PROCESSES AND THREADS
THREADING MODELS

CS124 – Operating Systems

Spring 2024, Lecture 7

Processes and Threads

- As previously described, processes have one sequential **thread** of execution
- Increasingly, operating systems offer the ability to have multiple concurrent threads of execution in a process
  - Individual threads can execute only one instruction at a time
  - Multiple threads in a process allow multiple tasks to be performed **concurrently**, “at the same time” (i.e. overlapping logical control-flows)
- Requires changes to the process model:
  - CPU state can no longer be managed on a per-process basis
  - Must manage CPU state on a per-thread basis
  - All other resources can be managed on a per-process basis

Processes and Threads (2)

Single-threaded process

- Per-process items:
  - Address space / page table
  - Program text (i.e. the code)
  - CPU registers
  - Program counter
  - Stack and stack pointer
  - Global variables
  - Memory heap
  - Signal handlers
  - Open files, sockets, etc.
  - Child processes

Multithreaded process

- Per-process items:
  - Address space / page table
  - Program text
  - Global variables
  - Memory heap
  - Signal handlers
  - Open files, sockets, etc.
  - Child processes
- Per-thread items:
  - CPU registers
  - Program counter
  - Stack and stack pointer

Why Multithreaded Processes?

- Two big reasons why multithreading is desirable:
  - <u>Reason 1</u>: Performance (obvious)
  - <u>Reason 2</u>: A cleaner abstraction for concurrent operations
- Lots of ways that multithreading can improve performance
- **Responsiveness**:
  - Apps that perform slow or long-running tasks can do them on background threads
  - A foreground thread responds to user interactions immediately
  - Responsive applications = happy users ☺
- Web browsers are a common example of this pattern
  - User-interface thread draws the web page, handles mouse clicks
  - A pool of background threads handles content downloads from remote servers
  - UI thread updates display as downloaded files become available

Multithreading: Responsiveness

- Common web browser pattern:
  - User-interface thread draws the web page, handles mouse clicks
  - A pool of background threads handles content downloads from remote servers
- Does this require multiple CPUs to yield a benefit?
  - **<u>NO</u>!**
  - Background threads will usually be blocked on I/O, or waiting for work to do – they won’t occupy the CPU
  - Similar case for UI thread – waiting for user interaction
- Even with a single physical processor, multithreading can greatly improve application responsiveness
  - Particularly in cases where most tasks are I/O bound

Multithreading: Scalability

- Example: a large scientific/mathematical computation
  - Instead of performing this computation in a single thread, split it into multiple concurrently executing threads
- Does this require multiple CPUs to yield a benefit?
  - **YES!**
  - Threads will mostly be CPU-bound, not I/O-bound
  - If there is only one CPU in the system, multiple threads will probably make the program slower instead of faster (extra context-switches, synchronization overhead, etc.)
- If there are multiple CPUs in the system:
  - A single-threaded process cannot take advantage of multiple CPUs
  - Only way to utilize multiple CPUs is to run multiple processes, or to run a process with multiple threads
  - Multithreading facilitates **scalability** with available hardware

Multithreading: Scalability (2)

- A major difference between concurrency and parallelism

- **Concurrency** means that multiple tasks have overlapping logical control flows
- Concurrency does not require multiple processors
  - A one-CPU system can achieve this by switching back and forth between concurrent tasks at appropriate points in time
  - Concurrency doesn’t necessarily imply that multiple tasks’ instructions are being executed at the same time, just that their execution is overlapping/interleaved in some way

- **Parallelism** means that multiple tasks are actually executing <u>at the same time</u>
  - i.e. multiple processors are executing different tasks’ instructions at exactly the same time

Multithreading: Scalability (3)

- Example: a large scientific/mathematical computation
  - Instead of performing this computation in a single thread, split it into multiple concurrently executing threads
- If a system has multiple CPUs, can improve computation's performance by running one thread per CPU
  - Threads will actually execute in parallel
- Assume program takes 1 unit of time to complete on 1 CPU
  - Ideally, running the program on *N* CPUs will result in it taking 1/*N* the time to complete (i.e. a **speedup** of *N*)
- The reality isn't always so nice…
  - Most computations have parts that must be performed sequentially, cannot be parallelized
  - The sequential parts restrict max possible speedup achievable by parallelizing the task

