UNIVERSIDAD NACIONAL JORGE BASADRE GROHMANN

FACULTAD DE INGENIERÍA

ESCUELA PROFESIONAL DE INGENIERÍA EN INFORMÁTICA Y SISTEMAS

PROYECTO:

Sistema Operativo UNIX (XV6)

ASIGNATURA:

Sistemas Operativos

DOCENTE:

MsC. Hugo Manuel Barraza Vizcarra

INTEGRANTES:

Arratia Paz, Russell Jhean Paul	**2023-119004**
Carita Pinchi, Nelly Adriana	**2023-119007**

TACNA - PERÚ

2025

ÍNDICE

I. INTRODUCCIÓN Y OBJETIVOS **3**

Objetivos Generales del Proyecto 3

II. DESCRIPCIÓN DE LAS MODIFICACIONES REALIZADAS **3**

Entregable 1: Instrumentación de llamadas al sistema 3

Entregable 2: Comandos de usuario relacionados con la Unidad II 4

Entregable 3: Contador de invocaciones por llamadas al sistema 5

III. FRAGMENTOS RELEVANTES DE CÓDIGO COMENTADO **6**

Entregable 1: Instrumentación de llamadas al sistema 6

Entregable 2: Comandos de usuario relacionados con la Unidad II 7

Entregable 3: Contador de invocaciones por llamadas al sistema 9

IV. RESULTADOS DE PRUEBAS **12**

Entregable 1: Instrumentación de llamadas al sistema 12

Entregable 2: Comandos de usuario relacionados con la Unidad II 13

Entregable 3: Contador de invocaciones por llamadas al sistema 14

V. CONCLUSIONES TÉCNICAS **15**

VI. REFERENCIAS BIBLIOGRÁFICAS **15**

VII. ANEXOS **15**

I. INTRODUCCIÓN Y OBJETIVOS

Este proyecto presenta la implementación de mejoras y extensiones al sistema operativo educativo XV6, un sistema basado en Unix versión 6 diseñado para enseñar conceptos fundamentales de sistemas operativos. El proyecto se desarrolla a través de tres entregables progresivos que incrementan la complejidad y funcionalidad del sistema. XV6 proporciona una base sólida para el aprendizaje de conceptos clave como: gestión de procesos y scheduling, memoria virtual y paginación, sistemas de archivos, entrada/salida, sincronización y concurrencia. A través de este proyecto, se implementan herramientas de diagnóstico, instrumentación del kernel y nuevos comandos de usuario que demuestran la interacción entre el espacio de usuario y el espacio del kernel.

Objetivos Generales del Proyecto

- Instrumentar el kernel XV6 para monitorear y registrar las llamadas al sistema (syscalls) en tiempo de ejecución, proporcionando visibilidad del comportamiento del sistema.
- Crear nuevas syscalls que permitan a los programas de usuario acceder a información del kernel de forma segura y controlada.
- Desarrollar comandos de usuario que utilicen las nuevas funcionalidades para consultar el estado del sistema operativo.
- Validar la seguridad mediante la implementación correcta de mecanismos de protección entre espacios de usuario y kernel.
- Demostrar la arquitectura de XV6 y los patrones establecidos para la extensión del sistema operativo.

II. DESCRIPCIÓN DE LAS MODIFICACIONES REALIZADAS

Entregable 1: Instrumentación de llamadas al sistema

Esta sección describe las modificaciones realizadas para implementar el tracing de syscalls, permitiendo registrar y mostrar el nombre de cada llamada al sistema junto con sus parámetros, activable mediante un comando de usuario.

Modificaciones Principales

- En ***syscall.c***: Se agregó una variable global syscall_tracing para controlar el estado del tracing. Se creó un arreglo syscall_names[] con los nombres de las syscalls existentes. Se modificó la función syscall() para imprimir el nombre y parámetros de cada syscall cuando el tracing está activado, usando un switch para manejar los parámetros específicos de cada syscall.

- En ***sysproc.c***: Se implementó una nueva syscall sys_trace() que permite activar (1) o desactivar (0) el tracing desde espacio de usuario.

- En ***syscall.h***: Se agregó la definición SYS_trace para la nueva syscall.

