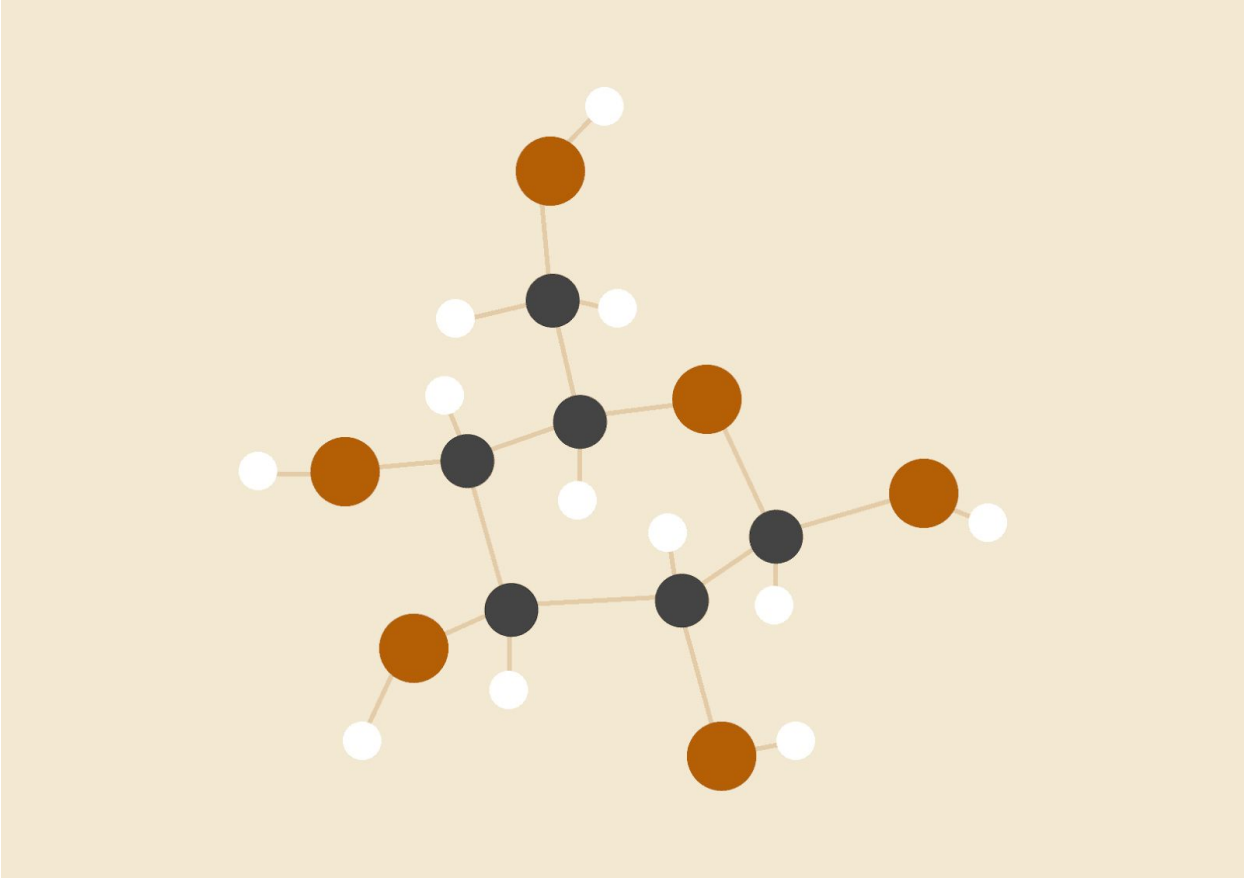
SISTEMAS OPERATIVOS PROYECTO

Práctica Final de Sistemas Operativos



Kevin de la Coba y José Manuel Freire

05/05/2020

Ingeniería Informática, UAM

INTRODUCCIÓN

El objetivo de este proyecto es implementar un sistema de ordenación multiproceso, al que tenemos que llegar modificando el sistema de ordenación monoproceso que se nos otorga al empezar la práctica. El sistema de ordenación estará dividido en varias tareas independientes de sort.c y utils.c.

