

OS Project Final Report

TEAM MEMBERS:

Group Number : 11

Team Members:

Aditya Kumar Akash (120050046)

Bhargav Chippada (120050053)

Shyam JVS (120050052)

Nishant Kumar Singh (120050043)

Ranveer Agarwal (120050020)

Prateek Chandan (120050042)

Note: At the end of each part there is a section which describes how to test and run the part.

Goal : Design and implementation of KLT in Geek OS

Implementation : shyam + aditya

Testing - Shyam

Aim of first session:

Study GeekOS and decide how kernel-level threads (KLTs) can be implemented in GeekOS.

Design KLTs and begin their implementation.

The following are features which have already been supported in the geekOS code :

- The data structure implementing the kernel thread is already defined in the file **kthread.h**. This struct (called Kernel_Thread) is used for both (what is called) user-level threads and kernel-level threads. Both of these are instances of kernel-level threads only (processes actually - see next point) , in the code, as both get scheduled by the scheduler. The difference in the name is only because of the modes in which both

operate, that is, user-level threads run in user mode while kernel-level threads run in kernel mode.

- Context loading from the executable is already supported by GeekOS.
- Currently each new thread creation stands for making a new context and attaching it to the thread.

DESIGN:

Overview -

For Implementing thread creation at kernel level we need to use the same context as the CURRENT THREAD and execute a function in the thread concurrently. The thread created is added to the queue of processes maintained by the schedulers which makes it kernel level.

Details of design :

- The userContext of CURRENT_THREAD is the context object used for the new thread created
- System Call supported for the creation of the thread
- The current design allows only functions which do not take any arguments to be concurrently executed.
- The kernel_thread structure allows to define **Entry Point** from which a given thread starts execution as if scheduled after pre-emption. We make use of this feature to specify the point of execution of the thread by passing the function pointer.
- Features to initialize the thread stack and making runnable (putting thread on queue) is already supported. These features are made use of for realizing the aim.

IMPLEMENTATION

Syscall defined -

int SpawnThread(void*(*func)(void)) - (Defined in "conio.h")

- Takes argument a function which can return void* and takes no argument to be processed by the thread
- Returns the pid of the thread if thread is created rightly

Sys_SpawnThread() -

- The Function realizing the feature of SpawnThread
- Creates a new thread using the CURRENT_THREAD->userContext
- Uses the function Pointer taken as argument to initialize the entry Point of new thread to this function

Functions added :-

Setup_Child_User_Thread() - in kthread.c

- This function is similar to SetUp_User_Thread(), with difference being the initialization of the entry point with the argument **eip (Instruction pointer)** passed to it

Start_Child_User_Thread() - in kthread.c

- Function is similar to Start_User_Thread(). Difference being that this used Setup_Child_User_Thread() to set the thread with the eip passed to this function.

Testing the implementation

z.c

This the code for demonstration of the implementation described above.

The program is a user end program which creates ≥ 1 thread and show how the same context is being referenced by it.

A count variable is being incremented by each instance of the thread.

The value of count is printed in each instance. If the value of count printed is increasing then it implies that same count variable is being referenced by the threads. This shows the presence of same context in both the threads.

How to run

1. Start the geek OS
2. Run z.exe and press enter.
3. See the output value of count to check for KLT correctness.

Note :

Include conio.h to use the syscall SpawnThread.

Please maintain a while(1); in each thread function created. This helps to let main create more threads before the threads destroy the context on exit.

Please maintain `exit(0);` in each thread function to exit from thread.
Not maintaining exit code may give unexpected exceptions.

Scheduling Policies Implementation

Work Division

- Aditya and Shyam worked on code implementation of RR and ML scheduling
- Nishant understood the code in detail and pointed out places where the scheduling decisions were being made.
- Ranveer helped in ideation, and the theoretical concepts behind what various schedulers do. This involves for instance, deciding the no. of levels in ML scheduling and assigning time slices to them.
- Bhargav was responsible for testing the code and checking if indeed the scheduling policies were actually in place and are working fine in various cases.
- Prateek helped in preparing the report and summarizing the progress made so far. He also has suggested things to be done in the coming days and the order in which they are to be done.

