Managing Coroutines in ST Systems

What we will learn

- DS are embedded systems that are responsible for reading sensor devices, then signaling actuator devices to perform some action; the combination of required cooperation and embedded system applications suggests that most ST software is really just one program that has multiple tasks to perform
- We study the part of the OS that handles the implementation of computational abstraction, AUCs: coroutine management
- Virtual abstract machines and coroutine-based AUCs
- VMs as Finite State Machines
- The success of the system depends on the cooperation of each AUC.
- System call mechanisms and data structures
- Scheduling

Coroutine management

<table>
<thead>
<tr>
<th>ST OS</th>
<th>Bag of functions &amp; procedures</th>
<th>weak AUCs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Coroutines</td>
<td></td>
</tr>
<tr>
<td>MT OS</td>
<td>Threads</td>
<td>AUCs</td>
</tr>
<tr>
<td>MP OS</td>
<td>Processes</td>
<td>AUCs</td>
</tr>
</tbody>
</table>

The coroutine model

- Only one coroutine can use the CPU at a time, and only one coroutine is logically enabled to execute at any given moment
- Each coroutine supports:
  - concurrency by resuming other coroutines whenever it foresees inactivity
  - overlapped operation of the CPU with I/O devices, provided that:
    - The computations performed by the AUC are arranged so that while the I/O operation is taking place, the AUC has other work to do.
    - The programming language and OS provide tools to allow the AUC to start an I/O operation, then to poll the device to see when the operation has completed.
Virtual abstract machines

- The goal for an ST OS is to provide a suitable AUC without relying on interrupts or CPU modes.
- Coroutines are a simplified AUC that captures some of the autonomy of threads and processes, yet which executes under the control of a single threaded user program.
- The coroutine AUC is an execution environment that is appropriate for executing a sequential computation under the coroutine assumptions: is a virtual machine
Virtual abstract machines

- A VM differs from a physical machine in that it includes an extended set of instructions, each of which is implemented as procedures.
- It is possible to write another program (the AUC manager part of the OS), that simulates the existence of multiple virtual machines.
Operating Systems and Distributed Systems

System calls

• \texttt{int CreateAUC(void *tFunction, void *argList, char *name)}: creates a coroutine that begins to execute at the entry point of the function named by the `tFunction` argument. The `tFunction()` function is passed the argument, `argList`. The coroutine can be referred to by the given string `name`. The coroutine will not begin to execute (at its entry point) until it is resumed.

• \texttt{int LookupHandle(char *name)}: looks up the coroutine specified by the `name` argument. Returns the handle associated with the named coroutine. Otherwise, returns -1.

• \texttt{int Resume(int cortnHandle)}: when a coroutine (including the main program) resumes the coroutine with the handle value of `cortnHandle`, the calling coroutine is suspended, and the resumed coroutine begins to execute using its internal state as created by `CreateAUC()` if it has not previously been resumed, or by using its state at the time it last called `Resume()`. If the designated coroutine does not exist, returns a value of -1 and continues as if it had been resumed by the nonexistent coroutine. If succeeded, it will not return until another coroutine resumes this called, in which case the value returned will be the handle of the resuming coroutine.

• \texttt{void * Return(int cortnHandle)}: the same behavior as `Resume()` except that it terminates the calling coroutine.

• \texttt{void * Exit(void *result)}: terminates the entire sequential computation, including all coroutines. It is normally only called by the main program coroutine.

System calls

FSM: Basic Model
**OS must keep track of the execution state of each VM**

**Level principle**

- Active entities are data structures when viewed from a lower level

**AUC Table**

<table>
<thead>
<tr>
<th>Unique internal identification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current state</td>
</tr>
<tr>
<td>Memory load map</td>
</tr>
<tr>
<td>Open Devices</td>
</tr>
<tr>
<td>Open files</td>
</tr>
<tr>
<td>Other system resources</td>
</tr>
<tr>
<td>CPU context: register contents (CPU general registers, address of the next instruction to execute, Code/Data/Stack segment registers, and so on)</td>
</tr>
<tr>
<td>If waiting for I/O: device identification and I/O arguments</td>
</tr>
</tbody>
</table>

