# Multithreading

Reading:

Silberschatz

chapter 5

Additional Reading:

Stallings

chapter 4

*Understanding Linux/Unix Programming*, Bruce Molay, Prentice-Hall, 2003.

# Outline

- Process and Threads
- Multithreading
- Motivation
- Advantages
- RPC using Thread(s)
- User-Level Threads
- Kernel-Level Threads
- Combined Approaches
- Phtreads
- Threading Issues
  - System Call Semantics
  - Thread Cancellation
  - Signal Handling
  - Thread Pools
  - Thread Specific Data
- Introduction: Linux, Win32, Solaris and Java Threads

# Process and Threads

- Process Management
  - **Resource Ownership**
    - Memory, I/O channel, I/O devices and file
  - **Scheduling/Execution**
    - *Execution state* and priority
- Independent treatment by OS
- Unit of dispatching
  - Thread
  - Lightweight Process (LWP, even KLT)
- Unit of resource ownership
  - Process
  - Task

# Multithreading

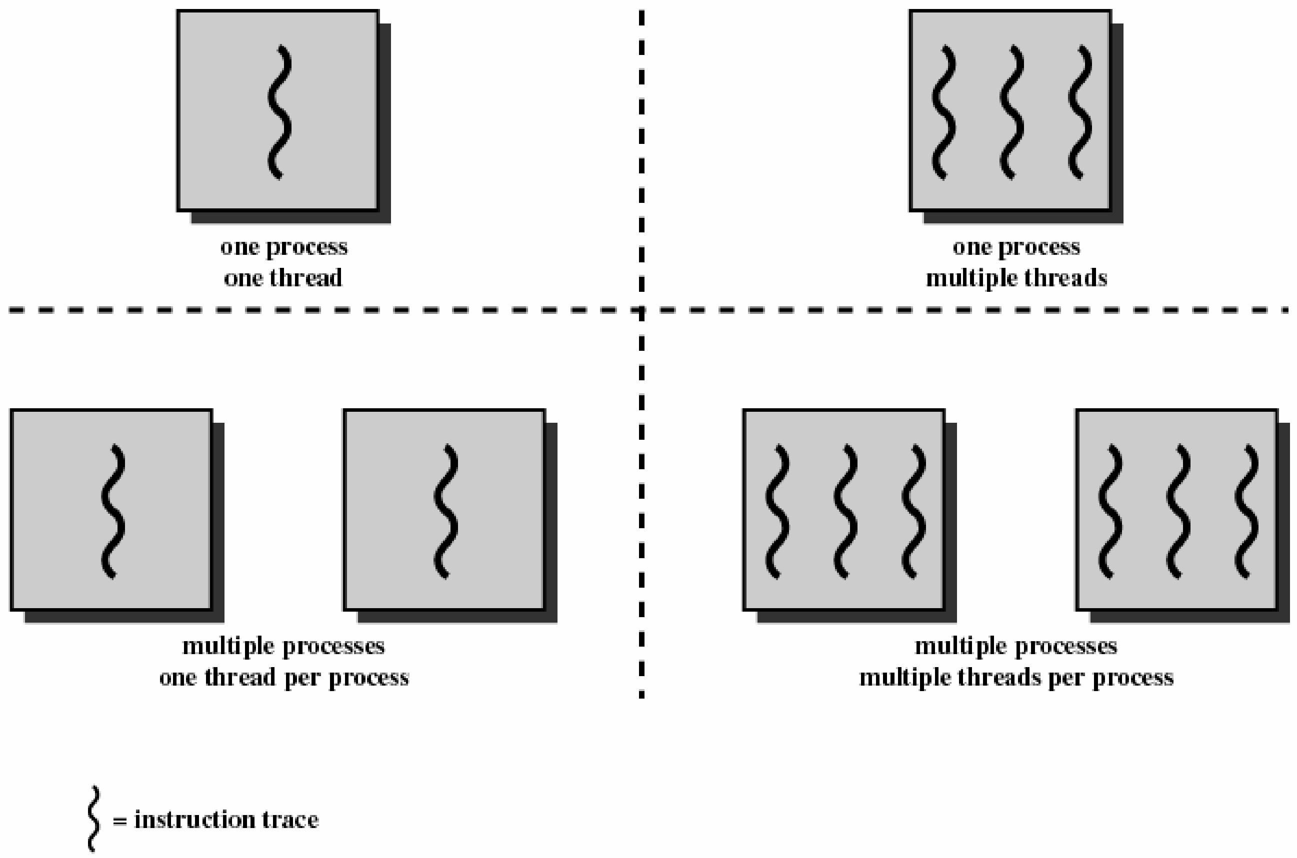
**Multiple threads of execution within single process**

- Single Thread: Traditional approach

- OS Support for Threads
  - MSDOS - a single user process and a single thread
  - UNIX - multiple user processes but only supports one thread per process
  - Windows, Solaris, Linux, Mach, and OS/2 - multiple threads

