Übungen Betriebssysteme (BS)

U4 – Speicherverwaltung

https://sys.cs.tu-dortmund.de/DE/Teaching/SS2021/BS/

Peter Ulbrich

peter.ulbrich@tu-dortmund.de

https://sys.cs.tu-dortmund.de/EN/People/ulbrich/

technische universität
dortmund

arbeitsgruppe
systemsoftware

Agenda

- Besprechung Aufgabe 3: Deadlock
- Quizfragen zum Vorlesungsstoff
- Fortsetzung Grundlagen C-Programmierung
- Aufgabe 4: *Speicherverwaltung*
  - Wiederholung Buddy-Algorithmus
  - Dynamische Speicherverwaltung in C: malloc/free
  - Speicherverwaltung im Eigenbau: Best Fit mit Bitliste
  - Exkurs: Bitoperationen in C

Besprechung Aufgabe 3

- Foliensatz *Besprechung A3*

Quizfragen zum Vorlesungsstoff

- **Ist die folgende Aussage wahr oder falsch? Warum?**

1. Werden in einem Programm nur Semaphore zur Synchronisation verwendet, können keine Livelocks (aktives Warten) auftreten.

Antwort: In der Regel **wahr**.

Allerdings bei Semaphore, die explizit abgefragt werden können, problematisch: Hiermit kann man auch aktiv warten.

Quizfragen zum Vorlesungsstoff

- **Ist die folgende Aussage wahr oder falsch? Warum?**

2. Bei *Shortest-Process-Next (SPN)* besteht die Gefahr der Aushungerung.

Antwort: Wahr, da E/A-lastige Prozesse (mit kurzen CPU-Stößen) immer CPU-lastigen Prozessen vorgezogen werden.
→ mögliche Aushungerung der CPU-lastigen Prozesse

Quizfragen zum Vorlesungsstoff

- **Ist die folgende Aussage wahr oder falsch? Warum?**

3. Der *Backerei-Algorithmus* ist eine mögliche Implementierung von Semaphore in Betriebssystemen.

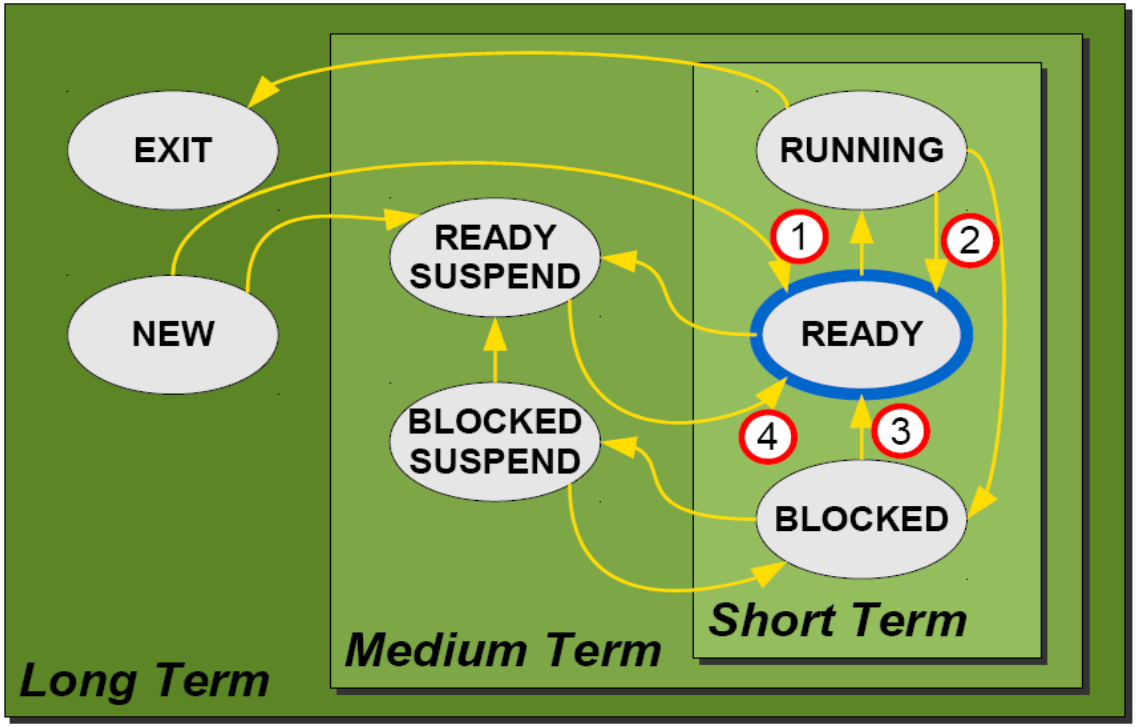
Antwort: Falsch, aktives Warten wird damit realisiert (Prozess zieht Wartenummer). Bei Semaphore wird den Prozessen (wenn wartend) Rechenzeit entzogen.

