Brendan's Multi-tasking Tutorial

Contents [hide]

Overview

This tutorial will describe a way to implement multi-tasking and task switching for a kernel that uses "kernel stack per task". It has been designed to allow the reader to implement a full featured scheduler (while avoiding common pitfalls) in steps that can be tested before moving on to the next step.

Note: I use the word "task" as generic term to mean "the thing that a scheduler schedules". For operating systems that don't support multi-threading, a task might be a process. For operating systems that do support multi-threading a task would be a thread.

Step 1: Initialization and `switch_to_task()`

Thread Control Block

Start by creating some kind of data structure that will keep track of a task's information. Typically this is called a "process control block" or "thread control block". At a minimum, this structure needs to contain:

- The task's kernel stack top.

- The task's virtual address space (e.g. the value for CR3 on 80x86 systems).
- A "next task" field that can be used for multiple different linked lists of tasks later on.
- A "state" field (also to be used later).

```
struct thread_control_block
{
    void* esp;
    void* esp0;
    void* cr3;
    thread_control_block* next;
    uint8_t state;
};
```

It might also contain other/optional fields, like:

- the scheduling policy of the task
- the scheduling priority of the task
- the ID of the process the task/thread belongs to
- a "task name" string
- the amount of CPU time the task has consumed so far

Don't forget that you can come back and add fields to this structure whenever you like - you don't need to have everything initially.

Initialization

Once you've created an initial data structure for keeping track of a task's information; create an `initialize_multitasking` function. The idea here (initially) is that there's already a task that has been running since boot, and you only need to create the information for it by allocating some memory for the task's information structure that you just created and setting the fields in that structure as appropriate.

```
void initialize_multitasking();
```

Switching Tasks

Next; write a low-level function to switch from one task (that is assumed to be running kernel code) to another task (that is also assumed to be running kernel code).

Note that the low-level task switch code should be written in pure assembly. If it's written in inline assembly in the middle of a C function then the compiler may add its own "function prologue" and "function epilogue" code that changes the layout of the stack. This is important because when a new task is being created the kernel needs to put values on the new task's kernel stack to match the values that the `switch_to_task` expects to pop off of the new task's stack.

For example (NASM syntax, 32-bit 80x86, `cdecl` calling convention, single-CPU only/globals used for "per CPU" variables, no support for FPU/MMX/SSE/AVX, untested):

```
extern void switch_to_task(thread_control_block* next_thread);
extern thread_control_block* current_task_TCB;
```

```
;C declaration:
;   void switch_to_task(thread_control_block *next_thread);
;
;WARNING: Caller is expected to disable IRQs before calling, and enable IRQs again after function returns

switch_to_task:

    ;Save previous task's state

    ;Notes:
    ;  For cdecl; EAX, ECX, and EDX are already saved by the caller and don't need to be saved again
    ;  EIP is already saved on the stack by the caller's "CALL" instruction
    ;  The task isn't able to change CR3 so it doesn't need to be saved
    ;  Segment registers are constants (while running kernel code) so they don't need to be saved

    push ebx
    push esi
    push edi
    push ebp

    mov edi,[current_task_TCB]    ;edi = address of the previous task's "thread control block"
    mov [edi+TCB.ESP],esp         ;Save ESP for previous task's kernel stack in the thread's TCB

    ;Load next task's state

    mov esi,[esp+(4+1)*4]         ;esi = address of the next task's "thread control block" (parameter passed on stack)
    mov [current_task_TCB],esi    ;Current task's TCB is the next task TCB

    mov esp,[esi+TCB.ESP]         ;Load ESP for next task's kernel stack from the thread's TCB
    mov eax,[esi+TCB.CR3]         ;eax = address of page directory for next task
    mov ebx,[esi+TCB.ESP0]        ;ebx = address for the top of the next task's kernel stack
    mov [TSS.ESP0],ebx            ;Adjust the ESP0 field in the TSS (used by CPU for for CPL=3 -> CPL=0 privilege level changes)
    mov ecx,cr3                   ;ecx = previous task's virtual address space

    cmp eax,ecx                   ;Does the virtual address space need to being changed?
    je .doneVAS                   ; no, virtual address space is the same, so don't reload it and cause TLB flushes
    mov cr3,eax                   ; yes, load the next task's virtual address space
.doneVAS:

    pop ebp
    pop edi
```

```
42      pop esi
43      pop ebx
44
45      ret                             ;Load next task's EIP from its kernel
    stack
```

Creating New Tasks

Next; write a function to create a new kernel task. This is a little bit like the code you already wrote for "initialise_multitasking()" (because it would allocate some memory for the structure used to keep track of the task's information and fill it in); except that you will need to allocate a new kernel stack for the new task and put values on the new kernel stack to match the values that your "switch_to_task(task)" function expects to pop off the stack after switching to the task. Don't forget that the values on the new kernel stack will include a "return EIP" (the address that the "switch_to_task(task)" function will return to), which should come from an input parameter of the "create_kernel_task()" function (e.g. like "create_kernel_task(void * startingEIP)" except probably with a function pointer instead of a void pointer if you can remember the correct syntax for that).

Note that you can have a "task start up function" that is executed when a new task first gets CPU time and does a few initialisation things and then passes control to the task's normal code. In this case the new kernel stack will include a "return EIP" that contains the address of the "task start up function", plus an extra "return EIP" (for when the "task start up function" returns) that contains the address of the task itself (taken from an input parameter of the "create_kernel_task()" function). This is a little slower, but avoids the need for each task's code to do any initialisation itself.

Finally; test everything you've done for this step by creating a second kernel task; where the existing first kernel task has loop that uses "switch_to_task()" to switch to the new second kernel task, and the new second task has a loop that does "switch_to_task" to switch to the existing first task. If everything works well you'll end up with two tasks that constantly switch to each other. If you like you can do more testing (e.g. have lots of tasks where one task switches to the next and the last task switches back to the first). It's also fun to see out how many task switches per second your code is capable of.

Step 2: "Schedule()" Version 1

Having tasks that explicitly switch to each other is a little inconvenient, and it'd be nice if the kernel could automatically select a task to switch to next, so the next step is to write a function that does this.

Note: If you're confident in your abilities you might be tempted to skip this step and go directly to "Step 4: "Schedule()" version 2" below; but this step is so simple that skipping it won't save much time and it's probably better to get a little experience with "extremely simple" before moving on to "very simple".

The code to select which task to give CPU time to next has a huge effect on the nature and "feel" of the OS; and it's possible for this code to become very complicated for this reason. However, it's better to start simple ("learn to walk before you run") so we're going to start with the simplest (and most awful) code possible. More specifically, we're going to start with a cooperative round robin scheduler; and then (later on) we're going to replace it with something a little more complex (that is a lot less awful).

