Procesos

Conceptos fundamentales

1. ¿QUÉ ES UN PROCESO?

→ Es un programa en ejecución

2. ¿Cuál es la diferencia entre un programa y un proceso?

Programas vs Procesos

Los programas son pasivos: son archivos ejecutables almacenados en memoria

Los procesos son activos: los programas se convierten en procesos cuando el archivo ejecutable es cargado en memoria

MÚLTIPLES PROCESOS PUEDEN EJECUTAR DIFERENTES INSTANCIAS DE UN PROGRAMA

3. ¿Cuáles son las partes de un proceso?

Partes de un proceso

Sección de texto (text section): Es el código como tal

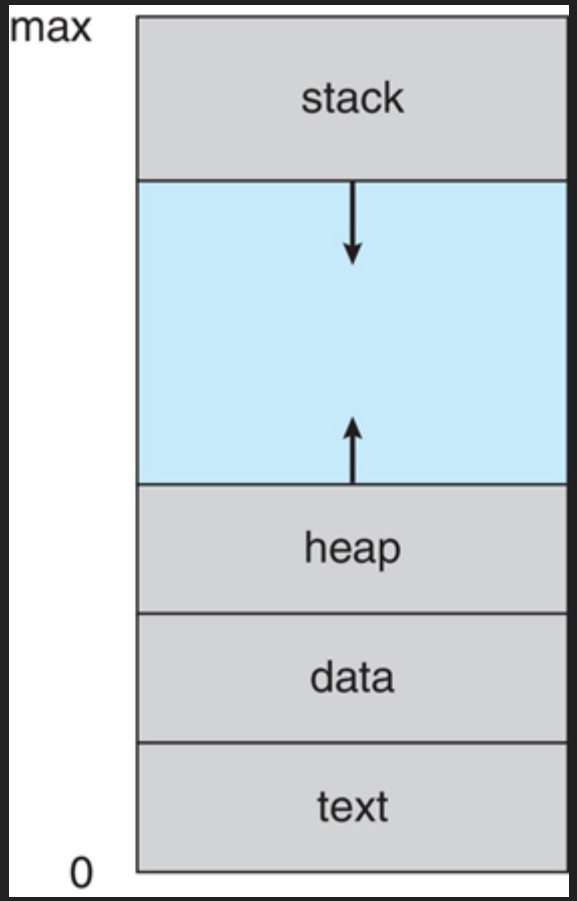
Su contexto: incluye el Program Counter (PC) y los registros del procesador

El Stack: contiene los parámetros de las funciones, las direcciones de retorno y las variables locales.