- En ***user.h***: Se declaró la función trace(int) para uso en programas de usuario.

- En ***usys.S***: Se agregó el enlace de ensamblador SYSCALL(trace) para conectar la syscall.

- En ***trace.c***: Se creó un nuevo comando de usuario que toma argumentos "on" o "off" y llama a trace(1) o trace(0), imprimiendo confirmaciones.

- En ***Makefile***: Se agregó _trace a la lista UPROGS para compilar el comando.

Entregable 2: Comandos de usuario relacionados con la Unidad II

Esta sección llama a una syscall para acceder a información de procesos (getprocs), emplea un comando para mostrar el tiempo de actividad del sistema (uptime), y un comando para visualizar información de procesos activos (schedinfo).

Modificaciones Principales

- En ***sysproc.c:*** Se implementó sys_getprocs() que valida el buffer del usuario con argptr(), obtiene el parámetro max con argint(), itera sobre ptable.proc[] copiando información de procesos activos (PID, nombre con strncpy() limitado a 15 caracteres, estado) y retorna el número de procesos copiados.
- En ***usys.S***: Se agregó el enlace SYSCALL(getprocs) para la transición a kernel-space.
- En ***uptime.c***: Se creó un nuevo comando que llama a uptime(), divide entre 100 para convertir ticks a segundos, e imprime el resultado.
- En ***schedinfo.c***: Se creó un nuevo comando que define un arreglo state_names[] con los nombres de estados, llama a getprocs(), imprime una

tabla formateada con PID, nombre y estado de cada proceso, y muestra el total de procesos.
- En ***Makefile***: Se agregaron _uptime y _schedinfo a la lista UPROGS para compilar e incluir en fs.img.

Entregable 3: Contador de invocaciones por llamadas al sistema

Esta sección describe las modificaciones realizadas para implementar un contador global de invocaciones de cada syscall en el kernel de xv6. Esto permite al usuario consultar el número total de veces que cada syscall ha sido invocada desde el arranque del sistema, sin afectar el rendimiento normal. Los contadores se almacenan en memoria del kernel y se acceden mediante una nueva syscall, con un comando de usuario para mostrar los resultados.

Modificaciones Principales

- En ***syscall.c***: Se agregó un arreglo global syscall_counts[NSYSCALLS] para almacenar los contadores de cada syscall. Se modificó la función syscall() para incrementar el contador correspondiente después de ejecutar cada syscall válida. Se agregó el extern para sys_getcounts.

- En ***syscall.h***: Se agregó el define SYS_getcounts para la nueva syscall y NSYSCALLS para definir el tamaño del arreglo de contadores.

- En ***sysproc.c***: Se implementó la nueva syscall sys_getcounts() que copia el arreglo de contadores al espacio de usuario mediante copyout.

- En ***user.h***: Se declaró la función getcounts(unsigned int*) para uso en programas de usuario.

- En ***usys.S***: Se agregó el enlace de ensamblador SYSCALL(getcounts) para conectar la syscall.

- En ***count.c***: Se creó un nuevo comando de usuario que llama a getcounts() para obtener los contadores y los muestra en pantalla, con nombres de syscalls. Soporta argumentos para mostrar contadores específicos por nombre o número

- En ***Makefile***: Se agregó _count a la lista UPROGS para compilar el comando.

III. FRAGMENTOS RELEVANTES DE CÓDIGO COMENTADO

Entregable 1: Instrumentación de llamadas al sistema

```
// Variable global para controlar el tracing de syscalls (0 = desactivado, 1 =
activado)
int syscall_tracing = 0;

// Arreglo con nombres de syscalls, indexado por número (SYS_fork = 1, etc.)
static char *syscall_names[] = {
  "",
  "fork", "exit", "wait", "pipe", "read", "kill", "exec",
  "fstat", "chdir", "dup", "getpid", "sbrk", "sleep",
  "uptime", "open", "write", "mknod", "unlink",
  "link", "mkdir", "close", "trace"  // Agregado para la nueva syscall
};
```

