FTL Modula-2 Z80

Multi-tasking Kernel

The kernel gives you a means of running multi-tasking programs. In fact, using the versions of some of the standard modules that are also supplied, you can do this under CP/M and still use CP/M as well. However in order to do this you need a timer interupt that you can 'hook' into reliably. Sadly that are few CP/M systems that provide this.

The kernel is called Kernel. We were going to call it Saunter, since things run slower because of the multi-tasking overhead, but we decided against it.

The kernel uses a lot of ideas from operating systems principles. If you are not familiar with these principles, it will be worthwhile getting hold of a book on the subject. One book which I heartily recommend is:

Operating Systems, Design and Implementation, Andrew S Tanenbaum,Prentice-Hall International ISBN 0-13-637331-3

All you really need from this book is chapter 2. There are probably other books which go into the area of most interest in more detail but few of them will be as elegantly written.

Here is a simple program which uses the multi-tasking executive. We shall refer to this program in the text that follows:

```
1:MODULE TestKern;
2:
3:(*   Kernel test module 1
4:
5:     This module tests simple things like message
6:     passing and scheduling using semaphores
7:*)
8:
9:FROM Terminal IMPORT WriteString,WriteLn;
10:FROMKernel IMPORT LockSemaphore,UnlockSemaphore,
11:           InitSemaphore,Semaphore,SendMessage,
12:           ReceiveMessage,InitMailBox,MailBox,
13:    StartTasking,AddTask,StartDosCall,EndDosCall,Wait;
14:PROCEDURE Task1;
15:BEGIN
16:    LOOP
17:           WriteString( Task1);WriteLn;
18:           END;
19:    END Task1;
20:PROCEDURE Task2;
21:BEGIN
22:    LOOP
23:           WriteString(Task 2);WriteLn;
24:           Wait(5);
25:           END;
26:    END Task2;
27:PROCEDURE Task3;
28:BEGIN
29:    LOOP
30:           WriteString(Task 3);WriteLn;
31:           Wait(10);
32:           END;
33:    END Task3;
34:BEGIN (*set up tasks*)
35:    AddTask(Task1,1,1000);
36:    AddTask(Task2,1,1000);
37:    AddTask(Task3,1,1000);
38:    StartTasking;
39:    END TestKern.
```

Initialising the Tasks

To run a number of tasks, you must first define the tasks that are to be executed. You then start the multi-tasking kernel with a call to `StartTasking`.

Each task is defined by a call to `AddTask`. `AddTask` takes as parameters, a parameterless procedure, the priority of the task, and the amount of space (in bytes) to allocate for the tasks work area.

The highest priority tasks run at priority one. The lowest priority task runs at priority given by the constant `PriorityLevels`. There can be any number of tasks at a given priority.

You should ensure that cpu intensive tasks run at the lowest priority since otherwise they will hog the cpu and no lower priority task will ever execute.

In `TestKern`, the tasks are called `Task1`, `Task2` and `Task3`. They are set up in lines 35 through 37. Line 38 actually starts the execution of the tasks. Any code after the call to `StartTasking` will not be executed until the multi-tasking is terminated with a call to `EndTasking`.

The three tasks simply print out messages on the terminal. Before you link this program, make sure you replace your standard `Terminal` with `Terminal.MTX`. If you do not do this, the system will almost certainly hang up.

`Task1` runs continuously. Thus most lines contain the message `Task1`. `Task2` and `Task3` contain calls to the `Wait` procedure. This procedure delays the task for the given number of clock periods. It is called with one parameter; the number of clock periods to wait.

Hence, you will see the message `Task2` every 5 clock periods and the message `Task3` every ten clock periods.

Semaphores

Normally, you see one message per line. Sometimes, however, two messages come out on one line and then you get a blank line. For example, you might see:

```
Task1
Task1Task2

Task1
```

This happens if the scheduler time-slices from one task to another after the first task has performed the `WriteString` but before it has performed the following `WriteLn`.

We can prevent this from happening by using a *semaphore*. A *semaphore* is a means of locking a resource so that one task has exclusive use of the resource, in this case, the screen.

