Threads

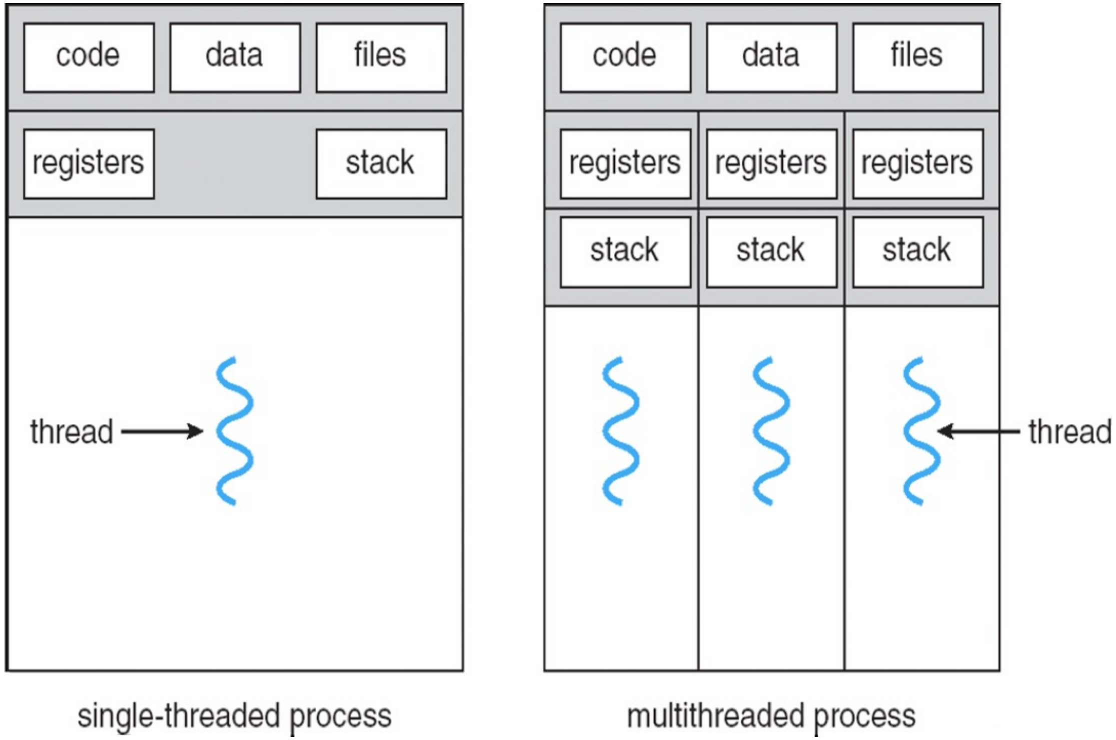
M1 MOSIG – Operating System Design

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 process uses only one thread
- But it is also possible to have a multi-threaded process
  - Multiple threads running in the same memory 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 (e.g., pipes, sockets, files)
  - 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 (parallel execution)**

- **Allows a program to overlap I/O and computation**
  - Do not block the whole process when only a part of it should be blocked
  - E.g., a threaded Web server can handle several clients simultaneously

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

- And also lots of support for synchronization (see next lectures)

- **Some important 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** 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 timer interrupts
  - **Multiple threads (within the same process) can run in parallel on multiple CPUs.**

- **Cooperative threads**
  - Within a given process, **at most a single thread** is allowed to run at a given point in time.
  - Within a given process, **a thread switch can only happen when**:
    - the thread explicitly releases the CPU (calls `yield()` or terminates)
    - the thread issues a blocking syscall (e.g., for disk or network I/O)
  - Note: parallel execution & preemption w.r.t. other processes remain possible.

