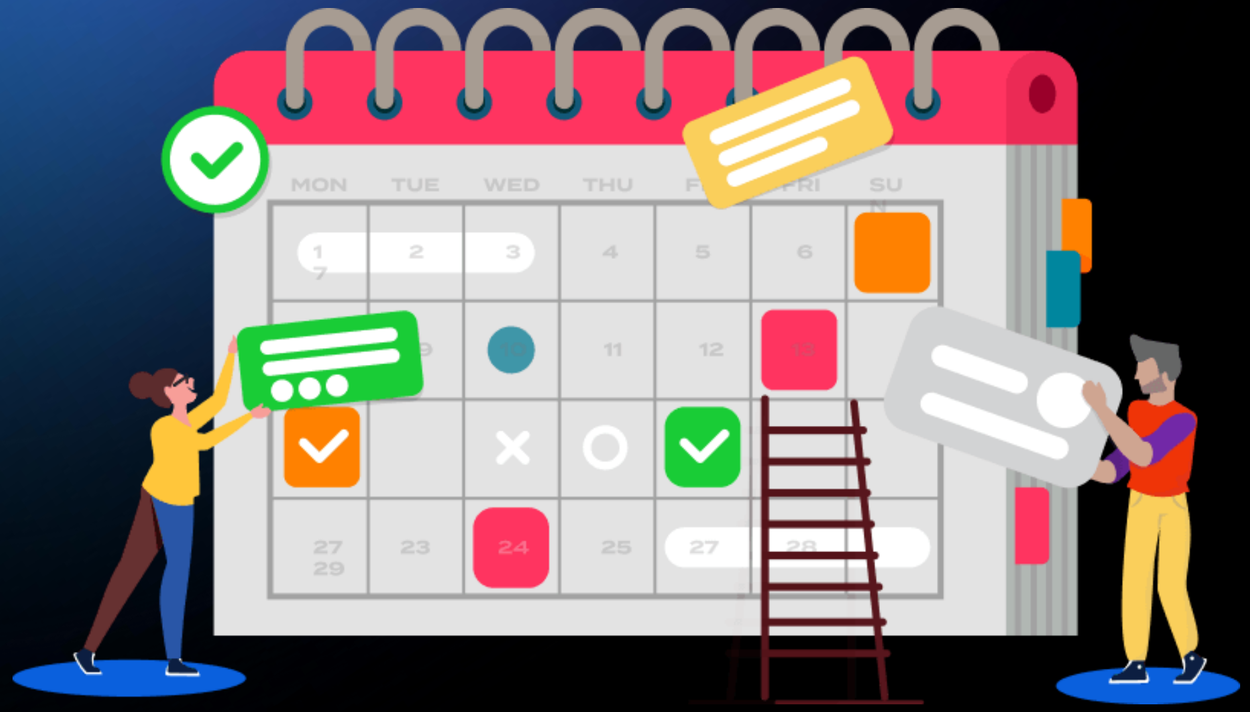
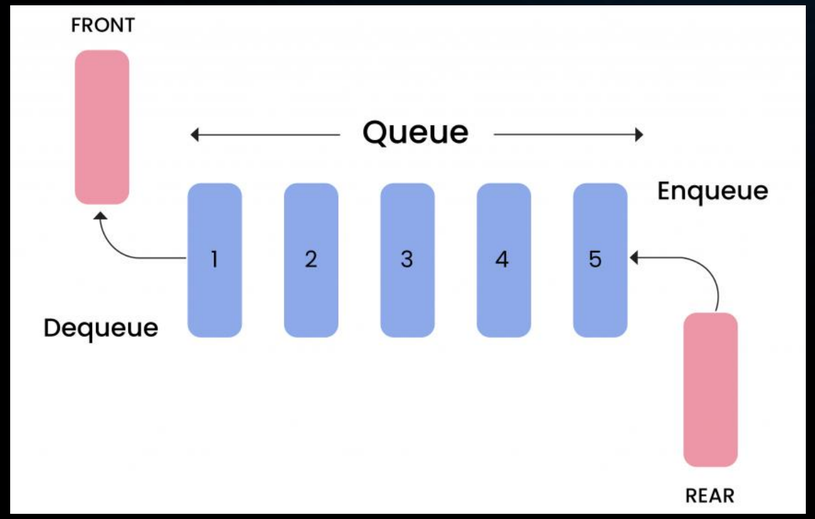
FIFO Quantum to SJF

