Threads

Chapters: 4.3 (all), 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)

# Benefits

* Benefits of multithreaded programming can be broken down into four categories:
  + **Responsiveness**: Even if part of program is blocked or performing a length operation, it can keep responding to the user.
  + **Resource sharing**: threads share memory and resources of a process, meaning they can share an address space without the OS having to facilitate inter-process communication.
  + **Economy**: more economical to create and context switch threads rather than processes. Only registers, stack and PC need changed, not code, data, and files.
  + **Scalability**: Threads can run in **parallel**, allowing for faster computation.#

# Challenges

* **Identifying tasks**: find areas that can be divided in to separate, concurrent tasks. Ideally these are independent such that they can run in parallel.
* **Balance**: Ensure split threads perform **equal work** of **equal value**.
* **Data Splitting**: Data accessed and manipulated by different tasks must be divided to run on separate cores
* **Data Dependency**: Execution of dependant tasks must be synchronised to accommodate data dependencies.

# 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: Idea

* Diagram

  Description automatically generatedKey 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.

# 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 vs 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

## Creating Threads

* Two strategies for creating multiple threads:
  + **Asynchronous Threading**: Once parent has created child thread, parent **resumes execution** and they execute **concurrently** and **independent** of one another. Typically used for designing responsive UIs.
  + **Synchronous Threading**: Parent waits for child(ren) to terminate before resuming. Threads of parent run concurrently. Uses a lot of data sharing.

### Pthreads

* Text, letter

  Description automatically generatedExample of synchronous threading
* POSIX standard for defining API for thread creation and specification.
* pthread\_t: type for representing thread IDs. Defaults are provided.
* pthread\_attr\_t: type for representing attributes for the thread. Defaults are provided.
* pthread.create():function for running a thread. Parameters include:
  + address of pthread id
  + address of pthread attributes
  + (void) pointer to function
  + (void) function to parameters for function
* pthread.exit():function for terminating thread. Takes exit status as parameter.
* pthread.join(): function for “waiting” for thread(s) to execute. Takes two parameters:
  + ID of thread
  + Pointer value for obtaining exit status of thread. Can be set to NULL if unnecessary.
* Example of thread summation function shown. Parent outputs value that thread summates.

## Thread Models

* Threads fall into two categories:
  + **user** threads: no OS involvement
  + **kernel** threads: OS supported and managed directly by kernel.
* Three different kinds of relationships between user and kernel threads:

### Many-to-one

* Many user-level threads mapped to one kernel thread
* Thread management handled by thread library in **user** space, so it is efficient.
* Entire process will block if thread blocks
* Multiple threads unable to run in parallel as only one user thread can run at a time.

### One-to-one Model

* Each user thread is mapped to a kernel thread
* Provides more **concurrency** as other threads can run one when blocks.
* Allows multiple threads to run in parallel on multi-processors.
* Only drawback: multiple user threads can result in multiple kernel threads, which could burden system performance.

### Many-to-many model

* Combines both approaches: multiplexes many user-level threads to smaller or equal number of kernel threads.
* Number of kernel threads either specific to particular application or particular machine.
* When a thread issues a blocking syscall, the kernel can schedule another thread.
* Most flexible out of three approaches but not trivial, and with more cores in modern processes, one-to-one mapping is easier to facilitate.

## 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 spaces
    - 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

## Thread Pools

* Two issues can arise with a multi-threading computer system:
  + **Overhead** of **creating** and **terminating** threads
  + Certain tasks may lead to an **infinite generation of threads** e.g. threads for dealing with server requests. These can exhaust system resources like CPU time or memory.
* **Thread pools** deal with this issue:
  + At start up create a certain number of threads and place them in pool to sit and wait for work.
  + Tasks are submitted to the thread pool and queued. When a thread becomes available, it completes the task, and upon completion, it returns to the thread poo and awaits another task.
* Benefits of thread pools are:
  + Servicing with existing threads much faster than waiting to create a thread
  + Thread pools limit the number of threads existing at any one point.
  + Separates execution from creation of task. Allows for different strategies of running the tasks e.g. using time delays or periodic execution.
* Number of threads in a pool can be set heuristically based on factors such as physical memory number of CPUs and cores and the expected number of concurrent tasks.
* Sophisticated systems can dynamically adjust pools based on demand.

