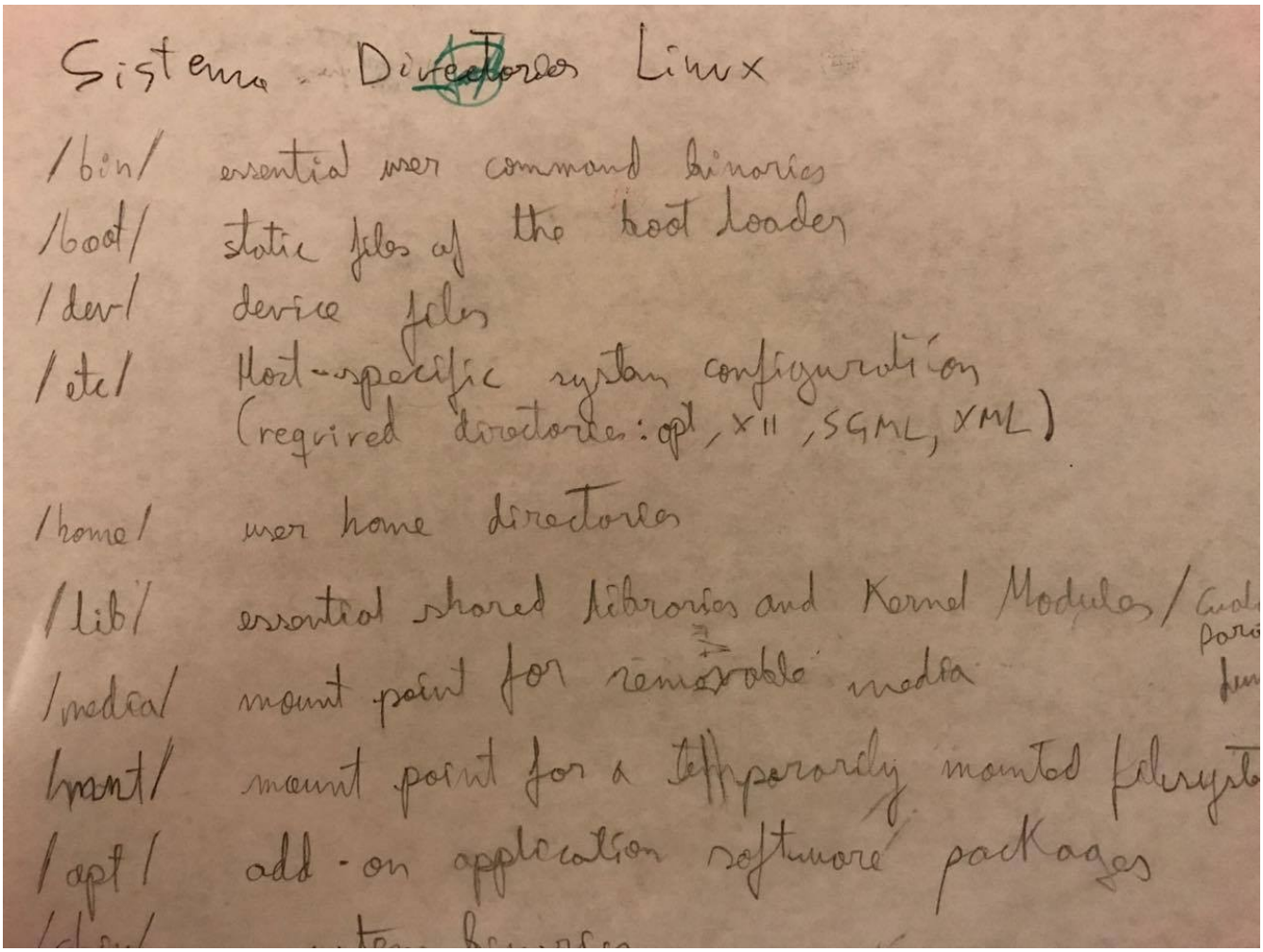
INSTITUTO POLITÉCNICO NACIONAL
ESCUELA SUPERIOR DE CÓMPUTO
SISTEMAS OPERATIVOS
GRUPO: 2CM8

