Introduction to Inter-Process Communication

Introduction

- We can divide process interactions into two broad categories
    - 1) coordination of operations with other processes
        - synchronization
        - exchange of signals (kill(2))
        - control operations (fork, exec, ptrace)
    - 2) Exchange of date between processes
        - uni-directional data processing pipelines
        - bi-directional interactions

Simple Uni-Directional Byte Streams

- Pipe can be opened by a parent and inherited by a child, who simply reads standard input and writes standard output
    - ex: macro-processor | compiler | assembler > output
- Each program accepts a byte-stream input, and produces a byte-stream output that is a well defined function of the input
- **Each program in the pipeline operates independently,** and is unaware that the others exist
- **Byte streams are inherently unstructured** and depend on the implementation of the parser
- Similar results can be obtained by creating one file and using that file as an input to another program
- **Pipes are temporary files** with a few special features
    - If the reader exhausts all the data in the pipe but there is still an open file descriptor, the user does not get an EOF alert and instead the reader is blocked until more data becomes available or the write side is closed
    - **Flow Control:** If the writer gets too far ahead of the reader, the operating system might block the writer until the reader catches up -> and vice versa
    - Writing to a pipe that no longer exists is illegal -> SIGPIPE
    - **When the read and write file descriptors are both closed, the files are automatically deleted**
- **Disadvantages**
    - Because a pipeline represents a closed system, the only data privacy mechanisms tend to be the protections on the initial input files and final output files
    - There is generally no authentication or encryption of the data being passed (also compression)

Named Pipes and Mailboxes

- **Named pipe: (fifo(7)):** can be thought of as a persistent pipe, whose readers and writers can open it by name rather than inheriting it from a pipe(2) system call

- Named pipe can be used as a rendezvous point for unrelated processes, while pipe(2) must be initiated by the user
    - Readers and writers have no way of authenticating one-another's identities
    - Writes from multiple writers may be interspersed, with no indications of which bytes came when
    - Do not enable clean fail-overs from a failed reader to its successor
    - All readers and writers must be running on the same node
- Solution to the above problems: **Mailbox**: More general inter-process communication mechanism
    - Data is not a byte-stream. Rather, each write is stored and delivered as a distinct message -> gives some indication of which bytes came from where
    - Each write is accompanied by authenticated identification about its sender
    - Unprocessed messages remain the mailbox after the death of a reader so that another reader can retrieve it
    - Still subject to a single node/OS restriction
- Named pipes and mailboxes provide us with a step forward into a more general inter process communication method by allowing unrelated processes to call a single rendezvous point to relate to each other and send data.

General Network Communication

- socket(2): create an IPC end-point with an associated protocol and data model
- bind(2): associate a socket with a local network address
- connect(2): establish a connection to a remote network address
- listen(2): await an incoming connection request
- accept(2): accepts an incoming connection request
- send(2): sends a message over a socket
- recv(2): receives a message over a socket
- The above API form a foundation for high level communication/service models
    - Remote Procedure Calls: distributed request/response APIs
    - RESTful service models: layered on top of HTTP GETs and PUTs
    - Publish/Subscribe services: content based information flow
- Using more general networking models enables processes to interact with services all over the world -> but adds complexity
    - Ensuring interoperability with software running under different operating systems and different ISA becomes difficult
    - Dealing with security issues associated with exchanging data
    - Discovering the addresses of a constantly changing set of servers
    - Detecting and recovering node failures
- Protocol stacks may be layers deep and data may be processed/copied numerous times ->network communication may have limited throughput and high latency

Shared Memory

- Sometimes performance is more important than generality

    - Network drivers, MPEG decoders, and video rendering in a set top box are guaranteed to be local. Making these operations more efficient can greatly reduce processing power
    - Protocol interpreters, write-back cache, RAID implementation, and back-end drivers ina storage array are guaranteed to be local -> we want to significantly reduce the execution time per write operation
- Fastest and most efficient way to move data between processes is through shared memory
    - **create a fil**e for communication
    - **each process maps that file into its virtual address space**
        - **same physical memory location is addressed**
    - **the shared segment might be locked-down so that it is never paged out**
    - Communicating processes agree on a set of data structures in the shared segment
    - anything written in the shared memory segment will be **immediately visible to all of the processes** that have mapped in to their address space
- Once the shared segment has been created and mapped into the participating process' address space, the OS plays no role in the subsequent data exchanges
    - This can only be used between processes on the **same memory bus**
    - A bug in one process can easily destroy the communications data structures
    - No authentication of which data came from which process

