Programación de Sistemas CCPG1008

Federico Domínguez, PhD.

Unidad 3 – Sesión 3: Gestión de memoria

Agenda

Espacio de memoria virtual

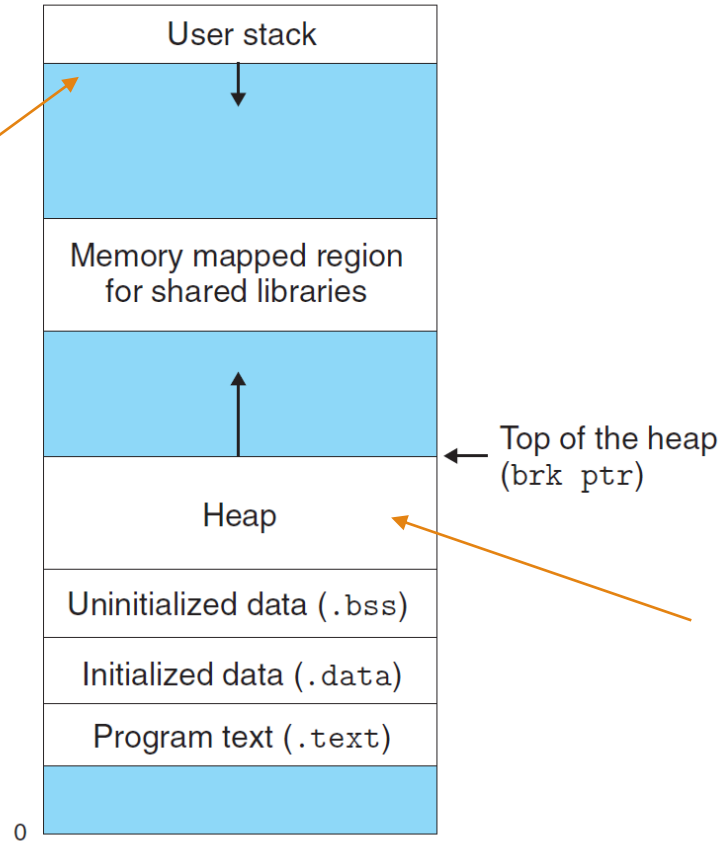
Memoria dinámica

Errores comunes de gestión de memoria

Demostración

Espacio de memoria de un proceso

El *stack*, reservadao para llamadas a funciones. Contiene variables locales, parámetros, valores de retorno, dirección de retorno, …



El *heap*, usado para la reserva de bloques en la memoria dinámica.

Memoria dinámica

Cuando es necesario usar más memoria en tiempo de ejecución es conveniente usar la memoria dinámica.

La memoria dinámica es obtenida del “montón” o *heap*.

Existen dos métodos genéricos de gestionar la memoria dinámica:

- **Explicito**: El programador se encarga de reservar y liberar espacios en la memoria dinámica. Así es en C y C++.
- **Implícito**: Un proceso gestionador de memoria detecta cuando un bloque reservado ya no es usado y lo libera automáticamente. También conocido como *garbage collection* (Java y otros lenguaje de alto nivel)

Las funciones **malloc** y **free** son los gestionadores de memoria más usados en C.

```
#include <stdlib.h>

void *malloc(size_t size);
```

Returns: ptr to allocated block if OK, NULL on error

```
#include <stdlib.h>

void free(void *ptr);
```

Returns: nothing

La función **malloc** retorna un puntero a un bloque de memoria de al menos *size* bytes.

