User: From this block diagram, implement a c project where the cyclic handling of the 2 thread that increment the counter with respect to the mutexes is done by a scheduler (with arrival and preemptions). The scheduler should be able to change the period of each task

I'll help you create a C project implementing this system. I'll break it down into multiple files for better organization. 1. First, let's create the counter and thread structures:

```
Code (c):
#ifndef COUNTER_H #define COUNTER_H #include typedef struct { uint32_t value;
pthread_mutex_t mutex; } Counter; void counter_init(Counter* counter); void
counter_increment(Counter* counter); uint32_t counter_get_value(Counter* counter);
#endif
```

```
Code (c):
#include "counter.h" #include void counter_init(Counter* counter) { counter->value = 0;
pthread_mutex_init(&counter-;>mutex, NULL); } void counter_increment(Counter* counter)
{ pthread_mutex_lock(&counter-;>mutex); counter->value++;
pthread_mutex_unlock(&counter-;>mutex); } uint32_t counter_get_value(Counter* counter)
{ uint32_t val; pthread_mutex_lock(&counter-;>mutex); val = counter->value;
pthread_mutex_unlock(&counter-;>mutex); return val; }
```

2. Now, let's create the thread task structure:

```
Code (c):
#ifndef TASK_H #define TASK_H #include #include #include "counter.h" typedef struct {
pthread_t thread; Counter* counter; uint32_t period_ms; bool is_running;
pthread_mutex_t mutex; pthread_cond_t cond; const char* name; } Task; void
task_init(Task* task, Counter* counter, uint32_t period_ms, const char* name); void
task_start(Task* task); void task_stop(Task* task); void task_set_period(Task* task,
uint32_t new_period_ms); #endif
```

3. Let's implement the scheduler:

```
Code (c):
#ifndef SCHEDULER_H #define SCHEDULER_H #include "task.h" typedef struct { Task*
tasks[2]; pthread_t scheduler_thread; bool is_running; } Scheduler; void
scheduler_init(Scheduler* scheduler, Task* task1, Task* task2); void
scheduler_start(Scheduler* scheduler); void scheduler_stop(Scheduler* scheduler);
#endif
```

4. Main implementation file:

```
Code (c):
#include #include #include #include "scheduler.h" #include "task.h" #include
"counter.h" void* task_function(void* arg) { Task* task = (Task*)arg; while
(task->is_running) { pthread_mutex_lock(&task-;>mutex);
counter_increment(task->counter); printf("%s: Counter value: %u\n", task->name,
counter_get_value(task->counter)); pthread_mutex_unlock(&task-;>mutex);
usleep(task->period_ms * 1000); // Convert to microseconds } return NULL; } void*
scheduler_function(void* arg) { Scheduler* scheduler = (Scheduler*)arg; while
(scheduler->is_running) { // Simple round-robin scheduling for (int i = 0; i < 2; i++) {
Task* current_task = scheduler->tasks[i]; pthread_mutex_lock(&current;_task->mutex); //
Signal the task to run pthread_cond_signal(&current;_task->cond);
pthread_mutex_unlock(&current;_task->mutex); // Let the task run for its period
usleep(current_task->period_ms * 1000); } } return NULL; } int main() { Counter counter;
counter_init(&counter;); Task task1, task2; task_init(&task1;, &counter;, 1000,
"Thread1"); // 1 second period task_init(&task2;, &counter;, 2000, "Thread2"); // 2
second period Scheduler scheduler; scheduler_init(&scheduler;, &task1;, &task2;);
scheduler_start(&scheduler;); task_start(&task1;); task_start(&task2;); // Let it run
for 10 seconds sleep(10); // Change periods printf("Changing periods...\n");
task_set_period(&task1;, 500); // 0.5 second task_set_period(&task2;, 1500); // 1.5
seconds // Run for another 10 seconds sleep(10); scheduler_stop(&scheduler;);
task_stop(&task1;); task_stop(&task2;); return 0; }
```

