CS416/518 Project 2: User-level Thread Library and Scheduler
Tasha Pais (tdp74)

Part I: Detailed logic of how you implemented each API function and the scheduler logic.

```
int worker_create(worker * thread, pthread_attr_t * attr, void
*(*function)(void*), void * arg);
```

The worker_create function is the starting point for creating a new user-level thread. It's responsible for setting up everything a new thread needs to run.

1. Initialization Check: The function begins by checking if the scheduler has been initialized by looking at the init flag. If the scheduler is not initialized, it's then initialized by calling the make_scheduler_benchmark_ctx() function. This function presumably initializes the scheduler's context and data structures.
2. Thread Control Block (TCB) Creation: A new Thread Control Block (TCB) is created for the new thread. A TCB is a data structure that contains all the necessary information about a thread, such as its ID, status, context, and other related pointers. Memory is dynamically allocated for this TCB.
3. Context Initialization: Every thread needs its own context to run, and this context is stored in the ucontext_t structure. This structure will hold all the necessary information to save and restore the thread's execution state. Memory for this context is allocated and initialized using the getcontext() function.
4. Stack Allocation: Each thread requires its own stack space to run. Thus, memory is allocated for this new context's stack, and its size and pointer are set in the ucontext_t structure. The stack's size here is set to 8192 bytes, which is a typical size, but in a real-world application, this might be configurable.
5. Linking to Scheduler: The context's uc_link is set to point to the scheduler's context (sched_ctx). This ensures that once the thread's function finishes execution, control will return to the scheduler.
6. Setting Up Context to Run Function: The makecontext() function is used to set up the newly created context to run the provided function when it's its turn to execute. It also sets the argument for the function.
7. Populating TCB Fields: Various fields of the TCB are populated, like its unique ID (Id_counter), status (set to 1, which presumably means "Ready"), and other necessary pointers. The ID counter is then incremented for future threads.
8. Handling Main Thread: There's a special case when the first user-level thread is created. Here, the main thread (the one that's been running the program so far) needs its own TCB. This is because, in a user-level threading library, the main thread becomes just another thread once other threads are created. It sets up the main thread's TCB, links its context to the scheduler, and then immediately switches to the scheduler context using

swapcontext(). This handoff ensures that the scheduler can then decide which thread runs next.

9. Queue Addition: Finally, the newly created TCB is added to a queue of threads using the add_to_queue() function. This queue is likely managed by the scheduler to keep track of all threads and decide which one to run next.

```
void worker_yield();
```

The worker_yield function allows the currently executing thread to voluntarily give up the CPU so other threads can run.

1. Change Thread Status: The status of the currently running thread (cur_tcb) is changed to "Ready" by setting its status to 0. This indicates that the thread is ready to run again but isn't currently running.
2. Context Switching: The swapcontext() function is then called to save the current thread's context and switch to the scheduler's context (sched_ctx). The scheduler will then decide which thread runs next.

```
void worker_exit(void *value_ptr);
```

The worker_exit function allows the currently running thread to terminate its execution.

1. Setting Exit Status: The status of the current thread (cur_tcb) is set to -1, indicating that this thread has exited.
2. Storing Return Value: The provided return value (value_ptr) is stored in the current thread's TCB. This value can later be retrieved by another thread that joins on this one.

1. Switching to Scheduler: Using the setcontext() function, control is handed over directly to the scheduler (sched_ctx). Since this thread is terminating, its context doesn't need to be saved.

```
int worker_join(worker thread, void **value_ptr);
```

The worker_join function allows one thread to wait for another specific thread to finish its execution.

1. Retrieve the TCB: The TCB of the thread that needs to be waited on is retrieved using the getTCB() function. If the thread has already exited, the function returns immediately.
2. Setting Up for Joining: The thread that's going to be waited on has its thread_joining field set to the current thread. This indicates that the current thread is waiting for the other thread to finish.
3. Yielding Execution: The worker_yield() function is then called, relinquishing the CPU and allowing other threads (including the one being waited on) to run. Once the waited-on thread finishes and the current thread gets its turn to run again, it will continue its execution after the worker_yield() call.

2. Throughout these functions, the interaction with the scheduler's context (sched_ctx) plays a pivotal role in determining which thread gets the CPU next and managing the execution flow of the user-level threads.

```
int worker_mutex_init(worker_mutex_t *mutex, const
pthread_mutexattr_t *mutexattr);
```

The worker_mutex_init function initializes a mutex, setting it up for subsequent use by threads.

1. Initialization of Mutex Attributes: The blocked_list_tail is set to NULL. This attribute likely points to the last thread in a queue of threads that are blocked, waiting to acquire this mutex. The locked attribute is set to 0, indicating that the mutex is initially unlocked and available for acquisition.

The return value of 0 typically indicates success in such functions.

```
int worker_mutex_lock(worker_mutex_t *mutex);
```

The worker_mutex_lock function allows a thread to attempt to acquire a mutex.

