CSE 3100: Systems Programming

Part 2

Lecture 5: Exam 2 Review

Exam Format (1)

- You can ONLY take the exam in your own lab section.
- This is a 110-minute exam.
- During the exam, you cannot communicate with and/or obtain help from people other than TAs and instructors on exam questions. Particularly, you cannot use any messaging applications.
- During the exam, you can only access data/files on HuskyCT, your Virtual Machine, and files on your own laptop and on the lab computer you use.

Exam Format (2)

- The exam has two parts:

Part 1: Multiple choice questions.

Part 2: 2 coding questions.

Open book/open notes/open VM. No internet other than HuskyCT/Linux manual pages.

Today's Lecture

1. Practice Exam 2 Solutions

Problem 1. pipe-sort.
In this problem, we use two child processes and pipes to sort an array of integers in the parent process into increasing order.

Specifically, the parent process generates `n` random integers, where `n` must be a positive even number, and saves them in array `u`. Then it creates pipes and forks two child processes so that child process 1 sorts the first half of array `u` and child process 2 sorts the second half. Both child processes just call `qsort()` to sort their array in place. Then they send the sorted integers to the parent process through pipes.

The parent process receives the sorted integers from both child processes through pipes, and saves them into array `u`. Then it merges the two sorted halves into array `sorted`.

The child processes call `qsort()` to sort arrays and the compare function for integers is provided in `compare_int()`.

My Approach To Coding Hard Problems:

1. Sketch = Write down the steps needed to do the algorithm/task.
2. Break = Break it into smaller pieces and solve each piece.
3. Build = Connect the pieces together one at a time.

Step 1: Sketch

- Sketch out the parts of the problem (on paper)

Problem 1. pipe-sort.
In this problem, we use two child processes and pipes to sort an array of integers in the parent process into increasing order.

Specifically, the parent process generates `n` random integers, where `n` must be a positive even number, and saves them in array `u`. Then it creates pipes and forks two child processes so that child process 1 sorts the first half of array `u` and child process 2 sorts the second half. Both child processes just call `qsort()` to sort their array in place. Then they send the sorted integers to the parent process through pipes.

The parent process receives the sorted integers from both child processes through pipes, and saves them into array `u`. Then it merges the two sorted halves into array `sorted`.

The child processes call `qsort()` to sort arrays and the compare function for integers is provided in `compare_int()`.

1. Parent: Generate an array with n random integers

Step 1: Sketch

- Sketch out the parts of the problem (on paper)

Problem 1. pipe-sort.
In this problem, we use two child processes and pipes to sort an array of integers in the parent process into increasing order.

Specifically, the parent process generates `n` random integers, where `n` must be a positive even number, and saves them in array `u`. Then it creates pipes and forks two child processes so that child process 1 sorts the first half of array `u` and child process 2 sorts the second half. Both child processes just call `qsort()` to sort their array in place. Then they send the sorted integers to the parent process through pipes.

The parent process receives the sorted integers from both child processes through pipes, and saves them into array `u`. Then it merges the two sorted halves into array `sorted`.

The child processes call `qsort()` to sort arrays and the compare function for integers is provided in `compare_int()`.

1. Parent: Generate an array u with n random integers

2. A. Child process sorts first half of u

Step 1: Sketch

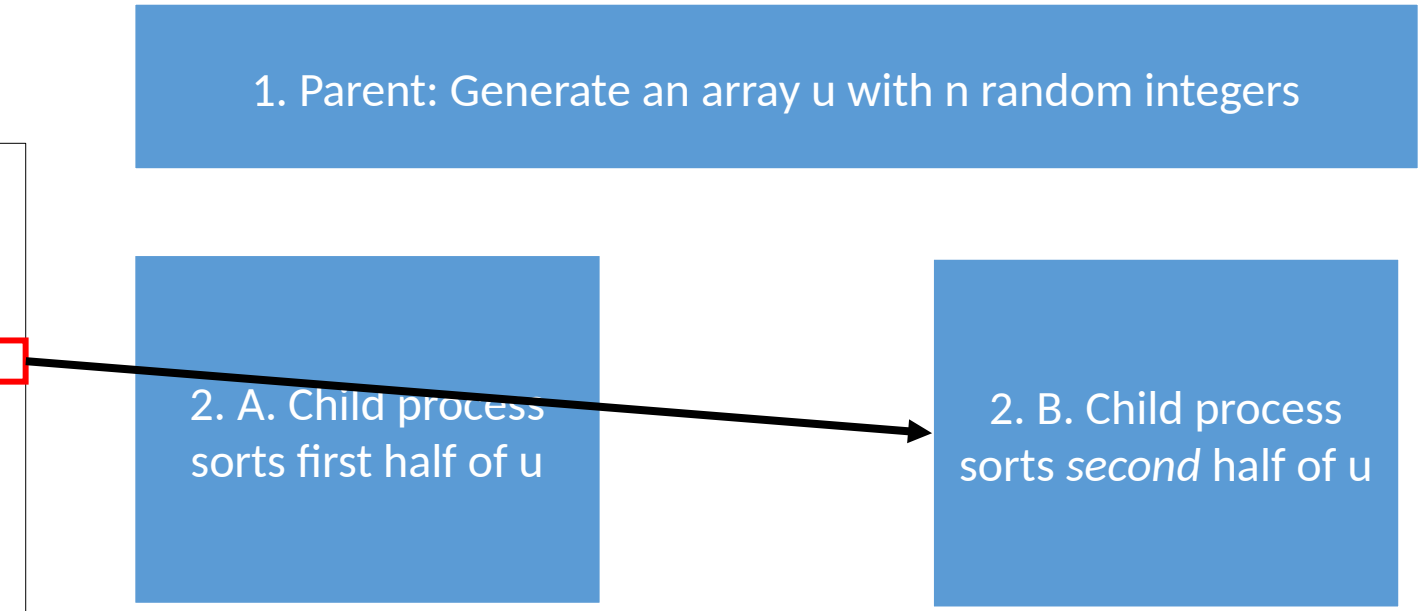
- Sketch out the parts of the problem (on paper)

Problem 1. pipe-sort.
In this problem, we use two child processes and pipes to sort an array of integers in the parent process into increasing order.

Specifically, the parent process generates `n` random integers, where `n` must be a positive even number, and saves them in array `u`. Then it creates pipes and forks two child processes so that child process 1 sorts the first half of array `u` and child process 2 sorts the second half. Both child processes just call `qsort()` to sort their array in place. Then they send the sorted integers to the parent process through pipes.

The parent process receives the sorted integers from both child processes through pipes, and saves them into array `u`. Then it merges the two sorted halves into array `sorted`.

The child processes call `qsort()` to sort arrays and the compare function for integers is provided in `compare_int()`.



Step 1: Sketch

- Sketch out the parts of the problem (on paper)

Problem 1. pipe-sort.
In this problem, we use two child processes and pipes to sort an array of integers in the parent process into increasing order.

Specifically, the parent process generates `n` random integers, where `n` must be a positive even number, and saves them in array `u`. Then it creates pipes and forks two child processes so that child process 1 sorts the first half of array `u` and child process 2 sorts the second half. Both child processes just call `qsort()` to sort their array in place. Then they send the sorted integers to the parent process through pipes.

The parent process receives the sorted integers from both child processes through pipes, and saves them into array `u`. Then it merges the two sorted halves into array `sorted`.

The child processes call `qsort()` to sort arrays and the compare function for integers is provided in `compare_int()`.

1. Parent: Generate an array u with n random integers

2. A. Child process sorts first half of u using qsort.

2. B. Child process sorts *second* half of u using qSort.

Step 1: Sketch

- Sketch out the parts of the problem (on paper)

Problem 1. pipe-sort.
In this problem, we use two child processes and pipes to sort an array of integers in the parent process into increasing order.

Specifically, the parent process generates `n` random integers, where `n` must be a positive even number, and saves them in array `u`. Then it creates pipes and forks two child processes so that child process 1 sorts the first half of array `u` and child process 2 sorts the second half. Both child processes just call `qsort()` to sort their array in place. Then they send the sorted integers to the parent process through pipes.

The parent process receives the sorted integers from both child processes through pipes, and saves them into array `u`. Then it merges the two sorted halves into array `sorted`.

The child processes call `qsort()` to sort arrays and the compare function for integers is provided in `compare_int()`.

1. Parent: Generate an array u with n random integers.
Create a pipe for child A and create a pipe for child B.

2. A. Child process sorts first half of u using qsort.

2. B. Child process sorts *second* half of u using qSort.

Step 1: Sketch

- Sketch out the parts of the problem (on paper)

Problem 1. pipe-sort.
In this problem, we use two child processes and pipes to sort an array of integers in the parent process into increasing order.

Specifically, the parent process generates `n` random integers, where `n` must be a positive even number, and saves them in array `u`. Then it creates pipes and forks two child processes so that child process 1 sorts the first half of array `u` and child process 2 sorts the second half. Both child processes just call `qsort()` to sort their array in place. Then they send the sorted integers to the parent process through pipes.

The parent process receives the sorted integers from both child processes through pipes, and saves them into array `u`. Then it merges the two sorted halves into array `sorted`.

The child processes call `qsort()` to sort arrays and the compare function for integers is provided in `compare_int()`.

1. Parent: Generate an array u with n random integers. Create a pipe for child A and create a pipe for child B.

2. A. Child process sorts first half of u using qsort. Send data back to parent.

2. B. Child process sorts *second* half of u using qSort. Send data back to parent.

Step 1: Sketch

- Sketch out the parts of the problem (on paper)

Problem 1. pipe-sort.
In this problem, we use two child processes and pipes to sort an array of integers in the parent process into increasing order.

Specifically, the parent process generates `n` random integers, where `n` must be a positive even number, and saves them in array `u`. Then it creates pipes and forks two child processes so that child process 1 sorts the first half of array `u` and child process 2 sorts the second half. Both child processes just call `qsort()` to sort their array in place. Then they send the sorted integers to the parent process through pipes.

The parent process receives the sorted integers from both child processes through pipes, and saves them into array `u`. Then it merges the two sorted halves into array `sorted`.

The child processes call `qsort()` to sort arrays and the compare function for integers is provided in `compare_int()`.

1. Parent: Generate an array u with n random integers.
Create a pipe for child A and create a pipe for child B.

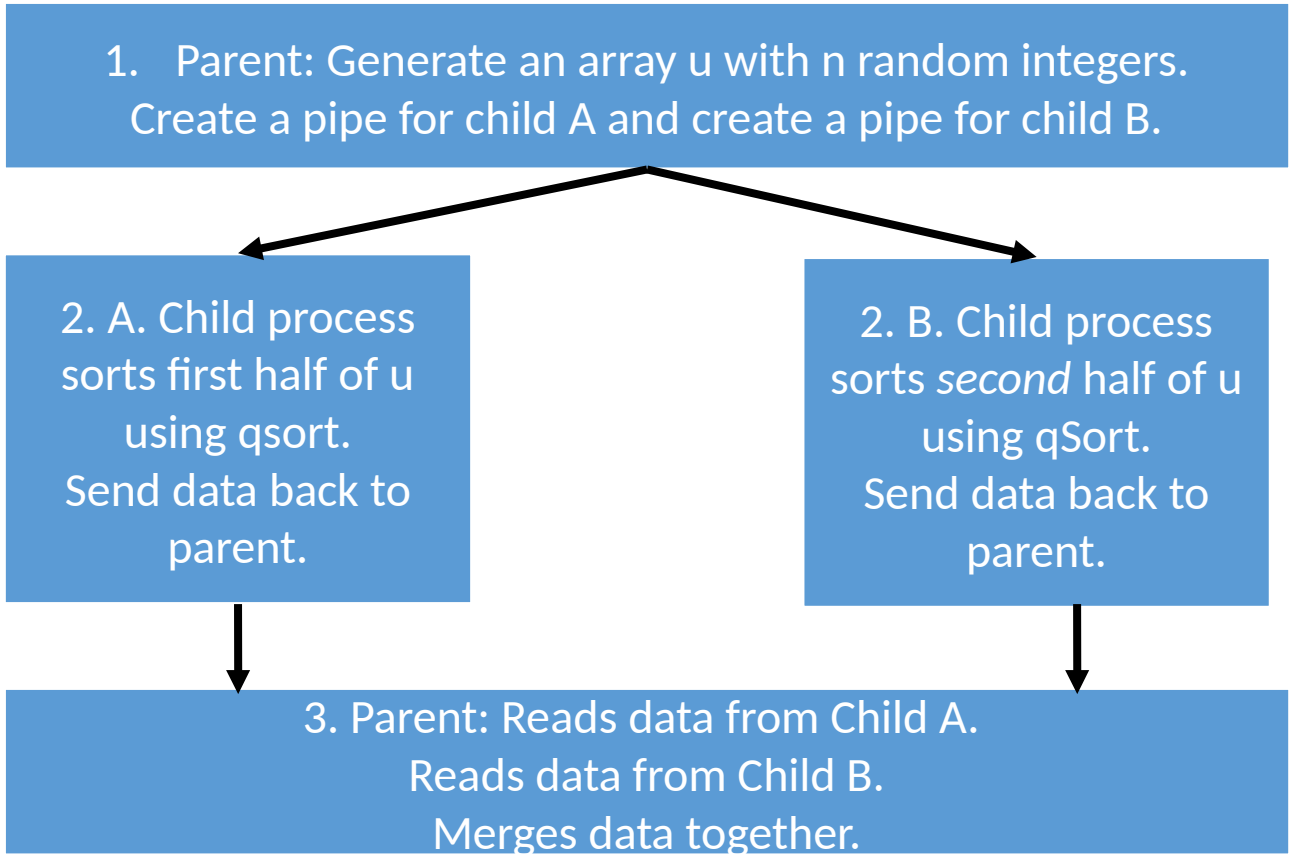
2. A. Child process sorts first half of u using qsort.
Send data back to parent.

2. B. Child process sorts *second* half of u using qSort.
Send data back to parent.

3. Parent: Reads data from Child A.
Reads data from Child B.
Merges data together.

Complete Sketch

- Sketch out the parts of the problem (on paper)



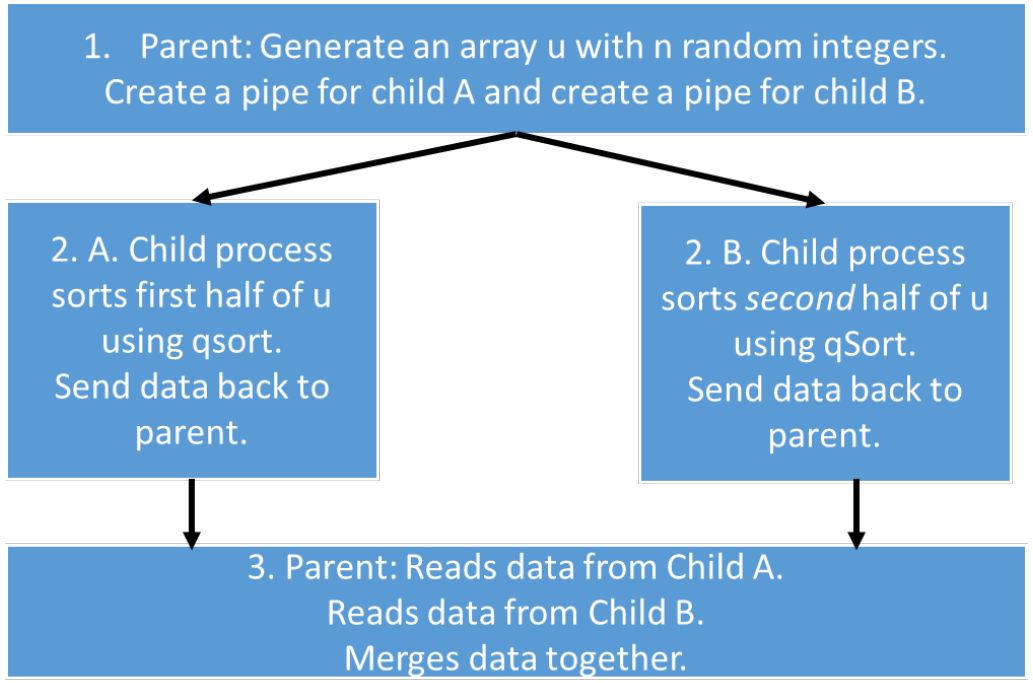
Next question: *What is actually given to us in the starter code?*

And what do we have to do ourselves?

Step 2: Break

- Figure out what is given to us and what we need to break down and solve

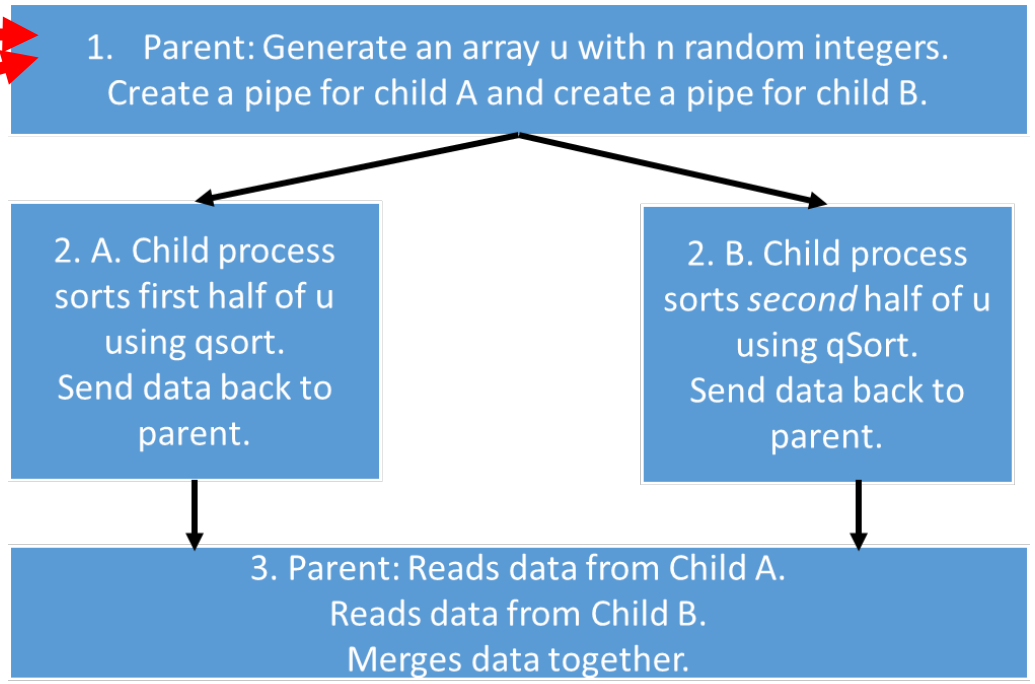
```
80  void pipe_sort(int seed, int n, int print_sorted, int num_printed){
81
82      srand(seed);    // set the seed
83
84      // prepare arrays
85      // u has all the integers to be sorted
86      // a is the first half and b is the second half
87      int u[n];
88      int *a, *b;
89      int half = n / 2;
90
91      a = u;
92      b = a + half;
93
94      for (int i = 0; i < n; i++)
95          u[i] = rand() % n;
96
97      if (! print_sorted) {
98          print_array(u, n, num_printed);
99          fprintf(stderr, "The unsorted array has been printed to stdout.\n");
100         exit(EXIT_SUCCESS);
101     }
102
103     int pd1[2], pd2[2];
104
```



Step 2: Break

- The array is already generated for us. Great!

