Facultad de Ingeniería en Electricidad y Computación

Programación de Sistemas CCPG1051

Federico Domínguez, PhD.

Unidad 3 – Sesión 3: Memoria estática y dinámica

Agenda

Espacio de memoria virtual

Memoria estática

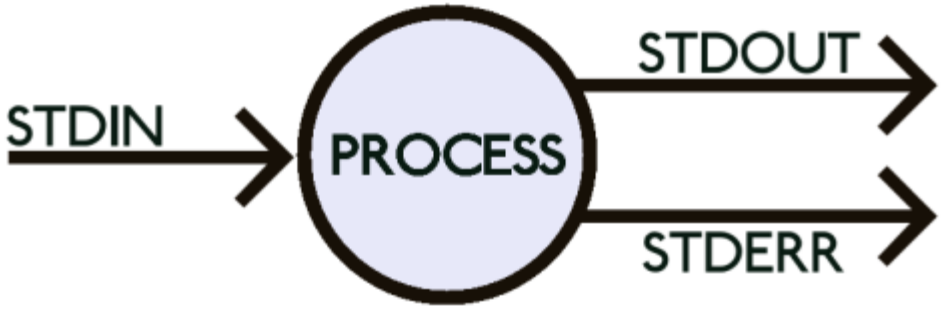
Memoria dinámica

Errores comunes de gestión de memoria

Demostración

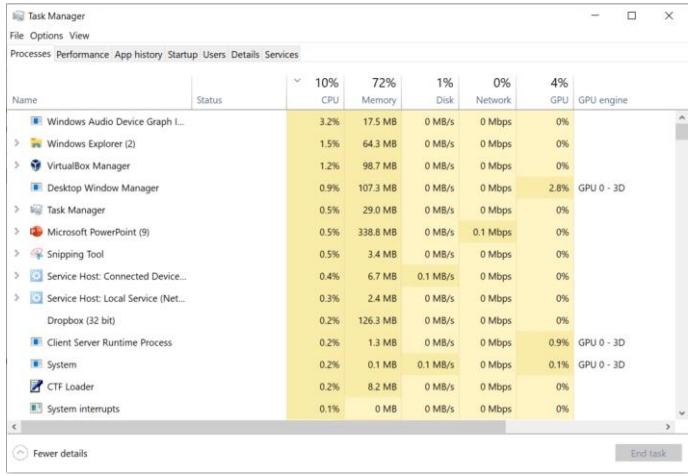
Un **proceso** es un programa en ejecución.

En la mayoría de sistemas operativos un proceso tiene tres standard streams: **stdin, stdout, stderr**



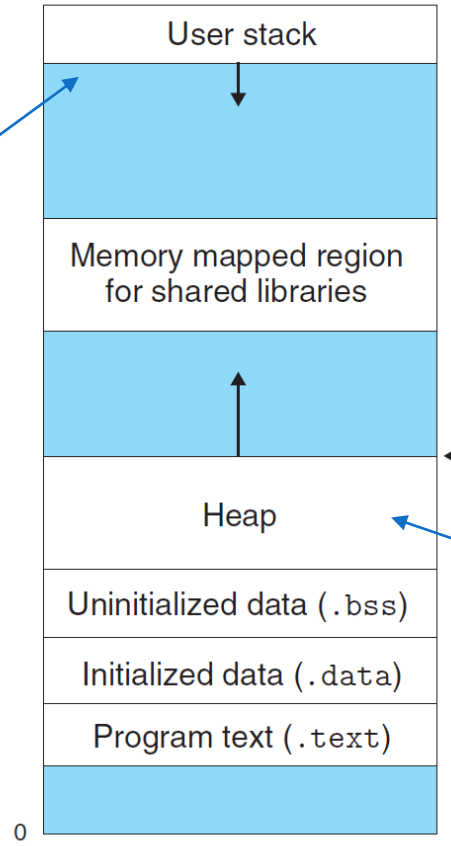

```
top - 16:16:17 up  5:26,  1 user,  load average: 0.06, 0.05, 0.00
Tasks: 198 total,   1 running, 197 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.8 us,  0.3 sy,  0.0 ni, 98.5 id,  0.3 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :   3936.2 total,   2052.9 free,    883.4 used,    999.8 buff/cache
MiB Swap:    448.5 total,    448.5 free,      0.0 used.   2786.5 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   1393 federico  20   0 4282568 472404 135204 S   1.0  11.7   1:25.73 gnome-shell
     11 root      20   0       0      0      0 I   0.3   0.0   0:01.23 rcu_sched
   1216 federico  20   0  100012   2144   1764 S   0.3   0.1   0:39.69 VBoxClient
   5179 federico  20   0   20608   4016   3416 R   0.3   0.1   0:00.03 top
      1 root      20   0  102272  11760   8520 S   0.0   0.3   0:02.36 systemd
      2 root      20   0       0      0      0 S   0.0   0.0   0:00.01 kthreadd
      3 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_gp
      4 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_par_gp
      6 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker/0:0H-kblockd
      9 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 mm_percpu_wq
     10 root      20   0       0      0      0 S   0.0   0.0   0:00.10 ksoftirqd/0
     12 root      rt   0       0      0      0 S   0.0   0.0   0:00.13 migration/0
     13 root     -51   0       0      0      0 S   0.0   0.0   0:00.00 idle_inject/0
     14 root      20   0       0      0      0 S   0.0   0.0   0:00.00 cpuhp/0
```