Quizfragen zum Vorlesungsstoff

- **Ist die folgende Aussage wahr oder falsch? Warum?**

4. Beim Short-Term-Scheduling kann ein Prozess direkt vom Zustand BLOCKED in den Zustand RUNNING wechseln.

Antwort: Falsch



Grundlagen C-Programmierung

- Foliensatz C-Einführung (Folie 47-52)

Buddy-Algorithmus

- Speicherplatzierungsstrategie
- sukzessives Halbieren des freien Speichers
  - bis zum „best-fit” der angeforderten Speichermenge

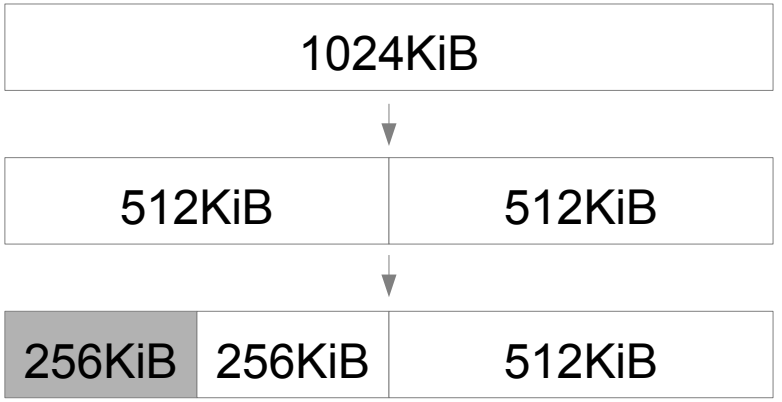
Buddy-Algorithmus: Reservierung

- Suche nach einem Speicherbereich, der die passende Größe hat → minimaler Block mit der Größe

  $2^n \geq$ angeforderter Speicher

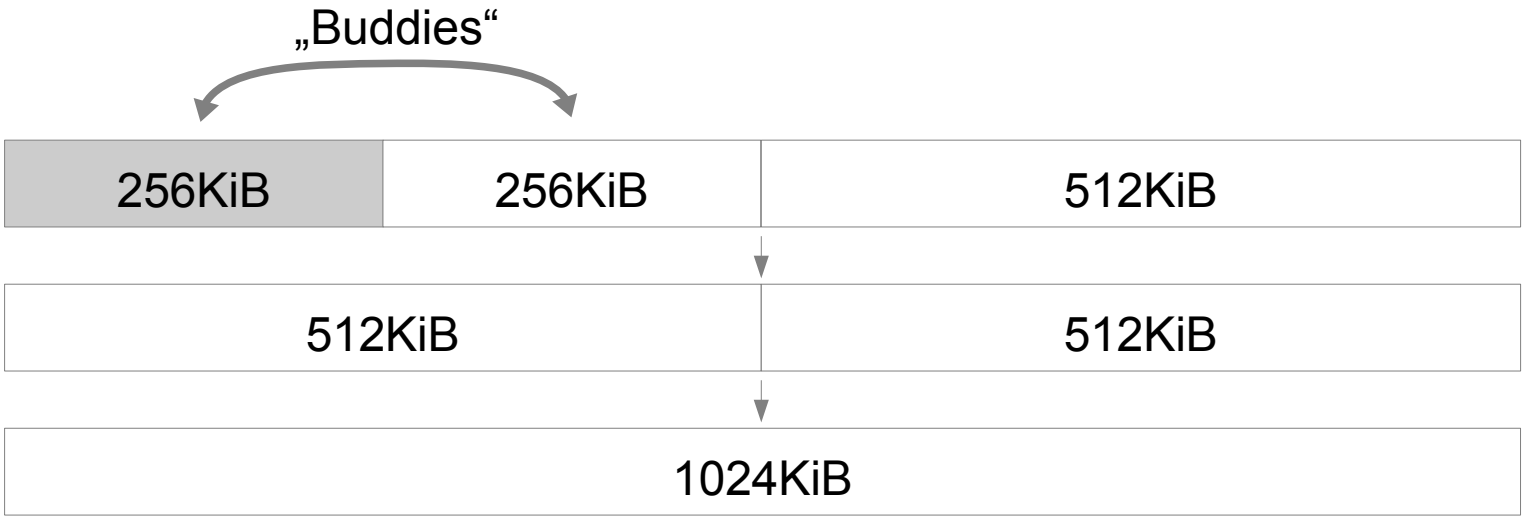
  - wird Speicherbereich der Größe 2^n gefunden → reservieren, Ende
  - Sonst versuche diesen wie folgt zu erzeugen
    1. teile einen freien Speicherbereich > 2^n (kleinstmöglich) in zwei Hälften
    2. ist eine Hälfte von der Größe 2^n (oder die untere Grenze erreicht) → reservieren, Ende
    3. gehe zu 1.

