Introduction to POSIX Threads

Asynchronous Programming in a Unix/Linux Environment

Class #308

Doug Abbott
Silver City, NM

575-388-4879
email: doug@intellimetrix.us
web: www. Intellimetrix.us

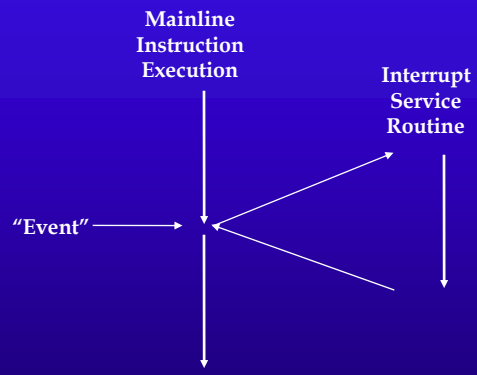
Intellimetrix

Topics

- *Asynchronous* programming
  - What is it and why do you care
- Threads Objects and their APIs
  - Threads
  - Mutexes and Condition variables
- Thread cancellation
- Threads implementations
  - User Space
  - Kernel Space

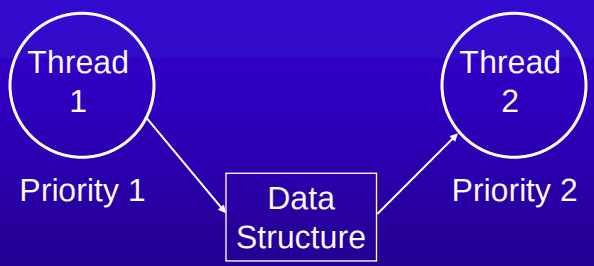
Intellimetrix

The Interrupt



Intellimetrix

Interrupts Cause Problems




```
while (1)
{
   wait for (event1);
   write (data_structure);
}
```

```
while (1)
{
   wait for (event2);
   read (data_structure);
}
```

Intellimetrix

The Real-time Programming Problem

Thread 2

```
while (1)
{
   Reading data structure;
```

INTERRUPT (event 1)!

Thread 1 executes!

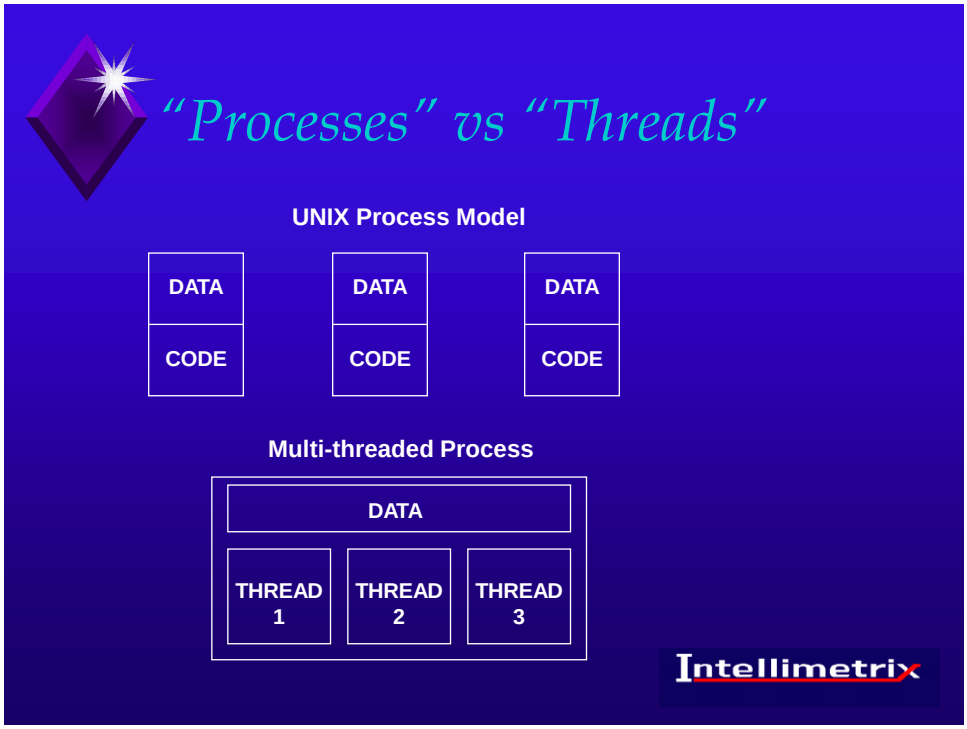
```
   Still reading data structure;
         (but data has changed!)
}
```

