Enunciado: Sistemas Operativos / Taller IPC / Anillo de pipes

Realice un esquema de comunicación en forma de anillo para intercomunicar a los N procesos hijo de la siguiente forma:

- Cada uno de los procesos hijo debe comunicarse con exactamente dos procesos: su antecesor y su sucesor (visto desde el orden de creación de los mismos). Al recibir un número entero de su antecesor, lo incrementan en 1, y lo envían a su sucesor usando pipes.
- El hijo $0 \leq K < N$ es especial, se encarga de iniciar y terminar el flujo de mensajes circulando por el anillo. Además de realizar las acciones comunes a todos los hijos, debe generar de forma aleatoria un número secreto entre 0 y 42 que solo él conoce. Luego, al recibir un número mayor a su número secreto, notifica al padre el número recibido y termina la comunicación dentro del anillo.
- Para comenzar la comunicación, el padre envía al hijo K un número aleatorio entre 0 y 10.
- El padre debe esperar a recibir del hijo K el máximo número alcanzado dentro del anillo e imprimirlo por stdout antes de finalizar.

Nota: Adapté el enunciado para que sea más claro y quité algunos requerimientos básicos no relacionados con IPC.

Contexto y conocimientos previos

El taller se presenta luego de haber dado las clases teóricas y prácticas de IPC. Por lo tanto se asume que las y los alumnos ya están familiarizados con los conceptos de IPC (creación de procesos hijo con fork, signals, pipes, file descriptors, syscalls read y write). El ejercicio seleccionado presenta un desafío interesante y una excelente oportunidad para introducir la topología de anillo, asentar el entendimiento del EOF (end-of-file), y cómo podemos usarlo a nuestro favor.

Considero que este ejercicio podría ser presentado como el último ejercicio dado en la clase práctica con resolución en vivo en vez de ser parte del taller. En esta prueba de oposición encaro la resolución del ejercicio con este enfoque.

Resolución

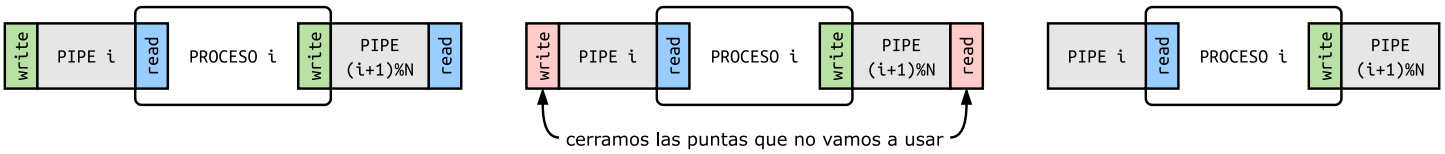
Antes de saltar directo al código conviene repasar el enunciado, dar algunos ejemplos y asegurarse que se entendió **qué** hay que hacer, sin necesariamente saber **cómo** hacerlo.

Por cuestiones de espacio, la resolución omite manejo de errores, includes y otros detalles menores de implementación. No obstante, en la clase mostraría el código completo y compilable.

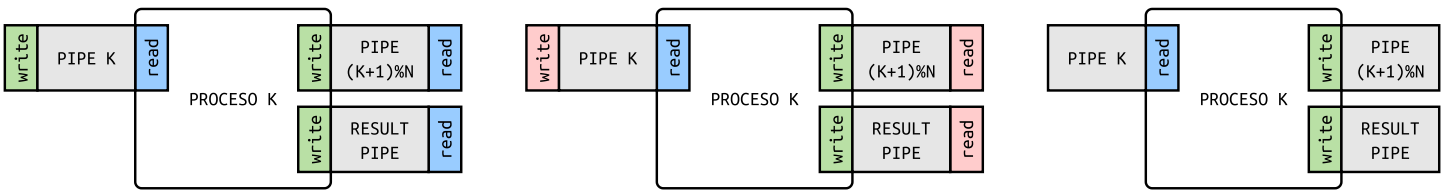
Crear procesos y pipes

El enunciado nos pide crear un anillo de N procesos. Si cada proceso se comunica con su antecesor y sucesor, necesitamos crear exactamente N pipes. Para cada proceso i, podemos utilizar el pipe i para comunicarnos con el antecesor, y el pipe $(i+1)\,\%\,N$ para comunicarnos con el sucesor. Notemos el uso de la aritmética módulo N, esto es lo que genera un anillo.

