien Speichers an den Prozess
        - die STD-Bibliothek verwaltet die einzelnen Zuweisungen

6. Free List und Strategien

- Idee: “Löcher” im Heap werden in einer Liste von freien Blöcken organisiert, das ist genau der *Free List*
- verwaltet beim Aufruf von `malloc` oder `free` von STD-Bibliothek
- `malloc`:

```
1. wähle einen freien Block aus der Free List
2. Spalte die angefragte Speichergröße davon ab
3. Füge den Rest des freien Blocks wieder in die Free List ein
```

- `free`:

```
füge Block wieder in die Free List ein
```

- Strategien:
  - `malloc`: first/best/worst/next fit
  - `free`: vereinige freie Nachbarblöcke
  - Buddy Allocator
    - im Baumstruktur organisiert
    - Strategie:
      - bei jeder Allokation werden möglichst kleine, noch freie Blöcke so lange halbiert, bis sie gerade noch groß genug sind, um die Anfrage zu erfüllen
      - bei jeder Freigabe wird der Block rekursiv mit seinem “Buddy” vereinigt, sofern dieser frei ist
      - relations between buddys:

        ```
        假设我们有一个内存块大小 s = 64 字节, 内存块的起始地址 a = 0x1000
        buddy_address = a XOR s = 0x1000 XOR 64 = 0x1000 XOR 0x40 = 0x1040
        # XOR 是异或按位运算（二进制）。相同返回 0，不同返回 1
        ```

      - Vorteil:vereinfacht die Verbindung der Blöcke
      - Nachteil: Feste kleinste Blockgröße, zu kleine Blöcke führen zu großem Baumstruktur
  - Interne Fragmentierung: Gibt der heap Manager Blöcke fester Größe aus(z.B. Buddy Allocator), werden Teile des Blocks vom Programm möglicherweise nicht genutzt und sind damit verschwendet
  - Externer Fragmentierung: Durch wiederholter Allokation und Deallokation sind die einzelnen Blöcke in der Free List/im Buddy System zu klein um eine Anfrage zu erfüllen, obwohl insgesamt noch genügend Speicher im Heap vorhanden wäre.
- Zuordung von Variablen

```c
int x;
int main(int argc, char *argv[]) {
  int y;
  int *z = malloc(sizeof(int));
}
// x : Statische Daten (oder Code)
// main : Code
// y : stack
// z : Stack
// *z : Heap
```

- Note: 指针与指针指向的内存地址的值

```c
  int main() {
    int *z = malloc(sizeof(int));  // 动态分配内存，z 指向这块内存

    *z = 10;  // 给指针 z 指向的内存位置赋值
```

```
    printf("z = %p\n", z);  // 输出指针 z 的值（内存地址）
    printf("*z = %d\n", *z); // 输出指针 z 指向的内存位置的值，即 10

    free(z);  // 释放动态分配的内存
    return 0;
}
```

7. Speicherzugriffe
   - ```
     int main(int argc, char *argv[]) {
       int x;
       x = x + 3;
     }
     ```
   - ```
     0x0b: mov -0x14 (%rbp), %eax
     0x0e: add $0x3, %eax
     0x11: mov %eax, -0x14(%rbp)
     ```
     - Explaination:
       - mov: 将数据从一个地方复制到另一个地方。mov –0x14(%rbp), %eax 表示从内存地址 –0x14(%rbp) 取出数据并将其加载到寄存器 %eax 中， mov %eax, –0x14(%rbp) 表示将 %eax 中的数据存回到内存地址 –0x14(%rbp)
       - add: 加法运算， `add $0x3`, %eax 表示将立即数 0x3 加到 %eax 寄存器的值上。
       - %rbp: 这是 64 位架构中的一个通用寄存器，通常用于存储函数的基指针（frame pointer），即当前函数栈帧的起始地址。-0x14(%rbp) 是基于这个寄存器的栈偏移地址，用来访问局部变量（在这里是 x）
       - `%rbp` ist der Basiszeiger, zeigt auf den Beginn des aktuellen Stack StackFrames
       - `%rip` ist der Befehlszeiger(program counter)

8. Speicher Virsualisierung
   - Problem:
     - Wie können mehrere Prozesse gleichzeitig ausgeführt werden?
     - Adressen sind im Maschinencode der Prozesse fest vorgegeben.
   - Mögliche Lösung:
     - time sharing
     - static relocation
     - dynamic relocation with Basisregister
     - Base + Bound
     - Segmentierung