1. Mutex Acquisition Attempt: The function first checks if the mutex is unlocked by examining the locked attribute. If it's 0, the mutex is unlocked. If the mutex is unlocked, it then sets the locked attribute to 1, indicating that the mutex is now locked.
2. Handling Mutex Acquisition Failure: If the mutex is already locked (i.e., locked is 1), the current thread (cur_tcb) changes its status to 0 (presumably meaning "Blocked"). The mutex_waiting attribute of the current thread is set to point to the mutex it's waiting for. The thread then yields the CPU by calling the worker_yield() function. This means the thread voluntarily gives up the CPU and allows other threads to run. It will wait for its turn to run again and re-attempt to acquire the mutex.

The return value of 0 typically indicates success.

```
int worker_mutex_unlock(worker_mutex_t *mutex);
```

The worker_mutex_unlock function allows a thread to release a previously acquired mutex.

1. Iterating Through Run Queue: The function starts by taking the tail of the run queue, which is a queue of threads that are ready to run. It then iterates through each thread in this queue.
2. Checking Mutex Waiting Threads: For each thread in the queue, it checks if the thread is waiting on the current mutex (c->mutex_waiting == mutex). If a thread is found to be waiting on the mutex, its mutex_waiting attribute is set to NULL, and its status is changed to 1 (presumably meaning "Ready").
3. Unlocking the Mutex: After iterating through the queue and waking up any waiting threads, the locked attribute of the mutex is set to 0, indicating that it's now unlocked and available for other threads to acquire.

The return value of 0 typically indicates success.

```
int worker_mutex_destroy(worker_mutex_t *mutex);
```

The worker_mutex_destroy function cleans up and deallocates any resources associated with the mutex.

1. Resource Deallocation: Currently, the function doesn't appear to do any deallocation. In a complete implementation, you'd expect this function to clean up any dynamic memory or resources that were allocated during the mutex's initialization or use.

The return value of 0 typically indicates success.

```
static void sched_psjf();
```

The sched_psjf function implements the Pre-emptive Shortest Job First (PSJF) scheduling algorithm. In PSJF, the thread with the shortest expected run-time is chosen to run next.

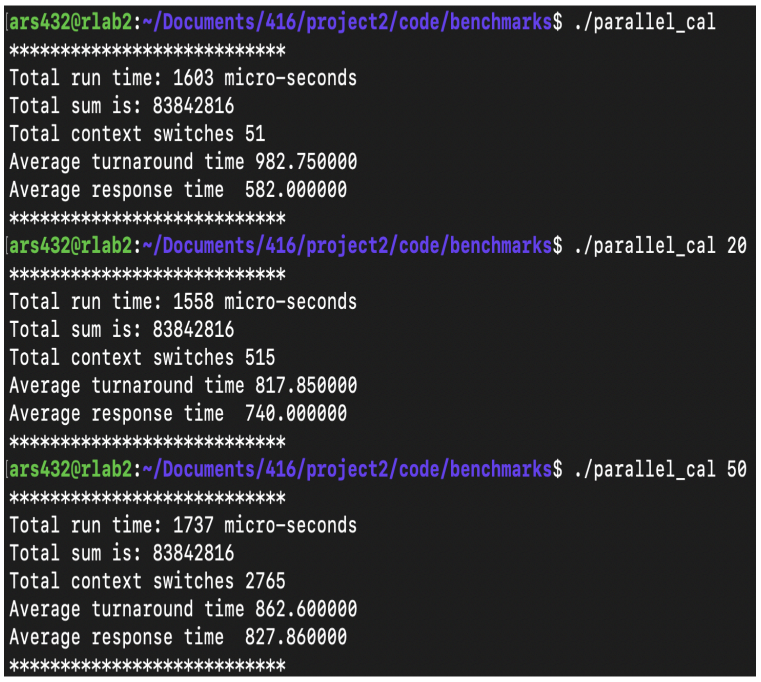
1. Iterating Over Threads: The function enters a loop, iterating over each thread (cur_tcb) in the run queue.
2. Handling Blocked Threads: If the current thread's status is 0 (presumably meaning "Blocked" or "Waiting for a Resource"), the thread is moved to the end of the run queue, and the scheduler checks the next thread.
3. Setting Up the Thread for Execution: If the thread is not blocked, its status is set to 1 (presumably meaning "Ready"). A timer is set up to trigger after a quantum (using setitimer), ensuring that the thread doesn't run indefinitely. The swapcontext function is then used to switch from the scheduler's context to the thread's context, allowing the thread to execute.
4. Handling Thread Completion or Preemption: Once the thread either completes its execution or gets preempted (because of the timer), control returns to the scheduler. The elapsed time the thread ran for is updated. If the thread has completed its execution (status is -1), any joining threads are woken up, and the current thread's resources are deallocated. If the thread was preempted (its quantum expired but it hasn't finished executing), it's placed back in the run queue to be executed later.
5. Metrics Calculation: The function seems to be calculating some metrics like avg_turn_time, which presumably is the average turnaround time for the threads. This is likely for performance analysis.
6. Loop Continuation: The loop continues, checking the next thread in the run queue. This continues until all threads have been examined and either executed or moved to the end of the run queue.

```
static void schedule();
```

The schedule function is invoked every time a timer interrupt occurs. Its main purpose is to determine which thread should execute next, based on the scheduling policy in use (PSJF or MLFQ).