5. Implementation of the task and scheduler functions:

```
Code (c):
#include "task.h" #include void task_init(Task* task, Counter* counter, uint32_t
period_ms, const char* name) { task->counter = counter; task->period_ms = period_ms;
task->is_running = false; task->name = name; pthread_mutex_init(&task-;>mutex, NULL);
pthread_cond_init(&task-;>cond, NULL); } void task_start(Task* task) { task->is_running
= true; pthread_create(&task-;>thread, NULL, task_function, task); } void
task_stop(Task* task) { task->is_running = false; pthread_join(task->thread, NULL); }
void task_set_period(Task* task, uint32_t new_period_ms) {
pthread_mutex_lock(&task-;>mutex); task->period_ms = new_period_ms;
pthread_mutex_unlock(&task-;>mutex); }
```

```
Code (c):
#include "scheduler.h" #include void scheduler_init(Scheduler* scheduler, Task* task1,
Task* task2) { scheduler->tasks[0] = task1; scheduler->tasks[1] = task2;
scheduler->is_running = false; } void scheduler_start(Scheduler* scheduler) {
scheduler->is_running = true; pthread_create(&scheduler-;>scheduler_thread, NULL,
scheduler_function, scheduler); } void scheduler_stop(Scheduler* scheduler) {
scheduler->is_running = false; pthread_join(scheduler->scheduler_thread, NULL); }
```

To compile this project, you'll need to create a Makefile:

```
Code (makefile):
CC = gcc CFLAGS = -Wall -pthread OBJS = main.o counter.o task.o scheduler.o all:
thread_scheduler thread_scheduler: $(OBJS) $(CC) $(OBJS) -o thread_scheduler $(CFLAGS)
%.o: %.c $(CC) -c $< $(CFLAGS) clean: rm -f *.o thread_scheduler
```

This implementation provides: - A counter that can be safely incremented by multiple threads - Two threads that increment the counter at different periods - A scheduler that manages the execution of both threads - The ability to change thread periods during runtime - Proper synchronization using mutexes - Clean shutdown handling To run the program: 1. Save all files in the same directory 2. Open terminal in that directory 3. Run `make` 4. Execute with `./thread_scheduler` The program will run for 20 seconds total, changing the periods of the threads halfway through to demonstrate the dynamic period adjustment capability.

User: in task c the task.c the task_function is one of the argumentss but it isnt even declared until there. The schedueler is not necessaraly cyclic executive as the tasks that have been started just execute in a loop rather than finishing execution and waiting for the next arrival

You're making two important observations. Let's fix both issues: 1. The `task_function` declaration issue 2. Making the scheduler properly handle cyclic execution with arrivals Here are the necessary changes: 1. First, let's properly declare the `task_function` in the header file:

```
Code (c):
#ifndef TASK_H #define TASK_H #include #include #include "counter.h" typedef struct {
pthread_t thread; Counter* counter; uint32_t period_ms; bool is_running;
pthread_mutex_t mutex; pthread_cond_t cond; const char* name; uint64_t next_arrival; //
Add next arrival time } Task; void* task_function(void* arg); // Add the function
declaration void task_init(Task* task, Counter* counter, uint32_t period_ms, const
char* name); void task_start(Task* task); void task_stop(Task* task); void
task_set_period(Task* task, uint32_t new_period_ms); #endif
```

2. Now, let's modify the task implementation to properly wait for arrivals:

```
Code (c):
#include "task.h" #include #include static uint64_t get_current_time_ms(void) { struct
timespec ts; clock_gettime(CLOCK_MONOTONIC, &ts;); return (ts.tv_sec * 1000ULL) +
(ts.tv_nsec / 1000000ULL); } void task_init(Task* task, Counter* counter, uint32_t
period_ms, const char* name) { task->counter = counter; task->period_ms = period_ms;
task->is_running = false; task->name = name; task->next_arrival = 0;
pthread_mutex_init(&task-;>mutex, NULL); pthread_cond_init(&task-;>cond, NULL); } void
task_start(Task* task) { task->is_running = true; task->next_arrival =
get_current_time_ms(); pthread_create(&task-;>thread, NULL, task_function, task); }
```

