El Proceso

> **Definición**
>
> - Un proceso es **la ejecución de un programa** de aplicación con derechos restringidos; el proceso **es la abstracción** que provee el Kernel del sistema operativo para la ejecución protegida - [DAH]
> - Un proceso es una entidad **abstracta**, definida por el Kernel, en la cual los recursos del sistema son asignados - [KERR]

Un proceso incluye:

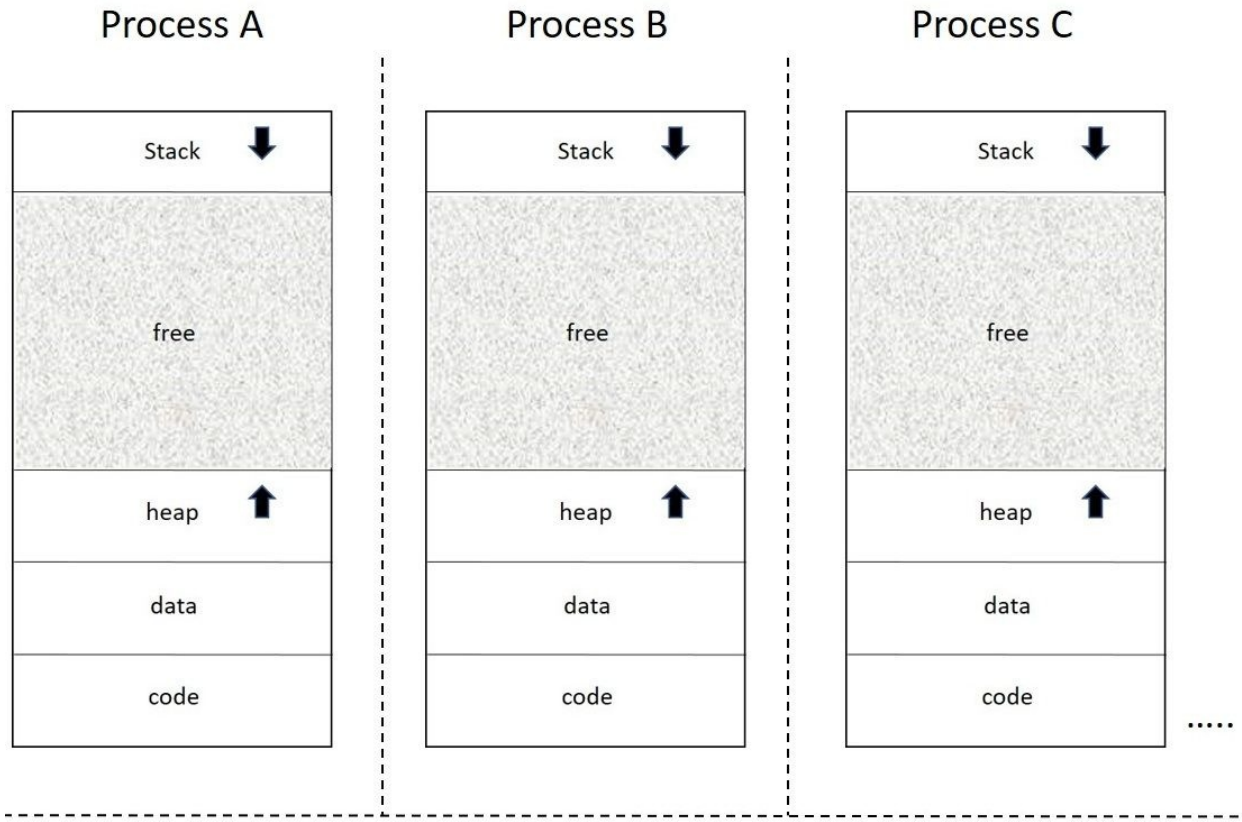
- Los archivos abiertos.
- Las señales (signals) pendientes.
- Datos internos del kernel.
- El estado completo del procesador.
- Un espacio de direcciones de memoria.
- Uno o más hilos de ejecución. Cada thread contiene:
  - Un único contador de programa.
  - Un Stack.
  - Un Conjunto de Registros.

