2014 / 2015 Taste of Research Summer Scholarships Program

• ELIGIBILITY
  – You must be a high achieving 3rd year undergraduate student enrolled in a full time program (2nd year students may be considered under special circumstances)
  – You must be enrolled in a relevant program at UNSW or another Australian University
  – You must submit an online application by August 22nd
  – You may be a local or international student

• HOW TO APPLY
  – Apply online now at:
  – http://www.engineering.unsw.edu.au/taste-of-research-scholarships
  – For further information please visit the website.

• Closing Date
  – Friday August 22nd

• NICTA/SSRG has extra scholarships available
  – Choose one of topics, we can change later.
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
Construction Approach

- 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

Kernel-only Memory  User Memory

- Event Loop
- Event Handler 1
- Event Handler 2
- Event Handler 3
- Data
- Stack

Scheduling?

User Code

User Data

Stack

PC

SP

REGS

• Huh?
• How to support multiple processes?
Event-based kernel on CPU with protection

Kernel-only Memory
- Trap Dispatcher
- Event Handler 1
- Event Handler 2
- Timer Event (Scheduler)
- Data
- Current Thread
- Stack

User Memory
- PCB A
- PCB B
- PCB C
- User Code
- User Data
- Stack

CPU
- PC
- SP
- REGS

- User-level state in PCB
- Kernel starts on fresh stack on each trap
- No interrupts, no blocking in kernel mode
Co-routines

• A subroutine with extra entry and exit points
• Via yield()
  – supports long running subroutines
  – variations in precise semantics (yieldto, asymmetric and symmetric)
Co-routines

- yield() saves state of routine A and starts routine B
  - or resumes B’s state from its previous yield() point.
- No preemption
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?

- Usually implemented with a stack per routine
- Preserves current state of execution of the routine
Co-routines

- Routine A state currently loaded
- Routine B state stored on stack
- Routine switch from A → B
  - saving state of A a
    - regs, sp, pc
  - restoring the state of B
    - regs, sp, pc
A hypothetical yield()

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

/* and return. */
    j      ra
    addi   sp, sp, 44  /* in delay slot */
Routine A

yield(a, b) {
}

yield(a, b) {
}

Routine B

yield(b, a) {
}

yield(b, a) {
}
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
    - it can potentially yield()
int a; /* global */

int func() {
    int t;
    lock_aquire(lock)
    a = 1;
    func2();
    if (a == 1) {
        a = 2;
    }
    t = a;
    lock_release(lock);
    return t;
}
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 to 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
    - regs, sp, pc
- Note: registers and PC can be stored on the stack, and only SP stored in TCB
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 */
}

Note: global variable curthread
mips_switch:

/*
 * a0 contains a pointer to the old thread's struct pcb.
 * a1 contains a pointer to the new thread's struct pcb.
 *
 * 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:
 *
 * s0-s8
 * gp, ra
 *
 * The order must match arch/mips/include/switchframe.h.
 */

/* Allocate stack space for saving 11 registers. 11*4 = 44 */
addi sp, sp, -44
OS/161 mips\_switch

/* 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)
nop
/* 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)
nop
/* delay slot for load */

/* and return. */
ja  ra
addi   sp, sp, 44
/* in delay slot */
.end mips_switch
Thread a

mips_switch(a, b) {
}

mips_switch(a, b) {
}

mips_switch(b, a) {
}

Thread b

mips_switch(b, a) {
}

Thread Switch
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()
Threads on simple CPU

Memory

- Code
- Data
- Stack
- Stack
- Stack
- TCB A
- TCB B
- TCB C
- Scheduling & Switching
Threads on CPU with protection

Kernel-only Memory

- Code
- Data
- Stack
- TCB A
- TCB B
- TCB C

User Memory

- Scheduling & Switching
- Stack

• What is missing?

CPU

- PC
- SP
- REGS
Threads on CPU with protection

Kernel-only Memory   User Memory

- Code
- Data
- Stack
- TCB A
- TCB B
- TCB C

Scheduling & Switching

User Code
User Data
Stack

CPU

PC
SP
REGS

• What happens on kernel entry and exit?
Switching Address Spaces on Thread Switch = Processes

Kernel-only Memory

User Memory

CPU

Code

Data

Stack

TCB A

TCB B

TCB C

Scheduling & Switching

User Code

User Data

Stack

PC

SP

REGS

© Kevin Elphinstone
Switching Address Spaces on Thread Switch = Processes

Kernel-only Memory

User Memory

CPU

Code
Data
Stack
TCB A
TCB B
TCB C

Scheduling & Switching

User Code
User Data
Stack
PC
SP
REGS

User Code
User Data
Stack
What is this?

