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.

Assembly for a Spin-Lock Counting Semaphore

```
S    fcb 1    semaphore counter init. to 1
wait sei    make read-modify-write atomic
ldaa S    current value of semaphore
bhi OK    available if >0
cli
bra wait  **interrupts can occur here**
OK
deca
staa S    S=S-1
cli
rts
signal inc S    S=S+1, this is atomic
rts
```
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);
        bSignal(&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

- `s` is a counter
- `wait` event
- `signal` event
- `block` event
- `wakeup one` event
- `rts` event
**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**

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

---

**Assembly to Block a Thread**

```
SWIhan lda x   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
  lda x  RunPt
  lds SP,x set SP for this new thread
  ldax TCNT Next thread get time slice
  addx #20000 interrupt after 10 ms
  std TOC5
  ldaa #$08 ($20 on the 6812)
  staa TFLG1 clear OC5F
  rti
```
Slide 13

Linked Lists

Slide 14

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
signal(&S1); signal(&S2);
wait(&S2); wait(&S1);

Thread 2

Slide 15

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);

Slide 16

Thread Communication Between Two Threads

- Thread 1 sends mail to thread 2.

<table>
<thead>
<tr>
<th>Send</th>
<th>Ack</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>No mail available, consumer not waiting</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>No mail available, consumer is waiting</td>
</tr>
<tr>
<td>+1</td>
<td>-1</td>
<td>Mail available and producer is waiting</td>
</tr>
</tbody>
</table>

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.

  ```c
  PutFifo
  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.