Note: Round robin is awful because when the OS is under heavy load and something happens you need to wait for all of those tasks to use the CPU before the OS can respond. For example, if there are 1234 tasks running and each one uses an average of 5 ms of CPU time, and the user presses a key; then it will take over 6 seconds for

the task that responds to the key press to get CPU time and the user will not be impressed with how laggy your OS is. Some people try to fix this problem by putting tasks that are unblocked at the start of the list, but that doesn't work well because a task can deliberately "block then unblock" to keep itself at the start of the list and hog all of the CPU time while preventing other tasks from getting any CPU time.

To implement a round robin scheduler, you mostly only need a circular linked list. For a circular linked list, each item in the list points to the next item in the list, and the last item in the list points back to the first item in the list. In other words, we just need a "next task" field to implement it; and when we created the data structure to keep track of a task's information we already included a field that can be used for this purpose.

Essentially; you just need to set the "next_task" field in the data structure for each task's information in both the code to initialise multi-tasking and the code to create a new task.

Once that's done, you can write a "schedule()" function that selects a task to switch to and then switches to the selected task. You only need single line of code to select the next task, like "selected_task = current_task_TCB->next_task;" and you already have a function to switch to a specific task; and both of these things can be combined. In other words, the "Schedule()" function can contain a single line of code like "switch_to_task(current_task_TCB->next_task);".

As soon as you've finished implementing "schedule()" you can test it using the kernel tasks you were using for testing earlier, just by replacing the calls to "switch_to_task()" with calls to "schedule()". If you had multiple additional kernel tasks with different code for each kernel task (so it could switch to the right next task) you can also delete the duplicates so that you end up with multiple additional kernel tasks executing the same code.

Step 3: Time Accounting (Optional)

Sooner or later you'll probably want to keep track of things like the total amount of CPU time each task has consumed (which can be used for things like calculating the average CPU load for each task, average time spent idle, etc).

Note: A timer is something that either generates an IRQ after a requested amount of time has passed, or generates an IRQ at a specific frequency. A counter is something that keeps a count that can be read whenever you like. A timer may not be a counter (it might not have a "current count" that you can read), and a counter may not be a timer (it might not be able to generate an IRQ).

To do this you'll need some kind of time counter, like the CPU's time stamp counter, HPET's main counter or ACPI's counter. If the computer is too old to have an actual counter you may need to implement one yourself using a timer (e.g. by configuring a timer to generate an IRQ at a specific frequency and then doing something like "ticks_since_boot++;" in the timer's IRQ handler).

For time accounting, when certain things happen you want to subtract the counter's current value from the value it had last time (to determine how much time elapsed since last time); then add the time elapsed to the amount of time that the current task has consumed and update the time the counter was read last, a little bit like this:

```
void update_time_used(void) {
    current_count = read_counter();
    elapsed = last_count - current_count;
    last_count = current_count;
    current_task_TCB->time_used += elapsed;
}
```

The first place this needs to be called is during task switches, to update the amount of time that the previously running task consumed before switching to the next task. For extra goodness "update_time_used()" can also be called when someone asks the kernel how much time a task has consumed, so that (e.g.) if a task has been running for 10 ms already then that 10 ms is included in the time a task has consumed.

The other place you might want to update the amount of time a task has consumed is immediately after the CPU changes from user-space code to kernel code and immediately before the CPU changes from kernel code to user-space code; so that you can keep track of "amount of time task spent running kernel code" and "amount of time task spent running user-space code" separately.

Note that it's a good idea to make the time accounting a little abstract and use a relatively high precision. For example, rather than just doing "current_task_TCB->time_used += elapsed;" you might convert "elapsed" into nanoseconds. That way (regardless of what kind of counter the computer supports and/or if you add support for more modern counters later) the time accounting will always use the same time scale.

Finally; test to make sure that your scheduler is keeping track of the time each task consumes correctly.

Step 4: Task States and "Schedule()" Version 2

For a real OS (regardless of which scheduling algorithm/s the scheduler uses) most tasks switches are caused because the currently running task has to wait for something to happen (data received from a different task or storage devices/file system or network, or time to pass, or for user to press a key or move the mouse, or for a mutex to become available, etc), or happen because something that a task was waiting for happened causing a task to unblock and preempt the currently running task.

To prepare for this, we want to add some kind of "state" to each task so that it's easy to see what a task is doing (e.g. if it's waiting and what it's waiting for).

For task's that aren't waiting; I find it easier (later, when you add support for blocking and unblocking tasks) to have a "running" state (when the task is currently using the CPU) and a separate "ready to run" state. The idea is that when a task is given CPU time it is removed from the scheduler's list of "ready to run" tasks and its state is changed from "ready to run" to "running"; and when a task stops using CPU time either it's state is changed back to "ready to run" and it is put back on the scheduler's list of "ready to run" tasks, or its state is changed to "waiting for something" and it is not put back on the scheduler's list of "ready to run" tasks.

Note that making this change will involve changing the scheduler's list of tasks from a circular linked list into a "non-circular" linked list; and (because we've been using a singly linked list and have been using the "current_task_TCB" variable as a temporary "start of linked list" variable) you will need to add two new global variables - one for the start of the linked list and one for the end of the linked list. Initially both of these variables will be NULL because there's no tasks on the list, when the first task is added both of the variables will be set to the first task's information (when there is only one task on the list it must be the first and last task), and when more tasks are added they just get appended to the list (like "last_task->next_task = new_task; last_task = new_task;").

After creating the new "first and last task on the scheduler's ready to run list" variables; update the function that initialises multi-tasking so that it sets the initial task's state to "running" (and delete any older code that put the initial task on the scheduler's list of tasks); and then update the code that creates new kernel tasks so that the new task's state is set to "ready to run" just before the new task is added to the scheduler's "ready to run" list.

The next change is to update the low-level "switch_to_task()" function, so that if and only if the previously running task is still in the "running" state the previously running task's state is changed from "running" to "ready to run" and

the previously running task is put back on the scheduler's list of "ready to run" tasks; and then set the state of the next task to be given CPU time to "running" (regardless of what state it was in before, and without removing it from any list).

Finally; the "schedule()" function needs to be changed so that it removes the task from the "ready to run" list before doing a task switch, making it something like this:

```
void schedule(void) {
    if( first_ready_to_run_task != NULL) {
        thread_control_block * task = first_ready_to_run_task;
        first_ready_to_run_task = task->next;
        switch_to_task(task);
    }
}
```

Note that if there's only one task you don't really want to do any task switches (because there's no other tasks to switch to), and in this case the task will be "running" and the list of "ready to run" tasks will be empty. The previous version of "schedule()" was a little less efficient (it would've done a task switch from one task to the same task).

Step 5: Race Conditions and Locking Version 1

So far we haven't had to care about things like race conditions and locking because its a cooperative scheduler and there's only one CPU. However, before the scheduler's code can progress much further we're going to need some way to avoid race conditions and ensure everything happens in a correct order.

