# Operating System - Process and Threads

## Process and Threads

- Process and Threads
  - Interprocess communication — Process
    - Process communicates via
      - Signals
      - Pipes
      - Sockets
      - Files
  - From Processes to Threads
    - Process is a running program
      - Process has following parts
        - Address Space
          - Code
          - Heap
        - Execution State (active state)
          - Program counter
          - Stack and stack pointer
        - Os Resources
          - Open file table
    - Big Idea
      - Separate the address space from the execution state
      - Then , multiple "threads of execution" can execute in a single address space
    - Sharing
      - Threads can solve problem concurrently and can easily dive cod, heap, and global variables
    - Why?
      - Lighter Weight
        - Faster to create and destroy
        - Potentially faster context switch times
      - Concurrent programming performance gains
        - Overlapping computation and I/O
  - Threads
    - Thread
      - Is a single control flow through a program
        - Question: How is control flow represented?
        - Question: What is a control flow?
        - Control flow is the order function calls, instructions, and statements are executed or evaluated when a program is running
      - Memory is shared
        - Is much more efficient than sending messages and waiting
  - Multithread
    - Is a program with multiple control flow
    - Modern OS have taken the execution aspect of a process and separated it out into thread abstraction
    - Kernel-level Threads
      - OS now manages threads and processes
      - Is called kernel-level threads or lightweight processes
        - Linux: Tasks
        - NT: Threads
        - Solaris: Light weight processes
      - Is managed by OS
    - Kernel Thread Limitations
      - Pros
        - Makes currency much cheaper than process
        - Requires less state to allocate and initialize
      - Cons
        - suffers from too much overhead
        - We want thread operations to be fast as procedure calls
        - Thread is too fine-grained
    - User-level threads
      - Is managed by runtime system (user-level library)
      - Is represented by
        - PC
        - Registers
        - Stack
        - Small Thread Control Block
      - Creating a thread, synchronizing threads, and (switching) between threads are done by procedure calls
      - Has no kernel involvement
      - Are invisible to the OS
      - It is not well integrated with the OS
    - User-level thread limitations
      - Pros
        - Lightweight and fast
        - It runs 100x faster than kernel threads
      - Cons
        - the OS can make poor decision
          - Scheduling process with idle threads
          - De-scheduling a process with a thread holding a lock
          - Blocking a process whose thread initiated an I/ O even though it has other threads that can execute
        - Solution: communication between user thread manager and kernel thread manager
- System Calls
  - Interrupt
    - Is a software interrupt that allows user program to ask for privileged service in the kernel of operating system
    - For hardware, signals CPU that a hardware device has an event that needs attention
      - Examples
        - Disk I/O is complete
        - Pressing a keyboard
        - Moving a mouse
        - Segmentation fault
        - Divide by 0
    - For software, signals errors or requests for OS intervention
      - Examples
        - System call
        - Exception
        - Trap
        - Interrupt
    - Once called, CPU jumps to a pre-defined routine
      - Is called the interrupt handler
      - Key: OS is an event driven program
    - Is a signal caused by hardware or software
    - Responds to requests coming through interrupt handlers
  - System Call
    - System call is an API
      - API is the same API used in web programming
      - Is between user application and kernel
    - Is caused by specific instructions
      - Asks for permission to use a resource managed by OS
      - User interacts with hardwares using set of system calls
    - Raises two questions
      - 1. How do you actually invoke the operating system
      - 2. How do you keep applications from just using a resource without asking permission?
  - Boundary Crossings
    - Going from user mode to kernel mode
      - Explicit system call
      - Hardware interrupt
        - Is done by
          - Software trap or exception
          - Saves registers before switching to kernel mode
          - Switches previlege from user mode to kernel mode
    - While in kernel mode
      - Runs instructions that can only be done by OS ( Privileged instructions)
      - Is executed when OS finished its task
    - Going from kernel mode back to user mode
      - Switches previlege from kernel mode to user mode
      - OS sets up registers, MMU, mode for application
      - Jumps to next application instruction
  - Enforcing Restrictions
    - Hardware runs in user mode or system mode
    - Can only be run in system mode
      - Some instructions are previlaged instructions
      - On a "system call interrupt", the mode bit is switched to allow previlaged instructions to occur
  - Limited Direct Execution
    - Is babyproofing CPU
    - Definition
      - Limited
        - Means user process cannot execute previlaged instructions
      - Direct Execution
        - A process executes its instructions on the CPU without going through controller
  - Previlaged Instructions
    - Are instructions allowed to be run only in kernel mode
    - Are enforced by the CPU hardware itself
      - CPU has 2 protection levels: kernel and user mode
      - CPU checks current protection level on each instruction
        - Question: What happens if user program tries to execute a previlaged instruction?
          - It will generate an interrupt, and a program will fail
    - instruction types
      - Access I/O devices
        - Perform Direct Memory Access
        - Poll for IO
      - Manipulating memory management
        - Setup page tables
        - Load/flush the TLB and CPU caches
      - Configuring various mode bits
        - Interrupt priority level
        - Software trap vectors
      - Call hali instruction
        - Put CPU into low-power or idle state until next interrupt
  - System Call Interface
    - is the basic interface between a program and the kernel
    - Similar to API between user and server for the web
    - How it works
      - 1. User program calls C library function with arguments
      - 2. C library function passes arguments to OS
      - 3. Execute special instruction to trap to system mode
      - 4. Syscall handler figures out which system call is needed and calls a routine for that operation
        - Question: How does this differ from a normal C language function call? Why is it done this way?