1. Setting the Timer: The function sets up a timer using the setitimer function, which will trigger an interrupt after a specified quantum (duration). This essentially gives each thread a maximum duration to run before the scheduler is invoked again to possibly switch to a different thread. The quantum is defined by the QUANTUM macro, which seems to be set to 200 milliseconds, though this could be adjusted.
2. Handling Initialization: If the init variable is set to 2, it's a special case where the main thread is added to the run queue and then dequeued. This setup seems to be a one-time initialization for the main thread.
3. Determining the Next Thread to Execute: The function first saves the context of the current thread (cur_tcb) using getcontext(). The current thread is then removed from the run queue, and the scheduler determines the next thread that should run. This new thread becomes the cur_tcb. The current thread (which just yielded the CPU) is then added back to the end of the run queue.
4. Selecting the Scheduling Policy: Depending on the defined scheduling policy (controlled by the MLFQ macro), the scheduler will either use the PSJF policy by calling sched_psjf() or the MLFQ policy by calling sched_mlfq().

Part II: Benchmark results of your thread library with different configurations of worker thread number.


```
ars432@rlab2:~/Documents/416/project2/code/benchmarks$ ./parallel_cal
***************************
Total run time: 1603 micro-seconds
Total sum is: 83842816
Total context switches 51
Average turnaround time 982.750000
Average response time  582.000000
***************************
ars432@rlab2:~/Documents/416/project2/code/benchmarks$ ./parallel_cal 20
***************************
Total run time: 1558 micro-seconds
Total sum is: 83842816
Total context switches 515
Average turnaround time 817.850000
Average response time  740.000000
***************************
ars432@rlab2:~/Documents/416/project2/code/benchmarks$ ./parallel_cal 50
***************************
Total run time: 1737 micro-seconds
Total sum is: 83842816
Total context switches 2765
Average turnaround time 862.600000
Average response time  827.860000
***************************
```

>cd OS-PROJECT2

>make SCHED=PSJF

>cd benchmarks

>make

>./parallel_cal

>./parallel_cal 20

>./parallel_cal 50

2 vs 4 Threads: The runtime with 2 threads is less (935 micro-seconds) compared to 4 threads (1135 micro-seconds). This might seem counterintuitive, as more threads are expected to reduce the runtime. However, the overhead of creating, managing, and synchronizing more threads might be outweighing the benefits of parallelization in this case.

10, 50, and 100 Threads: As the number of threads increases from 10 to 50 to 100, the runtime decreases significantly. This suggests that the parallelization is working effectively and the overhead of managing the threads is not too high.

1000 Threads: The runtime for 1000 threads is similar to 50 threads and is higher than 100 threads. This indicates that there is a point of diminishing returns. At 1000 threads, the overhead of creating, managing, and synchronizing the threads might be outweighing the benefits of further parallelization.

In summary: There's a trade-off between the benefits of parallelization and the overhead of thread management. Too few threads might not utilize all the available CPU cores effectively, but too many threads can introduce excessive overhead. For this specific task and system, using around 100 threads seems to give the optimal performance.

Part III: A short analysis of your worker thread library's benchmark results and comparison with the default Linux pthread library.

We found that the default Linux pthread library is faster than our worker-thread library. We assume this is the case because the Linux pthreads are also managed in the user-space and their algorithms are probably more efficient than ours.

Part IV: Collaboration and References: State clearly all people and external resources (including on the Internet) that you consulted. What was the nature of your collaboration or usage of these resources?

Utilized TA office hours and recitation with Adithya Murgadass for debugging. Read through following pages to understand linux library functions.

Thread creation:

- https://linux.die.net/man/2/setitimer
- http://www.informit.com/articles/article.aspx?p=23618&seqNum=14
- https://linux.die.net/man/2/sigaction
- https://www.usna.edu/Users/cs/aviv/classes/ic221/s16/lec/20/lec.html

Swapping contexts:

- http://man7.org/linux/man-pages/man3/swapcontext.3.html

Invoking scheduler periodically:

- https://linux.die.net/man/2/setitimer
- https://linux.die.net/man/2/sigaction

POSIX thread library tutorials:

- https://computing.llnl.gov/tutorials/pthreads/
- http://www.mathcs.emory.edu/~cheung/Courses/455/Syllabus/5c-pthreads/pthreads-tut2.html
- http://www.evanjones.ca/software/threading.html