Lecture 8

Remote Procedure Calls

Introduction to Multithreaded Programming

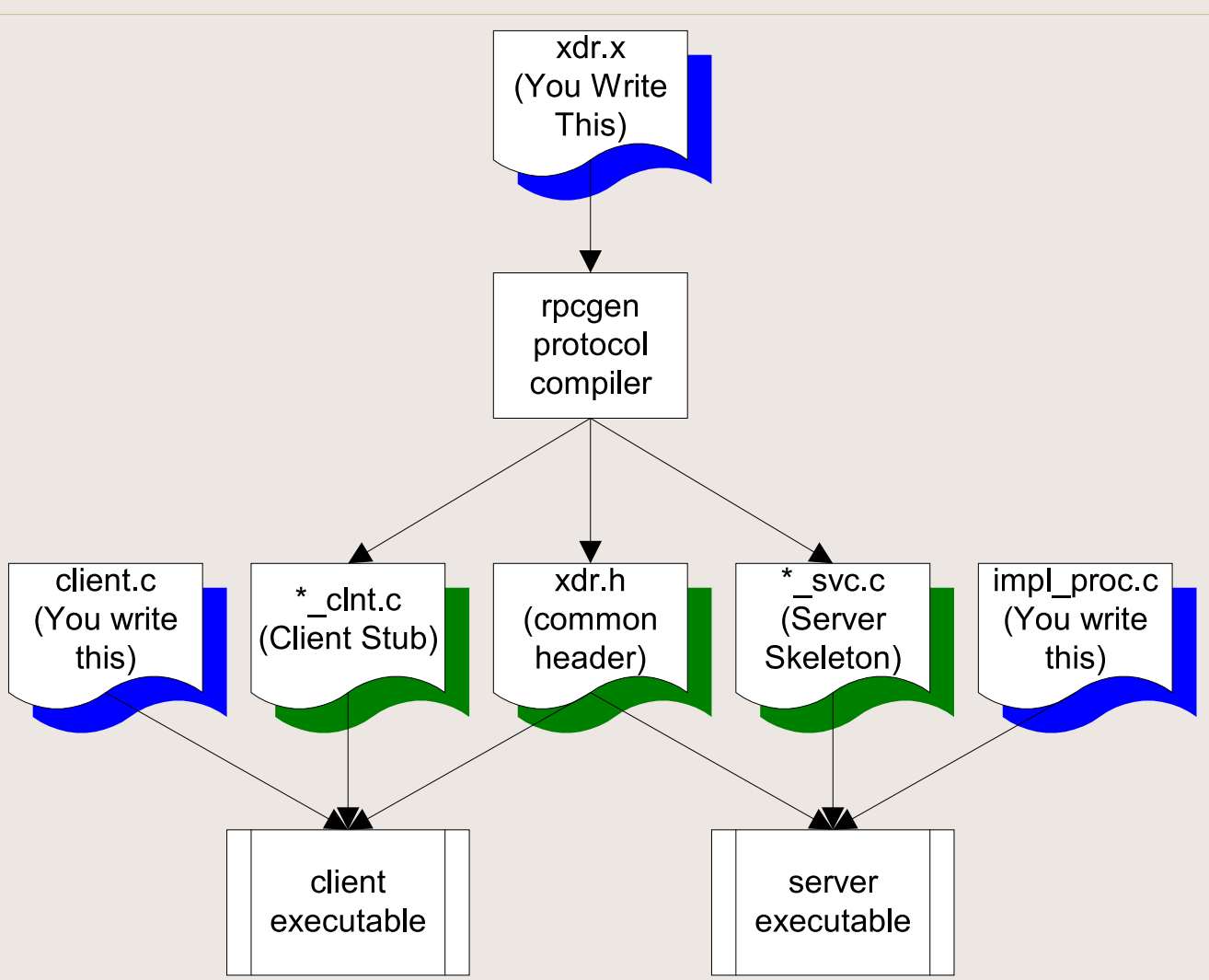
Remote Procedure Calls

Digital ONC RPC

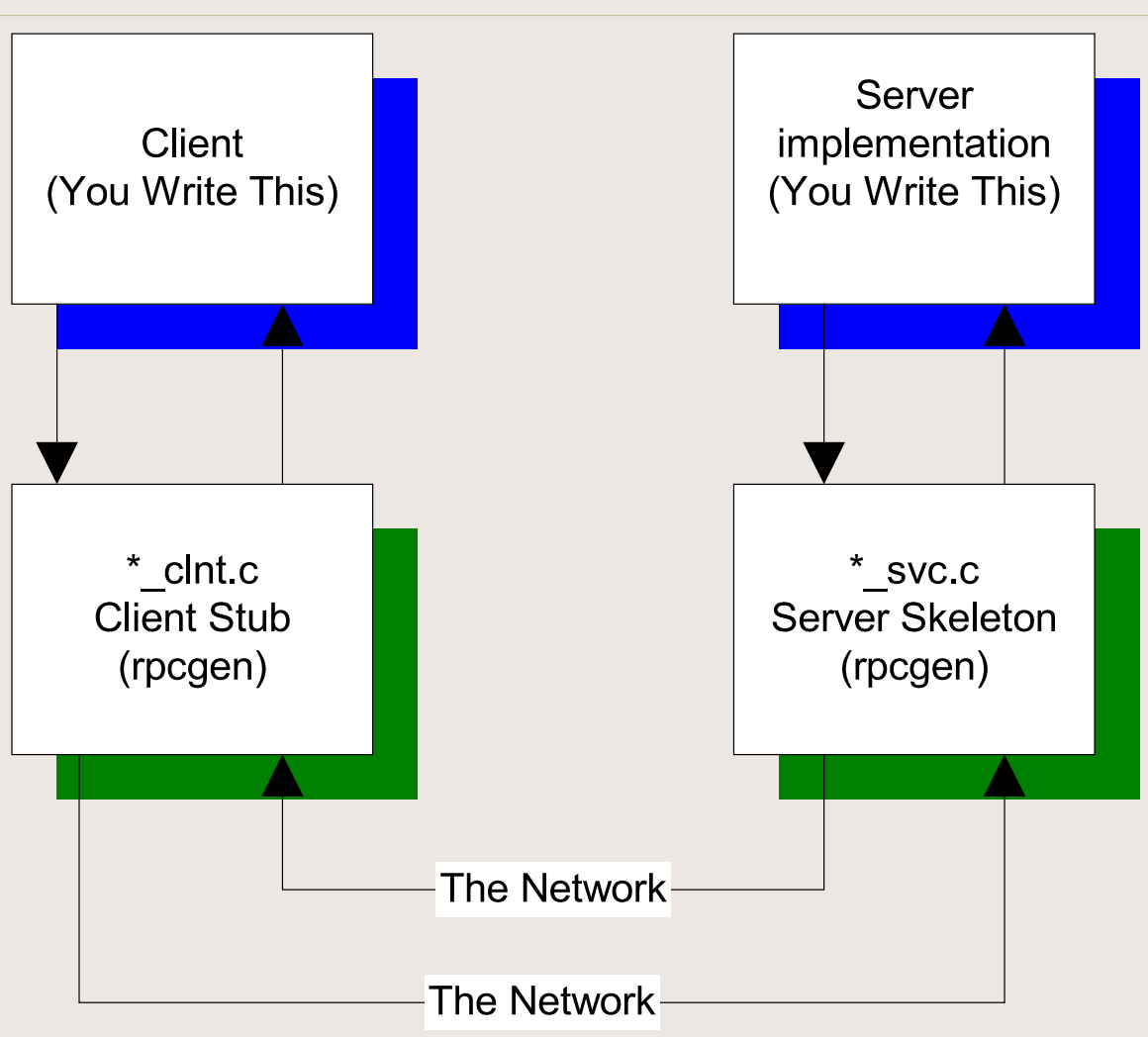
The Point

- "What's the difference between local and remote procedure calling?"
  - "Very little—that's the point"
- Remote Procedures generally accept and return *pointers* to data

The Process


xdr.x
(You Write This)
rpcgen protocol compiler
client.c
(You write this)
*_clnt.c
(Client Stub)
xdr.h
(common header)
*_svc.c
(Server Skeleton)
impl_proc.c
(You write this)
client executable
server executable

Call Sequence



Remote Services

- SUN Remote Procedure Call
  - If the time to transfer the data is more than the time to execute a remote command, the latter is generally preferable.
  - UDP protocol is used to initiate a remote procedure, and the results of the computation are returned.

SUN RPC

- Communication is message-based
- When a server starts, it binds an arbitrary port and publishes that port and the PROGRAM and VERSION with the portmapper daemon (port 111)
