

Operating System Principles: Threads, IPC, and Synchronization

CS 111

Operating Systems

Peter Reiher

Outline

- Threads
- Interprocess communications
- Synchronization
 - Critical sections
 - Asynchronous event completions

Threads

- Why not just processes?
- What is a thread?
- How does the operating system deal with threads?

Why Not Just Processes?

- 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
 - Multiple activities working cooperatively for a single goal
 - Mutually trusting elements of a system

What Is a Thread?

- Strictly a unit of execution/scheduling
 - Each thread has its own stack, PC, registers
 - But other resources are shared with other threads
- 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 (with voluntary yielding)
 - Scheduled system threads (with preemption)

When Should You Use Processes?

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

When Should You Use Threads?

- For parallel activities in a single program
- When there is frequent creation and destruction
- When all can run with same privileges
- When they need to share resources
- When they exchange many messages/signals
- When you don't need to protect them 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 to serialize resource use
 - Your program will be more complex to write
- TANSTAAFL

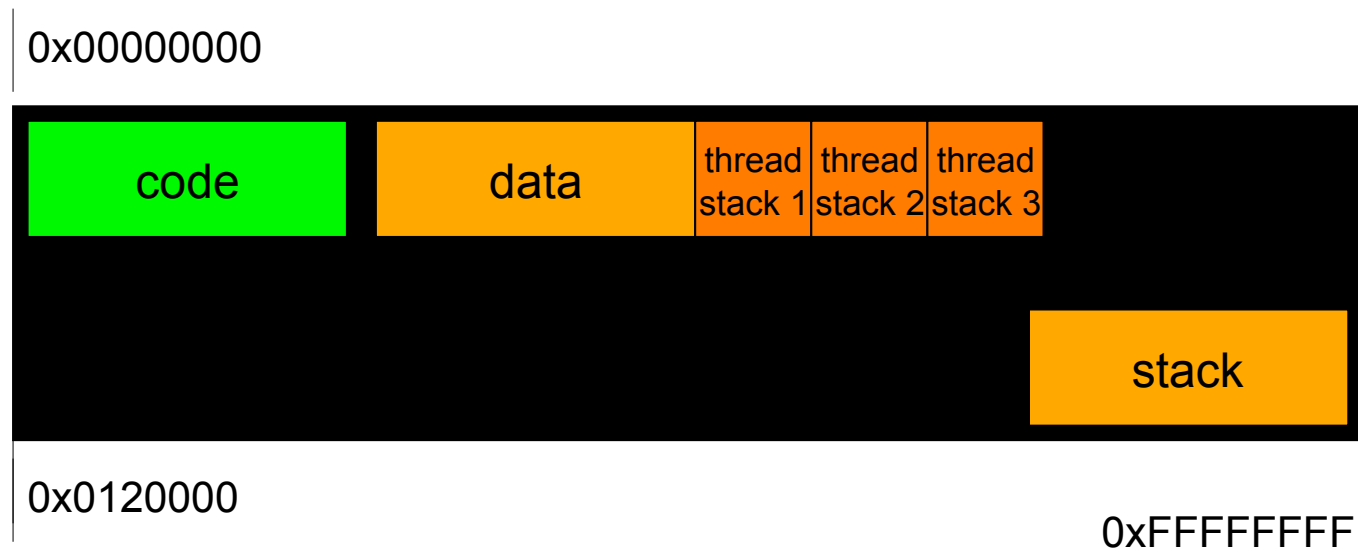
Thread State and Thread Stacks

- Each thread has its own registers, PS, PC
- Each thread must have its own stack area
- Maximum stack 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 Process Stack Space Management



Thread Stack Allocation



Inter-Process Communication

- Even fairly distinct processes may occasionally need to exchange information
- The OS provides mechanisms to facilitate that
 - As it must, since processes can't normally “touch” each other
- IPC

Goals for IPC Mechanisms

- We look for many things in an IPC mechanism
 - Simplicity
 - Convenience
 - Generality
 - Efficiency
 - Robustness and reliability
- Some of these are contradictory
 - Partially handled by providing multiple different IPC mechanisms

OS Support For IPC

- Provided through system calls
- Typically requiring activity from both communicating processes
 - Usually can't “force” another process to perform IPC
- Usually mediated at each step by the OS
 - To protect both processes
 - And ensure correct behavior

IPC: Synchronous and Asynchronous

- Synchronous IPC
 - Writes block until message sent/delivered/received
 - Reads block until a new message is available
 - Very easy for programmers
- 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
 - Much more efficient in some circumstances

Typical IPC Operations

- Create/destroy an IPC channel
- 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
 - Provide information like:
 - Who are end-points?
 - What is status of connections?