Here is a version of `Testkern` that overcomes the problem.

```
1:MODULE TestKern;
2:
3:(*   Kernel test module 1
4:
5:     This module tests simple things like message
6:     passing and scheduling using semaphores
7:*)
8:
9:FROM Terminal IMPORT WriteString,WriteLn;
10:FROMKernel IMPORT LockSemaphore,UnlockSemaphore,
11:           InitSemaphore,Semaphore,SendMessage,
12:           ReceiveMessage,InitMailBox,MailBox,Wait,
13:           StartTasking,AddTask,StartDosCall,EndDosCall
14:VAR ScreenSema:Semaphore;
```

```
14:PROCEDURE Task1;
15:BEGIN
16:     LOOP
17:             LockSemaphore(ScreenSema);
18:             WriteString(' Task1');WriteLn;
19:             UnlockSemaphore(ScreenSema);
20:             END;
21:     END Task1;
22:PROCEDURE Task2;
23:BEGIN
24:     LOOP
25:             LockSemaphore(ScreenSema);
26:             WriteString('Task 2');WriteLn;
27:             UnlockSemaphore(ScreenSema);
28:             Wait(5);
29:             END;
30:     END Task2;
31:PROCEDURE Task3;
32:BEGIN
33:     LOOP
34:             LockSemaphore(ScreenSema);
35:             WriteString('Task 3');WriteLn;
36:             UnlockSemaphore(ScreenSema);
31:             Wait(10);
32:             END;
33:     END Task3;
34:BEGIN (*set up tasks*)
35:     InitSemaphore(ScreenSema,1);
36:     AddTask(Task1,1,1000);
37:     AddTask(Task2,1,1000);
39:     AddTask(Task3,1,1000);
40:     StartTasking;
41:     END TestKern.
```

At line 35, we initialise the semaphore `ScreenSema`. The second parameter is the number of users that can simultaneously use the resource. This is usually one but it may be more than one.

Sometimes, you may also want to initialise a semaphore with the value zero. For example, if you were implementing a queue, you can initialise a semaphore to zero and use `UnlockSemaphore` every time you add an item to the list and `LockSemaphore` just before you remove an item from the list. This will be clearer in a minute.

Lines 34 and 36 are calls to `LockSemaphore` and `UnlockSemaphore`. A semaphore maintains a counter. This counter starts with the value you pass as the second parameter to `InitSemaphore`. When you call `LockSemaphore`, if the counter is greater than zero, it will be decremented by one and your task will continue as if nothing had happened.

If, however, the counter was zero, then the task is queued on the semaphore. The task stops executing and other tasks are scheduled.

When an `UnlockSemaphore` is executed, if there are no tasks queued on the semaphore, the counter is incremented by one. Thus, in `Task3`, if the counter was one when `LockSemaphore` was called, it will be set to zero while the `WriteString` and `WriteLn` statements are executed and then set back to one again, so that the value remains unchanged each time around the loop.

If there was a task queued, the counter is incremented and the queued task is re-scheduled. Once it regains control of the processor, the re-scheduled task will decrement the counter and will continue executing.

So, suppose that the time-slicer executes task 1 while task 3 is performing its `WriteString`. The counter in the semaphore `ScreenSema` will be zero, so task one will be queued and the kernel will look for another task to execute.

Perhaps this will be task 3. In any case, sooner or later, task 3 will get some CPU time.

Notice that task 1 will not be executed in the mean time because it is queued on the semaphore. It has been removed from the list of tasks that the time-slicer can execute.

Once task 3 has complete its `WriteLn` and has performed its `UnlockSemaphore`, task 1 will be back on the ready queue and can now be executed. Let us suppose that it gets some CPU time and performs its `LockSemaphore`. Now, if either task 2 or task 3 also attempt to lock the semaphore, they will be suspended until task 1 has finished executing its critical region.

Hence, this semaphore allows us to implement mutual exclusion. Only one task can be executing its `WriteString;WriteLn` pair at a time.

You can use a semaphore to protect any shared resource. For example, if you are accessing a global variable and updating its value, you do not want another task to be updating the same variable at the same time. A semaphore can be used to ensure that this will not happen.

Semaphores can protect large areas of code. The alternative is to execute the code with interrupts turned off. This alternative will run faster since the kernel is not required to re-schedule tasks, but it can cause your hardware to give spurious results, as often an interrupt must be acted upon quickly.

Message Passing

One way to pass data between tasks is to use common variables. This is fast since all that is needed is a few reads and writes of memory but it has two limitations:

1) The tasks that communicate have to be written as a group. Each task needs to know what the other does so that the passing of information is kept under control. This increases the complexity of programming.

2) It is difficult for a group of tasks to communicate with a single task. This is required, for example, if you write a general purpose task to do disk input-output.

The alternative is to use messages. The kernel supports message passing.

