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

Threads, Events, Continuations
- In the OS course you learned about threads and synchronisation
- Today:
  - Alternative execution models
  - More about threads

Sequential program
- Single flow of control
- Exclusive access to global state
- Execution state is maintained by the language runtime
  - local variables
  - stack
  - control-flow constructs

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

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
  - concurrency and synchronisation
  - scheduling
  - shared data

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

- No language support for maintaining control flow across event handlers:
  - No stack: what happens if a blocking operation occurs in a library function?
  - No local variables
  - No control flow constructs: what if a blocking operation occurs inside a loop?

Event-based design guidelines

- Events can be simpler and more efficient than threads.
- Use with caution:
  - Usually inappropriate for deep software stacks.
  - Refactoring is hard.
- Design control flow structure and communication protocols in advance

Co-routines

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

Event-based kernel on CPU with protection

- How to support multiple processes?
**Co-routines**

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

---

**Co-routine example**

```cpp
var q := new queue
coroutine produce
    loop
        while q is not full
            create some new items
            add the items to q
    yield to consume
end

coroutine consume
    loop
        while q is not empty
            remove some items from q
            use the items
    yield to produce
```


---

**Co-routines**

- Routine A state currently loaded
- Routine B state stored on stack
- Routine switch from A \(\rightarrow\) B
  - saving state of A
    - sp, pc
  - restoring the state of B
    - sp, pc

---

**A hypothetical yield()**

```cpp
/**
 * All contains a pointer to the previous routine's struct.
 * al contains a pointer to the new routine's struct.
 * The registers get saved on the stack, namely:
 *    sp
 *    s0-s7
 *    a0
 *    /* Allocate stack space for saving l1 registers. 11+4 = 44 */
 *    addi sp, sp, -44
 */
```

---
Save the registers that the 'C' procedure calling convention expects preserved

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

A Thread

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

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
    - spgs, sp, pc
  - restore the state of thread B
    - mips sp, pc
- Note: registers and PC can be stored on the stack, and only SP stored in TCB

OS/161 mips_switch

// Save the registers that the C procedure calling convention expects preserved

Approx OS

mi_switch();
{
    struct thread *cur, *next;
    next = scheduled[1];
    /* update curthread */
    cur = curthread;
    curthread = next;
    /* print the machine-dependent code that actually uses the 
    context switch. */
    mi_switch(kur=t_pb, knext=t_pb);
    /* Back running in same thread */
}

OS/161 mips_switch

// Get the new stack pointer from the new pb

mips_switch:
/*
 * sp contains a pointer to the old thread’s struct pb.
 * al contains a pointer to the new thread’s struct pb.
 */
/* The only thing we touch in the pb in the first word, which
 * we save the stack pointer in. The other registers get saved 
 * on the stack, namely:
 *  
 *  al
 *  sp
 *  gp, ra
 */
/* The nodes must match each/mips/include/switchframe.h. */
/* Allowing stack space for saving 31 registers. 11 * 4 = 44 */
/* Deal sp, sp, -44

OS/161 mips_switch

Thread Switch

Thread a

Thread b
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()

Switching Address Spaces on Thread Switch = Processes

Kernel-only Memory | User Memory

CPU

Threads on simple CPU

Memory

Kernel-only Memory | User Memory

CPU

Switching Address Spaces on Thread Switch = Processes

CPU

Threads on CPU with protection

Kernel-only Memory | User Memory

CPU

What is this?
What is this?

Kernel-only Memory  User Memory

Kernel-Level Threads

User Mode

Processor A

Processor B

Processor C

Scheduler

Kernel Mode

User 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

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

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 its own 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 stack when 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

“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 (interrupt model)
      - Interface and Execution Models in the Fluke Kernel. [Ford et al., 1999]

---

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

---

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

**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();
    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;
    }
}
```

**Single Kernel Stack**

- either continuations
  - complex to program
  - must be conservative in state saved (any state that might be needed)
  - Mach (Droese), L4K: Strawberry, NCTA Pistacho, OKLA, solL4
- 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 RISC 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

**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_pkt(cur_thread, msg_rcv_entry);
    }
    return RESCHEDULE;
}
```