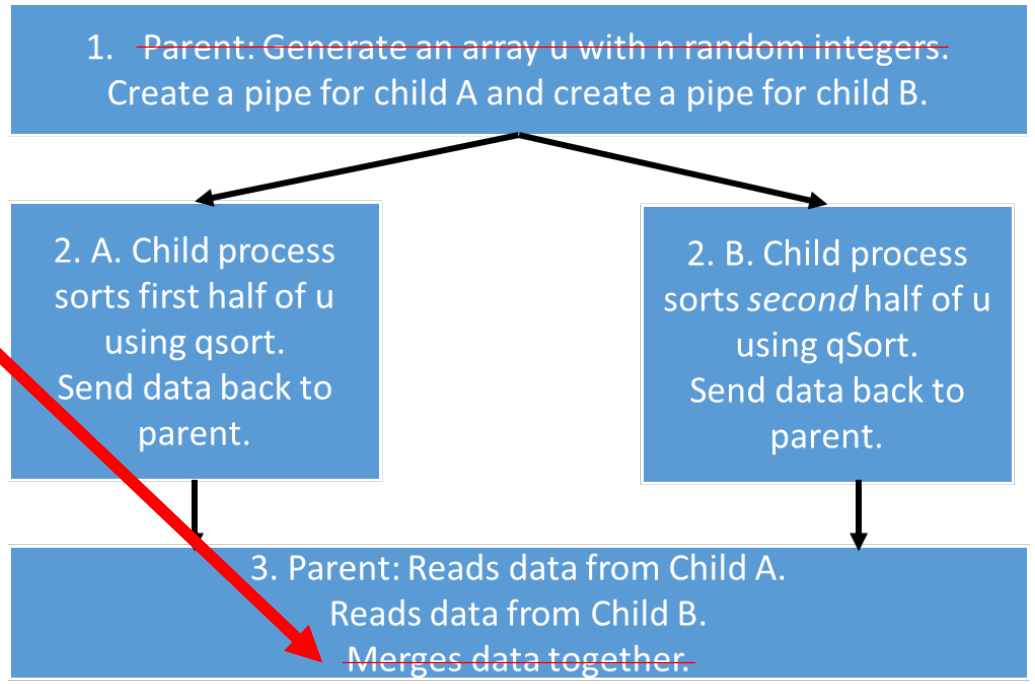
```
80  void pipe_sort(int seed, int n, int print_sorted, int num_printed){
81
82      srand(seed);    // set the seed
83
84      // prepare arrays
85      // u has all the integers to be sorted
86      // a is the first half and b is the second half
87      int u[n];
88      int *a, *b;
89      int half = n / 2;
90
91      a = u;
92      b = a + half;
93
94      for (int i = 0; i < n; i++)
95          u[i] = rand() % n;
96
97      if (! print_sorted) {
98          print_array(u, n, num_printed);
99          fprintf(stderr, "The unsorted array has been printed to stdout.\n");
100         exit(EXIT_SUCCESS);
101     }
102
103     int pd1[2], pd2[2];
104
```



Step 2: Break

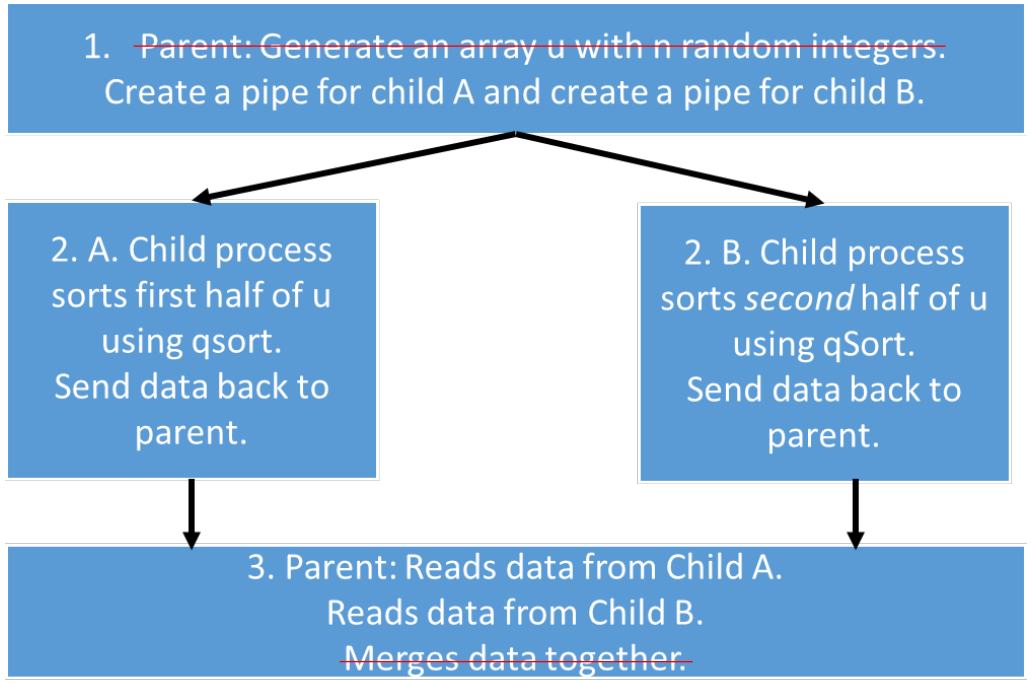
- Skimming through the code…

```
136
137         int sorted[n];
138         merge(a, b, sorted, half);
139         if (print_sorted)
140             print_array(sorted, n, num_printed);
141     }
142
```



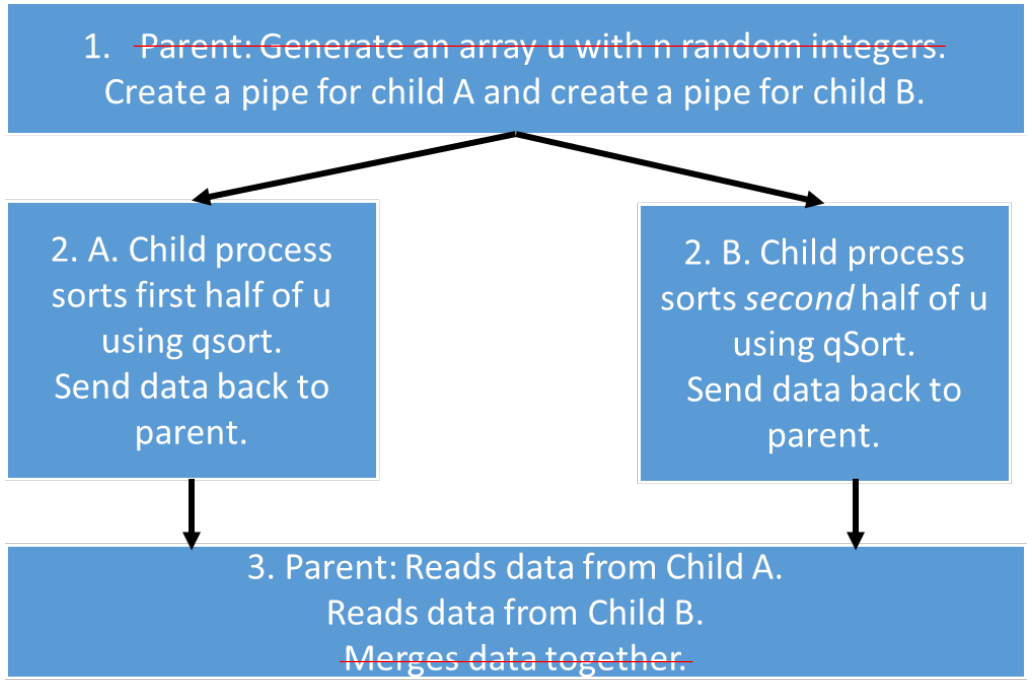
Step 2: Break

- So now from the starter code what are the pieces we need to do?



