Operating Systems and Concurrency

Lecture 5: Threads

G52OSC/COMP2007

Geert De Maere
(Isaac Triguero)
{Geert.DeMaere,Isaac.Triguero}@Nottingham.ac.uk

University Of Nottingham
United Kingdom

2018

Recap

Last Lecture

- **Types of schedulers**: preemptive/non-preemptive, long/medium/short term
- Performance **evaluation criteria**
- Scheduling **algorithms**: FCFS, SJF, Round Robin, Priority Queues

Goals for Today

Overview

1. Threads vs. processes
2. Different thread implementations
3. POSIX Threads (PThreads)

- A process consists of two **fundamental units**
  - **Resources**: all related resources are grouped together
    - A logical address space containing the process image (program, data, heap, stack)
    - Files, I/O devices, I/O channels, ...
  - **Execution trace**, i.e., an entity that gets executed
- A process can **share its resources** between **multiple execution traces**, i.e., multiple threads running in the same resource environment

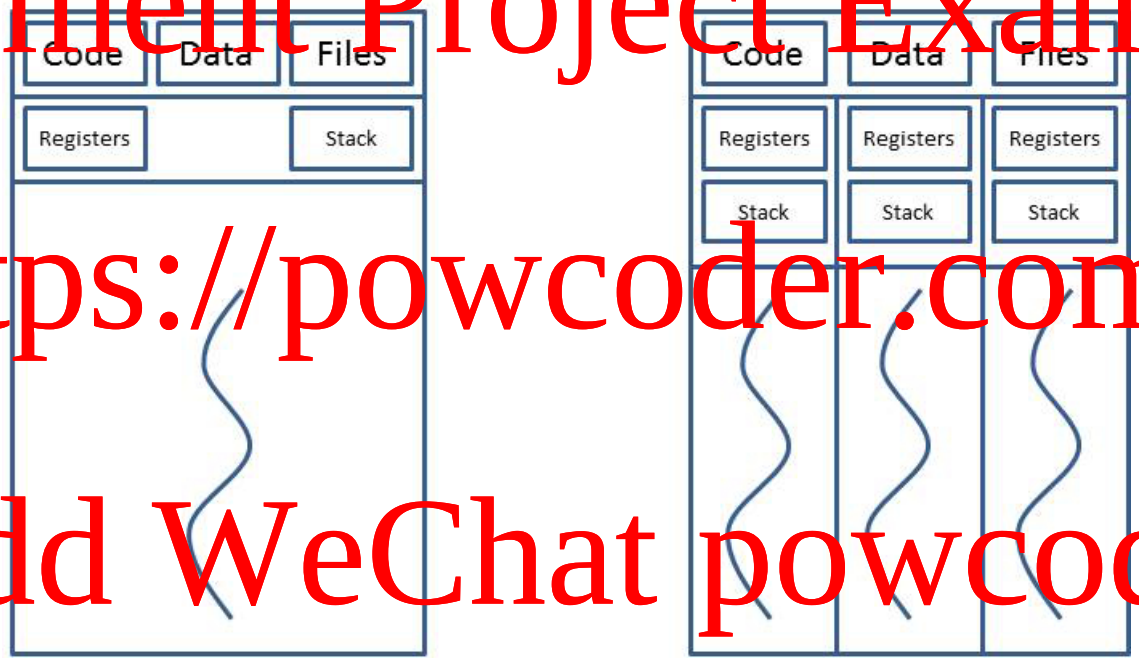


Figure: Single threaded process (left), multi-threaded process (right)

- Every thread has its own **execution context** (e.g. program counter, stack, registers)
- All threads have **access** to the process' **shared resources**
  - E.g. files, one thread opens a file, all threads of the same process can access the file
  - Global variables, memory, etc. ($\Rightarrow$ synchronisation!)
- Similar to processes, threads have:
  - **States** and **transitions** (new, running, blocked, ready, terminated)
  - A **thread control block**

Processes	Threads
Address space	Program Counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	Local vars
Signals and signal handlers	
Accounting information	

Table: Shared resources left, private resources right

- Threads incur **less overhead** to create/terminate/switch (address space remains the same for threads of the same process)
- Some CPUs (hyperthreaded ones) have direct **hardware support** for **multi-threading**
  - They can offer up to 8 hardware threads per core

- **Inter-thread communication** is easier/faster than **interprocess** communication (threads share memory by default)
- **No protection boundaries** are required in the address space (threads are cooperating, belong to the same user, and have a common goal)
- **Synchronisation** has to be considered carefully!

- Multiple **related activities** apply to the **same resources**, these resources should be accessible/shared
- Processes will often contain multiple **blocking tasks**
  - I/O operations (thread blocks, **interrupt** marks completion)
  - Memory access: pages faults are result in blocking
- Such activities should be carried out in **parallel/concurrently**
- **Application examples**: webservers, make program, spreadsheets, word processors, processing large data volumes

- **User** threads
- **Kernel** threads
- **Hybrid** implementations

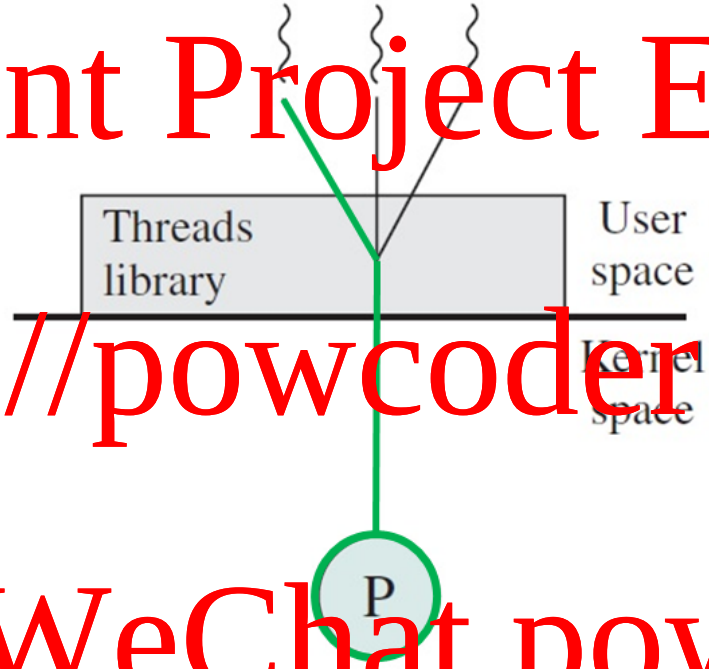
User Threads

Many-to-One

- **Thread management** (creating, destroying, scheduling, thread control block manipulation) is carried out **in user space** with the help of a **user library**

- The process maintains a **thread table** managed by the **runtime system** without the **kernel's knowledge**
  - Similar to **process table**
  - Used for **thread switching**
  - Tracks thread related information

