Threads

Chapters: 4.3, 4.4, 4.4.1, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.6.3, 4.6.4, 4.7, 4.7.2

# Concurrency and Parallelism

## Concurrency vs Parallelism

* Consider we have multiple tasks to finish:
* **Concurrent** execution involves partially completing multiple tasks, swapping between each until eventually they are all complete.
* As shown below, the single core processor works on task 1 for a bit, and then task 2, eventually coming back to task 1 etc.
* **Parallel** executioninvolves running multiple processes **at the same time** i.e. progress is made at the same time.
* This requires the physical resources to implement multiple tasks at once, unlike concurrency.

Diagram, table

Description automatically generated with medium confidenceA screenshot of a computer

Description automatically generated with medium confidence

## Concurrency **and** Parallelism

* You can then use both **concurrency and parallelism**
* A screenshot of a computer

  Description automatically generated with medium confidenceThis uses multiple cores implementing multiple processes.
* If they need to **communicate**, different methods have their drawbacks:
  + **message passing** with OS in the middle is **slow**
  + **Shared memory**: **cumbersome**. hard to set up with **limited shareability.**

# Concurrent/Parallel Communicating Processes

* Given **process abstraction** like fork():
  + Each of them can map to **same** **memory** so share data (shmget() API provides more info)
  + Can make them both open the **same OS resources**.
* This is however **cumbersome**, results in **limited shareability**  and is **inefficient** (time for creating structures and space for PCB, page tables etc.)

# From Processes to Threads

* Diagram

  Description automatically generatedAs a result of these complications, we move from processes to **threads**.
* Instead of using multiple processes, each with independent address spaces, we can use a process with threads.
* Each of these threads then has access to the same address space and all the same resources. No need for inter-process communication i.e. they “**share a process**”
* Threads have **different instruction flows**, with private stacks and CPU states.

# Threads

* Key idea: Separate foundational components of a process (address space, execution state, OS resources) into different **abstractions**/**entities**,
  + **Process**: address space + CPU resources
  + **Threads**: CPU state (execution state) including:
    - program counter
    - stack pointer.
    - registers
  + Most OSs support this format.
* Diagram

  Description automatically generatedSingle Threaded and Multithreaded Processes can be visualised as shown below:

## Use Case Scenarios

* When we have various **instruction flows**:
  + Run the same or different code
  + Access the same data (or part of it)
  + Have the same privileges
  + Use the same OS resources
* Each instruction flow i.e. each thread has a **hardware execution state**:
  + Execution stack and stack pointer: traces state of procedure calls made
  + PC (Program counter): next instruction to be executed
  + Set of general-purpose processer registers and their values.
* For each thread these must be saved to resume later

## Threads and Processes

* **Process**: defines address space and process’ OS resources
* **Thread**: defines sequential **execution flow** within process
* A thread is **bound** to single process (thus address space)
  + Processes can have **multiple threads**
  + Sharing data between them is cheap due to shared address space.
  + Instead of using fork() and exec() we can create them cheaply
* Threads become **unit of scheduling**
  + This depends on scheduling
  + Processes just become **containers** in which threads execute

## Communication

* Threads are diverse execution flows sharing an address space
* Address space provides **isolation** to related threads, where they are updating a **shared variable**
* Diagram

  Description automatically generatedConsidering threads results in a new address space diagram, as shown to the right.
* As shown each thread has their own SP and PC, and all share the same heap.
* Max stack size can be configured per thread.
* When a thread exceeds its stack space, it just crashes.

