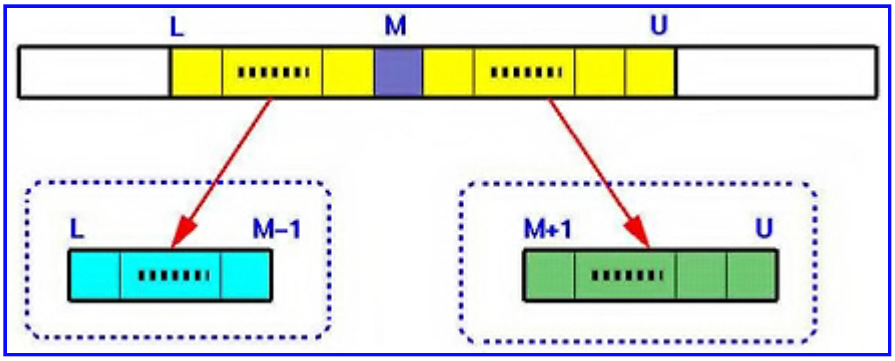
# Programming Assignment II

**Due on Wednesday, October 29th, 2021 @ 11:59pm**

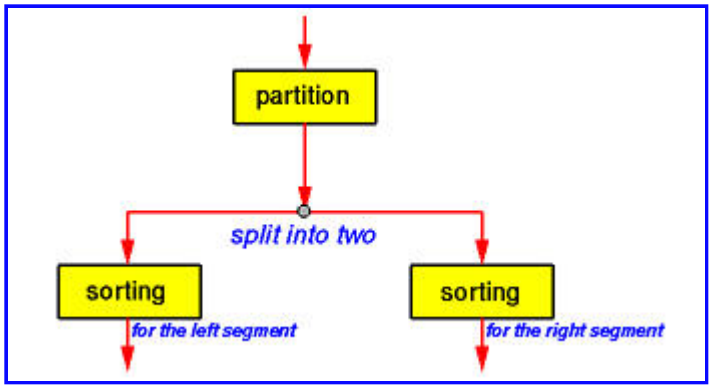
**100 points**

## Quicksort with Unix Shared Memory

Given an array section `a[L..R]`, a quicksort algorithm partitions it into two subsections with an array index `M` such that all elements in `a[L..M-1]` are smaller than or equal to `a[M]`, and all elements in `a[M+1..R]` are greater than or equal to `a[M]`. This means `a[M]` is at its final position in `a[L..R]`. This is shown below:

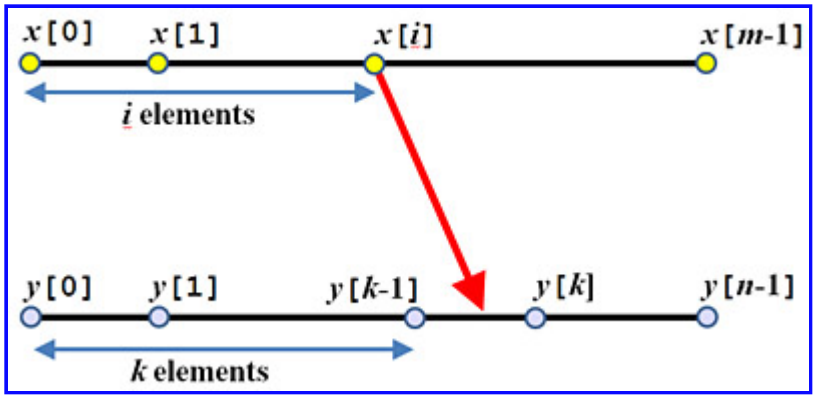



Since `a[L..M-1]` and `a[M+1..R]` can be sorted independently, we may use two processes, one for each array section as follows. This is the idea of this programming assignment.