Amdahl's Law

- Amdahl's Law is a simple formula that captures this issue
- Given: a task where S is the percentage of the task that must be executed serially (i.e. cannot be parallelized)
  - On a single-processor machine the task takes 1 unit of time to run
  - On an N-processor machine, the task will take $S + (1 - S) / N$ units of time to run
  - The speedup due to parallelism will be $(S + (1 - S) / N)^{-1}$
- Example: a task with 10% that must be run serially
  - 1.8x speedup on 2 CPUs
  - 3.1x speedup on 4 CPUs
  - 4.7x speedup on 8 CPUs
  - As $N \rightarrow \infty$, speedup $\rightarrow$ 10x, and that's it. ☹

Amdahl's Law (2)

- Amdahl's Law is bad news for speeding up fixed-size tasks with more CPUs…
- Many tasks are variable in size:
  - Given more computing resources, users will increase the size of the task to use all available computing resources
  - Focus isn't solely on reducing the time to complete the task
- Also, many variable-size tasks have this characteristic:
  - As the task's size increases, the size of parallelizable part of the task increases faster than size of the serial part of the task
  - Percentage of the task that must be executed serially will decrease!
- Such tasks still see improved performance by increasing parallelism
  - Formulated as Gustafson-Barsis' Law (1988)
  - Not a contradiction of Amdahl's Law, just different constraints

Multithreading: Economy

- Multithreaded processes have two other performance-related benefits: **resource sharing** and **economy**
- Threads are generally much faster to create and destroy than processes
  - Fewer resources to allocate or release: most resources managed on a per-process basis
- Context-switching between multiple threads in the same process tends to be much faster
  - Threads share one address space: don't need to change the page table being used, etc.
  - (Switching between threads in different processes is still slower.)
- Sharing resources (e.g. files, sockets) between threads is <u>much</u> easier than sharing them between processes

Multithreading: Abstractions

- Another benefit of threads: a cleaner abstraction
- Why are long-running system calls blocking, anyway?
  - i.e. why do they force the process to wait until request is completed
- Blocking operations are simply much easier to use
- Alternative: asynchronous (non-blocking) operations
  - Initiate a long-running operation in the system.
  - Periodically check to see if the operation is complete.
    If not, go do other things while you wait.
  - When operation finally completes, go on to next steps in your task.
- Most systems provide asynchronous I/O APIs alongside blocking I/O APIs
  - Primarily used for asynchronous networking I/O
  - Asynchronous filesystem APIs are becoming increasingly common

Asynchronous I/O

- UNIX API examples:

```
select(int nfds, fd_set *readfds, fd_set *writefds,
       fd_set *exceptfds, timeval *timeout)
poll(pollfd *fds, nfds_t nfds, int timeout)
```

- Both allow a collection of file-descriptors to be monitored
  - Returns if a file-descriptor can be read or written without blocking, if an error occurs on a file-descriptor, or if the call times out
- Applications usually use non-blocking I/O when they want to achieve very high performance
  - (OSes heavily optimize these functions to be fast and scalable)
- However, greatly increases implementation complexity

Example: Web Server

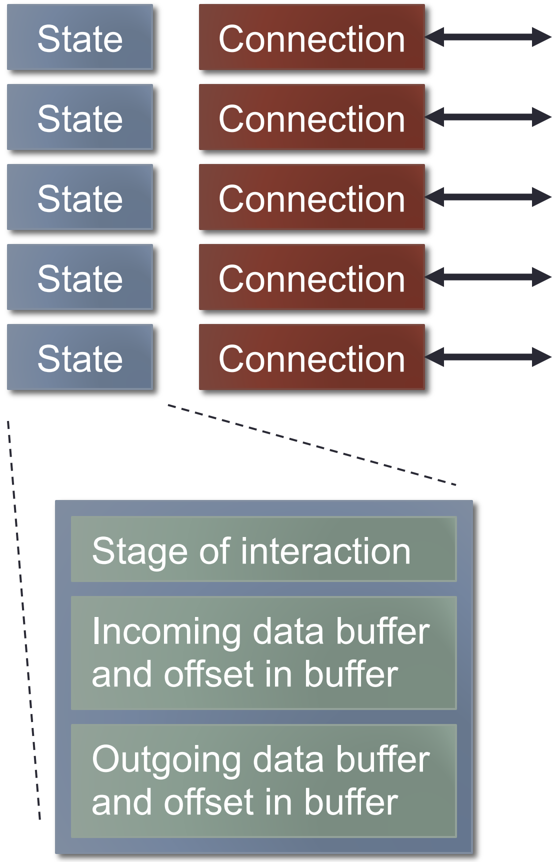
- Basic webserver operation, per request:
  - Accept an incoming socket connection
  - Receive the HTTP request over the socket
  - Access the file(s) specified in the HTTP request
  - Send an HTTP response back to the client
- Of course, want to handle requests as fast as possible
  - Even handle multiple incoming requests concurrently, if possible
- Most of these operations are long-running tasks
- Can imagine how webserver would be implemented with these approaches:
  - Single-threaded process with blocking network I/O
  - Single-threaded process with non-blocking network I/O
  - Multithreaded process with blocking network I/O