- Thread
  - **Basic unit of CPU utilization, consisting of**
    - PC
    - Register set
    - Stack

# Multithreading



# Examples - Motivation

- Web Browser
  - One thread to display images
  - Other thread retrieves data from network
- Word Processor
  - One thread for responding to keystrokes
  - Other thread for spelling and grammar checking
  - Other thread for displaying graphics
- File Server on LAN
  - *Controller* thread accepts file service requests and spawns *worker* thread for each request
  - Can handle many requests concurrently, thread finishes service - destroyed

# Process

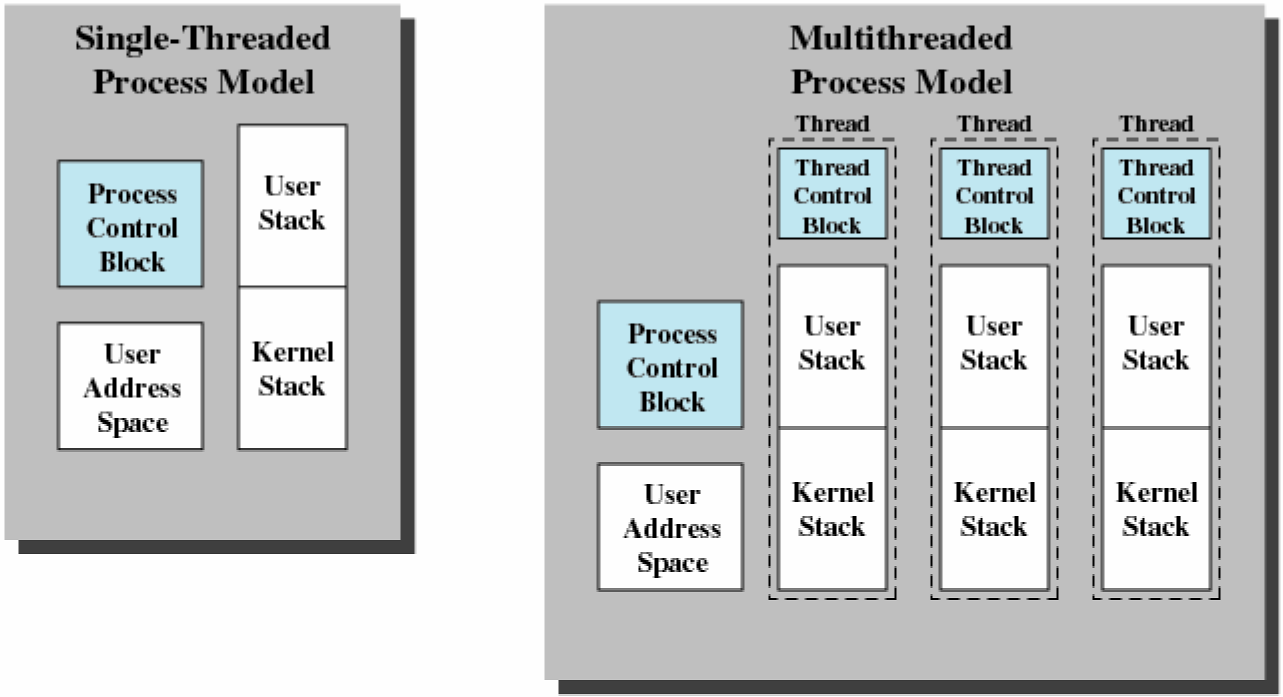
**Unit of resource allocation and a unit of protection, associated:**

- A virtual address space which holds the process image
- Protected access to processors, other processes, files, and I/O resources
  - Within the process there can be one or more threads

# Thread

- An execution state (running, ready, blocked)
- Saved thread context/state when not running
- Has an execution stack
- Some per-thread static storage for local variables
- Access to the memory and resources of its process
  - Shared by all threads of the process

# Multithreading




- All of the threads of a process share the state and resources of process
- They reside in same address space and have access to same data

