LECTURE- 31 UNIX EMULATION IN MACH

Reconsidering the Kernel Interface

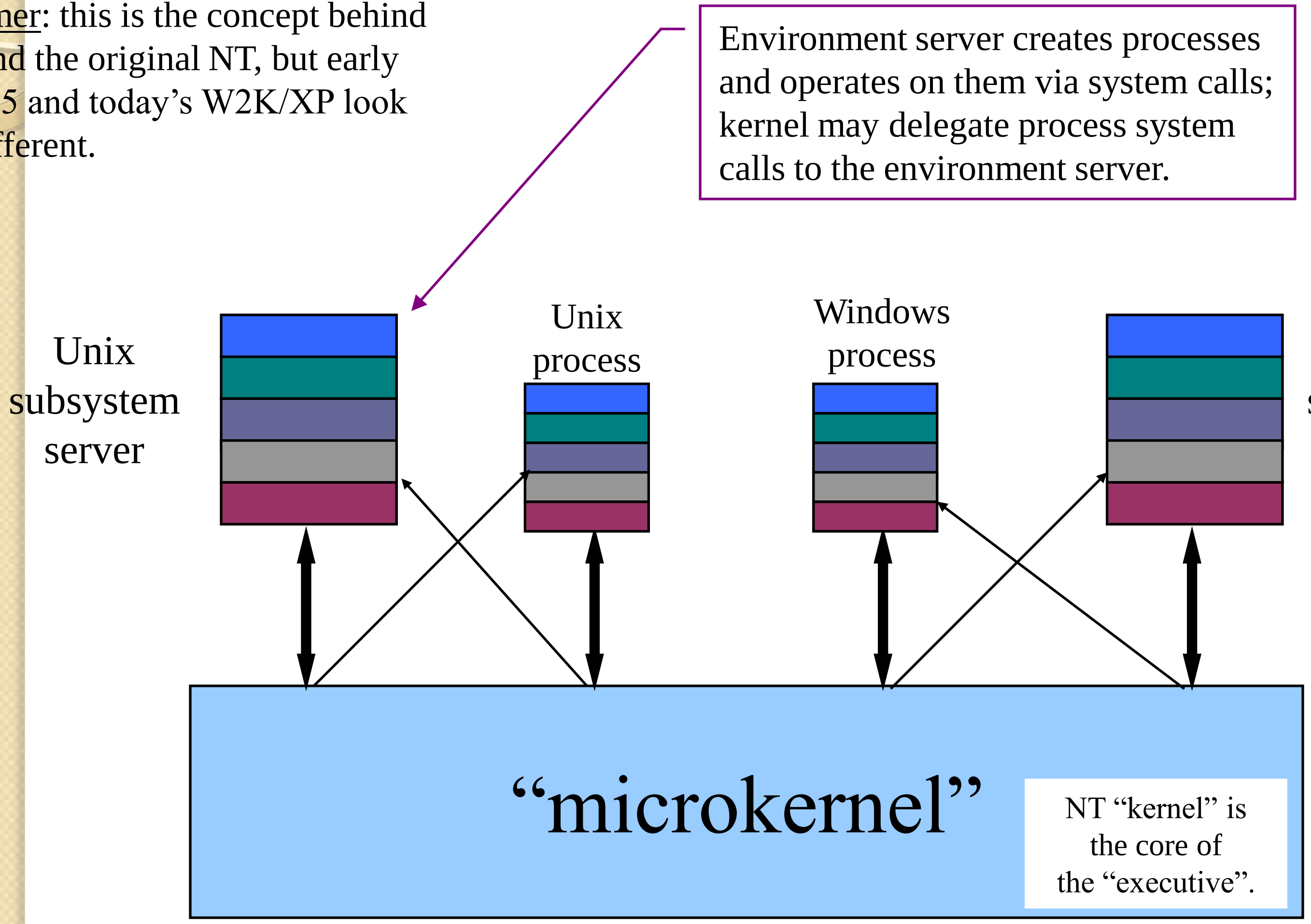
- Mach and NT are representative of systems that seek to provide richer, more general kernel interfaces than Unix.
  - decouple elements of process abstraction
    - virtual address space, memory segments, threads, resources, interprocess communication (IPC) endpoints
  - provide a fully general set of kernel primitives for combining these essential elements in arbitrary ways
    - powerful enough to implement Unix “as an application program”
    - *the kernel interface is not the programming interface*
  - rethink division of function between kernel and user space
    - Which features *must* be supported in the kernel?

