Practica 1 - Fork - SisOps

1. Información Administrativa y del Curso

- **Elementos Básicos:**
  - Página oficial de la materia, formulario de alta, calendario con fechas importantes y canales de comunicación (Discord, correo).
  - Se realizan 1 laboratorio, 3 trabajos prácticos y 1 parcial, sin examen final.
  - Instrucciones para la integración y manejo del repositorio de Git (clonar, ramas, pull request).

Esta sección es importante para conocer el régimen de cursada y las herramientas de la materia, aunque su contenido es más administrativo que técnico.

---

2. Introducción al Entorno Unix

- **Componentes Clave:**
  - **Sistema Operativo:** Conjunto de utilidades, kernel, drivers, shell y librerías.
  - **Unix y sus derivados:** Linux, GNU, BSD, entre otros.
  - **Estándares POSIX y SUSv4:** Definen interfaces portables, privilegiando la implementación mínima y orientada a aplicaciones.
  - **Filosofía Unix:**
    - Se favorece la creación de utilidades pequeñas que hagan “una cosa y la hagan bien”.
    - Se promueve el uso de pipes, donde la salida de un comando se convierte en la entrada de otro, facilitando el scripting y la automatización.

---

3. Conceptos Básicos de la Línea de Comandos y Navegación

- **Comandos Esenciales:**
  - **Navegación y manejo de archivos:** `pwd`, `ls`, `cd`, `mkdir`, `touch`, `mv`, `file`
  - **Ayuda y documentación:** `man`, `info`, `whatis`, `apropos`, `history`
  - **Identificación:** `whoami`, `uname`, `whereis`, `who`

Estos comandos son la base para interactuar con el entorno Unix y comprender la estructura de los sistemas operativos.

---

4. Creación y Gestión de Procesos con fork(2)

- **fork(2):**
  - Es la syscall que permite crear un nuevo proceso. El proceso creado (hijo) es una copia casi idéntica del proceso que lo invoca (padre).
  - La única diferencia es el valor de retorno:

    > **Al proceso padre se le devuelve el PID del hijo.**

    > **Al proceso hijo se le devuelve 0.**

  - Se debe controlar el posible fallo de fork y, en algunos ejemplos, se emplea para distinguir el comportamiento entre padre e hijo.
- **Ejemplos Prácticos:**

```c
int main(int argc, char* argv[]) {
  printf("Mi PID es: %d\n", getpid());

  int i = fork();

  if (i < 0) {
    printf("Error en fork! %d\n", i);
    exit(-1);
  }

  if (i == 0) {
    printf("Soy el proceso hijo y mi pid es: %d\n", getpid());
  } else {
    printf("Soy el proceso padre y mi pid es: %d\n", getpid());
  }

  printf("Terminando\n");

  exit(0);
}
```

Ejemplo 0: Imprime mensajes distintos en el padre y en el hijo, según el valor devuelto por fork.

```c
int main(int argc, char* argv[]) {
  int a = 4;

  int i = fork();

  a = 5;

  if (i < 0) {
    printf("Error en fork! %d\n", i);
    exit(-1);
  }

  if (i == 0) {
    printf("[hijo] mi pid es: %d\n", getpid());
    printf("[hijo] a=%d\n", a);
  } else {
    a = 6;
    printf("[padre] mi pid es: %d\n", getpid());
    printf("[padre] a=%d\n", a);
  }

  printf("Terminando\n");
  exit(0);
}
```

Ejemplo 1: Se muestra la modificación de variables tras el fork, evidenciando que la copia de memoria es independiente en cada proceso (aunque pueden tener la misma dirección en el espacio virtual, son distintas copias).

Comprender fork es esencial, ya que es la base para la multitarea en Unix. Recuerda que cada proceso tiene su propio espacio de memoria y una tabla de file descriptors.

---

5. Comunicación entre Procesos con pipe(2)

- **pipe(2):**
  - Crea un par de descriptores de archivo conectados, formando un “canal” unidireccional para comunicación.
  - Un extremo se utiliza para escribir y el otro para leer.