- When a client starts, it contacts the portmapper and asks where it can find the remote procedure, using PROGRAM and VERSION ids. The portmapper daemon returns the address and client and server communicate directly.

Sample protocol definition file (.x file)

this XDR file (somefile.x):

```
program NUMPROG
{
   version NUMVERS
   {
      int READNUM(int) = 1; /* version 1 */
   } = 1; /* version of functions */
} = 0x2000002; /* PROGRAM number */
```

is turned into this header file by rpcgen (somefile.h):

```
#define NUMPROG 0x2000002
#define NUMVERS 1

#if defined(__STDC__) || defined(__cplusplus)
#define READNUM 1
extern  int * readnum_1(int *, CLIENT *);
extern  int * readnum_1_svc(int *, struct svc_req *);
```

RPC Paradigms for Client Server

- Fat Client-DBMS (2 Tier)
  - VB <=> Sybase (ODBC)
  - Motif C++ <=> DBMS (ctlib)
- Fat Client-Application Server-DBMS
  - C Front End <=> C Business Logic <=> DBMS

RPC Under the Hood

- RPC is important because it handles network details for you:
  - Network Details
    - Byte Ordering (Big Endian, Little Endian)
  - Alignment Details
    - 2/4 Byte alignment
  - String Termination (NULL ?)
  - Pointers (how to handle migration of pointers?)