Kernel-only Memory       User Memory

- Code
- Data
- Stack
- TCB A
- TCB B
- TCB C

Scheduling & Switching

User Code
User Data
Stack
Stack
Stack

PC
SP
REGS

CPU
What is this?

Kernel-only Memory

User Memory

CPU

Code
Data
Stack
TCB A
TCB B
TCB C
Scheduling & Switching

Code
Data
Stack
Stack
TCB 1
TCB 2
TCB 3
Scheduling & Switching

PC
SP
REGS
User-level Threads

User Mode

Kernel Mode

Scheduler

Process A

Scheduler

Process B

Scheduler

Process C

Scheduler
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

User Mode

Kernel Mode

Scheduler
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
Simple Example

(define (f arg)
  (arg 2)
  3)

(display (f (lambda (x) x))); displays 3

(display (call-with-current-continuation f)); displays 2

Another Simple Example

(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.
    ;
    ; In this case, the function argument assigns that
    ; continuation to the variable the-continuation.
    ;
    (call/cc (lambda (k) (set! the-continuation k)))
    ;
    ; The next time the-continuation is called, we start here.
    (set! i (+ i 1))
    i))
Note

• For C-programmers, call/cc is effectively saving stack, and PC
Another Simple Example

> (test)
  1
  > (the-continuation)
  2
  > (the-continuation)
  3
  > ; stores the current continuation (which will print 4 next) away
  > (define another-continuation the-continuation)
  > (test); resets the-continuation
  1
  > (the-continuation)
  2
  > (another-continuation); uses the previously stored continuation
  4

Derived from http://en.wikipedia.org/wiki/Continuation

© Kevin Elphinstone
;;; 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)))

Derived from http://community.schemewiki.org/?call-with-current-continuation
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)))))

© Kevin Elphinstone
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, 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
  - no conceptual difference between kernel mode and user mode

```c
example(arg1, arg2) {
    P1(arg1, arg2);
    if (need_to_block) {
        thread_block();
        P2(arg2);
    } 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

```c
example(arg1, arg2) {
    P1(arg1, arg2);
    if (need_to_block) {
        save_arg_in_TCB;
        thread_block(example_continue);
        /* NOT REACHED */
    } else {
        P3();
    }
    thread_syscall_return(SUCCESS);
}

example_continue() {
    recover_arg2_from_TCB;
    P2(recovered arg2);
    thread_syscall_return(SUCCESS);
}
```
Stateless Kernel

- System calls can not 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 implementation examples
– Per thread stack

```c
msg_send_rcv(msg, option,
    send_size, rcv_size, ...) {

    rc = msg_send(msg, option,
      send_size, ...);

    if (rc != SUCCESS)
        return rc;
    rc = msg_rcv(msg, option, rcv_size, ...);
    return rc;
}
```

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

```c
msg_send_rcv(msg, option,
    send_size, rcv_size, ...) {
    rc = msg_send(msg, option,
        send_size, ...);
    if (rc != SUCCESS)
        return rc;
    cur_thread->continuation.msg = msg; cur_thread->continuation.option = option; cur_thread->continuation.rcv_size = rcv_size;
    ...
    rc = msg_rcv(msg, option, rcv_size, ...,
        msg_rcv_continue);  // The function to continue with if blocked
    return rc;
}
msg_rcv_continue() {
    msg = cur_thread->continuation.msg;
    option = cur_thread->continuation.option;
    rcv_size = cur_thread->continuation.rcv_size;
    ...
    rc = msg_rcv(msg, option, rcv_size, ...,
        msg_rcv_continue);
    return rc;
}
```

IPC Examples – stateless kernel

```c
msg_send_rcv(cur_thread) {
    rc = msg_send(cur_thread);
    if (rc != SUCCESS)
        return rc;

    rc = msg_rcv(cur_thread);
    if (rc == WOULD_BLOCK) {
        set_pc(cur_thread, msg_rcv_entry);
        return RESCHEDULE;
    }
    return rc;
}
```

Set user-level PC to restart `msg_rcv` only

RESCHEDULE changes `curthread` on exiting the kernel
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

- 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