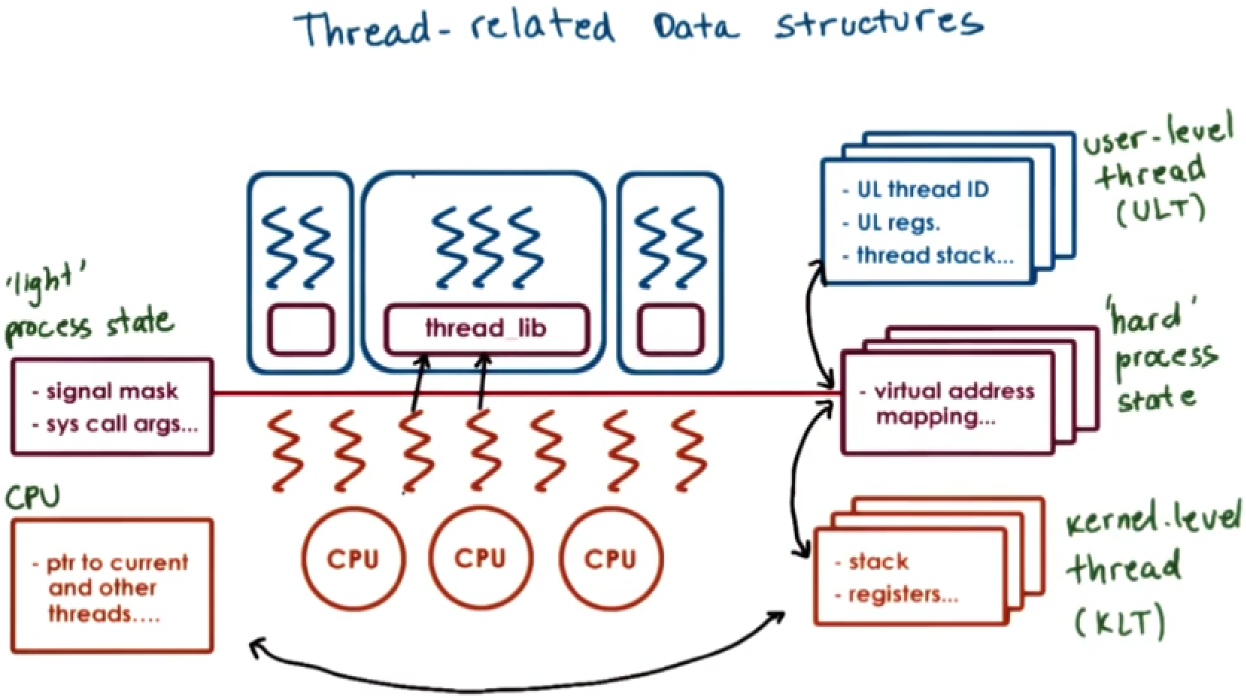
Thread Design Considerations

Kernel vs. User Level Threads

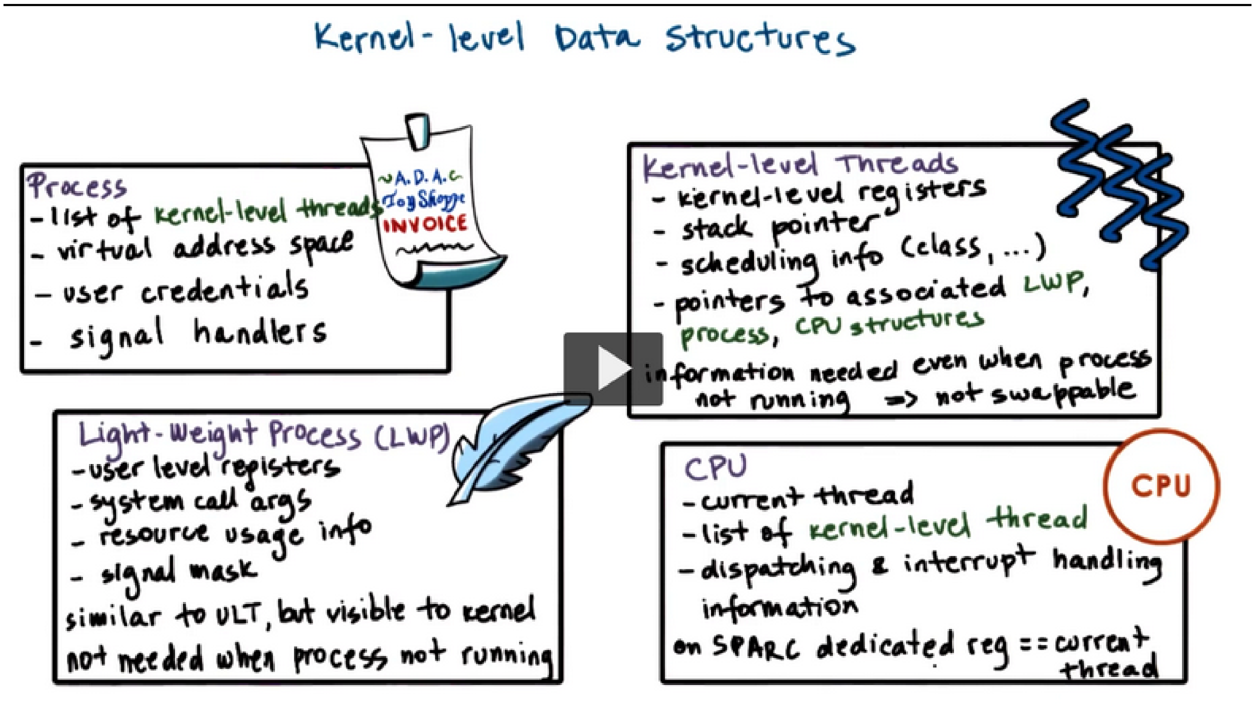
1. Both kernel- and user-level threads must provide:
    - Thread abstraction (data structure to define a thread)
    - Scheduling
    - Synchronization
2. The difference is whether these are implemented by the kernel or a library
3. The user-level threads must be mapped to kernel-level threads
    - Must support one-to-one, many-to-one, and many-to-many mappings

Thread Data Structures (Single CPU)

1. Process Control Block (split between user- and kernel-level threads)
    - Virtual address mapping
    - Stack
    - Registers
2. User-Level Thread
    - Thread ID
    - User-level registers
    - Thread stack
3. Kernel-level Thread
    - Stack
    - Registers
4. Light Process State
    - Signal Mask
    - System call arguments
5. Hard Process State
    - Virtual Address Mapping



