1. Switch debouncing

- Assume a minimum touch time 500ms
- Assume a maximum bounce time 10ms
- On touch
  - signal user, call user function (no latency)
  - Disarm. AddThread(&BounceWait)
- BounceWait
  - Sleep for more than 10, less than 500 ms
  - Rearm.
  - OS_Kill
2. Switch debounce

• Assume a maximum bounce time 10ms
• Interrupt on both rise and fall
  – If it is a rise, signal touch event
  – If it is a fall, signal release event
  – Disarm
  – AddThread(&DebounceTask)
• DebounceTask
  – Sleep for 10 ms
  – Rearm, Set a global with the input pin value
  – OS_Kill

Define latency for this interface

February 21, 2014
Jonathan Valvano
EE445M/EE380L.6
2. Switch debounce

• From Quiz 1 Question 9, 2012

```c
void static DebounceTask(void){
    OS_Sleep(10);   // foreground sleeping, must run within 50ms
    LastPD6 = PD6; // read while it is not bouncing
    GPIO_PORTD_ICR_R = 0x40;    // clear flag6
    GPIO_PORTD_IM_R |= 0x40;    // enable interrupt on PD6
    OS_Kill();
}

void GPIOPortD_Handler(void){
    if(LastPD6 == 0){  // if previous was low, this is rising edge
        (*PD6Task)();    // execute user task
    }
    GPIO_PORTD_IM_R &= ~0x40;   // disarm interrupt on PD6
    OS_AddThread(&DebounceTask);
}
```

