

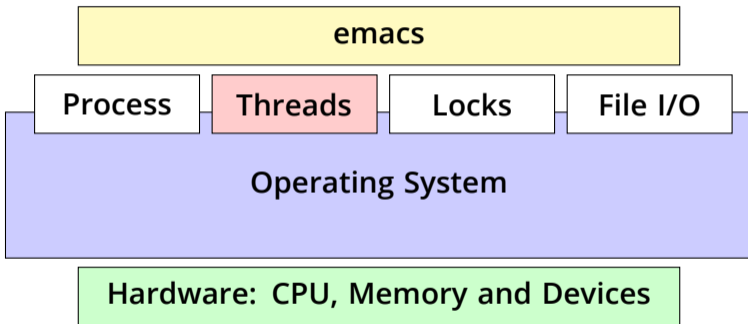
CS350: Operating Systems

Lecture 3: Threads

Ali Mashtizadeh

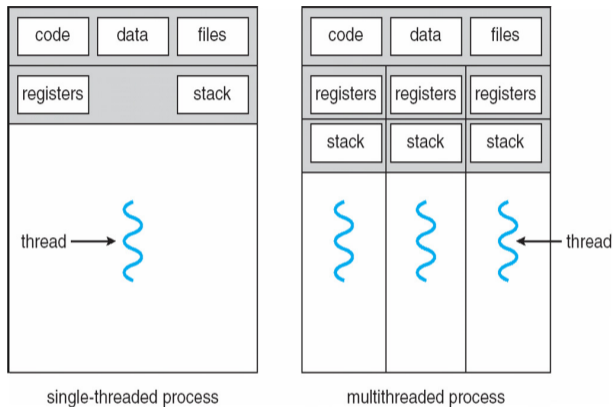
University of Waterloo

Today: Threads



- ① **Threads**
- ② Case Study: Go Language and Runtime
- ③ How to implement threads in OS/161

Threads



- 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?

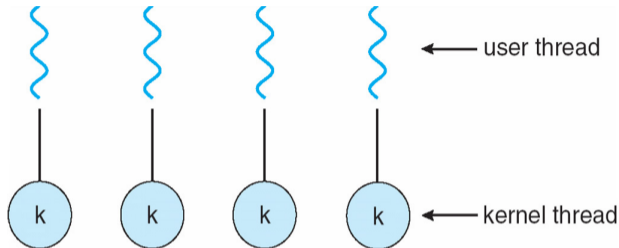
- Most popular abstraction for concurrency
 - ▶ Lighter-weight abstraction than processes
 - ▶ All threads in one process share memory, file descriptors, etc.
- Allows one process to use multiple CPUs or cores
- Allows program to overlap I/O and computation
 - ▶ Same benefit as OS running emacs & gcc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

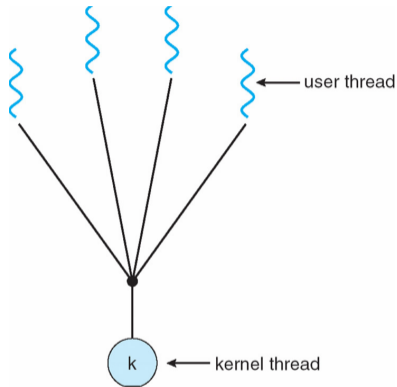


- Can implement `pthread_create` as a system call
- To add `pthread_create` to an OS:
 - ▶ Start with process abstraction in kernel
 - ▶ `pthread_create` like process creation with features stripped out
 - ▷ Keep same address space, file table, etc., in new process
 - ▷ `rfork/clone` syscalls actually 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

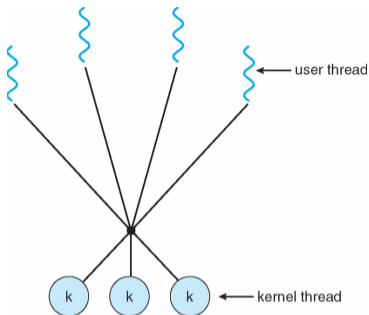
Implementing user-level threads

- 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

- ① Threads
- ② Case Study: Go Language and Runtime
- ③ How to implement threads in OS/161

