CS/ECE 6780/5780

Al Davis

Today's topics:

- **Threads**
  - **restart code from last lecture**
  - **move on to scheduling & semaphores**
- **Midterm (next Tues) covers Chaps & Labs 1-5**
  - **sample on the web**

Interrupts on the 6812

- **ISR's**
  - **declare ISR's and the vector that they're associated**

```
void interrupt n IsrFcn () { …code…}
```

  - » **interrupt n is mapped to a particular branch PC**
    - **maps n to IsrFcn**
    - **compiler reference manual pg. 538-539 shows what happens**

Table 10.11 Vector relationships

Vector Number	Vector Address	Vector Address Size
0	0xFFFE, 0xFFFF	2
1	0xFFFC, 0xFFFD	2
2	0xFFFA, 0xFFFB	2
...	...	...
n	0xFFFF - (n*2)	2

Mapping HW Events to Interrupt #'s

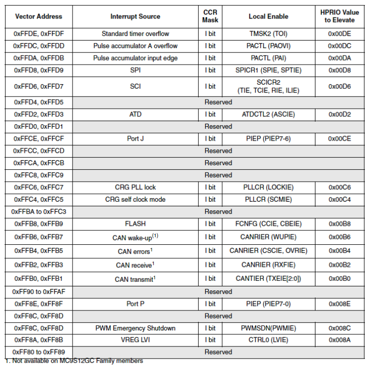
- **In more complex microcontrollers**
  - **this mapping can be set up in software**
- **For the 6812**
  - **the map is fixed**
    - » **see MC9S12C128V reference manual section 1.6.1 pg. 61-62**

First 18 Interrupts

Table 1-9. Interrupt Vector Locations

Int #	Vector Address	Interrupt Source	CCR Mask	Local Enable	HPRIO Value to Elevate
	0xFFFE, 0xFFFF	External reset, power on reset, or low voltage reset (see CRG flags register to determine reset source)	None	None	—
	0xFFFC, 0xFFFD	Clock monitor fail reset	None	COPCTL (CME, FCME)	—
	0xFFFA, 0xFFFB	COP failure reset	None	COP rate select	—
	0xFFF8, 0xFFF9	Unimplemented instruction trap	None	None	—
	0xFFF6, 0xFFF7	SWI	None	None	—
	0xFFF4, 0xFFF5	XIRQ	X-Bit	None	—
Int 7	0xFFF2, 0xFFF3	IRQ	I bit	INTCR (IRQEN)	0x00F2
	0xFFF0, 0xFFF1	Real time Interrupt	I bit	CRGINT (RTIE)	0x00F0
	0xFFEE, 0xFFEF	Standard timer channel 0	I bit	TIE (C0I)	0x00EE
	0xFFEC, 0xFFED	Standard timer channel 1	I bit	TIE (C1I)	0x00EC
	$FFEE, $FFEF	Reserved			
	$FFEC, $FFED	Reserved			
Int 13	0xFFEA, 0xFFEB	Standard timer channel 2	I bit	TIE (C2I)	0x00EA
	0xFFE8, 0xFFE9	Standard timer channel 3	I bit	TIE (C3I)	0x00E8
	0xFFE6, 0xFFE7	Standard timer channel 4	I bit	TIE (C4I)	0x00E6
	0xFFE4, 0xFFE5	Standard timer channel 5	I bit	TIE (C5I)	0x00E4
	0xFFE2, 0xFFE3	Standard timer channel 6	I bit	TIE (C6I)	0x00E2
	0xFFE0, 0xFFE1	Standard timer channel 7	I bit	TIE (C7I)	0x00E0

