Threads (2)

Dr. Jun Zheng

CSE325 Principles of Operating Systems

9/20/2019

NEW MEXICO TECH
SCIENCE · ENGINEERING · RESEARCH · UNIVERSITY

Threads: Concurrent Servers

- Using **fork()** to create new processes to handle requests in parallel is overkill for such a simple task
- Web server example:

```
– while (1) {
–   int sock = accept();
–   if ((child_pid = fork()) == 0) {
–       Handle client request
–       Close socket and exit
–   } else {
–       Close socket
–   }
– }
```

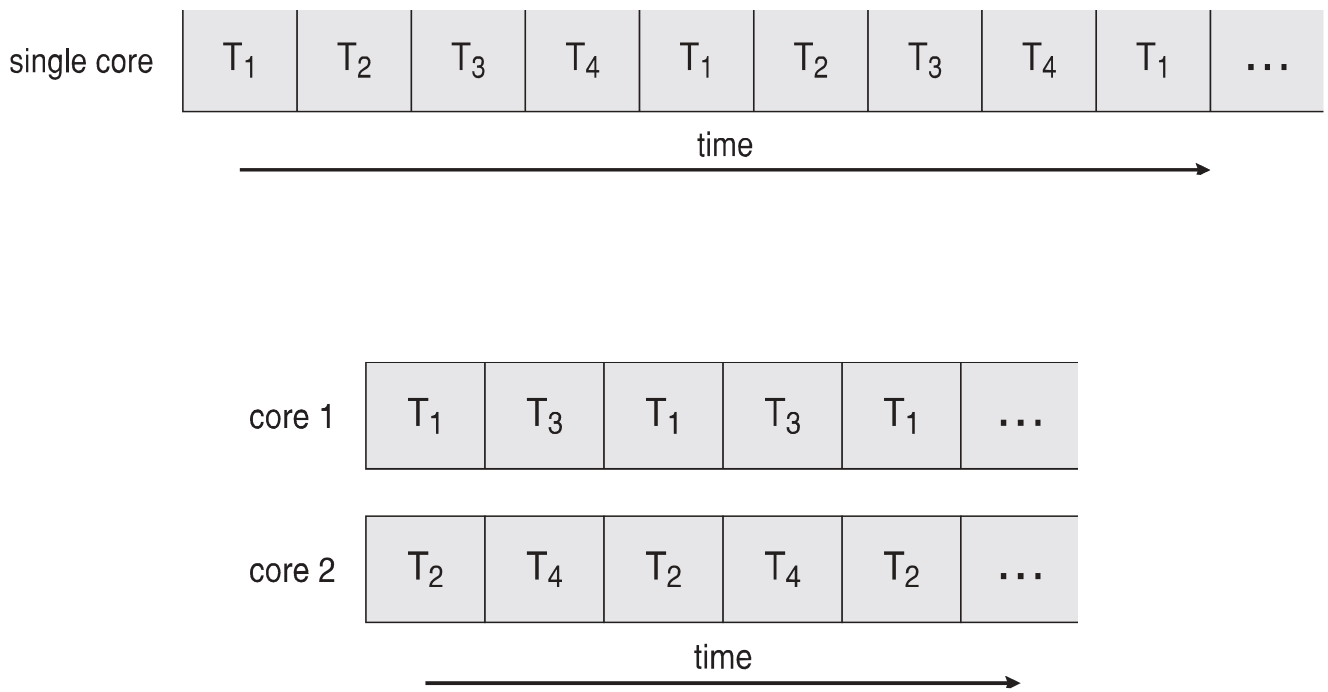
NEW MEXICO TECH
SCIENCE · ENGINEERING · RESEARCH · UNIVERSITY

Threads: Concurrent Servers

- Instead, we can create a new thread for each request

```
web_server() {
  while (1) {
    int sock = accept();
     thread_create(handle_request, sock);
  }
}
handle_request(int sock) {
    Process request
    close(sock);
}
```

Concurrency vs. Parallelism



NEW MEXICO TECH
SCIENCE · ENGINEERING · RESEARCH · UNIVERSITY

Benefits of Multithreaded Programming

- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing –** threads share resources of process, easier than shared memory or message passing
- **Economy –** cheaper than process creation, thread switching lower overhead than context switching
- **Scalability –** process can take advantage of multiprocessor architectures

Question

- If a process exits and there are still threads of that process running, will they continue to run? Please justify your answer.