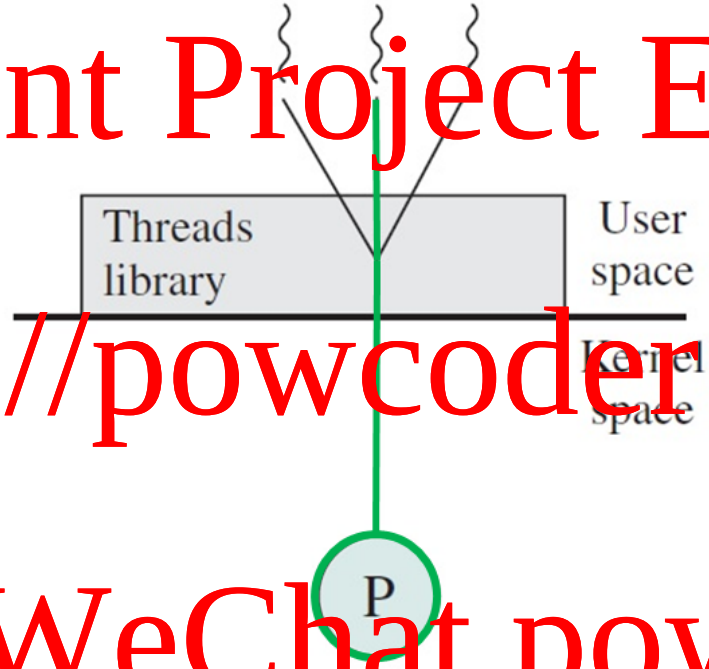
User Threads
Many-to-One



(a) Pure user-level

Figure: User threads (Stallings)

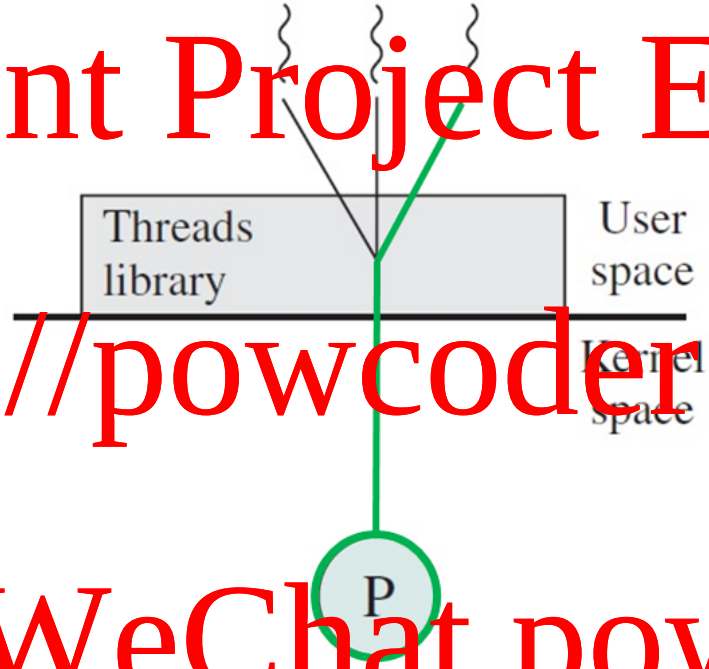
User Threads
Many-to-One



(a) Pure user-level

Figure: User threads (Stallings)

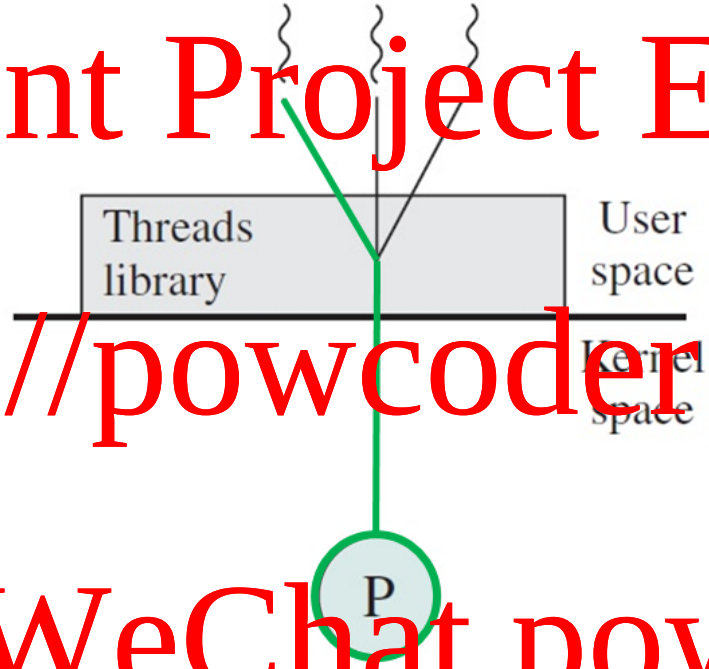
User Threads
Many-to-One



(a) Pure user-level

Figure: User threads (Stallings)

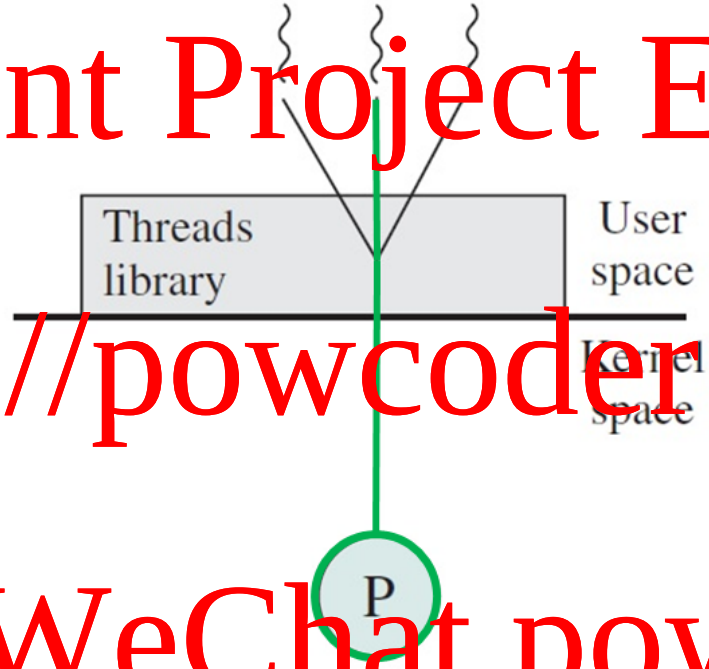
User Threads
Many-to-One



(a) Pure user-level

Figure: User threads (Stallings)