Los pipes son unidireccionales, y tienen dos puntas: la de lectura y la de escritura. El proceso i solo tiene que leer de su antecesor (solo necesitamos la punta de lectura del pipe i), y solo tiene que escribirle a su sucesor (solo necesitamos la punta de escritura del pipe $(i+1)\,\%\,N$). Las otras dos puntas las podemos cerrar, al igual que todos los otros pipes (lectura y escritura). Recordemos que al crear un proceso con fork, el hijo hereda todos los file descriptors abiertos. Por eso necesitamos cerrar todos estos file descriptors en cada hijo. En el siguiente gráfico podemos ver los únicos file descriptors que deben quedar abiertos en el proceso $i \neq K$.



Además de los pipes utilizados para el anillo, necesitamos un pipe adicional para la comunicación entre el hijo especial K y el padre. A través de este pipe, el hijo K le manda al padre el último número recibido. ¿Podríamos reutilizar el pipe $(k+1)\,\%\,N$? No, porque podemos tener un race condition entre el padre y el proceso sucesor de K. Podría pasar que el sucesor de K consuma ese dato antes que el padre, y necesitamos asegurarnos que este número le llegue al padre si o si.



A continuación presentamos la primer parte de `main()` que se encarga de crear los pipes y procesos hijo.

En una clase en vivo escribiría el código en tiempo real con el feedback contínuo de las y los alumnos. Presentar el código ya escrito, por más preámbulo que uno haga, no me parece que sea la mejor estrategia pedagógica. Escribir el código en vivo genera un ritmo más "lento". Esto permite que podamos pensar y procesar la solución a un ritmo más natural. También es una buena oportunidad para que puedan observar mis errores mientras suceden orgánicamente, con su respectivo debuggeo y resolución.

```c
int main(int argc, char **argv) {
    // Creamos los N pipes.
    int pipes[N][2];
    for (int i = 0; i < N; i++) pipe(pipes[i]);

    // Creamos el pipe utilizado por el hijo K para informar el número final al padre.
    int result_pipe[2];
    pipe(result_pipe);

    // Creamos los N hijos.
    for (int i = 0; i < N; i++) {
        if (fork() == 0) {
            // Cerramos todas las puntas de los pipes del anillo que no vamos a usar.
            for (int j = 0; j < N; j++) {
                if (j != i) close(pipes[j][READ]);
                if (j != (i+1) % N) close(pipes[j][WRITE]);
            }

            // Ningún hijo necesita leer del pipe de resultado.
            close(result_pipe[READ]);
            // Solo el hijo K necesita escribir en el pipe de resultado.
            if (i != K) close(result_pipe[WRITE]);

            // Llamamos a la función que ejecuta las acciones del hijo (worker).
            // Le pasamos únicamente los file descriptors que necesita (y que no cerramos).
            worker(i, pipes[i][READ], pipes[(i+1) % N][WRITE], result_pipe[WRITE]);
        }
    }
}
```