Thread-Related Data Structures



Kernel-level Data Structures

Rationale for Multiple Datastructures

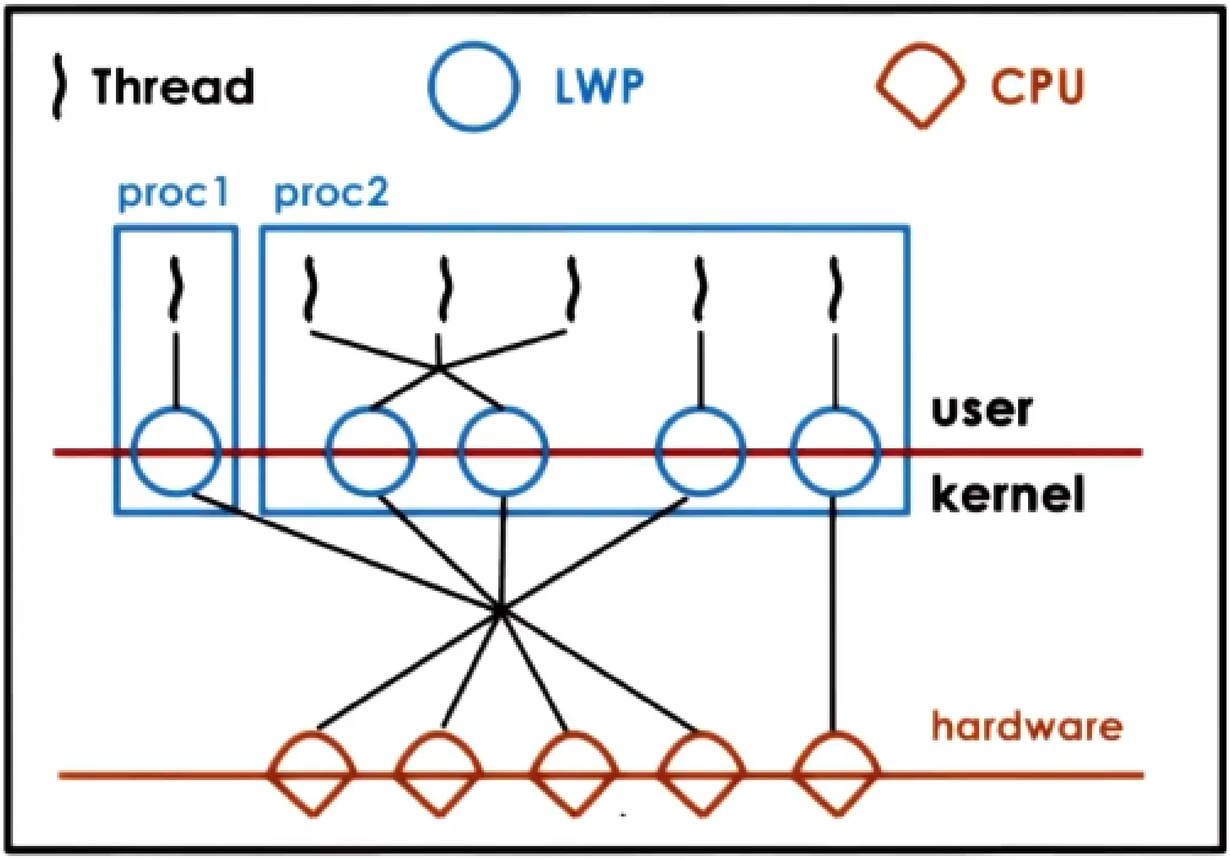
1. Single PCB

- Large, continuous data structure (limits scalability)
- Private for each entity (limits overhead)
- Saved and restored on each context switch (limits performance)
- Update for any changes (limits flexibility)

