Graduate Projects Ideas

1. Extend the OS with more features (do this if two students in group)
   - Efficient with 20 to 50 threads
   - Multiple cores (real-time scheduling algorithms & implementation)
   - Multiple Mailboxes, FIFOs
   - Multiple periodic/edge-triggered interrupts
   - Path expressions
   - Semaphores with timeout, priority inheritance/ceiling (algorithms & implementation)
   - Kill foreground threads that finish

2. Make your Lab3 OS portable and port to another platform
   - 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 memory management
   - Heap, malloc and free (one-person project)
   - Virtual memory, paging (two or more students)

5. Design, manufacture, and test a PCB for your or other robot (e.g. Freescale platform)

6. Design and test a DMA-based camera driver for the LaunchPad board
   - See LM3S811 example http://www.ece.utexas.edu/~valvano/arm/Camera_811.zip (one person project)
   - Implement object detection & recognition (self-driving car) (two or more students)

7. Networking, Internet-of-Things (IoT)
   - Port a TCP/IP stack onto board (e.g. using external WiFi module via UART)
   - Have robots communication with each other or base station (vehicle-to-vehicle / vehicle-to-)

Level of complexity depends on size of group

Due end of Feb
Semaphores

• \( P() \) or \( \text{wait}() \)
  – Dutch word \textit{proberen}, to test
  – \textit{probeer te verlagen}, try to decrease
  – \text{OS\_Wait} \text{ OS\_SemPend}

• \( V() \) or \( \text{signal}() \)
  – Dutch word \textit{verhogen}, to increase
  – \text{OS\_Signal} \text{ OS\_SemPost}

Semaphore Meaning

• Counting semaphore
  – Number of elements stored in FIFO
  – Space left in the FIFO
  – Number of printers available

• Binary semaphore (= mutex = flag)
  – Free (1), busy (0)
  – Event occurred (1), not occurred (0)
Spin-Lock Binary Semaphore

- `bWait(&s);`  
  - `s==0`  
    - `s==1`  
    - `s=0`  
      - `return`
- `bSignal(&s);`  
  - `s==0`  
    - `s=1`  
      - `return`

How do we use this to solve critical sections?  
Why is this a good solution for critical sections?

What does the semaphore mean?  
What would be a better name for `s`?  
What about atomic?

Spin-Lock Counting Semaphore

- `wait`  
  - `s==0`  
    - `s>0`  
      - `s=s-1`  
      - `return`
- `signal`  
  - `s=s+1`  
    - `return`

What does the semaphore mean?  
What about atomic?
Spin-Lock Semaphores

| OS_Wait ; R0 points to counter | void OS_Wait(long *s) {
| LDREX R1, [R0] ; counter     | DisableInterrupts();
| SUBS R1, #1 ; counter -1,    | while(*s <= 0){
| ITT PL ; ok if >= 0          | EnableInterrupts();
| STREXPL R2, R1, [R0] ; try update | DisableInterrupts();
| CMPPL R2, #0 ; succeed?      | }
| BNE OS_Wait ; no, try again  | (*s) = (*s) - 1;
| BX LR                         | EnableInterrupts();
| OS_Signal ; R0 points to counter |}
| LDREX R1, [R0] ; counter     | void OS_Signal(long *s) {
| ADD R1, #1 ; counter + 1     | long status;
| STREX R2, R1, [R0] ; try update | status = StartCritical();
| CMP R2, #0 ; succeed?        | (*s) = (*s) + 1;
| BNE OS_Signal ; no, try again| EndCritical(status);
| BX LR                         |}

LDREX
STREX

Cortex-M3/M4F Instruction Set, pg. 50

Mailbox

MailBox_Send(...)  MailBox_Recv(...)
- bWait(&BoxFree)  - bWait(&DataValid)
- Put data into Mailbox  - Retrieve data from Mailbox
- bSignal(&DataValid)  - bSignal(&BoxFree)

