Threads

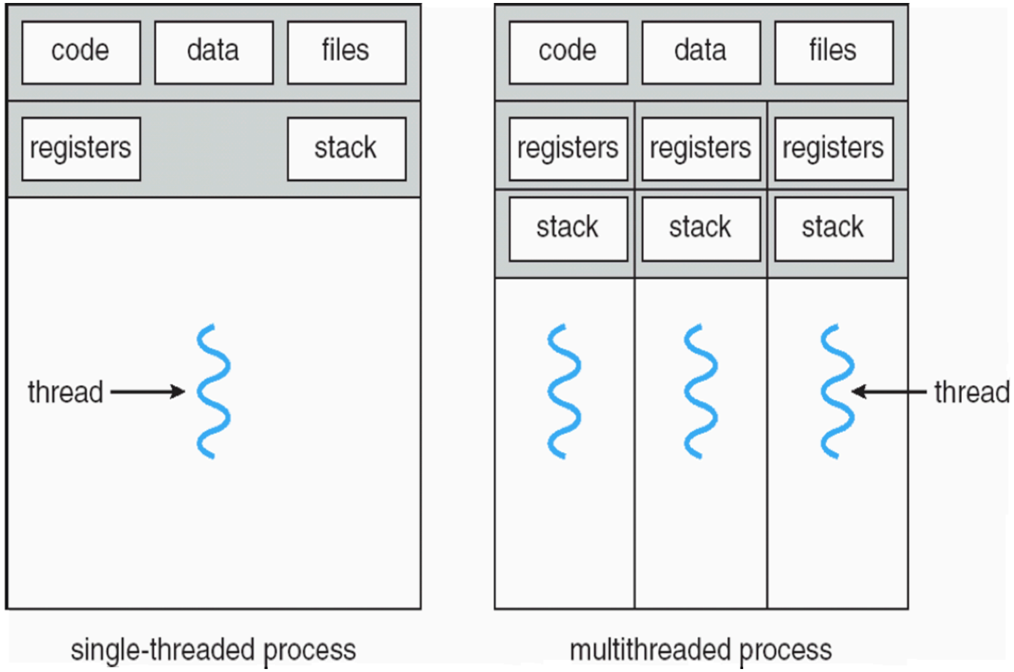
M1 MOSIG – Operating System Design

Renaud Lachaize

Acknowledgments

- Many ideas and slides in these lectures were inspired by or even borrowed from the work of others:
  - Arnaud Legrand, Noël De Palma, Sacha Krakowiak
  - David Mazières (Stanford)
    - (many slides/figures directly adapted from those of the CS140 class)
  - Remzi and Andrea Arpaci-Dusseau (U. Wisconsin)
  - Randall Bryant, David O'Hallaron, Gregory Kesden, Markus Püschel (Carnegie Mellon University)
    - Textbook: Computer Systems: A Programmer's Perspective (2nd Edition) a.k.a. "CSAPP"
    - CS 15-213/18-243 classes (many slides/figures directly adapted from these classes)
  - Textbooks (Silberschatz et al., Tanenbaum)

Threads



- A thread is a schedulable execution context
  - Program counter, stack, registers ...
- By default, a program uses only one thread per process
- But it is also possible to have multi-threaded programs
  - Multiple threads running in the same process address space

Why threads?

- Most popular abstraction for concurrency
  - All threads in a process share memory and file descriptors
  - A lighter-weight abstraction for communication than inter-process communication mechanisms
  - Lower resource consumption: a process context requires more resources (memory, initialization and context switching time) than a thread context

- Allows a process to use multiple CPUs
- Allows a program to overlap I/O and computation
  - Do not block the whole program when only a part of it should be blocked
    - Same benefit as OS running emacs and gcc simultaneously
  - E.g., a threaded Web server can service several clients simultaneously

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

Thread package (pseudo) API

- **`tid thread_create (void (*fn)(void *), void *arg);`**
  - Create a new thread, run **`fn`** with **`arg`**
- **`void thread_exit();`**
  - Destroy current thread
- **`void thread_join(tid thread)`**
  - Wait for thread **`thread`** to exit

- Plus lots of support for synchronization (see next lectures)

- Design choices (details on next slides):
  - A given thread package can provide either preemptive or non-preemptive (a.k.a. cooperative) threads
  - Kernel-level threads versus user-level threads

