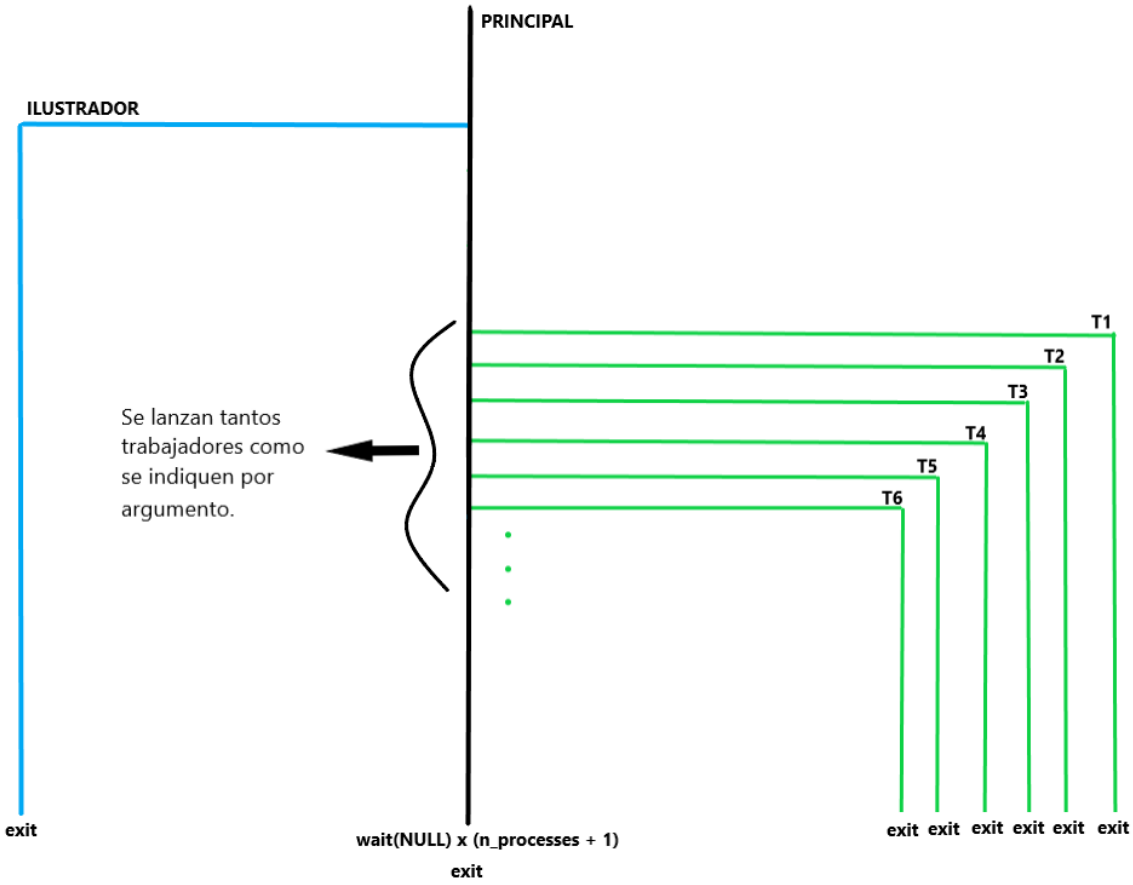
SISTEMAS OPERATIVOS
PROYECTO

Entregable 1: Memoria

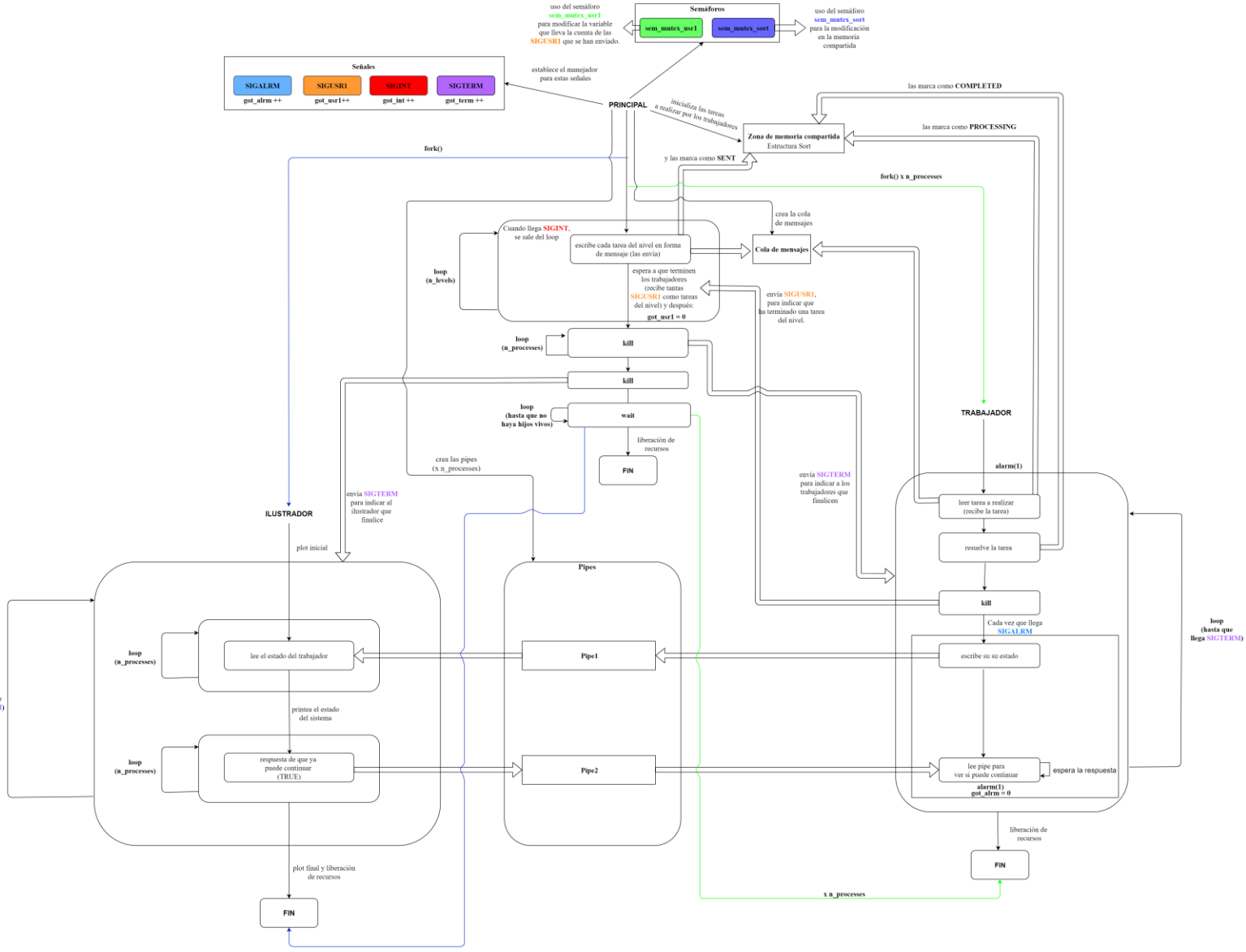
a) Primeramente, se va a realizar un esquema mostrando la jerarquía de los procesos involucrados en el proyecto.



En el anterior esquema, podemos observar el programa principal, que lanza tantos procesos trabajadores (Ti) como se le indiquen por argumento, n_processes, que realizarán las tareas, que se especificarán en el diagrama de más abajo. Además, también lanza el proceso ilustrador, que se encargará de mostrar por pantalla el estado del sistema cada segundo. Todos estos procesos, tanto el principal como todos sus hijos, se van a comunicar para llevar a cabo el propósito del sistema de manera ordenada y síncrona, tal y como se pide en el enunciado.

A continuación, se mostrará un diagrama general del sistema para tener una representación de cómo los procesos, con mecanismos de

comunicación y sincronización, llegan a realizar sus tareas. Para verlo más en detalle mirar el fichero diagrama.png.



Vamos a explicar brevemente el diagrama para hacerlo más comprensible.

En primer lugar, vamos a comentar los elementos del mismo. Se ha identificado las señales con colores, para que sea más fácil visualizarlas en el diagrama, así como los semáforos. Las fechas finas, si son negras indican el hilo de los procesos; si son azules, indican la relación entre el proceso principal y el ilustrador; y finalmente, si son verdes, indican la relación entre el proceso principal y los trabajadores. Las flechas gruesas indican flujo de datos o señales, y la punta de las mismas indica el sentido de ese flujo. Por ejemplo, enviar una señal o escribir en el pipe/cola/memoria

compartida… Los cuadros pequeños indican acciones que hacen los procesos. Por ejemplo, escribir en el pipe o enviar una señal. Cabe destacar que la creación de estructuras como la memoria compartida, colas de mensajes, y demás, así como establecer el manejador, no se ha indicado con estos cuadros, simplemente se ha indicado con una flecha fina y un comentario. Los cuadros grandes indican grupos, por ejemplo, un conjunto de señales o de pipes.

