μ-Kernel Construction

Fundamental Abstractions

- Thread
- Address Space

- What is a thread?
- How to implement?

- What conclusions can we draw from our analysis with respect to μK construction?

A "thread of control" has

- register set (e.g. general registers, IP and SP)
- stack
- status (e.g.FLAGS, privilege, OS-specific states (prio, time...))
- address space
- unique id
- communication status

Construction Conclusions (1)

- Thread state must be saved / restored on thread switch.
- We need a thread control block (tcb) per thread.
- Tcbs must be kernel objects.
- Tcbs implement threads.
- We need to find any thread’s tcb starting from its uid (per processor)

Thread Switch A → B

Processor

IP
SP
FLAGS
tcb A

user mode A

Thread Switch A → B

Processor

IP
SP
FLAGS
tcb A

user mode A

kernel
In Summary:

- Thread A is running in user mode.
- Thread A has experiences an end-of-time-slice or is preempted by an interrupt.
- We enter kernel mode.
- The microkernel has to save the status of the thread A on A's TCB.
- The next step is to load the status of thread B from B's TCB.
- Leave kernel mode and thread B is running in user mode.
Construction conclusion

From the view of the designer there are two alternatives.

**Single Kernel Stack**

Only one stack is used all the time.

**Per-Thread Kernel Stack**

Every thread has a kernel stack.
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

- 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_context_in_TCB;
    thread_block(example_continue);
    /* NOT REACHED */
  } else {
    P1();
  }
  thread_syscall_return(SUCCESS);
}
```

```c
example_continue() {
  recover_context_from_TCB;
  P2(recovered argv);
  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
- 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

```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.

**Block inside msg_rcv if no message available**

### 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;
}
```

```c
msg_rcv_continue(cur_thread) {
  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;
  set_pc(cur_thread, msg_rcv_entry);
  rc = msg_rcv(cur_thread);
  if (rc != SUCCESS)
    return SUCCESS;
  return SUCCESS;
}
```

- 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
- 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. L4

  - but larger cache footprint

Conclusion:
We have to look for a solution that minimizes the kernel stack size!