Events, Co-routines, Continuations and Threads
- OS (and application) Execution Models

System Building
- General purpose systems need to deal with
  - Many activities
    - potentially overlapping
    - may be interdependent
  - Activities that depend on external phenomena
    - may requiring waiting for completion (e.g. disk read)
    - reacting to external triggers (e.g. interrupts)
- Need a systematic approach to system structuring

Events
- External entities generate (post) events.
  - keyboard presses, mouse clicks, system calls
- Event loop waits for events and calls an appropriate event handler.
  - common paradigm for GUIs
- Event handler is a function that runs until completion and returns to the event loop.

Event Model
- The event model only requires a single stack
  - All event handlers must return to the event loop
    - No blocking
    - No yielding
  - No preemption of handlers
    - Handlers generally short lived
  - No concurrency issues within a handler

Event-based kernel on CPU with protection

Event-based kernel on CPU with protection

Co-routines

- A subroutine with extra entry and exit points
- Via `yield()` supports long running subroutines
  - subtle variations (yielddo, asymmetric and symmetric)
- Also called Fibers

**Co-routines**

- Usually implemented with a stack per routine
- `yield()` saves state of routine A and starts routine B
  - or resumes B’s state from its previous `yield()` point
- No preemption
- No concurrency issues/races as globals are exclusive between `yield()`
- Implementation strategy?

**A hypothetical yield()**

```c
yield:
    /*
     * a0 contains a pointer to the previous routine's struct.
     * a1 contains a pointer to the new routine's struct.
     * The registers get saved on the stack, namely:
     *    s0, ra
     */
    */ Allocate stack space for saving 11 registers. 11*4 = 44 */
    addi sp, sp, -44

    /* Save the registers */
    sw ra, 40(sp)
    sw gp, 36(sp)
    sw $s8, 32(sp)
    sw $s7, 28(sp)
    sw $s6, 24(sp)
    sw $s5, 20(sp)
    sw $s4, 16(sp)
    sw $s3, 12(sp)
    sw $s2, 8(sp)
    sw $s1, 4(sp)
    sw $s0, 0(sp)

    /* Store the old stack pointer in the old pcb */
    sw sp, 0(a0)
```

Save the registers that the `C` procedure calling convention expects preserved
Coroutines

- What about subroutines combined with coroutines
  - i.e. what is the issue with calling subroutines?
- Subroutine calling might involve an implicit yield()
  - potentially creates a race on globals
  - either understand where all yields lie, or
  - cooperative multithreading

Cooperative Multithreading

- Also called green threads
- Conservative assumes a multithreading model
  - i.e. uses synchronisation to avoid races,
  - and makes no assumption about subroutine behaviour
  - it can potentially yield()

A Thread

- Thread attributes
  - processor related
    - memory
    - program counter
    - stack pointer
    - registers (and status)
  - OS/package related
    - state (running/blocked)
    - identity
    - scheduler (queues, priority)
    - etc...

Thread Control Block

- To support more than a single thread we need store thread state and attributes
- Stored in thread control block
  - also indirectly in stack
Thread A and Thread B

Thread A state currently loaded
Thread B state stored in TCB B
Thread switch from A → B

Note: registers and PC can be stored on the stack, and only SP stored in TCB

Approx OS

```c
mips_switch()
{
    struct thread *cur, *next;
    next = scheduler();
    /* update curthread */
    cur = curthread;
    curthread = next;
    /* Call the machine-dependent code that actually does the */
    /* context switch. */
    mips_switch(cur->t_pcb, next->t_pcb);
    /* back running in same thread */
}
```

OS/161 mips_switch

```c
mips_switch:
/*@ */
/*@ a0 contains a pointer to the old thread's strucpcb. */
/*@ a1 contains a pointer to the new thread's strucpcb. */
/*@ The only thing we touch in the pcb is the first word, which */
/*@ we save the stack pointer in. The other registers get saved */
/*@ on the stack, namely: */
/*@ a0->a8 */
/*@ sp, ra */
/*@ The order must match arch/mips/include/schframe.h. */
/*@ */
/*@ Allocate stack space for saving 11 registers. 11*4 = 44 */
/*@ add sp, sp, -44 */
```

OS/161 mips_switch

```c
/* Save the registers that the C */
/* procedure calling convention */
/* expects preserved */
```
Preemptive Multithreading

- Switch can be triggered by asynchronous external event
  - timer interrupt
- Asynch event saves current state
  - on current stack, if in kernel (nesting)
  - on kernel stack or in TCB if coming from user-level
- call thread_switch()
Fast thread management (creation, deletion, switching, synchronisation…)

- Blocking blocks all threads in a process
  - Syscalls
  - Page faults
- No thread-level parallelism on multiprocessor

Slow thread management (creation, deletion, switching, synchronisation…)
- System calls
- Blocking blocks only the appropriate thread in a process
- Thread-level parallelism on multiprocessor
Continuations (in Functional Languages)

- **Definition of a Continuation**
  - representation of an instance of a computation at a point in time

