Project 1, Processes, and Threads

CS 4411
Spring 2020
Outline for Today

• Intro to EGOS and GitHub
• Address Space Layout
  • Processes vs. Threads
• Context Switching
  • Kernel vs. User-Level Threads
• Project 1 Overview
EGOS

- Microkernel OS running on a virtual machine
- Earth simulates a machine within a single process of your host OS
- Grass implements an OS kernel running on this VM
EGOS Demo
GitHub Logistics

• EGOS code will be distributed via a GitHub repository
• https://github.coecis.cornell.edu/cs4411-2020sp/egos
• To access this repository, you will need to join the cs4411-2020sp organization – fill out the Google form
• Suggestion: Fork the EGOS repo and share it with your partner
• Reminder: All EGOS repos must remain private, not public
• Question: Would you like a lecture on how to use Git?
Outline for Today

• Intro to EGOS and GitHub
• **Address Space Layout**
  • Processes vs. Threads
• Context Switching
  • Kernel vs. User-Level Threads
• Project 1 Overview
Memory Layout

• Two segments of memory: Kernel Space and User Space
• User processes cannot access Kernel Space memory
• User processes cannot access any other process’s memory
Within a Process

Kernel space reserved in every process

Interrupt stack

Unmapped memory

Virtual addresses

Physical addresses
Concurrent Execution

- What if our program needs to do 2 things at once?
  - Listen for user input, and also spell-check file
- Do we need 2 entire processes?
- What is the difference between these 2 processes?
Memory Layout with Threads

Interrupt stack

Thread 1 Registers

Thread 2 Registers

Virtual addresses

Kernel

Thread 1 Stack

Thread 2 Stack

Heap

Data

Code

Emacs

Mail

Shell

Kernel

"Kernel Space"

"User Space"
Outline for Today

• Intro to EGOS and GitHub
• Address Space Layout
  • Processes vs. Threads
• Context Switching
  • Kernel vs. User-Level Threads
• Project 1 Overview
What is Context?

• To switch from Thread 1 to Thread 2, what needs to be saved?
  • Stack Pointer
  • Program Counter
  • All other CPU registers
Where to Save Context?

- Where should Thread 1’s context go?
  - Ordinary registers: On Thread 1’s stack
  - SP and PC: In a TCB for Thread 1
  - Where does TCB go? Depends on type of threads

![Diagram showing memory layout with Kernel, Thread 1 Stack, Thread 2 Stack, Heap, Data, and Code sections]
Kernel vs. User-level Threads

**Kernel-level Threads**
- Kernel keeps track of TCBs for threads in all processes
- Creating and joining require system calls
- Scheduled by kernel, can be pre-empted
- pthreads library for C

**User-level Threads**
- Process keeps track of TCBs for its own threads
- Scheduled by process that created them
- No system calls required
- Cannot be pre-empted – user-level process can’t pre-empt itself
ULTs and Context Switching

- When does a user-level thread context switch to another user-level thread?
- Thread yields explicitly
- Thread blocks on a synchronization function
- Thread exits
User-Level Context Switch

```c
void ctx_switch(address_t* old_sp, address_t new_sp);
```

ctx_switch:
```
push %rbp
push %rbx
push %r15
push %r14
push %r13
push %r12
push %r11
push %r10
push %r9
push %r8
mov %rsp, (%rdi)
mov %rsi, %rsp
pop %r8
pop %r9
pop %r10
pop %r11
pop %r12
pop %r13
pop %r14
pop %r15
pop %rbx
pop %rbp
ret
```

Save current thread’s registers onto its stack
Copy current thread’s stack pointer to arg 1
Overwrite stack pointer with arg 2 (thread 2’s SP)
Pop next thread’s registers off its stack
Return to next thread at instruction where it called ctx_switch
User-Level Context Switch

- Why didn’t we save or restore the PC?

- Threads only switch by calling this function – no pre-emption

- Function call instructions already save/restore PC

```assembly
ctx_switch:
push %rbp
push %rbx
push %r15
push %r14
push %r13
push %r12
push %r11
push %r10
push %r9
push %r8
mov %rsp, (%rdi)
mov %rsi, %rsp
pop %r8
pop %r9
pop %r10
pop %r11
pop %r12
pop %r13
pop %r14
pop %r15
pop %rbx
pop %rbp
ret
```
Outline for Today

• Intro to EGOS and GitHub
• Address Space Layout
  • Processes vs. Threads
• Context Switching
  • Kernel vs. User-Level Threads
• Project 1 Overview
Project 1 Overview

• Part 1: Implement a user-level threading library
• Part 2: Implement semaphores for your user-level threads
• Write test cases for both parts
• You will need to use the context-switch code provided with EGOS
  • src/h/context.h
  • src/lib/asm_<platform>_<architecture>.s
• Your code will be an application that gets bundled with EGOS when you compile and start it (with make run)
void thread_init();
• Initializes the threading library, allowing us to create threads.

void thread_create(void (*f)(void* arg), void* arg, unsigned int stack_size);
• Creates a new thread that will run function \( f \), a void function with one argument. \textit{thread_create}'s argument \texttt{arg} will be passed to \texttt{f}.

void thread_yield();
• Causes the current thread to give up the CPU and allow another thread to run.

void thread_exit();
• Causes the current thread to terminate permanently.
Context Switch Interface

```c
void ctx_switch(address_t* old_sp, address_t new_sp);
```

- As discussed earlier: Saves current thread’s SP in location pointed to by `old_sp`, then switches to thread whose SP is `new_sp`

```c
void ctx_start(address_t* old_sp, address_t new_sp);
```

- Saves current thread’s SP in `*old_sp`, sets SP to `new_sp`, then jumps to a function named `ctx_entry()` whose job is to start a new thread
- You must write `ctx_entry()` as part of your threading library
Testing Your Code

• Technically this is a library for EGOS, but...
• ... since it makes no system calls, you can run it in Linux too
• Only platform-dependent code is the assembly that implements `ctx_switch()` and `ctx_start()`
• If you copy `src/h/context.h` and `src/lib/asm_*_.s` to a new project directory, you can build and run your code as a Linux or Mac executable
• This makes it easier to debug
Next Week?