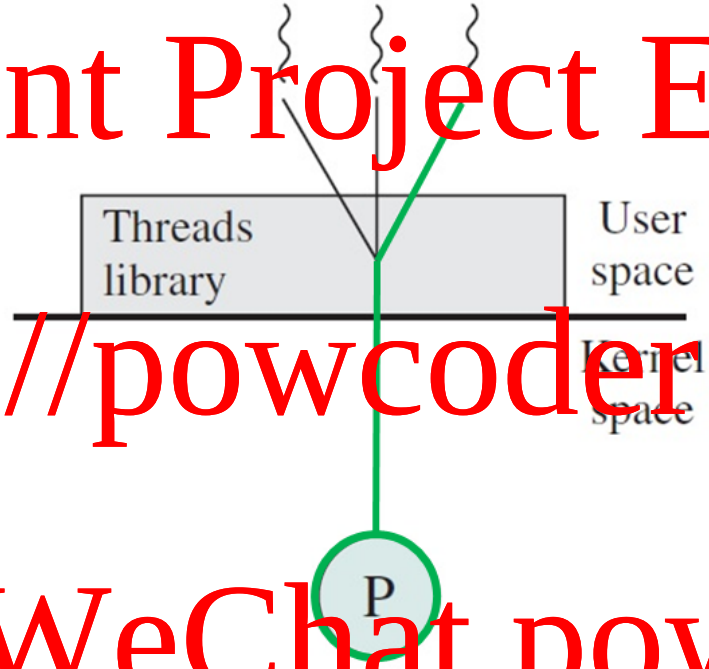
User Threads
Many-to-One



(a) Pure user-level

Figure: User threads (Stallings)

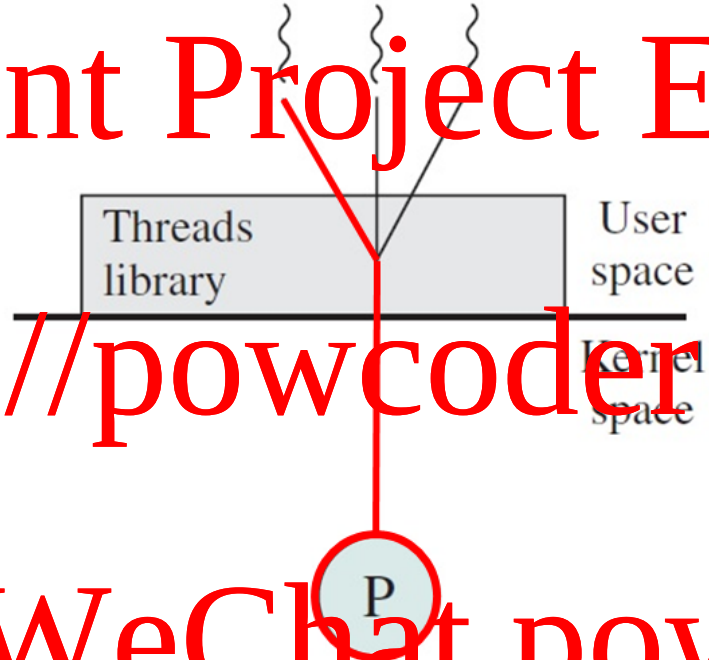
User Threads
Many-to-One



(a) Pure user-level

Figure: User threads (Stallings)

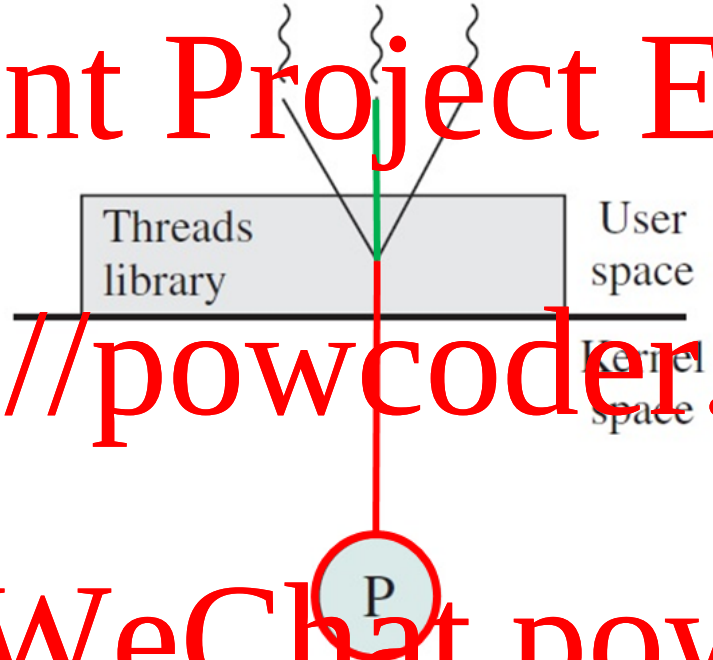
User Threads
Many-to-One



(a) Pure user-level

Figure: User threads (Stallings)

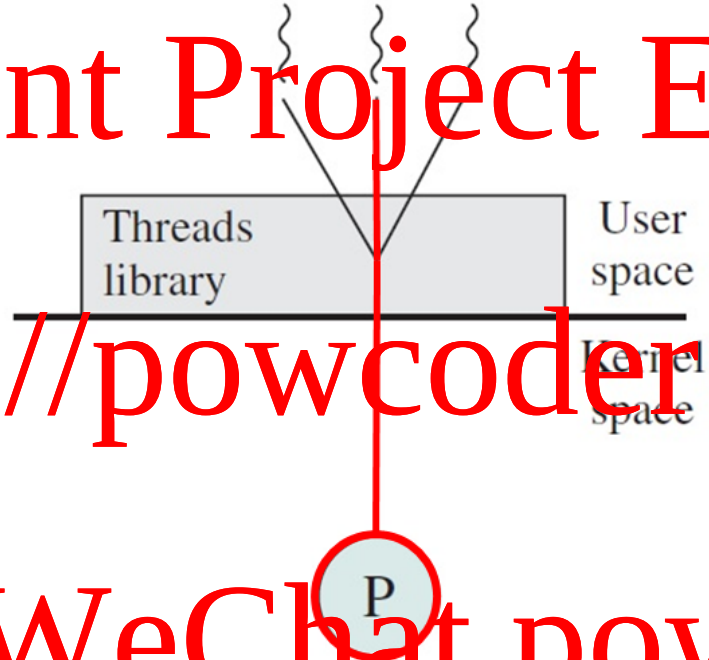
User Threads
Many-to-One



(a) Pure user-level

Figure: User threads (Stallings)