> **La lectura se bloquea hasta que haya datos, y la escritura se bloquea si el buffer se llena.**

- **Ejemplos:**

```
int main(int argc, char* argv[]) {
  int fds[2];
  int msg = 42;

  int r = pipe(fds);

  if (r < 0) {
    perror("Error en pipe");
    exit(-1);
  }

  printf("Lectura: %d, Escritura: %d\n", fds[0], fds[1]);

  // read(fds[0], &msg, sizeof(msg)); // (en este caso
                                      el read espera
                                      infinitamente)
  // Escribo en el pipe
  write(fds[1], &msg, sizeof(msg));

  int recibido = 0;
  read(fds[0], &recibido, sizeof(recibido));
  printf("Recibi: %d\n", recibido);

  close(fds[0]);
  close(fds[1]);
}
```

Ejemplo 0: Demuestra la escritura y posterior lectura de un mensaje (por ejemplo, el número 42).

  - Es importante cerrar los descriptores no utilizados en cada proceso para evitar “fugas” de file descriptors y para asegurar la sincronización entre procesos (por ejemplo, qué sucede si un proceso lee antes de que el otro escriba).

```
int main() {
   int pipefd[2];  // Array para el pipe: [0] lectura, [1] escritura
   pipe(pipefd);

   int pid = fork();
```

```
    if (pid == -1) {
        perror("Error en fork");
        exit(1);
    }

    if (pid == 0) {  // Proceso hijo
        char mensaje[20];
        // ❌ Error: No cerramos el descriptor de escritura en el hijo
        read(pipefd[0], mensaje, sizeof(mensaje));
        printf("Hijo recibió: %s\n", mensaje);
        // ❌ No cerramos pipefd[0] después de leer
    } else {  // Proceso padre
        char *mensaje = "Hola, hijo!";
        // ❌ Error: No cerramos el descriptor de lectura en el padre
        write(pipefd[1], mensaje, 12);
        // ❌ No cerramos pipefd[1] después de escribir
    }

    return 0;
}
```

```
int main() {
    int pipefd[2];
    pipe(pipefd);

    int pid = fork();

    if (pid == -1) {
        perror("Error en fork");
        exit(1);
    }

    if (pid == 0) {  // Proceso hijo
        close(pipefd[1]);  // ✅ Cierra escritura, solo leerá
```

```
        char mensaje[20];
        read(pipefd[0], mensaje, sizeof(mensaje));
        printf("Hijo recibió: %s\n", mensaje);

        close(pipefd[0]);  // ✅ Cierra lectura después de usarla

    } else {  // Proceso padre
        close(pipefd[0]);  // ✅ Cierra lectura, solo escribirá

        char *mensaje = "Hola, hijo!";
        write(pipefd[1], mensaje, 12);

        close(pipefd[1]);  // ✅ Cierra escritura después de usarla
    }

    return 0;
}
```

El uso correcto de pipe es fundamental en la creación de procesos que *colaboran*, por ejemplo, en el clásico “ping-pong” entre padre e hijo.

---

6. Sincronización y Terminación de Procesos: wait(2) y waitpid(2)

- **wait(2):**
  - Permite a un proceso padre esperar la terminación de uno de sus hijos.
  - Es una llamada bloqueante que retorna el PID del hijo finalizado y, a través de macros como `WEXITSTATUS`, se puede conocer el código de salida.
  - Se abordan conceptos importantes como procesos huérfanos y zombies:

```c
int main(int argc, char* argv[]) {
  int fds[2];
  pipe(fds);
  int msg;
  int i = fork();

  if (i == 0) {
    close(fds[1]);
    printf("[hijo] Mi pid es: %d\n", getpid());
    printf("[hijo] Mi ppid es: %d\n", getppid());
    read(fds[0], &msg, sizeof(msg));
    sleep(2); // Wait for parent to die

    printf("[hijo] Mi ppid es: %d\n", getppid());
    printf("[hijo] Termina\n");
    close(fds[0]);
    _exit(17);
  } else {
    close(fds[0]);
    printf("[padre] Mi pid es: %d\n", getpid());
    printf("[padre] Terminas sin esperar\n");
    close(fds[1]);
  }
}
```

Huérfanos: Cuando el padre termina sin esperar al hijo, el proceso hijo es adoptado por el proceso init (o un subreaper).

```c
int main(int argc, char* argv[]) {
  int i = fork();

  if (i == 0) {
    printf("[hijo] Mi pid es: %d\n", getpid());
    printf("[hijo] Mi ppid es: %d\n", getppid());
    printf("[hijo] Termina\n");
    _exit(17);
  } else {
    printf("[padre] Mi pid es: %d\n", getpid());
    // Simulamos que el padre hace
    /// otras tareas sin hacer wait
    while (1) {}
    printf("[padre] Terminas sin esperar\n");
  }
}
```

Zombies: Ocurren cuando un proceso hijo finaliza y el padre no recoge su estado mediante wait.

