Threads and Lightweight Processes

- Processes do not allow concurrency with other processes in common address space
- Traditional processes cannot take advantage of multiprocessor architectures; processes exist in separate address space and have to communicate with each other via shared memory and other synchronization methods
- Threads remove such limitations

Motivation
- Multiple instantiation of various programs such as database servers
- Process forks for each request
- I/O operations provide concurrency benefits
- `fork(2)` is an expensive system call, even with copy-on-write techniques
- Processes have to communicate via shared memory or message passing, with inherent overhead for these techniques
- Processes cannot share some resources such as network connections between different processes
- Thread abstraction
  * Computational unit that is part of overall processing work of application
  * Few interactions with each other and hence, low synchronization requirements
- Traditional Unix process is single threaded

- Multiple threads and processors
  - True parallelism can be achieved by running each thread on a different processor
  - Threads can be multiplexed if their number exceeds the number of available processors
  - Multithreaded processes have to be concerned with every object in their address space
  - There must be inter-thread synchronization to avoid corruption of data
  - With multiple processors, it complicates the issue even further

Concurrency and parallelism
- Parallelism
  * Number of processes actually running in parallel
  * Limited by the number of physical processors
- Concurrency
  * Maximum number of processes simultaneously possible with unlimited number of processors
  * Depends on the way the application is written
  * Possible at user or system level
- System concurrency
  * Provided by kernel by recognizing multiple threads of control
  * *Hot threads* within a process
  * Scheduled independently by the kernel
- User concurrency
  * Provided by the application through user-level thread libraries
  * *Cold threads*, or coroutines
  * Not recognized by the kernel
  * Scheduled and managed by the applications themselves
  * No true concurrency
Threads and Lightweight Processes

- Kernel threads allow parallel execution on multiprocessors but are not suitable for structuring user applications
- Dual concurrency model
  * Combines system and user concurrency
  * Kernel recognizes multiple threads in a process
  * Libraries add user threads not seen by the kernel
  * User threads can provide for synchronization between routines without the overhead of system calls

Fundamental abstractions

- Process divided into a set of threads and a set of resources
- Thread
  - Dynamic object to represent a control point in the process
  - Executes a sequence of instructions
  - Resources include address space, open files, user credentials, and such, and are shared by all threads in the process
  - Each thread has private objects, such as program counter, stack, and register context
  - Drawbacks of centralizing resource ownership in a process
    * Multithreading a server with suid privileges
    * Security is checked by single-threading all system calls
- Kernel threads
  - Need not be associated with a user process
  - Created and destroyed internally by the kernel
  - Shares kernel text and global data, and has its own kernel stack
  - Can be independently scheduled by kernel
  - Useful for operations such as asynchronous I/O
    * Request can be synchronously handled by the kernel thread
  - Inexpensive to create and use
    * Require space only for kernel stack and register context
    * Fast context switching as no memory mappings are to be flushed
- Lightweight processes
  - Kernel supported user thread
  - Requires kernel thread support by the system
  - Independently scheduled but shares the address space and other resources in the process
  - Can make system calls and block for I/O or resources
  - In addition to kernel stack and register context, needs to maintain some user state
    * Register context
  - Useful for independent tasks with little interaction with other lightweight processes
  - User code is pre-emptible and all LWPs in a process share a common address space
    * Concurrent access to critical data must be synchronized
    * Kernel provides facilities to lock shared variables and to block an LWP from accessing shared data
Threads and Lightweight Processes

- LWP operations – creation, destruction, synchronization – require system calls, making LWPs expensive

- Consider busy-waiting instead of blocking for resources held for a brief period of time, as blocking a thread requires kernel involvement and is expensive

- Each LWP consumes significant kernel resources (physical memory for kernel stack)
  * Not practical to support a large number of LWPs
  * LWPs are scheduled by kernel – applications transferring control from one thread to another cannot do so efficiently
  * User can monopolize CPU by creating a large number of LWPs

