Threads

Reference: Pradeep K Sinha,

"Distributed Operating Systems: Concepts and Design", Prentice Hall of India, 2007

Overview

- Threads
- Organizing Threads
- Issues in Designing Threads
- Thread Scheduling
- Thread Implementation

Process

- A basic unit of CPU utilization in traditional OS.
- A program in execution.
- Defines a data space.
- Has its own program counter, its own register states, its own stack and its own address space.
- Has at least one associated thread.
- Unit of distribution.

Threads

- It is a basic unit of CPU utilization used for improving a system performance through parallelism.
- A process consist of an address space and one or more threads of control.
- Threads share same address space but having its own program counter, register states and its own stack.
- Less protection due to the sharing of address space.

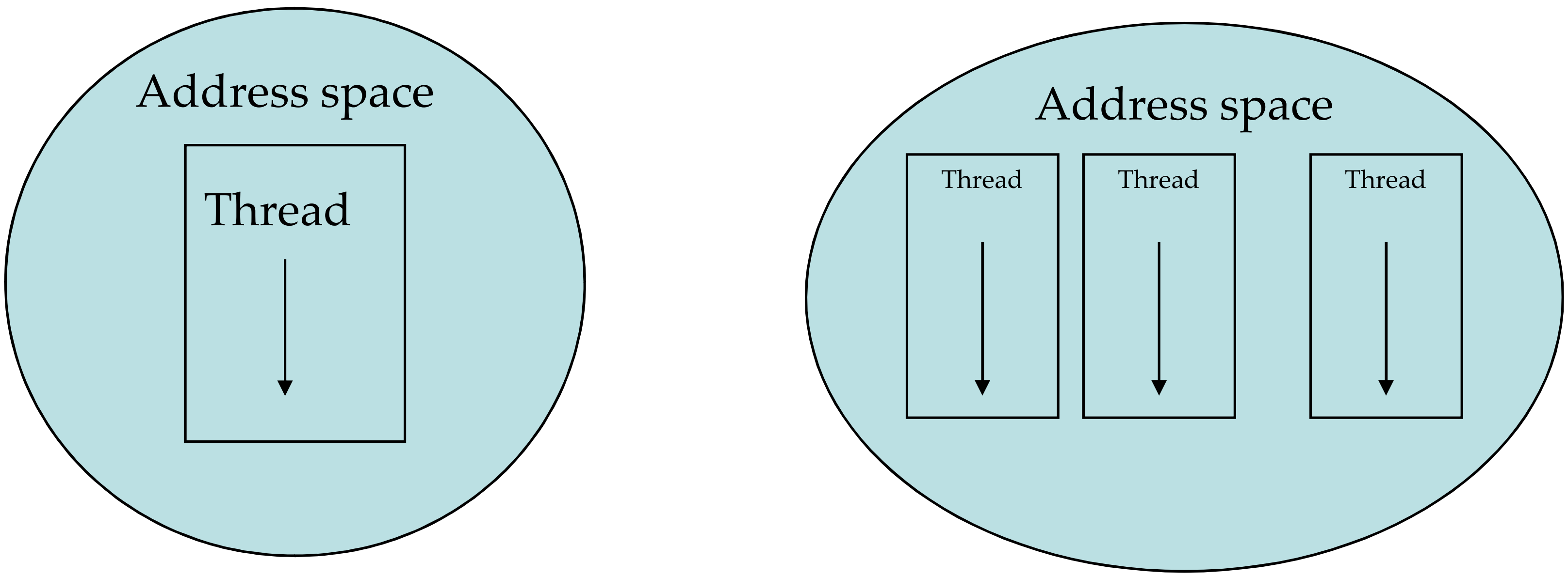
Threads

- On a uniprocessor, threads run in quasi-parallel (time sharing), whereas on a shared-memory multiprocessor, as many threads can run simultaneously as there are processors.

- States of Threads:
  - Running, blocked, ready, or terminated.

- Threads are viewed as mini-processes.