Web Server, Single-Threaded Style

- Webserver implemented as single-threaded process with blocking network IO:
  - Can code this very easily: write a loop that just processes each request and sends each response in sequence
  - Web server can't do anything else while receiving a request, or sending a response (basically always blocked on I/O)
  - Web clients will spend a lot of time waiting on the server
- Non-blocking I/O allows us to achieve concurrency without multiple threads
  - Allows us to overlap the networking operations of multiple requests/responses (concurrency!)
  - A given request/response will still take the same time to complete, but overall throughput will be <u>much</u> higher
  - Server is more likely to be CPU-bound, rather than I/O-bound

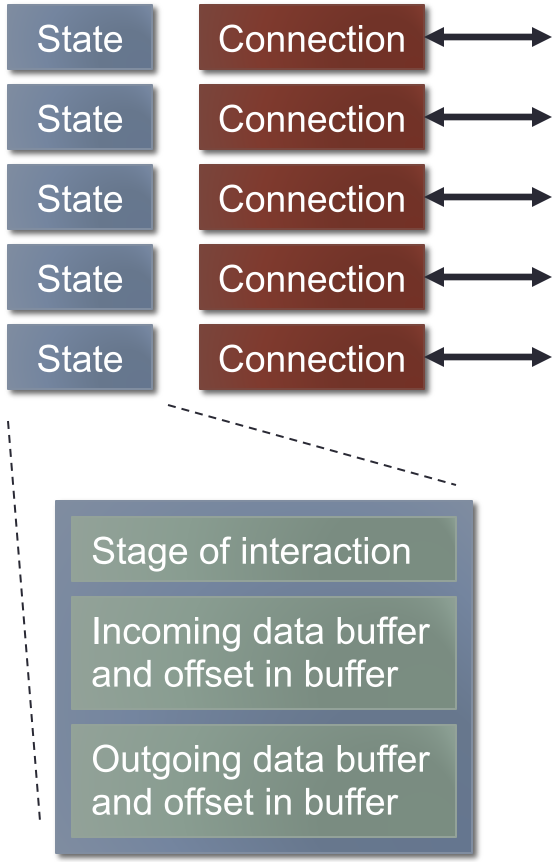
Web Server, Single-Threaded Style (2)

- Webserver with non-blocking I/O:
  - Will have many sockets open to many clients, servicing requests
- Must keep track of the state of every in-flight request/response interaction:
  - What stage of request/response cycle is each connection at?
  - Receiving the request? If so, where is request data being buffered, and where does new data get written in the buffer?
  - Sending the response? If so, how much of the file has been sent? Or, is the webserver sending an error response?



Web Server, Single-Threaded Style (3)