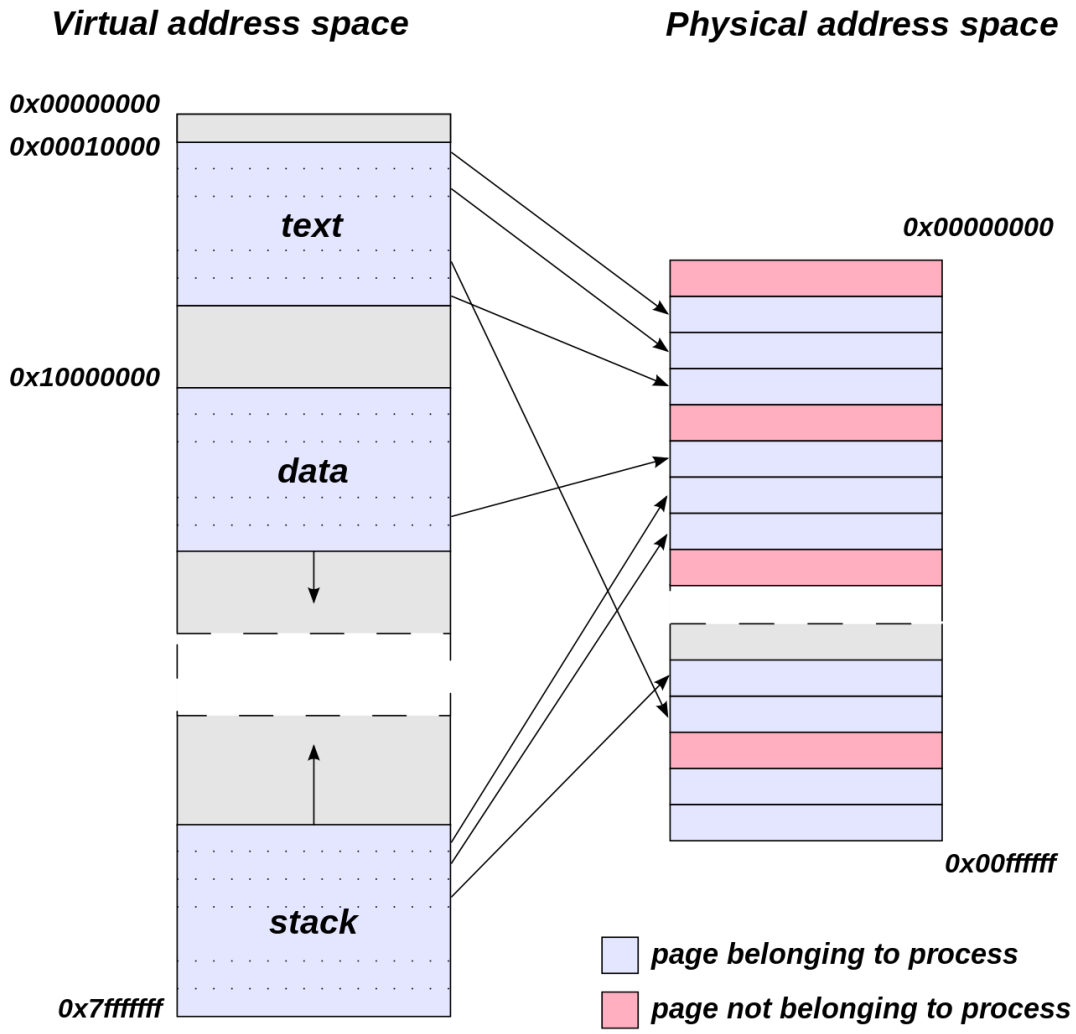
Modelo espacio de memoria virtual de un proceso