Se puede observar que en algunos cuadros hay loop's, esta palabra engloba tanto while como for. Si se trata de un while, aparecerá la palabra "hasta", para indicar que hasta que no ocurra tal, el loop continúa, en caso contrario, se trata de un for. Otras palabras que se pueden encontrar en el diagrama son kill, que indica que se envía una señal (en la punta de la flecha se indica qué señal es, y para qué se envía), y wait, que indica que se espera a algún hijo. A quién espera el padre lo indican las flechas que salen del cuadro de esta acción. Por último, la palabra n_processes indica que se realiza o se crea n_processes veces.

Por último, se va a comentar un aspecto que puede llevar a confusión. En el proceso trabajador, hay un cuadrado dentro del loop del proceso (se pueden distinguir por la forma), este cuadrado, como se indica justo encima del mismo, se ejecuta cada vez que el proceso recibe SIGALRM.

b) En este apartado se va a explicar los detalles del diseño, así como los hilos de ejecución.

Se lanza el programa principal y llama a la función sort_multy_process, que es el hilo principal del programa y el que creará los trabajadores y el ilustrador. Primero, creará todas las estructuras necesarias: arma el manejador de las cuatro señales del sistema (SIGURS1, SIGALRM, SIGTERM, SIGINT); crea la memoria compartida para la estructura Sort y lo mapea con mmap; inicializa la estructura Sort llamando a init_sort; crea la cola de mensajes bloqueante; crea los semáforos mutex_sort, que asegura la exclusión mutua a la estructura Sort, y mutex_usr1, que protege la variable got_usr1; y crea las pipes. En cuanto a las pipes, hay dos arrays de pipes, uno, donde los pipes tienen dirección trabajador-ilustrador (fd_work2ilu), y otro, que tiene dirección ilustrador-trabajador (fd_ilu2work). El primer array funciona de la siguiente manera, fd_work2ilu [2*i+1] será donde escriba el trabajador i con destino al proceso ilustrador, y fd_work2ilu [2*i] será donde el ilustrador lea al trabajador i. Y respecto al segundo array, los roles del trabajador i y el ilustrador están cambiados.

Ahora el principal lanza el proceso ilustrador, que inicialmente imprime los datos iniciales. Mientras no reciba la señal SIGTERM, leerá del pipe el estado de cada trabajador, imprimirá el vector y el estado de los trabajadores, y escribirá por el pipe a los trabajadores para que continúen. Cuando no reciba la señal de terminar, imprimirá el vector ordenado, y acaba liberando recursos.

Después lanza n_processes trabajadores, que inicialmente abren las estructuras que necesitan para realizar las tareas y comunicarse con el padre y el ilustrador. Cada trabajador establece una alarma de un segundo. Cada vez que se reciba la señal SIGALRM, en el propio manejador, escribirá su estado en el pipe para informar al ilustrador, y esperará su respuesta. Cuando la reciba, establecerá otra vez la alarma de un segundo. Después de establecer la primera alarma, mientras no reciba la señal SIGTERM, leerá su tarea de la cola de mensajes. Cuando reciba la tarea, la resolverá llamando a la función solve_task. Una vez resuelta, enviará la señal SIGUSR1 al padre.
Durante su ejecución, irá actualizando el estado de las tareas que reciba, y su propio estado.

Ahora, el proceso principal, por cada nivel, envía todas las partes del nivel (tareas) a la cola de mensajes, para que los trabajadores puedan recibirlas, y espera a recibir las señales SIGUSR1 de todas las partes del nivel. Una vez que acaba con todos los niveles, envía la señal SIGTERM a todos los trabajadores y al ilustrador para que terminen. Espera a que acaben con un wait, libera los recursos y acaba.

Si mientras el proceso principal está enviando tareas, recibe la señal SIGINT, dejará de enviarlas, y pasará a enviar la señal SIGTERM a sus hijos.