Consumer ➔ Send Mailbox ➔ Recv Display

What do the semaphores mean?  What are the initial values?
What if we remove bWait(&BoxFree) and bSignal(&BoxFree)?
FIFO, Queue, or Pipe

**FIFO_Put**
- Wait(&DataRoomLeft)
- Disable Interrupts
- Enter data into Fifo
- Enable Interrupts
- Signal(&DataAvailable)

**FIFO_Get**
- Wait(&DataAvailable)
- Disable Interrupts
- Remove data from Fifo
- Enable Interrupts
- Signal(&DataRoomLeft)

**FIFO_Put**
- Wait(&DataRoomLeft)
- bWait(&Mutex)
- Enter data into Fifo
- bSignal(&Mutex)
- Signal(&DataAvailable)

**FIFO_Get**
- Wait(&DataAvailable)
- bWait(&Mutex)
- Remove data from Fifo
- bSignal(&Mutex)
- Signal(&DataRoomLeft)

---

No Background Wait

- Redo Mailbox if **Send** in background
- Redo Fifo if **Put** in background (RX)
- Redo Fifo if **Get** in background (TX)
Cooperative Spin-Lock

Cooperative spin-lock

Regular spin-lock

Why would you want a timeout error?
How would you implement timeout?

Could be implemented with a catch and throw

Cooperative Semaphores

void OS_Wait(long *s)
{
    DisableInterrupts();
    while(*s <= 0){
        EnableInterrupts();
        OS_Suspend();
        DisableInterrupts();
    }
    (*s) = (*s) - 1;
    EnableInterrupts();
}

void OS_Signal(long *s)
{
    long status;
    status = StartCritical();
    (*s) = (*s) + 1;
    EndCritical(status);
}
Blocking Semaphore (Lab 3)

- Recapture time lost in the spin-lock
  - No spin operation, wakeup only on signal
  - Eliminate wasted time running threads that are not doing work (e.g., waiting)
- Implement **bounded waiting**
  - Once thread calls \texttt{Wait} and is not serviced,
  - There are a finite number of threads that will go ahead

![Blocking Semaphore Diagram]

What does the semaphore mean?
What about atomic?
Blocking Semaphore (V1)

• All threads exist on circular TCB list (active and blocked)
  – Each semaphore simply has a Value
  – No blocked threads if semaphore Value ≥ 0
    • e.g., if Value is -2, then two threads are blocked
  – No information about which thread has waited longest
  – Add to TCB, a BlockPt, 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 Scheduler
  – Find the next active thread from the TCB list
  – Only run threads with BlockPt equal to null

OS_Wait(Sema4Type *semaPt)
1) Disable interrupts, I=1
2) Decrement the semaphore counter, S=S-1
   (semaPt->Value)--; 
3) If the Value<0 then this thread will be blocked
   specify this thread is blocked to this semaphore
   RunPt->BlockPt = semaPt;
   suspend thread;
4) Enable interrupts, I=0
### Blocking Semaphore (V1)

**OS_Signal(Sema4Type *semaPt)**

1. Save I bit, then disable interrupts
2. Increment the semaphore Value, \( S = S + 1 \)
   
   \( (semaPt->Value)++; \)
3. If Value \( \leq 0 \) 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 `BlockPt == semaPt`
   
   set the `BlockPt` of this TCB to null
4. Restore I bit

### Blocking Semaphore (V2)

- Each semaphore has a blocked TCB linked list
  
  - contains the threads that are blocked
  
  - empty if semaphore Value \( \geq 0 \)
    
    - e.g., if Value == -2, then two threads are blocked
  
  - order on blocked list determine sequence of blocking
  
  - sequence of blocking determine which to wake up

How is the scheduler different?
Blocking Semaphore (V2)

**OS_Wait(Sem4Type *semaPt)**
1) Save the I bit and disable interrupts
2) Decrement the semaphore counter, S=S-1
   \[
   \text{(semaPt->Value)}--; \\
   \]
3) If the **Value < 0** then this thread will be blocked
   set the status of this thread to blocked, specify this thread blocked on this semaphore, suspend thread
4) Restore the I bit