Messages are passed using mail boxes. A mail box is an address that is know both to the sender of the message and to the receiver. Often, a message will itself contain a mailbox so that the receiver can send a reply. Here is a simple message passing program.

```
1:MODULE TestMess;
2:
3:(*    Kernel test module 1
4:
5:      This module tests simple things like message
6:passing and scheduling using semaphores
7:*)
8:
9:FROM Terminal IMPORT WriteString,WriteLn;
10:FROMKernel IMPORT LockSemaphore,UnlockSemaphore,
11:            InitSemaphore,Semaphore,SendMessage,
12:            ReceiveMessage,InitMailBox,MailBox,Wait,
13:            StartTasking,AddTask,StartDosCall,EndDosCall
14:VAR Message,Message1:ARRAY [0..20] OF CHAR;
15:    Box305:MailBox;
16:    SentMessage:MailBox;
17:PROCEDURE Task1;
18:BEGIN
19:    LOOP
20:            WriteString(' Task1');WriteLn;
21:            ReceiveMessage(Box305,Message1);
22:            WriteString(Message1);WriteLn;
23:            SendMessage(SentMessage,Message1);
24:            END;
25:    END Task1;
```

```
26:PROCEDURE Task2;
27:VAR i:CARDINAL;
28:     Dummy:ARRAY[0..20] OF CHAR;
29:BEGIN
30:     Message:='Message 0001';
31:     LOOP
32:             WriteString(' Task2');WriteLn;
33:             SendMessage(Box305,Message);
34:             i:=11;
35:             WHILE Message[i]='9' DO
36:                     Message[i]:='0';
37:                     DEC(i)
38:                     END;
39:             Message[i]:=CHR(ORD(Message[i])+1);
40:             ReceiveMessage(SentMessage,Dummy);
41:             END;
42:     END Task2;
43:BEGIN
44:     InitMailBox(Box305);
45:     InitMailBox(SentMessage);
46:     AddTask(Task1,1,1000);
47:     AddTask(Task2,1,1000);
48:     WriteString(' starting tasking ');WriteLn;
49:     StartTasking;
50:     END TestMess.
```

This looks not dissimilar to the previous examples. Note the addition of the calls to `InitMailBox` at lines 44 and 45. We have initialized two mail boxes. `Box305` (if this number is not familiar, you haven't been paying attention) and `SentMessage`.

`Box305` is used to send messages. `SentMessage` is used to notify that a message has been received.

Task 2 just keeps sending messages to `Box305` (note, it is not sending them to task 1). It then waits for a message to appear in `SentMessage` before generating a new message and sending that.

Task 1 just keeps reading messages from `Box305` and then sending messages to `SentMessage`.

If no message is available when a `ReceiveMessage` is performed, the task is suspended until a message becomes available. You can avoid this by checking that a message is in the Mailbox using the call `MessageCount` from `Kernel`.

Having a separate mail box for replies is fine in the above example because there is only one task sending messages. If there is more than one task sending messages this will not work because they could get each others return messages. Each sender must have a separate mailbox for replies.

Usually, either a return mailbox, or a pointer to such a mail box is passed with each message. Passing just the pointer saves having to initialise a mailbox every time you send a message. It also allows the sender to send messages to a number of places and process replies as they arrive.

In the above example, the message was a text string. The contents of the string are totally undefined by `Kernel`. You can pass any form of data at all.

Interrupt Processing

The procedure `DoIO` is used for interrupts. You must use this procedure rather than calling `IOTRANSFER` directly so that the kernel can perform task scheduling while the interrupt is pending.

There is a potential problem with `DoIO` if an interrupt occurs while you are processing a previous interrupt from the same vector.

One way to overcome this is to attach two processes to the interrupt vector. The second process only gets called if you get an unexpected interrupt. Of course, you are still in trouble if you get two unexpected interrupts in rapid succession.

Procedure Index

Here is a complete list of the procedures and types in `Kernel` with their uses.

AddTask(p:PROC;prio:CARDINAL;Work:CARDINAL)

Add a task to the list of tasks to be executed. This can be done at any time. `p` is a parameterless procedure which is the start of the task. `prio` is the priority (in the range 1 to `PriorityLevel`) at which the task is to run, with 1 being the highest priority. `Work` is the size of the required work area in bytes.

DeleteTask(t:Task):BOOLEAN;

Delete a task. Returns TRUE if the task was successfully deleted. `t` is the task descriptor for the task to delete. If it is `CurrentTask` (a variable exported from `Kernel`), the task making the call is deleted and the call never returns.

Wait(Time:CARDINAL);

Suspend the current task for a period of time. `Time` is the time to wait in clock ticks.

Clock():CARDINAL;

Get the current time in clock ticks. Not very useful because it wraps around every 65536 ticks.