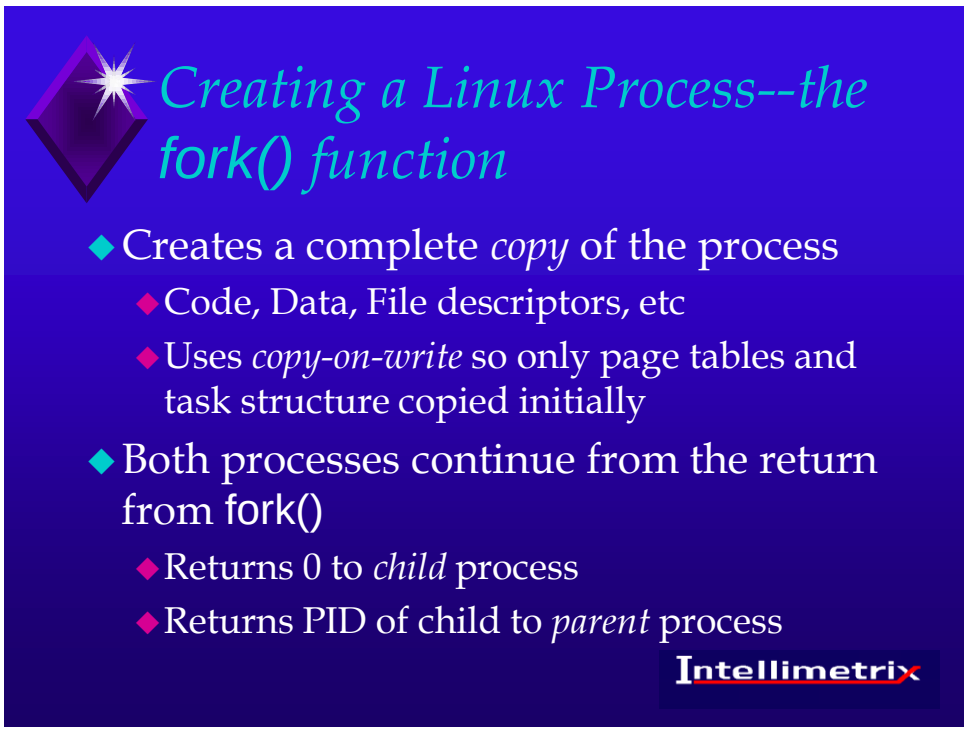
Intellimetrix

Multitasking is...

- A paradigm for handling asynchronous events reliably
- A way to break a large problem into smaller independent parts
- Multiple program parts running in "parallel"

Intellimetrix


"Processes" vs "Threads"
UNIX Process Model
DATA
CODE
DATA
CODE
DATA
CODE
Multi-threaded Process
DATA
THREAD 1
THREAD 2
THREAD 3
Intellimetrix


Creating a Linux Process--the fork() function
Creates a complete copy of the process
Code, Data, File descriptors, etc
Uses copy-on-write so only page tables and task structure copied initially
Both processes continue from the return from fork()
Returns 0 to child process
Returns PID of child to parent process
Intellimetrix

Thread API

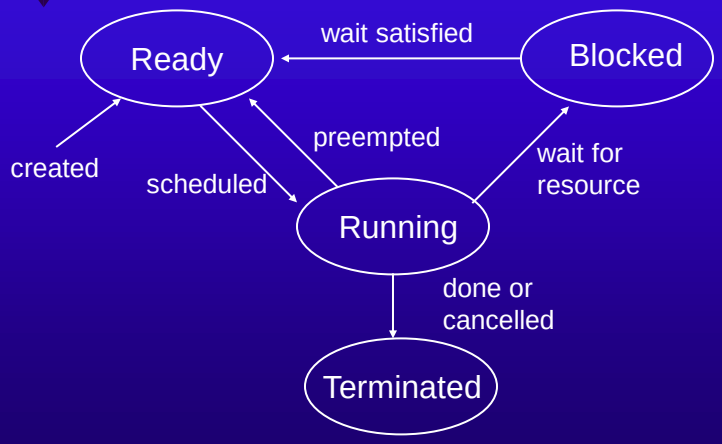
```
int pthread_create (pthread_t *thread, pthread_attr_t *attr,
        void *(* start_ routine) (void *), void *arg);
void pthread_exit (void *retval);
int pthread_join (pthread_t thread, void **thread_return);
pthread_t pthread_self (void);
int sched_yield (void);
```

