#uni/semester3/Betriebssysteme/chapter16/chapter-notes

➜ The base and bounds Methode verschwendet viel Speicher, weil der Speicher zwischen heap und stack nicht verwendet wird. Es ist auch nicht flexibel, dass schwer macht ein Programm auszuführen, wenn sein address space nicht im Speicher reinpasst.

1. Segmentation: Generalized Base/Bounds

➜ Segmentation ist eine Lösung, wo es mehrere bases und bounds Paare gibt, per logical segment. Somit kann man jedes Segment an unterschiedlichen Orte im physical memory speichern.

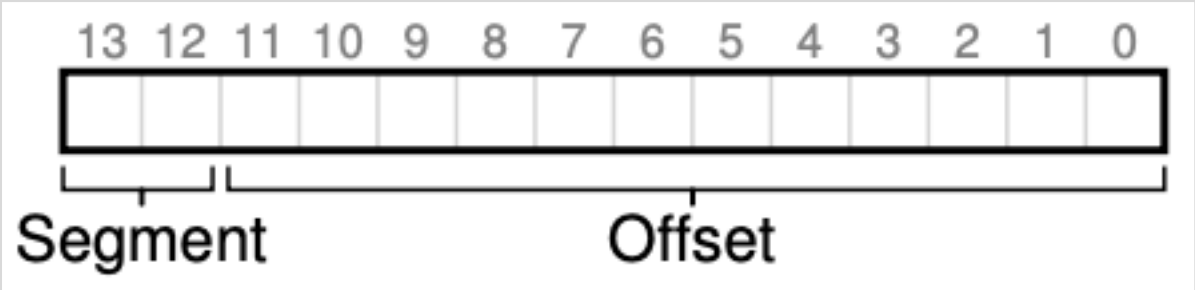
➜ Im physical memory befinden sich dann nur Speicher das verwendet wird. address spaces die viel unverwendtes address space haben, nennt man sparse address spaces.

➜ MMU verwendet dann als Hardware structure eine Liste mit den bases und sizes Paare.

➜ Bei illegal address, hardware sieht das die address out of bounds ist und traps im OS. Das Programm wird dann gekillt. Das nennt man dan segmentation fault.

2. Which Segment Are We Referring To?

➜ Woher weiß die hardware welchen offset die Adresse hat und welches segment es meint? Explicit approach ist indem man die VA (virtuelle adresse) in segment und offset einteilt



```
// get top 2 bits of 14-bit VA
Segment = (VirtualAddress & SEG_MASK) >> SEG_SHIFT
// now get offset
Offset  = VirtualAddress & OFFSET_MASK
if (Offset >= Bounds[Segment])
    RaiseException(PROTECTION_FAULT)
else
    PhysAddr = Base[Segment] + Offset
    Register = AccessMemory(PhysAddr)
```

In our running example, we can fill in values for the constants above. Specifically, `SEG_MASK` would be set to `0x3000`, `SEG_SHIFT` to `12`, and `OFFSET_MASK` to `0xFFF`.

➜ Wenn man nur drei Segmente hat, wird Platz für ein Segment verschwendet. Deshalb wird manchmal code und heap in einem Segment gemacht. Für die segment bits dann nur 1 bit. Dann unterscheidet man nur zwischen Code+Heap und Stack.

➜ Anderes Problem, ist dass jeder Segment eine maximale Größe dann hat (2 hoch offset bits). Wenn ein Programm ein segment vergrößern möchte, kann er nicht.

➜ Bei implicit approaches schaut wie die Adresse geformt wurde und wählt dann das segment.

- ➜ Adresse von program counter, Adresse dann im code segment
- ➜ Adresse von stack or base pointer, dann stack segment
- ➜ Alle andere, auf heap segment

3. What About The Stack?

➜ Der Stack wächst rückwärts, deshalb müssen die virtuellen Adressen anders berechnet werden. Wenn der Stack bei z.B. 28 kB startet, ist eigentlich der erste valide Byte 28kB - 1.

