Non-Preemptive Multitasking

by Joe Bartel, Hawthorne Technology

Multitasking, where one computer appears to be executing more than one program at the same time, is needed in many control applications. Usually a terminal or some kind of keyboard and display is communicating with the operator while a process is being monitored and/or controlled in the background. This is different from common programs being written for a PC where only one thing is happening at a time and one job is finished before the next job starts.

For every problem there is usually a simple solution and a complex solution. There are many multiuser, multitasking operating systems on the market today, of which Unix is just one example. There are also many small real-time kernels available on the market that implement preemptive multitasking without file management. If only some of the features of a full multitasking system are needed, then a special purpose program can do the job without many of the costs and problems that occur with a full generalized solution. By having a model of a nonpreemptive multitask system to follow, it is easier to conceptualize a solution and plan a program for a specific application.

How Preemptive Multitasking Works

In a normal or preemptive multitask system there is a scheduler module that takes over and takes control of the system away from one task and gives it to another task. The task swap can be in response to an interrupt from an external device or the result of a condition set by another task. The most common event is requesting I/O, which causes the task to wait until the request is filled. A task may also be swapped because the time allotted to it has expired and it is time for the next task to run awhile.

Programs start to get complicated when a task can be interrupted at any point instead of running to completion. First, any shared routines have to be reentrant. That means it must be possible for a second task to start executing the routine before the first task has finished its execution of the routine. Any data areas that are shared by two or more tasks must have locks and semaphores to prevent two tasks from trying to update the shared data at the same time and corrupting the data. Because the task swapping routine doesn't know anything about the task that is running it must save all the registers and status.

Most generalized multitasking systems have the ability to start new tasks and to delete old tasks. This requires sophisticated memory management to allocate and reclaim memory, and makes the control tables more complex because they can change in size. For many control multitask applications there is a fixed number of tasks and fixed memory allocation. The number of tasks is decided when the program is written and doesn't change when it runs.

The Value of Non Preemptive Multitasking

A nonpreemptive multitask system is one where the tasks cooperate with each other. A task runs for a while, long enough to accomplish something useful but not too long, and then it voluntarily gives up control of the system to the next ready task. Each task is also expected to be careful and not trash areas used by the other tasks in the system.

If a task cannot be swapped with another task without permission, then many things become much simpler. First you don't need the elaborate safeguards to prevent an update from being interrupted because the task will not be giving up control during the update process. Second you don't have to make shared routines reenterant because no other task will enter the routine until the using task is finished. Because a task knows it will be swapped out, the routine that gives up control can save any information that needs to be saved — the swap routine needs to save only the minimum.

For many applications, the use of nonpreemptive multitasking means that an existing single user/single task operating system can be used for character I/O and for disk file management. In our case this means the use of K-OS ONE, but the techniques will also work with CP/M and MS-DOS. A single task operating system is usually much less expensive, less complex, and smaller than a multiuser system. A single user system is also much more available.

In any multitask system it is important not to waste time that could be used by other tasks. There are two main areas where time gets wasted. One is waiting in a loop for a character from the console or for an I/O device to become ready. In this case the TESTIO command should be used to check for an available character and give up the machine to the next task if there is no character available. Another waste of time is a timing loop. To avoid this, the system time command should be used and if it is not yet time to act, then give up the system to the next task.

Implementation

Now that you are sold on the advantages of using a nonpreemptive multitask scheme how is it done? As simple as possible, that's how. Since everyone with K-OS ONE has an HTPL compiler, I will use that language to demonstrate the concept. The same principles can be used to multitask in almost any language but it will be more complex because more registers have to be saved and reloaded with each context switch. You need to find out which registers are used by the language you are using when you write the swap routine. The routines to start the system and to switch to the next task are prototypes that you change to fit your special application. The examples show how to use the task switching.

The program presented here is in several parts. There are three routines needed in your multitasking system. There are two HTPL routines, PROGRAM and NEXT, and one assembly language routine SWAPTSK. These are shown in Listing number 1. The versions here are for a system that has three tasks. The main program gets things started. NEXT changes to the next task.