Step 2: Break

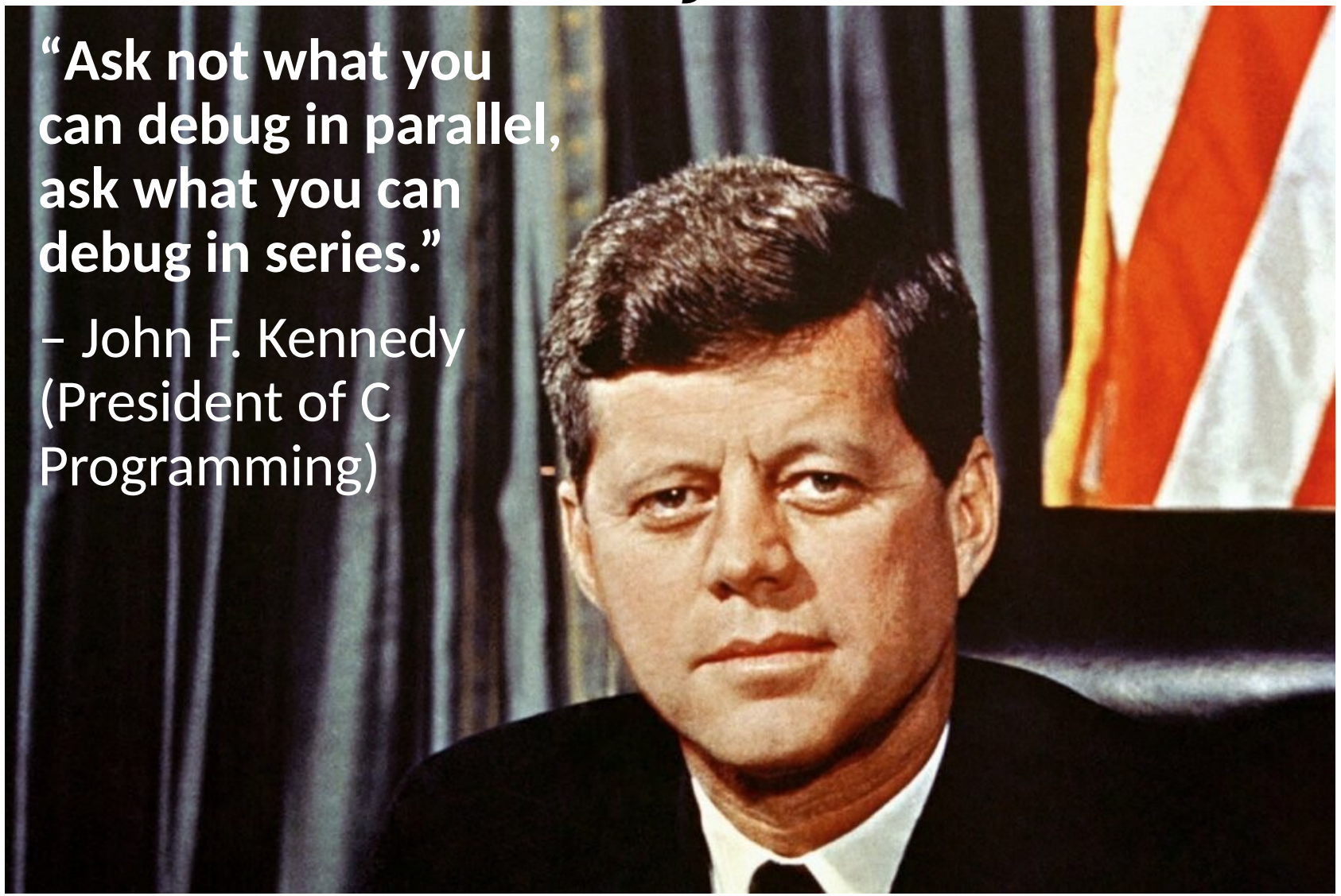
- So now from the starter code what are the pieces we need to do?



1. We have to setup pipes for child A and child B.
2. We have to create child A and child B.
3. We have to be able to use qSort.
4. We have to send data from the child A to pipeA (write).
5. We have to send data from child B to the pipeB (write).
6. Parent has to read from pipe A and pipe B (read).

Where do we go from here?

Does anyone remember this famous quote from JFK?



Which of these parts can be done *without* needing fancy fork and pipe calls?

1. We have to setup pipes for child A and child B.
2. We have to create child A and child B.
3. We have to be able to use qSort.
4. We have to send data from the child A to pipeA (write).
5. We have to send data from child B to the pipeB (write).
6. Parent has to read from pipe A and pipe B (read).

Step 2: Break – Using qSort

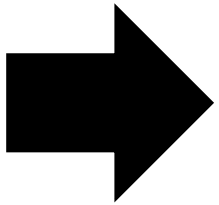
- Building is best if you can isolate and test simple examples

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>

int main(){
    return 0;
}

```

Step 2: Break – Using qSort

- Copy the array creation part from the starter code and make a simple test case

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>

int main(){
    return 0;
}

```

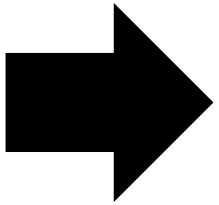


```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>

int main(){
    //create simple test case
    int seed = 0;
    srand(seed);    // set the seed
    int n = 10;
    int u[n];
    for (int i = 0; i < n; i++)
        u[i] = rand() % n;
    return 0;
}
```

Step 2: Break – Using qSort

- Add a print statement to make sure our array is randomly generated correctly:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>

int main(){
    //create simple test case
    int seed = 0;
    srand(seed);    // set the seed
    int n = 10;
     int u[n];
     for (int i = 0; i < n; i++)
        u[i] = rand() % n;
    return 0;
}
```

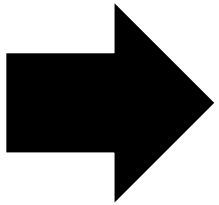


```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>

int main(){
    //create simple test case
    int seed = 0;
    srand(seed);    // set the seed
    int n = 10;
    int u[n];
    for (int i = 0; i < n; i++){
        u[i] = rand() % n;
    }
    //check the array before sorting
    printf("Before sorting\n");
    for (int i = 0; i < n; i++){
        printf("Array u[%d]=%d\n",i, u[i]);
    }
    return 0;
}
```

Step 2: Break – Using qSort

- Add a print statement to make sure our array is randomly generated correctly:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>

int main(){
    //create simple test case
    int seed = 0;
    srand(seed);    // set the seed
    int n = 10;
    int u[n];
    for (int i = 0; i < n; i++){
        u[i] = rand() % n;
    }
    //check the array before sorting
    printf("Before sorting\n");
    for (int i = 0; i < n; i++){
        printf("Array u[%d]=%d\n",i, u[i]);
    }
    return 0;
}
```



```
kaleel@CentralCompute:~$ ./test
Before sorting
Array u[0]=3
Array u[1]=6
Array u[2]=7
Array u[3]=5
Array u[4]=3
Array u[5]=5
Array u[6]=6
Array u[7]=2
Array u[8]=9
Array u[9]=1
```

Step 2: Break – Using qSort

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>

int main(){
    //create simple test case
    int seed = 0;
    srand(seed);    // set the seed
    int n = 10;
    int u[n];
    for (int i = 0; i < n; i++){
        u[i] = rand() % n;
    }
    //check the array before sorting
    printf("Before sorting\n");
    for (int i = 0; i < n; i++){
        printf("Array u[%d]=%d\n",i, u[i]);
    }
    return 0;
}
```

- Now we need to sort the list using qSort. The syntax for qSort was given in part 1 lecture 8:

Example: quicksort in C library

- The prototype (in <stdlib.h>)

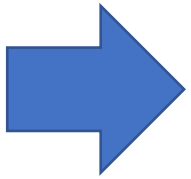
```c
void qsort(void * base,
           size_t nel,
           size_t width,
           int (*compare)(const void *, const void *));
```

- qsort takes…
  - base: the address of the array as an untyped pointer
  - nel: the number of elements in the array
  - width: the size (in byte) of ONE element of the array
  - compare: a **pointer to a function** that compares two values

Step 2: Break – Using qSort

- Copy in the compare_int function from the starter code and call qsort:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>

int main(){
    //create simple test case
    int seed = 0;
    srand(seed);    // set the seed
    int n = 10;
    int u[n];
    for (int i = 0; i < n; i++){
        u[i] = rand() % n;
    }
    //check the array before sorting
    printf("Before sorting\n");
    for (int i = 0; i < n; i++){
        printf("Array u[%d]=%d\n",i, u[i]);
    }
    return 0;
}
```



```c
// This function is the compare function used by the qsort()
int compare_int(const void *a, const void *b)
{
    return *((int *)a) - *((int *)b);
}

int main(){
    //create simple test case
    int seed = 0;
    srand(seed);    // set the seed
    int n = 10;
    int u[n];
    for (int i = 0; i < n; i++){
        u[i] = rand() % n;
    }
    //check the array before sorting
    printf("Before sorting\n");
    for (int i = 0; i < n; i++){
        printf("Array u[%d]=%d\n",i, u[i]);
    }
    ////void qsort(void *base, size_t nitems, size_t size, i
    qsort(u, n, sizeof(int), compare_int);
    return 0;
}
```

Step 2: Break – Using qSort

- Try printing to make sure it works…

```
9   // This function is the compare function used by the qsort()
10  int compare_int(const void *a, const void *b)
11  {
12      return *((int *)a) - *((int *)b);
13  }
14
15  int main(){
16      //create simple test case
17      int seed = 0;
18      srand(seed);    // set the seed
19      int n = 10;
20      int u[n];
21      for (int i = 0; i < n; i++){
22          u[i] = rand() % n;
23      }
24      //check the array before sorting
25      printf("Before sorting\n");
26      for (int i = 0; i < n; i++){
27          printf("Array u[%d]=%d\n",i, u[i]);
28      }
29      ////void qsort(void *base, size_t nitems, size_t size, ir
30      qsort(u, n, sizeof(int), compare_int);
31      printf("After sorting\n");
32      for (int i = 0; i < n; i++){
33          printf("Array u[%d]=%d\n",i, u[i]);
34      }
35      return 0;
36  }
```

```
kaleel@CentralCompute:~$ ./test
Before sorting
Array u[0]=3
Array u[1]=6
Array u[2]=7
Array u[3]=5
Array u[4]=3
Array u[5]=5
Array u[6]=6
Array u[7]=2
Array u[8]=9
Array u[9]=1
After sorting
Array u[0]=1
Array u[1]=2
Array u[2]=3
Array u[3]=3
Array u[4]=5
Array u[5]=5
Array u[6]=6
Array u[7]=6
Array u[8]=7
Array u[9]=9
```

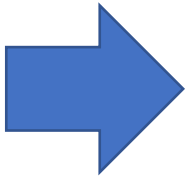
Now go back to our to do list…

1. We have to setup pipes for child A and child B.
2. We have to create child A and child B.
3. ~~We have to be able to use qSort.~~
4. We have to send data from the child A to pipeA (write).
5. We have to send data from child B to the pipeB (write).
6. Parent has to read from pipe A and pipe B (read).