Preemptive vs. cooperative threads (continued)

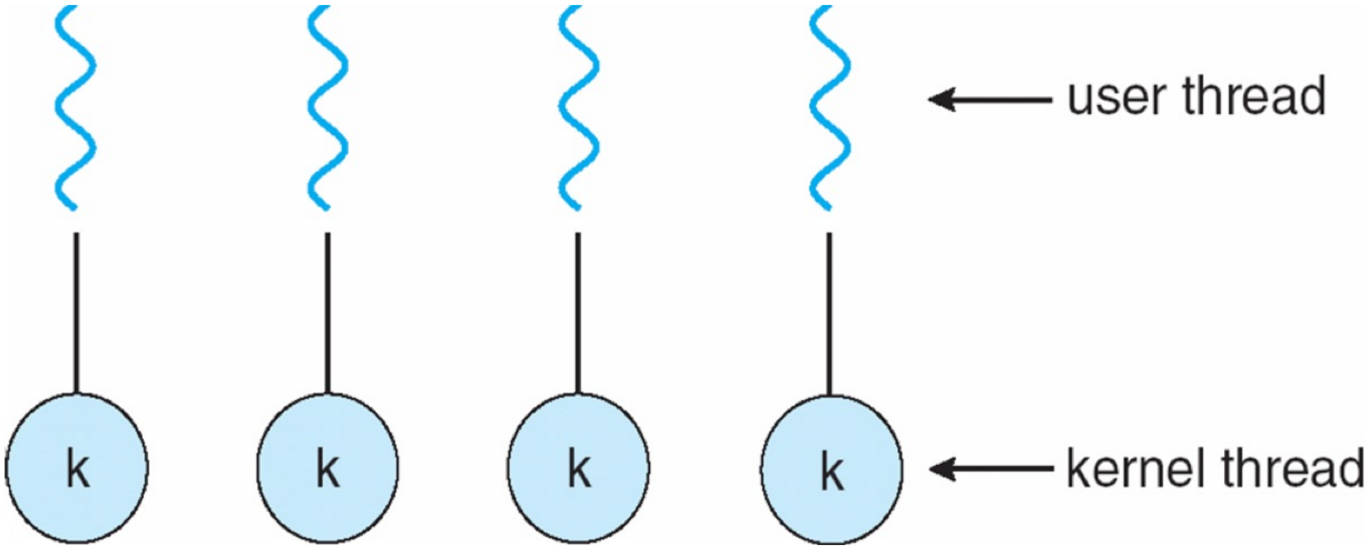
- **Discussion**
  - Preemptive threads cause/expose more concurrency bugs (studied in upcoming lectures) because there are many more possible thread interleavings)
    - Cooperative threads provide a simpler programming model for concurrent tasks
  - Cooperative threads cannot take advantage of multiple 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” (kernel-managed threads)**
  - The kernel is aware that a process may encapsulate several schedulable execution contexts.
  - The kernel manages these execution contexts.

- **“User threads” (user-managed 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

