## What is a Thread?

- Similar to a process
- Separate “thread of execution”
- Each thread has a separate stack, program counter, and run state

## Differences Between Processes and Threads

- Threads share the same address space
- Threads always have the same security context (i.e., UID)
- Threads share file state.
  - A child process will inherit open files from the parent, but won’t see newly-opened files. A child thread will see them.
- Non-preemptive scheduling

## Non-preemptive Scheduling

- There is no timer to make a thread yield the CPU
- Threads must voluntarily yield control to let another thread run
- Thread history isn’t taken into account by the scheduler
- Threads are *co-operative*, not competitive

## Why Threads?

- Perform operations in parallel on the same data
- Avoid complex explicit scheduling by applications
- Often more efficient than separate processes

## Example: Web Browser

- Download page in one thread
- Actually, use a separate thread for each separately-downloaded image
- Let another thread respond to mouse clicks
- Maybe have a separate thread for each tab or page image

## More Examples

- Web server
- Word processor
- Most graphical applications
- Anything where separate execution patterns need to share lots of data

## Implementing Threads

- Three basic strategies
- User-level: possible on most operating systems, even ancient ones
- Kernel-level: looks almost like a process (i.e., Linux, Solaris)
- Scheduler activations (Digital Tru Unix 64, NetBSD, some Mach)

## User-Level Threads

- Thread creation, destruction, and scheduling done at user level
- To create a thread, must allocate storage for stack, program counter, registers, state, etc, in a *thread table*
- When a thread yields the CPU or blocks, the thread library saves everything in the thread table
- The thread scheduler finds the highest-priority thread that's ready to run
- Its registers, program counter, etc., are restored from *its* thread table

## Efficiency Gains

- No system calls needed
- No context switch
- No copying data safely across a protection boundary
- No need to flush the hardware memory cache

## Disadvantages of User-Level Threads

- Blocking system calls
- Page faults
- Inability to benefit from multiprocessor CPUs
- Kernel can’t automatically grow per-thread stacks

## Growing the Stack

- The system allocates a certain amount of memory for the initial piece of the stack
- References to other potential parts of the stack cause a trap
- If this trap looks like an attempt to use an unallocated part of the stack, the kernel allocates more memory and extends the stack
- This only works if the kernel knows where the stack is

## Blocking System Calls

- What if a thread issues, say, a **`read()`** call and there's no data available?
- The kernel will make the whole *process* block because it's unaware of the thread structure

## Page Faults

- Part of the virtual memory system; won’t say much now
- Briefly, though — a trap occurs because referenced memory isn’t available
- No ability to suspend just one thread

## Wrapper Functions

- The thread library can contain alternate versions of system call functions
- These versions check for blocking *before* calling the kernel
- Example: do a non-blocking **`select()`** or **`poll()`** before a **`read()`**
- If the call would block, fire up another thread instead
- If no threads are ready, do a blocking **`select()`** or **`poll()`** and let the kernel run a different process

## Kernel Threads

- Similar thread table, but in the kernel

☞ Often integrated with process table

- No problem with blocking system calls — ordinary process scheduling does the right thing
- Handles multi-CPU case easily
- But — thread-switching is expensive, since it needs a context switch
- Still cheaper than process switches, since you can use the same virtual memory map

## Other Schemes

- Hybrid: some kernel threads, with each kernel thread supporting several user-level threads
- Scheduler activations
- Pop-up threads