IPC: Messages vs. Streams

- A fundamental dichotomy in IPC mechanisms
- Streams
 - A continuous stream of bytes
 - Read or write a 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)
 - Each message is typically read/written as a unit
 - Delivery of a message is typically all-or-nothing
- Each style is suited for particular kinds of interactions

IPC and Flow Control

- Flow control: making sure a fast sender doesn't overwhelm a slow receiver
- 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 for feedback to sender

IPC Reliability and Robustness

- Within a single machine, OS won't accidentally “lose” IPC data
- Across a network, requests and responses can be lost
- Even on single machine, though, a sent message may not be processed
 - The receiver is invalid, dead, or not responding
- And how long must the OS be responsible for IPC data?

Reliability Options

- When do we tell the sender “OK”?
 - When it’s queued locally?
 - When it’s added to receivers input queue?
 - When the receiver has read it?
 - When the receiver has explicitly acknowledged it?
- How persistently does the system attempt delivery?
 - Especially across a network
 - Do we try retransmissions? How many?
 - Do we try different routes or alternate servers?
- Do channel/contents survive receiver restarts?
 - Can a new server instance pick up where the old left off?

Some Styles of IPC

- Pipelines
- Sockets
- Mailboxes and named pipes
- Shared memory

Pipelines

- Data flows through a series of programs
 - `ls | grep | sort | mail`
 - Macro processor | compiler | 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: next program failed
- *Simple, but very limiting*

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'll discuss these issues later
- *Very general, but more complex*

Mailboxes and Named Pipes

- A compromise between sockets and pipes
- 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 same system
- *Some advantages/disadvantages of other options*

Shared Memory

- OS arranges for processes to share read/write memory segments
 - Mapped into multiple process' address spaces
 - Applications must provide their own control of sharing
 - OS is not involved in data transfer
 - Just memory reads and writes via limited direct execution
 - So very fast
- Simple in some ways
 - Terribly complicated in others
 - The cooperating processes must achieve whatever effects they want
- Only works on a local machine

Synchronization

- Making things happen in the “right” order
- Easy if only one set of things is happening
- Easy if simultaneously occurring things don’t affect each other
- Hideously complicated otherwise
- Wouldn’t it be nice if we could avoid it?
- Well, we can’t
 - We must have parallelism

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

Why Is There a Problem?

- 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

Synchronization Problems

- Race conditions
- Non-deterministic execution

Race Conditions

- What happens depends on execution order of processes/threads running in parallel
 - Sometimes one way, sometimes another
 - These happen all the time, most don't matter
- But 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

- Parallel execution reduces predictability of process behavior
 - 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
- Which can lead to many problems

What Is “Synchronization”?

- True parallelism is imponderable
 - We’re not smart enough to understand it
 - Pseudo-parallelism may be good enough
 - Mostly ignore it
 - But identify and control 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

The Critical Section Problem

- A *critical section* is a resource that is shared by multiple threads
 - By multiple concurrent threads, processes or CPUs
 - By interrupted code and interrupt handler
- Use of the resource changes its state
 - Contents, properties, relation to other resources
- Correctness depends on execution order
 - When scheduler runs/preempts which threads
 - Relative timing of asynchronous/independent events

Reentrant & MultiThread-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 Example 1: Updating a File

Process 1

```
remove("database");  
fd = create("database");  
write(fd,newdata,length);  
close(fd);
```

```
remove("database");  
fd = create("database");  
  
write(fd,newdata,length);  
close(fd);
```

Process 2

```
fd = open("database",READ);  
count = read(fd,buffer,length);
```

```
fd = open("database",READ);  
count = read(fd,buffer,length);
```

- Process 2 reads an empty database
 - This result could not occur with any sequential execution

Critical Section Example 2: Re-entrant Signals

First signal

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

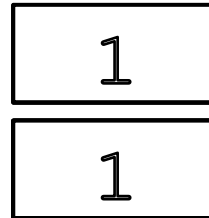
```
load r1,numsigs // = 0  
add r1,=1 // = 1
```

```
store r1,numsigs // =1
```

So numsigs is 1,
instead of 2

numsigs

r1



Second signal

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

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

The signal handlers share
numsigs and r1 ...