Intellimetrix

```
/*          Thread Example   */
#include <pthread.h>
#include "errors.h"

/*
 * Thread start routine.
 */
void *thread_routine (void *arg)
{
    printf ("%s\n"), arg);
    return arg;
}

main (int argc, char *argv[])
{
    pthread_t thread_id;
    void *thread_result;
    int status;

    status = pthread_create (&thread_id, NULL, thread_routine, (void *)argv[0]);
    printf ("New thread created\n");

    status = pthread_join (thread_id, thread_result);
    printf ("Thread returned %p\n", thread_result);
    return 0;
}
```

Intellimetrix

Thread State Diagram



Intellimetrix

Thread Termination

- Thread function just returns
- Thread function calls pthread_exit
- Thread is *cancelled* by another thread
- detachstate
  - PTHREAD_CREATE_JOINABLE - Can be "joined" by another thread when it terminates
  - PTHREAD_CREATE_DETACHED - Can't be joined. Resources reclaimed immediately.

Intellimetrix

Attribute Objects

- All pthread objects can have additional, separate attribute objects
- Extended argument list for object creation
- Attribute is second argument to create call
  - if NULL, use defaults
- Objects are "opaque"
  - access with specific attribute functions

Intellimetrix

Thread Attributes

```
int pthread_attr_init (pthread_attr_t *attr);
int pthread_attr_destroy (pthread_attr_t *attr);
int pthread_attr_getdetachstate (pthread_attr_t *attr, int *detachstate);
int pthread_attr_setdetachstate  (pthread_attr_t *attr, int detachstate);

#ifdef _POSIX_THREAD_ATTR_STACKSIZE
int pthread_attr_getstacksize (pthread_attr_t *attr, size_t *stacksize);
int pthread_attr_setstacksize (pthread_attr_t *attr, size_t stacksize);
#endif

#ifdef _POSIX_THREAD_ATTR_STACKADDR
int pthread_attr_getstackaddr (pthread_attr_t *attr, void **stackaddr);
int pthread_attr_setstackaddr (pthread_attr_t *attr, void *stackaddr);
#endif
```

Intellimetrix

```
/*          Thread Attributes       */
#include <limits.h>
#include <pthread.h>

void *thread_routine (void *arg)
{/ * Thread start routine. Reports it ran, and then exits. */
    printf ("The thread is here\n");
    return NULL;
}
int main (int argc, char *argv[])
{
    pthread_t thread_id;
    pthread_attr_t thread_attr;
    size_t stack_size;
    int status;

    status = pthread_attr_init (&thread_attr);
    status = pthread_attr_setdetachstate (&thread_attr, PTHREAD_CREATE_DETACHED);

#ifdef _POSIX_THREAD_ATTR_STACKSIZE
    / *  Set stack size to twice the minimum  */
    status = pthread_attr_getstacksize (&thread_attr, &stack_size);
    printf ("Default stack size is %u; minimum is %u\n", stack_size, PTHREAD_STACK_MIN);
    status = pthread_attr_setstacksize (&thread_attr, PTHREAD_STACK_MIN*2);
#endif
    status = pthread_create (&thread_id, &thread_attr, thread_routine, NULL);
    printf ("Main exiting\n");
    return 0;
}
```

Intellimetrix

Thread Scheduling

Optional Thread Attributes

```
int pthread_attr_setschedparam (pthread_attr_t *attr,
            const struct sched_param *param);
int pthread_attr_getschedparam (const pthread_attr_t *attr,
            struct sched_param *param);
int pthread_attr_setschedpolicy (pthread_attr_t *attr, int policy);
int pthread_attr_getschedpolicy (const pthread_attr_t *attr, int *policy);
int pthread_attr_setinheritsched (pthread_attr_t *attr, int inherit);
int pthread_attr_getinheritsched (const pthread_attr_t *attr, int *inherit);

int pthread_setschedparam (pthread_t pthread, int *policy,
             const struct sched_param *param);
int pthread_getschedparam (pthread_t pthread, int *policy,
            struct sched_param *param);
int sched_get_priority_max (int policy);
int sched_get_priority_min (int policy);
```