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



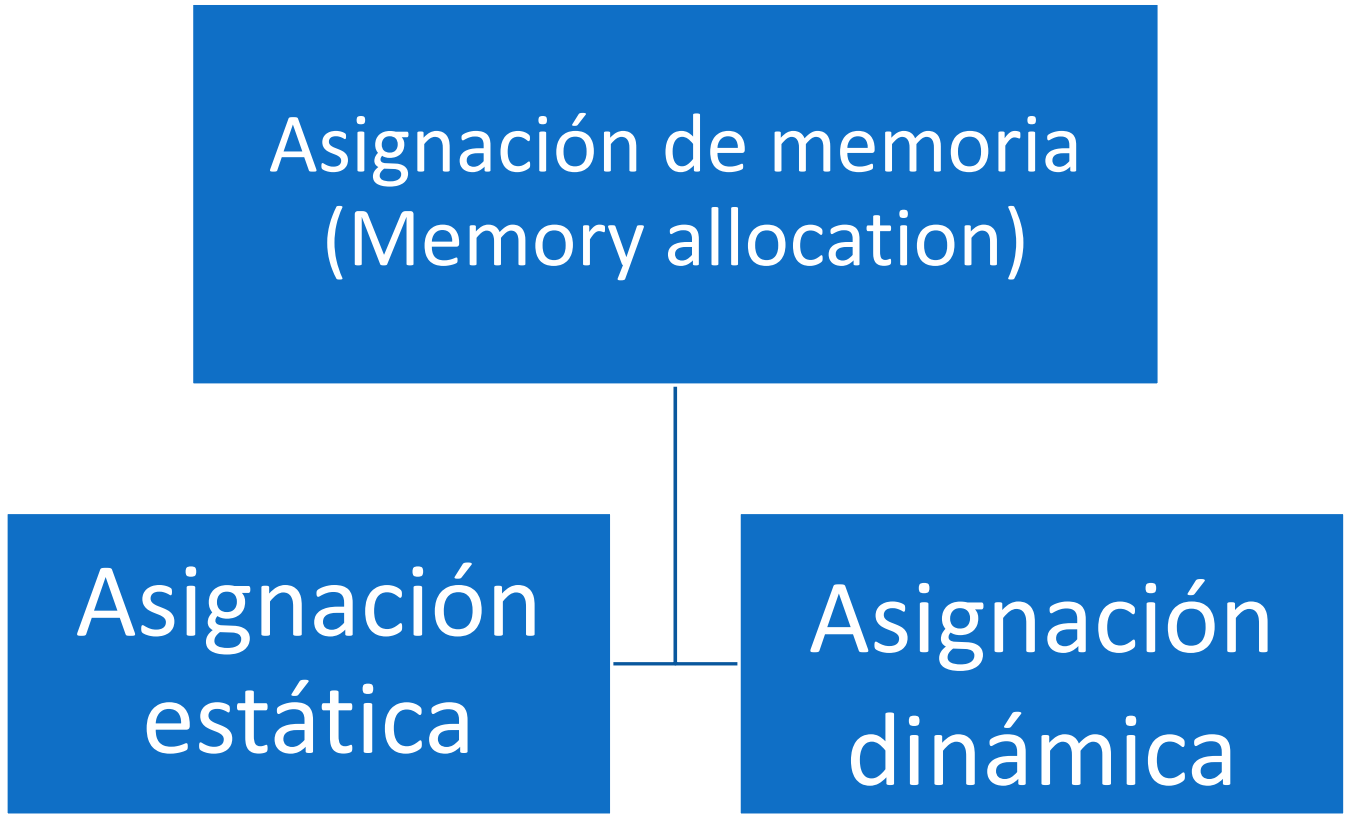
El *heap*, usado para la reserva de bloques en la memoria dinámica.