# Advantage Threads!

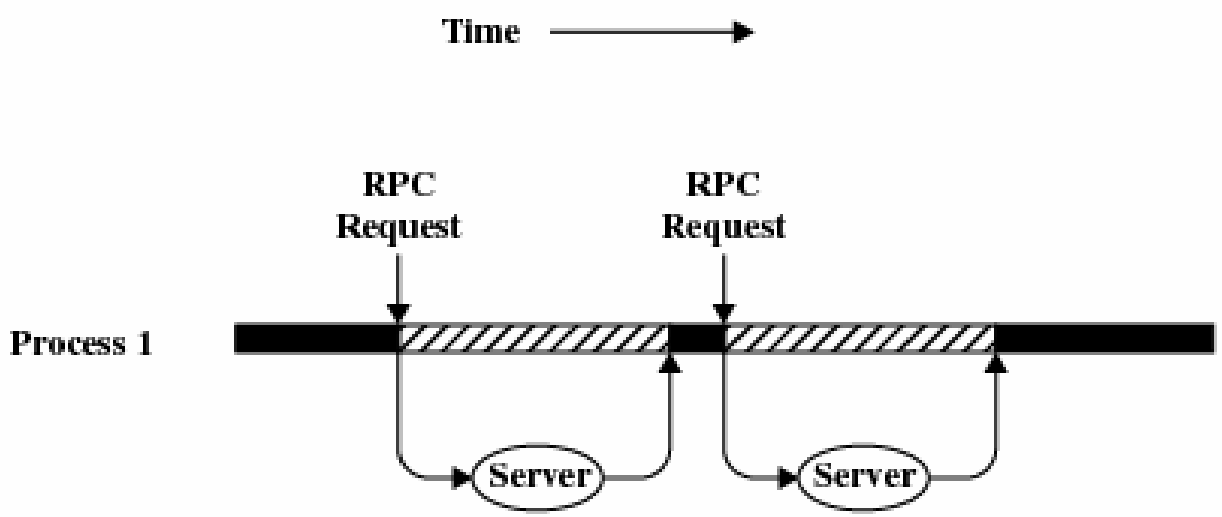
- Takes far less time to create a new thread in existing process than a new process; Factor 10
- Less time to terminate a thread than a process
- Less time to switch between two threads within the same process
- Since threads within the same process share memory and files, they can communicate with each other without invoking the kernel

# Functionality

- **Key States** – RRB but no suspend

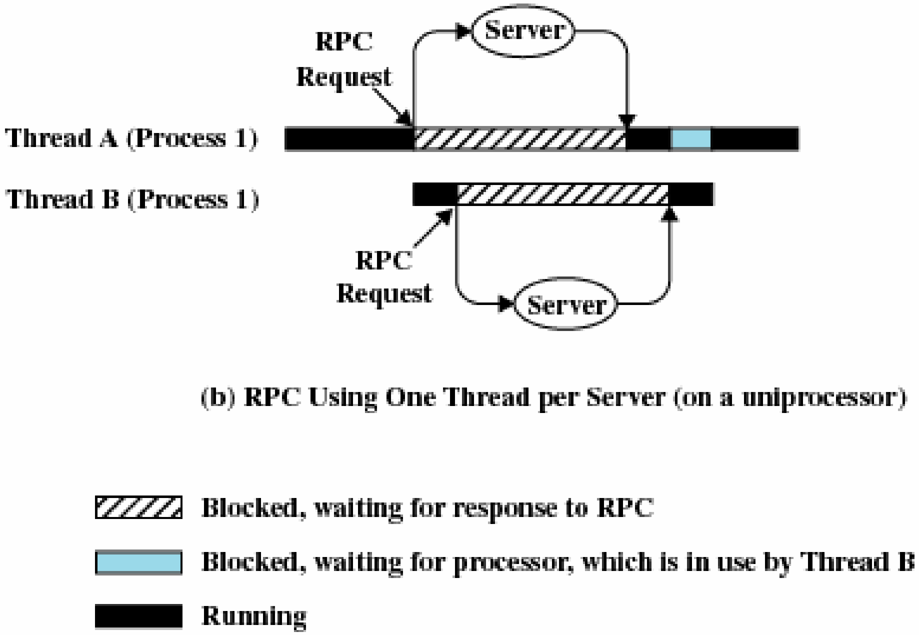
- **Operations**
  - **Spawn**
  - **Spawn another thread**
  - **Block**
  - **Unblock**
  - **Finish**