February 21, 2014

Jonathan Valvano

EE445M/EE380L.6
2. Switch debounce

• From Quiz 1 Question 9, Spring 2012

```c
PD6

// Switch debounce

DebounceTask runs

2ms max 50ms min

DebounceTask runs

PD6 latency

call (*PD6Task)();

LastPD6

February 21, 2014
Jonathan Valvano
EE445M/EE380L.6
Deadlock conditions

• Mutual exclusion
• Hold and wait
• No preemption of resources
• Circular waiting
Deadlock prevention

• No mutual exclusion
• No hold and wait
  – Ask for all at same time
  – Release all, then ask again for all
• No circular waiting
  – Number all resources
  – Ask for resources in a specific order
Deadlock avoidance

• Is there a safe sequence?
• Tell OS current and future needs
  – Request a resource
  – Specify future requests while holding
  – Yes, if there is one safe sequence
• OS can say no, even if available
  – Google search on Banker’s Algorithm
Deadlock detection

• Add timeouts to semaphore waits
• Cycles in resource allocation graph
• Kill threads and recover resources
  – Abort them all, and restart
  – Abort them one at a time until it runs
Thread A
wait(&bOLED); //1
wait(&bSDC); //4
use OLED and SDC
signal(&bSDC);
signal(&bOLED);

Thread B
wait(&bSDC); //2
wait(&bCAN); //5
use CAN and SDC
signal(&bCAN);
signal(&bSDC);

Thread C
wait(&bCAN); //3
wait(&bOLED); //6
use CAN and OLED
signal(&bOLED);
signal(&bCAN);
### No hold and wait

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
<th>Thread C</th>
</tr>
</thead>
<tbody>
<tr>
<td>wait(&amp;bOLED,&amp;bSDC); use OLED and SDC</td>
<td>wait(&amp;bSDC,&amp;bCAN); use CAN and SDC</td>
<td>wait(&amp;bCAN,&amp;bOLED); use CAN and OLED</td>
</tr>
<tr>
<td>signal(&amp;bOLED,&amp;bSDC);</td>
<td>signal(&amp;bSDC,&amp;bCAN);</td>
<td>signal(&amp;bCAN,&amp;bOLED);</td>
</tr>
</tbody>
</table>

### No circular waiting

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
<th>Thread C</th>
</tr>
</thead>
<tbody>
<tr>
<td>wait(&amp;bOLED); wait(&amp;bSDC); use OLED and SDC</td>
<td>wait(&amp;bSDC); wait(&amp;bCAN); use CAN and SDC</td>
<td>wait(&amp;bOLED); wait(&amp;bCAN); use CAN and OLED</td>
</tr>
<tr>
<td>signal(&amp;bSDC); signal(&amp;bOLED);</td>
<td>signal(&amp;bCAN); signal(&amp;bSDC);</td>
<td>signal(&amp;bCAN); signal(&amp;bOLED);</td>
</tr>
</tbody>
</table>
Where is the deadlock?
Graduate projects ideas

• 1) Extend the OS with more features (do this if two students in group)
  – Efficient with 20 to 50 threads
  – Multiple Mailboxes
  – Multiple Fifos
  – Multiple periodic interrupts
  – Multiple edge-triggered input interrupts
  – Path expression for LCD and serial port
  – Semaphores with timeout
  – Kill foreground threads that finish

• 2) Make your Lab3 OS portable
  – First implement Lab3 on another architecture (each students does their own)
  – Rewrite OS into two parts, OS.c and CPU.c
  – Common OS.c (maximize this part)
  – Separate CPU.c for each architecture (minimize this part)

• 3) Design and test a DMA-based eDisk driver for the LaunchPad board (one-person project)
  – Compare and contrast your Lab5 to FAT

• 4) Write your own malloc and free (one-person project)
  – Copy two examples code out of a book, or off internet
  – Compare and contrast your manager to the existing two implementations

• 5) Design, manufacture, and test a PCB for your robot

• 6) Design and test a DMA-based camera driver for the LaunchPad board (one-person project)
  – See LM3S811 example http://users.ece.utexas.edu/~valvano/arm/Camera_811.zip

• 7) Simple CAN driver without StellarisWare

• 8) Simple node to node Ethernet interface without Stellarisware on new LaunchPad in March

Level of complexity depends on size of group
Priority

• Some tasks are more important than others
• In order to do something first, something else must be second
• When to run the scheduler?
  - Periodically, systick and sleep
  - On OS_Wait
  - On OS_Signal
  - On OS_Sleep, OS_Kill

Reference EE345L book, chapter 5
Priority Scheduler

- Assigns each thread a priority number
  - Problem: How to assign priorities?
  - Solution: Performance measures
- 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
  - Problem: Starvation
  - Solution: Aging
How to find highest priority

- Search all for highest priority ready thread
  - Skip if blocked
  - Skip if sleeping
  - Linear search speed (number of threads)
- Sorted list by priority
  - Chain/unchain as ready/blocked
- Priority bit table (uCOS-II and uCOS-III)
  - See `OSUnMapTbl` in `os_core.c`
  - See `OS_Sched` (line 1606)
  - See `CPU_CntLeadZeros` in `cpu_a.asm`
Adaptive Priority- Aging

• Solution to starvation
• Real and temporary priorities in TCB
• Priority scheduler uses temporary priority
• Increase temporary priority periodically
  – If a thread is not running
• Reset temporary back to real when runs
Rate Monotonic Scheduler

- $n$ tasks that are periodic, running with periods $T_i$
- Priority according to this period
  - more frequent tasks having a higher priority
- Little interaction between tasks

\[
\sum_{i=0}^{n-1} \frac{E_i}{T_i} \leq n\left(2^{1/n} - 1\right) \leq \ln(2)
\]
Exponential Queue

• Multi-level feedback queue
  – Automatically adjusts priority
• High priority to I/O bound threads
  – Block a lot, need low latency
  – Every time it blocks on I/O,
    • Increase priority, Smaller time slice
• Low priority to CPU bound threads
  – Every time it runs to completion
    • Decrease priority, Longer time slice
Exponential Queue

- Exponential comes from doubling/halving
- A) Round robin with variable timeslices
  - Time slices 8,4,2,1 ms
- B) Priority with variable priority/timeslices
  - Time slices 8,4,2,1 ms and priorities 0,1,2,3
I/O Centric Scheduler

- Automatically adjusts priority
- High priority to I/O bound threads
  - I/O needs low latency
  - Every time it issues an input or output,
    - Increase priority by one, shorten time slice
- Low priority to CPU bound threads
  - Every time it runs to completion
    - Decrease priority by one, lengthen time slice
Path expression

• Specify the correct calling order
  – A group of related functions
  – Initialize before use

![Diagram showing the calling order of UART_Init, UART_InChar, UART_OutChar, and UART_Close functions]
Path expression

Each arrow is a ‘1’ in matrix

