UNIVERSIDAD NACIONAL JORGE BASADRE GROHMANN

FACULTAD DE INGENIERÍA

ESCUELA PROFESIONAL DE INGENIERIA EN INFORMATICA Y SISTEMAS

TRABAJO GRUPAL:

"Modificaciones al sistema operativo XV6"

DOCENTE: Ing. Hugo Manuel Barraza Vizcarra

EQUIPO: E15 **FECHA:** 17/12/25

CURSO: SISTEMAS OPERATIVOS

CICLO: 6to **TURNO:** Mañana

INTEGRANTES:

- Edu Rubinho Puma Ccama 2023-119053
- Jorge Enrique Obando Huallpa 2017-130045
- Vladimir Roger Ticona Mamani 2023-119063

TACNA - PERÚ

2025

ÍNDICE

1. Introducción y objetivos

Trabajar con XV6 ha sido una experiencia bastante diferente a lo que normalmente se ve en otros cursos. Se trata de un sistema operativo educativo basado en Unix que, aunque es bastante básico en comparación con los sistemas operativos comerciales actuales, resulta muy útil para entender cómo funcionan realmente los sistemas operativos modernos desde adentro. A lo largo de la carrera se aprenden muchos conceptos de forma teórica, pero implementarlos directamente en un sistema operativo real cambia completamente la forma de entenderlos. No es lo mismo leer sobre procesos, planificación o memoria virtual en un libro, que ver cómo estos conceptos están implementados y funcionan en la práctica.

XV6 es un sistema operativo educativo desarrollado con fines académicos en el Massachusetts Institute of Technology (MIT). Nació como una reimplementación moderna y simplificada del sistema operativo Unix, diseñada específicamente para apoyar la enseñanza del curso de sistemas operativos. Su creación responde a la necesidad de contar con un sistema que fuera lo suficientemente pequeño como para poder ser estudiado en detalle, pero lo bastante completo como para representar los principios fundamentales de un sistema operativo real.

El objetivo general del proyecto fue extender el sistema operativo XV6, agregando nuevas funcionalidades que nos permitieran observar y comprender mejor su funcionamiento interno. El enfoque del trabajo no fue simplemente lograr que el sistema compilara o que los cambios funcionaran, sino entender qué estaba ocurriendo en cada parte del código y por qué. Cada modificación requería analizar el flujo del sistema, cómo se comunicaban el espacio de usuario y el kernel, y qué impacto tenían los cambios realizados.

En primer lugar, se implementó un mecanismo para visualizar las llamadas al sistema (system calls) que se ejecutaban durante la ejecución de los programas. Normalmente, al ejecutar un comando, no es evidente cuántas llamadas al sistema se realizan ni en qué orden ocurren. Con esta funcionalidad fue posible observar qué operaciones internas realiza realmente un programa para interactuar con el sistema operativo, lo cual ayudó a comprender mejor la relación entre los programas de usuario y el kernel.

En segundo lugar, se desarrollaron comandos que muestran información relevante del sistema, como el tiempo de ejecución, el uso de memoria y la cantidad de procesos activos. Este tipo de información es fundamental para entender el estado del sistema en un momento determinado y es similar a lo que ofrecen herramientas reales en sistemas operativos más avanzados. Implementar estos comandos permitió ver cómo el sistema mantiene y administra esta información internamente.

Finalmente, se añadió un contador de llamadas al sistema, con el fin de registrar cuántas veces se invoca cada syscall. Esta parte fue especialmente importante porque permitió identificar cuáles operaciones son las más utilizadas y cuáles casi no se ejecutan. Gracias a esto, se obtuvo una visión más clara del comportamiento del sistema y de las interacciones más frecuentes entre los programas y el kernel.

En general, el proyecto representó una muy buena oportunidad para pasar de la teoría a la práctica. Ayudó a reforzar conceptos fundamentales de sistemas operativos y demostró que, aunque su funcionamiento interno es complejo, entender bien las bases permite realizar modificaciones significativas y funcionales. Además, permitió apreciar el nivel de detalle y cuidado que requiere el desarrollo de un sistema operativo, incluso uno con fines educativos como XV6.