# RPC Using Single Thread



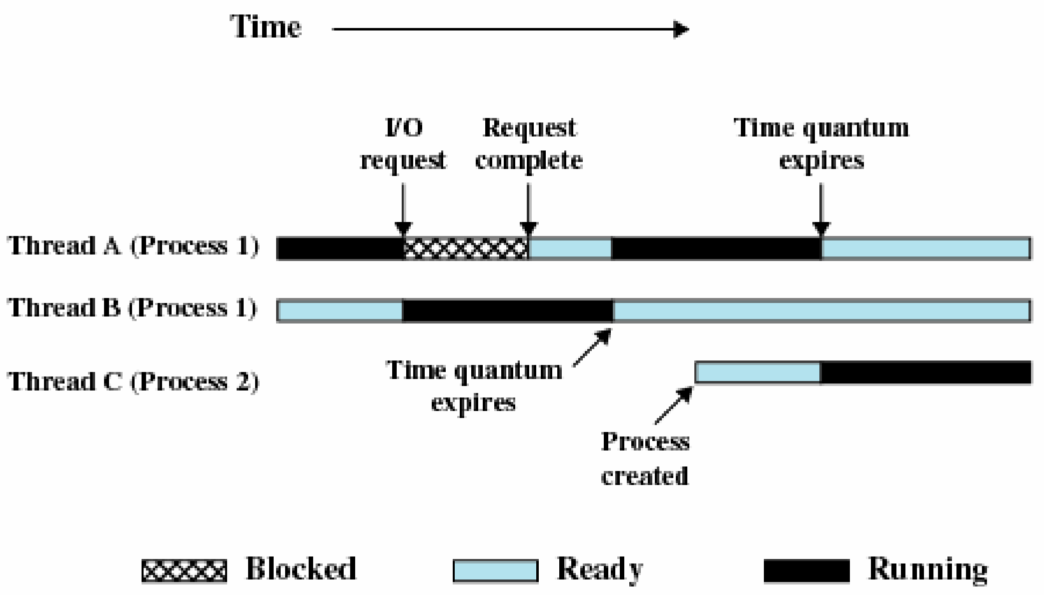

(a) RPC Using Single Thread

# RPC Using Threads



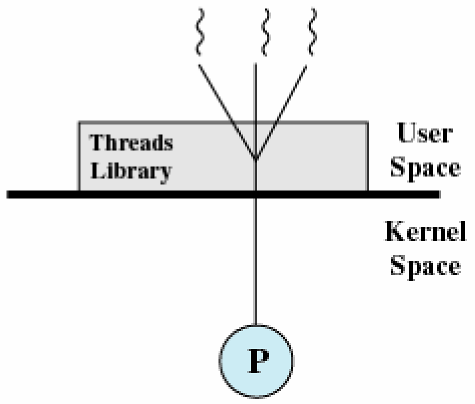

(b) RPC Using One Thread per Server (on a uniprocessor)

