A Modern Process (1)

- Stack
- Global Vars.
- Instructions

- SP1
- SP2
- SP3

- Thread Library

- Lightweight Processes
- User Space

- Kernel Threads
- Kernel Stack, Memory Map, File Descriptors, etc.

A Modern Process (2)

- Separate idea of execution from resource grouping
- Multithreading Support
  - One or more threads of control (Program Counters)
  - One stack for each thread of control
- Thread
  - A unit of local dispatching (scheduling) and has priority
  - Has a set of CPU registers
  - Mapped to a lightweight process (LWP)
- LWPs are mapped to processors and globally scheduled
- Global variables are shared by all threads
- System state (file descriptors, working directory, etc.) shared by threads

Thread Execution (1 Processor)

- Single-Threaded Processes
  - Process A
  - Process B
  - Unix Kernel

- Multithreaded Processes
  - Can overlap own I/O with own CPU usage
  - Process A
    - Thread 0
    - Thread 1
  - Process B
    - Thread 0
    - Unix Kernel
Database Server Example

- Each worker thread
  - Waits for its turn to read task list
  - Starts disk read; waits if entry is not in database cache
  - Sends reply when database entry is in database cache

**Thread Library Implementations**

- **User-Space**
  - Self-contained user-level library
  - All code and structures are in user-space
  - Depends on a small number of OS system calls

- **Kernel-Space**
  - Thin user-space layer
  - Substantial amount of kernel code and structures

**N:1 Model of Multithreading (1)**

- Many threads mapped onto ONE process
- Implementation
  - Put thread package entirely in user space
    - Thread creation, scheduling, and synchronization all done in user space
    - Kernel has no knowledge of threads
  - Thread Table
    - Analogous to process table, but contains only thread state
  - Dispatcher
    - An ordinary function called during startup: calls main()

  ```
  Choose thread to run:
  if (context switch) Load new hardware state;
  Resume selected thread execution (load CP);
  ```

**N:1 Model of Multithreading (2)**

- Implementation (cont)
  - Non-Blocking I/O Wrapper
    ```
    while (iorequest(...) is incomplete) {
        Update thread table (I/O wait; thread state); Jump to dispatcher;
        // Return here when dispatcher returns control
    }
    ```

- Advantages
  - Coroutine style control flow
  - Fast, but no speed-up on a multiprocessor
  - Scheduling done by user (within context of process)

- Disadvantages
  - Non-preemptive scheduling within a process
1:1 Model of Multithreading

**Features**
- Many threads can run simultaneously on different CPUs
- Allows 1 or more threads to issue blocking system calls while others run (even on a uniprocessor)
- Thread creation requires LWP creation (and a system call)
- Each LWP takes up kernel resources → Limited total number of threads

Lightweight Processes (LWPs)

**Kernel Thread**
- Created/Destroyed by OS kernel
- Has own kernel stack but shares text and globals
- Used for kernel operations (e.g., I/O, paging daemon)

**Lightweight Process (LWP)**
- User thread with kernel support
- Shares address space with other LWPs of same process
- Maintains some of user state (register context, ...), kernel stack, and kernel register context
- Scheduled by kernel
- Most LWP operations (create/destroy, synchronize) require system call → High overhead

M:N Model of Multithreading

**Strict M:N (M ≥ N)**
- Thread creation, scheduling, and some synchronization done in user space

**M:N + 1:1**
- Combines the best of M:N and 1:1
- Used in Solaris, IRIX, HP-UX
- Win32 fibers is a rough approximation

LWP Design (1)

**Semantics of fork**
- Option 1: Duplicate all LWPs of parent
  - If LWP in the parent is blocked on a system call
  - LWP state in child is undefined (although could mark as interrupted)
- Option 2: Duplicate only the calling LWP of the parent
  - More efficient than Option 1
    - Especially, if child immediately exits
  - References to other LWPs should not be used since they don’t exist in child
    - Deadlock if child tries to acquire a lock held by a thread that does not exist
- Most OSes allow user to choose option
  - e.g., Solaris fork() and fork1()

**Other System Calls**
- Many must be modified to work correctly
  - i.e., made *thread safe* (synchronize shared access)
  - e.g., File pointers, dynamic memory allocation, working directory
LWP Design (2)

- **Signal Delivery and Handling**
  - Signals are delivered to and handled by processes
  - But need to choose which LWP and which user thread
  - Kernel delivers signal to LWP, and thread library directs it to a thread
  - **Option 1:** Send signal to each thread
    - Very expensive but may make sense in case of SIGABORT
  - **Option 2:** Send to a "master" thread
    - Asymmetric and not compatible with symmetric multiprocessing approach
  - **Option 3:** Create new thread to handle signal
    - May be reasonable in some cases
  - **Option 4:** Send signal to applicable thread (determined by heuristic)
    - SIGSEGV (segmentation violation) and SIGILL (illegal exception) should be sent to violating thread
    - SIGTSTP (stop signal from terminal) has no clear recipient

- **Stack Growth**
  - Classic processes automatically grow the stack after a SIGSEGV (segmentation violation) from a stack overflow
  - Threaded processes have one fixed-size stack per thread → Thread is responsible for handling stack overflow
  - Allocate 1 extra write-protected page for each thread stack which will page fault on stack overflow

Threads Standards

- Defines an API and behavior of a threads paradigm
  - About 50 function calls
- **POSIX Threads**
  - IEEE 1003.1c (Pthreads)
  - Portable (Implementations on almost all Unix Systems)
  - Not adopted by Microsoft
- **Win32 and OS/2 Threads**
  - Not compatible with Pthreads
  - Proprietary (vendor-specific)
- **Solaris Threads (UI Threads)**
  - Used in Solaris 2 and developed before Pthreads standard was finalized
  - Virtually the same as Pthreads

