Operating Systems Principles
IPC, Threads, Races, Critical Sections

Mark Kampe
(markk@cs.ucla.edu)

IPC, Threads, Races, Critical Sections

7A. Threads
7B. Inter-Process Communication
7C. Critical Sections
7D. Asynchronous Event Completions

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

When to use threads

- parallel activities <u>in a single program</u>
- 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

Processes vs. Threads – trade-offs

- if you use multiple processes
  - your application may run much more slowly
  - it may be difficult to share some resources
- if you use multiple threads
  - you will have to create and manage them
  - you will have serialize resource use
  - your program will be more complex to write
- TANSTAAFL
  - there ain't no such thing as a free lunch

Thread state and thread stacks

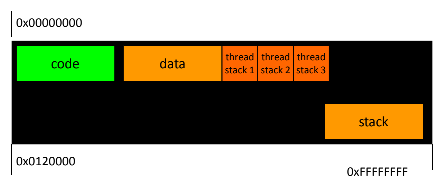
- each thread has its own registers, PS, PC
- each thread must have its own stack area
- maximum size specified when thread is created
  - a process can contain many threads
  - they cannot all grow towards a single hole
  - thread creator must know max required stack size
  - stack space must be reclaimed 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.

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

Thread Stack Allocation



Inter-Process Communication

- the exchange of data between processes
- Goals
  - simplicity
  - convenience
  - generality
  - efficiency
  - robustness and reliability
- some of these are contradictory

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
  - sender side: block sender or refuse message
  - receiving side: stifle sender, flush old 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 or best effort data-grams
  - 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
  - mapped 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!

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 <u>is</u> 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
- actually two interdependent problems
  - critical section serialization
  - notification of asynchronous completion
- they are often discussed as a single problem
  - many mechanisms simultaneously solve both
  - solution to either requires solution to the other
- they can be understood and solved separately

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 (e.g. multi-processor) parallelism
  - relative timing of independent events

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 (tmp >=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
  - and proper sharing often involves critical sections

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

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

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

Two Aspects of Atomicity

- there is Before or After atomicity
  - A enters critical section before B starts
  - A enters critical section after A completes
  - there is no overlap
- there is All or None atomicity
  - an update that starts will complete
  - an uncompleted update has no effect
- correctness generally needs both of these

Problem 2: asynchronous completion

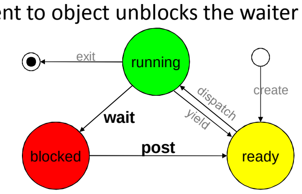
- most procedure calls are synchronous
  - we call them, they do their job, they return
  - when the call returns, the result is ready
- many operations cannot happen immediately
  - waiting for a held lock to be released
  - waiting for an I/O operation to complete
  - waiting for a response to a network request
  - delaying execution for a fixed period of time
- we call such completions asynchronous

Approaches to Waiting

- spinning … "busy waiting"
  - works well if event is independent and prompt
  - wasted CPU, memory, bus bandwidth
  - may actually delay the desired event
- yield and spin … "are we there yet?"
  - allows other processes access to CPU
  - wasted process dispatches
  - works very poorly for multiple waiters
- either may still require mutual exclusion

Condition Variables

- create a synchronization object
  - associate that object with a resource or request
  - requester blocks awaiting event on that object
  - upon completion, the event is "posted"
  - posting event to object unblocks the waiter



Blocking and Unblocking

- blocking
  - remove specified process from the "ready" queue
  - yield the CPU (let scheduler run someone else)
- unblocking
  - return specified process to the "ready" queue
  - inform scheduler of wakeup (possible preemption)
- only trick is arranging to be unblocked
  - because it is so embarrassing to sleep forever
  - the condition variable should ensure this

Awaiting Asynchronous Events

```
pthread_mutex_t lock = PTHEAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHEAD_COND_INITIALIZER;
…
pthread_mutex_lock(&lock);
while (ready == 0)
    pthread_cond_wait(&cond, &lock);
pthread_mutex_lock(&lock)
…
                        if (pthread_mutex_lock(&lock)) {
                                ready = 1;
                                pthread_mutex_signal(&cond);
                                pthread_mutex_unlock(&lock);
                        }
```


Waiting Lists

- Who wakes up when a CV is signaled
  - pthread_cond_wait … at least one blocked thread
  - pthread_cond_broadcast … all blocked threads
- this may be wasteful
  - if the event can only be consumed once
  - potentially unbounded waiting times
- a waiting queue would solve these problems
  - each post wakes up the first client on the queue

assignments

- reading for the next lecture
  - Arpaci ch 28 … Locks

Supplementary Slides

Using Multiple Processes: cc

```
# shell script to implement the cc command
cpp $1.c | cc1 | ccopt > $1.s
as $1.s
ld /lib/crt0.o $1.o /lib/libc.so
mv a.out $1
rm $1.s $1.o
```

Using Multiple Threads: telnet

```
netfd = get_telnet_connection(host);
pthread_create(&tid, NULL, writer, netfd);
reader(netfd);
pthread_join(tid, &status);
...
reader( fd ) { int cnt; char buf[100];
        while( cnt = read(0, 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);
}
```


Synchronization Objects

- combine exclusion and (optional) waiting
- operations implemented safely
  - with atomic instructions
  - with interrupt disables
- exclusion policies (one-only, read-write)
- waiting policies (FCFS, priority, all-at-once)
- additional operations (queue length, revoke)