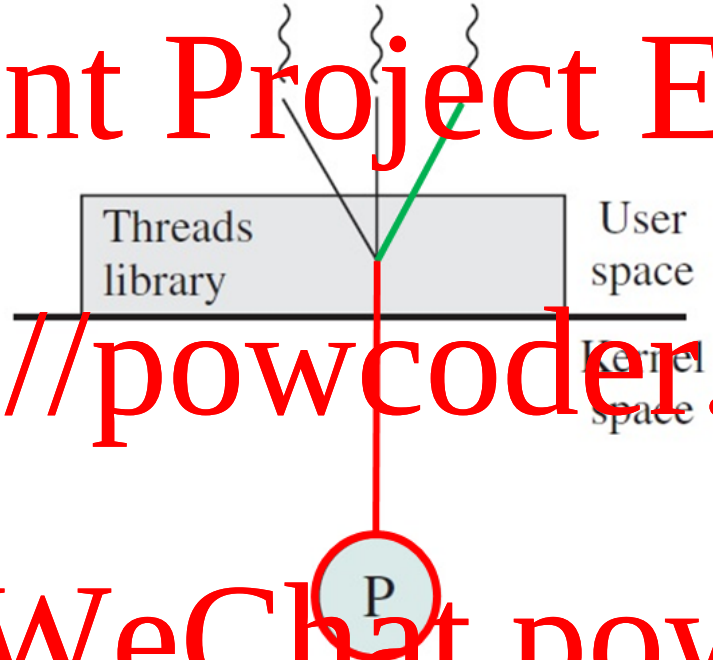
User Threads
Many-to-One



(a) Pure user-level

Figure: User threads (Stallings)

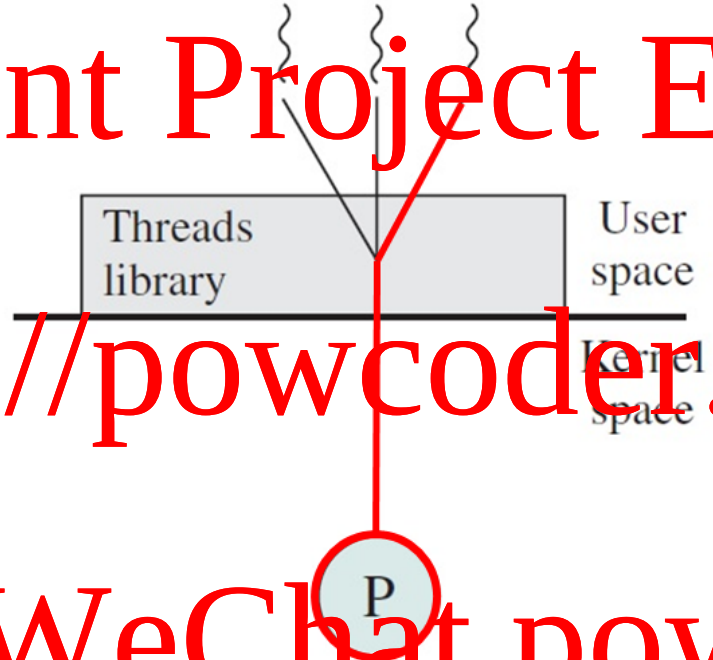
User Threads
Many-to-One



(a) Pure user-level

Figure: User threads (Stallings)

User Threads
Many-to-One



(a) Pure user-level

Figure: User threads (Stallings)

User Threads

Many-to-One

- Advantages:
  - Threads are in user space (i.e., **no mode switches** required)
  - **Full control** over the thread scheduler
  - **OS independent** (threads can run on OS that do not support them)
- Disadvantages:
  - **Blocking system calls** suspend the entire process (user threads are mapped onto a single process, managed by the kernel)
  - **No true parallelism** (a process is scheduled on a single CPU)
  - **Clock interrupts** are non-existent (i.e. user threads are non-preemptive)
  - **Page faults** result in blocking the process

User Threads

Many-to-One

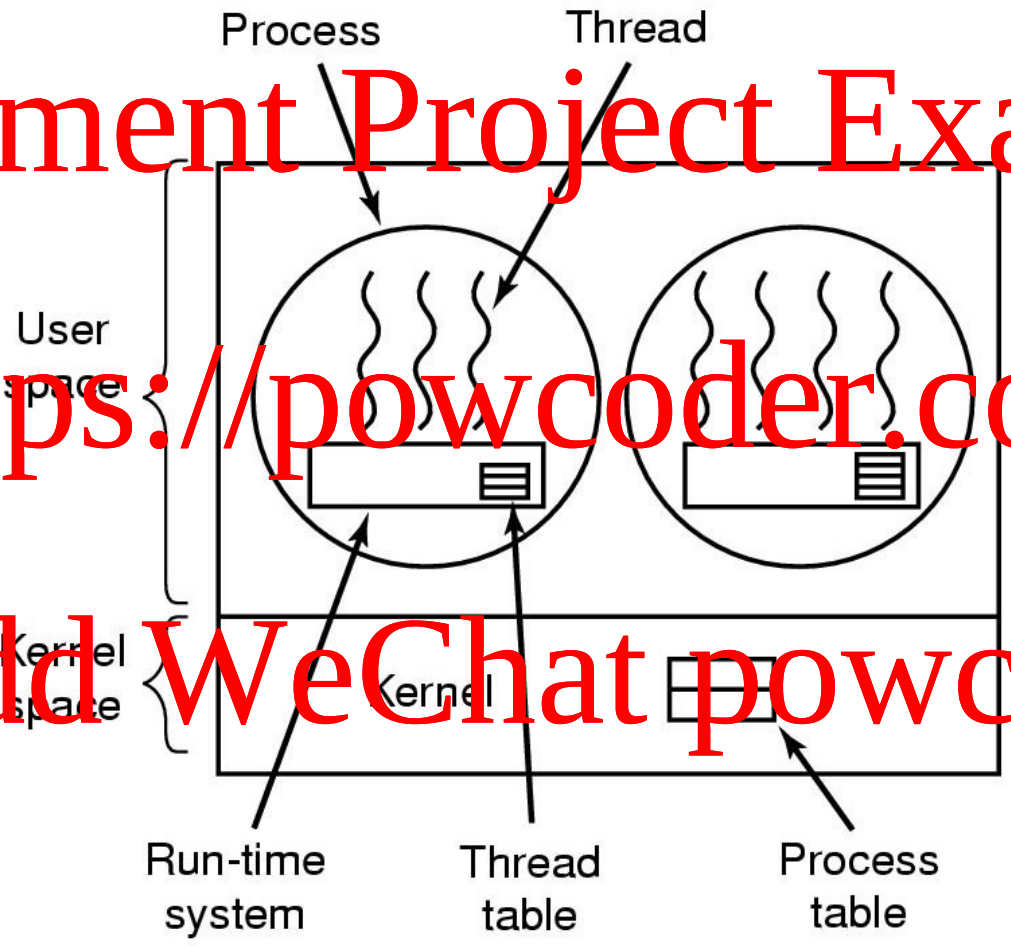


Figure: User threads (Tanenbaum 2014)

Kernel Threads

One-to-One

- The **kernel manages** the threads, user application accesses threading facilities through **API** and **system calls**
  - **Thread table** is in the kernel, containing thread control blocks (subset of process control blocks)
  - If a **thread blocks**, the kernel chooses thread from same or different process ($\leftrightarrow$ user threads)

- Advantages:
  - **True parallelism** can be achieved
  - No run-time system needed

- Frequent **mode switches** take place, resulting in lower performance
- Windows and Linux apply this approach

