Proyecto Final

FECHA DE ENTREGA: EL 8 DE MAYO
(HORA LÍMITE: 23:59)

INTRODUCCIÓN AL PROYECTO.

El objetivo de este proyecto es implementar un **sistema de ordenación multiproceso**. Para ello, se dividirá el método global de ordenación en diversas tareas, algunas de ellas independientes entre sí, de manera que se puedan paralelizar para reducir el tiempo de ejecución. Aunque se construirá una versión muy simplificada, este tipo de arquitecturas se utilizan para abordar problemas con grandes volúmenes de datos en aplicaciones reales.

Para realizar este proyecto se partirá de un sistema de ordenación monoproceso ya implementado, que se irá modificando utilizando los mecanismos vistos anteriormente hasta llegar a la versión multiproceso.

Algoritmos de Ordenación

Aunque no es necesario conocer los detalles algorítmicos para realizar el proyecto, es conveniente tener unas ciertas nociones sobre métodos de ordenación. A continuación se realiza una pequeña descripción de los algoritmos que se van a utilizar, pese a que se dan implementados en el código de partida.

Ordenamiento por Mezcla

Uno de los métodos de ordenación más eficientes es el de Ordenamiento por Mezcla o *MergeSort*, que utiliza una aproximación de tipo "divide y vencerás".

En particular, en su versión más sencilla este método se implementa de forma recursiva, según los siguientes pasos:

1. Si la lista contiene 0 ó 1 elementos, ya está ordenada.
2. Si la listas contiene más de 1 elemento:
   a) Dividir la lista desordenada por la mitad.

b) Ordenar cada mitad recursivamente aplicando el ordenamiento por mezcla.
c) Mezclar las dos mitades ordenadas en una sola lista ordenada.

Además de su eficiencia, la principal ventaja de *MergeSort* es la facilidad con la que se puede paralelizar. Como se puede ver en los pasos anteriores, una vez que una lista se ha dividido en dos sublistas, éstas se ordenan de forma completamente independiente, por lo que ambas tareas se pueden asignar a distintos procesos.

Se puede ver un ejemplo de cómo funciona este algoritmo en este enlace.

Ordenamiento de Burbuja

Otro método clásico de ordenación es el Ordenamiento de Burbuja o *BubbleSort*.

Aunque no es de los más eficientes, este método es sencillo de implementar y muy visual. Se basa en comparar un elemento con el siguiente, intercambiando su posición si es necesario, lo que hace subir hacia arriba a los elementos más grandes como si fueran burbujas.

En este enlace se puede ver un ejemplo de este método.

Algoritmo para el Proyecto

Descripción

El sistema que se implementará utilizará una versión híbrida de los dos métodos anteriores. El algoritmo *MergeSort* original requiere que se divida la lista sucesivamente (en 2, 4, 8, 16...) hasta tener como mucho 1 elemento en cada sublista, por lo que el número de particiones que se van a hacer depende del número inicial de datos. Sin embargo, en el sistema propuesto, el número de particiones que se van a hacer (niveles del algoritmo) será especificado por el usuario, por lo que las sublistas más pequeñas podrán tener más de 1 elemento. Estas listas serán ordenadas utilizando el algoritmo *BubbleSort*.

En particular, el sistema utilizará N niveles, y por tanto se realizarán $N-1$ divisiones de los datos. En el Nivel 0 habrá 2^{N-1} tareas, consistentes en ordenar las 2^{N-1} sublistas en las que se ha dividido el conjunto de datos inicial utilizando *BubbleSort*. El Nivel 1 consistirá en mezclar dos a dos las 2^{N-1} sublistas ordenadas resultantes del Nivel 0 utilizando *MergeSort*, produciendo por tanto 2^{N-2} sublistas ordenadas. Este proceso se repetirá hasta llegar al último nivel, el Nivel $N-1$, en el que se mezclarán con *MergeSort* las 2 sublistas ordenadas resultantes del Nivel $N-2$, produciendo una única lista ordenada con todos los datos.

Es fácil ver que las tareas de cada nivel son independientes entre sí (se pueden realizar en cualquier orden, incluso simultáneamente, ya que afectan a datos distintos). Por otro lado, una cierta tarea del Nivel i solo depende de las dos tareas en el Nivel $i-1$ que utilizan sus mismos datos; en cuanto estas dos tareas hayan terminado ya podrá empezar.

Ejemplo

La Figura 1 muestra un ejemplo de este algoritmo, para una lista de 32 elementos y 4 niveles. En concreto, el algoritmo mixto funcionará de la siguiente manera:

Nivel 0 Se empezará partiendo los datos en 8 sublistas de 4 elementos, que serán ordenadas aplicando 8 veces el algoritmo *BubbleSort*.
Nivel 1 Este nivel estará formado por 4 tareas, consistentes en mezclar la sublista 1 con la 2,

la sublista 3 con la 4, la 5 con la 6, y la 7 con la 8, resultando en 4 sublistas ordenadas de 8 elementos.

Nivel 2 En este nivel se realizarán 2 mezclas de las sublistas de 8 elementos (la 1 con la 2, y la 3 con la 4), para producir 2 sublistas ordenadas de 16 elementos.

Nivel 3 Este último nivel consistirá en 1 tarea, mezclar las dos listas de 16 elementos para producir una única lista ordenada completa.

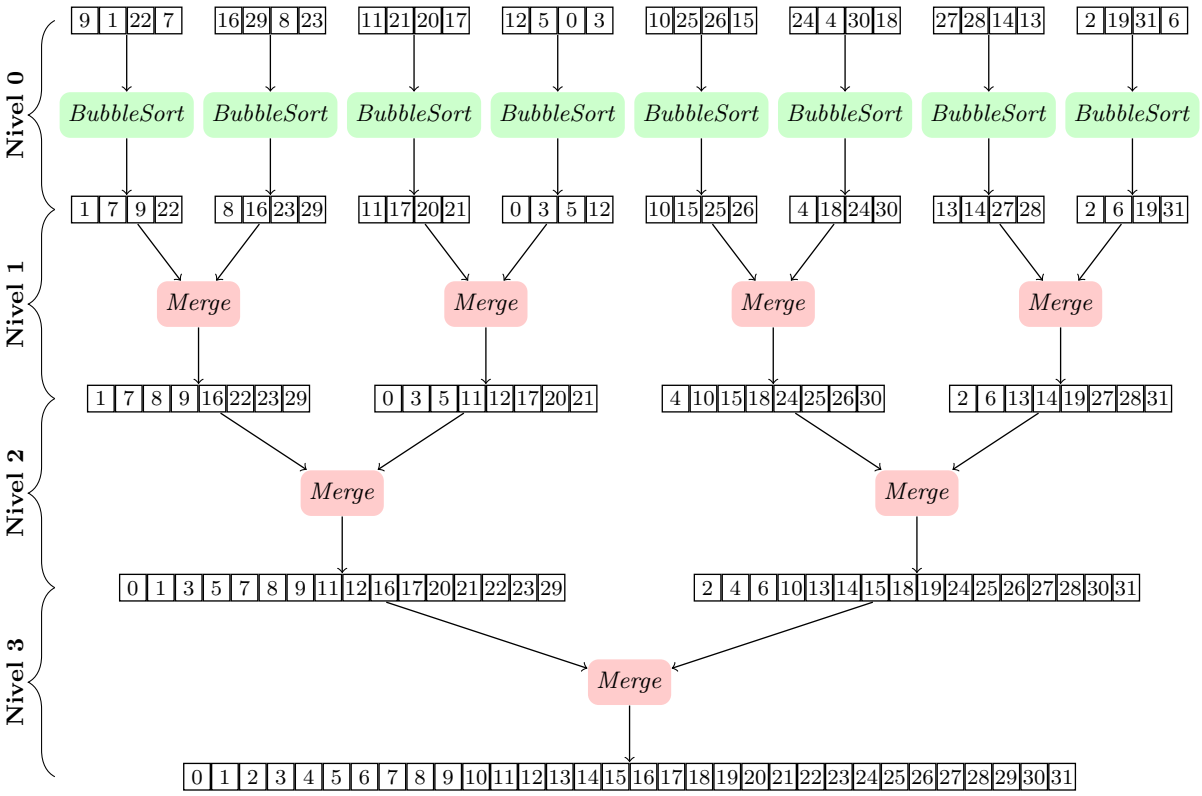

Figura 1: Ejemplo del algoritmo mixto de ordenación con 4 niveles y 32 elementos.

SISTEMA MONOPROCESO

El algoritmo descrito se puede implementar de manera muy sencilla en un sistema monoproceso. Para ello basta con iterarar sobre cada uno de los niveles, desde el 0 hasta el $N-1$, realizando secuencialmente cada una de las tareas de ese nivel.