Design

1. Understood the scheduling policy currently followed in GeekOS
2. The current policy uses Static priority policy
3. Incorporated two new scheduling policies - RR (Round Robin), ML (MultiLevel Adaptive)
4. For RR, instead of selecting the highest priority process we dispatched the process which was at the head of the ready queue (Runnable queue in GeekOS terminology). This ensured a RR policy was followed.
5. For ML Adaptive , we made 4 levels of priority queues which are contains ready processes. Each level was given different time slice {4, 5, 6, 7}. Promotion of process to a higher level was done when either process yielded the CPU voluntarily to some other

process or got blocked before time slice expired . Demotion done when process completely used the allocated time slice.

Implementation

1. RRS denotes Round Robin Scheduling, MLS denotes Multi Level Adaptive scheduling
2. A variable currentSchedulingMode decides which scheduling policy the OS would follow. This has to be manually set at present. Syscall to be added to facilitate the online change of scheduling policy. Changing this changes the policy used by OS.
3. The Get_Next_Runnable_Locked process is modified to allow scheduling a suitable process given the policy
4. Timer_handler modified to allow demotion of process upon exhaustion of its time slice (called level_quantum)
5. Yield method changed to allow promotion when a process voluntarily gives up the CPU
6. Wait method changed to allow a process to get promoted when it gets blocked
7. s_runQueue made an array of Thread_Queue to allow multi levels of priority based ready processes. s_runQueue[0] used for RRS.
8. levelQuantum made to allow for different time slice for different levels
9. MAX_QUEUE_LEVEL keeps track of number of queue levels for MLS
10. currentPriorityLevel attribute added to kthread struct to keep track of priority level of a given process