I'm going to continue working in the blank code. Why? So I can debug fast…

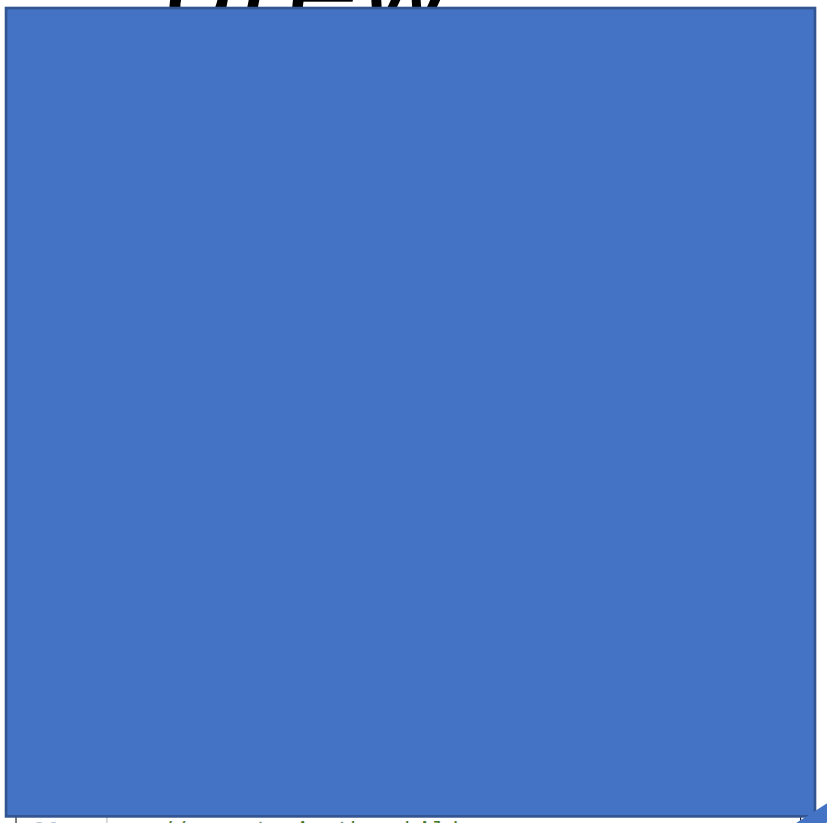
1. Copy the pipes from the starter code (keep same variable names)
2. Comment out the sorting (we'll use it later).
3. Call pipe and fork for the first child.

```c
int main(){
    //create simple test case
    int seed = 0;
    srand(seed);    // set the seed
    int n = 10;
    int u[n];
    for (int i = 0; i < n; i++){
        u[i] = rand() % n;
    }
    //check the array before sorting
    printf("Before sorting\n");
    for (int i = 0; i < n; i++){
        printf("Array u[%d]=%d\n",i, u[i]);
    }
    ////void qsort(void *base, size_t nitems, size_t size, ir
    qsort(u, n, sizeof(int), compare_int);
    printf("After sorting\n");
    for (int i = 0; i < n; i++){
        printf("Array u[%d]=%d\n",i, u[i]);
    }
    return 0;
}
```

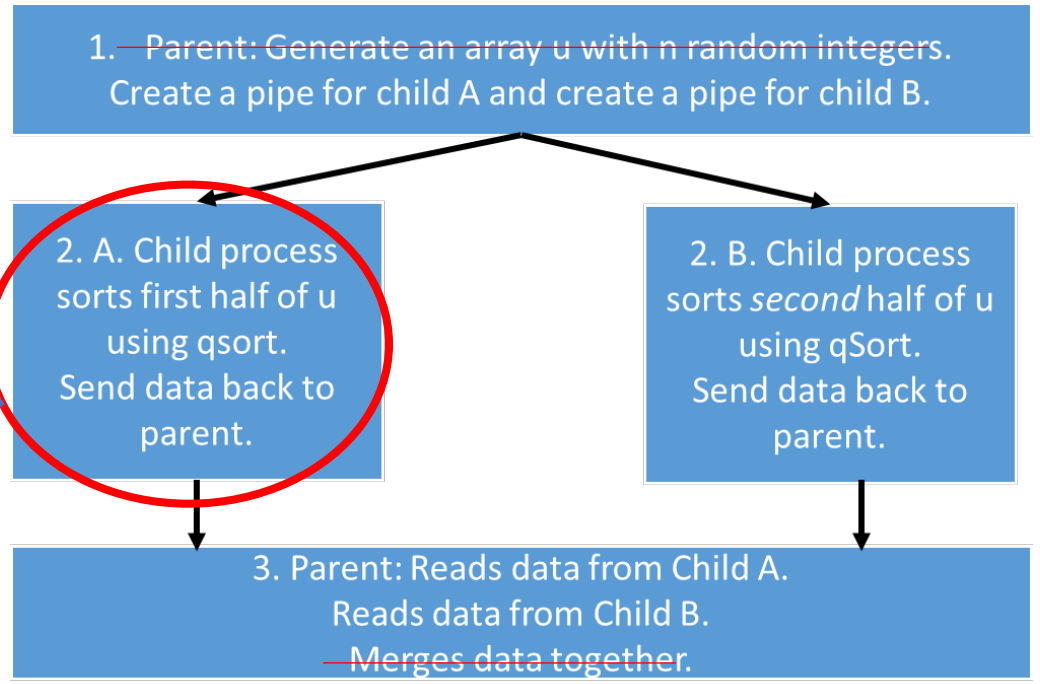


```c
int main(){
    //create simple test case
    int seed = 0;
    srand(seed);    // set the seed
    int n = 10;
    int u[n];
    for (int i = 0; i < n; i++){
        u[i] = rand() % n;
    }
    //check the array before sorting
    printf("Before sorting\n");
    for (int i = 0; i < n; i++){
        printf("Array u[%d]=%d\n",i, u[i]);
    }
    ////void qsort(void *base, size_t nitems, size_
    /*qsort(u, n, sizeof(int), compare_int);
    printf("After sorting\n");
    for (int i = 0; i < n; i++){
        printf("Array u[%d]=%d\n",i, u[i]);
    }*/
    //take the pipe variables from the starter code
    int pd1[2], pd2[2];
    pipe(pd1); //setup a pipe for child A
    pid_t childA = fork();
    //compute in the child process
    if(childA == 0){

    }
    return 0;
}
```

Now what does child A need to do? Let's look back at the diagram we drew



```
39      //compute in the child process
40      if(childA == 0){
41
42      }
43      return 0;
44  }
```



But we forgot to put in the splitting array from the starter code. So let's copy that in quickly…

From the starter code:

```
87      int u[n];
88      int *a, *b;
89      int half = n / 2;
90
91      a = u;
92      b = a + half;
```

We already know how to use qsort and we have half the list… so just put the two together:

```
15    int main(){
16        //create simple test case
17        int seed = 0;
18        srand(seed);    // set the seed
19        int n = 10;
20        int u[n];
21        int *a, *b;
22        int half = n / 2;
23        a = u;
24        b = a + half;
25        for (int i = 0; i < n; i++){
26            u[i] = rand() % n;
27        }
```

We already know how to use qsort and we have half the list… so just put the two together:

```
44      if(childA == 0){
45          //Child A only needs to sort half the array
46          //qsort(u, n, sizeof(int), compare_int)
47          qsort(a, half, sizeof(int), compare_int)
48      }
49      return 0;
```

But how do we know it actually worked?

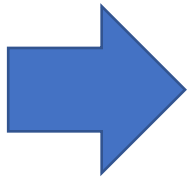
We already know how to use qsort and we have half the list… so just put the two together:

```
44      if(childA == 0){
45          //Child A only needs to sort half the array
46          //qsort(u, n, sizeof(int), compare_int)
47          qsort(a, half, sizeof(int), compare_int)
48      }
49      return 0;
```

```
44      if(childA == 0){
45          //Child A only needs to sort half the array
46          //qsort(u, n, sizeof(int), compare_int)
47          qsort(a, half, sizeof(int), compare_int);
48          //printing for debugging
49          printf("In child sorting\n");
50          for (int i = 0; i < half; i++){
51              printf("Array a[%d]=%d\n",i, a[i]);
52          }
```

We already know how to use qsort and we have half the list… so just put the two together:

```c
int main(){
    //create simple test case
    int seed = 0;
    srand(seed);    // set the seed
    int n = 10;
    int u[n];
    int *a, *b;
    int half = n / 2;
    a = u;
    b = a + half;
    for (int i = 0; i < n; i++){
        u[i] = rand() % n;
    }
    //check the array before sorting
    printf("Before sorting\n");
    for (int i = 0; i < n; i++){
        printf("Array u[%d]=%d\n",i, u[i]);
    }
    ////void qsort(void *base, size_t nitems, size_t s
    /*qsort(u, n, sizeof(int), compare_int);
    printf("After sorting\n");
    for (int i = 0; i < n; i++){
        printf("Array u[%d]=%d\n",i, u[i]);
    }*/
    //take the pipe variables from the starter code
    int pd1[2], pd2[2];
    pipe(pd1); //setup a pipe for child A
    pid_t childA = fork();
    //compute in the child process
    if(childA == 0){
        //Child A only needs to sort half the array
        //qsort(u, n, sizeof(int), compare_int)
        qsort(a, half, sizeof(int), compare_int);
        //printing for debugging
        printf("In child sorting\n");
        for (int i = 0; i < half; i++){
            printf("Array a[%d]=%d\n",i, a[i]);
        }
    }
    return 0;
}
```



```
kaleel@CentralCompute:~$ ./test
Before sorting
Array u[0]=3
Array u[1]=6
Array u[2]=7
Array u[3]=5
Array u[4]=3
Array u[5]=5
Array u[6]=6
Array u[7]=2
Array u[8]=9
Array u[9]=1
In child sorting
Array a[0]=3
Array a[1]=3
Array a[2]=5
Array a[3]=6
Array a[4]=7
```

NOTE: We didn’t properly close file descriptors yet…we’ll get to that.

