ECE/CS 5780/6780: Embedded System Design

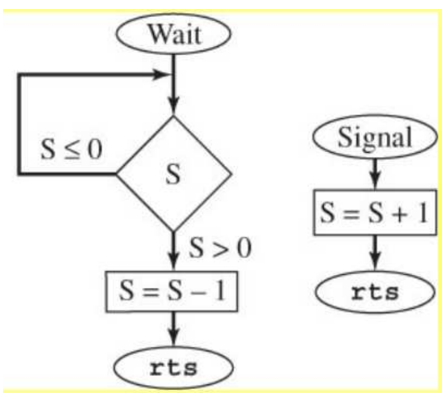
Chris J. Myers

Lecture 11: Semaphores

Introduction to Semaphores

- *Semaphores* used to implement synchronization, sharing, and communication between threads.
- A semaphore is a counter with two operations:
  - P or wait
  - V or signal
- A meaning is assigned to each counter value.
- In a binary semaphore, 1 means free and 0 means busy.

Spin-Lock Semaphore



Spin-Lock Counting Semaphore

```
// decrement and spin if less than 0
// input:  pointer to a semaphore
// output: none
void OS_Wait(short *semaPt){
  asm sei      // Test and set is atomic
  while(*semaPt <= 0){  // disabled
    asm cli             // disabled
    asm nop             // enabled
    asm sei             // enabled
  }
  (*semaPt)--;          // disabled
  asm cli               // disabled
}                       // enabled
```


Spin-Lock Counting Semaphore (cont)

```
// increment semaphore
// input:  pointer to a semaphore
// output: none
void OS_Signal(short *semaPt){
unsigned char SaveCCR;
  asm tpa
  asm staa SaveCCR   // save previous
  asm sei            // make atomic
  (*semaPt)++;
  asm ldaa SaveCCR   // recall previous
  asm tap            // end critical
}
```


Spin-Lock Binary Semaphore

```
void bWait(char *semaphore){
asm  clra          // new value for semaphore
asm loop: minm [2,x]    // test and set (ICC12 version 5)
asm      bcc loop
}
void bSignal(char *semaphore){
  (*semaphore) = 1; // compiler makes this atomic
}
```

Counting Semaphore

```
struct sema4
{   short value; // semaphore value
    char s1;     // binary semaphore
    char s2;     // binary semaphore
    char s3;     // binary semaphore
};
typedef struct sema4 sema4Type;
typedef sema4Type * sema4Ptr;
void Initialize(sema4Ptr semaphore, short initial){
  semaphore->s1 = 1;   // first one to bWait(s1) continues
  semaphore->s2 = 0;   // first one to bWait(s2) spins
  semaphore->s3 = 1;   // first one to bWait(s3) continues
  semaphore->value=initial;
}
```

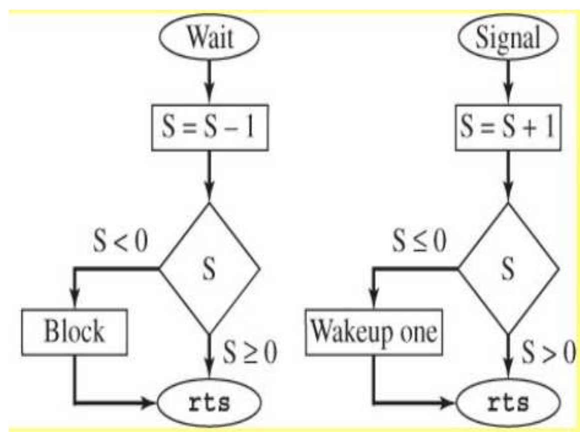

Counting Semaphore (cont)

```
void Wait(sema4Ptr semaphore){
  bWait(&semaphore->s3);  // wait if other caller here first
  bWait(&semaphore->s1);  // mutual exclusive access to value
  (semaphore->value)--;   // basic function of Wait
  if((semaphore->value)<0){
    bSignal(&semaphore->s1); // end of exclusive access
    bWait(&semaphore->s2);   // wait for value to go above 0
  }
  else
    bSignal(&semaphore->s1); // end of exclusive access
  bSignal(&semaphore->s3);   // let other callers in
}
```