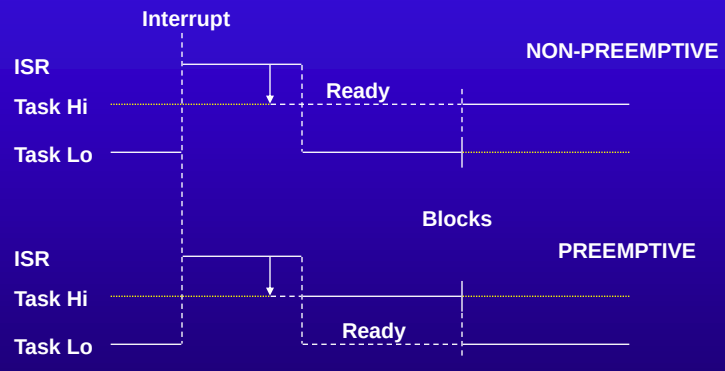
Intellimetrix

Scheduling Policies

- SCHED_FIFO
  - Preemptive scheduler
- SCHED_RR
  - Preemptive scheduler with time slicing
- SCHED_OTHER
  - Default policy
  - Tries to be "fair"

Intellimetrix

Scheduling: Non-Preemptive vs Preemptive



Intellimetrix

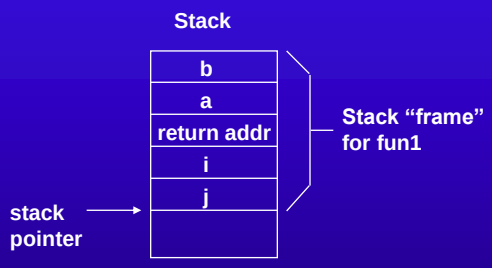
Reentrancy & "Thread-safe" Functions

- A function that may be called by more than one thread must be either *reentrant* or *thread-safe*
- Library functions must be reentrant or thread-safe
- A function is reentrant if it does not access any *statically allocated* resources

Intellimetrix

Reentrancy--What is it

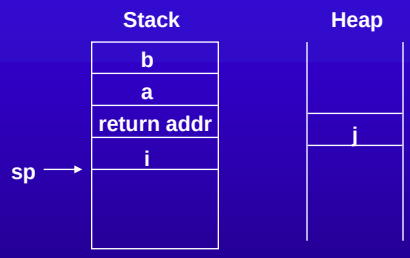
```
void fun1 (int a, int b)
{
        int i, j;

        i = a;
        j = b;
          .
          .
}
```



Intellimetrix

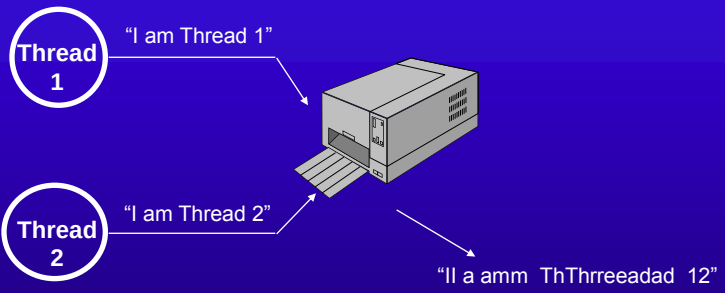
A Non-reentrant Function

```
int j;

void fun2 (int a, int b)
{
        int i;

        i = a;
        j = b;

        if (j == 0)
        {
        }
}
```



Intellimetrix

Sharing Resources

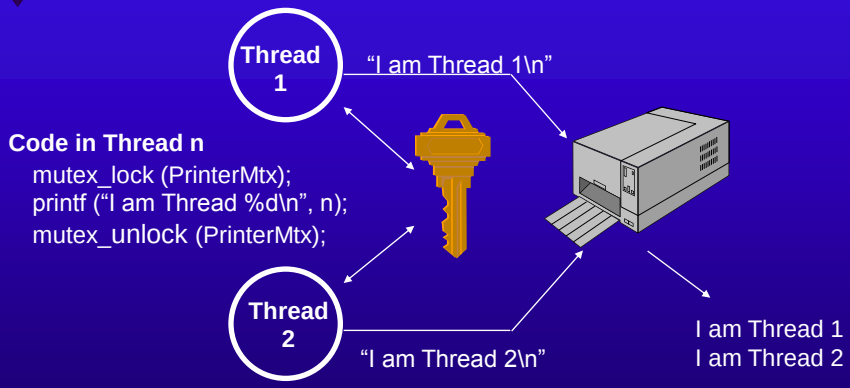


Code in Thread n

```
printf ("I am Thread %d\n", n);
```

Intellimetrix

Sharing Resources with a Mutex



Intellimetrix

Mutex API

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int pthread_mutex_init (pthread_mutex_t *mutex, const pthread_mutexattr_t
        *mutex_attr);
int pthread_mutex_destroy (pthread_mutex_t *mutex);