Como punto de partida para el proyecto se da una implementación del sistema monoproceso, que está debidamente comentada en los ficheros de cabecera. Aunque a continuación se describen brevemente los módulos y principales funciones de este sistema, es importante ***leer y entender todo el código provisto*** antes de empezar con la implementación del proyecto.

Módulo `Global`

Este módulo consta solamente de un archivo de cabecera, con las definiciones de dos tipos de datos que se utilizan como retorno de las funciones:

- `Status`: para indicar si una función ha terminado correctamente o no.

- `Bool`: para representar un dato booleano.

Módulo `Utils`

Este módulo contiene funciones que proveen utilidades de carácter general:

- `compute_log`: calcula el logaritmo entero en base 2. Esta función valdrá para saber el número máximo de niveles que tiene sentido utilizar según el tamaño de los datos, para evitar tareas vacías.
- `clear_screen`: limpia la pantalla.
- `print_vector`: imprime un vector por pantalla en modo texto.
- `plot_vector`: pinta un vector por pantalla. Si es lo suficientemente pequeño utilizará una representación gráfica. Si es grande, lo imprimirá en modo texto.
- `fast_sleep`: pone a dormir un proceso durante un cierto número de nanosegundos.

Módulo `Sort`

Este es el principal módulo, que contiene la lógica del sistema de ordenación. Para ello, define los siguientes tipos de datos:

- `Completed`: Representa el estado de una tarea, que puede estar incompleta, enviada, en proceso o completada.
- `Task`: Contiene la información relativa a una tarea, incluyendo su estado, y los datos que abarca. En concreto, los datos de la tarea irán desde la posición `ini` hasta `end - 1`. El valor de `mid` indicará en qué consiste la tarea, de manera que si tiene un valor inválido, hay que ordenar todos los datos entre `ini` y `end - 1` usando *BubbleSort*. Si tiene un valor válido, quiere decir que hay dos sublistas ordenadas, la primera desde `ini` hasta `mid - 1`, y la segunda desde `mid` hasta `end - 1`, que hay mezclar usando *MergeSort*.
- `Sort`: Esta es la estructura principal, que contiene los datos del sistema de ordenación. Almacena la descripción de todas las tareas de todos los niveles (incluyendo los datos que abarcan y su estado), los datos que se van a ordenar (tanto el array como su longitud), el retardo que se introducirá en el proceso de ordenación (para que se pueda visualizar), el número de niveles del sistema, el número de procesos que van a realizar la ordenación, y el PID del proceso principal del sistema.

Además, este módulo incluye las siguientes funciones:

- `bubble_sort`: Esta función realiza la ordenación de un vector utilizando *BubbleSort*. Introduce además un cierto retardo en cada comparación.
- `merge`: Esta función realiza la combinación de las dos partes ordenadas de un array usando *MergeSort*. Introduce además un cierto retardo en cada comparación.
- `get_number_parts`: Esta función devuelve el número de partes (tareas) que hay en un cierto nivel.
- `init_sort`: Esta función se encarga de inicializar la estructura de tipo `Sort`. Para ello, recibe el fichero del que se leerán los datos[1], la estructura que se va a inicializar, el número de niveles y procesos, y el retardo. Todos estos campos se guardarán en la estructura. Además, se calcularán qué datos están involucrados en las tareas de cada nivel, y se marcarán todas las tareas como incompletas.
- `check_task_ready`: Esta función indica si una tarea está o no lista para ser abordada. Para ello, comprobará si pertenece al Nivel 0 (en cuyo caso no depende de ninguna tarea), y si

[1] Este fichero contendrá una primera línea con el número de datos, y después un elemento por cada línea.

no fuera así, verificará que las tareas del nivel inferior que usaban los mismo datos hayan sido ya completadas.
- `solve_task`: Esta función resuelve una de las tareas, indicada por el nivel y la parte dentro del nivel. Llamará a `bubble_sort` o a `merge` según corresponda, pasándole los argumentos adecuados.
- `sort_single_process`: Esta es la función principal del sistema monoproceso. Inicializará el sistema, y tras ello resolverá por orden cada una de las tareas de cada nivel, mostrando tras cada una el estado de los datos.

SISTEMA MULTIPROCESO

El objetivo del proyecto es implementar una versión multiproceso del sistema de ordenación anterior.

Es importante destacar que ***se podrán realizar los cambios y modificaciones que se consideren necesarios en cualquiera de los módulos proporcionados***, añadiendo funciones y/o tipos de datos nuevos, o modificando los ya existentes.