Counting Semaphore (cont)

```
void Signal(sema4Ptr semaphore){
  bWait(&semaphore->s1);     // exclusive access
  (semaphore->value)++;      // basic function of Signal
  if((semaphore->value)<=0)
    bSignal(&semaphore->s2); // allow S2 spinner to continue
  bSignal(&semaphore->s1);   // end of exclusive access
}
```


Blocking Semaphore



Blocking Semaphore

- **Initialize**:
  1. Set the counter to its initial value.
  2. Clear associated blocked **tcb** linked list.
- **Wait**:
  1. Disable interrupts to make atomic
  2. Decrement the semaphore counter, S=S-1
  3. If semaphore counter < 0, then block this thread.
  4. Restore interrupt status.
- **Signal**:
  1. Disable interrupts to make atomic
  2. Increment the semaphore counter, S=S+1
  3. If counter ≤ 0, wakeup one thread.
  4. Restore interrupt status

Assembly to Initialize a Blocking Semaphore

```
S       rmb  1        ;semaphore counter
BlockPt rmb  2        ;Pointer to threads blocked on S
Init    tpa
        psha          ;Save old value of I
        sei           ;Make atomic
        ldaa #1
        staa S        ;Init semaphore value
        ldx  #Null
        stx  BlockPt  ;empty list
        pula
        tap           ;Restore old value of I
        rts
```

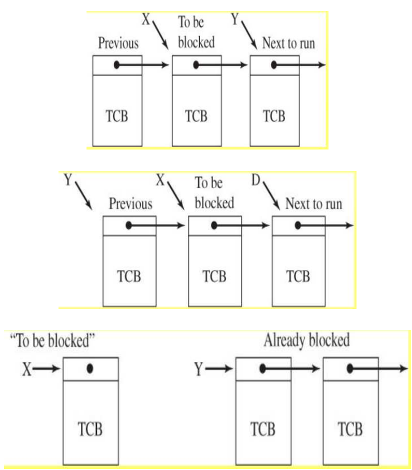
Assembly to Block a Thread

```
; To block a thread on semaphore S, execute SWI
SWIhan ldx  RunPt   ;running process "to be blocked"
       sts  SP,x    ;save Stack Pointer in its TCB
; Unlink "to be blocked" thread from RunPt list
       ldy  Next,x  ;find previous thread
       sty  RunPt   ;next one to run
look   cpx  Next,y  ;search to find previous
       beq  found
       ldy  Next,y
       bra  look
found  ldd  RunPt   ;one after blocked
       std  Next,y  ;link previous to next to run
```


Assembly to Block a Thread (cont)

```
; Put "to be blocked" thread on block list
       ldy  BlockPt
       sty  Next,x   ;link "to be blocked"
       stx  BlockPt
; Launch next thread
       ldx  RunPt
       lds  SP,x    ;set SP for this new thread
       ldd  TCNT    ;Next thread gets a full 10ms time slice
       addd #20000  ;interrupt after 10 ms
       std  TC5
       ldaa #$20
       staa TFLG1   ;clear C5F
       rti
```


Linked Lists



Thread Synchronization or Rendezvous

- Synchronize two threads at a *rendezvous* location.

S1	S2	Meaning
0	0	Neither thread at rendezvous location
-1	+1	Thread 2 arrived first, waiting for thread 1
+1	-1	Thread 1 arrived first, waiting for thread 2

Thread 1	Thread 2
`signal(&S1);`	`signal(&S2);`
`wait(&S2);`	`wait(&S1);`

Resource Sharing or Nonreentrant Code

- Guarantee mutual exclusive access to a critical section.

