SIMPLE SCHEDULER

The code implements a job scheduling system in C that uses shared memory and semaphores for inter-process communication. It allows users to submit jobs with different priorities, manage job queues, and track job statistics such as wait time, run time, and completion time. The system handles job submission, termination requests, and cleanup of resources while ensuring synchronization between multiple processes.

CODE STRUCTURE

1. `main()` Function

The `main()` function is the entry point of the program and is responsible for setting up shared resources, spawning processes, and handling cleanup.

```
int main(int argc, char *argv[]) {
    if (argc != 3) {
        printf("Enter number of CPU with time_slice\\n");
        return EXIT_FAILURE;
    }
    int NCPU = atoi(argv[1]);
    int TSLICE = atoi(argv[2]);

    // Create shared memory for shared state
    int shm_fd_state = shm_open("/shared_state", O_CREAT | O_RDWR, 0666);
    if (shm_fd_state == -1) {
        perror("Shared memory opening failed for shared state");
        exit(EXIT_FAILURE);
    }
    ftruncate(shm_fd_state, sizeof(SharedState));
    shared_state = mmap(0, sizeof(SharedState), PROT_READ | PROT_WRITE, MAP_S
HARED, shm_fd_state, 0);
    if (shared_state == MAP_FAILED) {
        perror("Shared memory mapping failed for shared state");
        exit(EXIT_FAILURE);
    }

    // Open semaphore
    sem_job_state = sem_open("/sem_job_state", O_CREAT, 0644, 1);
    if (sem_job_state == SEM_FAILED) {
        perror("Semaphore creation failed");
        exit(EXIT_FAILURE);
    }
    // Initialize shared state
    shared_state->terminate = 0;
    shared_state->job_point_index = 0;
    for (int i = 0; i < MAX_JOBS; i++) {
        shared_state->instruct[i] = 0;
    }

    int shm_fd1 = shm_open("/job_array1", O_CREAT | O_RDWR, 0666);
    ftruncate(shm_fd1, sizeof(Job) * MAX_JOBS);
    jobs = mmap(0, sizeof(Job) * MAX_JOBS, PROT_READ | PROT_WRITE, MAP_SHARE
D, shm_fd1, 0);
    if (jobs == MAP_FAILED) {
        perror("Shared memory mapping failed");
        exit(EXIT_FAILURE);
    }

    // Set up signal handler for clean termination
    signal(SIGINT, terminate_check);

    // Start scheduler
    pid_t scheduler_pid = fork();
    if (scheduler_pid == 0) {
        scheduler(NCPU, TSLICE);

        // Clean up on scheduler termination
        sem_close(sem_job_state);
        sem_unlink("/sem_job_state");
        shm_unlink("/job_array1");
        shm_unlink("/shared_state");
        munmap(jobs, sizeof(Job) * MAX_JOBS);
        munmap(shared_state, sizeof(SharedState));
        exit(0);
    }

    // Run the shell interface
    simple_shell(NCPU, TSLICE);

    // Terminate scheduler and clean up resources
    kill(scheduler_pid, SIGTERM);
    waitpid(scheduler_pid, NULL, 0);
    sem_close(sem_job_state);
    shm_unlink("/shared_state");
    sem_unlink("/sem_job_state");
    shm_unlink("/job_array1");
    munmap(shared_state, sizeof(SharedState));
    munmap(jobs, sizeof(Job) * MAX_JOBS);
    close(shm_fd_state);
    close(shm_fd1);
    return EXIT_SUCCESS;
}
```

Explanation:

- **Argument Parsing**: The program takes two arguments, `NCPU` and `TSLICE`, representing the number of CPUs and the time slice for each job.
- **Shared Memory Creation**:
    - `shm_open("/shared_state", O_CREAT | O_RDWR, 0666)`: Creates a shared memory region named `"/shared_state"` with read-write permissions.
    - `ftruncate(shm_fd_state, sizeof(SharedState))`: Sets the size of the shared memory segment to match the `SharedState` struct.
    - `mmap(0, sizeof(SharedState), PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd_state, 0)`: Maps this shared memory into the address space of the process.
- **Semaphore Creation**:
    - `sem_open("/sem_job_state", O_CREAT, 0644, 1)`: Creates a named semaphore, `sem_job_state`, with an initial value of 1, enabling mutual exclusion when accessing shared data.
- **Forking Scheduler**:
    - `fork()`: Creates a child process. The child process (`scheduler_pid == 0`) runs the `scheduler` function, while the main process (`scheduler_pid > 0`) continues to run `simple_shell`.
- **Signal Setup**:
    - `signal(SIGINT, terminate_check)`: Sets up the `terminate_check` function to handle the `SIGINT` signal, ensuring a safe shutdown.
- **Cleanup**: After `simple_shell` completes, the scheduler process is terminated, and all shared memory regions and semaphores are unlinked and unmapped for a clean exit.