**OS_Signal (Sema4Type *semaPt)**
1) Save I bit, then disable interrupts
2) Increment the semaphore counter, S=S+1
   \[
   \text{(semaPt->Value)}++; \\
   \]
3) If the **Value \leq 0** then
   Wake up one thread from the TCB linked list
   Bounded waiting -> the one waiting the longest
   Priority -> the one with highest priority
   Move TCB of the “wakeup” thread from the blocked list to the active list
   **What to do with the thread that called OS_Signal?**
   Round robin -> do not suspend
   Priority -> suspend if wakeup thread is higher priority
4) Restore I bit
Semaphore Applications

- Sequential execution
  - Run-A then Run-B then Run-C
- Rendezvous
- Event trigger
  - Event-A and Event-B
  - Event-A or Event-B
- Fork and join
- Readers-Writers Problem

Readers-Writers Problem

**Reader Threads**
1) Execute `ROpen(file)`
2) Read information from `file`
3) Execute `RClose(file)`

**Writer Threads**
1) Execute `WOpen(file)`
2) Read information from `file`
3) Write information to `file`
4) Execute `WClose(file)`
Readers-Writers Problem

**ReadCount=0**: number of Readers that are open  
**mutex=1**: semaphore controlling access to **ReadCount**  
**wrt=1**: semaphore is true if a writer is allowed access

```c
ROpen
wait(&mutex);
ReadCount++;
if(ReadCount==1) wait(&wrt)
signal(&mutex);

RClose
wait(&mutex);
ReadCount--;
if(ReadCount==0) signal(&wrt)
signal(&mutex);

WOpen
wait(&wrt);

WClose
signal(&wrt);
```

**Advanced Topics (Grad Students)**

- Bounded waiting  
- Time-out  
- Deadlock detection  
  - Wait-for-graph  
  - Resource allocation graph

Two names for the same thing  
Works for single instance resources
Deadlock

• Conditions
  – Mutual exclusion
  – Hold and wait
  – No preemption of resources
  – Circular waiting

Where is the deadlock?

Resource Allocation Graph

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);

Thread A
Thread C
Thread B
SDC
Assignment
Request
Assignment
OLED
Assignment
Request
CAN
Assignment
Request
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

Prevention

- No hold and wait

Thread A
  - wait(&bOLED,&bSDC);
  - use OLED and SDC
  - signal(&bOLED,&bSDC);

Thread B
  - wait(&bSDC,&bCAN);
  - use CAN and SDC
  - signal(&bSDC,&bCAN);

Thread C
  - wait(&bCAN,&bOLED);
  - use CAN and OLED
  - signal(&bCAN,&bOLED);

- No circular wait

Thread A
  - wait(&bOLED);
  - wait(&bSDC);
  - use OLED and SDC
  - signal(&bSDC);
  - signal(&bOLED);

Thread B
  - wait(&bSDC);
  - wait(&bCAN);
  - use CAN and SDC
  - signal(&bCAN);
  - signal(&bSDC);

Thread C
  - wait(&bCAN);
  - wait(&bOLED);
  - use CAN and OLED
  - signal(&bOLED);
  - signal(&bCAN);
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
• Detect cycles in resource allocation graph
• Kill threads and recover resources
  – Abort them all, and restart
  – Abort them one at a time until it runs
Semaphore Drawbacks

- 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

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

**Put**(item):
1) lock->Acquire();
2) put item on queue;
3) conditionVar->Signal();
4) lock->Release();

**Get():**
1) lock->Acquire();
2) while queue is empty
   conditionVar->Wait(lock);
3) remove item from queue;
4) lock->Release();
5) return item;

[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

- Signal() switches immediately vs. later

Hoare wait:
if(FIFO empty)
wait(condition)

Mesa wait:
while(FIFO empty)
wait(condition)