En cuanto a las dificultades que hemos tenido:

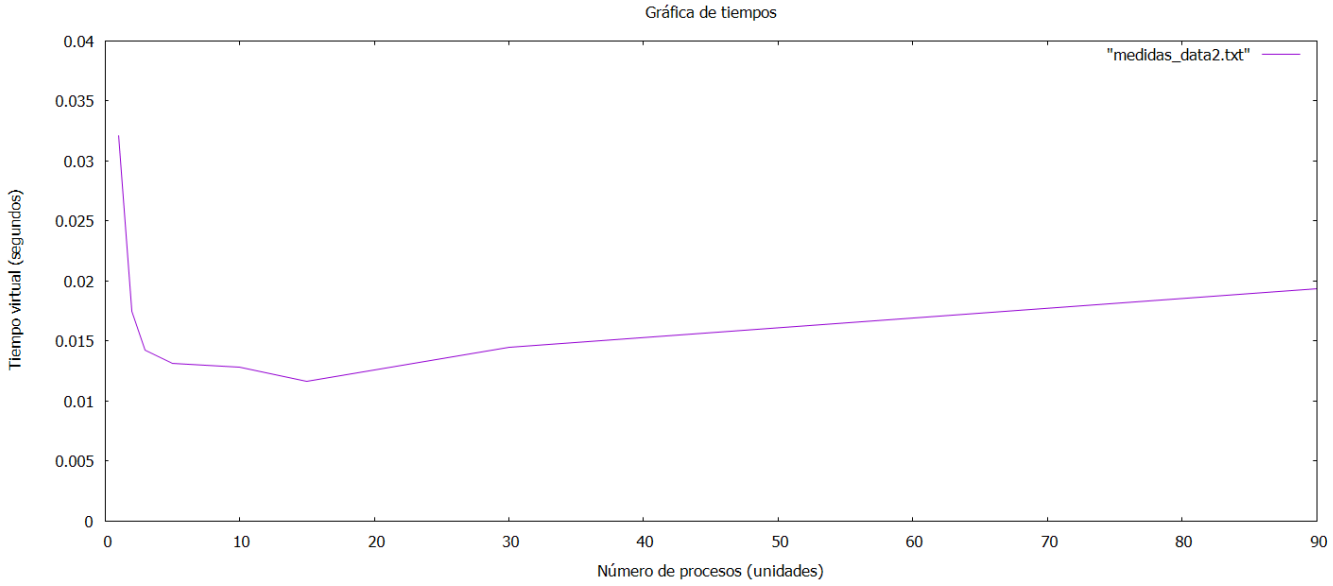
- Para evitar que las señales interrumpan llamadas bloqueantes (sem_wait, mq_receive, mq_send), hemos puesto un condicional en el que, si mientras se está bloqueado por la llamada, se recibe una señal, entonces se vuelve a ejecutar la llamada bloqueante, ya que esta no se ha podido ejecutar.
- Para evitar repetir código de error, se ha creado la función free_smp, que libera todos los recursos.
- Hemos añadido funcionalidad al manejador (con ayuda de variables globales), para evitar repetir código, ya que no se sabe cuándo va a llegar una señal a un proceso.

- Tuvimos un problema con condiciones de carrera, ya que no protegíamos la variable global got_usr1, y se podían machacar los valores.

c) Para medir los tiempos de ejecución y compararlos, mediremos con la función clock dos veces, una antes de llamar a sort_multy_process (t1) y otra justo después (t2). El tiempo que se tarda en ordenar (t2-t1)/CLOCKS_PER_SEC.

Se van a guardar en un fichero los datos del número de procesos usados y el tiempo empleado. Se va a graficar con GNUplot y se va a comparar con una función.

Un ejemplo es el siguiente, donde el vector a ordenar era el DataLarge.dat:



Se puede observar que cuando hay muy pocos procesos, tarda mucho, y que a medida que se van añadiendo procesos tarda menos. Pero hay un punto en el que meter más procesos hace que tarde más, porque cada vez hay más procesos que están esperando en la cola (usando recursos), debido a que todas las tareas del nivel están asignadas.

El tiempo virtual es el tiempo que está en ejecución, que no el real.

