1. Project questions
2. Questions from reviews:
    a. What about dynamic code generation?
        i. Can only do whole programs – no incremental specialization
        ii. Can incorporate trusted compiler into system that ensures rules are met
    b. Could malicious manifest corrupt system?
        i. Can check if still meets properties; similarly user could write malicious code directly
    c. Where did this work go?
        i. Others wrote fully verified kernels (Australia group)
        ii. People look at verifying pieces of OS – e.g. file system invariants, other things
        iii. People look at formally specifying device drivers – Termite & Dingo
    d. Why not in mainstream OS?
        i. Microsoft tried in Windows Vista
            1. Change to new language hard
            2. GC difficult to integrate with regular memory management, semantics under out-of-memory hard
3. Singularity Origins
    a. What is the goal of this work?
        i. Reinvestigate every design decision about operating systems
        ii. Build a platform for exploring design options
        iii. Build on Microsoft's experience with Windows – what are the real pressure points
    b. What are the big problems with operating systems that need to be addressed?
        i. Performance – lots of attention!
        ii. Complexity
        iii. Manageability
        iv. Reliability
        v. Security
    c. What are the problems that Singularity seeks to address?
        i. Complexity
        ii. Manageability
        iii. Security
        iv. Reliability
        v. **QUESTION: Why?**
            1. Observed as the biggest problems in Windows – not so much performance
    d. What is the motivating/enabling technology?
        i. Safe programming languages – Java

ii. Program verification tools – check various properties

4. General idea:
    a. Write code in a language that is checkable
    b. Write properties that can be checked
    c.
5. What are core philosophies/features of this work?
    a. Do as much as possible as early as possible, and only the minimum at runtime.
    b. Example:
        i. Make sure that code can't touch illegal memory → type safe programs
        ii. Make sure that programs follow protocols →channel contracts
        iii. Declare application dependencies with manifests
        iv. Declare driver dependencies on hardware resources statically in code
    c. Make the system as static as possible
        i. Jit code at install time
        ii. No dynamic loading of code / dynamic code generation
6. What do they give up?
    a. Compatibility with Unix/Linux
    b. Only run MSIL code or scripting languages
        i. QUESTION: Does this mean single-language?
        ii. ANSWER: No, but tools must emit a static MSIL program to be run
7. Properties
    a. Microkernel design:
        i. Small kernel, anything independent runs in a separate process
            1. Network stack components (tcp/ip, NIC drivers)
            2. Storage stack (fs, disk drivers)
            3. Kernel has minimal stuff:
                a. Communication
                b. Memory management (parts of it)
                c. Processes/scheduling
    b. Single unified names space
        i. Linux Unix, but includes networking:
            1. /tcp/192.168.0.1/80
            2. /fs/usr/ssc/…
    c. Software isolated processes
        i. Regular process first:
            1. Virtual address space – totally controlled in user mode
                a. VM hardware translates addresses or invalidates addresses
            2. Safe control transfer to kernel – trap to known address, index of known function to call
            3. Unit of recovery – can release everything it is using on exit
            4. Unit of isolation – cannot modify data outside it
            5. QUESTION: what are costs?
                a. Page tables

        b. TLB flushes, context switches
        c. TLB misses
    ii. Big idea: software provides protection
        1. E.g. Private variables , array bounds in Java
        2. Enforced by compiler: won't produce code that violates these rules
        3. Question: if have language enforcement, why need hardware (e.g. user/kernel mode, virtual memory?
        4.
    iii. Sealed at load time – no memory sharing or dynamic code loading
        1. Loading extensions a major cause of unreliability
        2. No ability to identify applications statically, as behavior may change. E.g. want to enforce that only your bib database can access bib file. Must use user identity instead
            a. Example: whole program optimization, dead code elimination remove 66% of code in their tests
        3. Extensions often circumvent language/interface to access & modify host application data
        4. **QUESTION: Why important?**
            a. Suppose you trust a process with your credit card number
            b. Then it loads an extension that can modify the code or read all the data
            c. It can steal your credit card
            d. **RESULT:** if seal the process, can make **guarantees** about what the code can do that cannot be violated by future actions, as the code cannot change
    iv. Just-in-time compilation from MSIL
        1. Can verify MSIL is correct, JIT (Bartok compiler) is trusted to produce good code
    v. Communication explicit via channels controlled by kernel
        1. Kernel can identify set of processes/services a process may contact as transitive closure of channels used.
    vi. Static type safety by compiler ensures code cannot access illegal memory
        1. So can be statically checked by compiler
        2. Can prove that a SIP cannot access data or invoke code outside the SIP even without hardware protection
        3. **What if not sealed?**
            a. Checkers guarantees no longer hold
        4. Makes compiling a bit slower, but Moore's law makes compiling faster
            a. Better to have slow compiling or fast execution?
            b. Optimizing C code is hard – imprecise – compared to type-safe code
    vii. Kernel API has no functions for manipulating other running processes

