Design Documentation

Explanation of code -

```
typedef struct Process {
    pid_t pid;
    char name[256];
    time_t start_time;
    int priority;
    int is_running;
    int slices_run;
} Process;

typedef struct {
    Process processes[MAX_PROCESSES];
    int front;
    int rear;
    int size;
} ProcessQueue;

typedef struct {
    pid_t job_pid;
    char name[256];
    int priority;
    int is_new;
    int completed;
    time_t start_time;
    time_t end_time;
} SharedJob;

typedef struct {
    SharedJob jobs[100];
    int job_count;
    int scheduler_ready;
} SharedMemory;
```

Process structure represents a process with its PID , name etc. A queue is represented to manage multiple processes. A shared job structure represents a job with metadata, including its PID, name, priority, and completion status.

A shared memory structure represents shared memory containing multiple jobs and a count of those jobs.

```
ProcessQueue ready_queue;
Process *running_processes;
int ncpu;
int tslice;
volatile sig_atomic_t timer_expired = 0;
volatile sig_atomic_t should_exit = 0;
SharedMemory *shared_mem;
```

These are the global variables serving following functions -

ready_queue is the queue of processes waiting for their turn to run, running_processes is the array of processes which are running currently,ncpu is the number of cpu cores, tslice is the time slice for each process , timer_expired is the flag for timer , should_exit is for shutdown while *shared_mem is the pointer to shared memory segment.

```c
void initQueue() {
    ready_queue.front = 0;
    ready_queue.rear = -1;
    ready_queue.size = 0;
}

void enqueue(Process p) {
    if (ready_queue.size >= MAX_PROCESSES) return;
    ready_queue.rear = (ready_queue.rear + 1) % MAX_PROCESSES;
    ready_queue.processes[ready_queue.rear] = p;
    ready_queue.size++;
}

Process dequeue() {
    Process empty = {0};
    if (ready_queue.size == 0) return empty;
    Process p = ready_queue.processes[ready_queue.front];
    ready_queue.front = (ready_queue.front + 1) % MAX_PROCESSES;
    ready_queue.size--;
    return p;
}
```

InitQueue initializes the process queue, enqueue adds a process to the queue, dequeue removes and returns a process from the front of the queue.

```c
void timer_handler(int signo) {
    timer_expired = 1;
}

void term_handler(int signo) {
    should_exit = 1;
}
```

Timer_handler sets timer_expired to indicate that the timer has expired while term_handler sets should_exit to indicate that a termination signal was received.

```c
void stop_running_processes() {
    for (int i = 0; i < ncpu; i++) {
        if (running_processes[i].pid != 0) {
            kill(running_processes[i].pid, SIGUSR2);
            if (!should_exit) {
                enqueue(running_processes[i]);
            }
            running_processes[i].pid = 0;
        }
    }
}

void check_completed_processes() {
    int status;
    pid_t pid;
    while ((pid = waitpid(-1, &status, WNOHANG)) > 0) {
        // Mark process as completed in shared memory
        for (int i = 0; i < shared_mem->job_count; i++) {
            if (shared_mem->jobs[i].job_pid == pid) {
                shared_mem->jobs[i].completed = 1;
                shared_mem->jobs[i].end_time = time(NULL);
                break;
            }
        }

        // Remove from running processes if present
        for (int i = 0; i < ncpu; i++) {
            if (running_processes[i].pid == pid) {
                running_processes[i].pid = 0;
            }
        }
    }
}
```