Threads



(a) Single-threaded (b) Multithreaded processes

Motivations for using Threads

- Overheads involved in creating a new process is more than creating a new thread.

- Context switching between threads is cheaper than processes due to their same address space.

- Threads allow parallelism to be combined with sequential execution and blocking system calls.

- Resource sharing can be achieved more efficiently and naturally between threads due to same address space.

Different models to construct a server process: As a single-thread process

- Use blocking system calls but without any parallelism.

- If a dedicated machine is used for the file server, the CPU remains idle while the file server is waiting for a reply from the disk space.

- No parallelism is achieved in this method and fewer client requests are processed per unit of time.

As a finite state machine

- Model support parallelism but with nonblocking system calls.
- Implemented as a single threaded process and is operated like a finite state machine.
- An event queue is maintained for request & reply.
- During time of a disk access, it records current state in a table & fetches next request from queue.
- When a disk operation completes, the appropriate piece of client state must be retrieved to find out how to continue carrying out the request.

As a group of threads

- Supports parallelism with blocking system calls.
- Server process is comprised of a single dispatcher thread and multiple worker threads.
- Dispatcher thread keeps waiting in a loop for request from the clients.
- A server process designed in this way has good performance and is also easy to program.

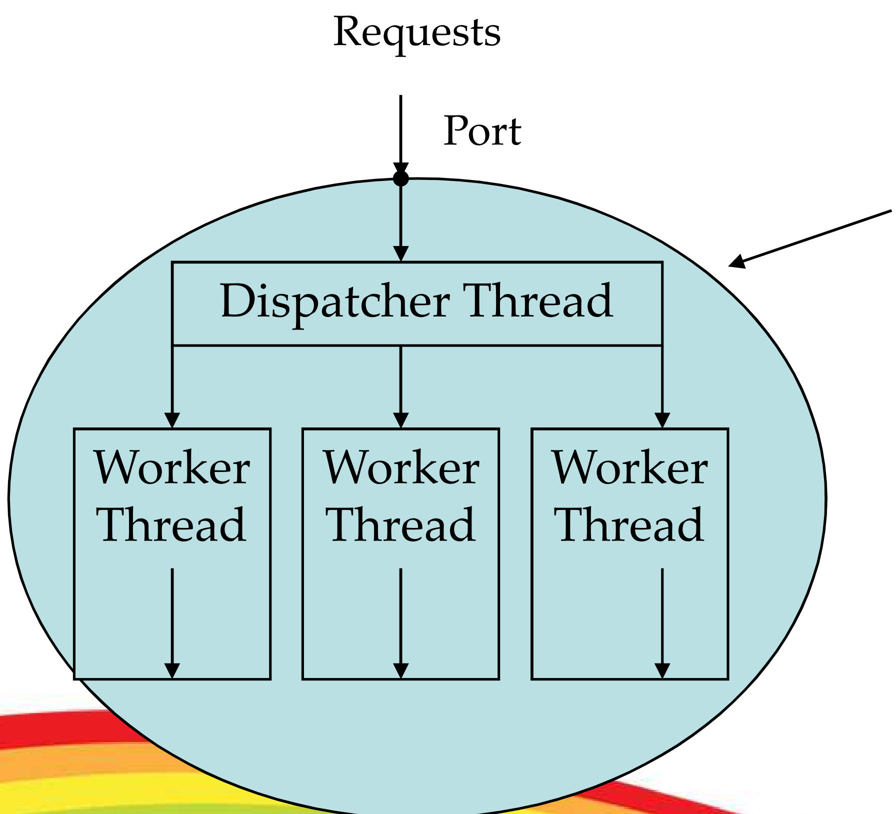
Models for organizing threads

- Dispatcher-workers model
- Team model
- Pipeline model

Dispatcher-worker model

- Single dispatcher thread and multiple worker threads.
- Dispatcher thread accepts requests from clients and after examining the request, dispatches the request to one of the free worker threads for further processing of the request.
- Each worker thread works on a different client request.