Hijos

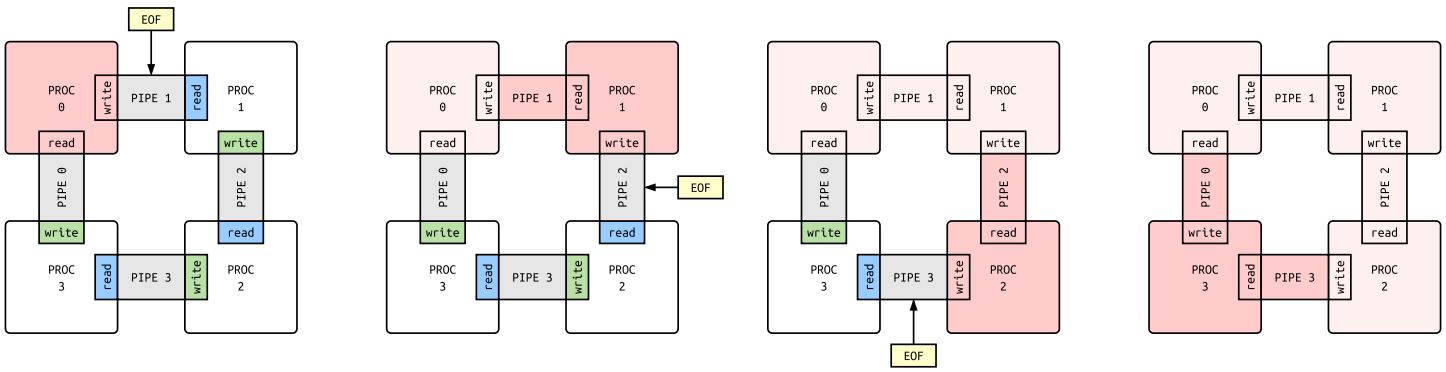
Más allá de buenas prácticas, cerrar inteligentemente las puntas (file descriptors) de los pipes que no vamos a usar nos permite utilizar el mecanismo de EOF (end-of-file) provisto por el SO para detectar cuándo tiene que terminar un proceso hijo dentro del anillo. La syscall `read()` bloquea hasta poder leer la cantidad de bytes estipulados en el llamado, o hasta detectar el EOF. La función nos retorna la cantidad de bytes leídos, y en el caso de EOF, retorna 0. El SO mantiene un contador de la cantidad de file descriptors abiertos a la punta de escritura de un pipe. Cuando ya no hay ninguno abierto, intentar leer de este pipe nos devuelve EOF, pues si nadie tiene un file descriptor para escribir, en efecto llegamos "al final" y no vamos a poder leer nada más.

A veces las y los alumnos no terminan de entender muy bien cuándo y por qué cerrar los file descriptors de los pipes, y por eso recaen en cerrar todo lo que no vayan a usar apostando a que las cosas simplemente funcionen. Considero importante entender a fondo este concepto para poder usarlo como una herramienta más a nuestro favor, y no fomentar una práctica de programación azarosa donde se escribe código sin saber bien por qué.

Esto nos permite utilizar un esquema en donde hacemos `while(read() > 0)` para procesar una cantidad arbitraria de datos hasta llegar al EOF. Observemos que esta estrategia funciona muy bien con este problema, ya que no sabemos cuántas vueltas va a girar el número por el anillo (es decir, no sabemos cuántos datos tiene que leer cada hijo de antemano).

Cuando el hijo especial K detecta la condición de salida, sale forzosamente del `while(read() > 0)` con un `break` y termina la ejecución del proceso con un `exit()`. El SO automáticamente libera todos los recursos de un proceso cuando este termina, en particular, cierra el file descriptor de la punta de escritura del pipe que comunica al hijo K con su sucesor. Esto dispara un efecto en cascada en donde todos los procesos en el anillo terminan su ejecución. El sucesor de K sale del `while(read() > 0)` porque `read()` retornó 0 (EOF). Igual que antes, el SO libera los recursos de este proceso generando que el "sucesor del sucesor" de K también salga del `while(read() > 0)`, y así vamos terminando secuencialmente todos los procesos del anillo, a partir del hijo K hasta dar toda la vuelta.

En el siguiente gráfico vemos un ejemplo con $N = 4$ y $K = 0$. Supongamos que el hijo K ya detectó la condición de salida y terminó su ejecución (por eso está pintado de rojo).



A continuación presentamos el código de los hijos.