El espacio de memoria virtual usado en un proceso es “mapeado” a espacios o páginas en la memoria RAM del sistema. Esto lo hace el sistema operativo.



Un programa, y luego su(s) respectivo(s), procesos necesitan **asignar** o **reservar** memoria RAM para manejar sus datos.

Existen, en general, dos tipos de asignación de memoria: **estática** y **dinámica**.



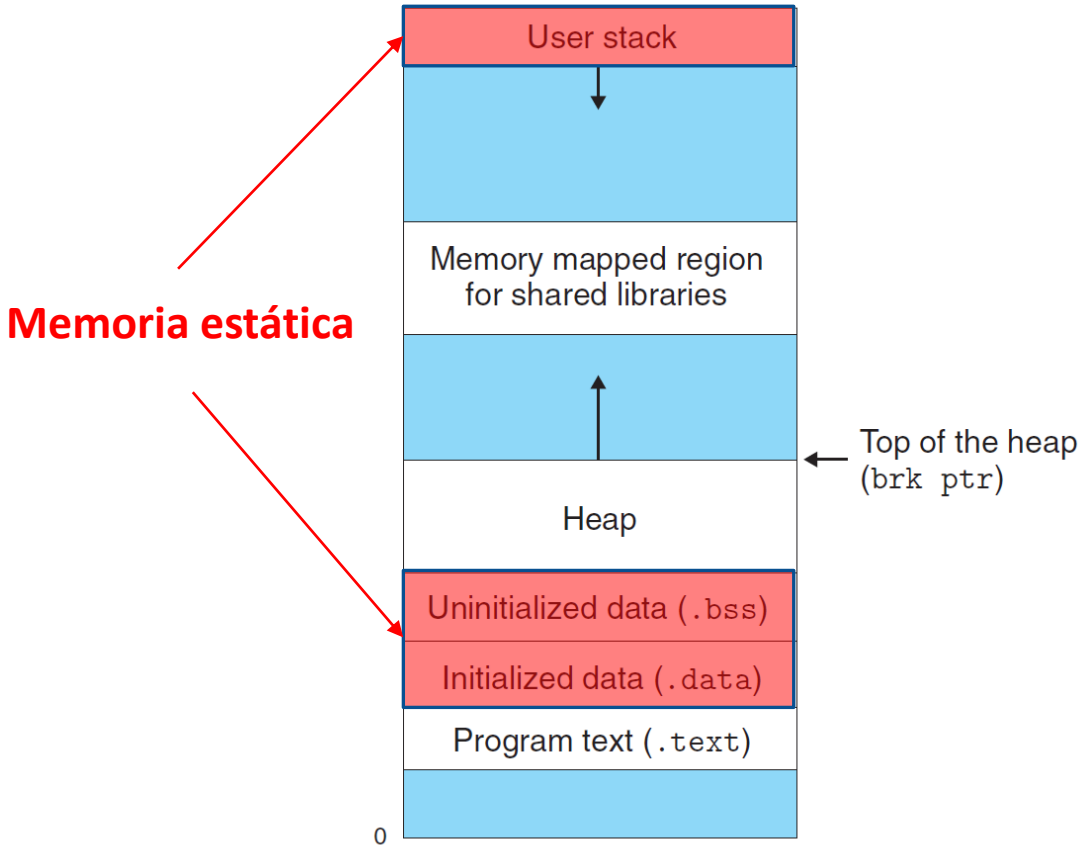
La asignación estática de memoria es efectuada en **tiempo de compilación**.

