How to Build a Scheduler

April 23, 2009

1 How to Build a Scheduler

Up till now we have used a very simple loop such as

```
while(1)
  {
  taskA();
  taskB();
  taskC();
  time_delay();
  }
```

To make a very basic non-premptive, round-robin scheduler. We can call this a *static* scheduler because the order of the task execution will never change. In this section we will expand our basic scheduler to give it some new features:

- The ability of a task to halt itself.
- The ability of a task to sleep for a while.

1.1 Part I Using function pointers to start a task

Throught the course we are using the terms "task" and "function" almost interchangably. This is because we implement our tasks with C functions. How can we manipulate these functions so that they do not nescessarily execute in a fixed order?

We learned that the syntax:

```
type (*name)(type arg1, type arg2, ...)
```

indicates a function pointer called `name` which can point to any function having the same prototype. For example

```
int (*fp)(char x, int y, double z)
```

creates a function pointer `fp` which can point to any function which returns an `int` and has three arguments which are `char, int, double` respectively. With function pointers we can write code that can start an arbitrary function. For example:

Example: start function

```
void start_function(void (*functionPTR)() )
  {
     functionPTR();
  }
```

Now our basic scheduler could be written:

```
while(1)
  {
  start_function( taskA);
  start_function( taskB);
  start_function( taskC);
  time_delay();
 }
```

1.2 Part II Keeping lists of functions

For our scheduler to be smarter than the basic `while(1)` loop,. we need to be able to put functions into lists. That way we can keep track of which functions are ready, waiting, and inactive. One approach would be a single list of tasks, with a corresponding array of integers to determine which state the task is in. Although logically OK. This approach is not preferred at this point, because the scheduler will have to do a lot of looking through the list to find which tasks are in a certain state.

We can create an array of function pointers as follows:

```
#define  NUMBER_OF_FUNCTIONS     10

void (*list_of_functions[NUMBER_OF_FUNCTIONS])(void *p)
```

In the example above we created an array which could hold 10 function pointers. Each function pointer in the array must point to a function with the prototype:

```
void function(void *p)
```

This is a good prototype for tasks because they do not need to return a value and the `void` pointer parameter is a flexible way to pass a parameter to a task that is being created (although we will not use this much).

Here then is some pseudocode for a main program and scheduler function:

```
//function prototypes for tasks
void taskA(void *p);
void taskB(void *p);
```

```
...

// function prototype for scheduler
void scheduler();


#define NTASKS     4

//lets make the task list global
 // an array of function pointers, each one has a void* parameter.
 void (*readytasks[NTASKS]) (void *p);

main()
   {
   // initialize array of pointers to tasks
  readytasks[0] = taskA;
  readytasks[1] = taskB;
  ...
  readytasks[3]= NULL;  // NULL signals the last task in the list.

 // now start scheduler
  while(1) {
      scheduler();
      time_delay();  // tune or use timer for 1ms loop time
     }
 } // end of main
```

Here, we have set up an array to contain pointers to the task as a global variable (so that it will be visible to both main and the scheduler function (not shown). Then, inside `main()`, we have initialized each element of the array to point to one of our tasks. The scheduler function will do the following (pseudocode)

Scheduler function (pseudocode)

```
scheduler() {
  if(readytasks[task_index] == NULL && task_index != 0)  task_index=0;
  if(readytasks[task_index] == NULL && task_index == 0)   {
             // figure out something to do because there are no tasks to run!
             }
  start_function(readytasks[task_index]);
  task_index++; // Round Robin/we're taking turns
  return;
 }
```

1.3 part III Manipulating the Task Lists

So far, we have just made a more complicated form of our basic `while(1)` loop scheduler. Now let's allow tasks to halt themselves by calling a function called `halt_me()`. First, we need to generate a second list for the tasks which might be halted. Let's call it `haltedtasks[]`. Now, the scheduler can work as before, but when a task calls `halt_me()`, the following things have to happen:

`halt_me` function

```
halt_me() {
   * identify which task is currently running  (i.e. look at task_index)
   * copy the function pointer from readytasks[task_index]
              to the haltedtasks array
   * move the remaining tasks up in readytasks[] to fill the
              empty hole and copy NULL into the last element.
   * increment the index of the haltedtasks array.
  return;
  }
```

When `halt_me` returns to the task, the task should `return` and then it will not be started again by the scheduler since the current task has been deleted from `readytasks`