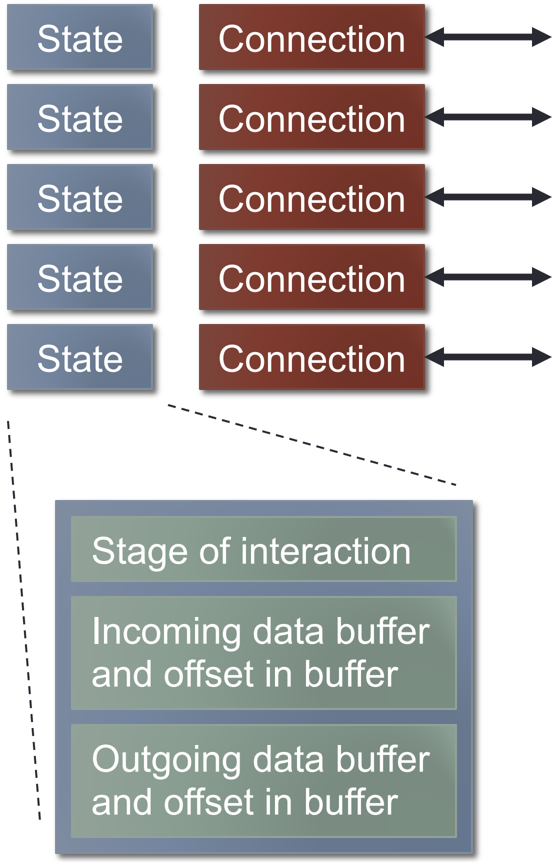
- Web server main-loop:
  - Wait for some socket(s) to become active (i.e. can send/receive without blocking)
  - For each active socket, get the current state of that socket's interaction, and do as much work as possible without blocking
  - Once all active sockets are handled, go back and wait some more!
- Server basically implements a finite state machine for each open connection



Web Server, Single-Threaded Style (4)

- Example pseudocode:

```
if stage is RECV_REQUEST:
    receive more data into input buffer
    if all data received:
        generate response into output buffer
        stage = SEND_RESPONSE
else if stage is SEND_RESPONSE:
    send more data from output buffer
    if all data sent:
        close connection
        remove state and connection from arrays
```



- Responsibility of implementing concurrency of tasks has fallen on the webserver, not on the OS ☹
  - (It's complicated, and prone to bugs.)

Web Server, Multithreaded Style

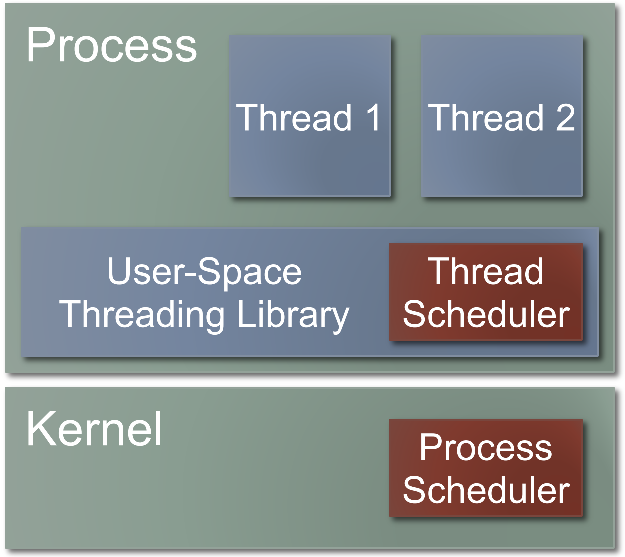
- Multithreaded processes allow applications to achieve concurrency while still using blocking system calls
    - The operating system implements the concurrency
    - Apps only have to worry about coordination between threads
- Multithreaded webserver with blocking network I/O:
    - Each thread executes a simple sequence of steps, identical to the original single-threaded webserver with blocking calls:
        - Receive the HTTP request over the socket
        - Access the file(s) specified in the HTTP request
        - Send an HTTP response back to the client
    - Can start as many threads as we need
    - (With an I/O-bound problem like this, can usually start many more threads than CPUs in the system, and still see performance gains)

Aside: Non-Blocking I/O

- Non-blocking I/O in a single-threaded process is pretty complicated…
- Nonetheless, it is often the fastest possible approach
- Used by highly scalable servers
    - Avoids a significant amount of overhead from e.g. context-switching between threads, kernel scheduler invocations, etc.
    - Reduces space requirements as well (e.g. don't need stacks for multiple threads, can optimize storage of task details
    - One thread waiting on a large collection of sockets is much more efficient than many threads each waiting on one socket
- Example: NGINX ("engine-x") web server
    - Easily supports 10000+ concurrent connections (C10K problem)
    - Used by Facebook, Dropbox, Wikipedia, Wordpress, etc.