Remaining Interrupts



Vector Address	Interrupt Source	CCR Mask	Local Enable	HPRIO Value to Elevate
0xFFDE, 0xFFDF	Standard timer overflow	I bit	TMSK2 (TOI)	0x00DE
0xFFDC, 0xFFDD	Pulse accumulator A overflow	I bit	PACTL (PAOVI)	0x00DC
0xFFDA, 0xFFDB	Pulse accumulator input edge	I bit	PACTL (PAI)	0x00DA
0xFFD8, 0xFFD9	SPI	I bit	SPICR1 (SPIE, SPTIE)	0x00D8
0xFFD6, 0xFFD7	SCI	I bit	SCICR2 (TIE, TCIE, RIE, ILIE)	0x00D6
0xFFD4, 0xFFD5	Reserved			
0xFFD2, 0xFFD3	ATD	I bit	ATDCTL2 (ASCIE)	0x00D2
0xFFD0, 0xFFD1	Reserved			
0xFFCE, 0xFFCF	Port J	I bit	PIEP (PIEP7-6)	0x00CE
0xFFCC, 0xFFCD	Reserved			
0xFFCA, 0xFFCB	Reserved			
0xFFC8, 0xFFC9	Reserved			
0xFFC6, 0xFFC7	CRG PLL lock	I bit	PLLCR (LOCKIE)	0x00C6
0xFFC4, 0xFFC5	CRG self clock mode	I bit	PLLCR (SCMIE)	0x00C4
0xFFBA to 0xFFC3	Reserved			
0xFFB8, 0xFFB9	FLASH	I bit	FCNFG (CCIE, CBEIE)	0x00B8
0xFFB6, 0xFFB7	CAN wake-up(1)	I bit	CANRIER (WUPIE)	0x00B6
0xFFB4, 0xFFB5	CAN errors1	I bit	CANRIER (CSCIE, OVRIE)	0x00B4
0xFFB2, 0xFFB3	CAN receive1	I bit	CANRIER (RXFIE)	0x00B2
0xFFB0, 0xFFB1	CAN transmit1	I bit	CANTIER (TXEIE[2:0])	0x00B0
0xFF90 to 0xFFAF	Reserved			
0xFF8E, 0xFF8F	Port P	I bit	PIEP (PIEP7-0)	0x008E
0xFF8C, 0xFF8D	Reserved			
0xFF8C, 0xFF8D	PWM Emergency Shutdown	I bit	PWMSDN(PWMIE)	0x008C
0xFF8A, 0xFF8B	VREG LVI	I bit	CTRL0 (LVIE)	0x008A
0xFF80 to 0xFF89	Reserved			

1. Not available on MC9S12GC Family members

not needed for Lab5 or this lecture

Thread Code

- **Admittedly somewhat silly & review from last lecture**

```
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?
what does "interrupt 13" mean?

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?

So Far

- We've talked about
  - thread scheduling
  - synchronization
    - » between main & ISR's
- Next
  - synchronization
    - » between threads
  - table driven scheduling

2 Threads & a communication FIFO

```
int Fifo_Put(char data)
{
  char *Ppt;
  // BEGIN CRITICAL
  Ppt=PutPt;
  *(Ppt++)=data;
  if (Ppt == &Fifo[FIFOSIZE]) Ppt = &Fifo[0];
  if (Ppt == GetPt ) {
    // END CRITICAL
    return(0);
  } else {
    PutPt=Ppt;
    // END CRITICAL
    return(1);
  }
}
```

what's missing?

Semaphores

- Used to implement mutual exclusion (MUTEX)
  - useful for sharing, synchronization, & communication
- 2 basic operations
  - classic terminology
    - » P → wait (Djikstra's Dutch "probeer te verlagen" ::- try to grab"
    - » V → signal ("verhogen" ::= increase)
  - semaphore is binary value
    - » 1 → free (resource available)
    - » 0 → busy (resource owned by some other thread)
- Numerous semaphore implementations
  - simplest is a "Spin-Lock" version
    - » thread calls `wait` to wait (spins) for semaphore to be free
      - when semaphore is free, `wait` sets semaphore to busy
      - and then return
    - » critical
      - enable interrupts during spin or preemption can't happen
      - read modify write on semaphore value must be atomic
        - – otherwise an interrupt might switch threads and chaos will result
    - » once thread is done it calls signal to return the semaphore to free

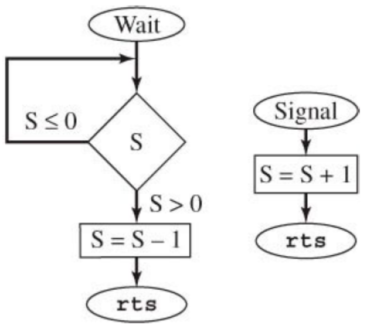
2 Common Semaphore Types

- Binary
  - simple lock
- Counter
  - useful when multiple resources are available
    - » say tables in a restaurant
      - >0 there is something available
      - <= 0 the place is busy
      - hence the semantics
- Note earlier lecture error
  - tst instruction is not "Test and Set"
    - » this instruction exists on a lot of machines
      - useful for binary semaphores
  - 6812 TST, TSTA, TSTB → test M,A, or B for 0 or minus
    - » does not change M, A, or B value – just changes the CC flags
    - » hence not useful for semaphore implementation

MINA & MINM Instructions

- MINM
  - stores minimum value
    - » MINM [opndA, opndB] → min(A, B) stored in B
      - CC bits N,Z,V,C set based on A-B
      - opndA can be
        - indexed addressing postbyte code
          - e.g. preincrement similar to "test and set"
        - integer in various ranges
      - opndB can be
        - X, Y, SP
  - useful for semaphore implementation
- MINA does the same thing
  - but result goes to register A rather than memory
  - only one REG A however so unlikely choice for semaphore implmentation

Wait & Signal



Remember the Atomicity Functions