When there's a single CPU, in theory, this can be done just by disabling IRQs in places that matter. For multiple CPUs this doesn't work (e.g. disabling IRQs on one CPU won't stop a different CPU from modifying the scheduler's list of "ready to run" threads at an unfortunate time) and you need something more robust (e.g. spinlock).

We won't be implementing support for multiple CPUs; but that doesn't mean we can't (and shouldn't) make it a little easier to add support for multiple CPUs much much later. For this reason I'm going to pretend that there's a "big scheduler lock" and implement functions to acquire and release this lock, but then I'm not going to actually have a lock and I'm just going to disable and enable IRQs instead.

Unfortunately there's a minor problem - what if IRQs are already disabled for some other reason? For assembly language this isn't a problem - you can PUSHFD to save the original flags and disable IRQs when you enter a critical section, and then POPFD the original flags when you leave the critical section. For higher level languages you can't just leave stuff on the stack like this because the compiler will get horribly confused and think that local variables are in the wrong places, etc.

To work around that problem, there's two alternatives - have a function that returns the old EFLAGS as an output parameter and disables IRQs, then another function that restores the old EFLAGS from an input parameter, where the caller has to store the old EFLAGS value somewhere itself. The other alternative is to keep a counter that keeps track of how many pieces of code have wanted IRQs disabled, where you'd increment the counter when disabling IRQs at the start of a critical section, and decrement the counter and only enable IRQs if the counter is zero at the end of a critical section.

The first option is probably more efficient; but we don't really care much about getting the highest performance possible at this stage, and because I want to make it look a little bit like a lock (to make it easier to support multiple

CPUs later) I'm going to use the second option. It might look like this:

```
int IRQ_disable_counter = 0;

void lock_scheduler(void) {
#ifndef SMP
    CLI();
    IRQ_disable_counter++;
#endif
}

void unlock_scheduler(void) {
#ifndef SMP
    IRQ_disable_counter--;
    if(IRQ_disable_counter == 0) {
        STI();
    }
#endif
}
```

Once that's done, you need to add a comment to your "schedule()" function and to your "switch_to_task()" function saying that the caller is responsible for making sure the scheduler is locked before calling.

Then you need to comply with the new comment by locking and unlocking the scheduler at the right places. Fortunately you only have a few kernel tasks that call "schedule()" directly; so you only need to add locking/unlocking to them. The end result might look a little like this:

```
void test_kernel_task(void) {
    // unlock_scheduler();

    for(;;) {
        lock_scheduler();
        schedule();
        unlock_scheduler();
    }
}
```

This can be a little confusing depending on your perspective. From the perspective of one task, it locks the scheduler, does a task switch, and then when it gets CPU time again it unlocks the scheduler; and this is fine (the scheduler is always unlocked after its locked). From the perspective of the CPU, one task locks the scheduler then the scheduler does a task switch and a completely different task unlocks the scheduler; and this is also fine (the scheduler is still unlocked after its locked).

Note that "one task locks the scheduler then the scheduler does a task switch and a completely different task unlocks the scheduler" means that one of the first things that a new task will need to do (when it gets CPU time) is unlock the scheduler's lock. If you decided to use a "task start up function" in Step 1, then the "task start up function" can unlock the schedulers lock before returning to the task's code, and if you decided not to do that then you'll need to have an "unlock_scheduler();" at the start of every kernel task.

Finally, you should test to make sure your multi-tasking still works the same.

Step 6: Blocking and Unblocking Tasks

When the currently running task blocks, the kernel will put the task on some other list or structure (e.g. a list of tasks that called "sleep()" and are waiting to be woken up, a list of tasks waiting to acquire a mutex, etc) and change the task's state to "blocked for whatever reason". When a task in unblocked it will be removed from whatever list or structure it was on, and either its state will be changed to "ready to run" and it'll be put back on the scheduler's "ready to run" list, or it's state will be changed to "running" and it won't be put on any list.

Rather than duplicating this logic in all of the places that block and unblock tasks; it's nice for the scheduler to provide functions that can be used for blocking and unblocking tasks.

Blocking a task is incredibly easy (mostly because we prepared for it in the last step by making sure a task in the "running" state is not on the scheduler's "ready to run" list). You only need to set the task's state and then find another task to run, which might look a little like this:

```
void block_task(int reason) {
    lock_scheduler();
    current_task_TCB->state = reason;
    schedule();
    unlock_scheduler();
}
```

Unblocking a task is a little more involved. The first step is to decide if the task being unblocked should (or shouldn't) preempt the currently running task. For now (because we don't have any fancy task priorities or anything) we could never preempt, or we could only preempt if there was only one running task (because it's likely that if there was only one running task that task has been able to use lots of CPU time anyway). For fun (and because it helps to show how preemption would work) I'm going to go with the latter option.

If the scheduler decides that no preemption happens, it just sets the unblocked task's state to "ready to run" and puts the task on the scheduler's "ready to run" list.

If the scheduler decides that preemption will happen, it just calls "switch_to_task()" to switch immediately to the unblocked task.

The code might look something like this:

```
void unblock_task(thread_control_block * task) {
    lock_scheduler();
    if(first_ready_to_run_task == NULL) {

        // Only one task was running before, so pre-empt

        switch_to_task(task);
    } else {
        // There's at least one task on the "ready to run" queue already, so
don't pre-empt

        last_ready_to_run_task->next = task;
        last_ready_to_run_task = task;
    }
```

```
    unlock_scheduler();
}
```

The other thing we'll need is a reason to block a task. For this, we can add the ability for a task to voluntarily pause itself until some other task un-pauses it, which is something that might be useful later. To implement this we only need to define a new task state (PAUSED).

Finally, you can test the blocking and unblocking by having one of the kernel tasks pause itself (e.g. "block_task(PAUSED);"), and then get the next kernel task that will be given CPU time to unpause the task that paused itself. Note that you will need to be a little careful here because you don't want to unpause the task before it pauses itself, but we're still using a cooperative round robin scheduler so it's easy to determine which order tasks will be given CPU time and avoid that problem.

Step 7: "Sleep()", "Nano_sleep()" and "Nano_sleep_until()"

Sometimes a task wants to wait for time to pass (e.g. maybe a text editor wants to create a backup of your unsaved work every 15 minutes, so it has a task that sleeps for 15 minutes and saves your work when it wakes up). Sometimes a task wants to wait for a very small amount of time to pass (e.g. maybe a device driver needs to reset its device and then wait for 20 milliseconds). For both of these cases you want to block the task so that it doesn't waste any CPU time, and then have a timer IRQ unblock the task when the requested amount of time has passed.

Of course it's impossible to guarantee that a task will be blocked for exactly the amount of time it requested; and even if it is possible there's no guarantee that the scheduler would give the task CPU time as soon as it's unblocked. The only guarantee that can be made is that the task will not be unblocked too early, and an OS should provide this guarantee (and should warn programmers that their task may be unblocked later than they wanted).