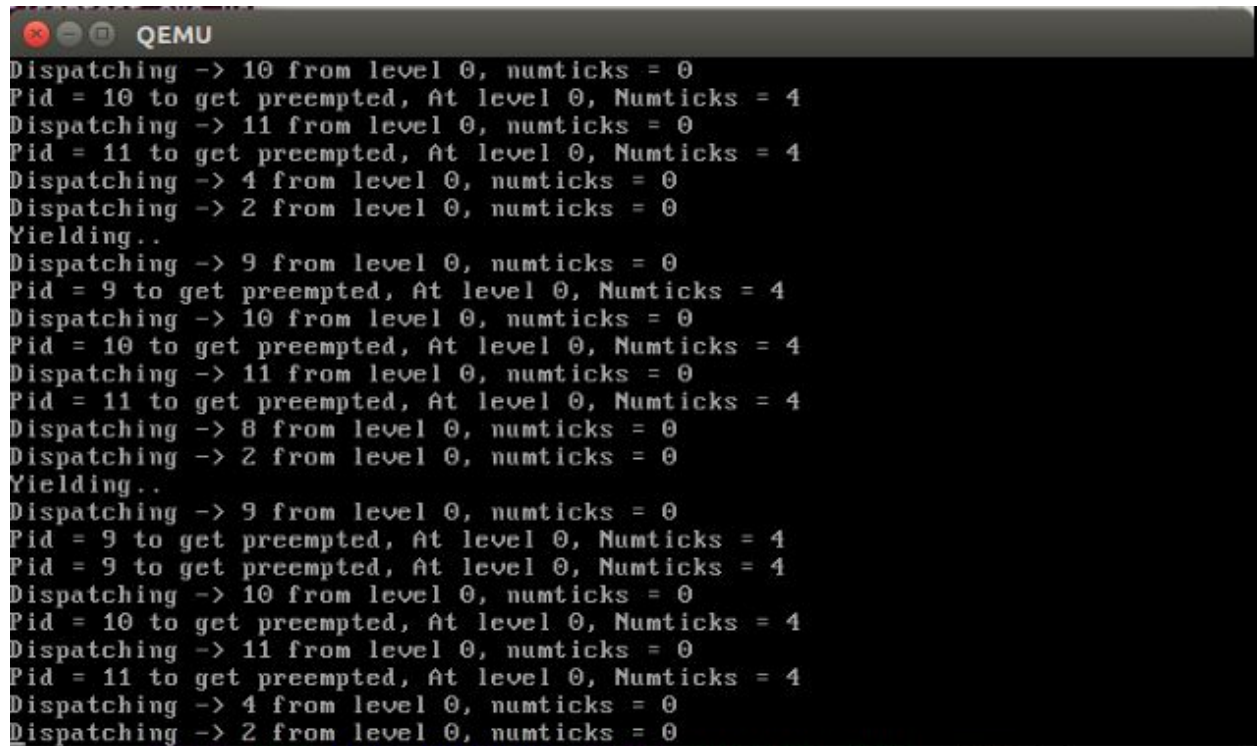
Testing

Different Policies were tested as below -

RR

3 process instances were run in background.

Each of them can be seen scheduled in RR fashion and being picked from Level 0 which is the level used for RR scheduling. The time slice consumed can be seen to be 4 ticks.



```
QEMU
Dispatching -> 10 from level 0, numticks = 0
Pid = 10 to get preempted, At level 0, Numticks = 4
Dispatching -> 11 from level 0, numticks = 0
Pid = 11 to get preempted, At level 0, Numticks = 4
Dispatching -> 4 from level 0, numticks = 0
Dispatching -> 2 from level 0, numticks = 0
Yielding..
Dispatching -> 9 from level 0, numticks = 0
Pid = 9 to get preempted, At level 0, Numticks = 4
Dispatching -> 10 from level 0, numticks = 0
Pid = 10 to get preempted, At level 0, Numticks = 4
Dispatching -> 11 from level 0, numticks = 0
Pid = 11 to get preempted, At level 0, Numticks = 4
Dispatching -> 8 from level 0, numticks = 0
Dispatching -> 2 from level 0, numticks = 0
Yielding..
Dispatching -> 9 from level 0, numticks = 0
Pid = 9 to get preempted, At level 0, Numticks = 4
Pid = 9 to get preempted, At level 0, Numticks = 4
Dispatching -> 10 from level 0, numticks = 0
Pid = 10 to get preempted, At level 0, Numticks = 4
Dispatching -> 11 from level 0, numticks = 0
Pid = 11 to get preempted, At level 0, Numticks = 4
Dispatching -> 4 from level 0, numticks = 0
Dispatching -> 2 from level 0, numticks = 0
```

ML Adaptive

Test1

Similar to above we ran 4 process instances in background. These can be seen being pushed to last level with more time slice compared to RR.

The processes just do computations, hence consume CPU.

```
QEMU
Pid = 10 to get preempted, At level 3, Numticks = 7
Dispatching -> 9 from level 3, numticks = 0
Pid = 9 to get preempted, At level 3, Numticks = 7
Dispatching -> 11 from level 3, numticks = 0
Pid = 11 to get preempted, At level 3, Numticks = 7
Dispatching -> 12 from level 3, numticks = 0
Pid = 12 to get preempted, At level 3, Numticks = 7
Dispatching -> 2 from level 3, numticks = 0
Yielding..
Dispatching -> 10 from level 3, numticks = 0
Pid = 10 to get preempted, At level 3, Numticks = 7
Dispatching -> 9 from level 3, numticks = 0
Pid = 9 to get preempted, At level 3, Numticks = 7
Dispatching -> 11 from level 3, numticks = 0
Pid = 11 to get preempted, At level 3, Numticks = 7
Dispatching -> 12 from level 3, numticks = 0
Pid = 12 to get preempted, At level 3, Numticks = 7
Dispatching -> 2 from level 3, numticks = 0
Yielding..
Dispatching -> 10 from level 3, numticks = 0
Pid = 10 to get preempted, At level 3, Numticks = 7
Dispatching -> 9 from level 3, numticks = 0
Pid = 9 to get preempted, At level 3, Numticks = 7
Dispatching -> 11 from level 3, numticks = 0
```

In above we can see Idle process PID = 2 yielding the CPU as soon as it is scheduled.

