MKC - Exercise 3

Benjamin Engel

2014-07-17
Overview

• Create new Execution Contexts (threads)
• Manage ECs in a (double linked ring) list
• Switch between them (cooperatively)

• User level: Requirements for threads: EIP+ESP

• Hands-on
  – User-level threading
  – 1\textsuperscript{st} “real” system call: create_ec
  – 2\textsuperscript{nd} system call: yield
  – 3\textsuperscript{rd} and 4\textsuperscript{th}: (blocking) send and recv
Scheduling

• Very very simple scheduler
  – No priorities, no time budgets
  – Cooperative multithreading
  – Single address space, uniprocessor

• Kernel: kern/include/ec.h
  – Registers (state)
  – Continuation (where to continue execution)
  – Management information (e.g. *prev, *next)

• User: user/src/user.cc
  – Code (instruction pointer)
  – Most likely a Stack (stack pointer)
What is a Thread/EC?
• Thread function: no parameter, nothing to return, but needs a stack
• Where to get the new stack from? malloc() → not available (so far)
• Put it statically in data segment or on local stack of the currently running thread:
  \texttt{char new\_stack[64];}
• Stack grows downwards, thus ESP should point to the end: \texttt{new\_stack + sizeof(new\_stack)}
Task 0: Minimal Thread User Code

- Write a new thread function in user/src/user.cc
  - Simple function doing nothing but spinning
  - Later it shall call `sys_yield()`, thus switching to the next thread

- New bindings for to-be-written syscalls:
  - `sys_create_ec` (2 (3) arguments):
    - Creates a shining new Execution Context
    - EIP of new EC (thread function's address)
    - ESP to be used – we need a user stack per EC
    - Capability of this new EC (unsigned integer)
  - `sys_yield` (no argument)
    - Simply switches to the next thread
Task 1: sys_create_ec

- Organize ECs in a ring list
  - add `prev` and `next` pointer (kern/include/ec.h)
  - Private `enqueue()` function, adding `this` to the tail of the list (kern/src/ec.cc)
  - Special case when creating very first EC, `Ec::current` is not yet set, watch out!

- Add a new system call
  - Two parameters (instruction and stack pointer)
  - `Ec::sys_regs()` and kern/include/regs.h
  - Create new EC, add it to the list, and sysexit
  - Verbose printf, newly created EC, its EIP/ESP, maybe even the whole list of ECs
Send/Recv

- Name receiver → caps
  - 1 page with EC pointers: static Ec** caps;
  - See kernel/include/kalloc.h

- Block sender/receiver until rendezvous → need dequeue() to remove EC from ready list
  - What happens if we dequeue the last EC?
  - No EC left to schedule → panic() or idle EC

- Ec::cont: the function to be called then this EC is scheduled, mostly ret_user_sysexit
- When receiving: set it to nullptr
Task 2: sys\_yield

- Switch from currently running EC (\texttt{Ec::current}) to next one (\texttt{current->next})
  - Every EC has a continuation – the function to execute whenever becoming ready (again)
  - The currently running thread shall continue with \texttt{ret\_user\_sysexit}, thus set \texttt{cont} accordingly
  - Switch to \texttt{current->next} via \texttt{make\_current()}

- Create more threads in user application, printf whenever they yield: \texttt{EC:%p} $\rightarrow$ \texttt{EC:%p}