Processes

CIS-4020
Vermont Technical College
Peter C. Chapin
What is a Process?

- **Defn**: A process is a program in execution.

  - Execution state
    - Current program counter (instruction pointer)
    - Register values
    - Stack
    - Global and heap data (address space)

  - Kernel state
    - Open files, network connections, and other handles.
    - Open IPC channels (shared memory, message queues...)
    - Signal handler definitions
    - Credentials (user ID, capabilities, etc)
    - Working directory
What is a Thread?

- Sometimes called a "lightweight process."
  - A path of execution inside a process.
  - Unique execution state
    - Except shares the address space with other threads in the same process.
      - Memory allocated by one thread visible to others (in the same process).
  - Shares kernel state (with other threads in process)
    - Files, network connections, etc.
    - IPC channels
    - Credentials
    - Working directory
Kernel Sees Threads

- Kernel must manage the threads separately
  - Must store execution state when suspended.
  - Must schedule many thread across available CPUs
- Linux treats threads as the *unit of scheduling*
  - ... doesn't really worry about processes.
    - Fairly common.
    - Not universal: Some systems schedule processes first and then the threads inside that process second.
      - "Multi-level scheduling."
      - I believe modern Linux allows for this possibility ("scheduling domains" and "control groups").
fork

- The fork system call copies parent to a child.
  - Entire address space is copied!
  - Child (almost) identical
    - Different PID, different PPID, and a few other things.
- Inefficient?
  - No because of copy on write (COW)
    - Memory shared between parent and child.
    - Marked read-only
    - Copied a page (4 KiB) at a time as needed.
**execve**

- The `execve` system call loads a new program
  - The currently executing program is replaced
    - Address space re-initialized
    - Some kernel state is retained...
      - Open files and network connections, for example.
      - Allows the new program to use files opened by parent.
  - Common idiom: First `fork` then `execve` in child.
    - Copying address space of parent just to reinitialize it?
      - Not really that bad because of COW.
      - Older systems also had `vfork` system call.
Linux's clone System Call

- Linux has a clone system call.
  - Creates a new process but allows sharing.
    - Shared address space
    - Shared open files
    - Shared signal handlers
  - Flags control what is shared so you can mix.
  - `fork` is like `clone` with nothing shared.
  - A new thread is like `clone` with everything shared.
    - In fact, threads are implemented on Linux using `clone`. 
fork/clone, Whatever!

- Internally Linux doesn't care (much)
  - Each "process" gets an entry in the task table.
  - Kernel tracks shared subsystems appropriately.
  - Must deal with PID values carefully.
    - POSIX requires all threads of a process to have the same PID.
    - Linux assigns a new "internal" PID to every thread (since they all just look like separate processes).
    - BUT... can translate internal PID values to user oriented values when necessary.
Namespaces

- It gets more complicated...
  - Modern Linux allows you to create namespaces.
    - Different "views" of the system.
    - A process can appear in multiple namespaces (or not)
    - A process can have different PIDs when seen from different namespaces.
    - PID namespaces are hierarchical (parent/child)
    - Kernel must keep this all straight.
struct task_struct

- Linux "task" information
  - Each process/thread/task is represented by a struct task_struct object.
  - [Show what this looks like using cscope]
Task States

• A task can be in several states...
  • **Running**: The task is executing now.
  • **Waiting**: The task could execute but must wait for access to the CPU
  • **Sleeping** (or **Blocked** or **Suspended**): The task is waiting for an event (I/O to complete or for some other task to do something).
    - Task can't execute and is not scheduled. Consumes no CPU time.
    - Most tasks sleep most of the time.
  • **Terminated**: The task has ended.
Context Switching

• Kernel suspends one task and resumes another
  • Outgoing task has execution state saved.
    – CPU registers (stack pointer, status register, etc).
  • Kernel decides who gets to run next.
    – Scheduling decision.
  • Incoming task has execution state restored.
    – Picks up where it left off
• Kernel can preempt a task at any time.
  • For example via a timer interrupt.
    – Hundred(s) of times per second!
Multi-Tasking

- Rapid preemption gives illusion of multitasking
  - ... even with one CPU.
- However, most tasks sleep most of the time.
  - Thus scheduling typically only involves a few tasks.
    - For example, out of 100 maybe only 3 are runnable.
- Multiple processors can run tasks in parallel.
  - But context switching still necessary in general.
    - For example, out of 100 tasks, 3 are runnable, but there are only 2 processors.
Multi-Tasking Uses CPU Better

- CPU is used more effectively. Less waiting.
  - Task thinks (CPU "burst")
  - Task starts I/O operation. Must wait for slow device. (I/O "burst")
  - Kernel puts task to sleep. Schedules other tasks.
  - Eventually I/O completes. Hardware interrupts.
  - Kernel marks waiting task as runnable.
  - Scheduler executes task when it feels like it.
    - Often tasks coming out of sleep are given priority.
Time Slices

• If a task does not wait for I/O what happens?
  • Tasks given a time slice (or quantum).
    – If they use it all, they are interrupted (preempted).
    – Context switch forced. Outgoing task still runnable.
    – Kernel schedules some other task.
    – Original task will eventually be rescheduled.

• How long is the time slice?
  – Varies... Linux adjusts its size dynamically.
    • Current Linux scheduler doesn't actually use time slices per se.
    – Often on the order of 10 ms. Could be longer (100 ms?)
      • Short time slice increases overhead. Long time slice is choppy.
Preemptable Kernel?

- Old Linux kernel **not** preemptable!
  - Once a thread entered the kernel it would either
    - Execute until it returned to the application OR
    - Execute until it went to sleep.
    - If a timer interrupt occurred, the task would be resumed at once.

- Simplifies programming.
  - Easier to maintain consistent data structures without preemption.

- Bad?
  - Execution in kernel usually short. No problem!
Along Came SMP

• SMP = Symmetric Multiprocessor
  • A machine with more than one CPU.
    - Can execute more than one thread in parallel
    - Both threads can enter the kernel at once even without interrupts or preemption.
  • Quick Fix: The "Big Kernel Lock" (BKL)
    - Tasks locked the entire kernel on entry.
    - Only one processor at a time let inside kernel.
    - Other processor had to wait "at the door."
  • Bad?
    - Kernel execution time short. Parallel execution still possible in applications. No problem!
BKL Problems

● Inefficient
  ● It would be nice to run parts of the kernel in parallel
    – File system handling + network stack handling.
    – Virtual memory updates + device driver code.
    – etc

● Not scalable
  ● It gets really ridiculous with 4 or 8 processors!
Fine Grained Locking

- As Linux matured, the BKL was deprecated.
  - Fine grained locks used to protect subsystems
    - A thread can lock part of the kernel
    - Two threads locking different parts and run concurrently.
- As a side effect kernel preemption now allowed