Answer

No. When a process exits, it takes everything with it—the process structure, the memory space, everything—including threads.

Kernel-Level Threads

- We have taken the execution aspect of a process and separated it out into threads
  - To make concurrency cheaper
- As such, the OS now manages threads *and* processes
  - All thread operations are implemented in the kernel
  - The OS schedules all of the threads in the system
- OS-managed threads are called kernel-level threads or lightweight processes
  - Windows: threads
  - Solaris: lightweight processes (LWP)
  - POSIX Threads (pthreads): PTHREAD_SCOPE_SYSTEM

NEW MEXICO TECH
SCIENCE · ENGINEERING · RESEARCH · UNIVERSITY

Kernel-level Thread Limitations

- Kernel-level threads make concurrency much cheaper than processes
  - Much less state to allocate and initialize
- However, for fine-grained concurrency, kernel-level threads still suffer from too much overhead
  - Thread operations still require system calls
    - Ideally, want thread operations to be as fast as a procedure call
- For such fine-grained concurrency, need even "cheaper" threads

User-Level Threads

- To make threads cheap and fast, they need to be implemented at user level
  - Kernel-level threads are managed by the OS
  - User-level threads are managed entirely by the run-time system (user-level library)
- User-level threads are small and fast
  - A thread is simply represented by a PC, registers, stack, and small thread control block (TCB)
  - Creating a new thread, switching between threads, and synchronizing threads are done via procedure call
    - No kernel involvement
  - User-level thread operations 100x faster than kernel threads
  - pthreads: PTHREAD_SCOPE_PROCESS

User-level Thread Limitations

- But, user-level threads are not a perfect solution
  - As with everything else, they are a tradeoff
- User-level threads are invisible to the OS
  - They are not well integrated with the OS
- As a result, the OS can make poor decisions
  - Scheduling a process with idle threads
  - Blocking a process whose thread initiated an I/O, even though the process has other threads that can execute
- Solving this requires communication between the kernel and the user-level thread manager

Kernel- vs. User-level Threads

- Kernel-level threads
  - Integrated with OS (informed scheduling)
  - Slow to create, manipulate, synchronize
- User-level threads
  - Fast to create, manipulate, synchronize
  - Not integrated with OS (uninformed scheduling)
- Understanding the differences between kernel- and user-level threads is important

Question

- A disadvantage of ULTs is that when a ULT executes a blocking system call (e.g. a I/O call), not only is that thread blocked, but all of the threads within the process are blocked. Why?

NEW MEXICO TECH
SCIENCE · ENGINEERING · RESEARCH · UNIVERSITY

Answer

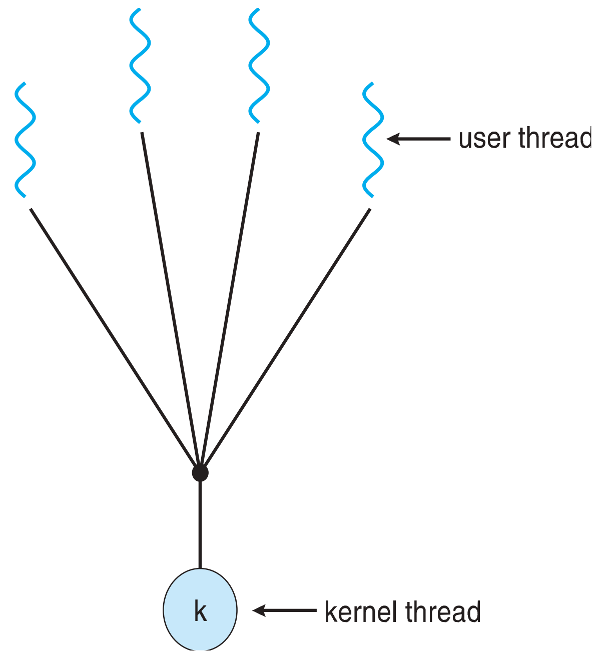
- Because, with ULTs, the thread structure of a process is not visible to the operating system, which only schedules on the basis of processes.
- For a blocking system call like a I/O call, the control is transferred to the kernel. The kernel invokes the I/O action, places process in the blocked state, and switches to another process.

Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

Many-to-One

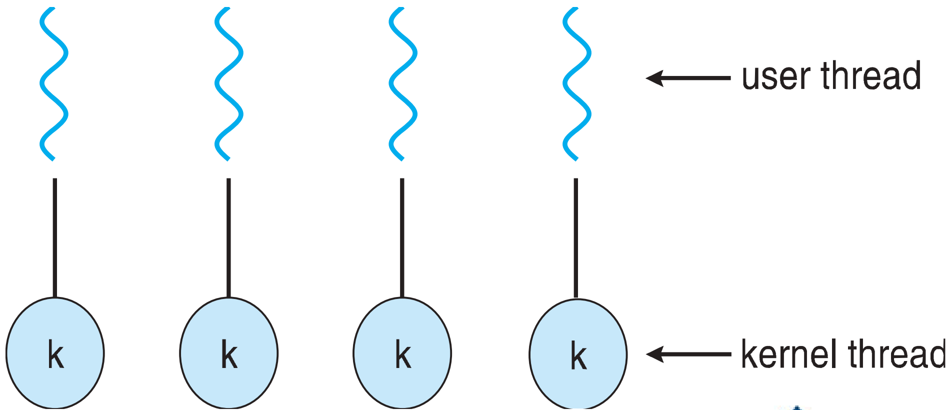
- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on muticore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**



NEW MEXICO TECH
SCIENCE · ENGINEERING · RESEARCH · UNIVERSITY

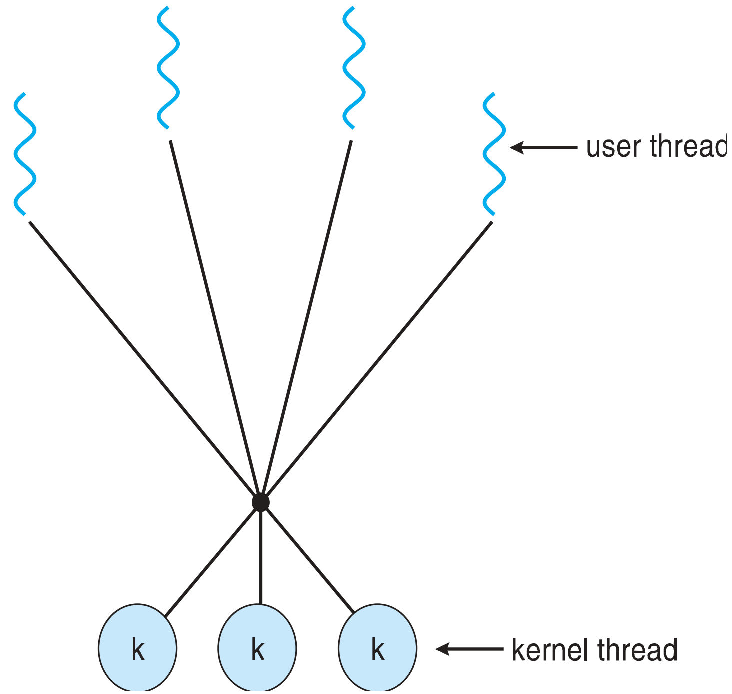
One-to-One

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later



Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows with the *ThreadFiber* package
- Extremely difficult to implement



Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier