Trabajo Práctico 3
Sincronización de procesos
Problemas de ordenamiento

Tecnología Digital II

El trabajo práctico debe realizarse en grupos de tres personas. Tienen dos semanas para realizarlo.
La fecha de entrega límite es el martes 22 de junio hasta las 23:59.
Se solicita no realizar consultas del trabajo práctico por los foros públicos. Limitar las preguntas al foro privado creado para tal fin.

Procesos y Threads

La creación de procesos se hace por medio de primitivas del sistema operativo como `fork`. Esta operación permite crear una copia nueva del proceso original, con la misma memoria que el proceso padre pero en un espacio de memoria diferente. Para que ambos procesos, padre e hijo compartan una parte de su memoria, esta debe ser compartida por medio de otros llamados al sistema.
Con el fin de simplificar toda esta operatoria, evitando llamados al sistema y reduciendo los costos de crear nuevos procesos, existen los *threads*. Estos ejecutan en el mismo espacio de memoria que el proceso padre, pero sobre un contexto de ejecución diferente.
En este trabajo práctico utilizaremos *threads* para construir algoritmos de ordenamiento. Los *threads* se crean de forma similar a utilizar `fork` y se espera por su terminación de forma similar a utilizar `wait`. Ambos llamados en términos de *threads* son respectivamente `pthread_create` y `pthread_join`. Estos llamados forman parte de la `libC` e implementan una interfaz para generar *threads* contra el sistema operativo.

A continuación se describen los parámetros de estas funciones:

- ```
  int pthread_create(pthread_t* thread,
                     const pthread_attr_t* attr,
                     void* (*routine)(void *),
                     void* arg);
  ```
  `thread`: Puntero al lugar donde se guardará el PID del thread creado.
  `attr`: Atributos del thread que será creado (`NULL` indica atributos por defecto).
  `routine`: Puntero a la rutina que será ejecutada en el nuevo thread.
  `arg`: Argumentos pasados al thread a crear (`NULL` indica sin parámetros).

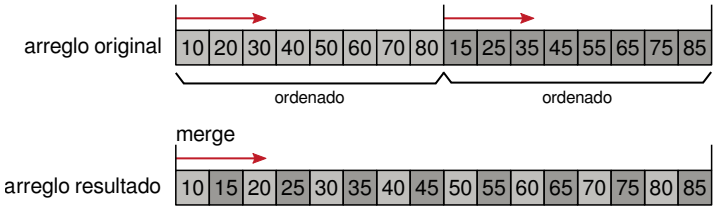
- `int pthread_join(pthread_t thread, void **status)`
  `thread`: Puntero al PID del thread por el que se quedará esperando.
  `status`: Valor de retorno del thread por el que se esperaba (`NULL` indica que debe ser ignorado).

Creando *threads* de esta forma, vamos a poder utilizar la misma memoria de datos entre todos contextos de ejecución.