Test 2

Here a single process was scheduled.

The process does computations for CPU burst and also has Get_Key calls for IO interrupt.

CPU burst consumes time slice causing the process to get pre-empted thus demotion.

IO blockage causes the process to get promoted as can be seen before the line containing KEY in the diagram below.

Promotion can demotion illustrates the working of ML Adaptive.

```
QEMU
nit process (/c/shell.exe)
Pid = 1 to get preempted, current at level 0
$ z2.exe
Dispatching -> 9 from level 0
Pid = 9 to get preempted, current at level 0
Dispatching -> 9 from level 1
Pid = 9 to get preempted, current at level 1
Dispatching -> 9 from level 2
Pid = 9 to get preempted, current at level 2
Dispatching -> 9 from level 3
Dispatching -> 9 from level 2
Key - f
Dispatching -> 9 from level 1
Key - a
Pid = 9 to get preempted, current at level 1
Dispatching -> 9 from level 2
Pid = 9 to get preempted, current at level 2
Dispatching -> 9 from level 3
Pid = 9 to get preempted, current at level 3
Dispatching -> 9 from level 3
Dispatching -> 9 from level 2
Key - a
Dispatching -> 9 from level 1
Key - a
$ _
```

HOW TO TEST

In order to test the scheduling mechanism following variables are of importance :

1. testMode in *kthread.c*- Set this variable to 1 when testing Scheduling policy. This allows important statements to be printed which show how the process is pre-empted or picked for scheduling
2. currentSchedulingMode in *kthread.c* - Set this variable to RRS to test for the Round Robin Scheduling, and MLS for MultiLevel scheduling.
3. Process with PID = 2 is the idle proces. In case you do not want to see the scheduling or pre-emption of this process set idleThreadMode = 0 in *kthread.c*

Make sure the system has printScreen option . This would be useful in taking snapShots at particular time instances.

This forms the necessary arrangement to start the testing.

We have provided three programs -

1. z1.c , z3.c :- After starting the geekOS . Run these two programs as z1 & and z3 &. These are to be run as background processes. If idle thread was not off then there would

be lot of messages printed on screen. The two processes have to be typed nevertheless on the screen even when messages are constantly pouring in. Take > 1 printScreens after these processes run and save them. Now you would be able to see messages related to processes being scheduled and preempted.

2. z2.c :- This program is meant to show the promotion of the processes. Run as z2. A message "Key" would be displaced. Type any key on such a message. This is a prompt for key event and hence a wait. Thus allowing the program to get promoted in MLS type of scheduling .

Note :-

For z1.c, z3.c : To see demotion turn off the idleThreadMode and make the count of iterations < 5. This is to see the messages in limited screen of the geekOS.

Aim :- To implement user level threads in GeekOS

Ideation - Bhargav, Ranveer

Implementation - Aditya , Nishant

Testing - Aditya

Main Idea:

In order to implement a user level thread we need to maintain a thread library and a list of thread_queue internally for a given process context. The thread library schedules the user-level threads. Scheduler has no knowledge of the inner threads.

So we added a pointer to queue of Kernel_Thread in Kernel_Thread struct which denotes the queue of processes with the same user context.

A function is added which schedules the inner kthreads. We can implement sophisticated techniques for scheduling here. MAX_THREAD_TICKS denotes the maximum clock ticks a single user thread can take.