Entregable 2: Sistema multiproceso

a) El funcionamiento general del proceso principal se basa en crear todas las estructuras de comunicación y sincronización, lanzar los procesos hijos (trabajadores e ilustrador), enviar las tareas a los trabajadores, así como enviar a los hijos la señal SIGTERM.

b) El funcionamiento general de un proceso trabajador se basa en, inicialmente, abrir las estructuras que necesita para realizar las tareas y comunicarse con el padre y el ilustrador. Además, establece una alarma de un segundo, y cada vez que se reciba la señal SIGALRM, en el propio manejador, escribirá su estado en el pipe para informar al ilustrador, y esperará su respuesta. Cuando la reciba, establecerá otra vez la alarma de un segundo. Después de establecer la primera alarma, mientras no reciba la señal SIGTERM, leerá su tarea de la cola de mensajes. Cuando reciba la tarea, la resolverá llamando a la función solve_task. Una vez resuelta, enviará la señal SIGUSR1 al padre.
Durante su ejecución, irá actualizando el estado de las tareas que reciba, y su propio estado.

c) El funcionamiento general del ilustrador se basa en, inicialmente, imprime los datos iniciales. Mientras no reciba la señal SIGTERM, leerá del pipe el estado de cada trabajador, imprimirá el vector y el estado de los trabajadores, y escribirá por el pipe a los trabajadores para que continúen. Cuando no reciba la señal de terminar, imprimirá el vector ordenado, y acaba liberando recursos.

d) Todos los procesos hijos mapean la memoria compartida para poder acceder, con un semáforo mutex, que asegura la exclusión mutua, a la estructura Sort.

e) Hay dos arrays de pipes, uno, donde los pipes tienen dirección trabajador-ilustrador (fd_work2ilu), y otro, que tiene dirección ilustrador-trabajador (fd_ilu2work). El primer array funciona de la siguiente manera, fd_work2ilu [2*i+1] será donde escriba el trabajador i con destino al proceso ilustrador, y fd_work2ilu [2*i] será donde el ilustrador lea al trabajador i. Y respecto al segundo array, los roles del trabajador i y el ilustrador están cambiados.

f) Cada vez que se recibe una señal, su correspondiente variable global, se incrementa en una unidad.

Cada vez que se recibe la señal SIGALRM, en el propio manejador, escribirá su estado en el pipe para informar al ilustrador, y esperará su respuesta. Cuando la reciba, establecerá otra vez la alarma de un segundo.

Cada vez que se recibe SIGTERM, el proceso, cuando vuelva del manejador, hará la rutina de liberar los recursos, y acabará.

Cada vez que se recibe SIGINT, el proceso principal, cuando vuelva del manejador, enviará la señal SIGTERM a los hijos, los esperará, y acabará. Los procesos hijo ignorarán SIGINT.

Cada vez que se recibe SIGUSR1, se hará un sem_post de mutex_usr1 (semáforo mutex) para indicar que acaba la zona protegida de la variable got_usr1. Esto implica que otro trabajador podrá salir de su sem_wait y enviar la señal SIGUSR1.

g) Hemos usado un semáforo mutex, concretamente mutex_sort. Léase d.

h) Hemos creado una cola de mensajes bloqueante para que el proceso principal envíe las tareas y los trabajadores las reciban.

En cuanto a los problemas con la cola bloqueante y las señales, se ha explicado anteriormente en el ejercicio Entregable 1, apartado b.

i) No se ha implementado el código, pero se ha pensado cómo se podría implementar.

Para que se puedan realizar tareas de un nivel superior al actual, después de recibir cada señal SIGUSR1, se mira con un bucle for si se puede añadir a la cola una tarea del nivel superior. En ese caso, se añadiría a la cola. También, antes de enviar cada tarea a la cola, se debe comprobar si la tarea está completada, porque ahora es posible que se haya completado en el nivel inferior.

De este modo solo se pueden realizar tareas de hasta un nivel superior, y si una tarea de ese nivel superior se completa, no es posible realizar una tarea del nivel superior a ese nivel hasta que el nivel actual no se haya completado entero.

Una forma de implementar este algoritmo de ordenación, de forma paralela para poder hacer las tareas de distintos niveles de manera concurrente sería, en vez de un diseño Bottom-Up, usar Top-Down. Esta implementación se basa en: el proceso principal divide el vector a ordenar en dos partes iguales; haría un fork () para que el hijo ordene la parte izquierda, otro fork () para que otro hijo ordene la parte derecha; y luego

dos wait () para esperar a que las dos partes estén ordenadas, y así proceder a unir las dos partes. Cada hijo actuará de forma recurrente como el padre, hasta que la longitud del vector que le toque ordenar sea menor al límite para hacer BubbleSort (n_levels) en el propio vector. Se puede limitar el número de procesos para que no se supere n_processes.