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.
C for a Spin-Lock Counting Semaphore

```c
struct sema4
{   int value; // semaphore value
    char s1; // binary semaphore
    char s2; // binary semaphore
    char s3; // binary semaphore
};
typedef struct sema4 sema4Type;
typedef sema4Type * sema4Ptr;
```

C for a Spin-Lock Counting Semaphore (cont)

```c
void Wait(sema4Ptr semaphore){
    bWait(&semaphore->s3);
    bWait(&semaphore->s1);
    (semaphore->value)--;
    if((semaphore->value)<0){
        bSignal(&semaphore->s1);
        bWait(&semaphore->s2);
    }
    else
    
        bSignal(&semaphore->s1);
        bWait(&semaphore->s3);
}
```

C for a Spin-Lock Counting Semaphore (cont)

```c
void Signal(sema4Ptr semaphore){
    bWait(&semaphore->s1);
    (semaphore->value)++;
    if((semaphore->value)<=0)
        bSignal(&semaphore->s2);
    bSignal(&semaphore->s1);
}

void Initialize(sema4Ptr semaphore, int initial){
    semaphore->s1=1; // first one to bWait(s1) cont.
    semaphore->s2=0; // first one to bWait(s2) spins
    semaphore->s3=1; // first one to bWait(s3) cont.
    semaphore->value=initial;}
```
**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 \( \leq 0\), wakeup one thread.
  4. Restore interrupt status

---

**Assembly to Initialize a Blocking Semaphore**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>semaphore counter</td>
</tr>
<tr>
<td>BlockPt</td>
<td>Pointer to threads blocked on S</td>
</tr>
<tr>
<td>Init tpa</td>
<td></td>
</tr>
<tr>
<td>psha</td>
<td>Save old value of I</td>
</tr>
<tr>
<td>sei</td>
<td>Make atomic</td>
</tr>
<tr>
<td>ldaa #1</td>
<td></td>
</tr>
<tr>
<td>staa S</td>
<td>Init semaphore value</td>
</tr>
<tr>
<td>ldax #Null</td>
<td></td>
</tr>
<tr>
<td>stx BlockPt</td>
<td>empty list</td>
</tr>
<tr>
<td>pula</td>
<td></td>
</tr>
<tr>
<td>tap</td>
<td>Restore old value of I</td>
</tr>
<tr>
<td>rts</td>
<td></td>
</tr>
</tbody>
</table>

---

**Assembly to Block a Thread**

- SWI:
  1. ldx RunPt running process "to be blocked"
  2. sts SP,x save Stack Pointer in its TCB
- *Unlink "to be blocked" thread from RunPt list
  1. ldy Next,x find previous thread
  2. sty RunPt next one to run
- look:
  1. cpx Next,y search to find previous
  2. beq found
  3. ldy Next,y
  4. bra look
- found:
  1. ldd RunPt one after blocked
  2. std Next,y link previous to next to run

---

**Assembly to Block a Thread (cont)**

- *Put "to be blocked" thread on block list
  1. ldy BlockPt
  2. sty Next,x link "to be blocked"
  3. stx BlockPt
- *Launch next thread
  1. ldx RunPt
  2. lds SP,x set SP for this new thread
  3. ldd TCNT Next thread get time slice
  4. addd #20000 interrupt after 10 ms
  5. std TC5
  6. ldaa #$08, ($20 on the 6812)
  7. staa TFLG1 clear OC5F
  8. rti
Linked Lists

(See Figures 5.10, 5.11, and 5.12)

Thread Synchronization or Rendezvous

- Synchronize two threads at a rendezvous location.

<table>
<thead>
<tr>
<th>S1</th>
<th>S2</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Neither thread at rendezvous location</td>
</tr>
<tr>
<td>-1</td>
<td>+1</td>
<td>Thread 2 arrived first, waiting for thread 1</td>
</tr>
<tr>
<td>+1</td>
<td>-1</td>
<td>Thread 1 arrived first, waiting for thread 2</td>
</tr>
</tbody>
</table>

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.