    - Example: Linux Write System Call
      - printf
        - How it Works
          - 1. int 0x80 generates software interrupt
          - 2. int 0x80 points to system_call function stored in interrupt descriptor table
            - 0x80 is the index to interrupt table
          - 3. Uses register %eax allows system_call to index into appropriate sys call handler in sys_call_table
            - %eax is from process register
            - %eax is set while in user mode
          - 4. Run sys_call
  - System Call Operation
    - Kernel must verify arguments that it is passed
      - Question: Why?
    - A fixed number of arguments can be passed in registers
      - Because it can allow process handlers that it is not allowed to get (security issue), and may damage the system
      - Offers is the address of a user buffer containing data (for writes)
        - 0x1234 containing "Hello world"
      - Kernel must copy data from user space into its own buffers
    - Result of system call is returned in register %eax
  - System Calls in Linux
    - Can invoke any system calls from userspace using the syscall() function
    - Tracing system calls
      - Powerful mechanism to trace system call for an application
      - Use 'strace' command
      - The 'strace(s)' system call is used to implement ' strace' (also used by gdb)
    - Tracing library calls
      - Is traced using 'ltrace' command
      - Question: Why do need system call table?
      - Required to go to the right routine
  - System Call Dispatch
    - System call number is passed into register %eax
    - A system call is identified by a unique number
      - Offers an index into an array of function pointers: the system Call table
    - System call table has approximately 300 entries
    - Steps
      - 1. Kernel assigns each system call type a ' system call number'
      - 2. Kernel initializes 'system call table', mapping system call number to the function implementing that system call
      - 3. User process sets up system call number and arguments
      - 4. User process runs int N (on Linux N=0x80)
        - int 0x80 is called 'int interrupt'
        - On software interrupt int interrupt navigates to system_call function stored in interrupt descriptor table
      - 5. Hardware switches to kernel mode and invokes kernel's interrupt handler for X ( interrupt@jiaxin)
      - 6. Kernel looks up syscall table using system call number
      - 7. Kernel invokes the corresponding function
      - 8. Kernel returns by running iret (interrupt return)
  - Passing System Call Parameter
    - First parameter is always the syscall number
    - Can pass upto 6 parameters
      - eax
      - ecx
      - edx
      - esi
      - edi
      - ebp
    - To make sure user pointer points to user address space
      - Question: Must validate user pointers Why?
      - Otherwise, user pointer would point to kernel address space and this is a huge security issue
- Processes and System Calls
  - Process
    - Is a running program
    - Contains all of the following states for a program in execution
      - Address Space
      - Program Counter
      - General purpose registers.
      - Process ID (PID)
      - Execution stack
      - Execution State
    - Operating System Resources
      - Open files
      - network work connections
      - signals
    - Address + data
  - Process Creation
    - Is done by another process except the first
      - First process is created using init(PID 1)
      - Parent is the creator
        - the parent defines resources and previlages for its children (in some systems)
      - Child is created
      - The claim is true in UNIX
    - In UNIX, is done using
      - fork()
        - Creates and initializes new PCB
        - Creates a new address space
        - Creates a copy of parent process
          - Initializes the address space with a copy of the entire contents of the address space of the parent***
          - Initializes the kernel resources to point to the resources used by parent
            - e.g. open files
          - Places PCB on the ready queue
        - fork() Returns in two processes
          - the child's PID to the parent,
          - 0 to the child
      - exec()
        - 1. Stops the current process
        - 2. Loads program into the process' address space
        - 3. Initializes hardware context and args for the new program
        - 4. Places the PCB onto the ready queue
        - Overwrites the existing process
        - Return value of exec() is -1
          - exec() only returns when there is an error
    - Steps
      - 1. Create new process control block
      - 2. Create user address space
        - 1. Create New Process
        - 2. Load Executable
        - 3. Allocate memory
      - 1. Initialize start state for process
      - 2. Change state to "ready"
        - 3. Dispatch Process
      - Change state to running
  - Process Control Block
    - Is data structure of process
  - Process Life Cycle
    - Ready
    - Running
      - Has three states
    - Blocked
    - Has One queue for each state
    - As process changes state, PCB is unlinked from one queue and linked into another
  - Context switch
    - Ready to Running
    - Switches the CPU to another process
    - How does it work?
      - Saves state of the old process
      - Loads saved state for new process
    - When can this happen?
      - 1. Process calls yield() system call (voluntarily)
      - 2. Process makes a system call and is blocked
      - 3. Timer interrupt handler is run and decides to switch process
    - Make sure to memorize context switch diagram (Lecture 8)
  - Process Destruction
    - In UNIX, is done using
      - exit()
        - After creation, the parent may either
          - 1. wait for child to finish
          - 2. Run in paralial
        - Doesn't free all resources immediately
          - PCB of child lingers until parent is finished
          - Child waits for the return value of parent
          - 1. Must stop running the process to free anything
          - 2. Requires context switch to another process
        - Cause a process to voluntarily release almost all resources
  - Zombie Process
    - Is a process that is placed in a final state that has exited but has not yet been cleaned up
    - Retains its process PID
    - Is in this state until its parent cleans up
  - Function Call
    - Is a request made by a program or script that performs a predetermined function
    - e.g Invoking printfunc() function in main()
  - Process Execution
    - CPU reads one instruction at a time
    - Steps
      - 1. Fetch
      - 2. Decode
      - 3. Execute
      - 1. Read instruction into instruction register
      - 1. Read any needed data into register(s)
      - 1. Pass values from registers to ALU
      - 2. Write result back to register
- Brief Intro to Scheduling
- Synchronization

XMind | Trial Mode