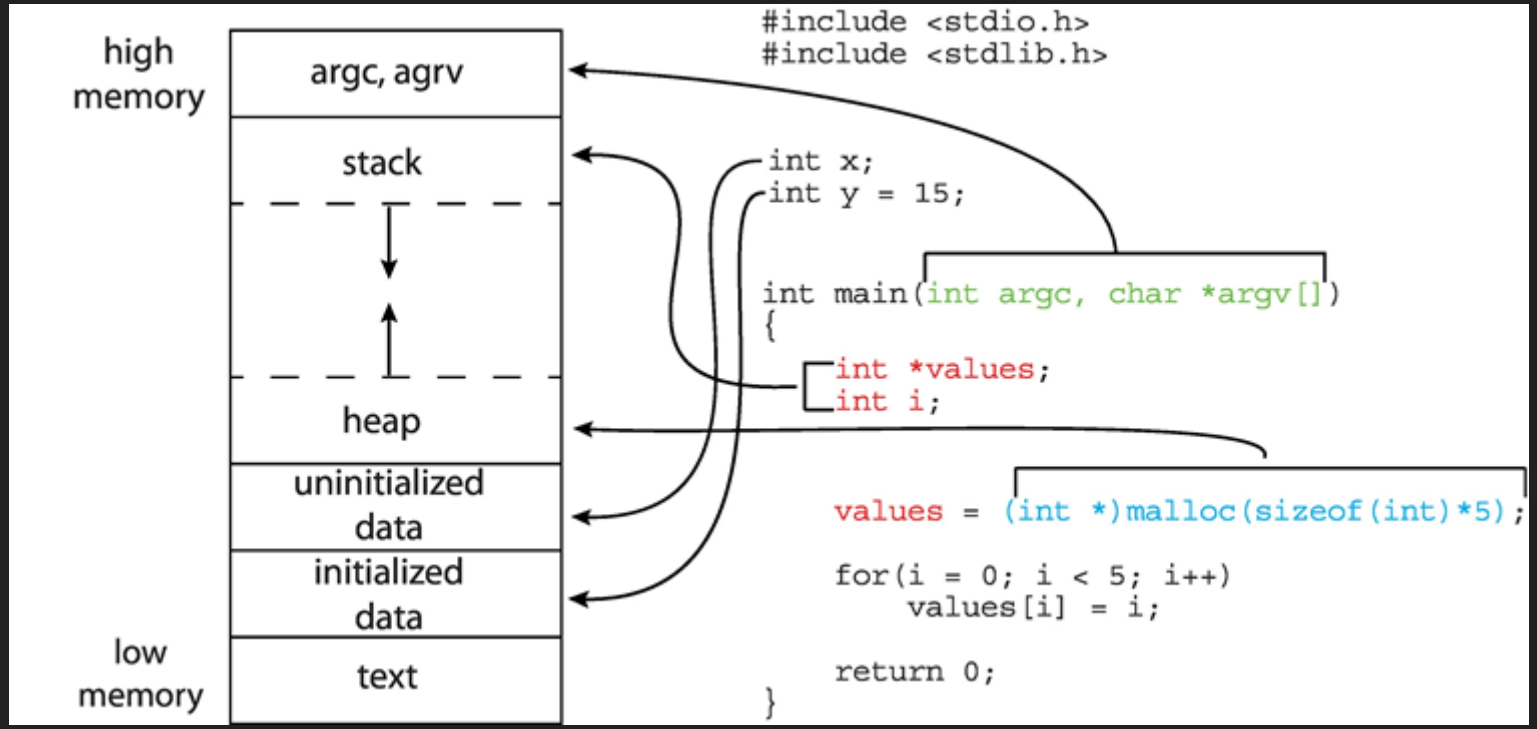
Sección de datos (data section): contiene las variables globales

Heap: contiene la memoria dinámica que se asigna en tiempo de ejecución

4. ¿Cómo se ve un proceso en memoria?


max
stack
heap
data
text
0

5. ¿Cómo es el diseño de memoria de un programa en C?


high memory
argc, agrv
stack
heap
uninitialized data
initialized data
low memory
text
#include <stdio.h>
#include <stdlib.h>
int x;
int y = 15;
int main(int argc, char *argv[])
{
int *values;
int i;
values = (int *)malloc(sizeof(int)*5);
for(i = 0; i < 5; i++)
values[i] = i;
return 0;
}

Posible estado de un proceso

Estados de un proceso

Nuevo (**new**): el proceso se está creando.

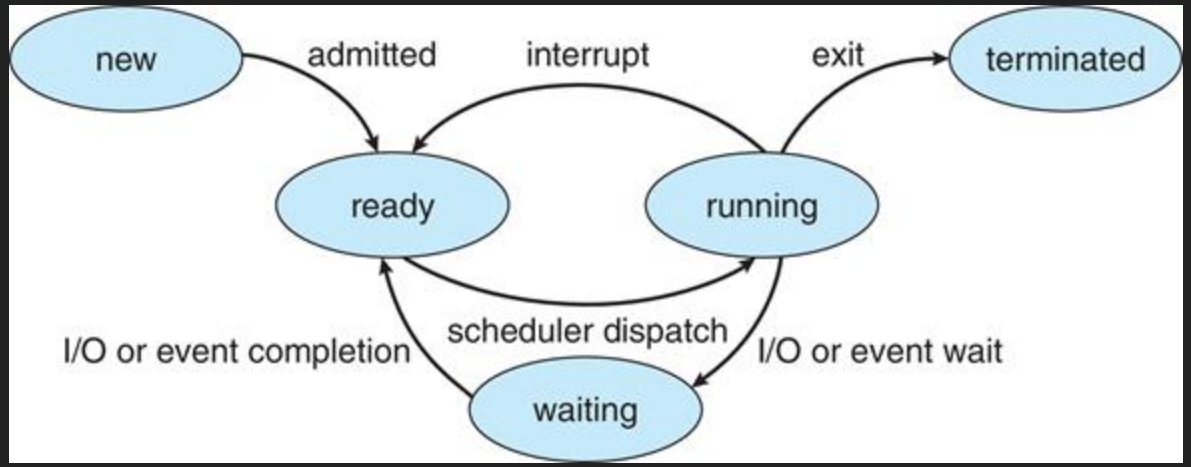
Corriendo (**running**): la CPU está ejecutando las instrucciones de su programa.