Kernel Threads

One-to-One

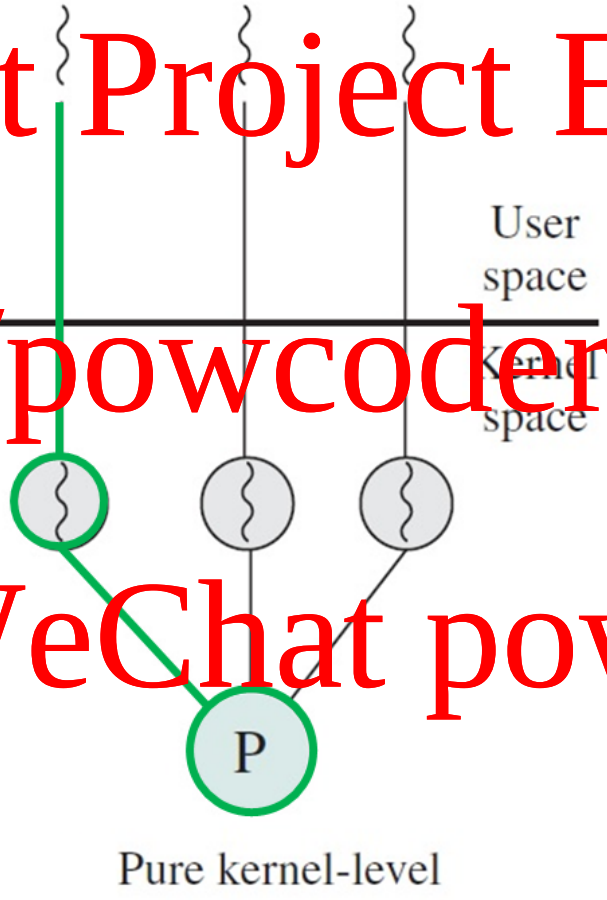


Figure: Kernel threads (Stallings 2014)

Kernel Threads

One-to-One

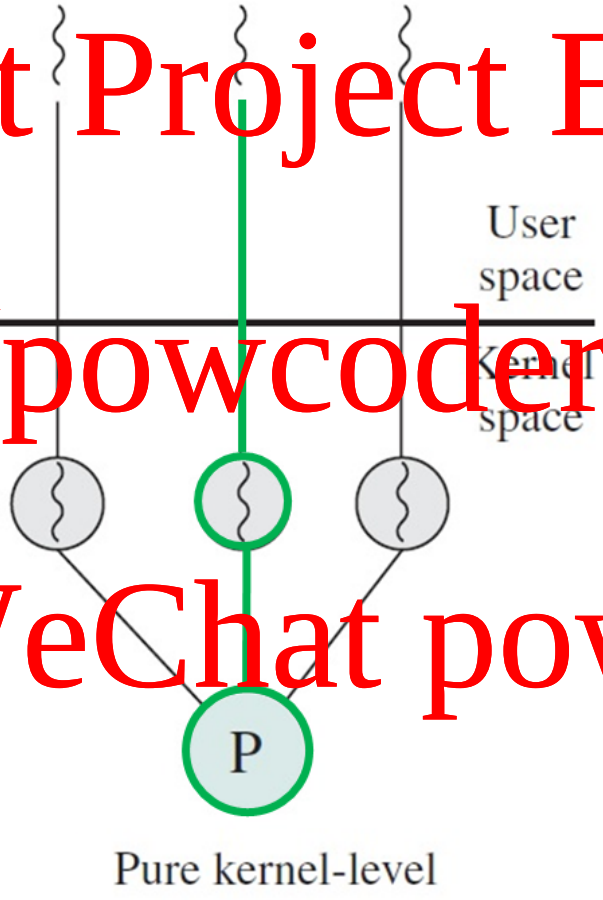


Figure: Kernel threads (Stallings 2014)

Kernel Threads

One-to-One

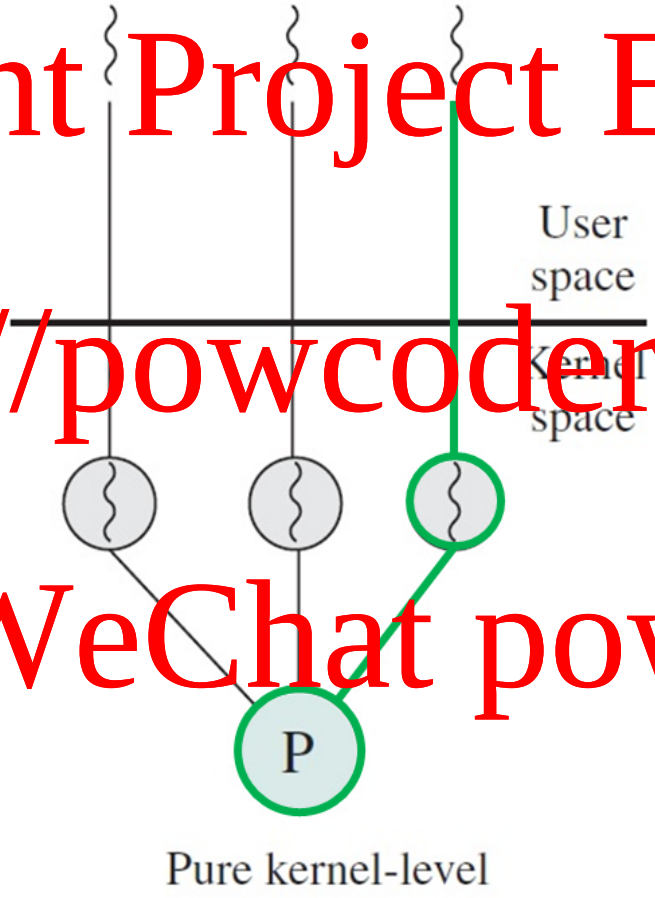


Figure: Kernel threads (Stallings 2014)

Kernel Threads

One-to-One

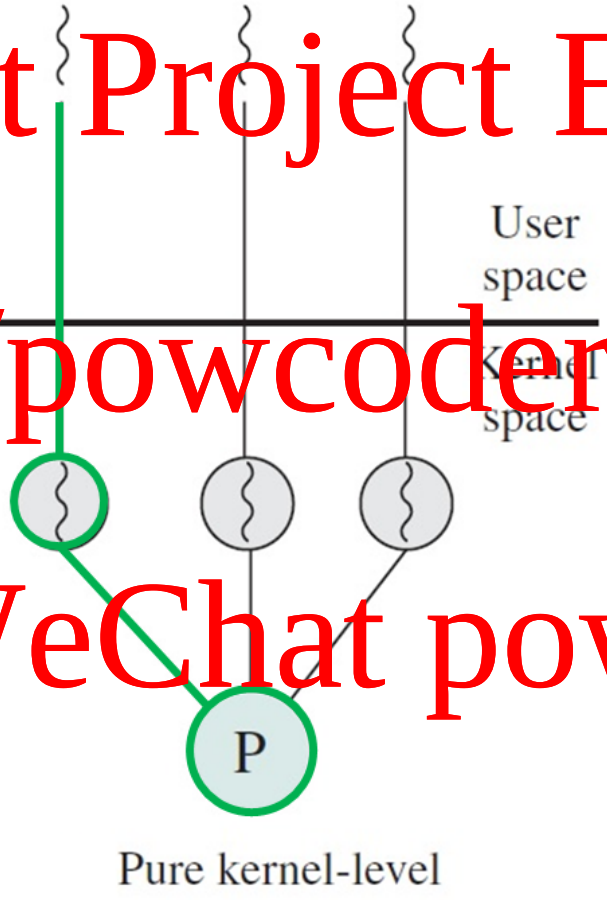


Figure: Kernel threads (Stallings 2014)