int pthread_mutex_lock (pthread_mutex_t *mutex);
int pthread_mutex_unlock (pthread_mutex_t *mutex);
int pthread_mutex_trylock (pthread_mutex_t *mutex);
```

Intellimetrix

Mutex Attributes

```
int pthread_mutexattr_init (pthread_mutexattr_t *attr);
int pthread_mutexattr_destroy (pthread_mutexattr_t *attr);

#ifdef _POSIX_THREAD_PROCESS_SHARED
int pthread_mutexattr_getpshared (pthread_mutexattr_t *attr, int *pshared);
int pthread_mutexattr_setpshared (pthread_mutexattr_t *attr, int pshared);
#endif

int pthread_mutexattr_getkind_np (pthread_mutexattr_t *attr, int *kind);
int pthread_mutexattr_setkind_np (pthread_mutexattr_t *attr, int kind);
```

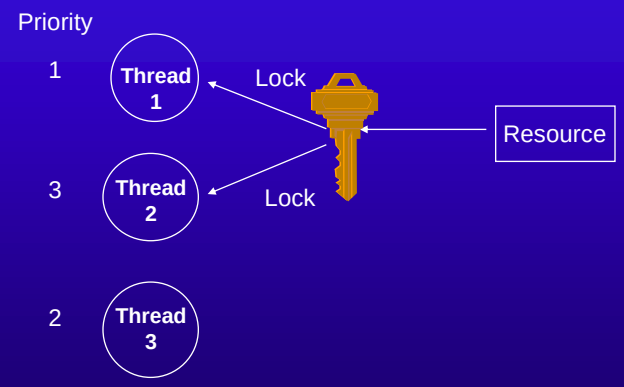
Intellimetrix

Mutex "Kind"

- Non-portable Linux extension
- Three "kinds"
  - PTHREAD_MUTEX_FAST_NP
  - PTHREAD_MUTEX_RECURSIVE_NP
  - PTHREAD_MUTEX_ERRORCHECK_NP

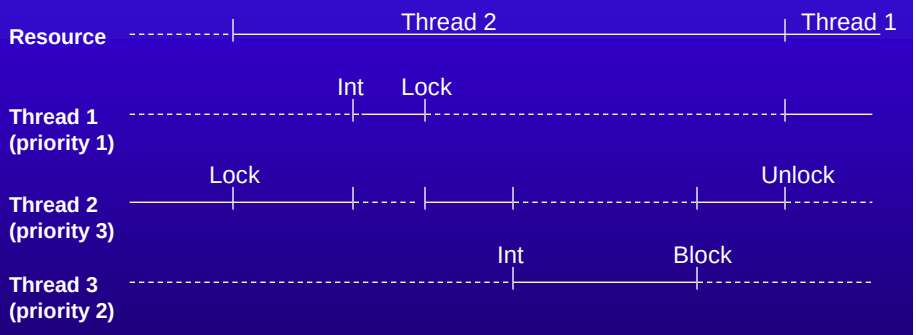
Intellimetrix

Priority Inversion



Intellimetrix

Priority Inversion 2



Intellimetrix

Priority Inversion Solutions

- Priority Inheritance
  - Temporarily raise priority of Thread 2 = Thread 1 when Thread 1 tries to lock the mutex
- Priority Ceiling
  - Raise priority of Thread 2 = priority of highest priority thread that will lock the mutex
  - Raise it when Thread 2 locks the mutex
- Set the mutex's *protocol*

Intellimetrix

Priority Inversion Solutions II

```
#if defined (_POSIX_THREAD_PRIO_PROTECT)
|| defined (_POSIX_THREAD_PRIO_INHERIT)
int pthread_mutexattr_getprotocol (const pthread_mutexattr_t *attr, int *protocol);
int pthread_mutexattr_setprotocol (pthread_mutexattr_t *attr, int protocol);
#endif