Priya Jeyaprakash, Xinyi Li, Zion Jones, Nick Garcia

CPU Schedulers

- CPU Schedulers Manage Processes that need to run on the CPU
- Good practices for a scheduler are :
  - Flexibility in CPU Occupation
  - Fairness
  - Increased Performance





FIFO

- First in First Out
  - Earliest arriving job will execute first, with the subsequent jobs running in the order in which they arrived.
  - Once a job is started it will complete before CPU is used by another scheduler

findNextJob()

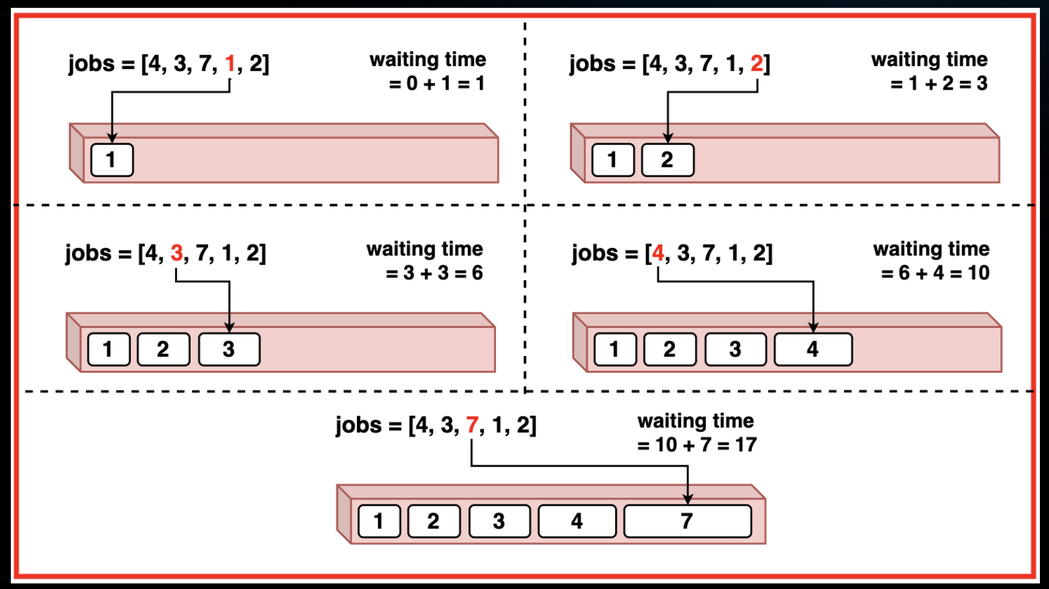
Function to be used in FIFO to find what job is the next in the arrival queue

```
int findNextJob(struct threads* jobs, int numjobs) {
    int minIndex = -1;
    int minTime = 10000000;

    for (int i = 0; i < numjobs; i++) {
        if (!jobs[i].completion_time && jobs[i].arrival_time < minTime) {
            minTime = jobs[i].arrival_time;
            minIndex = i;
        }
    }
    return minIndex;
}
```

FIFO()

FIFO function to be applied onto existing job queue to save CPU processing data

```
void fifo(struct threads* jobs, int numjobs) {
    int time = 0;
    int doneCount = 0;

    while (doneCount < numjobs) {
        int next = findNextJob(jobs, numjobs);
        if (next == -1) break;

        if (time < jobs[next].arrival_time)
            time = jobs[next].arrival_time;

        jobs[next].first_run = time;
        jobs[next].response_time = jobs[next].first_run - jobs[next].arrival_time;

        time += jobs[next].burst_time;

        jobs[next].completion_time = time;
        jobs[next].turnaround_time = time - jobs[next].arrival_time;

        doneCount++;
    }

}
```