## Binary Merge with Unix Shared Memory

Merging two sorted arrays can also be done concurrently. Suppose two sorted arrays $x[\ ]$ and $y[\ ]$, with $m$ and $n$ elements respectively, are to be merged into a sorted output array. Assume also that elements in $x[\ ]$ and $y[\ ]$ are all distinct. Consider an element $x[i]$. We know that it is larger than $i$ elements in array $x[\ ]$. If we are able to determine how many elements in array $y[\ ]$ are smaller than $x[i]$, we know the final location of $x[i]$ in the sorted array. This is illustrated in the diagram below:




With a slightly modified binary search, we can easily determine the location of $x[i]$ in the output array. There are three possibilities:

1. ***x[i] is less than y[0]***: In this case, $x[i]$ is larger than $i$ elements in array $x[\ ]$ and smaller than all elements of $y[\ ]$. Therefore, $x[i]$ should be in position $i$ of the output array.
2. ***x[i] is larger than y[n-1]***: In this case, $x[i]$ is larger than $i$ elements in $x[\ ]$ and $n$ elements in $y[\ ]$. Therefore, $a[i]$ should be in position $i+n$ of the output array.
3. ***x[i] is between y[k-1] and y[k]***: A slightly modified binary search will find a ***k*** such that ***x[i]*** is between ***y[k-1]*** and ***y[k]***. In this case, $x[i]$ is larger than $i$ elements in $x[\ ]$ and $k$ elements in $x[\ ]$. Therefore, $x[i]$ should be in position $i+k$ of the output array.

Hence, a program may create $m+n$ child processes, each of which handles an element in $x[\ ]$ or in $y[\ ]$. Each of these processes determines the location of its assigned element in the sorted array using $O(\log_2(m))$ or $O(\log_2(n))$ comparisons, and writes its value into the corresponding location. After all processes complete, the output array is sorted and is a combination of the two sorted input arrays.

## Program Specification

Write three programs `main.c`, `qsort.c` and `merge.c`. Program `main.c` does not require any command line arguments, `qsort.c` takes at least two command line arguments `Left` and `Right`, and you may determine the needed command line arguments for `merge.c`. The job for program `main.c` to do consists of the following:

1. Program `main.c` reads three input arrays $a[\ ]$, $x[\ ]$ and $y[\ ]$ into one or more shared memory segments. Let the number of elements of $a[\ ]$, $x[\ ]$ and $y[\ ]$ be $k$, $m$ and $n$, respectively.
2. `main.c` prints out the input arrays.
3. `main.c` creates a child process to run program `qsort.c` using the `execvp()` system call and passes the assigned `Left`, `Right` and other needed information to program `qsort.c`. Initially, `Left` and `Right` are 0 and $k$-1, respectively.
4. `main.c` creates a second child process to run program `merge.c` using the `execvp()` system call and passes the needed command line arguments to `merge.c`.
5. Then, `main.c` waits for both child processes to complete, prints out the results, and terminates itself.

The job for program `qsort.c` to do is the following:

1. When `qsort.c` runs, it receives the left and right section indices `Left` and `Right` and other information from its command line arguments.
2. Then, it partitions the array section `a[Left..Right]` into two at element `a[M]`. See your data structures and/or algorithms textbooks for this partitioning procedure. After the partition is obtained, two child processes are created, each of which runs `qsort.c` using system call `execvp()`. The first child

receives `Left` and `M-1`, while the second receives `M+1` and `Right`. The parent then waits until both child processes complete their job.
3. After this, program `qsort.c` exits.

The job for program `merge.c` to do is the following:

1. When `merge.c` runs, it receives whatever command line arguments.
2. Then, it creates *m*+*n* child processes, each of which is assigned to an element of *x*[ ] or an element of *y*[ ].
3. Each child process carries out a modified binary search as discussed earlier to determine the final location of the assigned element in the sorted array.
4. When the final location is found, the process writes its assigned value into that location of the output array.
5. `merge.c` waits until all of its child processes exit, and terminates itself.