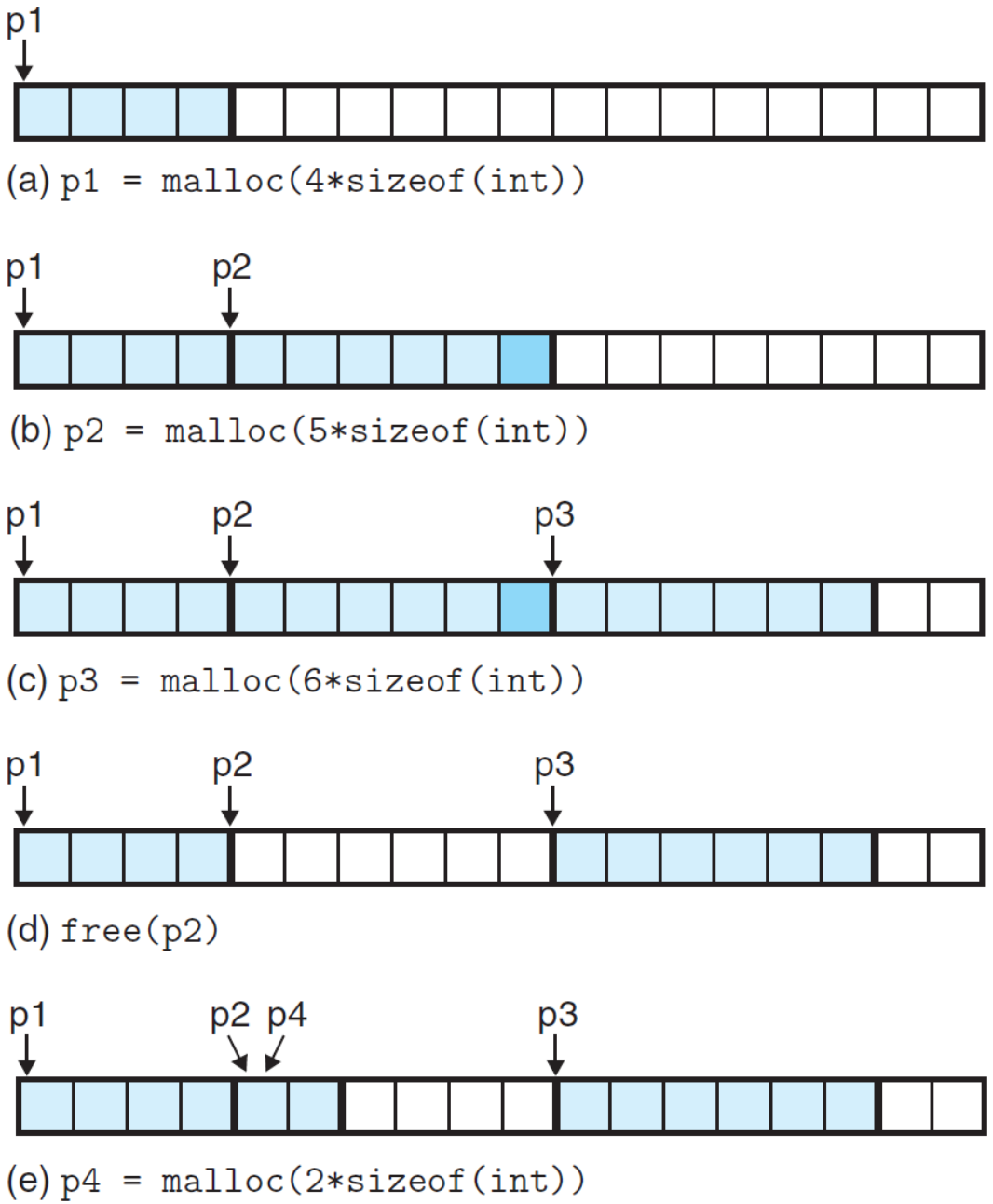
Una llamada exitosa a *malloc* separa un bloque de memoria dinámica en el *heap* correctamente alineado para contener cualquier tipo de datos. En una arquitectura de 64 bits, esta alineación es siempre en un borde de 16 bytes (*double word*).

```
#include <stdlib.h>

void *malloc(size_t size);
```

Returns: ptr to allocated block if OK, NULL on error

En esta imagen, cada bloque es de 4 bytes.



¿Por qué usar la memoria dinámica?

Es un uso más eficiente de memoria, es escalable y más fácil de mantener.