Dispatcher-worker model

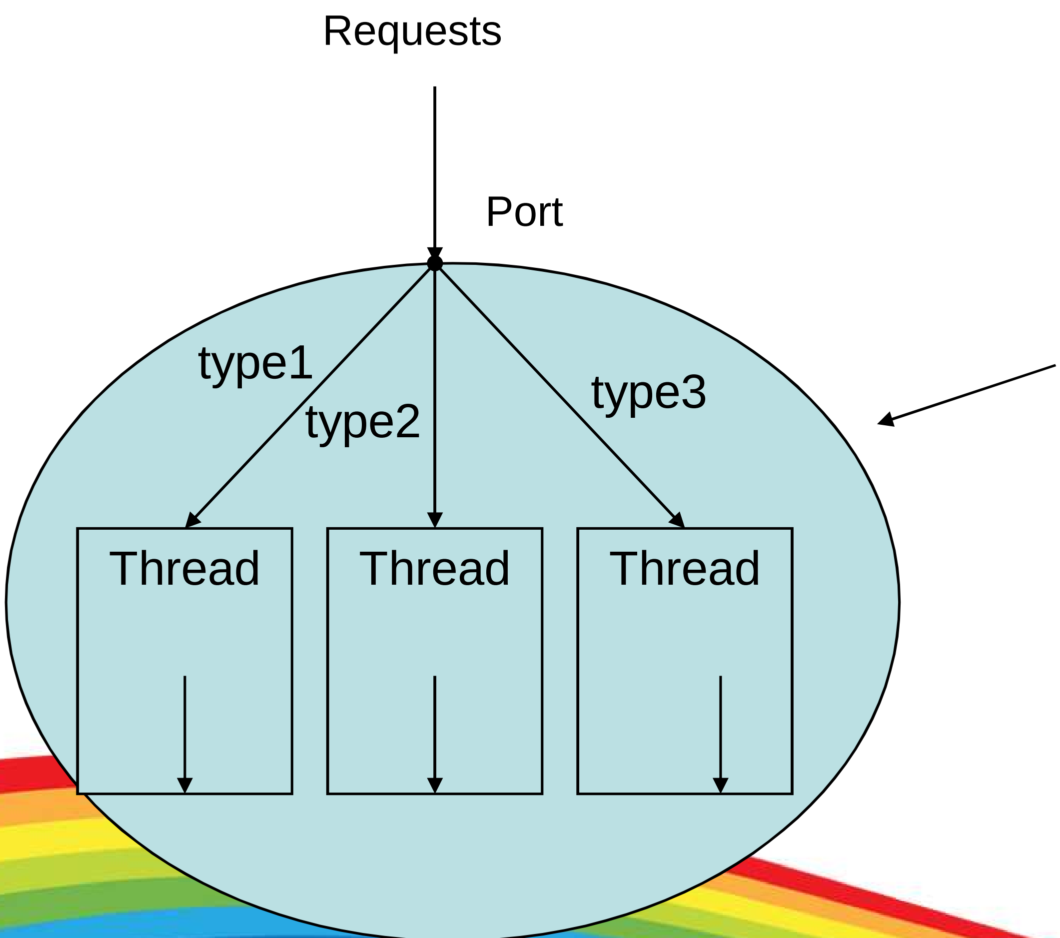


A Server process for processing incoming requests

Team Model

- There is no dispatcher-worker relationship for processing clients requests.
- Each thread gets and processes clients' requests on its own.
- Each thread of the process is specialized in servicing a specific type of request.
- Multiple types of requests can be simultaneously handled by the process.