**Important Notes**

- **You should only use shared memory and `execvp()` to solve this problem.**
- **You must use a modified binary search to determine the position of `x[i]` in `y[ ]`. Otherwise, you will receive no point for the binary merge component.**
- **The process structure must be as specified. Otherwise, you will receive no point.**

## Input and Output

The input to program `main.c` is in a file with the following format:

```
k           <--------------------- the number of elements of array a[ ]
a[0]  a[1]  a[2] ..... a[k-1] <--- elements of array a[ ]
m           <--------------------- the number of elements of array x[ ]
x[0]  x[1]  x[2] ..... x[m-1] <--- elements of array x[ ]
n           <--------------------- the number of elements of array y[ ]
y[0]  y[1]  y[2] ..... y[n-1] <--- elements of array y[ ]
```

You may assume the following:

- The value of `k`, `m` and `n` are integers larger than or equal to 5.
- The values for array `a[ ]` are ***distinct*** integers.
- The values for array `x[ ]` and array `y[ ]` are ***distinct*** integers.
- Array `x[ ]` and array `y[ ]` are sorted in increasing order.

The following shows a possible program output.

```
Quicksort and Binary Merge with Multiple Processes:

*** MAIN: shared memory key = 1627930027
*** MAIN: shared memory created
*** MAIN: shared memory attached and is ready to use

Input array for qsort has 8 elements:
    4  7  2  9  3  5  8  6

Input array x[] for merge has 6 elements:
    1  3  7  15  17  24

Input array y[] for merge has 5 elements:
```

```
     2  8  16  18  20

*** MAIN: about to spawn the process for qsort
   ### Q-PROC(4913): entering with a[0..7]
       ..........
   ### Q-PROC(3717): entering with a[3..6]
       XX  XX  XX  XX <---------------------- elements of a[3] to a[6]
       ..........
   ### Q-PROC(3717): pivot element is a[4] = XX
       ..........
   ### Q-PROC(3717): section a[3..6] sorted
       YY  YY  YY  YY
       ..........
   ### Q-PROC(3717): exits
       ..........
*** MAIN: sorted array by qsort:
     2  3  4  5  6  7  8  9

*** MAIN: about to spawn the process for merge
      $$$ M-PROC(5678): handling x[2] = 7
          ..........
      $$$ M-PROC(5678): x[2] = 7 is found between y[0] = 2 and y[1] = 8
          ..........
      $$$ M-PROC(5678): about to write x[2] = 7 into position 3 of the output array
          ..........
*** MAIN: merged array:
     1  2  3  7  8  15  16  17  18  20  24

*** MAIN: shared memory successfully detached
*** MAIN: shared memory successfully removed
```

It is also possible that an element may be smaller than every elements in the other array. In this case, you output should look like the following:

```
      $$$ M-PROC(7129): handling x[0] = 1
          ..........
      $$$ M-PROC(7129): x[0] = 1 is found to be smaller than y[0] = 2
          ..........
      $$$ M-PROC(7129): about to write x[0] = 1 into position 0 of the output array
          ..........
```

Or, if an element is larger than every elements in the other array. Then, print the following messages:

```
      $$$ M-PROC(3726): handling x[5] = 24
          ..........
      $$$ M-PROC(3726): x[5] = 24 is found to be larger than y[4] = 20
          ..........
      $$$ M-PROC(3726): about to write x[5] = 24 into position 10 of the output array
          ..........
```

**NOTE**: The number `nnnnn` in `PROC(nnnnn)` is the process ID of the process which generates the message.

**NOTE**: The above has the output of `qsort.c` and the output of `merge.c` grouped together in separate sections. However, this usually is not the case in reality. The output lines are mixed, and the order of output lines from different processes may also be very different.