Sleep

Now let's try to set up a `Sleep(int d)` function which can let our tasks go to sleep for `d` msec. For reasons we can go into later, we need to implement sleep in a slightly awkward way. A function which wants to sleep will have to look like:

Typical Task using sleep()

```
task_c(p) {
   compute for a while;
   sleep(10);
   return;
   }
```

We will impose the requirement that the new `sleep(d)` function will not put the task to sleep until the task that called it has `return`ed. How would the scheduler work now? As above, we need to generate a new list for tasks which are waiting for time delays. Lets call it `waitingtasks[]`. Then we also have to make an array for the time delays for each wainting task. When a function calls `sleep(d)` the following things should happen

Sleep function (pseudocode)

```
sleep(int d) {
  * copy function pointer from readytasks[task_index]
          to the waitingtasks[] array.
  * clean up readytasks[] as in halt_me();
  * copy the d into the delays array with the same index
          as the function pointer has in waitingtasks[]
  }
```

Now the scheduler must keep track of waiting times for any tasks which are sleeping and perhaps wake them up by moving them back to the readytasks array. Let's add some pseudocode to the bottom of the scheduler function above:

Keeping track of sleep delays

```
...
* for each element of waitingtasks which is not NULL:
      decrement the delay value d.
      if (d==0) move the function pointer to the end
            of the readytasks[] array and remove it
            from waitingtasks.
// end of scheduler
  return;
}
```

1.4 An Alternate Data Structure for Scheduling

So far we have used arrays of function pointers to keep track of the tasks in each state. We also used an array of delay values for the remaining delay of sleeping tasks. Another approach is to keep everything about the task in a struct and then manipulate the structs to change task states. We call such structs Task Control Blocks (TCB's). For example, we could define the TCB as follows: Let's set up a struct which contains everything we need to track about a Task:

```
typedef struct TCBstruct {
    void (*ftpr)(void *p);     // the function pointer
    void *arg_ptr;             // the argument pointer
    unsigned short int  state;  // the task state
    unsigned int delay;        // sleep delay
    } tcb;
```

Now we have a single place that stores everything about a task. Now we can set up an array of `tcb`'s to hold all tasks and define constants for the states:

```
#define  STATE_RUNNING   0
```

```
#define  STATE_READY     1
#define  STATE_WAITING   2
#define  STATE_INACTIVE  3

tcb TaskList[N_MAX_TASKS];
```

Then we can set up the task list as follows:

```
int j=0;
int task_B_Arg;

TaskList[j].ftpr = task_A();
TaskList[j].arg_ptr = NULL;
TaskList[j].state = STATE_INACTIVE;
TaskList[j].delay = -1;
j++;

TaskList[j].ftpr = task_B();
task_B_Arg = 56;        // some arbitrary value
int *ip = &task_B_Arg;
TaskList[j].arg_ptr = (void*)ip;
TaskList[j].state = STATE_READY;
TaskList[j].delay = -1;
j++;

TaskList[j].fptr = NULL;

... etc ...
```

We have initialized the scheduler with two tasks, A and B. Their information goes into the first two entries in `TaskList[]`. Task A has no argument (more precisely the single void pointer argument will not be used), and starts out in `STATE_INACTIVE`. This means that it will have to be started by some other task. We want to start task B with an argument of 56, so we set an integer to 56, make a pointer which points to it, and then cast it to `void*` before assigning it to the `arg.ptr` member. Task B starts in `STATE_READY` so that it will go into the CPU as soon as possible. The next element of `TaskList[]` is set to NULL which will be our signal for no more tasks.

Now, `halt_me()` just has to figure out which element of `TaskList[]` it currently belongs to and change its state from `STATE_RUNNING` to `STATE_INACTIVE`. Similarly, if Task B called a function to `start` Task A, then that function would just have to change Task A's state from `STATE_INACTIVE` to `STATE_RUNNING`.

In Class Exercise: Starting, Stopping, Delay

Write pseudocode for `halt_me()`, `start_task(int task)`, and `delay(int d)` a task. Assume `t_curr` is an integer which points to the currently running task.

```
halt_me() {
    TaskList[t_curr].state = STATE_INACTIVE;
    }

start_task(int task_id) {
    TaskList[task_id].state = STATE_READY;
    }

delay(int d) {
    TaskList[t_curr].delay = d;
    TaskList[t_curr].state = STATE_WAITING;
    }
```