Pre-emptive User Level Thread Library

Design Document

Asad Ali (asad_82@yahoo.com)

Introduction

In the first part of this document we'll explain how different functions collaborate in our implementation using a collaboration diagram, which will also show you the flow of control in our implementation. But before I move ahead into the details let me tell you some main functions that are the core of this implementation and they are:

1. **Scheduler()**
   The scheduler is responsible for picking the highest priority job from available jobs. As a job is found in the queue it is removed and given a chance to execute in the time slice allocated by the system. Scheduler either restores its context through a long jump to the previously stored context or it calls increment stack to allocate some stack space and call the user thread from there if the function is about to execute for the first time. An array with the name of pQueue having 128 priority queues is controlled and managed by the scheduler. As soon as a job is inserted into the queue with a given priority the count of the number of threads queued at that level is increased by one, and the scheduler scans this list of count associated with each level in bottom up fashion and selects the job from the queue at which the count is first found to be non-zero.

2. **Timer Handler()**
   In the thread system every job is given specific time and when its time span expires, an interrupt is generated by the system named SIGVTALRM, which is handled by timer handler. The timer handler on the interrupt saves the context of running thread and calls the scheduler to select the next job. The timer handler has been setup using the flag of SA_NOMASK, which allow the timer interrupt from within the timer handler. At present the system allocated 10ms to each thread to finish its job or other wise fall prey to the timer interrupt, resulting in its yielding. Also at the start the timer interrupt signal is added to a set of signals so they can be blocked or restored later in the code.

3. **Increment Stack()**
   The increment stack function is called every time a thread is about to execute for the first time. It calls the thread function provided by the user. But before that it moves stack forward by 10000 bytes. The most important point here is that this function never returns when the user thread exits but it performs a long jump to the scheduler to transfer the control back. Also as the thread returns from the user

called function a check is performed to see whether some other thread is waiting for the completion of this thread or not. The comparison is performed using the unique thread identification number and waiting threads are put in the ready queue, if found.

4. **Disabled Timer() & Enabled Timer()**
   Whole of the thread system is made atomic to assure that while executing the core code of the system no context switch occurs. For this purpose the interrupts generated by the system are no more treated and blocked and after the execution of that part of code completed, interrupts are unblocked calling Enabled Timer.

Thread System Collaboration and Implementation

1. The program begins its execution from the main function of the user program from where the control is transferred to the thread system by the t_start function.
2. As soon t_start begins its execution it allocates memory to some variables to be used by the thread system internally these include the multilevel priority queue, current thread which, is the thread in execution. In t_start timer interval is set for the thread system. Then it calls t_fork, which makes an entry into the ready queue for the first thread, then it calls the Scheduler which returns only when the priority queue gets empty.
3. In the implementation of the t_fork we first check from priority queue whether the current thread being forked has the priority greater than the thread in execution, if yes then we immediately preempt the current running thread and give chance to the higher priority thread and if no, then just put the forked thread in its appropriate location in priority queue. Remember there is no spawner mechanism. No signaling, all we do is create a TCB for this thread set its parameters and we put it into the priority queue and return the control back to the calling function normally.
4. As soon as the Scheduler begins its execution it performs a set jump and save its context at the start of the function for future use. Then it call another function named sitbottom() which allocates stack space for the thread system amounting to 50*5000 of integer type. Then this function saves its context using set jump and performs a long jump to the scheduler context, which was just saved previously. This is an important point, as we do not return from the function. The net result is that we have stack space reserved for our thread system.
5. Now the control is back to the scheduler who checks whether the bitmap count of priorities is empty or not. If not then it picks the highest priority thread from he queue and removes it from the queue. If it is empty then the control is back to the t_start.
6. If it’s a newborn thread then we call increment stack, which moves the stack forward by 10000 bytes and calls the thread function provided by the user. The increment stack function never returns even if the user thread function exits what it does is it performs a long jump back to the scheduler to transfer the control when the user thread exists.