- **Beispiel:**
  Anforderung von 200 KiB
  → auf 2^n aufgerundet: 256 KiB



Buddy-Algorithmus: Freigabe

- Gebe den Speicherbereich frei und betrachte den angrenzenden Buddy
  - ist dieser ebenfalls nicht belegt, so fasse diese beiden zusammen
  - wiederhole die Zusammenfassung von Buddies, bis ein Speicherbereich belegt oder ganze Speicher freigegeben ist
- **Beispiel:**



Dynamische Speicherverwaltung in C

- **malloc** („memory alloc”): Standardbibliotheksfunktion, reserviert dynamisch Speicher auf dem *Heap*
- aus **malloc(3)**:
  - **void** *malloc(**size_t** size)
  - reserviert *size* Bytes
  - liefert einen Pointer auf den Anfang des Speicherbereichs oder im **Fehlerfall (!): NULL**
  - **size_t:** plattformunabhängiger Typ für Speicherbereichsgrößen (sizeof() ist z.B. auch von Typ **size_t**)
- **free**: gibt zuvor mit malloc (oder calloc/realloc) belegten Speicher wieder frei
  - **void** free(**void** *ptr)
  - Speicher darf nur **einmal** freigegeben werden!

Dynamische Speicherverwaltung in C

- Beispiel für malloc/free:

```
int *first_n_squares(unsigned n) {
   int *array, i;
   array = malloc(n * sizeof(int));   /* kein Cast notwendig */
   if (array == NULL) {               /* Fehlerbehandlung */
      perror("malloc");
      exit(EXIT_FAILURE);
   }
   for (i = 0; i < n; ++i)            /* Array befüllen */
      array[i] = i * i;
   return array;
} /* Die Variable "array" hört hier auf zu existieren – nicht
   * aber der Speicherbereich, auf den sie zeigt! */
int main(void) {
   int *ptr;
   /* ... */
   ptr = first_n_squares(200);
   printf("10 * 10 = %d", ptr[10];
   free(ptr);
   return 0;
}
```

Dynamische Speicherverwaltung in C

- ptr zeigt jetzt auf einen Speicherbereich der **Länge 42**

```
char *ptr = malloc(42);
```

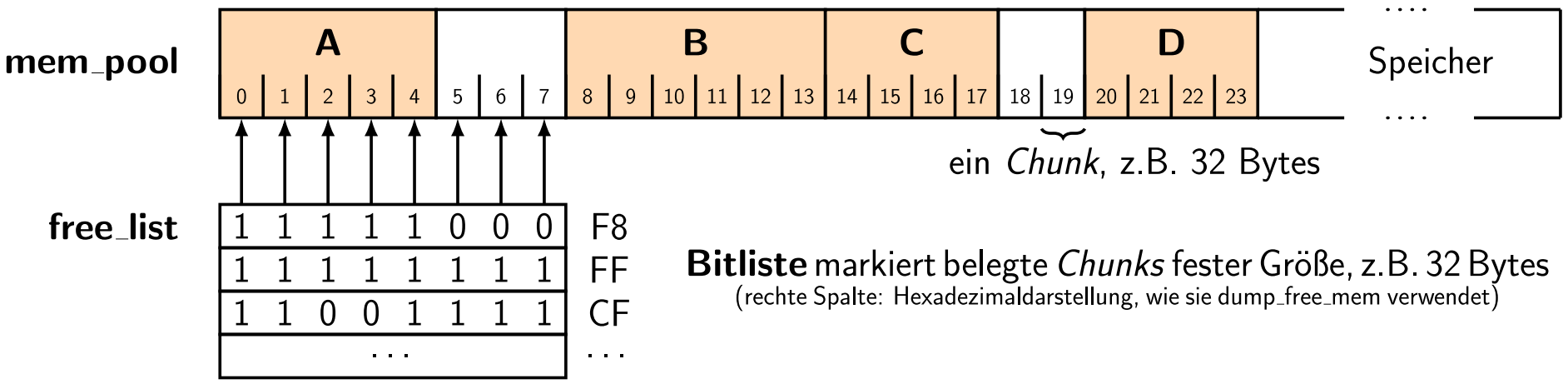
- Wie schreiben wir ein **int** mit dem Wert **0x12345678** an den Anfang?
  - ptr vom Typ **char** * („Zeiger auf **char**”)
  - (**int** *) ptr Cast auf den Typ **int** * („Zeiger auf **int**”)
  - und ab da wie üblich: Dereferenzieren (* davor), um den Wert „anfassen” zu können, auf den der Zeiger zeigt! (Klammerung)
  - *((**int** *) ptr) ist vom Typ **int** (**char** *, auf **int** * gecastet, dereferenziert)

```
*((int *)ptr) = 0x12345678;
```