#ifdef _POSIX_THREAD_PRIO_PROTECT
int pthread_mutexattr_getprioceiling (const pthread_mutexattr_t *attr, int *ceiling);
int pthread_mutexattr_setprioceiling (pthread_mutexattr_t *attr, int ceiling);
int pthread_mutex_getprioceiling (const pthread_mutex_t *mutex, int *ceiling);
int pthread_mutex_setprioceiling (pthread_mutex_t * mutex, int ceiling);
#endif
```

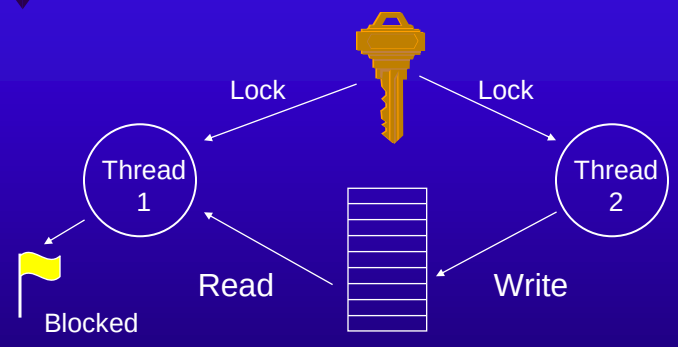
Intellimetrix

Conditional Variable

- "Signals" an event associated with shared data protected by a mutex
- Threads wait for an event to occur (with mutex locked!)
- Another thread signals that the event has occurred

Intellimetrix

Conditional Variable-- Example



Intellimetrix

Conditional Variable API

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

int pthread_cond_init (pthread_cond_t *cond, const pthread_condattr_t
        *cond_attr);
int pthread_cond_destroy (pthread_cond_t *cond);

int pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_timedwait (pthread_cond_t *cond, pthread_mutex_t *mutex,
        const struct timespec *abstime);
int pthread_cond_signal (pthread_cond_t *cond);
int pthread_cond_broadcast (pthread_cond_t *cond);
```

Intellimetrix

Conditional Variable Attributes

```
int pthread_condattr_init (pthread_condattr_t *attr);
int pthread_condattr_destroy (pthread_condattr_t *attr);

#ifdef _POSIX_THREAD_PROCESS_SHARED
int pthread_condattr_getpshared (pthread_condattr_t *attr, int *pshared);
int pthread_condattr_setpshared (pthread_condattr_t *attr, int pshared);
#endif
```

Intellimetrix

```
// Declare a queue structure type queue_t

void *write_queue_thread (void *arg)
{
      queue_t *q = (queue_t *) arg;
      int status;

      while (1)
      {
            wait_for_data();
            status = pthread_mutex_lock (&q->mutex);
            write_to_queue (q);
            status = pthread_cond_signal (&q->cond);
            status = pthread_mutex_unlock (&q->mutex);
      }
}
```

Continued on next slide

Intellimetrix

Continued from previous slide

```
void *read_queue_thread (void *arg)
{
     queue_t *q = (queue_t *) arg;
      int status;

      while (1)
      {
            status = pthread_mutex_lock (&q->mutex);
            if (queue_is_empty (q))
                  status = pthread_cond_wait (&q->cond);
            read_from_queue (q);
            status = pthread_mutex_unlock (&q->mutex);
      }
}
```

Intellimetrix

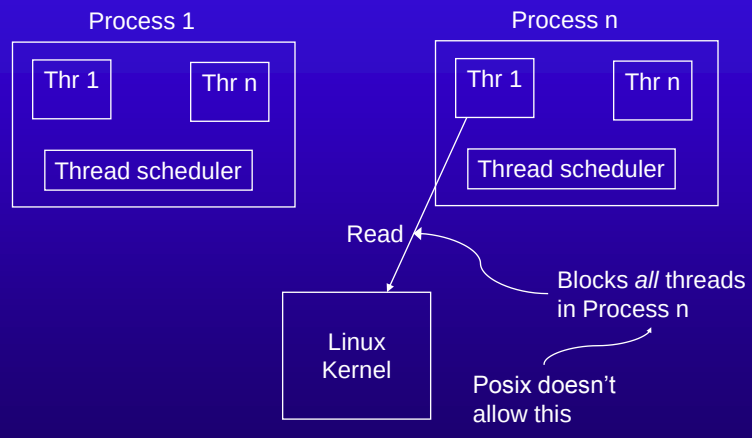
Contention Scope

int pthread_attr_getscope (const pthread_attr_t *attr, int *contentionscope);
int pthread_attr_setscope (const pthread_attr_t *attr, int contentionscope);