```
unsigned char begin_critical (void)
{
  unsigned char SaveSP;
  asm tpa
  asm staa SaveSP
  asm sei
  return SaveSP;
}

void end_critical (unsigned char SaveSP)
{
  asm ldaa SaveSP
  asm tap
}
```

Key idea

begin - save the predicate register
in SaveSP
disable interrupts

on end – restore the predicate register

why no cli to re-enable interrupts?

Spin-Lock Semaphore Wait

```
// decrement and spin if less than 0
// input:  pointer to a semaphore
// output: none
void OS_Wait(short *semaPt) {
  unsigned char SaveSP = begin_critical();
  while(*semaPt <= 0) {
    end_critical (SaveSP);
    asm nop
    SaveSP = begin_critical();
  }
  (*semaPt)--;
  end_critical (SaveSP);
}
```

key point: semaphore access is in critical section & MUTEX

Spin-Lock Semaphore Signal

```
// increment semaphore
// input:  pointer to a semaphore
// output: none
void OS_Signal(short *semaPt) {
  unsigned char SaveSP = begin_critical();
  (*semaPt)++;
  end_critical (SaveSP);
}
```

Spin-Lock Binary Semaphore

```
void bWait(char *semaphore) {
  asm       clra
  asm loop: minm [2,x]
  asm       bcc loop
}
void bSignal(char *semaphore) {
  (*semaphore) = 1; // compiler makes this atomic
}
```

clra sets new value for the semaphore
minm [2,X] is test and set in ICC version 5

not sure why this works – anybody know?

Counting Semaphore from Binary Semaphores

```
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'd)

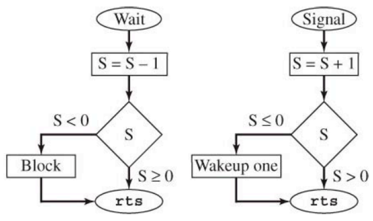
```
void Wait(sema4Ptr semaphore) {
  bWait(&semaphore->s3);  // wait if other caller here first
  bWait(&semaphore->s1);  // mutual exclusive access to value
  (semaphore->value)--;   // basic function of Wait
  if((semaphore->value)<0) {
    bSignal(&semaphore->s1); // end of exclusive access
    bWait(&semaphore->s2);   // 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'd)

```
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

- **Useful when multiple threads are blocked waiting on a resource**



Blocking Semaphore Stages

Initialize:
- Set the counter to its initial value.
- Clear associated blocked **tcb** linked list.

Wait:
- Disable interrupts to make atomic
- Decrement the semaphore counter, S=S-1
- If semaphore counter < 0, then block this thread.
- Restore interrupt status.

Signal:
- Disable interrupts to make atomic
- Increment the semaphore counter, S=S+1
- If counter ≤ 0, wakeup one thread.
- Restore interrupt status

Initialize

```
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
```

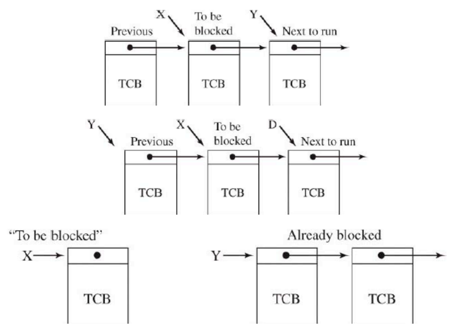
Block a Thread

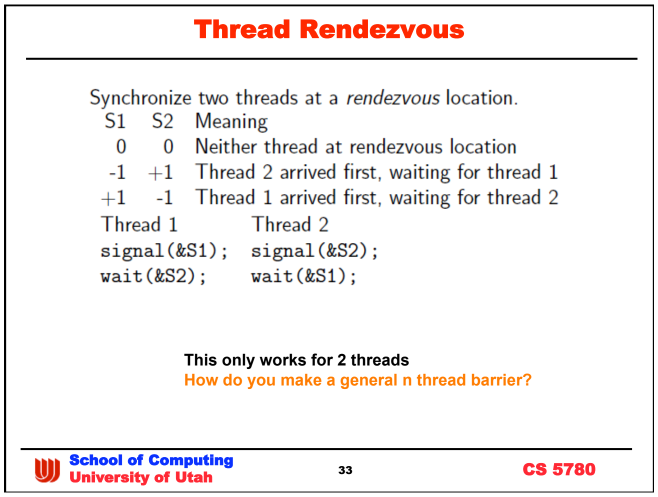
```
; To block a thread on semaphore S, execute SWI
SWIhan ldx  RunPt   ;running process "to be blocked"
       sts  SP,x    ;save Stack Pointer in its TCB
; Unlink "to be blocked" thread from RunPt list
       ldy  Next,x  ;find previous thread
       sty  RunPt   ;next one to run
look   cpx  Next,y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

Block and Launch Next

```
; 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 Rendezvous

Synchronize two threads at a *rendezvous* location.