```
int State=3;  // start in the Closed state
int const Path[4][4]={ /* Init InChar OutChar Close */
/*          column    0        1        2        3   */
/* Init      row 0*/ {   0   ,    1   ,    1   ,    1  },
/* InChar    row 1*/ {   0   ,    1   ,    1   ,    1  },
/* OutChar   row 2*/ {   0   ,    1   ,    1   ,    1  },
/* Close     row 3*/ {   1   ,    0   ,    0   ,    0  }};

void UART_Init(void){
    if(Path[State][0]==0) OS_Kill();  // kill if illegal
    State = 0;                        // perform valid Init
    xxxxx regular stuff xxxxx
}

char UART_InChar(void){
    if(Path[State][1]==0) OS_Kill();  // kill if illegal
    State = 1;                        // perform valid InChar
    xxxxx regular stuff xxxxx
}
```
Performance measures

• Maximum time running with \( I=1 \)
• Percentage of time it runs with \( I=1 \)
• Time jitter on periodic tasks
  \[ \Delta t-\delta t < t_n - t_{n-1} < \Delta t+\delta t \]  
  for all \( n \)
• CPU utilization
  – Percentage time running idle task
• Context switch overhead
  – Time to switch tasks
How long do you test?

- $n =$ number of times $T_1$ interrupts $T_2$
- $m =$ total number of assembly instructions in $T_2$
- Run this test until $n$ greatly exceeds $m$
- Think of this corresponding probability question
  - $m$ different cards in a deck
  - Select one card at random, with replacement
  - What is the probability after $n$ selections (with replacement) that a particular card was never selected?
  - Similarly, what is the probability that all cards were selected at least once?
How long do you test?

