```
char bigmem[8192]; /* hoffentlich ausreichend */
char* bigmem_free = bigmem;

void* alloc(size_t nbytes) {
   if (bigmem_free + nbytes > bigmem + sizeof bigmem) {
      return 0; /* kein Speicher mehr da */
   }
   void* p = bigmem_free;
   bigmem_free += nbytes;
   return p;
}
```

- Eine triviale Speicherverwaltung, die nur Speicher aus einer Fläche abgibt, ohne freiwerdenden Speicher entgegennehmen zu können, ist einfach zu implementieren.
- Hier dient das große Array *bigmem* als Speicher-Reservoir.
- Der Zeiger *bigmem_free* zeigt auf den Bereich aus dem Array, der noch nicht vergeben ist. Alles davor wurde bereits vergeben. Zu Beginn wird der Zeiger auf den Anfang des großen Arrays gesetzt.

```
char bigmem[8192]; /* hoffentlich ausreichend */
char* bigmem_free = bigmem;

void* alloc(size_t nbytes) {
   if (bigmem_free + nbytes > bigmem + sizeof bigmem) {
      return 0; /* kein Speicher mehr da */
   }
   void* p = bigmem_free;
   bigmem_free += nbytes;
   return p;
}
```

- Wenn es in der Speicherverwaltung darum geht, Adressarithmetik zu verwenden, dann wird **char**$*$ als Zeigertyp benutzt, da ein **char** die Größe eines Bytes hat, d.h. **sizeof**(**char**)$==1$.
- (Theoretisch legt der Standard nur fest, dass ein Byte mindestens 8 Bits hat. Prinzipiell kann ein Byte größer sein, aber der Standard definiert fest, dass ein Byte im Sinne des Standards von **char** repräsentiert wird und dass **sizeof**(**char**)$==1$ gilt.)
- Entsprechend ist **char** die kleinste adressierbare Größe.

badalign.c

```
char* cp = alloc(sizeof(char));
int* ip = alloc(sizeof(int));
if (cp && ip) {
   * cp = 'x'; * ip = 1;
} else {
   fprintf(stderr, "alloc failed\n");
}
```

- Was passiert, wenn wir über *alloc* unterschiedlich große Objekte anfordern?
- Dann zeigt *ip* auf einen ungeraden Wert!
- Manchen Plattformen macht das nichts aus, andere hingegen akzeptieren dies nicht:

```
theseus$ uname -a
SunOS theseus 5.10 Generic_147440-09 sun4u sparc SUNW,Sun-Fire-V490
theseus$ gcc -o badalign badalign.c
theseus$ ./badalign
Bus Error (core dumped)
theseus$
```

- Manche Architekturen erlauben keinen Zugriff auf größere Datentypen wie etwa **int** oder **double**, wenn sie an irgendwelchen ungeraden Adressen liegen.
- Die SPARC-Architektur beispielsweise besteht darauf, dass **int**-Variablen auf durch vier teilbaren Adressen liegen und **long long int** oder **double** auf durch acht teilbare Adressen.
- Andere Architekturen sind diesbezüglich großzügiger (wie etwa die x86-Plattform), aber mitunter ist die Zugriffszeit größer, wenn keine entsprechend ausgerichtete Adresse benutzt wird.
- Wenn es wegen eines Alignment-Fehlers zu einem Absturz kommt, wird das als „Bus Error“ bezeichnet (mit Verweis auf den Systembus, der sich geweigert hat, auf ein Objekt an einer nicht ausgerichteten Adresse zuzugreifen).
- Eine Speicherverwaltung muss darauf Rücksicht nehmen.

alignment.c

```
#include <stdalign.h>
#include <stdio.h>

int main() {
   printf("alignment for char: %zu\n", alignof(char));
   printf("alignment for int: %zu\n", alignof(int));
   printf("alignment for long long int: %zu\n", alignof(long long int));
   printf("alignment for double: %zu\n", alignof(double));
   printf("alignment for double [10]: %zu\n", alignof(double [10]));
   printf("alignment for long double: %zu\n", alignof(long double));
}
```

- Seit C11 gibt es in C den **alignof**-Operator. Zu beachten ist, dass dieser den **#include** <stdalign.h>-Header benötigt.

```
theon$ ./alignment
alignment for char: 1
alignment for int: 4
alignment for long long int: 8
alignment for double: 8
alignment for double [10]: 8
alignment for long double: 16
theon$
```

