A Modern Process (1)

- Separate idea of execution from resource grouping
- Thread
  - A unit of local dispatching (scheduling) and has priority
  - Has an execution path (is a thread of control)
  - Has a computation state (stack, set of CPU registers)
- Global variables are shared by all threads
- System state shared by threads
  - File descriptors, working directory, etc.

A Modern Process (2)

A Modern Process (3)
**Database Server Example**

- **Benefits**
  - Simplifies sharing of memory and file descriptors
  - Concurrent execution of relatively independent tasks
  - Increase overall throughput for some problems

**Thread Execution (1 Processor)**

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

- **Multithreaded Processes**
  - Can overlap own I/O with own CPU usage

**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
  - N:1 model
    - N user threads mapped to 1 kernel thread or process

- **Kernel-Space**
  - Thin user-space layer
  - Substantial amount of kernel code and structures
  - 1:1 model and N:M model
    - 1 user thread mapped to 1 kernel thread
    - or N user threads mapped to M kernel threads

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

- Many threads mapped onto ONE process
- Implementation
  - Put thread package entirely in user space
  - Thread creation-scheduling-synchronization done in user space
  - Allocate stack for each thread
  - 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()
  - Use setjmp(3)/longjmp(3) in place of function call/return

Choose thread to run:
if (context switch) Load new hardware state;
Reseue selected thread execution (load PC);

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
  - One process and threads are unknown to OS kernel
  - Scheduling done by user-thread package (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
    - Each LWP is associated with a unique kernel thread
  - 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
### N:M Model of Multithreading

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

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

### Resources Used

- **Kernel Thread**
  - Copy of kernel registers
  - Priority and scheduling
  - Ptrs to scheduler queue or resource wait queues for each thread
  - Ptrs to LWP and proc structure (if any)
  - Ptrs to list of all threads in a process and all threads in system
  - LWP info

- **LWP**
  - Copy of user-level registers
  - System call args, results, error code
  - Signal handling info
  - Resource usage and profiling info
  - Virtual time alarms
  - Ptr to kernel thread
  - Ptr to proc structure

### 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;
}
```

- **Mutexes**
  - Simple enough to implement entirely in user space
- **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
  - NOT used for mutual exclusion, but ...
  - 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 Variable Example

- Wait for W threads to finish

```c
pthread_mutex_t doneLock;
pthread_cond_t doneCv;

//Main thread:
pthread_mutex_lock(&doneLock);
if (nDone < W) pthread_cond_wait(&doneCv, &doneLock);
pthread_mutex_unlock(&doneLock);

// Other threads
pthread_mutex_lock(&doneLock);
nDone++;
if (nDone == 0) pthread_cond_signal(&doneCv);
pthread_mutex_unlock(&doneLock);
```

CVs are Stateless Signals (1)

- Abbreviations
  - `Csig` `pthread_cond_signal`
  - `Cwait` `pthread_cond_wait`
  - `Lock` `pthread_mutex_lock`
  - `Unlock` `pthread_mutex_unlock`
  - `Set(x)` `Lock(L); cond = x; Unlock(L);`

- Case A (OK)

```
[ Thread 1 ]
[ Thread 2 ]

Csig(x);
Lock(L);
Cwait(x, L);
Unlock(L);
```

Time
CVs are Stateless Signals (2)

**Case B (Lost Signal Problem)**

- [Thread 1] Csig(X); LOST!
- [Thread 2] Lock(L);
- Cwait(X, L);
- Unlock(L);

**Case B’ (Solve Lost Signal Problem)**

- [Thread 1] Lock(L);
- cond = 1; Csig(X); LOST!
- Unlock(L);
- [Thread 2] Lock(L);
- while (!cond) Cwait(X, L);
- Unlock(L);

CVs are Stateless Signals (3)

**Case B” (Alternative Solution)**

- [Thread 1] Set(1); Csig(X); LOST!
- [Thread 2] Lock(L);
- while (!cond) Cwait(X, L);
- Unlock(L);

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