Threads and Multi-processor/core Scheduling

OPS Lecture 6, G53OPS/G52OSC

Geert De Maere
(Jason Atkin – OSC)
Geert.DeMaere@Nottingham.ac.uk

University Of Nottingham
United Kingdom

2015

Recap
Last Lecture

1. Multi-level **feedback queues**
2. Processes vs **Threads**:
   - Parallel **execution traces** (i.e., must be scheduled) that **share resources**
   - Thread control blocks, containing less info than process control blocks
   - **Faster** to create/switch, alternate **I/O and CPU bound**, **parallelism**
   - They **abstract parallelism** from the programmer

Goals for Today

Overview

1. Different thread implementations
2. Multi-processor and multicore scheduling

OS Implementations

Thread Types

- **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
- They can be implemented on **OS** that **does not support multi-threading**

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)
  - The runtime system can **switch local blocking threads** in user space (program counter, store and reload registers)
- Disadvantages:
  - **Blocking system calls** suspend the entire process (User threads are onto a single process, managed by the kernel)
  - **Page faults** result in blocking the process
  - **No true parallelism** (a process is scheduled on a single CPU)
  - **Clock interrupts** are non-existent (i.e. user threads are non-preemptive)
  - Remember: threads are particularly **useful** when **code often blocks**!

User Threads
Many-to-One

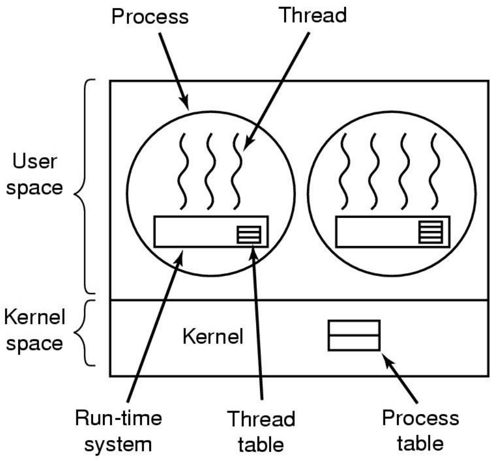


Figure: User threads (Tanenbaum 2014)

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)
  - **Thread blocks**, kernel chooses thread from same or different process ($\leftrightarrow$ user threads)
- Advantages:
  - **True parallelism** can be achieved
  - **No non-blocking** system calls needed
  - 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

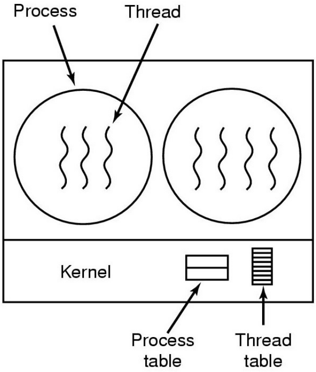


Figure: Kernel threads (Tanenbaum 2014)

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)

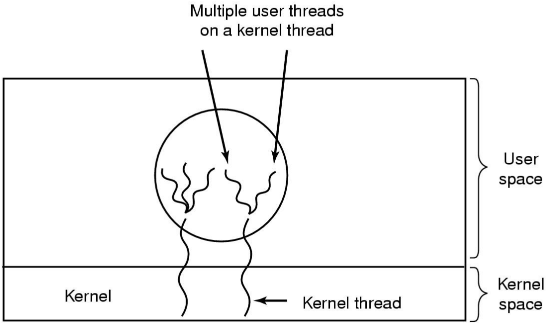


Figure: Kernel threads (Tanenbaum 2014)

Comparison
Thread Implementations

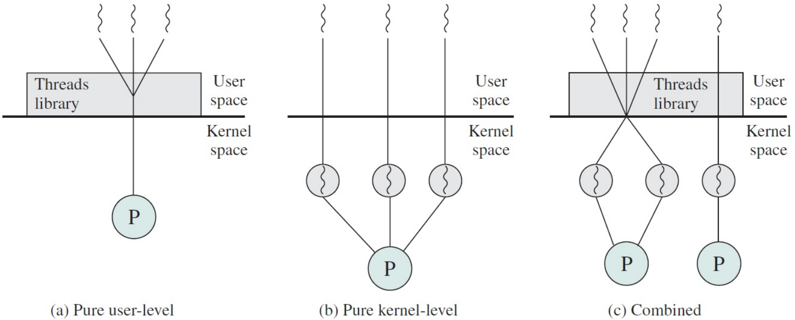


Figure: Comparison (Stallings)

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

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)

Scheduling on Multi-core/Processor Systems

Einstein's influence on multi-core processors :-)

- Electrical signals cannot propagate any faster than the **speed of light** (30cm/nsec - in copper 20cm/nsec)
- A 10GHz clock speed, i.e. 10×10^9 cycles per second, means the **maximum distance is 2cm**, and **2mm for 100GHz**
- The faster a computer runs, the more **heat** it dissipates
- This imposes a **fundamental limit on clock speed**, hence, **parallelism** is exploited to increase computational power