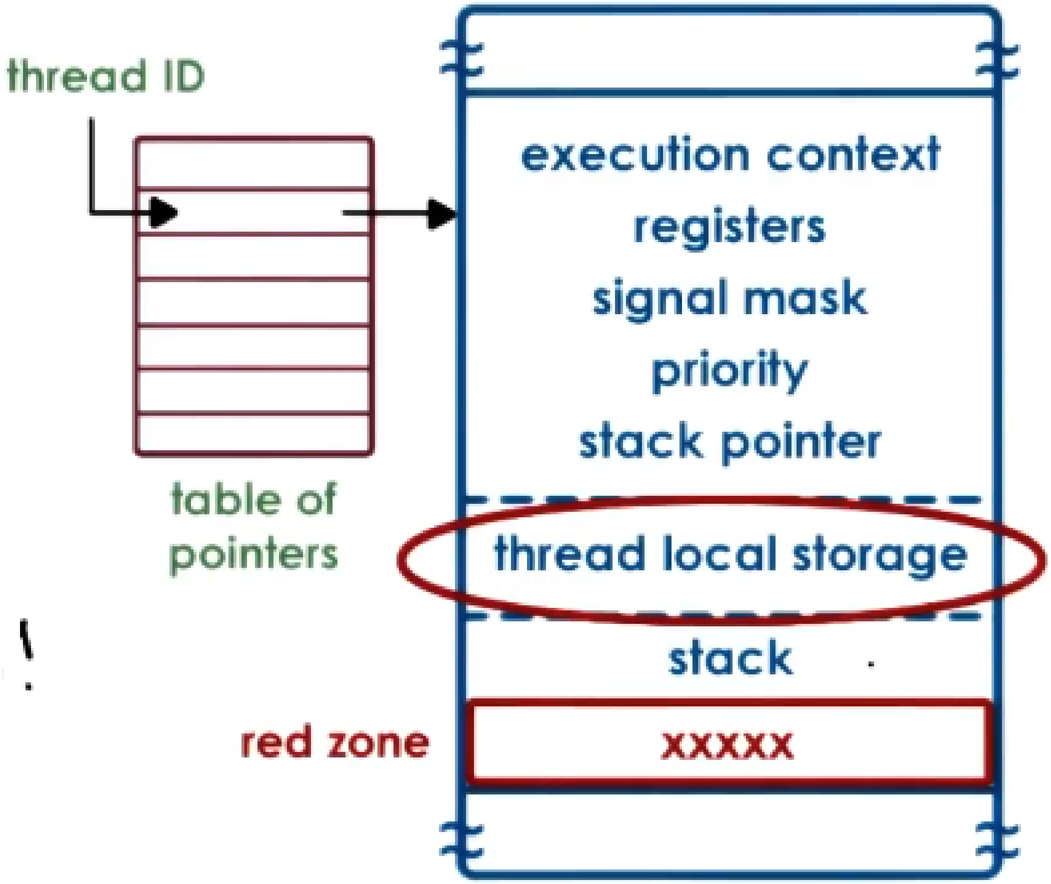
2. Multiple Data Structures
    - Smaller data structures
    - Easier to share (pass a pointer)
    - On context switch, only need to save and restore what needs to change
    - User-level library need only update portion of the state
    - Solves the scalability, overhead, performance, and flexibility issues

