Concurrencia y Paralelismo

GEI 2020

Práctica #3 (Compresión Concurrente con Paso de Mensajes)

Vamos a repetir la compresión concurrente de ficheros abordada en la práctica anterior,pero usando Erlang y paso de mensajes. El código proporcionado comprime un fichero utilizando un único proceso para realizar las tareas de compresión/descompresión. El sistema proporciona las siguientes funciones dentro del módulo `comp`:

- `comp(File)` , comprime un fichero `xxxx` a `xxxx.ch`.
- `comp(File, Chunk_Size)`, igual que la función anterior, pero permitiendo cambiar el tamaño de chunk.
- `decomp(File)`, descomprime un fichero `xxxx.ch` a `xxxx`.
- `decomp(Input_File, Output_File)`, descomprime `Input_File` a `Output_File`.

Compresión

La compresión se realiza utilizando dos procesos, uno que lee el fichero origen y otro que escribe en el archivo comprimido. El proceso lector se arranca desde la función `comp/2` usando la función `file_service:start_file_reader(File, Chunk_Size)`, que devuelve el pid del proceso lector. Ese proceso acepta dos mensajes:

- `{get_chunk, From}`, que lee el siguiente chunk de disco y lo envía al proceso `From` a través de un mensaje que puede ser:
  - `{chunk, Chunk_Number, Offset, Data}` si la lectura es correcta.
  - `eof` si no hay más chunks en el fichero.
  - `{error, Reason}` si la lectura falla.
- `stop`, que para el proceso.

El proceso escritor se arranca usando la función `archive:start_archive_writer(File)`, que devuelve el pid del proceso. El escritor acepta los siguientes mensajes:

- `{add_chunk, Num, Offset, Data}`, que añade el chunk con número `Num` y datos `Data` al fichero.
- `stop`, que para al proceso.

Descompresión

Para la descompresión se utilizan dos procesos similares, un lector de archivos comprimidos (`archive:start_archive_reader(File)`), con la misma interfaz que el lector de ficheros:

- `{get_chunk, From}`, que lee un chunk del fichero, y contesta con un mensaje que puede ser:
  - `{chunk, Chunk_Number, Offset, Data}`
  - `eof`
  - `{error, Reason}`
- stop.

Y un proceso escritor de ficheros (`file_service:start_file_writer(File)`), que acepta los siguiente mensajes:

- `{add_chunk, Offset, Data}`, que añade datos al fichero en la posición actual.
- `stop`
- `abort`, que borra el fichero que se estaba escribiendo.

Ejercicio 1 (**Implementar la compresión usando múltiples procesos**) La compresión se realiza en un único proceso que ejecuta la función `comp_loop`. Esta función se llama desde las funciónes `comp(File)` y `comp(File, Chunk_Size`. Implemente las funciones `comp_proc(File, Procs)` y `comp_proc(File, Chunk_Size, Procs)` que realicen la compresión utilizando `Procs` procesos. Todos los procesos arrancados deben pararse al terminar la compresión.

Ejercicio 2 (**Implementar la descompresión usando múltiples procesos**) La descompresión también se realiza en un único proceso en `decomp_loop`, que se llama desde `decomp(Archive)` y `decomp(Archive, Output_File)`. Implemente las funciones `decomp_proc(Archive, Procs)` y `decomp_proc(Archive, Output_file, Procs)` que realicen la descompresión utilizando `Procs` procesos. Todos los procesos iniciados deben pararse al terminar la descompresión.

Comprobación Finalización Procesos

Para comprobar la finalización correcta de los ficheros puede utilizarse el debugger, que se inicial con `debugger:start()`. A continuación seleccione los módulos de la práctica en `Module=>Interpret`. Durante la ejecución del programa podrá ver los procesos que ejecutan código en esos módulos.

Entrega

La fecha límite para la entrega es el viernes 27 de Marzo. La entrega debe realizarse en el proyecto cp-p3 del repositorio git.