Práctica de Organización del Computador II

Programación orientada a datos - Cuarta parte

Primer Cuatrimestre 2024

Organización del Computador II
DC - UBA

Introducción

Lo que sigue

Ahora vamos a ver:

- Uso del **stack** y del **heap**.
- Manejo de memoria dinámica.
- Estructura completa del espacio de memoria de una aplicación.

Visión de la memoria (vista de aplicación)

Naturaleza temporal de los datos

Vamos a introducir otra categoría para los datos. Ya habíamos estudiado al dato según:

- **Tipo:** esto lo vieron nativamente en Orga 1 (enteros con y sin signo, flotantes) y en alto nivel aquí (structs, chars, arrays, punteros).
- **Estructura:** que es lo que estuvimos explicando en la primera parte de la clase, o sea, su representación en memoria.
- Pero falta estudiar otra categoría que la **temporal**, que determina cuándo se crea, en qué contexto y cuánto tiempo está disponible dentro de la ejecución.
- Vamos a ver que los datos puede ser **estáticos**, **dinámicos** o **temporales**.

Naturaleza temporal de los datos

¿Cuál es la diferencia entre datos **dinámicos** y su complemento, los datos **estáticos**? ¿Qué son los datos temporales?

- **Los datos estáticos** se definen, conocen y potencialmente inicializan al momento de compilar el programa.
- **Los datos dinámicos** dependen de la ejecución del programa, pueden variar en cantidad o tamaño y su tiempo de vida puede ser disninto del tiempo de vida de la aplicación.
- **Los datos temporales** se definen y viven dentro del contexto de ejecución de una función (entre su `CALL` y su `RET`).

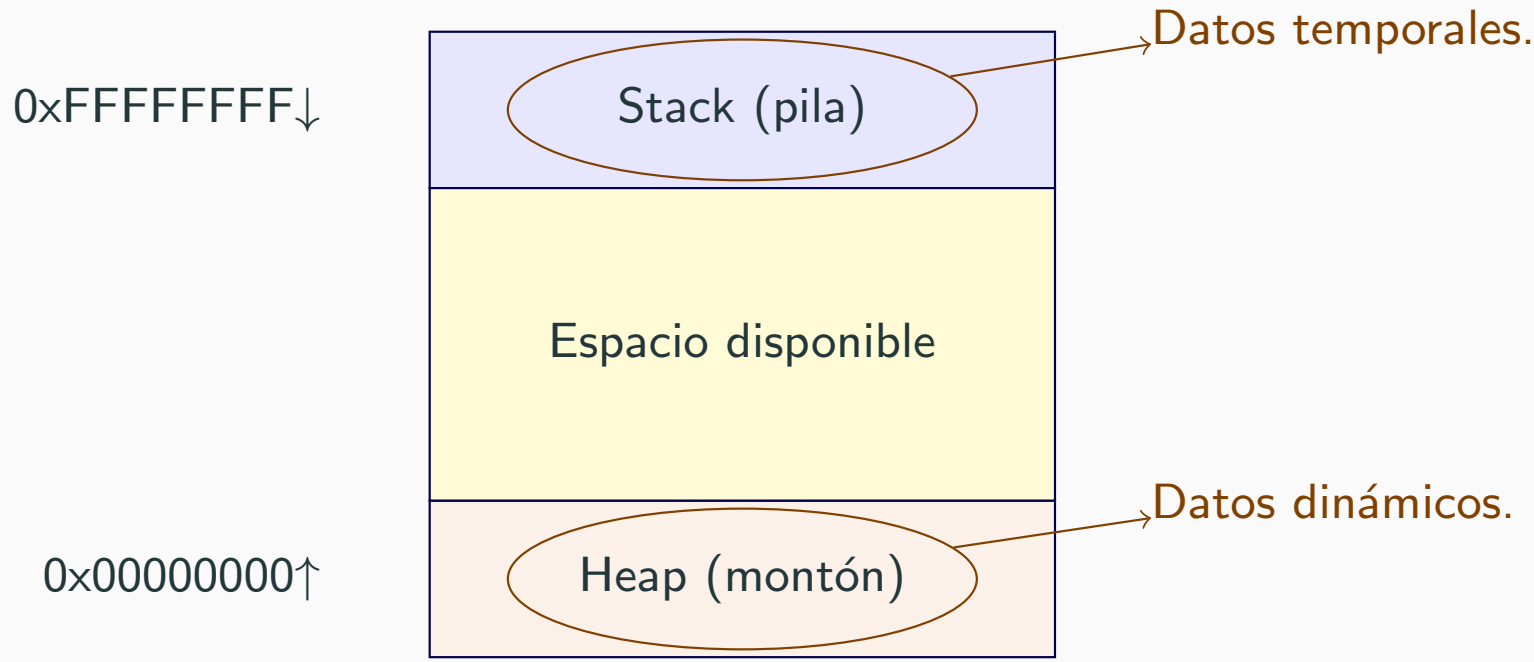
Estructura de la memoria

Ahora podemos estudiar una visión simplificada de la memoria principal desde la pespectiva de la aplicación, primero nos va a interesar definir dos secciones importantes que tienen que ver con la temporalidad de los datos.