Network Connections and Out-of-Band Signals

- In most cases, event completion can be reported simply by sending a message to the waiter
- Data sent down a network connection is FIFO
    - FIFO scheduling problem: delays waiting for the processing of earlier but longer messages
    - We want to make it possible so that an important message gains priority
- If the recipient was local, we might consider sending a signal that would invoke a registered handler, and flush all of the buffered data
    - Possible because the signal travels over a different channel than the buffered data -> **out-of-band**
- We can multiple channels
    - one heavily used channel for normal requests
    - another reserved for out-of-band requests
- Server on the far end periodically polls the out-of-band channel before taking requests from the normal communications channel
- Tradeoff between overhead to processing and how long it might take before noticing an out-of-band message

send(int sockfd, const void* buf, size_t len, int flags)

- system call used to transmit a message to another socket

- can only be used when the socket is in a connected state so that the intended recipient is known
- Difference between write(2): send has flags -> no flags means it's the same as write(2)
- For send() and sendto(), the message is found in buf and has length len
- The flags argument
    - MSG_CONFIRM: tell the link layer that forward progress happened -> you got a successful reply from the other side
    - MSG_DONTROUTE: Don't use a gateway to send out the packet
    - MSG_DONTWAIT: Enables non-blocking operation
    - MSG_EOR: terminates a record
    - MSG_MORE: caller has more data to send -> used with TCP sockets
    - MSG_NONSIGNAL: Don't generate a SIGPIPE (writing to pipe that no longer exists) signal if the peer on a stream-oriented socket has closed the connection -> EPIPE is still returned
    - MSG_OOB: Sends out-of-band data o sockets that support this notion

recv(int sockfd, void*buf, size_t len, int flags)

- used to receive messages from a socket, both connectionless and connection-oriented

Named Pipes (FIFOs)

- Basic concepts
    - Named pipes exist as a device special file in the file system
    - Processes of different ancestry can share data through a named pipe
    - When all I/O is done by sharing processes, the named pipe remains in the file system to use
- Creating a FIFO
    - directly from shell
        - mknod MYFIFO p
        - mkfifo a=rw MYFIFO
    - FIFO files can be quickly identified in a physical file system by the "p" indicator
    - In C, use the mknod() system call
- FIFO operations
    - Essentially the same as normal pipes except the open() operation should be used to open up a channel to the type
- Blocking Actions on a FIFO
    - Normally, **blocking occurs on a FIFO**: if the FIFO is opened for reading, the process will "block" until some other process opens it for writing -> other processes can run (can use the O_NONBLOCK flag)
- SIGPIPE signal
    - If a process tries to a pipe that has no reader, it will send a SIGPIPe signal to the kernel

mmap(void* addr, size_t length, int prot, int flags, int fd, off_t offset)

- creates a new mapping in the virtual address space of the calling process
    - naturally implements demand paging (the OS copies the disk page into physical memory only if an attempt to access it is made and that page is not already there) -> page faults
- The starting address for the new mapping is called in addr
- maps files or devices into memory (think of pages)

Chapter 25: Part II: Concurrency

- Imagine another peach
    - The goal is to find an approach that is both fast and correct
    - Ex: Best way to partition peaches: Fairness is found if people line up and wait for the peaches, efficient if everyone just grabs it
- **Thread**
    - an independent agent running around programs, doing things on the program's behalf
    - Threads access memory -> we must coordinate access to memory between threads -> think of the peaches
- The OS must support multi-threaded applications with primitives such as **locks and condition variables**
- The OS itself is a concurrent program

