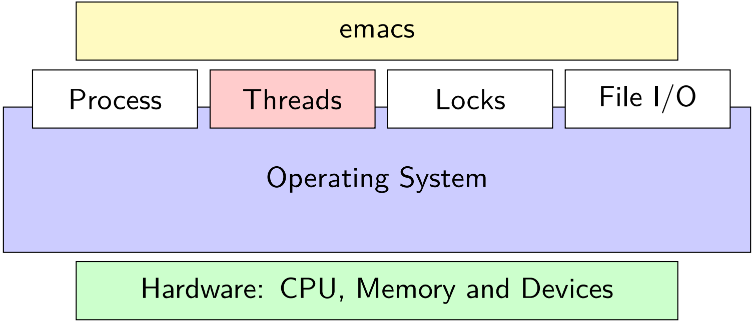
CS350: Operating Systems
Lecture 3: Threads

Ali Mashtizadeh

University of Waterloo

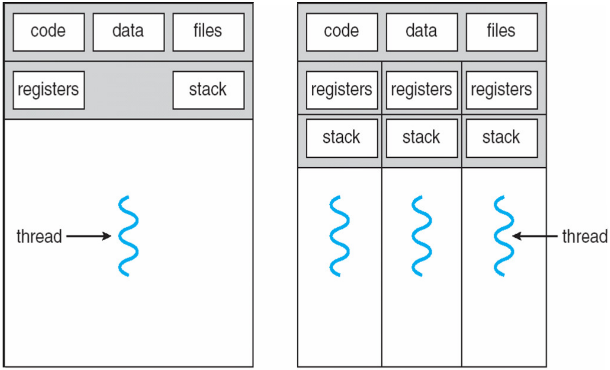
Today: Threads



Outline

1. Threads
2. Case Study: Go Language and Runtime
3. How to implement threads in COS

Threads



single-threaded process multithreaded process

- A thread is a schedulable execution context
  - Program counter, registers, stack (local variables) . . .
- Multi-threaded programs share the address space (global variables, heap, . . .)

Why threads?

- Most popular abstraction for concurrency
  - Lighter-weight abstraction than processes
  - All threads in one process share memory, file descriptors, etc.
- Allows one process to use multiple CPUs or cores
- Allows program to overlap I/O and computation
  - Same benefit as OS running emacs & gcc simultaneously
  - E.g., threaded web server services clients simultaneously:

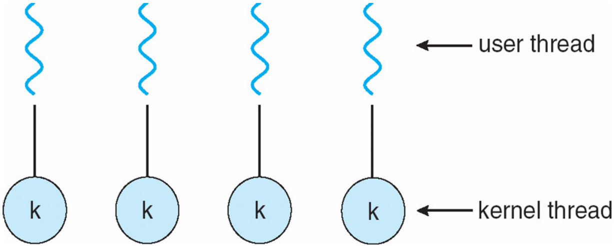
```
for (;;) {
  fd = accept_client ();
  thread_create (service_client, &fd);
}
```

- Most kernels have threads, too
  - Typically at least one kernel thread for every process

POSIX thread API

- `int pthread_create(pthread_t *thr, pthread_attr_t *attr, void *(*fn)(void *), void *arg);`
  - Create a new thread identified by `thr` with optional attributes, run `fn` with `arg`
- `void pthread_exit(void *return_value);`
  - Destroy current thread and return a pointer
- `int pthread_join(pthread_t thread, void **return_value);`
  - Wait for thread `thread` to exit and receive the return value
- `void pthread_yield();`
  - Tell the OS scheduler to run another thread or process
- Plus lots of support for synchronization (next Lecture and see [Birell])