ALUMNO:
NAVA VIVAS ANA PAOLA

REPORTE DE TAREAS

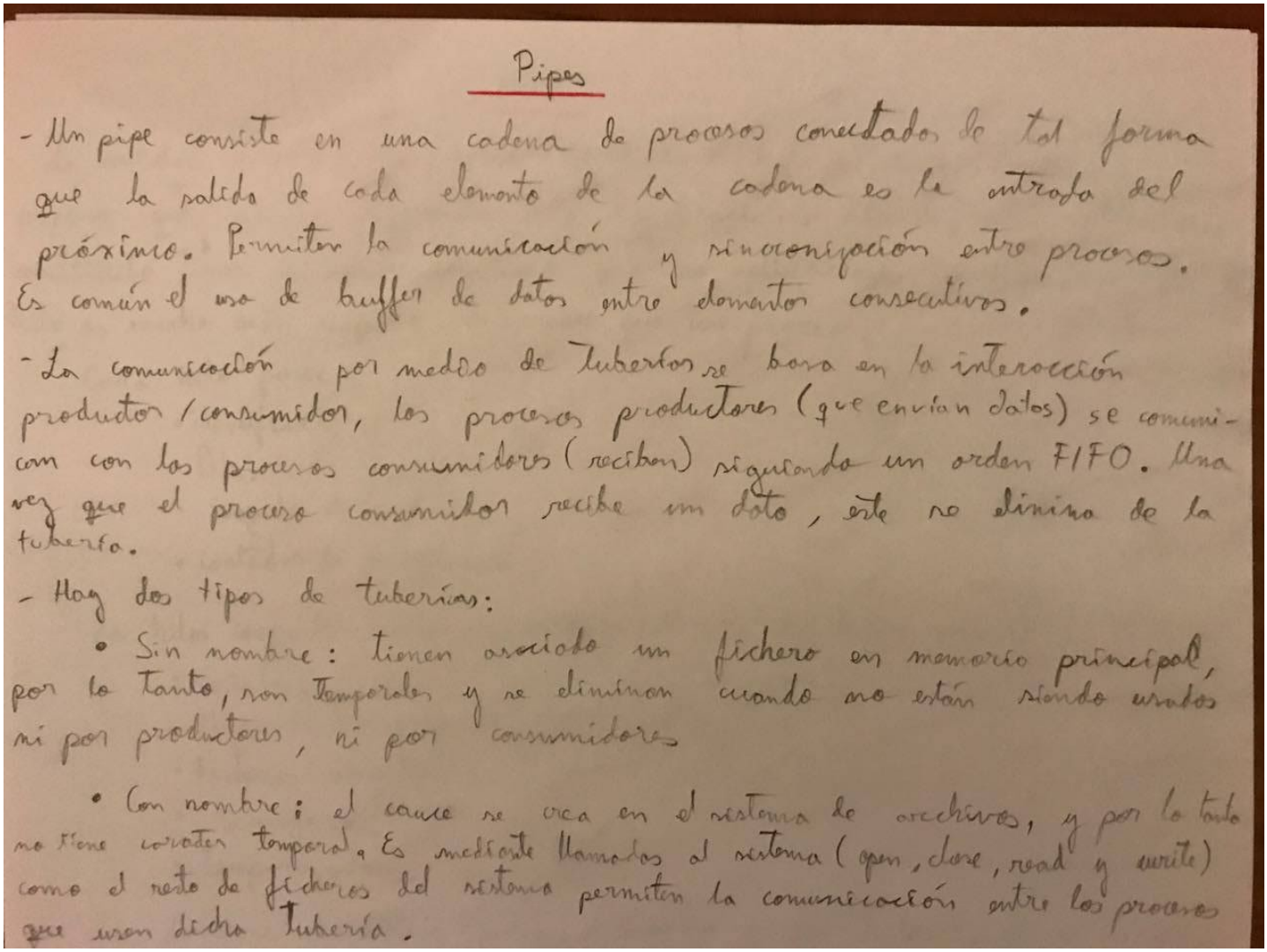
Directorios Linux:

La tarea consistía en mencionar los diferentes directorios de linux más una explicación sobre cada uno, qué archivos y comandos contienen, con la finalidad de conocer mejor este sistema operativo.



Pipes:

La tarea consist[ia en definir qué son los pipes, los tipos de pipes, con nombre o sin nombre, y su utilizaicón.



Pipes

- Un pipe consiste en una cadena de procesos conectados de tal forma que la salida de cada elemento de la cadena es la entrada del próximo. Permiten la comunicación y sincronización entre procesos. Es común el uso de buffer de datos entre elementos consecutivos.

- La comunicación por medio de Tuberías se basa en la interacción productor/consumidor, los procesos productores (que envían datos) se comunican con los procesos consumidores (reciben) siguiendo un orden FIFO. Una vez que el proceso consumidor recibe un dato, este no elimina de la tubería.

- Hay dos tipos de tuberías:

  • Sin nombre: tienen asociado un fichero en memoria principal, por lo tanto, son Temporales y se eliminan cuando no están siendo usados ni por productores, ni por consumidores

  • Con nombre: el cauce se crea en el sistema de archivos, y por lo tanto no tiene carácter temporal. Es mediante llamadas al sistema (open, close, read y write) como el resto de ficheros del sistema permiten la comunicación entre los procesos que usan dicha Tubería.

tubería.

- Hay dos tipos de tuberías:
  - Sin nombre: tienen asociado un fichero en memoria principal, por lo tanto, son temporales y se eliminan cuando no están siendo usados ni por productores, ni por consumidores
  - Con nombre: el cauce se crea en el sistema de archivos, y por lo tanto no tiene carácter temporal. Es mediante llamadas al sistema (open, close, read y write) como el resto de ficheros del sistema permiten la comunicación entre los procesos que usen dicha tubería.

Pipes en Linux, en C:

Hay que llamar a la función pipe y pedirle los dos descriptores de archivo que serán los extremos del tubo que se habrá construido.

```
int p[2];
pipe(p);
```