Evita el uso de estructura de datos de tamaño fijo, algo que en lo posible se debe evitar.

```
1   #include "csapp.h"
2   #define MAXN 15213
3
4   int array[MAXN];
5
6   int main()
7   {
8       int i, n;
9
10      scanf("%d", &n);
11      if (n > MAXN)
12          app_error("Input file too big");
13      for (i = 0; i < n; i++)
14          scanf("%d", &array[i]);
15      exit(0);
16  }
```

¿Por qué usar la memoria dinámica?

En este ejemplo, el programa reserva la cantidad justa de memoria que necesita. Esto solo lo puede saber en tiempo de ejecución ya que depende de información proporcionada por el usuario.

```
1   #include "csapp.h"
2
3   int main()
4   {
5       int *array, i, n;
6
7       scanf("%d", &n);
8       array = (int *)Malloc(n * sizeof(int));
9       for (i = 0; i < n; i++)
10          scanf("%d", &array[i]);
11      exit(0);
12  }
```

Llamadas a *malloc* **NO** inicializan la memoria reservada, si se desea hacer esto, se puede usar *calloc*.

```
void* calloc (size_t num, size_t size);
```

Llamadas a calloc inicializan la memoria reservada con ceros.

Ayuda a evitar errores de gestión de memoria a costa de menor rendimiento.

Errores comunes de gestión de memoria

Leer bloques no inicializados…

```
1   /* Return y = Ax */
2   int *matvec(int **A, int *x, int n)
3   {
4       int i, j;
5
6       int *y = (int *)Malloc(n * sizeof(int));
7
8       for (i = 0; i < n; i++)
9           for (j = 0; j < n; j++)
10              y[i] += A[i][j] * x[j];
11      return y;
12  }
```

Errores comunes de gestión de memoria

Desborde de buffers…

```
1    void bufoverflow()
2    {
3        char buf[64];
4
5        gets(buf); /* Here is the stack buffer overflow bug */
6        return;
7    }
```

Errores comunes de gestión de memoria

Asumir que los punteros son del mismo tamaño que los objetos a donde apuntan…

```
1   /* Create an nxm array */
2   int **makeArray1(int n, int m)
3   {
4       int i;
5       int **A = (int **)Malloc(n * sizeof(int));
6
7       for (i = 0; i < n; i++)
8           A[i] = (int *)Malloc(m * sizeof(int));
9       return A;
10  }
```

Errores comunes de gestión de memoria

Errores de pasarse por uno…

```
1   /* Create an nxm array */
2   int **makeArray2(int n, int m)
3   {
4       int i;
5       int **A = (int **)Malloc(n * sizeof(int *));
6
7       for (i = 0; i <= n; i++)
8           A[i] = (int *)Malloc(m * sizeof(int));
9       return A;
10  }
```

Errores comunes de gestión de memoria

No se entiende la aritmética de punteros …

```
1    int *search(int *p, int val)
2    {
3        while (*p && *p != val)
4            p += sizeof(int);
5        return p;
6    }
```

Errores comunes de gestión de memoria

Referenciar a variables que no existen …

```
1    int *stackref ()
2    {
3        int val;
4
5        return &val;
6    }
```

Errores comunes de gestión de memoria

Referenciar variables liberadas…

```
1   int *heapref(int n, int m)
2   {
3       int i;
4       int *x, *y;
5
6       x = (int *)Malloc(n * sizeof(int));
7
8       /* ... */   /* Other calls to malloc and free go here */
9
10      free(x);
11
12      y = (int *)Malloc(m * sizeof(int));
13      for (i = 0; i < m; i++)
14          y[i] = x[i]++;  /* Oops! x[i] is a word in a free block */
15
16      return y;
17  }
```

Errores comunes de gestión de memoria

Fugas de memoria (*memory leaks*)

```
1    void leak(int n)
2    {
3        int *x = (int *)Malloc(n * sizeof(int));
4
5        return;  /* x is garbage at this point */
6    }
```

Demostración

Para la próxima clase...

Lectura:

- Computer Systems, Bryant y O'Hallaron. Secciones 9.9.0 – 2 y 9.11
- Control de lectura!!

Práctica 7: Gestión de memoria y uso de gdb