- Für Alignments wird ebenfalls der Datentyp *size_t* verwendet.
- Alignments sind immer Zweierpotenzen.
- Die Datentypen **char**, **signed char** und **unsigned char** haben das niedrigste Alignment (typischerweise 1).
- *alignof*(*max_align_t*) liefert das maximale Alignment, das bei einer Plattform notwendig sein kann.
- Größere Alignments können zur Verbesserung der Performance in Betracht gezogen werden, etwa wenn Datenstrukturen auf Cache-Line-Kanten ausgerichtet werden sollen.

maxalign.c

```
#include <complex.h>
#include <stdalign.h>
#include <stddef.h>
#include <stdio.h>

int main() {
   printf("alignof(max_align_t) = %zu\n", alignof(max_align_t));
   printf("alignof(long double complex) = %zu\n",
      alignof(long double complex));
   printf("sizeof(long double complex) = %zu\n",
      sizeof(long double complex));
}
```

- Das maximale Alignment ergibt sich typischerweise durch die entsprechende Anforderung des größten elementaren Datentyps. Dies sind **long double** bzw. – sofern komplexe Zahlen unterstützt werden – **long double complex**.

```
theon$ maxalign
alignof(max_align_t) = 16
alignof(long double complex) = 16
sizeof(long double complex) = 32
theon$
```

```
theseus$ maxalign
alignof(max_align_t) = 8
alignof(long double complex) = 8
sizeof(long double complex) = 32
theseus$
```

alignas.c

```
#include <stdalign.h>
#include <stdio.h>

int main() {
   alignas(32) int int32_a;
   alignas(32) int int32_b;
   printf("sizeof(int32) =  %2zu\n", sizeof(int32_a));
   printf("alignof(int32) = %2zu\n", alignof(int32_a));
   printf("&int32_a = %p\n", &int32_a);
   printf("&int32_b = %p\n", &int32_b);
}
```

- Mit **alignas** kann bei einer Deklaration ein gewünschtes Alignment spezifiziert werden (nicht zulässig für Typdeklarationen).
- Zulässig sind 0 (dann bleibt es bei der Voreinstellung), ein natürlich vorkommendes Alignment größer oder gleich der Voreinstellung oder ein Alignment jenseits dem Maximum, wenn dies vom Übersetzer in dem Kontext unterstützt wird.

```
void* alloc(size_t nbytes, size_t align) {
   char* p = bigmem_free;
   p = (char*) (((uintptr_t) p + align - 1) & ~(align - 1));
   if (p + nbytes > bigmem + sizeof bigmem) {
      return 0; /* kein Speicher mehr da */
   }
   bigmem_free = p + nbytes;
   return p;
}
```

- Das Problem kann durch einen zusätzlichen Parameter mit der gewünschten Ausrichtung gelöst werden. Hier: *align*.
- Der Zeiger *p* wird dann auf die nächste durch *align* teilbare Adresse gesetzt unter der Annahme, dass *align* eine Zweierpotenz ist.
- Alternativ könnte auch immer *alignof*(*max_align_t*) zugrunde gelegt werden – so wird es von *malloc* und *calloc* umgesetzt.

```
p = (char*) (((uintptr_t) p + align - 1) & ~(align - 1));
```

$(uintptr\_t)p$	konvertiere p in eine ganze Zahl
$align - 1$	setzt die n niedrigswertigen Bits, wobei $n = log_2(align)$
$\sim(align - 1)$	bildet das Komplement davon, d.h. alle Bits außer den n niedrigstwertigen sind gesetzt
$\& \sim(align - 1)$	blendet die n niedrigswertigen Bits weg
$(uintptr\_t)p + align - 1$	die Addition stellt sicher, dass das Resultat nicht kleiner wird durch das Wegblenden von Bits

- Das setzt voraus, dass *align* eine Zweierpotenz ist. Davon ist bei Alignment-Größen immer auszugehen.
- *uintptr_t* aus <*stdint.h*> ist ein **unsigned**-Typ, der groß genug ist, um Zeigerwerte aufzunehmen.

```
#define NEW(T) alloc(sizeof(T), alignof(T))

char* cp = NEW(char);      printf("cp =   %p\n", cp);
int* ip = NEW(int);        printf("ip =   %p\n", ip);
char* cp2 = NEW(char);     printf("cp2 =  %p\n", cp2);
char* cp3 = NEW(char);     printf("cp3 =  %p\n", cp3);
double* dp = NEW(double);  printf("dp =   %p\n", dp);
long double* ldp = NEW(long double); printf("ldp =  %p\n", ldp);
```

```
theon$ goodalign
cp =   600ce0
ip =   600ce4
cp2 =  600ce8
cp3 =  600ce9
dp =   600cf0
ldp =  600d00
theon$
```