Files that needed to be changed :-

kthread.h

In the struct ***Kernel_Thread*** , three new fields were added

- *int thread_pid* : The pid of the thread which is currently running
- *int* tot_threads* : To store the total number of user threads that have been spawned by that process
- *int thread_ticks* : To enable switching of the threads when the time slice of the thread expires
- Thread_Queue *thread_list : Maintain the queue of user level threads which are present under the same context

New functions:

schedule_user_library

Start_Child_User_Thread_Library

kthread.c

Contains the definition of the new functions declared in kthread.h

- *schedule_user_library* : This function is the one responsible for selecting the thread which is to be run currently. The scheduling that has currently been implemented is round robin scheduling with time slice. The user threads are invisible to outer scheduler. The main scheduler simply calls this function on each clock tick.
- *Start_Child_User_Thread_Library* : This function is very similar to *Start_Child_User_Thread* except that there is no call to *Make_Runnable*. It starts a new thread with the user context and stack same as that provided in the argument. The thread that has been created is not added to the ready queue.

timer.c

In the function *Timer_Interrupt_Handler*, a call has been made to *schedule_user_library* function which is responsible for internal scheduling of the user level threads.

Next, a syscall was added which enabled the program to spawn user level threads.

conio.h

A function *SpawnUserThread* was declared

conio.c

- *SpawnUserThread* : This function is responsible for handling the syscall to create a new user level thread
A call to the macro `DEF_SYSCALL` has been made with *SpawnUserThread* as the handler function and `SYS_SPAWNUSERTHREAD` as the syscall

syscall.h

A new syscall has been declared `SYS_SPAWNUSERTHREAD`

syscall.c

- *Sys_SpawnUserThread* : This function handles the syscall which creates user level threads. It creates a new thread by making a call to the function *Start_Child_User_Thread_Library*, with the current thread's user context as the argument. After creating the thread, it is added back to the current thread's thread queue. The pointers for `thread_list` and `tot_threads` have been copied from `CURRENT_THREAD`. The new thread's `thread_pid` has been appropriately set to the next number which is the total thread count of the current thread. The new thread's pid is made to be the same as that of current thread.

HOW TO TEST THE IMPLEMENTATION :

Preparation for testing :

1. set `testMode = 1`; in `kthread.c` : This allows for printing of important information related to switching of user level threads and scheduling by the geekOS scheduler
2. Make sure `printScreen` is available to capture the output screen in order to analyze the message printed

Here a file `z5.c` has been provided which creates 2 user level threads in a similar fashion as the KLT. The threads increment the count variable and wait in `while(1)`.

Run `z5.c` & Now take `printScreen > 1` and see the scheduling and switching of userlevel threads as well the main processes by the geekOS scheduling.

Note: To off the scheduling messages set `testMode = 0`. This shows output similar to the KLT.

To see the scheduling messages set testMode = 1. testMode is in kthread.c

Aim : Implementation of Sys_PS in GeekOS

Ideation - Bhargav

Implementation - Prateek

Testing - Ranveer

Files to change :

1. src/geekos/syscall.c : *static int Sys_PS(struct Interrupt_State *state)*

The Input to this system call is a pointer to user memory containing array of Process Info Structs in *state->ebx* & Length of this memory in *state->ecx*

Now , the complete printing of the PS function is an Atomic operation

Then We loop over all threads in all thread list using the function

Get_Front_Of_All_Thread_List(&s_allThreadList);

Before looping over them We print the line which is header of the PS command which is

Print("PID\tPPID\tPRIO\tSTAT\tCOMMAND\n")

Now for each of the Thread we get the info and print them. The info are fetched as:

- pid : *kthread->pid*
- parent_pid : if *kthread->owner* is a valid entry then parent_pid is *kthread->owner->pid* else the parent_id is 0
- status : A process can be in one of four states on its way from being alive to being dead:

1. refCount=0, alive=false

In this case, it's a zombie that's "totally dead," as the child has done Exit to reduce its refCount, and if it had a parent at all, it reduced its refCount too. Thus, the process will soon be reaped.