# Example - Multithreading



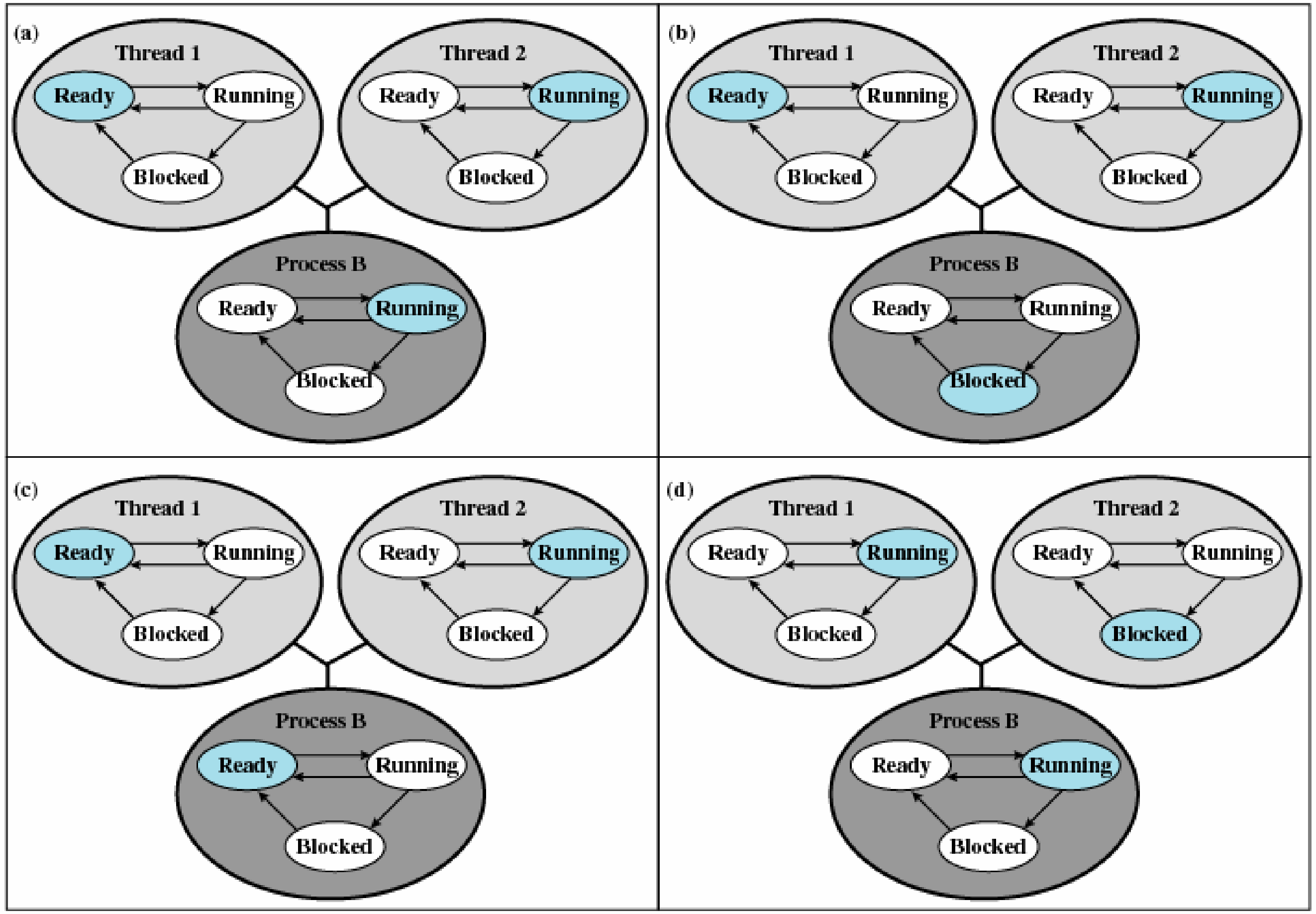
# User-Level Threads

- Thread Management → Application
- New threads within the same process




- Thread Library
- Control, Library ↔ Thread
- Context → user reg, pc & sp

# User-Level Threads




Colored state
is current state

# ULT Vs KLT

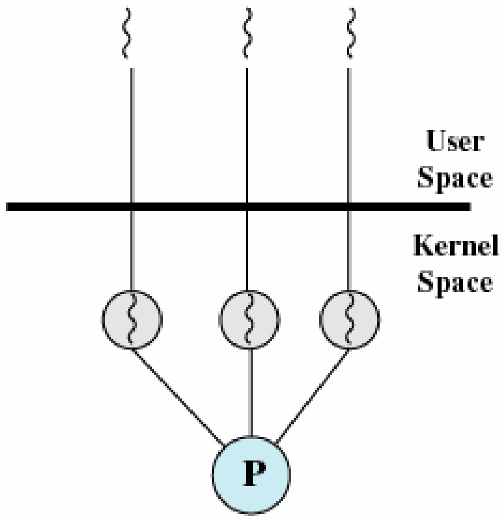
- Advantages of ULTs
  - Thread switching → Kernel mode privileges, *less overhead*
  - Scheduling can be application specific
  - Thread libraries → application utilities, Can run on any OS

- Disadvantages of ULTs
  - High blocking, OS → many system calls are blocking, all threads in process are blocked
  - Pure ULT strategy → cannot take advantage of multiprocessing

- Solutions?
  - Jacketing
  - Writing application as multiple processes rather than threads

# Kernel-Level Threads

- Thread Management → Kernel
- API to the kernel thread facility



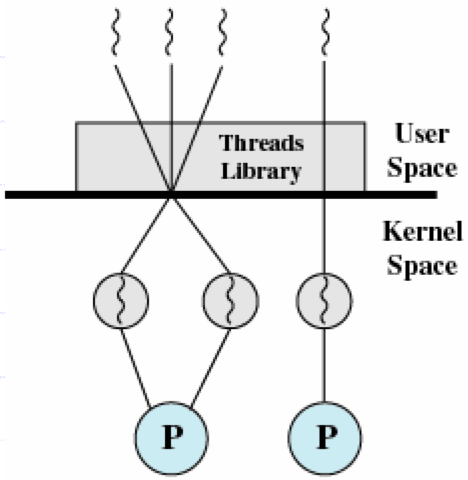

- Thread based *scheduling* by kernel
- Pure ULT strategy → cannot take advantage of multiprocessing
- Transfer of control → mode switch

# Kernel-Level Threads

**Table 4.1 Thread and Process Operation Latencies (μs) [ANDE92]**

| Operation | User-Level Threads | Kernel-Level Threads | Processes |
|---|---|---|---|
| **Null Fork** | 34 | 948 | 11,300 |
| **Signal Wait** | 37 | 441 | 1,840 |

# Combined Approaches




- Thread creation → user space, Solaris
- Bulk of scheduling and synchronization of threads within application
- Multiple threads within the same application → multiple processors
- Entire process is not blocked, Design

# Relationship Between Threads and Processes

| Threads:Processes | Description | Example Systems |
|---|---|---|
| **1:1** | Each thread of execution is a unique process with its own address space and resources. | Traditional UNIX implementations |
| **M:1** | A process defines an address space and dynamic resource ownership. Multiple threads may be created and executed within that process. | Windows NT, Solaris, Linux OS/2, OS/390, MACH |
| **1:M** | A thread may migrate from one process environment to another. This allows a thread to be easily moved among distinct systems. | Ra (Clouds), Emerald |
| **M:N** | Combines attributes of M:1 and 1:M cases. | TRIX |

# Threading Issues

- Semantics of System Calls
  - fork()
  - exec()

- Thread Cancellation
  - Asynchronous Cancellation
  - Deferred Cancellation

# Threading Issues

- Signal Handling
  - To thread to which its applicable
  - To every thread
  - To certain threads
  - To a specific thread assigned
- Thread Pools
  - Sit & Wait
  - Work, Return to pool
  - Faster than waiting to create a thread
  - Limits # that can exists at any point of time
- Thread Specific Data