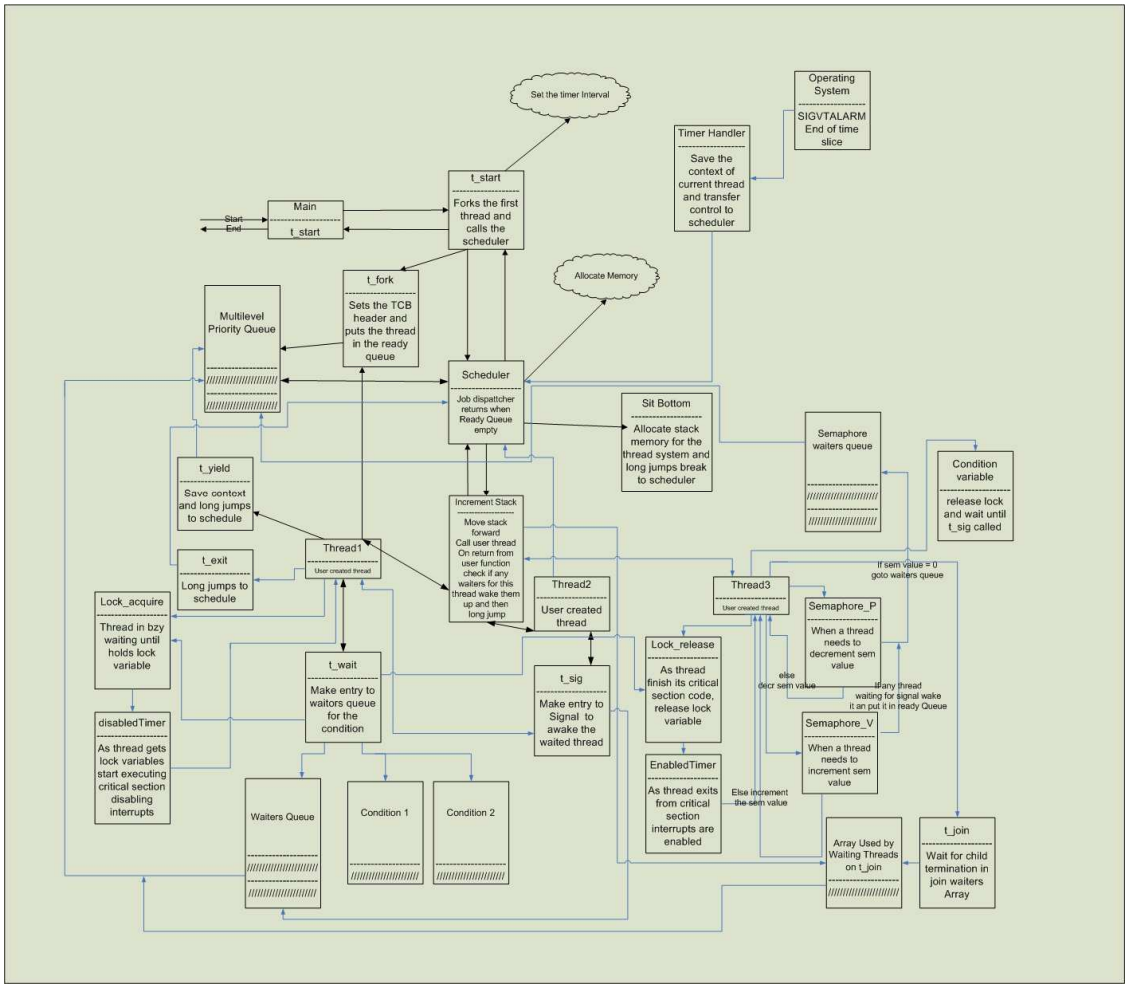


Figure 1 Collaboration Diagram showing various functions invoking others in our system

