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 require waiting for completion (e.g. disk read)
  - reacting to external triggers (e.g. interrupts)

Need a systematic approach to system structuring

Construction Approaches
Events
Coroutines
Threads
Continuations

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

What is ‘a’?

```c
int a; /* global */

int func()
{
    a = 1;
    if (a == 1) {
        a = 2;
    }
    return a;  // No concurrency issues within a handler
}
```
Event-based kernel on CPU with protection

Huh? How to support multiple processes?

Co-routines

Originally described in:

Analogous to a “subroutine” with extra entry and exit points.

Via yield()
- Supports long running subroutines
- An implement sync primitives that wait for a condition to be true

What is ‘a’?

```c
int a; /* global */

int func()
{
    a = 1;
    yield();
    if (a == 1) {
        a = 2;
    }
    return a;
}
```

What is ‘a’?

```c
int a; /* global */

int func()
{
    a = 1;
    if (a == 1) {
        yield();
        a = 2;
    }
    return a;
}

No concurrency issues/races as globals are exclusive between yields()
Co-routines Implementation strategy?

Co-routines

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-s8
 * gp, 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
```

/* Get the new stack pointer from the new pcb */
lw sp, 0(a1)
addi /* delay slot for load */

/* Now, restore the registers */
lw s0, 0(sp)
lw s1, 4(sp)
lw s2, 8(sp)
lw s3, 12(sp)
lw s4, 16(sp)
lw s5, 20(sp)
lw s6, 24(sp)
lw s7, 28(sp)
lw s8, 32(sp)
lw gp, 36(sp)
lw ra, 40(sp)
addi /* delay slot for load */

/* and return. */
jal /* in delay slot */

```

Yield
What is ‘a’?

```c
int a; /* global */

int func() {
    a = 1;
    func2();
    if (a == 1) {
        a = 2;
    }
    return a;
}
```

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
Conservatively assumes a multithreading model
- i.e. uses synchronisation to avoid races,
  - and makes no assumption about subroutine behaviour
  - Everything thing can potentially yield()

A Thread

<table>
<thead>
<tr>
<th>CPU</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>STACK</td>
<td>DATA</td>
</tr>
<tr>
<td>REGS</td>
<td>CODE</td>
</tr>
<tr>
<td>PCB</td>
<td>TCB</td>
</tr>
</tbody>
</table>

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 to 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
  • saving state of thread A
    – regs; sp; pc
  • restoring the state of thread B
    – sp; pc

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

OS/161 mips_switch

mips_switch:

#include arch/mips/include/switchframe.h.

/* Allocate stack space for saving 11 registers. 11*4 = 44 */
addi sp, sp, -44
/* Save the registers */
sw ra, 40(sp)
sw sp, 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)
/* Get the new stack pointer from the new pcb */
lw sp, 0(a1)
cmpeq a0 a1
/* delay slot for load */
/* Now, restore the registers */
lw s8, 0(sp)
lw s7, 4(sp)
lw s6, 8(sp)
lw s5, 12(sp)
lw s4, 16(sp)
lw s3, 20(sp)
lw s2, 24(sp)
lw s1, 28(sp)
lw s0, 32(sp)
ld sp, 36(sp)
ld ra, 40(sp)
/* delay slot for load */
/* and return. */
j ra
addi sp, sp, 44
/* in delay slot */
.end mips_switch

Approximate OS

mi_switch()

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

mips_switch(a,b)

mips_switch(b,a)

Thread Switch

Thread a

Thread b
Preemptive Multithreading

Switch can be triggered by asynchronous external event
- timer interrupt
- on current stack, if in kernel (reading)
- on kernel stack or in TCB if coming from user-level
   call thread_switch()
User-level Threads

- Fast thread management (creation, deletion, switching, synchronisation…)
- Blocking blocks all threads in a process
  - Syscalls
  - Page faults
- No thread-level parallelism on multiprocessor

Kernel-level Threads

- 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

\[
\text{call-with-current-continuation}(f)
\]

\[
(f \ x)
\]

\[
(x \ \text{return_arg})
\]

---

**Note**

For C-programmers, call/cc is effectively saving stack, and PC

---

**Simple Example**

\[
\text{(define } (f \ \text{arg})
\]

\[
(f \ (\lambda (x) x)); \text{ displays 3}
\]

\[
\text{(display } (\text{call-with-current-continuation } f))\;
\text{ displays 2}
\]

---

**Another Simple Example**

\[
\text{(define the-continuation #f)}
\]

\[
\text{(define } (test)
\]

\[
\text{let } (i 0)
\]

; call/cc calls its first function argument, passing
; a continuation variable representing this point in
; the program as the argument to that function.

; In this case, the function argument assigns that
; continuation to the variable the-continuation.

; \text{(call/cc (lambda (k) (set! the-continuation k)))}

; The next time the-continuation is called, we start here.
\[
(\text{set! i (+ i 1))}
\]

\[
(i)
\]

---

**Another Simple Example**

\[
> \text{(test)}
\]

\[
1
\]

1

\[
(\text{the-continuation})
\]

\[
2
\]

2

\[
(\text{the-continuation})
\]

\[
3
\]

3

\[
(\text{the-continuation})
\]

\[
4
\]

4
Yet Another Simple Example

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

Coroutine Example

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

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

Continuations

A method to snapshot current (stack) 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, setjmp()/longjump()

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

Processes Model

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 thread's stack until thread is resumed
• Resuming is simply switching back to the original stack
• Preemption is easy

example(xarg1, xarg2) {
  P1(xarg1, xarg2);  
  if (need_to_block) {
    thread_block();  
    P2(xarg2);  
  } else {
    P3();  
  }  
  /* return control to user */  
  return SUCCESS;  
}
Single Kernel Stack “Event” or “Interrupt” Model

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 (event model)
    – Interface and Execution Models in the Fluke Kernel. [Ford et al., 1999]
    – Also seL4

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

IPC implementation examples – Per thread stack

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

IPC Examples – stateless kernel

Set user-level PC to restart msg_rv only
RESCHEDULE changes curthread on exiting the kernel
Single Kernel Stack
per Processor, event model

either continuations
- complex to program
- must be conservative in state saved (any state that might be needed)
  - Mach (Draves), L4Ka:Strawberry, NICTA Pistachio, OKL4

or stateless kernel
- no kernel threads, kernel not interruptible, difficult to program
  - request all potentially required resources prior to execution
  - blocking syscalls must always be re-startable
  - Processor-provided stack management can get in the way
  - system calls need to be kept simple "atomic".
  - e.g. the fluke 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