Las secciones de memoria .bss y .data son asignadas en tiempo de compilación.

Cada función tiene un espacio definido (stack frame) en tiempo de compilación.

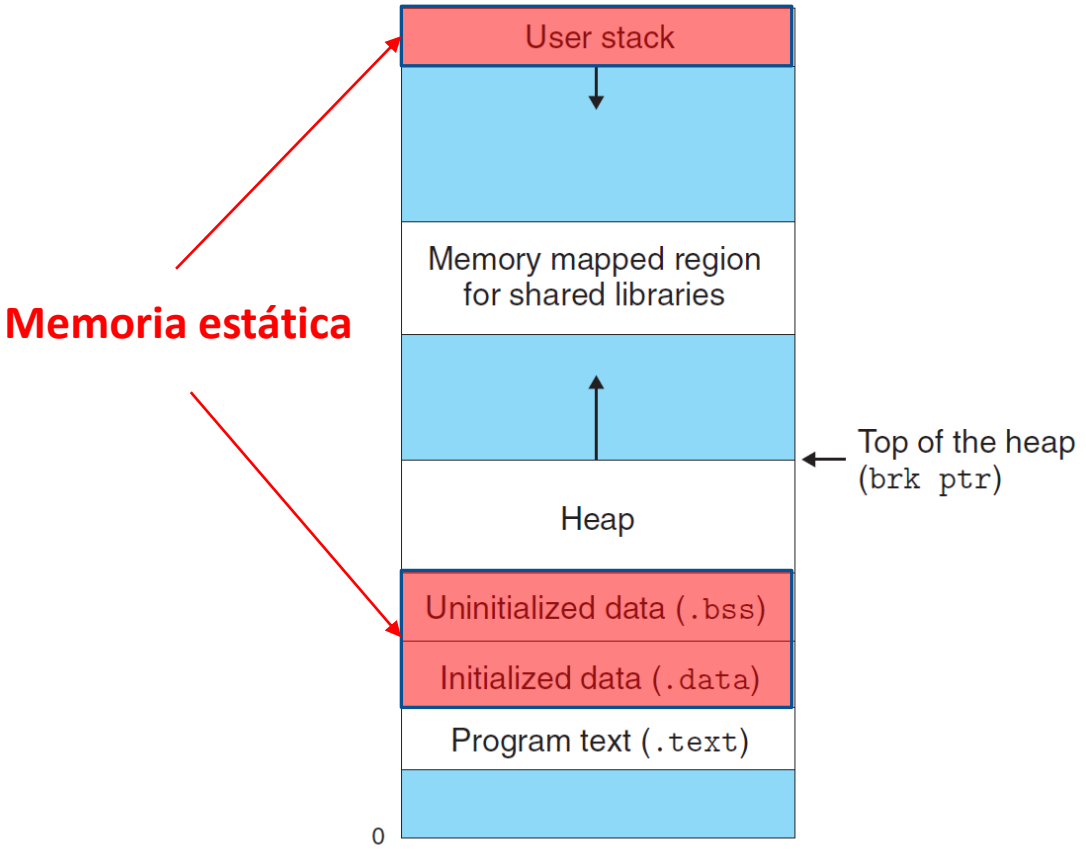
Variables en memoria estática:

- Variables globales inicializadas (.data)
- Variables globales no inicializadas (.bss)
- Variables locales (stack)
- Variables locales declaradas con **static** (.bss o .data)
- ¿ Constantes declaradas con **#define** ?