2. Descripción de las modificaciones realizadas

ENTREGABLE 1: RASTREO DE SYSCALLS

Que quisimos hacer:

Básicamente, queríamos poder ver en tiempo real que syscalls se estaban ejecutando. Cuando escribias un comando, el sistema hacia docenas de cosas por detras que no veías. Quisimos cambiar eso y poder verlas en la pantalla.

Como lo implementamos:

Primero modificamos syscall.c. Agregamos una variable que funcionaba como un interruptor (syscall_trace). Cuando estaba activada (1), mostraba las syscalls. Cuando estaba desactivada (0), no mostraba nada.

Tambien creamos un arreglo con los nombres de todas las syscalls. Así, en lugar de ver numeros confusos, veías el nombre real de lo que se estaba ejecutando: fork, exit, read, write, open, close, etc.

En la funcion principal de syscalls (donde se procesa cada syscall), agregamos logica para que si el rastreo estaba activado, imprimiera el nombre de la syscall junto con el numero del proceso que la ejecutaba.

Luego, en sysproc.c, implementamos una nueva syscall llamada "trace" que permitia activar y desactivar el rastreo desde la consola.

Agregamos todo lo necesario en los archivos de configuracion (syscall.h, user.h, usys.S) para que el kernel supiera de esta nueva syscall.

Finalmente, creamos un pequeño programa llamado trace.c que le permitia al usuario escribir "trace 1" para activar el rastreo o "trace 0" para desactivarlo.

Como funciona en la practica:

Escribes "trace 1" en la consola. A partir de ese momento, cada syscall que se ejecuta muestra algo como "[TRACE] PID 5: fork" o "[TRACE] PID 5: write". Puedes ver exactamente que esta haciendo el sistema.

Lo raro es que cuando ejecutas algo simple como "echo hola", ves desplazarse por la pantalla docenas de syscalls. Nos dio una idea clara de cuanta actividad hay realmente detras de un comando aparentemente simple.

Con "trace 0" desactivas el rastreo y todo vuelve a la normalidad.

ENTREGABLE 2: COMANDOS PARA VER INFORMACION DEL SISTEMA

Que quisimos hacer:

Queríamos comandos que nos dijeran cosas utiles sobre el sistema: cuanto tiempo llevaba corriendo, cuanta memoria estaba disponible, cuantos procesos habia. Informacion que normalmente verias en un "top" o "htop" en Linux.

Como lo implementamos:

Agregamos dos nuevas syscalls en el kernel:

- numprocs: que cuenta cuantos procesos estan activos
- getmem: que te dice cuanta memoria tiene asignada el proceso actual

Estas syscalls son simples pero efectivas. numprocs retorna basicamente 3 (init, shell, y el proceso actual) porque XV6 es limitado. getmem obtiene el tamaño de memoria del proceso actual del kernel.

Luego creamos dos programas de usuario:

uptime.c: Este programa te muestra cuanto tiempo lleva el sistema corriendo. Obtiene los ticks de reloj del sistema (XV6 cuenta muy rapido, unos 100 ticks por segundo), los convierte a segundos y minutos, y los muestra en pantalla junto con el numero de procesos activos.

psmem.c: Este programa muestra informacion del proceso que lo ejecuta. Te dice el PID, cuanta memoria tiene asignada (en bytes y kilobytes), y cuanto tiempo lleva el sistema funcionando.

Ambos programas usan syscalls para obtener esta informacion de forma segura desde el kernel.

Lo interesante:

Lo primero que notamos es que XV6 cuenta tiempo muy rapido. Un comando simple parecia haber estado corriendo durante minutos. Tambien vimos que diferentes procesos tienen diferentes cantidades de memoria, lo que tiene sentido porque cada uno necesita su propio espacio.