- User threads
  - Thread abstraction entirely at the user level, with no kernel involvement
  - Extremely lightweight, and consume no kernel resources
  - Accomplished through library packages, such as pthreads
  - Thread operations are entirely performed by the library
  - No kernel involvement, and hence, extremely fast operations
  - Multiplexing user threads on top of LWPs gives a powerful programming environment
  - Library acts as a miniature kernel for the threads it controls
  - User-level context of a thread is saved without kernel intervention
  - Kernel retains responsibility for process switching
    * Preemption of a process preempts all its user threads
    * If a user thread makes a blocking system call, it blocks the underlying LWP
    * If a process had only one LWP, all its threads are blocked
  - Library provides synchronization objects for shared data structures
    * Semaphore and a queue of threads blocked on it
  - Critical thread size
    * Number of instructions to be useful as a separate entity
    * A few hundred instructions
  - Limitations of user threads
    * Total separation of information between kernel and thread library
    * No inter-thread protection mechanism from kernel
    * Kernel may preempt a higher-priority user thread to schedule an LWP running a low-priority user thread
    * Without kernel support, user threads may improve concurrency but do not increase parallelism
      - User threads within an LWP do not execute in parallel even on a multiprocessor

Lightweight process design

- System calls
  - Need to preserve semantics of a single-threaded Unix environment
  - Multithreaded case should behave in a reasonable manner to approximate single-threaded semantics

- Semantics of fork(2)
  - Creates a child process which is almost an exact clone of parent
In multithreaded case, we have the option to duplicate all LWPs of the parent or only the one that invoked the fork

Case 1: Copy only the calling LWP of the parent
* More efficient
* Better if child immediately execs
* Problem: User process may contain references to other LWPs
* Child process must not try to acquire locks held by threads that do not exist in child (deadlock?)

Case 2: Copy all LWPs of parent
* Useful when entire process is to be cloned
* What if an LWP in the parent is blocked on a system call
  - Undefined state in child
  - Can return the status code EINTR (system call interrupted)
* An LWP may have open connections
  - Closing connections can send unexpected messages to remote host

Situation can be resolved by offering two variants of fork, to handle the above two cases

Other system calls

What if an LWP closes a file being used by another
What about file pointer being moved by two different LWPs
Dynamic memory allocation
These calls should be made thread safe

Signal delivery and handling

Signals are delivered to and handled by processes
Which LWP should handle the signals?
Kernel delivers the signal to an LWP; thread library directs it to a specific thread
How to handle signals?
1. Send it to each thread
   * Highly expensive
   * Useful when entire set of threads is to be sent a message, such as SIGABORT
   * SIGSTP and SIGINT are generated by external events and cannot be associated with any thread
2. Specify a master thread for all signals
   * Asymmetric treatment of threads
   * Not compatible with SMP approach
3. Send it to any arbitrarily chosen thread
4. Use heuristics to determine the thread for signal
   * SIGSEGV and SIGILL are caused by thread and should be delivered accordingly
5. Create a new thread to handle each signal
   * Only applicable in certain situations
   - Should all threads share a common set of signal handlers?

Stack growth
Stack overflow causes a SIGSEGV
Kernel sees the signal originating from stack and automatically extends the stack instead of signaling the process
Multithreaded process has one stack for each user thread, allocated at the user level by thread library
Threads and Lightweight Processes

* Incorrect for the kernel to extend stack
* Stack is to be handled by user thread library
  - In multithreaded systems, kernel has no knowledge of user stacks
  * **SIGSEGV** is sent by kernel to appropriate thread who will be responsible