La asignación estática de memoria es efectuada en **tiempo de compilación**.

```
#define MB 100
#define ITER 50
int var = ITER;
int main(int argc, char **argv)
{
        char *ptr;
        int i,j;
        pid_t pid = getpid();
        printf("Proceso heapleak creado con PID: %d\n",pid);
        for(i=1;i<var;i++){
                printf("Vamos reservando %d MB.\n",MB*i);
                ptr = (char *) malloc(MB*1024*1024);
                for(j=0; j < MB*1024*1024; j++)
```

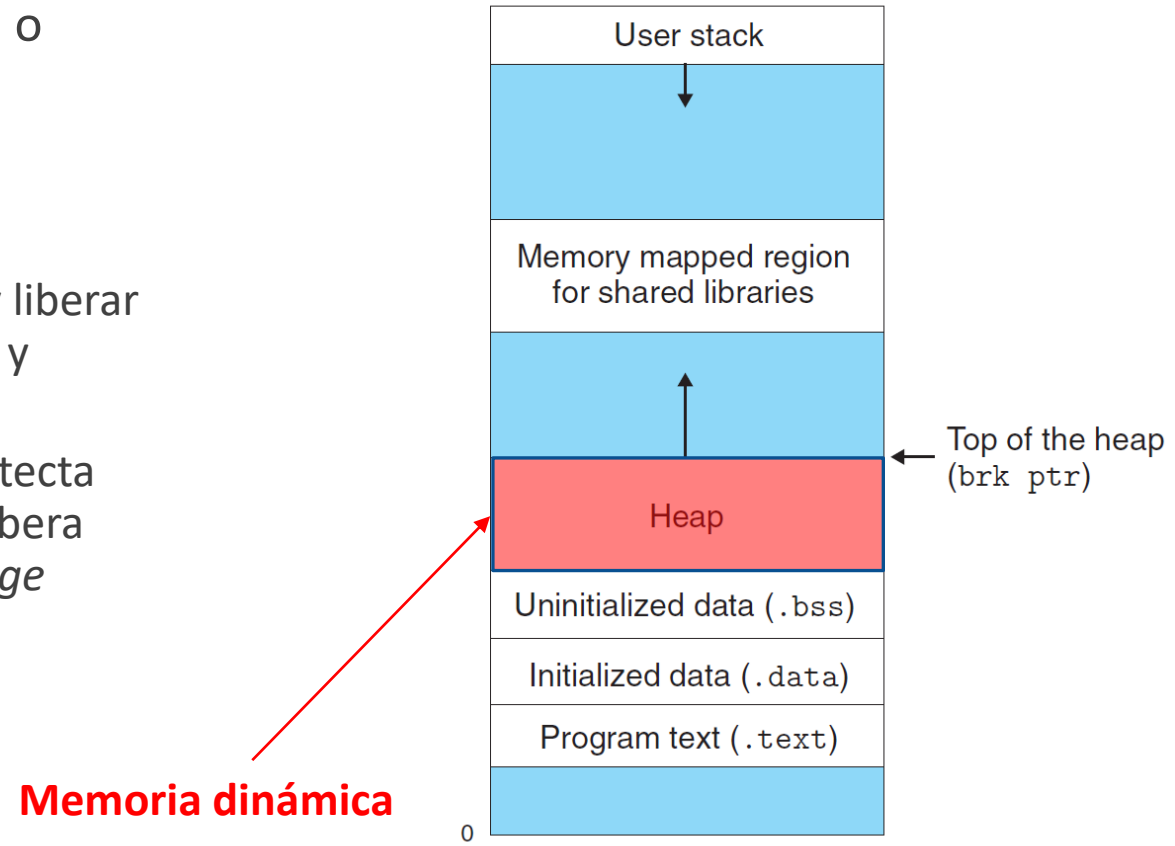


La asignación dinámica de memoria es efectuada en **tiempo de ejecución**.

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

Existen dos métodos genéricos de gestionar la memoria dinámica:

- **Explícito**: El programador se encarga de reservar y liberar espacios en la memoria dinámica. Así es en C/C++ y típicamente en sistemas embebidos.
- **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*).