Chapter 26: Concurrency: An Introduction

- A multi-threaded application has more than one point of execution (i.e. multiple PCs, each of which is being fetched and executed from)
- *Each thread can be thought of as a separate process, except that they **share** the same address space
- The state of a single thread is therefore very similar to the state of a process
    - PC that tracks where the program is fetching instructions from
    - Private set of registers it uses for computation
    - If there are two threads running on a single processor, switching from thread to thread still requires a context switch
    - registers of the threads must be saved and restored
    - With processes, we saved states to a **process control block (PCB)** but with threads we will save the states to a **thread control block**
    - Major difference: Because the address space is shared, we do not have to perform switches for pages
    - Another major difference: Stack
        - In a single-threaded application our stack simply just resided at the bottom of our address space
        - However, in a multi-threaded application, there will be one stack per thread running

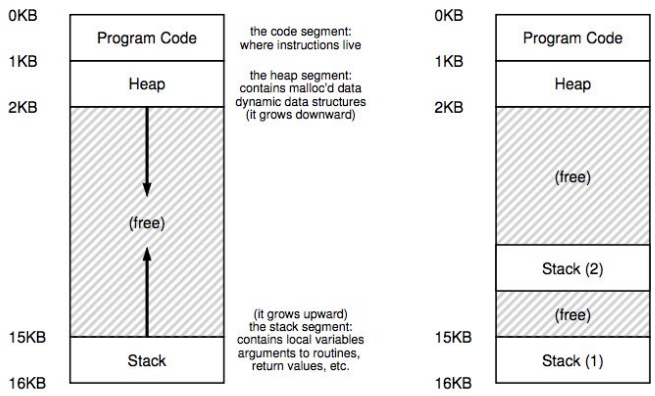

Figure 26.1: **Single-Threaded And Multi-Threaded Address Spaces**

-

26.1: Why Use Threads?

- Why should you use threads at all?
    - **Parallelism**: You can dramatically speed up processes of calculation through transforming your standard single-threaded program into a program that does similar work on multiple CPUs
        - Use one thread per CPU
    - **Avoid blocking program progress due to slow I/O:**
        - Instead of waiting for the process to finish and handle its I/O, yout application might want to do something else
        - Threading enables overlap of I/O with other activities within a single program
    - Threads share the same address space so it makes it very easy to share data
    - Processes are a more logical choice if you want to ensure more privacy and isolate the processes that are running

26.2: An Example: Thread Creation

```c
#include <stdio.h>
#include <assert.h>
#include <pthread.h>

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int
main(int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf("main: begin\n");
    rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
    rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
    // join waits for the threads to finish
    rc = pthread_join(p1, NULL); assert(rc == 0);
    rc = pthread_join(p2, NULL); assert(rc == 0);
    printf("main: end\n");
    return 0;
}
```

Figure 26.2: **Simple Thread Creation Code (t0.c)**

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
waits for T1		
	runs	
	prints "A"	
	returns	
waits for T2		
		runs
		prints "B"
		returns
prints "main: end"		

Figure 26.3: **Thread Trace (1)**

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
	runs	
	prints "A"	
	returns	
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
returns immediately; T1 is done		
waits for T2		
returns immediately; T2 is done		
prints "main: end"		

Figure 26.4: **Thread Trace (2)**

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
	runs	
	prints "A"	
	returns	
waits for T2		
returns immediately; T2 is done		
prints "main: end"		

Figure 26.5: **Thread Trace (3)**

- Which thread the OS is going to run depends on the scheduler -> immediately after a thread is created, it can run first

- One way to think of thread creation is that it is a bit like making a function call
    - However, instead of first executing the function and then returning to the caller, the system instead creates a new thread of execution for the routine that is being called and then runs independently on the caller, perhaps before returning from the create, but also perhaps much later
    - What thread is run depends on what the scheduler decides to run

