IPC, Threads, Races, Critical Sections

7A. Inter-Process Communication
3T. Threads
7B. The Critical Section Problem

Inter-Process Communication
• the exchange of data between processes
• Goals
  – simplicity
  – convenience
  – generality
  – efficiency
  – security/privacy
  – robustness and reliability
• some of these turn out to be contradictory

OS Support For IPC
• Wide range of semantics
  – may appear to be another kind of file
  – may involve very different APIs
    • provide more powerful semantics
    • more accurately reflect complex realities
• Connection establishment mediated by the OS
  – to ensure authentication and authorization
• Data exchange mediated by the OS
  – to protect processes from one-another
  – to ensure data integrity and authenticity

Typical IPC Operations
• channel creation and destruction
• write/send/put
  – insert data into the channel
• read/receive/get
  – extract data from the channel
• channel content query
  – how much data is currently in the channel
• connection establishment and query
  – control connection of one channel end to another
  – who are end-points, what is status of connections

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
  - ls | grep | sort | mail
  - macro processor | complier | 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 control of a single user
- error conditions
  - input: End of File output: SIGPIPE

**Generality: sockets**

- connections between addresses/ports
  - connect/listen/accept
  - lookup: registry, DNS, service discovery protocols
- many data options
  - reliable (TCP) or best effort data-grams (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
  - we have a whole lecture on this subject

**half way: mail boxes, named pipes**

- 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 same system

**Ludicrous Speed – Shared Memory**

- shared read/write memory segments
  - `mmap(2)` into multiple address spaces
  - perhaps locked in physical memory
  - applications maintain circular buffers
  - OS is not involved in data transfer
- simplicity, ease of use ... your kidding, right?
- reliability, security ... caveat emptor!
- generality ... 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" operation
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 the cheaper to create and run
- sharing the CPU between multiple threads
  - user level threads (w/ voluntary yielding)
  - scheduled system threads (w/preemption)

When to use processes

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

Using Multiple Processes: cc

# shell script to implement the cc command

cpp $1.c | cc1 | ccopt > $1.s
as $1.o
ld /lib/crt0.o $1.o /lib/libc.so
mv a.out $1
rm $1.s $1.o

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 many messages/signals
- no need to protect from each other

Using Multiple Threads: telnet

```c
netfd = get_telnent_connection(host);
pthread_create(&tid, NULL, writer, netfd);
reader(netfd);
pthread_join(tid, &status);
...
reader(fd) { int cnt; char buf[100];
    while( cnt = read(fd, buf, sizeof(buf) > 0 )
        write(fd, buf, cnt);
    }
}
writer(fd) { int cnt; char buf[100];
    while( cnt = read(fd, buf, sizeof(buf) > 0 )
        write(1, buf, cnt);
    }
```
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, PS, PC
- 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

UNIX stack space management

- Data segment starts at page boundary after code segment
- Stack segment starts at high end of address space
- Unix extends stack automatically as program needs more.

Thread Stack Allocation

- Data segment grows up; Stack segment grows down
- Both grow towards the hole in the middle. They are not allowed to meet.

Thread Safety - Reentranct

- 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
  - e.g. errno ... getting around C scalar returns
  - e.g. optarg ... implicit session state
- 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 - serially reusable resources
  - 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

The benefits of parallelism

- improved throughput
  - blocking of one activity does not stop others
- improved modularity
  - separating complex activities into simpler pieces
- improved robustness
  - the failure of one thread does not stop others
- a better fit to emerging paradigms
  - client server computing, web based services
  - our universe is cooperating parallel processes

What's the big deal?

