Secure Operating System Design and Implementation
Concurrency

Jon A. Solworth

Dept. of Computer Science
University of Illinois at Chicago

February 1, 2011

Part I

Concurrency Background

Concurrency

Concurrency plays a fundamental role in Operating Systems

- Concurrency is inherent in OSs
  - OSs support multiple users concurrently
  - OSs support multiple programs running concurrently
  - I/O devices which operate concurrently with CPUs
  - Multiple CPUs (Cores)

Concurrency

Concurrency plays a fundamental role in Operating Systems

- Concurrency is inherent in OSs
  - OSs support multiple users concurrently
  - OSs support multiple programs running concurrently
  - I/O devices which operate concurrently with CPUs
  - Multiple CPUs (Cores)
- Concurrency difficult to get right
  - Race conditions extraordinarily difficult to analyze
  - Concurrency explodes the number of interactions
  - Concurrency occurs at the hardware level, but programming abstractions are at the software level

Concurrency

Concurrency plays a fundamental role in Operating Systems

- Concurrency is inherent in OSs
  - OSs support multiple users concurrently
  - OSs support multiple programs running concurrently
  - I/O devices which operate concurrently with CPUs
  - Multiple CPUs (Cores)
- Concurrency difficult to get right
  - Race conditions extraordinarily difficult to analyze
  - Concurrency explodes the number of interactions
  - Concurrency occurs at the hardware level, but programming abstractions are at the software level
- Concurrency is difficult to exploit
  - Hard to find coarse grain concurrency
  - Hard to efficiently exploit fine-grain concurrency (beyond ILP)

Notation for sequential and concurrent execution

op_i be an operation, where $i \geq 0$

$op_0; op_1$ mean sequential execution, op_0 executes and then op_1

$op_0 || op_1$ means that op_0 executes concurrently with op_1

Atomicity

Fundamental to any discussion of concurrency is atomicity.

Definition (atomic operation)

A sequence of operations is atomic if either all the results of the operations are visible to external entities or none of them are.

What is atomic on a computer?

Architecture Machine instructions

Atomicity

Fundamental to any discussion of concurrency is atomicity.

Definition (atomic operation)

A sequence of operations is atomic if either all the results of the operations are visible to external entities or none of them are.

What is atomic on a computer?

Architecture Machine instructions

OS System calls

Atomicity

Fundamental to any discussion of concurrency is atomicity.

Definition (atomic operation)

A sequence of operations is atomic if either all the results of the operations are visible to external entities or none of them are.

What is atomic on a computer?

Architecture Machine instructions

OS System calls

- Its up to the architects to make machine instructions atomic,

Atomicity

Fundamental to any discussion of concurrency is atomicity.

Definition (atomic operation)

A sequence of operations is atomic if either all the results of the operations are visible to external entities or none of them are.

What is atomic on a computer?

Architecture Machine instructions

OS System calls

- Its up to the architects to make machine instructions atomic,
- up to OS designers to make system calls atomic

Atomicity

Fundamental to any discussion of concurrency is atomicity.

Definition (atomic operation)

A sequence of operations is atomic if either all the results of the operations are visible to external entities or none of them are.

What is atomic on a computer?

Architecture Machine instructions

OS System calls

- Its up to the architects to make machine instructions atomic,
- up to OS designers to make system calls atomic
- Syscall atomicity is provided in a variety of ways

Outcome of concurrent execution is non-deterministic

Assuming each op_i is atomic

- let $op_0 || op_1$ results in either
  - $op_0; op_1$ or
  - $op_1; op_0$
- Consider $(op_0; op_1) || (op_2; op_3)$ is equivalent to one of the following
  - $op_0; op_1; op_2; op_3$ or
  - $op_0; op_2; op_1; op_3$ or
  - $op_0; op_2; op_3; op_1$ or
  - $op_2; op_0; op_1; op_3$ or
  - $op_2; op_0; op_3; op_1$ or
  - $op_2; op_3; op_0; op_1$
- Note that the sequential ordering is preserved as a partial ordering

Sequential, Concurrency vs. Parallelism

Definition

Consider a set of operations to be executed $op_0, op_1, \ldots op_n$ and an execution order which is a partial order. If $op_i < op_j$ then in any valid execution the results of op_i are seen by op_j.

Definition

In a sequential execution, the execution order is a total order.

Definition

In a concurrent execution, the execution order is not a total order.

Definition

in a parallel execution, some operations are executed simultaneously.

Synchronization

- the purpose of synchronization is to impose additional ordering on executions
- For example, given a critical section once the critical section is entered by process p, no other process can enter it until p exits the critical section.
- A race condition occurs when there is inadequate synchronization and undesirable executions are possible.
- Other problems include deadlock and starvation.
- Starvation is generally handled by priority aging

Not all concurrency is hard

- Many things going on at the same time does not necessarily cause problems
- Problem is only when concurrent entities interact
- That is when one entity influences the behavior of another
- For example, one entity writes a location that another reads
- Or even one entity writes a location that another writes

Memory concurrency operations

Read and Write are atomic operations

Read-Read	Read X || Read X Value read does not depend other read
Read-Write	Read X || Write X Value read depends on whether the read occurs before/after write
Write-Read	same as Read-Write
Write-Write	Write X || Write X Last value written depends on order

Process concurrency

What are the process concurrency issues?