Estructura

El sistema estará formado por los siguientes actores:

- **Proceso principal**: Es el proceso inicial, y será el encargado de la gestión, y de crear el resto de procesos.
- **Procesos trabajador**: Son los procesos que realizarán las tareas de ordenación. Son procesos hijos del proceso principal.
- **Proceso ilustrador**: Es el proceso encargado de representar por pantalla la evolución de los datos, de manera que se vea cómo se van ordenando. Es hijo del proceso principal.

A continuación se describirán los mecanismos de comunicación y concurrencia que usarán estos actores entre sí, así como las tareas que realizará cada uno.

Mecanismos de Comunicación y Concurrencia

El sistema utilizará los siguientes mecanismos de comunicación y concurrencia, aunque ***se podrán añadir tantos mecanismos adicionales como se consideren necesarios***:

- Los datos del sistema de ordenación se almacenarán en memoria compartida.
- Aquellos elementos de la memoria compartida en los que distintos procesos vayan a leer o escribir concurrentemente deberán protegerse mediante semáforos.
- El proceso principal utilizará una cola de mensajes para enviar las distintas tareas a los trabajadores (siguiendo la mecánica del *pool* de trabajadores).
- Cada vez que un trabajador termine una tarea, enviará una señal `SIGUSR1` al proceso principal para notificarle que hay una nueva tarea terminada.
- Los trabajadores utilizarán una señal de alarma (cada segundo) para saber que deben notificar su estado al proceso ilustrador.
- Los trabajadores realizarán la notificación al ilustrador, y esperarán su respuesta, mediante tuberías.
- El sistema podrá terminar en cualquier momento con la llegada de una señal de interrupción `SIGINT` (pulsando Ctrl+C), y deberá hacerlo de forma ordenada.

- El proceso principal indicará a los trabajadores y al ilustrador que deben terminar utilizando una señal de terminación `SIGTERM`.

Actores

A continuación se describen las tareas de cada uno de los procesos que participan en el sistema.

Proceso Principal Es el proceso padre (el único que lanza el usuario), y realizará las siguientes actividades:

- Creará una estructura de tipo `Sort` en memoria compartida.
- Inicializará la estructura con el contenido del fichero y los parámetros especificados.
- Creará la cola de mensajes para comunicarse con los trabajadores.
- Creará las tuberías necesarias para la comunicación entre trabajadores e ilustrador.
- Creará los semáforos necesarios para controlar la concurrencia.
- Creará los procesos trabajador e ilustrador.
- Para cada uno de los niveles del algoritmo de ordenación.
  - Enviará todas las tareas por cola de mensajes.
  - Cada vez que reciba una señal `SIGUSR1`, comprobará si ya se ha terminado ese nivel de ordenación.
  - Si ha terminado el nivel, comenzará con el siguiente.
  - Si no ha terminado, se suspenderá, o seguirá enviando las tareas del nivel que aún no se hayan enviado, sin realizar esperas activas.
- Cuando se hayan completado todos los niveles, o se haya recibido la señal `SIGINT`, avisará al resto de procesos usando la señal `SIGTERM`, esperará a que terminen y saldrá de forma ordenada liberando todos los recursos.

Trabajadores Son los procesos que realizarán las tareas de ordenación. En concreto, cada trabajador:

- Fijará una alarma que se repetirá cada segundo.
- Leerá un mensaje de la cola.
- Indicará en la lista de tareas en memoria compartida que esta tarea ya está en proceso, y comenzará a resolverla.
- Cuando termine, marcará la tarea como completada, y volverá a leer otro mensaje de la cola.
- Cuando reciba la señal `SIGALRM`, informará a través de una tubería al ilustrador de la tarea que está realizando (o indicará que no está realizando ninguna tarea), y se quedará bloqueado esperando la respuesta del ilustrador, también a través de una tubería.
- Si recibe la señal `SIGTERM`, liberará todos los recursos y saldrá de forma ordenada.

Ilustrador Es el proceso que mostrará el estado del algoritmo por pantalla, mostrando tanto los datos (para que se vea cómo se están ordenando) como la lista de trabajadores y qué está haciendo cada uno en ese momento. Para ello:

- Leerá a través de una tubería el estado de cada trabajador.
- Una vez que los haya leído todos, imprimirá el estado del sistema (incluyendo tanto los datos como la actividad de cada proceso) por pantalla.