The Microkernel Philosophy

- The microkernel philosophy evolved in the mid-1980s as a reaction to the increasing complexity of Unix kernels.
  - V system [Cheriton]: kernel is a “software backplane”
    - advent of LAN networks: V supports distributed systems, and mirrors their structure internally (decomposed)
  - Mach: designed as a modern, portable, reconfigurable Unix
    - improve portability/reliability by “minimizing” kernel code
    - support multiple “personalities”; isolate kernel from API changes
    - support multiprocessors via threads and extensible VM system
- Microkernels are widely viewed as having “failed” today, but some key ideas (and code) survive in modern systems.

A Fuzzy Look at Mach and NT

Disclaimer: this is the concept behind Mach and the original NT, but early Mach 2.5 and today's W2K/XP look quite different.

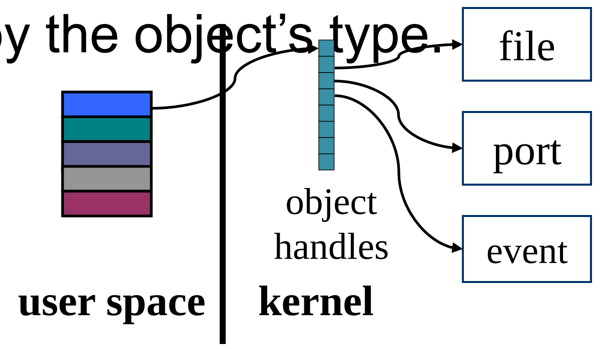


Environment server creates processes and operates on them via system calls; kernel may delegate process system calls to the environment server.
Unix subsystem server
Unix process
Windows process
Windows subsystem server
"microkernel"
NT "kernel" is the core of the "executive".

Microsoft NT Objects

- Most instances of NT kernel abstractions are "objects" named by protected *handles* held by processes.
  - Handles are obtained by create/open calls, subject to security policies that grant specific rights for each handle.
  - Any process with a handle for an object may operate on the object using operations (system calls).
    - Specific operations are defined by the object's type.

NT object handles are named, represented, and protected exactly like Unix file descriptors.