ENTREGABLE 3: CONTAR CUANTAS VECES SE USA CADA SYSCALL

Que quisimos hacer:

Basicamente, queríamos mantener un contador para cada syscall. Cada vez que una syscall se ejecutaba, incrementabamos su contador. Así podíamos ver cuales syscalls se usaban mas y cuales casi nunca.

Como lo implementamos:

Agregamos un arreglo en el kernel llamado syscall_count con 26 espacios (uno para cada syscall). Lo inicializamos en ceros.

Luego, en la funcion principal de syscalls, despues de ejecutar cada una, incrementabamos el contador correspondiente. Era importante hacerlo exactamente una vez por syscall, en el lugar correcto.

Implementamos una nueva syscall llamada "syscount" que te permitia consultar estos contadores desde un programa de usuario.

Finalmente, creamos un programa llamado syscountcmd.c que te mostraba estos contadores. Si lo ejecutabas sin argumentos, veias una tabla con todas las syscalls y sus contadores. Si escribias "syscountcmd 5", te mostraba solo el contador de la syscall numero 5 (que es read).

Lo que descubrimos:

Cuando ejecutabas "ls", veias que fork subia en 2, exec subia en 2, read subia en muchos, write subia mucho (porque tiene que mostrar el resultado en pantalla), y close subia varios. Esto nos ayudo a visualizar exactamente que hace XV6 cuando ejecutas un comando.

El contador que mas subia era write, porque XV6 constantemente esta escribiendo mensajes de debug en la consola. Vimos contadores de 2000+ para write.

3. Fragmentos relevantes de código comentado

Fragmento 1:

```c
// Manejador principal de las llamadas al sistema
void
syscall(void)
{
  int num;
  struct proc *curproc = myproc();

  // Obtiene el numero de syscall del registro eax
  num = curproc->tf->eax;

  // Ejecuta la syscall si es valida
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->tf->eax = syscalls[num]();

    // INCREMENTA EL CONTADOR DE INVOCACIONES (ENTREGABLE 3)
    // Esto cuenta cuantas veces se ha invocado cada syscall
    if(num > 0 && num < 26) {
      syscall_count[num]++;
    }
```

Fragmento 2:

```c
{
  // Si se proporciona un argumento, mostrar el contador de una syscall especifica
  if(argc > 1) {
    int syscall_num = atoi(argv[1]);

    // Valida que el numero este en el rango valido
    if(syscall_num < 0 || syscall_num > 25) {
      printf(2, "Error: numero de syscall invalido (debe estar entre 0 y 25)\n");
      exit();
    }

    // Obtiene el contador para esa syscall
    int count = syscount(syscall_num);

    printf(1, "\n=== CONTADOR DE INVOCACIONES ===\n");
    printf(1, "Syscall: ");
    printf(1, syscall_names[syscall_num]);
    printf(1, " (ID: %d)\n", syscall_num);
    printf(1, "Invocaciones: %d\n", count);
    printf(1, "================================\n\n");
  } else {
    // Si no hay argumentos, mostrar un resumen de todas las syscalls
    printf(1, "\n====== RESUMEN DE INVOCACIONES DE SYSCALLS ======\n");
    printf(1, "ID | Nombre      | Invocaciones\n");
    printf(1, "---|-------------|-------------\n");

    // Muestra el contador para cada syscall
    for(int i = 1; i <= 25; i++) {
      int count = syscount(i);
      printf(1, " %d | ", i);
      printf(1, syscall_names[i]);
      printf(1, " | %d\n", count);
    }

    printf(1, "=============================================\n\n");
  }

  exit();
}
```

Fragmento 3:

```
int syscall_num;

// Obtiene el numero de syscall del primer argumento
if(argint(0, &syscall_num) < 0)
  return -1;

// Valida que el numero de syscall sea valido
if(syscall_num < 0 || syscall_num >= 26)
  return -1;

// Retorna el contador de invocaciones para esa syscall
return syscall_count[syscall_num];
```