Thread 1	Thread 2	Thread 3
`bwait(&S);`	`bwait(&S);`	`bwait(&S);`
`printf("bye");`	`printf("tchau");`	`printf("ciao");`
`bsignal(&S);`	`bsignal(&S);`	`bsignal(&S);`

Thread Communication Between Two Threads

- Thread 1 sends mail to thread 2.

Send	Ack	Meaning
0	0	No mail available, consumer not waiting
-1	0	No mail available, consumer is waiting
+1	-1	Mail available and producer is waiting

Producer thread	Consumer thread
`Mail=4;`	`wait(&send);`
`signal(&send);`	`read(Mail);`
`wait(&ack);`	`signal(&ack);`

Thread Communication Between Many Threads

- In the *bounded buffer* problem, many threads put data into and take out of a finite-size FIFO.

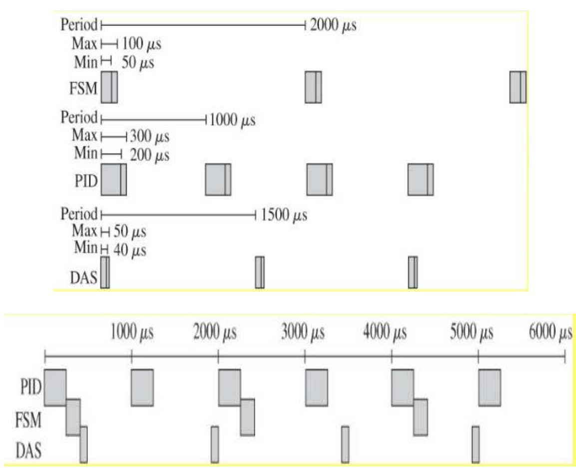
PutFifo	GetFifo
`wait(&RoomLeft);`	`wait(&CurrentSize);`
`wait(&mutex);`	`wait(&mutex);`
put data in FIFO	remove data from FIFO
`signal(&mutex);`	`signal(&mutex);`
`signal(&CurrentSize);`	`signal(&RoomLeft);`

- Could disable interrupts instead of using `mutex`, but would lock out threads that don't affect the FIFO.

Fixed Scheduling

- Thread sequence and allocated time-slices determined a priori.
- To create a fixed schedule, we need to:
  1. Assign a priority to each task.
  2. Define the resources required for each task.
  3. Determine how often each task must run.
  4. Estimate how long each task will require to complete.

Fixed Scheduling Example



Four User Threads

```
void FSM(void){ StatePtr Pt;
  Pt = SA;                 // Initial State
  DDRT = 0x03;             // PT1,PT0 outputs, PT3,PT2 inputs
  PTT = Pt->Out;           // Output
  for(;;) {
    OS_Sleep();            // Runs every 2ms
    Pt = Pt->Next[PTT>>2]; // Next state depends on the input
    PTT = Pt->Out;}}       // Output
void PID(void){ unsigned char speed,power;
  PID_Init();              // Initialize
  for(;;) {
    OS_Sleep();            // Runs every 1ms
    speed = PID_In();      // read tachometer
    power = PID_Calc(speed);
    PID_Out(power);}}      // adjust power to motor
```


Four User Threads (cont)

```
void DAS(void){ unsigned char raw;
  DAS_Init();            // Initialize
  for(;;) {
    OS_Sleep();          // Runs every 1.5ms
    raw = DAS_In();      // read ADC
    Result = DAS_Calc(raw); }}
void PAN(void){ unsigned char input;
  PAN_Init();            // Initialize
  for(;;) {
    input = PAN_In();    // front panel input
    if(input){
      PAN_Out(input);    // process
    }}}
```