Testing (1)

- How long do you test?
  - \( n \) = number of times T1 interrupts T2
  - \( m \) = total number of assembly instructions in T2
  - Run 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?
Testing (2)

```
Rx_Fifo_Get
0 02446 0xa0000FB4 4601 MOV r1,r0 ;int RxFifo_Get(rxDataType *datapt){
1 374028 0xa0000FB6 481D LDR r0,[pc,#116] ; if(RxPutPt == RxGetPt){
2 457111 0xa0000FB8 6800 LDR r0,[r0,#0x00]
3 402642 0xa0000FBA 4A1B LDR r2,[pc,#108]
4 204390 0xa0000FBC 6812 LDR r2,[r2,#0x00]
5 156684 0xa0000FBE 4290 CMP r0,r2
6 3916 0xa0000FCC 4770 BX lr ; return(RXFIFOFAIL);
7 9 0x000009B4 4601 MOV r1,r0 ;int RxFifo_Get(rxDataType *datapt){
8 8 0x000009B6 481D LDR r0,[pc,#116] ; if(RxPutPt == RxGetPt){
9 7 0x000009B8 6800 LDR r0,[r0,#0x00]
10 6 0x000009BA 4A1B LDR r2,[pc,#108]
11 5 0x000009BB 6812 LDR r2,[r2,#0x00]
12 4 0x000009BD 4290 CMP r0,r2
13 3 0x000009BF 4770 BX lr ; return(RXFIFOFAIL);
14 2 0x000009C0 D101 BNE 0x000009C6 ;    return(RXFIFOFAIL);
15 1 0x000009C2 2000 MOVS r0,#0x00
16 0 0x000009C4 4770 BX lr ;  }
17 0 0x000009C6 4818 LDR r0,[pc,#96] ;  *datapt = *(RxGetPt++);
18 0 0x000009C8 6800 LDR r0,[r0,#0x00]
19 0 0x000009CA 7800 LDRB r0,[r0,#0x00]
20 0 0x000009CC 7008 STRB r0,[r1,#0x00]
21 0 0x000009CE 4816 LDR r0,[pc,#88]
22 0 0x000009D0 6800 LDR r0,[r0,#0x00]
23 0 0x000009D2 1C40 ADDS r0,r0,#1
24 0 0x000009D4 4A10 LDR r2,[pc,#64]
25 0 0x000009D6 6010 STR r0,[r2,#0x00]
26 0 0x000009D8 4610 MOV r0,r2
27 0 0x000009DA 6802 LDR r2,[r0,#0x00]
28 0 0x000009DC 4770 BX lr ; return(RXFIFOSUCCESS);}
```

FIFO_4C123

Path Expressions (1)

- Specify and enforce correct calling order
  - A group of related functions (e.g., I/O)
  - Initialize before use

```
UART_Init
UART_InChar
UART_OutChar
UART_Close
```

Book Section 4.6.2
Path Expressions (2)

```c
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
  // xxxx regular stuff xxxx
}

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

Final exam 2004, Q9

Performance Measures

- Maximum time running with I=1
- Percentage of time it runs with I=1
- Time jitter $\delta t$ on periodic tasks
  \[ T_i - \delta t < t_n - t_{n-1} < T_i + \delta t \quad \text{for all } n \]
- CPU utilization
  - Percentage time running idle task
- Context switch overhead
  - Time to switch tasks
Context Switch Time

- Just like the Lab 1 measurement

![Graph showing context switch time with intervals]

Running with $I = 1$

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

- Record time $t1$ when $I=1$
  ```c
  #define OSCRITICAL_ENTER() { t1=OS_Time(); sr = SRSave(); }
  ```
- Record time $t2$ when $I=0$ again
- Measure difference
  ```c
  #define OSCRITICAL_EXIT() { SRRestore(sr);
  dt=OS_TimeDifference(OS_Time(),t1); }
  ```
- Record maximum and total
Summary

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