Team Model



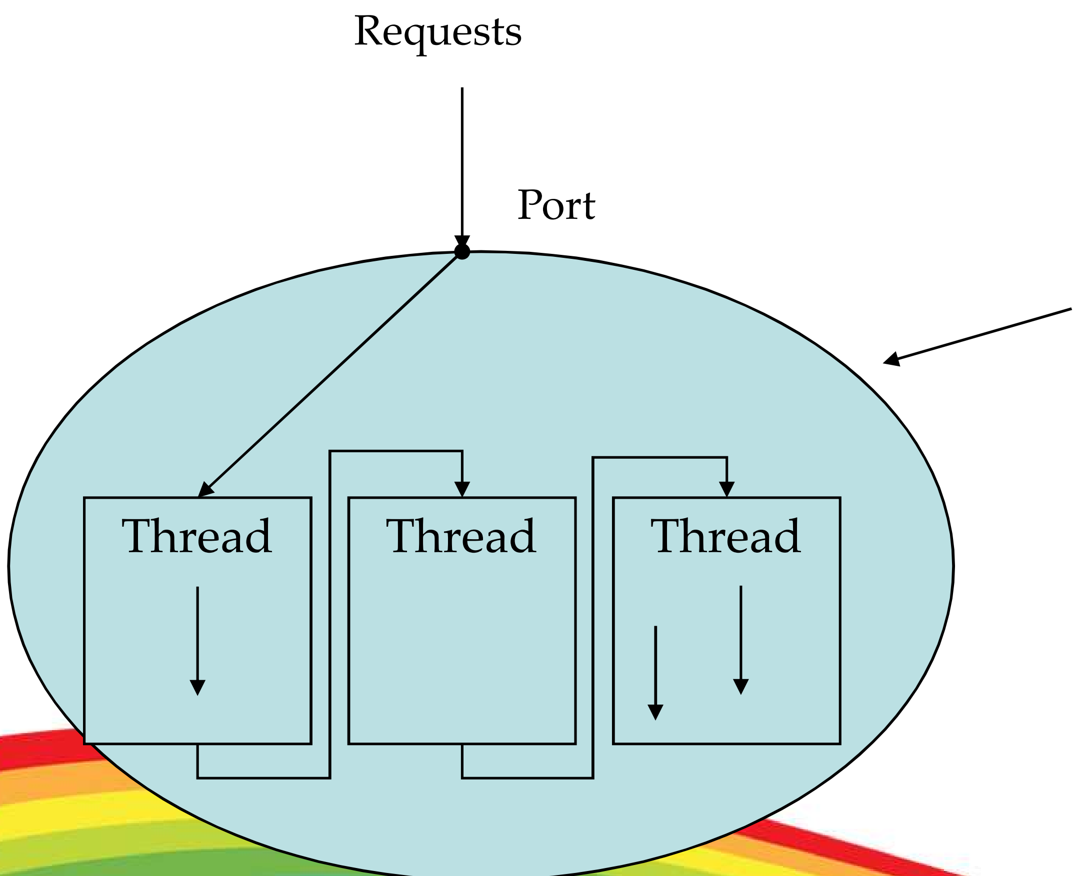
A Server process for processing incoming requests that may be of three different types, each types of request being handled by a different thread.

Pipeline Model

- Useful for application based on the producer-consumer model.

- The threads of a process are organized as a pipeline so that the output data generated by the first thread is used for processing by the second thread and so on.

Pipeline Model



A Server process for processing incoming requests, each request processed in three steps, each step handled by a different thread and output of one step as input to the next step

Issues in designing a Thread Package

1. Threads creation
2. Thread termination
3. Threads synchronization
4. Threads scheduling
5. Signal handling

Threads Creation

- Created either statically or dynamically.

- Static approach:
  - Threads remain fixed for its entire lifetime.
  - Fixed stack is allocated to each thread.
  - No. of threads of a process is decided at the time of writing a program or during compilation.

- Dynamic approach:
  - Number of threads changes dynamically.

Threads Termination

- A thread may either destroy itself when it finishes its job by making an exit call

or

- be killed from outside by using the kill command and specifying the thread identifier as its parameter.

Threads Synchronization

- Threads share a common address space, so it is necessary to prevent multiple threads from trying to access the same data simultaneously.

- Mechanism for threads synchronization:
  - Use of global variable within the process.
    - mutex variable and condition variable.

