Rutgers University CS416 Fall 2023 Operating Systems Project 2 Report
October 25th 2023
Nivesh Nayee: nn395
Rohith Malangi: rgm111

1 - Implementation

Part 1: Thread Library

1.1 Thread creation

int worker_create(worker_t * thread, pthread_attr_t * attr, void *(*function)(void*), void * arg)

The main part of the worker_create function that will be run every time it is called: We initialize a "tcb" struct (defined in our worker-thread.h file) for the new worker thread. In this tcb initialization, a thread ID (tid) is assigned to the new worker thread. We then assign this newly created tid to the thread that was passed into the worker_create function. One of the fields in the tcb is the "cctx" (context): we now use the getcontext function to set the context of the newly created tcb. We then malloc a stack for the tcb. We then make the current context be of the newly created worker thread using the makecontext() function using the function that was passed in by the user. We then use the clock_gettime function to set the start time that the thread was scheduled.

We check if this is the first time worker_create is being called. If it is, there are some data structures that need to be initialized before we can proceed with the main part of the function.

> If it IS the first time the function has been called, we will check what the scheduling policy is to make some initialization decisions. We then call setTimer() to set up the timer to call the handle function once it goes off. If the policy is PSJF, we will initialize a single queue as our sched_queue to hold all the threads to be scheduled. If the policy is MLFQ, we will initialize an array of queues of size MLFQ_LEVELS (a macro defined in thread_worker.h).
>
> We then initialize an additional two global queues, *blocked_queue* and *completed_queue*. Blocked_queue will be used to keep track of the threads that are blocked and what order in which they should be unblocked. Completed_queue keeps track of the threads that have finished executing.
>
> We then initialize the context of the scheduler. We also malloc a stack for the scheduler and set the context's other attributes. We then use makecontext() to save the schedule function for the context of the scheduler.
>
> We then initialize the main thread's tcb and also set a special flag in that tcb to denote it is the main function. Then we will use the *getcontext()* function to set a global variable "main_ctx". This global variable stores the context of the main thread.
>
> We then set the global variable current_thread to the newly created main_thread and set the global firstCall flag to 0, so that this block of code will never be executed again. The global timer is then started.

Now back outside of the scope of the firstCall block of code, we either enqueue the new thread to the sched_queue if the policy is PSJF, otherwise we enqueue it to the top level queue in our array of queues if the policy is MLFQ.

1.2 Thread Yield

int worker_yield()

First thing we do is to change the status field in the current_thread's tcb from RUNNING to READY. This denotes it is not the running thread anymore. We then set the context using getcontext().

Our next if statement decrements the priority of the current thread because by default the priority is incremented when the thread is initially scheduled. Following the MLFQ rules, we want the thread's priority to stay the same if it yields to the CPU. This won't affect the PSJF algorithm because priority is the same for every thread in PSJF.

We finally swap the context from the current thread to the scheduler context.

1.3 Thread Exit

void worker_exit(void *value_ptr)

First we check if the any thread is being blocked by the current thread:

> If it is blocked, we will find and remove it from the blocked queue and set its status to READY. Then we will enqueue it to the appropriate queue based on the scheduling policy.

If the thread was not blocked, we will first the the threads status to FINISHED. Then we add it to the queue of completed threads. We then use the clock_gettime function to get the time this thread ended at to calculate the turnaround time. If the thread is not main, we will then do the calculation of the turnaround time and then update the global avg_turn_time variable.

The thread is then freed and the context is set to the scheduler.

1.4 Thread Join

Firstly, we need to find the thread the user have provided as parameter and for that, we have created a function which finds the thread from the sched_queue. Next, if the thread is not null then we assign the id of current running thread to the thread we just found(joiningThread) and then we assign the status of current thread as Blocked and put it in blocked_queue, then we swap the context to the scheduler so that it can not assign the next appropriate thread and the blocked thread will not be in the sched_queue anymore.

If we dont find in the sched_queue then we check in the completed queue which we created to keep track of the finished threads. Once found then we let the main thread keep execute or else we return -1 as indicating thread doent exists.

1.5 Thread Synchronization

MUTEX_INIT:
Here we simply just initialize the variables created in mutex_t struts and mallocing the mutex queue and clearing the mutex clock (lock is free). I also have the mutex if which indicates the owner of that lock and it has the id of the thread which have the lock.