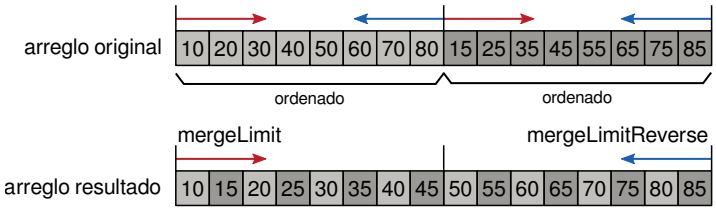
En los ejercicios que siguen, declararemos un arreglo en memoria y múltiples *threads* podrán acceder a este leyendo y modificando la memoria compartida. Es fundamental resolver la sincronización entre los *threads* para lograr un resultado correcto.
  ```

Enunciado

Los ejercicios buscan desarrollar algoritmos de ordenamiento utilizando múltiples procesos. Para esto vamos a partir de un algoritmo de ordenamiento básico como *bubble sort* para ordenar partes del arreglo total. Luego, las partes ordenadas las vamos a combinar utilizando un algoritmo de *merge* como se ilustra a continuación.



Para poder realizar el *merge* entre dos procesos, utilizarmos un algoritmo de *merge* limitado. Donde un proceso combina los valores más grandes y otro los más chicos como muestra la figura.

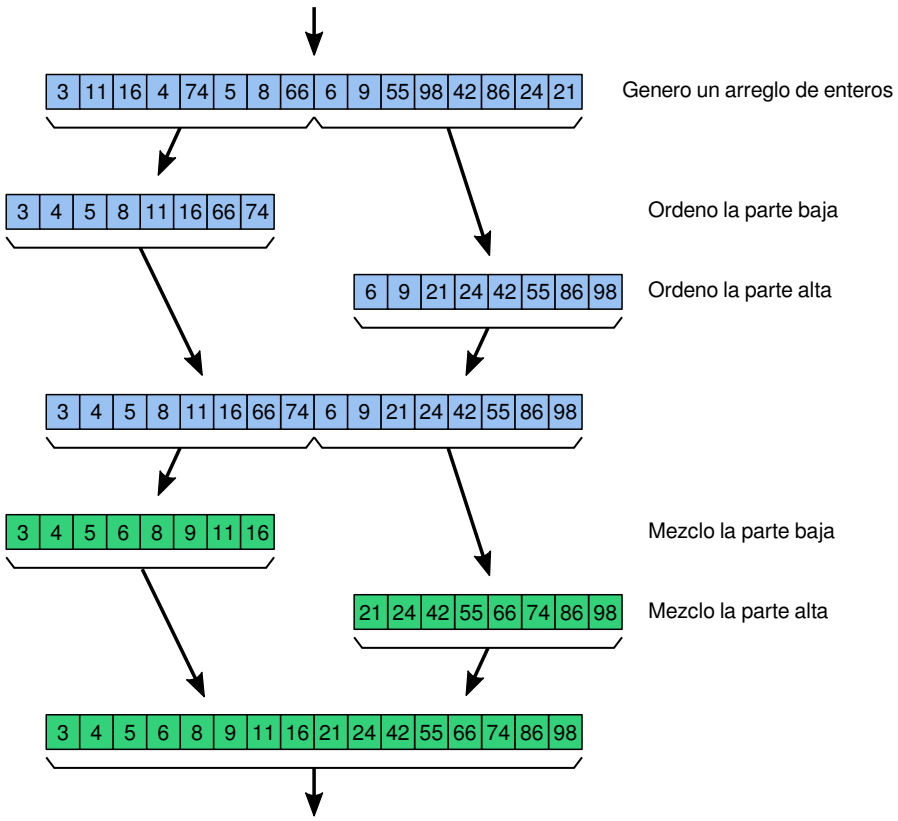


Ejercicio 1

En el archivo `common.c` encontrarán todas las funciones para hacer los ejercicios ya implementadas, a excepción del algoritmo de ordenamiento.
Se pide:

1. Estudiar las funciones provistas por la catedra. Probando su funcionamiento con ejemplos simples e incluso alterando parte del código provisto.

2. Implementar la función `bubbleSort`, escribiendo el algoritmo de un *ordenamiento burbuja*. En caso que gusten innovar e implementar un algoritmo más eficiente pueden hacerlo.

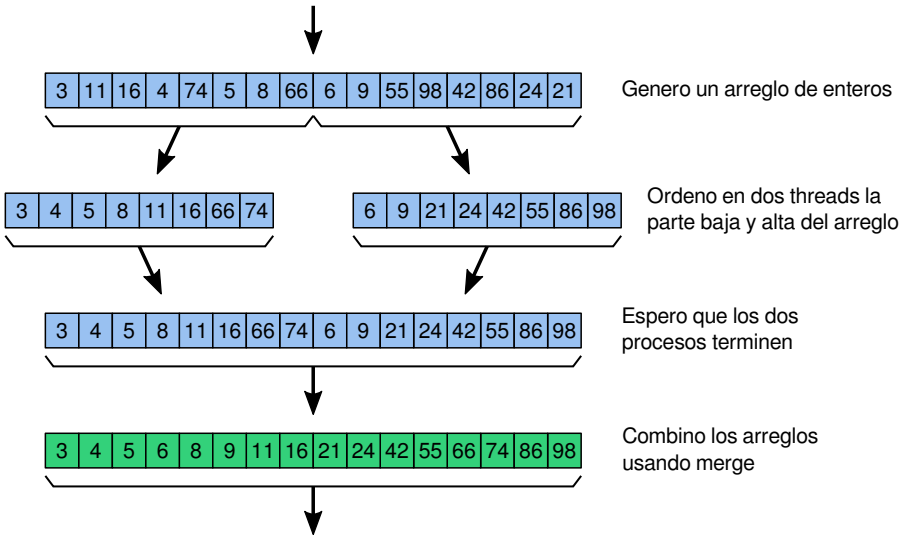


3. Verificar el correcto funcionamiento de las funciones de ordenamiento y *merge*, modificando el código de `sortProcess2.c` de forma que realice las mismas operaciones pero de forma secuencial. Es decir, utilizando un solo *thread*. Completar la solución en el archivo `sortProcess1Serial.c`.

Ejercicio 2

En el archivo `sortProcess2.c` se implementa un algoritmo de ordenamiento con dos *threads*. Teniendo en cuenta que al final de este proceso, la combinación de los datos ordenados se hace utilizando dos *threads*.
Se pide:

1. Modificar el algoritmo de ordenamiento con dos `threads` de forma que el paso de combinación de arreglos ordenados se realice de forma serial. Es decir, que un solo *thread* resuelva la combinación, para esto pueden utilizar la función `merge`. La solución debe ser completada en el archivo `sortProcess2MergeSerial.c`.

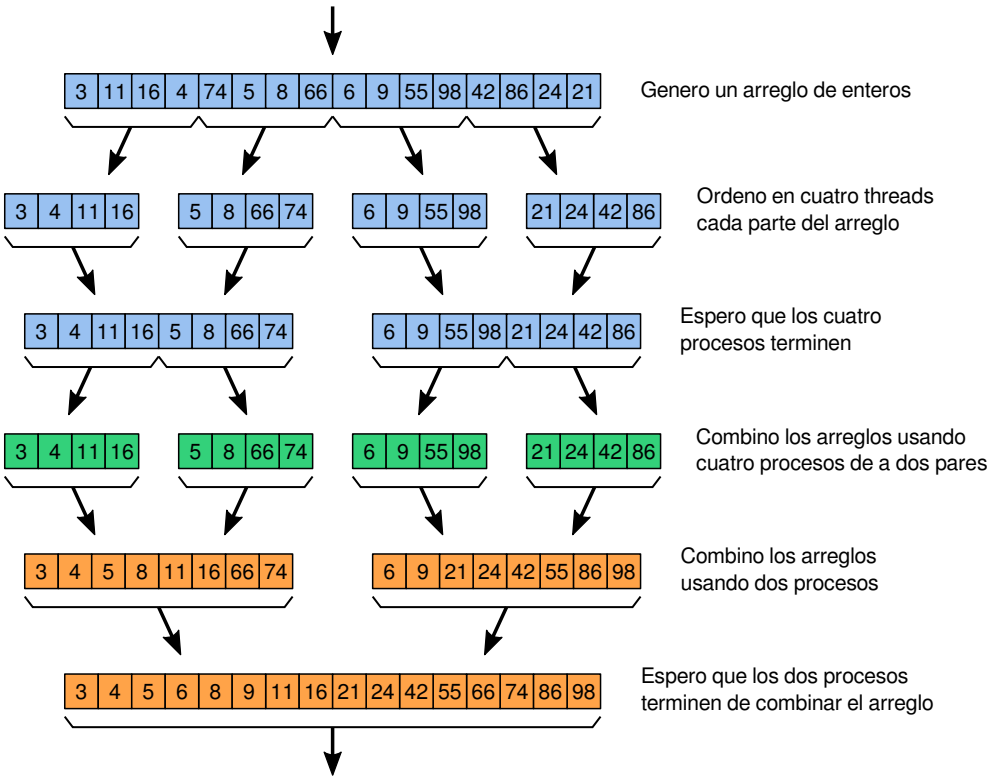


2. Utilizando el comando `time` en linux, calcular el porcentaje de diferencia en tiempo de ejecución entre ambas implementaciones. La implementación original y la nueva implementación utilizando un solo `thread` para combinar los arreglos ordenados. Tomar varias muestras para calcular cuanto tiempo demora en ejecutar, calculando un promedio de los resultados o tomando el mínimo valor observado. Indicar el tamaño de arreglo utilizado para el experimento, se recomienda un arreglo de 100000 valores.