3. Update the main task function implementation:

```
Code (c):
void* task_function(void* arg) { Task* task = (Task*)arg; while (task->is_running) {
pthread_mutex_lock(&task-;>mutex); // Wait for scheduler signal and check if it's time
for next arrival uint64_t current_time = get_current_time_ms(); while (current_time <
task->next_arrival) { struct timespec ts; ts.tv_sec = task->next_arrival / 1000;
ts.tv_nsec = (task->next_arrival % 1000) * 1000000;
pthread_cond_timedwait(&task-;>cond, &task-;>mutex, &ts;); current_time =
get_current_time_ms(); } // Execute task counter_increment(task->counter); printf("%s:
Counter value: %u (Time: %lu ms)\n", task->name, counter_get_value(task->counter),
current_time); // Calculate next arrival time task->next_arrival += task->period_ms;
pthread_mutex_unlock(&task-;>mutex); } return NULL; } void* scheduler_function(void*
arg) { Scheduler* scheduler = (Scheduler*)arg; while (scheduler->is_running) { uint64_t
current_time = get_current_time_ms(); // Check and signal tasks that are due for (int i
= 0; i < 2; i++) { Task* current_task = scheduler->tasks[i];
pthread_mutex_lock(&current;_task->mutex); if (current_time >=
current_task->next_arrival) { pthread_cond_signal(&current;_task->cond); }
pthread_mutex_unlock(&current;_task->mutex); } // Short sleep to prevent busy waiting
usleep(1000); // 1ms sleep } return NULL; }
```

These changes implement: 1. Proper function declaration in the header file 2. A true cyclic executive pattern where: - Tasks have specific arrival times - Tasks wait for their next arrival time before executing - The scheduler checks and signals tasks based on their arrival times - Tasks calculate their next arrival time after execution 3. Better timing control using monotonic clock 4. More accurate scheduling without relying on sleep delays The scheduler now properly implements a cyclic executive pattern where: - Tasks only execute when their arrival time is reached - Each task maintains its own next arrival time - The scheduler signals tasks when they're due to run - Tasks wait for both their arrival time and scheduler signal You'll need to add the following include at the top of the files that use the timing functions:

```
Code (c):
#include
```

This implementation provides better predictability and timing accuracy compared to the previous version, which was essentially just running tasks in continuous loops with delays.

User: same problem with scheduler function

You're right - we need to properly declare the `scheduler_function` in the header file. Here are the necessary changes: 1. First, update the scheduler header file:

```
Code (c):
#ifndef SCHEDULER_H #define SCHEDULER_H #include "task.h" typedef struct { Task*
tasks[2]; pthread_t scheduler_thread; bool is_running; } Scheduler; void*
scheduler_function(void* arg); // Add function declaration void
scheduler_init(Scheduler* scheduler, Task* task1, Task* task2); void
scheduler_start(Scheduler* scheduler); void scheduler_stop(Scheduler* scheduler);
#endif
```

2. Then move the scheduler function implementation from main.c to scheduler.c:

```
Code (c):
#include "scheduler.h" #include #include #include static uint64_t
get_current_time_ms(void) { struct timespec ts; clock_gettime(CLOCK_MONOTONIC, &ts;);
return (ts.tv_sec * 1000ULL) + (ts.tv_nsec / 1000000ULL); } void*
scheduler_function(void* arg) { Scheduler* scheduler = (Scheduler*)arg; while
(scheduler->is_running) { uint64_t current_time = get_current_time_ms(); // Check and
signal tasks that are due for (int i = 0; i < 2; i++) { Task* current_task =
scheduler->tasks[i]; pthread_mutex_lock(&current;_task->mutex); if (current_time >=
current_task->next_arrival) { pthread_cond_signal(&current;_task->cond); }
pthread_mutex_unlock(&current;_task->mutex); } // Short sleep to prevent busy waiting
usleep(1000); // 1ms sleep } return NULL; } // ...existing code...
```

This reorganization: 1. Properly declares the `scheduler_function` in the header file 2. Moves the implementation to the file where it belongs 3. Maintains the cyclic executive pattern we discussed earlier 4. Keeps all scheduler-related code in the same file Remember to remove the `scheduler_function` implementation from main.c since it's now in scheduler.c.