

# Dependable Kernel Architectures

Advanced Operating Systems (M)  
Lecture 13

# Lecture Outline

- The need for dependable kernels
- Kernel implementation languages
  - Benefits of moving away from C
- Microkernels and strongly isolated systems
  - Benefits of software isolated processes
  - Microsoft's Singularity as an example

# How to make the kernel dependable?

- Move away from C as an implementation language
  - Lack of type- and memory-safety leads to numerous bugs and security vulnerabilities
  - Limited support for concurrency – race conditions, locking problems – makes it unsuitable for modern machine architectures
- Move towards architectures with a minimal kernel, and strong isolation between other components of the operating system
  - The monolithic part of a kernel is a single failure domain; this needs to be reduced to a minimum → microkernel architecture
  - Easier to debug and manage components when they're isolated from each other, and communicate only through well-defined channels

# Kernel Implementation Languages

- Desirable to implement kernel in a safe language
  - The language should have a rigorously-defined strong type system, with clearly specified semantics
  - This does *not* prevent compilation to native code, if desired
  - This does *not* require a static type system, although one may be desirable to help find bugs early
- Desirable to support concurrency, since multicore processors are ubiquitous
  - The memory model needs to be formally defined, at least, as do any synchronisation primitives – when do memory operations made by a processor become visible to other processors?
  - The combination of the language and its standard library might provide higher-level communication mechanisms than traditional locking

# Memory Models

- Many multiprocessor systems use memory that is shared between processors
  - The system may have symmetric or non-uniform memory access (NUMA)
  - There may be multiple layers of caching between processors and memory
- When do a memory writes made by one processor become visible to other processors?
  - Prohibitively expensive for all threads on all processors to have the exact same view of memory (“sequential consistency”)
  - For performance, allow processors to have inconsistent views of memory, except at synchronisation points; introduce synchronisation primitives with well-defined semantics
  - Varies between different processors – even between variants of the same processor architecture – differences can generally be hidden by language runtime, if there is a language-specific memory model

# Example: The Java Memory Model

- Java has a formally defined memory model
- Between threads:
  - [Somewhat simplified: see the Java Language Specification, Chapter 17, for full details <http://java.sun.com/docs/books/jls/>]
  - Changes to a field made by one thread are visible to other threads if:
    - The writing thread has released a synchronisation lock, and that same lock has subsequently been acquired by the reading thread (writes with lock held are atomic to other locked code)
    - If a thread writes to a field declared `volatile`, *that* write is done atomically, and immediately becomes visible to other threads
    - A newly created thread sees the state of the system as if it had just acquired a synchronisation lock that had just been released by the creating thread
    - When a thread terminates, its writes complete and become visible to other threads
  - Access to fields is atomic
    - i.e., you can never observe a half-way completed write, even if incorrectly synchronised
    - Except for `long` and `double` fields, for which writes are only atomic if the field is `volatile`, or if a synchronisation lock is held
- Within a thread: actions are seen in program order

# Memory Models

- Defines the space in which the language runtime and processor architecture can innovate, without breaking programs
  - Synchronisation between threads occurs only at well-defined instants; memory may appear inconsistent between these times, if that helps the processor and/or runtime system performance
- Java is unusual in having such a clearly-specified memory model
  - Other languages are less well specified, running the risk that new processor designs can subtly break previously working programs
  - C and C++, in particular, have *very* poorly specified memory models