Threads Synchronization

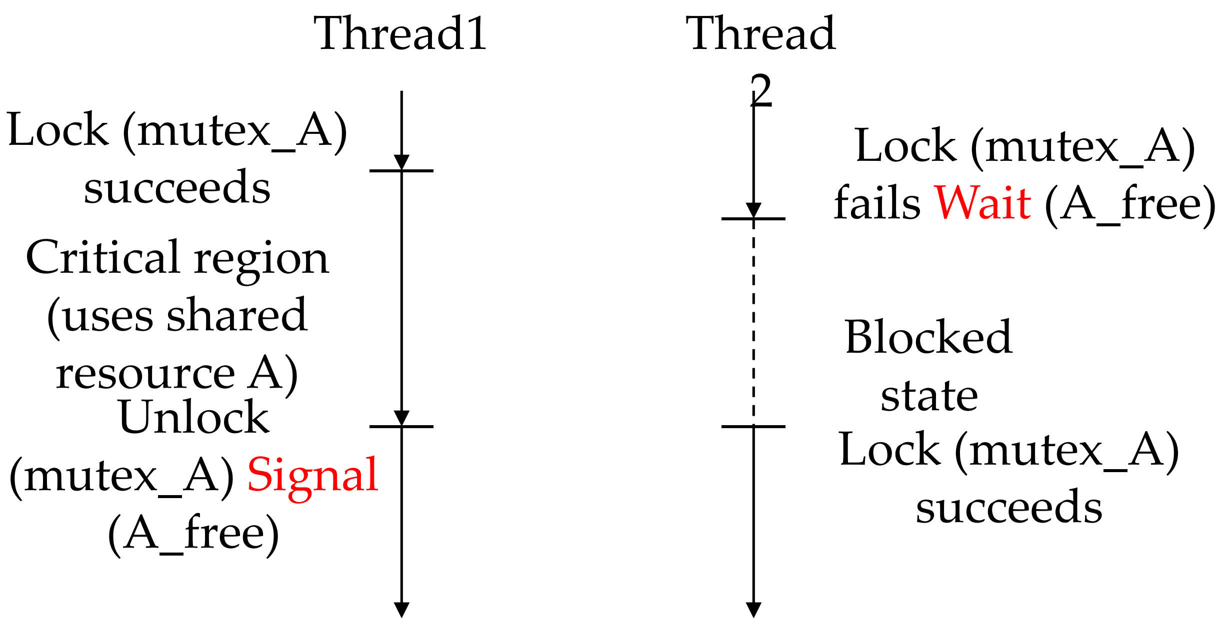
- Segment of code in which a thread may be accessing some shared variable is called a critical region.

- Two commonly used mutual exclusion techniques in a threads package are mutex variables and condition variables.

Threads Synchronization

- Mutex have only 2 states, hence simple to implement.
- Condition variable associated with mutex variable & reflects Boolean state of that variable.
- Condition – wait and signal operation.

Threads Synchronization



Mutex_A is a mutex variable for exclusive use of shared resource A.
A_free is a condition variable for resource A to become free.

Threads Scheduling

- Special features for threads scheduling :
  1. Priority assignment facility
  2. Flexibility to vary quantum size dynamically
  3. Handoff scheduling
  4. Affinity scheduling

Priority assignment facility

- Threads are scheduled on the first-in, first-out basis or the round-robin policy is used.

- Used to timeshares the CPU cycles.

- To provide more flexibility priority is assigned to the various threads of the applications.

- Priority thread maybe non-preemptive or preemptive.

Flexibility to vary quantum size dynamically

- Use of round-robin scheduling scheme.
- Varies the size of fixed-length time quantum to timeshare the CPU cycle among the threads.
- Not suitable for multiprocessor system.
- Gives good response time to short requests, even on heavily loaded systems.
- Provides high efficiency on lightly loaded systems.

Handoff Scheduling

- Allows a thread to name its successor if it wants to.

- Provides flexibility to bypass the queue of runnable threads and directly switch the CPU to the thread specified by the currently running thread.

Affinity scheduling

- A thread is scheduled on the CPU it last ran on, in hopes that part of its address space is still in that CPU's cache.

- Gives better performance on a multiprocessor system.

Signal handling

- Signals provide software-generated interrupts and exceptions.

