Übungen Betriebssysteme (BS)

U4 – Speicherverwaltung

Alwin Berger

TU Dortmund - AG Systemsoftware

Agenda

- Besprechung von A3
- Vorbesprechung zu A4
  - Dynamische Speicherverwaltung in C
  - Zeigerarithmetik
  - Casting
  - Fallstricke bei der Speicherverwaltung
  - Next Fit

Dynamische Speicherverwaltung in C

- Standardbibliotheksfunktion zur Allokation von Speicher:

```
void* malloc(size_t size)
```

- Reserviert zur Laufzeit Speicher auf dem Heap
- Das Argument **size** legt die Größe des zu reservierenden Bereichs in Bytes fest
- Rückgabewerte:
  - **≠ NULL**, wenn Allokation erfolgreich -> Rückgabewert ist Adresse
  - **= NULL**, wenn ein Fehler aufgetreten ist

Speicherplatz freigeben

- Gegenstück zu **malloc**:

```
void* free(void *ptr)
```

- Gibt Speicher an Adresse ptr wieder frei
- Speicher darf nur einmal freigegeben werden

Array im Heap

```c
int* first_n_squares(unsigned n);

int main(void) {
        int* ptr;
        ptr = first_n_squares(200);
        printf("10*10 = %d\n", ptr[10]) ;
        free(ptr);
        return 0;
}

int* first_n_squares(unsigned n) {
        int* array;
        array = malloc(n * sizeof(*array));
        if (array == NULL) {
                perror("malloc");
                exit(EXIT_FAILURE);
        }
        for (int i = 0; i < n; ++i) {
                array[i] = i * i;
        }
        return array;
}
```

Zeigerarithmetik

- Welche Werte haben **x** und **y**?

```
char *x = 0x42;
int *y = 0x42;

x++;
y++;
```

- **x** enthält die Adresse **0x43** und **y** den Wert **0x46**
- Bei Zeigerarithmetik hängt die Differenz vom Zeigertyp ab!

```
sizeof(char) /* vs */ sizeof(int)
```

Casting von Zeigern

- Wie schreibt man den Integer-Wert **0x12345678** in einen Speicherbereich der Größe 42?

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

- Der Zeiger **x** wird gecastet:
  - **x** ist ein Zeiger auf ein **char**
  - **(int*)(x)** castet x zu einen Zeiger auf ein **int**
  - Wie gewohnt dereferenzieren:

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

Fallstricke bei der Speicherallokation

- Wurden 0 Bytes angefragt?
- Wurden zu viele Bytes (größer als der gesamte Speicher) angefragt?
- Ist eine passende Lücke vorhanden?

Fallstricke bei der Speicherfreigabe

- Passt die Adresse überhaupt zu dem verwalteten Speicher?
- Wurde der Speicher schon freigegeben?
- Zusammenlegen von **Adjazenten** bzw. **freien Speicherbereichen**

Speicherplatzierungsstrategie ‘Next Fit’

- Sucht nächste passende Lücke im Speicher
- Merkt sich Position der letzten Allokation, setzt Suche bei nächster Anfrage dort fort
- Um Verwaltungskosten zu sparen, wird Speicher in Blöcken (Chunks) verwaltet:
  - Zum Beispiel **4096 Bytes** * **8 Blöcken** = **32 kB**

Speicherplatzierungsstrategie ‘Next Fit’ (Allokation)

- Metadaten pro Block (Chunk)
  - Status des Blocks: **frei** oder **belegt**
  - Länge des Blocks; falls 0, ist der Status irrelevant
- Suche nach freiem Bereich
  - Falls Bereich reserviert, um Anzahl Blöcke weiter springen
  - Wenn Ende des Speichers erreicht, dann Vorne beginnen

Ausgangszustand

	0	1	2	3	4	5	6	7
Heap								
Status	F							
Länge	8	→						

Allokation eines Blocks der Länge 3

	0	1	2	3	4	5	6	7
Heap	A	A	A					
Status	B	B	B	F				
Länge	3			5				

Speicherplatzierungsstrategie ‘Next Fit’ (Freigabe)

- Markiere Startblock als frei
- Angrenzende Bereich überprüfen
  - Bereich davor frei? → Verschmelzen
  - Bereich dahinter frei? → Verschmelzen

Vor der Freigabe von A

	0	1	2	3	4	5	6	7
Heap	A	A	A					
Status	B	B	B	F				
Länge	3	→	→	5	→	→	→	→

Freigabe von A

	0	1	2	3	4	5	6	7
Heap								
Status	F			F				
Länge	3	→	→	5	→	→	→	→

Verschmelzung

	0	1	2	3	4	5	6	7
Heap								
Status	F							
Länge	8	→						

Was haben wir heute gelernt?

- Zusammenfassung
  - Speicherverwaltung mit **malloc** und **free**
  - Etwas Zeigerarithmetik
  - ‘Next Fit’-Algorithmus