1. E.g. read memory, trace system calls, etc.
2. Only management functions:
    a. Create child process – finishes before code executes
    b. Stop child process

viii. Example:
1. Web server, file system, device driver, network driver, tcp/ip stack all in separate
2. Example use: web browser
    a. Chrome launches a separate full process for extensions, renderers for same web site – could use SIPs
3. Picture: show graph of processes communicating

ix. Benefits:
1. No HW for context switching, system calls → much faster, can do it more often
2. Can execute privileged code in line by running all code in privileged mode
    a. SW prevents accidental/malicious use of this

x. Drawback: no dynamic code generation
1. E.g. jitting for Java, Perl

xi. QUESTION: what problem does this solve?
1. Reliability: can reason about correctness of code in a process because you know it all

xii. QUESTION: no mention of virtual memory. Why not?
1. How do you do it?
    a. Garbage collect data to fewer pages, swap other data out.

xiii. OS design: all services are a SIP
1. File system, network, drivers
2. Kernel handles processes, low-level memory

d. Communication over channels
i. Not pipes, RPC, shared memory
ii. Bi-directional communication channel
1. Channel endpoints can be passed over channels
2. NOTE: like capabilities
3. Use: open a connection to a server, server can return an endpoint to you for further communication (a bit like FTP)

iii. Send messages never block
1. Question: why? A: no buffering – handled by client code using exchange heap
2. Never fail: why? Easier to write code that way
    a. Failure only on receive, with “channel closed” message

iv. Receive messages block
1. How do you handle large numbers of channels?
2. Like case statement:

a. Switch receive {
   case NicClient.PacketForReceive():
   case NicClient.GetReceivedPacket():
   unsatisfiable:

3. Unsatisfiable used if no case could be satisfied by future messages, otherwise blocks

v. Statically defined set of messages + state machine for protocol
vi. Example: Device Driver, n (! for Exp to Imp), and (? for Imp to Exp).
vii.

```
state START: one {
   DeviceInfo! → IO_CONFIGURE_BEGIN;
}
state IO_CONFIGURE_BEGIN: one {
   RegisterForEvents? →
   SetParameters? → IO_CONFIGURE_ACK;
}
state IO_CONFIGURE_ACK: one {
   InvalidParameters! → IO_CONFIGURE_BEGIN;
   Success! → IO_CONFIGURED;
}
state IO_CONFIGURED: one {
   StartIO? → IO_RUNNING;
   ConfigureIO? → IO_CONFIGURE_BEGIN;
}
state IO_RUNNING: one {
   PacketForReceive? → (Success! or BadPacketSize!)
   → IO_RUNNING;
   GetReceivedPacket? → (ReceivedPacket! or NoPacket!)
   → IO_RUNNING;
...
}
```

1. Note: “one” means only one of the message can arrive in this stae, “all” would mean arbitrary interleaving of all the message sequences (not useful)

viii. A verifier can make sure statically that a program doesn’t send a message when the channel is in the wrong state → avoids error checking code at runtime.
ix. Can verify that finite buffering is needed:
    1. Can’t send lots of messages without an ACK; each cycle in state machine must have one send and one receive.
    2. Can tell how big cycle is; how much buffering is needed for a channel statically
x. QUESTION: what problem does this solve?
    1. Reliability, security, complexity
    2. Improves performance
xi. **Implementation:**
    1. Channels are kernel objects, use handles to manipulate and pass around
    2. Guarantee finite buffering space needed – no sender sends indefinitely without receiver receiving a message
        a. In any cycle of a state graph (e.g. ready -> sent -> ready), have to send and receive one message

        b. Prevents unlimited sends without waiting for receiver
        c. Example: Flow-control window
            i. Can only send window data before waiting for an ACK, must have finite maximum window
    3. Given known size, can pre-allocate queues
e. Exchange heap for communication
    i. A SIP can have a single pointer into an object in the exchange heap
        1. Linear Type System: prevents aliasing; references go out of scope when appearing on the right side of an assignment or passed as a parameter
    ii. Communicate by passing that pointer to another SIP
        1. Invalidates local pointer, so can't be used
        2. Provides copy semantics without expense of copying
    iii. Example: file system read
        1. Caller can provide buffer in exchange heap for read
        2. FS can fill it in, knowing that SIP cannot read it
        3. FS can return it when done
f. Manifest-based programs (applications), not executable files
    i. Manifest – file describing:
        1. Code resources (where code is – inline or in a file)
            a. Can point to shared code files with other programs
        2. Required system resources (what services it depends on e.g. gui)
        3. Required capabilities/permissions (e.g. must send to internet)
        4. Dependencies on other programs (e.g. must have sql server installed)
    ii. Created by programmer –
    iii. Why
        1. Lots of metadata about programs – help systems decide what programmer's intent is, even if not directly in the code
        2. Like a manifest for a mobile app or a docker
            a. Resources, permissions, configuration information
        3. Comparison to Linux: windows: can run just an executable
    iv. All code is MSIL – like java bytecode, but more languages
    v. Used to :
        1. Discover configuration settings that affect a program, what values those settings must have
        2. Tell system what code to put in a SIP for execution
        3. Tell system what channels to connect
        4. Tell system what system resources to give to SIP
    vi. Benefits:
        1. Can verify at install time whether requirements have been met
        2. Can verify that programs won't do certain things – can't access channels if not declared in manifest