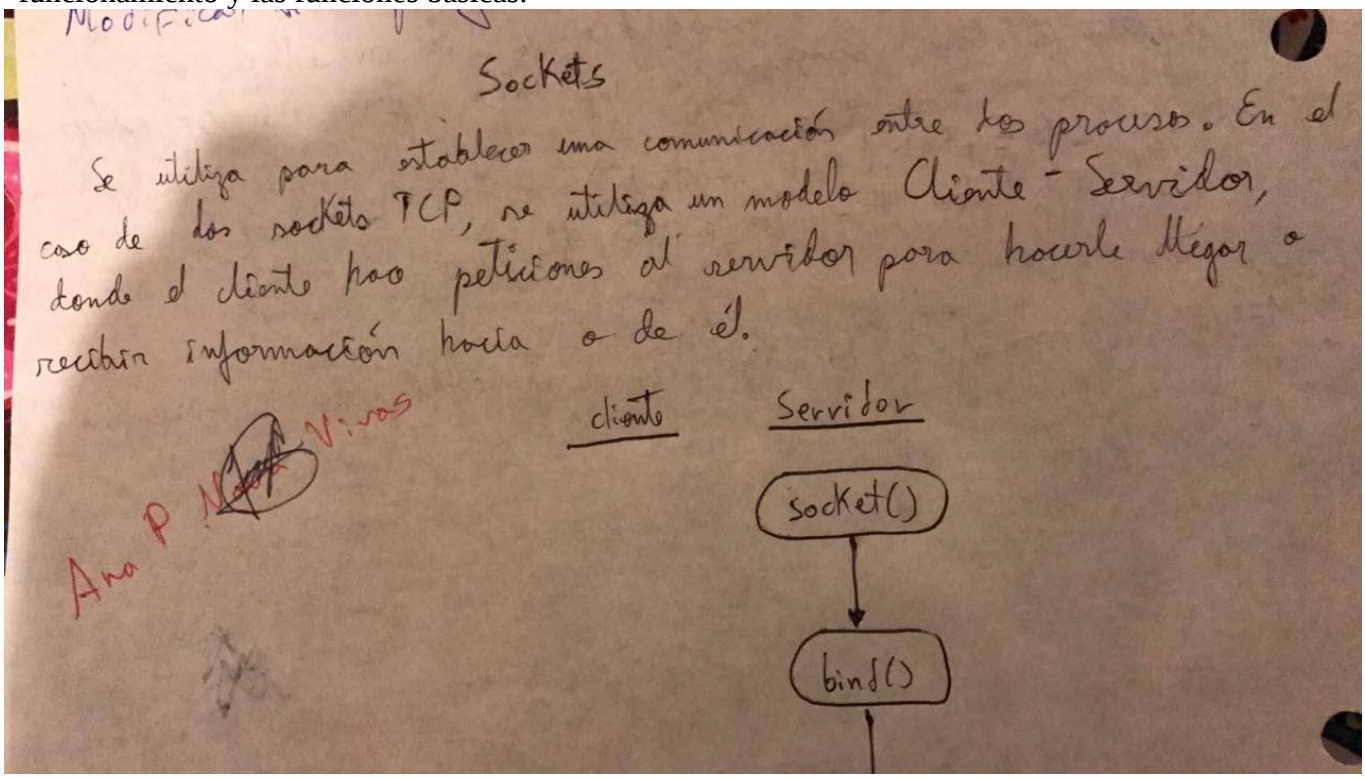
p contiene en p[0] un descriptor de archivo para leer
p contiene en p[1] un descriptor de archivo para escribir

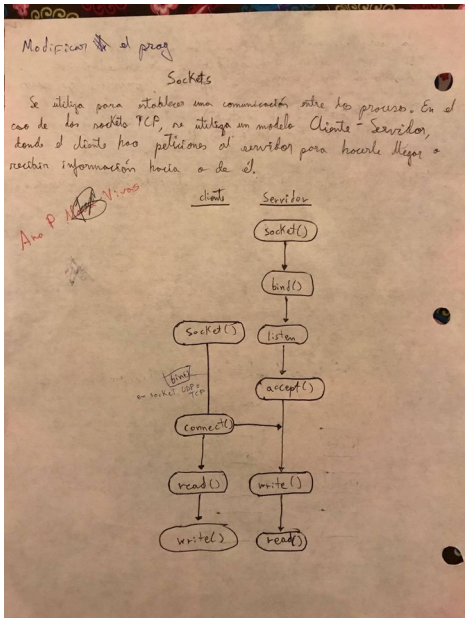
* Los pipes funcionan en una sola dirección

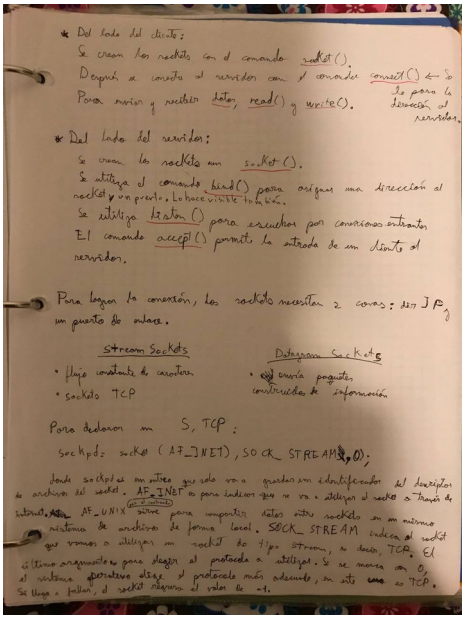
Los descriptores de archivos se pueden usar para leer y escribir bytes mediante las funciones read, write y close