Kernel Threads

One-to-One

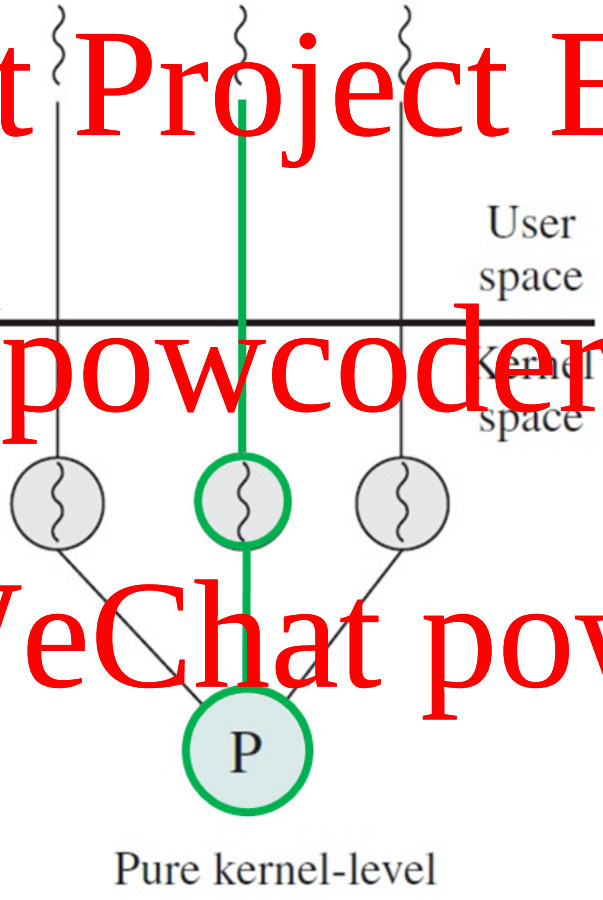


Figure: Kernel threads (Stallings 2014)

Kernel Threads

One-to-One

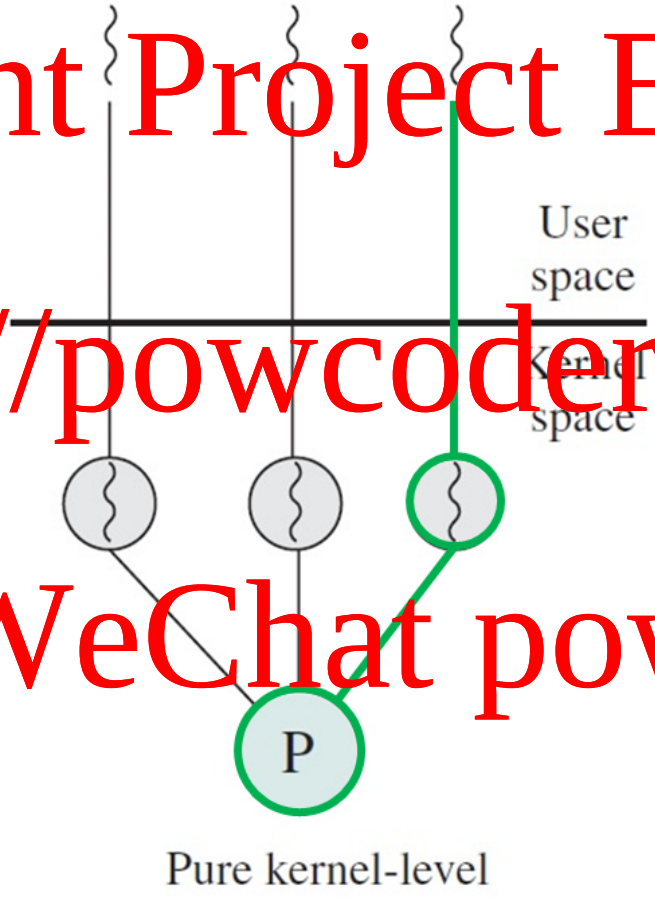


Figure: Kernel threads (Stallings 2014)

Kernel Threads

One-to-One

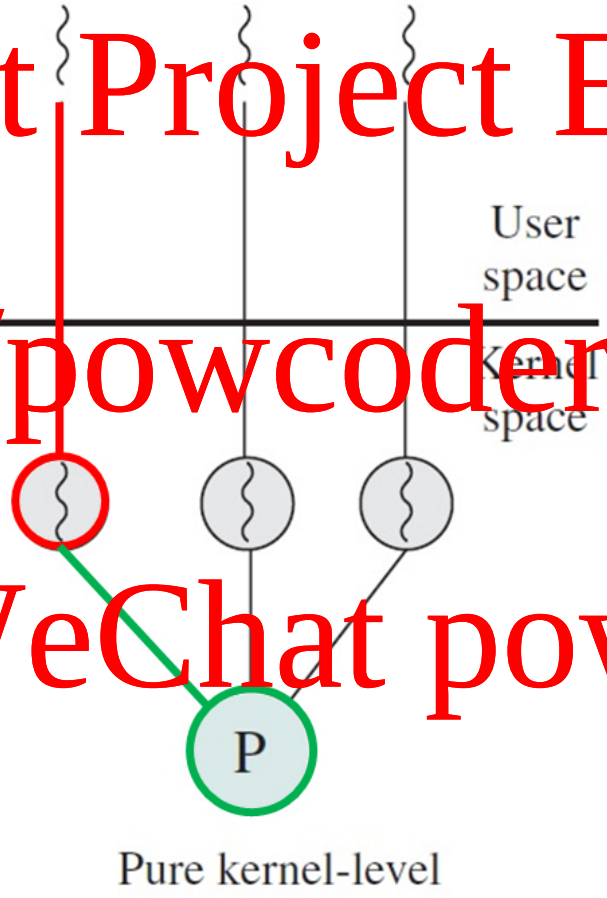


Figure: Kernel threads (Stallings 2014)