Preemptive vs. cooperative threads

- Preemptive threads
  - A thread can be preempted at any time in order to allocate the CPU to another execution context, e.g., another thread (from the same process) or another process
  - Rely on time multiplexing, thanks to hardware interrupts (kernel-level) or signals (user-level)
  - Multiple threads (within the same process) can run in parallel on multiple CPUs

- Cooperative threads
  - At most a single thread (within a given process) is allowed to run at a given point in time
  - A thread switch (within a given process) can only happen when:
    - The thread explicitly relinquishes the CPU (calls **`yield()`**)
    - The thread issues a blocking syscall (or terminates)

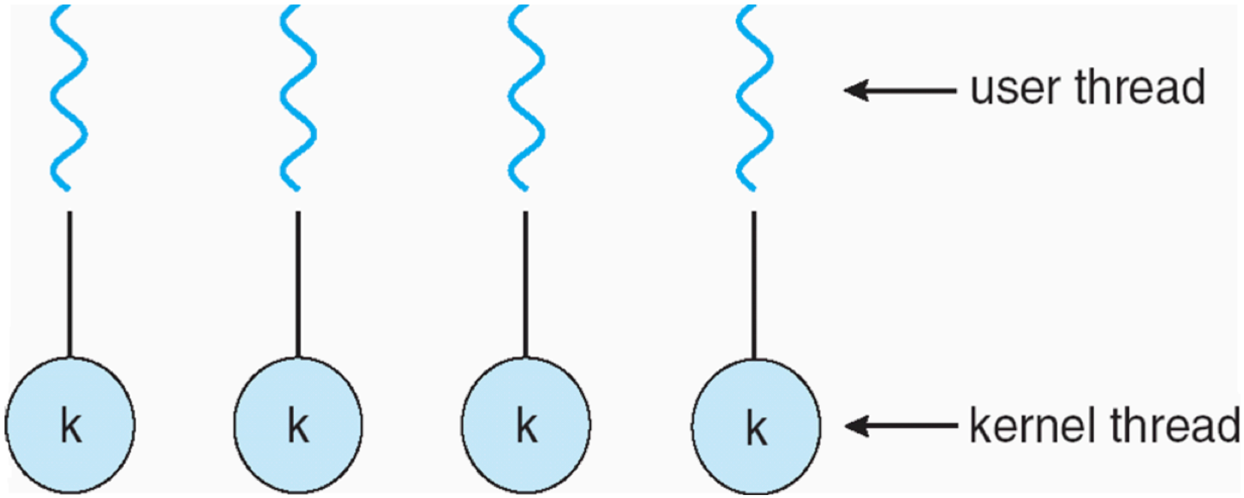
Preemptive vs. cooperative threads (continued)

- Discussion
  - Preemptive threads cause/expose more “race conditions” (i.e., concurrency bugs) because there are many more possible thread interleavings)
    - Cooperative threads provide a simpler programming model for concurrent tasks
  - Cooperative threads cannot take advantage of multiple CPUs
  - Cooperative threads may let a “misbehaving” thread monopolize the CPU
    - But only up to the CPU share of the enclosing process
  - Before multiprocessor architectures became prevalent, many threading implementations were cooperative

Kernel threads vs. user threads

- Kernel threads
  - The kernel is aware that a process may encapsulate several schedulable execution contexts
  - The kernel manages these execution contexts
- User threads
  - Such execution contexts are managed from a library running in user level – the kernel is not aware of them, it only manages the encapsulating process, with a single execution context