- **waitpid(2):**

- Permite esperar un proceso hijo específico, ofreciendo mayor control, con opciones como `WNOHANG` para no bloquear.

Estas syscalls son clave para evitar que queden procesos zombies y para gestionar adecuadamente la finalización de procesos en programas concurrentes.

---

7. Cambio de Imagen de Proceso con exec(3) y sus Variantes

- **execve(2) y Familia exec(3):**
  - Cambian la imagen del proceso, reemplazando el código, datos y entorno del proceso actual por los de otro programa, sin cambiar el PID ni los file descriptors abiertos.
  - Entre las variantes se destacan `execl`, `execlp`, `execle`, `execv`, `execvp` y `execvpe`, siendo `execvp` la más utilizada por su capacidad para buscar el binario en el PATH y recibir los argumentos en forma de array.
- **Ejemplos:**
  - Se muestran programas donde tras llamar a execvp, el mensaje posterior (por ejemplo, “Terminando”) no se imprime, ya que la imagen del proceso se ha reemplazado.

Esta familia de funciones es esencial para ejecutar nuevos programas desde un proceso, manteniendo la estructura de la aplicación pero sustituyendo su contenido.

---

8. Tareas Prácticas y Ejercicios

- **Lab fork – Ping Pong:**
  - Se implementa la comunicación entre padre e hijo usando fork y pipe, donde el padre envía un mensaje al hijo y este responde, demostrando la creación y comunicación entre procesos.
- **Tarea: Primes (Criba de Eratóstenes):**
  - Se plantea el desafío de imprimir números primos menores o iguales a N.

    - La idea es generar una secuencia de números y utilizar procesos “filtros” (cada uno eliminando los múltiplos del primer número recibido) para ir dejando pasar solo números primos.
    - Aspectos a cuidar: esperar a que los procesos terminen, manejo adecuado de file descriptors para evitar fugas y escalabilidad en función del valor de N.
- **Tarea: xargs:**
    - Se trata de leer líneas de entrada y ejecutar un comando pasando esas líneas como argumentos, de forma secuencial o paralela (con posibilidad de hasta 4 ejecuciones paralelas mediante un flag).
    - Esto refuerza el uso de execvp y la coordinación con wait para controlar la ejecución de procesos.
- **Uso de Git:**
    - Se incluyen instrucciones detalladas para clonar, crear ramas y realizar pull requests en el repositorio del lab.
    - Es importante seguir el flujo correcto (crear ramas base y de entrega, integrar cambios y finalmente crear el PR) para mantener un historial limpio.

---

9. Otras Syscalls y Recursos Adicionales

- **Otras llamadas útiles:**
    - `open(2)`, `read(2)`, `write(2)`, `dup(2)` (para duplicar file descriptors), `kill(2)` (para enviar señales), `sigaction(2)` (para manejar señales) y `syscalls(2)` (para ver la lista completa de syscalls).
- **Recursos de Consulta:**
    - Páginas de manual (`man`), el libro *The Linux Programming Interface* de Michael Kerrisk, *The Missing Semester*, y sitios como Linux Journey para profundizar en el uso de herramientas.

Resumen Final

La lectura “Fork” abarca desde aspectos administrativos del curso hasta los fundamentos de la programación en Unix. Se profundiza en la creación de procesos mediante fork(2), la comunicación entre procesos a través de pipe(2), y la sincronización de la terminación de procesos con wait(2) y waitpid(2). Además, se estudia el reemplazo de la imagen de proceso usando la familia exec, que permite ejecutar nuevos programas manteniendo ciertos atributos del proceso original. Complementariamente, se proponen tareas prácticas (como la implementación de la criba de Eratóstenes para calcular primos y el desarrollo de una versión de xargs) que refuerzan estos conceptos y promueven el aprendizaje a través de la práctica. Finalmente, se incluyen instrucciones para el manejo del repositorio Git, fundamentales para la integración y entrega de los ejercicios en la materia.