SAMPLE CODE

```
;------- HTPL CODE FOR MULTITASK      \
;------- SWAP TASK  ( OLD NEW -- )
SWAPTSK MOVE.L  (A4)+,A1        ;NEW POINTER
        MOVE.L  (A4)+,A0        ;OLD POINTER
        MOVE.L  (A7)+,A3        ;RETURN
        MOVE.L  A4,(A0)+        ;SAVE PARAM PNTR
        MOVE.L  A7,(A0)         ;SAVE STACK PNTR
        MOVE.L  (A1)+,A4        ;NEW PARAM PNTR
        MOVE.L  (A1),A7         ;NEW STACK PNTR
        JMP     (A3)            ;RETURN TO CALLER
;-----------------------------------------------


( HTPL NONPREEMPTIVE MULTITASK )

( these must be contiguous and in order )
long tsk0p tsk0r tsk1p tsk1r tsk2p tsk2r tsk3p tsk3r ;
long tskpn tskid ;

program
  512 allocate 512 + !tsk1p
  512 allocate 508 + #task1 !4 +4 !tsk1r
  512 allocate 512 + !tsk2p
  512 allocate 508 + #task2 !4 +4 !tsk2r
  512 alloacte 512 + !tsk3p
  512 allocate 508 + #task3 !4 +4 !tsk3r
  #tsk1p !tskpn    1 !tskid
  #tsk0p @tskpn swaptsk ( start system )
  return
  end

proc next
  @tskpn #tsk0p swaptsk   ( save old task )
  @tskid +1 dup !tskid
    if 3 > then 1 !tskid #tsk0p !tskpn end
  @tskpn 8 + !tskpn
  #tsk0p @tskpn swaptsk   ( start new task )
  end

( ---- user interface ---- )
proc task1
  repeat
    repeat next chkchr until
    upcase case
      [ 'A' ]  ( action for command A here )
      [ 'B' ]  ( action for command B here )
      [ 'C' ]  ( action for command C here )
      end
    false until
  end

( --- timed sequence task --- )
proc task2
  while true do  next
    gettime !now ( get current time )
    if @now @target >= then
      doaction ( do timed action )
      @target @interval !target end ( set new time target )
  end end

( --- generalized task --- )
proc task3
  ( any repeating loop goes here )
  end
```

SWAPTSK is used by next to change tasks.

The first part of the system is PROGRAM where the HTPL program will start executing. There needs to be a startup routine that gets each task ready to run. It will only execute once when the program starts. Its purpose is to initialize the tasks, the tables used to switch tasks, and start the first task. Because HTPL uses two stacks, space for the stacks must be allocated. The main program allocates space for the parameter stack for each task. Next the return stack space is allocated. The starting address for the task is placed at the top of the return stack that has been allocated. To start the process going the program calls SWAPTSK with a pointer to TSK0P where its own parameter stack pointer and its return pointer will be saved. The other pointer is to where it will find the parameter stack pointer and return stack pointer for the first task. When the return statement is executed then the first task starts.

The second part of the system is the routine that transfers control from one task to the next task. This routine, which we call simply NEXT, looks like an ordinary procedure call to the task giving up control. The difference is that it doesn't use any arguments and it doesn't return any results. The NEXT routine uses SWAPTSK twice, once to get into the scheduler and once to get out of it. If the task stack is used for the scheduler routine then it would be possible to rewrite this to use SWAPTSK only once. If there are routines that use variables and are used by more than one task, then the values of the variables should also be swapped. It is important to realize that this is a generic prototype routine that needs to be customized for the specific set of tasks for which it will be used.

The final and smallest part is SWAPTSK (swap task), a routine that needs to be added to the runtime library HTPLRTL or linked in as an include file. The SWAPTSK is in assembler because it is much easier to do that way. The first job of the SWAPTSK routine is to save the state of an HTPL routine. In this context, only the parameter stack pointer and the return stack pointers need to be saved. The routine takes two parameters. First is a pointer to a place to save the old task parameter pointer and return pointer. Second is a pointer to a place to get the parameter stack and return pointer for the new task. The SWAPTSK routine returns to the routine that called it. This leaves the return address of the routine that called NEXT as the return address on the old return stack. Because all the tasks are written together in a single HTPL program all the other registers are either the same or do not need to be saved because all computations are done on the stack.