- Each process has its own private address space
- No process can see another process's memory
- Hence there are no memory races
- OK to preempt a process
- Only way processes interact is via system calls
- system calls atomic
- Only system calls semantics determine process issues with concurrency
- Above assumed that there is no shared memory or threads

Kernel concurrency

What are the sources of kernel concurrency?

- Multiple CPUs (multi-core CPUs)
- Interrupts
- Doing something while waiting for an external event such as
  - Network packet
  - Keyboard press
  - Disk access
  - Time

  In a modern OS, most processes are waiting for external events.

Part II

Monitors

Monitor

The classical kernel is represented as a Monitor.

Definition (Monitor)

A monitor is an object which is invoked by a process, at most one process at any given time is executing in the monitor.

In particular, a process

- enters the monitor by invoking one of the monitor's method
- exits the monitor by returning from the invoked method
- can put itself to sleep (stop executing) by waiting for an event or
- can signal an event, causing all processes waiting on that event to be eligible to run.

Monitor example

```
monitor ProducerConsumer {

  private:
    uint dequeueIndex=0,
         enqueueIndex=0;
    const int Size = 128;
    Elmt buffer[Size];

  public:
    Elmt dequeue();

    void enqueue(Elmt e);
}
```


Monitor example (cont'd)

```
Elmt dequeue()
  {  while (dequeueIndex == enqueueIndex)
         wait(notEmpty);
     Elmt e = buffer[dequeueIndex%Size];
     dequeueIndex++;
     signal(notFull);
     return e;
  }

void enqueue(Elmt e)
  {  while ((enqueueIndex - dequeueIndex) == Size)
         wait(notFull);
     buffer[enqueueIndex%Size] = e;
     signal(notEmpty);
     enqueueIndex++;
  }
}
```


Kernel as a monitor

Each method is a system call.

```
monitor kernel {
  int fork();
  void exit();
  int exec(name, arguments);
  ...
}
```


Monitor waits

- We can use many event names
  - Thus waking up fewer processes on average and
  - eliminate processes being woken and then put right back to sleep
  - because the event wasn't for that process
- A variant on monitor semantics is that signal only wakes up one process

Monitor properties

- Monitor is non-preemptive
- (In the example, ordering of signal and increment not material)
- Many processes may be waiting to run
- Monitor signals wake up all processes (test whether condition still holds when process wakes up)
- Note that wake up means “make eligible to run”. A process can only run inside the monitor when no other process is running.
- Monitors ensure atomicity between
  - The later of monitor invocation and wait
  - and the next wait or monitor exit

Preemption vs. non-preemption

- Preemptive program inherently harder to write since need to reason about underlying machine instructions (which are atomic) and possible interleavings.
- Preemptive programs need to eliminate all possible memory interleavings
- Failure to write preemptive programs properly results in obscure bugs
- Non-preemptive programs allows the programmer to directly specify atomic unit, but
- Interrupts are inherently preemptive, so monitors do not describe this
- So lets turn look at this next

Part III

Interrupts

Interrupts

- Interrupts can occur at any time
- Could be executing inside or outside the kernel
- Source could be a page fault, timer interrupt, or I/O
- If no process executing in the kernel, not really a problem
- If process is executing in the kernel, then there is a conflict

Page fault interrupt

- Page faults must by **precise interrupts**, meaning that the faulting location can be repaired and the program which was running can be resumed
- the Motorola 68000 didn't implement this properly, early demand paging 68000 systems required two 68000s (the second to service the page fault)
- If page fault occurs in the process, go into the kernel and fix it up and then resume process execution
- if page fault occurs while executing in the kernel, its more subtle

Page fault interrupt from within the kernel

What happens if the page fault occurs from within the kernel?

- It is a requirement to ensure that the code performing the demand paging does not get paged out
- Any address that can be paged out adds complexity since the page fault means that there is an implicit conditional wait at every point which accesses it.
- Ethos avoids this complexity by not having demand paging
- Opinion: demand paging, especially of kernel objects, is an anachronism
- It still has page faults, but these only occur from user space for page allocation or to fix up page table entries.

Timer interrupts

- Timer interrupts are important only from user space
- They prevent a process which is doing an extensive computation (and does not make system calls) from monopolizing the CPU
- (The kernel is coded so that it does not monopolize the CPU)
- So timer interrupts, when executed in the kernel, are ignored
- Just before returning from the kernel to user space, any time-based events which are older than the current time are processed

Device interrupts

- It is necessary to keep devices "moving along" so that keyboard and mouse input are not lost and that disk and network are kept busy.
- The key is to separate the interrupt processing from changes that the rest of the kernel can see.
- Usually possible to use a queue to communicate
- For example, an Ethernet input would be queued on a list of waiting packets
- Need to guard against race conditions when enqueuing and dequeuing (e.g., memory allocation for queue elements)
- Sufficient on a uniprocessor to block interrupts when doing these operations in other than the device driver.