Sometimes a task wants to do something regularly (e.g. maybe it's displaying an animated icon and needs to update the icon every 250 ms). For this, using something like "nano_sleep()" doesn't work properly. If you have a loop that updates the icon then waits for 250 ms; what happens is that it takes a little time to update the icon and the OS doesn't guarantee you'll be blocked for exactly 250 ms (and CPU time used by other tasks will make the delay erratic), so the icon might be updated after 300 ms the first time, then after 600 ms the next time, an so on; and on average it might be updated every 500 ms. There's no way to fix this problem using something like "nano_sleep()" because it uses relative time (e.g. "250 ms from now").

To fix the problem you need a function that uses absolute time instead of relative time (e.g. "sleep until time since boot is >= 123456"). This might look a little like this:

```
    delay = 250000000;   // 250000000 nanoseconds is 250 milliseconds
    next_update = get_time_since_boot();
    for(;;) {
        do_something();
        next_update += delay;
        nano_sleep_until(next_update);
    }
```

For code like this, the time between calls to "do_something()" still varies, but if its possible to call "do_something" often enough you can guarantee that these variations will average out and that "do_something()" will be called

every 250 ms on average. Of course if there's a design flaw and "do_something()" takes too long (e.g. longer than 250 ms), or if there's so many other tasks consuming CPU time that this task can't keep up (which is a problem that can be fixed with task priorities that we don't have yet), then it isn't possible to call "do_something" often enough - if that happens, the "nano_sleep_until(next_update);" will do nothing (no delay at all) and "do_something()" will be called as often as possible, which is still the best behaviour anyone can expect in those conditions.

Fortunately, if you have "nano_sleep_until()" then you can use it to implement "nano_sleep()", like this:

```
void nano_sleep(uint64_t nanoseconds) {
    nano_sleep_until(get_time_since_boot() + nanoseconds);
}
```

In a similar way, if you have either "nano_sleep_until()" or "nano_sleep()" you can use them to implement "sleep()" - you only need to multiply seconds by 1000000000 to get nanoseconds.

What this means is that "nano_sleep_until()" is the only function that the scheduler really needs to care about.

To implement "nano_sleep_until()"; the general idea is to put the task onto a list of sleeping tasks and block the task; and then have some sort of timer (not a time counter like we used earlier, but a timer that has an IRQ) to check the list of tasks and unblock any tasks that need to be woken up.

The problem here is that the timer's IRQ handler might need to wake up several tasks at the same time, and as soon as it wakes up one task the scheduler might decide to do a task switch; and that task switch will make it impossible for the timer IRQ handler to wake up the other tasks at the right time. We need to fix that problem before implementing "nano_sleep_until()".

Of course this problem will happen on other places too (e.g. multiple tasks waiting for the same data from disk being unblocked at the same time, multiple tasks waiting for a "pthread_cond_broadcast()", etc). We want a generic solution that solves the problem everywhere, not just a hack for "nano_sleep_until()".

What we really want is a way to tell the scheduler to postpone any task switches until we're ready. More specifically, we want a way to tell the scheduler to postpone task switches while the kernel is holding any lock (and do the postponed task switch/es when we aren't holding a lock any more), and we want to do this for all locks including the scheduler's lock.

Step 8: Locking Version 2

To tell the scheduler to postpone task switches while we're holding one lock, you just need a "postpone task switches" flag. However, what if locks are nested (e.g. where you acquire two different locks, and then release one, then release the other)? To make it work for nested locks you need a counter to keep track of how many locks the CPU is currently holding.

Essentially; when you acquire a lock you increase the counter, and when you release the lock you decrease the counter; and task switches are postponed whenever the counter is non-zero.

So, how do you postpone task switches?

This is relatively easy - before doing a task switch (in both "schedule()" and "switch_to_task()") the scheduler checks if task switches are being postponed (by checking if the counter is zero or non-zero); and if task switches

are being postponed (counter is non-zero) the scheduler sets a "task switch was postponed" flag and doesn't do the task switch at all. Then, when a lock is released you decrement the counter and see if the counter became zero; and if the counter did become zero you check if the scheduler set the "task switch was postponed" flag, and if it was you clear it and ask the scheduler to do the postponed task switch by calling "schedule()" again.

We already have code for locking and unlocking the scheduler; but you can't really use the same lock that the scheduler uses because you don't want the scheduler to disable task switching every time it needs to do a task switch, so we're going to have a different lock for miscellaneous stuff instead. Ideally you'd have many locks, but that's overkill for a tutorial. Of course we still won't be adding support for multiple CPUs and we'll still be enabling/disabling IRQs instead of having an actual lock.

It might look a little like this:

```
int postpone_task_switches_counter = 0;
int task_switches_postponed_flag = 0;

void lock_stuff(void) {
#ifndef SMP
    CLI();
    IRQ_disable_counter++;
    postpone_task_switches_counter++;
#endif
}

void unlock_stuff(void) {
#ifndef SMP
    postpone_task_switches_counter--;
    if(postpone_task_switches_counter == 0) {
        if(task_switches_postponed_flag != 0) {
            task_switches_postponed_flag = 0;
            schedule();
        }
    }
    IRQ_disable_counter--;
    if(IRQ_disable_counter == 0) {
        STI();
    }
#endif
}
```

The "schedule()" function only needs a few lines to check "postpone_task_switches_counter" and set "task_switches_postponed_flag", and that might end up a looking like this:

```
void schedule(void) {
    if(postpone_task_switches_counter != 0) {
        task_switches_postponed_flag = 1;
        return;
    }
    if( first_ready_to_run_task != NULL) {
        thread_control_block * task = first_ready_to_run_task;
        first_ready_to_run_task = task->next;
        switch_to_task(task);
```

```
    }
}
```

You'd need to make the same little change at the start of the "switch_to_task()" function, but that's "architecture specific assembly". For 32-bit 80x86 it might just a few instructions inserted at the start of the function (before all other instructions), like:

```
    cmp dword [postpone_task_switches_counter],0
    je .continue
    mov dword [task_switches_postponed_flag],1
    ret

.continue:
```

Of course if you wanted to you could have a high level language wrapper that does this before calling the low level assembly function - either way is fine (I don't like wrappers much, but if you care about portability a wrapper would mean there's a little less code to change when porting).

Finally, test the code to make sure it still works!

Step 9: "Nano_sleep_until()" Again

Now that we've improved locking, we can acquire a lock (e.g. the scheduler's lock) and wake up lots of sleeping tasks with caring if that will or won't cause task switches (or if a task we unblock preempts the task we're using), and then simply release the lock; so we're ready to implement "nano_sleep_until()".

To start with we're going to need some kind of structure to keep track of sleeping tasks. Good ways to do this are complicated (e.g. "time buckets") and we don't want complicated, so let's use an unsorted linked list (which isn't great because we'll need to check every task on the list, but it will work and it can be replaced with something more complicated later).

The other thing I should explain is that there's two very different ways of using a timer. The easiest way is to use a timer that generates an IRQ at a fixed frequency. A much better way is to determine how long until the timer IRQ is needed next and then configure the timer to generate an IRQ at that time; because this gives you a lot less unnecessary IRQs and also gives you a lot better precision.