7. Now we can say that we have a user thread running, which during its execution can yield, so when the user thread yields we perform a set jump and save its context then we perform a long jump to the scheduler context saved previously, returning the control to scheduler to select a new job and begin its execution.
8. **Condition Variables:** Now the condition variables are implemented with an associated lock. When ever a thread calls t_wait while holding the lock, first of all the lock is released and then a setjmp is performed to save the context of the thread which is then put into the waiters queue for that condition. Remember that there is no signalers queue now.
9. When a thread signals on a condition, a check is performed in the conditions waiter queue which also has priorities, to see if there are any waiters for that signal is so then the lock is passed to them as argument which is then again acquired before the thread restores back into the critical region. The signal is simply lost if no waiters at any level are found. In the implementation terms a

specialized counter named conditionWaiters is associated with each condition variable to keep track of the number of waiting threads at different priority levels.

10. **Locks:** A lock structure having a single variable is declared which can have either 1 or 0 value depending upon the whether the lock is released or acquired.
11. When a thread call the acquire function with a lock as argument first a check is performed whether the lock has already been acquired or not, this implementation is based on busy waiting. As soon as the lock variable is freed the timer signal is masked and disabled and the lock is again set to 1 and the call simply returns.
12. For disabling the timer interrupt sigprocmask with SIG_BLOCK as argument has been used. Similarly in lock release sigprocmask with SIG_UNBLOCK as argument enables the timer interrupt back.
13. **t_join:** When a t_join is called by some thread a check is first performed to see if the ID of the thread to which the current thread wants to join is not among exited thread IDS, is so then a the current thread blocks until the required thread exits. Other wise if the thread has already exited the current call of the tjoin simply return with out waiting.
14. To keep the record of the exited threads as soon a thread exits and entry is made into an array for that thread, only the tid is recorded so that if some other thread tries to wait on it a bit later it can be checked.
15. **Semaphore_P:** A structure is declared which has a value field an ID field and an associated waiters queue. Semaphores are implemented to cope with the synchronization problems among multiple threads. Counting semaphores have been implemented in the thread system. Semaphore_P is called when a threads wants to decrement the semaphore counter. If the counter is at 0 no further decrement can take place, instead the thread that called the function would block on a waiters queue associated with that semaphore.
16. **Semaphore_V:** The function is used to increment the semaphore counter. When a thread calls this function, waiter's queue is checked and if there is thread waiting for signal then counter is not incremented instead waiting thread is brought in prioritized ready queue and removed from waiter's queue. If no thread in waiting in the queue then simply counter is incremented.
17. Some other functions for altering the time slice the thread system allocates has also been implemented they are fairly trivial to understand so no explicit explanation is provided here.

Important Points

- Preemption is implemented in the system, so that when a thread of higher priority comes, the lower priority thread is preempted and chance is given to higher proirity thread to execute.
- Round robin and Priority schemes are implemented in thread system. Higher priority threads are given chance to execute first and after a specified time interval, other threads of same priority are given chance to execute. If no other thread of same priority is currently in the system, chance is repeatedly given to the higher priority thread until it completes its execution. It is checked after every time slice whether any high priority thread is in system or not.

- Multilevel priority queues are implemented. Every priority level has a queue where threads of same priorities are stored. O(1) is achieved while finding the highest priority thread from these queues.
- Critical sections are made safe from being inconsistent by different threads. For this purpose timers are disabled and enabled many of the times.
- Semaphores Locks and condition variables are used to solve the synchronization problems like Producer Consumer Problem, Bridge problem and Elevator problem.

Answers of some important Questions

What timing information do I need to keep?

Timing information are kept when we need to analyze whether the Thread is CPU bound or I/O bound in order to allocate them the appropriate priority, since in this project we are not working on this so we don't need to store any timing information with a thread.

What is the maximum quantum of time that I want to allow a thread to run?

Deciding the appropriate time is really tough job. A timeslice that is too long will cause the system to have poor interactive performance; the system will no longer feel as if applications are being concurrently executed. A timeslice that is too short will cause significant amounts of processor time to be wasted in scheduler in selecting a new job and the job will take longer then usual time to complete.