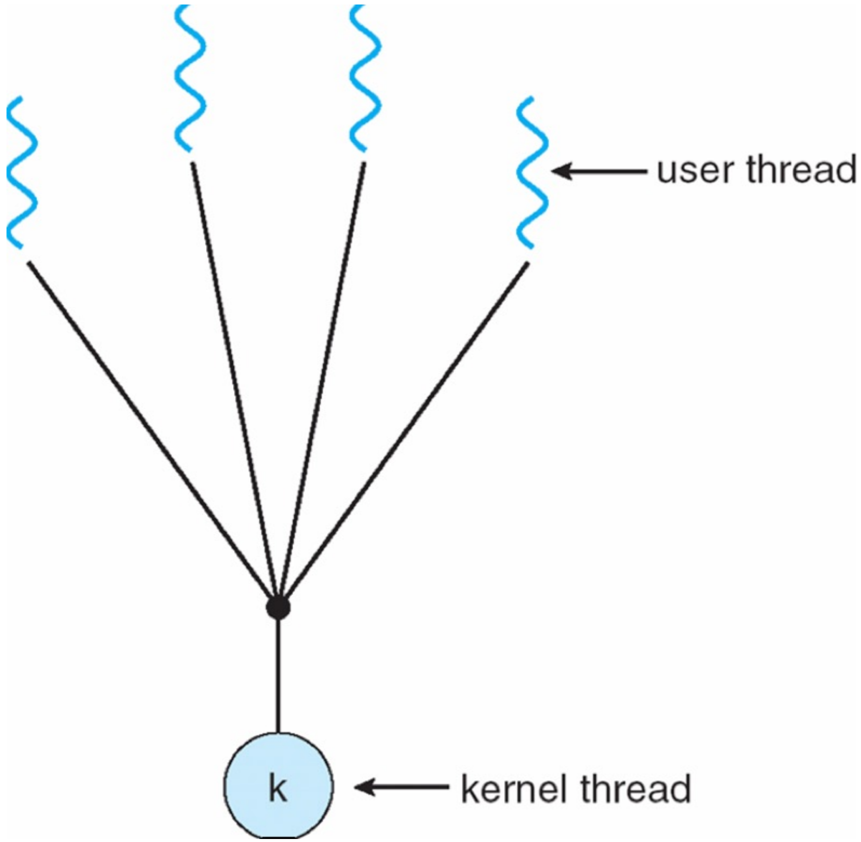


- **`thread_create()`** is implemented as a system call
- Faster than full process creation but still relatively heavy-weight

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
- **Heavier memory requirements**
  - E.g., each kernel thread requires a fixed-size stack within kernel (in addition to its user-level stack)
- One-size-fits-all thread implementation
  - Kernel threads must please all people
  - Maybe you pay (time and space overhead) for fancy features (priorities, etc.) that you do not need

User threads



- Thread management implemented in a user-level library
  - One kernel-thread per process
  - **`thread_create()`**, **`thread_exit()`**, ... are just library functions

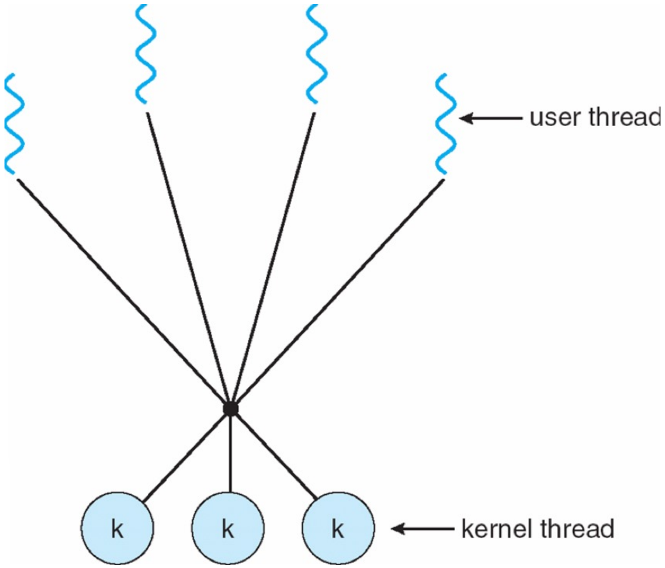
Implementing user-level threads (as a library) [Advanced]

- 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.) with non-blocking version.
  - If operation would block, switch and run different thread.
- Schedule periodic timer signal (**`setitimer()`** and **`SIGALRM`**).
  - Switch to another thread upon arrival of Unix signal triggered by user-defined timer (preemption).

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).
- **A page fault blocks all threads** within the same process.
  - (More on page faults in another lecture.)
- **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-level threads implemented on top of kernel-level threads**
  - Multiple kernel-level threads per process
  - **`thread_create()`**, **`thread_exit()`** are still library functions
- **Sometimes called "*N:M* threading" (or "*M:N*") 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 and 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 process 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 process 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
  - Example: `static __thread char buf[BUF_SIZE];`

Thread pools

- **A server application (for example, a Web server) could create a thread to handle each client request … but this brings issues:**
  - 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 above issues – Principle:**
  - 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, waiting for more work