POSIX Synchronization Primitives

- Each synchronization facility has a named data structure called a **synchronization variable**
- **Counting Semaphores**
  - Typically used to coordinate access to shared variable
- **Mutual Exclusion (mutex) Locks**
  - Used to serialize the execution of code
- **Condition Variables**
  - Enables threads to atomically block until a condition is satisfied
- **Multiple Readers, Single Writer Locks**
  - Allows many threads to have simultaneous read-only access to data while allowing only one thread to have write access at any given time

Examples of pthreads Functions (1)

- **Thread Creation/Termination**
  - `int pthread_create(pthread_t * T, pthread_attr_t *Attr, void *(*start)(void *), void *arg);
  - `void pthread_exit(void * ret);
  - `int pthread_join(pthread_t T, void **ret);
- **Mutex Lock**
  - `int pthread_mutex_lock(pthread_mutex_t *M)
  - `int pthread_mutex_trylock(pthread_mutex_t *M)
  - `int pthread_mutex_unlock(pthread_mutex_t *M)
  - `int pthread_mutex_init(pthread_mutex_t *M, const pthread_mutexattr_t *Attr))
Examples of pthreads Functions (2)

- **Condition Variable**
  - `int pthread_cond_wait(pthread_cond_t *Cv, pthread_mutex_t *M)`
  - `int pthread_cond_signal(pthread_cond_t *Cv)`
  - `int pthread_cond_init(pthread_cond_t *Cv, const pthread_condattr_t *Attr)`

- **Semaphores**
  - `int sem_wait(sem_t *S)`
  - `int sem_post(sem_t *S)`
  - `int sem_init(sem_t *S, int isShared, unsigned int V)`

 Mutex Lock Implementation

```c
pthread_mutex_lock(L) {
    while (TestAndSet(L)) { // someone has lock
        Put thread on wait queue for L;
        Suspend thread;
    }
    return;
}

pthread_mutex_unlock(L) {
    Unsuspend next thread in wait queue for L;
    L = 0;
    return;
}
```

- **Variation**
  - Spin for a short time instead of suspending in hopes of short blocking time

Spin Locks

- Blocking on a mutex lock will cause two context switches (switch out, switch in)
  - 150 usec on SC2000/Solaris 2.4
  - 25 usec on 300 MHz Pentium II/NetBSD
  - 35 usec on 167 MHz SPARC 5/Solaris 2.5

- A spin lock can be used to avoid the context switching, but wastes CPU time

```c
while (pthread_mutex_trylock(&mylock) == EBUSY)
    // ... Do Nothing ... ;
... Critical Section ...
pthread_mutex_unlock(&mylock);
```

Advantage Over Semaphore

- Uses little memory and is fast

<table>
<thead>
<tr>
<th>Type of Synchronization</th>
<th>Time (usec) on 20-Proc. 40 MHz SPARC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unbound Semaphore</td>
<td>292.0</td>
</tr>
<tr>
<td>Bound Semaphore</td>
<td>326.0</td>
</tr>
<tr>
<td>Unbound Mutex</td>
<td>2.1</td>
</tr>
<tr>
<td>Bound Mutex</td>
<td>2.3</td>
</tr>
</tbody>
</table>
Condition Variables (1)

**Use**
- Wait until a condition is satisfied without busy waiting
- Must be used in conjunction with a mutex lock

**Primitives**
- `int pthread_cond_wait(pthread_cond_t *Cv, pthread_mutex_t *M)`
  - Block until condition is signaled
  - Atomically release mutex lock before blocking and atomically reacquire it before returning
- `int pthread_cond_signal(pthread_cond_t *Cv)`
  - Unblock one thread waiting for the condition
  - No thread blocked on Cv ➔ No Effect
  - Call under protection of mutex associated with Cv
  - Retest condition after thread becomes unblocked

Condition Variables (2)

**Primitives (cont)**
- `int pthread_cond_init(pthread_cond_t *Cv, const pthread_condattr_t *Attr)`
  - Return 0 if successful; non-zero otherwise
  - Not needed if mutex variable is statically allocated

**Implementation**

```c
pthread_cond_wait(V, L) {
    do {
        Put thread on wait queue for V;
        L = 0;
        Suspend this thread;
    } while (TestAndSet(L) == 0) // until get lock
    return;
}
```

```c
pthread_cond_signal(V, L) {
    Unsuspend next thread in wait queue for V; return;
}
```

CVs Are Stateless Signals

**Example**
- Another thread on the mylock queue changes n

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>mutex_lock(&amp;mylock);</td>
<td>mutex_lock(&amp;mylock);</td>
</tr>
<tr>
<td>if (n == 0)</td>
<td>cond_signal(&amp;done, &amp;mylock); cond_wait(&amp;done, &amp;mylock);</td>
</tr>
<tr>
<td>mutex_unlock(&amp;mylock);</td>
<td>mutex_unlock(&amp;mylock);</td>
</tr>
</tbody>
</table>

**Guarantee the condition has not changed**
- Some other thread can change n before Thread 1 gets control
- n is not protected from change (unlike semaphore counter)

Thread Scheduling

**Local Scheduling (Process Contention Scope)**
- Scheduling done by the threads library
  - Very fast except for preemption (requires system call)
- Scheduling of LWP is global, but is independent of local scheduling
- Scheduling is by thread priority
  - Set by programmer; not adjusted by threads library

**Global Scheduling (System Contention Scope)**
- Scheduling done by OS kernel
- Thread blocks ➔ LWP goes to sleep