Now go back to our to do list…

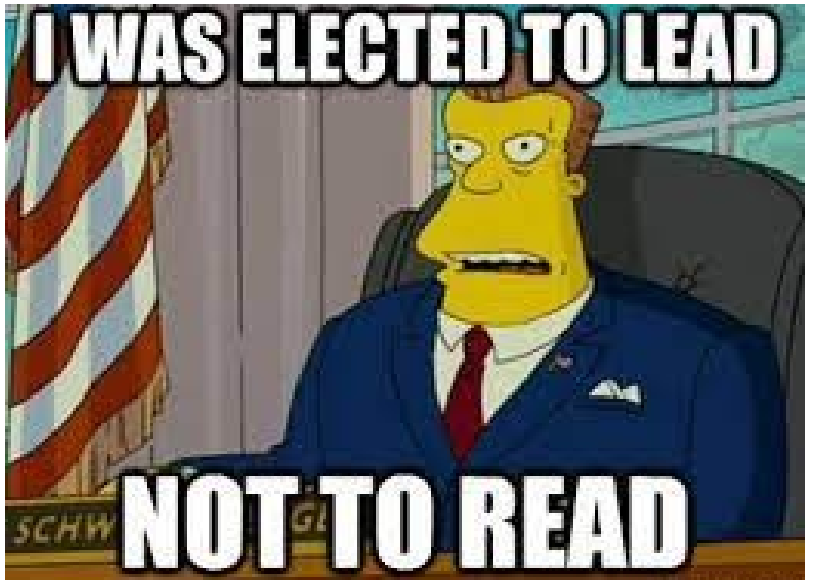
1. ~~We have to setup pipes for child A and child B.~~
2. ~~We have to create child A and child B.~~
3. ~~We have to be able to use qSort.~~
4. We have to send data from the child A to pipeA (write).
5. We have to send data from child B to the pipeB (write).
6. Parent has to read from pipe A and pipe B (read).

How do I write data using a pipe?

Given in the starter code:

```
65  void write_int(int pd, int value)
66  {
67      if (write(pd, &value, sizeof(int)) != sizeof(int))
68          die("write()");
69  }
70
71  //read an integer from a pipe
72  //the function returns the number of bytes read
73  int read_int(int pd, int *value)
74  {
75      return read(pd, value, sizeof(int));
76  }
```

But what if you didn't read the code carefully? Like me:



You can also do the write to pipe operation natively

- One important thing to note: Because we don't have a fixed size array for variable "a" we have to write one integer at a time. This is different than the slide example showed in class.

```
39      //take the pipe variables from the starter code
40      int pd1[2], pd2[2];
41      pipe(pd1); //setup a pipe for childA
42      pid_t childA = fork();
43      //compute in the child process
44      if(childA == 0){
45          //child A only needs to sort half the array
46          //qsort(u, n, sizeof(int), compare_int);
47          close(pd1[0]); //close the read end
48          qsort(a, half, sizeof(int), compare_int);
49          for(int i=0;i<half;i++){
50              write(pd1[1], &a[i], sizeof(a));
51          }
52          close(pd1[1]); //close the write end
53          return 0;
```

```
49          for(int i=0;i<half;i++){
50              write(pd1[1], &a[i], sizeof(a));
51          }
```

You can also do the write to pipe operation natively

- We now also handle the file descriptors properly in the child.
- We still need to handle them in the parent and other child later…

```
39      //take the pipe variables from the starter code
40      int pd1[2], pd2[2];
41      pipe(pd1); //setup a pipe for childA
42      pid_t childA = fork();
43      //compute in the child process
44      if(childA == 0){
45          //child A only needs to sort half the array
46          //qsort(u, n, sizeof(int), compare_int);
47          close(pd1[0]); //close the read end
48          qsort(a, half, sizeof(int), compare_int);
49          for(int i=0;i<half;i++){
50              write(pd1[1], &a[i], sizeof(a));
51          }
52          close(pd1[1]); //close the write end
53          return 0;
```

Close the read end of the pipe in the child.

AFTER writing, close the write end of the pipe.

Now go back to our to do list…

1. ~~We have to setup pipes for child A and child B.~~
2. ~~We have to create child A and child B.~~
3. ~~We have to be able to use qSort.~~
4. ~~We have to send data from the child A to pipeA (write).~~
5. We have to send data from child B to the pipeB (write).
6. Parent has to read from pipe A and pipe B (read).

You can also read from the pipe natively…

- One important thing to note: Because we don't have a fixed size array for variable "a" we have to READ one integer at a time. This is different than the slide example showed in class.

```
40      int pd1[2], pd2[2];
41      pipe(pd1); //setup a pipe for childA
42      pid_t childA = fork();
43      //compute in the child process
44      if(childA == 0){
```

```
56      //have the parent read from the pipe
57      close(pd1[1]); //close the write end of the pipe
58      for(int i=0; i<half;i++){
59          read(pd1[0], &a[i], sizeof(a));
60          printf("In the parent a[%d]=%d\n", i, a[i]);
61      }
62      close(pd1[0]); //close the read of the pipe
63      return 0;
64  }
```

```
56      //have the parent read from the pipe
57      close(pd1[1]); //close the write end of the pipe
58      for(int i=0; i<half;i++){
59          read(pd1[0], &a[i], sizeof(a));
60          printf("In the parent a[%d]=%d\n", i, a[i]);
61      }
62      close(pd1[0]); //close the read of the pipe
63      return 0;
64  }
```

You can also read from the pipe natively…

- Don't forget to close the file descriptors when they are not being used.

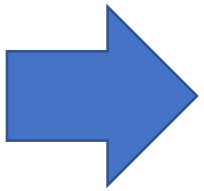
```
40      int pd1[2], pd2[2];
41      pipe(pd1); //setup a pipe for childA
42      pid_t childA = fork();
43      //compute in the child process
44      if(childA == 0){
56      //have the parent read from the pipe
57      close(pd1[1]); //close the write end of the pipe
58      for(int i=0; i<half;i++){
59          read(pd1[0], &a[i], sizeof(a));
60          printf("In the parent a[%d]=%d\n", i, a[i]);
61      }
62      close(pd1[0]); //close the read of the pipe
63      return 0;
64  }
```

Close the write end of the pipe in the parent.

AFTER reading, close the read end of the pipe.

Note: The use of print statements for debugging

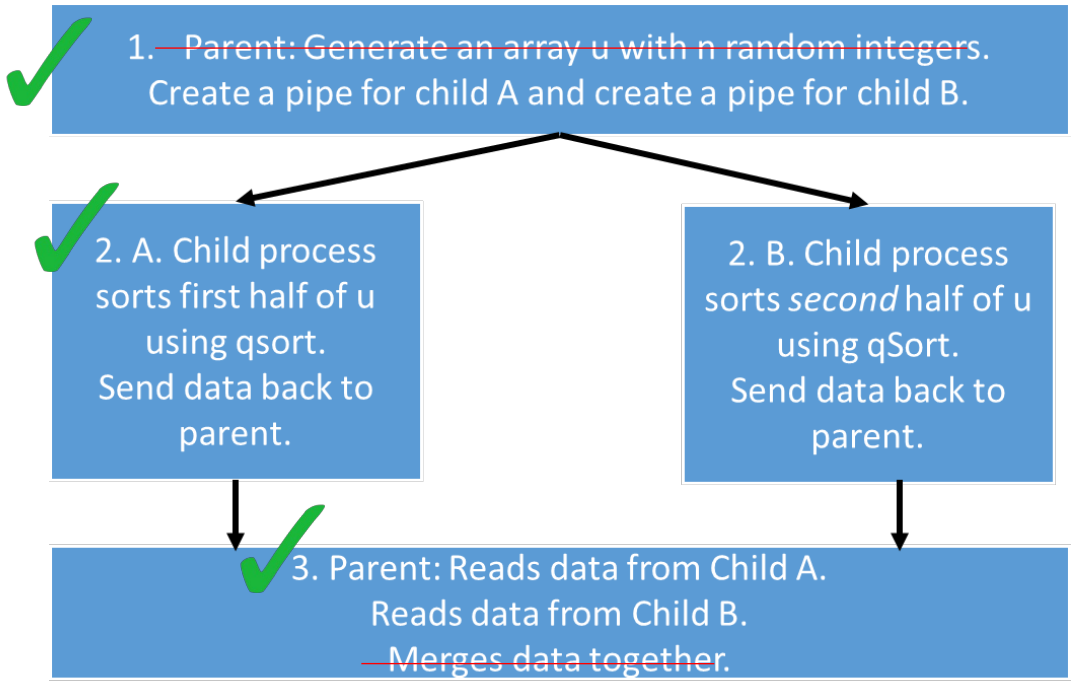
```
40      int pd1[2], pd2[2];
41      pipe(pd1); //setup a pipe for childA
42      pid_t childA = fork();
43      //compute in the child process
44      if(childA == 0){
45          //child A only needs to sort half the array
46          //qsort(u, n, sizeof(int), compare_int);
47          close(pd1[0]); //close the read end
48          qsort(a, half, sizeof(int), compare_int);
49          for(int i=0;i<half;i++){
50              write(pd1[1], &a[i], sizeof(a));
51          }
52          close(pd1[1]); //close the write end
53          return 0;
54          printf("I should not see this\n");
55      }
56      //have the parent read from the pipe
57      close(pd1[1]); //close the write end of the pipe
58      for(int i=0; i<half;i++){
59          read(pd1[0], &a[i], sizeof(a));
60          printf("In the parent a[%d]=%d\n", i, a[i]);
61      }
62      close(pd1[0]); //close the read of the pipe
63      return 0;
64  }
```



```
kaleel@CentralCompute:~$ ./test
Before sorting
Array u[0]=3
Array u[1]=6
Array u[2]=7
Array u[3]=5
Array u[4]=3
Array u[5]=5
Array u[6]=6
Array u[7]=2
Array u[8]=9
Array u[9]=1
In the parent a[0]=3
In the parent a[1]=3
In the parent a[2]=5
In the parent a[3]=6
In the parent a[4]=7
```