26.3: Why It Gets Worse: Shared Data

```c
#include <stdio.h>
#include <pthread.h>
#include "mythreads.h"

static volatile int counter = 0;

//
// mythread()
//
// Simply adds 1 to counter repeatedly, in a loop
// No, this is not how you would add 10,000,000 to
// a counter, but it shows the problem nicely.
//
void *
mythread(void *arg)
{
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}

//
// main()
//
// Just launches two threads (pthread_create)
// and then waits for them (pthread_join)
//
int
main(int argc, char *argv[])
{
    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", counter);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");

    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done with both (counter = %d)\n", counter);
    return 0;
}
```

Figure 26.6: **Sharing Data: Uh Oh (t1.c)**

- Interaction between threads can become very difficult to manage
- Example above: two threads wish to update a shared global variable
    - Sometimes everything works and we get the desired counter that we want
    - However, the values that are outputted seem to be fairly random

- Tip: Know and use your tools
    - **Disassembler:** we can run objdump on Linux to see the assembly code
    - You can also use memory checking programs such as valgrind or purify

26.4: The Heart of the Problem: Uncontrolled Scheduling

OS	Thread 1	Thread 2	(after instruction) PC	%eax	counter
	before critical section		100	0	50
	mov 0x8049a1c, %eax		105	**50**	50
	add $0x1, %eax		108	**51**	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	**50**	50
		add $0x1, %eax	108	**51**	50
		mov %eax, 0x8049a1c	113	51	**51**
interrupt					
save T2's state					
restore T1's state			108	51	51
	mov %eax, 0x8049a1c		113	51	**51**

Figure 26.7: **The Problem: Up Close and Personal**

-
- **Race Condition:** the results depend on the timing execution of the code
    - with some luck, we get the wrong results sometimes
    - Race conditions do not create deterministic solutions
    - Because multiple threads executing this code can result in a race condition, we call this code a **critical section**
        - Piece of code that accesses a shared variable (a shared resource) and must no be concurrently executed by one thread
- **Mutual Exclusion**
    - Property guarantees that if one thread is executing within the critical section, the officers will be prevented from doing so

26.5: The Wish for Atomicity

- One way to solve this problem would be to have more powerful instructions that, in a single step, did exactly whatever we needed done and thus remove the possibility of an untimely interrupt that leads to race conditions
- We can have hardware that guarantees that it can not be interrupted mid instruction
    - Atomically, in this context, means “as a unit”, which sometimes we take as all or none
    - We’d like to execute mov, add, and mov sequence atomically
- Instead, we will ask the hardware for a few useful instructions upon which we can build a general set of what we can **synchronization primitives**

    - By using hardware synchronization primitives, we can build multi-threaded code that accesses critical sections in a synchronized and controlled manner
- Tip: Use Atomic Operations
    - Idea behind making a series of actions atomic is simply expressed with the phrase “all or nothing”;
        - it should either appear as if all of the actions you wish to group together occurred, or that none of them occurred -> non visible in between
    - Grouping of a single atomic action is called a **transaction**
    - For example, file systems use techniques such as journaling or copy-on-write in order to atomically transition their on-disk state, critical for operating correctly in the face of system failures

26.6: One More Problem: Waiting for Another

- One thread must wait for another to complete some action before it continues
    - this action arises when a process performs a disk I/O and is put to sleep; when the I/O completes, the process must be woken up
- Mechanisms to support this sleeping/waking algorithm is needed

Aside: Key Concurrency Terms

- **Critical Section:** a piece of code that accesses a shared resource, usually a variable or data structure
- **Race condition**: Arises if multiple threads of execution to update the shared data structure, leading to a surprising outcome
- **Indeterminate:** A program that consists of one or more race conditions
    - the outcome depends from run to run so it is indeterminate
- **Mutual Exclusion Primitives:** Using these primitives avoid threads from encountering race conditions because it guarantees that only a single thread enters the critical section -> deterministic outputs

Chapter 27: Interlude: Thread API

Chapter 27.1: Thread Creation

- in POSIX, we can easily create a thread using the pthread_create() API
    - The second argument, attr, is used to specify any attributes this thread might have
        - Some examples include setting the stack size or perhaps information about the scheduling priority of the thread
        - In most cases, the default is fine so we usually just say NULL
    - Third argument: Simply put, which function should this thread start running in?
        - We call this function a function pointer in C
    - Fourth argument: arg is exactly the argument to be passed to the function where the thread begins execution