  - **Quizfrage:** In welchem Bereich des *Speicherlayouts* landet der angeforderte Speicher?

A4: Speicherverwaltung im Eigenbau

- **void** *bf_alloc(**size_t** size): belegt Speicher (wie **malloc(3)**)
- **void** bf_free(**void** *ptr, **size_t** size): gibt frei ([fast] wie **free(3)**)
- ein großer Speicherpool (globales **char**-Array: *mem_pool*)
- eine Freispeicher-Bitliste (**char**-Array: *free_list*)
  → verwaltet belegte *Chunks*

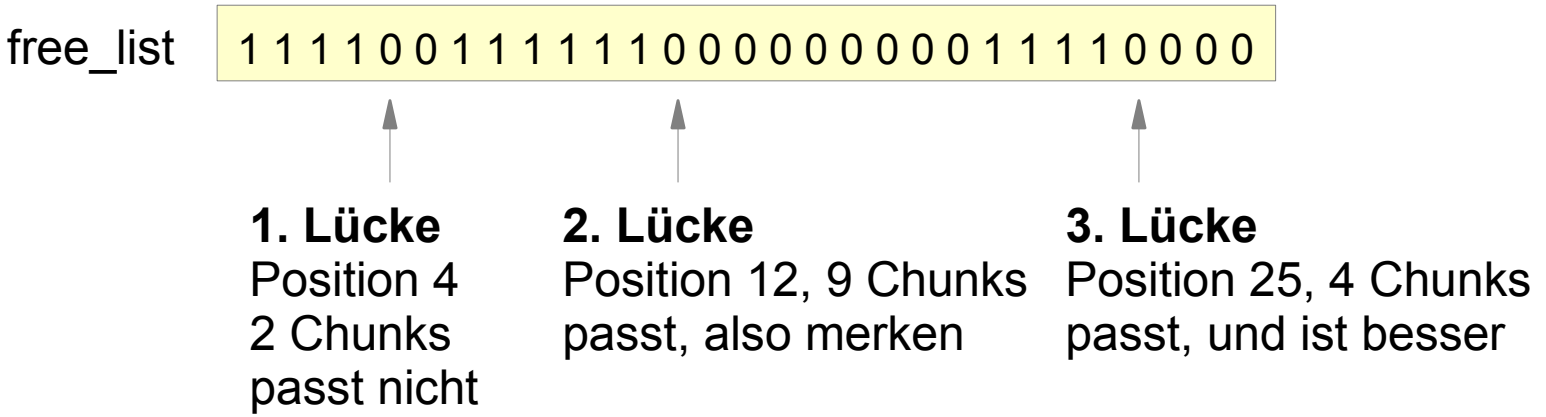


A4: Speicherverwaltung im Eigenbau

- **Best Fit** Strategie soll verwendet werden.
- Vorgehen beim Belegen: bf_alloc(**size_t** size)
  - Speichergröße in *Chunks* umrechnen (aufrunden, falls notwendig)
  - Freispeicher-Bitliste nach der am besten passenden Lücke durchsuchen
  - keine passende Lücke gefunden: NULL zurückliefern
  - Suche erfolgreich:
    - Speicherbereich belegen (Bits setzen!)
    - Adresse berechnen (beginnend bei mem_pool!) und zurückliefern
- Vorgehen beim Freigeben: bf_free(**void** *ptr, **size_t** size)
  - Adresse verwenden, um Nummer der *Chunks* zu berechnen
    - Differenz ((**char** *) ptr) – mem_pool verwenden!
  - Speicherbereich freigeben (Bits löschen!)

