

5. Threads

5.3.2. Resource sharing, or mutual exclusion

Interpreter	Consumer
<code>Wait(&DisplayFree);</code>	<code>Wait(&DisplayFree);</code>
<code>OutString("bye");</code>	<code>OutString("tchau");</code>
<code>Signal(&DisplayFree);</code>	<code>Signal(&DisplayFree);</code>

5.3.4. Thread communication using a fifo queue

bounded buffer problem

communication between two foreground threads

We need two counting semaphores called,

DataAvailable number of entries currently stored

DataRoomLeft number of empty spaces left

The **PutFifo** routine executes the following steps:

```

Wait(&DataRoomLeft);
save I bit
asm(" sei"); // mutually exclusive
Enter information into the FIFO structure
restore I bit // end of critical
Signal(&DataAvailable);

```

The **GetFifo** routine executes the following steps:

```

Wait(&DataAvailable);
save I bit
asm(" sei"); // mutually exclusive
Remove information from the FIFO structure
restore I bit // end of critical
Signal(&DataRoomLeft);

```

different situation when background to foreground

CAN NOT CALL WAIT FROM BACKGROUND

1) The **RxFifo_Put** routine is called from the SCI ISR

```

Input new data from SCI
Try to enter data into RxFifo
If successful
OS_Signal(&RxAvailable)

```

The `RxFifo_Get` routine is called when foreground, `InChar`

```
OS_Wait(&RxAvailable)
Remove 8-bit data from RxFifo
```

2) The `TxFifo_Get` routine is called from the SCI ISR

```
Try to remove data from TxFifo
If successful
    OS_Signal(&TxRoomLeft)
    output the data to the SCI
else
    // because empty
    disarm
```

The `TxFifo_Put` routine is called when a foreground, `OutChar`

```
OS_Wait(&TxRoomLeft)
Enter data into TxFifo
```

3) Producer ADC in background, consumer in foreground

```
int OS_SendData(unsigned short data){
// called from ISR
unsigned short *tempPt;
tempPt = DataPutPt;
*tempPt = data; // Try to Put data
tempPt++; // place to Put next
if(tempPt == &DataFifo[DATAFIFOSIZE]){
tempPt = &DataFifo[0];
}
if(tempPt == DataGetPt){
return(0); // Failed, fifo full
}
DataPutPt = tempPt; // Success, update
OS_Signal(&DataAvailable); /
return(1);
}
```

```
unsigned short OS_ReceiveData(void){ unsigned short data;
OS_Wait(&DataAvailable);
data = (*DataGetPt); // remove data
DataGetPt++; // Place to Put next
if(DataGetPt == DataFifo[DATAFIFOSIZE]){
DataGetPt = &DataFifo[0];
}
return(data);
}
```

5.1.2. Other Scheduling Algorithms

- A non-preemptive (cooperative) scheduler
 - trusts each thread to voluntarily release control
 - not appropriate for real time systems
 - Can be combined with preemptive for efficiency

Add preemption points and run again.

5.2.2. Blocking semaphore implementation

why

- recapture time lost in the spin operation of spinlock
- eliminate wasted time scheduling waiting threads
- implement **bounded waiting**
 - once thread calls **wait** and is not serviced,
 - there are a finite number of threads that will go ahead

how (method 1: hard, efficient, bounded waiting)

- each semaphore has a blocked **tcb** linked list
- contains the threads that are blocked
- empty if semaphore counter ≥ 0
- e.g., if counter = -2, then two threads are blocked
- order on blocked list determine sequence of blocking
- sequence of blocking determine which to wake up

Initialize:

- 1) Set the counter to its initial value
- 2) Clear the associated blocked **tcb** linked list

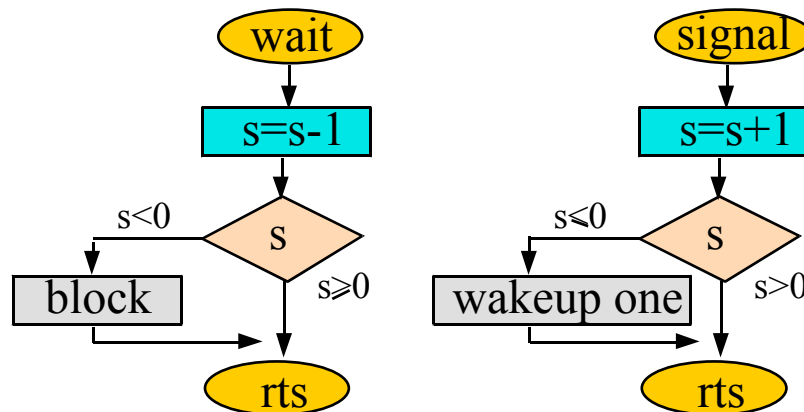


Figure 5.9. Flowcharts of a blocking counting semaphore.