StartTasking

Starts the multi-tasking executive. This procedure does not return until after a call is made to `EndTasking`

EndTasking

End tasking. This procedure causes the termination of multi-tasking and the continuation of the code after the call to `StartTasking`. You can call `StartTasking` again after a call to `EndTasking`. You should modify `EndTasking` for your application by turning off any interrupts that are pending and replacing the interrupt vectors with the default values.

StartDosCall

Locks the semaphore associated with DOS. DOS calls should only be made between a call to `StartDosCall` and a call to `EndDosCall`. The module `Terminal.MTX` has already been set up to do this.

EndDosCall

End a dos call or sequence of dos calls.

MailBox

The type to be used for a mail box for message passing.

InitMailBox(VAR b:MailBox);

Initilaise a mailbox. Must be called before the mailbox is used for message passing.

SendMessage(VAR b:MailBox;Message:ARRAY OF BYTE);

Send a message to the mailbox `b`. The message can be any data; the structure is entirely defined by the application. A copy of the message is made.

ReceiveMessage(VAR b:MailBox;VAR Message:ARRAY OF BYTE);

Receive a message from the mailbox `b`. If no message is available, the task is suspended until a message becomes available. The message is removed from the mailbox.

MessageCount(VAR b:MailBox):INTEGER;

Return the number of messages in a mailbox. Returns a negative value if there are tasks waiting for messages.

Semaphore

A type used to enforce mutual exclusion.

InitSemaphore(VAR s:Semaphore;SimUsers:CARDINAL);

Initilaise a semaphore. `SimUsers` is the maximum number of tasks that can use the critical region protected by this semaphore at any time.

LockSemaphore(VAR s:Semaphore);

Lock the critical region. The task may be suspended until the critical region is available. There may be a number of tasks waiting for a given semaphore.

UnlockSemaphore(VAR s:Semaphore);

Unlock the critical region. If another task is suspended waiting for the region, it is now re-scheduled.

Waiters(VAR s:Semaphore):INTEGER;

return the number of tasks waiting for a semaphore. If the returned value is negative, the semaphore is free.

CurrentTask

The task descriptor for the currently executing task. Do not update this variable. It should be treated as read-only.

IORecord

A record type used for passing input-output requests to the `DoIO` routine. In the supplied version, the only value is the number of an interrupt vector that the task wants to wait for.

DoIO(IOInfo:IORecord;VAR PreviousTask:Task);

Wait for an interrupt. `IOInfo` contains the number of the interrupt to wait for. `PreviousTask` should be `NIL` the first time you call `DoIO`. It returns the task descriptor of the task that was interrupted. If you do not set this to `NIL` before calling `DoIO` for the next interrupt, that task will be started up again. Otherwise, if it is set to `NIL`, a re-scheduling operation is performed and the interrupted task may not regain control until later.

An Example System

In this section, I shall describe a simple process control system and show its implementation using the kernel. This is a simplified version of a system we once implemented, though that was done in Basic because it was to run on an Hewlett Packard 80 series calculator. Needless to say, it would have been much easier in Modula-2.

A loading bay consists of a conveyor belt from a stock-pile, an input hopper, a weighing hopper and an exit chute. The system is used to weigh a dry product into trucks. There is a position sensor that detects when a truck is present.

The conveyor belt can be turned on and off under program control. Sensors detect when the hopper is nearly empty and nearly full. There are gates between the hopper and the weighing hopper and between the weighing hopper and the truck.

The system clearly generates three interrupts; Hopper full, hopper empty and truck present/absent. Each interrupt needs a task to process it.

We shall assume that all devices as controlled by a single output address. This is fairly typical since only one bit is needed for each gate and for the conveyor belt motor. Because we can only write this register, not read it, we need to keep a shadow value in memory which always reflects the value in the external register. So that two tasks cannot access this shadow register at once, we need to protect it by a semaphore. This is all handled by the module `ControlReg` in the `Example` module.