➜ Ausserdem wird hardware support gebraucht. Deshalb wird die Segmententabelle um ein Boolean wert erweitert, dass die Richtung des Wachstums verfolgt

| Segment | Base | Size (max 4K) | Grows Positive? |
|---|---|---|---|
| $\text{Code}_{00}$ | 32K | 2K | 1 |
| $\text{Heap}_{01}$ | 34K | 3K | 1 |
| $\text{Stack}_{11}$ | 28K | 2K | 0 |

Figure 16.4: **Segment Registers (With Negative-Growth Support)**

➜ Berechnung erfolgt so. Man nimmt die virtuelle Adresse, schreibt sie als segment bits und offset bits. Der offset ist dann der Anteil nur mit offset bits.
Dann berechnet man den negativen offset = offset - maximum segment size;
Maximum segment size = 2 hoch offset bits
Am Ende addiert man einfach den negativen offset zu den base aus der Tabelle. Um den bounds zu prüfen muss man abs(negativen offset) mit aktuelle Größe des Segments vergleichen. Valid heisst dann, kleiner oder gleich.
Maximum segment Größe ist nicht unbedingt gleich mit aktuelle grosse des segments.

4. Support for Sharing

➜ Indem man bestimmte memory segments zwischen address spaces teilt, wird es effizienter. Besonders code sharing wird heute noch verwendet.

➜ Dafür muss die Segmenttabelle um protection bits erweitert werden. Diese besagen ob ein Segment gelesen, geschrieben oder ausgeführt werden kann. Die Illusion bleibt, und die Prozesse denken weiter dass es auf sein eigener private memory zugreift.

| Segment | Base | Size (max 4K) | Grows Positive? | Protection |
|---|---|---|---|---|
| $\text{Code}_{00}$ | 32K | 2K | 1 | Read-Execute |
| $\text{Heap}_{01}$ | 34K | 3K | 1 | Read-Write |
| $\text{Stack}_{11}$ | 28K | 2K | 0 | Read-Write |

Figure 16.5: **Segment Register Values (with Protection)**

➜ Neben der bounds checking, muss die Hardware dann auch schauen ob der access zu einem bestimmten Segment permissible ist. Wenn nicht, muss die hardware eine exception aufrufen und OS muss mit dem Prozess handeln

5. Fine-grained vs. Coarse-grained Segmentation

➜ coarse-grained ist das Aufteilen in relativ grossen Segmente, wie code, stack und heap
➜ fine-grained segmentation, ist wenn man in sehr viele aber kleine Segmente aufteilt. Dafür wird mehr hardware support erfordert mit einer segment table.
➜ segment tables ermöglicht ein viel flexibleres System.
➜ durch fine-grained segments kann OS besser lernen welche Segmente gebraucht werden und welche nicht und somit main memory effizienter verwenden.

5. OS Support

➜ segmentation bringt auch ein paar Probleme mit sich
➜ was macht OS beim context switch? Die segment registers müssen gespeichert und restored werden
➜ OS interaction, wenn segments kleiner oder grosser werden. In den meisten Fällen kann heap bedienen, und via malloc bekommt man einen Pointer zu diesem free space. In anderen Fällen, muss das heap segment selber vergrößern werden, via system call sbrk() in UNIX. Dann OS wird mehr Speicher zur Verfügung stellen, indem er die segment size register auf den neuen Wert setzt. Die memory allocation library kann dann the space allokieren und wieder zurückgeben. OS kann auch nein sagen, weil kein physical memory mehr oder weil Prozess schon zu viel Speicher hat.
➜ wichtigstes problem ist free space in physical memory zu verwalten. Wenn ein neuer address space kommt, muss OS Platz im physical memory für seine Segmente finden. Wir haben eine verschiedene Anzahl an Segmente der Prozess und mit verschiedenen sizes. Das Problem ist, dass physical memory voll mit freien und kleinen holes wird. Es macht schwer neue Segmente zu allokieren oder grow existing ones. Das nennt man external fragmentation.
➜ also man hat theoretisch genug platz, aber dieser Platz ist nicht contiguous.
➜ eine solution ist alles umzuordnen zu einem compact physical memory. Compaction ist sehr teuer weil man muss Segmente kopieren und change die segment Registern. Compaction macht auch requests um manche Segmente zu grow, was ironisch ist.
➜ Einfachere solution ist free list management algorithm, um large extends of memory available zu halten. Es gibt best-fit, worst-fit, first-fit und complexer the buddy algorithm. Leider external fragmentation wird immer passieren, und diese Algorithmen sind da um es zu minimieren.
➜ Die einzige real solution, ist das problem zu verhindern, indem man nie memory in variablen sized chunks allokiert.