- sequential program execution is easy
  - first instruction one, then instruction two, ...
  - execution order is obvious and deterministic
- independent parallel programs are easy
  - if the parallel streams do not interact in any way
- cooperating parallel programs are hard
  - if the two execution streams are not synchronized
    - results depend on the order of instruction execution
    - parallelism makes execution order non-deterministic
    - results become combinatorially intractable

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 races (sleep/wakeup problem)
  - multi-object updates (all-or-none transactions)
  - distributed decisions based on inconsistent views
- 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 end 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 as a single problem
  - many mechanisms simultaneously solve both
  - solution to either requires solution to the other
A Synchronization Problem
(multi-thread, shared memory, circular buffer)
write(buf, toSend):
  while toSend > 0
    wait (nextWrite > lastRead)
    free = endOfBuffer – nextWrite
    count = min(free, toSend)
    copy(buf, nextWrite, count)
    nextWrite += count
    toSend -= count;
read(buf, desired):
  while desired > 0
    wait (nextWrite > lastRead)
    avail = nextWrite – lastRead
    count = min(avail, desired)
    copy(lastRead, buf, count)
    lastRead += count
    desired -= count

Critical Section - updating a file
Process #1
  remove( "database" );
  fd = create( "database" );
  write(fd, newdata, length);
  close(fd);

Process #2
  fd = open( "database", READ);
  count = read(fd, buffer, length);
  ...

What could go wrong with an add?

thread #1
  counter = counter + 1;
  mov counter, %eax
  add $0x1, %eax
  mov %eax, counter

thread #2
  counter = counter + 1;
  mov counter, %eax
  add $0x1, %eax
  mov %eax, counter

Achieving Mutual Exclusion

pthread_mutex_t lock;
pthread_mutex_init(&lock, NULL);
...
if (pthread_mutex_lock(&lock) == 0) {
  counter = counter + 1;
pthread_mutex_unlock(&lock);
}
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(s)
  - client 1 completes before client 2 starts

Critical Sections in Operating System

- Shared data used by concurrent threads
  - process state variables
  - resource pools
  - device driver state
- logical parallelism
  - created by preemptive scheduling
  - asynchronous interrupts
- physical parallelism
  - shared memory, symmetric multi-processors

Two Types of Atomicity

- Before or After (mutual exclusion)
  - A enters critical section before B starts
  - A enters critical section after B completes
- All or None (atomic transactions)
  - an update that starts will complete w/o interruption
  - an uncompleted update has no effect

Assignments

- Project
  - assemble and bring up your Edison
- Reading
  - AD 27.3-4 (synchronization APIs)
  - AD 28 (locks and implementation)
  - AD 30-30.1 condition variables

Supplementary Slides

IPC: communication fan-out

- point-to-point/unicast (1->1)
  - channel carries traffic from one sender to one receiver
- multi-cast (1->N)
  - messages are sent to specified receivers or group
- broadcast (1->N)
  - messages are sent to all receivers in a community
- publish/subscribe (N->M)
  - messages are distributed/filtered based on content
  - routing can be at sender, receiver, and in-between
IPC: in-band vs. out-of-band

• in-band messages
  – messages delivered in same order as sent
  – message n+1 won't be seen till after message n

• out-of-band messages
  – messages that leap ahead of queued traffic
  • often used to announce errors or cancel requests
  – use priority to “cut” ahead in the queue
  • priority must be honored on each link in the path
  – deliver them over a separate channel
  • a separate message channel, or perhaps a signal

IPC examples: UNIX sockets

• more powerful than pipes
  – can be bound to various protocols
  • tcp ... reliable stream, network protocol
  • udp ... unreliable datagrams, network protocol
  • unix ... named pipes
  – more versatile connection options
  • connect, listen, accept, broadcast, multicast

• both stream and message semantics
  – read/write ... synchronous stream
  – send/recv ... synchronous datagrams
  • socket is destroyed when creator dies

IPC examples: mail boxes

• named message queues
  – associated with a particular receiving process
  – any process can send messages to any mailbox
• additional semantics vary with implementations
  – trusted identification of sending process
  – synchronous and asynchronous options
  – confirmation of delivery (or receipt)
  – contents of queue may survive a kill and restart
• messages typically buffered in the OS
  – some flow control is usually provided

Synchronization – a Parable

• consider the “Garden of Eden”
  – we were warned of the “knowledge of good and evil”
  • which would forever end our innocent lives in paradise
  – but we were then tempted to eat that fruit
  • it would open our eyes, and give us God-like knowledge
  – we ate, lost our innocence, were banished from paradise
• consider the “Garden of computation”
  – we were warned about cooperating parallel processes
  • which would forever cost us our innocence
  • but we also saw the power that knowledge could unleash
  – again we took the bait, and lost our innocence
  • programming, once simple, has been made impossibly difficult