Preemptive Multithreading using Timer

Preemptive Scheduling

- Ability to switch thread even w/out thread-yield
- Mechanism: timer interrupt
  - hardware interrupts current thread, executes ISR
  - ISR saves current state, picks new thread, resumes it
- Issues
  - Granularity of timer (how often?)
  - Auto or manual reload? fixed interval or variable?
  - Other sources of interrupt? Atomic region

Timers on 8051

- 8051 has two timers: T0, T1
  - T0 accessed as TL0 (lower), TH0 (higher)
    T1 accessed as TL1, TH1 (SFRs)
  - Resolution: 1/12 of crystal oscillator frequency
    e.g., 12MHz XTAL => 1μs timer unit
- T1 is used as baud generator for UART
  => Use T0 for preemption timer
  - Options: 8-bit, 13-bit, 16-bit

How to use Timer T0

- Configure Timer Mode (0, 1, 2, or 3)
  - Whether to start by software or hardware trigger
- Load starting val into register pair TH0:TL0.
  - To delay *x* cycles, load negative *x*
- Start running: (from software), SETB TR0
- Polling: check flag TF0 for roll-over
- Stop running: (from software): CLR TR0

SFRs for Timer Control

Timer 1	Timer 0	purpose
TMOD<7:4>	TMOD<3:0>	timer mode
TH1, TL1	TH0, TL0	high/low bytes for timer value
TR1 (=TCON.6)	TR0 (=TCON.4)	start(1), stop(0) ('R' => "run")
TF1 (=TCON.7)	TF0 (=TCON.5)	rollover flag ('F' => "flag")

TCON register (timer control)

- TF1, TF0 => "flag"
- TR1, TR0 => "run"
- IE1, IE0 => "interrupt enable"
- IT1, IT0 =>

TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

TMOD register

- gate = 0 for software control, (=1 for hw)
  - set TR0 = 1 to run, = 0 to stop
- c/t = 0 for timer, (= 1 for counter)
- <M1:M0> = mode 0, 1, 2, or 3

Timer 1				Timer 0			
gate	c/t	M1	M0	gate	c/t	M1	M0

TMOD.7 TMOD.0

<M1:M0> in TMOD

- 00: Mode 0: 13-bit timer ($2^{13} = 8192$)
- 01: Mode 1: 16-bit timer ($2^{16} = 65536$)
- 10: Mode 2: 8-bit auto-reload ($2^{8} = 256$)
- 11: Mode 3: split timer
  (two 8-bit timers or one 8-bit counter)

Timer 1				Timer 0			
gate	c/t	M1	M0	gate	c/t	M1	M0

TMOD.7 TMOD.0

Timer-1 mode 0 or mode 1

- Mode 0: 13-bit timer; Mode 1: 16-bit timer
  - loaded into TL0,TH0
  - SETB TR0 to start timer
- Count-up timer
  - upon rollover from FFFF (16-bit) or 1FFF (13-bit) to 0000, hardware sets the TF0 flag (Timer Flag)
  - => opportunity for interrupt!
  - If polling, need to clear TF0; if interrupt, TF0 auto-cleared
  - To stop running the timer, CLR TR0

Setting up interrupt for Timer 0, mode 1, 10K cycles

- Configure timer-0 for mode 1
  - MOV TMOD, #1

gate	c/t	M1	M0	gate	c/t	M1	M0
0	0	0	0	0	0	0	1

- Load the starting value into <TH0:TL0>
  - to count for 10000 cycles, load 65536-10000 = 0xd8f0.

```
MOV TH0, #0D8H  ;; need to manually reload if you want 10K
MOV TL0, #0F0H   ;; or else it starts from 0 up to FFFF
```

