Operating Systems

Lecture 2b

Dr Ronald Grau School of Engineering and Informatics Spring term 2018

Previous Lecture

Processes

- Bootstrapping
- Processes
  - Creation
  - Management
  - Execution
  - Termination

Quick knowledge check

A process is a program in execution

- What does it mean when we say processes run concurrently?
- What kind of information is stored in a process image and what is this used for?
- How do processes share a computer's memory?
- What is exceptional control flow (ECF)?
- What is the significance of a mode switch in ECF?
- What happens during a process switch?
- What are the basic states a process can assume during execution?
- What is swapping and what is it useful for?
- What are the reasons why a process might terminate?

Today

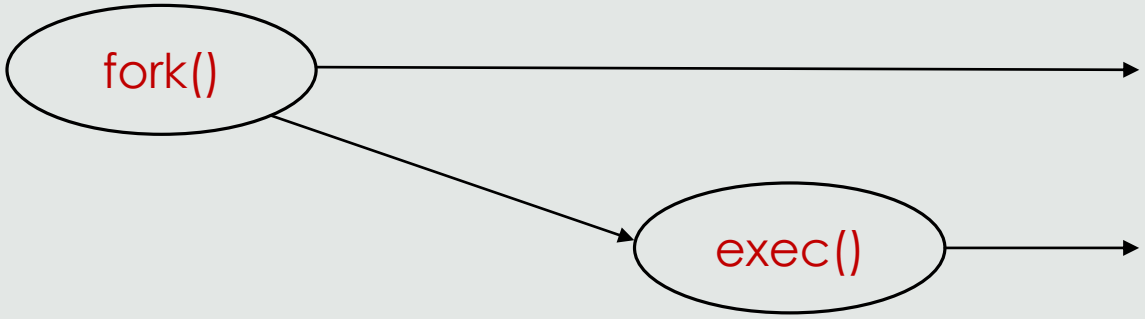
Threads

- Process mortality revisited
- Threads
  - Distinction to processes
  - Single- vs multi-threaded processes
  - Advantages of threads
  - Thread implementations
  - Multi-threading models

Process mortality revisited

Recall: Processes can create other processes:

- **fork** creates a copy of the (now) parent process
- **exec** transfers control to the child process

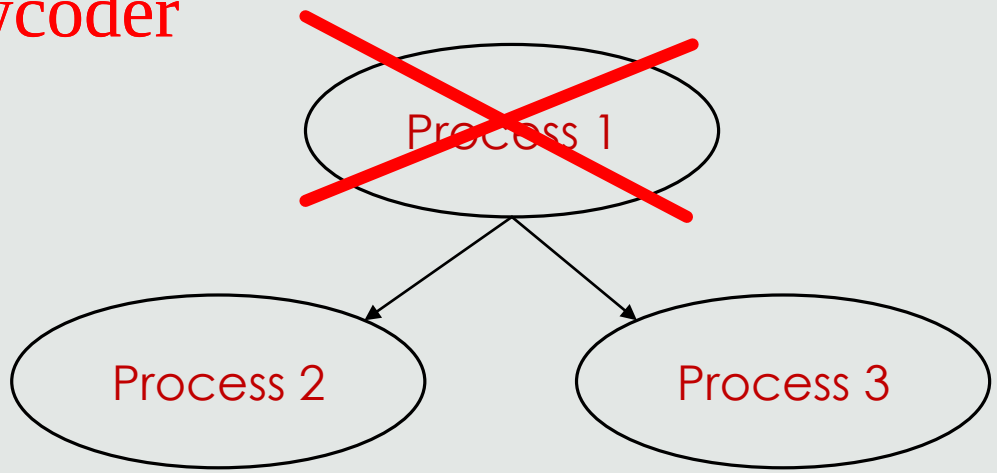


Process mortality revisited

Recall: Processes terminate when:

- Their program finished - job done.
- There is an exception handling routine triggered in the program.
- There is an unexpected fatal error.
- They are killed by another process.