- *calloc*, *malloc* und *realloc* haben jedoch keine *align*-Parameter.
- In der Praxis wird die Größe des gewünschten Datentyps und die maximal vorkommende Alignment-Größe *alignof*(*max_align_t*) in Betracht gezogen.
- Beginnend ab C11 steht auch

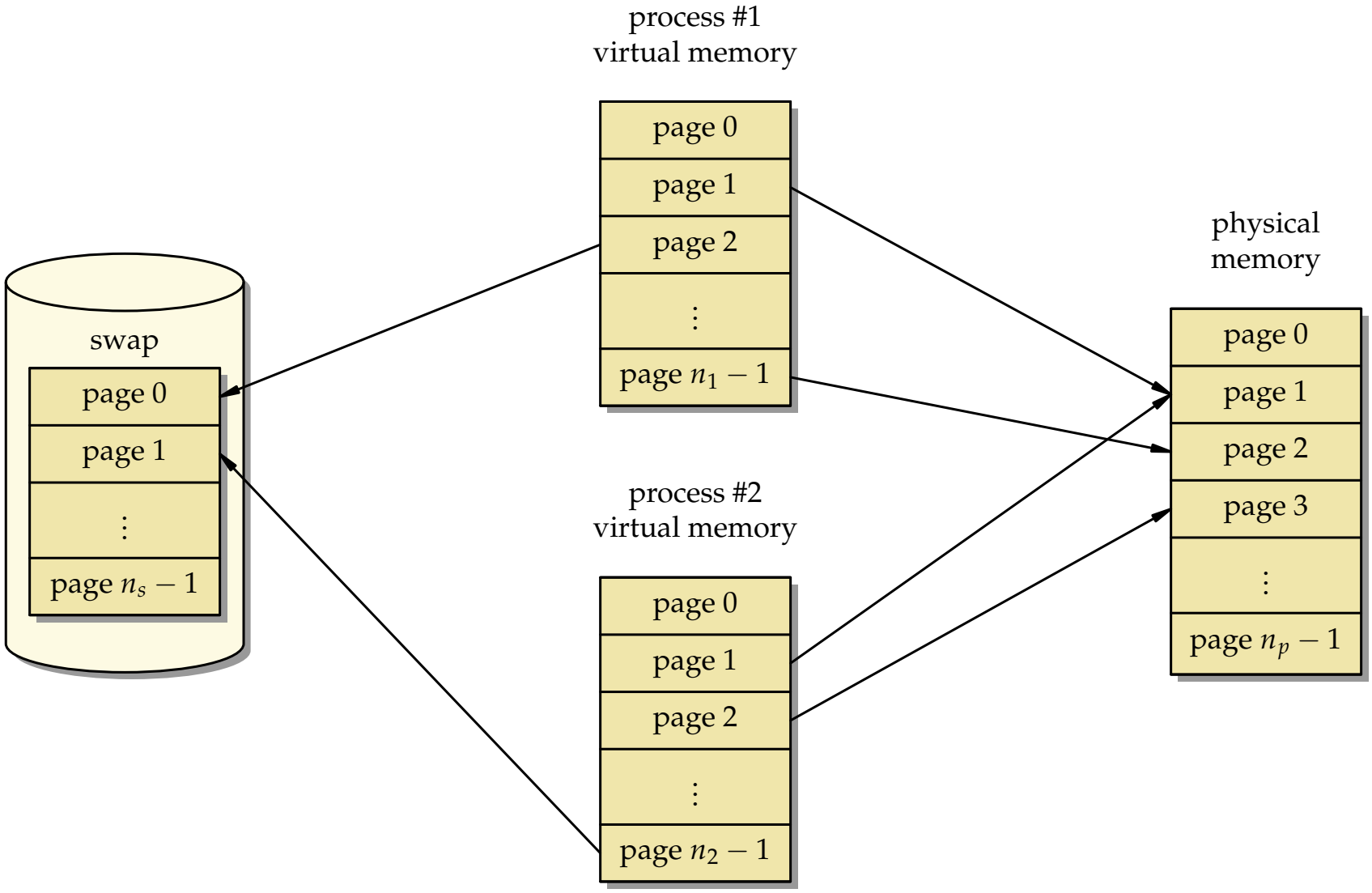
  void* *aligned_alloc*(*size_t alignment*, *size_t size*);