- Issues :
  - A signal must be handled properly no matter which thread of the process receives it.

  - Signals must be prevented from getting lost.

Signal handling

- Approach:
  - Create a separate exception handler thread in each process.
  - Assign each thread its own private global variables for signaling conditions.

Thread Implementation

- Often provided in form of a thread package.

- Two important approaches:
  - First is to construct thread library that is entirely executed in user mode.

  - Second is to have the kernel be aware of threads and schedule them.

User level threads

- It is cheap to create and destroy threads.
- Cost of creating or destroying thread is determined by the cost for allocating memory to set up a thread stack.
- Switching thread context is done in few instructions.
- Only CPU registers need to be stored & subsequently reloaded with the previously stored values of the thread to which it is being switched.
- There is no need to change memory maps, flush the TLB, do CPU accounting etc.

User level threads

- Drawback:
  - Invocation of a blocking system call will immediately block the entire process to which the thread belongs.

User and Kernel Level

- Threads are useful to structure large application into parts that could be logically executed at the same time.
- In user level thread, blocking on I/O does prevent other parts to be executed in the meantime.
- This can be circumvented by implementing threads in the OS Kernel.

User and Kernel Level

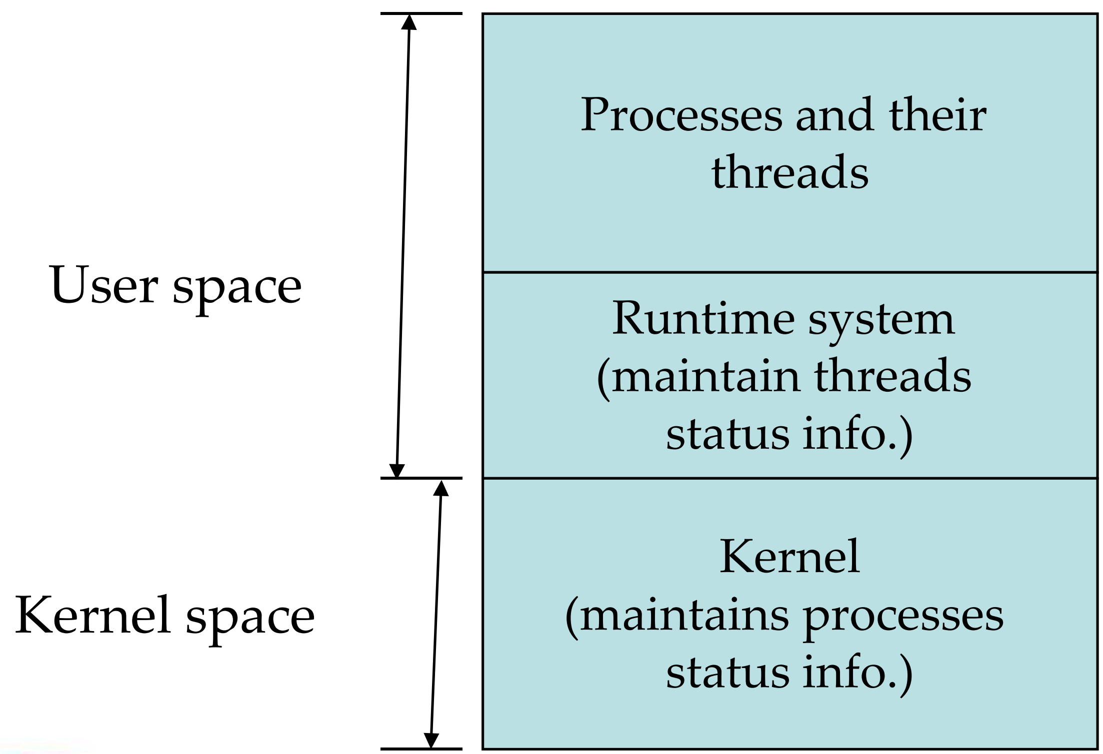
- Price:
  - Every thread operation will have to be carried out by kernel requiring system calls.
  - Hence benefits of using threads disappear.

