**Review**

* **Thread**
  + Fully describes program state (PC, registers, execution flags, stack)
* **Address Space** 
  + Set of memory addresses accessible to a program (for read or write)
  + May be distinct from memory space of the physical machine (in which case programs operate on virtual space)
  + Translation enables protected address spaces
* **Process**
  + Protected address space + One or more threads
* **Dual Mode Operation/Protection**
  + Kernel State, User State
  + Only certain operations are available in kernel state
  + ^ combined with translation allows programs to be isolated from each other
  + Dual mode enables multiplexing of hardware for processes

Illusion of multiple processors:

* Threads are virtual cores
* Multiple threads: multiplex hardware in time (load and store registers in memory)
* Thread executes when it is resident on process registers
* Each virtual core (thread) has
  + PC, registers, execution flags, stack pointer
* Thread is either on real (physical) core or in memory (Thread Control Block)

Address Space:

* Set of accessible addresses + state associated with them
* Virtual address space -> Processors view of memory (independent of physical memory, requires translation)
* Translation through page table (more soon!)
* Blue and thread can’t interact with each other’s memory (blue table won’t map to green physical memory)

A picture containing text, screenshot, diagram, line

Description automatically generated

Process: Protected address space w/ one or more threads

* Execution environment with restricted rights
  + One or more threads executing in protected address space
  + Owns memory (address space), file descriptors, network connections
* Instance of a running program
* Why processes?
  + Protected from each other
  + OS protected from them
* Anything that runs outside of kernel

Dual Mode Operation:

* Processes execute in user mode
  + To perform privileged actions, processes requests services from the OS kernel
  + Carefully controlled transition from user to kernel mode (system calls, interrupts, exceptions)
* Kernel executes in kernel mode
  + Performs privileged actions to support running processes
  + … and configures hardware to properly protect them
* Carefully controlled transitions between user mode and kernel mode
  + System calls, interrupts, exceptions

Threads: single execution context

* Provides abstraction: single execution sequence that represents a separately schedulable task
* Threads are a mechanism for concurrency (overlap execution)
* Protection is an orthogonal concept (protection domain can contain many threads)

**Motivation for Threads:**

* Operating systems must handle multiple things at once (MTAO)
  + Processes, interrupts, background system maintenance
* Network servers must handle MTAO
* Parallel programs must handle MTAO
* Programs with GUI
  + To achieve user responsiveness while doing computation
* Network and disk bound programs must handle MTAO
  + To hide network/disk latency
  + Sequence steps in access or communication
* Threads are a unit of concurrency provided by OS
  + Each thread can represent one thing or task

Some definitions:

* Multiprocessing -> Multiple CPU
* Multiprogramming -> Multiple jobs/processes (not necessary simultaneously)
* Multithreading -> Multiple threads/processes

What does it mean to run two threads concurrently?

* Scheduler is free to run threads in any order and any interleaving
* Threads may run to completion or time-slice in big chunks or small chunks

The second we have multiple threads, correctness needs to be ensured with scheduler

A picture containing diagram, line, plot, font

Description automatically generated

Concurrency != Parallelism (though both pictures above are concurrency)

Parallelism requires multiple things happening in any given two slice

Example: Two threads on a single core system

* Execute concurrently
* But not in parallel
* Each thread manages a separate thing or task but not necessarily the same time

Ex:

main(){

ComputePI(“pi.txt”);

PrintClassList(“classlist.txt”)

}

What’s the behavior here?

* ComputePI runs forever 🡪 PrintClassList never runs
* Using threads can allow both to run

main(){

create\_thread(ComputeP, “pi.txt”);

create\_thread(PrintClassList, “classlist.txt”);

}

Now scheduler interleaves these two operations 🡪 Should behave as if there are two CPU’s

Threads mask I/O latency (handle I/O in another thread to avoid blocking other progress)

A thread is in one of three following states:

* RUNNING (running)
* READY (eligible to run, but not currently running 🡪 OS puts in ready queue)
* BLOCKED (ineligible to run)
* If a thread is waiting for an I/O to finish, the OS marks it BLOCKED
* Once the I/O is finlly done, the OS marks it READY

Can’t go directly from BLOCKED to RUNNING (only BLOCKED -> READY queue)

A picture containing text, screenshot, font, line

Description automatically generated

main(){

create\_thread(ReadLargeFile, “pi.txt”);

create\_thread(RenderUserInterface);

}

What is the behavior here?