Hemos sido capaces de implementar todos los requisitos exceptuando el f) (a medias), ya que los trabajadores no son capaces de enviarse la señal SIGALRM de manera autónoma.

DESARROLLO

Para el desarrollo del programa, hemos optado por seguir los pasos recomendados al final del documento con el enunciado.

En el primer apartado tuvimos un problema con la función ftruncate, ya que nos ponía implicit declaration, y nos llevó bastante tiempo darnos cuenta que el problema era que no se podía usar esa función con -ansi al compilar.

Durante el segundo y tercer paso, los cuales hicimos a la vez, nos costó bastante mantener a los procesos coordinados, ya que algunos empezaban a hacer tareas del siguiente nivel cuando los del nivel anterior no habían terminado las suyas. Para solucionar esto añadimos un semáforo y una variable compartida que actuaba como contador (esto está mejor explicado en el apartado de soluciones).

En el cuarto apartado, nos llevó bastante tiempo darnos cuenta de que en el programa se producía un bloqueo cuando enviamos los mensajes con la cola de mensajes antes de crear los procesos, ya que no se recibían, y por tanto la cola no se vaciaba y se atascaba. La solución era simplemente crear primero los procesos y después enviar las tareas.

En el quinto apartado, durante la implementación de las máscaras de señales, para que cada proceso sólo fuese interrumpido por las que se pedía, tuvimos que reescribir gran parte del código que habíamos hecho hasta el momento, ya que los hijos se creaban en el mismo loop que el padre. En este caso teníamos que crear dos loops, para que el padre mandase su mensaje en uno y los hijos en otro. Además, con algunos errores con las máscaras de señales, llegamos a crear algún proceso que nos llevó un buen rato matar.

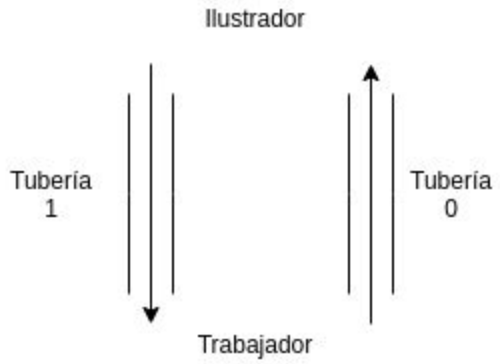
La sexta parte no nos fue demasiado complicada, solo hubo que crear otra máscara para que el proceso principal manejase SIGINT, cuando esta señal se recibe el manejador envía SIGTERM a todos los procesos hijos, liberando los recursos, y terminando su ejecución.

El problema se encuentra en la última parte, en esta última parte tuvimos demasiados problemas con las tuberías. Lo que hacíamos es crear 2 tuberías por proceso, para que cada proceso se comunicará con el ilustrador por una tubería. Si hacíamos esto el programa solo funcionaba bajo ciertas condiciones (10 niveles y 10 procesos). Lo que hicimos para resolver todos estos problemas es que en vez de hacer 2 tuberías por proceso, hacemos 2 tuberías, por una escriben los trabajadores y por otra el ilustrador.

SOLUCIONES

Para resolver ciertos problemas que surgieron durante el desarrollo hicimos lo siguiente. Creamos una variable compartida llamada contador, esto lo hicimos para saber cuando tiene que mandar tareas de otro nivel el proceso principal. Por ejemplo, si en el nivel ‘x’ tenemos 8 tareas, el proceso principal antes de mandar estas tareas cambia el valor de la variable contador a 8, los trabajadores cada vez que terminan una tarea decrementan el valor de esta variable en uno, todo esto obviamente está controlado con un semáforo. El proceso principal cuando termine de mandar tareas del nivel ‘x’ tratara de pasar al nivel ‘y’ pero no podrá ya que otro semáforo le impedirá el paso, el que deja pasar a el proceso principal es el trabajador que realice la última tarea (contador == 1) en ese momento hará post del semáforo que bloquea al padre y este empezara a mandar tareas del nivel ‘y’. Esto lo implementamos antes de crear la máscara para SIGUSR1, se podría hacer algo parecido usando esa máscara pero decidimos no usarla y usar esta solución.

Para solucionar el problema que tuvimos con las tuberías, lo que hicimos fue hacer dos tuberías. En la tubería 0 el trabajador escribe su estado, y el ilustrador lee el estado del trabajador, por lo tanto el trabajador cierra en esta tubería el extremo 0 ya que no lo usa y el ilustrador cierra el extremo 1. En la tubería 1 el ilustrador es el que escribe, y escribe tantas veces como trabajadores hay en el sistema, el ilustrador cierra el extremo 0 ya que no lo usa y el trabajador cierra el extremo 1.



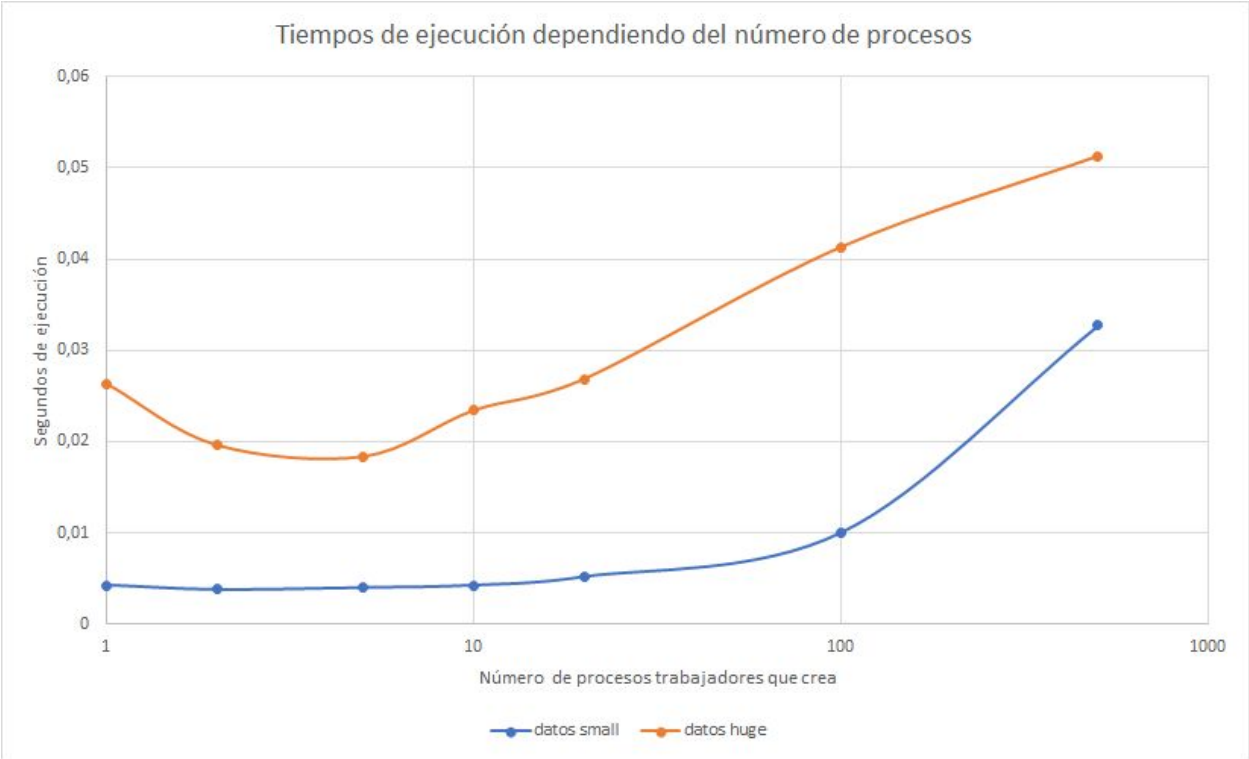
Este es un pequeño esquema en el que se puede ver como usan los distintos procesos las tuberías.

RESULTADOS

El algoritmo ordena sin problema alguno, abajo tenemos una tabla con los tiempos en los que ha ordenado los diferentes arrays ya incluidos en el código original. Cabe destacar que este código es sin que funcionen las tuberías.

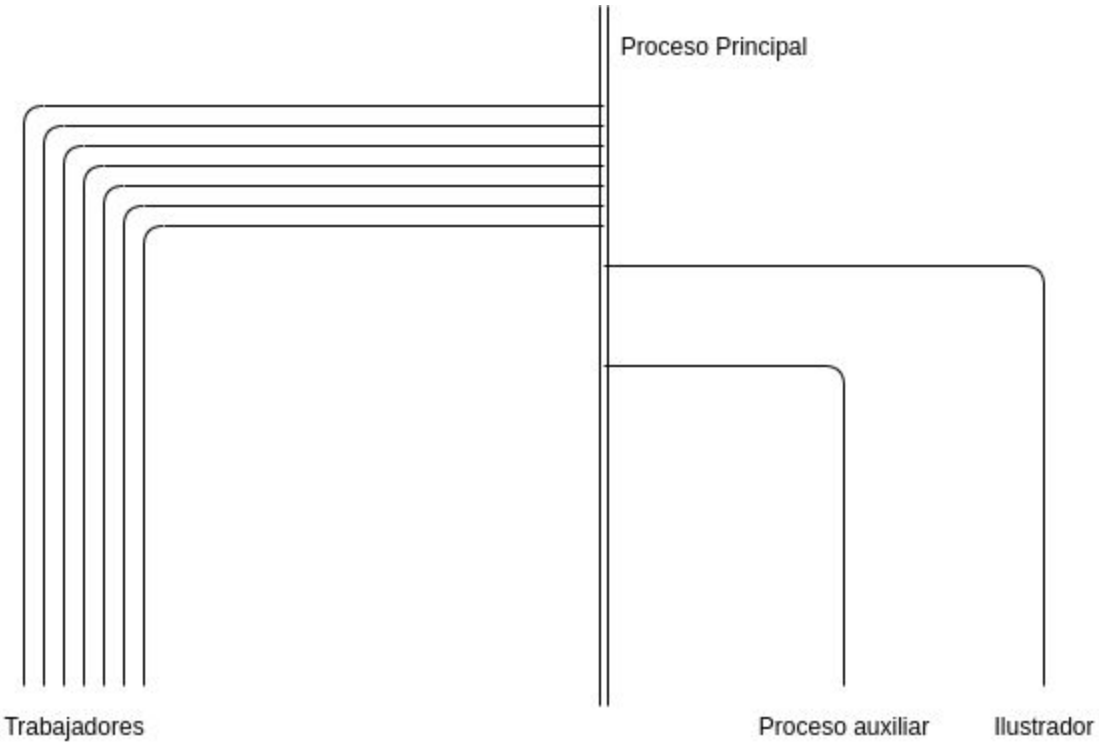
DATOS

n_processes	Tiempo de ejecución (small)	Tiempo de ejecución (huge)
1	0.004267	0.026375
2	0.003776	0.019607
5	0.003977	0.018331
10	0.004238	0.023402
20	0.005176	0.026860
100	0.010025	0.041323
500	0.032729	0.051281

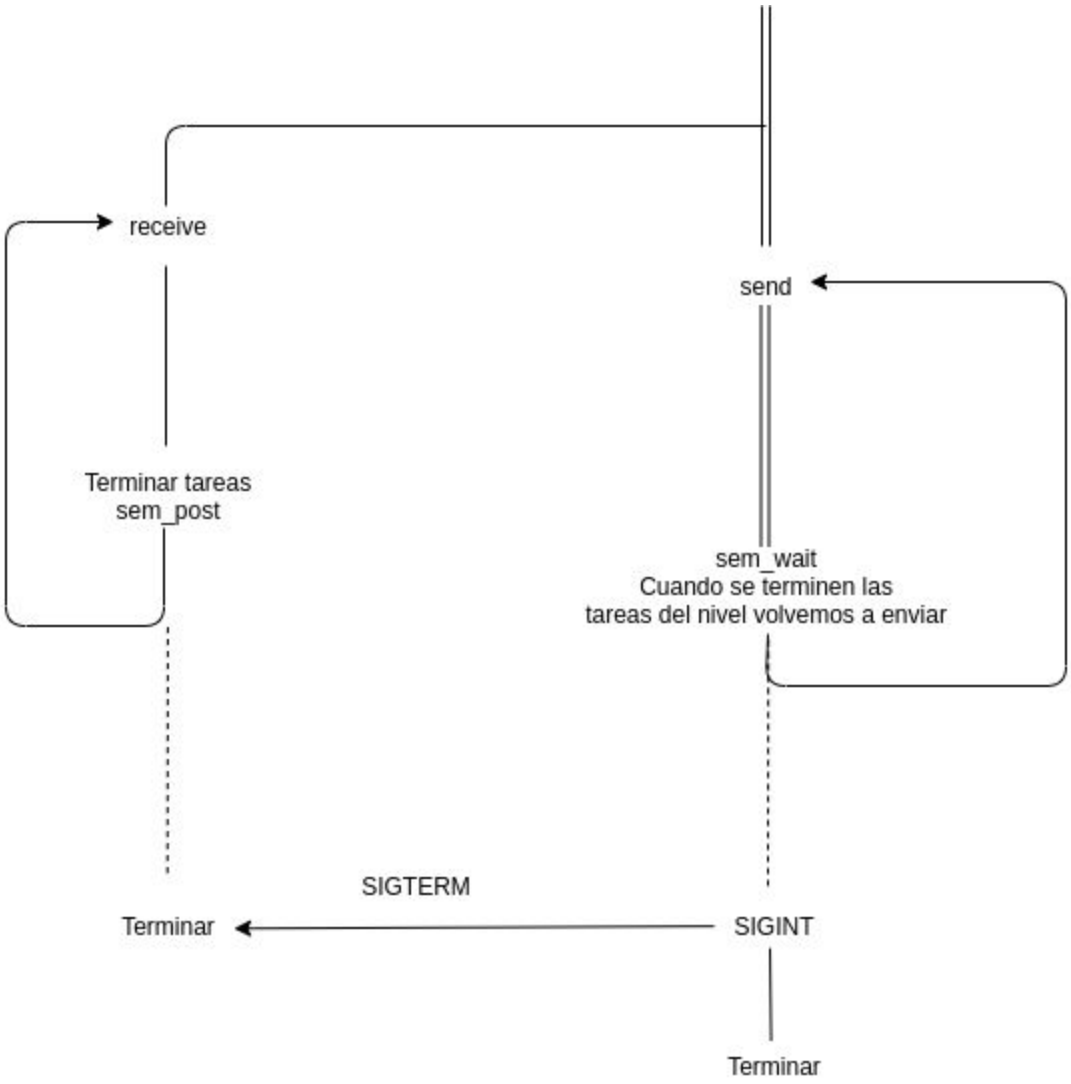


Podemos observar en la gráfica cómo el número de procesos optimiza la velocidad de ejecución a medida que aumenta hasta 2 en el caso de la tabla de datos pequeña y hasta 5 en la tabla más grande. A partir de estos puntos, cuantos más procesos, más ineficiente se vuelve. Creemos que esto se debe a el número de operaciones necesarias para gestionar cada uno de los procesos, requiriendo estos muchas comprobaciones en cuanto a control de errores y loops, que hacen que cuantos más procesos haya, más tiempo tarden, pero sin aportar estos una gran optimización al funcionamiento del programa, ya que muchos de ellos no van a estar en ejecución la mayoría del tiempo.

DIAGRAMA

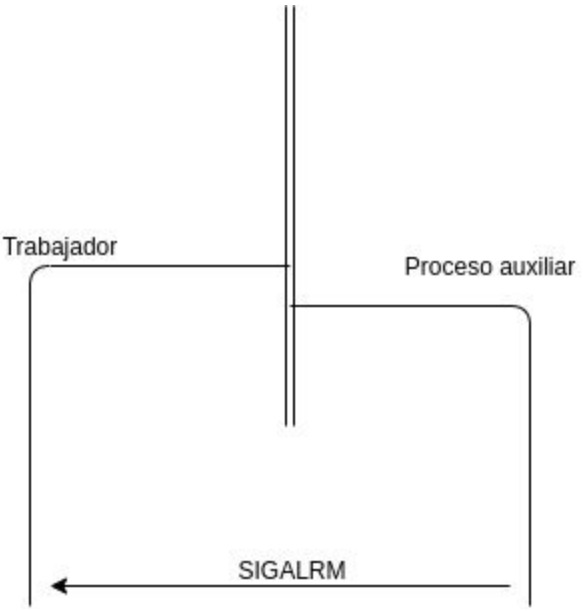


Esta es una imagen de cómo el proceso principal hace forks y crea los diferentes procesos.