Solaris 2.0 Threading Model

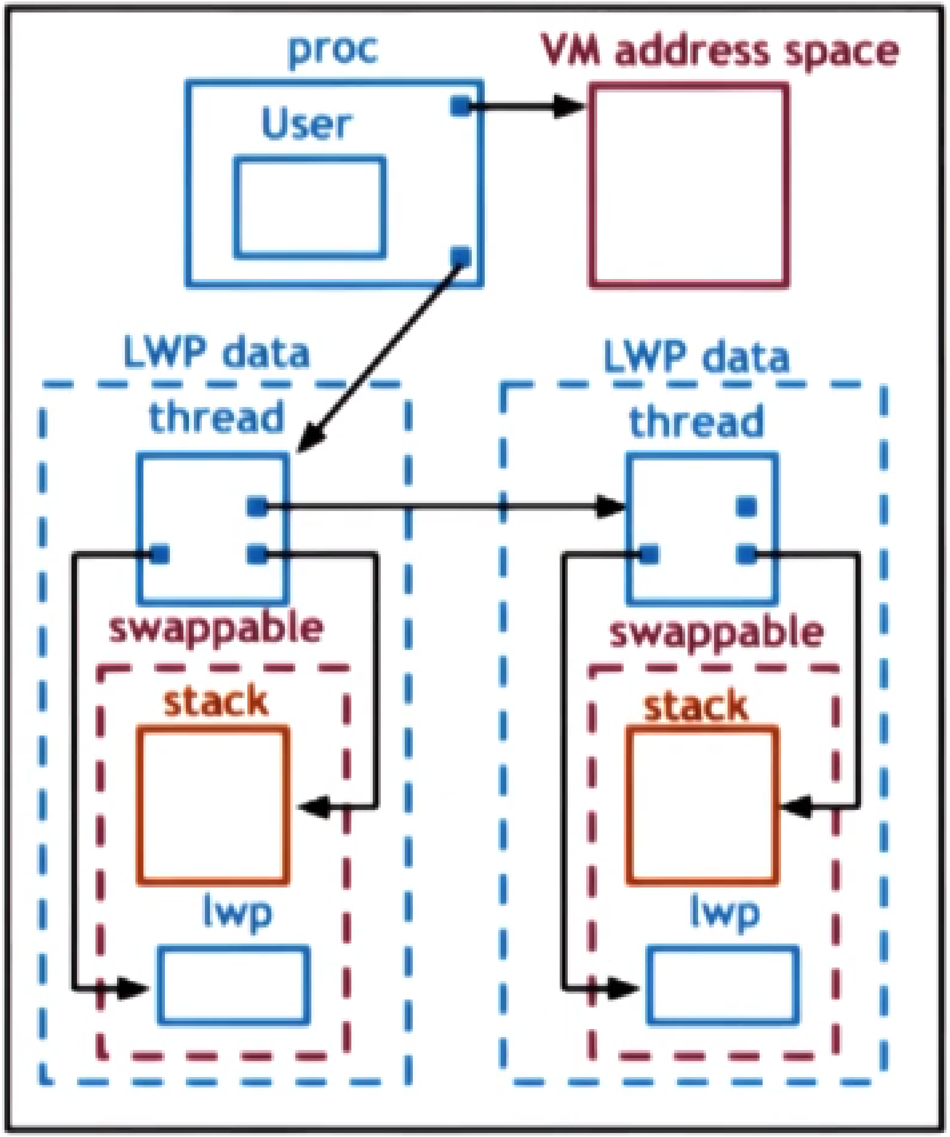
1. "Beyond Multiprocessing: Multithreading the Sun OS Kernel" by Eykholt
2. "Implementing Lightweight Threads" by Stein and Shah
    - Similar to POSIX threads
    - Thread creation -> Thread ID (tid)
    - Table of pointers to thread data structures
    - Stack growth can be dangerous
        - SunOS implemented a "red zone" between blocks of thread memory
3. Kernel-Level Data Structures
    - Process
        - List of kernel-level threads
        - Virtual Address Space
        - User credentials
        - Signal handlers
    - Light-weight Process (LWP)
        - User-level registers
        - System call arguments
        - Resource usage information
        - Signal mask
        - Similar to user-level thread, but visible to kernel
        - Not needed when process not running
    - Kernel-level Threads
        - Kernel-level registers
        - Stack pointer
        - Scheduling information (class, CPU, ...)
        - Pointers to associated LWP, process, CPU structures
        - Information needed even when process isn't running (not swappable)
    - CPU
        - Current thread
        - List of other kernel-level threads
        - Dispatching and interrupt handling information