The Thread Control Blocks

```
struct TCB{
  unsigned char *StackPt;       // Stack Pointer
  unsigned char MoreStack[91]; // 100 bytes of stack
  unsigned char InitialReg[7]; // initial CCR,B,A,X,Y
  void (*InitialPC)(void);};   // starting location
typedef struct TCB TCBType;
TCBType *RunPt;              // thread currently running
#define TheFSM &sys[0]       // finite state machine
#define ThePID &sys[1]       // prop.-int.-derivative
#define TheDAS &sys[2]       // data acquisition system
#define ThePAN &sys[3]       // front panel
TCBType sys[4]={
  { TheFSM.InitialReg[0],{ 0},{0x40,0,0,0,0,0,0},FSM},
  { ThePID.InitialReg[0],{ 0},{0x40,0,0,0,0,0,0},PID},
  { TheDAS.InitialReg[0],{ 0},{0x40,0,0,0,0,0,0},DAS},
  { ThePAN.InitialReg[0],{ 0},{0x40,0,0,0,0,0,0},PAN}};
```

The Schedule

```
struct Node{
  struct Node *Next;          // circular linked list
  TCBType *ThreadPt;          // which thread to run
  unsigned short TimeSlice;}; // how long to run it
typedef struct Node NodeType;
NodeType *NodePt;
```


The Schedule (cont)

```
NodeType Schedule[22]={
{ &Schedule[1], ThePID, 300}, // interval    0,  300
{ &Schedule[2], TheFSM, 100}, // interval  300,  400
{ &Schedule[3], TheDAS,  50}, // interval  400,  450
{ &Schedule[4], ThePAN, 550}, // interval  450, 1000
{ &Schedule[5], ThePID, 300}, // interval 1000, 1300
{ &Schedule[6], ThePAN, 600}, // interval 1300, 1900
{ &Schedule[7], TheDAS,  50}, // interval 1900, 1950
{ &Schedule[8], ThePAN,  50}, // interval 1950, 2000
{ &Schedule[9], ThePID, 300}, // interval 2000, 2300
{ &Schedule[10],TheFSM, 100}, // interval 2300, 2400
{ &Schedule[11],ThePAN, 600}, // interval 2400, 3000
{ &Schedule[12],ThePID, 300}, // interval 3000, 3300
{ &Schedule[13],ThePAN, 100}, // interval 3300, 3400
{ &Schedule[14],TheDAS,  50}, // interval 3400, 3450
{ &Schedule[15],ThePAN, 550}, // interval 3450, 4000
{ &Schedule[16],ThePID, 300}, // interval 4000, 4300
{ &Schedule[17],TheFSM, 100}, // interval 4300, 4400
{ &Schedule[18],ThePAN, 500}, // interval 4400, 4900
{ &Schedule[19],TheDAS,  50}, // interval 4900, 4950
{ &Schedule[20],ThePAN,  50}, // interval 4950, 5000
{ &Schedule[21],ThePID, 300}, // interval 5000, 5300
{ &Schedule[0], ThePAN, 700}  // interval 5300, 6000 };
```


The Scheduler

```
void main(void) {
  NodePt = &Schedule[0];     // first thread to run
  RunPt = NodePt->ThreadPt;
  TIOS |= 0x08;     // activate OC3
  TSCR1 = 0x80;     // enable TCNT
  TSCR2 = 0x02;     // usec TCNT
  TIE |= 0x08;      // Arm TC3
  TC3 = TCNT+NodePt->TimeSlice;
  TFLG1 = 0x08;     // Clear C3F
asm ldx RunPt
asm lds 0,x
asm rti     // Launch First Thread
}
```


The Scheduler (cont)

```
interrupt 11 void threadSwitchISR(void){
asm ldx RunPt
asm sts 0,x
  NodePt = NodePt->Next;
  RunPt = NodePt->ThreadPt;    // which thread to run
  TC3 = TC3+NodePt->TimeSlice; // Thread runs for time slice
  TFLG1 = 0x08;                // ack by clearing TC3F
asm ldx RunPt
asm lds 0,x
}
```


The Scheduler (cont)

```
void OS_Sleep(void){  // cooperative multitasking
  asm swi             // suspend this tread and run another
}
interrupt 4 void swiISR(void){
asm ldx RunPt     // cooperative multitasking
asm sts 0,x       // thread goes to sleep when it is done
  RunPt = ThePAN; // non-real time thread
asm ldx RunPt
asm lds 0,x
}
```