Device alternatives

Defer interrupts that occur w/i the kernel to end of system call processing

Guard add synchronization on the interrupt handler and where the rest of the code updates the same data structure.

Fast/slow Interrupt handlers are divided into two parts, a **fast** or **hard** component when the interrupt occurs ans a **slow** or **soft** component where most of the processing is deferred.

Ethos uses fast/slow Interrupts. It should be possible in Ethos to preallocate storage so that fast interrupts do not require any locking.

Part IV

Reentrancy

Re-entrancy

Definition

A procedure is reentrant if it is safe to concurrently execute it.

- Note that if a procedure is re-entrant it must only call reentrant procedures
- A reentrant procedure needs to ensure atomic access to global writable variables
- It must not have any static local variables which is modifiable
- It must operate only on data supplied by the parameter
- Must not modify global copies of data

Reentrancy and monitors

- Monitors simplify reentrancy because they are non-preemptive
- Monitors concurrently execute code
- Monitors should be implemented with reentrancy
- Must not have logical updates which spans wait
- Note that since monitors are not preemptive, no synchronization is needed to ensure atomicity

Part V

Multi-core parallelism

Multicore

- Multicore provides multiple processors on a chip
- Problem: we have assumed only one processor in the kernel at a time
- With multicore need to ensure this property
- The easiest way is with the *Big Kernel Lock (BKL)* which uses a semaphore to ensure that only one processor in the kernel
- We can have many processors executing in user space

Big kernel lock limitations

- Big kernel lock is very simple, but runs into limitations due to Amdahl's law
- Amdahl's law states that if f is the fraction of code which is sequential then the maximum speedup is $1/f$.
- Kernels use be 10–90% of CPU cycles
- Hence, traditional kernels have been designed to enable different subsystems to be run in parallel with explicit locking.
- But that adds considerable complexity.
- Ethos right now uses only a single core (specified in Xen)
- Easy to add a BKL

Part VI

Latches and semaphores

Latches and Semaphores

- Latches are simple code to note when something is in use
- Use when you don't need to worry about race conditions (non-preemptive code)

```
while (latch != 0)
   wait(latchSignal); // in use, wait til free
latch++; // available, get it
```

- Semaphores are needed where there can be race conditions (e.g., multicore)
- Semaphores work correctly even if being executed by multiple cores at a time

Why are latches needed?

- System calls are atomic
- Thus when a system call waits there are two choices for work already done:
  - Re-do it when the syscall wakes up
  - Reserve it so that other syscalls do not interfere with it
  - Latches are used to reserver the resource

Part VII

Memory barriers

Memory Barriers

- Processors uses caches to contain the contents of frequently accessed memory addresses
- Memory read requests are made first to the cache, and if not found there, are made to memory
- Memory write requests are written to the cache
- Hence, memory may not contain the current value associated with that address
- On a uniprocessor system, this causes no problem as the structure of the memory system ensures that the value read for an address is the last value written

Memory Barriers (cont'd)

- On a multiprocessor, however, different processors will have different caches
- When a value can be updated by one processor and used by another processor, there can be erroneous results
- For example, with volatile memory (in the C sense)
- When it is necessary to order the memory operations a barrier is used
- write memory barrier orders only the writes
- full memory barrier forces all memory operations before the barrier to complete before any after the barrier.

Architectures

- Barriers are implemented by macros in C.
- What the macro does, including possibly being changed into a NO-OP, depends on the architecture
- strongly ordered architectures such as the x86, require the completion order of memory operations to be the same as the initiation order
- weakly ordered architectures does not require strong ordering

Barrier use example

- Consider the case of writing a message to be sent between OSs
- The writing of the message consists of two parts
  - writing the content of the message
  - writing a synchronization variable
    (In Xen's case, incrementing a counter)
- it is only after the counter is incremented that the contents will be read
- must ensure that the message is written before counter is changed
- a write memory barrier is used between the two steps
- note in some architectures its not necessary. Put it in anyway.

Part VIII

Summary

Summary

- In an OS, the kernel needs to deal with concurrency
- A monitor-based solution provides simple control over concurrency within the kernel
- The monitor does not handle interrupts
- Interrupts when a process is executing in user space can be treated as system calls
- Interrupts when a process is executing in kernel space perform minimal operations at the time of the interrupt and defers most processing when coming out of the kernel.

Summary (cont'd)

- Latches are used for resources which are held across waits
- Or for transactions (held across syscalls)
- When a syscall needs a latched object it must wait until that object is available
- When using latches, must ensure deadlock does not occur
- One way is by having an ordering on process and ensuring greatest one always completes
- Need also to deal with starvation