    - The thread, once created, can simply cast its argument to the type it expects and thus unpack the arguments you desire

27.2: Thread Completion

- If you want to wait for a thread to complete -> use the pthread_join() call
- Two arguments
    - pthread_t
        - specifies which thread to wait for, initialized by the routine pthread_create()
    - A pointer to the return value you expect to get back. -> routine an return anything, so it is defined as a pointer to void
        - if we don't care about the return value, just pass null in
- *NEVER return a pointer which refers to something allocated on another thread's call stack
    - Example: if variable r is allocated on the stack, when it returns, it is automatically deallocated and thus passes a pointer back to a now deallocated variable
- A thread doesn't always need to join
    - Example: If a server creates a number of working threads

User-Mode Thread Implementation

Introduction

- For a long time, processes were the only unit of parallel computation
    - processes are very expensive to create and dispatch, due to the fact that each one has their own virtual address space
    - Cannot share in-memory resources with parallel processes
- Threads
    - An independently schedulable unit of execution
    - runs within the address space of a process
    - has access to all of the system resources owned by that process
    - has its own general registers
    - has its own stack (within the owning process' address space)

A Simple Threads Library

- The basic model is
    - Each time a new thread is created
        - allocate memory for a fixed size thread private stack from the heap
        - We create a new thread descriptor that contains identification information, scheduling information, and a pointer to the stack
        - add new thread to a ready queue
    - When a thread calls yield() or sleep() we save its general registers (on its own stack), and then select the next thread on the ready queue

    - To dispatch a new thread, we simply restore its saved registers, including the stack pointer, and return from the call that caused it to **yield**
    - If a thread called sle**e**p(), we could remove it from the ready queue. When it was re-awakened, we could put it back onto the ready queue
    - When a thread exited, we would free its stack and thread descriptor
- Eventually, people wanted preemptive scheduling to ensure good interactive response and prevent buggy threads from tying up the application
    - Linux processes can schedule the delivery of SIGALARM timer signals
    - Before dispatching a thread, we can schedule a SIGALARM that will interrupt the thread if it runs too long
    - If a thread runs too long, the SIGALARM handler can yield on behalf of that thread, saving its state, moving on to the next thread in the ready queue
- The addition of preemptive scheduling created new problems for critical sections that required before-or-after, all or none serialization
    - use sigprocmask(2) to block signals

Kernel Implemented Threads

- Two fundamental problems with implementing threads in a user mode library
    - what happens when a system call blocks?
        - If a user-mode thread issues a system call that blocks (e.g. open or read), the process is blocked until that operation completes. This means that when a thread blocks, all threads stop executing. -> if threads are implemented in user mode, the OS has no knowledge of this
    - exploiting multi-processors
        - If the CPU has multiple execution cores, the OS can schedule processes on each to run in parallel. However, if the OS is not aware that a process is comprised of multiple threads -> threads may not be executed in parallel
- Therefore, threads must be implemented by the OS rather than by a user-mode thread library

Performance Implication

- If non-preemptive scheduling can be used, user mode threads operating in with a sleep/yield model are much more efficient than doing context switches
- If preemptive scheduling is used, the costs of setting alarms and servicing the signals is much greater than the cost of simply allowing the OS to perform scheduling

Lecture

IPC: Messages vs Streams

- streams
    - a continuous stream of bytes
    - read or write few or many bytes at a time

    - write and read buffer sizes are unrelated
    - stream may contain app-specific record delimiters
- messages (aka datagrams)
    - a sequence of distinct messages
    - each message has its own length (subject to limits)
    - message is typically read/written as a unit
    - delivery of a message is typically all-or-nothing

IPC: Flow Control

- queued messages consume system resources
    - buffered in the OS until the receiver asks for them
- many things can increase required buffer space
    - fast sender, non-responsive receiver
- must be a way to limit required buffer space
    - back pressure: block sender or refuse message
    - receiver side: drop connection or messages
    - this is usually handled by network protocols
- mechanisms to report stifle/flush to sender

IPC: Reliability and Robustness

- reliable delivery (e.g. TCP vs UDP)
    - networks can lose requests and responses
- a sent message may not be processed
    - receiver invalid, dead, or not responding
- When do we tell the sender “OK”
    - queued locally? added to receivers input queue?
    - receiver has read? receiver has acknowledged?
- how persistent is system in attempting to deliver?
    - retransmission, alternate routes, back-up servers
- do channel/contents survive receiver restarts?
    - can new server instance pick up where the old left off?

