CS/ECE 6780/5780

Al Davis

Today's topics:

- Threads
  - basic control block
  - scheduling
  - semaphores
- Midterm (next Tues) covers Chaps & Labs 1-5
  - sample on the web

Lab 5 Logistics

- Problem – not enough interrupt pins
  - our mistake for not noticing this
- Solution
  - sense a change on the column pins – send to IRQ
  - run the column pins to input ports
  - on interrupt
    - » ISR checks the input ports to see what happened
  - iffy bit
    - » do we have the discrete logic in stock to pull this off?
      - checking

Implicit Threads

- We've already seen them in a sense
  - main (foreground thread) & ISR's (background threads)
    - » hardware support for interrupts and control path change
    - » on IRQ or XIRQ control is handed to the ISR
      - RTI returns control back to main
  - 3 common types
    - » input – some input triggers IRQ or XIRQ
    - » output – some "ready to receive" signal acts as the trigger
    - » periodic – periodic: employ a timer based interrupt
- Often this is enough
  - when applications are primarily I or O directed
    - » typical when system is small
      - ISR's do most of the work
      - main is just there to wait for an event to happen
- Larger projects w/ multiple modules
  - single foreground thread becomes more of a limitation
    - » so we'll focus on multiple foreground thread issues today

Explicit Thread Semantics

- In general CPU land
  - threads share memory
    - » threads are concurrent
      - hence shared memory access require ordering
  - threads have private registers and stack
  - thread scheduling
    - » supported by a multi-threaded OS scheduler
- In embedded land
  - microcontrollers do not have multi-threading support in the hardware
  - hence threads become independent control flows in concept
    - » but only one is running at any one time
  - all resources are shared
  - OS may not do the scheduling for you
    - » necessary "OS" functions may be in application code

Thread Model Illustrated

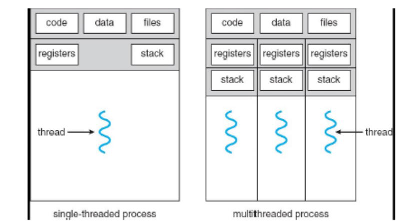
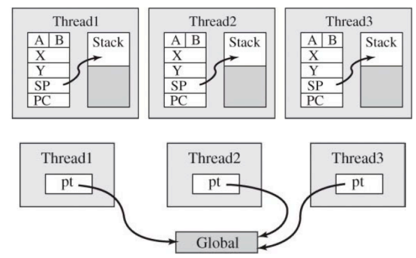


Figure is copyright Silberschatz, Galvin, and Gagne, 2005. (http://www.os-book.com)

Private vs. Shared Resources

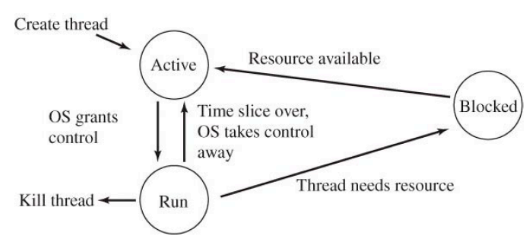
- **Illusion or reality?**
  - **6812: it's an illusion**
    - » **trick is to implement the abstraction to make it real**
    - » **all physical resources are global**



Why Use Threads?

- **Can improve program responsiveness**
  - **if done right and there is a functional need**
  - **similar to interrupt model**
- **Can improve program modularity**
  - **each threads function is self-contained**
  - **inherent decoupling from other thread actions**
  - **still need to appropriately synchronize the shared resources**
    - » **only one thread is every running at a time on the 6812**
      - **hence mutual exclusion is guaranteed**
        - – **but ordering and state save/restore become critical issues**
- **Blocking is a convenient abstraction for programmers**
  - **thread blocks when it needs something that is not available**
    - » **could be a physical resource**
    - » **or it could be based on time**
      - **e.g. thread gets a certain amount of time to be active**

3 Thread States



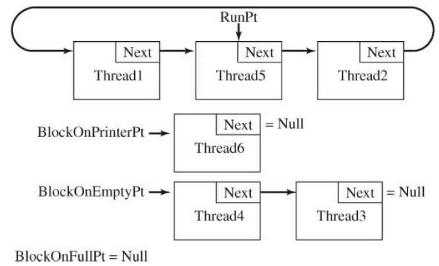
OS for ES systems may just be a scheduler

Thread Management

- **Fully featured CPU**
  - you don't have to worry about it too much
  - just use normal thread semantics (Pthreads for example)
    - » and let the OS do it's thing
- **Single threaded hardware such as the 6812**
  - then thread management has to be done in software
  - common tactic → linked lists
    - » single running thread → single RunPT
      - pointer into the list of ACTIVE threads
        - – RunPT points to the only one that is in RUN state
        - – rest in ACTIVE state
    - » other lists for BLOCKED threads
      - may be useful to have multiple BLOCKED lists
        - – one for each resource that is causing the blocked state