Kernel threads

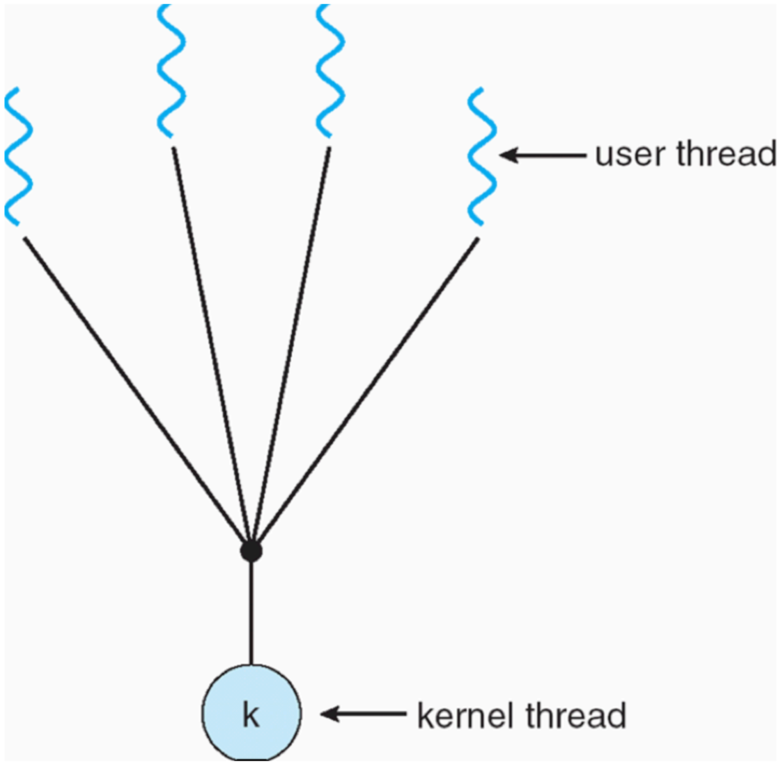


- **`thread_create()`** is implemented as a system call
- How to add **`thread_create()`** to an OS that does not have it?
  - Start with process abstraction in kernel
  - Introduce **`thread_create()`** like process creation with some features stripped out
    - Keep same address space, file table, etc. in new process
- Faster than full process creation but still relatively heavy-weight

Limitations of kernel-level threads

- Every thread operation must go through kernel
  - Create, exit, join, synchronize or switch for any reason
  - On a modern processor, a syscall takes (approx) 100+ cycles, while a function call takes 5 cycles
  - Result: threads 10x-30x slower when implemented in kernel
- One-size-fits-all thread implementation
  - Kernel threads must please all people
  - Maybe you pay for fancy features (priorities, etc.) that you do not need
- General heavy-weight memory requirements
  - E.g., requires a fixed-size stack within kernel
  - Other data structures designed for heavier processes

User threads



- An alternative: implement in user-level library
  - One kernel-thread per process
  - **`thread_create()`**, **`thread_exit()`**, ... are just library functions

Implementing user-level threads (as a library)

- Allocate a new stack for each invocation of **`thread_create()`**
- Keep a queue of runnable threads
- Replace some potentially blocking system calls (e.g., related to I/O: **`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 client
  - "Fake" read wrapper function makes **`read()`** syscall in non-blocking mode – if no data available, schedule another thread
  - On timer tick or when idle, check which connections have new data, and switch to a thread that can make progress

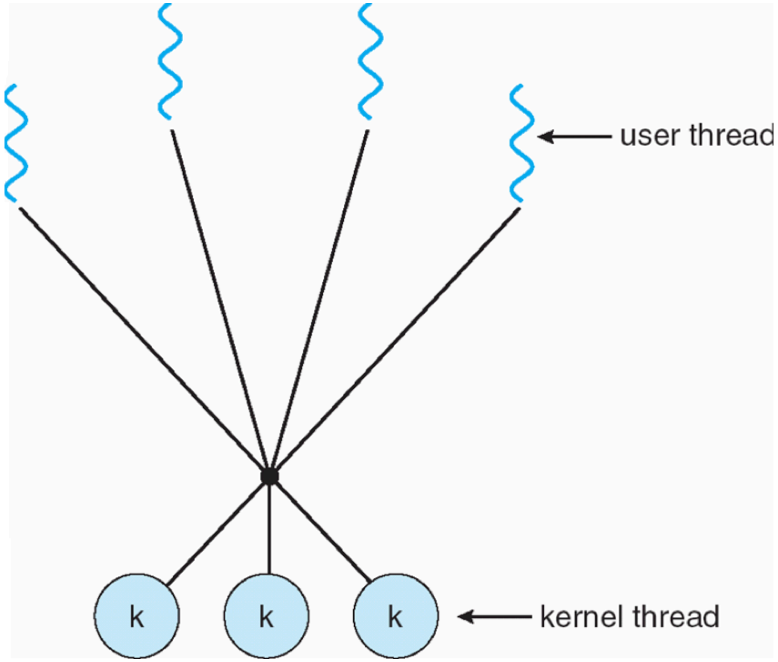
Limitations of user-level threads

- Cannot take advantage of multiple CPUs

- A blocking system call blocks all threads (within the same process)
  – Some system calls can be replaced by non blocking ones (e.g., to read from network connections)
  – But, depending on the OS, this is not always possible for all potentially-blocking system calls (e.g., for disk I/O)
    - Such system calls may block all the threads of a given process