- **La pila (o stack):**
  Donde vamos a encontrar los datos **temporales** y vinculados a la cadena de llamadas de funciones (call stack).
- **El montículo, montón o parva (heap):**
  Donde vamos a encontrar los datos **dinámicos**.
- En breve vamos a ver dónde se encuentran los datos **estáticos**.

Estructura de la memoria

Aquí hay un diagrama simplificado de la memoria.



Ambas regiones comparten un mismo espacio de memoria, debido a esto y por conveniencia una se ubica en la parte alta y se expande hacia abajo cuando hace falta (la pila) y la otra se ubica en la parte baja y se expande hacia arriba (heap).

Estructura de la memoria

Lo importante por ahora es recordar que hay dos regiones distinguidas del espacio de memoria que se utilizan para dos tipos de datos distintos.

- **La pila** para los datos temporales.
- **El heap** para los datos dinámicos.

Veamos ahora cómo nos conviene manejar los datos dinámicos.

Manejo de memoria dinámica

Datos dinámicos

Intentemos escribir el programa:

```
uint16_t *secuencia(uint16_t i);
```

Datos dinámicos

Supongamos que debemos escribir un programa `secuencia` que dado un entero n devuelve una lista con los números de 0 a $n-1$, su declaración sería:

```
uint16_t *secuencia(uint16_t n);
```

```
uint16_t *secuencia(uint16_t n){
        uint16_t *arr = foo(?);
        for(uint16_t i = 0; i < n; i++)
                arr[i] = i;
        return arr;
}
```

Observen que la forma de devolver un arreglo es con un puntero al tipo del arreglo.

Datos dinámicos

¿Qué debería hacer `foo(?)` en nuestro programa?

```
uint16_t *secuencia(uint16_t n){
        uint16_t *arr = foo(?);
        for(uint16_t i = 0; i < n; i++)
                arr[i] = i;
        return arr;
}
```

¿No podemos hacer `uint16_t arr[n]`?

Vamos a ver más adelante que este tipo de declaración trae problemas cuando queremos acceder a `arr` fuera del contexto de `secuencia`.

Intentemos explicitar algunas preguntas.

Datos dinámicos

Construyendo la función `foo(?)`

- ¿Qué parámetros debería tomar y qué tipo de dato debería devolver?
- ¿Dónde debería almacenarse `arr`?
- ¿A partir de qué momento y cuándo debería dejar de estar disponible `arr`?

Datos dinámicos

Construyendo la función `foo(?)`

- Vamos a suponer que se le indica el tamaño en bytes de la instancia a crear y devolverá la dirección de memoria dónde comienza la representación de la misma.
- Como ya habíamos dicho esta memoria se va a ubicar en el **heap** que es donde se almacenan los datos dinámicos.
- Deberíamos tener un mecanismo que nos permita indicar que la instancia ya no es necesaria y que esa porción de memoria en el **heap** vuelve a estar disponible para llamadas futuras.

Datos dinámicos

Construyendo la función `foo(?)`

- La declaración de la función será: `void *malloc(size_t size)`.
- Vamos a utilizar una función más que indica cuándo la región de memoria queda disponible: `void free(void *ptr)`.

Nótese que el tipo `void*` denota un puntero genérico, que no define a qué tipo apunta. Se puede utilizar en este caso porque todos los punteros tienen el mismo tamaño para una arquitectura dada. `size_t` es un tipo numérico utilizado para denotar tamaños.

Datos dinámicos

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

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

Por ahora vamos a suponer que estas funciones están disponibles para la aplicación.

Como mecanismo son un buen ejemplo de **arbritraje sobre un recurso compartido**, ya que desde varios puntos de un mismo programa o incluso varios procesos pueden estar pidiendo sus porciones de memoria dinámica a la misma función y en la misma región (**heap**).

Volvamos al programa con nuestra nueva función:

```
uint16_t *secuencia(uint16_t n){
        uint16_t *arr = malloc(n * sizeof(uint16_t));
        for(uint16_t i = 0; i < n; i++)
                arr[i] = i;
        return arr;
}
```

Recordemos que `sizeof` es un operador que nos indica el tamaño expresado en bytes para un tipo dado (se resuelve al momento de compilar el programa).

Datos dinámicos

Veamos un uso típico del par `malloc/free`:

```
uint16_t n = 5;
uint16_t *perm = secuencia(n);
for(uint8_t i = 0; i < n; i++)
        printf("%d ", perm[i]);
printf("\n");
free(perm);
```

Es importante liberar con `free` toda la memoria que se pide con llamadas a `malloc` porque en caso contrario podemos perjudicar el rendimiento del procesador o incuso llegar a la situación crítica de agotar la memoria dinámica disponible.