Esperando (**waiting**): el proceso está esperando a que ocurra algún evento.

Listo (**ready**): el proceso está esperando a que le asignen una CPU

Terminado (**terminated**): el proceso ha terminado su ejecución

Diagrama de estados de un proceso

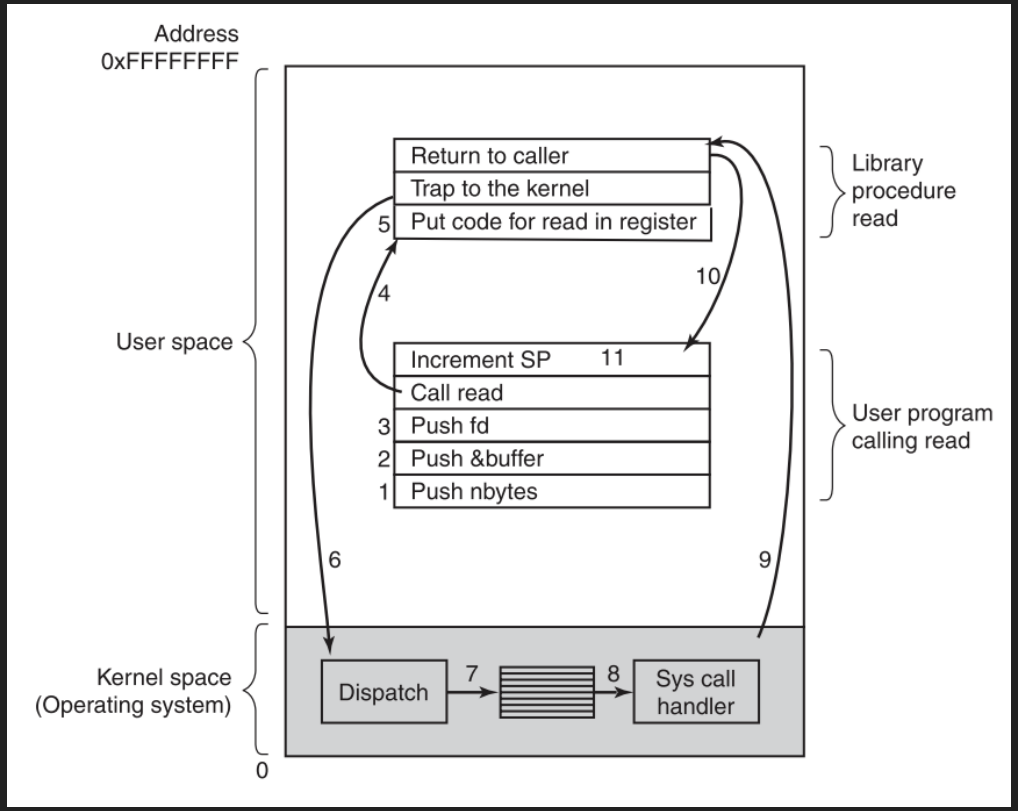


6. ¿Cómo administra los procesos un sistema operativo

Por medio de una estructura de datos
→ PCB: process control block

7. ¿Cómo hacen los programas para comunicarse con el sistema operativo?