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

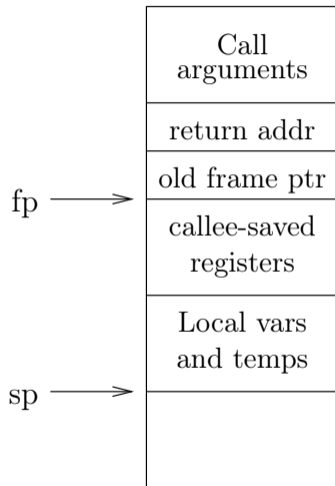
- Go routine communicate and synchronize through *channels*

```
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  
}
```

- ① Threads
- ② Case Study: Go Language and Runtime
- ③ How to implement threads in OS/161

Background: MIPS calling conventions

- Registers divided into 2 groups
 - ▶ Functions free to clobber *caller-saved* regs (%t0-%t9 on MIPS)
 - ▶ But must restore *callee-saved* ones to original value upon return (%s0-%s7, %fp)
- %sp register always base of stack
 - ▶ Frame pointer (%fp) is old %sp
- Local variables stored in registers and on stack
- Function arguments go in caller-saved regs and on stack
 - ▶ First four arguments in %a0-%a3
 - ▶ Remaining arguments on stack
- Return value %v0 and %v1



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?

OS/161 Kernel Threads

- OS/161 supports fork, exec, exit, and wait
 - ▶ You will implement these functions in Assignments 2a/2b
- One thread per process

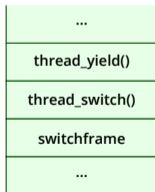
```
int thread_fork(const char *name,  
               struct proc *proc,  
               void (*entrypoint)(void *data1, unsigned long data2),  
               void *data1, unsigned long data2);
```

- OS/161 supports kernel threads (no user-level threading)
- Create a kernel thread with: thread_fork()
- Bad nameing: Not fork() this is actually pthread_create!

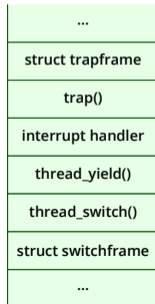
Switching Threads

- All thread switches go through `thread_yield()` and `thread_switch()`
- `thread_switch()` calls `switchframe_switch` **generates** `switchframe`
- `switchframe_switch` switches from one stack to other

General (from Kernel)



Hardware Interrupt (typically Timer)



OS/161 switchframe_switch – save old thread

From OS/161 kern/arch/mips/thread/switch.S

```
1  switchframe_switch:
2  /* a0: switchframe pointer to old thread */
3  /* a1: switchframe pointer to new thread */
4  /* Allocate space for saving 10 registers. 10*4 = 40 */
5  addi sp, sp, -40
6
7  sw ra, 36(sp) /* Save callee save registers */
8  sw gp, 32(sp) /* Caller saved registers saved by thread_switch() */
9  sw s8, 28(sp)
10 sw s6, 24(sp)
11 sw s5, 20(sp)
12 sw s4, 16(sp)
13 sw s3, 12(sp)
14 sw s2, 8(sp)
15 sw s1, 4(sp)
16 sw s0, 0(sp)
17
18 /* Store the old stack pointer in the old thread */
19 sw sp, 0(a0)
```

OS/161 switchframe_switch – restore new thread

```
1  /* Get the new stack pointer from the new thread */
2  lw sp, 0(a1)
3  nop /* Delay slot for load */
4
5  lw s0, 0(sp) /* Now, restore callee saved registers */
6  lw s1, 4(sp) /* Caller saved registers restored by thread_switch() */
7  lw s2, 8(sp)
8  lw s3, 12(sp)
9  lw s4, 16(sp)
10 lw s5, 20(sp)
11 lw s6, 24(sp)
12 lw s8, 28(sp)
13 lw gp, 32(sp)
14 lw ra, 36(sp)
15 nop /* Delay slot for load */
16
17 j ra /* jump register to return address. */
18 addi sp, sp, 40 /* Fix sp in delay slot for j */
```