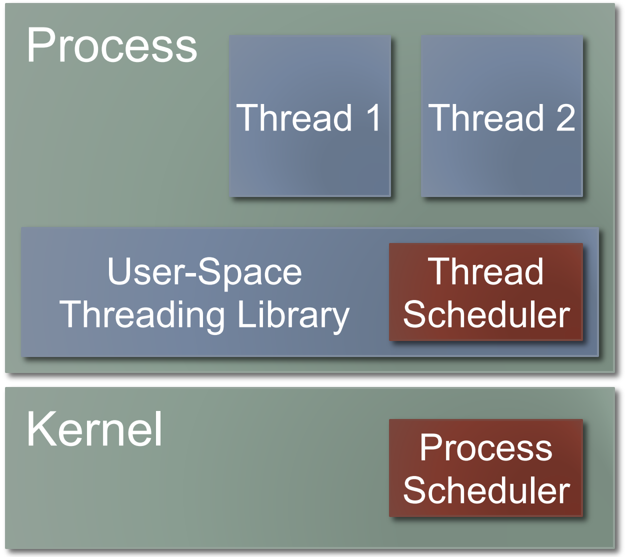
Implementing Threads

- Several different approaches to implementing threads
- Can implement multithreading entirely in user mode
  - a.k.a. user-mode/userspace threading libraries, or "**user threads**"
  - Kernel only provides a process abstraction, is unaware of threads
- Excellent for platforms that don't support multithreading at the kernel level (less common now)
- Such libraries often provide cooperative multithreading
  - Difficult / grungy to set up a periodic timer to drive thread preemption
  - Often, smallest timer interval available in user-space is still pretty large
  - Frequent timer interrupts can degrade performance of other applications, etc.



User Threads

- Benefit: user-mode thread management is <u>very</u> fast
  - No trapping to kernel to create/destroy threads, switch threads, etc.
- Problem: often want to use threads to achieve concurrency in programs with blocking system calls
  - Blocking system calls require a trap into the kernel…
  - The kernel will simply context-switch to another process!
  - When a thread makes a long-running call, other user-mode threads in the same process won't get to run
- Problem: often want to use threads to take advantage of multiple CPUs
  - Again, the kernel is unaware of user-mode threads; it only schedules <u>processes</u> on CPUs
- User threading is very limited



Kernel Threading Support

- Another option is to provide threading support in the kernel
  - Basically all modern operating systems have this capability now
- Kernel can be more intelligent about thread scheduling
- Multithreading and blocking system calls:
  - If one thread in a process makes a system call and blocks, but another thread in same process can proceed, switch to 2nd thread
  - Saves some overhead of context-switching (e.g. MMU updates)
- Multithreading and parallelism:
  - On multiprocessor systems, the kernel can schedule threads from the same process on different CPUs
- Drawback: thread-management calls now require a trap
  - Creating/destroying threads, context-switch between threads, etc.

Kernel Threads

- Ultimately, the operating system is what implements and provides multitasking support…
- Each thread a user application has, must correspond to some schedulable, kernel-level task
    - (Multiple user-level threads can map to the same kernel task)
- Minimal form of schedulable task inside the kernel is called a **kernel thread**
    - Thread's context contains CPU registers, program counter, stack, stack pointer, flags, etc.
    - This is not a process! Every process may have a corresponding kernel thread, but the kernel thread itself is very lightweight.
- Individual kernel threads can become blocked, can be resumed, etc.

Threading Models

- Different threading models have different ways of mapping “user threads” (threads in an application) to kernel threads
- The **many-to-one threading model** maps many user threads to a single kernel thread
  - In this case, the kernel thread basically manages a process
- This model corresponds to the user-mode threading library implementation
- Example:
  - All user threads in a process are mapped to one kernel thread
  - One user thread decides to perform a blocking operation…
  - The kernel thread becomes blocked, preventing all other user threads from progressing
- The GNU Portable Threads library follows this model

Threading Models (2)