I'm going to use the easy way, not just because it's easier but also because we'll be using an unsorted list (which makes it harder to determine how long until a timer IRQ is needed next).

For "nano_sleep_until()" you'd store the time the task is supposed to wake up somewhere, put the task on the unsorted list, then block the task. You'll need to add a "wake up time" field to the data structure you're using to keep track of tasks for this, and you'll need to define a new task state ("SLEEPING"). The code might look like this:

```
thread_control_block * sleeping_task_list = NULL;

void nano_sleep_until(uint64_t when) {
    lock_stuff();

    // Make sure "when" hasn't already occured
```

```
    if(when < get_time_since_boot()) {
        unlock_scheduler();
        return;
    }

    // Set time when task should wake up

    current_task_TCB->sleep_expiry = when;

    // Add task to the start of the unsorted list of sleeping tasks

    current_task_TCB->next = sleeping_task_list;
    sleeping_task_list = current_task_TCB;

    unlock_stuff();

    // Find something else for the CPU to do

    block_task(SLEEPING);
}
```

Note that we're using the same "next" field in the data structure that the kernel uses to keep track of a task's information that the scheduler uses for its list of "ready to run" tasks. A task can't be ready to run and sleeping at the same time, so this is fine.

The code for the timer IRQ handler depends a little on which timer you're using. If you're using the PIT chip on 80x86 PCs, and the PIT chip has been set to a frequency of about 1000 Hz, then the timer IRQ handler might look like this:

```
uint64_t time_since_boot = 0;
uint64_t time_between_ticks = 1000000;  // 1000 Hz = 1 ms between ticks =
1000000 nanoseconds between ticks

void PIT_IRQ_handler(void) {
    thread_control_block * next_task;
    thread_control_block * this_task;

    lock_stuff();
    time_since_boot += time_between_ticks;

    // Move everything from the sleeping task list into a temporary variable and
make the sleeping task list empty

    next_task = sleeping_task_list;
    sleeping_task_list = NULL;

    // For each task, wake it up or put it back on the sleeping task list

    while(next_task != NULL) {
        this_task = next_task;
        next_task = this_task->next;

        if(this_task->sleep_expiry <= time_since_boot) {
            // Task needs to be woken up
```

```
            unblock_task(task);
        } else {
            // Task needs to be put back on the sleeping task list
            task->next = sleeping_task_list;
            sleeping_task_list = task;
        }
    }

    // Done, unlock the scheduler (and do any postponed task switches!)

    unlock_stuff();
}
```

Finally, test the code to make sure tasks block and unblock correctly when they call "sleep()" or "nano_sleep()" or "nano_sleep_until()". Please note that currently there's a minor problem with the scheduler - if all tasks are blocked waiting for something (either paused or sleeping) there won't be any tasks that the scheduler can give CPU time to so the scheduler will give CPU time to a task that is supposed to be blocked (which is bad!). For now, just make sure that at least one task is running.

Step 10: Idle Time

For a real OS, often there's nothing for a CPU to do because all tasks are blocked waiting for something (e.g. waiting or the user to press a key). There are two very different ways to handle this situation - either you have a special "idle task" that never blocks so that (as far as the scheduler is concerned) there is always a task that can be given CPU time; or you add code to the scheduler to wait until a task is unblocked.

The first way is easier initially. The problem is that modern CPUs have power management involving multiple power saving states, where deeper power saving states save more power but take longer to return to full speed. The normal solution is to progressively move to deeper sleeps states so that if the CPU is only idle for a short time it's only in a light sleep state and can return to full speed quickly, and if the CPU was idle for a long time it'll be in a deep sleep state and save lots of power. What this means is that for modern power management you need to keep track of how long a CPU has been idle, and that's hard to do when you have a simple idle task that doesn't know when the CPU starts being idle or stops being idle.

For this, I'm going to try to cover both options.

Idle Time With Idle Task

To implement an idle task the first thing you'll need is an idle task(!). We can already create new kernel tasks, we just need code for the task itself. This code can be a loop that does nothing; but most CPU have an instruction you can use to wait for something (e.g. the "HLT" instruction on 80x87 that waits until an IRQ occurs) that saves a little power, and in that case it's probably a good idea to use that instruction in the idle task's loop.

The code might look like this:

```
void kernel_idle_task(void) {
    for(;;) {
        HLT();
```

```
    }
}
```

The next problem is that currently we're still using an awful cooperative round robin scheduler, which means that if the scheduler gives the idle task any CPU time it might hog all of the CPU time even after other tasks wake up. What we really need is a scheduler that isn't awful - e.g. a scheduler that understands task priorities, so that we can give the idle task the lowest possible priority and let all the other tasks (which will have higher priorities) preempt the idle task.

However; we don't really want to implement a complicated scheduler yet. Instead, we can use a few temporary hacks to simulate an extremely low priority without actually implementing full support for task priorities.

For this, you'll need a variable to keep track of which task is the idle task. Then you'd modify the "unblock_task()" code so that the idle task is always pre-empted, like this:

```
void unblock_task(thread_control_block * task) {
    lock_scheduler();
    if( (first_ready_to_run_task == NULL) || (current_task_TCB == idle_task) ) {

        // Only one task was running before, or the idle task is currently
running, so pre-empt

        switch_to_task(task);

    } else {
        // There's at least one task on the "ready to run" queue already, so
don't pre-empt

        last_ready_to_run_task->next = task;
        last_ready_to_run_task = task;
    }
    unlock_scheduler();
}
```

The other thing you'd want to do is modify "schedule()" so that it tries to avoid giving the idle task CPU time when other task's are "ready to run":

```
void schedule(void) {
    if(postpone_task_switches_counter != 0) {
        task_switches_postponed_flag = 1;
        return;
    }
    if( first_ready_to_run_task != NULL) {
        thread_control_block * task = first_ready_to_run_task;
        first_ready_to_run_task = task->next;
        if(task == idle_task) {
            // Try to find an alternative to prevent the idle task getting CPU
time
            if( first_ready_to_run_task != NULL) {
                // Idle task was selected but other task's are "ready to run"
                task = first_ready_to_run_task;
                idle_task->next = task->next;
```

```
                first_ready_to_run_task = idle_task;
            } else if( current_task_TCB->state == RUNNING) {
                // No other tasks ready to run, but the currently running task
wasn't blocked and can keep running
                return;
            } else {
                // No other options - the idle task is the only task that can be
given CPU time
            }
        }
        switch_to_task(task);
    }
}
```

Note that this is a relatively crude hack; but that's fine for now - later we're going to replace the scheduler with something less awful anyway.

Idle Time Without Idle Task

In this case we need to add support for "no task running" in every piece of code that currently assumes "current_task_TCB" points to a valid task's information, so that we can set "current_task_TCB" to NULL when the CPU is idle.