# Safe Languages

- “A safe language is one that protects its own abstractions”  
[B. Pierce, Types and Programming Languages, MIT Press, 2002]
- Undefined behaviour – as in the example on the right – is prohibited to the extent possible
  - The language specification can require that array bounds are respected, and specify the error response to violation
- Requires both compile- and run-time checking
  - The *type system* specifies legal properties of the program “for proving the absence of certain program behaviours”
  - Some properties can be statically checked by a compiler: a faulty program will not compile until the bug is fixed
  - Some properties require run-time checks: failure causes a controlled error
  - This does not guarantee that a program will work correctly, but helps ensure that it fails in a predictable and consistent way

```
-->cat tst.c
#include <stdio.h>

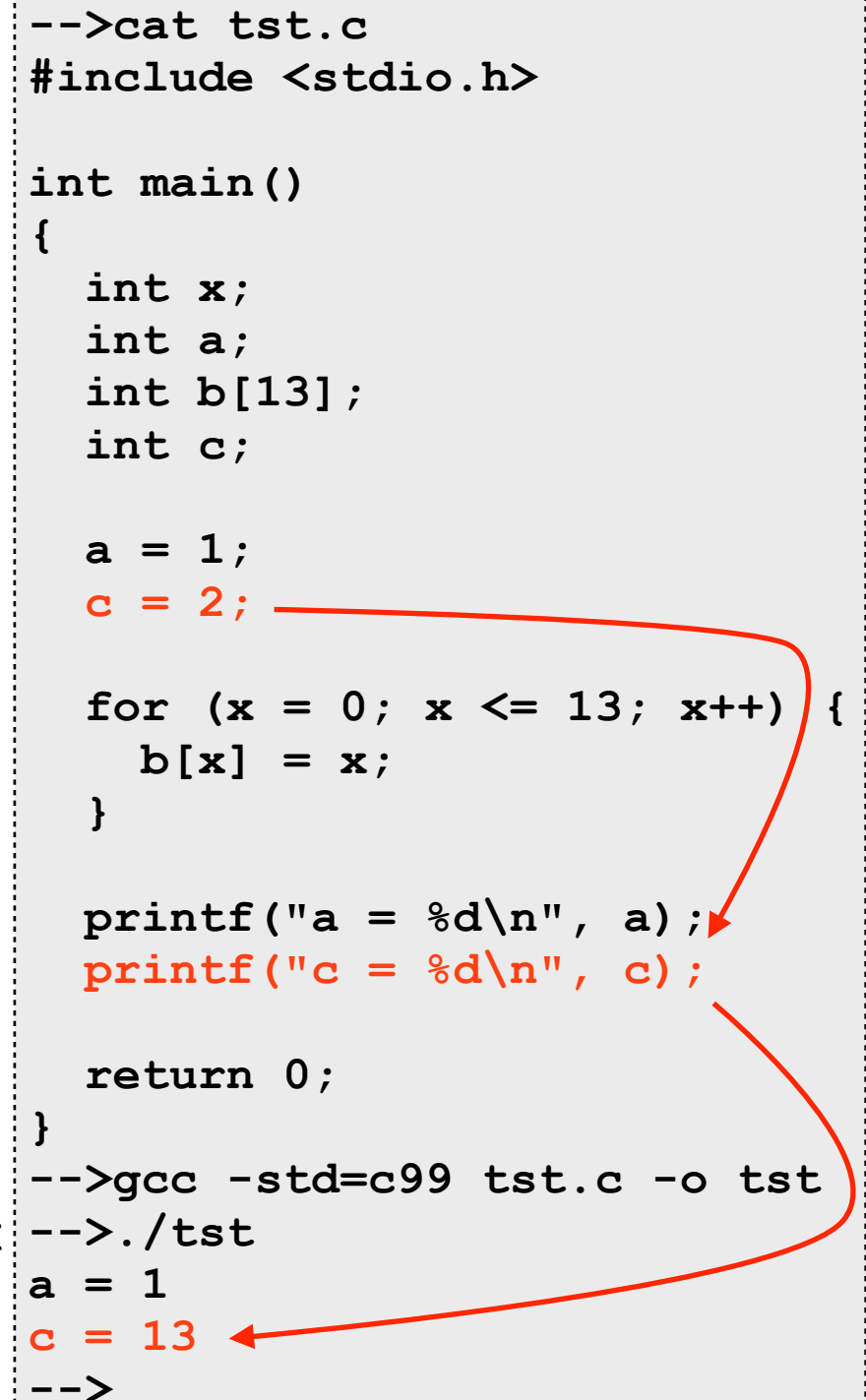
int main()
{
    int x;
    int a;
    int b[13];
    int c;

    a = 1;
    c = 2;

    for (x = 0; x <= 13; x++) {
        b[x] = x;
    }

    printf("a = %d\n", a);
    printf("c = %d\n", c);

    return 0;
}
-->gcc -std=c99 tst.c -o tst
-->./tst
a = 1
c = 13
-->
```





# Example: Banishing the Null Pointer

- Many languages allow references to nothing (i.e., the `NULL` pointer)
  - Often see APIs written to return a pointer to an object, or `NULL` if the object doesn't exist
  - A common failure is to forget to check for `NULL` before using the returned object
  - Causes runtime `NullPointerException` in Java, or (probably) a crash in C
- Can require references to be valid, use a different type to signal invalid values
  - Haskell has the `Maybe` type; Scala uses an abstract class `Option[X]` with subclasses `Some[X]` and `None`
  - Since `Option[X]` is abstract, must match on its subclasses to access result – language requires an exhaustive match, so the code won't compile if the `None` check is missing.
  - Turns a runtime failure into a compile time check

```
char *getParameter(request_t *data, char *param) {  
    ...  
}  
  
char *name = getParameter(request, "name")  
if (name != NULL) {  
    printf("%s", toupper(name));  
} else {  
    printf("No name value");  
}
```

C

```
class Request {  
    def getParameter(param : String):Option[String]  
    ...  
}  
  
val name = request.getParameter("name")  
name match {  
    case Some(name) =>  
        println(name.toUpperCase)  
    case None =>  
        println("No name value")  
}
```