SJF

- Shortest Job First
  - Will pick the shortest arrived job at time of scheduling and execute to completion

bubble()

Function used by SJF to sort all incoming jobs so it goes in order of shortest to longest using bubblesort algorithm.

swap()

Helper function used in bubble() to implement the bubblesort sorting algorithm.

Group Project > CSE-OS-Fall-2025 > CSE5305 > xinyi-priya-zion-nick > C SJF.c > ...

```c
#include "thread.h"

// Implement sorting alg (buble sort)
void swap(struct thread* a, struct thread* b) {
    struct thread temp = *a;
    *a = *b;
    *b = temp;
}

void bubble(struct thread* jobs, int num_jobs) {
    for (int i = 0; i < num_jobs - 1; i++) {
        for (int j = 0; j < num_jobs - i - 1; j++) {
            // sort the jobs by their burst time
            if (jobs[j].burst > jobs[j+ 1].burst) {swap(&jobs[j], &jobs[j + 1]);}
        }
    }
}
```

sjf_scheduler()

SJF function to be applied onto existing job queue to save CPU processing data
(e.g. turnaround and wait time)

Group Project > CSE-OS-Fall-2025 > CSE5305 > xinyi-priya-zion-nick > C SJF.c > ...

```
18
19  // sort shortest jobs (burst time) using BubbleSort
20  void sjf_scheduler(struct thread *jobs, int num_jobs) {
21      int current_time = 0;
22
23      bubble(jobs, num_jobs);
24      for (int i = 0; i < num_jobs; i++) {
25          current_time += jobs[i].burst;
26          //adding turnaround time calculation
27          jobs[i].turnaround = current_time - jobs[i].arrival;
28          jobs[i].wait = jobs[i].turnaround - jobs[i].burst;
29      }
30  }
```

Pros vs. Cons

- FIFO
  - Pros
    - Easy to understand
    - Simple to implement
    - No starvation
    - No context switching time
  - Cons
    - Can have long wait time for big job first
- SJF
  - Pros
    - Low average turn around time
  - Cons
    - Big jobs take a very long time to turn around
    - Big jobs can be starved
    - Low predictability

Quantum FIFO to SJF

- A **quantum** is a fixed time slice given to a CPU scheduler. In all, it is the maximum continuous time a process or job will be allowed to run before the scheduler pauses it and switches to another process. In our case, the quantum represents the time FIFO is allowed to run before switching to SJF