Simple Printer Example

- **Threads send output to a printer**
  - threads get input from a FIFO
    - » we're ignoring print order here
      - printf debug scenario – each print statement "Tn is at xx"



Scheduling

- **Process to determine which thread to run next**
  - decision points
    - » when RUN→BLOCK state change is made
    - » or when threads are created or killed
- **Scheduler types**
  - Nonpreemptive
    - » when new thread is chosen ONLY when
      - the current thread terminates or blocks
  - Preemptive
    - » scheduler may choose to run a new thread when the current thread is still active.
    - » can result in more responsive systems
      - but require more programmer effort to create a correct system

Scheduling Metrics

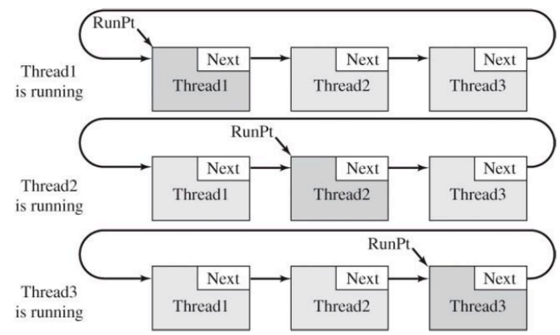
- **Minimize CPU utilization**
  - → minimizing busy waiting
- **Maximize throughput**
  - complete the most thread jobs per unit of time
    - » common metric for web servers
- **Minimize turnaround time**
  - minimize time from job request to job done
- **Minimize wait time**
  - e.g. minimize the time in the ACTIVE state
- **Minimize response time**
  - time from job request to job ACTIVE
- **Maintain QoS guarantee**
  - critical in real time systems

see any conflicting constraints? What's missing?
what do you care about – average, per thread, …?

Scheduling Strategies/Policies

- In order
  - simple donut shop mode – first come first served
    - » which metrics does this strategy serve?
- Shortest job first
  - how do you determine job length?
- Priority
  - based on what
    - » deadline
    - » simple predetermined value
    - » others?
- Round-Robin
  - simple yet reasonably fair policy
- Multi-Level Queue & Hybrids
  - e.g. priority levels
    - » within each level – round robin, shortest job first, ...

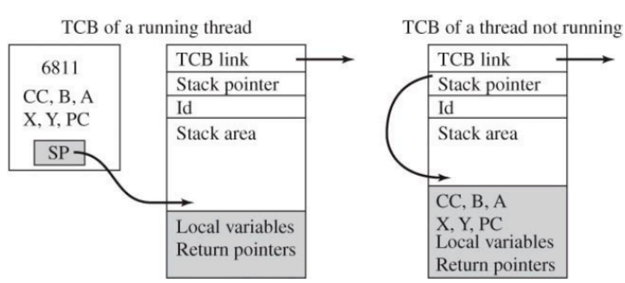
Round-Robin Scheduler



Thread Control Block

- TCB stores thread management information
  - must contain
    - » pointer slot so the linked list can be formed
    - » value of it's stack pointer
    - » stack area for local variables and saved registers
  - might also contain
    - » thread number, type, or name
    - » some sort of age information
      - how long it's been active
      - how long it's been in the run state
      - to be used in time based priority scheduling
    - » resources that this thread has been granted
      - if these resources are shared does it makes sense to hold them when blocked or active?
        - – why?

TCB Model



Thread Code

- **Admittedly somewhat silly**

```
int Sub(int j) { int i;
  PTM = 1;  // Port M
  i = j+1;
  return(i); }
void ProgA() { int i;
  i=5;
  while(1) {
    PTM = 2;
    i = Sub(i); }}
void ProgB() { int i;
  i=6;
  while(1) {
    PTM = 4;
    i = Sub(i); }}
```

PTM assignment used to provide external visibility of the running thread

Use of one-hot code on PortM pins is just a random choice

Key is that both ProgA & ProgB threads run forever

Hence preemptive scheduler is needed

Setting up the TCB

```
struct TCB
{   struct TCB *Next;      /* Link to Next TCB */
    unsigned char *SP;     /* Stack Pointer when idle  */
    unsigned short Id;     /* output to PortT */
    unsigned char MoreStack[49];  /* more stack */
    unsigned char CCR;     /* Initial CCR */
    unsigned char RegB;    /* Initial RegB */
    unsigned char RegA;    /* Initial RegA */
    unsigned short RegX;   /* Initial RegX */
    unsigned short RegY;   /* Initial RegY */
    void (*PC)(void);      /* Initial PC */
};
typedef struct TCB TCBType;
typedef TCBType * TCBPtr;
```

see anything fishy so far?

