Threads and IPC

Expensive Processes

- Creating processes takes resources
- To dispatch: freeing space
- Different processes cannot share address spaces or share resources
- Not all programs require strong separation
    - Multiple activities working together for a single goal
    - Mutually trusting elements of a system

Threads

- Strictly a unit of execution and scheduling
    - Each thread has its own stack, PC, and registers
    - Resources can be shared among threads
    - Threads are an interpreter

- Multiple threads can run a single process
    - Share the same code and data space (switching between threads doesn't take a context switch)
    - All have access to the same resources
    - Makes them cheaper to create and run

- Sharing the CPU between multiple threads
    - User level threads (using voluntary yielding)

Using Processes

- Running distinct programs
- Creating and destruction are rare events
    - (not as expensive if they are rare)
- When running agents with distinct privileges
    - (we don’t want sharing)
- When there are limited interactions and shared resources
    - Each process has a distinct resource pool
- To prevent interference between executing interpreters
- Needs to isolate failures from each other

  Multiple processes
    - Run slowly
    - Difficult to share resources

Using Threads

- Parallel activities in a single program
- When there is frequent creation and deletion
- Can be run with the same privileges
- When resource sharing is needed

Multiple threads

- Have to create and manage several threads
- Serialize resource use
- Added complexity
- Threads require protection from each other and must be managed by the programmer

Thread State and Thread Stacks

- Each thread has its own register, PS, PC
- Each thread has its own stack area
- Max stack size specified when thread is created
    - A process can contain many threads
    - Thread creator must know max required stack size (cannot grow stack as more threads added)
    - Stack space must be reclaimed when thread exits
- Procedure linkage conventions remain unchanged to processes
    - (threads can access libraries like processes)

User Level Threads vs Kernel Threads

- Kernel threads:
    - Abstraction provided by the kernel
    - Still share one address space but scheduled by the kernel
    - Multiple threads can use multiple cores

- User level threads:
    - Kernel knows nothing about them
    - Provided and managed via user level library (code in the library schedules the threads)
    - Scheduled by library, not kernel

Communications between processes

- Even fairly distinct processes may need to communicate
- OS provides mechanisms to facilitate communications
- Ex. Piping

IPC Goals

- Simplicity
- Convenience
- Generality
    - Should apply to any amount of data or signal
- Reliability
    - When data is transferred, should be certain that it works

IPC Support for OS

- Provided through sys calls
- Required activity from both communicating processes
- Usually mediated at each step by the OS
    - To protect both processes
    - Ensures correct behavior

OS IPC Mechanisms

- For local processes
- Data is in memory space of sender
- Data needs to get to memory space of receiver
- 2 choices:
    - OS copies the data
    - The OS uses VM techniques to switch ownership of memory to the receiver

Copying the data

- Conceptually simple
- Less likely to lead to user confusion
- Potentially high overhead

Using VM

- Much cheaper than copying bits
- Requires changing page tables
- Usually only one of the two processes sees the data at a time

IPC Forms

- Synchronous
    - Writes block until data is sent/delivered/received
    - Reads block until new data is available
    - Easy for programmers

- Asynchronous
    - Writes return when system accepts
    - No confirmation of transmission/delivery/reception (no error checking)
    - Reads return promptly if no data is available

Typical IPC Operations

- Create or destroy an IPC channel
- Write/send/out
    - Insert data into the channel

- Read/receive/get
    - Extract data from the channel

- Channel content query
    - Amount of data in teh channel

- Connection establishment

IPC Messages vs Streams

- Stream
    - Continuous stream of bytes
    - Read or write a few or many bytes at a time
    - Write and read buffer sizes are unrelated
    - Streams may contain app specific record delimiters

- Messages
    - Sequence of distinct messages
    - Each message has its own length
    - Each message is typically read/written as a unit
    - Delivery of a message is typically everything or nothing at all

Flow Control

- Flow Control: ensure a fast sender doesn't overwhelm a slow receiver
- Queued IPC consumes system resources
    - Buffered in the OS until the receiver asks for it
- Fast sender or non responsive receiver can increase the amount of buffer space needed
- Must be way to limit required buffer space
    - Sender side: block or refuse connection
    - Receiving side: either tell the sender to stop/slow down or drop packets
    - Handled by network protocols or OS mechanism
- Mechanism for receiver to send feedback to sender

  Reliability
    - Not a huge issue in a single machine, OS doesn't just lose IPC data
    - Across a network, requests and responses can be lost
    - On a single machine, a sent message might not ne processes
        - Infinite loop, buggy process, dead or invalid receiver

Pipelines

- Data flows through series of programs, like a pipe
    - Macro processor => compiler => assembler

- Data is a simple byte stream
    - Buffered in the OS
    - No need for temporary intermediate files

Sockets

- Connection between addresses and ports
    - Can connect and listen
    - One way communication
    - Sockets are complex to implement

- Many data options (flexible)
    - Reliability or best effort datagrams
    - Streams, messages

- Complex flow control and error handling
    - Retransmission and timeouts
    - Reconnection

Shared Memory

- OS arranges for processes to share read/write memory segments
    - Mapped into multiple process's address spaces
    - Application must provide their own control of sharing
    - OS not involved in the data transfer after it sets up the connection
    - Performance is very fast

- Varying complexity
    - Simple to understand
    - No synchronization, so the processes have to coordinate

- Only works on a local machine

Synchronization

- Making things happen in the right order
- Required for parallelism

Benefits of parallelism

- Improved throughput
    - Blocking on activity doesn't stop the others

- Modularity
    - Complex operations can be split

- Improved robustness
    - One thread failing doesn't stop the others

Problems with parallelism

- Sequential programs are easy to manage

- Independent parallel programs are easy if the parallel streams don't interact with each other
- Cooperating parallel programs are difficult
    - If the two execution streams aren't synchronized, results depend on the order of instructions
    - Order of execution is non deterministic

Race Conditions

- Different results occur when running threads in parallel since execution order changes every run
- Mutual exclusion: Conflicting updates
- Sleep problem: check/act races (waiting CPU should immediately start working when work is available)
- All or none transactions: multi-object updates (conflicting updates can occur)
- Distributed decisions: when there are disagreements to threads finishing

Non deterministic execution

- Parallel execution makes process behavior less predictable
    - Process block for I/O or resources
    - Time slice end preemption
    - Interrupt service routines
    - Unsynced execution on another core
    - Queuing delays
    - Time required to perform I/O operations
    - Message transmission time

Synchronization

- True parallelism is too difficult
- Pseudo parallelism is good enough
- We only need to synchronize key points of interaction

    Critical section serialization

    - Critical section: a resource shared by multiple interpreters
        - Sections which are shared by multiple concurrent threads, processes, or CPUs
        - Or by Interrupted code and interrupt handler

    - Use of resources changes its state (ex. Write to contents or properties)
    - Correctness depends on execution order
        - May not be a correct order, but it needs to be controlled

Editing and Reading files

- Updating a file by 2 processes
- One process wants to write to the file, the other wants to read

```
remove("inventory");
fd = create("inventory");
                        fd = open("inventory",READ);
                        count = read(fd,buffer,length);
write(fd,newdata,length);
close(fd);
```

- If Process 2 reads the inventory after its been created, then it reads an empty file
- We wanted to read it after it wrote the data

Re-entrant Signals

```
load r1,numsigs // = 0          load r1,numsigs // = 0
add r1,=1  // = 1               add r1,=1  // = 1
store r1,numsigs // =1          store r1,numsigs // =1
```

- Exact same code which handles interrupts

Even A Single Instruction Can Contain a Critical Section

thread #1 **thread #2**

counter = counter + 1; counter = counter + 1;

But what looks like one instruction in C gets compiled to:

```
mov counter, %eax
add $0x1, %eax
mov %eax, counter
```

- An interrupt after any instruction can cause a conflict
- We expect that the result of both running is 2
- Each has their own stack, but shares the use of registers. Saving registers onto the stack when a thread is paused
- Each thread has an inconsistent view if there is an incomplete computation left in a register on an interrupt

Critical Section and Mutual Exclusion

- Only one thread can execute a critical section at a time

  Interrupt Disabled
  - No interrupts: we would never interpret code working on a critical section
  - Can be done with privileged instructions
  - Prevents time slice ends (timer interrupts)
  - Prevent re entry of device driver code

    Downsides
    - Delays important operations
    - Bug may leave the program permanently disabled
    - Won't solve all sync problems
    - Can’t be done in user mode

  Other solutions
  - Avoid sharing data
  - Eliminate critical sections with atomic instructions (but difficult to only work with one instruction)
  - Use locks

Summary
- Processes too expensive for soe purposes
- Threads provide a cheaper alternative
- Threads communicate through memory
- Processes need IPC