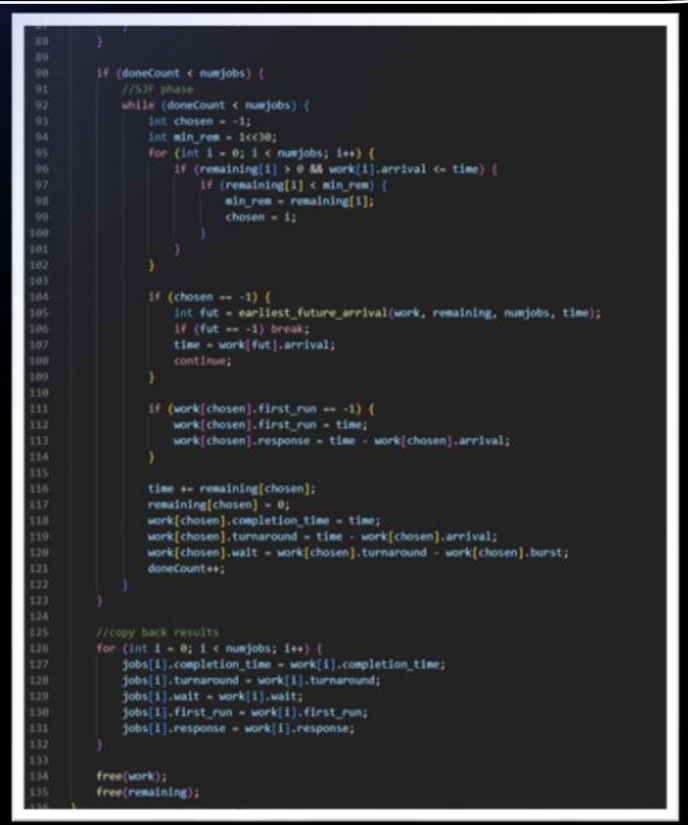
Quantum.c
A deep dive

This is our ***hybrid_quantum()*** function, which is responsible for implementing and maintaining our hybrid scheduler where we...

1. Start with FIFO in FIFO.c
2. Run until a fixed quantum specified expires
3. Then switch to SJF in SJF.c for the remaining jobs

```
    if (numjobs <= 0) return;

    struct thread *work = malloc(numjobs * sizeof(struct thread));
    int *remaining = malloc(numjobs * sizeof(int));
    if (!work || !remaining) {
        fprintf(stderr, "Memory allocation failed\n");
        free(work);
        free(remaining);
        return;
    }

    for (int i = 0; i < numjobs; i++) {
        work[i] = jobs[i];
        remaining[i] = jobs[i].burst;
        work[i].completion_time = 0;
        work[i].turnaround = 0;
        work[i].wait = 0;
        work[i].first_run = -1;
        work[i].response = 0;
    }

    int time = 0;
    int doneCount = 0;

    //FIFO phase
    while (time < quantum && doneCount < numjobs) {
        int next = find_next_fifo(work, remaining, numjobs, time);
        if (next == -1) {
            int fut = earliest_future_arrival(work, remaining, numjobs, time);
            if (fut == -1) break;
            time = (work[fut].arrival >= quantum) ? quantum : work[fut].arrival;
            continue;
        }

        if (work[next].first_run == -1) {
            work[next].first_run = time;
            work[next].response = work[next].first_run - work[next].arrival;
        }

        int time_available = quantum - time;
        if (remaining[next] <= time_available) {
            time += remaining[next];
            remaining[next] = 0;
            work[next].completion_time = time;
            work[next].turnaround = time - work[next].arrival;
            work[next].wait = work[next].turnaround - work[next].burst;
            doneCount++;
        } else {
            remaining[next] -= time_available;
            time += time_available;
```

This system handles the task of ensuring early jobs are handled predictably in an arrival-order fashion, then when the time window closes, the system will switch to shortest burst time to ensure optimization



```
    }

    if (doneCount < numjobs) {
        //SJF phase
        while (doneCount < numjobs) {
            int chosen = -1;
            int min_rem = 1<<30;
            for (int i = 0; i < numjobs; i++) {
                if (remaining[i] > 0 && work[i].arrival <= time) {
                    if (remaining[i] < min_rem) {
                        min_rem = remaining[i];
                        chosen = i;
                    }
                }
            }

            if (chosen == -1) {
                int fut = earliest_future_arrival(work, remaining, numjobs, time);
                if (fut == -1) break;
                time = work[fut].arrival;
                continue;
            }

            if (work[chosen].first_run == -1) {
                work[chosen].first_run = time;
                work[chosen].response = time - work[chosen].arrival;
            }

            time += remaining[chosen];
            remaining[chosen] = 0;
            work[chosen].completion_time = time;
            work[chosen].turnaround = time - work[chosen].arrival;
            work[chosen].wait = work[chosen].turnaround - work[chosen].burst;
            doneCount++;
        }
    }

    //copy back results
    for (int i = 0; i < numjobs; i++) {
        jobs[i].completion_time = work[i].completion_time;
        jobs[i].turnaround = work[i].turnaround;
        jobs[i].wait = work[i].wait;
        jobs[i].first_run = work[i].first_run;
        jobs[i].response = work[i].response;
    }

    free(work);
    free(remaining);
```

Quantum.c
A deep dive

For our **data structures** within quantum.c we have the following...