9. Zeitliche Aufteilung des Hauptspeichers(time sharing)
   - Idee: wie bei der Virtualisierung der CPU
   - Ansatz: wenn ein Prozess nicht läuft, lagern der Hauptspeicher auf die Festplatte aus.
   - Proleme: katastrophale Performanc durch Disk-I/O bei jedem Kontextwechsel
   - Alle weiteren Lösungen nutzen räumliche Aufteilung des Hauptspeichers

10. Statische Verschiebung(static relocation)
    - Das BS schreibt jedes Programm um, bevor es als Prozess in den Speicher geladen wird.
    - Beim Umschreiben für verschiedene Prozesse unterschiedliche Adressen verwenden
    - Nachteil:
      - Kein Schutz
        - Prozess kann Betriebssystem oder andere Prozess zerstören
        - keine Privatsphäre

- Adressraum kann nicht verschoben werden, nachdem er platziert wurde

11. Dynamische Verschiebung(dynamic relocation)
    - Ziel : Schutz der Prozesse voreinander
    - Hardware-Unterstützung: MMU(Speicherverwaltungseinheit/Memory Management Unit), um virtuellen Adressen mit Basisregister zu übersetzen
        - MMU ändert die Prozessadresse dynamisch bei jeder Speicherreferenz
        - Prozess erzeugt logische oder virtuelle Adressen(in seinem Adressraum)
        - Speicherhardware verwendet physische/reale Adressen

          ```
          (Prozess läuft hier)        BS kontrilliert MMU
                    CPU --logische Adresse--MMU -- Physische Adresse-- Memory
          ```

        - zwei Betriebsmodi
            - Privilegierter Modus: OS läuft
                - geschützt, kernel
                - beim Eintritt in das Betriebssystem(Trap, Systemaufruf, Unterbrechungen, AUsnahmen)
                - Erlaubt die AUsführung bestimmter Befehle
                - Kann den Inhalt der MMU manipulieren
                - Ermöglicht dem Betriebssystem den Zugriff auf den gesamten physischen Speicher
            - Benutzermodus: Benutzerprozess laufen
                - Übersetzung der logischen Adresse in die physishce Adresse durchführen
        - Basisregister für die Übersetzung
            - Basis: Startposition für den Adressraum
            - MMU addiert Basisregister zur logischen Adresse, um die physikalische Adresse zu bilden
            - **Ablauf**
                - virtuelle Adressen: Ein Prozess hat einen virtuellen Adressraum, der durch einen Startwert, das Basisregister, und eine Grenze (Größe des zugewiesenen Speichers) definiert wird.
                - Adresseübersetzung: Wenn der Prozess auf eine Adresse zugreifen möchte, wird der virtuelle Adressraum durch das Basisregister verschoben, um die physische Adresse zu berechnen. Dies geschieht in der Hardware mithilfe der Memory Management Unit (MMU).
                - Speicherzugriffsprüfung: Bei jedem Zugriff auf eine Adresse prüft die MMU, ob der Zugriff innerhalb des zugewiesenen Speicherbereichs des Prozesses liegt. Wenn der Zugriff außerhalb dieses Bereichs liegt, wird ein Speicherfehler (oft ein Segmentation Fault) ausgelöst.
            - Implemetierung und Idee: Übersetzung virtueller Adressen in physische, indem jedes Mal ein fester Offset hinzugefügt wird
                - Offset im Basisregister speichern
                - Jeder Prozess hat einen anderen Wert im Basisregister
            - Die **Hardare** soll die Übersetzung der Adressen mit dem Basisregister vornehmen
                - wegen Performance, Sicherheit und Transparenz. BS ist nicht direkt verantwortlich für die Übersetzung jeder einzelnen virtuellen Adresse während der Ausführung eines Prozesses.
            - **BS** soll das Basisregister ändern.
                - nicht Prozess wegen Isolation
                - nicht die Hardware, da die Zuweisung und Verwaltung des virtuellen Speichers eine hohe Abstraktionsebene ist
                - Das Betriebssystem ist verantwortlich für die Zuweisung von virtuellem Speicher an einen Prozess. Wenn der Speicher für einen neuen Prozess oder für einen bestehenden

Prozess erweitert oder verändert wird (z. B. bei einem Kontextwechsel), muss das Betriebssystem das Basisregister anpassen, um sicherzustellen, dass der virtuelle Adressraum des Prozesses korrekt auf den zugewiesenen physischen Speicher verweist.