Solaris 2.0 Threading Model



Solaris 2.0 User-Level Data Structures



Solaris 2.0 Kernel-Level Data Structures

Basic Thread Management Interactions

1. Kernel doesn't know what's happening in user-level library
2. User-level library doesn't know what's happening in the kernel
3. Need some way to communicate between (i.e., LWPs are blocking)
4. System calls and special signals allow kernel/ULT library to coordinate

Thread Management Visibility and Design

1. User-level Library sees:
    - User-level threads
    - Available kernel-level threads
2. Kernel sees:
    - Kernel-level threads

    - CPUs
    - Kernel-level scheduler
3. Can “bind” a user-level thread to a kernel-level thread
4. Process jumps to user-level scheduler when:
    - User-level threads explicitly yield
    - Timer set by user-level library expires
    - User-level threads call library functions like lock/unlock
    - Blocked threads become runnable
    - Signals from timer or kernel
5. Scheduling becomes more complex between multiple ULTs and KLTs on many CPUs
6. For short critical sections, it’s faster for a thread to spin than block (adaptive mutex)
7. Destroying threads
    - Faster to reuse threads than destroy if possible
    - When a thread exits…
        - Put on “death row”
        - Periodically destroyed by reaper thread
        - Otherwise thread structures/stacks are reused (performance gains)

Interrupts and Signals

1. Interrupt - Events generated externally by components other than CPU
    - IO devices, timers, other CPUs
    - Based on the physical platform
    - Appear asynchronously
2. Signals - Events triggered by CPU and software running on it
    - Determined based on the operating system
    - Appear synchronously or asynchronously
3. Both:
    - Have a unique ID depending on hardware or OS
    - Can be masked and disabled/suspended via corresponding mask
        - Per-CPU interrupt mask, per-process signal mask
    - If enabled, trigger corresponding handler
        - Interrupt handler set for entire system by OS
        - Signal handlers set on per process basis, by process
    - Handled in specific ways
        - Interrupt and signal handlers
    - Can be ignored
        - Interrupt/signal mask
    - Expected or unexpected
        - Appear synchronously or asynchronously

Interrupt Handling

1. MSI - Message/Signal Interrupt sent over connection (PCI-E)
2. Interrupts are defined by hardware, handling is defined by OS
3. When an interrupt occurs, if it’s enabled, look up the handler in a table
    - Then, jump to the interrupt service routine (ISR)

Signal Handling

1. Signals are defined by OS, handling is defined by process
2. Handlers/Actions - Terminate, ignore, terminate and core dump, stop, continue
3. Process installs handler - signal(), sigaction()
    - For most signals, some cannot be “caught”

4. Synchronous signals
    - SIGSEGV (access to protected memory)
    - SIGFPE (divide by zero)
    - SIGKILL (kill, id) - Can be directed to a specific thread
5. Asynchronous signals
    - SIGKILL (kill)
    - SIGALARM