---

2. `terminate_check(int signum)`

This function is called when a `SIGINT` signal (typically from pressing Ctrl+C) is received. It ensures a graceful shutdown by signaling all processes to terminate.

```
void terminate_check(int signum) {
    (void)signum;
    if (shared_state != NULL) {
        shared_state->terminate = 1;
    }
    printf("Shell and Scheduler Leaving..\\n");
}
```

Explanation:

- **Purpose**: Signals all components (scheduler, jobs) to begin shutdown by setting `shared_state->terminate` to 1. This flag is monitored by both the `scheduler` and `simple_shell`.
- **System Call**:
    - `(void)signum`: Casts `signum` to `void` to suppress unused variable warnings, as `signum` is not used in this function.
- **Effect**: The termination flag in `shared_state` is set to `1`, allowing other parts of the program to check this flag and terminate their operations.

---

3. `system_time()`

This function gets the current system time in milliseconds, which is used to track job arrival and completion times.

```
long long int system_time() {
    struct timeval te;
    gettimeofday(&te, NULL);
    return (long long int)te.tv_sec * 1000LL + te.tv_usec / 1000;
}
```

Explanation:

- **Purpose**: Returns the current system time in milliseconds.
- **System Call**:
    - `gettimeofday(&te, NULL)`: Retrieves the current time since the epoch (Jan 1, 1970) in seconds and microseconds.
- **Usage**: Converts the time into milliseconds by multiplying `te.tv_sec` (seconds) by 1000 and dividing `te.tv_usec` (microseconds) by 1000. The result is used for measuring job-related timings, like `submission`, `completion`, and `waiting`.

---

4. `simple_shell(int NCPU, int TSLICE)`

The `simple_shell` function runs a user interface that accepts commands to submit or exit jobs, spawning jobs with the required time slice.

```
void simple_shell(int NCPU, int TSLICE) {
    pthread_t input_thread;
    pthread_create(&input_thread, NULL, input_handler, &TSLICE);

    if (chc == 0) {
        offset = system_time();
        chc = 1;
    }

    while (!shared_state->terminate) {
        sem_wait(sem_job_state);
        for (int i = 0; i < shared_state->job_point_index; i++) {
            if (jobs[i].active == 1) {
                int status;
                int result = waitpid(jobs[i].pid, &status, WNOHANG);
                if (result > 0) {
                    jobs[i].active = 0;
                    jobs[i].c_t = system_time() - offset;
                }
            }
        }
        sem_post(sem_job_state);
    }

    pthread_join(input_thread, NULL);
}
```

Explanation:

- **Thread Creation**: Creates a `pthread` thread to handle user input by calling `input_handler`.
- **Time Initialization**: Sets `offset` to the current time to record arrival and completion times accurately.
- **Job Status Monitoring**:
  - Uses `waitpid` with `WNOHANG` to check if a job (process) has finished without blocking.
  - If a job terminates, `jobs[i].c_t` is set to the completion time.
- **Loop and Termination**: Continuously checks `shared_state->terminate`, stopping when termination is requested.

