

# \*- mode: org \*-  
#+STARTUP: indent

\* <2011-11-05 Sat>, <2012-10-21 Sun>: L14: Operating System Organization

\* Operating System Organization

students have completed building an exokernel in lab3 and lab4  
user-space fork (copy-on-write)  
sophisticated VM handling in libos  
assumption in this lecture: one cannot change the kernel

\* Plan: OS organization

\*\* goals for a kernel interface  
\*\* monolithic  
\*\* microkernel  
\*\* exokernel  
\*\* little data, much opinion

\* Goals for kernel

\*\* Apps can use hardware resources  
\*\* Apps are easy convenient to write  
\*\* Apps are multiplexed (isolation and sharing)  
\*\* Few kernel crashes  
\*\* Kernel can evolve

\* the topic is overall o/s design

lots of ways to structure an OS -- how to decide?  
what is the right kernel API?  
looking for principles and approaches

\* what's the traditional approach? (Linux, xv6)

kernel API ~ POSIX

virtualize some resources: cpu and memory

simulate a dedicated cpu and memory system for each app  
why? it's a simple model for app programmers

abstract others: storage, network, IPC

layer a sharable abstraction over h/w (file system, IP/TCP)

\* example: virtualize the cpu

goal: simulate a dedicated cpu for each process

we want transparent CPU multiplexing

< process need not think about how it interacts w/ other processes  
o/s runs different processes in turn, via clock interrupt  
clock means process doesn't need to do anything special to switch  
also prevents hogging

how to achieve transparency?

o/s saves state, then restores

what does o/s save?

eight regs, EIP, seg regs, eflags, page table base ptr

where does o/s save it?

o/s keeps per-process table of saved states

the return from clock interrupt restores a \*different\* process's state

the point: process doesn't have to worry about multiplexing!

this is the traditional approach to virtualizing the CPU  
what does the exokernel/JOS do?

\* example: virtualize memory

idea: simulate a complete memory system for each process  
so process has complete freedom how it uses that memory  
doesn't have to worry about other processes  
so addresses  $0..2^{32}$  all work, but refer to private memory  
convenient: all programs can start at zero  
and memory looks contiguous, good for large arrays &c  
safe: can't even \*name\* another process's memory  
again: traditional but we'll soon see it's a very limiting approach  
really want apps to have more control than this style of VM implies

\* level of indirection allows o/s to play other tricks

\*\* demand paging:

process bigger than available physical memory?

"page-out" (write) pages to disk, mark PTEs invalid

if process tries to use one of those pages, MMU causes page fault

kern finds phys mem, page-in from disk, mark PTE valid

this works because apps use only a fraction of mem at a given time

need h/w valid flag, page faults, and re-startable instructions

\*\* copy-on-write:

\*\*\* avoid copy implied by fork() -- won't be needed if exec()

make parent and child share the physical memory pages

\*\*\* if either writes, do the copy then

so need per-page write-protect flag

\*\*\* both of above are transparent to application

still thinks it has simple dedicated memory from  $0..2^{32}$

\*\*\* paging h/w has turned out to be one of the most fruitful ideas in o/s

you have been using it a lot in labs

\*\* can we make it safe for apps to play these tricks?

\* traditional organization: monolithic o/s

h/w, kernel, user

kernel is a big program: process ctl, vm, fs, network

all of kernel runs w/ full hardware privilege (very convenient)

good: easy for sub-systems to cooperate (e.g. paging and file system)

bad: interactions => complex, bugs are easy, no isolation within o/s

extensibility: dynamically-loadable modules, wait for next kernel release

\*\* philosophy: convenience (for app or o/s programmer)