- Enable timer-0 interrupt by
  - MOV IE, #82H

EA	--	ET2	ES	ET1	EX1	**ET0**	EX0
1	0	0	0	0	0	1	0

- Start timer 0 ("run")
  - SETB TR0

Setting up interrupt for Timer 0, mode 0, 8192 cycles

- Configure timer-0 for mode 0
  - MOV TMOD, #0

gate	c/t	M1	M0	gate	c/t	M1	M0
0	0	0	0	0	0	0	1

- 8192 just wraps around
  - MOV TH0, #00H ;; 13-bit range, no need to reload
    MOV TL0, #00H ;; wraps back to 0 after 1FFFH
- Enable timer-0 interrupt by
  - MOV IE, #82H

EA	--	ET2	ES	ET1	EX1	**ET0**	EX0
1	0	0	0	0	0	1	0

- Start timer 0 ("run")
  - SETB TR0

ISR for Timer 0

- Interrupt vector at 000BH
  - either ISR code fits in 8 bytes or jump elsewhere
    => most likely jump elsewhere
- Interrupt => flag TF0 is auto-cleared!!
  - hardware automatically clears it when calling ISR;
- but if polling => user needs to clear TF0
- May want to reload the timer if quantum is not the full range of counter

How to write ISR in SDCC

- void timer0_ISR(void) __interrupt(1) {
  { __asm
    ljmp _myTimer0Handler
    __endasm; }
  }
  - this __interrupt(1) code must be in the same source file as your main() function!
- Have it jump to your own ISR code
  - myTimer0Handler can be a function in another .c file
    => this is where you can do the context switch!!
  - it must return using RETI instruction

From Cooperative to Preemptive Threading

- Cooperative thread i
  - calls ThreadYield() --pushes return address on thread i's stack
- Cooperative ThreadYield()
  - save state
  - switch to thread j
  - restore state
  - RET from ThreadYield

- Preemptible thread i
  - timer0's interrupt pushes return address on thread i's stack
- Preemptive version of ThreadYield()
  - like nonpreemptive, but **disable interrupt** on entry, **reenable interrupt** on exit!
- myTimer0Handler() for preemptive threading
  - version 1: same as ThreadYield, except it must do RETI to return!
  - version 2: improved version that does context switch outside ISR

Issues with simple preemptive version

- Some functions should not be preempted! (all versions)
  - ThreadYield() and myTimer0Handler()
  - ThreadCreate()
  - ThreadExit()
- Need to minimize time spent in ISR, or time when interrupt disabled (future version)
  - could cause other critical events to be missed!
- Need to make scheduler more modular (future version)
  - scheduler is currently hardwired in dispatcher, but want a more modular structure to try different policies!

What about SDCC constructs for Supporting synchronization?

- __critical directive
  - annotation on either a function or statement block
  - effectively same as EA = 0 on entry, EA = 1 on exit
  - good for ThreadCreate, ThreadYield, etc
  - but careful! If you have to do your own RETI, it may not work!
    - Advice: don't use it unless you can verify what it does by looking at assembly!
- __naked directive
  - suppresses register saving/restore; may be appropriate for ISR
  - cancels out the __critical on function definition.
- __using(*bankNumber*) directive
  - specifies use of register bank for the ISR... but doesn't do anything..

Convert to preemptive: file: testpreempt.c

- Producer:

```
while(dataAvail) { }  // take out ThreadYield()
__critical {
   // update your shared vars
}
```

- Consumer:
  - similar, take out call to ThreadYield(), add __critical around shared update
  - TMOD |= 0x20; // OR in the bits instead of writing

```
void timer0_ISR(void) __interrupt(1) {
        __asm
              ljmp _myTimer0Handler
        __endasm;
} // must be in same file as main() for SDCC to work!
```

Convert to preemptive: file: preemptive.c

- void Bootstrap(void) {
  threadCount = 0; threadMask = 0;
  TMOD = 0; // timer 0 mode 0
  IE = 0x82; // enable timer 0 interrupt,
  TR0 = 1; // start running timer0
  ThreadCreate(main);
  currentThread = 0;
  RESTORESTATE;
}

- ThreadID ThreadCreate(FunctionPtr fp)
  __critical {
  }
- add __critical to other Thread related functions

- void myTimer0Handler(void) {
  EA = 0; // don't do __critial
  ...
  EA = 1;
  __asm
  reti
  __endasm;
}