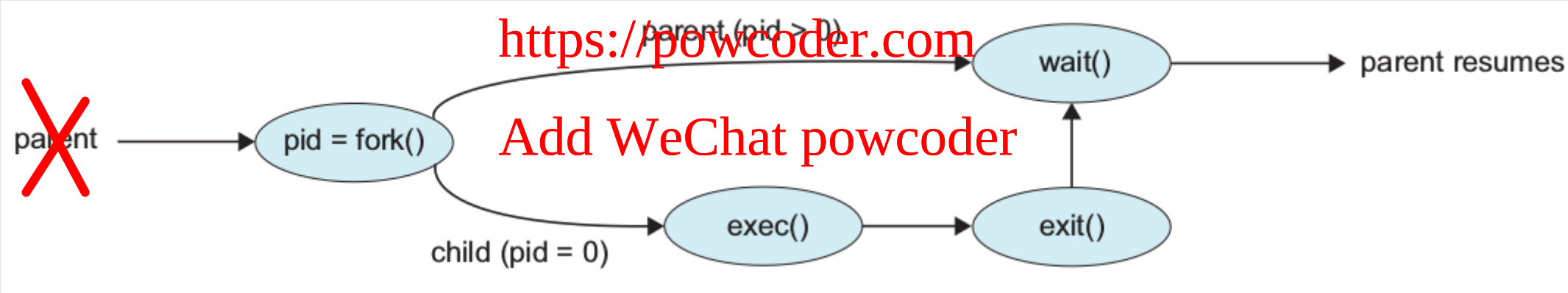
What happens to Process 2 and 3 when Process 1 terminates?



Process mortality revisited

What happens to a child if their parent terminates?

- Parent processes usually wait for their children to finish.



The parent process waits until the child process exits

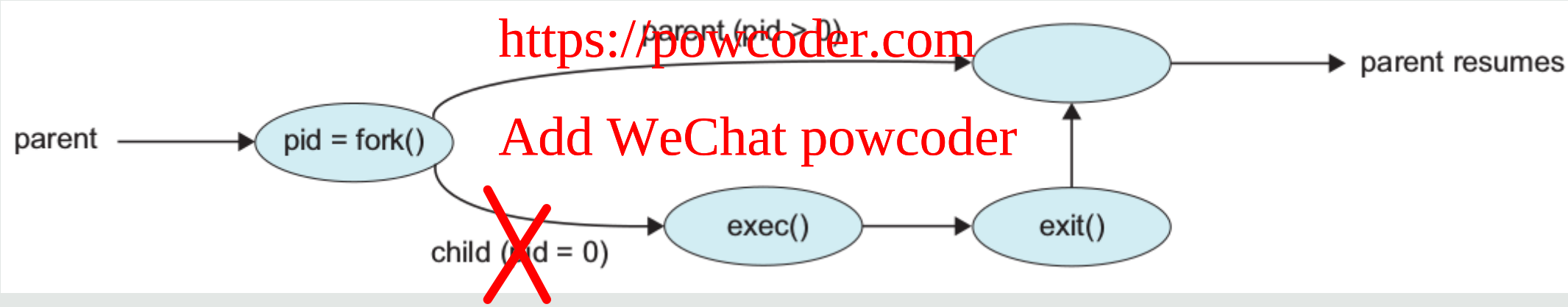
What happens if the parent process terminates?

Any existing child processes become "**orphans**", and get a "foster parent" - they are reassigned to the init process (LINUX)

Process mortality revisited

What happens if the child process terminates while no-one is waiting?

- Scenario: The child process terminates before their parents can call wait().



The parent process does not wait on the child process

Child processes that exit before their parent calls wait() become defunct, or "**zombies**". They cannot be killed but remain on the process table in terminated state until the parent calls wait() to retrieve their exit status.

Processes and Threads

A brief comparison

- A process presents a unit for resource management and scheduling
  - Process image / Process Control Block, virtual memory address space
  - Memory protection, file and I/O management
- However, having many processes can be disadvantageous due to process management and resource allocation overheads.

Processes and Threads

A brief comparison

- A process presents a unit for resource management and scheduling
  - Process image / Process Control Block, virtual memory address space
  - Memory protection, file and I/O management
- However, having many processes can be disadvantageous due to process management and resource allocation overheads.

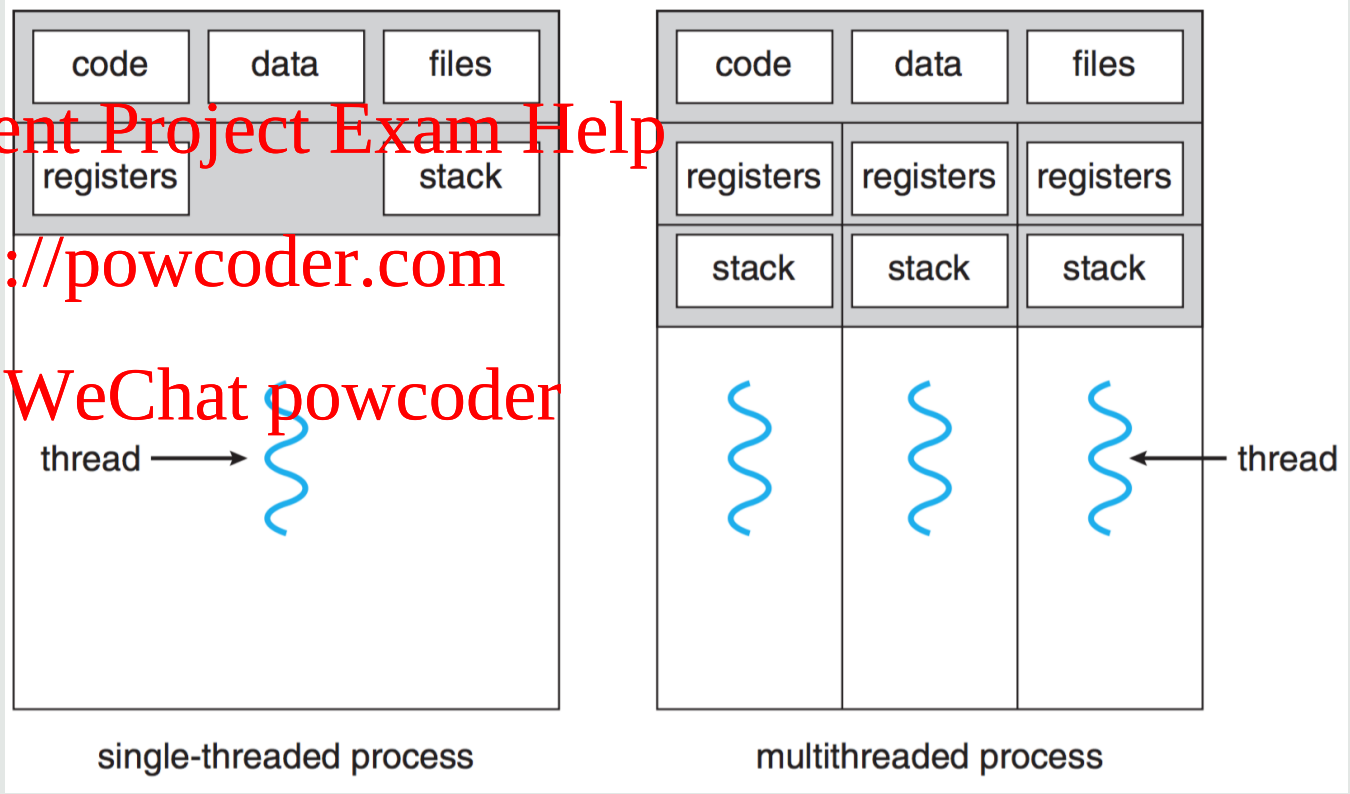
- A thread is a unit for scheduling only
  - Thread Control Block: Thread ID, state, context
- Threads are more light-weight than processes. They can access process resources.

Key advantage: We can have **multiple** threads at relatively low expense.

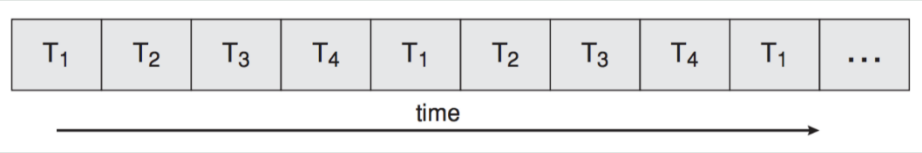
Processes and Threads

Single- vs multi-threaded processes

- Threads enable processes to maintain multiple threads of control
- A thread still needs to maintain some information (registers, stack memory)



Interleaved process thread execution on a (single-core) CPU



Processes and Threads

Multiple threads on multiple cores

- Benefits of multi-threading are amplified on a multi-core CPU architecture.
- Thread can run in parallel on different cores.
- Single-threaded processes can only run on a single CPU core, even if 15 others are available.
- Multi-threading on multi-core computers increases concurrency.
- Single-core CPUs effectively switch between threads as if they were processes.

Processes and Threads

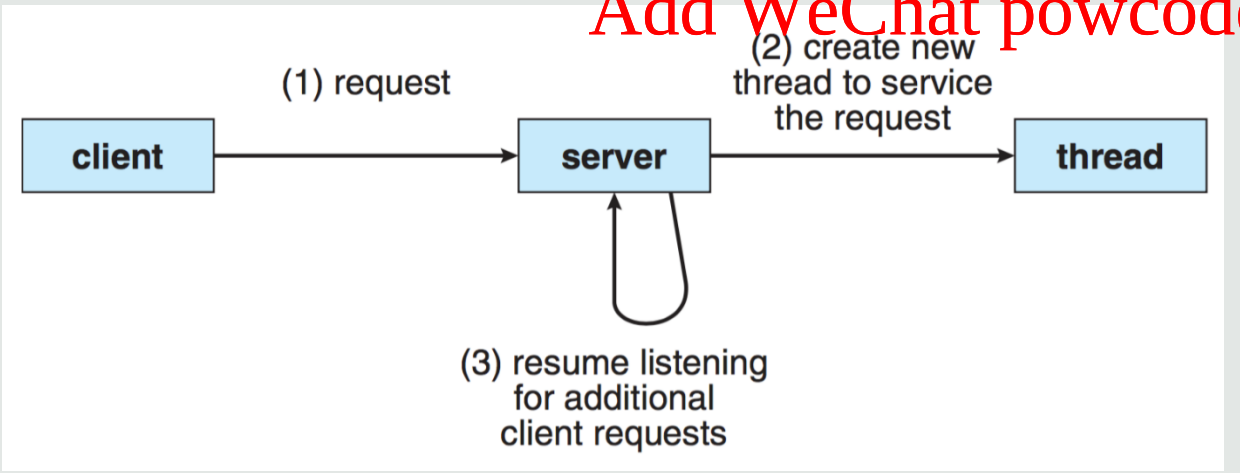
General advantages of threads

- Lightweight management
  - Creation
  - Context-switching
  - Termination
- Lightweight communication
  - Shared resources within the same address space
  - Direct communication within the process

Processes and Threads

General advantages of threads #2

- Application responsiveness
  - User requests have less impact on execution performance
  - Expensive computation can run separately



Example: A web server

Threads

Thread states

- Only three states required:
  - Ready (waiting for execution)
  - Running (executing code)
  - Blocked (waiting for something to happen, I/O)
- What happens to a thread when a process is suspended / terminates?

Thread implementations

User-level threads

- Implemented as a user-space threading library
- The threading library handles
  - Creation and termination of threads
  - Thread communication
  - Thread scheduling
  - Saving and restoring thread contexts

Thread implementations

User-level threads #2

- In principle they are invisible to the kernel, and OS-agnostic
- Thread switching happens in user mode
- Thread scheduling managed by the library
- When a thread goes into blocked state, the entire process is also blocked
- Multi-core execution not possible

Thread implementations

Kernel-level threads

- Thread API provided by kernel (triggers system calls)
- Specific to the OS
- Thread switching by kernel
- Thread-wise scheduling
- Thread-wise blocking
- Threads of the same process can execute on different cores