MUTEX_LOCK:
Here we are using the test_and_set function which repeatedly checks if the lock if free or not and if the lock is not free then it will go inside the loop where we simply adding the thread in the waiting list of the mutex queue and assigning the status as MUTEX_BLOCKED, so that shced_queue wont allow that thread to b e enqueued.

MUTEX_UNLOCK:
Here the thread releases the lock and we simply clear the lock using the atomic operation and set the mutex id as -1 indicating as no owner. We then release all the threads which were waiting for the threads to the sched_queue so that the scheduler can assing those threads back again and we assing the status of those threads as READY. If the sched_policy is MLFQ then only we check the priority of that thread which indicates which queue that thread should be going and we simply add that in that queu. If PSJF then we just add in the sched_queue.

MUTEX_DESTROY:

We set the queue of the threads waiting for this mutex to NULL and then we free all the threads that are waiting for this mutex lock since it’s being destroyed.

Part 2: Scheduler

PSJF
First we will make sure the thread is set to READY if it is set to running. Then we enqueue it to the end of the scheduler queue.

We then use our sortQueue() function to sort the scheduler queue based on a proxy variable for the amount of time it has taken so far. This follows the STCF principle.

We will then dequeue the thread which has the lowest expected completion time from the now sorted sched_queue and make that the current_thread.

Making sure that there was something to be dequeued (current_thread is not null), we will set the current_thread status to RUNNING, and increment its counter. Counter is the value that’s used to keep track of how long the process is taking (It increments every time the process is switched out). We also increment the global variable tot_ctx_switches. If it’s the first time the new thread is being scheduled, we will update the response time for that thread and do a calculation for avg_resp_time global variable.

MLFQ

Similarly to the beginning of the PSJF function we will set the state of the previously running thread from RUNNING to READY. The priority of the thread has already been decremented when it was initially scheduled so we enqueue it to the level priority that it has stored in its TCB. Since each MLFQ level follows RR scheduling, we can simply enqueue to the end of the queue.

Next step is to find the next thread to run. We run a for loop that will search through each level of the mlfq_levels array but will break once we've found a thread to run. (We have set up the array of queues so that the queue at index 0 is top priority and the priority decreases as index of the array increases). When we find a queue that is not empty, we will dequeue a thread from there, that will be the thread to run and it will be the highest priority based on our structure.

We do a safety check to make sure the dequeued thread isn't null. Then we set that thread's status to RUNNING and if it is not in the lowest priority queue already, we increment its priority. (A higher priority value means an actual lower priority in our implementation).

We then check if the thread hasn't been scheduled yet. If it hasn't we update the response time field of that thread. Then we do a calculation for the global response time and to the avg_resp_time global variable.

Lastly we will set the context to the current thread.

2 - Benchmark Results

PSJF

Parallel_cal

	Number of threads								
	3	**6**	**15**	**20**	**30**	**50**	**100**	**200**	**2000**
Total Run time (milliseconds)	**1.599**	**1.536**	**1.563**	**1.550**	**1.540**	**1.554**	**1.574**	**1.551**	**8.308**
Total sum	**83842816**	**83842816**	**83842816**	**83842816**	**83842816**	**83842816**	**83842816**	**83842816**	**83842816**
Total context switches	**73**	**72**	**84**	**89**	**98**	**119**	**169**	**266**	**2070**
Avg turnaro	**1575.000000**	**1417.166667**	**1534.800000**	**1326.9500**	**1135.266**	**1022.700**	**863.24000**	**833.650000**	**5911.11800**

und time				**00**	**667**	**000**	**0**		**0**
Avg response time	**51.333333**	**126.833333**	**354.600000**	**476.700000**	**507.566667**	**619.540000**	**649.930000**	**718.275000**	**5872.240500**

external_cal

	Number of threads								
	3	**6**	**15**	**20**	**30**	**50**	**100**	**200**	**2000**
Total Run time	**0.356**	**0.376**	**0.358**	**0.332**	**0.357**	**0.310**	**0.367**	**0.341**	**7.085**
Total sum	**-699566086**	**-699566086**	**-699566086**	**-699566086**	**-699566086**	**-699566086**	**-699566086**	**-699566086**	**-699566086**
Total context switches	**15**	**18**	**36**	**32**	**53**	**62**	**114**	**212**	**2012**
Avg turnaround time	**325.000000**	**331.500000**	**187.866667**	**247.650000**	**140.933333**	**244.300000**	**306.240000**	**293.200000**	**5389.437000**
Avg response time	**63.333333**	**144.500000**	**75.800000**	**199.050000**	**85.033333**	**225.780000**	**294.010000**	**287.815000**	**5378.996500**

