Memory Organization

Compiler Construction

2024-12-03

Prof. Dr. Michael Kuhn
michael.kuhn@ovgu.de

Parallel Computing and I/O
Institute for Intelligent Cooperating Systems
Faculty of Computer Science
Otto von Guericke University Magdeburg
https://parcio.ovgu.de

Outline

- Program memory is typically segmented
- Segments include code, data, heap and stack
- Memory is allocated in pages (of typically 4 KiB)

Outline

Group Exercise

- Draw chunks and page boundaries after each allocation with a page size of 4,000 bytes and an initial heap of one page.

```
int* ptr1 = malloc(76);
int* ptr2 = malloc(9976);
int* ptr3 = malloc(76);
free(ptr3);
free(ptr2);
free(ptr1);
```

- Draw chunks and page boundaries after each allocation with a page size of 4,000 bytes and an initial heap of one page.

100	(3,900)				
100	3,900	4,000	2,100	(1,900)	
100	3,900	4,000	2,100	100	(1,800)

Group Exercise

- Draw chunks and page boundaries after each allocation, the loop and each deallocation with a page size of 4,000 bytes and an initial heap of one page.

```c
int* ptr1 = malloc(76);
int* ptr2 = malloc(9976);
...
for (int i = 0; i < 20; i++) {
    ptr1[i] = 0;
}
...
free(ptr2);
free(ptr1);
```

- Draw chunks and page boundaries after each allocation, the loop and each deallocation with a page size of 4,000 bytes and an initial heap of one page.

100	(3,900)			
100	3,900	4,000	2,100	(1,900)
100	(3,900)	4,000	2,100	(1,900)
100	↯	4,000	2,100	(1,900)

- Draw chunks and page boundaries after each allocation, the loop and each deallocation with a page size of 4,000 bytes and an initial heap of one page.

```
int* ptr1 = malloc(76);
int* ptr2 = malloc(9976);
...
for (int i = 0; i < 20; i++) {
    ptr1[i] = 0;
}
...
int* ptr3 = malloc(76);
...
free(ptr3);
free(ptr2);
free(ptr1);
```

- Draw chunks and page boundaries after each allocation, the loop and each deallocation with a page size of 4,000 bytes and an initial heap of one page.

100	(3,900)					
100	3,900		4,000	2,100	(1,900)	
100	(3,900)		4,000	2,100	(1,900)	
100	100	(3,800)	4,000	2,100	(1,900)	

- Draw chunks and page boundaries at the program's end with a page size of 4,000 bytes for best fit, worst fit, first fit and next fit strategies. Count number of checks.

```
int* ptr1 = malloc(76);
int* ptr2 = malloc(976);
int* ptr3 = malloc(76);
int* ptr4 = malloc(76);
int* ptr5 = malloc(76);
free(ptr2);
free(ptr4);
// Same until here (count checks starting here)
int* ptr4 = malloc(76);
int* ptr2 = malloc(976);
```

- Draw chunks and page boundaries at the program's end with a page size of 4,000 bytes for best fit, worst fit, first fit and next fit strategies. Count number of checks.

100	(1000)	100	(100)	100

Best fit (5 + 5):

100	1000	100	100	100

Worst fit (5 + 6):

100	100	(900)	100	(100)	100	1000

First fit (2 + 6):

100	100	(900)	100	(100)	100	1000

Next fit (2 + 4):

100	100	(900)	100	(100)	100	1000

Group Exercise

- What happens on a system with 4 GiB RAM?

```
int* ptr1 = malloc(8000000000);
```

Group Exercise

- What happens on a system with 4 GiB RAM?

```c
int* ptr1 = malloc(8000000000);
```

- Pages are typically only allocated when they are accessed (page fault)
- Touching pages when allocating a large chunk might be problematic

Group Exercise

- What happens in the following code snippet?

```
int foo (int a) {
    int bar[1];
    bar[1] = 42;
    bar[2] = 42;
    bar[3] = 42;
}
```

Group Exercise

- What happens in the following code snippet?

```c
int foo (int a) {
    int bar[1];
    bar[1] = 42;
    bar[2] = 42;
    bar[3] = 42;
}
```

- `bar[1]` is outside the array and overwrites stack memory
- Might only be visible when stack smashing protection is enabled
- Different effects depending on data type (32 vs. 64 bits) etc.

Group Exercise

- What happens with a limited or an unlimited stack?

```c
int recinc (int a) {
    return recinc(a + 1);
}
int main (void) {
    recinc(0);
    return 0;
}
```

Group Exercise

- What happens with a limited or an unlimited stack?

```c
int recinc (int a) {
    return recinc(a + 1);
}
int main (void) {
    recinc(0);
    return 0;
}
```

- Limited stack: Program crashes after a certain number of recursions
- Unlimited stack: Program will likely be killed by out of memory killer

Outline

Memory Organization

- Overview
- Memory Allocation
- Summary

Summary

- Program memory is divided into logical segments
  - Code and data are determined at compile time
  - Heap and stack are controlled at runtime
- Page size influences overhead of page management
  - Huge pages can help reduce overhead
- Heap and stack management can be fragile
  - Overwriting metadata can lead to weird behavior or crashes

References

[Thain, 2020] Thain, D. (2020). ***Introduction to Compilers and Language Design: Second Edition.*** `http://compilerbook.org/`.