## Scheduler Activations

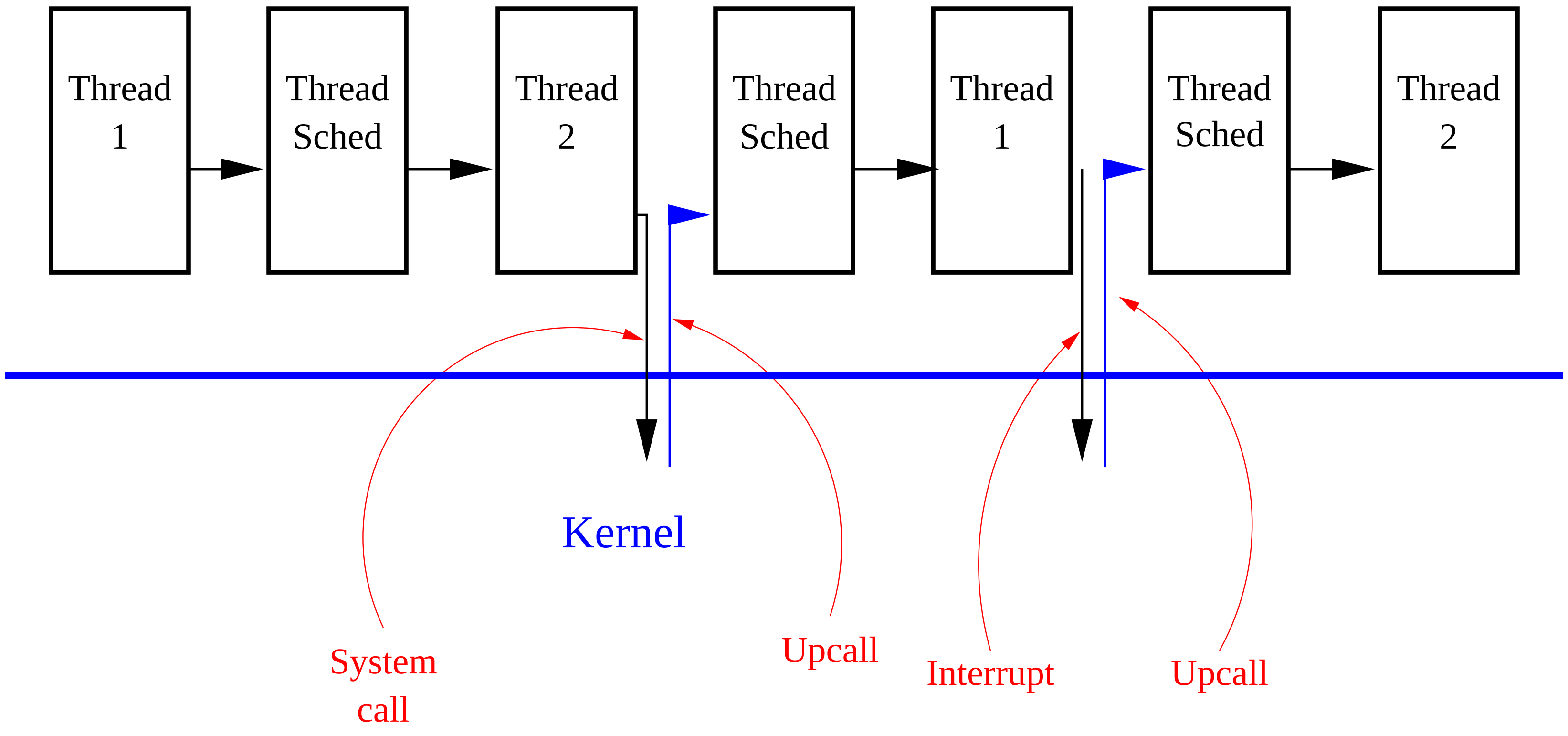
- Get the best of both worlds — the efficiency of user-level threads and the non-blocking ability of kernel threads
- Relies on *upcalls*

## What’s an Upcall?

- Normally, user programs call functions in the kernel
- Sometimes, though, the kernel calls a user-level process to report an event
- Sometimes considered unclean — violates usual layering

## Scheduler Activations

- Thread creation and scheduling is done at user level
- When a system call from a thread blocks, the kernel does an upcall to the thread manager
- The thread manager marks that thread as blocked, and starts running another thread
- When a kernel interrupt occurs that's relevant to the thread, another upcall is done to unblock it


Thread 1
Thread Sched
Thread 2
Thread Sched
Thread 1
Thread Sched
Thread 2
Kernel
System call
Upcall
Interrupt
Upcall

## Pop-Up Threads

- When a message arrives, the kernel creates a new thread
- Sometimes, the thread can run entirely in the kernel
- Messy and delicate — what process should own the thread, what sorts of events result in thread creation, etc.