Vector_multiply

	Number of threads								
	3	**6**	**10**	**15**	**20**	**30**	**100**	**200**	**2000**
Total Run time	**0.016**	**0.016**	**0.018**	**0.021**	**0.022**	**0.024**	**0.027**	**0.029**	**6.825**
Total sum	**63156048**	**63156048**	**63156048**	**631560480**	**631560480**	**631560480**	**631560480**	**631560480**	**631560480**

	0	**0**	**0**						
Total context switches	**4**	**7**	**11**	**16**	**21**	**31**	**101**	**201**	**2001**
Avg turnaround time	**10.000000**	**9.166667**	**9.600000**	**10.666667**	**11.200000**	**11.933333**	**13.310000**	**15.480000**	**5167.572500**
Avg response time	**5.000000**	**6.500000**	**7.800000**	**9.333333**	**10.100000**	**11.166667**	**13.050000**	**15.355000**	**5167.546500**

MLFQ

External_cal

	Number of threads								
	3	**6**	**10**	**15**	**20**	**50**	**100**	**200**	**2000**
Total Run time	**0.318**	**0.315**	**0.322**	**0.338**	**0.378**	**0.311**	**0.362**	**0.367**	**0.365**
Total sum	**-2005565287**	**-2005565287**	**-2005565287**	**-2005565287**	**-2005565287**	**-2005565287**	**-2005565287**	**-2005565287**	**-2005565287**
Total context switches	**15**	**19**	**29**	**27**	**33**	**63**	**113**	**215**	**2015**
Avg turnaround time	**291.333333**	**307.333333**	**218.000000**	**239.133333**	**271.550000**	**247.920000**	**286.300000**	**290.650000**	**291.892500**
Avg response time	**55.000000**	**136.333333**	**121.000000**	**170.000000**	**215.750000**	**229.180000**	**275.550000**	**285.290000**	**291.350000**

Parallel_cal

	Number of threads

	3	6	10	15	20	50	100	200	2000
Total Run time	**1.571**	**1.543**	**1.550**	**1.555**	**1.583**	**1.541**	**1.579**	**1.556**	**2.040**
Total sum	**83842816**	**83842816**	**83842816**	**83842816**	**83842816**	**83842816**	**83842816**	**83842816**	**83842816**
Total context switches	**74**	**77**	**78**	**83**	**101**	**116**	**167**	**268**	**2088**
Avg turnaround time	**1542.666667**	**1475.166667**	**1431.800000**	**1536.266667**	**1400.800000**	**1021.260000**	**858.920000**	**837.345000**	**948.447500**
Avg response time	**52.000000**	**126.000000**	**227.700000**	**350.933333**	**479.600000**	**617.040000**	**643.110000**	**720.800000**	**927.447000**

Vector_multiply

	Number of threads								
	3	**6**	**10**	**15**	**20**	**50**	**100**	**200**	**2000**
Total Run time	**0.016**	**0.016**	**0.018**	**0.020**	**0.022**	**0.029**	**0.024**	**0.029**	**0.038**
Total sum	**631560480**	**631560480**	**631560480**	**631560480**	**631560480**	**631560480**	**631560480**	**631560480**	**631560480**
Total context switches	**4**	**7**	**11**	**16**	**21**	**51**	**101**	**201**	**2001**
Avg turnaround time	**10.333333**	**9.166667**	**9.500000**	**10.600000**	**11.150000**	**14.900000**	**12.190000**	**15.195000**	**17.386500**
Avg response time	**5.000000**	**6.500000**	**7.700000**	**9.266667**	**10.050000**	**14.320000**	**11.950000**	**15.055000**	**17.375000**

3 - Our library vs Linux pThread

PSJF:

The run time of our worker-thread library is getting around ~1500 micro seconds and for the Pthread Library, it is getting around 300~500 micro seconds. Our library have some implementation which takes more runtime than actual pthread also, we were not required to have the precise time of the threads so that would one of the reason it gives move run time than Pthread.

MLFQ:

The run time of our worker-thread library is getting around ~1500 micro seconds and for the Pthread Library, it is getting around 300~500 micro seconds. Our library have some implementation which takes more runtime than actual pthread also, we were not required to havedifferent time for every queue levels, and not required to keep track of precise time of all the threads so that would one of the reason it gives move run time than Pthread.