Implementing a Threads Package

User-level approach

- Consists of a runtime system that is a collection of threads management routines.
- Threads run in the user space on the top of the runtime system and are managed by it.
- System maintains a status information table having one entry per thread.
- Two-level scheduling is performed in this approach.

User level

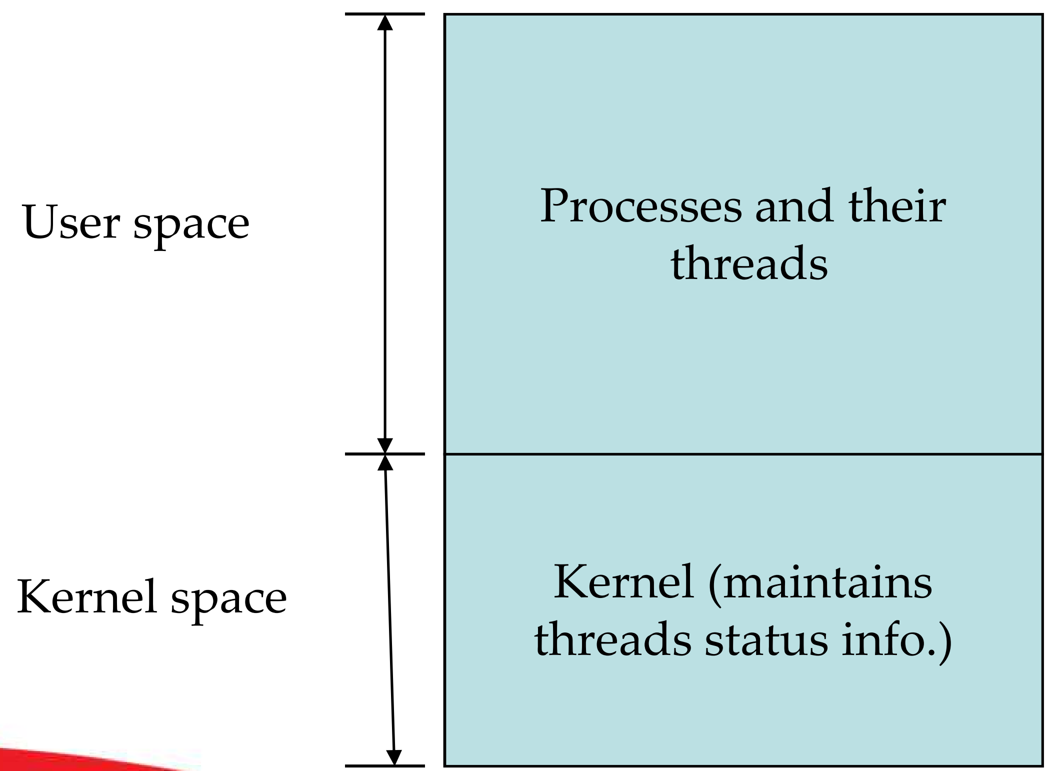

User space
Kernel space
Processes and their threads
Runtime system (maintain threads status info.)
Kernel (maintains processes status info.)

Implementing a Threads Package

Kernel-level approach

- No runtime system is used and threads are managed by the kernel.
- Threads status info. table maintained within kernel.
- Single-level scheduling is used in this approach.
- All calls that might block a thread are implemented as system calls that trap to the kernel.

Kernel level



Implementing a Threads Package

Advantages of the approaches

- User-level approach can be implemented on top of an existing OS that does not support threads.
- In user-level approach due to use of two-level scheduling, users have the flexibility to use their own customized algorithm to schedule the threads of a process.
- Switching the context from one thread to another is faster in the user-level approach than in the kernel approach.
- No status information cause poor scalability in the kernel as compared to the user-level approach.

Implementing a Threads Package

Disadvantages of the approaches:

- Since there is no clock interrupt within a single process, so once CPU is given to a thread to run, there is no way to interrupt it.

- To solve this runtime system can request a clock interrupt after every fixed unit of time

Thank You