- But: More management effort, any thread switch requires mode switching

Thread implementations

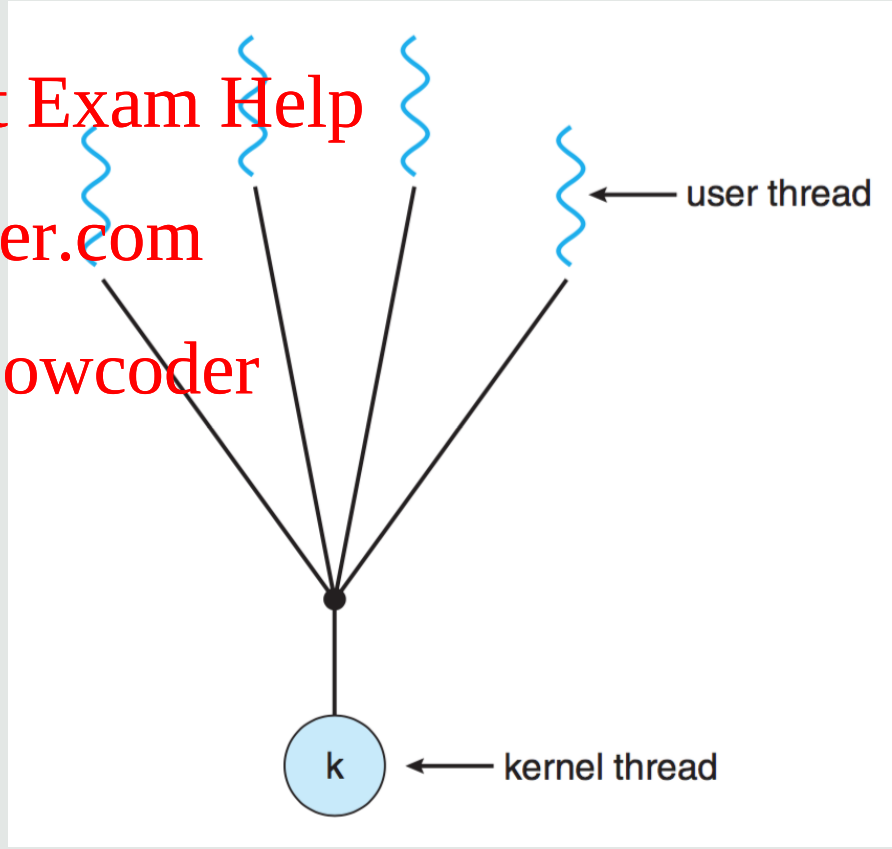
Example threading libraries

- POSIX Pthreads extension (user- or kernel-level)
- Windows thread lib (kernel-level)
- Java lib / JVM → mapping to native threading model through virtual machine

Multi-threading models

N:1 model

- One kernel thread running several user threads
- But: One thread blocking disables all the others

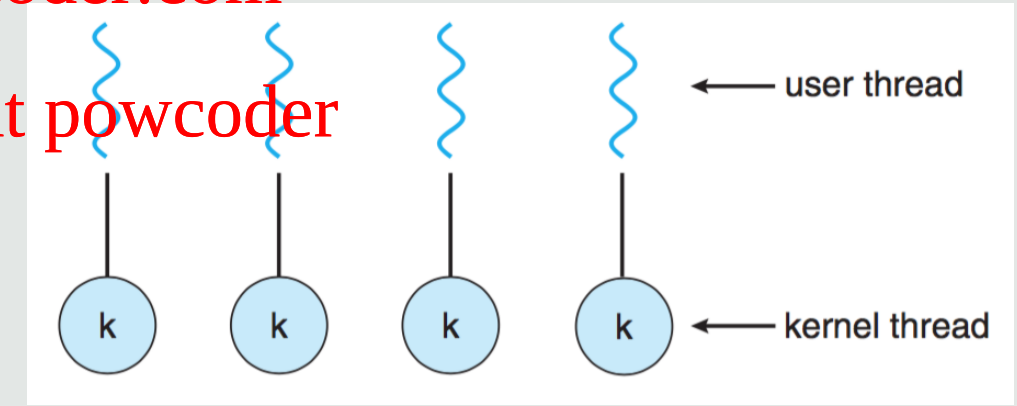


Many-to-one model of multi-threading

Multi-threading models

1:1 model

- User threads are mapped 1:1 to kernel threads
- Very common model, better concurrency than N:1
- Runs well on multiple cores
- But: Limited number of threads that can be created

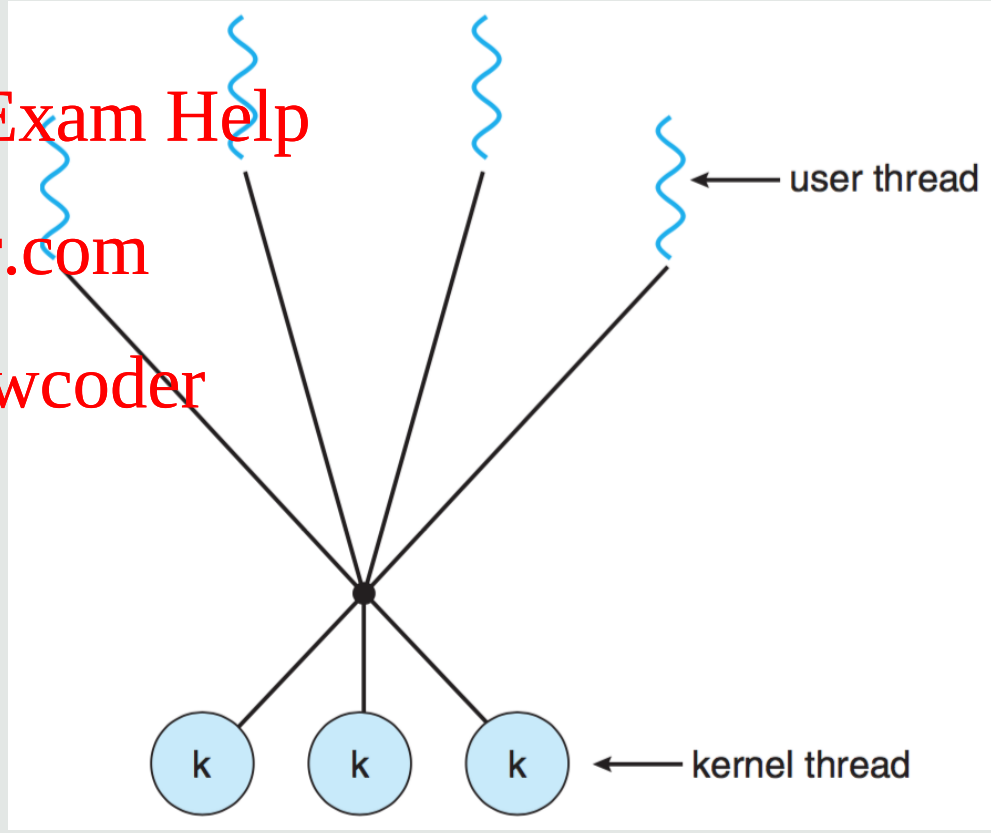


One-to-one model of multi-threading

Multi-threading models

N:M model

- Multiplexes many user level threads to a smaller or equal number of kernel threads
- More flexible than the other models, combining their benefits
- Number of kernel threads may be specific to a particular platform or application
- But: Higher communication & management effort



N:M multi-threading model

Multi-threading models

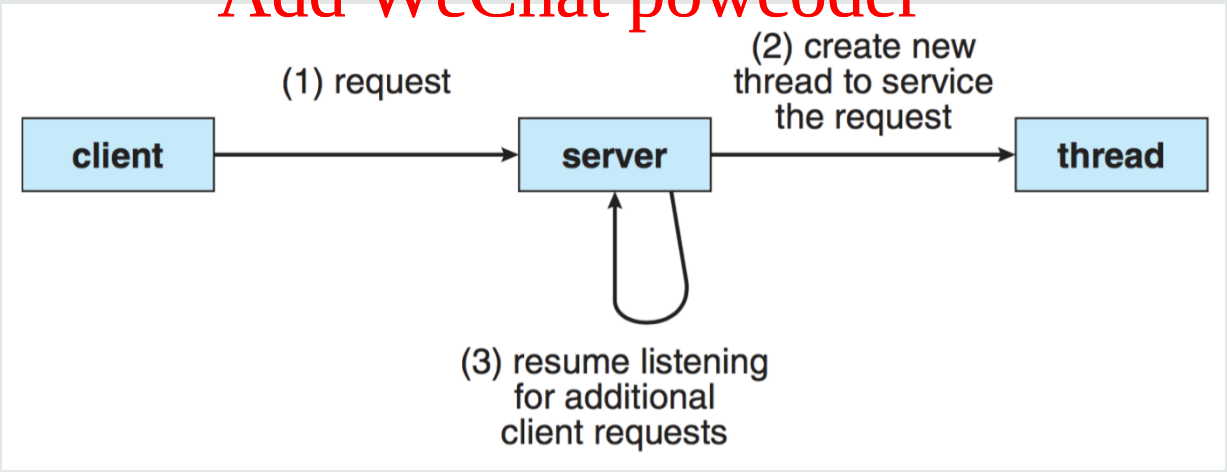
Scheduler Activations in N:M

- Kernel notifies the user-thread library (upcall)
  - When a thread blocks, the user-thread library schedules another thread
- Early versions of NetBSD and Solaris used this, now replaced by 1:1

Multi-threading limits

Resources are not infinite!

- Going back to our web server from slide #13, what would happen if the server
  - Has exceeded the limit and cannot create any more threads?
  - Has very many requests, i.e. threads running at the same time?

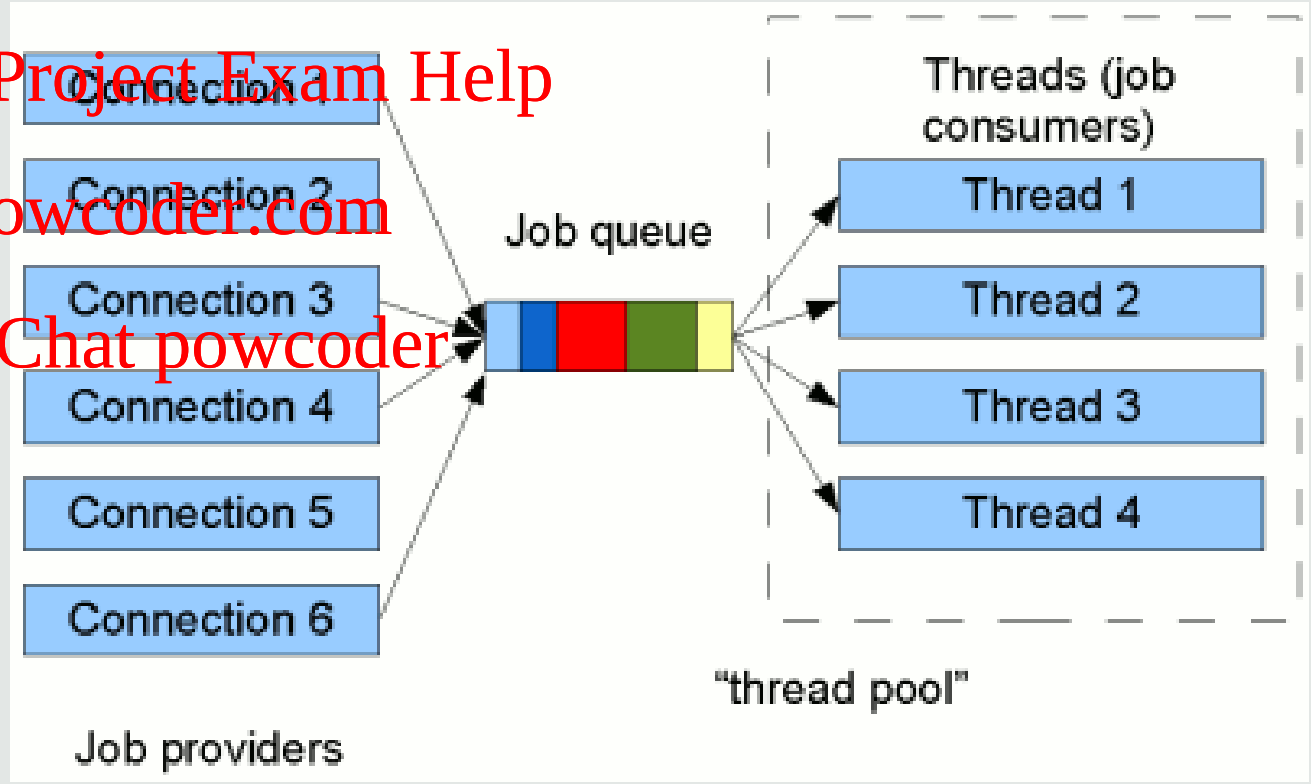


Every request will spawn a new thread

Implicit threading

Better management through Thread pools

- A number of "workers" is created in advance, waiting for jobs (*job consumers*)
- Incoming requests (*job providers*) are scheduled (*job queue*)
- If a request arrives, a worker is activated
- Once a request has been serviced, the worker goes back to waiting

