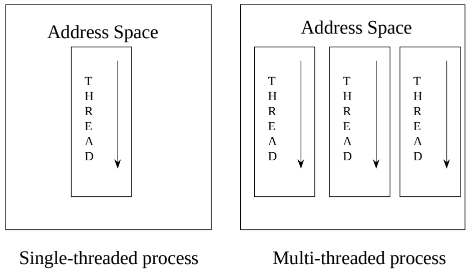
Threads



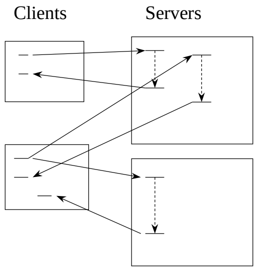
Why Threads?

- Means of effectively exploiting inherent parallelism in a distributed environment
- Different parts of an application can run in parallel
- Less overhead than processes: multiple threads share heap storage, static storage, and code, but each thread has its own registers and stack
- In multi-processor systems, threads can run concurrently on different processors

Threads in a Client/Server Environment

Servers can service multiple clients simultaneously

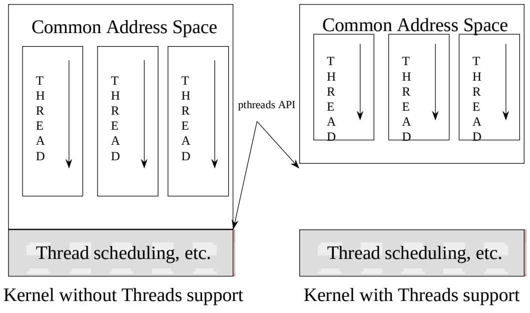
Clients can make multiple requests simultaneously



DCE Threads

- Threads are available on every DCE platform
- Other DCE services use threads for their operation
- Based on POSIX 1003.4a threads interface specification (known as pthreads)
  - The POSIX standard is now available on HP-UX, ULTRIX, OSF/1, etc.
  - DCE threads are based on pthreads.

Library Threads vs. Kernel Threads



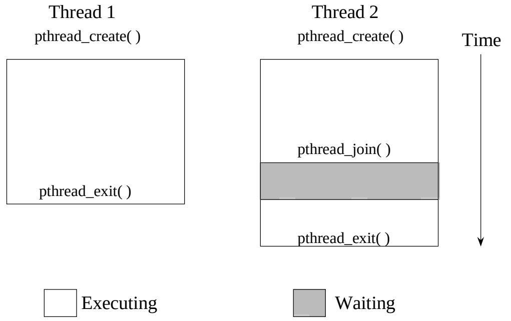
Concepts of Thread Operation

- Threads progress independently
- Threads within a process share same address space
- Threads can synchronize with one another
- Adding threads to a system may require changes
  - The process using threads must be reentrant
  - The system libraries must be thread-safe

Basic pthreads Routines

- pthread_create( )
- pthread_exit( )
- pthread_join( )
- pthread_yield( )

A Simple Example



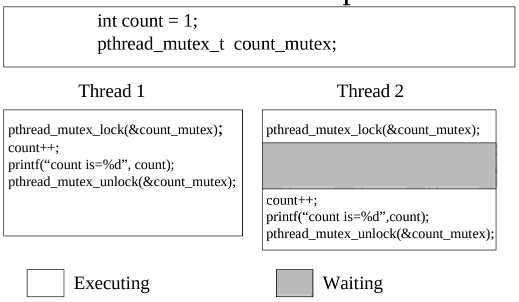
Thread Synchronization: Mutexes

- A mutex (mutual exclusion) is used to ensure integrity of shared resources
- Before using a shared resource each thread locks the mutex; it unlocks it after use
  - A thread attempting to lock an already locked mutex may block
- Mutexes are purely advisory; all threads must follow the rules

Some pthreads Routines for Mutexes

- pthread_mutex_init( )
- pthread_mutex_destroy( )
- pthread_mutex_lock( )
- pthread_mutex_trylock( )
- pthread_mutex_unlock( )

A Mutex Example



```
int count = 1;
pthread_mutex_t  count_mutex;
```

Thread 1

```
pthread_mutex_lock(&count_mutex);
count++;
printf("count is=%d", count);
pthread_mutex_unlock(&count_mutex);
```

Thread 2

```
pthread_mutex_lock(&count_mutex);
count++;
printf("count is=%d",count);
pthread_mutex_unlock(&count_mutex);
```

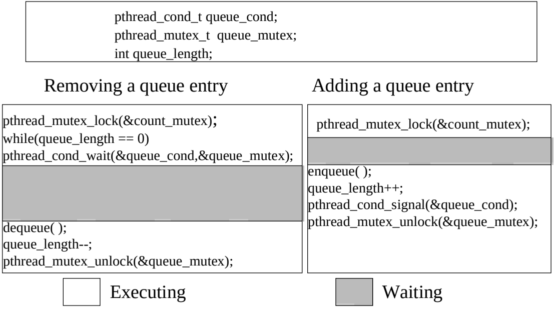
Thread Synchronization: Condition Variables

- A condition variable allows a thread to block its own execution until it is signaled by another thread that some shared data is in a specific state.
- A condition variable is used for thread synchronization

Some pthreads Routines for Condition Variables

- pthread_cond_init( )
- pthread_cond_destroy( )
- pthread_cond_wait( )
- pthread_cond_timedwait( )
- pthread_cond_signal( )
- pthread_cond_broadcast( )

A Condition Variable Example



```
pthread_cond_t queue_cond;
pthread_mutex_t  queue_mutex;
int queue_length;
```

Removing a queue entry

```
pthread_mutex_lock(&count_mutex);
while(queue_length == 0)
pthread_cond_wait(&queue_cond,&queue_mutex);

dequeue( );
queue_length--;
pthread_mutex_unlock(&queue_mutex);
```

Adding a queue entry

```
pthread_mutex_lock(&count_mutex);

enqueue( );
queue_length++;
pthread_cond_signal(&queue_cond);
pthread_mutex_unlock(&queue_mutex);
```

□ Executing ▨ Waiting

Thread-safe System Calls

- System calls can cause problems without kernel threads
  - If a call blocks, it might block the entire process instead of just the thread
- DCE provides *wrapper* routines for I/O
  - When a thread invokes a system call that could block, the wrapper is called
  - The wrapper ensures that the entire process is not blocked

Summary

- Threads are a modern concurrency mechanism
- General purpose, well-suited for distributed environment
- Comprehensive support for application development
- Integrated with an used by other DCE components
- Based on POSIX draft