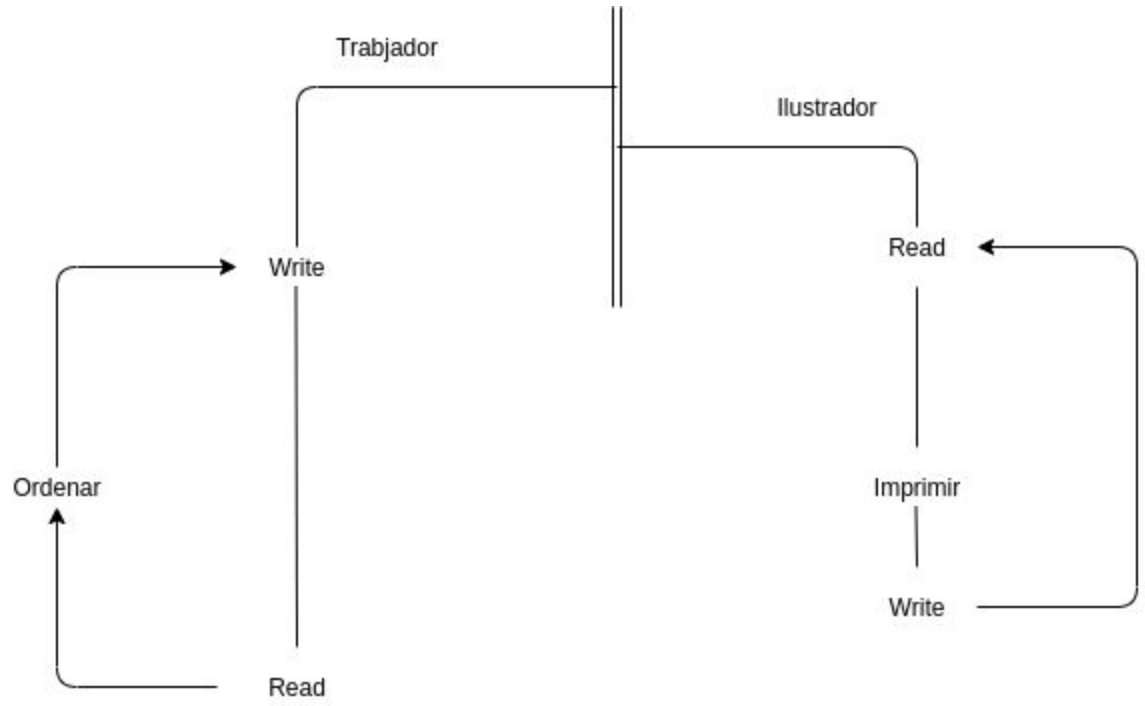


En esta segunda imagen podemos ver cómo se envían tareas desde el proceso principal hasta el trabajador. Cuando se crea el proceso trabajador este se bloquea hasta que recibe una tarea. El padre tras enviar las tareas de un nivel se bloquea en sem_wait, cuando el trabajador acabe todas las tareas del nivel hace sem_post y desbloquea al principal, dejando a este mandar las tareas del siguiente nivel. Si durante la ejecución se recibe una señal SIGINT el proceso principal manda SIGTERM a todos los trabajadores para liberar recursos.

Entre el ilustrador y el proceso principal no hay comunicación.



En esta imagen podemos ver como el proceso auxiliar es el que manda la señal SIGALRM al trabajador (esto es periódico, cada segundo).



En esta última imagen podemos ver cómo se comunican el ilustrador con los trabajadores. El ilustrador se bloquea haciendo read (hace read tantas veces como procesos hay en el sistema), una vez todos los trabajadores han hecho write el trabajador

imprime. Los trabajadores se bloquean en read tras hacer write, hasta que el ilustrador no haga write los trabajadores no se desbloquearan.

CONCLUSIÓN

Podemos concluir que en este proyecto hemos usado todos los conocimientos aprendidos durante el curso, desde la creación de procesos (fork), hasta la creación de colas de mensajes. Donde más problemas hemos tenido ha sido en el uso de las tuberías, pero una vez cambiamos el diseño para tener 2 tuberías, todo funcionó perfectamente.