Simplicity: Pipelines

- data flows through a series of programs
    - ex: ls | grep | sort | assembler
- data is a simple byte stream
    - buffered in the operating system
    - no need for intermediate temporary files
- there are no security/privacy/trust issues
    - all under a single user
- error conditions
    - Input: EOF, Output: SIGPIPE

Generality: sockets

- Connections between address/ports
    - connect/listen/accept
    - lookup: registry, DNS, service discovery protocols
- Many data options
    - reliable TCP or best effort datagrams (UDP)
    - streams, messages, remote procedure calls
- Complex flow control and error handling
    - retransmissions, timeouts, node failures
    - possibility of reconnection or fail-over
- Trust/security/privacy/integrity

Halfway: mailboxes, named piped

- A client/server rendezvous point
    - a name corresponds to a service
    - a server awaits client connections
    - once open, it may be as simple as a pipe
    - OS may authenticate message sender
- Limited fail-over capability
    - if server dies, another can take its place
    - but what about in progress requests?
- client/server must be on the same system

Ludicrous Speed - Shared Memory

- shared read/write memory segments
    - mmap(2) into multiple address spaces
        - any process can create/map shared segments
        - perhaps locked in physical memory
    - applications maintain circular buffers
        - data transferred w/ ordinary instructions
        - OS is not involved in data transfer
    - Simplicity, ease of use
    - Reliability, security, caveat emptor
    - genearlity ,.. locals only!

IPC: synchronous and asynchronous

- synchronous operations
    - writes block until message sent/delivered/received
    - reads block until a new message is available
    - easy for programmers, but no parallelism
- asynchronous operations
    - writes return when system accepts message

        - no confirmation of transmission/delivery/reception
        - requires auxiliary mechanism to learn of errors
    - reads return promptly if no message available
        - requires auxiliary mechanism to learn of new messages
        - often involves “wait for any of these”

A Brief History of Threads

- processes are very expensive
    - to create: they own resources
    - to dispatch: they have address spaces
- different processes are very distinct
    - they cannot share the same address space
    - they cannot (usually) share resources
- Not all programs require strong separation
    - cooperating parallel threads of execution
    - all are trusted, part of a single program

What is a thread?

- strictly a unit of execution/scheduling
    - each thread has its own stack, PC, registers
- Multiple threads can run in a process
    - they all share the same code and data space
    - they all have access to the same resources
    - this makes it cheaper to create and run
- Sharing the CPU between multiple threads
    - user level threads (w/ voluntary yielding)
        - periodically changes from one thread’s address space to another
    - scheduled system threads (w/ preemptions)
    - user level threads vs system level threads
        - *** user mode threads have no parallelism because if one thread is blocked, the others are blocked as well
        - kernel mode threads can be slower because of the context switching that is necessary, but threads will be able to run in parallel

When to use processes

- running multiple distinct programs
- creation/destruction are limited
- running agents with distinct privileges
- limited interactions and shared resources
- prevent interference between processes
- firewall one from failures of the other

When to use threads

- parallel activities in a single program
- frequent creation and destruction
- all can run with same privileges
- they need to share resources
- they exchange messages/signals
- no need to protect from each other (all in the same user, like pipes!)