RPC eXternal Data Representation

- XDR provides:
  - Network Transparency
    - Single Canonical Form using Big-Endian
    - 4-Byte alignment
    - XDR passes all data across the wire in a byte stream
  - Filters

XDR Filters

- Integer: int (4 bytes)
- Unsigned Integer: unsigned int (4 bytes)
- char: int (4 byte signed integer)
- Double: double (8 bytes IEEE754 FP)
- Float: float (4 bytes IEEE754 FP)
- int week[7]
- int orders <50> (variable length array)
- opaque data<1000> any data

Building an RPC Application

- Create XDR file
  (~mark/pub/518/rpc/[linux|sun]/numdisp.x)
- run rpcgen to create
  - client stub: numdisp_clnt.c
  - server skeleton: numdisp_svc.c
  - common header: numdisp.h
- write client.c and numdisp_proc.c
- compile client and server (in subdirs)
- run (client on devon, server on orcus)
- *example: ~mark/pub/518/rpc/linux, ~mark/pub/518/rpc/sun*

Introduction to Multithreaded Programming with POSIX Pthreads

Pthreads Information

Threads FAQ

Pthread Tutorial at Amherst

Pthreads Programming Bouncepoint

Processes Revisited

- A process is an active runtime environment that cradles a running program, providing an execution state along with certain resources, including file handles and registers, along with:
  - a program counter (Instruction Pointer)
  - a process id, a process group id, etc.
  - a process stack
  - one or more data segments
  - a heap for dynamic memory allocation
  - a process state (running, ready, waiting, etc.)
- Informally, a process is *an executing program*

Multiprocessing Revisited

- A multiprocessing or multitasking operating system (like Unix, as opposed to DOS) can have more than one process executing at any given time
- This simultaneous execution may either be
  - concurrent, meaning that multiple processes in a run state can be swapped in and out by the OS
  - parallel, meaning that multiple processes are actually running *at the same time* on multiple processors

What is a Thread?

- A thread is an encapsulation of some flow of control in a program, that can be *independently* scheduled
- Each process is given a single thread by default
- A thread is sometimes called a *lightweight* process, because it is similar to a process in that it has its own thread id, stack, stack pointer, a signal mask, program counter, registers, etc.
- All threads within a given process *share* resource handles, memory segments (heap and data segments), and code. THEREFORE HEAR THIS:
  - *All threads share the same data segments and code segments*

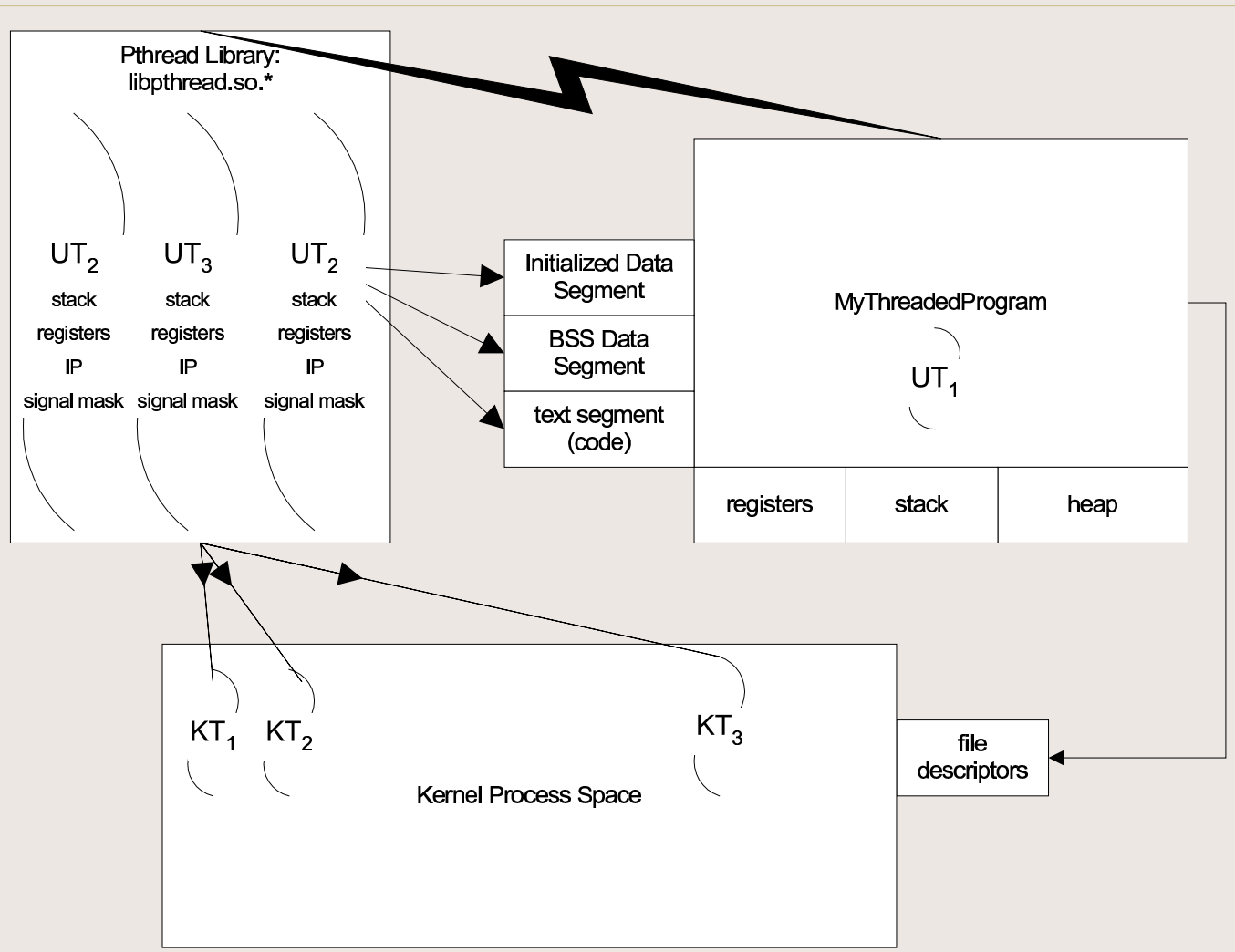
What's POSIX Got To Do With It?