# Pthreads

- A POSIX standard API for thread creation and sync
- API implementation dependent on OS
- Common in UNIX operating systems
- All programs → pthread.h
  - pthread_create()
  - pthread_suspend()
  - pthread_yield()
  - pthread_continue()
  - pthread_join()
  - pthread_exit(); pthread_sigmask(); sigwait()

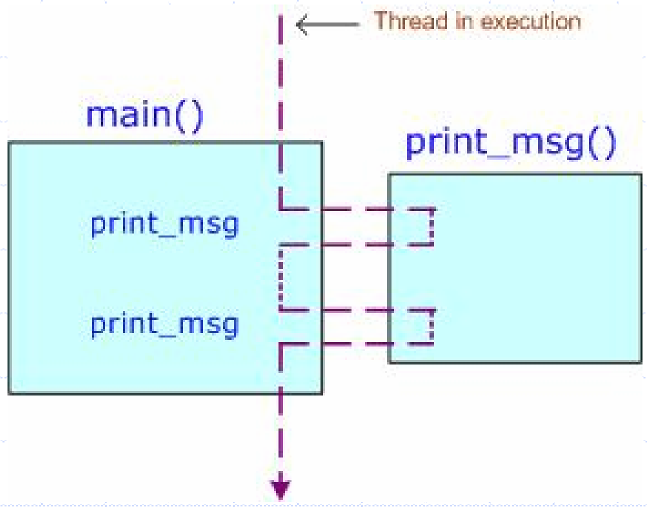
# Example: Running 2 functions simultaneously

## Single-threaded version

**Duration:** 10 seconds

**Source code:**

```
/* hello_single.c - a single threaded hello world program */

    #include <stdio.h>
    #define NUM 5

    main()
    {
      void print_msg(char *);

      print_msg("hello");
      print_msg("world\n");
    }

void print_msg(char *m)
    {
      int i;
      for(i=0 ; i<NUM ; i++){
      printf("%s", m);
      fflush(stdout);
      sleep(1);
       }
    }
```




**Screen output:**

```
[ajaykr@lib ~]$ ./hello_single
    hellohellohellohellohelloworld
    world
    world
    world
    world
```

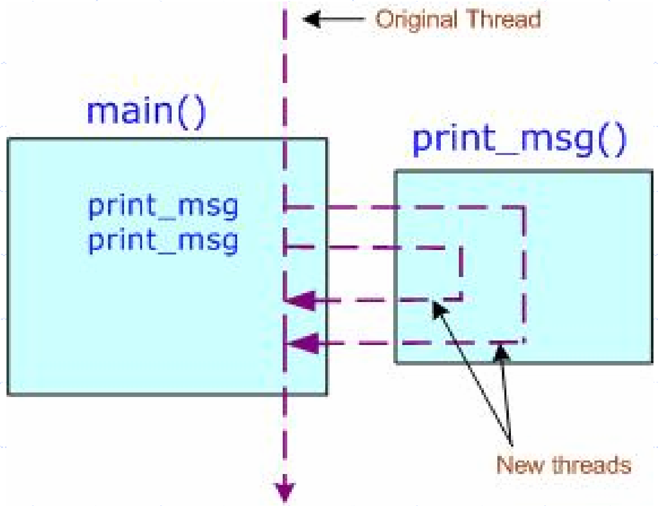
# Example: Running 2 functions simultaneously

**Multi-threaded version**

**Duration:** 5 seconds

**Source code:**

```
/* hello_multi.c - a multi-threaded hello world program */
      #include <stdio.h>
      #include <pthread.h>

      #define NUM 5

      main()
      {
        pthread_t t1, t2; /* two threads */

        void *print_msg(void *);

        pthread_create(&t1, NULL, print_msg, (void *)"hello");
        pthread_create(&t2, NULL, print_msg, (void *)"world\n");
        pthread_join(t1, NULL);
        pthread_join(t2, NULL);
      }

void *print_msg(void *m)
      {
        char *cp = (char *) m;
        int i;
        for(i=0 ; i<NUM ; i++){
        printf("%s", m);
        fflush(stdout);
        sleep(1);
      }
      return NULL;
      }
```




**Screen output:**

```
[ajaykr@lib ~]$ ./hello_multi
      helloworld
      helloworld
      helloworld
      helloworld
      helloworld
```