Llamados al sistema (system calls: count = read(fd, buffer, nbytes)

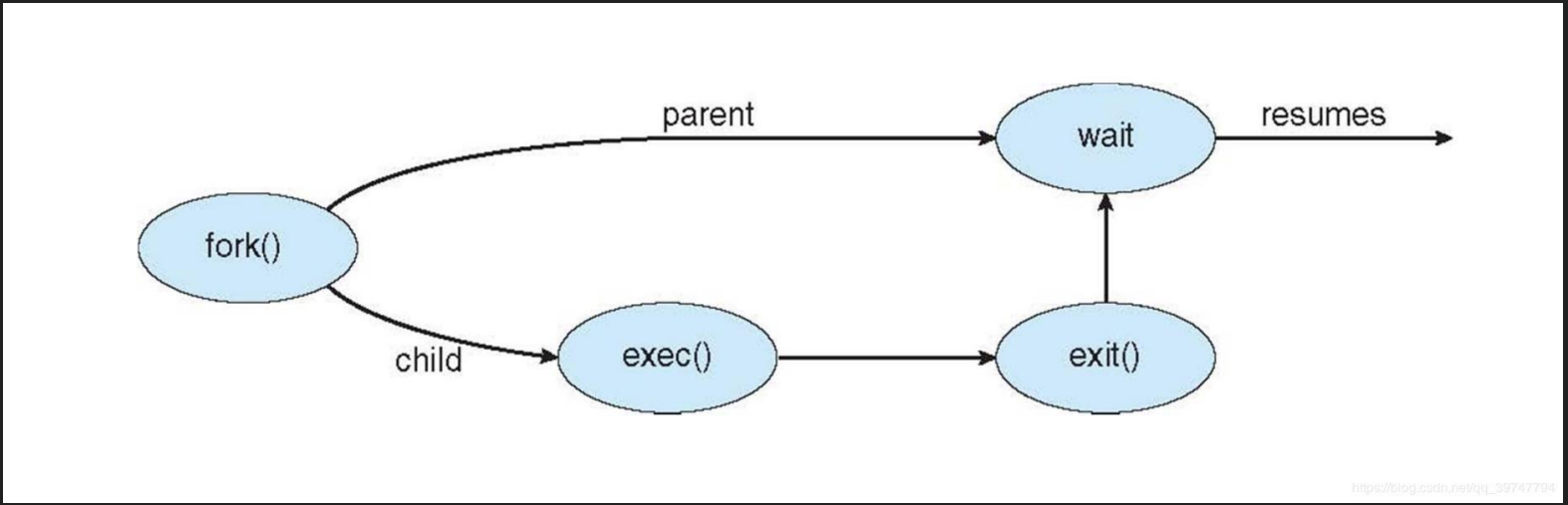


Llamados al Sistema fork() y exec()

fork(): Este llamado al sistema se utiliza para **crear un proceso separado y duplicado.**

exec(): Este llamado al sistema permite ejecutar un proceso específico (incluyendo todos sus hilos). Cuando se lanza después de fork() reemplaza el espacio de memoria con el nuevo proceso.

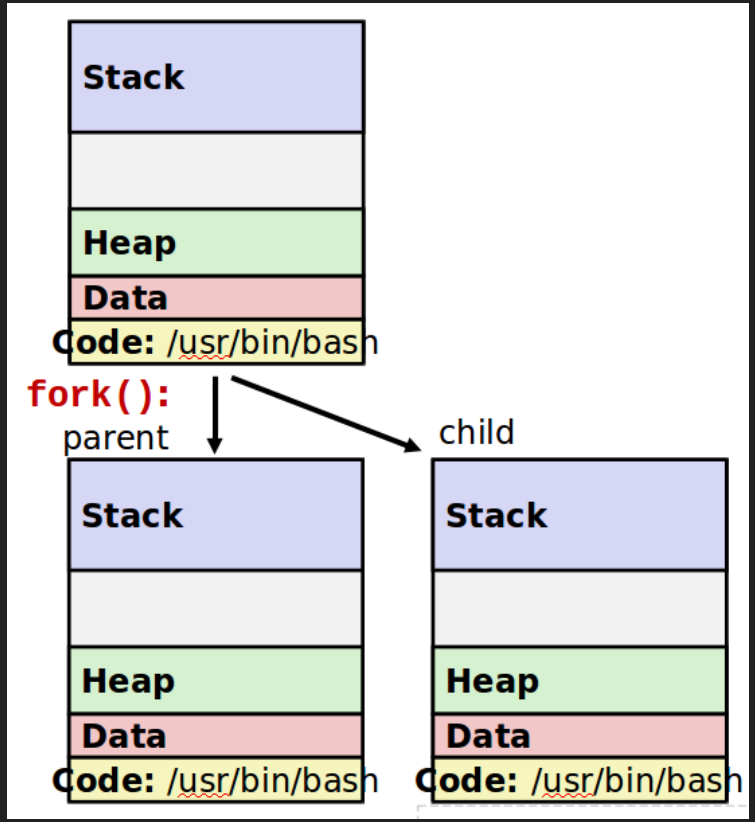
Miremos cómo se comportan **fork()** y **exec()**