Sockets:

En esta tarea tuvimos que investigar qué eran los sockets, escíficamente los de TCP, la lógica de su funcionamiento y las funciones básicas.

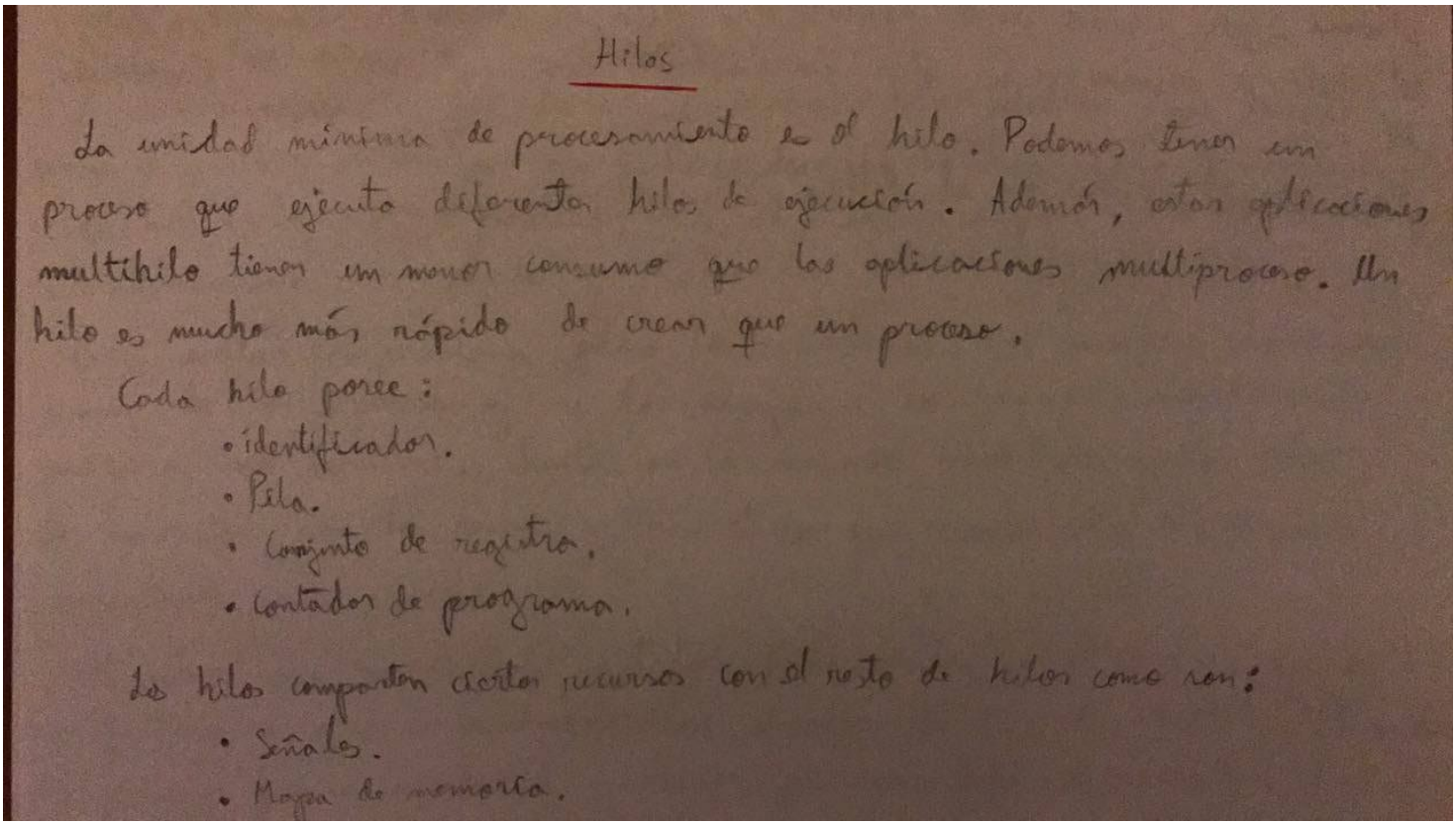






Hilos:

La tarea consistía en investigar la definición de hilos y su uso.



Hilos

La unidad mínima de procesamiento es el hilo. Podemos tener un proceso que ejecuta diferentes hilos de ejecución. Además, estas aplicaciones multihilo tienen un menor