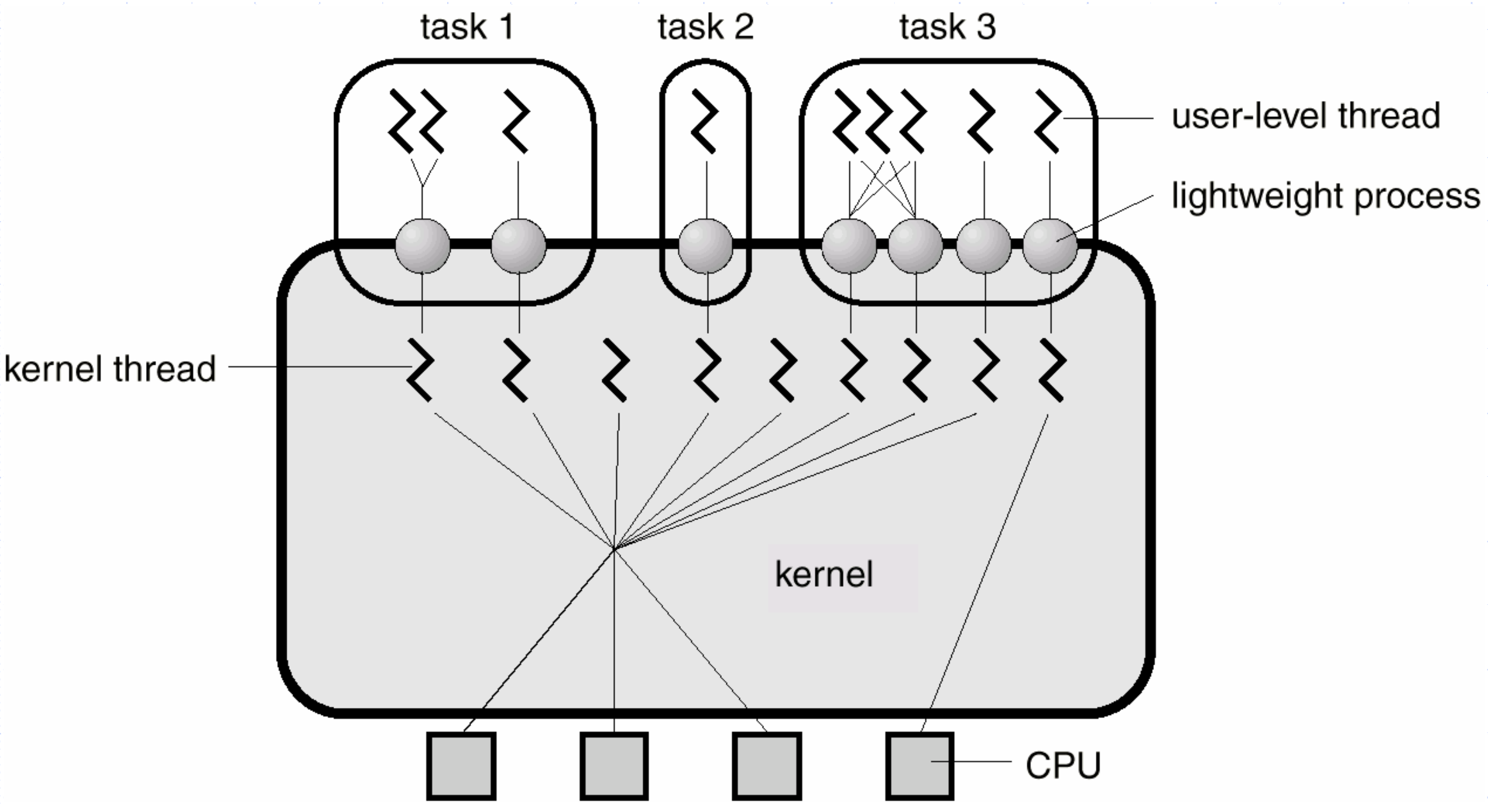
# Linux Threads

- Linux refers to them as *tasks* rather than *threads*
- Thread creation → Clone() system call, Flags
  - CLONE_FS
  - CLONE_VM
  - CLONE_SIGHAND
  - CLONE_FILES
- All flags → thread, No flags → fork (no sharing)
- Clone() allows a child task to share the address space of the parent task (process)