The code for blocking a task should be fine (even though it does use "current_task_TCB") because the scheduler will never call this function when there's no task running; and the code for unblocking a task is fine.

The (optional) code for time accounting does need to be fixed though, and it'd be nice to keep track of how much time the CPU spend idle too, so that can be changed to something like this:

```
uint64_t CPU_idle_time = 0;

void update_time_used(void) {
    current_count = read_counter();
    elapsed = last_count - current_count;
    last_count = current_count;
    if(current_task_TCB == NULL) {
        // CPU is idle
        CPU_idle_time += elapsed;
    } else {
        current_task_TCB->time_used += elapsed;
    }
}
```

That only leaves the "schedule()" function itself, which could end up like this:

```
void schedule(void) {
    thread_control_block * task;

    if(postpone_task_switches_counter != 0) {
        task_switches_postponed_flag = 1;
        return;
```

```
    }

    if( first_ready_to_run_task != NULL) {
        task = first_ready_to_run_task;
        first_ready_to_run_task = task->next;
        switch_to_task(task);
    } else if( current_task_TCB->state == RUNNING) {
        // Do nothing, currently running task can keep running
    } else {

        // Currently running task blocked and there's no other tasks

        task = current_task_TCB;                            // "task" is the task
that we're borrowing while idle
        current_task_TCB = NULL;                            // Make sure other
code knows we're not actually running the task we borrowed
        uint64_t idle_start_time = get_time_since_boot();  // For future power
management support

        // Do nothing while waiting for a task to unblock and become "ready to
run".  The only thing that is going to update
        // first_ready_to_run_task is going to be from a timer IRQ (with a
single processor anyway), but interrupts are disabled.
        // The timer must be allowed to fire, but do not allow any task changes
to take place.  The task_switches_postponed_flag
        // must remain set to force the timer to return to this loop.


        do {
            STI();          // enable interrupts to allow the timer to fire
            HLT();          // halt and wait for the timer to fire
            CLI();          // disable interrupts again to see if there is
something to do
        } while( (first_ready_to_run_task == NULL);

        // Stop borrowing the task

        current_task_TCB = task;

        // Switch to the task that unblocked (unless the task that unblocked
happens to be the task we borrowed)

        task = first_ready_to_run_task;
        first_ready_to_run_task = task->next;
        if(task != current_task_TCB) {
            switch_to_task(task);
        }
    }
}
```

Step 11: Time Slices

Most schedulers have some kind of limit on the amount of time that a task can consume CPU time; which can mean terminating a task (if the kernel assumes that the task locked up) or switching to a different task as part of a

time sharing scheme.

Because we've been using an awful cooperative round robin scheduler anyway; the next logical step is to add time slices to convert it into an awful preemptive round robin scheduler.

For this you will need some kind of timer again, where the timer's IRQ handler just calls "schedule()" when a task has used all of the CPU time that the scheduler gave it. However, if you're using a timer set to a fixed frequency typically a relatively high frequency is used (e.g. 1000 Hz) to get better precision from it, and you don't want to do a task switch so often. In that case the easy solution is for the timer's IRQ handler to decrease a "time slice length remaining" variable and only do the task switch if the time has run out.

The other thing that might be worth mentioning is that (for most scheduling algorithms) when there's only one task that can be running there's point having the overhead of "schedule()" because it's only going to decide to let the currently running task keep running; and if you implemented idle time without a special idle task there might be no tasks running. For both of these situations there's no point having a time slice length. For that we can just use a special value (e.g. zero) to signify that there's no time slice length.

For the earlier example (using the PIT on 80x86 at a frequency of 1000 Hz) it might look like this:

```
uint64_t time_slice_remaining = 0;

uint64_t time_since_boot = 0;
uint64_t time_between_ticks = 1000000;  // 1000 Hz = 1 ms between ticks =
1000000 nanoseconds between ticks

void PIT_IRQ_handler(void) {
    thread_control_block * next_task;
    thread_control_block * this_task;

    lock_stuff();
    time_since_boot += time_between_ticks;

    // Move everything from the sleeping task list into a temporary variable and
make the sleeping task list empty

    next_task = sleeping_task_list;
    sleeping_task_list = NULL;

    // For each task, wake it up or put it back on the sleeping task list

    while(next_task != NULL) {
        this_task = next_task;
        next_task = this_task->next;

        if(this_task->sleep_expiry <= time_since_boot) {
            // Task needs to be woken up
            unblock_task(task);
        } else {
            // Task needs to be put back on the sleeping task list
            task->next = sleeping_task_list;
            sleeping_task_list = task;
        }
    }
```

```
    // Handle "end of time slice" preemption

    if(time_slice_remaining != 0) {
        // There is a time slice length
        if(time_slice_remaining <= time_between_ticks) {
            schedule();
        } else {
            time_slice_remaining -= time_between_ticks;
        }
    }

    // Done, unlock the scheduler (and do any postponed task switches!)

    unlock_stuff();
}
```

The other change that's needed is that the "time_slice_remaining" variable needs to be set when a task switch happens. This can be done in the low level task switch code (which is "architecture specific assembly").

If you added a wrapper earlier (when updating locking in Step 8), and if you implemented idle time using an idle task, then it might look like this:

```
#define TIME_SLICE_LENGTH  50000000   // 50000000 nanoseconds is 50 ms

switch_to_task_wrapper(thread_control_block *task) {
    if(postpone_task_switches_counter != 0) {
        task_switches_postponed_flag = 1;
        return;
    }

    if(task == idle_task) {
        time_slice_remaining = 0;
    } else {
        time_slice_remaining = TIME_SLICE_LENGTH;
    }
    switch_to_task(task);
}
```

If you added a wrapper earlier and implemented idle time without an idle task, then it might look like this:

```
#define TIME_SLICE_LENGTH  50000000   // 50000000 nanoseconds is 50 ms

switch_to_task_wrapper(thread_control_block *task) {
    if(postpone_task_switches_counter != 0) {
        task_switches_postponed_flag = 1;
        return;
    }

    if((current_task_TCB == NULL) {
        // Task unblocked and stopped us being idle, so only one task can be
running
        time_slice_remaining = 0;
    } else if( (first_ready_to_run_task == NULL) && (current_task_TCB->state !=
```

```
RUNNING) ) {
        // Currently running task blocked and the task we're switching to is the
only task left
        time_slice_remaining = 0;
    } else {
        // More than one task wants the CPU, so set a time slice length
        time_slice_remaining = TIME_SLICE_LENGTH;
    }
    switch_to_task(task);
}
```

If you're not using a wrapper, it shouldn't be too hard to convert the changes above into assembly and add them to your low-level task switch code (for both cases; it's just some comparisons, branches and moves).

Step 12: Task Termination

Tasks are terminated for a variety of reasons (e.g. they crashed, called "exit()", received a kill signal, etc); and sooner or later you'll want to provide a low-level function that the kernel can use to terminate a task for any of these reasons.