consumo que las aplicaciones multiproceso. Un hilo es mucho más rápido de crear que un proceso.

Cada hilo posee:

- Identificador.
- Pila.
- Conjunto de registro.
- Contador de programa.

Los hilos comparten ciertos recursos con el resto de hilos como son:

- Señales.
- Mapa de memoria.

- Temporizadores.

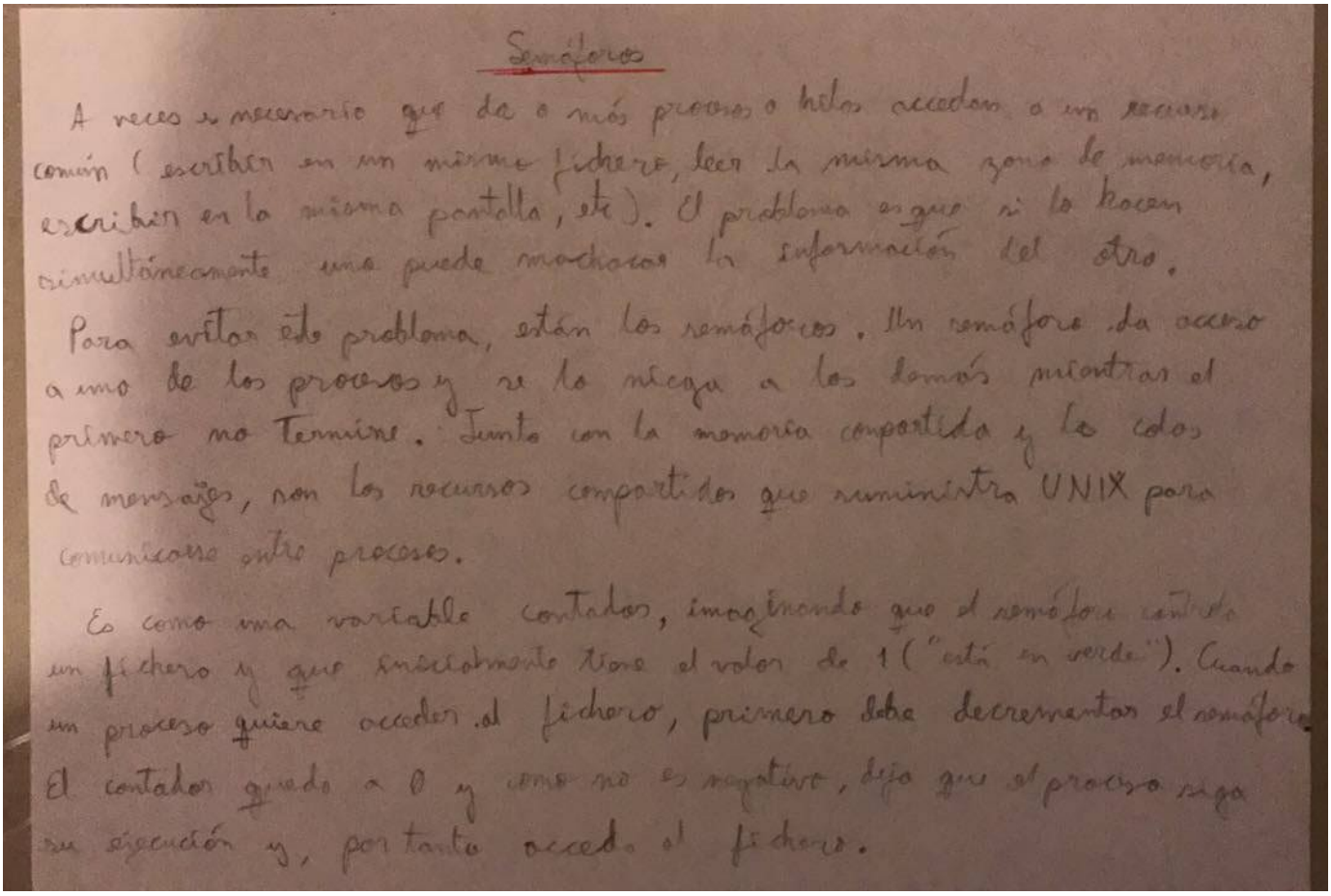
Las funciones necesarias para el manejo de los hilos en C se encuentra en la biblioteca pthread.h.