- **work[]**: a local copy of all jobs so the original input isn't modified or corrupted during the process
- **remaining[]**: keeps track of unfinished burst time of each job

```
47      for (int i = 0; i < numjobs; i++) {
48          work[i] = jobs[i];
49          remaining[i] = jobs[i].burst;
50          work[i].completion_time = 0;
51          work[i].turnaround = 0;
52          work[i].wait = 0;
53          work[i].first_run = -1;
54          work[i].response = 0;
55      }
```

```
125     //copy back results
126     for (int i = 0; i < numjobs; i++) {
127         jobs[i].completion_time = work[i].completion_time;
128         jobs[i].turnaround = work[i].turnaround;
129         jobs[i].wait = work[i].wait;
130         jobs[i].first_run = work[i].first_run;
131         jobs[i].response = work[i].response;
132     }
```

Quantum.c
A deep dive

This helper function ***find_next_fifo()*** is used to find the earliest arrival job that has arrived before the current time and is unfinished. This essentially allows the limited FIFO phase to behave like a ready queue processed in arrival order.

earliest_future_arrival() is used for when no job has arrived yet and we need to jump the clock forward to the next arrival. It also prevents CPU idling and correctly simulates scheduler behavior.

```c
#include <stdio.h>
#include <stdlib.h>
#include "thread.h"

//helper: find next FIFO job that has arrived and is unfinished
static int find_next_fifo(struct thread *work, int *remaining, int numjobs, int current_time) {
    int idx = -1;
    int min_arrival = 1<<30;
    for (int i = 0; i < numjobs; i++) {
        if (remaining[i] > 0 && work[i].arrival <= current_time) {
            if (work[i].arrival < min_arrival) {
                min_arrival = work[i].arrival;
                idx = i;
            }
        }
    }
    return idx;
}
```

```c
//helper: find earliest future arrival among unfinished jobs
static int earliest_future_arrival(struct thread *work, int *remaining, int numjobs, int current_time) {
    int idx = -1;
    int min_arrival = 1<<30;
    for (int i = 0; i < numjobs; i++) {
        if (remaining[i] > 0 && work[i].arrival > current_time) {
            if (work[i].arrival < min_arrival) {
                min_arrival = work[i].arrival;
                idx = i;
            }
        }
    }
    return idx;
}
```

Quantum.c
A deep dive

In **the FIFO phase**, behavior consists of running only until time < quantum

and in each loop we..

1. pick the next FIFO job with the helper function *find_next_fifo()*
2. then jump to next arrival if none arrived yet
3. compute response if this is the first time running
4. then compute the amount of time left in the quantum
5. lastly, either the job finishes before the quantum or the quantum ends and we must break and switch to SJF

```
61  while (time < quantum && doneCount < numjobs) {
62      int next = find_next_fifo(work, remaining, numjobs, time);
63      if (next == -1) {
64          int fut = earliest_future_arrival(work, remaining, numjobs, time);
65          if (fut == -1) break;
66          time = (work[fut].arrival >= quantum) ? quantum : work[fut].arrival;
67          continue;
68      }
69
70      if (work[next].first_run == -1) {
71          work[next].first_run = time;
72          work[next].response = work[next].first_run - work[next].arrival;
73      }
74
75      int time_available = quantum - time;
76      if (remaining[next] <= time_available) {
77          time += remaining[next];
78          remaining[next] = 0;
79          work[next].completion_time = time;
80          work[next].turnaround = time - work[next].arrival;
81          work[next].wait = work[next].turnaround - work[next].burst;
82          doneCount++;
83      } else {
84          remaining[next] -= time_available;
85          time += time_available;
86          break;
87      }
88  }
```

Quantum.c
A deep dive

In **the SJF phase**, behavior consists of running only for the work that remains after the quantum ends

and in each loop we..