Let's take a step back and see how much we've done…

1. ~~We have to setup pipes for child A and child B.~~
2. ~~We have to create child A and child B.~~
3. ~~We have to be able to use qSort.~~
4. ~~We have to send data from the child A to pipeA (write).~~
5. We have to send data from child B to the pipeB (write).
6. ~~Parent has to read from pipe A and pipe B (read).~~

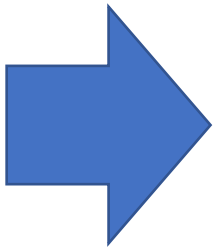


Question: How will we build pipe B?

Copy and paste!

What code do we need to copy? (and change)

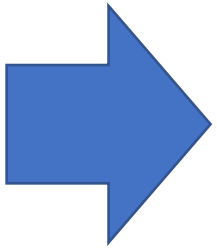
```
55      pipe(pd1); //setup a pipe for childA
56      pid_t childA = fork();
57      //compute in the child process
58      if(childA == 0){
59          close(pd1[0]);
60          qsort(a, half, sizeof(int), compare_int);
61          for(int i=0;i<half;i++){
62              write(pd1[1], &a[i], sizeof(a));
63          }
64          close(pd1[1]);
65          return 0;
66          printf("I should not see this\n");
67      }
68
69      //read in the parent
70      close(pd1[1]);
71      for(int i=0;i<half;i++){
72          read(pd1[0], &a[i], sizeof(a));
73      }
74      close(pd1[0]);
```



```
76      pipe(pd2); //setup a pipe for childB
77      pid_t childB = fork();
78      //compute in the child process
79      if(childB == 0){
80          close(pd2[0]);
81          qsort(b, half, sizeof(int), compare_int);
82          for(int i=0;i<half;i++){
83              write(pd2[1], &b[i], sizeof(b));
84          }
85          close(pd2[1]);
86          return 0;
87          printf("I should not see this\n");
88      }
89
90    //read in the parent
91      close(pd2[1]);
92      for(int i=0;i<half;i++){
93          read(pd2[0], &b[i], sizeof(b));
94      }
95      close(pd2[0]);
```

Test with print statements

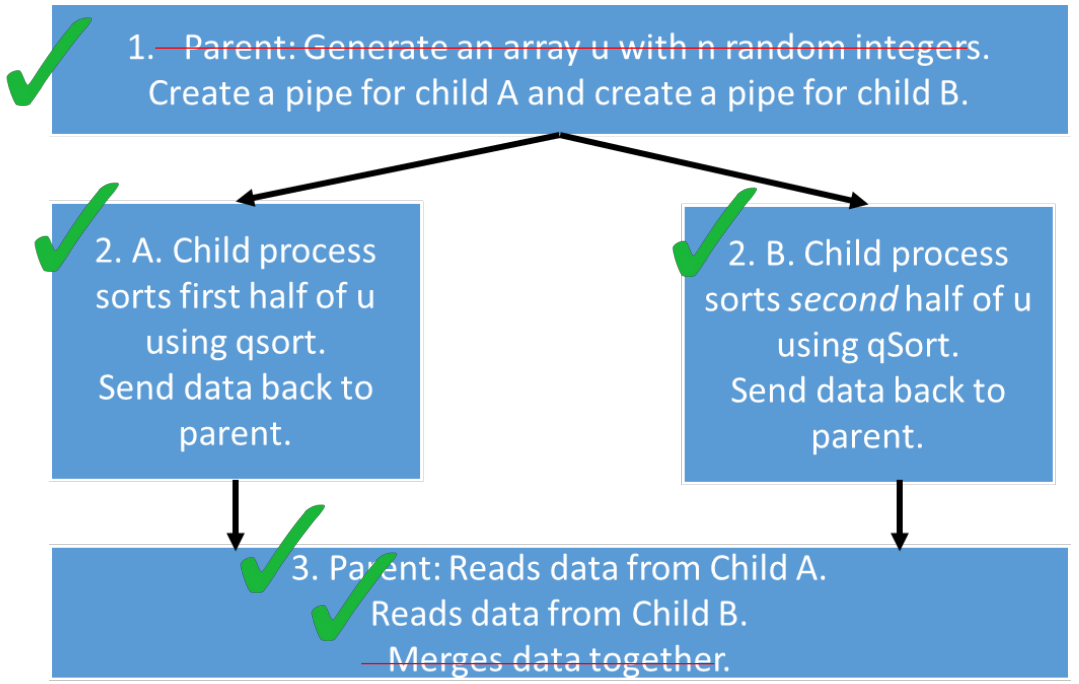
```
77      pipe(pd2); //setup a pipe for childB
78      pid_t childB = fork();
79      //compute in the child process
80      if(childB == 0){
81          close(pd2[0]);
82          qsort(b, half, sizeof(int), compare_int);
83          for(int i=0;i<half;i++){
84              write(pd2[1], &b[i], sizeof(b));
85          }
86          close(pd2[1]);
87          return 0;
88          printf("I should not see this\n");
89      }
90
91     //read in the parent
92      close(pd2[1]);
93      for(int i=0;i<half;i++){
94          read(pd2[0], &b[i], sizeof(b));
95          printf("In the parent b[%d]=%d\n", i, b[i]);
96      }
97      close(pd2[0]);
```



```
kaleel@CentralCompute:~$ ./test
Before sorting
Array u[0]=3
Array u[1]=6
Array u[2]=7
Array u[3]=5
Array u[4]=3
Array u[5]=5
Array u[6]=6
Array u[7]=2
Array u[8]=9
Array u[9]=1
In parent, a[0]=3
In parent, a[1]=3
In parent, a[2]=5
In parent, a[3]=6
In parent, a[4]=7
In parent, b[0]=1
In parent, b[1]=2
In parent, b[2]=5
In parent, b[3]=6
In parent, b[4]=9
```

Let's take a step back and see how much we've done…

1. ~~We have to setup pipes for child A and child B.~~
2. ~~We have to create child A and child B.~~
3. ~~We have to be able to use qSort.~~
4. ~~We have to send data from the child A to pipeA (write).~~
5. ~~We have to send data from child B to the pipeB (write).~~
6. ~~Parent has to read from pipe A and pipe B (read).~~



What is next? Put it back into the starter code and get an A!

The Big Picture (1)

- The entire code is hard to show on one slide. But here is most of it…

Variable Initialization

```
37  int main(){
38      //create simple test case
39      int seed = 0;
40      srand(seed);    // set the seed
41      int n = 10;
42      int u[n];
43      int *a, *b;
44      int half = n / 2;
45      a = u;
46      b = a + half;
47      for (int i = 0; i < n; i++){
48          u[i] = rand() % n;
49      }
50      //check the array before sorting
51      printf("Before sorting\n");
52      for (int i = 0; i < n; i++){
53          printf("Array u[%d]=%d\n",i, u[i]);
54      }
```

Sorting in Child A

```
55      pipe(pd1); //setup a pipe for childA
56      pid_t childA = fork();
57      //compute in the child process
58      if(childA == 0){
59          close(pd1[0]);
60          qsort(a, half, sizeof(int), compare_int);
61          for(int i=0;i<half;i++){
62              write(pd1[1], &a[i], sizeof(a));
63          }
64          close(pd1[1]);
65          return 0;
66          printf("I should not see this\n");
67      }
```

The Big Picture (2)

Sorting in Child B

```
76      pipe(pd2); //setup a pipe for childB
77      pid_t childB = fork();
78      //compute in the child process
79      if(childB == 0){
80          close(pd2[0]);
81          qsort(b, half, sizeof(int), compare_int);
82          for(int i=0;i<half;i++){
83              write(pd2[1], &b[i], sizeof(b));
84          }
85          close(pd2[1]);
86          return 0;
87          printf("I should not see this\n");
88      }
```

Read in the parent, combine the two arrays

```
90      //read in the parent
91       close(pd2[1]);
92       for(int i=0;i<half;i++){
93           read(pd2[0], &b[i], sizeof(b));
94       }
95       close(pd2[0]);
96       //combine the lists
97       int sorted[n];
98       merge(a, b, sorted, half);
99       printf("After sorting\n");
100      for(int i=0;i<n;i++){
101          printf("Array u[%d]=%d\n", i, sorted[i]);
102      }
103      return 0;
```

Recall: My Approach To Coding Hard Problems

1. Sketch = Write down the steps needed to do the algorithm/task.
2. Break = Break it into smaller pieces and solve each piece.
3. Build = Connect the pieces together one at a time.