8. Trust?

a. In Singularity, what do you trust for protection?
    i. Sing# -> MSIL Compiler
    ii. MSIL -> x8 ASM
b. In Unix, what do you trest?
    i. Kernel / OS c compiler (see Reflections on Trusting Trust)
    ii. HW to properly implement protection
        1. Not always done – ibm 360 has a bug where didn't work

9. Kernel design
    a. Language
        i. All written in a type-safe language
            1. Some challenge in accessing HW features
            2. Have unsafe variant of sing#, or use asm C++
            3. What is hard?
                a. I/O
                b. Debugging – access to registers
                c. Garbage collection – need to violate memory safety to copy objects
    b. Interface
        i. Small, generic messaging interface – like a microkernel
        ii. Versioned – can upgrade and provide backwards compatibility (SW evolution!)
        iii. System code can be run inside user processes – trusted functions
            1. No need to switch to privileged mode, as SW prevents calling instructions directly
            2. Note: same technique used in VMware for hypervisor code (later in the semester)
    c. Handles
        i. Generic OS design: opaque references to OS objects
            1. Table of handles; external code has handle, table points to the kernel object
            2. Basically, file descriptor, but typed (cannot just use an integer, as in C/Unix)
                a. Unix/Windows: handle table per process
                    i. Easy to know what to reclaim
                    ii. Easy to stop use of another process's handles
                b. Singularity: single global handle table
                    i. Typing prevents "creating" or "forging" a handle or re-using after release
                    ii. Only re-use with a SIP – e.g. assign some set of entries to a SIP
    d. Memory management
        i. Standard kernel:
            1. Physical page allocator
            2. Heap allocator for kernel structures

3. Reclaim like in C:
    a. Malloc/free
    b. Reclaim all process memory on exit(),
4. Shared memory possible

ii. Singularity:
1. Garbage collected memory
    a. No need to free()
    b. Can relocate things in memory with garbage collection
        i. Don't need page-based allocation
        ii. Can allocate large contiguous chunks with virtual addressing
    c. Allocate pages and use pieces for small allocations, let GC clean up and coalesce
    d. **COMPARE TO NORMAL SYSTEM**
2. Exchange heap
    a. **Question What for?**
        i. **Problem**: how communicate data objects without copying and without sharing?
            1. **SIP cannot have pointer to object in another SIP**
    b. General-purpose system-wide area for sharing
        i. Objects in the exchange heap only refer to objects in the exchange heap (never a SIP)
        ii. Only one SIP can have 1 reference to an object in exchange heap
            1. System prevents two SIPs from simultaneous access
        iii. Transfer objects by atomically giving up a reference, giving reference to another SIP via message
        iv. Can GC when process terminates, no need for locks (only 1 pointer -> 1 thread can access)
    c. Use with channels:
        i. Put pointer in message; atomically remove local pointer
        ii. Receiver can refer to passed data via pointer provided, now owns all of the object passed (like copying)

iii. GC
1. Each SIP can have its own GC, but not need own address space.
    a. **WHY?**
        i. No cross-sip structure; GC only need at one SIP
    b.

e. Threads

i. Stacks not need to be contiguous
    1. Managed language (Sync#) puts in code to test for stack growth, allocate new page and link in
ii. Scheduling
    1. **What are goals?**
        a. Expect lots of communication: split processes into SIPs with messages
            i. Many procedure calls become messages
        b. Want to run a process that wakes quickly, return quickly
        c. Donate rest of timeslice to a process that is awoken (common technique)
    2. Two lists:
        a. Long-running (preempted)
        b. Interactive (unblocked)
        c. Always run interactive (just unblocked first), and if none, run preempted
        d. At end of time slice, move all unblocked to preempted list
            i. Idea: Try to run them in same slice, but if can't, just go in normal order

10. Design choices
    a. Reflection
        i. Want to dynamically learn properties, have new behavior – e.g. load a class, figure out what methods it has, call them
        ii. Answer: compile-time reflection
            1. Figure out what use case is: e.g., templating code with config parameters, or arguments to a program
            2. Build template language to provide parameters (declaration), and a transform that says how to generate code from the declaration (CTR transform)
            3. Spits out Sing# code
    b. Hardware vs software isolation
        i. **QUESTION:** why need virtual memory and privilege levels?
            1. Solves contiguous memory allocation (move pages around), fragmentation
            2. Allows swapping to disk
            3. Address spaces isolate processes
            4. Privilege level prevents executing privileged instructions, modifying other processes
        ii. Is this necessary?
            1. GC: memory allocation
            2. Language protection: isolation, no privileged ops
            3. Swapping: buy more memory?
        iii. Singularity: can optionally use HW address spaces or not between SIPs, measure the cost

c. Heterogeneous multiprocessing
    i. Can have mix of CPUs in a single system – ARM, x86, multiple kinds of each
    ii. Normal OS: hard to migrate threads
    iii. Singularity: compiles for different architecture, communicate over channels
d. Typed assembly