  zur Verfügung. Hierbei muss *alignment* eine vom System tatsächlich verwendete Alignment-Größe sein und *size* ein Vielfaches von *alignment*.
- POSIX bietet auch noch

  int *posix_memalign*(**void**** *memptr*, *size_t alignment*, *size_t size*);

  an, wobei *alignment* eine Zweierpotenz sein muss und im Erfolgsfalle (Return-Wert gleich 0) der Zeiger hinter *memptr* auf die belegte Speicherfläche gesetzt wird.

- Wenn ein Prozess unter UNIX startet, wird zunächst nur Speicherplatz für den Programmtext (also den Maschinen-Code), die globalen Variablen, die Konstanten (etwa die von Zeichenketten) und einem Laufzeitstapel (Stack) angelegt.
- All dies liegt in einem sogenannten virtuellen Adressraum (typischerweise mit 32- oder 64-Bit-Adressen), den das Betriebssystem einrichtet.
- Die Betonung liegt auf virtuell, da die verwendeten Adressen nicht den physischen Adressen entsprechen, sondern dazwischen eine durch das Betriebssystem konfigurierte Abbildung durchgeführt wird.
- Diese Abbildung wird nicht für jedes einzelne Byte definiert, sondern für größere Einheiten, die sogenannten Kacheln (*page*).


process #1
virtual memory
page 0
page 1
page 2
⋮
page $n_1 - 1$
physical
memory
page 0
page 1
page 2
page 3
⋮
page $n_p - 1$
swap
page 0
page 1
⋮
page $n_s - 1$
process #2
virtual memory
page 0
page 1
page 2
⋮
page $n_2 - 1$

getpagesize.c

```
#include <stdio.h>
#include <unistd.h>

int main() {
   printf("page size = %d\n", getpagesize());
}
```

- Die Größe der Kacheln ist plattformabhängig und kann mit Hilfe des Systemaufrufs *getpagesize()* ermittelt werden.

```
theon$ uname -a
SunOS theon 5.11 11.3 i86pc i386 i86pc
theon$ getpagesize
page size = 4096
theon$
```

```
theseus$ uname -a
SunOS theseus 5.10 Generic_147440-09 sun4u sparc SUNW,Sun-Fire-V490
theseus$ getpagesize
page size = 8192
theseus$
```

- Sei $[0, 2^n - 1] \subset \mathbb{N}_0$ der virtuelle Adressraum, $P = 2^m$ die Größe einer Kachel mit $m < n$ und

  $$M : [0, 2^{n-m} - 1] \to [0, 2^{n-m} - 1] \cup \{\text{invalid}\}$$

  die Funktion, die eine virtuelle Kachelnummer in die korrespondierende physische Kachelnummer bzw. in „invalid“ abbildet.
- Dann lässt sich folgendermaßen aus der virtuellen Adresse a_v die zugehörige physische Adresse a_p ermitteln:

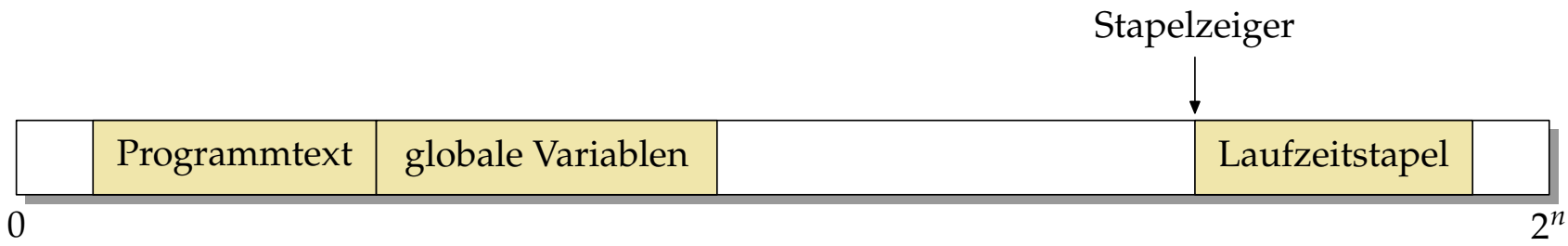
$$a_p = \begin{cases} \text{invalid} & \text{falls } M(a_v \textbf{ div } P) = \text{invalid} \\ M(a_v \textbf{ div } P) \times P + a_v \textbf{ mod } P & \text{sonst} \end{cases}$$