We've shown steps 1 and 2, hopefully step 3 should be very clear as our code pieces use the SAME variable names as the starter code, so it is a simple matter of cutting and pasting into the original.

My solution code will be posted later today on HuskyCT.

A few more things to consider:

Was this the ONLY way to write the sorting code?

My original code (ugly):

```
50      //check the array before sorting
51      printf("Before sorting\n");
52      for (int i = 0; i < n; i++){
53          printf("Array u[%d]=%d\n",i, u[i]);
54      }
55      ////void qsort(void *base, size_t nitems, size_t size
56      /*qsort(u, n, sizeof(int), compare_int);
57      printf("After sorting\n");
58      for (int i = 0; i < n; i++){
59          printf("Array u[%d]=%d\n",i, u[i]);
60      }*/
61      //take the pipe variables from the starter code
62      int pd1[2], pd2[2];
63      pipe(pd1); //setup a pipe for child A
64      pid_t childA = fork();
65      //compute in the child process
66      if(childA == 0){
67          //Child A only needs to sort half the array
68          //qsort(u, n, sizeof(int), compare_int)
69          qsort(a, half, sizeof(int), compare_int);
70          //write to the pipe
71          for(int i = 0 ; i< half; i++){
72              write(pd1[1], &a[i], sizeof(a));
73          }
74          close(pd1[1]);
75          return 0;
76          printf("I should not see this\n");
77          //printing for debugging
```

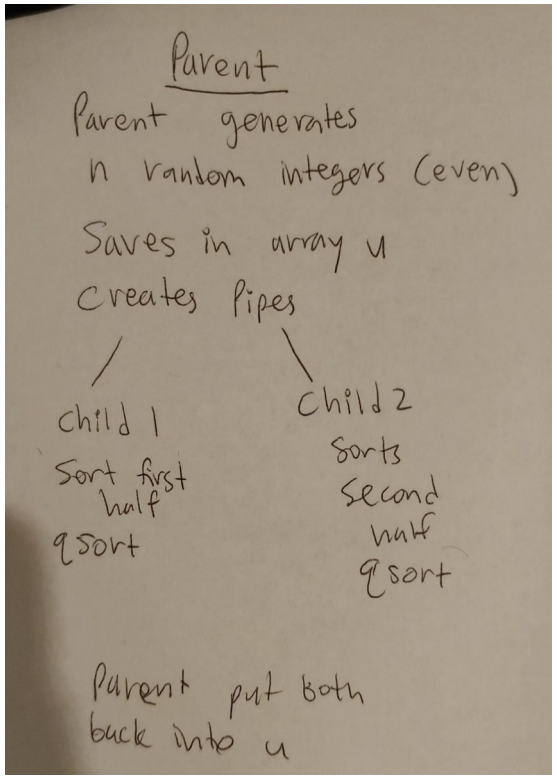
Solution code (pretty):

```
112     //_TMPL_ CUT
113     pid_t pid1 = fork();
114     if (pid1 == 0)
115     {
116         //child does not read from pd1, does not use pd2
117         close(pd1[0]);
118         close(pd2[0]);
119         close(pd2[1]);
120
121         //sort a[] and write to the pipe
122         qsort(a, half, sizeof(int), compare_int);
123         for(int i=0; i<half; i++)
124             write_int(pd1[1], a[i]);
125         //close this pipe after writing
126         close(pd1[1]);
127         exit(0);
128     }
129     //parent does not write to this pipe
130     close(pd1[1]);
131
132     pid_t pid2 = fork();
133     if (pid2 == 0)
134     {
135         // close read end of both pipes
136         close(pd1[0]);
137         close(pd2[0]);
138
139         //sort b[] here and write to the pipe
140         qsort(b, half, sizeof(int), compare_int);
141         for(int i=0; i<half; i++)
```

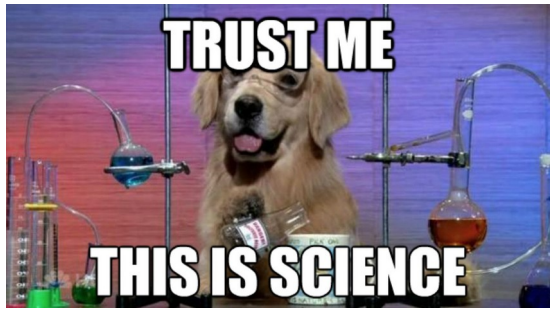
A few more things to consider:

Are the starting coding approaches always as clean as shown in the slides?

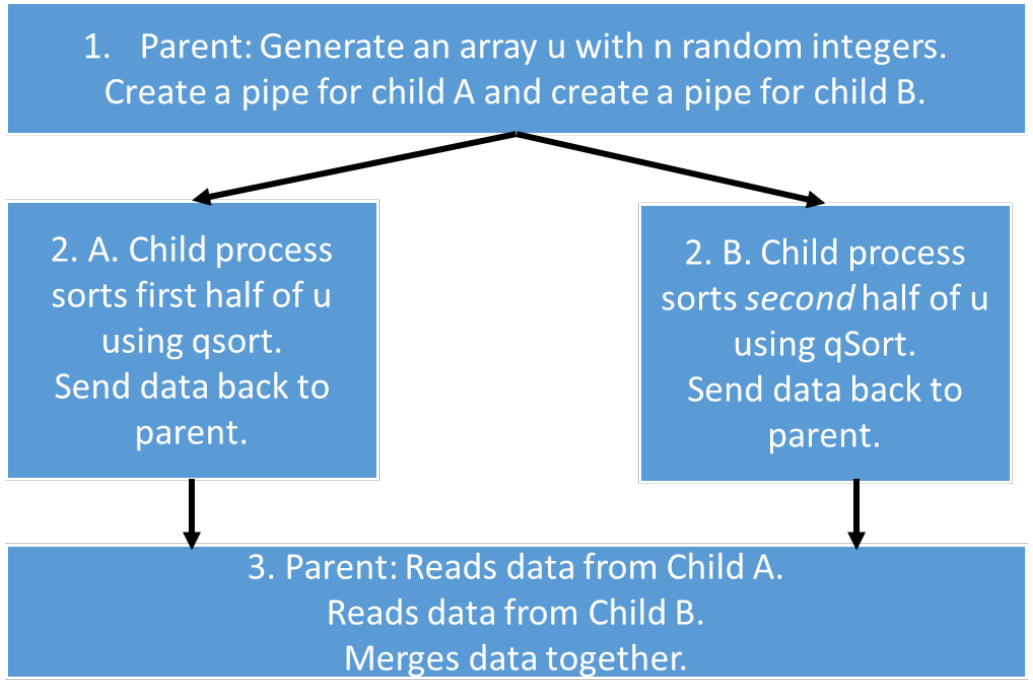
How I brainstorm:



My brain:



How the ideas are presented in slides:



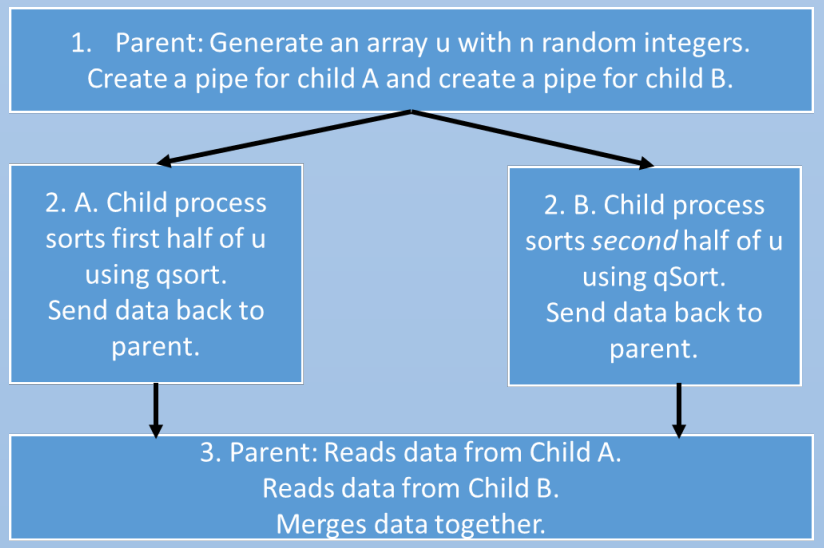
Last thing to consider:

What happens if you don't understand

or don't like my way of thinking?

Answer: You Fail (kidding!)

1. Sketch = Write down the steps needed to do the algorithm/task.
2. Break = Break it into smaller pieces and solve each piece.
3. Build = Connect the pieces together one at a time.



1. We have to setup pipes for child A and child B.
2. We have to create child A and child B.
3. We have to be able to use qSort.
4. We have to send data from the child A to pipeA (write).
5. We have to send data from child B to the pipeB (write).
6. Parent has to read from pipe A and pipe B (read).

Figure Sources

1. https://i.pinimg.com/originals/fa/9c/af/fa9cafecc650057373f0dc7d86cbcfa1.jpg
2. https://www.usnews.com/dims4/USNEWS/7f6a559/2147483647/thumbnail/970x647/quality/85/?url=https%3A%2F%2Fwww.usnews.com%2Fcmsmedia%2Fa1%2F96%2F7779721e4babbdcc0ac8628ac185%2F170526-jfk-editorial.jpg
3. https://i.imgflip.com/nqdi5.jpg
4. https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcSQzCFCiGzwxfpVUK4Q3s93O-5y5iYkFpG0dDJZOJMza9Lgq5O8Y-_fnDtfGBU-EJSmcJI&usqp=CAU
5. https://upload.wikimedia.org/wikipedia/commons/thumb/5/51/Green-checkmark.svg/1957px-Green-checkmark.svg.png