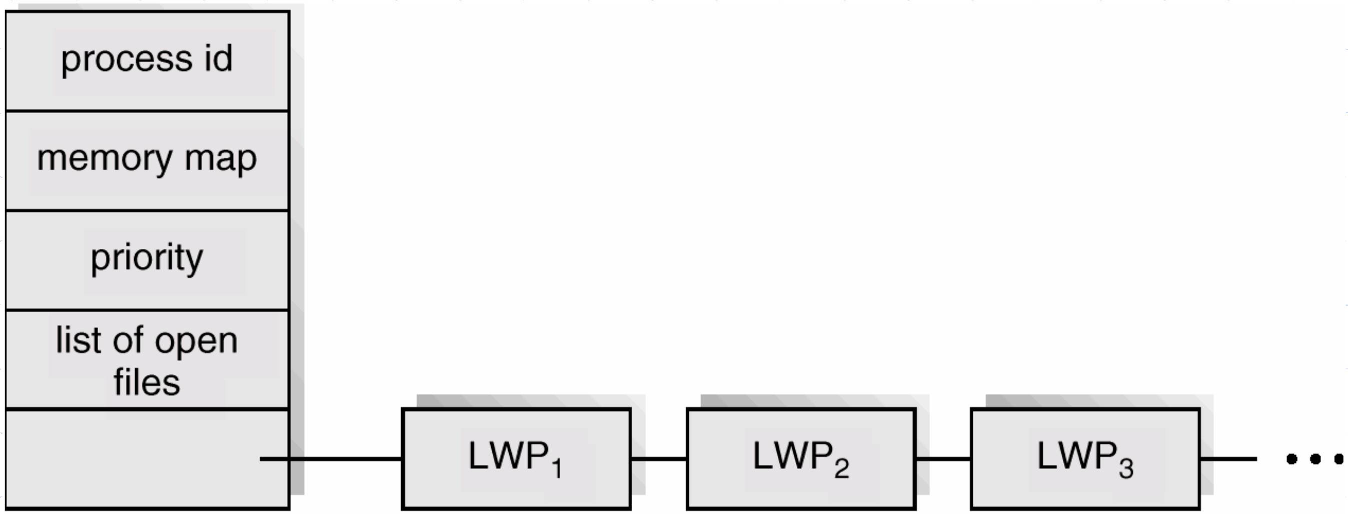
# Win32 Threads

- Primarily API for *Win XP, NT, 2000, 98, 95*
- windows.h
- Thread creation → CreateThread(), …
- Windows XP application → Separate Process
- Components of thread
  - a thread id
  - register set
  - separate user and kernel stacks
  - private data storage area
- 1:1 Mapping, *fiber* library

# Solaris 2 Threads



# Solaris Process

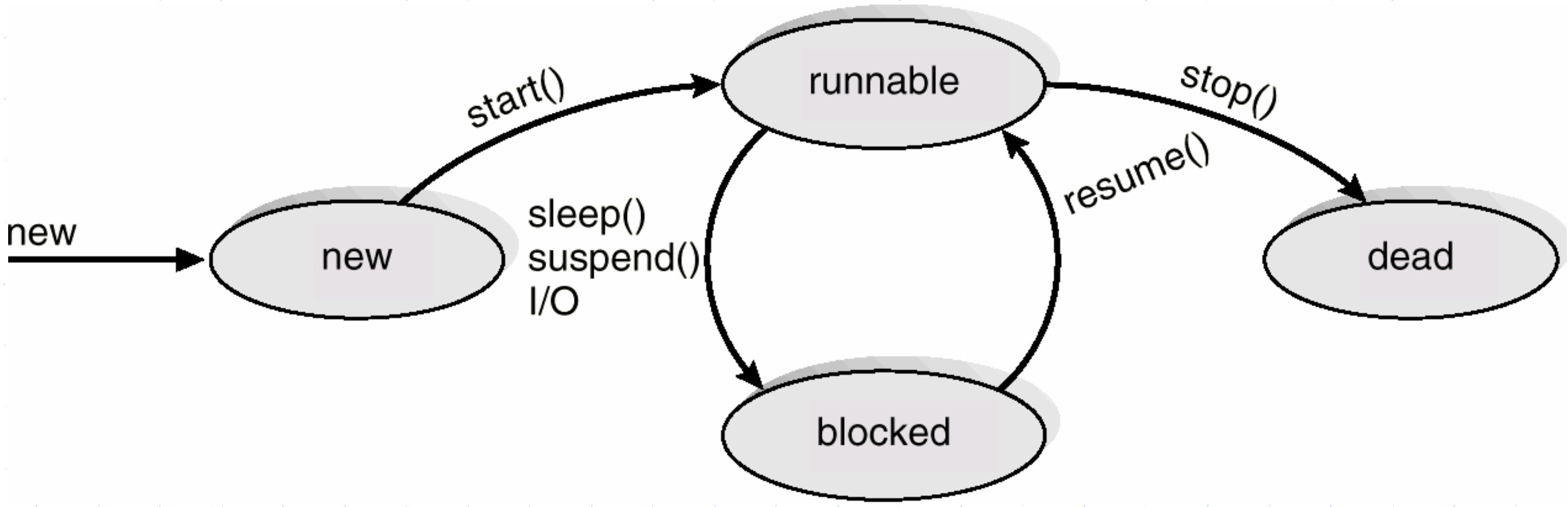



Solaris process

# Java Threads

- Java $\rightarrow$ language level support
- Management $\rightarrow$ JVM, alternative to user/kernel
- Thread Creation
  - Extending Thread class, new class $\rightarrow$ thread class
  - Implementing the Runnable interface
- Java threads mapping $\rightarrow$ depends on OS

# Java Thread States



# Questions

- Provide examples where multithreading does not provide better performance than single-threaded solutions

- Can a multithreaded solution using multiple user-level thread achieve better performance on a multiple CPU system than on a single processor system?

- Consider a multiprocessor system (CPU) and a multithreaded program written using combined model. Let the number of user level threads in program be more than # of processors in CPU. Discuss/Predict the performance in following scenarios.
  - The # KLTs allocated to the process {<, =} to the # processors
  - The # KLTs allocated to the process > the # of processors but < the # ULTs