## Using Threads

- Library routines may be used by more than one thread simultaneously
- Not all routines are prepared for this
- Static variables can be overwritten by another thread
- Must use *reentrant* routines

## What is a Reentrant Routine?

- A routine that can be invoked more than once simultaneously
- Can only use local variables or dynamically allocated variables
- Must not use static variables
- Many C library routines do use static buffers!

## Normal and Reentrant Versions

| | |
|---|---|
| **Normal** | struct tm *localtime(const time_t *timep); |
| **Reentrant** | struct tm *localtime_r(const time_t *timep, struct tm *result); |

The normal version stores store the result in a static buffer and passes a pointer back; the reentrant version requires the caller to supply a buffer.

## What if You Need Global Variables?

- Sometimes, there are global variables you have to use
- The whole purpose of threads is to share access to certain data
- Must *synchronize* access, but that’s a topic for another day

## Signals

- Short messages to a process
- More accurately, an interrupt sent to a process, much like hardware interrupts to the OS
- Often, signals not explicitly fielded will terminate the process

## Types of Signals

**Environment** Hangup, ˆC, coredump, power failure

**Bugs** Assertion failed, memory error, floating point error, resoure exceeded, pipe closed, bad system call, timeout

**Debugging** Tracing, profiling

**I/O** I/O possible, urgent message, window size change, background process wants to talk to terminal

**Other** Many more

## Default Signal Actions

- Ignore — harmless
- Terminate program
- Create a core dump and terminate program
- Suspend program
- Resume program

## Programs and Signals

- Send a signal to a process or thread
- Call a subroutine when a signal occurs
- Temporarily block signals
- Test for or wait for a signal

## Sending a Signal

- Signals can be generated by the kernel
- Example: when the terminal handler sees ˆC, it sends SIGINT to all processes associated with that terminal
- Signals can be sent explicitly by another process:
  **`kill(pid, signum)`**
- The shell's **`kill`** command translates directly into the **`kill()`** system call
- The default signal is catchable; you have to be more forceful to deal with that if you *really* want to terminate the process. . .

## Catching Signals

- The **`sigaction()`** subroutine specifies what to do when a signal occurs
- Choices: default action, call a subroutine, ignore the signal
- SIGKILL can't be caught; it always terminates the program
- When a signal is caught, other signals can be blocked until the subroutine returns

## Blocking Signals

- **`sigprocmask()`** blocks some signals
- The signals remain pending; when unblocked, they will occur
- SIGKILL can't be blocked, either

## Suspending a Process

- **`sigsuspend()`** suspends a process *and* changes the set of blocked signals
- Cannot accomplish the same thing with

```
sigprocmask(...);
sleep(...);
```

- There's a *race condition* — what if the signal occurs between the two statements?

## Race Conditions

**`sigprocmask(...);`**
Signal occurs
**`sleep(...);`**
*Never wakes up*

**`sigprocmask(...);`**
**`sleep(...);`**
Signal occurs
*Process awakened by signal*

Correct behavior here depends on which event occurs first. Testing is unlikely to find the flaw, since it's possible but improbable. To avoid race condition problems, you need to rely on *atomic operations*, such as **`sigsuspend()`**

## Signals and Processes

- Signal state is inherited by child processes
- Signal state is partially preserved across **`exec()`** calls:
  - Ignored signals remain ignored
  - Caught signals are reset to default action
- Forgetting about ignored signals can cause problems for child processes. (I have a script that resets the action for six different signals before running **gv** — I sometimes need this with my browser.)

## Signals and Threads

- Signal handlers are shared among all threads in a process
- Each thread can have its own signal mask
- If a signal occurs, it's randomly sent to a single thread that doesn't block it

## Signals and Interrupts

- Unix signals are the closest thing to hardware interrupts that you see in application programs
- Dealing with them can be just as messy. . .
- Issues include race conditions, blocking signals, reentrancy, and more