- Each OS had it's own thread library and style
- That made writing multithreaded programs difficult because:
  - you had to learn a new API with each new OS
  - you had to *modify your code* with each port to a new OS
- POSIX (IEEE 1003.1c-1995) provided a standard known as Pthreads
- DCE threads were based on an early 4th draft of the POSIX Pthreads standard (immature)
- Unix International (UI) threads (Solaris threads) are available on Solaris (which also supports POSIX threads)

Once Again....

A PROCESS	A THREAD
Process ID	*Thread* ID
Program Counter	Program Counter
Signal Dispatch Table	Signal Dispatch Table
Registers	Registers
Process Priority	*Thread Priority*
Stack Pointer & Stack	Stack Pointer & Stack
Heap	***All threads share the same memory, heap, and file handles (and offsets)***
Memory Map	
File Descriptor Table	

The Big Kahuna



Processes and Threads: Creation Times

- Because threads are by definition *lightweight*, they can be created more quickly that "heavy" processes:
  - Sun Ultra5, 320 Meg Ram, 1 CPU
    - 94 forks()/second
    - 1,737 threads/second (18x faster)
  - Sun Sparc Ultra 1, 256 Meg Ram , 1 CPU
    - 67 forks()/second
    - 1,359 threads/second (20x faster)
  - Sun Enterprise 420R, 5 Gig Ram, 4 CPUs
    - 146 forks()/second
    - 35,640 threads/second (244x faster)
  - Linux 2.4 Kernel, .5 Gig Ram, 2 CPUs
    - 1,811 forks()/second
    - 227,611 threads/second (125x faster)

Say What?

- Threads can be created and managed more quickly than processes because:
  - Threads have less overhead than processes, for example, threads *share* the process heap, all data and code segments
  - Threads can live entirely in *user* space, so that no kernel mode switch needs to be made to create a new thread
  - Processes don't need to be swapped to create a thread

Analogies

- Just as a multitasking operating system can have multiple processes executing concurrently or in parallel, so a single process can have multiple threads that are executing concurrently or in parallel
- These multiple threads can be taskswapped by a scheduler onto a single processor (via a LWP), or can run in parallel on separate processors

Benefits of Multithreading

- Performance gains
  - Amdahl's Law: speedup = $1 / (1 - p) + (p/n)$
  - the speedup generated from parallelizing code is the time executing the parallelizable work (p) divided by the number of processors (n) plus 1 minus the parallelizable work (1-p)
  - The more code that can run in parallel, the faster the overall program will run
  - If you can apply multiple processors for 75% of your program's execution time, and you're running on a dual processor box:
    - 1 / ((1 - .75) + (.75 / 2)) = 60% improvement
  - Why is it not strictly linear? How do you calculate p?

Benefits of Multithreading (continued)

- Increased *throughput*
- Increased application *responsiveness* (no more hourglasses)
- Replacing interprocess communications (you're in *one* process)
- Single binary executable runs on *both* multiprocessors as well as single processors (processor transparency)
- Gains can be seen even on single processor machines, because blocking calls no longer have to *stop* you.