- Responderá a todos los trabajadores indicando que ya ha terminado de mostrar el estado, y por tanto pueden continuar con su labor.
- Volverá a comenzar la lectura del estado de cada trabajador.
- Si recibe la señal `SIGTERM`, liberará todos los recursos y saldrá de forma ordenada.

Simplificaciones

A continuación se describen una serie de asunciones que se pueden hacer para el proyecto y que reducen bastante su complejidad:

- Al igual que en el sistema monoproceso provisto, se puede limitar el número máximo de niveles del sistema a 10, lo que permite saber de antemano que como mucho habrá 512 tareas en un nivel. Del mismo modo, el número máximo de procesos se truncará a 512, ya que es el número máximo de tareas que se van a poder resolver simultáneamente. Estas limitaciones permiten que todos los arrays se puedan reservar estáticamente utilizando la cota máxima que corresponda.
- Se puede salir, una vez recibida la señal `SIGINT` (en el caso del proceso principal) o `SIGTERM` (para los trabajadores y el ilustrador) directamente desde la función manejadora. Aunque no suele ser aconsejable, salir desde el cuerpo del programa requiere controlar la interrupción de varias señales durante el acceso a las colas y las tuberías, lo que aumenta mucho la complejidad si se quieren evitar todas las condiciones de carrera.
- Respecto al acceso concurrente a memoria:
  - Si el sistema está bien implementado, el proceso principal no introducirá tareas en cola hasta que las tareas de las que depende hayan terminado. Por tanto, nunca habrá dos trabajadores trabajando en los mismos datos a la vez, así que no hace falta controlar este acceso en las funciones de ordenación.
  - Si la interacción entre ilustrador y trabajadores es correcta, cuando el ilustrador muestre los datos los trabajadores estarán parados, y por tanto no tendrá que controlarse este acceso.
  - ***Sí es necesario*** controlar el acceso a la lista donde se indica el estado de las tareas, ya que varios procesos (los trabajadores y el proceso principal) pueden estar haciendo lecturas y escrituras al mismo tiempo. Sin embargo, ***no es necesario*** implementar un algoritmo de lectores–escritores, basta con una exclusión mutua.

Nota. *El funcionamiento del sistema multiproceso final debería ser similar al sistema monoproceso, salvo que permitirá la interrupción mediante la señal* `SIGINT`*, mostrará el estado de los datos cada segundo (en lugar de al acabar una tarea), y, por supuesto, que será resuelto por varios procesos en paralelo, lo que debería hacerlo considerablemente más rápido.*

Pasos para la Implementación

A continuación se sugiere una hoja de ruta para, a partir del sistema monoproceso provisto, llegar el sistema multiproceso con los requisitos especificados:

1. **Memoria Compartida**.
   - Transformar la memoria estática del sistema inicial en memoria compartida.
2. **Trabajador Único**
   - Modificar el proceso principal para:
     - Dentro cada nivel, y para cada tarea de ese nivel, crear un trabajador que ejecute esa tarea.
     - Esperar a que termine el único trabajador (no hay paralelismo real).