Disabling Interrupts and Signals

1. Don't want to get interrupted during critical section (mutex held)
    - Can cause deadlock if handler needs same mutex
2. Solutions:
    - Keep handler code simple (no mutexes)
        - Too restrictive
    - Control interruptions by handler code
        - Use interrupt/signal masks
        - 0 -> disabled, 1 -> enabled
        - If disabled, interrupt/signal will wait for it to enable
3. Interrupt masks are maintained on a per-CPU basis
    - If mask disables interrupt, hardware routing mechanism will not deliver it
4. Signal masks are maintained on a per-execution context (ULT on top of KLT)
    - If mask disables signal, kernel sees and will not interrupt thread
5. On multicore systems, can set interrupt on a per-core basis
    - Avoids overheads and perturbations on all other cores
6. Types of Signals
    - One-shot signals - "n signals pending == 1 signal pending" (at least once)
        - Must be explicitly re-enabled
    - Real Time signals - "n signals raised == n handler calls"
7. Signal Examples
    - SIGINT - Terminal interrupt signal
    - SIGURG - High bandwidth data is available on a socket
    - SIGTTOU - Background process attempting write
    - SIGXFSZ - File size limit exceeded

Interrupts as Threads

1. Can handle interrupts in a separate thread, but dynamic thread creation is expensive
2. Instead, make a dynamic decision
    - If handler doesn't lock -> Execute on interrupted thread's stack
    - If handler can block -> Turn into real thread
3. Optimization - Precreate and preinitialize thread structures for interrupt routines
4. Top half - Fast, non-blocking, minimum amount of processing (when interrupt occurs)
5. Bottom half - Arbitrary complexity (can be scheduled for a later time)
6. This is all motivated by performance
    - Overall cost - overhead of 40 SPARC instructions per interrupt
    - Saving of 12 instructions per mutex
        - No changes in interrupt mask, level, ...
    - Fewer interrupts than mutex lock/unlock operations
    - Optimize for the common case

Threads and Signal Handling

1. Need to define a policy for a ULT to tell a KLT that a signal is masked

2. If ULT mask == 1 and KLT mask == 1
    - No issue, signal handling occurs
3. If ULT1 mask == 0 and KLT1 mask == 1, ULT2 mask == 1
    - Library has a signal handler wrapper that intercepts all signals
    - Threading library sends signals to appropriate threads when received
        - ULT1 won't receive but ULT2 will
4. If ULT1 mask == 0 and KLT1 mask == 1, ULT2 mask == 1 and KLT2 mask == 1
    - Signal originates in KLT1, thread_lib knows ULT1 ignores but ULT2 doesn't
    - thread_lib generates directed signal to KLT2 for ULT2
    - KLT2 sends signal to ULT2
5. IF ULT1 mask == 0 and KLT1 mask == 1, ULT2 mask == 0 and KLT2 mask == 1
    - After sending signal, KLT1 will be changed to 0 (thread_lib ignores)
    - The OS will go to KLT2 since it's mask == 1 and try again
    - The process repeats until all KLT mask == 0
6. This handling optimizes the common case
    - Signals are less frequent than signal mask updates
    - System calls avoided - Cheaper to update UL mask
    - Signal handling is more expensive

Tasks in Linux

1. Main execution abstraction is a task
    - Essentially a kernel-level thread
    - Single threaded process has one task
    - Multithreaded process has many tasks
2. Task creation: Clone
    - clone(function, stack_ptr, sharing_flags, arg)
    - fork is implemented as clone with sharing_flags cleared
3. Native POSIX Threads Library (NPTL) "1:1 model"
    - Kernel sees each ULT info
    - Kernel traps are cheaper
    - More resources: memory, large range of IDs
    - Will still break down with very large number of threads (exascale computing)
    - Older LinuxThreads was a many-to-many model
        - Similar issues to those described in Solaris papers

```
struct task_struct {
    // ...
    pid_t pid;
    pid_t tgid;
    int prio;
    volatile long state;
    struct mm_struct *mm;
    struct files_struct *files;
    struct list_head tasks;
    int on_cpu;
    cpumask_t cpus_allowed;
    // ...
}
```

Task Struct in Linux