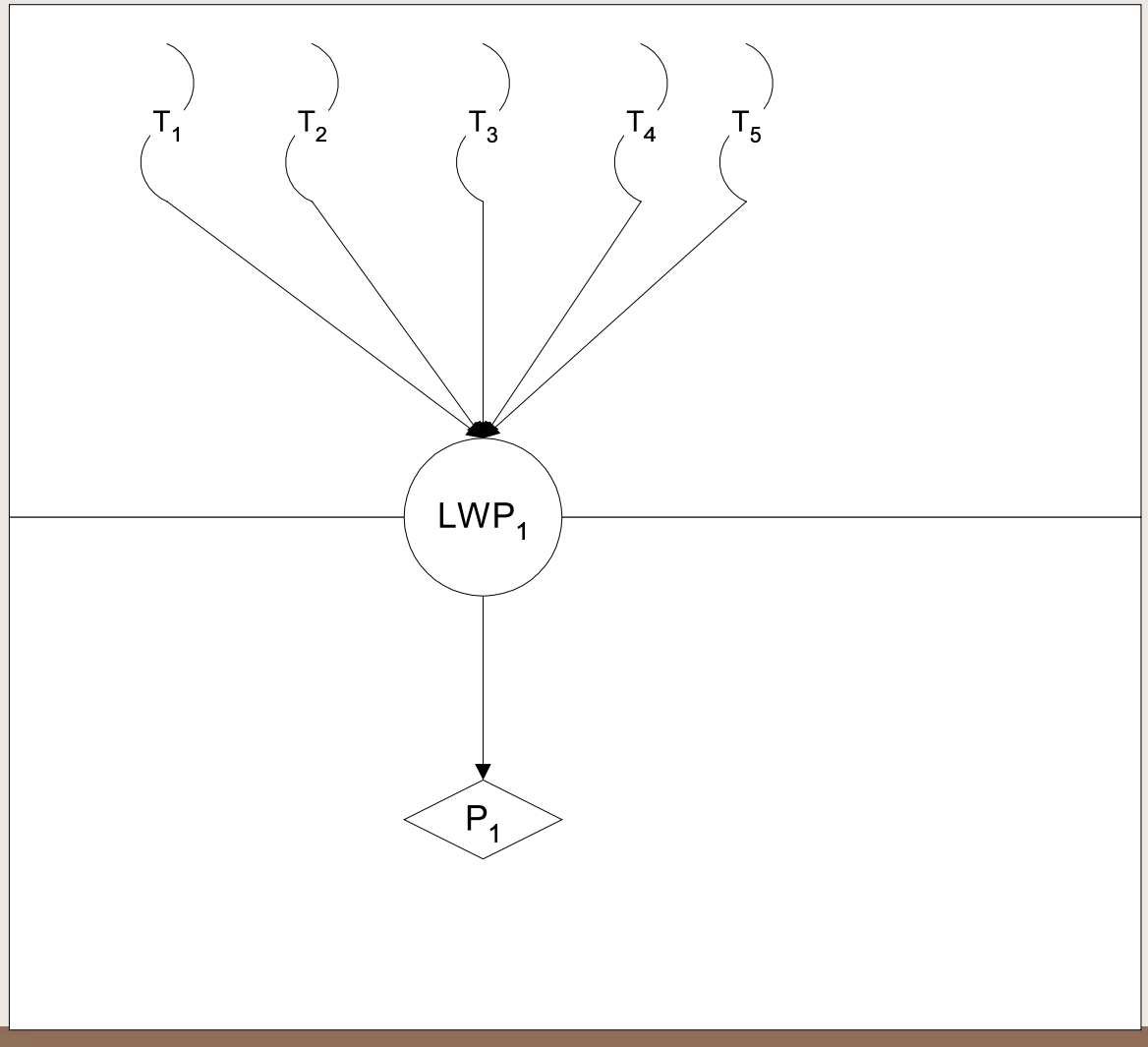
On the Scheduling of Threads

- Threads may be scheduled by the system scheduler (OS) or by a scheduler in the thread library (depending on the threading model).
- The scheduler in the thread library:
  - will preempt currently running threads on the basis of priority
  - does *NOT* time-slice (i.e., is not fair). A running thread will *continue to run forever* unless:
    - a thread call is made into the thread library
    - a blocking call is made
    - the running thread calls sched_yield()

Models

- Many Threads to One LWP
  - DCE threads on HPUX 10.20
- One Thread to One LWP
  - Windows NT
  - Linux (clone() function)
- Many Threads to Many LWPs
  - Solaris, Digital UNIX, IRIX, HPUX 11.0)

Many Threads to One LWP
DCE threads on HPUX 10.20



USER SPACE

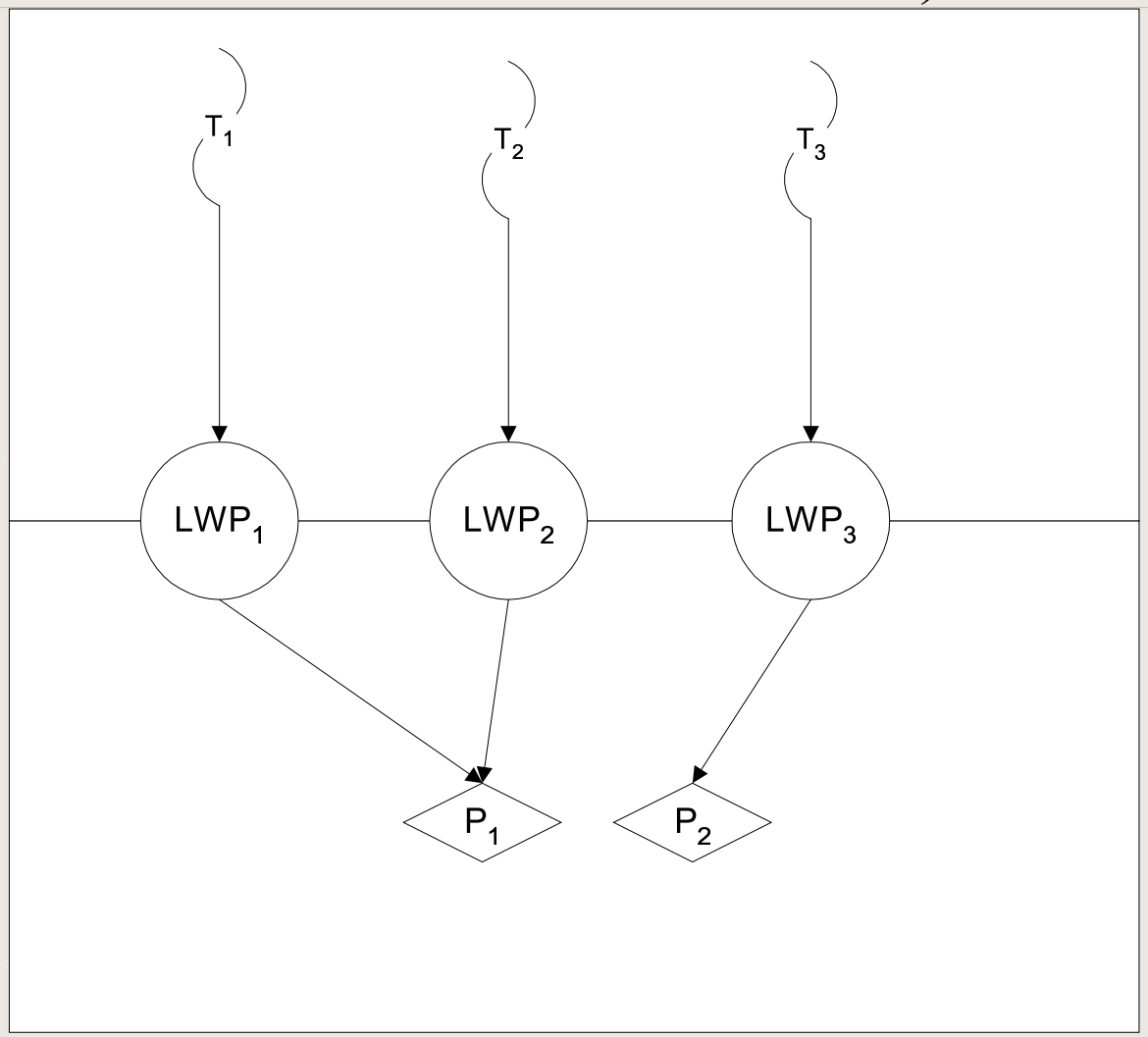
AKA "user space threads". All threads are "invisible" to the kernel (therefore cannot be schdeduled individually by the kernel). Since there's only a single LWP (kernel-scheduled entity), user space threads are multiplexed onto a single processor. The kernel sees this process as "single threaded" because it only sees a single LWP.