    - Tras esto, mostrar el estado del sistema y seguir con la siguiente tarea.
3. **Trabajadores Múltiples No Reutilizables**
    - Modificar el proceso principal para:
        - Dentro de cada nivel, crear tantos trabajadores como tareas haya en ese nivel.
        - Esperar a que terminen todos los trabajadores.
        - Tras esto, mostrar el estado del sistema (en este caso, se mostrará el estado del sistema una vez por nivel) y seguir con el siguiente nivel.
    - Modificar el trabajador para:
        - Realizar una única tarea recibida como argumento, y tras ello terminar.
4. **Cola de Mensajes**
    - Implementar la cola de mensajes.
    - Modificar el proceso principal para:
        - Dentro de cada nivel, crear tantos trabajadores como tareas haya en ese nivel.
        - Tras ello, meter en la cola todas las tareas de ese nivel.
        - Esperar a que terminen todos los trabajadores.
        - Tras esto, mostrar el estado del sistema y seguir con el siguiente nivel.
    - Modificar el trabajador para:
        - Leer de la cola una tarea.
        - Realizar la tarea y terminar.
5. **Trabajadores Múltiples Reutilizables**
    - Modificar el proceso principal para:
        - Crear los trabajadores especificados al inicio.
        - Dentro de cada nivel, meter en la cola todas las tareas de ese nivel.
        - Tras enviar todas las tareas de un nivel, esperar hasta que se completen.
        - Una vez que estén todas las tareas del nivel completadas, mostrar el estado del sistema y seguir con el siguiente nivel.
        - Cuando terminen todas las tareas, indicar a los trabajadores que terminen con una señal `SIGTERM`, liberar los recursos y terminar.
    - Modificar el trabajador para:
        - Leer de la cola una tarea.
        - Realizar la tarea y marcarla como completada.
        - Avisar al proceso padre con la señal `SIGUSR1`, y volver a leer otra tarea.
        - Salir si se recibe la señal `SIGTERM`.
    - Implementar los mecanismos de exclusión (semáforos) necesarios para acceder de forma concurrente a la lista de tareas y cambiar su estado.
6. **Señal de Interrupción**
    - Modificar el proceso principal para:
        - Capturar la señal `SIGINT`.
        - En caso de que se reciba la señal, avisar a los otros procesos con `SIGTERM`, liberar los recursos y terminar.
    - Modificar el trabajador para:
        - Ignorar la señal `SIGINT`.
7. **Ilustrador y Alarma**
    - Implementar las tuberías para comunicar el proceso ilustrador y los trabajadores.
    - Modificar el trabajador para:
        - Añadir la alarma cada segundo.
        - En caso de recibir la alarma, enviar el estado por la tubería, esperar la respuesta y continuar.
    - Como la alarma puede llegar en cualquier momento, garantizar que las llamadas bloqueantes se completan correctamente, y no por la llegada de la señal (tanto el

Down de los semáforos como la lectura de mensajes de la cola).

- Implementar el ilustrador para:
  - Leer el estado de los trabajadores.
  - Imprimir el estado del sistema por pantalla.
  - Notificar a los trabajadores de que ya se ha realizado la impresión, y volver a empezar.
  - Salir si se recibe la señal `SIGTERM`.
  - Ignorar la señal `SIGINT`.

Nota. *Esta no es una lista exhaustiva, y* ***no es de obligado cumplimento****. Se puede implementar directamente el sistema final, ya que no hay que entregar ninguna de las implementaciones intermedias. Sin embargo, estos pasos permiten probar el sistema tras modificaciones más pequeñas, manteniendo siempre su funcionalidad y facilitando su depuración.*

Entregables y Rúbrica

A continuación se detallan los entregables que deberán resultar de este proyecto, así como sus requisitos y la puntuación asociada a cada uno de ellos.

1,50 ptos. **Entregable 1: Memoria.**

0,50 ptos. a) Realizar un diagrama que muestre el diseño del sistema, incluyendo los distintos componentes (procesos) y sus jerarquías, así como los mecanismos de comunicación y sincronización entre ellos.

1,00 ptos. b) Describir el diseño del sistema, incluyendo los aspectos más relevantes, las limitaciones que tenga (si no se han implementado todos los requisitos, es importante especificarlo en este apartado), y los principales problemas que se han encontrado durante la implementación y cómo se han abordado.

+0,50 ptos. c) *(Opcional)* Comparar los tiempos de ejecución del sistema multiproceso con distintos números de procesos. Para ello, fijar el número de niveles a `10` y pintar una gráfica que muestre el tiempo de ejecución respecto al número de procesos. Comentar la evolución que se ve en esta gráfica.

8,50 ptos. **Entregable 2: Sistema Multiproceso.** Entregar el código del sistema multiproceso, incluyendo un fichero `Makefile` para compilarlo. Se valorarán los siguientes aspectos:

1,50 ptos. a) Funcionamiento general del proceso principal.

1,00 ptos. b) Funcionamiento general de los trabajadores.

1,00 ptos. c) Funcionamiento general del ilustrador.

1,00 ptos. d) Gestión correcta de la memoria compartida.

1,00 ptos. e) Gestión correcta de las tuberías.

1,00 ptos. f) Gestión correcta de las señales y alarmas.

1,00 ptos. g) Acceso concurrente correcto a la memoria compartida.

1,00 ptos. h) Gestión correcta de la cola de mensajes.

+0,50 ptos. i) *(Opcional)* Modificar el código del proceso principal para que, en lugar de esperar a que termine un nivel para meter en cola nuevas tareas, se compruebe tras la llegada de cada señal `SIGUSR1` si hay nuevas tareas que se puedan realizar, y en ese caso se pongan en cola. De esta forma, podrá haber tareas de distintos niveles ejecutándose simultáneamente, siempre que sean independientes entre sí.