Espacio de memoria del programa

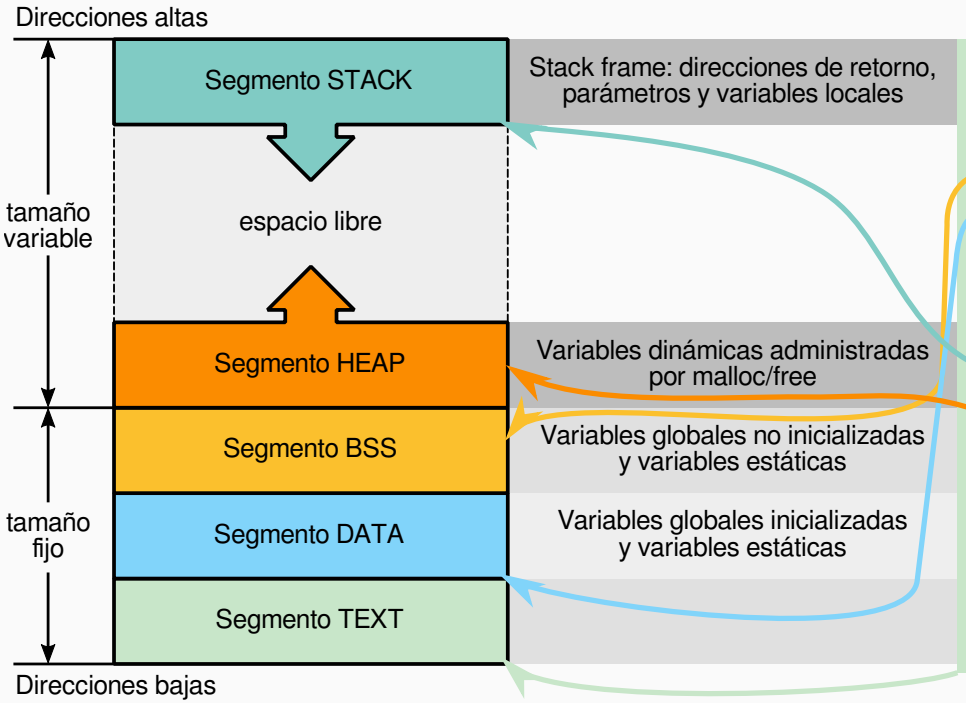
Espacio de memoria del programa

Ya estamos en condiciones de presentar una visión más completa del espacio de memoria del programa.

- Ya habíamos presentado el **heap** y la **pila**.
- Ahora vamos a introducir la sección de **text** o text, donde se almacena el código máquina del programa.
- La sección de **data** o datos estáticos inicializados.
- La sección de **bss** o datos estáticos no inicializados.

Espacio de memoria del programa

Aquí hay un diagrama simplificado de la memoria.




```
#include <stdio.h>
#include <stdlib.h>

int x;
int y = 35;

int main(int argc, char *argv[]) {

    int *data;
    int i;

    data = (int*)malloc(sizeof(int)*y);

    ...

    return 0;
}
```

Ubicación de los datos

Veamos donde se van a almacenar los datos de nuestra función:

```
uint16_t *seucencia(uint16_t n){
        uint16_t *arr =
        malloc(n * sizeof(uint16_t));

        arr∈stack, *arr∈heap

        for(uint16_t i = 0; i < n; i++)
                arr[i]        = i;

        i∈stack

        return arr;
}

i∉stack, arr∉stack
```

Aquello que viva en el stack se va a ir liberando a medida que vayamos retornando de las llamadas a función.

Ubicación de los datos

Analicemos un uso típico del par `malloc/free`:

```
uint16_t n = 5;
uint16_t *sec = secuencia(n);
```

`sec∈data,*sec∈heap`

```
for(uint8_t i = 0; i < n; i++)
        printf("%d ", sec[i]);
```

`i∈data`

```
printf("\n");
free(sec);
```

`*sec ∉ heap, sec ∈ data`

Ubicación de los datos

Noten que el puntero `sec` está en la sección de datos, pero el valor al que apunta está en el `heap`. ¿Esto siempre es así?

Respuesta:

No, podría estar apuntando a un dato estático sin problemas.

Recapitulando antes de cerrar. ¿Por qué no podíamos hacer esto?

```
uint16_t *secuencia(uint16_t n){
        uint16_t arr[n];
        for(uint16_t i = 0; i < n; i++)
                arr[i]        = i;
        return arr;
}
```

Porque `arr[n]` se crearía en el **stack** y perderíamos referencia al arreglo al salir de la función.

Cierre

Repaso

En la primera parte habíamos visto:

- Una introducción a la **perspectiva de datos**.
- Una interpretación de la memoria como un **espacio contiguo y ordenado de bits (bytes)**.
- Un repaso de los tipos de datos básicos y **cómo se representan en memoria**.
- Una **introducción a los punteros** y las operaciones de referencia y desreferencia.

Repaso

En esta última parte vimos:

- Uso del **stack** y del **heap**.
- Manejo de memoria dinámica (**malloc**,**free**).
- Estructura completa del espacio de memoria de una aplicación.

Consultas y trabajo en ejercicios