```c
Rx_Fifo_Get
0 424846 0x000009B4 4601 MOV r1,r0
1 374028 0x000009B6 481D LDR r0,[pc,#116]  ; if(RxPutPt == RxGetPt ){
2 457111 0x000009B8 6800 LDR r0,[r0,#0x00]
3 402642 0x000009BA 4A1B LDR r2,[pc,#108]
4 204390 0x000009BC 6812 LDR r2,[r2,#0x00]
5 156684 0x000009BE 4290 CMP r0,r2
6 211597 0x000009C0 D101 BNE 0x000009C6
7 242024 0x000009C2 2000 MOV r0,#0x00  ; return(RXFIFOFAIL);
8 3916 0x000009C4 4770 BX lr
9 417 0x000009C6 4818 LDR r0,[pc,#96]  ; *datapt = *(RxGetPt++);
10 828 0x000009C8 6800 LDR r0,[r0,#0x00]
11 1237 0x000009CA 7800 LDRB r0,[r0,#0x00]
12 3099 0x000009CC 7008 STRB r0,[r1,#0x00]
13 1859 0x000009CE 4816 LDR r0,[pc,#88]
14 0 0x000009D0 6800 LDR r0,[r0,#0x00]
15 2266 0x000009D2 1C40 ADDS r0,r0,#1
16 831 0x000009D4 4A14 LDR r2,[pc,#80]
17 0 0x000009D6 6010 STR r0,[r2,#0x00]
18 1870 0x000009D8 4610 MOV r0,r2
19 3090 0x000009DA 6802 LDR r2,[r0,#0x00]
20 5 0x000009DC 4811 LDR r0,[pc,#68]
21 1238 0x000009DE 3020 ADDS r0,r0,#0x20
22 3 0x000009E0 4282 CMP r2,r0  ; if(RxGetPt==&RxFifo[RXFIFOSIZE]){
23 0 0x000009E2 D102 BNE 0x000009EA
24 0 0x000009E4 3820 SUBS r0,r0,#0x20  ; RxGetPt = &RxFifo[0];
25 206 0x000009E6 4A10 LDR r2,[pc,#64]  ; }
26 2471 0x000009E8 6010 STR r0,[r2,#0x00]
27 1651 0x000009EA 2001 MOV r0,#0x01
28 0 0x000009EC E7EA B 0x000009C4  ; return(RXFIFOSUCCESS);}
```
Context Switch time

- Just like the Lab 1 measurement

x=6943ns, o=9936ns, xo=2993ns
Running with I = 1

#define OSCRITICAL_ENTER() { sr = SRSave(); }
#define OSCRITICAL_EXIT() { SRRestore(sr); }

• Record time t1 when I=1
  
  #define OSCRITICAL_ENTER() { t1=OS_Time(); sr = SRSave(); }

• Record time t2 when I=0 again

• Measure difference
  
  #define OSCRITICAL_EXIT() { SRRestore(sr);
  dt=OS_TimeDifference(OS_Time(),t1); }

• Record maximum and total
Time jitter

![Graph showing time jitter](image)

- **Disable/enable interrupts**
- **Using LDREX STREX**
Semaphores have drawbacks

• They are shared global variables
• Can be accessed from anywhere
• No connection between the semaphore and the data being controlled by the semaphore
• Used both for critical sections (mutual exclusion) and coordination (scheduling)
• No control or guarantee of proper usage
Monitors

• Proper use is enforced
• Synchronization attached to the data
• Removes hold and wait
• Threads enter
  – one active at a time

Queue of waiting threads trying to enter the monitor

http://lass.cs.umass.edu/~shenoy/courses/fall08/lectures/Lec10.pdf

February 21, 2014
Jonathan Valvano
EE445M/EE380L.6
Monitors

• Lock
  – Only one thread active at a time
  – Must have lock to access condition variables

• One or more condition variables
  – If cannot complete, leave data consistent
  – Threads can sleep inside by releasing lock
  – Wait (acquire or sleep)
  – Signal (if any waiting, wakeup else nop)
  – Broadcast
## FIFO Monitor

<table>
<thead>
<tr>
<th>Put(item)</th>
<th>Get()</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) lock-&gt;Acquire();</td>
<td>1) lock-&gt;Acquire();</td>
</tr>
<tr>
<td>2) put item on queue;</td>
<td>2) while queue is empty</td>
</tr>
<tr>
<td>3) conditionVar-&gt;Signal();</td>
<td>conditionVar-&gt;Wait(lock);</td>
</tr>
<tr>
<td>4) lock-&gt;Release();</td>
<td>3) remove item from queue;</td>
</tr>
<tr>
<td></td>
<td>4) lock-&gt;Release();</td>
</tr>
<tr>
<td></td>
<td>5) return item;</td>
</tr>
</tbody>
</table>

[http://lass.cs.umass.edu/~shenoy/courses/fall08/lectures/Lec10.pdf](http://lass.cs.umass.edu/~shenoy/courses/fall08/lectures/Lec10.pdf)
Hoare vs Mesa Monitor