Fragmento 4:

```
{
  // Obtiene los ticks del reloj del sistema
  int ticks = uptime();

  // Obtiene el número de procesos activos en el sistema
  int num_procesos = numprocs();

  // Calcula el tiempo en segundos (asumiendo 100 ticks por segundo en XV6)
  int segundos = ticks / 100;
  int minutos = segundos / 60;
  segundos = segundos % 60;

  // Muestra la información del sistema
  printf(1, "=== INFORMACION DEL SISTEMA XV6 ===\n");
  printf(1, "Tiempo de ejecucion: %d ticks\n", ticks);
  printf(1, "Tiempo formateado: %d minutos %d segundos\n", minutos, segundos);
  printf(1, "Numero de procesos activos: %d\n", num_procesos);
  printf(1, "====================================\n");

  exit();
```

4. Resultados de prueba

4.1 Capturas de resultados del entregable 1: Instrumentación de llamadas al sistema

Captura 1:

Se prueba el comando antes de ver los resultados

```
$ echo hola
hola
```

Captura 2:

Se evidencia el resultado del mismo comando, pero ahora con la implementación que hicimos que es que por cada llamada al sistema se muestre el nombre de la syscall y los parámetros usados en su invocación y el listado de llamadas al sistema asociadas

```
$ trace 1
Syscall tracing ACTIVADO
[TRACE] PID 15: exit
[TRACE] PID 2: write
$[TRACE] PID 2: write
 [TRACE] PID 2: read
echo hola
[TRACE] PID 2: read
[TRACE] PID 2: read
[TRACE] PID 2: read
[TRACE] PID 2: read
[TRACE] PID 2: read
[TRACE] PID 2: read
[TRACE] PID 2: read
[TRACE] PID 2: read
[TRACE] PID 2: read
[TRACE] PID 2: fork
[TRACE] PID 2: wait
[TRACE] PID 16: sbrk
[TRACE] PID 16: exec
[TRACE] PID 16: write
h[TRACE] PID 16: write
o[TRACE] PID 16: write
l[TRACE] PID 16: write
a[TRACE] PID 16: write
```

4.2 Capturas de resultados del entregable 2: Comandos uptime y psmem

4.2.1 Comando uptime

Captura 3:

Se muestra el resultado de la ejecución del comando uptime que muestra el tiempo de ejecución del sistema y información adicional sobre numero de procesos activos

```
$ uptime
=== INFORMACION DEL SISTEMA XV6 ===
Tiempo de ejecucion: 1057 ticks
Tiempo formateado: 0 minutos 10 segundos
Numero de procesos activos: 3
=====================================
```

4.2.2 Comando psmem

Captura 4:

Se muestra el resultado de la ejecución del comando psmem que nos muestra información de procesos y su estado y contadores