Ejercicio 3

Utilizando como ejemplo la solución de ordenamiento con dos *threads* implementada en el archivo `sortProcess2.c`.
Se pide:

1. Implementar un algoritmo de ordenamiento utilizando 4 *threads*. Este algoritmo debe dividir el arreglo en 4 partes y ordenar independientemente cada una de estas. Luego, usando 4 *threads*, debe combinar de a pares las dos pares más grandes y las dos partes más chicas. El resultado de este proceso debe ser combinado utilizando otro par de *threads*. En la figura se ilustra el proceso con un ejemplo, notar que los colores identifican los distintos arreglos en memoria utilizados. Para la solución deben completar el archivo `sortProcess4.c`.



Ejercicio 4

Considerando todas las implementaciones de ordenamiento realizadas.
Se pide:

1. Realizar una tabla de tiempos, midiendo para cada implementación cuanto demora.

	size			
	100	1000	10000	100000
`sortProcess1`				
`sortProcess2`				
`sortProcess4`				

Las mediciones las pueden realizar utilizando el comando `time` al igual que en el ejercicio 2. Tener en cuenta, tomar varias muestras de cada experimento, sobre todo de los experimentos que demoran menos tiempo total. En la tabla de ejemplo se mencionan algunos tamaños de arreglo que deben medir como mínimo, pueden utilizar otros tamaños o cualquier valor intermedio.

Además presentar valores promedio de las muestras o el valor mínimo obtenido como representante del mejor tiempo de ejecución. Recordar que todas las muestras deben ser tomadas en una misma computadora y que no pueden estar otros procesos corriendo simultáneamente. Esto podría generar ruido en las mediciones. Además deben comentar cualquier acceso a entrada/salida, como imprimir en pantalla, pudiendo así medir solo el tiempo que demora en ejecutar.

Interpretación del código

Los programas están incompletos e incluyen instrucciones y tipos de datos que no vimos en clase. Parte del objetivo del trabajo es que investiguen de qué se trata cada elemento del código que no conozcan.

Entregable

La entrega debe constar de los mismos archivos .c que recibieron, pero con las partes faltantes completadas. El código que les damos ya hecho no debe ser alterado.

Por último, el código que realicen debe estar comentado. Deben explicar qué aspecto del problema resuelve cada parte del código y cómo la resuelve. No explicar instrucción por instrucción.

Para la experimentación deberán entregar un pequeño informe donde consten las tablas de resultados pedidas. Pueden incluso realizar gráficos de los datos o calcular el *speedup* de su solución y estos resultados presentarlos como parte del documento.