Hoare wait
if(FIFO empty)
    wait(condition)
Mesa wait
while(FIFO empty)
    wait(condition)
Kahn Process Network

- Blocking read
- Non-blocking writes (never full)
- Tokens are data (no time stamp)
Kahn Process Network

- Deterministic
  - Same inputs result in same outputs
  - Independent of scheduler
- Non-blocking writes (never full)
- Monotonic
  - Needs only partial inputs to proceed
  - Works in continuous time
void Process3(void){
  long inA, inB, out;
  while(1){
    while(AFifo_Get(&inA)){};
    while(BFifo_Get(&inB)){};
    out = compute(inA,inB);
    CFifo_Put(out);
  }
}

void Process3(void){
  long inA, inB, out;
  while(1){
    if(AFifo_Size()==0){
      while(BFifo_Get(&inB)){};
      while(AFifo_Get(&inA)){};
    } else{
      while(AFifo_Get(&inA)){};
      while(BFifo_Get(&inB)){};
    }
    out = compute(inA,inB);
    CFifo_Put(out);
  }
}
Kahn Process Network

• Strictly bounded
  – Prove it never fills (undecidable)
  – dependent of scheduler

• Termination
  – All processed blocked on input

• Scheduler
  – Needs only partial inputs to proceed
  – Works in real time
Kahn Process Network

- Try to find a mathematical proof
- Experimentally adjust FIFO size
  - Needs a realistic test environment
  - Profile/histogram DataAvailable for each FIFO
  - Leave the profile in delivered machine
- Dynamically adjust size with malloc/free
- Use blocking write (not a KPN anymore)
- Discard the data
Thread switch with PSP (1)

- Bottom 8 bits of LR
- 0xE1 11110001 Return to Handler mode MSP (using floating point state)
- 0xE9 11101001 Return to Thread mode MSP (using floating point state)
- 0xED 11101101 Return to Thread mode PSP (using floating point state)
- 0xF1 11110001 Return to Handler mode MSP
- 0xF9 11111001 Return to Thread mode MSP
- **0xFD 11111101 Return to Thread mode PSP**

![Context Switch Diagram]

February 21, 2014

EE445M/EE380L.6
Thread switch with PSP (2)

; everyone uses MSP (Program 4.9 from book)

SysTick_Handler

CPSID   I  ; 2) Prevent interrupt during switch
PUSH    {R4-R11}  ; 3) Save remaining regs r4-11
LDR     R0, =RunPt  ; 4) R0=pointer to RunPt, old thread
LDR     R1, [R0]  ;  R1 = RunPt
STR     SP, [R1]  ; 5) Save SP into TCB
LDR     R1, [R1,#4]  ; 6) R1 = RunPt->next
STR     R1, [R0]  ;  RunPt = R1
LDR     SP, [R1]  ; 7) new thread SP; SP = RunPt->sp;
POP     {R4-R11}  ; 8) restore regs r4-11
CPSIE   I  ; 9) run with interrupts enabled
BX      LR  ; 10) restore R0-R3,R12,LR,PC,PSR
Thread switch with PSP (3)

; tasks use PSP, OS/ISR use MSP, Micrium OS-II
SysTick_Handler ; 1) R0-R3,R12,LR,PC,PSR on PSP
CPSID I ; 2) Prevent interrupt during switch

MRS R2, PSP ; R2=PSP, the process stack pointer
SUBS R2, R2, #0x20
STM R2, {R4-R11} ; 3) Save remaining regs r4-11
LDR R0, =RunPt ; 4) R0=pointer to RunPt, old thread
LDR R1, [R0] ; R1 = RunPt
STR R2, [R1] ; 5) Save PSP into TCB
LDR R1, [R1,#4] ; 6) R1 = RunPt->next
STR R1, [R0] ; RunPt = R1
LDR R2, [R1] ; 7) new thread PSP in R2
LDM R2, {R4-R11} ; 8) restore regs r4-11
ADDS R2, R2, #0x20
MSR PSP, R2 ; Load PSP with new process SP
ORR LR, LR, #0x04 ; 0xFFFFFFFD (return to thread PSP)
CPSIE I ; 9) run with interrupts enabled
BX LR ; 10) restore R0-R3,R12,LR,PC,PSR

February 21, 2014
Jonathan Valvano
EE445M/EE380L.6

OS calls implemented with TRAP
Reflections

• Use the logic analyzer
  – Visualize what is running
• Learn how to use the debugger
• What to do after a thread calls Kill?
• Breakpoint inside ISR
  – Does not seem to single step into ISR