## Fork-Join

* **Fork-Join model**: main parent threads forks one or more child threads, waits for their termination, and **joins** with them to retrieve and combine their results. It is **synchronous**
* Known for its use for explicit threading, but could be used for **implicit threading**.
  + Instead of constructing thread directly for task, a **library** determines the number of threads to create, and designated them to the parallel tasks.
* Main challenge: determining when a task is “small enough” to be executed directly. Requires careful timing trials of different thresholds.

# Threading Issues

## Fork() and Exec() on multithreaded processes

* What happens when a thread calls fork()?
  + Some systems use two versions of fork():
    - One duplicates all threads
    - The other duplicates only the thread that invoked fork()
* What happens when a thread calls exec()?
  + Program specified in exec() call replaces entire process: including all threads.

## Signal Handling

* For **multithreaded** systems, signals are handled in one of four ways:
  + Signal delivered to threat to which it applies
  + Signal delivered to all threads
  + Signal delivered to specified threads.
  + Thread assigned for receiving all signals.
* POSIX pthreads provide the following function for delivering signals to threads:
  + pthread\_kill(pthread\_t tid, int signal).
* Windows emulates support for signals by using **asynchronous procedure calls** (**APC’s**)
  + APC will always deliver to particular thread rather than process.

## Thread Cancellation

* **Thread cancellation**: terminating thread **before its completion**.
* Example: when searching a database. Once result found, cancel all other threads.
* Thread to be cancelled is called the **target thread**.
* Cancellation of a target thread may occur in two scenarios:
  + **Asynchronous Cancellation**: One thread immediately terminates target thread.
  + **Deferred Cancellation**: Target thread periodically checks if it should terminate.
* Difficulty occurs when resources have been allocated to a cancelled thread, or when it has been cancelled while updating shared data.
* Asynchronous cancellation can allow OS to reclaim **system resources** but not necessarily **all resources**

### Pthread Cancellation

* Pthreads use pthread\_cancel() to cancel threads, taking the thread ID as an argument.
* Handling of cancellation dependant on how target thread is set up to handle cancellation.
* Three cancellation modes set with a state and type: off, deferred, and asynchronous. These states can be disabled and enabled.
  + Thread can set and change its own cancellation state and type using an API
* Default cancellation type is deferred. Must reach a **cancellation point**.
  + Most **blocking syscalls** in POSIX and standard C are defined as cancellation points.
  + To check if cancellation request has been made, pthread\_testcancel() is used.
* Also uses a **cleanup handler** to release any resources a thread has acquired before cancellation.

## Thread Local Storage

* Threads may need their own copy of certain data. This is called **thread-local storage** (**TLS**).
* Similar to static data: available amongst all function calls made in thread.

## Scheduler Activations

* If communication between many-to-many or two-level thread models is facilitated, then the number of kernel threads can be dynamically adjusted to ensure best performance.
* **Lightweight Process** (**LWP**): an intermediate data structure between the user and kernel threads. Each LWP is attached to a kernel thread
* User-thread library sees LWP as **virtual processor**. Each is attached to a kernel thread.
* **Scheduler activation** is a communication scheme whereby the kernel provides an application with a set of LWPs, and the application can schedule user threads onto the available LWPs.
* Communication is established through using **upcalls** sent by the OS to **upcall handlers** which run on the LWPs issued by the kernel.

# Linux Threads

* Linux does not distinguish between processes and threads: it uses the term **task**.
* Each task is made using struct task\_struct
* This stores pointers to data structures where the task is stored
* To create a thread, clone() is used with 4 different flag options
  + CLONE\_FS: File-system information is shared
  + CLONE\_VM: Share virtual memory space
  + CLONE\_SIGHAND: Share signal handlers
  + CLONE\_FILES: share set of open files.
* fork() will create a new task with a copy of all associated data of the parent process
* clone() will create a new task with pointers to the data structures of the parent class, depending on the set of flags passed.