- Eine Abbildung zu „invalid“ führt zu einer Trap, d.h. der Betriebssystemkern übernimmt dann die Kontrolle und erzeugt entweder *SIGSEGV*-Signal (*segmentation violation*) oder passt M an durch das Nachladen von Kacheln von der Platte oder der automatisch erfolgenden Stackvergrößerung.

- Die Funktion M wird von der zur Hardware gehörenden MMU (*memory management unit*) implementiert in Abhängigkeit von Tabellen, die das Betriebssystem konfigurieren kann.
- Die Abbildung erfolgt an der Stelle der Speicherhierarchie zwischen der CPU, den einzelnen Cache-Ebenen und dem Hauptspeicher, bei dem zwischen virtuellen und physischen Adressen gewechselt wird.
- Die CPU selbst und die Cache-Ebenen, die nur einer CPU (bzw. nur einem Core) zugeordnet sind, arbeiten mit virtuellen Adressen, während bei gemeinsam genutzten Cache-Ebenen einheitliche physische Adressen notwendig sind.
- Entsprechend erfolgt die Abbildung genau an diesem Übergang, das ist etwa auf den PCs im Poolraum E.44 zwischen L2 und L3.
- Für die zu M gehörende Tabelle gibt es ebenfalls einen Cache, den sogenannten *translation lookaside buffer*.

- Eine erste Aufteilung des Adressraums mit der Platzierung der Segmente mit dem Programmtext und den globalen Variablen wird durch den *ld* vorgenommen.
- Beim Start des Programms wird ein Bereich für den Stack reserviert.
- Der Laufzeitlader bildet gemeinsam genutzte Bibliotheken (*shared libraries*) in den Adressraum ab.
- All dies hinterlässt einen Adressraum, in dem einzelne Bereiche genutzt werden.
- Für weite Teile des Adressraums bleibt M jedoch undefiniert. Ein Versuch, über einen entsprechenden Zeiger zuzugreifen, führt dann zu einem Abbruch des Programms (*segmentation violation*).

Aufteilung des Adressraums



- Wie der Adressraum zu Beginn belegt wird, liegt in der Freiheit des Betriebssystems bzw. des *ld*.
- Fast immer bleibt die Adresse 0 unbelegt, damit Versuche, auf einen 0-Zeiger zuzugreifen, zu einem Fehler führen. Ebenso bleibt der ganz hohe Bereich bei 2^n frei.
- Der Programmtext und die globalen Variablen liegen normalerweise in der Nähe, sind aber unterschiedlich konfiguriert (*read-only/executable* und *read/write*).
- Der Laufzeitstapel (Stack) wird davon getrennt konfiguriert, damit er genügend Platz hat zu wachsen, typischerweise von hohen zu niedrigen Adressen.

end.c

```
#include <stdio.h>
extern const void etext; extern const void edata; extern const void end;
int global[1000] = {1}; // some global initialized data
int main() { char local;
   printf("main ->   %16p\n", main); printf("etext ->  %16p\n", &etext);
   printf("global -> %16p\n", global); printf("edata ->  %16p\n", &edata);
   printf("end ->    %16p\n", &end); printf("local ->  %16p\n", &local);
}
```

- Bei traditionellen *ld*-Konfigurationen werden die Symbole *etext*, *edata* und *end* definiert, die entsprechend auf das Ende des Programmtexts, das Ende der initialisierten Daten und das Ende der uninitialisierten Daten verweisen.

```
theon$ end
main ->              4007c2
etext ->             400856
global ->            600bc0
edata ->             601b60
end ->               601b68
local ->  ffff80ffbfffe6bf
theon$
```