Using the Multitasker

Each task in the multitasking system needs to call NEXT at regular intervals to let the other tasks have some time to run. Most programs have a major wait loop where they get a character or wait for time to pass. If there is no wait loop, then any place in the main loop of a routine will do. It is also important to be aware of approximately how much time is used for each routine. Try to keep the time between calls to NEXT short and uniform. In many cases it is neccessary to fine tune the tasks that are running by changing where NEXT is called to get a smoother running system.

The check character procedure (chckchr) is an example of how a procedure that returns a variable number of arguments is used.

repeat next chckchr until

If there is a character available, the character and a true indicator are returned. If there is no character available then a false is returned. This causes the loop to continue until a character is available which it returns on the stack. This allows processing to occur in the background while waiting for the operator to input a character. After the input character is processed, then the wait loop can be entered to wait for the next character.

In most cases there is no need to wait to output characters. The BIOS on the HT-68k has a 256 character output queue. If this doesn't become full then there is no wait for output because the operating system merely places characters in the output queue. If there are several tasks running and a lot of characters need to be sent out, then a small number should be sent each time the task sending the characters is active. This allows the other tasks to run while K-OS ONE sends the characters from the output queue under interrupt control. If a larger buffer is needed then the BIOS

(Continued on page 28)

text substitution facility like #define in C.

I've always been mystified by the fact that some high level languages don't have certain high level features that are included in the most primative of assemblers. Until just recently, BASIC did not allow symbols for branch or call addresses, and C does not have symbolic constants that assemblers and other languages have. Text substitution is a perverse way to simulate symbolic constants (because of the huge time penalty). I was also shocked when I learned that C did not check the parameter list of the function definition when compiling a call to a function. In my first kludgy experiment in compilation of functions, I would not have dreamed of neglecting this kind of checking. Of course, you would have to draw and quarter me to make me do run-time checking.

Here's a summary of the current state of 32HL: It's a complete native assembler and assembles itself in one pass. It calls Z80 CP/M to get source text from a CP/M file. It has its own command loop and executes statements (assembly or otherwise) that are typed at the keyboard. It has character, string, and numeric I/O routines that are callable by client programs. I now have the disassembler integrated and STOP/CONT implemented, but not breakpoints or the high level words. The transcendentals and complex transcendentals are written and waiting to be installed. I'm now blacksmithing the expression parser to fit the new environment. When I get the expression parser in place, all the high-level keywords will fall in almost automatically.

Since I'm only a hobbyist running open loop, I welcome any comments about what I'm doing wrong or what would be neat to include.

Those interested in 32000 hacking should also contact Richard Rodman, 1923 Anderson Road, Falls Church VA 22043. He has started a small newsletter for 32000 hackers, which he currently gives free of charge. It is to be a forum on the creation of a public domain operating system, and a focal point for exchange of 32000 schematics, software, and information. He is starting an inventory of P.D. software that anyone can get for $8 a disk. The first volume contains my cross assembler, my Z80 assembler, and 32000 source code for transcendentals, complex math, complex transcendentals, a disassembler, and a math-intensive orbital game program in assembly. If anyone wants to see my old FORTH/BASIC system or my half-done 32HL system, I can send those for $8 each on CP/M SSSD 8" or on IBM 360k format (the IBM format for mailing only; no IBM code). ■

Neil R. Koozer
Kellogg Star Route Box 125
Oakland, Oregon 97462
(503)-459-3709

Multitasking

(Continued from page 38)

can be edited and reassembled with a larger queue. At 9600 baud, 960 characters can be sent each second. At low baud rates, fewer can be sent. Send small groups of characters so that the rate of sending will not exceed the rate of the hardware, causing a wait from the system.

A second common control action is for the program to do specific actions at regular time intervals. These actions might be taking a reading or creating some kind of a log record. The get time command gets the elapsed time from when the system started. The number has no meaning per se, but elapsed time can be measured by comparing the results of two different calls. Another way is to use a software timer with a flag, and check the flag to see if the correct amount of time has elapsed.

The routines presented here can be used as a starting point for multitasking process applications. They can help simplify what might otherwise have been a complex or unsovable problem. Even though the examples given use HTPL and K-OS ONE, it is possible to use the same techniques with other operating systems or for dedicated systems that don't have an operating system installed. Also, these techniques are a much lower cost solution than using a real time kernel.

■ ■ ■