stop_running_processes() stops currently running processes and enqueues them back to the ready queue if needed.
check_completed_processes() waits for child processes to complete and updates their status in shared memory.

```c
void schedule_processes() {
    // Check for completed processes
    check_completed_processes();

    // Stop currently running processes and update their slice count
    for (int i = 0; i < ncpu; i++) {
        if (running_processes[i].pid != 0) {
            kill(running_processes[i].pid, SIGUSR2);
            running_processes[i].slices_run++;

            if (!should_exit) {
                enqueue(running_processes[i]);
            }
            running_processes[i].pid = 0;
        }
    }

    // Check for new processes
    for (int i = 0; i < shared_mem->job_count; i++) {
        if (shared_mem->jobs[i].is_new && !shared_mem->jobs[i].completed) {
            Process new_process = {
                .pid = shared_mem->jobs[i].job_pid,
                .start_time = shared_mem->jobs[i].start_time,
                .priority = shared_mem->jobs[i].priority,
                .slices_run = 0
            };
            strncpy(new_process.name, shared_mem->jobs[i].name, sizeof(new_process.name) - 1);
            enqueue(new_process);
            shared_mem->jobs[i].is_new = 0;
        }
    }

    // Start new processes based on priority
    for (int i = 0; i < ncpu && ready_queue.size > 0; i++) {
        if (running_processes[i].pid == 0) {
            // Find highest priority process in ready queue
            int highest_priority_idx = ready_queue.front;
            int current_idx = ready_queue.front;

            for (int j = 0; j < ready_queue.size; j++) {
                if (ready_queue.processes[current_idx].priority >
                    ready_queue.processes[highest_priority_idx].priority) {
                    highest_priority_idx = current_idx;
                }
                current_idx = (current_idx + 1) % MAX_PROCESSES;
            }

            // Remove highest priority process from queue
            Process selected = ready_queue.processes[highest_priority_idx];
            // Shift remaining processes
            for (int j = highest_priority_idx; j < ready_queue.size - 1; j++) {
                ready_queue.processes[j] = ready_queue.processes[j + 1];
            }
            ready_queue.size--;

            running_processes[i] = selected;
            if (running_processes[i].pid != 0) {
                kill(running_processes[i].pid, SIGUSR1);
            }
        }
    }
}
```

This function checks for completed processes and updates their status, stops currently running processes and enqueues them, enqueues new processes that are marked as new in shared memory and selects and starts the highest priority processes based on the available CPU cores.

```c
int main(int argc, char *argv[]) {
    if (argc != 4) {
        fprintf(stderr, "Usage: %s <NCPU> <TSLICE> <SHMID>\n", argv[0]);
        return 1;
    }

    // Set up signal handlers
    struct sigaction sa;
    sa.sa_handler = timer_handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sigaction(SIGALRM, &sa, NULL);

    sa.sa_handler = term_handler;
    sigaction(SIGTERM, &sa, NULL);

    ncpu = atoi(argv[1]);
    tslice = atoi(argv[2]);
    int shmid = atoi(argv[3]);

    // Attach to shared memory
    shared_mem = (SharedMemory *)shmat(shmid, NULL, 0);
    if (shared_mem == (void *)-1) {
        perror("shmat failed");
        exit(1);
    }

    // Initialize
    initQueue();
    running_processes = calloc(ncpu, sizeof(Process));

    // Set up timer
    struct itimerval timer;
    timer.it_value.tv_sec = 0;
    timer.it_value.tv_usec = tslice * 1000;  // Convert milliseconds to microseconds
    timer.it_interval = timer.it_value;

    if (setitimer(ITIMER_REAL, &timer, NULL) < 0) {
        perror("setitimer failed");
        exit(1);
    }
```

```c
    // Main scheduling loop
    while (!should_exit) {
        if (timer_expired) {
            schedule_processes();
            timer_expired = 0;

            // Check if all processes are completed
            int active_processes = 0;
            for (int i = 0; i < shared_mem->job_count; i++) {
                if (!shared_mem->jobs[i].completed) {
                    active_processes++;
                }
            }

            if (active_processes == 0 && ready_queue.size == 0) {
                // Double check no processes are running
                int running = 0;
                for (int i = 0; i < ncpu; i++) {
                    if (running_processes[i].pid != 0) {
                        running = 1;
                        break;
                    }
                }
                if (!running) break;
            }
        }
        usleep(100); // Reduced sleep time for better responsiveness
    }

    // Cleanup
    stop_running_processes();
    free(running_processes);
    shmdt(shared_mem);

    return 0;
}
```

Expects three arguments (number of CPUs, time slice, shared memory ID). Sets up handlers for timer and termination signals. Attaches to shared memory to access job data. Configures the timer to trigger scheduling based on the specified time slice. Continuously checks for expired timers to schedule processes and handles process management until a termination signal is received and stops any running processes, frees allocated memory, and detaches from shared memory.

Contribution – Both Anshul Rawat (2023104) and Darsh Gupta (2023185) contributed equally.

Git Hub repo link - https://github.com/Anshul1734/OS-simple-scheduler