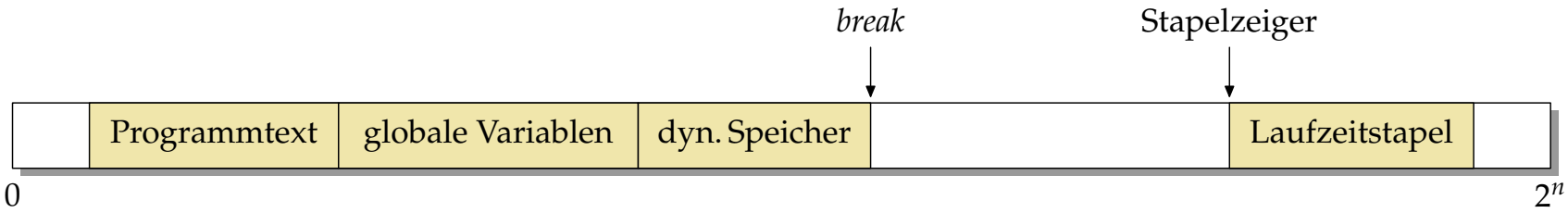
```
theon$ pmap 12175
12175:  endpause
0000000000400000          4K r-x----  /.../endpause
0000000000600000          8K rw-----  /.../endpause
FFFF80FFBF460000       1824K r-x----  /lib/amd64/libc.so.1
FFFF80FFBF638000         68K rw-----  /lib/amd64/libc.so.1
FFFF80FFBF649000          8K rw-----  /lib/amd64/libc.so.1
FFFF80FFBF68D000        352K r-x----  /lib/amd64/ld.so.1
FFFF80FFBF7B0000         24K rwx----    [ anon ]
FFFF80FFBF7C0000         64K rw-----    [ anon ]
FFFF80FFBF7D8000         12K r--s---    [ anon ]
FFFF80FFBF7E5000         16K rwx----  /lib/amd64/ld.so.1
FFFF80FFBF7E9000          4K rwx----  /lib/amd64/ld.so.1
FFFF80FFBF7EB000          4K r--s---    [ anon ]
FFFF80FFBF7FA000          4K r--s---    [ anon ]
FFFF80FFBFFFC000         16K rw-----    [ stack ]
         total         2408K
theon$
```

- Solange ein Prozess noch läuft, kann auf Solaris mit Hilfe des *pmap*-Programms der virtuelle Adressraum aufgelistet werden.
- Links steht in Hex jeweils die Anfangsadresse, dann in dezimal die Zahl der belegten Kilobyte, dann die Zugriffsrechte (r = *read*, w = *write*, x = *executable*) und schließlich, sofern vorhanden, die in den Adressraum abgebildete Datei.

Zur vierten Spalte:

.../endpause	Das ausführbare Programm, das gestartet wurde.
/lib/amd64/libc.so.1	Dynamisch ladbare C-Bibliothek, „so“ steht dabei für *shared object*, die „1“ dient als Versionsnummer.
/lib/amd64/ld.so.1	Dynamisches Ladeprogramm (war notwendig, um */lib/amd64/libc.so.1* zu laden).
[stack]	Bis zu einem gewissen Limit automatisch wachsender Stack.
[anon]	Durch die Speicherverwaltung belegter Speicher.

- Jede dynamische Speicherverwaltung benötigt einen Weg, mehr Speicher vom Betriebssystem anzufordern und diesen in einen bislang ungenutzten Bereich des virtuellen Adressraums abzubilden.
- Dafür gibt es im POSIX-Standard zwei Systemaufrufe:
  - *sbrk* – der traditionelle Ansatz: einfach, aber nicht flexibel
  - *mmap* – sehr flexibel, aber auch etwas komplizierter
- Gearbeitet wird in jedem Fall mit ganzen Kacheln.
- Eine Rückgabe von Kacheln an das Betriebssystem scheitert normalerweise an der Fragmentierung. Bei C findet das normalerweise nicht statt, d.h. der belegte Speicher wächst selbst bei einem gleichbleibenden Speichernutzungsumfang dank der zunehmenden Fragmentierung langsam aber stetig.