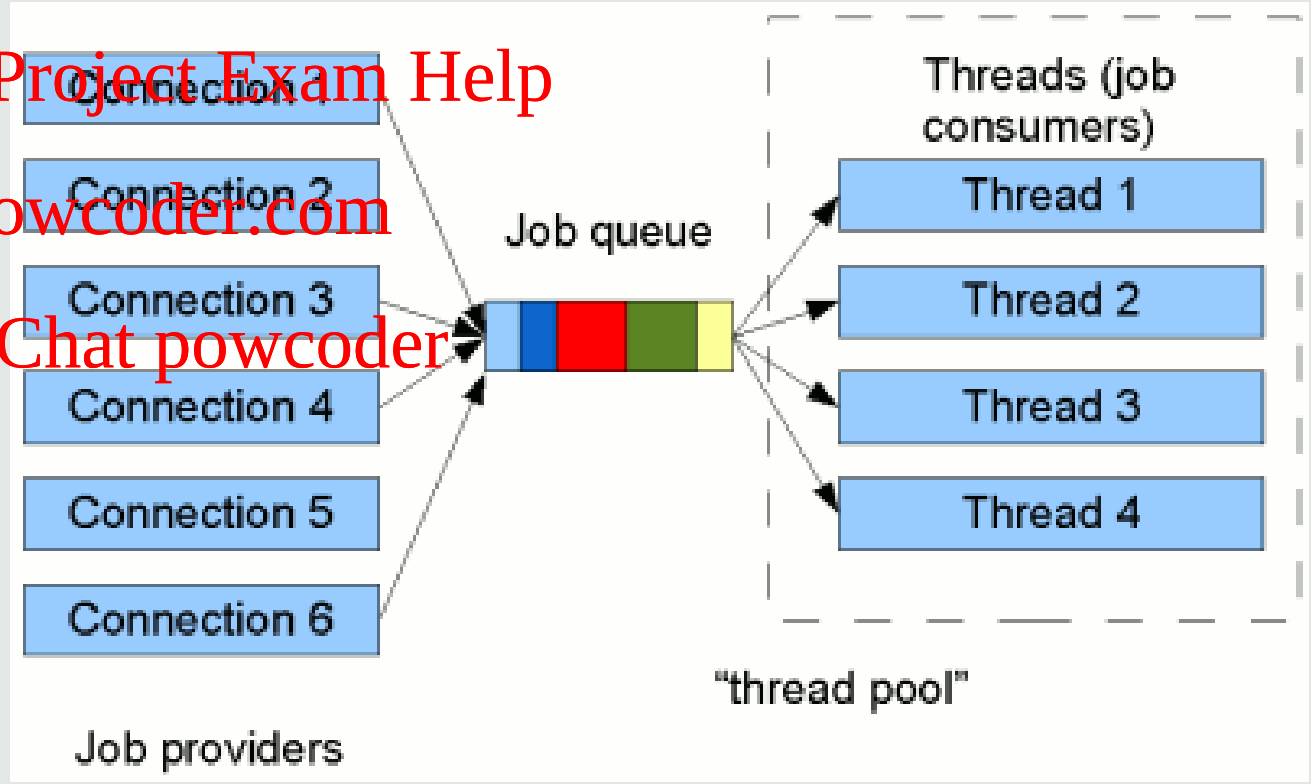


A thread pool

Implicit threading

Advantages of Thread pools

- Much less overhead as threads do not need to be created and terminated on the fly, leading to overall faster response times.
- The risk of exhausting the system is taken care of as the number of workers is predetermined.
- At the same time, the number of job consumers can be flexible, up to a certain limit.
- The smaller overhead allows more flexibility in how jobs are serviced (time schedules).



A thread pool

Threads

Example: POSIX pthreads

- POSIX: set of interface specifications for UNIX-like systems
- Also has instructions for threads

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run

Example POSIX instructions for thread handling

Threads

Example: POSIX pthreads


```
#define N 10
int matrix [N] [N] ;
int row sums [N] ;
void *run (void *param) {
    int row= (int) param;
    row sums [row]=0 ;
    for ( int i =0; i<N; i++)
    row sums [row]+=matrix [row][i] ;
}
void main ( ) {
    // put code for initialising the matrix here
    pthread_t threads [N] ;
    for (int i =0; i<N; i ++)
      pthread_create(&threads[i], NULL, run, (*void) i);
    for ( int i =0; i<N; i++)
      pthread_join ( threads [i] , NULL) ;
}
```

Summary

Threads

- Process mortality revisited
- Threads
  - Distinction to processes (lightweight sub-processes)
  - Single- and multi-threaded processes
  - Implementation
    - User-level threads
    - Kernel-level threads
    - Hybrid threads

Read

- Tanenbaum & Bos., Modern Operating Systems
  - Chapter 2
- Silberschatz et al., Operating System Concepts
  - Chapter 4

Next Lecture

- Introduction
- Operating System Architectures
- Processes
- **Threads - Programming**
- Process Scheduling
- Process Synchronisation
- Deadlocks
- Memory Management
- File Systems
- Input / Output
- Security and Virtualisation