**NOTE**: If you use more than one shared memory segments, you should repeat the output of shared memory segment and indicate the use of each. For example, if you allocate two shared memory segments, one for quicksort and the other for binary merge, your output should look like the following:

```
Quicksort and Binary Merge with Multiple Processes:
```

```
*** MAIN: qsort shared memory key = 1627930027
*** MAIN: qsort shared memory created
*** MAIN: qsort shared memory attached and is ready to use

*** MAIN: merge shared memory key = 1798523376
*** MAIN: merge shared memory created
*** MAIN: merge shared memory attached and is ready to use

<<<<<<<<<< Other Output >>>>>>>>>>

*** MAIN: qsort shared memory successfully detached
*** MAIN: qsort shared memory successfully removed

*** MAIN: merge shared memory successfully detached
*** MAIN: merge shared memory successfully removed
```

The output lines from `main.c` always starts with `*** MAIN:`. Program `main.c` first prints out the shared memory key being used, followed by messages indicating the shared memory being created and attached. Then, `main.c` prints out the input array.

Messages from `qsort.c` have an indentation of *three* spaces. The first message below indicates that the child process with process ID 3717 receives `Left` = 3 and `Right` = 6, and the current content of this array section. The second message below shows the pivot element `a[4]` and its value. The third message shows the sorted array section of `a[3..6]`. Finally, the fourth message indicates that this process exits.

```
### PROC(3717): entering with a[3..6]
    XX  XX  XX  XX <---------------------- elements of a[3] to a[6]
    ..........
### PROC(3717): pivot element is a[4] = XX
    ..........
### PROC(3717): section a[3..6] sorted
    YY  YY  YY  YY
    ..........
### PROC(3717): exits
```

Messages from `merge.c` have an indentation of *six* spaces.

```
$$$ M-PROC(5678): handling x[2] = 7
    ..........
$$$ M-PROC(5678): x[2] = 7 is found between y[0] = 2 and x[1] = 8
    ..........
$$$ M-PROC(5678): about to write x[2] = 7 into position 4 of the output array
    ..........
```

Each number printed uses 4 positions just like what is shown for printing input. Refer to the sample output format for indentation.

## Submission Guidelines

### General Rules

1. All programs must be written in C89.
2. Use `Canvas` to submit your work. This archive will contain both your program as well as your Readme.
3. Your program should be named as `main.c`, `qsort.c` and `merge.c`. Since Unix filename is case sensitive, `MAIN.c`, `main.C`, `Main.c`, etc are ***not*** acceptable.
4. We will use the following approach to compile and test your programs:

```
gcc     main.c -o main        <-- compile main.c
gcc     qsort.c -o qsort      <-- compile qsort.c
gcc     merge.c -o merge      <-- compile merge.c
./main < input-file           <-- test your program
```

This procedure may be repeated a number of times with different input files to see if your program works correctly.

5. Your implementation should fulfill the program specification as stated. Any deviation from the specifications will cause you to receive **zero** point.
6. A `README` file is always required.
7. **No late submission will be graded.**
8. **Programs submitted to wrong class and/or wrong section will not be graded.**

## Compiling and Running Your Programs

This is about the way of compiling and running your program. If we cannot compile your program due to syntax errors, wrong file names, etc, **we cannot test your program, and, as a result, you receive 0 point**. If your program compiles successfully but fails to run, **we cannot test your program, and, again, you receive 0 point**. **Therefore, before submitting your work, make sure your program can compile and run properly.**

1. **Not-compile programs receive 0 point.** By **not-compile**, I mean any reason that could cause an unsuccessful compilation, including missing files, incorrect filenames, syntax errors in your programs, and so on. Double check your files before you submit, since I will **not** change your program. Note again: Unix filenames are ***case sensitive***.
2. **Compile-but-not-run programs receive 0 point. Compile-but-not-run** usually means you have attempted to solve the problem to some degree but you failed to make it working properly.
3. **A meaningless or vague program receives 0 point even though it compiles successfully.** This usually means your program does not solve the problem but serves as a placeholder or template just making it to compile and run.

## Program Style and Documentation

This section is about program style and documentation.