12. Dynamisch mit Base + Bounds
    - Idee: Begrenzung des Adressraums durch ein Boundsregister
    - Ziel: Übersetzung bei jedem Speicherzugriff eines Benutzerprozesses
    - Ablauf:
        - MMU vergleicht logische Adresse mit Bounds-Register
        - MMU addiert Basisregister zur logischen Adresse, um die physikalische Adresse zu bilden
    - vs. Basisregister
        - Basisregister: Startadresse
        - Größe des virtuellen Adressraums dieses Prozesses
    - Das Betriebssysteme beendet den Prozess, wenn er über die Grenzen hinaus lädt/schreibt

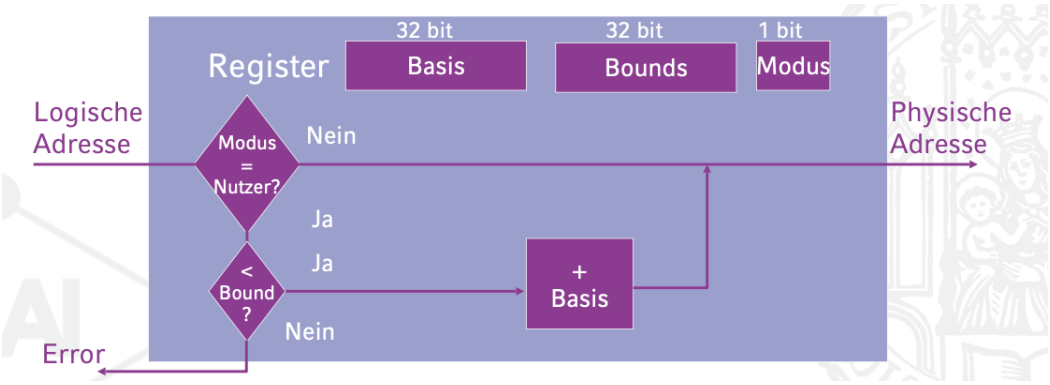

Figure 1: Implementation of Base + Bounds.

    - Verwaltung von Prozessen mit Base+Bounds
        - Kontextwechsel:
            - Hunzufügen von Basis- und Boundsregistern zu PCB
            - Schritte:
                - In den privilegierten Modus wechseln
                - Basis- und Boundsregister des alten Prozesses speichern
                - Laden der Basis- und Boundsregister des neuen Prozesses
                - In den Benutzermodus wechseln und zum neuen Prozess springen
            - Schutzbedingunen: Benutzerprozess kann :
                - nicht Basis- und Bounds-register ändern
                - nicht in den privilegierten Modus wechseln
    - Vorteile:
        - Schutz
        - Unterstützt dynamische Verschiebung
        - Einfache, kostengünstige Implementierung
        - Schnell
    - Nachteile:
        - Jeder Prozess muss zusammenhängend im physischen Speicher zugewiesen werden(muss Speicher zuweisen, der nicht von einem Prozess verwendet werden darf)
        - Keine teilweise gemeisame Nutzung: Begrenzte Teile des Adressraums dürfen nicht gemeinsam genutzt werden

13. Segmentierung(分段寻址)
    - Idee: Unterteilung des Adressraums in logische Segmente

  - Jedes Segment entspricht einer logischen Einheit im Adressraum
    - code, stack, heap
- **kann unabhängig**
  - separat im physischen Speicher platziert werden
  - wachsen und schrumpfen
  - geschützt werden(separate Lese-/Schreib-/Ausführungsschutzbits)
- 物理地址 = 基准地址 + 偏移量(Offset)
  - 一个虚拟地址要在某个段内有效，必须满足它的偏移量（相对于段的起始地址）小于该段的长度
  - otherwise: Segmentation Fault
- Vorteile:
  - ermöglicht vereinzelte Zuweisung von Adressraum
    - Stack und Heap können unabhängig voneinander wachsen
    - Heap: 如果当前没有足够的空闲内存块（即空闲内存空间）可供程序使用，程序就会向操作系统请求更多的内存。
      - `malloc` 调用 `sbrk()`, 向操作系统申请更多的内存空间
    - Stack: Betriebssystem erkennt Verweis außerhalb des legalen Segments, erweitert den Stack implizit.
    - Unterschiedlicher Schutz für verschiedene Segmente (eg. Read-only status für Code)
    - Ermöglicht die gemeinsame Nutzung von ausgewählten Segmenten
      - 代码段共享，数据共享
    - Unterstüzt die dynamische Verlagerung jedes Segments
      - 程序的段可以根据系统的内存情况在内存中动态地重定位。
- Nachteile:
  - Jedes Segment muss zusammenhängend zugewiesen werden
    - Für große Segmente steht möglicherweise nicht genügend physischer Speicher zur Verfügung (externe Fragmentierung)