Terminating a task is a little tricky - you want to do things like free any memory that the task was using for its kernel stack; but you can't do that while the CPU is currently using the task's kernel stack. To work around that, you cheat - you don't terminate the task immediately and just set the task's state to "TERMINATED" and make the task block. A little later (when you're running some other task) there's no problem with freeing things like the terminated task's kernel stack because you know the CPU isn't using it.

So, how do you tell another task that it needs to clean up after a recently terminated task? Usually I write micro-kernels with message passing, and when a task is terminated I send a message to a "cleaner" to tell it the task was terminated, and then the "cleaner" can do the clean up work. We don't have any message passing (and don't have pipes or IPC of any kind yet) but what we do have is the ability to pause and unpause tasks (from Step 6), and as long as we're a little bit careful that's good enough.

The task termination might look like this:

```
thread_control_block *terminated_task_list = NULL;

void terminate_task(void) {

    // Note: Can do any harmless stuff here (close files, free memory in user-
space, ...) but there's none of that yet

    lock_stuff();

    // Put this task on the terminated task list

    lock_scheduler();
    current_task_TCB->next = terminated_task_list;
    terminated_task_list = current_task_TCB;
    unlock_scheduler();

    // Block this task (note: task switch will be postponed until scheduler lock
```

```
is released)

    block_task(TERMINATED);

    // Make sure the cleaner task isn't paused

    unblock_task(cleaner_task);

    // Unlock the scheduler's lock

    unlock_stuff();
}
```

Note that we're using the same "next" field in the data structure that the kernel uses to keep track of a task's information that the scheduler uses for its list of "ready to run" tasks (which is used for sleeping tasks too!). A task can't be ready to run and terminated at the same time, and a task can't be sleeping and terminated at the same time, so this is fine.

Of course this won't work without a "cleaner" task. The cleaner task might look a little bit like this:

```
void cleaner_task(void) {
    thread_control_block *task;

    lock_stuff();

    while(terminated_task_list != NULL) {
        task = terminated_task_list;
        terminated_task_list = task->next;
        cleanup_terminated_task(task);
    }

    block_task(PAUSED);
    unlock_stuff();
}

void cleanup_terminated_task(thread_control_block * task) {
        kfree(task->kernel_stack_top - KERNEL_STACK_SIZE);
        kfree(task);
}
```

Step 13: Mutexes and Semaphores

There are a lot of similarities between mutexes and semaphores - the both cause tasks to block when they can't acquire them and unblock a task when they can be acquired later, and they both are used for similar purpose (to restrict the number of tasks accessing a piece of data at the same time). The only real difference is the number of tasks that are allowed to access a piece of data at the same time - for mutexes this number is always one, and for semaphores the number is may be higher than 1.

In fact; if you really wanted to you could implement semaphores and then not bother implementing mutexes (and just use a semaphore with a count of 1 instead). However, this isn't really a great idea because you can optimise a mutex more(because you know it will only allow one task) and semaphores are rarely used (e.g. I have never

needed one, ever). Even though it's not a great idea, this is exactly what we're going to do - there's so many similarities that it will avoid code duplication, and we don't really care about performance much anyway!

For multi-CPU systems, a semaphore (or mutex) begins with data structure to keep track of its state, and a spinlock to protect that state. We're not writing code for multi-CPU systems so we're not going to have a spinlock; and we already have (at least conceptually) a "big scheduler lock" we can use to protect the semaphore's data; so we're going to cheat to keep things simple.

The first thing we'll want is the semaphore's structure itself, which mostly just needs a few fields for counts plus a linked list for waiting tasks. For the linked list of waiting tasks I want to make sure tasks are taken from the front and added to the back to ensure that one task can be very unlucky and end up waiting forever while other tasks are lucky and keep acquiring the semaphore, which is slightly more tricky with singly linked lists (you need a "last task" field and a "first task" field). The semaphore's structure can be something like this:

```
typedef struct {
    int max_count;
    int current_count;
    thread_control_block *first_waiting_task;
    thread_control_block *last_waiting_task;
} SEMAPHORE;
```

Next we'll want some functions to create a semaphore or a mutex, like this:

```
SEMAPHORE *create_semaphore(int max_count) {
    SEMAPHORE * semaphore;

    semaphore = kmalloc(sizeof(SEMAPHORE));
    if(semaphore != NULL) {
        semaphore->max_count = max_count;
        semaphore->current_count = 0;
        semaphore->first_waiting_task = NULL;
        semaphore->last_waiting_task = NULL;
    }
}

SEMAPHORE *create_mutex(void) {
    return create_semaphore(1);
}
```

With that done, we want some functions to acquire a semaphore or a mutex:

```
void acquire_semaphore(SEMAPHORE * semaphore) {
    lock_stuff();
    if(semaphore->current_count < semaphore->max_count) {
        // We can acquire now
        semaphore->current_count++;
    } else {
        // We have to wait
        current_task_TCB->next = NULL;
        if(semaphore->first_waiting_task == NULL) {
```

```
            semaphore->first_waiting_task = current_task_TCB;
        } else {
            semaphore->last_waiting_task->next = current_task_TCB;
        }
        semaphore->last_waiting_task = current_task_TCB;
        block_task(WAITING_FOR_LOCK);    // This task will be unblocked when it
can acquire the semaphore
    }
    unlock_stuff();
}

void acquire_mutex(SEMAPHORE * semaphore) {
    acquire_semaphore();
}
```

Lastly, we need to be able to release the semaphore or mutex:

```
void release_semaphore(SEMAPHORE * semaphore) {
    lock_stuff();

    if(semaphore->first_waiting_task != NULL) {
        // We need to wake up the first task that was waiting for the semaphore
        // Note: "semaphore->current_count" remains the same (this task leaves
and another task enters)

        thread_control_block *task = semaphore->first_waiting_task;
        semaphore->first_waiting_task = task->next;
        unblock_task(task);
    } else {
        // No tasks are waiting
        semaphore->current_count--;
    }
    unlock_stuff();
}

void release_mutex(SEMAPHORE * semaphore) {
    release_semaphore();
}
```

Note that we're using the same "next" field in the data structure that the kernel uses to keep track of a task's information that the scheduler uses for its list of "ready to run" tasks (which is used for sleeping tasks and terminated tasks too!). A task can't be in multiple states at the same time, so this is fine.

Step 14: Inter-process Communication (IPC)

There's a lot of different types of IPC (remote procedure calls, signals, pipes, synchronous messages, asynchronous messages, ...) but most of them have a few similarities:

- they aren't just used for communication between different processes (e.g. can be used for communication between different threads in the same process, and for for a monolithic kernel, this can include communication

between drivers in kernel space and tasks in user-space).

- they cause tasks to block (e.g. when waiting for data to arrive, and sometimes when waiting to send data) and unblock (when data arrives or data can be sent)
- often (especially for micro-kernels) it's responsible for the majority of task switches

I'm not going to provide examples of any of the different kinds of IPC - I want you to implement whatever you feel like having by yourself to get more experience. Note that there's multiple examples of "tasks blocked waiting on a list" in the previous steps of this tutorial.