Critical Section Example 3:

Multithreaded Banking Code

Thread 1

```
load r1, balance // = 100
load r2, amount1 // = 50
add r1, r2        // = 150
store r1, balance // = 150
```

```
load r1, balance
```

```
load r2, amount1
```

```
add r1, r2
```

CONTEXT SWITCH!!!

```
store r1, balance // = 150
```

Thread 2

```
load r1, balance // = 100
load r2, amount2 // = 25
sub r1, r2        // = 75
store r1, balance // = 75
```

```
load r1, balance // = 100
load r2, amount2 // = 25
sub r1, r2        // = 75
store r1, balance // = 75
```

The \$25 debit was lost!!!

amount1

50

balance

150

amount2

25

r1

75

r2

50

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
```

Three instructions . . .

Why Is This a Critical Section?

thread #1

counter = counter + 1;

thread #2

counter = counter + 1;

This could happen:

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

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

```
mov %eax, counter
```

If counter started at 1, it should end at 3

In this execution, it ends at 2

These Kinds of Interleavings Seem Pretty Unlikely

- To cause problems, things have to happen exactly wrong
- Indeed, that's true
- But you're executing a billion instructions per second
- So even very low probability events can happen with frightening frequency
- Often, one problem blows up everything that follows

Critical Sections and Mutual Exclusion

- Critical sections can cause trouble when more than one thread executes them at a time
 - Each thread doing part of the critical section before any of them do all of it
- Preventable if we ensure that only one thread can execute a critical section at a time
- We need to achieve *mutual exclusion* of the critical section
- How?

One Solution: Interrupt Disables

- Temporarily block some or all interrupts
 - Can be done with a privileged instruction
 - Side-effect of loading new Processor Status Word
- Abilities
 - Prevent Time-Slice End (timer interrupts)
 - Prevent re-entry of device driver code
- Dangers
 - May delay important operations
 - A bug may leave them permanently disabled

What Happens During an Interrupt?

- What we discussed before
- The hardware traps to stop whatever is executing
- A trap table is consulted
- An Interrupt Service Routine (ISR) is consulted
- The ISR handles the interrupt and restores the CPU to its earlier state
 - Generally, interrupted code continues

Preventing Preemption

```
DLL_insert(DLL *head, DLL*element) {  
    int save = disableInterrupts();  
    DLL *last = head->prev;  
    element->prev = last;  
    element->next = head;  
    last->next = element;  
    head->prev = element;  
}
```

```
    restoreInterrupts(save);
```

```
DLL_insert(DLL *head, DLL*element) {  
    DLL *last = head->prev;  
    element->prev = last;  
    element->next = head;  
    last->next = element;  
    head->prev = element;  
    DLL_insert(DLL *head, DLL*element) {  
        head->prev = element;  
        DLL *last = head->prev;  
    }  
    element->prev = last;  
    element->next = head;  
    last->next = element;  
    head->prev = element;  
}
```

Downsides of Disabling Interrupts

- Not an option in user mode
 - Requires use of privileged instructions
- Dangerous if improperly used
 - Could disable preemptive scheduling, disk I/O, etc.
- Delays system response to important interrupts
 - Received data isn't processed until interrupt serviced
 - Device will sit idle until next operation is initiated
- May prevent safe concurrency

Interrupts and Resource Allocation

- Interrupt handlers are not allowed to block
 - Only a scheduled process/thread can block
 - Interrupts are disabled until call completes
- Ideally they should never need to wait
 - Needed resources are already allocated
 - Operations implemented with lock-free code
- Brief spins may be acceptable
 - Wait for hardware to acknowledge a command
 - Wait for a co-processor to release a lock

Interrupts – When To Disable Them

- In situations that involve shared resources
 - Used by both synchronous and interrupt code
 - Hardware registers (e.g., in a device or clock)
 - Communications queues and data structures
- That also involve non-atomic updates
 - Operations that require multiple instructions
 - Where pre-emption in mid-operation could lead to data corruption or a deadlock.
- Must disable interrupts in these critical sections
 - Disable them as seldom and as briefly as possible

Be Careful With Interrupts

- Be very sparing in your use of disables
 - Interrupt service time is very costly
 - Scheduled processes have been preempted
 - Devices may be idle, awaiting new instructions
 - The system will be less responsive
 - Disable as few interrupts as possible
 - Disable them as briefly as possible
- Interrupt routines cannot block or yield the CPU
 - They are not a scheduled thread that can block/run
 - Cannot do resource allocations that might block
 - Cannot do synchronization operations that might block

Evaluating Interrupt Disables

- Effectiveness/Correctness
 - Ineffective against multiprocessor/device parallelism
 - Only usable by kernel mode code
- Progress
 - Deadlock risk (if handler can block for resources)
- Fairness
 - Pretty good (assuming disables are brief)
- Performance
 - One instruction, much cheaper than system call
 - Long disables may impact system performance

Other Possible Solutions

- Avoid shared data whenever possible
- Eliminate critical sections with atomic instructions
 - Atomic (uninterruptable) read/modify/write operations
 - Can be applied to 1-8 contiguous bytes
 - Simple: increment/decrement, and/or/xor
 - Complex: test-and-set, exchange, compare-and-swap
- Use atomic instructions to implement locks
 - Use the lock operations to protect critical sections
- We'll cover these in more detail in the next class