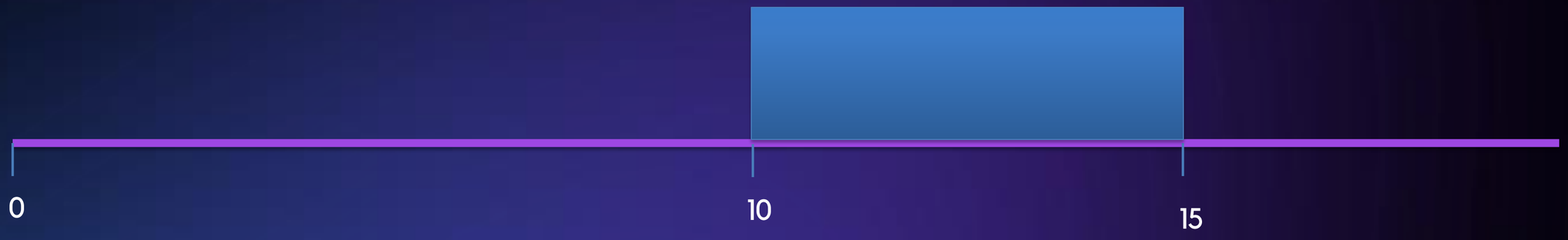
1. choose the job with the smallest *remaining[]* time
2. jump to next arrival if nothing is available
3. compute response if this is the first run
4. run job to completion
5. then update metrics

```
    if (doneCount < numjobs) {
        //SJF phase
        while (doneCount < numjobs) {
            int chosen = -1;
            int min_rem = 1<<30;
            for (int i = 0; i < numjobs; i++) {
                if (remaining[i] > 0 && work[i].arrival <= time) {
                    if (remaining[i] < min_rem) {
                        min_rem = remaining[i];
                        chosen = i;
                    }
                }
            }

            if (chosen == -1) {
                int fut = earliest_future_arrival(work, remaining, numjobs, time);
                if (fut == -1) break;
                time = work[fut].arrival;
                continue;
            }

            if (work[chosen].first_run == -1) {
                work[chosen].first_run = time;
                work[chosen].response = time - work[chosen].arrival;
            }

            time += remaining[chosen];
            remaining[chosen] = 0;
            work[chosen].completion_time = time;
            work[chosen].turnaround = time - work[chosen].arrival;
            work[chosen].wait = work[chosen].turnaround - work[chosen].burst;
            doneCount++;
        }
    }
```

Turnaround Time

Time from arrival to completion

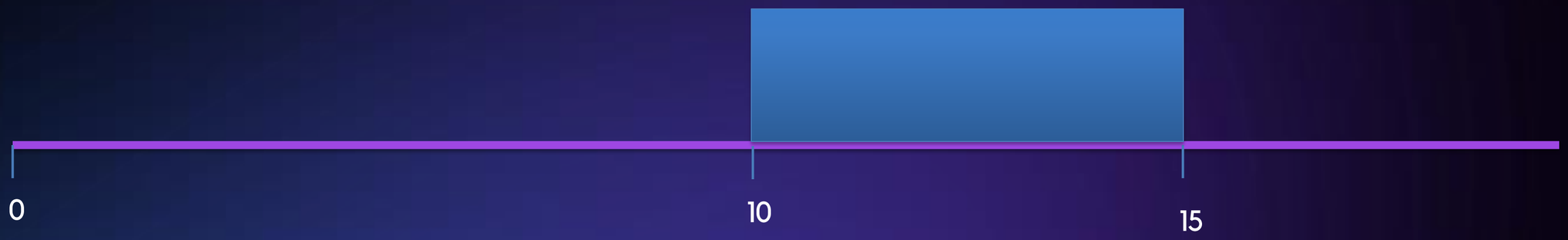
- Let's say a job arrives at 0ms starts on the CPU at 10ms and completes execution at 15ms.
  - Turnaround Time = 15ms - 0ms = 15ms