- int pthread_create(pthread_t *thread, const pthread_attr_t * attr, void * (*rutina)(void *), void*arg)
- pthread_t pthread_self(void)
- int pthread_join(pthread_t thread, void ** value)
- int pthread_exit(void * value)

En pthread_create tenemos 4 parámetros. El primer parámetro guardará el identificador del thread que creamos. El segundo valor es un puntero a struct con los atributos del hilo (podemos pasar NULL); el tercer parámetro es un puntero que recibirá una función. Esta función será lo que ejecute nuestro hilo.
pthread_t pthread_self devuelve el identificador del hilo que la ejecuta

pthread_join sirve para esperar otro hilo, 1 parámetro identificador del hilo a esperar, y otro parámetro con el valor de terminación del hilo.
pthread_exit finaliza el hilo.

Semáforos:



Semáforos

A veces es necesario que dos o más procesos o hilos accedan a un recurso común (escribir en un mismo fichero, leer la misma zona de memoria, escribir en la misma pantalla, etc). El problema es que si lo hacen simultáneamente una puede machacar la información del otro.

Para evitar este problema, están los semáforos. Un semáforo da acceso a uno de los procesos y se lo niega a los demás mientras el primero no termine. Junto con la memoria compartida y las colas de mensajes, son los recursos compartidos que suministra UNIX para comunicarse entre procesos.

Es como una variable contador, imaginando que el semáforo controla un fichero y que inicialmente tiene el valor de 1 ("está en verde"). Cuando un proceso quiere acceder al fichero, primero debe decrementar el semáforo. El contador queda a 0 y como no es negativo, deja que el proceso siga su ejecución y, por tanto acceda al fichero.

su ejecución y, por tanto acceso al fichero.

Ahora un segundo proceso lo intenta y para ello también decrementa el contador. Esta vez el contador se pone a -1 y como es negativo, el semáforo se encarga de que el proceso quede "bloqueado" y "dormido" en una cola de espera. Este segundo proceso no continuará por tanto su ejecución y no accederá al fichero.

Pasos para usar semáforos en un programa:

- Obtener una clave de semáforo. key_t ftok(char*, int). 1er parámetro: nombre y path de fichero, 2do parámetro: entero cualquiera.
- Obtener un array de semáforos. La función int semget(key_t, int, int) nos lo permite. 1er parámetro: la clave, 2do parámetro: número de semáforos que se quieren, 3er parámetro: flags.
- Inicialización, int semctl(int, int, int, int)
  - 1er Parámetro: identificador array de semáforos
  - 2do Parámetro: índice del semáforo.
  - 3er Parámetro: Flag ej → SETVAL
  - 4to Parámetro: valor del semáforo 1 "verde", 0 "rojo".