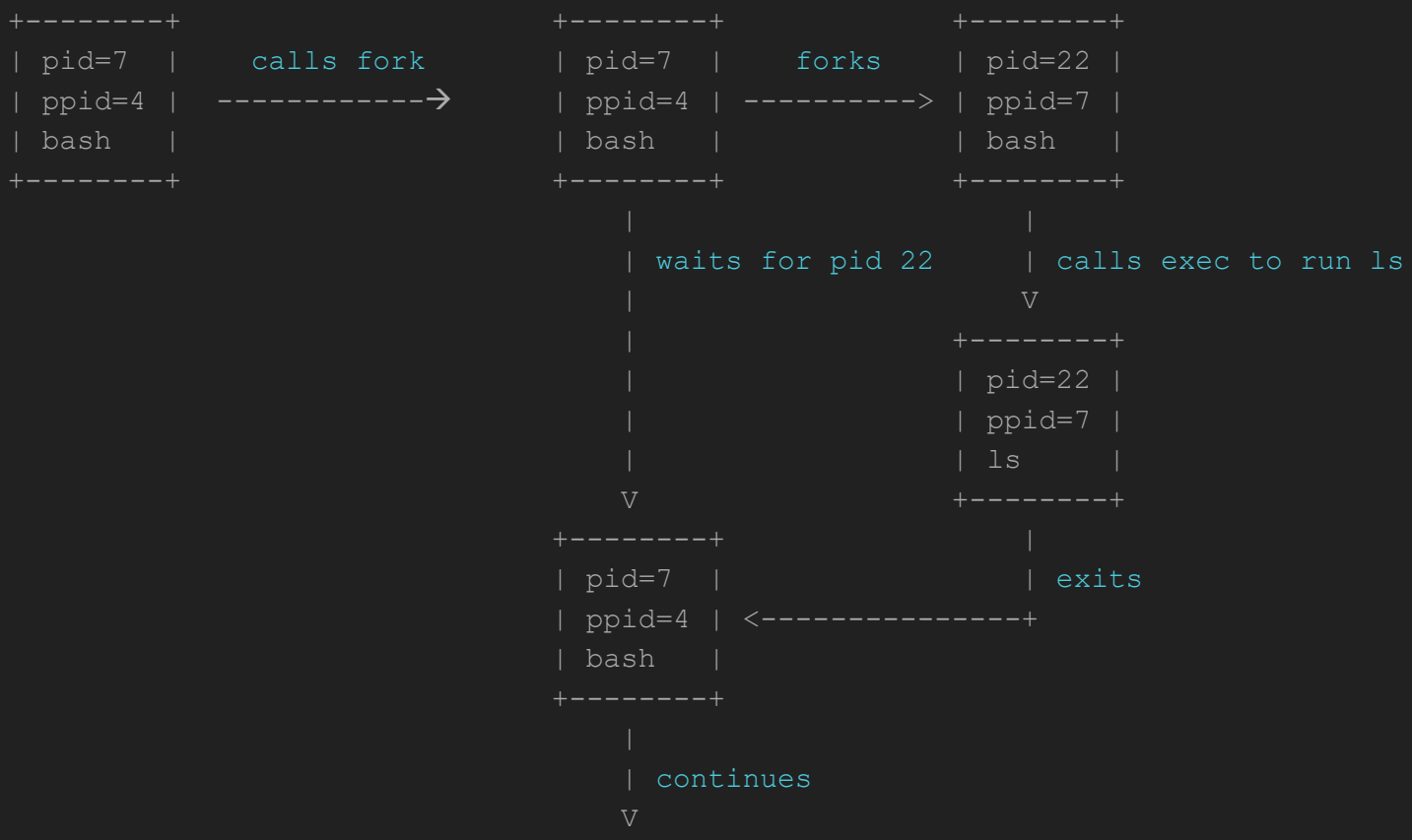


Fork()

- Crea un nuevo proceso que tiene una relación padre-hijo con el proceso principal
- El proceso secundario creado por **fork ()** copia completamente el proceso principal, por lo que el proceso secundario se ejecuta exactamente igual que el proceso principal y se procesa en paralelo con el proceso principal para competir por los recursos de la **CPU**.
- La diferencia es que el valor de retorno de **fork ()** en el proceso principal y el proceso secundario es diferente. El valor de retorno de **fork ()** en el proceso secundario