**AUC Descriptor**
typedef void *xCORoutineHandle;

/* Defines the prototype to which co-routine functions must conform. */
typedef void (*crCOROUTINE_CODE)( xCORoutineHandle, unsigned portBASE_TYPE );

typedef struct corCoRoutineControlBlock {
    crCOROUTINE_CODE pxCORoutineFunction;
    xListItem xGenericListItem; /* List item used to place the CRCB in ready and blocked queues. */
    xListItem xEventListItem; /* List item used to place the CRCB in event lists. */
    unsigned portBASE_TYPE uxPriority; /* The priority of the co-routine in relation to other co-routines. */
    unsigned portBASE_TYPE uxIndex; /* Used to distinguish between co-routines when multiple co-routines use the same co-routine function. */
    unsigned portSHORT uxState; /* Used internally by the co-routine implementation. */
} corCRCB; /* Co-routine control block. */

crDELAY( xHandle, xTicksToDelay )
crSTART( xHandle );

vCoRoutineSchedule( void )

Overlapping via cooperative behavior

• Trick: driver as a function  ->  driver as a coroutine

• Trick: driver as a function  ->  driver as a coroutine
Operating Systems and Distributed Systems

Overlapping via cooperative behavior

- Read() or Write() system call effectively resumes the driver to begin the I/O operation
- The driver starts the device and Resumes an AUC
- Whenever any AUC resumes the driver, the driver polls the device, determines that it is busy, and then resumes the coroutine that resumed it
- When the device completes, the driver consults its internal data structures to determine which AUC started the device, and then resumes that AUC
- This technique depends on the cooperative behavior of all AUCs: Every AUC must periodically resume the driver for every device, otherwise, the driver cannot check the device for completion

Overlapping via cooperative behavior: simplified model

- Assume we have Blocking Read() or Write() system calls
- AUC1 makes the blocking system call: while waiting, Resume (AUC2).
- AUC2 periodically Resume(AUC1):
  - If AUC1 blocked Resume(AUC2);
  - Else Return(AUC2)

FSM: revised model

- Problem: How can the language runtime system call functions if they are not actually linked into the memory load map?
- Dynamic linking: create an intermediary runtime mechanism that links the OS function when it is called by the application
- Solution: use a system call stub = skeletal function that:
  - exports the OS function prototype
  - contains an indirect address of the target OS function

System call mechanism
System call mechanism

Real stuff: adding a system call to SOS

1. Implementation
   - we add prefix `ker_` to the name of the system

```c
int8_t ker_sys_foo ( int32_t bar )
{
    // system call needs to know which module has just called
    sos_pid_t calling_pid = ker_get_current_pid ();

    //......

    // Do something about 'bar' related to 'calling_pid'
    //......

    return SOS_OK;
}
```

Real stuff: adding a system call to SOS

2. Add this new implementation to the jumptable
   - processor dependent: $SOSROOT/processor/avr/sys_jmptable.S

```
jmp ker_sys_shm_wait       ; 26
jmp ker_sys_shm_stopwait   ; 27
jmp ker_sys_foo            ; 28
```
typedef int8_t (*sys_foo_func_t)(int32_t bar); //defined for type casting

static inline int8_t sys_foo(int32_t bar)
{
#ifdef SYS_JUMP_TBL_START
  return ((sys_foo_func_t)(SYS_JUMP_TBL_START+SYS_JUMP_TBL_SIZE*28))(bar);
#else //Jumptable not implemented
  return ker_sys_foo(bar);
#endif
}

Operating Systems and Distributed Systems

Real stuff: adding a system call to SOS

• 3. Add system call interface to
  - $$SOSROOT/kernel/include/sys_module.h$$

Operating Systems and Distributed Systems

Real stuff: adding a system call to SOS

• 4. Declare the function prototype for the emulated version

```c
#ifdef SYS_JUMP_TBL_START
  //\cond NOTYPEDEF
  void* ker_sys_malloc(uint16_t size);
  void* ker_sys_realloc(void* ptr, uint16_t newSize);
  ...
  int8_t ker_sys_foo(int32_t bar);
  \endcond
#endif
```