Fragmento 1: Variable y arreglo de nombres

```
int
sys_trace(void)
{
  int enable;

  if(argint(0, &enable) < 0)
    return -1;
  syscall_tracing = enable;  // Activa (1) o desactiva (0) el tracing global
  return 0;
}
```

Fragmento 2: Nueva syscall

```
int
main(int argc, char *argv[])
{
  if(argc != 2){
    printf(2, "Uso: trace on|off\n");  // Mensaje de error si argumentos incorrectos
    exit();
  }
  if(strcmp(argv[1], "on") == 0){
    trace(1);  // Llamar a syscall para activar tracing
    printf(1, "Tracing activado\n");
  } else if(strcmp(argv[1], "off") == 0){
    trace(0);  // Desactivar
    printf(1, "Tracing desactivado\n");
  } else {
```

```
    printf(2, "Argumento invalido: use 'on' o 'off'\n");
    exit();
  }
  exit();
}
```

Fragmento 3: Comando de usuario

Entregable 2: Comandos de usuario relacionados con la Unidad II

```
// Definida en user.h
// Estructura para información de procesos
struct proc_info {
  int pid;          // Identificador único del proceso
  char name[16];    // Nombre del proceso (máximo 15 caracteres + null terminator)
  int state;        // Estado del proceso
                    // 0=UNUSED   : Slot no utilizado en ptable
                    // 1=EMBRYO   : Proceso siendo creado por fork
                    // 2=SLEEPING : Bloqueado esperando evento
                    // 3=RUNNABLE : Listo para ejecutarse
                    // 4=RUNNING  : Actualmente ejecutándose
                    // 5=ZOMBIE   : Terminado, esperando que padre haga wait
};
```

Fragmento 1: Estructura proc_info

```
// Definida en sysproc.c
int sys_getprocs(void)
{
  struct proc_info *buf;  // Buffer donde copiar información
  int max, count = 0;     // max=máximo de procesos a retornar, count=procesos
copiados
  struct proc *p;         // Puntero para iterar tabla de procesos

  // Validar que el buffer sea accesible desde el espacio de usuario
  // argptr() verifica que la dirección y tamaño sean válidos
  if(argptr(0, (void*)&buf, sizeof(struct proc_info)) < 0)
    return -1;

  // Obtener el parámetro max del usuario
  if(argint(1, &max) < 0)
    return -1;

  // Iterar sobre la tabla global de procesos (ptable.proc)
```

```
  for(p = ptable.proc; p < &ptable.proc[NPROC] && count < max; p++){
    if(p->state != UNUSED){  // Solo copiar procesos en uso
      buf[count].pid = p->pid;                    // PID del proceso
      strncpy(buf[count].name, p->name, 15);          // Nombre (max 15 chars)
      buf[count].name[15] = 0;                     // Null terminator
      buf[count].state = p->state;                 // Estado (0-5)
      count++;  // Incrementar contador de procesos copiados
    }
  }

  return count;  // Retornar cantidad de procesos obtenidos
}
```

Fragmento 2: Implementación de sys_getprocs()

```
// En syscall.c - variable global para control de tracing
int syscall_tracing = 0;  // 0 = desactivado, 1 = activado

// Tabla de nombres de syscalls para impresión legible
static char *syscall_names[] = {
  "",
  "fork", "exit", "wait", "pipe", "read", "kill", "exec",
  "fstat", "chdir", "dup", "getpid", "sbrk", "sleep",
  "uptime", "open", "write", "mknod", "unlink",
  "link", "mkdir", "close"
  // ... puede extenderse con nuevos nombres
};

// En la función syscall() - verificación de trazado
if(syscall_tracing && num > 0 && num < NELEM(syscall_names) &&
syscall_names[num]) {
  cprintf("syscall: %s", syscall_names[num]);  // Imprimir nombre
  // ... código que imprime parámetros específicos de cada syscall
  cprintf("\n");
}
```

Fragmento 3: Sistema de Instrumentación

```
int main(int argc, char *argv[])
{
  uint ticks;

  // uptime() es una syscall existente que retorna ticks desde inicio
  ticks = uptime();
```

```
  // XV6 usa ~100 ticks por segundo, así que dividimos para obtener segundos
  printf(1, "Uptime: %d ticks (%d seconds)\n", ticks, ticks / 100);

  exit();
}
```