User-level thread libraries

- Design issues: API and implementation
- Programming interface
  - Operations to be provided
    * Creation and termination of threads
    * Suspending and resuming threads
    * Priority assignment
    * Scheduling and context switching
    * Synchronization
    * Messaging
  - Minimize kernel involvement to avoid the overhead of mode switching
  - Kernel may not have knowledge of user threads
  - Thread library may use system calls to implement kernel functionality
    * Kernel priority and thread priority are independent
    * Thread priority is used by thread scheduler
- Implementing thread libraries
  - Acts as a miniature kernel, performing thread maintenance and scheduling at user level
  - Concurrency is provided by using asynchronous I/O facilities
  - Choice of implementation under LWPs
    * Bind each thread to a different LWP
      - Easy to implement but uses kernel overhead and does not offer added value
      - Kernel involvement in thread synchronization and scheduling
    * Multiplex user threads on a set of LWPs
      - More efficient, consumes fewer kernel resources
      - Works better if threads in a processes are roughly equivalent
      - Does not guarantee resources to a particular thread
    * Allow a mixture of bound and unbound threads in same process
      - Application can exploit concurrency and parallelism
      - Preferential treatment of bound threads by increasing priority of underlying LWPs, or by giving an LWP exclusive control of a processor
  - Thread library
    * Contains scheduling algorithm, may multiplex multiple threads on different processors
    * Maintains per-thread state and priority
    * Different threads could be in state **running** or **blocked**
      - Thread can enter a blocked state when it attempts to acquire a synchronization object held by another thread
      - Library unblocks the thread when the object is released
-- Mechanism is similar to kernel’s resource wait and scheduling algorithms

Scheduler activations

- User threads are not as efficient as the lwp's due to lack of kernel-level integration
- New architectures for user libraries tend to have closer integration between kernel and user threads
  - Kernel is responsible for processor allocation
  - Thread library provides scheduling
    - Thread library informs kernel of events affecting processor allocation
    - Library may request additional processors or give up processors
    - Kernel controls processor allocation and may randomly preempt a processor and allocate it to another process
    - Library has complete control over which threads to be scheduled on processors
    - If kernel takes away a processor, it informs the library which reallocates the threads
    - If a thread blocks inside the kernel, kernel informs the library which schedules another thread on the processor
- New abstractions to support the above
  - upcall
    - Call made by kernel to thread library
  - scheduler activation
    - Execution context used to run a user thread
    - Similar to an lwp and has its own kernel and user stacks
    - Upcall passes an activation to library to be used to process the event, run a new thread, or invoke a system call
    - Kernel does not time slice activations on a processor
    - At any time, a process has exactly one activation for each process
- Handling blocking operation in scheduler activation framework
  - When a thread blocks in kernel, kernel creates a new activation and upcalls to the library
  - Library saves the thread state from old activation and informs the kernel that it can reuse the old activation
  - Library then schedules another thread on the new activation
  - When blocking is complete, kernel makes another upcall to library to inform about the event, requiring a new activation
  - Kernel may assign a new processor to run this new activation, or preempt one of the current activations of the process
  - In the second case, kernel has to make two upcalls to inform about the two threads (preempted and scheduled)
  - Library puts both threads on ready list and then decides the one to schedule
- Advantages of scheduler activation
  - Extremely fast as the operations do not require kernel intervention
  - Kernel informs library of blocking and preemption; library can make better scheduling and synchronization decisions, and avoid deadlocks and incorrect semantics

Multithreading in Solaris and SVR4
Threads and Lightweight Processes

- Solaris supports kernel threads, lightweight processes, and user threads
  - User process may have several hundred threads
  - Thread library multiplexes the threads onto a small number of LWPs
  - User can control the number of LWPs and can also bind threads to individual LWPs

- Kernel threads
  - Lightweight objects that can be independently scheduled and dispatched
  - Need not be associated with any process
  - May be created, run, and destroyed by the kernel
  - Kernel does not have to remap the virtual address space to switch between threads
  - Kernel thread uses a small data structure and a stack
    * Saved copy of kernel registers
    * Priority and scheduling information
    * Pointer to put thread on scheduler queue or resource wait queue
    * Pointer to the stack
    * Pointer to associated lwp and proc structures, or NULL if thread is not bound to an LWP
    * Pointers to maintain a queue of all threads in a process and a queue of all threads in the system
    * Information about the associated LWP
  - Kernel is organized as a set of fully preemptible kernel threads
    * Synchronization primitives prevent priority inversion where a low-priority thread locks a resource needed by a high-priority thread
    * Used to handle asynchronous activity, such as deferred disk writes