\*\*\* for any problem, either hide it from app, or add a new system call

(we need philosophy because there is not much science here)

may take a while for the new system call is added

\*\*\* very successful approach

\* alterate organization: microkernel

philosophy: IPC and user-space servers

for any problem, make a new server, talk to it w/ RPC

h/w, kernel, server processes, apps

servers: VM, FS, TCP/IP, Print, Display

split up kernel sub-systems into server processes

some servers have privileged access to some h/w (e.g. FS and disks)

apps talk to them via IPC / RPC

kernel's main job: fast IPC

good: simple/efficient kernel, sub-systems isolated, enforced better modularity

bad: cross-sub-system optimization harder, lots of IPCs may be slow

extensibility: can replace servers

in the end, lots of good individual ideas, but overall plan didn't catch on for desktops/servers

\* alternate organization: exokernel

philosophy: eliminate all abstractions

for any problem, expose h/w or info to app, let app do what it wants

h/w, kernel, environments, libOS, app

an exokernel would not provide address space, virtual cpu, file system, TCP

instead, give control to app:

phys pages, addr mappings, clock interrupts, disk i/o, net i/o

let app build nice address space if it wants, or not

should give aggressive apps much more flexibility

challenges:

how to multiplex cpu/mem/&c if you expose directly to apps?

how to get security/isolation despite apps having low-level control?

how to multiplex w/o understanding: disk (file system), incoming tcp pkts

\* exokernel example: memory

what are the resources? (phys pages, mappings)

what does an app need to ask the kernel to do?

pa = AllocPage()

DeallocPage(pa)

TLBwr(va, pa)

and these kernel->app upcalls:

PageFault(va)

PleaseReleaseAPage()

what does o/s need to do to make multiplexing work?

ensure app only creates mappings to phys pages it owns

track what env owns what phys pages

decide which app to ask to give up a phys page when system runs out

that app gets to decide which of its pages

\* simple example: shared memory

two processes want to share memory, for fast interaction

note traditional "virtual address space" doesn't allow for this

process a: pa = AllocPage()

put 0x5000 -> pa in private table

PageFault(0x5000) upcall -> TLBwr(0x5000, pa)

give pa to process b (need to tell exokernel...)

process b:

put 0x6000 -> pa in private table

...

\* example cool thing you could do w/ exokernel-style memory

databases like to keep a cache of disk pages in memory

problem on traditional o/s:

assume an o/s with demand-paging to/from disk

if DB caches some disk data, and o/s needs a phys page,

o/s may page-out a DB page holding a cached disk block

but that's a waste of time: if DB knew, it could release phys

page w/o writing, and later read it back from DB file (not paging area)

1. exokernel needs phys mem for some other app
2. exokernel sends DB a PleaseReleaseAPage() upcall
3. DB picks a clean page, calls DeallocPage(pa)
4. OR DB picks dirty page, writes to disk, then DeallocPage(pa)

\* exokernel example: cpu

what does it mean to expose cpu to app?

kernel tells app when it is taking away cpu

kernel tells app when it gives cpu to app

so if app is running and timer interrupt causes end of slice

cpu jumps from app into kernel

kernel jumps back into app at "please yield" upcall

app saves state (registers, EIP, &c)

app calls Yield()

when kernel decides to resume app

kernel jumps into app at "resume" upcall

app restores those saved registers and EIP

\* what cool things could an app do w/ exokernel-style cpu management?

suppose time slice ends in the middle of

```
acquire(lock);
```

```
...
```

```
release(lock);
```

you don't want the app to be holding the lock the whole time!

then maybe other apps can't make forward progress

so the "please yield" upcall can first complete the critical section

\* fast RPC with direct cpu management

how does traditional o/s let apps communicate?

pipes (or sockets)

picture: buffer in kernel, lots of copying and system calls

RPC probably takes 8 kernel/user crossings (read()s and write()s)

how does exokernel help?

Yield() can take a target process argument

almost a direct jump to an instruction in target process

kernel allows only entries at approved locations in target

kernel leaves regs alone, so can contain arguments

(in contrast to traditional restore of target's registers)

target app uses Yield() to return

so only 4 crossings

MIT OpenCourseWare  
<http://ocw.mit.edu>

6.828 Operating System Engineering  
Fall 2012

For information about citing these materials or our Terms of Use, visit: <http://ocw.mit.edu/terms>.