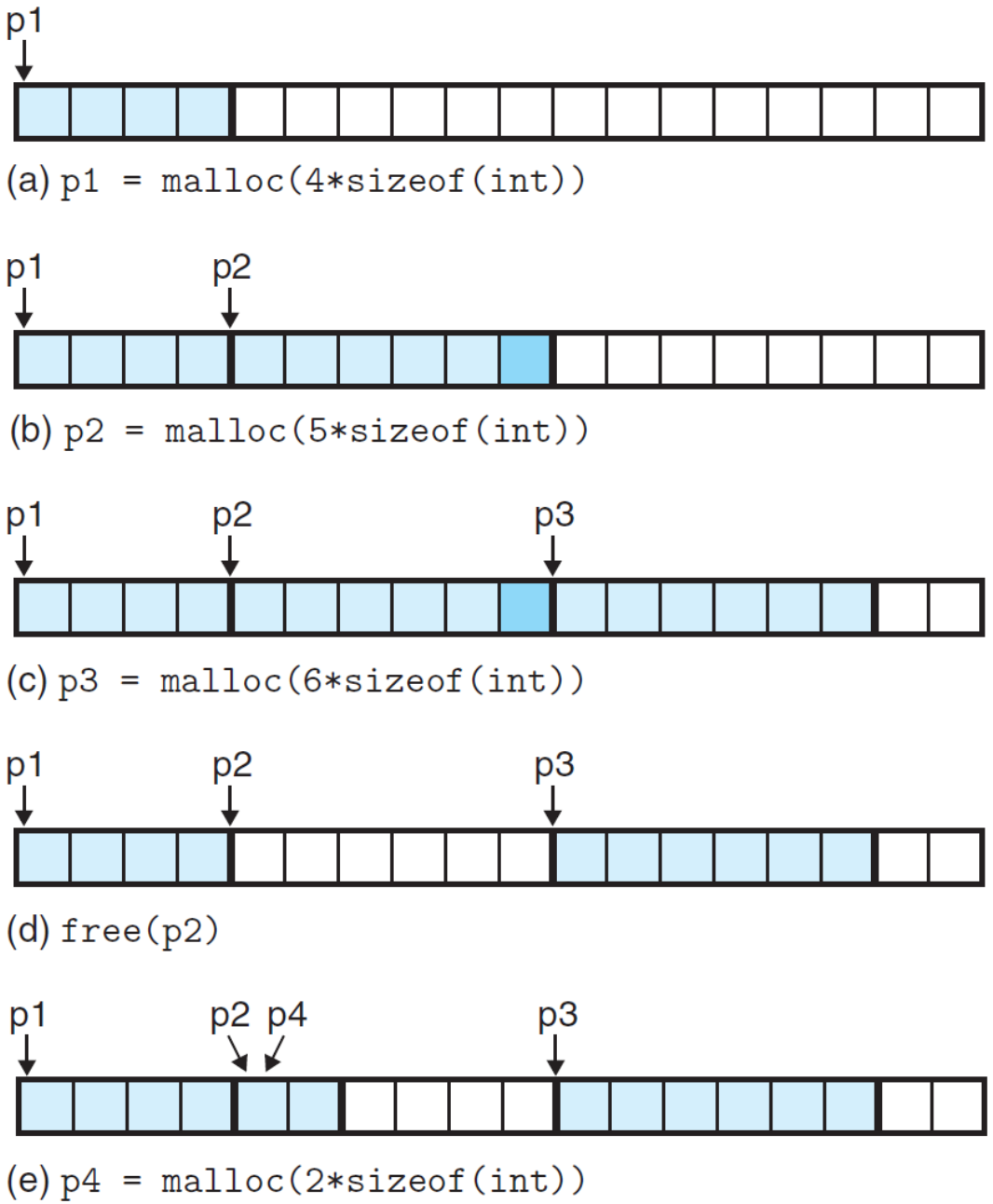
```
#include <stdlib.h>

void *malloc(size_t size);
```

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

En esta imagen, cada bloque es de 4 bytes.

La arquitectura es de 32-bits, por lo tanto la alineación es en **8 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 estructuras 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

Referencias

Computer Systems, Bryant y O’Hallaron. Secciones 9.9.0 – 2 y 9.11