A4: Speicherverwaltung im Eigenbau

- Es gibt unterschiedliche Varianten, die beste Lücke zu finden
- eine Möglichkeit: Bitliste nach passender Lücke durchsuchen und dabei die besten Lücke merken.
- **Beispiel:** bf_alloc(70)
  - 70 Bytes benötigen 3 Chunks (je 32 Bytes)



  - Also verwende die Lücke an Position 25

Exkurs: Bitoperationen

- Wie **setzt/löscht/testet** man ein einzelnes Bit in einem Byte (bzw. unsigned char)?
- Zur Verfügung stehende Operationen:
  - bitweises UND (Operator in C: **&**)
  - bitweises ODER (in C: **|**)
  - bitweise NEGATION (in C: **~**)
  - SHIFT nach links/rechts (in C: **<<** bzw. **>>**)
  - [bitweises EXKLUSIV ODER (in C: **^**)]

```
unsigned char c = 0xF3;
c = c & 0x0F;
c = c | 0x10;
c = ~c;
C = c << 1;
```

```
Kurzschreibweise:
c &= 0x0F;
c |= 0x10;
c = ~c;
c <<= 1;
```

OR	1 1 1 1 1 0 0 0	F8
	0 0 0 0 0 0 1 0	02
=	1 1 1 1 1 0 1 0	FA

Exkurs: Bitoperationen

- Setzen eines Bits
  - Ver**oder**n von Eingabe und Bitfolge, in der nur das entsprechende Bit gesetzt ist: **11011001 | 00000100 = 11011101**
  - „Herstellen" der Bitfolge in C mit Linksshift **(1 << 2)**
- Löschen eines Bits
  - Ver**und**en von Eingabe und Bitfolge, in der nur das entsprechende Bit **nicht** gesetzt ist: **10011011 & 11101111 = 10001011**
  - „Herstellen" der Bitfolge in C mit Linksshift und Negation: **~(1 << 4)**
- Testen eines Bits
  - Ver**und**en von Eingabe und Bitfolge, in der nur das entsprechende Bit gesetzt ist: **11011001 & 01000000 = 01000000**
  - Ergebnis > 0: gesuchtes Bit in der Eingabe ist gesetzt
  - Ergebnis = 0: gesuchtes Bit **nicht** gesetzt

Exkurs: Bitoperationen vs. logische Op.

- **Bitoperationen:** &, |, ~, ^, <<, >>

  vs.

- **logische Operationen:** &&, ||, !
  - für XOR und Shift gibt es kein Logik-Äquivalent in C!

- Wie ist der Wert von c in den folgenden Zeilen?

```
int a=1, b=2, c;
c = a & b;   /* c = 0, da bitweises und */
c = a && b;  /* c = 1, da a und b != 0 */
c = ~b;      /* c = -3 (0xFFFFFFFD, Bit geflippet */
c = !b;      /* c = 0, da b = 1 */
c = !!b;     /* c = 1, da !b == 0 */
c = a << 2;  /* c = 4, da 0001 zu 0100 wird */
```

Exkurs: Bitoperationen vs. logische Op.

- Vorsicht mit der Operatoren-Rangfolge!
  - Was tut dieser Code?

```
if (a & 3 == 3) { ... }
if ((a & 3) == 3) { ... }
```

- „==” hat höhere Priorität als „&”
- **1. if-Abfrage:** vergleicht zuerst 3 und 3. Ergebnis 1
  - Dann wird 1 bitweise mit a verundet.
- **2. if-Abfrage:** verundet zuerst bitweise a mit 3
  - Dann wird das Ergebnis daraus mit 3 verglichen

Pointerarithmetik

- Abschließende **Quizfrage** zu Pointerarithmetik
- Was ist der Unterschied zwischen …

```
/* ptr hat einen gültigen Wert */
return ((char *) ptr) + 1;
```

- … und …

```
/* ptr hat einen gültigen Wert */
return ((int *) ptr) + 1;
```

- … ?
- **Antwort:** Bei Berechungen (Addition, Subtraktion, …) mit Zeigern hängt die Adressdifferenz vom Zeigertyp ab!
  - Erhöhung um sizeof(**char**) vs. sizeof(**int**)!