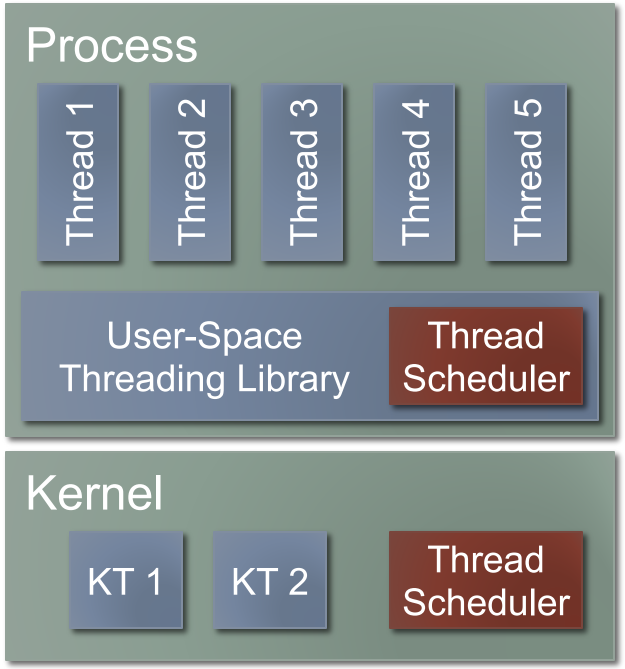
- The **one-to-one threading model** maps every user thread to its own kernel thread
- This model corresponds to a kernel-supported threading library implementation
- Example:
  - Each user thread in a process is mapped to its own kernel thread
  - One user thread decides to perform a blocking operation…
  - That kernel thread becomes blocked…
  - Since every other user thread has its own kernel thread, other user threads are unaffected by the blocked thread
- This is the model that most OSes now provide
  - Tends to be the most straightforward to implement

Threading Models (3)

- A few OSes implement a **many-to-many** or **hybrid threading model**
  - Many user threads mapped to many (usually fewer) kernel threads
- Premise:
  - Both user-mode threading and kernel threading have benefits!
  - User-space threading is very lightweight and inexpensive, but weak
  - Kernel threading is powerful, but slower and more resource-heavy
- Given: *N* user threads, *M* kernel threads (*M* < *N*)
  - Try to map user threads to kernel threads to maximize benefits
  - e.g. creating and destroying many short-lived threads will be cheap
  - e.g. many cooperating user threads can be mapped to one kernel thread, reducing syscalls and kernel-level context switches
  - e.g. if a user thread blocks on I/O frequently, assign it a dedicated kernel thread to keep it from blocking other user threads

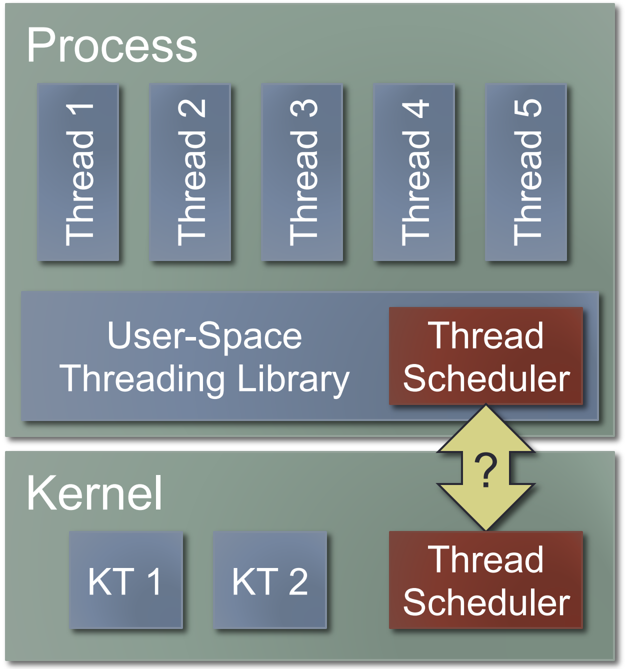
Threading Models (4)

- Many-to-many model appears to be the best solution…
- Unfortunately, it is extremely difficult to implement
  - So difficult that most OSes simply use the one-to-one model
  - Windows 7 implements a hybrid threading model
    - Previous versions of Windows implemented a one-to-one model
- Problem: thread management code is spread between userspace library and the kernel
  - These layers must collaborate closely to maximize the performance benefits of combining the two threading models