**call/cc in Scheme**
call/cc = call-with-current-continuation

- **A function**
  - takes a function \( f \) to call as an argument
  - calls that function with a reference to current continuation \( \text{cont} \) as an argument
  - when \( \text{cont} \) is later called, the continuation is restored.
  - The argument to \( \text{cont} \) is returned from to the caller of call/cc

**Simple Example**

```scheme
(define (f return)
  (return 2)
)
(display (f (lambda (x) x))); displays 3
(display (call-with-current-continuation f)); displays 2
```

**Another Simple Example**

```scheme
(define the-continuation #f)
(define (test)
  (let ((i 0)) ; call/cc calls its first function argument, passing
    ; a continuation variable representing this point in
    ; the program as the argument to that function.
    (lambda (k) (set! the-continuation k)))
    ; The next time the-continuation is called, we start here.
    (set! i (+ i 1))
    i)
)
```

**Yet Another Simple Example**

```scheme
;;; Return the first element in LST for which WANTED? returns a true
;;; value.
(define (search wanted? lst)
  (call/cc (lambda (return)
    (for-each (lambda (element)
      (if (wanted? element)
        (return element)))
      lst)
    #f)))
```

Source: http://community.schemewiki.org/?call-with-current-continuation
Coroutine Example

;; This starts a new thread running (proc).
(define (fork proc)
  (call/cc (lambda (k)
    (enqueue k)
    (proc)))))

;; This yields the processor to another thread, if there is one.
(define (yield)
  (call/cc
   (lambda (k)
    (enqueue k)
    ((dequeue))))))

Continuations

- A method to snapshot current state and return to the computation in the future
- In the general case, as many times as we like
- Variations and language environments (e.g. in C) result in less general continuations
  - e.g. one shot continuations

What should be a kernel’s execution model?

Note that the same question can be asked of applications

The two alternatives

No one correct answer

From the view of the designer there are two alternatives.

Single Kernel Stack

- Only one stack is used all the time to support all user threads.

Per-Thread Kernel Stack

- Every user thread has a kernel stack.

Per-Thread Kernel Stack

- A thread’s kernel state is implicitly encoded in the kernel activation stack
  - If the thread must block in-kernel, we can simply switch from the current stack, to another threads stack until thread is resumed
  - Resuming is simply switching back to the original stack
  - Preemption is easy
  - no conceptual difference between kernel mode and user mode

Single Kernel Stack

- How do we use a single kernel stack to support many threads?
  - Issue: How are system calls that block handled?

⇒ either continuations

Using Continuations to Implement Thread Management and Communication in Operating Systems. [Draves et al., 1991]

⇒ or stateless kernel (interrupt model)

- Interface and Execution Models in the Fluke Kernel. [Ford et al., 1999]
Continuations

- State required to resume a blocked thread is explicitly saved in a TCB
  - A function pointer
  - Variables
- Stack can be discarded and reused to support new thread
- Resuming involves discarding current stack, restoring the continuation, and continuing

```c
example(argc, argv2) {
  P1(argc, argv2);
  if (need_to_block) {
    save_arg_in_TCB;
    thread_block(example_continue);
  } else {
    P1();
    thread_syscall_return(SUCCESS);
  }
  example_continue();
  recover_arg2_from_TCB;
  P2(recovered argv2);
  thread_syscall_return(SUCCESS);
}
```

Stateless Kernel

- System calls cannot block within the kernel
  - If syscall must block (resource unavailable)
    - Modify user-state such that syscall is restarted when resources become available
    - Stack content is discarded (functions all return)
  - Preemption within kernel difficult to achieve.
    - Must (partially) roll syscall back to a restart point
  - Avoid page faults within kernel code
    - Syscall arguments in registers
      - Page fault during roll-back to restart (due to a page fault) is fatal.

IPC examples – Per thread stack

Send and Receive system call implemented by a non-blocking send part and a blocking receive part.

```c
msg_send_rcv(msg, option, send_size, rcv_size, ...)
  return rc;
```

IPC Examples – stateless kernel

Set user-level PC to restart msg_rcv only

Single Kernel Stack

- either continuations
  - complex to program
    - must be conservative in state saved (any state that might be needed)
    - Mach (Draves), L4Ka:Strawberry, NICTA Pistachio, OKL4
  - stateless kernel
    - no kernel threads, kernel not interruptible, difficult to program
    - request all potentially required resources prior to execution
    - blocking syscalls must always be restartable
    - Processor-provided stack management can get in the way
      - system calls need to be kept simple "atomic".
        - e.g. the fsl kernel from Utah
  - low cache footprint
    - always the same stack is used !
    - reduced memory footprint
Per-Thread Kernel Stack

- simple, flexible
  - kernel can always use threads, no special techniques
  - required for keeping state while interrupted / blocked
  - no conceptual difference between kernel mode and user mode
  - e.g. traditional L4, Linux, Windows, OS/161

- but larger cache footprint
- and larger memory consumption