---

5. `scheduler(int NCPU, int TSLICE)`

The `scheduler` function manages the scheduling of jobs, invoking jobs that fit within the given time slice.

```
void scheduler(int NCPU, int TSLICE) {
    int turn = 0;

    while (!shared_state->terminate) {
        sem_wait(sem_job_state);
        int num_invoked = 0;
        int running_q[NCPU];
        int len = shared_state->job_point_index;

        for (int ii = 0; ii < len; ii++) {
            int i = (ii + turn) % (len + 1);
            turn++;

            if (jobs[i].active) {
                if (num_invoked < NCPU) {
                    running_q[num_invoked] = i;
                    kill(jobs[i].pid, SIGCONT);
                    num_invoked++;
                    jobs[i

].w_t = system_time() - offset - jobs[i].s_t;
                } else {
                    kill(jobs[i].pid, SIGSTOP);
                }
            }
        }

        sem_post(sem_job_state);
        usleep(TSLICE * 1000);

        sem_wait(sem_job_state);
        for (int i = 0; i < num_invoked; i++) {
            kill(jobs[running_q[i]].pid, SIGSTOP);
        }
        sem_post(sem_job_state);
    }
}
```

Explanation:

- **Round-Robin Scheduling**: Cycles through jobs, starting with `turn` (an index offset), and checks whether each job should run or pause.
- **Job Execution Control**:
    - `kill(jobs[i].pid, SIGCONT)`: Sends `SIGCONT` to resume a job.
    - `kill(jobs[i].pid, SIGSTOP)`: Sends `SIGSTOP` to pause a job if the CPU limit (`NCPU`) is exceeded.
- **Timing**: Pauses (`usleep`) for the duration of `TSLICE` and then suspends running jobs.

Each component of this code is carefully coordinated to handle job scheduling, resource management, and clean exit in a concurrent system, allowing multiple jobs to be managed and tracked efficiently.

```
Main Process (main function)
├── Initializes Shared Resources
│   ├── Sets up shared memory for `shared_state` and `jobs` structures
│   ├── Initializes `sem_job_state` semaphore
│   └── Parses command-line arguments for `NCPU` and `TSLICE`
├── Forks Scheduler Process
│   └── Starts `scheduler()` function in child process
└── Calls Simple Shell
    ├── Starts input_handler thread
    ├── Monitors active jobs in the job queue
    │   ├── Checks if jobs have completed execution
    │   └── Updates job states
    └── Waits for termination signal
        ├── Calls `data_printer()` on termination
        ├── Signals termination to the scheduler process
        └── Cleans up shared resources and semaphore

Scheduler Process (scheduler function)
├── Continuously schedules jobs in a round-robin manner
│   ├── Checks active jobs based on NCPU and TSLICE limits
│   ├── Uses `sem_job_state` semaphore for shared resource access
│   ├── Sends SIGCONT to active jobs to resume execution
│   └── Tracks job execution and waiting times
└── Exits on termination signal
    ├── Calls `sem_post` on `sem_job_state` for cleanup
    └── Unmaps shared memory and closes semaphore

Simple Shell (simple_shell function)
├── Launches `input_handler` thread for user input
│   └── Handles user commands
│       ├── "submit": Adds jobs to the queue using `job_deposit_pre`
│       ├── "exit": Sets `terminate` flag and calls `data_printer`
│       └── Other commands: Displays error for unrecognized commands
├── Monitors Active Jobs
│   ├── Periodically checks job statuses using `waitpid`
│   ├── Updates job completion time and deactivates completed jobs
│   └── Calls `data_printer` on exit to print job stats
└── Exits on termination
    ├── Waits for the input_handler thread to finish
    └── Signals scheduler process for cleanup

Shared Resources
├── `sem_job_state` Semaphore
│   ├── Controls access to shared memory (mutual exclusion)
│   ├── Used by both `simple_shell` and `scheduler`
│   └── Released on termination
├── Shared Memory for `Job` Array
│   ├── Stores details of each job (pid, execution time, etc.)
│   └── Accessed by both processes for job status updates
└── Shared Memory for `SharedState` Struct
    ├── Holds global state such as `terminate` flag and job index
    └── Accessed by both processes for program control

Input Handler (input_handler thread in simple_shell)
├── Waits for user input from the shell
├── Parses commands and handles submission or termination requests
└── Calls `job_deposit_pre` for "submit" commands
    └── Adds jobs to the `Job` array in shared memory
        ├── Forks a new process for each job
        └── Initializes job parameters in shared memory
```

CONTRIBUTIONS

1. Priyanshu sharma 2023408

— code logic and implementation

— Github management

1. Vansh tyagi 2023582

— code logic and implementation

— Documentation

https://github.com/Priyanshu0phc/OS_66/tree/main