- Der Break ist eine vom Betriebssystem verwaltete Adresse, die mit Hilfe der Systemaufrufe *brk* und *sbrk* manipuliert werden kann.
- *brk* spezifiziert die absolute Position des Break, *sbrk* verschiebt diese relativ.
- Zu Beginn zeigt der Break auf den Anfang des Heaps, konventionellerweise liegt dieser hinter den globalen Variablen.
- Durch das Verschieben des Breaks zu höheren Adressen kann dann Speicher belegt werden.

trivial-alloc-break.c

```
void* alloc(size_t nbytes, size_t align) {
   static char* mem_break = 0;
   static char* mem_pos = 0;
   static int pagesize = 0;
   if (!pagesize) {
      mem_break = sbrk(0);
      mem_pos = mem_break;
      pagesize = getpagesize();
      size_t remainder = (uintptr_t) mem_break % pagesize;
      if (remainder) {
         char* oldbreak = sbrk(pagesize - remainder);
         if (oldbreak != mem_break) {
            return 0;
         }
         mem_break += pagesize - remainder;
      }
   }
   char* p = mem_pos;
   p = (char*) (((uintptr_t) p + align - 1) & ~(align - 1));
   if (p + nbytes > mem_break) {
      size_t increment = (nbytes + pagesize - 1) & ~(pagesize - 1);
      char* oldbreak = sbrk(increment);
      if (oldbreak == (char*)-1) {
         return 0;
      }
      mem_break = oldbreak + increment;
   }
   mem_pos = p + nbytes;
   return p;
}
```

trivial-alloc-break.c

```
if (!pagesize) {
   mem_break = sbrk(0);
   mem_pos = mem_break;
   pagesize = getpagesize();
   size_t remainder = (uintptr_t) mem_break % pagesize;
   if (remainder) {
      char* oldbreak = sbrk(pagesize - remainder);
      if (oldbreak != mem_break) {
         return 0;
      }
      mem_break += pagesize - remainder;
   }
}
```

- Der Break liegt nicht notwendigerweise auf einer Kachelkante. Es ist aber zu Beginn sinnvoll, *mem_break* auf eine solche zu setzen, denn dieser Bereich steht ohnehin bereits zur Verfügung.

trivial-alloc-break.c

```
if (p + nbytes > mem_break) {
   size_t increment = (nbytes + pagesize - 1) & ~(pagesize - 1);
   char* oldbreak = sbrk(increment);
   if (oldbreak == (char*)-1) {
      return 0;
   }
   mem_break = oldbreak + increment;
}
mem_pos = p + nbytes;
return p;
```

- Wenn der vorhandene Bereich nicht genügt, wird der Break herausgeschoben – um ein Vielfaches der Kachelgröße.
- Wie bei allen Systemaufrufen üblich, wird bei Fehlern -1 zurückgegeben selbst wenn es sich um Zeiger handelt.

trivial-alloc-mmap.c

```
void* alloc(size_t nbytes, size_t align) {
   static char* mem_end = 0;
   static char* mem_pos = 0;
   static int pagesize = 0;
   if (!pagesize) {
      pagesize = getpagesize();
   }
   char* p = mem_pos;
   p = (char*) (((uintptr_t) p + align - 1) & ~(align - 1));
   if (p + nbytes > mem_end) {
      size_t len = (nbytes + pagesize - 1) & ~(pagesize - 1);
      char* free_space = mmap(0, len,
         PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANON, -1, 0);
      if (free_space == MAP_FAILED) {
         return 0;
      }
      if (mem_end != free_space) {
         mem_pos = free_space;
         p = free_space;
      }
      mem_end = free_space + len;
   }
   mem_pos = p + nbytes;
   return p;
}
```

trivial-alloc-mmap.c

```
char* free_space = mmap(0, len,
   PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANON, -1, 0);
```

- Beim ersten Parameter kann *mmap* eine Adresse vorgeschlagen werden. Bei 0 wird das ganz dem System überlassen.
- Der zweite Parameter spezifiziert die Länge (sollte ein Vielfaches der Kachelgröße sein).
- Dann folgen die Zugriffsrechte. *PROT_READ* und *PROT_WRITE* geben Lese- und Schreibrechte, gewähren aber keine Ausführungsrechte.
- *MAP_PRIVATE* bedeutet, dass wir diesen Speicherbereich mit niemanden teilen.
- Mit *MAP_ANON* wird implizit */dev/zero* als abzubildende Datei gewählt. Dies gehört nicht zum Umfang von POSIX, wird aber weitgehend unterstützt (einschließlich Linux und Solaris).
- Die letzten beiden Parameter spezifizieren einen Dateideskriptor und einen Offset. In Verbindung mit *MAP_ANON* ist hier -1 und 0 anzugeben.