Port M vs. Port T

- **Essential difference between *program & thread***
  - **program is just the code**
    - » **note that code has no state**
    - » **it's just a specification of what will happen if it is executed**
  - **thread is an execution instance**
    - » **inherently has state**
      - **in this case initial state can be seen in the code**
      - **subsequent state will depend**
        - **TCB values if the thread isn't running**
        - **TCB values and registers if the thread is running**
- **In this simple example**
  - **Port M is used to show which Program is being executed**
  - **Port T is used to show which Thread is being executed**
  - **in this case**
    - » **M will be the same for threads 1 & 2**
    - » **in general**
      - **a thread could run more than 1 program in different thread phases**

Defining 3 Threads

```
TCBType sys[3]={
  { &sys[1],         /* Pointer to Next */
    &sys[0].CCR,     /* Initial SP */
    1,               /* Id */
    { 0},
    0x40,0,0,0,0,    /* CCR,B,A,X,Y */
    ProgA },        /* Initial PC */
  { &sys[2],         /* Pointer to Next */
    &sys[1].CCR,     /* Initial SP */
    2,               /* Id */
    { 0},
    0x40,0,0,0,0,    /* CCR,B,A,X,Y */
    ProgA },        /* Initial PC */
  { &sys[0],         /* Pointer to Next */
    &sys[2].CCR,     /* Initial SP */
    4,               /* Id */
    { 0},
    0x40,0,0,0,0,    /* CCR,B,A,X,Y */
    ProgB } };      /* Initial PC */
```

Thread n = sys[n]

threads 1 & 2 are the same code but work on different local data

CCR = 0x40
XIRQ disabled
IRQ enabled

Note all TCB variables values here influence only what happens the FIRST time the thread is executed

Why will these variables need to be changed for subsequent executions?

Preemptive Thread Scheduler in C

```
TCBPtr RunPt;            /* Pointer to current thread  */
void main(void) {
  DDRT = 0xFF;         /* Output running thread on Port T */
  DDRM = 0xFF;         /* Output running program on Port M */
  RunPt = &sys[0];     /* Specify first thread */
  asm  sei
  TFLG1 = 0x20;        /* Clear C5F */
  TIE = 0x20;          /* Arm C5F */
  TSCR1 = 0x80;        /* Enable TCNT*/
  TSCR2 = 0x01;        /* 2MHz TCNT */
  TIOS |= 0x20;        /* Output compare */
  TC5 = TCNT+20000;
  PTT = RunPt->Id;
  asm  ldx RunPt
  asm  lds 2,x
  asm  cli
  asm  rti
}    /* Launch First Thread */
```


Preemptive Thread Switch

```
void interrupt 13 ThreadSwitch() {
  asm ldx RunPt
  asm sts 2,x
  RunPt = RunPt->Next;
  PTT = RunPt->Id;        /* PortH=active thread */
  asm   ldx RunPt
  asm   lds 2,x
  TC5 = TCNT+20000;       /* Thread runs for 10 ms */
  TFLG1 = 0x20; }         /* ack by clearing C5F */
```

see any mistakes?

Dynamic Thread Allocator

```
int create(void (*startFunc)(void), int TheId) {
  TCBPtr NewPt;      // pointer to new thread control block
  NewPt = (TCBPtr)malloc(sizeof(TCBType)); // new TCB
  if(NewPt==0)return FAIL;
  NewPt->SP = &(NewPt->CCR);  /* Stack Pointer when not running */
  NewPt->Id = TheId;          /* Visualize active thread */
  NewPt->CCR = 0x40;          /* Initial CCR, I=0 */
  NewPt->RegB = 0;            /* Initial RegB */
  NewPt->RegA = 0;            /* Initial RegA */
  NewPt->RegX = 0;            /* Initial RegX */
  NewPt->RegY = 0;            /* Initial RegY */
  NewPt->PC = startFunc;      /* Initial PC */
  if(RunPt) {
    NewPt->Next = RunPt->Next;
    RunPt->Next = NewPt;}     /* will run Next */
  else
    RunPt = NewPt;            /* the first and only thread */
  return SUCCESS;
}
```


Concluding Remarks

- **Implementation of a very simple thread system**
  - **e.g. round robin preemptive**
  - **it's not that hard**
    - » **but note the tricks for setting the PC to the appropriate thread code start**
- **Preemptive scheduling**
  - **lies at the heart of an RTOS**
    - » **but in this case we didn't consider real time issues**
      - **making things significantly easier**
- **The hard part**
  - **designing correct embedded codes that use threads**
- **Note**
  - **this code shows the general idea**
  - **there are parts missing that will need to be coded for a full solution – future lab?**