Scala

# Example: Pattern Matching & Messages

- Many languages support *pattern matching* on the runtime type of a variable
  - Syntax like a C `switch` statement, but can match on the *type* of its argument, not just its value
  - Semantics like a nested sequence of `instanceof` tests in Java, but generally requires all possible subtypes to be considered
- Useful for implementing message passing
  - E.g., use a `ConcurrentLinkedQueue` to pass data between threads, where each data item is a subclass of some abstract class `Message`
  - Receiver uses pattern matching on `Message` subtypes to call handlers; compiler will enforce that all possible message subtypes are handled
    - The same design can be implemented without compiler support, but loses the guarantee that all subtypes of `Message` are handled, since its no longer checked by the type system

# Example: Immutable Data

- Imperative programming language generally make use of mutable data structures
- Functional languages prefer *immutable* data
  - Once an object is created, it cannot be modified
  - Data structures are updated by making a copy of the data with the change applied
    - e.g., to add an item to a list, you don't modify the list, you instead make a new list with the item to be inserted at the head
    - Programs are written as a sequence of functions that transform data, each returning a modified copy of the data item with some transformation applied
  - Straight-forward to enforce immutability at language level (c.f., Haskell)
  - Desirable when implementing concurrent systems, since immutable data doesn't need to be locked when accessed by multiple threads

[C. Okasaki, Purely Functional Data Structures, PhD thesis, CMU, 1996. <http://www.cs.cmu.edu/~rwh/theses/okasaki.pdf>]

# Example: Linear Types

- A variable with *linear* type may be used only once; it goes out of scope after use
- Potentially useful when sharing mutable data between threads
  - Implement sharing via a `sendMessage` function that takes a linear type for the data to be shared
  - Message data consumed by the `sendMessage` function and the receiver, and so can't be used by the sender once the message has been sent
  - Data doesn't need to be locked, since it can only be used by one thread at once
- The compiler enforces that linear data is not shared between threads
  - Disadvantage: requires an unusual programming style

```
linear int x = 5;  
int y = x;  
int z = x + 1; // error
```

```
linear int x = 5;  
linear int y = foo(x);  
sendMessage(dest, y);  
int z = y + 1 // error
```

# Discussion

- Language-level features that have the potential to make systems programming much easier
  - Complexity pushed from the systems programmer to the compiler writer
  - Generally require little in the way of runtime support, and so can be used within a kernel
- Nothing here is unknown: most ideas are available today in production-quality languages such as Haskell, Scala, Erlang

# Microkernels & Strongly Isolated Systems

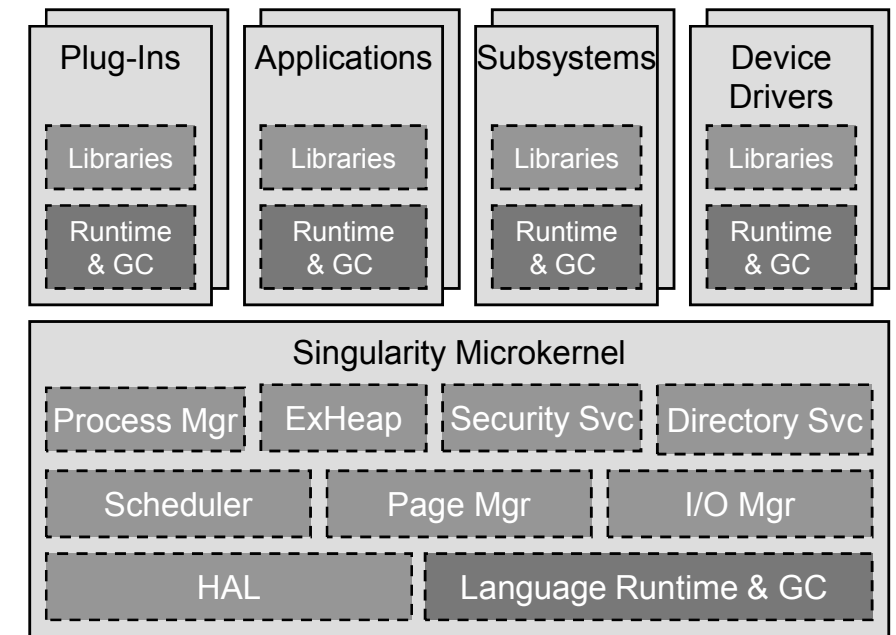
- Desirable to separate components of a system, so failure of a component doesn't cause failure of the entire system
- Traditional approach: microkernel
  - Strip-down the monolithic part of the kernel to only the most essential services; run everything else in user space
  - Device drivers/services run as separate user processes, communicate using some message passing API
  - Kernel just managing process scheduling, isolation, and message passing
  - Widely used in embedded systems, where robustness and flexibility to run devices for unusual hardware are essential features
  - But: difficult to make efficient, due to the need to manage page tables and memory protection settings on each context switch, coupled with frequent context switches