Fragmento 4: Comando uptime.c

```
int main(int argc, char *argv[])
{
  struct proc_info procs[64];  // Buffer: puede almacenar hasta 64 procesos
  int count, i;

  // Llamar syscall getprocs() para obtener lista de procesos
  // Retorna: número de procesos obtenidos, o -1 si error
  count = getprocs(procs, 64);

  if(count < 0) {
    printf(1, "schedinfo: error al obtener información de procesos\n");
    exit();
  }

  // Imprimir tabla de procesos
  printf(1, "PID    NAME       STATE\n");
  for(i = 0; i < count; i++) {
    printf(1, "%d    %s       %s\n",
           procs[i].pid,
           procs[i].name,
           state_names[procs[i].state]);  // Convertir estado a texto
  }

  printf(1, "\nTotal de procesos: %d\n", count);
  exit();
}
```

Fragmento 5: Comando schedinfo.c

Entregable 3: Contador de invocaciones por llamadas al sistema

```
// Arreglo para contar invocaciones de cada syscall (indexado por número)
unsigned int syscall_counts[NSYSCALLS] = {0};

// arreglo de nombres de syscalls para el trazado
static char *syscall_names[] = {
  "",
  "fork", "exit", "wait", "pipe", "read", "kill", "exec",
  "fstat", "chdir", "dup", "getpid", "sbrk", "sleep",
```

```
  "uptime", "open", "write", "mknod", "unlink",
  "link", "mkdir", "close", "trace", "getprocs", "getcounts"
};
```

Fragmento 1: Arreglo de contadores y nombre para trace

```
void
syscall(void)
{
  int num;
  struct proc *curproc = myproc();

  num = curproc->tf->eax;
  // Si el trazado de syscalls está activado, imprimir el nombre de la syscall
  if(syscall_tracing && num > 0 && num < NELEM(syscall_names) &&
syscall_names[num]) {
    // ... (código de tracing existente)
  }
  if(num > 0 && num < NSYSCALLS && syscalls[num]) {
    curproc->tf->eax = syscalls[num]();
    syscall_counts[num]++;  // Incrementar contador de la syscall
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            curproc->pid, curproc->name, num);
    curproc->tf->eax = -1;
  }
}
```

Fragmento 2: Inclusión de syscall_count en syscall

```
int
sys_getcounts(void)
{
  unsigned int *counts;
  if(argptr(0, (char**)&counts, NSYSCALLS * sizeof(unsigned int)) < 0)
    return -1;
  // Copiar los contadores al espacio de usuario
  if(copyout(myproc()->pgdir, (uint)counts, syscall_counts, NSYSCALLS *
sizeof(unsigned int)) < 0)
    return -1;
  return 0;
}
```

Fragmento 3: Definición de sys_counts

```
int
main(int argc, char *argv[])
{
  unsigned int counts[MAX_SYSCALLS];
  char *names[] = {
    "", "fork", "exit", "wait", "pipe", "read", "kill", "exec",
    "fstat", "chdir", "dup", "getpid", "sbrk", "sleep",
    "uptime", "open", "write", "mknod", "unlink",
    "link", "mkdir", "close", "trace", "getprocs", "getcounts"
  };

  if(getcounts(counts) < 0){
    printf(2, "Error al obtener contadores\n");
    exit();
  }

  if(argc == 1){
    printf(1, "Contadores de syscalls:\n");
    for(int i = 1; i < MAX_SYSCALLS; i++){
      if(counts[i] > 0){
        printf(1, "%s: %d\n", names[i], counts[i]);
      }
    }
  } else if(argc == 2){
    // Buscar por nombre o número
    char *arg = argv[1];
    int num = -1;
    if(arg[0] >= '0' && arg[0] <= '9'){
      num = atoi(arg);
    } else {
      for(int i = 1; i < MAX_SYSCALLS; i++){
        if(strcmp(names[i], arg) == 0){
          num = i;
          break;
        }
    }
    if(num < 1 || num >= MAX_SYSCALLS){
      printf(2, "Syscall '%s' no encontrado\n", arg);
      exit();
    }
    printf(1, "%s: %d\n", names[num], counts[num]);
  } else {
    printf(2, "Uso: count [nombre o número de syscall]\n");
    exit();
  }
  exit();
}
```