2. refCount=1, alive=false

In this case, the process has done Exit(), but the parent hasn't done Wait(). In this case, the process is also a zombie, but is not on the graveyard queue.

3. refCount=1, alive=true

In this case, the process is a background process, and is alive and well.

4. refCount=2, alive=true

In this case, the process is a foreground process, and is alive and well.

- priority : *kthread->priority*
- Name of Process : *kthread->threadName*

2. src/user/ps.c

In this file we changed it to create Process Info table and then We make a call to the system call named as PS which is defined in process.h

How to Test this

Start geekOS,
command “ps” on shell gives the list of current process of which geekOS kernel is aware.

Start process z1 &
type “ps” and see z1 added to the list of runnable processes in system

AIM : Detection of Deadlocks

ideation - prateek

Implementation + testing : Bhargav

Given a specification file, implement resource allocation and deadlock detection for SIMR (single instance multiple request).

Data to be maintained:

1. process name to pid map and vice versa
2. resource name to resource id map and vice versa
3. resource allocation table (size: #resources)
It tells which process id the resource was allocated to.
4. resource request queue (vector<queue<request> >, size of vector: #resources)
It gives the queue of requests waiting for a resource.
5. resource requested table (size: #processes * #resources)
Stores the info similar to resource request queue, it tells which process is waiting for which resource.
6. resource needed map
It tells which resources are needed by a process to run.
7. process start time (vector size: #processes)
Stores the process start time for each process. It stores -1 for process that hasn't started yet, -2 for a process that finished and positive integer denotes its start time.

8. priority queue of all requests.

Algorithm:

1. while(not all processes finished)
2. check for processes finishing (release their resources) or those that have to be started (initialize start time for them).
3. allocate resources to pending requests if possible. if allocated go to step 2.
4. pop a request from request queue if it's time and try to allocate resources to it. If allocation was successful go to step 2 again.
5. check for deadlock
6. increase timer count, go to step 1

Steps to run:

```
g++ -std=c++11 deadlock.cpp  
./a.out spec.txt
```

Example:

(spec.txt file) :-

P1,P2,P3

A,B,C

P1 1,4,-1

P2 -1,2,5

P3 6,-1,3

(Output):-

A allocated to P1 at 1

B allocated to P2 at 2

C allocated to P3 at 3

P1 requests B at 4

P2 requests C at 5

P3 requests A at 6

Deadlock detected among P1 P2 P3

AIM : The Sleep command

Ideation: Ranveer

Implementation: Ranveer + Nishant

Testing: Ranveer

The Idea

Basically, what we want to do here is, we want to implement a C/C++ styled sleep command on GeekOS. After a lot of ideation, the way we thought of doing this was through the GeekOS timer itself.

Implementation

We have a global variable called `g_numTicks`. Let us assume we want to implement a sleep for time = t_s . When sleep is called, the time (`g_numTicks`) is t_0 . Therefore until `g_numTicks` becomes $t_0 + t_s$, we would want the process to stop. Therefore, we designed a syscall which has a while loop that loops until `g_numTicks` becomes equal to $t_0 + t_s$. Now, since `g_numTicks` wouldn't increment in a while loop, we manually need to increment it. Which is what we did. So, in the while loop we have `g_numTicks++` which increments it, and the process is stopped till then.

In-code Details

All we needed to do in this part was to make a new syscall the usual way and implement the above. The syscall looks like this:

```
sys_sleep(long time)
{
    long tot_time = g_numTicks + time;
    while(g_numTicks < tot_time)
    {
        g_numTicks++;
    }
}
```

```
    }  
}
```

How to Test

To try out the sleep command, simply call `sleep(time);` in your code and the process sleeps for that amount of time.

eg code -> `z7.c`, Run as `z7.exe` to see effect of sleep