- PTHREAD_SCOPE_PROCESS
  - Thread competes only with other threads in process for resources
  - "cheap"
- PTHREAD_SCOPE_SYSTEM
  - Thread competes with all other threads in the system for resources
  - predictable

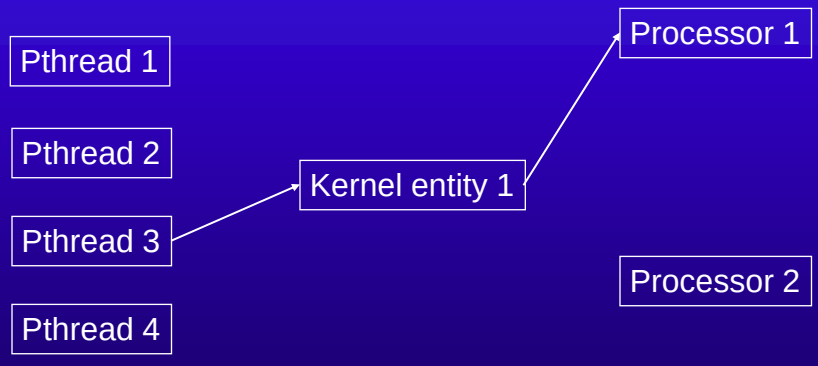
Intellimetrix

Threads in User Space



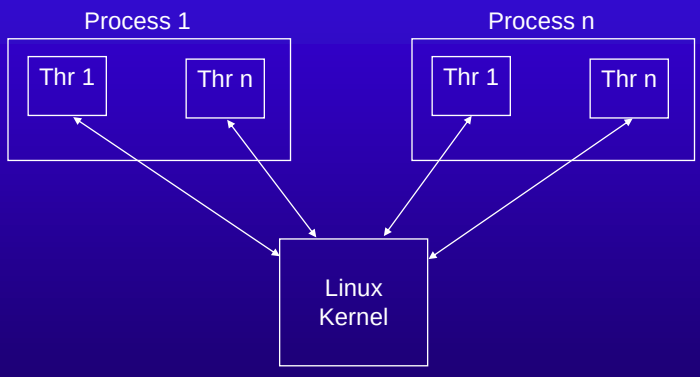
Intellimetrix

"Many-to-one" Thread Mapping



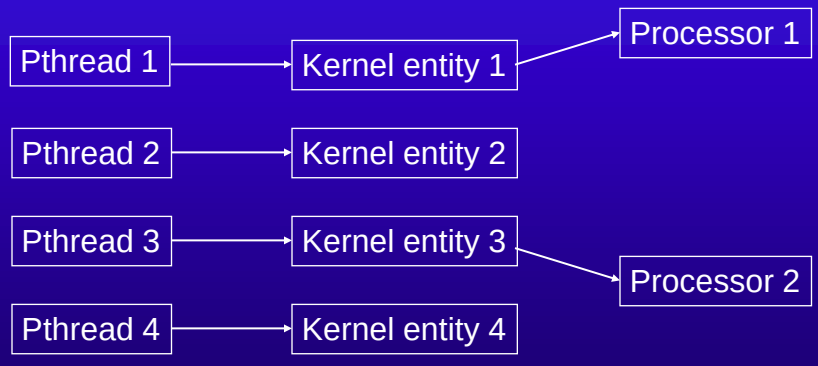
Intellimetrix

Threads managed by the kernel



Intellimetrix

"One-to-one" Thread Mapping



Intellimetrix

Thread Cancellation

```
int pthread_cancel (pthread_t thread);
int pthread_setcancelstate (int state, int *oldstate);
int pthread_setcanceltype (int type, int *oldtype);
void pthread_testcancel (void);
void pthread_cleanup_push (void (*routine)(void *), void
*arg);
void pthread_cleanup_pop (int execute);
```

Intellimetrix

Cancellation Type

- PTHREAD_CANCEL_DEFERRED
  - Only happens at a "cancellation point"
  - Most library functions that block have cancellation points
  - pthread_testcancel() is a cancellation point
- PTHREAD_CANCEL_ASYNCHRONOUS
  - Be very careful!
  - Can't allocate any resources
- Cancellation is asynchronous