Step 15: "Schedule()" Version 3

If you've got this far, you should have a fairly respectable scheduler with most of the features a scheduler in a modern OS needs. The main problem at the moment is that it's still an awful round robin scheduler. It's time to replace it with something better.

So, what is "better"?

It's impossible to answer that question without knowing what the OS will be used for and what kinds of tasks it will be running - an OS designed specifically for (e.g.) "hard real time" embedded systems (where the scheduler must provide strict guarantees) is very different to a (e.g.) an OS designed for scientific computer (where you expect a single task will hog all CPU time and the scheduler itself isn't very important).

However, for a general purpose OS that has to handle a wide range of different tasks with different requirements it becomes much easier to define "better" - for this case, "better" is something that is able to handle a wide range of different tasks with different requirements!

For this, you'd need to figure out what the different requirements are. Maybe you want to support "hard real-time" tasks (which is not recommended for beginners in general); and maybe you want to support "soft real-time" tasks (things like video decoders and sound systems that have to process data at a fixed rate to avoid dropping frames or causing audible glitches), and maybe you want to support "background tasks" that never get any CPU time unless there's nothing else the CPU can be used for.

Once you've decided what the different requirements are, you can define "scheduling policies" for each one. For an example; Linux uses four scheduling policies (SCHED_FIFO, SCHED_RR, SCHED_DEADLINE and SCHED_OTHER).

Then you'd need to figure out how the different policies relate to each other. The simplest way to do this is to assign them an order; where tasks using a lower policy only ever get CPU time when there are no tasks in a higher policy that want CPU time (and where "policy 0" might be sort of real-time policy and "policy 3" might be for background/idle tasks). In any case, you'll probably end up with a kind of "meta-scheduler" that decides which policy should be used. Once that's done, you need to choose a scheduling algorithm for each scheduling policy, and decide if each scheduling policy has (or doesn't have) task priorities within the policy, and decide on the rules that determine when one task (that's been created or unblocked) will preempt the currently running task.

For an full example (from a previous micro-kernel of mine), I used:

- Policy 0: Intended for low latency tasks (e.g. device driver IRQ handlers). Uses a "highest priority task that can run does run" scheduling algorithm, and tasks within this policy have a priority (0 to 255). These tasks will preempt all tasks in lower policies and all tasks with lower priority in the same policy.

- Policy 1: Intended for tasks with some latency requirements (e.g. user interfaces, GUIs, etc). Uses a "highest priority task that can run does run" scheduling algorithm, and tasks within this policy have a priority (0 to 255). These tasks will preempt all tasks in lower policies and all tasks with lower priority in the same policy.
- Policy 2: Intended for normal tasks. Uses a "variable frequency, fixed time slice" scheduling algorithm. Tasks within this policy have a priority (0 to 255) that determines how many (fixed length) time slices the task gets. Tasks will preempt all tasks in lower policies but will not preempt tasks in the same policy.
- Policy 3: Intended for background tasks. Uses a "variable time slice" scheduling algorithm, and tasks have a priority (0 to 255) that determines the length of the time slice they get. These tasks never preempt anything.

Once you've decided how your scheduler will work; you should only need to modify your "schedule()" and "unblock_task()" functions to make it happen.

Adding User-Space Support

Sooner or later you're probably going to want user-space tasks (e.g. processes and threads). This doesn't have much to do with the scheduler itself - it just means that while a kernel task is running it can switch between kernel and user-space where appropriate. It's important/useful to understand that when the CPU is running user-space code, something (system call, IRQ, exception, ...) will happen to cause the CPU to switch to kernel code, and then after kernel code is already running the kernel's code may decide to do a task switch. Because kernel code is always running before a task switch happens, kernel code will also be running after a task switch happens. In other words, task switches only ever happen between tasks running kernel code and other tasks running kernel code (and never involve a task running user-space code). This also means that the "switch_to_task()" code described in this tutorial (which is very fast because it doesn't save or load much) doesn't need to be changed when you add user-space support (until you start supporting things like FPU, MMX, SSE, AVX).

Typically; the main difficulty of adding user-space support is that you need to create a new virtual address space, have an executable loader (that might start the executable file directly, or might start a "user-space executable loader" that starts the executable file), create a new "first task for process" that includes a user-space stack, etc. I prefer to create a kernel task first and then do all this other work (to set up a process) while using the kernel task, because this works a lot better with task priorities (e.g. if a very high priority task creates a very low priority process, then most of the work needed to set up the low priority process is done by a low priority task, and the high priority task can continue without waiting for all that work).

Note: There are two common ways that processes can be started - spawning a new process from nothing, and "fork()". Forking is horribly inefficient and far more complicated (as you need to clone an existing process, typically including making all of the existing process' pages "copy on write", and then reversing all the work very soon after when the new child process calls "exec()"). For beginners I recommend not having "fork()" and just implementing a "spawn_process()" (and for people who are not beginners I recommend not having "fork()" because it's awful and has a risk of security problems - e.g. malicious code using "fork()" and then having its own implementation of "exec()" designed to inject malware into the new child process).

Once you have the ability to start a new process (including starting an initial task/thread for that process), adding a way to spawn a new task/thread for an existing process should be easy.

Adding FPU/MMX/SSE/AVX Support (80x86 only)

Originally (80386, single-CPU) Intel designed protected mode so that an OS can avoid saving/loading FPU state during task switches. The general idea was to keep track of an "FPU owner" and use a flag ("TS" in EFLAGS) to indicate when the currently running task isn't the FPU owner. If the CPU executes an instruction that uses FPU but the current task isn't the FPU owner, then the CPU raises an exception (because "TS" is set), and the exception handler saves the FPU state (belonging to a different task) and loads the FPU state for the currently running task. This can (in some cases) improve performance a lot - for example, if you have 100 tasks running where only one uses FPU, you'd never need to save or load FPU state. Intel continued this original idea when newer extensions (MMX, SSE, AVX) where added (although for AVX the implementation is significantly different).

However; when almost all tasks are using FPU/MMX/SSE/AVX state, it makes performance worse (due to the extra cost of an inevitable exception); and it doesn't work well for multi-CPU (where the currently running task's FPU state may still be in a completely different CPU).

The simplest solution is to always save and load FPU/MMX/SSE/AVX state during all task switches (and that would be a recommended starting point). There are also multiple more complex (and more efficient) ways of doing it; including always saving the previous task's FPU/MMX/SSE/AVX state during task switches if you know it was used but still postponing the work of load the next task's state until the task actually does need it; and including keeping track of "never used, sometimes used, always used" and pre-loading the FPU/MMX/SSE/AVX state when you know a task always uses it (or uses it often enough).

Categories: Pages using deprecated source tags | Tutorials | Multitasking