es 0, y el valor de retorno de **fork ()** en el proceso principal es la identificación del proceso del proceso secundario generado.

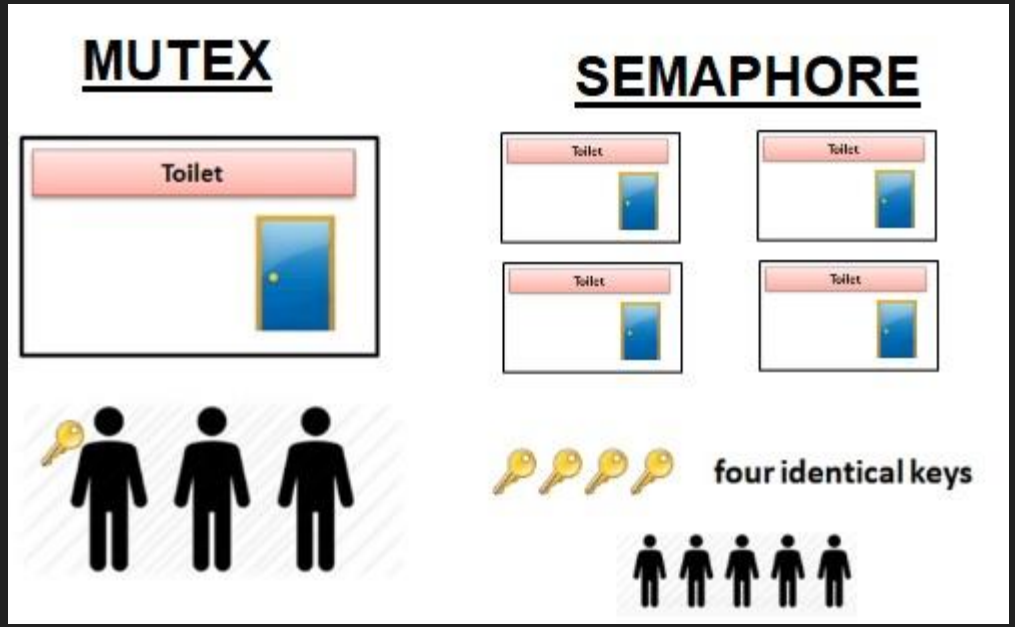
Qué sucede con memoria?




pid=7
ppid=4
bash
calls fork
pid=7
ppid=4
bash
forks
pid=22
ppid=7
bash
waits for pid 22
calls exec to run ls
pid=22
ppid=7
ls
exits
pid=7
ppid=4
bash
continues

Mutex y Semáforos

Conceptos fundamentales


MUTEX
Toilet
SEMAPHORE
Toilet
Toilet
Toilet
Toilet
four identical keys


Thread
Mutex
A mutex enables the item to be accessed by only one thread at a time
Image Credits: CodeGym