KERNEL SPACE

Mx1 Variances

- very fast context switches between threads is executed entirely in user space by the threads library
- unlimited number of user threads (memory limit) can support logical concurrency model only
- parallelism is not possible, because all user threads map to a single kernel-schedulable entity (LWP), which can only be mapped on to a single processor
- Since the kernel sees only a *single* process, when one user space thread blocks, the *entire* process is blocked, effectively block all other user threads in the process as well

One Thread to One LWP(Windows NT, Linux) (there may be no real distinction between a thread and LWP)



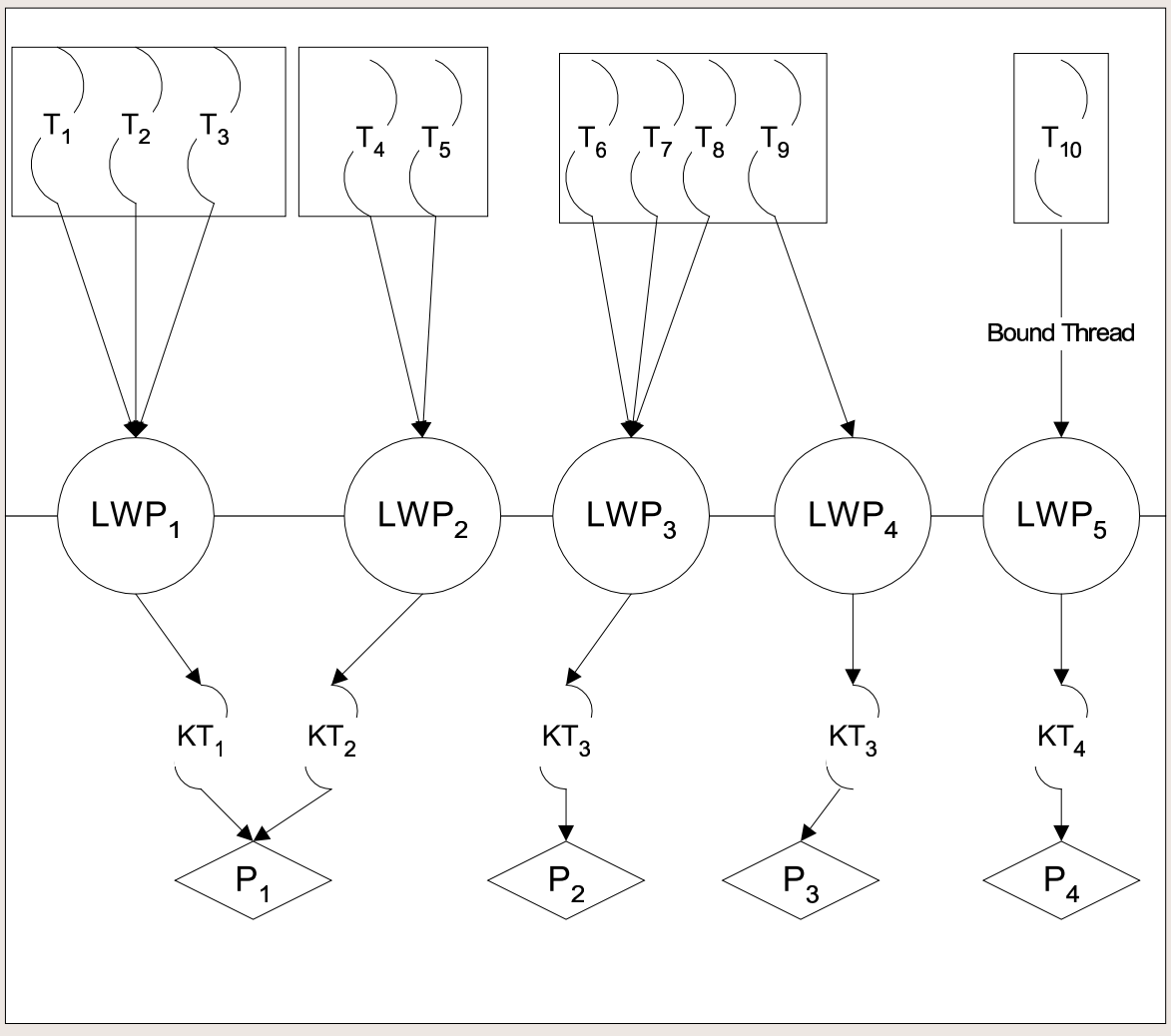
USER SPACE
Each user space thread is associated with a single kernel thread to which it is permanently bound. Because each user thread is essentially a kernel-schedulable entity, parallel execution is supported.