Kernel Threads

One-to-One

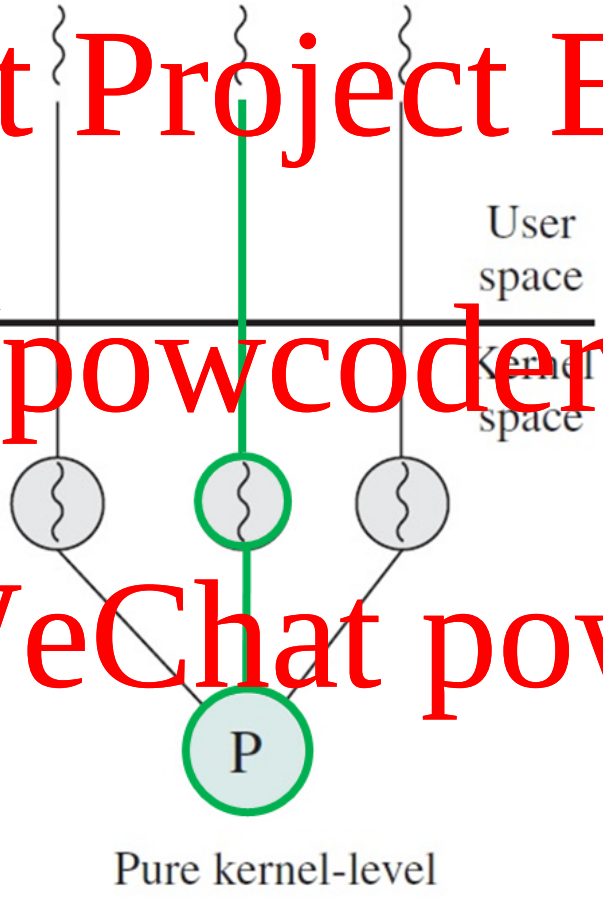


Figure: Kernel threads (Stallings 2014)

Kernel Threads

One-to-One

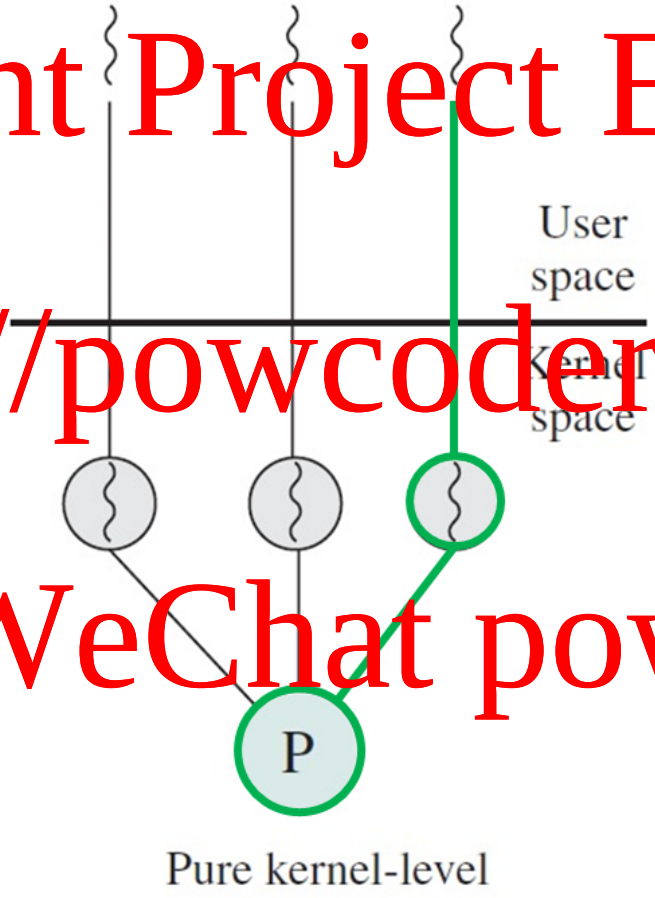


Figure: Kernel threads (Stallings 2014)

Kernel Threads

One-to-One

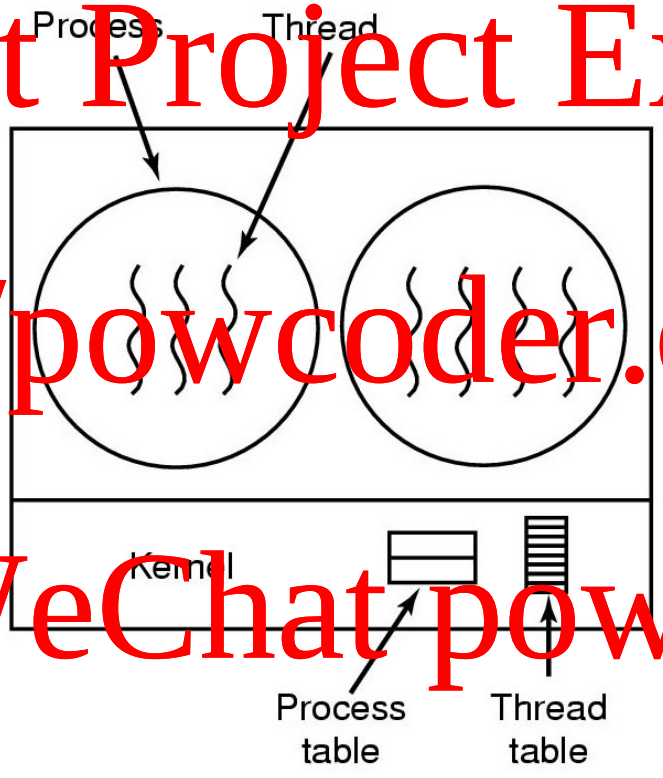


Figure: Kernel threads (Tanenbaum 2014)

Performance

User Threads vs. Kernel Threads vs. Processes

- Null fork: the overhead in creating, scheduling, running and terminating a null process/thread
- Signal wait: overhead in synchronising threads

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

Figure: Comparison, in μs (Stallings)

Hybrid Implementations

Many-to-Many

- User threads are **multiplexed** onto kernel threads
- Kernel sees and schedules the kernel threads (a limited number)
- User application sees user threads and creates/schedules these (an "unrestricted" number)