Intellimetrix

```
include <pthread.h>
include <stdio.h>

int counter;

void *thread_function (void *arg);
{
     printf ("thread_function starting\n");
     for (counter = 0; ; counter++)
          if ((counter % 1000) == 0) {
               printf ("Calling testcancel\n");
               pthread_testcancel();
          }
}
Int main (int argc, char *argv)
{
     pthread_t thread_id;
     void *result;
     int status;

     status = pthread_create (&thread_id, NULL, thread_function, NULL);
     sleep (2);
```

Continued on next slide

Intellimetrix

Continued from previous slide

```
    status = pthread_cancel (thread_id);
    status = pthread_join (thread_id, &result);

    if (result == PTHREAD_CANCELED)
        printf ("Thread canceled at iteration %d\n", counter):
    else
        printf ("Thread not canceled\n");
}
```

Intellimetrix

"Cleanup Handlers"

- What if …
  - Thread has memory allocated
  - Mutex is locked
- Disable cancellation
- Cleanup handlers
  - "Pushed" on a conceptual stack
  - Executed in reverse order when thread is canceled
  - Can be "popped" when no longer needed

Intellimetrix

```
typedef struct {
            .
            .
    pthread_mutext_t mutex;
} control_t;

void cleanup (void *arg)
{
    control_ *st = (control_t *) arg;
    int status;

    pthread_mutex_unlock (&st->mutex)
}
```

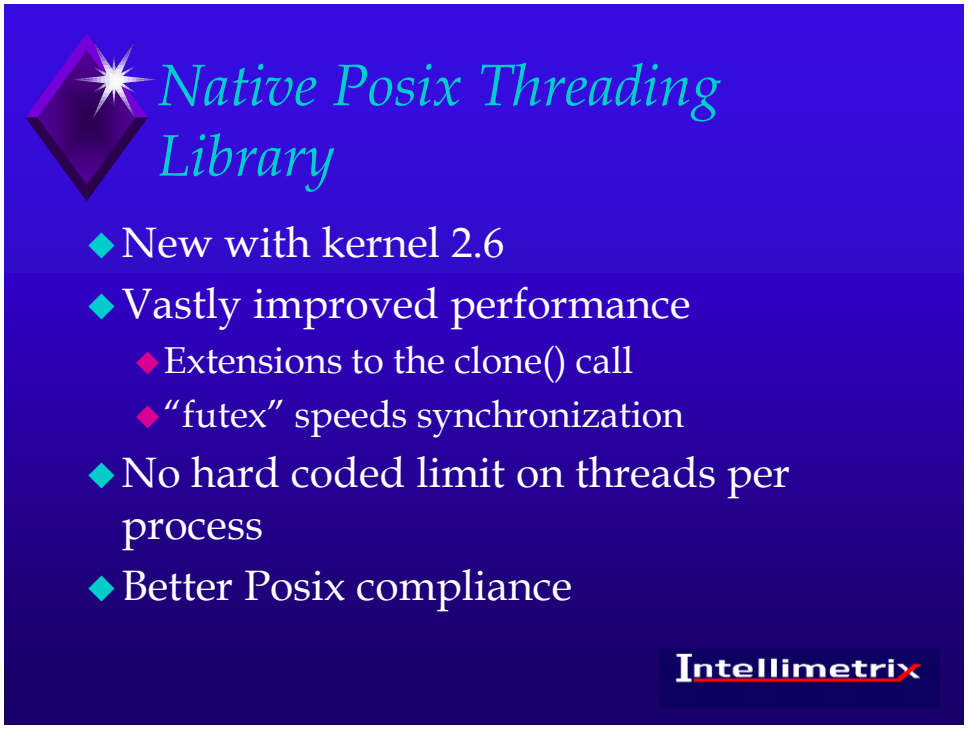
Continued on next slide

Intellimetrix

Continued from previous slide

```
void *thread_function (void *arg)
{
    control_t *control = (control_t *) arg;

    while (1) {
        // do something local

        pthread_cleanup_push (cleanup, (void *) control);
        status = pthread_mutex_lock (&control->mutex);

        // Access common resource

        pthread_cleanup_pop (1);

        // do something else local
    }
}
```

Intellimetrix


Native Posix Threading Library
New with kernel 2.6
Vastly improved performance
Extensions to the clone() call
"futex" speeds synchronization
No hard coded limit on threads per process
Better Posix compliance
Intellimetrix