KERNEL SPACE
The 1x1 model executes in kernel space, and is sometimes called the Kernel Threads model. The kernel selects kernel threads to run, and each process may have one or more threads

1x1 Model Variances

- Parallel execution is supported, as each user thread is directly associated with a single kernel thread which is scheduled by the OS scheduler
- slower context switches, as kernel is involved
- number of threads is *limited* because each user thread is directly associated with a single kernel thread (in some instances threads take up an entry in the process table)
- scheduling of threads is handled by the OS's scheduler, threads are seldom starved
- Because threads are essentially kernel entities, swapping involves the kernel and is less efficient than a pure user-space scheduler

Many Threads to Many LWPs
Solaris, Digital UNIX, IRIX, HPUX 11.0



MxN Model Variances

- Extraordinarily flexible, bound threads can be used to handle important events, like a mouse handler
- Parallel execution is fully supported
- Implemented in both user and kernel space
- Slower context switches, as kernel is often involved
- Number of user threads is virtually unlimited (by available memory)
- Scheduling of threads is handled by both the kernel scheduler (for LWPs) and a user space scheduler (for user threads). User threads can be starved as the thread library's scheduler does not preempt threads of equal priority (not RR)
- The kernel sees LWPs. It does NOT see threads

Creating a POSIX Thread: pthread_create()

```
#include <pthread.h>
void * pthread_create(pthread_t *thread, const
  pthread_attr_t attr, void *(*thrfunc)(void *),
  void *args);
```

- Each thread is represented by an identifier, of type pthread_t
- Code is encapsulated in a thread by creating a thread function (cf. “signal handlers”)
- Attributes may be set on a thread (priority, etc.). Can be set to NULL.
- An argument may be passed to the thread function as a void **

Detaching a Thread

```
int pthread_detach(pthread_t threadid);
```

- Detach a thread when you want to inform the operating system that the threads return result is unneeded
- Detaching a thread tells the system that the thread (*including* its resources—like a 1Meg default stack on Solaris!) is no longer being used, and can be recycled
- A detached thread's thread ID is *undetermined*.
- Threads are detached after a pthread_detach() call, after a pthread_join() call, and if a thread terminates and the PTHREAD_CREATE_DETACHED attribute was set on creation

“Wating” on a Thread: pthread_join()

```
int pthread_join(pthread_t thread,
  void** retval);
```

- pthread_join() is a blocking call on non-detached threads
- It indicates that the caller wishes to block until the thread being joined exits
- You cannot join on a detached thread, only non-detached threads (detaching means you are NOT interested in knowing about the threads exit)

Exiting from a Thread Function

```
int pthread_exit(void * retval);
```

- A thread ends when it returns from (falls out of) its thread function encapsulation
- A detached thread that ends will immediately relinquish its resources to the OS
- A non-detached thread that exists will release some resources but the thread id and exit status will hang around in a *zombie-like* state until some other thread requests its exit status via pthread_join()

Miscellaneous Functions

```
pthread_t pthread_self(void);
```

- pthread_self() returns the *currently executing* thread's ID

```
int sched_yield(void);
```

- sched_yield() politely informs the thread scheduler that your thread will willingly release the processor if any thread of equal or lower priority is waiting

```
int pthread_setconcurrency(int threads);
```

- pthread_setconcurrency() allows the process to *request* a fixed minimum number of light weight processes to be allocated for the process. This can, in some architectures, allow for more efficient scheduling of threads

Managing Dependencies and Protecting Critical Sections

- Mutexes
- Condition Variables
- Reader/Writer Locks
- Semaphores
- Barriers

Mutexes

- A Mutex (*Mut*ual *Ex*clusion) is a data element that allows multiple threads to synchronize their access to shared resources
- Like a binary semaphore, a mutex has two states, *locked* and *unlocked*
- Only one thread can lock a mutex
- Once a mutex is locked, other threads will *block* when they try to lock the same mutex, until the locking mutex unlocks the mutex, at which point one of the waiting thread's lock will succeed, and the process begins again

Statically Initialized Mutexes

- Declare and statically initialize a mutex:

```
pthread_mutex_t mymutex =
  PTHREAD_MUTEX_INITIALIZER;
```

- Then, lock the mutex:

```
pthread_mutex_lock(&mymutex);
```

- Then, unlock the mutex when done:

```
pthread_mutex_unlock(&mymutex);
```

NonStatically Initialized Mutexes

- Declare a mutex:

```
pthread_mutex_t mymutex;
```

- Initialize the mutex:

```
pthread_mutex_init(&mymutex,
   (pthread_mutexattr_t *)NULL );
```

- Lock the mutex

```
pthread_mutex_lock(&mymutex);
```

- Unlock the mutex:

```
pthread_mutex_unlock(&mymutex);
```

Dynamic Mutexes

- Declare a mutex pointer:

```
pthread_mutex_t * mymutex;
```

- Allocate memory for the mutex and pointer.
- *Optionally* declare a mutex attribute and initialize it

```
pthread_mutexattr_t mymutex_attr;
pthread_mutexattr_init(&mymutex_attr);
```