* Still respond to user input WHILE reading file in the background

Normally when you compile C program and run executable, this new process only has one thread in its own address space

Q: How can we make a multithreaded process?

A: Once the process starts, it issues system calls to create new threads

* These new threads are a part of the process, they share its address space

**System Calls**:

* Difference between use and system is at System Call Interface (Kernel Mode vs User Mode)
* Getting to Kernel from User Mode through System Call Interface is with system calls
* A diagram of a system call interface

  Description automatically generated with medium confidenceSystem calls are **NOT** standardized among OS
* OS Library issues Syscalls (libc linked with your source code)

A picture containing text, screenshot, font, number

Description automatically generated

* OS Library API for threads: pthreads
  + int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr, void \*(\*start\_routine)(void\*), void\* arg)
    - thread is executing *start\_routine* with *arg* as its sole argument
    - return is implicit call to pthread\_exit
  + void pthread\_exit(void\* value\_ptr);
    - terminates the thread and makes value\_ptr available to successful join
  + int pthreads\_join(pthread\_t thread, void \*\*value\_ptr)
    - suspends execution of the calling thread until the target thread terminates
    - On return with a non\_NULL value\_ptr, the value passed to pthread\_exit() by the terminating thread is made available in the location referenced by value\_ptr
  + What happens when pthread\_create() is called in a process?
    - It is a function 🡪
      * does some work
      * assembly code
      * trap instruction to jump into kernel mode(almost as an error)
      * Kernel:
        + get args from regs
        + dispatch to system func
        + DO the work to spawn new thread
        + Store return value in %eax
      * Back into user mode (switching mode is expensive)
    - Wrapper around a system call
  + Creating thread is creating a schedulable entity
* For-Join Pattern
  + Main thread creates (forks) collection of subthreads passing them args to work on …
  + … and then joins with them, collecting results

State shared by threads in process/address space

* Content of memory (global variables, heap)
* I/O state (file descriptors, network connections, etc)

A picture containing text, screenshot, font, number

Description automatically generated

State “private to each thread”:

* Kept in TCB = Thread Control Block
* CPU registers (including PC)
* Execution stack – what is this?

Execution stack:

* Parameters, temporary variables
* Return PCs are kept while called procedures are executing

Stacks holds temporary results

Permits recursive execution

A picture containing text, screenshot, font, number

Description automatically generatedMemory layout with two threads:

* Two set of CPU registers
* Two sets of stacks
* Process is protection, not threads 🡪 Threads can interfere with each other stack space
* Issues:
  + How do we position stacks relative to each other?
  + What maximum size should we choose for the stacks?
  + What happens if threads violate this?
  + How might you catch violations?

A picture containing text, font, screenshot, white

Description automatically generated

A picture containing text, screenshot, font, diagram

Description automatically generatedThreads introduce non-determinism:

* Scheduler can run threads in **any order**
* Scheduler can switch threads **at any time**
* This can make testing difficult

Independent threads

* No state shared with other threads
* Deterministic, reproducible conditions

Cooperating threads

* Shared state between multiple threads
* Race conditions

Synchronization: Coordination among threads, usually regarding shared data

Mutual exclusion: Ensuring only one thread does a particular thing at a time (one thread excludes others)

* Type of synchronization

Critical Section: Code exactly one thread can execute at once

* Result of mutual exclusion

Lock: An object only one thread can hold at a time

* Provides mutual exclusion

Locks provide two **atomic** operations:

* Lock.acquire() 🡪 Wait until lock is free; them mark it as busy
  + After this returns, was the calling thread *holds* the lock
* Lock.release() 🡪 Mark lock as free
  + Should only be called by a thread that currently holds the lock
  + A diagram of a set of data structure

    Description automatically generated with low confidenceAfter this returns, the calling thread no longer *holds* the lock

Mutex:

* int pthread\_mutex\_init(pthread\_mutex\_t \*mutex, const pthread\_mutexattr\_t \*attr)
* int pthread\_mutex\_lock(pthread\_mutex \*mutex)
* int pthread\_mutex\_unlock(pthread\_mutex \* mutex)

Ex:

int common = 162;

pthread\_mutex\_t common\_lock = PTHREAD\_MUTEX\_INITIALIZER;

pthread\_mutex\_lock(&common\_lock):

int my\_common = common++;

pthread\_mutex\_unlock(&common\_lock);

If process are created by other processes, how does the first processes start?