## Thread Control Blocks (TCB)

* Diagram

  Description automatically generatedEach process has a PCB. If we break this into two pieces:
  + Info on program execution stored in **Thread Control Block** (**TCB**)
    - Program counter
    - CPU registers
    - Scheduling information
    - Pending I/O information
  + Other info stored in PCB:
    - Memory management information
    - Accounting information

## Example Applications

* Useful for:
  + Handling concurrent events (e.g. web servers, clients etc.)
  + Building parallel programs (e.g. matrix multiply, ray tracing)
  + Improving program structure (divide and conquer)
  + Useful on **uniprocessor** despite only one thread running at a time

## Terminology

• There is the potential for some confusion

* + “process” == “address space + OS resources + **single** execution flow”
  + “process” == “address space + system resources + **multiple** execution flows”
  + We use **single-threaded** and **multiple threaded** processes

## Thread Management and Creation

### Kernel-Level Threading

* **OS Kernel** is responsible for creating/managing threads.
  + Kernel call to create a new thread would:

1. Allocate an execution stack within address space
2. Create and initialise TCB (SP, PC, register values)
3. Enqueue to ready queue

* There is scheduling involved, so somehow the OS needs to be involved in threads
* This is called **kernel-level** (**1-to-1**) **threading** 
  + There is a **thread name space** in the kernelwhere each thread as a TID (Thread ID)
  + These are integers similar to PIDs, and each thread has one.
* Therefore the OS is managing **threads** and **processes**
  + All thread operations implemented in kernel
  + OS schedules all threads in a system
    - If on thread in a process blocks, the OS knows about it and can run other threads **from that same process**.
    - This makes it possible to **overlap I/O** with computation **within a process**
* Threads are much cheaper to manage than processes as there is less state to allocate and initialise
* It can however be expensive for **fine-grained use** (tiny jobs):
  + Orders of magnitude more expensive than a procedure call
  + In this context, threads just become syscalls e.g. context switch, argument check etc.
  + Diagram

    Description automatically generatedKernel state has to be maintained for each thread

### User-Level Threading

* Instead of everything completed in kernel, we manage the threads at the user level **within the process**.
  + A **library** Is used to manage the threads
    - Thread manager does not need to manipulate address spaceses
    - Threads differ only in hardware contexts which can be manipulated by user-level code
    - Thread package multiplexes user-level threads in a process
* This is **user-level threading** (or **1:N** threading)
  + Kernel is unaware of threads existence
  + Diagram

    Description automatically generatedTCBs operate at user level
* All kernel sees are address spaces
* This is lightweight and fast
  + Managed entirely by user-level library
  + Each thread is represented simply by:
    - PC, registers, a stack
    - Small thread control block
  + Creating a thread, switching between threads, and synchronizing threads are done via **procedure calls**
    - No kernel involvement is necessary
* User-kernel threading operations cab be 10-100x faster than kernel threads

#### Implementation

1. **OS Schedules** Process
2. Process executes user code (
   * at user-level
   * includes thread support library and thread scheduler)
3. **Thread scheduler** determines when a user-level thread runs
   * Uses queues to keep track of what threads do (run, ready, wait)
     + Similar to OS but in user-space as library
4. **Context Switch** at user-level
   1. Saves context (state) of currently running thread by pushing CPU state onto thread stack
   2. Restores context of next thread by popping PCU state from next thread’s stack
   3. Returns as the new thread and execution resumes
   * Works at level of **procedure calling convention**
   * No changes to memory mapping required as a result.

* If a User-level thread starts to **hog the CPU** theretwo strategies:

1. Force everyone to cooperate
   * + Thread willingly gives up CPU by calling yield().
     + This calls the scheduler which context switches to another ready thread.
2. use **pre-emption**
   * + Scheduler requests a **timer interrupt** to be delivered by OS periodically
     + Usually delivered as UNIX signal (man signal)
     + Similar to software interrupt, but delivered to user-level via OS instead of to OS via hardware
     + At each timer interrupt, scheduler gains control and context switches appropriately

* If a thread attempts I/O, the process “powering” it is lost for the duration of the I/O operation
  + Process blocks in OS
  + OS is not aware of threads as OS only sees the process
  + No process’ threads make progress, but other processes can progress
* With kernel threading, the kernel knows about each process’ threads and can schedule other processes during I/O. This is an advantage over user-level threading

Diagram

Description automatically generated

#### N:M Threading Model

1. Here we combine both user-level and kernel-level threading
2. Kernel threads are linked to user-threads i.e. M threads created by user are linked to N kernel threads

Diagram

Description automatically generated