- El proceso que quiere acceder a un recurso debe decrementar el semáforo.

int semop(int, struct sembuf*, size_t)

- 1er Parámetro: identificador del array de semáforos obtenido con semget().
- 2do Parámetro: array de operaciones sobre el semáforo.
- 3er Parámetro: número de elementos en el array.

- Cuando el proceso termine de usar el recurso común, debe incrementar el semáforo. La función a utilizar es la misma, pero poniendo 1 en el campo sem_op de la estructura struct sembuf.

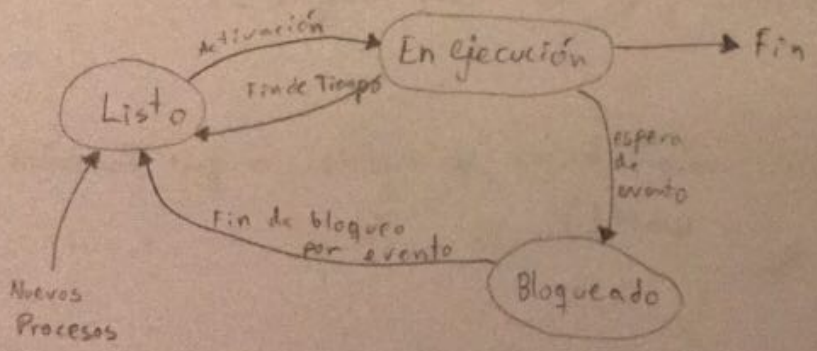
  - short sem_num: índice del array
  - short sem_op: valor que se le suma al valor del semáforo
  - short sem_flg: son flags que afectan la operación.

Procesos:

Procesos

Un proceso es un programa en ejecución. Está formado por el código del programa y el conjunto de datos asociados a la ejecución del programa. El proceso además posee una imagen de memoria, esto es el espacio de memoria en el que está autorizado. La imagen de memoria puede estar referida en memoria virtual o memoria física.

Ciclo de vida:

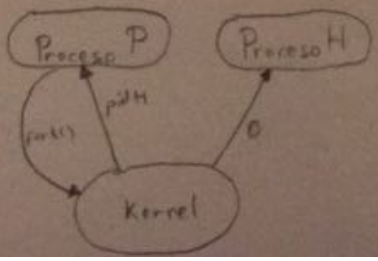


Podemos tener en ejecución tantos procesos como procesadores tenga nuestro equipo. El tiempo en ejecución lo decide el algoritmo o planificación que utilice nuestro sistema operativo.

proceso pesado (1 solo hilo de ejecución)

Procesos ligeros: los nuevos procesos obtienen directamente los recursos del sistema operativo o el proceso padre debe compartir recursos.

forK() : llamada al sistema para crear un nuevo proceso. fork crea una copia casi idéntica del padre (se copia todo el código) y continúan ejecutándose en paralelo. El proceso recibe de fork() el pid del hijo, mientras que el hijo recibe un 0. El hijo hereda recursos del padre (ficheros abiertos, estado de las variables, etc...), no hereda recursos como los señales pendientes, devuelve -1 en caso de error.



Por otra parte tenemos la función **exec()**. Esta función cambia la imagen del proceso actual. Lo que realiza es sustituir la imagen de memoria del programa por la de un programa diferente. Esta función normalmente la invocamos en un proceso hijo previamente generado por fork().

Funciones exec:
- int execl(const char *path, const char *arg, ...)
- int execv(const char *path, char *const argv[]);
- int execve(const char *path, char *const argv[], char *const envp[]);
- int execvp(const char *file, char *const argv[]);

* El padre puede crear más hijos o esperar a que termine el hijo.

wait()