Fragmento 4: Creación de count.c para el usuario

IV. RESULTADOS DE PRUEBAS

Entregable 1: Instrumentación de llamadas al sistema

Prueba 1: Activación de Tracing

```
init: starting sh
$ trace on
syscall: write(1, 2f8f, 1)
Tsyscall: write(1, 2f8f, 1)
rsyscall: write(1, 2f8f, 1)
```

Prueba 2: Tracing de syscall en ls

```
ls
syscall: read(0, 3f9f, 1)
syscall: read(0, 3f9f, 1)
syscall: fork
syscall: wait
syscall: sbrk(32768)
syscall: exec("ls", bfac)
syscall: open(".", 0)
syscall: fstat(3, 2d7c)
syscall: read(3, 2d6c, 16)
syscall: open("./.", 0)
```

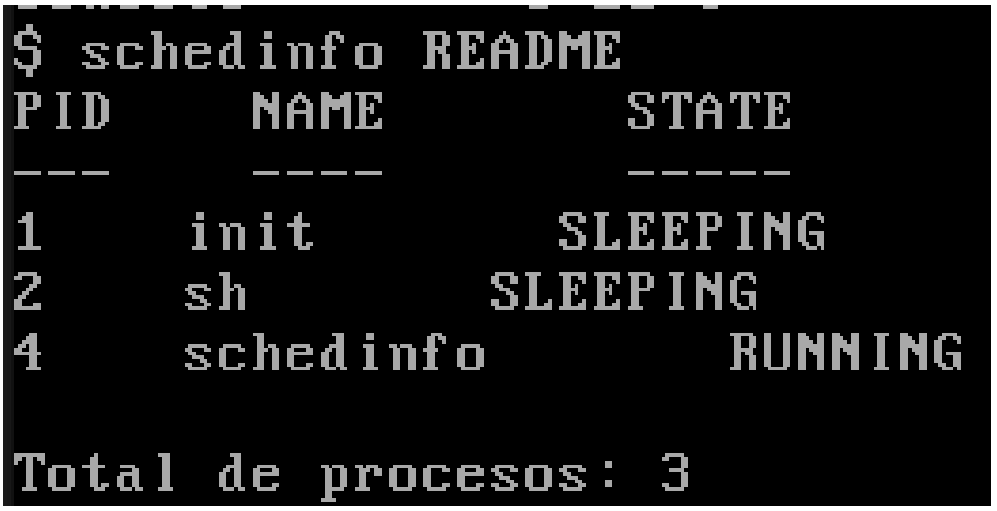
Prueba 3: Desactivación del tracing

```
trace off
syscall: read(0, 3f9f, 1)
syscall: read(0, 3f9f, 1)
syscall: read(0, 3f9f, 1)
syscall: read(0, 3f9f, 1)
syscall: read(0, 3f9f, 1)
syscall: read(0, 3f9f, 1)
syscall: read(0, 3f9f, 1)
syscall: read(0, 3f9f, 1)
syscall: read(0, 3f9f, 1)
syscall: fork
syscall: wait
syscall: sbrk(32768)
syscall: exec("trace", bfac)
syscall: trace(0)
Tracing desactivado
```