- initialize the mutex:

```
pthread_mutex_init(mymutex,
   &mymutex_attr);
```

- Lock and Unlock the mutex as normal...
- Finally, destroy the mutex

```
pthread_mutex_destroy(mymutex);
```

Condition Variables

- A Condition variable is synchronization mechanism that allows multiple threads to *conditionally* wait, until some defined time at which they can proceed
- Condition variables are different from mutexes because they don't protect *code*, but *procedure*
- A thread will *wait* on a condition variable until the variable signals it can proceed
- Some *other* thread *signals* the condition variable, allowing other threads to *continue*.
- Each condition variable, as a *shareable datum*, is associated with a particular mutex
- Condition Variables are supported on Unix platforms, but not on NT

How Condition Variables Work

1. A thread *locks* a mutex associated with a condition variable
2. The thread tests the condition to see if it can proceed
3. If it can (the condition variable is true):
    1. your thread does its work
    2. your thread unlocks the mutex
4. If it cannot (the condition variable is false)
    1. the thread sleeps by calling cond_wait(&c,&m), and the mutex is automatically released for you
    2. some other thread calls cond_signal(&c) to indicate the condition is true
    3. your thread wakes up from waiting with the mutex *automatically* locked, and it does its work
    4. your thread releases the mutex when it's done

General Details

Thread α		Thread β
T_1:	mutex_lock(&m);	
T_2:	while(! condition_ok)	
T_3:	while(cond_wait(&c,&m);	
T_4:		mutex_lock(&m);
T_5:		condition_ok = TRUE;
T_6:		cond_signal(&c);
T_7:		mutex_unlock(&m);
T_8:	go_ahead_and_do_it();	
T_9:	mutex_unlock(&m);	

Reader/Writer Locks

- Mutexes are powerful synchronization tools, but too broad a use of mutexes can begin to *serialize* a multithreaded application
- Often, a critical section only needs to be protected if multiple threads are going to be *modifying* (writing) the data
- Often, multiple reads can be allowed, but mutexes lock a critical section without regard to reading and writing
- Reader/Writer locks allow multiple threads in for *reading only* and *only one* writer thread in a given critical section

Barriers:
The Ultimate Top Ten Countdown

- Sometimes, you want several threads to work together in a group, and not to proceed past some point in a critical section (the *Barrier*) before *all* threads in the group have arrived at the same point
- A Barrier is created by setting its value to the *number of threads* in the group
- A Barrier can be created that acts as a counter (similar to a counting semaphore), and each thread that arrives at the Barrier *decrements* the Barrier counter and goes to sleep.
- Once all threads have arrived, the Barrier counter is 0, and *all threads* are signaled to awaken and continue
- A Barrier is made up of both a mutex and a condition variable
- Metaphor: A group of people are meeting for dinner at a restaurant. They all wait outside until all have arrived, and then go in.

Synchronization Problems

- Deadlocks
- Race Conditions
- Priority Inversion

Deadlocks (avoid with pthread_mutex_trylock())

- Deadlocks can occur when locks are locked out of order (interactive). *Neither* thread can execute in order to allow the other to continue :

	Thread α	Thread β
T_1:	pthread_mutex_lock(a);	pthread_mutex_lock(b)
T_2:	pthread_mutex_lock(b);	pthread_mutex_lock(a)

- Or when a mutex is locked by the same thread twice (recursive)

Thread α

T_1: pthread_mutex_lock(a);

...

T_n: pthread_mutex_lock(a);

Race Conditions

- Race conditions arise when variable assignment is undetermined, due to potential context swapping or parallelization:

Thread α		Thread β
T_1:	int x = 10;	
T_2:	/* context switch to β*/	
T_3:		x = 7;
T_4:		/* context switch to α */
T_5:	printf(“%d”,x);	

Priority Inversion

- Imagine the following scenario:

1. A low priority thread acquires mutex *m*
2. A medium priority thread preempts the lower priority thread
3. A high priority thread preempts the medium priority thread, and needs to *lock* mutex *m* in order to proceed:

- The mutex lock held by the *sleeping* low-priority thread blocks the high priority thread from *acquiring* the mutex and proceeding!

Inversion Solutions

- Priority Inheritance Protocol for mutexes:
  - any thread inherits the highest priority of all threads that block while holding a given mutex
  - In the previous example, when the high priority thread blocks on the mutex m being held by the low priority thread, the priority of that low priority thread is *bumped up* to the priority of the highest priority thread blocking, thus *increasing its chances for being scheduled*
- Priority Ceiling Protocol Emulation
  - associates a *priority with a mutex*, and this priority is set to *at least* the priority of the highest priority thread that can lock the mutex
  - When a thread locks a mutex, it's priority is raised to the mutex's priority

Threads and Signals

- NB: Signals are not supported on NT
- Under POSIX, a signal is delivered to the *process*, NOT to the thread or LWP
- The signal is of course handled by a thread, and the one chosen to handle it is determined based on it's priority, run-state, and most of all on the threads' *signal masks*.
- For synchronous signals (SIGFPE, SIGILL, etc.), the signal is delivered to the offending thread
- One recommendation (Bil Lewis) is to have *all threads but one* mask all signals, and have a single thread handle *all* asynchronous signals by blocking on a sigwait() call.