Threads and Events

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

One Approach - Threads
- What are threads?
- How do we implement them?

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
  - saving state of thread a
  - reg, sp, etc
  - restoring the state of thread B
  - reg, sp, etc
- Note: registers and PC can be stored on the stack, and only SP stored in TCB
Approx thread switch

/* update curthread */
curthread = next;
/* back running in same thread */
Discussion: What operations are in a thread package?

- Create, delete? = non-protection
- Yield? = switching
- Run = scheduling
- Sync = scheduling

Threads on CPU with protection

• What is missing?

Switching Address Spaces on Thread Switch = Processes

What is this?
What is this?

Kernel-only Memory User Memory

CPU

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

Thread Alternative - 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.
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.

**Processes 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
- 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, argv) {
  P1(argc, argv);
  if (need_to_block) {
    save_arg2_in_TCB;
    thread_block(example_continue);
    /* NOT REACHED */
  } else {
    P1();
  }
  thread_syscall_return(SUCCESS);
}
```

 IPC 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(cur_thread);
  if (rc == WOULD_BLOCK) {
    set_args(cur_thread, ...);
    set_pc(cur_thread, msg_rcv_entry);
    return BLOCKED;
  } else {
    return rc;
  }
}
```

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 freed (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 - Continuations

```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(cur_thread);
  if (rc == WOULD_BLOCK) {
    set_args(cur_thread, ...);
    set_pc(cur_thread, msg_rcv_entry);
    return BLOCKED;
  } else {
    return rc;
  }
}
```

 IPC Examples – Stateless kernel

```c
msg_send_rcv(...) {
  rc = msg_send(dest);
  if (rc != SUCCESS) return rc;
  rc = msg_rcv(cur_thread);
  if (rc == WOULD_BLOCK) {
    set_args(cur_thread, ...);
    set_pc(cur_thread, msg_rcv_entry);
  } else {
    BLOCKED changes (away from) continuad on exiting the kernel
    return rc;
  }
}
```

Single Kernel Stack
- either continuations
  - complex to program
  - must be conservative in state saved (any state that might be needed)
  - Mach (Digital), L4Ka: Strawberry, NICTA Patches, OKL4
- or Stateless kernel
  - no kernel threads, kernel not interruptible, difficult to program
  - request all potentially required resources prior to execution
  - blocking syscall must always be re-restartable
  - Processor-provided stack management can get in the way
  - system calls need to be kept simple “atomic”, ex: the fluke kernel from Utah
- low cache footprint
  - always the same stack is used
  - reduced memory footprint

- Note: we don’t get here if msg_rcv blocked
- The function to continue with if blocked
- Block inside msg_rcv if no message available
- • • •
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