- Lightweight process implementation
  - Each LWP bound to its own kernel thread for its lifetime
  - proc and u must be modified for per-process and per-LWP information
    * Solaris puts all per-process data in proc, including the process-specific part of u
  - LWP part of context is kept in an lwp structure
    * Saved values of user-level registers
    * System call arguments, results, and error code
    * Signal handling information
    * Resource usage and profiling data
    * Virtual time alarms
    * User time and CPU usage
    * Pointer to kernel thread
    * Pointer to proc
  - lwp is swapped out with the LWP
    * Information, such as signal masks, must be kept in associated thread structure
    * Solaris on Sparc reserves the global register %g7 to hold a pointer to current thread
    * All LWPs share a common set of signal handlers, but can have their own signal masks
      * Traps are always delivered to the LWP that generated it
      * Interrupts can be delivered to any LWP that has not masked the signal
    * LWPs have no global name space and are invisible to other processes
      * A process cannot directly communicate with a specific LWP in another process
Synchronization of lwp is achieved through mutex locks, condition variables, counting semaphores, and reader-writer locks

- User threads
  - Implemented by a threads library
  - Managed without invoking the kernel
  - Synchronization and scheduling is provided by threads library
  - Thread library hides the communication between user threads and lwp
    - Library multiplexes a number of threads on lwp
    - Application may specify the number of lwp to be created
    - Threads can be bound to an lwp or can be unbound in which case they share the common lwp pool
  - Number of lwp determines the maximum possible parallelism

- User thread implementation
  - State information maintained by each thread
    - Thread id
      - Allows threads within a process via signals
    - Saved register state
      - Program counter and stack pointer
    - User stack
      - Allocated by the library
      - Not visible to kernel
    - Signal mask
      - Used by library to route signals to appropriate threads
    - Priority
      - Used by thread scheduler
      - Not visible to kernel
    - Thread local storage
      - Private storage for supporting reentrant versions of C library interfaces
  - Solaris allows threads in different processes to synchronize using shared memory

- Interrupt handling
  - Interrupt handlers manipulate data shared by kernel
    - Kernel must synchronize access to shared data
    - Achieved in traditional systems by raising the interrupt priority level to block relevant interrupts
    - Raising interrupt level is expensive
    - Problem magnified in multiprocessor environments
      - Kernel has to block interrupts on multiple processors
    - Solaris implementation
      - Not dependent on priority levels
      - Uses different kernel synchronization objects such as mutex locks and semaphores
      - Interrupts are handled by a set of kernel threads, called interrupt threads
      - Interrupt threads are created dynamically and are assigned a higher priority than any other thread
      - Use same synchronization primitives as other threads and can block themselves on resources held by other threads
      - Kernel blocks interrupts in a few exceptional situations only
Threads and Lightweight Processes

- Kernel maintains a pool of preallocated and partially initialized interrupt threads
- One thread per interrupt level plus a single systemwide thread for clock
- Uses about 8Kbytes per thread, and that calls for reduction of pool on systems with scarce memory
  - Implementing interrupt handlers as threads adds overhead but avoids having to block interrupts for each synchronization object
  - Synchronization is more common than interrupts leading to performance improvement

- Handling system calls in Solaris
  - `fork(2)` duplicates each LWP of parent in the child
  - LWPs in the middle of a system call return with EINTR error
  - A new system call `fork1(2)` is similar to `fork(2)` but only duplicates the thread that invoked it
    - Use `fork1(2)` if child is to `exec` immediately

- A good way to create applications is to develop them using user threads and later optimize by manipulating the underlying LWPs to best provide the real concurrency