We have tried different time intervals to set the proper time slice. In the experiments we performed it has been observed that a time slice of less than 1ms is of no use as significant amount of this time slice is wasted by the Linux OS and when the control reaches to our timer handler it is again about time for the next interrupt. Also we have studied how Linux allocated time to its processes and after that we decided to allocate 10ms for the execution of each thread. This time slice was chosen as our thread system will only be used by threads, which are CPU bound and require no inputs from the user. Although outputs will be there but for them the system does not have to wait, as they are asynchronous. So a time slice of 10ms second gives a fair amount of time to the thread to complete its job. Also if we allocate a bigger time slice, then most of the threads will yield by them selves and time slicing will not be effective. Most of the test cases that we were required to implement result in the yielding to the control by the thread before the time slice is over. So to prevent that from happening we opted for a lower value of the time slice.

At what granularity do I want to receive clock interrupts?

As explained above the clock interrupts will be received by the thread system every 10ms resulting is better performance in the scenarios provided to us for implementation as test cases.

What will my timer interrupt handler do?

When timer interrupt comes after specified time interval, the timer handler will first of all save the context of the currently running thread using setjmp and would then make an entry into the prioritized ready queue and then would transfer the control to the scheduler to schedule a new job for the next time slice.

What events would cause a context switch between threads? (Note: A context switch may be performed in many instances. Think carefully about this and include these observations in your readme document)

Context switch occurs in following cases

1. A higher proirity thread causes context switch from running thread when forked.
2. When the time slice of running thread completes, there is context switch.
3. When a thread complete its execution.
4. When a thread explicitly yields.
5. When a thread goes to wait on a condition variable.
6. When a semaphore P and V are called.

How do asynchronous interrupts like timer affect my processes.

A thread could be in the middle of updating a shared variable when an asynchronous interrupt like timer will result in a context switch, which will lead to inconsistent state of the system that is using our preemptive library, because other threads during there execution will modify the shared variable and when the thread which was preempted in the middle of updating a shared variable is restored, it will assign incorrect values to the shared variable.

How do asynchronous interrupts affect the implementation of condition locks and semaphores.

The thread system could be in the middle of updating some queue used by itself say for example the ready queue, which would then be inconsistent and hence would result in a crash of the thread system itself. To handle and avoid this situation as soon as the thread system gains control of the processor it immediately disables the timer interrupt by masking the signal.

What about critical regions of code.

Critical regions of the user as well as the thread system code are protected from being interrupted in the middle by disabling and enabling the interrupts back.

Do I ever need to mask out (disable) interrupts? If so, how? Why?