Asynchronous Operations vs Multithreading

- Mutlthreading: everything runs synchronously in parallel while it is blocked
- Asynchronous waits for one to finish

Kernel vs User-Mode Threads

- Does OS schedule threads or processes?
- Advantages of Kernel implemented threads
    - multiple threads can truly run in parallel
    - one thread blocking does not block others
    - OS can enforce priorities and preemption
    - OS can provide atomic sleep/wakeup/signals
- Advantages of library implemented threads
    - fewer system calls
    - faster context switches
    - ability to tailor semantics to application needs

Thread State and Thread Stacks

- each thread has its own registers, PC, and PS
- each thread must have its own stack area
- max size specified when thread is created
    - a process can contain many threads
    - they cannot all grow towards a single hole
    - thread creator must know max required stack size
    - stack space must be reclaimed when thread exits
- procedure linkage conventions are unchanged
- Thread stacks are a fixed size that is specified when the thread is created

Thread Safety - Reentrancy

- Thread safe routines must be reentrant
    - any routine can be called by multiple threads
    - concurrent or interspersed execution
    - signals can also cause reentrancy
- State cannot be saved in static variables
    - errno
    - optarg
- transient state can be safely allocated on stack

- persistent session state must be client-owned
    - open returns a descriptor
    - descriptor is passed to all subsequent operations

Thread Safety-Shared Data/Events

- threads operate in a single address space
    - automatic (stack) locals are private
    - storage (from thread-safe malloc) can be private
    - read-only data causes no problems
    - shared read/write data is a problem
- signals are sent to processes
    - delivered to first available thread
    - chosen recipient may not have been expecting it
- a call to exit(2) terminates all threads

Synchronization - evolution of problem

- batch processing - a serially reusable resource
    - process A has tape drive, process B must wait
    - process A updates file first, then process B
- cooperating processes
    - exchanging messages with one-another
    - continuous updates against shared files
- shared data and multi-threaded computation
    - interrupt handlers, symmetric multi-processors
    - parallel algorithms, preemptive scheduling
- network-scale distributed computing

Race Conditions

- shared resources and parallel operations
    - where outcome depends on execution order
    - these happen all the time, most don't matter
- some race conditions affect correctness
    - conflicting updates (mutual exclusion)
    - check/act rates (sleep/wakeup problem)
    - multi-object updates
    - distributed decisions
- each of these classes can be managed
    - if we recognize the race condition and danger

Non-Deterministic Execution

- processes block for I/O or resources
- time-slice and preemption
- interrupt service routines

- unsynchronized execution on another core
- queuing delays
- time required to perform I/O operations
- message transmission/delivery time

What is “Synchronization”

- true parallelism is imponderable
    - pseudo-parallelism may be good enough
    - identify and serialize key points of interaction
- there are two interdependent problems
    - critical section serialization
    - asynchronous completions
- they are often discussed a single problem
    - many mechanisms simultaneously solve both
    - solution to either requires solution to the other

Problem 1: Critical Sections

- A resource shared by multiple threads
    - multiple concurrent threads, processes or CPUs
    - interrupted code and interrupt handler
- Use of the resource changes its state
    - contents, properties, relation to other resources
    - updates are non-atomic (or non-global)
- correctness depends on execution order
    - when scheduler runs/preempts which threads
    - true parallelism
    - relative timing of independent events
    - leading to indeterminate results

Reentrant & MT-safe code

- consider a simple recursive routine
    - int factorial(x) { tmp = factorial(x-1); return x*tmp}
- consider a possibly multi-threaded routine
    - void debit(amt) {tmp=bal-amt; if(amt>=0) bal=tmp)}
- neither would work if tmp was shared/static
    - must be dynamic, each invocation has own copy
    - this is not a problem with read-only information
- some variables must be shared

Recognizing Critical Sections

- Generally involves updates to object state
    - may be updates to a single object
    - may be related updates to multiple objects

- Generally involves multi-step operations
    - object state inconsistent until operation finishes
    - preemption compromises object or operation
- Correct operation requires mutual exclusion
    - only one thread at a time has access to object(x)
    - client 1 completes before client 2 starts

Two Types of Atomicity

- Before or After (mutual exclusion)
    - A enters critical section before B starts
    - A enters critical section after B completes
    - Most of the time, we don't care who goes first
- All or None (atomic transactions)
    - an update that starts will **complete w/o interruption**
        - **therefore, all or none**
    - an uncompleted update has no effect
- Atomic: means uncuttable