- A page fault blocks all threads (within the same process)
- Possible deadlock if one thread blocks on another
  – May block entire process and make no progress (more on deadlocks in another lecture)

Another possible threading design: user threads on (several) kernel threads



- User threads implemented on kernel threads
  - Multiple kernel-level threads per process
  - **`thread_create()`**, **`thread_exit()`** are still library functions
- Sometimes called *n:m* threading or "hybrid" threading
  - Have *n* user threads per *m* kernel threads
  - ("simple" user-level threads are *n:1* and "simple" kernel threads are *1:1*)

Limitations of *n:m* threading

- Many of the same problems as *n:1* threads
  - Blocked threads, deadlock, ...

- Hard to keep the number of kernel threads the same as available CPUs
  - The kernel knows how many CPUs are available
  - The kernel also knows which kernel-level threads are blocked
  - But tries to hide these things to applications for transparency
  - So a user-level thread scheduler might think that a thread is running while the underlying kernel thread is blocked

- The kernel does not know the relative importance of threads
  - Might preempt kernel thread in which library holds important lock

Advanced details

Threads: behavior upon `fork()`/`exec()`

- What happens if one thread of a program calls **`fork()`**?
  - Does the new process duplicate all threads? Or is the new process single-threaded?
  - Some Unix systems have chosen to have two versions of **`fork()`**
  - In general, only the calling thread is replicated in the child process
    - All of the other threads vanish in the child, without invoking thread-specific cleanup handlers

- What happens if one thread of a program calls **`exec()`**?
  - Generally, the program replaces the entire process, including all threads
    - Without invoking any thread-specific cleanup handler

Thread cancellation

- One may want to cancel a thread before it has completed
  - Example: when multiple threads concurrently search for a given data item in a database
  - Or when you hit the stop button of a Web browser, all the threads in charge of loading the code of the web page and the various images should be cancelled
- Asynchronous cancellation
  - One thread immediately terminates the target thread
  - Main issue: what if resources have been allocated and/or the target thread is in the midst of updating data shared with other threads?
  - May lead to incoherent state
- Deferred cancellation
  - The target thread periodically checks whether it should terminate, giving it an opportunity to terminate itself in an orderly fashion
  - Such points are called cancellation points

Signal handling

- Handling signals in a single-threaded program is straightforward
- In a multi-threaded program, who should receive the signal? Several possibilities:
  - Deliver the signal to the thread to which the signal applies (e.g., **`SIGSEGV`**)
  - Deliver the signal to every thread in the process
  - Deliver the signal to certain threads in the process
  - Assign a specific thread to receive all signals for the process
- POSIX threads have the **`pthread_kill(pthread_t tid, int signal)`** function
- In many Unix systems, the decision is usually made as follows:
  - Only a single thread receives a given signal instance within a process
  - If the signal is clearly related to a given thread, select this one
    - E.g., in case of a hardware fault (like **`SIGSEGV`**), or a call to **`pthread_kill()`**
    - Otherwise, select an arbitrary thread within the process

Thread-specific data

- All threads share the data of the enclosing process
- In some circumstances, each thread may need to have its own copy of certain data
- Most thread libraries provide some support for thread-specific data:
  - POSIX Thread-specific data (a relatively complex API)
  - "Thread local storage" (non-standard but simpler and implemented in different Unix variants like Linux, FreeBSD and Solaris)
- Thread-local storage – example:
  - Simply include the **`__thread`** specifier in the declaration of a global or static variable
  - **`static __thread char buf[BUF_SIZE];`**

Thread pools

- A Web server could create a thread to handle each client request
  - Although it is cheaper than creating a process, creating a thread is costly, especially regarding the request service time
  - If there is no bound on the number of concurrently active threads, we could exhaust the system resources (CPU, RAM) and cause thrashing
- Thread pools address these two issues
  - Create a number of threads when the (server application) process starts and place them into a pool where they wait for work
  - When a server receives a request, it awakens a thread from the pool if any available and waits otherwise
  - When the thread has finished servicing the request, it returns to the pool, awaiting for more work