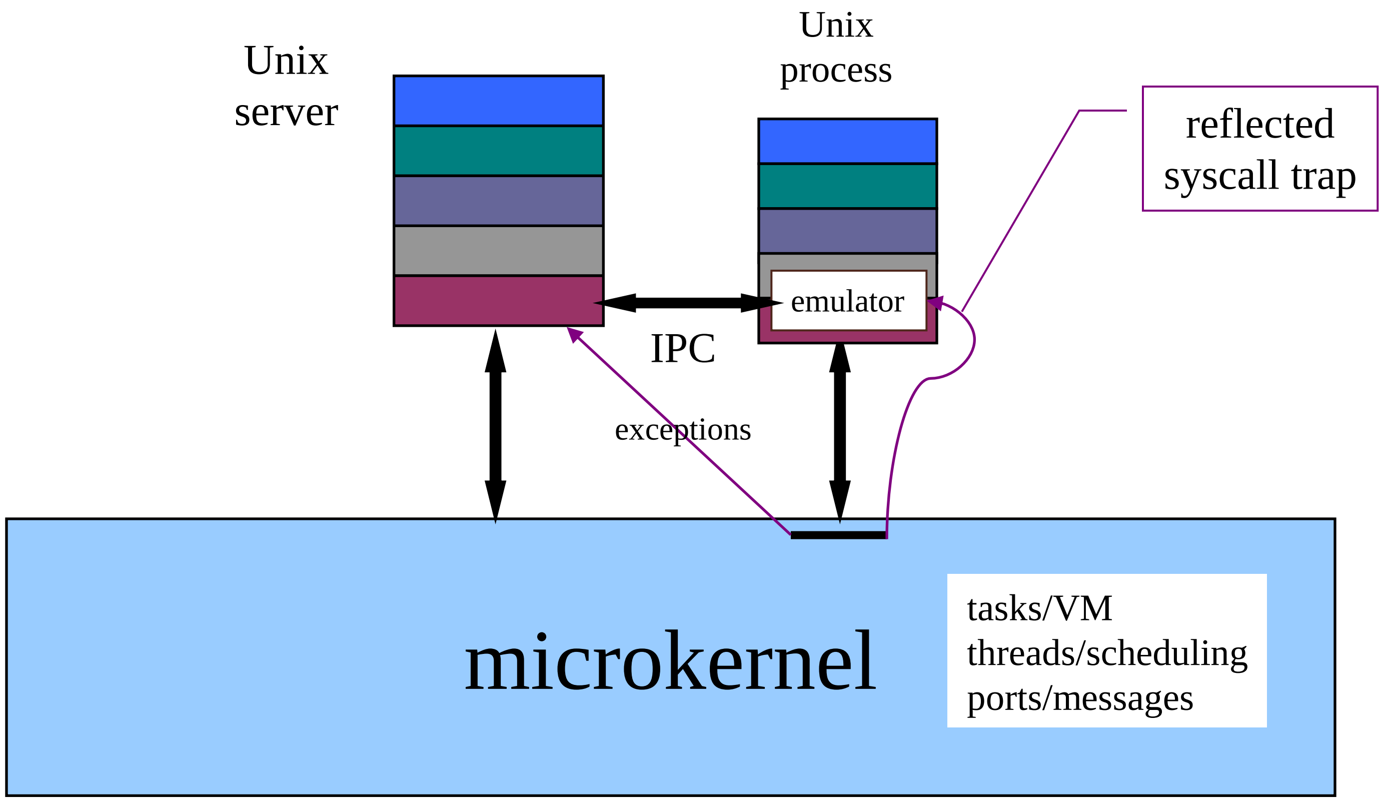
NT Processes

- 1. A raw NT process is just a virtual address space, a handle table, and an (initially empty) list of *threads*.
- 2. Processes are themselves objects named by handles, supporting specific operations.
  - create *threads*
  - map *sections* (VM regions)
- 3. *NtCreateProcess* returns an object handle for the process.
  - Creator may specify a separate (assignable) “parent” process.
  - Inherit VAS from designated parent, or initialize as empty.
  - Handles can be inherited; creator controls per-handle inheritance.

Mach Overview

- Mach is more general than NT in that objects named by handles can be served by user-mode servers.
  - All handles are references to message queues called *ports*.
  - Given an appropriate handle (*rights*) for the port, a thread can send or receive from a port: it's a *capability*.
    - Mach has a rich and complex set of primitives for sending/receiving on ports and transferring port rights.
  - Some ports are served by the kernel.
  - Ports can be served by user processes (*tasks*).
  - Everything is a port; all interactions are through ports.
  - Communication (IPC) performance is everything.

Mach


Unix server
Unix process
reflected syscall trap
emulator
IPC
exceptions
microkernel
tasks/VM
threads/scheduling
ports/messages