Virtualización de Memoria

La virtualización de memoria le hace creer al proceso que este tiene toda la memoria disponible para ser reservada y usada como si este estuviera siendo ejecutado sólo en la computadora (ilusión).
Todos los procesos en Linux, están divididos en 4 segmentos que conforman su **Address Space**:

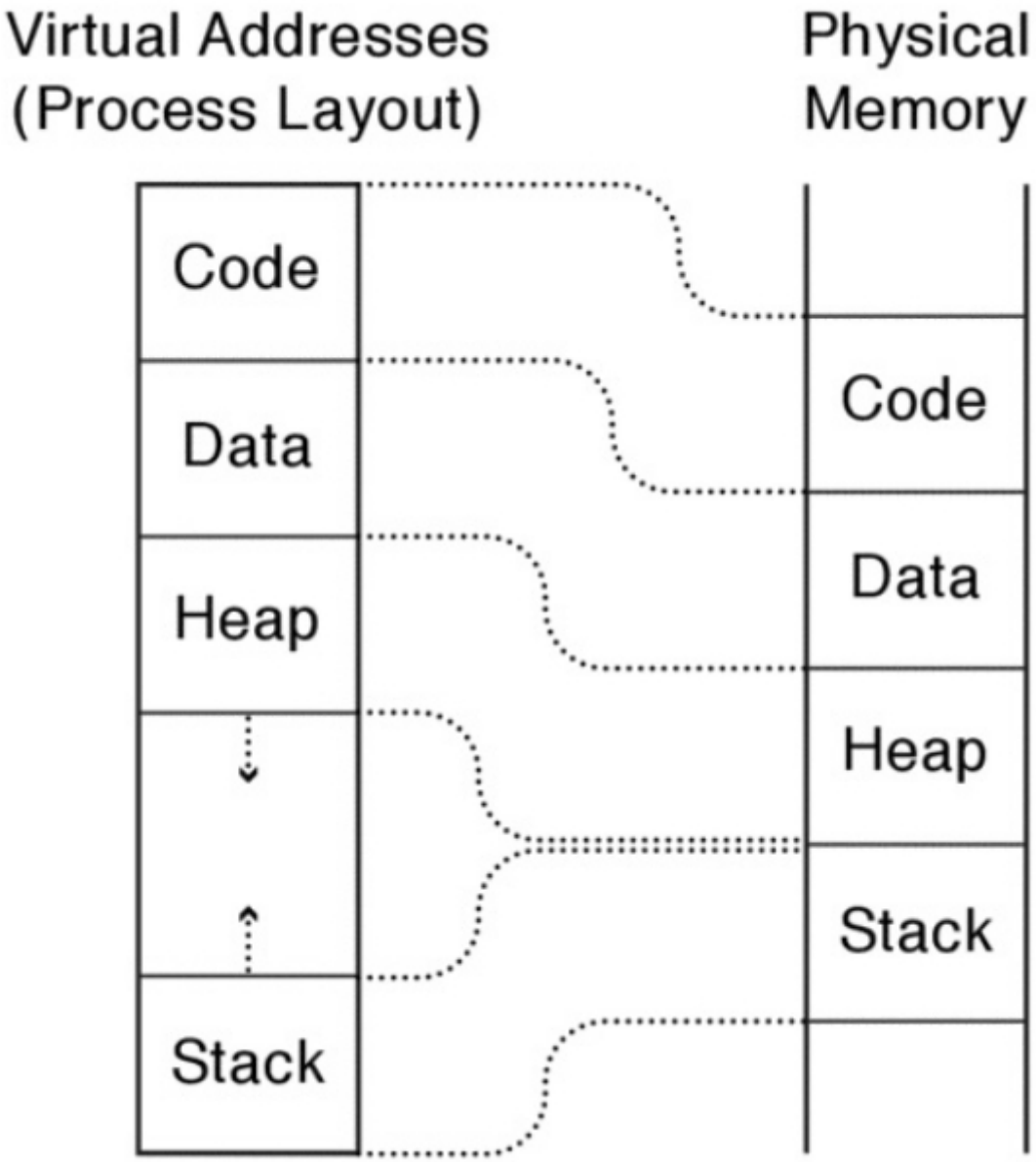
- **Text**: Instrucciones del Programa. (Program code)
- **Data**: Variables Globales (*extern* o *static* en C).
- **Heap**: Memoria Dinámica Alocable.