Operating Systems and Distributed Systems

Single program, single device

```c
{EntryPoint:
  runDiagnostics();
  determineMachineStatus();
  initMachine(...);
  /* Main part of the program begins here */
  while(TRUE) {
    preOpnProcessing(...);
    read(DEVICE, inData);
    switch (parseData(inData)) {
      case 0: handleCase_0(...);
      case 1: handleCase_1(...);
      ...
      default: handleCase_N(...);
    }
    postOpnProcessing(...);
  }
  halt();
}
```

Example: a thermostat

Operating Systems and Distributed Systems

Single program, single device: real-time task

\[ L <= P - S \]

```c
{EntryPoint:
  runDiagnostics();
  determineMachineStatus();
  initMachine(...);
  /* Main part of the program begins here */
  while(TRUE) {
    preOpnProcessing(...);
    read(DEVICE, inData);
    switch (parseData(inData)) {
      case 0: handleCase_0(...);
      case 1: handleCase_1(...);
      ...
      default: handleCase_N(...);
    }
    postOpnProcessing(...);
  }
  halt();
}
```

Deadlines

\[ L = \text{latency} \]

Operating Systems and Distributed Systems
Single program, multiple devices

{  
    MainEntryPoint:
    -
        while(TRUE) {
            preLoopProcessing(...);
            for(i=0; i<M; i++) {
                preOpnProcessing(...);
                if(poll(DEVICE[i])) {
                    read(DEVICE[i], inData[i]);
                    switch (inData.cmd) {
                        case 0: handleCase_0(...);
                        case 1: handleCase_1(...);
                        ...
                        default: handleCase_N(...);
                    }
                    postOpnProcessing(...);
                }
                if(poll(DEVICE[i])) {
                    read(DEVICE[i], inData[i]);
                    switch (inData.cmd) {
                        case 0: handleCase_0(...);
                        case 1: handleCase_1(...);
                        ...
                        default: handleCase_N(...);
                    }
                }
                postOpnProcessing(...);
            }
        }
}

Assumption: $P_i = P_j, \forall i, j$

Operating Systems and Distributed Systems

Earliest deadline first scheduling

/* entry point of the program*/
{
    MainEntryPoint:
    -
        initDeadlines(...); /* A queue of devices ordered by the */
        /* the time of the next required */
        /* service for any device */
        */
        while(TRUE) {
            deltaOtherProcessing(...);
            if(nextDeadline(...)) {
                preOpnProcessing(...);
                serviceDevice(j);
                setNextDeadline(j); /* For this service into the deadline */
                /* queue. */
            }
            postOpnProcessing(...);
        }
        postLoopProcessing(...);
        exitTProcessing();
}

Operating Systems and Distributed Systems

...and with different periods

{  
    MainEntryPoint:
    -
        while(TRUE) {
            /* Service devices with period up to 10 ms */
            L1: for(j=0; j<M'; j++) {
                serviceDevice(j);
                postOpnProcessing(...);
            }
            /* Service devices with period up to 100 ms */
            L2: for(j=M'; j<M''; j++) {
                serviceDevice(j);
                postOpnProcessing(...);
                if(deadlineApproaching(level_1))
                    goto L1;
            }
            /* Service devices with period up to 1 second */
            L3: for(j=M''; j<M'''; j++) {
                serviceDevice(j);
                postOpnProcessing(...);
                if(deadlineApproaching(level_2))
                    goto L2;
            }
            postOpnProcessing(...);
            exitTProcessing();
        }
}

Summary

• ST operating systems that provide a rudimentary VM model in which each coroutine AUC executes. Each coroutine executes on its own VM, and the OS provides the functions that manage the virtual machines.
• Blocking I/O operations can be introduced to facilitate such overlap: redesign of the AUC manager
• Embedded systems periodically read/write the attached device in the absence of interrupts: the software must be assured of servicing the device within a strict deadline – determined by the period during which the device needs to be read or written: necessary to determine the amount of time (latency) that could expire while various tasks execute.
• A single-program, multiple-device program has more work to do, packaged inside of tasks, and requiring more complex latency analysis. Complexity grows by adding devices to the system.

Operating Systems and Distributed Systems