The system is assumed to use three interrupts, one for changes in status of each of the feed hopper, the weigh hopper and the truck. When a change of status occurs the relevant task checks the new status and takes appropriate action.

```
MODULE Example;

FROM    PortIO IMPORT In,Out;
FROM    Kernel IMPORT Semaphore,LockSemaphore,
               UnlockSemaphore,InitSemaphore,DoIO,Wait;
CONST   HopperIntNo=8;
        HopperStatusReg=30h;
        HopperControlReg=38h;
        WeighIntNo=10h;
        TruckIntNo=18h;

        ConveyorBeltRunning=0;
        HopperGateOpen:=1;
        WeighGateOpen:=2;
        (*      status bits for gates etc *)

        TruckPresent=0;
        HopperFullFlag=1;
        HopperEmptyFlag=2;
        WeighHopperFullFlag=3;
        WeighHopperEmptyFlag=4;
VAR     WeighWaiting:BOOLEAN;
MODULE ControlReg;
IMPORT HopperControlReg,Semaphore,LockSemaphore,
        UnlockSemaphore,InitSemaphore,Out;
EXPORT SetBit,ClearBit;

VAR     Shadow:BITSET;
        MutualExclusion:Semaphore;

PROCEDURE SetBit(i:CARDINAL);
BEGIN
        LockSemaphore(MutualExclusion);
        INCL(Shadow,i);
        Out(HopperControlReg,Shadow);
        UnlockSemaphore(MutualExclusion);
        END SetBit;
```

```
PROCEDURE ClearBit(i:CARDINAL);
BEGIN
        LockSemaphore(MutualExclusion);
        EXCL(Shadow,i);
        Out(HopperControlReg,Shadow);
        UnlockSemaphore(MutualExclusion);
        END ClearBit;
BEGIN   (*mainline of ControlReg module*)
        Shadow:=BITSET{};          (*hopper gate shut,
stopped *)                         weigh gate shut, motor
        Out(HopperControlReg,Shadow);
        InitSemaphore(MutualExclusion,1);
        END ControlReg;
PROCEDURE HopperTask;
VAR     io:IORecord;
        p:Task;
        b:BITSET;
BEGIN
        LOOP
                p:=NIL;
                io.IntNo.=8;
                DoIO(io,p);
                (*      at this point, the status of the
                hopper has changed. See if empty or full
                and set controls accordingly *)

                b:=In(HopperStatusReg);
                IF HopperFullFlag IN b THEN
                        ClearBit(ConveyorBeltRunning);
                ELSIF HopperEmptyFlag IN b THEN
                        SetBit(ConveyerBeltRunning) END;
                END;
        END HopperTask;
```

```
PROCEDURE WeighTask;
VAR     io:IORecord;
        p:Task;
        b:BITSET;
BEGIN
        LOOP
                p:=NIL;
                io.IntNo:=WeighIntNo;
                DoIO(io,p);
                (*      at this point, the status of the
                        hopper has changed. See if empty or
                        full and set controls accordingly *)
                b:=In(HopperStatusReg);
                IF WeighFullFlag IN b THEN
                        ClearBit(HopperGateOpen);
                        Wait(10); (*give chance to shut*)
                        IF TruckPresent IN b THEN
                                SetBit(WeighGateOpen);
                        ELSE
                                WeighWaiting:=TRUE
                                END;
                ELSIF WeighEmptyFlag IN b THEN
                        ClearBit(WeighGateOpen);
                        Wait(10);
                        SetBit(HoppergateOpen);
                        END;
                END;
        END WeighTask;
```

```
PROCEDURE TruckTask;
VAR     io:IORecord;
        p:Task;
        b:BITSET;
BEGIN
        LOOP
                p:=NIL;
                io.IntNo:=TruckIntNo;
                DoIO(io,p);
                (*      at this point, the status of the
                hopper has changed. See if empty or full
                and set controls accordingly *)
                b:=In(HopperStatusReg);
                IF TruckPresent IN b THEN
                        IF WeighWaiting THEN
                                SetBit(WeighGateOpen);
                                WeighWaiting:=FALSE;
                                END;
                ELSE
                        ClearBit(WeighGateOpen);
                        END;
                END;
```

This example has been kept deliberately simple, because of this the code as it stands does not take full advantage of the multi-tasking facilities. However, there are several things it does not do. Consider how adding these things is easier when building on this multi-tasking structure than when using a simple program.

1) It does not check that actions requested have occured. For example, in a real system, it would be necessary to have position sensors on the gates and to check that requests to open and shut a gate had been satisfied before proceeding to the next step.

2) A real system would weigh out known amounts. Several weighings would be required for this (for example the truck might take 20 tonnes while the hopper only holds 500 kilograms). And a `WeighTask` would needed to control this.

3) A real system would have alarm conditions, such as stuck gates and product hang-up which would require alarms and operator intervention. The operator would typically have a number of controls which would need an interface.

In addition, there is not a lot of inherent parallelism in this system. The only parallelism is the ability to fill the supply hopper while the truck is being filled. Hence the control of the conveyor and the control of the lower gate on the weighing bin are totally separate. If you attempt to write a single-tasking program to do the job, all things which can happen at once need to be considered together and this produces a combinatorial explosion in the size of your code.

The Old School, Greenfield,
Bedford, MK45 5DE
Tel: (0525) 718181