Response Time

Time from arrival of job to start on the CPU

- Let's say a job arrives at 0ms starts on the CPU at 10ms and completes execution at 15ms.
  - Response Time= 10ms - 0ms = 10ms



Comparisons

This code gives us the average of wait time, turnaround time, and response time that we see used from the structs that are being used for the schedulers we've implemented. We used this function after calling test cases for our schedulers.

```c
#include <stdio.h>
#include "thread.h"
void print_comparison_table( struct thread* fifo, struct thread* sjf, struct thread* hybrid, int n)
{
    double fifo_wait = 0, fifo_turn = 0, fifo_resp = 0;
    double sjf_wait = 0, sjf_turn = 0, sjf_resp = 0;
    double hyb_wait = 0, hyb_turn = 0, hyb_resp = 0;

    for (int i = 0; i < n; i++) {
        fifo_wait += fifo[i].wait;
        fifo_turn += fifo[i].turnaround;
        fifo_resp += fifo[i].response;

        sjf_wait += sjf[i].wait;
        sjf_turn += sjf[i].turnaround;
        sjf_resp += sjf[i].response;

        hyb_wait += hybrid[i].wait;
        hyb_turn += hybrid[i].turnaround;
        hyb_resp += hybrid[i].response;
    }

    printf("Scheduler    | Avg Wait | Avg Turnaround | Avg Response\n");
    printf("=============================================================\n");
    printf("FIFO         | %8.2f | %14.2f | %12.2f\n", fifo_wait/n, fifo_turn/n, fifo_resp/n);
    printf("SJF          | %8.2f | %14.2f | %12.2f\n", sjf_wait/(n), sjf_turn/(n), sjf_resp/n);
    printf("Hybrid       | %8.2f | %14.2f | %12.2f\n", hyb_wait/(n), hyb_turn/(n), hyb_resp/(n));
    printf("=============================================================\n\n");
}
```

Key Takeaways

As we can see, different schedulers prioritize different goals.

- FIFO focuses on simplicity, SJF focuses on minimizing wait time, and our Quantum FIFO to SJF tries to balance both

The hybrid model shows better balance than using FIFO or SJF alone.

- It avoids the inefficiency of FIFO and the starvation issues that SJF has, which creates a more fair and responsive scheduler

```
TEST CASE 1: Basic 3 Jobs
    SJF Scheduler:
    FIFO Scheduler:
    Hybrid Quantum (Q=2):
Scheduler   | Avg Wait | Avg Turnaround | Avg Response
============================================================
FIFO        |     0.33 |           2.00 |         1.67
SJF         |     0.89 |           3.11 |         0.00
Hybrid      |     1.33 |           3.56 |         0.22
============================================================


TEST CASE 2: Same Arrival Time
    SJF Scheduler:
    FIFO Scheduler:
    Hybrid Quantum (Q=2):
Scheduler   | Avg Wait | Avg Turnaround | Avg Response
============================================================
FIFO        |     0.25 |           1.75 |         1.50
SJF         |     1.88 |           3.50 |         0.00
Hybrid      |     2.12 |           3.75 |         1.56
============================================================


TEST CASE 3: Short Job Arrives Later
    SJF Scheduler:
    FIFO Scheduler:
    Hybrid Quantum (Q=2):
Scheduler   | Avg Wait | Avg Turnaround | Avg Response
============================================================
FIFO        |     0.33 |           1.67 |         1.33
SJF         |     0.78 |           2.89 |         0.00
Hybrid      |     1.22 |           3.33 |         1.11
============================================================
```

Future Ideas

- Compare More Schedulers
  - Round Robin
  - Shortest Job Remaining First
- Create a Round Robin scheduler that is FIFO in one Quantum and SJF in the next, continuing until jobs are complete

Thank You For Your Attention! Any Questions?