Kernel threads

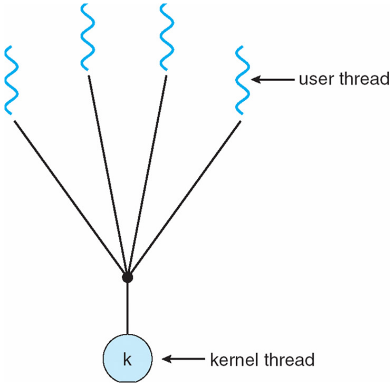


- Can implement `pthread_create` as a system call
- To add `pthread_create` to an OS:
  - Start with process abstraction in kernel
  - `pthread_create` like process creation with features stripped out
    - Keep same address space, file table, etc., in new process
    - `rfork`/`clone` syscalls actually allow individual control
- Faster than a process, but still very heavy weight

Limitations of kernel-level threads

- Every thread operation must go through kernel
  - create, exit, join, synchronize, or switch for any reason
  - Syscall takes 100 cycles, function call 2 cycles
  - Result: threads $10\times$–$30\times$ slower when implemented in kernel
  - Worse today because of SPECTRE/Meltdown mitigations
- One-size fits all thread implementation
  - Kernel threads must please all people
  - Maybe pay for fancy features (priority, etc.) you don't need
- General heavy-weight memory requirements
  - E.g., requires a fixed-size stack within kernel
  - Other data structures designed for heavier-weight processes

User threads



- An alternative: implement in user-level library
  - One kernel thread per process
  - `pthread_create`, `pthread_exit`, etc., just library functions

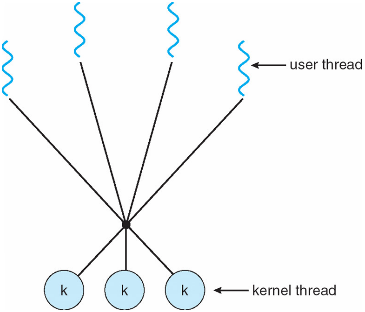
Implementing user-level threads

- Allocate a new stack for each `pthread_create`
- Keep a queue of runnable threads
- Replace blocking system calls (`read`/`write`/etc.)
  - If operation would block, switch and run different thread
- Schedule periodic timer signal (`setitimer`)
  - Switch to another thread on timer signals (preemption)
- Multi-threaded web server example
  - Thread calls `read` to get data from remote web browser
  - "Fake" `read` *function* makes `read` *syscall* in non-blocking mode
  - No data? schedule another thread
  - On timer or when idle check which connections have new data

Limitations of user-level threads

- Can't take advantage of multiple CPUs or cores
- A blocking system call blocks all threads
  - Can replace `read` to handle network connections
  - But usually OSes don't let you do this for disk
  - So one uncached disk read blocks all threads
- A page fault blocks all threads
- Possible deadlock if one thread blocks on another
  - May block entire process and make no progress
  - [More on deadlock in future lectures.]

User threads on kernel threads



- User threads implemented on kernel threads
  - Multiple kernel-level threads per process
  - `thread_create`, `thread_exit` still library functions as before
- Sometimes called $n : m$ threading
  - Have n user threads per m kernel threads
    (Simple user-level threads are $n : 1$, kernel threads $1 : 1$)

Limitations of $n : m$ threading

- Many of same problems as $n : 1$ threads
  - Blocked threads, deadlock, ...
- Hard to keep same # kthreads as available CPUs
  - Kernel knows how many CPUs available
  - Kernel knows which kernel-level threads are blocked
  - Tries to hide these things from applications for transparency
  - User-level thread scheduler might think a thread is running while underlying kernel thread is blocked
- Kernel doesn't know relative importance of threads
  - Might preempt kthread in which library holds important lock

Lessons

- Threads best implemented as a library
  - But kernel threads not best interface on which to do this
- Better kernel interfaces have been suggested
  - See Scheduler Activations [Anderson et al.]
  - Maybe too complex to implement on existing OSes (some have added then removed such features, now Windows is trying it)
- Today shouldn't dissuade you from using threads
  - Standard user or kernel threads are fine for most purposes
  - Use kernel threads if I/O concurrency main goal
  - Use $n : m$ threads for highly concurrent (e.g,. scientific applications) with many thread switches
- . . . though concurrency/synchronization lectures may
  - Concurrency greatly increases the complexity of a program!
  - Leads to all kinds of nasty race conditions