Entregable 2: Comandos de usuario relacionados con la Unidad II

Prueba 1: Uptime del proceso *grep*

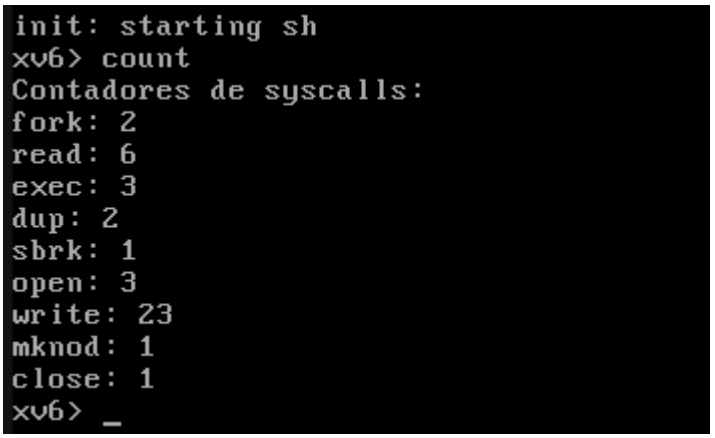
Prueba 2: schedinfo en README



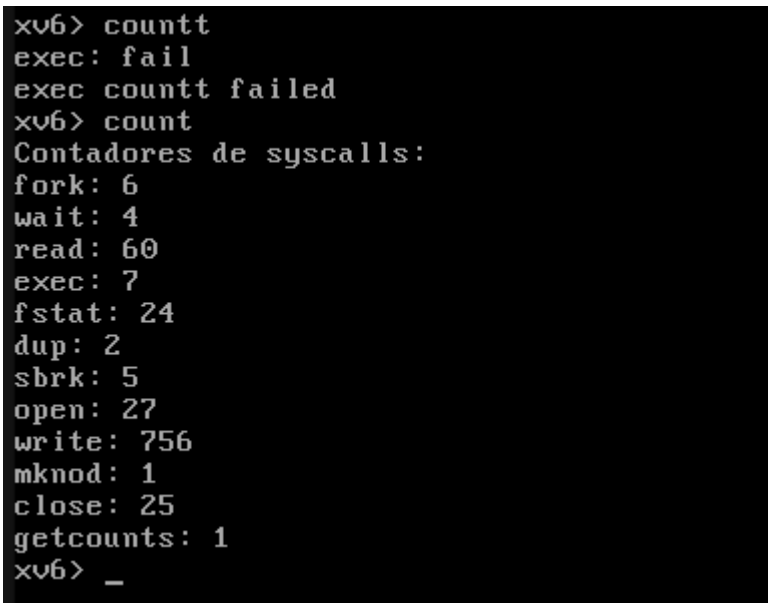
Prueba 3: Múltiples procesos con schedinfo

Entregable 3: Contador de invocaciones por llamadas al sistema

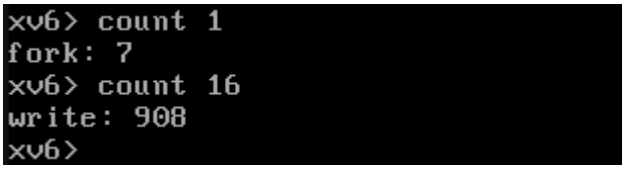
Prueba 1: Contadores iniciales



Prueba 2: Contadores después de comandos



Prueba 3: Consulta específica por identificador.



V. CONCLUSIONES TÉCNICAS

- Se implementó correctamente una herramienta de rastreo (trace) que permite ver qué llamadas al sistema (syscalls) se están ejecutando y con qué parámetros, lo cual es útil para depurar y entender el comportamiento del sistema operativo.
- Se desarrollaron comandos de usuario como uptime para ver el tiempo de actividad del sistema y schedinfo para listar y mostrar el PID, nombre y estado de los procesos activos, interactuando con nuevas funcionalidades del kernel (como sys_getprocs).
- Se añadió un contador global para registrar cuántas veces se invoca cada llamada al sistema (syscall) desde el inicio. Esto se demuestra con el comando count, permitiendo al usuario auditar la frecuencia de uso de las funcionalidades del kernel.

VI. REFERENCIAS BIBLIOGRÁFICAS

- Código fuente de XV6 – Repositorio oficial de MIT.
  Disponible en: https://github.com/mit-pdos/xv6-public
- Libro de XV6 – "Xv6, a simple Unix-like teaching operating system".
  MIT 6.828 / Fall 2014.
  Disponible en: http://pdos.csail.mit.edu/6.828/2014/xv6/book-rev8.pdf
- Documentación del código XV6 – "xv6.pdf" (código fuente comentado).
  Material complementario del curso MIT 6.828.)

VII. ANEXOS

Enlace al repositorio de github
https://github.com/LARMD-2/Proyecto-U2-SO.git