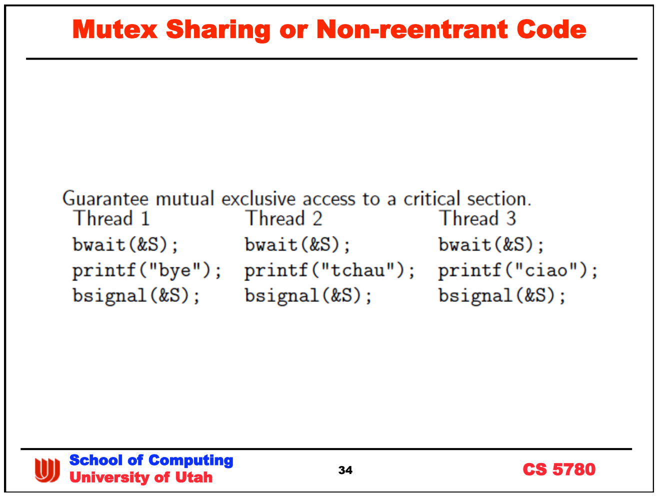
S1	S2	Meaning
0	0	Neither thread at rendezvous location
-1	+1	Thread 2 arrived first, waiting for thread 1
+1	-1	Thread 1 arrived first, waiting for thread 2

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

This only works for 2 threads

How do you make a general n thread barrier?

School of Computing
University of Utah
33
CS 5780

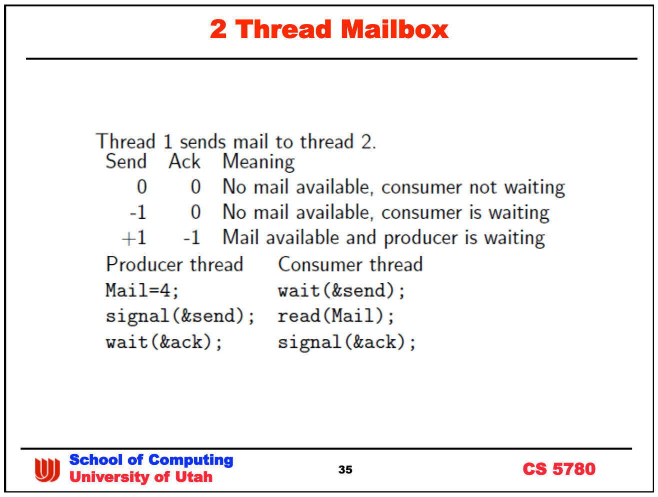


Mutex Sharing or Non-reentrant Code

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);
```

School of Computing
University of Utah
34
CS 5780



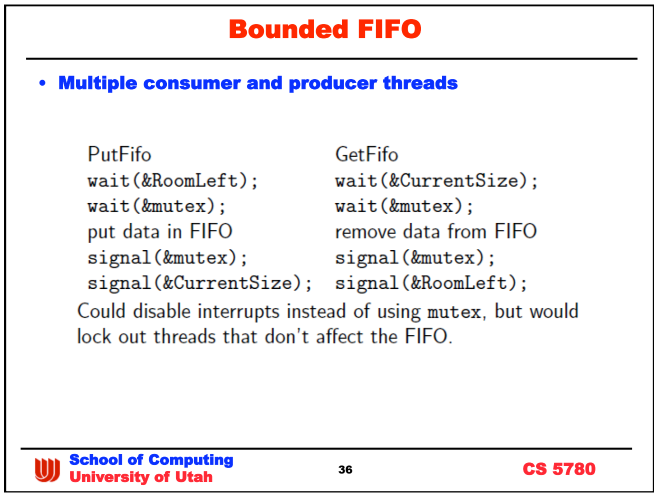
2 Thread Mailbox

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);
```

School of Computing
University of Utah
35
CS 5780



Bounded FIFO

- **Multiple consumer and producer threads**

```
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.

School of Computing
University of Utah
36
CS 5780

Concluding Remarks

- **Threads introduce concurrency**
  - decoupling makes thinking about complex tasks easier
  - there is a cost however
    - » scheduling is required
    - » shared resources requires additional control
      - semaphores are the control mechanism
      - but access must be mutually exclusive

- **Reminder**
  - midterm Tuesday
  - don’t be late