Outline

1. Threads
2. Case Study: Go Language and Runtime
3. How to implement threads in COS

Go Routines

- Go routines are very light-weight
  - Running 100k go routines is practical
  - Custom compiler enables stack segmentation, preemption, and garbage collection
  - Runs on segmented stack – stack allocated on demand to avoid memory use
  - OS thread typically allocate 2 MiB fixed stacks
- Go routines on top of Kernel threads (n:m Model)
  - Multi-core scalability and efficient user-level threads
  - One pthread (kernel-level thread) per CPU core
  - Supports many user-level threads as you like

Go Routine Continued

- Each kernel-level thread finds and runs a go routine (user-level thread)
- Every logical core is owned by a kernel thread when running
- Convert blocking system calls (when possible):
  - Converted to non-blocking by in the runtime yielding the CPU to another core
  - Cores poll using kernel event API `poll`, `epoll`, or `kqueue`
- Blocking system calls:
  - Release the "CPU" to another kernel-level thread before the call
  - Let the kernel thread sleep
  - Regain the "CPU" thread when done

Go Channels

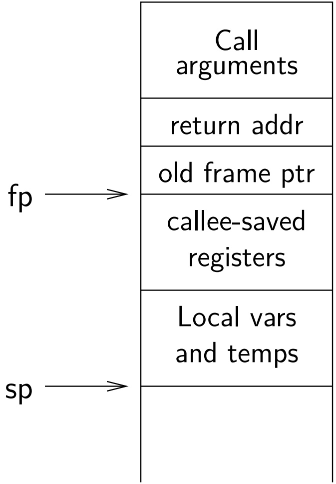
- Go routine communicate and synchronize through *channels*

```go
func worker(done chan bool) {
    // Notify the main routine
    done <- true
}

func main() {
    // Create a channel to notify us
    done := make(chan bool, 1)

    // Create go routine
    go worker(done)

    // Block until we receive a message
    <-done
}
```

Outline

Background: AMD64/x86-64 calling conventions

- Registers divided into 2 groups
  - Functions free to clobber *caller-saved* regs (`%r10`, `%r11`, arguments and return registers)
  - But must restore *callee-saved* ones to original value upon return (`%rbx`, `%r12`–`%r15`)
- `%rsp` register always base of stack
  - Frame pointer (or base pointer) (`%rbp`) is old `%rsp`
- Local variables stored in registers and on stack
- Function arguments go in caller-saved regs and on stack
  - First six arguments in `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`
  - Remaining arguments on stack
- Return value `%rax` and `%rdx`



Background: procedure calls

save active caller registers

call foo → saves used callee registers

...do stuff...

restores callee registers

jumps back to pc

restore caller regs

- Some state saved on stack
  - Return address, caller-saved registers
- Some state not saved
  - Callee-saved regs, global variables, stack pointer

Threads vs. procedures

- Threads may resume out of order:
  - Cannot use LIFO stack to save state
  - General solution: one stack per thread
- Threads switch less often:
  - Don't partition registers (why?)
- Threads can be involuntarily interrupted:
  - Synchronous: procedure call can use compiler to save state
  - Asynchronous: thread switch code saves all registers
- More than one than one thread can run at a time:
  - Procedure call scheduling obvious: Run called procedure
  - Thread scheduling: What to run next and on which CPU?