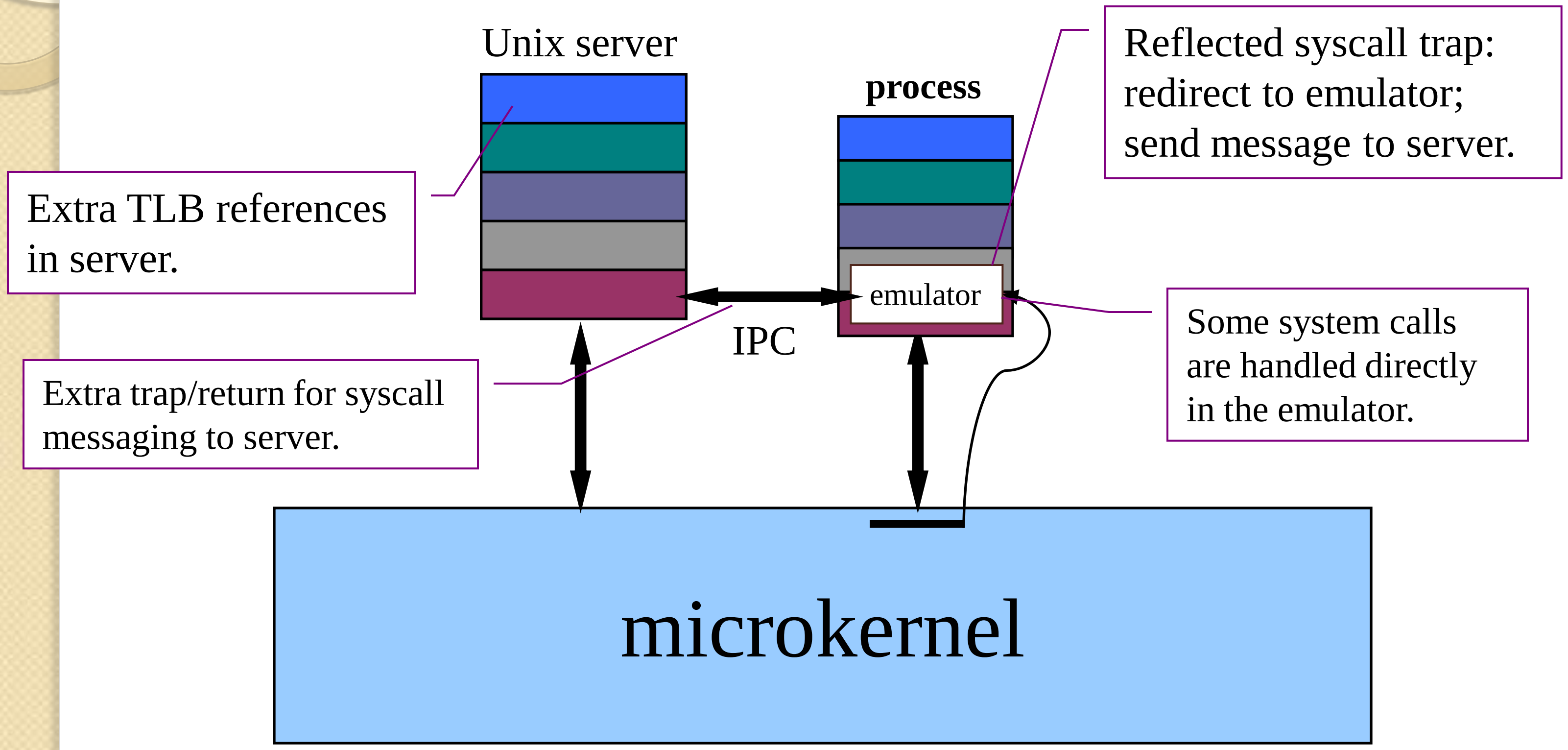
Evaluating OS Structures

- How do we evaluate OS structures and implementations?
  - Maintainability, extensibility, reliability, and elegance are difficult to quantify.
  - Systems research is a quantitative discipline.
    - “Performance is paramount.” [Cheriton]
- How can we identify and separate the effects caused by:
  - artifacts of the current state of hardware technology?
  - characteristics of the workload?
  - essential properties of structure or implementation?
- How can we draw conclusions of long-term significance from measurements of any particular system?

User/Kernel Division of Function

- 1. Many system calls don’t require access to mutable global data in the system (e.g., the Unix server).
  - examples: *getpid*, installing signal handlers
    - data items are constants or are used only within the emulator
- 2. The system can reduce the cost of these operations by executing them entirely within a library.
    - e.g., the Mach emulator, Unix *malloc* and *free*
- 3. A kernel or server primitive is needed only in cases involving resource allocation or protection.
    - thread libraries, user-level IPC [T. Anderson et. al]
    - logical conclusion: **Exokernel** “library operating systems”

Unix/Mach System Calls



Reflected syscall trap: redirect to emulator; send message to server.

Extra TLB references in server.

Some system calls are handled directly in the emulator.

Extra trap/return for syscall messaging to server.

Syscall trampoline offers binary compatibility at higher cost than a DLL (extra trap/return): either scheme can be used.

What to Know

1. The remaining slides in this batch serve to illustrate the effect of OS structures on performance. I did not cover them in class this year, but they may help to reinforce the discussions about OS structure and implementation.
2. Be sure that you understand the key ideas behind the three alternative structures we looked at: microkernels, library OS (Exokernel), and extensible kernels (SPIN). Be able to compare/contrast the goals, approaches, strengths, and weaknesses of each.
3. For the discussion of Exokernel and SPIN in class I referenced some of the high-level slides from Engler and Savage, which are available on the Engler and SPIN web sites linked through the readings page on the course web.

Issues and Questions for OS Performance

- 1. How is evaluating OS performance different from evaluating application performance?
- 2. (When) is OS performance truly important?
  - SPEC benchmarks: set me up and get out of my way.
    - Amdahl's Law says optimizing the OS won't improve performance here.
  - What workloads are OS-intensive? Do they matter?
- 3. How to characterize OS performance?
  - *macrobenchmarks* measure overall workload performance
  - analysis must decompose costs into essential elements
    - allows us to predict the effect of optimizing particular elements
  - *microbenchmarks* measure individual elements in isolation

Architectural Basis of OS Performance

- 1. OS demands on the architecture diverge from applications.
  - We don’t do much computation inside the OS kernel.
- 2. Basic OS functions have architecturally imposed costs.
  - OS spends relatively more time in “exceptional” operations.
  - mode switches (traps and returns), copy, save/restore registers, context switch, handle interrupts, page remap, page protect
  - These often depend on memory system behavior.
- 3. “Operating systems aren’t getting fast as fast as applications.”
  - [Ousterhout OSR90], and [Chen/Bershad SOSP93], [Rosenblum SOSP95]
  - Other things being equal, the *relative* importance of OS performance will increase with time.

OS Implications for Architecture

- Whose problem is this? [Levy et. al. ASPLOS91]
- ***Point***: architects must put some effort into optimizing operations that are critical to OS performance:
  - Design trap/exception mechanisms carefully: "exceptions are not exceptional". [also Levy/Thekkath 94]
  - Kernel code is more write-intensive ==> deepen the write buffer.
  - Design cache architectures to minimize user/kernel conflicts, etc.
- ***Counterpoint***: OS builders must design to minimize *inherent* architectural costs of OS choices.
  - Minimize expensive operations (e.g., traps and context switches).

Performance-Driven OS Design

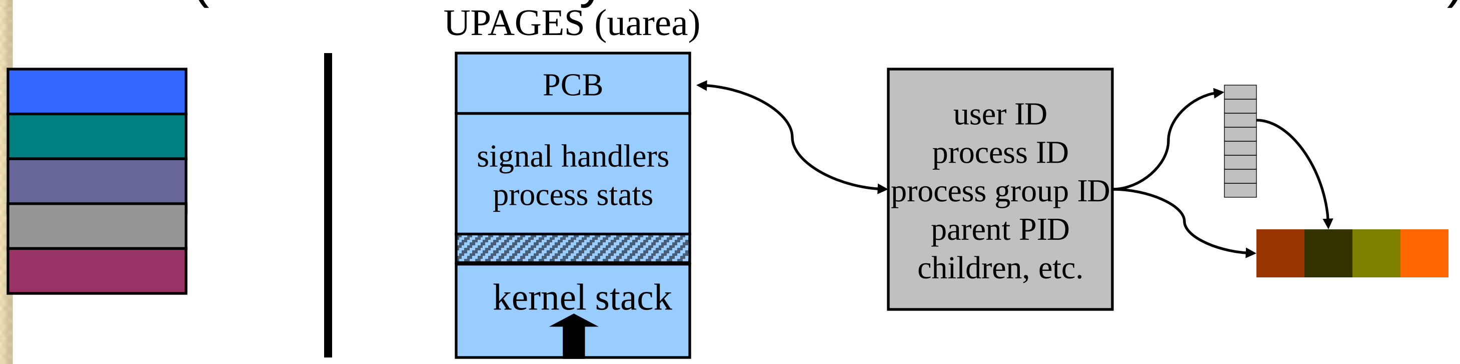
- 1. Design *implementations* to reduce costs of primitives to their architecturally imposed costs.
  - Identify basic architectural operations in a primitive, and streamline away any fat surrounding them.
  - “Deliver the hardware.”
- 2. Design system *structures* to minimize the architecturally imposed costs of the basic primitives.
  - If you can’t make it cheap, don’t use it as much.
- 3. Microbenchmarking is central to this process.
  - [Brown and Seltzer] is about microbenchmaking methodology.

Examples

- What are the architecturally defined costs of...
  - process creation?
    - *fork,* and *fork* by shell
    - *exec* of statically linked executable
    - *exec* with dynamic link
  - installing a signal handler?
  - reading from a file descriptor?
  - IPC?
- How can we determine these costs empirically?

Costs of Process Creation by *Fork*

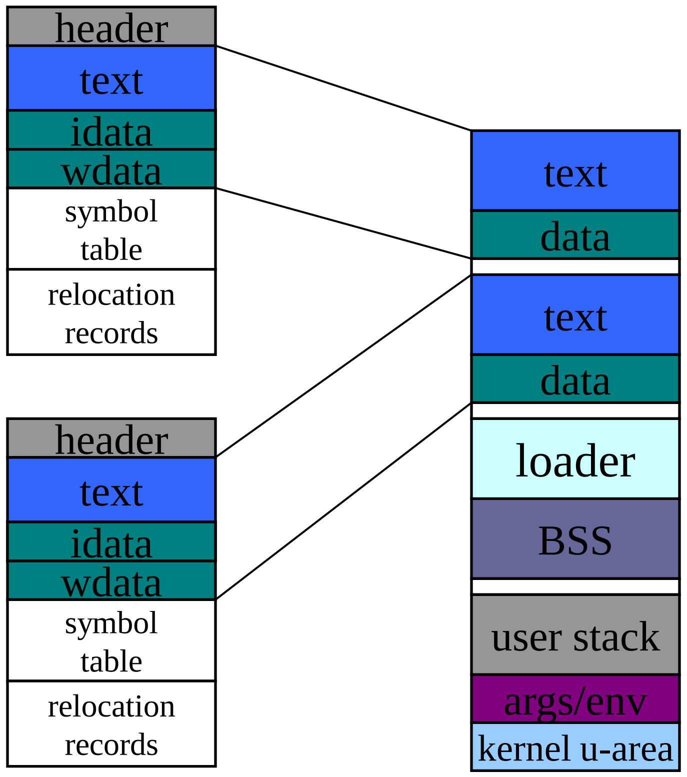
- one syscall trap, two return-from-trap, one process switch.
- allocate UPAGES, copy/zero-fill UPAGES through alias
- initialize page table, copy one stack page (at least)
    - fits in L2/L3? Depends on size of process?
- cache actions on UPAGES? for context switch?
    - (consider virtually indexed writeback caches and TLB)



Additional Costs of *Fork* on Mach or NT

- 1. Kernel call for task/process create with VM inheritance from the assigned parent.
  - inherit Mach emulator and emulated syscall table
- 2. Additional kernel call(s) to start a thread in the child.
  - task create/inherit does not clone any threads
- 3. Mach:
  - extra code/data/stack page references for emulator
    - clean/reset inherited emulator state in child (e.g., stacks, globals)
  - set up VM regions shared between server and emulator
  - set up IPC port to server in the child

Exec Revisited



Costs of Exec

- 1. Deallocate process pages and translation table.
  - invalidate v-cache (can delay until reallocated)
  - TLB invalidate
- 2. Allocate/zero-fill and *copyin/copyout* the arguments and base stack frame.
  - no v-cache push since no kernel aliasing is needed
- 3. Read executable file header and reset page translations.
  - map executable file sections on VAS segments
- 4. Handle any dynamic linking.
  - jump tables or load-time symbol resolution

Exec on a Microkernel

- 1. Each primitive action for *exec* is (typically) a kernel call.
- 2. *Copyin/copyout* of arguments/environment and base stack frame is problematic.
  - need a kernel call to read/write another address space
    - (but emulator handles argv/env internally)
    - how to do this copy efficiently?
  - similar problem for *read/write* system calls
    - Mach uses mapped files and handles read/write in emulator
  - L4 ukernel maps all user pages into the VM of the trusted servers.
- 3. *Solution*: add *exec* primitive to the "microkernel".
  - Most *exec* behavior is defined by the executable image header.

Lower Bounds on IPC

- IPC request/response requires a few primitive operations:
  - context switch
    - cache flush? TLB flush?
  - thread switch or stack switch
    - scheduling decision? Mach uses *handoff scheduling*.
    - LRPC [Bershad]: decouple threads from address spaces
      - threads cross address spaces for *cross*
    - adopted in NT *quick-LPC* and event
  - save/restore registers...how many?
  - copy arguments through the kernel?

Lesson: "optimize for the common case"; the common case for message passing is *local*.

ASSIGNMENT

- Q: Explain Unix emulation in MACH.