1. For each file, the first piece should be a program header to identify yourself like this:

```
/* ------------------------------------------------------------ */
/* NAME : John Smith                          User ID: xxxxxxxx */
/* DUE DATE : mm/dd/yyyy                                        */
/* PROGRAM ASSIGNMENT #                                         */
/* FILE NAME : xxxx.yyyy.zzzz (your unix file name)             */
/* PROGRAM PURPOSE :                                            */
/*    A couple of lines describing your program briefly         */
/* ------------------------------------------------------------ */
```

Here, **User ID** is the one you use to login. It is ***not*** your social security number nor your M number.

For each function in your program, include a simple description like this:

```
/* ------------------------------------------------------------ */
/* FUNCTION  xxyyzz : (function name)                           */
/*     the purpose of this function                             */
/* PARAMETER USAGE :                                            */
/*    a list of all parameters and their meaning                */
/* FUNCTION CALLED :                                            */
```

```
/*    a list of functions that are called by this one          */
/* ------------------------------------------------------------ */
```

2. Your programs must contain enough concise and to-the-point comments. Do not write a novel!
3. Your program should have good indentation.
4. Do not use global variables!

## Program Specification

**Your program must follow exactly the requirements of this programming assignment. Otherwise, you receive 0 point even though your program runs and produces correct output.**
The following is a list of potential problems.

1. Your program does not use the indicated algorithms/methods to solve this problem.
2. Your program does not follow the structure given in the specification. For example, your program is not divided into functions and files, etc when the specification says you should.
3. Any other significant violation of the given program specification.
4. **Incorrect output format.** This will cost you some points depending on how serious the violations are. The grader will make a decision. Hence, carefully check your program output against the required one.

## Program Correctness

If your program compiles and runs, we will check its correctness. We run your program with at least two sets of input data, one posted on this programming assignment page (the public one) and the other prepared by the grader (the private one). You program must deliver correct results for both data sets. Depending on the seriousness of the problem(s), significant deduction may be applied. For example, if your program delivers all wrong results for the public data set, you receive 0 point for that component.

## Shared Memory Issues

Since shared memory segments are system-wide entities and will stay in the system after you logout, you are responsible to remove **ALL** shared memory segments whether your program ended normally or abnormally. Each un-removed shared memory segment will cost you 10 points. Thus, if you have a correct program and if the grader finds out your program has two shared memory segments left behind, you receive no more than 80 points (*i.e.*, 20 points deduction).

## The `README` File

A file named `README` is required to answer the following questions:

1. The logic of your program
2. Why does your program work?
3. Explain the allocation and use of each shared memory segment.
4. Are there potential race conditions (*i.e.*, processes use and update a shared data item concurrently) in your program and in the program specification?
5. How do you construct an argument list that is passed from program `main` to program `qsort`?
6. How do you construct an argument list that is passed from program `main` to program `merge`?

You should elaborate your answer and provide details. **When answering the above questions, make sure each answer starts with a new line and have the question number (*e.g.*, Question X:) clearly shown. Separate two answers with a blank line.**

Note that the filename has to be `README` rather than `readme` or `Readme`. Note also that there is ***no*** filename extension, which means filename such as `README.TXT` is ***NOT*** acceptable.

**`README` must be a plain text file. We do not accept files produced by any word processor. Moreover, watch for very long lines. More precisely, limit the length of each line to no more than 80 characters with the Return/Enter key for line separation. Missing this file, submitting non-text file, file with long lines, or providing incorrect and/or vague answers will cost you many points. Suggestion:** Use a Unix text editor to prepare your `README` rather than a word processor.

## Final Notes

1. Your submission should include four files, namely: `main.c`, `qsort.c`, `merge.c` and `README`. Please note the way of spelling filenames.
2. **Always start early, because I will not grant any extension if your home machine, network connection, your phone line or the department machines crash in the last minute.**
3. **Since the rules are all clearly stated, no leniency will be given and none of the above conditions is negotiable. So, if you have anything in doubt, please ask for clarification.**
4. **Click here to see how your program will be graded.**