User-level Threading

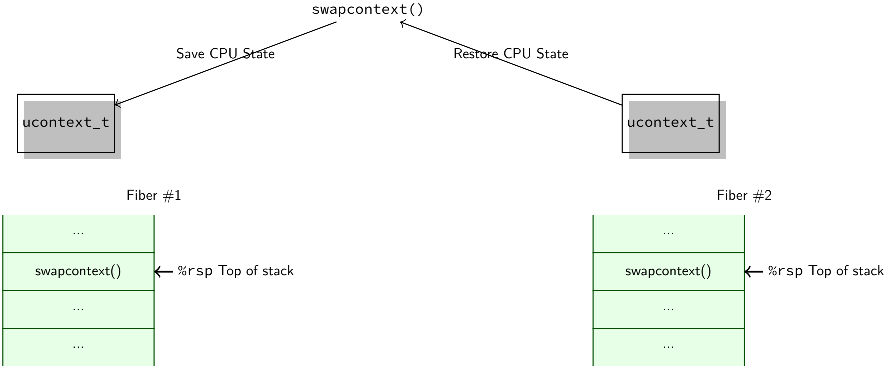
- POSIX even included some helper functions to help you build user-level threads.
- `void makecontext(ucontext_t *ucp, void (*func)(void),`
  `int argc, ...);`
  - Modify a previously created context with a function pointer and args
  - Use `getcontext` to create a valid context, then allocate a stack
- `int swapcontext(ucontext_t *old, ucontext_t *new);`
  - Save the current thread into `old`, restore the `new` thread
  - Saves and restores all CPU registers including the stack pointer
- `int getcontext(ucontext_t *ucp);`
  - Get the current thread context and save it in `ucp`

User-level Threading: Creating a Fiber

```
ucontext_t ctx;

// Use the current thread to initialize ctx to a valid state
// Why? Some CPU registers (e.g., rflags) require specific values
getcontext(&ctx);

// Allocate a stack
ctx.uc_stack.ss_sp = malloc(SIGSTKSZ);
ctx.uc_stack.ss_size = SIGSTKSZ;

// uc_link says what to do when done, NULL terminates the program.
ctx.uc_link = NULL;

// Initialize the context pointing to the function foo
makecontext(&ctx, (void (*)())foo, 0);
```

User-level Threading: Context Switching

- `swapcontext`: Saves and Restores the CPU state & swaps stacks
- `ucontext_t`: Contains all CPU state



COS: Thread Details

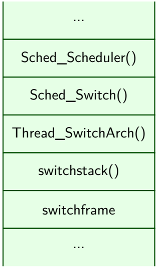
- Supports both kernel and user threads
- Basic Pthreads support see lib/libc/posix/pthread.c
- Mechanically similar to the fiber example

```
/* Create a kernel thread associated with the process */
Thread *Thread_KThreadCreate(void (*f)(void *), void *arg);

/* Create a userspace thread (and associated kernel thread */
Thread *Thread_UThreadCreate(Thread *oldThr,
                             uint64_t rip,
                             uint64_t arg);
```

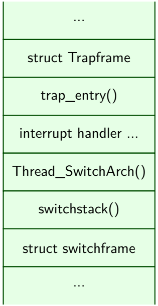
COS: Switching Threads

- All thread switches go through `Sched_Scheduler()` and `Sched_Switch()`
- `Sched_Switch()` calls `Thread_SwitchArch()` that runs `switchstack`
- `switchstack` switches from one stack to other while saving and restoring registers

General (from Kernel)



Hardware Interrupt (typically Timer)



COS: switchstack – switch kernel threads

From COS kern/amd64/switch.S

```
# switch(uint64_t *oldsp, uint64_t newsp)
# %rdi: oldsp
# %rsi: newsp
FUNC_BEGIN(switchstack)
    # Save callee saved registers of old thread
    pushq %rbp
    pushq %rdi
    pushq %rbx
    pushq %r12
    pushq %r13
    pushq %r14
    pushq %r15

    # Switch stack from old to new thread
    movq  %rsp, (%rdi)
    movq  %rsi, %rsp
```

COS: switchstack – switch kernel threads (con’t)

```

    # Restore callee saved registers of new thread
    popq  %r15
    popq  %r14
    popq  %r13
    popq  %r12
    popq  %rbx
    popq  %rdi
    popq  %rbp
    ret
FUNC_END(switchstack)
```