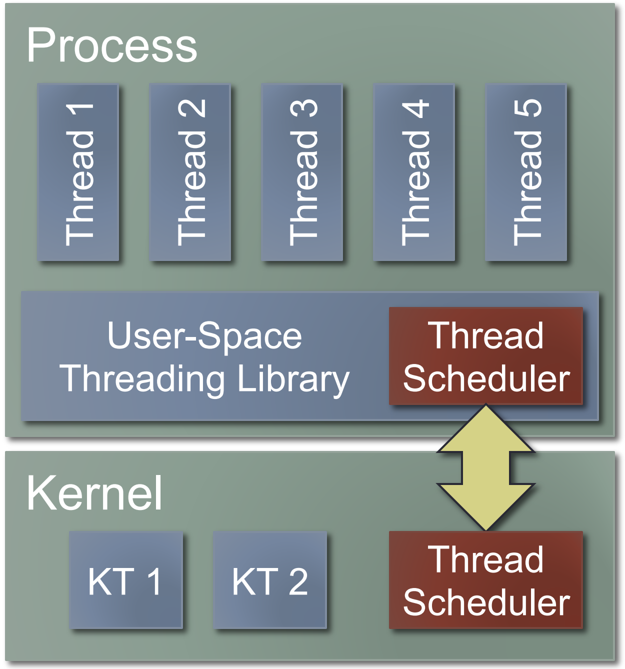
Threading Models (5)

- Without coordination, user threading library has little hope of effectively managing the mapping to kernel threads
- Can user-thread layer intercept blocking syscalls?
  - If so, other user threads on same kernel thread can be reassigned to prevent them from being blocked
  - If not, very likely that user threads sharing a kernel thread will become blocked
- Can user-thread layer access kernel-level details of thread behavior?
  - e.g. if kernel reports compute-intensive tasks, user thread library can assign them to different kernel threads to run on multiple CPUs
  - If not, system can't take full advantage of multiple CPUs to maximize performance



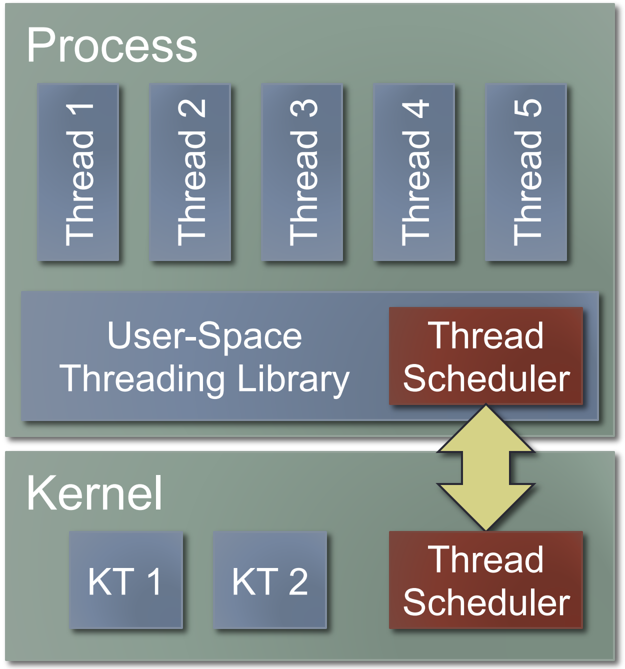
Scheduler Activations

- Clearly, user-space threading library and kernel threading layer must communicate for hybrid threading to work…
- Most widely used approach called **scheduler activations**
- Kernel allows processes to register for scheduling events
  - "A kernel thread was preempted"
  - "A kernel thread is about to block"
  - "A kernel thread is about to be unblocked"
  - "A kernel thread caused a page fault"
  - etc.
- When kernel scheduler detects such an event, it makes an **upcall** to the user-space event handler
  - The **upcall handler** responds to the event, then the kernel goes on with its tasks



Scheduler Activations (2)

- The user-space threading library can register an upcall handler to receive kernel scheduling events
  - Library can map user threads to kernel threads more intelligently!
  - Library can even request additional kernel threads on behalf of the application, depending on app's thread behavior
  - (Kernel threads are sometimes called "**lightweight processes**" in this approach)
- Problem: this mechanism can greatly affect system performance
  - Additional transitions between user-mode and kernel-mode during scheduling…
  - More time spent scheduling, and less time spent executing the application's code
- Approach hasn't seen widespread adoption at this point



Scheduler Activations (3)

- Marcel threading library is most notable example of “scheduler activations” mechanism
    - http://runtime.bordeaux.inria.fr/marcel/

Next Time

- More kernel thread implementation details