# High-level Kernels and Software Isolation

- A possible solution:
  - Microkernel system, that enforces all user-space code is written in a safe language (e.g., by only executing byte code, no native code)
    - This includes device drivers and system services running outside the microkernel
  - The type system prevents malicious code obtaining extra permissions by manipulating memory it doesn't own
  - Permissions enforcement can therefore be done entirely in software – no need to use the MMU to enforce process separation in hardware
  - *A software isolated process architecture*
- Example: Microsoft's Singularity operating system

# Singularity Architecture

- Microkernel written in C++
- All other code written in Sing#, an extension to C# that provides for strict specification of inter-process communication channels
  - Discussed in lecture 12, in the context of device drivers
- All non-kernel code runs as *sealed* software isolated processes
  - This includes device drivers, and subsystems such as the TCP/IP stack
  - No shared memory, no dynamic libraries, plugins, or other forms of run-time code loading/extension – such features implemented by starting new software isolated processes, with message passing communication
- Strong isolation make system more robust, easier to develop and test

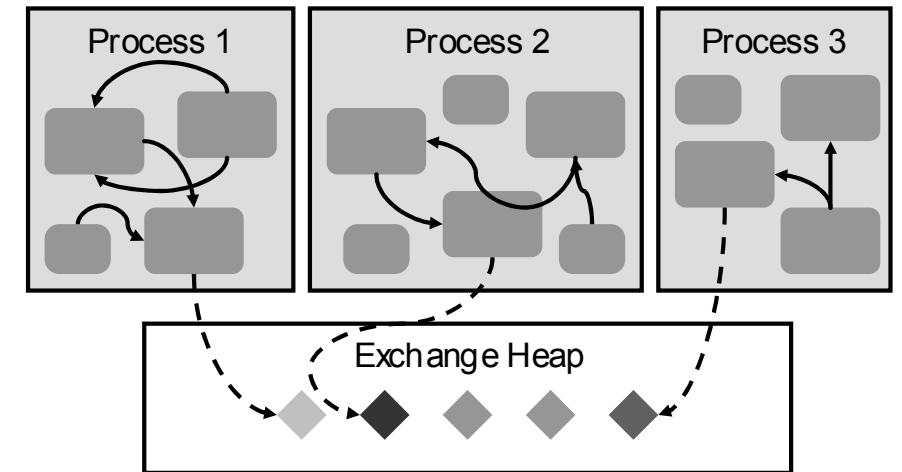


[G. Hunt *et al.*, Sealing OS processes to improve dependability and safety. In Proc. EuroSys 2007, Lisbon, Portugal. DOI 10.1145/1272996.1273032]



# Software Isolated Processes

- A software isolated process comprises a set of memory pages, threads of execution, and channel endpoints
  - Each process has its own garbage collector to manage its storage
- Communication is via typed channels; data is passed using a separate exchange heap
  - The type system enforces that messages contain primitive types only – no pointers or object references
  - Linear types are used to ensure that the sender does not retain a reference to a message after it has been sent
  - The message data does not need to be copied: a reference to it is passed via the channel
  - Receiver uses pattern matching to process messages; the channel contract specifies the legal message types
- Message passing is low overhead, and safe; the system supports large number of software isolated processes



[G. Hunt *et al.*, Sealing OS processes to improve dependability and safety. In Proc. EuroSys 2007, Lisbon, Portugal. DOI 10.1145/1272996.1273032]

# Discussion

- The microkernel, JIT, and garbage collector are written in unsafe C++
  - Much of this might be possible to eventually migrate to a safe language
  - Is it possible to implement the entire system in a safe language?
- Relies on correctness of type system and runtime to ensure isolation
  - Bugs cannot be caught by hardware process isolation, since it's not used
  - There may be an argument for using memory protection as “defence in depth”, to protect against software failures
- JIT compilation of safe byte code introduces some overhead, but system calls and context switches are much faster – performance is acceptable

# Summary

- Widely used operating systems implemented in C, using a monolithic kernel architecture unchanged for decades
- Modern programming languages provide features that can ease implementation
- These can improve the efficiency of microkernels, by introducing software isolated processes

# Further Reading

- G. Hunt *et al.*, Sealing OS processes to improve dependability and safety. In Proceedings of the European Conference on Computer Systems, Lisbon, Portugal, March 2007. ACM/EuroSys. DOI: 10.1145/1272998.1273032