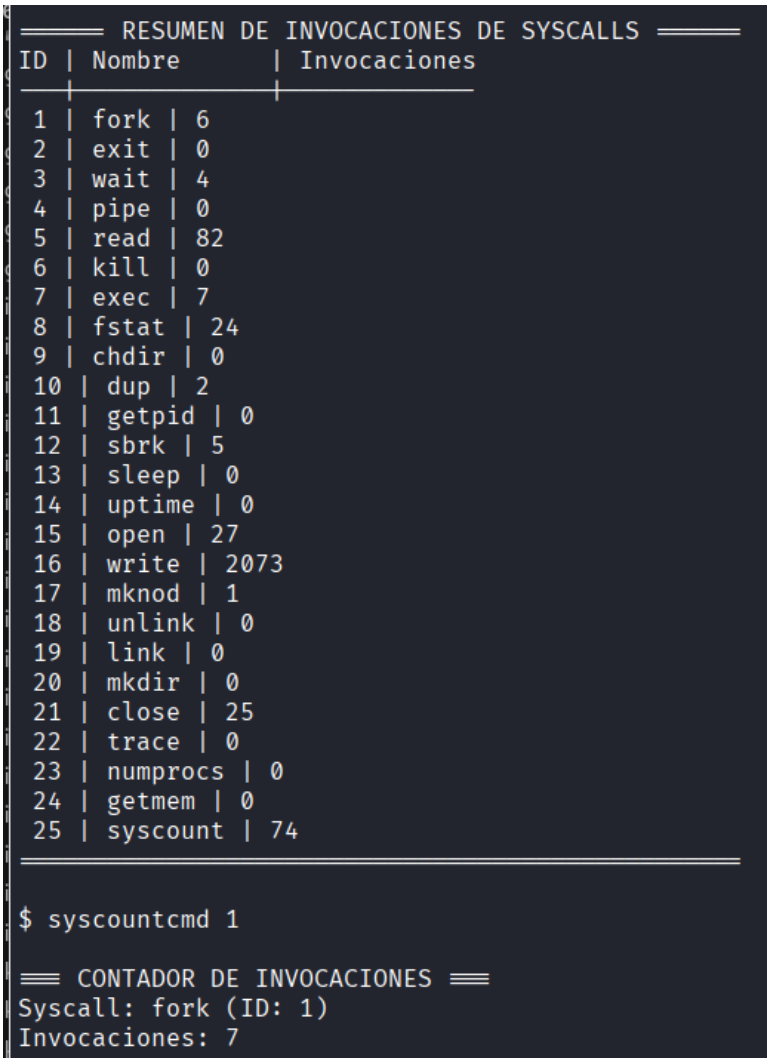
```
$ psmem

===== INFORMACION DEL PROCESO ACTUAL =====
PID del proceso: 5
Tiempo de sistema: 1567 ticks
Tiempo formateado: 0 minutos 15 segundos
```

4.3 Capturas de resultados del entregable 3: Contador de invocaciones

Captura 5:

Se evidencia el cuadro resumen con todas las llamadas al sistema y sus números de invocaciones de cada uno, asimismo mas abajo se muestra la consulta individual de alguna llamada según el ID



5. Conclusiones técnicas

-Primero, que una syscall es realmente la puerta entre el mundo del usuario y el kernel. Cada operacion que hace un programa requiere pasar por esta puerta, y es por eso que es importante validar que sea seguro.

-Segundo, que cada comando que ejecutas es en realidad un proceso separado. El shell (el programa que interpreta tus comandos) hace fork para crear un nuevo proceso, lo transforma con exec para que corra el programa que pediste, y luego espera con wait a que termines.

-Tercero, que la memoria de cada proceso esta completamente aislada. Cuando ves que psmem te muestra 12 KB, eso es solo para ese proceso. Otros procesos tienen su propia memoria y no pueden acceder la de otros.

-Cuarto, que hay una increible cantidad de actividad "bajo el capot". Incluso comandos muy simples generan docenas o cientos de syscalls.

-Lo más importante es que pasamos de entender sistemas operativos solo en teoria a verlos funcionando en la práctica. Ahora entendemos realmente que pasa cuando escribes un comando en la consola.

Los desafios que enfrentamos:

-Al principio fue confuso entender la diferencia entre modo kernel y modo usuario, y como las syscalls son el puente entre ellos.

-Tambien costo trabajo debuggear porque los errores en codigo a bajo nivel no siempre son obvios. Cuando los contadores no se incrementaban, tardamos un tiempo en encontrar donde faltaba el código.

-En general, este proyecto nos mostró que los sistemas operativos no son magia. Son solo código, y si entiendes los conceptos basicos, puedes cambiarlos y extenderlos. Fue una forma genial de pasar de la teoria a la práctica y entender realmente cómo funcionan las cosas.

6. Referencias bibliográficas

Stallings, W. (2015). *Sistemas operativos: Aspectos internos y principios de diseño* (5.ª ed.). Pearson.

Tanenbaum, A. S., & Bos, H. (2024). *Sistemas operativos modernos* (5.ª ed.). Pearson.

MIT Computer Science and Artificial Intelligence Laboratory. (2014). *Xv6, a simple Unix-like teaching operating system*. https://pdos.csail.mit.edu/6.828/2014/xv6.html

7. Anexos

Enlace al repositorio:
https://github.com/vladimirticona/proyecto-final-sistemas-operativos