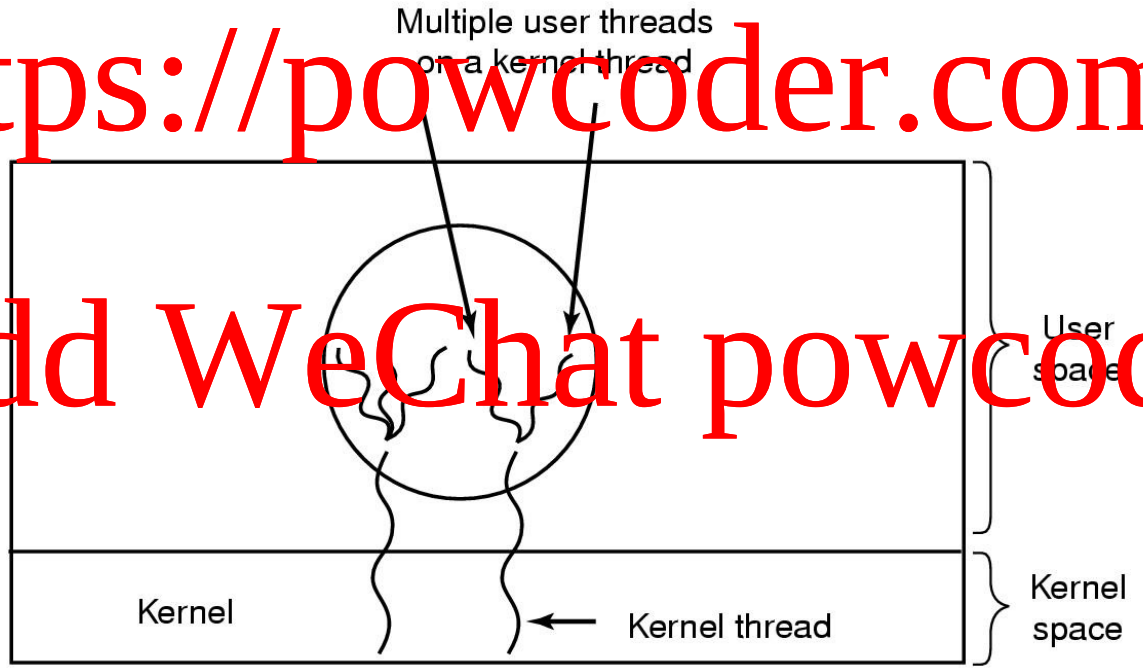


Figure: Kernel threads (Tanenbaum 2014)

Comparison
Thread Implementations

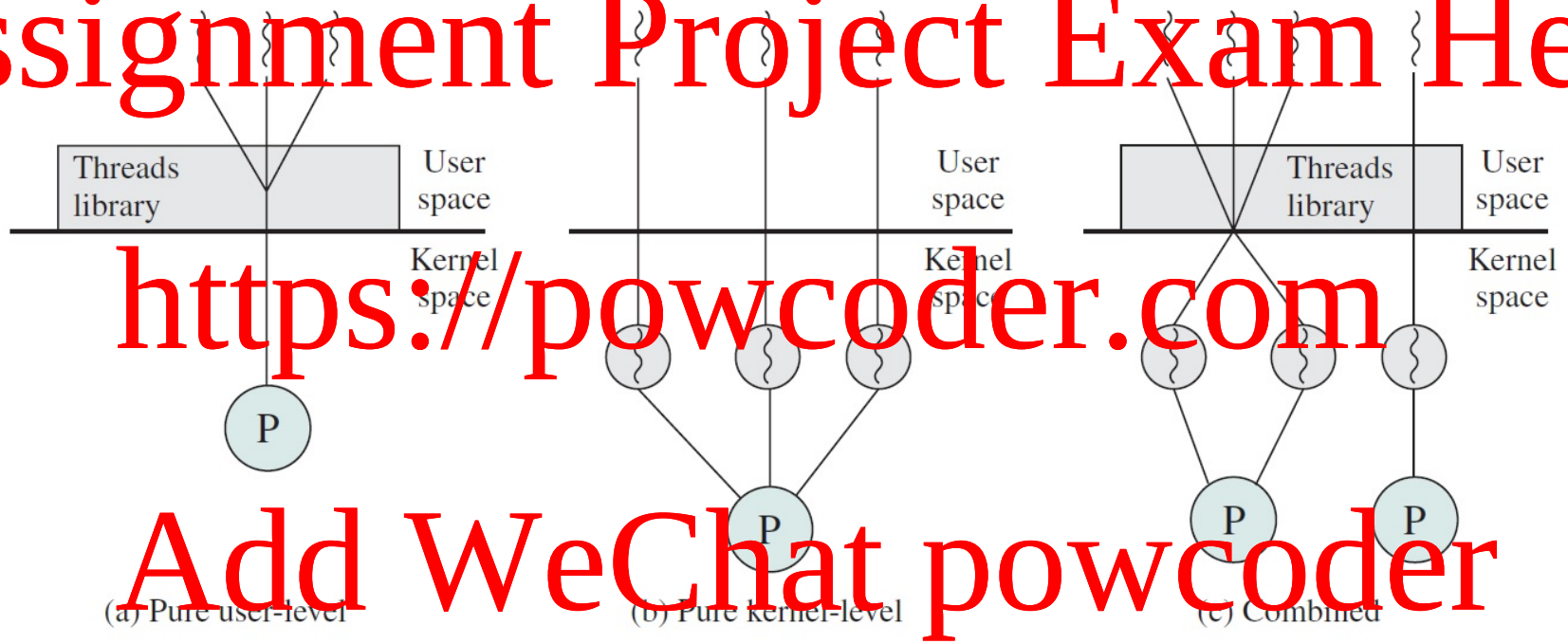


Figure: Comparison (Stallings)

Exam 2013-2014: In which situations would you favour user level threads? In which situation would you definitely favour kernel level threads?

- Thread libraries provide an **API/interface for managing threads** (e.g. creating, running, destroying, synchronising, etc.)
- Thread libraries can be implemented:
  - Entirely in **user space** (i.e. user threads)
  - Based on **system calls**, i.e., rely on the kernel for thread implementations
- Examples of thread APIs include **POSIX's PThreads**, Windows Threads, and Java Threads
  - The PThread specification can be implemented as user or kernel threads

POSIX Threads

Overview

- POSIX threads are a **specification** that **"anyone" can implement**, i.e., it defines a set of APIs (function calls, over 60 of them) and what they do
- Core functions of PThreads include:

Function Call	Summary
`pthread_create`	Create new thread
`pthread_exit`	Exit existing thread
`pthread_join`	Wait for thread with ID
`pthread_yield`	Release CPU
`pthread_attr_init`	Thread Attributes (e.g. priority)
`pthread_attr_destroy`	Release Attributes

Table: PThread examples

- More detailed descriptions can be found using `man function_name` on the command line

POSIX Threads

Example

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUMBER_OF_THREADS 10

void * hello_world(void * tid)
{
  printf("HELLO from thread %d\n", *((int *) tid));
  pthread_exit(NULL);
}

int main(int argc, char * argv[])
{
  int aiIds[] = {1,2,3,4,5,6,7,8,9,10};
  pthread_t threads[NUMBER_OF_THREADS];
  int i;
  for(i = 0; i < NUMBER_OF_THREADS; i++)
  {
    if(pthread_create(&threads[i], NULL, hello_world, (void *) &(aiIds[i])) == -1)
    {
      printf("Creating thread %d failed", i);
      exit(-1);
    }
  }
  for(i = 0; i < NUMBER_OF_THREADS; i++)
    pthread_join(threads[i], NULL);
  return 0;
}
```

- What are **threads** and why are they useful
- Different **thread** implementations from an OS perspective
- The principle/idea behind **PThreads**