Usamos una única función para todos los hijos, pero es muy probable que durante la clase alguien proponga tener dos funciones separadas. Es una solución igual de válida. Dado que el hijo K no es radicalmente distinto al resto, me pareció oportuno plantear una única función para reutilizar la lógica en común, y para reforzar la idea de que si bien hay un hijo especial, es también "uno más" del anillo. Si tenemos que modificar el comportamiento del anillo, ahora solo tenemos que modificar el código en un solo lugar.

```c
void worker(int i, int fd_read, int fd_write, int fd_result) {
    int number, secret;

    if (i == K) {
        // Si somos el hijo especial generamos el número secreto.
        secret = rand() % 42;
        printf("[worker %d] Número secreto: %d\n", i, secret);
    }

    // Leemos del pipe i (antecesor) hasta llegar al EOF.
    while(read(fd_read, &number, sizeof(number)) > 0) {
        if (i == K && number > secret) {
            // Llegamos a la condición de salida. Iniciamos la terminación en cascada de los procesos del anillo.
            printf("[worker %d] Recibí: %d, superamos mi número secreto\n", i, number);
            write(fd_result, &number, sizeof(number));
            break;
        } else {
            // Somos un hijo normal o no llegamos aún a la condición de salida.
            // Le pasamos el próximo número a nuestro sucesor a través del pipe (i+1) % N.
            number++;
            printf("[worker %d] Enviando número: %d\n", i, number);
            write(fd_write, &number, sizeof(number));
        }
    }

    // Muy importante hacer un exit() acá para terminar el proceso. Si no volvemos a main() y van a pasar cosas raras.
    exit(EXIT_SUCCESS);
}
```

Padre

Construimos un anillo de procesos donde inicialmente todos están bloqueados esperando leer de su antecesor. ¿Necesitamos un pipe adicional para enviar el número inicial al hijo especial K y disparar la reacción en cadena? No, podemos hacer algo más interesante. El padre todavía tiene abiertos todos los file descriptors de todos los pipes del anillo, y en particular, puede escribir en el pipe donde el hijo especial K está intentando leer. Podemos escribir ahí el número inicial desde el padre, y el hijo especial K va a empezar la reacción en cadena dentro del anillo. Nunca va a saber que este número lo mandó el padre, pero está bien eso, no necesita saberlo.

Siempre que podamos minimizar los "casos especiales" es bueno porque terminamos con una solución más simple y más fácil de adaptar a nuevos requerimientos. Pero hay que buscar un balance, no es bueno hacer cosas demasiado astutas o apoyándose en detalles de implementación externos muy específicos ya que a la larga esto resulta más difícil de mantener, o se introducen errores difíciles de detectar.

Luego de enviar el número inicial, el padre **debe** cerrar todos los file descriptors de escritura de los pipes del anillo. Esto es fundamental para que el SO pueda detectar y mandar los EOF en su debido momento. Por buena práctica también cerramos los de lectura, pero no sería mandatorio para que nuestra solución funcione correctamente.

Finalmente solo resta esperar a que nos avisen a través del pipe de resultado el número máximo alcanzado dentro del anillo. El llamado a `read()` bloquea hasta que haya un número en el pipe para leer. El hijo especial K es el único que va a escribir ahí cuando se cumpla la condición de salida. Si bien ya sabemos que ahí terminó el trabajo de los hijos, no sabemos si los procesos hijo efectivamente ya terminaron. Para ser ordenados, esperamos que todos los hijos terminen haciendo N veces `wait()` y así evitar dejar procesos zombies.

```c
int main(int argc, char **argv) {
    // Primera parte de main() donde creamos los pipes y procesos hijo.

    // Enviamos el número inicial al hijo especial K.
    int start_number = rand() % 10;
    printf("Empezando con el número: %d\n", start_number);
    write(pipes[K][WRITE], &start_number, sizeof(start_number));

    // Cerramos todos los pipes del anillo.
    for (int i = 0; i < N; i++) {
        close(pipes[i][READ]);
        close(pipes[i][WRITE]);
    }

    // Bloqueamos esperando al número final.
    int end_number;
    read(result_pipe[READ], &end_number, sizeof(end_number));

    // Esperamos que terminen todos los procesos hijo y evitamos dejar zombies.
    for (int i = 0; i < N; i++) wait(NULL);

    printf("Terminamos con el número: %d\n", end_number);
    exit(EXIT_SUCCESS);
}
```

Versión alternativa

Existe una version alternativa donde en vez de crear todos los pipes primero y luego cerrar los que correspondan en cada hijo, vamos creando los pipes a medida que creamos los hijos. De esta forma evitamos heredar en los hijos un montón de pipes que no vamos a usar. Pero esta solución es más compleja de explicar, y a efectos didácticos considero que no aporta un mejor entendimiento de los temas centrales del ejercicio. No obstante, quizás lo mencionaría en la clase para las y los curiosos.