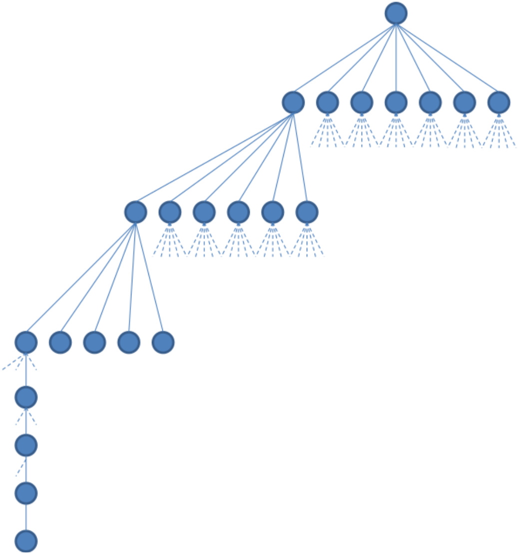
Multi-processor Scheduling

Scheduling Decisions

- **Single processor** machine: **which thread** to run next (one dimensional)
- Scheduling decisions on a **multi-processor/core** machine include:
  - Which process/thread to run **where**, i.e., which CPU?
  - Which process/thread to run **when**?
- Threads can be **related** or **unrelated** (multiple users)
  - **Related** threads: e.g. the **same process** creates **multiple threads** that **communicate** with one another and **ideally run** together (e.g. search algorithm)
  - **Unrelated** threads: e.g. threads that belong to **different processes**, possibly started by **different users** running **different programs**

Multi-processor Scheduling

Related Processes



Best Value Found

Scheduling Unrelated Threads
Shared Queues

- A single or multi-level queue **shared** between all CPUs
- Advantage: automatic **load balancing**
- Disadvantages:
  - **Contention** for the queues (locking is needed)
  - Mainly applicable for **unrelated threads/processes**
  - *"All CPUs are equal, but some are more equal than others"*: does not account for **processor affinity**:
    - **Cache** becomes invalid when moving to a different CPU
    - Translation look aside buffers (**TLBs**) become invalid
- Windows will allocate the **highest priority threads** to the individual CPUs/cores

Scheduling Unrelated Threads

Private Queues

- Each process/thread is assigned to a queue **private** to an individual CPU ($\Rightarrow$ two level scheduling)
- Advantages:
  - **CPU affinity** is automatically satisfied
  - **Contention** for shared queue is minimised
- Disadvantages: less **load balancing**
- **Push** and **pull migration** between CPUs is possible

Scheduling Related Threads

Working Together

- Threads belong to the same process and are **cooperating**, e.g. they **exchange messages** or **share information**
- The aim is to get threads **running**, as much as possible, at the **same time** across **multiple CPUs**
- Approaches include:
  - **Space** sharing
  - **Gang** scheduling

Scheduling Related Threads

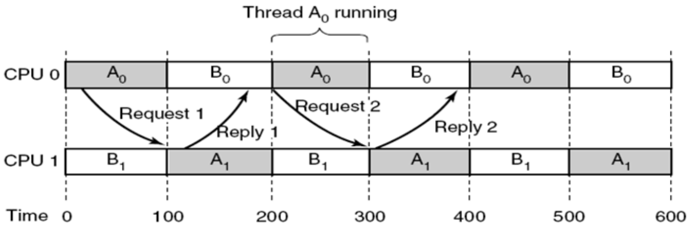
Space Scheduling

- Approach:
  - N threads are allocated to N **dedicated CPUs**
  - N threads are kept waiting until N CPUs are available
  - **Non-preemptive**, i.e. blocking calls result in **idle CPUs** (less context switching overhead but results in CPU idle time)
  - Scheduling decisions are made using **scheduling algorithms** (e.g., FCFS, PQ)
- The number N can be **dynamically adjusted** to match processor capacity

Scheduling Related Threads

Gang scheduling

- Imagine:
  - Process A has thread A1 and A2, A1 and A2 cooperate
  - Process B has thread B1 and B2, B1 and B2 cooperate
  - The scheduler selects A1 and B1 to run first, then A2 and B2, and A1 and A2, and B1 and B2 run on different CPUs
  - They try to send messages to the other threads, which are still in the ready state



Scheduling Related Threads

Gang scheduling

- Time slices are **synchronised** and the scheduler **groups threads** together to run simultaneously (as much as possible)
- A **preemptive** algorithm
- **Blocking threads** result in idle CPUs

CPU

Time slot	0	1	2	3	4	5
0	A_0	A_1	A_2	A_3	A_4	A_5
1	B_0	B_1	B_2	C_0	C_1	C_2
2	D_0	D_1	D_2	D_3	D_4	E_0
3	E_1	E_2	E_3	E_4	E_5	E_6
4	A_0	A_1	A_2	A_3	A_4	A_5
5	B_0	B_1	B_2	C_0	C_1	C_2
6	D_0	D_1	D_2	D_3	D_4	E_0
7	E_1	E_2	E_3	E_4	E_5	E_6

Summary
Take Home Message

- User, kernel, and hybrid **thread implementations**
- **Multi-processor/core** scheduling is “a bit different” (load balancing, processor affinity, etc.)
  - **Related** and **unrelated** threads
  - **Shared** or **private** queues
  - **Space** scheduling or **gang** scheduling