- **Stack**: Variable Locales y trace de llamadas.



Protección de Memoria

El **proceso** tiene que estar en memoria para poder ejecutarse y el **sistema operativo** tiene que estar ahí para iniciar la ejecución del programa, manejar las interrupciones y/o atender las systems call.
Además, otros procesos podrían estar simultáneamente en memoria y para poder compartir la memoria de forma segura, el sistema operativo debe proveer un mecanismo de **protección de memoria** que asegure que cada proceso pueda leer y escribir solo su propia memoria (no la memoria del sistema operativo ni la de otros procesos), ya que de no ser así el proceso en cuestión podría modificar al Kernel del sistema operativo.
Uno de estos mecanismos es denominado **Memoria Virtual**, una **asbtracción** por al cual la **memoria física** puede ser compartida por diversos procesos.



Virtualización del procesador

Consiste en dar la **ilusión** de la existencia de un único procesador para cualquier programa que requiera de su uso. Si bien la computadora está ejecutando más de un proceso, **todos** creen que la *CPU* está disponible en forma **exclusiva** para ese único proceso.
De esta forma, se provee:

- **Simplicidad en la programación**:
  - Cada proceso cree que tiene toda la CPU.
  - Cada proceso cree que todos los dispositivos le pertenecen.
  - Distintos dispositivos parecen tener el mismo nivel de interfaces.
  - Las interfaces con los dispositivos son más potentes que el bare metal.
- **Aislamiento frente a Fallas**:
  - Los procesos no pueden directamente afectar a otros procesos.
  - Los errores no colapsan toda la máquina

El sistema operativo lleva la contabilidad de todos los procesos que se están ejecutando en la computadora mediante la utilización de una estructura llamada **Process Control Block o PCB**.
La PCB almacena toda la información que un sistema operativo debe conocer sobre un proceso en particular:

- Donde se encuentra almacenado en memoria.
- Donde la imagen ejecutable esta en el disco.
- Que usuario solicito su ejecución.
- Que privilegios tiene ese proceso.

La idea general detrás de la abstracción es la de cómo **virtualizar una CPU** o **procesamiento**, es decir cómo hacer para que un único procesador actúe como tal para varios programas que requieren ser ejecutados utilizando el mismo hardware, en este caso un microprocesador.

Un **Proceso** necesita permisos del **Kernel** del SO para:

- Acceder a memoria perteneciente a otro proceso.
- Antes de escribir o leer en el disco.
- Antes de cambiar algún seteo del hardware del equipo.
- Antes de enviar información a otro proceso.

Pero además el S.O. crea la ilusión de la existencia de varios cientos o miles de procesadores, cuando en realidad tiene uno solo, mediante la virtualización de la CPU con el concepto de **Proceso** que es la ilusión creada para la virtualización de la CPU. El Kernel del S.O. provee esa abstracción.

El Contexto de un Proceso

El contexto de un proceso es la información necesaria para describir al proceso, cada proceso posee un contexto.
El contexto de un proceso consiste en la unión de:

1. **User-Level Context:** consiste en las secciones que forman parte de **Virtual Address Space** del proceso (text, data, stack, heap).
2. **Register Context**: consiste de:
    - Program Counter Register
    - Processor Status Register
    - Stack Pointer Register
    - General Purpose Registers
3. **System-level Context**: consiste en:

- La entrada en la **Process Table Entry.**
- La **u area**.
- La **Process Region Entry**, **Region Table** y **Page Table** que definen el mapeo de la **memoria virtual** vs **memoria física** del proceso.
- El Stack del Kernel, contiene los stack frames de las llamadas al kernel hechas por el proceso.

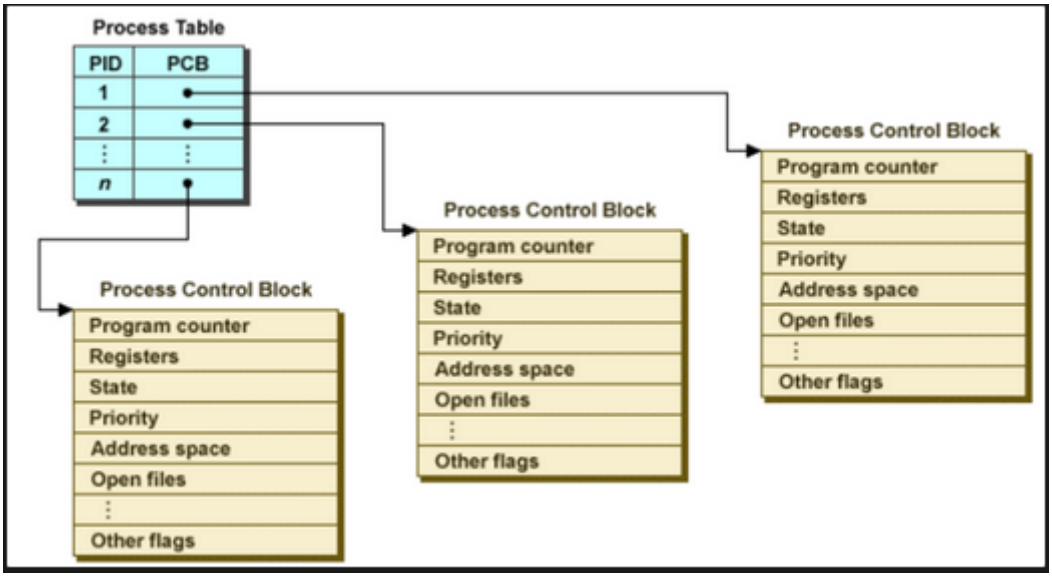
```
// the registers xv6 will save and restore
// to stop and subsequently restart a process

struct context
{
    int eip;
    int esp;
    int ebx;
    int ecx;
    int edx;
    int esi;
    int edi;
    int ebp;
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// the information xv6 tracks about each process
// including its register context and state
struct proc {
    char *mem;                  // Start of process memory
    uint sz;                    // Size of process memory
    char *kstack;               // Bottom of kernel stack
                                // for this process
    enum proc_state state;      // Process state
    int pid;                    // Process ID
    struct proc *parent;        // Parent process
    void *chan;                 // If non-zero, sleeping on chan
    int killed;                 // If non-zero, have been killed
    struct file *ofile[NOFILE]; // Open files
    struct inode *cwd;          // Current directory
    struct context context;     // Switch here to run process
    struct trapframe *tf;       // Trap frame for the
                                // current interrupt
};
```

Process Structure

Es una entrada en una tabla conocida como **process table**. Contiene información que siempre tiene que estar disponible para el kernel, incluso cuando el proceso no se está ejecutando.

Contiene:

- Identificación: cada proceso tiene un identificador único o *process ID* (PID) y además pertenece a un determinado grupo de procesos.
- Ubicación del mapa de direcciones del Kernel del u area del proceso.
- Estado actual del proceso.
- Un puntero hacia el siguiente proceso en el planificador y al anterior.
- Prioridad.
- Información para el manejo de señales.
- Información para la administración de memoria.



User Area

Es parte del espacio del proceso, se mapea y es visible por el proceso solo cuando éste está siendo ejecutado. Contiene información que solo es necesario acceder cuando el proceso se está ejecutando.

Contiene:

- Un puntero a la proc structure del proceso.
- El UID y GID real.
- Argumentos para, y valores de retorno o errores hacia, la system call actual.
- Manejadores de Señales.
- Información sobre las areas de memoria text,data, stack, heap y otra información.

- La tabla de descriptores de archivos abiertos (Open File descriptor Table).
- Un puntero al directorio actual.
- Datos estadísticos del uso de la cpu, información de perfilado, uso de disco y limites de recursos.

El API resumida:

1. `fork()` : Crea un proceso y devuelve su id.
2. `exit()` : Termina el proceso actual.
3. `wait()` : Espera por un proceso hijo.
4. `kill(pid)` : Termina el proceso cuyo pid es el parámetro.
5. `getpid()` : Devuelve el pid del proceso actual.
6. `exec(filename, argv)` : Carga un archivo y lo ejecuta.
7. `sbrk(n)` : Crece la memoria del proceso en n bytes.

Creación de un proceso

La única forma de que un usuario cree un proceso en UNIX es llamando a la system call `fork()`.
El proceso que invoca a fork es llamado proceso padre, el nuevo proceso creado es llamado hijo.

¿Qué hace fork?

1. Crea y asigna una nueva entrada en la **Process Table** para el nuevo proceso.
2. Asigna un número de ID **único** al proceso hijo.
3. Crea una copia lógica del contexto del proceso padre, algunas de esas partes pueden ser compartidas como la sección **text.**
4. Realiza ciertas operaciones de I/O.
5. Devuelve el número de ID del hijo al proceso **padre**, y un 0 al proceso hijo.

¿Qué hace *fork()*, el algoritmo?

- Chequear que haya **recursos** en el kernel;
- Obtener una entrada libre de la **Process Table**, como un PID único;
- Chequear que el usuario no esté ejecutando demasiados procesos;
- Marcar al proceso hijo en estado “siendo creado”;
- Copiar los datos de la entrada en la **Process Table** del padre a la del **hijo**;
- Incrementar el contador del **current directoty inode**;
- Incrementar el contador de archivos abiertos en la **File Table**;

- Hacer una copia del **contexto** del **padre** en memoria;
- Crear un contexto a nivel sistema falso para el **hijo**:
  - El contexto falso contiene datos para que el hijo se reconozca a sí mismo
  - y para que tenga un punto de inicio cuando el planificador lo haga ejecutarse.

```
if (el proceso en ejecución es el padre) {
        cambiar el estado del hijo a "ready to run";
        return (ID del hijo);
}
else /* se esta ejecutando el hijo */
{
        inicializar algunas cosas;
        return 0;
}
```

Ejemplo de fork()

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

int main(int argc, char *argv[]) {
        printf("hello world (pid:%d)\n", (int) getpid());
        int rc = fork();
        if (rc < 0) { // fork failed; exit
                fprintf(stderr, "fork failed\n");
                exit(1);
         }
         else if (rc == 0) { // child (new process)
            printf("hello, I am child (pid:%d)\n", (int) getpid());
         }
         else {// parent goes down this path (main)
                printf("hello, I am parent of %d (pid:%d)\n", rc, (int)
getpid());
         }
     return 0;
}
```

Forkbomb:

```
:(){
     :|:&
 };:
```

La conocida cadena `:(){ :|:& };:` no es nada más que una función de bash, la cual se ejecuta recursivamente.
`:()` - Define una función con nombre `:`. Es una función que no acepta argumentos.
`:|:` - La función se llama a sí misma y redirecciona el output o salida, utilizando el operador pipe `|`, a otra llamada de la misma función `:`. La magia está en que se llama dos veces a la función y así comienza el bombardeo al sistema.
`&` - Pone el llamado de la función en background de manera que los procesos hijos no mueran y así se comen los recursos del sistema.
`;` - Finalización de la definición de la función
`:` - Llamado de la función, AKA setteo de la bomba.

Ejemplo de forkbomb en C

```c
#include <stdio.h>
#include <sys/types.h>

int main() {
    while(1)
       fork();
    return 0;
}
```

System Call exit()

Generalmente un proceso tiene dos formas de terminar:

1. La **anormal**: a través de recibir una señal cuya acción por defecto es terminar el programa.
2. La **normal**: a través de invocar a la system call `exit()`.

exit(), el algoritmo:

- Ignora todas las signals.
- Cierra todos los archivos abiertos.
- En consecuencia, se liberan todos los locks mantenidos por este proceso sobre esos archivos.

- Libera el directorio actual.
- Los segmentos de memoria compartida del procesos se separan.
- Los contadores de los semáforos son actualizados.
- Libera todas las secciones y memoria asociada al proceso.
- Registra información sobre el proceso (accounting record).
- Pone el estado del proceso en “zombie”.
- Le asigna el parent PID de los procesos hijos al PID de *init*.
- Le manda una signal o señal de muerte al proceso padre.
- Context switch.

```c
#include <stdlib.h>

void exit(int status);
```

System Call wait()

Un proceso puede sincronizar su ejecución con la finalización de un proceso hijo mediante la ejecución de `wait()`.

- En ciertos casos el proceso padre necesita esperar que el proceso hijo realice cierta tarea para continuar con su ejecución.
- Para ello existe la system call `wait()` que retrasa la ejecución del proceso padre hasta que el proceso hijo termine su ejecución.

Ejemplo de fork() y wait()

```c
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
        pid_t pid;
        pid = fork();
        if (pid == 0) { /* Hijo */
                printf(" Soy Luke   ....    mi pid es : %d\n",
getpid());
                exit(0);
        }
        /* Padre */
        int status;
        pid_t p= wait(&status);
```

```
        printf(" Mi pid es %d .... Luke soy tu padre!!!!\n", getpid());
        return 0;
}
```

System Call exec()

La system call `exec()` invoca a otro programa, sobreponiendo el espacio de memoria del proceso con el programa ejecutable.

```
#include <unistd.h>

int execve(const char *filename, const char *argv[], const char *envp[]);
```

¿Qué hace *exec*?

- Obtiene el inodo del programa;
- Verifica si el archivo es ejecutable y si el usuario tiene los permisos para ejecutarlo;
- Lee el header del archivo;
- Copia los parámetros del exec del viejo address space al system space;
- Para cada región asociada al proceso las des-asocia.
- Para cada región especificada en el módulo ejecutable:
  - Alloca espacio para las nueva región; asocia (attach) la región,
  - Carga la región en la memoria.
- Copia los parámetros del exec en la nueva región o sección stack.
- Hace cierta magia.
- Inicializa a modo usuario.
- Libera el inodo.

Ejemplo de exec(), fork() y wait()

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/wait.h>

int main(int argc, char *argv[]){
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0){
```

```
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
        char *myArgs[3];
        myArgs[0]=strdup("wc");             // programa wc
        myArgs[1]=strdup("proceso2.c");     // arg: programa a ser
contado
        myArgs[2]=NULL;                     // marca el fin del
arreglo
        execvp(myArgs[0],myArgs);
        printf("Esto no deberia imprimirse pues es totalmente
reemplazado");
    } else {
        // parent goes down this path (main)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",rc,
wc, (int) getpid());
}
```

Un programa en Unix

Un programa es un archivo que posee toda la información de como construir un proceso en memoria. Un programa contiene:

- I**nstrucciones de Lenguaje de Máquina**: Almacena el código del algoritmo del programa.
- **Dirección del Punto de Entrada del Programa**: Identifica la dirección de la instrucción con la cual la ejecución del programa debe iniciar.
- **Datos**: El programa contiene valores de los datos con los cuales se deben inicializar variables, valores de constantes y de literales utilizadas en el programa.
- **Símbolos y Tablas de Realocación**: Describe la ubicación y los nombres de las funciones y variables de todo el programa, así como otra información que es utilizada por ejemplo para debugg.
- **Bibliotecas Compartidas**: describe los nombres de las bibliotecas compartidas que son utilizadas por el programa en tiempo de ejecución así como también la ruta del linker dinámico que debe ser usado para cargar dicha biblioteca.

Un programa en Linux: ELF

ELF (Extensible Linking Format) utilizado en la actualidad.