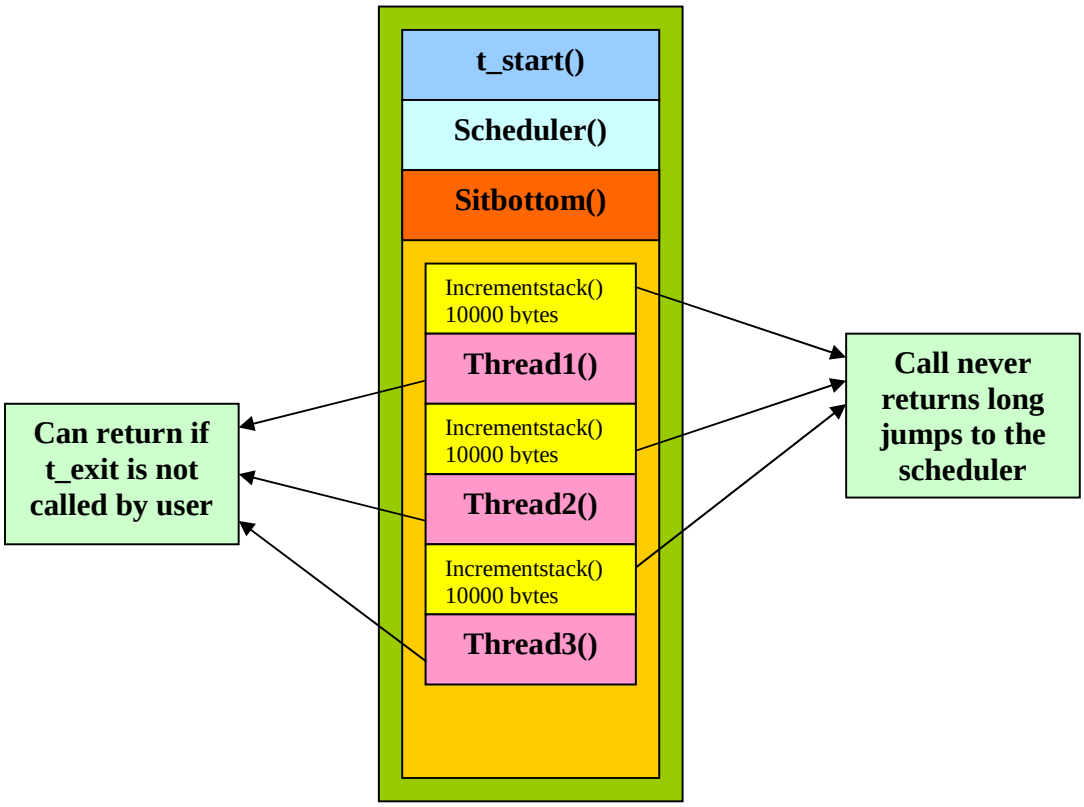
Whenever shared variables are accessed by a thread, it might be possible that a context switch occurs while thread is still working in its critical section that can cause inconsistent values (synchronization problems), to avoid switching when working in

critical region, timer interrupts are disabled using locks and when thread complete its work in critical region it enable the timer interrupt back by releasing the lock.

Interrupts are disabled with the help of sigprocmask, setting the flag to BLOCK blocks all the interrupts generated by SIGVTALARM to take action when a time slice completes.

How Stack is managed

1. Managing stack was the trickiest part of the project but we finally got around that. The diagram below shows the stack situation while three user threads are in execution mode.
2. t_start function calls allocate memory which after allocating memory to the system variable returns gracefully hence popped out.
3. Then Scheduler is called by t_start which does not return until ready queue becomes empty.
4. The scheduler function calls sitbottom function which allocates 50*5000 bytes for the whole thread system. It is the stack space in which other thread function will be called and will live. Shown in orange color in the diagram.



5. The sitbottom function never returns so the space allocated by it remains valid till the scheduler function returns. The sitbottom then long jumps to scheduler saved context and returns the control.
6. The scheduler then removes a thread from the ready queue and calls increment stack function which allocates the stack space for that specific thread here 5000*threadCount.
7. So we move a different amount of space forward each time. The reason for this is that we are not returning the threads, see the figure which says that increment stack never return it perform a long jump to the scheduler. So if we want to save the stack space of previously created user threads from corruption which are still in execution we need to skip the space occupied by those thread functions, which is done by multiplying 5000 by thread count.
8. The increment stack function is called every time a new thread is about to run for the first time, hence correcting the stack for it, and the process goes on.

Assumptions

1. There are no as such assumptions that have been made except that the sitbottom function allocates the stack space initially which can accommodate 50 user threads. If you want to accommodate more user threads then just increase the number internally in the sitbottom function other wise the stack of the 51st user thread will be corrupt. But this is no limitation it is an extra feature that enables the user to know how much stack space he can get from the operating system in advance, suppose if this feature had not been there then the thread library can crash in the middle with all threads running when it is unable to allocate more stack space for some new thread the user wants to run. So this feature enables the user of this thread system know in advance the stack space that this application can get from the OS so if its not available then do not create any thread protecting a crash in the middle of the execution and losing critical information.