New OS_wait

- 1) Disable interrupts, I=1
- 2) Decrement the semaphore counter, S=S-1
`(semaPt->Counter) -- ;`
- 3) If the **Counter** < 0 then this thread will be blocked
 set the status of this thread to blocked,

specify this thread is to be blocked to this semaphore
 suspend thread using **TC3=TCNT+15 ;**

4) Enable interrupts, I=0

New ThreadSwitch

- 1) Save SP into TCB
- 2) If this thread is to be blocked
 move **TCB** to blocked list of specified semaphore
- 3) Find the next active thread from the active list
- 4) Acknowledge C3F, set TC3, and launch next thread

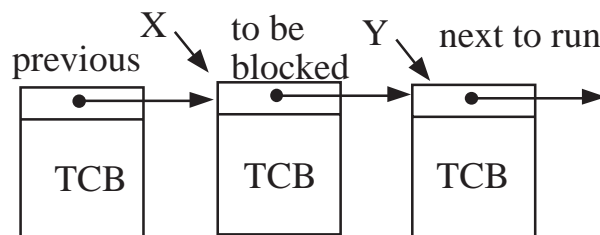


Figure 5.10. Linked list before the thread is blocked.

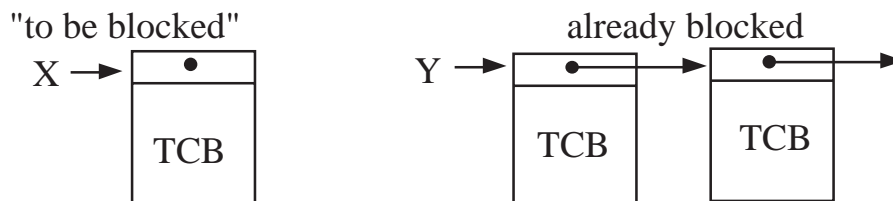


Figure 5.12. TCB's after the thread is blocked

New OS_Signal

- 1) Save I bit, then disable interrupts
- 2) Increment the semaphore counter, $S=S+1$
 $(\text{semaPt} \rightarrow \text{Counter}) ++;$
- 3) If the **Counter** ≤ 0 then
 wake up one thread from the **TCB** linked list
 (bounded waiting -> the one waiting the longest)
 do not suspend the thread that called **OS_Signal**
 simply move **TCB** of the "wakeup" thread
 from the blocked list to the active list
- 4) Restore I bit

how (method 2: easy, not efficient, no bounded waiting)

all threads exist on circular TCB list: active and blocked
 each semaphore simply has a counter

no blocked threads if semaphore counter ≥ 0
 e.g., if **Counter** is -2, then two threads are blocked
 no information about which thread has waited longest
 add to TCB, a **status**, of type **Sema4Type**
 initially, this pointer is **null**
null means this thread is active and ready to run
 if blocked, this pointer contains the semaphore address

New OS_Wait

- 1) Disable interrupts, I=1
- 2) Decrement the semaphore counter, S=S-1
`(semaPt->Counter) -- ;`
- 3) If the **Counter**<0 then this thread will be blocked
 specify this thread is blocked to this semaphore
`RunPt->status = semaPt;`
 suspend thread using `TC3=TCNT+15 ;`
- 4) Enable interrupts, I=0

New ThreadSwitch

- 1) Save SP into TCB
- 2) Find the next active thread from the TCB list
 only run threads with **status** equal to **null**
- 3) Acknowledge C3F, set TC3, and launch next thread

New OS_Signal

- 1) Save I bit, then disable interrupts
- 2) Increment the semaphore counter, S=S+1
`(semaPt->Counter) ++ ;`
- 3) If **Counter** ≤ 0 then
 wake up one thread from the **TCB** linked list
 (no bounded waiting)
 do not suspend the thread that called **OS_Signal**
 search TCBs for thread with **status == semaPt**
 set the **status** of this **TCB** to **null**
- 4) Restore I bit

5.1.2. Other Scheduling Algorithms

A priority scheduler

assigns each thread a priority number
 blocking semaphores and not spinlock semaphores
 priority 2 is run only if no priority 1 are ready
 priority 3 only if no priority 1 or priority 2 are ready
 if all have the same priority, use a round-robin system

reduce latency (response time) by giving high priority on a busy system, low priority threads may never be run.

starvation.

5.3.1. Thread synchronization or rendezvous

<u>S1</u>	<u>S2</u>	<u>meaning</u>
0	0	neither thread has arrived or both have passed
-1	+1	thread 2 arrived first and is waiting for thread 1
+1	-1	thread 1 arrived first and is waiting for thread 2

<u>Thread 1</u>	<u>Thread 2</u>
signal (&S1) ;	signal (&S2) ;
wait (&S2) ;	wait (&S1) ;

5.3.3. Thread communication using a mailbox

send mail from thread 1 to thread 2
 producer to specify when new mail is available
 consumer to specify when mail was received

<u>Send</u>	<u>Ack</u>	<u>meaning</u>
0	0	no mail available, consumer not waiting
-1	0	no mail available, consumer is waiting for mail
+1	-1	mail available and producer is waiting for ack

<u>Producer Thread</u>	<u>Consumer Thread</u>
Mail=4;	wait (&send) ;
signal (&send)	read (Mail) ;
wait (&ack)	signal (&ack)